Lumberyard

Developer Guide

Version 1.9

Lumberyard: Developer Guide

Copyright © 2017 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

Lumberyard for Programmers 1
AI 3
AI System Concepts 3
AI System Overview 4
Pathfinding Costs 7
Sensory Models 10
Flight 15
AI C++ Class Hierarchy 16
AI System Concept Examples 16
AI Bubbles System 18
Message Display Types 18
Specifying Notification Display Types 19
AI Tactical Point System 20
Tactical Point System Overview 21
TPS Query Execution Flow 22
TPS Querying with C++ 22
TPS Querying with Lua 24
TPS Query Language Reference 25
Point Generation and Evaluation 28
Integration with the Modular Behavior Tree System 30
Future Plans and Possibilities 30
Navigation Q & A 31
Big Triangles and Small Links Between Them 31
Path Following 32
Auto-Disabling 32
Path Following 32
Goalop "Followpath" 32
COPTrace::ExecuteTrace and COPTrace::Execute 33
COPTrace::Execute2D 33
Movement System 33
Using the Movement System 34
Potential Improvements 34
Auto-Disable 35
Global auto-disable 35
Per-AI auto-disable 35
AI Scripting 35
Communication System 36
Factions 42
Modular Behavior Tree 43
Refpoints 86
Signals 88
Animation 98
Animation Overview 98
Linear Animations 98
Interactive Animations 99
Scripted Animations 100
Animation Events 101
Marking Up Animations with Events 101
Receiving Animation Events in the Game Code 101
Limb IK Technical 101
Setting Up 101
Using LimbIK from Code 101
Animation Streaming 102
Animation Data 102

Animation Header Data 102
Animation Controller Data 102
Animation Debugging 104
Layered Transition Queue Debugging 105
CommandBuffer Debugging 109
Warning Level 109
Fall and Play 109
Time in the Animation System 110
Segmentation 111
Playback Speed 112
Segmented Parametric Animation 112
Animation with Only One Key 113
Direction of Time 113
Time within Controllers 113
Asset Builder API 115
Builder Modules 115
Creating a Builder Module 116
Main Entry Point 116
Lifecycle Component 117
Creating a Builder 118
Message Loggging 125
AZ Code Generator 126
Workflow Summary 127
Waf 127
Clang 128
Intermediate JSON Data 128
AZ Code Generator and Python 129
Template Drivers and Template Rendering 129
Generated Files 129
AZ Code Generator Integration with Waf 131
Basic Integration 131
Advanced Integration 132
Input Files 132
Template Drivers 133
Command Line Parameters 133
Waf Specific Options 133
AZ Code Generator Parameters 133
Waf Parameters 134
Clang Compilation Parameters 134
Intermediate Data 134
Front End 134
AZ Code Generator Parameter List 134
Code Generation Templates 137
Simple Example 137
Complex Example 138
Template Data 139
Template Drivers 139
Specifying Drivers in Waf 140
Creating a Template Driver in Python 140
Minimal Template Driver 142
Rendering Templates 142
Configuring Automatic Build Injection 142
Preprocessing Intermediate Data 143
Custom Code Generator Annotations 143
Reference Annotations 144
Helper Macros 144
Example Annotations 144

Waf Debugging with AZ Code Generator 148
Prerequisites 148
Identifying and Configuring Debug Output 148
Setting Up PyCharm for Debugging Waf 149
Template Driver Debugging 154
Debugging the AZ Code Generator Utility 155
Prerequisites 155
Debugging the AZ Code Generator Utility from the Waf build 156
Setting Visual Studio Debug Arguments 156
Setting Xcode Debug Arguments 156
Intermediate JSON Data Format 157
AZ Modules (Preview) 159
Comparing AZ Modules to Legacy Modules 159
A Self-Aware Method of Initialization 160
Relationship with the AZ Framework 161
Smarter Singletons 161
Current Lumberyard AZ Modules 161
LmbrCentral 161
LmbrCentralEditor 161
Parts of an AZ Module, Explained 161
The Module Class 162
The EBus 163
The System Component Class 164
Calling the Module from External Code 166
System Components 166
Smart Initialization Order 166
Easily Configurable Components 167
Writing System Components 167
Required System Components 168
Gems and AZ Modules 168
Structure of a Gem 168
Waf Integration 169
Gems Built as AZ Modules 169
Creating an AZ Module That Is Not a Gem 169
A. Start with a Gem 169
B. Modify the AZ Module Declaration 170
C. Remove CryEngine References (Optional) 170
D. Modify the Wscript and Waf Spec Files 171
E. Configure Your Project to Load the New Module 172
F. Add the Module's Public Interfaces to Your Project's Include Paths 172
Configuring System Entities 173
Application Descriptor Files 176
The AZ Bootstrapping Process 177
Cloud Canvas 178
Features 178
Example Uses 178
Tools 179
Knowledge Prerequisites 179
Overview 180
Prerequisites 180
AWS, Cloud Canvas, and Lumberyard 180
Amazon Web Services Supported by Cloud Canvas 181
Understanding Cloud Canvas Resource Manager 182
Cloud Gems Overview 187
Pricing 187
Tutorial: Getting Started with Cloud Canvas 188
Game Play Design and Engineering Guide 197

Tools in Lumberyard Editor 197
Managing Cloud Canvas Profiles 198
Using Resource Manager in Game Design 200
Cloud Gems 200
Flow Graph Node Reference 254
Software Engineering Guide 276
Resource Manager in Depth 276
Cloud Gem Framework 305
Administering Cloud Canvas 381
Setting Up a Project to Use Resource Manager 381
Working with Deployments 385
Understanding the Resource Manager Security System 393
Using the Command Line 408
Syntax 408
Configuration 408
Common Arguments 409
Commands 409
Command Reorganization 424
Component Entity System 428
Programmer's Guide to Entities and Components 428
Creating a Component 429
Registering Your Component 431
Reflecting a Component for Serialization and Editing 431
Defining and Using Component Services 436
Editor Components 437
Creating System Components 440
Components and EBuses 442
Tick Bus and Components 445
Exposing Custom Components to Track View for Animation 446
Components and EBuses: Best Practices 451
Behavior Context 453
Reflection API 454
Example 456
Gameplay Bus 458
GameplayNotificationId 458
GameplayNotifications 459
GameplayEventHandlerNode 459
GameplayEventSenderNode 460
Script Examples 460
Slices and Dynamic Slices 461
Anatomy of a Slice 462
Working with Dynamic Slices 462
Instantiating Dynamic Slices 463
Controller Devices and Game Input 465
Input in Amazon Lumberyard 465
AZ Framework Input 466
Migration Path 469
Legacy Input 469
Action Maps 469
CryInput 470
Setting Up Controls and Action Maps 472
CryCommon 478
CryExtension 478
Composites 479
Shared and raw interface pointers 479
GUIDs 479
ICryUnknown 480

ICryFactory 481
ICryFactoryRegistry 481
Additional Extensions 482
Glue Code Macros 484
CryExtension Samples 489
Using Extensions 490
Implementing Extensions Using the Framework 492
CryString 501
How to Use Strings as Key Values for STL Containers 501
Further Usage Tips 502
ICrySizer 502
How to use the ICrySizer interface 502
Serialization Library 502
Tutorial 503
Use Cases 505
Demo and Video Capture 514
Capturing Video and Audio 514
Preparation 514
Video Settings 514
Starting and Ending the Video Recording 516
Audio Settings 516
Configuration Files 517
Recording Time Demos 518
Overview 518
Recording Controls 518
Related Console Variables 518
Entity System 520
Entity Property Prefixes 520
Creating a New Entity Class 521
Entity Pool System 523
Editor Usage 523
Static versus Dynamic Entities 524
Entity Pool Definitions 524
Entity Pool Creation 526
Creating and Destroying Static Entities with Pools 526
Creating and Destroying Dynamic Entities with Pools 529
Serialization 529
Listener/Event Registration 530
Debugging Utilities 531
Entity ID Explained 532
Adding Usable Support on an Entity 532
Overview 532
Preparing the Script 533
Implementing IsUsable 533
Implementing OnUsed 533
Entity Scripting 533
Structure of a Script Entity 533
Using Entity State 537
Using Entity Slots 538
Linking Entities 539
Exposing an Entity to the Network 540
Event Bus (EBus) 543
Bus Configurations 543
Single Handler 543
Many Handlers 544
EBus with Addresses and a Single Handler 545
EBus with Addresses and Many Handlers 547

Synchronous vs. Asynchronous 549
Additional Features 549
Usage and Examples 550
Declaring an EBus 550
EBus Configuration Options 551
Implementing a Handler 552
Sending Messages to an EBus 553
Retrieving Return Values 553
Return Values from Multiple Handlers 553
Asynchronous/Queued Buses 554
File Access 555
CryPak File Archives 555
Features 555
Unicode and Absolute Path Handling 555
Layering 555
Slashes 556
Special Folder Handling 556
Internals 556
Creating a pak file using 7-Zip 556
Dealing with Large Pak Files 556
Accessing Files with CryPak 557
Tracking File Access 563
CVars 563
Where invalid access is defined 563
Graphics and Rendering 565
Render Nodes 565
Creating a New Render Node 566
TrueType Font Rendering 569
Supported Features 569
Useful Console Commands 570
Generating Stars DAT File 570
File Format 570
Anti-Aliasing and Supersampling 571
Controlling Anti-Aliasing 571
Controlling Supersampling 573
Lua Scripting 574
Learning Lua 574
Learning Lua in Lumberyard 574
Adding Lua Scripts to Component Entities 575
Basic Structure of a Component Entity Lua Script 576
Built-in Types and Methods 577
Properties 577
Supported Types 578
Attributes 580
Network Binding 580
Properties 580
RPCs 581
Communicating with Components 581
Order of Component Activation 581
Registering with a Component to Receive Notifications 582
Noncomponent Notifications 582
Sending Events to a Component 583
Communicating with Components Attached to Other Entities 584
Using AZStd::vector and AZStd::array 584
Using AZStd::any 585
Debugging Scripts 585
Logging to the Console 585

Using an Assert to Detect Potential Issues ... 585
Communicating Errors ... 586
Displaying a Warning When User Attention Is Required ... 586
The Lua Environment (Advanced) ... 586
Adding Other VMs ... 587
Reusing Code ... 587
Lua Editor ... 587
Tutorial: Using Lua Editor for Debugging with Lumberyard Editor ... 587
Maintaining Separate Search Results ... 591
Editing ... 592
Perforce Integration ... 593
Component Entity Lua API Reference ... 593
BehaviorTreeComponentRequestBus ... 593
NavigationComponentRequestBus ... 595
NavigationComponentNotificationBus ... 595
NavigationSystemRequestBus ... 597
AttachmentComponentRequestBus ... 597
AttachmentComponentNotificationBus ... 598
CharacterAnimationRequestBus ... 599
MannequinRequestsBus ... 600
SimpleAnimationComponentRequestBus ... 604
SimpleAnimationComponentNotificationBus ... 607
AudioEnvironmentComponentRequestBus ... 607
AudioListenerComponentRequestBus ... 608
AudioRtpcComponentRequestBus ... 609
AudioSwitchComponentRequestBus ... 609
AudioTriggerComponentRequestBus ... 610
AudioTriggerComponentNotificationBus ... 611
FloatGameplayNotificationBus (`AZ::GameplayNotificationBus<float>`) ... 612
Vector3GameplayNotificationBus (`AZ::GameplayNotificationBus<AZ::Vector3>`) ... 612
StringGameplayNotificationBus (`AZ::GameplayNotificationBus<const AZStd::string>`) ... 612
EntityIdGameplayNotificationBus (AZ::GameplayNotificationBus<AZ::EntityId>) ... 613
CryCharacterPhysicsRequestBus ... 613
ConstraintComponentRequestBus ... 613
ConstraintComponentNotificationBus ... 614
PhysicsComponentRequestBus ... 615
PhysicsComponentNotificationBus ... 618
PhysicsSystemRequestBus ... 619
RagdollPhysicsRequestBus ... 620
DecalComponentRequestBus ... 621
LensFlareComponentRequestBus ... 621
LensFlareComponentNotificationBus ... 622
LightComponentRequestBus ... 622
LightComponentNotificationBus ... 623
ParticleComponentRequestBus ... 624
SimpleStateComponentRequestBus ... 625
SimpleStateComponentNotificationBus ... 626
SpawnerComponentRequestBus ... 627
SpawnerComponentNotificationBus ... 628
TagComponentRequestBus ... 629
TagGlobalRequestBus ... 630
TagGlobalNotificationBus ... 631
TagComponentNotificationsBus ... 631
TriggerAreaRequestsBus ... 631
TriggerAreaNotificationBus ... 632
TriggerAreaEntityNotificationBus ... 633
BoxShapeComponentRequestsBus ... 634

CapsuleShapeComponentRequestsBus 634
CylinderShapeComponentRequestsBus 635
ShapeComponentRequestsBus 636
ShapeComponentNotificationsBus 637
SphereShapeComponentRequestsBus 638
EntityBus 638
TickBus 639
TickRequestBus 639
TransformNotificationBus 640
GameEntityContextRequestBus 640
RandomManagerBus 642
CameraRequestBus 643
HttpClientComponentNotificationBus 646
HttpClientComponentRequestBus 647
HMDDeviceRequestBus 647
ControllerRequestBus 649
VideoPlaybackRequestBus 650
VideoPlaybackNotificationBus 651
VR Lua Functions 652
Global Functions 652
HMDDeviceRequestBus 652
ControllerRequestBus 653
struct HMDDeviceInfo 653
struct TrackingState 653
struct PoseState 654
struct DynamicsState 654
enum HMDStatus 654
UI Lua Reference 654
LyShineLua.ShowMouseCursor 655
UIButtonComponent 655
UICanvasComponent 656
UICanvasManager 662
UICheckboxComponent 662
UIDynamicLayoutComponent 664
UIDynamicScrollBoxComponent 665
UIElementComponent 666
UIFaderComponent 668
UIImageComponent 669
UIInteractableComponent 671
UILayoutColumnComponent 678
UILayoutGridComponent 679
UILayoutRowComponent 682
UIMaskComponent 683
UIScrollBarComponent 684
UIScrollBoxComponent 687
UISliderComponent 693
UITextComponent 696
UITextInputComponent 700
UITooltipComponent 704
UITooltipDisplayComponent 704
UITransform2dComponent 707
UI World Components 710
Lua Scripting - Legacy 713
Working with Lua Scripting 713
Running Scripts 714
Reloading Scripts During Runtime 714
Recommended Reading 714

Using the Lua Remote Debugger ... 715
Using the Lua XML Loader ... 716
Loading Canvases in Lua ... 719
Entity System Script Callbacks ... 720
Default State Functions ... 720
Script State Functions ... 721
Game Rules Script Callbacks ... 722
Common Lua Globals and Functions ... 724
Globals ... 725
AIReload() ... 725
AIDebugToggle() ... 725
ShowTime() ... 726
count() ... 726
new() ... 726
merge() ... 726
mergef() ... 726
Vec2Str() ... 727
LogError() ... 727
LogWarning() ... 727
Log() ... 727
dump() ... 727
EmptyString() ... 728
NumberToBool() ... 728
EntityName() ... 728
EntityNamed() ... 728
SafeTableGet() ... 728
EntityUtils Lua Functions ... 729
DumpEntities() ... 729
CompareEntitiesByName() ... 729
CompareEntitiesByDistanceFromPoint() ... 729
BroadcastEvent() ... 730
MakeDerivedEntity() ... 730
MakeDerivedEntityOverride() ... 730
MakeUsable() ... 731
MakePickable() ... 731
MakeSpawnable() ... 731
EntityCommon.PhysicalizeRigid() ... 731
Lua Vector and Math Functions ... 732
Global Vectors ... 732
Constants ... 733
IsNullVector() ... 733
IsNotNullVector() ... 733
LengthSqVector() ... 733
LengthVector() ... 733
DistanceSqVectors() ... 734
DistanceSqVectors2d() ... 734
DistanceVectors() ... 734
dotproduct3d() ... 734
dotproduct2d() ... 734
LogVec() ... 735
ZeroVector() ... 735
CopyVector() ... 735
SumVectors() ... 735
NegVector() ... 736
SubVectors() ... 736
FastSumVectors() ... 736
DifferenceVectors() ... 736

FastDifferenceVectors() ... 737
ProductVectors() ... 737
FastProductVectors() ... 737
ScaleVector() ... 737
ScaleVectorInPlace(a,b) ... 737
ScaleVectorInPlace(dest,a,b) ... 738
NormalizeVector() ... 738
VecRotate90_Z() ... 738
VecRotateMinus90_Z() ... 738
crossproduct3d() ... 738
RotateVectorAroundR() ... 739
ProjectVector() ... 739
DistanceLineAndPoint() ... 739
LerpColors() ... 739
Lerp() ... 740
__max() ... 740
__min() ... 740
clamp() ... 740
Interpolate() ... 741
sgn() ... 741
sgnnz() ... 741
sqr() ... 741
randomF() ... 741
iff() ... 742
Physics Lua Functions ... 742
Physics.RegisterExplosionShape() ... 742
Physics.RegisterExplosionCrack() ... 743
Lua ScriptBind Reference ... 743
ScriptBind Engine Functions ... 743
ScriptBind Action Functions ... 899
ScriptBind_Boids ... 945
Integrating Lua and C++ ... 949
Accessing Script Tables ... 949
Exposing C++ Functions and Values ... 949
Networking System ... 951
Tutorial: Getting Started with Multiplayer ... 951
Prerequisites ... 952
Step 1: Creating a Level and Adding a Sphere and a Box ... 952
Step 2: Binding Sphere Transform Components to the Network ... 956
Step 3: Connecting a Client to the Server ... 957
Related Tasks and Tutorials ... 959
Configuring the Multiplayer Sample for Amazon GameLift ... 959
Overview ... 963
NetBinding ... 964
GridMate ... 965
Other GridMate Features ... 966
CryNetwork Backward Compatibility (Deprecated) ... 966
Networking Architecture ... 966
Carrier ... 969
Marshalling ... 972
Sessions ... 976
Replicas ... 984
Replica Manager ... 994
Using Lumberyard Networking ... 996
Synchronizing Game State Using Components ... 997
Synchronizing Game State Using Scripts ... 1002
Using Encyrption ... 1002

HttpRequestor Gem 1007
Controlling Bandwidth Usage 1010
Setting up a Lobby 1012
Creating Dedicated Servers 1012
Using Amazon GameLift 1012
Useful Console Commands 1013
CryNetwork Backward Compatibility 1014
RMI Functions 1014
Network Serialization and Aspects 1017
Physics 1018
Physics Scripting Guide 1018
Threading 1018
Physics Scripting Tips and Tricks 1020
Scripting with Physics Example 1020
Geometries 1021
Geometry Management Functions 1021
Physical Entities 1022
Creating and managing entities 1023
Functions for Entity Structures 1025
Common Functions 1025
Living Entity-Specific Functions 1027
Particle Entity-Specific Functions 1029
Articulated Entity-Specific Functions 1029
Rope Entity-Specific Functions 1031
Soft Entity-Specific Functions 1031
Collision Classes 1032
Setup 1032
Code 1033
Types 1034
Filtering the collision 1034
Interface 1034
Functions for World Entities 1035
Advancing the Physical World Time State 1035
Returning Entities with Overlapping Bounding Boxes 1035
Casting Rays in an Environment 1036
Creating Explosions 1037
Profiler 1038
Profiler Tutorial 1038
Registering Your Application 1039
Launching Profiler 1039
Capturing Data 1039
Inspecting Data 1041
Playing Back Data 1043
Exporting Data 1048
Creating and Using Annotations 1048
Using Annotations 1049
Creating Annotations 1050
Viewing Annotations in Trace Messages Profiler 1051
Using Profiler for Networking 1051
Prerequisites 1052
Carrier Profiler 1052
Replica Activity Profiler 1053
Using the Profiler for CPU Usage 1060
Understanding the Tree View 1061
Controlling the Display 1061
Using Profiler for VRAM 1063
Notes 1064

Understanding the Captured Data 1064
Inspecting the Data 1065
Using GridHub 1066
Registering an Application in GridHub 1066
Viewing and Configuring GridHub 1067
Troubleshooting GridHub 1068
System 1070
Memory Handling 1070
Hardware Memory Limitations 1070
Choosing a Platform to Target 1070
Budgets 1071
Allocation Strategy with Multiple Modules and Threads 1071
Caching Computational Data 1071
Compression 1071
Disk Size 1071
Total Size 1071
Address Space 1072
Bandwidth 1072
Latency 1072
Alignment 1072
Virtual Memory 1073
Streaming 1073
Streaming System 1073
Low-level Streaming System 1073
Streaming and Levelcache Pak Files 1079
Single Thread IO Access and Invalid File Access 1080
High Level Streaming Engine Usage 1080
Text Localization and Unicode Support 1082
Terminology 1082
What encoding to use? 1083
How does this affect me when writing code? 1084
How does this affect me when dealing with text assets? 1085
Utilities provided in CryCommon 1085
Further reading 1086
CryLog 1086
CryLog Logging Functionality 1086
Verbosity Level and Coloring 1086
Log Files 1087
Console Variables 1087
CryConsole 1088
Color coding 1088
Dumping all console commands and variables 1088
Console Variables 1088
Adding New Console Commands 1089
Console Variable Groups 1089
Deferred execution of command line console commands 1091
CVar Tutorial 1092
Lumberyard Blog, Forums, and Feedback 1094

Lumberyard for Programmers

The Lumberyard Developer Guide is intended for programmers or anyone working directly with the Lumberyard code.

This guide includes the following sections:

- **AI (p. 3)**

  Describes a variety of AI features that process navigation and individual and group behaviors, and describes convenient tools such as a Visual AI debugger, behavior tree visual editor, and a visual flow graph editor.
- **Animation (p. 98)**

  Contains tools to create both linear (video) and interactive animation. Interactive animation conveys AI and avatar (player) behavior, with sequences dependent on player choices in gameplay.
- **Asset Builder API (p. 115)**

  Use the asset builder API to develop a custom asset builder that can process any number of asset types, generate outputs, and return the results to the asset processor for further processing. A custom builder can be especially useful in a large project that has custom asset types.
- **AZ Code Generator (p. 126)**

  AZ Code Generator is a command line utility that generates source code (or any data or text) from specially tagged source code. You can use it when the structure of the intended code is known in advance.
- **AZ Modules (Preview) (p. 159)**

  AZ modules are new, Amazon-created code libraries that plug into Lumberyard games and tools. These AZ modules implement specific initialization functions. When a Lumberyard application starts, it loads each AZ module and calls the corresponding
- **Cloud Canvas (p. 178)**

  Cloud Canvas is Lumberyard's technology for connecting your game to Amazon Web Services. With Cloud Canvas, you can use AWS to implement cloud-hosted features and create asynchronous multiplayer games. Using AWS means you no longer have to acquire, configure, or operate host servers to implement connected gameplay.
- **Component Entity System (p. 428)**

  The component entity system is a new Amazon-created way of creating components that is superior to (and that will eventually replace) the legacy Entity System (p. 520).

- **Controller Devices and Game Input (p. 465)**

  Describes Lumberyard's support for input devices such as keyboards, mice, and joysticks, and shows how to set up controls and action maps.
- **CryCommon (p. 478)**

  Describes game engine interfaces, including CryExtension, which you can use to refactor Lumberyard features into extensions for ease of use; CryString, which is a custom reference-counted string class; and a serialization library, which separates user serialization code from actual storage format and makes it easy to change formats.
- **Demo and Video Capture (p. 514)**

  Describes how to use Lumberyard Editor or the Lumberyard standalone Launcher to record benchmarking videos and capture audio.
- **Entity System (p. 520)**

  Describes the creation and management of entities, which are objects placed inside a level that players can interact with. This section contains topics such as creating a new entity class, entity pools, and entity scripting.
- **Event Bus (EBus) (p. 543)**

  Event buses are Lumberyard's general purpose system for dispatching messages. EBuses minimize hard dependencies between systems, are event-driven (which eliminates polling), handle concurrency well, and enable predictability by providing support for the ordering of handlers on a given bus.
- **File Access (p. 555)**

  Describes how to compress game content files and how to track invalid file reads that can potentially stall the performance of a game.
- **Graphics and Rendering (p. 565)**

  Lumberyard's shading core uses the same physical parameters that are used in high end film rendering pipelines. This section covers render nodes, true type font rendering, and the star data used in sky rendering. It also describes how to control anti-aliasing so that you can produce graphics from very sharp images to softer blurred images.
- **Lua Scripting (p. 574)**

  Lua is Lumberyard's scripting language. This section contains a guide to writing Lua scripts for Lumberyard's component entity system, a tutorial on Lumberyard's Lua Editor, and Lua API references.
- **Networking System (p. 951)**

  Describes GridMate, Lumberyard's networking subsystem, and contains topics on multiplayer setup, the session service, controlling bandwidth usage, and synchronizing game state using the GridMate replica framework.
- **Physics (p. 1018)**

  Describes the Lumberyard physics system and how to interact with the physics engine. This section shows you how to create a physical world object, fill the world with geometries and physical entities, and control the entities with the functions described.
- **Profiler (p. 1038)**

  Profiler is a Lumberyard tool that can capture, save, and analyze network, CPU, and VRAM usage statistics. You can used the saved data to analyze network usage frame by frame, fix problems in the use of network bandwidth, and optimize the performance of your game.
- **System (p. 1070)**

  Contains topics on memory handling, streaming, localization, logging, and console tools.

AI

This section describes the AI system. It includes a general overview of key concepts, describes system components, and provides an AI scripting manual.

This section includes the following topics:

- AI System Concepts (p. 3)
- AI Bubbles System (p. 18)
- AI Tactical Point System (p. 20)
- Navigation Q & A (p. 31)
- Path Following (p. 32)
- Movement System (p. 33)
- Auto-Disable (p. 35)
- AI Scripting (p. 35)

AI System Concepts

Key features of the AI system include the following:

Navigation

- Navigation with little or no assistance from the designers
- Multi-layer navigation (flying, swimming, zero-gravity) or simple 2D navigation
- Smart objects for special navigation and interactions

Individual AI

- Easy-to-use static and dynamic covers (such as behind movable vehicles)
- Dynamic tactical points (such as cover points, ambush points, patrol waypoints)
- Behavior trees, to select behaviors based on values of Boolean variables
- Customizable perception (such as vision, sound, memory, sixth sense)

Group and Global AI

- Group behavior trees, to define group tactics

- Formations, to move AI characters in some orderly fashion
- Factions (such as friends, neutrals, enemies)
- Visual flow graphs of game logic, with macro-nodes for reused sub-flow graphs

MMO-ready

- Support for streaming big maps

User-friendly

- Visual AI debugger to log signals, behavior changes, goal changes, user comments
- Behavior tree visual editor
- Visual flow graph editor and debugger (with visual flow propagation and break points)

This section includes the following topics:

- AI System Overview (p. 4)
- Pathfinding Costs (p. 7)
- Sensory Models (p. 10)
- Flight (p. 15)
- AI C++ Class Hierarchy (p. 16)
- AI System Concept Examples (p. 16)

AI System Overview

This section outlines basic concepts related to the AI system.

Navigation

- Default navigation system
  - Triangulation
    - 2D terrain-based navigation
    - Uses cylindrical objects (such as trees) and forbidden areas
  - Navigation modifiers
    - Human waypoints – Need to be place manually but connections can be generated automatically
    - Flight – Information about navigable volumes for flying entities
    - Volume – General volume navigation, such as for oceans
- Multi-layer navigation system
- Smart object system: allows AI agents to move in special ways
- AI territories & waves
  - Control number of active AI agents (through flow graph logic)
  - Activate, deactivate, and spawn all AI agents assigned to a territory using a single FG node
  - AI waves can be attached to AI territories and allow independent AI activations
  - AI waves automatically handle entity pool issues for assigned AI agents, such as loading/unloading

In general, a search is time-sliced to use 0.5 ms per AI frame (configured using the console variable `ai_PathfinderUpdateTime`). Options for pathfinding techniques include high priority, straight, and partial. Updates for human waypoints are heavy but time-sliced. The navigation graph is optimized but

needs memory. Navigation data is generated offline in Editor. With multi-layer navigation, the navigation mesh is regenerated when the designer modifies the map.

Decision Making

- Behavior selection system – Uses behavior trees to select AI behaviors
- Cover system – Provides AI agents with static and dynamic covers
- Smart object system – Allows AI agents to interact with their environment
- Interest system – Allows AI agents to perform intelligent actions when not alerted

Tactical

- Tactical point system (TPS) – Allows AI agents to ask intelligent questions about their environment (such as where to hide or where to attack)
- Faction system – Determines levels of hostility between AI agents
- Group coordination system – Uses coordination selection trees to select group behaviors
- Formation system – Allows AI agents to move in formations
- Cluster detector – detects clusters of points in space and subdivides them into separate groupings that satisfy specific properties (using a modified K-mean algorithm); used with AISquadManager to group different AI agents into dynamic squads

World-Interfacing

- Signals – To trigger events and/or change behaviors
- Perception system
  - Perception handler (legacy, usually per game)
  - Target track system – Uses configurable ADSR envelopes to represent incoming stimuli
- Communication system – Allows AI agents to play sound/voice/animation events

Development Environment

The design and development environment includes the following components:

- Game object model – Entity, movement controller, extensions
- Actor & vehicle system – Health, camera, IK, weapons, animation, etc.
- Flow graph – Visual definition of game logic
- AI debug renderer – HUD, geometric primitives, text labels, graphs, etc.
- Editor
  - AI entities – Properties, flow graphs, scripts
  - Entity archetypes – Templates for properties of individual AI agents
  - AI shapes – AI territories, AI paths, forbidden areas
  - Navigation – Navigation modifiers used instead of triangulation
  - Cover surfaces – CoverSurface anchors to indicate where cover should be
  - Visual AI debugger – Recording AI signals, active behaviors, goals, stimuli, etc.
- Scripting with Lua
  - Entity definitions (including entity flow graph nodes)
  - AI behavior definitions

  - Group behavior definitions
  - Library or shared Lua code (game rules, basic entities)
  - Blackboards to share information globally or among groups
  - Examples of AI functionality available in Lua:
    - AI.Signal
    - AI.FindObjectOfType
    - AI.GetAttentionTargetType (Visual, Memory, Sound, None)
    - AI.GetAttentionTargetAIType (Actor, Grenade, Car, etc.)
    - AI.GetRefPointPosition
    - AI.DistanceToGenericShape
    - AI.SetBehaviorVariable (to change behavior)
    - AI.CanMelee
    - AI.RecComment (make comment for Visual AI Debugger
- Scripting with XML
  - Behavior/coordination trees
  - AI communications
  - Items (e.g., weapons)
- Entity system
  - Spatial queries – GetPhysicalEntitiesInBox()
  - AI agents and vehicles are entities in the Entity system
  - To spawn an entity, its Entity class is required – Can be defined either using the `.ent` file in `Game\Entities` OR through a C++ call to RegisterFactory() in game code
  - An entity pool can be used to limit the number of active AI agents per each specified Entity class.
- AI Debugger and AI Debug Draw
  - Use AI Debugger to track multiple features, including active behavior, signal received, attention target, comments, etc.
  - `ai_DebugDraw`
    - 1 – Basic info on AI agents (selected by `ai_DrawAgentStats`)
    - 74 – All of the navigation graph (can be slow)
    - 79 – Parts of the navigation graph around the player
  - `ai_statsTarget <AI name>` – Detailed info for the specified AI
  - `ai_DebugTargetTracksAgent <AI name>` – Perception information on the specified AI
  - `ai_Recorder_Auto` – Record AI activity in Editor game mode for AI Debugger
  - `ai_DebugTacticalPoints` – Debug TPS queries
  - `ai_DrawPath <AI name>` – Draw the path of the specified AI (optionally specify "all" for all AI agents)
  - `ai_DrawPathFollower` – Draw the actual path being followed
  - `ai_DrawSmartObjects` – Display smart objects and their classes and attributes
  - `ai_DebugDrawEnabledActors` – List currently enabled AI agents.

Execution Context

- AI update is called every frame, but are fully updated only at ~10Hz
- Some AI subsystems use independent time-slicing (pathfinding, tactical point, dynamic waypoints updating, smart object, interest, and dead bodies removal)
- Some services can be called synchronously from game code (such as tactical point system (TPS) queries)

Pathfinding Costs

For agents to behave in a believable way, they need to find paths that are appropriate for their current state. Sometimes these paths will take the most direct route; other times they will be longer paths to maximize use of roads, cover, or other properties of the environment. The current system needs to be extended to support this. The pathfinding system uses A* to find minimal-cost paths.

The cost of a path is given by the sum of the costs of the links that make up that path. Currently the cost traversing a link in the navigation graph is normally simply the physical (3D) length of that link. However, the A* implementation makes it easy for the requester to modify these distance-based costs with simple code changes to extend the current system . For example, the cost of traveling between two road nodes can be scaled by a factor of 0.1 so that road-traveling agents have a strong preference for finding road-based paths.

The cost of a path link connecting two graph nodes should be determined by two sets of properties:

- Link properties, including the path's length.
- Pathfinding agent properties in relation to link properties. For example, a stealthy agent might evaluate a link passing through trees as a lower cost-per-unit-length than one passing along a road. However, the same agent might reach a different conclusion when leading a convoy containing vehicles.

In general, the cost of a link is determined by the product of these two factors: the link-length multiplied by a relative cost-per-unit-length. The latter is what needs to be determined.

Problem: Calculating Costs at Run Time

We want to use the same navigation graph for different kinds of agents. This means that link cost should be calculated at run time by combining the inherent link properties with the agent properties.

Link properties

Associate these properties with each link:

Link.Length

Length of the link (in meters).

Link.Resistance

The link's resistance to traversal. A road would be close to 0, off-road would be larger, water deep enough to require swimming might be close to 1.

Link.Exposure

How exposed the link is. Trees and dense vegetation would be close to 0, low vegetation would be larger, and a road/open space would be close to 1.

Link.DeepWaterFraction

Fraction of the link that contains deep water (e.g., > 1.5m).

Link.DestinationDanger

Additional "danger value" associated with the destination node. A dead body might be 1. This value can be stored in the destination node itself to save memory.

Agent properties

Associate these properties with each agent (normally set when the agent is created):

Agent.CanTraverseTriangular

True/false indicator determining if the agent can traverse triangular nodes.

Agent.CanTraverseWaypoint

True/false indicator determining if the agent can traverse waypoint nodes.

Associate these properties with an agent if relevant for the link type:

Agent.CanSwim

True/false indicator determining if the agent can swim.

Pathfinder request properties

Associate these properties with each agent pathfinder request:

Agent.TriangularResistanceFactor

Extra link cost factor when the link is of type Triangular and its resistance is 1.

Agent.WaypointResistanceFactor

Extra link cost factor when the link is of type Waypoint and its resistance is 1.

Agent.RoadResistanceFactor

Extra link cost factor when the link is of type Road and its resistance is 1.

Associate these properties with an agent pathfinder request if relevant for the link type (note: if a path link has different start/end node types, the result is obtained by averaging):

Agent.SwimResistanceFactor

Extra link cost factor when the link deep water fraction is 1.

Agent.ExposureFactor

Extra link cost factor when the link's exposure is 1.

Agent.DangerCost

Extra link cost when the link danger value is 1.

All link properties, except for Link.DestinationDanger, are calculated when the triangulation is generated. Link.DestinationDanger is initially set to 0 and then calculated as the game runs. For example, whenever a character dies, each link going into the death node will have its DestinationdangerCost incremented by 1. This will cause an agent with Agent.DangerCost = 100 to prefer paths up to 100m longer (assuming no other path cost differences) in order to avoid this death node. These link modifications need to be serialized to support load/save.

In addition, extra costs can be calculated at run time. For example, an extra cost associated with exposure could be added when an agent wishes to find a path that avoids the player; this can be done by using raycasts in the A* callback that calculates costs.

When determining pathfinding costs, there are two problems that need to be solved:

- How should the link properties be calculated?
- How should the link and agent properties be combined to give a numerical cost for traversing each graph link?

Keep in mind that link properties represent the average nature of the environment over the length of the link. If the region has not been triangulated reasonably finely, this may negatively impact the quality of pathfinding results. If the impact is significant, it may be necessary to add additional triangulation points.

An additional issue to consider: should pathfinding differentiate between variable visibility conditions, such as night vs. day or fog vs. clear weather? This would involve splitting the link exposure into terms derived from physical cover and shadow cover. Given the number of links involved, adding too much information of this type to each link should be considered carefully. A simpler solution might be to have stealthy agents be less likely to request a stealthy path in these conditions, or to set the agent's ExposureFactor lower.

Solution

Calculating link properties

Because link resistance is only dependant on the actual type of nodes involved in the link, it can be stored in a lookup table. Here's an example set of resistance values for each node type:

Node type	Resistance
Triangular-no-water	0.5
Triangular-water	1.0
Waypoint	0.6
Road	0
Flight/Volume	0

Note

- Consider adding a separate resistance for Flight/Volume in underwater terrain.
- For links between nodes of different types, the resistance values can be averaged.

The Link.Exposure value, which is stored in the link, is determined by the environment properties sampled over the length of the link. For triangular, waypoint and volume navigation regions, this can be done by firing rays from points along the link. (This is done by using IPhysicalWorld::RayWorldIntersection and checking for HIT_COVER | HIT_SOFT_COVER with COVER_OBJECT_TYPES.) It does not make sense to try to get a really accurate value, because in practice the beautified path will not follow the link directly.

Combining link and agent properties

Link cost must account for intersections between link properties and agent properties. For example: if a link is marked as containing deep water and the agent cannot swim, the link should be treated as impassable.

A factor representing the extra costs associated with travel resistance and exposure will be calculated, and the total link cost should be set as follows:

```
Cost = Link.DestinationDanger * Agent.DangerCost + (1 + Factor) * Link.Length
```

where

```
Factor = Agent.[link-type]ResistanceFactor * Link. [link-type]Resistance +
 Agent.ExposureFactor * Link.Exposure
```

Consider this scenario: with no exposure/destination costs, and assuming that road links have Link.Resistance {{0}} while off-road links have Link.Resistance {{0.5}}, then in order to make road travel ten times more attractive than off-road (such as if the agent is a car), the agent could have Agent.TriangularResistanceFactor set to {{(10-1)/0.5}} (or 18) and Agent.RoadResistanceFactor set to 0.

If the agent is a human character that always moves at about the same speed whether or not it is on or off a road, then it could have both Agent.TriangularResistanceFactor and Agent.RoadResistanceFactor set to 0.

Assuming the agent can traverse deep water or is not affected by water (such as a hovercraft), Agent.SwimResistanceFactor could be set to 0. For a human agent, this factor might be set to a value as high as 3.0, so that the agent will take significant detours to avoid swimming across a river.

Sensory Models

Overview

This topic describes the modelling and principal operation of the sensors implemented in the Lumberyard AI system. These include the visual sensors, sound sensors, and a general-purpose signalling mechanism.

Sensory information is processed during a full update of each enemy (the actual time that a sensory event was received is asynchronous). These sensors are the only interface the enemy has with the outside world, and provide the data that the enemy will use to assess their situation and select potential targets. All sensors are completely configurable, and they can be turned on/off at run-time for any individual enemy.

Vision

The visual sensory model is the heart of the AI system. It is an enemy's most important sense. The model is designed to simulate vision as realistically as possible, while still maintaining a low execution cost, using a combination of compromises and optimizations.

During a full update for an individual enemy, the system traverses all potential targets from the enemy's point of view and runs each one through a visibility determination routine. All targets that survive this filtering procedure are placed in a visibility list that is maintained until the next full update. For a target to persist as "visible" it must pass the visibility test in each full update. Targets that change from visible to not visible during an update are moved to a memory targets list. If a previously visible target becomes visible again, it is moved from the memory target list back to the visibility list. Memory targets have an expiration time to simulate the enemy "forgetting" the target; this time interval is determined by several factors, including the threat index of the target and the length of time it was visible. Visible targets are given the highest priority and will become the current attention target even if there is another target with a higher threat index. This approach simulates the natural tendency of humans to act based on what they see faster than on what they remember (or hear).

Visibility Determination

The visibility determination routine determines whether a target is considered visible to an enemy. It is run against each of the enemy's potential targets during a full update.

Identifying Targets

Visibility determination can be very CPU intensive; to mitigate this cost, only potential targets are evaluated for visibility. There is a mechanism to register any AI object as an object that should be

included in the visibility determination (including user-defined objects). This includes objects such as the grenades in Lumberyard, flashlights, etc. There are also special objects called attributes, which will be discussed in more detail later in this topic.

To be considered a potential target, an AI object must be:

- currently active
- of a different species than the enemy (enemies don't need to keep track of members of their own team)

In addition, the visibility determination test is performed automatically against the player, even if the player is of the same species as the enemy. This rule ensures that the player is accurately specified as an object type and is always taken into account when checking visibility.

The game developer can also designate certain AI object types for visibility determination. These user-defined types are added to a list maintained by the AI system identifying object types to be included in the visibility check. Objects can be freely added to and removed from this list, even from script. To include an object in the list, specify an assessment multiplier to the desired object type. For example, refer to the file `aiconfig.lua`, which can be found in the `/scripts` directory. For more about assessment multipliers, see the topics on threat assessment.

Checking Visibility

Each potential target identified is evaluated for visibility using a series of tests. In situations where the player is facing a single species, no visibility determination is performed between AI enemy objects, only against the player.Key measures determining visibility include:

Sight-range test

This check is done first, as it is fast and cheap to filter out all AI objects that are outside the enemy's sight range. This is done by comparing the distance between enemy and target against the enemy's sight range value.

enemy sight range

Floating point value that determines how far the enemy can see (in meters); the value represents the radius of a sphere with the enemy at the center.

Field-of-view test

Objects that are inside the enemy's sight range sphere are then checked for whether they are also inside the enemy's field of view (FOV).

enemy field of view

Floating point value that determines the angle of the enemy's visibility cone (in degrees); the cone's tip is at the enemy's head and extends outward in the direction the enemy is facing.

The FOV is the angle that determines how far the enemy can see to the left and to the right of his current forward orientation (that is, the scope of his peripheral vision). For example, an FOV of 180 degrees means that the enemy can see everything which is 90 degrees or less to the left and 90 degrees or less to the right of the direction in which he is currently facing. An FOV of 90 degrees means that he can see 45 degrees or less to the left and 45 degrees to the right of his current forward orientation. The FOV check is performed using a simple dot product between the enemy's orientation vector and the vector created as the difference between the positions of the potential target and the enemy. The resulting scalar is then compared to the value of the FOV. Note that by using a conical shape, FOV is not limited to 2D representations.

Physical ray test

Objects that survive the two initial checks are very likely to be seen. The next check is an actual ray trace through the game world, which is an expensive process. Because the low layer of the AI system performs distributed updates over all frames, it is very seldom that a large number of rays needs to be shot per frame. Exceptions include scenes with a high number of objects belonging to different species and huge combat scenes, such as those with more than 20 participants per species.

The visibility physical ray is used to determine whether there are any physical obstacles between the enemy and the target. It originates from the head bone of the enemy character (or if the enemy does not have an animated character, it originates from the entity position – which is often on the ground) and is traced to the head bone of the target (if it has one, otherwise the entity position is used). If this visibility ray hits anything in its path, then the target is considered not visible. If the ray reaches the target without hitting any obstacles, then the target has passed this tests and can be added to the visibility list for this update.

Not all obstacles are the same. The physical ray test distinguishes between hard cover and soft cover obstacles. For more information on how cover type affects enemy behavior, see the section on soft cover later in this topic.

Perception test

This test is for player AI objects only (and other AI objects as defined by the game developer). Once the player has passed all the visibility tests for an enemy, this final test determines whether or not the enemy can see the player object. Each enemy calculates a perception coefficient for the player target, which ultimately describes the likelihood that the enemy can see the target.

perception coefficient (SOM value)

> Floating point value (between 0 and 10) that defines how close the enemy is to actually seeing the target.

The perception coefficient is calculated based on a range of factors, including the distance between enemy and target, height of the target, and whether the target is moving. The value must reach the maximum value (currently 10) before it can receive a definite visual stimulus--that is, see the target.

For more details on how a perception value is derived, see the section on calculating perception later in this topic.

Soft Cover Visibility and Behavior

The physical ray test also evaluates the surface type of obstacles when determining visibility. The AI system can discriminate between two types of surfaces: soft cover and hard cover. The primary difference in a physical sense is that game players can pass through soft cover but cannot pass through had cover. Players can hide behind soft cover objects but the visibility determination is slightly “skewed” when a target is behind a soft cover object rather than a hard cover object or just in the open. When determining a target's visibility behind soft cover, the AI system takes into account whether or not the enemy already identified the target as "living" (not a memory, sound or other type of target). If the enemy does not have a living target, then the soft cover is considered equal to hard cover and normal visibility determination is performed. This occurs when the enemy is idle--or when the enemy is looking for the source of a sound but has not yet spotted it.

However, the behavior is slightly different when the enemy already has a target identified. During the physical ray test, if only soft cover is detected between the enemy and their target, then the target will remain visible for short length of time--between 3 and 5 seconds. If the target remains behind soft cover during this time, the enemy will eventually lose the target and place a memory target at the last known position. However, if the target leaves soft cover within this time, then the timer is reset and normal visibility rules are put into effect.

This behavior simulates the following example: when a soldier perceives that the target has run inside a bush, they do not immediately forget about it because they can make out the target's silhouette even inside the bush. But following a target like that is difficult over time, and after a while the soldier will lose track of the target. The same rules apply to covers made of penetrable cover, like wood, but the rationale is a bit different. If a target runs behind a thin wooden wall, the soldier knows that bullets will still pierce the wall, so for a short time the target's position is still known, and the enemy continues to shoot through it. This can make for some really intense situations in a Lumberyard game.

In order for this process to work in a closed and rational system, all surfaces in the game need to be properly physicalized (wood, grass, and glass should be soft cover, while rock, concrete, metal should be hard cover). This is consistently done in Lumberyard.

Perception Calculation

Unlike visibility between AI agents, visibility of player objects to enemy AI agents in Lumberyard is not an on/off switch. This added layer of complexity is designed to allow for variations in game playing style (such as action versus stealth). Perception allows the player to make a certain number of mistakes and still be able to recover from them. (This is one of the reasons why a player AI object is specifically defined even in the lowest layer of the AI system hierarchy.) It is not used with other AI objects, where "switch" vision is used (that is, the target is visible as soon as a ray can be shot to its position). Note that it is possible to declare some AI objects should also trigger use of a perception coefficient.

An enemy is given a perception coefficient that describes how close the enemy is to actually seeing a particular target. The initial value of the perception coefficient is 0 and increases or decreases based on a defined set of rules. If a player target passes all prior visibility tests, the enemy begins applies the perception coefficient. Once the maximum value has been reached, the player target is visible to the enemy. This statement contains several corollaries:

- Each enemy has a perception coefficient for each player target it is processing.
- Each enemy will receive notification that the player target is visible only after the perception coefficient reaches maximum value.
- The perception coefficient of two different enemies are unrelated, even for the same player target.
- There is no game-level perception coefficient (that is, a value that determines how any enemy perceives a player target), although this information could be derived by statistics.

When an enemy starts receiving notification that a player target is passing the visibility determination routine, it begins calculating the perception coefficient. This is done by evaluating the following factors, all of which impact the rate at which the coefficient increases. Keep in mind that a player target must still pass all other phases of the visibility determination routine before the perception coefficient is applied.

Distance

Distance between the enemy and the player target has the highest influence on perception. The closer the player target is to the enemy, the faster the coefficient rises, while greater distances cause the coefficient to rise slower. The increase function is a basic quadratic function. At distances very close to the enemy, the time to reach maximum perception is almost non-existent and the target is instantly seen. In contrast, a player target may be able to move more freely along the boundaries of the enemy's sight range, as the perception value rises more slowly.

Height from ground

This factor takes into account the player target's distance above the ground. The rationale for this behavior is that a prone target is much harder to spot than one who is standing upright. The AI system measures the distance of the target from the ground based on the "eye height" property of an AI object. This property is set when the AI object is initialized, and can be changed at any time during execution of the game. If enemies and players are represented in the game by animated

characters, the eye height is calculated using the actual height of the character's head bone. This factor influences the rate of increase in the perception coefficient as follows: if the player target has a height above ground of less than one meter, the increase due to distance is lowered by 50%.

Target motion

The perception coefficient is affected by whether or not the player target is moving. Movement attracts attention, while stationary targets are harder to spot. This factor influences the rate of increase in the perception coefficient as follows: if the player target is standing still, the increase due to other factors is lowered by additional 50%.

Artificial modifiers

Additional values can define how fast the perception coefficient increases. Some affect all enemies in the game world, while some affect only particular targets. An example of a modifier that affects all enemies is the console variable `ai_SOM_SPEED`. Its default value varies depending on a game's difficulty level, but it provides a constant multiplier that is applied on top of all other calculations, and it applies to all enemies. In contrast, it is possible to set a custom multiplier for a specified object type that is used only for certain player targets; however, this setting is limited to the lowest level of the AI system and is not available for tweaking.

The effect of perception is cumulative while the target is considered visible to the enemy. A floating point value is calculated based on the factors described above, and each time the enemy fully updated, this value is added to the perception coefficient (along with an updated visibility determination). So, for example, a player target that is within the enemy's range of sight might remain unperceived by the enemy significantly longer if they are crouching and motionless.

At the same time, a non-zero perception coefficient can fall back to zero over time if value is not increased constantly with each full update. For example, a player target might become visible for a few seconds, raise the coefficient up to 5, and then break visual contact. In this scenario, the coefficient will drop slowly to zero. This scenario was implemented to reward players that tactically advance and then pause before continuing; players can wait for the coefficient to drop to zero before continuing to sneak.

A statistical overview of the perception coefficients of all enemies for a player is used for the HUD stealth-o-meter, showing as a small gauge to the left and right of the radar circle in the HUD. It represents the highest perception coefficient of the player across all enemies that currently perceive him. In effect, it shows the perception coefficient of the one enemy that is most likely to see the player. so, a full stealth-o-meter does not mean that all enemies see the player; it means that there is at least one enemy that can. An empty stealth-o-meter means that currently no enemy can see the player.

Attribute Objects

An attribute object is not a full AI object; instead, it is more of a special helper that can be attributed to an existing AI object. The attribute is a special class of AI object, specifically defined at the lowest level in the AI system. Every attribute object must have a principal object associated with it. The principal object can be any type of an object (including puppet, vehicle, player, etc,.) but cannot be an attribute.

Attributes can impact visibility determination. When an enemy determines that it sees an attribute object, the system will switch the attribute with the principal object before adding it into the visibility list of the enemy. Thus, an enemy who sees an attribute will believe it is seeing the principal object attached to the attribute.

Essentially, attributes are a systematic way of connecting certain events to a single target. For example, a player switches on a flashlight and the beam hits a nearby wall. The light hitting the wall creates an attribute object associated with the principal object, which is the player. In this scenario, the player is hidden, but because an enemy sees the attribute object (the light on the wall), it will in fact "see" the player. The rationale is that enemies have enough intelligence to interpolate the origin of the light ray and thus know the player’s position.

This feature is also used with regard to rocks, grenades, rockets etc. It can be extended to add more features to a game; for example, a target might leave footprints on the ground that evaporate over time. The footprints spawn attribute objects, which enable any enemy who sees them to perceive the location of the target who left them. Another application might be blood tracks.

To ensure that attribute objects are included in the visibility determination, they must have an assessment multiplier set. Refer to `aiconfig.lua` in the `Scripts\AI` directory to see where the AI system defines the multiplier for attribute objects.

Flight

Use these archetypes and flow nodes in conjunction with entities to control flying vehicles. See these archetypes in the characters archetype library:

- **CELL/Helicopter.Regular**
- **Ceph/Dropship.Regular**
- **Ceph/Gunship.Regular**

The following flow nodes to be used with these entities are found under the Helicopter category.

FollowPath

This flow node sets the current path that the flight AI uses.

- When the AI is not in combat mode.
  - If the AI is set to loop through the flow node path, the AI tries to go from its current location to the closest point of the path and then follows it to the end. The node outputs indicating that the AI has reached the end of the path is sent once only.
  - Without looping, the AI tries to go from its current location to the beginning of the path and then follows it to the end.
- When the AI is in combat mode.
  - If the target is visible, the path is used to position the AI in the best location to attack the target. It is also used to navigate between positions within the path.
  - If the target is not visible, the path is used as a patrol path. Where possible, it simplifies setup to have paths in combat mode be set to loop.

EnableCombatMode

This flow node enables or disables the AI's ability to position itself within the combat path in order to engage and shoot at its current target. By default, an AI is not in combat mode until it's explicitly allowed to go into combat mode.

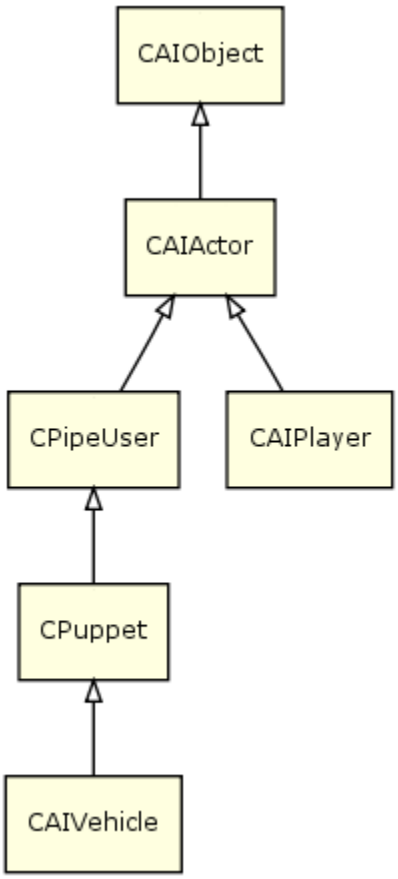
- When an AI is in combat mode and has identified a target location, it will try to reposition itself within the current path to a position from which it can engage.
- Any character of an opposing faction is a good candidate for a target.

EnableFiring

This flow node enables or disables the ability of the AI to shoot at its current target when in combat mode. By default, an AI is allowed to fire when in combat mode until it's explicitly disabled using this node.

AI C++ Class Hierarchy

C++ classes for AI objects are structured as follows.



CAIObject

Defines basic AI object properties (entity ID, position, direction, group ID, faction, etc.)

CAIActor

Basic perception and navigation, behavior selection, coordination, blackboard, AI territory awareness, AI proxy

CAIPlayer

AI system's representation of an actual game player

CPuppet

Aiming, firing, stances, covers, a full-fledged AI agent

CAIVehicle

Vehicle-specific code

AI System Concept Examples

This section includes the following topics:

- AI Multi-Layered Navigation (p. 17)
- Individual AI: Dynamic Covers (p. 17)
- Individual AI: Tactical Points (p. 17)
- Group and Global AI: Factions (p. 18)
- Group and Global AI: Flow Graphs (p. 18)

AI Multi-Layered Navigation

Useful AI debug draw:

- ai_useMNM=1
- ai_DebugDraw=1
- ai_DebugDrawNavigation=1
- ai_DrawPath=all
- ai_DrawPathFollower=1

Individual AI: Dynamic Covers

Example: CoverSurface and HMMWV

This example shows the use of dynamic covers that are generated offline and adjusted during run time.

Useful AI debug draw:

- ai_DebugDraw=1
- ai_DebugDrawCover=2
- [AI/Physics] is on

Individual AI: Tactical Points

Example: A very shy civilian who always wants to hide from the player

- Tactical point system (TPS) query:

```
AI.RegisterTacticalPointQuery({
    Name = "Civilian_HideFromEnemy",
    {
        Generation =
        {
            cover_from_attentionTarget_around_puppet = 25
        },
            Conditions =
        {
            reachable = true,
        },
        Weights =
        {
            distance_from_puppet = -1,
        },
    },
});
```

- Useful AI debug draw:
  - ai_DebugTacticalPoints=1

  - ai_StatsTarget=Grunt1
  - ai_TacticalPointsDebugTime=10
- For more realism, add the following before goalop TacticalPos:

```
<Speed id="Sprint"/>
```

Group and Global AI: Factions

Example: AI formations of different factions

Place on a map three grunts of the following factions. Note who is hostile to who.

- grunts
- assassins
- civilians

For example:

```
<Factions>
    <Faction name = "Players">
        <Reaction faction- "Grunts" reaction="hostile"/>
        <Reaction faction- "Civilians" reaction="friendly"/>
        <Reaction faction- "Assassins" reaction="hostile"/>
    </Faction>
    <Faction name="Civilians default="neutral"/>
    ...
</Factions>
```

(see `Game/Scripts/AI/Factions.xml`)

Group and Global AI: Flow Graphs

Flow Graph Editor allows non-programmers to build global AI logic visually. Experiment with flow graph debugger features, such as signal propagation highlighting and breakpoints.

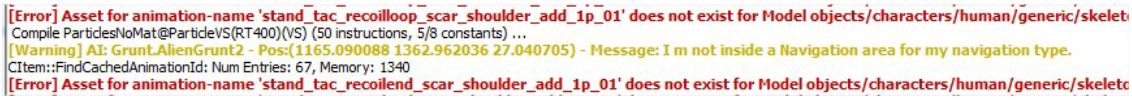
AI Bubbles System

The AI Bubbles system collects AI error messages for level designers to address. This system streamlines the debugging process by helping to track down which system(s) are connected to a problem. To use the AI Bubbles system, programmers need to push important messages into the system, which will then provide notification to the level designers when a problem is occurring.

Message Display Types

Message include a set of information (agent name, position, etc.) that help the designer to understand that something is wrong in the normal flow. Message notifications can be displayed in any of the following ways:

- Speech bubble over the AI agent
- Error message in the console



- Blocking Windows message box

Specifying Notification Display Types

Use one of the following ways to specify a display type for error messages:

Console

ai_BubblesSystem

Enables/disables the AI Bubbles System.

ai_BubblesSystemDecayTime

Specifies the number of seconds a speech bubble will remain on screen before the next message can be drawn.

ai_BubblesSystemAlertnessFilter

Specifies which notification types to show to the designer:

- 0 - No notification types
- 1 - Only logs in the console
- 2 - Only bubbles
- 3 - Logs and bubbles
- 4 - Only blocking popups
- 5 - Blocking popups and logs
- 6 - Blocking popups and bubbles
- 7 - All notifications types

ai_BubblesSystemUseDepthTest

Specifies whether or not the notification needs to be occluded by the world geometries.

ai_BubblesSystemFontSize

Specifies the font size for notifications displayed in the 3D world.

C++

In C++, use the method AIQueueBubbleMessage() to define how to display the message notification.

Method signature:

```
bool AIQueueBubbleMessage(const char* messageName, const IAIObject* pAIObject, const char*
 message, uint32 flags);
```

Parameters:

messageName

String describing the message. This is needed to queue the same message error more than once. (The message can be pushed into the system again when it expires is deleted from the queue.)

pAIObject

Pointer to the AI object that is connected to the message.

message

Text of the message to be displayed.

flags

Notification type. This parameter can include one or more flags; multiple flags are separated using a pipe (|).

- eBNS_Log
- eBNS_Balloon
- eBNS_BlockingPopup

Example:

```
AIQueueBubbleMessage("COPStick::Execute PATHFINDER_NOPATH non continuous", pPipeUser, "I
 cannot find a path.", eBNS_Log|eBNS_Balloon);
```

Lua Script

```
local entityID = System.GetEntityIdByName("Grunt.AlienGrunt1");
                AI.QueueBubbleMessage(entityID,"I cannot find a path.");
```

AI Tactical Point System

The Tactical Point System (TPS) provides the AI system with a powerful method of querying an AI agent's environment for places of interest. It includes the GetHidespot functionality and expands on the "hide" goalop.

TPS is a structured query language over sets of points in the AI's world. Using TPS, AI agents can ask intelligent questions about their environment and find relevant types of points, including hidespots, attack points, and navigation waypoints. The TPS language is simple, powerful, and designed to be very readable.

For example, this query requests all points that match the following criteria:

- Generate locations within 7 meters of my current location where I can hide from my attention target.
- Only accept places with excellent cover that I can get to before my attention target can.
- Prefer locations that are closest to me.

```
hidespots_from_attentionTarget_around_puppet = 7
coverSuperior = true, canReachBefore_the_attentionTarget = true
distance_from_puppet = -1
```

TPS uses a highly efficient method to rank points, keeping expensive operations like raycasts and pathfinding to an absolute minimum. Queries are optimized automatically.

This section includes the following topics:

- Tactical Point System Overview (p. 21)

- TPS Query Execution Flow (p. 22)
- TPS Querying with C++ (p. 22)
- TPS Querying with Lua (p. 24)
- TPS Query Language Reference (p. 25)
- Point Generation and Evaluation (p. 28)
- Integration with the Modular Behavior Tree System (p. 30)
- Future Plans and Possibilities (p. 30)

Tactical Point System Overview

Key features of the Tactical Point system (TPS) include:

- Use of a structured query language
  - Powerful and quick to change in C++ and Lua
- Query support for a variety of point characteristics, beyond conventional hiding places behind objects:
  - Points near entity positions
  - Points along terrain features
  - Points suggested by designers
  - Arbitrary resolutions of nearby points in the open or on terrain
- Query combinations, such as:
  - "Find a point somewhere behind me AND to my left, AND not soft cover, AND not including my current spot"
  - "Find a point hidden from my attention target AND visible to the companion"
- Preferential weighting, such as:
  - Find a point nearest to (or furthest from) a specified entity
  - Balance between points near an entity and far from the player
  - Prefer points in solid cover over soft cover
- Query fallback options, such as:
  - Prioritize good cover nearby; if none exists, go backwards to any soft cover
- Query visualization:
  - See which points are acceptable and which are rejected, as well as their relative scores
  - See how many expensive tests are being used by a query and on which points
- Automatic query optimization
  - Understands the relative expense of individual evaluations comprising queries
  - Dynamically sorts points based on potential fitness, according to weighting criteria
  - Evaluates the "fittest" points first, in order to minimize the use of expensive tests
  - Recognizes when the relative fitness of a point indicates that it can't be beat, in order to further reduce evaluations
  - Provides framework for further optimization specific to architecture, level, or locale

In addition to these key feature benefits, this framework offers these advantages from a coding perspective:

- Separates query from action
  - Arbitrary queries can be made at any time without changing the agent's state
- Query language is easy to expand
- Easily adapted for time-slicing (and in principle multi-threading):

  - Progression through query is fine-grained
  - Progress is tracked as state, so it can be easily paused and resumed
- Provides mechanism for delaying expensive validity tests on generated points until needed

TPS Query Execution Flow

The following steps summarize the definition and execution stages of a TPS query. Note that only stages 3 and 4 have a significant impact on performance.

1. Parse query:
   - Parse query strings and values.
   - This step is usually performed once and cached.
2. Make query request:
   - Query is made using C++, ScriptBind, goalops, etc.
   - A query is stateless; it does not imply a movement operation.
3. Generate points:
   - Create a set of candidate points.
   - Point candidates are based on the query's Generation criteria.
4. Evaluate points (this is by far the most intensive stage):
   - Accept or reject points based on Conditions criteria.
   - Assign relative scores to points based on Weights criteria.
5. Consider query fallbacks:
   - If no point matches the Conditions criteria, consider fallback options.
   - Where there is a fallback, return to step 3.
6. Visualize points:
   - If visualization is required, draw all points to screen.
   - Include point data such as its accepted/rejected status and relative scores.
7. Return results:
   - Return one or more points, if any fit the query conditions.
   - Each point is returned as a structure that describes the selected point.

There are some optimizations possible that depend on the execution flow. For example, relevant query results can be cached between fallback queries.

TPS Querying with C++

These C++ interfaces allow you to use TPS from other C++ code and within goalops. Lua queries are translated through it.

There are two C++ interfaces:

- Internal - For use only within the AI system
  - Uses a CTacticalPointQuery object to build queries
  - Allows you to create or adapt queries on the fly
  - Provides greater access to relevant AI system classes
- External - For use from any module
  - Suitable for crossing DLL boundaries
  - Simpler, not object-oriented, just as powerful

- Uses stored queries for efficiency

Internal Interface Syntax

In the example below, some parsing is obviously taking place here. This is crucial to the generality of the system.

```
// Check for shooter near cover using TPS
static const char *sQueryName = "CHitMiss::GetAccuracy";
ITacticalPointSystem *pTPS = gEnv->pAISystem->GetTacticalPointSystem();
int iQueryId = pTPS->GetQueryID( sQueryName );
if ( iQueryId == 0 )
{
    // Need to create query
    iQueryId = pTPS->CreateQueryID( sQueryName );
    pTPS->AddToGeneration( iQueryId, "hidespots_from_attentionTarget_around_puppet", 3.0f);
    pTPS->AddToWeights( iQueryId, "distance_from_puppet", -1.0f);
}
pTPS->Query( iQueryId, CastToIPuppetSafe( pShooter->GetAI() ),vHidePos, bIsValidHidePos );
```

TPS Syntax Examples

The following examples and explanations illustrate the use of TPS query syntax. For a more detailed discussion of the TPS query language, see the topic on TPS Query Language Syntax and Semantics.

```
option.AddToGeneration("hidespots_from_attTarget_around_puppet", 50.0)
```

This query request is expressed as generation criteria and specifies a float to represent distance. The query is broken up into five words:

- "hidespots" indicates that generated points should positioned behind known cover as is conventional
- "from" and "around" are glue words to aid readability
- "target" specifies the name of the object to hide from
- "puppet" identifies a center location that points will be generated around
- The float value indicates the radial distance, measured from the center location, that defines the area within which points should be generated

Note that no raycasts are performed at this stage. We have here considerable flexibility, for example, how we choose to hide from a player: (1) somewhere near the player, (2) somewhere near us, or (3) somewhere near a friend. We can also specify a completely different target to hide from, such as an imagined player position. By providing flexibility at the point generation stage, we can support more powerful queries and allow users to focus computations in the right areas.

```
option2.AddToConditions("visible_from_player",true)
```

This query request is expressed as condition criteria, so we can expect a Boolean result. The query specifies points that are visible to the player, which is curious but perfectly valid. The term "visible" specifies a ray test, with "player" specifying what object to raycast to from a generated point.

```
option2.AddToConditions("max_coverDensity",3.0)
```

This query is expressed as a condition, so we can expect a Boolean result. The term "Max" specifies that the resulting value must be compared to the given float value--and be lower than. The term "coverDensity" identifies this as a density query (measuring the density of things like cover, friendly AI agents, etc.) and specifies measurement of covers.

```
option1.AddToWeights("distance_from_puppet",-1.0)
```

This query is expressed as a weight component; the query result will be a value between zero and one (normalized as required). Boolean queries are allowed to indicate preference (such as primary cover over secondary cover), with return values of 0.0 for false and 1.0 for true.

This query component indicates a preference for points at a certain location relative to an object. The term "distance" identifies this as a distance query, with the given float values specifying the distance amount. The term "puppet" identifies the object to measure the distance from.

TPS Querying with Lua

In Lua, there are two ways to use the TPS:

- Scriptbinds allow you to use TPS queries from a Lua behavior and have the results returned as a table without any side-effects. This can be useful for higher-level environmental reasoning, such as:
  - Choose behaviors based on suitability of the environment (for example, only choose a "sneaker" behavior if there's lots of soft cover available).
  - Run final, very specific tests on a short list of points, rather than adding a very obscure query to the TPS system.
  - Enable greater environmental awareness (for example, tell me three good hidespots nearby, so I can glance at them all before I hide).

- With goal pipes, you can use goalops to pick a point and go there, using a predefined TPS table:
  - Use a "tacticalpos" goalop, which is equivalent to a previous "hide" goalop.
  - Use fallback queries to avoid lists of branches in goalpipes.
  - More flexible goalops can be provided to decouple queries from movement.

Both methods define query specifications using the same table structure, as shown in the following example:

```
Hide_FindSoftNearby =
{
  -- Find nearest soft cover hidespot at distance 5-15 meters,
  -- biasing strongly towards cover density
  {
    Generation= {   hidespots_from_attentionTarget_around_puppet = 15 },
    Conditions= {   coverSoft = true,
            visible_from_player = false,
            max_distance_from_puppet = 15,
            min_distance_from_puppet = 5},
    Weights =   {   distance_from_puppet = -1.0,
    coverDensity = 2.0},
  },
  -- Or extend the range to 30 meters and just accept nearest
  {
    Generation ={   hidespots_from_attentionTarget_around_puppet = 30 },
    Weights =   {   distance_from_puppet = -1.0}
  }
}
AI.RegisterTacticalPointQuery( Hide_FindSoftNearby );
```

Note
Registering a query returns a query ID that then refers to this stored query.

Querying with Scriptbind

The following script runs a query using an existing specification. See comments in `Scriptbind_AI.h` for details.

```
AI.GetTacticalPoints( entityId, tacPointSpec, pointsTable, nPoints )
```

Querying with Goalops

The following script runs an existing query. Because queries can have fallbacks built in, branching is usually unnecessary (the branch tests are still supported).

```
AI.PushGoal("tacticalpos",1, Hide_FindSoftNearby);
```

TPS Query Language Reference

There are ways to define a query in both C++ and Lua (and potentially in XML), but the same core syntax is used. This page formally defines the TPS query language, with query components expressed in Generation, Conditions or Weights, and defines and discusses the query language semantics.

Query Syntax

Note
Non-terminal symbols are in bold. Not all of the symbols are implemented, but are shown for illustration.

```
Generator ::= GenerationQuery '_' 'around' '_' Object
Condition ::= BoolQuery | (Limit '_' RealQuery)
Weight    ::= BoolQuery | (Limit '_' RealQuery) | RealQuery
GenerationQuery ::= ( 'hidespots' '_' Glue '_' Object)
                    | 'grid' | 'indoor'
BoolQuery ::= BoolProperty | (Test '_' Glue '_' Object)
BoolProperty ::= 'coverSoft' | 'coverSuperior' | 'coverInferior' | 'currentlyUsedObject' |
 'crossesLineOfFire'
Test ::= 'visible' | 'towards' | 'canReachBefore' | 'reachable'
RealQuery = ::= RealProperty | (Measure '_' Glue '_' Object)
RealProperty ::= 'coverRadius' | 'cameraVisibility' | 'cameraCenter'
Measure ::= 'distance' | 'changeInDistance' | 'distanceInDirection' | 'distanceLeft' |
 'directness' | 'dot' | 'objectsDot' | 'hostilesDistance'
Glue ::= 'from' | 'to' | 'at' | 'the'
Limit ::= 'min' | 'max'
Object ::= 'puppet' | 'attentionTarget' | 'referencePoint' | 'player'
       | 'currentFormationRef' | 'leader' | 'lastOp'
```

Query Semantics

Note

- "Tunable" denotes that the exact values used should be possible to tweak/tune later.
- "Real" means that it returns a float value (rather than a boolean).

Objects

puppet

AI agent making a query

attentionTarget

Object that is the target of the AI agent's attention

referencePoint

AI agent's point of reference, perspective

player

Human player (chiefly useful for debugging and quick hacks)

Glue

from | to | at | the

Glue words used for readability in a query statement. Each query must have a glue word, but it has not active function and the parser doesn't distinguish between them. Readability is encouraged to aid in debugging and long-term maintenance.

Generation

Hidespot

Individual point just behind a potential cover object with respect to a "from" object (as in "hide from object"). There is typically one point per cover object. Use of this symbol should generate multiple points behind large cover objects and cope with irregularly shaped and dynamic objects.

Around

A glue word with special meaning. This word should be followed by the name of an object around which to center the generation radius.

Conditions/Weight Properties (use no object)

These properties relate to a specified point:

coverSoft

Boolean property, value is true if the specified point is a hidespot using soft cover.

coverSuperior

Boolean property, value is true if the specified point is a hidespot using superior cover.

coverInferior

Boolean property, value is true if the specified point is a hidespot using inferior cover.

currentlyUsedObject

Boolean property, value is true if the specified point is related to an object the puppet is already using (such as the puppet's current hide object).

coverRadius

Real (float) property, representing the approximate radius of the cover object associated with the specified point, if any, or 0.0 otherwise. When used for condition tests, returns an absolute value in meters. When used as a weight, returns a normalized value, mapping the range [0.0-5.0m] to [0.0-1.0]. (Tunable)

coverDensity

Real property, representing the number of potential hidespots that are close by to the specified point. When used for condition tests, returns an absolute value representing an estimate of the number of hidespots per square meter using a 5m radius sample. When used as a weight, returns a normalized value, mapping the range (0.0-1.0) to [0.0-1.0] (hidespots per square meter). (Tunable)

Conditions/Weight Test/Measures (require object)

These properties relate to a specified object, such as distance_to_attentionTarget or visible_from_referencePoint.

distance

Real (float) measure, representing the straight-line distance from a point to the specified object. When used for condition tests, returns an absolute value in meters. When used as a weight, returns a normalized value, mapping the range [0.0-50.0m] to [0.0-1.0]. (tunable)

changeInDistance

Real (float) measure representing how much closer the puppet will be to the specified object if it moves to a certain point. Takes the distance to the specified object from the current location and subtracts it from the distance to the object from the proposed new location. When used for condition tests, returns an absolute value in meters. When used as a weight, returns a normalized value, mapping the range [0.0-50.0m] to [0.0-1.0]. (tunable)

distanceInDirection

Real (float) measure representing the distance of the point in the direction of the specified object. Takes the dot product of the vector from the puppet to the point and the normalized vector from the puppet to the object. When used for tests, returns an absolute value in meters. When used as a weight, returns a normalized value, mapping the range [0.0-50.0m] to [0.0-1.0]. (tunable)

directness

Real (float) measure representing the degree to which a move to a certain point approaches the specified object. Takes the difference in distance to the object (changeInDistance) and divides it by the distance from the puppet to the point. Always uses the range [-1.0 to 1.0], where 1.0 is a perfectly direct course and negative values indicate movement further away from the object.

Limits

min | max

Limits can be used to test a real value in order to product a Boolean. Useful for conditions that can also be used as coarse Weights; for example, the condition MAX_DISTANCE = 10 can be used to express that a distance of less than 10 is preferable, but without favoring nearer points in a more general way.

Failing Queries

There are a few different ways queries can fail, and it's important to understand how each case is handled.

- **No points matched the conditions of the query.** This is a valid result, not a failure; the AI can move to fallback queries or try a different behavior.
- **The query does not make sense in the context of an individual point.** Sometimes a query doesn't make sense for a certain point or at a certain time. In this case, the query tries to return the "least surprising" results. For example: a query about a point generated in the open asks "is this soft cover?" The result will be "false", because this isn't any kind of cover. Query names should be chosen carefully to help avoid potential confusion.
- **The query does not make sense in the context of the puppet, at this time and for any point.** As with the point context issue, the query tries to return the "least surprising" results. For example: a query about a puppet asks "am I visible to my attention target?" when the puppet doesn't have an attention target. The query could return false, but it would disqualify every point. This case will usually

indicate a code error--the puppet should have an attention target at this point, but does not. Note: This situation can cause a similar problem in the point generation phase, with a query like "generate hidespots from my attention target". both of these situations are flagged as code errors.

- **The query failed due to a code error.** You can test for errors in the TPS queries and raise them there also. For example, a query or combination hasn't been fully implemented yet or is being used as a kind of assert to test variables.

Point Generation and Evaluation

An AI agent makes a TPS point generation query in order to generate a set of points for consideration. Once generated, each point can be evaluated based on its position and any available metadata.

Generating Points

Input

The following information are required to generate points:

- Specific criteria defining the types of points to generate.
- A central or focal position around which to generate points. This might be the current position of the puppet itself, an attention target, or some other given position.
- For some queries, the position of a secondary object, such as a target to hide from.

It is possible to specify multiple sets of point generation criteria. For example, a query might request point generation around both the puppet and an attention target.

Processing

Based on the input, TPS begins generating points to evaluate. All points fall into two main types:

- Hidepoints. These are generated based on the following calculations:
  - Hideable objects
  - Generated only if a position to hide from was provided
  - Hidepoints represent final positions, for example calculating positions behind cover
  - Using the object and delaying finding an actual point is a possibility
- Open points. These are generated based on query specifications and the following calculations:
  - Usually on terrain, but may be on surfaces, etc.
  - Resolution/pattern (such as triangular with 1-meter spacing)
  - Potentially may perform more general sampling to find an exact point, but an initial resolution is still required
  - Radial/even distributions

Output

The result of a point generation query is a list of point objects. Each point object includes the point's position and available metadata, such as any associated hide objects.

Evaluating Points

Once a generation query generates a set of points, they can be evaluated. Point evaluation tries to establish the "fitness" of each point, that is, how well the point matches the specified criteria. The goal is to choose either one good point, or the best *N* number of good points.

Input

The following elements are required to evaluate points:

- List of candidate points from the point generator
- Point evaluation criteria:
  - Boolean – Condition criteria used to include or exclude a point independently of any other points
  - Weight – Criteria that, combined, give a measure of fitness relative to other points (those included by the boolean criteria)

Processing

The primary goal is to find an adequately good point as quickly as possible. Often, "adequately good" also means "the best", but there is a lot of potential for optimization if a user-specified degree of uncertainty is allowed.

The order of evaluation has a non-trivial and crucial impact on query efficiency. As a result, evaluation uses the following strategy to minimize the number of expensive operations:

1. Cheap booleans, with an expense on the order of one function call or some vector arithmetic. These allow the system to completely discount many points without significant cost. For example: *Is this point a primary or secondary hidespot? Is this point less than 5 meters from the target?*
2. Cheap weights, with an expense similar to cheap booleans. These allow the system to gauge the likelihood that a given point will eventually be the optimal choice; by focussing on points with a high likelihood, the number of expensive tests can be reduced. For example: *closeness_to_player * 3 + leftness * 2*.
3. Expensive booleans, approximately 100 times costlier. These are yes/no questions that will require expensive calculations to answer, but further eliminate points from contention. For example, the question *Is this point visible by the player?* requires an expensive ray test.
4. Expensive weights, with an expense similar to expensive booleans. These help to rank the remaining points. For example: *nearby_hidepoint_density * 2*

Algorithmic Details

It turns out that the system can go further with this by interleaving the final two steps and making evaluation order completely dynamic. Unlike conditions (booleans), weights don't explicitly discount points from further evaluation. However, by tracking the relative "fitness" of points during evaluation, we can still employ weights to dramatically reduce evaluations by employing two basic principles:

- Evaluate points in order of their maximum possible fitness, to fully evaluate the optimal point as quickly as possible.
- If, based on the initial weight evaluation, a point can be established as better than any other point, then immediately finish evaluating it against the remaining conditions. If the point passes all condition criteria, then it is the optimal point and no other points need to be evaluated. In addition, this point does not need to be evaluated on any remaining weights.

This implementation is based on a heap structure that orders points according to their maximum possible fitness and tracks the evaluation progress of each point separately. Each weight evaluation collapses some of the uncertainty around the point, adjusting both the minimum and maximum possible fitness. If the weight evaluation scored highly, the maximum will decrease a little and the minimum increase a lot; if it scored badly, the maximum will decrease a lot and the minimum increase a little.

In each iteration, the next most expensive evaluation is done on the point at the top of the heap, after which the point is re-sort into place if necessary. If all evaluations on a point have been completed and

it still has the maximum possible fitness, then it must be the optimal point. This approach tends towards evaluation of the optimal point with relatively few evaluations on all other points.

Output

The result of point generation evaluation is a single point or group of *N* number of points, and the opportunity to request all metadata leading to its selection. As a result, behaviors can adapt their style to reflect the nature the hidepoint received.

Integration with the Modular Behavior Tree System

From inside the Modular Behavior Tree (MBT), the **<QueryTPS>** node can be used to call pre-defined TPS queries by name. The **<QueryTPS>** node will return either success or failure.

The most common usage pattern involving the **<QueryTPS>** node is to use it in conjunction with the **<Move>** node inside a **<Sequence>** to determine the status of a specified position. The example below illustrates a call to a pre-defined TPS query called **SDKGrunt_TargetPositionOnNavMesh**, with the expected inputs. If the query succeeds, the AI agent will move to the queried position.

```
<Sequence>
    <QueryTPS name="SDKGrunt_TargetPositionOnNavMesh" register="RefPoint"/>
    <Move to="RefPoint" speed="Run" stance="Alerted" fireMode="Aim" avoidDangers="0"/>
</Sequence>
```

The definition of the pre-defined query **SDKGrunt_TargetPositionOnNavMesh** is as follows.

```
AI.RegisterTacticalPointQuery({
    Name = "SDKGrunt_TargetPositionOnNavMesh",
    {
        Generation =
        {
            pointsInNavigationMesh_around_attentionTarget = 20.0
        },
        Conditions =
        {
        },
        Weights =
        {
            distance_to_attentionTarget = -1.0
        },
    },
});
```

Future Plans and Possibilities

The following topics represent potential areas of development for TPS.

Higher-level environmental reasoning

One possible application of TPS: rather than simply using TPS to choose a point and move to it, there is the potential for some nice environmental deductions based on results.

For example: The player runs around a corner, followed by an AI puppet. When the AI puppet turns the corner, the player is no longer visible. The puppet queries TPS for places it would choose to hide from itself, with the following possible results.

- TPS returns that 1 hidepoint is much better than any other. This is because there's a single large box in the middle of an empty room. The AI puppet assumes the player is there and charges straight at the box, firing.

- TPS returns that there are several good hiding places. This is because there's a stand of good cover trees. All the hidepoints are stored in a group blackboard, and the AI puppet (or a group) can approach each spot in turn to discover the player.

This scenario is workable with some extra code, and much easier when built upon TPS.

Sampling methods

When generating points in the open, generate points in a grid or radially around objects and treat each point individually. This supports a basic sampling method. Where an area must be sampled, some kind of coherency in the evaluation functions can be assumed, and so could use some adaptive sampling approaches instead.

Dynamic cost evaluation

A crucial aspect of optimizing TPS involves adjusting the relative expense function of queries. The costs of evaluations will vary across platforms, levels, and even locations within levels, and will change over time as the code changes. It is critical to make sure that the evaluation order is correct, to prevent more expensive evaluations from being favored over cheaper ones. The need to profile the evaluation function in all these difference circumstances suggests an automatic profiling solution at run-time.

In addition, the relative weighting of weight criteria should also be considered; a cheap query may not be worth doing first if it only contributes 10% of the final fitness value, while an expensive query that contributes 90% may actually save many other evaluations.

Relaxing the optimality constraint

When evaluating points the maximum and minimum potential fitness is always known at every stage; this provides the error bounds, or a relative measure of uncertainty about the point.

It may make sense to relax the optimality constraint and accept a point when it becomes clear that no other point could be significantly better. For example, the minimum potential fitness of a point may be less than 5% lower than the maximum potential fitness of the next best point. This information could be used to stop evaluation early and yield a further performance saving.

Navigation Q & A

Big Triangles and Small Links Between Them

Q: I have created a big flat map, placed an AI agent on it, and generated AI navigation triangulation. I noticed that the AI agent doesn't always take the shortest straight path from point A to point B. Why?

A: To illuminate the issue, use the following tools:

- AI debug console variable `ai_DebugDraw` set to "74". This value draws the AI navigation graph. (Note: a value of 79 will run faster, but limits the result to the area close to the player (with 15 m).
- AI debug console variable `ai_DrawPath` set to "all". This variable draws AI agent paths, including links (the corridors between adjacent triangles).
- The **Ruler** tool in Editor, used to visualize paths. You don't even need actual AI agents on the map to run experiments. (Note: this tool is located between **Snap Angle** and **Select Object(s)**.)

The AI navigation triangulation is intended to be fast and have a small memory footprint. One of the decisions made in this regard was to use 16-bit signed integers to store corridor (or "link") radius measurements between two adjacent triangles. Using centimeters as the uint of measure, this means

that the maximum link radius is 32767 cm (327.67 m). When an AI agent moves to another triangle, it can only go through this corridor, which is naturally very narrow if the triangles are still very large. This problem does not exist for triangles with edges less than 2 * 327.67 = 655.34 m.

This problem can only appear in the very initial stages of map development. Every forbidden area, tree or other map irregularity makes triangulation more developed, which results in more triangles that are smaller in size. As a result, the problem goes away.

Path Following

Q: How does path following actually work? Where to start?

A: See the topic on Path Following (p. 32).

Auto-Disabling

Q: How do you keep patrols always active, regardless of their distance from the player?

A: See the topic on Auto-Disable (p. 35).

Path Following

This topic provides some high-level insight on how path following is done in Lumberyard. To illustrate some concepts, we'll use the relatively simplistic example of Racing HMMWVs, which is a good representation of classic path following as presented in many AI texts.

Path following with Racing HMMWVs adheres to the following sequence.

1. Get the closest (to the AI agent) point on path.
2. Get the path parameter of this point. Paths usually have some kind of parametrization, *t -> (x,y,z).*
3. Add a certain value, usually called a "lookahead", to this parameter.
4. Get the path point that corresponds to this new parameter. This is called the look-ahead position.
5. Use this point as the navigation target.
6. If the vehicle is stuck, beam it straight to the closest point on the path.

Goalop "Followpath"

Use the goalop *followpath* to instruct an AI agent to follow a path. You can observe this sequence in action by setting a breakpoint at the beginning of a call to COPFollowPath::Execute. In the call stack window in Visual Studio, you'll be able to see the update operations for all (active) AI agents being called as part of the AI system update procedure. This action in turn calls the execute operations of the currently active goalops being run by the AI.

COPFollowPath::Execute accomplishes the following tasks:

- Uses the goalop *pathfind* to find a path leading to the beginning of a path. Optionally, it finds a path to the closest point on a path using a parameter passed to the *followpath* goalop.
- Traces the path by following it using the goalop *trace*
- Listens for the signal "OnPathFollowingStuck" to make sure the AI agent isn't stuck

The goalops *pathfind* and *trace* are commonly used for navigational goalops, including *approach* and *stick*.

COPTrace::ExecuteTrace and COPTrace::Execute

COPTrace::ExecuteTrace is used to clean up path-following issues, including handling edge cases and smart objects. The core of this call is as follows:

```
IPathFollower* pPathFollower = gAIEnv.CVars.PredictivePathFollowing ? pPipeUser-
>GetPathFollower() : 0;
bTraceFinished = pPathFollower ? ExecutePathFollower(pPipeUser, bFullUpdate,
 pPathFollower) : Execute2D(pPipeUser, bFullUpdate);
```

COPTrace::Execute does the same work plus a bit more. For the AI following a path, when its lookahead position hits the end of the path, this operation sends the signal "OnEndWithinLookAheadDistance" to the AI. In the sample scenario, this allows our racing HMMWVs to start looking for a new path to follow while they're still moving along the current path. Normally AI agents stop moving when the path following process is completed. The following Lua script is also useful to maintain movement:

```
AI.SetContinuousMotion(vehicle.id, true);
```

COPTrace::Execute2D

This operation can be used as a fallback if an AI agent (CPipeUser, at least) doesn't have a path follower. COPTrace::Execute2D accomplishes the following tasks:

- Gets the lookahead path position and the path direction at this position.
- Executes a maneuver, if necessary. For example, it makes cars go backwards to make a U-turn.
- Considers a number of reasons to slow down, including:
  - The angle between current and desired (aforementioned path direction) directions.
  - The curvature of the path.
  - Approaching the end of the path.
  - Approaching the top of a hill.

It then sets members *fDesiredSpeed* and *vMoveDir* of the AI agent's SOBJECTSTATE structure, which are brought to the game code later. For an example of how this data can be used for actual steering, take a look at CVehicleMovementArcadeWheeled::ProcessAI.

Note that COPTrace::Execute2D is not the only operation that sets *vMoveDir*. For example, obstacle avoidance code can overwrite it.

Movement System

Key priorities for the AI Movement system include the following features.

- Robust and predictable. Navigation can be very unreliable, with no guarantee that a character will carry out the requested movement and end up at the desired destination. This is a very organic problem with no clear resolutions. The AI Movement system solves this by providing more explicit information about failure reasons.
- Central, clear ownership and easy debugging. Rather than having contextual movement information – style, destination, requester, etc. – tied to a specific goalop and getting lost when a behavior switch

occurs, Lumberyard maintains this information in a central location and separated from the goalop. In practice, a movement request can be sent from anywhere and the movement system handles it centrally. when the goalop requester is no longer interested, it simply cancels the request. This doesn't mean the character stops immediately and all information is lost, it just means that interest in the request has expired.

- Planning. In Lumberyard, logic is handled in blocks for ease of use and organization. Movement blocks are responsible for their own isolated tasks, such as FollowPath, LeaveCover and UseSmartObject. A collection of blocks in sequence make up a plan, which is produced by a controller with a string-pulled path as input. This types of organization helps clarify a larger picture about what is being processed right now and what is coming up.

  Note
  This system is still a work in progress, and it's design was focused on solving some critical problems with an existing code base. It may not be suitable for all game titles.

Using the Movement System

Using the movement system is pretty straightforward. Create a MovementRequest object with information about the destination, style and a callback. Queue it in MovementSystem and receive a MovementRequestID. Use this if you want to cancel the request. Then wait for MovementSystem to process to your request. Once your request is processed, you'll be notified via the callback.

Here's what's happening internally to process your request:

1. Once MovementSystem receives your request, it creates an internal representation of the character, called a MovementActor. This is a container for all internal states and the proxy to all external states/logic related to a character. It binds a MovementController to the actor. Currently there's only one controller available – GenericController, which is the result of what was done before. (The term "controller" is also used on the game side for a similar but different entity. These entities may be merged in the future, and multiple types of controllers added, such as for the Pinger, Scorcher, or BipedCoverUsed.)
2. MovementSystem informs the controller that there's a new request to start working on. GenericController kicks off the path finder.
3. Once the pathfinding result is in, the GenericController produces a plan that it starts to follow.
4. When the GenericController finishes the last block in the plan, it informs MovementSystem that the task is finished.
5. MovementSystem notifies the requester of success, and moves on to the next request.

Potential Improvements

The following areas of improvement or enhancement are under consideration:

- Change request processing. Currently there is a request queue, with movement requests processed one at a time, in FIFO order. Requests are immutable, so it's impossible to change a request once it's been queued; as a result, the only option is to cancel a request and queue a new one. These issues could be resolved by removing the request queue and allowing only one request at a time. If a request comes in while one is already being processed, interrupt the current one and report it.
- Validate a pipe user before proceeding with the update.
- When a UseSmartObject block detects that the exact positioning system fails to position a character at the start of a smart object, it reports this failure through the agent's bubble and in the log. It then resolves the problem by teleporting the character to the end of the smart object and proceeds to the next block in the plan.

- The GenericController is only allowed to start working on a new request while it is executing a FollowPath block. It then shaves off all subsequent blocks so that the actor doesn't find itself in the middle of a smart object when planning takes place. This could be improved by allowing the controller to produce a part of the plan, looking further ahead, and then patch it with the current plan.
- The plan isn't removed when a request is canceled. This is because a subsequent 'stop' or 'move' request should follow the cancellation. However, until this request has been received, the controller has no way to know what to do.
- The pathfinding request is being channeled through the pipe user, and the result is returned to the pipe user as well as stored in `m_path`. This path is then extracted by the movement controller. It would be better if the pathfinder could be employed directly by the movement controller and skip the pipe user as a middle layer.
- The movement controller code would fit better on the game side, since that's where the information about the characters should live. It could be merged with the movement transitions that are handled on the game side.
- Being able to pull out a movement request at any time makes the code slightly more complex, because we can't rely on that fact that the controller is always working on a request that still exists. It may be better to keep the request, flag it as abandoned and clear the callback.
- The code could be improved by separating planning and plan execution into two different code paths instead of one.

Auto-Disable

You can save CPU time by not updating distant AI agents. Use the auto-disable feature to controlled updates either on a per-AI basis or globally.

Global auto-disable

- To control auto-disable for all vehicles: use the console variable `v_autoDisable`.
- To control auto-disable for all AI agents: use the console variable `ai_UpdateAllAlways`.

Per-AI auto-disable

Per-AI auto-disable is controlled by the entity property AutoDisable. Refer to the Lumberyard User Guide for more details on AI and vehicle entities. You can also change this property (and behavior) at run time.

- C++: `pAIActorProxy->UpdateMeAlways(true);`
- Lua: `AI.AutoDisable(entity.id, 1);`
- In Flow Graph Editor: turn **AI:AutoDisable** on or off for each AI.

AI Scripting

This collection of topics describes how to handle some key AI capabilities using scripting.

This section includes the following topics:

- Communication System (p. 36)
- Factions (p. 42)
- Modular Behavior Tree (p. 43)
- Refpoints (p. 86)

- Signals (p. 88)

Communication System

AI communication is about playing sound/voice and/or animations at the right times in the course of the game.

Setting up communication for an AI agent requires the following steps:

- General set up:
  - Define communication channels. Channels are used to track the status of communication events for an AI.
  - Define communications. Communications detail specifically what activity should occur (and how) when the communication is called for. Communications are grouped into configurations.
  - Set up voice libraries. Voice libraries support localized dialogs, subtitles, and lip-syncing.
- Specify communication types for an AI using AI properties:
  - Point an AI's CommConfig property to a communication configuration, which contains the set of communications for that AI.
  - Point an AI's esVoice property to a voice library to use for that AI.
- Trigger a communication event:
  - Specify the name of a communication channel for the event.
  - Specify the name of a communication to fire.

Communications, channels, and voice libraries are defined in a set of XML files. At game start-up, the directory `Game/Scripts/AI/Communication` and all subfolders are scanned for XML files containing these configurations.

Defining Communication Channels

A communication channel determines whether an AI can play a communication at a given moment, depending on whether or not the communication channel is occupied. Channels are a self-contained concept, independent of other AI communication concepts. They have a sole purpose: to be in one of two possible states, "occupied" or "free".

AI communication channels are defined in an XML file stored in `Game/Scripts/AI/Communication`. The SDK includes a template channel configuration XML file, called `ChannelConfg.xml`. Communication channels are configured in a hierarchy of parent and child channels. The hierarchical structure determines how a channel's occupied status affects the status of other channels (for example, a parent of an occupied child channel).

Channel Elements & Attributes

Communication channels are defined in a `<ChannelConfig>` element with the following attributes:

name

Channel name.

priority

minSilence

Minimum time (in seconds) that the channel should remain occupied after a communication has been completed.

flushSilence

Time (in seconds) that the channel should remain occupied after it has been flushed. This value overrides the imposed silence time (**minSilence**) after playing a communication. If not specified, the value set for **minSilence** is used.

actorMinSilence

Minimum time (in seconds) to restrict AI agents from playing voice libraries after starting a communication.

ignoreActorSilence

Flag indicating that AI agent communication restrictions from the script should be ignored.

type

Type of communication channel. Valid values are "personal", "group" or "global".

Example

Game/Scripts/AI/Communication/ChannelConfig.xml

```
<Communications>
    <ChannelConfig>
        <Channel name="Global" minSilence="1.5" flushSilence="0.5" type="global">
            <Channel name="Group" minSilence="1.5" flushSilence="0.5" type="group">
                <Channel name="Search" minSilence="6.5" type="group"/>
                <Channel name="Reaction" priority="2" minSilence="2" flushSilence="0.5"
 type="group"/>
                <Channel name="Threat" priority="4" minSilence="0.5" flushSilence="0.5"
 type="group"/>
            </Channel>
            <Channel name="Personal" priority="1" minSilence="2" actorMinSilence="3"
 type="personal"/>
        </Channel>
    </ChannelConfig>
</Communications>
```

Configuring Communications for an AI

Communication configurations determine what communication activity AI agents can perform and how it will manifest. Communications for a particular type of AI are grouped into configurations. For example, your game might have both human and non-human AI agents, each with its own set of communication activities. In this scenario, you might group all the human communications into a configuration object named "human" while communications for non-humans might be grouped into a "non-human" configuration. For a particular AI, you'll specify the configuration to use with the AI's *CommConfig* property. With this configuration structure, you can define a communication (such as "surprise") differently in each configuration so that, when triggered, the communication activity fits the AI involved.

For each communication, you also have the option to define multiple variations of action and specify how the variations are used.

AI communication channels are defined in one or more XML files stored in `Game/Scripts/AI/Communication`. The SDK includes a template channel configuration XML file, called `BasicCommunications.xml`.

Communication Elements & Attributes

Communications are configured using the following elements and attributes:

Config

Communication configurations are grouped into `<Config>` elements and use the following attributes. Each configuration must contain at least one communication.

name

Configuration name, which can be referenced in the AI's CommConfig property.

Communication

A communication is defined in a `<Communication>` element with the following attributes. Each communication should contain at least one variation.

name

Communication name.

choiceMethod

Method to use when choosing a variation. Valid values include "Random", "Sequence", "RandomSequence" or "Match" (uses only the first variation).

responseName

responseChoiceMethod

Similar to **choiceMethod**.

forceAnimation

Boolean flag.

Variation

Each variation is defined in a `<Variation>` element with the following attributes.

animationName

Animation graph input value.

soundName

voiceName

lookAtTarget

Boolean flag indicating whether or not the AI should look at the target during the communication.

finishMethod

Method that determines when communication is finished, such as after the communication type has finished or after a time interval. Valid values include "animation", "sound", "voice", "timeout" or "all".

blocking

AI behavior to disable during communication. Valid values include "movement", "fire", "all", or "none".

animationType

Valid values include "signal" or "action".

timeout

Example

Game/Scripts/AI/Communication/BasicCommunications.xml

```
<Communications>
<!--sound event example-->
    <Config name="Welcome">
        <Communication name="comm_welcome" finishMethod="sound" blocking="none">
            <Variation soundName="sounds/dialog:dialog:welcome" />
        </Communication>
    </Config>
<!--example showing combined animation + sound event (needs state using action/signal in
 the animation graph)-->
    <Config name="Surprise">
        <Communication name="comm_anim" finishMethod="animation" blocking="all"
 forceAnimation="1">
            <Variation animationName="Surprise" soundName="sounds/
interface:player:heartbeat" />
        </Communication>
    </Config>
</Communications>
```

Setting Up Voice Libraries

To support localized dialogs, subtitles, and lip syncing, you need to set up voice libraries. Once set up, you can assign a voice library to an AI (or entity archetype) using the AI's *esVoice* property.

Voice libraries are defined in a set of XML Excel files stored in `GameSDK/Libs/Communication/Voice`. The SDK includes a template voice library file at `GameSDK/Libs/Communication/Voice/npc_01_example.xml`.

Each voice library must include the following information.

Language

Localization type for this library.

File Path

Location where the sound files for this library are stored.

Signal

Communication name associated with a sound file.

Sound File

File name of a sound file, listed by signal.

Example

Comment field used to describe or illustrate a sound file.

Example

GameSDK/Libs/Communication/Voice/npc_01_example.xml

Language	American English

File Path	languages/dialog/ai_npc_01/	
Signal	**Sound File**	**SDK NPC 01 Example**
see		
	see_player_00	i see you
	see_player_01	hey there you are
	see_player_02	hey i have been looking for you
pain		
	pain_01	ouch
	pain_02	ouch
	pain_03	ouch
death		
	death_01	arrrhh
	death_02	arrrhh
	death_03	arrrhh
alerted		
	alerted_00	watch_out
	alerted_01	be careful
	alerted_02	something there

Setting Communication for an AI

An AI's communication methods are set using the AI agents properties. You can set AI properties in several ways. For information about using the Lumberyard Editor to set AI properties, see "Using Database View to Set AI Communication" in the Lumberyard User Guide).

Set the following properties:

- *CommConfig* – Set this property to the name of the communication configuration you want this AI to use. Communication configurations are defined in XML files in `Game/Scripts/AI/Communication`, using `<Config>` elements.
- *esVoice* – Set this property to the name of the XML file containing the voice library you want this AI to use. Voice libraries are defined in XML files in `GameSDK/Libs/Communication/Voice`.

Turning Animation and Voice Off

Communication animation and/or voice can be turned off for an AI agent using the agent's Lua script (as in the example below) or the entity properties in Lumberyard Editor Editor.

Example

`Game/Scripts/Entities/AI/Shared/BasicAITable.lua`

```
Readability =
{
   bIgnoreAnimations = 0,
   bIgnoreVoice = 0,
},
```

Triggering a Communication Event

To trigger a communication event, use the goalop *communicate* with the following attributes. Note that communication animations are not played if the AI is currently playing a smart object action.

name

Name of the communication to trigger (sound, voice, and/or animation). Communication names are defined in an XML file referred to by the CommConfig property of this AI.

channel

Communication channel being used by this AI. An AI's communication channel is defined in an XML file in `Game/Scripts/AI/Communication`.

expirity (expiry)

Maximum allowable delay in triggering the communication when the communication channel is temporarily occupied. If a communication can't be triggered within this time period, it is discarded.

To trigger communications using flow graph logic, use the Flow Graph node **AI:Communication**.

Example

```
<GoalPipe name="Cover2_Communicate">
   <Communicate name="comm_welcome" channel="Search" expirity="0.5"/>
</GoalPipe>
```

Debugging

To get debug information on AI communication issues, use the following console variables (`ai_DebugDraw` should be set to "1"):

- ai_DebugDrawCommunication
- ai_DebugDrawCommunicationHistoryDepth
- ai_RecordCommunicationStats

Debug output is shown in the console as illustrated here:

```
Playing communication: comm_welcome[3007966447] as playID[84]
CommunicationPlayer::PlayState: All finished! commID[-1287000849]
CommunicationManager::OnCommunicationFinished: comm_welcome[3007966447] as playID[84]
CommunicationPlayer removed finished: comm_welcome[3007966447] as playID[84] with
 listener[20788600]
```

Troubleshooting

[Warning] Communicate(77) [Friendly.Norm_Rifle1] Communication failed to start

You may get this message or a similar one if your AI's behavior tree calls a communication but the communication configuration is not set up properly. In this example message, "77" refers to line 77 in your AI's behavior tree script (or goalop script). This line is probably communication trigger such as this:

```
<Communicate name="TargetSpottedWhileSearching" channel="Reaction" expirity="1.0"
 waitUntilFinished="0" />
```

Some things to check for::

- Does the specified communication name "TargetSpottedWhileSearching" exist in your communication configuration files (XML files located in `Game/Scripts/AI/Communication/`)?
- Check the *CommConfig* property for the AI. Is it set to the name of a `<Config>` element defined in your communication configuration files? If so, is the communication name "TargetSpottedWhileSearching" defined inside this `<Config>` element? This issue, calling communications that aren't configured for the AI is a common source of this error.
- Check the communication's variation definition. Does it point to a resource (animation, sound) that exists? If using a voice library, does it point to a valid voice library file name?

Factions

AI agents use factions to determine their behavior when encountering other AI agents. There are a base set of behaviors such as neutral, friendly and hostile. For example, when an AI in the "Grunt" faction encounters an AI in the "Players" faction, the encounter will be hostile. Players encountering "Civilians" will be friendly, etc.

To set up faction communications:

- Create an XML file that defines all the factions in your game and their reactions to each other (see the example). This file should be placed in `\Games\Scripts\AI\`. The SDK includes a template faction XML file, called `Factions.xml`.
- Set the Faction property for all of your AI agents to one of the defined factions. You can also set factions using Flow Graph

Example: Faction setup

```
Factions.xml

<Factions>
    <Faction name="Players">
        <Reaction faction="Grunts" reaction="hostile" />
        <Reaction faction="Civilians" reaction="friendly" />
        <Reaction faction="Assassins" reaction="hostile" />
    </Faction>
    <Faction name="Grunts">
        <Reaction faction="Players" reaction="hostile" />
        <Reaction faction="Civilians" reaction="neutral" />
        <Reaction faction="Assassins" reaction="hostile" />
    </Faction>
    <Faction name="Assassins">
        <Reaction faction="Players" reaction="hostile" />
        <Reaction faction="Civilians" reaction="hostile" />
        <Reaction faction="Grunts" reaction="hostile" />
    </Faction>
    <Faction name="HostileOnlyWithPlayers" default="neutral">
        <Reaction faction="Players" reaction="hostile" />
    </Faction>
    <Faction name="Civilians" default="neutral" />
```

```
    <Faction name="WildLife" default="neutral" />
</Factions>
```

Modular Behavior Tree

Modular behavior tree (MBT) is a collection of concepts for authoring behaviors for artificial intelligent (AI) agents in your game. Instead of writing complicated code in C++ or other general purpose programming language, MBT lets you describe AI behaviors at a high level without having to think about mechanics such as pointers, memory, and compilers. MBT concepts and implementation are optimized for rapid iteration and re-use.

Core Concepts

Conceptually, MBT is based on two key objects: the *node* and the *tree*.

Node

The node is the most fundamental concept; it is a building block that can be combined with others to build behaviors. A node consists of a block of code that represents a simple task. All nodes have the same interface: when processed, they carry out a task and either succeed or fail.

Nodes can be standalone or may have child nodes, which are processed as part of the parent node processing. When processed, the success of a parent node often (but not always) depends on the success of each child node.

Nodes follow several common patterns, such as action, composite, and decorator nodes. These common node patterns are more fully described in later in this topic.

Game developers can create the nodes needed for their game. In addition, Lumberyard provides a set of standard nodes for general use. These include nodes for tasks related to AI, animation, flying, and common game activities, as well as generic nodes useful when building behaviors, such as for timeouts and looping tasks. These provided nodes are documented in the Modular Behavior Tree Node Reference (p. 52).

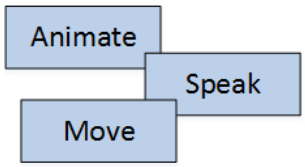
Tree

Behaviors are constructed by building trees of nodes, collections of individual tasks that, when positioned as a root with branches that extend out into leaves, define how an AI agent will behave in response to input.

Common Node Patterns

Action Nodes

An action node represents some sort of simple action. Action nodes might cause the AI agent to speak, play an animation, or move to a different location.



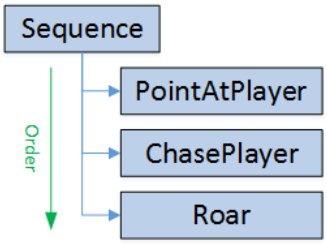
Composite Nodes

A composite node represents a series of actions to be performed in a certain order. Composite nodes consist of a parent node and two or more child nodes. Whether or not a child node is processed (and in

what order) can depend on the success or failure of previously processed nodes. Common composite patterns include sequential, selector, and parallel.

Sequential node

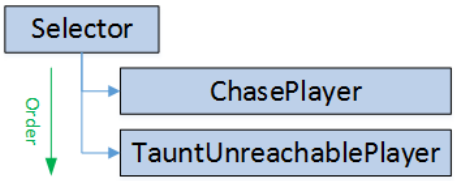
This composite pattern describes child nodes that are processed consecutively in a specified sequence. All child nodes are processed regardless of whether the previous child node succeeded or failed. For example, a sequential node might cause an AI monster to point at the player, roar, and then run toward the player. In this pattern, each child node in the sequence must succeed for the next child node to start processing; if any child node fails, the parent node immediately fails and processing is stopped.



Selector node

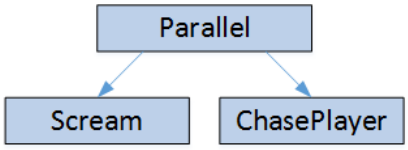
This composite pattern describes child nodes that are processed consecutively and in sequence only until one succeeds. As soon as one child node succeeds, the parent node succeeds immediately and stops processing child nodes. If all child nodes are attempted and all fail, the parent node fails. This pattern is useful for setting up AI agents to try multiple different tactics, or for creating fallback behaviors to handle unexpected outcomes.

Imagine, for example, that we want our AI monster to chase the player, but if it can't reach the player it should scream “Come and fight me, you coward!” To implement this scenario, a selector parent node is set up with two children, one for each possible action. The parent node first processes the “chase player” child node. If it succeeds, then the selector node stops there. However, if the “chase player node fails, then the parent node continues and processes the “taunt player” child node.



Parallel node

This composite pattern describes child nodes that are processed concurrently. In this scenario, Imagine we want our AI monster to scream and chase the player at the same time rather than one after the other.

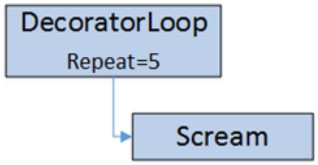


Decorator Nodes

A decorator node represents some sort of functionality that can be added to another node and behaves regardless of how the other node works or what it does. Common decorator functionality includes looping and limiting concurrent functionality.

Looping

Looping functionality can be used to process any other node multiple times. Rather than creating custom nodes every time you want to repeat a task, you can wrap any node in a parent loop decorator node. By setting a parameter for the loop node, you can dictate the number of times the child nodes will be processed. Each time the child node succeeds, the loop node count is updated and the child node is re-processed. Once the loop count meets the set parameter, the loop node succeeds.

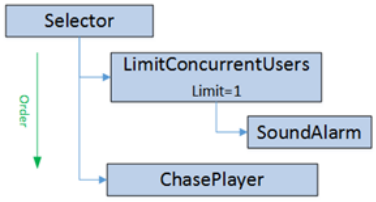


Limiting concurrent users

This functionality lets you specify how many users should be allowed to concurrently use a specified node. It is a good way to ensure variations in behavior among a group of AI agents. A typical scenario illustrating this function is as follows: The player is spotted by a group of three monsters. You want one monster to sound an alarm while the others chase the player.

Limiting concurrent users works with a selector node, which steps through a sequence of child nodes until one succeeds. By wrapping one of a selector node's child nodes in a limit decorator node, you can cause the child node to fail due to concurrent users, which in turn causes the selector node to move to the next child.

To handle the scenario described, the selector node would have two child nodes, "sound alarm" and "chase player". The "sound alarm" node is wrapped in a limit node, with the user limit set to 1. Monster #1 flows through the selector node to the limit node; as there is no one currently using the "sound alarm" node, the Monster #1 takes this action. The limit node records that one AI agent is processing the child node, so effectively locks the door to it. Monsters #2 and #3 also flow through the selector node to the limit node, but because the limit node has reached its limit of user, it reports a failure. Consequently, the selector node moves on to the next child node in the sequence, which is "chase player". So monsters #2 and #3 chase the player.



Describing Behavior Trees in XML

Behavior trees are described using XML markup language. Behavior trees are hot-loaded every time the user jumps into the game in the editor.

The following XML example describes the behavior tree for a group of monsters. In this example, only one monster at a time is allowed to chase the player. The remaining monsters stand around and taunt the player.

```
<BehaviorTree>
    <Root>
        <Selector>
            <LimitConcurrentUsers max="1">
                <ChasePlayer />
```

```
            </LimitConcurrentUsers>
            <TauntPlayer />
        </Selector>
    </Root>
</BehaviorTree>
```

C++ Implementation

You'll find all MBT code encapsulated in the BehaviorTree namespace.

Understanding the Memory Model

MBT has a relatively small memory footprint. It accomplishes this by (1) sharing immutable (read-only) data between instances of a tree, and (2) only allocating memory for things that are necessary to the current situation.

Memory is divided into two categories: configuration data and runtime data. In addition, MBT uses smart pointers.

Configuration data

When a behavior tree such as the following example is loaded, a behavior tree template is created that holds all the configuration data shown in the example. This includes a sequence node with four children: two communicate nodes, an animate node, and a wait node. The configuration data is the animation name, duration, etc., and this data never changes.

```
<Sequence>
    <Communicate name="Hello" />
    <Animate name="LookAround" />
    <Wait duration="2.0" />
    <Communicate name="WeShouldGetSomeFood" />
</Sequence>
```

Memory for the configuration data is allocated from the level heap. When running the game through the launcher, this memory is freed on level unload; alternatively, it is freed when the player exits game mode and returns to edit mode in Lumberyard Editor.

Runtime data

When spawning an AI agent using a behavior tree, a behavior tree Instance is created and associated with the agent. The instance points to the behavior tree template for the standard configuration data, which means that the instance contains only instance-specific data such as variables and timestamps.

When the tree instance is accessed for the AI agent, it begins by executing the Sequence node. If the core system detects that this is the first time the behavior has been run for this AI agent, it allocates a runtime data object specifically for this node and agent. This means that every AI agent gets its own runtime data object when executing a behavior tree node. The runtime data object persists as long as the AI agent is executing a node (this can be several frames) but is freed when the AI agent leaves a node.

Memory for runtime data is allocated from a bucket allocator. This design minimizes memory fragmentation, which is caused by the fact that runtime data is usually just a few bytes and is frequently allocated and freed. The bucket allocator is cleaned up on level unload.

Smart pointers

MBT uses Boost smart pointers to pass around data safely and avoid raw pointers as much as possible. Memory management is taken care of by the core system. (While there are circumstances in which a *unique_ptr* from C++11 would work well, Lumberyard uses Boost's *shared_ptr* for compatibility reasons.)

Implementing an MBT Node

To implement a new MBT node in C++, you'll need to do the following tasks:

- Create the node
- Expose the node to the node factory
- Set up error reporting for the node

Creating a node

The following code example illustrates a programmatic way to create a behavior tree node. When naming new nodes, refer to Recommended Naming Practices (p. 52).

```
#include <BehaviorTree/Node.h>

class MyNode : public BehaviorTree::Node
{
    typedef BehaviorTree::Node BaseClass;

    public:
    // Every instance of a node in a tree for an AI agent will have a
    // runtime data object. This data persists from when the node
    // is visited until it is left.
    //
    // If this struct is left out, the code won't compile.
    // This would contain variables like 'bestPostureID', 'shotsFired' etc.
    struct RuntimeData
    {
    };

    MyNode() : m_speed(0.0f)
    {
    }

    // This is where you'll load the configuration data from the XML file
    // into members of the node. They can only be written to during the loading phase
    // and are conceptually immutable (read-only) once the game is running.
    virtual LoadResult LoadFromXml(const XmlNodeRef& xml, const LoadContext& context)
    {
        if (BaseClass::LoadFromXml(xml, context) == LoadFailure)
            return LoadFailure;
        xml->getAttr("speed", m_speed);
        return LoadSuccess;
    }

protected:
    // Called right before the first update
    virtual void OnInitialize(const UpdateContext& context)
    {
        BaseClass::OnInitialize(context);

        // Optional: access runtime data like this
        RuntimeData& runtimeData = GetRuntimeData<RuntimeData>(context);
    }

    // Called when the node is terminated
    virtual void OnTerminate(const UpdateContext& context)
    {
        BaseClass::OnTerminate(context);

        // Optional: access runtime data like this
        RuntimeData& runtimeData = GetRuntimeData<RuntimeData>(context);
```

```
    }

    virtual Status Update(const UpdateContext& context)
    {
        // Perform your update code and report back whether the
        // node succeeded, failed or is running and needs more
        // time to carry out its task.

        // Optional: access runtime data like this
        RuntimeData& runtimeData = GetRuntimeData<RuntimeData>(context);
        return Success;
    }

    // Handle any incoming events sent to this node
    virtual void HandleEvent(const EventContext& context, const Event& event)
    {
        // Optional: access runtime data like this
        RuntimeData& runtimeData = GetRuntimeData<RuntimeData>(context);
    }

private:
    // Store any configuration data for the node right here.
    // This would be immutable things like 'maxSpeed', 'duration',
    // 'threshold', 'impulsePower', 'soundName', etc.
    float m_speed;
};

// Generate an object specialized to create a node of your type upon
// request by the node factory. The macro drops a global variable here.
GenerateBehaviorTreeNodeCreator(MyNode);
```

Exposing a node

To use the newly created node, you'll need to expose it to the node factory, as shown in the following code snippet.

```
BehaviorTree::INodeFactory& factory = gEnv->pAISystem->GetIBehaviorTreeManager()-
>GetNodeFactory();
ExposeBehaviorTreeNodeToFactory(factory, MyNode);
```

Setting up error reporting

Use the class `ErrorReporter` to report errors and warnings in the new node. It will let you log a printf-formatted message and automatically include any available information about the node, such as XML line number, tree name, and node type.

```
ErrorReporter(*this, context).LogError("Failed to compile Lua code '%s'", code.c_str());
```

Variables

Variables are statically declared in XML, with information about how they will change in response to signals from AI agents (named text messages within the AI system).

The following code snippet illustrates the use of variables to receive input from the AI system. In this example, the AI agent takes action based on whether or not it can "see" the target.

```
<BehaviorTree>
    <Variables>
        <Variable name="TargetVisible" />
    </Variables>
```

```
    <SignalVariables>
        <Signal name="OnEnemySeen" variable="TargetVisible" value="true" />
        <Signal name="OnLostSightOfTarget" variable="TargetVisible" value="false" />
    </SignalVariables>
    <Root>
        <Selector>
            <IfCondition condition=”TargetVisible”>
                <Move to=”Target” />
            </IfCondition>
            <Animate name=”LookAroundForTarget” />
        </Selector>
    </Root>
</BehaviorTree>
```

Lua Scripting

Lua code can be embedded in a behavior tree and executed along with the tree nodes. This is useful for running fire-and-forget code or for controlling the flow in a tree. It's useful for prototyping or extending functionality without having to create new nodes.

The code is compiled once when the level is loaded in pure game to reduce fragmentation. Only code for behavior trees that are actually used in that level will be compiled.

All Lua nodes provide access to the *entity* variable.

- `ExecuteLua` runs a bit of Lua code. It always succeeds.

```
<ExecuteLua code=”DoSomething()” />
```

- `LuaWrapper` inserts a bit of Lua code before and after running child node. The post-node code is run regardless of whether the child node succeeded or failed.

```
<LuaWrapper onEnter=”StartParticleEffect()” onExit=”StopParticleEffect()”>
    <Move to=”Cover” />
</LuaWrapper>
```

- `LuaGate` uses a bit of Lua code to control whether or not a child node should be run. If the Lua code returns true, the child node is run and `LuaGate` returns the status of the child node (success or failure). If the code returns false or fails to execute, the child node is not run, and `LuaGate` returns failure.

```
<LuaGate code=”return IsAppleGreen()”>
    <EatApple />
</LuaGate>
```

- `AssertLua` lets you make a statement. If the statement is true, the node succeeds; if it's false the node fails.

```
<Sequence>
    <AssertLua code=”return entity.someCounter == 75” />
    <AssertCondition condition=”TargetVisible” />
    <Move to=”Target” />
</Sequence>
```

Timestamps

A timestamp identifies a point in time when an event happened. A lot of AI behavior depends on tracking the timestamp of certain events and measuring the amount of time from those points. For example, it

can be useful to tie behavior to how long it's been since the AI agent was last shot at or hit, when it last saw the player, or how long it's been since moving to the current cover location.

Timestamps can be declared as mutually exclusive, that is, both timestamps can't have a value at the same time. For instance, `TargetSpotted` and `TargetLost` can both have a value because the AI agent can't see a player and at the same time consider them lost. With exclusive timestamps, when one timestamp has a value written to it, the other timestamp is automatically cleared.

The following code snippet illustrates the use of timestamps.

```
<BehaviorTree>
    <Timestamps>
        <Timestamp name="TargetSpotted" setOnEvent="OnEnemySeen" />
        <Timestamp name="ReceivedDamage" setOnEvent="OnEnemyDamage" />
        <Timestamp name="GroupMemberDied" setOnEvent="GroupMemberDied" />
    </Timestamps>
    <Root>
        <Sequence>
            <WaitUntilTime since=”ReceivedDamage” isMoreThan=”5” orNeverBeenSet=”1” />
            <Selector>
                <IfTime since="GroupMemberDied" isLessThan=”10”>
                    <MoveCautiouslyTowardsTarget />
                </IfTime>
                <MoveConfidentallyTowardsTarget />
            </Selector>
        </Sequence>
    </Root>
</BehaviorTree>
```

Events

Communication with AI agents is done using AI signals, which essentially are named text messages. Signals such as OnBulletRain and OnEnemySeen communicate a particular event, which, when broadcast to other AI agents, can be reacted to based on each AI agent's behavior tree. This design allows AI behavior to remain only loose coupled with AI signals. AI Signals are picked up and converted to MBT events, then dispatched to the root node, which passes them along down the running nodes in the tree.

```
<Sequence>
    <WaitForEvent name=”OnEnemySeen” />
    <Communicate name=”ThereHeIs” />
</Sequence>
```

Debugging and Tree Visualization

This section provides help with debugging behavior trees by providing a tree visualization view during debugging. This view allows you to track an AI agent's progress through the tree as the game progresses.

"Slashing" Agents

This feature allows you to view the behavior tree for a specific AI agent in DebugDraw. To enable this feature:

1. Set ai_DebugDraw to 0 or 1 (default is -1).
2. Select the AI agent you want to view a behavior tree for:
   - Place the selected AI agent in the center of the camera view and press the numpad "/" key.
   - Call "ai_DebugAgent closest" to select the agent closest to the camera.
   - Call "ai_DebugAgent centerview" to select the agent closest to the center of the camera view (same as slash).

- Call "ai_DebugAgent <AgentName>" to select a specific agent by its name.
- Call "ai_DebugAgent" without a parameter to remove the tree visualization.

The tree visualization displays the AI agent's name at the top of the screen and identifies the agent on the screen with a small green dot. Tree nodes are displayed and color coded as follows, with line numbers from the XML file shown on the left.

- White – nodes with custom data
- Blue – leaf nodes, which often carry special weight when debugging
- Gray – all other nodes

Adding Custom Debug Text

Tree visualization supports custom node information. This allows you to get a more in-depth view of the currently running parts of a behavior tree. For example, you can see the name of the event that the `WaitForEvent` node is waiting for, or how much longer `Timeout` is going to run before it times out.

To use this feature, override `GetDebugTextForVisualizer`, as follows.

```
#ifdef STORE_INFORMATION_FOR_BEHAVIOR_TREE_VISUALIZER
virtual void GetDebugTextForVisualizer(
        const UpdateContext& updateContext,
        stack_string& debugText) const
{
    debugText.Format("Speed %f", m_speed);
}
#endif
```

Logging and Tracing

Tracing log messages is a critical tool for diagnosing problems. Lumberyard provides native support for logging, as shown in the following code snippet.

```
<Sequence>
    <QueryTPS name="CoverFromTarget" _startLog="Finding cover" _failureLog="Failed to find
 cover" />
    <Move to="Cover" _startLog="Advancing" _failureLog="Failed to advance"
 _successLog="Advanced" />
</Sequence>
```

(The reserved attributes _startLog, _successLog, and _failureLog are automatically read in.)

Log messages are routed through an object deriving from the `BehaviorTree::ILogRouter` interface. This allows you to determine where the logging messages end up. For example, one option would be to route the info to a personal log and store a short history of log messages for each AI agent; with this approach, log messages can be displayed when debugging as part of an AI agent's tree visualization.

The AI Recorder also retains all log messages; use this tool to explore sequences of events.

Compiling with Debug Information

To compile a game with debug information, you need to define DEBUG_MODULAR_BEHAVIOR_TREE.

```
#if !defined(_RELEASE) && (defined(WIN32) || defined(WIN64))
# define DEBUG_MODULAR_BEHAVIOR_TREE
#endif
```

Viewing Completed Trees

When a behavior tree finishes executing—either by failing or succeeding all the way through the root node, a notification is displayed in the console window along with a list of recently visited nodes and their line numbers.

[Error] Modular Behavior Tree: The root node for entity 'HumanSoldier' FAILED. Rebooting the tree next frame. (124) Move. (122) Selector. (121) Sequence.

Note that in the example above the tree will be rebooted in the next frame. This suggests that the behavior tree was not designed to handle a failure at this point.

Recommended Naming Practices

The following suggestions help streamline code clarity and communication in a development team.

Naming Nodes

For action nodes, use names that identify the action the node will perform. These are usually action verbs.

Good

- Loop
- Animate
- LimitConcurrentUsers
- ExecuteLua
- Shoot
- AdjustCoverStance

Bad

- Fast
- PathPredictor
- Banana
- Script
- ActivationProcess

Naming Timestamps

Name timestamps based on the event they're related to. Because timestamps describe an event that has already happened, use the past tense (TargetSpotted, not TargetSpots).

- TargetSpotted
- ReceivedDamage
- GroupMemberDied

Modular Behavior Tree Node Reference

This section contains reference information on modular behavior tree (MBT) node types. MBT node types are organized here based on the system they are defined into.

It is possible to expose MBT nodes from anywhere in Lumberyard code. A node can have parameters that configure the behavior of its execution. If an invalid value is passed to the node, causing the node's parsing to fail, an error message is written to either `Editor.log` or `Game.log`.

Node Index

Generic Nodes (p. 54)

- Loop (p. 54)
- LoopUntilSuccess (p. 55)
- Parallel (p. 55)
- Selector (p. 56)
- Sequence (p. 56)
- StateMachine (p. 57)
- State & Transitions (p. 57)
- SuppressFailure (p. 58)
- Timeout (p. 58)
- Wait (p. 58)

AI Nodes (p. 59)

- AdjustCoverStance (p. 59)
- Aim (p. 59)
- AimAroundWhileUsingAMachingGun (p. 60)
- Animate (p. 60)
- AnimationTagWrapper (p. 61)
- AssertCondition (p. 61)
- AssertLua (p. 62)
- AssertTime (p. 62)
- Bubble (p. 62)
- CheckIfTargetCanBeReached (p. 63)
- ClearTargets (p. 63)
- Communicate (p. 64)
- ExecuteLua (p. 64)
- GroupScope (p. 65)
- IfCondition (p. 65)
- IfTime (p. 65)
- Log (p. 66)
- Look (p. 66)
- LuaGate (p. 67)
- LuaWrapper (p. 67)
- MonitorCondition (p. 68)
- Move (p. 68)
- Priority & Case (p. 69)
- PullDownThreatLevel (p. 70)
- QueryTPS (p. 70)
- RandomGate (p. 71)
- SendTransitionSignal (p. 71)

- SetAlertness (p. 71)
- Shoot (p. 72)
- ShootFromCover (p. 73)
- Signal (p. 74)
- SmartObjectStatesWrapper (p. 74)
- Stance (p. 74)
- StopMovement (p. 75)
- Teleport (p. 75)
- ThrowGrenade (p. 76)
- WaitUntilTime (p. 76)

CryAction Nodes (p. 77)

- AnimateFragment (p. 77)

Game Nodes (p. 77)

- InflateAgentCollisionRadiusUsingPhysicsTrick (p. 77)
- KeepTargetAtADistance (p. 78)
- Melee (p. 78)
- ScorcherDeploy (p. 80)
- SuppressHitReactions (p. 81)

Flying Nodes (p. 81)

- Hover (p. 81)
- FlyShoot (p. 82)
- WaitAlignedWithAttentionTarget (p. 82)
- Fly (p. 82)
- FlyForceAttentionTarget (p. 84)
- FlyAimAtCombatTarget (p. 84)
- HeavyShootMortar (p. 84)
- SquadScope (p. 85)
- SendSquadEvent (p. 85)
- IfSquadCount (p. 86)

Generic Nodes

These nodes provide the basic functionality of MBT.

Loop

Executes a single child node a specified number of times or until the child fails its execution.

Parameters

count

Maximum number of times the child node will be executed. If left blank, it is assumed to be infinite and the node will continue running until failure.

Success/Failure

The node SUCCEEDS if the maximum number of repetitions is reached. The node FAILS if execution of the child node FAILS.

Example

```
<Loop count="3">
    <SomeChildNode />
</Loop>
```

LoopUntilSuccess

Executes a child node a specified number of times or until the child node succeeds its execution.

Parameters

attemptCount

Maximum number of times the child node will be executed. If left blank or set to <=0, it is assumed to be infinite and the node will continue running until success.

Success/Failure

The node SUCCEEDS if the child SUCCEEDS. The node FAILS if the maximum amount of allowed attempts is reached.

Example

```
<LoopUntilSuccess attemptCount="5">
    <SomeChildNode />
</LoopUntilSuccess>
```

Parallel

Executes its child nodes in parallel.

Note

- A maximum number of 32 child nodes is allowed.
- When success and failure limits are reached at the same time, the node will succeed.

Parameters

failureMode

Method to use to evaluate when the node fails. Acceptable values include "any" or "all". Default: "any".

successMode

Method to use to evaluate when the node succeeds. Acceptable values include "any" or "all". Default: "all".

Success/Failure

When `successMode` is set to "all", the node SUCCEEDS if all the child nodes SUCCEEDS.

When `successMode` is set to "any", the node SUCCEEDS if any of the child nodes SUCCEED.

When `failureMode` is set to "any", the node FAILS if any of the child nodes FAILS.

When `failureMode` is set to "all", the node FAILS if all of the child nodes FAIL.

Example

```
<Parallel successMode="any" failureMode="all">
    <SomeChildNode1 />
    <SomeChildNode2 />
    <SomeChildNode3 />
</Parallel>
```

Selector

Executes its child nodes consecutively, one at a time, stopping at the first one that succeeds.

Parameters

None.

Success/Failure

The node executes the child nodes in sequential order and SUCCEEDS as soon as one of the child SUCCEEDS. Once the node succeeds, the child nodes that follow are not executed. The node FAILS if all the child nodes FAIL.

Example

```
<Selector>
    <SomeChildNode1 />
    <SomeChildNode2ToExecuteIfSomeChildNode1Fails />
    <SomeChildNode3ToExecuteIfSomeChildNode2Fails />
</Selector>
```

Sequence

Executes its child nodes one at a time in order.

> **Note**
> A maximum of 255 child nodes is allowed.

Parameters

None.

Success/Failure

The node SUCCEEDS if all the child nodes SUCCEED. The node FAILS if any of the child nodes FAILS.

Example

```
<Sequence>
    <SomeChildNode1 />
    <SomeChildNode2 />
    <SomeChildNode3 />
</Sequence>
```

StateMachine

Executes child nodes of type `State` one at a time. The first child node defined is the first to be executed. The current status of a StateMachine node is the same as that of the child that is currently being executed.

Parameters

None.

Success/Failure

The node SUCCEEDS if the current State child node SUCCEEDS. The node FAILS if the current State child node FAILS.

Example

```
<StateMachine>
    <State />
    <State name="State1" />
    <State name="State2" />
</StateMachine>
```

State & Transitions

Executes the content of its BehaviorTree node. This node can transition to another state (or itself). If a State node is instructed to transition into itself while running, it will first be terminated, re-initialized, and then updated again.

A State node has the following characteristics:

- Is a basic block of a StateMachine node.
- MUST have a BehaviorTree node.
- MAY have a Transitions element.

Transitions

Transitions elements are described inside a State node, and can contain the definitions of as many transitions as are needed. The transitions elements are not MBT nodes. If a transition specifies a destination state that doesn't exist, an error message will be displayed when parsing the MBT node.

Parameters

`<State />` elements must include the following parameters:

name

Name of the state. It must be unique within the scope of the StateMachine it is in.

`<Transition />` elements must include the following parameters:

onEvent

Name of the event that may cause the transition to happen. These events are of type AISignal.

to

Name of the state to transition to.

Success/Failure

The node SUCCEEDS if the content of the BehaviorTree node SUCCEEDS.

The node FAILS if the content of the BehaviorTree node FAILS.

Example

```
<State name="StateName">
 <Transitions>
  <Transition onEvent="EventOrTransitionSignalName" to="OtherStateName" />
 </Transitions>
 <BehaviorTree>
  <SomeChildNode />
 </BehaviorTree>
</State>
```

SuppressFailure

Owns and executes one child node. This node will succeed regardless of whether the child node succeeds.

Parameters

None.

Success/Failure

The node always SUCCEEDS once the child node has been executed.

Example

```
<SuppressFailure>
    <SomeChildThatCanFail />
</SuppressFailure>
```

Timeout

Fails once a certain amount of time has passed.

Parameters

duration

Amount of time (in seconds) before failure occurs.

Success/Failure

The node FAILS if it runs for more than the amount of time specified in the `duration` parameter.

Example

```
<Timeout duration=5" />
```

Wait

Succeeds once a certain amount of time has passed.

Parameters

duration

Amount of time (in seconds) before the request succeeds.

variation

Maximum additional amount of time that may be randomly added to the value of `duration`, in the range [0, `variation`]. Setting this value causes the wait time to have random variations between different executions of the node.

Success/Failure

The node SUCCEEDS once it has run the duration specified (plus random variation).

Example

```
<Wait duration="5" variation="1" />
```

AI Nodes

These nodes provide MBT functionality for the AI system.

AdjustCoverStance

Updates the AI agent's cover stance based on the maximum height at which its current cover is effective.

Parameters

duration

(Optional) Length of time (in seconds) the node will execute. Set to **`continuous`** to specify an unlimited time span.

variation

(Optional) Maximum additional time (in seconds) that may be randomly added to the value of `duration`, in the range [0, `variation`]. Setting this value causes the wait time to have random variations between different executions of the node.

Success/Failure

The node SUCCEEDS if execution of the child runs the length of the specified duration. The node FAILS if the child is not in cover.

Example

```
<AdjustCoverStance duration="5.0" variation="1.0"/>
```

Aim

Sets a location for the AI agent to aim at, and then clears the location when the node stops executing.

Parameters

at

Location to aim at. Allowed values include:

- RefPoint
- Target

angleThreshold

(Optional) Tolerance angle for aim accuracy.

durationOnceWithinThreshold

(Optional) Amount of time (in seconds) to continue aiming.

Success/Failure

The node SUCCEEDS after aiming at the desired location for the specified duration, if the location is not valid or if the timeout elapses.

Example

```
<Aim at="Target" durationOnceWithinThreshold="2.0" />
```

AimAroundWhileUsingAMachingGun

Updates the aim direction of the AI agent when using a mounted machine gun.

Parameters

maxAngleRange

(Optional) Maximum angle to deviate from the original direction.

minSecondsBeweenUpdates

(Optional) Minimum amount of delay (in seconds) between updates.

useReferencePointForInitialDirectionAndPivotPosition

Boolean.

Success/Failure

The node does not succeed or fail.

Example

```
<AimAroundWhileUsingAMachingGun minSecondsBeweenUpdates="2.5" maxAngleRange="30"
 useReferencePointForInitialDirectionAndPivotPosition="1"/>
```

Animate

Sets the AI agent to play an animation.

Parameters

name

Animation to be played.

urgent

(Optional) Boolean indicating whether or not to add the urgent flag to the animation.

loop

(Optional) Boolean indicating whether or not to add the loop flag to the animation.

setBodyDirectionTowardsAttentionTarget

(Optional) Boolean indicating whether or not to change the AI's body target direction to face the attention target.

Success/Failure

The node SUCCEEDS when the animation has finished playing, or if the animation failed to be initialized.

Example

```
<Animate name="LookAround" loop="1" />
```

AnimationTagWrapper

Adds an animation tag to the execution of a child node and clears it at the end.

Parameters

name

Animation tag to be set.

Success/Failure

The node returns the result of the execution of its child node.

Example

```
<AnimationTagWrapper name="ShootFromHip">
    <Shoot at="Target" stance="Stand" duration="5" fireMode="Burst" />
</AnimationTagWrapper>
```

AssertCondition

Checks whether or not a specified condition is satisfied.

Parameters

condition

Condition to be checked.

Success/Failure

The node SUCCEEDS if the condition is true, otherwise it FAILS.

Example

```
<AssertCondition condition="HasTarget" />
```

AssertLua

Executes a Lua script that returns true/false and translates the return value to success/failure. The result can be used to build preconditions in the MBT.

Parameters

code

Lua script to be executed.

Success/Failure

The node SUCCEEDS if the Lua script returns a value of true, otherwise it FAILS.

Example

```
<AssertLua code="return entity:IsClosestToTargetInGroup()" />
```

AssertTime

Checks whether or not a time condition is satisfied.

Parameters

since

Name of the time stamp to check for the condition.

isMoreThan

Condition statement used to test whether the time stamp is greater than a specified value. Cannot be used with the parameter `isLessThan`.

isLessThan

Condition statement used to test whether the time stamp is less than a specified value. Cannot be used with the parameter `isMoreThan`.

orNeverBeenSet

(Optional) Boolean indicating whether or not to set the node to succeed if the time stamp was never set.

Success/Failure

The node SUCCEEDS if the time condition is true, and FAILS if it is false. If the specified time stamp was not previously set, the node FAILS, unless the parameter `orNeverBeenSet` is true, in which case it SUCCEEDS.

Example

```
<AssertTime since="GroupLostSightOfTarget" isLessThan="10" orNeverBeenSet="1" />
```

Bubble

Displays a message in a speech bubble above the AI agent. See AI Bubbles System (p. 18).

Parameters

message

Message string to be shown in the speech bubble.

duration

Number of seconds to display the message. Default is 0.0.

balloon

Boolean indicating whether or not to display the message in a balloon above the AI agent. Default is true.

log

Boolean indicating whether or not to write the message to the general purpose log. Default is true.

Success/Failure

The node SUCCEEDS immediately after having queued the message to be displayed.

Example

```
<Bubble message="MessageToBeDisplayedAndOrLogged" duration="5.0" balloon="true"
 log="true" />
```

CheckIfTargetCanBeReached

Checks whether or not the AI agent's attention target can be reached.

Parameters

mode

Target to check for. Allowed values include:

- UseLiveTarget
- UseAttentionTarget

Success/Failure

The node SUCCEEDS if the target can be reached, otherwise it FAILS.

Example

```
<CheckIfTargetCanBeReached mode="UseLiveTarget" />
```

ClearTargets

Clears the AI agent's targets information.

Parameters

None.

Success/Failure

The node always SUCCEEDS.

Example

```
<ClearTargets />
```

Communicate

Sends a request to the communication manager to play one of the AI agent's communications. See Communication System (p. 36).

Parameters

name

The name of the communication to be played.

channel

The channel on which the communication is to be set.

waitUntilFinished

(Optional) Specifies if the execution should wait for the end of the communication before finishing.

timeout

(Optional) The threshold defining the maximum amount of seconds the node will wait.

expiry

(Optional) The amount of seconds the communication can wait for the channel to be clear.

minSilence

(Optional) The amount of seconds the channel will be silenced after the communication is played.

ignoreSound

(Optional) Sets the sound component of the communication to be ignored.

ignoreAnim

(Optional) Sets the animation component of the communication to be ignored.

Success/Failure

If the node is set to wait, the node SUCCEEDS when the communication is complete. Otherwise, it SUCCEEDS once the timeout elapses.

Example

```
<Communicate name="Advancing" channel="Tactic" expiry="1.0" waitUntilFinished="0" />
```

ExecuteLua

Executes a Lua script.

Parameters

code

Script to be executed.

Success/Failure

The node always SUCCEEDS.

Example

```
<ExecuteLua code="entity:SetEyeColor(entity.EyeColors.Relaxed)" />
```

GroupScope

Makes execution of a child node conditional on entering the AI agent in a group scope. Groups allow a limited number of concurrent users.

Parameters

name

Name of the group scope to enter.

allowedConcurrentUsers

(Optional) Maximum number of simultaneous users allowed in the specified group scope.

Success/Failure

The node FAILS if the AI agent cannot enter the group scope; otherwise, it returns the result of executing the child node.

Example

```
<GroupScope name="DeadBodyInvestigator" allowedConcurrentUsers="1">
    <SendTransitionSignal name="GoToPrepareToInvestigateDeadBody" />
</GroupScope>
```

IfCondition

Executes a child node if a specified condition is satisfied.

Parameters

condition

Condition statement to be checked.

Success/Failure

If the condition is satisfied, the node returns the result of executing the child node. If the condition is not satisfied, the node FAILS.

Example

```
<IfCondition condition="TargetVisible">
    <Communicate name="AttackNoise" channel="BattleChatter" expiry="2.0"
 waitUntilFinished="1" />
</IfCondition>
```

IfTime

Executes a child node if a time condition is satisfied.

Parameters

since

Name of the time stamp to check for the condition.

isMoreThan

Condition statement test whether the time stamp is greater than a specified value. Cannot be used with the parameter `isLessThan`.

isLessThan

Condition statement test whether the time stamp is less than a specified value. Cannot be used with the parameter `isMoreThan`.

orNeverBeenSet

(Optional) Boolean indicating whether or not to set the node to succeed if the time stamp was never set.

Success/Failure

If the time condition is true, the node returns the result of executing the child node. It FAILS if the time condition is false. If the specified time stamp was not previously set, the node FAILS, unless the parameter `orNeverBeenSet` is true, in which case it SUCCEEDS.

Example

```
<IfTime since="FragGrenadeThrownInGroup" isMoreThan="5.0" orNeverBeenSet="1">
    <ThrowGrenade type="frag" />
</IfTime>
```

Log

Adds a message to the AI agent's personal log.

Parameters

message

Message to be logged.

Success/Failure

The node always SUCCEEDS.

Example

```
<Log message="Investigating suspicious activity." />
```

Look

Adds a location for the AI agent to look at, and clears it when the node stops executing.

Parameters

at

Location to look at. Allowed values are:

- ClosestGroupMember
- RefPoint
- Target

Success/Failure

This node does not succeed or fail.

Example

```
<Look at="ClosestGroupMember" />
```

LuaGate

Executes a child node only if the result from running a Lua script is true.

Parameters

code

Lua script to be executed.

Success/Failure

The node SUCCEEDS if the result of the Lua script is true, and FAILS if the result is not true. On success, the node returns the result of executing the child node.

Example

```
<LuaGate code="return AI.GetGroupScopeUserCount(entity.id, 'DeadBodyInvestigator') == 0">
    <Animate name="AI_SearchLookAround" />
</LuaGate>
```

LuaWrapper

Runs a Lua script before and/or after the execution of a child node.

Parameters

onEnter

(Optional) Script to be executed at the start.

onExit

(Optional) Script to be executed at the end.

Success/Failure

The node returns the result of executing the child node.

Example

```
<LuaWrapper onEnter="entity:EnableSearchModule()" onExit="entity:DisableSearchModule()">
    <Animate name="AI_SearchLookAround" />
</LuaWrapper>
```

MonitorCondition

Continuously checks the state of a specified condition.

Parameters

condition

Specifies the condition to be checked.

Success/Failure

The node SUCCEEDS when the condition is satisfied.

Example

```
<MonitorCondition condition="TargetVisible" />
```

Move

Moves the AI agent from its current position to a specified destination. If the destination is a target, then the end position is updated if it is not reached when the target moves. See Movement System (p. 33).

Parameters

speed

Speed of movement. Allowed values include:

- Walk
- Run
- Sprint

stance

Body stance while moving. Allowed values include:

- Relaxed
- Alerted
- Stand (default)

bodyOrientation

Direction the AI agents body should face during the move. Allowed values include:

- FullyTowardsMovementDirection
- FullyTowardsAimOrLook
- HalfwayTowardsAimOrLook (default)

moveToCover

Boolean indicating whether or not the AI agent is moving into cover. Default is false.

turnTowardsMovementDirectionBeforeMovingx

Boolean indicating whether or not the AI agent should first turn to the direction of movement before actually moving. Default is false.

strafe

Boolean indicating whether or not the AI agent is allowed to strafe. Default is false.

glanceInMovementDirection

Boolean indicating whether or not the AI agent can glance in the direction of movement. If false, the AI agent will always look at its look-at target. Default is false.

to

Movement destination. Allowed values include:

- Target - Current attention target.
- Cover - Current cover position.
- RefPoint - Current reference position.
- LastOp - Position of the last successful position-related operation.

stopWithinDistance

Distance from the target that the AI agent can stop moving. Default is 0.0.

stopDistanceVariation

Maximum additional distance that may be randomly added to the value of `stopDistanceVariation`, in the range [0, `stopDistanceVariation`]. Setting this value causes the stop distance to vary randomly between different executions of the node. Default is 0.0.

fireMode

Firing style while moving. Allowed values are listed for the Shoot (p. 72) node.

avoidDangers

Boolean indicating whether or not the AI agent should avoid dangers while moving. Default is true.

avoidGroupMates

Boolean indicating whether or not the AI agent should avoid group mates while moving. Default is true.

considerActorsAsPathObstacles

Boolean indicating whether or not an AI agent's pathfinder should avoid actors on the path. Default is false.

lengthToTrimFromThePathEnd

Distance that should be trimmed from a pathfinder path. Use positive values to trim from the path end , or negative values to trim from the path start. Default is 0.0.

Success/Failure

The node SUCCEEDS if the destination is reached. The node FAILS if the destination is deemed unreachable.

Example

```
<Move to="Target" stance="Alerted" fireMode="Aim" speed="Run" stopWithinDistance="3" />
```

Priority & Case

Prioritizes to selects from a set of possible child nodes to execute. Within a `<Priority>` node, each child node is listed inside a `<Case>` node, which defines a condition statement. A child node is selected and executed based on (1) the first child to have its condition met, and (2) in the case of ties, the order the child nodes are listed in. All but the last child must have a condition statement; the last child listed is the default case, so it's condition must always be true.

Parameters

The `<Priority>` node has no parameters.

The `<Case>` node has the following parameters:

condition

Condition statement used to prioritize a child node.

Success/Failure

The node returns the result of the executed child node.

Example

```
<Priority>
    <Case condition="TargetInCloseRange and TargetVisible">
<Melee target="AttentionTarget" />
    </Case>
    <Case>
<Look at="Target" />
    </Case>
</Priority>
```

PullDownThreatLevel

Lower's the AI agent's perception of the target's threat.

Parameters

to

Success/Failure

The node always SUCCEEDS.

Example

```
<PullDownThreatLevel to="Suspect" />
```

QueryTPS

Performs a TPS query to find a tactical position for the AI agent, and waits for a result. See AI Tactical Point System (p. 20).

Parameters

name

Name of the TPS query to run.

register

Location to store the result of the TPS query. Allowed values include:

- RefPoint

- Cover (default)

Success/Failure

The node SUCCEEDS if the TPS returns a tactical position, or FAILS if it does not find a tactical position.

Example

```
<QueryTPS name="queryName" register="Cover" />
```

RandomGate

Executes a child node (or not) based on random chance.

Parameters

opensWithChance

Probability to use to determine whether the child node will be executed. Allowed values include floats 0.0 to 1.0.

Success/Failure

The node FAILS if the child node is not executed. If it is executed, the node SUCCEEDS AND returns the result of the execution of its child node.

Example

```
<RandomGate opensWithChance="0.5">
    <ThrowGrenade type="frag" />
</RandomGate>
```

SendTransitionSignal

Sends a signal, destined for a state machine node on the behavior tree, with the explicit intent of causing a change of state.

Parameters

name

Name of the signal to be sent.

Success/Failure

This node does not succeed or fail.

Example

```
<SendTransitionSignal name="LeaveSearch" />
```

SetAlertness

Sets the AI agent's alertness level.

Parameters

value

Alertness level. Allowed values include integers 0 to 2.

Success/Failure

The node always SUCCEEDS.

Example

```
<SetAlertness value="1" />
```

Shoot

Sets the AI agent to shoot at a target or location.

Parameters

duration

Length of time (in seconds) the AI agent should continue shooting.

at

Location to shoot at. Allowed values include:

- AttentionTarget
- ReferencePoint
- LocalSpacePosition

fireMode

Firing style. Allowed values include:

- Off - Do not fire (default).
- Burst - Fire in bursts at living targets only.
- Continuous - Fire continuously at living targets only.
- Forced - Fire continuously at any target.
- Aim - Aim only at any target.
- Secondary - Fire secondary weapon (grenades, etc.).
- SecondarySmoke - Fire smoke grenade.
- Melee - Melee.
- Kill - Shoot at the target without missing, regardless of the AI agent's aggression/attackRange/accuracy settings.
- BurstWhileMoving - Fire in bursts while moving and too far away from the target.
- PanicSpread - Fire randomly in the general direction of the target.
- BurstDrawFire - Fire in bursts in an attempt to draw enemy fire.
- MeleeForced - Melee without distance restrictions.
- BurstSwipe - Fire in burst aiming for a head shot.
- AimSweep - Maintain aim on the target but don't fire.
- BurstOnce - Fire a single burst.

stance

Body stance while shooting. Allowed values include:

- Relaxed
- Alerted
- Crouch
- Stand

position

(Required if the target is a local space position) Local space position to be used as the target.

stanceToUseIfSlopeIsTooSteep

(Optional) Alternative stance style if the slope exceeds a specified steepness. Allowed values are the same as for `stance`.

allowedSlopeNormalDeviationFromUpInDegrees

(Optional) Maximum allowed steepness (in degrees of inclination above horizontal) to set the primary stance. At positions that exceed this slope, the alternative stance is used.

aimObstructedTimeout

(Optional) Length of time (in seconds) the AI agent's aim can be obstructed before the node will fail.

Success/Failure

The node SUCCEEDS if it executes for the specified duration. The node FAILS if the aim is obstructed for longer than the specified timeout.

Example

```
<Shoot at="Target" stance="Crouch" fireMode="Burst" duration="5"
 allowedSlopeNormalDeviationFromUpInDegrees="30" stanceToUseIfSlopeIsTooSteep="Stand" />
```

ShootFromCover

Sets the AI agent to shoot at the target from cover and adjusts its stance accordingly.

Parameters

duration

Length of time (in seconds) the node should execute.

fireMode

Firing style. Allowed values are listed for the Shoot (p. 72) node.

aimObstructedTimeout

(Optional) Length of time (in seconds) the AI agent's aim can be obstructed before the node will fail.

Success/Failure

The node SUCCEEDS if it executes for the specified duration. The node FAILS if the AI agent is not in cover, if there's no shoot posture, or if the aim is obstructed for longer than the specified timeout.

Example

```
<ShootFromCover duration="10" fireMode="Burst" aimObstructedTimeout="3" />
```

Signal

Sends a signal to the AI system. See Signals (p. 88).

Parameters

name

Name of the signal to be sent.

filter

(Optional) Signal filter to use when sending the signal, which determines which AI agents will receive it.

Success/Failure

The node always SUCCEEDS.

Example

```
<Signal name="StartedJumpAttack" />
```

SmartObjectStatesWrapper

Sets the states of certain smart objects immediately before and/or after the execution of a child node.

Parameters

onEnter

(Optional) Smart object states to set at the start.

onExit

(Optional) Smart object states to set at the end.

Success/Failure

The node returns the result of executing the child node.

Example

```
<SmartObjectStatesWrapper onEnter="InSearch" onExit="-InSearch">
    <Animate name="LookAround" />
</SmartObjectStatesWrapper>
```

Stance

Sets the stance of the AI agent.

Parameters

name

Primary stance style. Allowed values include:

- Relaxed
- Alerted

- Crouch
- Stand

stanceToUseIfSlopeIsTooSteep

(Optional) Alternative stance style if the slope exceeds a specified steepness. Allowed values are the same as for `stance`.

allowedSlopeNormalDeviationFromUpInDegrees

(Optional) Maximum allowed steepness (in degrees of inclination above horizontal) to set the primary stance. At positions that exceed this slope, the alternative stance is used.

Success/Failure

The node always SUCCEEDS.

Example

```
<Stance name="Crouch" allowedSlopeNormalDeviationFromUpInDegrees="30"
 stanceToUseIfSlopeIsTooSteep="Stand" />
```

StopMovement

Sends a request to the Movement system to stop all movements. See Movement System (p. 33).

> **Note**
> This may not immediately stop the AI agent The Movement system may be dependent on animations and physics that dictate a 'natural' stop rather than an immediate cessation of movement.

Parameters

waitUntilStopped

Boolean indicating whether or not the node should wait for the Movement System to finish processing the request.

waitUntilIdleAnimation

Boolean indicating whether or not the node should wait until the Motion_Idle animation fragment begins running in Mannequin.

Success/Failure

The node SUCCEEDS if the stop request has been completed.

Example

```
<StopMovement waitUntilStopped="1" waitUntilIdleAnimation="0" />
```

Teleport

Moves the AI agent when both the destination point and source point are outside the camera view.

Parameters

None.

Success/Failure

The node always SUCCEEDS.

Example

```
<Teleport />
```

ThrowGrenade

Triggers the AI agent to attempt a grenade throw.

Parameters

timeout

Maximum length of time (in seconds) to wait for the grenade to be thrown.

type

Grenade type to throw. Allowed values include:

- emp
- frag
- smoke

Success/Failure

The node SUCCEEDS if a grenade is thrown before it times out, otherwise the node FAILS.

Example

```
<ThrowGrenade type="emp" timeout="3" />
```

WaitUntilTime

Executes until a time condition is satisfied.

Parameters

since

Name of the time stamp to check for the condition.

isMoreThan

Condition statement used to test whether the time stamp is greater than a specified value. Cannot be used with the parameter `isLessThan`.

isLessThan

Condition statement used to test whether the time stamp is less than a specified value. Cannot be used with the parameter `isMoreThan`.

succeedIfNeverBeenSet

(Optional) Boolean indicating whether or not to set the node to succeed if the time stamp was never set.

Success/Failure

The node SUCCEEDS if the time condition is true. If the specified time stamp was not previously set, the node FAILS, unless the parameter `succeedIfNeverBeenSet` is true, in which case it SUCCEEDS.

Example

```
<WaitUntilTime since="BeingShotAt" isMoreThan="7" />
```

CryAction Nodes

These nodes provide MBT functionality for CryAction features.

AnimateFragment

Plays a Mannequin animation fragment and waits until the animation finishes.

Parameters

name

Name of the animation to play.

Success/Failure

The node SUCCEEDS if the animation is correctly played or if no operation was needed. The node FAILS if an error occurs while trying to queue the animation request.

Example

```
<AnimateFragment name="SomeFragmentName" />
```

Game Nodes

These nodes offer game-specific MBT functionality. These allow a game with multiple character types to trigger specific logic and perform actions involving each type's peculiarities. Game-specific nodes not likely to be good for "general use" will probably need customization for each game.

Character types are defined in a Lua file, which contains a table of settings for game nodes.

InflateAgentCollisionRadiusUsingPhysicsTrick

Enlarges an AI agent's capsule radius for collisions with a player. This node employs a trick in the physics system inflate the capsule radius for agent-player collisions while leaving the radius unchanged for collisions between the agent and the world.

> **Note**
> This trick is entirely isolated within this node. The node does not clean up after itself, so the capsule remains inflated after it has been used.

This trick works as follows:

1. Sets the player dimensions with the agent-vs.-player collision radius. The physics system is multi-threaded, so there's a short wait while until the player dimensions are committed.
2. Periodically inspects the player dimensions to check that the agent-vs.-player collision radius has been successfully committed. This can sometimes fail to happen, such as when the AI agent is in a tight spot and can't inflate.

3. Once the agent-vs.-player radius has been committed, goes into the geometry and sets the capsule's radius in place, using the agent-vs.-world radius. This will not affect the agent-vs.-player dimensions.

Parameters

radiusForAgentVsPlayer

Size of capsule to use when calculating collisions between the AI agent and the player.

radiusForAgentVsWorld

Size of capsule to use when calculating collisions between the AI agent and the world.

Success/Failure

The node does not SUCCEED or FAIL. Once executed, it continues running until it is out of the scope of the executed nodes.

Example

```
<InflateAgentCollisionRadiusUsingPhysicsTrick radiusForAgentVsPlayer="1.0"
 radiusForAgentVsWorld="0.5" />
```

KeepTargetAtADistance

Keeps the live target at a distance by physically pushing the target away when it is within a specified distance. This node is useful when there is some sort of action close to the player and you want to avoid clipping through the camera. Use of this node is preferable over increasing the AI agent's capsule size, which will also affect how the character fits through tight passages. This node is generally used in parallel with other actions that need to be performed while the player cannot come too close to the AI agent; for example, when playing an animation on the spot that can move the AI agent without moving the locator, causing camera clipping.

Parameters

distance

Minimum distance allowed between the player and the AI agent.

impulsePower

Amount of impulse used to keep the player at least at the minimum distance.

Success/Failure

The node does not SUCCEED or FAIL. Once executed, it continues running until it is out of the scope of the executed nodes.

Example

```
<KeepTargetAtADistance distance="1.8" impulsePower="1.5" />
```

Melee

Triggers a melee attack against the AI agent's target. The melee attack is performed if the following condition are satisfied:

- If `failIfTargetNotInNavigationMesh` is set, the target must be on a valid walkable position. Some melee animations can move the character to a position outside the navigable area if trying to melee a target outside the navigation mesh.
- If the target is not within the threshold angle specified by the entity Lua value `melee.angleThreshold`.

Parameters

target

Target of the melee attack. This parameter could be set with the AI agent's AttentionTarget or a generic RefPoint.

cylinderRadius

Radius of the cylinder used for the collision check of the hit.

hitType

Type of hit that will be reported to the game rules. Default is CGameRules::EHitType::Melee.

failIfTargetNotInNavigationMesh

Boolean indicating whether or not the node should try to melee a target that is outside the navigation mesh.

materialEffect

Name of the material effect used when the melee attack hits the target.

Success/Failure

This node succeeds regardless of whether or not a melee attack is executed and, if it is, whether or not the attack damages the target. This is because a failure in this node is not important for behavior tree logic. If it's important for the game to react to this situation, a fail option can be added.

Example

```
<Melee target="AttentionTarget" cylinderRadius="1.5" hitType="hitTypeName"
 materialEffect="materialEffectName" />
```

Lua table settings

The Lua table `melee` contains the following settings:

```
melee =
{
    damage = 400,
    hitRange = 1.8,
    knockdownChance = 0.1,
    impulse = 600,
    angleThreshold = 180,
},
```

damage

Amount of damage a melee attack inflicts on the target.

hitRange

Height of the cylinder used to check whether or not the melee attack can hit the target.

knockdownChance

Probability that a successful melee attack knocks down the player.

impulse

Amount of impulse applied to the player in the case of a successful melee attack.

angleThreshold

Maximum angle allowed between the AI agent's direction of movement and the direction of a path between the AI agent and the target for melee attack to be attempted.

ScorcherDeploy

Manages how the Scorcher character type handles certain activity while deploying or undeploying as part of its shooting phase. This node relies on some external Lua scripts and various signals to work properly, but is useful in obfuscating some common functionality in the AI libraries.

Before and after the node runs, the following Lua functions are called: `EnterScorchTargetPhase()` and `LeaveScorchTargetPhase()`. When the node starts running, the "ScorcherScorch" animation tag is requested by Mannequin. When the node stops , if it stops normally, the "ScorcherNormal" tag is requested again. If it is terminated prematurely, it is up to the behavior tree script to define a proper exit strategy, such as requesting the "ScorcherTurtle" tag.

On requesting animation tags, the node waits for the following animation events to be received (this ensures that the transition blend animations are not interrupted):

1. "ScorcherDeployed" – when the scorcher is ready to start firing
2. "ScorcherUndeployed" – when the scorcher is again ready to walk around

The node encapsulates the following child nodes: `RunWhileDeploying` and `RunWhileDeployed`, each of which can contain exactly one child node.

RunWhileDeploying

Causes activity to happen while the Scorcher is in the process of deploying, that is, getting ready for an attack. As an example, this node might be used to control aiming before actually shooting.

The node will continue running until one of the following events occur, after which the node will be forcefully stopped:

- `ScorcherFriendlyFireWarningModule` sends one of these signals to the entity: "OnScorchAreaClear" or OnScorchAreaNotClearTimeOut"
- Mannequin animation sequence sends a "ScorcherDeployed" signal
- An internal timeout elapses

The node does not support any parameters. The node SUCCEEDS or FAILS depending on whether the child node succeeds or fails. The node is allowed to SUCCEED prematurely.

RunWhileDeployed

Controls actual aiming and firing during an attack. Duration and execution of the attack is controlled via this node.

The node does not support any parameters. The node SUCCEEDS or FAILS depending on whether the child node succeeds or fails. The node is allowed to SUCCEED prematurely. If the node SUCCEEDS, this triggers the parent node to start the undeployment sequence.

Parameters

maxDeployDuration

Length of time (in seconds) to allow the "RunWhileDeploying" child node to run. Default is 2.0.

Success/Failure

The node SUCCEEDS if the entire deploy and undeploy sequence is completed. The node FAILS if either the `RunWhileDeploying` or `RunWhileDeployed` nodes FAILED.

Example

```
<ScorcherDeploy maxDeployDuration="1.0">
    <RunWhileDeploying>
        <SomeChildNode>
    </RunWhileDeploying>
    <RunWhileDeployed>
        <SomeOtherChildNode>
    </RunWhileDeployed>
</ScorcherDeploy>
```

SuppressHitReactions

Enables or disables the Hit Reaction system for the AI agent.

Parameters

None.

Success/Failure

The node SUCCEEDS or FAILS based on success of failure of its child node.

Example

```
<SuppressHitReactions>
    <SomeChildNode />
</SuppressHitReactions>
```

Flying Nodes

These nodes provide MBT functionality related to flying vehicles.

Hover

Causes a flying AI agent to hover at its current position.

Parameters

None.

Success/Failure

The node does not SUCCEED or FAIL. Once executed, it continues running until forced to terminate.

Example

```
<Hover />
```

FlyShoot

Allows the AI agent to shoot at its attention target when possible from its current position.

If the AI agent's secondary weapon system is used, the node will only open fire if the weapons are able to hit close enough to the target. Otherwise normal firing rules are applied.

Parameters

useSecondaryWeapon

Boolean indicating whether or not the secondary weapon system (such as rocket launchers) should be used. Default is 0.

Success/Failure

The node does not SUCCEED or FAIL. Once executed, the AI agent continues to shoot until forced to terminate.

Example

```
<FlyShoot useSecondaryWeapon="1" />
```

WaitAlignedWithAttentionTarget

Waits until the AI agent is facing its attention target.

Parameters

toleranceDegrees

Maximum angle (in degrees) between the attention target and the forward direction of the AI agent to consider the AI agent to be "facing" the attention target. Allowed values include the range [0.0,180.0]. Default is 20.0.

Success/Failure

The node SUCCEEDS if the angle between the AI agent's forward direction and its attention target is within the allowed range. The node FAILS if the AI agent has no attention target.

Example

```
<WaitAlignedWithAttentionTarget toleranceDegrees="40" />
```

Fly

Allows an AI agent to fly around by following a path. Paths should be assigned to the AI agent using Flow Graph.

Parameters

desiredSpeed

Speed of movement (in meters per second) along the path to move along the path. Default is 15.0.

pathRadius

Radius of the path (in meters). While flying, the AI agent tries to stay within this distance from the path's line segments. Defaults is 1.0.

lookAheadDistance

Distance (in meters) to look forward along the path for 'attractor points' to fly to. Default is 3.0.

decelerateDistance

Distance (in meters) from the end of the path that the AI agent starts to decelerate. Default is 10.0.

maxStartDistanceAlongNonLoopingPath

Maximum distance (in meters) to look ahead for the closest point to link with another path. This parameter is used to link with non-looping paths; for example, it is useful to prevent the AI agent from snapping to the new path at a position that seems closer but is actually behind a wall after a U-turn. Defaults is 30.0.

loopAlongPath

Boolean indicating whether or not the AI agent should follow a path in an endless loop. Default is 0.

startPathFromClosestLocation

Boolean indicating at what point the AI agent should start following a path. Default is 0.

- 1 - at its closest position
- 2 - at the first path waypoint

pathEndDistance

Distance (in meters) from the end of the path that this node should start sending arrival notification events. Defaults is 1.0.

goToRefPoint

Boolean indicating whether or not the current reference point should be appended to the end of the path. Default is 0.

Success/Failure

The node SUCCEEDS if the AI agent reached the end of the path. The node FAILS if no valid path was assigned to the AI agent.

Example

```
<Fly lookaheadDistance="25.0" pathRadius="10.0" decelerateDistance="20.0"
 pathEndDistance="1" desiredSpeed="15" maxStartDistanceAlongNonLoopingPath="30"
 loopAlongPath="0" goToRefPoint="1" startPathFromClosestLocation="1" />
```

Lua table settings

The following properties in the AI agent's Lua script table can override the default XML tags. This will allow for changes to be made at run-time through (Flow Graph) scripting.

When	Lua variable	XML tag
Each node tick	`Helicopter_Speed`	`desiredSpeed`
Node activation	`Helicopter_Loop`	`loopAlongPath`
Node activation	`Helicopter_StartFromClosestLocation`	`startPathFromClosestLocation`

Upon arrival, the following events will be emitted:

- ArrivedCloseToPathEnd

- ArrivedAtPathEnd

FlyForceAttentionTarget

Keeps an attention target on a flying vehicle by force. The attention target is acquired during each tick of the node from the `Helicopter_ForcedTargetId` Lua script variable. When the node is deactivated, a ForceAttentionTargetFinished event is emitted.

Parameters

None.

Success/Failure

The node does not SUCCEED or FAIL. Once executed, it continues to force the attention target until deactivation.

Example

```
<FlyForceAttentionTarget />
```

FlyAimAtCombatTarget

Aims a flying AI agent at its target, taking into account special aiming adjustments for weapons.

Parameters

None.

Success/Failure

The node does not SUCCEED or FAIL. Once executed, it continues to force the AI agent to rotate its body towards the attention target until termination.

Example

```
<FlyAimAtCombatTarget />
```

HeavyShootMortar

Controls shooting the mortar (or Heavy X-Pak) weapon. It tries to simplify and centralize the pre-condition check and initialization of the weapon, plus re-selection of the primary weapon.

Parameters

to

(Optional) Shooting target. Allowed values include:

- Target (default)
- Refpoint

fireMode

(Optional) Type of firing. Allowed values include:

- Charge (default)
- BurstMortar

timeout

(Optional) Maximum time (in seconds) to continue shooting. Default is 5.0.

aimingTimeBeforeShooting

(Optional) Time (in seconds) to spend aiming before starting to shoot. Value must be longer than the global timeout. Default is 1.0.

minAllowedDistanceFromTarget

(Optional) Minimum distance (in meters) to the target required to start shooting. Default is 10.0.

Success/Failure

The node FAILS if the weapon is closer to the target than the value of `minAllowedDistanceFromTarget`. The node FAILS if there are obstructions less than two meters in front of the weapon; a cylinder check is done to avoid this. The node FAILS if the timeout is reached. The node SUCCEEDS if the shooting SUCCEEDS.

Example

```
<HeavyShootMortar to="RefPoint" fireMode="Charge" aimingTimeBeforeShooting="2"
 timeout="7" />
```

SquadScope

Makes execution of a child node conditional on adding the AI agent to a squad scope. Squads allow a limited number of concurrent users.

> **Note**
> The dynamic squad system uses the AI system's cluster detector. This tool is used with `AISquadManager` to group AI agents into dynamic squads.

Parameters

name

Name of the squad scope to enter.

allowedConcurrentUsers

(Optional) Maximum number of simultaneous users allowed in the specified squad scope. Default is 1.

Success/Failure

The node SUCCEEDS when the child SUCCEEDS. The node FAILS if the AI agent can't enter the squad scope or if the child FAILS.

Example

```
<SquadScope name="SomeScopeName" allowedConcurrentUsers="5">
    <SomeChildNode />
</SquadScope>
```

SendSquadEvent

Sends an event to squad members only.

> **Note**
> The dynamic squad system uses the AI system's cluster detector. This tool is used with `AISquadManager` to group AI agents into dynamic squads.

Parameters

name

Name of the event to be sent.

Success/Failure

The node always SUCCEEDS after sending the event.

Example

```
<SendSquadEvent name="SomeEventName" />
```

IfSquadCount

Makes execution of a child node conditional on whether or not the number of squad members meets a specified condition. Although all parameters are optional, at least one parameter must be used.

> **Note**
> The dynamic squad system uses the AI system's cluster detector. This tool is used with `AISquadManager` to group AI agents into dynamic squads.

Parameters

isGreaterThan

(Optional) Condition statement used to test whether the number of squad members exceeds a specified value.

isLesserThan

(Optional) Condition statement used to test whether the number of squad members is under a specified value.

equals

(Optional) Condition statement used to test whether the number of squad members exactly equals a specified value.

Success/Failure

The node SUCCEEDS if the number of squad members satisfies the specified condition statement, and FAILS if not.

Example

```
<IfSquadCount isGreaterThan="1">
    <SomeChildNode />
</IfSquadCount>
```

Refpoints

A refpoint, or reference point, is a special AI object used by goalpipes. It primarily specifies a position and, as needed, a direction. The following examples illustrate how refpoints are used.

Example 1: Updating a refpoint involving sub-goalpipes

In this example, a refpoint position is set, and a goalpipe is created containing three goalops: Locate, Stick, and Signal. Using the refpoint, Locate sets a value called LASTOP, which is used in Stick to pinpoint a destination.

Notice that the goalop Stick is defined as "+stick". This ensures that Stick is grouped with the previous goalop (Locate). As a result, if the interrupting goalpipe affects values that Stick depends on (such as LASTOP), it will return to the appropriate goalop to update the dependent values.

```
ACT_GOTO = function(self, entity, sender, data)
    if (data and data.point) then
        AI.SetRefPointPosition(entity.id, data.point);

        -- use dynamically created goal pipe to set approach distance
        g_StringTemp1 = "action_goto"..data.fValue;
        AI.CreateGoalPipe(g_StringTemp1);
        AI.PushGoal(g_StringTemp1, "locate", 0, "refpoint");
        AI.PushGoal(g_StringTemp1, "+stick", 1, data.point2.x, AILASTOPRES_USE, 1,
 data.fValue); -- noncontinuous stick
        AI.PushGoal(g_StringTemp1, "signal", 0, 1, "VEHICLE_GOTO_DONE",
 SIGNALFILTER_SENDER);
        entity:InsertSubpipe(AIGOALPIPE_SAMEPRIORITY, g_StringTemp1, nil, data.iValue);
    end
end,
```

Example 2: Using an AI anchor to set a refpoint

In this example, the Smart Object system spots a relevant AI anchor using `OnBiomassDetected`. This anchor is used to set both the position and direction of the refpoint. As a result, the AI agent walks to the refpoint, turns to the indicated direction, and then selects the next goalpipe.

```
OnBiomassDetected = function(self, entity, sender, data)
    entity:SetTargetBiomass(sender);
    entity:SelectPipe(0, "AlienTick_ReachBiomass");
end,
```

```
function AlienTick_x:SetTargetBiomass(biomass)
    self.AI.targetBiomassId = biomass.id;
    AI.SetRefPointPosition(self.id, biomass:GetWorldPos());
    AI.SetRefPointDirection(self.id, biomass:GetDirectionVector(1));
end
```

```
<GoalPipe name="AlienTick_ReachBiomass">
    <Speed id="Walk"/>
    <Locate name="refpoint"/>
    <Stick distance="0.3" useLastOp="true"/>
    <Signal name="OnBiomassReached"/>
</GoalPipe>
```

```
OnBiomassReached = function(self, entity)
    entity.actor:SetForcedLookDir(AI.GetRefPointDirection(entity.id));
    entity:SelectPipe(0, "AlienTick_CollectBiomass");
end,
```

Note
The tag <Group> was not used in this example because this particular goalpipe is not intended to be interrupted (which is not generally the case).

Signals

The Lumberyard AI system includes a fully customizable Signal system that enables AI entities to communicate with each other. Communication consists of signal events that can be sent by an AI agent to another single agent (including itself), or to a group of AI agents currently active in the game.

This topic describes how to send and receive signals between AI agents.

Signals Reference (p. 90)

Sending Signals

Signals are sent from an AI agent's behavior to one or more other AI agents using the method `AI:Signal()`.

```
AI:Signal(Signal_filter, signal_type, *MySignalName*, sender_entity_id);
```

Signal_filter

Group of AI agents to receive the signal. Allowed values include:

- 0 – AI agent specified with the entity_id parameter (usually but not always the sender itself).
- SIGNALFILTER_LASTOP – AI agent's last operation target (if it has one).
- SIGNALFILTER_TARGET – AI agent's current attention target.
- SIGNALFILTER_GROUPONLY – All AI agents in the sender's group (same group id) within communication range.
- SIGNALFILTER_SUPERGROUP – All AI agents in the sender's group (same group id) within the whole level.
- SIGNALFILTER_SPECIESONLY – All AI agents of the sender's species within communication range.
- SIGNALFILTER_SUPERSPECIES – All AI agents of the sender's species within the whole level.
- SIGNALFILTER_HALFOFGROUP – Half the AI agents in the sender's group, randomly selected.
- SIGNALFILTER_NEARESTGROUP – Nearest AI agent in the sender's group.
- SIGNALFILTER_NEARESTINCOMM – Nearest AI agent in the sender's group within communication range.
- SIGNALFILTER_ANYONEINCOMM – All AI agents within communication range.
- SIGNALID_READIBILITY – Special signal used to make the recipient perform a readability event (sound/animation).

signal_type

Type of signal, which determines how the recipient will process it. Allowed values include:

- 1 – Recipient processes signal only if it is enabled and not set to "ignorant" (see `AI:MakePuppetIgnorant`).
- 0 – The entity receiving the signal will process it if it's not set to ignorant.
- -1 – The entity receiving the signal will process it unconditionally.

MySignalName

The actual identifier of the signal. It can be any non-empty string; for the signal recipient, it must exist a function with the same name either in its current behavior, its default behavior or in the `Scripts/AI/Behaviors/Default.lua` script file in order to react to the received signal.

entity_id

The entity id of the signal's recipient. Usually you may want to put entity.id (or self.id if it's called from the entity and not from its behavior), to send the signal to the sender itself, but you can also put any other id there to send the signal to another entity.

Receiving Signals

The action to be performed once a signal is received is defined in a function like this:

```
MySignalName = function(self, entity, sender)
```

self

The recipient entity's behavior.

entity

The recipient entity.

sender

The signal's sender.

This function is actually a callback which, exactly like the system events, can be defined in the recipient entity's current behavior, the default idle behavior (if it's not present in current behavior) or in the `Scripts/AI/Behaviors/Default.lua` script file (if not present in the default idle behavior).

As for system events, a signal can be used also to make a character change its behavior; if we add a line like the following in a character file:

```
Behaviour1 = {
    OnEnemySeen    = *Behaviour1*,
    OnEnemyMemory = *Behaviour2*,
    &#8230;
    MySignalName   = *MyNewBehaviour*,
}
```

This means that if the character is currently in Behaviour1, and receives the signal MySignalName, after having executed the callback function above it will then switch its behavior to MyNewBehaviour.

Signal Example

A typical example is when a player's enemy spots the player: its OnEnemySeen system event is called, and let's suppose he wants to inform his mates (The guys with his same group id). In his default idle behavior (i.e., `CoverAttack.lua` if the character is Cover), we modify its OnEnemySeen event like this:

```
OnEnemySeen = function( self, entity, fDistance )
    -- called when the enemy sees a living enemy

    AI:Signal(SIGNALFILTER_GROUPONLY, 1, "ENEMY_SPOTTED",entity.id);
end,
```

Here we have defined a new signal called ENEMY_SPOTTED.

The next step is to define the callback function. Let's assume the other members in the group have the same character, we then add the callback function to the same idle behavior in which we have just modified OnEnemySeen.

```
ENEMY_SPOTTED = function (self, entity, sender)
    entity:Readability("FIRST_HOSTILE_CONTACT");
    entity:InsertSubpipe(0, "DRAW_GUN");
```

```
End,
```

This will make the guys (including the signal sender itself, who has the same behavior) change their animation and producing some kind of alert sound (readability), and then draw their gun. Notice that by modifying its idle behavior, we create a default callback which will be executed for any behavior the character is in. Later on, we may want to override this callback in other behaviors. For example, if we wanted the character to react differently whether it's in idle or attack behavior, we'll add the following callback function in the `CoverAttack.lua` file:

```
ENEMY_SPOTTED = function (self, entity, sender)
    entity:SelectPipe(0, "cover_pindown");
End,
```

Where "cover_pindown" is a goalpipe that makes the guy hide behind the nearest cover place to the target.

We can extend this to other characters: if there are group members with different characters (i.e. Scout, Rear etc) and we want them to react as well, we must add the ENEMY_SPOTTED callback also to their idle/attack behavior. Finally, we want the guys to switch their behavior from idle to attack if they see an enemy.

We'll then add the following line to the character (`Scripts/AI/Characters/Personalities/Cover.lua` in the example):

```
CoverIdle = {
    &#8230;
    ENEMY_SPOTTED = *CoverAttack*,
},
```

Behavior Inheritance

If specific signals are to be used in more than one behavior, there is an inheritance mechanism. Behavior classes can either directly inherit a more general implementation by keyword `Base = [CRYENGINE:ParentBehaviorName]` or indirectly, as a character's Idle behavior as well as the default behavior (defined in file DEFAULT.lua) are considered as fallback behaviors if a signal is not implemented in the current behavior.

Signals Reference

A typical signal handler looks something like this:

```
OnEnemySeen = function(self, entity, distance)
    -- called when the AI sees a living enemy
end,
```

Parameters self (behavior table) and entity (entity table) are passed to every signal. Additional parameters are specific to the signal being used.

See also: `\Game\Scripts\AI\Behaviors\Template.lua`.

Perception Signals

The following signals are sent to AI agents when perception types of their attention targets change.

Note that AITHREAT_SUSPECT < AITHREAT_INTERESTING < AITHREAT_THREATENING < AITHREAT_AGGRESSIVE.

No Target

Name	Parameters	Description
OnNoTarget		Attention target is lost

Sound

Sound heard (no visible target).

Name	Parameters	Description
OnSuspectedSoundHeard		Threat is AITHREAT_SUSPECT
OnInterestingSoundHeard		Threat is AITHREAT_INTERESTING
OnThreateningSoundHeard		Threat is AITHREAT_THREATENING
OnEnemyHeard		Threat is AITHREAT_AGGRESSIVE

Memory

The target is not visible and is in memory.

Name	Parameters	Description
OnEnemyMemory		Threat is AITHREAT_THREATENING
OnLostSightOfTarget		Threat is AITHREAT_AGGRESSIVE
OnMemoryMoved		Threat is AITHREAT_AGGRESSIVE and its location or owner has changed

Visual

The target is visible.

Name	Parameters	Description
OnSuspectedSeen		Threat is AITHREAT_SUSPECT
OnSomethingSeen		Threat is AITHREAT_INTERESTING
OnThreateningSeen		Threat is AITHREAT_THREATENING
OnEnemySeen	distance	Threat is AITHREAT_AGGRESSIVE
OnObjectSeen	distance, data	AI sees an object registered for this signal. data.iValue = AI object type (e.g. AIOBJECT_GRENADE or AIOBJECT_RPG)
OnExposedToExplosion	data	AI is affected by explosion at data.point
OnExplosionDanger		Destroyable object explodes

Awareness of Player

Name	Parameters	Description
OnPlayerLooking	sender, data	Player is looking at the AI for entity.Properties.awarenessOfPlayer seconds. data.fValue = player distance
OnPlayerSticking	sender	Player is staying close to the AI since <entity.Properties.awarenessOfPlayer> seconds
OnPlayerLookingAway	sender	Player has just stopped looking at the AI
OnPlayerGoingAway	sender	Player has just stopped staying close to the AI

Awareness of Attention Target

Name	Parameters	Description
OnTargetApproaching		
OnTargetFleeing		
OnNewAttentionTarget		
OnAttentionTargetThreatChanged		
OnNoTargetVisible		
OnNoTargetAwareness		
OnSeenByEnemy	sender	AI is seen by the enemy

Weapon Damage

Name	Parameters	Description
OnBulletRain	sender	Enemy is shooting
OnDamage	sender, data	AI was damaged by another friendly/unknown AI. data.id = damaging AI's entity id
OnEnemyDamage	sender, data	AI was damaged by an enemy AI. data.id = damaging enemy's entity id

Proximity

Name	Parameters	Description
OnCloseContact		enemy gets at a close distance to an AI (defined by Lua Property "damageRadius" of this AI)
OnCloseCollision		

Vehicles

Name	Parameters	Description
OnVehicleDanger	sender, data	vehicle is going towards the AI. data.point = vehicle movement direction, data.point2 = AI direction with respect to vehicle
OnEndVehicleDanger		
OnTargetTooClose	sender, data	attention target is too close for the current weapon range (it works only if AI is a vehicle)
OnTargetTooFar	sender, data	attention target is too close for the current weapon range (it works only if AI is a vehicle)
OnTargetDead		

User-defined

Custom signals can be sent when an attention target enters or leaves certain ranges. This is configured using the following Lua functions:

```
AI.ResetRanges(entityID);
AI.AddRange(entityID,range, enterSignal, leaveSignal);
AI.GetRangeState(entityID, rangeID);
AI.ChangeRange(entityID, rangeID, distance);
```

Weapon-Related Signals

Name	Parameters	Description
OnLowAmmo		
OnMeleeExecuted		
OnOutOfAmmo		
OnReload		AI goes into automatic reload after its clip is empty
OnReloadDone		reload is done
OnReloaded		

Navigation Signals

Pathfinding

Name	Parameters	Description
OnEndPathOffset	sender	AI has requested a path and the end of path is far from the desired destination
OnNoPathFound	sender	AI has requested a path which is not possible

Name	Parameters	Description
OnPathFindAtStart		
OnBackOffFailed	sender	AI tried to execute a "backoff" goal which failed
OnPathFound	sender	AI has requested a path and it's been computed successfully

Steering

Name	Parameters	Description
OnSteerFailed		

Smart Objects

Name	Parameters	Description
OnEnterNavSO		
OnLeaveNavSO		
OnUseSmartObject		

Navigation Shapes

Name	Parameters	Description
OnShapeEnabled		
OnShapeDisabled		

Tactics Signals

Tactical Point System

Name	Parameters	Description
OnTPSDestNotFound		
OnTPSDestFound		
OnTPSDestReached		

Cover

Name	Parameters	Description
OnHighCover		

Name	Parameters	Description
OnLowCover		
OnMovingToCover		
OnMovingInCover		
OnEnterCover		
OnLeaveCover		
OnCoverCompromised		

Groups Signals

Name	Parameters	Description
OnGroupChanged		
OnGroupMemberMutilated		
OnGroupMemberDiedNearest		

Formation

Name	Parameters	Description
OnNoFormationPoint	sender	AI couldn't find a formation point
OnFormationPointReached		
OnGetToFormationPointFailed		

Group Coordination

Group target is the most threatening target of the group.

Name	Parameters	Description
OnGroupTargetNone		
OnGroupTargetSound		
OnGroupTargetMemory		
OnGroupTargetVisual		
PerformingRole		

Flow Graph Signals

These are signals sent by corresponding Flow Graph nodes when they are activated.

Name	Parameters	Description
ACT_AIMAT		AI:AIShootAt
ACT_ALERTED		AI:AIAlertMe
ACT_ANIM		AI:AIAnim
ACT_ANIMEX		AI:AIAnimEx
ACT_CHASETARGET		Vehicle:ChaseTarget
ACT_DIALOG		AI:ReadabilityDialog (also sent by Dialog System)
ACT_DIALOG_OVER		Sent by Dialog System
ACT_DUMMY		
ACT_DROP_OBJECT		AI:AIDropObject
ACT_ENTERVEHICLE		Vehicle:Enter
ACT_EXECUTE		AI:AIExecute
ACT_EXITVEHICLE		Vehicle:Exit, Vehicle:Unload
ACT_FOLLOW		AI:AIFollow
ACT_FOLLOWPATH		AI:AIFollowPath, AI:AIFollowPathSpeedStance, Vehicle:FollowPath
ACT_GRAB_OBJECT		AI:AIGrabObject
ACT_GOTO		AI:AIGoto, AI:AIGotoSpeedStance, and the AI Debugger when the user clicks the middle mouse button.
ACT_JOINFORMATION		AI:AIFormationJoin
ACT_SHOOTAT		AI:AIShootAt
ACT_USEOBJECT		AI:AIUseObject
ACT_VEHICLESTICKPATH		Vehicle:StickPath
ACT_WEAPONDRAW		AI:AIWeaponDraw
ACT_WEAPONHOLSTER		AI:AIWeaponHolster
ACT_WEAPONSELECT		AI:AIWeaponSelect

Other Signals

Forced Execute

Name	Parameters	Description
OnForcedExecute		
OnForcedExecuteComplete		

Animation

Name	Parameters	Description
AnimationCanceled		

Game

Name	Parameters	Description
OnFallAndPlay		

Vehicle-related

Name	Parameters	Description
OnActorSitDown	Actor has entered a vehicle	

Squads

Name	Parameters	Description
OnSomebodyDied		
OnBodyFallSound		
OnBodyFallSound		
OnUnitDied		
OnSquadmateDied		
OnPlayerTeamKill		
OnUnitBusy		
OnPlayerDied		

Name	Parameters	Description
OnFriendInWay	sender	AI is trying to fire and another friendly AI is on his line of fire
URPRISE_ACTION		
OnActionDone	data	AI action of this agent was finished. data.ObjectName is the action name, data.iValue is 0 if action was cancelled or 1 if it was finished normally, data.id is the entity id of "the object" of the AI action

Animation

This section describes Lumberyard's Animation system. It includes discussions of key concepts and provides information on working with the system programmatically.

This section includes the following topics:

- Animation Overview (p. 98)
- Animation Events (p. 101)
- Limb IK Technical (p. 101)
- Animation Streaming (p. 102)
- Animation Debugging (p. 104)
- Fall and Play (p. 109)
- Time in the Animation System (p. 110)

Animation Overview

One of Lumberyard's goals is to push the boundaries of animations, which are all rendered in real time. Lumberyard provides tools to create both linear and interactive animations:

- Linear animation is the kind of animation seen in movies and cut-scenes, which play as a video.
- Interactive animation is used to convey AI and avatar (player) behavior, with sequences dependent on player choices in gameplay.

There is a big difference between how each type of animation is incorporated into a game, although this difference may not be obvious to the player, who simply sees characters moving on-screen. The key difference is in the decision-making process: who decides what a character on the screen is going to do next?

Linear Animations

In linear animation, the decision-making process happens inside the head of the people designing the animation. During this process, an animator has direct control over every single keyframe. They don't need to deal with collision detection, physics and pathfinding; characters only run into walls or collide with each other when the animator wants them to. AI behavior does not need to react to player behavior; the person who writes the storyboard decides how intelligent or stupid the characters are.

To show interactions between characters, you can put them in motion-capture suits and record their performances.

A linear animation sequence needs to show action from a single camera angle because the audience won't be moving during the animation; as a result, animators don't need to deal with transitions and motion combinations; they control every aspect of the motion clip. Because everything is fixed and predicable, it's possible to guarantee a consistent motion quality. Animators can always go back and adjust details in the scene, such as add or delete keyframes, adjust the lighting, or change the camera position.

The technical challenges with creating linear animation primarily involve rendering issues, such as not dropping the frame rate and ensuring that facial and body animations are in sync.

All linear animations in Lumberyard are created with **Track View** editor.

Interactive Animations

Creating interactive animations presents significantly tougher challenges. Animators and programmers do not have direct control over a character's on-screen movements. It is not always obvious where and how the decision-making process happens. It is usually a complex combination of AI systems, player input, and sometimes contextual behavior.

By definition, interactive animation is responsive. It looks visibly different depending on an individual user's input and adapts automatically to actions on the screen. Moving from linear animation to interactive animation requires more than just a set of small tweaks or a change in complexity—it requires a completely different technology under the hood. With interactive animation, an animator cannot precisely plan and model a character's behavior. Instead, animators and programmers develop a system that allows them to synthesize motion automatically and define rules for character behavior.

Automatic motion synthesis is a crucial feature in making animation more interactive. A system that synthesizes motion must be very flexible, because it is difficult to predict the sequence of actions that a character may take, and each action can start at any time.

Imagine, for example, a character moving through an outdoor environment. At a minimum, the designer needs to specify the style, speed, and direction of the character's locomotion. There should also be variations in motion while running uphill or downhill, leaning when running around corners or carrying objects of different sizes and weights—the character should run faster while carrying a pistol than when hefting a rocket launcher. It might also be necessary to interactively control emotional features such as happiness, anger, fear, and tiredness. Additionally, the character may need to perform multiple tasks simultaneously, such as walking in one direction, turning head and eyes to track a bird in another direction, and aiming a gun at a moving object in third direction. Providing unique animation assets for every possible combination and degree of freedom is nearly impossible and would involve an incredibly large amount of data. A mechanism for motion modifications is needed to keep the asset count as low as possible.

Developing such a system involves close collaboration and a tight feedback loop between programmers, animators, and designers. Problems with the behavior and locomotion systems (either responsiveness or motion quality) are usually addressed from several sides.

Interactive animation can be divided into two categories: **Avatar control** and **AI control**. In both cases, animators and programmers have indirect control over the actual behavior of a character in gameplay, because decision making for the character's next action happens elsewhere. Let's take a closer look at the situation in game environments.

Avatar control

An avatar character is controlled by the game player, whose decisions determine all of the avatar's actions. The locomotion system takes the player's input and translates it on the fly into skeleton

movements (using procedural and data-driven methods). With avatar control, high responsiveness is the top priority, while motion quality might be limited by the game rules. This means that many well-established rules for 'nice'-looking animations are in direct conflict with the responsiveness you need for certain types of gameplay.

The quality of animations as executed on the screen depends largely on the skills and decisions of each player controlling the character—they decide what the avatar will do next. Because a player's actions are unpredictable, motion planning based on predictions is not possible. Complex emotional control is not possible (and probably not needed). It's only possible on a raw level, such as soft punch versus an aggressive punch. However, it might be possible to let the player control the locomotion of the avatar, and to let the game code control the emotional behavior of the avatar by blending in "additive animations" based on the in-game situation.

In all these scenes, the player is controlling the character with a game pad. The character's presentation on the screen is using animation assets created by animators.

AI control

For AI characters, the decision-making process happens entirely inside the game code. Game developers design a system to generate behavior, which acts as an intermediary between the game creators and players. For the system to perform this task, it is necessary for game designers to explicitly specify behavioral decisions and parameters for AI characters, including a clear definition of the rules of movements for each character type. Interactive animation for AI characters is much harder to accomplish than animations for avatars, but at the same time it offers some (not always obvious) opportunities to improve motion quality. High responsiveness is still the primary goal but, because character choices happen inside the game code, it is possible in certain circumstances to predict a character's actions. If the AI system knows what the AI character wants to do next, then it is possible to incorporate this knowledge into motion planning. With good motion planning, interactive animation might be able to use more classical or 'nice' animation rules. As a result, AI control can have a somewhat higher motion quality than avatar control, though at the cost of having more complex technology under the hood.

The only source of uncertainty in such a prediction system is the player: the AI reacts to the player, and predicting the player's actions is impossible. As a result, it's nearly impossible to create the right assets for every in-game situation, and this in turn makes it impossible to guarantee a consistent motion quality. For an animator working on interactive animation, it can be a significant problem to have no direct control over the final animation—it's never clear when the work is complete. This is one reason why the linear animation in movies and cut-scenes look superior, and why interactive animations can be troublesome.

Lumberyard tackles the problem with interactive animation in multiple levels:

- In the low-level CryAnimation system library, the engine provides support for animation clips, parametrized animation, and procedural modification of poses. Animations can be sequenced together or layered on top of each other in a layered transition queue.
- In the high-level CryAction library, the CryMannequin system helps to manage the complexity of animation variations, transitions between animations, animations that are built up out of many others, sequencing of procedural code, links to game code, and so on.

Scripted Animations

Because interactive animation is much more difficult than linear animation, many games blur the line between cut-scenes and in-game actions by using interactive scripted sequences.

In this case, characters act on a predefined path. The quality of this kind of motion can be very high. Because it is not fully interactive, animators have more control over the entire sequence, a kind of manually designed motion planning. These are perfectly reasonable cheats to overcome hard-to-solve animation problems. It may be even possible to script the entire AI sequence to allow near-cut-scene

quality. The action feels interactive and looks absolutely cinematic, but it is actually more an illusion of interactivity.

In the game Crysis, Crytek designers made use of scripted animations in many scenes. In the "Sphere" cut-scene, the Hunter is shown walking uphill and downhill and stepping over obstacles. This is a scripted sequence where the assets were made for walking on flat ground, but Crytek used CCD-IK to adapt the character's legs to the uneven terrain. In the "Fleet" cut-scene with the Hunter on the carrier deck, the player can move around while the Hunter is fighting other non-playing characters.

Both scenes look and feel highly interactive but they are not. The Hunter doesn't respond to the player and the player cannot fight the Hunter. The scenes are fully linear and scripted, basically just animated background graphics. These sequences were created in **Track View** editor. Some of them used the Flow Graph Editor. When the cut-scene is over, the Hunter turns into an AI-controlled interactive character.

Animation Events

Animations in Lumberyard can be marked up to send custom events at a specific time in an animation. This markup is used for time-aligned blending; for example, to match footplants in animations. Another application of animation events is to spawn particle effects at the right moment.

These events can also be used by a variety of systems that need to receive information about when an animation has reached a certain point, such as in combination with a melee system.

Marking Up Animations with Events

Events for animations are stored in an XML file that is loaded when the character starts up. For this to happen automatically, the database must be included in the `chrparams` file.

Receiving Animation Events in the Game Code

Animation events are passed on to the game object once they have been triggered. The Actor and Player implementations both handle these animation events. See either `Actor.cpp` or `Player.cpp` for the function:

```
void AnimationEvent(ICharacterInstance *pCharacter, const AnimEventInstance &event)
```

Limb IK Technical

Lumberyard's animation system allows the setup of IK chains for characters.

When an IK chain is active, the system calculates the joint angles in the chain so that the end effector (typically a hand or foot) reaches the target position.

Setting Up

IK chains are defined in the `chrparams` file.

Using LimbIK from Code

To activate a Limb IK chain from outside the Animation system, use the function `SetHumanLimbIK`, accessible through the `ISkeletonPose` interface. The `SetHumanLimbIK` function needs to be called in each frame in which you want the IK chain to be active. The name of the Limb IK chain is defined in the `chrparams` file:

```
ISkeletonPose& skeletonPose = ...;
skeletonPose.SetHumanLimbIK(targetPositionWorldSpace, "RgtArm01");
```

Animation Streaming

Animation is very memory-intensive and tends to use a large amount of resources. Limited memory budgets, high numbers of animated joints, and requirements for high animation quality make it wasteful for a project to keep all animations constantly loaded in memory.

Lumberyard's animation system alleviates this issue by streaming in animation resources (file granularity level) when needed, and unloading them when not needed. Streaming of asset files is achieved by using the DGLINK Streaming System. Streaming assets in and out allows the system to keep only the needed resources in memory—which is done at the expense of complexity, as you must now plan how and when animation resources are used.

Animation Data

Animation data usage is divided into two main sections:

- The **header** section contains generic information for an animation (filename, duration, flags, etc).
- The **controller** section contains the animation curves. For each joint involved, this section contains information on all the position and orientation values that the joint needs in order to play that animation. Even when compressed, controller data can easily take up more than 95% of the total memory required for an animation.

Animation Header Data

Header data for animations is stored in `CAF` files and in the `animations.img` file.

`CAF` files contain the header information on a single animation, while `animations.img` contains header information for all animations in the build. The `animations.img` is obtained as a result of processing all the animations with the Resource Compiler.

The engine usually loads all the animation files' headers from the `animations.img` file instead of loading from individual files (reading the information from individual files can considerably slow down loading time).

Because of the extreme size difference between controllers and headers, Lumberyard streams only the controller data in and out of memory. The header data for all animations is kept at all times in memory, as it is practical to have that information available at all times.

> **Note**
> During development—for example, when working with local animation files—you must disable usage of `animations.img` and load the header information from individual `CAF` files instead. To do so, set the `ca_UseIMG_CAF` console variable to 0 before the engine starts.

Animation Controller Data

The controller data for animations is stored in CAF files or DBA files.

- `CAF` files contain controller information for a single animation.
- `DBA` files contain controller information for a group of animations.

When a DBA is loaded, controllers for all animations contained in that DBA are available until the DBA is unloaded. For this reason, it is useful to group animations that are used together in the same DBA. An extra benefit of putting similar animations together in a DBA is that equal controllers are only stored once. This reduces the memory usage of your animations.

Loading Controller Data

The animation system properly plays animations only when their controllers are in memory.

If controller data is not available when playback of an asset is requested, the animation system streams it in from disk. Streaming of controller data is performed asynchronously—the animation system does not wait until after asset playback is requested. This prevents stalling the system.

If high level systems fail to notify the animation system that they require controller data (see the preload functions section), the animation system does not know that an asset is required until it is requested to play. This is dangerously close to when the controller data is needed. If the controller data is not available in time, it typically leads to visual glitches, which can sometimes be observed, for example, only the first time an animation is played.

Therefore, it is important to have controller data streamed in before playback of an animation is requested. This minimizes undesired glitches that occur while waiting for animation streaming to end.

The amount of time required for streaming to complete depends on many factors, such as the current system load, streaming speed of the target system, size of the resource that needs to be loaded, and so on.

Unloading Controller Data

The animation system will not unload controller data that is currently in use.

It is possible to prevent unloading of animation data entirely by setting `ca_DisableAnimationUnloading` to 1.

Controllers in `CAF` files are unloaded after the system detects that they are no longer in use. To prevent controllers in CAF files from being unloaded, set `ca_UnloadAnimationCAF` to 0.

Controllers in DBA files remain in memory until a certain amount of time passes after the animations in them are used. However, if the DBA is locked, controllers are not unloaded until the lock status is set back to 0.

To change the time that the animation system waits to unload controllers in DBA files, use the following `cvars`:

- `ca_DBAUnloadUnregisterTime` – Timeout in seconds after the last usage of a controller and all animations using that DBA; when this timeout is reached, the DBA marks their controller data as 'unloaded'.
- `ca_DBAUnloadRemoveTime` – Timeout in seconds after the last usage of a controller in a DBA; when this timeout is reached, the DBA performs an actual unload from memory. This value should be greater than or equal to `ca_DBAUnloadUnregisterTime`.

The following section describes how to lock individual resources in memory to prevent the system from unloading them.

Preloading and Keeping Controllers in Memory

Preload functions are performed by high level systems or user code (usually game code), as these contain most of the information on when and how assets are accessed. For example, trackview looks a number of seconds ahead in the timeline for any animations that appear, and calls the preload functions.

Preloading Controllers in DBA files

To preload and trigger the streaming of a DBA file:

```
gEnv->pCharacterManager->DBA_PreLoad(dbaFilename, priority);
```

To trigger the streaming of a DBA file, and request a change to the locked state (which specifies whether it should be locked in memory):

```
gEnv->pCharacterManager->DBA_LockStatus(dbaFilename, lockStatus, priority);
```

To unload all controller data in a DBA from memory (unloads data only if none of the controllers are currently being used):

```
gEnv->pCharacterManager->DBA_Unload(dbaFilename);
```

Note
To make the system automatically load and lock a DBA file while a character is loaded, use the `flags="persistent"` in the `chrparams` file.

Preloading Controllers in CAF files

To increase the reference count of a CAF file:

```
gEnv->pCharacterManager->CAF_AddRef(lowercaseAnimationPathCRC);
```

Controllers for a CAF file start streaming in when its reference count goes from 0 to 1.

To decrease the reference count of a CAF file:

```
gEnv->pCharacterManager->CAF_Release(lowercaseAnimationPathCRC);
```

Controllers for a CAF file are unloaded by the animation system only after the reference count reaches 0 (the animation system, when playing a CAF file, also increases this reference count, so that an animation is not unloaded while in use).

To check whether the controllers for a CAF file are loaded:

```
gEnv->pCharacterManager->CAF_IsLoaded(lowercaseAnimationPathCRC);
```

To synchronously load the controllers for a CAF file:

```
gEnv->pCharacterManager->CAF_LoadSynchronously(lowercaseAnimationPathCRC);
```

Synchronously loading CAF assets is strongly discouraged unless absolutely necessary, as it will likely result in stalls.

Animation Debugging

Several tools are available for debugging animation issues.

Layered Transition Queue Debugging

You can enable on-screen debug information to see which animations are queued and playing, as well as information about the applied pose modifiers and IK.

Show Per Entity

To show the transition queue for all the character instances of a specified entity:

```
es_debuganim <entityname> [0 | 1]
```

<entityname>

Name of the entity to debug. In a single player game, the player is typically called "dude." Note that the GameSDK example player has both a first person and a third person character instance.

[0 | 1]

Specify 1 or no second parameter to turn it on for this specific entity. Specify 0 to turn it off.

Examples

To turn on debugging for a player with the entity name "dude":

```
es_debuganim dude 1
```

To turn off debugging for an entity called "npc_flanker_01":

```
es_debuganim npc_flanker_01 0
```

Show Per CharacterInstance

You can show the transition queue for all character instances or the ones that have a specific model name.

```
ca_debugtext [<modelname-substring> | 1 | 0]
```

<modelname-substring>

Shows information for all character instances whose modelname contains the specified string.

[0 | 1]

If 1 is specified, all character instances are shown. If 0 is specified, the debug text is turned off.

Examples

To show information on all character instances with "player" in their model name:

```
ca_debugtext player
```

To turn off all transition queue information:

```
ca_debugtext 0
```

Interpreting the Output

Each animation in the transition queue is displayed as in the following example. Key elements of this display are described following the example.

```
AnimInAFIFO 02: t:1043 _stand_tac_idle_scar_3p_01 ATime:0.84 (1.17s/1.40s) ASpd:1.00
 Flag:00000042 (----------I-K----) TTime:0.20 TWght:1.00 seg:00 inmem:1
(Try)UseAimIK: 1 AimIKBlend: 1.00 AimIKInfluence: 1.00 (Try)UseLookIK: 0 LookIKBlend: 0.00
 LookIKInfluence: 0.00
MoveSpeed: 4.49 locked: 1
PM class: AnimationPoseModifier_OperatorQueue, name: Unknown
...
LayerBlendWeight: 1.00
...
ADIK Bip01 RHand2RiflePos_IKTarget: 0.24 Bip01 RHand2Aim_IKTarget: 1.00 Bip01
 LHand2Aim_IKTarget: 0.00
```

Text Color

- When an animation is not yet active, it is in black or green.
- When an animation is active, it is in red or yellow.

Or in detail:

- Red Channel = Animation Weight
- Green Channel = (layerIndex > 0)
- Alpha Channel = (Weight + 1)*0.5

AnimInAFIFO Line (one per animation)

```
AnimInAFIFO 02: t:1043 _stand_tac_idle_scar_3p_01 ATime:0.84 (1.17s/1.40s) ASpd:1.00
 Flag:00000042 (----------I-K----) TTime:0.20 TWght:1.00 seg:00 inmem:1
```

AnimInAFIFO 02

Layer index (decimal, zero-based)

t:1043

User token (decimal)

_stand_tac_idle_scar_3p_01

Animation name (alias) of the currently playing animation, aim/look-pose or bspace

ATime:0.84 (1.17s/1.40s)

ATime:XXXX (YYYYs/ZZZZs)

- XXXX = Current time in 'normalized time' (0.0...1.0) within the current segment
- YYYY = Current time (seconds) within the current segment
- ZZZZ = Expected duration (seconds) of the current segment

ASpd:1.00

Current animation speed (1.0 = normal speed)

Flag:00000042 (----------I-K----)

Animation Flags

Flag:XXXXXXXX (+ybVFx3nSIAKTRLM)

The first number is the animation flags in hexadecimal

Between parentheses you see the individual flags:

char	flag	value
+	CA_FORCE_TRANSITION_TO_ANIM	0x008000
y	CA_FULL_ROOT_PRIORITY	0x004000
b	CA_REMOVE_FROM_FIFO	0x002000
V	CA_TRACK_VIEW_EXCLUSIVE	0x001000
F	CA_FORCE_SKELETON_UPDATE	0x000800
x	CA_DISABLE_MULTILAYER	0x000400
3	CA_KEYFRAME_SAMPLE_30Hz	0x000200
n	CA_ALLOW_ANIM_RESTART	0x000100
S	CA_MOVE2IDLE	0x000080
I	CA_IDLE2MOVE	0x000040
A	CA_START_AFTER	0x000020
K	CA_START_AT_KEYTIME	0x000010
T	CA_TRANSITION_TIMEWARPING	0x000008
R	CA_REPEAT_LAST_KEY	0x000004
L	CA_LOOP_ANIMATION	0x000002
M	CA_MANUAL_UPDATE	0x000001

TTime:0.20

Transition Time

Total length of transition into this animation in seconds (this is static after pushing the animation)

TWght:1.00

Transition Weight

Current weight of this animation within the transition (0 = not faded in yet, 1 = fully faded in)

seg:00

Current segment index (zero-based)

inmem:1

Whether or not the animation is in memory (0 basically means it's not streamed in yet)

Aim/Look-IK Line

```
(Try)UseAimIK: 1 AimIKBlend: 1.00 AimIKInfluence: 1.00 (Try)UseLookIK: 0 LookIKBlend: 0.00
 LookIKInfluence: 0.00
```

(Try)UseAimIK: 1

Whether Aim IK is turned on or not (set using PoseBlenderAim::SetState)

AimIKBlend: 1.00

Weight value requested for Aim IK (could go up and down based on fade times, etc.)

AimIKInfluence: 1.00

Final influence weight value of AimIK (== smoothed(clamped(AimIKBlend)) * weightOfAllAimPoses)

(Try)UseLookIK: 0

Whether Look IK is turned on or not

LookIKBlend: 0.00

Weight value requested for Look IK (could go up and down based on fade times, etc.)

LookIKInfluence: 0.00

Final influence weight value of LookIK (== smoothed(clamped(LookIKBlend)) * weightOfAllLookPoses)

Parameter Line(s) (only for blend spaces)

```
MoveSpeed: 4.500000 locked: 1
TravelAngle: 0.000000 locked: 0
```

MoveSpeed: 4.500000

Value for the specified blend space parameter (MoveSpeed in this case)

locked: 1

Whether or not the parameter is locked (= unable to change after it is set for the first time)

PoseModifier Lines (if running)

```
PM class: AnimationPoseModifier_OperatorQueue, name: Unknown
```

Displays which pose modifiers are running in this layer. Shows the class as well as the name (if available).

LayerBlendWeight Line (not on layer 0)

```
LayerBlendWeight: 1.00
```

The weight of this layer (0.00 - 1.00)

ADIK Line(s) (only if animation driven IK is applied)

```
ADIK Bip01 RHand2RiflePos_IKTarget: 0.24 Bip01 RHand2Aim_IKTarget: 1.00 Bip01
 LHand2Aim_IKTarget: 0.00
```

Displays a list of the animation driven IK targets and their current weight. For more detailed position/rotation information, use the separate cvar `ca_debugadiktargets 1`.

CommandBuffer Debugging

At the lowest level, the animation system executes a list of simple commands to construct the final skeleton's pose.

These commands are, for example, "sample animation x at time t, and add the result with weight w to the pose". Or "clear the pose".

To enable on-screen debug information to see what is pushed on the command buffer (for all characters), use the following command:

```
ca_debugcommandbuffer [0 | 1]
```

Warning Level

To control when the animation system produces warnings using the ca_animWarningLevel cvar:

```
ca_animWarningLevel [0 | 1 | 2 | 3]
```

0

Non-fatal warnings are off.

1

Warn about illegal requests.

For example, requesting to start animations with an invalid index.

2

Also warn about things like 'performance issues.'

For example, animation-queue filling up. This might 'spam' your console with a dump of the animation queue at the time of the issue.

3 (default)

All warnings are on. This includes the least important warnings; for example, a warning when playing uncompressed animation data.

Fall and Play

"Fall and Play" activates when a character is ragdollized (on an interface level, it is called `RelinquishCharacterPhysics`) with a >0 stiffness. This activates angular springs in the physical ragdoll that attempts to bring the joints to the angles specified in the current animation frame. The character also tries to select an animation internally based on the current fall and play stage. If there are none, or very few, physical contacts, this will be a falling animation; otherwise it will be the first frame of a standup animation that corresponds to the current body orientation.

Standup is initiated from outside the animation system through the fall and play function. During the standup, the character physics is switched back into an alive mode and his final physical pose is blended into a corresponding standup animation. This, again, is selected from a standup anims list to best match this pose.

Filename convention for standup animations: When an animation name starts with "standup", it is registered as a standup animation. Also, a type system exists which categorizes standup animations by the string between "standup_" and some keywords ("back", "stomach", "side"). You can control which type to use with `CSkeletonPose::SetFnPAnimGroup()` methods. At runtime, the engine checks the most similar standup animation registered to the current lying pose and blends to it.

Some example filenames:

- `standUp_toCombat_nw_back_01`
- `standUp_toCombat_nw_stomach_01`

While the character is still a ragdoll, it is also possible to turn off the stiffness with a `GoLimp` method.

Time in the Animation System

The Animation system uses different units of 'time,' depending on the system. How those units of time compare is best explained using an example.

The definition of 'frames': The Animation system uses a fixed rate of 30 frames per second (fps). Of course, games can run at higher frame rates, but some operations in the Editor that use the concept of 'frames'—or operations that clamp the animation duration to 'one frame'—assume a frame rate of 30 fps.

Assume then that you have an animation with a duration of 1.5 seconds. This means that the animation has 46 frames (note that this includes the final frame). So, in the case of Real Time, assume an animation starts at time 0, has no segmentation, and is played back at normal speed. However, rather than using Real Time, the Animation system typically uses Animation Normalized Time. This is compared with Real Time in the following table:

Frame Index	Real Time (seconds)*	Animation Normalized Time**
0	0.0 s	0.0
1	0.033.. s = 1/30 s	0.022.. = 1/45
..	..	..
30	1.0 s	0.666.. = 30/45
..	..	..
44	1.466.. s = 44/30 s	0.977.. = 44/45
45	1.5 s = 45/30 s	1.0

* Real time is used to define duration:

- Duration = lastFrame.realTime - firstFrame.realTime. That's 1.5s in our example.
- `IAnimationSet::GetDuration_sec()` returns the duration of an animation.

  Note: For a parametric animation, this returns only a crude approximation—the average duration of all its examples, ignoring parameters or speed scaling.
- `CAnimation::GetExpectedTotalDurationSeconds()` returns the duration of an animation that is currently playing back.

Note: For a parametric animation, this returns only a crude approximation, assuming the parameters are the ones that are currently set and never change throughout the animation.

- No function exists that returns the Real Time of an animation. To calculate that, you must manually multiply Animation Normalized Time with the duration.

** Animation Normalized Time:

- Time relative to the total length of the animation.
- Starts at 0 at the beginning of the animation and ends at 1 (= RealTime/Duration = Keytime/LastKeyTime).
- Used by functions such as `ISkeletonAnim::GetAnimationNormalizedTime()` and `ISkeletonAnim::SetAnimationNormalizedTime()`.
- Is not well-defined for parametric animations with examples that have differing numbers of segments. For more information, see the following section, Segmentation.

Segmentation

In practice, the animation system does not use Animation Normalized Time; this terminology was used to make the introduction easier to understand. Typically, Segment Normalized Time is used. To understand Segment Normalized Time, you must first understand segmentation.

For time warping (phase matching) purposes, animations can be split into multiple segments. For example, to time warp from a walk animation with 2 cycles to a walk animation with 1 cycle, you have to annotate the first animation and split it into two (these are segments). To achieve this segmentation, you must add a segment1 animation event at the border between the cycles.

> **Note**
> An animation without segmentation has exactly 1 segment, which runs from beginning to end.

Segmentation introduces a new unit for time, Segment Normalized Time, which is time relative to the current segment duration.

Extending our example further, observe what happens when a segment1 animation event at 1.0s is added to split the animation into two segments.

Frame Index	Real Time	AnimEvents	(Animation) Normalized Time	Segment Index*	Segment Normalized Time**
0	0.0 s		0.0	0	0.0
1	0.033.. s		0.022..	0	0.033.. = 1/30
..	..		..	..	..
30	1.0 s	segment1	0.666..	1	0.0
..	..		..	..	..
44	1.466.. s		0.977..	1	0.933.. = 14/15
45	1.5 s		1.0	1	1.0

* Segment index:

- Identifies which segment you are currently in. Runs from 0 to the total number of segments minus 1.
- While an animation is playing, you can use `CAnimation::GetCurrentSegmentIndex()` to retrieve it.
- When using ca_debugtext or es_debuganim, then this index is displayed after "seg:".

** Segment normalized time:

- Time relative to the current segment's duration.
- 0 at the beginning of the segment, 1 at the end (only 1 for the last segment, as you can see in the table).
- While an animation is playing, you can use `CAnimation::Get/SetCurrentSegmentNormalizedTime()` to get or set the Segment Normalized Time.
- As the names suggest, `CAnimation::GetCurrentSegmentIndex()` retrieves the current segment index and `CAnimation::GetCurrentSegmentExpectedDurationSecondsx()` retrieves the duration of the current segment.
- When representing time within parametric animations, it is more convenient to use Segment Normalized Time than Animation Normalized Time; therefore, Segment Normalized Time is used at runtime.
- AnimEvent time is specified using Animation Normalized Time (except for the special case of parametric animation; see the following section).
- When using `ca_debugtext` or `es_debuganim`, Segment Normalized Time is displayed after "ATime:". Following that, the real time within the segment and the segment duration are displayed within the parentheses.

Playback Speed

Playback speed does not impact the functions that compute duration of playing animations, such as `CAnimation::GetExpectedTotalDurationSeconds()` or `ISkeletonAnim::CalculateCompleteBlendSpaceDuration()`.

Segmented Parametric Animation

Animation Normalized Time, Segment Index, and Duration all create ambiguity for segmented parametric animations. This is because each example animation within the parametric animation can have its own number of segments. To avoid ambiguity, animation events in or on segmented parametric animations use Segment Normalized Time. As a result, an animation event will be fired multiple times (once per segment) during the animation.

- `ISkeletonAnim::GetAnimationNormalizedTime()` uses a heuristic: It currently looks for the example animation with the largest number of segments and returns the animation normalized time within that example.
- `ISkeletonAnim::GetCurrentSegmentIndex()` uses a different heuristic: It currently returns the segment index in the example animation, which happens to be the first in the list.

Given this, we are considering redefining the above based on the following observation: You can define the total number of segments in a parametric animation as the number of segments until repetition starts.

So, say you have a parametric animation consisting of 2 examples—one with 2 segments and the other with 3 segments. This will start to repeat after 6 segments (the lowest common multiple of 2 and 3).

However, you can uniquely identify each possible combination of segments using any number from 0 to 5.

The Character Tool uses this method to achieve a well-defined duration. The `ISkeletonAnim::CalculateCompleteBlendSpaceDuration()` function calculates the duration until the parametric animation starts to repeat (assuming the parameters remain fixed). It reverts to the regular `GetExpectedTotalDurationSeconds()` implementation for non-parametric animations so that the function can be used in more general situations.

Animation with Only One Key

Normally your animations have at least two keys. However, when you convert these into additive animations, the first frame is interpreted as the base from which to calculate the additive, leaving only 1 frame in the additive animation (this means that, in respect to the asset, both the start and end time of the asset are set to 1/30 s).

Functions retrieving the total duration of this animation will return 0.0 (for example, `IAnimationSet::GetDuration_sec()`, `ISkeletonAnim::CalculateCompleteBlendSpaceDuration()`, and `CAnimation::GetExpectedTotalDurationSeconds()`).

However, for playback purposes, the animation system handles these animations as if they have a duration of 1/30th of a second. For example, Animation Normalized Time still progresses from 0 to 1, while real time goes from 0 to 1/30th of a second. `CAnimation::GetCurrentSegmentExpectedDurationSecondsx()` also returns 1/30th of a second in this case.

Direction of Time

Time typically cannot run backward when playing an animation. You can move time backward only if you do it manually by setting the flag `CA_MANUAL_UPDATE` on the animation and using `CAnimation::SetCurrentSegmentNormalizedTime`. See the example DGLINK `CProceduralClipManualUpdateList::UpdateLayerTimes()`.

Time within Controllers

Different units are used for controllers that contain the actual key data and are used for animation sampling.

Frame Index	Real Time	I_CAF Ticks*	Keytime**
0	0.0 s	0	0.0
1	0.033.. s	160	1.0
..	..	..	..
30	1.0 s	4800	30.0
..	..	..	..
44	1.466.. s	7040	44.0
45	1.5 s	7200	45.0

* I_CAF Ticks:

- Used within I_CAF files to represent time
- There are 4800 I_CAF ticks per second (this is currently expressed by the fact that TICKS_CONVERT = 160 in Controller.h, which assumes 30 keys/second)

** Keytime

- Used at runtime to pass time to the controllers for sampling animation
- Used within CAF files to represent time
- A floating point version of 'frame index'
- Can represent time in between frames
- Use `GlobalAnimationHeaderCAF::NTime2KTime()` to convert from Animation Normalized Time to Keytime
- All animation controllers in the runtime use Keytime

Animation assets can also have a StartTime other than 0.0s—this complicates matters slightly, but only for the controllers. Typically, for everywhere but the controllers, time is taken relative to this StartTime.

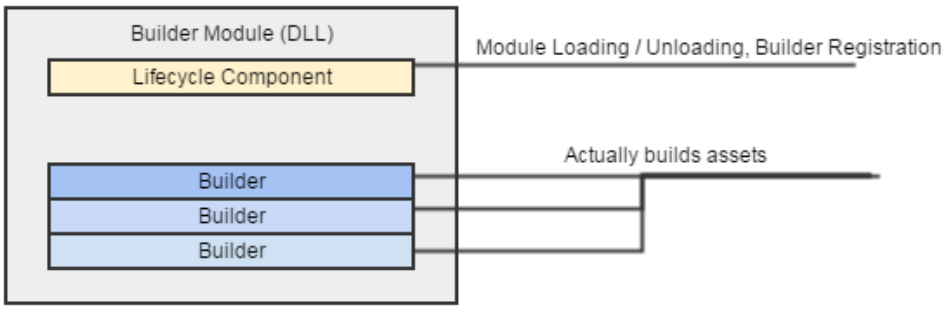
Asset Builder API

The asset builder API and builder SDK are in preview release for Lumberyard 1.5. and are subject to change as they undergo improvements.

You can use the asset builder API to develop a custom asset builder that creates your own asset types. Your asset builder can process any number of asset types, generate outputs, and return the results to the asset processor for further processing. This can be especially useful in a large project that has custom asset types.

Builder Modules

A builder module is a `.dll` module that contains a *lifecycle component* and one or more *builders*. The lifecycle component is derived from `AZ::Component`. The builders can be of any type and have no particular base class requirements.



The job of the lifecycle component is to register its builders during the call to `Activate()` and to make sure that resources that are no longer being used are removed in the calls to `Deactivate` and `Destructor`.

Creating a Builder Module

To create a builder module, you must perform the following steps.

- Create the exported `.dll` entry point functions and invoke the `REGISTER_ASSETBUILDER` macro, which creates a forward declaration for the entry point functions.
- Register your lifecycle component's `Descriptor`
- Add your lifecycle component to the `Builder` entity
- Register your builder instances when your lifecycle component's `Activate()` function is called
- Shut down safely

> **Note**
> A complete example of a builder module is in the Lumberyard `dev\Code\tools\AssetProcessor\Builders` directory. We recommend that you follow the commented example as you read this documentation. The asset builder SDK is located in the Lumberyard directory `\dev\Code\Tools\AssetProcessor\AssetBuilderSDK\AssetBuilderSDK`.

Main Entry Point

The following code shows an example of a `main.cpp` file for an asset builder module.

```
#include <AssetBuilderSDK/AssetBuilderSDK.h>
#include <AssetBuilderSDK/AssetBuilderBusses.h>

// Use the following macro to register this module as an asset builder.
// The macro creates forward declarations of all of the exported entry points for you.
REGISTER_ASSETBUILDER

void BuilderOnInit()
{
    // Perform any initialization steps that you want here. For example, you might start a
 third party library.
}

void BuilderRegisterDescriptors()
{
    // Register your lifecycle component types here.
    // You can register as many components as you want, but you need at least one component
 to handle the lifecycle.
    EBUS_EVENT(AssetBuilderSDK::AssetBuilderBus, RegisterComponentDescriptor,
 ExampleBuilder::BuilderPluginComponent::CreateDescriptor());
    // You can also register other descriptors for other types of components that you might
 need.
}

void BuilderAddComponents(AZ::Entity* entity)
{
    // You can attach any components that you want to this entity, including management
 components. This is your builder entity.
    // You need at least one component that is the lifecycle component.
    entity->CreateComponentIfReady<ExampleBuilder::BuilderPluginComponent>();
}

void BuilderDestroy()
{
    // By the time you leave this function, all memory must have been cleaned up and all
 objects destroyed.
    // If you have a persistent third party library, you could destroy it here.
```

```
}
```

Lifecycle Component

The lifecycle component reflects the types that you want to serialize and registers the builder or builders in your module during its `Activate()` function.

The following shows example code for the lifecycle component.

```
//! This is an example of the lifecycle component that you must implement.
//! You must have at least one component to handle your module's lifecycle.
//! You can also make this component a builder by having it register itself as a builder
 and
//! making it listen to the builder bus. In this example it is just a lifecycle component
 for the purposes of clarity.

class BuilderPluginComponent
    : public AZ::Component
{
public:
    AZ_COMPONENT(BuilderPluginComponent, "{8872211E-F704-48A9-B7EB-7B80596D871D}")
    static void Reflect(AZ::ReflectContext* context);

    BuilderPluginComponent(); // Avoid initializing here.

    ////////////////////////////////////////////////////////////////////////
    // AZ::Component
    virtual void Init(); // Create objects, allocate memory and initialize without reaching
 out to the outside world.
    virtual void Activate(); // Reach out to the outside world and connect to and register
 resources, etc.
    virtual void Deactivate(); // Unregister things, disconnect from the outside world.
    ////////////////////////////////////////////////////////////////////////

    virtual ~BuilderPluginComponent(); // free memory and uninitialize yourself.

private:
        ExampleBuilderWorker m_exampleBuilder;
};
```

In the following example, the `Activate()` function registers a builder, creates a builder descriptor, and then provides the details for the builder.

```
void BuilderPluginComponent::Activate()
{
    // Activate is where you perform registration with other objects and systems.

    // Register your builder here:
    AssetBuilderSDK::AssetBuilderDesc builderDescriptor;
    builderDescriptor.m_name = "Example Worker Builder";

 builderDescriptor.m_patterns.push_back(AssetBuilderSDK::AssetBuilderPattern("*.example",
 AssetBuilderSDK::AssetBuilderPattern::PatternType::Wildcard));
    builderDescriptor.m_createJobFunction = AZStd::bind(&ExampleBuilderWorker::CreateJobs,
 &m_exampleBuilder, AZStd::placeholders::_1, AZStd::placeholders::_2);
    builderDescriptor.m_processJobFunction = AZStd::bind(&ExampleBuilderWorker::ProcessJob,
 &m_exampleBuilder, AZStd::placeholders::_1, AZStd::placeholders::_2);
    // The above binds are an example of how to bind to a class. You could just use a set
 of functions instead; there is no particular requirement to use a class.
    builderDescriptor.m_busId = ExampleBuilderWorker::GetUUID(); // Shutdown is
 communicated on this bus address.
```

```
    m_exampleBuilder.BusConnect(builderDescriptor.m_busId); // You can use a global
 listener for shutdown instead of
                                                            // for each builder; it's up to
 you.

    EBUS_EVENT(AssetBuilderSDK::AssetBuilderBus, RegisterBuilderInformation,
 builderDescriptor);
}
```

Notes

- The example calls an EBus to register the builder. After you register a builder, the builder receives requests for assets from its two registered callback functions.
- If the application needs to shut down, the asset processor broadcasts the `Shutdown()` message on the builder bus using the address of the registered builder's UUID.
- Your builders do not have to be more than functions that create jobs and then process those jobs. But if you want your builder to listen for `Shutdown()` messages, it must have a listener that connects to the bus.

Creating a Builder

Your next step is to create a builder. You can have any number of builders, or even all of your builders, inside your module. After registering your builders as described in the previous section, implement the two `CreateJobFunction` and `ProcessJobFunction` callbacks.

The following example code declares a builder class:

```
#include <AssetBuilderSDK/AssetBuilderSDK.h>

class ExampleBuilderWorker  : public AssetBuilderSDK::AssetBuilderCommandBus::Handler //
 This handler delivers the "shut down!"
                                                                                       //
 message on another thread.
{
   public:
       ExampleBuilderWorker();
       ~ExampleBuilderWorker();

       //! Asset Builder Callback Functions
       void CreateJobs(const AssetBuilderSDK::CreateJobsRequest& request,
 AssetBuilderSDK::CreateJobsResponse& response);
       void ProcessJob(const AssetBuilderSDK::ProcessJobRequest& request,
 AssetBuilderSDK::ProcessJobResponse& response);

       ////////////////////////////////////////////////////////////////////////
       //!AssetBuilderSDK::AssetBuilderCommandBus interface
       void ShutDown() override; // When this is received, you must fail all existing jobs
 and return.
       ////////////////////////////////////////////////////////////////////////

       static AZ::Uuid GetUUID();

   private:
       bool m_isShuttingDown = false;
};
```

The asset processor calls the `Shutdown()` function to signal a shutdown. At this point, the builder should stop all tasks and return control to the asset processor.

Notes

- Failure to terminate promptly can cause a hang when the asset processor shuts down and restarts. The shutdown message comes from a thread other than the `ProcessJob()` thread.
- The asset processor calls the `CreateJobs(const CreateJobsRequest& request,CreateJobsResponse& response)` function when it has jobs for the asset types that the builder processes. If no work is needed, you do not have to create jobs in response to `CreateJobsRequest`, but the behavior of your implementation should be consistent.
- For the purpose of deleting stale products, the job that you spawn is compared with the jobs spawned in the last iteration that have the same input, platform, and job key.
- You do not have to check whether a job needs processing. Instead, at every iteration, emit all possible jobs for a particular input asset on a particular platform.
- In general, in the `CreateJobs` function, you create a job descriptor for each job that you want to emit, and then add the job to the list of job descriptors for the response.

The following code shows an example `CreateJobs` function.

```
// This function runs early in the file scanning pass.
// This function should always create the same jobs, and should not check whether the job
 is up to date.

void ExampleBuilderWorker::CreateJobs(const AssetBuilderSDK::CreateJobsRequest& request,
 AssetBuilderSDK::CreateJobsResponse& response)
{
    // The following example creates one job descriptor for the PC platform.
    // Normally, you create a job for each platform that you can make assets for.
    if (request.m_platformFlags & AssetBuilderSDK::Platform_PC)
    {
        AssetBuilderSDK::JobDescriptor descriptor;
        descriptor.m_jobKey = "Compile Example";
        descriptor.m_platform = AssetBuilderSDK::Platform_PC;

        // You can also place whatever parameters you want to save for later into this map:
        descriptor.m_jobParameters[AZ_CRC("hello")] = "World";
        response.m_createJobOutputs.push_back(descriptor);

        response.m_result = AssetBuilderSDK::CreateJobsResultCode::Success;
    }
}
```

How to Declare Source File Dependencies in CreateJobs

You can use the Builder SDK API to declare dependencies for a source file on other source files. These other files can be any file within the project directory or directories. They do not have to be source files consumed by a builder.

Declaring dependencies for a source file implies that the data in its output changes if the files that the source file depends on change. If any of the source dependency files are modified, the Asset Processor retriggers the `CreateJobs` and `ProcessJobs` sequence for those files. This causes the data to be recompiled by your builder when needed.

The Asset Processor recurses source file dependencies automatically. If the source files depended on emit their own dependencies when they are queried, you do not have to recurse in your own source files to the full tree. Just emit your local dependencies for each node in the tree, and the Asset Processor takes care of the rest.

To declare dependencies, add them during CreateJobs to `m_sourceFileDependencyList` in your `CreateJobsResponse` structure.

Metafiles do not have to be added as dependencies. Metafiles are a special case and cause your asset to rebuild automatically.

The `SourceFileDependency` structure contains `m_sourceFileDependencyPath` and `m_sourceFileDependencyUUID`. The builder must supply a value for only one of these fields. For example, if the UUID of the file to be added as a source dependency is known inside the `CreateJobs` method, the builder can just fill in the field `m_sourceFileDependencyUUID`. Otherwise the builder would need to fill in the field `m_sourceFileDependencyPath` with the appropriate source dependency file path.

It is important to note that the field `m_sourceFileDependencyPath` can take both absolute and relative file paths. If a relative path is specified, the appropriate overriding asset is used if present.

If the builder is populating the `m_sourceFileDependencyPath` field with a relative file path, then it has to be in relative to one of the watched directories. However, if both the source and the source dependency file exist in the same directory, you can provide just the filename without a path.

The following code snippet that shows how to use the Builder SDK API to add a source file dependency.

```
//! CreateJobsResponse contains job data that will be send by the builder to the
 assetProcessor in response to CreateJobsRequest
struct CreateJobsResponse
{
    CreateJobsResultCode         m_result = CreateJobsResultCode::Failed;   // The result
 code from the create jobs request

    AZStd::vector<SourceFileDependency> m_sourceFileDependencyList; // This is required for
 source files that want to declare dependencies on other source files.
    AZStd::vector<JobDescriptor> m_createJobOutputs;
};

//! Source file dependency information that the builder will send to the assetprocessor
//! It is important to note that the builder do not need to provide both the
 sourceFileDependencyUUID or sourceFileDependencyPath info to the asset processor,
//! any one of them should be sufficient
struct SourceFileDependency
{
    //! Filepath on which the source file depends, it can be either be a relative or an
 absolute path.
    //! if it's relative, the asset processor will check every watch folder in the order
 specified in the assetprocessor config file until it finds that file.
    //! For example if the builder sends the sourcedependency info with
 sourceFileDependencyPath = "texture/blah.tiff" to the asset processor,
    //! it will check all watch folders for a file whose relative path with regard to it is
 "texture/blah.tiff".
    //! then "C:/dev/gamename/texture/blah.tiff" would be considered the source file
 dependency, if "C:/dev/gamename" is only watchfolder that contains such a file.
    //! You can also send absolute path to the asset processor in which case the asset
 processor
    //! will try to determine if there are any other file which overrides this file based
 on the watch folder order specified in the assetprocessor config file
    //! and if an overriding file is found, then that file will be considered as the source
 dependency.
    AZStd::string m_sourceFileDependencyPath;

    //!  UUID of the file on which the source file depends
    AZ::Uuid m_sourceFileDependencyUUID = AZ::Uuid::CreateNull();
    ...
}

void ExampleBuilderWorker::CreateJobs(const AssetBuilderSDK::CreateJobsRequest& request,
 AssetBuilderSDK::CreateJobsResponse& response)
{
    //Looping over all the available platforms to spawn jobs.  note that source
 dependencies are independent of jobs and platforms.

    // This example shows how you would be able to declare source file dependencies on
 source files inside a builder and forward the info to the asset processor
```

```
    // Basically here we are creating source file dependencies as follows
    // the source file .../test.examplesource depends on the source file  .../
test.exampleinclude and
    // the source file .../test.exampleinclude depends on the source file .../
common.exampleinclude
    // the source file .../common.exampleinclude depends on the non-source file .../
common.examplefile
    // Important to note that both file extensions "exampleinclude" and "examplesource" are
 being handled by this builder.
    // Also important to note is that files with extension "exampleinclude" are not
 creating any jobdescriptor here, which imply they do not create any jobs.

    AZStd::string fullPath;
    AzFramework::StringFunc::Path::ConstructFull(request.m_watchFolder.c_str(),
 request.m_sourceFile.c_str(), fullPath, false);
    AzFramework::StringFunc::Path::Normalize(fullPath);
    AZStd::string relPath = request.m_sourceFile;
    AssetBuilderSDK::SourceFileDependency sourceFileDependencyInfo;

    // source files in this example generate dependencies and also generate a job to
 compile it for each platform:
    if (AzFramework::StringFunc::Equal(ext.c_str(), "examplesource"))
    {
        AzFramework::StringFunc::Path::ReplaceExtension(relPath, "exampleinclude");

        // declare and add the dependency on the .exampleinclude file:
        sourceFileDependencyInfo.m_sourceFileDependencyPath = relPath;
        response.m_sourceFileDependencyList.push_back(sourceFileDependencyInfo);

        // since we're a source file, we also add a job to do the actual compilation:
        for (size_t idx = 0; idx < request.GetEnabledPlatformsCount(); idx++)
        {
            AssetBuilderSDK::JobDescriptor descriptor;
            descriptor.m_jobKey = "Compile Example";
            descriptor.m_platform = request.GetEnabledPlatformAt(idx);
            // you can also place whatever parameters you want to save for later into this
 map, and it will be available in your process function
            descriptor.m_jobParameters[AZ_CRC("hello", 0x3610a686)] = "World";
            // add the job:
            response.m_createJobOutputs.push_back(descriptor);
        }
    }
    else if (AzFramework::StringFunc::Equal(ext.c_str(), "exampleinclude"))
    {
        // if we're processing an 'include file' then we still generate dependency
 information but emit no actual compilation jobs
        // similar to how CPP/H works.
        if (AzFramework::StringFunc::Find(request.m_sourceFile.c_str(),
"common.exampleinclude") != AZStd::string::npos)
        {
            // Add any dependencies that common.exampleinclude would like to depend on
 here, we can also add a non source file as a dependency like we are doing here
            sourceFileDependencyInfo.m_sourceFileDependencyPath = "common.examplefile";
        }
        else
        {
            AzFramework::StringFunc::Path::ReplaceFullName(fullPath,
"common.exampleinclude");
            // Assigning full path to sourceFileDependency path
            sourceFileDependencyInfo.m_sourceFileDependencyPath = fullPath;
        }
        response.m_sourceFileDependencyList.push_back(sourceFileDependencyInfo);
    }
    response.m_result = AssetBuilderSDK::CreateJobsResultCode::Success;
}
```

CreateJobsRequest Helper Functions

`CreateJobsRequest` provides helper functions for operations related to operating system platforms. The following code shows the helper functions provided by `CreateJobsRequest`.

```
//! Enum used by the builder for sending platform information.
    enum Platform: AZ::u32
    {
        Platform_NONE       = 0x00,
        Platform_PC         = 0x01,
        Platform_ES3        = 0x02,
        Platform_IOS        = 0x04,
        Platform_OSX        = 0x08,
        Platform_XBOXONE    = 0x10,
        Platform_PS4        = 0x20,

        //! If you add a new platform entry to this enum, you must also add it to
 AllPlatforms in order for the platform to be considered valid.
        AllPlatforms = Platform_PC | Platform_ES3 | Platform_IOS | Platform_OSX |
 Platform_XBOXONE | Platform_PS4
    };

        //! CreateJobsRequest contains input job data that is sent by the AssetProcessor to
 the builder to create jobs.
    struct CreateJobsRequest
    {
                ...
        //! Platform flags informs the builder about the platforms in which the
 AssetProcessor is interested.
        int m_platformFlags;

        // Return the number of platforms that are enabled for the source file.
        size_t GetEnabledPlatformsCount() const;

        // Return the enabled platform by index. If no platform is found, then
 Platform_NONE is returned.
        AssetBuilderSDK::Platform GetEnabledPlatformAt(size_t index) const;

        // Determine whether the inputted platform is enabled. Returns true if enabled,
 otherwise false.
        bool IsPlatformEnabled(AZ::u32 platform) const;

        // Determine whether the inputted platform is valid. Returns true if valid,
 otherwise false.
        bool IsPlatformValid(AZ::u32 platform) const;
    };
```

- `GetEnabledPlatformsCount()` – For the specified platform flags, returns the number of platforms that are enabled for the create jobs request.
- `IsPlatformEnabled(AZ::u32 platform)` – For the specified platform, returns whether the platform is enabled. True if the platform is enabled, false otherwise.
- `IsPlatformValid(AZ::u32 platform)` – For the specified platform value, checks whether it is a valid platform value in the platform enum. True if the platform value is valid, false otherwise.

ProcessJob

The asset processor calls the `ProcessJob` function when it has a job for the builder to begin processing:

```
ProcessJob(const AssetBuilderSDK::ProcessJobRequest& request,
 AssetBuilderSDK::ProcessJobResponse& response)
```

`ProcessJob` is given a job request that contains the full job descriptor that `CreateJobs` emitted, as well as additional information such as a temporary directory for it to work in.

This message is sent on a worker thread, so the builder must not spawn threads to do the work. Be careful not to interact with other threads during this call.

> **Warning**
> Do not alter files other than those in the temporary directory while `ProcessJob` is running. After your job indicates success, the asset processor copies your registered products to the asset cache, so be sure not to write to the cache. You can use the temporary directory in any way that you want.

After your builder has finished processing assets, your response structure should list all of the assets that you have created. Because only the assets that you list are added to the cache, you can use the temporary directory as a scratch space for processing.

The following code shows an example `ProcessJob` function.

```
// This function is called for jobs that need processing.
// The request contains the CreateJobResponse you constructed earlier, including the
// keys and values you placed into the hash table.

void ExampleBuilderWorker::ProcessJob(const AssetBuilderSDK::ProcessJobRequest& request,
 AssetBuilderSDK::ProcessJobResponse& response)
{
        // The following example shows how to listen for cancellation requests. You should
 listen for cancellation requests and then cancel work if possible.
        // You can derive from the job cancel listener and reimplement Cancel() if you need
 to do more processing like signaling a
        // semaphore or doing other threading work.

    AssetBuilderSDK::JobCancelListener jobCancelListener(request.m_jobId);
    AZ_TracePrintf(AssetBuilderSDK::InfoWindow, "Starting Job.");
    AZStd::string fileName;
    AzFramework::StringFunc::Path::GetFullFileName(request.m_fullPath.c_str(), fileName);
    AzFramework::StringFunc::Path::ReplaceExtension(fileName, "example1");
    AZStd::string destPath;

    // Do all of your work inside the tempDirPath.
    // Do not write outside of this path
    AzFramework::StringFunc::Path::ConstructFull(request.m_tempDirPath.c_str(),
 fileName.c_str(), destPath, true);

    // Use AZ_TracePrintF to communicate job details. The logging system automatically
 places the
    // text in the appropriate log file and category.

    AZ::IO::LocalFileIO fileIO;
    if (!m_isShuttingDown && !jobCancelListener.IsCancelled() &&
 fileIO.Copy(request.m_fullPath.c_str(), destPath.c_str()) == AZ::IO::ResultCode::Success)
    {
        // If assets were successfully built into the temporary directory, push them back
 into the response's product list.
        // The assets that you created in your temporary path can be specified using paths
 relative to the temporary path.
        // It is assumed that your code writes to the temporary path.
        AZStd::string relPath = destPath;
        response.m_resultCode = AssetBuilderSDK::ProcessJobResult_Success;
```

```
        AssetBuilderSDK::JobProduct jobProduct(fileName);
        response.m_outputProducts.push_back(jobProduct);
    }
    else
    {
        if (m_isShuttingDown)
        {
            AZ_TracePrintf(AssetBuilderSDK::ErrorWindow, "Cancelled job %s because shutdown
 was requested", request.m_fullPath.c_str());
            response.m_resultCode = AssetBuilderSDK::ProcessJobResult_Cancelled;
        }
        else
        {
            AZ_TracePrintf(AssetBuilderSDK::ErrorWindow, "Error during processing job %s.",
 request.m_fullPath.c_str());
            response.m_resultCode = AssetBuilderSDK::ProcessJobResult_Failed;
        }
    }
}
```

JobCancelListener

Builders can use `JobCancelListener` to listen for job cancellation requests in their `processJob` method. You should listen for cancellation requests and then cancel work if possible. The address of this listener is the job ID of the job that can be found in `processJobRequest`. You can derive from the `JobCancelListener` and reimplement `Cancel()` if you must do additional processing like signaling a semaphore or other threading work. The following code example shows the use of `JobCancelListener`.

```
//! JobCancelListener can be used by builders in their processJob method to listen for job
 cancellation requests.
//! The address of this listener is the job ID that can be found in the process job
 request.
class JobCancelListener : public JobCommandBus::Handler
{
public:
explicit JobCancelListener(AZ::u64 jobId);
~JobCancelListener() override;
JobCancelListener(const JobCancelListener&) = delete;
////////////////////////////////////////////////////////////////////////
//JobCommandBus::Handler overrides
//Note: This is called on a thread other than your processing job thread.
//You can derive from JobCancelListener and reimplement Cancel if you need to do something
 special in order to cancel your job.
void Cancel() override;
////////////////////////////////////////////////////////////////////////

bool IsCancelled() const;

private:
AZStd::atomic_bool m_cancelled;
};
```

Notes

- So that critical files are not missed, the editor is blocked until all jobs are created. For this reason, you should execute the code in `CreateJobs` as quickly as possible. We recommend that your code do minimal work during `CreateJobs` and save the heavy processing work for `ProcessJob`.
- In `CreateJobs`, you can place arbitrary key–value pairs into the descriptor's `m_jobParameters` field. They key–value pairs are copied back when `ProcessJob` executes, which removes the need for you to add them again.

- All of the outputs for your job should be placed into your temporary workspace. However, if you just need to copy an existing file into the asset cache as part of your job, you can emit as a product the full absolute source path of the file without copying it into your temporary directory first. The asset processor then copies the file into the cache and registers it as part of the output of your job. All other files are moved from your temporary directory into the asset cache in an attempt to perform an atomic cache update in case your job succeeds.

Message Loggging

You can use `BuilderLog(AZ:Uuid builderId, char* message, ...)` to log any general builder related messages or errors. `BuilderLog` cannot be used during job processing; use it during startup, shutdown, or registration.

For job related messages, use `AZ_TracePrintf(window, msg), AZ_Warning(...), AZ_Error(...), AZ_Assert(...)`, and so on. . The function automatically records the messages in the log file for the job.

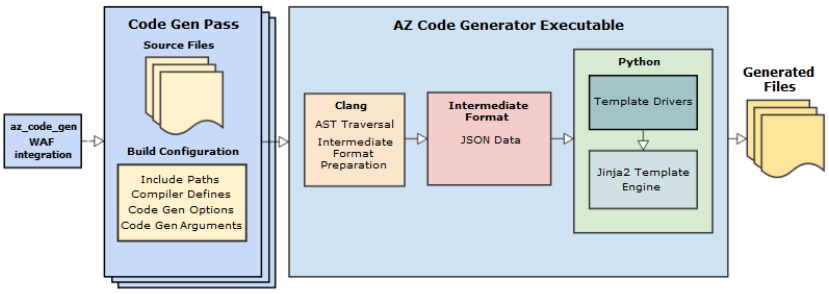
AZ Code Generator

AZ Code Generator is in preview release and is subject to change.

AZ Code Generator is a command line utility that generates source code (or any data or text) from specially tagged source code. You can use it when the structure of the intended code is known in advance so that templates can be made for it. For example, you could generate boilerplate code for serialization or reflection.

AZ Code Generator parses a list of existing C++ source files and/or header files and generates intermediate data in JSON format. It passes the intermediate data to a series of templates.

The templates provide the format for the code that is generated. Templates make increased coding efficiency possible because they enable automatic updates of boilerplate code. When a template is updated, all related generated code is regenerated in the next build. This removes the need to update the glue code manually or to use error-prone find-and-replace operations.



Topics

- Workflow Summary (p. 127)
- Waf (p. 127)
- Clang (p. 128)
- Intermediate JSON Data (p. 128)

- AZ Code Generator and Python (p. 129)
- Template Drivers and Template Rendering (p. 129)
- Generated Files (p. 129)
- AZ Code Generator Integration with Waf (p. 131)
- AZ Code Generator Parameters (p. 133)
- Code Generation Templates (p. 137)
- Template Drivers (p. 139)
- Custom Code Generator Annotations (p. 143)
- Waf Debugging with AZ Code Generator (p. 148)
- Template Driver Debugging (p. 154)
- Debugging the AZ Code Generator Utility (p. 155)
- Intermediate JSON Data Format (p. 157)

Workflow Summary

The following steps describe how AZ Code Generator works with Waf to generate code.

1. The Waf build system invokes AZ Code Generator for the `.h` and `.cpp` source files that are specified in the `wscript` file.
2. AZ Code Generator runs one or more passes with the specified files.
3. Each pass includes the following:
   a. AZ Code Generator uses the Clang front-end compiler to produce an abstract syntax tree (AST) for each provided source file. The Clang parser attempts to compile the input. For increased speed, Clang can be instructed to not follow `#include` statements and to suppress all errors.
   b. The AST is translated into an intermediate JSON format.
   c. The intermediate JSON object is passed into a template driver as a Python script and then into a Jinja2 template. Each driver and template implements specific code generation tasks.
   d. The template driver performs any preprocessing that you want on the intermediate JSON object.
   e. The intermediate JSON is then passed to Jinja2 templates.
   f. Each template driver can have an arbitrary number of templates, which can output to an arbitrary number of output files. Multiple templates can have the same output file or different output files as the template driver creator wants.
4. AZ Code Generator returns a list of generated files to the Waf build system.
5. The Waf build system completes the build process, including the generated code in the build.

The following sections provide more detail about this process.

Waf

The AZ Code Generator is fully integrated into the Waf build system. You can use the Waf `az_code_gen` feature to invoke the AZ Code Generator. We recommend that you use Waf rather than the command line to start the `AzCodeGenerator.exe` utility.

For examples and more information about the Waf integration, see AZ Code Generator Integration with Waf (p. 131).

Clang

The default front end of the AZ Code Generator is a Clang parser/compiler for C++ source code. AZ Code Generator uses Clang to parse source code (which might include user-defined tags) and generate the intermediate JSON data object. AZ Code Generator fully controls Clang's parser and compilation phase so that it can selectively suppress or enable features such as diagnostics. This gives AZ Code Generator the flexibility to ignore source code that might fail to compile and still attempt to generate a complete intermediate object.

Intermediate JSON Data

The Clang front end compiler outputs an intermediate JSON data structure that the generator passes to templates for further processing. An example intermediate JSON data object follows.

```
[
    {
        'name' : 'Component',
        'qualified_name' : 'AZ::Component',
        'fields' : [],
        'bases' : [],
        'meta' : {
            'path' : 'D:\\Repo\\Ly\\branches\\AzComponents\\Code\\Tools\\AzCodeGenerator\
\CodeGenTest.h'
        },
        'type' : 'class',
        'annotations' : {},
        'methods' : []
    },
    {
        'name' : 'TestingClass',
        'qualified_name' : 'TheNamespace::TestingClass',
        'fields' : [
            {
                'type' : 'float',
                'name' : 'm_field2',
                'qualified_name' : 'TheNamespace::TestingClass::m_field2',
                'annotations' : {}
            }
        ],
        'bases' : [
            {
                'name' : 'Component',
                'qualified_name' : 'AZ::Component'
            }
        ],
        'meta' : {
            'path' : 'D:\\Repo\\Ly\\branches\\AzComponents\\Code\\Tools\\AzCodeGenerator\
\CodeGenTest.h'
        },
        'type' : 'class',
        'annotations' : {},
        'methods' : [
            {
                'params' : ['type', 'int', 'name', 'version'],
                'name' : 'CreateArgumentAnnotation',
                'qualified_name' : 'TheNamespace::TestingClass::CreateArgumentAnnotation',
                'annotations' : {}
            }
        ]
    }
```

```
]
```

For complete syntax of the intermediate JSON data object, see Intermediate JSON Data Format (p. 157).

AZ Code Generator and Python

AZ Code Generator depends on Python 2.7 to run template drivers and render Jinja templates. The Python C API is used to extend Python with methods in the `azcg_extension` module that permit template drivers to report dependencies, errors, and useful informational output. In Windows, Python 2.7 is included in the Lumberyard `dev/Tools/Python` directory. On macOS, AZ Code Generator uses the version of Python that is included with the operating system.

> **Note**
> To debug Python C API calls when using AZ Code Generator, you must download CPython. Then make a build for your intended debug platform.

Template Drivers and Template Rendering

You can use template drivers written in Python to alter the intermediate data structure before passing it to the template engine. After preprocessing, the template driver might direct the Jinja2 template engine to render one or many templates, depending on the generated code that you want.

AZ Code Generator uses the Jinja2 template engine, which is downloaded by the Python easy_install script in the `\dev\Tools\Python\2.7.11\windows\Scripts` directory. The engine is then copied into the Lumberyard `3rdParty\jinja2` directory. Lumberyard also provides a `jinja_extensions` module, which contains helper methods that you can use inside templates. These extensions are stored in the `dev/Code/Tools/AzCodeGenerator/Scripts/jinja_extensions/` directory. For examples and more information about Jinja templates, see Code Generation Templates (p. 137).

Generated Files

The following sample output was generated from a serialization template. The reference JSON object has been formatted for readability.

```
////////////////////////////////////////////////////////
////////////////////////////////////////////////////////
// THIS CODE IS AUTOGENERATED, DO NOT MODIFY
////////////////////////////////////////////////////////
////////////////////////////////////////////////////////
#include "stdafx.h"
#include <AZCore/Rtti/ReflectContext.h>
#include <AzCore/Rtti/Rtti.h>
#include <AzCore/Serialization/SerializeContext.h>
#include <AzCore/Math/Vector3.h>
#include "D:/Repo/Ly/branches/AzComponents/Code/Tools/AzCodeGenerator/CodeGenTest.h"
namespace Components
{
    void TestingClassReflect(AZ::ReflectContext* reflection)
    {
        AZ::SerializeContext* serializeContext =
 azrtti_cast<AZ::SerializeContext*>(reflection);
```

```
        if (serializeContext)
        {
            serializeContext->Class<TestingClass>()
                    ->SerializerForEmptyClass()                 ;
        }
    }
}
////////////////////////////////////////////////////////////////
/*
// Reference JSON object
[{
   'name':'Component',
   'qualified_name':'AZ::Component',
   'fields':[

   ],
   'bases':[

   ],
   'meta':{
      'path':'D:\\Repo\\Ly\\branches\\AzComponents\\Code\\Tools\\AzCodeGenerator\
\CodeGenTest.h'
   },
   'type':'class',
   'annotations':{

   },
   'methods':[

   ]
},
{
   'name':'TestingClass',
   'qualified_name':'TheNamespace::TestingClass',
   'fields':[
      {
         'type':'float',
         'name':'m_field2',
         'qualified_name':'TheNamespace::TestingClass::m_field2',
         'annotations':{

         }
      }
   ],
   'bases':[
      {
         'name':'Component',
         'qualified_name':'AZ::Component'
      }
   ],
   'meta':{
      'path':'D:\\Repo\\Ly\\branches\\AzComponents\\Code\\Tools\\AzCodeGenerator\
\CodeGenTest.h'
   },
   'type':'class',
   'annotations':{

   },
   'methods':[
      {
         'params':[
            'type',
            'int',
            'name',
            'version'
         ],
```

```
            'name':'CreateArgumentAnnotation',
            'qualified_name':'TheNamespace::TestingClass::CreateArgumentAnnotation',
            'annotations':{

            }
        }
    ]
}]
*/
```

AZ Code Generator Integration with Waf

AZ Code Generator is in preview release and is subject to change.

AZ Code Generator is fully accessible for any Waf target as the feature `az_code_gen`. The `dev/Code/Tools/waf-1.7.13/lmbrwaflib/az_code_generator.py` file contains the core of the Waf integration code. It includes the `az_code_gen` feature that can be used by any `wscript` file.

The minimum required information is a list of the files to pass into the code generator and at least one template driver. This list feeds the code generator one file at a time and invokes the templates specified by the driver. The files output from the driver are added as dependencies of the build task. Output files also have the option to be reinjected back into the C++ build for compilation. Output file paths are automatically added as include paths both for the current target build and as `export_header` entries. This allows written source code to reference the generated source code from both internal and external targets.

Topics

- Basic Integration (p. 131)
- Advanced Integration (p. 132)
- Input Files (p. 132)
- Template Drivers (p. 133)
- Command Line Parameters (p. 133)
- Waf Specific Options (p. 133)

Basic Integration

In the `wscript` file for the target requiring generated code, add the `az_code_gen` feature as follows.

```
features = ['az_code_gen'],
```

Next, specify the files to pass as input to the code generator, as in the following example.

```
az_code_gen = [
    {
        'files' : ['MySourceFile.h'],
        'scripts' : ['MyTemplateDriver.py']
    }
],
```

The paths given are relative to the target path in both cases.

Whenever the specified target is compiled, a code generation task passes in the `MySourceFile.h` file to the code generator. It also invokes the `MyTemplateDriver.py` file to control the output. For information on how to write a template driver, see Template Drivers (p. 139).

Advanced Integration

The AZ Code Generator Waf integration uses passes to define the code generator tasks that must be run during build time. Each pass determines the set of files, drivers, and environment settings with which to run the code generator. Currently, all passes are run in parallel without any dependency checking between passes.

The following example shows the configuration of multiple passes.

```
az_code_gen = [
    {
        'files' : ['MyCode/MySourceFile.h'],
        'scripts' : ['MyCode/MyTemplateDriver.py']
    },
    {
        'files' : ['MyOtherCode/MyOtherSourceFile.h'],
        'scripts' : ['MyOtherCode/MyOtherTemplateDriver.py']
    }
],
```

This example generates the following two code generation tasks.

1. Pass in the `MyCode/MySourceFile.h` file to the code generator and invoke the `MyCode/MyTemplateDriver.py` file to control the output.
2. Pass in `MyOtherCode/MyOtherSourceFile.h` to the code generator and invoke `MyOtherCode/MyOtherTemplateDriver.py` to control the output.

Input Files

Each pass provides a list of files that will be used as input to the code generator. This list can also contain string paths, nodes, and lists. Top-level string paths and nodes are passed individually to the code generator. Note the following:

- If you provide a list, all files or nodes in that list are used by the code generator at the same time. This allows for maximum flexibility, but typical usage is one input per task.
- The overhead of the Waf task and AZ Code Generator bootstrapping can be significant. To improve performance, you can pass in multiple input files in one list.
- The code generator invokes the same Clang and template driver pipeline for each input file.

The following example shows several input file specifications.

```
# Finds this file relative to the build context source node
'files' : [bld.srcnode.find_or_declare('Code/Framework/AzCore/Tests/CodeGen.h')],

'files' : [
# Pass both MyClass.h and MyClass.cpp at the same time to code generator to get more
# information about MyClass than just the header. Note the nested lists.
    ['MyClass.h', 'MyClass.cpp']
]

'files' : [
```

```
    # Any and all variations are allowed, but because lists provide only one layer of
 grouping,
    # lists are allowed only at the top level.
    'MySourceFile.h',
    'MyOtherSourceFile.cpp',
    bld.srcnode.find_or_declare('Code/Framework/AzCore/Tests/CodeGen.h'),
    ['MyClass.h', 'MyClass.cpp']
]
```

Template Drivers

To specify template drivers to use for each code generation pass, provide a list of string paths, relative to the target path, as in the following example.

```
'scripts' : [
    '../../../Framework/AzFramework/CodeGen/AzClassCpp.py',
    '../../../Framework/AzFramework/CodeGen/AzEBusInline.py',
    '../../../Framework/AzFramework/CodeGen/AzReflectionCpp.py',
    '../../../Framework/AzFramework/CodeGen/AzClassInline.py'
],
```

Command Line Parameters

All command line parameters for the code generation utility can be specified in each code generation pass. To do this, provide a list of arguments, as in the following example.

```
'arguments' : [
    '-OnlyRunDiagnosticsOnMainFile=true',
    '-SuppressDiagnostics=false',
    '-SuppressErrorsAsWarnings=false',
    '-output-redirection=file',
],
```

For a full list of parameters, see AZ Code Generator Parameters (p. 133).

Waf Specific Options

The Waf integration provides additional options that can be specified in a list for each code generation pass, as in the following example.

```
'options' : ['PrintOutputRedirectionFile'],
```

`PrintOutputRedirectionFile` – This option, when used in combination with the `-output-redirection=file` parameter, directs Waf to provide AZ Code Generator a path to save extra output during code generation. The path to this file is listed for each task during the build if errors occur.

`Profile` – This option enables profiler timings of clang parsing and script execution within the AZ Code Generator tool.

AZ Code Generator Parameters

> AZ Code Generator is in preview release and is subject to change.

For best results, pass the options for AZ Code Generator in to the Waf build system. However, you can also specify the parameters for `AzCodeGenerator.exe` on the command line.

Topics

- Waf Parameters (p. 134)
- Clang Compilation Parameters (p. 134)
- Intermediate Data (p. 134)
- Front End (p. 134)
- AZ Code Generator Parameter List (p. 134)

Waf Parameters

Most parameters for AZ Code Generator are specified by the Waf integration. Parameters such as input, output, and include paths are automatically detected and forwarded. Other AZ Code Generator parameters control how AZ Code Generator deals with the source code input and the intermediate data that is generated.

Specify any of these in the `arguments` section of the `az_code_gen` pass in the `wscript` file.

Clang Compilation Parameters

The following `AzCodeGenerator.exe` parameters apply to Clang compilation.

Parameter	Description
`-SuppressIncludeNotFoundError`	Suppresses unknown `#include` statements at compile time.
`-OnlyRunDiagnosticsOnMainFile`	Ignores build warnings and errors on all except the main file specified for compilation.
`-SuppressDiagnostics`	Ignores build warnings and errors on all files.
`-SuppressErrorsAsWarnings`	Downgrades any build errors to warnings. Allows Clang to succeed even if there are errors.

Intermediate Data

To include information about code outside of the input file in the intermediate JSON data, use the following option.

```
-inclusion-filter=<wildcard filter for files to allow>
```

Front End

You can choose the front end to use by specifying either the `-Clang` (the default) or `-JSON` option.

AZ Code Generator Parameter List

The following list shows all AZ Code Generator parameters.

Usage: `AzCodeGenerator.exe [options]`

Option	Category	Description
`-Clang`	General	Uses the Clang compiler front end.
`-clang-settings-file=<string>`	Code parsing	The path to the file that contains Clang configuration settings.
`-codegen-script=<string>`	Python	The absolute path and file name of the code generation script to invoke.
`-debug`	General	Enables debug output.
`-debug-buffer-size=<uint>`	General	Buffers the last `n` characters of debug output until program termination. The default is `0`, which specifies immediate print out.
`-debug-only=<debug string>`	General	Enables a specific type of debug output.
`-define=<string>`	Code parsing	Specifies a preprocessor definition.
`-DelayedTemplateParsing`	AST traversal	Consumes and stores template tokens for parsing at the end of the translation unit.
`-EnableIncrementalProcessing`	AST traversal	Enables incremental processing.
`-force-include=<string>`	Code parsing	List of headers to forcibly include in Clang parsing.
`-help`	General	Displays basic options in categorized format.
`-help-hidden`	General	Displays all available options in categorized format.
`-help-list`	General	Displays basic options in list format.
`-help-list-hidden`	General	Displays all available options in list format.
`-include-path=<string>`	Code parsing	The header includes the path.
`-inclusion-filter=<string>`	Code filtering	Specifies a wildcard filter so that files other than those specified by `input-files` are parsed by Clang into intermediate data.
`-info-output-file=<filename>`	General	File to which to append `-stats` output.
`-input-file=<string>`	Code parsing	(Required) Path to input file relative to the value of `input-path`.
`-input-path=<string>`	Code parsing	(Required) The absolute path to input folder. All `input-file` paths must be relative to this folder.
`-intermediate-file=<string>`	Code parsing	Path to a file that stores the JSON AST from Clang parsing.
`-JSON`	General	Uses raw JSON input for the front end.
`-noscripts`	General	Disables the running of code generation scripts.

Option	Category	Description
`-OnlyRunDiagnosticsOnMainCompilation`	Clang compilation	Runs diagnostics (error and warning checking) only on the main file that is compiled. Ignores errors and warnings from all other files.
`-output-path=<string>`	Code parsing	(Required) The absolute path to the output folder.
`-output-redirection`	Output	Redirects output and error messages from Clang and Python internal utilities. Options: `=none` – No output redirection. Clang and Python output to `stdout` and `stderr`. `=null` – Redirect Clang and Python to null, effectively suppressing output. `=file` – Redirect Clang and Python to disk. Use `redirect-output-file` to specify the path.
`-output-using-json`	Output	Outputs using JSON objects instead of plain text. Use this option to ease parsing for calling applications.
`-print-all-options`	General	Prints all option values after command line parsing.
`-print-options`	General	Prints nondefault options after command line parsing.
`-profile`	General	Enables AZ Code Generator's internal profiler and emits timings for Clang parsing and script execution.
`-python-debug-path=<string>`	Python	Path to Python debug libraries and scripts for `AzCodeGenerator.exe` to use in debugging.
`-python-home=<string>`	Python	(Required) The equivalent of the `PYTHONHOME` environment variable, which is ignored.
`-python-home-debug=<string>`	Python	The equivalent of the debug Python `PYTHONHOME` environment variable, which is ignored.
`-python-path=<string>`	Python	The path to Python libraries and scripts for `AzCodeGenerator.exe`.
`-redirect-output-file=<string>`	Output	The file path for redirected output. Use in combination with the `-output-redirection=file` option. The default file name is `output.log`.
`-resource-dir=<string>`	Code parsing	The path to the resource directory for Clang.
`-stats`	General	Enables statistics output from program (available with asserts). Use the `-info-output-file=<filename>` option to specify the output file.
`-SkipFunctionBodies`	AST traversal	Does not traverse function bodies.
`-SuppressDiagnostics`	Clang compilation	Hides Clang compilation diagnostic information.

Option	Category	Description
`-SuppressErrorsAsWarnings`	Clang compilation	Suppresses compilation errors during parsing by reporting them as warnings.
`-SuppressIncludeNotFoundErrors`	AST traversal	Suppresses `#include` not found errors.
`-track-memory`	General	Enables `-time-passes` memory tracking. Performance might be slow when this option is used.
`-v`	General	Outputs verbose debug information.
`-version`	General	Displays the version of `AzCodeGenerator.exe`.
`-view-background`	General	Executes the graph viewer in the background. This option creates a `.tmp` file that must be deleted manually.

Code Generation Templates

AZ Code Generator is in preview release and is subject to change.

AZ Code Generator uses the Jinja2 template engine for Python to render its output. The Jinja template engine outputs plain text with embedded variable and logic statements.

Jinja templates are designed to be highly readable and mimic the overall structure of the desired output. They are processed top to bottom. Any text outside of the control block in the template is sent directly to the output.

The following are some example templates. For more information about creating Jinja templates, refer to the Jinja Template Designer Documentation.

Topics

- Simple Example (p. 137)
- Complex Example (p. 138)
- Template Data (p. 139)

Simple Example

A Jinja template can use text variables to replace text at predetermined locations in the output, as in the following example:

```
// Here's a {{ variable_name }} !!
int {{ variable_name }} = {{ variable_value }};
```

In this example, the Jinja template is given the following input.

```
{
    'variable_name' = 'foo',
    'variable_value' = 42
}
```

The following output results.

```
// Here's a foo !!
int foo = 42;
```

Complex Example

Jinja allows for fairly complicated logic, branching and looping control structures. The following example template generates a class that has the public and private variables specified by the input:

```
// This class is auto-generated!
class {{ class.name }}
{
public:
    virtual ~{{ class.name }}() = default;

{% if class.members is defined %}
    {% for member_var in class.members if member_var.visibility is 'public' -%}
    {{ member_var.type }} m_{{ member_var.name }}{{ if member_var.value is defined }} =
 {{ member_var.value }}{{ endif }};
    {%- endfor %}
{% endif %}
private:
{% if class.members is defined %}
    {% for member_var in class.members if member_var.visibility is 'private' -%}
    {{ member_var.type }} m_{{ member_var.name }}{{ if member_var.value is defined }} =
 {{ member_var.value }}{{ endif }};
    {%- endfor %}
{% endif %}
};
```

In this example, the Jinja template is given the following input.

```
{
    'class' : {
        'name' : 'MyClass',
        'members' : [
            {
                'name' : 'foo',
                'type' : 'int',
                'visibility' : 'public'
            },
            {
                'name' : 'bar',
                'type' : 'long',
                'visibility' : 'public',
            },
            {
                'name' : 'secretSauce',
                'type' : 'float',
                'visibility' : 'private',
                'value' : '98.6f'
            }
        ]
    }
}
```

The template produces the following output.

```
// This class is auto-generated!
class MyClass
{
public:
```

```
    virtual ~MyClass() = default;

    int m_foo;
    long m_bar;
private:
    float m_secretSauce = 98.6f;
};
```

Template Data

The data that is available to the template is fully controlled by the Python template driver (p. 139).

The following table lists the variables that are automatically added to the Jinja environment.

Variable	Description
`extra_data`	Python object that contains data returned by the apply_transformations (p. 140) method of the template driver.
`extra_str`	String that contains the contents of `extra_data` in JSON format.
`json_object`	Python object that contains the decoded intermediate JSON after it has been processed by the template driver.
`json_str`	String that contains the encoded intermediate JSON after it has been processed by the template driver.

For information about the intermediate output, see Intermediate JSON Data Format (p. 157).

> **Note**
> Because Jinja contains a limited feature set, attempting to do complex data transformations in Jinja templates produces overly complicated and generally unreadable templates. For this reason, we recommend that you perform any major data manipulation in the template driver before it is passed into the Jinja template engine. For more information, see Template Drivers (p. 139).

Template Drivers

AZ Code Generator is in preview release and is subject to change.

Template drivers are Python scripts that process the intermediate JSON data and route it into the Jinja2 output templates. The scripts preprocess the data from the Clang front end, execute the template rendering, and control where the generated output is written to disk.

These scripts are usually called by one or more code generation passes in WAF `wscript` files. Each Python script can reference multiple templates. This offers great flexibility in implementation, especially when multiple templates rely on the same preprocessed data.

Topics

- Specifying Drivers in Waf (p. 140)
- Creating a Template Driver in Python (p. 140)
- Minimal Template Driver (p. 142)
- Rendering Templates (p. 142)

- Configuring Automatic Build Injection (p. 142)
- Preprocessing Intermediate Data (p. 143)

Specifying Drivers in Waf

Drivers are specified by file name in each code generation pass. The file path is relative to the root of the `wscript` target. All drivers are invoked on each input file.

The following shows the structure of a sample Waf entry.

```
'az_code_gen' = [
    {
        'files': [ <files to gen> ],
        'scripts': [ <list of script file paths relative to current wscript folder> ]
    }
]
```

For more details on how to specify passes, see AZ Code Generator Integration with Waf (p. 131).

Creating a Template Driver in Python

To create a template driver in Python, you must import the `TemplateDriver` base class and override its methods. The code for the class can be found in the `dev/Code/Tools/AzCodeGenerator/Scripts/az_code_gen/base.py` file.

This class is automatically injected into Python by AZ Code Generator and only needs to be imported as `az_code_gen.base`, as in the following example.

```
from az_code_gen.base import *
```

Methods to Override in the TemplateDriver Class

To implement your template driver, override the following methods in the `TemplateDriver` class.

add_dependency

Call the `add_dependency` method to manually add a dependency to the `az_code_gen` task in Waf. The file path given should be absolute so that the render template can specify additional dependencies that Waf does not automatically include. These dependencies might be external data files used to render the templates, or files that were used to generate the input data.

Syntax

```
add_dependency(self, dependency_file)
```

apply_transformations

Override the `apply_transformations` method to manipulate the raw JSON object, which is passed in as the `obj` parameter. Manipulations are performed in place on the object. The object is then forwarded through the pipeline and is eventually passed to `jinja_args` of `render_templates`. Any object returned by this method is provided to the Jinja environment as `extra_data`.

Syntax

```
apply_transformations(self, obj)
```

For an example of this method, see Preprocessing Intermediate Data (p. 143).

get_expected_tags

Override the `get_expected_tags` method to return a list of tags that must be found in any input file. If the required tags are not present, this driver is skipped.

Important
This method is deprecated as of Lumberyard v1.6. After Lumberyard v1.6, all scripts will be processed regardless of expected tags, and `get_expected_tags` will not be invoked.

Syntax

```
get_expected_tags(self)
```

render_template_to_file

Renders a template to disk. This method also adds the value of `output_file` as a dependency of the `az_code_gen` task in Waf.

Syntax

```
render_template_to_file(self, template_file, template_kwargs, output_file,
 should_add_to_build=False)
```

Parameters

Parameter	Description
`template_file`	Specifies the path to a template relative to the directory that contains the template driver `.py` file.
`template_kwargs`	Specifies a dictionary of key–value pairs to be passed to Jinja. Generally this should be treated as a passthrough variable for the `jinja_args` given to `render_templates`, but you can add additional key–value pairs.
`output_file`	Specifies the target file for the rendered Jinja output. The path is relative to the target output folder.
`should_add_to_build`	A Boolean value that specifies whether Waf should add this file to the C++ build and linker. The default is false.

render_templates

Override `render_templates` to invoke template rendering by calling `render_template_to_file`.

Syntax

```
render_templates(self, input_file, **jinja_args)
```

Parameters

Parameter	Description
`input_file`	The path relative to the input path that is used to invoke Clang.

Parameter	Description
`jinja_args`	The raw data from the intermediate JSON object after the template driver performs preprocessing on the object.

Minimal Template Driver

The minimum code required for a template driver is to derive from the `TemplateDriver` base class and implement a factory function to construct the template driver.

```
from az_code_gen.base import *

class MyTemplateDriver(TemplateDriver):
    pass

# Factory function - called from launcher
def create_drivers(env):
    return [MyTemplateDriver(env)]
```

The `az_code_gen` module is automatically provided by AZ Code Generator. It contains the `TemplateDriver` and other useful methods from the `base.py` file.

The `create_drivers` function simply forwards the Jinja environment that is used to render templates. However, you can alter the function to perform other work when the driver is instantiated.

Note
The above bare-bones implementation works but does not generate any output.

Rendering Templates

To generate some output, you must implement the `render_templates` method, as in the following example.

```
from az_code_gen.base import *

class MyTemplateDriver(TemplateDriver):
    def render_templates(self, input_file, **jinja_args):
        self.render_template_to_file("MyTemplate.tpl", jinja_args, 'GeneratedCode.cpp')

# Factory function - called from launcher
def create_drivers(env):
    return [MyTemplateDriver(env)]
```

The `render_templates` method takes the relative `input_file` path and any arguments that were passed in from the `AZCodeGenerator.exe` utility. The `input_file` path usually contains inputs such as the intermediate `json_object` created by Clang.

Template drivers can extend this information by implementing the `apply_transformations` method. For more information, see Preprocessing Intermediate Data (p. 143).

The `render_template_to_file` method takes a template file and argument key–value pairs to pass into the template engine directly and an output path to write the template engine render output to disk.

Configuring Automatic Build Injection

At this point, the example generates a minimal `.cpp` file. The example above does not compile or link the `.cpp` file. This is appropriate if you intend to include the generated code manually using an `#include` in another file.

To inject the generated file automatically, add the `should_add_to_build` parameter to the `render_template_to_file` method and pass the parameter the value of `true`. The `should_add_to_build` parameter informs Waf that the generated file needs to be built and linked into the current target.

Note

Using the `should_add_to_build` parameter is not recommended for header files or other generated files that are not C++ code that must be compiled and linked.

The following example shows some build injected output.

```
from az_code_gen.base import *

class MyTemplateDriver(TemplateDriver):
    def render_templates(self, input_file, **jinja_args):
        self.render_template_to_file("MyTemplate.tpl", jinja_args, 'GeneratedCode.cpp',
 should_add_to_build=True)

# Factory function - called from launcher
def create_drivers(env):
    return [MyTemplateDriver(env)]
```

Preprocessing Intermediate Data

Some cases require preprocessing of the intermediate data for easier consumption by the template engine. To do this, implement the `apply_transformations` method in your template driver. You can use this method to access the intermediate JSON data object directly before it gets passed to `render_templates`. An example follows.

```
from az_code_gen.base import *

class MyTemplateDriver(TemplateDriver):
    def render_templates(self, input_file, **jinja_args):
        self.render_template_to_file("MyTemplate.tpl", jinja_args, 'GeneratedCode.cpp')

    def apply_transformations(self, obj):
        obj['my_custom_data'] = 42

# Factory function - called from launcher
def create_drivers(env):
    return [MyTemplateDriver(env)]
```

For information on the contents of the `obj` variable, see Intermediate JSON Data Format (p. 157).

Custom Code Generator Annotations

AZ Code Generator is in preview release and is subject to change.

You can provide additional data to your template driver by attaching annotations and tags to your source code.

Topics

- Reference Annotations (p. 144)
- Helper Macros (p. 144)
- Example Annotations (p. 144)

Reference Annotations

When you create custom code generator annotations, it is a good idea to refer for examples to the existing annotations in the `dev/Code/Framework/AZCore/AZCore/Preprocessor/CodeGen.h` file. The existing annotations use macros extensively as a workaround for the lack of proper annotations in C++.

Clang provides an `annotate` attribute that can be read at parse time. You can use the helper macros provided to create new annotations, as in the following example.

```
__attribute__((annotate("<Some string here>")))
```

This attribute is wrapped with a macro that converts its contents into strings that can be parsed by the AZ Code Generator utility.

Helper Macros

AZ Code Generator has two helper macros for annotations: `AZCG_CreateAnnotation` and `AZCG_CreateArgumentAnnotation`.

AZCG_CreateAnnotation

`AZCG_CreateAnnotation` is the core macro that exposes the underlying Clang `annotate` attribute. The macro definition follows.

```
// AZCG_CreateAnnotation
#define AZCG_CreateAnnotation(annotation) __attribute__((annotate(annotation)))
```

Any argument passed to `AZCG_CreateAnnotation` must be a string.

AZCG_CreateArgumentAnnotation

The `AZCG_CreateArgumentAnnotation` macro is commonly used for annotation macros. The macro definition follows.

```
// AZCG_CreateArgumentAnnotation
#define AZCG_CreateArgumentAnnotation AZCG_CreateAnnotation(AZ_STRINGIZE(annotation_name)
 "(" AZ_STRINGIZE((__VA_ARGS__)) ")")
```

The `AZCG_CreateArgumentAnnotation` macro takes an `annotation_name` argument and a number of variable arguments. The values passed to the variable arguments are collapsed into a single string for parsing by the AZ Code Generator.

Example Annotations

This section provides example annotations. One example forwards arguments to the underlying macro, one places an annotation inside a class, and one injects code back into the originating file.

Simple Annotation

The following example creates a new annotation called `AzExample` that forwards its arguments to the underlying macro.

```
//Sample Annotation
#define AzExample(...) AZCG_CreateArgumentAnnotation(AzExample, __VA_ARGS__)
```

In this example, the private and public names of the annotation are the same. However, the external and internal names do not have to match.

You can attach the `AzExample` annotation to most items in C++, as in the following example.

```
// Sample Tag Usage
class ExampleClass
{
    AzExample(description("I am data!"))
    int m_myData;
}
```

The tags inside the annotation are placed in JSON format in the generated intermediate data object, as in the following example. Some data has been removed for readability.

```
// Sample Tag JSON
{
    "type": "class",
    "name": " ExampleClass",
    "annotations" : {},
    "fields": [
        {
            "name": "m_myData",
            "annotations" : {
                "description" : "I am data!"
                }
            }
        }
    ]
}
```

Class Annotation Example

The following example directs the AZ Code Generator utility to attach a free-floating annotation to a class.

```
// Class Tag Macro
#define AzExampleClass(...) AZCG_CreateArgumentAnnotation(AzExampleClass, Class_Attribute,
 __VA_ARGS__) int AZ_JOIN(m_azCodeGenInternal, __COUNTER__);
```

`AzExampleClass` – Specifies the annotation name `AzExampleClass` (instead of `AzExample`, as in the previous example).

`Class_Attribute` – Causes the AZ Code Generator utility to attach the attribute to the class that contains the annotation. The annotation belongs to the `annotations` property of the class object.

`__VA_ARGS__` – Specifies additional parameters that are converted into a single string and passed into the AZ Code Generator utility for parsing.

`int AZ_JOIN(m_azCodeGenInternal, __COUNTER__)` – `AZ_JOIN` is a helper macro that takes two macro-level entries and joins them together without converting them to strings. Because Clang requires annotation attributes be attached to a function or variable, this example uses `AZ_JOIN` and a temporary integer member variable to do this. The temporary integer member variable is then ignored.

Adding the new tag to the previous example produces the following code:

```
//Class Tag Example
class ExampleClass
{
    AzExampleClass(MyExampleClassTags::description("I am an example class!"));
```

```
    AzExample(MyExamplePropertyTags::description("I am data!"))
    int m_myData;
}
```

This produces the following intermediate JSON object. Some data has been removed for ease of comprehension.

```
// Class Tag JSON
"type": "class",
"name": "SampleClass",
"annotations" : {
    "MyExampleClassTags::description" : "I am an example class!"
},
"fields": [
    {
        "name": "m_myData",
        "annotations" : {
            "MyExamplePropertyTags::description" : "I am data!"
        }
    }
]
```

Notice that the above JSON does not look exactly like the JSON in the intermediate files provided as part of AZ framework. This is because Lumberyard uses namespaces on its tags to also provide a hierarchy for the tags on its drivers and templates. We recommend that you import the `clang_cpp.py` file and run the `format_cpp_annotations(json_object)` function on the intermediate JSON. When you do, you can use all of the convenient patterns and functions in our drivers and scripts.

The following example shows the same intermediate JSON object after processing by `format_cpp_annotations()`.

```
// Output of format_cpp_annotations()
"type": "class",
    "name": "SampleClass",
    "annotations" : {
        "MyExampleClassTags": {
           "description" : "I am an example class!"
        }
    },
    "fields": [
        {
            "name": "m_myData",
            "annotations" : {
                "MyExamplePropertyTags": {
                    "description" : "I am data!"
                 }
            }
        }
    ]
```

Generated Code Injection Example

The following example shows how to automatically inject generated code back into the original file. The example extends the previously created `AzExampleClass` annotation by injecting code into the example class.

```
// Code Injection Macro
#if defined(AZ_CODE_GENERATOR)
#   define AzExampleClass(ClassName, ...) AZCG_CreateArgumentAnnotation(AzExampleClass,
 Class_Attribute, identifier(ClassName), __VA_ARGS__) int AZ_JOIN(m_azCodeGenInternal,
 __COUNTER__);
```

```
#else
#   define AzExampleClass(ClassName, ...) AZ_JOIN(AZ_GENERATED_CODE_,ClassName)
#endif // AZ_CODE_GENERATOR
```

The updated annotation adds a new required parameter called `ClassName`, which is an identifier that is used to inject the code. The identifier is passed in to Clang as `identifier(ClassName)`, and the data is provided to the intermediate JSON.

Up until this point, the annotation macro outside of `AZ_CODE_GENERATOR` has been blank. The next step is to have it expand to the identifier of the code-generated macro. This causes the generated code to replace the macro annotation when the generated file is put in an `#include` statement.

To implement this, the example sets the macro to become `AZ_JOIN(AZ_GENERATED_,ClassName)`. As before, `AZ_JOIN` in this example renders this as `AZ_GENERATED_CODE_ExampleClass`. The `ClassName` parameter provides a name at compile time for the generated macro.

Note
It is not required that `ClassName` be the actual name of the class where the tag is used. Other tags that use this mechanism can simply require any unique identifier.

When the previous example code is updated, the following code is produced:

```
// Generated Injection Code
class ExampleClass
{
      AzExampleClass(ExampleClass, description("I am an example class!"));
      AzExample(description("I am data!"))
      int m_myData;
}
```

This code produces the following intermediate JSON. Note the new identifier annotation on the class. Some data has been removed for readability.

```
// Generated Code Injection JSON
"type": "class",
"name": "SampleClass",
"annotations" : {
       "AzExampleClass" :  {
              "identifier" : "ExampleClass",
              "description" : "I am an example class!"
       }
},
"fields": [
       {
              "name": "m_myData",
              "annotations" : {
                     "AzExample" : {
                            "description" : "I am data!"
                     }
              }
       }
]
```

This result doesn't compile until the following template code used with the annotation produces the intended macro.

```
// Template Code
{% if class.annotations.identifier is defined %}
#define AZ_GENERATED_CODE_{{ asStringIdentifier(class.annotations.identifier) }}\
public: \
{# This method is injected for all classes with the AzExampleClass tag #}
```

```
bool IsExampleClass(void) { return true; }
{% endif %}
```

This code generates the following code for injection:

```
// Generated Code for Injection
#define AZ_GENERATED_CODE_ExampleClass \
     bool IsExampleClass(void) { return true; }
```

If the generated header is placed in an `#include` statement in the original code, any code in this macro will be injected into `ExampleClass`.

Waf Debugging with AZ Code Generator

> AZ Code Generator is in preview release and is subject to change.

You can debug the integration output of Waf's Python scripts by using PyCharm and a few key debugging entry points. For more information about Waf integration itself, see AZ Code Generator Integration with Waf (p. 131).

Topics

- Prerequisites (p. 148)
- Identifying and Configuring Debug Output (p. 148)
- Setting Up PyCharm for Debugging Waf (p. 149)

Prerequisites

Before you start, follow the instructions for Setting Up PyCharm for Debugging Waf (p. 149). The PyCharm debugger must be set up to debug `lmbr_waf` before you can continue.

Identifying and Configuring Debug Output

All AZ Code Generator Waf integration output is prefixed with `az_code_gen`. To see additional output from both task creation and task execution, add `--zones=az_code_gen` to the Waf command line. This exposes the commands that invoke AZ Code Generator and are useful for debugging the AZ Code Generator utility itself. For more information, see Debugging the AZ Code Generator Utility (p. 155).

Debugging Wscript Configuration

To debug most configuration problems, it is best to set a breakpoint in the `create_code_generator_tasks` method in `Code\Tools\waf-<version>\lmbrwaflib\az_code_generator.py` . This method is called for each `wscript` file that uses the `az_code_gen` feature. It directly interprets the given passes and generates an `az_code_gen` task for each input file in each pass.

Debugging `az_code_gen` Task Creation

The `create_az_code_generator_task` feature creates `az_code_gen` tasks. It gathers most information and inserts it into the task. Each task gets added to the `az_code_gen_group` Waf task to ensure it is executed prior to other tasks.

Debugging az_code_gen Task Execution

The `run` and `handle_code_generator_output` commands are important points in task execution.

The `run` command takes the available information and generates a Clang-style arguments file prefixed with the `@` symbol. The arguments file is passed on the command line to the AZ Code Generator utility.

`handle_code_generator_output` - The AZ Code Generator utility returns a JSON object with one or more entries that are parsed by `handle_code_generator_output`. If the AZ Code Generator utility returns an invalid, non-JSON response due to errors during execution, the Waf task returns the error message `No JSON-Object could be decoded`. To discover the return value that could not be handled, run the command outside of Waf.

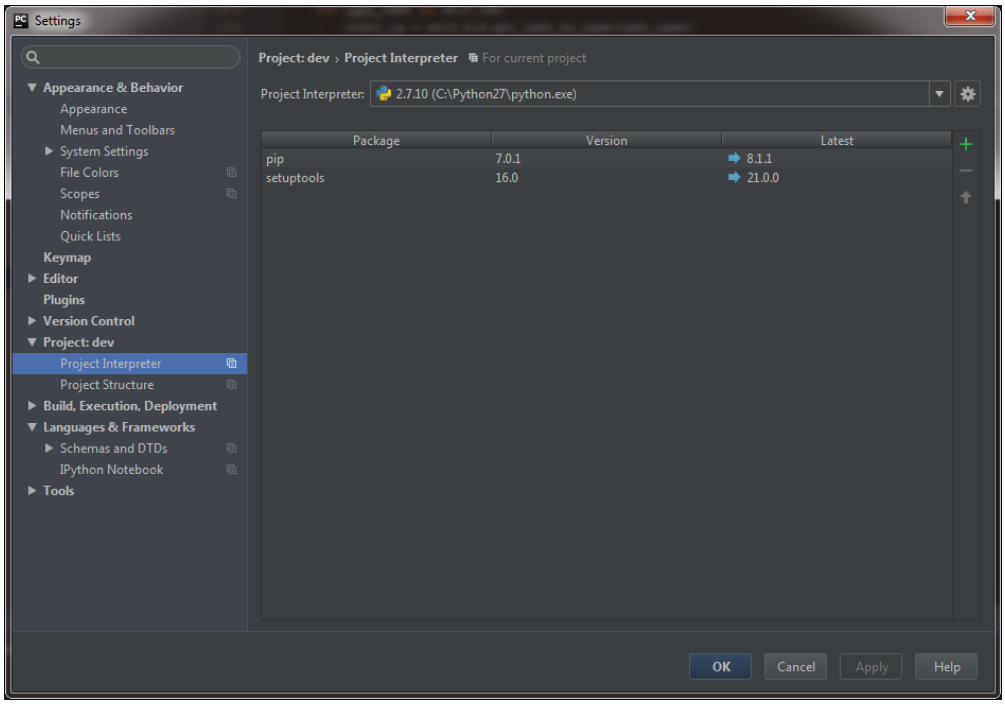
Setting Up PyCharm for Debugging Waf

AZ Code Generator is in preview release and is subject to change.

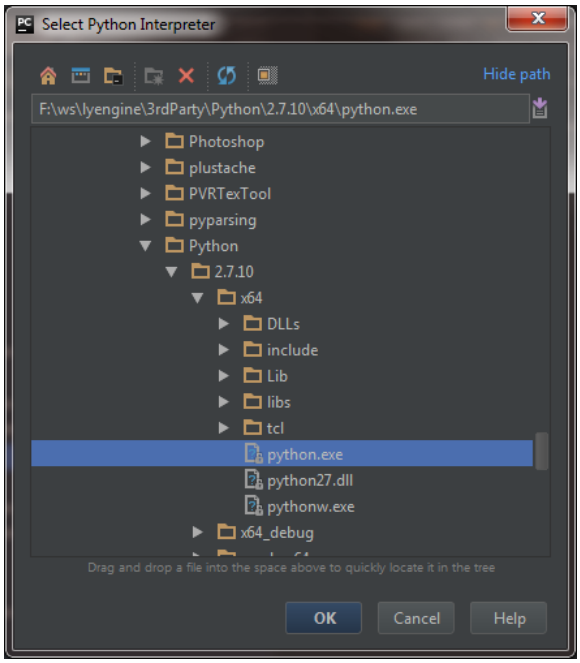
PyCharm is an integrated development environment for Python which includes a graphical debugger that is useful for debugging Waf.

To set up PyCharm and Waf for debugging

1. Download PyCharm Community Edition.
2. Start PyCharm.
3. At the welcome screen, choose **Open Directory**.
4. From the Lumberyard root directory, navigate to any branch that contains a `_WAF_` or `dev` directory. There should be a file called `wscript` and `waf_branch_spec.py` under this folder.
5. Configure the Python interpreter.
   a. Choose **File**, **Settings**, **Project:dev**, **Project Interpreter** to open the project interpreter page.
   b. Click the gear icon on the right of the **Project Interpreter** and choose **Add Local**.

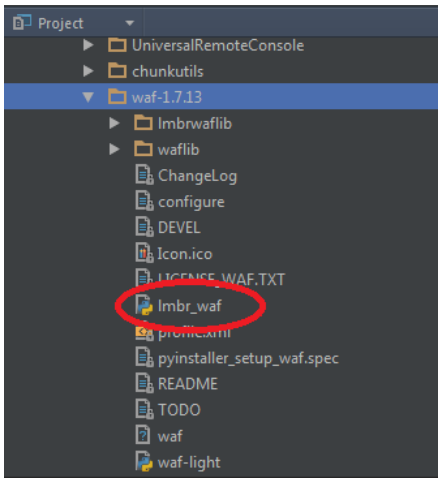


c. Navigate to the folder where `python.exe` resides. The Python executable file must be in the same folder as the project or you may have issues running Waf.



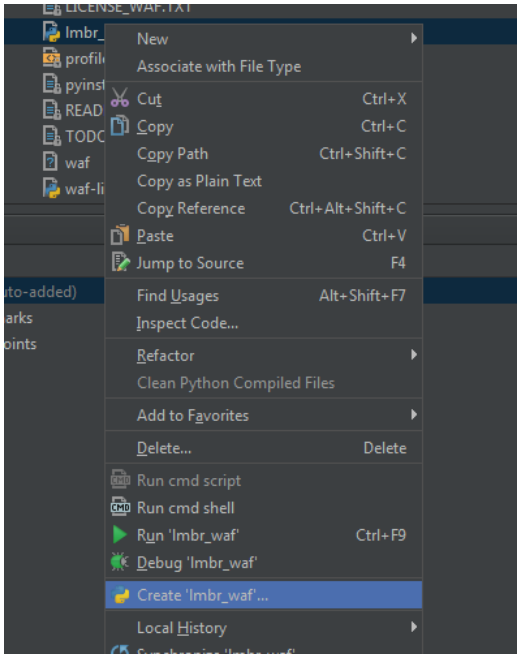
6. Set up a debugging profile for Waf.

   a. To set up Waf for debugging, use the project explorer in the left pane. If you don't see the project explorer, press **Alt+1**). Navigate to the `Code/Tools/waf-<version>` node and expand it. You should see a file called `lmbr_waf` inside this node.



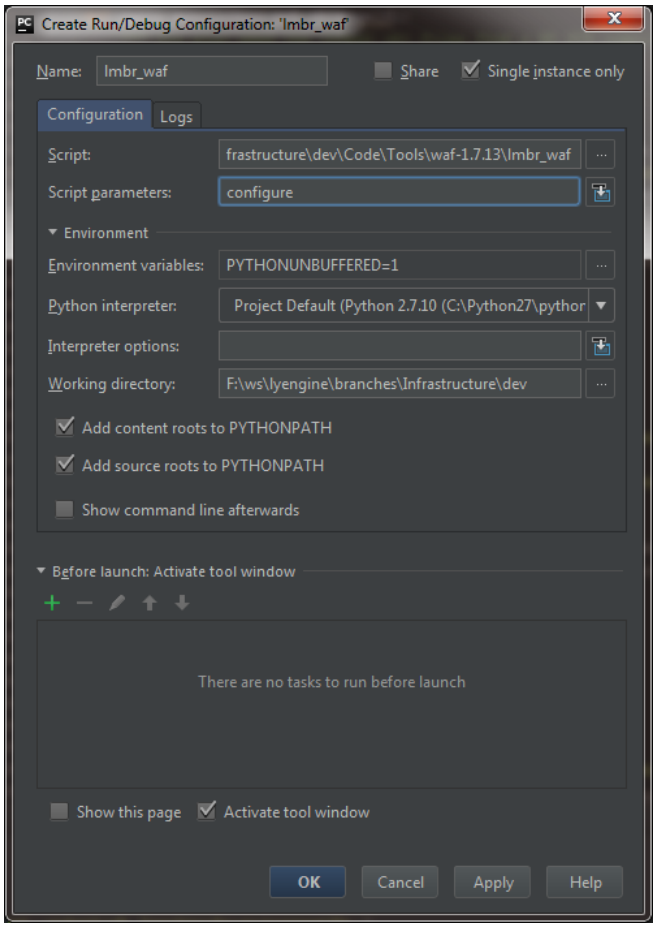
b. Right-click **lmbr_waf** and choose **Create lmbr_waf**

> **Note**
> The **Indexing...** operation must finish before the option appears. You can verify status in the bar at the bottom.



c. In the **Create Run/Debug Configuration** dialog, ensure that the following values are configured correctly:

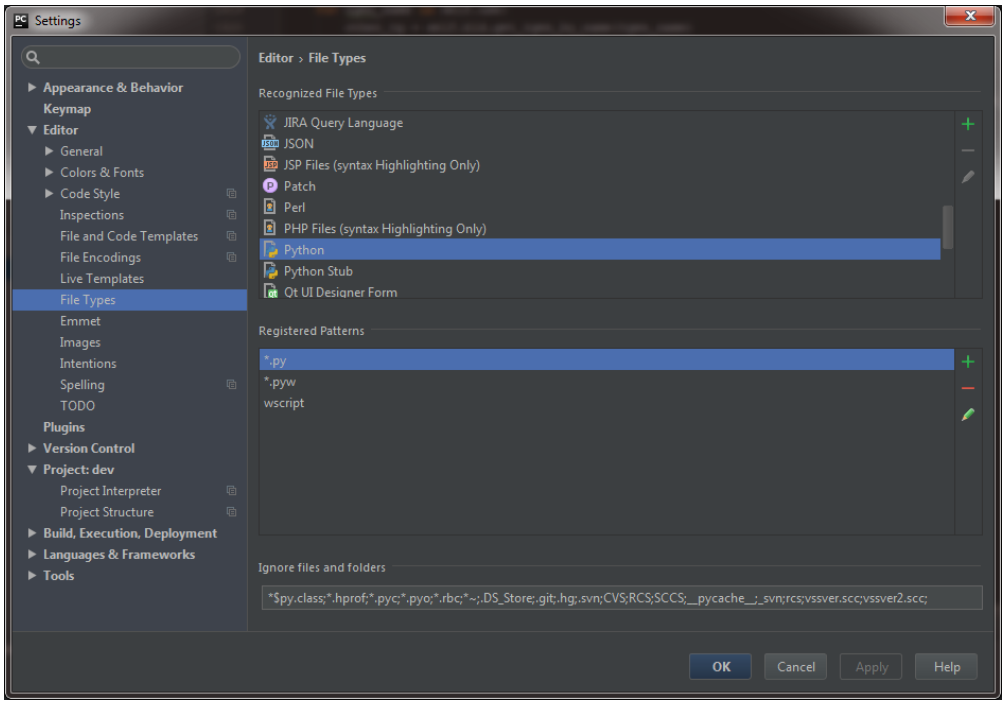
- **Single instance only** should be selected.
- **Script Parameters** is the command to use to run Waf for the run/debug session.
- **Python Interpreter** should be the interpreter that you specified earlier.
- The **Working directory** must be the root of the project (for example, the `dev` directory).



Next, you must to set up `wscript` files as debuggable Python files. Waf uses files called `wscript` to define the build rules per project. These are dynamically loaded Python modules that can be debugged like any other Python module.

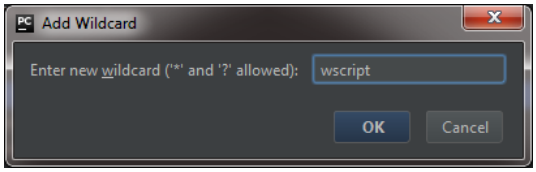
d. Choose **File**, **Settings**, **Editor**, **File Types**, **Python**.

e. To add a registered pattern for **wscript**, choose **Python** in **Recognized File Types**.



f. Under **Registered Patterns**, click the green plus sign (**+**).

g. In the **Add Wildcard** dialog box, type in **`wscript`**.



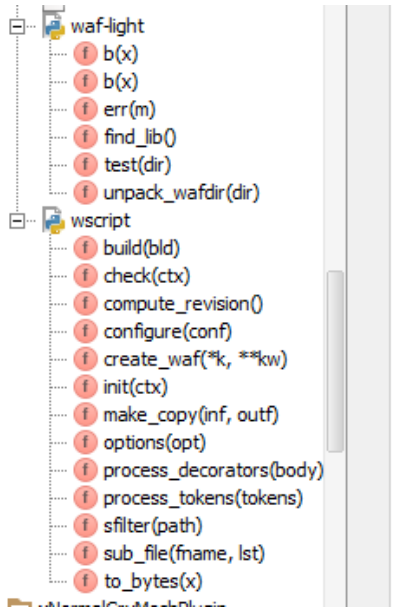
7. Make sure **IncrediBuild** is turned off.

   a. Open the `_WAF_/usersettings.options` file.

   b. Verify that `use_incredibuild` is set to false, as in the following example. `use_incredibuild = False`

8. (Optional) Enable file outlining.

   By default, file outlining is switched off in PyCharm. This feature facilitates navigation in the source files, as the following image shows.



To enable file outlining, right-click the **Project** tab and choose **Show Members**.

Template Driver Debugging

> AZ Code Generator is in preview release and is subject to change.

Because template drivers are run from the AZ Code Generator executable using Python, you can't debug them directly. However, you can debug your driver and template code (and even Jinja2 itself) by using the `debug.py` file included with AZ Code Generator.

To debug a template driver with a Python debugger like PyCharm or Visual Studio

1. Set the debugger to execute the `Bin64\azcg\debug.py` file. This file launches the utility to generate input JSON and emulates a code-generation pass in Python so that you can debug as if you were attached to the utility.
2. Set the working directory to `Bin64\azcg`.
3. Type the arguments for `AzCodeGenerator.exe` into a file with one argument per line. Or use a Waf-generated arguments file as described in Waf Debugging with AZ Code Generator (p. 148).
4. Set the arguments file, prefixed with `@`, as the argument to the script.

   The following arguments are required:

   - `-codegen-script` – Absolute path to the driver script that you want to debug.
   - `-input-path` – Absolute path on which source file paths are based. Usually this path is the same as the location of the wscript for a given target.
   - `-input-file` – Relative path from input path to the source file that is used for input.
   - `-output-file` – Absolute path where generated code will be written.

After you have completed the preceding steps, you should be able to launch your debugger and set breakpoints in your driver script.

For complete AZ Code Generator parameter information, see AZ Code Generator Parameters (p. 133).

Debugging the AZ Code Generator Utility

> AZ Code Generator is in preview release and is subject to change.

When using Waf and the AZ Code Generator utility, you might need to debug Waf Python scripts (p. 148) and your template drivers (p. 154). You can also debug the AZ Code Generator utility itself, although it is less likely to be necessary. You can debug the AZ Code Generator utility by using Visual Studio in Windows or Xcode in macOS.

Topics

- Prerequisites (p. 155)
- Debugging the AZ Code Generator Utility from the Waf build (p. 156)
- Setting Visual Studio Debug Arguments (p. 156)
- Setting Xcode Debug Arguments (p. 156)

Prerequisites

The required preliminary steps depend on your operating system.

Windows Debugging

To debug AZ Code Generator using Visual Studio in Windows, you must generate a Visual Studio HostTools solution (`.sln`) file.

To generate a Visual Studio HostTools solution file

1. Run the following command line from the `dev` folder.

```
lmbr_waf.bat configure --enabled-game-projects= --specs-to-include-in-project-
generation=host_tools --visual-studio-solution-name=HostTools
```

2. In Visual Studio, open the `dev/Solutions/HostTools.sln` file.

macOS Debugging

To enable Waf support for Xcode, perform the following steps to generate an Xcode project.

To generate an Xcode project

1. Open the `dev/_WAF_/specs/all.json` file.
2. Temporarily add `AzCodeGenerator` to `modules`.
3. Run `./lmbr_waf.sh configure` to regenerate the Xcode project.
4. Open the `dev/Solutions/LumberyardSDK.xcodeproj` file.

Debugging the AZ Code Generator Utility from the Waf build

To debug the AZ Code Generator Utility from the Waf build, you must find the arguments file generated by Waf.

Waf generates an arguments file that is passed to AZ Code Generator as a command line parameter. All command line parameters from Waf for AZ Code Generator are contained inside the arguments file. This file is useful for debugging specific Waf AZ Code Generator invocations. To make the arguments file that you use available to Waf, add the `--zones=az_code_gen` option to the Waf command line.

When you use the `--zones=az_code_gen` option, the output looks like the following.

```
lmbr_waf build_win_x64_vs_2013_release -p all --zones=az_code_gen
[   1/3150] az_code_gen (win_x64|release): BinTemp\win_x64_release\Code\Launcher
\WindowsLauncher\GameSDKWindowsLauncherStaticModules.json
14:24:17 az_code_gen Invoking code generator with command: g:\lyengine\Systems\dev
\Bin64\azcg\AzCodeGenerator.exe  @g:\lyengine\Systems\dev\BinTemp\win_x64_release
\CodeGenArguments\arguments_file_ee625f9186107e30ab3126cc30cc9b49.args
```

In this example Waf output, the following is the arguments file.

```
@g:\lyengine\Systems\dev\BinTemp\win_x64_release\CodeGenArguments
\arguments_file_ee625f9186107e30ab3126cc30cc9b49.args
```

Setting Visual Studio Debug Arguments

To set up debugging of AZ Code Generator from Visual Studio, perform the following steps.

To debug AZ Code Generator from Visual Studio

1. Perform the steps to set up Windows debugging as described in Prerequisites (p. 155).
2. In the **Visual Studio Solution Explorer**, right-click **AzCodeGenerator**, and then select **Properties**.
3. Under **Debugging**, paste the path to the arguments file into **Command Arguments**.
4. Click **OK** to close the **Property** window.
5. Right-click **AzCodeGenerator** and then click **Set as StartUp Project**.
6. Press **F5** to launch the debugger.

Setting Xcode Debug Arguments

To set up debugging of AZ Code Generator from Xcode, perform the following steps.

To debug AZ Code Generator from Xcode

1. Perform the steps to set up macOS debugging as described in Prerequisites (p. 155).
2. In Xcode, under the **Product**, **Scheme** menu, choose **AzCodeGenerator**.
3. At the bottom of the **Product**, **Scheme** menu, choose **Edit Scheme**.
4. Under **Arguments,** add a new entry to **Arguments Passed On Launch** that contains your debug arguments.
5. Under **Info**, from the **Executable** drop down, select **Other**.

    a. Navigate to the directory `dev/BinMac64.Debug/azcg/AzCodeGenerator`.

b. Click **Choose**.

6. **Close** the scheme editor.
7. Choose **Product, Run** to launch the debugger.

Intermediate JSON Data Format

AZ Code Generator is in preview release and is subject to change.

The following JSON shows the intermediate data format consumed by Jinja2 user-defined templates.

```
{
    "meta": {
        "path": "<Path/To/Code/Generator/Input/File.ext>"
    },
    "objects": [
        {
            "name": "<Name of class/struct>",
            "qualified_name": "<Fully qualified name of class or struct>",
            "fields": [
                {
                    "type": "<member variable type>",
                    "canonical_type": "<member variable canonical type>",
                    "name": "<member variable name>",
                    "qualified_name": "<fully qualified member variable name>",
                    "annotations": {
                        "<annotation name>": {
                            "<annotation variable name>": "<annotation variable value (can
 be empty string)>",
                            ...
                        },
                        ...
                    }
                },
                ...
            ],
            "traits": {
                "isAbstract": <true if abstract class, false if concrete>,
                "isPOD": <true of plain old data type; otherwise, false>,
                "isPolymorphic": <true if polymorphic type; otherwise, false>
            },
            "bases" : [
                {
                    "name" : "<Base Class Name>",
                    "qualified_name" : "<Fully qualified name of base class>"
                },
                ...
            ],
            "meta": {
                "path": "<Path/To/File/Containing/This/Object.ext>"
            },
            "type": <"class" or "struct">,
            "annotations": {
                "<annotation name>": {
                    "<annotation variable name>": "<annotation variable value (can be empty
 string)>",
                    ...
                },
                ...
            },
```

```
            "methods": [
                {
                    "name" : "<method_name>",
                    "qualified_name": "<Fully qualified name of method>",
                    "is_virtual": <true if virtual method; otherwise, false>,
                    "annotations": {
                        "<annotation name>": {
                            "<annotation variable name>": "<annotation variable value (can
 be empty string)>",
                            ...
                        },
                    "access": "<Access level of method, one of: public, private,
 protected>",
                    "params" : [
                        {
                            "type" : "<parameter type>",
                            "canonical_type" : "<parameter canonical type>",
                            "name" : "<parameter name>"
                        },
                        ...
                    ],
                    "uses_override": <true if override keyword is present; otherwise,
 false>,
                    "return_type": "<return type of method>"
                },
                ...
            ]
        },
        ...
    ]
}
```

AZ Modules (Preview)

AZ modules are in preview release and subject to change.

AZ modules are code libraries designed to plug into Lumberyard games and tools. An AZ module is a collection of C++ code built as a static or dynamic library (`.lib` or `.dll` file) that implements specific initialization functions. When a Lumberyard application starts, it loads each module and calls these initialization functions. These initialization functions allow the module to connect to core technologies such as reflection, serialization, event buses (p. 543), and the Component Entity System (p. 428).

Modules are not a new concept in Lumberyard. In fact, the Lumberyard game engine is a collection of older style modules. These legacy modules have served the game engine well, but they have a number of shortcomings which are addressed by AZ modules, as presented in the next section.

Lumberyard currently supports both legacy modules and AZ modules but going forward will use AZ modules. Beginning in Lumberyard 1.5, a gem can contain AZ module code. Creating a new gem is the easiest way to get up a new AZ module up and running.

Note
AZ is the namespace of the AZCore C++ library upon which AZ modules are built. The letters *AZ* refer to Amazon; the term is a preview name that has nothing to do with Amazon Availability Zones and may be subject to change.

Comparing AZ Modules to Legacy Modules

AZ modules have significant advantages over legacy modules, as the following table shows:

Topic	Legacy Modules	AZ Modules
Compatibility	Modules can be converted to AZ modules with no loss of functionality.	Anything that can be done in a legacy module can also be done in an AZ module. Most AZ module code could live within a legacy module, but legacy modules are not likely to be compatible with future

		AZ module–based Lumberyard tools.
Ease of adding services (singleton classes) to modules	Adding services usually requires editing files in `CryCommon`. A file for the singleton's class interface must exist in the `CryCommon` directory, and a variable to hold the singleton in `gEnv`must exist.	Modules create components and attach them to the system entity. No editing of game engine files is required.
Ease of use for low-level application features	Modules load late, which prevents them from contributing low-level features to an application. All critical features must be in a single module that loads before others.	Modules load early in the application's startup sequence and are initialized in discrete stages. This allows *any* module to provide a low-level feature at an early stage that other modules can take advantage of later.
Exposure of properties	Modules have no uniform way to let users control settings for their service. Some services read settings from `.xml` files in the assets directory, which must be edited by hand.	AZ modules expose the properties of system components to the Lumberyard reflection system. The reflection system makes information about these properties available to all other components.
Game engine dependency	Modules must run in the game engine and are difficult to extend for use in tools that do not have game code.	Modules are not specific to the game engine and can be used outside it.
Initialization functions	Function parameters are specific to CryEngine.	Function parameters are specific to the AZ framework; for more information, see the following section.
Order of initialization	Singleton code often depends on services offered by other singletons, so modules must be initialized in a very particular order. However, the order is not obvious. If someone is unfamiliar with the code in the modules, their loading order is difficult to ascertain.	Each module explicitly states its dependencies on system components. After all system components are examined, they are sorted according to these dependencies and initialized in the appropriate order. Each module is a first-class citizen.

A Self-Aware Method of Initialization

Legacy modules are loaded in a particular order. Because `CrySystem` is loaded and initialized before the game module, it must provide all low-level systems such as logging and file I/O that a subsequent module might depend on. The game module itself cannot provide such low-level systems because it's initialized too late.

AZ modules, on the other hand, are all loaded as early as possible, and then initialized in stages. Because each module explicitly states its dependencies on system components, all system components can be examined beforehand, sorted according to dependencies, and initialized in the appropriate

order (p. 166). This makes it possible for low-level functionality (like a custom logging system) to be implemented from a game module. For more information about the initialization order of components, see The AZ Bootstrapping Process (p. 177).

Relationship with the AZ Framework

AZ modules are designed to work with the AZ framework, which is a collection of Lumberyard technologies such as reflection, serialization, event buses (p. 543), and the component entity system. The AZ framework supports game development but can also be used outside it. For example, Lumberyard tools like the Setup Assistant, Asset Processor and the component entity system use the AZ framework and AZ modules, but contain no game code. When the Resource Compiler builds slices, it loads AZ modules to extract reflection information about components within them.

AZ modules are code libraries that are built to use the AZ framework. When an AZ framework application loads an AZ module, the AZ module knows how to perform tasks such as gathering reflection information about the data types defined within that library.

Smarter Singletons

AZ modules build their services (which are singleton classes) by using the same component entity system that Lumberyard uses to build in-game entities. A module simply places a system component on the system entity. This solves many of the problems associated with singletons in legacy modules.

The GUI in Lumberyard Editor uses the reflection system to expose the properties of entities (gameplay components) to designers. In the same way, Lumberyard uses the reflection system to expose the properties of system components so that you can customize your settings for a particular game. Because system components are really no different from gameplay components, you can use the Project Configurator to edit the properties of system components (p. 173) just as you edit the properties of in-game components.

Current Lumberyard AZ Modules

The gems (p. 168) provided with Lumberyard are all built as AZ modules. In addition, there are two AZ modules that are not built as gems.

LmbrCentral

`LmbrCentral` contains components that wrap functionality from legacy modules. For example, the `MeshComponent` utilizes `IRenderNode` under the hood. `LmbrCentral` is used by game applications.

LmbrCentralEditor

Components can have editor-specific implementations that integrate with technology not available in the game runtime environment. Therefore, a separate module, `LmbrCentralEditor`, is used by Lumberyard Editor. This module contains all the code from `LmbrCentral`, plus code that is only for use in tools. The `LmbrCentralEditor` module is not for use in standalone game applications.

Parts of an AZ Module, Explained

AZ modules are in preview release and subject to change.

An AZ module has three key components: a class that inherits from `AZ::Module`, one or more public facing event buses, and a system component class.

This page describes module initialization, the use of system components as singletons, how EBus calls communicate with this singleton, and how to call the module externally after you have created it.

The Module Class

Each AZ module must contain a class that inherits from `AZ::Module`. When the module is loaded by an application, an instance of the class is created very early in the application's lifetime and its virtual functions are called at the appropriate times as the application goes through its bootstrapping process (p. 177). This class reflects (p. 431) the components declared in the module and adds critical components to the system entity.

The following skeleton code shows the basic structure of an `AZ::Module` class.

```
namespace AZ
{
    /**
     * AZ::Module enables static and dynamic modules (aka LIBs and DLLs) to
     * connect with the running \ref AZ::ComponentApplication.
     *
     * Each module should contain a class which inherits from AZ::Module.
     * This class must perform tasks such as reflecting the classes within
     * the module and adding critical components to the system entity.
     */
    class Module
    {
    public:
        Module();
        virtual ~Module();

        /// Override to require specific components on the system entity.
        virtual ComponentTypeList GetRequiredSystemComponents() const;
    };
}
```

The `AZ::Module` class exposes all points of integration with the AZ framework as virtual functions. These points of integration have been created as virtual functions on a class so that, whether initialization code is in a static or dynamic library, it's written the same way as much as possible. The very first actual initialization calls do need to be different for static and dynamic libraries. Lumberyard provides a macro to define this uninteresting glue code and let you write the interesting initialization code within your `AZ::Module` class.

We recommend that your `AZ::Module` class contain as little implementation code as possible. When the `AZ::Module` class is created, the application is just starting up and many systems are unavailable. If the `AZ::Module` class spawns a singleton or manager class, there is no guarantee that the systems on which this singleton relies will be ready for use. Instead, you should build your singletons as Lumberyard system components (p. 166), which can control their initialization order.

Beginning in Lumberyard 1.5, gems are built using AZ modules. The following example "HelloWorld" AZ module was made by creating a new gem. The `CryHooksModule` class in this example is a helper wrapper around `AZ::Module` and provides your entire module access to `gEnv`.

```
// dev/Gems/HelloWorld/Code/Source/HelloWorldModule.cpp
#include "StdAfx.h"
#include <platform_impl.h>

#include "HelloWorldSystemComponent.h"
```

```
#include <IGem.h>

namespace HelloWorld
{
    class HelloWorldModule
        : public CryHooksModule
    {
    public:
        AZ_RTTI(HelloWorldModule, "{39C21561-D456-413F-8C83-4214F6DBC5A5}",
 CryHooksModule);

        HelloWorldModule()
            : CryHooksModule()
        {
            // Create descriptors for components declared within this module.
            m_descriptors.insert(m_descriptors.end(), {
                HelloWorldSystemComponent::CreateDescriptor(),
            });
        }

        // Add required system components to the system entity.
        AZ::ComponentTypeList GetRequiredSystemComponents() const override
        {
            return AZ::ComponentTypeList{
                azrtti_typeid<HelloWorldSystemComponent>(),
            };
        }
    };
}

// DO NOT MODIFY THIS LINE UNLESS YOU RENAME THE GEM
// The first parameter should be GemName_GemIdLower
// The second should be the fully qualified name of the class above
AZ_DECLARE_MODULE_CLASS(HelloWorld_010c14ae7f0f4eb1939405d439a9481a,
 HelloWorld::HelloWorldModule)
```

The EBus

External code can call into your module, and receive events from your module, through the module's public event buses (p. 543) (EBus). The EBus allows simple and safe function calls between different modules of code.

A new gem comes with one EBus by default, as shown in the following example.

```
// dev/Gems/HelloWorld/Code/Include/HelloWorld/HelloWorldBus.h
#pragma once
#include <AzCore/EBus/EBus.h>
namespace HelloWorld
{
    class HelloWorldRequests
        : public AZ::EBusTraits
    {
    public:
        ////////////////////////////////////////////////////////////////////////
        // EBusTraits overrides
        // These settings are for a "singleton" pattern.
        // A single handler can connect to the EBus.
        static const AZ::EBusHandlerPolicy HandlerPolicy = AZ::EBusHandlerPolicy::Single;
        // A single address exists on the EBus.
        static const AZ::EBusAddressPolicy AddressPolicy = AZ::EBusAddressPolicy::Single;
        ////////////////////////////////////////////////////////////////////////

        // Put your public methods here
```

```
        virtual void SayHello(const char* name) = 0;
    };
    using HelloWorldRequestBus = AZ::EBus<HelloWorldRequests>;
} // namespace HelloWorld
```

Calls to this EBus are handled by the system component, as described in the following section.

The System Component Class

Any major systems in your module that require a singleton should be built as system components. New gems come with a system component by default. The system component class is created during application startup and attached to the system entity (see `GetRequiredSystemComponents()` in `HelloWorldModule.cpp`).

In the current example, the system component class handles calls to the public EBus declared in `HelloWorldBus.h`. The following code shows the `HelloWorldSystemComponent` class.

```
// dev/Gems/HelloWorld/Code/Source/HelloWorldSystemComponent.h
#pragma once
#include <AzCore/Component/Component.h>
#include <HelloWorld/HelloWorldBus.h>

namespace HelloWorld
{
    // The HelloWorldSystemComponent is placed on the system entity
    // and handles calls to the HelloWorldRequestBus.
    class HelloWorldSystemComponent
        : public AZ::Component
        , protected HelloWorldRequestBus::Handler
    {
    public:
        // Every component definition must contain the AZ_COMPONENT macro,
        // specifying the type name and a unique UUID.
        AZ_COMPONENT(HelloWorldSystemComponent, "{72DFB0EE-7422-4CEB-9A40-426F26530A92}");

        static void Reflect(AZ::ReflectContext* context);

        static void GetProvidedServices(AZ::ComponentDescriptor::DependencyArrayType&
 provided);
        static void GetIncompatibleServices(AZ::ComponentDescriptor::DependencyArrayType&
 incompatible);
        static void GetRequiredServices(AZ::ComponentDescriptor::DependencyArrayType&
 required);
        static void GetDependentServices(AZ::ComponentDescriptor::DependencyArrayType&
 dependent);

    protected:
        ////////////////////////////////////////////////////////////////////////
        // AZ::Component interface implementation
        void Init() override;
        void Activate() override;
        void Deactivate() override;
        ////////////////////////////////////////////////////////////////////////

        ////////////////////////////////////////////////////////////////////////
        // HelloWorldRequestBus interface implementation
        void SayHello(const char* name) override;
        ////////////////////////////////////////////////////////////////////////
    };
}
// dev/Gems/HelloWorld/Code/Source/HelloWorldSystemComponent.cpp
#include "StdAfx.h"
#include <AzCore/Serialization/SerializeContext.h>
```

```
#include <AzCore/Serialization/EditContext.h>
#include "HelloWorldSystemComponent.h"

namespace HelloWorld
{
    void HelloWorldSystemComponent::Reflect(AZ::ReflectContext* context)
    {
        // Reflect properties that developers may want to customize.
        if (AZ::SerializeContext* serialize = azrtti_cast<AZ::SerializeContext*>(context))
        {
            serialize->Class<HelloWorldSystemComponent, AZ::Component>()
                ->Version(0)
                ->SerializerForEmptyClass();

            if (AZ::EditContext* ec = serialize->GetEditContext())
            {
                ec->Class<HelloWorldSystemComponent>("HelloWorld", "Says hello")
                    ->ClassElement(AZ::Edit::ClassElements::EditorData, "")
                        ->Attribute(AZ::Edit::Attributes::AppearsInAddComponentMenu,
 AZ_CRC("System"))
                        ->Attribute(AZ::Edit::Attributes::AutoExpand, true)
                    ;
            }
        }
    }

    void
 HelloWorldSystemComponent::GetProvidedServices(AZ::ComponentDescriptor::DependencyArrayType&
 provided)
    {
        provided.push_back(AZ_CRC("HelloWorldService"));
    }

    void
 HelloWorldSystemComponent::GetIncompatibleServices(AZ::ComponentDescriptor::DependencyArrayType&
 incompatible)
    {
        // Enforce singleton behavior by forbidding further components
        // which provide this same service from being added to an entity.
        incompatible.push_back(AZ_CRC("HelloWorldService"));
    }

    void
 HelloWorldSystemComponent::GetRequiredServices(AZ::ComponentDescriptor::DependencyArrayType&
 required)
    {
        // This component does not depend upon any other services.
        (void)required;
    }

    void
 HelloWorldSystemComponent::GetDependentServices(AZ::ComponentDescriptor::DependencyArrayType&
 dependent)
    {
        // This component does not depend upon any other services.
        (void)dependent;
    }

    void HelloWorldSystemComponent::Init()
    {
    }

    void HelloWorldSystemComponent::Activate()
    {
        // Activate() is where the component "turns on".
        // Begin handling calls to HelloWorldRequestBus
```

```
        HelloWorldRequestBus::Handler::BusConnect();
    }

    void HelloWorldSystemComponent::Deactivate()
    {
        // Deactivate() is where the component "turns off".
        // Stop handling calls to HelloWorldRequestBus
        HelloWorldRequestBus::Handler::BusDisconnect();
    }

    void HelloWorldSystemComponent::SayHello(const char* name)
    {
        AZ_Printf("HelloWorld", "Hello %s, you certainly look smashing tonight.", name);
    }
}
```

For more information about system components, see System Components (p. 166).

Calling the Module from External Code

To call your module, invoke your public function through EBus. This example uses the `SayHello` function.

```
#include <HelloWorld/HelloWorldBus.h>

void InSomeFunctionSomewhere()
{
    // ...
 // Invoke the call through EBus.
    EBUS_EVENT(HelloWorld::HelloWorldRequestBus, SayHello, "Bruce");
    // ...
}
```

System Components

> AZ modules are in preview release and subject to change.

A traditional game engine contains many singleton classes, each in charge of a major system. In Lumberyard, these singletons are built using the same component entity system that powers gameplay entities. When an application is starting up, a *system entity* is created. Any components placed on this entity are known as *system components*. The system entity always has the ID `AZ::SystemEntityId (0)`.

When you build singletons as Lumberyard system components, you are using a powerful suite of complementary technologies that facilitate problem resolution through established patterns. This topic describes system components in detail.

Smart Initialization Order

As a game engine grows in size, it tends to develop many singleton classes. A singleton class often requires communication with other singletons to function. This means that the order in which singletons are initialized is very important. Lumberyard solves this by building singletons as components.

A component can declare which services it provides, and it can declare which other services it depends on. When components are activated, they are sorted according to these declared dependencies, ensuring proper initialization order.

The following example shows two components that Lumberyard has ordered for initialization.

```
class AssetDatabaseComponent : public Component
{
    ...

    static void GetProvidedServices(ComponentDescriptor::DependencyArrayType& provided)
    {
        provided.push_back(AZ_CRC("AssetDatabaseService"));
    }

    ...
};

class AssetCatalogComponent : public AZ::Component
{
    ...
    static void GetRequiredServices(AZ::ComponentDescriptor::DependencyArrayType& required)
    {
        required.push_back(AZ_CRC("AssetDatabaseService"));
    }
    ...
};
```

The example shows how `AssetDatabaseComponent` is activated before `AssetCatalogComponent`. In the `AssetDatabaseComponent` class, the `GetProvidedServices` function reveals that the class provides a service called `AssetDatabaseService`. In the `AssetCatalogComponent` class, the `GetRequiredServices` function reveals that `AssetCatalogComponent` depends on `AssetDatabaseService`. Lumberyard understands this dependency and orders the initialization order accordingly.

For more information about the initialization order of components, see The AZ Bootstrapping Process (p. 177).

Easily Configurable Components

Often, a singleton has settings that are configurable for each game. It can be difficult for a low-level singleton to access configuration data because the system used to process this data hasn't yet started. Therefore, low-level singletons often rely on simple data sources such as command line parsers or `.ini` files.

A system component can expose its configuration through AZ reflection. The Advanced Settings dialog box in the Project Configurator (p. 173) uses this feature to enable you to configure system components on a per-game basis. The Project Configurator saves an application descriptor file (p. 176) that contains the settings for each system component, and this file is used to bootstrap the application and configure each component before it is activated. This is the same technology that the Entity Inspector uses to configure gameplay entities in the Lumberyard Editor. For more information, see Configuring System Entities (p. 173).

Writing System Components

To designate a component as a system component, rather than a gameplay component, you must set the `AppearsInAddComponentMenu` field to `System` when you reflect to the `EditContext`.

The following example code designates the `MemoryComponent` as a system component.

```
void MemoryComponent::Reflect(ReflectContext* context)
{
    if (SerializeContext* serializeContext = azrtti_cast<SerializeContext*>(context))
    {
        ...
        if (EditContext* editContext = serializeContext->GetEditContext())
        {
```

```
            editContext->Class<MemoryComponent>("Memory System", "Manages memory
 allocators")
                ->ClassElement(AZ::Edit::ClassElements::EditorData, "")
                    ->Attribute(AZ::Edit::Attributes::AppearsInAddComponentMenu,
 AZ_CRC("System"))
                ...
        }
    }
}
```

Required System Components

Often, a module requires the existence of a system component. This requirement can be established through the module's `GetRequiredSystemComponents()` function. Any component type declared here is guaranteed to exist when the application starts.

In the following example, the `OculusDevice` component is required by the Oculus Gem.

```
AZ::ComponentTypeList OculusGem::GetRequiredSystemComponents() const override
{
    return AZ::ComponentTypeList{
        azrtti_typeid<OculusDevice>(),
    };
}
```

If a system component is optional, you can add it from Advanced Settings in the Project Configurator (p. 173).

Gems and AZ Modules

> AZ modules are in preview release and subject to change.

The gems system was developed to make it easy to share code between projects. Gems are reusable packages of module code and/or assets which can be easily added to or removed from a Lumberyard game. Gems also promote writing code in a way that is more modular than that found in legacy libraries. For example, each gem has its own include folder for its public interface code files. Gems also come with package management metadata such as semantic versioning and the ability to state dependencies on other gems.

Structure of a Gem

A gem's directory contents are organized as follows:

```
GemDirectory/
    Assets/
        (assets usable to projects)
    Code/
        Include/
            (public interface code files)
        Source/
            (private implementation code files)
        Tests/
            (code files for tests)
        wscript (waf build info)
```

```
gem.json (gem metadata)
```

Waf Integration

Each game project must explicitly list the gems that it uses. When the Waf build system runs, it builds only those gems which are actively in use. Waf also makes a gem's `include/` directory accessible to any gems or projects that explicitly depend upon the gem.

Gems Built as AZ Modules

Beginning with Lumberyard 1.5, all gems that ship with Lumberyard are built as AZ modules. When you build a gem as an AZ module, the gem uses the initialization functions expected by the AZ framework. An AZ module gem has public interfaces that are event buses (p. 543) and is better integrated with the new component entity system. Although legacy gems are still supported, it is highly recommended that you use gems based on AZ modules going forward. For information on migrating a legacy gem, see Converting Your Gems.

When you use the Project Configurator to enable or disable a gem, Lumberyard updates the application descriptor file (p. 176) accordingly to ensure it references all AZ modules. If you edit the `dev \<project_asset_directory>\gems.json` list of gems by hand, you can use the following command to bring the application descriptor file up to date:

```
dev\Bin64\lmbr.exe projects populate-appdescriptors
```

About Gem Versioning

The `GemFormatVersion` value is versioning for how a gem is built. Gem version numbers like `0.1.0` refer to the gem's API version.

Gems from Lumberyard 1.4 and earlier (legacy gems) all have a `GemFormatVersion` value of `2`. Starting in Lumberyard 1.5, all the gems included with Lumberyard are AZ modules and have a `GemFormatVersion` value of `3`. This tells Lumberyard that the gem is an AZ module and that it should be loaded accordingly.

A gem may also have an API version number like `0.1.0`. This is independent of the `GemFormatVersion`. The API version alerts your users to API changes. If the API version number changes, then users of the gem may need to make changes to their code. For example, the Rain Gem will stay at version `0.1.0` until its API changes. If you were using the Rain Gem from Lumberyard 1.4, you can still use the Rain Gem from Lumberyard 1.5 without changing any of your data or code.

For more information about gems, see Gems in the Amazon Lumberyard User Guide.

Creating an AZ Module That Is Not a Gem

AZ modules are in preview release and subject to change.

Beginning with Lumberyard 1.5, gems are AZ modules, so the preferred way to build an AZ module is to simply create a new gem. However, if your project requires an AZ module that must not be built as a gem, follow the steps provided here.

A. Start with a Gem

Because gems have all the required code for an AZ module, it's easier to create a gem first and then modify it not to be a gem. As an added convenience, the new gem names the code for you in an

intuitive way. For an explanation of the code that you get in a new gem, see Parts of an AZ Module, Explained (p. 161).

To create and modify a gem

1. First, create a gem by performing the following steps:
   a. Go to your Lumberyard `\dev\Bin64\` directory, then run `ProjectConfigurator.exe`.
   b. Select your project (the default is **SamplesProject**).
   c. Click **Enable Gems**.
   d. Click **Create a New Gem**.
   e. Type the name for your new module. (The example on this page uses the name "HelloWorld".)
   f. Click **Ok**.
2. Move and rename the `code` directory from the new gem to your desired location. For example, move the directory

   `dev/Gems/HelloWorld/Code`

   to

   `dev/Code/<optional subfolder>/HelloWorld`
3. To remove the remaining noncode pieces of the gem, delete the directory `dev/Gems/HelloWorld`.

B. Modify the AZ Module Declaration

AZ modules that are not gems must not have UUIDs in their names, so you must modify the gem's `.cpp` file accordingly.

To modify the `.cpp` file

1. Remove the code that looks like the following:

```
// DO NOT MODIFY THIS LINE UNLESS YOU RENAME THE GEM
// The first parameter should be GemName_GemIdLower
// The second should be the fully qualified name of the class above
AZ_DECLARE_MODULE_CLASS(HelloWorld_010c14ae7f0f4eb1939405d439a9481a,
 HelloWorld::HelloWorldModule)
```

2. Replace the `AZ_DECLARE_MODULE_CLASS` declaration with one that follows this syntax:

```
AZ_DECLARE_MODULE_CLASS(HelloWorld, HelloWorld::HelloWorldModule)
```

   The first argument (`HelloWorld`) is a unique identifier to be included in your `project.json` file, and should match the `target` field of your wscript. You will do these steps later. The second argument is the same fully qualified name of the class already defined in your `.cpp` file.

C. Remove CryEngine References (Optional)

If your module does not access code from CryEngine (for example, it does not access `gEnv`), perform these additional steps.

To remove CryEngine references

1. Make the following changes to your `.cpp` file (in this example, `HelloWorldModule.cpp`).

a. Remove `#include <platform_impl.h>`
b. Remove `#include <IGem.h>`
c. Add `#include <AzCore/Module/Module.h>`
d. Change `HelloWorldModule` to inherit directly from `AZ::Module` instead of from `CryHooksModule`.

2. Remove the following include statement from the `StdAfx.h` file:

```
#include <platform.h> // Many CryCommon files require that this be included first.
```

D. Modify the Wscript and Waf Spec Files

Next, you must modify the default wscript file to remove gem-specific commands, add your module directory to the wscript file, and add your module to the appropriate waf spec files.

To modify the wscript and waf spec files

1. Modify the wscript contents to resemble the following:

```
def build(bld):
    bld.CryEngineModule(
        target          = 'HelloWorld',
        vs_filter       = 'Game', # visual studio filter path
        file_list       = 'HelloWorld.waf_files',
        platforms       = ['all'],
        configurations  = ['all'],
        pch             = ['source/StdAfx.h'],
        use             = ['AzFramework'],
        includes        = ['include', 'source'],
    )
```

2. Modify the wscript in a parent directory so that waf recurses your module's directory, as in the following example.

```
# ...

SUBFOLDERS = [
    # ...,
    'HelloWorld'
    ]

# ...
```

3. To enable waf to build your module, add the module to the appropriate waf spec files in your Lumberyard directory (`dev\_WAF_\specs\*.json`), as in the following example:

```
{
    // ...
    "modules":
    {
        // ...
        "HelloWorld"
    }
    // ...
}
```

E. Configure Your Project to Load the New Module

When your project launches, it loads the modules listed in the `dev/<project_assets>/Config/Game.xml` file (the `Editor.xml` file is used when the Lumberyard Editor is launched). These files are automatically generated and should not be edited by hand.

To configure your project to load your AZ module

1. To ensure your non-gem module is included in these automatically generated lists, add the following lines to your `project.json` file (path location `dev/<project_asset_folder>/project.json`):

```
{
    // ...
    "flavors": {
        "Game": {
            "modules": [
                "LmbrCentral",
                "HelloWorld"
            ]
        },
        "Editor": {
            "modules": [
                "LmbrCentralEditor",
                "HelloWorld"
            ]
        }
    }
}
```

   Note
   The flavors section may be missing from your project. If it is not present, Lumberyard assumes that the `LmbrCentral` module is used for `Game`, and that the `LmbrCentralEditor` module is used for `Editor`.

2. From the dev directory, run the following command from a command prompt.

```
Bin64\lmbr.exe projects populate-appdescriptors
```

   This command modifies the `Game.xml` and `Editor.xml` files to list the `HelloWorld` module.

F. Add the Module's Public Interfaces to Your Project's Include Paths

Finally, to make your AZ module's public interfaces available to the rest of your project, you must inform them project of your module's `include` directory.

To make your AZ modules public interfaces available to your project

- In your project's wscript file, edit the `includes` line to point to your project's include directory, as in the following example.

```
# ...
includes = [..., bld.Path('Code/Engine/HelloWorld/include')],
# ...
```

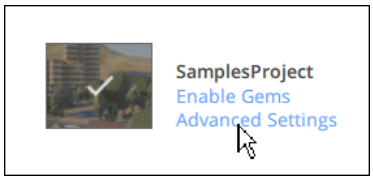
Configuring System Entities

AZ modules are in preview release and subject to change.

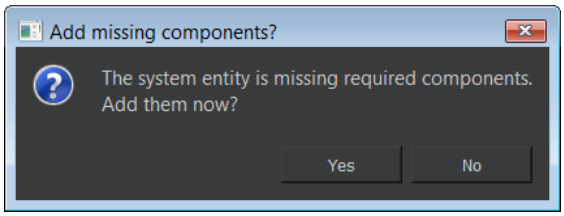
A single *system entity* lives at the heart of every Lumberyard application. This entity's components, known as system components (p. 166), power major systems within Lumberyard. You can use the **Advanced Settings** dialog of the Project Configurator to choose the components for your project and configure them. Editing a system entity is like editing an entity in the Entity Inspector.

To configure system entities

1. Compile a profile build of your project so that the Project Configurator can load your project's compiled code.
2. Go to your Lumberyard `\dev\Bin64\` directory, and then launch `ProjectConfigurator.exe`
3. In Project Configurator, select your project.
4. Click **Advanced Settings**.

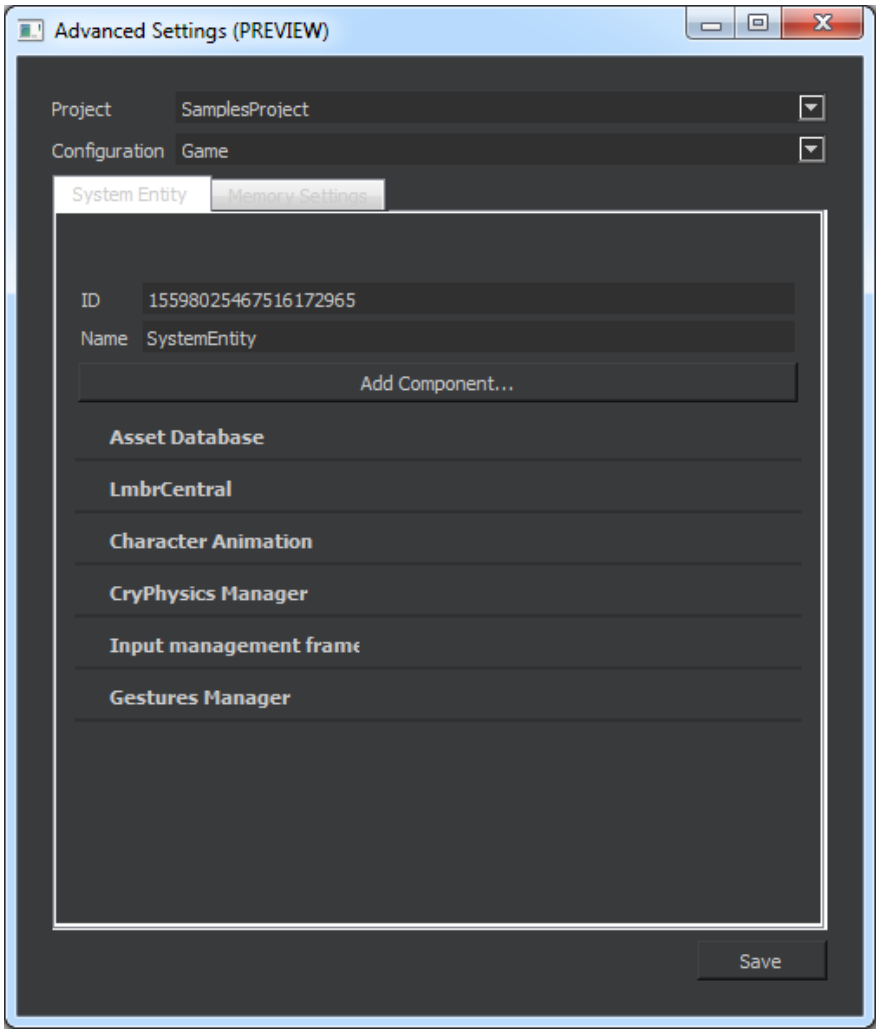


The first time a system entity configuration is loaded, you are prompted to add any required components that are missing from the system entity.



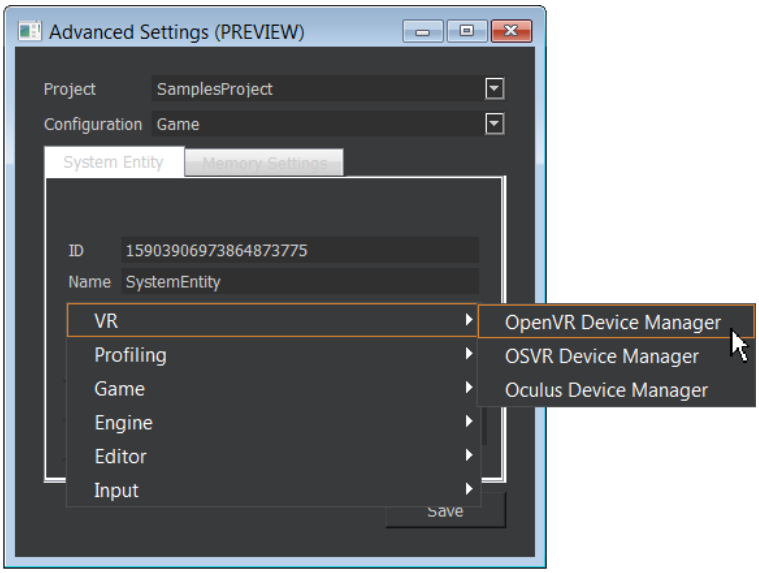
Some system components are optional, and some are required. Both the Lumberyard engine and the gems used by your project may require certain components.

5. Click **Yes**. Even if you decline, the required components are created at run time.
6. Use the **Project** option at the top of the **Advanced Settings** dialog box to select the project that you want to edit. For the **Configuration** option, choose **Game** if you want to make changes to the `Game` (launcher) system entity, or **Editor** to modify the `Editor` system entity.

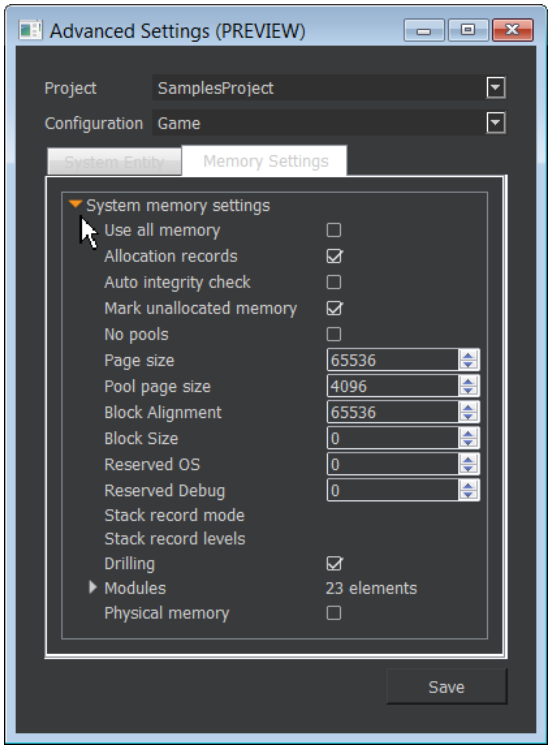


The **System Entity** tab lists components that have been added.

7. Click **Add Component** to select from a variety of components to add.



8. To remove a component, right-click the component in the list and choose **Remove Component "*<ComponentName>*"**.
9. On the **Memory Settings** tab, expand **System memory settings** to configure system memory options.



10. Click **Save** to save your changes to disk. The changes are saved to an application descriptor file, described next.

Application Descriptor Files

When you edit a system entity's configuration by using the **Advanced Settings** dialog box of Project Configurator, you are actually editing an application descriptor file.

Application descriptor files are new to Lumberyard 1.5 and list all modules that a project uses. Currently, each project requires two application descriptor files in its asset directory:

```
dev/<project_asset_directory>/Config/Game.xml
```

```
dev/<project_asset_directory>/Config/Editor.xml
```

In the Project Configurator **Advanced Settings** dialog box, these files correspond to the **Game** and **Editor** options in the **Configuration** menu.

The following example shows the beginning of a `Game.xml` file. Both the `Game.xml` file and the `Editor.xml` file have the same structure.

```
<ObjectStream version="1">
   <Class name="ComponentApplication::Descriptor"
 type="{70277A3E-2AF5-4309-9BBF-6161AFBDE792}">
      <Class name="bool" field="useExistingAllocator" value="false" type="{A0CA880C-
AFE4-43CB-926C-59AC48496112}"/>
      <Class name="bool" field="grabAllMemory" value="false" type="{A0CA880C-
AFE4-43CB-926C-59AC48496112}"/>
      <Class name="bool" field="allocationRecords" value="true" type="{A0CA880C-
AFE4-43CB-926C-59AC48496112}"/>
      <Class name="bool" field="autoIntegrityCheck" value="false" type="{A0CA880C-
AFE4-43CB-926C-59AC48496112}"/>
      <Class name="bool" field="markUnallocatedMemory" value="true" type="{A0CA880C-
AFE4-43CB-926C-59AC48496112}"/>
      <Class name="bool" field="doNotUsePools" value="false" type="{A0CA880C-
AFE4-43CB-926C-59AC48496112}"/>
      <Class name="bool" field="enableScriptReflection" value="true" type="{A0CA880C-
AFE4-43CB-926C-59AC48496112}"/>
      <Class name="unsigned int" field="pageSize" value="65536"
 type="{43DA906B-7DEF-4CA8-9790-854106D3F983}"/>
      <Class name="unsigned int" field="poolPageSize" value="4096"
 type="{43DA906B-7DEF-4CA8-9790-854106D3F983}"/>
      <Class name="unsigned int" field="blockAlignment" value="65536"
 type="{43DA906B-7DEF-4CA8-9790-854106D3F983}"/>
      <Class name="AZ::u64" field="blockSize" value="0" type="{D6597933-47CD-4FC8-
B911-63F3E2B0993A}"/>
      <Class name="AZ::u64" field="reservedOS" value="0" type="{D6597933-47CD-4FC8-
B911-63F3E2B0993A}"/>
      <Class name="AZ::u64" field="reservedDebug" value="0" type="{D6597933-47CD-4FC8-
B911-63F3E2B0993A}"/>
      <Class name="char" field="recordsMode" value="2" type="{3AB0037F-AF8D-48CE-BCA0-
A170D18B2C03}"/>
      <Class name="unsigned char" field="stackRecordLevels" value="5"
 type="{72B9409A-7D1A-4831-9CFE-FCB3FADD3426}"/>
      <Class name="bool" field="enableDrilling" value="true" type="{A0CA880C-
AFE4-43CB-926C-59AC48496112}"/>
      <Class name="AZStd::vector" field="modules" type="{2BADE35A-6F1B-4698-
B2BC-3373D010020C}">
         <Class name="DynamicModuleDescriptor" field="element" type="{D2932FA3-9942-4FD2-
A703-2E750F57C003}">
            <Class name="AZStd::string" field="dynamicLibraryPath" value="LmbrCentral"
 type="{EF8FF807-DDEE-4EB0-B678-4CA3A2C490A4}"/>
         </Class>
[...]
```

The list of system components in the application descriptor file corresponds to the list of components on the **System Entity** tab in the **Advanced Settings** dialog box. Each component can have its own settings. The application descriptor file also contains properties that determine how to allocate memory. These correspond to the settings on the **Memory Settings** tab in the **Advanced Settings** dialog box.

The AZ Bootstrapping Process

AZ modules are in preview release and subject to change.

An AZ framework application initializes modules based on the dynamic libraries listed in the application descriptor file (p. 176), and the static libraries referenced from the `CreateStaticModules()` function.

When an `AzFramework::Application` starts, the following order of events takes place:

1. The executable starts.
2. The `AzFramework::Application` class is initialized. It takes a path to an application descriptor file and a pointer to a function that will create the `AZ::Modules` from static libraries.
3. The application bootstraps itself just enough to read the application descriptor file.
4. The application descriptor file is read to get memory allocator settings and the list of dynamic libraries to load. Lumberyard is not yet able to read the system entity from the file.
5. Lumberyard shuts down the bootstrapped systems, configures them according to the settings it just loaded, and starts these systems back up.
6. Each dynamic library is loaded.
7. Each dynamic library's `InitializeDynamicModule()` function is run, which attaches the DLL to the global `AZ::Environment`.
8. Each static library's `AZ::Module` instance is created using the function pointer passed in during step 2.
9. Each dynamic library's `AZ::Module` instance is created by its `CreateModuleClass()` function.
10. Each AZ module's `RegisterComponentDescriptors()` function is called. Now the application knows how to serialize any components defined within a library.
11. The application descriptor file is read again to extract the system entity along with its components and their settings.
12. Each AZ module's `GetRequiredSystemComponents()` function is called. If any components are missing from the system entity, they are added.
13. The system entity is activated, and all of its system components are activated in the proper order.

At this point, initialization has been completed and the game is running.

Cloud Canvas

Deeply integrated with AWS, Cloud Canvas is a suite of tools and solutions (cloud gems, resource groups, flow nodes) designed to achieve two main goals:

1. Make it easy for you to build cloud-connected features, so you can focus on innovation rather than on undifferentiated backend infrastructure.
2. Make it possible for you to create fantastic new experiences made possible by the availability of the on-demand, global storage and compute provided by AWS.

With Cloud Canvas, you can add a cloud-connected feature to your game in as little as 30 minutes. A single engineer can do this, freeing up the rest of your team to think about innovation and player experience.

Features

Cloud Canvas offers a wide range of helpful components:

- The Cloud Gems Framework, on top of which cloud gems are built, allows you to add pre-packaged, cloud-connected features in a few clicks.
- Tools to enable a team to build a game with cloud-connected features.
- Flow graph nodes to communicate directly from within the client to AWS services such as Amazon S3, Amazon DynamoDB, Amazon Cognito, AWS Lambda, Amazon SQS, and Amazon SNS.
- Tools to manage AWS resources and permissions that determine how developers and players access them.
- Management of AWS deployments so that development, test, and live resources are maintained separately.
- Methods for players to be authenticated (anonymous and authenticated). Players can be authenticated from a variety of devices and access their game data by logging in with an Amazon, Facebook, or Google account.

Example Uses

Consider the many ways you can use Amazon Web Services for connected games:

- Store and query game data such as player state, high scores, or world dynamic content: Amazon S3 and DynamoDB
- Trigger events in real time and queue data for background processing: Amazon SQS and Amazon SNS
- Execute custom game logic in the cloud without having to set up or manage servers: AWS Lambda
- Employ a daily gift system that tracks user visits and rewards frequent visits: Amazon Cognito, Amazon S3, DynamoDB, AWS Lambda
- Present a message of the day or news ticker that provides updates on in-game events: Amazon Cognito, Amazon S3, AWS Lambda

For tutorials on Cloud Canvas, see Tutorial: Getting Started with Cloud Canvas (p. 188) and Lumberyard Tutorials.

Tools

You can access Cloud Canvas functionality by using any of the following:

- **Cloud Gems (p. 200)** – Fully fledged cloud-connected features such as Message of the Day, Leaderboards, and Dynamic Content that can be used in a few clicks as is, or as samples to fuel your custom developments and ideas.
- **Cloud Gem Portal (p. 200)** – A web application that makes backend administration and management accessible to everyone.
- **Flow Nodes** – For designers to leverage the AWS cloud. For detailed information on the Cloud Canvas flow graph nodes, see the Cloud Canvas Flow Graph Node Reference (p. 254).
- **Cloud Canvas C++ APIs** – For software development.
- **Using the Cloud Canvas Command Line (p. 408)** – For managing resource groups, mappings, deployments, and projects.
- **Cloud Canvas Tools in Lumberyard Editor (p. 197)** – For managing AWS resources, deployments, and credentials, and for navigating directly to the AWS consoles supported by Cloud Canvas.

Knowledge Prerequisites

You need the following to take advantage of Cloud Canvas:

- **An understanding of AWS CloudFormation Templates** – Cloud Canvas uses the AWS CloudFormation service to create and manage AWS resources. Our goal is for Cloud Canvas to minimize what you need to know about AWS CloudFormation and AWS in general.
- **Familiarity with JSON** – Cloud Canvas leverages JSON for storing configuration data, including AWS CloudFormation Templates. Currently, you'll need to be familiar with this text format to work with the Cloud Canvas resource management system. A JSON tutorial can be found here.

Topics

- Cloud Canvas Overview (p. 180)
- Cloud Canvas Game Play Design and Engineering Guide (p. 197)
- Cloud Canvas Software Engineering Guide (p. 276)
- Administering Cloud Canvas (p. 381)
- Using the Cloud Canvas Command Line (p. 408)

Cloud Canvas Overview

Cloud Canvas helps you manage cloud resources and connect your game with the AWS cloud. Understanding its concepts will benefit anyone on your team who interacts with the cloud-connected components of your game, including designers, programmers, and testers.

This section covers the following:

- What Cloud Canvas is and how it relates to your AWS account
- The Amazon Web Services that Cloud Canvas supports
- How Cloud Canvas helps you manage your resources
- How your game can communicate with the cloud through the flow graph visual scripting system

Prerequisites

Before reading this topic, you should have a basic understanding of the Lumberyard engine and the flow graph system.

AWS, Cloud Canvas, and Lumberyard

Amazon Web Services (AWS) is an extensive and powerful collection of cloud-based services. You can use these services to upload or download files, access databases, execute code in the cloud, and perform many other operations. A cloud service saves you the trouble of maintaining the infrastructure that it relies on.

Cloud-Based Resources

When you want to use an AWS cloud service, you do so through a resource, a cloud-based entity that is available for your use, help, or support. Resources include a database, a location for storing files, the code that a service runs, and more.

When you create a resource, it exists in the cloud, but you can use it and manage its content. You also specify the permissions that individuals or groups have to access or use the resource. For example, you might allow anyone in the public to read from your database but not write to it or modify it.

Resource Groups

In order to create a connected game feature such as a high score table, you create a resource group in Cloud Canvas. The resource group defines the AWS resources that your feature requires. Each connected game feature therefore is implemented as a resource group in Cloud Canvas.

AWS Accounts

Your resources are owned by an AWS account. The AWS account allows you and your team to share access to the same resources. For example, your team's AWS account might own a database resource so that you and your teammate can both work with the same database.

You, or someone on your team, is an administrator. The administrator creates the AWS account for your team and gives individuals on the team access to the account's resources.

Lumberyard, Cloud Canvas, and Flow Graph

Cloud Canvas is a Lumberyard Gem (extension) that enables your Lumberyard games to communicate with AWS resources. To integrate the communication with Amazon Web Services directly into your game logic, you use Lumberyard's flow graph visual scripting system.

The flow graph nodes that you create use Cloud Canvas to make the actual calls from your game to AWS resources. For example, when a player's game ends, you can add flow graph nodes to submit the player's score to a high score table in the cloud. Later, you can use flow graph to call the high score table to request the top 10 scores.

Amazon Web Services Supported by Cloud Canvas

Several AWS offerings are available through Cloud Canvas that can enhance your game.

File Storage in the Cloud

For storing files in the cloud, Cloud Canvas supports Amazon Simple Storage Service (Amazon S3). Amazon S3 offers a storage resource called a bucket, which you can think of as a large folder. You can build a directory structure in an Amazon S3 bucket just like a directory on a local computer. Amazon S3 buckets have a number of uses in games, including the following:

- Storing files that your game can download. These files can be levels, characters, or other extensions for your game. You can add new files after your game has shipped. Because your game uses Cloud Canvas to download and integrate this content, your customers do not need to download a new client.
- Your game can upload user-generated content. For example, your game might take a screenshot whenever a player beats the last boss. Cloud Canvas uploads the screenshot to your bucket, and your game makes the screenshot available on a website or to other players of the game.

Databases

For storing data like a person's name, phone number, and address in the cloud, Cloud Canvas supports the Amazon DynamoDB database service. Amazon DynamoDB operates on resources called tables. These tables grow and adapt as you build and iterate your game.

Here are some ways in which you can use Amazon DynamoDB table resources in your game:

- Track account details and statistics for a player. Give each player a unique ID so that you can look up a player's hit points, inventory, gold, and friends.
- Add or remove fields to accommodate new resource groups in your game.
- Perform data analyses. For example, you can run complex queries to find out how many players have unlocked a particular achievement.
- Manage game-wide resource groups or events such as a querying a high score table or retrieving a message of the day.

Executing Cloud-Based Logic

For executing code in the cloud, Cloud Canvas supports the AWS Lambda service. AWS Lambda executes resources called functions. You provide the code for a Lambda function, and your game calls the Lambda service through Cloud Canvas to run the function. The Lambda service returns the data from the function to the game.

Your Lambda functions can even call other Amazon Web Services like Amazon DynamoDB and perform operations on their resources. Following are some examples:

- **Submit a high score** – A Lambda function can accept a player's ID and new score, look up the player ID in the database, compare the score with existing scores, and update the highest score if necessary.
- **Sanitize your data** – A Lambda function can check for malicious or unusual input. For example, if a player tries to upload a new high score of 999,999,999 when the best players can't reach 1,000, your Lambda function can intercept the submission and either reject it or flag it for review.

- **Perform server-side authoritative actions** – Cloud Canvas can call your Lambda functions to control in-game economies. For example, when a player tries to purchase an item, your Lambda function can check a database to verify that the player has enough money to pay for the item. The function can then deduct the amount from the player's account, and add the item to the player's inventory.

Identity and Permissions

For managing the identity of the player and controlling access to AWS resources in the cloud, Cloud Canvas supports the Amazon Cognito service.

Amazon Cognito can create unique anonymous identities for your users tied to a particular device. It can also authenticate identities from identity providers like Login with Amazon, Facebook, or Google. This provides your game with a consistent user IDs that can seamlessly transition from anonymous use on a single device to authenticated use across multiple devices. Consider these examples:

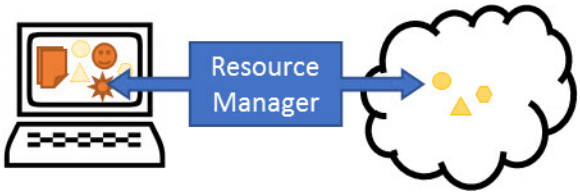
- Players start playing your game anonymously and store their progress locally on their device. Later, to "upgrade" their experience, you ask them to be authenticated through one of the login providers mentioned. After players provide an authenticated ID, you can store their progress in the cloud, and they can access their progress across multiple devices.
- You can specify which AWS resources players are allowed to access. For example, you can enable the "Get the Latest High Scores" Lambda function to be called not only by your game, but by anyone, including external websites. But you could specify that the "Submit High Scores" function only be called by players of your game so that your high score table remains secure. You can use Cloud Canvas to manage these permissions.

Understanding Cloud Canvas Resource Manager

Game development is an inherently local activity. You have a local copy of your game code, assets, and other resources. You build, test, and tweak over and over on your local computer.

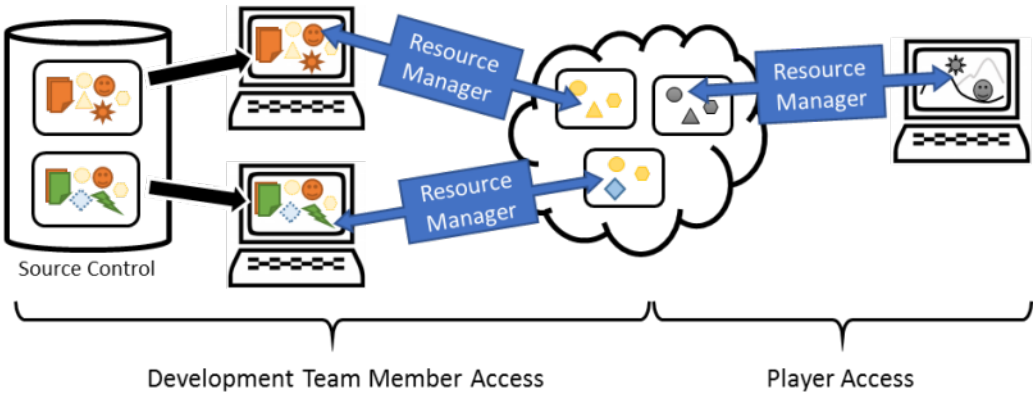
The cloud is different. It is an alien environment. You put resources "out there" that the game depends on. But those resources don't live on your computer system. The process of using and modifying the resources in the cloud isn't the same as for resources that are local.

Cloud Canvas Resource Manager bridges this gap. It lets you have local *descriptions* of the AWS resources in the cloud that your game needs and provides ways to create and interact with the actual instances of those resources in AWS. Your resource could be a database table, a file storage bucket, or code that runs in response to an event.



For team projects, the source code and assets that you are using likely come from a source control system. The changes you make are shared with other people who work on the project through that source control system. Different people can be working at the same time with different versions ("branches") of the code and with different versions of assets without interfering with each other.

When you develop a game that uses cloud resources in AWS, those resources may be shared by different people who work on the game at the same time. Sometimes you need different versions of those resources to exist in the cloud. You also want to ensure that the people developing the game use the version of the resources in the cloud that matches the version of the code and assets they are working with.



After the game is released, the players will use a production copy while your team uses another, private copy to work on bug fixes and new content.

You'll also want to do the following:

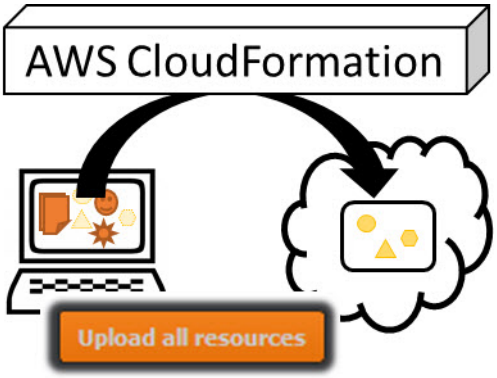
- Be sure that players cannot access the development versions of game resources
- Prevent the development team from making changes that could break the released game
- Protect player information like e-mail addresses from unauthorized access by team members

The **Cloud Canvas Resource Manager** provides the tools you need to do the following:

- Maintain descriptions of the AWS resources that your game depends on
- Create as many copies of the AWS resources as needed for your releases and development teams
- Help you secure access to those resources

The Role of AWS CloudFormation

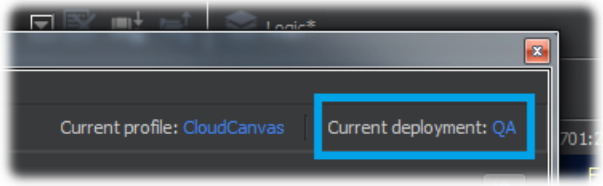
The **Cloud Canvas Resource Manager** integrates the use of AWS CloudFormation into the Lumberyard game development environment. With AWS CloudFormation you can maintain descriptions of the AWS resources you need in text file templates that you can check into your source control system. These descriptions can be branched and merged along with the rest of your game code and assets. When you need actual instances of the resources to be created in AWS, **Cloud Canvas Resource Manager** passes the descriptions to AWS CloudFormation, which uses the template files to create, update, or delete resources in AWS to match the descriptions.



You can use resource manager to organize your descriptions into any number of ***resource groups***. Each group can describe all the resources needed by a game feature, such as a high score tracking system. For details, see Resource Definitions.

With resource manager you can create as many ***deployments*** of the resources as you need. You could have a deployment for the dev team, another for the QA team, and another for the released game, or any other arrangement that suits your needs. Each deployment contains a complete and independent instance of all of the project's resources. Deployments are implemented using AWS CloudFormation ***stack*** resources. For details, see Resource Deployments.

You can choose the deployment that you want to work with in Lumberyard Editor. For example, if you create a "QA" deployment and use it to test your game, Lumberyard Editor automatically maps the references to resources in your game code and Flow Graphs to the "QA" instance of those resources.



Similarly, you can also specify the deployment to be used for release builds of the game. For details, see Resource Mappings.

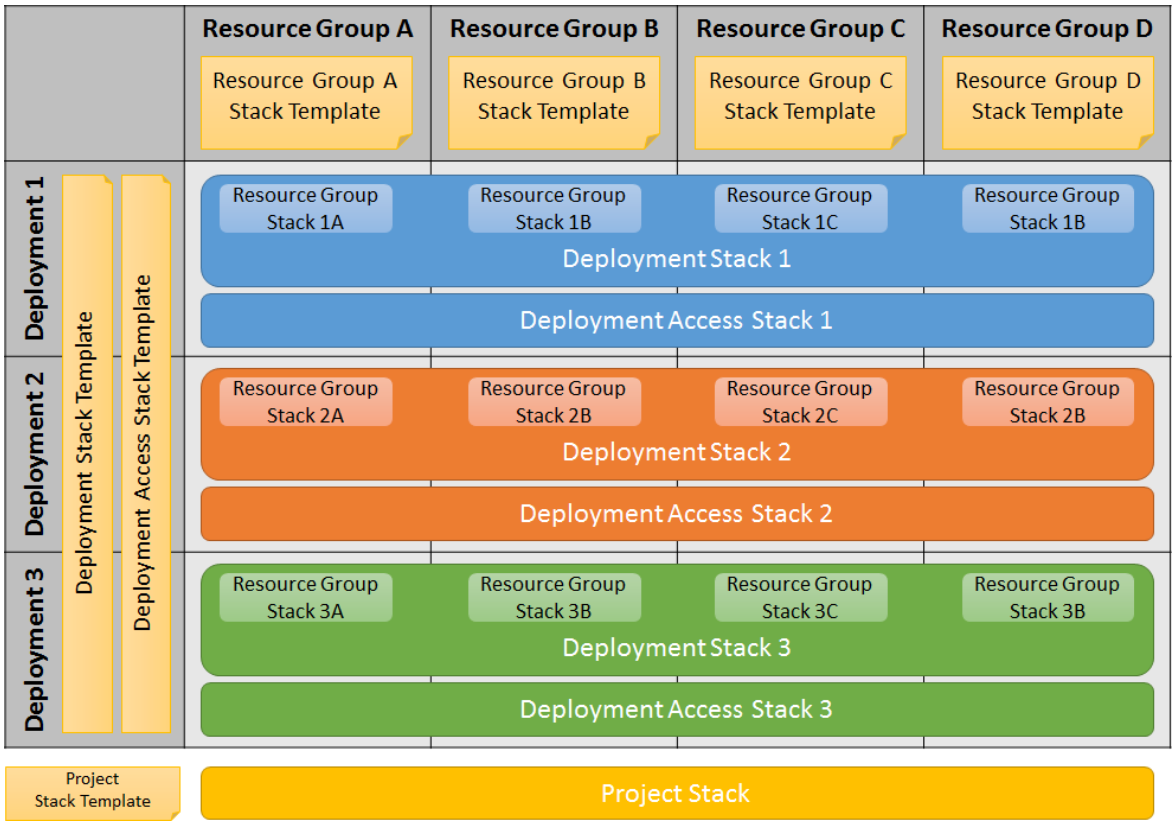
Each deployment comes with an AWS managed policy and an AWS role that you can use to grant specific AWS users and groups access to that deployment. For example, players are granted access to specific resources within a deployment. For details, see Controlling Access to Resources (p. 393).

A Closer Look at AWS CloudFormation Stacks

A Cloud Canvas Resource manager project consists of one or more AWS CloudFormation stacks. All stack templates are stored as files in the project's source control system.

- A single *project stack* that contains the resources that support Cloud Canvas Resource Manager itself. The project stack template is stored as a file in the project's source control system.
- Any number of *deployment stacks*. A deployment represents a complete and independent set of all the resources needed by the game. Each deployment stack contains a child stack for each resource group. All deployment stacks are defined using a single AWS CloudFormation stack template.
- One *deployment access stack* per deployment stack. A deployment's access stack defines the resources used to grant access to a deployment, including IAM Roles and Amazon Cognito Identity Pools. They are distinct from deployment stacks because developers need to be able to update deployment stacks while not being able to change the permissions associated with the deployment.
- Any number of *resource group stacks*. Each resource group represents a set of resources related to an arbitrary game feature. Resource group stacks exists only as children of deployment stacks. Each resource group has its own stack template, which defines the that resource group's stack for each deployment.

So, if there are 3 deployments and 4 resource groups, you have a total of 12 resource group stacks, 3 deployment stacks, 3 deployment access stacks, and 1 project stack (19 stacks total). You'll also have a total of 7 stack templates, one for the project stack, one for all the deployment stacks, one for all the deployment access stacks, and one each for each resource group. The following image illustrates this scenario.



Cloud Canvas Resource Management

In addition to communicating with Amazon Web Services, Cloud Canvas can also help you manage your resources. Amazon Web Services can help create and manage any cloud resources that a game resource group needs. Once you implement the resource group you can use Cloud Canvas deployments to manage the resources for development, test, and live versions of your game.

Defining the Resources

You can create cloud resources by using AWS CloudFormation templates. AWS CloudFormation is an Amazon Web Service with which you can define, create, and manage AWS resources predictably and repeatedly by using templates. The templates are JSON-formatted text files that you use to specify the collection of resources that you want to create together as a single unit (a stack).

In a template, each resource gets its own AWS CloudFormation definition in which you specify the parameters that govern the resource. AWS CloudFormation templates are beyond the scope of this topic, but for now it's enough to understand that you can define (for example) a template with an Amazon DynamoDB table and two AWS Lambda functions. For an example AWS CloudFormation template that creates an Amazon DynamoDB table, see the AWS CloudFormation User Guide.

Deployments

While you are working on a new resource group, your quality assurance team might have to test it. You want to provide a version of your resource group that the test team can use while you continue to work on your own version. To keep the corresponding resources of the different versions distinct, Cloud Canvas gives you the ability to create separate deployments. Deployments are distinct instances of your product's features.

In a scenario like the one described, you might create three deployments: one for the development team, one for the test team, and one for live players. Each deployment's resources are independent of each other and can contain different data because (for example) you don't want the data entered by the test team to be visible to players.

With Cloud Canvas you can manage each of these deployments independently of one another, and you can switch between deployments at will. After making changes, you can use Cloud Canvas to update your feature or deployment and update the corresponding AWS resources.

Team Workflow Using Deployments

The following workflow example illustrates how Cloud Canvas deployments work:

1. The test team finds a bug. You fix the bug in your Lambda code.
2. You switch to the dev deployment and upload the new version of the Lambda function. The Lambda code in the test and live deployments remain untouched for now, and they continue working as is.
3. After you are satisfied that the bug has been fixed, you update the Lambda code in the test deployment. The test team can now test your fix. The live deployment continues unchanged.
4. After the test team approves the fix, you update the live deployment, propagating the fix to your live players without requiring them to download a new version of the game.

Communicating with Cloud Resources using Flow Graph

As your game communicates with its AWS resources, you can use Lumberyard's flow graph system to implement the interaction between your game and AWS. Cloud Canvas-specific flow graph nodes function just like other flow graph nodes, but they make calls to AWS services. For example, if your feature uses two Lambda functions that are needed in different situations, you can use the Lumberyard flow graph system to specify that the functions get called under the appropriate conditions in your game.

You can also use flow graph to take appropriate actions depending on the success or failure of a function. For example, your function might return failure when no Internet connection exists, or when the function lacks sufficient permissions to contact the resource. Your game can parse any failures and handle them appropriately, such as asking the user to retry or skip retrying.

When you have multiple deployments, Cloud Canvas keeps an internal mapping of friendly names to AWS instances so that your game knows which AWS resources to use. Cloud Canvas maps the currently selected deployment to the corresponding set of resources.

Thus, when you release your game to customers, you use a deployment specifically set aside for live players. If you are using the dev version of one feature and switch your deployment to test, your game calls the Lambda function associated with the test deployment.

Managing Permissions Using Cloud Canvas

Managing permissions is an important part of building a secure cloud-connected game. Maintaining separate and distinct permissions is important in the phases of development, testing, and production. You can apply permissions to your development and test teams, to the AWS resources that your game uses, and to the players of your game. A key objective is to secure your game's AWS resources against hackers and other forms of abuse.

You can use permissions to specify exactly who is allowed to do what to the AWS resources that are part of your game. For example, if you have a game feature that uploads screenshots, you can create an Amazon S3 bucket to store the screenshots. You can set permissions for the game to be able to write (send files) to the bucket, but not read from the bucket. This prevents inquisitive users from examining the files that have been uploaded. On the other hand, you can give your team members permissions to read files from the bucket so that they can review and approve them. With Cloud Canvas you can also set

the permissions for individual deployments. For example, live and test deployments can have different permission sets.

Like features, you can define permissions through AWS CloudFormation templates. The permissions are applied any time that you update your cloud resources using the Cloud Canvas resource management tools.

For more information, see Controlling Access to Resources (p. 393).

Cloud Gems Overview

A Lumberyard gem is an individual package of specific functionality and assets. The gem includes everything required for you to include that functionality in your project.

Lumberyard **cloud** gems are Lumberyard gems that provide AWS functionality for a game. A more complete definition of a cloud gem might be the following:

A **cloud** gem is an individual package of specific **cloud-connected** functionality, assets and **AWS resource definitions**. A **cloud** gem includes everything required for you to include that **cloud-connected** functionality in your project.

Cloud Gems Included with Lumberyard

Lumberyard includes the following cloud gems. To enable them in a project, use the Lumberyard Project Configurator tool.

- Message of the Day (p. 203) – Schedules the delivery of messages (for example, new product announcements or holiday greetings) to game clients.
- Leaderboard (p. 211) – Stores a player's high scores and provides leader boards that show player rankings.
- Dynamic Content (p. 220) – Ensures that your clients have the latest game content by automatically downloading new and updated game assets that you upload to the cloud.
- Player Account (p. 241) – Provides a standalone player authentication and management solution for your game that you can customize to fit your needs.

Cloud Gem Portal

The Cloud Gem Portal provides webpages that can be used to configure the cloud gem and perform maintenance operations. The Cloud Gem Portal also provides access to Cloud Canvas Resource Manager functionality, such as deployment management. For more information, see Cloud Gem Portal (p. 200).

Cloud Gem Framework

The Cloud Gem Framework is a software development kit (SDK) that is included with Lumberyard. You can use the framework to implement the cloud gems that come with Lumberyard, as well as your own cloud gems. For more information, see Cloud Gem Framework (p. 305).

Pricing

Cloud Canvas uses AWS CloudFormation templates to deploy AWS resources to your account. Although there is no additional charge for Cloud Canvas or AWS CloudFormation, charges may accrue for using the associated AWS services. You pay for the AWS resources created by Cloud Canvas and AWS CloudFormation as if you created them manually. You only pay for what you use as you use it. There are no minimum fees and no required upfront commitments, and most services include a free tier.

For pricing information on the AWS services that Cloud Canvas supports, visit the following links.

Amazon Cognito Pricing

Amazon DynamoDB Pricing

AWS Lambda Pricing

Amazon S3 Pricing

Amazon SNS Pricing

Amazon SQS Pricing

To see pricing for all AWS services, visit the Cloud Services Pricing page.

Tutorial: Getting Started with Cloud Canvas

This tutorial walks you through the steps of getting started with Cloud Canvas, including signing up for an Amazon Web Services (AWS) account, providing your AWS credentials, and using the command line tools to initialize Cloud Canvas. At the end of the tutorial you will have used your AWS credentials to administer a Cloud Canvas-enabled Lumberyard project.

Specifically, this tutorial guides you through the following tasks:

- Obtain an Amazon Web Services account.
- Navigate the AWS Management Console.
- Create an AWS Identity and Access Management (IAM) user with suitable permissions to administer a Cloud Canvas project.
- Get credentials from your IAM user and type them into the Cloud Canvas tools.
- Use the command line tool to initialize a Lumberyard project for use with Cloud Canvas.
- Dismantle the project, removing all AWS resources that were allocated by Cloud Canvas.

Prerequisites

Before starting this tutorial, you must complete the following:

- Install a working version of Lumberyard Editor.
- Set up a Lumberyard project with the Cloud Canvas Gem (extension) enabled.
- Read through the Cloud Canvas introduction and Cloud Canvas concepts.

Step 1: Sign up for AWS

When you sign up for Amazon Web Services (AWS), you can access all its cloud capabilities. Cloud Canvas creates resources in your AWS account to make these services accessible through Lumberyard. You are charged only for the services that you use. If you are a new AWS customer, you can get started with Cloud Canvas for free. For more information, see AWS Free Tier.

If you or your team already have an AWS account, skip to Step 2 (p. 189).

To create an AWS account

1. Open https://aws.amazon.com/ and then choose **Create an AWS Account**.
2. Follow the instructions to create a new account.

   Note

   - As part of the sign-up procedure, you will receive a phone call and enter a PIN using your phone.

- You must provide a payment method in order to create your account. Although the tutorials here fall within the AWS Free Tier, be aware that you can incur costs.

3. Wait until you receive confirmation that your account has been created before proceeding to the next step.
4. Make a note of your AWS account number, which you will use in the next step.

You now have an AWS account. Be sure to have your AWS account number ready.

Step 2: Create an AWS Identity and Access Management (IAM) User for Administering the Cloud Canvas Project

After you confirm that you have an AWS account, you need an AWS Identity and Access Management (IAM) user with adequate permissions to administer a Cloud Canvas project. IAM allows you to manage access to your AWS account.

AWS services require that you provide credentials when you access them to verify that you have the appropriate permissions to use them. You type these credentials into Lumberyard Editor as part of setting up your project.

The IAM user that you will create will belong to a group that has administrator permissions to install the Cloud Canvas resources and make them accessible through Lumberyard. Administrative users in this group will have special permissions beyond the scope of a normal Cloud Canvas user.

In a team environment, you—as a member of the administrator's group—can create IAM users for each member of your team. With IAM you can set permissions specifically for each person's role in a project. For example, you might specify that only designers may edit a database, or prevent team members from accidentally writing to resources with which your players interact.

For more information on IAM and permissions, see the IAM User Guide.

This section guides you through IAM best practices by creating an IAM user and an administrator group in your account to which the IAM user belongs.

Create an IAM User and an Administrator Group

It's time to create your IAM administrative user.

To create an IAM user in your account

1. Sign into the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane, click **Users**.
3. Click **Add user**.
4. For **User name**, type a user name. This tutorial uses the name `CloudCanvasAdmin`. The name can consist of letters, digits, and the following characters: plus (+), equal (=), comma (,), period (.), at (@), underscore (_), and hyphen (-). The name is not case sensitive and can be a maximum of 64 characters in length.
5. Select the check box next to **Programmatic access**.
6. Select the check box next to **AWS Management Console access**, select **Custom password**, and then type the new password in the text box.

   Note
   When you create a user for someone other than yourself, you can select **Require password reset** to force the user to create a new password when first signing in.
7. Click **Next: Permissions**.
8. Click **Create group**.

9. In the **Create group** dialog box, type the name for the new group. The name can consist of letters, digits, and the following characters: plus (+), equal (=), comma (,), period (.), at (@), underscore (_), and hyphen (-). The name is not case sensitive and can be a maximum of 128 characters in length. This tutorial uses the name *CloudCanvasAdministrators*.
10. In the **Policy name** list, select the check box next to **AdministratorAccess**. This policy provides the necessary permissions for creating and administering a Cloud Canvas project.

    Warning
    The **AdministratorAccess** policy allows almost all permissions within the AWS account and should be attached only to the administrator of the account. Otherwise, other team members could perform actions that incur unwanted charges in your AWS account.
11. Click **Create group**.
12. Back in the list of groups, select the check box for your new group if it is not already selected. If necessary, click **Refresh** to see the group in the list.
13. **Click Next: Review** to review your choices. When you are ready to proceed, choose **Create user**.

    Your IAM user is created along with two important credentials: an access key and a secret access key. Later, you will enter these credentials into Cloud Canvas in order to access the AWS resources in your project.
14. Click **Show** to view your secret access key and password, or click **Download .csv** to download the credentials in a `.csv` file. You can also click **Send email** to receive login instructions by email. Make sure you preserve the credentials in a safe place before you proceed. After this point, you cannot view the secret access key from the AWS Management Console.

    Important
    Do not share your credentials with anyone. Anyone with access to these credentials can access your AWS account, incur charges, or perform malicious acts.
15. You have now created an IAM user called *CloudCanvasAdmin* and a *CloudCanvasAdministrators* administrator group to which the user belongs. To confirm this, click **Groups** in the navigation pane. Under **Group Name**, click *CloudCanvasAdministrators*. The *CloudCanvasAdmin* user appears in the list of users for the group.

In this tutorial, you add only one IAM user to the administrator group, but you can add more if required.

Step 3: Sign in as Your IAM User

Now you're ready to try out your new user.

To sign in as your IAM user

1. Get the AWS account ID that you received when you created your AWS account. To sign in as your *CloudCanvasAdmin* IAM user, use this AWS account ID.
2. In a web browser, type the URL `https://<your_aws_account_id>.signin.aws.amazon.com/console/`, where *<your_aws_account_id>* is your AWS account number without the hyphens. For example, if your AWS account number is *1234-5678-9012*, your AWS account ID would be *123456789012*, and you would visit `https://123456789012.signin.aws.amazon.com/console/`.

   For convenience, you might want to bookmark your URL for future use.
3. Type the *CloudCanvasAdmin* IAM user name you created earlier.
4. Type the password for the user and choose **Sign In**.

You are now successfully signed into the AWS Management Console.

Note
Throughout the tutorial, you must be signed into the AWS Management Console. If you are signed out, follow the preceding steps to sign back in.

Step 4: Enabling the Cloud Canvas Gem (extension) Package

Cloud Canvas functionality is enabled in Lumberyard through a Gem package. Gem packages, or Gems, are extensions that share code and assets among Lumberyard projects. You access and manage Gems through the Project Configurator.

This section of the tutorial shows you how to enable the Cloud Canvas Gem package in a new project.

Enable Cloud Canvas in a New Project

If you are working on a new project, follow these steps to enable Cloud Canvas functionality.

> **Note**
> Adding the Cloud Canvas Gem package to a project that is not already configured requires rebuilding the project in Visual Studio.

To enable Cloud Canvas in a new project

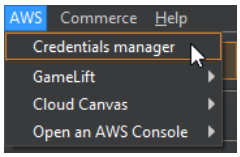
1. Launch `ProjectConfigurator.exe` from your Lumberyard `dev\Bin64\` binary directory.
2. Click **Enable packages** to navigate to the Gems packages screen.
3. Ensure that the check box for the **Cloud Canvas (AWS)** Gem package is checked. If it is already checked, close the **ProjectConfigurator** and go to Step 5: Add Administrator Credentials to Lumberyard (p. 191).
4. Click **Save**, and then close the **ProjectConfigurator**.
5. If you had to add the Cloud Canvas (AWS) Gem to the project, open a command line window and run **`lmbr_waf configure`** to configure your new project.
6. Recompile and build the resulting Visual Studio solution file. Your Lumberyard project is now ready for Cloud Canvas.

Step 5: Add Administrator Credentials to Lumberyard

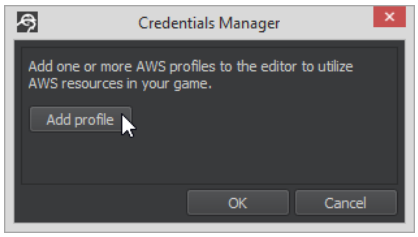
In order to begin managing a Cloud Canvas project, you add the IAM user credentials that you generated earlier to a profile that Cloud Canvas can easily reference. To do this, you can use either Lumberyard Editor or a command line prompt.

To enter your credentials in Lumberyard Editor

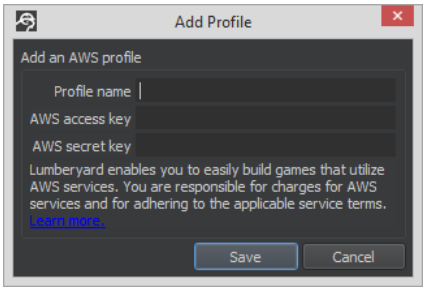
1. In Lumberyard Editor, click **AWS**, **Credentials manager**.



2. In the **Credentials Manager** dialog, click **Add profile**.



3. In the **Add profile** dialog box, enter the information requested. For **Profile name**, type a name of your choice (for example,**`CloudCanvasAdminProfile`**). For **AWS access key** and **AWS secret key**, type the secret key and access key that you generated in Step 2 (p. 189).



4. Click **Save**.
5. In **Credentials Manager**, click **OK**.

To add your credentials by using the command line

1. Open a command line window and change to the root Lumberyard directory, which is the `dev` subdirectory of your Lumberyard installation directory (for example, `C:\lumberyard\dev`).
2. Type the following at the command prompt, and then press **Enter**. Replace *<profile-name>* with a name of your choice (for example, *CloudCanvasAdminProfile*). Replace *<secret-key>* and *<access-key>* with the secret key and access key that you generated in Step 2 (p. 189).

```
lmbr_aws profile add --profile <profile-name> --make-default --aws-secret-key <secret-key> --aws-access-key <access-key>
```

The profile name is now associated with your credentials, and saved locally on your machine in your AWS credentials file. This file is normally located in your `C:\Users\`*<user name>*`\.aws\` directory. As a convenience, other tools such as the AWS Command Line Interface or the AWS Toolkit for Visual Studio can access these credentials.

The profile has also been established as your default profile for Cloud Canvas. The default profile eliminates the need to specify the profile each time you use Lumberyard Editor or run an `lmbr_aws` command.

Important
Do not share these credentials with anyone, and do not check them into source control. These grant control over your AWS account, and a malicious user could incur charges.

You have now created a profile for administering a Cloud Canvas project.

Step 6: Initializing Cloud Canvas from the Command Line

In this step, you configure your Lumberyard project to use Cloud Canvas capabilities. It sets up all of the initial AWS resources required by Cloud Canvas. You perform this step only once for any project.

To initialize Cloud Canvas

1. Open a command line window and change to your Lumberyard `\dev` directory.
2. You must provide Cloud Canvas with the region to which AWS resources will be deployed. Cloud Canvas requires selecting a region that is supported by the Amazon Cognito service. You can check the availability of this service by visiting the Region Table. This tutorial deploys resources to US East (Ohio), which supports Amazon Cognito.

   Type the following command:

```
lmbr_aws project create --region us-east-2
```

The command initializes the contents of the *<root>**<game>*\AWS directory, including a project-settings.json file, and creates the resources Cloud Canvas needs in order to manage your project in your AWS account.

Wait until the initialization process is complete before you proceed. The initialization process can take several minutes.

Note
The initialization process has to be done only once for a given Lumberyard project.

3. You can see the resources created in your AWS account by typing the following command:

```
lmbr_aws project list-resources
```

4. If you are using source control, check in the contents of the *<root>**<game>*\AWS directory so that other users on your team can access the AWS resources.

Your Lumberyard project is now ready to use Cloud Canvas.

Step 7: Locating and Adding Resource Groups

Cloud Canvas lets you organize the AWS resources required by your Lumberyard project into any number of separate resource groups. This step shows you how to add a resource group and optionally add some example resources to your project.

Adding a Resource Group to a New Project

To add a resource group to a new project

1. If you have checked your Lumberyard project into source control, ensure that the *<root>**<game>*\AWS\deployment-template.json file has been checked out and is writeable.
2. Add a new resource group definition by typing the following command:

```
lmbr_aws resource-group add --resource-group Example --include-example-resources
```

After executing this command, the resource definitions for the resource group can be found in the *<root>**<game>*\AWS\resource-group*Example*\resource-template.json file. This file is an AWS CloudFormation template. It will be used to create the AWS resources required by your project in the next step of this tutorial.

3. You can see that the resource group is part of the Lumberyard project by typing the following command:

```
lmbr_aws resource-group list
```

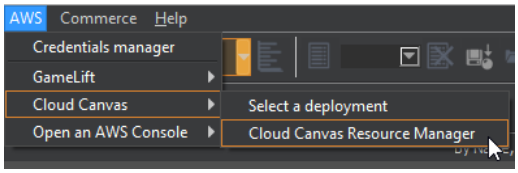
Step 8: Creating Deployments

To create the AWS resources in your AWS account for a project resource group, you create a Cloud Canvas deployment. Cloud Canvas allows you to create any number of deployments. Each deployment will have a complete and independent set of the resources needed by your Lumberyard project. This can be useful when you want to have (for example) separate development, test, and production resources for your game. This step shows you how to create a deployment for a project.

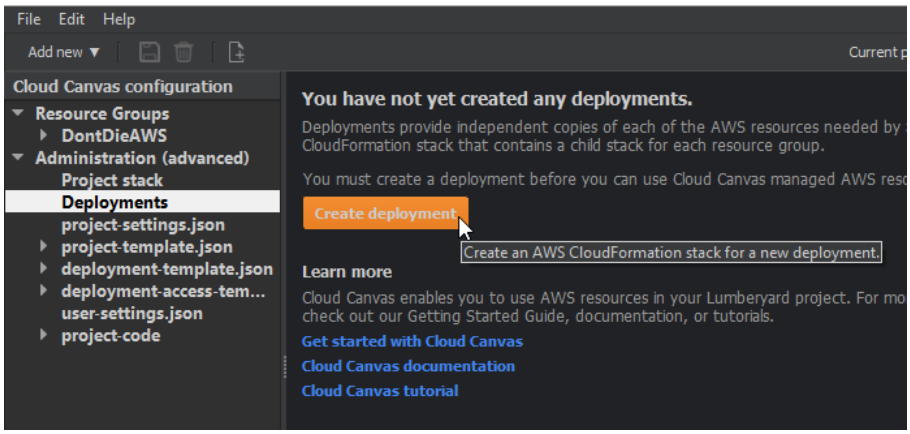
Note
Only project administrators (anyone with full AWS account permissions) can add or remove deployments.

Create a deployment from Cloud Canvas Resource Manager

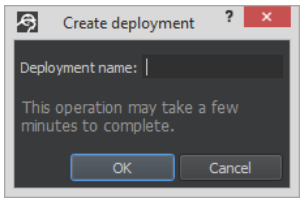
1. If you have checked your Lumberyard project into source control, ensure that the `<root>\<game>\AWS\project-settings.json` file has been checked out and is writeable.
2. In Lumberyard Editor, click **AWS**, **Cloud Canvas**, **Cloud Canvas Resource Manager**.



3. In the **Cloud Canvas configuration** navigation pane, expand **Administration (advanced)**, and then select **Deployments**.
4. In the details pane, click **Create deployment**.



5. In the **Create deployment** dialog, provide a name for the deployment.



   Lumberyard appends the name that you provide to the project stack name to create an AWS CloudFormation stack for the deployment.
6. Click **OK** to start the deployment creation process.

   In the Resource Manager navigation tree, a node for the deployment appears under **Deployments**. In the detail pane, the Viewing the Cloud Canvas Progress Log (p. 279) provides details about the creation process.
7. To make the deployment the default, see Making a Deployment the Default (p. 389).

Create a deployment from the command line

1. If you have checked your Lumberyard project into source control, ensure that the `<root>\<game>\AWS\project-settings.json` file has been checked out and is writeable.
2. Create a deployment by typing the following command:

```
lmbr_aws deployment create --deployment TestDeployment
```

3. You can see that the deployment is now part of the Lumberyard project by typing the following command:

```
lmbr_aws deployment list
```

4. To make the deployment that you created the default deployment in Lumberyard Editor, type the following command:

```
lmbr_aws deployment default --set TestDeployment
```

5. You can see the resources created with the deployment by typing the following command:

```
lmbr_aws deployment list-resources --deployment TestDeployment
```

Step 9: Inspecting Your Resources in AWS

This step in the tutorial shows you the AWS CloudFormation stacks that the previous steps of this tutorial created for you.

To inspect your resources in AWS

1. In a web browser, use your IAM credentials to sign in to the AWS Management Console (see Step 3 (p. 190)).
2. Ensure the AWS region, which appears on the upper right of the console screen, is set to the one that you specified when you had Cloud Canvas deploy its resources in Step 6 (p. 192). If you selected the region in this tutorial, you will see **N. Virginia**.
3. Click **Services**, **CloudFormation**.
4. Note that a number of other stacks have been created as a result of the previous tutorial steps. If a stack update operation is still under way, the stack will show the status **UPDATE_IN_PROGRESS**. Otherwise, the status shows **CREATE_COMPLETE**. You may need to click **Refresh** to update the status.

The next step shows how, as an administrator, you can grant your team members access to Cloud Canvas.

Step 10: Using IAM to Administer a Cloud Canvas Team

In this step, you create Cloud Canvas IAM users for your team, create a group for your users, attach a Cloud Canvas managed policy to the group, and then add the users to the group. This helps you manage your users' access to AWS resources.

The policies that Cloud Canvas creates for your IAM users are much more restrictive than those for an administrator. This is so that your team members don't inadvertently incur charges without administrator approval.

As you add new resource groups and AWS resources to your project, Cloud Canvas automatically updates these managed policies to reflect the updated permissions.

Create IAM users

You start by creating one or more IAM users.

To create IAM users

1. Sign in to the AWS Management Console using your *CloudCanvasAdmin* credentials (see Step 3 (p. 190)).

2. Click **Services**, **IAM**.
3. In the navigation pane, click **Users**.
4. Click **Create New Users**.
5. Type IAM user names for each team member.
6. Be sure that the **Generate an access key for each user** check box is checked.
7. Click **Create**.
8. Choose the option to download the access key and secret access key for each user. The keys for all users that you created are downloaded in a single `.csv` file. Make sure you preserve the credentials in a safe place now. After this point, you cannot view the secret access key from the AWS Management Console. You must deliver each user his or her keys securely.
9. Click **Close**.

Create a group

Next, you create an IAM group for the newly created users.

To create a group for the Cloud Canvas IAM users

1. In the left navigation pane of the IAM console, click **Groups**.
2. Click **Create New Group**.
3. Give the group a name. This tutorial uses the name *`CloudCanvasDevelopers`*.
4. Click **Next Step**.
5. To find the IAM managed policy that Cloud Canvas created for you, click the link next to **Filter** and click **Customer Managed Policies**.
6. Select the check box next to the policy that includes your project name.
7. Click **Next Step**.
8. Review the proposed group that you are about to create.
9. Click **Create Group**.

Add IAM users to a group

Finally, you add your IAM users to the group you just created.

To add your Cloud Canvas IAM users to the group

1. If it is not already selected, click **Groups** in the left navigation pane.
2. Click the name of the newly created *`CloudCanvasDevelopers`* group (not the check box adjacent to it).
3. If it is not already active, click the **Users** tab.
4. Choose **Add Users to Group**.
5. Select the check boxes next to the IAM users that you want to belong to the *`CloudCanvasDevelopers`* group.
6. Click **Add Users**. The team's user names now appear in the list of users for the group.
7. Open the `credentials.csv` file that you downloaded earlier. Securely deliver the secret and access keys to each user in the group. Stress the importance to each user of keeping the keys secure and not sharing them.
8. Have each user in the *`CloudCanvasDevelopers`* group perform the following steps:

    a. In Lumberyard Editor, click **AWS**, **Cloud Canvas**, **Permissions and Deployments**.
    b. Type a new profile name and his or her access and secret access keys.

Important
As an administrator, it is your responsibility to keep your team and your AWS account secure. Amazon provides some best practices and options for how to manage your team's access keys on the Managing Access Keys for IAM Users page. You are encouraged to read this thoroughly.

For information regarding limits on the number of groups and users in an AWS account, see Limitations on IAM Entities and Objects in the IAM User Guide.

Step 11: Remove Cloud Canvas Functionality and AWS Resources

To remove the Cloud Canvas functionality and AWS resources from your project, see Deleting Cloud Canvas Deployments and Their Resources (p. 392).

Cloud Canvas Game Play Design and Engineering Guide

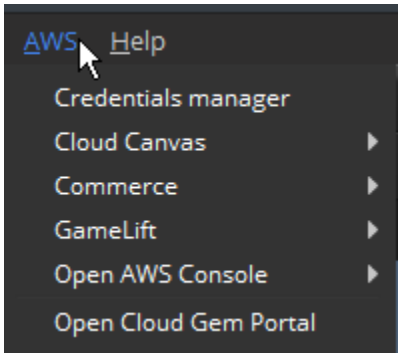
You can use Cloud Canvas's deployments, resource groups, and cloud gems (provided by yourself or others) to design and engineer the cloud connected features of your game.

Topics

- Cloud Canvas Tools in Lumberyard Editor (p. 197)
- Managing Cloud Canvas Profiles (p. 198)
- Using Resource Manager in Game Design (p. 200)
- Cloud Gems (p. 200)
- Cloud Canvas Flow Graph Node Reference (p. 254)

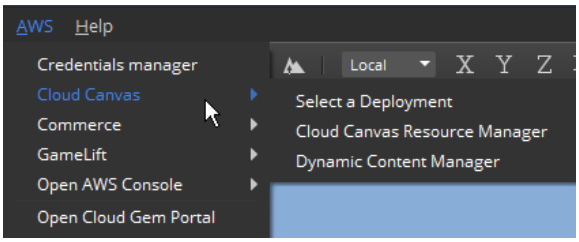
Cloud Canvas Tools in Lumberyard Editor

The tools in Lumberyard Editor make it easy for you to connect your game to AWS. To get started, click **AWS** in the Lumberyard Editor menu bar:

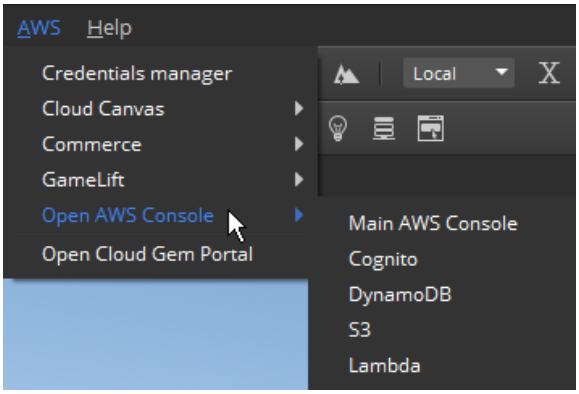


The **AWS** menu has the following options:

- **Credentials manager** – Select or manage one or more AWS profiles that provide credentials to access your AWS account. For more information, see Managing Cloud Canvas Profiles (p. 198).
- **Cloud Canvas** – Choose from among the following options:



  - **Select a Deployment** – Specify the set of AWS resources for the project that you want Lumberyard Editor to work with. For more information, see Making a Cloud Canvas Deployment Active (p. 388).
  - **Cloud Canvas Resource Manager** – Define and manage the AWS resources for your Lumberyard project. For a conceptual introduction to resource manager, see Understanding Cloud Canvas Resource Manager (p. 182).
  - **Dynamic Content Manager** – Manage dynamic content updates for your game through AWS. For more information, see Using Dynamic Content Manager (p. 220).
- **Commerce** – Choose from among the following options for monetizing your game:
  - **Merch by Amazon** – Create themed T-shirts for your game.
  - **Publishing on Amazon** – Distribute your Windows and macOS applications to customers.
- **GameLift** – Use the Amazon GameLift service to rapidly deploy and scale session-based multiplayer games with no upfront costs. For more information, see Amazon GameLift. The **GameLift** menu itself also has links to more information.
- **Open AWS Console** – Get quick access to the main AWS Management Console and to the consoles for Amazon Cognito, DynamoDB, Amazon S3, and Lambda:



  These links use your currently active AWS profile to connect to AWS. You can use **Credentials Manager** to select which profile is active.
- **Open Cloud Gem Portal** – Manage cloud-connected game features that you implement from a cloud gem (p. 187). For more information, see Cloud Gem Portal (p. 200).

Managing Cloud Canvas Profiles

Use the **Credentials Manager** in Lumberyard Editor or the command line to manage one or more AWS profiles that provide the credentials required to access your AWS account.

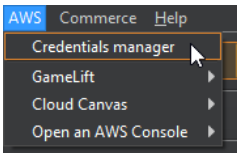
The profile is saved locally on your machine in your AWS credentials file. This file is normally located in your `C:\Users\<user name>\.aws\` directory. The AWS Command Line Interface and the AWS Toolkit for Visual Studio can access these credentials.

> **Important**
> Do not share these credentials with anyone, and do not check them into source control. These grant control over your AWS account, and a malicious user could incur charges.

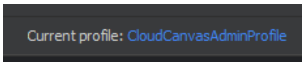
For more information, see **AWS Security Credentials**.

To open Credentials Manager

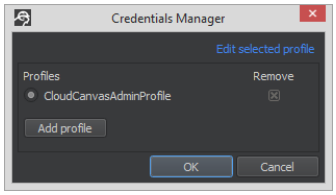
- To open **Credentials Manager**, do one of the following:
  - In Lumberyard Editor, click **AWS**, **Credentials manager**.



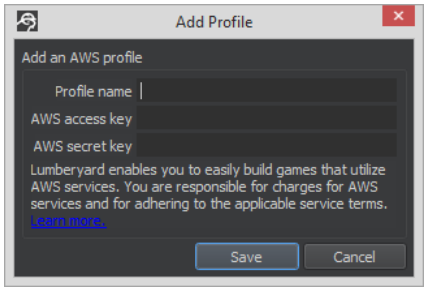
  - In **Cloud Canvas Resource Manager**, click the name of the current profile in the Resource Manager toolbar:



You can use the **Credentials Manager** to select an existing AWS profile, edit an AWS profile, or add a new AWS profile.



To edit an existing AWS profile, click **Edited selected profile**. To add an AWS profile, click **Add profile**.



When adding or editing a profile, Lumberyard prompts you for the following:

Profile name – The name used for the profile.

AWS Secret Key – The AWS secret key needed to access the account.

AWS Access Key – The AWS access key needed to access the account.

To add your credentials by using the command line

1. Open a command line window and change to the root Lumberyard directory, which is the `dev` subdirectory of your Lumberyard installation directory (for example, `C:\lumberyard\dev`).
2. Type the following at the command prompt, and then press **Enter**. Replace *<profile-name>* with a name of your choice (for example, *CloudCanvasAdminProfile*). Replace *<secret-key>* and *<access-key>* with the secret key and access key of your AWS account.

```
lmbr_aws profile add --profile <profile-name> --make-default --aws-secret-key <secret-key> --aws-access-key <access-key>
```

The `--make-default` option establishes the profile as your default profile for Cloud Canvas. The default profile eliminates the need to specify the profile each time you use Lumberyard Editor or run an `lmbr_aws command`.

Using Resource Manager in Game Design

You can use Cloud Canvas Resource Manager to make deployments active, test mappings, use protected deployments, and to delete deployments and resources.

For information, see the following links in the Working with Deployments (p. 385) section of the Cloud Canvas documentation.

- Making a Cloud Canvas Deployment Active (p. 388)
- Testing Different Mappings (p. 390)
- Using Protected Deployments (p. 391)
- Deleting Cloud Canvas Deployments and Their Resources (p. 392)

Cloud Gems

A cloud gem is an individual package of specific cloud-connected functionality, assets, and AWS resource definitions. A cloud gem includes everything required for you to add cloud-connected functionality to your project.

Lumberyard includes the following cloud gems. You can enable them in a project from the Lumberyard Project Configurator tool.

- Message of the Day (p. 203) – Schedules the delivery of messages (for example, new product announcements or holiday greetings) to game clients.
- Leaderboard (p. 211) – Stores a player's high scores and provides leader boards that show player rankings.
- Dynamic Content (p. 220) – Allows Lumberyard `.pak` files that contain new and updated game assets to be uploaded to the cloud for later automatic downloading to the game client.

Topics

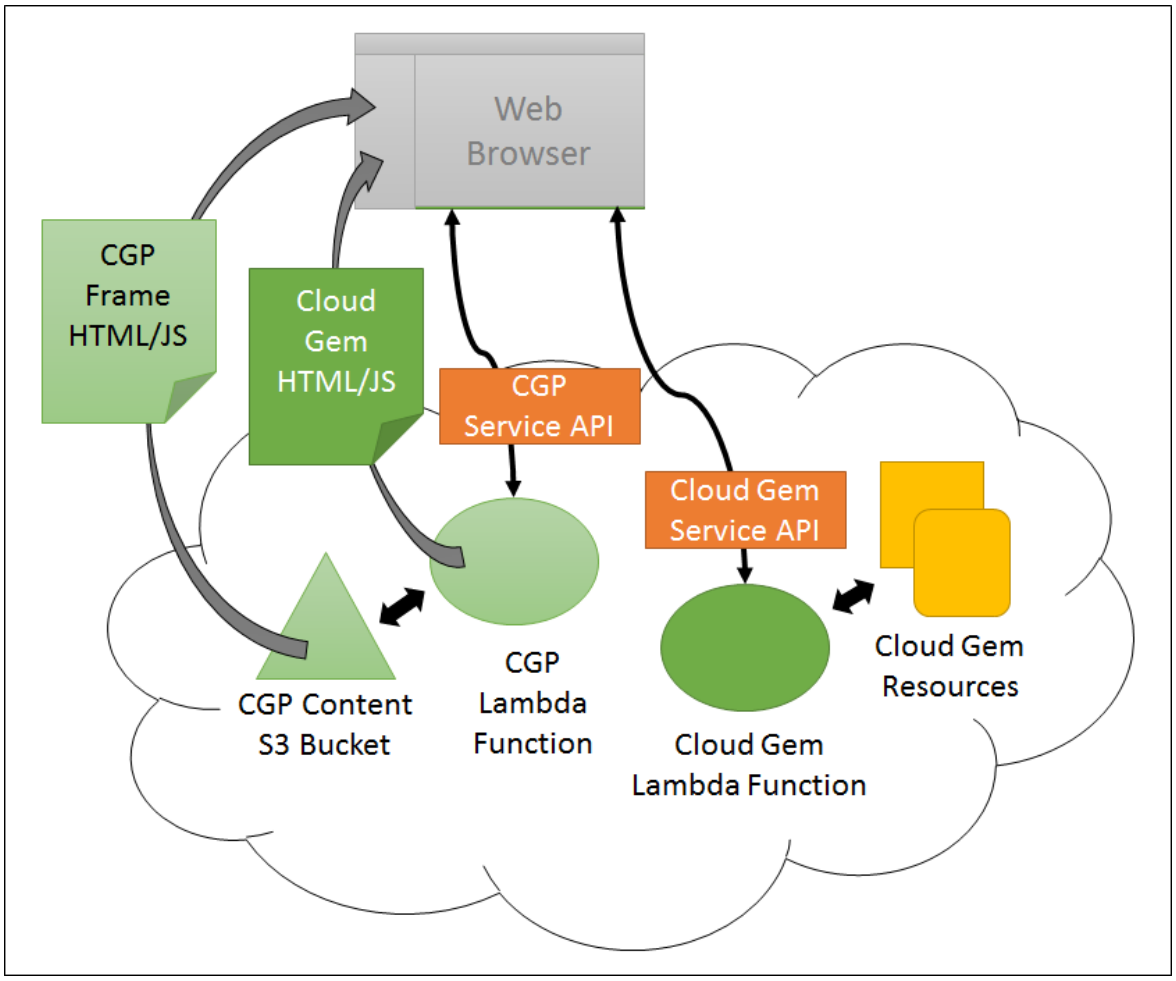
- Cloud Gem Portal (p. 200)
- Setting Up and Accessing the Cloud Gem Portal (p. 201)
- Message Of The Day Cloud Gem Portal (p. 203)
- Leaderboard Cloud Gem Portal (p. 211)
- Using Dynamic Content Manager (p. 220)
- Player Account Cloud Gem (p. 241)

Cloud Gem Portal

A cloud gem (p. 187) is a gem in Lumberyard whose scripts and assets use AWS resources to implement cloud-connected game features. A cloud gem creates a Cloud Gem Portal (CGP) that you can use to manage the cloud data for your game. For example, the Leaderboard Cloud Gem Portal (p. 211) can provide a way for a support person to remove fraudulent high scores. The Cloud Gem Portal is hosted in your AWS account and is a website that you own. The Cloud Gem Portal is part of the Cloud Gem Framework (p. 305) and can be customized.

How a Cloud Gem Portal Works

Cloud gem portals use Amazon Web Services to implement their functionality, as shown in the following diagram.



Amazon S3 stores a Cloud Gem Portal as a web application. AWS Lambda functions implement the required server-side logic. JavaScript code that runs in the web browser accesses the Lambda functions through the Amazon API Gateway. The configuration for the API Gateway is handled by the Cloud Canvas Cloud Gem Framework Service API (p. 352).

Cloud Gem Workflow

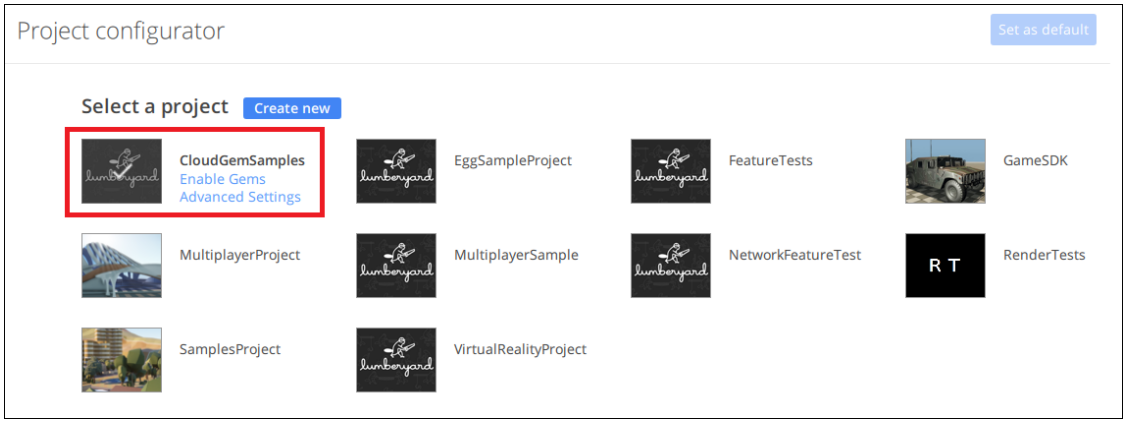
A cloud gem provides content (for example, HTML, JavaScript, or images) from its `/dev/Gems/<gem>/AWS/cgp-resource-code` subdirectory. When you deploy a cloud gem, the Cloud Canvas Resource Manager (p. 182) uploads this content to the project's configuration bucket. To determine which cloud gems are available to it, the Cloud Gem Portal page uses the Cloud Gem Portal service API to call the Cloud Gem Portal Lambda function. When you select a cloud gem, its content is loaded into the Cloud Gem Portal webpage. To query the state of the cloud gem or perform other operations, JavaScript code can use a service API defined for the cloud gem. The service API calls the cloud gem's Lambda function that implements the cloud gem's functionality.

Setting Up and Accessing the Cloud Gem Portal

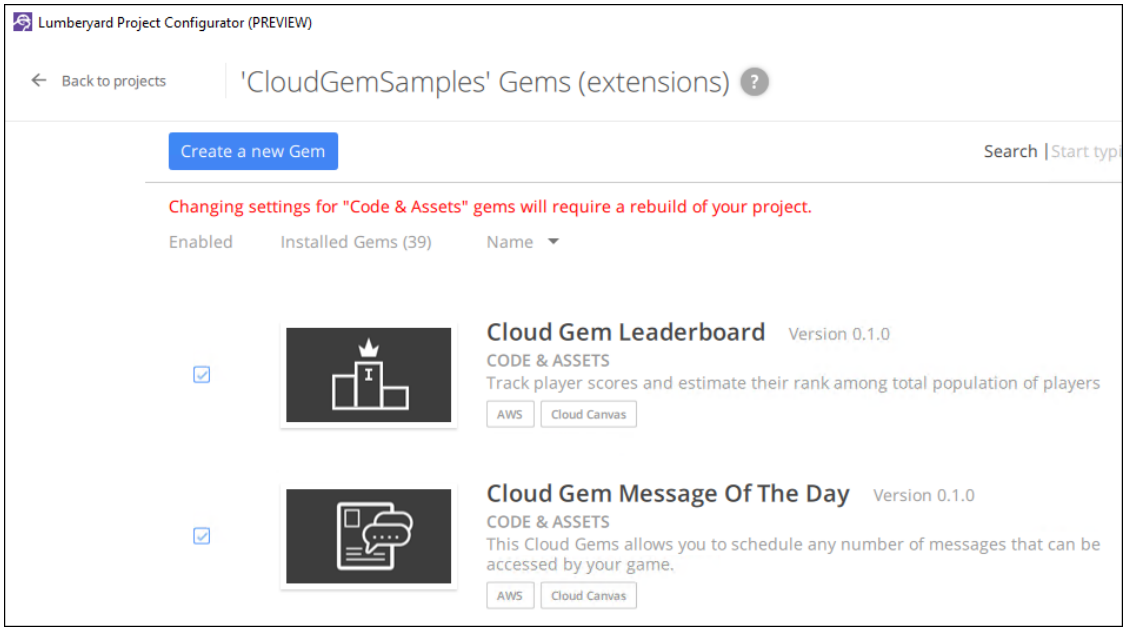
Before you can use a Cloud Gem Portal, you must install one or more cloud gems.

To add a cloud gem to your project

1. Open the Project Configurator.
2. Click **Enable Gems** next to the project for which you want to enable the new gems. In this example, the **CloudGemSamples** project is selected.



3. Select the cloud gems that you want to enable. In this example, the **Cloud Gem Leaderboard** and **Cloud Gem Message of the Day** gems are selected.



4. After you have enabled a cloud gem, you must build the project so that the `.dll` libraries for the gem are included. For information on building your project, see Compiling Game Code.

Accessing the Cloud Gem Portal

To access the Cloud Gem Portal, you can use either Lumberyard Editor or a shared link.

To access the Cloud Gem Portal through Lumberyard Editor, click **AWS**, **Open Cloud Gem Portal**. Your default browser opens a link to the Cloud Gem Portal page. Clicking **Open Cloud Gem Portal** in Lumberyard Editor also generates a link to the webpage with a unique key-pair that you can share. Apart from Lumberyard Editor, this link is the only way to access your project's Cloud Gem Portal. To share the link, click the share icon next to the project stack name in the upper-left section of the Cloud Gem Portal navigation bar. When you generate a link, all previously generated links cease to function. Because this

shareable link is temporary, we recommend that you always use Lumberyard Editor to launch the Cloud Gem Portal.

Note
You can override the session length by going to your Lumberyard installation directory and executing the following command.

```
lmbr_aws cloud-gem-framework cloud-gem-portal --duration-seconds 7200
```

Message Of The Day Cloud Gem Portal

You can use the Message of the Day Cloud Gem to schedule messages that you game consumes. You can customize your project's message of the day using the Cloud Gem Portal.

Prerequisites

This tutorial assumes the following:

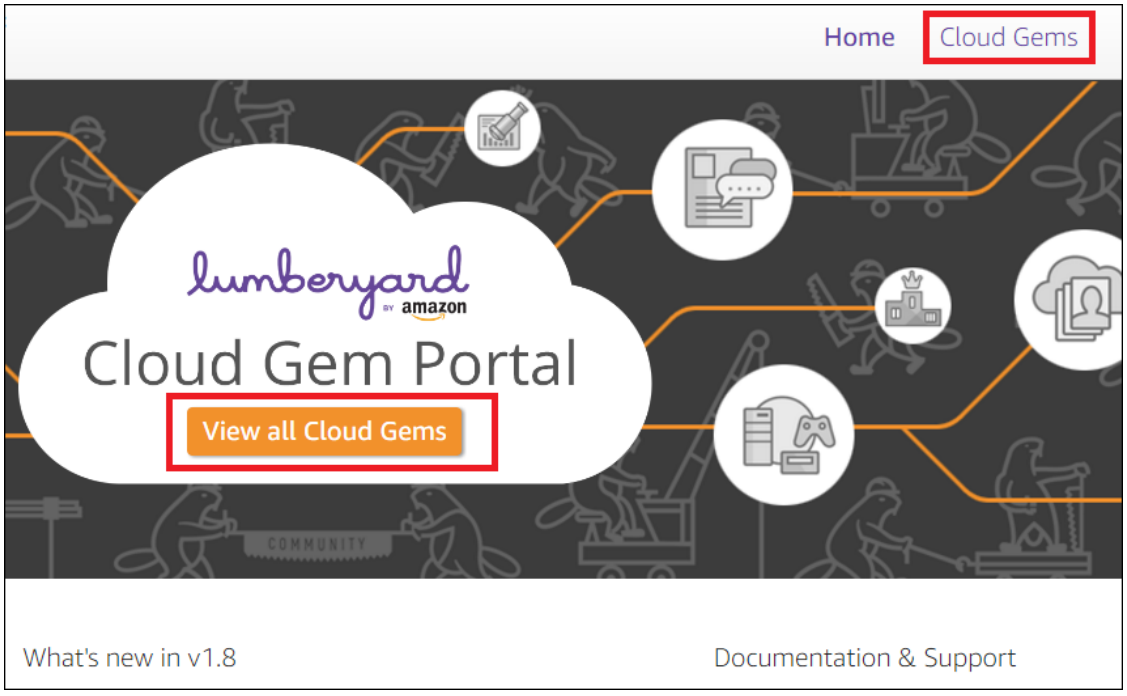
- You are using a Lumberyard project that has the Message of the Day Cloud Gem enabled (in Project Configurator, select **Cloud Gem Message of the Day**).
- You have created a project stack in Cloud Canvas Resource Manager (p. 182).
- You have created a deployment stack in resource manager.
- You have opened the cloud gem portal. In Lumberyard Editor, click **AWS**, **Open Cloud Gem Portal**.

If you don't meet the prerequisites, follow the steps in Setting Up and Accessing the Cloud Gem Portal (p. 201) to add **Cloud Gem Message of the Day** in the Project Configurator.

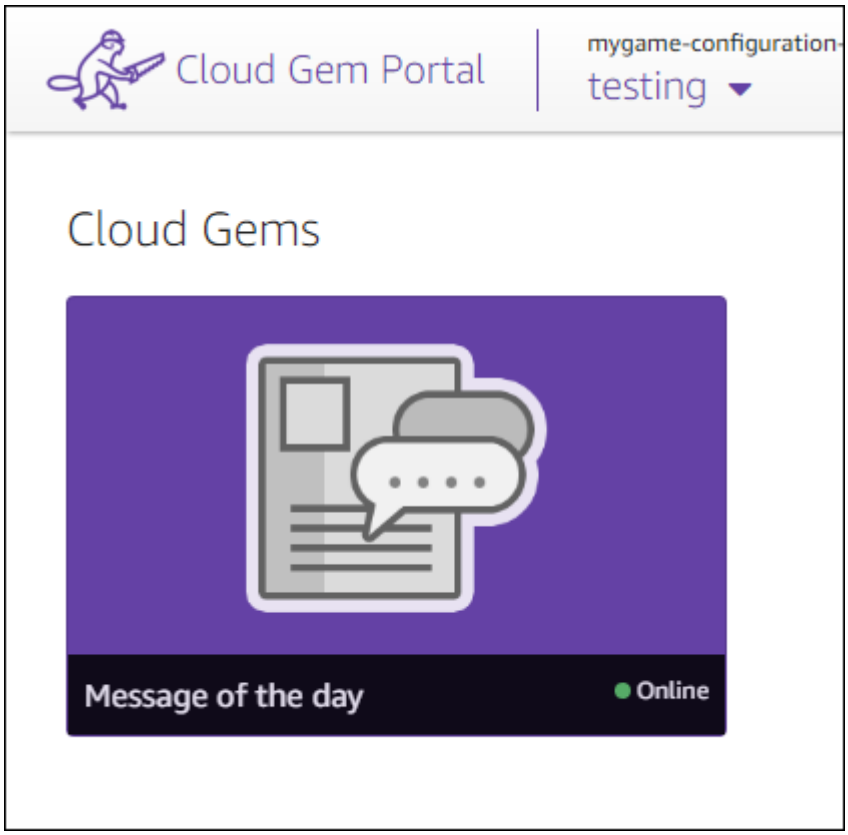
Accessing and Using the Message of the Day Cloud Cloud Gem

To customize the Message of the Day Cloud Gem

1. To view the cloud gems in the Cloud Gem Portal, click **View all Cloud Gems** in the banner, or click **Cloud Gems** on the upper right.

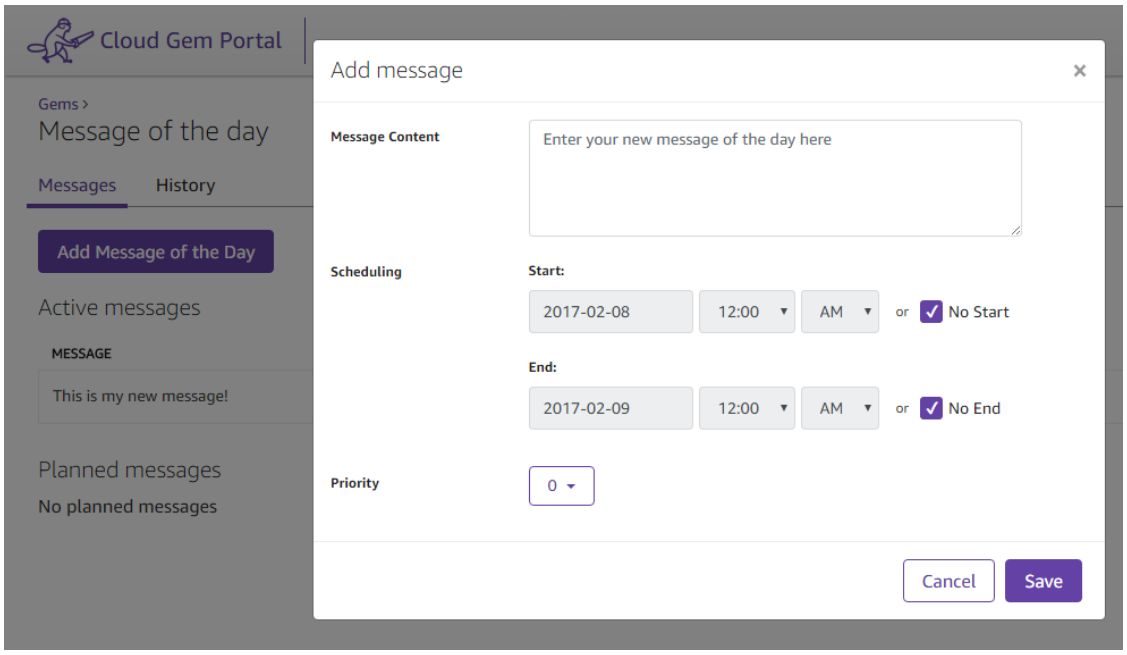


2. Click **Message of the day**.

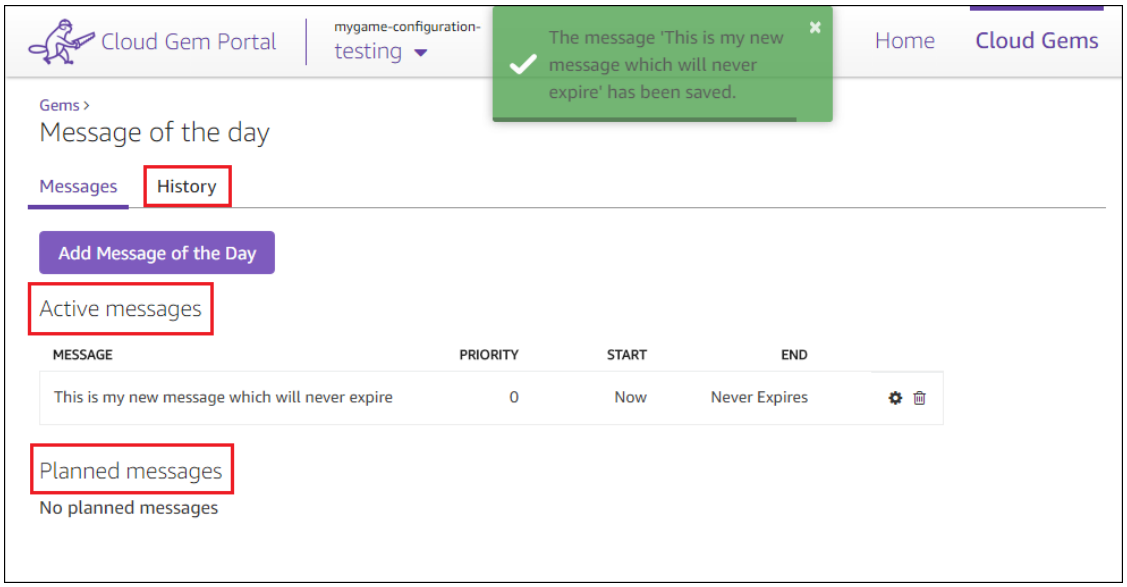


3. Click **Add Message of the Day** to create a new message.
4. In the **Add Message** dialog box, provide the following information:

   - For **Message Content**, type a message that to be displayed to the players of your game.
   - For **Scheduling**, clear the **No Start** and/or **No End** check boxes and specify the **Start** and/or **End** time during which players see the message. Currently, all times are in UTC. The message is shown to players that have a game clock that is between –12 hours or +12 hours of the set UTC time.
   - To make the message permanent, specify both **No Start** and **No End**.
   - If you have more than one message, you can use the **Priority** option to specify the priority of each. By default, 0 is the highest priority. The game client determines how to use the priority value. If you have only one active message, the priority has no effect.



5. Click **Save**. The portal shows your new message in the **Active** category.



The message of the day dashboard has three categories: **Active Messages**, **Planned Messages**, and **Expired Messages**.

- **Active Messages** – The currently active messages that are returned by the game client.
- **Planned Messages** – Messages that are not currently active but will be active at the planned start date and time.
- **Expired Messages** – Outdated messages, which are stored in the **History** tab so that you can edit and reuse them if you want.

You can now use the message in your game.

Other Cloud Gems

Visit the following links for other cloud gems that might interest you.

- Leaderboard Cloud Gem: Leaderboard Cloud Gem Portal (p. 211)
- Dynamic Content Cloud Gem: Using Dynamic Content Manager (p. 220)

Topics

- Testing the Message of the Day Sample Level (p. 206)
- Cloud Gem Message of the Day Implementation Details (p. 206)
- Cloud Gem Message of the Day API Calls (p. 208)

Testing the Message of the Day Sample Level

A sample level for testing is located in the `\dev\CloudGemSamples\Levels\MsgOfTheDaySample` directory. The **MsgOfTheDaySample** level contains a one-button UI for retrieving and displaying successive messages of the day.

To test the Message of the Day sample level

1. Use the Cloud Canvas Resource Manager to upload all the resources that you need to have an updated stack.
2. Open the Cloud Gem Portal and add some messages that are scheduled for today. If you set no scheduling information for a message, the message is always current.
3. To display the messages in a specific order, set the priority field. `0` signifies highest priority.
4. Press **Ctrl+G** to start the level. The test user interface appears.

5. Click **Get Messages**.

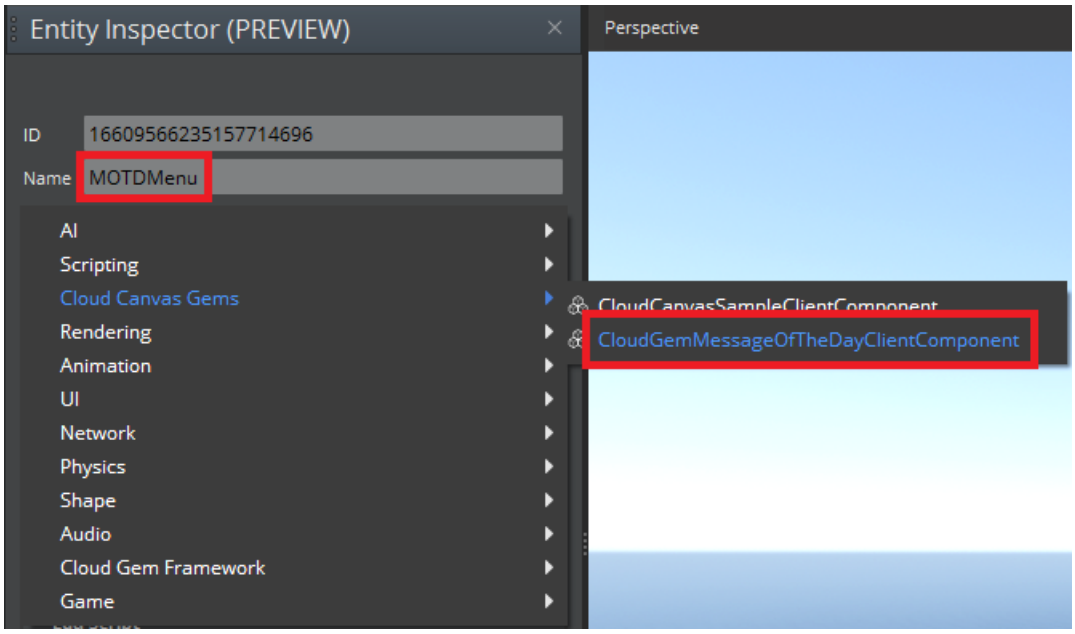
With each click, the messages are displayed in succession.

Cloud Gem Message of the Day Implementation Details

This section describes the entity, component, Lua script, and resources used by the Message of the Day Cloud Gem.

Entity and Component

The Message of the Day sample uses an entity called `MOTDMenu` that contains the `CloudGemMessageOfTheDayClientComponent`.



The `CloudGemMessageOfTheDayClientComponent` component gives the entity access to the game client API operations so that it can receive callback results.

The motdmainmenu.lua Script

The `MOTDmenu` entity also contains a Lua script component that executes the `dev\CloudGemSamples\Scripts\motdmainmenu.lua` script.

The following code from the script shows how `CloudGemMessageOfTheDayRequestBus` is used to call `GetPlayerMessages`.

```
function motdmainmenu:OnAction(entityId, actionName)
    Debug.Log(tostring(entityId) .. ": " .. actionName)

    if actionName == "GetMessages" then
        if CloudGemMessageOfTheDayRequestBus.Event == nil then
            Debug.Log("No Message Request Event found")
            return
        end

        local timeVal = os.date("%b %d %Y %H:%M")
        local lang = "Eng"
        Debug.Log(timeVal);
        CloudGemMessageOfTheDayRequestBus.Event.GetPlayerMessages(self.entityId, timeVal,
 lang, nil)
    end
end
```

The following code from the script shows the `CloudGemMessageOfTheDayNotificationBus` and the result callback functions `OnGetPlayerMessagesRequestSuccess(response)` and `OnGetPlayerMessagesRequestError(errorMsg)` that it connects to.

```
function motdmainmenu:OnActivate()
```

```
...
    self.notificationHandler = CloudGemMessageOfTheDayNotificationBus.Connect(self,
 self.entityId)
end

...

function motdmainmenu:OnGetPlayerMessagesRequestSuccess(response)
    Debug.Log("GetPlayerMessages succeeded")
    self.messageQueue = {}
    self.displayTimer = 0.0
    Debug.Log("Response messages: "..tostring(#response.list));
    --This is a callback from C++ with an object containing a vector called list.
    --Therefore we cannot treat it as a regular Lua table and must rely on the reflected
 methods and operators of the reflected vector class
    --for msgCount = 1, table.getn(response) do
    --    Debug.Log(response[msgCount])
    --    table.insert(self.messageQueue, response[msgCount])
    for msgCount = 1, #response.list do
        Debug.Log(tostring(response.list[msgCount].message))
  table.insert(self.messageQueue, response.list[msgCount])

    end
end

function motdmainmenu:OnGetPlayerMessagesRequestError(errorMsg)
    Debug.Log("GetPlayerMessages Error")
end
```

Cloud Gem Message of the Day Resources

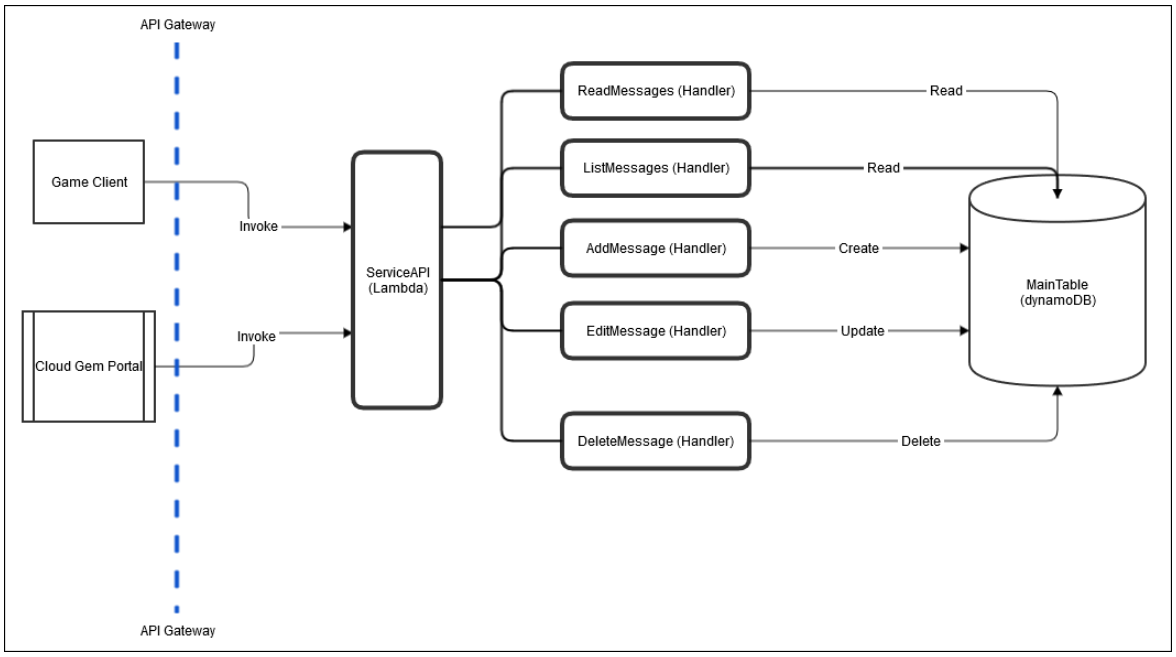
The Cloud Canvas Message of the Day sample has a resource group that contains an Amazon DynamoDB table called Main Table, which is the database for the messages. Table entries are keyed on a server-generated unique message ID. Each entry has the following attributes.

Main Table Attributes

Attribute	Type	Description
`UniqueMsgID`	`string`	The server-generated unique message ID.
`startTime`	`datetime`	Time when the message should start appearing.
`endTime`	`datetime`	Time when the message expires.
`priority`	`integer`	Relative priority of the message. 0 is the highest priority.
`message`	`string`	The message body. The string size is limited to 768 characters.

Cloud Gem Message of the Day API Calls

The game client and Cloud Gem Portal use Amazon API Gateway to securely invoke service API Lambda functions. The game client can read messages while the Cloud Gem Portal can add, list, edit, and delete messages.



The Game Client Service API Calls: /player/messages

The game client read-message functionality is implemented by the `player_messages.py` file located in the `\dev\Gems\CloudGemMessageOfTheDay\AWS\lambda-function-code\api` directory.

get (Read Messages)

The `get` function reads messages for the player.

Parameters

The `get` function has the following parameters.

```
{
    "clientTime" : datetime as string : Opt [default UTC] : The datetime value used to
 retrieve the messages. The default is UTC.
    "clientLanguage" : String : Opt [default "Eng"] : The language used for localization.
}
```

Description

The `get` function returns a list of messages whose `startTime` <= `clientTime` <= `endTime`.

Return Value

The `get` function returns a list of messages of the day.

```
{
    "list" : a list of objects
    {
        "message": string
        "priority": integer,
        "startTime": datetime,
        "endTime": datetime
    }
```

```
}
```

Cloud Gem Portal Admin Service API Calls: /admin/messages

The Cloud Gem Portal message functionality is implemented by code in the `admin_messages.py` file located in the `\dev\Gems\CloudGemMessageOfTheDay\AWS\lambda-function-code\api` directory.

post (Add Message)

The `post` function adds a message of the day.

Parameters

The `post` function has the following parameters.

```
{
    "message": String: Required
    "priority": Integer, : Opt [default 0]
    "startTime": datetime: Opt [default Jan 01 1900 00:00 ]
    "endTime": datetime: Opt [default Dec 31 2100 23:59 ]
}
```

Description

The `post` function adds a message entry to the main table in DynamoDB and returns a `UniqueMsgID` string.

put (Edit Message)

The `put` function edits an existing message.

Parameters

The `put` function has the following parameters.

```
{
    "UniqueMsgID": String: Required
    "message": String: Opt [default leaves current message unchanged]
    "priority": Integer, : Opt [default 0]
    "startTime": datetime: Opt [default Jan 01 1900 00:00 ]
    "endTime": datetime: Opt [default Dec 31 2100 23:59 ]
}
```

Description

The `put` function updates the message in the main table that matches the specified message ID. To update the message, the function overwrites its attributes. You can also use the `put` function to reschedule and reprioritize a message by updating the start time, end time, and priority values.

Return Value

The `put` function returns a string that indicates the result status of the operation.

delete (Delete Message)

The `delete` function deletes the specified message.

Parameters

The `delete` function has the following parameters.

```
{
    "UniqueMsgID": String: Required
}
```

Description

Deletes the message from the main table that matches the specified message ID.

Return Value

The `delete` function returns a string that indicates the result status of the operation.

get (List Messages)

The `get` function retrieves a list of messages of the day.

Parameters

The `get` function has the following parameters.

```
{
    "index": Integer: Opt [default 0]
    "count": Integer: Opt [default 9999999]
    "filter": Opt [default all] Can be one of the following: "active", "expired", "planned"
 or "all"
}
```

Description

The `get` function returns the filtered count of entries in the table, including all fields starting at the specified index.

Return Value

The `get` function returns a list of messages of the day in the following format.

```
{
    "list" : a list of objects
    {
        "UniqueMsgID": String
        "message": String
        "priority": Integer,
        "startTime": datetime,
        "endTime": datetime
    }
}
```

Leaderboard Cloud Gem Portal

You can use the Leaderboard cloud gem portal to create, view, and customize leaderboards by deleting scores, banning players, and removing bans.

Prerequisites

This tutorial assumes the following:

- You are using a Lumberyard project that has the Leaderboard Cloud Gem enabled (in Project Configurator, select **Cloud Gem Leaderboard**).

- You have created a project stack in Cloud Canvas Resource Manager (p. 182).
- You have created a deployment stack in Cloud Canvas Resource Manager.

If you don't meet the prerequisites, follow the steps in Setting Up and Accessing the Cloud Gem Portal (p. 201) to add **Cloud Gem Leaderboard** in Project Configurator.

Accessing and Using the Leaderboard Cloud Gem Portal

You use Lumberyard Editor to work with the Leaderboard Cloud Gem Portal.

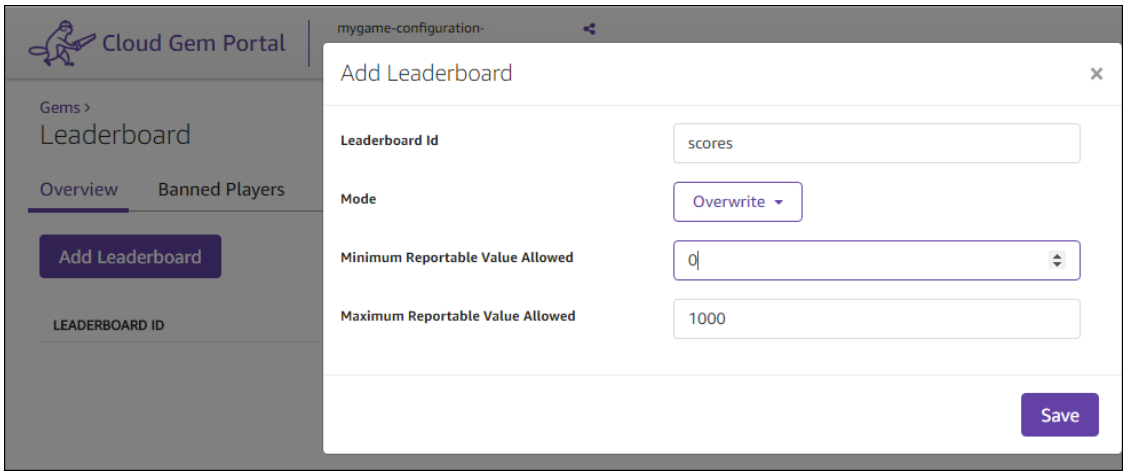
To access and use the Leaderboard Cloud Gem Portal

1. **Opening the Cloud Gem Portal**

   In Lumberyard Editor, click **AWS**, **Open Cloud Gem Portal** and then choose the **Leaderboard** cloud gem.

2. **Creating a Leaderboard**

   You must create a leaderboard for your game before you can start posting scores to it. In the Leaderboard Cloud Gem, click **Add Leaderboard** to create a leaderboard so that your game can use it. If your game tries to send data to a leaderboard that doesn't exist, the data is not be sent or saved anywhere.

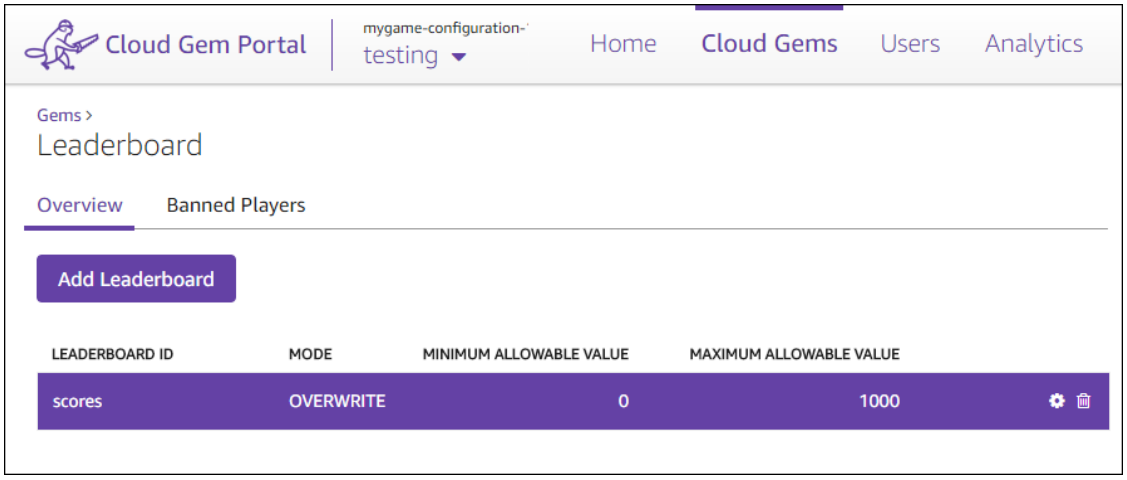


   A leaderboard has the following four options:

   - **Leaderboard ID –** This is the identifier for your leaderboard. The game client uses this ID to send data to the leaderboard. Currently, this ID does not support spaces. If you're creating a leaderboard for a particular region or language, it's good to put that information in the ID. For example, you could designate scores for North America with the ID "scores-NA" instead of just "scores".
   - **Mode –** At launch, leaderboards have two modes: **Overwrite** and **Increment**. Overwrite mode overwrites a previous score for a user. Increment mode adds the value passed in from the game client to the previous value to keep a running total of a particular statistic. For example, you might use increment mode to track career statistics in a game.
   - **Minimum Value Allowed –** The minimum value for the leaderboard. If the game client sends scores below the minimum value allowed, the scores are not recorded.
   - **Maximum Value Allowed –** The maximum value for the leaderboard. If the game client sends scores above the maximum value allowed, the scores are not recorded.

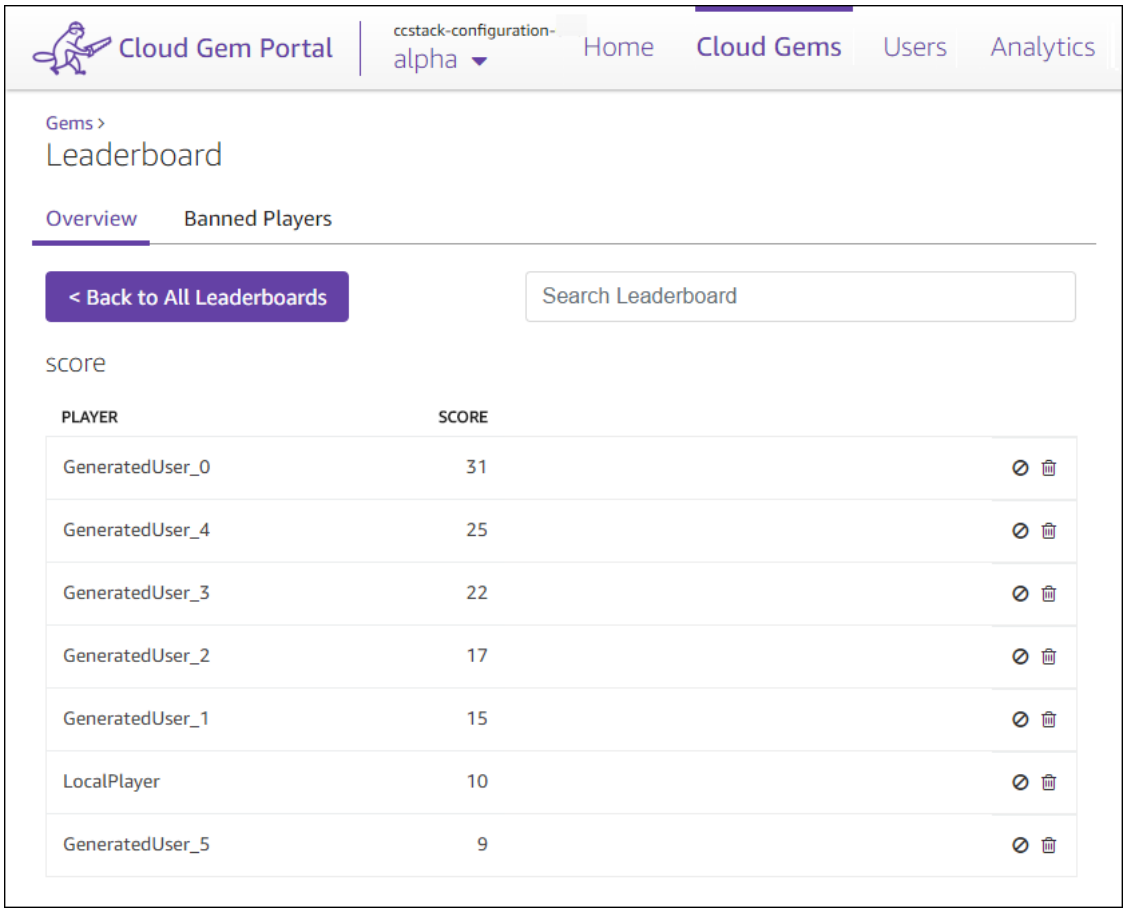
3. **Editing a Leaderboard**

   To edit all options (except for **Leaderboard ID**), click the settings icon on the right side of the leaderboard. If you want to change the value for **Leaderboard ID**, you must delete and recreate the leaderboard. When you delete a leaderboard, its data is also deleted.



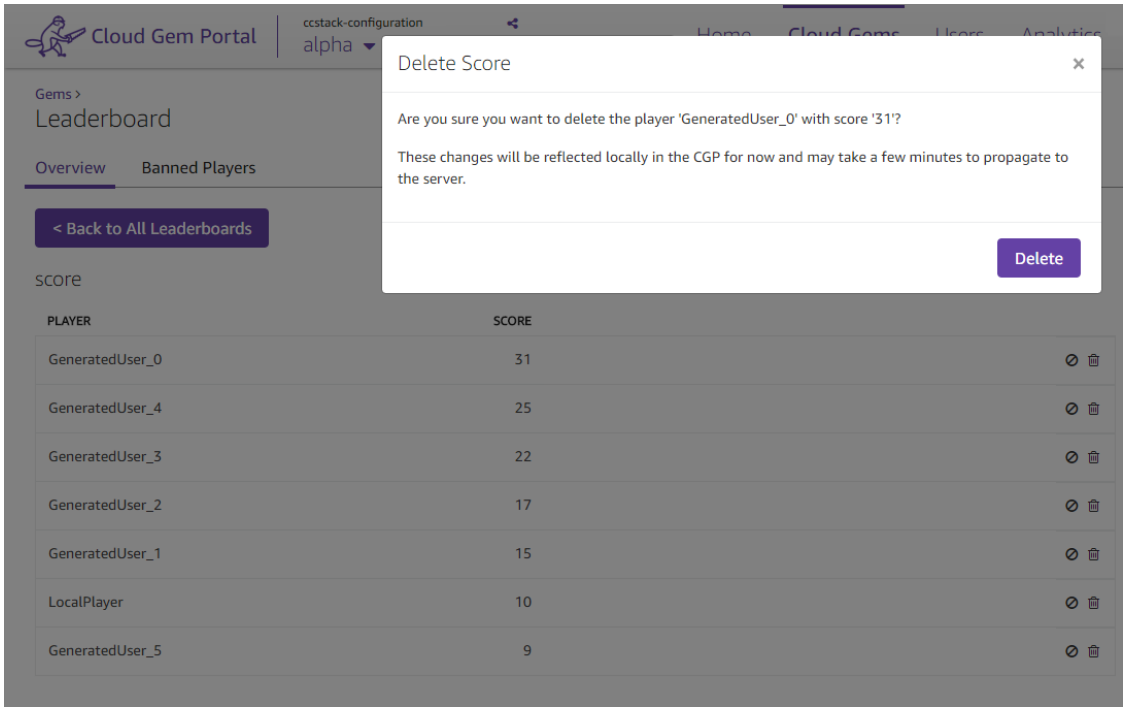
4. **Viewing Leaderboards**

   To view a leaderboard, click its name. This shows you a list of players and their scores in the leaderboard. You can also delete a player's scores or ban a player from all leaderboards.



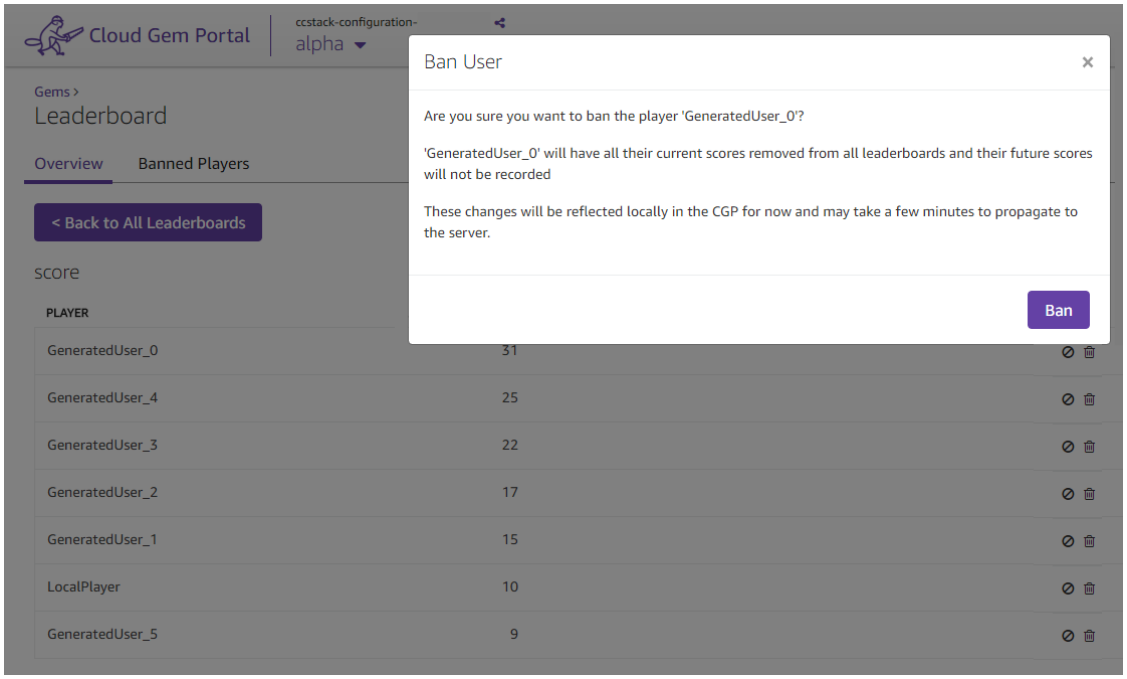
5. **Deleting Scores**

   To delete a score, click the delete icon on the right of the player's score. Then click **Delete** in the confirmation window.



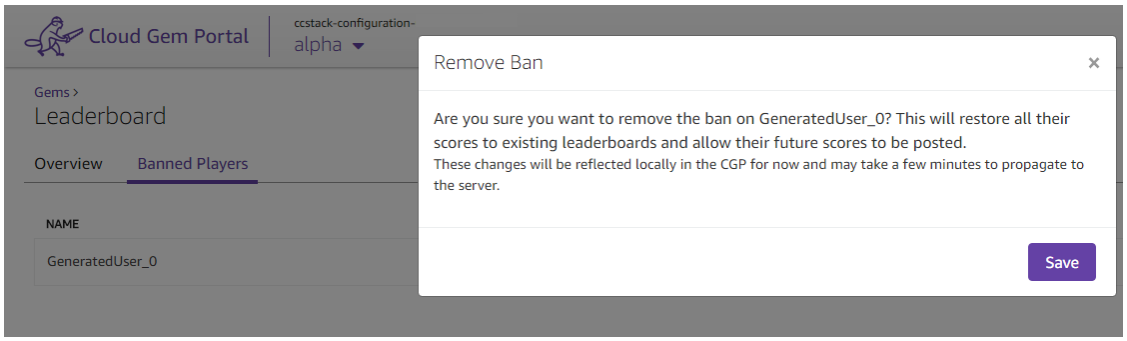
6. **Banning a Player**

To ban a player, click the ban icon on the right side of the player table next to the delete icon. When you ban a player, the player's scores are removed from all leaderboards, and the player is prevented from posting on other leaderboards. This is a reversible action; you can remove a ban if done by mistake.



7. **Removing a ban**

   To remove a ban, click the **Banned Players** tab. The tab displays a list of banned players in all your leaderboards. Click the ban icon again, and then click **Save** in the confirmation window.



Topics

- Testing the Leaderboard Sample Level (p. 216)
- Leaderboard Sample Level Implementation Details (p. 217)
- Leaderboard Cloud Gem Resources (p. 218)
- Leaderboard Cloud Gem Service API (p. 219)

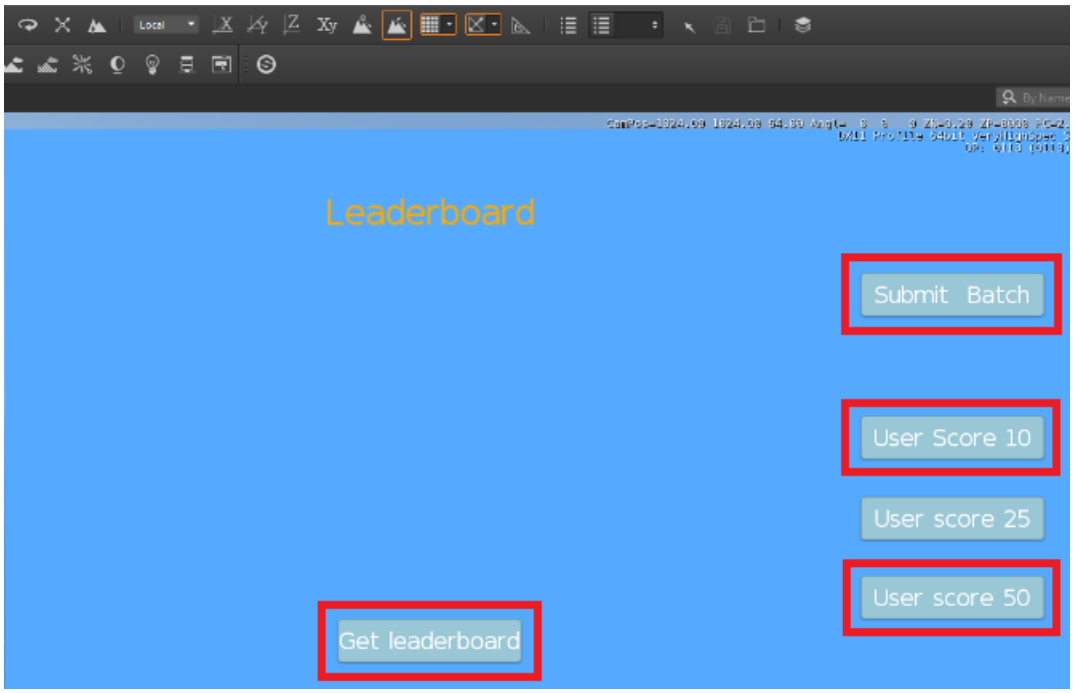
Testing the Leaderboard Sample Level

The `CloudGemLeaderboard` Cloud Gem is included in the LeaderboardSample level, which is located in the `dev\CloudGemSamples\Levels\LeaderboardSample` directory. The LeaderboardSample level contains a UI canvas with multiple buttons that you can use to test the Leaderboard system.

> **Note**
> By default, the Lambda function that processes score records is on a 10-minute timer. This timer avoids possible consistency issues and costs that might occur because of reduced throughput capacity. The configuration for the timer is located in the `resource-template.json` file for the gem. When you test the gem, you might want to reduce the timer value in the template file. Alternatively,you can use the AWS Lambda console to disable the timer trigger, and then trigger it manually by using the **Test** button.

To test the Leaderboard using the sample level

1. Using the **Cloud Canvas Resource Manager**, upload all resources to ensure that you have an updated stack.
2. Open the Cloud Gem Portal (CGP) and add a statistic to the leaderboard. This creates a description of a statistic to track (if no definitions exist, you cannot submit a score). The default properties in the test level expect a statistic called `score`.
3. Press **Ctrl+G** to start the level. The test user interface appears.

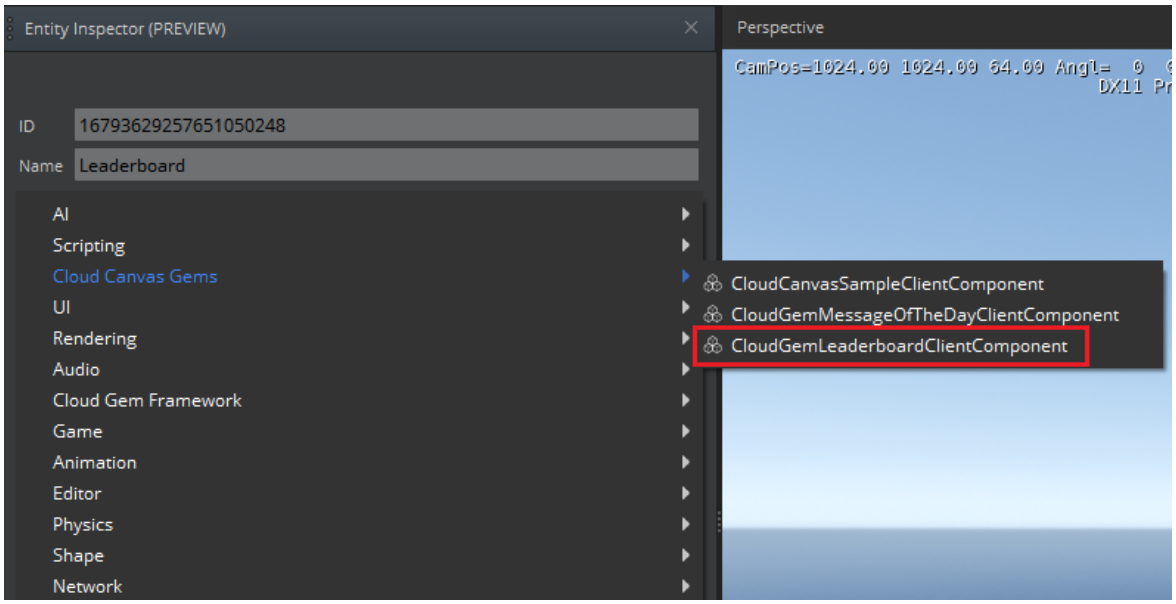


4. Click **Submit Batch** to populate the leaderboard with data. You can click the button multiple times to simulate a larger population. Messages in the editor logs show whether the submissions were successful.
5. Click **Get Leaderboard** to see the top five scores in the sample.
6. Click **User Score 10** to submit a score for the **LocalPlayer** user.
7. To verify the submission, click the leaderboard in the Cloud Gem Portal. A score with the value of **10** should appear under the name **LocalPlayer**.
8. Click **User Score 50** to submit another score for **LocalPlayer**. Because the new score is higher, it should overwrite the earlier score. To verify the submission, click the leaderboard in the Cloud Gem Portal again.

Note
There is a caching mechanism on the backend, so the scores might not be updated immediately.

Leaderboard Sample Level Implementation Details

An entity named `Leaderboard` contains the `CloudGemLeaderboardClientComponent`.



The `CloudGemLeaderboardClientComponent` enables the gem's API to be called from other components on that entity and receives notifications for those calls.

The `Leaderboard` entity also contains a Lua script component that uses the `leaderboardmainmenu.lua` script. The script creates the testing user interface and calls the following APIs through the attached client component:

- `PostScore` (to submit scores)
- `PostLeaderboard` (to request a leaderboard for a stat)

The script also implements the notification methods for these API calls:

- `OnPostScoreRequestSuccess`
- `OnPostScoreRequestError`
- `OnPostLeaderboardRequestSuccess`
- `OnPostLeaderboardRequestError`

Leaderboard Cloud Gem Resources

The following table lists the resources used by the Leaderboard Cloud Gem.

Resource	Description
`BannedPlayerTable`	A list of players that are no longer allowed to submit scores.
`LeaderboardInfo`	Information for each statistic about the state of the leaderboard, including population and maximum sample size.
`LeaderboardTable`	A table that contains a set of scores for each statistic. The set is a sample gathered using a reservoir sampling method. The sets are used to estimate where an arbitrary user's score sits within the entire population. This allows the cloud gem to estimate global rankings.

Resource	Description
`MainTable`	The main database for scores. Entries are keyed on `userID` with a score attribute for each metric. This table allows you to gather the most up-to-date information on a specified set of players.
`RecordsBucket`	An Amazon S3 bucket that holds all the records written by `ScoreStreamReader`.
`RecordSetProcessor`	A Lambda function that reads through all pending records and decides whether the samples on `LeaderboardTable` need to be updated. This Lambda function is triggered by the timer `RecordSetProcessorSchedulerRule` that has a default of 10 minutes.
`RecordSetStatus`	A table that keeps track of the status of all record sets. Sets can be `OPEN` (accepting new records), `PENDING` (not accepting records, to be processed soon), or `CLOSED` (processed, waiting for cleanup). The information in `RecordSetStatus` determines which record set `RecordSetProcessor` processes and where `ScoreStreamReader` puts its next record.
`ScoreStreamReader`	A Lambda function triggered by DynamoDB updates to `MainTable`. Writes records to the Amazon S3 `RecordsBucket` for later processing.
`StatsSettings`	A table that specifies how to process specific statistics from submissions. Holds minimum and maximum values for each statistic and specifies whether it is to be overwritten or incremented on submission.
`UserIdentityMap`	Maps user IDs to Amazon Cognito IDs to ensure that scores can't be submitted on another player's behalf. After an ID is used for the first time, it is maintained by the Amazon Cognito ID.

Leaderboard Cloud Gem Service API

The following table lists the service API calls for the Leaderboard Cloud Gem.

API	Description
`/score POST`	Submits a new score. Expects a payload including `user`, `value`, and `stat` fields. Returns the updated score for that statistic. The updated score returned is not necessarily the score that was submitted. This operation is player callable. **Note** The leaderboard can store integer scores up to the maximum integer value or decimal scores up to `1e+20`.
`/score/{stat}/{user}` `GET`	Gets a specific score for the provided user. This operation is player callable.
`/score/{stat}/{user}` `DELETE`	Deletes a specific score for the provided user.
`/scores/{user} GET`	Gets all scores (multiple statistics) for the provided user. This operation is player callable.
`/stats GET`	Lists all valid statistics that are registered with the leaderboard.
`/stats POST`	Register a new statistic. Expects a payload with `name`, and `mode` fields. The `min` and `max` fields are optional. Returns the list of all valid statistics.

API	Description
`/stats/{stat_name}` `DELETE`	Deletes the specified statistic definition. Returns the list of all valid stats.
`/player/ban_list GET`	Lists all the banned players.
`/player/ban/{user}` `POST`	Bans the specified player.
`/player/ban/{user}` `DELETE`	Lifts the ban on the specified player.
`/leaderboard/{stat}` `POST`	Takes an optional payload that expects a `users` field. The `users` fields is a list of the user IDs to include in the leaderboard sample. Returns a leaderboard sample for the provided statistic with estimated ranks. This operation is player callable.

Using Dynamic Content Manager

You can use the Dynamic Content Cloud Gem and its Dynamic Content Manger to manage dynamic content updates for your game through AWS. You can use Dynamic Content Manger to create manifests and packages, add files to the packages, and upload the manifest and packages to the AWS Cloud. This ensures that your clients have the latest content for your game.

Topics

- Managing Dynamic Content Packages (p. 228)
- Using lmbr_aws for Dynamic Content (p. 233)
- Testing the Dynamic Content System (p. 235)
- Dynamic Content Engineering Details (p. 239)

Prerequisites

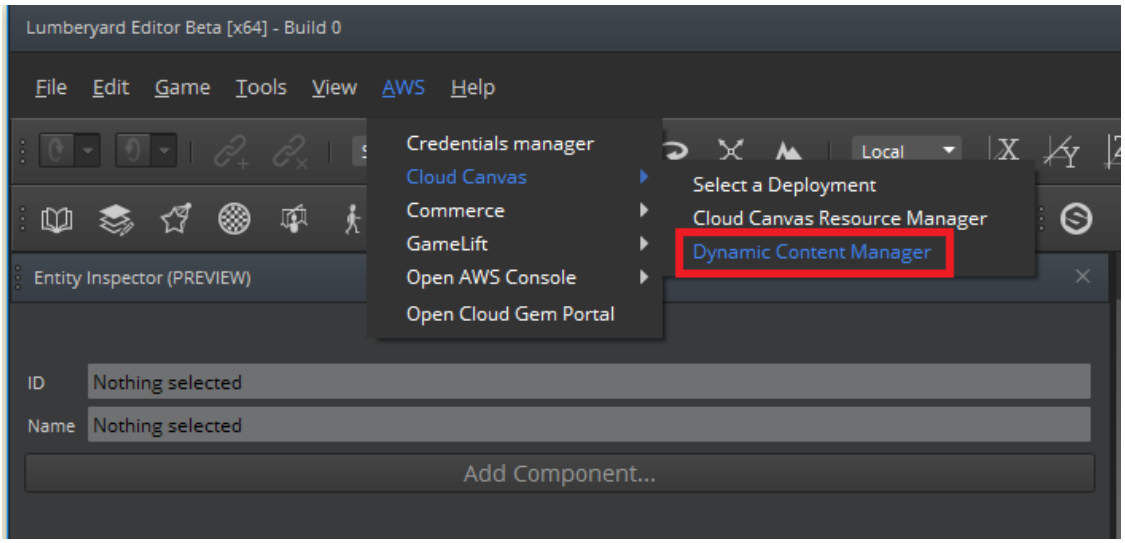
This tutorial assumes the following:

- You are using a Lumberyard project that has the Dynamic Cloud Gem enabled (in Project Configurator, select **Cloud Gem Dynamic Content**).
- You have created a project stack and deployment stack in Cloud Canvas Resource Manager (p. 182) with the `CloudGemDynamicContent` resource group.

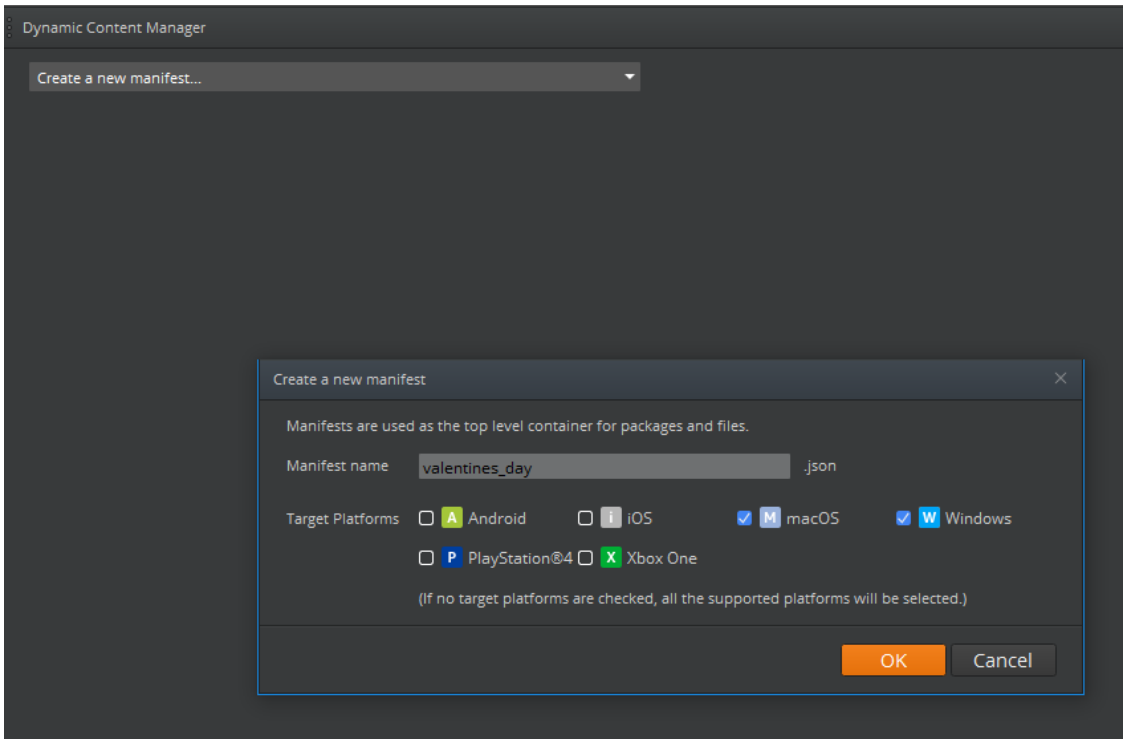
> **Note**
> If you do not have a deployment with the `CloudGemDynamicContent` resource group, the **Dynamic Content Manager** interface is disabled.

To use the Dynamic Content Manager to upload dynamic content

1. In Lumberyard Editor, choose **AWS**, **Cloud Canvas**, **Dynamic Content Manager**.



2. In **Dynamic Content Manager**, click **Create a new manifest**.

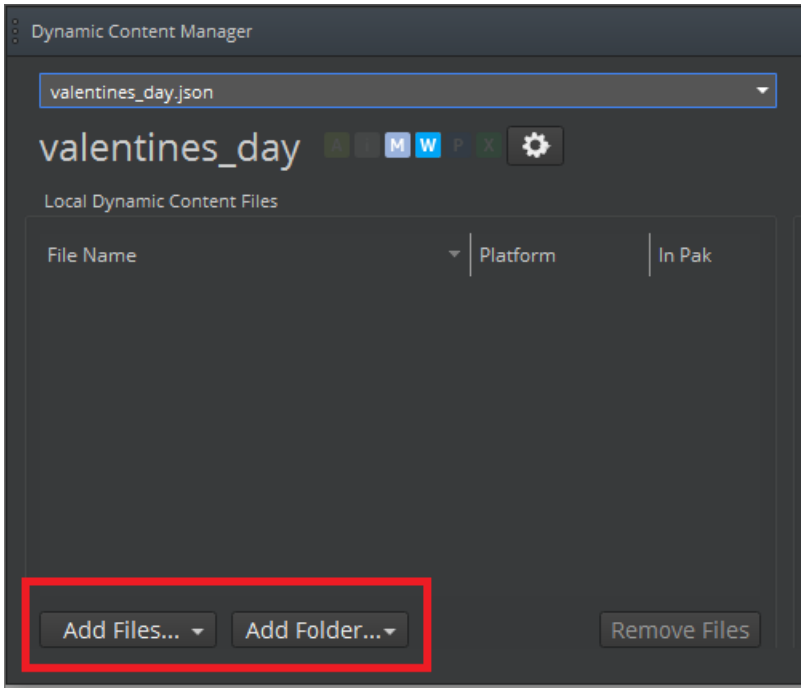


A manifest is a file that keeps track of the dynamic content files that you deliver to customers. A manifest records the following information:

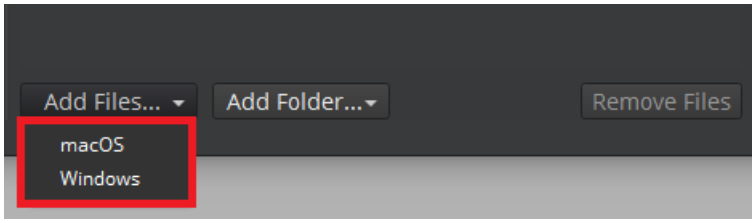
- The content files that have changed locally
- The files that are included in any file packages
- Differences between the local packages and the packages in the cloud that are staged for delivery to customers
- The target platforms for this manifest

You can create as many manifests as you require. For example, you can create specialized manifests for different types of assets.

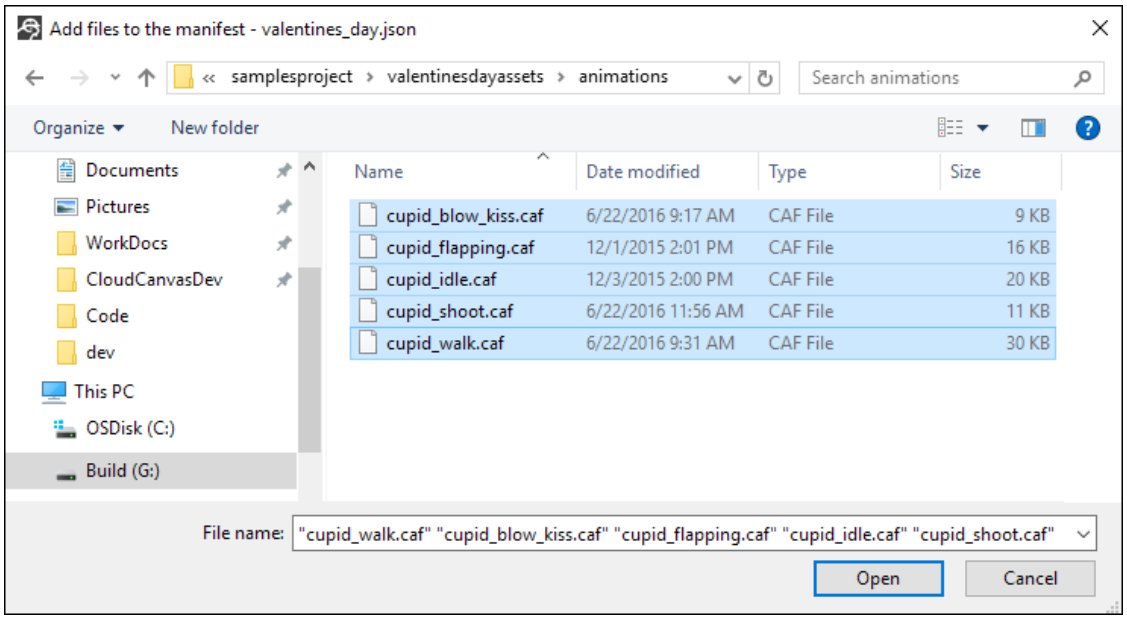
3. Choose a name for the manifest. The name should reflect the purpose of the manifest.
4. Select the target platforms that you prefer to work on for this manifest.
5. When you create a manifest, the manifest is empty and has no files or packages.



Click **Add Files** or **Add Folder** and select one of the target platforms.

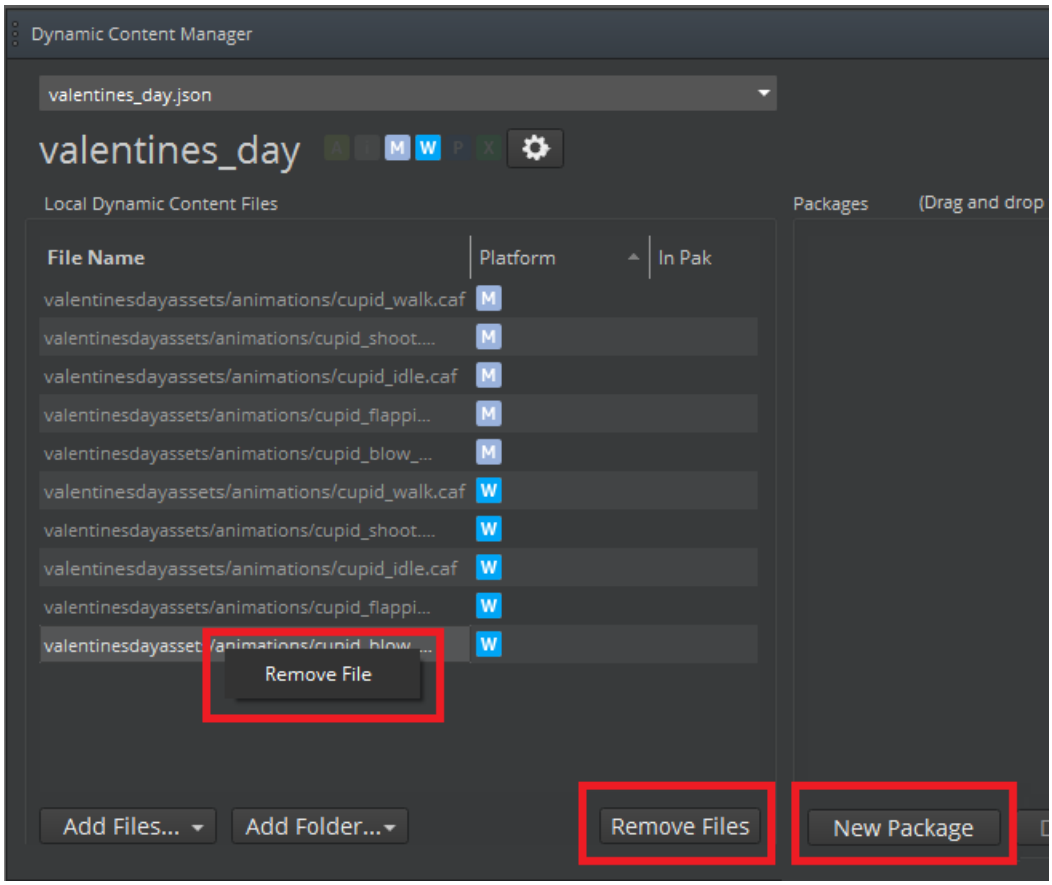


6. In the file browser, choose the files that you want to add to the manifest.



Because the original assets might not be in a form that your game can consume, the file browser opens to location of your game projects asset cache. For example, your game platform might require textures to be in `.dds` format, but the original asset for a texture might be a `.png` file. The asset processor converts the files to the appropriate format and stores them in the asset cache, so the asset cache is your safest choice. However, if you have processed assets elsewhere in your file system, you are free to include them.

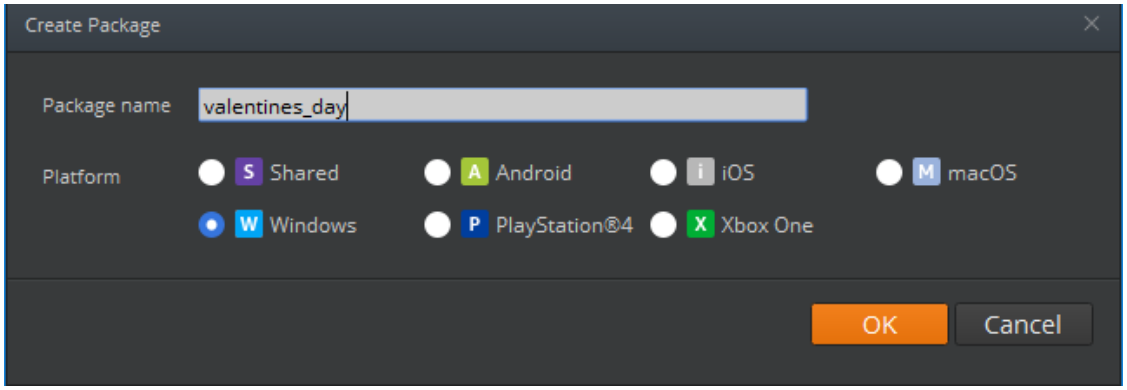
7. The left pane of **Dynamic Content Manager** shows the files that you added to the manifest. You can add files for different target platforms and sort the files by their name, platform, or status. After you have some files in the file manager, you can select one or more files and remove them. You can remove the files by right-clicking them or by clicking **Remove Files**.



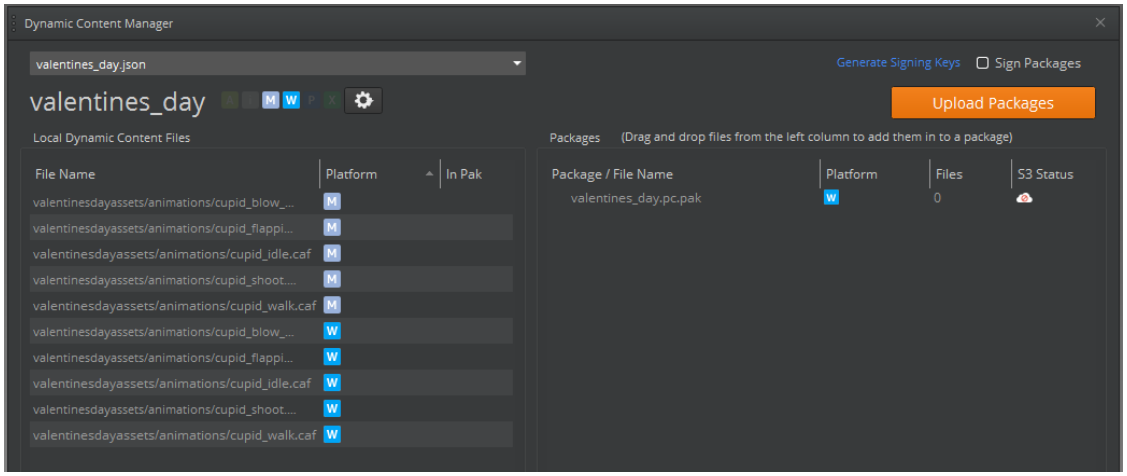
Now you are ready to add a package to the manifest.

Click **New Package**.

8. Type a name for the package and choose the platform type for it.

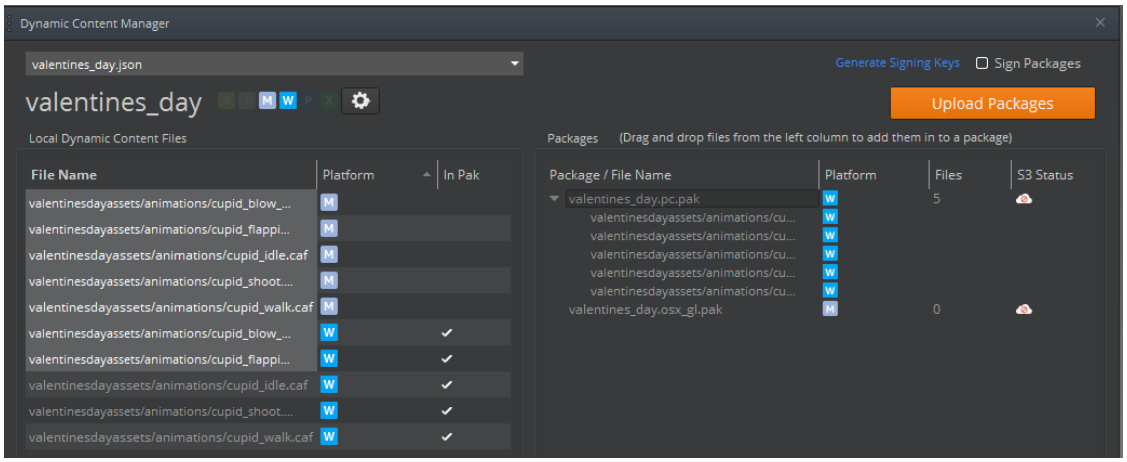


9. The new package appears in the right pane of **Dynamic Content Manager**. You can sort the packages by column.

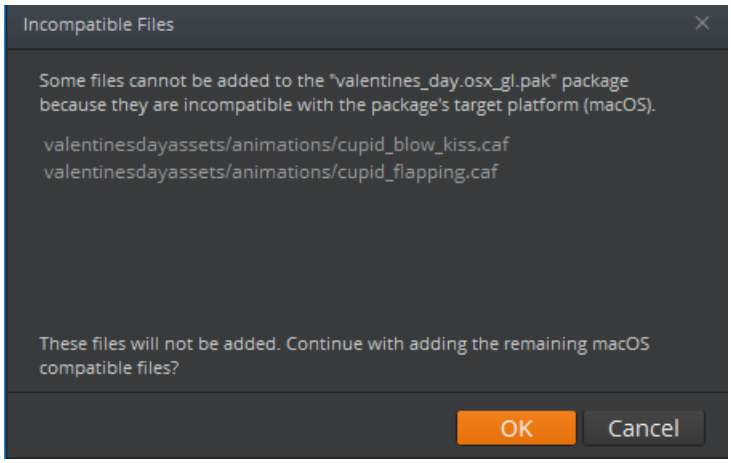


Each package is stored in a `.pak` file, which is the archived file format that Lumberyard uses for its released game assets. You can have one or more packages per manifest, and you can assign the same file to multiple packages. This is useful when you have common files and platform-specific files that you need to deliver in one package. It is also useful if you want to create bundles of items that might share assets.

10. To add files to the package that you created, drag the files from the manifest on the left to the package on the right. You can drag multiple files into the same `.pak` file.



You can drag only platform-specific files to a package with that platform type. If you drag multiple files to a package and the platform type of these files does not match that of the package, Lumberyard warns you about the files that cannot be added.



The icons that appear in **Packages** pane are described as follows.

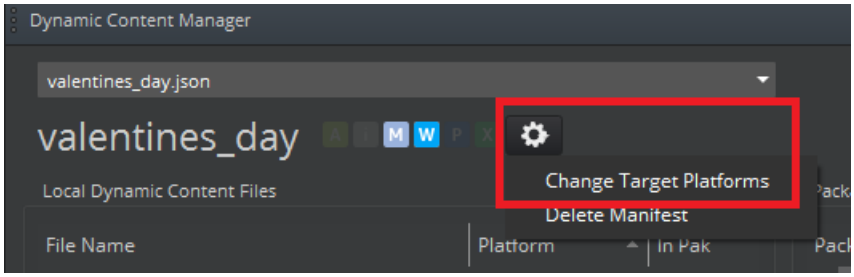
Icon	Description
	A file in a package has been updated in the asset cache. The contents of the package have been updated, so you might want to upload the package again.
	The package is not in the cloud. This can occur when the package has not yet been uploaded.
	A package has changed with respect to its copy in the cloud. The contents of the package have been updated, so you might want to upload the package again.
	The local package matches the package in the cloud. No action is necessary.

Notes

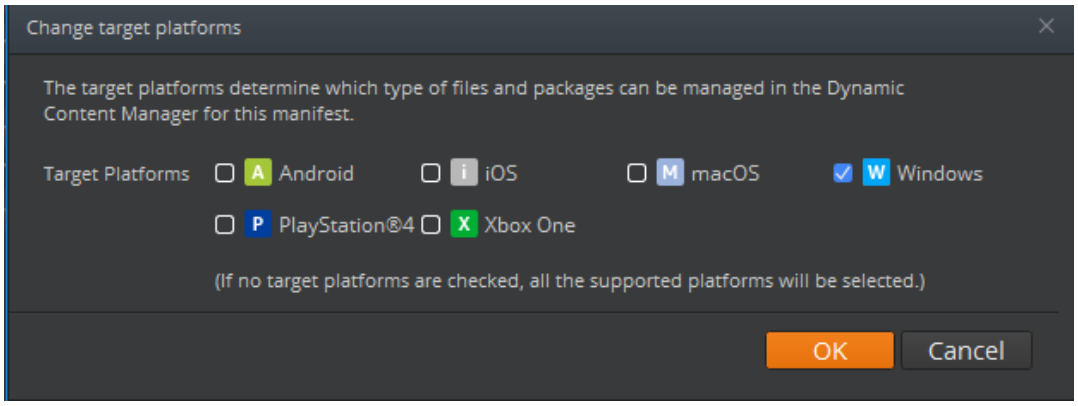
- You can remove individual files from a package, but to help prevent inadvertent removal, this action is limited to one file at a time.
- You can also use **Dynamic Content Manager** to delete packages, but deleting a package does not delete the staged packages in the cloud. To delete staged packages in the cloud, you must use the web-based Cloud Gem Portal that manages the service side of a cloud gem.

11. You can change the target platforms for the current manifest at any time.

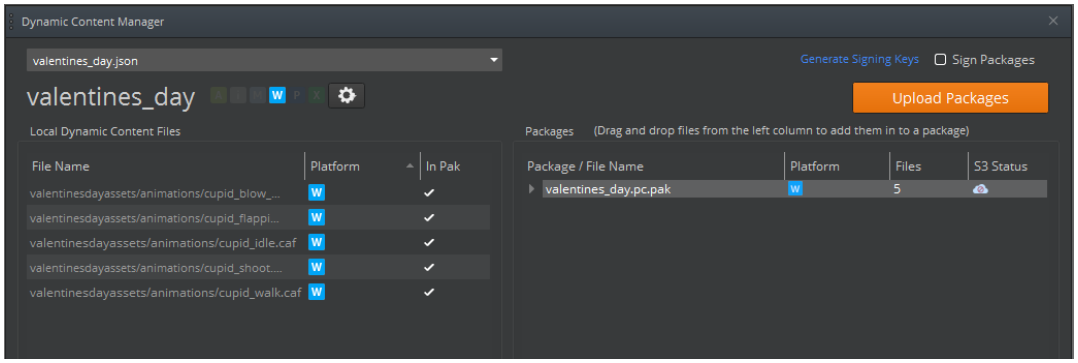
    a. Click the settings button and choose **Change Target Platforms**.



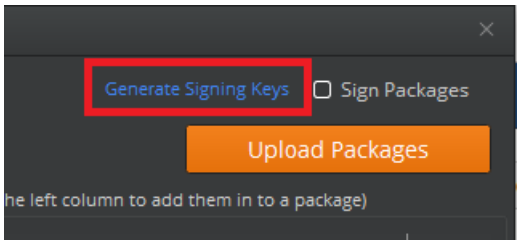
    b. Choose the platforms that you want. If no target platforms are selected, all supported platforms are added.



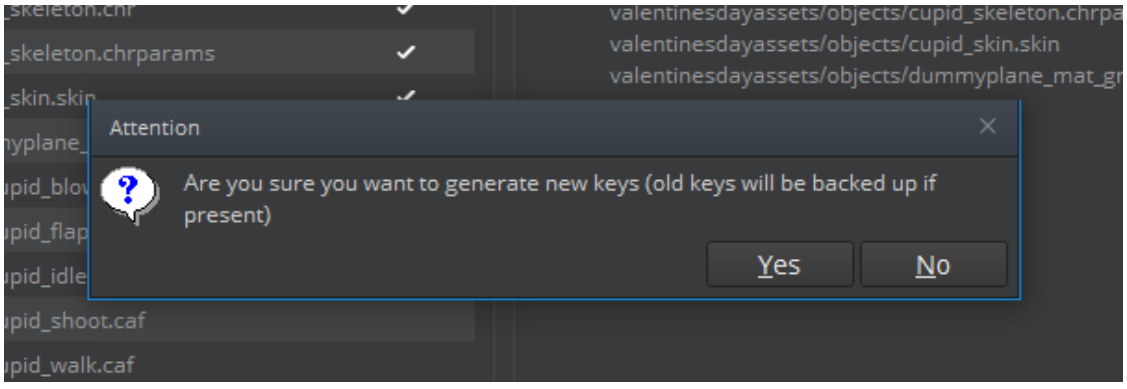
c. After you update the target platforms, **Dynamic Content Manager** lists only the files and packages for the platforms that you selected. You can add files and folders only for the selected target platform.



12. To optionally generate key pairs to sign your package, click **Generate Signing Keys**. This security option is useful if you have concerns that third parties might tamper with your packages.

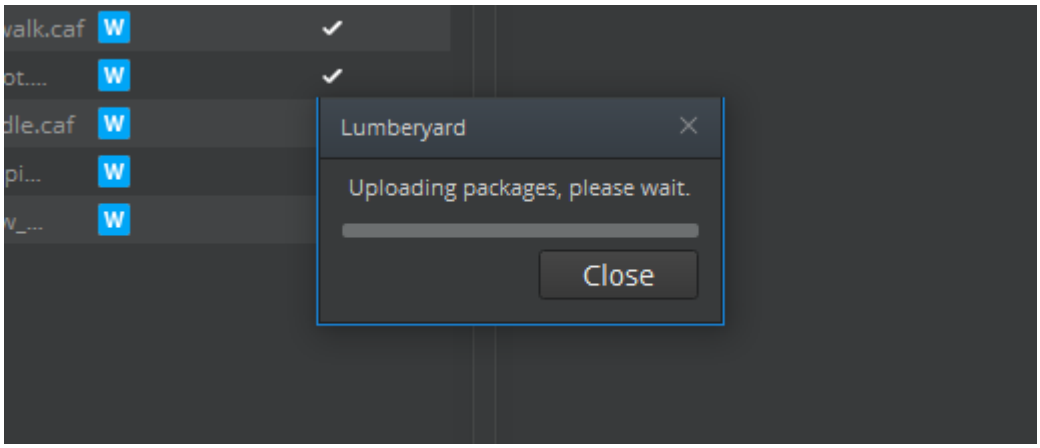


13. Click **Yes** at the confirmation prompt.



Generating new keys is nondestructive. A time-stamped backup is always created for each key pair.

14. When you are ready, click **Upload Packages**. This action both creates the package files locally and uploads them to your private staging bucket in the AWS Cloud.



The packages appear in the **Dynamic Content** section of the Cloud Gem Portal. By default, Lumberyard marks them as **Private**. When you are ready, you can move the packages to the **Public** stage to make them available for download by game clients. For more information on this step, see Managing Dynamic Content Packages (p. 228).

Managing Dynamic Content Packages

The Dynamic Content Cloud Gem provides an easy way for you to update your game content for your customers. After a release, you can update content without an app submission or traditional patch process. The Dynamic Content Cloud Gem provides a framework for you to group outdated assets into manifests. You can assign groups of assets together into `.pak` files (`.zip` files), and distribute these files to customers according to your own rules or schedules.

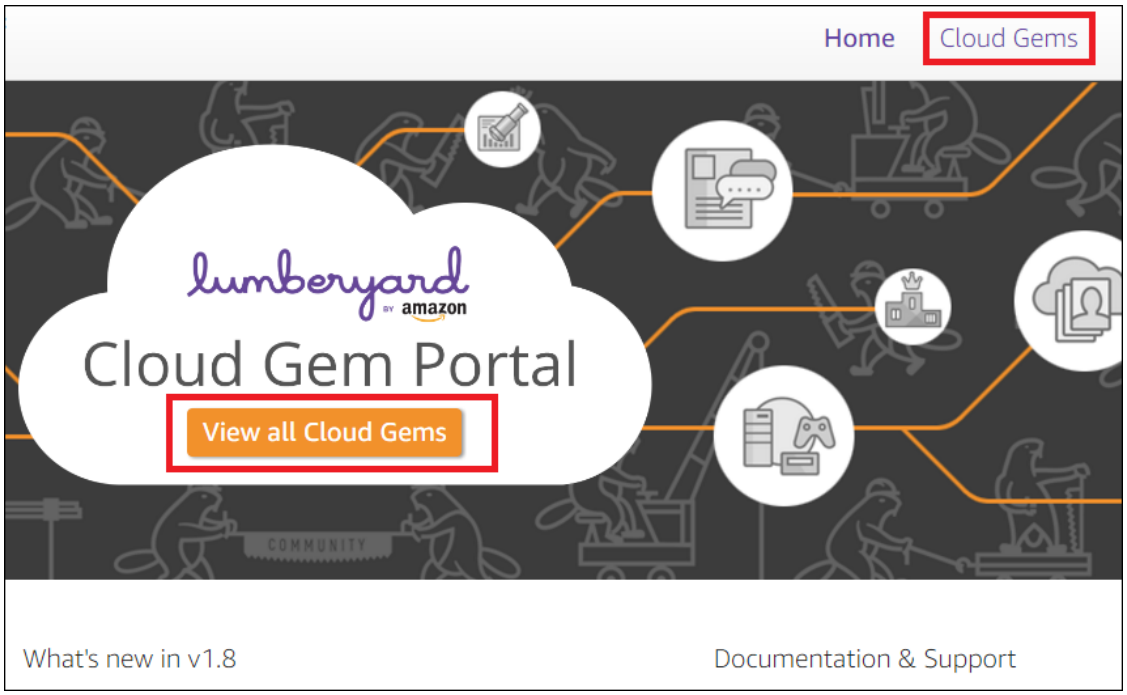
Prerequisites

To use the Dynamic Content Cloud Gem, you must meet these requirements:

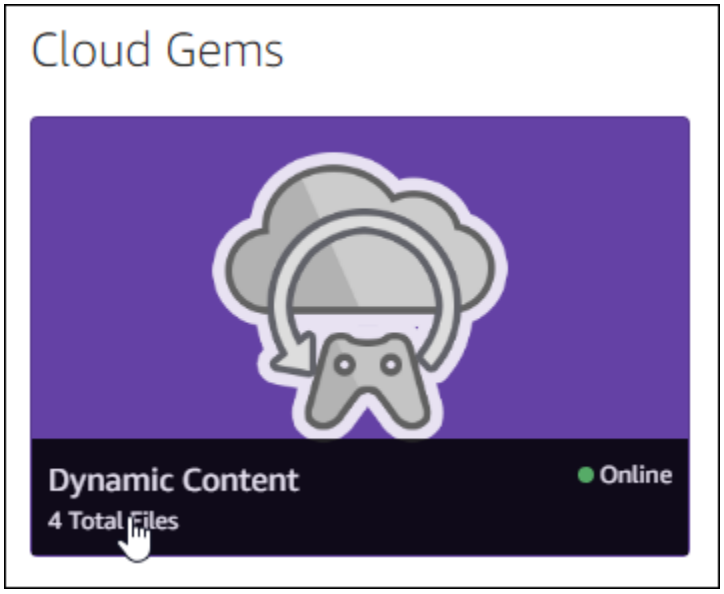
- You must be using a Lumberyard Editor project that has the Dynamic Content Cloud Gem enabled (in Project Configurator, select **Cloud Gem Dynamic Content**).
- You have a project stack created in Cloud Canvas Resource Manager (p. 182).
- You have created a deployment stack in Cloud Canvas Resource Manager.
- You have created dynamic content packages that contain the updates. For more information, see Using Dynamic Content Manager (p. 220).
- You have the Cloud Gem Portal open. In Lumberyard Editor, click **AWS**, **Open Cloud Gem Portal**.

To view packages in the Dynamic Content Gem Portal

1. On the welcome page of the Cloud Gem Portal, click **Cloud Gems** or **View all Cloud Gems**.



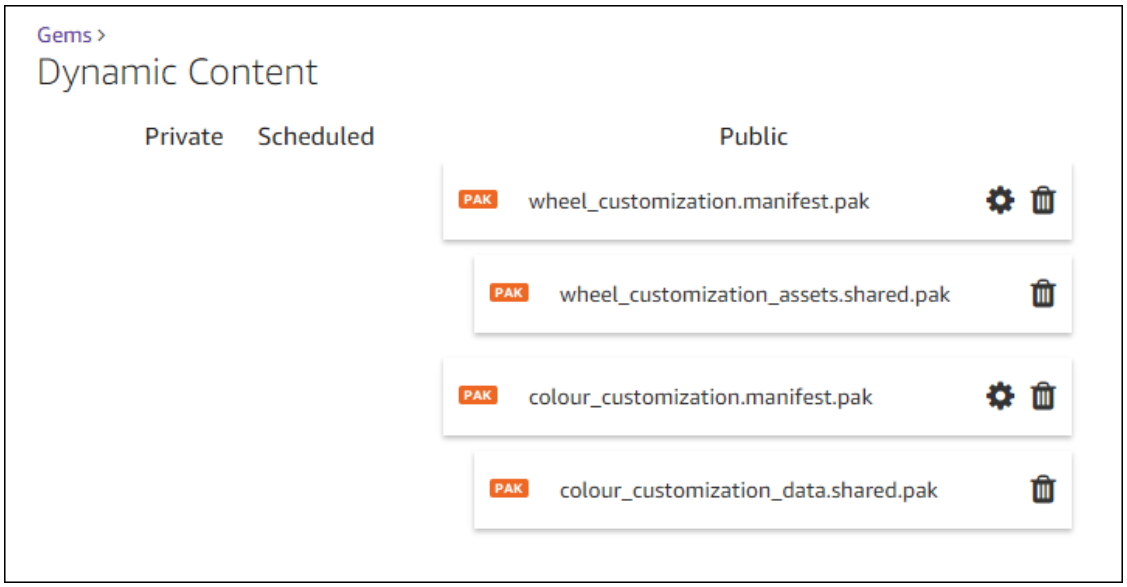
2. In the list of enabled cloud gems, pause on the thumbnail for the Dynamic Content Gem.



   The thumbnail shows the number of files in the gem and the status of the cloud API (**Online** or **Offline**). The online status comes from an API Gateway call to the Dynamic Content Cloud Gem.

3. Click the thumbnail. The dynamic content might take a few seconds to appear as AWS resources load.

   The loaded page shows the packages (`.pak` files) that you added when you used **Dynamic Content Manager**.



Dynamic Content Package Stages

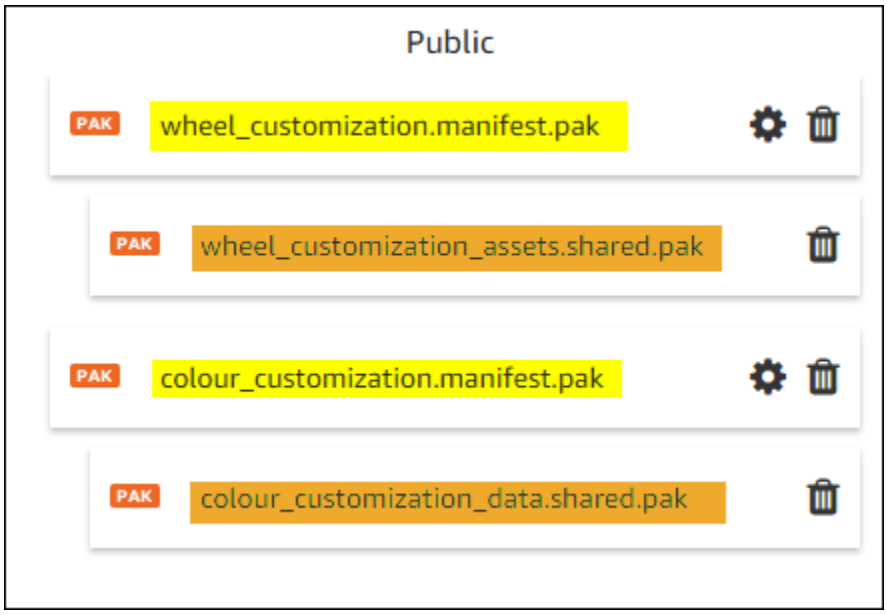
You can use separate deployment stacks to handle different release workflows. For this purpose, the Dynamic Content Cloud Gem currently supports the following publishing stages.

- **Private** – The packages are never downloaded by the game client. When you upload new or altered content to the cloud from **Dynamic Content Manager**, your packages and manifest are always put in the **Private** stage. This empowers you to choose when and how your changes are released to customers.
- **Scheduled** – The packages are conditionally downloaded by the game client based on a specific date and time.
- **Public** – The packages are available to everyone that uses the deployment.

You can use these stages to add an additional layer of protection against inadvertent deployments of dynamic content to your players.

Package Nesting

Each stage includes entries for your manifests and packages. Packages are nested according to the hierarchy that you determine when you create them. In the following example, the two root-level manifest packages highlighted in yellow have one child asset package each, which are highlighted in gold.

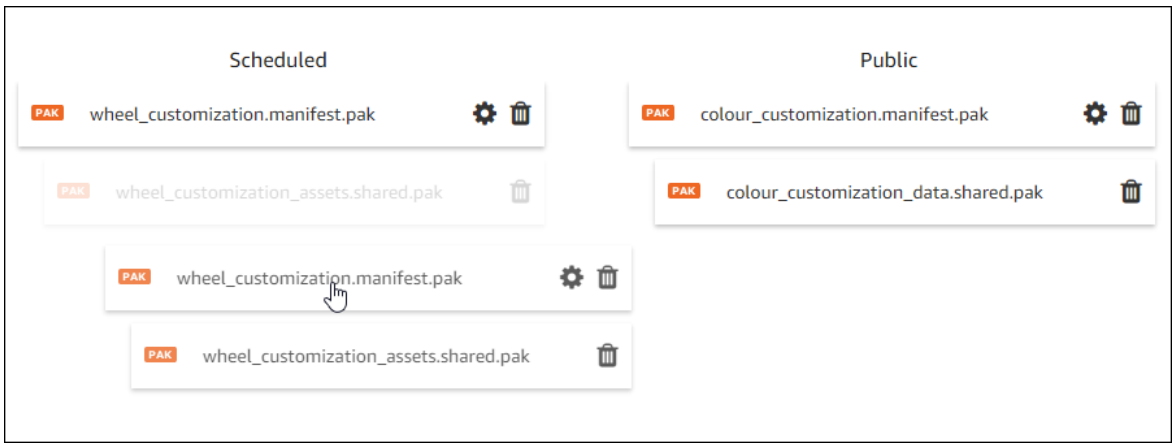


The icons next to the package names show the actions available for the packages.

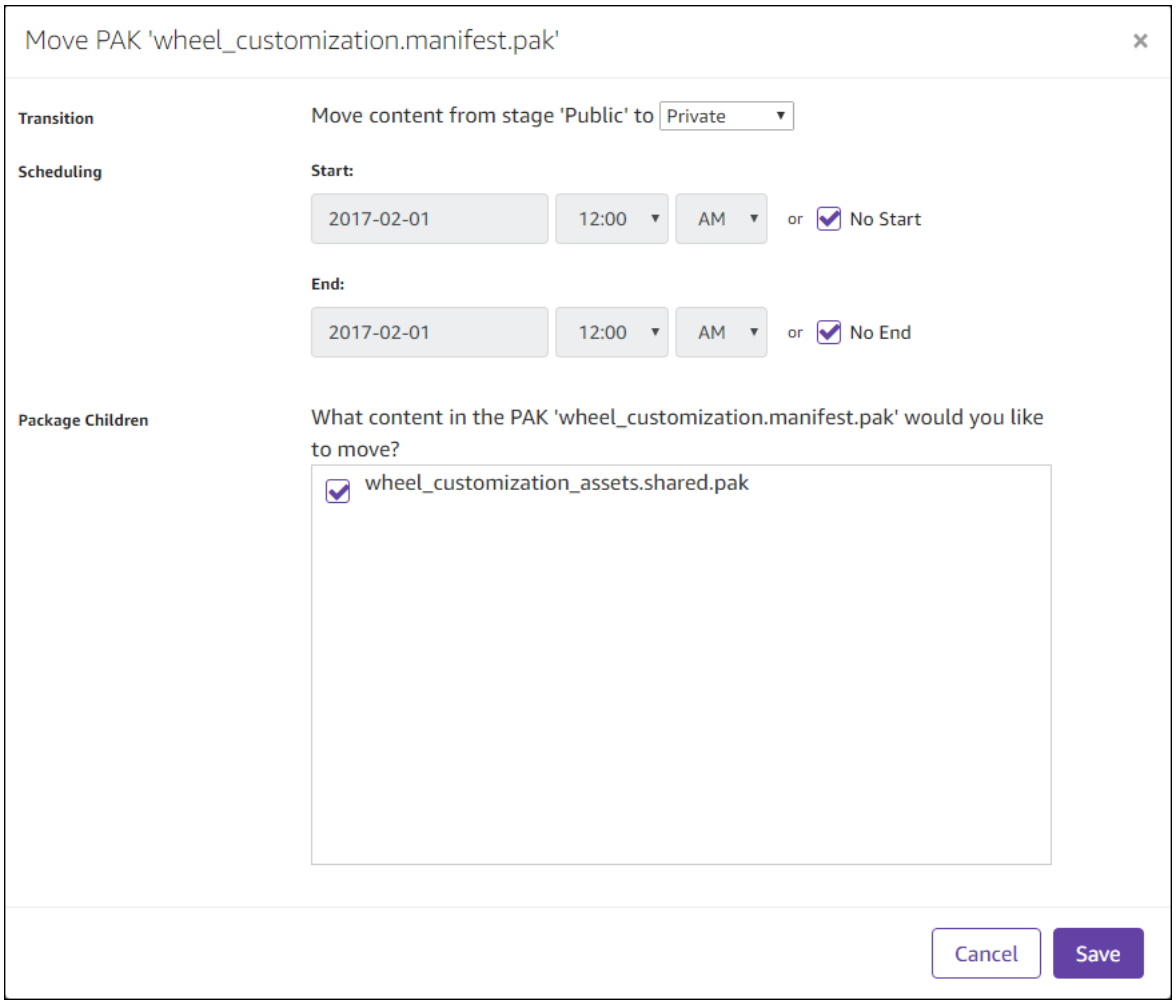
Icon	Description
	You can edit or delete root level packages.
	You can only delete child packages.

Editing Packages

Editing is triggered when you drag and drop a root-level package from one stage to another.



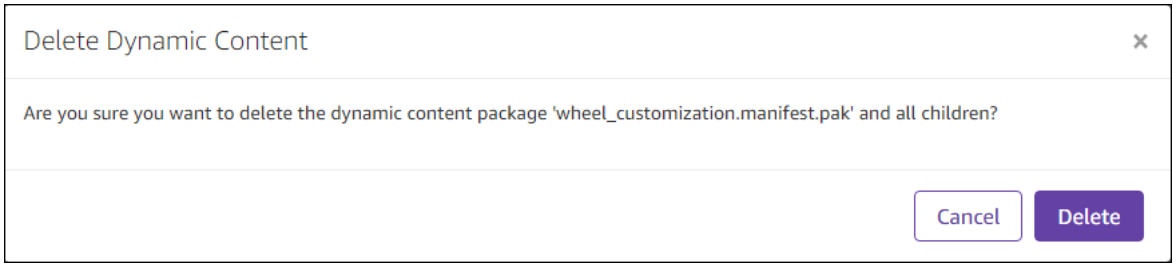
When you edit a package, you can alter its **Transition**, **Scheduling**, and **Package Children**, as the following image shows.



- **Transition** – Choose the target stage that you want to move the package to.
- **Scheduling** – Set packages to `scheduled` or `indefinite`. Packages that you place in the public stage cannot have a date; they are always indefinite.
- **`Package Children`** – `Select which children of the root package you would like to include in this transition. By default all children move with the root package. If you clear the selection for some children, they become orphans in the current stage and no longer visible to the game client.`

Deleting Packages

Deleting a root package causes its child packages to become orphans that are no longer visible to the game client. Accordingly, when you delete a package, the following confirmation message appears.



Displaying Package Metadata

To display the metadata for a package, click the space next to the package name. The metadata for the package appears at the bottom of the browser.

wheel_customization.manifest.pak

File state: PUBLIC Description:

File parent:

Start:

Stop:

Details:


```
{
  "StagingEnd": null,
  "StagingStatus": "PUBLIC",
  "StagingStart": null,
  "FileName": "wheel_customization.manifest.pak",
  "children": [
    {
      "StagingEnd": null,
      "StagingStatus": "PUBLIC",
      "StagingStart": null,
      "Parent": "wheel_customization.manifest.pak",
      "FileName": "wheel_customization_assets.shared.pak",
      "children": [],
      "checked": true
    }
  ]
}
```


Using lmbr_aws for Dynamic Content

When you enable the Dynamic Content Cloud Gem, associated commands become available in the `lmbr_aws` CLI. You can use these commands to update dynamic content and perform operations on manifest files and buckets.

Updating Dynamic Content with lmbr_aws

To perform dynamic content updates, type the following command.

```
lmbr_aws dynamic-content upload-manifest-content --manifest-path <manifest name> --staging-status <PUBLIC|PRIVATE>
```

- *<manifest name>* – specifies the name of the manifest in the `manifests` directory (for example, `DynamicContentTest.json`).
- *<PUBLIC|PRIVATE>* – (Optional) Specify PUBLIC to release the content immediately. The default is PRIVATE.

> **Note**
> For information on using the Lumberyard user interface to upload content, see Using Dynamic Content Manager (p. 220) and Managing Dynamic Content Packages (p. 228).

lmbr_aws Extensions

The following `lmbr_aws` CLI commands are enabled by the Dynamic Content Cloud Gem.

add-manifest-file

Adds a file to the content manifest for the project.

The command has the following arguments:

- `--file-name` *<filename>* – (Required) The name of the file, including the local directory (for example, `staticdata/csv/gameproperties.csv`, where local directory is `staticdata/csv/`.
- `--cache-root` *<cache directory>* – The reference for the local cache directory (for example, `@assets@`). The default is `@assets@`.
- `--bucket-prefix` *<prefix>* – The bucket prefix under which to store the file in the content bucket.
- `--manifest-path` *<path>* – The path to a manifest other than the default.
- `--output-root` *<default directory path>* – The path to the default directory to which to write. The default is `@user@`.
- `--platform-type` *<platform type>* – The type of platform type to which the asset belongs. The value defaults to the current platform.

compare-bucket-content

Compares manifest content to the bucket contents.

The command has the following argument:

- `--manifest-path` *<path>* – The path to a manifest other than the default.

empty-content-bucket

Empties the content bucket contents.

The command has the following argument:

- `--manifest-path` *<path>* – The path to a manifest other than the default.

list-bucket-content

Lists the manifest files in the content bucket.

The command has the following argument:

- `--manifest-path <path>` – The path to a manifest other than the default.

remove-manifest-file

Removes a file from the content manifest for the project.

The command has the following arguments:

- `--file-name <filename>` – (Required) Removes a file from the content manifest for the project. The file removed matches the value for `--file-name` that is specified in the `add-manifest-file` command.

- `--manifest-path <path>` – The path to a manifest other than the default.

update-manifest

Updates the manifest with current file hashes.

The command has the following argument:

- `--manifest-path <path>` – The path to a manifest other than the default.

upload-manifest-content

Updates the manifest and uploads changed manifest content to the content bucket.

The command has the following arguments:

- `--manifest-path <path>` – The path to a manifest other than the default.

- `--staging-status <PUBLIC|PRIVATE>` – (Optional) Specify PUBLIC to make the new content public immediately. The default is PRIVATE.

- `--all` – (Optional) Updates all content in the manifest regardless of whether it appears to have been updated already.

Testing the Dynamic Content System

Lumberyard contains a basic level called **DynamicContentTest** that you can use to verify that the dynamic content system is functioning properly. The level is located in the `dev\CloudGemSamples\Levels\CloudGemTests\DynamicContentTest` directory.

Prerequisites

To test any cloud gem, you must have a deployment with the necessary resources. These resources are created through the **Cloud Canvas Resource Manager**. For detailed instructions on creating these resources, see Initializing **Cloud Canvas Resource Manager** (p. 382).

For the test, you are likely to play the role of both content creator and game client. Because the editor prefers loose files to `.pak` files, you might want to run the engine in game client mode instead of editor mode. When you run the engine in editor mode, updated data does not take precedence during run time. If you do want to run the test in editor mode, you can change this setting by typing the console

command `sys_PakPriority=1` before you start the test. After testing, if you want the editor to return to preferring loose files (the default), type the console command `sys_PakPriority=0`.

To run the level as a game client, you must export the mappings from your deployment stack to your *<projectname>*`\config` directory. To export the mappings, select your deployment in **Cloud Canvas Resource Manager** and click **Export Mapping**. The mappings file has the format *<deploymentname>*`.awslogicalmappings.json`. Because old mapping files can create conflicts, ensure that the `config` directory contains only the single mappings file that you want to use.

The test level contains a button that requests an update of a predefined dynamic content manifest that describes a single `.pak` file. The `.pak` file references a `gameproperties.csv` file. The static data system uses the properties file to reflect a property value to Lua and to a text control that displays the property value.

When the level starts, the text box displays the current value of the property. Pressing the button starts the update process. During the update, the client checks the dynamic content system for new data. When it detects new data, it downloads the data and reflects the new value.

Data Preparation

To create data that is considered "new" for the purposes of the test, perform the following steps.

To prepare data for the test

1. Open the `gameproperties.csv` file in the `dev\CloudGemSamples\StaticData\CSV` directory. The file contents are similar to the following:

```
GameProperty,Value
DynamicMessage,"LOCAL DATA"
```

2. Set the `"LOCAL DATA"` message to a string that you want to receive when the data is updated (for example, `"UPDATED"`).
3. Save the file.
4. Do one of the following:

   - At a command line prompt, type the following command:

```
lmbr_aws dynamic-content upload-manifest-content --manifest-path
 DynamicContentTest.json --staging-status PUBLIC
```

   - Use the packer tool to upload the manifest content. The manifest file that packs up `gameproperties.csv` is `DynamicContentTest.json`. Normally you should not have to alter `DynamicContentTest.json` for testing purposes. However, if you want to use it to ensure that your test contains no old test data, configure it like this:

```
{
    "Files": [
        {
            "bucketPrefix": "",
            "cacheRoot": "@assets@",
            "hash": "",
            "keyName": "gameproperties.csv",
            "localFolder": "StaticData/CSV",
            "outputRoot": "@user@",
            "pakFile": "",
            "platformType": ""
```

```
        }
    ],
    "Paks": [
    ]
}
```

At this point, you have created and uploaded a `.pak` file that contains the current version of your `gameproperties.csv` file and your manifest.

5. Stage the uploads as **PUBLIC**, which means that they are ready for download. If you used the command line option in the previous step to upload your data, this was done for you with the `--staging-status PUBLIC` option. If you used the packer tool to perform the upload, open the Cloud Gem Portal from your AWS menu, open your Dynamic Content Gem, and move the ***DynamicContentTest.manifest.pak*** file from the **Private** column to the **Public** column.
6. Change your `gameproperties.csv` file message back to `"LOCAL DATA"` or its original message. Do this so that you can be sure that your updated message comes from the downloaded content.

Running the Test

After you have saved the `gameproperties.csv` file to reflect local data, you can run the test.

To test the dynamic content system

1. Start the **DynamicContentTest** level. When you start the level, you should see the most recent message that you entered in the game properties file ("LOCAL DATA" in step 6 of Data Preparation (p. 236)).
2. Click **Update**. You should see the update in 3 to 10 seconds, depending on Lambda function latency.

Diagnosing Failures

When diagnosing a failure, you must determine how far your updated data went in the processing chain.

Check the .pak File for Upload

First, open the `.pak` file that was to be uploaded with 7-Zip or a similar utility. The file is in the location `CloudGemSamples\DynamicContent\Paks\DynamicContentTest.shared.pak`. The `.pak` file contains the `.csv` file that the end client should have received. If something is wrong with the `.pak` file, the packing process was faulty and should be repeated. If so, delete the `.pak` file. In your `gameproperties.csv` file, edit the message you want. Configure the `DynamicContentManifest.json` as indicated in the Data Preparation (p. 236) section earlier in this document. Run the upload command again. Finally, recheck that your `.pak` file has the correct `.csv` file and data.

Check the Downloaded .pak File

If your local `.pak` file for upload is free of errors, check the file that was downloaded. The downloaded file is located in the user directory for your game project (for example, `CloudGemSamples\pc\user\dynamiccontent\paks\DynamicContentTest.shared.pak`). Perform the same inspection on this file as you did for the file in the upload location. If the data is what you expect, the client did not receive the new data. This can occur because of a download failure, or because the data in the Amazon S3 bucket is incorrect.You can use the AWS console to inspect the corresponding files in your Amazon S3 bucket.

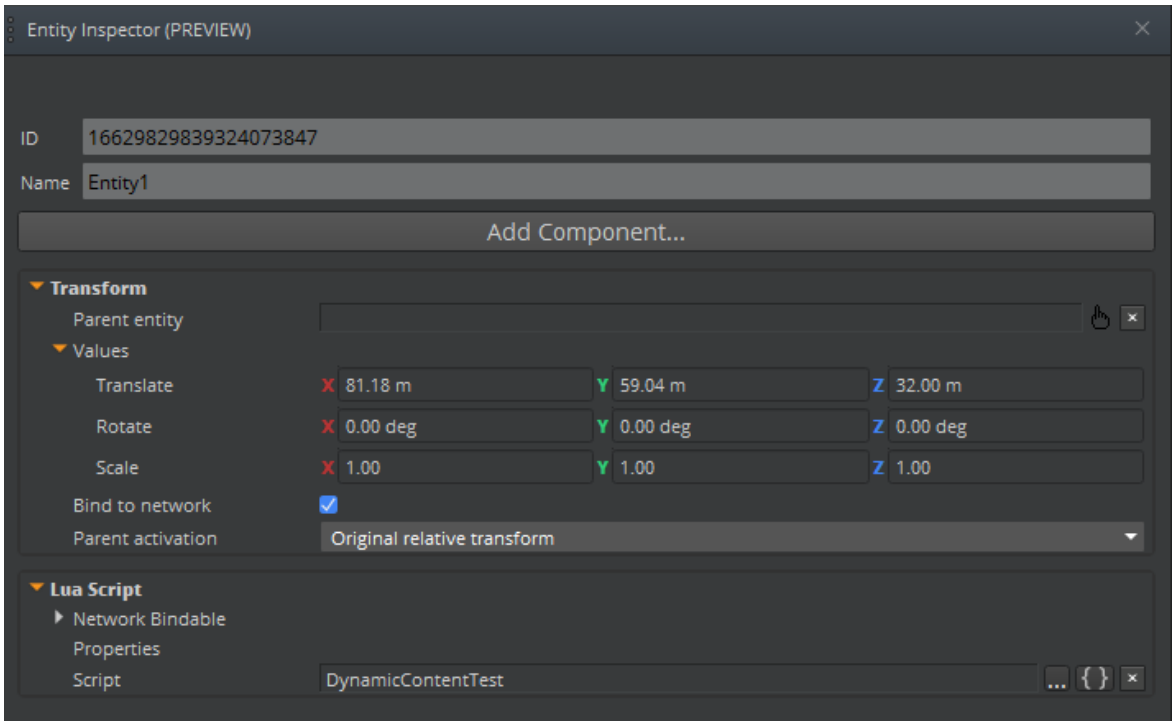
If the downloaded `.pak` file was updated correctly, then the problem might be one of the following:

- The dynamic content system itself did not reload the file.
- The dynamic content system did not accept the new data. Check whether `sys_PakPriority` is set to `0`. In editor mode, you must set this to `1`. For more information, see the Prerequisites (p. 235) section earlier in this document.

- The test is not reflecting the file correctly and requires further diagnosis.

Test Implementation Details

Because `DynamicContent` and `StaticData` are system components, they do not have to be added to the entity in the test level. Since in most cases the dynamic content that Lumberyard loads would persist longer than any entity, the example here shows you how to use the `SystemComponent`. The entity in this level contains only Lua script and test logic.



1. In the script, the example connects to the EBus, loads the canvas, and waits for the button event.

```
function DynamicContentTest:OnAction(entityId, actionName)
    Debug.Log("Received Action: " .. actionName)

    if actionName == "ButtonPressed" then
        if DynamicContentRequestBus.Event == nil then
            Debug.Log("No Content Request Events found")
            return
        end
        DynamicContentRequestBus.Event.RequestManifest(SystemEntityId,"DynamicContentTest.json")
        self:UpdateText()
    end
end
```

When the button is pressed, the code requests the new manifest. The content specified by the manifest is automatically retrieved.

2. After the content is retrieved, a `NewPakContentReady` event fires.

```
function DynamicContentTest:NewPakContentReady(outputFile)
    Debug.Log("Received pak update: " .. outputFile)
    self:UpdateGameProperties()
end
```

In the test example, you already know that the `.pak` file contains game properties data. In more complex implementations, you might need to query the contents of the file.

3. The `UpdateGameProperties()` function requests that the static data system load the `.csv` file.

```
function DynamicContentTest:UpdateGameProperties()
    if StaticDataRequestBus.Event == nil then
        Debug.Log("No StaticData Request Events found")
        return
    end
    StaticDataRequestBus.Event.LoadRelativeFile(SystemEntityId,"StaticData/CSV/gameproperties.csv")
end
```

4. The file load triggers the `TypeReloaded` event, which calls the `UpdateText()` function.

```
function DynamicContentTest:TypeReloaded(outputFile)
    Debug.Log("Static Data type reloaded:" .. outputFile)
    if outputFile == "gameproperties" then
        self:UpdateText()
    end
end

function DynamicContentTest:UpdateText()
    local displayEntity = UiCanvasLuaBus.Event.FindElementById(self.canvasEntityId, 4)
    local wasSuccess = false
    local returnValue = StaticDataRequestBus.Event.GetStrValue(SystemEntityId,"gameproperties","DynamicMessage", "Value", wasSuccess)
    Debug.Log("Got DynamicMessage value: " .. returnValue)
    UiTextBus.Event.SetText(displayEntity, returnValue)
end
```

At this point, the user interface text has been updated with the new game property value.

5. Because the example attached to the `SystemComponent`, and the sample might be running in the editor, the cleanup routine manually removes the dynamic content.

```
function DynamicContentTest:OnDeactivate()
    DynamicContentRequestBus.Event.ClearAllContent(SystemEntityId)
    self.uiCanvasNotificationLuaBusHandler:Disconnect()
    self.dynamicContentUpdateBus:Disconnect()
    self.staticDataUpdateBus:Disconnect()
end
```

Dynamic Content Engineering Details

This topic provides programmatic details about the dynamic content update process. This includes manifest file information, Dynamic Content Cloud Gem EBus events, and Dynamic Content Cloud Gem service API. For information about the `lmbr_aws` CLI extensions enabled by the Dynamic Content Cloud Gem, see Using lmbr_aws for Dynamic Content (p. 233).

Manifest File

In your Lumberyard installation, the default location of the manifest file is *<GameFolder>*`/AWS/DynamicContent/DynamicContentManifest.json`.

The following is a simple example manifest for the `SamplesProject DontDie` sample.

```
"Files": [
{
"hash": "3bebdb5bdb8cff74642e5f7f3dc4e900",
"outputRoot": "@user@",
"bucketPrefix": "static-data",
"keyName": "gameproperties.csv",
"cacheRoot": "@assets@",
"platformType": "",
"localFolder": "staticdata/csv"
}
]
```

The following table describes the properties in the manifest file.

Property	Description
`hash`	MD5 hash of the file.
`outputRoot`	Base output directory.
`bucketPrefix`	Prefix inside the bucket for the file
`keyName`	Name of the file key in the bucket, which will be appended to the beginning of the hash. The final key name has the format `bucketPrefix/keyName`.
`cacheRoot`	Root directory to search for copies of the outdated file asset.
`platformType`	Windows (`pc`), macOS (`osx_gl`), or Linux (`linux`). An empty value specifies all platforms.
`localFolder`	Directory to write locally within the `outputRoot`. The full output has the format `outputRoot/localFolder/keyName`.

EBus Events

The Dynamic Content Cloud Gem provides an EBus API and includes calls exposed to Lua. The basic top-level update request looks like this:

```
EBUS_EVENT_RESULT(requestSuccess,
      CloudCanvas::DynamicContent::DynamicContentRequestBus,
      RequestManifest, manifestName)
```

`requestSuccess (bool)` – Specifies whether the request was successfully sent.

`manifestName(char*)` – Specifies the plaintext name of the manifest (for example, `DynamicContentTest.json`). The system handles `.pak` file and platform naming conventions (for example, `DynamicContentTest.shared.pak`).

Note
A legacy **RequestManifest** flow graph node exists, which calls the same EBus.

Manifest Received

The following EBus events are triggered when a manifest has been received successfully or unsuccessfully.

Success

```
EBUS_EVENT(CloudCanvas::DynamicContent::DynamicContentRequestBus, ManifestUpdated,
 bucketName, bucketPrefix)
```

When all `.pak` files are complete, a `RequestCompleted` event is broadcast.

Failure

```
EBUS_EVENT(CloudCanvas::DynamicContent::DynamicContentRequestBus, ManifestFailed,
 bucketName, bucketPrefix, errorStr)
```

Service API

The Dynamic Content Cloud Gem exposes API calls through Amazon API Gateway for both the Cloud Gem Portal and the game client.

The following tables list the calls for the portal.

Portal API Call	Description
`/service/status GET`	Returns the service's status.
`/portal/info/{file_name} GET`	Return detailed information about a specific file. This includes the file's name, staging status, staging start and end dates (optional), and parent (optional).
`/portal/info/{file_name} DELETE`	Request deletion of an existing item from the bucket and table.
`/portal/content GET`	Request the list of files to display in the web portal.
`/portal/content DELETE`	Request to deletion of all content from the bucket and staging table.
`/portal/content POST`	Request alteration of the staging settings on a provided list of files.

The following table lists the calls for the client.

Client API Call	Description
`/client/content POST`	Request presigned URLs for a list of files. Returns the URLs or a failure message.

Player Account Cloud Gem

The Player Account Cloud Gem provides a standalone player authentication and management solution that you can customize to fit your needs.

Prerequisites

This tutorial assumes the following:

- You are using a Lumberyard project that has the Player Account Cloud Gem enabled (in Project Configurator, select **Cloud Gem Player Account**). The CloudGemSamples project already has this gem enabled.
- You have created a project stack in Cloud Canvas Resource Manager (p. 182).
- You have created a deployment stack in Resource Manager.

If you don't meet the prerequisites, follow the steps in Setting Up and Accessing the Cloud Gem Portal (p. 201) to add **Cloud Gem Player Account** in Project Configurator.

The Player Account Sample Level

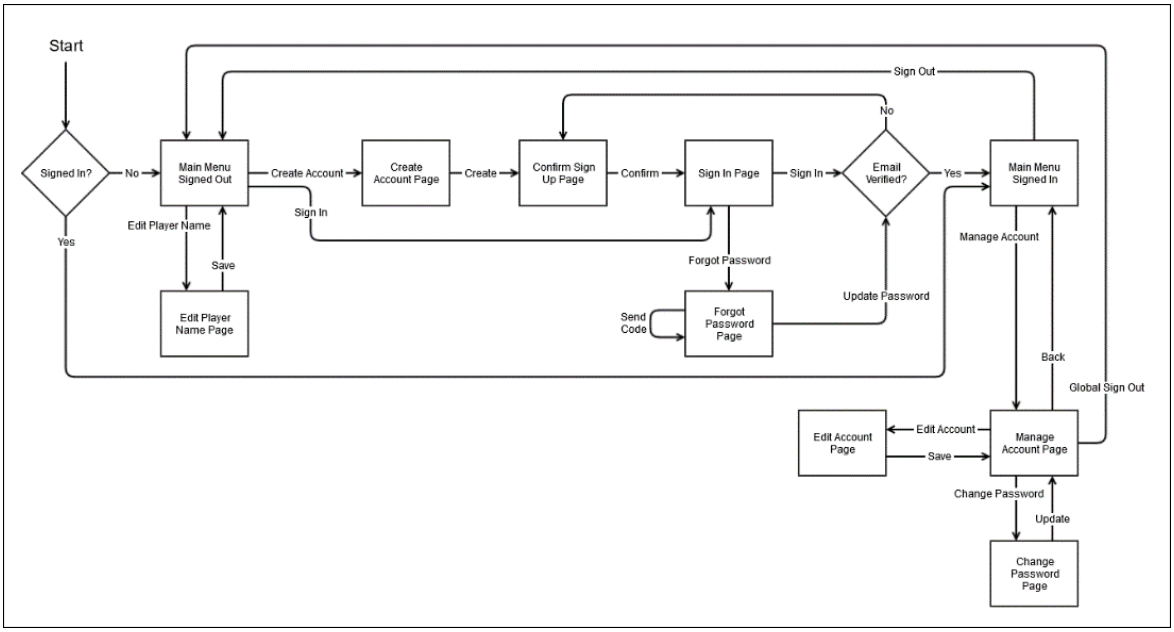
The CloudGemSamples project contains a sample level named `PlayerAccountSample`. The player account sample level provides a menu from which users can create an account, verify their email addresses, sign in and out of their accounts, and recover their passwords by email.

The following table describes the files for the player account sample level.

Location	Description
`\dev\CloudGemSamples\Levels\PlayerAccountSample`	A level that contains an entity with the Lua script component.
`\dev\CloudGemSamples\Scripts\PlayerAccount`	The Lua scripts that manage the UI canvases and interact with the Player Account Cloud Gem's EBus API.
`\dev\CloudGemSamples\UI\Canvases\PlayerAccount`	The UI canvases displayed for the player.

Note that in the sample, UI canvases do not include functionality for displaying error messages graphically; instead, error messages are output to the console. If you integrate this gem into an existing project, you should use the UI canvas to show error messages. For example, you could do this when the confirmation code is incorrect or the password does not meet the minimum password length.

The following diagram shows how the UI canvases are connected. For more information on the Amazon Cognito user pool signup process, see the Signing Up and Confirming User Accounts page in the Amazon Cognito Developer Guide.



Integrating the Player Account Cloud Gem into an Existing Project

To integrate the Player Account Cloud Gem into an existing project, you must enable the Player Account Cloud Gem, update your deployment stack, and integrate the gem into your game menu.

To integrate the Player Account Cloud Gem into an existing project

1. Enable the gem. In Project Configurator, click **Cloud Gem Player Account**.
2. Create or update the project stack (p. 383).

   The Player Account Cloud Gem uses an Amazon Cognito user pool. If you created your project using a version of Lumberyard earlier than 1.9, update it by opening a command prompt window in the \dev directory and typing the following command.

```
lmbr_aws project update-code
```

3. Create or update the deployment stack (p. 385). Be sure the `CloudGemPlayerAccount` resource group is added to your deployment. If your deployment doesn't have the `CloudGemPlayerAccount` resource group, make sure that the Cloud Gem Player Account gem is enabled in Project Configurator.
4. At a command prompt, type the following command to update the deployment access stack.

   ```
   lmbr_aws deployment update-access
   ```

   This ensures that the `CloudGemPlayerAccount` resource group is correctly linked to the `PlayerAccess` identity pool in the deployment access stack.
5. Integrate the gem into your game menu. We recommend that you include the following features:

   - **A global signout** – This security feature allows players to invalidate an account's tokens across all devices if a device has been lost or stolen. The sample level shows how to offer this as a standalone feature. If the previous password was compromised and already used to obtain authentication tokens, you can also have the menu automatically globally sign out after a password change. To implement this functionality, do the following:
     1. Monitor the `CloudGemPlayerAccountNotifications::OnChangePasswordComplete` EBus event.
     2. When you receive confirmation that the password change has succeeded, send the following event.

        ```
        EBUS_EVENT(CloudGemPlayerAccountRequestBus, GlobalSignOut, username)
        ```

   - **Email verification** – An email verification step ensures that email can be used to recover account access if a password has been forgotten. You can customize the content of the verification emails by using Lambda triggers. To add a new Lambda trigger to the AWS CloudFormation template, use the `LambdaConfig` property of `PlayerUserPool` in the `dev\Gems\CloudGemPlayerAccount\AWS\resource-template.json` file. For more information, see Customizing User Pool Workflows by Using AWS Lambda Triggers.

Building the Menus

To build the menus, use the code for `CloudGemPlayerAccountRequestBus` and `CloudGemPlayerAccountNotificationBusHandler` in the `dev\Gems\CloudGemPlayerAccount\Code\Include\CloudGemPlayerAccount\CloudGemPlayerAccountBus.h` file. Keep in mind the following points:

- Most of the EBus events are wrappers for Amazon Cognito user pool API operations. For the Amazon Cognito API reference, see Amazon Cognito Identity Provider.
- To return the user name for the currently logged-in user, call the `GetCurrentUser` function, which has a corresponding `OnGetCurrentUserComplete` event. This function and EBus event are not part of the Amazon Cognito user pool API but are needed for most calls to the API.
- The `GetCurrentUser` function also loads the returned user's credentials and caches them inside the gem's system component. As a result, subsequent calls to `GetCurrentUser` are faster.
- The act of signing in also caches the user's credentials inside the gem's system component.

Storing Data Associated with a Player Account

Each player account has a randomly generated account ID. When you store data associated with a player, use this ID instead of the identifiers provided by the Amazon Cognito user and identity pools. You should do this for the following reasons:

- Lumberyard uses the `PlayerAccounts` table defined in the `dev\Gems\CloudGemPlayerAccount\AWS\resource-template.json` file to map account IDs to Amazon Cognito IDs. By associating a player's data with the account ID, you can move all of the player's data from one identity to another by simply updating the mapping.

- The Amazon Cognito ID on the Amazon Cognito identity pool is not guaranteed to stay the same for a particular player. For example, it can change if identities are merged. Amazon Cognito identity pools are public facing and do not prevent the merging of identities.

- If you configure the `PlayerAccess` identity pool to support identity providers other than the Amazon Cognito user pool, the user pool's user name cannot be used for players that use another provider.

- If a user name is deleted from the Amazon Cognito user pool, the name can be reused. Amazon Cognito user pools are publicly visible and do not prevent a user from deleting his or her own user name.

A `PlayerName` field is provided in the `PlayerAccounts` table as an example of how to store player data for an account. The Amazon Cognito user pool supports storing a variety of attributes and could also be used for storing player data. However, this is not recommended if you configure the `PlayerAccess` identity pool to use more than one identity provider. In such a scenario, some players might not have a user in the Amazon Cognito user pool.

Warning
Do not store the data for unauthenticated (guest or anonymous) players who are using an Amazon Cognito ID directly or indirectly on the server. The reasons not to store this data are the following:

- A person who knows the Amazon Cognito ID for an unauthenticated account can claim the identity by linking it to a user that the person controls. The identity's original owner cannot recover the account if an original device is lost or the identity is stolen. For more information on identity pools, see Identity Pools.

- The local copy of the unauthenticated Amazon Cognito ID is overwritten with a new one when the player logs in and out.

- The player has no way to recover an unauthenticated Amazon Cognito ID after it has been overwritten locally. The identity still exists in the identity pool, but the identity and data associated with its Amazon Cognito ID effectively become orphaned.

- The buildup of orphaned identities can be misleading when trying to determine the actual number of players.

- If a player uses an unauthenticated identity and then creates a new account, the account ID from the unauthenticated identity does not carry over to the new account.

Obtaining Player Account IDs

You can use the following techniques to obtain player account IDs and account information programmatically.

- **To get a player's account ID in game** – Call `GetPlayerAccount` on the `CloudGemPlayerAccountRequestBus`. This EBus is defined in the `dev\Gems\CloudGemPlayerAccount\Code\Include\CloudGemPlayerAccount\CloudGemPlayerAccountBus.h` file. The player's account ID is in the `AccountResultInfo` parameter of `CloudGemPlayerAccountNotificationBusHandler::OnGetPlayerAccountComplete`.

- **To get the account ID from a Lambda function located behind an API Gateway** – Because Amazon API Gateway provides the Amazon Cognito ID to the Lambda function, the Lambda function can look up the account ID. To quickly look up the account ID, you can use the global secondary index `CognitoIdentityIdIndex` of the `PlayerAccounts` table. The `PlayerAccounts` table is defined in the `dev\Gems\CloudGemPlayerAccount\AWS\resource-template.json` file.

- **To get the caller's account information without relying on the caller to provide its account ID** – Use the request's authentication parameters. For an example, see the code in the `dev\Gems\CloudGemPlayerAccount\AWS\lambda-function-code\api\account.py` file.

Cloud Gem Portal

The Player Account Cloud Gem integrates with the Cloud Gem Portal (p. 200). You can use the portal to search for player accounts and update their information. The portal includes some default Amazon Cognito user pool attributes that you can view and edit, such as given name, middle name, and family name. To change the attributes that can be viewed or edited, you must modify the gem code.

Amazon Cognito Client Apps

The Player Account Cloud Gem has a single client app named `DefaultClientApp` that the sample level and the `PlayerAccess` identity pool use. If you require more client apps, you can add them to the `PlayerUserPool` resource in the `dev\Gems\CloudGemPlayerAccount\AWS\resource-template.json` file.

Testing the Player Account Cloud Gem

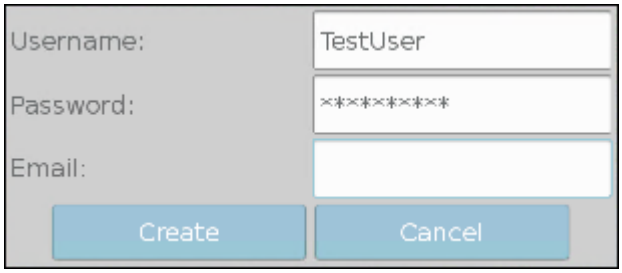
To test the Player Account Cloud Gem, you can use the CloudGemSamples project that is included with Lumberyard.

To test the Player Account Cloud Gem

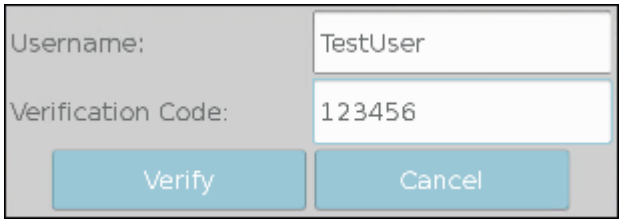
1. In Project Configurator, select the **CloudGemSamples** project.
2. Start Lumberyard Editor.
3. On the **Welcome to Lumberyard Editor** screen, click **Open level**, **Levels**, **PlayerAccountSample**, **Open**.
4. Click **AWS**, **Cloud Canvas**, **Select a Deployment**.
5. Click the deployment that you want to use, and then click **OK**.
6. Press **Ctrl+G** to start the game.
7. In the main menu of the sample, click **Create Account**.

Note
If the **Create Account** option does not appear, click **Sign Out** and try again.

8. Type a user name, password, and an email account to which you have access.



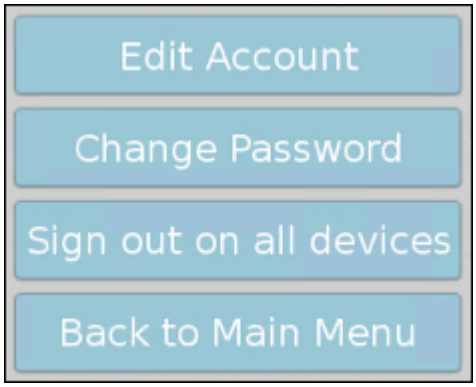
9. Click **Create**.
10. From the email account, copy the confirmation code that you receive.
11. In Lumberyard, paste the confirmation code into the **Verification Code** box, and then click **Verify**.



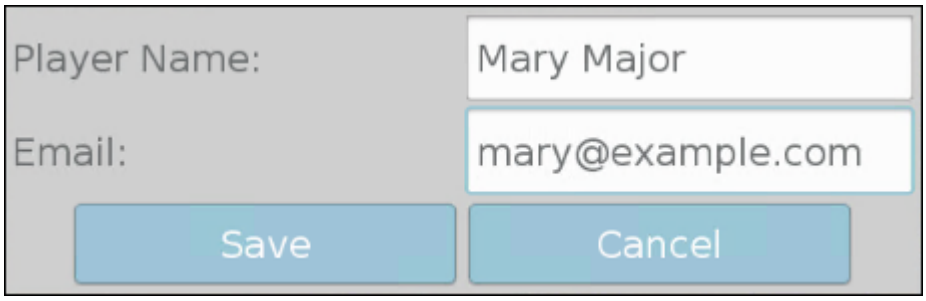
12. Sign in using the password that you specified earlier. This tests the custom authentication flow.

13. Click **Manage Account**.

14. Click **Edit Account**.

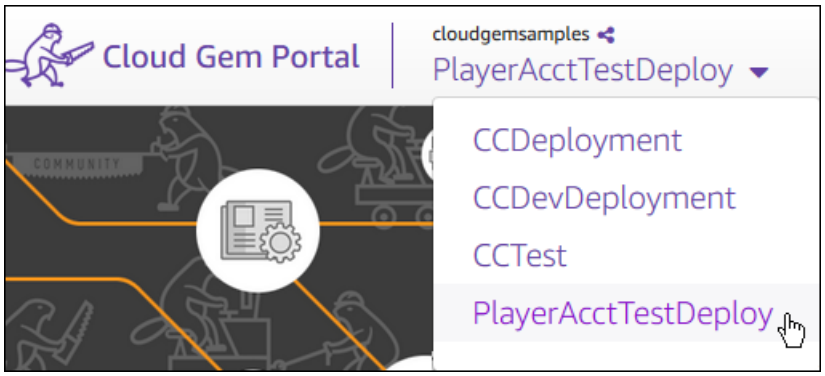


15. Type in a name for **Player Name**, and then click **Save**. This step tests the player service API.

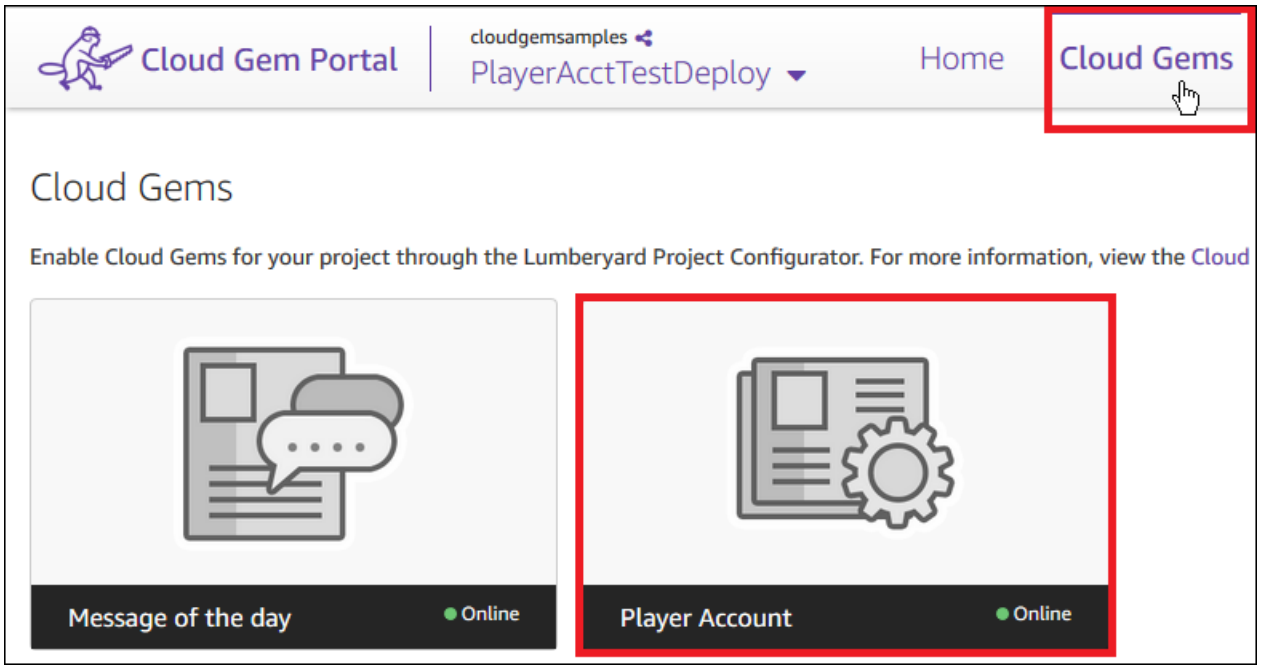


The main menu displays the player name that you specified.

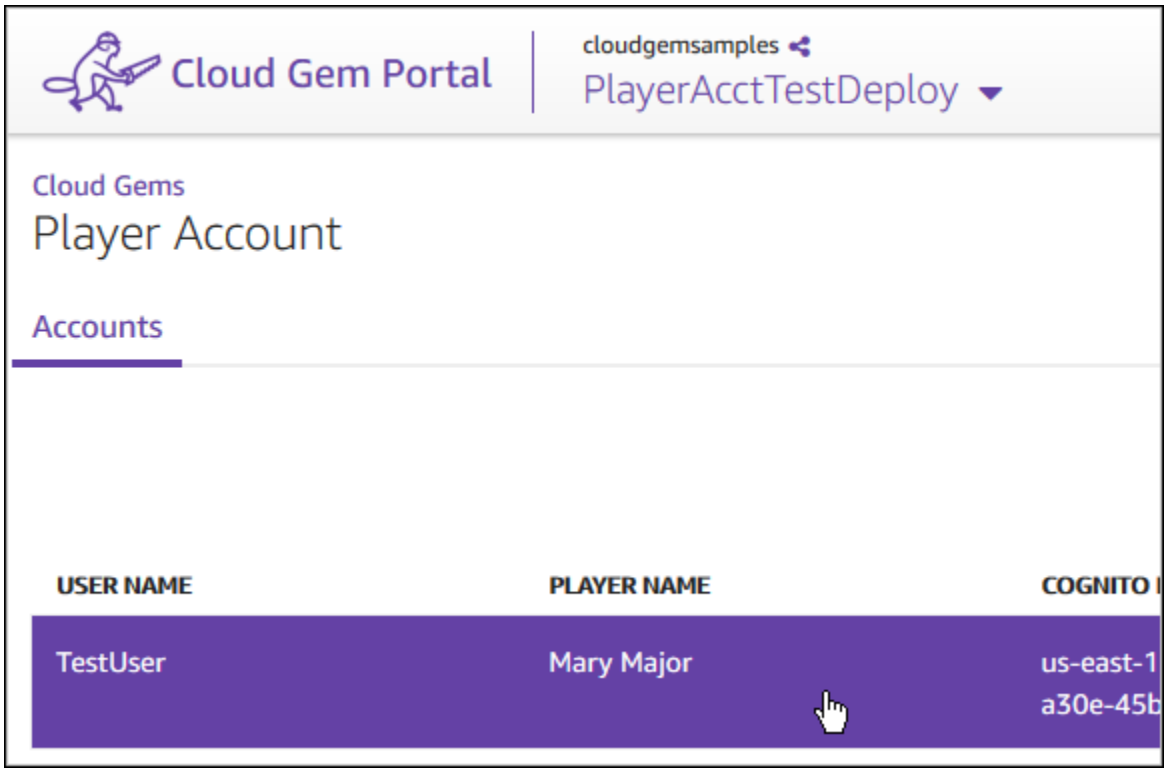
16. Press **ESC** to stop the game.
17. In Lumberyard Editor, click **AWS**, **Open Cloud Gem Portal**.
18. In the Cloud Gem Portal, choose the deployment that you are using for the test.



19. Click **Cloud Gems**, and then click **Player Account**.

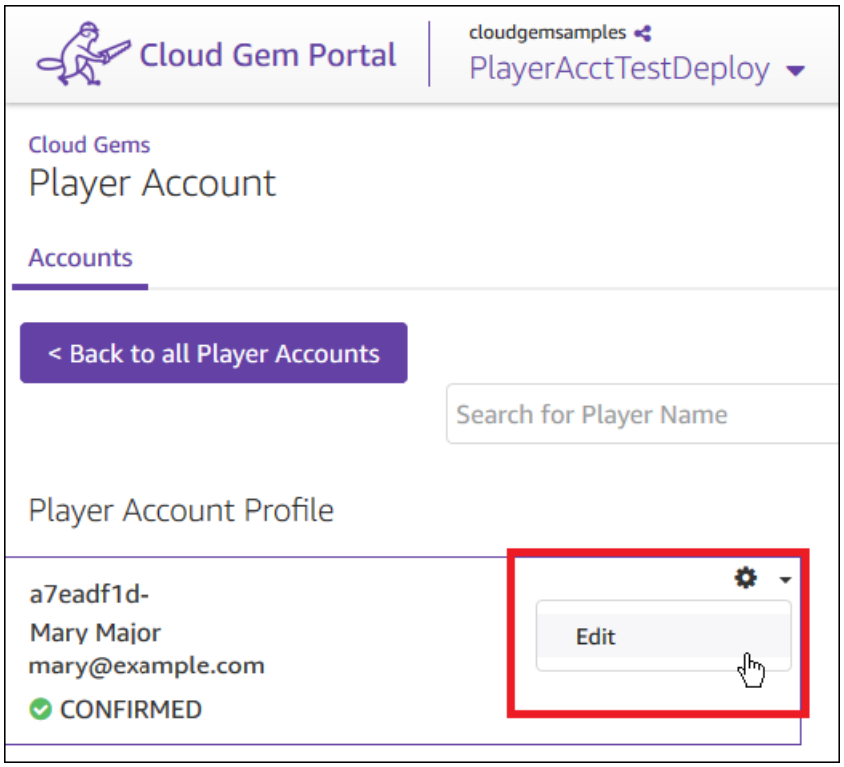


20. The gem displays the account that you created.

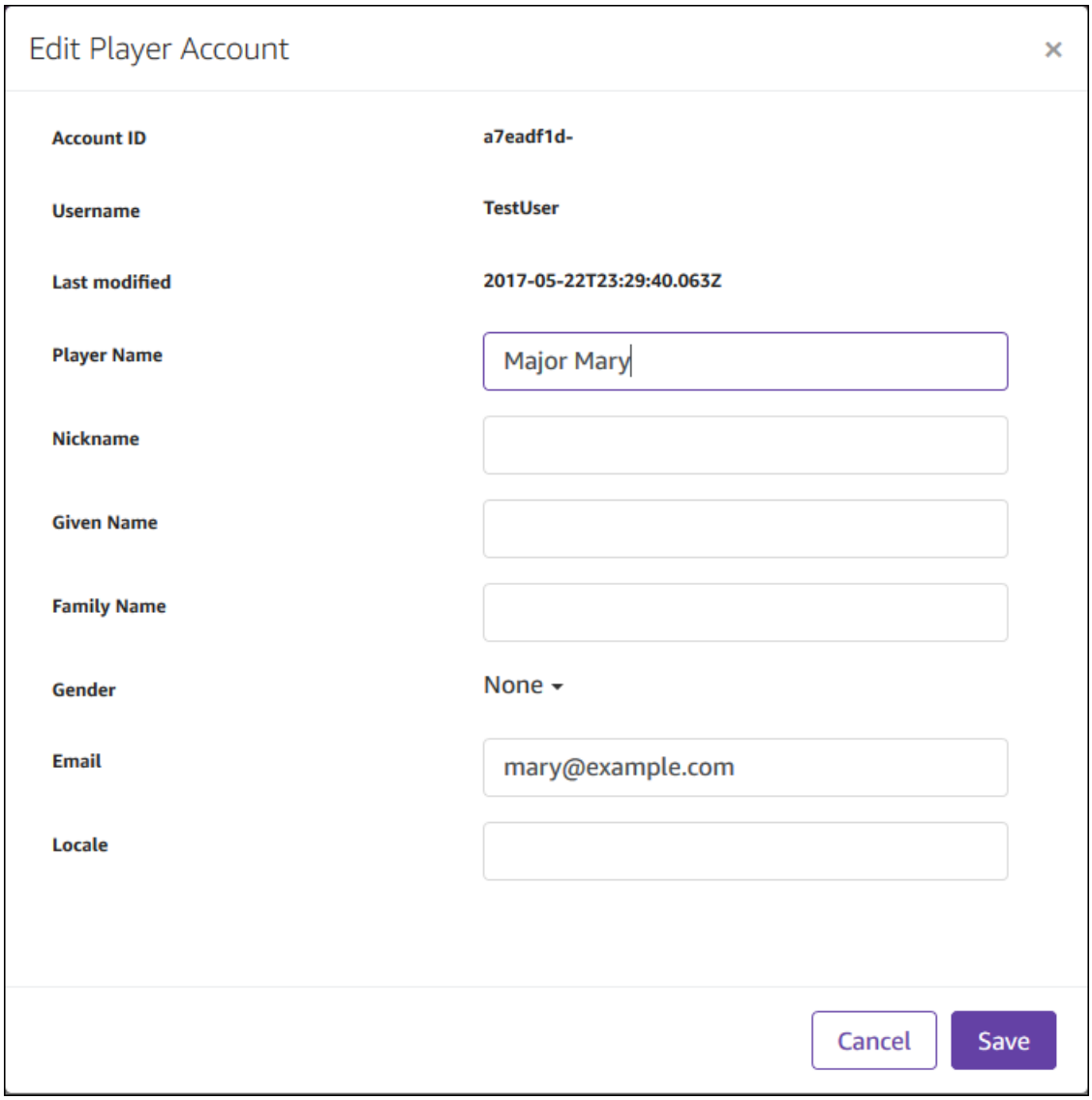


Click the account so that you can edit its information. These steps test the administrative service API.

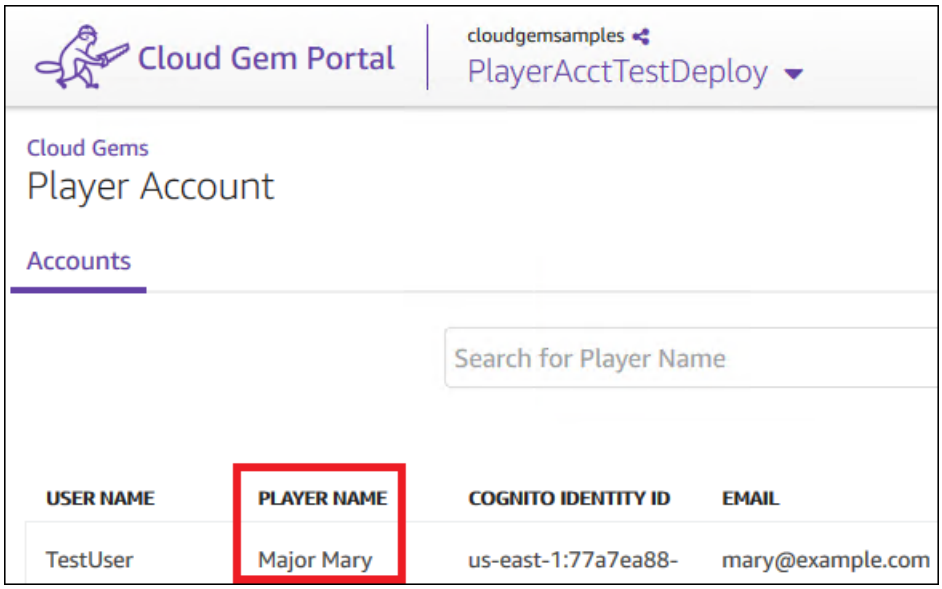
21. In **Player Account Profile**, click the down arrow, and then click **Edit**.



22. Make a change to the player name, and then click **Save**.



23. Verify the change in the profile for the player.



Player Account Cloud Gem Implementation Details

You can use the API operations described on this page to manage user accounts with the Player Account Cloud Gem (p. 241).

Functions and Callbacks

Most functions for the Player Account Cloud Gem have a corresponding callback that supplies the results of the call. The results contain the information gathered or error information. To make calls to and receive responses from user registration and login component API operations, you use the Event Bus (EBus) (p. 543).

The callbacks use this naming convention: For every function *x*, the corresponding callback function is named `On`*x*`Complete`. For example, when a response to an EBus call of `SignUp()` is ready, the response is sent through the EBus to `OnSignUpComplete()`. For more examples, see `dev\Gems\CloudGemPlayerAccount\Code\Include\CloudGemPlayerAccount\CloudGemPlayerAccountBus.h`.

`HasCachedCredentials()` is an exception to this convention because it is a synchronous call and has no corresponding handler.

Functions for Key Player Account Cloud Gem Tasks

You can use the functions described in this section for new account creation, password recovery, user sign-in, and account management. Most of the source code for the classes and functions can be found in the Lumberyard directory `\dev\Gems\CloudGemPlayerAccount\Code\Source`.

Creating an Account (CloudGemPlayerAccountBus)

Use the following functions for creating new accounts.

```
SignUp(username, password, attributes)
```

Creates an account. The `attributes` parameter contains relevant data (for example, email, address, name, and profile). A confirmation email or text message is sent to the user as part of the signup. Password recovery requires a valid email address.

`OnSignUpComplete(resultInfo, deliveryDetails, wasConfirmed)`

A callback function that returns the results (p. 253) of the signup. The deliveryDetails (p. 253) attribute contains details about where the confirmation code was sent.

`ResendConfirmationCode(username)`

Resends the confirmation code to the user. The confirmation code received from the user is passed as an argument to `ConfirmSignUp(username, confirmationCode)`.

Password Recovery (CloudGemPlayerAccountSystemComponent)

Use the following functions if a user has forgotten a password and has an account with a valid email or phone number.

`ForgotPassword(username)`

Causes a confirmation code to be sent to the user.

`OnForgotPasswordComplete(resultInfo, deliveryDetails)`

Contains information regarding the password reset.

`ConfirmForgotPassword(username, password, confirmationCode)`

Receives the confirmation code, user name, and new password from the user and sets the new password.

Sign In and Sign Out

Use the following functions to sign users in and out of your game.

`InitiateAuth(username, password)`

Call this function to sign in users. Sign-in is a two-step process. When `InitiateAuth` finishes, the server issues a challenge to which the client must respond correctly. Currently, authorization by user name and password is supported; MFA and other forms of additional authentication are not.

You have two ways to sign out: `SignOut(username)` and `GlobalSignOut(username)`.

`SignOut`

Clears all cached authentication information from memory, so that the user's credentials are essentially forgotten. No server-side changes occur, so the authentication information, if it were known, could still be used. The `SignOut` function always succeeds, even if the user is not actually signed in.

`GlobalSignOut`

Invalidates all access and refresh tokens for the user on all devices that the user might be signed into. Unlike `SignOut`, `GlobalSignOut` can succeed only when the user is signed in.

Account Management (CloudGemPlayerAccountSystemComponent)

All of the following functions require that the user be signed in.

`ChangePassword(username, previousPassword, proposedPassword)`

Allows a user to update a password without the use of a confirmation code.

`DeleteOwnAccount(username)`

Deletes the user's account. Use this function with caution.

`GetUser(username)`

Retrieves attributes (for example, email, address, name, and profile) that have been associated with the user's account.

`UpdateUserAttributes(username, attributes)`

Updates the specified set of attributes. If any updates require verification, confirmation codes are sent. Details are found in the `deliveryDetailsArray` parameter that is passed to `OnUpdateUserAttributesComplete`. Note that it is possible to update attributes that previously did not have any associated information.

`VerifyUserAttribute(username, attributeName, confirmationCode)`

Following a call to `UpdateUserAttributes`, verifies a single attribute that requires confirmation. Most attribute updates do not require verification.

`DeleteUserAttributes(username, attributesToDelete)`

Removes the specified set of attributes from the user's account. The attributes can be added again with a call to `UpdateUserAttributes`.

Key Player Account Cloud Gem Classes and Functions

This section describes key classes and functions in the Player Account Cloud Gem.

BasicResultInfo Class

The `BasicResultInfo` class bundles together information returned in almost all responses from the component. The class has no functions to be used at run time and contains the following values, which are public:

`wasSuccessful`

A Boolean that indicates whether the request succeeded.

`username`

The name of the user for which a request was made.

`errorTypeName`

The name of the error type.

`errorTypeValue`

The numeric value of the error type. If `errorTypeValue` is greater than or equal to 0, it corresponds to a standard `CognitoIdentityProviderErrors` value. Otherwise, an unexpected error occurred. Most commonly, the user has not yet signed in for the operation. In this case, see the error information in `errorTypeName` for details.

`errorMessage`

A human readable string that describes the error.

DeliveryDetails Class

The `DeliveryDetails` class provides details related to the sending of confirmation codes to the user. it contains the following string functions:

`GetAttributeName()`

The name of the attribute for which a confirmation code was sent (for example, `email`).

`GetDeliveryMedium()`

The medium used to send the confirmation code (for example, `EMAIL`).

`GetDestination()`

The destination to which the confirmation code was sent. The destination is partially obscured for security (for example, `d***@a***.com`).

DeliveryDetailsArray Class

The `DeliveryDetailsArray` class is a collection of `DeliveryDetails` objects and contains the following functions:

`GetSize()`

The number of `DeliveryDetails` objects in the array.

`At(index)`

Returns a copy of the object at the specified index.

UserAttributeValues Class

`UserAttributeValues` maps attribute names like `address`, `email`, and `family_name` to attribute values. An attribute value can be a string up to 256 characters. In general, the attributes are those used by the OpenID Connect specification. See the `UserAttributeValues.h` file for details.

Phone numbers must follow these formatting rules:

- The phone number must start with a plus (+) sign, followed immediately by the country code.
- The phone number can contain only the + sign and digits.
- You must remove characters such as parentheses, spaces, or dashes from the phone number before submitting the value. For example, a United States–based phone number must follow the format `+14325551212`.

Resource Group

The Player Account Cloud Gem uses an Amazon Cognito user pool that contains all registered users. It is set as an authentication provider with `PlayerAccessIdentityPool`. For more information, see Controlling Access to Resources (p. 393).

Cloud Canvas Flow Graph Node Reference

This section provides a reference of the flow graph nodes available for Cloud Canvas.

- Cloud Canvas Configuration Nodes (p. 255)
- Cognito (Player Identity) Nodes (p. 257)
- DynamoDB (Database) Nodes (p. 259)
- Lambda (Cloud Functions) Node (p. 266)
- S3 (Storage) Nodes (p. 266)
- SNS (Notification Service) Nodes (p. 269)
- SQS (Message Queuing Service) Nodes (p. 271)

- Static Data (PROTOTYPE) Nodes (p. 272)

For general information on how to use flow graph nodes, see Flow Graph System.

Cloud Canvas Configuration Nodes

You can use these flow graph nodes to configure Cloud Canvas settings.

Topics

- ApplyConfiguration node (p. 255)
- SetConfigurationVariable node (p. 255)
- ConfigureProxy node (p. 256)
- GetConfigurationVariableValue node (p. 256)
- SetDefaultRegion node (p. 257)

ApplyConfiguration node

Applies AWS configuration to all managed clients.

Inputs

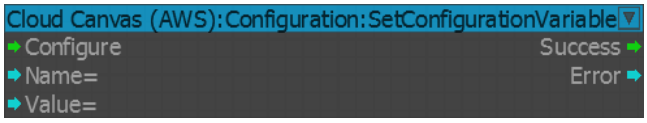
Port	Type	Description
Apply	Any	Applies the current AWS configuration to all managed clients

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message

SetConfigurationVariable node

Sets a configuration parameter value.



Inputs

Port	Type	Description
Configure	Any	Sets the parameter value
Name	String	Name of the parameter to set

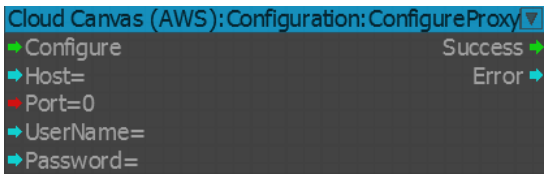
Port	Type	Description
Value	String	Value to which the parameter will be set; may contain `$param-name$` substrings

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message

ConfigureProxy node

Sets the proxy configuration used by all AWS clients.



Inputs

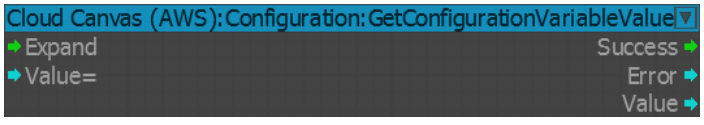
Port	Type	Description
Configure	Any	Sets the proxy configuration
Host	String	Proxy host
Port	Integer	Proxy port
UserName	String	Proxy user name
Password	String	Proxy password

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message

GetConfigurationVariableValue node

Inserts configuration value parameters into a string.



Inputs

Port	Type	Description
Expand	Any	Expands parameter references
Value	String	Value containing `$param-name$` substrings

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
Value	String	Value with `$param-name$` substring replaced by parameter values

SetDefaultRegion node

Sets (overrides) the region for all AWS clients in the current project.

Inputs

Port	Type	Description
Activate	Any	Sets the region for all AWS clients in the current project
Region	String	The region name to set as the default region for all AWS clients

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message

Choose **Apply** if you want to apply the configuration change to all AWS clients immediately. If **Apply** is set to **false**, you must add an ApplyConfiguration (p. 255) flow node to activate the changes.

Cognito (Player Identity) Nodes

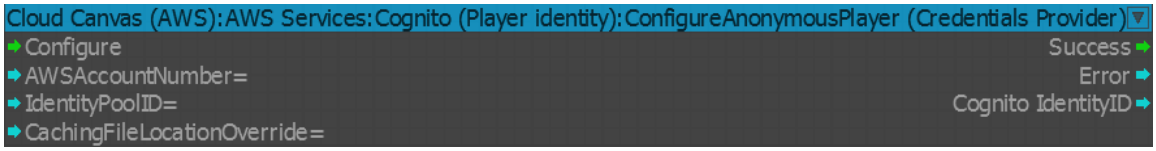
Use Amazon Cognito to configure player identity with these flow graph nodes.

Topics

- ConfigureAnonymousPlayer node (p. 258)
- ConfigureAuthenticatedPlayer node (p. 258)

ConfigureAnonymousPlayer node

Creates an anonymous identity on the device in your AWS account.



Inputs

Port	Type	Description
Configure	Any	Configure your game to use Amazon Cognito for anonymous players
AWSAccountNumber	String	Your AWS account number. This is needed to access Amazon Cognito.
IdentityPoolID	String	The unique ID of your Amazon Cognito identity pool. To create an identity pool ID, sign in to the AWS Management Console and use the Amazon Cognito console at https://console.aws.amazon.com/cognito/.
CachingFileLocationOverride	String	If specified, causes the Amazon Cognito ID to be cached to the path specified instead of to *<HOME_DIR>*/.aws/.identities.

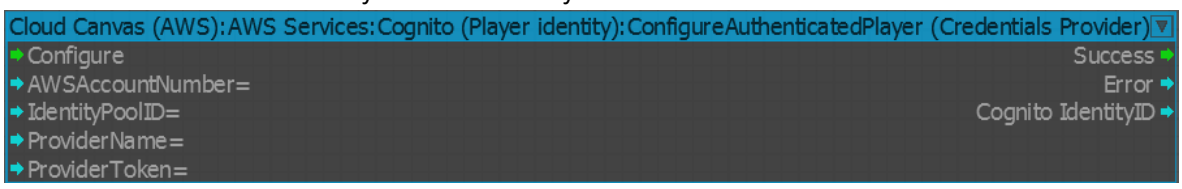
Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
CognitoIdentityID	String	The unique ID of the user

The first time the player runs the game and this node is triggered, an anonymous ID is generated for the player. This ID is persisted locally, and future runs of the game use the same identity.

ConfigureAuthenticatedPlayer node

Creates an authenticated identity on the device in your AWS account.



Inputs

Port	Type	Description
Configure	Any	Configure your game to use Amazon Cognito with the values specified.
AWSAccountNumber	String	Your AWS account number. This is needed for configuring Amazon Cognito.
IdentityPoolID	String	The unique ID of your Amazon Cognito identity pool. To edit your identity pool ID, open the AWS Management Console and choose **Cognito**.
ProviderName	String	Specifies the provider that authenticates the user
ProviderToken	String	Provider token with which to authenticate the user

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
CognitoIdentityID	String	The unique ID of the user

The first time the player runs the game and this node is triggered, an authenticated ID is generated for the player. The same ID is returned any time the user logs in with the same account, even on a second device.

DynamoDB (Database) Nodes

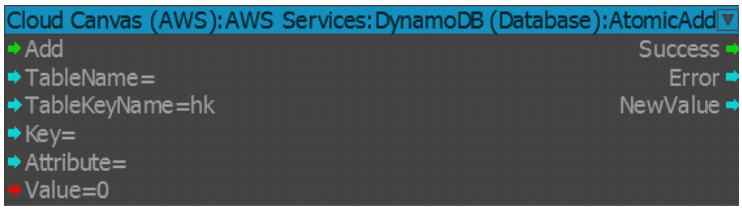
You can use these flow graph nodes to connect your game to Amazon DynamoDB.

Topics

- AtomicAdd node (p. 259)
- DeleteItem node (p. 260)
- GetItem node (p. 261)
- PutItem node (p. 262)
- Query node (p. 263)
- ScanTable node (p. 263)
- UpdateItem node (p. 264)
- GetStringSet node (p. 265)

AtomicAdd node

Add a number to an attribute in DynamoDB and return the number.



Inputs

Port	Type	Description
Add	Any	Writes the `val` specified in the **Value** port to DynamoDB
TableName	String	The name of the DynamoDB table to which to write
TableKeyName	String	The key name used in the table
Key	String	Specifies the key to which to write
Attribute	String	Specfies the attribute to which to write
Value	Integer	Specifes the value to write

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
NewValue	String	The value of the attribute after the addition.

This is an atomic operation. You do not need to create the attribute before you use it.

DeleteItem node

Deletes a record in DynamoDB.

Inputs

Port	Type	Description
DeleteItem	Any	Deletes the specified item from DynamoDB.
TableName	String	The name of the DynamoDB table from which to delete
TableKeyName	String	The key name used in the table

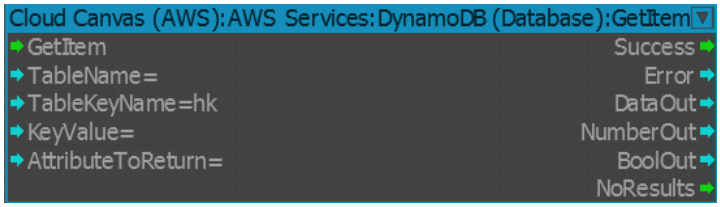
Port	Type	Description
KeyValue	String	Specifies the key to delete

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
DeletedItems	Any	Activated when matches were found to delete
NoResults	Any	No matching results were found

GetItem node

Gets values from DynamoDB.



Inputs

Port	Type	Description
GetItem	Any	Retrieves the item specified from DynamoDB
TableName	String	The name of the DynamoDB table from which to read
TableKeyName	String	The key name used in the table
KeyValue	String	Specifies the key to read
AttributeToReturn	String	Specfies the attribute to read

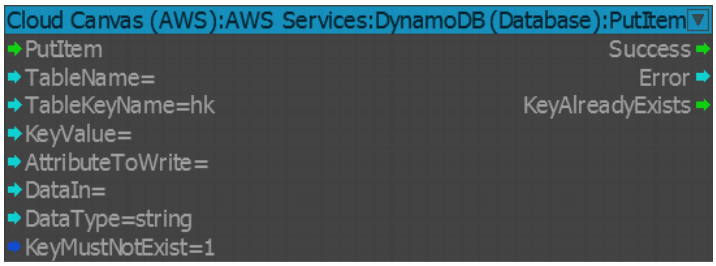
Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
DataOut	String	String data that was read from DynamoDB
NumberOut	String	Number data that was read from DynamoDB
BoolOut	String	Boolean value that was read from DynamoDB

Port	Type	Description
NoResults	Any	No matching results were found for the table, key, and attribute specified

PutItem node

Writes values to DynamoDB.



Inputs

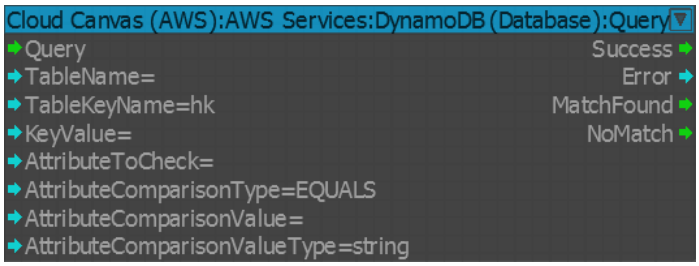
Port	Type	Description
PutItem	Any	Writes the item specified to DynamoDB
TableName	String	The name of the DynamoDB table to which to write
TableKeyName	String	The key name used in the table
KeyValue	String	Specifies the key to write
AttributeToWrite	String	Specfies the attribute to write
DataIn	String	The data to write
DataType	String	The data type that the data will be written as
KeyMustNotExist	Boolean	When true, specifies that the key must not already exist; the default is true. Setting this to false allows you to overwrite an existing key in the table, including all of its existing attributes, and replace them with the new key and attribute values.

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
KeyAlreadyExists	Any	The key already exists; no change was made

Query node

Queries values in DynamoDB.



Inputs

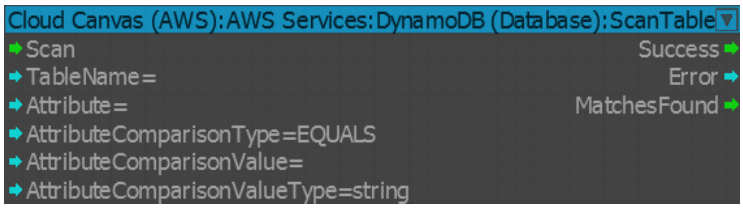
Port	Type	Description
Query	Any	Queries table data in DynamoDB
TableName	String	The name of the DynamoDB table to query
TableKeyName	String	The name of the table key to query
KeyValue	String	The value of the key to query
AttributeToCheck	String	The attribute to query
AttributeComparisonType	String	The comparison type to make against the attribute; the default is `EQUALS`. Other possible values are `GREATER_THAN`, `GREATER_THAN_OR_EQUALS`, `LESS_THAN`, `LESS_THAN_OR_EQUALS`.
AttributeComparisonValue	String	The value to compare against the attribute
AttributeComparisonValueType	String	The data type of `AttributeComparisonValue` (`string`, `bool`, or `number`); the default is `string`

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
MatchFound	Any	A match was found
NoMatch	Any	No match was found

ScanTable node

Scans for entries which pass a comparison test in DynamoDB.



Inputs

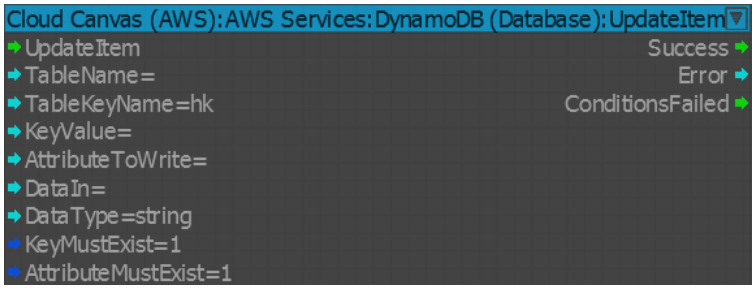
Port	Type	Description
Scan	Any	Scans for matches in DynamoDB table data using the specified attributes
TableName	String	The name of the DynamoDB table to scan
Attribute	String	The attribute to query for
AttributeComparisonType	String	The comparison type to make against the attribute; this defaults to `EQUALS`.
AttributeComparisonValue	String	The value to compare against the attribute
AttributeComparisonValueType	String	The data type of `AttributeComparisonValue` (`string`, `bool`, or `number`); the default is `string`

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
MatchesFound	Any	The number of matches found on a successful scan

UpdateItem node

Updates attribute values of an existing item in DynamoDB.



Inputs

Port	Type	Description
UpdateItem	Any	Updates an item in DynamoDB

Port	Type	Description
TableName	String	The name of the DynamoDB table to use
TableKeyName	String	The name of the key in the table
KeyValue	String	The value of the key to write
AttributeToWrite	String	The attribute to write to
DataIn	String	The data to write
DataType	String	The data type to write the data as
KeyMustExist	Boolean	True if the key specified must already exist in the table; the default is true.
AttributeMustExist	Boolean	True if the attribute must exist for the key specified; the default is true

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
ConditionFailed	Any	Key or attribute not found (either the `KeyMustExist` or `AttributeMustExist` condition failed)

GetStringSet node

Retrieves the members of a string set.

Inputs

Port	Type	Description
GetItem	Any	Reads data from DynamoDB
TableName	String	The name of the DynamoDB table to use
TableKeyName	String	The name of the key in the table
KeyValue	String	The value of the key to write
AttributeToWrite	String	The attribute to write to

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
DataOut	String	The data read from DynamoDB. The **DataOut** port fires once for each member of the set.

The success port indicates that all members of the set have been output.

Lambda (Cloud Functions) Node

You can use this flow graph node to invoke AWS Lambda functions.

Invoke node

Inputs

Port	Type	Description
Invoke	Any	Invokes a Lambda function, optionally providing JSON data as arguments through the **Args** port. For more information, see AWS Lambda Invoke Request Syntax.
FunctionName	String	The name of the Lambda function to call
Args	String	The input data that will be sent to the Lambda function call as arguments in JSON format. For more information, see AWS Lambda Invoke Request Syntax.

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
Result	String	The data that was output by the Lambda function if no error occured

S3 (Storage) Nodes

You can use these flow graph nodes to download and upload files from the Amazon Simple Storage Service (Amazon S3), and to generate a public URL that points to a specific location in Amazon S3.

Topics

- DownloadFile node (p. 267)

- UploadFile node (p. 267)
- GeneratePublicUrl node (p. 268)

DownloadFile node

Downloads a file from Amazon S3.

Inputs

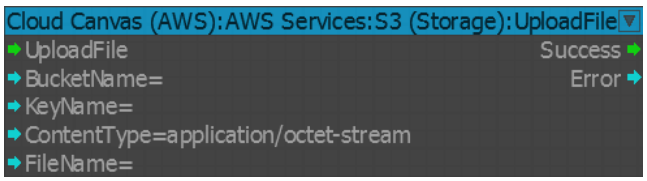
Port	Type	Description
DownloadFile	Any	Reads file data from an Amazon S3 bucket
BucketName	String	The name of the Amazon S3 bucket to use
KeyName	String	The name of the file to download from Amazon S3
FileName	String	The filename to use for the downloaded object

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message

UploadFile node

Uploads a file to Amazon S3.



Inputs

Port	Type	Description
UploadFile	Any	Uploads a file to an Amazon S3 bucket
BucketName	String	The name of the Amazon S3 bucket to use
KeyName	String	What to name the uploaded object on Amazon S3. If this value is not updated on subsequent uses, the existing Amazon S3 object is overwritten.

Port	Type	Description
ContentType	String	The mime-content type to use for the uploaded object (for example, `text/html`, `video/mpeg`, `video/avi`, or `application/zip`). The type is stored in the Amazon S3 record. You can use this type to help identify or retrieve a specific type of data later. The default is `application/octet-stream`.
FileName	String	The name of the file to upload

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message

GeneratePublicUrl node

Generates a presigned URL that points to an Amazon S3 location that you specify.

Inputs

Port	Type	Description
PresignUrl	Any	Generates a presigned URL for the Amazon S3 location specified
BucketName	String	The name of the Amazon S3 bucket to use
KeyName	String	What to name the uploaded object on Amazon S3. If this value is not updated on subsequent uses, the existing Amazon S3 object is overwritten.
Http Request Method	String	The HTTP method against which to presign (`DELETE`, `GET`, `POST`, or `PUT`)

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
Url	String	The signed URL

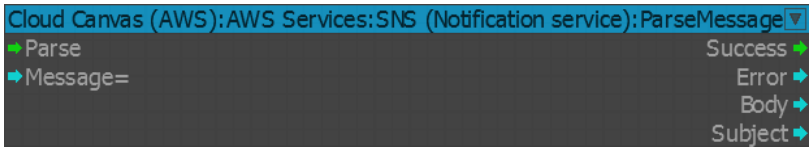
SNS (Notification Service) Nodes

You can use these flow graph nodes to process Amazon Simple Notification Service (Amazon SNS) messages.

Topics

- ParseMessage node (p. 269)
- Notify node (p. 269)
- CheckArnSubscribed node (p. 270)
- SubscribeToTopic node (p. 270)

ParseMessage node



Inputs

Port	Type	Description
Parse	Any	Extract the subject and body text in JSON format from an Amazon SNS message
Message	String	The JSON message to deserialize.

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
Body	String	The message body
Subject	String	The message subject

Notify node

Publishes messages to an Amazon SNS topic.

Inputs

Port	Type	Description
Notify	Any	Sends a notification to an Amazon SNS topic

Port	Type	Description
Message	String	The message to send
Subject	String	The subject of the message
TopicARN	String	The Amazon Resource Name for your Amazon SNS topic

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message

CheckArnSubscribed node

Checks if an ARN is subscribed to an Amazon SNS topic.

Inputs

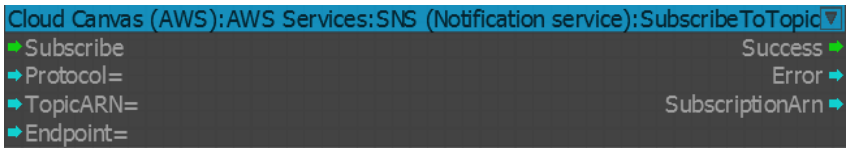
Port	Type	Description
Check	Any	Checks if an ARN is subscribed to an Amazon SNS topic
TopicARN	String	The Amazon SNS topic ARN to check
Endpoint	String	The endpoint to check for subscription to the specified topic. The endpoint can be an email address, an Amazon SQS queue, or any other endpoint type supported by Amazon SNS.

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
True	Any	The ARN is subscribed to the Amazon SNS topic
False	Any	The ARN is not subscribed to the Amazon SNS topic

SubscribeToTopic node

Subscribes to an Amazon SNS topic.



Inputs

Port	Type	Description
Subscribe	Any	Subscribes to a topic to receive messages published to that topic. For more information, see Subscribe to a Topic.
Protocol	String	The protocol of the endpoint to which to subscribe
TopicARN	String	The ARN of the Amazon SNS topic to which to subscribe
Endpoint	String	The address of the endpoint to subscribe (for example, an email address). For information on sending to HTTP or HTTPS, see Sending Amazon SNS Messages to HTTP/HTTPS Endpoints.

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
SubscriptionArn	String	The ARN of the created subscription

For more information on Amazon SNS, see the Amazon Simple Notification Service Developer Guide.

SQS (Message Queuing Service) Nodes

You can use these flow graph nodes to start polling AWS queues and to push messages to AWS queues.

Topics

- PollAndNotify node (p. 271)
- Push node (p. 272)

PollAndNotify node

Inputs

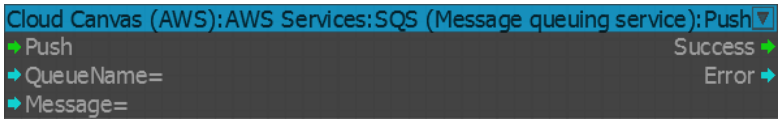
Port	Type	Description
Start	Any	Start polling an AWS queue
QueueName	String	The name of an AWS queue that has already been created

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
OnMessageReceived	String	The most recent message on the stack
QueueArn	String	The ARN (Amazon Resource Name) of the queue

Push node

Pushes a message to an AWS queue



Inputs

Port	Type	Description
Push	Any	Pushes a message to an AWS queue
QueueName	String	The name of an AWS queue that has already been created
Message	String	The message to send

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message

Static Data (PROTOTYPE) Nodes

Static Data is a Lumberyard system for managing game data that changes less frequently through monitored Amazon S3 buckets. You can use these flow graph nodes to update or query your buckets and/or monitor them at regular intervals for changes.

Topics

- Add Monitored Bucket node (p. 273)
- Get Static Data node (p. 273)
- Load Static Data node (p. 274)
- Remove Monitored Bucket node (p. 275)
- Request Bucket node (p. 275)
- Set Update Frequency node (p. 276)

Add Monitored Bucket node

Adds an Amazon S3 bucket to monitor.

Inputs

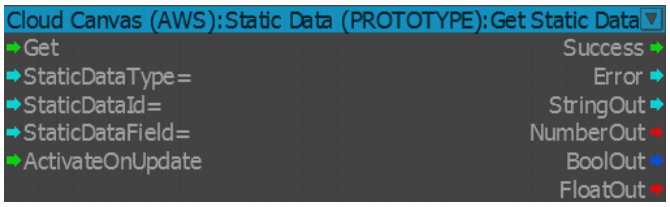
Port	Type	Description
AddBucket	Void	Adds a bucket to watch for updates
BucketName	String	The name of the Amazon S3 bucket to watch

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
Finished	String	The bucket was added

Get Static Data node

Retrieves a field from a static data definition.



Inputs

Port	Type	Description
Get	Void	Retrieves a value from static data
StaticDataType	String	The type of the static data to retrieve

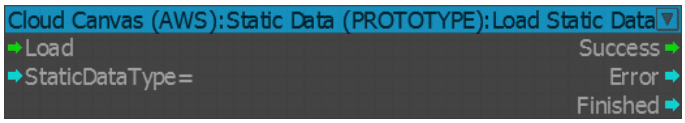
Port	Type	Description
StaticDataId	String	The identifier for the static data definition in the table
StaticDataField	String	The field name of the data to retrieve
ActivateOnUpdate	Void	Fire the node again the next time an update of the data takes place

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
StringOut	String	The output of a string field
NumberOut	Integer	The output of a numeric field
BoolOut	Boolean	The output of a Boolean
FloatOut	Integer	The output of a floating point numeric field

Load Static Data node

Attempts to load static data of the type specified.



Inputs

Port	Type	Description
Load	Any	Load a type of static data
StaticDataType	String	The type of static data to load

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
Finished	String	Finished attempting to load

Remove Monitored Bucket node

Removes a bucket name from the list of monitored buckets.

Inputs

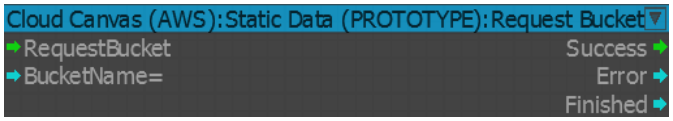
Port	Type	Description
Remove	Any	Removes a bucket from the list of monitored buckets
BucketName	String	The name of the bucket to remove

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
Finished	String	Finished removing the bucket

Request Bucket node

Requests an update of a specific bucket, or of all monitored buckets.



Inputs

Port	Type	Description
RequestBucket	Any	Requests an update of a specific bucket or of all monitored buckets
BucketName	String	The name of the bucket for which to request an update. To request updates for all buckets, leave this value blank.

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message

Port	Type	Description
Finished	String	Finished sending the request

Set Update Frequency node

Sets or clears a recurring timer to poll monitored buckets.

Inputs

Port	Type	Description
SetTimer	Void	Sets a recurring timer to the value specified in `TimerValue`
TimerValue	Integer	The time interval at which to poll. Possible values are from 0 to 100. A value of 0 clears the timer; 0 is the default.

Outputs

Port	Type	Description
Success	Any	Activated upon a successful operation
Error	String	Activated upon an error being detected; the value of the port is the error message
Set	String	The timer has been set

Cloud Canvas Software Engineering Guide

This section provides in-depth information for Cloud Canvas Resource Manager and the cloud gem framework.

Topics

- Resource Manager in Depth (p. 276)
- Cloud Gem Framework (p. 305)

Resource Manager in Depth

The effective use of Cloud Canvas Resource Manager is key to the success of any cloud-connected game project. This section provides in-depth information for software engineers.

For information on permissions in Resource Manager, see Understanding the Resource Manager Security System (p. 393).

Topics

- Understanding Resource Status Descriptions (p. 277)
- Editing Resource Manager Files (p. 277)
- Working with JSON Files (p. 278)
- Viewing the Cloud Canvas Progress Log (p. 279)
- Working with Resource Groups (p. 280)
- Resource Definitions (p. 287)
- Permissions Metadata for Resource Definitions (p. 290)
- Importing Resource Definitions into Cloud Canvas (p. 291)
- Resource Deployments (p. 296)
- Resource Mappings (p. 298)
- Resource Manager Resource Group Parameters (p. 299)
- Custom Resources (p. 300)

Understanding Resource Status Descriptions

The status of AWS resources is displayed in the **Cloud Canvas Resource Manager** in places such as the progress log. The following list provides descriptions of common resource status codes. To see the reason for the current status, you can pause your mouse on the status text in the resource manager.

Create in progress – The resource is in the process of being created in AWS.

Create complete – The resource has been successfully created in AWS.

Create failed – The resource could not be created in AWS.

Update in progress – The resource is in the process of being updated in AWS.

Update complete – The resource was successfully updated in AWS.

Update failed – The resource could not be updated in AWS.

Delete in progress – The resource is in the process of being deleted in AWS.

Delete complete – The resource has been deleted in AWS.

Rollback in progress – An operation has failed and AWS CloudFormation is attempting to restore the resource to its previous state.

Rollback failed – A rollback has failed. The AWS resources in a CloudFormation stack that have this status are in an inconsistent state. You may have to delete and recreate the stack.

Editing Resource Manager Files

The navigation pane in the **Cloud Canvas Resource Manager** dialog contains a number of nodes that represent text files that are stored on disk. The resource-template.json (p. 279) node is one example.

The child nodes of template files each represent one section of the parent node template file. These child nodes can help you locate and edit the resource definition sections of the parent node template file.

Using the Internal Editor

When you select a text file node in the navigation pane, the file content and text editing options are shown in the detail pane of **Cloud Canvas Resource Manager**. You can use the detail pane to view and

edit the contents of the file. Use the **Edit**, **Search** menu item to search for text, and the **Previous** and **Next** buttons to navigate from one match to the next. After you have modified a file, you can save it by clicking **Save** in the toolbar or by choosing **File**, **Save**.

> **Note**
> The changes that you make in the template file child nodes are always saved to the parent node template file.

Using an External Editor

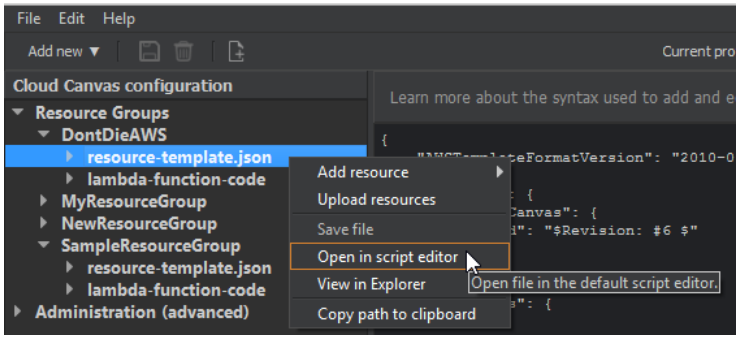
You can use an external script editor instead of the **Cloud Canvas Resource Manager** to edit files. You can specify which editor to use in Lumberyard Editor.

To specify an external script editor

- In Lumberyard Editor, click **File**, **Global Preferences**, **Editor Settings**, **General Settings**, **Files**, **External Editors**, **Scripts Editor**.

To open a file in an external script editor

- Right-click the file in the navigation pane and choose **Open in script editor**:



To copy the path of the template file to the clipboard, right-click the file in the navigation pane and choose **Copy path to clipboard**.

Notes

Note the following:

- Opening a child node of a template file in a script editor opens the full (parent) file for editing.
- If your project files are under source control, Lumberyard prompts you to check out files before they can be edited. The source control icon on the toolbar dynamically displays the status of a selected file in source control.
- If the contents of the file change on disk while there are unsaved changes in the editor, Lumberyard prompts you to load the updated contents from disk and replace the modified contents in the editor.

Working with JSON Files

Some of the nodes in the **Cloud Canvas Resource Manager** navigation pane represent JSON template or settings files for your project. The content of these files is described in detail in Resource Definitions. When you select one of these nodes in the navigation pane, the detail pane shows the contents of that file. You can edit the file directly in the resource manager or use an external editor. For more information, see Editing Resource Manager Files (p. 277).

In the navigation pane, some template file nodes have child nodes. Each of the child nodes represents one section of its parent node template file. These child nodes make it easier to find and edit the corresponding sections of the parent node template file. Any changes that you make in a child node are always saved in the corresponding section of the parent template file.

The following template is found in each resource group under the **Resource Groups** node:

resource-template.json

Each resource group has a **resource-template.json** node and a **lambda-function-code** child node. The `resource-template.json` file defines the group's resources. For more information, see Resource Definitions. In the navigation pane, each of the nodes under **resource-template.json** represents one of the resources defined in a section of the `resource-template.json` file.

The following templates are found under the **Administration (advanced)** node:

project-settings.json

The `project-settings.json` file contains project configuration data. For more information, see Resource Definitions.

project-template.json

The `project-template.json` file defines the resources used by **Cloud Canvas Resource Manager**. For more information, see Resource Definitions.

deployment-template.json

The `deployment-template.json` file defines the AWS CloudFormation stack resources for each of the projects resource groups. For more information, see Resource Definitions.

deployment-access-template.json

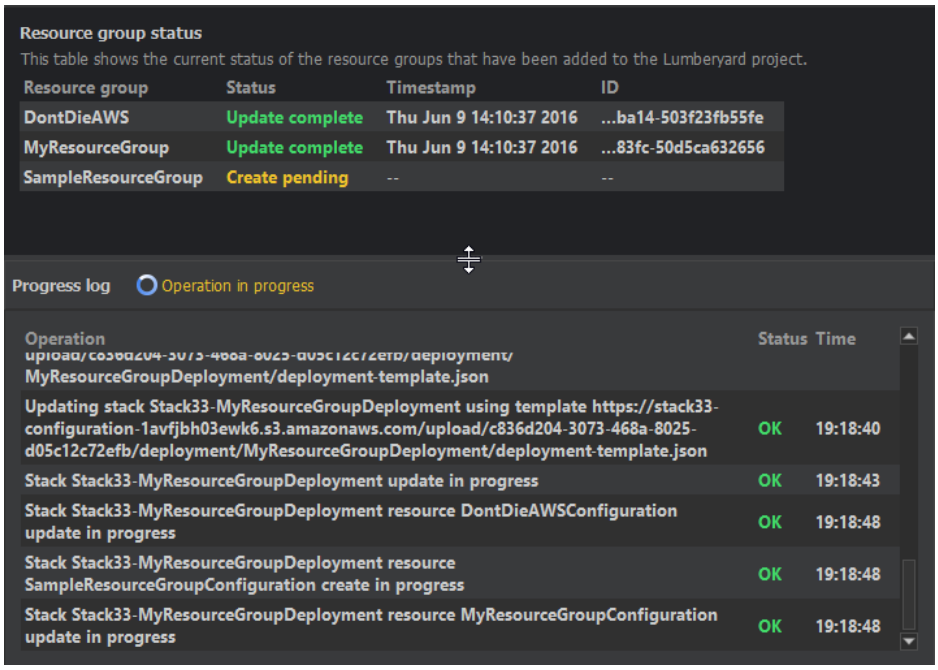
The `deployment-access-template.json` file defines the AWS CloudFormation stack resources that control access to each deployment's resources. For more information, see Resource Definitions and Controlling Access to Resources (p. 393).

user-settings.json

The `user-settings.json` file contains user specific settings. For more information, see Resource Definitions.

Viewing the Cloud Canvas Progress Log

The **Cloud Canvas Resource Manager Progress log** shows the progress of AWS CloudFormation stack updates. During the update, the progress log expands from the bottom of the detail pane to display the progress of the update. You can adjust the amount of space the log uses by dragging the divider line between the panes.



To hide the progress log, drag the divider downward.

Working with Resource Groups

Resource groups define the AWS resources that cloud-connected game features such as high score tables require. You define the resource groups locally and then upload them to AWS, where the features you specify are created in the cloud.

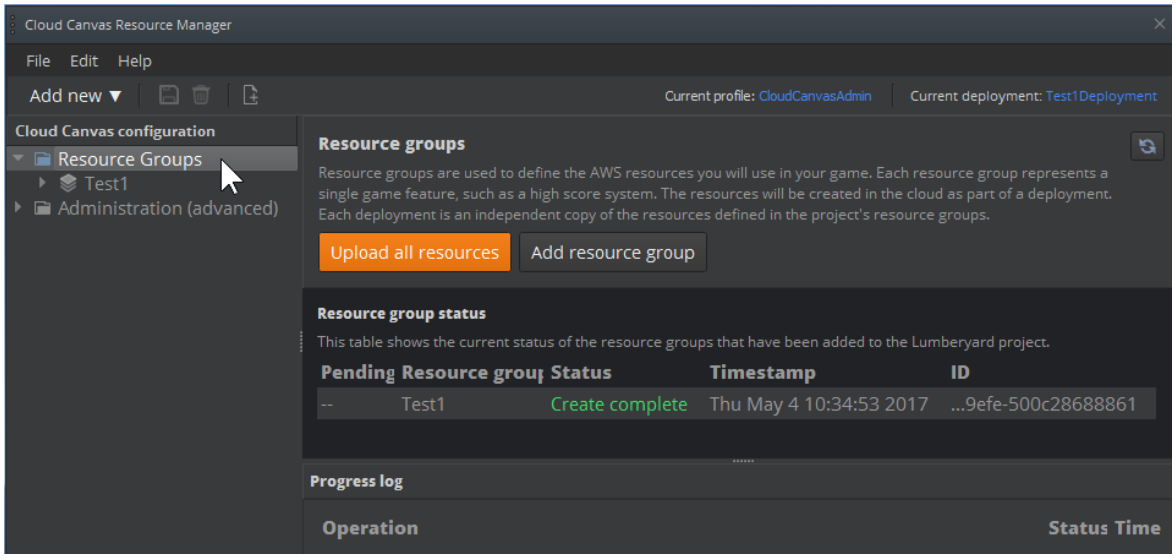
You can use the **Cloud Canvas Resource Manager** to manage resource groups collectively or individually. You can create a resource group locally, specify the AWS resources that you want the resource group to have, and then upload the resource group definition to have it created in AWS. This document shows you how to use **Cloud Canvas Resource Manager** to perform these and other resource group-related tasks.

Managing Resource Groups

To see the status of the resource groups that belong to your project's current deployment, click **Resource Groups** in the **Cloud Canvas configuration** navigation pane . Note the following:

- If you select **Resource Groups** and no AWS profile is configured, Lumberyard prompts you to provide one. A profile is required for Lumberyard to display the status of your project's resources. For more information, see Managing Cloud Canvas Profiles (p. 198).
- If you select **Resource Groups** and deployments exist but no deployment is active, Lumberyard prompts you to select one. For more information, see Making a Cloud Canvas Deployment Active (p. 388).

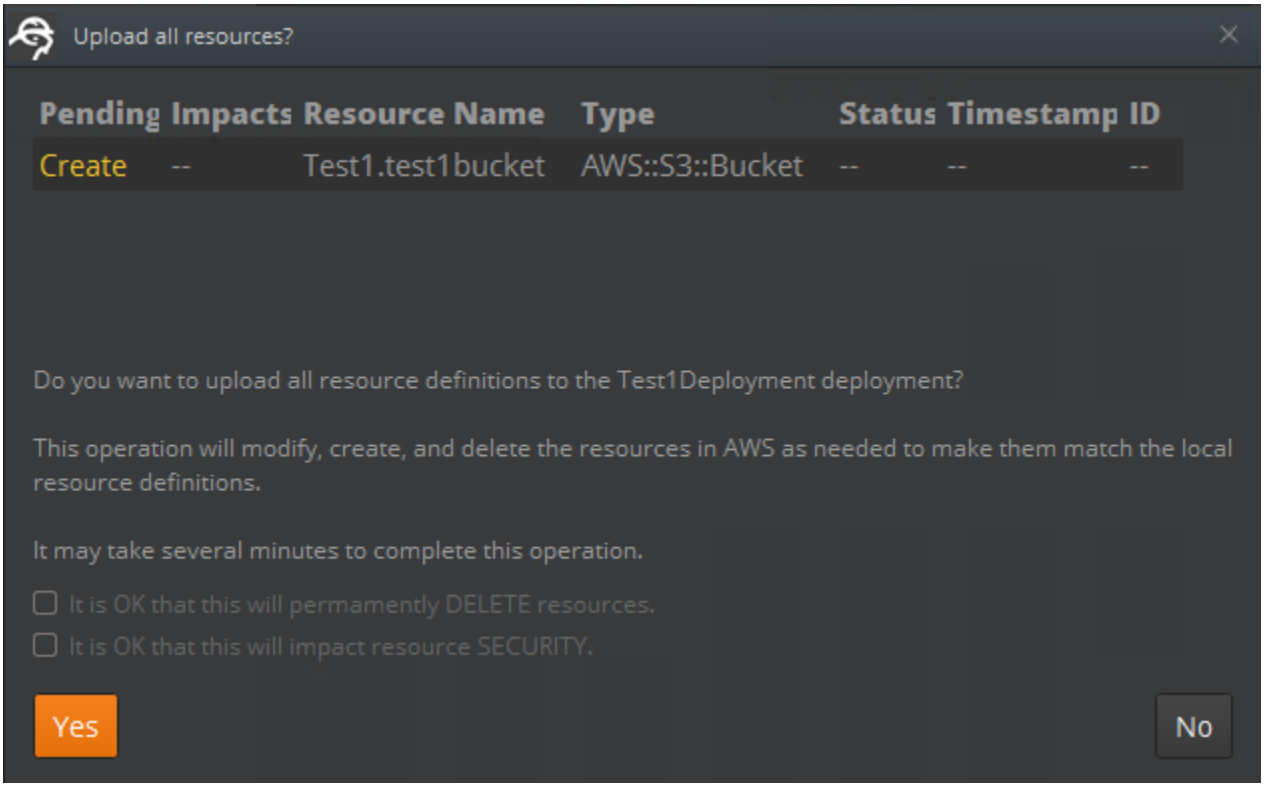
The **Resource Groups** detail pane lists the resource groups in your current deployment:



The **Resource Groups** detail pane has the following options:

Upload all resources

Starts the process of modifying your resources in AWS as needed to match all the definitions in all of your local resource groups. As the update proceeds, resource groups with the **Pending** status of **Create** change to the **Status** of **Create complete**. The update might take a few minutes.

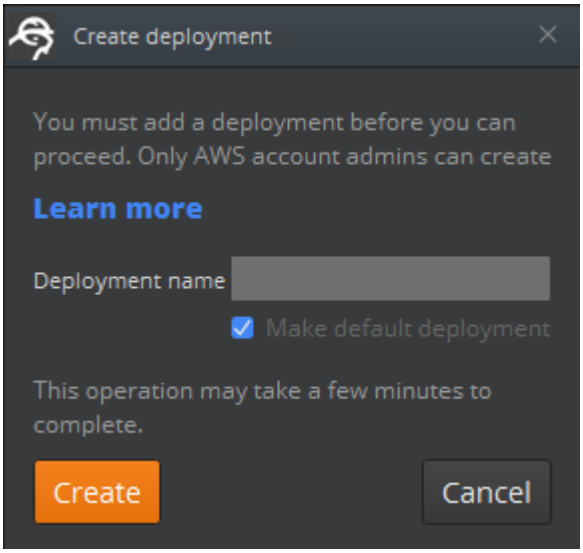


Note the following:

- If you have not yet initialized your Lumberyard project with an AWS account for the resources that you want to upload, Lumberyard prompts you to do so. To prepare your Lumberyard project

for use with AWS, you must be an administrator of the AWS account that you use. For more information, see Initializing **Cloud Canvas Resource Manager** (p. 382).

- After you have initialized the project, Lumberyard prompts you to create a deployment for it. A deployment creates all the AWS resources that are specified in your resource group definition.

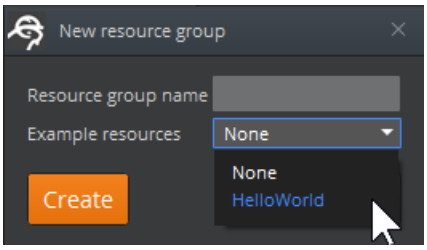


For more information, see Create Deployment (p. 385).

Add resource group

Adds a new resource group definition to your Lumberyard project. A resource group definition represents a single game feature like a high score system. The definition specifies the AWS resources that the feature will use.

Clicking **Add resource group** opens the **New resource group** dialog:



Provide the following information:

- **Resource group name** – The name of the resource group. The name must be alphanumeric. Lumberyard uses this name to create an AWS CloudFormation stack resource definition in the deployment-template.json file.
- **Example resources** – (Optional) Choose to include example resources in the resource group. You can study the examples to see how resources are defined in a resource group, or modify the examples to turn them into a feature for your project.

Resource group status

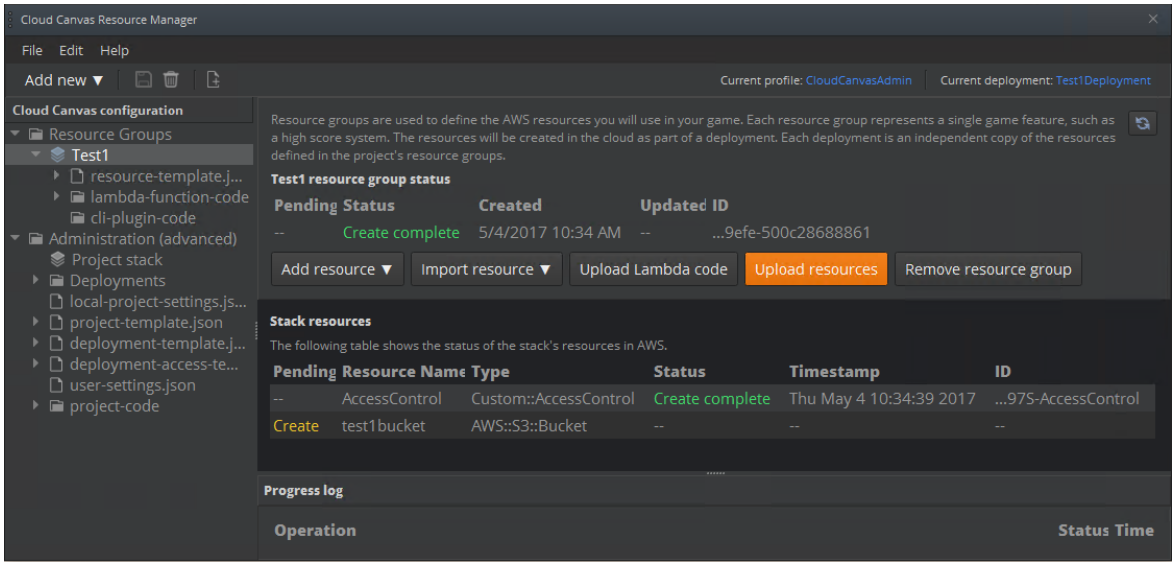
Shows the status of the AWS CloudFormation stack of each resource group in the active deployment. **Resource group** shows the resource group name. For descriptions of the remaining fields in this table, see Stack Resources Table (p. 384) in the Working with Project Stacks (p. 383) section.

Progress log

Shows the progress of AWS CloudFormation stack updates. For more information, see Viewing the Cloud Canvas Progress Log (p. 279).

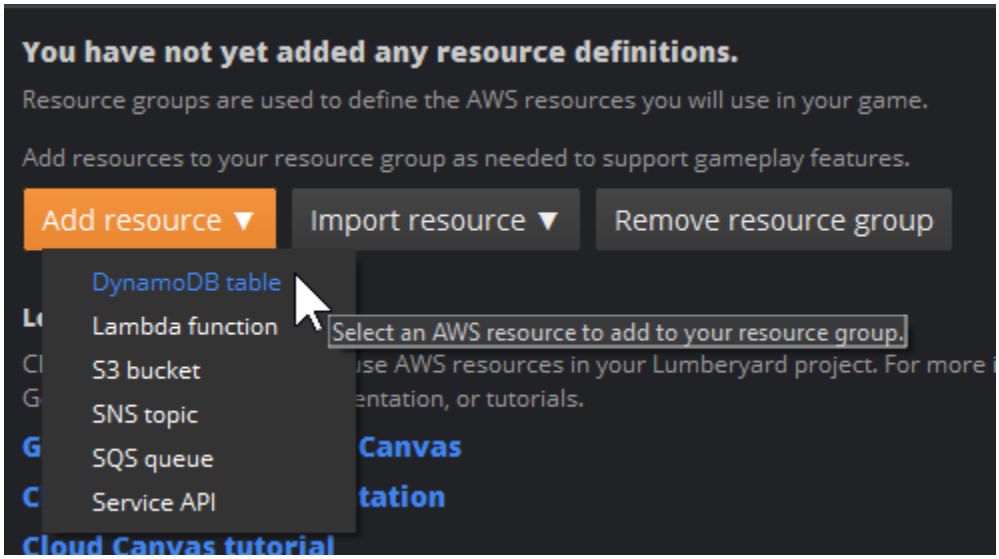
Managing Individual Resource Groups

Each child node of **Resource Groups** represents a resource group in your Lumberyard project. When you select one of these resource groups, the detail pane shows the status of the resource group.



Adding Resources to a New Resource Group

When you create a resource group, the group does not yet have any AWS resource definitions. Use the **Add resource** option to add one:

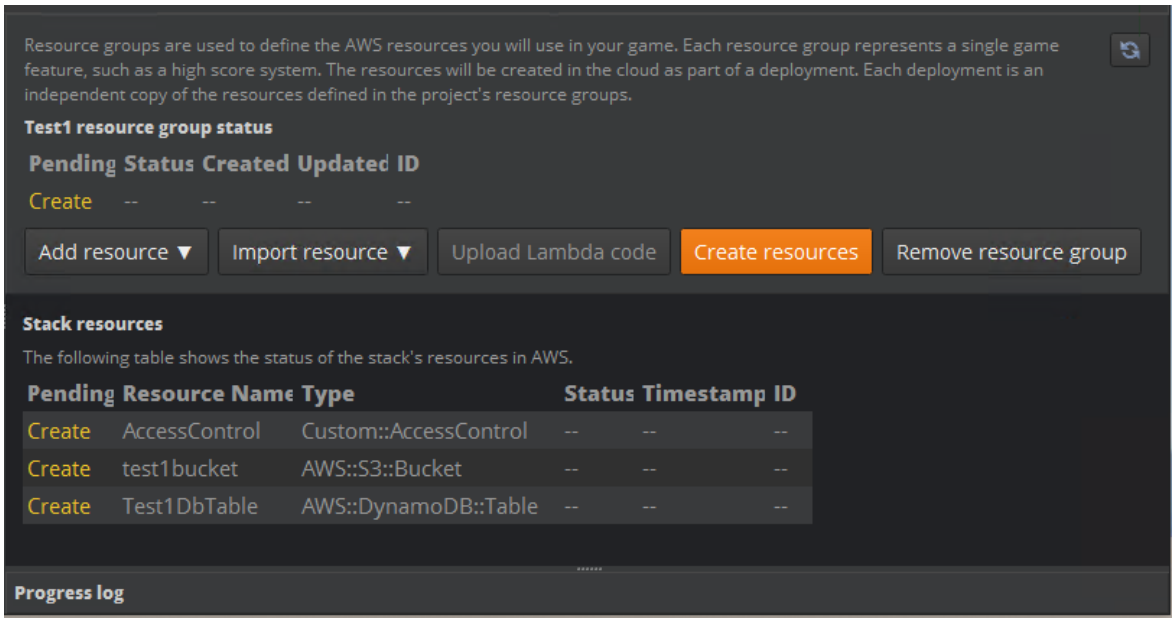


Importing Resources

You can also click **Import resource** to import an AWS resource that already exists in your AWS account. For more information, see Importing Resource Definitions into Cloud Canvas (p. 291).

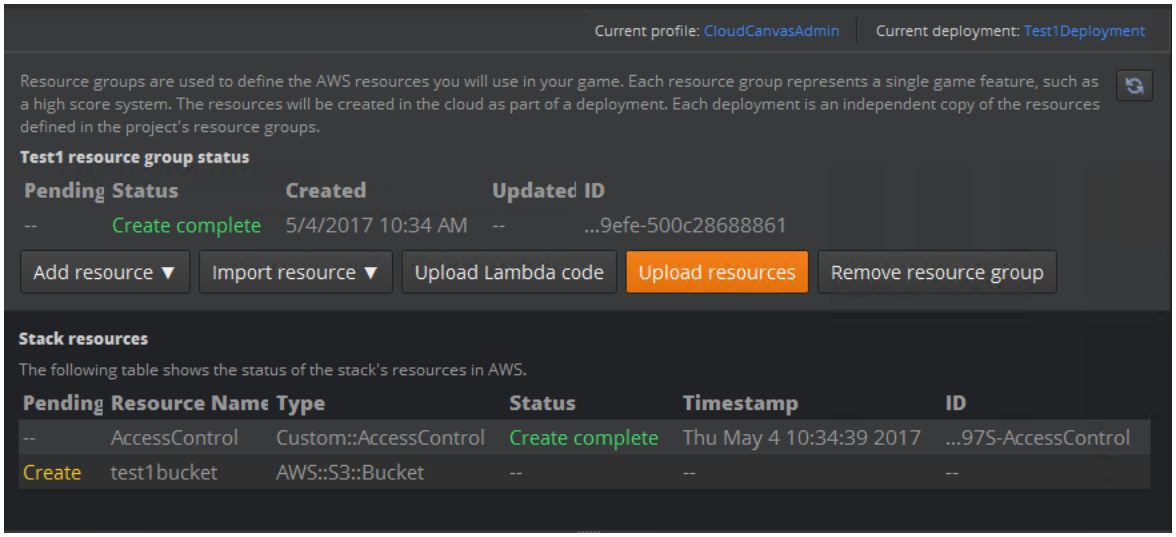
Creating Resources in AWS

The resource definitions that you add are created locally and only describe the AWS resources that you want to use. The resources themselves are not created in AWS until you click **Create resources**:



Individual Resource Group Status

You can use a resource group's status pane to manage the resource group.



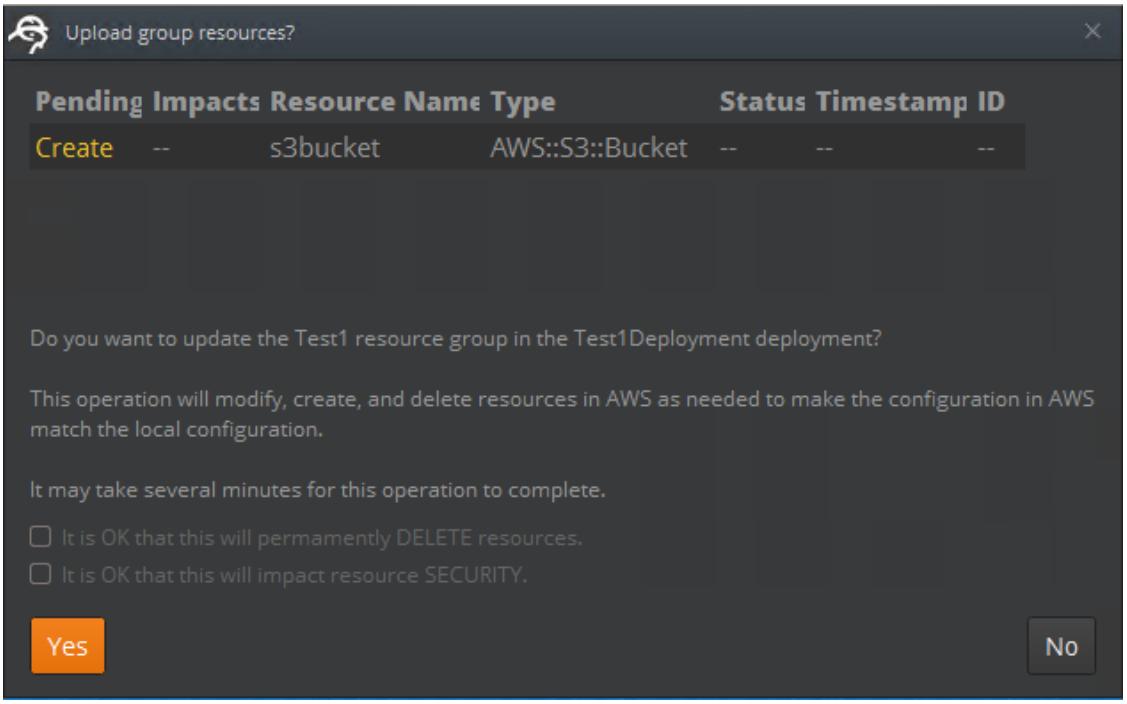
In addition to **Add resource** and **Import resource**, the status pane for a resource group has the following options:

Upload Lambda code

Use this option to upload Lambda function code without updating the entire project stack.

Upload resources

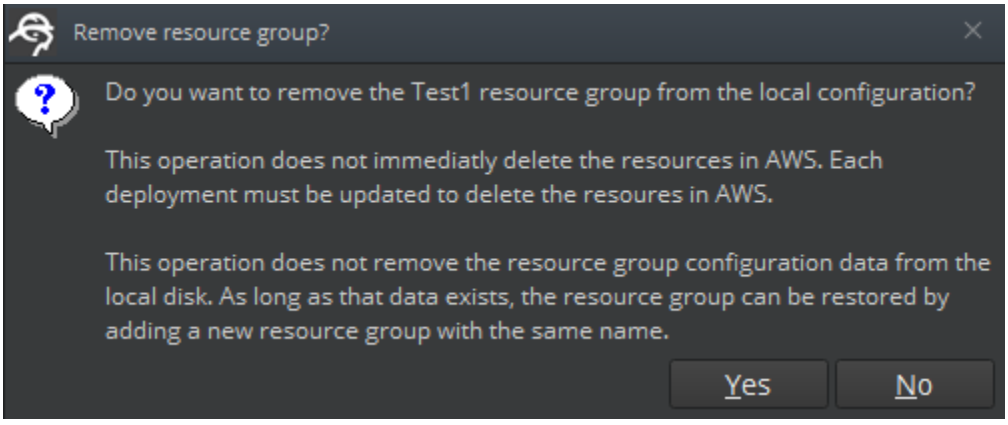
After you have created one or more resource definitions, you click **Upload resources** to start the process of creating the resources in AWS. Resource creation follows the local resource definitions that you created with the **Add resource** option.



As the update proceeds, the resources with the **Pending** status of **Create** change to the **Status** of **Create complete**.

Remove resource group

Click **Remove resource group** to remove the selected resource group from your local configuration.

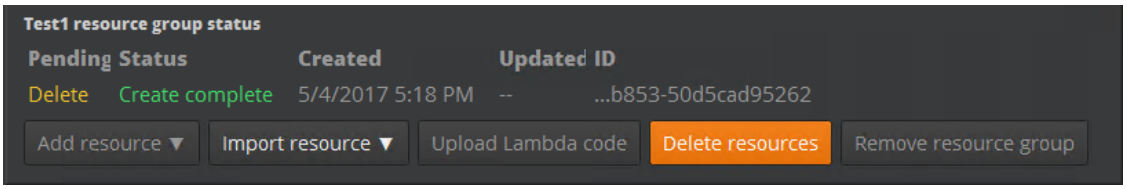


To delete the actual resources from AWS, use the **Delete resources** option as described in the section that follows.

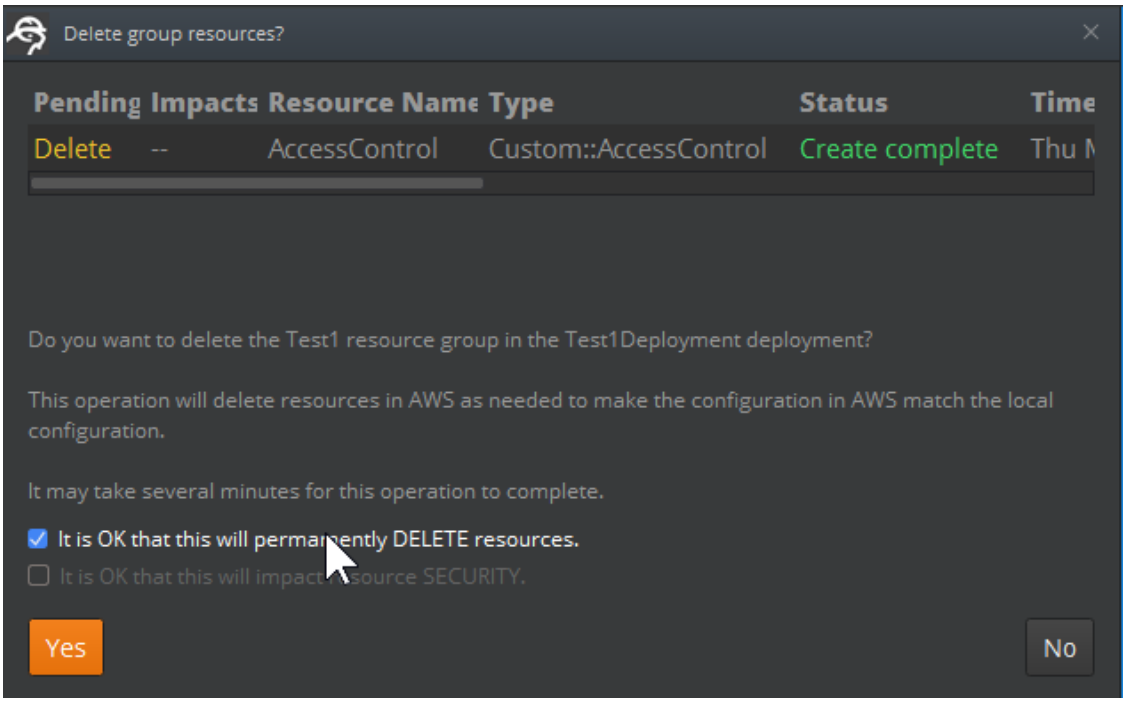
> **Note**
> The remove resource operation does not remove the resource group's configuration data from the local disk. As long as that data exists on disk, you can restore the resource group by adding a new resource group with the same name.

Delete resources

The **Delete resources** option appears after you have removed a resource group from your local configuration (for example, by clicking **Remove resource group**) when the resources defined by the resource group still exist in AWS.



When you click **Delete resources**, Lumberyard prompts you to confirm the deletion of resources in AWS for the deployment that is currently active in Lumberyard Editor. You must select the **It is OK that this will permanently DELETE resources** check box before you can click **Yes**.



After you click **Yes**, the deletion operation might take several minutes to complete.

Stack resources

The **Stack resources** table shows the status of each of the AWS resources that you defined for the resource group. For descriptions of the fields in this table, see Stack Resources Table (p. 384) in the Working with Project Stacks (p. 383) section.

resource-template.json

For information about this node, see Working with JSON Files (p. 278).

lambda-function-code

The **lambda-function-code** node and its child nodes correspond to the `lambda-function-code` directory in your project. The `lambda-function-code` directory contains the code that implements the AWS Lambda function resources that are defined by your resource group. For more information, see lambda-function-code Directory Also see related information for the project-code (p. 287) node.

project-code

This node is located at the bottom of the **Administration (advanced)** section in the resource manager navigation tree. The `project-code` directory contains the code that implements the AWS Lambda function resources that **Cloud Canvas Resource Manager** uses. For more information, see Resource Definitions. The **project-code** node contains file and directory child nodes. Click a file node to see or edit its contents in the detail pane. For more information, see Editing Resource Manager Files (p. 277).

Resource Definitions

Resource definitions are specifications in the form of AWS CloudFormation template files that determine the resources (for example, DynamoDB databases, Lambda functions, and access control information) that will be created in AWS for the game. Game code and flow graphs use AWS resources and expect those resources to exist and to be configured in a specific way. The resource definitions determine this architecture and configuration.

Resource Definition Location

A description of the resources required by the game is stored in files under the `dev\{game}\AWS` directory, where `dev` is the Lumberyard installation `\dev` subdirectory and *{game}* is the directory identified by the `sys_game_folder` property in the `\dev\bootstrap.cfg` file. For example, if your game is the Samples Project, your resource definition path might be `C:\lumberyard\dev\SamplesProject\AWS`. These files should be checked into the project's source control system along with your other game code and data.

The default *{game}*`\AWS` directory contents are created by the `lmbr_aws` project create (p. 419) command.

In addition, some user-specific configuration data is kept in the `dev\Cache\{game}\{platform}\user\AWS` directory. The contents of this directory should not be checked into the project's source control system.

The following shows the contents of these `AWS` directories.

```
dev\{game}\AWS\
    project-code\
        (Lambda function Code)
    resource-group\
        {resource-group-name}\
            lambda-function-code\
                (Lambda function Code)
            resource-template.json
    deployment-access-template.json
    deployment-template.json
    local-project-settings.json
    project-template.json

dev\Cache\{game}\{platform}\user\AWS\
    user-settings.json
```

Each of these `.json` files is described in the following sections.

The project-code Directory

The `project-code` subdirectory contains the source code for the AWS CloudFormation Custom Resource handler that is used in the project's AWS CloudFormation templates. For information about custom resources, see Custom Resources (p. 300).

It also contains the code that implements the token exchange step of the player login process. For more information, see Controlling Access to Resources (p. 393).

resource-group*{resource-group}* Directory

The AWS resources used by the game are organized into separate resource groups. The `resource-group` directory contains these in individual *`{resource-group}`* subdirectories. Each *`{resource-group}`* subdirectory is typically named after your game project and contains a `lambda-function-code` directory and a `resource-template.json` file.

The lambda-function-code Directory

The `lambda-function-code` subdirectory is present when a resource template defines Lambda function resources. This directory is where you put the source files that implement those functions.

Lumberyard provided tools uploads the code from this directory when using the template to update the AWS CloudFormation stack.

resource-template.json

A `resource-template.json` file is an AWS CloudFormation template that defines the AWS resources associated with each resource group. You can specify any AWS resource type supported by AWS CloudFormation in your `resource-template.json` file. For a list of the available resource types, see the AWS CloudFormation AWS Resource Types Reference.

deployment-access-template.json

The `deployment-access-template.json` file is an AWS CloudFormation Template that defines the resources used to secure a deployment. For information about Cloud Canvas Resource Manager, see Understanding Cloud Canvas Resource Manager (p. 182). For security-related information, see Understanding the Resource Manager Security System (p. 393).

Parameters

The deployment access stack defines parameters that identify the deployment and other resources that are needed to set up security for the deployment. A value for each of these parameters is provided by Cloud Canvas when a deployment is created.

Resources

This section describes some of the key resources that are defined in the example `deployment-access-template.json` file.

Player

The `Player` resource describes the IAM role that determines the access granted to the player. For example, for the game to invoke a Lambda function, the player must be allowed the `lambda:InvokeFunction` action on the Lambda function resource.

The role's policies are determined by the `PlayerAccess` metadata elements found on resources in the project's resource templates (see `resource-template.json (p. 288)`). The role's policies are updated by the `PlayerAccess` custom resources that appear in the `deployment-access-template.json (p. 288)` and in the `resource-template.json (p. 288)` files. The `PlayerAccessIdentityPool` Amazon Cognito identity pool resource allows players to assume this role.

For more information, see PlayerAccessIdentityPool (p. 289) and Controlling Access to Resources (p. 393).

PlayerLoginRole

The `PlayerLoginRole` resources describes the IAM role that is temporarily assumed by the player as part of the login process.

For more information, see Controlling Access to Resources (p. 393).

PlayerLoginIdentityPool

The `PlayerLoginIdentityPool` resource describes the Amazon Cognito identity pool that provides the player with a temporary identity during the login process.

For more information, see Controlling Access to Resources (p. 393).

PlayerAccessIdentityPool

The `PlayerAccessIdentityPool` resource describes the Amazon Cognito identity pool that provides the player with a temporary identity during the login process.

For more information, see Controlling Access to Resources (p. 393).

deployment-template.json

The `deployment-template.json` file is an AWS CloudFormation Template that defines a child AWS CloudFormation stack resource for each of the project's resource groups. As described below, each resource group is an arbitrary grouping of the AWS resources that a game uses. For information about Cloud Canvas Resource Manager, see Understanding Cloud Canvas Resource Manager (p. 182).

local-project-settings.json

The `dev\<project name>\local-project-settings.json` file contains an identifier that points to AWS. The file content resembles the following:

```
{
    "ProjectStackId": "arn:aws:cloudformation:{aws-region}:{aws-access-id}:stack/{project-name}/{uuid}"
}
```

> **Note**
> As of Lumberyard 1.8, the `project-settings.json` file is stored in the project's Configuration Bucket (p. 297). The bucket is defined by the project's AWS CloudFormation stack template.

ProjectStackId Property

The `ProjectStackId` property identifies the AWS CloudFormation stack for the project. This stack contains the resources used by Cloud Canvas to manage your Lumberyard project.

The `ProjectStackId` property is set by the project create (p. 419) command. If for some reason you want to associate the project with an existing project stack, you can use the AWS Management Console to look up the stack's ARN and paste it into the `local-project-settings.json` file (navigate to AWS CloudFormation, select the stack, select **Overview**, and then copy the value of the `Stack Id` property).

project-template.json

The `project-template.json` file is an AWS CloudFormation template that defines project-wide resources that support the Cloud Canvas resource management system. For information about Cloud Canvas Resource Manager, see Understanding Cloud Canvas Resource Manager (p. 182). For security-related information, see Understanding the Resource Manager Security System (p. 393).

user-settings.json

The `user-settings.json` file contains user-specific configuration data.

File Location

The `user-settings.json` file is found at `dev\Cache\{game}\{platform}\user\AWS\user-settings.json`. It is not in the `dev\{game}\AWS` directory along with the other files described in this section because it should not be checked into the project's source control system.

Permissions Metadata for Resource Definitions

To determine what permissions should go into a policy, the `Custom::CloudGemFramework::AccessControl` resource looks for Cloud Canvas permissions metadata on resource definitions in resource group stacks. In the following example, the metadata on the `Messages` resource gives the `SayHello` AWS Lambda function permission to put items into a Amazon DynamoDB table. The metadata on the `SayHello` resource gives players permission to invoke the `SayHello` Lambda function.

```
...

"Messages": {
    "Type": "AWS::DynamoDB::Table",
    "Properties": {
        "AttributeDefinitions": [
            {
                "AttributeName": "PlayerId",
                "AttributeType": "S"
            }
        ],
        "KeySchema": [
            {
                "AttributeName": "PlayerId",
                "KeyType": "HASH"
            }
        ],
        "ProvisionedThroughput": {
            "ReadCapacityUnits": { "Ref": "ReadCapacityUnits" },
            "WriteCapacityUnits": { "Ref": "WriteCapacityUnits" }
        }
    },
    "Metadata": {
        "CloudCanvas": {
            "Permissions": [
                {
                    "AbstractRole": "SayHello",
                    "Action": "dynamodb:PutItem"
                }
            ]
        }
    }
},


"SayHello": {
    "Type": "AWS::Lambda::Function",
    "Properties": {
        "Description": "Example of a function called by the game to write data into a
 DynamoDB table.",
        "Handler": "main.say_hello",
        "Role": { "Fn::GetAtt": [ "SayHelloConfiguration", "Role" ] },
        "Runtime": { "Fn::GetAtt": [ "SayHelloConfiguration", "Runtime" ] },
        "Code": {
            "S3Bucket": { "Fn::GetAtt": [ "SayHelloConfiguration",
 "ConfigurationBucket" ] },
            "S3Key": { "Fn::GetAtt": [ "SayHelloConfiguration", "ConfigurationKey" ] }
        }
```

```
    },
    "Metadata": {
        "CloudCanvas": {
            "Permissions": [
                {
                    "AbstractRole": "Player",
                    "Action": "lambda:InvokeFunction"
                }
            ]
        }
    }
},

...
```

You can use the `lmbr_aws` command line tool to manage permissions metadata on the resource definitions in a resource group's `resource-template.json` file. For more information, see Permission Metadata Management (p. 404).

Properties

Each Cloud Canvas `Permission` metadata object can have the following properties.

-

Property	Description
`AbstractRole`	Required string or list of strings. Identifies the role whose policy will reflect the permission. For more information, see Role Mapping Metadata (p. 398).
`Action`	Required string or list of strings. Identifies the actions to be placed in the policy. This is used as the Action property of a Statement in the policy. See AWS Service Actions and Condition Context Keys for Use in IAM Policies for a list of actions supported by IAM.
`ResourceSuffix`	Optional string or list of strings. Provides values that are appended to the ARN in the Resource property of a statement in the policy. There is one `Resource` property value for each suffix listed. If no suffixes are listed, the `Resource` property value is the resource ARN with no suffix.

See Also

For an overview of security in Cloud Canvas Resource Manager, see Understanding the Resource Manager Security System (p. 393).

Importing Resource Definitions into Cloud Canvas

You can use the Cloud Canvas resource importer to add definitions of existing AWS resources to a Cloud Canvas resource group. You can add resources by using the Cloud Canvas Resource Manager in Lumberyard Editor or at a command line prompt.

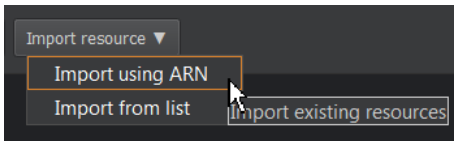
Importing Resources using Lumberyard Editor

In Lumberyard Editor, you can import a resource by specifying an Amazon Resource Name (ARN) or by choosing from a list.

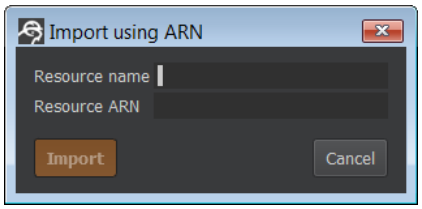
To import a resource by using an ARN

1. From the Lumberyard Editor top menu, choose **AWS**, **Cloud Canvas**, **Resource Manager**.
2. In the navigation pane, select a resource group.

3. In the detail window, click **Import resource**, **Import using ARN**. You can also open the context (right-click) menu for the resource in the navigation pane and choose **Import resource**, **Import using ARN**.



4. In the **Import using ARN** dialog box, provide the ARN and name of the resource that you are going to import. Both are required.

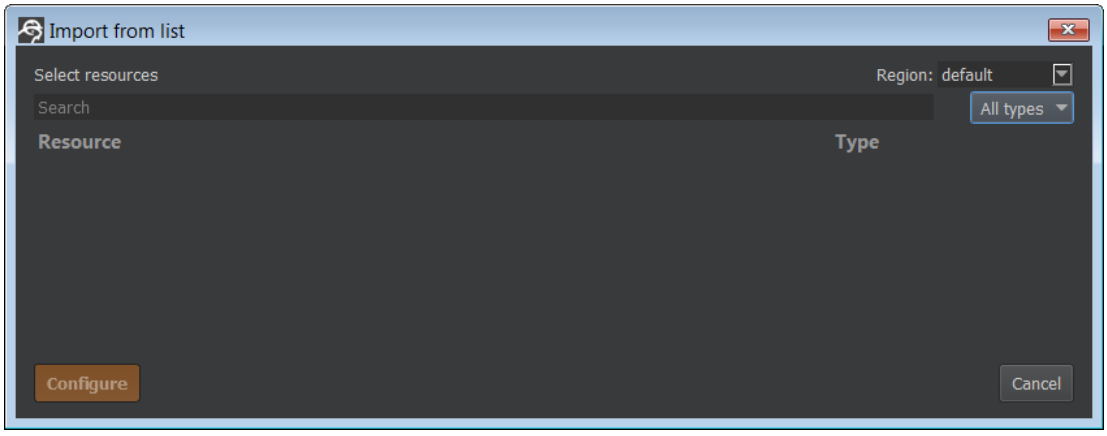


   After you have provided both items of information, the **Import** button is enabled.

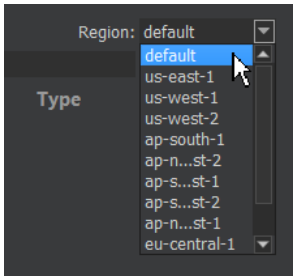
5. **Import**.

To import a resource by choosing from a list

1. From the Lumberyard Editor top menu, choose **AWS**, **Cloud Canvas**, **Resource Manager**.
2. In the navigation pane, select a resource group.
3. In the detail window, choose **Import resource**, **Import using ARN**. You can also open the context (right-click) menu for the resource in the navigation pane and choose **Import resource**, **Import using ARN**.

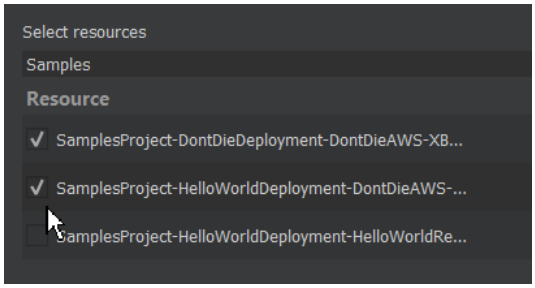


4. In the **Import from list** dialog box, choose the AWS Region of the resource for **Region**. The default value is the region of the project stack if it exists. Resources start loading in the list as soon as you choose a region that has importable resources.

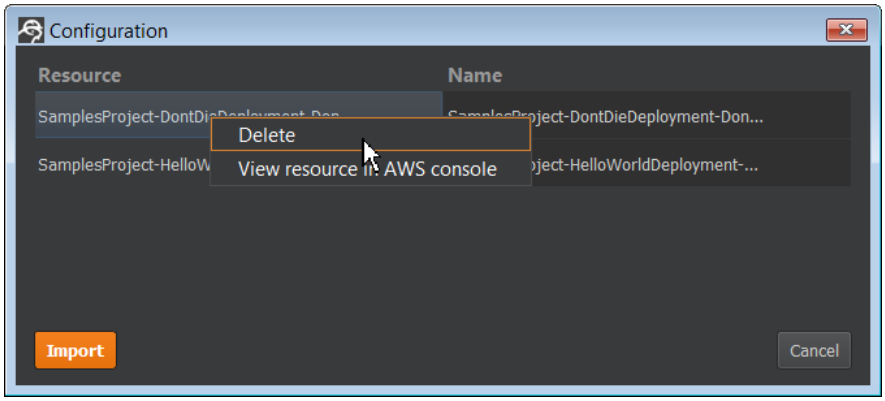


5. You can use the AWS service selector to filter the resources by service, and then use the **Search** box to filter resources by name.

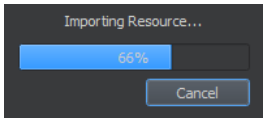
6. Select the check box to the left of each resource that you want to import.



7. **Configure**.
8. In the **Configuration** dialog box, provide a reference name for each resource, or accept the default. The default name is the original name of the resource on AWS.
9. To delete a selected resource from the list, open the context (right-click) menu for the resource and choose **Delete**.



10. When you are ready, click **Import**. A progress bar displays. An **Import Error** message informs you of any errors that occur.



11. Click **X** to close the **Import from list dialog** box. The resources that you imported are listed in the details pane of Cloud Canvas Resource Manager.

Importing Resource Definitions Using the Command Line

To list and import resources using the Cloud Canvas command line, see resource-importer list-importable-resources (p. 424) and resource-importer import-resource (p. 424).

Understanding Resource Definitions

When you use the Cloud Canvas resource importer to import the definition of a resource, it is important to understand that you are importing the resource's definition, not the resource itself. For example, suppose you use the AWS console to create a high score table in DynamoDB called Table A. You create a game client that uploads scores, and send out the client to your players. Table A begins to populate with data from the users who play your game.

You then decide to use Cloud Canvas to manage your resources and deployments. Using the Cloud Canvas Resource Manager, you import Table A because it has the exact configuration values that you want, and it has worked well for your use cases.

When you create a deployment with the imported resource, the deployment contains Table B, which is a new table with Table A's structure but not its data. Table B is managed by Cloud Canvas and has the same behavior as Table A. However, Table B is not a reference to Table A, and it does not have Table A's data or history. Keep this distinction in mind when you import resource definitions.

Automatically Imported Resource Definitions

Some of the existing resources that you select might be related to other resources. For example, Lambda functions can respond to events from the selected triggers. You can use event notifications from an Amazon S3 bucket to send alerts or trigger workflows. Cloud Canvas imports the related resources for you automatically.

Cloud Canvas uses the following naming conventions for automatically imported resource definitions.

Source	Naming Convention	Example Name of Imported Resource
DynamoDB table, Lambda function, Amazon SNS topic, Amazon SQS queue	*<resource_name>* + "AutoAdded" + *<resource_type>* + *<counter>*	`LambdaFunctionAutoAddedtable0`
Lambda function configuration resource??	*<lambda_function_name>* + "Configuration"	`LambdaFunctionConfiguration`
Lambda function policy resource	*<lambda_function_name>* + "Permission"	`LambdaFunctionPermission`
DynamoDB table Lambda function event source	*<DynamoDB_table_name>* + "EventSource"	`DynamoTableEventSource`

Resources Supported for Import

The following sections list the resource attributes and related resources that Cloud Canvas imports for each supported AWS service.

Dynamo DB Tables

For DynamoDB tables, Cloud Canvas imports the following resource attributes:

- `AttributeDefinitions`
- `GlobalSecondaryIndexes`
- `KeySchema`
- `LocalSecondaryIndexes`

- `ProvisionedThroughput`
- `StreamSpecification`

Amazon S3 Buckets

For Amazon S3 buckets, Cloud Canvas imports the following resource attributes:

- `CorsConfiguration`
- `LifecycleConfiguration`
- `NotificationConfiguration`
- `Tags`
- `VersioningConfiguration`
- `WebsiteConfiguration`

For Amazon S3 buckets, Cloud Canvas also imports the following related resources:

- Lambda functions
- Amazon SQS queues
- Amazon SNS topics

Lambda Functions

For Lambda functions, Cloud Canvas imports the following resource attributes:

- `Code`
- `Description`
- `Handler`
- `MemorySize`
- `Role`
- `Runtime`
- `Timeout`
- `VpcConfig`

For Lambda functions, Cloud Canvas also imports the following related resources:

- Lambda function configurations
- Lambda function permissions
- DynamoDB tables
- Event source mappings

Amazon SNS Topics

For Amazon SNS topics, Cloud Canvas imports the following resource attributes:

- `DisplayName`
- `Subscription`

For Amazon SNS topics, Cloud Canvas also imports any Lambda functions that are related resources.

SQS Queues

For SQS queues, Cloud Canvas imports the following resource attributes:

- `DelaySeconds`
- `MaximumMessageSize`
- `MessageRetentionPeriod`
- `ReceiveMessageWaitTimeSeconds`
- `RedrivePolicy`
- `VisibilityTimeout`

Resource Deployments

You implement deployments using AWS CloudFormation stacks. You create and manage the stacks using tools provided by Lumberyard.

A project may define any number of deployments, up to the limits imposed by AWS CloudFormation (for more information, see AWS CloudFormation Limits). Each deployment contains a completely independent set of the resources that the game requires. For example, you can have separate development, test, and release deployments so that your development and test teams can work independently of the deployment used for the released version of the game.

An AWS account that hosts a Lumberyard project contains the following resources:

- *{project}* – An AWS CloudFormation stack that acts as a container for all the project's deployments.
- *{project}*`-Configuration` – An S3 bucket used to store configuration data.
- *{project}*`-ProjectResourceHandler` – A Lambda function that implements the handler for the custom resources used in the templates. See Custom Resources (p. 300).
- *{project}*`-ProjectResourceHandlerExecution` – An IAM role that grants the permissions used by the `ProjectResourceHandler` Lambda function when it is executing.
- *{project}*`-ProjectPlayerAccessTokenExchangeHandler` – A Lambda function that implements the token exchange step in the player login process. For more information, see Controlling Access to Resources (p. 393).
- *{project}*`-ProjectPlayerAccessTokenExchangeHandlerRole` – An IAM role that grants the permissions used by the `ProjectPlayerAccessTokenExchangeHandler` Lambda function when it runs.
- *{project}*-*{deployment}* – AWS CloudFormation stacks for each of the project's deployments.
- *{project}*-*{deployment}*`Access` – AWS CloudFormation stacks that control access to each of the project's deployments.
- *{project}*-*{deployment}*`Access-PlayerAccessIdentityPool` – An Amazon Cognito identity pool used for player identity. For more information, see Controlling Access to Resources (p. 393).
- *{project}*-*{deployment}*`Access-PlayerLoginIdentityPool` – An Amazon Cognito identity pool that provides the temporary player identity used during the player login process. For more information, see Controlling Access to Resources (p. 393).
- *{project}*-*{deployment}*-*{resource-group}* – An AWS CloudFormation stack for each resource group of the project.
- *{project}*-*{deployment}*-*{resource-group}*-*{resource}* – The resources defined by a resource group. Because of how AWS CloudFormation works, parts of these names have unique identifiers appended to them. For example, for a project named MyGame with a deployment named Development and a feature named HighScore, the actual name of a Scores resource would be something like: `MyGame-Development-1FLFSUKM3MC4B-HighScore-1T7DK9P46SQF8-Scores-1A1WIH6MZKPRI`. The tools provided by Lumberyard hide these actual resource names under most circumstances.
- *{project}*-*{iam-policy-or-role}* – An IAM policy or role that manages resource and deployment permissions. For more information, see Cloud Canvas Built-In Roles and Policies (p. 396).

Configuration Bucket

The configuration Amazon S3 bucket is used to store configuration data for the project. The bucket is located at `/s3/buckets/<projectname>-configuration-<ID>/`. The tools provided with Cloud Canvas manage uploads to this bucket.

The `cgp-resource-code` directory contains resource and deployment information for cloud gems in the Cloud Gem Portal. The `upload` directory contains objects that are uploaded from the *{game}*`/AWS` directory by the Cloud Canvas tools when stack management operations are performed. The uploads for each operation get assigned a unique *{upload-id}* value to prevent concurrent operations from impacting each other.

project-settings.json

The `project-settings.json` file contains project configuration data.

The structure of the `project-settings.json` file is as follows:

```
{
    "{key}": "{value}",
    "deployment": {
        "{deployment}": {
            "{key}": "{value}",
            "resource-group": {
                "{resource-group}": {
                   "{key}": "{value}"
                }
            }
        }
    }
}
```

The *{key}* and *{value}* pairs represent individual settings. The pairs at the root apply to the project. The pairs under *{deployment}* apply to that deployment. The pairs under *{resource-group}* apply to that resource group. Either or both of *{deployment}* and *{resource-group}* can be `*`, to indicate the settings they contain apply to all deployments or resource groups, respectively. Settings under a named entry take precedence over settings under a `*` entry.

> **Note**
> As of Lumberyard 1.8, the `ProjectStackId` property is stored in the `dev\`*<project name>*`\AWS\local-project-settings.json` file. For more information, see ProjectStackId Property (p. 289).

DefaultDeployment Property

The `DefaultDeployment` property identifies the deployment that is to be used by default when working in Lumberyard Editor. The `DefaultDeployment` property in the `user-settings.json (p. 289)` file overrides this setting. The project and user defaults can be set using the `lmbr_aws` deployment default (p. 409) command. The `DefaultDeployment` setting is also used by the `lmbr_aws` mappings update (p. 416) command.

ReleaseDeployment Property

The `ReleaseDeployment` property identifies the deployment that is to be used in release builds of the game. The `ReleaseDeployment` setting is used by the `lmbr_aws` mappings update (p. 416) command.

DeploymentStackId Property, PendingDeploymentStackId

The `DeploymentStackId` property identifies the AWS CloudFormation stack for a deployment. The `PendingDeploymentStackId` property identifies a stack whose deployment is pending. The project's resource groups are children of these stacks. For more information, see Resource Deployments (p. 296).

The `DeploymentStackId` property is set by the deployment create (p. 409) command. If for some reason you want to associate the deployment with an existing deployment, you can use the AWS Management Console to look up the stack's ARN and paste it into the `project-settings.json` file (navigate to AWS CloudFormation, select the stack, select **Overview**, and then copy the value of the `Stack Id` property).

DeploymentAccessStackId Property

The `DeploymentAccessStackId` property identifies the AWS CloudFormation stack for the resources that control access to a deployment.

The `DeploymentAccessStackId` is set by the deployment create (p. 409) command. If for some reason you want to associate the deployment with an existing deployment stack, you can use the AWS Management Console to look up the stack's ARN and paste it into the `project-settings.json` file (navigate to AWS CloudFormation, select the stack, select **Overview**, and then copy the value of the `Stack Id` property).

parameter Property

The `parameter` property provides the values for resource template parameters. The property must be in the following format.

```
{
    ...
                    "parameter": {
                        "{template-parameter-name-1}": {template-parameter-value-1},
                        ...
                        "{template-parameter-name-n}": {template-parameter-value-n}
                    }
    ...
}
```

Resource Mappings

Resource mappings map the friendly names used in a game's Resource Definitions (p. 287) to the actual names of the resources created for one or more specific Resource Deployments (p. 296). For example, a DynamoDB table name like DailyGiftTable would get mapped to a name like `SamplesProject-DontDieDeployment-78AIXR0N0O4N-DontDieAWS-1I1ZC6YO7KU7F-DailyGiftTable-1G4G33K16D8ZS` where `SamplesProject` is the name of the project, `DontDieDeployment` is the name of a deployment, and `DontDieAWS` is the name of a resource group. The `78AIXR0N0O4N`, `1I1ZC6YO7KU7F` and `1G4G33K16D8ZS` parts of the resource name are inserted by AWS CloudFormation to guarantee that the resource name is unique over time. Thus, even if a resource is deleted and a new one with the same logical name is created, the physical resource ID will be different.

Usually different deployments, and consequently different mappings, are used for game development and for the released version of a game. Furthermore, different development, test, and other teams often work with their own deployments so that each team has distinct mappings.

The deployment used by default during development is specified in the **`project-settings.json`** `(p. 297)` file. The file is located in the Amazon S3 configuration bucket at `/s3/buckets/<projectname>-configuration-<ID>/project-settings.json`. The file can be overridden for each user by the `dev\Cache\{game}\pc\user\AWS\user-settings.json (p. 289)` file. You can change the default deployment by using the `lmbr_aws` deployment default (p. 409) command or by using the Cloud Canvas Resource Manager (p. 389).

The mappings used during development when the game is launched from the Lumberyard IDE by pressing **Ctrl+G** are stored in the `user-settings.json (p. 289)` file just mentioned. This file is updated automatically when the default deployment changes, when the default deployment is updated, and when Lumberyard Editor is started. It can be refreshed manually by using the `lmbr_aws` mappings update (p. 416) command.

When a game launcher application created in Lumberyard launches a release build of a game, the mappings for the game are stored in the *{root}**{game}*\Config\awsLogicalMappings.json file. These mappings can be updated manually using the lmbr_aws mappings update --release (p. 416) command, which produces the awsLogicalMappings.json file. You can specify the deployment for the release mappings in the ReleaseDeployment property of the **project-settings.json** (p. 297) file.

For more information, see Running AWS API Jobs Using the Cloud Gem Framework (p. 350).

Using Mappings in AWS Flow Nodes

AWS flow nodes that define TableName (DynamoDB), FunctionName (Lambda), QueueName (Amazon SQS), TopicARN (Amazon SNS), or BucketName (Amazon S3) ports work with mappings. Set the port to a value like *{resource-group}*.*{resource}* where *{resource-group}* is the name of the resource group that defines the resource, and where *{resource}* is the name of the resource that appears in the *Resources* section of the resource group's resource-template.json file. For detailed information on the Cloud Canvas flow graph nodes, see the Cloud Canvas Flow Graph Node Reference (p. 254).

Using Mappings in Lambda Functions

Lambda function resources defined as part of a resource group often need to access other resources defined by that resource group. To do this, the function code needs a way to map a friendly resource name to the actual resource name used in AWS API calls. The LambdaConfiguration resource provides a way to such mappings, as well as other settings, to the lambda code. For more information, see LambdaConfiguration (p. 303).

Resource Manager Resource Group Parameters

Cloud Canvas Resource Manager uses AWS CloudFormation templates to describe the AWS resources needed for a project. AWS CloudFormation templates can have parameters. You provide the values for these parameters when a template creates or updates a stack.

AWS CloudFormation template parameters are useful when you want separate deployments of a resource group to be configured differently. For example, you could configure an internal test instance of Amazon DynamoDB to use lower throughput than a public instance.

Follow the steps below to use AWS CloudFormation template parameters for your resource groups.

To use template parameters

1. Define a parameter in the AWS CloudFormation template. You must provide a default value for the parameter. Be sure not to remove or modify the parameter definitions used by Cloud Canvas (for example, ConfigurationBucket or ConfigurationKey).
2. Reference the parameter when you define a resource. You can use parameters to provide values for resource properties, including passing settings to Lambda functions through the Cloud Canvas Custom::CloudGemFramework::LambdaConfiguration (p. 303) resource.
3. Use the lmbr_aws parameter list, parameter set, and parameter clear commands to view and set parameter values. These commands are described later in this document.
4. Update (or create) the resource group stack using the lmbr_aws resource-group upload command, or click **Upload Resources** in the Cloud Canvas Resource Manager window in Lumberyard Editor.

Parameter Configuration

A project's parameter configuration is stored in a **project-settings.json** (p. 297) file object in the project's configuration bucket. The bucket is defined by the project's AWS CloudFormation stack template.

You can specify parameter values for a specific deployment or for all deployments by using the `*` character as a wildcard in place of a deployment name. You can also specify parameter values for a specific resource group or for all resource groups by using the `*` in place of a resource group name. If you provide a parameter value for both a wildcard (`*`) entry and a named entry, the value from the named entry overrides the value from the wildcard entry.

The following table shows the `lmbr_aws` commands for listing, setting, and clearing parameter values. Your project must be initialized (that is, a project stack must have been created) before you can list, set, or clear parameter values. For usage details, visit the corresponding links in the table.

`lmbr_aws` command	**Description**
parameter list (p. 417)	Lists the parameters currently configured for your project.
parameter set (p. 417)	Sets parameter configuration for your project.
parameter clear (p. 416)	Clears the specified parameter configuration for your project.

Custom Resources

Cloud Canvas provides a number of AWS CloudFormation custom resources that can be used in the AWS CloudFormation template files for the project, deployment, and resource group. These custom resources are implemented by the Lambda function code found in the `dev\{game}\AWS\project-code` directory and the `ProjectResourceHandler` resource defined in the `dev\{game}\AWS\project-template.json` file. Rather than static entities, these resources act more like library functions. Each custom resource has input and output properties.

A summary list of custom resources follows.

- CognitoIdentityPool (p. 300) – Manages Amazon Cognito identity pool resources.
- CognitoUserPool (p. 301) – Manages Amazon Cognito user pool resources.
- EmptyDeployment (p. 302) – Used in the `deployment-template.json` when there are no resource groups defined.
- ResourceGroupConfiguration (p. 302) – Provides configuration data for a resource-group's AWS CloudFormation stack resource.
- LambdaConfiguration (p. 303) – Provides configuration data for Lambda function resources and maintains the Lambda function's execution role.
- Helper (p. 304) – Provides convenience functions for use in templates.
- Cloud Gem Framework Service API (p. 352) – Provides programmatic access to cloud gem functionality that you can use to implement cloud-connected features in your game.

For information on controlling access to resources, see Controlling Access to Resources (p. 393).

CognitoIdentityPool

The `Custom::CloudGemFramework::CognitoIdentityPool` resource is used in the `deployment-access-template.json` file to create and configure Amazon Cognito identity pool resources.

Input Properties

- `ConfigurationBucket`

  Required. The name of the Amazon S3 bucket that contains the configuration data.

- `ConfigurationKey`

  Required. The Amazon S3 object key prefix where project configuration data is located in the configuration bucket. This property causes the custom resource handler to be executed by AWS CloudFormation for every operation.

- `IdentityPoolName`

  Required. The name of the identity pool.

- `UseAuthSettingsObject`

  Required. Must be either `true` or `false`. Determines whether the Amazon Cognito identity pool is configured to use the authentication providers that were created from the `add-login-provider` command.

- `AllowUnauthenticatedIdentities`

  Required. Must be either `true` or `false`. Determines whether the Amazon Cognito identity pool is configured to allow unauthenticated identities. See Identity Pools for more information on Amazon Cognito's support for authenticated and unauthenticated identities.

- `Roles`

  Optional. Determines the IAM role assumed by authenticated and unauthenticated users. See SetIdentityPoolRoles for a description of this property.

Output Properties

- `IdentityPoolName`

  The name of the identity pool (same as the `IdentityPoolName` input property).

- `IdentityPoolId`

  The physical resource name of the identity pool.

CognitoUserPool

The `Custom::CloudGemFramework::CognitoUserPool` resource is used in the Player Account Cloud Gem (p. 241) `resource-template.json` file to create and configure Amazon Cognito user pool resources. You can use this resource to add Amazon Cognito user pools and link them to an Amazon Cognito identity pool.

Input Properties

- `PoolName`

  Required. The name of the user pool.

- `ConfigurationKey`

  Required. The Amazon S3 object key prefix where project configuration data is located in the configuration bucket. This property causes the custom resource handler to be executed by AWS CloudFormation for every operation.

- `ServiceToken`

  Required. The handler for this custom resource type.

- `ClientApps`

  Required. A list of client apps to be created for the user pool. At least one must be provided.

- `LambdaConfig`

  Optional. A map of a Lambda trigger name to a Lambda function ARN. This map is passed to the Amazon Cognito `CreateUserPool` API when a user pool is created and to the Amazon Cognito `UpdateUserPool` API when a user pool is updated.

Metadata: Linking a User Pool to an Identity Pool

To link a `Custom::CloudGemFramework::CognitoUserPool` to a `Custom::CloudGemFramework::CognitoIdentityPool`, you add an `Identities` section to the `Metadata` `CloudCanvas` section, as in the following example.

```
"Metadata": {
    "CloudCanvas": {
        "Identities": [
            {
                "IdentityPoolName": "PlayerAccess",
                "ClientApp": "DefaultClientApp"
            }
        ]
    }
}
```

The handler for `Custom::CloudGemFramework::CognitoIdentityPool` looks for this metadata on `Custom::CloudGemFramework::CognitoUserPool` resources when it configures the identity pool. The `IdentityPoolName` in the user pool's metadata must match the `IdentityPoolName` in the identity pool's properties. `ClientApp` must be one of the apps listed in the user pool's `ClientApps` property.

Output Properties

- `UserPoolName`

  The name of the user pool.

- `UserPoolId`

  The ID of the user pool.

- `ClientApps`

  A list that contains the `ClientId` and `ClientName` of each of the user pool's client apps.

- `PhysicalResourceId`

  The physical resource ID of the user pool.

EmptyDeployment

The `Custom::CloudGemFramework::EmptyDeployment` resource is used in the `deployment-template.json` file when there are no resource groups defined. This is necessary to satisfy the AWS CloudFormation requirement that a template define at least one resource.

This resource supports no input or output properties.

ResourceGroupConfiguration

The `Custom::CloudGemFramework::ResourceGroupConfiguration` resource is used in the `deployment-template.json` to identify the location of the copy of the `resource-template.json` file in the configuration bucket that should be used for a specific resource group.

Input Properties

- `ConfigurationBucket`

  Required. The name of the Amazon S3 bucket that contains the configuration data.

- `ConfigurationKey`

  Required. The Amazon S3 object key prefix where the deployment configuration data is located in the configuration bucket.

- `ResourceGroup`

  Required. The name of the resource group that is to be configured.

Output Properties

- `ConfigurationBucket`

  The name of the Amazon S3 bucket that contains the configuration data. This is always the same as the `ConfigurationBucket` input property.

- `ConfigurationKey`

  The Amazon S3 object key prefix where the specified resource group's configuration data is located in the configuration bucket. This is the input `ConfigurationKey` with the string `ResourceGroup` and the value of `ResourceGroup` appended.

- `TemplateURL`

  The Amazon S3 URL of the resource group's copy of the `resource-template.json` in the configuration bucket. This value should be used as the resource group's `TemplateURL` property value.

LambdaConfiguration

The `Custom::CloudGemFramework::LambdaConfiguration` resource is used in `resource-template.json` files to provide configuration data for Lambda function resources.

Input Properties

- `ConfigurationBucket`

  Required. The name of the Amazon S3 bucket that contains the configuration data.

- `ConfigurationKey`

  Required. The Amazon S3 object key prefix where configuration data for the resource group is located in the configuration bucket.

- `FunctionName`

  Required. The friendly name of the Lambda function resource being configured.

- `Settings`

  Optional. Values that are made available to the Lambda function code.

- `Runtime`

  Required. Identifies the runtime used for the Lambda function. Cloud Canvas currently supports the following Lambda runtimes: `nodejs`, `python2.7`.

Output Properties

- `ConfigurationBucket`

  The name of the Amazon S3 bucket that contains the configuration data. This is always the same as the `ConfigurationBucket` input property.
- `ConfigurationKey`

  The Amazon S3 object key prefix where the specified function's zipped code is located in the configuration bucket.
- `Runtime`

  The Lambda runtime used by the function. This is always the same as the input `Runtime` property value.
- `Role`

  The ID of the Lambda function execution created for this function.

For information on how the `LambdaConfiguration` custom resource is used to allow Lambda functions to perform specified actions on specific project resources, see Controlling Access to Resources (p. 393).

Helper

The `Custom::CloudGemFramework::Helper` resource is used in templates to perform simple tasks similar to those performed by AWS CloudFormation's Intrinsic Functions.

Input Properties

- `Input`

  Provides a JSON object that is processed by the custom resource handler.

Output

The output is the processed version of the JSON object that is provided by the `Input` property.

Helper Function

The `Helper` resource supports the following function. The function is applied to the `Input` property value by resource handler to produce its output.

HelperFn::LowerCase

Converts a string to lowercase.

Syntax

```
{ "HelperFn::LowerCase" : "converted-string-value" }
```

Example

The following example from the `project-template.json` file uses `HelperFn::LowerCase` to lower the case of a stack name.

```
{
    ...
    "Resources": {
```

```
    ...
        "Helper": {
            "Type": "Custom::CloudGemFramework::Helper",
            "Properties": {
                "Input": {
                    "LowerCaseStackName": { "HelperFn::LowerCase": { "Ref":
 "AWS::StackName" } }
                },
                "ServiceToken": ...
            }
        },
    ...
            { "Fn::Join": [ "", [ "arn:aws:s3:::", { "Fn::GetAtt": [ "Helper",
 "LowerCaseStackName" ] }, "-*" ] ] }
    ...
}
```

Cloud Gem Framework

The Cloud Gem Framework and this documentation are in preview release and are subject to change. More topics are planned.

The Lumberyard Cloud Gem Framework makes it easy to build popular cloud-connected features, such as dynamic content, leaderboards, and daily messages. The Cloud Gem Framework has two components:

- **Cloud Gem Portal** – A web application for visually managing and administering your cloud features (like scheduling messages, releasing dynamic content, or deleting a fraudulent leaderboard score)
- **Cloud gems** – Modular packages of discrete functionality and assets that include everything necessary for a developer to include that functionality into their project, including backend and client functionality.

Cloud gems can be used out of the box in production, and they come with full source code in case you want to customize their behavior.

Topics

- Getting Started with the Cloud Gem Framework (p. 305)
- Getting Started With Development on the Cloud Gem Portal (p. 308)
- Making HTTP Requests Using the Cloud Gem Framework (p. 328)
- Cloud Gem Framework Resource Manager Hooks (p. 330)
- Running AWS API Jobs Using the Cloud Gem Framework (p. 350)
- Cloud Gem Framework Service API (p. 352)
- Using the Cloud Gem Framework Command Line (p. 379)

Getting Started with the Cloud Gem Framework

Lumberyard cloud gems make it easy to create cloud-connected functionality for your games. You can use the cloud gems included with Lumberyard, or use Lumberyard's Cloud Gem Framework to create your own cloud gems.

Cloud Gems

Cloud gems are Lumberyard gems that use Cloud Canvas Resource Manager (p. 182) (CCRM) to define AWS resources that support their operation. CCRM provides the infrastructure needed to create,

configure, and access AWS resources. A cloud gem defines resources and implements the required operations to create a complete game subsystem.

Cloud gems use general purpose AWS features that operate in your AWS account. They are not game-specific services operated by Amazon. Cloud gems follow a serverless microservice architecture that makes use of AWS Lambda functions for computation, and Amazon DynamoDB, Amazon RDS, and Amazon S3 for data storage. However, a cloud gem can use any AWS service.

For example, a cloud gem could provide a leaderboard system that uses a DynamoDB table to store high scores. The gem could use an AWS Lambda function to validate submitted scores and allow fraudulent high scores to be deleted.

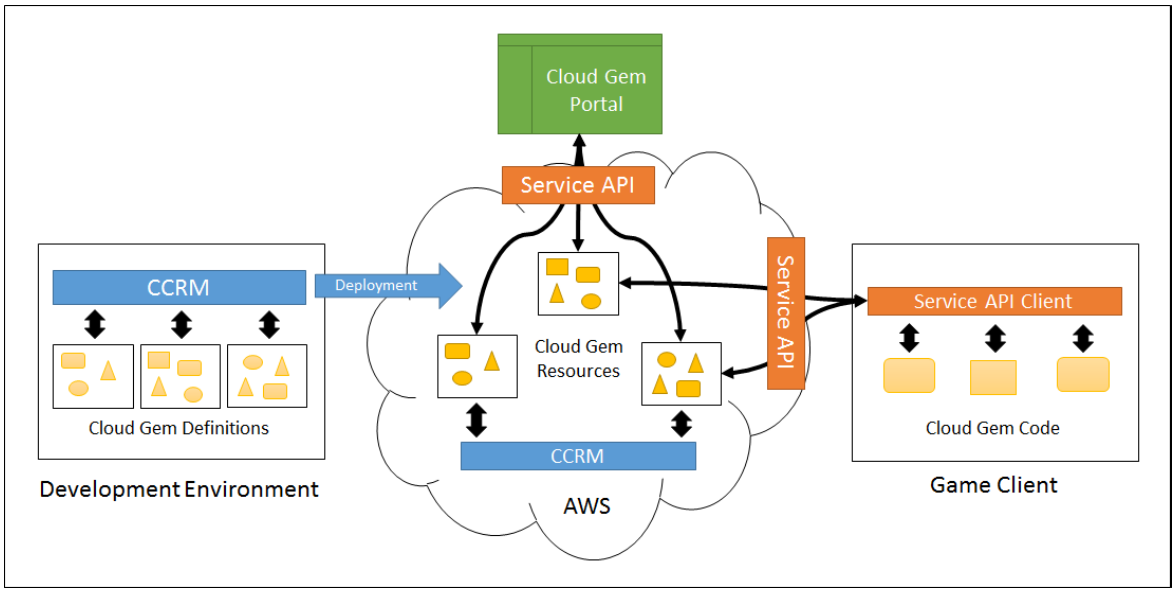
In this scenario, CCRM performs the following tasks:

- Creates the DynamoDB table and Lambda function defined by the gem.
- Enables the game client to invoke the API to submit a score.
- Protects the integrity of the leaderboard by removing fraudulent scores automatically.
- Permits the creation of a user interface that enables an authorized employee to remove fraudulent scores manually.

Cloud Gem Framework

The Cloud Gem Framework is a collection of tools and API operations that support the creation and operation of cloud gems.

The following diagram shows a high-level view of the Cloud Gem Framework architecture.



The Cloud Gem Framework contains the following major components:

- Cloud Gem Portal (p. 200) – A web application for operating and managing cloud gems.
- Cloud Gem Framework Service API (p. 352) – An API that provides a secure interface to a cloud gem's functionality.
- AWS API Jobs (p. 350) – Integrates the AWS API with the Lumberyard job execution systems (not illustrated).
- HTTP Request Job (p. 328) – A Lumberyard job for executing HTTP requests (not illustrated).

Creating a Cloud Gem

Lumberyard includes some cloud gems, such as leaderboards and dynamic content distribution, that provide full-featured services. You can use these gems in your games with a minimum of effort. Because the full source code for these cloud gems is included with Lumberyard, you can customize them or use them as examples for developing your own cloud gems.

To create a new cloud gem

1. Use the Lumberyard Project Configurator to create a new gem.
2. In Project Configurator, enable the gem for your project.
3. In Project Configurator, enable the **Cloud Gem Framework** gem for your project.
4. In the `\dev\Gems\<gem-name>` directory for your new gem, add the following dependency to the `gem.json` file.

```
"Dependencies": [
    {
        "Uuid" : "6fc787a982184217a5a553ca24676cfa",
        "VersionConstraints": [ "~>0.1" ],
        "_comment": "CloudGemFramework"
    },
    {
        "Uuid" : "102e23cf4c4c4b748585edbce2bbdc65",
        "VersionConstraints": [
            "~>0.1"
        ],
        "_comment": "CloudCanvasCommon"
    }
],
```

5. Type the following command line to add a CCRM resource group to your gem. The command creates a `\dev\Gems\<gem-name>\AWS` directory with a `resource-template.json` file and a `lambda-function-code` directory.

```
lmbr_aws resource-group add --gem --resource-group <gem-name>
```

6. Type the following command line to add service API resources to the resource group.

```
lmbr_aws cloud-gem-framework add-service-api-resources --resource-group <gem-name>
```

   The `add-service-api-resources` command adds `ServiceApi`, `ServiceLambda`, and `ServiceLambdaConfiguration` resources to the resource group. It also adds an `Output` for the service API URL. The `AccessControl` resource is configured as described in Controlling Access to Resources (p. 393). The `add-service-api-resources` command also creates a `swagger.json` file that describes the API for the cloud gem. For more information, see Cloud Gem Framework Service API (p. 352).
7. Add any additional resource definitions to the `resource-template.json` file that the cloud gem requires.
8. Implement the cloud gem's functionality by adding code to the `lambda-function-code` directory. You upload this code to the `ServiceLambda` Lambda function resource that you created earlier.
9. Add paths and operations to the `swagger.json` file to create an API operation that exposes the cloud gem's functionality. For more information, see Cloud Gem Framework Service API (p. 352).
10. Create a CCRM project stack and deployment stack by following the guidance in the Getting Started with Cloud Canvas (p. 188) tutorial. This process uses the resource descriptions in your `resource-template.json` file to create resources in AWS for your cloud gem.
11. Type the following command to generate a service API client for the cloud gem.

```
lmbr_aws cloud-gem-framework generate-service-api-code --resource-group <gem-name>
```

The `generate-service-api-code` command reads the cloud gem's API definitions from the `swagger.json` file and generates C++ code in the `/dev/Gems/<gem-name>/Code/AWS` directory. This code implements a Lumberyard component that you can use to invoke the cloud gem's service API operations from C++ or Lua.

12. Use Lumberyard components, EBuses, and other features to implement game client functionality for the gem. Place the code in the cloud gem's `/dev/Gems/<gem-name>/Code` directory. You can use the service API client that you generated to access the functionality in the cloud gem's `ServiceLambda` Lambda function.

Getting Started With Development on the Cloud Gem Portal

To help you get started with development on the Cloud Gem Portal (CGP), this tutorial shows you how to do the following:

- Set up your environment for CGP development.
- Create a cloud gem and a dynamic Cloud Gem Portal.
- Test your Cloud Gem Portal locally.
- Deploy your Cloud Gem Portal to AWS and view it in production.

Setting Up Your Environment

Setting up your environment includes installing Node.js packages, Git, creating a test URL, and configuring Microsoft Visual Studio if you are using it.

Installing Node.js and Git

The Cloud Gem Portal uses the Angular 2 version of AngularJS for development and the Node.js package manager (npm) for managing dependencies.

To install Node.js and git

1. Download and install Node.js from the Node.js website.
2. Download and install Git from the Git download page.
3. Obtain a GitHub account if you do not have one. You must have a GitHub account to download the GitHub packages for the Cloud Gem Portal and for the optional JSPM package management tool.

Updating Packages and Running a Local Server

From a command prompt, run the following commands to change to the `CloudGemPortal` directory, install packages, and run a gulp server.

To update packages and run a local server

1. Type the following command to change to the `CloudGemPortal` directory.

```
cd <Lumberyard directory path>\dev\Gems\CloudGemFramework\Website\CloudGemPortal\
```

2. If you want to use JSPM, type the following command.

```
npm install -g jspm
```

3. Run the following commands.

```
npm install -g gulp
npm install
gulp serve
```

   The `npm install` command installs the packages that you need to run the Cloud Gem Portal website. The `gulp serve` command runs the default gulp task to set up a browsersync server. You do not have to run `npm install` again unless you add or update packages.

4. The output of the `gulp serve` command looks like the following.

```
[  ] Access URLs:
------------------------------------
       Local: http://localhost:3000
       External: http://00.00.00.0:3000
------------------------------------
       UI: http://localhost:3001
      UI External: http://00.00.00.0:3001
------------------------------------
```

   Make a note of the port number that the web server is using.

5. Leave the terminal window running so that gulp can act as your web server.

   For faster development, you can use the `gulp serve-watch` command instead of `gulp serve`. The `gulp serve-watch` command watches for changes to TypeScript files and injects the new versions without requiring a reloading of all the components of the website. This command also compiles Sass to CSS at run time so that it can monitor changes to your `.scss` files.

   Note
   For Lumberyard version 1.9, the `gulp serve-watch` command supports hot loading of Cloud Gem Portal scripts but not cloud gem scripts.

Your website is now running on a local server. Because it's missing the encrypted AWS credentials in the URL, you cannot use it publicly yet.

Creating a Test URL

The default temporary URL that Lumberyard generates is valid for a relatively short time (about 15 minutes). In order to create a longer lasting test URL, perform the following steps.

To generate a new token with a longer timeout

1. In the `CloudGemFramework\AWS\resource-manager-code` directory, open the `cloud_gem_portal.py` file. Comment out lines 70 and 71.

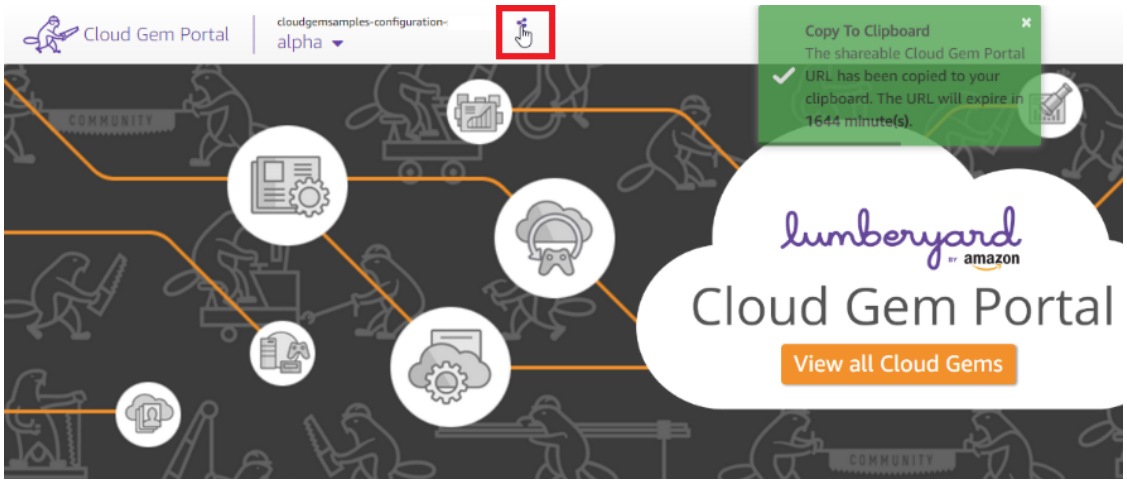
```
70  # if expiration < 900 or expiration > 3600:
71  #     raise HandledError('the --duration-seconds option value must be in the range 900-3600, inclusive.')
```

2. From the *<Lumberyard>*`\dev` directory, run the following command. The command generates a longer lasting URL (129600 seconds = 1.5 days).

```
lmbr_aws cloud-gem-framework cloud-gem-portal --duration-seconds 129600
```

   After you run the command, your Amazon S3–hosted Cloud Gem Portal page opens in your default browser.

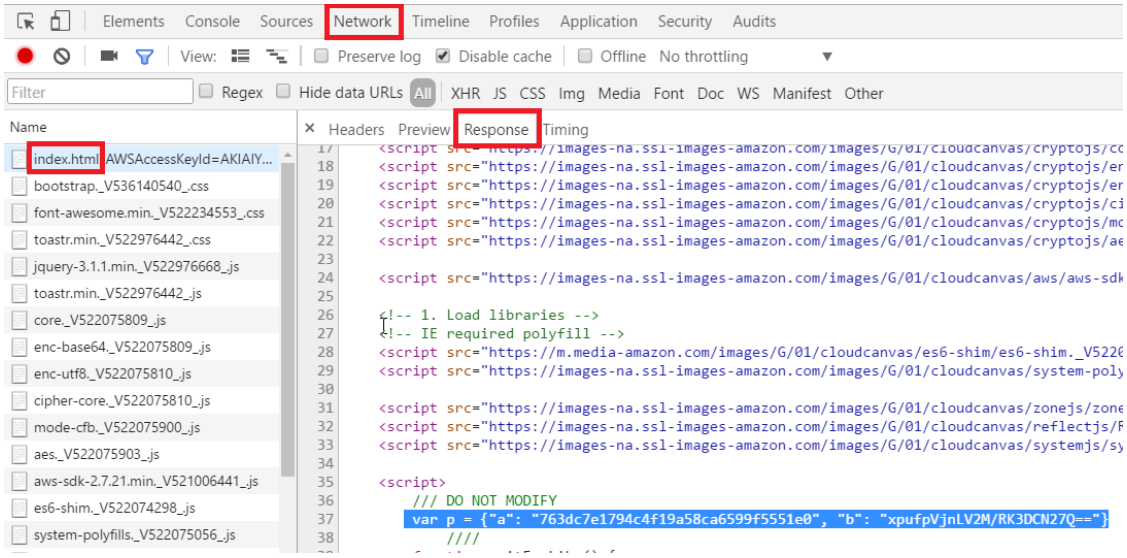
3. Click the **Share** icon at the top of the Cloud Gem Portal website. This copies your Cloud Gem Portal URL to the clipboard. You could copy the URL straight from the browser address bar, but it would be unavailable after a short time.



Next, you must modify your `CloudGemFramework\Website\CloudGemPortal\index.html` file to match your newly generated encrypted AWS tokens.

To modify your Cloud Gem Portal development `index.html` file

1. Open your browser dev tools. (Press **Ctrl** + **Shift** + **i** in Google Chrome and Mozilla Firefox or **F12** in Microsoft Edge.)
2. In the browser dev tools, click the **Network** tab.
3. Do one of the following to reload the page:
   - Click the **Cloud Gem Portal** label in the upper left corner of the website.
   - Paste the URL into the browser address bar and press **Enter**.
4. After the network calls are displayed, scroll up to the `index.html` file. When sorted by timestamp, the file should be the first network call.
5. Click the `index.html` file.
6. Click the **Response** tab of the `index.html` request.
7. Scroll down in the `.html` file until you find the declaration of the `p` variable.

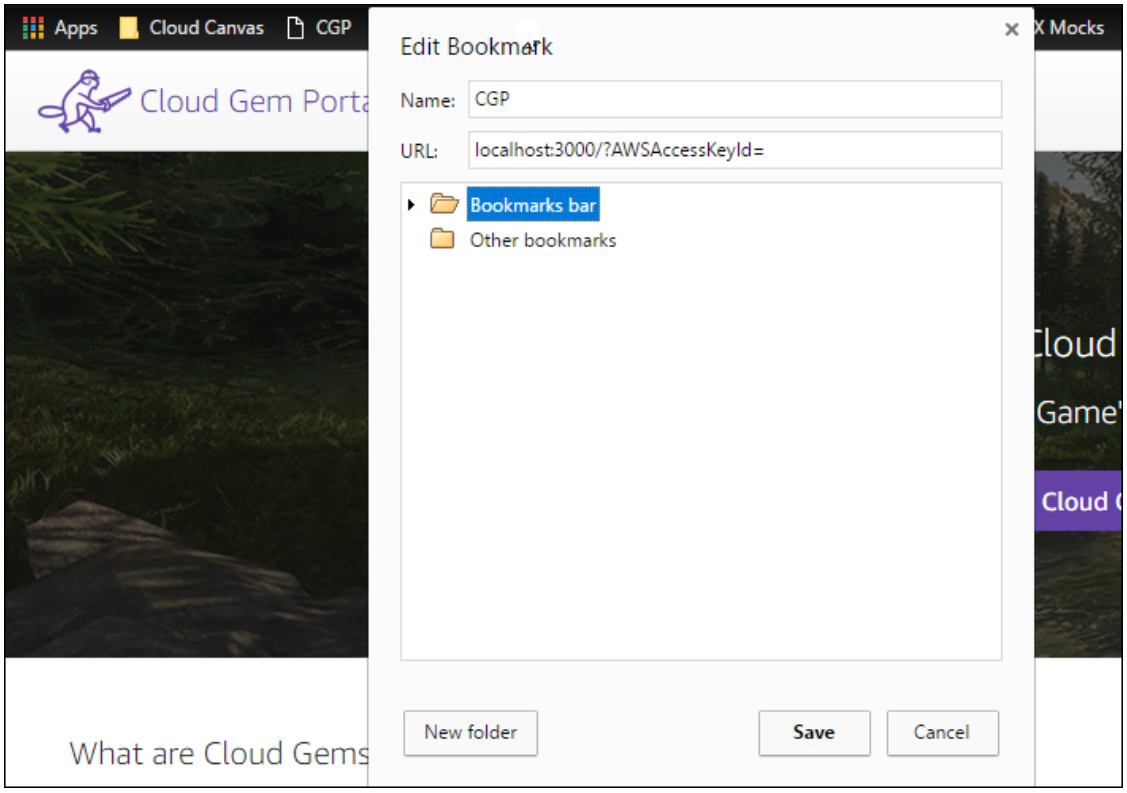


The declaration resembles the following.

```
var p = {"a": "763dc7e1794c4f19a58ca6599f5551e0", "b": "k8VJEHr4mNuwko2kU8ZySg=="}
```

8. Copy the `var p` line.
9. Open the `CloudGemFramework\Website\CloudGemPortal\index.html` file for editing.
10. Paste the line into the corresponding location of your local `index.html` file.

11. To test your portal locally, you must change the server to point to your development computer instead of to Amazon S3. To do so, press **Ctrl+D** to create a bookmark of the URL that you copied in To generate a new token with a longer timeout (p. 309).
12. In the bookmark URL, replace the part before the query string (everything before the **?**) with `localhost:<portnumber>`. For example, `localhost:3000?<querystring>`.



The port number can vary. Type the port that was shown in the terminal window when you ran `gulp serve` in To update packages and run a local server (p. 308).

13. To test your local instance of the Cloud Gem Portal, make sure that your server is still running in the terminal window with `gulp serve` command. Then click your bookmark to open the **Cloud Gem Portal** welcome page.

Note
The newer versions of the Cloud Gem Portal add caching of the URL for the session. Therefore, when you generate a new URL, be sure to delete the old URL from your browser's session cache.

At this point you can begin creating your own cloud gems.

Troubleshooting

If your browser displays a blank page, press **Ctrl+Shft+i** to check your browser developer tools console for errors. Common errors include the following:

- Encryption keys found in the index.html page are unable to decrypt the URL fragment.

  Console Error

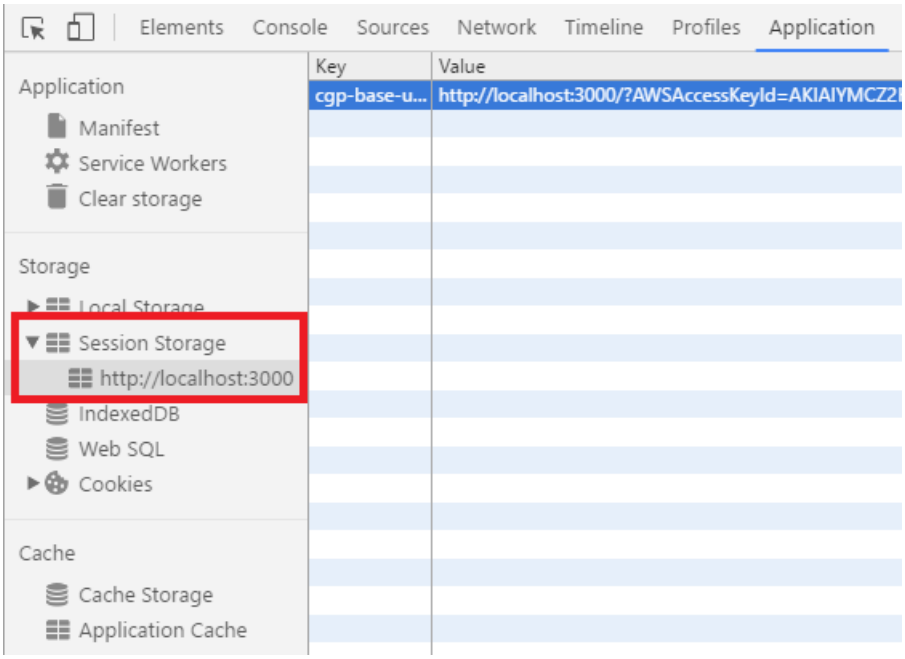
  ```
  EXCEPTION: Uncaught (in promise): Error: Malformed UTF-8 data
  ```

  Resolution

  1. Verify that the production URL query string and fragment are copied correctly to your development URL bookmark.
  2. Ensure that the source code `p` object in your production URL `index.html` file matches the `p` object in your development `index.html` file. The `p` object looks is similar to the following:

     ```
     var p = { "a": "763dc7e1794c4f19a58ca6599f5551e0", "b": "U+FvRIOPY8TlAjM9X+hyw==" }
     ```

  3. Your browser might be using an older, cached URL. Delete the cached URL from your browser's session storage, as shown in the following image.



- The `gulp serve` command causes errors.

  Example Terminal Error

  ```
  error TS2304: Cannot find name '<LibraryName>'
  ```

  Resolution

  The `tsconfig.json` file located in your *<Lumberyard directory path>*`\dev\Gems\CloudGemFramework\Website\CloudGemPortal\` directory is missing the proper library references. In the `compilerOptions` section, include the line `"lib": [ "es2015", "dom", "scripthost" ]`, as in the following example.

  ```
  {
  "compilerOptions": {
     ...
     "lib": [ "es2015", "dom", "scripthost"
     ...
     }
  }
  ```

Microsoft Visual Studio Setup

To use Visual Studio 2015 to create cloud gems, some additional setup is required. This includes configuring Visual Studio to work with Node.js, adding some TypeScript extensions, and optionally installing Task Runner Explorer to run gulp tasks.

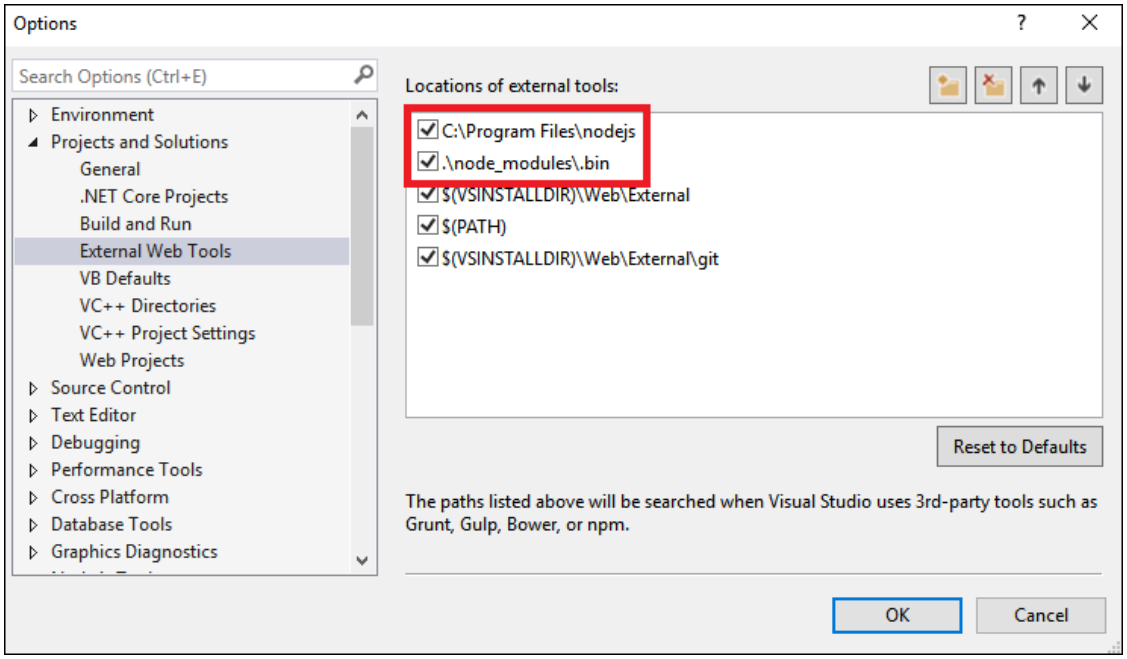
Configuring Visual Studio to Work with Node.js

Perform the following steps to configure Visual Studio to work with Node.js.

To configure Visual Studio to work with Node.js

1. Open the *<lumberyard>*`\dev\Gems\CloudGemFramework\Website\CloudGemPortal\Portal.csproj` solution file from Visual Studio.

2. In the navigation bar, click **Tools**, **Options**, **Projects and Solutions**, **External Web Tools**.
3. Add your `nodejs` and `node_modules\.bin` directories to the external tools configuration so that Visual Studio can find them. If you use the default Node.js paths, the configuration resembles the following image.



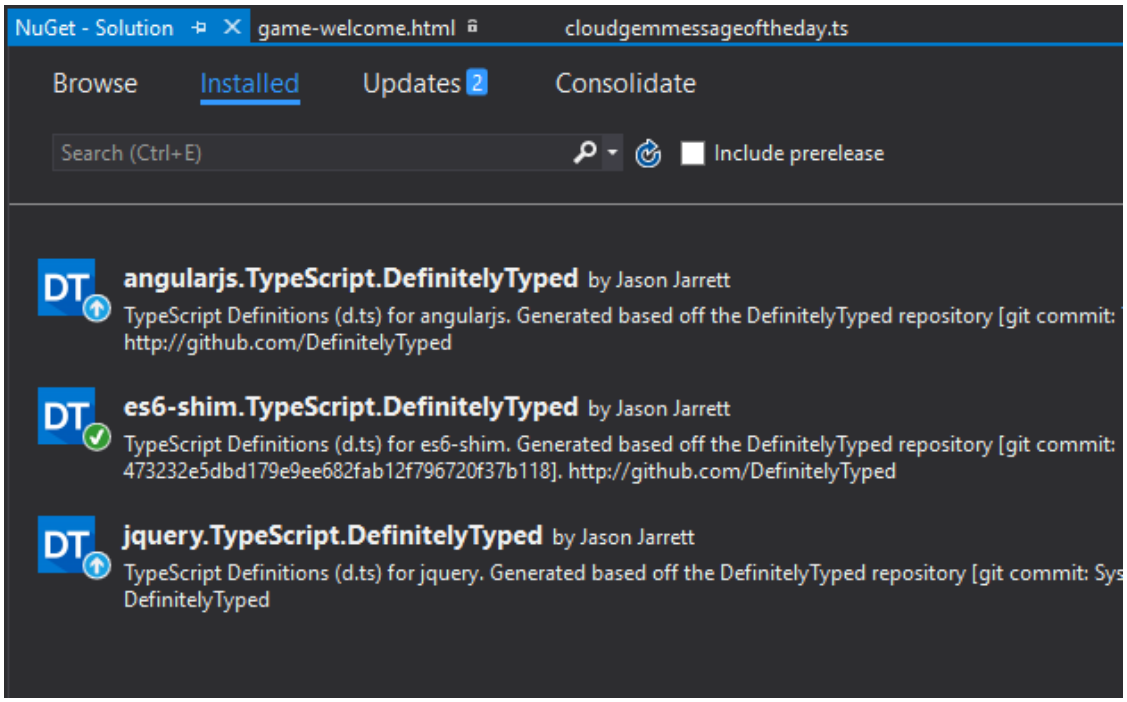
Tools and Extensions Setup

Next, you must install TypeScript-related extensions for Visual Studio.

To install TypeScript-related extensions

1. Click **Tools**, **NuGet Package Manager**, **Manage NuGet Packages for Solution**.
2. Install the following packages:

   - **angularjs.TypeScript.DefinitelyTyped**
   - **es6-shim.TypeScript.DefinitelyTyped**
   - **jquery.TypeScript.DefinitelyTyped**



After you install these packages, you can develop cloud gems in Visual Studio.

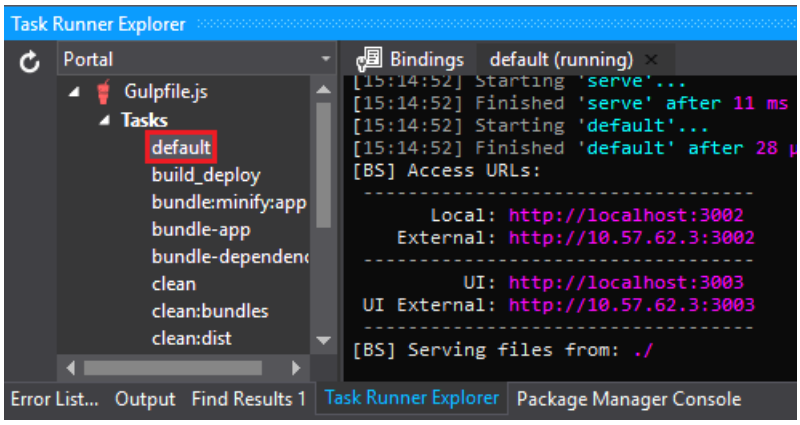
Note
If you are using Visual Studio 2015, it is possible to update npm packages or run a local server by right-clicking **package.json** and then choosing **Restore Packages**. However, you might find that using the `npm install` command from the *`<Lumberyard directory path>`*`\dev\Gems\CloudGemFramework\Website\CloudGemPortal\` directory works more consistently.

Using the Task Runner Explorer (Optional)

To run the gulp task, you can use Task Runner Explorer, a Visual Studio UI extension that executes any grunt or gulp task or target.

To use Task Runner Explorer

1. In Visual Studio, right-click **gulpfile.js**.
2. Choose **Task Runner Explorer**. The tool displays all the current gulp tasks.
3. To start a task, double-click the task.
4. To start a server, run **default** or **gulp serve** in the list of gulp **Tasks**.



Troubleshooting

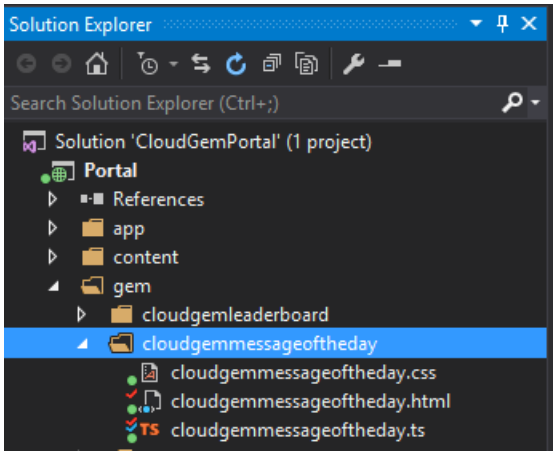
If opening a `.ts` file causes Visual Studio to crash, make sure that no other Visual Studio plugins are installed that open `.ts` files (QT Visual Studio and QT Linguist are two examples).

Developing Cloud Gems

Now you are ready to create cloud gems, which you can easily modify and redistribute to other developers. To develop a cloud gem, you add base files, configure a constructor map, and set up your TypeScript, `.html`, and `.css` files.

To add base files and configure a constructor map

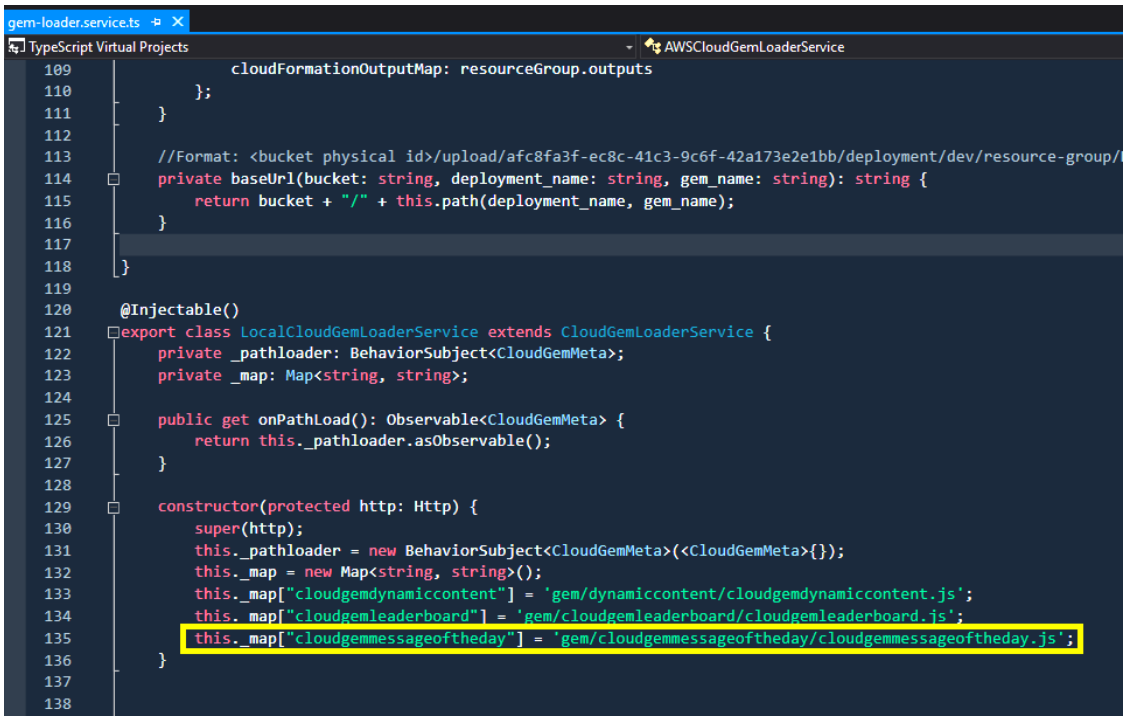
1. Create a subdirectory for your cloud gem. This tutorial uses the example of a Message of the Day Cloud Gem, whose directory structure resembles the following.

   The `cloudgemmessageoftheday` directory contains three files:

   - `cloudgemmessageoftheday.ts` – The TypeScript file that is the point of entry for your cloud gem. This links to the `.html` and `.css` files. This file is automatically transpiled into a ES5 JavaScript file with the same name. You will use this JavaScript file in step 3.
   - `cloudgemmessageoftheday.html` – The content for your gem.
   - `cloudgemmessageoftheday.css` – The styles for your gem.
2. Open the `CloudGemFramework\Website\CloudGemPortal\app\feature\gem\gem-loader.service.ts` file.

3. In the `gem-loader.service.ts` file, add your cloud gem's `.js` file to the `LocalCloudGemLoaderService` constructor map. The constructor map tells the Cloud Gem Portal that your new cloud gem exists.



   Make sure that you add the `.js` file and not the `.ts` file to the constructor map. Important differences between the two file types are explained in the next section.

At this point, your file system is ready. Now you can add skeleton code to each of the TypeScript and `.html` files.

Creating the TypeScript File

The Cloud Gem Portal is built on the Angular 2 version of AngularJS. The default language for Angular 2 is TypeScript. TypeScript is a typed superset of the JavaScript language. TypeScript files have a `.ts` extension.

To get started, create a new `.ts` file with your gem name (for example, `cloudgemmessageoftheday.ts`). The example that follows contains the starter code for the TypeScript file. This file is the entry point for the gem.

The following three classes form the core of a dynamic cloud gem:

1. Cloud gem web definition and view controller – class `Motd` in the example.
2. Cloud gem web view models – classes `ListModel`, `EditModel`, and `SerializationModel` in the example.
3. Cloud gem REST API Handler – class `MotdAPI` in the example.

```
import { Gemifiable, Measurable, Tackable, TackableStatus, TackableMeasure} from 'app/
feature/gem/gem-interfaces';
import { DynamicGem } from 'app/feature/gem/dynamic-gem.class'
import { ApiHandler } from 'app/feature/gem/rest/apihandler.class'
import { Observable } from 'rxjs/rx'
import { Http } from '@angular/http';
```

```
/**
 * Handy function for displaying toast messages
* Usage:
*    toastr.clear();
*    toastr.success("<message>");
*/
declare const toastr: any;

/*
*  Entry point for the cloud gem factory
*/
export function getInstance(context: any): Gemifiable {
    let gem = new Motd();
    gem.initialize(context);
    return gem;
}

/**
 * Dynamic cloud gem web definition and view controller
*/
export class Motd extends DynamicGem {
    /*
    * Inherited implementations of methods and field members
    */
    public identifier = "motd";

    public classType(): any {
        const clone = Motd
        return clone;
    }

    public initialize(context: any) {
        super.initialize(context);
        this.apiHandler = new
 MotdAPI(this.context.meta.cloudFormationOutputMap["ServiceUrl"].outputValue,
 this.context.provider.http);
        this.apiHandler.report(this.tackable.metric);
        this.apiHandler.assign(this.tackable.state);
    }

    public tackable: Tackable = <Tackable>{
        displayName: "Message of the day",
        iconSrc: "https://m.media-amazon.com/images/G/01/cloudcanvas/images/
message_of_the_day._V536715120_.png",
        cost: "Low",
        state: new TackableStatus(),
        metric: new TackableMeasure()
    };

    public get htmlTemplateUrl(): string {
        return this.context.provider.definition.isProd ?
 this.getSignedUrl(this.context.meta.key + "/cloudgemmessageoftheday.html") : "gem/
cloudgemmessageoftheday/cloudgemmessageoftheday.html";
    }
    public get styleTemplateUrl(): string {
        return this.context.provider.definition.isProd ?
 this.getSignedUrl(this.context.meta.key + "/cloudgemmessageoftheday.css") : "gem/
cloudgemmessageoftheday/cloudgemmessageoftheday.css";
    }
    /* end region */

    // define field members
    public apiHandler: MotdAPI;
    /* end region */
```

```
    //Required.  Firefox will generate an error if the constructor is not present without
 arguments
    constructor() {
        super();
    }
    /* end region */

    // initialize values
    ngOnInit() {
        // bind proper scopes for callbacks that have a this reference
        // example
        this.validate = this.validate.bind(this);
        this.onSubmit = this.onSubmit.bind(this);
        // end of bind scopes
    }
    /* end region */

    protected validate(model): boolean {
        return true;
    }

    protected onSubmit(model): void {
        let body = {
            message: model.message.value,
            priority: model.priority.value
        }
        this.apiHandler.post(body);
    }
}

/**
* View model defintions
*/
class ListModel {
    UniqueMsgID: string;
    message: string;
    priority: number;
}

class EditModel {
    UniqueMsgID: string;
    message: string;
    priority: number;
    start?: any;
    end?: any;
    hasStart?: boolean;
    hasEnd?: boolean;
    serialization: SerializationModel;
}

class SerializationModel {
    UniqueMsgID: string;
    message: string;
}

/**
 * API Handler for the Message of the Day RESTful services defined in the cloud gem
 swagger.json
*/
export class MotdAPI extends ApiHandler {
    constructor(serviceBaseURL: string, http: Http) {
        super(serviceBaseURL, http);
    }

    public report(metric: Measurable) {
        metric.name = "Total Active Messages";
```

```
        metric.value = "Loading...";
        super.get("admin/messages?count=500&filter=active&index=0").subscribe(response => {
            let obj = JSON.parse(response.body.text());
            metric.value = obj.result.list.length;
        });
    }

    public assign(tackableStatus: TackableStatus) {
        tackableStatus.label = "Loading";
        tackableStatus.styleType = "Loading";
        super.get("service/status").subscribe(response => {
            let obj = JSON.parse(response.body.text());
            tackableStatus.label = obj.result.status == "online" ? "Online" : "Offline";
            tackableStatus.styleType = obj.result.status == "online" ? "Enabled" :
 "Offline";
        }, err => {
            tackableStatus.label = "Offline";
            tackableStatus.styleType = "Offline";
        })
    }

    public delete(id: string): Observable<any> {
        return super.delete("admin/messages/" + id);
    }

    public put(id: string, body: any): Observable<any> {
        return super.put("admin/messages/" + id, body);
    }

    public post(body: any): Observable<any> {
        return super.post("admin/messages/", body);
    }

    public getMessages(filter: string, startIndex: number, count: number): Observable<any>
 {
        return super.get("admin/messages?count=" + count + "&filter=" +
            filter + "&index=" + startIndex);
    }
}
/* end region */
```

The following sections analyze the code in detail.

Import

The code imported in the example is the minimum required to start your gem. You can import more code if your gem requires it.

```
import { Gemifiable, Measurable, Tackable, TackableStatus, TackableMeasure} from 'app/
feature/gem/gem-interfaces';
import { DynamicGem } from 'app/feature/gem/dynamic-gem.class'
import { ApiHandler } from 'app/feature/gem/rest/apihandler.class'
import { Observable } from 'rxjs/rx'
import { Http } from '@angular/http';
```

Cloud Gem Factory Entry Point

The following required code provides the entry point for the cloud gem factory.

```
/*
*  Entry point for the cloud gem factory
*/
```

```
export function getInstance(context: any): Gemifiable {
    let gem = new Motd();
    gem.initialize(context);
    return gem;
}
```

The `getInstance()` function is located outside of the new class definition and returns an instance of the gem to the gem factory.

Export Class

The code defines a class and exports it so that Angular 2 can use it.

```
/**
 * Dynamic Cloud Gem web definition and view controller
*/
export class Motd extends DynamicGem {
    /*
    * Inherited implementations of methods and field members
    */
    public identifier = "motd";

    public classType(): any {
        const clone = Motd
        return clone;
    }

    public initialize(context: any) {
        super.initialize(context);
        this.apiHandler = new
 MotdAPI(this.context.meta.cloudFormationOutputMap["ServiceUrl"].outputValue,
 this.context.provider.http);
        this.apiHandler.report(this.tackable.metric);
        this.apiHandler.assign(this.tackable.state);
    }

    public tackable: Tackable = <Tackable>{
        displayName: "Message of the day",
        iconSrc: "https://m.media-amazon.com/images/G/01/cloudcanvas/images/
message_of_the_day._V536715120_.png",
        cost: "Low",
        state: new TackableStatus(),
        metric: new TackableMeasure()
    };

    public get htmlTemplateUrl(): string {
        return this.context.provider.definition.isProd ?
 this.getSignedUrl(this.context.meta.key + "/cloudgemmessageoftheday.html") : "gem/
cloudgemmessageoftheday/cloudgemmessageoftheday.html";
    }
    public get styleTemplateUrl(): string {
        return this.context.provider.definition.isProd ?
 this.getSignedUrl(this.context.meta.key + "/cloudgemmessageoftheday.css") : "gem/
cloudgemmessageoftheday/cloudgemmessageoftheday.css";
    }
    /* end region */

    // define field members
    public apiHandler: MotdAPI;
    /* end region */

    //Required.  Firefox will generate an error if the constructor is not present without
 arguments
    constructor() {
```

```
        super();
    }
    /* end region */

    // initialize values
    ngOnInit() {
        // bind proper scopes for callbacks that have a this reference
        // example
        this.validate = this.validate.bind(this);
        this.onSubmit = this.onSubmit.bind(this);
        // end of bind scopes
    }
    /* end region */

    protected validate(model): boolean {
        return true;
    }

    protected onSubmit(model): void {
        let body = {
            message: model.message.value,
            priority: model.priority.value
        }
        this.apiHandler.post(body);
    }
}
```

This export technique is a common convention in Angular 2. For more information, see the Angular QuickStart guide.

Initial Tasks

Following the class definition, the example performs the following tasks:

- `public identifier = "motd";` – Defines a public text identifier for your gem.
- `public classType(): any {...}` – Uses the exported class name (`Motd` in this case) to create a clone of the class and return it.
- The code registers the gem's context with the parent. It also registers a reporting function for the gem.

```
   public initialize(context: any) {
       super.initialize(context);
       this.apiHandler = new
 MotdAPI(this.context.meta.cloudFormationOutputMap["ServiceUrl"].outputValue,
 this.context.provider.http);
       this.apiHandler.report(this.tackable.metric);
       this.apiHandler.assign(this.tackable.state);
   }
```

The Tackable Object

The code creates a `Tackable` object.

```
public tackable: Tackable = <Tackable>{
        displayName: "Message of the day",
        iconSrc: "https://m.media-amazon.com/images/G/01/cloudcanvas/images/
message_of_the_day._V536715120_.png",
        cost: "Low",
        state: new TackableStatus(),
        metric: new TackableMeasure()
    };
```

The `Tackable` object shows the user an overview of the cloud gem on the Cloud Gems page of the Cloud Gem Portal. The `Tackable` object has the following attributes.

- **`displayName`** – The name displayed on the gem index page.
- **`iconSrc`** – The source of the display icon for your cloud gem. If not specified, a default image is used.
- **`cost`** – The expected cost of AWS resources. Options are `Low`, `Medium`, `High`, or `None`.
- **`state`** – The object that stores the state of the gem.
- **`metric`** – The object that stores the metrics of your gem.

URL Presigning

After the code instantiates the `Tackable` object, it declares the `get htmlTemplateUrl()` and `get styleTemplateUrl()` functions.

```
public get htmlTemplateUrl(): string {
    return this.context.provider.definition.isProd ?
 this.getSignedUrl(this.context.meta.key + "/cloudgemmessageoftheday.html") : "gem/
cloudgemmessageoftheday/cloudgemmessageoftheday.html";
}
public get styleTemplateUrl(): string {
    return this.context.provider.definition.isProd ?
 this.getSignedUrl(this.context.meta.key + "/cloudgemmessageoftheday.css") : "gem/
cloudgemmessageoftheday/cloudgemmessageoftheday.css";
}
```

These two functions are required because of the way in which Lumberyard presigns URLs. Presigning the URL means including the encrypted AWS secret key and password with each request. This validates that the request is from an authorized source.

When the Cloud Gem Portal is hosted in Amazon S3 (and not on your desktop), it presigns the URLs when it tries to retrieve the `.html` and `.css` files. The presigning is important to avoid access denied exceptions.

When the Cloud Gem Portal is hosted on your computer for development purposes (not in the cloud), it uses your local `.html` and `.css` files without any presigning.

The super() Function

The constructor calls its parent constructor through the `super()` function. This call compensates for a bug in some current versions of Firefox. If you omit the call to `super();`, your gem might not work in Firefox.

```
//Required.  Firefox will generate an error if the constructor is not present without
 arguments.
    constructor() {
        super();
    }
```

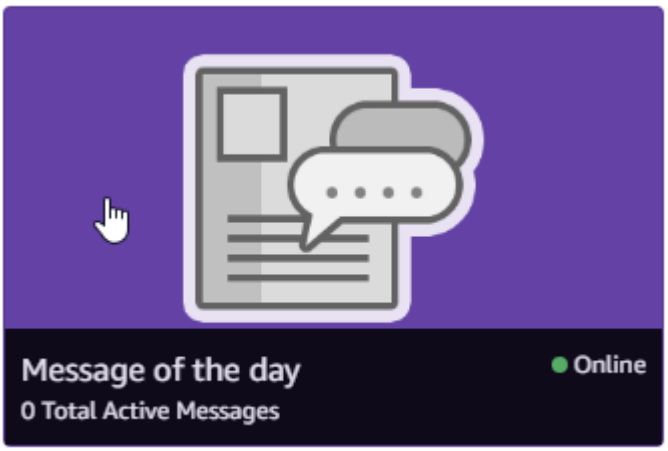
The report Function

The required `report` function is located in the API Handler class for the cloud gem.

```
public report(metric: Measurable) {
        metric.name = "Total Active Messages";
        metric.value = "Loading...";
```

```
        super.get("admin/messages?count=500&filter=active&index=0").subscribe(response => {
            let obj = JSON.parse(response.body.text());
            metric.value = obj.result.list.length;
        });
    }
```

The `report` function determines the messaging on the Cloud Gems index page for the metric. For message of the day this is the **0 Total Active Messages** shown in the following image.



This metric provides a convenient snapshot of the status of the gem when you place your mouse pointer over the gem's thumbnail. The metric name is the text after the numerical value in the thumbnail. In the example, it is set to "Total Active Messages".

```
metric.name = "Total Active Messages";
```

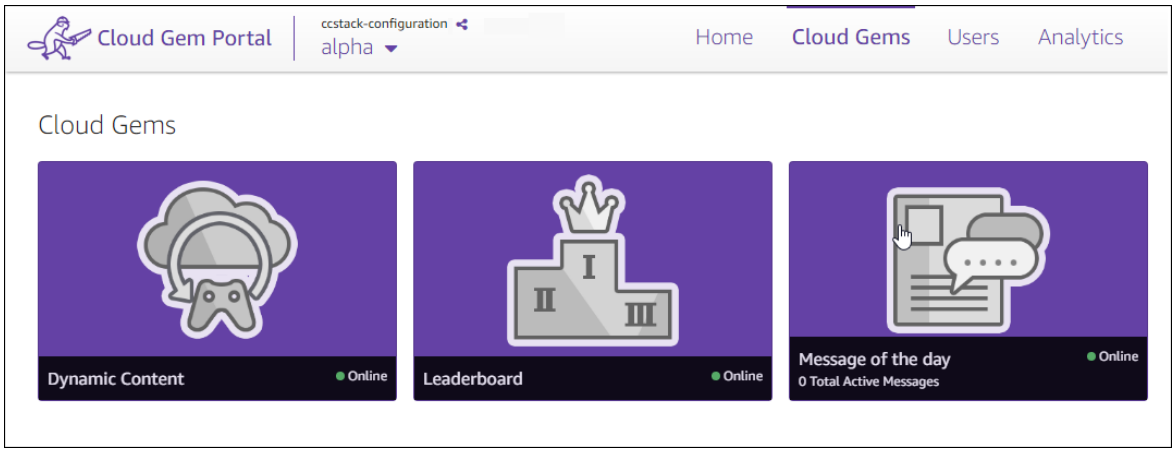
The metric value is the number of messages. The example sets the initial value to "Loading" for when the data is not yet available.

```
metric.value = "Loading...";
```

After you set these initial values, you can send a request to AWS Lambda to retrieve the actual metric value. The message of the day example gets the first 10 messages, parses the result to an object, and uses the length of that object as its metric value.

Testing the Results

At this point your setup of the `.ts` file is complete. If you are running your local server with `gulp serve` or `gulp serve-watch`, navigate to `localhost:3000`. Your new gem should appear in the **Cloud Gems** section.



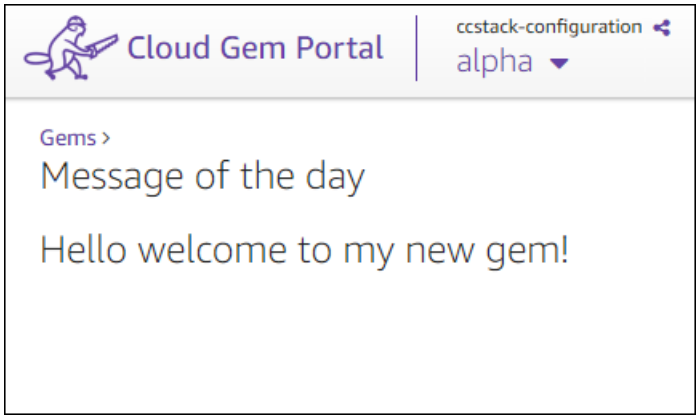
If you click the gem thumbnail, the page that appears is blank, but that's expected at this point.

Creating an HTML File

You can verify that your gem is working by adding a header like the following to your `.html` file.

```
<h1> Hello welcome to my new gem! </h1>
```

If you reload the Cloud Gem Portal page and navigate to your gem, your new text appears.



Creating Dynamic Content

To make your content more dynamic, you can use Angular 2 directives in your `.html` and `.ts` files. To see this in action, modify the `.html` and `.ts` files to create a test button on the Message of the Day Cloud Gem, as in the following example.

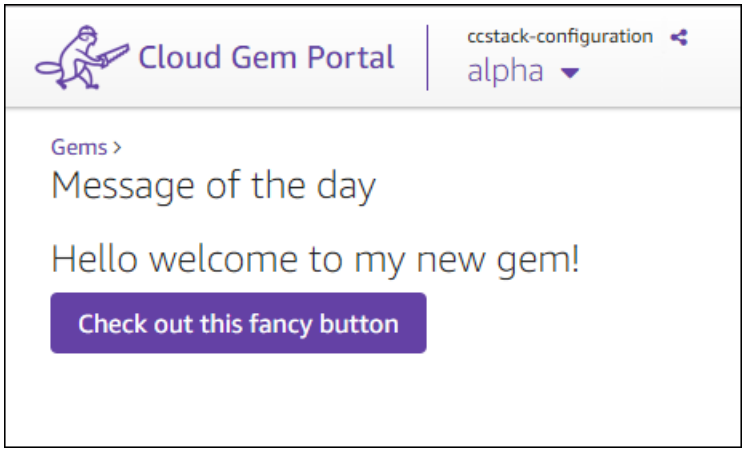
In the `.html` file, add the following markup.

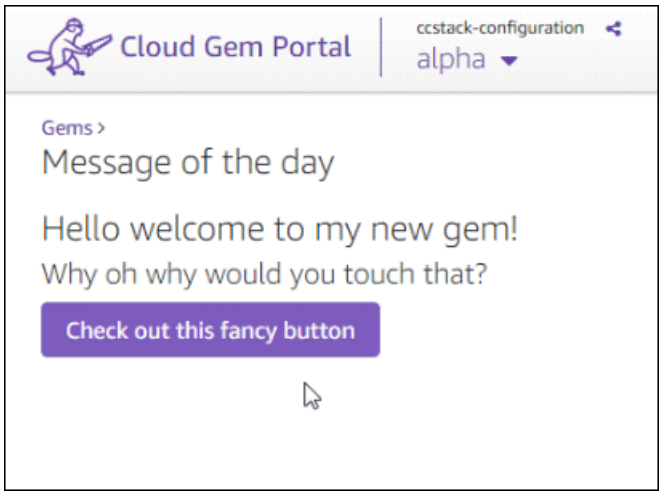
```
<h1> Hello welcome to my new gem! </h1>
<h2 *ngIf="testButton"> Why oh why would you touch that? </h2>

<form>
    <button class="btn l-primary" (click)="testButton = !testButton">
        Check out this fancy button
    </button>
</form>
```

Modify your `.ts` file as in the following example.

```
export class Motd extends DynamicGem {
    ...
    private testButton: boolean;

      ngOnInit() {
          this.testButton = false;
      }
}
```

The following images show what happens when the button is clicked.





Creating a CSS File

You can use standard CSS to style your cloud gems. For convenience, Lumberyard's cloud gems use the Bootstrap 4 CSS framework. When you use standard HTML tags, the your tags automatically pick up CSS changes that provide a professional looking, consistent user interface.

Deploying Your Changes

After you are satisfied with your gem, you can upload it to AWS to verify that it is working. Deploying your changes includes three steps: packaging your changes, copying your files to a local AWS directory, and uploading your cloud gem and Cloud Gem Portal files to AWS.

1. Packaging Your Changes

To package your changes, perform one of the following tasks:

- In Visual Studio, run the gulp task `build_deploy`.
- From a command line prompt, run `gulp build_deploy`.

Doing so packages the three Cloud Gem Portal files `index.html`, `bundles\app.bundle.js`, and `bundles\dependencies.bundles.js` and copies them to the `lumberyard\Gems\CloudGemFramework\AWS\www` publishing directory.

2. Copying Your Files to a Local AWS Directory

Until now, you've been working in the `\dev\Gems\CloudGemFramework` directory. To have your cloud gem files associated with the directory for the gem, and not the framework, you must copy these files to a specific local AWS directory.

To copy your files to the `AWS\cgp-resource-code` directory

1. From your *<Lumberyard>*`\dev\Gems\CloudGemFramework\Website\CloudGemPortal\cloudgemmessageoftheday` directory, copy your `.js`, `.html`, and `.css` files (you should have only 3 files total).
2. If the *<Lumberyard>*`\dev\Gems\CloudGemMessageOfTheDay\AWS\cgp-resource-code` directory doesn't exist, create it.
3. Change to the *<Lumberyard>*`\dev\Gems\CloudGemMessageOfTheDay\AWS\cgp-resource-code` directory, and then paste the files there.

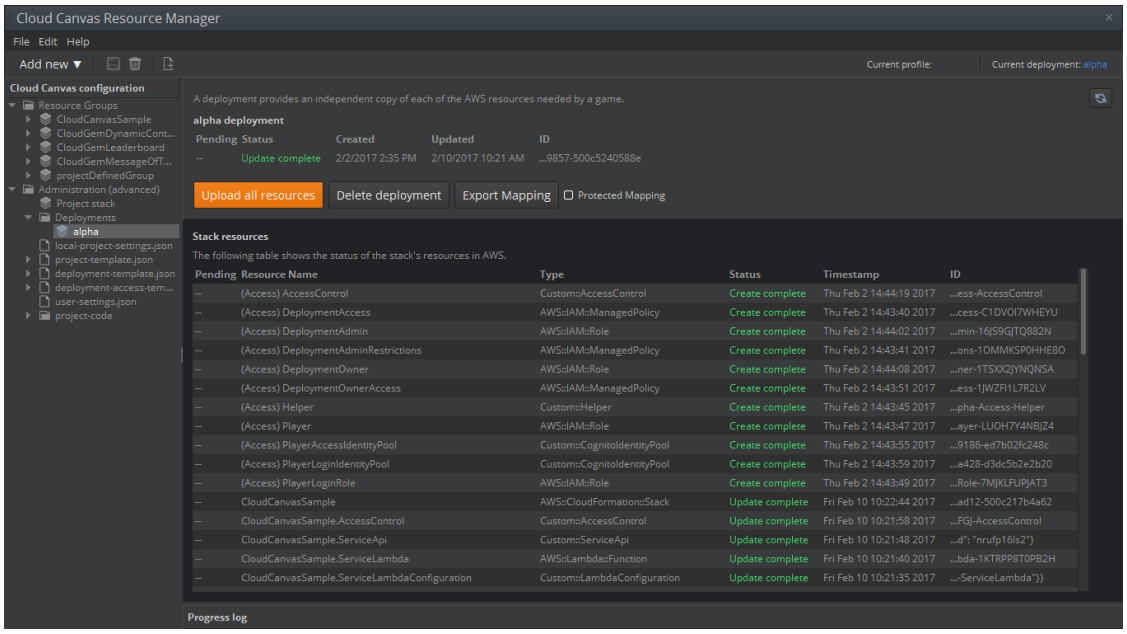
The files in your `AWS\cgp-resource-code` directory are uploaded to Amazon S3 when you update your gem.

3. Uploading the Cloud Gem With Your Cloud Gem Portal Content

Before you upload your cloud gem to AWS, make sure you have you added your new cloud gem to your project with the Project Configurator. To perform the upload, you can use either Lumberyard Editor or a command-line prompt.

To use Lumberyard Editor to upload your cloud gem and Cloud Gem Portal content to AWS

1. In Lumberyard Editor, click **AWS**, **Cloud Canvas**, **Resource Manager**.
2. Under **Administration**, **Deployments**, click your active deployment. If you don't have a deployment, create one first.
3. Click **Upload All Resources**.



4. After the upload has completed, click **AWS**, **Open Cloud Gem Portal** in Lumberyard Editor to access the Cloud Gem Portal that you uploaded. Your changes should be visible on the live website.

To use a command line prompt to upload your cloud gem and Cloud Gem Portal content to AWS

1. To update all of the files in the `Gems\cgp-resource-code` directory in the cloud, type the following command:

```
lmbr_aws cloud-gem-framework upload-portal
```

2. To update your Amazon S3 Cloud Gem Portal bucket with the latest files from your `lumberyard\Gems\CloudGemFramework\AWS\www` directory, type the following command.

```
lmbr_aws cloud-gem-framework upload-portal --project
```

Summary

In this tutorial, you accomplished the following tasks:

1. Added a gem to the `CloudGemPortal` framework.
2. Used TypeScript and Angular 2 to create three required `.js`, `.html`, and `.css` files.
3. Used Lumberyard Editor to upload your changes to Amazon S3.
4. Viewed your changes in production.

Making HTTP Requests Using the Cloud Gem Framework

The Cloud Gem Framework and this documentation are in preview release and are subject to change.

The Cloud Gem Framework Gem provides C++ classes and EBus interfaces to execute HTTP requests using the `AZ::Job` system. Your game client can use this feature to make HTTP requests for data from a public API such as Twitter or from a custom API. For example, your game could make HTTP requests to Twitter to see who is tweeting about your game.

To enable your game code to make HTTP requests

1. In Lumberyard Project Configurator, enable the **Cloud Canvas Common** and **Cloud Gem Framework** gems for your project.
2. In Lumberyard Editor, in **Entity Inspector**, click **Add Component**.
3. From the **Cloud Gem Framework** section, add the **HttpClientComponent** to an entity in your scene.
4. To make HTTP requests from your game code, perform one of the following steps:

   - From a Lua script component attached to your entity, add code based on the following example.

   ```
   self.requestSender = HttpClientComponentRequestBusSender(self.entityId)
   local url = "https://my.url.com"
   local http_method = "GET"
   local json_body = "{}"
   HttpClientComponentRequestBus.Event.MakeHttpRequest(self.entityId, url, http_method,
    json_body)
   ```

   - From C++, use Lumberyard's Event Bus (EBus) (p. 543), as in the following example.

   ```
   AZStd::string url = "https://my.url.com"
   AZStd::string httpMethod = "GET"
   AZStd::string jsonBody= "{}"
   EBUS_EVENT(HttpClientComponentRequestBus, MakeHttpRequest, url, httpMethod,
    jsonBody);
   ```

   - From C++, use `HttpRequestJob`, as in the following example.

   ```
   AZStd::string url = "https://my.url.com"
   AZStd::string httpMethod = "GET"
   AZStd::string jsonBody= "{}"

   auto job = aznew HttpRequestJob(true, ServiceJob::GetDefaultConfig(),
               [this](int responseCode, AZStd::string content)
               {
                   // handle success
               },
               [this](int responseCode)
               {
                   // handle failure
               }
           );
   job->SetUrl(url.c_str());
   job->SetHttpMethod(httpMethod );
   job->SetJsonBody(jsonBody.c_str());
   job->Start();
   ```

Getting HTTP Responses Using Script

To get responses from a HTTP request, your script class needs an `HttpClientComponentNotificationBusHandler` as in the following example.

```
function httpClientUsageExample:OnActivate()
    self.notificationHandler = HttpClientComponentNotificationBus.Connect(self,
 self.entityId);
```

```
end
```

Next, your script class must implement the `HttpClientComponentNotificationBusHandler` functions `OnHttpRequestSuccess and OnHttpRequestFailure` as in the following example.

```
function myscript:OnHttpRequestSuccess(responseCode, responseBody)
    Debug.Log("HTTP RESPONSE -- " .. responseCode);
    Debug.Log("HTTP BODY -- " .. responseBody);
end

function myscript:OnHttpRequestFailure(errorCode)
    Debug.Log("HTTP Error-- " .. errorCode);
end
```

Getting HTTP Responses Using C++

To get the notifications in C++, you must create a component that inherits from `HttpClientComponentNotificationBus::Handler`. This class must implement `OnHttpRequestSuccess` and `OnHttpRequestFailure` and should be placed on the same entity as the `HttpClientComponent` in your level.

Cloud Gem Framework Resource Manager Hooks

The Cloud Canvas Resource Manager ensures that the AWS resources that the game uses match the definitions of those resources in the game's source code. However, you might need to add functionality to these processes. For example, for the Dynamic Content Cloud Gem, you must upload any default packages for the project immediately after you create a new deployment.

The resource manager provides some Python hook modules for cloud gems. The code for these modules is located in the `resource-manager-code` directory found in the cloud gem's AWS directory (for example, `\dev\Gems\CloudGemDynamicContent\AWS\resource-manager-code`).

The following modules are supported:

- `update.py` – Contains functions that are called before and after stack update operations. For more information, see Update Hook Functions (p. 347).
- `command.py` – Contains functions that add new CLI and GUI support to the resource manager. For more information, see Command Hook Functions (p. 348).

Hook Function Parameters

All hook functions are called with the following:

- A parameter that represents the hook function
- Some keyword arguments that are specific to each hook function

The parameter that represents the hook function is a `HookModule` object (p. 339). The most important property provided by the `HookModule` object is `context`, which is a Context Object (p. 331). The `Context` object gives your hook function access to the Cloud Canvas Resource Manager configuration data for the project.

Futureproofing Your Functions

When you write a hook function, always add Python's `**kwargs` construct as the last argument. That way your function can gracefully accept (and ignore) arguments that future versions of resource manager might add after your function is written.

For example, suppose resource manager passes `arg_a` and `arg_b` arguments to your hook function. You declare the hook function as follows:

```
def a_hook(hook, arg_a, arg_b, **kwargs)
```

Later, when resource manager adds an `arg_c` argument, your hook function does not break. Your function still accepts `arg_a` and `arg_b` but can ignore `arg_c` because `arg_c` was passed through the `kwargs` parameter.

Cloud Canvas Resource Manager Objects

Cloud Canvas Resource Manager includes Python objects that you can use to access project configuration data and perform various project operations.

> **Note**
> The source code for these objects is located in the Lumberyard `dev\Tools\lmbr_aws \AWSResourceManager` directory.

Context Object

The `Context` object has properties that provide access to project configuration and that perform operations such as adding resource definitions to a resource group template.

You can get a `Context` object instance from the `context` property of the `HookModule` object instance passed to hook functions.

Name	Description
`aws`	An `AWSContext` object.
`config`	A `ConfigContext` object.
`gem`	A `GemContext` object.
`hooks`	A `HookContext` object.
`resource_groups`	A `ResourceGroupContext` object.
`stack`	A `StackContext` object.
`view`	A `ViewContext` object.

Other properties or functions of this object are internal to resource manager and should not be used.

AWSContext Object

The `AWSContext` object has helper functions and properties related to AWS clients and credentials.

You can get an `AWSContext` object instance from the `aws` property of a `Context` object.

Name	Description
`assume_role(logical_role_id, deployment_name)`	Assumes a role that uses the user's current AWS credentials. New clients that were created from the `client` function use the temporary credentials that were created by assuming the role. These credentials have a 3600 second (5 minute) lifetime.

Name	Description
	The `logical_role_id` parameter can be the logical name of a `AWS::IAM::Role` resource defined in the `project-template.json` or `deployment-access-template.json` files. In the case of a role that is defined in the `deployment-access-template.json` file, the `deployment_name` identifies the actual role that will be assumed.
`client(service_name, region=None, use_role=True)`	Creates a Python AWS API (boto3) client for the specified AWS service. If `use_role` is `True`, the client uses the credentials that are created when `assume_role` was called. The returned object is a wrapper around the actual boto3 client object. The wrapper logs all API call attempts and responses when the `--verbose` option is passed to `lmbr_aws`. This aids in debugging. The client wrapper also performs retries with exponential back off.
`get_credentials_file_path()`	Returns the full path to the user's `.aws/credentials` file.
`get_default_profile()`	Gets the AWS credentials profile name that is used by default when clients are created.

`get_temporary_credentials(logical_role_id, deployment_name, duration_seconds)`	Creates temporary credentials by using the user's current credentials to assume a role.
	
	The `logical_role_id` parameter can be the logical name of a `AWS::IAM::Role` resource defined in the `project-template.json` or `deployment-access-template.json` files. In the case of a role defined in the `deployment-access-template.json` file, the `deployment_name` identifies the actual role that will be assumed.
	
	The return value is a Python `dict` with the following properties:
	
	`AccessKeyId`
	
	`SecretAccessKey`
	
	`SessionToken`

`load_credentials()`	Returns an `AWSCredentials` object that contains the data read from the user's `.aws/credentials` file.
`profile_exists(profile_name)`	Determines whether the specified profile exists in the user's `.aws/credentials` file.
`save_credentials(credentials)`	Saves a modified `AWSCredentials` object into the user's `.aws/credentials` file.

Name	Description
session	The boto3 session object that is used by client().
set_default_profile	Sets the AWS credentials profile that is used when clients are created. This resets the session property.

Other properties or functions of this object are internal to resource manager and should not be used.

AWSCredentials Object

The AWSCredentials object contains AWS credential information that is read from the user's .aws/credentials file. This is essentially a wrapper around a Python ConfigParser object. The AWSCredentials object handles the AWS credential file's use of the default section, which conflicts with how ConfigParser handles defaults.

To get an AWSCredentials object instance, use the load_credentials function on an AWSContext object.

Name	Description
add_section(section_name)	Adds a section.
get(section_name, option_name)	Gets the value of an option in a section.
has_option(section_name, option_name)	Determines whether an option exists in a section.
has_section(section_name)	Determines whether a section exists.
items(section_name)	Returns a list of name value pairs of the options in a section.
options(section_name)	Returns a list of options in a section.
read(path)	Reads credentials from a file with the specified path.
remove_option(section_name, option_name)	Removes an option from a section.
remove_section(section_name)	Removes a section.
sections()	Returns a list of section names. Each profile is represented by a section.
set(section_name, option_name, value)	Sets an option value.
write(path)	Writes credentials to a file with the specified path.

Other properties or functions of this object are internal to resource manager and should not be used.

ConfigContext Object

The ConfigContext object has properties and functions that provide access to project and deployment configuration data.

You can get a ConfigContext object instance from the config property of a Context object.

Name	Description
`aws_directory_path`	The full path to the currently enabled game project (as determined by the `bootstrap.cfg` file or options passed to `lmbr_aws`).
`base_resource_group_directory_path`	The directory where resource groups that are defined by the project (as opposed to those defined by gems) are located. This is typically the `resource-group` subdirectory that is found in the directory identified by `aws_directory_path`.
`clear_user_default_deployment()`	Clears the user's default deployment setting.
`clear_user_default_profile()`	Clears the user's default AWS credentials profile name. This does not update the user's `.aws/credentials` file. The name of the default profile is saved in the Cloud Canvas Resource Manager's user settings file.
`configuration_bucket_name`	The name of the project's `Configuration` Amazon S3 bucket.

`copy_default_lambda_function_content`
 	
`(destination_path)`
	Copies the default Lambda function content to the specified location.

`copy_default_project_content(destination_path)`	Copies the content from the default project AWS directory to the specified location.
`copy_default_resource_group_content(destination_path)`	Copies the content from the default resource group to the specified location.
`copy_example_resource_group_content(destination_path)`	Copies the content from the example resource group to the specified location.
`default_deployment`	The name of the deployment that is the default for the current user if a default name has been specified. If a default has been specified, the value is that of `user_default_deployment`; otherwise, it is that of `project_default_deployment`.
`deployment_access_template`	A Python object that contains the content of the deployment access template file.
`deployment_access_template_path`	The full path to the project's `deployment-access-template.json` file.
`deployment_names`	A list that contains the names of the project's deployments.
`deployment_stack_exists(deployment_name)`	Determines whether a deployment stack exists.
`finalize_deployment_stack_ids(deployment_name)`	Commits the pending deployment and deployment access stack IDs as the deployment's actual stack IDs. See `set_pending_deployment_stack_id` and `set_pending_deployment_access_stack_id`.

Name	Description
`game_directory_path`	The full path to the currently enabled game project (as determined by the `bootstrap.cfg` file or the options that are passed to `lmbr_aws`).
`gem_directory_path`	The full path to the directory where gems are defined. This is the `Gems` subdirectory of the directory identified by `root_directory_path`.
`get_default_deployment_stack_name(deployment_name)`	Gets the default stack name for a deployment with the specified name.
`get_deployment_access_stack_id(deployment_name, optional=False)`	Gets the ARN of a deployment access stack. If the specified deployment stack doesn't exist, raises a `HandledError` or returns `None` if `optional` is `True`.
`get_deployment_stack_id(deployment_name, optional=False)`	Gets the ARN of a deployment stack. If the specified deployment stack doesn't exist, raises a `HandledError` or returns `None` if `optional` is `True`.
`get_game_directory_name()`	Gets the name of the game directory from the `bootstrap.cfg` file.
`get_project_stack_name()`	Gets the name of the project stack.
`get_protected_depolyment_names()`	Returns a list of the protected deployment names. For more information, see `protect_deployment` and `unprotected_deployment`.
`get_resource_group_stack_id(deployment_name, resource_group_name, optional=False)`	Gets the ARN of a resource group stack. If the specified resource group stack doesn't exist, raises a `HandledError` or returns `None` if `optional` is `True`.
`local_project_settings_path`	Full path to the file that stores project's settings that are kept in a source-controlled file (as opposed to project settings stored in AWS). Typically, this is the `local-project-settings.json` file in the directory that is identified by `aws_directory_path`.
`no_prompt`	When `True`, specifies that the `--no-prompt` option was passed to `lmbr_aws`. This option is typically used when calling `lmbr_aws` from scripts when there is no user to respond to prompts. When this option is set, the hook function should raise an error instead of prompting the user.
`project_code_path`	The path to the project's `project-code` directory.
`project_default_deployment`	The name of the deployment that has been designated as the default for the project, if any. See also `user_default_deployment` and `default_deployment`.
`project_initialized`	`True` if the project has been initialized and `project_stack_id` has a value; `False` otherwise.

Name	Description
`project_resource_handler_id`	The ARN of the `ProjectResourceHandler` Lambda function. Cloud Canvas–defined resource templates require that this value be provided as the `ProjectResourceHandler` parameter value.
`project_settings`	A `ProjectSettings` object that contains project settings that are loaded from AWS (as opposed to the content of `local_project_settings`).
`project_stack_id`	The project stack ID, if the project stack has been created.
`project_template`	A Python object that contains the content of the project template file.
`project_template_path`	The full path to the project's `project-template.json` file.
`protect_deployment(deployment_name)`	Marks a deployment as protected. It requires special confirmation before you can connect debug builds of the game to protected deployments. See also `unprotect_deployment` and `get_protected_deployment_names`.
`refresh_user_settings()`	A function that reloads the `user_settings` object from the user settings file.
`release_deployment`	The name of the deployment that has been designed as the release deployment of the project.
`remove_deployment(deployment_name)`	Removes a deployment from the project's configuration.
`root_directory_path`	The full path to the Lumberyard root directory (the `\dev` directory), which contains project-specific directories.
`save_deployment_access_template()`	Saves the current `deployment_access_template` object to the project's `deployment-access-template.json` file.
`save_deployment_template()`	Saves the current `deployment_template` object to the project's `deployment-template.json` file.
`save_project_template()`	A function that saves the current `project_template` object to the project's `project-template.json` file.
`set_pending_deployment_access_stack_id(deployment_name, deployment_access_stack_id)`	Sets the ID for a deployment access stack that is being created. For more information, see `finalize_deployment_stack_ids` and `set_pending_deployment_stack_id`.
`set_pending_deployment_stack_id(deployment_name, deployment_stack_id)`	Sets the ID for a deployment stack that is being created. For more information, see `finalize_deployment_stack_ids` and `set_pending_deployment_access_stack_id`.

Name	Description
`set_project_default_deployment(deployment_name)`	Sets the default deployment for the project. Overridden by the value for user's default deployment if one is specified.
`set_release_deployment(deployment_name)`	Sets the release deployment for the project.
`set_user_default_deployment(deployment_name)`	Sets the default deployment for the user. After it is set, overrides the project default deployment.
`set_user_default_profile(profile_name)`	Sets the user's default AWS credentials profile. This does not update the user's `.aws/credentials` file; the name of the default profile is saved in the Cloud Canvas Resource Manager's user settings file.
`unprotect_deployment(deployment_name)`	Reverses the action of `protect_deployment`.
`user_default_deployment`	The name of the deployment that has been designated as the default by the current user. For more information, see `project_default_deployment` and `default_deployment`.
`user_default_profile`	The user's default AWS profile name as read from user settings.
`user_directory_path`	The full path to the user directory in the Lumberyard cache directory. This is where user-specific configuration is stored.
`user_settings`	A Python object that contains the contents of the user settings file.
`user_settings_path`	The full path to the JSON format text file that resource manager uses to store user-specific configuration. This configuration includes default deployment and AWS credentials profile names. This path is typically the `user-settings.json` file in the directory specified by `user_directory_path`.

Other properties or functions of this object are internal to resource manager and should not be used.

Gem Object

The `Gem` object provides access to gem-specific configuration data. To get `Gem` object instances, use the `enabled_gems` property of a `GemContext` object.

Name	Description
`aws_directory`	The full path to the gem's AWS directory.
`aws_directory_exists`	True if the gem's AWS directory exists.
`display_name`	The gem display name.

Name	Description
`file_object`	A Python object that contains the contents of the gem's `gem.json` file.
`name`	The name of the gem.
`project_resources`	Project resource definitions that are inserted into the project's stack template before the template is uploaded. These definitions are read from a `project-template.json` file in the gem's AWS directory.
`resource_group_name`	The gem's resource group name, if any. A gem has a resource group name only if it has a `resource-template.json` file.
`root_directory`	The full path to the gem's directory.
`uuid`	The unique ID of the gem.

Other properties or functions of this object are internal to the resource manager and should not be used.

GemContext Object

The properties and methods of the `GemContext` object provide access to the project's gem configuration.

You can get a `GemContext` object instance using the `gems` property of a `Context` object.

Name	Description
`enabled_gems`	A list containing a gem object for each gem that is enabled for the project.
`get_by_resource_group_name(resource_group_name)`	Returns the gem object associated with the specified resource group, or `None` if the resource group doesn't exist or isn't associated with a gem. By default the resource group name is the same as the gem name, but a **ResoruceGroupAlias** property can be put in the gem's **gem.json** file to override this default.

Other properties or functions of this object are internal to resource manager and should not be used.

HandledError Object

A `HandledError` object is Python `Exception` object. You can use it in a hook function to cause an expected error to be displayed to the user without producing a stack trace. In general, the resource manager considers other kinds of exceptions to be unexpected errors and might show additional debug information that users should not see for expected errors.

The **HandledError** class is defined in the `AWSResourceManager.errors` module (`\dev\Tools\lmbr_aws\AWSResourceManager\errors.py`). To create an instance, see the following example.

```
from AWSResourceManager.errors import HandledError

def my_function():
    if something_is_wrong:
```

```
        raise HandledError('Something is wrong.')
```

HookModule Object

The first argument (and the only positional argument) passed to a hook function is a `HookModule` object that represents the hook module itself. This object's properties give you access to project configuration data.

Name	Description
`context`	Provides access to a `Context` object.
`hook_name`	The name of the cloud gem that defines the hook module.
`hook_path`	The full path to the cloud gem's AWS directory where the hook module is defined.

Other properties or functions of this object are internal to resource manager and should not be used.

HookContext Object

The `HookContext` object provides functionality for working with Cloud Canvas Resource Manager hooks.

You can get a `HookContext` object instance from the `hooks` property of the `Context` object.

Name	Description
`call_module_handlers(module_name, handler_name, args=(), kwargs={}, deprecated=False)`	Calls the hook function identified by `handler_name` in the module identified by `module_name`. Use the `kwargs` parameter to pass keyword arguments to the handler function. The `args` parameter can only be used if `deprecated` is `True`. If `deprecated` is `True`, a warning regarding the deprecated function is displayed to the user.
`call_single_module_handler(module_name, handler_name, defining_module_name, args=(), kwargs={}, deprecated=False)`	Calls a hook function defined by the gem or resource group that is specified by the `defining_module_name` parameter. For a description of the other parameters, see `call_module_handlers`.

Other properties or functions of this object are internal to resource manager and should not be used.

ProjectSettings Object

The `ProjectSettings` object manages the project configuration data that is stored in the `project-settings.json` file in the project's Amazon S3 `Configuration` bucket.

Name	Description
`get_deployment(deployment_name)`	Returns the configuration dictionary for the specified deployment. The dictionary contains the following keys: `DeploymentStackId` – The ARN of the deployment stack.

Name	Description
	`DeploymentAccessStackId` – The ARN of the deployment access stack. `resource-group` – The resource group configuration dictionary for the deployment. For a description of this value, see `get_resource_group_settings`.
`get_deployments()`	Returns the entire configuration dictionary for deployments. The deployment name is the key. The special deployment name `*`, which contains settings for all deployments, can be present. For a description of the value dictionary, see `get_deployment`.
`get_project_default_deployment()`	Returns the default deployment for the project.
`get_release_deployment()`	Returns the release deployment for the project.
`get_resource_group_settings(deployment_name)`	Returns a dictionary containing the resource group configuration for the specified deployment. The resource group name is the key. The special resource group name `*`, which contains settings for all resource groups, can be present. Each value is a dictionary with the following key: `parameter` – A dictionary containing parameter values for the resource group template.
`remove_deployment(deployment_name)`	Removes the configuration for the specified deployment name. To commit this change, call `save`.
`remove_project_default_deployment()`	Clears the default deployment for the project. To commit this change, call `save`.
`remove_release_deployment()`	Clears the release deployment for the project. To commit this change, call `save`.
`save()`	Saves the current configuration to the Amazon S3 bucket.
`set_project_default_deployment(deployment_name)`	Sets the default deployment for the project. To commit this change, call `save`.
`set_release_deployment(deployment_name)`	Sets the release deployment for the project. To commit this change, call `save`.

Other properties or functions of this object are internal to resource manager and should not be used.

ResourceGroup Object

The `ResourceGroup` object encapsulates a resource group's configuration. If multiple resource group stacks for a given resource group exist, each resource group stack is associated with a single deployment.

To get `ResourceGroup` object instances, use a `ResourceGroupContext` object.

Name	Description
`add_output(logical_id, description, value, force=False)`	Adds an output value to the resource group's resource template. If `force` is `False`, an existing output that has the same name is not replaced but is replaced if `force` is `True`. To save the changes, call `save_template`. Returns `True` if outputs are added.
`add_parameters(parameter_definitions, force=False)`	Adds parameter definitions to the resource group's resource template. The `parameter_definitions` parameter is a dictionary that is merged into the parameter definitions in the template. If `force` is `False`, existing definitions are not replaced but are replaced if **force** is True. To save the changes, call `save_template`. Returns `True` if parameter definitions are added.
`add_resources(resource_definitions, force=False, dependencies=None)`	Adds resource definitions to the resource group's resource template. The `resource_definitions` parameter is a dictonary that is merged into the template's current `Resources` object value. If `force` is `False`, existing definitions are not be replaced but are replaced if `force` is `True`. The `dependencies` parameter can be a dictionary that specifies a list of values that are added to the existing resources `DependsOn` list. New resources are always added to the `AccessControl` resource's `DependsOn` list even if the `dependencies` parameter is not specified. To save the changes, call `save_template`. Returns `True` if resource definitions are added.
`copy_directory(source_path, relative_destination_path='.', force=False)`	Copies the contents of a specified directory into the resource group's directory. If a value for `relative_destination_path` is specified, the content is written into the specified subdirectory in the resource group directory. If `force` is `False`, existing files are not replaced, but are replaced if `force` is `True`. Returns `True` if any files are copied.
`copy_file(source_path, relative_destination_path, force=False)`	Copies a file to the resource group directory. The `relative_destination_path` parameter can contain the destination file name or a destination path and file name. If `force` is `False`, existing files are not replaced but are replaced if `force` is `True`. Returns `True` if the file is copied.
`create_file(relative_destination_path, initial_content, force=False)`	Creates a file in the resource group directory. The `relative_destination_path` parameter can contain the destination file name or a destination path and file name. The `initial_content` parameter should contain the initial text content for the file. If `force` is False, existing files are not replaced but are replaced if `force` is `True`. Returns `True` if the file is created.

Name	Description
`directory_path`	The full path to the resource group directory.
`game_cpp_code_path`	The full path to the directory where the C++ code associated with the resource group should be written. This parameter is used when service API client code is generated.
`get_pending_resource_status(deployment_name)`	Gets a dictionary that describes the resource group's pending resource status. The status is determined by comparing the configured resource definitions, Lambda code, and parameter values to the current definitions, code, and values.
`get_stack_id(deployment_name, optional=False)`	Gets the resource group's stack ID for the specified deployment. If no stack exists for the resource group, raises a `HandledError` or returns `None` if optional is `True`.
`get_stack_parameters(deployment_name, uploader)`	Get stack parameter for the specified deployment. The `uploader` parameter must be an `Uploader` object, which is used to determine the values for the `ConfigurationBucket` and `ConfigurationKey` parameters.
`get_template_with_parameters(deployment)`	Returns a Python object that contains the contents of the resource group's `resource-template.json` file. The template's default parameter values are set according to the parameter configuration for the specified deployment.
`is_gem`	`True` if the resource group is defined by a gem; `False` if the resource group is defined in the project's `resource-group` directory.
`name`	The name of the resource group.
`remove_output(logical_id)`	Removes an output value from the resource group's resource template. Returns `True` if the output value existed and was removed. To save the changes, call `save_template`.
`remove_parameters(parameter_names)`	Removes parameter definitions from the resource group's resource template. The `parameter_names` parameter must be a list of the names of the parameters to remove. Returns `True` if the parameter definitions are removed.
`remove_resources(resource_names)`	Removes resource definitions from the resource group's resource template. The `resource_names` parameter must be a list of the logical names of the resources to remove. Returns `True` if the resource definitions are removed. To save the changes, call `save_template`.

Name	Description
`save_template()`	Saves the current value of the template property to the resource group's `resource-template.json` file.
`template`	A Python object that contains the content of the resource group's `resource-template.json` file.
`template_path`	The full path to the resource group's `resource-template.json` file.

Other properties or functions of this object are internal to resource manager and should not be used.

ResourceGroupContext Object

The `ResourceGroupContext` object provides access to resource group configuration data. To get a `ResourceGroupContext` object, use the `resource_groups` property of the `Context` object.

Name	Description
`get(name, optional=False)`	Returns a `ResourceGroup` object for the specified resource group. If the resource group doesn't exist, raises a `HandledError` or none if `optional` is `True`.
`keys()`	Returns the names of the resource groups in a list.
`values()`	Returns a list of `ResourceGroup` objects.

Other properties or functions of this object are internal to resource manager and should not be used.

StackContext Object

The `StackContext` object provides a number of helper functions that are useful when working with AWS CloudFormation stacks. To get a `StackContext` object instance, use the `stack` property of the `Context` object.

Name	Description
`confirm_stack_operation(stack_id, stack_description, args, pending_resource_status, ignore_resource_types = [])`	Displays pending changes and prompts the user to confirm that they want to perform a stack operation. The `args` parameter should be the parsed command line arguments for the command. It is used to avoid prompting the user if the necessary `--confirm-...` options are provided. The `ignore_resource_types` parameter is a list of resource types that are not included in the list of changes shown to the user. This parameter filters changes to the `Custom::CloudGemFramework::EmptyDeployment` resource.

Name	Description
`create_using_template(stack_name, template_body, region, created_callback=None, capabilities=[])`	Uses the provided template to start a stack creation operation. Displays stack events until the create operation is complete. If the value for `created_callback` is not `None`, the value must be a called function that has the created stack ID as its only parameter.
`create_using_url(stack_name, template_url, region, parameters=None, created_callback=None, capabilities=[])`	Uses a template identified by a URL to start a stack creation operation. Displays stack events until the create operation is complete. If the value for `created_callback` is not `None`, the value must be a called function that has the created stack ID as its only parameter.
`delete(stack_id, pending_resource_status=None)`	Starts a stack delete operation and displays stack events until the delete operation is complete. If provided, the `pending_resource_status` parameter specifies the resources to be deleted from an Amazon S3 bucket in preparation for deletion of the bucket itself. To retrieve a value for this parameter, use the `get_pending_resource_status` function on a `ResourceGroup` object, or use the `get_pending_resource_status` function on the `StackContext` object. Preparing an S3 bucket for deletion deletes all the objects from the bucket unless the bucket specifies a `DeletionPolicy` property value of `Retain`.
`describe_resources(stack_id, recursive=True, optional=False)`	Uses the AWS CloudFormation `describe_stack_resources` API to return a description of a stack's resources. The content of the `StackResources` structure is converted to a dictionary keyed on the resource's logical ID. If `recursive` is `True`, the descriptions of the resources in the nested stacks are also returned. In this case the resource's logical ID has the form *<nested-stack-logical-id>.<nested-resource-logical-id>*. If the stack does not exist, a `HandledError` is raised if `optional` is `False`. Otherwise, an empty dictionary is returned.
`describe_stack(stack_id, optional=False)`	Returns a dictionary that contains the data returned by a call to the AWS CloudFormation `describe_stacks` API. If the stack does not exist, raises a `HandledError` or returns `None` if optional is `True`.
`get_current_parameters(stack_id)`	Gets a stack's current AWS CloudFormation template parameter values.

Name	Description
`get_current_template(stack_id)`	Gets a Python object that contains the stack's current AWS CloudFormation template.
`get_pending_resource_status(stack_id, new_template={}, new_parameter_values={}, new_content_paths={})`	Gets a dictionary that describes a stack's pending resource status. The status is determined by comparing the configured resource definitions, Lambda code, and parameter values to the current definitions, code, and values. The `new_template` parameter is the resource definition template to which the stack's current template will be compared. The `new_parameter_values` parameter is a dictionary of parameter values that will be compared to the stack's current parameter values. The `new_content_paths` parameter is a dictionary that maps logical resource names to a list of full directory or file paths. The paths specify where content related to each resource is stored. A resource has pending update status if any of these files have changed since the last resource status update.
`get_physical_resource_id(stack_id, logical_resource_id, expected_type=None, optional=False)`	Gets the physical ID of a resource in the specified stack. If `expected_type` is specified, the resource type is verified to be that type. If the type returned is not the type specified, raises a `HandledError`. If the specified resource does not exist, raises a `HandledError` or returns `None` if `optional` is `True`.
`get_resource_arn(stack_id, logical_resource_id)`	Gets the ARN for a resource in the specified stack.
`get_stack_status(stack_id)`	Gets status information for the stack. This function calls the AWS CloudFormation `describe_stacks` API and returns the `StackStatus` property of the first entry in the `Stacks` array of the response.
`id_exists(stack_id)`	Determines whether a stack with the specified ID (ARN) exists.
`name_exists(stack_name, region)`	Determines whether a stack with the specified name exists in the specified region.

Name	Description
`update(stack_id, template_url, parameters={}, pending_resource_status={}, capabilities={})`	Starts a stack update operation and displays stack events until the update operation is complete. If provided, the `pending_resource_status` parameter specifies the resources to be deleted from an S3 bucket in preparation for deletion of the bucket itself. To retrieve a value for this parameter, use the `get_pending_resource_status` function on a `ResourceGroup` object. Or use the `get_pending_resource_status` function on the `StackContext` object. Preparing an S3 bucket for deletion deletes all the objects from the bucket unless the bucket specifies a `DeletionPolicy` property value of `Retain`.

Other properties or functions of this object are internal to resource manager and should not be used.

ViewContext Object

The `ViewContext` object contains methods that produce Cloud Canvas Resource Manager output messages. Hook functions typically do not require these functions. For more information, see the source code for this object.

Uploader Object

You can use an `Uploader` object function to upload content to the project global area of the project's `Configuration` bucket. You pass `Uploader` object instances to the hook functions `before_project_update`, `after_project_update`, `before_resource_group_update`, and `after_resoruce_group_update`.

Name	Description
`context`	The current `Context` object.
`bucket`	The name of the project's `Configuration` bucket.
`key`	The object name prefix that is used when naming uploaded objects.
`upload_content(name, content, description)`	Uploads the specified content using *`key`* + '/' + *`name`* as the object name. A description of the upload is shown to the user.
`upload_file(name,path)`	Uploads a file using *`key`* + '/' + *`name`* as the object name.
`upload_dir(name, path, alternate_root = None)`	Uses *`key`* + '/' + *`name`* as the base object name to recursively upload the contents of a directory. If `alternate_root` is not `None`, the value specified is used as the object name prefix instead of `key`.
`zip_and_upload_directory(directory_path, file_name=None, aggregated_directories=None, aggregated_content=None)`	Recursively compresses the contents of a directory into a `.zip` file. It uses *`key`* + '/' + *`directory-name`*.zip as the object name to upload the file. The *`directory-name`* is the name of the directory at the end of `directory_path`. You can use the `file_name` argument to override the *`directory-name`*.zip part of the object name.

Name	Description
	The `aggregated_directories` argument can be an optional dictionary that specifies the paths of additional directories whose contents are included in the `.zip` file. The keys are the path location in the `.zip` file where the content is put. The `aggregated_content` argument can be a dictionary that contains additional content to include in the `.zip` file. The keys are the path location in the `.zip` file where the content is put.

Update Hook Functions

Update hooks are implemented in a cloud gem's `Gem\<gem-name>\AWS\resource-manager-code\update.py` file. If update hooks are defined in the module, the resource manager uses the parameters that are described in the Hook Function Parameters (p. 330) section to call the following functions.

after_project_updated

The `after_project_updated` hook function is called after a project stack update operation finishes successfully.

Parameter	Description
`project_uploader`	An `Uploader` object that you can use to upload additional data.

after_resource_group_updated

The `after_resource_group_updated` hook function is called after a resource group stack update operation finishes successfully.

Parameter	Description
`deployment_name`	The name of the deployment that was updated.
`resource_group_name`	The name of the resource group that was updated.
`resource_group_uploader`	An `Uploader` object that you can use to upload additional data.

before_project_updated

The `before_project_updated` hook function is called before a project stack update operation begins.

Parameter	Description
`project_uploader`	An `Uploader` object that you can use to upload data for the operation.

before_resource_group_updated

The `before_resource_group_updated` hook function is called before a resource group stack update operation begins.

Parameter	Description
`deployment_name`	The name of the deployment being updated.
`resource_group_name`	The name of the resource group being updated.
`resource_group_uploader`	An `Uploader` object that can be used to upload additional data.

gather_writable_check_list

The `gather_writable_check_list` hook function is called before an update operation to gather a list of writable files. If any of the local files to be updated are read-only, the resource manager gives the user an opportunity to make the files writable (for example, with a source control system).

Parameter	Description
`check_list`	A list of the full paths of writeable files.

Command Hook Functions

If the following command line hook functions exist in a module, they are defined in a cloud gem's `Gem \<gem-name>\AWS\resource-manager-code\command.py` file. The resource manager uses the parameters in the Hook Function Parameters (p. 330) section to call the following functions.

add_cli_commands

Adds additional commands to the command line parser. Called before command line argument parsing. Cloud Canvas Resource Manager uses the Python `argparse` module for command line parsing. All commands are grouped into a number of different subparsers. For example, the commands `lmbr_aws project list-resources` and `lmbr_aws deployment list` contain the subparsers `project` and `deployment`, which define a `list-resources` and `list` command, respectively.

Parameter	Description
`subparsers`	The `subparser` collection object returned by the `argparse add_subparsers` function. Use the hook function to add a subparser to the collection. Then, add the commands to the subparser.

`add_common_args`	Adds a set of common arguments to a command. This function adds the following arguments. For a description of these arguments, see Using the Cloud Canvas Command Line (p. 408).
	
	`--aws-access-key`
	
	`--aws-secret-key`
	
	`--profile`
	
	`--assume-role`

Parameter	Description

	`--root-directory`
	
	`--game-directory`
	
	`--aws-directory`
	
	`--user-directory`
	
	`--verbose`
	
	`--no-prompt`
	
	The common arguments are processed by `lmbr_aws`. The hook does not typically process these options.
	
	This function can take the following parameters:
	
	`parser` – The `argparse` defined parser object to which the arguments are added.
	
	`no_assume_role` – `True` specifies that the `--assume-role` common argument is not added; `False` specifies that it is.

add_cli_view_commands

Adds additional methods to the `ViewContext` object. Called before other command line commands.

Parameter	Description
`view_context`	A `ViewContext` object.

add_gui_commands

Adds commands to the Cloud Canvas Resource Manager in Lumberyard Editor. When the resource manager window is first opened, Lumberyard Editor initializes the resource manager Python subsystem, which calls `add_gui_commands`.

Parameter	Description
`handlers`	A dictionary that maps user interface command names to the handler functions that process them. The command names are passed to Python from the user interface.

add_gui_view_commands

Adds additional methods to the `ViewContext` object. Called before a GUI command is executed.

Parameter	Description
`view_context`	A `ViewContext` object.

Running AWS API Jobs Using the Cloud Gem Framework

The Cloud Gem Framework and this documentation are in preview release and are subject to change.

The Cloud Gem Framework Gem provides C++ classes that can execute any C++ AWS API call by using the Lumberyard job execution system. This allows the operation to be performed on background threads that are managed by the job system.

To use AWS API Jobs in your project

1. In Lumberyard Project Configurator, enable the **Cloud Canvas Common** and **Cloud Gem Framework** gems for your project.
2. We recommend that you put the code that uses AWS in a gem, but this is not required. If you do use a gem, make the **Cloud Gem Framework** and **Cloud Canvas Common** gems dependencies of your gem by adding the following to your gem's `gem.json` file.

```
"Dependencies": [
    {
        "Uuid" : "6fc787a982184217a5a553ca24676cfa",
        "VersionConstraints": [ "~>0.1" ],
        "_comment": "CloudGemFramework"
    },
    {
        "Uuid" : "102e23cf4c4c4b748585edbce2bbdc65",
        "VersionConstraints": [
            "~>0.1"
        ],
        "_comment": "CloudCanvasCommon"
    }
],
```

3. Activate your gem for your project.
4. In your gem or game project's `.wscript` file, make the following changes:

   a. To the list of includes, add:

   ```
   bld.Path('Code/SDKs/AWSNativeSDK/include')
   ```

   b. To the list of used static libraries, add `CloudGemFrameworkStaticLibrary`.

   c. Add `AWS_CPP_SDK_CORE` and other AWS API dynamic libraries as required. For a list of available aliases like `AWS_CPP_SDK_LAMBDA` and other library names, see the `dev\_WAF_\3rd_party\aws_native_sdk_shared.json` file.

   d. Add the security libraries for platforms other than Windows, as in the following `.wscript` file for a gem.

   ```
   SUBFOLDERS = []

   def build(bld):

       import lumberyard_sdks

       bld.DefineGem(
           includes = [bld.Path('Code/SDKs/AWSNativeSDK/include')],
           file_list = ['cloudcanvassample.waf_files'],
           use = ['CloudGemFrameworkStaticLibrary'],
           uselib = ['AWS_CPP_SDK_CORE', 'AWS_CPP_SDK_LAMBDA'],
   ```

```
        darwin_lib = ['curl'],
        linux_lib = ['curl'],
        ios_lib = ['curl'],
        appletv_lib = ['curl'],
        durango_lib = ['msxml6'],
        ios_framework = [ 'security' ],
        appletv_framework = [ 'security' ]
    )

    bld.recurse(SUBFOLDERS)
```

5. Include the `CloudGemFramework\AwsApiJob.h` header and the AWS SDK header files that are required for calling an API, as in the following example:

```
#include <CloudGemFramework/AwsApiRequestJob.h>

#pragma warning(disable: 4355) // <future> includes ppltasks.h which throws a C4355
 warning: 'this' used in base member initializer list
#include <aws/lambda/LambdaClient.h>
#include <aws/lambda/model/InvokeRequest.h>
#include <aws/lambda/model/InvokeResult.h>
#include <aws/core/utils/Outcome.h>
#include <aws/core/utils/memory/stl/AWSStringStream.h>
#pragma warning(default: 4355)
```

6. Using code similar to the following, run an AWS API job. An alternative approach is to extend the job class (like `LambdaInvokeRequestJob` in the example) and provide overrides for the `OnSuccess` and `OnFailure` methods.

```
using LambdaInvokeRequestJob = AWS_API_REQUEST_JOB(Lambda, Invoke);

auto job = LambdaInvokeRequestJob::Create(
    [](LambdaInvokeRequestJob* job) // OnSuccess handler - runs on job thread
    {
        Aws::IOStream& stream = job->result.GetPayload();
        std::istreambuf_iterator<AZStd::string::value_type> eos;
        AZStd::string content =
 AZStd::string{std::istreambuf_iterator<AZStd::string::value_type>(stream),eos};
        AZ_Printf("Example", "Got response %s", content.c_str());
    },
    [](LambdaInvokeRequestJob* job) // OnError handler (optional) - runs on job thread
    {
        AZ_Printf("Example", "Was error %s", job->error.GetMessageA().c_str());
    }
);

AZStd::string content = "...";

std::shared_ptr<Aws::StringStream> stream = std::make_shared<Aws::StringStream>();
*stream << content.c_str();

job->request.SetFunctionName("...");
job->request.SetBody(stream);
job->Start();
```

7. If your project uses the **Cloud Canvas Resource Manager**, get the physical resource ID and the logical resource ID of the AWS resource for each resource group. These IDs cause your AWS API call to use the correct resource for the active deployment. This ensures that your development, test, and released versions of a game don't interfere with each other.

```
#include <CloudCanvasCommon/CloudCanvasCommonBus.h>

AZStd::string functionName;
```

```
EBUS_EVENT_RESULT(functionName, CloudCanvasCommon::CloudCanvasCommonRequestBus,
 GetLogicalToPhysicalResourceMapping, "RESOURCE-GROUP.RESOURCE");

job->request.SetFunctionName(functionName.c_str());
```

8. If your project uses the **Cloud Canvas Resource Manager**, the AWS API is called using the player's AWS credentials. These credentials are provided by the anonymous Amazon Cognito Identitypool that Cloud Canvas creates for your project. If you do not use **Cloud Canvas Resource Manager** or want to use other credentials, you can use code like the following to override the default configuration.

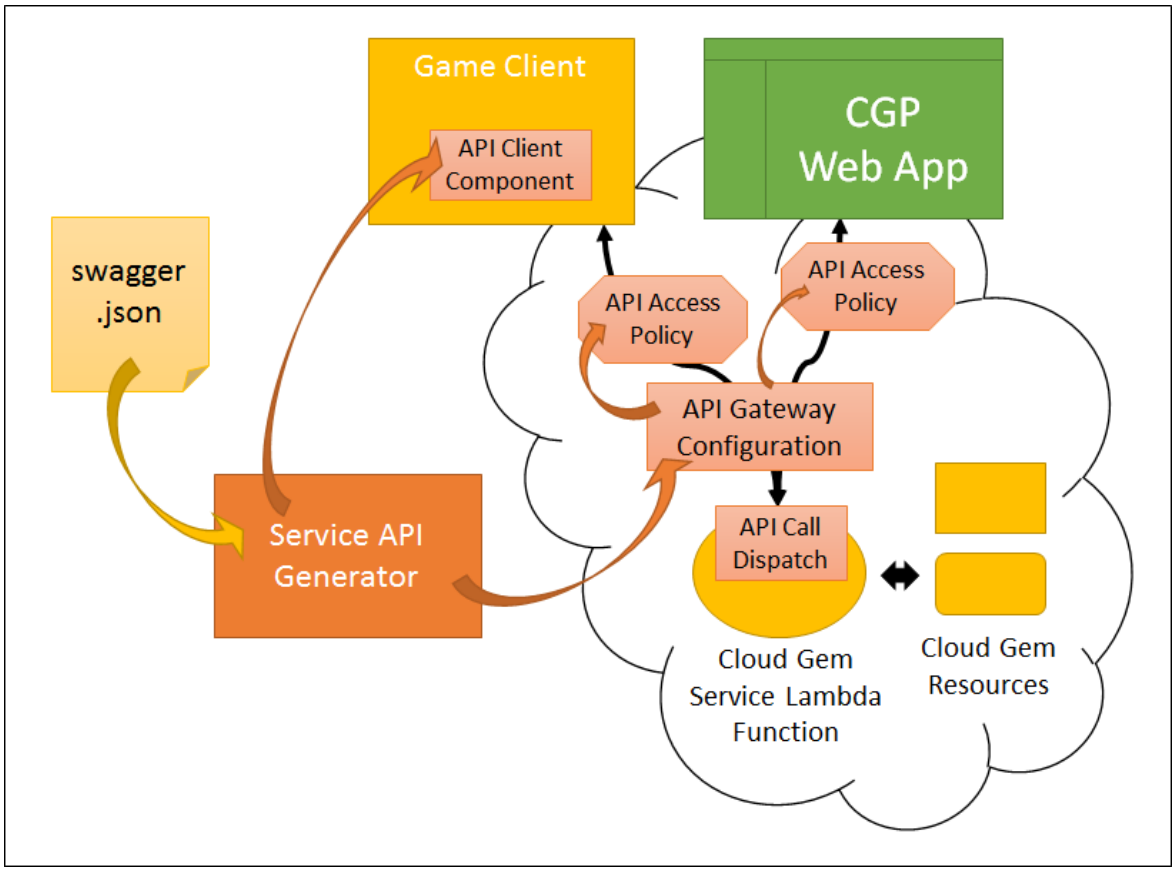
```
#include <aws/core/auth/AWSCredentialsProvider.h>

LambdaInvokeRequestJob::Config config(LambdaInvokeRequestJob::GetDefaultConfig());
const char* accessKey = "...";
const char* secretKey = "...";
config.credentialsProvider =
 std::make_shared<Aws::Auth::SimpleAWSCredentialsProvider>(accessKey, secretKey);
config.requestTimeoutMs = 20000;

auto job = LambdaInvokeRequestJob::Create(
    ..., // OnSuccess handler
    ..., // OnError handler
    &config
);
```

Cloud Gem Framework Service API

This documentation is preliminary and is subject to change.

Lumberyard cloud gems provide services like the Cloud Gem Portal (p. 200) (CGP) that configure and manage a game's operation. The services are implemented by AWS Lambda function resources. The game and tools access these services through the Amazon API Gateway. The code that you provide runs in a Lambda function. API Gateway manages access to the service, provides caching for frequently used results, and supports request throttling. It can even publish your APIs, potentially bringing in revenue.

The following diagram shows how the Cloud Gem Framework service API interacts with the game client, the Cloud Gem Portal, AWS Lambda, and API Gateway.



Cloud Gem Swagger API Descriptions

The Lumberyard game engine and tools use API operations to communicate with your cloud gem service. These APIs are described in the cloud gem's `/dev/Gems/<gem-name>/AWS/swagger.json` file. The `swagger.json` file uses the swagger API definition format, which is an open source framework for RESTful API operations.

Cloud Canvas uses these API descriptions to generate Lumberyard engine components that execute service APIs for the game. The Cloud Gem Portal (p. 200) web application also uses the API descriptions when it makes service requests. In addition, the API descriptions are used to configure API Gateway to work with your service.

Topics

- Resources (p. 354)
- Operations (p. 355)
- Security (p. 358)
- Cloud Gem Framework Extension Object (p. 360)
- Game Clients (p. 362)
- Generated Game Client Code Example (p. 363)
- Calling a Game API (p. 376)
- Publishing Your API (p. 379)

Resources

A cloud gem's service API is implemented based on the resources that are defined in the cloud gem's `resource-template.json` file. The following are the key resources for a cloud gem:

- `ServiceAPI` – An AWS CloudFormation custom resource provided by the Cloud Gem Framework. The handler for this resource configures API Gateway to use the cloud gem's `swagger.json` file.
- `ServiceLambda` – An AWS Lambda function that implements the cloud gem's functionality.
- `ServiceLambdaConfiguration` – An AWS CloudFormation custom resource provided by Cloud Canvas Resource Manager that configures (p. 303) the `ServiceLambda` resource.

To add definitions for these resources to a `resource-template.json` file, type the following command.

```
lmbr_aws cloud-gem-framework add-service-api-resources --resource-group <gem-name>
```

In addition to adding the resources mentioned, the `add-service-api-resources` command does the following:

- Adds a `swagger.json` file to the cloud gem's AWS directory, if one doesn't already exist.
- Adds the `ServiceAPI` and `ServiceLambda` resources to the `AccessControl` resource definition's `DependsOn` property. This insures that the `AccessControl` resource is processed by AWS CloudFormation after the `ServiceAPI` and `ServiceLambda` resources have been processed.

To use the service API resources that you add to a `resource-template.json` file, you must upload those resources to AWS. To upload them, you can use the `lmbr_aws resource#group upload` command, the `lmbr_aws deployment upload` command, or click **Upload Resources** in the **Resource Manager** feature in Lumberyard Editor.

Custom::CloudGemFramework::ServiceApi Resource

The handler for the `Custom::CloudGemFramework::ServiceApi` AWS CloudFormation resource is provided by the Cloud Gem Framework. This handler creates, updates, and deletes API Gateway REST API, operation, deployment, and stage resources as needed.

`ServiceApi` resource definitions accept the following parameters:

```
{
    "Resources": {
        ...
        "ServiceApi": {
            "Type": "Custom::CloudGemFramework::ServiceApi",
            "Properties": {
                "ServiceToken": { "Ref": "ProjectResourceHandler" },
                "ConfigurationBucket": { "Ref": "ConfigurationBucket" },
                "ConfigurationKey": { "Ref": "ConfigurationKey" },
                "CacheClusterSize": { "Ref": "ServiceApiCacheClusterSize" },
                "CacheClusterEnabled": { "Ref": "ServiceApiCacheClusterEnabled" },
                "MethodSettings": { ... },
                "SwaggerSettings": {
                    "ServiceLambdaArn": { "Fn::GetAtt": [ "ServiceLambda", "Arn" ] }
                }
            },
            ...
```

`ServiceToken`

Identifies the Lambda function that implements the custom resource handler.

`ConfigurationBucket`

Identifies the bucket that contains the uploaded `swagger.json` file.

`ConfigurationKey`

Identifies the location in the bucket where the `swagger.json` file is uploaded.

`CacheClusterSize`

Provides the API Gateway cacheClusterSize value when you create or update the API Gateway stage.

`CacheClusterEnabled`

Provides the API Gateway cacheClusterEnabled value when your create or update the API Gateway stage.

`MethodSettings`

Not implemented.

`SwaggerSettings`

Provides values that you insert into the uploaded `swagger.json` file before it is passed to API Gateway. For example, you can use `$ServiceLambdaArn$` in the `swagger.json` file to insert the value of the `SwaggerSettings` `ServiceLambdaArn` property.

The following settings are automatically defined for you:

`ResourceGroupName`

The name of the resource group that is defined the `ServiceApi` resource.

`DeploymentName`

The name of the deployment that the `ServiceApi` resource is in.

`RoleArn`

The ARN of the role that grants API Gateway the permission to invoke the `ServiceLambda` (or other permissions configured by the Cloud Canvas Resource Manager Security System (p. 393)).

`Region`

The AWS region where the RESTful API resides.

`RestApiResourceName`

The name to use for the API Gateway REST API resource. API Gateway takes this value from the swagger infoObject `title` property (set to `$RestApiResourceName$` in the default `swagger.json` file). This is the stack name of the resource group with the `ServiceApi` logical resource ID appended (usually `-ServiceApi`).

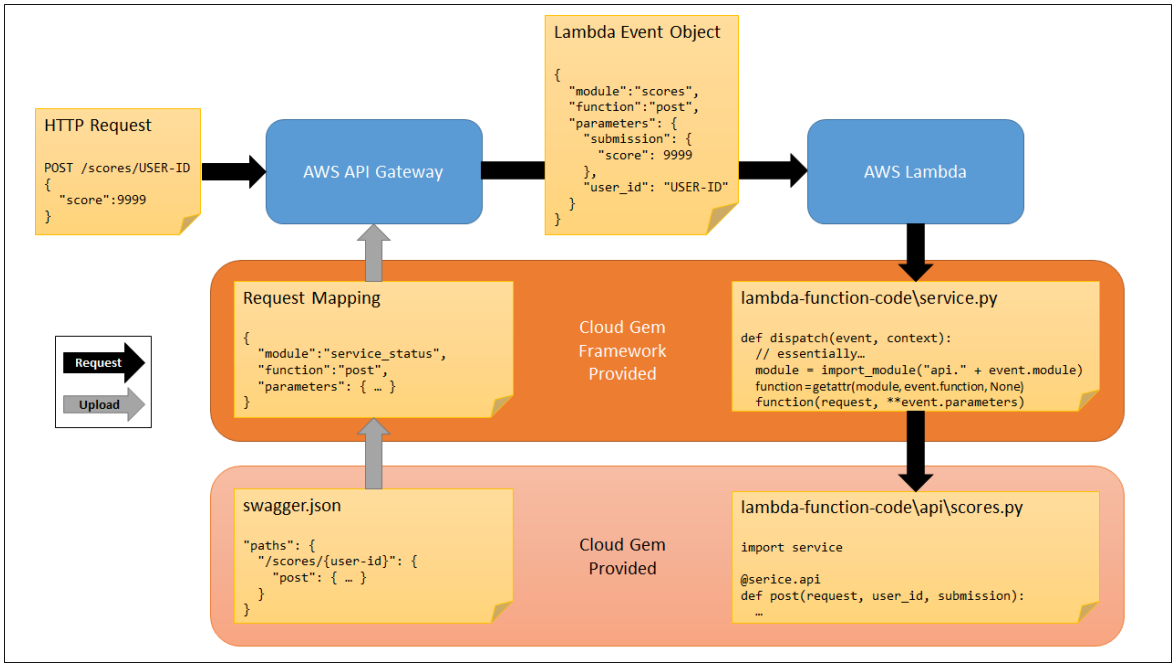
Operations

A cloud gem's service API can implement multiple distinct operations. You define operations in the `swagger.json` file by adding `operationObject` instances to a `pathItemObject`. For each operation, you can define the input data that the operation requires and the output data that it produces. The input data can include path, query, and body parameters. For more information on using swagger to define APIs, see http://swagger.io/ .

API Gateway routes API requests to the cloud gem's Lambda function. When you upload the cloud gem's resources to AWS, the cloud gem's `swagger.json` file is processed. This processing creates the request and response mappings that API Gateway uses to call the cloud gem's Lambda function. This configuration is controlled by the `x-amazon-cloud-canvas-lambda-dispatch` (p. 360) extension objects in the `swagger.json` file.

Service API operations are implemented in the `ServiceLambda` AWS Lambda function resource. The code for the Lambda function comes from the cloud gem's `lambda-function-code` directory. The Cloud Gem Framework provides a service request dispatch module. For more information, see Request Execution (p. 357).

The following diagram illustrates both request and upload processing.



Topics

- Default Request Mapping (p. 356)
- Default Response Mapping (p. 357)
- Request Execution (p. 357)

Default Request Mapping

A JSON object that implements the request operation is sent to the Lambda function. The JSON object contains the module and function name and the operation's parameters, as the following skeletal syntax shows.

```
{
  "module": "<module-name>",
  "function": "<function-name>",
  "parameters": {
    "<parameter-name>": <parameter-value>,
    ...
  }
}
```

The default module name is derived from the operation path. A path like `/player/{id}/highscores` results in a module name like `player_highscores`. Parameters in the path are skipped. The paths `/player` and `/player/{id}` both map to the module name `player`. In this case the handler function in that module can use the existence of an `id` parameter value to determine the corresponding behavior. The path `/` (and `/{param}`) are mapped to the module name `root`.

The default function name is the operation name like `GET`, `POST`, or `PUT`.

These defaults can be overridden by specifying the module and/or function properties of an `x-amazon-cloud-canvas-lambda-dispatch` extension object. For more information, see Cloud Gem Framework Extension Object (p. 360).

Parameter names are taken from the parameter definitions in the `swagger.json` file. Path, query, and body parameter types are supported.

Default Response Mapping

The Lambda function returns the value to the client as a JSON object such as the following.

```
{
  "result":<lambda-return-value>
}
```

If the Lambda function exits with an error (for example, by raising an unhandled exception), it returns the JSON object in the following format.

```
{
  "errorMessage":"<error-message>",
  "errorType":"<error-type>"
}
```

If the error message received from the Lambda function starts with the text `Client Error:`, then an `HTTP 400` response is sent to the client. The `errorMessage` and `errorType` properties received from the Lambda function are forwarded to the client.

If the error message received from the Lambda function does not start with `Client Error:`, or no error message is received at all, an `HTTP 500` response is sent to the client. As a security measure, the `errorMessage` returned to the client is always *An internal server error has occurred*, and `errorType` is `ServiceError`. This is to avoid sending the exception message to the client. Doing so could provide information that could allow an attacker to discover exploits in your service implementation.

In both the `HTTP 400` and `HTTP 500` cases, no other information (such as a stack trace) is sent to the client; this also is a security measure.

Request Execution

The Cloud Gem Framework has built-in support for service API Lambda functions that are implemented in Python. No built-in support is provided for Node.js or Java Lambda functions. To support these languages, implement the Lambda function handler to look for the `module` and `function` properties on the event object that the Lambda function provides. You also might need to override the default values generated for the `x-amazon-cloud-canvas-lambda-dispatch` (p. 360) `module` and `function` properties during the `swagger.json` file processing.

A `service.py` module file is provided for you in the service's AWS Lambda function. This module's `dispatch` function uses the `module` and `function` properties of the `event` object to dispatch the request to the service code that you provide. These properties are set by the request mapping that configures API Gateway.

Place your service code in your resource group's `lambda-function-code\api` directory. The dispatcher uses `importlib.import_module` to load the modules from that directory. If the specified function has the `@service.api` decorator, the dispatcher invokes the function on that module, as in the following example.

```
import service

@service.api
```

```
def post(request, submission, user_id):
    ...
```

The first argument passed to the function is a `dispatch.Request` object. The `request` object has the following properties.

- **`event`** – The event object that AWS Lambda passes to the dispatch handler. The contents of the object are determined by the request template. Additional properties added to the template through the `x-amazon-cloud-canvas-lambda-dispatch` object's `additional-request-template-content` property are also located here.

- **`context`** – The Python context object that AWS Lambda passes to the dispatch handler.

The request object's parameters are passed to the handler function as keyword arguments (that is, by using Python `**parameters`).

Error Handling

A `ClientError` class is provided in the `errors.py` file. This class extends `RuntimeException` and ensures that the error message is prefixed with `Client Error:`. This triggers an `HTTP 400` response from API Gateway as described in Default Response Mapping (p. 357).

If other exceptions are raised during processing, a generic `Internal Service Error` message is sent to the client.

Security

When you make APIs available on the Internet, you must be concerned with security. A best practice is to limit API access to only the people who require it. However, some APIs must be called by the game client. These APIs can be called by any game player or potentially anyone on the Internet.

For all APIs, consider the following:

- Use the access control mechanism described in the next section to limit API access to only those who require it.

- Don't trust parameter values provided by the client. Verify that the parameter values match expectations before you use them. Be careful when inserting parameter values into query strings that are sent to DynamoDB or other services. For more information, see code injection.

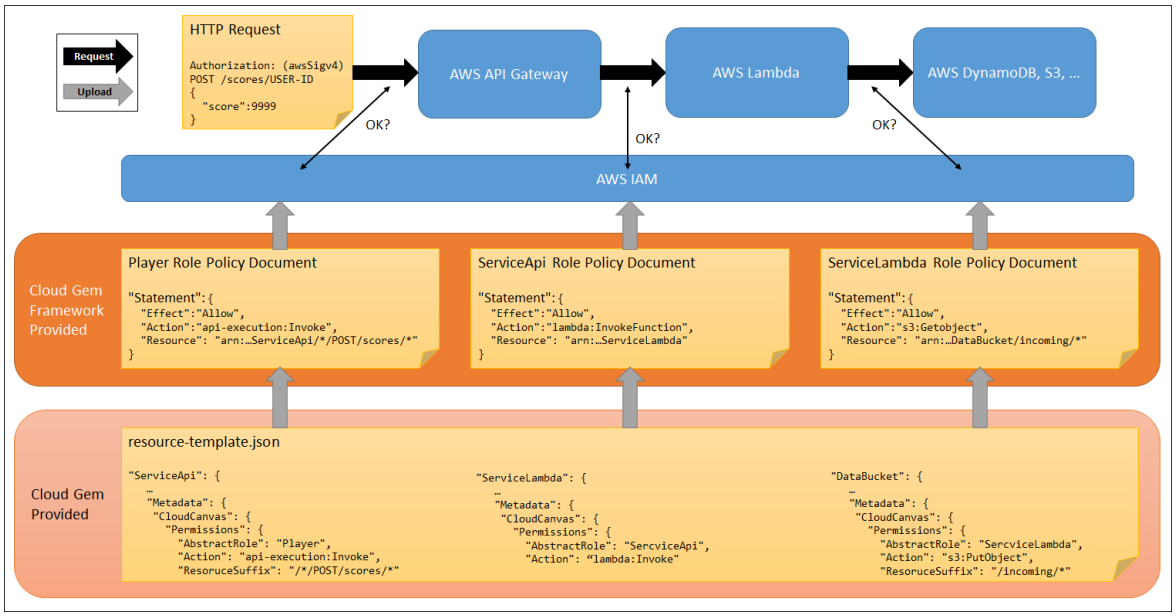
- API Gateway automatically protects your backend systems from distributed denial-of-service (DDoS) attacks, whether attacked with counterfeit requests (Layer 7) or SYN floods (Layer 3). However, this does not protect from less frequent requests that do not trigger API Gateway's protections. These other requests might still have a significant impact on your operating costs due to excessive I/O or on game performance.

Access Control

Configuring access control for a service API involves setting three distinct sets of permissions:

1. The `execute-api` operation, enforced by API Gateway.
2. The Lambda `Invoke` function, enforced by AWS Lambda
3. Access to the cloud gem resource, enforced by the resource's AWS service (for example, Amazon DynamoDB or Amazon S3.)

In all three cases, you use the Cloud Canvas Resource Manager Security (p. 393) system to configure access. This involves putting `Permissions` metadata on the `ServiceApi` and `ServiceLambda` resource definitions, as well as on the definitions of resources accessed by the `ServiceLambda` code. This is illustrated in the following diagram:



The permissions granted by `ServiceApi` are described in detail in the next section. `ServiceLambda` gives `ServiceApi` permission to invoke the Lambda function. Other resources give `ServiceLambda` the permissions that the Lambda function requires.

Service API Permissions

You can secure service API operations on an individual basis. For example, you could give permission to submit high scores to the game, and permission to delete fraudulent high scores to someone who manages operations.

By default, only valid AWS IAM credentials can execute operations. This is done using swagger security requirement and security definitions objects that are added to each operation during upload processing (p. 361), unless the swagger operation object already defines a security object. You can configure API Gateway to use other forms of security, or none at all, by putting security objects in swagger object definitions in your `swagger.json` file.

To grant permission to execute an operation, modify your `resource-template.json` file to include metadata on the `ServiceApi` resource definition, as in the following example.

```
{
    "Resources": {
        ...
        "ServiceApi": {
            "Type": "Custom::CloudGemFramework::ServiceApi",
            "Properties": { ... },
            "Metadata": {
                "CloudCanvas": {
                    "Permissions": [
                        {
                            "AbstractRole": [ "ProjectOwner", "DeploymentOwner" ],
                            "Action": "execute-api:*",
                            "ResourceSuffix": "/*"
                        },
                        {
```

```
                            "AbstractRole": "Player",
                            "Action": "execute-api:Invoke",
                            "ResourceSuffix": "/*/POST/score/*"
                        },
                        {
                            "AbstractRole": "DevOps",
                            "Action": "execute-api:Invoke",
                            "ResourceSuffix": "/*/DELETE/score/*"
                        }
                    ]
                }
            }
        }
    ...
}
```

The `AbstractRole` property determines who has permission to call the API. Cloud Canvas has built in `Player`, `ProjectOwner`, and `DeploymentOwner` roles. You can create others as required. The `abstract` role specified here is mapped to an actual AWS IAM role using metadata on the role definitions. For details, see Understanding the Resource Manager Security System (p. 393).

The `Action` and `ResourceSuffix` are used by the Cloud Canvas Resource Manager access control to generate an AWS IAM policy document statement. This process is described in Statement Reference of IAM Policies for Executing API in API Gateway.

The `Action` property determines what the permission allows someone to do. When you grant permissions to the `Player` abstract role, specify the permission `execute-api:Invoke`. This gives the player permission to invoke the API. When you grant permissions to the `ProjectOwner` and `DeploymentOwner` abstract roles, specify the permission `execute-api:*`. This grants permission to invoke the API and manage the API's cache. For other roles, use your own requirements to determine whether to grant a role permission to manage the cache, invoke the API, or both.

The access control system calculates the ARN of the API Gateway REST API resource to which the action can be applied. The `ResourceSuffix` property from the `Permission` metadata provides only the `/stage-name/HTTP-VERB/resources-path-specifier` part of the ARN described in the Statement Reference of IAM Policies for Executing API in API Gateway.

For `ServiceAPI`, the `stage-name` is always `api`. You can specify either `/api/...` or `/*/...` in the `ResourceSuffix` property value. A `ResourceSuffix` value of `/api/*` or `/*` grants permissions for all operations on all paths of the service API.

Cloud Gem Framework Extension Object

The swagger specification allows tools like the Cloud Gem Framework and API Gateway to define extension objects. These objects allow the `swagger.json` file to provide custom configuration data for the tool. The extension object that the Cloud Gem Framework uses is `x-amazon-cloud-canvas-lambda-dispatch`. This extension object simplifies the configuration of API Gateway for use with an AWS Lambda function.

You can place the `x-amazon-cloud-canvas-lambda-dispatch` object in any of the following swagger objects.

- swagger object – Sets the defaults for all paths and operations.
- path item object – Sets the defaults for all the path's operations and overrides the defaults set on the parent swagger object.
- operation object – Sets values for the operation and overrides the defaults set on the parent path and swagger objects.

The `x-amazon-cloud-canvas-lambda-dispatch` object supports the following properties:

- **`lambda`** – The ARN of the Lambda function that the operation invokes.
- **`module`** – The name of the module that defines the service function that processes the request.
- **`function`** – The name of the function that processes the request. The dispatch module in the Lambda function uses the `function` property and the `module` property to call your code when it receives a request. The dispatch module is described in the Upload Processing (p. 361) section that follows.
- **`additional-properties`** – An object that provides properties that are added to the generated `x-amazon-apigateway-integration` object described in the next section.
- **`additional-request-template-content`** – A string that is inserted into the generated `application/json` request template, which is described in the next section.
- **`additional-response-template-content`** – An object that specifies additional content that is inserted into the generated `application/json` response template. This template is described in the next section. Properties named `200`, `400`, and `500` are supported. These correspond to the successful (200) and error (400 and 550) responses.

See the API Gateway documentation for more information about mapping templates.

Upload Processing

Before the `swagger.json` file is uploaded to API Gateway, the `x-amazon-cloud-canvas-lambda-dispatch` extension objects in the file are processed. This produces the `x-amazon-apigateway-integration` extension objects that configure API Gateway to call your cloud gem's AWS Lambda function.

The `x-amazon-cloud-canvas-lambda-dispatch` object and processing hide a lot of the complexity, and flexibility, of the `x-amazon-apigateway-integration` extension object. The Cloud Gems Framework provides a straightforward and recommended mapping to the API operations that are implemented in Lambda functions. API Gateway offers many other features that can be extremely useful when you must match an API that was implemented elsewhere or use API Gateway as a proxy for existing backend implementations.

You can still use all the power of the `x-amazon-apigateway-integration` object in your cloud gem API operations. If you include the object in the operation objects of your `swagger.json` file, the processing described here is skipped for the operation.

> **Note**
> You can also use the `lmbr_aws cloud-gem-framework service-api-process-swagger` command to process the `swagger.json` file.

At minimum, the swagger object can include an `x-amazon-cloud-canvas-lambda-dispatch` object with a Lambda property. You can allow the module and function for each operation be determined automatically, or you can use `x-amazon-cloud-canvas-lambda-dispatch` objects to specify them.

The default `swagger.json` file provided by the Cloud Gem Framework contains the following `x-amazon-cloud-canvas-lambda-dispatch` object. The string `$ServiceLambdaArn$` is replaced with the `ServiceLambdaArn` `SwaggerSettings` property value provided in the `ServiceApi` resource definition.

```
"x-amazon-cloud-canvas-lambda-dispatch": {
    "lambda": "$ServiceLambdaArn$"
}
```

When the `swagger.json` file is processed, an `x-amazon-apigateway-integration` object is added to every swagger operation object that does not have one. These objects have the following properties:

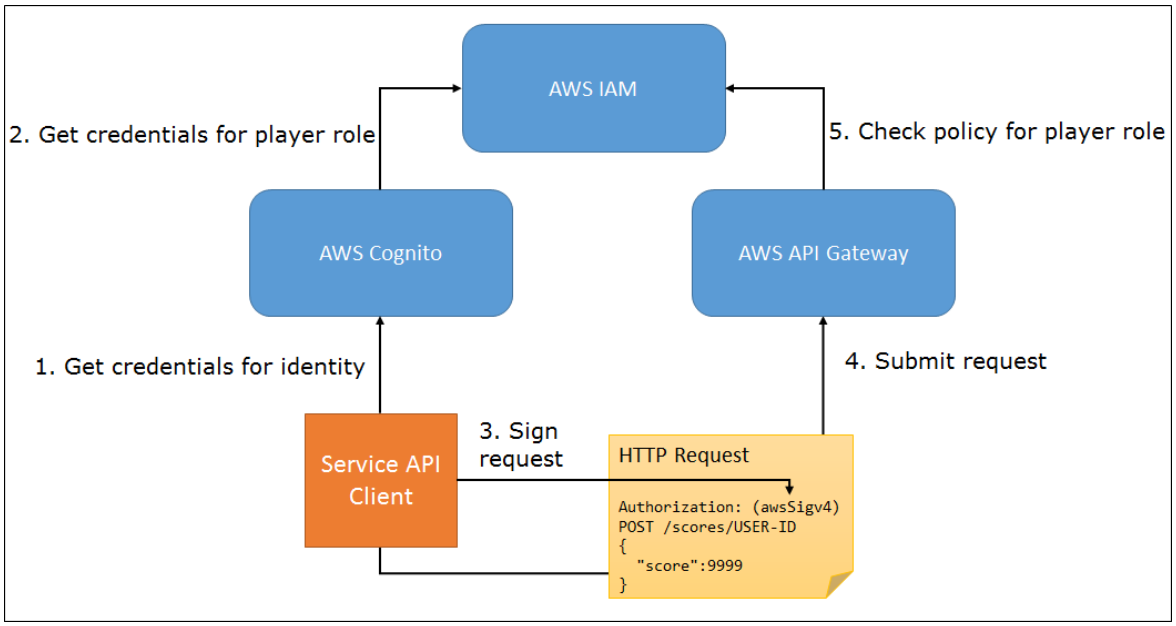
- **`type`** – Specify `AWS` to enable AWS Lambda function integration.
- **`uri`** – Construct the URI of the Lambda function with the value of the `x-amazon-cloud-canvas-lambda-dispatch` object's `lambda` property.

- **`credentials`** – The `Custom::CloudGemFramework::ServiceApi` resource creates the ARN of a role. The role has a policy that is described in Access Control (p. 358).
- **`requestTemplates`** – A `application/json` template that causes a request as described in Default Request Mapping (p. 356). You can include additional content by using the `x-amazon-cloud-canvas-lambda-dispatch` object's `additional-request-template-content` property. This property can be used to pass other values, such as those defined by the `$context` object, to the Lambda function. The additional request template content should start with a `','` (and for consistent formatting, should start with `',\n '` and use `',\n '` between properties).
- **`responses`** Specifies – `application/json` templates for 200 (success), 400 (client error) and 500 (service error) responses as described in Default Response Mapping (p. 357). The 400 and 500 responses are inserted into the swagger responses object. However, you should define a 200 response that specifies a scheme that describes the data that the operation returns.
- Additional properties as specified by the `x-amazon-apigateway-integration-properties` property.

Game Clients

The Cloud Gem Framework can generate game clients for your service API.

The game client can call some API operations on behalf of the player. Other operations are callable only by other roles for use in the Cloud Gem Portal (p. 200) or other tools. By default, the client uses the AWS credentials by assuming the `Player` role through an Amazon Cognito identity pool. This process, described in detail in Player Identity (p. 400), is illustrated in the following diagram.



Generating a Game Client

Generating a game client is straightforward.

To generate a game client

Type the following command:

```
lmbr_aws cloud-gem-framework generate-service-api-code --resource-group <group>
```

The `generate-service-api-code` command reads the `swagger.json` file that contains definitions for the resource group's service API. The command creates a C++ header file that implements a Lumberyard component. The header file is named *<group>*`ClientComponent.h`.

The directory location of the header file depends on whether *<group>* refers to a cloud gem or a project-defined resource group.

Generated Component Header File Locations

Parameter Type	Header File Location
cloud gem	`Gems\`*<gem>*`\Code\AWS\ServiceApi`
resource group	`Code\`*<game>*`\AWS\`*<group>*`\ServiceAPI`

Component Header File Accessibility

For a gem, the default location of the generated header file does not make it accessible outside of that gem. Typically, a cloud gem's service API is private to the gem, and the gem exposes its functionality through its own custom component. If you want to provide direct access to the cloud gem's API from other gems or game code, you can move the generated header file into the gem's `Code\Include` directory.

Notes

- To use the generated client, add the generated file to your gem or project's WAF file list, and then rebuild the gem or project.
- You can use the generated client component to invoke the API from C++ or Lua. For more information, see Calling a Game API (p. 376).
- For sample code, see Generated Game Client Code Example (p. 363).

Generated Game Client Code Example

The following client code was generated by the `swagger.json` file for the Message of the Day Gem that is included with Lumberyard. The swagger file for the gem defines many operations. For brevity, the example shows client code for only the following `GET` and `PUT` operations:

- `GET /admin/messages?index={index}&count={count}&filter={filter}`
- `PUT /admin/messages/{msg_id}`

For comments, see the Examining the Generated Game Client Code (p. 374) section that follows the example.

```
// Example Generated Game Client
  1  #pragma once
  2  #include <AzCore/Component/Component.h>
  3  #include <AzCore/Component/Entity.h>
  4  #include <AzCore/Component/ComponentBus.h>
  5  #include <AzCore/EBus/EBus.h>
  6  #include <AzCore/Memory/SystemAllocator.h>
  7  #include <AzCore/Rtti/BehaviorContext.h>
  8  #include <AzCore/Serialization/EditContext.h>
  9  #include <AzCore/Serialization/SerializeContext.h>
 10  #include <AzCore/std/smart_ptr/scoped_ptr.h>
 11  #include <AzCore/std/string/conversions.h>
 12
 13  #if defined (PLATFORM_SUPPORTS_AWS_NATIVE_SDK)
 14  #include <aws/core/http/HttpRequest.h>
```

```
 15  #include <aws/core/http/HttpResponse.h>
 16  #endif // (PLATFORM_SUPPORTS_AWS_NATIVE_SDK)
 17
 18  #include <LmbrAWS/ILmbrAWS.h>
 19
 20  #include <CloudGemFramework/ServiceRequestJob.h>
 21
 22  #include "StdAfx.h"
 23
 24  namespace CloudGemMessageOfTheDay {
 25  namespace ServiceAPI {
 26
 27      const char* LmbrAWS_CodeGen_PutAdminMessagesReturnType_UUID= "{ec889bb0-c329-11e6-
b753-80a589a02a3d}";
 28      const char* LmbrAWS_CodeGen_Component_UUID= "{ec8874a1-c329-11e6-
accd-80a589a02a3d}";
 29      const char* LmbrAWS_CodeGen_ResponseHandler_UUID= "{ec8874a4-c329-11e6-
a067-80a589a02a3d}";
 30      const char* LmbrAWS_CodeGen_NotificationBus1_UUID= "{ec8874a2-c329-11e6-
a661-80a589a02a3d}";
 31      const char* LmbrAWS_CodeGen_RequestBus1_UUID= "{ec8874a3-c329-11e6-
a1e9-80a589a02a3d}";
 32      const char* LmbrAWS_CodeGen_DeleteAdminMessagesReturnType_UUID= "{ec889bb1-
c329-11e6-aa94-80a589a02a3d}";
 33      const char* LmbrAWS_CodeGen_DetailedMessageList_UUID= "{ec889bb4-
c329-11e6-8650-80a589a02a3d}";
 34      const char* LmbrAWS_CodeGen_ServiceStatus_UUID= "{45baaccf-c88b-11e6-
b813-80a589a02a3d}";
 35      const char* LmbrAWS_CodeGen_DetailedMessageData_UUID= "{ec889bb3-c329-11e6-
bf48-80a589a02a3d}";
 36
 37
 38      // redefs
 39
 40
 41      bool WriteJson(CloudGemFramework::JsonWriter& writer, const int& item)
 42      {
 43          return writer.Int(item);
 44      }
 45
 46      bool WriteJson(CloudGemFramework::JsonWriter& writer, const AZStd::string& item)
 47      {
 48          return writer.String(item);
 49      }
 50
 51      bool WriteJson(CloudGemFramework::JsonWriter& writer, const float& item)
 52      {
 53          return writer.Double(static_cast<double>(item));
 54      }
 55
 56      struct PutAdminMessagesReturnType
 57      {
 58
 59          AZ_TYPE_INFO(PutAdminMessagesReturnType,
 LmbrAWS_CodeGen_PutAdminMessagesReturnType_UUID)
 60          AZ_CLASS_ALLOCATOR(PutAdminMessagesReturnType, AZ::SystemAllocator, 0)
 61
 62          bool OnJsonKey(const char* key, CloudGemFramework::JsonReader& reader);
 63          static void Reflect(AZ::ReflectContext* reflection);
 64
 65      };
 66
 67      bool WriteJson(CloudGemFramework::JsonWriter& writer, const
 PutAdminMessagesReturnType& item)
 68      {
 69              bool ok = true;
```

```
70              ok = ok && writer.StartObject();
71              ok = ok && writer.EndObject();
72              return ok;
73      }
74
75      struct DetailedMessageData
76      {
77
78          AZ_TYPE_INFO(DetailedMessageData, LmbrAWS_CodeGen_DetailedMessageData_UUID)
79          AZ_CLASS_ALLOCATOR(DetailedMessageData, AZ::SystemAllocator, 0)
80
81          int priority;
82          AZStd::string message;
83          AZStd::string endTime;
84          AZStd::string UniqueMsgID;
85          AZStd::string startTime;
86
87          bool OnJsonKey(const char* key, CloudGemFramework::JsonReader& reader);
88          static void Reflect(AZ::ReflectContext* reflection);
89
90      };
91
92      bool WriteJson(CloudGemFramework::JsonWriter& writer, const DetailedMessageData&
item)
93      {
94              bool ok = true;
95              ok = ok && writer.StartObject();
96              ok = ok && writer.Key("priority");
97              ok = ok && WriteJson(writer, item.priority);
98              ok = ok && writer.Key("message");
99              ok = ok && WriteJson(writer, item.message);
100             ok = ok && writer.Key("endTime");
101             ok = ok && WriteJson(writer, item.endTime);
102             ok = ok && writer.Key("UniqueMsgID");
103             ok = ok && WriteJson(writer, item.UniqueMsgID);
104             ok = ok && writer.Key("startTime");
105             ok = ok && WriteJson(writer, item.startTime);
106             ok = ok && writer.EndObject();
107             return ok;
108     }
109
110     using DetailedMessageListPropertyList = AZStd::vector<DetailedMessageData>;
111
112     bool WriteJson(CloudGemFramework::JsonWriter& writer, const
 DetailedMessageListPropertyList& list)
113     {
114         bool ok = true;
115         ok = ok && writer.StartArray();
116         for (auto item : list)
117         {
118             ok = ok && WriteJson(writer, item);
119         }
120         ok = ok && writer.EndArray();
121         return ok;
122     }
123
124     struct DetailedMessageList
125     {
126
127         AZ_TYPE_INFO(DetailedMessageList, LmbrAWS_CodeGen_DetailedMessageList_UUID)
128         AZ_CLASS_ALLOCATOR(DetailedMessageList, AZ::SystemAllocator, 0)
129
130         DetailedMessageListPropertyList list;
131
132         bool OnJsonKey(const char* key, CloudGemFramework::JsonReader& reader);
133         static void Reflect(AZ::ReflectContext* reflection);
```

```
134
135     };
136
137     bool WriteJson(CloudGemFramework::JsonWriter& writer, const DetailedMessageList&
 item)
138     {
139             bool ok = true;
140             ok = ok && writer.StartObject();
141
142             ok = ok && writer.Key("list");
143             ok = ok && WriteJson(writer, item.list);
144
145             ok = ok && writer.EndObject();
146             return ok;
147     }
148
149     // Service RequestJobs
150     CLOUD_GEM_SERVICE(CloudGemMessageOfTheDay);
151
152     void Configure()
153     {
154         // Insert any necessary CloudGemFramework configuration here
155     }
156
157     class PutAdminMessagesRequest
158         : public CloudGemFramework::ServiceRequest
159     {
160     public:
161         SERVICE_REQUEST(CloudGemMessageOfTheDay, HttpMethod::HTTP_PUT, "/admin/
messages/{msg_id}");
162
163         struct Parameters
164         {
165
166             AZStd::string msg_id;
167
168             MessageData msg;
169
170
171             bool BuildRequest(CloudGemFramework::RequestBuilder& request)
172             {
173                 bool ok = true;
174
175                 ok = ok && request.SetPathParameter("{""msg_id""}", msg_id);
176
177
178                 ok = ok && request.WriteJsonBodyParameter(*this);
179                 return ok;
180             }
181
182             bool WriteJson(CloudGemFramework::JsonWriter& writer) const
183             {
184                 bool ok = true;
185
186                 ok = ok && CloudGemMessageOfTheDay::ServiceAPI::WriteJson(writer,
 msg);
187
188                 return ok;
189             }
190         };
191
192
193         PutAdminMessagesReturnType result;
194
195
196         Parameters parameters;
```

```
197      };
198
199
200      using PutAdminMessagesRequestJob =
 CloudGemFramework::ServiceRequestJob<PutAdminMessagesRequest>;
201
202      class GetAdminMessagesRequest
203          : public CloudGemFramework::ServiceRequest
204      {
205
206      public:
207
208          SERVICE_REQUEST(CloudGemMessageOfTheDay, HttpMethod::HTTP_GET, "/admin/
messages");
209
210          struct Parameters
211          {
212
213              int index;
214              int count;
215              AZStd::string filter;
216
217              bool BuildRequest(CloudGemFramework::RequestBuilder& request)
218              {
219                  bool ok = true;
220                  ok = ok && request.AddQueryParameter("index", index);
221                  ok = ok && request.AddQueryParameter("count", count);
222                  ok = ok && request.AddQueryParameter("filter", filter);
223                  ok = ok && request.WriteJsonBodyParameter(*this);
224                  return ok;
225              }
226
227              bool WriteJson(CloudGemFramework::JsonWriter& writer) const
228              {
229                  bool ok = true;
230                  return ok;
231              }
232
233          };
234
235          DetailedMessageList result;
236          Parameters parameters;
237
238      };
239
240      using GetAdminMessagesRequestJob =
 CloudGemFramework::ServiceRequestJob<GetAdminMessagesRequest>;
241
242      // Notification bus for this component
243      class CloudGemMessageOfTheDayNotifications
244          : public AZ::ComponentBus
245      {
246      public:
247
248          /**
249           * Sent when the request is a success
250           *
251           * Params:
252           *    jsonOutput:       The output receieved from the Lambda call
253           *    request:          The AWS Lambda request object
254           */
255          virtual void OnPutAdminMessagesRequestSuccess(const PutAdminMessagesReturnType
 response) { }
256
257          /**
258           * Sent when the request fails
```

```
259           *
260           * Params:
261           *    error:           The output receieved from the Lambda call;
262           *                     could be function error or an issue with the request
263           *    request:         The AWS Lambda request object
264           */
265          virtual void OnPutAdminMessagesRequestError(const CloudGemFramework::Error
 error) { }
266
267          /**
268           * Sent when the request is a success
269           *
270           * Params:
271           *    jsonOutput:       The output receieved from the Lambda call
272           *    request:          The AWS Lambda request object
273           */
274
275          virtual void OnGetAdminMessagesRequestSuccess(const DetailedMessageList
 response) { }
276
277          /**
278           * Sent when the request fails
279           *
280           * Params:
281           *    error:           The output receieved from the Lambda call;
282           *                     could be function error or an issue with the request
283           *    request:         The AWS Lambda request object
284           */
285          virtual void OnGetAdminMessagesRequestError(const CloudGemFramework::Error
 error) { }
286
287      };
288
289      using CloudGemMessageOfTheDayNotificationBus =
 AZ::EBus<CloudGemMessageOfTheDayNotifications>;
290
291      class BehaviorCloudGemMessageOfTheDayNotificationBusHandler
292          : public CloudGemMessageOfTheDayNotificationBus::Handler, public
 AZ::BehaviorEBusHandler
293      {
294      public:
295          AZ_EBUS_BEHAVIOR_BINDER(BehaviorCloudGemMessageOfTheDayNotificationBusHandler,
 LmbrAWS_CodeGen_NotificationBus1_UUID, AZ::SystemAllocator
296          , OnPutAdminMessagesRequestSuccess
297          , OnPutAdminMessagesRequestError
298          , OnGetAdminMessagesRequestSuccess
299          , OnGetAdminMessagesRequestError
300          );
301
302          void OnPutAdminMessagesRequestSuccess(const PutAdminMessagesReturnType
 response) override
303          {
304              Call(FN_OnPutAdminMessagesRequestSuccess, response);
305          }
306
307          void OnPutAdminMessagesRequestError(const CloudGemFramework::Error error)
 override
308          {
309              Call(FN_OnPutAdminMessagesRequestError, error);
310          }
311
312          void OnPostAdminMessagesRequestSuccess(const DetailedMessageData response)
 override
313          {
314              Call(FN_OnPostAdminMessagesRequestSuccess, response);
315          }
```

```
316
317          void OnPostAdminMessagesRequestError(const CloudGemFramework::Error error)
 override
318          {
319              Call(FN_OnPostAdminMessagesRequestError, error);
320          }
321
322      };
323
324      class CloudGemMessageOfTheDayResponseHandler;
325
326      // Request bus for this component
327      class CloudGemMessageOfTheDayRequests
328          : public AZ::ComponentBus
329      {
330
331      public:
332
333          virtual ~CloudGemMessageOfTheDayRequests() {}
334
335          virtual void PutAdminMessages(const AZStd::string& msg_id, const MessageData&
 msg, CloudGemMessageOfTheDayResponseHandler* responseHandler) { }
336          virtual void GetAdminMessages(const int& index, const int& count, const
 AZStd::string& filter, CloudGemMessageOfTheDayResponseHandler* responseHandler) { }
337
338      };
339
340      using CloudGemMessageOfTheDayRequestBus =
 AZ::EBus<CloudGemMessageOfTheDayRequests>;
341
342      // This class is used as a parameter for all requests and throws the response on
 the CloudGemMessageOfTheDayNotificationBus
343      // Inherit from this to define custom behavior.
344      class CloudGemMessageOfTheDayResponseHandler
345      {
346      public:
347          AZ_TYPE_INFO(CloudGemMessageOfTheDayResponseHandler,
 LmbrAWS_CodeGen_ResponseHandler_UUID)
348          AZ_CLASS_ALLOCATOR(CloudGemMessageOfTheDayResponseHandler,
 AZ::SystemAllocator, 0)
349
350          virtual ~CloudGemMessageOfTheDayResponseHandler() {}
351
352          virtual void HandlePutAdminMessagesSuccess(PutAdminMessagesRequestJob* job,
 AZ::Entity* entity)
353          {
354              EBUS_EVENT_ID(entity->GetId(), CloudGemMessageOfTheDayNotificationBus,
 OnPutAdminMessagesRequestSuccess, job->result);
355          }
356
357          virtual void HandlePutAdminMessagesError(PutAdminMessagesRequestJob* job,
 AZ::Entity* entity)
358          {
359              EBUS_EVENT_ID(entity->GetId(), CloudGemMessageOfTheDayNotificationBus,
 OnPutAdminMessagesRequestError, job->error);
360          }
361
362          virtual void HandleGetAdminMessagesSuccess(GetAdminMessagesRequestJob* job,
 AZ::Entity* entity)
363          {
364              EBUS_EVENT_ID(entity->GetId(), CloudGemMessageOfTheDayNotificationBus,
 OnGetAdminMessagesRequestSuccess, job->result);
365          }
366
367          virtual void HandleGetAdminMessagesError(GetAdminMessagesRequestJob* job,
 AZ::Entity* entity)
```

```
368          {
369              EBUS_EVENT_ID(entity->GetId(), CloudGemMessageOfTheDayNotificationBus,
 OnGetAdminMessagesRequestError, job->error);
370          }
371
372      };
373
374      class CloudGemMessageOfTheDayClientComponent
375          : public AZ::Component
376          , public CloudGemMessageOfTheDayRequestBus::Handler
377      {
378      public:
379          AZ_COMPONENT(CloudGemMessageOfTheDayClientComponent,
 LmbrAWS_CodeGen_Component_UUID);
380          virtual ~CloudGemMessageOfTheDayClientComponent() = default;
381
382          AZStd::scoped_ptr<CloudGemMessageOfTheDayResponseHandler>
 m_defaultResponseHandler;
383
384          void Init() override
385          {
386              m_defaultResponseHandler.reset(new
 CloudGemMessageOfTheDayResponseHandler());
387              CloudGemMessageOfTheDay::ServiceAPI::Configure();
388          }
389
390          void Activate() override
391          {
392              CloudGemMessageOfTheDayRequestBus::Handler::BusConnect(m_entity->GetId());
393          }
394
395          void Deactivate() override
396          {
397              CloudGemMessageOfTheDayRequestBus::Handler::BusDisconnect();
398          }
399
400          static void Reflect(AZ::ReflectContext* reflection)
401          {
402              MessageData::Reflect(reflection);
403              PutAdminMessagesReturnType::Reflect(reflection);
404              DetailedMessageData::Reflect(reflection);
405              DetailedMessageList::Reflect(reflection);
406              MessageList::Reflect(reflection);
407
408              AZ::SerializeContext* serializeContext =
 azrtti_cast<AZ::SerializeContext*>(reflection);
409              if (serializeContext)
410              {
411                  // we must include any fields we want to expose to the editor or lua
 in the serialize context
412                  serializeContext->Class<CloudGemMessageOfTheDayClientComponent>()
413                      ->Version(1);
414
415                  AZ::EditContext* editContext = serializeContext->GetEditContext();
416                  if (editContext)
417                  {
418                      editContext-
>Class<CloudGemMessageOfTheDayClientComponent>("CloudGemMessageOfTheDayClientComponent",
 "CloudGemMessageOfTheDay Component")
419                          ->ClassElement(AZ::Edit::ClassElements::EditorData, "")
420                              ->Attribute(AZ::Edit::Attributes::Category, "Cloud Canvas
 Gems")
421                              -
>Attribute(AZ::Edit::Attributes::AppearsInAddComponentMenu, AZ_CRC("Game"));
422                  }
423              }
```

```
424
425             AZ::BehaviorContext* behaviorContext =
 azrtti_cast<AZ::BehaviorContext*>(reflection);
426             if (behaviorContext)
427             {
428                 behaviorContext-
>EBus<CloudGemMessageOfTheDayRequestBus>("CloudGemMessageOfTheDayRequestBus")
429                     // one of these for each function
430                     ->Event("PutAdminMessages",
 &CloudGemMessageOfTheDayRequestBus::Events::PutAdminMessages)
431                     ->Event("GetAdminMessages",
 &CloudGemMessageOfTheDayRequestBus::Events::GetAdminMessages)
432                     ;
433                 behaviorContext-
>EBus<CloudGemMessageOfTheDayNotificationBus>("CloudGemMessageOfTheDayNotificationBus")
434                     ->Handler<BehaviorCloudGemMessageOfTheDayNotificationBusHandler>()
435                 ;
436             }
437         }
438
439         // Functions from the swagger definitions
440
441
442         void PutAdminMessages(const AZStd::string& msg_id, const MessageData& msg,
 CloudGemMessageOfTheDayResponseHandler* responseHandler) override
443         {
444
445
446             if (responseHandler == nullptr)
447             {
448                 responseHandler = AZStd::get_pointer(m_defaultResponseHandler);
449             }
450
451             // create job
452             PutAdminMessagesRequestJob* job = PutAdminMessagesRequestJob::Create(
453                 [responseHandler, this](PutAdminMessagesRequestJob* job)
454                 {
455                     // handle success
456                     responseHandler->HandlePutAdminMessagesSuccess(job, m_entity);
457                 },
458                 [responseHandler, this](PutAdminMessagesRequestJob* job)
459                 {
460                     // handle error
461                     responseHandler->HandlePutAdminMessagesError(job, m_entity);
462                 }
463             );
464
465             job->parameters.msg = msg;
466             job->parameters.msg_id = msg_id;
467
468             job->Start();
469
470         }
471
472         void GetAdminMessages(const int& index, const int& count, const AZStd::string&
 filter, CloudGemMessageOfTheDayResponseHandler* responseHandler) override
473         {
474
475             if (responseHandler == nullptr)
476             {
477                 responseHandler = AZStd::get_pointer(m_defaultResponseHandler);
478             }
479
480             // create job
481             GetAdminMessagesRequestJob* job = GetAdminMessagesRequestJob::Create(
482                 [responseHandler, this](GetAdminMessagesRequestJob* job)
```

```
483                 {
484                     // handle success
485                     responseHandler->HandleGetAdminMessagesSuccess(job, m_entity);
486                 },
487                 [responseHandler, this](GetAdminMessagesRequestJob* job)
488                 {
489                     // handle error
490                     responseHandler->HandleGetAdminMessagesError(job, m_entity);
491                 }
492             );
493 
494             job->parameters.index = index;
495             job->parameters.count = count;
496             job->parameters.filter = filter;
497 
498             job->Start();
499 
500         }
501 
502     };
503 
504     bool MessageData::OnJsonKey(const char* key, CloudGemFramework::JsonReader&
 reader)
505     {
506 
507         if (strcmp(key, "priority") == 0) return reader.Accept(priority);
508 
509         if (strcmp(key, "message") == 0) return reader.Accept(message);
510 
511         if (strcmp(key, "endTime") == 0) return reader.Accept(endTime);
512 
513         if (strcmp(key, "startTime") == 0) return reader.Accept(startTime);
514 
515         return reader.Ignore();
516     }
517 
518     void MessageData::Reflect(AZ::ReflectContext* reflection)
519     {
520         AZ::SerializeContext* serializeContext =
 azrtti_cast<AZ::SerializeContext*>(reflection);
521 
522         if (serializeContext)
523         {
524             serializeContext->Class<MessageData>()
525                 ->Version(1);
526         }
527 
528         AZ::BehaviorContext* behaviorContext =
 azrtti_cast<AZ::BehaviorContext*>(reflection);
529         if (behaviorContext)
530         {
531             behaviorContext->Class<MessageData>("CloudGemMessageOfTheDay_MessageData")
532                 ->Attribute(AZ::Script::Attributes::Storage,
 AZ::Script::Attributes::StorageType::Value)
533 
534                 ->Property("priority", BehaviorValueProperty(&MessageData::priority))
535 
536                 ->Property("message", BehaviorValueProperty(&MessageData::message))
537 
538                 ->Property("endTime", BehaviorValueProperty(&MessageData::endTime))
539 
540                 ->Property("startTime",
 BehaviorValueProperty(&MessageData::startTime))
541 
542             ;
543         }
```

```
544      }
545
546      bool PutAdminMessagesReturnType::OnJsonKey(const char* key,
 CloudGemFramework::JsonReader& reader)
547      {
548          return reader.Ignore();
549      }
550
551      void PutAdminMessagesReturnType::Reflect(AZ::ReflectContext* reflection)
552      {
553          AZ::SerializeContext* serializeContext =
 azrtti_cast<AZ::SerializeContext*>(reflection);
554
555          if (serializeContext)
556          {
557              serializeContext->Class<PutAdminMessagesReturnType>()
558                  ->Version(1);
559          }
560
561          AZ::BehaviorContext* behaviorContext =
 azrtti_cast<AZ::BehaviorContext*>(reflection);
562          if (behaviorContext)
563          {
564              behaviorContext-
>Class<PutAdminMessagesReturnType>("CloudGemMessageOfTheDay_PutAdminMessagesReturnType")
565                  ->Attribute(AZ::Script::Attributes::Storage,
 AZ::Script::Attributes::StorageType::Value)
566
567              ;
568          }
569      }
570
571      bool DetailedMessageData::OnJsonKey(const char* key,
 CloudGemFramework::JsonReader& reader)
572      {
573          if (strcmp(key, "priority") == 0) return reader.Accept(priority);
574          if (strcmp(key, "message") == 0) return reader.Accept(message);
575          if (strcmp(key, "endTime") == 0) return reader.Accept(endTime);
576          if (strcmp(key, "UniqueMsgID") == 0) return reader.Accept(UniqueMsgID);
577          if (strcmp(key, "startTime") == 0) return reader.Accept(startTime);
578          return reader.Ignore();
579      }
580
581      void DetailedMessageData::Reflect(AZ::ReflectContext* reflection)
582      {
583
584          AZ::SerializeContext* serializeContext =
 azrtti_cast<AZ::SerializeContext*>(reflection);
585
586          if (serializeContext)
587          {
588              serializeContext->Class<DetailedMessageData>()->Version(1);
589          }
590
591          AZ::BehaviorContext* behaviorContext =
 azrtti_cast<AZ::BehaviorContext*>(reflection);
592          if (behaviorContext)
593          {
594              behaviorContext-
>Class<DetailedMessageData>("CloudGemMessageOfTheDay_DetailedMessageData")
595                  ->Attribute(AZ::Script::Attributes::Storage,
 AZ::Script::Attributes::StorageType::Value)
596                  ->Property("priority",
 BehaviorValueProperty(&DetailedMessageData::priority))
597                  ->Property("message",
 BehaviorValueProperty(&DetailedMessageData::message))
```

```
598                     ->Property("endTime",
 BehaviorValueProperty(&DetailedMessageData::endTime))
599                     ->Property("UniqueMsgID",
 BehaviorValueProperty(&DetailedMessageData::UniqueMsgID))
600                     ->Property("startTime",
 BehaviorValueProperty(&DetailedMessageData::startTime))
601                 ;
602             }
603
604         }
605
606         bool DetailedMessageList::OnJsonKey(const char* key,
 CloudGemFramework::JsonReader& reader)
607         {
608             if (strcmp(key, "list") == 0) return reader.Accept(list);
609             return reader.Ignore();
610         }
611
612         void DetailedMessageList::Reflect(AZ::ReflectContext* reflection)
613         {
614
615             AZ::SerializeContext* serializeContext =
 azrtti_cast<AZ::SerializeContext*>(reflection);
616
617             if (serializeContext)
618             {
619                 serializeContext->Class<DetailedMessageList>()->Version(1);
620             }
621
622             AZ::BehaviorContext* behaviorContext =
 azrtti_cast<AZ::BehaviorContext*>(reflection);
623             if (behaviorContext)
624             {
625                 behaviorContext-
>Class<DetailedMessageList>("CloudGemMessageOfTheDay_DetailedMessageList")
626                     ->Attribute(AZ::Script::Attributes::Storage,
 AZ::Script::Attributes::StorageType::Value)
627                     ->Property("list", BehaviorValueProperty(&DetailedMessageList::list))
628                 ;
629             }
630
631         }
632
633     } // ServiceAPI
634     } // CloudGemMessageOfTheDay
```

Examining the Generated Game Client Code

The following table provides additional information for the generated game client code example in the previous section. The line numbers indicate the location in the code that the comments refer to.

Line Number	Description
24, 25	The C++ namespace for the API. This is always *<resource-group-name>*`::ServiceApi`.
56, 75, 110, 124	The `PutAdminMessageReturnType` struct contains the PUT operation's return data as defined by the swagger file. In this case, it is an empty object and has no properties.
67, 92, 112, 137	The `WriteJson` method is generated to handle the serialization of `PutAdminMessageReturnType` objects to JSON text format. JSON format is required for the data to be sent to the service. A

Line Number	Description
	similar function is generated for each of the type definitions in the `swagger.json` file.
150	The `CLOUD_GEM_SERVICE` macro defines a class that provides information that is common to all of the service's requests.

152	The `Configure` function is called when the generated component is initialized. You can add code to this function to change the default configuration for the service. For example, the following code increases the timeout for `GetServiceStatus` requests.
	`void Configure()`
	`{`
	`GetServiceStatusRequestJob::GetDefaultConfig()->requestTimeoutMs = 20000;`
	`}`
	Note
	If you regenerate the client, your changes to the `Configure` method are lost.

157, 202	A `Request` class such as `PutAdminMessagesRequest` is generated for each of the operations defined in the `swagger.json` file. This class encapsulates the HTTP method and path that are used to make the request. The `Parameters` struct in the class defines the fields for each of the request's parameters that are specified by the swagger file.
200, 240	The Lumberyard job system executes API requests asynchronously. This work is performed by the `CloudGemFramework::ServiceRequestJob` class and specialized by the `PutAdminMessagesRequest` class declared previously. The C++ `using` statement creates an alias for that type.
243	Defines the `CloudGemMessageOfTheDayNotifications` EBus that signals when a request has completed. Both a success and an error notification method are defined for each operation specified in the `swagger.json` file.
291	Defines a handler (`BehaviorCloudGemMessageOfTheDayNotificationBusHandler`) for the notification EBus that forwards notifications to Lua code.
340	Defines an EBus (`CloudGemMessageOfTheDayRequestBus`) that initiates requests.
344	Defines a class (`CloudGemMessageOfTheDayResponseHandler`) that handles responses. By default, responses are dispatched to the notification EBus previously defined.
374	Defines a client component (`CloudGemMessageOfTheDayClientComponent`) that uses the using the request job classes `PutAdminMessagesRequest` (line 157) and `GetAdminMessagesRequest` (line 202) to implement the request EBus handler `BehaviorCloudGemMessageOfTheDayNotificationBusHandler` (line 291).

Calling a Game API

To call your API operations, you can use C++, the client component, a request job, or Lua.

Topics

- Calling an API from C++ (p. 376)
- Using Lua to Call an API (p. 378)

Calling an API from C++

To invoke a service API from C++, you can use an EBus event handler from the generated client component. You can also use the generated service request job class directly. If you want to make requests from a component, consider using the client component. If you want to perform a sequence of operations, consider using the job class.

Using the Client Component to Call an API

The following example from the `CloudGemLeaderboard` service API shows a component that invokes `GetServiceStatusRequestJob`. There are two ways to receive the request's response:

1. Use an object that inherits from the service's response handler.
2. Connect to the generated component's notification bus.

The example shows both methods.

```
// Sample API Caller
#include <AzCore/Component/Component.h>
#include <AzCore/Component/Entity.h>
#include <AzCore/Serialization/EditContext.h>
#include <AzCore/Serialization/SerializeContext.h>
#include <AzCore/EBus/EBus.h>


#include <AWS/ServiceAPI/CloudGemLeaderboardClientComponent.h>
#include <CloudGemLeaderboard/CloudGemLeaderboardBus.h>
class SampleResponseHandler
    : public CloudGemLeaderboard::ServiceAPI::CloudGemLeaderboardResponseHandler
{
public:
    void
 HandleGetServiceStatusSuccess(CloudGemLeaderboard::ServiceAPI::GetServiceStatusRequestJob*
 job, AZ::Entity* entity) override
    {
        //Look at job->result for response data
        AZ_Printf("Got response: %s", job->result.status.c_str());
    }

    void
 HandleGetServiceStatusError(CloudGemLeaderboard::ServiceAPI::GetServiceStatusRequestJob*
 job, AZ::Entity* entity) override
    {
        //Look at job->error for error data
    }
};

class SampleAPICallerComponent
    : public AZ::Component
    , public CloudGemLeaderboard::ServiceAPI::CloudGemLeaderboardNotificationBus::Handler
{
public:
    SampleResponseHandler* m_responseHandler;
```

```
    AZ_COMPONENT(SampleAPICallerComponent, "{aedd6408-e2f0-4250-a181-b0ef41085a94}");
    virtual ~SampleAPICallerComponent() = default;
    static void Reflect(AZ::ReflectContext* reflection)
    {
        AZ::SerializeContext* serializeContext =
 azrtti_cast<AZ::SerializeContext*>(reflection);
        if (serializeContext)
        {
            // We must include any fields we want to expose to the editor or Lua in the
 serialize context
            serializeContext->Class<SampleAPICallerComponent>()
                ->Version(1);

            AZ::EditContext* editContext = serializeContext->GetEditContext();
            if (editContext)
            {
                editContext->Class<SampleAPICallerComponent>("SampleAPICallerComponent",
 "Component to call CloudGemLeaderboard GetServiceStatus")
                    ->ClassElement(AZ::Edit::ClassElements::EditorData, "")
                        ->Attribute(AZ::Edit::Attributes::AppearsInAddComponentMenu,
 AZ_CRC("Game"));
            }
        }
    }
    void Activate() override
    {

 CloudGemLeaderboard::ServiceAPI::CloudGemLeaderboardNotificationBus::Handler::BusConnect(m_entity-
>GetId());
        CallGetServiceStatus();
    }

    void CallGetServiceStatus()
    {
        // To get the response on the CloudGemLeaderboardNotificationBus
        EBUS_EVENT_ID(m_entity->GetId(),
 CloudGemLeaderboard::ServiceAPI::CloudGemLeaderboardRequestBus, GetServiceStatus,
 nullptr);
        // To get the response in a special response handler
        m_responseHandler = new SampleResponseHandler();
        EBUS_EVENT_ID(m_entity->GetId(),
 CloudGemLeaderboard::ServiceAPI::CloudGemLeaderboardRequestBus, GetServiceStatus,
 m_responseHandler);
    }

    void OnGetServiceStatusRequestSuccess(const
 CloudGemLeaderboard::ServiceAPI::ServiceStatus response) override
    {
        // handle success
        AZ_Printf("Got response: %s", response.status.c_str());
    }

    void OnGetServiceStatusRequestError(const CloudGemFramework::Error error) override
    {
        // handle failure
    }

    void Deactivate() override
    {

 CloudGemLeaderboard::ServiceAPI::CloudGemLeaderboardNotificationBus::Handler::BusDisconnect();
        delete m_responseHandler;
    }
};
```

Using the Request Job to Call an API

In some situations you might want more control over how your background jobs are executed. For example, you might want to change the thread pool that certain requests use or perform multiple requests on a background thread. You can use the generated request job classes directly to make these changes.

In its simplest form, you can use the request job class in the following example. This is the same code that is in line 452 of the generated request component.

```
PutAdminMessagesRequestJob* job = PutAdminMessagesRequestJob::Create(
    [responseHandler, this](PutAdminMessagesRequestJob* job)
    {
        // Handle success
        responseHandler->HandlePutAdminMessagesSuccess(job, m_entity);
    },
    [responseHandler, this](PutAdminMessagesRequestJob* job)
    {
        // Handle error
        responseHandler->HandlePutAdminMessagesError(job, m_entity);
    }
);

job->parameters.msg = msg;
job->parameters.msg_id = msg_id;

job->Start();
```

To learn about other ways to use the request job class, explore the `ServiceApiRequestJob` class definition and see Running AWS API Jobs Using the Cloud Gem Framework (p. 350).

Using Lua to Call an API

The following code example shows how to call a service API from a Lua script.

```
-- Service status getter
local leaderboardstatusgetter = {
    Properties = {
    }
}

function leaderboardstatusgetter:OnActivate()
    self.notificationHandler = CloudGemLeaderboardNotificationBus.Connect(self,
 self.entityId)
    CloudGemLeaderboardRequestBus.Event.GetServiceStatus(self.entityId, nil)
end

function leaderboardstatusgetter:OnDeactivate()
    self.notificationHandler:Disconnect()
end

function leaderboardstatusgetter:OnGetServiceStatusRequestSuccess(response)
    Debug.Log(response.status)
end

function leaderboardstatusgetter:OnGetServiceStatusRequestError(error)
    Debug.Log(error.message)
end

return leaderboardstatusgetter
```

Publishing Your API

Service API operations are implemented by API Gateway, which has an open source serverless web application that you can use to create your own developer portal. A developer portal on API Gateway empowers you to do the following:

- List your API operations in catalog form.
- Allow developers to sign up.
- Display documentation that helps developers understand your API.
- Let developers test your API and provide feedback.
- Grow a developer ecosystem.
- Monetize your API and grow API product revenue.

For more information, see Generate Your Own API Gateway Developer Portal.

Using the Cloud Gem Framework Command Line

You can use the `lmbr_aws cloud-gem-framework` command line for working with Cloud Canvas Cloud Gems (p. 200), the Cloud Gem Portal (p. 200), and the Cloud Gem Framework Service API (p. 352).

> **Note**
> For general information on the `lmbr_aws` command line tool, including common syntax, configuration, and arguments, see Using the Cloud Canvas Command Line (p. 408).

Commands

Following are details of the `lmbr_aws cloud-gem-framework` commands.

add-service-api-resources

Adds the resources to implement a `swagger.json` file-based service API to a resource group.

The `add-service-api-resources` command accepts the following arguments:

- `--resource-group` *`{resource-group-name}`* or `-r` *`{resource-group-name}`*

  Required. The name of the resource group or cloud gem.
- `--force`

  Optional. Forces the replacement of existing resource definitions. By default, existing resource definitions with the same names are not changed.

For more information about the `add-service-api-resources` command, see Resources (p. 354) in the Cloud Gem Framework Service API (p. 352) documentation and Getting Started with the Cloud Gem Framework (p. 305).

generate-service-api-code

Generates component and Lambda function code to support a service API described by a `swagger.json` file that a resource group provides.

The `generate-service-api-code` command accepts the following arguments:

- `--resource-group` *`{resource-group-name}`* or `-r` *`{resource-group-name}`*

  Required. The name of a resource group or cloud gem that provides a `swagger.json` file.

- `--component-client-path` *`{component-client-path}`*

  Optional. The output path for the component client code. The default location is the `Code\`*`{game}`*`\AWS\`*`{group}`*`\ServiceAPI` directory. If a gem defines the resource group, the default location is the `Gem\Code\Include\AWS\ServiceAPI` directory.

For more information about the `generate-service-api-code` command, see Generating a Game Client and Getting Started with the Cloud Gem Framework (p. 305).

open-cloud-gem-portal, open-portal, cloud-gem-portal

Generate a presigned URL and open the Cloud Gem Portal in the default browser using the presigned URL.

The `open-cloud-gem-portal` command and its equivalents accept the following arguments:

- `--deployment` *`{deployment-name}`* or `-d` *`{deployment-name}`*

  Optional. If the *`{role-name}`* that is specified in the `--role` argument is a deployment access role, identifies the deployment used. If the `--deployment` argument is not specified, the default deployment for the project is used.
- `--duration-seconds` *`{duration-in-seconds}`*

  Optional. Specifies the number of seconds before the URL and temporary credentials expire. You can use this argument to override the session length. For more information, see Setting Up and Accessing the Cloud Gem Portal (p. 201).
- `--show-current-encryption-object`

  Optional. Displays the active Cloud Gem Portal AES passphrase and initialization vector without generating a new encrypted payload.
- `--show-encryption-object`

  Optional. Displays the newly generated AES passphrase and initialization vector of the encryption object that the Cloud Gem Portal uses for decrypting data that Lumberyard sends.
- `--role` *`{role-name}`*

  Specifies an IAM role to be assumed by the Cloud Gem Portal website. Can be `ProjectOwner`, `DeploymentOwner`, or any other project or deployment access role. The credentials taken from the `~/.aws/credentials` file must be able to assume the role that is specified.

remove-service-api-resources

Remove the resources that implement a `swagger.json` file–based service API operation from a resource group.

The `remove-service-api-resources` command accepts the following arguments:

- `--resource-group` *`{resource-group-name}`* or `-r` *`{resource-group-name}`*

  Required. The name of the resource group.

service-api-process-swagger

Processes the Cloud Canvas extension objects that are defined in a `swagger.json` file and produces swagger definitions that can be imported into Amazon API Gateway.

The `service-api-process-swagger` command accepts the following arguments:

- `--resource-group` *`{resource-group-name}`* or `-r` *`{resource-group-name}`*

  Required. The name of the resource group.

- `--input` *`{file-path}`*

  Optional. The file from which the `swagger` JSON is read. For the cloud gems that are included with Lumberyard, the default is `dev\Gems\`*`{gem-name}`*`\swagger.json`.

- `--output` *`{file-path}`*

  Optional. The file to which the processed swagger JSON is written. By default, the output is written to `stdout` (the command line console window).

For more information about the `service-api-process-swagger` command, see Cloud Gem Framework Extension Object (p. 360).

upload-cloud-gem-portal, upload-portal

Upload all Cloud Gem Portal content.

The `upload-cloud-gem-portal` command and its `upload-portal` equivalent accept the following arguments:

- `--deployment` *`{deployment-name}`* or `-d` *`{deployment-name}`*

  Optional. The name of the deployment for which portal content will be uploaded. If not specified, the default deployment is updated.

- `--resource-group` *`{resource-group-name}`* or `-r` *`{resource-group-name}`*

  Optional. The name of the resource group to which portal content will be uploaded. The default is to upload portal content for all resource groups.

- `--project`

  Optional. Specifies that the global portal content of the project will be updated instead of the deployment and resource group content.

Administering Cloud Canvas

This section provides information for administering your Lumberyard project's AWS account and its use of the AWS cloud.

Topics

- Setting Up a Project to Use Resource Manager (p. 381)
- Working with Deployments (p. 385)
- Understanding the Resource Manager Security System (p. 393)

Setting Up a Project to Use Resource Manager

After initializing Cloud Canvas Resource Manager so that your Amazon Lumberyard project can use AWS, you can use Resource Manager to configure the stacks for your project.

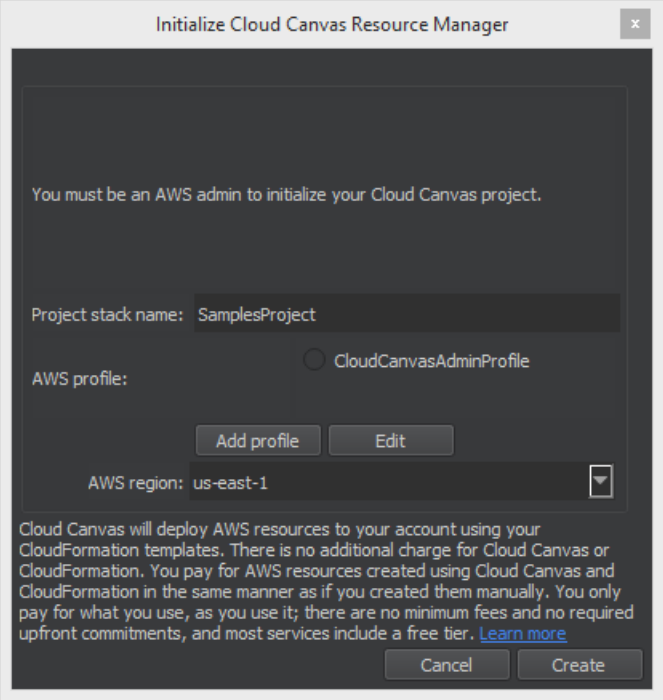
Topics

- Initializing Cloud Canvas Resource Manager (p. 382)

- Working with Project Stacks (p. 383)

Initializing **Cloud Canvas Resource Manager**

When you perform an operation that requires an AWS account, and no account has been associated with your Lumberyard project, the **Initialize Cloud Canvas Resource Manager** dialog prompts you for the required information.



To initialize Cloud Canvas Resource Manager

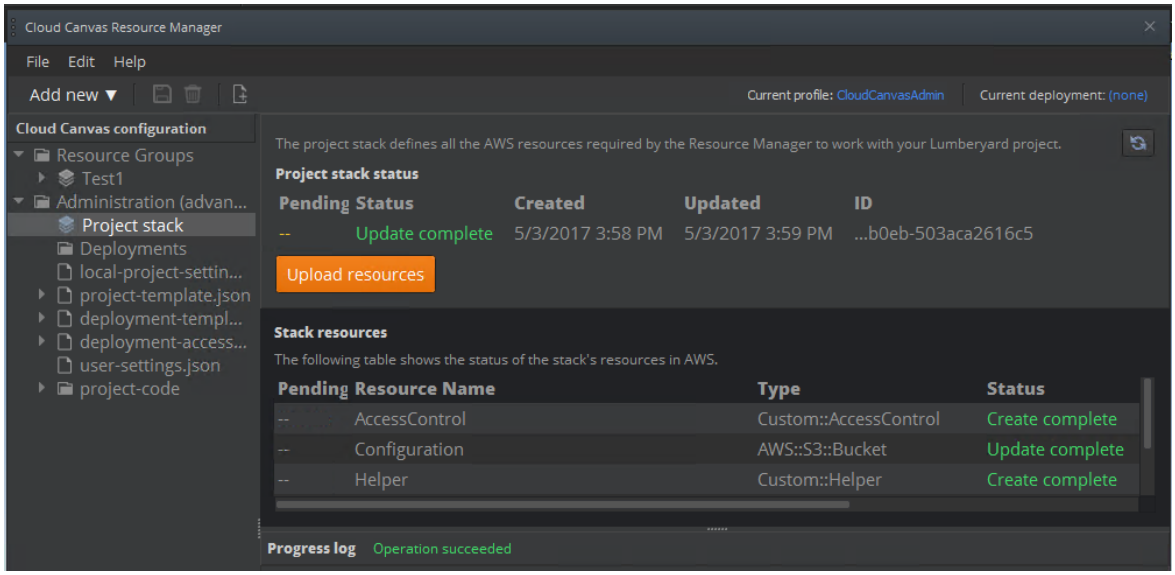
1. When prompted to initialize the **Cloud Canvas Resource Manager**, provide the following information:

   - For **Project stack name**, type the name of an AWS CloudFormation stack that you will create. The stack will contain the AWS resources that **Cloud Canvas Resource Manager** will use for your project. By default, Lumberyard uses the name of your project for the stack name. A stack with the name that you specify must not already exist in your AWS account for the region you select.
   - For **AWS Credentials**, select from the list of available profiles or create a new one. If you have no AWS profiles on your computer, you are prompted for an AWS secret key and an AWS access key. You can also edit an existing one.

     In order to use Lumberyard with AWS, you must provide administrative credentials for your AWS account either directly, or through an AWS profile. For information on how to get these credentials from AWS, see the Getting Started with Cloud Canvas Tutorial.
   - For **AWS region**, specify the AWS data center where your resources will reside. You must choose a region that supports all the AWS services that your game uses. The region you choose must also support the Amazon Cognito service, which Lumberyard uses to establish player identity, and AWS CloudFormation, which Lumberyard uses to create and manage resources. For more information about the capabilities of different regions, see AWS Regions and Endpoints.
2. Click **Create** to start the initialization process. In the navigation tree, the Working with Project Stacks (p. 383) node is selected, and in the detail pane, the Viewing the Cloud Canvas Progress Log (p. 279) shows the progress of the initialization.

Working with Project Stacks

When you select the **Project stack** node in the **Cloud Canvas Resource Manager** navigation pane, the detail pane shows information about the AWS CloudFormation stack that Cloud Canvas is using.



Note the following:

- If you select a project stack node and no AWS profile is configured, Lumberyard prompts you to provide one. A profile is required for Lumberyard to display the status of your project's resources. For more information, see Managing Cloud Canvas Profiles (p. 198).
- If you select the **Project stack** node when the project has not been initialized for use with Cloud Canvas, Lumberyard prompts you to initialize the project and create a project stack. For more information, see Initializing **Cloud Canvas Resource Manager** (p. 382).

Project Stack Status Table

The **Project stack status** table shows the status of the AWS CloudFormation stack that contains the resources used by your project's resource groups.

This table has the following columns:

Pending – Indicates when changes have been made to the local stack templates or code but not yet uploaded to AWS. Pausing your mouse pointer on a value in this column displays the reason why the change is pending. Following are the possible values:

- **Create** – The stack is defined locally but has not been created in AWS.
- **Update** – The stack's template and/or Lambda function code has been changed locally but not yet uploaded to AWS.
- **Delete** – The stack has been removed from the local definitions but has not been deleted in AWS. If you are using a source control system, this status can indicate that a new stack has been added to the project. However, your local copy of the source control is out of date and doesn't contain the definition for the new stack.
- **--** – No change is pending.

Status – The status of the AWS CloudFormation stack. See Understanding Resource Status Descriptions (p. 277) for a description of the values this column can have. To see additional status information, pause your pointer on the status indicator.

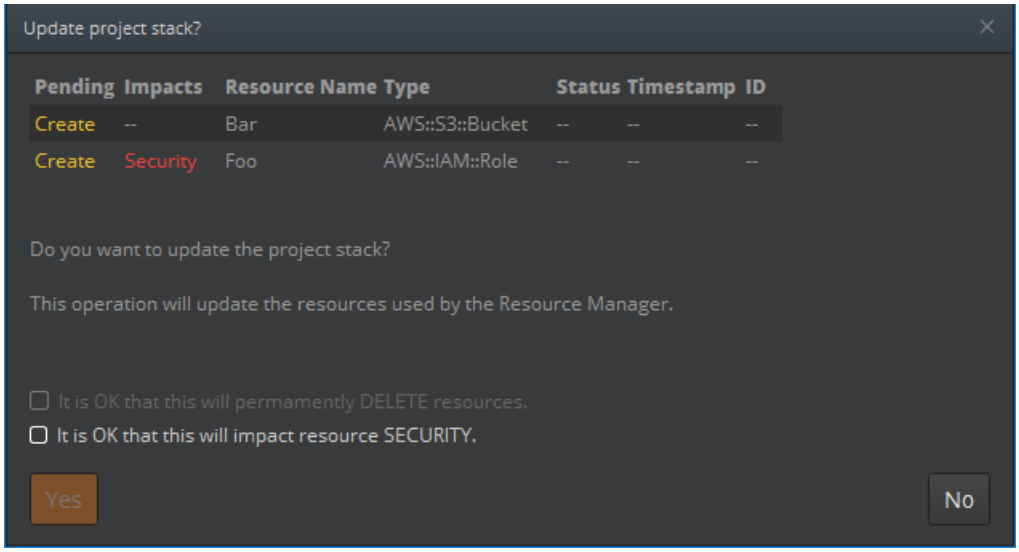
Created – The time the stack was created.

Updated – The time the stack status was updated.

ID - A truncated version of the AWS ID for the stack. To see the full ID, pause your pointer on the truncated ID.

Upload Resources

Click **Upload resources** to start the process of modifying, creating, or deleting resources in AWS so that they match your local definitions of them. A dialog box shows the changes that will be applied.



The table of changes has the same columns as the stack resource status table but also includes a column for **Impacts**. If a change has the potential to affect the security of your project, the **Impacts** column contains the text **Security**. Before you can start an operation that has a security impact, you must select the option **It is OK that this will impact resource SECURITY**. Review any security changes carefully before you apply them.

Similarly, before you can start an operation that deletes one or more resources, you must select the option **It is OK that this will permanently DELETE resources**.

Stack Resources Table

The **Stack resources** table shows the status of the resources that your project is using.

This table has the following columns:

Pending – Indicates that changes have been made locally but not yet uploaded to AWS. Pausing your mouse on a value in this column displays the reason why the change is pending. Following are the possible values:

- **Create** – The resource is defined locally but has not been created in AWS.
- **Update** – The resource has been changed locally but not yet uploaded to AWS.
- **Delete** – The resource has been removed from the local definitions but has not been deleted on AWS. If you are using a source control system, this status can indicate that a new resource has been added

to the project. However, your local copy of the source control is out of date and doesn't contain the definition for the new resource.

- **--** – No change is pending.

Resource Name – The logical name of the resource. You can reference the resource in flow graph nodes by adding this resource name to the resource group name.

Type – The type of the resource (for example, a Lambda function, Amazon S3 bucket, or a custom resource).

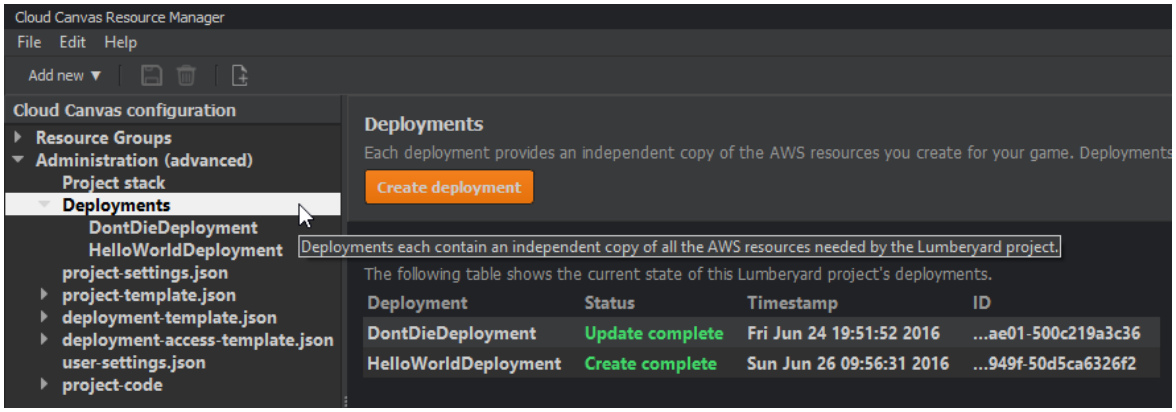
Status – The current condition of the resource. For a description of the possible status values, see Understanding Resource Status Descriptions (p. 277). To see additional status information, pause your pointer on the status.

Timestamp – The time of the most recent change.

ID - A truncated version of the AWS ID for the stack. To see the full ID, pause your pointer on the truncated ID.

Working with Deployments

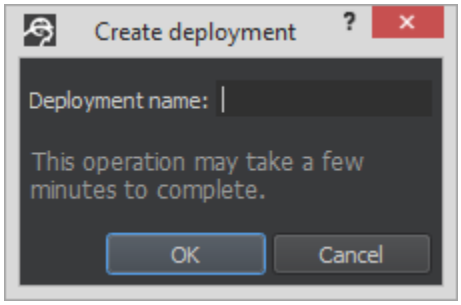
A deployment is an independent copy of the AWS resources that your game uses. Deployments are useful for maintaining a safe separation among game lifecycle phases such as development, test, and production. In the resource manager navigation pane, the **Deployments** node shows you the status of your project's deployments. You can also use it to create a new deployment.



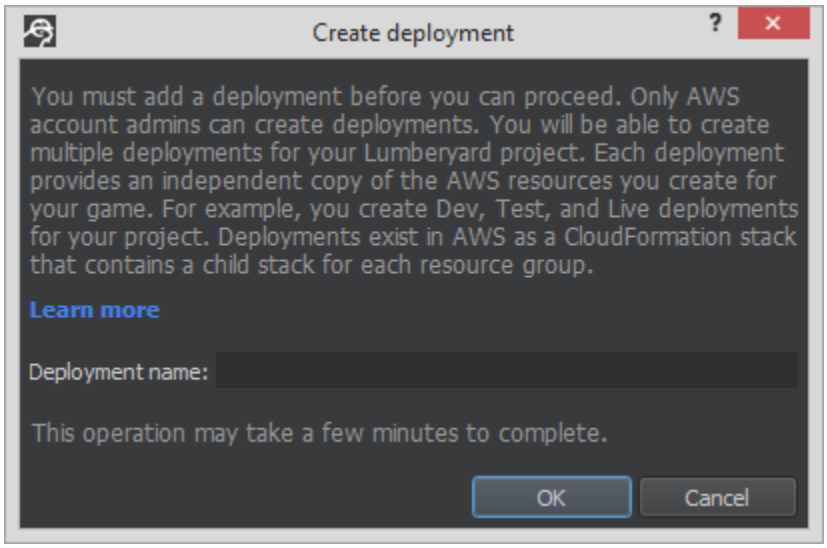
Note: If the **Deployments** node is selected when no AWS profile is configured, Lumberyard prompts you to provide a profile. The status of the project's deployments cannot be displayed unless a profile is provided. For more information, see Managing Cloud Canvas Profiles (p. 198).

Create Deployment

Click **Create deployment** to start the creation of a deployment:



When uploading resources for the first time, you may see this version of the dialog:



Provide a name for **Deployment name**. Lumberyard appends this name to the project stack name to create an AWS CloudFormation stack for the deployment.

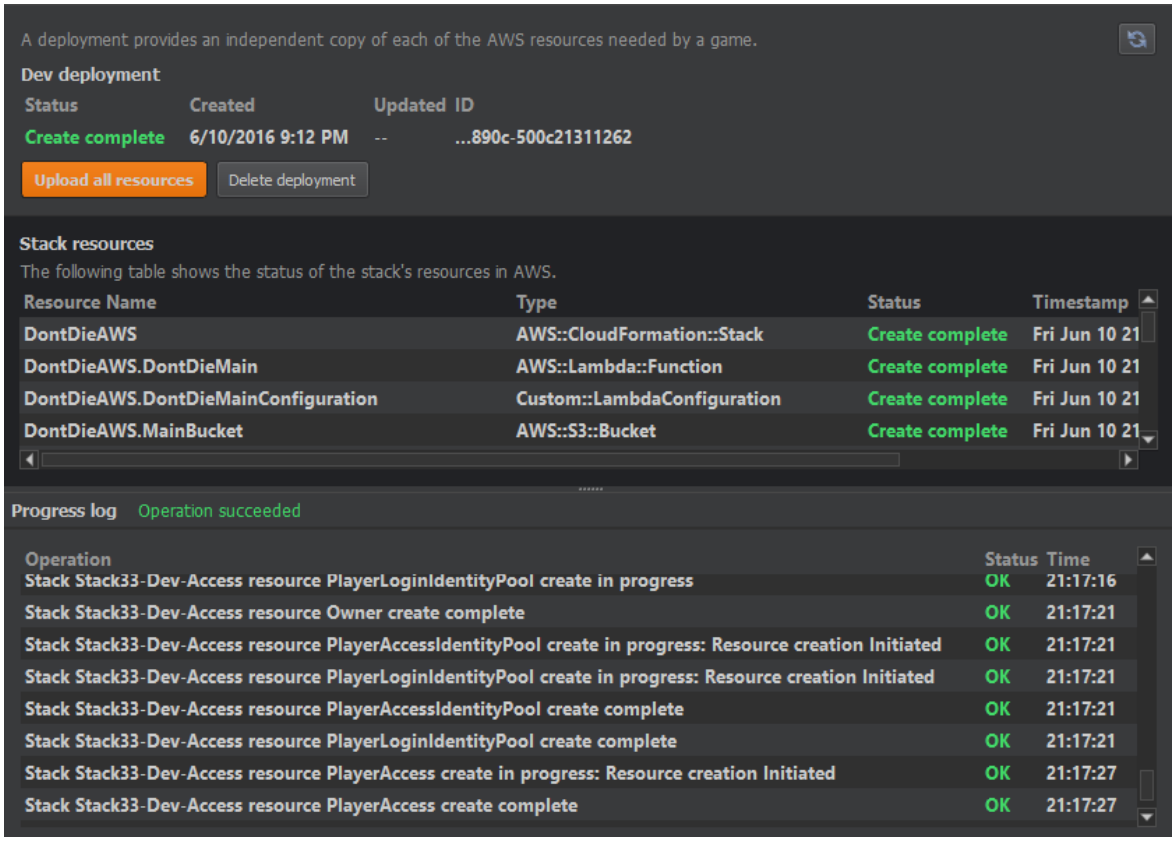
To start the deployment creation process, click **OK**. In the resource manager navigation pane, a node for the deployment appears under **Deployments**. In the detail pane, the Viewing the Cloud Canvas Progress Log (p. 279) provides details about the creation process.

Deployment Status Table

The **Deployment status** table shows the status of the AWS CloudFormation stack for each deployment. **Deployment** shows the deployment name. For descriptions of the remaining fields in this table, see Stack Resources Table (p. 384) in the Working with Project Stacks (p. 383) section.

Individual Deployment Nodes

The child nodes of the **Deployment** node each represent one of the Lumberyard project's deployments. When a **Deployment** node is selected, the detail pane shows the current status of the selected deployment.



Note
If a **Deployment** node is selected when no AWS profile is configured, Lumberyard prompts you to provide a profile. The status of the project's deployments cannot be displayed unless a profile is provided. For more information, see Managing Cloud Canvas Profiles (p. 198).

Individual Deployment Status Table

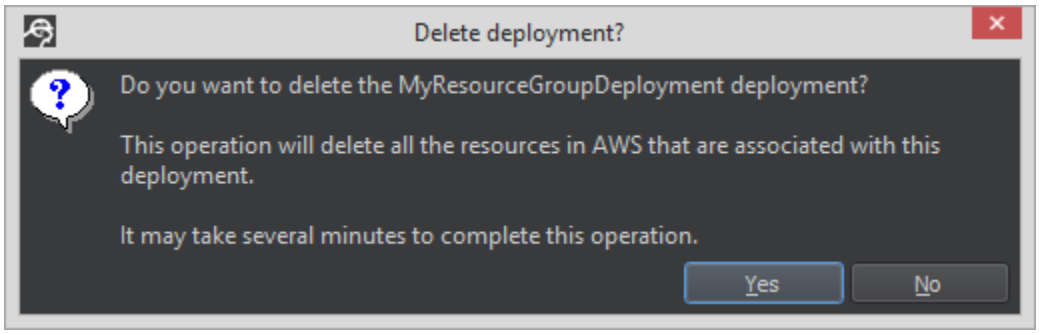
The **<Deployment Name> deployment status** table shows the status of the AWS CloudFormation stack for the selected deployment. For descriptions of the contents of this table, see Project Stack Status Table (p. 383) in the Working with Project Stacks (p. 383) section.

Upload All Resources

Click **Upload all resources** to start the process of modifying, creating, or deleting resources in the current AWS deployment so that they match your local definitions for all resource groups.

Delete Deployment

Click **Delete deployment** to start the process of deleting the deployment's resources from AWS. The resources defined by all resource groups will be deleted.



For more information about deleting deployments, see Deleting Cloud Canvas Deployments and Their Resources (p. 392).

Stack Resources Table

The **Stack resources** table shows the status of each of the resources defined by all the resource groups for the selected deployment. For descriptions of the fields in this table, see Stack Resources Table (p. 384) in the Working with Project Stacks (p. 383) section.

Topics

- Making a Cloud Canvas Deployment Active (p. 388)
- Testing Different Mappings (p. 390)
- Using Protected Deployments (p. 391)
- Deleting Cloud Canvas Deployments and Their Resources (p. 392)

Making a Cloud Canvas Deployment Active

You can select the deployment that you want Lumberyard Editor to consider active. The active deployment is the deployment that you work with in Lumberyard Editor. Lumberyard Editor uses the active deployment's resources when you launch your game. When you select the Working with Resource Groups (p. 280) node or an Managing Individual Resource Groups (p. 283) node in the **Cloud Canvas Resource Manager** navigation pane, the status information that appears corresponds to the active deployment.

You can also select the deployment that you want to be active by default for all team members.

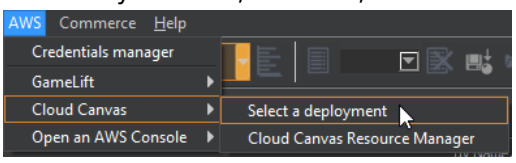
> **Note**
> To select a deployment, you must have initialized **Cloud Canvas Resource Manager** to work with your AWS account and created a deployment. For more information, see Initializing **Cloud Canvas Resource Manager** (p. 382) and Create Deployment (p. 385).

Making a Deployment Active

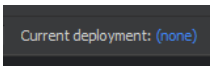
You have several ways to make a deployment active in **Cloud Canvas Resource Manager**.

To make a deployment active

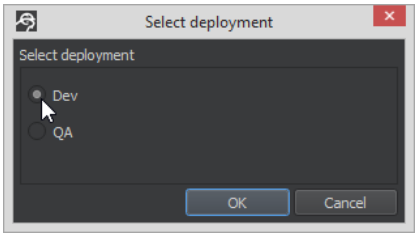
- To make a deployment active, do one of the following:
  - In Lumberyard Editor, click **AWS**, **Cloud Canvas**, **Select a deployment**.



- In the **Cloud Canvas Resource Manager** toolbar, click the name of the current deployment, or click **(none)** if none is configured:

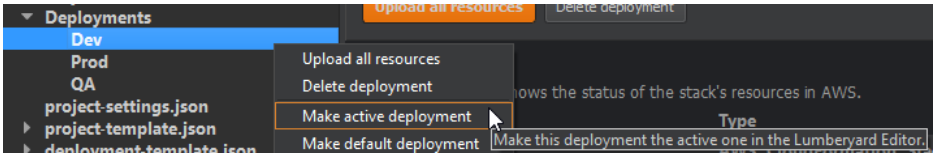
  

  When prompted, choose the deployment that you want to make active:

  One or more of the deployments may be marked **protected**. For more information, see Using Protected Deployments (p. 391).

- In the **Cloud Canvas Resource Manager** navigation pane, right-click the deployment that you want to make active, and then click **Make active deployment**:

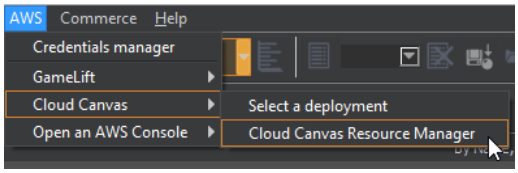
  

Making a Deployment the Default

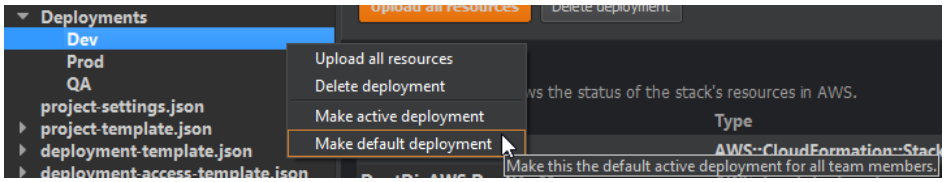
You can use the **Cloud Canvas Resource Manager** to make a deployment the default.

To make a deployment active by default for all team members

1. In Lumberyard Editor, click **AWS**, **Cloud Canvas**, **Cloud Canvas Resource Manager**.

2. In the **Cloud Canvas configuration** navigation tree, expand **Administration (advanced)**, and then expand **Deployments**.
3. Right-click the deployment that you want to make the default, and then click **Make default deployment**:

To use the command line to make a deployment the default

- To use the command line to make a deployment the default, type the following command:

```
lmbr_aws deployment default --set <deployment name>
```

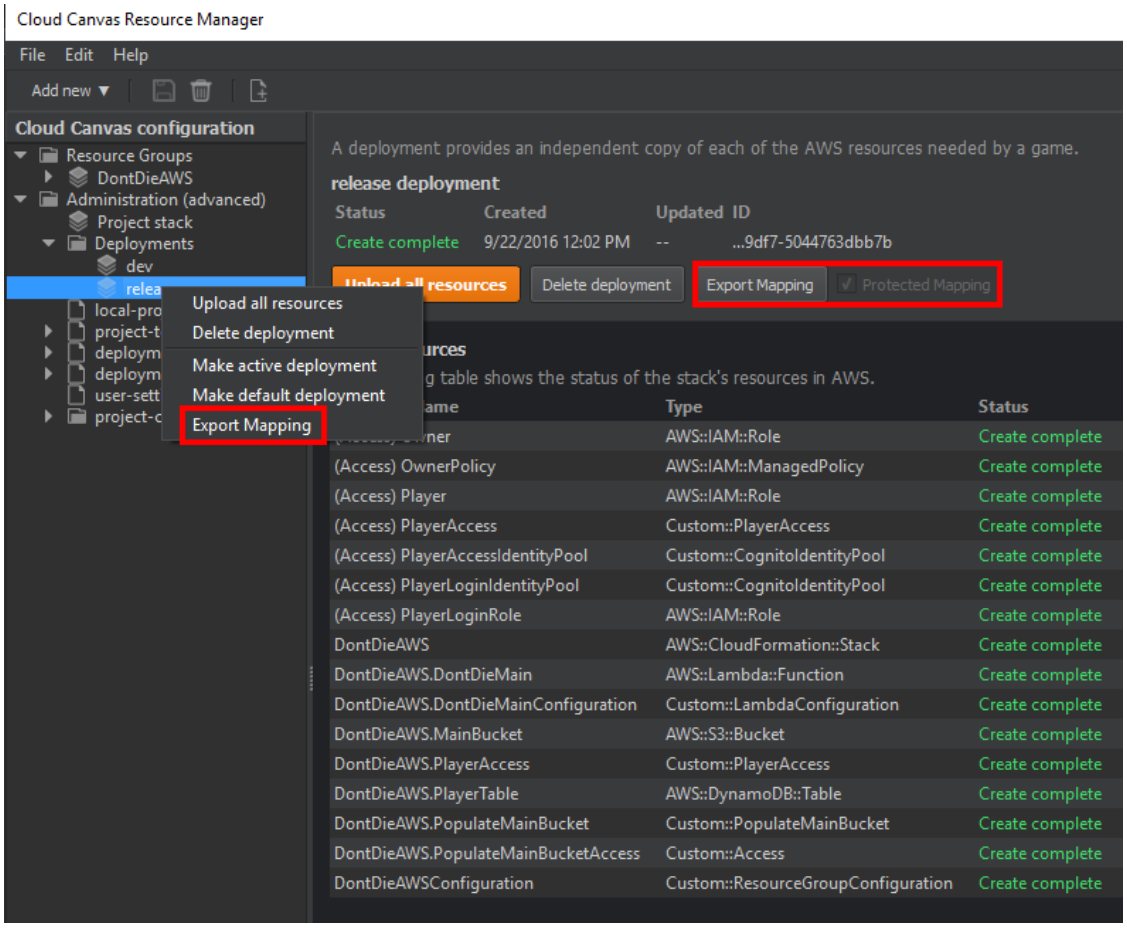
Testing Different Mappings

To test your client with different Cloud Canvas resource deployments, you can export mappings by using Cloud Canvas Resource Manager or the `lmbr_aws` command line.

To export a mapping from Cloud Canvas Resource Manager

- In Resource Manager, do one of the following:
  - Left click a deployment and click **Export Mapping** in the main window.
  - Right click the name of a deployment name the list, and then select **Export Mapping** from the context menu.

  The following image shows both options.



To exporting a mapping from the command line

- Type the following command, where *<name>* is the name of your deployment.

```
lmbr_aws mappings update --deployment <name>
```

The mapping file for the specified deployment is created in the *<project_directory>*\Config directory and has the format *<deployment_name>*.awsLogicalMappings.json.

Tip
Using the command line to export mappings makes it easy for you to create scripts for testing or development.

Selecting a Deployment with a PC Launcher

After you have exported one or more mappings, you can choose the mapping to use when you run a game launcher such as the one at dev\Bin64\SamplesProjectLauncher.exe.

To direct the launcher to use a specific deployment, use the command line option cc_override_resource_map, as in the following example.

```
SamplesProjectLauncher.exe -cc_override_resource_map Config\dev.awsLogicalMappings.json
```

The argument for the cc_override_resource_map parameter specifies the mapping file that you want to use.

If you have exported a single mapping file to the launcher, the launcher uses that mapping file by default. If you have exported multiple mapping files to the launcher, you must select a mapping by using the cc_override_resource_map parameter. If you don't specify a mapping after multiple mappings have been exported, the launcher gives an error message, and no mapping is loaded.

Using Protected Deployments

You can use Cloud Canvas to mark specific deployments as protected. Protected status makes it more difficult for users (typically, testers or developers) to inadvertently connect a development game client to live resources.

When a user starts a protected game, a message box notifies the user that he or she is attempting to use a protected deployment. The user is given the option to not connect before any potentially harmful data is transmitted.

The protection feature purposely uses a message box that "breaks" automation. If the scripts that run tests are configured to use a protected deployment, the Lumberyard client will not continue without human intervention.

When Protected Deployments Are Detected

When a game is run from Lumberyard Editor, protection is always detected. When a game is run from a Windows launcher, protection is detected only when the launcher is running in debug mode.

Marking a Deployment as Protected

Currently, you must set the protection from the lmbr_aws command line tool by using the protect-deployment command.

The protect-deployment command uses the following parameters.

--set *<deployment_name>* – Specifies that the deployment is protected.

`--clear` *<deployment_name>* – Specifies the deployment is not protected.

`--show` - Displays a list of currently protected deployments.

To display the protected status of deployments, you can also use either the `list-deployments` or `list-mappings` command.

Viewing Protected Status in Cloud Canvas Resource Manager

In Cloud Canvas Resource Manager, you can view, but not change, the status of protected deployments. The ability to change the protected status of deployments from Lumberyard Editor is planned for a future release.

> **Note**
> Setting a deployment to protected does not prevent you from deploying or deleting resources by using Cloud Canvas Resource Manager or the `lmbr_aws` command line tool; it only enables the warning functionality. For this reason, be careful not to make unnecessary changes to critical deployments. A more comprehensive model for protecting deployments is planned for a future version of Lumberyard.

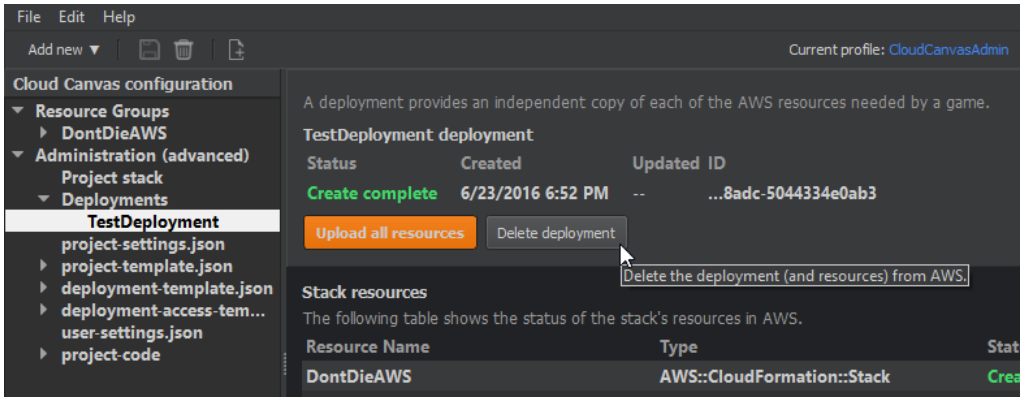
Deleting Cloud Canvas Deployments and Their Resources

To remove Cloud Canvas functionality from your Lumberyard project and the AWS resources related to it, you can use **Cloud Canvas Resource Manager** or the Cloud Canvas command line.

> **Warning**
> Only administrators should perform these actions. If you remove all AWS resources managed by Cloud Canvas for your Lumberyard project, the players of your game will not be able to access any of the Cloud Canvas resource groups that implement your game's cloud connected features.

To use Cloud Canvas Resource Manager to delete Cloud Canvas deployments and their resources

1. If you have checked Lumberyard into source control, ensure that the *<root>*`\`*<game>*`\AWS\project-settings.json` file has been checked out and is writeable.
2. In Lumberyard Editor, choose **AWS**, **Cloud Canvas**, **Cloud Canvas Resource Manager**.
3. In the **Cloud Canvas configuration** navigation pane, expand **Administration (advanced)**, and then expand **Deployments**. The list of the deployments in the project appears.
4. Select the deployment to delete and click **Delete deployment**.



5. When prompted to confirm, click **Yes** to start the process of deleting the deployment's resources from AWS. The process might take a few minutes.
6. To remove all of the project's resources from AWS, follow the same steps to delete each of the project's deployments.

To use the command line to delete Cloud Canvas deployments and their resources

1. If you have checked Lumberyard into source control, ensure that the *<root>**<game>*\AWS\project-settings.json file has been checked out and is writeable.
2. Open a command line prompt and change to your the Lumberyard \dev directory.
3. Determine the project's deployment names by typing the following command:

```
lmbr_aws deployment list
```

4. Type the following command for each of the deployments that you want to delete:

```
lmbr_aws deployment delete --deployment <deployment name>
```

   Note
   To remove all Cloud Canvas functionality from your project, use the `delete-deployment` command to delete all of deployments that were listed by `list-deployments`. Then remove the project stack as described in the step that follows.

5. After you have deleted all deployments, you can delete the resources that Cloud Canvas uses to manage your project by typing the following command:

```
lmbr_aws project delete
```

This removes all AWS resources that are related to your Cloud Canvas project.

Understanding the Resource Manager Security System

Cloud Canvas Resource Manager provides a robust set of features for securing your Amazon Lumberyard project and its cloud-connected resources. You can control the access to project resources of team members, players, Lambda functions, and specify other custom access control configuration for your project.

Topics

- Controlling Access to Resources (p. 393)
- Cloud Canvas Built-In Roles and Policies (p. 396)
- Player Identity (p. 400)
- Using the Cloud Canvas Command Line to Manage Roles and Permissions (p. 402)

Controlling Access to Resources

Setting access permissions correctly is key to ensuring that Cloud Canvas Resource Manager manages your project's cloud-connected features securely.

Access Scenarios and ProjectResourceHandler

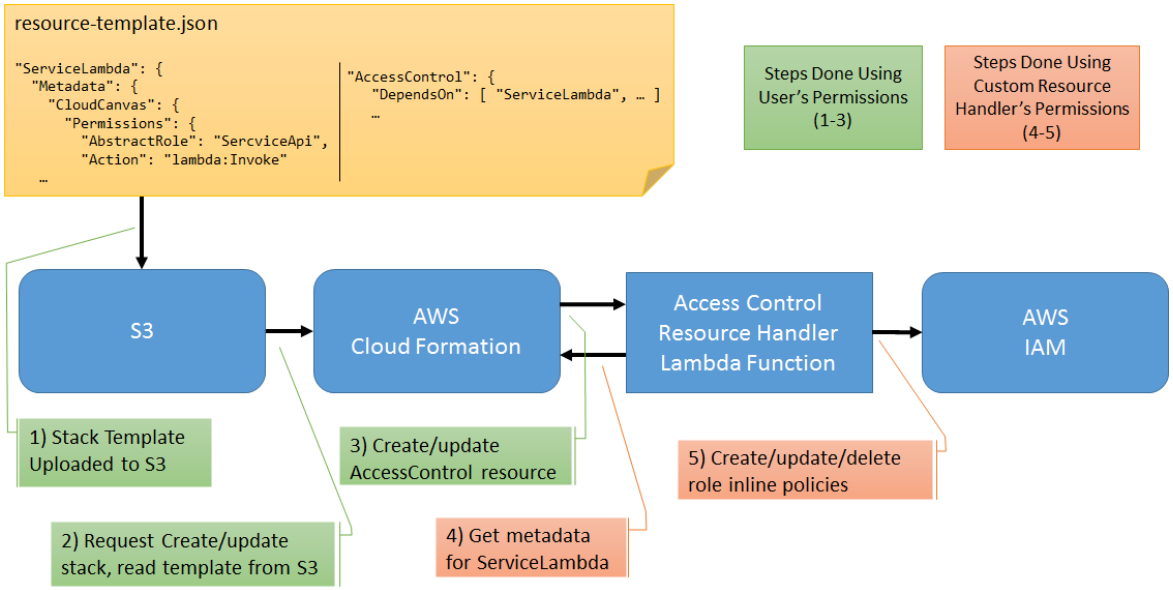
Cloud Canvas Resource Manager requires that the following access scenarios be supported. Additional roles with more nuanced permissions can be created, but the table below describes the core access requirements.

	Project Administrator	Project Team Member
Project Stack	Full Access	Read
Deployment Stack	Full Access	Read/Update
Deployment Access Stack	Full Access	Read
Resource Group Stack	Full Access	Create/Update/Delete
Roles and Policies	Full Access	Assume

A project team member must be able to create resource group stacks that contain arbitrary resources, but not be able to create or modify roles and policies. This introduces a significant complexity. Some resources like Lambda functions require that the developer also provide a role that is assumed by the resource. The developer must be able to create such roles and manage their policies. However, granting IAM permissions such as these directly to team members would effectively make them administrators.

To enable the required functionality while still limiting what a project team member can do directly, Cloud Canvas Resource Manager uses AWS CloudFormation custom resources. The custom resource handlers for Cloud Canvas Resource Manager are implemented in the `ProjectResourceHandler` Lambda function in the project stack. The Lambda function's execution role (`ProjectResourceHandlerExecution`) grants permissions that Cloud Canvas Resource Manager requires. These permissions are not granted to project team members.

For example, the `Custom::CloudGemFramework::AccessControl` resource, described in detail later in this document, is responsible for managing inline policies on various roles. It can perform these actions on the project team member's behalf. However, the `Custom::CloudGemFramework::AccessControl` handler also must know what to put into these policies. It can't trust the project team member to provide this information directly. Instead, it must construct the information from trusted sources. To do this, `AccessControl` uses metadata on resource definitions from AWS CloudFormation. It also constructs ARNs for the stack's resources identified by AWS CloudFormation. In this way, only a user with permission to update the stack can influence the policies that are constructed for the resources in that stack.



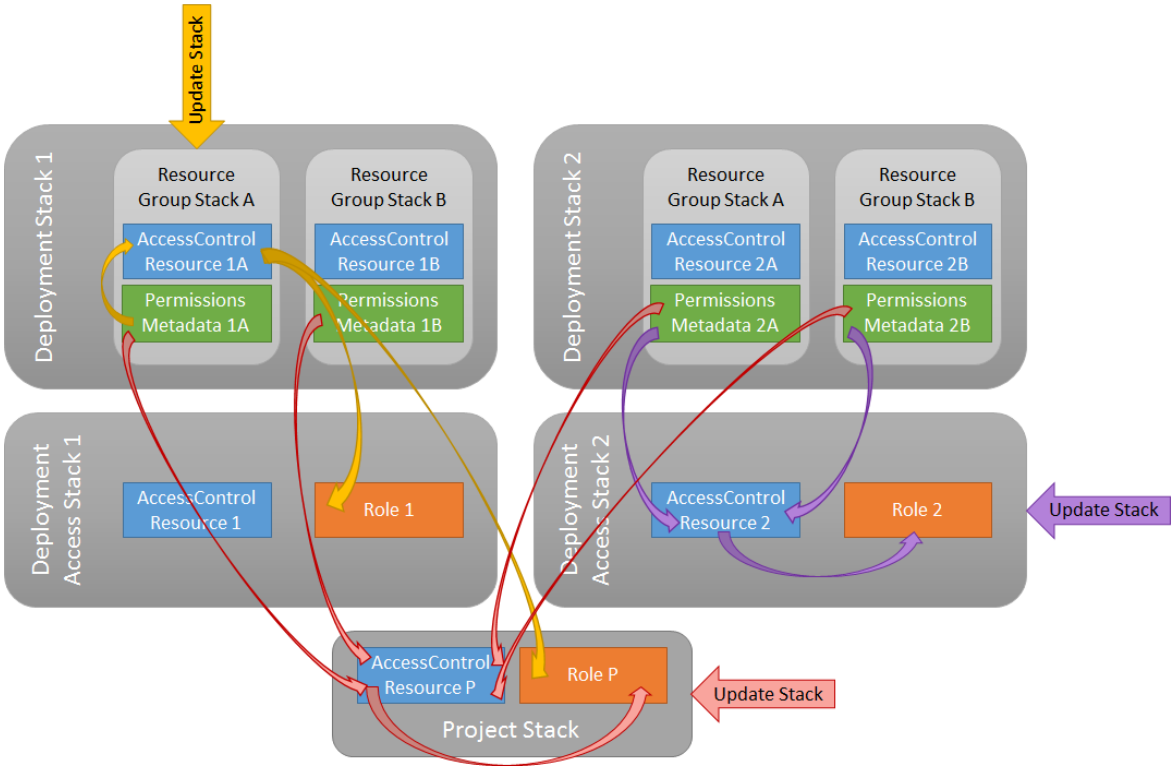
Using the Custom::CloudGemFramework::AccessControl Resource

As described above, Cloud Canvas Resource Manager security depends on IAM Roles and the credentials used from assuming such roles. The Access Scenarios and ProjectResourceHandler (p. 393) section earlier in this topic explains why Cloud Canvas Resource Manager has the responsibility of managing the inline policies attached to these roles.

This section describes the data used by the `Custom::CloudGemFramework::AccessControl` resource handler to configure the project's roles. A `Custom::CloudGemFramework::AccessControl` resource must be defined in the following templates:

Template	Description
`project-template.json`	Causes policies on the roles defined in the `project-template.json` file to be updated. These roles can provide access to any resource defined in any resource groups across all deployments.
`deployment-access-template.json`	Causes policies on the roles defined in the `deployment-access-template.json` file to be updated. These roles can provide access to any resource in any resource of a given deployment.
`resource-group-template.json`	Causes policies on the roles defined in the `project-template.json` and `deployment-access-template.json` files to be updated. Only permissions for the resource defined in the `resource-group-template.json` file are updated. For roles defined in the `deployment-access-template.json` file, only the instances of those roles for the deployment that contains the resource group stack are updated.

This process is illustrated in the following diagram. The diagram shows the metadata that is read and roles that are updated when a resource group stack, deployment access stack, or project stack is updated.



Custom::CloudGemFramework::AccessControl Resource Definitions

The `Custom::CloudGemFramework::AccessControl` resource supports the following properties:

Property	Description
`ConfigurationBucket`	The name of the project's configuration bucket. This property must be provided.
`ConfigurationKey`	Identifies the location in the configuration bucket where data for the stack operation is stored. However, the custom resource handler depends on this value changing on each update. Property changes such as this cause AWS CloudFormation to invoke the custom resource handler on each stack operation.
`ServiceToken`	Identifies the Lambda function that is invoked for the custom resource. This should be the project global `ProjectResourceHandler` Lambda function that is defined in the `project-template.json` file.

The `DependsOn` attribute of the `Custom::CloudGemFramework::AccessControl` resource definition must list the following resources.

- All the resources in the `project-template.json`, `deployment-access-template.json`, or `resource-group-template.json` files that provide permissions metadata.
- All the `AWS::IAM::Role` resources that have `RoleMapping` metadata.
- Any custom resources that create implicit roles, such as `Custom::CloudGemFramework::LambdaConfiguration` and `Custom::CloudGemFramework::ServiceApi` resources.

When you use the AWS CLI to manage roles and permissions, these resources are listed for you. However, if you edit these files yourself, it is important that you maintain these dependencies. Without these dependencies, the `Custom::CloudGemFramework::AccessControl` resource might be updated before the other resources have been updated. If this occurs, the `Custom::CloudGemFramework::AccessControl` no longer has access to the latest metadata from the resources, and the changes that were intended might not be made.

Setting Access Permissions

For information on setting `Custom::CloudGemFramework::AccessControl` permissions, see Permissions Metadata for Resource Definitions (p. 290).

Cloud Canvas Built-In Roles and Policies

You can use the built-in Cloud Canvas roles and policies to manage resource and deployment permissions for your project.

Roles

You can use the AWS::IAM:Role resource to define roles in your `project-template.json` or `deployment-access-template.json` files. Cloud Canvas Resource Manager defines the following roles for you:

Role	File	Description
`ProjectResourceHandlerExecution`	`project-template.json`	Grants the `ProjectResourceHandler` Lambda function the permissions it has at run time.
`ProjectServiceLambdaExecution`	`project-template.json`	Grants the `ProjectServiceLambda` Lambda function the permissions that it has at run time.
`ProjectOwner`	`project-template.json`	Grants project administrators full access to all the project's resources. To modify the default permissions granted by this role, edit the `ProjectOwnerAccess` policy definition. **Note** Permissions added to `ProjectOwnerAccess` are also granted to `ProjectAdmin` unless they are denied by `ProjectAdminRestrictions`.
`ProjectAdmin`	`project-template.json`	Similar to the `ProjectOwner` role, but cannot create, update, or delete deployments with names that start with "Release." This is a convenience measure that makes using `ProjectAdmin` safer than using `ProjectOwner`. To add additional restrictions to the `ProjectAdmin` role, edit the `ProjectAdminRestrictions` definition.
`DeploymentOwner`	`deployment-access-template.json`	Grants developers full access to all the resources in a deployment. To modify the default permissions granted by this role, edit the `DeploymentOwnerAccess` policy definition. **Note** Permissions added to `DeploymentOwnerAccess` are also granted to `DeploymentAdmin` unless they are denied by `DeploymentAdminRestrictions`.
`DeploymentAdmin`	`deployment-access-template.json`	Similar to the `DeploymentOwner` role, but cannot create or delete resource group stacks. This is a convenience measure that makes using `DeploymentAdmin` safer than using `DeploymentOwner`. To add additional restrictions to the `DeploymentAdmin` role, edit the `DeploymentAdminRestrictions` policy definition.
`Player`	`deployment-access-template.json`	Grants players limited access to specific resources in a deployment.

The configuration file in which you define a role determines the resources to which the role provides access.

File	Scope
`project-template.json`	Applies to all resources in all resource groups for `all` deployments. Only a single instance of the role is created for the entire project.
`deployment-template.json`	Applies to all resources in all resource groups for a `specific` deployment. A separate instance of the role is created for each deployment.

You can use the `lmbr_aws` command line tool to manage the role definitions in the `project-template.json` and `deployment-access-template.json` files. For more information, see Using the Cloud Canvas Command Line to Manage Roles and Permissions (p. 402).

Implicit Roles

Some Cloud Canvas custom resources also create roles. For example, when a Lambda function is executed, it assumes the role that the `Custom::CloudGemFramework::LambdaConfiguration` resource creates. When API Gateway invokes a Lambda function or accesses other resources, it assumes the role that the `Custom::CloudGemFramework::ServiceApi` resource creates. Including these custom resources in a `resource-group-template.json` file causes these implicit roles to be created (and deleted when the resource is deleted). For information on implicit role names, see Implicit Role Mappings (p. 400).

Managed Policies

You can use `AWS::IAM::ManagedPolicy` resources to define permissions that are shared across any number of roles. Cloud Canvas defines the following managed policies for you:

Policy	File	Description
`ProjectAccess`	`project-template.json`	Defines the permissions needed to access the project and deployment configuration that must be read before a project-wide role can be assumed. For more information, see Assuming a Role (p. 402).
`ProjectOwnerAccess`	`project-template.json`	Defines the default permissions granted to the `ProjectOwner` and `ProjectAdmin` roles.
`ProjectAdminRestrictions`	`project-template.json`	Defines restrictions to the `ProjectOwnerAcess` policy that apply only to the `ProjectAdmin` role.
`DeploymentAccess`	`deployment-access-template.json`	Defines the permissions needed to access project and deployment configuration that must be read before a deployment-specific role can be assumed. For more information, see Assuming a Role (p. 402).
`DeploymentOwnerAccess`	`deployment-access-template.json`	Defines the default permissions granted to the `DeploymentOwner` and `DeploymentAdmin` roles.
`DeploymentAdminRestrictions`	`deployment-access-template.json`	Defines restrictions to the `DeploymentOwnerAccess` policy that apply only to the `DeploymentAdmin` role.

The `ProjectAdmin` and `DeploymentAdmin` roles are granted the same permissions as the `ProjectOwner` and `DeploymentOwner` roles, minus any permissions specifically denied by the `ProjectAdminRestrictions` and `DeploymentAdminRestrictions` managed policies, respectively. In effect an, "admin" is granted all the permissions of an "owner" minus any special actions that the "admin" should not be able to perform.

Role Mapping Metadata

The `AbstractRole` property in the `Permission` metadata object does not directly specify the actual role that receives the described permission. These values must be mapped to actual IAM roles. This makes it possible to setup roles in whatever way makes sense for your project. It also removes the need to modify the permissions defined by individual resource groups.

The ability to map abstract roles to actual IAM roles is important when you use a cloud gem across multiple projects or from a third party. Cloud gems acquired from a third party might have roles that are

different from the roles that you use in your organization. (A cloud gem is a Lumberyard gem that uses the AWS resources defined by a Cloud Canvas Resource Group. For more information, see Cloud Gems.)

The `Custom::CloudGemFramework::AccessControl` resource looks for CloudCanvas `RoleMappings` metadata on `AWS::IAM::Role` resources to determine which abstract roles map to that physical role. In the following example, the `CustomerSupport` abstract role from all resource groups is mapped to the `DevOps` physical role.

```
...

"DevOps": {
    "Type": "AWS::IAM::Role",
    "Properties": {
        "Path": { "Fn::Join": [ "", [ "/", { "Ref": "ProjectStack" }, "/", { "Ref":
 "DeploymentName" }, "/" ]] }
    },
    "Metadata": {
        "CloudCanvas": {
            "RoleMappings": [
                {
                    "AbstractRole": [ "*.CustomerSupport" ],
                    "Effect": "Allow"
                }
            ]
        }
    }
},

...
```

Each Cloud Canvas `RoleMapping` metadata object can have the following properties.

Property	Description
`AbstractRole`	Required string or list of strings of the form *<resource-group-name>*.*<abstract-role-name>*. To match abstract roles defined in any resource group, use * for the *<resource-group-name>*. To match abstract roles defined only in a specific resource group, use an actual resource group name.
`Effect`	Required string. Must be either `Allow`, to allow an action defined by permission metadata, or `Deny` to deny an action. This is used as the Effect property in a statement in the policy.

You can use the `lmbr_aws` command line tool to manage `RoleMappings` metadata on role resource definitions in the `project-template.json` and `deployment-access-template.json` files. For more information, see Using the Cloud Canvas Command Line to Manage Roles and Permissions.

Default Role Mappings

Cloud Canvas defines role mappings for the following roles:

Role	File	Default Role Mapping
`ProjectResourceHandlerExecution`	`project-template.json`	none
`ProjectServiceLambdaExecution`	`project-template.json`	none
`ProjectOwner`	`project-template.json`	`*.ProjectAdmin` `*.ProjectOwner`

Role	File	Default Role Mapping
`ProjectAdmin`	`project-template.json`	`*.ProjectAdmin`
`DeploymentOwner`	`deployment-access-template.json`	`*.DeploymentAdmin` `*.DeploymentOwner`
`DeploymentAdmin`	`deployment-access-template.json`	`*.DeploymentAdmin`
`Player`	`deployment-access-template.json`	`*.Player`

Implicit Role Mappings

As mentioned in Implicit Roles (p. 398), role mappings are automatically defined for the implicit roles created by Cloud Canvas resources like `Custom::CloudGemFramework::LambdaConfiguration`. These mappings are only used with permission metadata in the same `resource-group-template.json` file as the custom resource that creates the role. The name of the abstract role used in permission metadata to reference an implicit role depends on the custom resource type.

Resource	Implicit Role Name
`Custom::CloudGemFramework::LambdaConfiguration`	The name of the Lambda function, as specified by the `Function` property in the `Custom::CloudGemFramework::LambdaConfiguration` resource definition. This is also the logical ID of the `AWS::Lambda::Function` resource.
`Custom::CloudGemFramework::ServiceApi`	The logical ID of the `Custom::CloudGemFramework::ServiceApi` resource.

Player Identity

A Lumberyard cloud-connected game must use AWS credentials that grant the desired access when calling AWS APIs (using either the C++ AWS SDK or the AWS flow nodes). Cloud Canvas uses an Amazon Cognito identity pool to get these credentials.

Using a Amazon Cognito identity pool has the benefit of providing the game with a unique identity for each individual player. This identity can be used to associate the player with their saved games, high scores, or any other data stored in DynamoDB tables, Amazon S3 buckets, or other locations. If you use the Player Account Cloud Gem, this identity is mapped to an account ID that can be used instead.

Amazon Cognito identity pools support both unauthenticated and authenticated identities. Unauthenticated identities are associated with a single device such as a PC, tablet, or phone, and have no associated user name or password.

Authenticated identities are associated with the identity of an user as determined by an external identity provider such as Amazon, Facebook, or Google, or an Amazon Cognito user pool. This allows Amazon Cognito to provide the game with the same player identity everywhere a user plays a game. The user's saved games, high scores, and other data effectively follow the user from device to device.

Cloud Canvas supports both anonymous (unauthenticated) and authenticated player identities. The Player Account Cloud Gem is the easiest way to add authentication. Authenticated identity support for other providers is more complex and requires additional setup and coding.

Anonymous (Unauthenticated) Player Login

The Cloud Canvas client configuration system is automatically initialized by calling `gEnv->lmbrAWS->GetClientManager()->ApplyConfiguration()`, or by using a Cloud Canvas `(AWS):Configuration:ApplyConfiguration` flow node. An existing anonymous identity is loaded from the local identities cache. If an anonymous identity is not found in the cache, a new identity is acquired from the pool. The identity cache is stored in an `.aws/.identities` file in your home directory.

Authenticated Player Login

In order to understand how to use Cloud Canvas to implement authenticated player identities for your game, you must be familiar with Amazon Cognito's Enhanced (Simplified) Authflow. For information, see the article Authentication Flow in the Amazon Cognito Developer Guide.

The login process for authenticated player identities is more complex than the anonymous player login process. The Player Account Cloud Gem handles this for Amazon Cognito user pools. For other providers, this login process requires additional setup beyond what Cloud Canvas provides by default.

The authenticated player login process takes place automatically when the Cloud Canvas client configuration system is initialized by calling `gEnv->lmbrAWS->GetClientManager()->ApplyConfiguration()`, or by using a Cloud Canvas `(AWS):Configuration:ApplyConfiguration` flow node. If an authenticated identity is found in the local identities cache, it is loaded and the access token is refreshed if needed using the stored refresh token. If no existing identity is found, it falls back to using the anonymous identity for AWS calls.

All the code that implements the authenticated login flow can be found in the *{root}*`\Code\CryEngine\LmbrAWS\Configuration` directory. A description of the files follows.

- **`ClientManagerImpl.*`** – Configures the game's AWS clients to use the `TokenRetrievingPersistentIdentityProvider` identity provider.

- **`ResourceManagementLambdaBasedTokenRetrievalStrategy.*`** – Implements the token exchange process that calls the `ProjectPlayerAccessTokenExchangeHandler` Lambda function.

- **`TokenRetrievingPersistentIdentityProvider.*`** – An implementation of the `PersistentCognitoIdentityProvider` interface defined in the AWS SDK that uses `ResourceManagementLambdaBasedTokenRetrievalStrategy` instances to implement the token exchange process.

Configuring a Cognito Identity Provider (Cognito User Pool)

Cloud Canvas provides the `Custom::CloudGemFramework::CognitoUserPool` resource for adding the Amazon Cognito user pools and linking them to a Amazon Cognito identity pool. The Player Account Cloud Gem (p. 241) uses this custom resource. It also provides an EBus interface so that you can work with the user pool and the sample level that has an in-game menu.

Configuring External Identity Providers

Cloud Canvas does not automate the process of retrieving an auth code from an external identity provider. This is your responsibility as a game developer. After retrieving the auth code, call `gEnv->lmbrAWS->->GetClientManager()->Login(providerName, code)`.

External identity providers are configured using the `lmbr_aws` login-provider add (p. 413), login-provider update (p. 415), and login-provider remove (p. 414) commands. These commands save the

configuration in a `/player-access/auth-settings.json` object in the project's configuration bucket so that the `ProjectPlayerAccessTokenExchangeHandler` Lambda function can access it.

> **Note**
> You must run `lmbr_aws project update` after running `login-provider add`, `login-provider update`, or `login-provider remove` so that the PlayerAccessIdentityPool (p. 289) configuration will be updated to reflect the change.

Automatic Token Refresh

When using Amazon Cognito with external identity providers, it is necessary to periodically refresh the token from that provider and then get updated credentials for that token from Amazon Cognito. Cloud Canvas performs this token refresh process automatically by using the `ProjectPlayerAccessTokenExchangeHandler` Lambda function.

Using the Cloud Canvas Command Line to Manage Roles and Permissions

You can use the `lmbr_aws` command line tool to manage Cloud Canvas Resource Manager access control. For example, you can use the tool to assume a role when you run a command, or to manage roles, permissions, and role mappings.

Assuming a Role

Most `lmbr_aws` commands support an *--assume-role* `<role-name>` argument. You can use this argument to assume a role when you run a command.

If specified, `<role-name>` must be the logical resource ID of an IAM role resource defined in either the `project-template.json` or `deployment-access-template.json` files.

> **Note**
> You should avoid defining roles that have the same name in both files. If you do, the role from the deployment file takes precedence.

If you specify a deployment access role, the actual role used depends on the deployment on which the command is operating. If the `--deployment` argument has been specified, then the specified deployment is used. If the `--deployment` argument has not been specified and the user has specified a default deployment, the default deployment is used. If a default deployment has not been specified, the project's default deployment is used.

`lmbr_aws` uses uses your configured AWS credentials to assume the specified role. See Using the Cloud Canvas Command Line - Configuration for a description of how the credentials are determined. The credentials must have permission to assume the role. For more information, see Granting a User Permission to Switch Roles.

Before assuming the role, the `lmbr_aws` tool uses credentials to read project configuration data from AWS. The `ProjectAccess` managed policy in the `project-template.json file` and the `DeploymentAccess` managed policy in the `deployment-access-template.json` file grant the permissions necessary to read this information. You can attach the corresponding managed policy to any IAM user that works on a project or deployment.

Note that administrative users created for an AWS account normally have permissions to assume roles and read project configuration. Administrative users typically have permission to perform any action on any resource owned by an account.

Role Management Commands

Role management commands manage the AWS::IAM::Role (p. 396) resource definitions in the `project-template.json` and `deployment-access-template.json` files. After you use these commands to make

changes, you must update the project or deployment access stacks for the changes to take effect. For information about the permissions to perform this action, see Setting Access Permissions.

lmbr_aws role add

Adds an AWS IAM Role resource definition to the `project-template.json file` or `deployment-access-template.json` file.

Argument	Description
`##role <role#name>`	Required. The name of the role resource definition.
`##project`	Optional. When present, specifies that the role resource definition be added to the `project-template.json file`. Otherwise, the role resource definition is added to the `deployment-access-template.json` file.

lmbr_aws role remove

Removes an AWS IAM role resource definition from the `project-template.json file` or `deployment-access-template.json` file.

Argument	Description
`##role <role#name>`	Required. The name of the role resource definition.
`##project`	Optional. When present, specifies that the role resource definition be removed from the `project-template.json file`. Otherwise, the role resource definition is removed from the `deployment-access-template.json` file.

lmbr_aws role list

Lists the AWS IAM role definitions in the `project-template.json` and/or `deployment-access-template.json` files.

Argument	Description
`##deployment`	Optional. Either `--deployment` or `--project` can be specified. If `--deployment` is specified, only the roles in the `deployment-access-template.json` file are listed.
`##project`	Optional. Either `--deployment` or `--project` can be specified. If `--project` is specified, only the roles in the `project-template.json file` are listed.

Output

The output is similar to the following example.

```
Scope      Name
---------- ------------------------------------------
Deployment DeploymentAdmin
Deployment DeploymentOwner
Deployment Player
Deployment PlayerLoginRole
Project    ProjectAdmin
Project    ProjectOwner
```

```
Project    ProjectPlayerAccessTokenExchangeHandlerRole
Project    ProjectResourceHandlerExecution
```

Column	Description
Scope	Indicates whether the role is defined in the `deployment-access-template.json` or `project-template.json` file.
Name	Shows the resource definition name. This is the "logical" resource name, not the "physical" resource name which identifies an actual instance of the role. To see the physical resource names, use the `lmbr_aws project list-resources` or `lmbr_aws deployment list-resources` command.

Permission Metadata Management

The permission metadata management commands manage CloudCanvas `Permissions` metadata (p. 290) on resource definitions in the `resource-group-template.json` files. After you use these commands to make changes, you must update the project or deployment access stacks for the changes to take effect. For information about the permissions to perform this action, see Setting Access Permissions.

lmbr_aws permission add

Adds Cloud Canvas `Permissions` metadata to an resource definition in a `resource-group-template.json` file.

Argument	Description
`##resource#group <resource#group#name>`	Required. The name of a resource group. The metadata will be added to a resource definition in that resource group's `resource-group-template.json` file.
`##resource <resource#name>`	Required. The name of the resource definition in the `resource-group-template.json` file.
`##role <abstract-role#name>`	Required. Identifies the role that is granted the permission.
`##action <action> [<action> ...]`	Required. The action that is allowed. You can specify more than one action.
`##suffix <suffix> [<suffix> ...]`	Optional. A string appended to the resource ARN. You can specify more than one suffix.

lmbr_aws permission remove

Removes Cloud Canvas `Permissions` metadata from a resource definition in a `resource-group-template.json` file.

Argument	Description
`##resource#group <resource#group#name>`	Required. The name of a resource group. The metadata is removed from a resource definition in the specified resource group's `resource-group-template.json` file.
`##resource <resource#name>`	Required. The name of the resource definition in the `resource-group-template.json` file.

Argument	Description
`##role <abstract#role#name>`	Required. Identifies the roles from which permissions are removed.
`##action <action> [<action>...]`	Optional. The action that is removed. You can specify more than one action. If not specified, all permissions for the role are removed.
`##suffix <suffix> [<suffix>...]`	Optional. A string appended to the resource ARN, which is removed. You can specify more than one suffix.

lmbr_aws permission list

Removes Cloud Canvas `Permissions` metadata from an resource definition in a `resource-group-template.json` file.

Argument	Description
`##resource#group <resource#group#name>`	Optional. Lists the metadata from resource definitions in the resource group's `resource-group-template.json` file. The default lists permissions from all resource groups.
`##resource <resource#name>`	Optional. The name of the resource definition in the `resource-group-template.json` file. The default lists metadata from all resource definitions.
`##role <abstract#role#name>`	Optional. Lists metadata for the specified abstract role. The default lists metadata for all abstract roles.

Output

The output is similar to the following example.

```
Resource Group Resource             Resource Type         Roles          Actions
                                   ARN Suffixes
-------------- -------------------- --------------------- --------------
 ------------------------------------------------- -------------
DynamicContent ContentBucket        AWS::S3::Bucket       ServiceLambda  s3:GetObject
                                   /*
DynamicContent ContentBucket        AWS::S3::Bucket       ContentRequest s3:*
                                   /*
DynamicContent ContentRequest       AWS::Lambda::Function Player
 lambda:InvokeFunction
DynamicContent ServiceLambda        AWS::Lambda::Function ServiceApi
 lambda:InvokeFunction
DynamicContent StagingSettingsTable AWS::DynamoDB::Table  ServiceLambda  dynamodb:GetItem,
 dynamodb:Scan, dynamodb:UpdateItem
DynamicContent StagingSettingsTable AWS::DynamoDB::Table  ContentRequest dynamodb:GetItem
```

Column	Description
Resource Group	Shows the resource group where the permission metadata was found.
Resource	Shows the name of the resource definition with the metadata.
Resource Type	Shows the type of the resource definition with the metadata.

Column	Description
Roles	Shows the abstract roles specified by the permission metadata.
Actions	Shows the actions specified by the permission metadata.
ARN Suffixes	Shows the suffix added to the resource ARN, as specfied by the permission metadata.

Tip

To see all the resources players have access to through the game client, use the command:

```
lmbr_aws permission list --role Player
```

Role Mapping Metadata Management

Role mapping metadata management commands manage CloudCanvas `RoleMappings` metadata on `AWS::IAM:Role` resource definitions in the `project-template.json` and `deployment-access-template.json` files. After you use these commands to make changes, you must update the project or deployment access stacks for the changes to take effect. For information about the permissions to perform this action, see Setting Access Permissions.

lmbr_aws role-mapping add

Adds Cloud Canvas `RoleMappings` metadata to an AWS IAM Role definition in the `project-template.json` or `deployment-access-template.json` file.

Argument	Description
`##role` *`<role#name>`*	Required. The name of the role resource definition.
`##pattern` *`<abstract#role#pattern>`*	Identifies the abstract roles mapped to the role. Has the form *`<resource-group-name>`*.*`<abstract-role-name>`*, where *`<resource-group-name>`* can be *.
`##allow`	Either `--allow` or `--deny` must be specified. Indicates that the permissions requested for the abstract role are allowed.
`##deny`	Either `--allow` or `--deny` must be specified. Indicates that the permissions requested for the abstract role are denied.
`##project`	Optional. Indicates that the role definition is in the `project-template.json` file. The default is for the role definition to be in the `deployment-access-template.json` file.

lmbr_aws role-mapping remove

Removes an AWS IAM Role resource definition from the `project-template.json file` or `deployment-access-template.json` file.

Argument	Description
`##role` *`<role#name>`*	Required. The name of the role resource definition.
`##pattern` *`<abstract#role#pattern>`*	Identifies the abstract roles mapped to the role. Has the form ***`<resource-group-name>`***.***`<abstract-role-name>`***, where *`<resource-group-name>`* can be *.

Argument	Description
##project	Optional. Indicates that the role definition is in the project-template.json file. The default is for the role definition to be in the deployment-access-template.json file.

lmbr_aws role-mapping list

Lists the AWS IAM Role definitions in the project-template.json and/or deployment-access-template.json files.

Argument	Description
##role *<role#name>*	Required. The role definition with the metadata to list. The default is to list metadata from all role definitions.
##pattern *<abstract#role#abstract>*	The abstract role pattern specified by the metadata listed. The default is to list metadata with any abstract role pattern.
##deployment	Optional. Either --deployment or --project can be specified. Lists metadata from role definitions in the deployment-access-template.json file. The default is to list metadata from role definitions in the project-template.json and deployment-access-template.json files.
##project	Optional. Either --deployment or --project can be specified. Lists metadata from role definitions in the project-template.json file. The default is to list metadata from role definitions in the project-template.json and deployment-access-template.json files.

Output

The output is similar to the following example.

```
Scope      Actual Role                    Abstract Role                 Effect
---------- ------------------------------ ----------------------------- ------
Deployment DeploymentAdmin                *.DeploymentAdmin             Allow
Deployment DeploymentOwner                *.DeploymentAdmin             Allow
Deployment DeploymentOwner                *.DeploymentOwner             Allow
Deployment Player                         *.Player                      Allow
Project    ProjectAdmin                   *.ProjectAdmin                Allow
Project    ProjectOwner                   *.ProjectAdmin                Allow
Project    ProjectOwner                   *.ProjectOwner                Allow
```

Column	Description
Scope	Shows whether the role mapping came from the project-template.json or deployment-access-template.json files.
Actual Role	Shows the name of the role resource definition with the mapping metadata.
Abstract Role	Shows the abstract roles (as specified on permission metadata) that map to the role.
Effect	Shows whether the permissions requested for the abstract role are allowed or denied.

Using the Cloud Canvas Command Line

Cloud Canvas provides the `\dev\lmbr_aws.cmd` command line tool for working with AWS resources. The tool invokes Python code that is located in the `\dev\Tools\lmbr_aws` directory.

Notes

- Lumberyard 1.9 has renamed some `lmbr_aws` commands. The older commands still work but might be removed at a future date. For a list of the older commands and their newer equivalents, see Command Reorganization (p. 424).
- For information on using Dynamic Content Cloud Gem `lmbr_aws` commands to update dynamic content, see Using lmbr_aws for Dynamic Content (p. 233).
- For information on `lmbr_aws` commands for the Cloud Gem Portal and the cloud gem framework, see Using the Cloud Gem Framework Command Line (p. 379).
- For information on `lmbr_aws` commands that manage roles and permissions, see Using the Cloud Canvas Command Line to Manage Roles and Permissions (p. 402).

Syntax

```
lmbr_aws {command} {command-arguments}
```

{command} is one of commands in the command summary section that follows. *{command-arguments}* are the arguments accepted by the command. Arguments common to most commands are listed in the Common Arguments (p. 409) section. Arguments unique to a command are listed in the detail section for the command.

Configuration

The tool gets its default AWS configuration from the same `~/.aws/credentials` and `~/.aws/config` files as the AWS command line tools (for information, see Configuring the AWS Command Line Interface). The `lmbr_aws` tool does not require that the AWS command line interface be installed.

Environment Variables

As with the AWS command line tools, the default AWS configuration can be overridden by using the following environment variables.

- `AWS_ACCESS_KEY_ID` The access key for your AWS account.
- `AWS_SECRET_ACCESS_KEY` The secret key for your AWS account.
- `AWS_DEFAULT_REGION` The default region to use; for example, `us-east-2`.
- `AWS_PROFILE` The default credential and configuration profile to use, if any.

Configuration Arguments

The following arguments can be used to override the AWS configuration from all other sources:

- `--assume-role`*{role-name}* or `-R` *{role-name}* – Specifies the IAM role to assume to perform the requested actions. The credentials in the `~/.aws/credentials` file must be able to assume the specified role.
- `--aws-access-key` *{access-key}* – The AWS access key that is used.
- `--aws-secret-key` *{secret-key}* – The AWS secret key that is used.

- `--profile` *{profile-name}* or `-P` *{profile-name}* – The AWS command line tool profile that is used.

Common Arguments

Most of the `lmbr_aws` commands accept the following arguments, in addition to their own individual arguments:

- `--aws-directory` *{aws}* – Identifies the *{game}*`\AWS` directory to use. The default is the value of the `sys_game_folder` property from *{root}*`\bootstrap.cfg` with `AWS` appended.
- `--game-directory` *{directory}* – Location of the game project directory. The default is *{root}*`\`*{game}* where *{game}* is determined by the `sys_game_folder` setting in the *{root}*`\bootstrap.cfg` file.
- `--help` or `-h` – Display help for the command.
- `--no-prompt` – Suppresses calls that request user input.
- `--root-directory` *{root}* – Identifies the Lumberyard`\dev` directory. The default is the current working directory.
- `--user-directory` *{user}* – Location of the user cache directory. The default is *{root}*`\Cache\`*{game}*`\AWS` where *{game}* is determined by the `sys_game_folder` setting in the *{root}*`\bootstrap.cfg` file.
- `--verbose` – Shows additional output when executing commands.

Commands

Following are details of the `lmbr_aws` commands.

deployment create

Create a complete and independent copy of all the resources needed by the Lumberyard project.

In addition to the Common Arguments (p. 409), the `deployment create` command accepts the following arguments:

- `--deployment` *{deployment-name}* or `-d` *{deployment-name}*

  Required. The name of the deployment to create.
- `--enable-capability`*{capability}*

  Optional. A list of capabilities that you must specify before AWS CloudFormation can create or update certain stacks. Some stack templates might include resources that affect permissions in your AWS account. For those stacks, you must explicitly acknowledge their capabilities by specifying this argument. Possible values include: `CAPABILITY_IAM`.
- `--confirm-aws-usage` or `-c`

  Optional. Confirms that you know that the `deployment create` command will create AWS resources for which you may be charged and that may perform actions that can affect permissions in your AWS account. Specify this argument to disable the related confirmation prompt.

deployment default

Set or show the default user and project deployments.

In addition to the Common Arguments (p. 409), the `deployment default` command accepts the following arguments:

- `--set` *{deployment}*

  Optional. Sets the default to the provided deployment name.

- `--clear`

  Optional. Clears the defaults.

- `--show`

  Optional. Shows the defaults.

- `--project`

  Optional. Applies `--set` and `--clear` to the project default instead of the user default. Ignored for `--show`.

  Only one of the `--set`, `--clear`, and `--show` arguments is allowed.

  If `--set` or `--clear` is specified, this command updates the *{root}*`\user\AWS\user-settings.json` file. If `--project` is provided, the *{root}*`\`*{game}*`\AWS\project-settings.json` file is updated.

deployment delete

Delete a complete and independent copy of all the resources needed by the Lumberyard project.

In addition to the Common Arguments (p. 409), the `deployment delete` command accepts the following arguments:

- `--deployment` *{deployment-name}* or `-d` *{deployment-name}*

  Required. The name of the deployment to delete.

- `--enable-capability`*{capability}*

  Optional. A list of capabilities that you must specify before AWS CloudFormation can create or update certain stacks. Some stack templates might include resources that can affect permissions in your AWS account. For those stacks, you must explicitly acknowledge their capabilities by specifying this argument. Possible values include: `CAPABILITY_IAM`.

- `--confirm-resource-deletion`

  Optional. Acknowledges that the command will permanently delete the resources belonging to the specified deployment. If not specified, the user is prompted to confirm the deletion.

  Note
  AWS CloudFormation cannot delete stacks that define Amazon S3 buckets that contain data. To allow project stacks to be deleted, the `project-template.json` file specifies a `DeletionPolicy` of `Retain` for the configuration bucket. This causes AWS CloudFormation to not delete the bucket when the project stack is deleted. After the project stack has been deleted, the command removes all the objects from the configuration bucket and then deletes the bucket.

deployment list

List all deployments in the local project.

Example output:

```
Name                Status           Reason
                        Timestamp           Id
```

```
-----------------  ---------------
 ------------------------------------------------------------------------------
 -----------------
 ------------------------------------------------------------------------------------------------------------
AnotherDeployment  CREATE_PENDING   Resource is defined in the local project template but
 does not exist in AWS.
Development        CREATE_COMPLETE
                                    03/04/16 18:43:11  arn:aws:cloudformation:us-
east-2:<ACCOUNTID>:stack/foo-hw-Development-ZDLXUB7FKR94/8e6492f0-
e248-11e5-8e7e-50d5ca6e60ae


User Default Deployment:    (none)
Project Default Deployment: Development
Release Deployment:         (none)
```

deployment list-resources

List all of the resources associated with the project.

In addition to the Common Arguments (p. 409), the `deployment list-resources` command accepts the following arguments:

- `--deployment` *{deployment-name}* or `-d` *{deployment-name}*

  Optional. The name of the deployment to list resources for. If not specified, lists all the project's resources.

deployment protect

Marks a deployment as protected and issues a warning when a user (for example, a developer or tester) attempts to connected a development game client to live resources. For more information, see Using Protected Deployments (p. 391).

In addition to the Common Arguments (p. 409), the `deployment protect` command accepts the following arguments:

- `--set` *{deployment-name}*

  Optional. Specifies that the deployment is protected.

- `--clear` *{deployment-name}*

  Optional. Specifies that the deployment is not protected.

- `--show`

  Optional. Displays a list of the deployments that are currently protected.

  Note
  To display the protected status of deployments, you can also use either the deployment list (p. 410) or mappings list (p. 415) command.

deployment release

Sets, shows, or removes the release deployment.

In addition to the Common Arguments (p. 409), the `deployment release` command accepts the following arguments:

- `--set` *`{deployment-name}`*

  Required. The name of the deployment to set as the release.

- `--clear`*`{capability}`*

  Removes the release designation from the current release deployment.

- `--show`

  Shows the deployment that is currently configured as the release deployment.

deployment update-access

Sets, shows, or removes the release deployment.

In addition to the Common Arguments (p. 409), the `deployment update-access` command accepts the following arguments:

- `--confirm-aws-usage` or `-c`

  Optional. Confirms that you know this command creates AWS resources for which you can be charged and that it may perform actions that can affect permissions in your AWS account. Specify this argument to disable the related confirmation prompt.

- `--confirm-resource-deletion`

  Optional. If the operation will delete resources permanently, confirms your acknowledgement and approval. If this argument is not specified, you are prompted to confirm completion of the operation. Specify this argument to disable the related confirmation prompt.

- `--confirm-security-change`

  Optional. Confirms that you know that this command may make security-related changes. Specify this argument to disable the related confirmation prompt.

- `--deployment` *`{deployment-name}`* or `-d` *`{deployment-name}`*

  Optional. The name of the deployment whose access stack is updated. If omitted, the default deployment is updated. Use * to update all deployments.

deployment upload

Updates a deployment. `deployment update` can be used instead of `deployment upload`.

In addition to the Common Arguments (p. 409), the `deployment upload` command accepts the following arguments:

- `--confirm-aws-usage` or `-c`

  Optional. Confirms that you know this command creates AWS resources for which you can be charged and that it may perform actions that can affect permissions in your AWS account. Specify this argument to disable the related confirmation prompt.

- `--confirm-resource-deletion`

  Optional. If the operation will delete resources permanently, confirms your acknowledgement and approval. If this argument is not specified, you are prompted to confirm completion of the operation. Specify this argument to disable the related confirmation prompt.

- `--confirm-security-change`

Optional. Confirms that you know that this command may make security-related changes. Specify this argument to disable the related confirmation prompt.

- `--deployment` *{deployment-name}* or `-d` *{deployment-name}*

  Optional. The name of the deployment to update. If omitted, the default deployment is updated.

function get-log

Retrieves data from a CloudWatch Logs log file.

In addition to the Common Arguments (p. 409), the `function get-log` command accepts the following arguments:

- `--deployment` *{deployment-name}* or `-d` *{deployment-name}*

  Optional. The name of a deployment. If this argument is specified, the `--resource-group` argument must also be specified. If this argument is omitted, then the function must exist in the project stack.

- `--function` *{function-name}* or `-f` *{function-name}*

  Required. The logical name of a Lambda function resource.

- `--resource-group` *{resource-group-name}* or `-r` *{resource-group-name}*

  Optional. The name of a resource group. If specified, the `--deployment` argument must also be specified.

- `--log-stream-name` *{log-stream-name}* or `-l` *{log-stream-name}*

  Optional. The log stream name or partial log stream name. If omitted, the most recent log stream is shown.

function upload-code

asdf

In addition to the Common Arguments (p. 409), the `function upload-code` command accepts the following arguments:

- `--deployment` *{deployment-name}* or `-d` *{deployment-name}*

  Optional. The name of a deployment to update. If this argument is not specified, the default deployment is updated.

- `--function` *{function-name}*

  Required. The name of the Lambda function to update. If not specified, all Lambda functions in the resource group are updated.

- `--resource-group` *{resource-group-name}* or `-r` *{resource-group-name}*

  Optional. The name of a resource group to update. If not specified, all resource groups in the deployment are updated.

login-provider add

Add a player login provider to the Amazon Cognito identity pool configuration. Login providers allow your game's players to log in using their social network identity, such as Facebook or using their Amazon user identity. For more information, see Controlling Access to Resources (p. 393).

In addition to the Common Arguments (p. 409), the `login-provider add` command accepts the following arguments:

- `--provider` *`{provider-name}`*

  Required. The name of the provider. The name must be `amazon`, `google` or `facebook`, or, if you are using a generic OpenID provider, a name that you choose.

- `--app-id` *`{application-id}`*

  Required. The application id from your login provider (this is usually different from your client ID).

- `--client-id` *`{client-id}`*

  Required. The unique application client ID for the login provider.

- `--client-secret` *`{client-secret}`*

  Required. The secret key to use with your login provider.

- `--redirect-uri` *`{redirect-uri}`*

  Required. The redirect URI to use with your login provider.

- `--provider-uri` *`{provider-uri}`*

  Optional. The URI for a generic open ID connect provider. This is only use for generic OpenID providers.

- `--provider-port` *`{provider-port}`*

  Optional. The port your provider listens on for its API. This is only used for generic OpenID providers.

- `--provider-path` *`{provider-path}`*

  Optional. The path portion of your provider's URI. This is only used for generic OpenID providers.

  This command saves its configuration in a `player-access/auth-settings.json` object in the project's configuration bucket so that the `ProjectPlayerAccessTokenExchangeHandler` Lambda function can access it.

  Note
  You must run `project upload` after running this command so that the PlayerAccessIdentityPool (p. 289) configuration is updated to reflect the change.

login-provider remove

Remove a player login provider from the Amazon Cognito identity pool configuration.

In addition to the Common Arguments (p. 409), the `login-provider remove` command accepts the following argument:

- `--provider` *`{provider-name}`*

  Required. The name of the provider.

  The `login-provider remove` command saves the configuration in a `/player-access/auth-settings.json` object in the project's configuration bucket so that the `ProjectPlayerAccessTokenExchangeHandler` Lambda function can access it.

  Note
  You must run `lmbr_aws project upload` after running this command so that the PlayerAccessIdentityPool (p. 289) configuration is updated to reflect the change.

login-provider update

Update a player login provider in the Amazon Cognito identity pool configuration. Login providers allow your game's players to log in using their social network identity, such as Facebook, or using their Amazon user identity. For more information, see Controlling Access to Resources (p. 393).

In addition to the Common Arguments (p. 409), the `login-provider update` command accepts the following arguments:

- `--provider` *`{provider-name}`*

  Required. The name of the updated provider. The name must be `amazon`, `google` or `facebook`, or, if you are using a generic OpenID provider, the name that you chose when the provider was added.

- `--app-id` *`{application-id}`*

  Optional. The application ID from your login provider (this is usually different from your client ID).

- `--client-id` *`{client-id}`*

  Optional. The unique application client ID for the login provider.

- `--client-secret` *`{client-secret}`*

  Optional. The secret key to use with your login provider.

- `--redirect-uri` *`{redirect-uri}`*

  Optional. The redirect URI to use with your login provider.

- `--provider-uri` *`{provider-uri}`*

  Optional. The URI for a generic open id connect provider. This argument is used only for generic OpenID providers.

- `--provider-port` *`{provider-port}`*

  Optional. The port the provider listens on for the provider's API. This argument is used only for generic OpenID providers.

- `--provider-path` *`{provider-path}`*

  Optional. The path portion of the provider's URI. This argument is used only for generic OpenID providers.

  The `login-provider update` command saves its configuration in a `/player-access/auth-settings.json` object in the project's configuration bucket so that the `ProjectPlayerAccessTokenExchangeHandler` Lambda function can access it.

  Note
  You must run `lmbr_aws project upload` after running this command so that the PlayerAccessIdentityPool (p. 289) configuration is updated to reflect the change.

mappings list

Show the logical to physical resource name mappings.

Example output:

```
Name                                    Type                          Id
--------------------------------------  ----------------------------
 ------------------------------------------------------------------
```

```
HelloWorld.SayHello                       AWS::Lambda::Function        foo-hw-Development-
ZDLXUB7FKR94-HelloWo-SayHello-1FADMFNE5M1CO
PlayerAccessIdentityPool                  Custom::CloudGemFramework::CognitoIdentityPool
 us-east-2:108f6d6a-f929-4212-9947-a03269b9582e
PlayerLoginIdentityPool                   Custom::CloudGemFramework::CognitoIdentityPool
 us-east-2:3020e175-0ddd-4860-8dad-1db57162cbb2
ProjectPlayerAccessTokenExchangeHandler   AWS::Lambda::Function        foo-hw-
ProjectPlayerAccessTokenExchangeHandler-1BG6JJ94IZAUV
account_id                                Configuration                <ACCOUNTID>
region                                    Configuration                us-east-2
```

mappings update

Update the friendly name to physical resource ID mappings to reflect the current default deployment or the release deployment.

In addition to the Common Arguments (p. 409), the `mappings update` command accepts the following arguments:

- `--release`

  Optional. Causes the release mappings to be updated. By default, only the mappings used when launching the game from inside the editor are updated.

  The command looks in the resource-template.json (p. 288) file for `Metadata.CloudCanvas.PlayerAccess` properties on resource definitions. It then queries AWS CloudFormation for the physical names of those resources in the current default deployment. If the `--release` argument is specified, the release deployment is queried.
- `--deployment` *`{deployment-name}`* or `-d` *`{deployment-name}`*

  Optional. Exports a mapping file for the specified deployment to the *`{project_directory}`*`\Config` directory in the format *`{deployment-name}`*`.awsLogicalMappings.json`.

  When you run a game launcher such as the one at `dev\Bin64\SamplesProjectLauncher.exe`, you can choose the mapping to use by using the `-cc_override_resource_map` argument. For more information, see Selecting a Deployment with a PC Launcher (p. 391).

parameter clear

Clears the specified parameter configuration for your project. The project must be initialized (a project stack must have been created) before you can clear parameters.

In addition to the Common Arguments (p. 409), the `parameter clear` command accepts the following arguments:

- `--deployment` *`{deployment-name}`* or `-d` *`{deployment-name}`*

  Optional. Clears the parameter value for the specified deployment. *`{deployment-name}`* can be *, in which case the parameter value used for all deployments that do not override the value is cleared. If omitted, the parameter value is cleared for all deployments, including *.
- `--resource-group` *`{resource-group-name}`* or `-r` *`{resource-group-name}`*

  Optional. Clears the parameter value for the specified resource-group. *`{resource-group-name}`* can be *, in which case the parameter value used for all resource groups that do not override the value is cleared. If omitted, the parameter value is cleared for all resource groups, including *.
- `--parameter` *`{parameter-name}`* or `-p` *`{parameter-name}`*

  Required. The parameter to clear.

parameter list

Lists the parameters currently configured for your project. The project must be initialized (a project stack must have been created) before you can list parameters.

In addition to the Common Arguments (p. 409), the `parameter list` command accepts the following arguments:

- `--deployment` *`{deployment-name}`* or `-d` *`{deployment-name}`*

  Required. Limits the list to the specified deployment. *`{deployment-name}`* can be *, in which case parameters that apply to all deployments are listed.
- `--resource-group` *`{resource-group-name}`* or `-r` *`{resource-group-name}`*

  Required. Limits the list to the specified resource group. *`{resource-group-name}`* can be *, in which case parameters that apply to all resource groups are listed.
- `--parameter` *`{parameter-name}`* or `-p` *`{parameter-name}`*

  Optional. Limits the list to the specified parameter.

parameter set

Sets parameter configuration for your project. The project must be initialized (a project stack must have been created) before you can set parameters.

In addition to the Common Arguments (p. 409), the `parameter set` command accepts the following arguments:

- `--deployment` *`{deployment-name}`* or `-d` *`{deployment-name}`*

  Required. Sets the parameter value for the specified deployment. *`{deployment-name}`* can be *, in which case the parameter value is used for all deployments that do not override the value.
- `--resource-group` *`{resource-group-name}`* or `-r` *`{resource-group-name}`*

  Required. Sets the parameter value for the specified resource group. *`{resource-group-name}`* can be *, in which case the parameter value is used for all resource groups that do not override the value.
- `--parameter` *`{parameter-name}`* or `-p` *`{parameter-name}`*

  Required. Specifies the parameter whose value will be set.
- `--value` *`{parameter-value}`* or `-v` *`{parameter-value}`*

  Required. Specifies the value to set.

profile add

Add an AWS profile to the AWS command line tool configuration.

In addition to the Common Arguments (p. 409), the `profile add` command accepts the following arguments:

- `--aws-access-key`*`{accesskey}`*

  Required. The AWS access key associated with the added profile.
- `--aws-secret-key`*`{secretkey}`*

Required. The AWS secret key associated with the added profile.

- `--profile` *{profile-name}* or `-P` *{profile-name}*

  Required. The name of the AWS profile to add.

- `--make-default`

  Optional. Make the new profile the default profile.

profile default

Set, clear, or show the default profile in the AWS command line tool configuration.

In addition to the Common Arguments (p. 409), the `profile default` command accepts the following arguments:

- `--set` *{deploymentname}*

  Optional. Set the default profile to the provided deployment name.

- `--clear`

  Optional. Clear the default profile.

- `--show`

  Optional. Show the default profile.

profile list

List the AWS profiles that have been configured.

profile remove

Remove an AWS profile from the AWS command line tool configuration.

In addition to the Common Arguments (p. 409), the `profile remove` command accepts the following argument:

- `--profile` *{profile-name}* or `-P` *{profile-name}*

  Required. The name of the AWS profile to remove.

profile rename

Rename an AWS profile in the AWS command line tool configuration.

In addition to the Common Arguments (p. 409), the `profile rename` command accepts the following arguments:

- `--old` *{old-profile-name}*

  Required. The name of the AWS profile to change.

- `--new` *{new-profile-name}*

  Required. The new name of the AWS profile.

profile update

Update an AWS profile.

In addition to the Common Arguments (p. 409), the `profile update` command accepts the following arguments:

- `--aws-access-key`*`{accesskey}`*

  Optional. The AWS access key associated with the updated profile. The default is to not change the AWS access key associated with the profile.

- `--aws-secret-key`*`{secretkey}`*

  Optional. The AWS secret key associated with the updated profile. The default is to not change the AWS secret key associated with the profile.

- `--profile` *`{profilename}`* or `-P` *`{profile-name}`*

  Required. The name of the AWS profile to update.

  Note
  To make an existing profile the default profile, use the profile default (p. 418) command.

project create

Initialize Cloud Canvas resource management for a Lumberyard project. This includes creating a set of default Resource Definitions (p. 287) in the `dev\`*`{game}`*`\AWS` directory and a AWS CloudFormation stack that contains the resources that the Cloud Canvas resource manager uses to manage your game resources.

In addition to the Common Arguments (p. 409), the `project create` command accepts the following arguments:

- `--stack` *`{stack-name}`*

  Optional. The name used for the project's AWS CloudFormation stack. The default is the name of the *`{game}`* directory.

- `--confirm-aws-usage` or `-C`

  Optional. Confirms that you know this command will create AWS resources for which you may be charged and that it may perform actions that can affect permissions in your AWS account. Specify this argument to disable the related confirmation prompt.

- `--enable-capability` *`{capability}`* `[`*`{capability}`* `...]`

  Optional. A list of capabilities that you must specify before AWS CloudFormation can create or update certain stacks. Some stack templates might include resources that can affect permissions in your AWS account. For those stacks, you must explicitly acknowledge their capabilities by specifying this argument. Possible values include `CAPABILITY_IAM`.

- `--files-only`

  Optional. Writes the default configuration data to the *`{game}`*`\AWS` directory and exits. The directory must be empty or must not exist.

- `--region` *`{region}`*

  Required. The AWS region in which the project stack will be created.

Note
The `region` argument can be used only with the `project create` and `resource-importer list-importable-resources` commands.

How `project create` works

1. The `project create` command creates the project's AWS CloudFormation stack using a bootstrap template that defines only the Configuration Bucket (p. 297) resource.
2. The project-template.json (p. 289) file and the zipped up contents of the project-code subdirectory (p. 287) are uploaded to the Configuration Bucket (p. 297).
3. An AWS CloudFormation stack update operation is performed by using the uploaded `project-template.json` file. The `project-code .zip` file is used to create the Lambda function resources defined by the `project-template.json` file.

Note

- If the *{root}**{game}*\AWS directory is empty or does not exist, `project create` creates the directory if necessary and copies the contents of the `dev\Tools\lmbr_aws\AWSResourceManager\default-project-content` directory to that directory.
- `project create` fails if a stack with the specified name already exists in the configured AWS account and region. In this case you can use the `--stack` argument to specify a different name for the project stack.
- `project create` fails if the `dev\`*{game}*`\AWS\local-project-settings.json` file has a non-empty `ProjectStackId` property. The `ProjectStackId` property will be set to the project's AWS CloudFormation stack ID after the project stack is created in step 1.
- If the stack update process in step 2 fails on the first attempt, you can retry by using the `project upload` command.

project delete

Delete the AWS CloudFormation stack that contains your project's resources. You must delete all of the project's deployments before deleting the project stack. After deleting the project stack, you must create a new project stack before you can use AWS CloudFormation resource manager for your project.

In addition to the Common Arguments (p. 409), the `project delete` command accepts the following argument:

- `--confirm-resource-deletion`

  Optional. Confirms your acknowledgement and approval that the operation will delete resources permanently. If this argument is not specified, you will be prompted to confirm completion of the operation. Specify this argument to disable the related confirmation prompt.

  AWS CloudFormation cannot delete stacks that define Amazon S3 buckets that contain data. To allow project stacks to be deleted, the `project-template.json` file specifies a `DeletionPolicy` of `Retain` for the configuration bucket. This causes AWS CloudFormation to not delete the bucket when the project stack is deleted. After the project stack has been deleted, the command removes all the objects from the configuration bucket and then deletes the bucket.

project list-resources

List all of the resources associated with the project.

In addition to the Common Arguments (p. 409), the `project list-resources` command accepts the following argument:

- `--stack-id` *{stackid}*

  Optional. The ARN of the stack to list resources for. Defaults to project, deployment, or resource group ID as determined by the `--deployment` and `--resource-group` arguments.

project update-code

Updates the code for a project.

In addition to the Common Arguments (p. 409), the `project update-code` command accepts the following argument:

- `--confirm-replacement`

  Optional. Confirm that it is OK to replace existing files. Specify this argument to disable the related confirmation prompt.

project upload

Update the project's AWS CloudFormation stack. `project update` can be used instead of `project upload`.

In addition to the Common Arguments (p. 409), the `project upload` command accepts the following arguments:

- `--confirm-aws-usage` or `-c`

  Optional. Confirms that you know this command will create AWS resources for which you may be charged and that it may perform actions that can affect permission in your AWS account. Specify this argument to disable the related confirmation prompt.

- `--confirm-resource-deletion`

  Optional. If the operation will delete resources permanently, confirms your acknowledgement and approval. If this argument is not specified, you are prompted to confirm completion of the operation. Specify this argument to disable the related confirmation prompt.

- `--enable-capability` *{capability}* [*{capability}* ...]

  Optional. A list of capabilities that you must specify before AWS CloudFormation can create or update certain stacks. Some stack templates might include resources that can affect permissions in your AWS account. For those stacks, you must explicitly acknowledge their capabilities by specifying this argument. Possible values include `CAPABILITY_IAM`.

How `project upload` works

1. The project-template.json (p. 289) file and the zipped up contents of the project-code subdirectory (p. 287) are uploaded to the Configuration Bucket (p. 297).
2. An AWS CloudFormation stack update operation is performed by using the uploaded `project-template.json` file. The `project-code.zip` file is used when creating the Lambda function resources defined by the templates.

Note
The `project upload` command fails if the `dev\{game}\AWS\local-project-settings.json` file does not exist or does not have a valid `ProjectStackId` property.

resource-group add

Add a `ResourceGroupConfiguration` and AWS CloudFormation stack resources to your `deployment-template.json` file. For more information, see deployment-template.json (p. 289).

The command also creates a *{game}*\resource-group*{resource-group-name}* directory with a default `resource-template.json` file and `lambda-function-code` subdirectory.

In addition to the Common Arguments (p. 409), the `resource-group add` command accepts the following argument:

- `--resource-group` *{resource-group-name}* or `-r` *{resource-group-name}*

  Required. The name of the resource group to add.
- `--gem`

  Optional. Specifies that a gem resource group definition is in the `\dev\Gem\`*{resource-group-name}* directory. The *{resource-group-name}* is the resource group that is specified in the `--resource-group` argument.
- `--include-example-resources`

  Optional. Includes "Hello World" example resources.

resource-group list

List all the resource groups found in the local deployment template and in the selected deployment in AWS.

In addition to the Common Arguments (p. 409), the `resource-group list` command accepts the following argument:

- `--deployment` *{deployment-name}* or `-d` *{deployment-name}*

  Optional. The name of the deployment to list resource groups for. If not given, the default deployment is used.

Example output:

```
Name             Status           Reason
                        Timestamp          Id
--------------  ---------------
 ------------------------------------------------------------------------------
 -----------------
 ------------------------------------------------------------------------------------------------------------------------
AnotherResourceGroup  CREATE_PENDING   Resource is defined in the local deployment template
 but does not exist in AWS.
HelloWorld       CREATE_COMPLETE
                                    03/04/16 18:42:57  arn:aws:cloudformation:us-
east-2:<ACCOUNTID>:stack/foo-hw-Development-ZDLXUB7FKR94-HelloWorld-WSGZ15EUWX52/9b909d20-
e238-11e5-a98d-50fae987c09a
```

resource-group list-resources

List all of the resources associated with the project.

In addition to the Common Arguments (p. 409), the `resource-group list-resources` command accepts the following arguments:

- `--deployment` *`{deployment-name}`* or `-d` *`{deployment-name}`*

  Optional. The name of the deployment to list resources for. If not specified, lists all the project's resources.
- `--resource-group` *`{resource-group-name}`* or `-r` *`{resource-group-name}`*

  Optional. The name of the resource group to list resources for. If specified, `deployment` must also be specified. If not specified, all deployment or project resources are listed.

resource-group remove

Remove a resource group's `ResourceGroupConfiguration` and AWS CloudFormation stack resources from your `deployment-template.json` file. You must update your deployment stacks before the resources defined by your resource group can be removed from AWS.

The command does not delete the *`{game}`*\resource-group*`{resource-group-name}`* directory.

In addition to the Common Arguments (p. 409), the `resource-group remove` command accepts the following argument:

- `--resource-group` *`{resource-group-name}`* or `-r` *`{resource-group-name}`*

  Required. The name of the resource group to be removed.

AWS CloudFormation cannot delete stacks that define Amazon S3 buckets that contain data. To allow project stacks to be deleted, the `project-template.json` file specifies a `DeletionPolicy` of `Retain` for the configuration bucket. This causes AWS CloudFormation to not delete the bucket when the project stack is deleted. After the project stack has been deleted, the command removes all the objects from the configuration bucket and then deletes the bucket.

resource-group upload

Update a resource group's AWS CloudFormation stack in a deployment. `resource-group update` can be used instead of `resource-group upload`.

In addition to the Common Arguments (p. 409), the `resource-group upload` command accepts the following arguments:

- `--confirm-aws-usage` or `-c`

  Optional. Confirms that you know this command will create AWS resources for which you may be charged and that it may perform actions that can affect permissions in your AWS account. Specify this argument to disable the related confirmation prompt.
- `--confirm-resource-deletion`

  Optional. If the operation will delete resources permanently, confirms your acknowledgement and approval. Specify this argument to disable the related confirmation prompt.
- `--confirm-security-change`

  Optional. Confirms that you know that this command may make security-related changes. Specify this argument to disable the related confirmation prompt.
- `--deployment` *`{deployment-name}`* or `-d` *`{deployment-name}`*

  Optional. The name of the deployment to update. If not specified, the default deployment is updated.

- `--resource-group` *{resource-group-name}*, or `-r` *{resource-group-name}*

  Required. The name of the resource group to update. If not specified, all resource groups in the deployment are updated.

resource-importer import-resource

Import a resource to a resource group.

In addition to the Common Arguments (p. 409), the `resource-importer import-resource` command accepts the following arguments:

- `--type` *{dynamodb|s3|lambda|sns|sqs}*

  Optional. The type of the AWS resource to import. Choose from **`dynamodb`**, **`s3`**, **`lambda`**, **`sns`** or **`sqs`**.
- `--arn` *ARN*

  Required. The ARN of the AWS resource to import.
- `--resource-name` *{resource-name}*

  Required. The name of the resource to import.
- `--resource-group` *{resource-group-name}* or `-r` *{resource-group-name}*

  Required. The name of the resource group to import.
- `--download`

  Optional. If specified, downloads the contents of the Amazon S3 bucket.

resource-importer list-importable-resources

List all supported resources currently existing on AWS.

In addition to the Common Arguments (p. 409), the `resource-importer list-importable-resources` command accepts the following arguments:

- `--type` *{dynamodb|s3|lambda|sns|sqs}*

  Required. The type of the AWS resource to list. Choose from **`dynamodb`**, **`s3`**, **`lambda`**, **`sns`** or **`sqs`**.
- `--region` *{region}*

  Optional. The AWS region of the resources. The default value is the region of the project stack, if it exists.

  Note
  The `region` argument can be used only with the `resource-importer list-importable-resources` and `project create` commands.

Command Reorganization

As of Lumberyard 1.9, the commands made available by the `lmbr_aws` command line tool have been reorganized. The previous commands are still functional but are subject to removal in a future release.

The following table shows how the previous commands map to the new commands. In a few cases single commands have been split into two or more commands for clarity and ease of use. These are

commented accordingly. For information about the commands, see Using the Cloud Canvas Command Line (p. 408).

Previous Command	New Command
`add#login#provider`	`login#provider add`
`add#profile`	`profile add`
`add#resource#group`	`resource#group add`
`clear#parameter`	`parameter clear`
`create#deployment`	`deployment create`
`create#project#stack`	`project create`
`default#deployment`	`deployment default`
`default#profile`	`profile default`
`delete#deployment`	`deployment delete`
`delete#project#stack`	`project delete`
`get#function#log`	`function get#log`
`import#resource`	`resource#importer import#resource`
`list#deployments`	`deployment list`
`list#importable#resources`	`resource#importer list#importable#resources`
`list#mappings`	`mappings list`
`list#parameters`	`parameter list`
`list#profiles`	`profile list`
`list#resource#groups`	`resource#group list`

`list#resources`	`project list#resources`
	
	`deployment list#resources`
	
	`resource#group list#resources`
	
	The `list#resources` command supported `##deployment` and `##resource#group` arguments. It also supported a `##stack#id` argument.
	
	The `project list#resources` command is the same as `list#resources` without the `##deployment` and `##resource#group` arguments. The `project list#resources` command still supports the `##stack#id` argument.
	
	The `deployment list#resources` command is the same as `list#resources` with only the `##deployment` argument. The `deployment list#resources` command does not support the `##stack#id` argument.
	
	The `resource#group list#resources` command is the same as `list#resources` with both the `##deployment` and `##resource#group` arguments. The

Previous Command	New Command
	`resource#group list#resources` command does not support the `##stack#id` argument.
`protect#deployment`	`deployment protect`
`release#deployment`	`deployment release`
`remove#login#provider`	`login#provider remove`
`remove#profile`	`profile remove`
`remove#resource#group`	`resource#group remove`
`rename#profile`	`profile rename`
`set#parameter`	`parameter set`
`update#deployment#access`	`deployment update#access`
`update#login#provider`	`login#provider update`
`update#mappings`	`mappings update`
`update#profile`	`profile update`
`update#project#code`	`project update#code`
`update#project#stack`	`project upload` You can use `project update` instead of `project upload`.
`upload#lambda#code`	`function upload#code`

`upload#resources`	`deployment upload`
	
	`resource#group upload`
	
	The `upload#resources` command supported a `##resource#group` argument.
	
	The `deployment upload` command is the same as `upload#resources` without the `##resource#group` argument.
	
	The `resource#group upload` command is the same as `upload#resources` with the `##resource#group` argument.
	
	You can use `deployment update` and `resource#group update` instead of `deployment upload` and `resource#group upload`.

Argument Aliases

As a convenience, aliases have been added for the following common arguments.

Argument	Alias Equivalent
`##assume#role`	`#R`
`##confirm#aws#usage`	`#C`

Argument	Alias Equivalent
##deployment	#d
##function	#f
##log#stream#name	#l
##parameter	#p
##profile	#P
##resource#group	#r
##value	#v

Component Entity System

> The component entity system is currently in preview and is undergoing active development. It will replace the legacy Entity System (p. 520).

Amazon Lumberyard's component entity system provides a modular and intuitive way to construct games. The component entity system works at both the system level and the entity level. It employs reflection, serialization, event bus (EBus) messaging, and fully cascading prefabs (slices).

This section of the Amazon Lumberyard Developer Guide offers a Programmer's Guide to Entities and Components (p. 428) that covers these topics in depth. For information on using the component entity system in Lumberyard Editor, see Component Entity System in the Amazon Lumberyard User Guide.

Topics

- Programmer's Guide to Entities and Components (p. 428)
- Behavior Context (p. 453)
- Gameplay Bus (p. 458)
- Slices and Dynamic Slices (p. 461)

Programmer's Guide to Entities and Components

> The component entity system is currently in preview and is undergoing active development. It will replace the legacy Entity System (p. 520).

This guide provides engine and game programmers with examples and best practices for creating and reflecting custom Lumberyard components in C++. For information on using the component entity system in Lumberyard Editor, see Component Entity System in the Amazon Lumberyard User Guide.

Topics

- Creating a Component (p. 429)
- Registering Your Component (p. 431)
- Reflecting a Component for Serialization and Editing (p. 431)

- Defining and Using Component Services (p. 436)
- Editor Components (p. 437)
- Creating System Components (p. 440)
- Components and EBuses (p. 442)
- Tick Bus and Components (p. 445)
- Exposing Custom Components to Track View for Animation (p. 446)
- Components and EBuses: Best Practices (p. 451)

Creating a Component

A component in Lumberyard is a simple class that inherits from Lumberyard's `AZ::Component`. A component's behavior is determined by its reflected data and the actions it takes when it is activated. This section shows you how to create Lumberyard components programmatically. For information about adding and customizing the components available in Lumberyard Editor, see Component Entity System in the Amazon Lumberyard User Guide.

Component Example

An example component class skeleton follows:

```
class MyComponent
      : public AZ::Component
{
public:
      AZ_COMPONENT(MyComponent, "{0C09F774-DECA-40C4-8B54-3A93033EC381}");

      // AZ::Component interface implementation.
      void Init() override            {}
      void Activate() override      {}
      void Deactivate() override      {}

      // Required Reflect function.
      static void Reflect(AZ::ReflectContext* context);

      // Optional functions for defining provided and dependent services.
      static void GetProvidedServices(AZ::ComponentDescriptor::DependencyArrayType&
 provided);
      static void GetDependentServices(AZ::ComponentDescriptor::DependencyArrayType&
 dependent);
      static void GetRequiredServices(AZ::ComponentDescriptor::DependencyArrayType&
 required);
      static void GetIncompatibleServices(AZ::ComponentDescriptor::DependencyArrayType&
 incompatible);
};
```

Component Members

The required and optional members that a component comprises are as follows:

AZ::Component

Every component must include `AZ::Component` somewhere in its inheritance ancestry. Noneditor components generally inherit directly from `AZ::Component`, as in the following example:

```
class MyComponent
      : public AZ::Component
```

You can also create your own component class hierarchies.

AZ_COMPONENT Macro

Every component must specify the `AZ_COMPONENT` macro in its class definition. The macro takes two arguments:

1. The component type name.
2. A unique UUID. You may use any UUID generator to produce the value. Visual Studio provides this functionality through **Tools**, **Create GUID**. Use the **Registry Format** setting, and then copy and paste the value that is generated.

A sample `AZ_COMPONENT` macro follows:

```
AZ_COMPONENT(MyComponent, "{0C09F774-DECA-40C4-8B54-3A93033EC381}");
```

AZ::Component Functions

To define a component's behavior, you generally override three `AZ::Component` functions: `Init`, `Activate`, and `Deactivate`:

```
void Init() override       {}
void Activate() override   {}
void Deactivate() override {}
```

These functions are as described as follows:

Init()

(Optional) Called only once for a given entity. It requires minimal construction or setup work, since the component may not be activated anytime soon. An important best practice is to minimize your component's CPU and memory overhead while the component is inactive.

Activate()

(Required) Called when the owning entity is being activated. The system calls your component's `Activate()` function only if all dependent or required services are present. Your `Activate` function is always called after any components that it depends on. In addition, the component makeup of an entity never changes while the entity is active. Consequently, it is safe to cache pointers or references to other components on the entity when performance is critical.

Deactivate()

(Required) Called when the owning entity is being deactivated. The order of deactivation is the reverse of activation, so your component is deactivated before the components it depends on. As a best practice, make sure your component returns to a minimal footprint when it is deactivated. In general, deactivation should be symmetric to activation.

Note
Deactivation does not necessarily precede destruction. An entity can be deactivated and then activated again without being destroyed, so ensure that your components support this efficiently. However, when you do destroy your entity, Lumberyard ensures that your `Deactivate()` function is called first. Components must be authored with this in mind.

Reflect()

(Required) All components are AZ reflected classes. Because all components must be serializable and editable, they must contain a `Reflect()` function, as in the following example:

```
// Required Reflect function.
```

```
static void Reflect(AZ::ReflectContext* context);
```

For more information, see Reflecting a Component for Serialization and Editing (p. 431).

Logical Services

(Optional) Components can define any combination of logical services that they provide, depend on, require, or are incompatible with. To define these logical services, use the following functions:

```
// Optional functions for defining provided and dependent services.
static void GetProvidedServices(AZ::ComponentDescriptor::DependencyArrayType&
 provided);
static void GetDependentServices(AZ::ComponentDescriptor::DependencyArrayType&
 dependent);
static void GetRequiredServices(AZ::ComponentDescriptor::DependencyArrayType&
 required);
static void GetIncompatibleServices(AZ::ComponentDescriptor::DependencyArrayType&
 incompatible);
```

Registering Your Component

You register a component during application startup. You implement the registration in the `AZ::Module class` of the module in which the component is written.

The following example module creates descriptors for the components that are declared within the module.

```
//MyModule.cpp

MyModule::MyModule()
{
    // Create descriptors for components declared within this module.
    m_descriptors.insert(m_descriptors.end(), {
        MyComponent::CreateDescriptor(),
    });
}
```

For more information, see the Module Class section of the AZ Modules (p. 159) documentation.

Reflecting a Component for Serialization and Editing

Components use AZ reflection to describe the data they serialize and how content creators interact with them.

The following example reflects a component for serialization and editing:

```
class MyComponent
    : public AZ::Component
{
    ...

    enum class SomeEnum
    {
        EnumValue1,
        EnumValue2,
    }
    float m_someFloatField;
    AZStd::string m_someStringField;
```

```
    SomeEnum m_someEnumField;
    AZStd::vector<SomeClassThatSomeoneHasReflected> m_things;

    int m_runtimeStateNoSerialize;
}

/*static*/ void MyComponent::Reflect(AZ::ReflectContext* context)
{
    AZ::SerializeContext* serialize = azrtti_cast<AZ::SerializeContext*>(context);
    if (serialize)
    {
        // Reflect the class fields that you want to serialize.
        // In this example, m_runtimeStateNoSerialize is not reflected for serialization.
        serialize->Class<MyComponent>()
            ->Version(1)
            ->Field("SomeFloat", &MyComponent::m_someFloatField)
            ->Field("SomeString", &MyComponent::m_someStringField)
            ->Field("Things", &MyComponent::m_things)
            ->Field("SomeEnum", &MyComponent::m_someEnumField)
            ;

        AZ::EditContext* edit = serialize->GetEditContext();
        if (edit)
        {
            edit->Class<MyComponent>("My Component", "The World's Most Clever Component")
                ->DataElement(AZ::Edit::DefaultHandler, &MyComponent::m_someFloatField,
 "Some Float", "This is a float that means X.")
                ->DataElement(AZ::Edit::DefaultHandler, &MyComponent::m_someStringField,
 "Some String", "This is a string that means Y.")
                ->DataElement("ComboBox", &MyComponent::m_someEnumField, "Choose an Enum",
 "Pick an option among a set of enum values.")
                    ->EnumAttribute(MyComponent::SomeEnum::EnumValue1, "Value 1")
                    ->EnumAttribute(MyComponent::SomeEnum::EnumValue2, "Value 2")
                ->DataElement(AZ::Edit::DefaultHandler, &MyComponent::m_things, "Bunch of
 Things", "A list of things for doing Z.")
            ;
        }
    }
}
```

The preceding example adds five data members to `MyComponent`. The first four data members will be serialized. The last data member will not be serialized because it contains only the run-time state. This is typical; components commonly contain some members that are serialized and others that are not.

It is common for fields to be reflected for serialization, but not for editing, when using advanced reflection features such as change callbacks (p. 435). In these cases, components may conduct complex internal calculations based on user property changes. The result of these calculations must be serialized but not exposed for editing. In such a case, you reflect the field to `SerializeContext` but do not add an entry in `EditContext`. An example follows:

```
serialize->Class<MyComponent>()
    ->Version(1)
    ...
    ->Field("SomeFloat", &MyComponent::m_someFloatField)
    ->Field("MoreData", &MyComponent::m_moreData)
    ...
    ;

...

AZ::EditContext* edit = serialize->GetEditContext();
if (edit)
{
    edit->Class<MyComponent>("My Component", "The World's Most Clever Component")
```

```
        ->DataElement(AZ::Edit::DefaultHandler, &MyComponent::m_someFloatField, "Some
 Float", "This is a float that means X.")
            ->EnumAttribute("ChangeNotify", &MyComponent::CalculateMoreData)
        // m_moreData is not reflected for editing directly.
        ;
}
```

Lumberyard has reflection contexts for different purposes, including the following:

- `SerializeContext` – Contains reflection data for serialization and construction of objects.
- `EditContext` – Contains reflection data for visual editing of objects.
- `BehaviorContext` – Contains reflection for run-time manipulation of objects from Lua, flow graph, or other external sources.
- `NetworkContext` – Contains reflection for networking purposes, including marshaling, quantization, and extrapolation.

> **Note**
> This guide covers only `SerializeContext` and `EditContext`.

All of Lumberyard's reflection API operations are designed to be simple, human readable, and human writable, with no forced dependency on code generation.

A component's `Reflect()` function is invoked automatically for all relevant contexts.

The following code dynamically casts the anonymous context provided to a serialize context, which is how components discern the type of context that `Reflect()` is being called for.

```
AZ::SerializeContext* serialize = azrtti_cast<AZ::SerializeContext*>(context);
```

Serialization

Reflecting a class for serialization involves a builder pattern style markup in C++, as follows:

```
serialize->Class<TestAsset>()
         ->Version(1)
         ->Field("SomeFloat", &MyComponent::m_someFloatField)
         ->Field("SomeString", &MyComponent::m_someStringField)
         ->Field("Things", &MyComponent::m_things)
         ->Field("SomeEnum", &MyComponent::m_someEnumField)
         ;
```

The example specifies that `m_someFloatField`, `m_someStringField`, `m_things`, and `m_someEnumField` should all be serialized with the component. Field names must be unique and are not user facing.

> **Tip**
> We recommend that you keep field names simple for future proofing. If your component undergoes significant changes and you want to write a data converter to maintain backward data compatibility, you must reference the field names directly.

The preceding example reflects two primitive types—a float, and a string—as well as a container (vector) of some structure. AZ reflection, serialization, and editing natively support a wide variety of types:

- Primitive types, including integers (signed and unsigned, all sizes), floats, and strings
- Enums
- `AZStd` containers (flat and associative), including `AZStd::vector`, `AZStd::list`, `AZStd::map`, `AZStd::unordered_map`, `AZStd::set`, `AZStd::unordered_set`, `AZStd:pair`, `AZStd::bitset`, `AZStd::array`, fixed C-style arrays, and others.

- Pointers, including `AZStd::smart_ptr`, `AZStd::intrusive_ptr`, and raw native pointers.
- Any class or structure that has also been reflected.

> **Note**
> The example omits the reflection code for `SomeClassThatSomeoneHasReflected`. However, you need only reflect the class. After that, you can freely reflect members or containers of that class in other classes.

Editing

When you run Lumberyard tools such as Lumberyard Editor, an `EditContext` and a `SerializeContext` are provided. You can use the robust facilities in these contexts to expose your fields to content creators.

The following code demonstrates basic edit context reflection:

```
AZ::EditContext* edit = serialize->GetEditContext();
if (edit)
{
    edit->Class<TestAsset>("My Component", "The World's Most Clever Component")
        ->DataElement(AZ::Edit::DefaultHandler, &MyComponent::m_someFloatField, "Some
 Float", "This is a float that means X.")
        ->DataElement(AZ::Edit::DefaultHandler, &MyComponent::m_someStringField, "Some
 String", "This is a string that means Y.")
        ->DataElement("ComboBox", &MyComponent::m_someEnumField, "Choose an Enum", "Pick an
 option among a set of enum values.")
            ->EnumAttribute(MyComponent::SomeEnum::EnumValue1, "Value 1")
            ->EnumAttribute(MyComponent::SomeEnum::EnumValue2, "Value 2")
        ->DataElement(AZ::Edit::DefaultHandler, &MyComponent::m_things, "Bunch of Things",
 "A list of things for doing Z.")
    ;
}
```

Although this example demonstrates the simplest usage, many features and options are available when you reflect structures (including components) to the edit context. For the fields to be exposed directly to content creators, the example provides a friendly name and a description (tooltip) as the third and fourth parameters of `DataElement`. For three fields, the first parameter of `DataElement` is the default UI handler `AZ::Edit::DefaultHandler`. The property system's architecture supports the ability to add any number of UI handlers, each valid for one or more field types. A given type can have multiple available handlers, with one handler designated as the default. For example, floats by default use the `SpinBox` handler, but a `Slider` handler is also available.

An example of binding a float to a slider follows:

```
->DataElement("Slider", &MyComponent::m_someFloatField, "Some Float", "This is a float that
 means X.")
      ->Attribute("Min", 0.f)
      ->Attribute("Max", 10.f)
      ->Attribute("Step", 0.1f)
```

The `Slider` UI handler expects `Min` and `Max` attributes. Optionally, a value for `Step` may also be provided. The example provides incremental increases of `0.1`. If no `Step` value is provided, a default stepping of `1.0` is used.

> **Note**
> The property system supports external UI handlers, so you can implement your own UI handlers in your own modules. You can customize the behavior of the field, the `Qt` control that it uses, and the attributes that it observes.

Attributes

The example also demonstrates the use of attributes. Attributes are a generic construct on the edit context that allows the binding of literals, or functions that return values, to a named attribute. UI handlers can retrieve this data and use it to drive their functionality.

Attribute values can be bound to the following:

Literal values

```
    Attribute("Min", 0.f)
```

Static or global variables

```
    Attribute("Min", &g_globalMin)
```

Member variables

```
    Attribute("Min", &MyComponent::m_min)
```

Static or global functions

```
    Attribute(AZ::Edit::Attributes::ChangeNotify, &SomeGlobalFunction)
```

Member functions

```
    Attribute(AZ::Edit::Attributes::ChangeNotify, &MyComponent::SomeMemberFunction)
```

Change Notification Callbacks

Another commonly used feature of the edit context is its ability to bind a change notification callback:

```
->DataElement(AZ::Edit::DefaultHandler, &MyComponent::m_someStringField, "Some String",
 "This is a string that means Y.")
    ->Attribute("ChangeNotify", &MyComponent::OnStringFieldChanged)
```

The example binds a member function to be invoked when this property is changed, which allows the component to conduct other logic. The `ChangeNotify` attribute also looks for an optional returned value that tells the property system if it needs to refresh aspects of its state. For example, if your change callback modifies other internal data that affects the property system, you can request a value refresh. If your callback modifies data that requires attributes be reevaluated (and any bound functions be reinvoked), you can request a refresh of attributes and values. Finally, if your callback conducts work that requires a full refresh (this is not typical), you can refresh the entire state.

The following example causes the property grid to refresh values when `m_someStringField` is modified through the property grid. `RefreshValues` signals the property grid to update the GUI with changes to the underlying data.

```
->DataElement(AZ::Edit::DefaultHandler, &MyComponent::m_someStringField, "Some String",
 "This is a string that means Y.")
    ->Attribute("ChangeNotify", &MyComponent::OnStringFieldChanged)
...
AZ::u32 MyComponent::OnStringFieldChanged()
{
    m_someFloatField = 10.0f;

    // We've internally changed displayed data, so tell the property grid to refresh values
 (cheap).
    return AZ_CRC("RefreshValues");
}
```

`RefreshValues` is one of three refresh modes that you can use:

- `RefreshValues` – Refreshes only values. The property grid updates the GUI to reflect changes to underlying data that may have occurred in the change callback.
- `RefreshAttributesAndValues` – Refreshes values but also reevaluates attributes. Since attributes can be bound to data members, member functions, global functions, or static variables, it's sometimes necessary to ask the property grid to reevaluate them. Doing so might include reinvoking bound functions.
- `RefreshAll` – Completely reevaluates the property grid. This is seldom needed, as `RefreshAttributesAndValues` should cover all requirements for rich dynamic editor reflection.

The following more complex example binds a list of strings as options for a combo box. The list of strings is attached to a string field *Property A*. Suppose you want to modify the options available in the combo box for Property A with the values from another *Property B*. In that case you can bind the combo box `StringList` attribute to a member function that computes and returns the list of options. In the `ChangeNotify` attribute for Property B, you tell the system to reevaluate attributes, which in turn reinvokes the function that computes the list of options.

```
...

bool m_enableAdvancedOptions;
AZStd::string m_useOption;

...

->DataElement(AZ::Edit::DefaultHandler, &MyComponent::m_enableAdvancedOptions, "Enable
 Advanced Options", "If set, advanced options will be shown.")
    ->Attribute("ChangeNotify", AZ_CRC("RefreshAttributesAndValues"))
->DataElement("ComboBox", &MyComponent::m_useOption, "Options", "Available options.")
    ->Attribute("StringList", &MyComponent::GetEnabledOptions)
...

AZStd::vector<const char*> MyComponent::GetEnabledOptions()
{
    AZStd::vector<const char*> options;
    options.reserve(16);

    options.push_back("Basic option");
    options.push_back("Another basic option");

    if (m_enableAdvancedOptions)
    {
        options.push_back("Advanced option");
        options.push_back("Another advanced option");
    }

    return options;
}
```

Defining and Using Component Services

Components can optionally specify a list of services that they provide, are incompatible with, depend on, or require. When you create a component, you can use this service specification to define relationships between various components. The component entity system uses this list for the conditional addition and removal of components at both edit time and at run time. The service specification also defines the order in which components are activated when an entity is activated. Specifically, components that provide services that another component depends on are activated first.

The following example shows a service specification.

```
static void GetProvidedServices(AZ::ComponentDescriptor::DependencyArrayType& provided)
```

```
{
      provided.push_back(AZ_CRC("ProvidedService"));
      provided.push_back(AZ_CRC("AnotherProvidedService"));
}

static void GetRequiredServices(AZ::ComponentDescriptor::DependencyArrayType& required)
{
      required.push_back(AZ_CRC("RequiredService"));
      required.push_back(AZ_CRC("AnotherRequiredService"));
}

static void GetIncompatibleServices(AZ::ComponentDescriptor::DependencyArrayType&
 incompatible)
{
      incompatible.push_back(AZ_CRC("IncompatibleService"));
}

static void GetDependentServices(AZ::ComponentDescriptor::DependencyArrayType& dependent)
{
      dependent.push_back(AZ_CRC("DependentOnService"));
}
```

ProvidedService – Specifies the service that the component provides. For example, a `TransformComponent` could provide a `TransformService` that in turn provides locational information.

RequiredService – Specifies a service that the component requires. The components that provide the required services are guaranteed to be present and active before this component is activated. For example, an audio component might need to know where it is located and therefore require a `TransformService`. Because of this requirement, the audio component can be added only to entities that have the component that provides the `TransformService`.

DependentService – Specifies a service on which the component depends but does not require. The component entity system guarantees that the components that provide dependent services are activated before the component itself is activated. For example, an audio component could depend on the `physics` component. If the entity has physics, the audio component can query the `physics` component for physical material information. However, the audio component does not require that physics be present.

IncompatibleService – Specifies a service that cannot work with the component. Consider these examples:

- An entity can have only one type of collider. Therefore, the `PrimitiveColliderService` specifies that the `MeshColliderService` is incompatible with it and vice versa.
- The same effect can be achieved if two collider components already provide the `ColliderService` themselves and therefore specify the `ColliderService` as incompatible. Marking a component as incompatible with `ColliderService` ensures that no other component that has the same service is added to the entity.
- The `IncompatibleService` specification is frequently used to specify that multiples of the same component cannot exist on an entity.

Editor Components

Some components in Lumberyard have separate `editor` and `runtime` versions. The editor version is active in the editor. The run-time version is used for running the level in game or in the editor by pressing **Ctrl+G** or clicking **AI/Physics** below the viewport. Lumberyard uses editor components to maintain a clean separation between tools-specific code and data on one hand, and leaner run-time component data on the other. In general, run-time game components do not require editor counterparts. Components rarely need to be fully active at edit time. The light and mesh components are exceptions because they must behave the same at edit time as at run time.

`EditContext` reflection is fully supported in run-time components. Edit time is the only time when editor components are active. At run time, when Lumberyard processes a level or dynamic slice, it uses the run-time equivalents of editor components. Using the `EditContext` from a run-time component is usually sufficient to provide a rich editing experience.

Important
Editor components are not required. An editor component is necessary only if one of the following is true:

- Your component must be fully active at edit time. Edit time refers to standard editing mode; run-time components are used for the **AI/Physics** mode and gameplay (**Ctrl-G**).
- You must add special tools functionality to your component that requires that you compile only into your editor binaries.
- Your component provides functionality only in the editor and does not export a run-time component (for example, if your component manages selection logic).

Sample Editor Component

The following code shows a sample editor component.

```
class MyEditorComponent
      : public AzToolsFramework::Components::EditorComponentBase
      , private AzFramework::EntityDebugDisplayEventBus::Handler
{
public:
      AZ_EDITOR_COMPONENT(MyEditorComponent, "{5034A7F3-63DB-4298-83AA-915AB23EFEA0}");

      // AZ::Component interface implementation.
      void Init() override            {}
      void Activate() override      {}
      void Deactivate() override      {}


      // AzFramework::EntityDebugDisplayEventBus::Handler.
      void DisplayEntity(bool& handled) override;

      // Required Reflect function.
      static void Reflect(AZ::ReflectContext* context);

      // Optional functions for defining provided and dependent services.
      static void GetProvidedServices(AZ::ComponentDescriptor::DependencyArrayType&
 provided)
      static void GetDependentServices(AZ::ComponentDescriptor::DependencyArrayType&
 dependent);
      static void GetRequiredServices(AZ::ComponentDescriptor::DependencyArrayType&
 required);
      static void GetIncompatibleServices(AZ::ComponentDescriptor::DependencyArrayType&
 incompatible);


      void BuildGameEntity(AZ::Entity* gameEntity) override;

};
```

Editor Component and Run-Time Component Differences

The code for editor components is similar to the code for run-time components. The following sections list the key differences. It is safe to assume that editor component code is the same as it is for run-time component code other than the differences listed. For more information, see Creating a Component (p. 429).

Base Classes

All editor components include the `AzToolsFramework::Components::EditorComponentBase` class somewhere in their inheritance ancestry. If a component must display edit-time visualization, it must be a handler on the `AzFramework::EntityDebugDisplayEventBus::Handler` bus, as in the following example.

```
class MyComponent
      : public AzToolsFramework::Components::EditorComponentBase
      , private AzFramework::EntityDebugDisplayEventBus::Handler
```

Macro

Every editor component must specify the `AZ_EDITOR_COMPONENT` macro within its class definition. The macro takes two arguments:

1. The component type name.
2. A unique UUID. You may use any UUID generator to produce the value. Visual Studio provides this functionality through **Tools**, **Create GUID**. Use the **Registry Format** setting, and then copy and paste the value that is generated.

A sample `AZ_EDITOR_COMPONENT` macro follows.

```
AZ_EDITOR_COMPONENT(MyEditorComponent, "{5034A7F3-63DB-4298-83AA-915AB23EFEA0}");
```

Note
Some older editor components specify `AzToolsFramework::Components::EditorComponentBase` as the base class but use the `AZ_COMPONENT` instead of the `AZ_EDITOR_COMPONENT` macro, as in the following example.

```
AZ_COMPONENT(EditorMannequinComponent, "{C5E08FE6-E1FC-4080-A053-2C65A667FE82}",
 AzToolsFramework::Components::EditorComponentBase);
```

The DisplayEntity Method

To render special visualizations in the editor, implement the `DisplayEntity` method of the `AzFramework::EntityDebugDisplayEventBus` interface. Use this location for custom primitive edit-time visualization code.

```
// AzFramework::EntityDebugDisplayEventBus::Handler
void DisplayEntity(bool& handled) override;
```

The BuildGameEntity Method

The `BuildGameEntity` method facilitates the translation of an editor component into a run-time component. Its syntax is as follows.

```
void BuildGameEntity(AZ::Entity* gameEntity) override;
```

A typical implementation of the `BuildGameEntity` method performs the following actions:

1. Create a run-time component based on the editor component of the specified `gameEntity`.
2. Copy the configuration data from the editor component into the run-time component.
3. Add the run-time component to the `gameEntity` that was specified.

At this point, the run-time component serializes the `gameEntity` and reloads it to create a new, clean version of the run-time entities.

The Transform Component Example

The `TransformComponent` is a good example of how editor and run-time components can differ. In the run-time component, values are stored in a fully composed `AZ::Transform`. In the editor component, values are stored in decomposed format. Position, rotation, and scale values are stored separately, and rotation is represented as Euler angles. This difference in format enables the editor component to provide user-friendly display and storage while providing optimal storage in the run-time component.

Creating System Components

System components are similar to other components in Lumberyard's component entity framework. However, instead of creating game entity behavior, they control the behavior of the engine itself. Currently Lumberyard enables the creation of custom system components through AZ modules and gems. (Gems are a specialization of AZ modules; for more information, see Gems and AZ Modules).

Most games organize their game code in one or more gems that are specific to the game. These gems can contain both components that you can use on game entities and system components that you want to integrate with the engine. The system components that a given gem or module specifies are first-class elements of the game engine and are included at a deep level early in the initialization process. For more information, see System Components (p. 166) in the AZ Modules (p. 159) section.

Like any Lumberyard component, a system component can provide services and can be dependent on or require other system component services. This is an elegant way to control engine initialization order and system dependencies.

When you author system components, follow the best practices for component authoring. For example, your system components should use the following:

- EBuses (p. 543) to expose their interfaces
- Reflection (p. 431) to serialize and edit settings in the Advanced Settings dialog of the Project Configurator (p. 173)
- The same AZ::Component Functions (p. 430) for activation and deactivation

> **Important**
> Just like game components, system components often provide request and notification buses. However, because system components are global systems, they should not specify IDs for their buses like game components do. Your users should be able to call your system's EBuses without having to deal with or know about the system entity that contains all system components.

The following code example shows a system component EBus.

```
class AnimationGraphAssetRequests : public AZ::EBusTraits
{
    public:
        virtual ~AnimationGraphAssetRequests() = default;
        virtual AZStd::vector<AZ::Uuid> GetGraphNodeTypes(AZ::SerializeContext&
 serializeContext) = 0;
        virtual GraphAssetPtr CreateGraphAsset() = 0;
        virtual NodeId CreateGraphNode(const GraphAssetPtr& asset, const AZ::Uuid&
 nodeTypeId, AZ::SerializeContext& serializeContext) = 0;
        virtual bool DeleteGraphNode(const GraphAssetPtr& asset, NodeId nodeId) = 0;
        virtual NodePtr FindGraphNode(const GraphAssetPtr& asset, NodeId nodeId) = 0;
};
using AnimationGraphAssetRequestBus = AZ::EBus<AnimationGraphAssetRequests>;
```

The following code shows part of the system component itself.

```
class AnimationGraphSystemComponent
      : public AZ::Component
      , private AnimationGraphSystemRequestBus::Handler
{
public:
      AZ_COMPONENT(AnimationGraphSystemComponent, "{2D497170-E4C7-40B0-
A9D1-2D987CC8932A}");
      AnimationGraphSystemComponent();
      ~AnimationGraphSystemComponent() override;

      // AnimationGraphSystemRequestBus::Handler
      AZStd::intrusive_ptr<PoseBufferStorage> AllocatePoseBufferStorage(AZ::u32 jointCount)
 override;
      void FreePoseBufferStorage(PoseBufferStorage* storage) override;
      AZ::u32 GetActivePoseBufferCount() override;
      void ExecuteGraph(const AZ::Data::AssetId& assetId, const PlaybackContext& context,
 PoseBuffer& targetBuffer) override;
      void UpdateGraph(const AZ::Data::AssetId& assetId, const PlaybackContext& context,
 float& expectedDuration, float& normalizedTime) override;

    ....

      // AZ::Component
      static void Reflect(AZ::ReflectContext* reflect);
      static void GetProvidedServices(AZ::ComponentDescriptor::DependencyArrayType&
 provided);
      static void GetRequiredServices(AZ::ComponentDescriptor::DependencyArrayType&
 required);

private:
      // AZ::Component
      void Activate() override;
      void Deactivate() override;
```

Defining a System Component

To designate a component as a system component, you must set the `AppearsInAddComponentMenu` field to `System` when you reflect to the `EditContext`, as in the following example.

```
if (AZ::EditContext* editContext = serializeContext->GetEditContext())
{
      editContext->Class<AnimationGraphSystemComponent>(
            "Animation Graph", "Manages animation graph system component and features.")
            ->ClassElement(AZ::Edit::ClassElements::EditorData, "")
                  ->Attribute(AZ::Edit::Attributes::Category, "Engine")
                  ->Attribute(AZ::Edit::Attributes::AppearsInAddComponentMenu,
 AZ_CRC("System"))
            ;
}
```

This code exposes the component to the editor and other parts of the user interface that are visible to users. The optional `Category` field determines which submenu contains the component in the **Add Component** menu in Lumberyard Editor Entity Inspector. In the following example, the **Area Light** component appears in the submenu **Add Component**, **Rendering**, **Lights**.

```
...
    editContext->Class<EditorAreaLightComponent>(
        "Area Light", "Attach area lighting to an entity.")
        ->ClassElement(AZ::Edit::ClassElements::EditorData, "")
```

```
            ->Attribute(AZ::Edit::Attributes::Category, "Rendering/Lights")
...
```

Components and EBuses

EBuses are not required for components, nor are they directly bound to components in any way. But because they form the backbone of communication among all Lumberyard components, EBuses offer many benefits. We highly recommend that you learn how to use them in your game, systems, and components. For more information, see Event Bus (EBus) (p. 543).

Most components provide two EBuses to facilitate communication: a request bus and a notification bus. Both these EBuses use the `EBusAddressPolicy::ById` address policy and the ID of the entity for identification.

Request Bus

A component's request bus allows other components or external systems to make requests of the component. Usually the run-time version of the component implements the request bus. However, the editor component can service the bus in special cases.

The following sections examine the individual parts of an example request bus.

Transform Request Event Group

The following example defines a group of events that the `TransformComponent` handles.

```
class TransformComponentRequests
      : public AZ::ComponentBus // EBus traits for component buses: identification is based
 on an entity ID.
{
      public:

      // EBusTraits overrides - Only a single handler is allowed for a given entity ID.
      // Only one component on a entity can implement the events.
      static const EBusHandlerPolicy HandlerPolicy = EBusHandlerPolicy::Single;

      // Returns the local transform (parent transform excluded).
      virtual const Transform& GetLocalTM() = 0;

      // Sets the local transform and notifies all interested parties.
      virtual void SetLocalTM(const Transform& /*tm*/) {}

      // Returns the world transform (including parent transform).
      virtual const Transform& GetWorldTM() = 0;

      // Sets the world transform and notifies all interested parties.
      virtual void SetWorldTM(const Transform& /*tm*/) {}

      // Returns both local and world transforms.
      virtual void GetLocalAndWorld(Transform& /*localTM*/, Transform& /*worldTM*/) {}

...
 };
```

Base Class and Trait Specification

The base class for most `AZ::Component` request buses is `AZ::ComponentBus`. This class is a convenience to help set up EBus traits typical of component EBuses. You could also set up EBus traits by inheriting the

default `AZ::EbusTraits`. Then you could optionally override any or all of the following traits. For more information, see EBus Configuration Options.

- Address policy
- Bus ID type
- Connection policy
- Handler policy
- Lock type
- Priority sorting

These two approaches are shown in the following examples.

```
// Example using AZ::ComponentBus.
class TransformComponentRequests
      : public AZ::ComponentBus
{...}
```

```
// Example using AZ::EBusTraits
class TransformComponentRequests
      : public AZ::EBusTraits
{
...
      // EBusTraits overrides.
      static const EBusAddressPolicy AddressPolicy = EBusAddressPolicy::ById; // OR YOUR
 CHOSEN POLICY
      static const AZ::EBusHandlerPolicy HandlerPolicy =
 AZ::EBusHandlerPolicy::Multiple; // OR YOUR CHOSEN POLICY
      using BusIdType = EntityId;
...
}
```

EBus Request Bus Events

EBus event definitions are the main part of the bus specification. This interface defines what your component does. In the following example, the `TransformComponent` allows the retrieval and modification of the local and world transforms. It also creates interfaces for setting parent–child relationships.

```
...

// Returns the local transform (parent transform excluded).
virtual const Transform& GetLocalTM() = 0;

// Sets the local transform and notifies all interested parties.
virtual void SetLocalTM(const Transform& /*tm*/) {}

// Returns the world transform (including parent transform).
virtual const Transform& GetWorldTM() = 0;

// Sets the world transform and notifies all interested parties.
virtual void SetWorldTM(const Transform& /*tm*/) {}

// Returns both local and world transforms.
virtual void GetLocalAndWorld(Transform& /*localTM*/, Transform& /*worldTM*/) {}

...
```

EBus Request Bus Definition

After the event group has been declared, the EBus must be defined. Although you can use `AZ::EBus<TransformComponentRequests>` to define an EBus, we recommend that you use a `typedef` instead, as in the following example. This improves readability at bus call sites.

```
typedef AZ::EBus<TransformComponentRequests> TransformComponentRequestBus;
```

Another best practice is to use descriptive names in EBuses and avoid overloaded functions. Explicit and descriptive function names prevent future API name collisions as classes inherit (potentially many of) your EBus interfaces. Avoiding overloaded functions improves the experience of using your EBuses from scripting environments. In Lua and in visual scripting, the extra expressiveness improves readability and clarity.

Notification Bus

A component uses its notification bus to inform other components and the rest of the engine about relevant changes. To do this, it sends notifications in the form of EBus events to any class that monitors the bus. To monitor the bus, classes implement the notification bus handler interface (in the case of `TransformComponent`, this is `AZ::TransformNotificationBus::Handler`.)

> **Note**
> A request bus sends messages **to** a component; a notification bus sends messages **from** a component.

Transform Notification Event Group

The following example defines a group of notification events that the `TransformComponent` sends.

```
class TransformNotifications
      : public AZ::ComponentBus
{
    public:
    ...
      // Called when the local transform of the entity has changed. Local transform update
 always implies world transform change too.
      virtual void OnTransformChanged(const Transform& /*local*/, const Transform& /
*world*/) {}
    ...
};

typedef AZ::EBus<TransformNotifications>    TransformNotificationBus;
```

The notification bus can also change its `EBusTrait` specification if required.

Components as EBus Handlers

After you have created the EBus event groups and defined the EBuses, your component can implement the EBus interface by deriving the EBus handler. The following example is from the `TransformComponent`.

```
class TransformComponent
      : public AZ::Component
    , private AZ::TransformComponentRequestBus::Handler
{
      ...

      // TransformBus.
```

```
      /// Returns true if the tm was set to the local transform.
      const AZ::Transform& GetLocalTM() override { return m_localTM; }

      /// Sets the local transform and notifies all interested parties.
      void SetLocalTM(const AZ::Transform& tm) override;

      /// Returns true if the transform was set to the world transform.
      const AZ::Transform& GetWorldTM() override { return m_worldTM; }

      /// Sets the world transform and notifies all interested parties.
      void SetWorldTM(const AZ::Transform& tm) override;

      /// Returns both local and world transforms.
      void GetLocalAndWorld(AZ::Transform& localTM, AZ::Transform& worldTM) override
 { localTM = m_localTM; worldTM = m_worldTM; }

      ...
}
```

At this point you can implement the defined methods in the `TransformComponent`. After the `TransformComponent` connects to the EBus for its entity ID, its event handlers are invoked whenever an event is sent on that bus or ID.

Tick Bus and Components

The tick bus is the primary mechanism by which components subscribe to events that occur per CPU tick. If your component requires tick-based functionality, it implements the tick bus interface's `OnTick` method and connects to the tick bus for the required period of time. To avoid poor scalability of polling-based update structures, components should limit the time that they are connected to the tick bus.

Your components should connect to the tick bus only in the `Activate` method and disconnect only in the `Deactivate` method. If possible, components should connect only when necessary and disconnect as soon as feasible. For example, a component might use the `OnTick` method to check the state continuously but actively do processing for only a fraction of the connected time. Instead, the component should connect to the tick bus only when the state changes and disconnect after processing is complete. This approach is closer to that of event-based programming, in which polling is conducted only for short periods of time or not at all.

As an example, imagine a component that monitors the state of an entity after the entity enters a trigger. The component should avoid subscribing to the tick bus until the entity has entered the trigger. It should then disconnect from the tick bus as soon as the entity has left the trigger.

In the following example, the `NavigationComponent` implements the `OnTick` method.

```
class NavigationComponent
      : public AZ::Component
    , public NavigationComponentRequestBus::Handler
    , public AZ::TickBus::Handler
{
      ...

    // TickBus
    virtual void OnTick(float deltaTime, AZ::ScriptTimePoint time);

      ...
}
```

To connect and disconnect from the tick bus, the component uses code like the following.

```
AZ::TickBus::Handler::BusConnect();
```

```
AZ::TickBus::Handler::BusDisconnect();
```

Event-Based Programming and Event-Based Polling: Best Practices

It is important to know when to use the tick bus and when to use event-driven programming patterns instead.

Event-Based Polling

It is often convenient to tick a component every frame and monitor the state of other entities. For example, a `LookAt` camera component is commonly implemented to tick each frame, retrieve the transform of the target entity, and update its own transform accordingly.

Event-Based Programming

In Lumberyard, a more event-driven approach is to use the `TransformBus` to monitor the target entity for transform changes in a purely event-driven fashion. If the target entity doesn't move, no work is done and no polling is required. When the target entity moves, the `LookAt` component adjusts its own entity's transform accordingly.

Use Notifications to Make Your Components Easy to Use

When authoring a component, try to anticipate the requirements of components that might depend on yours. Use a notification bus to expose the appropriate notifications for your component. This approach enables others to write code that consumes the services of your components in a faster and more scalable way.

For more best practices, see Components and EBuses: Best Practices (p. 451).

Exposing Custom Components to Track View for Animation

To include custom components in cinematic cut scenes and movies rendered to disk, you must expose animatable component properties to Lumberyard's **Track View** editor and **Entity Inspector**. To expose a custom component and its properties, you must perform three steps:

1. Create getter and setter methods for the animated property on one of the component's request event buses.
2. Implement the getter and setter request handlers in your component.
3. Reflect your component to the edit context and the behavior context. Edit context reflection exposes your component in **Entity Inspector**, and behavior context reflection exposes it in the **Track View** editor.

Exposing a Custom Component: Example

The following example assumes that a custom component called `ImaginaryTargetComponent` has been created. The component has a `Vector3` property called `ImaginaryPosition` that you want to animate in **Track View**. A request bus called `ImaginaryTargetComponentBus` has also been created for the component. This example assumes that you are familiar with programming event buses and

component handlers for them. For more information, see Event Bus (EBus) (p. 543) and Creating a Component (p. 429).

To expose a custom component to Track View

1. **Create getter and setter methods**

   Each property must provide a method to set its value and get its current value. To implement this, create setter and getter methods on one of the component's request buses. Then reflect those methods to the behavior context as part of the class reflection for the component.

   The following example creates setter and getter requests on the `ImaginaryTargetComponentRequestBus`.

```
/*!
* ImaginaryTargetComponentRequests EBus Interface
* Messages serviced by ImaginaryTargetComponents.
*/
class ImaginaryTargetComponentRequests
    : public AZ::ComponentBus
{
public:

    // EBusTraits overrides - Application is a singleton.
    // Only one component on an entity can implement the events.
    static const AZ::EBusHandlerPolicy HandlerPolicy = AZ::EBusHandlerPolicy::Single;

    // Getter/Setter methods for ImaginaryTargetPosition.
    virtual AZ::Vector3 GetImaginaryTargetPosition() = 0;
    virtual void SetImaginaryTargetPosition(const AZ::Vector3& newPosition) = 0;
};
using ImaginaryTargetComponentRequestBus = AZ::EBus<ImaginaryTargetComponentRequests>;
```

2. **Implement handlers in your component**

   Implement handlers in your component for the setter and getter requests that you declared in the first step, as in the following example.

```
class ImaginaryTargetComponent
    : public AzToolsFramework::Components::EditorComponentBase
    , public LmbrCentral::ImaginaryTargetComponentRequestBus::Handler
{

public:
    AZ_EDITOR_COMPONENT(ImaginaryTargetComponent, "{4491D282-C120-4B2E-BC63-
AC86296956A2}");

    ImaginaryTargetComponent() : m_imaginaryPosition(.0f) {};

    // ImaginaryTargetComponentRequestBus::Handler implementation.

    // Implementations for Getter/Setter methods for ImaginaryTargetPosition.
    // Presumably these would be used for something useful; this example just
    // stores and returns the value.
    AZ::Vector3 GetImaginaryTargetPosition() override { return m_imaginaryPosition; }
    void SetImaginaryTargetPosition(const AZ::Vector3& newPosition) override
 { m_imaginaryPosition = newPosition; }

protected:
    // Required Reflect function.
    static void Reflect(AZ::ReflectContext* context);

private:
```

```
    AZ::Vector3 m_imaginaryPosition;
};
```

3. **Reflect your component**

   Using the edit context and behavior contexts, reflect the component's class, request event bus, and setter and getter methods. **Track View** uses the setter and getter methods that you reflect in this step to set and get values for your animated property. You must also reflect a `VirtualProperty` declaration that tells **Track View** that your component is capable of being animated.

```
/*static*/ void ImaginaryTargetComponent::Reflect(AZ::ReflectContext* context)
{
    AZ::SerializeContext* serializeContext =
 azrtti_cast<AZ::SerializeContext*>(context);

    if (serializeContext)
    {
        serializeContext->Class<ImaginaryTargetComponent,
 AzToolsFramework::Components::EditorComponentBase>()
            ->Version(0)
            ->Field("ImaginaryPosition",
 &ImaginaryTargetComponent::m_imaginaryPosition);

        AZ::EditContext* editContext = serializeContext->GetEditContext();
        if (editContext)
        {
            editContext->Class<ImaginaryTargetComponent>("ImaginaryTarget", "A Code
 Sample enabling Track View Animation")
                ->ClassElement(AZ::Edit::ClassElements::EditorData, "")
                    ->Attribute(AZ::Edit::Attributes::Category, "Game")
                    ->Attribute(AZ::Edit::Attributes::AppearsInAddComponentMenu,
 AZ_CRC("Game", 0x232b318c))
                ->DataElement(0, &ImaginaryTargetComponent::m_imaginaryPosition,
 "Imaginary Target Pos", "Imaginary Target Position")
                ;
        }
    }

    AZ::BehaviorContext* behaviorContext = azrtti_cast<AZ::BehaviorContext*>(context);
    if (behaviorContext)
    {
        // Reflect the setter and getter methods and create a virtual property that
 refers to them.
        behaviorContext-
>EBus<ImaginaryTargetComponentRequestBus>("ImaginaryTargetRequestBus")
            ->Event("GetImaginaryTargetPosition",
 &ImaginaryTargetComponentRequestBus::Events::GetImaginaryTargetPosition)
            ->Event("SetImaginaryTargetPosition",
 &ImaginaryTargetComponentRequestBus::Events::SetImaginaryTargetPosition)
            ->VirtualProperty("ImaginaryPosition", "GetImaginaryTargetPosition",
 "SetImaginaryTargetPosition");

        // Attach the "ImaginaryTargetRequestBus" EBus that you reflected to the
 behavior context of the ImaginaryTargetComponent class.
        behaviorContext->Class<ImaginaryTargetComponent>()-
>RequestBus("ImaginaryTargetRequestBus");
    }
}
```

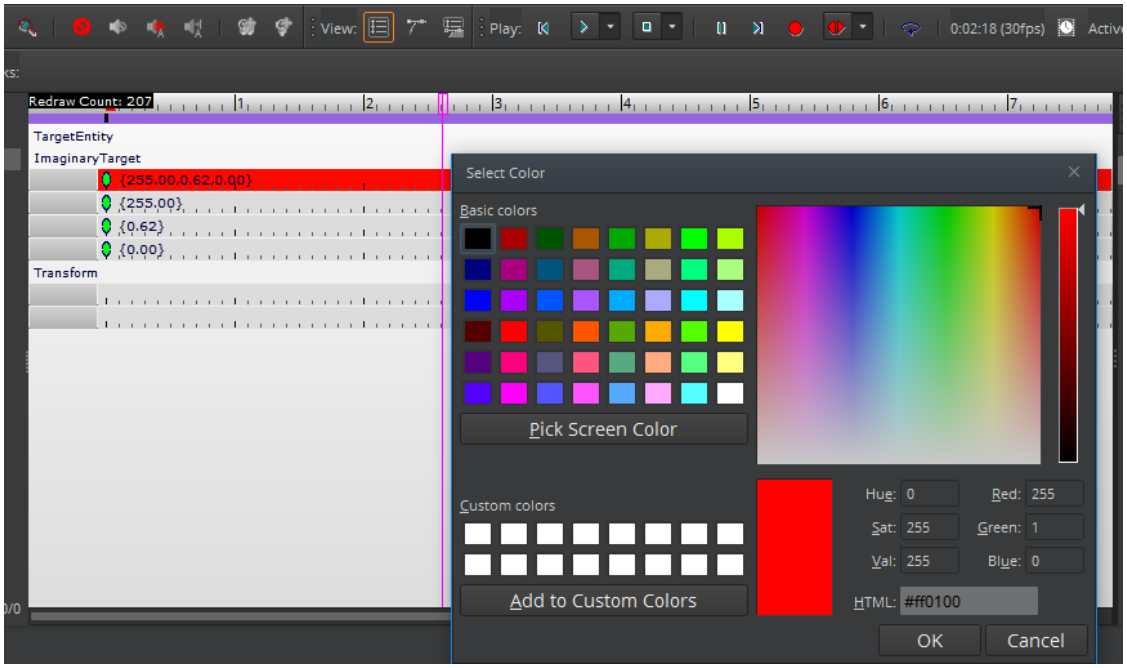
4. **(Optional) Place Unit Attributes on Getters**

   The **Track View** user interface depends on the data type that the getter and setter use. The foregoing example uses a type of `AZ::Vector3`, so **Track View** creates a compound `x,y,z` track from the property. By contrast, if the getter and setters use a `bool`, **Track View** creates a Boolean track.

For the majority of animatable properties, the type is sufficient. However, in some cases you might have to set units for a reflected property. For example, if your property's `AZ::Vector3` represents a color, you must add an attribute to the reflection of the getter event. The attribute instructs **Track View** to use a color picker for that property. If you have a property called `ImaginaryTargetColor` that calls a getter event called `GetImaginaryTargetColor`, use reflection code like the following:

```
->Event("GetImaginaryTargetColor",
 &ImaginaryTargetComponentRequestBus::Events::GetImaginaryTargetColor)
    ->Attribute("Units", AZ::Edit::Attributes:: PropertyUnits8BitColor)
```

Track View then uses a color track for the property, as the following image shows.



Other units can be found in the file `dev\Code\Framework\AZCore\AZCore\Serialization\EditContextConstants.inl`. As of Lumberyard release 1.8, these units are the following.

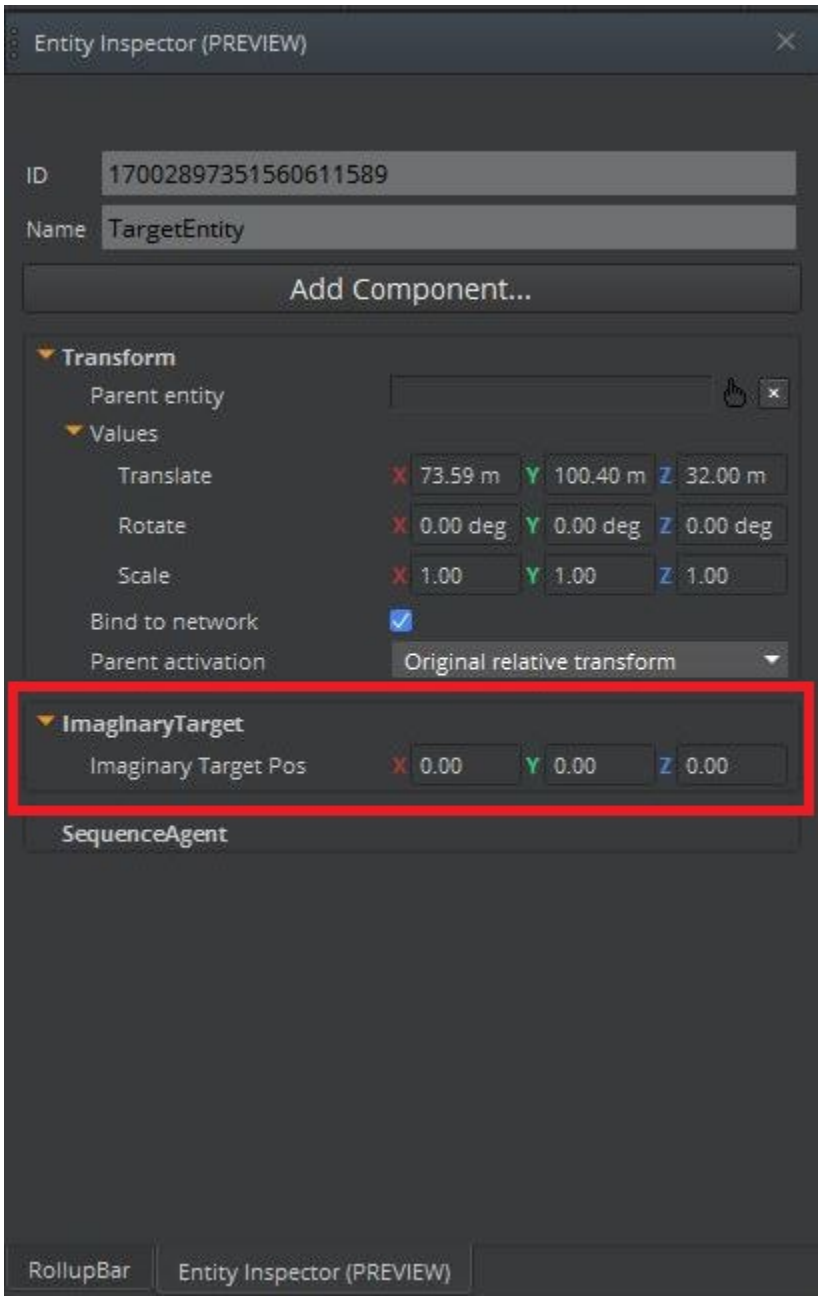
```
const static AZ::Crc32 PropertyUnitsRadian = AZ_CRC("Radians");
const static AZ::Crc32 PropertyUnits8BitColor = AZ_CRC("8BitColor");
```

If you have an angular parameter in radians that you want to **Track View** to convert to degrees in its user interface, use `AZ::Crc32 PropertyUnitsRadian`.

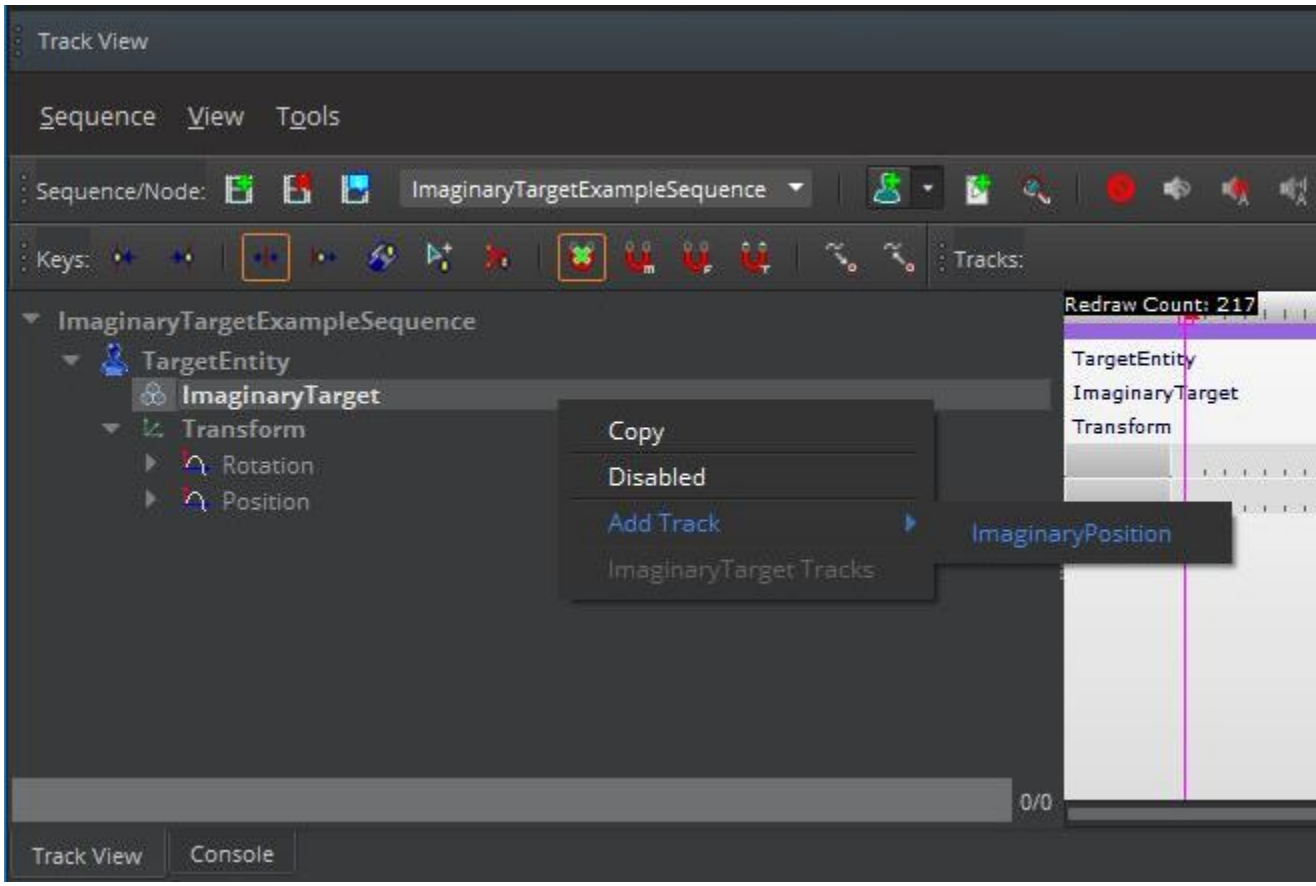
Viewing the Result

Now you can view how the example component and property appear in the **Entity Inspector** and the **Track View** editor.

In the following **Entity Inspector** image, `EditContext` reflection has exposed the **ImaginaryTarget** component and its **Imaginary Target Pos** property.



In the following **Track View** image, `BehaviorContext` reflection has exposed the **ImaginaryTarget** component and the **ImaginaryPosition** track from the corresponding virtual property.



Components and EBuses: Best Practices

Follow these best practices for authoring components.

EBus Names

The following EBus naming conventions remove ambiguity and provide consistency.

- Use the name format `MyComponentRequestBus` for the bus that others use to invoke functions on `MyComponent`, as in the following example.

```
class CheeseburgerComponentRequests : public AZ::ComponentBus
{
      bool ICanHasCheeseburger() const = 0;
};
using CheeseburgerComponentRequestBus = AZ::EBus<CheeseburgerComponentRequests>;
```

- Use the name format `MyComponentNotificationBus` for events that are broadcast from `MyComponent`, as in the following example.

```
class CheeseburgerComponentNotifications : public AZ::ComponentBus
{
      void OnCheeseburgerEaten(AZ::u8 yelpRating) {};
};
using CheeseburgerComponentNotificationBus =
 AZ::EBus<CheeseburgerComponentNotifications>;
```

Provide Default Implementations of Methods

Notification buses typically provide default implementations of the methods within the interface. Many other components can monitor your component's events, but not all of them are interested in every event that your component sends. If you provide default implementations for all your methods, other components that subscribe to your events can implement only those events that are relevant to them.

EBus Event Naming

Good EBus event names are verbose. Classes can monitor multiple buses, so descriptive event names makes it clear which bus the function corresponds to. This practice also prevents potential name collisions among event interfaces from different buses.

The following example is a clearly named `PhysicsComponentNotificationBus` event.

```
virtual void OnPhysicsEnabled() = 0;
```

The following example is an ambiguously named `PhysicsComponentNotificationBus` event.

```
virtual void OnEnabled() = 0;
```

Avoid Using Type Definitions for Serialized Data

An instructive example from Lumberyard shows the importance of using classes instead of type definitions for serialized data. Formerly, `EntityId` used the type definition `uint32_t`. When the decision was made to change this to 64-bit, upgrade functions had to be written for every class that contained an `EntityId`. If `EntityId` had been a class, a single upgrade function could have been written for the class, and no further work would have been required. Obviously, this principle does not apply to primitive types like `bool`, `float`, `int`, and `string`. However, if you have a specific type that is serialized and might change in the future, implement it as a reflected class. This provides a single context where you can easily make the conversion for the class or type.

EBus Results

Always initialize a variable before calling an EBus event that overwrites the variable. Even if you are sure that a particular class or component is listening on the bus, it's worth handling the exceptional case. This is especially true in distributed environments in which entities can come and go as part of area-of-interest or other dynamic patterns.

The following example initializes a result variable before calling an EBus event that produces a result.

```
AZ::Transform targetEntityTransform = AZ::Transform::Identity(); // initialize result
 variable...
EBUS_EVENT_ID_RESULT(targetEntityTransform, targetEntityId, AZ::TransformBus,
 GetWorldTM); // ...in case of no response
```

EBus Timing

Following are some best practices for the timing of EBus actions.

- In the Activate() (p. 430) function, make sure that connecting to buses is the last step.
- In the Deactivate() (p. 430) function, make sure that disconnecting from buses is the first step.
- In a multithreaded environment, it's possible to receive bus events from the moment that you connect to the bus until the moment you disconnect. For this reason, make sure of the following:

- Your component is fully activated before it starts reacting to events.
- Your component stops receiving events before it starts deactivation.

This practice prevents your component from being in a half-activated state when it starts reacting to events, or in a half-deactivated while still receiving events.

- When you send events on a notification-style bus, the last step in a function should ensure that the data is fully populated.

The following is an example to avoid.

```
EBUS_EVENT_ID(GetEntityId(), OnTransformChanged, newTransform);
m_transform = newTransform;
```

If a component is monitoring the `OnTransformChanged` event and sets your transform in response to the event, the component's action will be undone by the `m_transform = newTransform;` assignment.

Making Functions Public or Protected

Consider the following when deciding to make functions public or private.

- Make your bus functions `public` if they constitute the public interface for your class. While it's discouraged, Lumberyard does not prevent users from getting direct pointers to components and calling functions directly. To avoid this, make sure that your useful functions are public. For example, `MyComponent` should probably implement functions from `MyComponentRequestBus` publicly.
- Make your bus functions `protected` if they contain the private workings of your class. For example, your component's reaction to the `TransformNotificationBus::OnTransformChanged` event would likely be a private implementation detail.

Avoid Using the const Type Qualifier in EBus Event Group Functions

In general, all EBus methods are meant to be implemented by external code. As the author of an EBus interface, you do not want to force the use of `const` declarations upon any particular implementation of a method. Because end developers will write these for their own data, it is best to avoid `const` in EBus event group functions.

Note
This is true for request buses but is particularly important with regard to notification buses.

Behavior Context

In Lumberyard 1.8, the *behavior context* replaces *script context*. The behavior context works with *serialize context*, *edit context*, and *network context* to provide rich C++ reflection. The behavior context focuses on the runtime aspects of C++ code and allows you to manipulate C++ code and objects while they are being created. All script bindings, including Lua, use this reflection. Reflection is also used for modification of objects while in running state (such as animating object properties) and reading of current properties for component state transitions. You can have multiple behavior contexts that are specialized for different purposes, and you can unreflect the behavior contexts in order to implement reloading. At a high level, the behavior context uses only a few primitives on which to build: properties, methods, classes, EBuses and attributes.

With the introduction of the behavior context, you can no longer reflect directly into the script context. For more information, see the migration notes for Lumberyard 1.8.

Reflection API

This section describes how methods, properties, classes, and other primitives are used in the Lumberyard reflection API.

Method

Methods reflect a C++ function. You can have global or class methods. Each method must have a unique name for its scope (global or class). You can also provide default values; this allows you to call reflected methods with fewer arguments. Default values are used right to left. You can also provide a debug description for your method. It's highly recommended that you do so. The debug description information is provided to users for context when they use the reflection, as is done with `ClassView` in the Lua editor. The system automatically generates a description that includes result and argument types. However, adding intent and additional details to your descriptions greatly improves the usability of your reflection. This is especially important when a method has many attributes.

Property

Properties access data and can be global properties or class properties. Each property must have a unique name for its scope. As is customary, a property has getter and setter methods. If you don't provide a setter method for a property, the property is read only. If you don't provide a getter method, the property is write only.

Lumberyard does support global functions, member functions, and lambda functions as property getters and setters.

Lumberyard provides macros that you can use to wrap a class value. You can use `BehaviorValueProperty(&value)` to implement getter and setter methods, or you can implement them individually by using `BehaviorValueGetter` and `BehaviorValueSetter`. These macros implement lambda functions for those values. When the state of your object is modified, you might have to perform operations other than simply setting the value. For this reason, it is a best practice to always implement your getters and setters. You can always change your implementation later.

Constant

Constants are implemented as read-only properties and can be global or restricted to a class. A behavior context macro called `BehaviorConstant` implements the lambda getter for you.

Enum

Because class enums often require casting, Lumberyard currently treats all enums values as `int`. Enums are implemented as read-only `int` properties.

Class

Reflects a C++ class or struct. You can provide an optional name. If you do not provide a name, the class name from `AzTypeInfo` is used. That name must be unique for the scope. Because the system uses `AzRTTI` to build the class hierarchy, you can use RTTI if you want to reflect base class functionality.

- **Allocator** - You can provide a custom allocator/ deallocator for your class. This allows you to override any existing allocation schema. If you do not provide a custom allocator, aznew/delete is used (`AZ_CLASS_ALLOCATOR`).
- **Constructor** - Allows you to enumerate the class constructors that you want to reflect. You must pass all constructor arguments as template augments.

- **Wrapping/WrappingMember** - Allows code to inform the system that it is a wrapper of another class. This is useful when you reflect smart pointers and string wrappers.
- **Userdata** - Allows you to provide a pointer to user data. The pointer is accessible from all callbacks (like a custom allocator) that you implement for the class.
- **Method** - Reflects a C++ class function. The first argument is the class pointer. This is the same usage as global methods.
- **Property** - Reflects class data. The first argument is the class pointer. This is the same usage as global properties.
- **Enum** - Enums are int read-only properties.
- **Constant** - Constants are read-only properties.

Nested Classes

To bind a nested class to the behavior context, you must do it from within a function on the nested class. C++ rules about nested class member access from outside scopes make this requirement necessary.

The following counterexample shows a poorly formed nested class. The code does not work.

```
//Bad nested class
class Outer
{
public:
    AZ_TYPE_INFO(Outer, "...");
    static void Reflect(AZ::ReflectContext* context);
    class Inner
    {
    public:
        AZ_TYPE_INFO(Inner, "...");

        int m_member = 0;
    };
};

void Outer::Reflect(AZ::ReflectContext* context)
{
    if (AZ::BehaviorContext* behavior = azrtti_cast<AZ::BehaviorContext*>(context))
    {
        behavior->Class<Inner>("OuterInner")
            ->Property("member", BehaviorValueProperty(&Inner::m_member))
            ;
    }
}
```

The following code shows a well formed nested class.

```
//Good nested class
class Outer
{
public:
    AZ_TYPE_INFO(Outer, "...");
    static void Reflect(AZ::ReflectContext* context);

    class Inner
    {
    public:
        AZ_TYPE_INFO(Inner, "...");
        static void Reflect(AZ::ReflectContext* context);

        int m_member = 0;
```

```
    };
};

void Outer::Reflect(AZ::ReflectContext* context)
{
    Inner::Reflect(context);
}

void Outer::Inner::Reflect(AZ::ReflectContext* context)
{
    if (AZ::BehaviorContext* behavior = azrtti_cast<AZ::BehaviorContext*>(context))
    {
        behavior->Class<Inner>("OuterInner")
            ->Property("member", BehaviorValueProperty(&Inner::m_member))
            ;
    }
}
```

EBus

`EBus` Reflects Lumberyard event bus messages. Depending on your EBus configuration, `Broadcast`, `Event` (with ID) and `Queuing` are reflected. Queuing is a generic function to be executed when the bus messages are consumed.

- **Event** - Reflects an EBus event. Depending on your EBus configuration, Lumberyard automatically reflects `Broadcast`, `Event`, `QueueBroadCast`, and `QueueEvent`.
- **Handler** - Reflects a class that you must implement to forward messages from the EBus to behavior context methods. You must create a class that can monitor the specified EBus and forward messages to the behavior context. This is a requirement because the behavior context can not guarantee that there is a handler for each message. If a message expects a result, you must provide a default result in case the message is not handled by the behavior context user. Keep in mind that the system creates as many of these handlers as the behavior context requires. Handlers can also execute in different threads. As a result, you should avoid static storage for values that change. The best way to understand this is to examine the example that follows.

Example

The following code example shows the use of the Lumberyard reflection API.

```
// Global Property
behaviorContext.Property("globalProperty",&GlobalPropertyGetter, &GlobalPropertySetter); //
 Property with getter and setter function
behaviorContext.Property("globalProperty",[]() { return g_globalValue; }, [](int value)
 { g_globalValue = value;}); // Property with lambdas functions.
behaviorContext.Property("globalProperty", BehaviorValueProperty(&g_globalValue)); //
 Property from a value with helper macro. The macro implements the code above.
behaviorContext.Property("globalReadOnlyProperty", &GlobalPropertyGetter, nullptr); // Read
 only property with a getter function.
behaviorContext.Property("globalReadOnlyProperty", []() { return g_globalValue; },
 nullptr); // Read only property with lambda function.
behaviorContext.Property("globalReadOnlyProperty", BehaviorValueGetter(&g_globalValue),
 nullptr); // Read only property with a value and a helper macro.
// Write only is the same as ReadOnly, but with the setter enabled and the getter set to
 nullptr. These properties are rare.

// Global Methods
behaviorContext.Method("GlobalMethod",&GlobalMethod);

// Global Constants and Enums (implemented using properties). The functions are provided
 for clarity.
```

```
behaviorContext.Constant("PI", []() { return 3.14f; });
behaviorContext.Constant("PI", BehaviorConstant(3.14f));
behaviorContext.Enum<EnumIntValue>("EnumIntValue");

// Class - When you declare a class, if you want to reflect base class functionality, just
 use RTTI.
behaviorContext.Class<MyClass>() // The name of the class comes from AzTypeInfo. In this
 case the name is "MyClass".
                                 // AzType information is a requirement for all classes
 used with reflection in
                                 // Lumberyard (including serialization and networking)
    ->Constructor<int>() // Optional additional constructors. You can have as many as
 needed.
    ->Constant("epsilon",BehaviorConstant(0.001f)) // Class constant. All features from the
 global versions apply.
    ->Enum<MyClass::ENUM_VALUE>("ENUM_VALUE") // Class enum. All features from the global
 versions apply.
    ->Method("Method",&MyClass::Method)    // Class method. All features from the global
 versions apply.
    ->Property("data", &MyClass::GetData(), &MyClass::SetData) // Class features. All
 features from the global versions apply.
    ;

// EBus
class MyEBusBehaviorHandler : public MyEBus::Handler, public AZ::BehaviorEBusHandler
{
public:
    AZ_EBUS_BEHAVIOR_BINDER(MyEBusBehaviorHandler ,"{19F5C8C8-4260-46B1-
B624-997CD3F10CBD}", AZ::SystemAlloctor, // Name, TypeId and default allocator.
                                OnEvent); // List of event names to handle and support for
 BehaviorContext.

    void OnEvent(int a) override // This is an event listener like other EBus listeners.
    {c
        Call(FN_OnEvent,a); // Forward the event to a behavior listener if there is one.
 FN_***EventName*** events are declared by the AZ_EBUS_BEHAVIOR_BINDER macro.
    }
};

behaviorClass.EBus<MyEBus>("MyEBus") // EBuses are not required to have TypeInfo, so you
 must always provide a name.
    ->Handler<MyEBusBehaviorHandler >() // Allow systems that use behavior context to
 create handlers for this EBus every time
                                          // they must listen for events. If you reflect a
 bus without a handler, behavior context users can only send events.
    ->Event("OnEvent",&amp;amp;MyEBus::Events::OnEvent) // Allow behavior context system to
 send an "OnEvent" event. The code automatically generates
                                                 // Broadcast, Event, QueueBroadcast,
 QueueEvent, and QueueFunctions if the EBus configuration
                                                 // supports them. You don't have to
 provide events; you can provide only a handler if
                                                 // you don't have behavior context
 systems to send events.
    ;

// Properties, methods, classes and ebuses can have attributes. An attribute is a
 combination of a Crc32 ID and a value. The value
// can be a constant, a variable address, a global function, a class member function, or a
 class member variable address.
behaviorClass.Method("GlobalMethod",&GlobalMethod)
    ->Attribute("ValueAttr",10)        // Value attribute.
    ->Attribute("MethodAttr", &SomeOtherGlobalMethod)
    ;
// You add the same attributes to a property...
behaviorClass.Property("GlobalProperty", BehaviorValueProperty(&g_globalValue)
    ->Attribute("MyAttr",20)
```

```
    ;

// or to a class or class method or property.
behaviorClass.Class<MyClass>()
    ->Attribute("ClassAttr",100)
    ->Attribute("CalssAttr1",&MyClass::SomeMethod)
    ->Method("Method",&MyClass::Method)
        ->Attribute("MethodAttr",100)
    ->Property("data", BehaviorValueProperty(&MyClass::m_data))
        ->Attribute("PropertyAttr",500)
    ;
```

Gameplay Bus

The gameplay bus alerts gameplay systems that an event has occurred. You can use the bus to send contextual messages between the visual scripting, scripting, and code parts of your game in a generic and extensible way.

The gameplay bus passes an `AZStd::any`, a class that uses type erasure to hold any C++ reflected type or any Lua primitive (string, number, Boolean, etc.). It includes mechanisms for type safety to ensure that it returns the same type that it is passed. In Lua, the type can be inspected if it has been exposed to the behavior context through the `typeid()` function, which uses the `AZ_RTTI` system.

Example

```
function MyComponent:OnEventBegin(param)
    if typeid(param) == typeid(Uuid) then
        -- param is a Uuid
    elseif typeid(param) == typeid(Vector3) then
        -- param is a Vector3
    end
end
```

GameplayNotificationId

The `GameplayNotificationId` is the type used as the ID for the gameplay bus. Its syntax is as follows.

```
GameplayNotificationId(const AZ::EntityId& entityChannel, AZ::Crc32 actionNameCrc)
```

The `entityChannel` and `actionNameCrc` parameters create a unique bus ID. You can use this ID to communicate with the bus.

entityChannel

When you write your events, choose the ID of an entity channel that makes sense in the context of your game. Components are automatically aware of the following two entity IDs, which you can use for channels. These IDs do not require an entity reference.

1. The component's own entity ID. To obtain it, call `GetEntityId()`.
2. The default `AZ::EntityId()`.

To communicate directly to a specific entity, use `GetEntityId()`. To communicate indirectly to a generic audience, use `AZ::EntityId()`.

actionNameCrc

You can pass the `actionNameCrc` parameter as a string or as the `AZ::Crc32` of that string when you construct the ID. This parameter should be the name of the event that gives context to the event data.

GameplayNotifications

The `GameplayNotifications` class contains the gameplay bus type traits. It establishes the `GameplayNotificationId` as the bus ID. It defines the following events.

```
void OnEventBegin(const AZStd::any& value)
```

```
void OnEventUpdate(const AZStd::any& value)
```

```
void OnEventEnd(cosnt AZStd::any& value)
```

GameplayEventHandlerNode

You can use **GameplayEventHandlerNode** to control the flow of gamplay events.

Input Ports

The following table describes the flow graph nodes for handling a gameplay event.

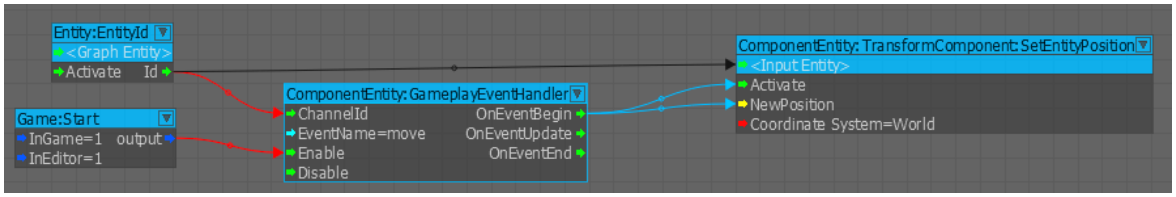
Input Port	Description
ChannelId	The channel on which to send your event. For information on choosing a channel, see `GameplayNotificationId`.
EventName	The name of the event that you want to handle. For information on available events, see `GameplayNotifications`.
Enable	Calls the `BusConnect` method and connects to `GameplayNotificationBus`. When the node connects, it begins handling events that are received when the flowgraph updates. **Note** If you send multiple events in a flowgraph update, only the last event is used.
Disable	Calls `BusDisconnect`, disconnects from `GameplayNotificationBus`, and stops the handling of events.

Output Ports

The output ports for the **GameplayEventHandlerNode** are the same as those for the `GameplayNotifications` class (`OnEventBegin`, `OnEventUpdate`, and `OnEventEnd`).

GameplayEventHandlerNode Flow Graph Example

The following image shows how the flow graph might be used be to change the position of (move) an entity.



GameplayEventSenderNode

You can use **GameplayEventSenderNode** to sent gameplay events.

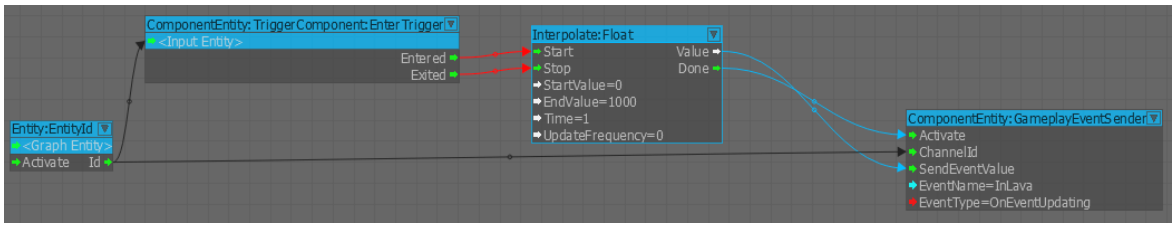
Input Ports

The following table describes the flow graph nodes for sending a gameplay event.

Input Port	Description
Activate	Causes a gameplay event to be sent.
ChannelId	The channel on which to send your event. For information on choosing a channel, see `GameplayNotificationId`.
SendEventValue	Stores the `AZStd::any` value for delivery with the event.
EventName	The name of the event that you want to handle. For information on choosing a channel, see `GameplayNotificationId`.
EventType	Specifies the event. The options are `OnEventBegin`, `OnEventUpdating`, and `OnEventEnd`. These events are defined in the `GameplayNotfications` class.

GameplayEventSenderNode Flow Graph Example

The following image shows how the flow graph might be used to send an event when an entity enters lava.



Script Examples

The Lua script examples in this section illustrate the use of `GameplayNotificationBus` to control what happens when an entity enters and then exits lava.

1. The following example implements the `OnActivate` function when the entity enters lava.

```
local InLavaBehavior ={
    Properties =
    {
    },
}
```

```
function InLavaBehavior:OnActivate()
    local gameplayId = GameplayNotificationId(self.entityId, "InLava")
    self.gameplayBus = GameplayNotificationBus.CreateHandler(self, inputBusId)
end
```

2. The following example implements the `OnEventBegin` function to start an animation associated with the event.

```
function InLavaBehavior:OnEventBegin(floatValue)
    local animInfo = AnimatedLayer("LavaHotFootDance", 0, true, 1.0, 0.0);
    SimpleAnimationComponentRequestBus.Event.StartAnimation(self.entityId, animInfo)

    -- tell our flow graph HUD to transition the screen effect to "In Lava"
    local gameplayId = GameplayNotificationId(EntityId(), "TransitionScreenEffect")
    GameplayNotificationBus.Event.OnEventBegin(gameplayId, "In Lava")
end
```

3. The following example implements the `OnEventUpdating` function to update a health component regarding the status of the entity.

```
function InLavaBehavior:OnEventUpdating(floatValue)
    -- alert the health component (this gameplay component is an example only) that we are
 taking damage, it can handle any death transitions
    HealthComponentBus.Event.TakeDamage(self.entityId, floatValue)
end
```

4. The following example implements the `OnEventEnd` function to end the animation and return status to normal.

```
function InLavaBehavior:OnEventEnd(floatValue)
    local animInfo = AnimatedLayer("Idle", 0, true, 1.0, 0.0);
    SimpleAnimationComponentRequestBus.Event.StartAnimation(self.entityId, animInfo)

    -- tell our flow graph HUD to transition the screen effect to "Normal"
    local gameplayId = GameplayNotificationId(EntityId(), "TransitionScreenEffect")
    GameplayNotificationBus.Event.OnEventBegin(gameplayId, "Normal")
end
```

5. The following example implements the `OnDeactivate` function to disconnect from the gameplay bus.

```
function InLavaBehavior:OnDeactivate()
    self.gameplayBus:Disconnect()
end

return InLavaBehavior
```

Slices and Dynamic Slices

> The Component Entity System is currently in preview and is undergoing active development. It will replace the legacy Entity System (p. 520).

A slice is a collection of configured entities (p. 428) that is stored as a single unit in a reusable asset. You can use slices to conveniently group entities and other slices for reuse. Slices are similar to prefabs but are part of the new Component Entity system. Slices can contain component entities, whereas prefabs cannot. Unlike prefabs, slices can be nested into a fully cascasding hierarchy. For example, a

level, a house, a car, and an entire world are all slices that depend on (cascade) from a number of other slices.

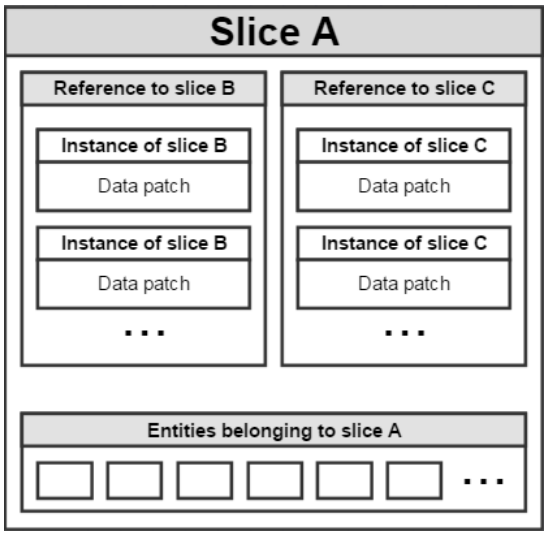
You can generate a slice asset that contains any number of entities that you have placed and configured. These entities can have arbitrary relationships. For example, they can exist in a parent/child transform hierarchy, although this is not required.

After you have created the slice asset, you can use the editor to instantiate the slice asset in your worlds, either by right-clicking in the viewport and choosing **Instantiate Slice**, or by dragging a slice asset into the viewport directly from the **File Browser**. Just as with standard prefab systems, you can then modify the entities in your slice instance. You can optionally push the changes back to the slice asset, which will affect all instances of that slice asset, as well as any other slices cascading from it.

A slice can contain instances of other slices. Modifications of a slice instance within another slice causes the changes to be stored in the instance as overrides (in the form of a data differential or delta). The modifications stored can be changes such as entity additions, entity removals, or component property changes.

Anatomy of a Slice

The following diagram illustrates an example slice A, which contains references to two other slices B and C. Slice A has two instances each of B and C:



Each instance contains a data patch, which may be empty if no changes or overrides are present. If the instantiation of slice B in slice A has been modified in comparison with the source asset B, the data patch contains the differences. When slice A is instantiated again, it contains instances of slice B, but with the modifications applied. Any nonoverridden fields propagate through the hierarchy. If you change a property value in the slice B asset on disk, the instance of B contained in slice A will reflect that change — if the property for that instance has not already been overridden, as reflected in the instance's data patch.

In addition to references to other slices, slices can contain zero or more entities. These entities are original to this slice and are not acquired through referenced slice instances. A slice does not have to contain references to other slices. A slice that contains only original entities (as represented by the bottom box in the diagram) and no references to other slices is called a *leaf slice*.

Working with Dynamic Slices

Slices are a powerful tool for organizing entity data in your worlds. In the editor, you can choose to cascade slices and organize entity data in any desired granularity and still receive the benefits of data

sharing and inheritance throughout the hierarchy. A level-based game, for example, implements each level as its own slice asset that contains instances of many other slices. These slices can potentially cascade many levels deep. You can even choose to create slices from other slices and inherit only the elements that you want.

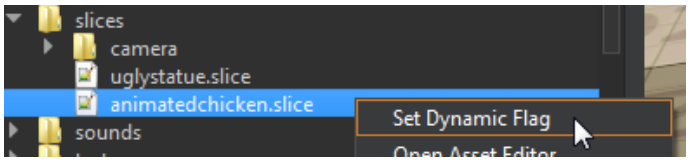
Standard slice assets (`.slice` files) rely on the editor and cannot be instantiated at run time. However, Lumberyard provides a mechanism for designating any `.slice` asset that you've built as a *dynamic slice*. When you designate a slice as a dynamic slice, the Asset Processor processes and optimizes the slice for you, producing a `.dynamicslice` file asset. A dynamic slice is simply the run-time version of its source slice, containing only run-time components; the editor-dependent components have been converted to their run-time counterparts. Furthermore, dynamic slices are flattened and no longer maintain a data hierarchy, as doing so would increase memory footprint and reduce instantiation performance.

In the level-based game example previously mentioned, you could designate your giant level slice as a dynamic slice. When your game loads the level, it does so by instantiating the resulting `.dynamicslice` file.

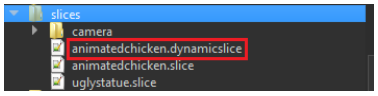
You can choose to generate dynamic slices at whatever granularity is appropriate for your game. Because slices are loaded entirely asynchronously, they are a good choice for streaming strategies. For example, a driving game might represent each city block as a separate slice and choose to load them predictively based on player driving behavior.

To generate a dynamic slice

Right-click any `.slice` asset in the **File Browser**, and click **Set Dynamic Flag**.



The Asset Processor processes the source `.slice` file and generates a `.dynamicslice` file. The new `.dynamicslice` file appears in the File Browser as its own asset:



To remove the dynamic slice

Right-click the source `.slice` file and choose **Unset Dynamic Flag**.

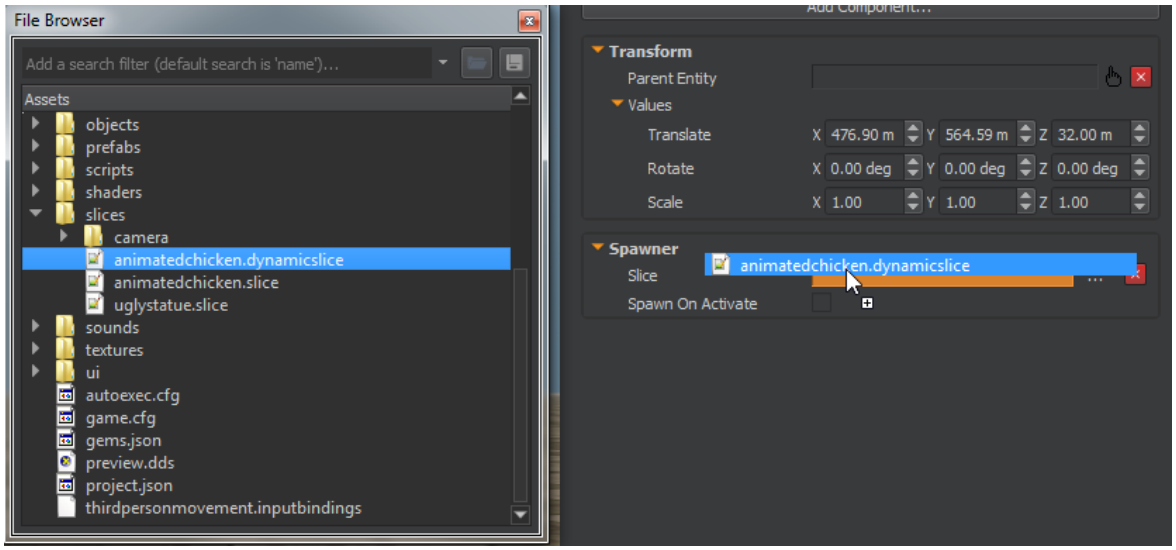
The Asset Processor deletes the `.dynamicslice` file from the asset cache for you.

Instantiating Dynamic Slices

You can instantiate dynamic slices from your own components. To do so, reflect (p. 431) a `DynamicSlice` asset reference. You can populate the reference in the editor in the usual way, such as dragging a `.dynamicslice` asset from the **File Browser** onto your component's reflected asset property. You can then use the following EBus call to instantiate the referenced dynamic slice at a desired location in the world.

```
// Asset reference member, which must be reflected.
AZ::Data::Asset<AZ::DynamicPrefabAsset> m_sliceAsset;

// Create an instance of the dynamic slice.
AZ::Transform location = ...;
EBUS_EVENT(AzFramework::GameEntityContextRequestBus, InstantiateDynamicSlice, m_sliceAsset,
 location);
```

Lumberyard includes a spawner component that is a good example of this behavior. You can use the spawner component directly or as an example from which to build your own.



You can see the source code for the spawner component at the file location `dev\Code\Engine\LmbrCentral\source\Scripting\SpawnerComponent.cpp` in the folder in which you installed Lumberyard.

For information on creating an `AZ::Module`, see Creating an AZ Module. For more information about working with slices, see Working with Slices.

Controller Devices and Game Input

Use the information in this section to learn about game input devices and channels in Amazon Lumberyard.

> **Note**
> For information on the **Input** component that replaces legacy action maps, see Input in the Amazon Lumberyard User Guide.

Topics

- Input in Amazon Lumberyard (p. 465)
- Legacy Input (p. 469)

Input in Amazon Lumberyard

The AZ framework input system is in preview release and is subject to change.

Input is what differentiates interactive experiences from all other entertainment media. Regardless of genre or platform, every game is driven by some form of player interaction with a physical input device. There are many different types of input devices, each of which can produce a wide range of input data. Furthermore, the way this data is delivered to an application is rarely consistent across devices or platforms. As a cross-platform game engine, Lumberyard's goal is to provide platform-independent access to input data from any supported device. The goal is to provide a common interface with these qualities:

- Platform and device agnostic
- Extensible
- Efficient

In addition, input data should have these attributes:

- Must be obtainable by subscribing to input device events
- Must be obtainable on demand by polling the state or value of an input device

- Can be a custom payload, but adds no overhead when not used

Lumberyard has created an input system that satisfies all these requirements.

A new input interface and `IInput` implementation act as a bridge between our new input system and the existing `IInput` interface. If you are dependent on the existing `IInput` interface and `CryInput` implementation, this bridge gives you the least disruptive migration path to the new interface. For more information, see Migration Path (p. 469).

In this preview version, AZ framework input has been disabled by default. To enable it, see Migration Path (p. 469).

AZ Framework Input

The new Lumberyard input interface is called AZ framework input. AZ framework already provides abstracted interfaces for platform-specific features like file I/O and application lifecycle management, so the AZ framework code location is a natural fit.

The AZ framework input interface uses the `AZCore` EBus system and defines the following classes and EBuses:

- `InputDevice` – A base class that represents a physical input device.
- `InputDeviceId` – A device name and device index that together uniquely identify an `InputDevice`.
- `InputDeviceNotificationBus` – An EBus interface that subscribes to events from input devices when they connect or disconnect.
- `InputDeviceRequestBus` – An EBus interface that queries input devices about their associated input channels or current connected state.
- `InputChannel` – A base class that represents a specific source of input data (for example, the left mouse button). Input devices typically have multiple input channels.
- `InputChannelId` – A name that uniquely identifies an `InputChannel`.
- `InputChannelNotificationBus` – An EBus interface that subscribes to events from input channels when the channels are active or when their state or value changes.
- `InputChannelRequestBus` – An EBus interface that obtains an input channel from the input channel's ID. You can then query the input channel directly for its current state or value.
- `InputChannelEventListener` – An event monitor that inherits from `InputChannelNotificationBus` but provides additional features. Subscribers can use the additional features to receive events in priority order or filter events based on the source device or channel. Subscribers can also consume events so that the events aren't passed on to lower priority monitors.

Processing the Input

To process input, you can simply inherit from `AzFramework::InputChannelEventListener`. You can also create a filter to receive events only from specific input devices or channels, then override `OnInputChannelEvent` to process the input data.

If you want to query the current input state directly, use `AzFramework::InputChannelRequestBus` to obtain an input channel's ID, and then query the input channel for its current state or value.

Text Input Interfaces

The following interfaces are designed to process text input. The input is delivered as a complete string of UTF-8 code points. This eliminates the requirement to keep track of and interpret individual code units or convert from other encodings.

- `InputTextEventNotificationBus` – An EBus interface that subscribes to text events from input devices or input channels.
- `InputTextEntryRequestBus` – An EBus interface that sends text entry requests. The requests inform input devices that the user is expecting to start or stop entering text.
- `InputTextEventListener` – An event monitor that inherits from `InputTextNotificationBus` but provides additional features. Subscribers can use the additional features to receive events in priority order or consume events so that the events aren't passed on to lower priority monitors.

Auxiliary Input Interfaces

The following AZ framework input auxiliary interfaces are only implemented by certain input devices. You can use these interfaces to query or post data related to device activity like vibration effects, motion sensors, or the mouse cursor.

- `InputHapticFeedbackBus` – An EBus interface that sends haptic feedback requests to connected input devices.
- `InputMotionSensorRequestBus` – An EBus interface that sends motion sensor requests to connected input devices.
- `InputSystemCursorRequestBus` – An EBus interface that queries or changes the state, position, or appearance of the system cursor.

Input Devices

All input device classes inherit from `AzFramework::InputDevice`. While multiple instances of the same class can exist, every input device is identified by a unique `InputDeviceId`.

Input Devices Included in Lumberyard

The base class is designed so that you can inherit from it to implement new types of input devices. However, Lumberyard already includes implementations for the following devices:

Device Type	Supported Platforms
Mouse	Windows, macOS
Keyboard	Windows and macOS
Gamepad	Windows, macOS, iOS, Android, consoles
Touch	iOS, Android
Motion	iOS, Android
Virtual keyboard	iOS, Android, consoles
Virtual reality controller	Oculus, OpenVR

This core set of input devices are managed by the `AzFramework::InputSystemComponent`. You can use this component to configure the type and number of input devices that your application creates at startup.

Creating New Input Devices

You can create and implement new types of input devices by inheriting from `AzFramework::InputDevice` and creating or destroying instances of the new class as required. For examples, see the Oculus

and OpenVR Gems, which define and create new types of input devices that inherit from `AzFramework::InputDevice`. Their code is located at the Lumberyard directory locations `\dev\Gems\Oculus` and `\dev\Gems\OpenVR`, respectively.

The implementation details of each input device differ depending on the device type and platform. However, almost all devices follow a similar pattern. They use a platform-specific API to obtain raw input data for each frame and then update all of the device's associated input channels accordingly.

When you implement an input device, use only your device's `TickInput` function to update the value of the input channels. This ensures that input events are delivered at the same time every frame.

Depending on how the input device gets the raw, platform-specific input, you might have to use one of the following techniques:

- Polling the underlying input API on each call to your input device's `TickInput` function. For examples, see all `InputDeviceGamepad` implementations in the directory `\dev\Code\Framework\AzFramework\AzFramework\Input\Devices\Gamepad`.
- Queuing or coalescing input data from the application's main message loop until the next call to `TickInput`. For an example, see `InputDeviceMouse` in the directory `\dev\Code\Framework\AzFramework\AzFramework\Input\Devices\Mouse`.
- Ensuring input channels are updated in a thread-safe manner if no raw input is received on the main thread. For an example, see `InputDeviceTouchAndroid` in the directory `\dev\Code\Framework\AzFramework\AzFramework\Input\Devices\Touch`.

Input Channels

Input devices can have multiple input channels. Each channel represents a discrete source of input data that is uniquely identified by an `InputChannelId`.

Input channels use `AzFramework::InputChannelNotificationBus::OnInputChannelEvent` to broadcast events. A channel broadcasts events when the channel's state or value changes from one frame to the next or if the channel is still active or held. Input channels can also be accessed directly and polled at any time to query their current state and value.

Creating Input Channels

Like the `AzFramework::InputDevice` base class, you can inherit from `AzFramework::InputChannel` to implement new types of input data sources.

Input Channels Included in Lumberyard

Lumberyard provides the following input channel implementations that are used by the devices listed in the Input Devices (p. 467) section. You can also use these implementations for new input devices.

Input Channel	Example Use
`InputChannelAnalog`	Gamepad trigger
`InputChannelAnalogWithPosition2D`	Touch with pressure at a position
`InputChannelAxis1D`	Gamepad thumbstick x or y
`InputChannelAxis2D`	Gamepad thumbstick x and y together
`InputChannelAxis3D`	Motion sensor acceleration, rotation, or magnetic field

Input Channel	Example Use
`InputChannelDelta`	Mouse wheel
`InputChannelDeltaWithSharedPosition2D`	Mouse movement
`InputChannelDigital`	Gamepad button or keyboard key
`InputChannelDigitalWithPosition2D`	Touch without pressure at a position
`InputChannelDigitalWithSharedPosition2D`	Mouse button at a position
`InputChannelQuaternion`	Motion sensor orientation

Migration Path

The new AZ framework input system was released in Lumberyard version 1.9. It is currently disabled by default.

Enabling AZ Framework Input

To enable the AZ framework input system, use the `AZ_FRAMEWORK_INPUT_ENABLED` compile time `define`. For more information, see the header file at the location `\dev\Code\Framework\AzFramework\AzFramework\Input\System\InputSystemComponent.h`.

When you enable AZ framework input, Lumberyard uses both the new AZ framework input system and the `AzToLyInput` implementation of the `IInput` interface. These replace the existing `CryInput` implementation of the `IInput` interface.

Backward Compatibility

All existing dependencies on `IInput` continue to function identically when using the new `AzToLyInput` implementation of this interface. Note that although the new input system is functional, it has not yet been thoroughly tested. In Lumberyard version 1.9, AZ framework input has been released as a preview so that you can try it before it becomes the default in a future release.

Legacy Input

This section contains information on legacy game input in Amazon Lumberyard. For information on Lumberyard's new input system, see Input in Amazon Lumberyard (p. 465).

Topics

- Action Maps (p. 469)
- CryInput (p. 470)
- Setting Up Controls and Action Maps (p. 472)

Action Maps

Note
Lumberyard's **Input** component replaces legacy action maps. For more information, see Input in the Amazon Lumberyard User Guide.

The Action Map Manager provides a high-level interface to handle input controls inside a game. The Action Map system is implemented in Lumberyard, and can be used directly by any code inside Lumberyard or the GameDLL.

Initializing the Action Map Manager

The Action Map Manager is initialized when Lumberyard is initialized. Your game must specify the path for the file `defaultProfile.xml` (by default, the path is `Game/Libs/Config/defaultProfile.xml`). You can do this by passing the path to the manager. For example:

```
IActionMapManager* pActionMapManager = m_pFramework->GetIActionMapManager();
if (pActionMapManager)
{
    pActionMapManager->InitActionMaps(filename);
}
```

Upon initialization, the Action Map Manager clears all existing initialized maps, filters, and controller layouts.

Receiving Actions During Runtime

You can enable the feature that allows action maps to receive actions during runtime. Use the following code to enable or disable an action map during runtime:

```
pActionMapMan->EnableActionMap("default", true);
```

To receive actions, implement the `IActionListener` interface in a class.

CryInput

The main purpose of CryInput is to provide an abstraction that obtains input and status from various input devices such as a keyboard, mouse, joystick, and so on.

It also supports sending feedback events back to input devices—for example, in the form of force feedback events.

The common interfaces for the input system can be found in `IInput.h`, in the CryCommon project.

IInput

`IInput` is the main interface of the input system. An instance implementing this interface is created automatically during system initialization in the `InitInput` function (`InitSystem.cpp` in CrySystem, see also `CryInput.cpp` in CryInput).

Only one instance of this interface is created. CrySystem also manages the update and shutdown of the input system.

This `IInput` instance is stored in the `SSystemGlobalEnvironment` structure gEnv. You can access it through `gEnv->pInput` or, alternatively, through the system interface by `GetISystem()->GetIInput()`. Access through the `gEnv` variable is the most commonly used method.

IInputEventListener

A common use case within the input system is to create listener classes in other modules (for example, CryGame) by inheriting from `IInputEventListener` and registering/unregistering the listener class with the input system for notifications of input events.

For example, the Action Map System registers itself as an input listener and forwards game events only for the keys defined in the profile configuration files to further abstract the player input from device to the game.

SInputEvent

`SInputEvent` encapsulates information that is created by any input device and received by all input event listeners.

IInputDevice

Input devices normally relate directly to physical input devices such as a joypad, mouse, keyboard, and so on. To create a new input device, you must implement all functions in the `IInputDevice` interface and register an instance of it with the Input System using the `AddInputDevice` function.

The `Init` function is called when registering the `IInputDevice` with the Input System; it is not necessary to manually call it when creating the input devices.

The `Update` function is called at every update of the Input System—this is generally where the state of the device should be checked/updated and the Input Events generated and forwarded to the Input System.

It is common for input devices to create and store a list in SInputSymbol of each symbol the input device is able to generate in the Init function. Then, in the update function, the symbols for the buttons/axes that changed are looked up and used (via their AssignTo function) to fill in most of the information needed for the events, which are then forwarded to the input system.

Example:

```
// function from CInputDevice (accessible only within CryInput)
MapSymbol(...)
{
     SInputSymbol* pSymbol = new SInputSymbol( deviceSpecificId, keyId, name, type );
     pSymbol->user = user;
     pSymbol->deviceId = m_deviceId;
     m_idToInfo[ keyId ] = pSymbol;
     m_devSpecIdToSymbol[ deviceSpecificId ] = pSymbol;
     m_nameToId[ name ] = deviceSpecificId;
     m_nameToInfo[ name ] = pSymbol;

     return pSymbol;
}
bool CMyKeyboardInputDevice::Init()
{
      ...
     //CreateDeviceEtc();
      ...
     m_symbols[ DIK_1 ] = MapSymbol( DIK_1, eKI_1, "1" );
     m_symbols[ DIK_2 ] = MapSymbol( DIK_2, eKI_2, "2" );
      ...
}
void CMyKeyboardInputDevice::Update( ... )
{
     // Acquire device if necessary
      ...
     // Will probably want to check for all keys, so the following section might be part of
 a loop
     SInputSymbol* pSymbol = m_symbols[ deviceKeyId ];
      ...
     // check if state changed
```

```
     ...
    // This is an example for, when pressed, see ChangeEvent function for axis type
 symbols
    pSymbol->PressEvent( true );

    SInputEvent event;
    pSymbol->AssignTo( event, modifiers );

    gEnv->pInput->PostInputEvent( event );
}
```

To forward events to the input system so that event listeners can receive them, use the `PostInputEvent` function from `IInput`.

If adding your input device to CryInput, it may be useful to inherit directly from `CInputDevice`, as it already provides a generic implementation for most functions in IInputDevice.

Note
This file is included with the full source of CryEngine and is not available in the FreeSDK or GameCodeOnly solutions. For these licenses please derive from IInputDevice directly.

Setting Up Controls and Action Maps

This section describes how to create and modify action maps to customize the controls to the needs of your game.

Action map profiles for all supported platforms are located in `Game\Libs\Config\Profile\DefaultProfile.xml`. This default XML file organizes controls into the following sections, each of which is controlled by its own action map:

- multiplayer
- singleplayer
- debug
- flycam
- default
- player
- vehicle
- land vehicle
- sea vehicle
- helicopter

Each action map can be enabled or disabled during runtime from Flow Graph, in Lua scripts, or in C++ code.

See the topic Default Controller Mapping (p. 475) for an overview of the controls in the SDK package.

Action Maps

An action map is a set of key/button mappings for a particular game mode. For example, there is an `<actionmap>` section for helicopter controls called "Helicopter", which means that everything inside that section consists of key and button bindings that apply only when flying a helicopter. To change your common in-game bindings, go to the section starting with `<actionmap name="default">`. There are also sections for multiplayer-specific bindings and, of course, any other vehicles or modes you need.

The following is an overview of a standard action map, in this case the standard debug one:

```
<actionmap name="debug" version="22">
 <!-- debug keys - move to debug when we can switch devmode-->
  <action name="flymode" onPress="1" noModifiers="1" keyboard="f3" />
  <action name="godmode" onPress="1" noModifiers="1" keyboard="f4" />
  <action name="toggleaidebugdraw" onPress="1" noModifiers="1" keyboard="f11" />
  <action name="togglepdrawhelpers" onPress="1" noModifiers="1" keyboard="f10" />
  <action name="ulammo" onPress="1" noModifiers="1" keyboard="np_2" />
  <action name="debug" onPress="1" keyboard="7" />
  <action name="thirdperson" onPress="1" noModifiers="1" keyboard="f1" />
  <!-- debug keys - end -->
 </actionmap>
```

Versioning

```
<actionmap name="debug" version="22">
```

When the version value is incremented, Lumberyard ensures that the user profile receives the newly updated action map. This is quite useful when deploying new actions in a patch of a game that is already released. If the version stays the same, changes or additions to the action maps are not propagated to the user profile.

Activation Modes

The following activation modes are available:

- onPress – The action key is pressed
- onRelease – The action key is released
- onHold – The action key is held
- always – Permanently activated

The activation mode is passed to action listeners and identified by the corresponding Lua constant:

- eAAM_OnPress
- eAAM_OnRelease
- eAAM_OnHold
- eAAM_Always

Modifiers available:

- retriggerable
- holdTriggerDelay
- holdRepeatDelay
- noModifiers – Action takes place only if no **Ctrl**, **Shift**, **Alt**, or **Win** keys are pressed
- consoleCmd – Action corresponds to a console command
- pressDelayPriority
- pressTriggerDelay
- pressTriggerDelayRepeatOverride
- inputsToBlock – Specify the input actions to block here

- inputBlockTime – Time to block the specified input action

Action Filters

You can also define action filters directly in your `defaultProfile.xml` file. The following attributes are available:

- **name** – How the filter will be identified.
- **type** – Specify `actionFail` to cause an action to fail. Specify `actionPass` to allow the action to succeed.

A sample action filter follows:

```
<actionfilter name="no_move" type="actionFail">
 <!-- actions that should be filtered -->
  <action name="crouch"/>
  <action name="jump"/>
  <action name="moveleft"/>
  <action name="moveright"/>
  <action name="moveforward"/>
  <action name="moveback"/>
  <action name="sprint"/>
  <action name="xi_movey"/>
  <action name="xi_movex"/>
 <!-- actions end -->
</actionfilter>
```

Controller Layouts

Links to the different controller layouts can also be stored in this file:

```
<controllerlayouts>
 <layout name="Layout 1" file="buttonlayout_alt.xml"/>
 <layout name="Layout 2" file="buttonlayout_alt2.xml"/>
 <layout name="Layout 3" file="buttonlayout_lefty.xml"/>
 <layout name="Layout 4" file="buttonlayout_lefty2.xml"/>
</controllerlayouts>
```

Note
The "file" attribute links to a file stored in "libs/config/controller/" by default.

Working with Action Maps During Runtime

In Lumberyard, you can use the console command `i_reloadActionMaps` to re-initialize the defined values. The ActionMapManager sends an event to all its listeners to synchronize the values throughout the engine. If you're using a separate GameActions file like GameSDK, make sure this class will receive the update to re-initialize the actions/filters in place. Keep in mind that it's not possible to define action maps, filters, or controller layouts with the same name in multiple places (for example, action filter `no_move` defined in `defaultProfile.xml` and the `GameActions` file).

To handle actions during runtime, you can use flow graphs or Lua scripts.

- Flow Graph – Input nodes can be used to handle actions. Only digital inputs can be handled from a flow graph. For more information, see Flow Graph System in the Amazon Lumberyard User Guide.
- Lua script – While actions are usually not intended to be received directly by scripts, it is possible to interact with the Action Map Manager from Lua.

Default Controller Mapping

The default mapping for input on the PC is shown in the following table. To reconfigure the controls for your game, follow the instructions in Setting Up Controls and Action Maps (p. 472) and Action Maps (p. 469).

Player Action	PC
Player Movement	W, A, S, D
Player Aim	Mouse XY
Jump	Spacebar
Sprint	Shift
Crouch	C
Slide (when sprinting)	C
Fire	Mouse 1
Zoom	Mouse 2
Melee	V
Fire Mode	2
Reload	R
Use	F
Toggle Weapon	1
Toggle Explosive	3
Toggle Binoculars	B
Toggle Light (attachment)	L
Third Person Camera	F1

Vehicle Action	PC
Accelerate	W
Boost	Shift
Brake/Reverse	S
Handbrake	Spacebar
Steer	A/D
Look	Mouse XY
Horn	H
Fire	Mouse 1

Vehicle Action	PC
Change Seat	C
Headlights	L

Helicopter Action	PC
Ascend	W
Descend	S
Roll Left	A
Roll Right	D
Yaw Left	Mouse X (left)
Yaw Right	Mouse X (right)
Pitch Up	Mouse Y (up)
Pitch Down	Mouse Y (down)

Multiplayer Action	PC
Show Scoreboard	TAB

Key Naming Conventions

This page lists some of the name conventions used for action maps.

Key Gestures

Letters	"a" - "z"
Numbers	"1" - "0"
Arrows	"up", "down", "left", "right"
Function keys	"f1" - "f15"
Numpad	"np_1" - "np_0", "numlock", "np_divide", "np_multiply", "np_subtract", "np_add", "np_enter", "np_period"
Esc	"escape"
~	"tilde"
Tab	"tab"
CapsLock	"capslock"
Shift	"lshift", "rshift"
Ctrl	"lctrl", "rctrl"

Alt	"lalt", "ralt"
spacebar	"space"
-	"minus"
=	"equals"
Backspace	"backspace"
[ ]	"lbracket", "rbracket"
"\"	"backslash"
;	"semicolon"
'	"apostrophe"
Enter	"enter"
,	"comma"
.	"period"
/	"slash"
Home	"home"
End	"end"
Delete	"delete"
PageUp	"pgup"
PageDown	"pgdn"
Insert	"insert"
ScrollLock	"scrolllock"
PrintScreen	"print"
Pause/Break	"pause"

Mouse Gestures

Left/primary mouse button	"mouse1"
Right/secondary mouse button	"mouse2"
Mouse wheel up	"mwheel_up"
Mouse wheel down	"mwheel_down"
New position along x-axis	"maxis_x"
New position along y-axis	"maxis_y"

CryCommon

The `Code\CryCommon` directory is the central directory for all the engine interfaces (as well as some commonly used code stored there to encourage reuse).

This section includes the following topics:

- CryExtension (p. 478)
- CryString (p. 501)
- ICrySizer (p. 502)
- Serialization Library (p. 502)

CryExtension

The complexity of Lumberyard can be challenging to both newcomers and experienced users who want to understand, configure, run, and extend it. Refactoring Lumberyard into extensions makes it easier to manage. Existing features can be unplugged (at least to some degree), replaced, or customized, and new features added. Extensions can consolidate code for a single feature in one location. This avoids having to implement a feature piecemeal across a number of the engine's base modules. Refactoring into extensions can also make the system more understandable at a high level.

Lumberyard's extension framework is loosely based on some fundamental concepts found in Microsoft's Component Object Model (COM). The framework defines two base interfaces that each extension needs to implement, namely `ICryUnknown` and `ICryFactory`. These are similar to COM's `IUnknown` and `IClassFactory`. The interfaces serve as a base to instantiate extensions, allow interface type casting, and enable query and exposure functionality.

The framework utilizes the concept of shared pointers and is implemented in a way to enforce their consistent usage to help reduce the chance of resource leaks. A set of C++ templates wrapped in a few macros is provided as Glue Code Macros (p. 484) that encourage engine refactoring into extensions. The glue code efficiently implements all base services and registers extensions within the engine. Additionally, a few helper functions implement type-safe casting of interface pointers, querying the IDs of extension interfaces, and convenient instantiation of extension classes. Hence, repetitive writing of tedious boilerplate code is unnecessary, and the potential for introducing bugs is reduced. An example is provided in the section Using Glue Code (p. 493). If the provided glue code is not applicable, then you must implement the interfaces and base services manually, as described in the section Without Using Glue Code (p. 495).

Clients access extensions through a system wide factory registry. The registry allows specific extension classes to be searched by either name or ID, and extensions to be iterated by using an interface ID.

Composites

The framework allows extensions to expose certain internal objects that they aggregate or are composed of. These so called *composites* are extensions themselves because they inherit from `ICryUnknown`. Composites allow you to reuse desired properties like type information at runtime for safe casting and loose coupling.

Shared and raw interface pointers

Although the framework was designed and implemented to utilize shared pointers and enforce their usage in order to reduce the possibility of resource leaks, raw interface pointers can still be acquired. Therefore, care needs to be taken to prevent re-wrapping those raw interface pointers in shared pointer objects. If the original shared pointer object is not passed during construction so that its internal reference counter can be referred to, the consistency of reference counting will be broken and crashes can occur. A best practice is to use raw interface pointers only to operate on interfaces temporarily, and not store them for later use.

GUIDs

You must use globally unique identifiers (GUIDs) to uniquely identify extensions and their interfaces. GUIDs are essentially 128-bit numbers generated by an algorithm to ensure they only exist once within a system such as Lumberyard. The use of GUIDs is key to implementing the type-safe casting of extension interfaces, which is particularly important in large scale development projects. To create GUIDs, you can use readily available tools like the **Create GUID** feature in Visual Studio or the macro below.

GUIDs are defined as follows.

```
struct CryGUID
{
 uint64 hipart;
 uint64 lopart;

 ...
};

typedef CryGUID CryInterfaceID;
typedef CryGUID CryClassID;
```

Declared in the following framework header files:

- `CryCommon/CryExtension/CryGUID.h`
- `CryCommon/CryExtension/CryTypeID.h`

The following Visual Studio macro can be used to generate GUIDs conveniently within the IDE. The macro writes GUIDs to the current cursor location in the source code editor window. Once added to **Macro Explorer**, the macro can be bound to a keyboard shortcut or (custom) toolbar.

```
Public Module CryGUIDGenModule

    Sub GenerateCryGUID()
        Dim newGuid As System.Guid
        newGuid = System.Guid.NewGuid()

        Dim guidStr As String
```

```
        guidStr = newGuid.ToString("N")
        guidStr = guidStr.Insert(16, ", 0x")
        guidStr = guidStr.Insert(0, "0x")

        REM guidStr = guidStr + vbNewLine
        REM guidStr = guidStr + newGuid.ToString("D")

        DTE.ActiveDocument.Selection.Text = guidStr
    End Sub

End Module
```

ICryUnknown

`ICryUnknown` provides the base interface for all extensions. If making it the top of the class hierarchy is not possible or desired (for example, in third party code), you can apply an additional level of indirection to expose the code by using the extension framework. For an example, see If ICryUnknown Cannot Be the Base of the Extension Class (p. 499).

ICryUnknown is declared as follows.

```
struct ICryUnknown
{
 CRYINTERFACE_DECLARE(ICryUnknown, 0x1000000010001000, 0x1000100000000000)

 virtual ICryFactory* GetFactory() const = 0;

protected:
 virtual void* QueryInterface(const CryInterfaceID& iid) const = 0;
 virtual void* QueryComposite(const char* name) const = 0;
};

typedef boost::shared_ptr<ICryUnknown> ICryUnknownPtr;
```

- `GetFactory()` returns the factory with which the specified extension object was instantiated. Using the provided glue code this function has constant runtime.

- `QueryInterface()` returns a void pointer to the requested interface if the extension implements it, or NULL otherwise. This function was deliberately declared as protected to enforce usage of type-safe interface casting semantics. For information on casting semantics, see Interface casting semantics (p. 482). When the provided glue code is used, this function has a (worst case) run time that is linear in the number of supported interfaces. Due to glue code implementation details, no additional internal function calls are needed. A generic code generator produces a series of instructions that compares interface IDs and returns a properly cast pointer.

- `QueryComposite()` returns a void pointer to the queried composite if the extension exposes it; otherwise, NULL. As with `QueryInterface()`, this function was deliberately declared as protected to enforce type querying. For information on type querying, see Querying composites (p. 483). The function has a (worst case) run time linear in the number of exposed composites.

- Unlike in COM, `ICryUnknown` does not have `AddRef()` and `Release()`. Reference counting is implemented in an non-intrusive way by using shared pointers that are returned by the framework when extension classes are instantiated.

Declared in the following framework header file:

- `CryCommon/CryExtension/ICryUnknown.h`

ICryFactory

`ICryFactory` provides the base interface to instantiate extensions. It is declared as follows.

```
struct ICryFactory
{
 virtual const char* GetClassName() const = 0;
 virtual const CryClassID& GetClassID() const = 0;
 virtual bool ClassSupports(const CryInterfaceID& iid) const = 0;
 virtual void ClassSupports(const CryInterfaceID*& pIIDs, size_t& numIIDs) const = 0;
 virtual ICryUnknownPtr CreateClassInstance() const = 0;

protected:
 virtual ~ICryFactory() {}
};
```

- `GetClassName()` returns the name of the extension class. This function has constant run time when the provided glue code is used.

- `GetClassID()` returns the ID of the extension class. This function has constant run time when the provided glue code is used.

- `ClassSupports(iid)` returns true if the interface with the specified ID is supported by the extension class; otherwise, false. This function has a (worst case) run time linear in the number of supported interfaces when the provided glue code is used.

- `ClassSupports(pIIDs, numIIDs)` returns the pointer to an internal array of IDs enumerating all of the interfaces that this extension class supports as well as the length of the array. This function has constant run time when the provided glue code is used.

- `CreateClassInstance()` dynamically creates an instance of the extension class and returns a shared pointer to it. If the extension class is implemented as a singleton, it will return a (static) shared pointer that wraps the single instance of that extension class. This function has constant run time when the provided glue code is used, except for the cost of the constructor call for non-singleton extensions.

- The destructor is declared protected to prevent explicit destruction from the client side by using `delete`, `boost::shared_ptr<T>`, etc. `ICryFactory` instances exist (as singletons) throughout the entire lifetime of any Lumberyard process and **must not** be destroyed.

Declared in the following framework header file:

- `CryCommon/CryExtension/ICryFactory.h`

ICryFactoryRegistry

ICryFactoryRegistry is a system-implemented interface that enables clients to query extensions. It is declared as follows.

```
struct ICryFactoryRegistry
```

```
{
 virtual ICryFactory* GetFactory(const char* cname) const = 0;
 virtual ICryFactory* GetFactory(const CryClassID& cid) const = 0;
 virtual void IterateFactories(const CryInterfaceID& iid, ICryFactory** pFactories, size_t&
 numFactories) const = 0;

protected:
 virtual ~ICryFactoryRegistry() {}
};
```

- `GetFactory(cname)` returns the factory of the extension class with the specified name; otherwise, NULL.
- `GetFactory(cid)` returns the factory of the extension class with the specified ID; otherwise, NULL.
- `IterateFactory()` if `pFactories` is not NULL, `IterateFactory` copies up to `numFactories` entries of pointers to extension factories that support `iid`. `numFactories` returns the number of pointers copied. If `pFactories` is NULL, `numFactories` returns the total amount of extension factories that support `iid`.
- The destructor was declared protected to prevent explicit destruction from the client side by using `delete`, `boost::shared_ptr<T>`, etc. `ICryFactoryRegistry` is a system interface and that exists throughout the entire lifetime of any CryEngine process and **must not** be destroyed.

Declared in the following framework header file:

- `CryCommon/CryExtension/ICryFactoryRegistry.h`

Additional Extensions

Use the methods defined in `ICryUnknown` for additional functionality.

Interface casting semantics

Interface casting semantics have been implemented to provide syntactically convenient and type-safe casting of interfaces. The syntax was designed to conform with traditional C++ type casts and respects `const` rules.

```
ICryFactory* pFactory = ...;
assert(pFactory);

ICryUnknownPtr pUnk = pFactory->CreateClassInstance();

IMyExtensionPtr pMyExtension = cryinterface_cast<IMyExtension>(pUnk);

if (pMyExtension)
{
 // it's safe to work with pMyExtension
}
```

Interface casting also works on raw interface pointers. However, please consider the guidelines described in the section Shared and raw interface pointers (p. 479).

Declared in the following framework header file:

- `CryCommon/CryExtension/ICryUnknown.h`

Querying interface identifiers

Occasionally, it is necessary to know the ID of an interface, e.g. to pass it to `ICryFactoryRegistry::IterateFactories()`. This can be done as follows.

```
CryInterfaceID iid = cryiidof<IMyExtension>();
```

Declared in the following framework header file:

- `CryCommon/CryExtension/ICryUnknown.h`

Checking pointers

Use this extension to check whether pointers to different interfaces belong to the same class instance.

```
IMyExtensionAPtr pA = ...;
IMyExtensionBPtr pB = ...;

if (CryIsSameClassInstance(pA, pB))
{
 ...
}
```

This works on both shared and raw interface pointers.

Declared in the following framework header file:

- `CryCommon/CryExtension/ICryUnknown.h`

Querying composites

Extensions can be queried for composites as follows.

```
IMyExtensionPtr pMyExtension = ...;

ICryUnknownPtr pCompUnk = crycomposite_query(pMyExtension, "foo");

IFooPtr pComposite = cryinterface_cast<IFoo>(pCompUnk);
if (pComposite)
{
 // it's safe to work with pComposite, a composite of pMyExtention exposed as "foo"
 implementing IFoo
}
```

A call to `crycomposite_query()` might return NULL if the specified composite has not yet been created. To gather more information, the query can be rewritten as follows.

```
IMyExtensionPtr pMyExtension = ...;

bool exposed = false;
ICryUnknownPtr pCompUnk = crycomposite_query(pMyExtension, "foo", &exposed);

if (exposed)
{
 if (pCompUnk)
```

```
 {
  // "foo" exposed and created

  IFooPtr pComposite = cryinterface_cast<IFoo>(pCompUnk);
  if (pComposite)
  {
   // it's safe to work with pComposite, a composite of pMyExtention exposed as "foo"
 implementing IFoo
  }
 }
 else
 {
  // "foo" exposed but not yet created
 }
}
else
{
 // "foo" not exposed by pMyExtension
}
```

As with interface casting composite, queries work on raw interface pointers. However, please consider the guidelines described in the section Shared and raw interface pointers (p. 479).

Declared in the following framework header file:

- `CryCommon/CryExtension/ICryUnknown.h`

Glue Code Macros

The following macros provide glue code to implement the base interfaces and services to support the framework in a thread-safe manner. You are strongly encouraged to use them when you implement an extension.

For examples of how these macros work together, see Using Glue Code (p. 493).

Declared in the following framework header files:

- `CryCommon/CryExtension/Impl/ClassWeaver.h`
- `CryCommon/CryExtension/CryGUID.h`

CRYINTERFACE_DECLARE(iname, iidHigh, iidLow)

Declares an interface and associated ID. Protects the interfaces from accidentally being deleted on client side. That is, it allows destruction only by using `boost::shared_ptr<T>`. This macro is required once per interface declaration.

Parameters

iname

The (C++) name of the interface as declared.

iidHigh

The higher 64-bit part of the interface ID (GUID).

iidLow

The lower 64-bit part of the interface ID (GUID).

CRYINTERFACE_BEGIN()

Start marker of the interface list inside the extension class implementation. Required once per extension class declaration.

CRYINTERFACE_ADD(iname)

Marker to add interfaces inside the extension class declaration. It has to be declared in between `CRYINTERFACE_BEGIN()` and any of the `CRYINTERFACE_END*()` markers. Only declare the interfaces that the class directly inherits. If deriving from an existing extension class or classes, the inherited interfaces get added automatically. If an interface is declared multiple times, duplicates will be removed. It is not necessary to add `ICryUnknown`.

> **Warning**
> Other interfaces that are not declared will not be castable by using `cryinterface_cast<T>()`.

Parameters

iname

The (C++) name of the interface to be added.

CRYINTERFACE_END()

End marker of the interface list inside the extension class declaration. Use this if not inheriting from any already existing extension class. Required once per extension class declaration. Mutually exclusive with any of the other `CRYINTERFACE_END*()` markers.

CRYINTERFACE_ENDWITHBASE(base)

End marker of the interface list inside the extension class declaration. Use this if inheriting from an already existing extension class. Required once per extension class declaration. Mutually exclusive with any of the other `CRYINTERFACE_END*()` markers.

Parameters

base

The (C++) name of the extension class from which derived.

CRYINTERFACE_ENDWITHBASE2(base0, base1)

End marker of the interface list inside the extension class declaration. Use this if inheriting from two already existing extension classes. Required once per extension class declaration. Mutually exclusive with any of the other `CRYINTERFACE_END*()` markers.

Parameters

base0

The (C++) name of the first extension class from which derived.

base1

The (C++) name of the second extension class from which derived.

CRYINTERFACE_ENDWITHBASE3(base0, base1, base2)

End marker of the interface list inside the extension class declaration. Use this if inheriting from three already existing extension classes. Required once per extension class declaration. Mutually exclusive with any of the other `CRYINTERFACE_END*()` markers.

Parameters

base0

The (C++) name of the first extension class from which derived.

base1

The (C++) name of the second extension class from which derived.

base2

The (C++) name of the 3rd extension class from which derived.

CRYINTERFACE_SIMPLE(iname)

Convenience macro for the following code sequence (probably the most common extension case):

```
CRYINTERFACE_BEGIN()
 CRYINTERFACE_ADD(iname)
CRYINTERFACE_END()
```

Parameters

iname

The (C++) name of the interface to be added.

CRYCOMPOSITE_BEGIN()

Start marker of the list of exposed composites.

CRYCOMPOSITE_ADD(member, membername)

Marker to add a member of the extension class to the list of exposed composites.

Parameters

member

The (C++) name of the extension class member variable to be exposed. It has to be of type `boost::shared_ptr<T>`, where `T` inherits from `ICryUnknown`. This condition is enforced at compile time.

membername

The name (as C-style string) of the composite by which the composite can later be queried at runtime.

CRYCOMPOSITE_END(implclassname)

End marker of the list of exposed composites. Use this if not inheriting from any extension class that also exposes composites. Mutually exclusive with any of the other `CRYCOMPOSITE_END*()` markers.

Parameters

implclassname

The (C++) name of the extension class to be implemented.

CRYCOMPOSITE_ENDWITHBASE(implclassname, base)

End marker of the list of exposed composites. Use this if inheriting from one extension class that also exposes composites. Queries will first search in the current class and then look into the base class to find a composite that matches the requested name specified in `crycomposite_query()`. Mutually exclusive with any of the other `CRYCOMPOSITE_END*()` markers.

Parameters

implclassname

The (C++) name of the extension class to be implemented.

base

The (C++) name of the extension class derived from.

CRYCOMPOSITE_ENDWITHBASE2(implclassname, base0, base1)

End marker of the list of exposed composites. Use this if inheriting from two extension classes that also expose composites. Queries will first search in the current class and then look into the base classes to find a composite matching the requested name specified in `crycomposite_query()`. Mutually exclusive with any of the other `CRYCOMPOSITE_END*()` markers.

Parameters

implclassname

The (C++) name of the extension class to be implemented.

base0

The (C++) name of the first extension class from which derived.

base1

The (C++) name of the second extension class which derived.

CRYCOMPOSITE_ENDWITHBASE3(implclassname, base0, base1, base2)

End marker of the list of exposed composites. Use this if inheriting from three extension classes that also expose composites. Queries will first search in the current class and then look into the base classes to

find a composite matching the requested name specified in `crycomposite_query()`. Mutually exclusive with any of the other `CRYCOMPOSITE_END*()` markers.

Parameters

implclassname

The (C++) name of the extension class to be implemented.

base0

The (C++) name of the first extension class from which derived.

base1

The (C++) name of the second extension class from which derived.

base2

The (C++) name of the third extension class from which derived.

CRYGENERATE_CLASS(implclassname, cname, cidHigh, cidLow)

Generates code to support base interfaces and services for an extension class that can be instantiated an arbitrary number of times. Required once per extension class declaration. Mutually exclusive to CRYGENERATE_SINGLETONCLASS().

Parameters

implclassname

The C++ class name of the extension.

cname

The extension class name with which it is registered in the registry.

cidHigh

The higher 64-bit part of the extension's class ID (GUID) with which it is registered in the registry.

cidLow

The lower 64-bit part of the extension's class ID (GUID) with which it is registered in the registry.

CRYGENERATE_SINGLETONCLASS(implclassname, cname, cidHigh, cidLow)

Generates code to support base interfaces and services for an extension class that can be instantiated only once (singleton). Required once per extension class declaration. Mutually exclusive with `CRYGENERATE_CLASS()`.

Parameters

implclassname

The C++ class name of the extension.

cname

The extension class name with which it is registered in the registry.

cidHigh

The higher 64-bit part of the extension's class ID (GUID) with which it is registered in the registry.

cidLow

The lower 64-bit part of the extension's class ID (GUID) with which it is registered in the registry.

CRYREGISTER_CLASS(implclassname)

Registers the extension class in the system. Required once per extension class at file scope.

Parameters

implclassname

The C++ class name of the extension.

MAKE_CRYGUID(high, low)

Parameters

Constructs a CryGUID. Useful when searching the registry for extensions by class ID.

high

The higher 64-bit part of the GUID.

low

The lower 64-bit part of the GUID.

CryExtension Samples

Sample 1 - Implementing a Source Control Plugin by Using Extensions

```
//////////////////////////////////////////////////////////////////////////
// source control interface

struct ISourceControl : public ICryUnknown
{
 CRYINTERFACE_DECLARE(ISourceControl, 0x399d8fc1d94044cc, 0xa70d2b4e58921453)

 virtual void GetLatest(const char* filename) = 0;
 virtual void Submit() = 0;
};

typedef cryshared_ptr<ISourceControl> ISourceControlPtr;

//////////////////////////////////////////////////////////////////////////
// concrete implementations of source control interface

class CSourceControl_Perforce : public ISourceControl
{
```

```
 CRYINTERFACE_BEGIN()
  CRYINTERFACE_ADD(ISourceControl)
 CRYINTERFACE_END()

 CRYGENERATE_SINGLETONCLASS(CSourceControl_Perforce, "CSourceControl_Perforce",
 0x7305bff20ee543e3, 0x820792c56e74ecda)

 virtual void GetLatest(const char* filename) { ... };
 virtual void Submit() { ... };
};

CRYREGISTER_CLASS(CSourceControl_Perforce)


class CSourceControl_SourceSafe : public ISourceControl
{
 CRYINTERFACE_BEGIN()
  CRYINTERFACE_ADD(ISourceControl)
 CRYINTERFACE_END()

 CRYGENERATE_SINGLETONCLASS(CSourceControl_SourceSafe, "CSourceControl_SourceSafe",
 0x1df62628db9d4bb2, 0x8164e418dd5b6691)

 virtual void GetLatest(const char* filename) { ... };
 virtual void Submit() { ... };
};

CRYREGISTER_CLASS(CSourceControl_SourceSafe)

//////////////////////////////////////////////////////////////////////////
// using the interface (submitting changes)

void Submit()
{
 ICryFactoryRegistry* pReg = gEnv->pSystem->GetFactoryRegistry();

 ICryFactory* pFactory = 0;
 size_t numFactories = 1;
 pReg->IterateFactories(cryiidof<ISourceControl>(), &pFactory, numFactories);

 if (pFactory)
 {
  ISourceControlPtr pSrcCtrl = cryinterface_cast<ISourceControl>(pFactory-
>CreateClassInstance());
  if (pSrcCtrl)
  {
   pSrcCtrl->Submit();
  }
 }
}
```

Using Extensions

Working with Specific Extension Classes

To work with a specific extension class, a client needs to know the extension's class name or class id and the interface(s) that the class supports. With this information, the class factory can be queried from the registry, an instance created and worked with as in the following example.

```
// IMyExtension.h
#include <CryExtension/ICryUnknown.h>
```

```
struct IMyExtension : public ICryUnknown
{
 ...
};

typedef boost::shared_ptr<IMyExtension> IMyExtensionPtr;
```

```
// in client code
#include <IMyExtension.h>
#include <CryExtension/CryCreateClassInstance.h>

IMyExtensionPtr pMyExtension;

#if 0
// create extension by class name
if (CryCreateClassInstance("MyExtension", pMyExtension))
#else
// create extension by class id, guaranteed to create instance of same kind
if (CryCreateClassInstance(MAKE_CRYGUID(0x68c7f0e0c36446fe, 0x82a3bc01b54dc7bf),
 pMyExtension))
#endif
{
 // it's safe to work with pMyExtension
}
```

```
// verbose version of client code above
#include <IMyExtension.h>
#include <CryExtension/ICryFactory.h>
#include <CryExtension/ICryFactoryRegistry.h>

ICryFactoryRegistry* pReg = ...;

#if 0
// search extension by class name
ICryFactory* pFactory = pReg->GetFactory("MyExtension");
#else
// search extension by class id, guaranteed to yield same factory as in search by class
 name
ICryFactory* pFactory = pReg->GetFactory(MAKE_CRYGUID(0x68c7f0e0c36446fe,
 0x82a3bc01b54dc7bf));
#endif

if (pFactory) // see comment below
{
 ICryUnknownPtr pUnk = pFactory->CreateClassInstance();
 IMyExtensionPtr pMyExtension = cryinterface_cast<IMyExtension>(pUnk);
 if (pMyExtension)
 {
  // it's safe to work with pMyExtension
 }
}
```

As an optimization, you can enhance the `if` check as follows.

```
if (pFactory && pFactory->ClassSupports(cryiidof<IMyExtension>()))
{
 ...
```

This version of the `if` statement will check interface support before the extension class is instantiated. This check prevents the unnecessary (and potentially expensive) construction and destruction of extensions that are incompatible with a given interface.

Finding Extension Classes that Support a Specific Interface

To determine how many extension classes in the registry support a given interface, and to list them, clients can submit queries similar to the following.

```
// IMyExtension.h
#include <CryExtension/ICryUnknown.h>

struct IMyExtension : public ICryUnknown
{
 ...
};

// in client code
#include <IMyExtension.h>
#include <CryExtension/ICryFactory.h>
#include <CryExtension/ICryFactoryRegistry.h>

ICryFactoryRegistry* pReg = ...;

size_t numFactories = 0;
pReg->IterateFactories(cryiidof<IMyExtension>(), 0, numFactories);

ICryFactory** pFactories = new ICryFactory*[numFactories];

pReg->IterateFactories(cryiidof<IMyExtension>(), pFactories, numFactories);

...

delete [] pFactories;
```

Implementing Extensions Using the Framework

The following section explains in detail how to implement extensions in Lumberyard. It provides examples that use glue code and do not use glue code. The section also shows you how to utilize the framework in cases where `ICryUnknown` cannot be the base of the extension interface.

Recommended Layout for Including Framework Header Files

The public interface header that will be included by the client should look like the following.

```
// IMyExtension.h
#include <CryExtension/ICryUnknown.h>

struct IMyExtension : public ICryUnknown
{
 ...
};
```

If you are using glue code, declare the implementation class of the extension in the header file as follows.

```
// MyExtension.h
#include <IMyExtension.h>
#include <CryExtension/Impl/ClassWeaver.h>

class CMyExtension : public IMyExtension
{
 ...
```

```
};
```

Using Glue Code

The first example shows a possible implementation of the `IMyExtension` class in the previous examples.

```
//////////////////////////////////////////
// public section

// IMyExtension.h
#include <CryExtension/ICryUnknown.h>

struct IMyExtension : public ICryUnknown
{
 CRYINTERFACE_DECLARE(IMyExtension, 0x4fb87a5f83f74323, 0xa7e42ca947c549d8)

 virtual void CallMe() = 0;
};

typedef boost::shared_ptr<IMyExtension> IMyExtensionPtr;

//////////////////////////////////////////
// private section not visible to client

// MyExtension.h
#include <IMyExtension.h>
#include <CryExtension/Impl/ClassWeaver.h>

class CMyExtension : public IMyExtension
{
 CRYINTERFACE_BEGIN()
  CRYINTERFACE_ADD(IMyExtension)
 CRYINTERFACE_END()

 CRYGENERATE_CLASS(CMyExtension, "MyExtension", 0x68c7f0e0c36446fe, 0x82a3bc01b54dc7bf)

public:
 virtual void CallMe();
};

// MyExtension.cpp
#include "MyExtension.h"

CRYREGISTER_CLASS(CMyExtension)

CMyExtension::CMyExtension()
{
}

CMyExtension::~CMyExtension()
{
}

void CMyExtension::CallMe()
{
 printf("Inside CMyExtension::CallMe()...");
}
```

The following example shows how the extension class `MyExtension` can be customized and expanded to implement two more interfaces, `IFoo` and `IBar`.

```
//////////////////////////////////////////
// public section
```

```
// IFoo.h
#include <CryExtension/ICryUnknown.h>

struct IFoo : public ICryUnknown
{
 CRYINTERFACE_DECLARE(IFoo, 0x7f073239d1e6433f, 0xb59c1b6ff5f68d79)

 virtual void Foo() = 0;
};

// IBar.h
#include <CryExtension/ICryUnknown.h>

struct IBar : public ICryUnknown
{
 CRYINTERFACE_DECLARE(IBar, 0xa9361937f60d4054, 0xb716cb711970b5d1)

 virtual void Bar() = 0;
};

//////////////////////////////////////////
// private section not visible to client

// MyExtensionCustomized.h
#include "MyExtension.h"
#include <IFoo.h>
#include <IBar.h>
#include <CryExtension/Impl/ClassWeaver.h>

class CMyExtensionCustomized : public CMyExtension, public IFoo, public IBar
{
 CRYINTERFACE_BEGIN()
  CRYINTERFACE_ADD(IFoo)
  CRYINTERFACE_ADD(IBar)
 CRYINTERFACE_ENDWITHBASE(CMyExtension)

 CRYGENERATE_CLASS(CMyExtensionCustomized, "MyExtensionCustomized", 0x07bfa7c543a64f0c,
 0x861e9fa3f7d7d264)

public:
 virtual void CallMe(); // chose to override MyExtension's impl
 virtual void Foo();
 virtual void Bar();
};

// MyExtensionCustomized.cpp
#include "MyExtensionCustomized.h"

CRYREGISTER_CLASS(CMyExtensionCustomized)

CMyExtensionCustomized::CMyExtensionCustomized()
{
}

CMyExtensionCustomized::~CMyExtensionCustomized()
{
}

void CMyExtensionCustomized::CallMe()
{
 printf("Inside CMyExtensionCustomized::CallMe()...");
}

void CMyExtensionCustomized::Foo()
{
```

```
 printf("Inside CMyExtensionCustomized::Foo()...");
}

void CMyExtensionCustomized::Bar()
{
 printf("Inside CMyExtensionCustomized::Bar()...");
}
```

Without Using Glue Code

If for any reason using the glue code is neither desired nor applicable, extensions can be implemented as follows. It is recommended to implement `ICryUnknown` and `ICryFactory` such that their runtime cost is equal to the one provided by the glue code. For more information, see ICryUnknown (p. 480) and ICryFactory (p. 481).

```
//////////////////////////////////////////
// public section

// INoMacros.h
#include <CryExtension/ICryUnknown.h>

struct INoMacros : public ICryUnknown
{
 // befriend cryiidof and boost::checked_delete
 template <class T> friend const CryInterfaceID& InterfaceCastSemantics::cryiidof();
 template <class T> friend void boost::checked_delete(T* x);
protected:
 virtual ~INoMacros() {}

private:
 // It's very important that this static function is implemented for each interface!
 // Otherwise the consistency of cryinterface_cast<T>() is compromised because
 // cryiidof<T>() = cryiidof<baseof<T>>() {baseof<T> = ICryUnknown in most cases}
 static const CryInterfaceID& IID()
 {
  static const CryInterfaceID iid = {0xd0fda1427dee4cceull, 0x88ff91b6b7be2a1full};
  return iid;
 }

public:
 virtual void TellMeWhyIDontLikeMacros() = 0;
};

typedef boost::shared_ptr<INoMacros> INoMacrosPtr;

//////////////////////////////////////////
// private section not visible to client

// NoMacros.cpp
//
// This is just an exemplary implementation!
// For brevity the whole implementation is packed into this cpp file.

#include <INoMacros.h>
#include <CryExtension/ICryFactory.h>
#include <CryExtension/Impl/RegFactoryNode.h>

// implement factory first
class CNoMacrosFactory : public ICryFactory
{
 // ICryFactory
public:
 virtual const char* GetClassName() const
```

```
 {
  return "NoMacros";
 }
 virtual const CryClassID& GetClassID() const
 {
  static const CryClassID cid = {0xa4550317690145c1ull, 0xa7eb5d85403dfad4ull};
  return cid;
 }
 virtual bool ClassSupports(const CryInterfaceID& iid) const
 {
  return iid == cryiidof<ICryUnknown>() || iid == cryiidof<INoMacros>();
 }
 virtual void ClassSupports(const CryInterfaceID*& pIIDs, size_t& numIIDs) const
 {
  static const CryInterfaceID iids[2] = {cryiidof<ICryUnknown>(), cryiidof<INoMacros>()};
  pIIDs = iids;
  numIIDs = 2;
 }
 virtual ICryUnknownPtr CreateClassInstance() const;

public:
 static CNoMacrosFactory& Access()
 {
  return s_factory;
 }

private:
 CNoMacrosFactory() {}
 ~CNoMacrosFactory() {}

private:
 static CNoMacrosFactory s_factory;
};

CNoMacrosFactory CNoMacrosFactory::s_factory;

// implement extension class
class CNoMacros : public INoMacros
{
 // ICryUnknown
public:
 virtual ICryFactory* GetFactory() const
 {
  return &CNoMacrosFactory::Access();
 };

 // befriend boost::checked_delete
 // only needed to be able to create initial shared_ptr<CNoMacros>
 // so we don't lose type info for debugging (i.e. inspecting shared_ptr)
 template <class T> friend void boost::checked_delete(T* x);

protected:
 virtual void* QueryInterface(const CryInterfaceID& iid) const
 {
  if (iid == cryiidof<ICryUnknown>())
   return (void*) (ICryUnknown*) this;
  else if (iid == cryiidof<INoMacros>())
   return (void*) (INoMacros*) this;
  else
   return 0;
 }

 virtual void* QueryComposite(const char* name) const
 {
  return 0;
 }
```

```
 // INoMacros
public:
 virtual void TellMeWhyIDontLikeMacros()
 {
  printf("Woohoo, no macros...\n");
 }

 CNoMacros() {}

protected:
 virtual ~CNoMacros() {}
};

// implement factory's CreateClassInstance method now that extension class is fully visible
 to compiler
ICryUnknownPtr CNoMacrosFactory::CreateClassInstance() const
{
 boost::shared_ptr<CNoMacros> p(new CDontLikeMacros);
 return
 ICryUnknownPtr(*static_cast<boost::shared_ptr<ICryUnknown>*>(static_cast<void*>(&p)));
}

// register extension
static SRegFactoryNode g_noMacrosFactory(&CNoMacrosFactory::Access());
```

Exposing Composites

The following example shows how to expose (inherited) composites. For brevity, the sample is not separated into files.

```
//////////////////////////////////////////////////////////////////////////
struct ITestExt1 : public ICryUnknown
{
 CRYINTERFACE_DECLARE(ITestExt1, 0x9d9e0dcfa5764cb0, 0xa73701595f75bd32)

 virtual void Call1() = 0;
};

typedef boost::shared_ptr<ITestExt1> ITestExt1Ptr;

class CTestExt1 : public ITestExt1
{
 CRYINTERFACE_BEGIN()
  CRYINTERFACE_ADD(ITestExt1)
 CRYINTERFACE_END()

 CRYGENERATE_CLASS(CTestExt1, "TestExt1", 0x43b04e7cc1be45ca, 0x9df6ccb1c0dc1ad8)

public:
 virtual void Call1();
};

CRYREGISTER_CLASS(CTestExt1)

CTestExt1::CTestExt1()
{
}

CTestExt1::~CTestExt1()
{
}

void CTestExt1::Call1()
```

```
{
}

//////////////////////////////////////////////////////////////////////////
class CComposed : public ICryUnknown
{
 CRYINTERFACE_BEGIN()
 CRYINTERFACE_END()

 CRYCOMPOSITE_BEGIN()
  CRYCOMPOSITE_ADD(m_pTestExt1, "Ext1")
 CRYCOMPOSITE_END(CComposed)

 CRYGENERATE_CLASS(CComposed, "Composed", 0x0439d74b8dcd4b7f, 0x9287dcdf7e26a3a5)

private:
 ITestExt1Ptr m_pTestExt1;
};

CRYREGISTER_CLASS(CComposed)

CComposed::CComposed()
: m_pTestExt1()
{
 CryCreateClassInstance("TestExt1", m_pTestExt1);
}

CComposed::~CComposed()
{
}

//////////////////////////////////////////////////////////////////////////
struct ITestExt2 : public ICryUnknown
{
 CRYINTERFACE_DECLARE(ITestExt2, 0x8eb7a4b399874b9c, 0xb96bd6da7a8c72f9)

 virtual void Call2() = 0;
};

DECLARE_BOOST_POINTERS(ITestExt2);

class CTestExt2 : public ITestExt2
{
 CRYINTERFACE_BEGIN()
  CRYINTERFACE_ADD(ITestExt2)
 CRYINTERFACE_END()

 CRYGENERATE_CLASS(CTestExt2, "TestExt2", 0x25b3ebf8f1754b9a, 0xb5494e3da7cdd80f)

public:
 virtual void Call2();
};

CRYREGISTER_CLASS(CTestExt2)

CTestExt2::CTestExt2()
{
}

CTestExt2::~CTestExt2()
{
}

void CTestExt2::Call2()
{
}
```

```
//////////////////////////////////////////////////////////////////////////
class CMultiComposed : public CComposed
{
 CRYCOMPOSITE_BEGIN()
  CRYCOMPOSITE_ADD(m_pTestExt2, "Ext2")
 CRYCOMPOSITE_ENDWITHBASE(CMultiComposed, CComposed)

 CRYGENERATE_CLASS(CMultiComposed, "MultiComposed", 0x0419d74b8dcd4b7e,
0x9287dcdf7e26a3a6)

private:
 ITestExt2Ptr m_pTestExt2;
};

CRYREGISTER_CLASS(CMultiComposed)

CMultiComposed::CMultiComposed()
: m_pTestExt2()
{
 CryCreateClassInstance("TestExt2", m_pTestExt2);
}

CMultiComposed::~CMultiComposed()
{
}

...

//////////////////////////////////////////////////////////////////////////
// let's use it

ICryUnknownPtr p;
if (CryCreateClassInstance("MultiComposed", p))
{
 ITestExt1Ptr p1 = cryinterface_cast<ITestExt1>(crycomposite_query(p, "Ext1"));
 if (p1)
  p1->Call1(); // calls CTestExt1::Call1()
 ITestExt2Ptr p2 = cryinterface_cast<ITestExt2>(crycomposite_query(p, "Ext2"));
 if (p2)
  p2->Call2(); // calls CTestExt2::Call2()
}
```

If ICryUnknown Cannot Be the Base of the Extension Class

There are cases where making `ICryUnknown` the base of your extension class is not possible. Some examples are legacy code bases that cannot be modified, third party code for which you do not have full source code access, or code whose modification is not practical. However, these code bases can provide useful functionality (for example, for video playback or flash playback) if you expose them as engine extensions. The following sample illustrates how an additional level of indirection can expose a third party API.

```
//////////////////////////////////////////
// public section

// IExposeThirdPartyAPI.h
#include <CryExtension/ICryUnknown.h>
#include <IThirdPartyAPI.h>

struct IExposeThirdPartyAPI : public ICryUnknown
{
 CRYINTERFACE_DECLARE(IExposeThirdPartyAPI, 0x804250bbaacf4a5f, 0x90ef0327bb7a0a7f)
```

```
 virtual IThirdPartyAPI* Create() = 0;
};

typedef boost::shared_ptr<IExposeThirdPartyAPI> IExposeThirdPartyAPIPtr;

//////////////////////////////////////////
// private section not visible to client

// Expose3rdPartyAPI.h
#include <IExposeThirdPartyAPI.h>
#include <CryExtension/Impl/ClassWeaver.h>

class CExposeThirdPartyAPI : public IExposeThirdPartyAPI
{
 CRYINTERFACE_BEGIN()
  CRYINTERFACE_ADD(IExposeThirdPartyAPI)
 CRYINTERFACE_END()

 CRYGENERATE_CLASS(CExposeThirdPartyAPI, "ExposeThirdPartyAPI", 0xa93b970b2c434a21,
 0x86acfe94d8dae547)

public:
 virtual IThirdPartyAPI* Create();
};

// ExposeThirdPartyAPI.cpp
#include "ExposeThirdPartyAPI.h"
#include "ThirdPartyAPI.h"

CRYREGISTER_CLASS(CExposeThirdPartyAPI)

CExposeThirdPartyAPI::CExposeThirdPartyAPI()
{
}

CExposeThirdPartyAPI::~CExposeThirdPartyAPI()
{
}

IThirdPartyAPI* CExposeThirdPartyAPI::Create()
{
 return new CThirdPartyAPI; // CThirdPartyAPI implements IThirdPartyAPI
}
```

Custom Inclusion and Exclusion of Extensions

To enable easy inclusion and exclusion of extensions, Lumberyard provides a global "extension definition" header much like `CryCommon/ProjectDefines.h` that is automatically included in all modules by means of the `platform.h` file. To wrap your extension implementation code, you include a `#define` statement in the extension definition header. To exclude unused extension code from your build, you can also comment out extensions that you are not interested in. Interface headers are not affected by the `#if defined` statements, so the client code compiles as is with or without them.

```
//////////////////////////////////////////
// public section

// IMyExtension.h
#include <CryExtension/ICryUnknown.h>

struct IMyExtension : public ICryUnknown
{
 ...
```

```
};

typedef boost::shared_ptr<IMyExtension> IMyExtensionPtr;

// ExtensionDefines.h
...
#define INCLUDE_MYEXTENSION
...

//////////////////////////////////////////
// private section not visible to client

// MyExtension.h
#if defined(INCLUDE_MYEXTENSION)

#include <IMyExtension.h>
#include <CryExtension/Impl/ClassWeaver.h>

class CMyExtension : public IMyExtension
{
 ...
};

#endif // #if defined(INCLUDE_MYEXTENSION)

// MyExtension.cpp
#if defined(INCLUDE_MYEXTENSION)

#include "MyExtension.h"

CRYREGISTER_CLASS(CMyExtension)

...

#endif // #if defined(INCLUDE_MYEXTENSION)
```

Because extensions can be removed from a build, clients must write their code in a way that does not assume the availability of an extension. For more information, see Using Extensions (p. 490).

CryString

Lumberyard has a custom reference-counted string class `CryString` (declared in `CryString.h`) which is a replacement for `STL std::string`. `CryString` should always be preferred over `std::string`. For convenience, `string` is used as a typedef for `CryString`.

How to Use Strings as Key Values for STL Containers

The following code shows good (efficient) and bad usage:

```
const char *szKey= "Test";

map< string, int >::const_iterator iter = m_values.find( CONST_TEMP_STRING( szKey ) );   //
 Good

map< string, int >::const_iterator iter = m_values.find( szKey );  // Bad
```

By using the suggested method, you avoid the allocation, deallocation, and copying of a temporary string object, which is a common problem for most string classes. By using the macro `CONST_TEMP_STRING`, the string class uses the pointer directly without having to free data afterwards.

Further Usage Tips

- Do not use `std::string` or `std::wstring`. Instead, use only `string` and `wstring`, and never include the standard string header `<string>`.
- Use the `c_str()` method to access the contents of the string.
- Because strings are reference-counted, never modify memory returned by the `c_str()` method. Doing so could affect the wrong string instance.
- Do not pass strings via abstract interfaces; all interfaces should use `const char*` in interface methods.
- `CryString` has a combined interface of `std::string` and the MFC `CString`, so you can use both interface types for string operations.
- Avoid doing many string operations at runtime as they often cause memory reallocations.
- For fixed size strings (e.g. 256 chars), use `CryFixedStringT`, which should be preferred over static `char` arrays.

ICrySizer

The `ICrySizer` interface can be implemented to record detailed information about the memory usage of a class.

Note
This information is also available in the Editor under **Engine Memory info**.

How to use the ICrySizer interface

The following example shows how to use the `ICrySizer` interface.

```
void GetMemoryUsage( ICrySizer *pSizer )
{
 {
    SIZER_COMPONENT_NAME( pSizer, "Renderer (Aux Geometries)" );
    pSizer->Add(*this);
 }
 pSizer->AddObject(<element_prow>,<element_count>);
 pSizer->AddObject(<container>);
 m_SubObject.GetMemoryUsage(pSizer);
}
```

Serialization Library

The `CryCommon` serialization library has the following features:

- Separation of user serialization code from the actual storage format. This makes it possible to switch between XML, JSON, and binary formats without changing user code.
- Re-usage of the same serialization code for editing in the PropertyTree. You can write the serialization code once and use it to expose your structure in the editor as a parameters tree.
- Enables you to write serialization code in non-intrusive way (as global overloaded functions) without modifying serialized types.
- Makes it easy to change formats. For example, you can add, remove, or rename fields and still be able to load existing data.

Tutorial

The example starts with a data layout that uses standard types, enumerations, and containers. The example adds the `Serialize` method to structures with fixed signatures.

Defining data

```
#include "Serialization/IArchive.h"
#include "Serialization/STL.h"

enum AttachmentType
{
 ATTACHMENT_SKIN,
 ATTACHMENT_BONE
};
struct Attachment
{
 string name;
 AttachmentType type;
 string model;
 void Serialize(Serialization::IArchive& ar)
 {
  ar(name, "name", "Name");
  ar(type, "type", "Type");
  ar(model, "model", "Model");
 }
};
struct Actor
{
 string character;
 float speed;
 bool alive;
 std::vector<Attachment> attachments;
 Actor()
 : speed(1.0f)
 , alive(true)
 {
 }
 void Serialize(Serialization::IArchive& ar)
 {
  ar(character, "character", "Character");
  ar(speed, "speed", "Speed");
  ar(alive, "alive", "Alive");
  ar(attachments, "attachments", "Attachment");
 }
};

// Implementation file:
#include "Serialization/Enum.h"

SERIALIZATION_ENUM_BEGIN(AttachmentType, "Attachment Type")
SERIALIZATION_ENUM(ATTACHMENT_BONE, "bone", "Bone")
SERIALIZATION_ENUM(ATTACHMENT_SKIN, "skin", "Skin")
SERIALIZATION_ENUM_END()
```

Why are two names needed?

The `ar()` call takes two string arguments: one is called `name`, and the second `label`. The `name` argument is used to store parameters persistently; for example, for JSON and XML. The `label` parameter is used for the PropertyTree. The `label` parameter is typically longer, more descriptive, contains white space, and may be easily changed without breaking compatibility with existing data. In contrast, `name` is a C-style

identifier. It is also convenient to have `name` match the variable name so that developers can easily find the variable by looking at the data file.

Omitting the `label` parameter (the equivalent of passing `nullptr`) will hide the parameter in the PropertyTree, but it will be still serialized and can be copied together with its parent by using copy-paste.

Note
The `SERIALIZATION_ENUM` macros should reside in the .cpp implementation file because they contain symbol definitions.

Serializing into or from a file

Now that the data has been defined, it is ready for serialization. To implement the serialization, you can use `Serialization::SaveJsonFile`, as in the following example.

```
#include <Serialization/IArchiveHost.h>


Actor actor;
Serialization::SaveJsonFile("filename.json", actor);
```

This will output content in the following format:

```
{
 "character": "nanosuit.cdf",
 "speed": 2.5,
 "alive": true,
 "attachments": [
  { "name": "attachment 1", "type": "bone", "model": "model1.cgf" },
  { "name": "attachment 2", "type": "skin", "model": "model2.cgf" }
 ]
}
```

The code for reading data is similar to that for serialization, except that it uses `Serialization::LoadJsonFile`.

```
#include <Serialization/IArchiveHost.h>

Actor actor;
Serialization::LoadJsonFile(actor, "filename.json");
```

The save and load functions used are wrappers around the `IArchiveHost` interface, an instance of which is located in `gEnv->pSystem->GetArchiveHost()`. However, if you have direct access to the archive code (for example, in `CrySystem` or `EditorCommon`), you can use the archive classes directly, as in the following example.

```
#include <Serialization/JSONOArchive.h>
#include <Serialization/JSONIArchive.h>

Serialization::JSONOArchive oa;

Actor actor;
oa(actor);
oa.save("filename.json");

// to get access to the data without saving:
const char* jsonString = oa.c_str();

// and to load
Serialization::JSONIArchive ia;
```

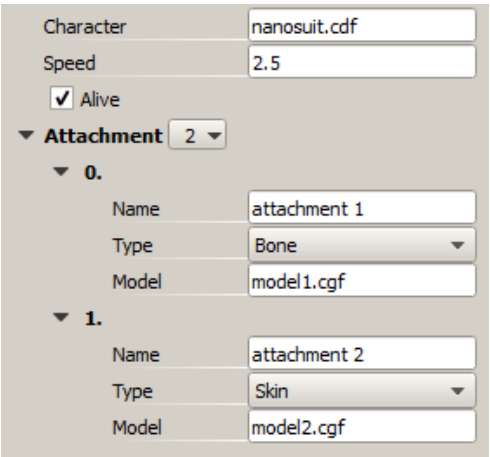
```
if (ia.load("filename.json"))
{
 Actor loadedActor;
 ia(loadedActor);
}
```

Editing in the PropertyTree

If you have the `Serialize` method implemented for your types, it is easy to get it exposed to the `QPropertyTree`, as the following example shows.

```
#include <QPropertyTree/QPropertyTree.h>

QPropertyTree* tree = new QPropertyTree(parent);

static Actor actor;
tree->attach(Serialization::SStruct(actor));
```

You can select enumeration values from the list and add or remove vector elements by using the [2] button or the context menu.



In the moment of attachment, the `Serialize` method will be called to extract properties from your object. As soon as the user changes a property in the UI, the `Serialize` method is called to write properties back to the object.

> **Note**
> It is important to remember that `QPropertyTree` holds a reference to an attached object. If the object's lifetime is shorter than the tree, an explicit call to `QPropertyTree::detach()` should be performed.

Use Cases

Non-intrusive serialization

Normally when `struct` or a class instance is passed to the archive, the `Serialize` method of the instance is called. However, it is possible to override this behavior by declaring the following global function:

```
bool Serialize(Serialization::IArchive&, Type& value, const char* name, const char* label);
```

The return value here has the same behavior as `IArchive::operator()`. For input archives, the function returns false when a field is missing or wasn't read. For output archives, it always returns true.

Note
The return value does not propagate up. If one of the nested fields is missing, the top level block will still return true.

The global function approach is useful when you want to:

- Add serialization in non-intrusive way
- Transform data during serialization
- Add support for unsupported types like plain pointers

The following example adds support for `std::pair<>` type to the `Serialize` function:

```
template<class T1, class T2>
struct pair_serializable : std::pair<T1, T2>
{
  void Serialize(Serialization::IArchive& ar)
  {
    ar(first, "first", "First");
    ar(second, "second", "Second");
  }
}

template<class T1, class T2>
bool Serialize(Serialization::IArchive& ar, std::pair<T1, T2>& value, const char* name,
 const char* label)
{
  return ar(static_cast<pair_serializable<T1, T2>&>(value), name, label);
}
```

The benefit of using inheritance is that you can get access to protected fields. In cases when access policy is not important and inheritance is undesirable, you can replace the previous code with following pattern.

```
template<class T1, class T2>
struct pair_serializable
{
  std::pair<T1, T2>& instance;

  pair_serializable(std::pair<T1, T2>& instance) : instance(instance) {}
  void Serialize(Serialization::IArchive& ar)
  {
    ar(instance.first, "first", "First");
    ar(instance.second, "second", "Second");
  }
}

template<class T1, class T2>
bool Serialize(Serialization::IArchive& ar, std::pair<T1, T2>& value, const char* name,
 const char* label)
{
  pair_serializable<T1, T2> serializer(value);
  return ar(serializer, name, label);
}
```

Registering Enum inside a Class

Normally, `SERIALIZATION_ENUM_BEGIN()` will not compile if you specify enumeration within a class (a "nested `enum`"). To overcome this shortcoming, use `SERIALIZATION_ENUM_BEGIN_NESTED`, as in the following example.

```
SERIALIZATION_ENUM_BEGIN_NESTED(Class, Enum, "Label")
SERIALIZATION_ENUM(Class::ENUM_VALUE1, "value1", "Value 1")
SERIALIZATION_ENUM(Class::ENUM_VALUE2, "value2", "Value 2")
SERIALIZATION_ENUM_END()
```

Polymorphic Types

The Serialization library supports the loading and saving of polymorphic types. This is implemented through serialization of a smart pointer to the base type.

For example, if you have following hierarchy:

IBase

- ImplementationA
- ImplementationB

You would need to register derived types with a macro, as in the following example.

```
SERIALIZATION_CLASS_NAME(IBase, ImplementationA, "impl_a", "Implementation A");
SERIALIZATION_CLASS_NAME(IBase, ImplementationA, "impl_b", "Implementation B");
```

Now you can serialize a pointer to the base type:

```
#include <Serialization/SmartPtr.h>

_smart_ptr<IInterfface> pointer;

ar(pointer, "pointer", "Pointer");
```

The first string is used to name the type for persistent storage, and the second string is a human-readable name for display in the PropertyTree.

Customizing presentation in the PropertyTree

There are two aspects that can be customized within the PropertyTree:

1. The layout of the property fields. These are controlled by control sequences in the label (the third argument in `IArchive::operator()`).
2. Decorators. These are defined in the same way that specific properties are edited or represented.

Control characters

Control sequences are added as a prefix to the third argument for `IArchive::operator()`. These characters control the layout of the property field in the PropertyTree.

Layout Control Characters

Prefix	Role	Description
!	Read-only field	Prevents the user from changing the value of the property. The effect is non-recursive.

Prefix	Role	Description
^	Inline	Places the property on the same line as the name of the structure root. Can be used to put fields in one line in a horizontal layout, rather than in the default vertical list.
^^	Inline in front of a name	Places the property name before the name of the parent structure. Useful to add check boxes before a name.
<	Expand value field	Expand the value part of the property to occupy all available space.
>	Contract value field	Reduces the width of the value field to the minimum. Useful to restrict the width of inline fields.
>*N*>	Limit field width to *N* pixels	Useful for finer control over the UI. Not recommended for use outside of the editor.
+	Expand row by default.	Can be used to control which structures or containers are expanded by default. Use this only when you need per-item control. Otherwise, `QPropertyTree::setExpandLevels` is a better option.
[*S*]	Apply *S* control characters to children.	Applies control characters to child properties. Especially useful with containers.

Combining control characters

Multiple control characters can be put together to combine their effects, as in the following example.

```
ar(name, "name", "^!<Name"); // inline, read-only, expanded value field
```

Decorators

There are two kinds of decorators:

1. Wrappers that implement a custom serialization function that performs a transformation on the original value. For example, `Serialization/Math.h` contains `Serialization::RadiansAsDeg(float&)` that allows to store and edit angles in radians.
2. Wrappers that do no transformation but whose type is used to select a custom property implementation in the PropertyTree. Resource Selectors are examples of this kind of wrapper.

Decorator	Purpose	Defined for types	Context needed

`AnimationPath`	Selection UI for full animation path.	Any string-like type, like:
		`std::string`,
		`string (CryStringT)`,
		`SCRCRef`
		`CCryName`	

Decorator	Purpose	Defined for types	Context needed
`CharacterPath`	UI: browse for character path (cdf)		
`CharacterPhysicsPath`	UI: browse for character .phys-file.		
`CharacterRigPath`	UI: browse for .rig files.		
`SkeletonPath`	UI: browse for .chr or .skel files.		
`JointName`	UI: list of character joints	`ICharacterInstance*`	
`AttachmentName`	UI: list of character attachments	`ICharacterInstance*`	
`SoundName`	UI: list of sounds		
`ParticleName`	UI: particle effect selection		
`Serialization/ Decorators/Math.h`			
`RadiansAsDeg`	Edit or store radians as degrees	float, Vec3	
`Serialization/ Decorators/Range.h`			
`Range`	Sets soft or hard limits for numeric values and provides a slider UI.	Numeric types	
`Serialization/ Callback.h`			
`Callback`	Provides per-property callback function. See Adding callbacks to the PropertyTree (p. 510).	All types apart from compound ones (structs and containers)	

Decorator example

The following example uses the `Range` and `CharacterPath` decorators.

```
float scalar;
ar(Serialization::Range(scalar), 0.0f, 1.0f); // provides slider-UI
string filename;
ar(Serialization::CharacterPath(filename), "character", "Character"); // provides UI for
 file selection with character filter
```

Serialization context

The signature of the `Serialize` method is fixed. This can prevent the passing of additional arguments into nested `Serialize` methods. To resolve this issue, you can use a serialization context to pass a pointer of a specific type to nested `Serialize` calls, as in the following example.

```
void Scene::Serialize(Serialization::IArchive& ar)
{
  Serialization::SContext sceneContext(ar, this);
  ar(rootNode, "rootNode")
}

void Node::Serialize(Serialization::IArchive& ar)
{
  if (Scene* scene = ar.FindContext<Scene>())
  {
    // use scene
  }
}
```

Contexts are organized into linked lists. Nodes are stored on the stack within the `SContext` instance.

You can have multiple contexts. If you provide multiple instances of the same type, the innermost context will be retrieved.

You may also use contexts with the PropertyTree without modifying existing serialization code. The easiest way to do this is to use `CContextList` (`QPropertyTree/ContextList.h`), as in the following example.

```
// CContextList m_contextList;
tree = new QPropertyTree();
m_contextList.Update<Scene>(m_scenePointer);
tree->setArchiveContext(m_contextList.Tail());
tree->attach(Serialization::SStruct(node));
```

Serializing opaque data blocks

It is possible to treat a block of data in the archive in an opaque way. This capability enables the Editor to work with data formats it has no knowledge of.

These data blocks can be stored within `Serialization::SBlackBox`. `SBlackBox` can be serialized or deserialized as any other value. However, when you deserialize `SBlackBox` from a particular kind of archive, you must serialize by using a corresponding archive. For example, if you obtained your `SBlackBox` from `JSONIArchive`, you must save it by using `JSONOArchive`.

Adding callbacks to the PropertyTree

When you change a single property within the property tree, the whole attached object gets de-serialized. This means that all properties are updated even if only one was changed. This approach may seem wasteful, but has the following advantages:

- It removes the need to track the lifetime of nested properties, and the requirement that nested types be referenced from outside in safe manner.
- The content of the property tree is not static data, but rather the result of the function invocation. This allows the content to be completely dynamic. Because you do not have to track property lifetimes, you can serialize and de-serialize variables constructed on the stack.
- The removal of the tracking requirement results in a smaller amount of code.

Nevertheless, there are situations when it is desirable to know exactly which property changes. You can achieve this in two ways: 1) by using the `Serialize` method, or 2) by using `Serialization::Callback`.

1. Using the `Serialize` method, compare the new value with the previous value, as in the following example.

```
void Type::Serialize(IArchive& ar)
{
  float oldValue = value;
  ar(value, "value", "Value");
  if (ar.IsInput() && oldValue != value)
  {
 // handle change
  }
}
```

2. The second option is to use the `Serialization::Callback` decorator to add a callback function for one or more properties, as the following example illustrates.

```
#include <Serialization/Callback.h>
using Serialization::Callback;

ar(Callback(value,
            [](float newValue) { /* handle change */ }),
   "value", "Value");
```

   Note
   `Callback` works only with the PropertyTree, and should be used only in Editor code.

   `Callback` can also be used together with other decorators, but in rather clumsy way, as the following example shows.

```
ar(Callback(value,
            [](float newValue) { /* handle change*/ },
            [](float& v) { return Range(v, 0.0f, 1.0f); }),
   "value", "Value");
```

Of the two approaches, the callback approach is more flexible, but it requires you to carefully track the lifetime of the objects that are used by the callback lambda or function.

PropertyTree in MFC window

If your code base still uses MFC, you can use the PropertyTree with it by using a wrapper that makes this possible, as the following example shows.

```
#include <IPropertyTree.h> // located in Editor/Include

int CMyWindow::OnCreate(LPCREATESTRUCT pCreateStruct)
{
  ...
  CRect clientRect;
  GetClientRect(clientRect);
  IPropertyTree* pPropertyTree = CreatePropertyTree(this, clientRect);
  ...
}
```

The `IPropertyTree` interface exposes the methods of `QPropertyTree` like `Attach`, `Detach` and `SetExpandLevels`.

Documentation and validation

`QPropertyTree` provides a way to add short documentation in the form of tool tips and basic validation.

The `Doc` method allows you to add tool tips to `QPropertyTree`, as in the following examples.

```
void IArchive::Doc(const char*)
```

```
void SProjectileParameter::Serialize(IArchive& ar)
{
  ar.Doc("Defines projectile physics.");

  ar(m_velocity, "velocity", "Velocity");
  ar.Doc("Defines initial velocity of the projectile.");
}
```

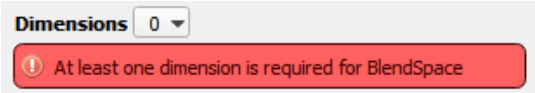
The `Doc` method adds a tool tip to last serialized element. When used at the beginning of the function, it adds the tool tip to the whole block.

The `Warning` and `Error` calls allow you to display warnings and error messages associated with specific property within the property tree, as in the following examples.

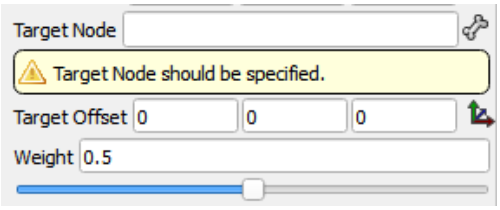
```
template<class T> void IArchive::Warning(T& instance, const char* format, ...)
template<class T> void IArchive::Error(T& instance, const char* format, ...)
```

```
void BlendSpace::Serialize(IArchive& ar)
{
 ar(m_dimensions, "dimensions, "Dimensions");
 if (m_dimensions.empty())
  ar.Error(m_dimensions, "At least one dimension is required for BlendSpace");
}
```

The error message appears as follows.



Warning messages look like this:



Drop-down menu with a dynamic list

If you want to specify an enumeration value, you can use the `enum` registration macro as described in the Defining data (p. 503) section.

There are two ways to define a drop-down menu: 1) transform your data into `Serialization::StringListValue`, or 2) implement a custom PropertyRow in the UI.

A short example of the first approach follows. The example uses a custom reference.

```
// a little decorator that would annotate string as a special reference
struct MyReference
{
 string& str;
 MyReference(string& str) : str(str) {}
};

inline bool Serialize(Serialization::IArchive& ar, MyReference& wrapper, const char* name,
 const char* label)
{
 if (ar.IsEdit())
 {
  Serialization::StringList items;
  items.push_back("");
  items.push_back("Item 1");
  items.push_back("Item 2");
  items.push_back("Item 3");
  Serialization::StringListValue dropDown(items, wrapper.str.c_str());
  if (!ar(dropDown, name, label))
   return false;
  if (ar.IsInput())
   wrapper.str = dropDown.c_str();
  return true;
 }
 else
 {
        // when loading from disk we are interested only in the string
  return ar(wrapper.str, name, label);
 }
}
```

Now you can construct `MyReference` on the stack within the `Serialize` method to serialize a string as a dropdown item, as in the following example.

```
struct SType
{
  string m_reference;
  void SType::Serialize(Serialization::IArchive& ar)
  {
 ar(MyReference(m_reference), "reference", "Reference");
  }
};
```

The second way to define a drop-down menu requires that you implement a custom PropertyRow in the UI. This takes more effort, but makes it possible to create the list of possible items entirely within editor code.

Demo and Video Capture

This section contains information on recording videos for benchmarking. Capturing audio and video is also discussed, using either the Perspective view of the Lumberyard Editor or in pure-game mode via the Launcher.

Topics

- Capturing Video and Audio (p. 514)
- Recording Time Demos (p. 518)

Capturing Video and Audio

This tutorial explains how to set up Lumberyard editor (or game) to capture video. Lumberyard outputs video as single frames. If required, it can also output stereo or 5.1 surround sound audio in `.wav` file format. You can edit the output with commonly available video editing software.

Preparation

Before you can start video and audio streams in preparation for capture, you must configure some settings that determine how the video will be captured. You configure these settings by using console commands. To save time, you can create configuration files that execute the necessary commands for you instead of typing the commands directly into the console. Example configuration files are presented later in this topic.

The next sections describe the settings and the console commands that configure them.

Video Settings

Frame Size and Resolution

The height and width of the captured frames in the editor is normally set to the exact view size of your rendered perspective window. To resize the view size, re-scale the perspective window, or right click in the top right of the perspective viewport where the frame size is displayed.

You can also capture higher than rendered images from Lumberyard Editor and Launcher.

The console variables that are now used in conjunction with Capture Frames are:

- `r_CustomResHeight=N` - Specifies the desired frame height in *N* pixels.
- `r_CustomResWidth=M` - Specifies the desired frame width in *M* pixels.

- `r_CustomResMaxSize=P` - Specifies the maximum resolution at which the engine will render the frames in *P* pixels.
- `r_CustomResPreview=R` - Specifies whether or how the preview is displayed in the viewport. Possible values for *R* are:

r_CustomResPreview	Preview status
0	No preview
1	Scaled to match the size of the viewport
2	Cropped to the size of the viewport

Frames Per Second

When deciding the number of frames per second to specify, keep in mind the following:

- NTSC standard video is approximately 30 frames per second, which is a good compromise between quality and file size.
- High quality video can have up to 60 frames per second, but the difference in quality of the increased number of frames is barely noticeable and can take up a lot of file space.
- Video at less than 24 FPS (a cinema standard) will not look smooth.

To specify a fixed frame rate, use the command:

```
t_fixedstep N
```

N specifies the time step. Time step is calculated by using the formula

```
step = 1 second/<number of frames>
```

A table of common time step values follows.

FPS	Time Step
25 (PAL)	0.04
30	0.033333333
60	0.0166666667

Video Capture File Format

You can capture pictures in several different file formats. A good choice for average quality is the `.jpeg` file format. The `.tga` or `.bmp` file formats are better for higher quality, and `.hdr` for pictures that use high-dynamic-range imaging.

To specify the capture file format, use the console command

```
capture_file_format N
```

N is `jpg`, `bmp`, `tga` or `hdr`.

Video Capture File Location

By default, recorded frames are stored in the directory *<root>*`\CaptureOutput`. To specify a custom directory, use the command:

```
capture_folder N
```

N is the name of the custom directory.

> **Warning**
> When you start a recording, the captured frames are placed in the currently specified directory and will overwrite existing files with the same name. To avoid losing work, create a directory for each recording, or move the existing files to another directory before you start.

Starting and Ending the Video Recording

After you have specified the values mentioned in the previous sections, you can start the recording by using the command:

```
capture_frames N
```

Setting *N* to `1` starts the recording, and setting *N* to `0` stops it.

Audio Settings

Before you begin, decide if you require audio in stereo or in 5.1 surround format, and then change your audio settings accordingly in the Windows control panel.

Deactivating the Sound System

After loading the level of your game that you want to capture, you must deactivate the sound system so that you can redirect the sound output to a file. To deactivate the sound system, use the command:

```
#Sound.DeactivateAudioDevice()
```

This redirects the sound output to a `.wav` file in the root directory of the game. The sound will not run in realtime, but be linked precisely to the time step that you set previously.

To write the sound capture, use the command:

```
s_OutputConfig N
```

Setting *N* to `3` activates the non-realtime writing of sound to the `.wav` file. Setting *N* to `0` specifies auto-detection (the default).

Reactivating the Sound System

To reset the sound system use the command:

```
#Sound.ActivateAudioDevice()
```

This creates a `.wav` file in the root directory of the game. The file will continue to be written to until you run the following combination of commands to deactivate the audio device:

```
#Sound.DeactivateAudioDevice()
```

```
s_OutputConfig 0
```

```
#Sound.ActivateAudioDevice()
```

Tip
Although these commands reset the sound system, some sounds won't start until they are correctly triggered again. This applies particularly to looped sounds. To get looped sounds to play, start the recording of video and sound first, and then enter any area that triggers the looped sounds that you want to record.

Configuration Files

Creating Configuration Files

- To ensure that multiple recordings use exactly the same settings, create a configuration file that you can use for each of them. This will ensure that all of your captured files have the same format.

  An example configuration file:

  ```
  sys_spec = 4
  Fixed_time_step 0.0333333333
  Capture_file_format jpg
  Capture_folder myrecording
  r_width 1280
  r_height 800
  ```

  The command `sys_spec = 4` sets the game graphic settings to "very high" to generate the best appearance.
- To speed up the process of starting and stopping the recording, you can create two configuration files: one to start the video, and one to stop it.
  - To start recording, use a config file like the following:

    ```
    #Sound.DeactivateAudioDevice()
    s_OutputConfig 3
    #Sound.ActivateAudioDevice()
    Capture_frames 1
    ```
  - To stop recording, use a config file like the following:

    ```
    Capture_frames 0
    #Sound.DeactivateAudioDevice()
    ```

```
s_OutputConfig 0
#Sound.ActivateAudioDevice()
```

Executing the Config Files

To run the config file, open the console and type the following command:

```
Exec N
```

N is the name of the config file.

Recording Time Demos

Overview

Lumberyard Editor can record and play back player input and camera movement.

> **Note**
> Recording of some player actions such as vehicle movement are not supported.

To use the feature, you must start game mode in Lumberyard Editor and then record in it. To start game mode, press **Ctrl+G** after a level has fully loaded, or load the level in pure-game mode.

Output like the following appears both in the console and in the `timedemo.log` file in the directory corresponding to the level used:

```
TimeDemo Run 131 Finished.
Play Time: 3.96s, Average FPS: 50.48
Min FPS: 0.63 at frame 117, Max FPS: 69.84 at frame 189
Average Tri/Sec: 14037316, Tri/Frame: 278071
Recorded/Played Tris ratio: 0.99
```

Recording Controls

Optional Title

Command	Keystroke	Console Commands
Start Recording	**Ctrl + PrintScreen**	`record`
End Recording	**Ctrl + Break**	`stoprecording`
Start Playback	**Shift + PrintScreen**	`demo`
Stop Playback	**Ctrl + Break**	`stopdemo`

Related Console Variables

- `stopdemo` – Stops playing a time demo.
- `demo` *demoname* – Plays the time demo from the specified file.
- `demo_fixed_timestep` – Specifies the number of updates per second.

- `demo_panoramic` – Uses a panoramic view when playing back the demo.
- `demo_restart_level` *N* – Restarts the level after each loop. Possible values for *N*: 0 = Off; 1 = use quicksave on first playback; 2 = load level start.
- `demo_ai` – Enables or disables AI during the demo.
- `demo_savestats` – Saves level stats at the end of the loop.
- `demo_max_frames` – Specifies the maximum number of frames to save.
- `demo_screenshot_frame` *N* – Makes a screenshot of the specified frame during demo playback. If a negative value for *N* is supplied, takes a screenshot every *N* frame.
- `demo_quit` – Quits the game after the demo run is finished.
- `demo_noinfo` – Disables the information display during the demo playback.
- `demo_scroll_pause` – Enables the use of the **ScrollLock** key to pause demo play and record.
- `demo_num_runs` – Specifies the number of times to loop the demo.
- `demo_profile` – Enables demo profiling.
- `demo_file` – Specifies the time demo file name.

Entity System

The Entity system is currently on a path to deprecation in favor of the Lumberyard Component Entity System (p. 428).

This section covers topics related to the Entity system. Entities are objects, placed inside a level, that players can interact with.

This section includes the following topics:

- Entity Property Prefixes (p. 520)
- Creating a New Entity Class (p. 521)
- Entity Pool System (p. 523)
- Entity ID Explained (p. 532)
- Adding Usable Support on an Entity (p. 532)
- Entity Scripting (p. 533)

Entity Property Prefixes

The Lumberyard Editor supports typed properties where the type is derived from special prefixes in the property name. For a complete list of supported prefixes, refer to the `s_paramTypes` array, defined in `Objects/EntityScript.cpp`. This array maps prefixes to variable types.

The following prefixes are supported by Lumberyard:

```
{ "n",                  IVariable::INT, IVariable::DT_SIMPLE, SCRIPTPARAM_POSITIVE },
{ "i",                  IVariable::INT, IVariable::DT_SIMPLE,0 },
{ "b",                  IVariable::BOOL, IVariable::DT_SIMPLE,0 },
{ "f",                  IVariable::FLOAT, IVariable::DT_SIMPLE,0 },
{ "s",                  IVariable::STRING, IVariable::DT_SIMPLE,0 },

{ "ei",                 IVariable::INT, IVariable::DT_UIENUM,0 },
{ "es",                 IVariable::STRING, IVariable::DT_UIENUM,0 },

{ "shader",             IVariable::STRING, IVariable::DT_SHADER,0 },
{ "clr",                IVariable::VECTOR, IVariable::DT_COLOR,0 },
{ "color",              IVariable::VECTOR, IVariable::DT_COLOR,0 },

{ "vector",             IVariable::VECTOR, IVariable::DT_SIMPLE,0 },

{ "snd",                IVariable::STRING, IVariable::DT_SOUND,0 },
```

```
{ "sound",              IVariable::STRING, IVariable::DT_SOUND,0 },
{ "dialog",             IVariable::STRING, IVariable::DT_DIALOG,0 },

{ "tex",                IVariable::STRING, IVariable::DT_TEXTURE,0 },
{ "texture",            IVariable::STRING, IVariable::DT_TEXTURE,0 },

{ "obj",                IVariable::STRING, IVariable::DT_OBJECT,0 },
{ "object",             IVariable::STRING, IVariable::DT_OBJECT,0 },

{ "file",               IVariable::STRING, IVariable::DT_FILE,0 },
{ "aibehavior",         IVariable::STRING, IVariable::DT_AI_BEHAVIOR,0 },
{ "aicharacter",        IVariable::STRING, IVariable::DT_AI_CHARACTER,0 },
{ "aipfpropertieslist", IVariable::STRING, IVariable::DT_AI_PFPROPERTIESLIST,0 },
{ "aiterritory",        IVariable::STRING, IVariable::DT_AITERRITORY,0 },
{ "aiwave",             IVariable::STRING, IVariable::DT_AIWAVE,0 },

{ "text",               IVariable::STRING, IVariable::DT_LOCAL_STRING,0 },
{ "equip",              IVariable::STRING, IVariable::DT_EQUIP,0 },
{ "reverbpreset",       IVariable::STRING, IVariable::DT_REVERBPRESET,0 },
{ "eaxpreset",          IVariable::STRING, IVariable::DT_REVERBPRESET,0 },

{ "aianchor",           IVariable::STRING, IVariable::DT_AI_ANCHOR,0 },

{ "soclass",            IVariable::STRING, IVariable::DT_SOCLASS,0 },
{ "soclasses",          IVariable::STRING, IVariable::DT_SOCLASSES,0 },
{ "sostate",            IVariable::STRING, IVariable::DT_SOSTATE,0 },
{ "sostates",           IVariable::STRING, IVariable::DT_SOSTATES,0 },
{ "sopattern",          IVariable::STRING, IVariable::DT_SOSTATEPATTERN,0 },
{ "soaction",           IVariable::STRING, IVariable::DT_SOACTION,0 },
{ "sohelper",           IVariable::STRING, IVariable::DT_SOHELPER,0 },
{ "sonavhelper",        IVariable::STRING, IVariable::DT_SONAVHELPER,0 },
{ "soanimhelper",       IVariable::STRING, IVariable::DT_SOANIMHELPER,0 },
{ "soevent",            IVariable::STRING, IVariable::DT_SOEVENT,0 },
{ "sotemplate",         IVariable::STRING, IVariable::DT_SOTEMPLATE,0 },
{ "gametoken",          IVariable::STRING, IVariable::DT_GAMETOKEN, 0 },
{ "seq_",               IVariable::STRING, IVariable::DT_SEQUENCE, 0 },
{ "mission_",           IVariable::STRING, IVariable::DT_MISSIONOBJ, 0 },
```

Creating a New Entity Class

The following example creates an entity class called **Fan**.

- Create a new entity definition file with the extension ".ent", for example "`GameSDK\Entities\Fan.ent`". This file will expose the entity to the engine.

```
<Entity
  Name="Fan"
  Script="Scripts/Entities/Fan.lua"
/>
```

- Create a new Lua script file, for example `GameSDK\Entities\Scripts\Fan.lua`. The Lua file will define the entity logic.

```
Fan = {
  type = "Fan",                                    -- can be useful for scripting

  -- instance member variables
  minrotspeed = 0,
  maxrotspeed = 1300,
```

```
  acceleration = 300,
  currrotspeed = 0,
  changespeed = 0,
  currangle = 0,

  -- following entries become automatically exposed to the editor and serialized (load/
save)
  -- type is defined by the prefix (for more prefix types, search for s_paramTypes in /
Editor/Objects/EntityScript.cpp)
  Properties = {
  bName = 0,                                -- boolean example, 0/1
  fName = 1.2,                              -- float example
  soundName = "",                           -- sound example
  fileModelName = "Objects/box.cgf",        -- file model
  },

  -- optional editor information
  Editor = {
  Model = "Editor/Objects/Particles.cgf",   -- optional 3d object that represents this
 object in editor
  Icon = "Clouds.bmp",                      -- optional 2d icon that represents this
 object in editor
  },
}

-- optional.  Called only once on loading a level.
-- Consider calling self:OnReset(not System.IsEditor()); here
function Fan:OnInit()
  self:SetName( "Fan" );
  self:LoadObject( "Objects/Indoor/Fan.cgf", 0, 0 );
  self:DrawObject( 0, 1 );
end

-- OnReset() is usually called only from the Editor, so we also need OnInit()
-- Note the parameter
function Fan:OnReset(bGameStarts)
end

-- optional.  To start having this callback called, activate the entity:
-- self:Activate(1); -- Turn on OnUpdate() callback
function Fan:OnUpdate(dt)
  if ( self.changespeed == 0 ) then
  self.currrotspeed = self.currrotspeed - System.GetFrameTime() * self.acceleration;
  if ( self.currrotspeed < self.minrotspeed ) then
  self.currrotspeed = self.minrotspeed;
  end
  else
  self.currrotspeed = self.currrotspeed + System.GetFrameTime() * self.acceleration;
  if ( self.currrotspeed > self.maxrotspeed ) then
  self.currrotspeed = self.maxrotspeed;
  end
  end
  self.currangle = self.currangle + System.GetFrameTime() * self.currrotspeed;
  local a = { x=0, y=0, z=-self.currangle };
  self:SetAngles( a );
end

-- optional serialization
function Fan:OnSave(tbl)
 tbl.currangle = self.currangle;
end

-- optional serialization
function Fan:OnLoad(tbl)
 self.currangle = tbl.currangle;
end
```

```
-- optional
function Fan:OnSpawn()
end

-- optional
function Fan:OnDestroy()
end

-- optional
function Fan:OnShutDown()
end

-- optional
function Fan:OnActivate()
  self.changespeed = 1 - self.changespeed;
end
```

Entity Pool System

The topics in this section describe the entity pool system, including how it is implemented, how to register a new entity class to be pooled, and how to debug it. For more information on using entity pools in the Lumberyard Editor, see the Lumberyard User Guide.

This section includes the following topics:

- Entity Pool Definitions (p. 524)
- Entity Pool Creation (p. 526)
- Creating and Destroying Static Entities with Pools (p. 526)
- Creating and Destroying Dynamic Entities with Pools (p. 529)
- Serialization (p. 529)
- Listener/Event Registration (p. 530)
- Debugging Utilities (p. 531)

The following processes must take place when creating an entity pool and preparing it for use. Each of these processes is described in more detail.

1. An entity pool is created by using the information in an entity pool definition.
2. An entity pool is populated with entity containers.
3. An entity pool is validated by testing the entity pool signature of one of the entity containers against the entity pool signature of each `Entity` class mapped to the pool.
4. All entities marked to be created through the pool have an entity pool bookmark created for them.
5. An entity pool bookmark is prepared from or returned to the entity pool, which is mapped to its `Entity` class on demand.

Editor Usage

When running in the Lumberyard Editor, the entity pool system is not fully enabled. All entities are created outside the pools when playing in-game in the Editor. However, all flow node actions with entity pools will still work in the Lumberyard Editor, mimicking the final results that you will see in-game.

> **Note**
> The entity pool listeners `OnEntityPreparedFromPool` and `OnEntityReturnedToPool` are still called in the Editor, even though the entity itself is not removed/reused.

Static versus Dynamic Entities

Entities can be either static or dynamic. A static entity is placed in the Editor and exported with the level. This entity always exists. A property associated with the exported information determines whether it should be pooled (and not created during level load) or instead have an entity pool bookmark made for it. A dynamic entity is created at run-time, usually from game code. The information is constructed at run-time, usually just before it is created, and passed on to the Entity system for handling. This information also indicates whether or not it should go through an entity pool.

Entity Pool Definitions

Entity pools must be defined in the file `\Game\Scripts\Entities\EntityPoolDefinitons.xml`. An entity pool definition is responsible for defining the following:

- the empty class that will be used by entity containers when they're not in use
- the entity classes mapped to the pool
- other properties that describe the pool and how it is used.

In general, a pool is initially filled with a defined number of entity containers; that is, empty `CEntity` classes with all the required entity proxies and game object extensions that are normally created when an entity belonging to an entity class mapped to the definition is fully instantiated. For example, a normal AI entity will have the following entity proxies: sound extension, script extension, render extension, and the game object as its user extension; as its game object extension, it will have the `CPlayer` class. All of these classes are instantiated for each empty `CEntity` instance, and is reused by the entities as they are created from the pool.

The following illustrates an entity pool definition:

EntityPoolDefinitions.xml

```
<Definition name="AI" emptyClass="NullAI" maxSize="16" hasAI="1" defaultBookmarked="0"
 forcedBookmarked="0">
    <Contains>
        <Class>Grunt</Class>
        <Class>Flyer</Class>
    </Contains>
</Definition>
```

Empty Class

The empty class is defined using the `emptyClass` attribute, which takes the name of a valid entity class. The purpose of the empty class is to:

- satisfy the engine's requirement to have an entity class associated with an entity at all times; an empty container is initialized/reused to this entity class
- prepare all needed entity proxies and game object extensions needed for the entities

For example, building on the definition shown in the previous section, you would create an empty class called "NullAI" and register it the same way as the other AI classes above. Then:

1. Declare the entity class and map it to its Lua script via the game factory.

   GameFactory.cpp

```
REGISTER_FACTORY(pFramework, "NullAI", CPlayer, true);
```

2. Create the Lua script for it. View sample code at `\Game\Scripts\Entities\AI\NullAI.lua`.

These steps will allow Lumberyard to see "NullAI" as a valid entity class. In addition, by mapping `CPlayer` to it, you ensure that the correct game object extension is instantiated for the entity containers. The Lua script needs to create all the entity proxies for the entity containers. In the sample code, a render proxy is created, even though we aren't loading an asset model for this entity. For more details, see the discussion of entity pool signatures in Entity Pool Creation (p. 526).

Entity Class Mapping

In an entity pool definition file, the `<Contains>` section should include maps to all the entity classes that an entity must belong to when it is created through this pool. You can map as many as you want by adding a new `<Class>` node within this section. It is important that each entity have the same dynamic class hierarchy as the empty class when fully instantiated. See Debugging Utilities (p. 531)for useful debugging tools to verify that this is the case.

Other Properties

An entity pool definition can define the following additional properties.

name

Unique identity given to an entity pool, useful for debugging purposes. The name should be unique across all definitions.

maxSize

Largest pool size this pool can reach. By default, this is also the number of entity containers created to fill the pool when loading a level. This value can be overwritten for a level by including an `EntityPools.xml` file inside the level's root directory. This file can only be used to decrease the number of entity containers created per pool; it cannot exceed the `maxSize` value defined here. This is useful when you need to reduce the memory footprint of the entity pools per level. The following example file adjusts the size of an AI entity pool to "2".

```
LevelEntityPools.xml

<EntityPools>
    <AI count="2" />
</EntityPools>
```

hasAI

Boolean value that indicates whether or not the entity pool will contain entities that have AI associated with them. It is important to set this property to TRUE if you are pooling entities with AI.

defaultBookmarked

Boolean value that indicates whether or not an entity belonging to one of the entity classes mapped to this pool is flagged as "created through pool" (see Creating and Destroying Static Entities with Pools (p. 526)). This flag determines whether or not, during a level load, an entity pool bookmark is created for the entity instead of being instantiated.

forcedBookmarked

Boolean value that indicates whether or not an entity belonging to one of the entity classes mapped to this pool must be created through the pool. This property overrides an entity's "created through pool" flag (see Creating and Destroying Static Entities with Pools (p. 526)).

Entity Pool Creation

When loading a level, an entity pool is created for each entity pool definition. On creation, the pool is filled with empty containers (instances of `CEntity` using the `emptyClass` attribute value as the entity class. These empty containers come with some expectations that must be satisfied:

- Containers should be minimal in size. This means you should not load any assets or large amounts of data into them. For example, in the sample Lua script (`\Game\Scripts\Entities\AI\NullAI.lua`), the NullAI entity does not define a character model, animation graph, body damage definition, etc.
- Containers should have the same entity proxies and game object extensions created for them as compared to a `CEntity` fully instantiated using each of the mapped entity classes.

Once the pool is created, an entity pool signature is generated using one of the empty containers. An entity pool's signature is a simple container that maps the dynamic class hierarchy of an entity.

One of the functions of the entity pool system is to avoid as much as possible dynamic allocation for delegate classes used by entities. Key examples of these are the entity proxies and game object extensions used by entities. When an entity pool's empty containers are first created, the delegate classes that will be used by the real entities contained in them are also supposed to be created. To ensure that this is the case, the entity pool signature is used. It works as follows:

1. A `TSerialize` writer is created. It is passed to each entity proxy and game object extension that exists in the entity.
2. Each proxy and extension is expected to write some info to the `TSerialize` writer. This information should be unique.
3. Two signatures can then be compared to see if they contain the same written information, verifying they contain the same dynamic class hierarchy.

All of the entity proxies have already been set up to write their information to the `TSerialize` writer. However, if you create a new game object extension (or a new entity proxy), then you will need to set the class up to respond to the Signature helper when needed. To do this, implement the virtual method (Entity Proxy: `GetSignature`; Game Object Extension: `GetEntityPoolSignature`) and write information about the class to the `TSerialize` writer. Generally, all that is needed is to just begin/end a group with the class name. The function should then return TRUE to mark that the signature is valid thus far.

CActor::GetEntityPoolSignature Example

```
bool CActor::GetEntityPoolSignature( TSerialize signature )
{
 signature.BeginGroup("Actor");
 signature.EndGroup();
 return true;
}
```

The section Debugging Utilities (p. 531) discusses how to view the results of entity pool signature tests in order to verify that everything is working as expected.

Creating and Destroying Static Entities with Pools

This topic covers issues related to handling static entities.

Entity Pool Bookmarks

When an entity is marked to be created through the pool, it is not instantiated during the level load process. Instead, an entity pool bookmark is generated for it. The bookmark contains several items:

- Entity ID reserved for the entity, assigned when the level was exported. You will use this entity ID later to tell the system to create the entity.
- Static instanced data that makes the entity unique. This includes the `<EntityInfo>` section from the `mission.xml` file, which contains area information, flow graph information, child/parent links, PropertiesInstance table, etc.
- Serialized state of the entity if it has been returned to the pool in the past. See more details in Serialization (p. 529).

In each entity's `<EntityInfo>` section in the `mission.xml` file (generated when the level is exported from the Editor), there's a `CreatedThroughPool` property. This property can be referenced from the `SEntitySpawnParams` struct. If set to TRUE, the `EntityLoadManager` module will **not** create a `CEntity` instance for the entity. Instead, it will delegate the static instanced data and reserved entity ID to the `EntityPoolManager` to create a bookmark.

```
CEntityLoadManager::ParseEntities

SEntityLoadParams loadParams;
if (ExtractEntityLoadParams(entityNode, loadParams))
{
    if (bEnablePoolUse && loadParams.spawnParams.bCreatedThroughPool)
    {
        CEntityPoolManager *pPoolManager = m_pEntitySystem->GetEntityPoolManager();
        bSuccess = (pPoolManager && pPoolManager->AddPoolBookmark(loadParams));
    }

    // Default to creating the entity
    if (!bSuccess)
    {
        EntityId usingId = 0;
        bSuccess = CreateEntity(loadParams, usingId);
    }
}
```

Preparing a Static Entity

To prepare a static entity, call `IEntityPoolManager::PrepareFromPool`, passing in the entity ID associated with the static entity you want to create. In response, the following execution flow takes place:

1. System determines if the request can be processed in this frame. It will attempt to queue up multiple requests per frame and spread them out. If the parameter `bPrepareNow` is set to TRUE or if no prepare requests have been handled this frame, the request will be handled immediately. Otherwise, it will be added to the queue. Inside `CEntityPoolManager::LoadBookmarkedFromPool`, the EntityLoadManager is requested to create the entity.

   Note
   Note: If this activity is happening in the Editor, the entity will simply have its Enable event called. This will mimic enabling the entity via Flow Graph (unhide it). In this situation, the execution flow skips to the final step.
2. System searches for an entity container (either empty, or still in use) to hold the requested entity. The function `CEntityPoolManager::GetPoolEntity` looks through the active entity pools to find one that contains the entity class of the given static entity. Once the correct pool is found, the container is retrieved from it. The actual order is as follows:
   a. If a `forcedPoolId` (entity ID of one of the empty containers created to populate the pool) is requested, find that entity container and return it.
   b. If no `forcedPoolId` is requested, get an entity container from the inactive set (entity containers not currently in use).

   c. If no inactive containers are available, get one from the active set (entity containers currently in use). This action uses a "weight" value to determine which container to return. A special Lua function in the script is used to request weights for each empty container (`CEntityPoolManager::GetReturnToPoolWeight`). A negative weight means it should not be used at all if possible. The system might pass in an urgent flag, which means the pool is at its maximum size.
   d. If an empty container can still not be found, an urgent flag will be ignored and the system will try to grow the pool. This is only possible if the pool was not created at its maximum size (this happens when the maximum pool size is overridden for a level with a smaller maximum size). In this case, a new entity container is generated, added to the pool, and immediately used.
3. The retrieved entity container, along with the static instanced data and reserved entity ID gathered from its bookmark, is passed on through the function`CEntityLoadManager::CreateEntity`, which begins the Reload process. `CreateEntity` uses the provided entity container instead of creating a new `CEntity` instance. It will handle calling the Reload pipeline on the entity container, and then install all the static instanced data for the prepared static entity. The Reload pipeline is as follows:
   a. The function `CEntity::ReloadEntity` is called on the entity container. The `CEntity` instance will clean itself up internally and begin using the static instanced data of the entity being prepared. The Lua script also performs cleanup using the function `OnBeingReused`.
   b. The Entity system's salt buffer and other internal containers are updated to reflect that this entity container now holds the reserved entity ID and can be retrieved using it.
   c. Entity proxies are prompted to reload using the static instanced data provided. This is done by calling `IEntityProxy::Reload`; each proxy is expected to correctly reset itself with the new data provided. The Script proxy is always the first to be reloaded so that the Lua script can be correctly associated before the other proxies attempt to use it.

      If the game object is being used as the User proxy, all the game object extensions for the container are also prompted to reload. This is done by calling `IGameObjectExtension::ReloadExtension` on all extensions. If this function returns FALSE, the extension will be deleted. Once this is done, `IGameObjectExtension::PostReloadExtension` is called on all extensions. This behavior mimics the Init and PostInit logic. Each extension is expected to correctly reset itself with the new data provided.
4. If any serialized data exists within the bookmark, the entity container is loaded with that data. This ensures that the static entity resumes the state it was in last time it was returned to the pool. This process is skipped if this is the first time the static entity is being prepared.

At this point, calling `CEntity::GetEntity` or `CEntity::FindEntityByName` will return the entity container that is now housing the static entity and its information.

Returning a Static Entity to the Pool

To return a static entity, call the function `IEntityPoolManager::ReturnToPool`. You must pass in the entity ID associated with the static entity. In response, the following execution flow takes place:

1. The function `CEntityPoolManager::ReturnToPool` finds the bookmark and the entity pool containing the current entity container housing the static entity.
2. Depending on the `bSaveState` argument, the `CEntity` instance is (saved) and its serialized information is added to the bookmark. This ensures that if the static entity is prepared again later, it will resume its current state.
3. The entity container goes through the Reload process again. This time, however, the entity container is reloaded using its empty class, effectively removing all references to loaded assets/content and put it back into a minimal state.

At this point, calling `CEntity::GetEntity` or `CEntity::FindEntityByName` to find the static entity will return NULL.

Creating and Destroying Dynamic Entities with Pools

The processes for creating and destroying dynamic entities are similar to those for static entities, which one key exception: dynamic entities have no entity pool bookmarks (at least initially). Because they are not exported in the level, they have no static instanced data associated with them and so no bookmark is made for them.

Creating a Dynamic Entity

As with static entities, creating a dynamic entity with the pool starts with calling `IEntitySystem::SpawnEntity`. Construct an `SEntitySpawnParams` instance to describe its static instanced data. When filling in this struct, set the `bCreatedThroughPool` property to TRUE if you wish to have the entity be created through the pool. In the following example, a vehicle part from the Vehicle system is being spawned through the pool:

```
SEntitySpawnParams spawnParams;
spawnParams.sName = pPartName
spawnParams.pClass = gEnv->pEntitySystem->GetClassRegistry()-
>FindClass("VehiclePartDetached");
spawnParams.nFlags = ENTITY_FLAG_CLIENT_ONLY;

spawnParams.bCreatedThroughPool = true;

IEntity* pSpawnedDebris = gEnv->pEntitySystem->SpawnEntity(spawnParams);
```

Once `SpawnEntity`, the following execution flow takes place:

1. `CEntitySystem::SpawnEntity` will check for an entity pool associated with the provided entity class. If so, it will delegate the workload to the entity pool manager.
2. From within `CEntityPoolManager::PrepareDynamicFromPool`, an entity pool bookmark is created for the new entity. This is done primarily for serialization purposes.
3. The execution flow follows the same sequence as preparing a static entity (see Creating and Destroying Static Entities with Pools (p. 526).
4. If the process is successful, the entity container now housing the information is returned. Otherwise, `SpawnEntity` creates a new `CEntity` instance to satisfy the request.

At this point, calling `CEntity::GetEntity` or `CEntity::FindEntityByName` will return the entity container now housing the dynamic entity and its information.

Destroying a Dynamic Entity with the Pool

As with static entities, use `IEntitySystem::RemoveEntity` or any other method that can destroy an entity. The entity pool manager will return the entity container to the pool, freeing it for use elsewhere and removing the dynamic entity in the process. The resulting execution flow differ from destroying static entities in two ways:

- Dynamic entities are not serialized when they are returned.
- The entity pool bookmark associated with the dynamic entity is removed. It is no longer needed.

At this point, calling `CEntity::GetEntity` or `CEntity::FindEntityByName` will return NULL.

Serialization

All entities created or prepared through the entity pool system are serialized by the system for game save/load. For this reason, do not serialize those entities marked as coming from the

pool (`IEntity:IsFromPool`) in your normal serialization. This is handled in Lumberyard's default implementation for saving and loading the game state.

The entity pool system is serialized from the Entity system's implementation of the `Serialize` function.

Saving Entity Pools

The following process occurs when the game state is being saved:

1. All active entity containers in all entity pools are updated. This results in `CEntityPoolManager::UpdatePoolBookmark` being called for each active entity container. As long as the entity does not have the ENTITY_FLAG_NO_SAVE flag set on it, the bookmark is serialized as follows:
   a. Serialize Helper writes to the bookmark's `pLastState` (an `ISerializedObject`), which contains the serialized state of the entity.
   b. The callback `CEntityPoolManager::OnBookmarkEntitySerialize` runs through the serialization process on the entity. This ensures that the general information, properties and all entity proxies are serialized using their overloaded `Serialize()` implementation.
   c. Any listeners subscribed to the `OnBookmarkEntitySerialize` callback are able to write data into the bookmark at this time. This is used to also bookmark AI objects along with the entity.
2. All entity pool bookmarks are saved, including the static entity and dynamic entity usage counts.
3. If any prepare requests are currently queued, the prepare request queue is saved.

Loading Entity Pools

The following process occurs when the game state is being loaded:

1. The saved entity pool bookmarks are read in. If the bookmark is marked as containing a dynamic entity, it is read to ensure it exists. Each bookmark's `pLastState` is read in and updated.
2. If the entity pool bookmark contains an entity that was active at the time the game was saved, the entity is created/prepared from the pool once more.
   a. While the entity is being created/prepared, it will load its internal state using the `pLastState` at its final step, because the object contains information at this point.
   b. This will also call the `OnBookmarkEntitySerialize` listener callback, allowing other systems to read data from the bookmark.

Listener/Event Registration

There are several listener and various event callbacks dealing with entity pool usage. These callbacks are important for sub-systems that rely on entity registration. They can notify you when an entity has been prepared or returned to the pool so that you can register and unregister it with your subsystems as needed.

IEntityPoolListener

This listener can be subscribed to via `IEntityPoolManager::AddListener`. It contains the following callbacks:

OnPoolBookmarkCreated

Called when an entity pool bookmark has been created. The reserved entity ID for the pooled entity is passed in, along with the static instanced data belonging to it.

OnEntityPreparedFromPool

Called when an entity (static or dynamic) has been prepared from the pool. You are given both the entity ID and the entity container that is now housing the entity. This is called at the end of the prepare entity process.

OnEntityReturnedToPool

Called when an entity (static or dynamic) has been returned to the pool. You are given both the entity ID and the entity container that is currently housing the entity. This is called at the start of the return entity process.

OnPoolDefinitionsLoaded

Called at initialization, with information allowing listeners to set up their own resources for working with the pool. Currently passes the total number of pooled entities that have AI attached.

OnBookmarkEntitySerialize

Called during reads and writes from entity bookmarks, allowing listeners to store additional data in the bookmark.

IEntitySystemSink

This listener has a special callback, OnReused, that notifies you when an entity has been reloaded. This is the process an entity container goes through when a static entity is being prepared into it, or a dynamic entity is being created inside it. You are given the entity container that houses the entity as well as the static instanced data belonging to it.

Debugging Utilities

There are several debugging utilities you can use to manage the entity pools and see how they are being used during gameplay.

Debugging Entity Pool Bookmarks

To see the status of all entity pool bookmarks that currently exist during the game, use the following console command.

```
es_dump_bookmarks [filterName] [dumpToDisk]
```

This command causes text to be written to the console and game log file for every bookmark requested.

Arguments

filterName

(Optional) Allows you to filter your request to get bookmarks only for entities whose names contain the specified value as a substring. To display all bookmarks, set this argument to "all" or leave it empty.

dumpToDisk

(Optional) Allows you to output to disk all static instanced data associated with the displayed bookmarks. If supplied and its a non-zero numerical file, data will be stored at `\User\Bookmarks\LevelName\EntityName.xml`.

Data displayed

The following information is displayed for each bookmark:

- Name of the bookmarked entity.
- Layer the bookmarked entity belongs to.
- Entity class name the bookmarked entity uses.
- Reserved entity ID associated with the bookmarked entity.
- If the bookmarked entity has the No Save Entity Flag associated with it.
- If the bookmarked entity is static or dynamic.
- If the bookmarked entity contains any serialized data (and the memory footprint of the information if available).
- If the bookmarked entity contains any static instanced data (and the memory footprint of the information if available).

Entity ID Explained

When referring to a dynamic C++ object, pointers and reference counting can be used, but a better method is to use a weak reference that allows you to remove an object and have all references become invalid. This option limits the need to iterate over all objects to invalidate objects being removed, which results in performance costs.

With each reference, Lumberyard stores a number called the "salt" (also called a "magic number"). This number, together with the **index**, gives the object a unique entity ID over the game lifetime. Whenever an object is destroyed and the index is reused, the salt is increased and all references with the same index become invalid. To get an entity position/pointer, the entity manager needs to resolve the entity ID; as the salt is different, the method fails.

The class `CSaltBufferArray` handles adding and removing objects and does the required adjustments to the salt. The object array is kept compact for more cache-friendly memory access. Storing `EntityId` references to disc is possible and used for saved games and by the Editor game export. However, when loading a saved game of a level that has been patched and now has more entities, this can result in a severe conflict. To solve this problem, dynamic entities are created starting with a high index counting down, while static entities are created starting with a low index counting up.

Entity IDs have the following limitations:

- A 16-bit index allows up to approximately 65,000 living objects. This should be enough for any non-massive multiplayer game. In a massive multiplayer game, the method described here should not be used by the server. However, it can be used between specific clients and the server.
- A 16-bit salt value allows a game to reuse an index up to approximately 65,000 times. If that happens, the index can no longer be used. This should be enough for any non-massive multiplayer game, when used with some care—don't create and destroy objects (such as bullets) too rapidly. A massive multiplayer game, or any game that supports multi-day game sessions, can run into this limit.

Adding Usable Support on an Entity

Overview

Players may be able to interact with an entity using a key press ('F' by default). Entities that can be interacted with will be enabled with a special on-screen icon inside the game to inform the player that interaction is possible.

To use this feature, you need to create a script that implements two functions: `IsUsable()` and `OnUsed()`.

Preparing the Script

The script should look like this:

```
MakeUsable(NewEntity)

function NewEntity:IsUsable(userId)
  -- code implementation
  return index;
end

function NewEntity:OnUsed(userId, index)
  -- code implementation
end
```

Implementing IsUsable

The `IsUsable()` function is called when a player is aiming the cross-hairs towards the entity. The function will determine if the entity can be interacted with by the player doing the aiming. The function only accepts a single parameter: the player's entity ID.

If the player cannot interact with the entity, the function should return 0. This value causes the UI to not render the "USE" icon over the entity.

If the player can interact with the entity, the function should return a positive value. This value will be stored and later used when calling the `OnUsed()` function.

Implementing OnUsed

The `OnUsed()` function is called when a player presses interacts with the entity (such as by pressing the Use key when the USE icon is active. This function accepts two parameters: (1) the player's entity ID, and (2) the value returned by `IsUsable()`.

Entity Scripting

This section contains topics on using Lua scripting to work with the Entity system.

This section includes the following topics:

- Structure of a Script Entity (p. 533)
- Using Entity State (p. 537)
- Using Entity Slots (p. 538)
- Linking Entities (p. 539)
- Exposing an Entity to the Network (p. 540)

Structure of a Script Entity

To implement a new entity using Lua, two files need to be created and stored in the game directory:

- The Ent file tells the Entity system the location of the Lua script file.

- The Lua script file implements the desired properties and functions.

With the SDK, both the `.ent` and `.lua` files are stored inside the `<Game_Folder>\Scripts.pak` file.

Ent File

The Ent files are all stored inside the `<Game_Folder>\Entities` directory and need to have the `.ent` file extension. The content is XML as follows:

```
<Entity
 Name="LivingEntity"
 Script="Scripts/Entities/Physics/LivingEntity.lua"
/>
```

Entity properties set in the Ent file include:

Name

Name of the entity class.

Script

Path to the Lua script that implements the entity class.

Invisible

Flag indicating whether or not the entity class is visible in Lumberyard Editor.

Lua Script

The Lua script, in addition to implementing the entity class, provides a set of information used by Lumberyard Editor when working with entities on a level. The property values set inside the Lua script are default values assigned to new entity instances. Editor variables specify how entities are drawn in Lumberyard Editor.

The following code excerpt is from the sample project files in your Lumberyard directory (`...\dev\Cache\SamplesProject\pc\samplesproject\scripts\entities\physics\livingentity.lua`).

```
LivingEntity = {
 Properties = {
  soclasses_SmartObjectClass = "",
  bMissionCritical = 0,
  bCanTriggerAreas = 1,
  DmgFactorWhenCollidingAI = 1,

  object_Model = "objects/default/primitive_capsule.cgf",
  Physics = {
   bPhysicalize = 1, -- True if object should be physicalized at all.
   bPushableByPlayers = 1,
  },
  Living = {
   height = 0, -- vertical offset of collision geometry center
   vector_size = {0.4, 0.4,0.9}, -- collision cylinder dimensions
   height_eye = 1.8, -- vertical offset of camera
   height_pivot = 0.1, -- offset from central ground position that is considered entity
 center
   head_radius = 0.3, -- radius of the 'head' geometry (used for camera offset)
   height_head = 1.7, -- center.z of the head geometry
   groundContactEps = 0.004, --the amount that the living needs to move upwards before
 ground contact is lost. defaults to which ever is greater 0.004, or 0.01*geometryHeight
   bUseCapsule = 1,--switches between capsule and cylinder collider geometry
```

```
  inertia = 1, -- inertia koefficient, the more it is, the less inertia is, 0 means no
inertia
  inertiaAccel = 1, -- inertia on acceleration
  air_control = 1, -- air control koefficient 0..1, 1 - special value (total control of
movement)
  air_resistance = 0.1, -- standard air resistance
  gravity = 9.8, -- gravity vector
  mass = 100, -- mass (in kg)
  min_slide_angle = 60, -- if surface slope is more than this angle, player starts sliding
(angle is in radians)
  max_climb_angle = 60, -- player cannot climb surface which slope is steeper than this
angle
  max_jump_angle = 45, -- player is not allowed to jump towards ground if this angle is
exceeded
  min_fall_angle = 65, -- player starts falling when slope is steeper than this
  max_vel_ground = 10, -- player cannot stand of surfaces that are moving faster than this
  timeImpulseRecover = 0.3, -- forcefully turns on inertia for that duration after
receiving an impulse
  nod_speed = 1, -- vertical camera shake speed after landings
  bActive = 1,-- 0 disables all simulation for the character, apart from moving along the
requested velocity
  collision_types = 271, -- (271 = ent_static | ent_terrain | ent_living | ent_rigid |
ent_sleeping_rigid) entity types to check collisions against


 },
 MultiplayerOptions = {
  bNetworked= 0,
 },

 bExcludeCover=0,
},

Client = {},
Server = {},

-- Temp.
_Flags = {},

 Editor={
 Icon = "physicsobject.bmp",
 IconOnTop=1,
  },

}
```

This information is followed by functions that implement the entity class.

Properties

Entity properties are placed inside the entity class. These properties are assigned to all new instances of the entity class created, visible and editable in Lumberyard Editor as the instance's **Entity Properties** table. The property values set for individual entity instances placed on a level are saved in the level file. When a property of an entity instance is changed in Lumberyard Editor, the `OnPropertyChange()` function called (if it is has been implemented for the script entity).

Lumberyard Editor provides the Archetype tool for assigning a common set of properties reused for multiple instance (even across multiple levels). For more information on Archetypes, see Archetype Entity in the Amazon Lumberyard User Guide.

When specifying entity class property names, use the following prefixes to signal the data type expected for a property value. This enables Lumberyard Editor to validate a property value when set.

Entity class property prefixes

Prefix	Data Type
b	boolean
f	float
i	integer
n	positive integer
s	string
clr	color
object_	object compatible with Lumberyard (CFG, CGA, CHR or CDF file)

You can add special comments to property values that can be utilized by the engine. For example:

```
--[25,100,0.1,"Damage threshold"]
```

This comment tells the engine the following:

- Value is limited to between 25 and 100.
- The float value uses a step of 0.01 (this limits the fidelity of values).
- The string "Damage threshold" will be displayed in the Lumberyard Editor as a tool tip.

Editor Table

The Editor table provides additional configuration information to Lumberyard Editor on how to handle instances of the entity.

Entity class editor variables

Variable	Description
Model	CGF model to be rendered over an entity instance.
ShowBounds	Flag indicating whether or not a bounding box is drawn around an entity instance when selected.
AbsoluteRadius	
Icon	BMP icon to be drawn over an entity instance.
IconOnTop	Flag indicating whether or not the icon is drawn over or under an entity instance.
DisplayArrow	
Links	

Functions

A script entity can include several callback functions called by the engine or game system. See Entity System Script Callbacks (p. 720) for more information.

Using Entity State

The Entity system provides a simple state-switching mechanism for script entities.

Each state consists of the following:

- Name (string)
- Lua table within the entity table, identified with the state name
- **OnEndState()** function (optional)
- **OnBeginState()** function (optional)
- Additional callback functions (optional) (See Entity System Script Callbacks (p. 720))

To declare states for an entity:

All entity states must be declared in the entity's main table to make the Entity system aware of them. The following examples show how to declare "Opened", "Closed", and "Destroyed" states.

```
AdvancedDoor =
{
    Client = {},
    Server = {},
    PropertiesInstance = ...
    Properties = ...
    States = {"Opened","Closed","Destroyed"},
}
```

To define an entity state:

Entity states can be either on the server or client (or both). The definition for a server-side "Opened" state might look as follows:

```
AdvancedDoor.Server.Opened =
{
 OnBeginState = function( self )
  if(self.Properties.bUsePortal==1)then
   System.ActivatePortal(self:GetWorldPos(),1,self.id);
  end;
  self.bUpdate=1;
  self.lasttime=0;
  AI.ModifySmartObjectStates( self.id, "Open-Closed" );
  self:Play(1);
  end,

 OnUpdate = function(self, dt)
  self:OnUpdate();
 end,
}
```

To set an entity's initial state:

Initially, an entity has no state. To set an entity's state, use one of the entity's callback functions (not to be confused with an entity state's callback function) to call its `GotoState()` method, shown in the following example. Once the entity state is set, the entity resides in that state and events will also be directed to that state.

```
function AdvancedDoor:OnReset()
    self:GotoState("Opened");
```

```
end
```

To change an entity's state:

Transitioning from the current state to any other state can also be done using the `GotoState()` method, as follows.

```
function AdvancedDoor.Server:OnHit(hit)
    ...
    if(self:IsDead())then
        self:GotoState("Destroyed");
    end
end
```

To query an entity's state:

Querying the state the entity is currently in can be done using the **GetState()** method, as follows.

```
if (self:GetState()=="Opened") then ...

if (self:GetState()~="Opened") then ...
```

Using Entity Slots

Each entity can have slots that are used to hold different resources available in Lumberyard. This topic describes how to work with entity slots.

Allocating a Slot

The following table lists the resources that can be allocated in a slot, along with the ScriptBind function used to allocate it.

Lumberyard resource	Function
static geometry	`LoadObject()` or `LoadSubObject()`
animated character	`LoadCharacter()`
particle emitter	`LoadParticleEffect()`
light	`LoadLight()`
cloud	`LoadCloud()`
fog	`LoadFogVolume()`
volume	`LoadVolumeObject()`

Modifying Slot Parameters

Each of these resource may be moved, rotated, or scaled relative to the entity itself.

- `SetSlotPos()`
- `GetSlotPos()`

- `SetSlotAngles()`
- `GetSlotAngles()`
- `SetSlotScale()`
- `GetSlotScale()`

You can add a parenting link between the slots, making it possible to have related positions.

- `SetParentSlot()`
- `GetParentSlot()`

Slot Management

To determine whether or not a specified slot is allocated, call the function `!IsSlotValid()`.

To free one slot, call `!FreeSlot()`

To free all allocated slots within the entity, call `!FreeAllSlots()`.

Loading a Slot

The following example illustrates loading a slot in a script function.

```
local pos={x=0,y=0,z=0};
self:LoadObject(0,props.fileModel);
self:SetSlotPos(0,pos);
self:SetCurrentSlot(0);
```

Linking Entities

In Lumberyard Editor, you can link an entity to other entities. These links are organized inside the Entity system. Each entity can link to multiple entities. Each link has a name associated to it. See the Amazon Lumberyard User Guide for more information about grouping and linking objects.

The following example Lua script searches the Entity system for any links to other entities that are named "Generator".

```
function RadarBase:IsPowered()
 local i=0;
 local link = self:GetLinkTarget("Generator", i);

 while (link) do
  Log("Generator %s", link:GetName());

  if (link:GetState() == "PowerOn") then
   if (link.PowerConnect) then
    link:PowerConnect(self.id);
    return true;
   end
  end

  i=i+1;
  link=self:GetLinkTarget("Generator", i);
 end

 return false;
end
```

The following functions are used to read or create entity links:

- `CountLinks`
- `CreateLink`
- `GetLink`
- `GetLinkName`
- `GetLinkTarget`
- `RemoveAllLinks`
- `RemoveLink`
- `SetLinkTarget`

Exposing an Entity to the Network

A script entity can be a serialized value on the network. This approach is done by setting the values on the server and having them automatically synchronized on all the clients. It also makes it possible to invoke client/server RMI functions.

Keep in mind the following limitations:

- There is no notification when a serialized value has changed.
- Values are controlled on the server only, there is no way to set values on the client.

Exposing a Script Entity to CryNetwork

To define the network features of an entity, call the ScriptBind function `Net.Expose()`, as illustrated in the following code. This code is written inside a Lua script within the global space, rather than in a function.

```
Net.Expose {
 Class = DeathMatch,
 ClientMethods = {
  ClVictory              = { RELIABLE_ORDERED, POST_ATTACH, ENTITYID, },
  ClNoWinner             = { RELIABLE_ORDERED, POST_ATTACH, },

  ClClientConnect        = { RELIABLE_UNORDERED, POST_ATTACH, STRING, BOOL },
  ClClientDisconnect     = { RELIABLE_UNORDERED, POST_ATTACH, STRING, },
  ClClientEnteredGame    = { RELIABLE_UNORDERED, POST_ATTACH, STRING, },
 },
 ServerMethods = {
  RequestRevive          = { RELIABLE_UNORDERED, POST_ATTACH, ENTITYID, },
  RequestSpectatorTarget = { RELIABLE_UNORDERED, POST_ATTACH, ENTITYID, INT8 },
 },
 ServerProperties = {
  busy  = BOOL,
 },
};
```

RMI functions

The RMI function is defined in either the ClientMethods and ServerMethods tables passed to the `Net.Expose()` function.

Order flags:

- UNRELIABLE_ORDERED

- RELIABLE_ORDERED
- RELIABLE_UNORDERED

The following descriptors control how the RMI is scheduled within the data serialization.

RMI attach flag	Description
NO_ATTACH	No special control (preferred)
PRE_ATTACH	Call occurs before data serialized
POST_ATTACH	Call occurs after the data serialized

The following example shows a function declaration:

```
function DeathMatch.Client:ClClientConnect(name, reconnect)
```

The following examples illustrate a function call:

```
self.allClients:ClVictory( winningPlayerId );
```

```
self.otherClients:ClClientConnect( channelId, player:GetName(), reconnect );
```

```
self.onClient:ClClientConnect( channelId, player:GetName(), reconnect );
```

See RMI Functions (p. 1014) for more details.

Note
Note: Script networking doesn't have an equivalent to the dependent object RMIs.

ServerProperties table

The entity table also contains a ServerProperties table that indicates which properties need to be synchronized. This is also the place to define the variable type of the value.

Exposing a Script Entity to CryAction

In addition, you must create a game object in CryAction and bind the new game object to the network game session. The following example shows the code placed in the `OnSpawn()` function:

```
CryAction.CreateGameObjectForEntity(self.id);
CryAction.BindGameObjectToNetwork(self.id);
```

You can also instruct the game object to receive a per-frame update callback, as in the following function call to CryAction:

```
CryAction.ForceGameObjectUpdate(self.id, true);
```

The script entity receive the `OnUpdate()` function callback of its Server table.

```
function Door.Server:OnUpdate(frameTime)
-- some code
```

```
end
```

Note
Adding update callback code to your script entity can decrease the performance of a game.

Event Bus (EBus)

Event buses (or EBus for short) are a general purpose system for dispatching messages. Ebuses have many advantages:

- **Abstraction** – Minimize hard dependencies between systems.
- **Event-driven programming** – Eliminate polling patterns for more scalable and high performing software.
- **Cleaner application code** – Safely dispatch messages without concern for what is handling them or whether they are being handled at all.
- **Concurrency** – Queue events from various threads for safe execution on another thread or for distributed system applications.
- **Predictability** – Provide support for ordering of handlers on a given bus.
- **Debugging** – Intercept messages for reporting, profiling, and introspection purposes.

The EBus source code can found in the Lumberyard directory location `<root>\dev\Code\Framework\AZCore\AZCore\EBus\EBus.h`.

Bus Configurations

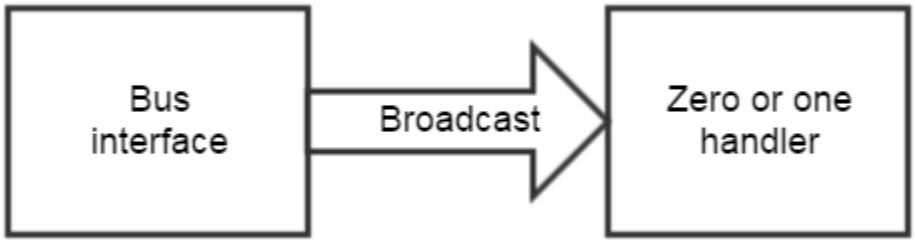
You can configure EBuses for various usage patterns. This section presents common configurations and their applications.

Topics

- Single Handler (p. 543)
- Many Handlers (p. 544)
- EBus with Addresses and a Single Handler (p. 545)
- EBus with Addresses and Many Handlers (p. 547)

Single Handler

The simplest configuration is a many-to-one (or zero) communication bus, much like a singleton pattern.

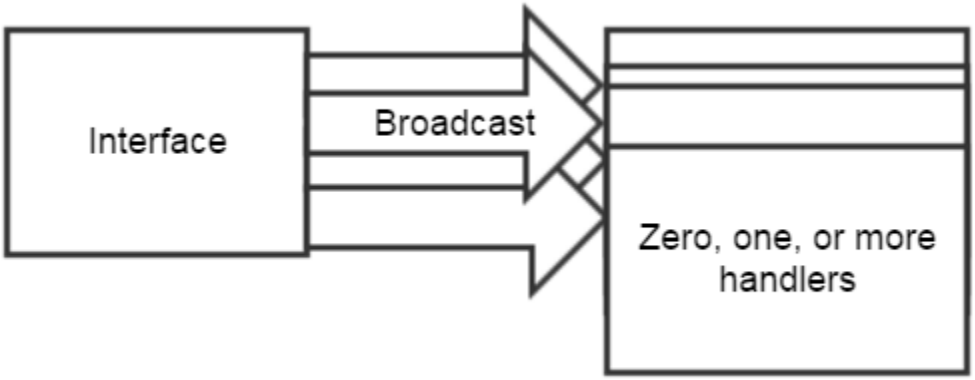


There is at most one handler, to which any sender can dispatch events. Senders need not manually check and de-reference pointers. If no handler is connected to the bus, the event is simply ignored.

```
// One handler is supported.
static const AZ::EBusHandlerPolicy HandlerPolicy = AZ::EBusHandlerPolicy::Single;

// The EBus uses a single address.
static const AZ::EBusAddressPolicy AddressPolicy = AZ::EBusAddressPolicy::Single;
```

Many Handlers

Another common configuration is one in which many handlers can be present. You can use this configuration to implement observer patterns, subscriptions to system events, or general-purpose broadcasting.



Events to the handlers can be received in defined or undefined order. You specify which one in the `HandlerPolicy` trait.

Example Without Handler Ordering

To handle events in no particular order, simply use the `Multiple` keyword in the `HandlerPolicy` trait, as in the following example:

```
// Multiple handlers. Events received in undefined order.
static const AZ::EBusHandlerPolicy HandlerPolicy = AZ::EBusHandlerPolicy::Multiple;

// The EBus uses a single address.
```

```
static const AZ::EBusAddressPolicy AddressPolicy = AZ::EBusAddressPolicy::Single;
```

Example with Handler Ordering

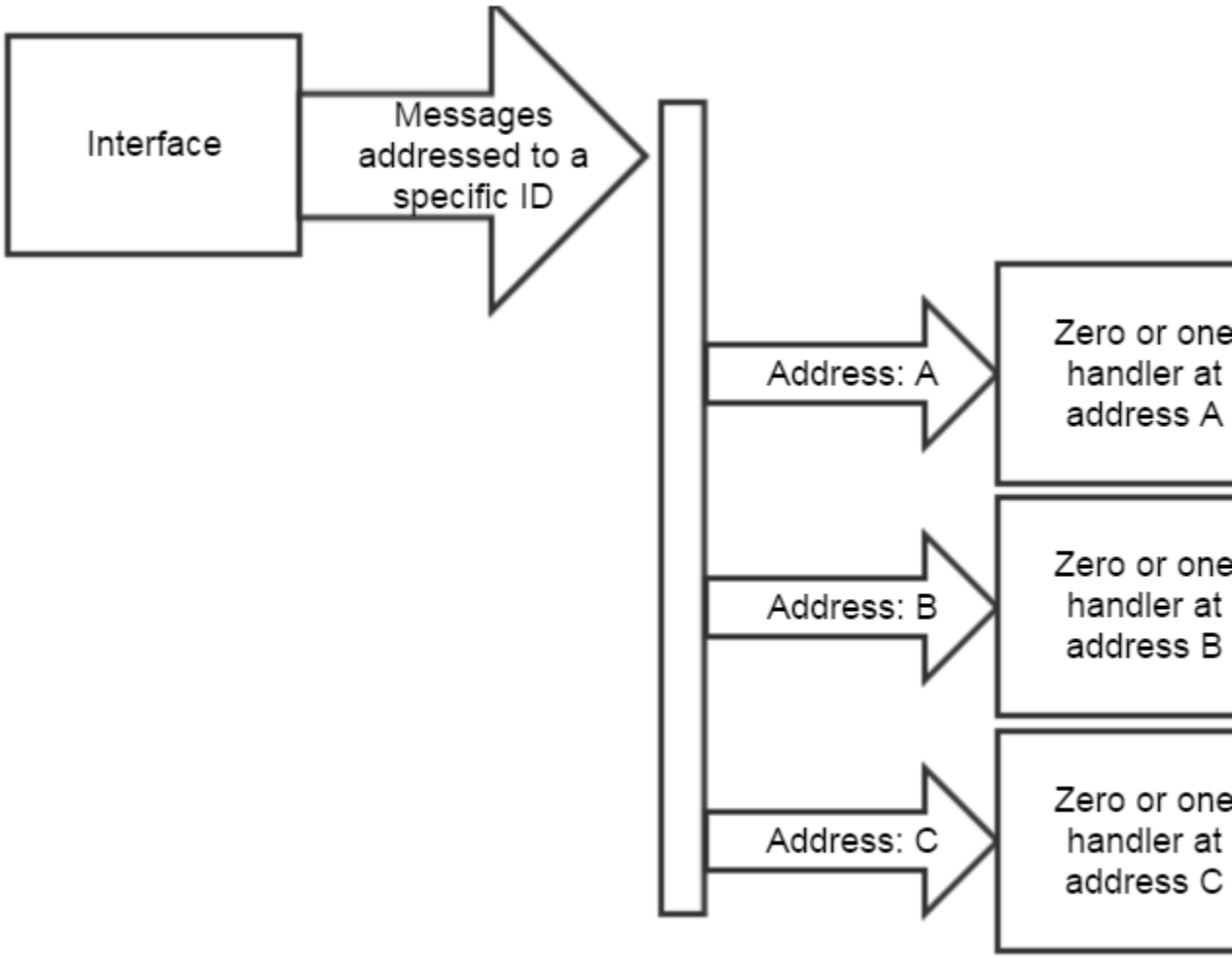
To handle events in a particular order, use the `MultipleAndOrdered` keyword in the `HandlerPolicy` trait, and then implement a custom handler-ordering function, as in the following example:

```
// Multiple handlers. Events received in defined order.
static const AZ::EBusHandlerPolicy HandlerPolicy =
 AZ::EBusHandlerPolicy::MultipleAndOrdered;

// The EBus uses a single address.
static const AZ::EBusAddressPolicy AddressPolicy = AZ::EBusAddressPolicy::Single;

// Implement a custom handler-ordering function
struct BusHandlerOrderCompare : public AZStd::binary_function<MyBusInterface*,
 MyBusInterface*, bool>
{
    AZ_FORCE_INLINE bool operator()(const MyBusInterface* left, const MyBusInterface*
 right) const { return left->GetOrder() < right->GetOrder();  }
};
```

EBus with Addresses and a Single Handler

EBuses also support addressing based on a custom ID. Events addressed to an ID are received by handlers connected to that ID. If an event is broadcast without an ID, it is received by handlers at all addresses.

A common use for this approach is for communication among the components of a single entity, or between components of a separate but related entity. In this case the entity ID is the address.



Example Without Address Ordering

In the following example, messages broadcast with an ID arrive at each address in no particular order.

```
// One handler per address is supported.
static const AZ::EBusHandlerPolicy HandlerPolicy = AZ::EBusHandlerPolicy::Single;

// The EBus has multiple addresses. Addresses are not ordered.
static const AZ::EBusAddressPolicy AddressPolicy = AZ::EBusAddressPolicy::ById;

// Messages are addressed by EntityId.
using BusIdType = AZ::EntityId;
```

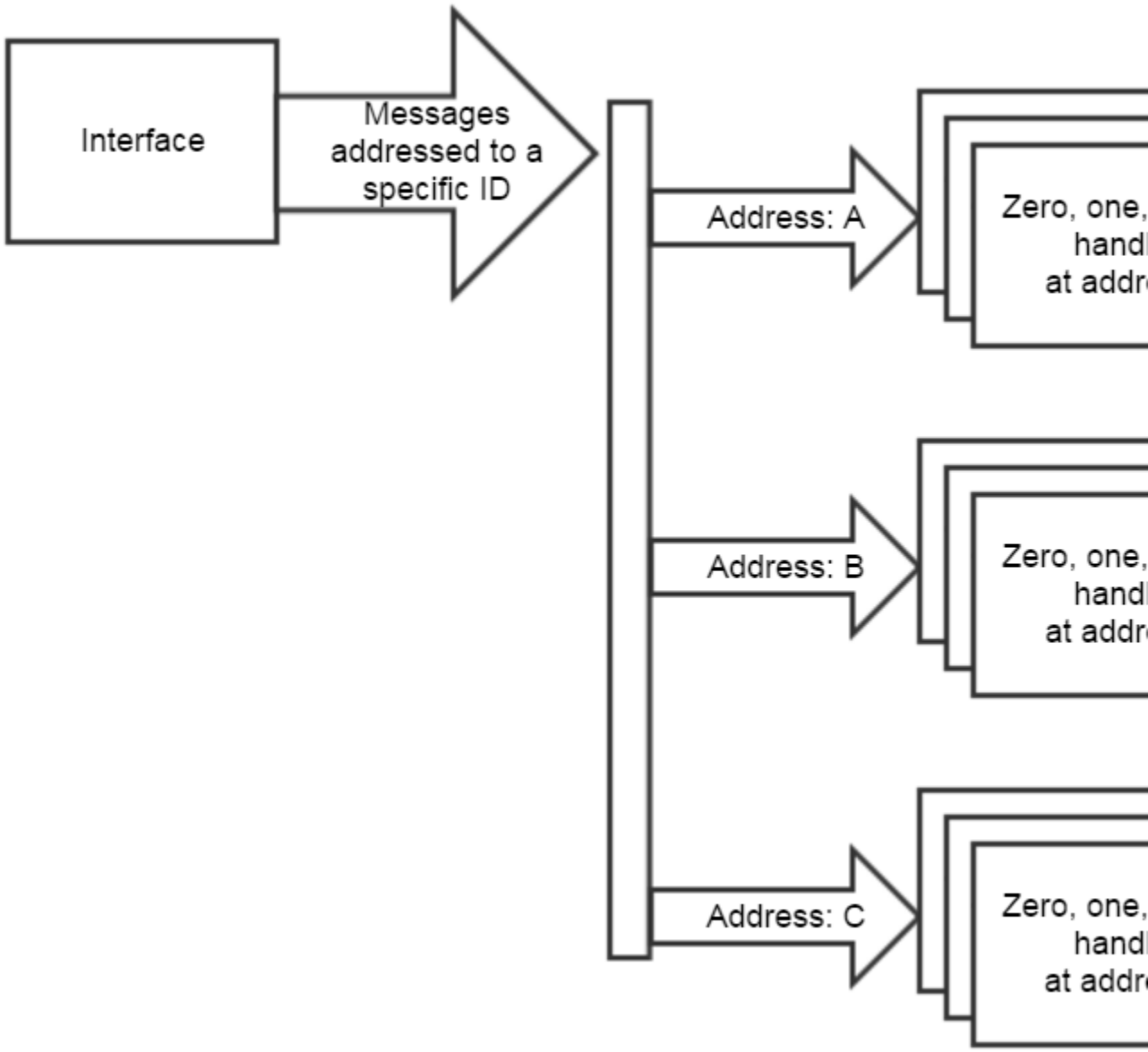
Example With Address Ordering

In the following example, messages broadcast with an ID arrive at each address in a specified order.

```
// One handler per address is supported.
static const AZ::EBusHandlerPolicy HandlerPolicy = AZ::EBusHandlerPolicy::Single;

// The EBus has multiple addresses. Addresses are ordered.
static const AZ::EBusAddressPolicy AddressPolicy = AZ::EBusAddressPolicy::ByIdAndOrdered;

// Messages are addressed by EntityId.
using BusIdType = AZ::EntityId;

// Addresses are ordered by EntityId.
using BusIdOrderCompare = AZStd::greater<BusIdType>;
```

EBus with Addresses and Many Handlers

In the previous configuration, only one handler is allowed per address. This is often desirable to enforce ownership of an EBus for a specific ID, as in the singleton case above. However, if you want more than one handler per address, you can configure the EBus accordingly:



Example: Without Address Ordering

In the following example, messages broadcast with an ID arrive at each address in no particular order. At each address, the order in which handlers receive the message is defined by `EBusHandlerPolicy`, which in this example is simply `ById`:

```
// Allow any number of handlers per address.
static const AZ::EBusHandlerPolicy HandlerPolicy = AZ::EBusHandlerPolicy::Multiple;

// The EBus has multiple addresses. Addresses are not ordered.
static const AZ::EBusAddressPolicy AddressPolicy = AZ::EBusAddressPolicy::ById;

// Messages are addressed by EntityId.
```

```
using BusIdType = AZ::EntityId;
```

Example: With Address Ordering

In the following example, messages broadcast with an ID arrive at each address in a specified order. At each address, the order in which handlers receive the message is defined by the `EBusHandlerPolicy`, which in this example is `ByIdAndOrdered`.

```
// Allow any number of handlers per address.
static const AZ::EBusHandlerPolicy HandlerPolicy = AZ::EBusHandlerPolicy::Multiple;

// The EBus has multiple addresses. Addresses are ordered.
static const AZ::EBusAddressPolicy AddressPolicy = AZ::EBusAddressPolicy::ByIdAndOrdered;

// We address the bus EntityId.
using BusIdType = AZ::EntityId;

// Addresses are ordered by EntityId.
using BusIdOrderCompare = AZStd::greater<BusIdType>;
```

Synchronous vs. Asynchronous

EBus supports both synchronous and asynchronous (queued) messaging.

Synchronous Messaging

Synchronous messages are sent to any and all handlers when an EBus event is invoked. Synchronous messages limit opportunities for asynchronous programming, but they offer the following benefits:

- They don't require storing a closure. Arguments are forwarded directly to callers.
- They let you retrieve an immediate result from a handler (event return value).
- They have no latency.

Asynchronous Messaging

Asynchronous messages have the following advantages:

- They create many more opportunities for parallelism and are much more future proof.
- They support queuing messages from any thread, dispatching them on a safe thread (like the main thread, or any thread that you choose).
- The code used to write them is inherently tolerant to latency and is easily migrated to actor models and other distributed platforms.
- The performance of the code that initiates events doesn't rely on the efficiency of the code that handles the events.
- In performance-critical code, asynchronous messages can improve i-cache and d-cache performance because they require fewer virtual function calls.

For information on declaring an EBus for queing and sending messages asynchronously, see Asynchronous/Queued Buses (p. 554) later in this topic.

Additional Features

EBuses contain other features that address various patterns and use cases:

- **Cache a pointer to which messages can be dispatched** – This is handy for EBuses that have IDs. Instead of looking up the EBus address by ID for each event, you can use the cached pointer for faster dispatching.
- **Queue any callable function on an EBus** – When you use queued messaging, you can queue a lambda or bound function against an EBus for execution on another thread. This is useful for general purpose thread-safe queuing.

Usage and Examples

This section provides examples of how to declare and configure an EBus, implement a handler, send messages, and receive return values.

Topics

- Declaring an EBus (p. 550)
- EBus Configuration Options (p. 551)
- Implementing a Handler (p. 552)
- Sending Messages to an EBus (p. 553)
- Retrieving Return Values (p. 553)
- Return Values from Multiple Handlers (p. 553)
- Asynchronous/Queued Buses (p. 554)

Declaring an EBus

Declaring an EBus is much like declaring any virtual interface class in C++. However, you can specify various configuration options that control how the EBus is generated at compile time and how it behaves.

Here is a simple example of a basic interface and associated EBus.

```
class ExampleInterface : public AZ::EBusTraits
{
public:
    // ------------------ EBus Configuration -------------------
    // These override the defaults in EBusTraits.

    // One handler per address is supported.
    static const AZ::EBusHandlerPolicy HandlerPolicy = AZ::EBusHandlerPolicy::Single;

    // The EBus contains a single address.
    static const AZ::EBusAddressPolicy AddressPolicy = AZ::EBusAddressPolicy::Single;
    // ------------------------ Other -------------------------

    virtual ~ExampleInterface() { };

    // ------------------ Handler Interface -------------------
    // Handlers inherit from ExampleInterfaceBus::Handler

    // Handlers are required to implement this because it's pure virtual.
    virtual void DoSomething() = 0;

    // Handlers can override this, but are not required to.
    virtual void SomeMessage() { }

    // Returns a value and has a parameter.
```

```
    virtual bool ReturnsValue(int x) = 0;
};

using ExampleInterfaceBus = AZ::EBus<ExampleInterface>;
```

Tip

Use descriptive names in EBuses, and avoid overloaded functions. Explicit and descriptive function names prevent future API name collisions when classes inherit your EBus interfaces. Avoiding overloaded functions improves the experience of using your EBuses. This is especially true from scripting environments such as Lua, in which descriptive names improve readability and clarity.

EBus Configuration Options

EBus configuration options are key to controlling how the EBus behaves. The configuration options used in the previous example are explained in the following sections.

HandlerPolicy

The `HandlerPolicy` trait determines how many handlers connect to an address on the EBus and the order in which handlers at each address receive events. The following example specifies a single handler (p. 543):

```
// One handler per address is supported.
static const AZ::EBusHandlerPolicy HandlerPolicy = AZ::EBusHandlerPolicy::Single;
```

The `HandlerPolicy` has two common uses:

- A singleton pattern in which various systems post messages or requests to a single system elsewhere in the codebase.
- A pattern where a specific component or an entity handles messages to the EBus. For example, you might have a mesh component that owns an entity. The mesh component handles all mesh-related queries addressed to the entity's ID.

Address Policy

The `AddressPolicy` trait defines how many addresses exist on the EBus. The following example specifes only a single address. An ID is not required.

```
// The EBus contains a single address.
static const AZ::EBusAddressPolicy AddressPolicy = AZ::EBusAddressPolicy::Single;
```

Practical use cases for a single address policy include any global EBus that is not tied to a specific entity, application-specific ID, or object.

EBusAddressPolicy Options

The `EBusAddressPolicy` has the following options:

- **`Single`** – The EBus uses a single address. No ID is used. The EBus can have a single handler (p. 543) or many handlers (p. 544).
- **`ById`** – The EBus has multiple addresses. The order in which addresses are notified when broadcasting events without an ID is not specified.

- **`ByIdAndOrdered`** – The EBus has multiple addresses. However, when broadcasting events without an ID, we want to control the order in which individual addresses are notified. The `BusIdOrderCompare` definition allows for arbitrary customization of ordering.

EBusHandlerPolicy Options

The `EBusHandlerPolicy` has the following options:

- **`Single`** – One handler per address is supported. Uses include an EBus with a single handler (p. 543) or an EBus with addresses and a single handler (p. 545).
- **`Multiple`** – Any number of handlers are supported. Ordering is ignored. Uses include many handlers (p. 544) or an EBus with addresses and many handlers (p. 547).
- **`MultipleAndOrdered`** – Any number of handlers are supported, and handlers are notified in a particular order. The `BusHandlerOrderCompare` definition allows for arbitrary customization of ordering.

Implementing a Handler

A handler of an EBus derives from `AZ::EBus<x>::Handler`. For convenience this was defined as `ExampleInterfaceBus` in the previous example (p. 550). This means that the handler can be derived from `ExampleInterfaceBus::Handler`.

```
#include "ExampleInterface.h"

// note: derives from bus handler, rather than directly from ExampleInterface
class MyHandler : protected ExampleInterfaceBus::Handler
{
public:
   void Activate();

protected:
   // Implement the handler interface:
   void DoSomething() override; // note:  Override specified.
   void SomeMessage() override;
   bool ReturnsValue(int x) override;
};
```

Note that handlers are not automatically connected to an EBus, but are disconnected automatically because the destructor of `Handler` calls `BusDisconnect`.

In order to actually connect to the EBus and start receiving events, your handler must call `BusConnect()`:

```
void MyHandler::Activate()
{
    // For a single EBus, this would be just BusConnect().
    // For multiple EBuses, you must specify the EBus to connect to:
    ExampleInterfaceBus::Handler::BusConnect();
}
```

You can call `BusConnect()` at any time and from any thread.

If your EBus is addressed, connect to the EBus by passing the EBus ID to `BusConnect()`. To listen on all addresses, call `BusConnect()` without passing in an ID.

```
// connect to the EBus at address 5.
ExampleAddressBus::Handler::BusConnect(5);
```

Sending Messages to an EBus

Anyone who can include the header can send messages to the EBus at any time. Using the previous example, a completely unrelated class can issue a `DoSomething` call on the EBus:

```
#include "ExampleInterface.h" // Note: You don't need to include MyHandler.h.
...
ExampleInterfaceBus::Broadcast(&ExampleInterfaceBus::Events::DoSomething);
```

EBuses also support a macro-based syntax. This syntax is being phased out, but uses of it can still be found in Lumberyard source code. The macro syntax for the previous example is as follows.

```
#include "ExampleInterface.h"
// Note:  You don't need to include MyHandler.h.
...
EBUS_EVENT(ExampleInterfaceBus, DoSomething);
```

If your EBus is addressed, you can send events to a specific address ID. Events broadcast globally are received at all addresses.

```
// Broadcasts to ALL HANDLERS on this EBus regardless of address (even if the EBus has
 addresses)
ExampleAddressBus::Broadcast(&ExampleAddressBus::Events::Test);

// Broadcasts only to handlers connected to address 5.
ExampleAddressBus::Event(5, &ExampleAddressBus::Events::Test);
```

Retrieving Return Values

If you make a synchronous call, you can also supply a variable in which to place the result:

```
// ALWAYS INITIALIZE YOUR RESULT!!!
// Since there may be nobody connected to the EBus, your result may not be populated.
bool result = false;
ExampleInterfaceBus::BroadcastResult(result, &ExampleInterfaceBus::Events::ReturnsValue,
 2);
```

In this example, if there are no handlers connected to the EBus, the `result` variable is not modified. If one or more handlers are connected to the EBus, `operator=()` is called on the `result` variable for each handler.

Return Values from Multiple Handlers

In certain cases you might have to aggregate the return value of a function when there are multiple handlers. For example, suppose you want to send a message to all handlers that asks whether any one handler objects to shutting down an application. If any one handler returns true, you should abort the shutdown. The following would not suffice:

```
// Counterexample: returnValue contains only the result of the final handler.
bool returnValue = false;
SomeInterfaceBus::BroadcastResult(returnValue,
 &SomeInterfaceBus::Events::DoesAnyoneObject);
```

Because the EBus issues `operator=` for each handler, `returnValue` would contain only the result of the final handler.

Instead, you can create a class to collect your results that overrides `operator=`. There are several built-in types for this, and you can make your own:

```
#include <AZCore/EBus/Results.h>

...
AZ::EBusAggregateResults<bool> results;
SomeInterfaceBus::BroadcastResult(results, &SomeInterfaceBus::Events::DoesAnyoneObject);

// results now contains a vector of all results from all handlers.

// alternative:
AZ::EBusLogicalResult<bool, AZStd::logical_or<bool>> response(false);
SomeInterfaceBus::BroadcastResult(response, &SomeInterfaceBus::Events::DoesAnyoneObject);

// response now contains each result, using a logical OR operation. So all responses are
 OR'd with each other.
```

Note
Additional building blocks (for example, arithmetic results) are available inside the `results.h` file.

Asynchronous/Queued Buses

To declare an EBus on which events can be queued and sent asynchronously, add the following to the EBus declaration:

```
static const bool EnableEventQueue = true;
```

You can use `QueueBroadcast` and `QueueEvent` to enqueue events on an EBus so that you can flush them later from a controlled location or thread.

To flush the queue at the appropriate location or thread, invoke the following:

```
ExampleInterfaceBus::ExecuteQueuedEvents();
```

File Access

This section covers tools available for tracking and accessing game files.

This section includes the following topics:

- CryPak File Archives (p. 555)
- Tracking File Access (p. 563)

CryPak File Archives

The CryPak module enables you to store game content files in a compressed or uncompressed archive.

Features

- Compatible with the standard zip format.
- Supports storing files in an archive or in the standard file system.
- Data can be read in a synchronous and asynchronous way through `IStreamCallback` (max 4GB offset, 4GB files).
- Files can be stored in compressed or uncompressed form.
- Uncompressed files can be read partially if required.
- File name comparison is not case sensitive.
- Supports loading of `.zip` or `.pak` files up to 4GB in size.

Unicode and Absolute Path Handling

Internally, all path-handling code is ASCII-based; as such, no Unicode (16-bit characters for different languages) functions can be used—this is to save memory and for simplicity. Because games can and should be developed with ASCII path names, no real need for Unicode exists. Game productions that don't follow these requirements have issues integrating other languages. For example, because a user might install a game to a directory with Unicode characters, absolute path names are explicitly avoided throughout the whole engine.

Layering

Usually the game content data is organized in several `.pak` files, which are located in the game directory. When a file is requested for an opening operation, the CryPak system loops through all registered `.pak`

files. `.pak` files are searched in order of creation. This allows patch `.pak` files, which have been added to the build later, to be in a preferred position. It is also possible to mix `.pak` files with loose files, which are stored directly in the file system (not in a `.pak` file). If a file exists as a loose file as well as in a `.pak` archive, the loose file is preferred when the game is in **devmode**. However, to discourage cheating in the shipped game, the file stored in the `.pak` is preferred over the loose file when the game is not run in devmode.

Slashes

Usually forward slashes (/) are used for internal processing, but users may enter paths that contain backslashes.

Special Folder Handling

You can use the path alias `@USER@` to specify a path relative to the user folder. This might be needed to store user-specific data. Windows can have restrictions on where the user can store files. For example, the program folder might not be writable at all. For that reason, screenshots, game data, and other files should be stored in the user folder. The following are examples of valid file names and paths:

```
@USER@/ProfilesSingle/Lisa.dat
game/Fred.dat
```

Internals

- A known implementation flaw exists where using more than approximately 1000 files per directory causes problems.
- Format properties:
  - The `.zip` file format stores each file with a small header that includes its path and filename in uncompressed text form. For faster file access, a directory is listed at the end of the file. The directory also stores the path and filename in uncompressed text form (redundant).

Creating a pak file using 7-Zip

To create a `.pak` file with 7-Zip's `7za.exe` command line tool, use the following syntax:

```
7za a -tzip -r -mx0 PakFileName [file1 file2 file3 ...] [dir1 dir2 ...]
```

Dealing with Large Pak Files

The zip RFC specifies two types of `.zip` files, indicated by `.zip` format version 45. Old `.zip` files can have a 4GB offset, but if legacy I/O functions are used, it is only possible to seek +- 2GB, which becomes the practical limit. The 4GB offsets have nothing to do with native machine types and do not change size across platforms and compilers, or configurations. The offsets for older versions of `.zip` files are in a machine independent `uint32`; the offsets for the new version `.zip` files are in `uint64`, appended to the old version `structs`. The version a `.zip` file uses is located in the header of the `.zip` file. Applications are free to not support the newer version. For more information, see the .ZIP File Format Specification.

Manual splits are not necessary, as RC supports auto-splitting:

- `zip_sizesplit` – Split `.zip` files automatically when the maximum configured or supported compressed size has been reached. The default limit is 2GB.
- `zip_maxsize` – Maximum compressed size of the `.zip` file in kilobytes (this gives an explicit limit).

Splitting works in all cases and supports multi-threading and incremental updates. It expands and shrinks the chain of necessary zip-parts automatically. Sorting is honored as much as possible, even in face of incremental modifications, but individual files can be appended to the end of the parts to fill in the leftover space even if this violates the sort order.

For more information about zip files, see Zip File Format Reference by Phil Katz.

Accessing Files with CryPak

In this tutorial you will learn how file reading and writing works through `CryPak`. The tutorial teaches you how to add new files to your project, read files from the file system and from pak archives, and write files to the file system.

Topics

- Preparation (p. 557)
- Reading Files with CryPak (p. 558)
- Writing to File System Files With CryPak (p. 560)
- Modifying Paks With CryArchive (p. 561)
- CryPak Details (p. 562)

Preparation

This tutorial demonstrates two different methods of loading a file: from inside a `.pak` archive, and directly from the file system. Before you can start, you need a file in a `.pak` archive, and a file with the same name (but with different content) in the file system. To verify which file is loaded, the example makes use of the content inside each text file.

To prepare sample files

1. Create a text file named `ExampleText.txt`.
2. Using a text editor, open `ExampleText.txt` and type in the following text:
3.
```
This sample was read from the .pak archive
```
4. Save the file.
5. Inside the `GameSDK` directory, create a subfolder called `Examples`.
6. Add the `ExampleText.txt` file to the `Examples` folder so that the path looks like this:

   <root>`\GameSDK\Examples\ExampleText.txt`
7. Run the following command from the directory *root*`\GameSDK`:

   ```
   ..\Tools\7za.exe a -tzip -r -mx0 Examples.pak Examples
   ```

   This command uses the executable file `7za.exe` (located in the `Tools` folder) to create an archive of the `Examples` folder called `Examples.pak`. Because you ran the command from the `GameSDK` folder, the archive was saved to the `GameSDK` folder. The `.pak` file contains only the file `Examples\ExampleText.txt`.
8. Using a text editor, change the text inside the *<root>*`\GameSDK\Examples\ExampleText.txt` file to something different, for example:

   ```
   This sample was read from the file system
   ```

Now you have two different text files with the same destination path, except that one is stored directly in the file system, and the other is inside the `.pak` file.

Reading Files with CryPak

Now you can write some code to read the information from the `ExampleText.txt` file that you created.

1. Type the following, which contains the `if-else` statement that frames the code. The `ReadFromExampleFile()` function will read the contents of the file and return `true` if it succeeds, and `false` if not.

```
    char* fileContent = NULL;
    if (!ReadFromExampleFile(&fileContent))
    {
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "ReadFromExampleFile()
 failed");
    }
    else
    {
        CryLogAlways("ExampleText contains %s", fileContent);
        [...] // this line will be added later on
    }
```

   If `ReadFromExampleFile()` is successful in reading `ExampleText.txt`, `fileContent` will be the space in memory that contains the text that it read.
2. Type the following, which stubs out the `ReadFromExampleFile()` function.

```
bool ReadFromExampleFile(char** fileContent)
{
    CCryFile file;
    size_t fileSize = 0;
    const char* filename = "examples/exampletext.txt";

    [...]
}
```

   - `file` of type `CCryFile` can make use of `CryPak` to access files directly from the file system or from inside a `.pak` archive.
   - `fileSize` - Defines the end of the message. In this case, reading does not end by detecting the null character '`\0`'.
   - `filename` - Specifies the path of the file to be loaded and is case-insensitive.
3. Type the following, which uses `CryPak` to search the file.

```
    char str[1024];
    if (!file.Open(filename, "r"))
    {
        sprintf(str, "Can't open file, (%s)", filename);
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "%s", str);
        return false;
    }
```

   - `Open()` invokes `CryPak` to search the file specified by `filename`.
   - File access mode `"r"` specifies that a plain text file is going to be read. To read a binary file, use `"rb"` instead.
4. Type the following, which gets the length of the file. If the file is not empty, it the allocates the memory required as indicated by the file length. It then reads the file content. It aborts if the size of the content is not equal to the file length.

```
    fileSize = file.GetLength();
    if (fileSize <= 0)
    {
        sprintf(str, "File is empty, (%s)", filename);
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "%s", str);
        return false;
    }

    char* content = new char[fileSize + 1];
    content[fileSize] = '\0';

    if (file.ReadRaw(content, fileSize) != fileSize)
    {
        delete[] content;
        sprintf(str, "Can't read file, (%s)", filename);
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "%s", str);
        return false;
    }
```

- `content` is the local pointer to a `char` array in memory which gets initialized by the length returned by `GetLength()` and an extra null character.
- `ReadRaw` fills `content` with the information read from the text file. In case of a failure, the allocated memory of `content` is freed.

5. Type the following, which closes the file handle and sets the `fileContent` pointer so that the locally created data can be used outside the function. Finally, it returns `true` since the reading was successful.

```
    file.Close();

    *fileContent = content;
    return true;
```

Note
In the example, the caller of `ReadFromExampleFile()` is responsible for freeing the heap memory which has been allocated to store the data from the text file. Thus, after the data has been used, be sure to add the call `delete[] fileContent;`.

6. To check if the reading was successful, run the game and check the `Game.log` file.

Complete example code (file reading)

`Calling ReadFromExampleFile()`

```
    char* fileContent = NULL;
    if (!ReadFromExampleFile(&fileContent))
    {
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "ReadFromExampleFile()
 failed");
    }
    else
    {
        CryLogAlways("ExampleText contains %s", fileContent);
        delete[] fileContent;
    }
```

`ReadFromExampleFile() implementation`

```
bool ReadFromExampleFile(char** fileContent)
```

```
{
    CCryFile file;
    size_t fileSize = 0;
    const char* filename = "examples/exampletext.txt";

    char str[1024];
    if (!file.Open(filename, "r"))
    {
        sprintf(str, "Can't open file, (%s)", filename);
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "%s", str);
        return false;
    }

    fileSize = file.GetLength();
    if (fileSize <= 0)
    {
        sprintf(str, "File is empty, (%s)", filename);
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "%s", str);
        return false;
    }

    char* content = new char[fileSize + 1];
    content[fileSize] = '\0';

    if (file.ReadRaw(content, fileSize) != fileSize)
    {
        delete[] content;
        sprintf(str, "Can't read file, (%s)", filename);
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "%s", str);
        return false;
    }

    file.Close();

    *fileContent = content;
    return true;
}
```

Writing to File System Files With CryPak

Writing a file is similar to the process for reading one. To write to files, you use `CCryFile::Write`, which always writes to the file system and never to `.pak` archives. For information on writing files to archive files, see Modifying Paks With CryArchive (p. 561).

1. Type the following, which contains the `if-else` statement that frames the code for writing to a file. The `WriteToExampleFile()` function write will write the contents of the file and return `true` if it succeeds, and `false` if not.

```
    char* newContent = "File has been modified";
    bool appendToFile = false;
    if (!WriteToExampleFile(newContent, strlen(newContent), appendToFile))
    {
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "WriteToExampleFile()
 failed");
    }
    else
    {
        CryLogAlways("Text has been written to file, %s", newContent);
    }
```

   - `WriteToExampleFile()` takes the following three parameters:
     - `newContent` - The text which will be written to `ExampleText.txt` on the file system.

- `strlen(newContent)` - Returns size of `newContent`, which is the number of bytes to be written.
- `appendToFile` - `true` if `newContent` will be added to the already existing content; `false` if the file will be overwritten.

2. Type the following for the `WriteToExampleFile)` function.

```
bool WriteToExampleFile(char* text, int bytes, bool appendToFile)
{
    CCryFile file;
    const char* filename = "examples/exampletext.txt";

    assert(bytes > 0);
    char* mode = NULL;
    if (appendToFile)
        mode = "a";
    else
        mode = "w";

    char str[1024];
    if (!file.Open(filename, mode))
    {
        sprintf(str, "Can't open file, (%s)", filename);
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "%s", str);
        return false;
    }

 [...]

    file.Close();
    return true;
}
```

- `mode` specifies if the text is to be appended to the existing file or if it will overwrite existing file contents. `"w"` means 'write' to a clean file, and `"a"` means 'append' to the existing file.

3. The final step writes the text to the file and returns the number of bytes written, or an error message if none were written.

```
    int bytesWritten = file.Write(text, bytes);
    assert(bytesWritten == bytes);

    if (bytesWritten == 0)
    {
        sprintf(str, "Can't write to file, (%s)", filename);
        CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "%s", str);
        return false;
    }
```

- `bytesWritten` tells how many bytes were written by calling the `Write()` function.

Modifying Paks With CryArchive

This section contains a short example that shows how files are added, updated and removed from an archive. The example intentionally uses the `USER` folder instead of the `GameSDK` folder because the `.pak` files inside the `GameSDK` folder are loaded by default at startup and therefore are marked as `Read-Only`. (Files in the `USER` folder are not loaded by default at startup.)

```
    string pakFilename = PathUtil::AddSlash("@USER@") + "Examples.pak";
    const char* filename = "Examples/ExampleText.txt";
    char* text = "File has been modified by CryArchive";
```

```
    unsigned length = strlen(text);

    _smart_ptr<ICryArchive> pCryArchive = gEnv->pCryPak->OpenArchive(pakFilename.c_str(),
 ICryArchive::FLAGS_RELATIVE_PATHS_ONLY | ICryArchive::FLAGS_CREATE_NEW);
    if (pCryArchive)
    {
        pCryArchive->UpdateFile(filename, text, length, ICryArchive::METHOD_STORE, 0);
    }
```

- `UpdateFile()` - Modifies an existing file inside the `.pak` archive or creates a new one if it does not exist.
- `ICryArchive::FLAGS_CREATE_NEW` - Forces a new `.pak` file to be created. If you want to add (append) files, remove this flag.
- To remove files or folders from an archive, use one of the following commands in place of `UpdateFile()`: `RemoveFile()`, `RemoveDir()` or `RemoveAll()`.

CryPak Details

Initialization

To ensure that `.pak` files can be accessed from game code at anytime, the `CrySystem` module initializes `CryPak` in `CSystem::Init` by calling the following functions:

- `InitFileSystem(startupParams.pGameStartup);`
- `InitFileSystem_LoadEngineFolders();`

Tip
A good spot to test game initialization is in inside `Game.cpp` at the beginning of `CGame::Init`.

Pak file type priorities

Whether CryPak processes files in the file system first, or files in `.pak` files first, depends on the value of `pakPriority`. The default value of `pakPriority` depends on the configuration settings of your build, but it can also manually be changed by assigning the cvar `sys_PakPriority` the values `0, 1, 2 or 3`. The meaning of these values is show in the enum `EPakPriority`:

`PakVars.h`

```
enum EPakPriority
{
    ePakPriorityFileFirst = 0,
    ePakPriorityPakFirst  = 1,
    ePakPriorityPakOnly   = 2,
    ePakPriorityFileFirstModsOnly = 3,
};
```

Pak loading and search priorities

The reason for adding the new pak file to the `GameSDK` folder in this example is because `.pak` files are loaded from the `GameSDK` path first. The loading order and search order of .pak file folders are as follows. Note that the loading order and the search order are the reverse of each other.

.pak file load order

1. `GameSDK`: `<root>\GameSDK\*.pak`

2. `Engine: <root>\Engine\`
   a. `Engine.pak`
   b. `ShaderCache.pak`
   c. `ShaderCacheStartup.pak`
   d. `Shaders.pak`
   e. `ShadersBin.pak`
3. Mods: `root\Mods\MyMod\GameSDK\*.pak` (this assumes that you run the game with the command argument `-mod "MyMod"`)

.pak file search order

1. `Mods` If more than one mod folder exists, they will be checked in the reverse order in which they were added.
2. `Engine`
3. `GameSDK`

Tracking File Access

It's possible to track invalid file reads that occur during game run time. The error message `Invalid File Access` occurs when an attempt is made to read or open files from a thread that is not the streaming thread. These file access operations can cause stalls that can be quite severe.

> **Note**
> Only access attempts from the main thread and render thread are logged. This feature is disabled in RELEASE builds.

CVars

The following cvars enable different options for tracking file access.

`sys_PakLogInvalidFileAccess`

`1` (default):

- Access is logged to `game.log`.
- Generates a `perfHUD` warning.
- The warning is displayed in red in the upper left corner of the screen.
- A 3 second-stall in non-release builds is induced.

`sys_PakMessageInvalidFileAccess`

- When a file is accessed, creates a popup dialog on the PC. At this point, you can choose to break into the debugger, or continue.

Where invalid access is defined

The points which define when a file access attempt is considered invalid are set by implementing `ICryPak::DisableRuntimeFileAccess` to return true or false. The points may need to be tweaked for single player and multiplayer games.

Exceptions

To add exceptions to file access tracking so that you can ignore files like `game.log`, create an instance of `CDebugAllowFileAccess` in the scope which accesses the file.

Resolving file access callstacks

The files that you collect with `pak_LogInvalidFileAccess 2` must have their callstacks resolved. To do this requires the following tools from the `XenonStackParse` folder of the `Tools` directory.:

- The `.pdb` files from the build
- The `XenonStackParse` tool
- The `ProcessFileAccess.py` helper script

The directory structure for running `ProcessFileAccess.py` should resemble the following:

```
<Root>
--> XenonStackParse
--> FileAccessLogs (this folder should contain the .pdb files)
------> Processed (this folder contains the output from XenonStackParse)
```

Run `ProcessFileAccess.py` from the `FileAccessLogs` directory (`XenonStackParse` uses the working directory to search for the `.pdb` files). The script creates a folder called `Processed` and a file within it that contains the resolved callstack for each of the log files.

Graphics and Rendering

Lumberyard's rendering technology starts with a modern, physically-based shading core that renders materials based on real world physical parameters (such as base color, metalicity, smoothness, and specularity), allowing you to achieve realistic results using the same physically based parameters used in the highest end film rendering pipelines.

The rendering core is supplemented by a rich set of the most frequently used real time lighting, shading, special effects, and post effects features, such as physical lights, global illumination, volumetric fog, procedural weathering, particle systems, dynamic real time shadows, motion blur, bokeh depth of field, post color correction, and more.

Lumberyard's rendering engine is tightly integrated with Lumberyard Editor, so the graphical fidelity and performance achieved in your game is what you see in the editor. Changes made in the editor are instantly reflected in the fully rendered scene, allowing for immediate feedback and rapid iteration.

The Lumberyard rendering technology is designed to take maximum advantage of today's high-end PC and console platforms, while maintaining compatibility with older hardware by scaling down graphical features and fidelity without compromising the core visual elements of your scene.

This section includes the following topics:

- Render Nodes (p. 565)
- TrueType Font Rendering (p. 569)
- Generating Stars DAT File (p. 570)
- Anti-Aliasing and Supersampling (p. 571)

Render Nodes

To visualize objects in a world, Lumberyard defines the concepts of the render node and render element. Render nodes represent general objects in the 3D engine. Among other things, they are used to build a hierarchy for visibility culling, allow physics interactions (optional), and rendering.

For actual rendering, render nodes add themselves to the renderer, passing an appropriate render element that implements the actual drawing of the object. This process happens with the help of render objects, as shown in the sample code below

Creating a New Render Node

The following example creates a render node called **PrismObject**. It is derived from `IRenderNode`, defined in `Code/CryEngine/CryCommon/IEntityRenderState.h`.

1. Add the interface for `IPrismObjectRenderNode` to `CryEngine/CryCommon/IEntityRenderState.h` to make it publicly available.

```
struct IPrismRenderNode : public IRenderNode
{
  ...
};
```

2. Add a new enum to the list of already defined render nodes in `CryEngine/CryCommon/IEntityRenderState.h`.

```
enum EERType
{
  ...
  eERType_PrismObject,
  ...
};
```

3. Add PrismObjectRenderNode.h to Cry3DEngine.

```
#ifndef _PRISM_RENDERNODE_
#define _PRISM_RENDERNODE_

#pragma once

class CPrismRenderNode : public IPrismRenderNode, public Cry3DEngineBase
{
public:
  // interface IPrismRenderNode
  ...

  // interface IRenderNode
  virtual void SetMatrix(const Matrix34& mat);
  virtual EERType GetRenderNodeType();
  virtual const char* GetEntityClassName() const { return "PrismObject"; }
  virtual const char* GetName() const;
  virtual Vec3 GetPos(bool bWorldOnly = true) const;
  virtual bool Render(const SRendParams &rParam);
  virtual IPhysicalEntity* GetPhysics() const { return 0; }
  virtual void SetPhysics(IPhysicalEntity*) {}
  virtual void SetMaterial(IMaterial* pMat) { m_pMaterial = pMat; }
  virtual IMaterial* GetMaterial(Vec3* pHitPos = 0) { return m_pMaterial; }
  virtual float GetMaxViewDist();
  virtual void GetMemoryUsage(ICrySizer* pSizer);
  virtual const AABB GetBBox() const { return m_WSBBox; }
  virtual void SetBBox( const AABB& WSBBox ) { m_WSBBox = WSBBox; }

private:
  CPrismRenderNode();

private:
  ~CPrismRenderNode();

  AABB m_WSBBox;
  Matrix34 m_mat;
  _smart_ptr< IMaterial > m_pMaterial;
  CREPrismObject* m_pRE;
```

```
};

#endif // #ifndef _PRISM_RENDERNODE_
```

4. Add PrismObjectRenderNode.cpp to Cry3DEngine.

```
#include "StdAfx.h"
#include "PrismRenderNode.h"

CPrismRenderNode::CPrismRenderNode() :m_pMaterial(0)
{
                m_mat.SetIdentity();
                m_WSBBox = AABB(Vec3(-1, -1, -1), Vec3(1, 1, 1));
                m_pRE = (CREPrismObject*) GetRenderer()->EF_CreateRE(eDATA_PrismObject);
                m_dwRndFlags |= ERF_CASTSHADOWMAPS | ERF_HAS_CASTSHADOWMAPS;
}

CPrismRenderNode::~CPrismRenderNode()
{
                if (m_pRE)
                                m_pRE->Release(false);

                Get3DEngine()->FreeRenderNodeState(this);
}

void CPrismRenderNode::SetMatrix(const Matrix34& mat)
{
                m_mat = mat;
                m_WSBBox.SetTransformedAABB(mat, AABB(Vec3(-1, -1, -1), Vec3(1, 1, 1)));
                Get3DEngine()->RegisterEntity(this);
}


const char* CPrismRenderNode::GetName() const
{
                return "PrismObject";
}

void CPrismRenderNode::Render(const SRendParams& rParam, const SRenderingPassInfo
 &passInfo)
{
  FUNCTION_PROFILER_3DENGINE;

                if(!m_pMaterial)
                                return;

                // create temp render node to submit this prism object to the renderer
                CRenderObject *pRO = GetRenderer()-
>EF_GetObject_Temp(passInfo.ThreadID());                    // pointer could be cached

                if(pRO)
                {
                                // set basic render object properties
                                pRO->m_II.m_Matrix = m_mat;
                                pRO->m_ObjFlags |= FOB_TRANS_MASK;
                                pRO->m_fSort = 0;
                                pRO->m_fDistance = rParam.fDistance;

                                // transform camera into object space
                                const CCamera& cam(passInfo.GetCamera());
                                Vec3 viewerPosWS(cam.GetPosition());

                                // set render object properties
                                m_pRE->m_center = m_mat.GetTranslation();
```

```
                                SShaderItem& shaderItem(m_pMaterial->GetShaderItem(0));

                                GetRenderer()->EF_AddEf(m_pRE, shaderItem, pRO, passInfo,
 EFSLIST_GENERAL, 0, SRendItemSorter(rParam.rendItemSorter));
                }
}

void CPrismRenderNode::GetMemoryUsage(ICrySizer* pSizer) const
{
                SIZER_COMPONENT_NAME(pSizer, "PrismRenderNode");
                pSizer->AddObject(this, sizeof(*this));
}

void CPrismRenderNode::OffsetPosition(const Vec3& delta)
{
                if (m_pRNTmpData) m_pRNTmpData->OffsetPosition(delta);
                m_WSBBox.Move(delta);
                m_mat.SetTranslation(m_mat.GetTranslation() + delta);
                if (m_pRE) m_pRE->m_center += delta;
}

void CPrismRenderNode::FillBBox(AABB & aabb)
{
                aabb = CPrismRenderNode::GetBBox();
}

EERType CPrismRenderNode::GetRenderNodeType()
{
                return eERType_PrismObject;
}

float CPrismRenderNode::GetMaxViewDist()
{
                return 1000.0f;
}

Vec3 CPrismRenderNode::GetPos(bool bWorldOnly) const
{
                return m_mat.GetTranslation();
}

IMaterial* CPrismRenderNode::GetMaterial(Vec3* pHitPos)
{
                return m_pMaterial;
}
```

5. To allow client code to create an instance of the new render node, extend the following function in `/Code/CryEngine/Cry3DEngine/3DEngine.cpp`

```
...
#include "PrismRenderNode.h"
...
IRenderNode * C3DEngine::CreateRenderNode(EERType type)
{
  switch (type)
  {
  ...
  case eERType_PrismObject:
  {
    IPrismRenderNode* pRenderNode = new CPrismRenderNode();
    return pRenderNode;
  }
  ...
```

TrueType Font Rendering

CryFont is used to generate font textures that are required to render text on the screen. The various features of font rendering can be seen by using the `r_DebugFontRendering` console variable.

The output is not only to test the functionality but also to document how the features can be used.

Supported Features

CryFont supports the following features:

- Font shaders – Used to configure the appearance of fonts. Multiple passes with configurable offset and color are supported to enable generation of shadows or outlines. A sample font shader is shown in the following XML example.

```
<fontshader>
  <font path="VeraMono.ttf" w="288" h="416"/>
  <effect name="default">
    <pass>
      <color r="0" g="0" b="0" a="1"/>
      <pos x="1" y="1"/>
    </pass>
  </effect>
  <effect name="console">
    <pass>
      <color r="0" g="0" b="0" a="0.5"/>
      <pos x="2" y="2"/>
    </pass>
  </effect>
</fontshader>
```

  The attributes *w* and *h* of the XML font element specify the width and height of the font texture. The order of the passes in XML defines the order in which the passes are rendered. A `<pass>` element without child elements means that the pass is rendered with the default settings. The `<pos>` tag is used to offset the font, while the `<color>` tag is used to set font color and define the transparency (with the alpha channel *a*).
- Unicode – The default font used does not support all Unicode characters (to save memory), but other fonts can be used.
- TrueType fonts as source – Cached in a small texture. Common characters are pre-cached, but runtime updates are possible and supported.
- Colored text rendering
- Adjustable transparency
- Color variations within a string – Use a value of **`$0..9`** to set one of the 10 available colors. Use **`$$`** to print the $ symbol, and **`$o`** to switch off the feature.
- Returns and tabs within a string
- Text alignment
- Computation of a string's width and height – Used internally to handle center and right alignment.
- Font size variations – Bilinear filtering allows some blurring, but no mipmaps are used so this feature has limitations in minification.
- Proportional and monospace fonts
- Pixel-perfect rendering with exact texel-to-pixel mapping for best quality.

Useful Console Commands

The following console commands provide information about font rendering.

r_DebugFontRendering

Provides information on various font rendering features, useful for verifying function and documenting usage.

- 0=off
- 1=display

r_DumpFontNames

Logs a list of fonts currently loaded.

r_DumpFontTexture

Dumps the texture of a specified font to a bitmap file. You can use `r_DumpFontTexture` to get the loaded font names.

Generating Stars DAT File

The Stars DAT file contains star data that is used in sky rendering. This topic provides information you'll need if you want to modify the data in this file. It assumes you have some familiarity with generating binary files.

Star data is located in `Build\Engine\EngineAssets\Sky\stars.dat`. This data is loaded in the function `CStars::LoadData`, implemented in the file `CRESky.cpp`.

File Format

The Stars DAT file uses a simple binary format; it can be easily modified using an editing tool. The file starts with a header, followed by entries for each star. The header specifies the number of entries in the file.

All types stored in little-endian format, float32 in IEEE-754 format.

Star data provided in the SDK is based on real-world information. Typically, you can also use existing star catalogs to populate this information for you.

The file elements are as follows:

Header (12 bytes)

Name	Offset	Type	Value
Tag	0	uint32	0x52415453 (ASCII: STAR)
Version	4	uint32	0x00010001
NumStars	8	uint32	Number of star entries in the file

Entry (12 bytes)

Name	Offset	Type	Value
RightAscension	0	float32	in radians

Name	Offset	Type	Value
Declination	4	float32	in radians
Red	8	uint8	star color, red channel
Green	9	uint8	star color, green channel
Blue	10	uint8	star color, blue channel
Magnitude	11	uint8	brightness, normalized range

Anti-Aliasing and Supersampling

Perceived graphics quality in a game is highly dependent on having clean and stable images. Lumberyard offers an efficient, post-processing-based, anti-aliasing solution that can be controlled in the Console using the console variable `r_AntialiasingMode`. This solution allows game developers to set the amount of anti-aliasing needed to produce graphics that fit their needs, from very sharp images to softer blurred images. Lumberyard also supports supersampling for very high-quality rendering.

Controlling Anti-Aliasing

The following table lists the currently available anti-aliasing modes available in Lumberyard using the CVar `r_AntialiasingMode`.

Mode	CVar Value	Description
No anti-aliasing	0	Disables post-processing-based anti-aliasing. Useful for debugging. Some game developers opt to use a higher resolution rather than spending system resources on anti-aliasing.
SMAA_Low (1X)	1	Enables sub-pixel morphological anti-aliasing (SMAA), which removes jaggies (staircase artifacts) on polygon edges. This mode does not address sub pixel details.
SMAA_Med (1TX)	2	Enables SMAA with basic temporal re-projection to reduce pixel crawling.
SMAA_High (2TX)	3	Enables SMAA with enhanced temporal re-projection, including matrix jittering. This mode usually provides the best image quality but can suffer from occasionally flickering edges.

The images below illustrate the range of graphics quality that can be achieved depending on the anti-alias setting used.

Controlling Supersampling

In addition to anti-aliasing, Lumberyard supports supersampling for very-high-quality rendering. Supersampling renders the scene at a higher resolution and downscales the image to obtain smooth and stable edges. Due to the high internal rendering resolution, supersampling is very performance-heavy and only suitable for games intended to be played on high-end PCs.

Writing Lua Scripts for the Component Entity System

You can use Lua in Lumberyard to facilitate quick iteration of your game project. Lua is a powerful, fast, lightweight, embeddable scripting language. When you construct new gameplay and game systems, you can run your changes immediately, without compiling your source code.

Learning Lua

For learning the Lua language itself, the lua.org website is a good place to start.

- Official Lua Documentation – Provides a central location for information about Lua, including a Getting started page.
- Programming in Lua – This text is a resource for getting started with Lua programming.
- Lua 5.1 Reference Manual – Provides a reference of all the functions that are available by default in Lua.

Learning Lua in Lumberyard

After you read through this tutorial on writing Lua scripts for the component entity system, learn more about using Lua in Lumberyard by consulting the following resources.

- For information on Lumberyard's built-in Lua editor, see Lua Editor (p. 587).
- For sample Lua scripts, see the Lumberyard `\dev\SamplesProject\Scripts` directory and its subdirectories.
- For information about Lua API operations in Lumberyard, see the Component Entity Lua API Reference.

- For information about the Lumberyard EBus, see Event Bus (EBus) (p. 543).

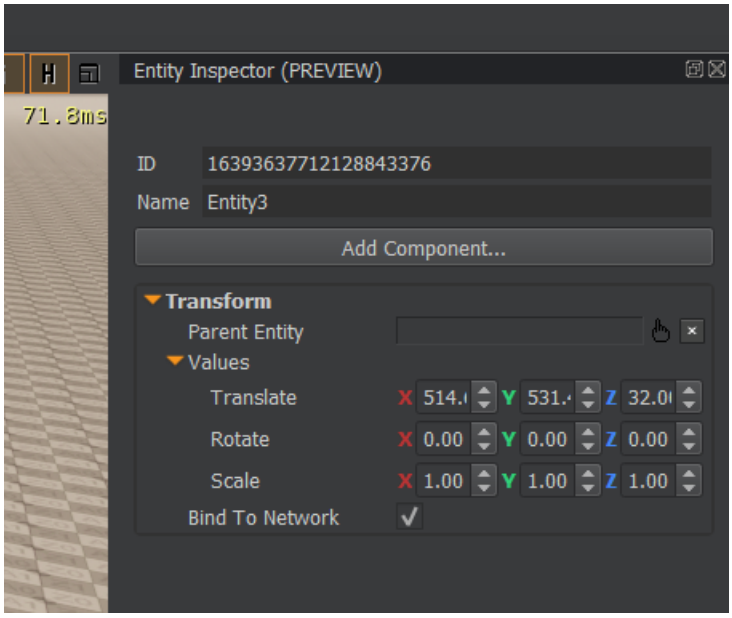
Note
Starting with Lumberyard 1.8, Lua scripts use the new behavior context that replaces the legacy script context. Scripts that were written before the integration of the behavior context no longer work in Lumberyard versions 1.8 and later. For information on updating code from legacy script context to the new behavior context, see the migration notes for Lumberyard 1.8. For information on the behavior context, see Behavior Context (p. 453).

Adding Lua Scripts to Component Entities

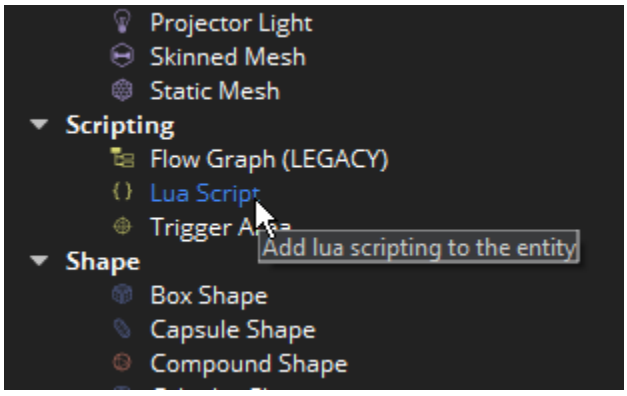
Lumberyard makes it easy for you to add script functionality to your game entities by using the **Lua Script** component. The following steps show you how to do this in Lumberyard Editor.

To add a Lua script to a component entity in Lumberyard Editor

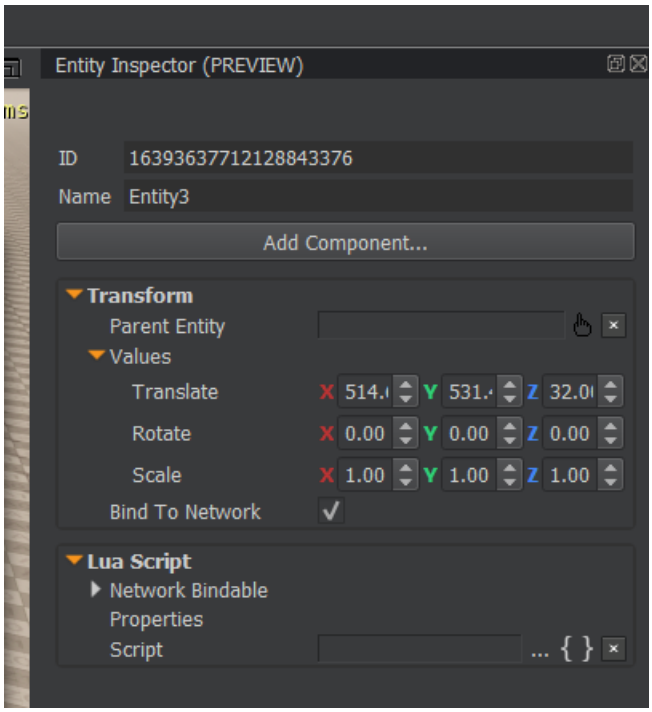
1. With the **Entity Inspector** view pane visible, select the entity in the viewport.
2. Click **Add Component**, and then open **Scripting**, **Lua Script**.



3. Scroll down to the **Scripting** section, and then click **Lua Script**.



4. A **Lua Script** component appears in the inspector. Use the file selection button (**...**) to select the Lua script from the file hierarchy that you want to use.



   You can select either a `.lua` file (a text copy of the original), or a `.luac` file (a precompiled version of the script). The functionality should be the same. The precompiled version is preferable because it loads faster and is usually smaller. However, you can use `.lua` files if you experience any issues.
5. After the script is loaded, click **Edit Script** (**{}**) to launch the Lua IDE and make changes to your script.

Basic Structure of a Component Entity Lua Script

Scripts to be used as components contain a table (referred to as the script table), which provides the functionality for the script. In Lua, this table is treated like a class. The script table generally consists of the following:

- An optional `Properties` table within the script table. The `Properties` table provides an interface that you can use to customize the script behavior from the editor.
- An `OnActivate()` function that the engine calls when the entity that has the script is activated.
- An `OnDeactivate()` function called by the engine when the entity that has the script is deactivated.

The following example shows a skeleton script.

```
-- ScriptName.lua

local ScriptName =
{
    Properties =
    {
        -- Property definitions
    }
}
```

```
function ScriptName:OnActivate()
     -- Activation Code
end

function ScriptName:OnDeactivate()
     -- Deactivation Code
end

return ScriptName
```

For each Lua script component, Lumberyard creates a table called the entity table. The script table in the referenced script is the metatable for the entity table. Because of this relationship, when any method in the script is called, the self parameter (implicit in most cases) refers to the entity table.

The entity table then has the following properties and methods available to it:

- A `Properties` table, copied from the script table's `Properties` table. Default values are provided where appropriate.
- An `entityId` property, which contains an object of type `EntityId` that refers to the current entity.
- An `IsMaster` function, callable by the script, to check whether the currently executing script is on the master node or a proxy node. This function is available only if the script component is network enabled.

Built-in Types and Methods

The Lumberyard engine provides a number of types and methods that are useful for making games. Many of the types and methods available are listed in the class view available in Lumberyard's Lua IDE. For more information on the class view, see Lua Editor.

Properties

The `Properties` table configures the editor interface for customizing the behavior of a script. With the properties table, users can modify numeric values, select states, and turn flags on and off. The table can even provide a reference to entities that your script can interact with.

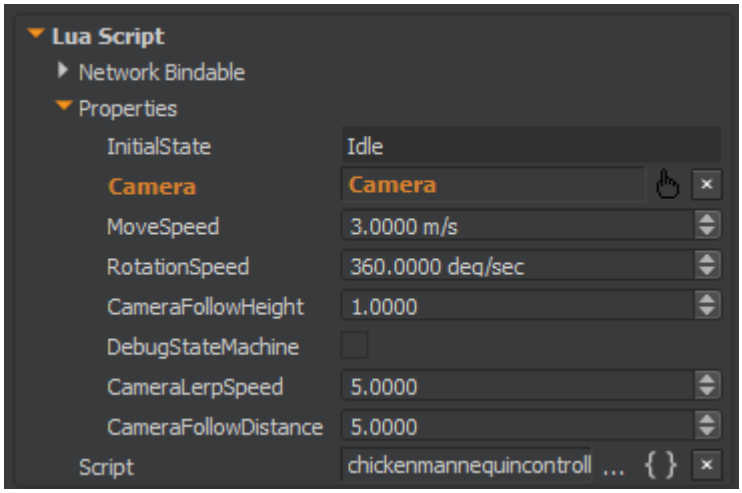
The properties inside the `Properties` table are exposed to the editor. Properties outside the `Properties` table are private and not displayed in the editor.

The following example is a properties table from the **Controllable Chicken** sample level.

```
-- Example Properties Table
local ChickenMannequinControllerSM =
{
    Properties =
    {
        MoveSpeed = { default = 3.0, description = "How fast the chicken moves.", suffix =
 " m/s" },
        RotationSpeed = { default = 360.0, description = "How fast (in degrees per second)
 the chicken can turn.", suffix = " deg/sec"},
        CameraFollowDistance = {default = 5.0, description = "Distance (in meters) from
 which camera follows character."},
        CameraFollowHeight = {default = 1.0, description = "Height (in meters) from which
 camera follows character."},
        CameraLerpSpeed = {default = 5.0, description = "Coefficient for how tightly camera
 follows character."},
        Camera = { default = EntityId() },
```

```
        InitialState = "Idle",
        DebugStateMachine = false,
    },
...
```

The result is the following **Properties** user interface in Lumberyard Editor:



The type that you provide as the default value determines how the property is appears in the editor user interface. You can further customize the representation of the property in the editor by specifying additional attributes in a table format. All property types support a description field that appears when you pause your mouse on the property name in the editor.

Supported Types

Properties can have the types described in this section.

Boolean Values (True, False)

The following examples are Boolean values.

```
DebugMovement = false,
AllowMovement = { default = true, description = "Allow or restrict movement of the
 object." },
```

In Lumberyard Editor, Boolean values are represented by a check box.

Numeric Values (Integer or Floating Point Numbers)

The following examples are numeric values.

```
Count = 5,
Velocity = { default = 1.0, suffix = "m/s", description = "Initial Velocity Of The
 Object" },
Distance = { default = 5.0, min = 2.0, max = 10.0, step = 2.0, suffix = "m", description =
 "The Distance An Object Can Travel In Meters" },
```

In Lumberyard Editor, numeric values are represented by an edit field with increase/decrease arrows. Numeric values can do the following:

- Provide a custom suffix to indicate units.

- Set minimum and maximum values.
- Provide a step value (how much the value increases or decreases when the user clicks the arrows on the right side of the edit field).

Strings

The following examples are strings.

```
DebugPrefix = "d_",
Name = { default = "Default Name", description = "The name of the entity" },
StartingState = { "Idle", description = "Specify the starting state. Valid starting states
 are Idle and Fidget" },
```

In Lumberyard Editor, string values are represented by a text edit box.

Reflected Classes

You can use any class that is reflected to both the `BehaviorContext` and the `EditContext` as a property. A good example of this is the `EntityId` type, which references other entities.

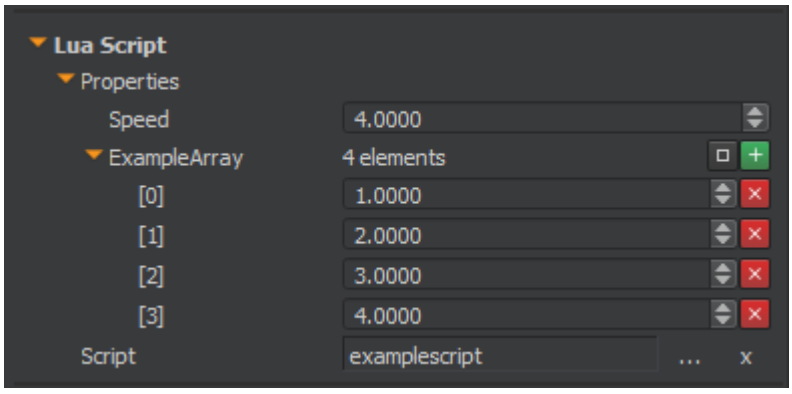
```
-- Entity Examples
ParentEntity = { default = EntityId(), description = "The Entity that this one will
 follow"},
Target = EntityId()
```

The editor representation is the default editor for the type reflected. For example, for `EntityId`, it's the entity reference picker. For most reflected types, it is a tree of the type's properties.

Arrays

Properties can contain resizable arrays of any of the types mentioned. To create the array, declare the default value as a keyless table of values. For example, the property definitions in the following code produce the properties shown in the image that follows.

```
local ExampleScript = {
      Properties = {
          Speed = 4,
          ExampleArray = { default = { 1, 2, 3, 4 } },
      }
}
return ExampleScript
```



Attributes

You can add attributes to a property by placing them alongside the default value in a property table. Attribute keys are not case sensitive. The following common attributes can be added to any property.

Common Attributes

Attribut	Description
`Descript`	A string that is the text of the tool tip for the property.
`UI`	Specifies (overrides) the UI handler that the property uses.

Network Binding

For network binding features to function, you must have a Net Binding Component (p. 998).

Properties

You can configure networking binding for properties by adding the `netSynched` table to the description of the variable inside of the `Properties` table.

```
local ExampleScript = {
    Properties = {
        Speed = {
            default = 0,    -- Supports numbers, strings, booleans, and nils for net
 bindings.
            min = 0,
            max = 100,
            step = 1,
            description = "Speed in m/s for the ...",

            -- If this table is missing, it is assumed the value is not networked.
            netSynched =
            {
                -- Optional fields
                OnNewValue = <function> -- OnNewValue is called whenever the property has a
                                        -- new value. OnNewValue accepts one
 parameter,which
                                        -- is the entity table for the instance that
 changed.

                -- The following flags are mainly here for debugging and profiling
 convenience.
                Enabled = true          -- Controls whether the field is network enabled.
 If
                                        -- missing, assumes true.
                ForceIndex = [1..32]    -- Profiling helper tool to force a property to use
 a
                                        -- specific DataSet to make understanding what data
 is
                                        -- being used where easier.
            }``
        }
    }
}
return ExampleScript
```

After you add networking to a property, any changes to the property are reflected across the network.

RPCs

Exposing RPCs to scripts involves creating a new table inside of the component table, but outside of the properties table, as shown in the following example.

```
local ExampleScript = {
    Properties = {
        -- ...
    },

    -- Table of remote procedure calls (RPCs) that the script wants to implement.
    NetRPCs =
    {
        RPCNoParam = {
            OnMaster = <function> -- The function to be called on the Master Script.
                                  -- The function should return a bool value that
                                  -- indicates whether or not proxy components can
                                  -- execute the RPC on themselves. Required.
            OnProxy = <function>  -- The function to be called on the Proxy Script.
                                  -- This function is optional and can be excluded if
                                  -- the master never allows proxies to execute the
 function call.
        }
    }
}

return ExampleScript
```

You can invoke the RPC just like any other function. There is no need to specify the `OnMaster/OnProxy` from the calling script. For example, you can call RPCs as in the following example.

```
self.NetRPCs.RPCNoParam()
self.NetRPCs.RPCParam(1.0)
```

Communicating with Components

Components provide interfaces that allow scripts to send them information and receive notifications when certain actions take place. Communication is established by creating two different objects in Lua: senders and handlers. A sender or a handler is an interface to an Event Bus (EBus), a communication system used extensively in the Lumberyard Engine. When a sender is created, it can call functions, which in turn send information to a component. When a handler is created, the component calls certain functions that the Lua script defines. These senders and handlers are created with an entity ID. You can use the entity ID to communicate with components that are attached to entities other than the one the script itself is running on. The main script table always provides a field called `entityId` that contains the ID of the entity to which the script is attached. Other entity IDs can be passed to the script through the `Properties` interface.

Order of Component Activation

Keep in mind the following points regarding the order of activation of Lua components:

- Lua components are activated after all C++ components have been activated.
- If an entity has multiple Lua components, there is no guarantee regarding which Lua component is activated first.

Registering with a Component to Receive Notifications

When a Lua script creates a handler object, it notifies a component attached to an entity that it should call the script handler functions when certain events occur. For example, in the first sample below, the script creates a Spawner Component notification bus handler when `OnActivate()` is called. This tells the spawner component attached to the entity that has the script to call the `OnSpawnBegin()`, `OnSpawnEnd()`, and `OnEntitySpawned()` functions when the spawner instantiates a new dynamic slice. Subsequently, the handler is explicitly disconnected and set back to nil in the `OnDeactivate` function. This ensures that processing time is not wasted when the entity attached to the script isn't active. As long as the entity is active, these functions are called by the spawner component at the appropriate time.

The following code example shows a spawner component handler.

```
local SpawnerScriptSample = { }

function SpawnerScriptSample:OnActivate()
    -- Register our handlers to receive notification from the spawner attached to this
 entity.
    if( self.spawnerNotiBusHandler == nil ) then
        self.spawnerNotiBusHandler = SpawnerComponentNotificationBus.CreateHandler(self,
 self.entityId)
    end
end

-- This handler is called when we start spawning a slice.
function SpawnerScriptSample:OnSpawnBegin(sliceTicket)
    -- Do something so we know if/when this is being called
    Debug.Log("Slice Spawn Begin")
end

-- This handler is called when we're finished spawning a slice.
function SpawnerScriptSample:OnSpawnEnd(sliceTicket)
    -- Do something so we know if/when this is being called
    Debug.Log("Slice Spawn End")
end

-- This handler is called whenever an entity is spawned.
function SpawnerScriptSample:OnEntitySpawned(sliceTicket, entityId)
    -- Do something so we know if/when this is being called
    Debug.Log("Entity Spawned: " .. tostring(entityId) )
end

function SpawnerScriptSample:OnDeactivate()
    -- Disconnect our spawner notificaton
    if self.spawnerNotiBusHandler ~= nil then
        self.spawnerNotiBusHandler:Disconnect()
        self.spawnerNotiBusHandler = nil
    end
end

return SpawnerScriptSample
```

Noncomponent Notifications

There are event buses that are available to Lua that are not associated with components. For example, a script can create a handler to receive notifications from the system's tick bus whenever the engine ticks. It provides both the amount of time that has passed since the last tick and the current time point. To gain access to this information, the script simply implements the `OnTick()` function and creates the handler.

The following example shows how to register an EBus.

```
local TestScript = { }
function TestScript:OnActivate()
    -- Inform the tick bus that you want to receive event notifications
    self.tickBusHandler = TickBus.CreateHandler(self)
    self.tickBusHandler:Connect()
end

-- This callback is called every frame by the tick bus after this entity activates
function TestScript:OnTick(deltaTime, timePoint)
    -- Add script to be executed every frame here...
end

function TestScript:OnDeactivate()
    -- Inform the tick bus that you no longer want to receive notifications
    self.tickBusHandler:Disconnect()
end

return TestScript
```

Sending Events to a Component

In addition to receiving notifications from components, a script must sometimes exercise control over components. Control is accomplished by sending events to components using the `Event` table and calling the functions implemented on it. In the example script that follows, the spawner component is sent an event that tells the component to spawn a dynamic slice by calling the `Spawn()` function. The first argument to an `Event` function is always the ID of the listener that you send the event to; the remaining arguments follow.

The following example shows how to send EBus events.

```
local SpawnerScript = { }
function SpawnerScript:OnActivate()
    SpawnerComponentRequestBus.Event.Spawn(self.entityId)
end

return SpawnerScript
```

You can request information from some event sending functions that return values. The next example script uses a `TransformBus` to get the current local transform of the entity and uses the `GetLocalTM()` function, which returns a transform object. This object is stored in a variable in the main script table. `TransformBus` is used again to reset the transform of the object to the identity.

The following example shows how to use the transform bus.

```
function samplescript:OnActivate()

    -- Retrieve the object's local transform and store it for later use
    self.myOldTransform = TransformBus.Event.GetLocalTM(self.entityId)

    -- Reset the object's local transform to the identity matrix
    TransformBus.Event.SetLocalTM(self.entityId, Transform.CreateIdentity())
end
```

Communicating with Components Attached to Other Entities

You can also send events and create handlers to communicate with components that are attached to other entities. The following example defines a parent entity in the properties table and requests its transform. This allows it to set its transform to that of another entity.

The following example code shows the use of a parent entity.

```
local ParentScriptSample=
{
    Properties =
    {
        ParentEntity = { entity="" },
    }
}

function ParentScriptSample:OnActivate()
    if self.Properties.ParentEntity.IsValid() then
        local parentTransform = TransformBus.Event.GetLocalTM(self.Properties.ParentEntity)
        TransformBus.Event.SetLocalTM(self.entityId, parentTransform)
    end
end

return ParentScriptSample
```

Using AZStd::vector and AZStd::array

Vectors and arrays in Lua behave very simarly to tables, with a few limitations. Both vector and array have the following features.

Length Operator #

You can obtain the length of a collection by prefixing the name of the collection with the length operator `#`, as in the following example.

```
#myCollection
```

Indexing []

To obtain the elements in a collection, use indexing in square brackets as the following syntax shows. Indexing is 1 based, just like Lua tables.

```
myCollection[index]
```

`Vector` also has the following methods for mutating the collection.

push_back

Use the `push_back` method to append elements to the vector, as in the following example.

```
myCollection:push_back(5)
```

pop_back

Use the `pop_back` method to remove the last element of the vector, as in the following example.

```
myCollection:pop_back()
```

clear

Use the `clear` method to remove all elements from the vector, as in the following example.

```
myCollection:clear()
```

Using AZStd::any

You can pass any Lua primitive type excluding tables to any bus or function that takes `AZStd::any` as a parameter (for example, `GameplayNotificationBus::OnEventBegin`). You can also pass any type reflected from C++ (for example, vectors or `EntityId` values). There is no syntax required to pass a value as an `any`—just call the bus or function.

The following example shows the use of `AZStd::any`.

```
GameplayNotificationBus.Broadcast.OnEventBegin(self.eventId, "The value I'd like to pass to
 the handler")
```

Debugging Scripts

Lumberyard provides Lua scripts with several functions to make debugging easier.

Logging to the Console

To print text to the Lumberyard Editor and game console, use the `Debug.Log()` function.

The following example shows the use of the `Debug.Log()` function.

```
local LoggingTest = { }
function LoggingTest:OnActivate()
    componentName = "MyComponent"
    Debug.Log(ComponentName .. " has been activated.")
end

return LoggingTest
```

Using an Assert to Detect Potential Issues

You can use the `assert()` or `Debug.Assert()` functions to display an error message in the console when conditions are detected that might result in an execution fault. The assert functions take two arguments: a condition that evaluates to true or false, and a message to display if the condition is false.

The following example shows the use of the `assert` and `Debug.Assert()` functions.

```
function SampleScript:DoStuff()
    -- This value should never be negative
    assert( self.positiveValue >= 0, "Expected a positive value! Got: " ..
 self.positiveValue )
end

-- Console output when the value of self.positiveValue is -5:
-- [Error] Lua error (2 - [string "q:/lyengine/branches/systems/dev/samplespro..."]:61:
 Expected a positive value! Got: -5) during call samplescript:DoStuff
```

```
-- ALTERNATIVE SYNTAX:

function SampleScript:DoStuff()
    -- This value should never be negative
    Debug.Assert( self.positiveValue >= 0, "Expected a positive value! Got: " ..
 self.positiveValue )
end

-- Console output when the value of self.positiveValue is -5:
-- [Error] Assert on argument 0: Expected a positive value! Got: -5
```

Communicating Errors

You can use the `Debug.Error()` function to display an error in the console and halt execution of the current script function. This does not halt all execution of the script. If you have active handlers, they can still be called when the engine posts notifications. The `Debug.Error()` function takes arguments similar to the `Debug.Assert` function: a condition and a message. The message is displayed in bright red and execution halts only if the condition is false.

The following example shows the use of the `Debug.Error()` function.

```
function SampleScript:CheckAndError()
    -- This value should never be negative
    Debug.Error( self.positiveValue >= 0, "Detected a negative value: " ..
 self.positiveValue )
end

-- Console output when the value of self.positiveValue is -5:
-- [Error] Error on argument 0: Detected a negative value: -5
```

Displaying a Warning When User Attention Is Required

A script condition can occur that does not adversely affect the execution of the script but might be useful for the user to know about. The `Debug.Warning()` function uses arguments similar to those of the `Error` and `Assert` functions but just displays an orange warning message in the console. It does not halt execution.

The following example shows the use of the `Debug.Warning()` function.

```
function SampleScript:CheckValue()
    -- This value should probably never be negative
    Debug.Warning( self.positiveValue >= 0, "Detected a negative value: " ..
 self.positiveValue )
end

-- Console output when the value of self.positiveValue is -5:
-- [Warning] Warning on argument 0: Detected a negative value: -5
```

The Lua Environment (Advanced)

By default, the Lumberyard component entity Lua environment is a single Lua environment (or `lua_State`). This environment is bound to the `BehaviorContext` that is owned by the `ComponentApplication`. Because of this, it has access to all API operations that are reflected on startup.

Adding Other VMs

You may add more `ScriptContext` instances using the `ScriptSystemBus` (either call `AddContextWithId`, or create your own and call `AddContext`). If you want your new context to be available for debugging, you must register it with `ScriptDebugAgentBus::RegisterContext`.

Reusing Code

Lua provides the capability to load and execute scripts from other Lua files using the built-in Lua require function. It's important to note that this function requires a special path format. The file path is delimited by periods instead of slashes, has no `.lua` file name extension, and is relative to the Lumberyard assets directory. For example, if you want to use the `require` function to give your scripts some common functionality from the project's `Scripts` directory, you can use code similar to the following example.

```
local library = require("Scripts.MyLibraryFile")
```

Lua Editor

Lua Editor is in preview release and is subject to change.

Lumberyard Lua Editor (Lua IDE) offers an intuitive integrated development environment (IDE) that makes it easy to author, debug, and edit Lua scripts when you create or extend your game. Lua Editor is a standalone application, but can be opened directly from Lumberyard Editor.

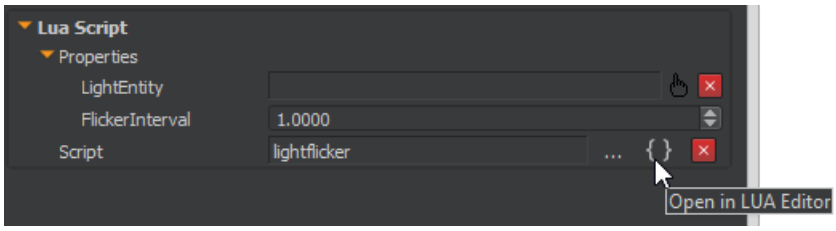
Tutorial: Using Lua Editor for Debugging with Lumberyard Editor

This tutorial shows you how to use Lumberyard Editor to create a sample level with a component entity that contains a Lua script component. You open the script in Lua Editor from Lumberyard Editor and perform some sample debugging steps on the script.

To use Lua Editor for debugging

1. In Lumberyard Editor, create a new level by performing one of the following steps:
   - In the **Welcome to Lumberyard Editor** window, click **New level**
   - Click **File**, **New**
   - Press **Ctrl+N**
2. In the **New Level** dialog box, give the level a name, and then click **OK**.
3. In the **Generate Terrain Texture** dialog box, click **OK** to accept the defaults.
4. Right-click the Lumberyard Editor viewport and select **Create Entity**.
5. In **Entity Inspector**, click **Add Component**, and then choose **Rendering**, **Point Light**.
6. In **Entity Inspector**, click **Add Component**, and then choose **Scripting**, **Lua Script**.
7. Scroll down to the bottom of the **Entity Inspector** window and, in the **Lua Script** section, click **...** to open the **Pick Lua Script** window.
8. In the **Pick Lua Script** window, expand **Samples Project**, **Scripts**,and **Components**.
9. Click **lightflicker.lua**, and then click **OK**. (Note: additional sample scripts are located in the Lumberyard directory `\dev\SamplesProject\Scripts`.)

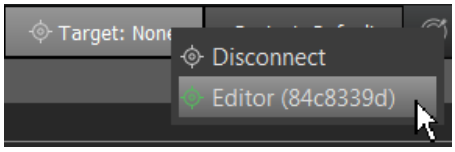
10. In **Entity Inspector**, in the **Lua Script** section, click the empty braces **{ }** to launch Lua Editor.



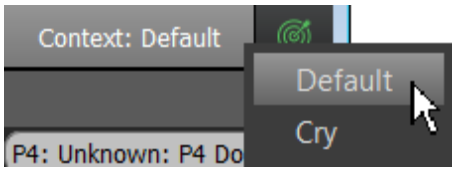
Because the debugging functionality is enabled through network sockets, you must connect Lua Editor to the target that is running the script before you can debug. In this tutorial, you connect to Lumberyard Editor.

> **Note**
> Connection is facilitated by GridHub (p. 1066), which is Lumberyard's central connection hub for debugging. GridHub starts automatically when Lua Editor is started and must be running in the background for Lua Editor to find targets it can connect to. If for some reason you need to start it manually, you can launch it from `\dev\Bin64\LuaIDE.exe`.

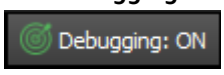
11. In the Lua Editor toolbar, click **Target: None**, and then click **Editor(*ID*)** to connect to Lumberyard Editor.



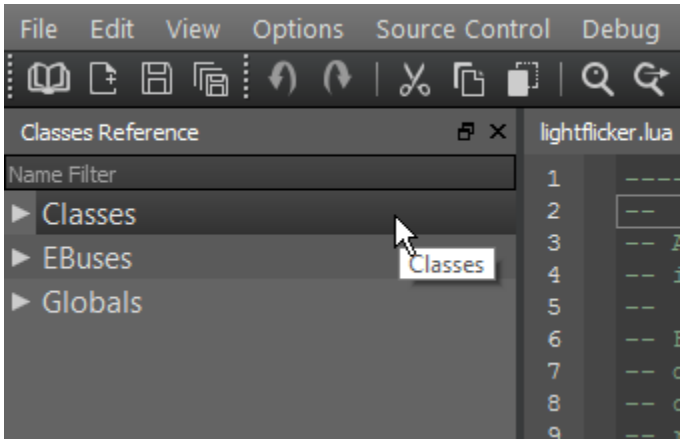
12. In the Lua Editor toolbar, leave **Context** setting at **Default** for the debugging context. The default setting is good for debugging component entity scripts such as the one in this tutorial. The **Cry** context option is for debugging legacy scripts such as those associated with Cry entities or the Game SDK.

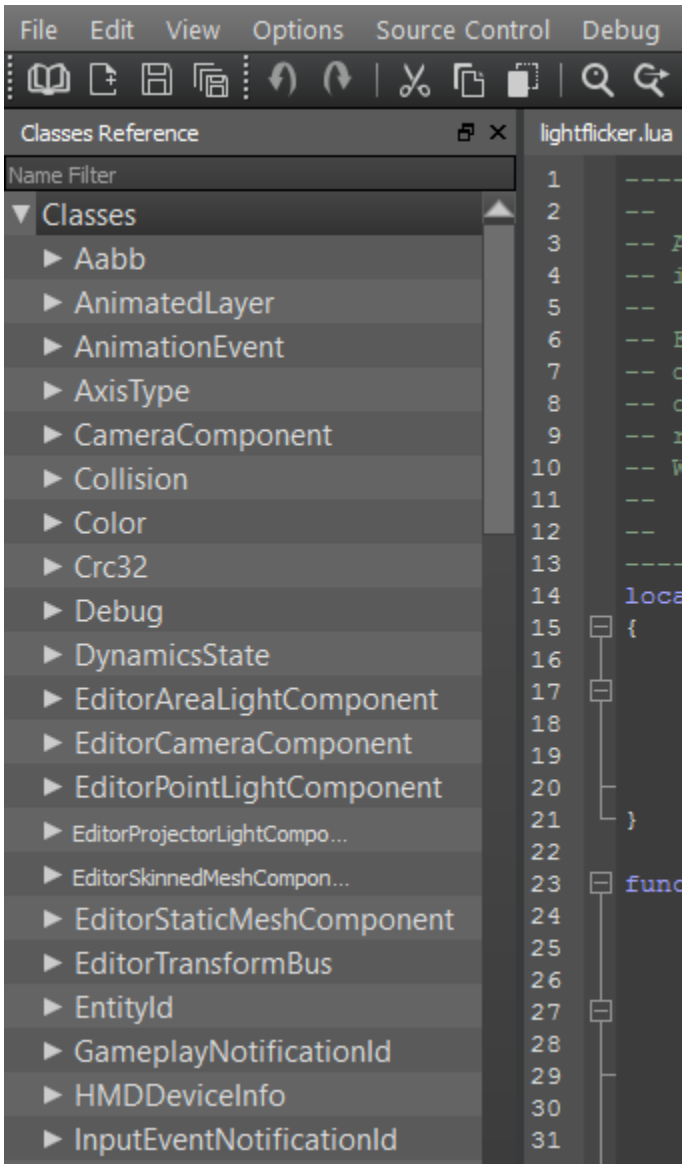


13. The **Debugging** icon turns green to show that Lua Editor and Lumberyard Editor are connected:



Click **Classes** in the **Class Reference** to show the available Lua libraries. You can do the same for **EBuses** and **Globals**.





Note
The class reference feature is active only for the default context and component entity scripts. This feature is not active in the Cry context, which exists only for backward compatibility.

After you connect, you can pause the execution of a given script by setting breakpoints.

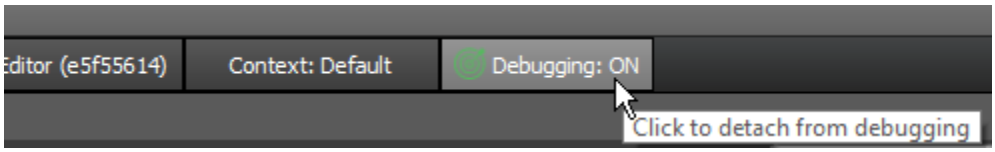
14. In the Lua Editor toolbar, click the **Breakpoints** icon to show the **Breakpoints** window.
15. In Lua Editor, single-click or double-click one or more line numbers in the `lightflicker.lua` script to set one or more breakpoints. As you add breakpoints, the line number and script path for each are added to the **Breakpoints** window.
16. In Lumberyard Editor, press **Ctrl+G** to run the game, or click **AI/Physics** at the bottom of the viewport to enable game simulation and run scripts. Lua Editor opens with a yellow marker stopped on the first breakpoint that it encounters.

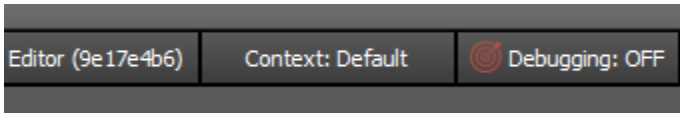
```
11  function lightflicker:OnActivate()
12
13      self.FlickerCountdown = self.Properties.FlickerInterval
14      local attachToEntity = self.entityId
15      if (self.Properties.LightEntity.id ~= 0) then
16          attachToEntity = self.Properties.LightEntity
17      end
```

When execution is halted at a breakpoint, more information becomes available in the **Lua Locals**, **Stack**, and **Watched Variables** panes.

17. Click the **Stack** icon to show the **Stack** window.
18. Click the **Lua Locals** icon to show local Lua variables.
19. Click **Watched Variables** icon to open the **Watched Variables** window, where you can specify variables to watch.
20. Press **F11** a few times to step through the code. Note how the contents of the **Stack**, **Lua Locals**, and **Watched Variables** windows change.

    Tip
    For greater convenience, you can float or dock these windows.
21. To detach from debugging, click **Debugging**.





22. In Lumberyard Editor, Press **Esc** to stop the game.

Options Available While Debugging

The following table summarizes common options available while debugging.

Icon	Action	Keyboard Shortcut	Description
	Run in Editor	Alt+F5	Run in Lumberyard Editor.
	Run on Target	Ctrl+F5	Send script to the connected target and run it.
	Run/Continue	F5	Run or continue running the current script.
	Step Into	F11	Step into the function called on the current line.

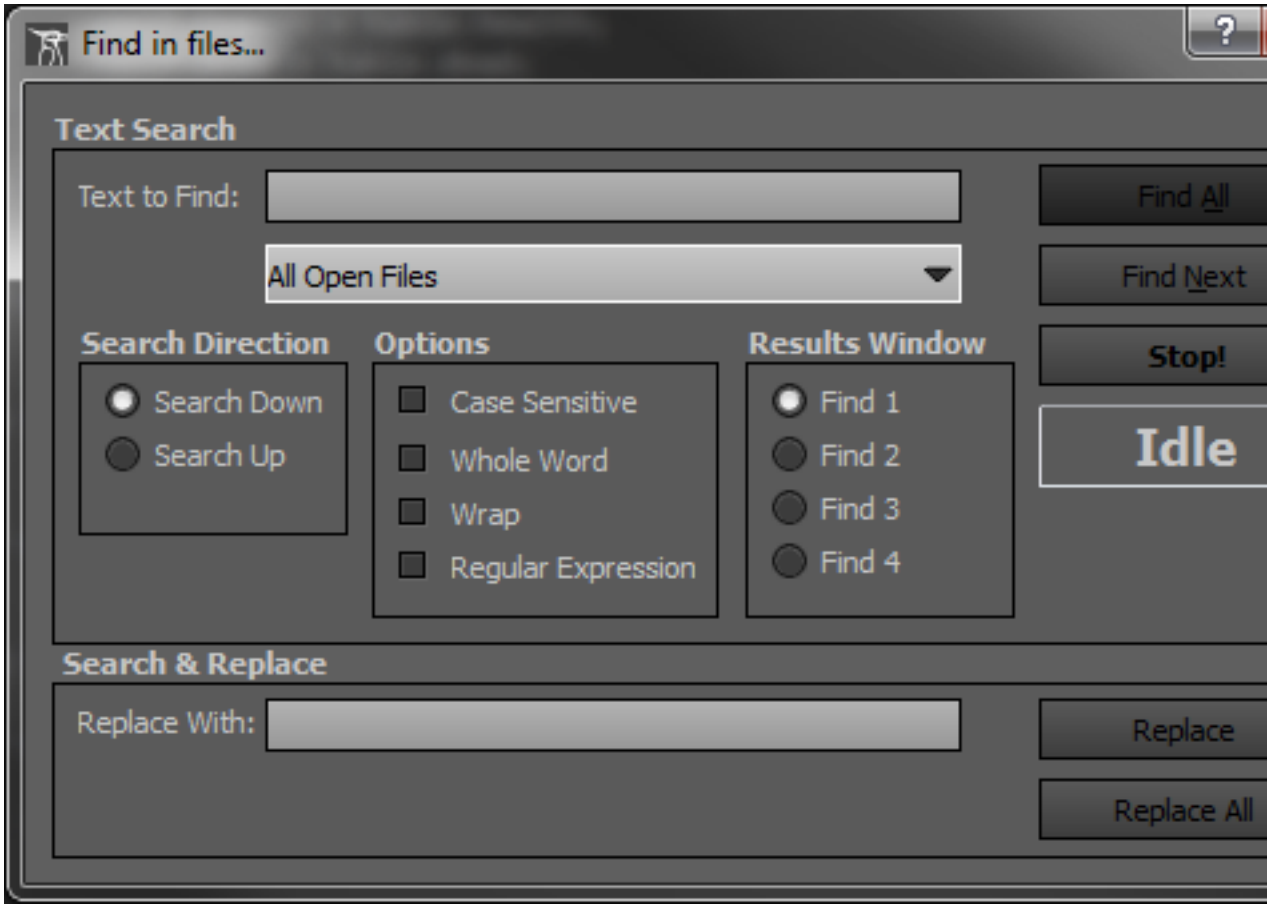
Icon	Action	Keyboard Shortcut	Description
	Step Out	Shift+F11	Step out of the called function.
	Step Over	F10	Step over the function called on the current line.
	Toggle Breakpoint	F9	Enable or disable a breakpoint on the current line.

Maintaining Separate Search Results

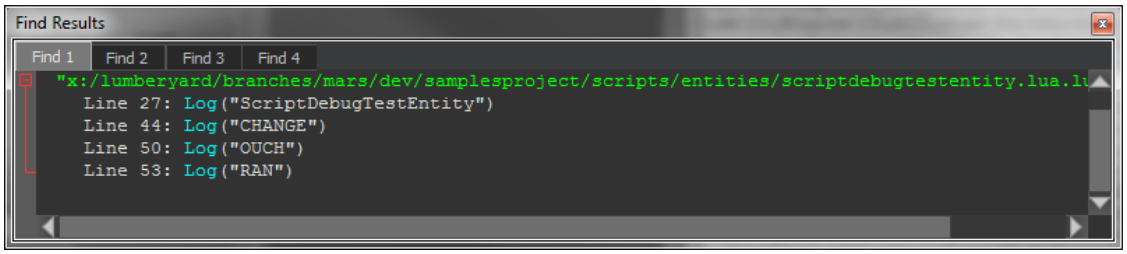
In addition to the usual search capabilities, the **Find** feature can display the results of four different searches separately.

To maintain separate search results

1. Click the **Find** icon or press **Ctrl+F** to perform searches in the currently open file, or in all open files.



2. Before starting a search, choose **Find 1**, **Find 2**, **Find 3**, or **Find 4** to choose the the window in which you want to see the results. You can maintain the results of four searches separately in the tabbed windows. The search results in the other windows remain unchanged.



3. To go directly to the line in the code which a search result was found, double-click the line in the search results.

   Note
   In Lua Editor Preview, the line number shown in the **Find Results** window and the line number in the script pane differ by one.

   Tip
   For convenience, you can also dock or float the **Find Results** window.

Editing

Lua Editor can open multiple scripts at the same time. Each script has its own tab in the editor. The editor provides a standard set of capabilities for text editing but also includes useful features for editing source code.

The following table summarizes the options available while editing and debugging.

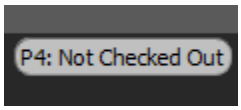
Action	Keyboard Shortcut
Comment selected block	Ctrl+K
Copy	Ctrl+C
Cut	Ctrl+X
Find	Ctrl+F
Find in open files	Ctrl+Shift+F
Find next	F3
Fold source functions	Alt+0
Go to line	Ctrl+G
Paste	Ctrl+V
Quick find local	Ctrl+F3
Quick find local reverse	Ctrl+Shift+F3
Redo	Ctrl+Y
Replace	Ctrl+R
Replace in open files	Ctrl+Shift+R

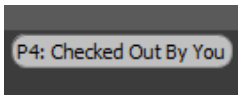
Action	Keyboard Shortcut
Select all	Ctrl+A
Select to brace[1]	Ctrl+Shift+]
Transpose lines down	Ctrl+Shift+Down Arrow
Transpose lines up	Ctrl+Shift+Up Arrow
Uncomment selected block	Ctrl+Shift+K
Undo	Ctrl+Z
Unfold source functions	Alt+Shift+0

[1] Select to brace selects a block bounded by braces. Before using this option, the cursor must be immediately next to the beginning or ending brace of the block.

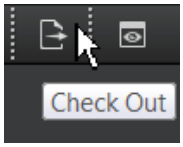
Perforce Integration

Lua Editor includes Perforce integration features. When you open a file from your Perforce environment, Lua Editor displays the file's status in the top right of the text editing window.





The **Source Control** menu offers **Check Out/Check In** functionality.



Component Entity Lua API Reference

This documentation is preliminary and subject to change.

You can use these Lua API calls for scripting the component entity system in Lumberyard. For Lua scripting functions that load and unload canvases in Lumberyard Editor, see the UI Lua Reference (p. 654).

BehaviorTreeComponentRequestBus

Represents a request submitted by a user of the current component.

StartBehaviorTree

Starts an inactive behavior tree associated with the current entity.

Syntax

```
void BehaviorTreeComponent::StartBehaviorTree()
```

StopBehaviorTree

Stops an active behavior tree associated with the current entity.

Syntax

```
void BehaviorTreeComponent::StopBehaviorTree()
```

GetVariableNameCrcs

Gets a list of cyclic redundancy check values for variable names.

Syntax

```
AZStd::vector<AZ::Crc32> GetVariableNameCrcs()
```

Returns: A list of the 32-bit cyclic redundancy check values for all variable names.

Return Type: `AZStd::vector`

Default Return: `s_defaultEmptyVariableIds`

GetVariableValue

Gets the value for the specified variable name CRC-32 checksum.

Syntax

```
bool GetVariableValue(AZ::Crc32 variableNameCrc)
```

Parameter	Type	Description
`variableNameCrc`	`AZ::Crc32`	The CRC-32 checksum for the variable name.

Returns: `true` if successful; otherwise, false.

Return Type: `bool`

Default Return: `false`

SetVariableValue

Set the value associated with a variable.

Syntax

```
void SetVariableValue(AZ::Crc32 variableNameCrc, bool
      newValue)
```

Parameter	Type	Description
`variableNameCrc`	`AZ::Crc32`	The CRC-32 checksum for the variable name.
`newValue`	`bool`	The new value for the variable.

NavigationComponentRequestBus

Requests serviced by the navigation component.

FindPathToEntity

Creates a path finding request to navigate towards the specified entity.

Syntax

```
PathfindRequest::NavigationRequestId FindPathToEntity(AZ::EntityId entityId)
```

Parameter	Type	Description
`entityId`	`AZ::EntityId`	Request `EntityId` of the entity we want to navigate towards.

Returns: A unique identifier to the pathfinding request.

Return Type: `PathfindRequest::NavigationRequestId`

Default Return: `PathfindResponse::kInvalidRequestId`

Stop

Stops all pathfinding operations for the specified `requestId`. The ID is used to make sure that the request being cancelled is the request that is currently being processed. If the specified `requestId` is different from the ID of the current request, the stop command can be safely ignored.

Syntax

```
void Stop(PathfindRequest::NavigationRequestId requestId)
```

Parameter	Type	Description
`requestId`	`PathfindRequest::NavigationRequestId`	ID of the request that is being cancelled.

NavigationComponentNotificationBus

Notifications sent by the Navigation component.

OnSearchingForPath

Indicates that the pathfinding request has been submitted to the navigation system.

Syntax

```
void OnSearchingForPath(PathfindRequest::NavigationRequestId requestId)
```

Parameter	Type	Description
requestId	PathfindRequest::NavigationRequestId	ID of the request for the path that is being searched.

OnTraversalStarted

Indicates that traversal for the indicated request has started.

Syntax

```
void OnTraversalStarted(PathfindRequest::NavigationRequestId requestId)
```

Parameter	Type	Description
requestId	PathfindRequest::NavigationRequestId	ID of the request for which traversal has started.

OnTraversalInProgress

Indicates that traversal for the indicated request has started.

Syntax

```
void OnTraversalInProgress(PathfindRequest::NavigationRequestId requestId, float
 distanceRemaining)
```

Parameter	Type	Description
requestId	PathfindRequest::NavigationRequestId	ID of the request for which traversal is in progress.
distanceRemaining	float	The remaining distance in the current path.

OnTraversalComplete

Indicates that traversal for the indicated request has completed successfully.

Syntax

```
void OnTraversalComplete(PathfindRequest::NavigationRequestId requestId)
```

Parameter	Type	Description
requestId	PathfindRequest::NavigationRequestId	ID of the request for which traversal has finished.

OnTraversalCancelled

Indicates that traversal for the indicated request was cancelled before it could be successfully completed.

Syntax

```
void OnTraversalCancelled(PathfindRequest::NavigationRequestId requestId)
```

Parameter	Type	Description
`requestId`	`PathfindRequest::NavigationRequestId`	ID of the request for which traversal was cancelled.

NavigationSystemRequestBus

Requests serviced by the navigation system component. This currently contains the single function `RayCast`.

RayCast

Creates a path finding request to navigate towards the specified entity.

Syntax

```
virtual NavRayCastResult RayCast(const AZ::Vector3& begin, const AZ::Vector3& direction,
 float maxDistance) { return NavRayCastResult() }
```

Parameter	Type	Description
`begin`	`Vector3`	The origin of the ray.
`direction`	`Vector3`	The direction for the ray to travel.
`maxDistance`	`float`	The maximum distance the ray travels.

Returns: A `NavRayCastResult`.

`NavRayCastResult` has the following structure.

Parameter	Type	Description
`bool m_collision`	`Boolean`	Returns true if there was a collision. The default is false.
`m_position`	`Vector3`	The position of the hit in world space. The default is `AZ::Vector3::CreateZero()`.
`m_meshId`	`NavigationMeshID`	The mesh ID of the navigation mesh hit. This is callable from Lua script. The default is 0.

AttachmentComponentRequestBus

Messages serviced by the `AttachmentComponent`. The `AttachmentComponent` lets an entity "stick" to a particular bone on a target entity.

Attach

Change the attachment target. The entity will detach from any previous target.

Syntax

```
void Attach(AZ::EntityId targetId, const char* targetBoneName, const AZ::Transform& offset)
```

Parameter	Type	Description
`targetId`	`AZ::EntityId`	Specifies the ID of the entity to attach to.
`targetBoneName`	`char`	Specifies the bone on the target entity to attach to. If the target bone is not found, then attach to the target entity's transform origin.
`offset`	`AZ::Transform`	The attachment's offset from the target.

Detach

Detaches an entity from its target.

Syntax

```
void Detach()
```

SetAttachmentOffset

Update an entity's offset from its target.

Syntax

```
void SetAttachmentOffset(const AZ::Transform& offset)
```

Parameter	Type	Description
`offset`	`AZ::Transform`	The offset from the target.

AttachmentComponentNotificationBus

This EBus interface handles events emitted by the `AttachmentComponent`. The `AttachmentComponent` lets an entity "stick" to a particular bone on a target entity.

OnAttached

The entity has attached to the target.

Syntax

```
void OnAttached(AZ::EntityId targetId)
```

Parameter	Type	Description
`targetId`	`AZ::EntityId`	The target being attached to.

OnDetached

The entity is detaching from the target.

Syntax

```
void OnDetached(AZ::EntityId targetId)
```

Parameter	Type	Description
`targetId`	`AZ::EntityId`	The target being detached from.

CharacterAnimationRequestBus

General character animation requests serviced by the `CharacterAnimationManager` component.

SetBlendParameter

Sets a custom blend parameter.

Syntax

```
void SetBlendParameter(AZ::u32 blendParameter, float value)
```

Parameter	Type	Description
`blendParameter`	`AZ::u32`	Corresponds to `EMotionParamID`.
`value`	`float`	The value to set.

SetAnimationDrivenMotion

Enables or disables animation-driven root motion.

Syntax

```
void SetAnimationDrivenMotion(bool useAnimDrivenMotion)
```

Parameter	Type	Description
`useAnimDrivenMotion`	`bool`	Specify `true` to enable animation-driven root motion; `false` to disable.

MannequinRequestsBus

Services provided by the Mannequin component.

QueueFragment

Queues the specified Mannequin fragment.

Syntax

```
FragmentRequestId QueueFragment(int priority, const char* fragmentName, const char*
 fragTags, bool isPersistent)
```

Parameter	Type	Description
`priority`	`int`	Specifies priority. A higher number means higher priority
`fragmentName`	`char`	Name of the fragment to be played.
`fragTags`	`char`	Fragment tags to be applied.
`isPersistent`	`bool`	Specifies persistence.

Returns: A request ID that can be used to identify and make modifications to the request.

Return Type: `FragmentRequestId`

Default Return: `MannequinRequests::s_invalidRequestId`

PauseAll

Pauses all actions being managed by the current Mannequin component

Syntax

```
void PauseAll()
```

ResumeAll

Resumes all actions being managed by the current Mannequin component.

Syntax

```
void ResumeAll(IActionController::EResumeFlags resumeFlag)
```

Parameter	Type	Description
`resumeFlag`	`IActionController::EResumeFlags`	Flag that indicates how the animations are to be resumed. See the `EResumeFlags` enum for possible values.

```
enum EResumeFlags
```

```
    {
        ERF_RestartAnimations              = BIT(0),
        ERF_RestoreLoopingAnimationTime    = BIT(1),
        ERF_RestoreNonLoopingAnimationTime = BIT(2),
        ERF_Default = ERF_RestartAnimations | ERF_RestoreLoopingAnimationTime |
ERF_RestoreNonLoopingAnimationTime
    };
```

SetTag

Sets the specified tag for the action controller.

Syntax

```
void SetTag(const char* tagName)
```

Parameter	Type	Description
tagName	char	The name of the tag to set.

ClearTag

Clears the specified tag for the action controller.

Syntax

```
void ClearTag(const char* tagName)
```

Parameter	Type	Description
tagName	char	The name of the tag to be cleared.

SetGroupTag

Sets a tag in the specified group.

Syntax

```
void SetGroupTag(const char* groupName, const char* tagName)
```

Parameter	Type	Description
groupName	char	The name of the group.
tagName	char	The name of the tag.

ClearGroup

Clears tags for the indicated group

Syntax

```
void ClearGroup(const char* groupName)
```

Parameter	Type	Description
groupName	char	The name of the group.

SetScopeContext

Sets the scope context for the current animation controller.

Syntax

```
void SetScopeContext(const char* scopeContextName, const AZ::EntityId entityId, const char*
 animationDatabaseName)
```

Parameter	Type	Description
scopeContextName	char	Name of the scope context that the animation database (.adb) file is to be attached to.
entityId	AZ::EntityId	Reference to an entity whose character instance will be bound to this scope context.
animationDatabaseName	char	The path to the animation database file.

ClearScopeContext

Clears the specified scope context.

Syntax

```
void ClearScopeContext(const char* scopeContextName)
```

Parameter	Type	Description
scopeContextName	char	Name of the scope context that is to be cleared.

StopRequest

Stops the actions associated with the specified request.

Syntax

```
void StopRequest(FragmentRequestId requestId)
```

Parameter	Type	Description
requestId	FragmentRequestId	Specifies the ID of the request for which actions should be stopped.

GetRequestStatus

Retrieves the status of the specified request

Syntax

```
IAction::EStatus GetRequestStatus(FragmentRequestId requestId)
```

Parameter	Type	Description
`requestId`	`FragmentRequestId`	The ID of the request to retrieve status for.

Returns: The status of the request.

Return Type: `IAction::EStatus`

Default Return: `IAction::EStatus::None`

ForceFinishRequest

Forces the actions associated with the specified request to finish.

Syntax

```
void ForceFinishRequest(FragmentRequestId requestId)
```

Parameter	Type	Description
`requestId`	`FragmentRequestId`	The ID of the request.

SetRequestSpeedBias

Sets the speed bias for the actions associated with the specified request.

Syntax

```
void SetRequestSpeedBias(FragmentRequestId requestId, float speedBias)
```

Parameter	Type	Description
`requestId`	`FragmentRequestId`	The request ID.
`speedBias`	`float`	The speed bias for this animation

GetRequestSpeedBias

Gets the speed bias for the actions associated with the specified request

Syntax

```
float GetRequestSpeedBias(FragmentRequestId requestId)
```

Parameter	Type	Description
`requestId`	`FragmentRequestId`	The ID of the request.

Returns: The speed bias for the indicated request.

Return Type: `float`

Default Return: `-1`

SetRequestAnimWeight

Sets the animation weight for the actions associated with the specified request.

Syntax

```
void SetRequestAnimWeight(FragmentRequestId requestId, float animWeight)
```

Parameter	Type	Description
`requestId`	`FragmentRequestId`	The ID of the request.
`animWeight`	`float`	The weight for the animation.

GetRequestAnimWeight

Gets the animation weight for the actions associated with the specified request.

Syntax

```
float GetRequestAnimWeight(FragmentRequestId requestId)
```

Parameter	Type	Description
`requestId`	`FragmentRequestId`	The ID of the request.

Returns: The animation weight for the indicated request.

Return Type: `float`

Default Return: `-1`

SimpleAnimationComponentRequestBus

Services provided by the Simple Animation component. The Simple Animation component provides basic animation functionality for the entity. If the entity has a mesh component with a skinned mesh attached (a .chr or .cdf file), the Simple Animation component will provide a list of all valid animations specified in the associated .chrparams file. The Simple Animation component does not provide interaction with Mannequin and should be used for light-weight environment or background animation.

StartDefaultAnimations

Plays the default animations along with default looping and speed parameters that were set up as a part of the current component. Components allow for multiple layers to be set up with defaults. The `StartDefaultAnimations` method starts the playback of all the default animations of the component.

Syntax

```
SimpleAnimationComponentRequests::Result StartDefaultAnimations()
```

Returns: A `Result` indicating whether the animations were started successfully.

Return Type: `SimpleAnimationComponentRequests::Result`

Default Return: `SimpleAnimationComponentRequests::Result::Failure`

StartAnimation

Starts playback of the animation of the specified `animatedLayer`.

Syntax

```
SimpleAnimationComponentRequests::Result StartAnimation(const AnimatedLayer& animatedLayer)
```

Parameter	Type	Description
`animatedLayer`	`AnimatedLayer`	A layer configured with the animation that is to be played on it.

Returns: A `Result` indicating whether the animation was started.

Return Type: `SimpleAnimationComponentRequests::Result`

Default Return: `SimpleAnimationComponentRequests::Result::Failure`

StartAnimationByName

Plays the animation with the specified name.

Syntax

```
Result StartAnimationByName(const char* name, AnimatedLayer::LayerId layerId)
```

Parameter	Type	Description
`name`	`char`	The name of the animation to play.
`layerId`	`AnimatedLayer::LayerId`	The layer in which to play the animation

Returns: A `Result` indicating whether the animation was started.

Return Type: `SimpleAnimationComponentRequests::Result`

Default Return: `SimpleAnimationComponentRequests::Result::Failure`

StopAllAnimations

Stops all animations that are being played on all layers.

Syntax

```
Result StopAllAnimations()
```

Returns: A `Result` indicating whether all animations were stopped.

Return Type: `SimpleAnimationComponentRequests::Result`

Default Return: `SimpleAnimationComponentRequests::Result::Failure`

StopAnimationsOnLayer

Stops the animation currently playing on the specified layer.

Syntax

```
Result StopAnimationsOnLayer(AnimatedLayer::LayerId layerId, float blendOutTime)
```

Parameter	Type	Description
`layerId`	`AnimatedLayer::LayerId`	Identifier for the layer that is to stop its animation (0 - `AnimatedLayer::s_maxActiveAnimatedLayers`)
`blendOutTime`	`float`	Time that the animations take to blend out.

Returns: A `Result` indicating whether the animation on the indicated layer was stopped.

Return Type: `SimpleAnimationComponentRequests::Result`

Default Return: `SimpleAnimationComponentRequests::Result::Failure`

SetPlaybackSpeed

Changes the playback speed for a particular layer.

Syntax

```
Result SetPlaybackSpeed(AnimatedLayer::LayerId layerId, float playbackSpeed)
```

Parameter	Type	Description
`layerId`	`AnimatedLayer::LayerId`	Identifier for the layer whose speed should be changed.
`playbackSpeed`	`float`	The playback speed.

Returns: A `Result` indicating whether the animation on the indicated layer was updated or not. A failure likely indicated that no animation is playing on the specified layer.

Return Type: `SimpleAnimationComponentRequests::Result`

Default Return: `SimpleAnimationComponentRequests::Result::Failure`

SimpleAnimationComponentNotificationBus

This EBus interfaces handles events sent by the simple animation component.

OnAnimationStarted

Informs all listeners about an animation being started on a layer.

Syntax

```
void OnAnimationStarted(const AnimatedLayer& animatedLayer)
```

Parameter	Type	Description
`animatedLayer`	`AnimatedLayer`	Specifies the name and parameters of the animation that was started.

OnAnimationStopped

Informs all listeners about an animation being stopped on the indicated layer

Syntax

```
void OnAnimationStopped(const AnimatedLayer::LayerId animatedLayer)
```

Parameter	Type	Description
`animatedLayer`	`AnimatedLayer::LayerId`	Specifies the name and parameters of the animation that was stopped.

AudioEnvironmentComponentRequestBus

This EBus interface handles messages serviced by `AudioEnvironmentComponent` instances.
The environment refers to the effects (primarily the auxiliary effects) that the bus sends. See `AudioEnvironmentComponent.cpp` for details.

SetAmount

Sets an environment amount on the default assigned environment.

Syntax

```
void SetAmount(float amount)
```

Parameter	Type	Description
`amount`	`float`	The amount for the environment.

SetEnvironmentAmount

Set an environment amount, specify an environment name at run time (that is, a script).

Syntax

```
void SetEnvironmentAmount(const char* environmentName,float amount)
```

Parameter	Type	Description
`environmentName`	`char`	The name of the environment.
`amount`	`float`	The amount for the environment.

AudioListenerComponentRequestBus

This EBus interface handles messages serviced by `AudioListenerComponent` instances.

SetRotationEntity

Sets the entity for which the audio listener tracks rotation.

Syntax

```
void SetRotationEntity(const AZ::EntityId entityId)
```

Parameter	Type	Description
`entityId`	`AZ::EntityId`	The ID of the entity.

SetPositionEntity

Sets the entity for which the audio listener tracks position.

Syntax

```
void SetPositionEntity(const AZ::EntityId entityId)
```

Parameter	Type	Description
`entityId`	`AZ::EntityId`	The ID of the entity.

SetFullTransformEntity

Essentially the same as calling `SetRotationEntity` and `SetPositionEntity` on the same entity.

Syntax

```
void SetFullTransformEntity(const AZ::EntityId entityId)
```

Parameter	Type	Description
entityId	AZ::EntityId	The ID of the entity.

AudioRtpcComponentRequestBus

This EBus interface handles messages serviced by AudioRtpcComponent instances. RTPC stands for Real-Time Parameter Control. The AudioRtpcComponent is used by the game to configure parameters in the audio engine. See AudioRtpcComponent.cpp for details.

SetValue

Sets an RTPC value for the RTPC name that has been serialized with the component.

Syntax

```
void SetValue(float value)
```

Parameter	Type	Description
value	float	The RTPC value to set.

SetRtpcValue

Use to manually specify an RTPC name and value at run time for use in scripting.

Syntax

```
void SetRtpcValue(const char* rtpcName, float value)
```

Parameter	Type	Description
rtpcName	char	Specifies an RTPC name to use.
value	float	Specifies a value for the RTPC name supplied.

AudioSwitchComponentRequestBus

This EBus interface handles messages serviced by AudioSwitchComponent instances. A Switch is an object that can be in one State at a time, but whose State value can be changed at run time. For example, a Switch called SurfaceMaterial might have states such as 'Grass', 'Snow', 'Metal', or 'Wood'. See AudioSwitchComponent.h for details.

SetState

Sets the name of the state on the default assigned switch.

Syntax

```
void SetState(const char* stateName)
```

Parameter	Type	Description
stateName	char	Specifies the name of the state to set.

SetSwitchState

Sets the specified switch to the specified state.

Syntax

```
void SetSwitchState(const char* switchName, const char* stateName)
```

Parameter	Type	Description
switchName	char	The name of the switch to set.
stateName	char	The name of the state to set on the specified switch.

AudioTriggerComponentRequestBus

This EBus interface handles messages serviced by `AudioTriggerComponent` instances. You can use the `AudioTriggerComponent` to execute, stop, and control ATL triggers. You can serialize the name of the trigger with the component or manually specify the name at run time for use in scripting. Only one `AudioTriggerComponent` is allowed on an entity, but the interface supports firing multiple ATL triggers.

Play

Executes the play trigger if the play trigger is set.

Syntax

```
void Play()
```

Stop

Executes the stop trigger if one is set; otherwise, stops the play trigger.

Syntax

```
void Stop()
```

ExecuteTrigger

Executes the specified ATL trigger.

Syntax

```
void ExecuteTrigger(const char* triggerName)
```

Parameter	Type	Description
`triggerName`	`char`	Specifies the name of the trigger to execute.

KillTrigger

Kills the specified ATL Trigger.

Syntax

```
void KillTrigger(const char* triggerName)
```

Parameter	Type	Description
`triggerName`	`char`	Specifies the name of the trigger to kill.

KillAllTriggers

Forces a kill of triggers that are active on the underlying proxy.

Syntax

```
void KillAllTriggers()
```

SetMovesWithEntity

Specifies whether the trigger should be repositioned as the entity moves.

Syntax

```
void SetMovesWithEntity(bool shouldTrackEntity)
```

Parameter	Type	Description
`shouldTrackEntity`	`bool`	Specify `true` to have the trigger track the entity. Specify `false` to have the trigger not track the entity.

AudioTriggerComponentNotificationBus

This EBus interface handles messages sent by `AudioTriggerComponent` instances.

OnTriggerFinished

Notifies when a trigger instance has finished.

Syntax

```
void OnTriggerFinished(const Audio::TAudioControlID triggerID)
```

Parameter	Type	Description
triggerID	Audio::TAudioControlID	The ID of the trigger.

FloatGameplayNotificationBus (AZ::GameplayNotificationBus<float>)

This version of the GameplayNotificationBus EBus interface handles float-based game play notifications.

OnGameplayEventAction

Event sent when the specified GameplayEventAction has occurred.

OnGameplayEventFailed

Event sent when the given GameplayEventAction has failed.

Vector3GameplayNotificationBus (AZ::GameplayNotificationBus<AZ::Vector3>)

This version of the GameplayNotificationBus EBus interface handles Vector3-based game play notifications.

OnGameplayEventAction

Event sent when the given GameplayEventAction has occurred.

OnGameplayEventFailed

Event sent when the given GameplayEventAction has failed.

StringGameplayNotificationBus (AZ::GameplayNotificationBus<const AZStd::string>)

This version of the GameplayNotificationBus EBus interface handles string-based game play notifications.

OnGameplayEventAction

Event sent when the given GameplayEventAction has occurred.

OnGameplayEventFailed

Event sent when the given GameplayEventAction has failed.

EntityIdGameplayNotificationBus (AZ::GameplayNotificationBus<AZ::EntityId>)

This EBus interface handles `EntityId`-based game play notifications. It is a specialization of the `GameplayNotificationBus`.

OnGameplayEventAction

Event sent when the given `GameplayEventAction` has occurred.

OnGameplayEventFailed

Event sent when the given `GameplayEventAction` has failed.

CryCharacterPhysicsRequestBus

This EBus interface handles messages serviced by Cry character physics.

Move

Requests movement from Living Entity.

Syntax

```
void Move(const AZ::Vector3& velocity, int jump)
```

Parameter	Type	Description
`velocity`	`AZ::Vector3`	Requested velocity (direction and magnitude).
`jump`	`int`	Controls how the value for the velocity parameter is applied within a Living Entity. To change the velocity to the new value, specify `1`. To add the value to the current velocity, specify `2`.

ConstraintComponentRequestBus

This EBus interface handles messages serviced by instances of the Constraint component. A Constraint component facilitates the creation of a physics constraint between two entities or an entity and a point in the world. Both entities must have a component that provides the physics service.

SetConstraintEntities

Sets the entity that owns the constraint and the target of the constraint.

Syntax

```
void SetConstraintEntities(const AZ::EntityId& owningEntity, const AZ::EntityId&
 targetEntity)
```

Parameter	Type	Description
owningEntity	AZ::EntityId	Specifies the ID of the entity that owns the constraint.
targetEntity	AZ::EntityId	Specifies the ID of the entity that is the target of the constraint. The target is invalid if constrained to world space.

SetConstraintEntitiesWithPartIds

Sets the entity that owns the constraint, the target entity, and the animation part IDs (bone IDs) for the constraint to be attached to.

Syntax

```
void SetConstraintEntitiesWithPartIds(const AZ::EntityId& owningEntity, int ownerPartId,
 const AZ::EntityId& targetEntity, int targetPartId)
```

Parameter	Type	Description
owningEntity	AZ::EntityId	Specifies the ID of the entity that owns the constraint.
ownerPartId	int	Specifies the ID of the owner part (the bone ID) for the constraint.
targetEntity	AZ::EntityId	Specifies the ID of the entity that is the target of the constraint.
targetPartId	int	Specifies the ID of the target part (the bone ID) for the constraint.

EnableConstraint

Enable all constraints on the current entity.

Syntax

```
void EnableConstraint()
```

DisableConstraint

Disable all constraints on the current entity.

Syntax

```
void DisableConstraint()
```

ConstraintComponentNotificationBus

This EBus interface handles messages dispatched by the Constraint component.

OnConstraintEntitiesChanged

This event fires when either the constraint owner or target changes. The target is invalid if constrained to world space.

> **Note**
> This event also fires when `partId` values change.

Syntax

```
void OnConstraintEntitiesChanged(const AZ::EntityId& oldOwner, const AZ::EntityId&
 oldTarget, const AZ::EntityId& newOwner, const AZ::EntityId& newTarget)
```

Parameter	Type	Description
`oldOwner`	`AZ::EntityId`	Specifies the ID of the entity that owned the constraint.
`oldTarget`	`AZ::EntityId`	Specifies the ID of the entity that was the target of the constraint.
`newOwner`	`AZ::EntityId`	Specifies the ID of the entity that is the new owner of the constraint.
`newTarget`	`AZ::EntityId`	Specifies the ID of the entity that is the new target of the constraint.

OnConstraintEnabled

Fires when constraints have been enabled on the current entity.

Syntax

```
void OnConstraintEnabled()
```

OnConstraintDisabled

Fires when a constraint has been disabled.

Syntax

```
void OnConstraintDisabled()
```

PhysicsComponentRequestBus

This EBus interface handles messages serviced by the in-game Physics component.

EnablePhysics

Makes the entity a participant in the physics simulation.

Syntax

```
void EnablePhysics()
```

DisablePhysics

Stops the entity from participating in the physics simulation

Syntax

```
void DisablePhysics()
```

IsPhysicsEnabled

Checks if physics are enabled on the current entity.

Syntax

```
bool IsPhysicsEnabled()
```

Returns: `true` if physics are enabled; `false` otherwise.

Return Type: `bool`

Default Return: `false`

AddImpulse

Applies the specified impulse to the entity.

Syntax

```
void AddImpulse(const AZ::Vector3& impulse)
```

Parameter	Type	Description
`impulse`	`AZ::Vector3`	Vector of the impulse.

AddAngularImpulse

Applies an angular impulse to the entity.

Syntax

```
void AddAngularImpulse(const AZ::Vector3& /*impulse*/, const AZ::Vector3& worldSpacePivot)
```

Parameter	Type	Description
`impulse`	`AZ::Vector3`	Vector of the impulse.
`worldSpacePivot`	`AZ::Vector3`	Vector of the world space pivot to apply to the entity.

GetVelocity

Retrieves the velocity of the entity.

Syntax

```
AZ::Vector3 GetVelocity()
```

Returns: The velocity of the entity.

Return Type: `AZ::Vector3`

Default Return: `AZ::Vector3::CreateZero`()

SetVelocity

Sets the velocity of the entity.

Syntax

```
void SetVelocity(const AZ::Vector3& velocity)
```

Parameter	Type	Description
`velocity`	`AZ::Vector3`	Specifies the velocity to set.

GetAcceleration

Gets the linear acceleration of the entity.

Syntax

```
AZ::Vector3 GetAcceleration()
```

Returns: A vector containing the linear acceleration of the entity.

Return Type: `AZ::Vector3`

Default Return: `AZ::Vector3::CreateZero`()

GetAngularVelocity

Gets the angular velocity of the entity.

Syntax

```
AZ::Vector3 GetAngularVelocity()
```

Returns: A vector containing the angular velocity of the entity.

Return Type: `AZ::Vector3`

Default Return: `AZ::Vector3::CreateZero()`

SetAngularVelocity

Sets the angular velocity of the entity to the specified amount.

Syntax

```
void SetAngularVelocity(const AZ::Vector3& angularVelocity)
```

Parameter	Type	Description
`angularVelocity`	`AZ::Vector3`	The angular velocity to set.

GetAngularAcceleration

Gets the angular acceleration of the entity

Syntax

```
AZ::Vector3 GetAngularAcceleration()
```

Returns: A vector containing the angular acceleration of the entity.

Return Type: `AZ::Vector3`

Default Return: `AZ::Vector3::CreateZero()`

GetMass

Retrieves the mass of the entity.

Syntax

```
float GetMass()
```

Returns: The mass of the entity.

Return Type: `float`

Default Return: `0.0f`

PhysicsComponentNotificationBus

This bus handles events emitted by a Physics component and by the Physics system.

OnPhysicsEnabled

Fires when an entity begins participating in the physics simulation. If the entity is active when a handler connects to the bus, then `OnPhysicsEnabled()` is immediately dispatched.

> **Note**
> If physics is enabled, `OnPhysicsEnabled` fires immediately upon connecting to the bus.

Syntax

```
void OnPhysicsEnabled()
```

OnPhysicsDisabled

Fires when an entity ends its participation in the physics simulation.

Syntax

```
void OnPhysicsDisabled()
```

OnCollision

Fires when an entity collides with another entity.

Syntax

```
void OnCollision(const Collision& collision)
```

Parameter	Type	Description
`collision`	`Collision`	Contains information about the collision that occurred. See the following `Collision` struct.

```
struct Collision
        {
            AZ_TYPE_INFO(Collision, "{33756BD4-24D4-4DAE-A849-537114D52F7D}");
            AZ_CLASS_ALLOCATOR(Collision, AZ::SystemAllocator, 0);

            AZ::EntityId m_entity;        // ID of other entity involved in event
            AZ::Vector3 m_position;       // Contact point in world coordinates
            AZ::Vector3 m_normal;         // Normal to the collision
            float m_impulse;              // Impulse applied by the collision resolver
            AZ::Vector3 m_velocityA;      // Velocities of the first entity involved in the
 collision
            AZ::Vector3 m_velocityB;      // Velocities of the second entity involved in
 the collision
            float m_massA;                // Masses of the first entity involved in the
 collision
            float m_massB;                // Masses of the second entity involved in the
 collision
        }
```

PhysicsSystemRequestBus

Requests for the physics system

RayCast

Casts a ray and retrieves a list of results.

Syntax

```
RayCastHit RayCast(const AZ::Vector3& begin, const AZ::Vector3& direction, float
 maxDistance, AZ::u32 maxHits, AZ::u32 query)
```

Parameter	Type	Description
`begin`	`const AZ::Vector3&`	The origin of the ray
`direction`	`const AZ::Vector3&`	The direction for the ray to travel
`maxDistance`	`float`	The maximum distance the ray will travel
`maxHits`	`AZ::u32`	The maximum number of hits found before the search is aborted
`query`	`AZ::u32`	The entity types to hit. See the `PhysicalEntityTypes` enum that follows.

Returns: A `RayCastHit` struct. For details, see the code listing that follows.

Return Type: `PhysicsSystemRequests::RayCastHit`

Default Return: `RayCastHit()`

```
struct RayCastHit
{
   AZ_TYPE_INFO(RayCastHit, "{3D8FA68C-A145-44B4-BA18-F3405D83A9DF}");
   AZ_CLASS_ALLOCATOR(RayCastHit, AZ::SystemAllocator, 0);

   float m_distance = 0.0f;    // The distance from RayCast begin to the hit.
   AZ::Vector3 m_position;     // The position of the hit in world space.
   AZ::Vector3 m_normal;       // The normal of the surface hit.
   AZ::EntityId m_entityId;    // The ID of the AZ::Entity hit, or
                               // AZ::InvalidEntityId if hit object is not an AZ::Entity.
};
```

RagdollPhysicsRequestBus

Messages serviced by the Cry character physics ragdoll behavior.

EnterRagdoll

Causes an entity with a skinned mesh component to disable its current physics and enable ragdoll physics.

Syntax

```
void EnterRagdoll()
```

ExitRagdoll

Causes the ragdoll component to deactivate itself and reenable the entity's physics component.

Syntax

```
void ExitRagdoll()
```

DecalComponentRequestBus

This EBus interface handles messages serviced by the Decal component.

SetVisibility

Specifies the decal's visibility.

Syntax

```
void SetVisibility(bool visible)
```

Parameter	Type	Description
`visible`	`bool`	Specify `true` to make the decal visible, `false` to hide it.

Show

Makes the decal visible.

Syntax

```
void Show()
```

Hide

Hides the decal.

Syntax

```
void Hide()
```

LensFlareComponentRequestBus

This EBus interface handles messages serviced by the Lens Flare component.

SetLensFlareState

Controls the lens flare state.

Syntax

```
void SetLensFlareState(State state)
```

Parameter	Type	Description
`state`	`State`	Specify `On` to turn on the lens flare; specify `Off` to turn it off.

TurnOnLensFlare

Turns the lens flare on.

Syntax

```
void TurnOnLensFlare()
```

TurnOffLensFlare

Turns the lens flare off.

Syntax

```
void TurnOffLensFlare()
```

ToggleLensFlare

Toggles the lens flare state.

Syntax

```
void ToggleLensFlare()
```

LensFlareComponentNotificationBus

This EBus interface handles events dispatched by the Lens Flare component.

LensFlareTurnedOn

Notifies that the lens flare has been turned on.

Syntax

```
void LensFlareTurnedOn()
```

LensFlareTurnedOff

Notifies that the lens flare has been turned off.

Syntax

```
void LensFlareTurnedOff()
```

LightComponentRequestBus

This EBus interfaces handles messages serviced by the light component.

SetLightState

Controls the light state.

Syntax

```
void SetLightState(State state)
```

Parameter	Type	Description
`state`	`State`	Specify `On` to turn on the light; specify `Off` to turn it off.

TurnOnLight

Turns the light on.

Syntax

```
void TurnOnLight()
```

TurnOffLight

Turns the light off.

Syntax

```
void TurnOffLight()
```

ToggleLight

Toggles the light state.

Syntax

```
void ToggleLight()
```

LightComponentNotificationBus

Light component notifications.

LightTurnedOn

Event sent when a light component is turned on.

Syntax

```
void LightTurnedOn()
```

LightTurnedOff

Event sent when a light component is turned off.

Syntax

```
void LightTurnedOff()
```

ParticleComponentRequestBus

Provides access to the particle component.

SetVisibility

Specifies the visibility of the particle component.

Syntax

```
void SetVisibility(bool visible)
```

Parameter	Type	Description
`visible`	`bool`	Specify `true` to make the particle component visible; `false` to hide it.

Show

Makes the particle component visible.

Syntax

```
void Show()
```

Hide

Hides the particle component.

Syntax

```
void Hide()
```

SetupEmitter

Sets up an effect emitter with the specified name and settings.

Syntax

```
void SetupEmitter(const AZStd::string& emitterName, const ParticleEmitterSettings&
 settings)
```

Parameter	Type	Description
`emitterName`	`const AZStd::string&`	The name of the emitter to set up.
`settings`	`const ParticleEmitterSettings&`	Contains particle emitter settings. For more information, see `ParticleComponent.cpp`.

SimpleStateComponentRequestBus

This EBus interface handles messages serviced by the Simple State component. The Simple State component provides a simple state machine. Each state is represented by a name and zero or more entities that are activated when the state is entered and deactivated when the state is left.

SetState

Sets the active state

Syntax

```
void SetState(const char* stateName)
```

Parameter	Type	Description
`stateName`	`char`	The name of the state.

SetStateByIndex

Sets the active state using a 0-based index.

Syntax

```
void SetStateByIndex(AZ::u32 stateIndex)
```

Parameter	Type	Description
`stateIndex`	`AZ::u32`	The 0-based index of the state.

SetToNextState

Advances to the next state. If the next state is null, the first state is set.

Syntax

```
void SetToNextState()
```

SetToPreviousState

Sets the previous state. If the previous state is null, the end state is set.

Syntax

```
void SetToPreviousState()
```

SetToFirstState

Sets the first state.

Syntax

```
void SetToFirstState()
```

SetToLastState

Sets the last state.

Syntax

```
void SetToLastState()
```

GetNumStates

Get the number of states.

Syntax

```
AZ::u32 GetNumStates()
```

Returns: The number of states.

Return Type: `AZ::u32`

Default Return: `0`

GetCurrentState

Gets the current state.

Syntax

```
const char* GetCurrentState()
```

Returns: The current state.

Return Type: `const char*`

Default Return: `nullptr`

SimpleStateComponentNotificationBus

This EBus interface handles events dispatched by the Simple State component.

OnStateChanged

Notify that the state has changed from `oldState` to `newState`.

Syntax

```
void OnStateChanged(const char* oldState, const char* newState)
```

Parameter	Type	Description
`oldState`	`char`	The name of the old state.

Parameter	Type	Description
`newState`	`char`	The name of the new state.

SpawnerComponentRequestBus

This EBus interface handles messages serviced by the `SpawnerComponent`.

Spawn

Spawns the selected slice at the entity's location.

Syntax

```
AzFramework::SliceInstantiationTicket Spawn()
```

Returns: A slice instantiation ticket.

Return Type: `AzFramework::SliceInstantiationTicket`

Default Return: `AzFramework::SliceInstantiationTicket`()

SpawnRelative

Spawns the selected slice at the entity's location with the specified relative offset.

Syntax

```
AzFramework::SliceInstantiationTicket SpawnRelative(const AZ::Transform& relative)
```

Parameter	Type	Description
`relative`	`AZ::Transform`	Relative offset from the entity's location.

Returns: A slice instantiation ticket.

Return Type: `AzFramework::SliceInstantiationTicket`

Default Return: `AzFramework::SliceInstantiationTicket`()

SpawnAbsolute

Spawns the selected slice at the specified world transform.

Syntax

```
AzFramework::SliceInstantiationTicket SpawnAbsolute(const AZ::Transform& world)
```

Parameter	Type	Description
`world`	const AZ::Transform&	Specifies the world transform at which to spawn the selected slice.

Returns: A slice instantiation ticket.

Return Type: `AzFramework::SliceInstantiationTicket`

Default Return: `AzFramework::SliceInstantiationTicket`()

SpawnerComponentNotificationBus

This EBus interface handles events dispatched by the `SpawnerComponent`.

OnSpawnBegin

Notifies that a slice has been spawned, but that its entities have not yet been activated. `OnEntitySpawned` events are about to be dispatched.

Syntax

```
void OnSpawnBegin(const AzFramework::SliceInstantiationTicket& ticket)
```

Parameter	Type	Description
`ticket`	`AzFramework::SliceInstantiationTicket`	The slice instantiation ticket.

OnSpawnEnd

Notifies that a spawn has been completed. All `OnEntitySpawned` events have been dispatched.

Syntax

```
void OnSpawnEnd(const AzFramework::SliceInstantiationTicket& ticket)
```

Parameter	Type	Description
`ticket`	`AzFramework::SliceInstantiationTicket`	The slice instantiation ticket.

OnEntitySpawned

Notifies that an entity has spawned. This event is called once for each entity spawned in a slice.

Syntax

```
void OnEntitySpawned(const AzFramework::SliceInstantiationTicket& ticket, const
 AZ::EntityId& spawnedEntities)
```

Parameter	Type	Description
`ticket`	`AzFramework::SliceInstantiationTicket`	The slice instantiation ticket.
`spawnedEntities`	`AZ::EntityId`	The ID of the spawned entity.

TagComponentRequestBus

Provides services for managing tags on entities.

HasTag

Checks for a specified tag on an entity.

Syntax

```
bool HasTag(const Tag&)
```

Parameter	Type	Description
`tag`	`Tag`	The tag to query for.

Returns: `true` if the entity has the specified tag; `false` otherwise.

Return Type: `bool`

Default Return: `false`

AddTag

Adds the specified tag to the entity if it doesn't already have it.

Syntax

```
void AddTag(const Tag&)
```

Parameter	Type	Description
`Tag`	`Tag`	The tag to add.

AddTags

Adds a specified list of tags to the entity if the list does not exist on the entity.

Syntax

```
void AddTags(const Tags& tags)
```

Parameter	Type	Description
`tags`	`Tags`	The list of tags to add.

RemoveTag

Removes a specified tag from the entity if the tag is present.

Syntax

```
void RemoveTag(const Tag&)
```

Parameter	Type	Description
`tag`	`Tag`	The tag to remove.

RemoveTags

Removes the specified list of tags from the entity if the list exists on the entity.

Syntax

```
void RemoveTags(const Tags& tags)
```

Parameter	Type	Description
`tags`	`Tags`	The list of tags to remove.

GetTags

Retrieves the list of tags on the entity.

Syntax

```
const Tags& GetTags()
```

Returns: A list of the tags on the entity.

Return Type: `static Tags`

Default Return: `s_emptyTags`

TagGlobalRequestBus

Provides services for querying Tags on entities.

RequestTaggedEntities

Queries for tagged entities. Handlers respond if they have the tag (that is, they are listening on the tag's channel). Use `AZ::EbusAggregateResults` to handle more than the first responder.

Syntax

```
const AZ::EntityId RequestTaggedEntities()
```

Returns: The ID of an entity that has a tag.

Return Type: `const AZ::EntityId`

Default Return: `s_invalidEntityId`

TagGlobalNotificationBus

Handler for global Tag component notifications.

OnEntityTagAdded

Notifies that a tag has been added to an entity. When connecting to the tag global notification bus, your `OnEntityTagAdded` handler fires once for each entity that already has a tag. After the initial connection, you are alerted whenever a new entity gains or loses a tag.

Syntax

```
void OnEntityTagAdded(const AZ::EntityId&)
```

OnEntityTagRemoved

Notifies that a Tag has been removed from an entity.

Syntax

```
void OnEntityTagRemoved(const AZ::EntityId&)
```

TagComponentNotificationsBus

Provides notifications regarding tags on entities.

OnTagAdded

Notifies listeners when a tag has been added.

Syntax

```
void OnTagAdded(const Tag&)
```

OnTagRemoved

Notifies listeners when a tag is removed.

Syntax

```
void OnTagRemoved(const Tag&)
```

TriggerAreaRequestsBus

This EBus interface services requests made to the Trigger Area component.

AddRequiredTag

Adds a required tag to the activation filtering criteria of the current component.

Syntax

```
void AddRequiredTag(const Tag& requiredTag)
```

Parameter	Type	Description
requiredTag	Tag	The tag to add to the activation filtering criteria.

RemoveRequiredTag

Removes a required tag from the activation filtering criteria of the current component.

Syntax

```
void RemoveRequiredTag(const Tag& requiredTag)
```

Parameter	Type	Description
requiredTag	Tag	The tag to remove from the activation filtering criteria.

AddExcludedTag

Adds an excluded tag to the activation filtering criteria of the current component.

Syntax

```
void AddExcludedTag(const Tag& excludedTag)
```

Parameter	Type	Description
excludedTag	Tag	The excluded tag to add to the activation filtering criteria.

RemoveExcludedTag

Removes an excluded tag from the activation filtering criteria of the current component.

Syntax

```
void RemoveExcludedTag(const Tag& excludedTag)
```

Parameter	Type	Description
excludedTag	Tag	The excluded tag to remove from the activation filtering criteria.

TriggerAreaNotificationBus

This EBus handles events for a given trigger area when an entity enters or leaves.

OnTriggerAreaEntered

Notifies when an entity enters the trigger area.

Syntax

```
void OnTriggerAreaEntered(AZ::EntityId enteringEntityId)
```

Parameter	Type	Description
`enteringEntityId`	`AZ::EntityId`	The ID of the entity that entered the trigger area.

OnTriggerAreaExited

Notifies when an entity exits the trigger area.

Syntax

```
void OnTriggerAreaExited(AZ::EntityId exitingEntityId)
```

Parameter	Type	Description
`exitingEntityId`	`AZ::EntityId`	The ID of the entity that exited the trigger area.

TriggerAreaEntityNotificationBus

Events fired for a specified trigger when the trigger area has been entered or exited.

OnEntityEnteredTriggerArea

Notifies when an `enteringEntityId` instance has entered the specified trigger area.

Syntax

```
void OnEntityEnteredTriggerArea(AZ::EntityId triggerId)
```

Parameter	Type	Description
`triggerId`	`AZ::EntityId`	The ID of the trigger that has been entered.

OnEntityExitedTriggerArea

Notifies when an `enteringEntityId` instance has exited the specified trigger area.

Syntax

```
void OnEntityExitedTriggerArea(AZ::EntityId triggerId)
```

Parameter	Type	Description
`triggerId`	`AZ::EntityId`	The ID of the trigger that has been exited.

BoxShapeComponentRequestsBus

Services provided by the Box Shape component.

GetBoxConfiguration

Retrieves the box configuration.

Syntax

```
BoxShapeConfiguration GetBoxConfiguration()
```

Return Type: `BoxShapeConfiguration`

Default Return: `BoxShapeConfiguration()`

SetBoxDimensions

Sets new dimensions for the Box Shape.

Syntax

```
void SetBoxDimensions(AZ::Vector3 newDimensions)
```

Parameter	Type	Description
`newDimensions`	`AZ::Vector3`	Specifies dimensions along the X, Y, and Z axes.

CapsuleShapeComponentRequestsBus

Services provided by the Capsule Shape Component.

GetCapsuleConfiguration

Retrieves the capsule configuration.

Syntax

```
CapsuleShapeConfiguration GetCapsuleConfiguration()
```

Returns: The capsule configuration.

Return Type: `CapsuleShapeConfiguration`

Default Return: `CapsuleShapeConfiguration()`

SetHeight

Sets the end to end height of capsule, including the cylinder and both caps.

Syntax

```
void SetHeight(float newHeight)
```

Parameter	Type	Description
`newHeight`	`float`	Specifies the new height of the capsule.

SetRadius

Sets the radius of the capsule.

Syntax

```
void SetRadius(float newRadius)
```

Parameter	Type	Description
`newRadius`	`float`	Specifies the new radius of the capsule.

CylinderShapeComponentRequestsBus

This EBus interface handles messages for the Cylinder Shape component.

GetCylinderConfiguration

Retrieves the cylinder configuration.

Syntax

```
CylinderShapeConfiguration GetCylinderConfiguration()
```

Returns: The cylinder configuration.

Return Type: `CylinderShapeConfiguration`

Default Return: `CylinderShapeConfiguration`()

SetHeight

Sets the height of the cylinder.

Syntax

```
void SetHeight(float newHeight)
```

Parameter	Type	Description
`newHeight`	`float`	Specifies the height of the cylinder.

SetRadius

Sets the radius of the cylinder.

Syntax

```
void SetRadius(float newRadius)
```

Parameter	Type	Description
`newRadius`	`float`	Specifies the radius of the cylinder.

ShapeComponentRequestsBus

Handles requests for services provided by the Shape component.

GetShapeType

Retrieves the type of shape of a component.

Syntax

```
AZ::Crc32 GetShapeType()
```

Returns: A Crc32 value that indicates the type of shape of the current component.

Return Type: `AZ::Crc32`

Default Return: `AZ::Crc32()`

IsPointInside

Checks if a given point is inside or outside a shape.

Syntax

```
bool IsPointInside(const AZ::Vector3& point)
```

Parameter	Type	Description
`point`	`AZ::Vector3`	Specifies the coordinates of the point to be tested.

Returns: A `bool` value that indicates whether the point is inside or out.

Return Type: `bool`

Default Return: `false`

DistanceFromPoint

Retrieves the minimum distance the specified point is from the shape.

Syntax

```
float DistanceFromPoint(const AZ::Vector3& point)
```

Parameter	Type	Description
`point`	`AZ::Vector3`	Specifies the coordinates of the point from which to calculate distance.

Returns: A float that indicates the distance the point is from the shape.

Return Type: `float`

Default Return: `0.f`

DistanceSquaredFromPoint

Retrieves the minimum squared distance the specified point is from the shape.

Syntax

```
float DistanceSquaredFromPoint(const AZ::Vector3& point)
```

Parameter	Type	Description
`point`	`AZ::Vector3`	Specifies the coordinates of the point from which to calculate the squared distance.

Returns: A float that contains the minimum squared distance the specified point is from the shape.

Return Type: `float`

Default Return: `0.f`

ShapeComponentNotificationsBus

Notifications sent by the shape component.

OnShapeChanged

Notifies that the shape component has been modified.

Syntax

```
void OnShapeChanged(ShapeChangeReasons changeReason)
```

Parameter	Type	Description
changeReason	ShapeChangeReasons	Informs listeners of the reason for this shape change (transform change, the shape dimensions being altered.)

SphereShapeComponentRequestsBus

Services provided by the Sphere Shape Component

GetSphereConfiguration

Retrieves the sphere configuration.

Syntax

```
SphereShapeConfiguration GetSphereConfiguration()
```

Returns: The sphere configuration.

Return Type: SphereShapeConfiguration

Default Return: SphereShapeConfiguration()

SetRadius

Sets the specified radius for the sphere shape component.

Syntax

```
void SetRadius(float newRadius)
```

Parameter	Type	Description
newRadius	float	Specifies the radius of the sphere shape.

EntityBus

Dispatches events specific to a given entity.

OnEntityActivated

Notifies when entity activation has completed. If the entity is active when a handler connects to the bus, then the OnEntityActivated event is sent immediately.

Syntax

```
void OnEntityActivated(const AZ::EntityId&)
```

OnEntityDeactivated

Notifies when the entity is about to be deactivated.

Syntax

```
void OnEntityDeactivated(const AZ::EntityId&)
```

TickBus

Tick events are executed on the main game or component thread.

> **Note**
> Warning: Adding mutex to the tick bus degrades performance in most cases.

OnTick

Notifies the delta time if the delta from the previous tick (in seconds) and time point is its absolute value.

Syntax

```
void OnTick(float deltaTime, ScriptTimePoint time)
```

Parameter	Type	Description
`deltaTime`	`float`	The latest time between ticks.
`time`	`ScriptTimePoint`	The time at the current tick.

TickRequestBus

Make requests from this bus to get the frame time or return the current time as seconds.

GetTickDeltaTime

Gets the latest time between ticks.

Syntax

```
float GetTickDeltaTime()
```

Returns: The latest time between ticks.

Return Type: `float`

Default Return: `0.f`

GetTimeAtCurrentTick

Gets the time in seconds at the current tick.

Syntax

```
ScriptTimePoint GetTimeAtCurrentTick()
```

Returns: The time in seconds at the current tick.

Return Type: `ScriptTimePoint`

Default Return: `ScriptTimePoint()`

TransformNotificationBus

This EBus is a listener for transform changes.

OnTransformChanged

Notifies when the local transform of the entity has changed. A local transform update always implies a world transform change.

Syntax

```
void OnTransformChanged(const Transform& local, const Transform& world)
```

Parameter	Type	Description
`local`	`Transform`	The local transform of the entity.
`world`	`Transform`	The world transform.

OnParentChanged

Notifies when the parent of an entity has changed. When the old or new parent is invalid, the invalid `EntityId` is equal to `InvalidEntityId`.

Syntax

```
void OnParentChanged(EntityId oldParent, EntityId newParent)
```

Parameter	Type	Description
`oldParent`	`EntityId`	The entity ID of the old parent.
`newParent`	`EntityId`	The entity ID of the new parent.

GameEntityContextRequestBus

This EBus interfaces makes requests to the game entity context component.

DestroyGameEntity

Destroys an entity. The entity is deactivated immediately and is destroyed in the next tick.

Syntax

```
void DestroyGameEntity(const AZ::EntityId& id)
```

Parameter	Type	Description
id	AZ::EntityId	The ID of the entity to be destroyed.

DestroyGameEntityAndDescendants

Destroys an entity and all its descendants, the entity and its descendants are deactivated immediately and will be destroyed the next tick.

Syntax

```
void DestroyGameEntityAndDescendants(AZ::EntityId& id)
```

Parameter	Type	Description
id	AZ::EntityId	The ID of the entity to be destroyed. The entity's descendants will also be destroyed.

ActivateGameEntity

Activates an entity by the specified ID.

Syntax

```
void ActivateGameEntity(AZ::EntityId& id)
```

Parameter	Type	Description
id	AZ::EntityId	The ID of the entity to activate.

DeactivateGameEntity

Deactivates an entity by the specified ID.

Syntax

```
void DeactivateGameEntity(AZ::EntityId& id)
```

Parameter	Type	Description
id	AZ::EntityId	The ID of the entity to deactivate.

DestroySliceByEntity

Destroys the slice instance that contains the entity with the specified ID.

Syntax

```
bool DestroySliceByEntity(AZ::EntityId& id)
```

Parameter	Type	Description
id	AZ::EntityId	

Returns: `true` if the slice instance was successfully destroyed.

Return Type: `bool`

Default Return: `false`

RandomManagerBus

Provides functions for random numbers.

RandomFloat

Generates a random float value.

Syntax

```
float RandomFloat()
```

Parameter	Type	Description
tag	AZStd::string	The tag.

Returns: A random value between [0.0f, 1.0f).

Return Type: `float`

Default Return: `0.0f`

RandomBool

Generates a random Boolean value.

Syntax

```
bool RandomBool(const AZStd::string& tag)
```

Parameter	Type	Description
tag	AZStd::string	The tag.

Returns: A random Boolean value.

Return Type: `bool`

Default Return: `false`

RandomInt

Generates a random unsigned integer value.

Syntax

```
unsigned int RandomInt(const AZStd::string& tag)
```

Parameter	Type	Description
`tag`	`AZStd::string`	The tag.

Returns: A random unsigned integer value.

Return Type: `unsigned int`

Default Return: `0`

RandomInRange

Generates a random unsigned integer value within a specified range.

Syntax

```
unsigned int RandomInRange(const AZStd::string& tag, unsigned int min, unsigned int max)
```

Parameter	Type	Description
`tag`	`AZStd::string`	The tag.
`min`	`unsigned int`	The minimum value that can be returned.
`max`	`unsigned int`	The maximum value that can be returned.

Returns: A random unsigned integer value within the specified range.

Return Type: `unsigned int`

Default Return: `0`

CameraRequestBus

Provides access to camera properties and services.

GetFov

Gets the camera's field of view in degrees

Syntax

```
float GetFOV()
```

Returns: The camera's field of view as a float.

Return Type: `float`

Default Return: `s_defaultFoV`

GetNearClipDistance

Gets the camera's distance from the near clip plane in meters.

Syntax

```
float GetNearClipDistance()
```

Returns: The camera's distance from the near clip plane as a float in meters.

Return Type: `float`

Default Return: `s_defaultNearPlaneDistance`

GetFarClipDistance

Gets the camera's distance from the far clip plane in meters.

Syntax

```
float GetFarClipDistance()
```

Returns: The camera's distance from the far clip plane as a float in meters.

Return Type: `float`

Default Return: `s_defaultFarClipPlaneDistance`

GetFrustumWidth

Gets the camera frustum's width.

Syntax

```
float GetFrustumWidth()
```

Returns: The camera frustum's width as a float.

Return Type: `float`

Default Return: `s_defaultFrustumDimension`

GetFrustumHeight

Gets the camera frustum's height.

Syntax

```
float GetFrustumHeight()
```

Returns: The camera frustum's height as a float.

Return Type: `float`

Default Return: `s_defaultFrustumDimension`

SetFov

Sets the camera's field of view in degrees.

Syntax

```
void SetFov(float fov)
```

Parameter	Type	Description
`fov`	`float`	The field of view in degrees. Possible values are 0 < `fov` < 180.

SetNearClipDistance

Sets the near clip plane to the specified distance from the camera in meters.

Syntax

```
void SetNearClipDistance(float nearClipDistance)
```

Parameter	Type	Description
`nearClipDistance`	`float`	The distance from the camera in meters. The value should be small, but greater than 0.

SetFarClipDistance

Sets the far clip plane to the specified distance from the camera in meters.

Syntax

```
void SetFarClipDistance(float farClipDistance)
```

Parameter	Type	Description
`farClipDistance`	`float`	The distance from the camera in meters.

SetFrustumWidth

Sets the camera frustum's width.

Syntax

```
void SetFrustumWidth(float width)
```

Parameter	Type	Description
`width`	`float`	The camera frustum's width.

SetFrustumHeight

Sets the camera frustum's height.

Syntax

```
void SetFrustumHeight(float height)
```

Parameter	Type	Description
`height`	`float`	The camera frustum's height.

MakeActiveView

Makes the camera the active view.

Syntax

```
void MakeActiveView()
```

HttpClientComponentNotificationBus

Event handler for Http requests.

OnHttpRequestSuccess

Notifies when an HTTP request is successful.

Syntax

```
void OnHttpRequestSuccess(int responseCode, AZStd::string responseBody)
```

Parameter	Type	Description
`responseCode`	`int`	The response code.
`responseBody`	`AZStd::string`	The body of the response.

OnHttpRequestFailure

Sent when an HTTP request failed.

Syntax

```
void OnHttpRequestFailure(int responseCode)
```

Parameter	Type	Description
responseCode	int	The response code.

HttpClientComponentRequestBus

Provides services to make HTTP requests.

MakeHttpRequest

Makes an HTTP request.

Syntax

```
void MakeHttpRequest(AZStd::string url, AZStd::string method, AZStd::string jsonBody)
```

Parameter	Type	Description
url	AZStd::string	The request URL.
method	AZStd::string	The HTTP request method.
jsonBody	AZStd::string	The JSON body of the request.

HMDDeviceRequestBus

HMD device bus used to communicate with the rest of the engine. Every device supported by the engine lives in its own Gem and supports this bus. A device wraps the underlying SDK into a single object for easy use by the rest of the system. Every device created should register with the EBus in order to be picked up as a usable device during initialization by the EBus function `BusConnect`().

GetTrackingState

Gets the most recent HMD tracking state.

Syntax

```
TrackingState* GetTrackingState()
```

Returns: The tracking state.

Return Type: `TrackingState*`

Default Return: `nullptr`

RecenterPose

Center the current pose for the HMD based on the current direction in which the viewer is looking.

Syntax

```
void RecenterPose()
```

SetTrackingLevel

Set the current tracking level of the HMD. Supported tracking levels are defined in struct `TrackingLevel`.

Syntax

```
void SetTrackingLevel(const AZ::VR::HMDTrackingLevel level)
```

Parameter	Type	Description
`level`	`AZ::VR::HMDTrackingLevel`	The tracking level to use with the current HMD. Possible values: `kHead` - The sensor reads as if the player is standing. `kFloor` - The sensor reads as if the player is seated or on the floor.

OutputHMDInfo

Outputs the information about the currently connected HMD (contained in the `HMDDeviceInfo` object) to the console and log file.

Syntax

```
void OutputHMDInfo()
```

GetDeviceInfo

Get the device info object for this particular HMD.

Syntax

```
HMDDeviceInfo* GetDeviceInfo()
```

Returns: A pointer to the current HMD's `HMDDeviceInfo` struct.

Return Type: `HMDDeviceInfo*`

Default Return: `nullptr`

IsInitialized

Gets whether or not the HMD has been initialized. The HMD has been initialized when it has fully established an interface with its required SDK and is ready to be used.

Syntax

```
bool IsInitialized()
```

Returns: `true` if the device has been initialized and is usable; otherwise, returns `false`.

Return Type: `bool`

Default Return: `false`

ControllerRequestBus

Provides information about HMD device controllers.

GetTrackingState

Returns a `TrackingState` object that contains tracking info about a connected controller. For more information, see struct TrackingState.

Syntax

```
TrackingState* GetTrackingState(ControllerIndex controllerIndex)
```

Parameter	Type	Description
`controllerIndex`	`int`	Specify 0 for the left controller; 1 for the right controller.

Returns: A pointer to the `TrackingState` object for the connected controller.

Return Type: `TrackingState*`

Default Return: `nullptr`

IsConnected

Returns whether the specified controller is connected.

Syntax

```
bool IsConnected(ControllerIndex controllerIndex)
```

Parameter	Type	Description
`controllerIndex`	`int`	Specify 0 for the left controller; 1 for the right controller.

Returns: A Boolean that indicates whether the specified controller is connected.

Return Type: `bool`

Default Return: `false`

VideoPlaybackRequestBus

Provides access to video playback services.

Play

Start or resume playing a movie that is attached to the current entity.

Syntax

```
void Play()
```

Pause

Pause a movie that is attached to the current entity.

Syntax

```
void Pause()
```

Stop

Stop playing a movie that is attached to the current entity.

Syntax

```
void Stop()
```

EnableLooping

Set whether or not the movie attached to the current entity loops.

Syntax

```
void EnableLooping(bool enable)
```

Parameter	Type	Description
`enable`	`bool`	Specify `true` to loop; `false` to not loop.

IsPlaying

Returns whether or not the video is currently playing

Syntax

```
bool IsPlaying()
```

Returns: `true` if the video is currently playing; `false` if the video is paused or stopped.

Return Type: `bool`

Default Return: `false`

SetPlaybackSpeed

Sets the playback speed based on a factor of the current playback speed.

Syntax

```
void SetPlaybackSpeed(float speedFactor)
```

Parameter	Type	Description
`speedFactor`	`float`	The speed modification factor to apply to playback speed. For example, specify 0.5f to play at half speed or 2.0f to play at double speed.

VideoPlaybackNotificationBus

This bus contains event handlers for video playback services.

OnPlaybackStarted

Event that fires when the movie starts playback.

Syntax

```
void OnPlaybackStarted()
```

OnPlaybackPaused

Event that fires when the movie pauses playback.

Syntax

```
void OnPlaybackPaused()
```

OnPlaybackStopped

Event that fires when the movie stops playback.

Syntax

```
void OnPlaybackStopped()
```

OnPlaybackFinished

Event that fires when the movie completes playback.

Syntax

```
void OnPlaybackFinished()
```

VR Lua Functions

You can use Lua bindings to interact programmatically with head-mounted display (HMD) devices that provide Virtual Reality (VR) experiences.

For general information on configuring your Lumberyard game project for VR, see Virtual Reality in the Amazon Lumberyard User Guide.

Global Functions

The following functions provide programming interfaces for HMD devices.

Function	Description
`HMDDeviceRequestBusSender HMDDeviceRequestBusSender(EntityId)`	Returns an `HMDDeviceRequestBusSender` object that is connected to the specified entity. For more information, see HMDDeviceRequestBus (p. 652).
`ControllerRequestBusSender ControllerRequestbusSender(EntityId)`	Returns a `ControllerRequestBusSender` object that is connected to the specified entity. For more information, see ControllerRequestBus (p. 653).

HMDDeviceRequestBus

Contains functions that return information about an HMD and control its pose and tracking level.

Function	Description
`Bool IsInitialized()`	Returns `true` if an HMD has successfully initialized on the bus. Returns `false` if no HMD is connected or failed to initialize.
`Void RecenterPose()`	Causes the direction that the HMD is currently facing to be considered forward'.
`Void OutputHMDInfo()`	Outputs the information about the currently connected HMD (contained in the HMDDeviceInfo object) to the console and log file.
`Void SetTrackingLevel(int)`	Sets the tracking level for the HMD. 0 specifies head level tracking (the player is standing); 1 is floor level tracking (the player is seated or on the floor).
`HMDDeviceInfo GetDeviceInfo()`	Returns an `HMDDeviceInfo` object that contains device information about a connected HMD. For more information, see struct HMDDeviceInfo (p. 653).
`TrackingState GetTrackingState()`	Returns a `TrackingState` object that contains the most recent tracking information about a connected HMD. For more information, see struct TrackingState (p. 653).

ControllerRequestBus

Returns status information about an HMD controller.

Function	Description
`Bool IsConnected(int controllerIndex)`	Returns true if the given controller is connected, false if a controller is not connected. Pass 0 for the left controller, pass 1 for the right controller.
`TrackingState GetTrackingState(int controllerIndex)`	Returns a `TrackingState` object that contains tracking info about a connected controller. Pass 0 for the left controller, pass 1 for the right controller. For more information, see struct TrackingState (p. 653).

struct HMDDeviceInfo

Contains information about a device that displays on the screen when the device is detected.

Field	Description
`String productName`	Name of the connected HMD. The default is `nullptr`.
`String manufacturer`	Name of the company that manufactured the connected HMD. The default is `nullptr`.
`Int renderWidth`	The render width for the HMD in pixels. This is normally half the full resolution of the device (rendering is per eye). The default is `0`.
`Int renderHeight`	The render height in pixels for a single eye of the HMD. The default is `0`.
`Float fovH`	The horizontal field of view for both eyes in radians. The default is `0.0f`.
`Float fovV`	The vertical field of view in radians. The default is `0.0f`.

struct TrackingState

Stores position and connection state information about the HMD. When an HMD is in use, certain parts of the device can go offline or online. For example, a controller can be disconnected, or the HMD can temporarily lose rotational tracking. You can use the `TrackingState` to determine what part of the pose is currently valid.

Field	Description
`PoseState pose`	The position and orientation in object space of the HMD. For more information, see struct PoseState (p. 654).
`DynamicsState dynamics`	Contains the current state of the physics dynamics for the current device such as linear velocity, angular velocity, and acceleration. For more information, see struct DynamicsState (p. 654).
`Int statusFlags`	Bit field that describes the current tracking state. For bit flags, see the enum HMDStatus (p. 654).

struct PoseState

A specific pose of the HMD device. Each HMD device has its own way of representing its current pose in three dimensional space. This structure acts as a common data set between a connected device and the rest of the system. All data is in a local coordinate space.

Field	Description
`Quaternion orientation`	A quaternion representing the current orientation in object space of the HMD.
`Vector3 position`	A three dimensional vector representing the current position of the HMD in object space as an offset from the centered pose.

struct DynamicsState

Dynamics (accelerations and velocities) of the current HMD. Many HMDs have the ability to track the current movements of VR devices for prediction. Not all devices support velocities and accelerations. All data is in a local coordinate space.

Field	Description
`Vector3 angularVelocity`	A three dimensional vector representing angular velocity in object space.
`Vector3 angularAcceleration`	A three dimensional vector representing angular acceleration in object space.
`Vector3 linearVelocity`	A three dimensional vector representing linear velocity in object space.
`Vector3 linearAcceleration`	A three dimensional vector representing linear acceleration in object space.

enum HMDStatus

The following code shows the status flags for HMDStatus.

```
enum HMDStatus
{
    HMDStatus_OrientationTracked = BIT(1),
    HMDStatus_PositionTracked = BIT(2),
    HMDStatus_CameraPoseTracked = BIT(3),
    HMDStatus_PositionConnected = BIT(4),
    HMDStatus_HmdConnected = BIT(5),
    HMDStatus_IsUsable = HMDStatus_HmdConnected | HMDStatus_OrientationTracked,
    HMDStatus_ControllerValid = HMDStatus_OrientationTracked | HMDStatus_PositionConnected,
};
```

UI Lua Reference

You can use the following Lua scripting functions when you work with UI canvases in Lumberyard Editor. For non-UI component entity Lua scripting functions, see the Component Entity Lua API Reference.

Topics

- LyShineLua.ShowMouseCursor (p. 655)
- UIButtonComponent (p. 655)
- UICanvasComponent (p. 656)
- UICanvasManager (p. 662)
- UICheckboxComponent (p. 662)
- UIDynamicLayoutComponent (p. 664)
- UIDynamicScrollBoxComponent (p. 665)
- UIElementComponent (p. 666)
- UIFaderComponent (p. 668)
- UIImageComponent (p. 669)
- UIInteractableComponent (p. 671)
- UILayoutColumnComponent (p. 678)
- UILayoutGridComponent (p. 679)
- UILayoutRowComponent (p. 682)
- UIMaskComponent (p. 683)
- UIScrollBarComponent (p. 684)
- UIScrollBoxComponent (p. 687)
- UISliderComponent (p. 693)
- UITextComponent (p. 696)
- UITextInputComponent (p. 700)
- UITooltipComponent (p. 704)
- UITooltipDisplayComponent (p. 704)
- UITransform2dComponent (p. 707)
- UI World Components (p. 710)

LyShineLua.ShowMouseCursor

Controls the visibility of the mouse cursor. `1` specifies that the mouse cursor is displayed; `0` specifies that it is hidden.

Syntax

```
void ShowMouseCursor(bool visible)
```

UIButtonComponent

You can use a button component to make an element behave like a button.

UiButtonBus

Services messages for the `UiButtonComponent`.

GetOnClickActionName

Returns the action triggered when the button is released.

Syntax

```
AZStd::string UiButtonBus::GetOnClickActionName()
```

SetOnClickActionName

Sets the action triggered when the button is released.

Syntax

```
void UiButtonBus::SetOnClickActionName(const AZStd::string& actionName)
```

UiButtonNotificationBus

Services notifications for the `UiButtonComponent`.

OnButtonClick

Occurs when a button has been clicked.

Syntax

```
void OnButtonClick()
```

UICanvasComponent

Contains buses and notification buses for UI canvas and animation.

UiCanvasBus

Services messages for the `UiCanvasComponent`.

GetDrawOrder

Returns the draw order on the canvas.

Syntax

```
int GetDrawOrder()
```

SetDrawOrder

Sets the draw order on the canvas.

Syntax

```
void SetDrawOrder(int drawOrder)
```

GetKeepLoadedOnLevelUnload

Returns `true` if canvas stays loaded when the level unloads; `false` otherwise.

Syntax

```
bool GetKeepLoadedOnLevelUnload()
```

SetKeepLoadedOnLevelUnload

Sets whether the canvas stays loaded when the level unloads. Specify true to keep the canvas loaded on level unload, false otherwise.

Syntax

```
void SetKeepLoadedOnLevelUnload(bool keepLoaded)
```

GetNumChildElements

Returns the number of child elements that the canvas has.

Syntax

```
int GetNumChildElements()
```

GetChildElement

Returns the entity ID of the child element that has the specified index.

Syntax

```
AZ::EntityId GetChildElement(int index)
```

GetChildElements

Returns a list of entity IDs of the child elements of the canvas.

Syntax

```
AZStd::vector<AZ::EntityId> GetChildElements()
```

FindElementByName

Returns the entity ID of a canvas element that has the specified name.

Syntax

```
AZ::EntityId FindElementByName(const AZStd::string& name)
```

GetIsPixelAligned

Returns true if the visual element's vertices snap to the nearest pixel; false otherwise.

Syntax

```
bool GetIsPixelAligned()
```

SetIsPixelAligned

Specify true to make the element's vertices snap to the nearest pixel, false otherwise.

Syntax

```
void SetIsPixelAligned(bool isPixelAligned)
```

GetEnabled

Returns the enabled flag of the canvas. True if the canvas is enabled; false otherwise. Enabled canvases are updated and rendered each frame.

Syntax

```
bool GetEnabled()
```

SetEnabled

Sets whether the canvas is enabled. Enabled canvases are updated and rendered each frame. Specify true to enable the canvas, false to disable.

Syntax

```
void SetEnabled(bool enabled)
```

GetIsRenderToTexture

Returns true if the canvas is rendered to a texture instead of the full screen; false otherwise.

Syntax

```
bool GetIsRenderToTexture()
```

SetIsRenderToTexture

Sets whether the canvas is rendered to a texture instead of the full screen. Specify true to render the canvas to a texture, false otherwise.

Syntax

```
void SetIsRenderToTexture(bool isRenderToTexture)
```

GetRenderTargetName

Returns the name of the texture that is created when the current canvas renders to a texture.

Syntax

```
AZStd::string GetRenderTargetName()
```

SetRenderTargetName

Specifies the name of the texture that is created when the current canvas renders to a texture.

Syntax

```
void SetRenderTargetName(const AZStd::string& name)
```

GetIsPositionalInputSupported

Returns true if the canvas accepts input from mouse or touch input; false otherwise.

Syntax

```
bool GetIsPositionalInputSupported()
```

SetIsPositionalInputSupported

Specify true to make the canvas accept input from mouse or touch, false otherwise.

Syntax

```
void SetIsPositionalInputSupported(bool isSupported)
```

GetIsNavigationSupported

Returns true if the canvas accepts navigation input; false otherwise.

Syntax

```
bool GetIsNavigationSupported()
```

SetIsNavigationSupported

Specifies true to make the canvas accept navigation input, false otherwise.

Syntax

```
void SetIsNavigationSupported(bool isSupported)
```

GetTooltipDisplayElement

Returns the entity ID of the tooltip element that is to be displayed when the mouse pauses on an interactable element.

Syntax

```
AZ::EntityId GetTooltipDisplayElement()
```

SetTooltipDisplayElement

Sets the element that is to be displayed when hovering over an interactable.

Syntax

```
void SetTooltipDisplayElement(AZ::EntityId entityId)
```

UiCanvasNotificationBus

Services notifications for the `UiCanvasComponent`.

OnAction

Specifies the name of an action and the entity ID of a canvas element that triggers the action.

Syntax

```
void OnAction(AZ::EntityId entityId, const AZStd::string& actionName)
```

UiAnimationBus

Animation messages serviced by the `UiCanvasComponent`.

StartSequence

Starts playing the specified sequence.

Syntax

```
void StartSequence(const AZStd::string& sequenceName)
```

StopSequence

Stop playing the specified sequence.

Syntax

```
void StopSequence (const AZStd::string& sequenceName)
```

AbortSequence

Abort the specified sequence.

Syntax

```
void AbortSequence (const AZStd::string& sequenceName)
```

PauseSequence

Pause the specified sequence.

Syntax

```
void PauseSequence (const AZStd::string& sequenceName)
```

ResumeSequence

Resume the specified sequence.

Syntax

```
void ResumeSequence (const AZStd::string& sequenceName)
```

ResetSequence

Reset the specified sequence to the start.

Syntax

```
void ResetSequence (const AZStd::string& sequenceName)
```

GetSequencePlayingSpeed

Returns the current playing speed of the specified sequence.

Syntax

```
float GetSequencePlayingSpeed(const AZStd::string& sequenceName)
```

SetSequencePlayingSpeed

Sets the current playing speed of the specified sequence.

Syntax

```
void SetSequencePlayingSpeed(const AZStd::string& sequenceName, float speed)
```

GetSequencePlayingTime

Returns the current playing time of the specified sequence.

Syntax

```
float GetSequencePlayingTime(const AZStd::string& sequenceName)
```

IsSequencePlaying

Returns true if the specified sequence is playing; false otherwise.

Syntax

```
bool IsSequencePlaying(const AZStd::string& sequenceName)
```

UiAnimationNotificationBus

Services animation notifications for the `UiCanvasComponent`.

OnUiAnimationEvent

Specifies the animation event for the specified sequence.

Syntax

```
void OnUiAnimationEvent(eUiAnimationEvent uiAnimationEvent, AZStd::string animSequenceName)
```

Following are possible values for the `eUiAnimationEvent` flag.

```
enum eUiAnimationEvent
    {
        eUiAnimationEvent_Started,
        eUiAnimationEvent_Stopped,
        eUiAnimationEvent_Aborted,
```

```
        eUiAnimationEvent_Updated
    };
```

UICanvasManager

Contains functions for loading, unloading, and finding UI canvases.

UiCanvasManagerBus

Services messages for the `UiCanvasManager` component.

LoadCanvas

Loads a canvas and returns the entity ID of the loaded canvas.

Syntax

```
AZ::EntityId LoadCanvas(const AZStd::string& canvasPathname)
```

UnloadCanvas

Unloads a UI canvas.

Syntax

```
void UnloadCanvas(AZ::EntityId canvasEntityId)
```

FindLoadedCanvasByPathName

Returns the entity ID of the loaded canvas that has the specified canvas pathname.

Syntax

```
AZ::EntityId FindLoadedCanvasByPathName(const AZStd::string& canvasPathname)
```

UICheckboxComponent

You can use this component to implment a UI check box in Amazon Lumberyard.

UiCheckboxBus

Services messages for the `UiCheckboxComponent`.

GetState

Returns the state of the check box. True if selected; false otherwise.

Syntax

```
bool GetState()
```

SetState

Sets the state of the check box. True if selected; false otherwise.

Syntax

```
void SetState(bool checked)
```

ToggleState

Toggles the state of the check box.

Syntax

```
bool ToggleState()
```

GetCheckedEntity

Returns the child element that is shown when the check box is selected.

Syntax

```
AZ::EntityId GetCheckedEntity()
```

SetCheckedEntity

Sets the child element to show when the check box is selected.

Syntax

```
void SetCheckedEntity(AZ::EntityId entityId)
```

GetUncheckedEntity

Returns the child element that is shown when the check box is cleared.

Syntax

```
AZ::EntityId GetUncheckedEntity()
```

SetUncheckedEntity

Sets the child element to show when the check box is cleared.

Syntax

```
void SetUncheckedEntity(AZ::EntityId entityId)
```

GetTurnOnActionName

Returns the name of the action that is triggered when the check box is selected.

Syntax

```
const AZStd::string& GetTurnOnActionName()
```

SetTurnOnActionName

Sets the action triggered when the check box is is selected.

Syntax

```
void SetTurnOnActionName(const AZStd::string & actionName)
```

GetTurnOffActionName

Returns the action triggered when the check box is is cleared.

Syntax

```
const AZStd::string & GetTurnOffActionName()
```

SetTurnOffActionName

Sets the action triggered when the check box is is cleared.

Syntax

```
void SetTurnOffActionName(const AZStd::string & actionName)
```

GetChangedActionName

Returns the action triggered when the check box value changes.

Syntax

```
AZStd::string & GetChangedActionName()
```

SetChangedActionName

Sets the action triggered when the check box value changes.

Syntax

```
void SetChangedActionName(const AZStd::string& actionName)
```

UiCheckboxNotificationBus

Services notifications for the `UiCheckboxComponent`.

OnCheckboxStateChange

Notifies that the check box state has changed.

Syntax

```
void OnCheckboxStateChange(bool checked)
```

UIDynamicLayoutComponent

Clones a prototype element to achieve the specified number of child elements.

UiDynamicLayoutBus

Services messages for the `UiDynamicLayoutComponent`.

SetNumChildElements

Specifies the number of child elements to be created dynamically.

Syntax

```
void SetNumChildElements(int numChildren)
```

UIDynamicScrollBoxComponent

Dynamically sets up scrollbox content as a horizontal or vertical list of elements that are cloned from a prototype element. For efficient scrolling, only the minimum number of elements are created.

UiDynamicScrollBoxBus

Services messages for the `UiDynamicScrollBoxComponent`.

RefreshContent

Refreshes the content. You should call this when the list size or element content has changed.

Syntax

```
void RefreshContent()
```

GetLocationIndexOfChild

Returns the location index of the specified child element. Returns -1 if not found.

Syntax

```
int GetLocationIndexOfChild(AZ::EntityId childElement)
```

GetChildElementAtLocationIndex

Returns the child element at the specified location index.

Syntax

```
AZ::EntityId GetChildElementAtLocationIndex(int index)
```

UiDynamicScrollBoxDataBus

Create this handler to provide a dynamic scrollbox with the number of elements to display.

GetNumElements

Returns the total number of elements that the dynamic scrollbox is to display.

Syntax

```
int GetNumElements()
```

UiDynamicScrollBoxElementNotificationBus

Create this handler to receive notifications of dynamic scrollbox element state changes, such as when an element is about to scroll into view.

OnElementBecomingVisible

An element is about to become visible. Use this event to populate the element with data for display. Specifies the entity ID of the element that is about to become visible and its location index.

Syntax

```
void OnElementBecomingVisible(AZ::EntityId entityId, int index)
```

UIElementComponent

Provides functionality for UI elements.

UiElementBus

Services messages for the `UiElementComponent`.

GetName

Returns the name of the current element.

Syntax

```
AZStd::string GetName()
```

GetCanvas

Returns the canvas that contains the current element. Returns `AZ::InvalidEntityId` if the element has no canvas.

Syntax

```
AZ::EntityId GetCanvas()
```

GetParent

Returns the parent of this element. Returns `AZ::InvalidEntityId` if the element has no parent.

Syntax

```
AZ::EntityId GetParent()
```

GetNumChildElements

Returns the number of child elements of the current element.

Syntax

```
int GetNumChildElements()
```

GetChild

Returns the child at the specified index. Index must be less than `GetNumChildElements()`.

Syntax

```
AZ::EntityId GetChild(int index)
```

GetIndexOfChildByEntityId

Returns the index of the specified child element.

Syntax

```
int GetIndexOfChildByEntityId(AZ::EntityId childId)
```

GetChildren

Returns the child elements of the current element.

Syntax

```
AZStd::vector<AZ::EntityId> GetChildren()
```

FindChildByName

Returns the first immediate child with the specified name. Returns `AZ::InvalidEntityId` if there is no match.

Syntax

```
AZ::EntityId FindChildByName(const AZStd::string & name)
```

FindDescendantByName

Returns the first descendant entity with the specified name. Returns `AZ::InvalidEntityId` if there is no match.

Syntax

```
AZ::EntityId FindDescendantByName(const AZStd::string & name)
```

IsEnabled

Returns true if the element is enabled; false otherwise.

Syntax

```
bool IsEnabled()
```

SetIsEnabled

Sets the enabled state of the element.

Syntax

```
void SetIsEnabled(bool isEnabled)
```

UIFaderComponent

You can use a fader component to simultaneously adjust the transparency of an element and its children.

UiFaderBus

Services messages for the `UiFaderComponent`.

GetFadeValue

Returns the fade value. The fade value is a float between zero and one. One means no fade; zero means complete fade to invisible.

Syntax

```
float GetFadeValue()
```

SetFadeValue

Sets the fade value.

Syntax

```
void SetFadeValue(float fade)
```

Fade

Triggers a fade animation.

Syntax

```
void Fade(float targetValue, float speed)
```

Parameter	Description
`targetValue`	The value at which to end the fade [0,1].
`speed`	The speed of the fade animation in full fade amount per second; 0 means instant.

IsFading

Returns whether a fade animation is taking place.

Syntax

```
bool IsFading()
```

UiFaderNotificationBus

Services notifications for the `UiFaderComponent`.

OnFadeComplete

The fade animation is done.

Syntax

```
void OnFadeComplete()
```

OnFadeInterrupted

The fade animation has been interrupted.

Syntax

```
void OnFadeInterrupted()
```

OnDestroyed

The fader component has been destroyed.

Syntax

```
void OnFaderDestroyed()
```

UIImageComponent

Controls sprites, images, and textures.

UiImageBus

Services messages for the `UiImageComponent`.

GetColor

Returns the color tint for the image.

Syntax

```
AZ::Color GetColor()
```

SetColor

Sets the color tint for the image.

Syntax

```
void SetColor(const AZ::Color& color)
```

GetSpritePathname

Returns the source location of the image to be displayed by the element.

Syntax

```
AZStd::string GetSpritePathname()
```

SetSpritePathname

Sets the source location of the image to be displayed by the element.

Syntax

```
void SetSpritePathname(AZStd::string spritePath)
```

GetRenderTargetName

Returns the name of the render target associated with the sprite.

Syntax

```
AZStd::string GetRenderTargetName()
```

SetRenderTargetName

Sets the name of the render target associated with the sprite.

Syntax

```
void SetRenderTargetName(AZStd::string renderTargetName)
```

GetSpriteType

Returns the type of the sprite.

Syntax

```
eUiSpriteType GetSpriteType()
```

Possible sprite types are as follows.

```
enum eUiSpriteType
    {
        eUiSpriteType_SpriteAsset,
        eUiSpriteType_RenderTarget
    };
```

SetSpriteType

Sets the type of the sprite.

Syntax

```
void SetSpriteType(eUiSpriteType spriteType)
```

For possible sprite types, see GetSpriteType (p. 670).

GetImageType

Returns the type of the image. Affects how the texture or sprite is mapped to the image rectangle.

Syntax

```
eUiImageType GetImageType()
```

Possible values for `eUiImageType` are as follows.

```
enum eUiImageType
    {
        eUiImageType_Stretched,
        eUiImageType_Sliced,
        eUiImageType_Fixed,
        eUiImageType_Tiled,
        eUiImageType_StretchedToFit,
        eUiImageType_StretchedToFill
    };
```

SetImageType

Sets the type of the image. Affects how the texture or sprite is mapped to the image rectangle.

Syntax

```
void SetImageType(eUiImageType imageType)
```

For possible values for `eUiImageType`, see GetImageType (p. 671).

UIInteractableComponent

Controls elements that respond to user input.

UiInteractableBus

Services messages for the `UiInteractableComponent`.

IsHandlingEvents

Returns true if event handling is enabled; false otherwise.

Syntax

```
bool IsHandlingEvents()
```

SetIsHandlingEvents

Sets whether event handling is enabled.

Syntax

```
void SetIsHandlingEvents(bool isHandlingEvents)
```

UiInteractableActionsBus

Services actions for the `UiInteractableComponent`.

GetHoverStartActionName

Returns the current hover start action name.

Syntax

```
AZStd::string& GetHoverStartActionName()
```

SetHoverStartActionName

Sets the hover start action name.

Syntax

```
SetHoverStartActionName(const AZStd::string& actionName)
```

GetHoverEndActionName

Returns the current hover end action name.

Syntax

```
AZStd::string& GetHoverEndActionName()
```

SetHoverEndActionName

Sets the hover end action name.

Syntax

```
SetHoverEndActionName(const AZStd::string& actionName)
```

GetPressedActionName

Returns the pressed action name.

Syntax

```
AZStd::string& GetPressedActionName()
```

SetPressedActionName

Sets the pressed action name.

Syntax

```
SetPressedActionName(const AZStd::string& actionName)
```

GetReleasedActionName

Returns the released action name.

Syntax

```
AZStd::string& GetReleasedActionName()
```

SetReleasedActionName

Sets the released action name.

Syntax

```
SetReleasedActionName(const AZStd::string& actionName)
```

UiInteractableStatesBus

Services states for the `UiInteractableComponent`.

GetStateColor

Returns the color to be used for the specified target when the interactable element is in the specified state.

Syntax

```
AZ::Color GetStateColor(eUiInteractableState state, AZ::EntityId target)
```

Possible values for `eUiInteractableState` are as follows.

```
enum eUiInteractableState
    {
        eUiInteractableState_Normal,
        eUiInteractableState_Hover,
        eUiInteractableState_Pressed,
        eUiInteractableState_Disabled
    };
```

SetStateColor

Sets the color to be used for the specified target when the interactable element is in the specified state. If the interactable element already has a color action for the current state and target combination, then `SetStateColor` replaces the color action.

Syntax

```
void SetStateColor(eUiInteractableState state, AZ::EntityId target, const AZ::Color& color)
```

HasStateColor

Returns true if the interactable element has a color action for the specified state and target combination..

Syntax

```
bool HasStateColor(eUiInteractableState state, AZ::EntityId target)
```

GetStateAlpha

Returns the alpha to be used for the specified target when the interactable element is in the specified state.

Syntax

```
float GetStateAlpha(eUiInteractableState state, AZ::EntityId target)
```

SetStateAlpha

Sets the alpha to be used for the specified target when the interactable element is in the specified state. If the interactable element already has an alpha action for this state and target combination, then `SetStateAlpha` replaces the alpha action.

Syntax

```
void SetStateAlpha(eUiInteractableState state, AZ::EntityId target, float alpha)
```

HasStateAlpha

Returns true if the interactable element has an alpha action for the specified state and target combination..

Syntax

```
bool HasStateAlpha(eUiInteractableState state, AZ::EntityId target)
```

GetStateSpritePathname

Returns the sprite path to be used for the specified target when the interactable element is in the specified state..

Syntax

```
AZStd::string GetStateSpritePathname(eUiInteractableState state, AZ::EntityId target)
```

SetStateSpritePathname

Sets the sprite path to be used for the specified target when the interactable element is in the specified state. If the interactable element already has a sprite action for the specified state and target combination, then `SetStateSpritePathname` replaces the sprite action.

Syntax

```
void SetStateSpritePathname(eUiInteractableState state, AZ::EntityId target, const
 AZStd::string& spritePath)
```

HasStateSprite

Returns true if the interactable element has a sprite action for the specified state and target combination..

Syntax

```
bool HasStateSprite(eUiInteractableState state, AZ::EntityId target)
```

GetStateFontPathname

Returns the font path to be used for the specified target when the interactable element is in the specified state..

Syntax

```
AZStd::string GetStateFontPathname(eUiInteractableState state, AZ::EntityId target)
```

GetStateFontEffectIndex

Returns the font effect to be used for the specified target when the interactable element is in the specified state..

Syntax

```
unsigned int GetStateFontEffectIndex(eUiInteractableState state, AZ::EntityId target)
```

SetStateFont

Sets the font to be used for the specified target when the interactable element is in the specified state. If the interactable element already has a font action for the specified state and target combination, then `SetStateFont` replaces the font action.

Syntax

```
void SetStateFont(eUiInteractableState state, AZ::EntityId target, const AZStd::string&
 fontPathname, unsigned int fontEffectIndex)
```

HasStateFont

Returns true if the interactable element has a font action for the specified state and target combination..

Syntax

```
bool HasStateFont(eUiInteractableState state, AZ::EntityId target)
```

UiInteractableNotificationBus

Services notifications for the `UIInteractableComponent`.

OnHoverStart

Called on hover start.

Syntax

```
void OnHoverStart()
```

OnHoverEnd

Called on hover end.

Syntax

```
void OnHoverEnd()
```

OnPressed

Called when an element has been pressed.

Syntax

```
void OnPressed()
```

OnReleased

Called when an element has been released.

Syntax

```
void OnReleased()
```

UiNavigationBus

Services navigation for the `UIInteractableComponent`.

GetNavigationMode

Returns the navigation mode.

Syntax

```
eUiNavigationMode GetNavigationMode()
```

Possible values for `eUiNavigationMode` are as follows.

```
enum eUiNavigationMode
    {
        eUiNavigationMode_Automatic,
        eUiNavigationMode_Custom,
        eUiNavigationMode_None
    };
```

SetNavigationMode

Sets the navigation mode.

Syntax

```
void SetNavigationMode(eUiNavigationMode navigationMode)
```

GetOnUpEntity

Returns the ID of the entity that receives focus when up is pressed.

Syntax

```
AZ::EntityId GetOnUpEntity()
```

SetOnUpEntity

Sets the entity to receive focus when up is pressed.

Syntax

```
void SetOnUpEntity(AZ::EntityId entityId)
```

GetOnDownEntity

Returns the ID of the entity that receives focus when down is pressed.

Syntax

```
AZ::EntityId GetOnDownEntity()
```

SetOnDownEntity

Sets the entity to receive focus when down is pressed.

Syntax

```
void SetOnDownEntity(AZ::EntityId entityId)
```

GetOnLeftEntity

Returns the ID of the entity to receive focus when left is pressed.

Syntax

```
AZ::EntityId GetOnLeftEntity()
```

SetOnLeftEntity

Sets the entity to receive focus when left is pressed.

Syntax

```
void SetOnLeftEntity(AZ::EntityId entityId)
```

GetOnRightEntity

Returns the ID of the entity to receive focus when right is pressed.

Syntax

```
AZ::EntityId GetOnRightEntity()
```

SetOnRightEntity

Sets the entity to receive focus when right is pressed.

Syntax

```
void SetOnRightEntity(AZ::EntityId entityId)
```

UILayoutColumnComponent

Controls column layout.

UiLayoutColumnBus

Services messages for the `UiLayoutColumnComponent`.

GetPadding

Returns, in pixels, the padding inside the edges of the element.

Syntax

```
UiPadding GetPadding()
```

Following are the possible values for `UiPadding`.

```
class UiPadding
{
     int left;
     int right;
     int top;
     int bottom;
};
```

SetPadding

Sets the padding inside the edges of the element to the pixel value specified.

Syntax

```
void SetPadding(UiPadding padding)
```

For the possible values for `UiPadding`, see GetPadding (p. 678).

GetSpacing

Returns, in pixels, the spacing between child elements.

Syntax

```
float GetSpacing()
```

SetSpacing

Sets the spacing between child elements to the pixel value specified.

Syntax

```
void SetSpacing(float spacing)
```

GetOrder

Returns the vertical order for the current layout.

Syntax

```
eUiVerticalOrder GetOrder()
```

Following are the possible values for `eUiVerticalOrder`.

```
enum eUiVerticalOrder
    {
        eUiVerticalOrder_TopToBottom,
        eUiVerticalOrder_BottomToTop
    };
```

SetOrder

Sets the vertical order for the current layout.

Syntax

```
void SetOrder(eUiVerticalOrder order)
```

For possible values for `eUiVerticalOrder`, see GetOrder (p. 679).

UILayoutGridComponent

Controls grid layout characteristics.

UiLayoutGridBus

Services messages for the `UiLayoutGridComponent`.

GetPadding

Returns, in pixels, the padding inside the edges of the element.

Syntax

```
UiPadding GetPadding()
```

Following are the attributes for `UiPadding`.

```
class UiPadding
{
    int left;
    int right;
    int top;
    int bottom;
};
```

SetPadding

Sets the padding inside the edges of the element to the specified number of pixels.

Syntax

```
void SetPadding(UiPadding padding)
```

For possible values for `UiPadding`, see GetPadding (p. 679).

GetSpacing

Returns, in pixels, the spacing between child elements.

Syntax

```
AZ::Vector2 GetSpacing()
```

SetSpacing

Sets the spacing between child elements to the specified number of pixels.

Syntax

```
void SetSpacing(AZ::Vector2 spacing)
```

GetCellSize

Returns, in pixels, the size of a child element.

Syntax

```
AZ::Vector2 GetCellSize()
```

SetCellSize

Sets the size of a child element to the specified number of pixels.

Syntax

```
void SetCellSize(AZ::Vector2 size)
```

GetHorizontalOrder

Returns the horizontal order for the layout.

Syntax

```
eUiHorizontalOrder GetHorizontalOrder()
```

Following are possible values for `eUiHorizontalOrder`.

```
enum eUiHorizontalOrder
```

```
    {
        eUiHorizontalOrder_LeftToRight,
        eUiHorizontalOrder_RightToLeft
    };
```

SetHorizontalOrder

Sets the horizontal order for the layout.

Syntax

```
void SetHorizontalOrder(eUiHorizontalOrder order)
```

For possible values for `eUiHorizontalOrder`, see GetHorizontalOrder (p. 680).

GetVerticalOrder

Returns the vertical order for the layout.

Syntax

```
eUiVerticalOrder GetVerticalOrder()
```

Following are possible values for `eUiVerticalOrder`.

```
enum eUiVerticalOrder
    {
        eUiVerticalOrder_TopToBottom,
        eUiVerticalOrder_BottomToTop
    };
```

SetVerticalOrder

Sets the vertical order for the layout.

Syntax

```
void SetVerticalOrder(eUiVerticalOrder order)
```

For possible values for `eUiVerticalOrder`, see GetVerticalOrder (p. 681).

GetStartingDirection

Returns the starting direction for the layout.

Syntax

```
eUiLayoutGridStartingDirection GetStartingDirection()
```

Following are possible values for `eUiLayoutGridStartingDirection`.

```
enum eUiLayoutGridStartingDirection
    {
        eUiLayoutGridStartingDirection_HorizontalOrder,
        eUiLayoutGridStartingDirection_VerticalOrder
```

```
    };
```

SetStartingDirection

Sets the starting direction for the layout.

Syntax

```
void SetStartingDirection(eUiLayoutGridStartingDirection direction)
```

For possible values for `eUiLayoutGridStartingDirection`, see GetStartingDirection (p. 681).

UILayoutRowComponent

Controls the grid layout of rows.

UiLayoutRowBus

Messages serviced by the `UiLayoutRowComponent`.

GetPadding

Returns, in pixels, the padding inside the edges of the element.

Syntax

```
UiPadding GetPadding()
```

Following are possible values for `UiPadding`.

```
class UiPadding
{
    int left;
    int right;
    int top;
    int bottom;
};
```

SetPadding

Sets the padding inside the edges of the element to the specified number of pixels.

Syntax

```
void SetPadding(UiPadding padding)
```

For possible values for `UiPadding`, see GetPadding (p. 682).

GetSpacing

Returns, in pixels, the spacing between child elements.

Syntax

```
float GetSpacing()
```

SetSpacing

Sets the spacing between child elements to the specified number of pixels.

Syntax

```
void SetSpacing(float spacing)
```

GetOrder

Returns the horizontal order for the current layout.

Syntax

```
eUiHorizontalOrder GetOrder()
```

Following are possible values for `eUiHorizontalOrder`.

```
enum eUiHorizontalOrder
    {
        eUiHorizontalOrder_LeftToRight,
        eUiHorizontalOrder_RightToLeft
    };
```

SetOrder

Sets the horizontal order for the current layout.

Syntax

```
void SetOrder(eUiHorizontalOrder order)
```

For possible values for `eUiHorizontalOrder`, see GetOrder (p. 683).

UIMaskComponent

You can add a mask component to an element to show only a part of the content of its child elements (for example, example, image or text).

UiMaskBus

Services messages for the `UiMaskComponent`.

GetIsMaskingEnabled

Returns whether masking is enabled.

Syntax

```
bool GetIsMaskingEnabled()
```

SetIsMaskingEnabled

Sets whether masking is enabled.

Syntax

```
void SetIsMaskingEnabled(bool enableMasking)
```

GetDrawBehind

Returns whether the mask is drawn behind the child elements.

Syntax

```
bool GetDrawBehind()
```

SetDrawBehind

Sets whether the mask is drawn behind the child elements.

Syntax

```
void SetDrawBehind(bool drawMaskVisualBehindChildren)
```

GetDrawInFront

Returns whether the mask is drawn in front of child elements.

Syntax

```
bool GetDrawInFront()
```

SetDrawInFront

Sets whether the mask is drawn in front of child elements.

Syntax

```
void SetDrawInFront(bool drawMaskVisualInFrontOfChildren)
```

GetUseAlphaTest

Returns whether to use the alpha channel in the mask visual's texture to define the mask.

Syntax

```
bool GetUseAlphaTest()
```

SetUseAlphaTest

Sets whether to use the alpha channel in the mask visual's texture to define the mask.

Syntax

```
void SetUseAlphaTest(bool useAlphaTest)
```

UIScrollBarComponent

Controls scrollbar characteristics.

UiScrollBarBus

Services messages for the `UiScrollBarComponent`.

GetHandleSize

Returns the size of the handle relative to the scrollbar (0 - 1).

Syntax

```
float GetHandleSize()
```

SetHandleSize

Sets the size of the handle relative to the scrollbar (0 - 1).

Syntax

```
void SetHandleSize(float size)
```

GetMinHandlePixelSize

Returns the minimum size of the handle in pixels.

Syntax

```
float GetMinHandlePixelSize()
```

SetMinHandlePixelSize

Sets the minimum size of the handle in pixels.

Syntax

```
void SetMinHandlePixelSize(float size)
```

GetHandleEntity

Returns the handle entity.

Syntax

```
AZ::EntityId GetHandleEntity()
```

SetHandleEntity

Sets the handle entity.

Syntax

```
void SetHandleEntity(AZ::EntityId entityId)
```

UiScrollerBus

Services scrolling for the `UiScrollBarComponent`.

GetValue

Returns the current value for the scroller (0 - 1).

Syntax

```
float GetValue()
```

SetValue

Sets the value of the scroller (0 - 1).

Syntax

```
void SetValue(float value)
```

GetOrientation

Returns the orientation of the scroller.

Syntax

```
eUiScrollerOrientation GetOrientation()
```

Following are possible values for `eUiScrollerOrientation`.

```
enum eUiScrollerOrientation
    {
        eUiScrollerOrientation_Horizontal,
        eUiScrollerOrientation_Vertical
    };
```

SetOrientation

Sets the orientation of the scroller.

Syntax

```
SetOrientation(eUiScrollerOrientation orientation)
```

For possible values for `eUiScrollerOrientation`, see GetOrientation (p. 686).

GetValueChangingActionName

Returns the name of the action triggered while the value is changing.

Syntax

```
AZStd::string& GetValueChangingActionName()
```

SetValueChangingActionName

Sets the name of the action triggered while the value is changing.

Syntax

```
void SetValueChangingActionName(const AZStd::string& actionName)
```

GetValueChangedActionName

Returns the name of the action triggered when the value has changed.

Syntax

```
const AZStd::string& GetValueChangedActionName()
```

SetValueChangedActionName

Sets the action triggered when the value has changed.

Syntax

```
void SetValueChangedActionName(const AZStd::string& actionName)
```

UiScrollerNotificationBus

Services notifications for the `UiScrollBarComponent`.

OnScrollerValueChanging

Called when the scroller value (0 - 1) is changing.

Syntax

```
void OnScrollerValueChanging(float value)
```

OnScrollerValueChanged

Called when the scroller value (0 - 1) has changed.

Syntax

```
void OnScrollerValueChanged(float value)
```

UIScrollBoxComponent

Controls the characteristics of a scroll box.

UiScrollBoxBus

Services messages for the `UiScrollBoxComponent`.

GetScrollOffset

Returns the scroll offset of the scroll box. The scroll offset is the offset from the content element's anchor point to the content element's pivot.

Syntax

```
AZ::Vector2 GetScrollOffset()
```

SetScrollOffset

Sets the scroll offset of the scroll box.

Syntax

```
void SetScrollOffset(AZ::Vector2 scrollOffset)
```

GetNormalizedScrollValue

Returns the scroll value from 0 – 1.

Syntax

```
AZ::Vector2 GetNormalizedScrollValue()
```

HasHorizontalContentToScroll

Returns whether there is content to scroll horizontally.

Syntax

```
bool HasHorizontalContentToScroll()
```

HasVerticalContentToScroll

Returns whether there is content to scroll vertically.

Syntax

```
bool HasVerticalContentToScroll()
```

GetIsHorizontalScrollingEnabled

Returns whether the scroll box allows horizontal scrolling.

Syntax

```
bool GetIsHorizontalScrollingEnabled()
```

SetIsHorizontalScrollingEnabled

Sets whether the scroll box allows horizontal scrolling.

Syntax

```
void SetIsHorizontalScrollingEnabled(bool isEnabled)
```

GetIsVerticalScrollingEnabled

Returns whether the scroll box allows vertical scrolling.

Syntax

```
bool GetIsVerticalScrollingEnabled()
```

SetIsVerticalScrollingEnabled

Sets whether the scroll box allows vertical scrolling.

Syntax

```
void SetIsVerticalScrollingEnabled(bool isEnabled)
```

GetIsScrollingConstrained

Returns whether the scroll box restricts scrolling to the content area.

Syntax

```
bool GetIsScrollingConstrained()
```

SetIsScrollingConstrained

Sets whether the scroll box restricts scrolling to the content area.

Syntax

```
void SetIsScrollingConstrained(bool isConstrained)
```

GetSnapMode

Returns the snap mode for the scroll box.

Syntax

```
eUiScrollBoxSnapMode GetSnapMode()
```

Following are possible values for `eUiScrollBoxSnapMode`.

```
enum eUiScrollBoxSnapMode
    {
        eUiScrollBoxSnapMode_None,
        eUiScrollBoxSnapMode_Children,
        eUiScrollBoxSnapMode_Grid
    };
```

SetSnapMode

Sets the snap mode for the scroll box.

Syntax

```
void SetSnapMode(eUiScrollBoxSnapMode snapMode)
```

For possible values for `eUiScrollBoxSnapMode`, see GetSnapMode (p. 689).

GetSnapGrid

Returns the snapping grid of the scroll box.

Syntax

```
AZ::Vector2 GetSnapGrid()
```

SetSnapGrid

Sets the snapping grid of the scroll box.

Syntax

```
void SetSnapGrid(AZ::Vector2 snapGrid)
```

GetHorizontalScrollBarVisibility

Returns the visibility behavior for the horizontal scroll bar of the scroll box.

Syntax

```
eUiScrollBoxScrollBarVisibility GetHorizontalScrollBarVisibility()
```

Following are possible values for `eUiScrollBoxScrollBarVisibility`.

```
enum eUiScrollBoxScrollBarVisibility
    {
        eUiScrollBoxScrollBarVisibility_AlwaysShow,
        eUiScrollBoxScrollBarVisibility_AutoHide,
        eUiScrollBoxScrollBarVisibility_AutoHideAndResizeViewport
    };
```

SetHorizontalScrollBarVisibility

Sets the visibility behavior for the horizontal scroll bar of the scroll box.

Syntax

```
void SetHorizontalScrollBarVisibility(eUiScrollBoxScrollBarVisibility visibility)
```

For possible values for `eUiScrollBoxScrollBarVisibility`, see GetHorizontalScrollBarVisibility (p. 690).

GetVerticalScrollBarVisibility

Returns the visibility behavior for the vertical scroll bar of the scroll box.

Syntax

```
eUiScrollBoxScrollBarVisibility GetVerticalScrollBarVisibility()
```

SetVerticalScrollBarVisibility

Sets the visibility behavior for the vertical scroll bar of the scroll box.

Syntax

```
void SetVerticalScrollBarVisibility(eUiScrollBoxScrollBarVisibility visibility
```

GetScrollOffsetChangingActionName

Returns the action triggered while the scroll box is being dragged.

Syntax

```
AZStd::string& GetScrollOffsetChangingActionName()
```

SetScrollOffsetChangingActionName

Sets the action triggered while the scroll box is being dragged.

Syntax

```
void SetScrollOffsetChangingActionName(const AZStd::string& actionName)
```

GetScrollOffsetChangedActionName

Returns the action triggered when the scroll box drag is completed.

Syntax

```
const AZStd::string& GetScrollOffsetChangedActionName()
```

SetScrollOffsetChangedActionName

Sets the action triggered when the scroll box drag is completed.

Syntax

```
void SetScrollOffsetChangedActionName(const AZStd::string& actionName)
```

GetContentEntity

Returns the content element for the scroll box.

Syntax

```
AZ::EntityId GetContentEntity()
```

SetContentEntity

Sets the content element for the scroll box.

Syntax

```
void SetContentEntity(AZ::EntityId entityId)
```

GetHorizontalScrollBarEntity

Returns the horizontal scroll bar element for the scroll box.

Syntax

```
AZ::EntityId GetHorizontalScrollBarEntity()
```

SetHorizontalScrollBarEntity

Sets the horizontal scroll bar element for the scroll box.

Syntax

```
void SetHorizontalScrollBarEntity(AZ::EntityId entityId)
```

GetVerticalScrollBarEntity

Returns the vertical scroll bar element for the scroll box.

Syntax

```
AZ::EntityId GetVerticalScrollBarEntity()
```

SetVerticalScrollBarEntity

Sets the vertical scroll bar element for the scroll box.

Syntax

```
void SetVerticalScrollBarEntity(AZ::EntityId entityId)
```

FindClosestContentChildElement

Finds the child of the content element that is closest to the content anchors at the current scroll offset (the currently selected child).

Syntax

```
AZ::EntityId FindClosestContentChildElement()
```

UiScrollBoxNotificationBus

Services scroll offset change notifications for the `UiScrollBoxComponent`.

OnScrollOffsetChanging

Called when the scroll offset is changing.

Syntax

```
void OnScrollOffsetChanging(AZ::Vector2 newScrollOffset)
```

OnScrollOffsetChanged

Called when the scroll offset has changed.

Syntax

```
void OnScrollOffsetChanged(AZ::Vector2 newScrollOffset)
```

UiScrollableNotificationBus

Services scrollable value change notifications for the `UiScrollBoxComponent`.

OnScrollableValueChanging

Called when the scroll value (0 - 1) is changing.

Syntax

```
void OnScrollableValueChanging(AZ::Vector2 value)
```

OnScrollableValueChanged

Called when the scroll value (0 - 1) has changed.

Syntax

```
void OnScrollableValueChanged(AZ::Vector2 value)
```

UISliderComponent

Controls the values of a slider.

UiSliderBus

Services messages for the `UiSliderComponent`.

GetValue

Returns the value of the slider.

Syntax

```
float GetValue()
```

SetValue

Sets the value of the slider.

Syntax

```
void SetValue(float value)
```

GetMinValue

Returns the minimum value of the slider.

Syntax

```
float GetMinValue()
```

SetMinValue

Sets the minimum value of the slider.

Syntax

```
void SetMinValue(float value)
```

GetMaxValue

Returns the maximum value of the slider.

Syntax

```
float GetMaxValue()
```

SetMaxValue

Sets the maximum value of the slider.

Syntax

```
void SetMaxValue(float value)
```

GetStepValue

Returns the smallest increment allowed between values. Zero means no restriction.

Syntax

```
float GetStepValue()
```

SetStepValue

Sets the smallest increment allowed between values. Use zero for no restriction.

Syntax

```
void SetStepValue(float step)
```

GetTrackEntity

Returns the track element.

Syntax

```
AZ::EntityId GetTrackEntity()
```

SetTrackEntity

Sets the track element.

Syntax

```
void SetTrackEntity(AZ::EntityId entityId)
```

GetFillEntity

Returns the fill element.

Syntax

```
AZ::EntityId GetFillEntity()
```

SetFillEntity

Sets the fill element.

Syntax

```
void SetFillEntity(AZ::EntityId entityId)
```

GetManipulatorEntity

Returns the entity ID of the manipulator element.

Syntax

```
AZ::EntityId GetManipulatorEntity()
```

SetManipulatorEntity

Sets the manipulator element.

Syntax

```
void SetManipulatorEntity(AZ::EntityId entityId)
```

GetValueChangingActionName

Returns the name of the action triggered while the value is changing.

Syntax

```
const AZStd::string& GetValueChangingActionName()
```

SetValueChangingActionName

Sets the action triggered while the value is changing.

Syntax

```
void SetValueChangingActionName(const AZStd::string& actionName)
```

GetValueChangedActionName

Returns the action triggered when the value has finished changing.

Syntax

```
const AZStd::string& GetValueChangedActionName()
```

SetValueChangedActionName

Sets the action triggered when the value is done changing.

Syntax

```
void SetValueChangedActionName(const AZStd::string& actionName)
```

UiSliderNotificationBus

Services notifications for the `UiSliderComponent`.

OnSliderValueChanging

The slider value is changing.

Syntax

```
void OnSliderValueChanging(float value)
```

OnSliderValueChanged

The slider value has finished changing.

Syntax

```
void OnSliderValueChanged(float value)
```

UITextComponent

Controls the text and formatting of a text element.

UiTextBus

Services messages for the `UiTextComponent`.

GetText

Returns the text string being displayed by the element.

Syntax

```
AZStd::string GetText()
```

SetText

Sets the text string being displayed by the element.

Syntax

```
void SetText(const AZStd::string& text)
```

GetColor

Returns the color to draw the text string.

Syntax

```
AZ::Color GetColor()
```

SetColor

Sets the color to draw the text string.

Syntax

```
void SetColor(const AZ::Color& color)
```

GetFont

Returns the pathname to the font.

Syntax

```
AZStd::string GetFont()
```

SetFont

Sets the pathname to the font.

Syntax

```
void SetFont(const AZStd::string& fontPath)
```

GetFontEffect

Returns the font effect.

Syntax

```
int GetFontEffect()
```

SetFontEffect

Sets the font effect.

Syntax

```
void SetFontEffect(int effectIndex)
```

GetFontSize

Returns the size of the font in points.

Syntax

```
float GetFontSize()
```

SetFontSize

Sets the size of the font in points.

Syntax

```
void SetFontSize(float size)
```

GetHorizontalTextAlignment

Returns the horizontal text alignment.

Syntax

```
eUiHAlign GetHorizontalTextAlignment()
```

Following are possible values for `eUiHAlign`.

```
enum eUiHAlign
    {
        eUiHAlign_Left,
        eUiHAlign_Center,
        eUiHAlign_Right
    };
```

SetHorizontalTextAlignment

Sets the horizontal text alignment.

Syntax

```
void SetHorizontalTextAlignment(eUiHAlign alignment)
```

For possible values for `eUiHAlign`, see GetHorizontalTextAlignment (p. 698).

GetVerticalTextAlignment

Returns the vertical text alignment.

Syntax

```
eUiVAlign GetVerticalTextAlignment()
```

Following are possible values for eUiVAlign.

```
enum eUiVAlign
    {
        eUiVAlign_Top,
        eUiVAlign_Center,
        eUiVAlign_Bottom
    };
```

SetVerticalTextAlignment

Sets the vertical text alignment.

Syntax

```
void SetVerticalTextAlignment(eUiVAlign alignment)
```

For possible values for `eUiVAlign`, see GetVerticalTextAlignment (p. 698).

GetOverflowMode

Returns the overflow behavior of the text.

Syntax

```
eUiTextOverflowMode GetOverflowMode()
```

Following are possible values for `eUiTextOverflowMode`.

```
enum eUiTextOverflowMode
    {
        eUiTextOverflowMode_OverflowText,
        eUiTextOverflowMode_ClipText
    };
```

SetOverflowMode

Sets the overflow behavior of the text.

Syntax

```
void SetOverflowMode(eUiTextOverflowMode overflowMode)
```

For possible values for `eUiTextOverflowMode`, see GetOverflowMode (p. 699).

GetWrapText

Returns whether text is wrapped.

Syntax

```
eUiTextWrapTextSetting GetWrapText()
```

Following are possible values for `eUiTextWrapTextSetting`.

```
enum eUiTextWrapTextSetting
    {
        eUiTextWrapTextSetting_NoWrap,
        eUiTextWrapTextSetting_Wrap
    };
```

SetWrapText

Sets whether text is wrapped.

Syntax

```
void SetWrapText(eUiTextWrapTextSetting wrapSetting)
```

For possible values for `eUiTextWrapTextSetting`, see GetWrapText (p. 699).

UITextInputComponent

You can use a text input component to provide player text input capability.

UiTextInputBus

Services messages for the `UiTextInputComponent`.

GetTextSelectionColor

Returns the color to be used for the text background when it is selected.

Syntax

```
AZ::Color GetTextSelectionColor()
```

SetTextSelectionColor

Sets the color to be used for the text background when it is selected.

Syntax

```
void SetTextSelectionColor(const AZ::Color& color)
```

GetTextCursorColor

Returns the color to be used for the text cursor.

Syntax

```
AZ::Color GetTextCursorColor()
```

SetTextCursorColor

Sets the color to be used for the text cursor.

Syntax

```
void SetTextCursorColor(const AZ::Color& color)
```

GetCursorBlinkInterval

Returns the cursor blink interval of the text input.

Syntax

```
float GetCursorBlinkInterval()
```

SetCursorBlinkInterval

Sets the cursor blink interval of the text input.

Syntax

```
void SetCursorBlinkInterval(float interval)
```

GetMaxStringLength

Returns the maximum number of characters that can be entered.

Syntax

```
int GetMaxStringLength()
```

SetMaxStringLength

Sets the maximum number of characters that can be entered.

Syntax

```
void SetMaxStringLength(int maxCharacters)
```

GetChangeAction

Returns the action triggered when the text is changed.

Syntax

```
const AZStd::string& GetChangeAction()
```

SetChangeAction

Sets the action triggered when the text is changed.

Syntax

```
void SetChangeAction(const AZStd::string& actionName)
```

GetEndEditAction

Returns the action triggered when the editing of text is finished.

Syntax

```
const AZStd::string& GetEndEditAction()
```

SetEndEditAction

Sets the action triggered when the editing of text is finished.

Syntax

```
void SetEndEditAction(const AZStd::string& actionName)
```

GetEnterAction

Returns the action triggered when enter is pressed.

Syntax

```
const AZStd::string& GetEnterAction()
```

SetEnterAction

Sets the action triggered when enter is pressed.

Syntax

```
void SetEnterAction(const AZStd::string& actionName)
```

GetTextEntity

Returns the text element.

Syntax

```
AZ::EntityId GetTextEntity()
```

SetTextEntity

Sets the text element.

Syntax

```
void SetTextEntity(AZ::EntityId textEntity)
```

GetText

Returns the text string being displayed or edited by the element.

Syntax

```
AZStd::string GetText()
```

SetText

Sets the text string being displayed or edited by the element.

Syntax

```
void SetText(const AZStd::string& text)
```

GetPlaceHolderTextEntity

Returns the placeholder text element.

Syntax

```
AZ::EntityId GetPlaceHolderTextEntity()
```

SetPlaceHolderTextEntity

Sets the placeholder text element.

Syntax

```
void SetPlaceHolderTextEntity(AZ::EntityId textEntity)
```

GetIsPasswordField

Returns whether the text input is configured as a password field.

Syntax

```
bool GetIsPasswordField()
```

SetIsPasswordField

Sets whether the text input is configured as a password field.

Syntax

```
void SetIsPasswordField(bool passwordField)
```

GetReplacementCharacter

Returns the replacement character used to hide password text.

Syntax

```
char GetReplacementCharacter()
```

SetReplacementCharacter

Sets the replacement character used to hide password text.

Syntax

```
void SetReplacementCharacter(char replacementChar)
```

UiTextInputNotificationBus

Services notifications for the `UiTextInputComponent`.

OnTextInputChange

Called when a character is added, removed, or changed.

Syntax

```
void OnTextInputChange(const AZStd::string& textString)
```

OnTextInputEndEdit

Called when edit of text is completed.

Syntax

```
void OnTextInputEndEdit(const AZStd::string& textString)
```

OnTextInputEnter

Called when **Enter** is pressed on the keyboard.

Syntax

```
void OnTextInputEnter(const AZStd::string& textString)
```

UITooltipComponent

You can use a tooltip component to provide the text of a tooltip.

UiTooltipBus

Services messages for the `UiTooltipComponent`.

GetText

Returns the tooltip text.

Syntax

```
AZStd::string GetText()
```

SetText

Sets the tooltip text.

Syntax

```
void SetText(const AZStd::string& text)
```

UITooltipDisplayComponent

Controls the display behavior of a tooltip.

UiTooltipDisplayBus

Services messages for the `UiTooltipDisplayComponent`.

GetAutoPosition

Returns whether the tooltip display element is auto positioned.

Syntax

```
bool GetAutoPosition()
```

SetAutoPosition

Sets whether the tooltip display element is auto positioned.

Syntax

```
void SetAutoPosition(bool autoPosition)
```

GetAutoPositionMode

Returns the auto position mode.

Syntax

```
eUiTooltipDisplayAutoPositionMode GetAutoPositionMode()
```

Following are possible values for `eUiTooltipDisplayAutoPositionMode`.

```
enum eUiTooltipDisplayAutoPositionMode
    {
        eUiTooltipDisplayAutoPositionMode_OffsetFromMouse,
        eUiTooltipDisplayAutoPositionMode_OffsetFromElement
    };
```

SetAutoPositionMode

Sets the auto position mode.

Syntax

```
void SetAutoPositionMode(eUiTooltipDisplayAutoPositionMode autoPositionMode)
```

For possible values for `eUiTooltipDisplayAutoPositionMode`, see GetAutoPositionMode (p. 705).

GetOffset

Returns the offset from the tooltip display element's pivot to the mouse position.

Syntax

```
const AZ::Vector2& GetOffset()
```

SetOffset

Sets the offset from the tooltip display element's pivot to the mouse position.

Syntax

```
void SetOffset(const AZ::Vector2& offset)
```

GetAutoSize

Returns whether the tooltip display element should be resized so that the text element size matches the size of the string.

Syntax

```
bool GetAutoSize()
```

SetAutoSize

Sets whether the tooltip display element should be resized so that the text element size matches the size of the string.

Syntax

```
void SetAutoSize(bool autoSize)
```

GetTextEntity

Returns the entity ID of the text element that is used for resizing.

Syntax

```
AZ::EntityId GetTextEntity()
```

SetTextEntity

Sets the entity ID of the text element that is used for resizing. The text element must be a child of the current entity.

Syntax

```
void SetTextEntity(AZ::EntityId textEntity)
```

GetDelayTime

Returns the amount of time to wait before showing the tooltip display element after hover start.

Syntax

```
float GetDelayTime()
```

SetDelayTime

Sets the amount of time to wait before showing the tooltip display element after hover start.

Syntax

```
void SetDelayTime(float delayTime)
```

GetDisplayTime

Returns the amount of time the tooltip display element is to remain visible.

Syntax

```
float GetDisplayTime()
```

SetDisplayTime

Sets the amount of time the tooltip display element is to remain visible.

Syntax

```
void SetDisplayTime(float displayTime)
```

UITransform2dComponent

Controls positioning, scaling, rotation, anchor, and offset settings for UI elements.

UiTransformBus

Services messages for the `UiTransform2dComponent`.

GetZRotation

Returns the rotation about the Z axis.

Syntax

```
float GetZRotation()
```

SetZRotation

Sets the rotation about the Z axis.

Syntax

```
void SetZRotation(float rotation)
```

GetScale

Returns the scale.

Syntax

```
AZ::Vector2 GetScale()
```

SetScale

Sets the scale.

Syntax

```
void SetScale(AZ::Vector2 scale)
```

GetPivot

Returns the pivot point.

Syntax

```
AZ::Vector2 GetPivot()
```

SetPivot

Sets the pivot point.

Syntax

```
void SetPivot(AZ::Vector2 pivot)
```

GetScaleToDevice

Returns whether the element and all its children are scaled to allow for the difference between the authored canvas size and the actual viewport size.

Syntax

```
bool GetScaleToDevice()
```

SetScaleToDevice

Sets whether the element and all its children are scaled to allow for the difference between the authored canvas size and the actual viewport size.

Syntax

```
void SetScaleToDevice(bool scaleToDevice)
```

GetCanvasPosition

Returns the position for the current element in canvas space.

Syntax

```
AZ::Vector2 GetCanvasPosition()
```

SetCanvasPosition

Sets the position for the current element in canvas space.

Syntax

```
void SetCanvasPosition(const AZ::Vector2& position)
```

GetLocalPosition

Returns the position for this element relative to the center of the element's anchors.

Syntax

```
AZ::Vector2 GetLocalPosition()
```

SetLocalPosition

Sets the position for this element relative to the center of the element's anchors.

Syntax

```
void SetLocalPosition(const AZ::Vector2& position)
```

MoveCanvasPositionBy

Move the current element in canvas space.

Syntax

```
void MoveCanvasPositionBy(const AZ::Vector2& offset)
```

MoveLocalPositionBy

Move the current element relative to the center of the element's anchors.

Syntax

```
void MoveLocalPositionBy(const AZ::Vector2& offset)
```

UiTransform2dBus

Services anchors and offsets for the `UITransform2dComponent`.

GetAnchors

Returns the UI anchors.

Syntax

```
UiAnchors GetAnchors()
```

Following are possible values for `UiAnchors`.

```
class UiAnchors
{
    float left;
    float top;
    float right;
    float bottom;
};
```

SetAnchors

Sets the anchors.

Syntax

```
void SetAnchors(UiAnchors anchors, bool adjustOffsets, bool allowPush)
```

For possible values for `UiAnchors`, see GetAnchors (p. 709).

Parameter	Description
`adjustOffsets`	If true, the offsets are adjusted to keep the rectangle in the same position.
`allowPush`	Only has effect if the anchors are invalid. If true, when an anchor is changed to overlap the anchor opposite it, the oppose anchor moves.

GetOffsets

Returns the UI offsets.

Syntax

```
UiOffsets GetOffsets()
```

Following are possible values for `UiOffsets`.

```
class UiOffsets
{
    float left;
    float top;
    float right;
    float bottom;
};
```

SetOffsets

Sets UI offsets.

Syntax

```
void SetOffsets(UiOffsets offsets)
```

For possible values for `UiOffsets`, see GetOffsets (p. 710).

SetPivotAndAdjustOffsets

Sets the pivot and adjusts the offsets to stay in same place.

Syntax

```
void SetPivotAndAdjustOffsets(AZ::Vector2 pivot)
```

UI World Components

Controls the loading and unloading of UI canvases.

UiCanvasAssetRefBus

Services messages for the `UiCanvasAssetRefComponent`.

LoadCanvas

Loads the UI canvas using the stored asset reference. Returns the entity ID of the loaded canvas.

Syntax

```
AZ::EntityId LoadCanvas()
```

UnloadCanvas

Unloads the UI canvas using the stored asset reference.

Syntax

```
void UnloadCanvas()
```

UiCanvasAssetRefNotificationBus

Handles events sent by the `UiCanvasAssetRefComponent`.

OnCanvasLoadedIntoEntity

Called when the canvas asset reference loads a UI canvas.

Syntax

```
void OnCanvasLoadedIntoEntity(AZ::EntityId uiCanvasEntity)
```

UiCanvasProxyRefBus

Services messages for the `UiCanvasProxyRefComponent`.

SetCanvasRefEntity

Sets the entity that manages the UI canvas for the current proxy.

Syntax

```
void SetCanvasRefEntity(AZ::EntityId canvasAssetRefEntity)
```

UiCanvasRefBus

Services messages for the `UiCanvasAssetRefComponent` and `UiCanvasProxyRefComponent`.

GetCanvas

Returns the UI canvas associated with the current entity

Syntax

```
AZ::EntityId GetCanvas()
```

UiCanvasRefNotificationBus

Provides notifications of when the UI canvas reference changes.

OnCanvasRefChanged

Called when the canvas referenced by a `UiCanvasAssetRefComponent` has changed. This can happen when LoadCanvas (p. 711), UnloadCanvas (p. 711), or SetCanvasRefEntity (p. 711) is called.

Syntax

```
void OnCanvasRefChanged(AZ::EntityId uiCanvasRefEntity, AZ::EntityId uiCanvasEntity
```

Legacy Lua Scripting Reference

CryLua is deprecated and will be removed in a future version of Lumberyard.

The topics in this section document Lua functionality for the legacy `script context`. Starting with Lumberyard 1.8, Lua scripts use the new behavior context (p. 453) that replaces the legacy script context. Scripts that were written before the integration of the behavior context no longer work in Lumberyard versions 1.8 and later. For information on updating Lua code from legacy script context to the new behavior context, see the migration notes for Lumberyard 1.8.

Topics

- Working with Lua Scripting (p. 713)
- Loading Canvases in Lua (p. 719)
- Entity System Script Callbacks (p. 720)
- Game Rules Script Callbacks (p. 722)
- Common Lua Globals and Functions (p. 724)
- EntityUtils Lua Functions (p. 729)
- Lua Vector and Math Functions (p. 732)
- Physics Lua Functions (p. 742)
- Lua ScriptBind Reference (p. 743)
- Integrating Lua and C++ (p. 949)

Working with Lua Scripting

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lumberyard uses Lua for its scripting language.

The Entity system can attach a script proxy to any entity, which is in the form of a table that can include data and functions. AI behaviors are often written in scripts. Additionally, several game systems, including Actor, Item, Vehicle, and GameRules, rely on scripting to extend their functionality.

The advantages of using scripts include:

- Fast iteration – Scripts can be reloaded within the engine.
- Runtime performance – Careful usage of available resources can result into scripts that run nearly as fast as compiled code.
- Easy troubleshooting – An embedded Lua debugger can be invoked at any time.

Most of the systems in Lumberyard expose ScriptBind functions, which allow Lua scripts to call existing code written in C++. See the Lua ScriptBind Reference (p. 743)for more details.

Running Scripts

You can run scripts either by calling script files directly from code or by using console commands.

In code

Scripts are stored in the `\Game\Scripts` directory. To invoke a script file, call the `LoadScript` function from your C++ code. For more information, see Integrating Lua and C++ (p. 949). Another option is to create a script entity, as described in Entity Scripting (p. 533).

In the Console

Script instructions can be executed using the in-game console. This can be done by appending the **#** character before the instructions. This functionality is limited to Lumberyard Editor or when running the launcher in dev mode (using the `-DEVMODE` command-line argument).

Reloading Scripts During Runtime

In Lumberyard Editor it is always possible to reload entities within the user interface. When reloading a script entity, choose the **Reload Script** button, which is found in the Rollup Bar.

You can also use the following ScriptBind functions to reload scripts.

- `Script.ReloadScript(filename)`
- `Script.ReloadScripts()`

To invoke these functions from the console, use the following syntax:

```
#Script.ReloadScript("Scripts\\EntityCommon.lua")
```

Recommended Reading

The following resources on the Lua language are recommended reading when working with scripts with Lumberyard.

- Lua 5.1 Reference Manual
- Programming in Lua, Third Edition
- Other books

Topics

- Using the Lua Remote Debugger (p. 715)
- Using the Lua XML Loader (p. 716)

Using the Lua Remote Debugger

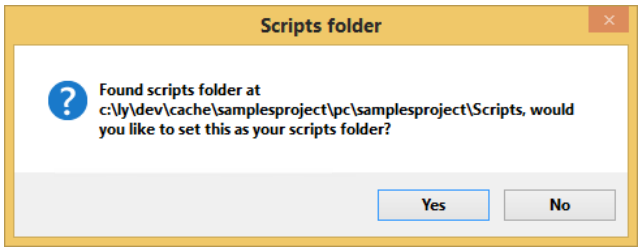
Lumberyard includes a standalone visual script debugger for Lua. To start the debugger, you first enable it in the console, and then run the `LuaRemoteDebugger.exe` executable file.

1. In the Lumberyard Editor console or game console, type **`lua_debugger 1`** or **`lua_debugger 2`**. This enables enable debugging in one of the following two modes:

   - Mode 1 – The debugger breaks on both breakpoints and script errors.
   - Mode 2 – The debugger breaks only on script errors.
2. Run the Lua remote debugger executable file at the Lumberyard directory location `\dev\Tools\LuaRemoteDebugger\LuaRemoteDebugger.exe`.
3. In the Lua remote debugger, on the **File** menu, choose **Connect**.
4. If you are running the game in the editor (you pressed **`Ctrl-G`**) and want to debug your scripts, choose **PC (Editor)**. If you want to attach the debugger to the built game executable, choose **PC (Game)**.

   For **IP address** and **Port**, type the IP address and port of the computer to which you want to connect. The default options connect to the game on your local computer. The default IP address is 127.0.01 (localhost). For **PC (Editor)**, the default port is 9433. For **PC (Game)**, the default port is 9432.
5. Choose **Connect**. In Lumberyard Editor, the console window displays **Lua remote debug client connected**.

   The first time you run Lua remote debugger, it prompts you for the scripts folder:

   The default folder is the `Scripts` folder of the project that you are running. For example, if you are running the samples project, the folder is `samplesproject/Scripts`.
6. To accept the default location, click **Yes**.

   Note
   To change the scripts folder location, choose **File**, **Set Scripts Folder**.

   After you choose the location for your scripts folder, the folder's contents are shown in the navigation tree on the left.

Performing Tasks in the Lua Remote Debugger

To perform specific tasks in the Lua remote debugger, see the following table:

To do this	Do this
Open a script file	Double click the script file in the navigation pane, or press **`Ctrl+O`** to open the **Find File** dialog.
Set a break point	Place the cursor on the line in the script where you want the break to occur, and then click the red dot in the toolbar or press **`F9`**. When

To do this	Do this
	program execution stops on a break point, the **Call Stack** and **Locals** tabs populate.
Remove a break point	Place the cursor on the line with the breakpoint that you want to remove, and then click the red dot in the toolbar or press `F9`.
Use the Breakpoints tab	The **Breakpoints** tab window displays each of your breakpoints with a check box next to it. To enable or disable a breakpoint, select or clear its check box. In the script window, the breakpoint's status is indicated by its color: red is active; gray is disabled.
To watch (inspect) variable values	When execution is paused on a breakpoint, click the **Watch** tab, click the first column of a blank row, and then type the name of the variable that you want to watch.
Pause execution	Click the pause (break) button on the toolbar or press `Ctrl+Alt+Pause`.
Resume execution	Click the play button on the toolbar or press `F5`.
Step over a procedure	Click the toolbar icon or press `F10`.
Step into a procedure	Click the toolbar icon or press `F11`.
Step out of a procedure	Click the toolbar icon or press `Shift+F11`.
Close a script file	Choose **File**, **Close**, or press `Ctrl+W`
Disconnect from the editor or game	In the Lua debugger, choose **File**, **Disconnect**. The Lumberyard console displays a **network connection terminated** message.

Note
Code changes that you make in the debugger window do not change the loaded script and are discarded after the debugger window is closed.

Using the Lua XML Loader

CryLua is deprecated and will be removed in a future version of Lumberyard.

There is a generic interface for parsing and translating XML files into Lua files. This interface uses an XML file as a definition format that declares what kind of XML is included in a file and what kind of Lua to create from the XML. The format includes some simple validation methods to ensure that the data received is what is expected.

XML Data

The XML loader can distinguish between three kinds of data: properties, arrays, and tables.

Tables

This table represents a Lua-based table:

```
letters = { a="a", b="b", c="c" }
```

In an XML data file, this table would look like this:

```
<letters a="a" b="b" c="c"/>
```

The XML definition file would look like this:

```
<Table name="letters">
  <Property name="a" type="string"/>
  <Property name="b" type="string"/>
  <Property name="c" type="string"/>
</Table>
```

Each element can be marked as optional in the definition file using the attribute `optional="1"`.

Arrays

There are two possible types of arrays. The first type is a simple group of elements, shown in Lua like this:

```
numbers = {0,1,2,3,4,5,6,7,8,9}
```

In the XML data file, the array would look like this:

```
<numbers>
  <number value="0"/>
  <number value="1"/>
  <number value="2"/>
  <number value="3"/>
  <number value="4"/>
  <number value="5"/>
  <number value="6"/>
  <number value="7"/>
  <number value="8"/>
  <number value="9"/>
</numbers>
```

The data definition file would look like this:

```
<Array name="numbers" type="int" elementName="number"/>
```

The second array type is an array of tables. In Lua:

```
wheels = {
  {size=3, weight=10},
  {size=2, weight=1},
  {size=4, weight=20},
}
```

In the XML data file:

```
<wheels>
```

```
  <wheel size="3" weight="10"/>
  <wheel size="2" weight="1"/>
  <wheel size="4" weight="20"/>
</wheels>
```

The XML definition file:

```
<Array name="wheels" elementName="wheel"> <!-- note no type is attached -->
  <Property name="size" type="float"/>
  <Property name="weight" type="int"/>
</Array>
```

Loading and Saving a Table from Lua

To load and initialize a Lua table:

```
someTable = CryAction.LoadXML( definitionFileName, dataFileName )
```

When storing XML files for scripts, the recommended practice is to keep the definition files with the scripts that use them, but store the data files in a directory outside the Scripts directory.

To save a table from Lua:

```
CryAction.SaveXML( definitionFileName, dataFileName, table )
```

Data Types

The following data types are available, and can be set wherever a "type" attribute is present in the definition file.

- float – Floating point number.
- int – Integer.
- string – String.
- bool – Boolean value.
- Vec3 – Floating point vectors with three components. Values of this type are expressed as follows:
  - XML – **`"1,2,3"`**
  - Lua – **`{x=1,y=2,z=3}`**

Enums

For string type properties, an optional `<Enum>` definition can be used. Property values will be validated against the enum.

Example:

```
<Property name="view" type="string">
 <Enum>
  <Value>GhostView</Value>
  <Value>ThirdPerson</Value>
  <Value>BlackScreen</Value>
 </Enum>
</Property>
```

Enum support for other data types can be added, if necessary.

Example

XML definition file:

```
<Definition root="Data">

 <Property name="version" type="string"/>

 <Table name="test">
  <Property name="a" type="string"/>
  <Property name="b" type="int" optional="1"/>
  <Array name="c" type="string" elementName="Val"/>
  <Array name="d" elementName="Value">
     <Property name="da" type="float"/>
     <Property name="db" type="Vec3"/>
  </Array>
  <Property name="e" type="int"/>
 </Table>

</Definition>
```

Corresponding XML data file:

```
<Data version="Blag 1.0">
 <test
  a="blag"
  e="3">
  <c>
     <Val value="blag"/>
     <Val value="foo"/>
  </c>
  <d>
     <Value da="3.0" db="2.1,2.2,2.3"/>
     <Value da="3.1" db="2.1,2.2,2.3"/>
     <Value da="3.2" db="3.1,3.2,2.3"/>
  </d>
 </test>
</Data>
```

Loading Canvases in Lua

CryLua is deprecated and will be removed in a future version of Lumberyard.

The Lumberyard UI Editor uses the concept of a UI canvas as an invisible backdrop for user interface elements. You can use the Lua scripting language to load and unload UI canvases in Lumberyard.

To load a canvas in Lua

1. Create a new, plain text file in your game project directory with a `.lua` file extension.
2. Type or paste the following sample script into your new `Lua` file:

   Note
   The following script uses a Lua file named `loadcanvas.lua` and loads a canvas file named `menu.uicanvas` saved at the root of the game project directory. Substitute the appropriate file names for your script.

```
loadcanvas =
{
   Properties =
   {
   },
}

function loadcanvas:OnActivate()
    self.uiCanvasLuaProxy = UiCanvasLuaProxy()
    self.uiCanvasLuaProxy:LoadCanvas("menu.uicanvas")
end
```

3. In Lumberyard Editor, right-click in the **Viewport** and click **Create Component Entity**.
4. If the **Entity Inspector** does not open automatically, click **Tools**, **Entity Inspector**.
5. Click **Add Component**.
6. Select **Scripting**, **Lua Script**.
7. Under **Lua Script**, click **...** and open the Lua script file that you created.
8. In Lumberyard Editor, click **Game**, **Switch to Game** to enter game mode. Verify that your canvas file loads.

See Also

For more information, consult the following resources.

- For a full listing of UI Lua API calls, see UI Lua Reference (p. 654).
- For general information about UI canvases, see Working with UI Canvases in the Amazon Lumberyard User Guide.

Entity System Script Callbacks

CryLua is deprecated and will be removed in a future version of Lumberyard.

This topic describes all callbacks for the Entity system. Use of these callbacks functions is not obligatory, but some cases require that entities behave properly within the Lumberyard Editor. For example, the `OnReset` callback should be used to clean the state when a user enters or leaves the game mode within the Lumberyard Editor.

Default State Functions

Callback Function	Description
OnSpawn	Called after an entity is created by the Entity system.
OnDestroy	Called when an entity is destroyed (like `OnShutDown()` gets called).
OnInit	Called when an entity gets initialized via ENTITY_EVENT_INIT, and when its ScriptProxy gets initialized.
OnShutDown	Called when an entity is destroyed (like `OnDestroy()` gets called).
OnReset	Usually called when an editor wants to reset the state.

Callback Function	Description
OnPropertyChange	Called by Lumberyard Editor when the user changes one of the properties.

Script State Functions

Callback Function	Description
OnBeginState	Called during `Entity.GotoState()` after the state has been changed (that is, after `OnEndState()` is called on the old state).
OnBind	Called when a child entity is attached to an entity. Parameters include: • script table for the child entity
OnCollision	Called when a collision between an entity and something else occurs. Parameters include: • script table with information about the collision
OnEndState	Called during `Entity.GotoState()` while the old state is still active and before `OnBeginState()` is called on the new state.
OnEnterArea	Called when an entity has fully entered an area or trigger. Parameters include: • **areaId** (int) • **fade fraction** (float) This value is 1.0f if the entity has fully entered the area, or 0.0f in the case of trigger boxes.
OnEnterNearArea	Called when an entity enters the range of an area. Works with Box-, Sphere- and Shape-Areas if a sound volume entity is connected. Takes OuterRadius of sound entity into account to determine when an entity is near the area.
OnLeaveArea	Called when an entity has fully left an area or trigger. Parameters include: • **areaId** (int) • **fade fraction** (float) This value is always 0.0f.
OnLeaveNearArea	Called when an entity leaves the range of an area. Works with Box-, Sphere- and Shape-Areas if a sound volume entity is connected. Takes OuterRadius of sound entity into account to determine when an entity is near the area.
OnMove	Called whenever an entity moves through the world.
OnMoveNearArea	Called when an entity moves. Works with Box-, Sphere- and Shape-Areas if a sound volume entity is connected. Takes OuterRadius of sound entity into account to determine when an entity is near the area.
OnProceedFadeArea	Called when an entity has recently entered an area and fading is still in progress. Parameters include: • **areaId** (int) • **fade fraction** (float)
OnSoundDone	Called when a sound stops. Parameters include:

Callback Function	Description
	• **soundId** (int) The ID of the sound played, which was provided with the request to play the sound.
OnStartGame	Called when a game is started.
OnStartLevel	Called when a new level is started.
OnTimer	Called when a timer expires. Parameters include: • **timerId** (int) The ID of the time, provided by `Entity.SetTimer()`. • **period** (int) Length of time, in milliseconds, that the timer runs
OnUnBind	Called when a child entity is about to be detached from an entity. Parameters include: • script table for the child entity
OnUpdate	Called periodically by the engine on the entity's current state. This assumes the console variable `es_UpdateScript` is set to 1.

Game Rules Script Callbacks

CryLua is deprecated and will be removed in a future version of Lumberyard.

This topic provides reference information on callbacks used with the GameRules scripts.

Callback Function	Description
OnAddTaggedEntity	Called when a player is added as a tagged player on the minimap. Called on the server only. • **shooterId** – Entity that tagged the target player. • **targetId** – Tagged player.
OnClientConnect	Called when a player connects. Called on the server only. • **channelId**
OnClientDisconnect	Called when a player disconnects. Called on the server only. • **channelId**
OnClientEnteredGame	Called when a player enters the game and is part of the game world. Called on the server only. • **channelId** – Channel identifier of the player. • **playerScriptTable** – The player's script table. • **bReset** – Boolean indicating whether or not the channel is from the reset list. • **bLoadingSaveGame** – Boolean indicating whether or not the call was made during a saved game loading.
OnDisconnect	Called when the player disconnects on the client. Called on the client only.

Callback Function	Description
	• **cause** – Integer identifying the disconnection cause. See `EDisconnectionCause`. • **description** – Human readable description of the disconnection cause.
OnChangeSpectatorMode	Called when a player changes the spectator mode. Called on the server only. • **entityId** – Player who made the change. • **mode** – New spectator mode (1=fixed, 2=free, 3= follow). • **targetId** – Possible target entity to spectate. • **resetAll** – Boolean indicating whether or not to reset player-related things like the inventory.
OnChangeTeam	Called when a player switches teams. Called on the server only. • **entityId** – Player who switched teams. • **teamId** – New team identifier.

Callback Function	Description
OnExplosion	Called when an explosion is simulated. Called on the server and client. • **pos** – Position of the explosion in the game world. • **dir** – Direction of the explosion. • **shooterId** • **weaponId** • **shooter** • **weapon** • **materialId** • **damage** • **min_radius** • **radius** • **pressure** • **hole_size** • **effect** • **effectScale** • **effectClass** • **typeId** • **type** • **angle** • **impact** • **impact_velocity** • **impact_normal** • **impact_targetId** • **shakeMinR** • **shakeMaxR** • **shakeScale** • **shakeRnd** • **impact** • **impact_velocity** • **impact_normal** • **impact_targetId** • **AffectedEntities** – Affected entities table. • **AffectedEntitiesObstruction** – Affected entities obstruction table.

Common Lua Globals and Functions

CryLua is deprecated and will be removed in a future version of Lumberyard.

- File location: `Game/Scripts/common.lua`
- Loaded from: `Game/Scripts/main.lua`

Globals

Use the following globals to avoid temporary Lua memory allocations:

Name	Description
g_SignalData_point	Basic 3D vector value used by **g_SignalData.**
g_SignalData_point2	Basic 3D vector value used by **g_SignalData.**
g_SignalData	Used to pass signal data in AI behavior scripts (see: Signals (p. 88)).
g_StringTemp1	Commonly used for temporary strings inside Lua functions.
g_HitTable	Commonly used by the **Physics.RayWorldIntersection** function.

A **g_HitTable** used with **Physics.RayWorldIntersection** can contain the following parameters:

Parameter	Description
pos	3D vector world coordinates of the ray hit.
normal	3D normal vector of the ray hit.
dist	Distance of the ray hit.
surface	Type of surface hit.
entity	Script table of entity hit (if one was hit).
renderNode	Script handle to a foliage or static render node.

A **g_SignalData** table can contain the following parameter types:

Type	Description
Vec3	3D vector.
ScriptHandle	Normally used to pass along an entity ID.
Floating Point	Floating point value.
Integer	Integer or number value.
String	String value.

AIReload()

Reloads the **aiconfig.lua** Lua script (`Game/Scripts/AI/`).

AIDebugToggle()

Toggles the ai_DebugDraw console variable on and off.

ShowTime()

Logs the current system time to the console. Format is Day/Month/Year, Hours:Minutes.

count()

Returns the number of key-value pairs in a given table.

Parameter	Description
_tbl	Table to retrieve the number of key-value pairs from.

new()

Creates a new table by copying an specified existing table. This function is commonly used to create a local table based on an entity parameter table.

Parameter	Description
_obj	Existing table you want to create a new one from.
norecurse	Flag indicating whether or not to recursively recreate all sub-tables. If set to TRUE, sub-tables will not be recreated.

merge()

Merges two tables without merging functions from the source table.

Parameter	Description
dst	Destination table to merge source table information into.
src	Source table to merge table information from.
recurse	Flag indicating whether or not to recursively merge all sub-tables.

mergef()

Merges two tables including merging functions from the source table.

Parameter	Description
dst	Destination table to merge source table information into.
src	Source table to merge table information from.
recursive	Flag indicating whether or not to recursively merge all sub-tables.

Vec2Str()

Converts a 3D vector table into a string and returns it in the following format: (x: X.XXX y: Y.YYY z: Z.ZZZ).

Parameter	Description
vec	3D vector table to convert. Example: `{x=1,y=1,z=1}`.

LogError()

Logs an error message to the console and the log file. Message appears in red text in the console.

Parameter	Description
fmt	Formatted message string.
...	Optional argument list. For example: `LogError("MyError: %f", math.pi);`

LogWarning()

Logs a warning message to the console and the log file. Message appears in yellow text in the console.

Parameter	Description
fmt	Formatted message string.
...	Optional argument list. For example: `LogWarning("MyError: %f", math.pi);`

Log()

Logs a message to the console and the log file. Commonly used for debugging purposes.

Parameter	Description
fmt	Formatted message string.
...	Optional argument list. For example: `Log("MyLog: %f", math.pi);`

dump()

Dumps information from a specified table to the console.

Parameter	Description
_class	Table to dump to console. For example: `g_localActor`
no_func	Flag indicating whether or not to dump the table functions.

Parameter	Description
depth	Depth of the tables tree dump information from.

EmptyString()

Checks whether or not a given string is set and its length is greater than zero. Returns `TRUE` or `FALSE`.

Parameter	Description
str	String to check for.

NumberToBool()

Checks whether or not a number value is true (non-zero) or false (zero).

Parameter	Description
n	Number to check for.

EntityName()

Retrieves the name of a specified entity ID or entity table. If the entity doesn't exist, this function returns an empty string.

Parameter	Description
entity	Entity table or entity ID to return a name for.

EntityNamed()

Checks whether or not an entity with the specified name exists in the entity system. Returns `TRUE` or `FALSE`. Commonly used for debugging.

Parameter	Description
name	Name of entity to check for.

SafeTableGet()

Checks whether or not a sub-table with a specified name exists in a table. If the sub-table exists, this function returns it; otherwise the function returns `nil`.

Parameter	Description
table	Table to check for the existence of a sub-table.

Parameter	Description
name	Sub-table name to check for.

EntityUtils Lua Functions

CryLua is deprecated and will be removed in a future version of Lumberyard.

This topic describes the commonly used Lua entity utility functions.

- File location: `Game/Scripts/Utils/EntityUtils.lua`
- Loaded from: `Game/Scripts/common.lua`

DumpEntities()

Dumps to console all entity IDs, names, classes, positions, and angles that are currently used in a map. For example:

```
[userdata: 00000002]..name=Grunt1 clsid=Grunt pos=1016.755,1042.764,100.000
 ang=0.000,0.000,1.500
[userdata: 00000003]..name=Grunt2 clsid=Grunt pos=1020.755,1072.784,100.000
 ang=0.000,0.000,0.500
...
```

CompareEntitiesByName()

Compares two entities identified by name. This function is commonly used when sorting tables.

Parameter	Description
ent1	Name of first entity table.
ent2	Name of second entity table.

Example

```
local entities = System.GetEntitiesByClass("SomeEntityClass")
table.sort(entities, CompareEntitiesByName)
```

CompareEntitiesByDistanceFromPoint()

Compares the distance of two entities from a specified point. If the distance is greater for Entity 1 than for Entity 2 (that is, Entity 1 is further away), this function returns `TRUE`, otherwise it returns `FALSE`.

Parameter	Description
ent1	Entity 1 table

Parameter	Description
ent2	Entity 2 table
point	3D position vector identifying the point to measure distance to.

Example

```
local ent1 = System.GetEntityByName("NameEntityOne")
local ent2 = System.GetEntityByName("NameEntityTwo")

if(CompareEntitiesByDistanceFromPoint( ent1, ent2, g_localActor:GetPos()))then
   Log("Entity One is further away from the Player than Entity two...")
end
```

BroadcastEvent()

Processes an entity event broadcast.

Parameter	Description
sender	Entity that sent the event.
event	String based entity event to process.

Example

```
BroadcastEvent(self, "Used")
```

MakeDerivedEntity()

Creates a new table that is a derived version of a parent entity table. This function is commonly used to simplify the creation of a new entity script based on another entity.

Parameter	Description
_DerivedClass	Derived class table.
_Parent	Parent or base class table.

MakeDerivedEntityOverride()

Creates a new table that is a derived class of a parent entity. The derived table's properties will override those from the parent.

Parameter	Description
_DerivedClass	Derived class table.

Parameter	Description
_Parent	Parent or base class table.

MakeUsable()

Adds usable functionality, such as an **OnUsed** event, to a specified entity.

Parameter	Description
entity	Entity table to make usable.

Example

```
MyEntity = { ... whatever you usually put here ... }

MakeUsable(MyEntity)

function MyEntity:OnSpawn() ...

function MyEntity:OnReset()
  self:ResetOnUsed()
  ...
end
```

MakePickable()

Adds basic "pickable" functionality to a specified entity. The `bPickable` property is added to the entity's properties table.

Parameter	Description
entity	Entity table to make pickable.

MakeSpawnable()

Adds spawn functionality to a specified entity. Commonly used for **AI** actors during creation.

Parameter	Description
entity	Entity table to make spawnable.

EntityCommon.PhysicalizeRigid()

Physicalizes an entity based on the specified entity slot and its physics properties.

Parameter	Description
entity	Entity table to physicalize.

Parameter	Description
nSlot	Entity slot to physicalize.
Properties	Physics properties table
bActive	Not used.

Lua Vector and Math Functions

CryLua is deprecated and will be removed in a future version of Lumberyard.

This topic describes the commonly used math global vectors, constants, and functions.

- File location: `Game/Scripts/Utils/Math.lua`
- Loaded from: `Game/Scripts/common.lua`

Global Vectors

The following globals should be used to avoid temporary Lua memory allocations:

Global Name	Description
g_Vectors.v000	Basic zero vector.
g_Vectors.v001	Positive z-axis direction vector.
g_Vectors.v010	Positive y-axis direction vector.
g_Vectors.v100	Positive x-axis direction vector.
g_Vectors.v101	The x and z-axis direction vector.
g_Vectors.v110	The x and y-axis direction vector.
g_Vectors.v111	The x, y and z-axis vector.
g_Vectors.up	Positive z-axis direction vector.
g_Vectors.down	Negative z-axis direction vector.
g_Vectors.temp	Temporary zero vector.
g_Vectors.tempColor	Temporary zero vector. Commonly used for passing rgb color values.
g_Vectors.temp_v1	Temporary zero vector.
g_Vectors.temp_v2	Temporary zero vector.
g_Vectors.temp_v3	Temporary zero vector.
g_Vectors.temp_v4	Temporary zero vector.
g_Vectors.vecMathTemp1	Temporary zero vector.

Global Name	Description
g_Vectors.vecMathTemp2	Temporary zero vector.

Constants

Constant Name	Description
g_Rad2Deg	Basic radian-to-degree conversion value.
g_Deg2Rad	Basic degree-to-radian conversion value.
g_Pi	Basic Pi constant based on `math.pi`.
g_2Pi	Basic double-Pi constant based on `math.pi`.
g_Pi2	Basic half-Pi constant based on `math.pi`.

IsNullVector()

Checks whether or not all components of a specified vector are null.

Parameter	Description
a	Vector to check.

IsNotNullVector()

Checks whether or not any components of a specified vector is not null.

Parameter	Description
a	Vector to check.

LengthSqVector()

Retrieves the squared length of a specified vector.

Parameter	Description
a	Vector to retrieve length for.

LengthVector()

Retrieves the length of a specified vector.

Parameter	Description
a	Vector to retrieve length for.

DistanceSqVectors()

Retrieves the squared distance between two vectors.

Parameter	Description
a	First vector.
b	Second vector.

DistanceSqVectors2d()

Retrieves the squared distance between two vectors in 2D space (without z-component).

Parameter	Description
a	First vector.
b	Second vector.

DistanceVectors()

Retrieves the distance between two vectors.

Parameter	Description
a	First vector.
b	Second vector.

dotproduct3d()

Retrieves the dot product between two vectors.

Parameter	Description
a	First vector.
b	Second vector.

dotproduct2d()

Retrieves the dot product between two vectors in 2D space (without z-component).

Parameter	Description
a	First vector.
b	Second vector.

LogVec()

Logs a specified vector to console.

Parameter	Description
name	Descriptive name of the vector.
v	Vector to log.

Example

```
LogVec("Local Actor Position", g_localActor:GetPos())
```

Console output:

```
<Lua> Local Actor Position = (1104.018066 1983.247925 112.769440)
```

ZeroVector()

Sets all components of a specified vector to zero.

Parameter	Description
dest	Vector to zero out.

CopyVector()

Copies the components of one vector to another.

Parameter	Description
dest	Destination vector.
src	Source vector.

SumVectors()

Adds up the components of two vectors.

Parameter	Description
a	First vector.
b	Second vector.

NegVector()

Negates all components of a specified vector.

Parameter	Description
a	Vector to negate.

SubVectors()

Copies the componentwise subtraction of two vectors to a destination vector.

Parameter	Description
dest	Destination vector.
a	First vector.
b	Second vector.

FastSumVectors()

Copies the componentwise addition of two vectors to a destination vector.

Parameter	Description
dest	Destination vector.
a	First vector.
b	Second vector.

DifferenceVectors()

Retrieves the difference between two vectors.

Parameter	Description
a	First vector.
b	Second vector.

FastDifferenceVectors()

Copies the componentwise difference between two vectors to a destination vector.

Parameter	Description
dest	Destination vector.
a	First vector.
b	Second vector.

ProductVectors()

Retrieves the product of two vectors.

Parameter	Description
a	First vector.
b	Second vector.

FastProductVectors()

Copies the product of two vectors to a destination vector.

Parameter	Description
dest	Destination vector.
a	First vector.
b	Second vector.

ScaleVector()

Scales a specified vector *a* by a factor of *b*.

Parameter	Description
a	Vector.
b	Scalar.

ScaleVectorInPlace(a,b)

Retrieves a new vector based on a copy of vector *a* scaled by a factor *b*.

Parameter	Description
a	First vector.
b	Scalar.

ScaleVectorInPlace(dest,a,b)

Copies vector *a* scaled by the factor of *b* to a destination vector.

Parameter	Description
dest	Destination vector.
a	First vector.
b	Scalar.

NormalizeVector()

Normalizes a specified vector.

Parameter	Description
a	Vector to normalize.

VecRotate90_Z()

Rotates a specified vector by 90 degree around the z-axis.

Parameter	Description
v	Vector to rotate.

VecRotateMinus90_Z()

Rotates a specified vector by -90 degree around the z-axis.

Parameter	Description
v	Vector to rotate.

crossproduct3d()

Copies the result of the cross product between two vectors to a destination vector.

Parameter	Description
dest	Destination vector.
p	First vector.
q	Second vector.

RotateVectorAroundR()

Copies to a destination vector the result of the vector rotation of vector *p* around vector *r* by a specified angle.

Parameter	Description
dest	Destination vector.
p	First vector.
r	Second vector.
angle	Rotation angle.

ProjectVector()

Copies to a destination vector the result of the vector projection of vector *P* to the surface with a specified normal *N*.

Parameter	Description
dest	Destination vector.
P	Vector to project.
N	Surface normal.

DistanceLineAndPoint()

Retrieves the distance between point *a* and the line between *p* and *q*.

Parameter	Description
a	Point to measure from.
p	Vector p.
q	Vector q.

LerpColors()

Performs linear interpolation between two color/vectors with a factor of *k*.

Parameter	Description
a	Color/vector a.
b	Color/vector b.
k	Factor k.

Lerp()

Performs linear interpolation between two scalars with a factor of *k*.

Parameter	Description
a	Scalar a.
b	Scalar b.
k	Factor k.

__max()

Retrieves the maximum of two scalars.

Parameter	Description
a	Scalar a.
b	Scalar b.

__min()

Retrieves the minimum of two scalars.

Parameter	Description
a	Scalar a.
b	Scalar b.

clamp()

Clamps a specified number between minimum and maximum.

Parameter	Description
_n	Number to clamp.

Parameter	Description
_min	Lower limit.
_max	Upper limit.

Interpolate()

Interpolates a number to a specified goal by a specified speed.

Parameter	Description
actual	Number to interpolate.
goal	Goal.
speed	Interpolation speed.

sgn()

Retrieves the sign of a specified number (0 returns 0).

Parameter	Description
a	Number to get sign for.

sgnnz()

Retrieves the sign of a specified number (0 returns 1).

Parameter	Description
a	Number to get sign for.

sqr()

Retrieves the square of a specified number.

Parameter	Description
a	Number to square.

randomF()

Retrieves a random float value between two specified numbers.

Parameter	Description
a	First number.
b	Second number.

iff()

Checks the condition of a test value and returns one of two other values depending on whether the test value is `nil` or not.

Parameter	Description
c	Test value.
a	Return value if test value is not nil.
b	Return value if test value is nil.

Physics Lua Functions

CryLua is deprecated and will be removed in a future version of Lumberyard.

These functions are commonly used to register new explosion and crack shapes in the physics engine.

File location: `Game/Scripts/physics.lua`

- Loaded from: `Game/Scripts/main.lua`

Physics.RegisterExplosionShape()

Registers a boolean carving shape for breakable objects in the physics engine.

Parameter	Description
sGeometryFile	Name of a boolean shape cgf file.
fSize	Shape's characteristic size.
BreakId	Breakability index (0-based) used to identify the breakable material.
fProbability	Shape's relative probability; when several shapes with the same size appear as candidates for carving, these relative probabilities are used to select one.
sSplintersfile	Name of a splinters cgf file, used for trees to add splinters at the breakage location.
fSplintersOffset	Size offset for the splinters.
sSplintersCloudEffect	Name of splinters particle fx; this effect is played when a splinters-based constraint breaks and splinters disappear.

Physics.RegisterExplosionCrack()

Registers a new explosion crack for breakable objects in the physics engine.

Parameter	Description
sGeometryFile	Name of a crack shape cgf file. This type of file must have three helpers to mark the corners, named "1","2" and "3".
BreakId	Breakability index (0-based) used to identify the breakable material.

Lua ScriptBind Reference

CryLua is deprecated and will be removed in a future version of Lumberyard.

You can use `ScriptBind` functions in Lua scripts to call legacy code written in C++.

Topics

- ScriptBind Engine Functions (p. 743)
- ScriptBind Action Functions (p. 899)
- ScriptBind_Boids (p. 945)

ScriptBind Engine Functions

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists C++ engine functions that you can call from Lua script.

Topics

- ScriptBind_AI (p. 743)
- ScriptBind_Entity (p. 810)
- ScriptBind_Movie (p. 862)
- ScriptBind_Particle (p. 863)
- ScriptBind_Physics (p. 866)
- ScriptBind_Script (p. 869)
- ScriptBind_Sound (p. 871)
- ScriptBind_System (p. 873)

ScriptBind_AI

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists C++ AI functions that can be called from Lua scripts.

AbortAction

Aborts execution of a specified action.

Syntax

```
AI.AbortAction(userId [, actionId ])
```

Parameter	Description
`userId`	The ID of the entity.
`actionId (optional)`	Unique ID of the action to be aborted. If 0 (or nil), all actions on the specified entity are aborted.

AddAggressiveTarget

Adds a target to a specified entity's list as an aggressive potential target.

Syntax

```
AI.AddAggressiveTarget(entityId, targetId)
```

Returns True if successfully added.

Parameter	Description
`entityId`	The ID of the entity.
`targetId`	Target's entity ID to add.

AddCombatClass

Creates new combat class.

Syntax

```
AI.AddCombatClass(int combatClass, SmartScriptTable pTable, const char* szCustomSignal)
```

Parameter	Description
`combatClass`	Combat class to add.
`pTable`	Parameters table.
`szCustomSignal`	Specifies optional custom `OnSeen` signal.

AddFormationPoint

Adds a follow-type node to a formation descriptor.

Syntax

```
AI.AddFormationPoint(name, sightangle, distance, offset, [unit_class [,distanceAlt,
 offsetAlt]])
```

Parameter	Description
`name`	Name of the formation descriptor.
`sightangle`	Angle of sight of the node (-180,180; 0 = the entity looks forward).
`distance`	Distance from the formation's owner.
`offset`	Offset along the following line (negative = left, positive = right).
`unit_class`	Class of soldier (see `eSoldierClass` definition in `IAgent.h`). `DistanceAlt` (optional): alternative distance from the formation owner `offsetAlt` (optional): alternative offset.

AddFormationPointFixed

Adds a node with a fixed offset to a formation descriptor.

Syntax

```
AI.AddFormationPointFixed(name, sightangle, x, y, z [,unit_class])
```

Parameter	Description
`name`	Name of the formation descriptor.
`sightangle`	Angle of sight of the node (-180,180; 0 = the entity looks forward).
`x, y, z`	Offset from formation owner.
`unit_class`	Class of soldier (see `eSoldierClass` definition in `IAgent.h`).

AddPatternBranch

Creates a branch pattern at the specified node. When the entity has approached the specified node (nodeName) and it is time to choose a new point, the rules defined by this function are used to select the new point. This function can associate multiple target points and an evaluation rule.

Syntax

```
AI.AddPatternBranch(nodeName, method, branchNode1, branchNode2, ..., branchNodeN)
```

Parameter	Description
`nodeName`	Name of the node to add branches to.
`method`	Method used to choose the next node. Valid values include:

Parameter	Description
	• AITRACKPAT_CHOOSE_ALWAYS – Choose the next point from the list in linear sequence. • AITRACKPAT_CHOOSE_LESS_DEFORMED – Choose the least deformed point in the list. Each node is associated with a deformation value (percentage), which describes how much it must move in order to stay within the physical world. These deformation values are summed down to the parent nodes so that deformation at the end of the hierarchy will be caught down the hierarchy. • AITRACKPAT_CHOOSE_RANDOM – Choose a point in the list randomly.

AddPatternNode

Adds a point to the track pattern.

When validating the points, the test is performed from the start position to the end position. Start position is either the pattern origin or, if the parent is provided, the parent position. The end position is either the relative offset from the start position or from the pattern origin; this position is chosen based on the node flag. The offset is clamped to the physical world based on the test method. The points will be evaluated in the same order they are added to the descriptor, and the system does not try to correct the evaluation order. If hierarchies are used (parent name is defined), it is up to the pattern creator to make sure the nodes are created in such an order that the parent is added before it is referenced.

Syntax

```
AI.AddPatternNode(nodeName, offsetx, offsety, offsetz, flags, [parent], [signalValue])
```

Parameter	Description
nodeName	Name of the new point. , Point names are local to the current pattern.
offsetx, offsety, offsetz	Offset from the start position or from the pattern center. See AITRACKPAT_NODE_ABSOLUTE.
flags	Track pattern functionality flags. Node evaluation flags: • AITRACKPAT_NODE_START – Node can be used as the first node in the pattern. There can be multiple start nodes. In that case the closest one is chosen. • AITRACKPAT_NODE_ABSOLUTE – Interpret offset as an offset from the pattern center (otherwise the offset is from the start position). • AITRACKPAT_NODE_SIGNAL – A signal "OnReachedTrackPatternNode" will be sent when the node is reached. • AITRACKPAT_NODE_STOP – Advancing will be stopped. It can be continued by calling entity:ChangeAIParameter(AIPARAM_TRACKPATTERN_ADVANCE, 1). • AITRACKPAT_NODE_DIRBRANCH – For the direction at each pattern node, use the average direction to the branch nodes (otherwise use the direction from the node position to the center of the pattern).
parent (optional)	Parent node position, which will be used as the start position instead of the pattern center.

Parameter	Description
`signalValue (optional)`	If the signal flag is set, this value is passed as a signal parameter, accessible from the signal handler in `data.iValue`.

AddPersonallyHostile

Syntax

```
AI.AddPersonallyHostile(ScriptHandle entityID, ScriptHandle hostileID)
```

AgentLookAtPos

Causes the specified entity to look at a certain position.

Syntax

```
AI.AgentLookAtPos(entityId, Vec3 pos)
```

Parameter	Description
`entityId`	The ID of the entity.
`pos`	Vec3 to look at.

AllowLowerBodyToTurn

Syntax

```
AI.AllowLowerBodyToTurn(entityID, bAllowLowerBodyToTurn)
```

Parameter	Description
`entityId`	Entity ID of the agent you want to set the look style to.
`bAllowLowerBodyToTurn.`	True if you want to allow the turning movement of the body, false otherwise.

BeginTrackPattern

Begins the definition of a new track pattern descriptor. The pattern is created by calling `AI.AddPatternPoint()` and `AI.AddPatternBranch()`, and finalized by calling `AI.EndTrackPattern()`.

Syntax

```
AI.BeginTrackPattern(patternName, flags, validationRadius, [stateTresholdMin],
```

Parameter	Description
`patternName`	Name of the new track pattern descriptor.

Parameter	Description
`flags`	Track pattern functionality flags. Validation flags describe how the pattern is validated to fit the physical world: • `AITRACKPAT_VALIDATE_NONE` – No validation. `AITRACKPAT_VALIDATE_SWEPTSPHERE` – Validate using swept sphere tests, where the spehre radius equals the validation radius plus the entity pass radius. • `AITRACKPAT_VALIDATE_RAYCAST` – Validate using raycasting, where the hit position is pulled back by the amount of validation radius plus the entity pass radius. Alignment flags describe how, when the pattern is selected to be used, the alignment of the pattern can be changed. Flags are evaluated in the following order: • `AITRACKPAT_ALIGN_TO_TARGET` – Align the pattern so that the y-axis points towards the target each time it is set. If the agent does not have a valid attention target at the time, the pattern is aligned to the world. • `AITRACKPAT_ALIGN_RANDOM` – Align the pattern randonly each time it is set. The rotation ranges are set using `SetRandomRotation()`.
`validationRadius`	Validation radius is added to the entity pass radius when validating the pattern along the offsets.
`stateTresholdMin (optional)`	If the state of the pattern is 'enclosed' (high deformation) and the global deformation < stateTresholdMin, the state becomes exposed. Default 0.35.
`stateTresholdMax (optional)`	If the state of the pattern is 'exposed' (low deformation) and the global deformation > stateTresholdMax, the state becomes enclosed. Default 0.4.
`globalDeformTreshold (optional)`	Deformation of the whole pattern is tracked in range [0..1]. This treshold value can be used to clamp the bottom range, so that values in range [trhd..1] becomes [0..1], default 0.0.
`localDeformTreshold (optional)`	Deformation of the each node is tracked in range [0..1]. This treshold value can be used to clamp the bottom range, so that values in range [trhd..1] becomes [0..1], default 0.0.
`exposureMod (optional)`	Importance of the node exposure (how much it is seen by the tracked target) to consider when branching. Valid range is [-1..1], where -1 means to favor unseen nodes and 1 means to favor seen, exposed nodes. Default is 0 (no effect).
`randomRotAng (optional)`	Flag indicating whether or not to randomly rotate the pattern each time it is set. Rotation is performed in XYZ order. This parameter defines angles (in degrees) around each axis.

CanFireInStance

Syntax

```
AI.CanFireInStance(entityId, stance)
```

Returns true if AI can fire at his target in the specified stance at his current position

Parameter	Description
`entityId`	The ID of the entity.
`stance.`	Stance Id (STANCE_*).

CanMelee

Determines whether or not the AI is able to do melee attack.

Syntax

```
AI.CanMelee(entityId)
```

Returns True or false (1 or 0).

Parameter	Description
`entityId`	The ID of the entity.

CanMoveStraightToPoint

Determines whether or not a specified entity can move in a straight line from its current position to a specified point.

Syntax

```
AI.CanMoveStraightToPoint(entityId, position)
```

Parameter	Description
`entityId`	The ID of the entity.
`position`	Position to check path to.

ChangeFormation

Changes the formation descriptor for the current formation of a specified entity's group (if one exists).

Syntax

```
AI.ChangeFormation(entityId, name [,scale])
```

Returns True if the formation change was successful.

Parameter	Description
`entityId`	Unique entity ID used to identify the group.

Parameter	Description
`name`	Name of the formation descriptor.
`scale (optional)`	Scale factor for the formation (1 = default).

ChangeMovementAbility

Changes the value of an AI movement ability parameter for the entity specified.

Syntax

```
AI.ChangeMovementAbility(entityId, paramEnum, paramValue)
```

Parameter	Description
`entityId`	The ID of the entity.
`paramEnum`	Index of the parameter to change. Valid values include: • `AIMOVEABILITY_OPTIMALFLIGHTHEIGHT` – Optimal flight height in meters while finding path. • `AIMOVEABILITY_MINFLIGHTHEIGHT` – Minimum flight height in meters while finding path. • `AIMOVEABILITY_MAXFLIGHTHEIGHT` – Maximum flight height in meters while finding path.
`paramValue`	New value for the specified parameter.

ChangeParameter

Updates a parameter value for a specified entity.

Syntax

```
AI.ChangeParameter(entityId, paramEnum, paramValue)
```

Parameter	Description
`entityId`	The ID of the entity.
`paramEnum`	The enum of the parameter.
`paramValue`	The new value for the specified parameter.

CheckForFriendlyAgentsAroundPoint

Syntax

```
AI.CheckForFriendlyAgentsAroundPoint(ScriptHandle entityID, Vec3 point, float radius)
```

CheckMeleeDamage

Determines whether or not the AI performing melee is actually hitting target.

Syntax

```
AI.CheckMeleeDamage(entityId, targetId, radius, minheight, maxheight, angle)
```

Returns (distance,angle) pair between entity and target (degrees) if melee is possible, nil otherwise

Parameter	Description
`entityId`	The ID of the entity.
`targetId`	Target's entity ID.
`radius.`	max distance in 2d to target.
`minheight.`	min distance in height.
`maxheight.`	max distance in height.
`angle.`	FOV to include target.

ClearAnimationTag

Syntax

```
AI.ClearAnimationTag(ScriptHandle entityID, const char* tagName)
```

Parameter	Description
`entityId`	AI's entity.
`tagName.`	.

ClearMovementContext

Resets the specified entity's movement context.

Syntax

```
AI.ClearMovementContext(entityId)
```

Parameter	Description
`entityId`	The ID of the entity.
`context.`	context value.

ClearPotentialTargets

Clears all the potential targets from a specified entity's perception handler.

Syntax

```
AI.ClearPotentialTargets(entityId)
```

Parameter	Description
`entityId`	The ID of the entity.

ClearTempTarget

Removes the specified entity's temporary potential target so that it is no longer considered for target selection.

Syntax

```
AI.ClearTempTarget(entityId)
```

Returns True if successfully updated.

Parameter	Description
`entityId`	The ID of the entity.

ConstrainPointInsideGenericShape

Syntax

```
AI.ConstrainPointInsideGenericShape(position, shapeName[, checkHeight])
```

Returns Nearest point inside the specified shape.

Parameter	Description
`position`	Position to check.
`shapeName`	Name of the shape to test (returned by `AI.GetEnclosingGenericShapeOfType`).
`checkHeight (optional)`	Flag indicating whether or not to test for shape height. (default=false). If set to true, the test will check the space between shape.aabb.min.z and shape.aabb.min.z+shape.height.

CreateFormation

Creates a formation descriptor and adds a fixed node at 0,0,0 (owner's node).

Syntax

```
AI.CreateFormation(name)
```

Parameter	Description
`name`	Name of the new formation descriptor.

CreateGroupFormation

Creates a group formation with leader (or updates leader).

Syntax

```
AI.CreateGroupFormation(entityId, leaderId)
```

Parameter	Description
`entityId`	AI's entity.
`leaderId.`	New leader.

CreateStimulusEvent

Creates a target track stimulus event for the specified entity.

Syntax

```
AI.CreateStimulusEvent(ScriptHandle ownerId, ScriptHandle targetId, const char*
 stimulusName, SmartScriptTable pData)
```

Parameter	Description
`ownerId`	Unique ID of the entity that owns and receives the event.
`targetId`	Unique ID of the entity that sends the event and becomes the target.
`stimulusName`	Name of the stimulus event.
`pData`	Event data (see `TargetTrackHelpers::SStimulusEvent`).

CreateTempGenericShapeBox

Creates a temporary box-shaped generic shape. This temporary shape will be destroyed upon an AI system reset.

Syntax

```
AI.CreateTempGenericShapeBox(Vec3 center, float radius, float height, int type)
```

Returns Shape name.

Parameter	Description
`center.`	Center point of the box.

Parameter	Description
`radius.`	Size of the box in x and y directions.
`height`	Height of the box.
`type`	Box shape type (AIAnchor).

DebugReportHitDamage

Creates a debug report for the hit damage.

Syntax

```
AI.DebugReportHitDamage(pVictimEntity, pShooterEntity)
```

Parameter	Description
`pVictimEntity.`	Victim ID.
`pShooterEntity.`	Shooter ID.

DestroyAllTPSQueries

Destroys all the tactical point system queries.

Syntax

```
AI2.DestroyAllTPSQueries()
```

DistanceToGenericShape

Syntax

```
AI.DistanceToGenericShape(Vec3 position, const char* shapeName[, int checkHeight])
```

Returns True if the point is inside the specified shape.

Parameter	Description
`position`	Position to check.
`shapeName`	Name of the shape to test (returned by `AI.GetEnclosingGenericShapeOfType`).
`checkHeight (optional)`	Flag indicating whether or not to test for shape height. (default=false). If set to true, the test will check the space between shape.aabb.min.z and shape.aabb.min.z+shape.height.

DropTarget

Clears the target from a specified entity's perception handler.

Syntax

```
AI.DropTarget(entityId, targetId)
```

Parameter	Description
`entityId`	The ID of the entity.
`targetId`	Target's entity ID.

EnableCoverFire

Enables or disables fire when the `FIREMODE_COVER` is selected.

Syntax

```
AI.EnableCoverFire(entityId, enable)
```

Parameter	Description
`entityId`	The ID of the entity.
`enable`	Boolean.

EnableFire

Enables or disables fire.

Syntax

```
AI.EnableFire(entityId, enable)
```

Parameter	Description
`entityId`	The ID of the entity.
`enable`	Boolean.

EnableUpdateLookTarget

Syntax

```
AI.EnableUpdateLookTarget(ScriptHandle entityID, bool bEnable)
```

EnableWeaponAccessory

Enables or disables certain weapon accessory usage.

Syntax

```
AI.EnableWeaponAccessory(entityId, int accessory, bool state)
```

Parameter	Description
`entityId`	The ID of the entity.

`accessory`	Enum of the accessory to enable. Possible values (see `enum EAIWeaponAccessories` in the `IAgent.h` file):
	`AIWEPA_NONE = 0,`
	`AIWEPA_LASER = 0x0001`
	`AIWEPA_COMBAT_LIGHT = 0x0002`
	`AIWEPA_PATROL_LIGHT = 0x0004`

`state`	Set to true or false to enable or disable.

EndTrackPattern

Finalizes the track pattern definition. This function should always called to finalize the pattern. Failing to do so will cause erratic behavior.

Syntax

```
AI.EndTrackPattern()
```

Error

The fallback error message used when the system experiences an unhandled exception. The code following should continue if it is running in the editor so that the original cause of the problem can be fixed, but halt execution when it is running in the game.

Syntax

```
AI.Error(szMessage)
```

Parameter	Description
`szMessage`	The message to write to the log.

EvalPeek

Determines whether or not specified entity can peek from its current position.

Syntax

```
AI.EvalPeek(entityId [, bGetOptimalSide])
```

Returns One of the following values:

- -1 – don't need to peek

- 0 – cannot peek
- 1 – can peek from left
- 2 – can peek from right
- 3 – can peek from left & right

Parameter	Description
`entityId`	The ID of the entity.
`bGetOptimalSide (optional)`	Flag indicating whether or not to return the side that best fits the attention target's current location, if the AI object can peek from both sides. The default is false.

ExecuteAction

Executes an action on a set of participants.

Syntax

```
AI.ExecuteAction(action, participant1 [, participant2 [, ... , participantN ] ])
```

Parameter	Description
`action`	The smart object action name or ID.
`participant1`	The entity ID of the first participant in the action.
`participant2..N (optional)`	The entity ID of additional participants.

FindObjectOfType

Searches for the closest AIObject of a specified type in an area around a specified entity or position. Once an AIObject is found, it is devalued and can't be found again for a certain number of seconds (unless turned off in flags).

Syntax

```
AI.FindObjectOfType(entityId, radius, AIObjectType, flags [,returnPosition
 [,returnDirection]]) AI.FindObjectOfType(position, radius, AIObjectType, [,returnPosition
 [,returnDirection]])
```

Returns The found AIObject's name.

Parameter	Description
`entityId`	Unique entity ID used to determine the center position of the search.
`position`	Vector specifying the center position of the search.
`radius`	Radius of the search area.

Parameter	Description
`AIObjectType`	AIObject type to search for (see `ScriptBindAI.cpp` and `Scripts/AIAnchor.lua` for a complete list of AIObject types).
`flags.`	A combination of one or more of the following search filter flags: • `AIFAF_VISIBLE_FROM_REQUESTER` – Requires whoever is requesting the object to also have a line of sight to it. • `AIFAF_VISIBLE_TARGET`– Requires a line of sight between target and anchor. • `AIFAF_INCLUDE_DEVALUED` – Include devalued objects. • `AIFAF_INCLUDE_DISABLED` – Include disabled objects.
`returnPosition (optional)`	Position of the found object.
`returnDirection (optional)`	Direction of the found object.

FindStandbySpotInShape

Syntax

```
AI.FindStandbySpotInShape(centerPos, targetPos, anchorType)
```

FindStandbySpotInSphere

Syntax

```
AI.FindStandbySpotInSphere(centerPos, targetPos, anchorType)
```

FreeSignal

Sends a signal to anyone in a specified radius around a position.

Syntax

```
AI.FreeSignal(signalType, signalText, position, radius [, entityID [,signalExtraData]])
```

Parameter	Description
`signalType`	See `AI.Signal`.
`signalText`	See `AI.Signal`.
`position`	The center point ({x,y,z} coordinates) from which the signal is sent.
`radius`	The inside radius of the area to which the signal is sent.
`entityID`	Optional. The ID of an entity that is a member of a group that should not receive the signal. Entities whose group ID is the value specified will not be sent the signal.
`signalExtraData`	Optional. See `AI.Signal`.

GetAIObjectPosition

Retrieves a specified AIObject's position.

Syntax

```
AI.GetAIObjectPosition(entityId | AIObjectName)
```

Returns AI Object position vector {x,y,z}.

Parameter	Description
`entityId | AIObjectName`	Unique entity ID or AIObject name.

GetAnchor

Searches for the closest anchor of a specified type in an area around a specified entity. Once an anchor is found, it is devalued and can't be found again for a certain number of seconds (unless turned off in flags).

Syntax

```
AI.GetAnchor(entityId, radius, AIAnchorType, searchType [,returnPosition
 [,returnDirection]])
```

Returns The found anchor's name.

Parameter	Description
`entityId`	Unique entity ID used to determine the center position of the search.
`radius`	Radius of the search area. Alternatively a search range can be specified (min=minRad,max=maxRad).
`AIAnchorType`	Anchor type to search for. See `Scripts/AIAnchor.lua` for a complete list of anchor types available.
`searchType`	A combination of one or more of the following search filter flags: • `AIANCHOR_NEAREST` – Nearest anchor of the specified type (default). • `AIANCHOR_NEAREST_IN_FRONT` – Nearest anchor of the specified type inside the front cone of the entity. • `AIANCHOR_NEAREST_FACING_AT` – Nearest anchor of the specified type that is oriented towards entity's attention target • `AIANCHOR_RANDOM_IN_RANGE` – Random anchor of the specified type. • `AIANCHOR_NEAREST_TO_REFPOINT` – Anchor of the specified type that is nearest to the entity's reference point.
`(optional) returnPosition`	Position of the found object.
`(optional) returnDirection`	Direction of the found object.

GetAttentionTargetAIType

Retrieves the AI type (`AIOBJECT_*`) of a specified entity's attention target .

Syntax

```
AI.GetAttentionTargetAIType(entityId)
```

Returns Attention target's AI type, or `AIOBJECT_NONE` if no target.

Parameter	Description
`entityId`	The ID of the entity.

GetAttentionTargetDirection

Retrieves the direction of a specified entity's attention target.

Syntax

```
AI.GetAttentionTargetDirection(entityId, returnDir)
```

Returns Attention target's direction vector {x,y,z}, passed as a return value.

Parameter	Description
`entityId`	The ID of the entity.

GetAttentionTargetDistance

Retrieves the distance from a specified entity to its attention target.

Syntax

```
AI.GetAttentionTargetDistance(entityId)
```

Returns distance to the attention target.

Parameter	Description
`entityId`	The ID of the entity.

GetAttentionTargetEntity

Retrieves a specified entity's attention target entity (if it is a specified entity), or the owner entity of a dummy object's attention target (if there is an owner entity).

Syntax

```
AI.GetAttentionTargetEntity(ScriptHandle entityID)
```

Returns Attention target's entity.

Parameter	Description
`entityId`	The ID of the entity.

GetAttentionTargetOf

Retrieves a specified entity's attention target.

Syntax

```
AI.GetAttentionTargetOf(entityId)
```

Returns Name of attention target. Null if there is no target.

Parameter	Description
`entityId`	The ID of the entity.

GetAttentionTargetPosition

Retrieves the position of a specified entity's attention target.

Syntax

```
AI.GetAttentionTargetPosition(entityId, returnPos)
```

Returns Attention target 's position vector {x,y,z}, passed as a return value ().

Parameter	Description
`entityId`	The ID of the entity.

GetAttentionTargetThreat

Syntax

```
AI.GetAttentionTargetThreat(ScriptHandle entityID)
```

GetAttentionTargetType

Retrieves the type (`AITARGET_*`) of a specified entity's attention target.

Syntax

```
AI.GetAttentionTargetType(entityId)
```

Returns Attention target's type, or `AITARGET_NONE` if no target.

Parameter	Description
`entityId`	The ID of the entity.

GetAttentionTargetViewDirection

Retrieves the view direction of a specified entity's attention target.

Syntax

```
AI.GetAttentionTargetViewDirection(entityId, returnDir)
```

Returns Attention target's view direction vector {x,y,z}, passed as a return value.

Parameter	Description
`entityId`	The ID of the entity.

GetBeaconPosition

Gets the beacon position for a specified entity/object's group.

Syntax

```
AI.GetBeaconPosition(entityId | AIObjectName, returnPos)
```

Returns True if the beacon is found and the position set.

Parameter	Description
`entityId | AIObjectName`	Unique entity ID or AI object name.
`returnPos`	Beacon position vector {x,y,z}.

GetBehaviorBlackBoard

Retrieves a specified `AIActor` current behaviour's black board (a Lua table).

Syntax

```
AI.GetBehaviorBlackBoard(entity)
```

Returns black board – if there was one nil – Otherwise

Parameter	Description
`entityId or entityName.`	An `AIActor` identifier.

GetBehaviorVariable

Returns a behavior variable for the specified actor.

Syntax

```
AI.GetBehaviorVariable(ScriptHandle entityId, const char* variableName)
```

GetBiasedDirection

Retrieves biased direction of certain point.

Syntax

```
AI.GetBiasedDirection(entityId)
```

Parameter	Description
`entityId`	The ID of the entity.

GetCurrentHideAnchor

Retrieves the name of the anchor that the entity currently is using for cover.

Syntax

```
AI.GetCurrentHideAnchor(entityId)
```

Parameter	Description
`entityId`	The ID of the entity.

GetDirectAnchorPos

Retrieves the position of a cover point that a specified entity can use to directly attack its attention target.

Syntax

```
AI.GetDirectAttackPos(entityId, searchRange, minAttackRange)
```

Returns Point value, or none if no attack point is available.

Parameter	Description
`entityId`	The ID of the entity.
`AIAnchorType`	Anchor type (see `Scripts/AIAnchor.lua` for a complete list of anchor types).
`maxDist`	Maximum size of search range.

GetDirLabelToPoint

Retrieves a direction label (front=0, back=1, left=2, right_3, above=4, -1=invalid) to the specified point.

Syntax

```
AI.GetDirLabelToPoint(entityId, point)
```

Parameter	Description
`entityId`	The ID of the entity.
`point`	The point to evaluate.

GetEnclosingSpace

Returns the estimated surrounding navigable space in meters.

Syntax

```
AI.GetEnclosingSpace(entityId, Vec3 pos, float rad)
```

Parameter	Description
`entityId`	The ID of the entity.
`pos`	Check position.
`rad`	Check radius.

GetDistanceAlongPath

Retrieves the distance between a first and second entity, measured along the first entity's path.

Syntax

```
AI.GetDistanceAlongPath(entityId1, entityid2)
```

Returns Distance along a path. Value can be negative if the second entity is ahead along the path.

Parameter	Description
`entityId1`	ID for the first entity.
`entityId2`	ID for the second entity.

GetDistanceToClosestGroupMember

Syntax

```
AI.GetDistanceToClosestGroupMember(ScriptHandle entityId)
```

GetEnclosingGenericShapeOfType

Retrieves the first shape of a certain type that encloses a specified point.

Syntax

```
AI.GetEnclosingGenericShapeOfType(position, type[, checkHeight])
```

Returns Shape name.

Parameter	Description
`position`	Point to search for an enclosing shape.
`type`	Shape type to search for (uses anchor types).
`checkHeight (optional)`	Flag indicating whether or not to test for shape height. (default=false). If set to true, the test checks the space between shape.aabb.min.z and shape.aabb.min.z+shape.height.

GetExtraPriority

Retrieves the extra priority value for a specified enemy entity.

Syntax

```
AI.GetExtraPriority(enemyEntityId)
```

Parameter	Description
`enemyEntityId.`	The ID of the entity.

GetFactionOf

Retrieves the faction of the specified entity.

Syntax

```
AI.GetFactionOf(ScriptHandle entityID)
```

Returns the faction of the specified entity.

Parameter	Description
`entityId`	The ID of the entity whose faction to return.

GetFormationLookingPoint

Retrieves the looking point position inside the formation.

Syntax

```
AI.GetFormationLookingPoint(entityId)
```

Returns v3 – table with format {x,y,z} storing the looking point position

Parameter	Description
entityId	AI's entity.

GetFormationPointClass

Adds a follow-type node to a formation descriptor.

Syntax

```
AI.GetFormationPointClass(name, position)
```

Returns class of formation point (-1 if none found).

Parameter	Description
name	Name of the formation descriptor.
position	Point index in the formation (1..N).

GetFormationPointPosition

Retrieves an entity's formation point position.

Syntax

```
AI.GetFormationPointPosition(entityId, pos)
```

Returns true if the formation point has been found.

Parameter	Description
entityId	The ID of the entity.
pos	Return value for the position of the entity's formation point.

GetFormationPosition

Retrieves the relative position inside the formation.

Syntax

```
AI.GetFormationPosition(entityId)
```

Returns v3 – table with format {x,y,z} storing the relative position

Parameter	Description
`entityId`	AI's entity.

GetGroupAveragePosition

Retrieves the average position of a group's members.

Syntax

```
AI.GetGroupAveragePosition(entityId, properties, returnPos)
```

Returns the average position.

Parameter	Description
`entityId`	Unique entity ID used to determine the group.
`unitProperties`	Binary mask of unit properties type for which the attack is requested, in the following form: `UPR_* + UPR* (UPR_COMBAT_GROUND + UPR_COMBAT_FLIGHT)` See `IAgent.h` for a definition of unit properties UPR_*.

GetGroupCount

Retrieves the member count of a specified entity's group.

Syntax

```
AI.GetGroupCount(entityId, flags, type)
```

Returns the count of members for the specified group.

Parameter	Description
`entityId`	The entity or group ID.
`flags`	A combination of one or more of the following flags: • `GROUP_ALL` – Counts all agents in the group (default). • `GROUP_ENABLED` – Counts enabled agents only (exclusive with all). • `GROUP_MAX` – Include the maximum number of agents during the game (can be combined with all or enabled).
`type`	The AI object type for which to filter. Counts only the AI objects of the type specified. This parameter cannot be used with the `GROUP_MAX` flag.

GetGroupMember

Returns entity that is at a specified index position in the specified group.

Syntax

```
AI.GetGroupMember(entityId|groupId, idx, flags, type)
```

Returns the script handler of the requested entity, or null if the requested index value is out of range.

Parameter	Description
`entityId|groupId`	The entity ID or group ID.
`idx`	Position in the index from 1 to `n`.
`flags`	A combination of one or more of the following flags: • `GROUP_ALL` – Counts all agents in the group (default). • `GROUP_ENABLED` – Counts enabled agents only (exclusive with all).
`type`	The AI object type for which to filter. Returns only the AI objects of the type specified. This parameter cannot be used with the `GROUP_MAX` flag..

GetGroupOf

Retrieves the group ID of a specified entity ID.

Syntax

```
AI.GetGroupOf(entityId)
```

Returns the group ID of the specified entity.

Parameter	Description
`entityId`	The ID of the entity whose group ID to return.

GetGroupScopeUserCount

Syntax

```
AI.GetGroupScopeUserCount(ScriptHandle entityIdHandle, const char* groupScopeName)
```

Returns the number of actors inside the group scope if greater than or equal to zero, or nil if an error occurs.

Parameter	Description
`entityId`	The entity ID of the agent for whom you want to access the group scope.
`groupScopeName.`	The group scope name.

GetGroupScriptTable

Syntax

```
AI.GetGroupScriptTable(int groupID)
```

GetGroupTarget

Retrieves the most threatening attention target among the AI agents in a specified entity's group. See `IAgent.h` for a definition of alert status.

Syntax

```
AI.GetGroupTarget(entityId [,bHostileOnly [,bLiveOnly]])
```

Parameter	Description
`entityId`	Unique entity ID used to determine the group.
`bHostileOnly (optional)`	Flag indicating whether or not to include only hostile targets in group.
`bLiveOnly (optional)`	Flag indicating whether or not to include only live targets in group.

GetGroupTargetCount

Retrieves the number of attention targets among the AI agents in a specified entity's group.

Syntax

```
AI.GetGroupTargetCount(entityId [,bHostileOnly [,bLiveOnly]])
```

Parameter	Description
`entityId`	Unique entity ID used to determine the group.
`bHostileOnly (optional)`	Flag indicating whether or not to include only hostile targets in group.
`bLiveOnly (optional)`	Flag indicating whether or not to include only live targets in group.

GetGroupTargetEntity

Syntax

```
AI.GetGroupTargetEntity(int groupID)
```

GetGroupTargetThreat

Syntax

```
AI.GetGroupTargetThreat(int groupID)
```

GetGroupTargetType

Syntax

```
AI.GetGroupTargetType(int groupID)
```

GetLastUsedSmartObject

Retrieves the smart object last used by the user specified.

Syntax

```
AI.GetLastUsedSmartObject(userEntityId)
```

Returns nil if there is no last used smart object or if an error has occurred; otherwise, returns the script table of the entity that was the smart object last used by the user specified.

Parameter	Description
`userEntityId`	The entity ID of the user to query for the last used smart object.

GetLeader

Gets the name of a specified group leader.

Syntax

```
AI.GetLeader(groupID | entityID)
```

Returns the leader name.

Parameter	Description
`groupID`	Unique group ID.
`entityID`	The ID of the entity.

GetMemoryFireType

Syntax

```
AI.GetMemoryFireType(entityId)
```

Returns the method that the puppet uses for firing at its memory target.

Parameter	Description
`entityId`	The ID of the entity.

GetNavigationType

Retrieves the navigation type at a specified entity's position.

Syntax

```
AI.GetNavigationType(entityId)
```

Returns Navigation type, such as `NAV_TRIANGULAR`, `NAV_WAYPOINT_HUMAN`, `NAV_ROAD`, `NAV_VOLUME`, `NAV_WAYPOINT_3DSURFACE`, `NAV_FLIGHT`, `NAV_SMARTOBJECT`. See the `IAISystem::ENavigationType` definition for a complete list.

Parameter	Description
`entityId`	The ID of the entity.

GetNearestEntitiesOfType

Syntax

```
AI.GetNearestEntitiesOfType(entityId|objectname|position, AIObjectType, maxObjects,
 returnList [,objectFilter [,radius]])
```

Returns the number of found entities.

Parameter	Description
`entityId | objectname |position.`	Unique entity ID, AI object name, or position used to pinpoint the center position of the search.
`radius`	Radius of the search area.
`AIObjectType`	AIObject type to search for (see `ScriptBindAI.cpp` and `Scripts/AIAnchor.lua` for a complete list of AIObject types).
`maxObjects`	Maximum number of objects to find.
`return list`	Lua table to hold the list of found entities (Lua handlers).
`(optional) objectFilter`	A combination of one or more of the following search filter flags: • `AIOBJECTFILTER_SAMEFACTION` – Include only AI objects of the same species as the querying object. • `AIOBJECTFILTER_SAMEGROUP` – Include only AI objects of the same group as the querying object (or with no group). • `AIOBJECTFILTER_NOGROUP` – Include only AI objects with a Group ID of AI_NOGROUP. • `AIOBJECTFILTER_INCLUDEINACTIVE` – Include objects that are inactive.

GetNearestHidespot

Retrieves a specified entity's nearest hidepoint within a specified range.

Syntax

```
AI.GetNearestHidespot(entityId, rangeMin, rangeMax [, center])
```

Returns Point position, if found.

Parameter	Description
`entityId`	The ID of the entity.
`rangeMin`	Minimum range of search area.
`rangeMax`	Maximum range of search area
`centre (optional)`	Center point of the search area. If not specified, the entity's current position is used.

GetNearestPathOfTypeInRange

Retrieves the type of path nearest to a specified point of interest for a specified entity. Paths use the same types as anchors and are specified in the path properties. The function only returns paths that match the entity's navigation caps. Navigation type is also specified in the path properties.

Syntax

```
AI.GetNearestPathOfTypeInRange(entityId, pos, range, type [, devalue, useStartNode])
```

Parameter	Description
entityId	The ID of the entity.
pos	Vector specifying to the point of interest.
range	Search range.
type	Type of path to search for.
devalue (optional)	Time the returned path is marked as occupied.
useStartNode (optional)	Flag indicating whether or not to look a path with any point inside the range (useStartNode=0) or one with a start node inside the range (useStartNode=0).

GetNearestPointOnPath

Locates the point on a path nearest to a specified position.

Syntax

```
AI.GetNearestPointOnPath(entityId, pathname, vPos)
```

Parameter	Description
`entityId`	The ID of the entity.
`pathname`	Name of path.
`vPos`	Position to measure from.

GetObjectBlackBoard

Retrieves a specified object's black board (a Lua table).

Syntax

```
AI.GetObjectBlackBoard(entity)
```

Returns black board – if there is one; otherwise, nil.

Parameter	Description
`entityId or entityName.`	An AI entity identifier.

GetObjectRadius

Retrieves the radius of a specified AI object.

Syntax

```
AI.GetObjectRadius(entityId)
```

Returns the radius size.

Parameter	Description
`entityId`	The ID of the entity.

GetParameter

Retrieves the value of an enumerated AI parameter for a specified entity.

Syntax

```
AI.GetParameter(entityId, paramEnum)
```

Returns the value of the parameter.

Parameter	Description
`entityId`	The ID of the entity.
`paramEnum`	The index of the parameter to get. See `AI.ChangeParameter()` for a complete list.

GetPathLoop

Syntax

```
AI.GetPathLoop(entityId, pathname)
```

Returns true if path is successfully looped.

Parameter	Description
`entityId`	The ID of the entity.
`pathname`	Name of the path.

GetPathSegNoOnPath

Syntax

```
AI.GetPathSegNoOnPath(entityId, pathname, vPos)
```

Returns Segment ratio (0.0 start point, 100.0 end point).

Parameter	Description
`entityId`	The ID of the entity.
`pathname`	Name of path.
`vPos`	Position.

GetPeakThreatLevel

Syntax

```
AI.GetPeakThreatLevel(ScriptHandle entityID)
```

GetPeakThreatType

Syntax

```
AI.GetPeakThreatType(ScriptHandle entityID)
```

GetPointOnPathBySegNo

Syntax

```
AI.GetPointOnPathBySegNo(entityId, pathname, segNo)
```

Returns Point by segment ratio (0.0 start point, 100.0 end point).

Parameter	Description
`entityId`	The ID of the entity.
`pathname`	Name of path.
`segNo`	Segment ratio.

GetPosturePriority

Sets the specified entity's posture priority.

Syntax

```
AI.GetPosturePriority(ScriptHandle entityId, const char* postureName)
```

GetPotentialTargetCount

Retrieves the total number of a specified entity's potential targets.

Syntax

```
AI.GetPotentialTargetCount(ScriptHandle entityID)
```

Parameter	Description
`entityId`	The ID of the entity.

GetPotentialTargetCountFromFaction

Retrieves the number of an entity's potential targets that belong to a specified faction.

Syntax

```
AI.GetPotentialTargetCountFromFaction(ScriptHandle entityID, const char* factionName)
```

Parameter	Description
`entityId`	The ID of the entity.
`name`	Faction name.

GetPredictedPosAlongPath

Retrieves the predicted position of an AI agent along its path at a specified time.

Syntax

```
AI.GetPredictedPosAlongPath(entityId, time, retPos)
```

Returns True if successful.

Parameter	Description
`entityId`	The ID of the entity.
`time`	Time (in seconds) to predict position.
`retPos`	Return point value of the predicted position .

GetPreviousBehaviorName

Syntax

```
AI.GetPreviousBehaviorName(ScriptHandle entityID)
```

GetPreviousPeakThreatLevel

Syntax

```
AI.GetPreviousPeakThreatLevel(ScriptHandle entityID)
```

GetPreviousPeakThreatType

Syntax

```
AI.GetPreviousPeakThreatType(ScriptHandle entityID)
```

GetProbableTargetPosition

Retrieves the probable target position of a specified entity.

Syntax

```
AI.GetProbableTargetPosition(entityId)
```

Parameter	Description
`entityId`	The ID of the entity.

GetRefPointDirection

Retrieves a specified entity's reference point direction.

Syntax

```
AI.GetRefPointDirection(entityId)
```

Returns a script vector (x,y,z) reference point direction.

Parameter	Description
`entityId`	The ID of the entity.

GetRefPointPosition

Retrieves a specified entity's reference point "world" position.

Syntax

```
AI.GetRefPointPosition(entityId)
```

Returns a script vector (x,y,z) reference point position.

Parameter	Description
`entityId`	The ID of the entity.

GetRefShapeName

Retrieves the name of a specified entity's reference shape.

Syntax

```
AI.GetRefShapeName(entityId)
```

Returns a reference shape name.

Parameter	Description
`entityId`	The ID of the entity.

GetSoundPerceptionDescriptor

Retrieves information about how the specified entity perceives sound types.

Syntax

```
AI.GetSoundPerceptionDescriptor(entityId, soundType, descriptorTable)
```

Returns true if the information is successfully returned.

Parameter	Description
`entityId`	Entity to get perception data on.
`soundType`	Type of sound stimulus to get data for.
`descriptorTable`	Location to store retrieved data.

GetStance

Retrieves the specified entity's stance.

Syntax

```
AI.GetStance(entityId)
```

Returns entity stance (`STANCE_*`)

Parameter	Description
`entityId`	The ID of the entity.

GetSubTypeOf

Retrieves a specified entity's sub type.

Syntax

```
AI.GetSubTypeOf(entityId)
```

Returns the entity sub type (as defined in `IAgent.h`).

Parameter	Description
`entityId`	The ID of the entity.

GetTacticalPoints

Retrieves a point matching a description, related to a specified entity. Format of a point is: { x,y,z }.

Syntax

```
AI.GetTacticalPoints(entityId, tacPointSpec, point)
```

Returns true if a valid point is found; otherwise, false.

Parameter	Description
`entityId`	AI's entity.
`tacPointSpec.`	A table specifying the points required.
`point`	Coordinates of the point found.

GetTargetSubType

Retrieves the subtype of a specified entity's current attention target.

Syntax

```
AI.GetTargetSubType(entityId)
```

Returns an attention target subtype. See `IAgent.h` for a list of target type definitions.

Parameter	Description
`entityId`	The ID of the entity.

GetTargetType

Retrieves the type of a specified entity's current attention target.

Syntax

```
AI.GetTargetType(entityId)
```

Returns an attention target type, such as `AITARGET_NONE`, `AITARGET_MEMORY`, `AITARGET_BEACON`, `AITARGET_ENEMY`. See `IAgent.h` for a list of target type definitions.

Parameter	Description
`entityId`	The ID of the entity.

GetTotalLengthOfPath

Retrieves total length of the specified path.

Syntax

```
AI.GetTotalLengthOfPath(entityId, pathname)
```

Parameter	Description
`entityId`	The ID of the entity.
`pathname`	Name of path.

GetTypeOf

Retrieves a specified entity's type.

Syntax

```
AI.GetTypeOf(entityId)
```

Returns the entity type (as defined in `IAgent.h`).

Parameter	Description
`entityId`	The ID of the entity.

GetUnitCount

Retrieves the number of units the leader knows about. The leader is identified based on the group ID of the entity.

Syntax

```
AI.GetUnitCount(entityId, unitProperties)
```

Parameter	Description
`entityId`	The ID of the entity.
`unitProperties`	Binary mask of unit properties type for which the attack is requested, in the following form: `UPR_* + UPR* (UPR_COMBAT_GROUND + UPR_COMBAT_FLIGHT)` See `IAgent.h` for a definition of unit properties UPR_*.

GetUnitInRank

Retrieves the entity that holds the specified rank position in the specified group.

Syntax

```
AI.GetUnitInRank(groupID [,rank])
```

Returns entity script table of the ranked unit.

Parameter	Description
`groupID`	The ID of the group that contains the entity to retrieve.
`rank`	The rank position of the entity to retrieve. If null or a value less than or equal to zero is specified, retrieves the entity with the highest rank in the entity. The value of the highest rank is 1.

GoTo

Allows the specified entity to move to a certain destination.

Syntax

```
AI.GoTo(entityId, vDestination)
```

Parameter	Description
`entityId`	AI's entity.
`vDestination.`	.

Hostile

Determines whether or not two entities are hostile.

Syntax

```
AI.Hostile(entityId, entity2Id|AIObjectName)
```

Returns true if the entities are hostile.

Parameter	Description
`entityId`	ID of the first AI entity.
`entity2Id | AIObjectName`	ID of the second AI entity, or AIobject name.

IgnoreCurrentHideObject

Marks the current hide object as unreachable; it will be omitted from future hidespot selections.

Syntax

```
AI.IgnoreCurrentHideObject(entityId)
```

Parameter	Description
`entityId`	The ID of the entity.

IntersectsForbidden

Determines whether or not the specified line is in a forbidden region.

Syntax

```
AI.IntersectsForbidden(Vec3 start, Vec3 end)
```

Returns intersected position or end (if there is no intersection).

Parameter	Description
`start`	Vector in format {x,y,z}.
`end`	Vector in format {x,y,z}.

IsAgentInAgentFOV

Determines whether or not one entity is in the field of view of another entity.

Syntax

```
AI.IsAgentInAgentFOV(entityId, entityId2)
```

Returns the first value true if the agent is within the entity FOV; the second value true if the agent is within the entity's primary FOV, or false if the agent is within the entity's secondary FOV.

Parameter	Description
`entityId`	The AI entity whose FOV to check.
`entityId2.`	The entity ID of the agent.

IsAgentInTargetFOV

Determines whether or not the entity is in the FOV of the attention target.

Syntax

```
AI.IsAgentInTargetFOV(entityId, fov)
```

Returns true if in the FOV of the attention target; otherwise, false.

Parameter	Description
`entityId`	The ID of the entity.
`fov`	FOV of the enemy in degrees.

IsAimReady

Syntax

```
AI.IsAimReady(ScriptHandle entityIdHandle)
```

IsCoverCompromised

Syntax

```
AI.IsCoverCompromised(entityId)
```

Returns true if the cover has been compromised; otherwise, nil.

Parameter	Description
`entityId`	AI's entity.

IsEnabled

Checks that the entity is AI-enabled.

Syntax

```
AI.IsEnabled(entityId)
```

Parameter	Description
`entityId`	The ID of the entity.

IsFireEnabled

Determines whether or not the AI is allowed to fire or not.

Syntax

```
AI.IsFireEnabled(entityId)
```

Returns true if AI is enabled to fire

Parameter	Description
`entityId`	The ID of the entity.

IsInCover

Determines whether or not the agent is using cover.

Syntax

```
AI.IsInCover(entityId)
```

IsLowHealthPauseActive

Syntax

```
AI.IsLowHealthPauseActive(ScriptHandle entityID)
```

IsLowOnAmmo

Syntax

```
AI.IsLowOnAmmo(entityId)
```

Parameter	Description
`entityId`	AI's entity.
`threshold`	The ammo percentage threshold.

IsMoving

Determines whether or not the agent wants to move.

Syntax

```
AI.IsMoving(entityId)
```

Parameter	Description
`entityId`	The ID of the entity.

IsMovingInCover

Syntax

```
AI.IsMovingInCover(entityId)
```

Returns true – Agent is moving in cover nil – if not

Parameter	Description
entityId	AI's entity.

IsMovingToCover

Determines whether or not the agent is running to cover.

Syntax

```
AI.IsMovingToCover(entityId)
```

Parameter	Description
entityId	AI's entity.

IsOutOfAmmo

Syntax

```
AI.IsOutOfAmmo(entityId)
```

Returns true if the specified entity is out of ammunition; otherwise, nil.

Parameter	Description
entityId	The ID of the AI entity.

IsPersonallyHostile

Syntax

```
AI.IsPersonallyHostile(ScriptHandle entityID, ScriptHandle hostileID)
```

IsPointInFlightRegion

Determines whether or not a specified point is in the flight region.

Syntax

```
AI.IsPointInFlightRegion(point)
```

Returns true if the point is in the flight region.

Parameter	Description
`point`	Vector in format {x,y,z}.

IsPointInsideGenericShape

Determines whether or not a point is inside a specified shape.

Syntax

```
AI.IsPointInsideGenericShape(position, shapeName[, checkHeight])
```

Parameter	Description
`position`	Position to check.
`shapeName`	Name of the shape to test (returned by `AI.GetEnclosingGenericShapeOfType`).
`checkHeight (optional)`	Flag indicating whether or not to test for shape height. (default=false). If set to true, the test will check the space between shape.aabb.min.z and shape.aabb.min.z+shape.height.

IsPointInWaterRegion

Determines whether or not the point is in the water region.

Syntax

```
AI.IsPointInWaterRegion(point)
```

Returns a value that indicates water or ground level. Values greater than 0 mean there is water.

IsPunchableObjectValid

Determines whether or not a punchable object is valid.

Syntax

```
AI.IsPunchableObjectValid(userId, objectId, origPos)
```

Parameter	Description
`userId.`	User ID.
`objectId.`	Object ID.
`origPos.`	Object position in the world.

IsTakingCover

Syntax

```
AI.IsTakingCover(entityId, [distanceThreshold])
```

Returns true if the specfied agent is either in cover or running to cover; otherwise, nil.

Parameter	Description
`entityId`	AI's entity.
`distanceThreshold.`	(Optional) Distance over which an agent that is running to cover is considered to not yet have taken cover.

LoadBehaviors

Syntax

```
AI.LoadBehaviors(const char* folderName, const char* extension)
```

LogComment

Writes additional information to the log for debugging purposes.

Syntax

```
AI.LogComment(szMessage)
```

Parameter	Description
`szMessage`	The message to write to the log.

LogEvent

Writes event-driven information to the log that for debugging purposes. Events may occur on a per-frame or a per AI update basis.

Syntax

```
AI.LogEvent(szMessage)
```

Parameter	Description
`szMessage`	The message to write to the log.

LogProgress

Writes progress messages to the log.

Syntax

```
AI.LogProgress(szMessage)
```

Parameter	Description
szMessage	The message to write to the log.

MeleePunchableObject

Syntax

```
AI.MeleePunchableObject(entityId, objectId, origPos)
```

Parameter	Description
entityId	The ID of the AI entity.
objectId	Object ID.
origPos	Position of the melee punchable object.

ModifySmartObjectStates

Adds or removes smart object states for a specified entity.

Syntax

```
AI.ModifySmartObjectStates(entityId, listStates)
```

Parameter	Description
entityId	The ID of the entity.
listStates	The list of state names to be added or removed (such as "Closed, Locked", "Open, Unlocked, Busy").

ParseTables

Syntax

```
AI.ParseTables(int firstTable, bool parseMovementAbility, IFunctionHandler* pH,
 AIObjectParams& aiParams, bool& updateAlways)
```

Parameter	Description
firstTable	Properties table.
parseMovementAbility	True to parse movement ability, false otherwise.
aiParams	AI parameters.
updateAlways	True to always update; false otherwise.

PlayCommunication

Plays communication on the AI agent.

Syntax

```
AI.PlayCommunication(ScriptHandle entityId, const char* commName, const char* channelName,
 float contextExpiry)
```

Parameter	Description
`entityId`	The ID of the entity.
`commName`	The name of the communication to play.
`channelName`	The name of the channel where the communication will play.

PlayReadabilitySound

Plays readability sound on the AI agent. This call does not do any filtering like playing readability using signals.

Syntax

```
AI.PlayReadabilitySound(entityId, soundName)
```

Parameter	Description
`entityId`	The ID of the entity.
`soundName`	The name of the readability sound signal to play.
`stopPreviousSounds (Optional)`	True if any currently playing readability should be stopped in favor of this one.
`responseDelayMin (Optional)`	Minimum (or exact, if no maximum) delay for the response readability to play.
`repsonseDelayMax (Optional)`	Maximum delay for the response readability to play.

ProcessBalancedDamage

Processes balanced damage.

Syntax

```
AI.ProcessBalancedDamage(pShooterEntity, pTargetEntity, damage, damageType)
```

Parameter	Description
`pShooterEntity`	Shooter ID.

Parameter	Description
pTargetEntity	Target ID.
damage	Hit damage.
damageType	Hit damage type.

QueueBubbleMessage

Syntax

```
AI.QueueBubbleMessage(ScriptHandle entityID, const char* message)
```

RecComment

Records a comment with AI Debug Recorder. For information about the AI Debug Recorder, see Using the AI Debug Recorder.

Syntax

```
AI.RecComment(szMessage)
```

Parameter	Description
szMessage	Message line to be displayed in Recorder view.

RegisterDamageRegion

Registers a spherical region that causes damage (so should be avoided in pathfinding). The owner entity position is used as the region center. The function can be called multiple times to update the region position.

Syntax

```
AI.RegisterDamageRegion(entityId, radius)
```

Parameter	Description
entityId	The ID of the entity.
radius	The radius of the spherical region. If less than or equal to zero, the region is disabled.

RegisterInterestedActor

Registers the interested actor with the interest system. Any errors go to the error log.

Syntax

```
AI.RegisterInterestedActor(ScriptHandle entityId, float fInterestFilter, float
 fAngleInDegrees)
```

Returns true if a valid update was performed; otherwise, nil. Nil can be returned if the interest system is disabled or the parameters are not valid.

Parameter	Description
`entityId`	The ID of the AI entity.

RegisterInterestingEntity

Registers the specified entity with the interest system. Any errors go to the error log.

Syntax

```
AI.RegisterInterestingEntity(ScriptHandle entityId, float radius, float baseInterest, const
 char* actionName, Vec3 offset, float pause, int shared)
```

Returns true if a valid update was performed; otherwise, nil. Nil can be returned if the interest system is disabled or the parameters are not valid.

Parameter	Description
`entityId`	The ID of the entity.

RegisterTacticalPointQuery

Retrieves a query ID for the specified tactical point query.

Syntax

```
AI.RegisterTacticalPointQuery(querySpecTable)
```

Returns > 0 – If the query was parsed successfully 0 – Otherwise

Parameter	Description
`querySpecTable`	Table specifying the query. For more information, see AI Tactical Point System (p. 20).

RegisterTargetTrack

Registers an AI entity to use a specified target track configuration for target selection. The parameter `ai_TargetTracking` must be set to '2'.

Syntax

```
AI.RegisterTargetTrack(entityId, configuration, targetLimit, classThreat)
```

Returns true if successfully registered.

Parameter	Description
`entityId`	The ID of the entity.

Parameter	Description
`configuration`	Target track configuration.
`targetLimit`	The number of agents who can target the AI at any specified time (0 for infinite).
`classThreat (optional)`	Initial class threat value.

RemovePersonallyHostile

Syntax

```
AI.RemovePersonallyHostile(ScriptHandle entityID, ScriptHandle hostileID)
```

RequestAttack

In a group with a leader, allows the leader to issue a request for a group attack behavior against the enemy. After this request, the `Cleader` may create an attack leader action (`CLeaderAction_Attack_*`).

Syntax

```
AI.RequestAttack(entityId, unitProperties, attackTypeList [,duration])
```

Parameter	Description
`entityId`	Unique entity ID used to determine the group leader.
`unitProperties`	Binary mask of unit properties type for which the attack is requested, in the following form: `UPR_* + UPR* (UPR_COMBAT_GROUND + UPR_COMBAT_FLIGHT)` See `IAgent.h` for a definition of unit properties `UPR_*`.
`attackTypeList`	Lua table containing a prioritized list of preferred attack strategies (leader action subtypes). The list must be in the following format: `{LAS_*, LAS_*,..} (LAS_ATTACK_ROW,LAS_ATTACK_FLANK)` which means that the first attempt will be an Attack_row action , and if that fails an attack_flank. See `IAgent.h` for a definition of LeaderActionSubtype (`LAS_*`) action types.
`duration (optional)`	Maximum duration in seconds (default = 0).

RequestToStopMovement

Syntax

```
AI.RequestToStopMovement(ScriptHandle entityId)
```

ResetAgentLookAtPos

Resets the specified entity's previous call to `AgentLookAtPos()`.

Syntax

```
AI.ResetAgentLookAtPos(entityId)
```

Parameter	Description
`entityId`	The ID of the entity.

ResetAgentState

Resets a particular aspect of the agent's state, such as "lean".

Syntax

```
AI.ResetAgentState(ScriptHandle entityId, const char * stateLabel)
```

Returns nil

Parameter	Description
`entityId`	The ID of the AI entity.
`stateLabel`	String describing the state that must be reset to default.

ResetParameters

Resets all parameters for a specified entity.

Syntax

```
AI.ResetParameters(entityId, bProcessMovement, PropertiesTable, PropertiesInstanceTable)
```

Parameter	Description
`entityId`	The ID of the entity whose parameters you want to reset.
`bProcessMovement`	True to reset movement data; otherwise, false.
`PropertiesTable`	The Lua table that contains the entity's properties.
`PropertiesInstanceTable`	The Lua table that contains instance-specific entity properties.

ResetPersonallyHostiles

Syntax

```
AI.ResetPersonallyHostiles(ScriptHandle entityID, ScriptHandle hostileID)
```

ScaleFormation

Changes the scale factor of a specified entity's formation (if one exists).

Syntax

```
AI.ScaleFormation(entityId, scale)
```

Returns true if formation scaling was successful.

Parameter	Description
`entityId`	The ID of the entity.
`scale`	Scale factor.

SequenceBehaviorReady

Syntax

```
AI.SequenceBehaviorReady(ScriptHandle entityId)
```

SequenceInterruptibleBehaviorLeft

Syntax

```
AI.SequenceInterruptibleBehaviorLeft(ScriptHandle entityId)
```

SequenceNonInterruptibleBehaviorLeft

Syntax

```
AI.SequenceNonInterruptibleBehaviorLeft(ScriptHandle entityId)
```

SetAlarmed

Sets the entity to be "perception alarmed".

Syntax

```
AI.SetAlarmed(entityId)
```

SetAnimationTag

Sets a mannequin animation tag.

Syntax

```
AI.SetAnimationTag(ScriptHandle entityID, const char* tagName)
```

Returns a default result code (in Lua: void).

Parameter	Description
`entityId`	The ID of the AI entity on which to set the animation tag.

Parameter	Description
`tagName`	The name of the animation tag that should be set (case insensitive).

SetAssesmentMultiplier

Sets the assesment multiplier factor for the specified AIObject type.

Syntax

```
AI.SetAssesmentMultiplier(AIObjectType, multiplier)
```

Parameter	Description
`AIObjectType`	Type of AIObject. See `ScriptBindAI.cpp` for a complete list of AIObject types.
`multiplier`	Assesment multiplier factor.

SetAttentiontarget

Sets a new attention target.

Syntax

```
AI.SetAttentiontarget(entityId, targetId)
```

Parameter	Description
`entityId`	The ID of the entity.
`targetId`	Target's entity ID.

SetBeaconPosition

Sets the beacon's position for the specified entity/object's group.

Syntax

```
AI.SetBeaconPosition(entityId | AIObjectName, pos)
```

Parameter	Description
`entityId | AIObjectName`	Unique entity ID or AI object name.
`pos`	Vector {x,y,z} where the beacon position will be set.

SetBehaviorTreeEvaluationEnabled

Syntax

```
AI.SetBehaviorTreeEvaluationEnabled(ScriptHandle entityID, bool enable)
```

SetBehaviorVariable

Sets a behaviour variable for the specified actor.

Syntax

```
AI.SetBehaviorVariable(ScriptHandle entityId, const char* variableName, bool value)
```

SetCollisionAvoidanceRadiusIncrement

Syntax

```
AI.SetCollisionAvoidanceRadiusIncrement(ScriptHandle entityId, float radius)
```

SetContinuousMotion

Syntax

```
AI.SetContinuousMotion(ScriptHandle entityID, bool continuousMotion)
```

SetCoverCompromised

Syntax

```
AI.SetCoverCompromised(entityId)
```

Parameter	Description
`entityId`	The ID of the AI entity.

SetEntitySpeedRange

Syntax

```
AI.SetEntitySpeedRange(userEntityId, urgency, defaultSpeed, minSpeed, maxSpeed, stance =
 all)
```

Returns true if the operation was successful and false otherwise

Parameter	Description
`usedEntityId`	The entity ID of the user for which its last used smart object is needed.
`urgency`	The integer value specifying the movement urgency (see `AgentMovementSpeeds::EAgentMovementUrgency`).
`defaultSpeed`	Floating point value that specfies the default speed.
`minSpeed`	Floating point value that specifies the minimum speed.

SetExtraPriority

Sets a extra priority value to the specified enemy entity.

Syntax

```
AI.SetExtraPriority(enemyEntityId, increment)
```

Parameter	Description
`enemyEntityId`	The ID of the entity.
`float increment`	Value to add to the target's priority.

SetFactionOf

Sets the faction to which the specified entity belongs.

Syntax

```
AI.SetFactionOf(ScriptHandle entityID, const char* factionName)
```

Parameter	Description
`entityId`	The ID of the entity whose faction to return.
`factionName`	The name of the faction to assign to the specified entity.

SetFactionThreatMultiplier

Sets the threat multiplier factor for the specified species. A return value of 0 indicates that the species is not hostile to any other species.

Syntax

```
AI.SetFactionThreatMultiplier(nSpecies, multiplier)
```

SetFireMode

Sets fire mode immediately.

Syntax

```
AI.SetFireMode(entityId, mode)
```

Parameter	Description
`entityId`	The ID of the entity.
`firemode`	New fire mode.

SetFormationAngleThreshold

Sets the relative position inside the formation.

Syntax

```
AI.SetFormationAngleThreshold(entityId, fAngleThreshold)
```

Parameter	Description
`entityId`	The ID of the AI entity.
`fAngleThreshold`	New leader orientation angle threshold in degrees.

SetFormationLookingPoint

Sets the relative looking point position inside the formation.

Syntax

```
AI.SetFormationLookingPoint(entityId, v3RelativePosition)
```

Parameter	Description
`entityId`	The ID of the AI entity.
`v3RelativePosition`	Table with format {x,y,z} storing the new relative looking point.

SetFormationPosition

Sets the relative position inside the formation.

Syntax

```
AI.SetFormationPosition(entityId, v2RelativePosition)
```

Parameter	Description
`entityId`	The ID of the AI entity.
`v2RelativePosition`	Table with format {x,y} storing the new relative position.

SetFormationUpdate

Sets the update flag for a specified entity's formation (if one exists). If this flag is false, the formation is no longer updated.

Syntax

```
AI.SetFormationUpdate(entityId, update)
```

Returns true if the request was successful.

Parameter	Description
`entityId`	The ID of the entity.
`update`	True to update the flag; otherwise, false.

SetFormationUpdateSight

Sets a random angle rotation for a specified entity's formation sight directions.

Syntax

```
AI.SetFormationUpdateSight(entityId, range, minTime, maxTime)
```

Parameter	Description
`entityId`	The ID of the entity.
`range`	Angle of rotation (0,360) around the default sight direction.
`minTime (optional)`	Minimum timespan for changing the direction (default = 2).
`maxTime (optional)`	Minimum timespan for changing the direction (default = minTime).

SetIgnorant

Sets the specified AI entity to ignore system signals, visual stimuli and sound stimuli.

Syntax

```
AI.SetIgnorant(entityId, ignorant)
```

Parameter	Description
`entityId`	The ID of the AI entity.
`ignorant`	A flag indicating whether or not the entity ignores system signals. 0 specifies do not ignore; 1 specifies ignore.

SetInCover

Syntax

```
AI.SetInCover(entityId, bool inCover)
```

Parameter	Description
`entityId`	The ID of the AI entity.
`inCover`	Specifies whether the entity should be set to be in cover or not.

SetLeader

Sets a specified entity as the group leader. This action associates a `CLeader` object with the entity, creating it if one doesn't exist. Only one leader can be set per group.

Syntax

```
AI.SetLeader(entityID)
```

Returns true if successful.

Parameter	Description
`entityID`	Unique entity ID to set as leader.

SetMemoryFireType

Sets how the AI agent handles firing at its memory target.

Syntax

```
AI.SetMemoryFireType(entityId, type)
```

Parameter	Description
`entityId`	The ID of the entity.

`type`	Memory fire type. Possible values from `enum EMemoryFireType` in `IAgent.h`:
	`eMFT_Disabled = 0,  // Never allowed to fire at memory`
	`eMFT_UseCoverFireTime,  // Can fire at memory using the weapon's cover fire time`
	`eMFT_Always,  // Always allowed to fire at memory`

SetMovementContext

Sets the specified entity's movement context.

Syntax

```
AI.SetMovementContext(ScriptHandle entityId, int context)
```

Parameter	Description
`entityId`	The ID of the entity.
`context`	context value .

SetPathAttributeToFollow

Sets the attribute of a specified entity's path.

Syntax

```
AI.SetPathAttributeToFollow(entityId, flag)
```

Parameter	Description
`entityId`	The ID of the entity.
`flag`	Attribute to set.

SetPathToFollow

Sets the path for a specified entity to follow.

Syntax

```
AI.SetPathToFollow(entityId, pathName)
```

Parameter	Description
`entityId`	The ID of the entity.
`pathName`	Name of the path to be followed.

SetPFBlockerRadius

Syntax

```
AI.SetPFBlockerRadius(entityId, blocker, radius)
```

Parameter	Description
`entityId`	The ID of the entity.

SetPointListToFollow

Sets a point list for a specified entity's path.

Syntax

```
AI.SetPointListToFollow(entityId, pointlist, howmanypoints, bspline [, navtype])
```

Parameter	Description
`entityId`	The ID of the entity.
`pointList`	List of points for the entity to follow, expressed as a set of local vectors: `{{x=0.0, y=0.0, z=0.0},.`
`howmanypoints`	Number of points in the list.

Parameter	Description
bspline	Flag indicating whether or not the path line is recalculated using spline interpolation.
navtype	(Optional) Navigation type (default = IAISystem::NAV_FLIGHT).

SetPosturePriority

Sets the specified entity's posture priority.

Syntax

```
AI.SetPosturePriority(ScriptHandle entityId, const char* postureName, float priority)
```

SetPostures

Sets the specified entity's postures.

Syntax

```
AI.SetPostures(ScriptHandle entityId, SmartScriptTable postures)
```

Parameter	Description
entityId	The ID of the entity.
postures	The table of postures.

SetRefPointAtDefensePos

Sets a specified entity's reference point position to an intermediate distance between the entity's attention target and a specified point.

Syntax

```
AI.SetRefPointAtDefensePos(entityId, point2defend, distance)
```

Parameter	Description
entityId	The ID of the entity.
point2defend	Point to defend.
distance	Maximum distance between reference point and point to defend.

SetRefPointDirection

Sets a specified entity's reference point direction.

Syntax

```
AI.SetRefPointDirection(vRefPointDir)
```

Parameter	Description
vRefPointDir	Direction as a (script)vector (x,y,z) value.

SetRefPointPosition

Sets a specified entity's reference point "world" position.

Syntax

```
AI.SetRefPointPosition(entityId, vRefPointPos)
```

Parameter	Description
entityId	The ID of the entity.
vRefPointPos	World position as a (script)vector (x,y,z) value.

SetRefPointRadius

Sets a specified entity's reference point radius.

Syntax

```
AI.SetRefPointRadius(entityId, radius)
```

Parameter	Description
entityId	The ID of the entity.
radius	The reference point radius.

SetRefpointToAnchor

Sets a reference point to an anchor.

Syntax

```
AI.SetRefpointToAnchor(entityId, rangeMin, rangeMax, findType, findMethod)
```

Parameter	Description
entityId	The ID of the AI entity.
rangeMin	Minimum range.
rangeMax	Maximum range.
findType	Finding type.
findMethod	Finding method.

SetRefpointToPunchableObject

Sets the reference point to the punchable object.

Syntax

```
AI.SetRefpointToPunchableObject(entityId, range)
```

Parameter	Description
`entityId`	The ID of the AI entity.
`range`	Range for the punchable object.

SetRefShapeName

Sets the name of a specified entity's reference shape.

Syntax

```
AI.SetRefShapeName(entityId, name)
```

Parameter	Description
`entityId`	The ID of the entity.
`name`	Name of the reference shape.

SetSmartObjectState

Sets a single smart object state, replacing all other states.

Syntax

```
AI.SetSmartObjectState(entityId, stateName)
```

Parameter	Description
`entityId`	The ID of the entity.
`stateName`	The name of the new state to set for the smart object (such as "Idle").

SetSoundPerceptionDescriptor

Sets information about how the specified entity perceives sound types.

Syntax

```
AI.SetSoundPerceptionDescriptor(entityId, soundType, descriptorTable)
```

Returns True if information successfully saved.

Parameter	Description
`entityId`	Entity to set perception data for.
`soundType`	Type of sound stimulus to set data for.
`descriptorTable`	Perception data to saved.

SetSpeed

Sets the entity's current speed (urgency).

Syntax

```
AI.SetSpeed(entityId, urgency)
```

Parameter	Description
`entityId`	AI's entity.
`urgency`	A float value that specifies the movement urgency (see `AgentMovementSpeeds::EAgentMovementUrgency`).

SetStance

Sets the specified entity's stance.

Syntax

```
AI.SetStance(entityId, stance)
```

Parameter	Description
`entityId`	The ID of the entity.
`stance`	The stance value (`STANCE_*`).

SetTargetTrackClassThreat

Sets the class threat for a specified entity's target track.

Syntax

```
AI.SetTargetTrackClassThreat(entityId, classThreat)
```

Parameter	Description
`entityId`	The ID of the entity.

Parameter	Description
`classThreat`	New class threat value.

SetTempTargetPriority

Sets a specified entity's selection priority for a temporary target over other potential targets.

Syntax

```
AI.SetTempTargetPriority(entityId, priority)
```

Returns True if successfully updated.

Parameter	Description
`entityId`	The ID of the entity.
`priority`	New priority value.

SetTerritoryShapeName

Sets the territory shape of the specified AI entity.

Syntax

```
AI.SetTerritoryShapeName(entityId, shapeName)
```

Parameter	Description
`entityId`	The ID of the entity.
`shapeName`	Name of the shape to set.

SetUnitProperties

Sets the leader's knowledge about the unit's combat capabilities. The leader is identified based on the group ID of the entity.

Syntax

```
AI.SetUnitProperties(entityId, unitProperties)
```

Parameter	Description
`entityId`	The ID of the entity.
`unitProperties`	Binary mask of unit properties in the following form: `UPR_* + UPR* (UPR_COMBAT_GROUND + UPR_COMBAT_FLIGHT)` See `IAgent.h` for a definition of the `UPR_*` unit properties.

SetUseSecondaryVehicleWeapon

Enables or disables the AI object's ability to use the secondary weapon when firing from a vehicle gunner seat if possible.

Syntax

```
AI.SetUseSecondaryVehicleWeapon(entityId, bUseSecondary)
```

Parameter	Description
`entityId`	The ID of the entity.
`bUseSecondary`	Specify true to use the secondary weapon; otherwise, false.

Signal

Adds a signal to the sender's signal queue even if another signal with same text is present.

Syntax

```
AI.Signal(signalFilter, signalType, signalText, senderId [, signalExtraData])
```

Parameter	Description
`signalFilter`	The signal filter.
`signalType`	The signal type.
`signalText`	Signal text that is processed by the receivers, either in a Lua callback with the same name as the text or directly by the `CAIObject`.
`senderId`	The ID of the sender.
`signalExtraData`	Optional. A Lua table containing additional data. It can contain the following data types: • `point` – A vector in the format {x,y,z}. • `point2` – A vector in the format {x,y,z}. • `ObjectName` – A string. • `id` – An entity ID. • `fValue` – A float value. • `iValue` – An integer value. • `iValue2` – A second integer value.

SmartObjectEvent

Executes a smart action.

Syntax

```
AI.SmartObjectEvent(actionName, userEntityId, objectEntityId [, vRefPoint])
```

Returns 0 if a smart object rule was not found or if a non-zero ID was inserted to execute the action.

Parameter	Description
`actionName`	The name of the smart action.
`usedEntityId`	The entity ID of the user who wants to execute the smart action, or none if the user is unknown.
`objectEntityId`	The entity ID of the object on which the smart action is to be executed, or none if the object is unknown.
`vRefPoint`	Optional. The reference point to be used instead of the user's attention target position.

SoundEvent

Generates a sound event with the specified parameters in the AI system.

Syntax

```
AI.SoundEvent(position, radius, threat, interest, entityId)
```

Parameter	Description
`position`	Origin of the sound event.
`radius`	Area the sound event is heard in.
`threat`	Sound event property.
`interest`	Sound event property.
`entityId`	Unique entity ID that generates the sound event.

StopCommunication

Stops specified communication.

Syntax

```
AI.StopCommunication(ScriptHandle playID)
```

Parameter	Description
`playID`	The ID of the communication to stop.

ThrowGrenade

Throws a specified grenade at a target type without interrupting the fire mode.

Syntax

```
AI.ThrowGrenade(entityId, grenadeType, regTargetType)
```

Parameter	Description
`entityId`	The ID of the entity.
`grenadeType`	Requested grenade type (see `ERequestedGrenadeType`).
`regTargetType`	The grenade target type (`see AI_REG_*`).

UnregisterInterestedActor

Unregisters the entity with the interest system. Any errors are recorded in the error log.

Syntax

```
AI.UnregisterInterestedActor(ScriptHandle entityId)
```

Parameter	Description
`entityId`	The ID of the entity.

UnregisterInterestingEntity

Unregisters the specified entity with the interest system. Any errors are recorded in the error log.

Syntax

```
AI.UnregisterInterestingEntity(ScriptHandle entityId)
```

Parameter	Description
`entityId`	The ID of the entity.

UnregisterTargetTrack

Unregisters an AI object from the target track manager. The parameter `ai_TargetTracking` must be set to '2'.

Syntax

```
AI.UnregisterTargetTrack(entityId)
```

Returns true if successfully unregistered.

Parameter	Description
`entityId`	The ID of the entity.

UpdateGlobalPerceptionScale

Syntax

```
AI.UpdateGlobalPerceptionScale(float visualScale, float audioScale)
```

UpdateTempTarget

Updates the position of the specified entity's temporary potential target.

Syntax

```
AI.UpdateTempTarget(entityId, vPos)
```

Returns true if successfully updated.

Parameter	Description
`entityId`	The ID of the entity.
`vPos`	New position of the temporary target.

UpTargetPriority

Changes a specified entity's target priority value for a specified target, if the target is on the entity's target list.

Syntax

```
AI.UpTargetPriority(entityId, targetId, increment)
```

Parameter	Description
`entityId`	The ID of the entity.
`targetId`	The entity ID of the target.
`increment`	New value for the target priorty.

VisualEvent

Generates a visual event with the specified parameters in the AI system.

Syntax

```
AI.VisualEvent(entityId, targetId)
```

Parameter	Description
`entityId`	The ID of the entity that receives the visual event.
`targetId`	The ID of the visual target (that the entity is seeing).

Warning

Writes a warning message to the log about data or script errors.

Syntax

```
AI.Warning(szMessage)
```

Parameter	Description
`szMessage`	The message to write to the log.

ScriptBind_Entity

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists the entity related Lua script bind functions.

Activate

Activates or deactivates the entity. `Activate` ignores the update policy and forces an entity to activate or deactivate. All active entities are updated every frame.

> **Warning**
> Having too many active entities can affect performance.

Syntax

```
Entity.Activate(int bActive)
```

Parameters

Parameter	Description
`bActive`	Specify true to make the entity active; false to make it inactive.

ActivateOutput

Syntax

```
Entity.ActivateOutput()
```

ActivatePlayerPhysics

Syntax

```
Entity.ActivatePlayerPhysics(bool bEnable)
```

AddConstraint

Syntax

```
Entity.AddConstraint()
```

AddImpulse

Apply an impulse to the entity. At least four parameters need to be provided for a linear impulse. For an additional angular impulse, at least seven parameters need to be provided.

Syntax

```
Entity.AddImpulse(ipart, position, direction, linearImpulse, linearImpulseScale,
 angularAxis, angularImpulse, massScale)
```

Parameters

Parameter	Description
`ipart`	The index of the part that receives the impulse.
`position`	The point (in world coordinates) where the impulse is applied. Set this to (0, 0, 0) to ignore it.
`direction`	The direction in which the impulse is applied.
`linearImpulse`	The force of the linear impulse.
`linearImpulseScale`	Scaling of the linear impulse. (Default: 1.0)
`angularAxis`	The axis on which the angular impulse is applied.
`angularImpulse`	The force of the angular impulse.
`massScale`	Mass scaling of the angular impulse. (Default: 1.0)

AttachChild

Syntax

```
Entity.AttachChild(ScriptHandle childEntityId, int flags)
```

AttachSurfaceEffect

Syntax

```
Entity.AttachSurfaceEffect(int nSlot, const char *effect, bool countPerUnit, const char
 *form, const char *typ, float countScale, float sizeScale)
```

AuxAudioProxiesMoveWithEntity

Set whether AuxAudioProxies should move with the entity or not.

Syntax

```
Entity.AuxAudioProxiesMoveWithEntity(bool const bCanMoveWithEntity)
```

Returns: nil

Parameters

Parameter	Description
`bCanMoveWithEntity`	Boolean parameter to enable or disable

AwakeCharacterPhysics

Syntax

```
Entity.AwakeCharacterPhysics(int nSlot,const char *sRootBoneName,int nAwake)
```

AwakeEnvironment

Syntax

```
Entity.AwakeEnvironment()
```

AwakePhysics

Syntax

```
Entity.AwakePhysics(int nAwake)
```

BreakToPieces

Breaks static geometry in slot 0 into sub objects and spawn them as particles or entities.

Syntax

```
Entity.BreakToPieces(int nSlot, int nPiecesSlot, float fExplodeImp, Vec3 vHitPt, Vec3
 vHitImp, float fLifeTime, bool bSurfaceEffects)
```

CalcWorldAnglesFromRelativeDir

Syntax

```
Entity.CalcWorldAnglesFromRelativeDir(Vec3 dir)
```

CancelSubpipe

Syntax

```
Entity.CancelSubpipe()
```

ChangeAttachmentMaterial

Syntax

```
Entity.ChangeAttachmentMaterial(const char *attachmentName, const char *materialName)
```

CharacterUpdateAlways

Syntax

```
Entity.CharacterUpdateAlways(int characterSlot, bool updateAlways)
```

CharacterUpdateOnRender

Syntax

```
Entity.CharacterUpdateOnRender(int characterSlot, bool bUpdateOnRender)
```

CheckCollisions

Syntax

```
Entity.CheckCollisions()
```

CheckShaderParamCallbacks

Check all the currently set shader param callbacks on the renderproxy with the current state of the entity.

Syntax

```
Entity.UpdateShaderParamCallback()
```

CloneMaterial

Replace material on the slot with a cloned version of the material. Cloned material can be freely changed uniquely for this entity.

Syntax

```
Entity.CloneMaterial(int slot)
```

Parameters

Parameter	Description
`slot`	ID of the slot on which to clone material.
`sSubMaterialName`	If this is a non empty string this specific sub-material is cloned.

CopySlotTM

Copies the TM (Transformation Matrix) of the slot.

Syntax

```
Entity.CopySlotTM(int destSlot, int srcSlot, bool includeTranslation)
```

Parameters

Parameter	Description
`destSlot`	Destination slot identifier.

Parameter	Description
srcSlot	Source slot identifier.
includeTranslation	True to include the translation, false otherwise.

CountLinks

Counts all outgoing links of the entity.

Syntax

```
Entity.CountLinks()
```

Returns: Number of outgoing links.

CreateAuxAudioProxy

Creates an additional AudioProxy managed by the EntityAudioProxy. The created AuxAudioProxy will move and rotate with the parent EntityAudioProxy.

Syntax

```
Entity.CreateAuxAudioProxy()
```

Returns: Returns the ID of the additionally created AudioProxy.

CreateBoneAttachment

Syntax

```
Entity.CreateBoneAttachment(int characterSlot, const char *boneName, const char
 *attachmentName)
```

CreateCameraComponent

Create a camera component for the entity. Allows the entity to serve as camera source for material assigned to the entity.

Syntax

```
Entity.CreateCameraComponent()
```

CreateDRSProxy

Creates a Dynamic Response System Proxy

Syntax

```
Entity.CreateDRSProxy()
```

Returns: Returns the ID of the created proxy.

CreateLink

Creates a new outgoing link for this entity.

Syntax

```
Entity.CreateLink(const char *name)
```

Returns: nothing

Parameters

Parameter	Description
`name`	Name of the link. Does not have to be unique among all the links of this entity. Multiple links with the same name can coexist.
`(optional) targetId`	If specified, the ID of the entity the link shall target. If not specified or 0 then the link will not target anything. Default value: 0

CreateRenderComponent

Create a render component object for the entity. Allows an entity to be rendered immediately without loading any assets.

Syntax

```
Entity.CreateRenderComponent()
```

CreateSkinAttachment

Syntax

```
Entity.CreateSkinAttachment(int characterSlot, const char *attachmentName)
```

Damage

Syntax

```
Entity.Damage()
```

DeleteParticleEmitter

Deletes particles emitter from 3dengine.

Syntax

```
Entity.DeleteParticleEmitter(int slot)
```

Parameters

Parameter	Description
`slot`	slot number

DeleteThis

Deletes the current entity.

Syntax

```
Entity.DeleteThis()
```

DestroyAttachment

Syntax

```
Entity.DestroyAttachment(int characterSlot, const char *attachmentName)
```

DestroyPhysics

Syntax

```
Entity.DestroyPhysics()
```

DetachAll

Syntax

```
Entity.DetachAll()
```

DetachThis

Syntax

```
Entity.DetachThis()
```

DisableAnimationEvent

Syntax

```
Entity.DisableAnimationEvent(int nSlot,const char *sAnimation)
```

DrawSlot

Enables/Disables drawing of object or character at specified slot of the entity.

Syntax

```
Entity.DrawSlot(int nSlot,int nEnable)
```

Parameters

Parameter	Description
`nSlot`	Slot identifier.
`nEnable`	1-Enable drawing, 0-Disable drawing.

EnableBoneAnimation

Syntax

```
Entity.EnableBoneAnimation(int characterSlot, int layer, const char *boneName, bool status)
```

EnableBoneAnimationAll

Syntax

```
Entity.EnableBoneAnimationAll(int characterSlot, int layer, bool status)
```

EnableDecals

Enables decals.

Syntax

```
Entity.EnableDecals(int slot, bool enable)
```

EnableInheritXForm

Enables/Disable entity from inheriting transformation from the parent.

Syntax

```
Entity.EnableInheritXForm(bool bEnable)
```

EnableMaterialLayer

Syntax

```
Entity.EnableMaterialLayer(bool enable, int layer)
```

EnablePhysics

Syntax

```
Entity.EnablePhysics(bool bEnable)
```

EnableProceduralFacialAnimation

Syntax

```
Entity.EnableProceduralFacialAnimation(bool enable)
```

ExecuteAudioTrigger

Execute the specified audio trigger and attach it to the entity. The created audio object will move and rotate with the entity.

Syntax

```
Entity.ExecuteAudioTrigger(ScriptHandle const hTriggerID, ScriptHandle const
 hAudioProxyLocalID)
```

Returns: nil

Parameters

Parameter	Description
`hTriggerID`	the audio trigger ID handle
`hAudioProxyLocalID`	The ID of the `AuxAudioProxy` that is local to the `EntityAudioProxy`. To address the default `AuxAudioProxy`, pass `1`. To address all `AuxAudioProxy` instances, pass `0`.

FadeGlobalDensity

Sets the fade global density.

Syntax

```
Entity.FadeGlobalDensity(int nSlot, float fadeTime, float newGlobalDensity)
```

Parameters

Parameter	Description
`nSlot`	nSlot identifier.
`fadeTime`	.
`newGlobalDensity`	.

ForceCharacterUpdate

Syntax

```
Entity.ForceCharacterUpdate(int characterSlot, bool updateAlways)
```

ForwardTriggerEventsTo

Syntax

```
Entity.ForwardTriggerEventsTo(ScriptHandle entityId)
```

FreeAllSlots

Delete all objects on every slot part of the entity.

Syntax

```
Entity.FreeAllSlots()
```

FreeSlot

Delete all objects from specified slot.

Syntax

```
Entity.FreeSlot(int nSlot)
```

Parameters

Parameter	Description
nSlot	Slot identifier.

GetAIName

Syntax

```
Entity.GetAIName()
```

GetAllAuxAudioProxiesID

Returns the ID used to address all AuxAudioProxy of the parent EntityAudioProxy.

Syntax

```
Entity.GetAllAuxAudioProxiesID()
```

Returns: Returns the ID used to address all AuxAudioProxy of the parent EntityAudioProxy.

GetAngles

Gets the angle of the entity.

Syntax

```
Entity.GetAngles()
```

GetAnimationLength

Syntax

```
Entity.GetAnimationLength(int characterSlot, const char *animation)
```

GetAnimationTime

Syntax

```
Entity.GetAnimationTime(int nSlot,int nLayer)
```

GetArchetype

Retrieve the archetype of the entity.

Syntax

```
Entity.GetArchetype()
```

Returns: name of entity archetype, nil if no archetype.

GetAttachmentBone

Syntax

```
Entity.GetAttachmentBone(int characterSlot, const char *attachmentName)
```

GetAttachmentCGF

Syntax

```
Entity.GetAttachmentCGF(int characterSlot, const char *attachmentName)
```

GetBoneAngularVelocity

Syntax

```
Entity.GetBoneAngularVelocity(int characterSlot, const char *boneName)
```

GetBoneDir

Syntax

```
Entity.GetBoneDir()
```

GetBoneLocal

Syntax

```
Entity.GetBoneLocal(const char *boneName, Vec3 trgDir)
```

GetBoneNameFromTable

Syntax

```
Entity.GetBoneNameFromTable()
```

GetBonePos

Syntax

```
Entity.GetBonePos()
```

GetBoneVelocity

Syntax

```
Entity.GetBoneVelocity(int characterSlot, const char *boneName)
```

GetCenterOfMassPos

Gets the position of the entity center of mass.

Syntax

```
Entity.GetCenterOfMassPos()
```

GetCharacter

Gets the character for the specified slot if there is any.

Syntax

```
Entity.GetCharacter(int nSlot)
```

GetChild

Syntax

```
Entity.GetChild(int nIndex)
```

GetChildCount

Syntax

```
Entity.GetChildCount()
```

GetCurAnimation

Syntax

```
Entity.GetCurAnimation()
```

GetDefaultAuxAudioProxyID

Returns the ID of the default AudioProxy of the parent EntityAudioProxy.

Syntax

```
Entity.GetDefaultAuxAudioProxyID()
```

Returns: Returns the ID of the default AudioProxy of the parent EntityAudioProxy.

GetDirectionVector

Syntax

```
Entity.GetDirectionVector()
```

GetDistance

Syntax

```
float Entity.GetDistance(entityId)
```

Returns: The distance from entity specified with entityId/

GetEntitiesInContact

Syntax

```
Entity.GetEntitiesInContact()
```

GetEntityMaterial

Syntax

```
Entity.GetEntityMaterial()
```

GetExplosionImpulse

Syntax

```
Entity.GetExplosionImpulse()
```

GetExplosionObstruction

Syntax

```
Entity.GetExplosionObstruction()
```

GetFlags

Syntax

```
Entity.GetFlags()
```

GetFlagsExtended

Syntax

```
Entity.GetFlagsExtended()
```

GetGeomCachePrecachedTime

Gets time delta from current playback position to last ready to play frame.

Syntax

```
Entity.GetGeomCachePrecachedTime()
```

GetGravity

Syntax

```
Entity.GetGravity()
```

GetHelperAngles

Syntax

```
Entity.GetHelperAngles()
```

GetHelperDir

Syntax

```
Entity.GetHelperDir()
```

GetHelperPos

Syntax

```
Entity.GetHelperPos()
```

GetLink

Returns the link at given index.

Syntax

```
Entity.GetLink()
```

Returns: The script table of the entity that the i'th link is targeting or nil if the specified index is out of bounds.

Parameters

Parameter	Description
`ith`	The index of the link that shall be returned.

GetLinkName

Returns the name of the link that is targeting the entity with given ID.

Syntax

```
Entity.GetLinkName(ScriptHandle targetId)
```

Returns: The name of the i'th link targeting given entity or nil if no such link exists.

Parameters

Parameter	Description
`targetId`	ID of the entity for which the link name shall be looked up.
`(optional) ith`	If specified, the i'th link that targets given entity. Default value: 0 (first entity)

GetLinkTarget

Returns the ID of the entity that given link is targeting.

Syntax

```
Entity.GetLinkTarget(const char *name)
```

Returns: The ID of the entity that the link is targeting or nil if no such link exists.

Parameters

Parameter	Description
`name`	Name of the link.
`(optional) ith`	If specified, the i'th link with given name for which to look up the targeted entity. Default value: 0 (first link with given name)

GetLocalAngles

Syntax

```
Vec3 Entity.GetLocalAngles(vAngles)
```

GetLocalBBox

Syntax

```
Entity.GetLocalBBox()
```

GetLocalPos

Syntax

```
Vec3 Entity.GetLocalPos()
```

GetLocalScale

Syntax

```
float Entity.GetLocalScale()
```

GetLodRatio

Syntax

```
Entity.GetLodRatio()
```

GetMass

Syntax

```
Entity.GetMass()
```

GetMaterial

Syntax

```
Entity.GetMaterial()
```

GetMaterialFloat

Change material parameter.

Syntax

```
Entity.GetMaterialFloat(int slot,int nSubMtlId,const char *sParamName)
```

Returns: Material parameter value.

Parameters

Parameter	Description
`slot`	ID of the slot on which slot to change material.
`nSubMtlId`	Specify submaterial by Id.
`sParamName`	Name of the material parameter.

GetMaterialVec3

Syntax

```
Entity.GetMaterialVec3(int slot,int nSubMtlId,const char *sParamName)
```

GetName

Syntax

```
Entity.GetName()
```

GetParent

Syntax

```
Entity.GetParent()
```

GetParentSlot

Syntax

```
Entity.GetParentSlot(int child)
```

GetPhysicalStats

Some more physics related.

Syntax

```
Entity.GetPhysicalStats()
```

GetPos

Gets the position of the entity.

Syntax

```
Entity.GetPos()
```

GetProjectedWorldBBox

Syntax

```
Entity.GetProjectedWorldBBox()
```

GetRawId

Returns entityId in raw numeric format.

Syntax

```
Entity.GetRawId()
```

GetScale

Gets the scaling value for the entity.

Syntax

```
Entity.GetScale()
```

GetSlotAngles

Gets the slot angles.

Syntax

```
Entity.GetSlotAngles(int nSlot)
```

Parameters

Parameter	Description
`nSlot`	nSlot identifier.

GetSlotCount

Gets the count of the slots.

Syntax

```
Entity.GetSlotCount()
```

GetSlotHelperPos

Syntax

```
Entity.GetSlotHelperPos(int slot, const char *helperName, bool objectSpace)
```

GetSlotPos

Gets the slot position.

Syntax

```
Entity.GetSlotPos(int nSlot)
```

Parameters

Parameter	Description
nSlot	nSlot identifier.

GetSlotScale

Gets the slot scale amount.

Syntax

```
Entity.GetSlotScale(int nSlot)
```

Parameters

Parameter	Description
nSlot	nSlot identifier.

GetSlotWorldDir

Gets the World direction of the slot.

Syntax

```
Entity.GetSlotWorldDir(int nSlot)
```

Parameters

Parameter	Description
nSlot	Slot identifier.

GetSlotWorldPos

Gets the World position of the slot.

Syntax

```
Entity.GetSlotWorldPos(int nSlot)
```

Parameters

Parameter	Description
nSlot	Slot identifier.

GetSpeed

Syntax

```
Entity.GetSpeed()
```

GetState

Syntax

```
Entity.GetState()
```

GetSubmergedVolume

Syntax

```
Entity.GetSubmergedVolume(int slot, Vec3 planeNormal, Vec3 planeOrigin)
```

GetTimeOfDayHour

Syntax

```
Entity.GetTimeOfDayHour()
```

Returns: current time of day as a float value.

GetTimeSinceLastSeen

Syntax

```
Entity.GetTimeSinceLastSeen()
```

GetTouchedPoint

Retrieves point of collision for rigid body.

Syntax

```
Entity.GetTouchedPoint()
```

GetTouchedSurfaceID

Syntax

```
Entity.GetTouchedSurfaceID()
```

GetTriggerBBox

Syntax

```
Entity.GetTriggerBBox()
```

GetUpdateRadius

Syntax

```
Entity.GetUpdateRadius()
```

GetVelocity

Syntax

```
Entity.GetVelocity()
```

GetVelocityEx

Syntax

```
Entity.GetVelocityEx()
```

GetViewDistanceMultiplier

Get the view distance multiplier.

Syntax

```
Entity.GetViewDistanceMultiplier()
```

GetVolume

Syntax

```
Entity.GetVolume(int slot)
```

GetWorldAngles

Syntax

```
Vec3 Entity.GetWorldAngles(vAngles)
```

GetWorldBBox

Syntax

```
Entity.GetWorldBBox()
```

GetWorldBoundsCenter

Gets the world bbox center for the entity (defaults to entity position if no bbox present).

Syntax

```
Entity.GetWorldBoundsCenter()
```

GetWorldDir

Syntax

```
Vec3 Entity.GetWorldDir()
```

GetWorldPos

Syntax

```
Vec3 Entity.GetWorldPos()
```

GetWorldScale

Syntax

```
float Entity.GetWorldScale()
```

GotoState

Syntax

```
Entity.GotoState(const char *sState)
```

HasFlags

Syntax

```
Entity.HasFlags(int flags)
```

HasFlagsExtended

Syntax

```
Entity.HasFlagsExtended(int flags)
```

Hide

Syntax

```
Entity.Hide()
```

HideAllAttachments

Syntax

```
Entity.HideAllAttachments(int characterSlot, bool hide, bool hideShadow)
```

HideAttachment

Syntax

```
Entity.HideAttachment(int characterSlot, const char *attachmentName, bool hide, bool
 hideShadow)
```

HideAttachmentMaster

Syntax

```
Entity.HideAttachmentMaster(int characterSlot, bool hide)
```

IgnorePhysicsUpdatesOnSlot

Ignore physics when updating the position of a slot.

Syntax

```
Entity.IgnorePhysicsUpdatesOnSlot(int nSlot)
```

Parameters

Parameter	Description
nSlot	Slot identifier.

InsertSubpipe

Syntax

```
Entity.InsertSubpipe()
```

IntersectRay

Syntax

```
Entity.IntersectRay(int slot, Vec3 rayOrigin, Vec3 rayDir, float maxDistance)
```

InvalidateTrigger

Syntax

```
Entity.InvalidateTrigger()
```

IsActive

Retrieve active status of entity.

Syntax

```
Entity.IsActive(bActivate)
```

Returns: true - Entity is active. false - Entity is not active.

IsAnimationRunning

Syntax

```
Entity.IsAnimationRunning(int characterSlot, int layer)
```

Returns: nil or not nil

Parameters

Parameter	Description
`characterSlot`	Index of the character slot.
`layer`	Index of the animation layer.

IsColliding

Syntax

```
Entity.IsColliding()
```

IsEntityInside

Syntax

```
float Entity.IsEntityInside(entityId)
```

IsEntityInsideArea

Syntax

```
Entity.IsEntityInsideArea(int areaId, ScriptHandle entityId)
```

IsFromPool

Returns if the entity came from an entity pool.

Syntax

```
Entity.IsFromPool()
```

Returns: true - Entity is from a pool. (Bookmarked) false - Entity is not from a pool. (Not bookmarked)

IsGeomCacheStreaming

Syntax

```
Entity.IsGeomCacheStreaming()
```

Returns: true if geom cache is streaming.

IsHidden

Syntax

```
Entity.IsHidden()
```

IsInState

Syntax

```
Entity.IsInState(const char *sState)
```

IsPointInsideArea

Syntax

```
Entity.IsPointInsideArea(int areaId, Vec3 point)
```

IsSlotCharacter

Checks if the slot is a character.

Syntax

```
Entity.IsSlotCharacter(int slot)
```

Parameters

Parameter	Description
`slot`	Slot identifier.

IsSlotGeometry

Checks if the slot is a geometry.

Syntax

```
Entity.IsSlotGeometry(int slot)
```

Parameters

Parameter	Description
`slot`	Slot identifier.

IsSlotLight

Checks if the slot is a light.

Syntax

```
Entity.IsSlotLight(int slot)
```

Parameters

Parameter	Description
slot	Slot identifier.

IsSlotParticleEmitter

Checks if the slot is a particle emitter.

Syntax

```
Entity.IsSlotParticleEmitter(int slot)
```

Parameters

Parameter	Description
slot	Slot identifier.

IsSlotValid

Checks if the slot is valid.

Syntax

```
Entity.IsSlotValid(int nSlot)
```

Parameters

Parameter	Description
nSlot	Slot identifier.

IsUsingPipe

Syntax

```
Entity.IsUsingPipe()
```

Returns: True if the entity is running a goalpipe or has it inserted; otherwise, false.

KillTimer

Syntax

```
Entity.KillTimer()
```

LoadCharacter

Load CGF geometry into the entity slot.

Syntax

```
Entity.LoadCharacter(int nSlot,const char *sFilename)
```

Parameters

Parameter	Description
`nSlot`	Slot identifier.
`sFilename`	CGF geometry file name.

LoadCloud

Loads the cloud XML file into the entity slot.

Syntax

```
Entity.LoadCloud(int nSlot, const char *sFilename)
```

Parameters

Parameter	Description
`nSlot`	Slot identifier.
`sFilename`	Filename.

LoadFogVolume

Loads the fog volume XML file into the entity slot.

Syntax

```
Entity.LoadFogVolume(int nSlot, SmartScriptTable table)
```

Parameters

Parameter	Description
`nSlot`	Slot identifier.
`table`	Table with fog volume properties.

LoadGeomCache

Load geom cache into the entity slot.

Syntax

```
Entity.LoadGeomCache(int nSlot,const char *sFilename)
```

Parameters

Parameter	Description
nSlot	Slot identifier.
sFilename	CAX file name.

LoadLight

Load CGF geometry into the entity slot.

Syntax

```
Entity.LoadLight(int nSlot,SmartScriptTable table)
```

Parameters

Parameter	Description
nSlot	Slot identifier.
table	Table with all the light information.

LoadObject

Load CGF geometry into the entity slot.

Syntax

```
Entity.LoadObject(int nSlot,const char *sFilename)
```

Parameters

Parameter	Description
nSlot	Slot identifier.
sFilename	CGF geometry file name.

LoadObjectLattice

Load lattice into the entity slot.

Syntax

```
Entity.LoadObjectLattice(int nSlot)
```

LoadObjectWithFlags

Load CGF geometry into the entity slot.

Syntax

```
Entity.LoadObjectWithFlags(int nSlot,const char *sFilename, const int nFlags)
```

Parameters

Parameter	Description
`nSlot`	Slot identifier.
`sFilename`	CGF geometry file name.
`nFlags`	entity load flags

LoadParticleEffect

Loads CGF geometry into the entity slot.

Syntax

```
Entity.LoadParticleEffect(int nSlot, const char *sEffectName, SmartScriptTable table)
```

Parameters

Parameter	Description
`nSlot`	Slot identifier.
`sEffectName`	Name of the particle effect (Ex: "explosions/rocket").
`(optional) bPrime`	Whether effect starts fully primed to equilibrium state.
`(optional) fPulsePeriod`	Time period between particle effect restarts.
`(optional) fScale`	Size scale to apply to particles
`(optional) fCountScale`	Count multiplier to apply to particles
`(optional) bScalePerUnit`	Scale size by attachment extent
`(optional) bCountPerUnit`	Scale count by attachment extent
`(optional) sAttachType`	string for EGeomType
`(optional) sAttachForm`	string for EGeomForm

LoadSubObject

Load geometry of one CGF node into the entity slot.

Syntax

```
Entity.LoadSubObject(int nSlot,const char *sFilename,const char *sGeomName)
```

Parameters

Parameter	Description
nSlot	Slot identifier.
sFilename	CGF geometry file name.
sGeomName	Name of the node inside CGF geometry.

LoadVolumeObject

Loads volume object.

Syntax

```
Entity.LoadVolumeObject(int nSlot, const char* sFilename)
```

Parameters

Parameter	Description
nSlot	Slot identifier.
sFilename	File name of the volume object.

LookAt

Orient the entity to look at a world space position.

Syntax

```
Entity.LookAt(Vec3 target, Vec3 axis, float angle)
```

Parameters

Parameter	Description
target	The position to look at.
axis	The correction axis. The quat type is not supported.
angle	The correction angle inradians. The quat type is not supported.

MultiplyWithSlotTM

Multiplies with the TM (Transformation Matrix) of the slot.

Syntax

```
Entity.MultiplyWithSlotTM(int slot, Vec3 pos)
```

Parameters

Parameter	Description
`slot`	Slot identifier.
`pos`	Position vector.

NetPresent

Syntax

```
Entity.NetPresent()
```

NoBulletForce

Syntax

```
Entity.NoBulletForce(bool state)
```

NoExplosionCollision

Syntax

```
Entity.NoExplosionCollision()
```

PassParamsToPipe

Syntax

```
Entity.PassParamsToPipe()
```

Physicalize

Create physical entity from the specified entity slot.

Syntax

```
Entity.Physicalize(int nSlot,int nPhysicsType,SmartScriptTable physicsParams)
```

Parameters

Parameter	Description
`nSlot`	Slot identifier of the entity to physicalize. Specify -1 to use geometries from all slots.
`nPhysicsType`	Type of physical entity to create. For possible values, see the **nPhysicsType Keys** table later in this section.

Parameter	Description
physicsParams	Table with physicalization parameters. For more information, see the **physicsParams Table Keys** table later in this section.

nPhysicsType Keys

Physics Type	Meaning
PE_AREA	Physical Area (Sphere,Box,Geometry or Shape).
PE_ARTICULATED	Ragdolls or other articulated physical entities that consist of rigid bodies connected by joints.
PE_LIVING	Live physical entity that can move through the physical world and interact with it.
PE_NONE	No physics.
PE_PARTICLE	A physical particle entity that it has only mass and radius.
PE_RIGID	Rigid body physical entity. Can have infinite mass (specified by setting mass to 0).
PE_ROPE	A physical representation of a rope. The rope can hang freely or connect two physical entities.
PE_SOFT	A system of non-rigidly connected vertices that can interact with the environment. Used for soft body physics like cloth simulation.
PE_STATIC	A static, immovable physical entity.
PE_WHEELEDVEHICLE	Physical vehicle with wheels.

Note
For more information about physical entity types, see Physical Entities (p. 1022).

physicsParams Table Keys

Physics Parameter	Description
area	This table must be set when Physics Type is PE_AREA. For more information, see the **Area Table Keys** table later in this section.
density	Object density, only used if mass is not specified or -1.
flags	Physical entity flags.
living	This table must be set when Physics Type is PE_LIVING. For more information, see the **Living Table Keys** table later in this section.
mass	Object mass, only used if density is not specified or -1.
particle	This table must be set when Physics Type is PE_PARTICLE. For more information, see the **Particle Table Keys** table later in this section.
partid	Index of the articulated body part to which the new physical entity will be attached.
stiffness_scale	Scale of character joint stiffness (multiplied with stiffness values specified from the exported model)

Particle Table Keys

Particle Parameter	Description
accel_lift	Acceleration that lifts particle with the current speed
accel_thrust	Acceleration along direction of movement
air_resistance	The air resistance coefficient, F = kv
constant_orientation	(0,1) Keep constant orientation
gravity	Gravity force vector to the air
mass	Particle mass
min_bounce_vel	Minimal velocity at which particle bounces off the surface
no_path_alignment	(0,1) Do not align particle orientation to the movement path
no_roll	(0,1) Do not roll particle on terrain
no_spin	(0,1) Do not spin particle in air
radius	Particle pseudo radius
single_contact	(0,1) Calculate only one first contact
thickness	Thickness when lying on a surface (if 0, the radius is used)
velocity	Velocity direction and magnitude vector
water_gravity	Gravity force vector when in the water.
water_resistance	Water resistance coefficient, F = kv

Living Table Keys

Living Parameter	Description
air_resistance	Air control coefficient 0..1, 1 - special value (total control of movement)
gravity	Vertical gravity magnitude
head_radius	Radius of the head
height	Vertical offset of collision geometry center
height_eye	Vertical offset of the camera
height_head	Vertical offset of the head
height_pivot	Offset from central ground position that is considered the entity center
inertia	Inertia coefficient, the greater the value, the less the inertia; 0 means no inertia.
mass	Mass of the player (in kg)
max_climb_angle	Player cannot climb surface which slope is steeper than this angle (in radians)
max_jump_angle	Player is not allowed to jump towards ground if this angle is exceeded (in radians)

Living Parameter	Description
max_vel_ground	Player cannot stand on surfaces that are moving faster than this (in radians)
min_fall_angle	Player starts falling when slope is steeper than this (in radians)
min_slide_angle	If surface slope is more than this angle, player starts sliding (in radians)
size	Collision cylinder dimensions vector (x,y,z).

Area Table Keys

Area Parameter	Description
box_max	Max vector of bounding box, must be specified if type is AREA_BOX
box_min	Min vector of bounding box, must be specified if type is AREA_BOX
falloff	Ellipsoidal falloff dimensions; 0,0,0 specifies no falloff
gravity	Gravity vector inside the physical area
height	Height of the 2D area (AREA_SHAPE), relative to the minimal Z in the points table
points	A table that contains an indexed collection of vectors in local entity space that define the 2D shape of the area (AREA_SHAPE)
radius	Radius of the area sphere; must be specified if type is AREA_SPHERE.
type	Type of the area, valid values are: AREA_SPHERE, AREA_BOX, AREA_GEOMETRY, or AREA_SHAPE
uniform	Same direction in every point, or always point to the center.

PhysicalizeAttachment

Syntax

```
Entity.PhysicalizeAttachment(int characterSlot, const char* attachmentName, bool
 physicalize)
```

PhysicalizeSlot

Syntax

```
Entity.PhysicalizeSlot(int slot, SmartScriptTable physicsParams)
```

PlayFacialAnimation

Syntax

```
Entity.PlayFacialAnimation(char* name, bool looping)
```

PreLoadParticleEffect

Pre-loads a particle effect.

Syntax

```
Entity.PreLoadParticleEffect(const char *sEffectName)
```

Parameters

Parameter	Description
sEffectName	Name of the particle effect (Ex: "explosions/rocket").

ProcessBroadcastEvent

Syntax

```
Entity.ProcessBroadcastEvent()
```

RagDollize

Syntax

```
Entity.RagDollize(int slot)
```

ReattachSoftEntityVtx

Syntax

```
Entity.ReattachSoftEntityVtx(ScriptHandle entityId, int partId)
```

RedirectAnimationToLayer0

Syntax

```
Entity.RedirectAnimationToLayer0(int characterSlot, bool redirect)
```

RegisterForAreaEvents

Registers the script proxy so that it receives area events for this entity.

Syntax

```
Entity.RegisterForAreaEvents(int enable)
```

Parameters

Parameter	Description
enable	Specify 0 to disable, or any other value to enable.

RemoveAllLinks

Removes all links of an entity.

Syntax

```
Entity.RemoveAllLinks()
```

Returns: nothing

RemoveAuxAudioProxy

Removes the `AuxAudioProxy` corresponding to the passed ID from the parent `EntityAudioProxy`.

Syntax

```
Entity.RemoveAuxAudioProxy(ScriptHandle const hAudioProxyLocalID)
```

Returns: nil

Parameters

Parameter	Description
`hAudioProxyLocalID`	The ID of the `AuxAudioProxy` to be removed from the parent `EntityAudioProxy`.

RemoveDecals

Syntax

```
Entity.RemoveDecals()
```

RemoveLink

Removes an outgoing link from the entity.

Syntax

```
Entity.RemoveLink(const char *name)
```

Returns: nothing

Parameters

Parameter	Description
`name`	Name of the link to remove.
`(optional) ith`	If specified, the `<i>`th link with the name specified that will be removed. Default value: 0 (first link with given name)

RenderAlways

Enables 'always render' on the render node, skipping any kind of culling.

Syntax

```
Entity.RenderAlways(int enable)
```

Parameters

Parameter	Description
enable	Specify 0 to disable, or any other value to enable.

RenderShadow

Syntax

```
Entity.RenderShadow()
```

ReplaceMaterial

Syntax

```
Entity.ReplaceMaterial(int slot, const char *name, const char *replacement)
```

ResetAnimation

Syntax

```
Entity.ResetAnimation(int characterSlot, int layer)
```

ResetAttachment

Syntax

```
Entity.ResetAttachment(int characterSlot, const char *attachmentName)
```

ResetMaterial

Syntax

```
Entity.ResetMaterial(int slot)
```

ResetPhysics

Syntax

```
Entity.ResetPhysics()
```

SelectPipe

Syntax

```
Entity.SelectPipe()
```

SetAIName

Syntax

```
Entity.SetAIName()
```

SetAngles

Sets the angle of the entity.

Syntax

```
Entity.SetAngles(Ang3 vAngles)
```

Parameters

Parameter	Description
vAngles	Angle vector.

SetAnimateOffScreenShadow

Syntax

```
Entity.SetAnimateOffScreenShadow(bool bAnimateOffScreenShadow)
```

SetAnimationBlendOut

Syntax

```
Entity.SetAnimationBlendOut(int characterSlot, int layer, float blendOut)
```

SetAnimationEvent

Syntax

```
Entity.SetAnimationEvent(int nSlot,const char *sAnimation)
```

SetAnimationFlip

Syntax

```
Entity.SetAnimationFlip(int characterSlot, Vec3 flip)
```

SetAnimationKeyEvent

Syntax

```
Entity.SetAnimationKeyEvent(nSlot, sAnimation, nFrameID, sEvent)
```

SetAnimationSpeed

Syntax

```
Entity.SetAnimationSpeed(int characterSlot, int layer, float speed)
```

SetAnimationTime

Syntax

```
Entity.SetAnimationTime(int nSlot,int nLayer,float fTime)
```

SetAttachmentAngles

Syntax

```
Entity.SetAttachmentAngles(int characterSlot, const char *attachmentName, Vec3 angles)
```

SetAttachmentCGF

Syntax

```
Entity.SetAttachmentCGF(int characterSlot, const char *attachmentName, const char*
 filePath)
```

SetAttachmentDir

Syntax

```
Entity.SetAttachmentDir(int characterSlot, const char *attachmentName, Vec3 dir, bool
 worldSpace)
```

SetAttachmentEffect

Syntax

```
Entity.SetAttachmentEffect(int characterSlot, const char *attachmentName, const char
 *effectName, Vec3 offset, Vec3 dir, float scale, int flags)
```

SetAttachmentLight

Syntax

```
Entity.SetAttachmentLight(int characterSlot, const char *attachmentName, SmartScriptTable
 lightTable, int flags)
```

SetAttachmentObject

Syntax

```
Entity.SetAttachmentObject(int characterSlot, const char *attachmentName, ScriptHandle
 entityId, int slot, int flags)
```

SetAttachmentPos

Syntax

```
Entity.SetAttachmentPos(int characterSlot, const char *attachmentName, Vec3 pos)
```

SetAudioEnvironmentID

Sets the ID of the audio environment that an entity will specify for the entities that it contains.

Syntax

```
Entity.SetAudioEnvironmentID(ScriptHandle const hAudioEnvironmentID)
```

Returns: nil

Parameters

Parameter	Description
`hAudioEnvironmentID`	audio environment ID

SetAudioObstructionCalcType

Set the Audio Obstruction/Occlusion calculation type on the underlying `GameAudioObject`.

Syntax

```
Entity.SetAudioObstructionCalcType(int const nObstructionCalcType, ScriptHandle const
 hAudioProxyLocalID)
```

Returns: nil

Parameters

Parameter	Description
`nObstructionCalcType`	Obstruction/Occlusion calculation type; Possible values: 0 - ignore Obstruction/Occlusion 1 - use single physics ray 2 - use multiple physics rays (currently 5 per object)
`hAudioProxyLocalID`	The ID of the `AuxAudioProxy` that is local to the `EntityAudioProxy`. To address the default `AuxAudioProxy`, pass `1`. To address all `AuxAudioProxy` instances, pass `0`.

SetAudioProxyOffset

Set offset on the audio proxy attached to the entity.

Syntax

```
Entity.SetAudioProxyOffset(Vec3 const vOffset, ScriptHandle const hAudioProxyLocalID)
```

Returns: nil

Parameters

Parameter	Description
`vOffset`	The offset vector
`hAudioProxyLocalID`	The ID of the `AuxAudioProxy` that is local to the `EntityAudioProxy`. To address the default `AuxAudioProxy`, pass `1`. To address all `AuxAudioProxy` instances, pass `0`.

SetAudioRtpcValue

Set the specified audio RTPC to the specified value on the current entity.

Syntax

```
Entity.SetAudioRtpcValue(ScriptHandle const hRtpcID, float const fValue, ScriptHandle const
 hAudioProxyLocalID)
```

Returns: nil

Parameters

Parameter	Description
`hRtpcID`	The audio RTPC ID handle
`fValue`	The RTPC value
`hAudioProxyLocalID`	The ID of the `AuxAudioProxy` that is local to the `EntityAudioProxy`. To address the default `AuxAudioProxy`, pass `1`. To address all `AuxAudioProxy` instances, pass `0`.

SetAudioSwitchState

Set the specified audio switch to the specified state on the current Entity.

Syntax

```
Entity.SetAudioSwitchState(ScriptHandle const hSwitchID, ScriptHandle const hSwitchStateID,
 ScriptHandle const hAudioProxyLocalID)
```

Returns: nil

Parameters

Parameter	Description
`hSwitchID`	The audio switch ID handle
`hSwitchStateID`	The switch state ID handle

Parameter	Description
`hAudioProxyLocalID`	The ID of the `AuxAudioProxy` that is local to the `EntityAudioProxy`. To address the default `AuxAudioProxy`, pass `1`. To address all `AuxAudioProxy` instances, pass `0`.

SetCharacterPhysicParams

Syntax

```
Entity.SetCharacterPhysicParams()
```

SetCloudMovementProperties

Sets the cloud movement properties.

Syntax

```
Entity.SetCloudMovementProperties(int nSlot, SmartScriptTable table)
```

Parameters

Parameter	Description
`nSlot`	Slot identifier.
`table`	Table property for the cloud movement.

SetColliderMode

Syntax

```
Entity.SetColliderMode(int mode)
```

SetCurrentAudioEnvironments

Sets the correct audio environment amounts based on the entity's position in the world.

Syntax

```
Entity.SetCurrentAudioEnvironments()
```

Returns: nil

SetDefaultIdleAnimations

Syntax

```
Entity.SetDefaultIdleAnimations()
```

SetDirectionVector

Syntax

```
Entity.SetDirectionVector(Vec3 dir)
```

SetEnvironmentFadeDistance

Sets the distance over which this entity fades the audio environment for all approaching entities.

Syntax

```
Entity.SetEnvironmentFadeDistance(float const fEnvironmentFadeDistance)
```

Returns: nil

Parameters

Parameter	Description
`fEnvironmentFadeDistance`	The fade distance in meters.

SetFadeDistance

Sets the distance at which this entity executes fade calculations.

Syntax

```
Entity.SetFadeDistance(float const fFadeDistance)
```

Returns: nil

Parameters

Parameter	Description
`fFadeDistance`	The fade distance in meters.

SetFlags

Mode: 0: or 1: and 2: xor

Syntax

```
Entity.SetFlags(int flags, int mode)
```

SetFlagsExtended

Mode: 0: or 1: and 2: xor

Syntax

```
Entity.SetFlagsExtended(int flags, int mode)
```

SetGeomCacheDrawing

Activates or deactivates geom cache drawing.

Syntax

```
Entity.SetGeomCacheDrawing(bool active)
```

SetGeomCacheParams

Sets geometry cache parameters.

Syntax

```
Entity.SetGeomCacheParams(bool looping, const char *standIn, const char *standInMaterial,
 const char *firstFrameStandIn, const char* firstFrameStandInMaterial, const char*
 lastFrameStandIn, const char* lastFrameStandInMaterial, float standInDistance, float
 streamInDistance)
```

SetGeomCachePlaybackTime

Sets the playback time.

Syntax

```
Entity.SetGeomCachePlaybackTime(float time)
```

SetGeomCacheStreaming

Activates/deactivates geom cache streaming.

Syntax

```
Entity.SetGeomCacheStreaming(bool active, float time)
```

SetLightColorParams

changes the color related params of an existing light.

Syntax

```
Entity.SetLightColorParams(int nSlot, Vec3 color, float specular_multiplier)
```

SetLinkTarget

Specifies the entity that an existing link shall target. Use this function to change the target of an existing link.

Syntax

```
Entity.SetLinkTarget(const char *name, ScriptHandle targetId)
```

Returns: nothing

Parameters

Parameter	Description
`name`	Name of the link that shall target given entity.

Parameter	Description
`targetId`	The ID of the entity the link shall target. Pass in NULL_ENTITY to make the link no longer target an entity.
`(optional) ith`	If specified, the `<i>`th link with given name that targets the specified entity. Default value: 0 (first link with given name)

SetLocalAngles

Syntax

```
Entity.SetLocalAngles(Ang3 vAngles)
```

SetLocalBBox

Syntax

```
Entity.SetLocalBBox(Vec3 vMin,Vec3 vMax)
```

SetLocalPos

Syntax

```
Entity.SetLocalPos(Vec3 vPos)
```

SetLocalScale

Syntax

```
Entity.SetLocalScale(float fScale)
```

SetLodRatio

Syntax

```
Entity.SetLodRatio()
```

SetMaterial

Syntax

```
Entity.SetMaterial()
```

SetMaterialFloat

Change material parameter.

Syntax

```
Entity.SetMaterialFloat(int slot,int nSubMtlId,const char *sParamName,float fValue)
```

Parameters

Parameter	Description
`slot`	ID of the slot on which to change material.
`nSubMtlId`	Specify sub-material by ID.
`sParamName`	Name of the material parameter.
`fValue`	New material parameter value.

SetMaterialVec3

Syntax

```
Entity.SetMaterialVec3(int slot,int nSubMtlId,const char *sParamName,Vec3 fValue)
```

SetName

Syntax

```
Entity.SetName()
```

SetParentSlot

Syntax

```
Entity.SetParentSlot(int child, int parent)
```

SetPhysicParams

Syntax

```
Entity.SetPhysicParams()
```

SetPos

Sets the position of the entity.

Syntax

```
Entity.SetPos(Vec3 vPos)
```

Parameters

Parameter	Description
`vPos`	Position vector.

SetPublicParam

Sets a shader parameter.

Syntax

```
Entity.SetPublicParam()
```

Parameters

Parameter	Description
`paramName`	The name of the shader parameter.
`value`	The new value of the parameter.

SetRegisterInSectors

Syntax

```
Entity.SetRegisterInSectors()
```

SetScale

Sets the scaling value for the entity.

Syntax

```
Entity.SetScale(float fScale)
```

Parameters

Parameter	Description
`fScale`	The scale amount.

SetScriptUpdateRate

Syntax

```
Entity.SetScriptUpdateRate(int nMillis)
```

SetSelfAsLightCasterException

Makes the entity render node a caster exception for the light loaded in `nLightSlot`.

Syntax

```
Entity.SetSelfAsLightCasterException(int nLightSlot)
```

Parameters

Parameter	Description
`nLightSlot`	Slot where the light is loaded.

SetSlotAngles

Sets the slot angles.

Syntax

```
Entity.SetSlotAngles(int nSlot, Ang3 v)
```

Parameters

Parameter	Description
nSlot	nSlot identifier.
v	Angle to be set.

SetSlotHud3D

Setup flags for use as 3D HUD entity.

Syntax

```
Entity.SetSlotHud3D(int nSlot)
```

Parameters

Parameter	Description
nSlot	Slot identifier.

SetSlotPos

Sets the slot position.

Syntax

```
Entity.SetSlotPos(int slot,Vec3 v)
```

Parameters

Parameter	Description
slot	slot identifier.
v	Position to be set.

SetSlotPosAndDir

Sets the slot position and direction.

Syntax

```
Entity.SetSlotPosAndDir(int nSlot, Vec3 pos, Vec3 dir)
```

Parameters

Parameter	Description
nSlot	nSlot identifier.
pos	Position to be set.
dir	Direction to be set.

SetSlotScale

Sets the slot scale amount.

Syntax

```
Entity.SetSlotScale(int nSlot,float fScale)
```

Parameters

Parameter	Description
nSlot	Slot identifier.
fScale	Scale amount for the slot.

SetSlotWorldTM

Sets the World TM (transformation matrix) of the slot.

Syntax

```
Entity.SetSlotWorldTM(int nSlot, Vec3 pos, Vec3 dir)
```

Parameters

Parameter	Description
nSlot	Slot identifier.
pos	Position vector.
dir	Direction vector.

SetStateClientside

Syntax

```
Entity.SetStateClientside()
```

SetTimer

Syntax

```
Entity.SetTimer()
```

SetTriggerBBox

Syntax

```
Entity.SetTriggerBBox(Vec3 vMin,Vec3 vMax)
```

SetUpdatePolicy

Changes the update policy for the entity. Update policy controls when an entity becomes active or inactive (for example, when visible or when in close proximity).

> **Note**
> Because all active entities are updated every frame, having too many active entities can affect performance.

Syntax

```
Entity.SetUpdatePolicy(int nUpdatePolicy)
```

Parameters

Parameter	Description
`nUpdatePolicy`	Update policy constant. See the following table for possible values.

nUpdatePolicy Possible Values

Update Policy	Meaning
`ENTITY_UPDATE_NEVER`	Never update this entity.
`ENTITY_UPDATE_IN_RANGE`	Activate entity when in specified radius.
`ENTITY_UPDATE_POT_VISIBLE`	Activate entity when potentially visible.
`ENTITY_UPDATE_VISIBLE`	Activate entity when visible in frustum.
`ENTITY_UPDATE_PHYSICS`	Activate entity when physics awakes, deactivate when physics go to sleep.
`ENTITY_UPDATE_PHYSICS_VISIBLE`	Same as ENTITY_UPDATE_PHYSICS, but also activates when visible.
`ENTITY_UPDATE_ALWAYS`	Entity is always active and updated every frame.

> **Note**
> For update policies that require a radius, use SetUpdateRadius (p. 858).

SetUpdateRadius

Syntax

```
Entity.SetUpdateRadius()
```

SetVelocity

Syntax

```
Entity.SetVelocity(Vec3 velocity)
```

SetVelocityEx

Syntax

```
Entity.SetVelocityEx(Vec3 velocity, Vec3 angularVelocity)
```

SetViewDistanceMultiplier

Set the view distance multiplier.

Syntax

```
Entity.SetViewDistanceMultiplier()
```

SetViewDistUnlimited

Syntax

```
Entity.SetViewDistUnlimited()
```

SetVolumeObjectMovementProperties

Sets the properties of the volume object movement.

Syntax

```
Entity.SetVolumeObjectMovementProperties(int nSlot, SmartScriptTable table)
```

Parameters

Parameter	Description
`nSlot`	Slot identifier.
`table`	Table with volume object properties.

SetWorldAngles

Syntax

```
Entity.SetWorldAngles(Ang3 vAngles)
```

SetWorldPos

Syntax

```
Entity.SetWorldPos(Vec3 vPos)
```

SetWorldScale

Syntax

```
Entity.SetWorldScale(float fScale)
```

StartAnimation

Syntax

```
Entity.StartAnimation()
```

StopAnimation

Syntax

```
Entity.StopAnimation(int characterSlot, int layer)
```

StopAudioTrigger

Stop the audio event generated by the trigger with the specified ID on this entity.

Syntax

```
Entity.StopAudioTrigger(ScriptHandle const hTriggerID, ScriptHandle const
 hAudioProxyLocalID)
```

Returns: nil

Parameters

Parameter	Description
`hTriggerID`	The audio trigger ID handle
`hAudioProxyLocalID`	The ID of the `AuxAudioProxy` that is local to the `EntityAudioProxy`. To address the default `AuxAudioProxy`, pass `1`. To address all `AuxAudioProxy` instances, pass `0`.

ToGlobal

Syntax

```
Entity.ToGlobal(int slotId, Vec3 point)
```

ToLocal

Syntax

```
Entity.ToLocal(int slotId, Vec3 point)
```

TriggerEvent

Syntax

```
Entity.TriggerEvent()
```

UnSeenFrames

Syntax

```
Entity.UnSeenFrames()
```

UpdateAreas

Syntax

```
Entity.UpdateAreas()
```

UpdateLightClipBounds

Update the clip bounds of the light from the linked entities.

Syntax

```
Entity.UpdateLightClipBounds(int nSlot)
```

Parameters

Parameter	Description
nSlot	Slot identifier.

UpdateSlotPhysics

Syntax

```
Entity.UpdateSlotPhysics(int slot)
```

VectorToGlobal

Syntax

```
Entity.VectorToGlobal(int slotId, Vec3 dir)
```

VectorToLocal

Syntax

```
Entity.VectorToLocal(int slotId, Vec3 dir)
```

ScriptBind_Movie

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists C++ movie functions that can be called from Lua scripts.

AbortSequence

Aborts the specified sequence.

Syntax

```
Movie.AbortSequence(const char *sSequenceName)
```

Parameter	Description
sSequenceName	Sequence name.

PauseSequences

Pauses all the sequences.

Syntax

```
Movie.PauseSequences()
```

PlaySequence

Plays the specified sequence.

Syntax

```
Movie.PlaySequence(const char *sSequenceName)
```

Parameter	Description
sSequenceName	Sequence name.

ResumeSequences

Resume all the sequences.

Syntax

```
Movie.ResumeSequences()
```

StopAllCutScenes

Stops all the cut scenes.

Syntax

```
Movie.StopAllCutScenes()
```

StopAllSequences

Stops all the video sequences.

Syntax

```
Movie.StopAllSequences()
```

StopSequence

Stops the specified sequence.

Syntax

```
Movie.StopSequence(const char *sSequenceName)
```

Parameter	Description
sSequenceName	Sequence name.

ScriptBind_Particle

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists C++ particle functions that you can call from Lua script.

Attach

Attaches an effect.

Syntax

```
Particle.Attach()
```

CreateDecal

Creates a decal with the specified parameters.

Syntax

```
Particle.CreateDecal(Vec3 pos, Vec3 normal, float size, float lifeTime, const char
 *textureName)
```

Parameter	Description
pos	The decal position vector.

Parameter	Description
`normal`	The decal normal vector.
`size`	The decal size, expressed as a float.
`lifeTime`	The decal lifetime, expressed as a float.
`textureName`	The name of the texture.

CreateEffect

Creates a particle effect.

Syntax

```
Particle.CreateEffect(const char *name, SmartScriptTable params)
```

Parameter	Description
`name`	The name of the particle effect.
`params`	A `SmartScriptTable` of effect parameters.

CreateMatDecal

Creates a material decal.

Syntax

```
Particle.CreateMatDecal(Vec3 pos, Vec3 normal, float size, float lifeTime, const char
 *materialName)
```

Parameter	Description
`pos`	The decal position vector.
`normal`	The decal normal vector.
`size`	The decal size, expressed as a float.
`lifeTime`	The decal lifetime, expressed as a float.
`materialName`	The name of the material.

DeleteEffect

Deletes the specified particle effect.

Syntax

```
Particle.DeleteEffect(const char *name)
```

Parameter	Description
name	The name of the particle effect to delete.

Detach

Detaches an effect.

Syntax

```
Particle.Detach()
```

IsEffectAvailable

Checks if the specified particle effect is available.

Syntax

```
Particle.IsEffectAvailable(const char *name)
```

Parameter	Description
name	The name of the particle effect to check for availability.

SpawnEffect

Spawns an effect.

Syntax

```
Particle.SpawnEffect(const char *effectName, Vec3 pos, Vec3 dir)
```

Parameter	Description
effectName	The name of the effect to spawn.
pos	The position vector of the effect.
dir	The direction vector of the effect.

SpawnEffectLine

Spawns an effect line.

Syntax

```
Particle.SpawnEffectLine(const char *effectName, Vec3 startPos, Vec3 endPos, Vec3 dir,
 float scale, int slices)
```

Parameter	Description
`effectName`	The name of the effect.
`startPos`	The start position of the effect.
`endPos`	The end position of the effect.
`dir`	The direction of the effect.
`scale`	The scale value of the effect, expressed as a float.
`slices`	The number of slices.

SpawnParticles

Spawns a particle effect.

Syntax

```
Particle.SpawnParticles(SmartScriptTable params, Vec3 pos, Vec3 dir)
```

Parameter	Description
`params`	A `SmartScriptTable` of particle effect parameters.
`pos`	The position vector of the particle effect.
`dir`	The direction vector of the particle effect.

ScriptBind_Physics

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists C++ physics functions that you can call from Lua script.

RayTraceCheck

Checks if a ray segment intersects anything from its source to its destination.

Syntax

```
Physics.RayTraceCheck(Vec3 src,Vec3 dst,ScriptHandle skipEntityId1,ScriptHandle
 skipEntityId2)
```

Parameter	Description
`src`	The origin point of the ray segment.
`dst`	The end point of the ray segment.

Parameter	Description
skipEntityId1	Entity ID to skip when checking for intersection.
skipEntityId2	Entity ID to skip when checking for intersection.

RayWorldIntersection

Checks if a ray segment intersects anything from its source to its destination.

Syntax

```
Physics.RayWorldIntersection(Vec3 vPos, Vec3 vDir, int nMaxHits, int iEntTypes [,
 skipEntityId1 [, skipEntityId2]])
```

Parameter	Description
vPos	The origin point of the ray.
vDir	The direction of the ray.
nMaxHits	The maximum number of hits to return, sorted in nearest to farthest order.
iEntTypes	A bitmask of physical entity types. The ray intersects only with entities that the mask specifies (ent_all,...).
skipEntityId1	Optional. An entity ID to skip when checking for intersection.
skipEntityId2	Optional. An entity ID to skip when checking for intersection.

RegisterExplosionCrack

Registers a new crack for a breakable object.

Syntax

```
Physics.RegisterExplosionCrack(const char *sGeometryFile,int nIdMaterial)
```

Parameter	Description
sGeometryFile	The name of the static geometry file for the crack (CGF).
nMaterialId	The ID of the breakable material to which the crack is applied.

RegisterExplosionShape

Registers a new explosion shape from static geometry.

Note
`RegisterExplosionShape` applies only physical forces; it does not apply any game related explosion damages.

Syntax

```
Physics.RegisterExplosionShape(RegisterExplosionShape(IFunctionHandler *pH,const char
 *sGeometryFile,float fSize,int nIdMaterial,float fProbability,const char *sSplintersFile,
 float fSplintersOffset, const char *sSplintersCloudEffect)
```

Parameter	Description
`sGeometryFile`	The name of the static geometry file (CGF).
`fSize`	The scale for the static geometry.
`nIdMaterial`	The ID of the breakable material on which the shape is applied.
`fProbability`	The preference ratio for using this shape instead of other registered shapes.
`sSplintersFile`	The name of a CGF file that contains additional non-physicalized splinters to place on cut surfaces.
`fSplintersOffset`	The lower splinters position in relation to the upper one.
`sSplintersCloudEffect`	The particle effect when the splinters constraint breaks.

SamplePhysEnvironment

Find the physical entities touched by a sphere.

Syntax

```
Physics.SamplePhysEnvironment(pt, r [, objtypes])
```

Parameter	Description
`pt`	The center of the sphere.
`r`	The radius of the sphere.
`objtypes`	Optional. The types of physical entities that the sphere touches.

SimulateExplosion

Simulates a physical explosion.

Note
`SimulateExplosion` applies only physical forces; it does not apply any game related explosion damages.

Syntax

```
Physics.SimulateExplosion(SmartScriptTable explosionParams)
```

`explosionParams` is a `SmartScriptTable` whose elements are as follows:

`explosionParams` Elements

Parameter	Description
`pos`	The epicenter of the explosion.
`radius`	The radius of the explosion.
`direction`	The direction of the explosion impulse.
`impulse_pos`	The position of the explosion impulse. This value can be different from the epicenter of the explosion.
`impulse_presure`	The pressure of the explosion impulse at the specified radius from the epicenter.
`rmin`	The minimal radius of the explosion. At this radius, full pressure is applied.
`rmax`	The maximum radius of the explosion. At this radius, the impulse pressure is close to zero.
`hole_size`	The size of the hole that the explosion creates in breakable objects.

ScriptBind_Script

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists C++ script-related functions that you can call from Lua script.

DumpLoadedScripts

Dumps all loaded scripts.

Syntax

```
Script.DumpLoadedScripts()
```

KillTimer

Stops a timer set by the `Script.SetTimer` function.

Syntax

```
Script.KillTimer(ScriptHandle nTimerId)
```

Parameter	Description
`nTimerId`	The ID of the timer returned by the `Script.SetTimer` function.

LoadScript

Loads the specified script.

Syntax

```
Script.LoadScript(scriptName)
```

Parameter	Description
`scriptName`	The name of the script to load.

ReloadEntityScript

Reloads the specified entity script.

Syntax

```
Script.ReloadEntityScript(const char *className)
```

Parameter	Description
`className`	Name of the entity script.

ReloadScript

Reload the script.

Syntax

```
Script.ReloadScript(scriptName)
```

Parameter	Description
`scriptName`	The name of the script to reload.

ReloadScripts

Reloads all the scripts.

Syntax

```
Script.ReloadScripts()
```

SetTimer

Sets a script timer. When the timer expires, `SetTimer` calls the Lua function specified.

Syntax

```
Script.SetTimer(int nMilliseconds, HSCRIPTFUNCTION hFunc)
```

Returns the ID assigned to the timer or nil if no ID was specified.

Parameter	Description
`nMilliseconds`	Delay of the trigger in milliseconds.

`luaFunction`	The Lua function to call. If `userData` is specified, `luaFunction` must be in the format:
	`LuaCallback = function(userData,nTimerId)`
	`-- function body`
	`end;`
	.
	If `userData` is not specified, `luaFunction` must be in the format:
	`LuaCallback = function(nTimerId)`
	`-- function body`
	`end;`

`userData`	Optional. Specifies a user defined table. If `userData` is specified, the table is passed as the first argument of the callback function.
`bUpdateDuringPause`	Optional. The timer is updated and triggered even if the game is in pause mode.

SetTimerForFunction

Sets a timer for the specified function.

Syntax

```
Script.SetTimerForFunction(int nMilliseconds, const char *sFunctionName)
```

Returns the ID assigned to the timer, or nil if no ID was specified.

This function has the same parameters as the `SetTimer` function.

UnloadScript

Unloads the specified script.

Syntax

```
Script.UnloadScript(scriptName)
```

Parameter	Description
`scriptName`	The name of the script to unload.

ScriptBind_Sound

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists C++ sound functions that can be called from Lua scripts.

GetAudioEnvironmentID

Get the audio environment `TAudioEnvironmentID` (wrapped into a `ScriptHandle`).

Syntax

```
Sound.GetAudioEnvironmentID(const char* const sEnvironmentName)
```

Returns: `ScriptHandle` with the `TAudioEnvironmentID` value, or nil if the `sEnvironmentName` is not found.

Parameter	Description
`sEnvironmentName`	The unique name of an audio environment.

GetAudioRtpcID

Get the RTPC `TAudioControlID` (wrapped into a `ScriptHandle`).

Syntax

```
Sound.GetAudioRtpcID(const char* const sRtpcName)
```

Returns: `ScriptHandle` with the `TAudioControlID` value, or nil if the `sRtpcName` is not found.

Parameter	Description
`sRtpcName`	The unique name of an audio RTPC.

GetAudioSwitchID

Get the switch `TAudioControlID` (wrapped into a `ScriptHandle`).

Syntax

```
Sound.GetAudioSwitchID(const char* const sSwitchName)
```

Returns: `ScriptHandle` with the `TAudioControlID` value, or nil if the `sSwitchName` is not found.

Parameter	Description
`sSwitchName`	The unique name of an audio switch.

GetAudioSwitchStateID

Get the `SwitchState` `TAudioSwitchStateID` (wrapped into a `ScriptHandle`).

Syntax

```
Sound.GetAudioSwitchStateID(const ScriptHandle hSwitchID, const char* const
 sSwitchStateName)
```

Returns: `ScriptHandle` with the `TAudioSwitchStateID` value, or nil if the `sSwitchStateName` is not found.

Parameter	Description
`sSwitchStateName`	The unique name of an audio switch state.

GetAudioTriggerID

Get the trigger `TAudioControlID` (wrapped into a `ScriptHandle`).

Syntax

```
Sound.GetAudioTriggerID(const char* const sTriggerName)
```

Returns: `ScriptHandle` with the `TAudioControlID` value, or nil if the `sTriggerName` is not found.

Parameter	Description
`sTriggerName`	The unique name of an audio trigger.

SetAudioRtpcValue

Globally sets the specified audio RTPC to the specified value.

Syntax

```
Sound.SetAudioRtpcValue( hRtpcID, fValue )
```

Returns: nil

Parameter	Description
`hRtpcID`	The audio RTPC ID handle.
`fValue`	The RTPC value.

ScriptBind_System

CryLua is deprecated and will be removed in a future version of Lumberyard.

This class implements Lua script functions that expose system functionalities.

ActivatePortal

Activates or deactivates a portal.

Syntax

```
System.ActivatePortal(Vec3 vPos, bool bActivate, ScriptHandle nID)
```

Parameters

Parameter	Description
vPos	Position vector.
bActivate	True to activate the portal, false to deactivate.
nID	Entity identifier.

AddCCommand

Adds a C command to the system.

Syntax

```
System.AddCCommand(const char* sCCommandName, const char* sCommand, const char* sHelp)
```

Parameters

Parameter	Description
sCCommandName	C command name.
sCommand	Command string.
sHelp	Help for the command usage.

ApplicationTest

Test the application with the specified parameters.

Syntax

```
System.ApplicationTest(const char* pszParam)
```

Parameters

Parameter	Description
pszParam	Parameters.

Break

Breaks the application with a fatal error message.

Syntax

```
System.Break()
```

BrowseURL

Browses a URL address.

Syntax

```
System.BrowseURL(const char* szURL)
```

Parameters

Parameter	Description
`szURL`	URL string.

CheckHeapValid

Checks the heap validity.

Syntax

```
System.CheckHeapValid(const char* name)
```

Parameters

Parameter	Description
`name`	Name string. The default is `<noname>`.

ClearConsole

Clears the console.

Syntax

```
System.ClearConsole()
```

ClearKeyState

Clear the key state.

Syntax

```
System.ClearKeyState()
```

CreateDownload

Syntax

```
System.CreateDownload()
```

DebugStats

Syntax

```
System.DebugStats(bool cp)
```

DeformTerrain

Deforms the terrain.

Syntax

```
System.DeformTerrain()
```

DeformTerrainUsingMat

Deforms the terrain using material.

Syntax

```
System.DeformTerrainUsingMat()
```

Draw2DLine

Draws a 2D line.

Syntax

```
System.Draw2DLine(p1x, p1y, p2x, p2y, float r, float g, float b, float alpha)
```

Parameters

Parameter	Description
`p1x`	X value of the start point of the line.
`p1y`	Y value of the start point of the line.
`p2x`	X value of the end point of the line.
`p2y`	Y value of the end point of the line.
`r`	Red component for the label color. Default is 1.
`g`	Green component for the label color. Default is 1.
`b`	Blue component for the label color. Default is 1.
`alpha`	Alpha component for the label color. Default is 1.

DrawLabel

Draws a label with the specified parameter.

Syntax

```
System.DrawLabel(Vec3 vPos, float fSize, const char* text [, float r [, float g [, float b
 [, float alpha]]]])
```

Parameters

Parameter	Description
`vPos`	Position vector.
`fSize`	Size for the label.
`text`	Text of the label.
`r`	Red component for the label colour. Default is 1.
`g`	Green component for the label colour. Default is 1.
`b`	Blue component for the label colour. Default is 1.
`alpha`	Alpha component for the label colour. Default is 1.

DrawLine

Draws a line.

Syntax

```
System.DrawLine(Vec3 p1, Vec3 p2, float r, float g, float b, float alpha)
```

Parameters

Parameter	Description
`p1`	Start position of the line.
`p2`	End position of the line.
`r`	Red component for the label color. Default is 1.
`g`	Green component for the label color. Default is 1.
`b`	Blue component for the label color. Default is 1.
`alpha`	Alpha component for the label color. Default is 1.

DrawText

Draws text.

Syntax

```
System.DrawText(float x, float y, const char* text, const char* fontName, float size, float
 r, float g, float b, float alpha)
```

Parameters

Parameter	Description
`x`	X position for the text. The default is 0.
`y`	Y position for the text. The default is 0.

Parameter	Description
`text`	Text to be displayed. The default is an empty string.
`fontName`	Font name. The default is `default`.
`size`	Text size. The default is 16.
`r`	Red component for the label color. The default is 1.
`g`	Green component for the label color. The default is 1.
`b`	Blue component for the label color. The default is 1.
`alpha`	Alpha component for the label color. The default is 1.

DumpMemoryCoverage

Dumps memory coverage.

Syntax

```
System.DumpMemoryCoverage()
```

This function is useful for investigating memory fragmentation. When `#System.DumpMemoryCoverage()` is called from the console, `DumpMemoryCoverage` adds a line to the `MemoryCoverage.bmp` file, which is generated the first time there is a maximum line count.

DumpMemStats

Dumps memory statistics.

Syntax

```
System.DumpMemStats(bUseKB)
```

Parameters

Parameter	Description
`bUseKB`	True to use KB, false otherwise. The default is false.

DumpMMStats

Dumps the MM statistics.

Syntax

```
System.DumpMMStats()
```

DumpWinHeaps

Dumps windows heaps.

Syntax

```
System.DumpWinHeaps()
```

EnableOceanRendering

Enables/disables ocean rendering.

Syntax

```
System.EnableOceanRendering()
```

Parameters

Parameter	Description
bOcean	True to activate the ocean rendering, false to deactivate it.

EnumAAFormats

Enumerates multisample anti-aliasing formats.

Syntax

```
System.EnumAAFormats()
```

EnumDisplayFormats

Enumerates display formats.

Syntax

```
System.EnumDisplayFormats()
```

Error

Shows a message text with the error severity.

Syntax

```
System.Error(const char* sParam)
```

Parameters

Parameter	Description
sParam	Text to be logged. The default is an empty string.

ExecuteCommand

Executes a command.

Syntax

```
System.ExecuteCommand(const char* szCmd)
```

Parameters

Parameter	Description
szCmd	Command string.

GetConfigSpec

Gets the config specification.

Syntax

```
System.GetConfigSpec()
```

GetCurrAsyncTime

Gets the current asynchronous time.

Syntax

```
System.GetCurrAsyncTime()
```

GetCurrTime

Gets the current time.

Syntax

```
System.GetCurrTime()
```

GetCVar

Gets the value of a console variable.

Syntax

```
System.GetCVar(const char* sCVarName)
```

Parameters

Parameter	Description
sCVarName	Name of the variable.

GetEntities

Returns a table with all the entities currently present in a level.

Syntax

```
System.GetEntities(Vec3 center, float radius)
```

Parameters

Parameter	Description
`center`	Center position vector for the area where to get entities. The default is (0, 0, 0).
`radius`	Radius of the area. The default is 0.

GetEntitiesByClass

Gets all the entities of the specified class.

Syntax

```
System.GetEntitiesByClass(const char* EntityClass)
```

Parameters

Parameter	Description
`EntityClass`	Entity class name.

GetEntitiesInSphere

Gets all the entities contained into the specified sphere.

Syntax

```
System.GetEntitiesInSphere(Vec3 center, float radius)
```

Parameters

Parameter	Description
`center`	center position vector for the sphere where to look at entities.
`radius`	Radius of the sphere.

GetEntitiesInSphereByClass

Gets all the entities contained into the specified sphere for the specific class name.

Syntax

```
System.GetEntitiesInSphereByClass(Vec3 center, float radius, const char* EntityClass)
```

Parameters

Parameter	Description
`center`	center position vector for the sphere where to look at entities.
`radius`	Radius of the sphere.

Parameter	Description
`EntityClass`	Entity class name.

GetEntity

Gets an entity from its ID.

Syntax

```
System.GetEntity(entityId)
```

Parameters

Parameter	Description
`entityId`	Entity identifier (svtNumber or ScriptHandle).

GetEntityByName

Retrieve entity table for the first entity with specified name. If multiple entities with same name exist, first one found is returned.

Syntax

```
System.GetEntityByName(const char *sEntityName)
```

Parameters

Parameter	Description
`sEntityName`	Name of the entity to search.

GetEntityClass

Gets an entity class from its ID.

Syntax

```
System.GetEntityClass(entityId)
```

Parameters

Parameter	Description
`entityId`	Entity identifier (svtNumber or ScriptHandle).

GetEntityIdByName

Retrieve entity Id for the first entity with specified name. If multiple entities with same name exist, first one found is returned.

Syntax

```
System.GetEntityIdByName(const char *sEntityName)
```

Parameters

Parameter	Description
sEntityName	Name of the entity to search.

GetFrameID

Gets the frame identifier.

Syntax

```
System.GetFrameID()
```

GetFrameTime

Gets the frame time.

Syntax

```
System.GetFrameTime()
```

GetHDRDynamicMultiplier

Gets the HDR dynamic multiplier.

Syntax

```
System.GetHDRDynamicMultiplier()
```

GetLocalOSTime

Gets the local operating system time.

Syntax

```
System.GetLocalOSTime()
```

GetNearestEntityByClass

Gets the nearest entity with the specified class.

Syntax

```
System.GetNearestEntityByClass(Vec3 center, float radius, const char *className)
```

Parameters

Parameter	Description
center	Center position vector for the area where to look at entities.

Parameter	Description
radius	Radius of the sphere.
className	Entity class name.

GetOutdoorAmbientColor

Gets the outdoor ambient color.

Syntax

```
System.GetOutdoorAmbientColor()
```

GetPhysicalEntitiesInBox

Gets all the entities contained into the specified area.

Syntax

```
System.GetPhysicalEntitiesInBox(Vec3 center, float radius)
```

Parameters

Parameter	Description
center	Center position vector for the area where to look at entities.
radius	Radius of the sphere.

GetPhysicalEntitiesInBoxByClass

Gets all the entities contained into the specified area for the specific class name.

Syntax

```
System.GetPhysicalEntitiesInBoxByClass(Vec3 center, float radius, const char *className)
```

Parameters

Parameter	Description
center	Center position vector for the area where to look at entities.
radius	Radius of the sphere.
className	Entity class name.

GetPostProcessFxParam

Gets a post processing effect parameter value.

Syntax

```
System.GetPostProcessFxParam(const char* pszEffectParam, value)
```

Parameters

Parameter	Description
`pszEffectParam`	Parameter for the post processing effect.
`value`	Value for the parameter (float or string).

GetScreenFx

Gets a post processing effect parameter value.

> **Note**
> This is a convenience wrapper function for `GetPostProcessFxParam`.

Syntax

```
System.GetScreenFx(const char* pszEffectParam, value)
```

Parameters

Parameter	Description
`pszEffectParam`	Parameter for the post processing effect.
`value`	Value for the parameter (float or string).

GetSkyColor

Retrieve color of the sky (outdoor ambient color).

Syntax

```
Vec3 System.GetSkyColor()
```

Returns: Sky color as an `{x,y,z}` vector (x=r,y=g,z=b).

GetSkyHighlight

Retrieves sky highlighing parameters. See SetSkyHighlight (p. 896) for a description of the parameters.

Syntax

```
System.GetSkyHighlight(SmartScriptTable params)
```

GetSunColor

Retrieve color of the sun outdoors.

Syntax

```
Vec3 System.GetSunColor()
```

Returns: Sun Color as an `{x,y,z}` vector (x=r,y=g,z=b).

GetSurfaceTypeIdByName

Gets the surface type identifier by its name.

Syntax

```
System.GetSurfaceTypeIdByName(const char* surfaceName)
```

Parameters

Parameter	Description
`surfaceName`	Surface name.

GetSurfaceTypeNameById

Gets the surface type name by its identifier.

Syntax

```
System.GetSurfaceTypeNameById(int surfaceId)
```

Parameters

Parameter	Description
`surfaceId`	Surface identifier.

GetSystemMem

Gets the amount of the memory for the system.

Syntax

```
System.GetSystemMem()
```

GetTerrainElevation

Gets the terrain elevation of the specified position.

Syntax

```
System.GetTerrainElevation(Vec3 v3Pos)
```

Parameters

Parameter	Description
`v3Pos`	Position of the terraint to be checked.

GetUserName

Gets the username on this machine.

Syntax

```
System.GetUserName()
```

GetViewCameraAngles

Gets the view camera angles.

Syntax

```
System.GetViewCameraAngles()
```

GetViewCameraDir

Gets the view camera direction.

Syntax

```
System.GetViewCameraDir()
```

GetViewCameraFov

Gets the view camera fov.

Syntax

```
System.GetViewCameraFov()
```

GetViewCameraPos

Gets the view camera position.

Syntax

```
System.GetViewCameraPos()
```

GetViewCameraUpDir

Gets the view camera up-direction.

Syntax

```
System.GetViewCameraUpDir()
```

GetWind

Gets the wind direction.

Syntax

```
System.SetWind()
```

IsDevModeEnable

Checks if game is running in dev mode (cheat mode), which enables certain script function facilities (god mode, fly mode etc.).

Syntax

```
System.IsDevModeEnable()
```

IsEditing

Checks if the system is in pure editor mode - that is, not editor game mode.

Syntax

```
System.IsEditing()
```

IsEditor

Checks if the system is the editor.

Syntax

```
System.IsEditor()
```

IsHDRSupported

Checks if the HDR is supported.

Syntax

```
System.IsHDRSupported()
```

IsMultiplayer

Checks if the game is multiplayer.

Syntax

```
System.IsMultiplayer()
```

IsPointIndoors

Checks if a point is indoors.

Syntax

```
System.IsPointIndoors(Vec3 vPos)
```

Parameters

Parameter	Description
vPos	Position vector. The default is (0, 0, 0).

IsPointVisible

Checks if the specified point is visible.

Syntax

```
System.IsPointVisible(Vec3 point)
```

Parameters

Parameter	Description
point	Point vector.

IsPS20Supported

Checks if the PS20 is supported.

Syntax

```
System.IsPS20Supported()
```

IsValidMapPos

Checks if the position is a valid map position.

Syntax

```
System.IsValidMapPos(Vec3 v)
```

Parameters

Parameter	Description
v	Position vector. The default is (0, 0, 0).

LoadFont

Loads a font.

Syntax

```
System.LoadFont(const char* pszName)
```

Parameters

Parameter	Description
pszName	Font name.

LoadLocalizationXml

Loads Excel exported XML file with text and dialog localization data.

Syntax

```
System.LoadLocalizationXml(const char *filename)
```

Log

Logs a message to the log file and console.

Syntax

```
System.Log(const char* sText)
```

Parameters

Parameter	Description
sText	Text to be logged.

LogAlways

Logs data even if the verbosity setting is 0.

Syntax

```
System.LogAlways(const char* sText)
```

Parameters

Parameter	Description
sText	Text to be logged.

LogToConsole

Logs a message to the console.

Syntax

```
System.LogToConsole(const char* sText)
```

Parameters

Parameter	Description
sText	Text to be logged.

PrepareEntityFromPool

Prepares the given bookmarked entity from the pool, bringing it into existence.

Syntax

```
System.PrepareEntityFromPool(entityId)
```

Parameters

Parameter	Description
`entityId`	Entity identifier (number or ScriptHandle).
`bPrepareNow`	(optional) When another entity preparation is already in progress, specifies whether the pooled entity should be prepared immediately instead of putting it in a queue.

ProjectToScreen

Projects to the screen (not guaranteed to work if used outside Renderer).

Syntax

```
System.ProjectToScreen(Vec3 vec)
```

Parameters

Parameter	Description
`vec`	Position vector.

Quit

Quits the program.

Syntax

```
System.Quit()
```

QuitInNSeconds

Quits the application in the specified number of seconds.

Syntax

```
System.QuitInNSeconds(float fInNSeconds)
```

Parameters

Parameter	Description
`fInNSeconds`	Number of seconds before quitting.

RayTraceCheck

Checks world and static objects.

Syntax

```
System.RayTraceCheck(Vec3 src, Vec3 dst, int skipId1, int skipId2)
```

RayWorldIntersection

Shoots rays into the world.

Syntax

```
System.RayWorldIntersection(Vec3 vPos, Vec3 vDir, int nMaxHits, int iEntTypes)
```

Parameters

Parameter	Description
vPos	Position vector. The default is (0, 0, 0).
vDir	Direction vector. The default is (0, 0, 0).
nMaxHits	Maximum number of hits.
iEntTypes	

RemoveEntity

Removes the specified entity.

Syntax

```
System.RemoveEntity(ScriptHandle entityId)
```

Parameters

Parameter	Description
entityId	Entity identifier.

ResetPoolEntity

Resets the entity's bookmarked, which frees memory.

Syntax

```
System.ResetPoolEntity(entityId)
```

Parameters

Parameter	Description
entityId	Entity identifier (svtnumber or ScriptHandle).

ReturnEntityToPool

Syntax

```
System.ReturnEntityToPool(entityId)
```

Returns: the bookmarked entity to the pool, destroying it.

Parameters

Parameter	Description
`entityId`	Entity identifier (svtnumber or ScriptHandle).

SaveConfiguration

Saves the configuration.

Syntax

```
System.SaveConfiguration()
```

ScanDirectory

Scans a directory.

Syntax

```
System.ScanDirectory(const char* pszFolderName, int nScanMode)
```

Parameters

Parameter	Description
`pszFolderName`	Folder name.
`nScanMode`	Scan mode for the folder. Can be: `SCANDIR_ALL` (0), `SCANDIR_FILES` (1), or `SCANDIR_SUBDIRS` (2).

ScreenToTexture

Syntax

```
System.ScreenToTexture()
```

SetBudget

Sets system budget.

Syntax

```
System.SetBudget(int sysMemLimitInMB, int videoMemLimitInMB, float frameTimeLimitInMS,
 int soundChannelsPlayingLimit, int soundMemLimitInMB, int soundCPULimitInPercent, int
 numDrawCallsLimit)
```

Parameters

Parameter	Description
`sysMemLimitInMB`	Limit of the amount of system memory in MB. The default is 512.

Parameter	Description
videoMemLimitInMB	Limit of the amount of video memory in MB. The default is 256.
frameTimeLimitInMS	Limit of the frame time in MS. The default is 50.0f.
soundChannelsPlayingLimit	Limit of the number of sound channels playing. The default is 64.
soundMemLimitInMB	Limit of the sound memory in MB. The default is 64.
soundCPULimitInPercent	Limit of the sound CPU usage in percent. The default is 5.
numDrawCallsLimit	Limit of the number of draw calls. The default is 2000.

SetConsoleImage

Sets the console image.

Syntax

```
System.SetConsoleImage(const char* pszName, bool bRemoveCurrent)
```

Parameters

Parameter	Description
pszName	The name of the texture image.
bRemoveCurrent	True to remove the current image; otherwise false.

SetCVar

Sets the value of a console variable.

Syntax

```
System.SetCVar(const char* sCVarName, value)
```

Parameters

Parameter	Description
sCVarName	Name of the variable.
value	Value of the variable (float or string).

SetGammaDelta

Sets the gamma/delta value.

Syntax

```
System.SetGammaDelta(float fDelta)
```

Parameters

Parameter	Description
`fDelta`	Delta value. The default is 0.

SetOutdoorAmbientColor

Sets the outdoor ambient color.

Syntax

```
System.GetOutdoorAmbientColor(v3Color)
```

Parameters

Parameter	Description
`v3Color`	Outdoor ambient color value.

SetPostProcessFxParam

Sets a post processing effect parameter value.

Syntax

```
System.SetPostProcessFxParam(const char* pszEffectParam, value)
```

Parameters

Parameter	Description
`pszEffectParam`	Parameter for the post processing effect.
`value`	Value for the parameter (`svtNumber`, `svtObject`, or `svtString`).

SetScissor

Sets the scissoring screen area.

Syntax

```
System.SetScissor(float x, float y, float w, float h)
```

SetScreenFx

Sets a post processing effect parameter value.

> **Note**
> This is a convenience wrapper function for `SetPostProcessFxParam`.

Syntax

```
System.SetScreenFx(pszEffectParam, value)
```

Parameters

Parameter	Description
`pszEffectParam`	Parameter for the post processing effect.
`value`	Value for the parameter (`svtNumber`, `svtObject`, or `svtString`).

SetSkyColor

Set color of the sky (outdoors ambient color).

Syntax

```
System.SetSkyColor(Vec3 vColor)
```

Parameters

Parameter	Description
`vColor`	Sky Color as an `{x,y,z}` vector (x=r,y=g,z=b).

SetSkyHighlight

Set sky highlighing parameters.

Syntax

```
System.SetSkyHighlight(SmartScriptTable params)
```

Parameters

Parameter	Description
`params`	Table with sky highlighting parameters.

Params Table Parameters

Highlight Parameter	Description
`color`	Sky highlight color
`direction`	Direction of the sky highlight in world space.
`pos`	Position of the sky highlight in world space.
`size`	Sky highlight scale.

SetSunColor

Set the color of the sun, only relevant outdoors.

Syntax

```
System.SetSunColor(Vec3 vColor)
```

Parameters

Parameter	Description
`vColor`	Sun color as an `{x,y,z}` vector (x=r,y=g,z=b).

SetViewCameraFov

Sets the view camera fov.

Syntax

```
System.SetViewCameraFov(float fov)
```

SetVolumetricFogModifiers

Sets the volumetric fog modifiers.

Syntax

```
System.SetVolumetricFogModifiers(float gobalDensityModifier, float
 atmosphereHeightModifier)
```

Parameters

Parameter	Description
`gobalDensityModifier`	Modifier for the global density.
`atmosphereHeightModifier`	Modifier for the atmosphere height.

SetWaterVolumeOffset

`SetWaterLevel` is not supported by the 3D engine for now.

Syntax

```
System.SetWaterVolumeOffset()
```

SetWind

Sets the wind direction.

Syntax

```
System.SetWind(Vec3 vWind)
```

Parameters

Parameter	Description
`vWind`	Wind direction. The default value is (0, 0, 0).

ShowConsole

Shows or hides the console.

Syntax

```
System.ShowConsole(int nParam)
```

Parameters

Parameter	Description
nParam	1 to show the console, 0 to hide. The default is 0.

ShowDebugger

Shows the debugger.

Syntax

```
System.ShowDebugger()
```

SpawnEntity

Spawns an entity.

Syntax

```
System.SpawnEntity(SmartScriptTable params)
```

Parameters

Parameter	Description
params	Entity parameters.

ViewDistanceGet

Gets the view distance.

Syntax

```
System.ViewDistanceSet()
```

ViewDistanceSet

Sets the view distance.

Syntax

```
System.ViewDistanceSet(float fViewDist)
```

Parameters

Parameter	Description
`fViewDist`	View distance.

Warning

Shows a message text with the warning severity.

Syntax

```
System.Warning(const char* sParam)
```

Parameters

Parameter	Description
`sParam`	The text to be logged. The default value is an empty string.

ScriptBind Action Functions

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists C++ action functions that can be called from Lua scripts.

Topics

- ScriptBind_Action (p. 899)
- ScriptBind_ActionMapManager (p. 913)
- ScriptBind_ActorSystem (p. 915)
- ScriptBind_GameStatistics (p. 916)
- ScriptBind_GameToken (p. 918)
- ScriptBind_Inventory (p. 919)
- ScriptBind_ItemSystem (p. 921)
- ScriptBind_Network (p. 924)
- ScriptBind_UIAction (p. 924)
- ScriptBind_Vehicle (p. 936)
- ScriptBind_VehicleSeat (p. 943)
- ScriptBind_VehicleSystem (p. 944)

ScriptBind_Action

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists the action related Lua script bind functions. When parameters are present, the data types indicated in the signatures reflect those of the underlying C++ function.

ActivateEffect

Activates the effect specified.

Syntax

```
Action.ActivateEffect(const char * name)
```

Parameter	Description
`name`	Specifies the effect to activate.

ActivateExtensionForGameObject

Activates a specified extension for a game object.

Syntax

```
Action.ActivateExtensionForGameObject(ScriptHandle entityId, const char *extension, bool
 activate)
```

Parameter	Description
`entityId`	The identifier of the entity.
`extension`	The name of the extension.
`activate`	Specify `true` to activate the extension or `false` to deactivate it.

AddAngleSignal

Adds an angle for the signal.

Syntax

```
Action.AddAngleSignal(ScriptHandle entityId, float fAngle, float fFlexibleBoundary, const
 char *sSignal)
```

Parameter	Description
`entityId`	The identifier of the entity.
`fAngle`	The angle value.
`fFlexibleBoundary`	The size of the flexible boundary.
`sSignal`	The string for the signal.

AddRangeSignal

Adds a range for the signal.

Syntax

```
Action.AddRangeSignal(ScriptHandle entityId, float fRadius, float fFlexibleBoundary, const
 char *sSignal)
```

Parameter	Description
`entityId`	The identifier of the entity.
`fRadius`	The adius of the range area.
`fFlexibleBoundary`	Flexible boundary size.
`sSignal`	String for signal.

AddTargetRangeSignal

Adds a target range signal that has the parameters specified.

Syntax

```
Action.AddTargetRangeSignal(ScriptHandle entityId, ScriptHandle targetId, float fRadius,
 float fFlexibleBoundary, const char *sSignal)
```

Parameter	Description
`entityId`	The identifier of the entity.
`targetId`	The identifier of the target.
`fRadius`	The radius of the range area.
`fFlexibleBoundary`	The size of the flexible boundary.
`sSignal`	The string for the signal.

BanPlayer

Bans a specified player.

Syntax

```
Action.BanPlayer(ScriptHandle entityId, const char* message)
```

Parameter	Description
`entityId`	The identifier of the entity.
`message`	The message for the ban.

BindGameObjectToNetwork

Binds a specifed game object to the network.

Syntax

```
Action.BindGameObjectToNetwork(ScriptHandle entityId)
```

Parameter	Description
`entityId`	The identifier of the entity to bind to the network.

CacheItemGeometry

Caches an item geometry.

Syntax

```
Action.CacheItemGeometry(const char *itemName)
```

Parameter	Description
`itemName`	The string name of the item.

CacheItemSound

Caches an item sound.

Syntax

```
Action.CacheItemSound(const char *itemName)
```

Parameter	Description
`itemName`	The string name of the item.

ClearEntityTags

Clears the tag for the specified entity.

Syntax

```
Action.ClearEntityTags(ScriptHandle entityId)
```

Parameter	Description
`entityId`	The identifier of the entity.

ClearStaticTag

Clears the specified static tag for the specified entity.

Syntax

```
Action.ClearStaticTag(ScriptHandle entityId, const char *staticId)
```

Parameter	Description
`entityId`	The identifier of the entity.
`staticId`	The identifier of the static tag.

ConnectToServer

Connects to the server specified.

Syntax

```
Action.ConnectToServer(char* server)
```

Parameter	Description
`server`	String that specifies the server to connect to.

CreateGameObjectForEntity

Creates a game object for the entity ID specified.

Syntax

```
Action.CreateGameObjectForEntity(ScriptHandle entityId)
```

Parameter	Description
`entityId`	The identifier of the entity.

DestroyRangeSignaling

Removes range signaling.

Syntax

```
Action.DestroyRangeSignaling(ScriptHandle entityId)
```

Parameter	Description
`entityId`	The identifier of the entity.

DisableSignalTimer

Disables the signal timer.

Syntax

```
Action.DisableSignalTimer(ScriptHandle entityId, const char *sText)
```

Parameter	Description
`entityId`	The identifier of the entity.
`sText`	The text for the signal.

DontSyncPhysics

Instructs the engine to not synchronize physics for the specified entity.

Syntax

```
Action.DontSyncPhysics(ScriptHandle entityId)
```

Parameter	Description
`entityId`	The identifier of the entity.

EnableRangeSignaling

Enables or disables range signaling for the specified entity.

Syntax

```
Action.EnableRangeSignaling(ScriptHandle entityId, bool bEnable)
```

Parameter	Description
`entityId`	The identifier of the entity.
`bEnable`	Enable or disable range signaling.

EnableSignalTimer

Enables the signal timer.

Syntax

```
Action.EnableSignalTimer(ScriptHandle entityId, const char *sText)
```

Parameter	Description
`entityId`	The identifier of the entity.

Parameter	Description
`sText`	The text for the signal.

ForceGameObjectUpdate

Forces the game object to be updated.

Syntax

```
Action.ForceGameObjectUpdate(ScriptHandle entityId, bool force)
```

Parameter	Description
`entityId`	The identifier of the entity.
`force`	Specify `true` to force the update; specify `false` otherwise.

GetClassName

Returns the class name, if available, for specified `classId`.

Syntax

```
Action.GetClassName(int classId)
```

GetPlayerList

Retrieves the current players list.

Syntax

```
Action.GetPlayerList()
```

GetServer

Gets the server that corresponds to the number specified.

Syntax

```
Action.GetServer(int number)
```

Parameter	Description
`number`	The number of the server.

GetServerTime

Gets the current time on the server.

Syntax

```
Action.GetServerTime()
```

GetWaterInfo

Gets information about the water at the position specified.

Syntax

```
Action.GetWaterInfo(Vec3 pos)
```

Parameter	Description
`pos`	The position for which information will be returned.

HasAI

Returns `true` if the entity has an AI object associated with it and has been registered with the AI System

Syntax

```
Action.HasAI(ScriptHandle entityId)
```

IsChannelOnHold

Checks if the channel specified is on hold.

Syntax

```
Action.IsChannelOnHold(int channelId)
```

Parameter	Description
`channelId`	The identifier of the channel.

IsChannelSpecial

Returns `true` if the channel is special.

Syntax

```
Action.IsChannelSpecial()
```

IsClient

Returns `true` if the current script runs on a client.

Syntax

```
Action.IsClient()
```

IsGameObjectProbablyVisible

Returns `true` if the specifed object is likely visible.

Syntax

```
Action.IsGameObjectProbablyVisible(ScriptHandle gameObject)
```

Parameter	Description
`gameObject`	The game object to check for likely visibilty.

IsGameStarted

Returns `true` if the game has started.

Syntax

```
Action.IsGameStarted()
```

IsImmersivenessEnabled

Returns `true` if immersive multiplayer is enabled.

Syntax

```
Action.IsImmersivenessEnabled()
```

IsRMIServer

Returns `true` if the current script is running on an RMI (Remote Method Invocation) server.

Syntax

```
Action.IsRMIServer()
```

IsServer

Returns `true` if the current script runs on a server.

Syntax

```
Action.IsServer()
```

LoadXML

Loads XML data from the file specified. For more information, see Using the Lua XML Loader (p. 716).

Syntax

```
Action.LoadXML(const char * definitionFile, const char * dataFile)
```

Parameter	Description
`definitionFile`	Name of an XML file that declares the kind of data that is included in `dataFile`.
`dataFile`	The name of the XML file that contains the Lua data described in `definitionFile`.

PauseGame

Puts the game into pause mode.

Syntax

```
Action.PauseGame(bool pause)
```

Parameter	Description
`pause`	Specify `true` to set the game in pause mode. Specify `false` to resume the game.

Persistent2DText

Adds persistent 2D text.

Syntax

```
Action.Persistent2DText(const char* text, float size, Vec3 color, const char* name, float
 timeout)
```

Parameter	Description
`text`	The text to be displayed.
`size`	The size of the 2D text.
`color`	The color of the 2D text.
`name`	The name assigned to the 2D text.
`timeout`	The timeout for the 2D text.

PersistentArrow

Adds a persistent arrow to the world.

Syntax

```
Action.PersistentArrow(Vec3 pos, float radius, Vec3 dir, Vec3 color, const char* name,
 float timeout)
```

Parameter	Description
pos	The position of the arrow.
radius	The radius of the arrow.
dir	The direction of the arrow.
color	The color of the arrow.
name	The name assigned to the arrow.
timeout	The timeout for the arrow.

PersistentEntityTag

Adds a persistent entity tag.

Syntax

```
Action.PersistentEntityTag(ScriptHandle entityId, const char *text)
```

Parameter	Description
entityId	The identifier of the entity.
text	The text for the entity tag.

PersistentLine

Adds a persistent line to the world.

Syntax

```
Action.PersistentLine(Vec3 start, Vec3 end, Vec3 color, const char* name, float timeout)
```

Parameter	Description
start	The starting position of the line.
end	The ending position of the line.
color	The color of the line.
name	The name assigned to the line.
timeout	The timeout for the line.

PersistentSphere

Adds a persistent sphere to the world.

Syntax

```
Action.PersistentSphere(Vec3 pos, float radius, Vec3 color, const char* name, float
 timeout)
```

Parameter	Description
pos	The position of the sphere.
radius	The radius of the sphere.
color	The color of the sphere.
name	The name assigned to the sphere.
timeout	The timeout for the sphere.

PreLoadADB

Use this function to pre-cache ADB files.

Syntax

```
Action.PreLoadADB(const char* adbFileName)
```

Parameter	Description
adbFileName	The path and filename of the animation ADB file which is to be pre-loaded.

RefreshPings

Refreshes pings for all servers.

Syntax

```
Action.RefreshPings()
```

RegisterWithAI

Registers the entity to the AI system and creates an AI object associated with it.

Syntax

```
Action.RegisterWithAI()
```

ResetRangeSignaling

Resets range signaling.

Syntax

```
Action.ResetRangeSignaling(ScriptHandle entityId)
```

Parameter	Description
`entityId`	The identifier of the entity.

ResetSignalTimer

Resets the rate for the signal timer.

Syntax

```
Action.ResetSignalTimer(ScriptHandle entityId, const char *sText)
```

Parameter	Description
`entityId`	The identifier of the entity.
`sText`	Th text for the signal.

ResetToNormalCamera

Resets the camera to the last valid view stored.

Syntax

```
Action.ResetToNormalCamera()
```

SaveXML

Saves the specified XML data to the file system.

Syntax

```
Action.SaveXML(const char * definitionFile, const char * dataFile, SmartScriptTable
 dataTable)
```

Parameter	Description
`definitionFile`	Name of an XML file that declares the kind of data that is included in `dataFile`. For more information, see Using the Lua XML Loader (p. 716).
`dataFile`	The name of the XML file that contains the Lua data described in `definitionFile`.
`dataTable`	The name of the data table.

SendGameplayEvent

Sends an event for the gameplay.

Syntax

```
Action.SendGameplayEvent(ScriptHandle entityId, int event)
```

Parameter	Description
`entityId`	The identifier of the entity.
`event`	The integer of the event.

SetAimQueryMode

Sets the aim query mode for the AI proxy. By default, the AI proxy queries the movement controller if the character is aiming. You can override this behavior by using a different keyword for the `mode` parameter.

Syntax

```
Action.SetAimQueryMode(ScriptHandle entityId, int mode)
```

Parameter	Description
`entityId`	The identifier of the entity.
`mode`	Specifies one of the following values: `QueryAimFromMovementController` (the default), `OverriddenAndAiming`, or `OverriddenAndNotAiming`

SetNetworkParent

Sets the network parent.

Syntax

```
Action.SetNetworkParent(ScriptHandle entityId, ScriptHandle parentId)
```

Parameter	Description
`entityId`	The identifier of the entity.
`parentID`	The identifier of the parent network.

SetSignalTimerRate

Sets the rate for the signal timer.

Syntax

```
Action.SetSignalTimerRate(ScriptHandle entityId, const char *sText, float fRateMin, float
 fRateMax)
```

Parameter	Description
`entityId`	The identifier of the entity.

Parameter	Description
sText	The text for the signal.
fRateMin	The minimum rate for the signal timer.
fRateMax	The maximum rate for the signal timer.

SetViewCamera

Saves the previous valid view and overrides it with the current camera settings.

Syntax

```
Action.SetViewCamera()
```

ScriptBind_ActionMapManager

CryLua is deprecated and will be removed in a future version of Lumberyard.

The action map manager provides a high-level interface to handle input controls inside a game. An action map is a set of key or button mappings for a particular game mode (such as controlling a helicopter). For more information, see Controller Devices and Game Input (p. 465).

EnableActionFilter

Enables or disables a specified action filter. An action filter allows actions like `moveleft` or `moveright` to succeed or fail. For more information, see Action Filters (p. 474).

Syntax

```
ActionMapManager.EnableActionFilter( name, enable )
```

Parameter	Description
name	The name of the filter.
enable	Specify `true` to enable the filter, or `false` to disable it.

EnableActionMap

Enables or disables an action map.

Syntax

```
ActionMapManager.EnableActionMap(const char *name, bool enable)
```

Parameter	Description
name	The name of the action map to enable or disable.

Parameter	Description
`enable`	Specify `true` to enable the action map, or `false` to disable it.

EnableActionMapManager

Enables or disables the action map manager.

Syntax

```
ActionMapManager.EnableActionMapManager( enable, resetStateOnDisable )
```

Parameter	Description
`enable`	Enables or disables the action map manager. Specify `true` to enable the action map manager, or `false` to disable it.
`resetStateOnDisable`	Resets the action states when the action map manager is disabled.

GetDefaultActionEntity

Retrieves the currently set default action entity.

Syntax

```
ActionMapManager.GetDefaultActionEntity()
```

InitActionMaps

Initializes the action maps and filters found in the file specified.

Syntax

```
ActionMapManager.InitActionMaps( path )
```

Parameter	Description
`path`	The XML file path.

IsFilterEnabled

Queries whether the filter specfied is currently enabled.

Syntax

```
ActionMapManager.IsFilterEnabled( filterName )
```

Parameter	Description
`filterName`	The name of the filter whose status to check.

LoadControllerLayoutFile

Loads the given controller layout into the action map manager.

Syntax

```
ActionMapManager.LoadControllerLayoutFile( layoutName )
```

Parameter	Description
`layoutName`	The name of the layout.

LoadFromXML

Loads information from an XML file.

Syntax

```
ActionMapManager.LoadFromXML(const char *name)
```

Parameter	Description
`name`	The name of the XML file to load.

SetDefaultActionEntity

Sets a new default action entity. The action map manager assigns new action maps to the action entity that you set as the default.

Syntax

```
ActionMapManager.SetDefaultActionEntity( id, updateAll )
```

Parameter	Description
`id`	Specifies the `EntityId` of the action entity that is to become the default.
`updateAll`	Updates all existing action map assignments.

ScriptBind_ActorSystem

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists C++ actor system functions that can be called from Lua scripts.

CreateActor

Creates an actor.

Syntax

```
ActorSystem.CreateActor(ScriptHandle channelId, SmartScriptTable actorParams)
```

Parameter	Description
`channelId`	Identifier for the network channel.
`actorParams`	Parameters for the actor.

ScriptBind_GameStatistics

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists C++ game statistics functions that can be called from Lua script.

AddGameElement

Adds a game element to specified scope.

Syntax

```
GameStatistics.AddGameElement(scopeID, elementID, locatorID, locatorValue [, table])
```

Parameter	Description
`scopeID`	The identifier of the scope.
`elementID`	The identifier of the element to be added.
`locatorID`	The identifier of the locator.
`locatorValue`	The value of the locator.
`table`	Optional. The table of the element.

BindTracker

Syntax

```
GameStatistics.BindTracker(name, tracker)
```

Parameter	Description
`name`	The name of the tracker to bind.

Parameter	Description
`tracker`	The `IStatsTracker*` to be bound.

CurrentScope

Returns the ID of current scope, or `-1` if the stack is empty.

Syntax

```
GameStatistics.CurrentScope()
```

Event

Syntax

```
GameStatistics.Event()
```

PopGameScope

Removes the scope from the top of the stack.

Syntax

```
GameStatistics.PopGameScope([checkScopeId])
```

Parameter	Description
`checkScopeId`	Optional. The identifier of the scope to be removed from the top of the stack.

PushGameScope

Pushes a scope on top of the stack.

Syntax

```
GameStatistics.PushGameScope(scopeID)
```

Parameter	Description
`scopeID`	The identifier of the scope to be placed on top of the stack.

RemoveGameElement

Removes the element that has the supplied parameter values from the specified scope.

Syntax

```
GameStatistics.RemoveGameElement(scopeID, elementID, locatorID, locatorValue)
```

Parameter	Description
`scopeID`	The identifier of the scope.
`elementID`	The identifier of the element to be removed.
`locatorID`	The identifier of the locator.
`locatorValue`	The value of the locator.

StateValue

Syntax

```
GameStatistics.StateValue()
```

UnbindTracker

Syntax

```
GameStatistics.UnbindTracker(name, tracker)
```

Parameter	Description
`name`	The name of the tracker to unbind.
`tracker`	The `IStatsTracker*` to unbind.

ScriptBind_GameToken

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists C++ game token functions that can be called from Lua script.

DumpAllTokens

Dump all game tokens with their values to the log.

Syntax

```
GameToken.DumpAllTokens()
```

GetToken

Gets the value of a game token.

Syntax

```
GameToken.GetToken(const char *sTokenName)
```

Parameter	Description
sTokenName	The name of the token whose value to get.

SetToken

Sets the value of a game token.

Syntax

```
GameToken.SetToken(const char* tokenName, any tokenValue)
```

Parameter	Description
tokenName	The name of the token.
tokenValue	The value to set.

ScriptBind_Inventory

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists C++ inventory managment functions that you can call from Lua script.

Clear

Clears the inventory.

Syntax

```
Inventory.Clear()
```

Destroy

Destroys the inventory.

Syntax

```
Inventory.Destroy()
```

Dump

Dumps the inventory.

Syntax

```
Inventory.Dump()
```

GetAmmoCapacity

Gets the capacity for the specified ammunition.

Syntax

```
Inventory.GetAmmoCapacity(const char *ammoName)
```

Parameter	Description
ammoName	The name of the ammunition.

GetAmmoCount

Gets the amount of the specified ammunition name.

Syntax

```
Inventory.GetAmmoCount(const char *ammoName)
```

Parameter	Description
ammoName	The name of the ammunition.

GetCurrentItem

Gets the current item.

Syntax

```
Inventory.GetCurrentItem()
```

GetCurrentItemId

Gets the identifier of the current item.

Syntax

```
Inventory.GetCurrentItemId()
```

GetGrenadeWeaponByClass

Gets grenade weapon by class name.

Syntax

```
Inventory.GetGrenadeWeaponByClass(const char *className)
```

Parameter	Description
className	The name of the class.

GetItemByClass

Gets item by class name.

Syntax

```
Inventory.GetItemByClass(const char *className)
```

Parameter	Description
`className`	The name of the class.

HasAccessory

Checks if the inventory contains the specified accessory.

Syntax

```
Inventory.HasAccessory(const char *accessoryName)
```

Parameter	Description
`accessoryName`	The name of the accessory.

SetAmmoCount

Sets the amount of the specified ammunition.

Syntax

```
Inventory.SetAmmoCount(const char *ammoName, int count)
```

Parameter	Description
`ammoName`	The name of the ammunition.
`count`	The count of the ammunition.

ScriptBind_ItemSystem

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists C++ functions for actor items and items in packs that you can call from Lua script.

GetPackItemByIndex

Gets a pack item from its index.

Syntax

```
ItemSystem.GetPackItemByIndex(const char *packName, int index)
```

Parameter	Description
packName	The name of the pack.
index	The index of the item to retrieve.

GetPackNumItems

Get the number of items in the specified pack.

Syntax

```
ItemSystem.GetPackNumItems(const char* packName)
```

Parameter	Description
packName	The name of the pack whose item count to retrieve.

GetPackPrimaryItem

Gets the primary item of the specified pack.

Syntax

```
ItemSystem.GetPackPrimaryItem(const char *packName)
```

Parameter	Description
packName	The name of the pack whose primary item to retrieve.

GiveItem

Gives the specified item.

Syntax

```
ItemSystem.GiveItem(const char *itemName)
```

Parameter	Description
itemName	The name of the item.

GiveItemPack

Gives the item pack specified to the actor specified .

Syntax

```
ItemSystem.GiveItemPack(ScriptHandle actorId, const char *packName)
```

Parameter	Description
actorId	The actor identifier.
packName	The name of the pack.

Reset

Resets the item system.

Syntax

```
ItemSystem.Reset()
```

SerializePlayerLTLInfo

Serializes player level to level (LTL) information.

Syntax

```
ItemSystem.SerializePlayerLTLInfo(bool reading)
```

Parameter	Description
reading	Boolean value.

SetActorItem

Sets an actor item.

Syntax

```
ItemSystem.SetActorItem(ScriptHandle actorId, ScriptHandle itemId, bool keepHistory)
```

Parameter	Description
actorId	The actor identifier.
itemId	The item identifier.
keepHistory	True to keep the history; otherwise, false.

SetActorItemByName

Sets an actor item by name.

Syntax

```
ItemSystem.SetActorItemByName(ScriptHandle actorId, const char *name, bool keepHistory)
```

Parameter	Description
`actorId`	The actor identifier.
`name`	The name of the actor item.
`keepHistory`	True to keep the history; otherwise, false.

ScriptBind_Network

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists C++ network functions that you can call from Lua script.

Expose

Syntax

```
Network.Expose()
```

ScriptBind_UIAction

CryLua is deprecated and will be removed in a future version of Lumberyard.

CallFunction

Calls a function of the UI flash asset or the UIEventSystem.

Syntax

```
UIAction.CallFunction(elementName, instanceID, functionName, [arg1], [arg2], [...])
```

Parameter	Description
`elementName`	The UI element name as defined in the XML or `UIEventSystem` name as defined in a `.cpp` file.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances. If used on `UIEventSystem`, no instance ID is ignored.
`functionName`	Function or event name.
`args`	List of arguments (optional).

DisableAction

Disables the UI Action.

Syntax

```
UIAction.DisableAction(actionName)
```

Parameter	Description
`actionName`	UI Action name.

EnableAction

Enables the UI Action.

Syntax

```
UIAction.EnableAction(actionName)
```

Parameter	Description
`actionName`	UI Action name.

EndAction

Ends a UI Action. This can be only used within a `UIAction` Lua script!

Syntax

```
UIAction.EndAction(table, disable, arguments)
```

Parameter	Description
`table`	Must be `self`.
`disable`	If true, this action is disabled when it terminates.
`arguments`	The arguments to return from this action.

GetAlpha

Get move clip alpha value.

Syntax

```
UIAction.GetAlpha(elementName, instanceID, mcName)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.

Parameter	Description
instanceID	The ID of the instance (if an instance with the specified ID does not exist, it will be created). -1 specifies all instances.
mcName	The movie clip name as defined in the XML.

GetArray

Returns a table with values of the array.

Syntax

```
UIAction.GetArray(elementName, instanceID, arrayName)
```

Parameter	Description
elementName	The UI element name as defined in the XML.
instanceID	The ID of the instance (if an instance with the specified ID does not exist, it will be created). -1 specifies all instances.
arrayName	Array name as defined in the XML.

GetPos

Get movie clip position.

Syntax

```
UIAction.GetPos(elementName, instanceID, mcName)
```

Parameter	Description
elementName	The UI element name as defined in the XML.
instanceID	The ID of the instance (if an instance with the specified ID does not exist, it will be created). -1 specifies all instances.
mcName	The movie clip name as defined in the XML.

GetRotation

Get movie clip rotation.

Syntax

```
UIAction.GetRotation(elementName, instanceID, mcName)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). '-1' for all instances
`mcName`	The movie clip name as defined in the XML.

GetScale

Get movie clip scale.

Syntax

```
UIAction.GetScale(elementName, instanceID, mcName)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances.
`mcName`	The movie clip name as defined in the XML.

GetVariable

Gets a variable of the UI flash asset.

Syntax

```
UIAction.GetVariable(elementName, instanceID, varName)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances.
`varName`	Variable name as defined in the XML.

GotoAndPlay

Call `GotoAndPlay` on a movie clip.

Syntax

```
UIAction.GotoAndPlay(elementName, instanceID, mcName, frameNum)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances.
`mcName`	The movie clip name as defined in the XML.
`frameNum`	The frame number.

GotoAndPlayFrameName

Call `GotoAndPlay` on a movie clip by frame name.

Syntax

```
UIAction.GotoAndPlayFrameName(elementName, instanceID, mcName, frameName)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances.
`mcName`	The movie clip name as defined in the XML.
`frameName`	The name of the frame.

GotoAndStop

Call GotoAndStop on a movie clip.

Syntax

```
UIAction.GotoAndStop(elementName, instanceID, mcName, frameNum)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances.
`mcName`	The movie clip name as defined in the XML.
`frameNum`	The frame number.

GotoAndStopFrameName

Call GotoAndStop on a movie clip by frame name.

Syntax

```
UIAction.GotoAndStopFrameName(elementName, instanceID, mcName, frameName)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances.
`mcName`	The movie clip name as defined in the XML.
`frameName`	The name of the frame.

HideElement

Hide the UI flash asset.

Syntax

```
UIAction.HideElement(elementName, instanceID)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances.

IsVisible

Get movie clip visible state.

Syntax

```
UIAction.IsVisible(elementName, instanceID, mcName)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances.

Parameter	Description
`mcName`	The movie clip name as defined in the XML.

RegisterActionListener

Register a callback function for a `UIAction` event. The callback function must have form: `CallbackName(actionName, eventName, argTable)`

Syntax

```
UIAction.RegisterActionListener(table, actionName, eventName, callbackFunctionName)
```

Parameter	Description
`table`	The script that receives the callback (can be `self` to refer the current script).
`actionName`	The UI action name.
`eventName`	The name of the event that is fired from the UI action (can be `OnStart` or `OnEnd`) **Warning** If an empty string is specified, all events will be received.
`callbackFunctionName`	The name of the script function that will receive the callback.

RegisterElementListener

Register a callback function for a `UIElement` event. The callback function must have form: `CallbackName(elementName, instanceId, eventName, argTable)`

Syntax

```
UIAction.RegisterElementListener(table, elementName, instanceID, eventName,
 callbackFunctionName)
```

Parameter	Description
`table`	The script that receives the callback (can be `self` to refer the current script).
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances.
`eventName`	The name of the event that is fired from the UI element. If an empty string is specified, all events will be received.

Parameter	Description
`callbackFunctionName`	name of the script function that will receive the callback.

RegisterEventSystemListener

Register a callback function for a `UIEventSystem` event. The callback function must have form: `CallbackName(actionName, eventName, argTable)`

Syntax

```
UIAction.RegisterEventSystemListener(table, eventSystem, eventName, callbackFunctionName)
```

Parameter	Description
`table`	The script that receives the callback (can be `self` to refer the current script).
`eventSystem`	The UI event system name.
`eventName`	The name of the event that is fired from `UIEventSystem`. If an empty string is specified, all events will be received.
`callbackFunctionName`	name of the script function that will receive the callback.

ReloadElement

Reloads the UI flash asset.

Syntax

```
UIAction.ReloadElement(elementName, instanceID)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances.

RequestHide

Send the fade out signal to the UI flash asset.

Syntax

```
UIAction.RequestHide(elementName, instanceID)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances.

SetAlpha

Set movie clip alpha value.

Syntax

```
UIAction.SetAlpha(elementName, instanceID, mcName, fAlpha)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances.
`mcName`	The movie clip name as defined in the XML.
`fAlpha`	Alpha value (0-1).

SetArray

Sets an array of the UI flash asset.

Syntax

```
UIAction.SetArray(elementName, instanceID, arrayName, values)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances.
`arrayName`	The array name as defined in the XML.
`values`	Table of values for the array.

SetPos

Set movie clip position.

Syntax

```
UIAction.SetPos(elementName, instanceID, mcName, vPos)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances.
`mcName`	The movie clip name as defined in the XML.
`vPos`	position.

SetRotation

Set movie clip rotation.

Syntax

```
UIAction.SetRotation(elementName, instanceID, mcName, vRotation)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances.
`mcName`	The movie clip name as defined in the XML.
`vRotation`	The rotation.

SetScale

Set movie clip scale.

Syntax

```
UIAction.SetScale(elementName, instanceID, mcName, vScale)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances.
`mcName`	The movie clip name as defined in the XML.

Parameter	Description
vScale	scale.

SetVariable

Sets a variable of the UI flash asset.

Syntax

```
UIAction.SetVariable(elementName, instanceID, varName, value)
```

Parameter	Description
elementName	The UI element name as defined in the XML.
instanceID	The ID of the instance (if an instance with the specified ID does not exist, it will be created). -1 specifies all instances.
varName	Variable name as defined in the XML.
value	Value to set.

SetVisible

Set movie clip visible state.

Syntax

```
UIAction.SetVisible(elementName, instanceID, mcName, bVisible)
```

Parameter	Description
elementName	The UI element name as defined in the XML.
instanceID	The ID of the instance (if an instance with the specified ID does not exist, it will be created). -1 specifies all instances.
mcName	The movie clip name as defined in the XML.
bVisible	visible.

ShowElement

Displays the UI flash asset.

Syntax

```
UIAction.ShowElement(elementName, instanceID)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances.

StartAction

Starts a UI Action.

Syntax

```
UIAction.StartAction(actionName, arguments)
```

Parameter	Description
`actionName`	UI Action name.
`arguments`	The arguments to pass to this action.

UnloadElement

Unloads the UI flash asset.

Syntax

```
UIAction.UnloadElement(elementName, instanceID)
```

Parameter	Description
`elementName`	The UI element name as defined in the XML.
`instanceID`	The ID of the instance (if an instance with the specified ID does not exist, it will be created). `-1` specifies all instances.

UnregisterActionListener

Unregister callback functions for a UIAction event.

Syntax

```
UIAction.UnregisterActionListener(table, callbackFunctionName)
```

Parameter	Description
`table`	The script that receives the callback (can be `self` to refer the current script).

Parameter	Description
`callbackFunctionName`	The name of the script function that receives the callback. If "" is specified, all callbacks for this script will be removed.

UnregisterElementListener

Unregister callback functions for a UIElement event.

Syntax

```
UIAction.UnregisterElementListener(table, callbackFunctionName)
```

Parameter	Description
`table`	The script that receives the callback (can be `self` to refer the current script).
`callbackFunctionName`	The name of the script function that receives the callback. If "" is specified, all callbacks for this script will be removed.

UnregisterEventSystemListener

Unregister callback functions for a UIEventSystem event.

Syntax

```
UIAction.UnregisterEventSystemListener(table, callbackFunctionName)
```

Parameter	Description
`table`	The script that receives the callback (can be `self` to refer the current script).
`callbackFunctionName`	The name of the script function that receives the callback. If "" is specfied, all callbacks for this script will be removed.

ScriptBind_Vehicle

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists C++ vehicle system functions that you can call from Lua script.

AddSeat

Adds a seat to the vehicle.

Syntax

```
Vehicle.AddSeat(SmartScriptTable paramsTable)
```

Parameter	Description
`paramsTable`	The seat parameters in `SmartScriptTable` format.

ChangeSeat

Makes the actor change the seat inside the vehicle.

Syntax

```
Vehicle.ChangeSeat(ScriptHandle actorHandle, int seatId, bool isAnimationEnabled)
```

Parameter	Description
`actorHandle`	The actor identifier.
`seatId`	The seat identifier.
`isAnimationEnabled`	True if animation is enabled; otherwise, false.

Destroy

Destroys the vehicle.

Syntax

```
Vehicle.Destroy()
```

DisableEngine

Disables or enables the engine of the vehicle.

Syntax

```
Vehicle.DisableEngine(bool disable)
```

Parameter	Description
`disable`	True to disable the engine; false to enable.

EnableMovement

Enables or disables the movement of the vehicle.

Syntax

```
Vehicle.EnableMovement(bool enable)
```

Parameter	Description
enable	True to enable movement; false to disable.

EnterVehicle

Makes the specified actor enter the vehicle.

Syntax

```
Vehicle.EnterVehicle(ScriptHandle actorHandle, int seatId, bool isAnimationEnabled)
```

Parameter	Description
actorHandle	The actor identifier.
seatId	The seat identifier.
isAnimationEnabled	True if animation is enabled; otherwise, false.

ExitVehicle

Makes the actor leave the vehicle.

Syntax

```
Vehicle.ExitVehicle(ScriptHandle actorHandle)
```

Parameter	Description
actorHandle	The actor identifier.

GetComponentDamageRatio

Gets the damage ratio of the specified component.

Syntax

```
Vehicle.GetComponentDamageRatio(const char* pComponentName)
```

Parameter	Description
pComponentName	The name of the component.

GetHelperDir

Gets the helper direction.

Syntax

```
Vehicle.GetHelperDir(const char* name, bool isInVehicleSpace)
```

Parameter	Description
`name`	The name of the helper.
`isInVehicleSpace`	True if the helper is in the vehicle space; otherwise, false.

GetHelperPos

Gets the helper position.

Syntax

```
Vehicle.GetHelperPos(const char* name, bool isInVehicleSpace)
```

Parameter	Description
`name`	The name of the helper.
`isInVehicleSpace`	True if the helper is in the vehicle space; otherwise, false.

GetHelperWorldPos

Gets the helper position in the world coordinates.

Syntax

```
Vehicle.GetHelperWorldPos(const char* name)
```

Parameter	Description
`name`	The name of the helper.

GetSeatForPassenger

Returns a vehicle seat ID for the specified passenger.

Syntax

```
Vehicle.GetSeatForPassenger(ScriptHandle passengerId)
```

Parameter	Description
`passengerId`	The passenger ID.

GetVehicle

Gets the vehicle identifier.

Syntax

```
Vehicle.GetVehicle()
```

HasHelper

Checks if the vehicle has the specified helper.

Syntax

```
Vehicle.HasHelper(const char* name)
```

Parameter	Description
`name`	The name of the helper.

IsDestroyed

Checks if the vehicle is destroyed.

Syntax

```
Vehicle.IsDestroyed()
```

IsInsideRadius

Checks if the vehicle is inside the specified radius.

Syntax

```
Vehicle.IsInsideRadius(Vec3 pos, float radius)
```

Parameter	Description
`pos`	The {x,y,z} position vector.
`radius`	The radius, expressed as a float.

IsUsable

Checks if the vehicle is usable by the user.

Syntax

```
Vehicle.IsUsable(ScriptHandle userHandle)
```

Parameter	Description
`userHandle`	The user identifier.

MultiplyWithWorldTM

Multiplies with the world transformation matrix.

Syntax

```
Vehicle.MultiplyWithWorldTM(Vec3 pos)
```

Parameter	Description
`pos`	The {x,y,z} position vector.

OnHit

Triggers an event that occurs after the vehicle is hit.

Syntax

```
Vehicle.OnHit(ScriptHandle targetId, ScriptHandle shooterId, float damage, Vec3 position,
 float radius, int hitTypeId, bool explosion)
```

Parameter	Description
`targetId`	The target identifier.
`shooterId`	The shooter identifier.
`damage`	The amount of damage, expressed as a float.
`position`	The {x,y,z} position vector.
`radius`	Radius of the hit, expressed as a float.
`hitTypeId`	The type of damage, expressed as an integer.
`explosion`	True if the hit causes an explosion, otherwise false.

OnSpawnComplete

Calls back into the game code for when vehicle spawn has been completed.

Syntax

```
Vehicle.OnSpawnComplete()
```

OnUsed

Triggers an event when the user uses the specified vehicle.

Syntax

```
Vehicle.OnUsed(ScriptHandle userHandle, int index)
```

Parameter	Description
`userHandle`	The user identifier.
`index`	The seat identifier.

ProcessPassengerDamage

Processes passenger damages.

Syntax

```
Vehicle.ProcessPassengerDamage(ScriptHandle passengerId, float actorHealth, float damage,
 int hitTypeId, bool explosion)
```

Parameter	Description
`passengerId`	The passenger identifier.
`actorHealth`	The health of the actor.
`damage`	The amount of damage.
`hitTypeId`	The type of damage.
`explosion`	True if there is an explosion; otherwise, false.

Reset

Resets the vehicle.

Syntax

```
Vehicle.Reset()
```

ResetSlotGeometry

Syntax

```
Vehicle.ResetSlotGeometry(int slot, const char* filename, const char* geometry)
```

Parameter	Description
`slot`	The number of the slot.
`filename`	The filename.
`geometry`	The slot geometry.

ScriptBind_VehicleSeat

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists C++ vehicle seat functions that you can call from Lua script.

GetPassengerId

Gets the passenger identifier.

Syntax

```
VehicleSeat.GetPassengerId()
```

GetVehicleSeat

Gets the vehicle seat identifier.

Syntax

```
VehicleSeat.GetVehicleSeat()
```

GetWeaponCount

Gets the number of weapons available on this seat.

Syntax

```
VehicleSeat.GetWeaponCount()
```

GetWeaponId

Gets the weapon identifier.

Syntax

```
VehicleSeat.GetWeaponId(int weaponIndex)
```

Parameter	Description
`weaponIndex`	Weapon identifier.

IsDriver

Checks if the seat is the driver seat.

Syntax

```
VehicleSeat.IsDriver()
```

IsFree

Checks if the seat is free.

Syntax

```
VehicleSeat.IsFree(ScriptHandle actorHandle)
```

Parameter	Description
`actorHandle`	Passenger identifier.

IsGunner

Checks if the seat is the gunner seat.

Syntax

```
VehicleSeat.IsGunner()
```

Reset

Resets the vehicle seat.

Syntax

```
VehicleSeat.Reset()
```

SetAIWeapon

Sets the weapon artificial intelligence.

Syntax

```
VehicleSeat.SetAIWeapon(ScriptHandle weaponHandle)
```

Parameter	Description
`weaponHandle`	Weapon identifier.

ScriptBind_VehicleSystem

CryLua is deprecated and will be removed in a future version of Lumberyard.

Lists C++ vehicle system functions that you can call from Lua script.

GetOptionalScript

Get an (optional) script for the named vehicle.

Syntax

```
VehicleSystem.GetOptionalScript(char* vehicleName)
```

GetVehicleImplementations

Get a table of all implemented vehicles.

Syntax

```
VehicleSystem.GetVehicleImplementations()
```

ReloadSystem

Reloads the vehicle system with default values.

Syntax

```
VehicleSystem.ReloadSystem()
```

SetTpvDistance

Distance of camera in third person view.

Syntax

```
VehicleSystem.SetTpvDistance(float distance)
```

SetTpvHeight

Height of camera in third person view.

Syntax

```
VehicleSystem.SetTpvHeight(float height)
```

ScriptBind_Boids

CryLua is deprecated and will be removed in a future version of Lumberyard.

These functions create simulated flocks of bird-like objects (boids) or other animals and control their behavior.

CanPickup

Syntax

Checks if the boid can be picked up.

```
Boids.CanPickup(flockEntity, boidEntity)
```

Parameter	Description
`flockEntity`	Valid entity table containing flock.
`boidEntity`	Valid entity table containing boid.

CreateBugsFlock

Creates a bugs flock and binds it to the given entity.

Syntax

```
Boids.CreateBugsFlock(entity,paramsTable)
```

Parameter	Description
`entity`	Valid entity table.
`paramTable`	Table with parameters for flock (see sample scripts).

CreateFishFlock

Creates a fish flock and binds it to the given entity.

Syntax

```
Boids.CreateFishFlock(entity,paramsTable)
```

Parameter	Description
`entity`	Valid entity table.
`paramTable`	Table with parameters for flock (see sample scripts).

CreateFlock

Creates a flock of boids and binds it to the given entity.

Syntax

```
Boids.CreateFlock(entity,paramsTable)
```

Parameter	Description
`entity`	Valid entity table.
`nType`	The type of flock. Possible values are `Boids.FLOCK_BIRDS`, `Boids.FLOCK_FISH`, or `Boids.FLOCK_BUGS`.

Parameter	Description
paramTable	Table with parameters for flock (see sample scripts).

EnableFlock

Enables or disables a flock in the entity.

Syntax

```
Boids.EnableFlock(entity,paramsTable)
```

Parameter	Description
entity	Valid entity table containing flock.
bEnable	Specify true to enable the flock; false to disable.

GetUsableMessage

Gets the appropriate localized UI message for the specified flock.

Syntax

```
Boids.GetUsableMessage(flockEntity)
```

Parameter	Description
flockEntity	Valid entity table containing flock.

OnBoidHit

Event that occurs on boid hit.

Syntax

```
Boids.OnBoidHit(flockEntity,boidEntity,hit)
```

Parameter	Description
flockEntity	Valid entity table containing flock.
boidEntity	Valid entity table containing boid.
hit	Valid entity table containing hit information.

OnPickup

Forwards the appropriate pickup action to the boid object.

Syntax

```
Boids.OnPickup(flockEntity, boidEntity, bPickup, fThrowSpeed)
```

Parameter	Description
`flockEntity`	Valid entity table containing flock.
`boidEntity`	Valid entity table containing boid.
`bPickup`	Pickup, or drop or throw.
`fThrowSpeed`	Specifies the throw speed. By default, a value greater than 5.f kills the boid. This has no effect on the pickup action.

SetAttractionPoint

Sets the one time attraction point for the boids.

Syntax

```
Boids.SetAttractionPoint(entity,paramsTable)
```

Parameter	Description
`entity`	Valid entity table containing flock.
`point`	The one time attraction point.

SetFlockParams

Sets the parameters of the flock for the specified entity.

Syntax

```
Boids.SetFlockParams(entity, paramsTable)
```

Parameter	Description
`entity`	Valid entity table containing flock.
`paramTable`	Table with parameters for flock (see sample scripts).

SetFlockPercentEnabled

Specifies the percentage of boid objects that are rendered in flocks. You can use this to enable flocks gradually.

Syntax

```
Boids.SetFlockPercentEnabled(entity, paramsTable)
```

Parameter	Description
`entity`	Valid entity table containing flock.
`nPercent`	Possible values are from 0 through 100. If 0, no boids are rendered; if 100, all boids are rendered.

Integrating Lua and C++

CryLua is deprecated and will be removed in a future version of Lumberyard.

The CryScript system abstracts a Lua virtual machine for use by the other systems and the game code. It includes the following functionality:

- calling script functions
- exposing C++-based variables and functions to scripts
- creating script tables stored in virtual machine memory

The CryScript system is based on Lua 5. More information on the Lua language can be found at http://www.lua.org.

Accessing Script Tables

A global script table can be retrieved by calling `IScriptSystem::GetGlobalValue()`. The IScriptTable is used to represent all script tables/variables.

Exposing C++ Functions and Values

To expose C++ functions and variables to scripts, you'll need to implement a new class. The easiest way is to derive the `CScriptableBase` class, which provides most of the functionality.

Exposing Constants

To expose constant values to scripts, use the `IScriptSystem::SetGlobalValue()`. For example, to expose a constant named MTL_LAYER_FROZEN to our scripts, use the following code:

```
gEnv->pScriptSystem->SetGlobalValue("MTL_LAYER_FROZEN", MTL_LAYER_FROZEN);
```

Exposing Functions

To expose C++ functions to scripts, implement a new class derives from `CScriptableBase`, as shown in the following example.

```
classCScriptBind_Game :
    publicCScriptableBase
{
```

```
public:
    CScriptBind_Game( ISystem* pSystem );
    virtual ~CScriptBind_Game() {}

    intGameLog(IFunctionHandler* pH, char* pText);
};
```

Add the following code inside the class constructor:

```
Init(pSystem->GetIScriptSystem(), pSystem);
SetGlobalName("Game");

#undef SCRIPT_REG_CLASSNAME
#define SCRIPT_REG_CLASSNAME &CScriptBind_Game::

SCRIPT_REG_TEMPLFUNC(GameLog, "text");
```

In a Lua script, you can access your new ScriptBind function as follows:

```
Game.GameLog("a message")
```

Networking System

GridMate is Lumberyard's networking subsystem. GridMate is designed for efficient bandwidth usage and low-latency communications. You can synchronize objects over the network with GridMate's replica framework. GridMate's session management integrates with major online console services and lets you handle peer-to-peer and client-server topologies with host migration. GridMate also supports in-game achievements, leaderboards, and cloud-based saved games through third-party social services such as Xbox Live, PlayStation Network, and Steam. For an example of how to set up a multiplayer project, see Multiplayer Sample in the Amazon Lumberyard User Guide.

This section discusses the various components of, and setup requirements for, your Amazon Lumberyard game networking environment. For information about a diagnostic tool for networking, see Profiler (p. 1038).

Topics

- Tutorial: Getting Started with Multiplayer (p. 951)
- Overview (p. 963)
- Using Lumberyard Networking (p. 996)
- CryNetwork Backward Compatibility (p. 1014)

Tutorial: Getting Started with Multiplayer

This tutorial walks you through the steps to create a simple multiplayer game test level. These steps include binding an entity to the network and connecting a client to the host. At the end of the tutorial, you should have a level with a simple network bound entity that is ready for a multiplayer game.

This tutorial guides you through the following tasks:

- Create a level and add in new entities.
- Bind an entity's transform component to the network.
- Connect a client to the server and verify network replication.

Prerequisites

This tutorial assumes the following:

- You have installed Amazon Lumberyard.
- You have created a game project.
- Your game project has the **Multiplayer** gem and the **User Login: Default** gem enabled. You can enable these gems in Lumberyard's Project Configurator.

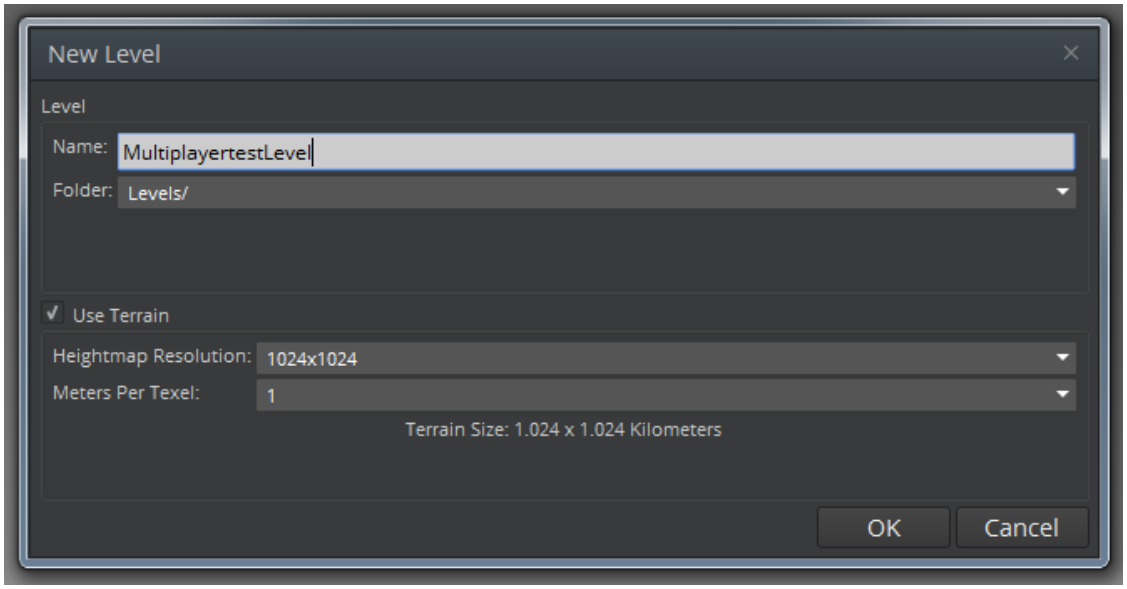
> **Note**
> This tutorial uses Visual Studio 2013, but you can also use Visual Studio 2015.

Step 1: Creating a Level and Adding a Sphere and a Box

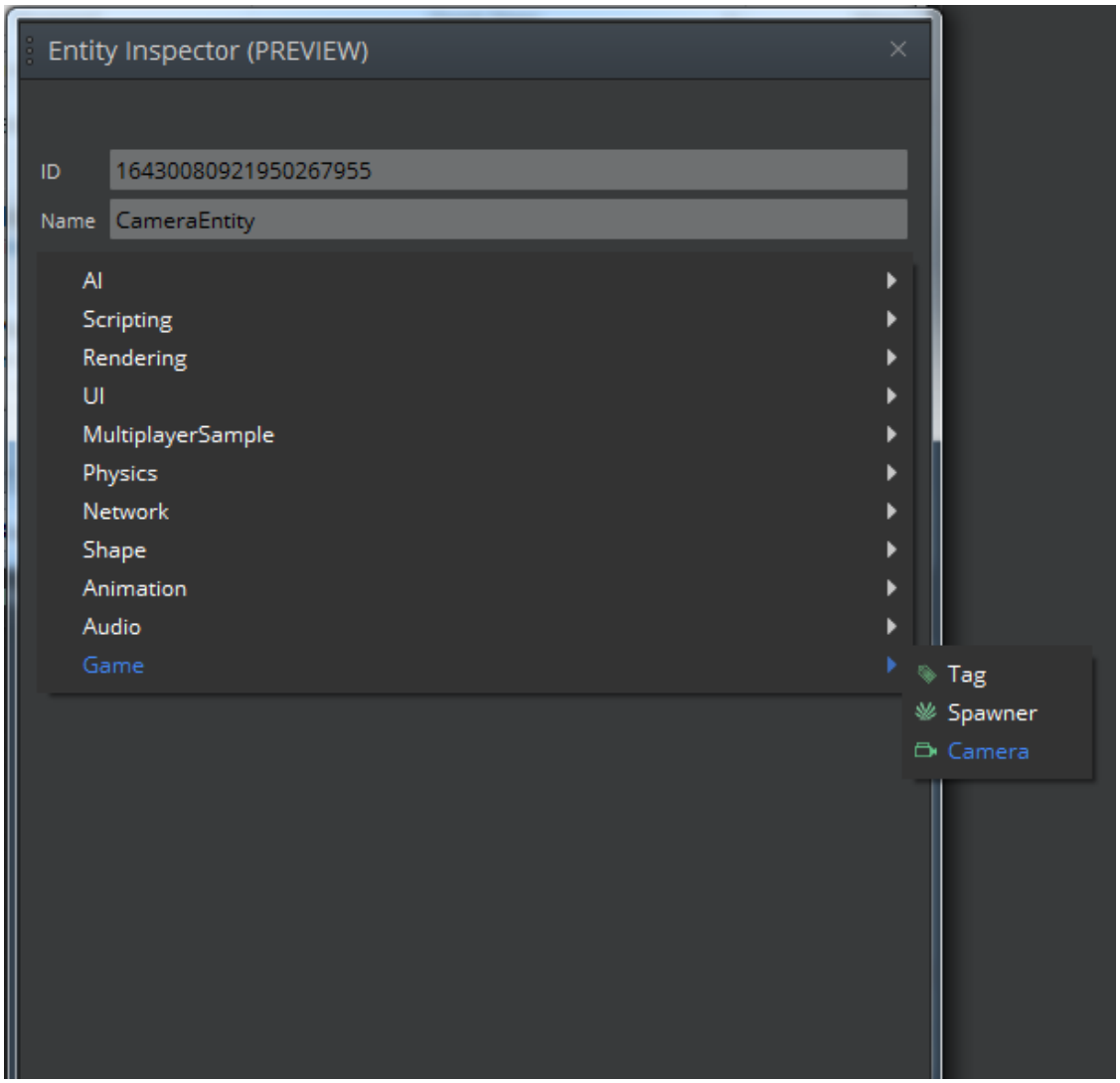
Your first step is to create a level and prepare a simple sphere and box shape so that you can test Lumberyard's networking features.

To create a level, sphere, and box

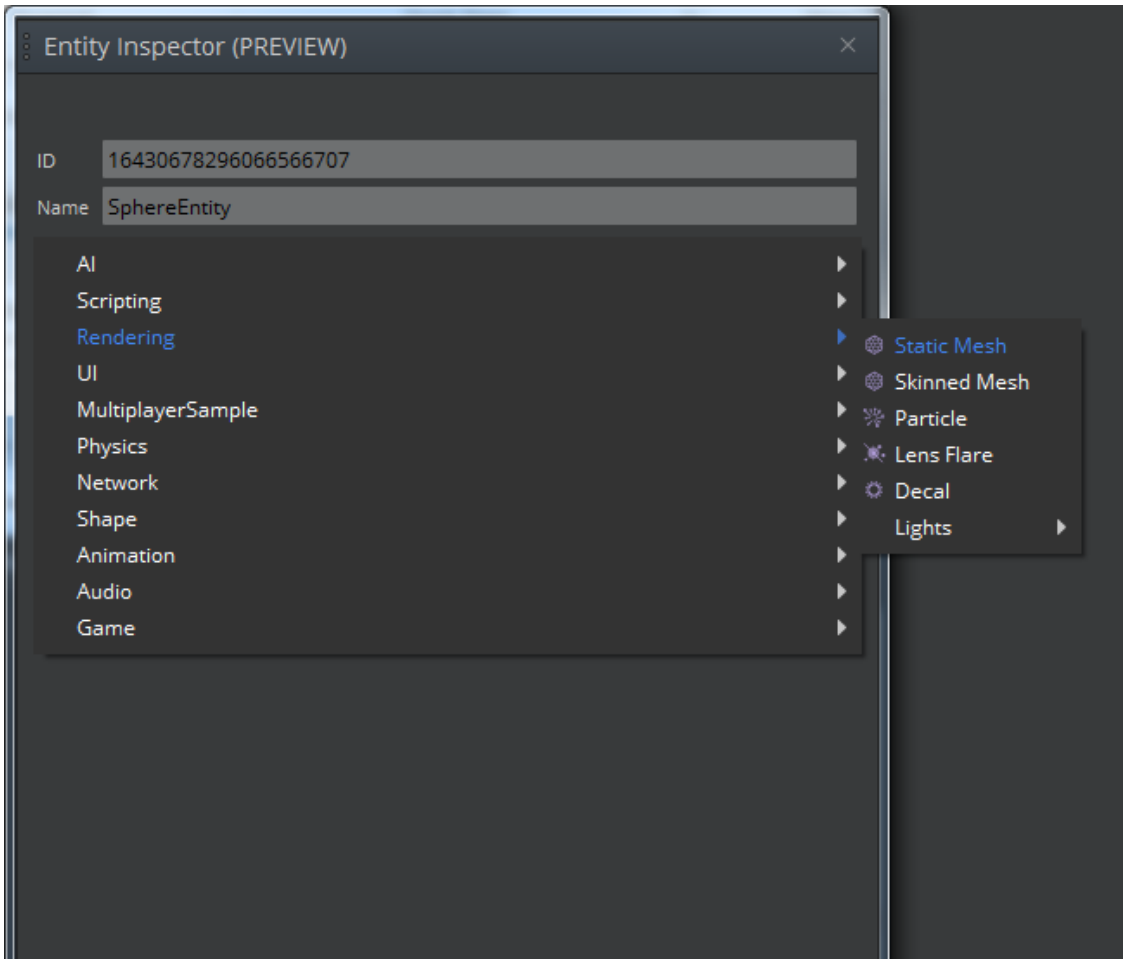
1. In the Lumberyard Project Configurator, choose a project that has the Multiplayer Gem enabled, and then click **Set as default**.
2. Open Lumberyard Editor, create a level, and give it a name.

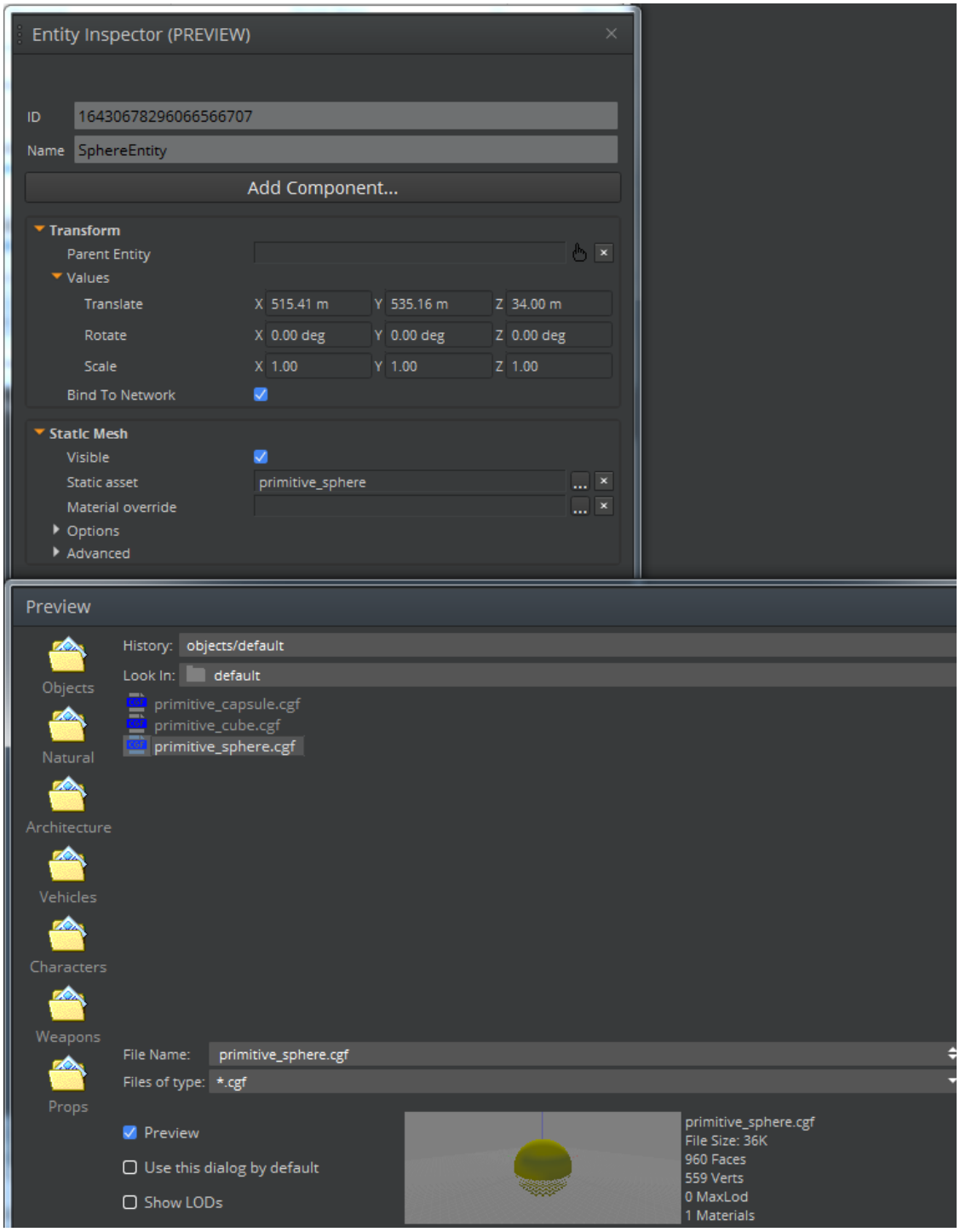


3. In the Lumberyard Editor viewport, right-click and select **Create new component entity**.
4. With the entity selected, use the **Entity Inspector** to name the entity **CameraEntity**.
5. Click **Add Component**.
6. Select **Game**, **Camera** to assign a camera component to the entity.



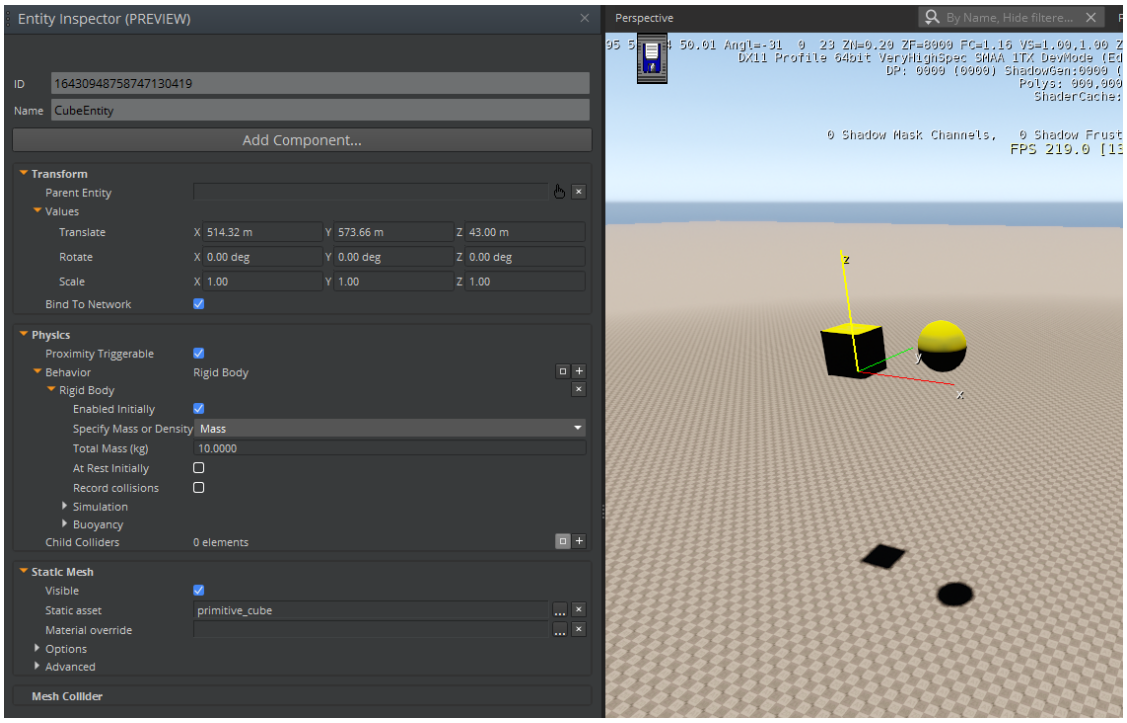
7. In the Lumberyard Editor viewport, right-click and select **Create new component entity**.
8. With the entity selected, use the **Entity Inspector** to name the entity **SphereEntity**.
9. Click **Add Component**, **Shape**, **Sphere Shape**.
10. In **Entity Inspector**, click **Add Component**, **Rendering**, **Static Mesh** to assign a static mesh component to **SphereEntity**.
11. In **Entity Inspector**, under **Static Mesh**, click the **...** next to **Static asset**.
12. In the **Preview** window, click **Objects**, **default**, and choose **primitive_sphere.cgf**.
13. Click **Open**.
14. In the Lumberyard Editor viewport, right-click and select **Create new component entity**.
15. With the entity selected, use the **Entity Inspector** to name the entity **BoxEntity**.
16. Click **Add Component**, **Shape**, **Box Shape**.
17. In **Entity Inspector**, click **Add Component**, **Rendering**, **Static Mesh** to assign a static mesh component to **BoxEntity**.
18. In **Entity Inspector**, under **Static Mesh**, click the **...** next to **Static asset**.
19. In the **Preview** window, click **Objects**, **default**, and choose **primitive_cube.cgf**.
20. Click **Open**.


Entity Inspector (PREVIEW)
ID
16430678296066566707
Name
SphereEntity
AI
Scripting
Rendering
UI
MultiplayerSample
Physics
Network
Shape
Animation
Audio
Game
Static Mesh
Skinned Mesh
Particle
Lens Flare
Decal
Lights



21. Select the **SphereEntity**. In **Entity Inspector**, click **Add Component**, **Physics**, **Physics**.
22. In **Entity Inspector**, under **Physics**, click the **+** next to **Behavior**.
23. In the **Class to create** dialog box, choose **Rigid Body** to attach rigid body physics to the component.
24. Select the **BoxEntity**, and follow the same steps to attach rigid body physics to it.
25. In the viewport, move the sphere and box entities above the plane so that they have room to fall.

26. In **Entity Inspector**, set **Physics, At Rest Initially** to **false** to allow the sphere and box to begin simulating after the level is loaded.



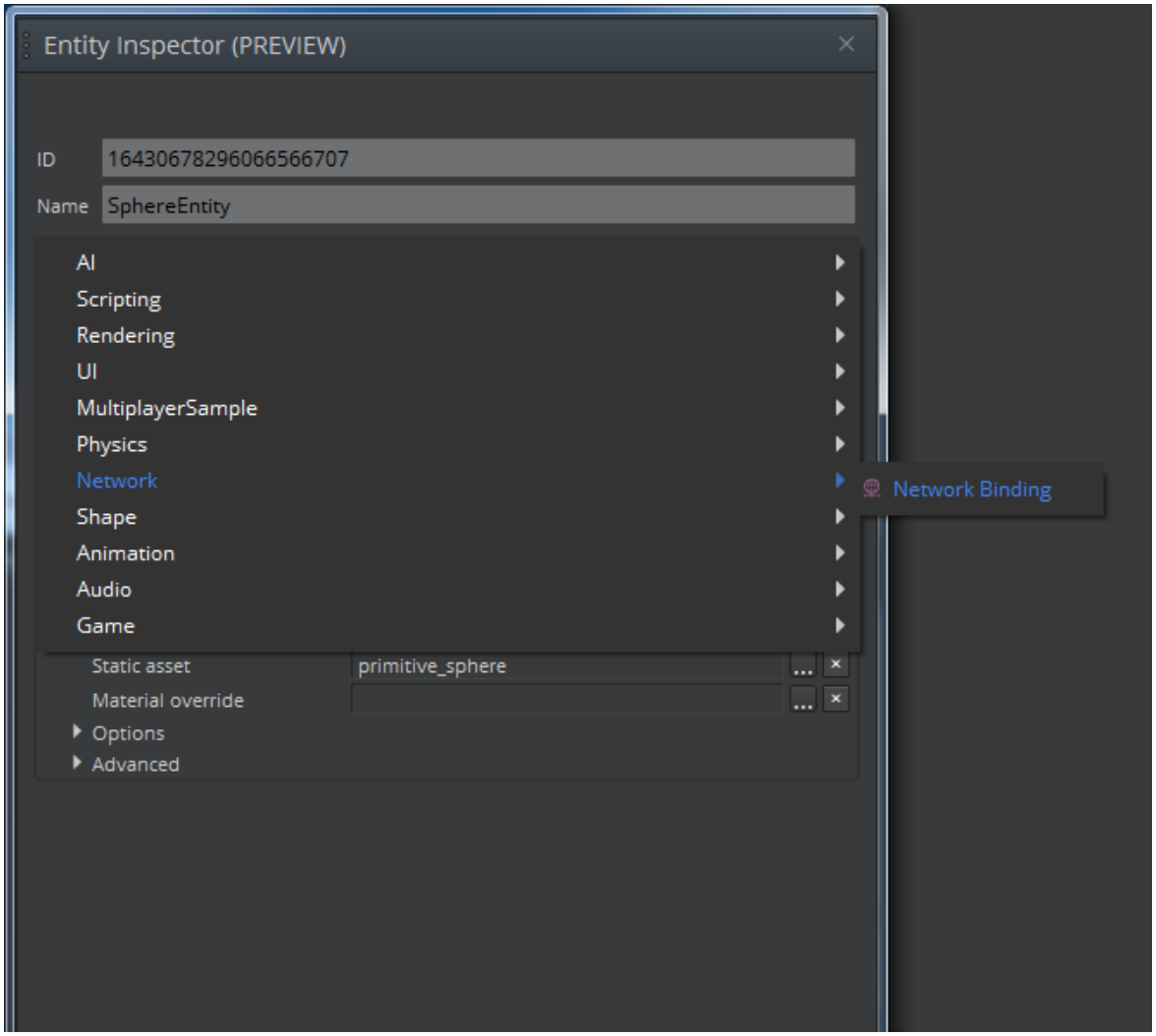
You now have two simple component entities with physics in the level that you created.

Step 2: Binding Sphere Transform Components to the Network

After you create the initial level with the sphere and the box, you bind the sphere's transform component to the network. This allows clients to replicate the sphere and see changes over the network.

To bind the sphere's transform to the network

- Select the sphere entity. In **Entity Inspector**, click **Add Component**, **Network**, **Network Binding** to add a `NetBinding` component to it. This allows the host to replicate the transform component of the sphere to all clients.



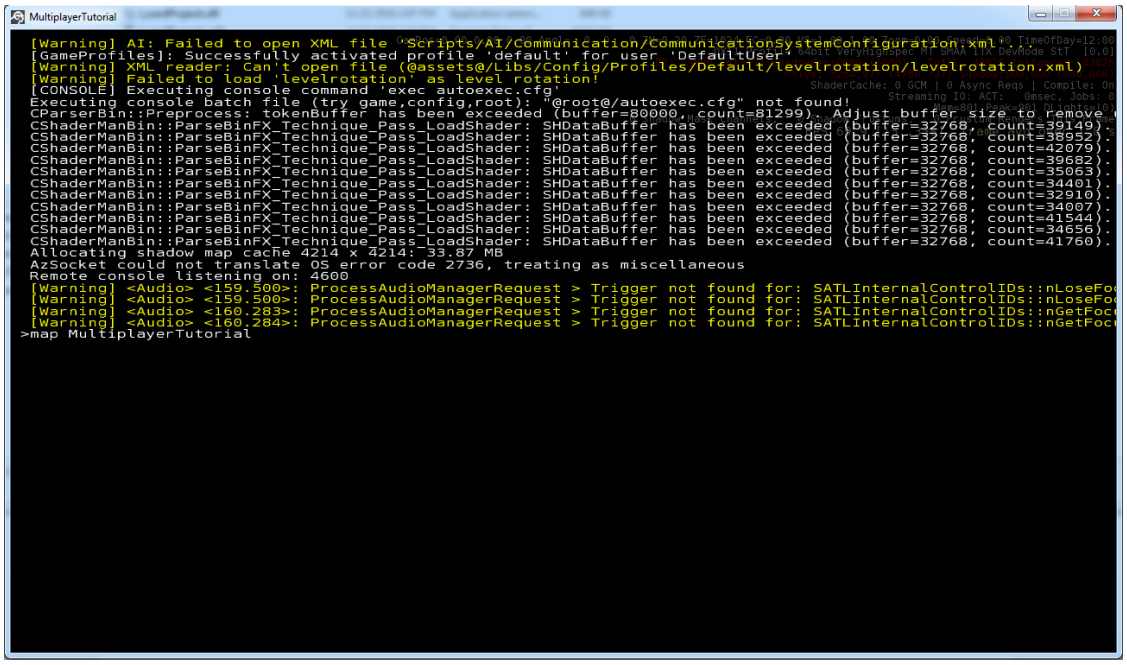
You have now created a server authoritative sphere entity.

Step 3: Connecting a Client to the Server

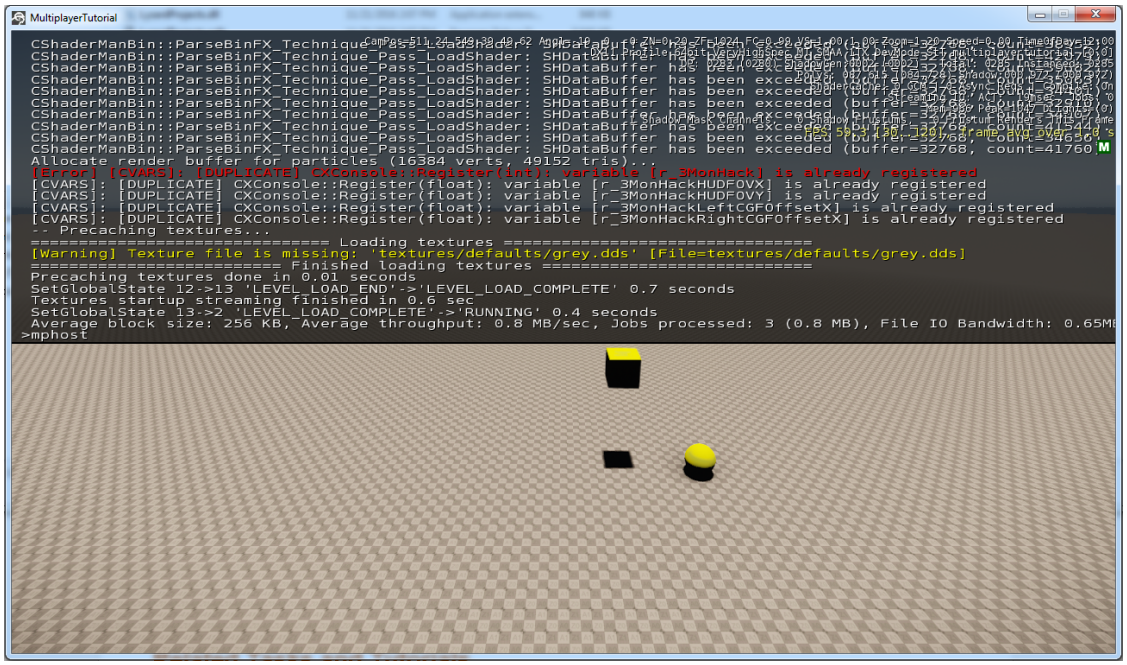
This step shows you how to connect a client to the server instance and then observe your networked sphere in action.

To connect a client game to the host game

1. Choose **File**, **Export to Engine**, or press **Ctrl+E** to export your level.
2. Run the game launcher from the `Bin` directory that you are using. If you are using Visual Studio 2013, the directory is `\dev\Bin64vc120\`. If you are using Visual Studio 2015, the path is `\dev\Bin64vc140\`. The name of your launcher is *<your-game-project-name>*`WindowsLauncher.exe`.
3. Press the ` key (above the **TAB** key) to open the console.
4. Run the command `map` *<MultiplayerTutorial>* where *<MultiplayerTutorial>* is the name of the level to load.



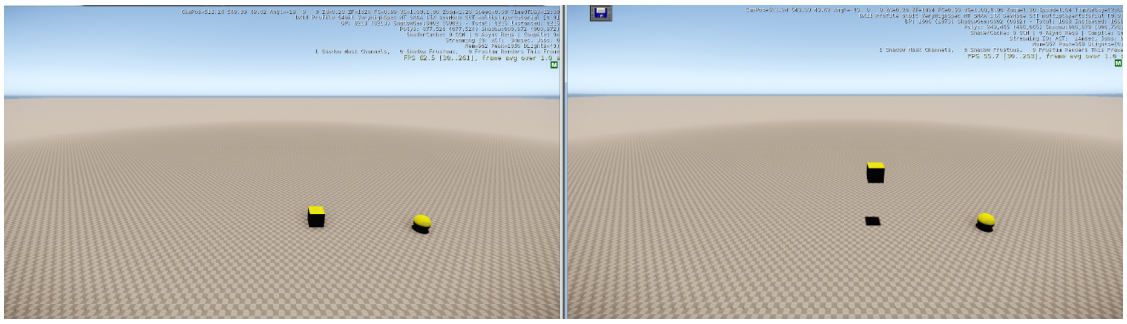
5. Press the ` key to open the console. Run the command `mphost` to make your client host a network session.



6. Use the launcher to open another game and press the ` key to open the console.
7. Run the command **sv_port 0** to set the client port to **0** and the ephemeral port.

   Note
   On a single computer, only one process is allowed to bind to a particular port. Therefore, to run more than one game process on the same computer (as in this multiplayer sample), you must use ephemeral ports. The `sv_port` CVar defines the UDP port on the local machine for the multiplayer sample, and the setting of `0` specifies the ephemeral port. This allows two clients on the same computer to talk to each other.

8. Run the command `mpjoin` to join to the host game. You should see the sphere synchronized by location on the client. However, the box will be desynchronized and have different locations on the client and host.



Congratulations! You have successfully created a simple networked level. You can now use the Network Binding component to synchronize transforms of entities and connect clients to servers.

Related Tasks and Tutorials

You have created a simple networking sample to see the effects of networking in Lumberyard. Now visit the following links to see what else you can add to your game:

- Configuring the Multiplayer Sample for Amazon GameLift (p. 959)
- Tutorial: Overview of GameLift
- Tutorial: Overview of Cloud Canvas

Configuring the Multiplayer Sample for Amazon GameLift

To prepare the multiplayer sample for use with Amazon GameLift, follow the required procedures for server-side and client-side configuration.

Server-Side Configuration

On the server side, overwrite the `GridMate::OnSessionStarted()` handler. In the handler, synchronize the session state and load the corresponding map if the CVAR `sv_map` is set in the `Multiplayer::Utils::SynchronizeSessionState()` function.

The following example shows code for server-side configuration.

```
void GameManager::OnSessionCreated(GridMate::GridSession* session)
{
     m_gameSession = session;
     if (m_gameSession)
     {
          if (m_gameSession->IsHost())
          {
               if (gEnv->IsDedicated())
               {
                    Multiplayer::Utils::SynchronizeSessionState(m_gameSession);
               }
          }
```

```
    }
}
```

Client-Side Configuration

On the client side, you must configure the following CVARs:

`sv_port`

`sv_map`

`gamelift_aws_access_key`

`gamelift_aws_secret_key`

`gamelift_fleet_id` or `gamelift_alias_id`

`gamelift_end_point`

`gamelift_playerid`

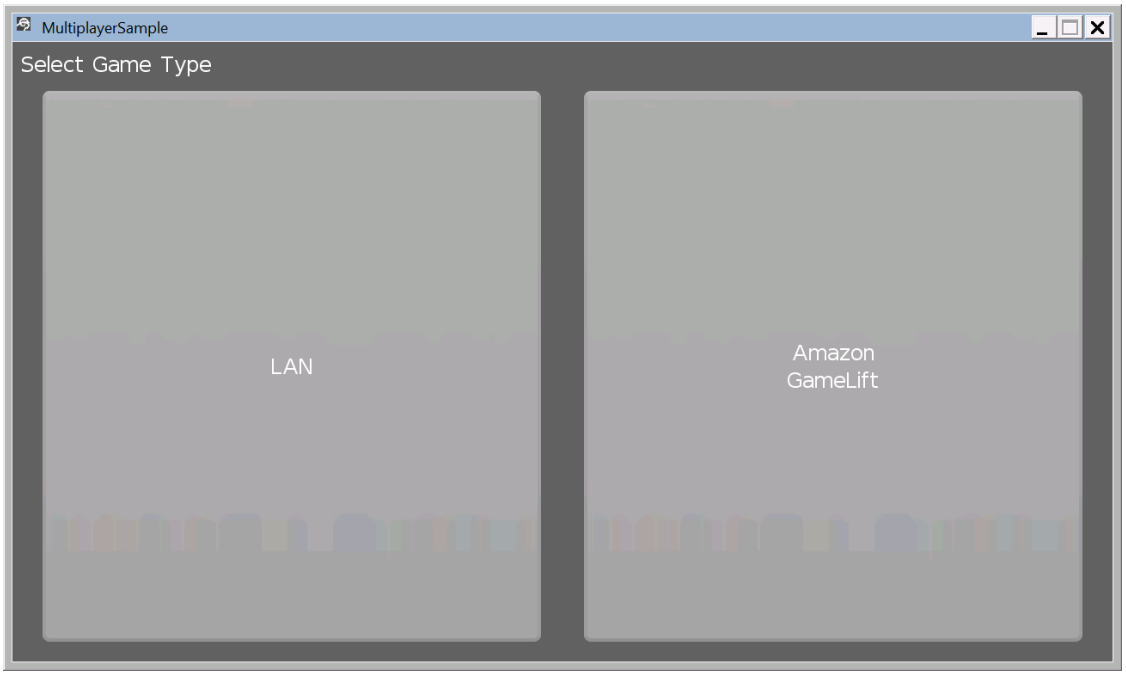
You can set these CVARs with a console command or with the multiplayer sample user interface.

To use CVARs to set the client side configuration, type the following console command.

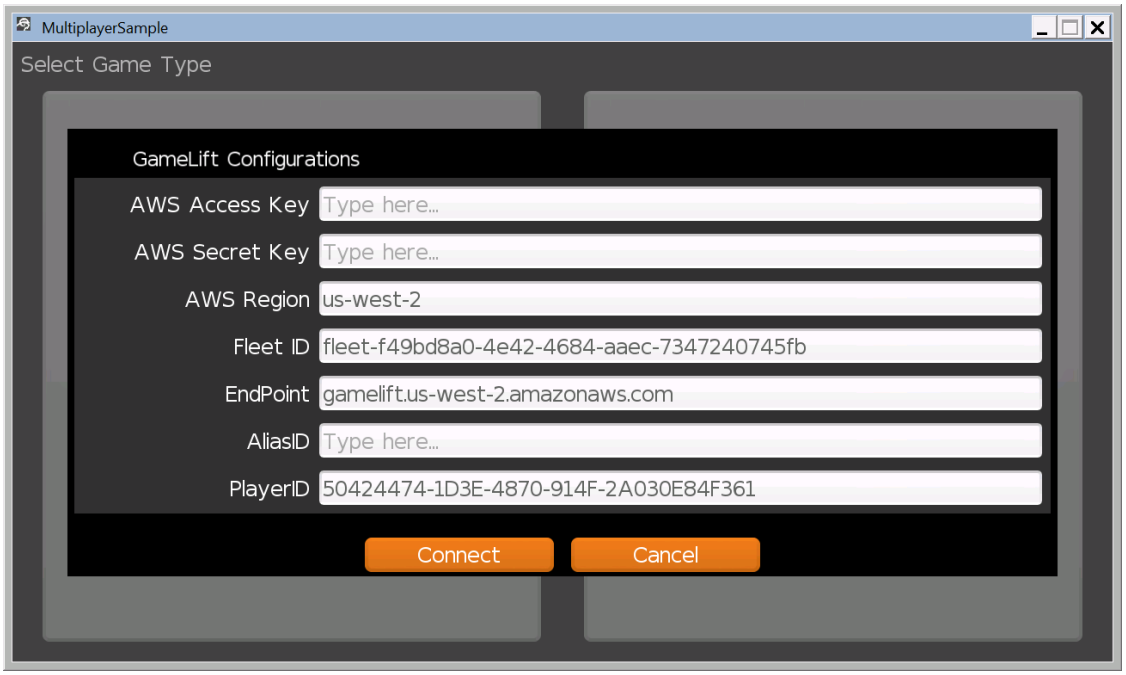
```
+sv_port 33435 +gamelift_fleet_id <fleet> +gamelift_aws_access_key <aws access key>
 +gamelift_aws_secret_key <aws secret key>
```

To use the multiplayer sample user interface to configure Amazon GameLift

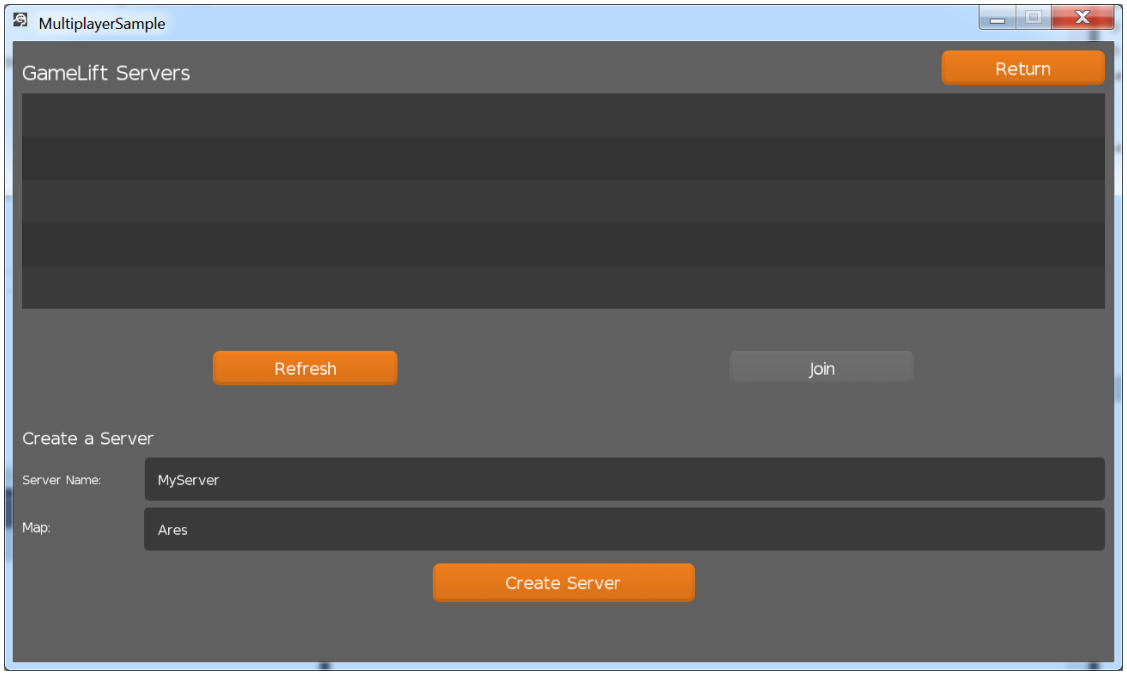
1. By default, the multiplayer sample loads the **Game Lobby** map. To add or modify the CVARs, click **Amazon GameLift**.



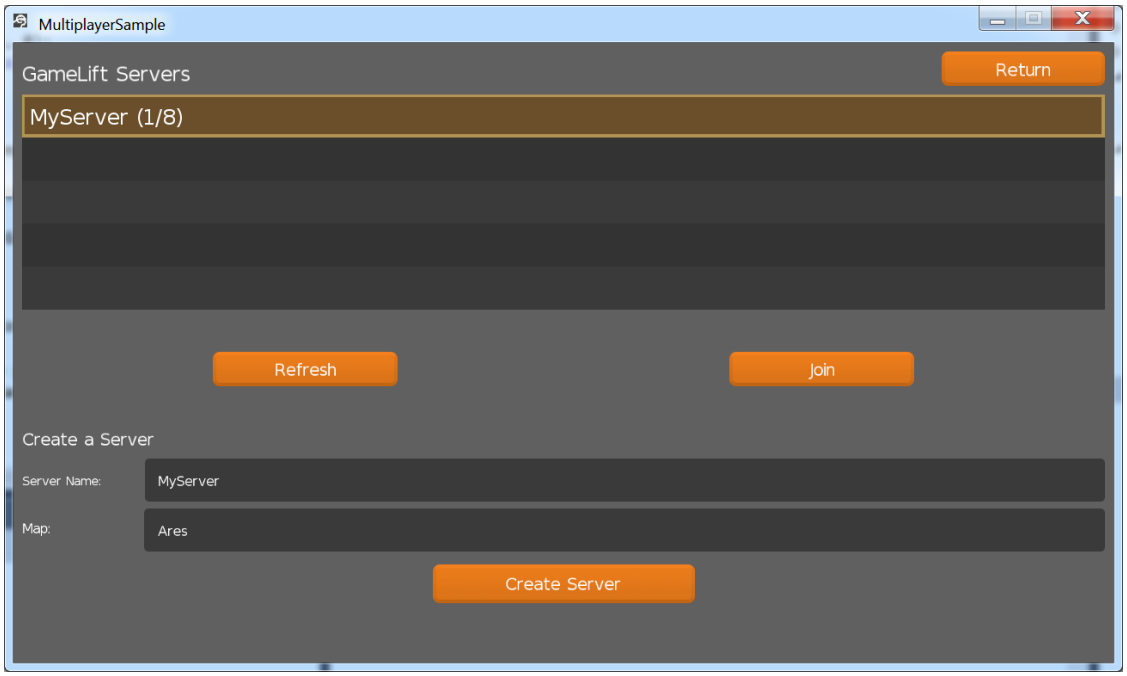
2. Click **Connect**.



3. Specify the **Server Name** and the **Map** (`sv_map`) to load.



4. To join automatically, click **Create Server**. To search active sessions and select a session to join, click **Refresh**, and then click **Join**.



Create an Amazon GameLift Package

To create a Amazon GameLift package, complete the following steps.

To create a Amazon GameLift package

1. Before you create a Amazon GameLift package, do the following:
   - Compile game assets
   - Build the Lumberyard executable
2. Run the following commands to create the Amazon GameLift package:

```
mkdir GameLiftPackageWindows
```

```
cp -r MultiplayerSample_pc_Paks_Dedicated/* GameLiftPackageWindows/
```

```
cp -r Bin64vc120.Dedicated/* GameLiftPackageWindows/
```

3. Upload your build and create a fleet from the Amazon GameLift console. For more information, see Uploading Your Game to Amazon GameLift.

Secured Connection (Not Amazon GameLift Specific)

Amazon GameLift uses the OpenSSL-based secure socket driver to create a secured connection. However, instead of verifying the server, the secure socket driver can verify the client.

To enable a secured connection, make the following change to the `game.cfg` file:

```
gm_netsec_enable = 1
```

If client verification is needed, make the following change to the `game.cfg` file:

```
gm_netsec_verify_client = 1
```

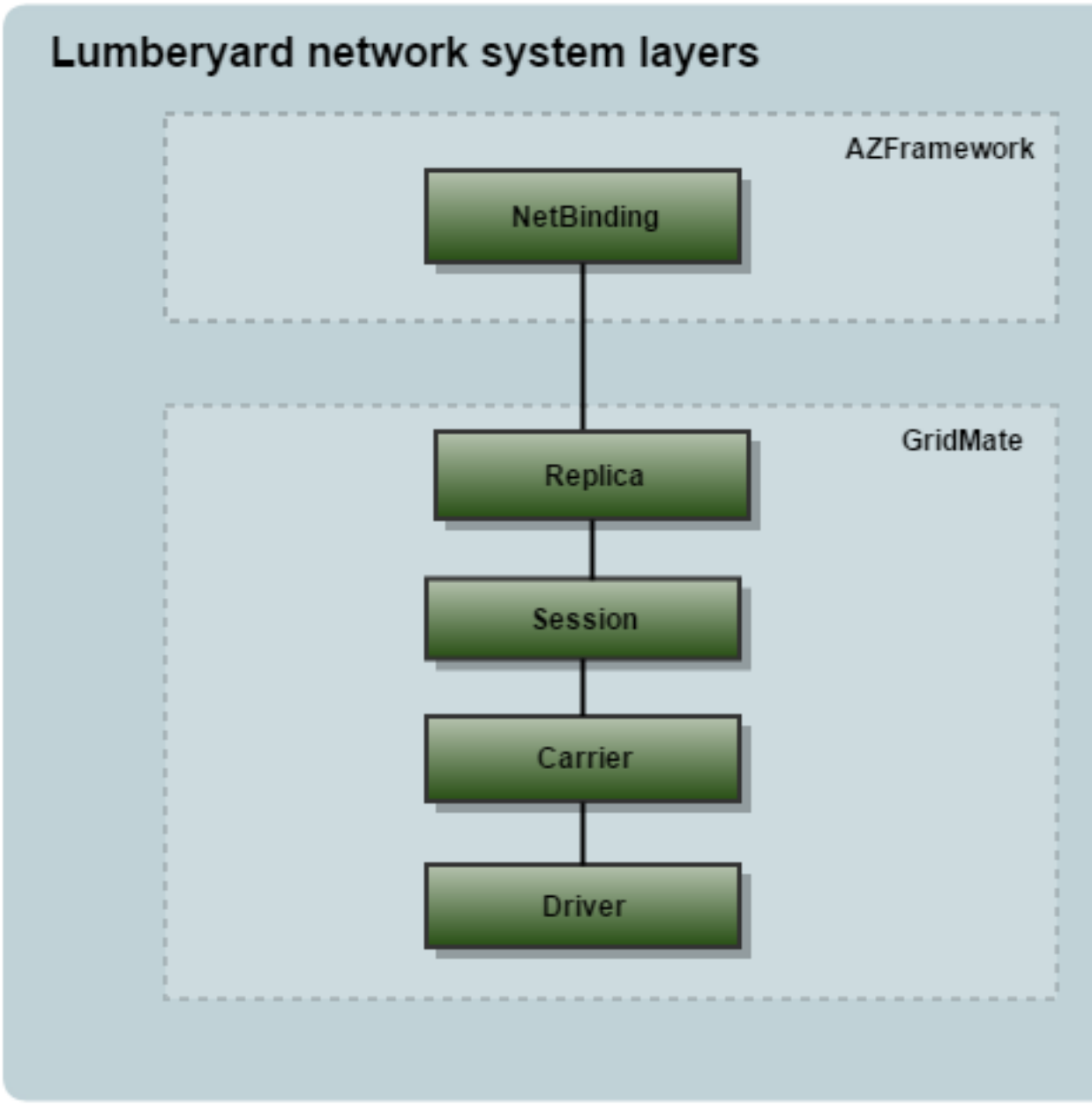
Note
By default, the certificate and private key are loaded from the `multiplayersample.cert.pem` file (shared by the certificate and CA root) and from the `multiplayersample.key.pem` file. To specific different files, use the `gm_netsec_certificate` and `gm_netsec_private_key` CVARs.

Overview

Lumberyard enables multiplayer functionality through the following software layers:

- AZFramework
  - Netbinding
- GridMate
  - Replica
  - Session
  - Carrier
  - Driver

These layers are illustrated in the following diagram.



NetBinding

The network binding API of the AzFramework library provides a way for components to synchronize their state over the network. The API is implemented on top of GridMate replicas. A special `NetBindingComponent` is responsible for the actual binding process, so entities that need to be synchronized need to have a `NetBindingComponent` added to them. When a game enters a multiplayer session, the `NetBindingComponent` collects replica chunks from the `NetBindables` on the entity, and adds them to a replica master.

GridMate

GridMate is a multiplatform library that enables you to easily add online features to your games. The GridMate API library has two general categories: network synchronization and online platforms. Each API is designed to be modular and extensible. Services can be enabled independently of each other, and different implementations can be provided for each API. Optional features are implemented as plugins for ease of customization. GridMate is built on top of Lumberyard's AzCore library. Service APIs are implemented using EBuses (AzCore's implementation of signal/slots) to improve modularity and extensibility. All GridMate allocations are piped through two specific allocators: `GridMateAllocatorMP` is used for allocations from the network synchronization APIs. `GridMateAllocator` is used for all other allocations, such as those from the online platform APIs and core system allocations. GridMate also supports debugging through AzCore's Driller framework. All network and replication events are reported and can be captured for logging and debugging purposes.

Replica

GridMate uses a single-master replication model. For each replica, one node in the session owns the master copy, and everyone else has a proxy copy. Replicas can be individually migrated from node to node at any time.

At the core of GridMate's replication model is the replica. Replicas, along with the chunks, datasets and RPCs that make up the replica, provide a mechanism for capturing and propagating the game state. Replicas also serve as the point of interaction for external game systems. Replicas can be owned by any node in the network and can be migrated to whichever node that can process them most efficiently.

Each node in the replication network runs a local instance of replica manager. As a node establishes connections to other nodes, it adds them to the replica manager as peers. This builds out its replication network.

One important design element of GridMate replicas is the broadcast nature of the system. Many replication systems allow users to specify replication targets directly, either per replica or per update. This attempt to enable bandwidth optimizations is error prone and puts the implementation burden on gameplay programmers who are often less familiar with network desynchronization issues. Instead, GridMate's approach follows the rule "when something happens, it happens for everyone".

Session

The session service is responsible for managing and maintaining the connectivity required to other members in a game session. GridMate's session service consists of a simple matchmaking API to facilitate integration with existing matchmaking services, and a session implementation that supports three topologies: P2P full mesh, client/server and a hybrid mode that consists of a full mesh network connected to a client/server network. Host migration is available when using full mesh topology. Host migration is a multi-step process that begins as nodes lose connectivity to the session host. The first step is host election: as nodes disconnect from the host, they broadcast a request for a new host election, and go through a serious of voting rounds, until a majority is reached or the election process times out. The new host(s) then starts the migration process, dropping problematic connections and migrating replicas until the session is stable again before resuming normal operations. During this time the connection graph can be very unstable, and a variety of steps are taken to improve success rate.

Carrier

GridMate's carrier implementation provides reliable and unreliable messaging. Messages are sent over a channel. Each channel represents an independent stream of messages. Reliable and unreliable messages can be sent over the same channel. Within a channel, message delivery is always ordered, and out-of-order unreliable messages are always discarded. GridMate supports multiple channels to

compartmentalize the effect of packet losses and reordering. GridMate provides separate dedicated channels for replication and voice chat traffic. To minimize impact to and from the game thread, the current carrier implementation performs network sends and receives from a separate IO thread. Decoupling sends and receives into separate threads and incorporating epoll/IOCP is planned. The carrier API provides hooks for congestion control, connection handshakes and network simulators. Users can use the default implementations in GridMate or provide their own custom implementations.

Driver

The driver is the interface for the lowest level of the transport layer. Lumberyard ships with several driver implementations: `SocketDriver` is a generic socket driver that supports BSD/WinSock/Posix sockets on the corresponding platforms. The `SecureSocketDriver` supports encrypted communication through the DTLS protocol by using OpenSSL.

Other GridMate Features

Other GridMate features include:

- **Online Service** - Provides essential user information used by the other APIs.
- **Achievements** - An API for in-game achievements support.
- **Leaderboards** - An API for leaderboard support.
- **Online Storage** - An API for online storage support.

CryNetwork Backward Compatibility (Deprecated)

Lumberyard has a backwards compatibility layer for the deprecated legacy networking system called "CryNetwork". This layer is mostly encapsulated inside the CryNetwork library and exposed through the `INetwork` interface. The layer is intended only for projects that were built using CryNetwork so that you can transition them to Lumberyard's network technology (NetBinding components and GridMate). Because the CryNetwork backward compatibility API layer uses CPU and bandwidth inefficiently, we strongly recommend that you do not build or release multiplayer games using it.

Topics

- Networking Architecture (p. 966)
- Carrier (p. 969)
- Marshalling (p. 972)
- Sessions (p. 976)
- Replicas (p. 984)
- Replica Manager (p. 994)

Networking Architecture

Fundamental Concepts

Lumberyard provides a network layer that supports a wide variety of game types on multiple platforms and does not restrict game developers to using any particular network topology. You are able to create games using three network topologies: P2P full mesh, client/server, and a hybrid mode that consists of a full mesh network connected to a client/server network. You can create gameplay objects that are server authoritative, and gameplay objects that are client authoritative.

In this discussion, peer and host have the following meanings:

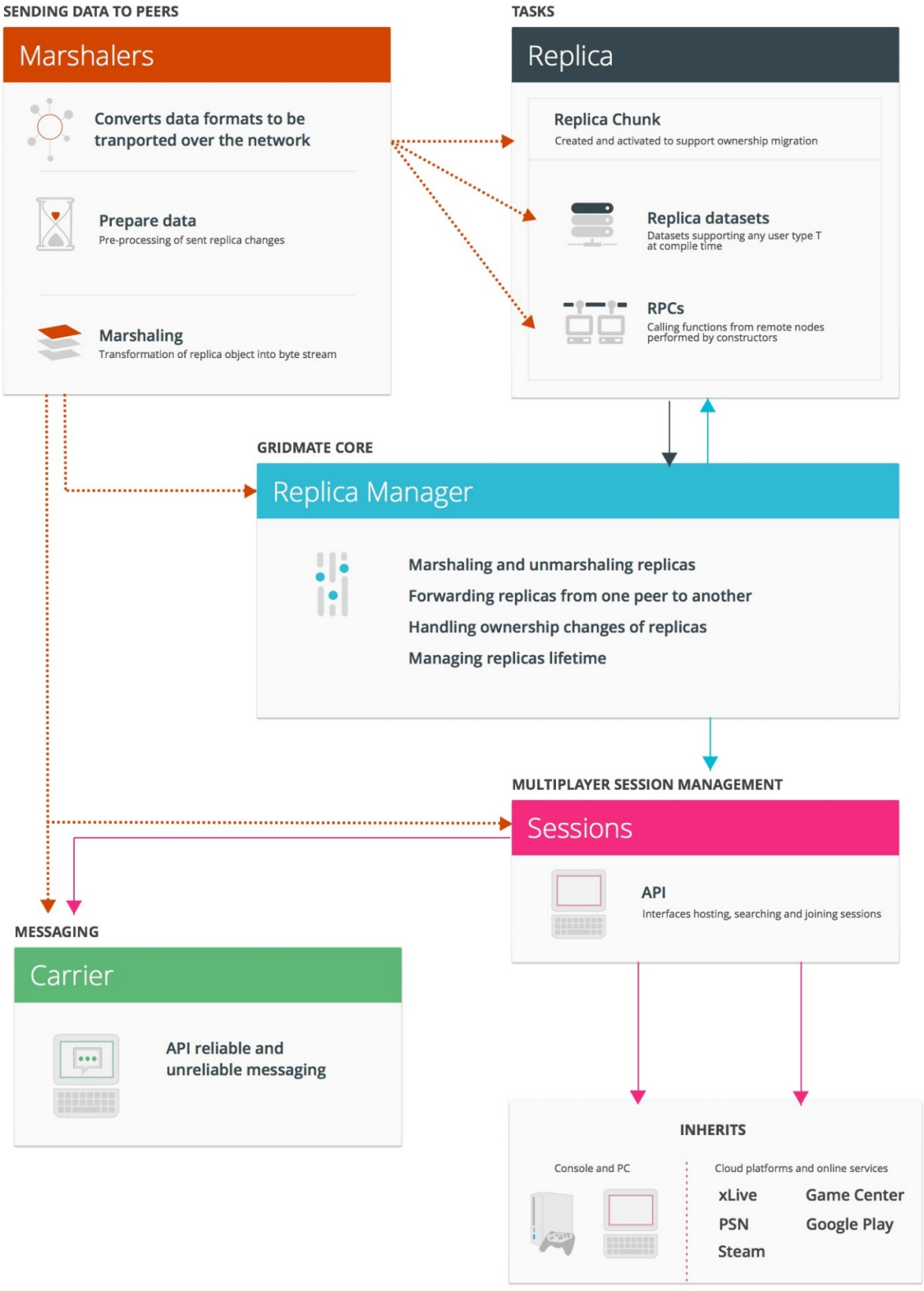
Peer - A network node that is participating in a game session.

Host - A special kind of a peer that manages the game session. The host can run on one of the game clients or be a dedicated server.

Synchronization of the states of various networked game objects is achieved through the GridMate replication model. One important design element is the concept of a `horizon`. GridMate does not maintain a full graph of the replication network at each node. Instead, each node is only aware of the peers that it has a direct connection to; everything else is considered the "horizon". Nodes keep track of which replica updates are arriving from which peer (upstream) only for purposes of routing, so they know where to forward upstream requests and, in the case of hub nodes, where to send them downstream. Basically, if a node receives a request for a replica it doesn't own, it forwards it upstream.

GridMate Architecture

The following diagram shows the major components of the GridMate architecture and their relationships.



For more information, see the following pages.

- Carrier (p. 969)
- Marshalling (p. 972)
- Sessions (p. 976)
- Replicas (p. 984)
- Replica Manager (p. 994)

Carrier

Carrier is GridMate's messaging API. GridMate's reliable UDP implementation supports both reliable and unreliable messages. There is no out-of-order delivery. Out-of-order messages are queued if sent reliably, or discarded if sent unreliably.

The carrier sends messages through channels. The purpose of channels is to separate unrelated traffic, such as game state and voice chat. Message ordering is not enforced across channels.

The carrier API also provides hooks for congestion control and traffic simulation.

Channels and Message Priorities

Messages can be sent on different channels and have different priorities. Message ordering is always maintained between messages with the same priority sent on the same channel.

Channels provide a way to separate unrelated messages so that their ordering does not affect one other. When messages arrive out of order, they are either discarded or queued (and therefore delayed) depending on their reliability. Using different channels prevents unrelated messages from being unnecessarily dropped or delayed. For example, object replication traffic and voice chat traffic can be sent on different channels, so a missing reliable message for object replication would not cause voice chat data to be dropped, and vice versa.

Customizable Classes

You can customize the following classes to implement your own networking features:

- **Driver** - Carrier defers actual network operations to the driver, so different implementations can be provided for different platforms. This abstraction makes it possible to use platform-specific protocols from service providers such as Steam or XboxLive. The default implementation uses UDP and supports IPv4 and IPv6.
- **Simulator** - If a network simulator is present, the carrier passes all inbound and outbound traffic through it so different network conditions can be simulated. One simulator instance can be supplied per carrier instance. The default implementation can simulate different patterns for inbound and/or outbound latency, bandwidth caps, packet loss and packet reordering.
- **Traffic Control** - The traffic control module has two primary functions: provide network statistics and congestion control. Whenever messages are sent or received, they are passed along to the traffic control module so it can update its statistics, and also so it can provide feedback to limit the amount of data being sent. It also decides if messages should be considered lost and resent by the carrier.

CarrierDesc

`CarrierDesc` is the carrier descriptor. When you create a carrier, you use the `CarrierDesc` structure to specify the parameters for the current session.

CarrierDesc Parameters

The following parameters can be supplied during carrier initialization:

Parameter	Data Type	Description
`m_address`	const char *	Specifies the local communication address to which the driver will bind. A value of 0 specifies any address. The default is `nullptr`.
`m_connectionEvaluationThreshold`	float	When a disconnection is detected, specifies the threshold at which all other connections are checked using *m_connectionTimeoutMS * m_connectionEvaluationThreshold* to see if they are also failing because of a network failure. The default is `0.5f`.
`m_connectionTimeoutMS`	unsigned int	Determines the time to allow for a connection attempt. The default is 5000 milliseconds.
`m_disconnectDetectionPacketLossThreshold`	float	Packet loss percentage threshold. Possible values are from 0.0 to 1.0, where 1.0 is 100 percent. The connection will be dropped after packet loss exceeds the value specified. The default is `0.3f`.
`m_disconnectDetectionRttThreshold`	float	Specifies the RTT (round-trip time) threshold in milliseconds. The connection is dropped when the measured RTT is greater than the value specified. The default is `500.0f`.
`m_driver`	class Driver *	Specifies a custom driver implementation. The default is `nullptr`.
`m_driverIsCrossPlatform`	bool	Specifies whether the driver maintains cross-platform compatibility. When true, the default driver drops to the most restrictive MTU (maximum transmission unit) across all supported platforms. The default is `false`.
`m_driverIsFullPackets`	bool	Specifies whether the driver ignores MTU limits. This parameter applies only to socket drivers and local area networks. An internet packet is usually around 1500 bytes. A value of `true` enables a maximum packet size of 64 KB. These big packets fail on the Internet but typically do not on local networks. The default is `false`.
`m_driverReceiveBufferSize`	unsigned int	Specifies the size of the internal receive buffer that the driver uses. A value of 0 specifies the default buffer size. This parameter can be used only if `m_driver == null`. The default is `0`.
`m_driverSendBufferSize`	unsigned int	Specifies the size of the internal send buffer that the driver uses. A value of 0 specifies the default buffer size. This parameter can be used only if `m_driver == null`. The default is `0`.
`m_enableDisconnectionDetection`	bool	Specifies whether the carrier drops connections when traffic conditions are bad. The default is `true`. **Note** This parameter should be set to `false` only when debugging.
`m_familyType`	int	Specifies the protocol family that the driver uses. A value of `0` specifies the default family.

Parameter	Data Type	Description
m_port	unsigned int	Specifies the local communication port to which the driver binds. A value of 0 specifies the port assigned by the system.
m_securityData	const char *	Specifies a pointer to a string with security data. For example, on XBox One, this value is the security template name. The default is nullptr.
m_simulator	class Simulator *	Optionally specifies a simulator through which all network messages are filtered. When specified, the carrier passes all inbound and outbound traffic through the specified simulator so that different network conditions can be simulated. You can specify one simulator instance per carrier instance.
m_threadCpuID	int	Restricts the carrier thread to a specific CPU core. The values that can be specified are platform dependent. A value of -1 specifies no restriction. The default is -1.
m_threadInstantResponse	bool	Specifies whether IO events wake up the carrier thread immediately. The default is false. **Note** Setting this value to true typically uses more bandwidth because messages (especially small messages) are grouped less efficiently.
m_threadPriority	int	Specifies the thread priority for the carrier thread. The values that can be specified are platform dependent. A value of -100000 inherits the priority from calling thread. The default is -100000.
m_threadUpdateTimeMS	int	Specifies, in milliseconds, how often the carrier thread is updated. This parameter is ignored if m_threadInstantResponse is true. Possible values are from 0 through 100. In general, the time interval should be higher than 10 milliseconds. Otherwise, it is more efficient to set m_threadInstantResponse to true. The default is 30 milliseconds.
m_trafficControl	class TrafficControl *	Specifies a custom traffic control implementation that controls traffic flow to all connections and that handles issues like network congestion.
m_version	VersionType	Specifies the version of Carrier API that is being used. Carriers with mismatching version numbers are not allowed to connect to each other. The default is 1.

Topics

- Carrier Message Structure (p. 971)

Carrier Message Structure

This topic describes the message structure used by the CarrierImpl networking class found in the Carrier.cpp file in the GridMate source code.

In the following sections, values in parentheses indicate the field's length in bits. For fields with variable length, the value indicates the minimum length.

Datagram Format

The overall datagram has the following structure.

```
DgramID (16) | Msg1 (64+) | Msg2 (24+) | ...
```

Message Format

The following diagram shows the possible message fields. Only the first two fields are present in every message header. All the other fields are sent only as necessary. In general, `ChannelId` and `NumChunks` are rarely sent. `SeqNum` and `RelSeqNum` are usually sent once per datagram.

```
Flags (8) | Length (16) | ChannelId (8) | NumChunks (16) | SeqNum (16) | RelSeqNum (16) |
 Payload (0+)
```

System Messages

Carrier system messages include `ACK` and `ClockSync`.

ACK

The ACK system message is used to ACK any received messages and to keep the connection alive. When there is no activity, an ACK containing only the first two fields is sent, otherwise, the actual fields sent vary depending on the pattern being ack'ed. At the very least, `LastToAck` is sent. If the sequence acke'd contains gaps, a variable-length bit set is used; otherwise, the first sequence number being ack'ed is included. These possible message formats are shown in the following diagram.

```
MsgId (8) | Flags (8)
MsgId (8) | Flags (8) | LastToAck (16) | AckHistoryBits (1+)
MsgId (8) | Flags (8) | LastToAck (16) | (FirstToAck (16)
```

ClockSync

A `ClockSync` message is sent about once per second to keep all the clocks in the session in sync. The message format is as follows.

```
MsgId (8) | Time (32)
```

Marshalling

Data is written to the network using `WriteBuffer`, and data received is read using `ReadBuffer`. Each buffer specifies the endianness used.

All data marshalling, whether for a `DataSet` or RPC, is written using a specialization of the `Marshaler` type. There are a number of pre-defined marshalers for fundamental types (`int32`, `uint16`, `bool`, `float`, etc), as well as other common types like containers and bitfields.

Marshalers and read/write buffers have a close relationship. A marshaler reads or writes its data types from or to the buffer. If the type is a complex type like a class or container, then that marshaler marshals each of its fields with nested marshalers. The nested invocation of marshaler types continues until a fundamental type is written to the buffer with the endianness of the network. Additional custom marshalers can be implemented to support custom types or to perform domain-based compression. Default marshalers are implemented through template specialization.

The base `Marshaler` class in GridMate follows.

```
namespace GridMate
{
    template<typename T>
    class Marshaler
    {
    public:
        void Marshal(WriteBuffer& wb, const T& value);
        void Unmarshal(T& value, ReadBuffer& rb);
    };
}
```

If a `Marshaler` instance is not specified with the data set or RPC declaration, the template specialization is used.

Implementation of the default marshaler for AZCore's `Vector3` math type can be found in `Code/Framework/GridMate/GridMate/Serialize/MathMarshal.h`:

```
namespace GridMate
{
    template<>
    class Marshaler<AZ::Vector3>
    {
    public:
        typedef AZ::Vector3 DataType;
        static const AZStd::size_t MarshalSize = sizeof(float) * 3;
        void Marshal(WriteBuffer& wb, const AZ::Vector3& vec) const
        {
            Marshaler<float> marshaler;
            marshaler.Marshal(wb, vec.GetX());
            marshaler.Marshal(wb, vec.GetY());
            marshaler.Marshal(wb, vec.GetZ());
        }
        void Unmarshal(AZ::Vector3& vec, ReadBuffer& rb) const
        {
            float x, y, z;
            Marshaler<float> marshaler;
            marshaler.Unmarshal(x, rb);
            marshaler.Unmarshal(y, rb);
            marshaler.Unmarshal(z, rb);
            vec.Set(x, y, z);
        }
    };
}
```

Markers

Notice the declaration of `MarshalSize` above. `WriteBuffer` supports the concept of markers. A marker is a placeholder that can be inserted into the buffer, so its value can be filled after additional data is written to the buffer. This is useful for prepending a length field in front of the actual data. Markers require that the data that is inserted be of fixed length, and `MarshalSize` is used to query this length. Therefore, marshalers that write data to the marker need to declare a valid `MarshalSize`.

Buffers

Write Buffers

Write buffers are backed by the following three types of allocation schemes:

Dynamic – Dynamically allocated and automatically grown

Static – Fixed size, allocated on the stack

Static In Place – Uses another buffer as its backing store

By default, the `write` function uses the default marshaler for the data type, but you can override the marshaler to create a custom marshaler.

There are two ways to write a type to a network buffer:

1) The following example uses the default marshaler for the type passed into `Write()`. In this example, the float marshaler is used.

```
WriteBuffer wb;
wb.Write(1.0f);
```

2) The following example uses the `HalfMarshaler`, which compresses the float by half.

```
WriteBuffer wb;
wb.Write(1.0f, HalfMarshaler());
```

Read Buffers

Read buffers have built-in overflow detection and do not read any data fields after the end of the buffer has been reached. You can check this by looking at the return value of the `Read` method. Note that if data isn't read for a given value, then the value is left uninitialized.

Predefined Marshalers

GridMate includes the following predefined marshalers:

Fundamental C++ Types

Floating point	Misc	Unsigned	Signed
`float`	`char`	`AZ::u8`	`AZ::s8`
`double`	`bool`	`AZ::u16`	`AZ::s16`
	`enum` (specify marshaled size by inheriting `enum` from a type)	`AZ::u32`	`AZ::s32`
		`AZ::u64`	`AZ::s64`

Container Types

Sequence	Associative	Explicit Marshalers
`vector`	`map`	`ContainerMarshaler`
`list`	`set`	`MapContainerMarshaler`
`string`	`unordered_map`	(Use these marshalers when the subtypes of the container require a non-default marshaler)
	`unordered_set`	
	`multimap`	

Sequence	Associative	Explicit Marshalers
	`multiset`	

Utility Types

Name	Description
ConversionMarshaler<SerializedType, OriginalType>	Performs static casts between `SerializedType` (type on the wire) and `OriginalType` (type declared in user code).
`AZ::Crc32`	A `CRC32` value.
`AZStd::bitset`	A class for arbitrary flags.
`AZStd::pair`	A `std` pair class. Implicitly used by the map, unordered_map, and multimap marshalers.
`AZ::Aabb`	An axis aligned bounding box.
`AZStd::chrono::duration`	A time duration in 32 bit milliseconds.
`GridMate::UnionDataSet`	A type safe tagged union designed for network transmission.

Compression Types

Name	Description
`Float16Marshaler`	Compresses a `float32` to `float16`.
`HalfMarshaler`	Compresses a float to half precision.
`IntegerQuantizationMarshaler<Min, Max, Bytes>`	Quantizes an integer in the range [`Min`, `Max`] to the number of bytes specified in `Bytes`.

Custom Marshalers

Creating a custom data marshaler is as simple as specializing the `GridMate::Marshaler` type, and implementing the expected `Marshal` and `Unmarshal` methods. If the data written is constant size, adding the member `MarshalSize` allows you to use the marshaler in scenarios where fixed sizes are required (such as markers).

Fixed Size Custom Marshaler

The following is an example of a fixed size custom marshaler.

```
namespace GridMate
{
    template<>
    class Marshaler<MyClass>
    {
    public:
        static const AZStd::size_t MarshalSize = sizeof(m_field1) + sizeof(m_field2);
```

```
        void Marshal(GridMate::WriteBuffer& wb, const MyClass& value) const
        {
            wb.Write(value.m_field1);
            wb.Write(value.m_field2);
        }
        void Unmarshal(MyClass& value, ReadBuffer& rb) const
        {
            rb.Read(value.m_field1);
            rb.Read(value.m_field2);
        }
    };
}
```

Sessions

GridMate session service provides session connectivity and management. Both hub-and-spoke (client/server) and P2P full-mesh topologies are supported.

You can also create multiple sessions for each GridMate instance. Each session creates its own carrier and replica manager instances, so there is no interaction between sessions. GridMate sessions support host migration when running in P2P mode.

Topics

- Starting and Stopping the Session Service (p. 976)
- Hosting a Session (p. 978)
- Searching for a Session (p. 979)
- Joining a Session (p. 981)
- Reacting to Session Events (p. 982)

Starting and Stopping the Session Service

The session service is responsible for hosting or joining sessions and is represented by the `GridMate::SessionService` abstract class.

When a session service is created, a descriptor class derived from `GridMate::SessionServiceDesc` is passed in as a constructor argument.

The implementations of `GridMate::SessionService` that are included with the base Lumberyard engine are as follows.

Implementation	Descriptor	Description
`GridMate::LANSessionService`	`GridMate::SessionServiceDesc`	Sessions hosted over a local area network.

Starting a Session Service

Only one session service can be present per `GridMate::IGridMate` instance.

> **Note**
> Attempting to register multiple session services causes an assert and overrides any previously registered session services.

You have two ways to start a session service:

- Create a session service object and register it with GridMate.
- Register an existing session service object with GridMate.

Starting Method	Details
`GridMate::StartGridMateService()`	Creates a session service object and registers it with `GridMate::IGridMate`.
`GridMate::RegisterService()`	Registers an existing session service object with `GridMate::IGridMate`.

Stopping a Session Service

The method for stopping the session service depends on how the session service was started.

Starting Method	Stopping Method	Details
`GridMate::StartGridMateService()`	See details.	The session service is stopped when `GridMate::IGridMate` is destroyed by using the `GridMate::GridMateDestroy()` method.
`GridMate::RegisterService()`	`GridMate::UnregisterService()`	The session is service is stopped and memory freed when `GridMate::UnregisterService()` is called.

Examples

The following examples assume that GridMate has been initialized.

Starting and Stopping with GridMate::StartGridMateService

The following example uses `GridMate::StartGridMateService`.

```
void MyClass::StartSessionService()
{
    IGridMate* gridMate = gEnv->pNetwork->GetGridMate();

    if(gridMate)
    {
        // The session service is started and will be stopped when IGridMate is destroyed.
        GridMate::SessionServiceDesc desc;
        GridMate::StartGridMateService<GridMate::LANSessionService>(gridMate, desc);
    }
}
```

Starting and Stopping with GridMate::RegisterService() and GridMate::UnregisterService()

The following example uses `GridMate::RegisterService()` and `GridMate::UnregisterService()`.

```
void MyClass::StartSessionService()
{
    IGridMate* gridMate = gEnv->pNetwork->GetGridMate()
    GridMate::SessionService* sessionService = nullptr;

    if(gridMate)
    {
        GridMate::SessionServiceDesc desc;
        sessionService = aznew GridMate::LANSessionService(desc);
        gridMate->RegisterService(sessionService);
```

```
    }

    return sessionService;
}

void MyClass::StopSessionService(GridMate::SessionService* sessionService)
{
    IGridMate* gridMate = gEnv->pNetwork->GetGridMate()

    if(gridMate)
    {
        // Unregister the session service and free the session service pointer.
        gridMate->UnregisterService(sessionService);
    }
}
```

Hosting a Session

A session can be hosted by calling `IGridMate::HostSession()` after the session service has been started. The session settings and configuration are set in the `GridMate::SessionParams` argument, which acts as a base class for certain implementations of `GridMate::SessionService`.

Implementation of `GridMate::SessionService`	**Implementation of `GridMate::SessionParams`**
`GridMate::LANSessionService`	`GridMate::LANSessionParams`

GridMate::SessionParams

The following table shows the supported parameters in `GridMate::SessionParams`.

Parameter	Required	Default	Description
`m_localMember`	Yes		This is not required for a LAN session, only for consoles.
`m_topology`	No	`ST_PEER_TO_PEER`	`ST_CLIENT_SERVER`: A client is only connected to the server. `ST_PEER_TO_PEER`: A client is connected to all other clients.
`m_peerToPeerTimeout`	No	10000	The time without a response, in seconds, after which a peer is disconnected.
`m_numPublicSlots`	Yes		The maximum number of players that can join the session.

GridMate::LANSessionParams

`GridMate::LANSessionParams` has the following additional parameter.

Parameter	Required	Default	Description
`m_port`	No	0	The port to monitor for search requests from other clients. If 0, this session is hidden to searches. Otherwise, the port number falls in the range from 1 through 65536.

Events

The following table describes GridMate session service events.

Event	Description
`OnSessionCreated`	A new session has just been created.
`OnMemberJoined`	A player has joined the session.
`OnMemberLeaving`	A player has left the session.

Examples

The following example hosts a session. The example assumes that GridMate has been initialized and a session service registered.

```
bool MyClass::HostSession()
{
    GridMate::IGridMate* gridMate = gEnv->pNetwork->GetGridMate();

    if(gridMate)
    {
        GridMate::LANSessionParams params;
        params.m_topology = Gridmate:ST_CLIENT_SERVER;
        params.m_numPublicSlots = 10;
        params.m_port = 10000;
        params.m_flags = 0;
        params.m_localMember = gridMate->GetOnlineService()->GetUser();

        GridMate::Session session = gridMate->HostSession(&params,
 GridMate::CarrierDesc());
        if(session != nullptr)
        {
            // Failed to create the session..
            return true;
        }
    }
    return false;
}
```

Searching for a Session

You search for a session by calling `GridMate::StartGridSearch()` after the session service has been started. The session settings and configuration are set in the `GridMate::SearchParams`, which acts as a base class for certain implementations of `GridMate::SessionService`.

Implementation of `GridMate::SessionService`	**Implementation of `GridMate::SearchParams`**
GridMate::LANSessionService	GridMate::LANSearchParams

GridMate::SearchParams

The following table shows the supported parameters in `GridMate::SearchParams`.

Parameter	Required	Default
m_localMember	Yes	
m_maxSessions	No	8
m_timeOutMs	No	2000
m_version	No	1

GridMate::LANSearchParams

GridMate::LANSessionParams has the following additional parameters.

Parameter	Required	Default	Description
m_serverAddress	No	Empty	The address of a server to search for. If empty, a broadcast address is used.
m_serverPort	Yes		The port that game servers monitor for searches.
m_broadcastFrequencyMs	No	1000	The interval, in milliseconds, between search broadcast requests.

Search Results

When a search is complete, the OnGridSearchComplete() event is called. The results are found in the GridMate::GridSearch argument.

GridMate::GridSearch contains an array of search results.

To query the size of the array, use GridMate::GridSearch::NumResults().

To query individual results, use GridMate::GridSearch::GetResult().

The GridMate::SearchInfo object contains more details about the session (for example, the number of used and free player slots) and can be used when Joining a Session (p. 981).

Events

The following table describes GridMate session search events.

Event	Description
OnGridSearchStart	A grid search has started.
OnGridSearchComplete	A grid search has finished and contains the results.

Examples

The following example searches for all available sessions. The example assumes that GridMate has been initialized, a session service has been registered, and the class MyClass is listening for session events.

```
void MyClass::StartSearch()
{
    GridMate::IGridMate* gridMate = gEnv->pNetwork->GetGridMate();
```

```
    if(gridMate)
    {
        GridMate::LANSearchParams params;
        params.m_serverPort = 20000;
        params.m_localMember = gridMate->GetOnlineService()->GetUser();
        gridMate->StartGridSearch(&params);
    }
}

void MyClass::OnGridSearchComplete(GridMate::GridSearch* search)
{
    if(search->GetNumResults() > 0)
    {
        // Found sessions that match the specified criteria
    }
}
```

Joining a Session

You have two ways to join a session:

- By Searching for a Session (p. 979) and using a `GridMate::SearchInfo` object from the results.
- Directly to an existing game session by using a `GridMate::SessionIdIffo` object.

Regardless of the method, a session is joined using one of the overloaded `IGridMate::JoinSession()` functions after the session service has been started.

Note
The argument `GridMate::JoinParams` currently has no supported parameters.

Events

The following table describes GridMate session join events.

Event	Description
`OnSessionJoined`	The client has been successfully added to the session.
`OnMemberJoined`	A player has joined the session.
`OnMemberLeaving`	A player has left the session.

Example

The following example joins a session that has been found as the result of a session search.

```
void MyClass::OnGridSearchComplete(const GridMate::GridSearch* search)
{
    GridMate::IGridMate* gridMate = gEnv->pNetwork->GetGridMate();

    if(gridMate)
    {
        if(search->GetNumResults() > 0)
        {
            GridMate::Session* session = gridMate->JoinSession(search->GetResult(0),
 GridMate::JoinParams(), GridMate::CarrierDesc());
        }
    }
```

```
}

void MyClass:OnSessionJoined(GridMate::GridSession* session)
{
    // Joined the session successfully
}
```

Reacting to Session Events

Much of the session functionality is asynchronous because functions can be called, but the response is often not immediately available. For example, messages may be slowed by network transfer time, server processing, or the required response time.

The Event Bus (EBus) (p. 543) in Lumberyard is an event bus system that can send out events when asynchronous session functions are complete. This topic shows you how to set up your application to use the event bus and to connect and disconnect from it.

Setup

Your application must derive a class from `GridMate::SessionEventBus::Handler`. This class must contain certain overridden session events. However, not all events need to be implemented. An example follows.

```
class MyClass : public GridMate::SessionEventBus::Handler
{
    public:
        void OnSessionJoined(GridMate::GridSession* session) override;
        void OnMemberJoined(GridMate::GridSession* session, GridMate::GridMember* member)
 override;
        void OnMemberLeaving(GridMate::GridSession* session, GridMate::GridMember* member)
 override;
};
```

Connect

The following example shows how to connect to the session event bus and start receiving session events.

```
void MyClass::Init()
{
    GridMate::IGridMate* gridMate = gEnv->pNetwork->GetGridMate();

    if(gridMate)
    {
        GridMate::SessionEventBus::Handler::BusConnect(gridMate);
    }
}
```

Disconnect

The following example shows how to disconnect from the session event bus and stop receiving session events.

```
void MyClass::Term()
{
    GridMate::IGridMate* gridMate = gEnv->pNetwork->GetGridMate();

    if(gridMate)
    {
        GridMate::SessionEventBus::Handler::BusDisconnect(gridMate);
    }
}
```

Network Session Service Event Descriptions

A description of each session event follows.

```
virtual void OnSessionServiceReady()
```

Callback that occurs when the session service is ready to process sessions.

```
virtual void OnGridSearchStart(GridSearch* gridSearch)
```

Callback when a grid search begins.

```
virtual void OnGridSearchComplete(GridSearch* gridSearch)
```

Callback that notifies the title when a game search query is complete.

```
virtual void OnGridSearchRelease(GridSearch* gridSearch)
```

Callback when a grid search is released (deleted). It is not safe to hold the grid pointer after this event.

```
virtual void OnMemberJoined(GridSession* session, GridMember* member)
```

Callback that notifies the title when a new member joins the game session.

```
virtual void OnMemberLeaving(GridSession* session, GridMember* member)
```

Callback that notifies the title that a member is leaving the game session.

Warning
The member pointer is not valid after the callback returns.

```
virtual void OnMemberKicked(GridSession* session, GridMember* member)
```

Callback that occurs when a host decides to kick a member. An `OnMemberLeaving` event is triggered when the actual member leaves the session.

```
virtual void OnSessionCreated(GridSession* session)
```

Callback that occurs when a session is created. After this callback it is safe to access session features. The host session is fully operational if client waits for the `OnSessionJoined` event.

```
virtual void OnSessionJoined(GridSession* session)
```

Called on client machines to indicate that the session has been joined successfully.

```
virtual void OnSessionDelete(GridSession* session)
```

Callback that notifies the title when a session is about to be terminated.

Warning
The session pointer is not valid after the callback returns.

```
virtual void OnSessionError(GridSession* session, const string& errorMsg )
```

Called when a session error occurs.

```
virtual void OnSessionStart(GridSession* session)
```

Called when the game (match) starts.

```
virtual void OnSessionEnd(GridSession* session)
```

Called when the game (match) ends.

```
virtual void OnMigrationStart(GridSession* session)
```

Called when a host migration begins.

```
virtual void OnMigrationElectHost(GridSession* session,GridMember*& newHost)
```

Called to enable the user to select a member to be the new Host.

Note
The value is ignored if it is null, if the value is the current host, or if the member has an invalid connection ID.

```
virtual void OnMigrationEnd(GridSession* session,GridMember* newHost)
```

Called when the host migration is complete.

```
virtual void OnWriteStatistics(GridSession* session, GridMember* member, StatisticsData& data)
```

Called at the last opportunity to write statistics data for a member in the session.

Replicas

Game sessions use replicas to synchronize the state of the session. To use a replica, you simply declare the states that must be synchronized and the remote procedure calls (RPCs) that are supported. After you bind the replica object to the network, the engine does the work. There is no need to worry about how to properly route messages or discover remote objects. When you add a local (master) replica to the network, the replica is automatically discovered by remote nodes. In addition, corresponding remote proxy replica objects are created on the remote nodes. Only the owner of the replica is allowed to change states, and new states are automatically propagated to all other nodes. RPCs can be called from any node but are routed to the master (owner) node for verification and processing.

Topics

- Replica (p. 984)
- Replica Chunks (p. 987)
- Datasets (p. 989)
- Remote Procedure Calls (RPCs) (p. 991)

Replica

Replicas are core components of GridMate's replication system that are created by network-connected GridMate peers. When a peer creates a replica, GridMate propagates the replica over the network to synchronize the replica's state across the session. A locally created and owned replica is called a *master replica*. The copy of the master replica that connected peers receive is called a *proxy replica*. The synchronization and instantiation of replicas is handled by Replica Manager (p. 994).

Replica Chunks

Every replica holds a collection of user-defined `ReplicaChunk` (p. 987) objects that are synchronized with all the peers in the current session. A replica chunk is a container for user-defined DataSet (p. 989) objects and Remote Procedure Calls (RPCs) (p. 991). Any change to a `DataSet` object or a call to an RPC causes the replica to synchronize its state across the session.

Limitations

Replica chunks have the following limitations:

- Each replica can contain only 32 chunks.
- Chunks can be attached or detached only when a replica is not bound to a replica manager.

Creating a Replica and Attaching Chunks

To create a replica, invoke the following method:

```
GridMate::ReplicaPtr replica = GridMate::Replica::CreateReplica();
```

Most use cases require only one chunk per replica. To create a chunk and attach it to a replica by using a single call, use the `CreateAndAttachReplicaChunk` helper function, as in the following example:

```
GridMate::CreateAndAttachReplicaChunk<MyReplicaChunk>(replica, ...);
```

If you just want to attach a chunk to a replica, do the following:

```
replica->AttachReplicaChunk(myChunk);
```

For more information about the creation and propagation of replica chunks, see Replica Chunks (p. 987).

Binding a Replica to the Session Replica Manager

In order for a replica to be synchronized, it must be bound to the session replica manager. After you create a replica and attach chunks to it, get the replica manager from the GridMate session (p. 976). Then, bind the replica to it as follows:

```
GridMate::ReplicaManager* replicaManager = session->GetReplicaMgr();
replicaManager->AddMaster(replica);
```

Proxy replicas are automatically instantiated by remote peers' replica managers and, therefore, automatically bound.

Replica Ownership

When a peer creates a replica and binds it to the session replica manager, that peer becomes the owner of the replica. Each replica can be owned by only one peer. The replica owner is the only peer on the network that has the authority to change the state of the replica. For example, it can change the chunks' datasets or directly execute its RPCs. Any state changes performed on a proxy replica are considered invalid and do not propagate throughout the session. RPCs can be called on a proxy replica, but the calls are forwarded to the owner for confirmation before they can be executed. Once this confirmation is given, the RPC is sent to all proxies and also executed locally by the peer. If the master replica denies the execution, no peers receive the RPC call.

Changing Ownership

Replica ownership can be transferred from one peer to another, but the current owner of the replica must agree to the transfer. For information on how a replica owner can prevent transfer of ownership, see Replica Chunks (p. 987).

Ownership transfer happens automatically when a session performs host migration on a peer-to-peer network. You can also request it explicitly by invoking the following method:

```
replica->RequestChangeOwnership(); // Request ownership of a given replica for the local
 peer
```

Ownership transfer is an asynchronous process. When an ownership transfer is completed, each replica chunk is notified of the change by the `OnReplicaChangeOwnership` callback function.

Replica ID

Each replica has a unique ID associated with it. The replica ID is guaranteed to be unique within a particular GridMate session. You can use the replica ID to retrieve a replica from the session replica manager, as in the following example:

```
GridMate::ReplicaManager* replicaManager = session->GetReplicaMgr();
GridMate::ReplicaPtr replica = replicaManager->FindReplica(myReplicaId);

if (replica == nullptr)
{
    // Replica with given ID does not exist
    return;
}

if (replica->IsProxy())
{
    // This is a proxy replica
}

if (replica->IsMaster())
{
    // This is a master replica
}
```

Lifetime

The lifetime of a replica is controlled by a `GridMate::ReplicaPtr`, which is a reference-counted smart pointer. The replica manager retains a reference to every replica that is bound to it. It is therefore safe to omit a reference to the replica from user code; the replica is not destroyed as long as the reference is held in replica manager. However, you can force the replica manager to release its reference and free the replica by invoking the following method:

```
replica->Destroy();
```

Sample Code

This example creates a user-defined chunk, creates a replica, attaches the chuck to the replica, and binds the replica to the session replica manager.

```
// User-defined ReplicaChunk class to be carried with the replica
class MyChunk : public GridMate::ReplicaChunk
{
public:
    GM_CLASS_ALLOCATOR(MyChunk);
    typedef AZStd::intrusive_ptr<MyChunk> Ptr; // smartptr to hold the chunk
    static const char* GetChunkName() { return "MyChunk"; } // Unique chunk name
    bool IsReplicaMigratable() override { return false; } // Replica ownership
                                                          // cannot be changed

    MyChunk () : m_data("Data", 0) { } // chunk constructor
    void OnReplicaActivate(const ReplicaContext& rc) override // Called when replica is
 bound
                                                              // to the replica manager
 (both
                                                              // on local and remote peers)
    {
        // printing out whether it is a proxy or a master replica
        if (IsMaster())
            printf("I am master!\n");
        if (IsProxy())
            printf("I am proxy!\n");
    }

    GridMate::DataSet<int> m_data; // data this chunk holds
 };
```

```
GridMate::ReplicaPtr replica = GridMate::Replica::CreateReplica(); // Creating a replica
GridMate::CreateAndAttachReplicaChunk<MyChunk>(replica); // Creating chunk of our custom
 type
                                                          // and attaching it to the replica

GridMate::ReplicaManager* replicaManager = session->GetReplicaMgr(); // Getting replica
 manager instance
                                                                     // from current
 session
replicaManager->AddMaster(replica); // Binding replica to the replica manager,
                                    // making local peer the owner of this replica

...
// Starting from this point and up until replica destruction, the replica and MyChunk
 object
// that the replica is carrying are synchronized with other peers.
// Other peers receive the new replica and bind it to their replica managers. When this is
 done,
// OnReplicaActivate is triggered, and the "I am proxy" message is printed out on the
 remote peers.
// Every change of m_data DataSet results in the synchronization of the new value in
// the master replica with all of the proxy replicas.
```

Replica Chunks

A replica chunk is a user extendable network object. One or more `ReplicaChunk` objects can be owned by a replica (p. 984), which is both a container and manager for replica chunks. A replica is owned by a master peer and is propagated to other network nodes as a proxy replica. The data that a replica chunk contains should generally be related to the other data stored within it. Since multiple chunks can be attached to a replica, unrelated data can be stored in other chunks within the same replica.

A replica chunk can contain Datasets (p. 989) and/or Remote Procedure Calls (RPCs) (p. 991). Data sets store arbitrary data, which only the master replica is able to modify. Any changes are propagated to the chunks in proxy replicas on the other nodes. RPCs are methods that can be executed on remote nodes. They are first invoked on the master, which decides whether the invocation will be propagated to the proxies.

Replica Chunk Requirements and Limitations

A replica chunk has several important attributes:

- It can have up to 32 `DataSet` definitions.
- It can have up to 256 RPC definitions.
- It is reference counted and therefore must be held by a smart pointer.
- It is not synchronized across the session until the replica manager is ready.

Implementing a New Replica Chunk Type

You have two ways to implement a new replica chunk type: handle data set changes and RPC calls ("game logic") inside the chunk, or outside the chunk. In both cases, the following apply:

- The name of the chunk type must be unique throughout the system. To achieve this, every replica chunk type must implement the static member function `const char* GetChunkName()`. The string returned by the `GetChunkName` function must uniquely identify the chunk type.
- To indicate whether the ownership of this type of chunk is transferrable, every chunk type needs to override the `bool IsReplicaMigratable()` virtual function. If any chunk in a replica is not migratable, the replica's ownership cannot be transferred from one peer to another.
- Every chunk type must define a smart pointer that holds the chunk type instances.

Declaring a Replica Chunk Type with Internal Game Logic Handling

To have your replica chunk class handle game logic directly, it should inherit from `ReplicaChunk`:

```
class MyChunk : public GridMate::ReplicaChunk
{
public:
    GM_CLASS_ALLOCATOR(MyChunk); // Using GridMate's allocator

    MyChunk()
        : m_data("Data", 0)                 // Initializing integer DataSet to zero, and
 assigning a name for it
        , MyRpcMethodRpc("MyRpcMethodRpc")  // Initializing RPC by passing in its name; the
 RPC name is for debugging purposes
    {
    }


    typedef AZStd::intrusive_ptr<DataSetChunk> Ptr;          // Defining smart pointer type
 for this chunk
    static const char* GetChunkName() { return "MyChunk"; }  // Unique chunk type name
    bool IsReplicaMigratable() override { return false; }    // Specify whether the chunk
 can participate in replica's ownership changes

    bool MyRpcMethod(int value, const GridMate::RpcContext& context)
    {
                    // Handle event here
        return true; // Propagate this call to all proxies
    }

    GridMate::Rpc<GridMate::RpcArg<int>>::BindInterface<MyChunk, &CustomChunk::MyRpcMethod>
 MyRpcMethodRpc;
    GridMate::DataSet<int> m_data;
 };
```

Declaring a Replica Chunk Type with External Game Logic Handling

To have your replica chunk class act as a simple data carrier and forward data changes and events to a designated handler (an external class), inherit your handler class from `ReplicaChunkInterface`, and your replica chunk class from `ReplicaChunkBase`:

```
class CustomHandler : public GridMate::ReplicaChunkInterface
{
public:
    GM_CLASS_ALLOCATOR(CustomHandler); // using GridMate's allocator

    void DataSetHandler(const int& value, const GridMate::TimeContext& context)
    {
        // Handle changes
    }

    bool RpcHandler(AZ::u32 value, const GridMate::RpcContext &context)
    {
                    // Handle event here
        return true; // Propagate this call to all proxies
    }
};

class MyChunk : public GridMate::ReplicaChunkBase
{
public:
    GM_CLASS_ALLOCATOR(MyChunk); // Using GridMate's allocator
```

```
    MyChunk()
        : m_data("Data", 0)                // Initializing integer DataSet to zero and
assigning a name for it
        , MyRpcMethodRpc("MyRpcMethodRpc") // Initializing RPC by passing its name; the
RPC's name is used for debugging purposes
    {
    }


    typedef AZStd::intrusive_ptr<DataSetChunk> Ptr;         // Defining smart pointer type
for this chunk
    static const char* GetChunkName() { return "MyChunk"; } // Unique chunk type name
    bool IsReplicaMigratable() override { return false; }   // Whether chunk can
participate in replica's ownership changes

    GridMate::DataSet<int>::BindInterface<CustomHandler, &CustomHandler::DataSetHandler>
m_data;
    GridMate::Rpc<GridMate::RpcArg<AZ::u32>>::BindInterface<CustomHandler,
&CustomHandler::RpcHandler> MyRpcMethodRpcPC;
};
```

Registering Chunk Type

Every user-defined replica chunk type should be registered with `ReplicaChunkDescriptorTable` to create the factory required by the Replica Manager (p. 994).

To register replica chunks, use this call:

```
GridMate::ReplicaChunkDescriptorTable::Get().RegisterChunkType<MyChunk>();
```

Attaching a Replica Chunk to the Replica

You must add a replica chunk to a replica before you bind the replica to replica manager. After you bind the replica to replica manager, you cannot add or remove replica chunks to or from the replica.

To create a replica chunk, use this call:

```
MyChunk::Ptr myChunk = GridMate::CreateReplicaChunk<MyChunk>(<...>);
```

Where `<...>` is forwarded to the `MyChunk` constructor.

To attach the chunk to a replica, use this call:

```
replica->AttachReplicaChunk(myChunk);
```

Alternatively, you can create the chunk and attach it in one step:

```
GridMate::CreateAndAttachReplicaChunk<MyChunk>(replica, <...>);
```

After you add the chunk to the replica, the replica retains a smart pointer to the chunk. The chunk is released only when its replica is destroyed.

Datasets

You can use `DataSet` objects to synchronize the state of a session across the network. When a value in the dataset changes, the updates are propagated automatically. Datasets can be of any type, but they must support the assignment and comparison operators. Your `DataSet` declaration can specify a custom marshaler. If you do not specify a marshaler, the `DataSet` object uses `GridMate::Marshaler<T>`.

A `DataSet` must be declared inside a `ReplicaChunk` object. A `ReplicaChunk` object can contain up to 32 `DataSet` objects. You must supply a debug name to the dataset constructor.

The following example declares a `ReplicaChunk` object that has two `DataSet` objects of type `float`. One dataset uses the default marshaler. The other dataset uses a custom marshaler called `MyCustomMarshaler`.

```
class MyChunkType : public GridMate::ReplicaChunk
{
    public:
    MyChunkType()
        : m_synchedFloat("SynchedFloat")
        , m_synchedHalf("SynchedHalf")
    {
    }

    GridMate::DataSet<float> m_synchedFloat;
    GridMate::DataSet<float, MyCustomMarshaler> m_synchedHalf;
};
```

Datasets can be optionally bound to a callback on the chunk interface so that the callback is called when new data arrives.

```
class MyChunkType : public GridMate::ReplicaChunk
{
public:
    MyChunkType()
        : m_synchedFloat("SynchedFloat")
    {
    }

    // Callback to call when new data arrives.
    void OnSynchedFloatData(const float& newValue, const GridMate::TimeContext&
 timeContext);

    GridMate::DataSet<float>::BindInterface<MyChunkType,
 amp;MyChunkType::OnSynchedFloatData> m_synchedFloat;
};
```

Eventual consistency is guaranteed for datasets. Normally, datasets propagate unreliably. To compensate for potential packet losses, and to minimize latency, GridMate handles events in the following order:

1. A user changes a value in the dataset.
2. The new value is broadcast to the remote peers.
3. The dataset stops changing.
4. A user-configurable grace period elapses.
5. A last update is sent reliably.
6. To conserve bandwidth, propagation is suspended until the next change.

You can change the length of the grace period in step 4 by calling `SetMaxIdleTime`:

```
...
GridMate::DataSet<Vector3> m_pos;
...

...
m_pos.SetMaxIdleTime(5.f);  // Suspend sending if m_pos has not changed for 5 ticks
...
```

Examples

The examples in this section show three different ways to create datasets.

Example 1

The following example creates a `DataSet` object that stores a `u32` value, using the default marshaler for `u32`.

```
GridMate::DataSet<AZ::u32> m_data;
```

Example 2

The following example creates a `DataSet` object that stores a float. The data written to the network is half float size because of the specified marshaler.

```
GridMate::DataSet<float, HalfMarshaler> m_data;
```

Example 3

The following example creates a `DataSet` object that stores an `s32` value using the default marshaler for `s32`. Whenever the `DataSet` value changes, the `DataSetHandler` function is called on the `MyReplicaChunk` instance. This is true for both master and proxy nodes; the event is triggered on local data changes for the master, and upon received data changes for the proxies.

```
class MyReplicaChunk : public GridMate::ReplicaChunk
{
    void DataSetHandler(const AZ::s32& value, const GridMate::TimeContext& context) { /*
 Data Changed Logic */ }
    GridMate::DataSet<AZ::s32>::BindInterface<MyReplicaChunk,
 &MyReplicaChunk::DataSetHandler> Data;
};
```

Throttlers

Datasets can be throttled based on an optional throttler parameter to the template. The throttler can choose to send data or withhold downstream updates unless a certain condition has been met. The throttler must implement the `WithinThreshold` method using the following syntax.

```
bool WithinThreshold(T previousValue, T currentValue);
```

The return value of the method determines whether to send the data to the proxy peers.

Remote Procedure Calls (RPCs)

RPCs allow games to send events or requests to remote nodes through replicas. They can be used to send messages to a specific node, or to route function calls to the authoritative node. For example, you can use RPCs to implement functions that change the position of an object. This ensures that changes happen only at the node that owns the object. For server-authoritative games, reliable RPCs can be used for sending frequent client input commands.

RPCs have the following characteristics:

- RPC arguments can be of any type, as long as a valid marshaler is provided.
- All RPC requests are routed to the master replica.
- The RPC handler function in the master replica chooses whether to propagate the RPC to proxy replicas.

- RPCs are not kept in the history, and late-joining clients might not receive RPCs requested before the client joined.

Like datasets, RPCs are declared as replica chunk members. An RPC handler function is bound to the RPC as part of the declaration. RPC requests are forwarded to the handler function along with the arguments and an `RpcContext` associated with the request.

The RPC handler function can perform additional checks before executing the request.

The handler for an RPC returns a Boolean value to GridMate. This value is used on the replica's master node to determine whether the RPC is propagated to all proxies.

Remote procedure calls are always invoked first on the master node for the replica. This is true whether the initial caller is a master or proxy. The master node's RPC handler decides whether the RPC should be propagated to the proxy nodes based on the return value of the RPC handler. The user returns `true` to mean "propagate to all replica proxies," and `false` to mean "only invoke this RPC on the master."

RPCs have a constructor that requires a string. This is used for debugging and statistical purposes. Any debugging or network monitoring exposes the given RPC name. Using modern C++, the name can also be specified inline, as in the following example.

```
Rpc<RpcArg<AZ::u32>>::BindInterface<MyClass,
&MyClass::Func> Rpc = {"My RPC"};
```

Examples

The following examples show how RPCs can be used in GridMate.

Example 1

In the following example, `Rpc1` is an RPC that takes a single parameter of type `u32`. It uses the default `u32` marshaler.

```
class MyReplicaChunk : public GridMate::ReplicaChunk
{
    bool Rpc1Handler(AZ::u32 val, const GridMate::RpcContext& context) { /* RPC Logic */ }
    GridMate::Rpc<GridMate::RpcArg<AZ::u32>>::BindInterface<MyReplicaChunk,
 &MyReplicaChunk::Rpc1Handler> Rpc1;
};
```

Example 2

In the following example, `Rpc2` is an RPC that takes a single parameter of type `s32`. It uses `IntegerQuantizationMarshaler`, with a range from `-100` to `100` and writes one byte to the wire.

```
class MyReplicaChunk : public GridMate::ReplicaChunk
{
    bool Rpc2Handler(AZ::s32 val, const GridMate::RpcContext& context) { /* RPC Logic */ }
    GridMate::Rpc<GridMate::RpcArg<AZ::s32, GridMate::IntegerQuantizationMarshaler<-100,
 100, 1>>>::BindInterface<MyReplicaChunk, &MyReplicaChunk::Rpc2Handler> Rpc2;
};
```

Example 3

In the following example, `Rpc3` is an RPC that takes two parameters; a `u8` and a string. It uses the default marshalers for each argument.

```
class MyReplicaChunk : public GridMate::ReplicaChunk
```

```
{
    bool Rpc3Handler(AZ::u8 val, const AZStd::string& str, const GridMate::RpcContext&
 context) { /* RPC Logic */ }
    GridMate::Rpc<GridMate::RpcArg<AZ::u8>, GridMate::RpcArg<const
 AZStd::string&>>::BindInterface<MyReplicaChunk, &MyReplicaChunk::Rpc3Handler> Rpc3;
};
```

Example 4

If you want to send a custom class as an RPC parameter, you must first write a marshaler for it, as in the following example.

```
struct MyClass
{
    AZ::Crc32 m_name;
    AZ::u32 m_value;
};

namespace GridMate
{
    template<>
    class Marshaler<MyClass>
    {
    public:
        static const AZStd::size_t MarshalSize = Marshaler<AZ::Crc32>::MarshalSize +
 sizeof(AZ::u32);

        void Marshal(WriteBuffer& wb, const MyClass& value) const
        {
            wb.Write(value.m_name);
            wb.Write(value.m_value);
        }
        void Unmarshal(MyClass & value, ReadBuffer& rb) const
        {
            rb.Read(value.m_name);
            rb.Read(value.m_value);
        }
    };
}
```

An RPC that passes a parameter of the foregoing class might be declared like this:

```
class MyReplicaChunk : public GridMate::ReplicaChunk
{
    bool Rpc4Handler(const MyClass& value, const GridMate::RpcContext& context) { /* RPC
 Logic */ }
    GridMate::Rpc<GridMate::RpcArg<const MyClass&>>::BindInterface<MyReplicaChunk,
 &MyReplicaChunk::Rpc4Handler> Rpc4;
};
```

For `Rpc4`, the first and only argument is a `const` reference to the `MyClass` object. The `const MyClass&` is specified to indicate that the `Rpc4Handler` function takes a `const` reference. This allows you to avoid making a copy of the object when it is passed to the handler function. Behind the scenes, GridMate stores a temporary value of `MyClass`, which is what the reference binds to. The temporary referent is removed after the RPC has been called. You can also use this technique to marshal objects that are wrapped in smart pointers.

Example 5

In order to invoke an RPC on a given chunk instance, you can simply call the RPC, as in the following example.

```
class MyReplicaChunk : public GridMate::ReplicaChunk
{
    bool Rpc5Handler(AZ::u32 val, const GridMate::RpcContext& context) { /* RPC Logic */ }
    GridMate::Rpc<GridMate::RpcArg<AZ::u32>>::BindInterface<MyReplicaChunk,
 &MyReplicaChunk::Rpc1Handler> Rpc5;
};

void Foo(MyChunkType* myChunkInstance)
{
    myChunkInstance->Rpc5(1);
}
```

`Rpc5` is an RPC that takes a single parameter of type `u32`. It uses the default `u32` marshaler. Calling `Foo` invokes the RPC on the replica chunk instance and passes in a value of `1`.

RPC Type Traits

RPCs have an optional `typetraits` parameter. The following traits are expected in the `traits` class.

Trait	Default Value	Description
`s_isReliable`	`true`	Uses reliable transmission to send the RPC.
`s_isPostAttached`	`true`	Forces any dirty datasets to also be sent reliably in advance. This is useful if the RPC relies on the data in the datasets to be up to date on the destination peer.

Replica Manager

The replica manager is a subsystem that is responsible for managing the synchronization of replicas. The replica manager is responsible for the following:

- Marshaling and unmarshaling the replicas in each peer
- Forwarding replicas from one peer to another
- Handling ownership changes of replicas
- Managing replica lifetimes

Managing Replica Lifecycle

The replica manager must do the following:

- Keep track of all replicas by holding a reference-counted pointer to every master and proxy replica object.
- Guarantee consistency across the session by capturing and propagating the last state of every replica before a replica is destroyed.
- Guarantee that all proxies reach eventual consistency before a replica is deactivated.
- Release all GridMate references to a replica object when the object has been destroyed.

Topics

- Binding a New Master Replica to Replica Manager (p. 995)
- Retrieving Replicas from Replica Manager (p. 995)
- How Replica Manager Updates Replicas (p. 995)

- Task Manager (p. 996)

Binding a New Master Replica to Replica Manager

After a new master replica is created, it must be bound to the replica manager as follows:

```
GridMate::ReplicaManager* replicaManager = session->GetReplicaMgr(); // Get replica manager
 from the current session
replicaManager->AddMaster(myReplica1); // Bind replica to replica manager
replicaManager->AddMaster(myReplica2); // Bind replica to replica manager
```

Proxy replicas are bound to their session's replica managers automatically. Each `ReplicaManager` instance holds a reference to every replica that is bound to it. That changes only when the user calls `Destroy()` on the replica or when the `ReplicaManager` itself is destroyed.

Retrieving Replicas from Replica Manager

Every replica has a numeric identifier that is unique in the session. To find a replica by its ID, invoke `FindReplica(<ReplicaId>)`, as in the following example:

```
GridMate::ReplicaPtr replica = replicaManager->FindReplica(<myReplicaId>);
AZ_Assert(replica != nullptr, "Replica with id=%d not found.", <myReplicaId>);
```

How Replica Manager Updates Replicas

The GridMate session triggers the replica manager to perform replica updates on a continuous basis. These updates include the following actions:

- Unmarshaling
- Update from replica
- Update replicas
- Marshaling

Marshaling: Sending Data to Other Peers

Changes in a replica must be replicated to every remote peer in the GridMate session. To communicate a change in one of its replicas, a peer's replica manager serializes the replica object into a send buffer. It then sends the object to the network. Replica marshaling occurs in two main phases:

- **Data Preparation** – A premarshaling phase that, based on changes in the replica, determines which RPCs and `DataSet` objects to send. This phase also validates the data integrity of the objects to be sent.
- **Actual Marshaling** – The transformation of a replica object into a byte stream. The actual data that must be marshaled depends on how much new information the master replica has relative to its corresponding remote proxy replica. For example, new proxy replicas require all information about the master replica. This includes its datasets (p. 989), RPCs (p. 991), and construction metadata. Previously synchronized proxy replicas require only the information from the master replica that is different, including any pending RPC calls.

Unmarshaling: Receiving Data from Other Peers

In unmarshaling, the replica manager communicates with the remote peers, receives and parses new data from them, and updates its own replicas accordingly. These updates can include accepting new peers, instantiating new proxy replicas, handling ownership changes, or destroying proxy replicas.

Note
For more information about marshaling, see Marshalling (p. 972).

Update from Replica: Updating Proxy Replicas

A change in a custom ReplicaChunk (p. 987) results in an `UpdateFromChunk` callback that causes all proxy replicas to update their state. RPCs from proxy and master replicas are processed and invoked during this step.

Update Replicas: Updating Master Replicas Locally

A change in a custom replica chunk results in an `UpdateChunk` callback that causes all master replicas on a local peer to update their states.

Task Manager

The replica manager holds two task manager instances: one for updating and one for marshaling replicas. Updating tasks are executed within the replica manager's `UpdateFromReplica` step, while marshaling tasks are executed in the `Marshal` step. Tasks can execute other tasks while running. `TaskManager::Add` queues the tasks in an ordered list. `TaskManager::Wait` executes a task and waits until it finishes. When an event fires in the replica system, replica manager adds the corresponding task into `TaskSystem`.

Here are few examples of this behavior:

- A user changes a dataset's value within a replica. The change needs to be marshaled to other peers. The `OnReplicaChanged` event is called on `ReplicaManager`, and `ReplicaMarshalTask` is queued for execution. Because replicas must be sent in the order of their creation, the task's priority is based on the replica's creation time. The queued task is executed at the appropriate time within the `Marshal` step.
- A new proxy replica is unmarshaled. When this happens, `OnReplicaUnmarshaled()` is called and `ReplicaUpdateTask` is queued. This task's priority is always zero because the order of execution is not important. `UpdateFromReplica` is called to notify the user of the new replica's data.

Using Lumberyard Networking

Lumberyard's GridMate networking system offers many ways to improve your game:

- Synchronize game state by using components or scripts.
- Use encryption for enhanced security.
- Control bandwidth.
- Take advantage of powerful features that enable you to create a professional-grade game.

Topics

- Synchronizing Game State Using Components (p. 997)
- Synchronizing Game State Using Scripts (p. 1002)
- Using Encyrption (p. 1002)
- HttpRequestor Gem (p. 1007)
- Controlling Bandwidth Usage (p. 1010)
- Setting up a Lobby (p. 1012)
- Creating Dedicated Servers (p. 1012)

- Using Amazon GameLift (p. 1012)
- Useful Console Commands (p. 1013)

Synchronizing Game State Using Components

The network binding API in the AZ framework provides a way for components to synchronize their states over the network.

To enable network synchronization for a component, you must do the following:

1. Derive the component from `NetBindable` and implement the network binding interfaces.
2. Implement a new replica chunk type and add the datasets and RPCs necessary to provide synchronization.

Topics

- Synchronizing an Entity with a NetBindingComponent (p. 997)
- Binding Process on Remote Nodes (p. 997)
- Unbinding Process (p. 997)
- NetBindable Component Flexibility (p. 997)
- Entity IDs (p. 998)
- Creating a NetBindable Component (p. 998)

Synchronizing an Entity with a NetBindingComponent

Because a special `NetBindingComponent` is responsible for the actual binding process, entities that need to be synchronized must have a `NetBindingComponent` added to them. When a game enters a multiplayer session, the `NetBindingComponent` collects replica chunks from the `NetBindable` instances on the entity and adds them to a Replica master. A special `NetBindingChunk` captures and stores spawning and other binding information for the entity. `NetBindingComponent` instances activated during a multiplayer session automatically start the binding process.

Binding Process on Remote Nodes

As replicas arrive at remote nodes, `NetBindingChunk` starts the entity spawning and binding process on the remote node. The binding process is completely asynchronous. The replicas become active first. Then an entity spawn request is queued. After the entity becomes available, its `NetBindable` components are bound to their corresponding chunks. Finally, the entity is activated.

Unbinding Process

When replicas are removed, affected `NetBindingComponent` instances start the unbinding process. By default, entities that are unbound from proxy replicas are deleted, but this doesn't have to be always the case. A game can choose to keep all entities in place and seamlessly switch to single-player mode.

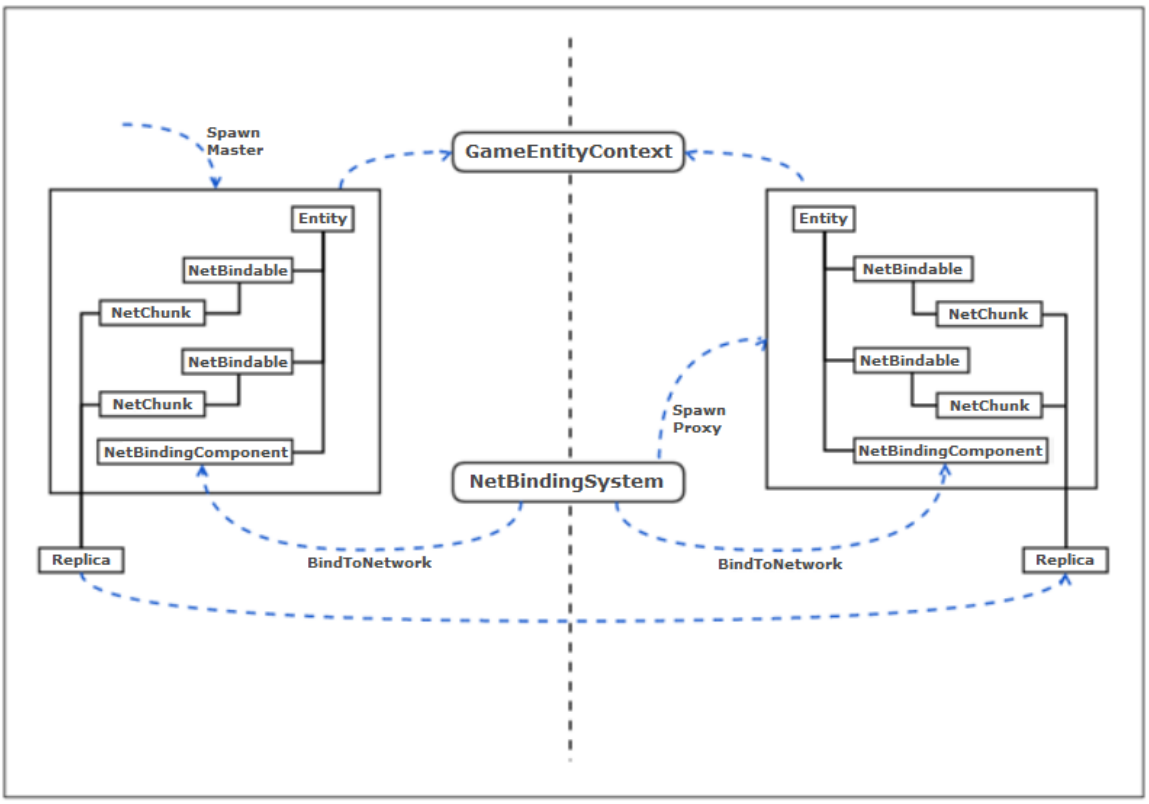
NetBindable Component Flexibility

A `NetBindingComponent` must exist for an entity to be bound to the network. This allows `NetBindable` components to be used in single-player modes without any additional runtime cost. `NetBindable` instances can also be disabled for each instance. This gives you the additional flexibility: The transform component can provide entity transform synchronization by default, but for special entities, a physics or animation component can provide more advanced synchronization.

Entity IDs

In Lumberyard, every entity has a unique ID so that it can be referenced in the game. Entity IDs are 64-bit strings generated using an algorithm that ensures uniqueness across computing devices. To reduce binding complexity, the net binding system spawns entities to be bound to proxy replicas using the same ID as the master.

The following diagram shows how the net binding system binds an entity to the network and spawns an entity. It does this with the same ID that it binds to a proxy replica.



Creating a NetBindable Component

For a Lumberyard component to share data on the network, it must include the `NetBindingComponent`. The `NetBindingComponent` creates a replica (p. 984) for the component and can bind any replica chunk (p. 987) that a component creates to the replica.

To enable networking on a component

1. Inherit the component from `AzFramework::NetBindable`:

```
#include <AzFramework/Network/NetBindable.h>
class ShipComponent
: public Component
, public AzFramework::NetBindable
```

2. Modify the `AZ_COMPONENT` definition to include `AzFramework::NetBindable`:

```
AZ_COMPONENT(ShipComponent,"{D466FD68-96C9-45AF-8A89-59402B0350F7}",
 AzFramework::NetBindable);
```

3. Modify `SerializeContext` to include `AzFramework::NetBindable`:

```
if (serialize)
{
serializeContext->Class<ShipComponent, AzFramework::NetBindable, AZ::Component>()
      ...
}
```

4. Implement the `AzFramework::NetBindable` interfaces:

```
// Called during network binding on the master. Implementations should create and
 return a new binding.
virtual GridMate::ReplicaChunkPtr GetNetworkBinding() = 0;

// Called during network binding on proxies.
virtual void SetNetworkBinding(GridMate::ReplicaChunkPtr chunk) = 0;

// Called when network is unbound. Implementations should release their references to
 the binding.
virtual void UnbindFromNetwork() = 0;
```

Notes

- If the `AZ_COMPONENT` definition change is missing, the `NetBindingComponent` does not recognize the component when it checks for components to add to the replica.
- If the `SerializeContext` definition is missing, the master replica still functions correctly. However, the proxy cannot match the IDs because it is not serialized as an `AzFramework::NetBindable` interface.
- Changes to these definitions require a re-export of levels for the static IDs to match correctly.

Network Binding Function Details

The following functions are available for working with component entities on the network.

GetNetworkBinding

The component uses this function to create its `ReplicaChunk` and initialize any state it wants to synchronize across the session. This function is called only on the master `ComponentEntity`. The `ReplicaChunk` that is returned is automatically attached to the appropriate `Replica`.

SetNetworkBinding

This function passes a `ReplicaChunk` to the component and initializes the internal data of the component to match that of the `ReplicaChunk`. This function is called only on the proxy `ComponentEntity` instances that are already bound to an appropriate `Replica`.

UnbindFromNetwork

The `UnbindFromNetwork` function is called to stop the component from reacting to data updates from the network. This can happen, for example, when the master no longer exists, has been deactivated, or has relinquished control to the local source.

Creating a Chunk

After you have enabled the `NetBindable` interface on the *component*, you must create a `ReplicaChunk` object that will store any state that the component wants to share.

```
class ShipComponentReplicaChunk : public GridMate::ReplicaChunkBase
```

```
{
public:
    AZ_CLASS_ALLOCATOR(ShipComponentReplicaChunk, AZ::SystemAllocator, 0);

    static const char* GetChunkName() { return "ShipComponentReplicaChunk"; }

       ShipComponentReplicaChunk()
        : SetFiring("SetFireLaser")
        , m_playerEntityId("PlayerEntityId")
    {
    }

    bool IsReplicaMigratable()
    {
        return true;
    }

    GridMate::Rpc< GridMate::RpcArg<bool> >::BindInterface<ShipComponent,
 &ShipComponent::SetFiringRPC, NetworkUtils::ShipControllerRPCTraits> SetFiring;
    GridMate::DataSet<AZ::EntityId>::BindInterface<ShipComponent,
 &ShipComponent::OnNewNetPlayerEntityId> m_playerEntityId;
};
```

Note
You must reflect this new replica chunk's datasets and RPCs in the component's `Reflect` function.

```
AzFramework::NetworkContext* netContext =
 azrtti_cast<AzFramework::NetworkContext*>(context);

if (netContext)
{
    netContext->Class<ShipComponent>()
        ->Chunk<ShipComponentReplicaChunk>()
        ->RPC<ShipComponentReplicaChunk, ShipComponent>("SetFireLaser",
 &ShipComponentReplicaChunk::SetFiring)
        ->Field("PlayerEntityId", &ShipComponentReplicaChunk::m_playerEntityId)
        ;
}
```

In order for the component to react to a change in the `DataSet` object, one of the following must occur:

- The replica chunk must signal to the component when the change occurs (in the example, this is done using the `BindInterface` extension to `DataSet`).
- The component must poll the replica chunk and check the `DataSet` object for changes.

Example: Filling Out the AzFramework::NetBindable Interface

The examples below illustrate the use of `GetNetworkBinding`, `SetNetworkBinding` and `UnbindFromNetwork`.

GetNetworkBinding

In the following example, the component creates the new replica chunk and initializes the data to be networked. This function is called by the master replica to retrieve the binding from the component.

```
GridMate::ReplicaChunkPtr ShipComponent::GetNetworkBinding()
{
    ShipComponentReplicaChunk* replicaChunk =
 GridMate::CreateReplicaChunk<ShipComponentReplicaChunk>();
```

```
    replicaChunk->SetHandler(this);
    m_replicaChunk = replicaChunk;

    return m_replicaChunk;
}
```

SetNetworkBinding

In the following example, the component is bound to the supplied replica chunk. It also relinquishes its local state to the state specified by the replica chunk. This function is called on proxies to hand their binding over to the component.

```
void ShipComponent::SetNetworkBinding(GridMate::ReplicaChunkPtr chunk)
{
    chunk->SetHandler(this);
    m_replicaChunk = chunk;

    ShipComponentReplicaChunk* shipControllerChunk =
 static_cast<ShipComponentReplicaChunk*>(m_replicaChunk.get());
    SetPlayerEntityIdImpl(shipControllerChunk->m_playerEntityId.Get());
}
```

UnbindFromNetwork

```
void ShipComponent::UnbindFromNetwork()
{
    m_replicaChunk->SetHandler(nullptr);
    m_replicaChunk = nullptr;
}
```

Maintaining State

The last step is to create checks to make sure that any local modifications to the preferred networkable state do not overwrite the networked state. In addition, you must update the replica chunk whenever the local state changes and the component is in control of the state.

```
void ShipComponent::OnNewNetPlayerEntityId(const AZ::EntityId& playerEntityId, const
 GridMate::TimeContext& tc)
{
    (void)tc;
    SetPlayerEntityIdImpl(playerEntityId);
}

bool ShipComponent::SetFiringRPC(bool firing, const GridMate::RpcContext& rpcContext)
{
    if (AllowRPCContext(rpcContext))
    {
        SetFiring(firing);
    }

    return false;
}

// Component implementation of to set firing
void ShipComponent::SetFiring(bool firing)
{
    m_isFiring = firing;

    if (!AzFramework::NetQuery::IsEntityAuthoritative(GetEntityId()))
    {
```

```
        // If the ship component is not authoritative, send an RPC update to the replica
 chunk
        ShipComponentReplicaChunk* shipChunk =
 static_cast<ShipComponentReplicaChunk*>(m_replicaChunk.get());
        shipChunk->SetFiring(firing);
    }
    else
    {
        if (m_isFiring)
        {
            EBUS_EVENT_ID(GetGun(), ShipGunBus, StartFire);
        }
        else
        {
            EBUS_EVENT_ID(GetGun(), ShipGunBus, StopFire);
        }
    }
}

void ShipComponent::SetPlayerEntityIdImpl(AZ::EntityId playerEntityId)
{
    AZ_Error("ShipControllerComponent",!m_playerEntityId.IsValid() || !
playerEntityId.IsValid(),"Trying to rebind an already bound ship");
    if (m_playerEntityId != playerEntityId)
    {
        m_playerEntityId = playerEntityId;
        HandleShipSetup();

        if (m_replicaChunk && AzFramework::NetQuery::IsEntityAuthoritative(GetEntityId()))
        {
            // If you are authoritative over the entity and the component is replicated,
 update the value of the DataSet and propagate to clients
            ShipComponentReplicaChunk* shipChunk =
 static_cast<ShipComponentReplicaChunk*>(m_replicaChunk.get());
            shipChunk->m_playerEntityId.Set(m_playerEntityId);
        }
    }
}
```

Synchronizing Game State Using Scripts

You can synchronize game state by using the Lua `ScriptComponent`. The initial steps of synchronizing game state using the Lua `ScriptComponent` are similar to any other component. There are two main steps:

1. You must add a `NetBindingComponent` to the definition of the entity that contains the script and the `ScriptComponent` and whose state you want to synchronize.
2. Inside the script, any properties that need to be synchronized must be tagged accordingly. For more information, see Network Binding (p. 580)Network Binding in the Writing Lua Scripts (p. 574) topic.

When these steps are completed, your game state data should synchronize correctly.

Using Encyrption

GridMate uses the OpenSSL implementation of Datagram Transport Layer Security (DTLS) to support encryption of all UDP traffic sent between clients and servers.

Limitations

GridMate's implementation of encryption has the following limitations:

- Only 64-bit Windows is supported.
- Only client-server topologies are supported.

Implementation Support

GridMate supports encryption for the following implementations:

- Server and client authentication
- Self-signed certificates
- A single strong OpenSSL cipher

Cipher

GridMate uses the following single OpenSSL cipher for all encrypted connections: `ECDHE-RSA-AES256-GCM-SHA384`.

This cipher uses the technologies listed in the following table:

Cipher Technologies in GridMate

Technology	Role	Description
ECDHE	Master key exchange	Ephemeral Elliptic Curve Diffie-Hellman anonymous key agreement protocol
RSA	Peer authentication	RSA algorithm used to authenticate client and server
AES256	Symmetric encryption cipher	Advanced Encryption Standard that uses a 256-bit master key
GCM	Block cipher mode of operation	Galois/Counter Mode authenticated encryption algorithm
SHA384	Hashing algorithm	SHA-2 with a 384-bit digest size

Topics

- Building with Encryption (p. 1003)
- Enabling Encryption (p. 1004)

Building with Encryption

When you include the GridMate library in your project, encryption support is automatically provided. However, because the GridMate library is statically linked, you must first make some modifications to the WAF build script (wscript) that uses GridMate.

Building Your Project with Encryption

To use encryption with GridMate, you must modify your `.wscript` file to add a dependency on GridMate, link the OpenSSL library, and specify OpenSSL library paths.

To modify your .wscript file to use OpenSSL with GridMate

1. Add the following line to create a dependency on GridMate:

```
use = ['GridMate']
```

2. Add the following line to link the OpenSSL library:

```
win_lib = ['ssleay32', 'libeay32']
```

3. Add the OpenSSL library paths, as in the following example. Within the Lumberyard install directory, these paths are in the folder `dev\Code\SDKs\OpenSSL\lib\`:

```
win_x64_debug_libpath = [ bld.Path('Code/SDKs/OpenSSL/lib/vc120_x64_debug') ],
win_x64_profile_libpath = [ bld.Path('Code/SDKs/OpenSSL/lib/vc120_x64_release') ],
win_x64_release_libpath = [ bld.Path('Code/SDKs/OpenSSL/lib/vc120_x64_release') ],
win_x64_debug_dedicated_libpath = [ bld.Path('Code/SDKs/OpenSSL/lib/
vc120_x64_debug') ],
win_x64_profile_dedicated_libpath = [ bld.Path('Code/SDKs/OpenSSL/lib/
vc120_x64_release') ],
win_x64_release_dedicated_libpath = [ bld.Path('Code/SDKs/OpenSSL/lib/
vc120_x64_release') ]
```

Building Without Encryption

If your project uses GridMate, but does not require support for encryption, ensure that the `GridMateForTools` line is in your `.wscript` file:

```
use = ['GridMateForTools']
```

Enabling Encryption

To enable encryption with OpenSSL in a GridMate session, perform the following steps.

To enable encryption in a GridMate session

1. To set the encryption parameters, create an instance of `SecureSocketDesc`. The parameters are described in SecureSocketDesc (p. 1004).
2. Create an instance of `SecureSocketDriver` that passes in the instance of `SecureSocketDesc`. The instance of `SecureSocketDesc` must be available for the duration of the GridMate session.
3. Before hosting or joining a GridMate session, define `CarrierDesc` by setting the `CarrierDesc::m_driver` property to the instance of `SecureSocketDriver`. If no instance of `SecureSocketDriver` is provided, an unencrypted driver is used that provides plaintext communication.
4. You can delete the `SecureSocketDriver` instance at the end of the GridMate session, ideally in the `OnSessionDelete` event on the `SessionEventBus`.

The GridMate Session Encryption Example (p. 1006) at the end of this topic has sample code for these steps.

SecureSocketDesc

The constructor for `SecureSocketDriver` requires a `SecureSocketDesc` object that provides all encryption configuration required for the secure connection. The configuration parameters are described in the following table.

SecureSocketDesc Configuration Parameters

Parameter	Description
`m_privateKeyPEM`	Base-64 encoded string PEM private key.
`m_certificatePEM`	Base-64 encoded string PEM public certificate.
`m_certificateAuthorityPEM`	Base-64 encoded string PEM certificate authority.
`m_authenticateClient`	If set to 1, the client is expected to provide a signed certificate for authentication. To implement this, `m_certificatePEM` must be set on the client, and the server needs to set up `m_certificateAuthorityPEM`. The default setting is 0.

Server Authentication Only

You can use the server authentication only configuration when the client needs to verify the authenticity of the server to which it connects. The server has a secret private key and a public certificate signed by a certificate authority. This is the most common configuration.

Server Authentication Only Configuration

Role	Parameters
Client	`m_certificateAuthorityPEM`
Server	`m_privateKeyPEM`, `m_certificatePEM`, `m_certificateAuthorityPEM`

Client and Server Authentication

Use this configuration when the client must verify authenticity of the server and the server must verify authenticity of the client. The client has its own unique private key and corresponding signed public certificate. The server has its own unique private key and corresponding signed public certificate.

It's possible to share or use the same certificate authority for both, but keys and certificates must be unique to each peer.

Client and Server Authentication Configuration

Role	Parameters
Client	`m_privateKeyPEM`, `m_certificatePEM`, `m_certificateAuthorityPEM`
Server	`m_privateKeyPEM`, `m_certificatePEM`, `m_certificateAuthorityPEM`

Self-signed Certificates

You can use self-signed certificates for development purposes.

> **Warning**
> Do not use self-signed certificates for production environments.

When you use self-signed certificates, there is no certificate authority to sign the public certificates. To permit the absence of a certificate authority, set `m_certificateAuthorityPEM` to the same value as `m_certificatePEM`.

GridMate Session Encryption Example

The following code snippet enables encryption in a GridMate session.

```
class MyClass : public GridMate::SessionEventBus::Handler
{
public:
  void OnSessionDelete(GridMate::GridSession* session) override;

private:
  GridMate::SecureSocketDriver* m_secureDriver;
};

void MyClass::JoinSession() {
  // ...

  // Create an instance of SecureSocketDesc and set its encryption parameters.
  GridMate::SecureSocketDesc secureDesc;
  secureDesc.m_privateKeyPEM = "..."
  secureDesc.m_certificatePEM = "..."
  secureDesc.m_certificateAuthorityPEM = "..."

  // Create an instance of SecureSocketDriver that passes in the instance of
  // SecureSocketDesc.
  m_secureDriver = new GridMate::SecureSocketDriver(secureDesc);

  // Before hosting or joining a GridMate session, set the CarrierDesc::m_driver
  // property to the instance of SecureSocketDriver.
  carrierDesc.m_driver = m_secureDriver;

  // ...
}

  // At the end of the GridMate session, delete the SecureSocketDriver instance in
  // the OnSessionDelete event.
void MyClass::OnSessionDelete(GridMate::GridSession* session) {
  // ...

  delete m_secureDriver;
  m_secureDriver = nullptr;

  // ...
}
```

How To Generate a Private Key and Public Certificate

You can use the `openssl req` command to generate a self-signed certificate from OpenSSL, as in the following example.

```
dev/Code/SDKs/OpenSSL/bin/openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem
 -days 365 -nodes
```

The arguments are as follows.

- `-x509` – The certificate format.
- `-newkey` – The type of key. This example generates an RSA key with 2048 bits.
- `-keyout` – The name of the key PEM file that will be generated
- `-out` – The name of the cert PEM file that will be generated.

Upon execution, the command prompts for additional user input required to generate the certificate.

HttpRequestor Gem

You can use the HttpRequestor Gem to make asynchronous HTTP/HTTPS requests and return data through a user-provided call back function. This gem uses EBuses (p. 543) for communication and provides all requests asynchronously.

Note
This feature is only supported on Windows and replaces `CryAction::HttpRequest`.

Getting Started

In order to use the HttpRequestor Gem, you must enable it in your project.

To enable the HttpRequestor Gem

1. In Project Configurator, find your project in the list, and then click **Enable Gems**.
2. In the list of gems, select the checkbox next to **HttpRequestor**.
3. Click **Save**.
4. Type the following command to rebuild your project.

```
lmbr_waf configure
```

C++ API Using EBUS_EVENT

The HttpRequestor Gem has separate sets of APIs for adding requests and adding text requests.

AddRequest, AddRequestWithHeaders, AddRequestWithHeadersAndBody

You can use the `AddRequest`, `AddRequestWithHeaders`, and `AddRequestWithHeadersAndBody` APIs to send generic HTTP requests to any website and receive the return data in JSON format. The methods return the data received in the `callback` parameter.

Syntax

```
EBUS_EVENT(HttpRequestor::HttpRequestorRequestBus,AddRequest, URI, method, callback)
```

```
EBUS_EVENT(HttpRequestor::HttpRequestorRequestBus, AddRequestWithHeaders, URI, method,
 headers, callback)
```

```
EBUS_EVENT(HttpRequestor::HttpRequestorRequestBus, AddRequestWithHeadersAndBody, URI,
 method, headers, body, callback)
```

Each of the add request methods requires the URI, a method and a callback.

Parameters

Paramet	Type	Description
`URI`	`AZStd::String`	The fully qualified web address, in the form of `scheme://[user:password@]host[:port]][/]path[?query][#fragment]`

Paramet	Type	Description
method	Aws::Http::HttpMethod	The method type, only supported values are HTTP_GET, HTTP_POST, HTTP_DELETE, HTTP_PUT, HTTP_HEAD, and HTTP_PATCH.
callback		This function is called when the HTTP request has completed. The response body and code are present in the callback.
headers	HttpRequestor::Headers	The list of header fields for the HTTP request.
body	AZStd::String	Optional body to send with the request.

Return: No return value.

JSON Request Callback

This callback is returned for the AddRequest, AddRequestWithHeaders, and AddRequestWithHeadersAndBody methods.

```
void Callback(const Aws::Utils::Json::JsonValue& json, Aws::Http::HttpResponseCode
 responseCode);
```

Parameters

Parameter	Type	Description
json	Aws::Utils::Json::JsonValue	The JSON object. The life span of this object is valid only during the scope of the callback.
responseCode	Aws::Http::HttpResponseCode	The HTTP response code.

Return: No return value.

AddTextRequest, AddTextRequestWithHeaders, AddTextRequestWithHeadersAndBody

You can use the AddTextRequest, AddTextRequestWithHeaders, and AddTextRequestWithHeadersAndBody APIs to send a generic HTTP request to any website and receive the return data in a text string. The methods return the data received in the callback parameter.

Syntax

```
EBUS_EVENT(HttpRequestor::HttpRequestorRequestBus, AddTextRequest, URI, method, callback)
```

```
EBUS_EVENT(HttpRequestor::HttpRequestorRequestBus, AddTextRequestWithHeaders, URI, method,
 headers, callback)
```

```
EBUS_EVENT(HttpRequestor::HttpRequestorRequestBus, AddTextRequestWithHeadersAndBody, URI,
 method, headers, body, callback)
```

Each of the add text request methods requires the URI, a method and a callback.

Parameters

Paramet	Type	Description
URI	AZStd::String	The fully qualified web address, in the form of scheme://[user:password@]host[:port]][/]path[?query][#fragment]
method	Aws::Http::HttpMethod	The method type. Supported values are HTTP_GET, HTTP_POST, HTTP_DELETE, HTTP_PUT, HTTP_HEAD, and HTTP_PATCH.
callback		This function is called when the HTTP request has completed. The response body and code are present in the callback.
headers	HttpRequestor::Headers	The list of header fields for the HTTP request.
body	AZStd::String	Optional body to send with the request.

Return: No return value.

Text Request Callback

This callback is returned for AddTextRequest, AddTextRequestWithHeaders, AddTextRequestWithHeadersAndBody methods.

```
void Callback(const AZStd::string& response, Aws::Http::HttpResponseCode responseCode);
```

Parameters

Parameter	Type	Description
response	AZStd::string&	The text returned from the server. The life span of this object is valid only during the scope of the callback.
responseCode	Aws::Http::HttpResponseCode	The HTTP response code.

Return: No return value.

Example

The following example uses the Metastream C++ API to to obtain a gateway IP address.

```
EBUS_EVENT(HttpRequestor::HttpRequestorRequestBus, AddRequest, "https://httpbin.org/ip",
 Aws::Http::HttpMethod::HTTP_GET, [this](AXStd::string && data, Aws::Http::HttpResponseCode
 code)
{
     AZStd::string resultData = std::move(data);
     resultCode = code;

    /* Upon success:
    **  resultCode = Aws::Http::HttpResponceCode::Ok
    **  resultData = "{"origin": "xxx.xxx.xxx.xxx"}"  /* Your IP will be displayed */
    */
 }));
```

Controlling Bandwidth Usage

GridMate (p. 951) provides several ways to control the bandwidth that your game uses, including bandwidth throttling and the prioritization of replica (p. 984) updates.

Controlling the Send Rate

You can use GridMate to control the server send rate, which is a common technique for reducing bandwidth usage in multiplayer games. In this scenario, a multiplayer game is hosted by a dedicated server to which clients send their replica changes at their default rate (for example, 60 frames per second). To reduce bandwidth usage, you lower the server send rate (for example, to 20 transmissions per second). To avoid jitter when this technique is used, the client must be able to interpolate the game state that it receives from the server.

To control the server send rate in GridMate, set the time interval for replica data transmissions:

```
ReplicaMgr* replicaManager = session->GetReplicaMgr(); // Get the replica manager instance.
 This assumes the session has been established.
replicaManager->SetSendTimeInterval(100); // Set the send interval to 100 milliseconds. 10
 updates per second will be sent.
```

Setting the `SetSendTimeInterval` to `0` sends the data at the engine's frame rate. The default is `0`.

Bandwidth Limiter

Another technique is to limit outgoing bandwidth in exchange for increased latency in the replication of objects. In GridMate, you can do this by setting a bandwidth limit on replica manager. To do so, specify a byte limit for `SetSendLimit`, as in the following example:

```
ReplicaMgr* replicaManager = session->GetReplicaMgr(); // Get the replica manager instance.
 This assumes the session has been established.
replicaManager->SetSendLimit(10000); // Set the transmission limit to 10 kilobytes per
 second.
```

Setting `SetSendLimit` to `0` disables the bandwidth limiter. The default is `0`.

Controlling Burst Length

You can use the GridMate limiter to accommodate short bursts in bandwidth if your bandwidth usage is not already at its maximum. This can be useful in many game applications. For example, when a user is in a multiplayer lobby, the corresponding bandwidth usage is quite low. However, when the user joins the game, the bandwidth usage spikes as the initial game state replicates from the server to the client. To control the length of the burst permitted, specify the desired number of seconds for `SetSendLimitBurstRange`, as in the following example:

```
ReplicaMgr* replicaManager = session->GetReplicaMgr(); // Get the replica manager instance.
 This assumes the session has been established.
replicaManager->SetSendLimitBurstRange(5.f); // Set the maximum permitted length of the
 burst to 5 seconds.
```

Bursts in bandwidth usage are allowed for the number of seconds specified, after which the bandwidth is capped to the value set by `SetSendLimit`. The default value for `SetSendLimitBurstRange` is 10 seconds. If bandwidth usage has already reached its limit when the burst occurs, bandwidth usage continues to be capped, and the `SetSendLimitBurstRange` setting has no effect.

Prioritization of Replica Updates

Every replica chunk (p. 987) has a priority that you can assign. The priority is represented by an integer from `0` through `65534`. Larger integers represent higher priorities. Replicas with higher priorities are sent first. The default is `32768`.

This prioritization is especially important when you use the bandwidth limiter because you can use it to define which objects are more important and which are less important. If your game has a bandwidth cap and you have prioritized your replicas appropriately, the objects with higher priority are sent more often. The objects with lower priority are sent only when there is enough bandwidth to accommodate them.

For convenience, GridMate provides five predefined priorities that you can use for custom replica chunks:

```
static const ReplicaPriority k_replicaPriorityHighest = 0xFFFE; // Decimal 65534, highest
 priority.

static const ReplicaPriority k_replicaPriorityHigh = 0xC000;    // Decimal 49152, high
 priority.

static const ReplicaPriority k_replicaPriorityNormal = 0x8000;  // Decimal 32768, normal
 priority. This is the Default.

static const ReplicaPriority k_replicaPriorityLow = 0x4000;     // Decimal 16384, low
 priority.

static const ReplicaPriority k_replicaPriorityLowest = 0x0000;  // Decimal 0, lowest
 possible priority.
```

By default, all chunks have normal priority (`k_replicaPriorityNormal`). You can use these predefined priorities as is, or use them to create your own, as in the following example:

```
// A replica chunk with this priority will be sent before all the chunks with Normal
 priority, but after all the chunks with High priority:
const ReplicaPriority k_myCustomPriority = (k_replicaPriorityNormal
 + k_replicaPriorityHigh) / 2; // (=Decimal 40960)
```

The priority for the whole replica is the maximum priority found in its chunks. Priority for a chunk can be set after the chunk is created, or at any point during its lifetime, as in the following example:

```
MyChunk::Ptr myChunk = GridMate::CreateReplicaChunk<MyChunk>(...);
myChunk->SetPriority(k_replicaPriorityLow); // Sets low priority for myChunk.
```

Chunks with the same priority are sent and received in the order of their creation. Replicas created earlier are sent and received first.

Tuning Bandwidth at Runtime

You can tune bandwidth usage while the game is running by using the following configuration variables (CVars):

CVar	Description
`gm_replicasSendTime`	The time, in milliseconds, between replica transmissions. A value of 0 binds the interval to the GridMate tick rate.

CVar	Description
gm_replicasSendLimit	The limit, in bytes, of the amount of replica data that can be sent per second. A value of 0 disables the limit.
gm_burstTimeLimit	The time, in seconds, that bursts in bandwidth are allowed. Bursts are allowed only if the bandwidth is not capped when the burst occurs.

Setting up a Lobby

By default, the Lumberyard engine does not provide any specific lobby implementation, but instead provides the code interface required to construct one. The Multiplayer Gem does, however, provide some useful constructs that aid in lobby creation using flow graph nodes, and a basic lobby implementation using Components, that can be used as is, or as a reference.

For more information, see the Multiplayer Gem Documentation.

Creating Dedicated Servers

In Lumberyard, you can create a build of your game that runs on a headless dedicated server. A dedicated server build omits the packages and modules that a headless server does not require and has a smaller compiled size.

Resources to Include

The package that you build for the dedicated server must include the resources that are required to run the game on the client. You can use Lumberyard's asset pipeline to exclude certain resources that are not needed on the server. For more information, see Configuring the Asset Pipeline.

Client and Server

Amazon GameLift splits sessions between client and server, but Lumberyard does not. Lumberyard recommends that you not compile critical server logic on clients or include client data in the server package.

Waf Build Configuration

To generate a dedicated server build that you can use locally or upload to Amazon GameLift, attach the `_dedicated` label to your build configuration in Waf. For more information on using Waf to set up the dedicated server, see Creating a Dedicated Server in the Amazon Lumberyard User Guide. The topic is part of the Multiplayer Sample, which includes an example of a dedicated server.

Linux Support

Lumberyard has a Linux version of the dedicated server. For information on how to compile and build dedicated server packages in Linux, see Linux Support.

Using Amazon GameLift

Lumberyard supports hosting dedicated servers on the cloud by using Amazon GameLift. Amazon GameLift is a managed AWS service for deploying, operating, and scaling session-based multiplayer games. Amazon GameLift is built on AWS's highly available cloud infrastructure and allows you to quickly scale high-performance game servers up and down to meet player demand – without any additional engineering effort or upfront costs. It reduces the time required to build a multiplayer backend from thousands of hours to just minutes.

To use GameLift in your project, there are two options:

- Enable the GameLift Gem in your project. Lumberyard has integrated Amazon GameLift, which makes it easier for you to use GameLift.
- Enable the Lumberyard Multiplayer Gem in your project (which requires the GameLift Gem).

For information about configuring GameLift for the multiplayer sample, see Configuring the Multiplayer Sample for Amazon GameLift (p. 959). For information about gems, see Gems in the Amazon Lumberyard User Guide. For more information about GameLift, see Amazon GameLift.

Additional Links

- Tutorial: Creating and connecting to a game session (pdf)
- Amazon GameLift - Creating game sessions and connecting (video)
- Amazon GameLift Developer Guide
- Amazon GameLift API Reference

Useful Console Commands

Use the following commands in Lumberyard when working with a network server.

`gm_debugdraw debug_draw_level`

Sets the debug draw level. Accepts as a parameter a number whos bits represent the flags for the debug data to draw. For example, when set to 1, displays an overlay with GridMate network statistics and information.

The available bit flags come from the enum `DebugDrawBits` and are as follows:

```
enum DebugDrawBits
        {
            Basic           = BIT(0),
            Trace           = BIT(1),
            Stats           = BIT(2),
            Replicas        = BIT(3),
            Actors          = BIT(4),
            EntityDetail    = BIT(5),

            Full            = Basic | Trace | Stats | Replicas | Actors,
            All             = 0xffffffff,
        };
```

`gm_disconnectDetection`

When set to 0, disables disconnect detection. This is useful when you are debugging a server or client and don't want to be disconnected when stepping through code. The default value is 1.

`gm_dumpstats`

Write GridMate network profiling stats to file.

`gm_dumpstats_file`

The file to which GridMate profiling stats are written. The default is `net_profile.log`.

`gm_net_simulator`

Activate GridMate network simulator to simulate latency, packet loss, bandwidth restrictions, and other conditions. For available options, type `gm_net_simulator help` .

gm_setdebugdraw

Display an overlay with detailed GridMate networking statistics and information. A user-friendly helper command for **gm_debugdraw debug_draw_level**. Possible parameters are `Basic`, `Trace`, `Stats`, `Replicas`, and `Actors`.

gm_stats_interval_msec

Set the interval, in milliseconds, for gathering network profiling statistics. The default is 1000.

gm_tracelevel trace_level

Set the GridMate debugging trace verbosity level. The default is 0. The higher the value, the greater the verbosity. Typical values range from 1 to 3.

mpstart [*<local_port>*]

Starts a LAN session by initializing the network system and optionally setting the local UDP port that initializes the socket. The default port is 64090. To use the ephemeral port, set the port to 0. This is useful if you want to connect to a server on the same computer as the client.

mphost

Create a session as host. The server listens for incoming connections on the port specified in `mpstart`.

mpjoin [*<server_addr>*] [*<server_port>*]

Connect to a server at the optionally specified *<server_addr>* and *<server_port>*. The defaults are `localhost` and `64090`, respectively.

map *<map_name>*

Loads the level with the specified map name. Replace *<map_name>* with the name of the map you want to use. To view a list of available levels, type **map**, and then press the tab key.

mpdisconnect

Terminate the current game instance session.

mpstop

Terminate the multiplayer service.

CryNetwork Backward Compatibility

CryNetwork has been deprecated and removed, and is no longer be supported in Lumberyard. There were several systems added to provide backwards compatibility for GridMate to the networked systems in CryEngine, namely remote method invocations, network serialization, and aspects. For more information, see the following sub topics.

Topics

- RMI Functions (p. 1014)
- Network Serialization and Aspects (p. 1017)

RMI Functions

To send remote method invocations (RMIs), use the `InvokeRMI` function, which has the following syntax:

```
void InvokeRMI( IRMIRep& <rep>, ParamsType&& <params>, uint32 <where>, ChannelId <channel>
 = kInvalidChannelId );
```

Parameters

<rep>

Represents the remote function to be called (the RMI ID).

<params>

Specifies the parameters to pass into the remote function.

<where>

Specifies a flag that determines the category of clients to which the RMI will be sent. For information, see the RMI Function Flags (p. 1016) section later in this document.

<channel>

Specifies specific clients to which the RMI will be sent, or specific clients to exclude. For information, see the RMI Function Flags (p. 1016) section later in this document.

Ordering RMI Functions

The `IGameObject.h` file includes macros for declaring RMI classes (for example, those beginning with `DECLARE_SERVER_RMI_<...>`). The different declaration types are as follows:

- `PREATTACH` – The RMI is attached at the top of the data update for the object. You can use this declaration type to prepare the remote entity for new incoming data.
- `POSTATTACH` – The RMI is attached at the bottom of the data update, so it is called after the data is serialized. You can use this declaration type to complete an action with the new data.
- `NOATTACH` – The RMI is not attached to a data update, so the RMI cannot rely on the data. You can use this declaration type for calls that don't rely on data.

Ordering Rules

The order for RMIs is only applicable within an object and attachment type set.

For example, in the following ordered list, PLAYER RMI 1, 2, and 3 will arrive in that order; however, ITEM RMI 1 might arrive before or after the following PLAYER RMIs:

- PLAYER RMI 1
- PLAYER RMI 2
- ITEM RMI 1
- PLAYER RMI 3

Using declaration types adds a layer of complication to the order of incoming data:

- `PREATTACH` – Messages are ordered within themselves.
- `POSTATTACH` – Messages are ordered within themselves.
- `NOATTACH` – Messages are ordered within themselves; however, `NOATTACH` can only fall on either side of the following diagram and never in between:

RMI Function Flags

To specify the clients that will receive an RMI, replace the *<where>* parameter in the `InvokeRMI` function with one of the following flags.

Server RMIs

`eRMI_ToClientChannel`

Sends an RMI from the server to a specific client. Specify the destination channel in the *<channel>* parameter.

`eRMI_ToOwningClient`

Sends an RMI from the server to the client that owns the actor.

`eRMI_ToOtherClients`

Sends an RMI from the server to all clients except the client specified. Specify the client to ignore in the *<channel>* parameter.

`eRMI_ToRemoteClients`

Sends an RMI from the server to all remote clients. Ignores the local client.

`eRMI_ToOtherRemoteClients`

Sends an RMI from the server to all remote clients except the remote client specified. Ignores the local client. The remote client to ignore is specified in the *<channel>* parameter.

`eRMI_ToAllClients`

Sends an RMI from the server to all clients.

Client RMIs

`eRMI_ToServer`

Sends an RMI from the client to the server.

Examples

To define a function to be implemented as RMI, use the `IMPLEMENT_RMI #define` from `IGameObject.h`.

```
#define IMPLEMENT_RMI(cls, name)
```

The following example implements a new function called `Cl_SetAmmoCount` in the `CInventory` class to be used as a client-side RMI, taking one argument of type `TRMIInventory_Ammo`:

```
Class CInventory : public CGameObjectExtensionHelper<CIventory, IInventory>
{
  // …
  DECLARE_CLIENT_RMI_NOATTACH(Cl_SetAmmoCount, TRMIInventory_Ammo, eNRT_ReliableOrdered);
  // …
};

IMPLEMENT_RMI(CInventory, Cl_SetAmmoCount)
{
  // Game code:
  TRMIInventory_Ammo Info(params);
```

```
  IEntityClass* pClass = gEnv->pEntitySystem->GetClassRegistry()-
>FindClass(Info.m_AmmoClass.c_str());
  If (pClass)
    SetAmmoCount(pClass, Info.m_iAmount);

  return true;    // Always return true - false will drop connection
}
The following line will invoke the function:
pInventory->GetGameObject()->InvokeRMI(CInventory::Cl_SetAmmoCount(),
 TRMIInventory_Ammo(“Pistol”, 10), eRMI_ToAllClients);
```

The following line will invoke the function:

```
pInventory->GetGameObject()->InvokeRMI(CInventory::C1_SetAmmoCount(),
 TRMIInventory_Ammo("Pistol", 10), eRMI_ToAllClients);
```

Network Serialization and Aspects

All objects that are intended to be synchronized over the network should have a function called `NetSerialize()`. In the `GameObject`, this appears as: `IGameObject::NetSerialize()`.

The `NetSerialize()` function uses a `TSerialize` object of type `ISerialize` to transform relevant data to a stream. The serialization uses different aspects and profiles to distinguish the various types of streams.

> **Note**
> Serialized data for a given aspect and profile must remain fixed. For example, if you serialized four floats, you must always serialize four floats.

Aspects

You use aspects to logically group data together.

Aspects are defined as follows:

- `eEA_GameClient` – Information sent from the client to the server, if the client has authority over the object.
- `eEA_GameServer` – The normal server to client data stream.
- `Dynamic/Static` – Data that is constantly changing should be added to the `Dynamic` aspect. Objects that rarely change should be added to the `Static` aspect. Updates are not sent if only one value changes.
- `eEA_Script` – Used where script network data is transported, including any script RMI calls.
- `eEA_Physics` – Used where physics data is transported. It is not divided into client/server because it always uses the same path: (controlling-client) to serve other clients.

Profiles

Profiles allow an aspect's fixed format data to be different. There are potentially eight profiles per aspect, and they are only used for physics aspects (for example, switching between ragdoll and living entity).

Physics

This section describes the Physics system and how to interact with the physics engine.

To create a physical world object, you use the `CreatePhysicalWorld()` function. The `CreatePhysicalWorld()` function returns a pointer to the `IPhysicalWorld` interface. You then fill the world with geometries and physical entities. You control the entities with the functions described in this section. Some functions apply to almost all entities, while others apply to specific entity structures. Other functions control how entities interact and how the world affects them.

The following sections describe these topics in detail:

Topics

- Physics Scripting Guide (p. 1018)
- Geometries (p. 1021)
- Physical Entities (p. 1022)
- Functions for Entity Structures (p. 1025)
- Collision Classes (p. 1032)
- Functions for World Entities (p. 1035)

Physics Scripting Guide

When scripting physics, deciding on an appropriate threading strategy can be important. This topic offers physics scripting advice on threading and troubleshooting and provides some sample code.

Threading

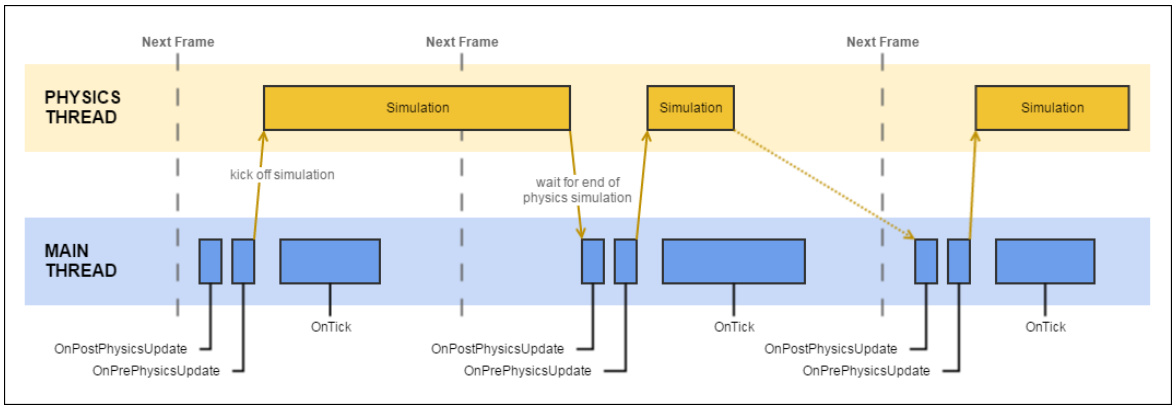
When writing scripts that interact with physics, it is important to understand how the physics system processes requests. You can configure the physics system to run in several different ways, each with its own benefits and drawbacks. This guide covers the two most common configurations.

Physics on a Separate Thread

Running the physics simulation on its own thread takes advantage of a machine's multiple cores and should lead to faster frame rates on physics-intensive games. However, this approach can make gameplay scripting more complex.

Note
This is the default configuration starting in Lumberyard 1.8.

The following diagram shows the relationships between the physics thread and main thread.



If a change request arrives when the simulation is busy, the request is queued and is processed the next time the simulation runs. If the simulation is idle, the change request is processed immediately.

Similarly, if a query is made when the simulation is busy, the response contains the state from the end of the last simulation. If the simulation is idle when the query is made, the response represents the true latest state.

In the diagram, note how `OnPrePhysicsUpdate` and `OnPostPhysicsUpdate` run between simulations. This means that requests are processed immediately and queries represent the true latest state. You should avoid performing time-consuming tasks in the `OnPrePhysicsUpdate` and `OnPostPhysicsUpdate` functions. Such tasks delay the start of simulation on the physics thread.

During the `OnTick` event, the state of the simulation can vary. Requests are either queued or processed immediately depending on whether the simulation has been completed for the frame.

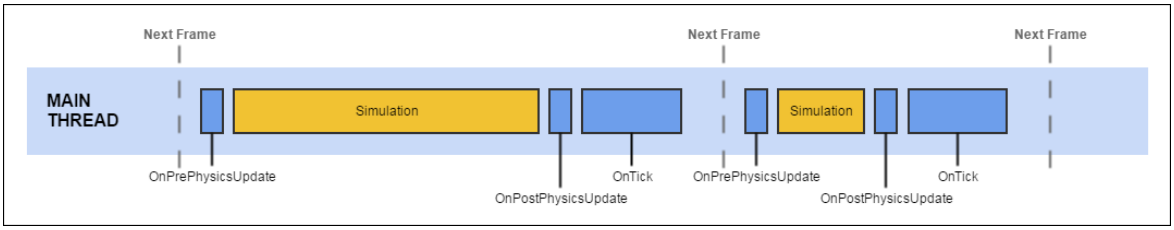
The following table provides some tips on using the `OnTick`, `OnPrePhysicsUpdate`, and `OnPostPhysicsUpdate` functions.

Function	Good For	Bad For	EBus
`OnTick`	Making fire-and-forget requests	Knowing when a request is done processing	`TickBus`
`OnPrePhysicsUpdate` `OnPostPhysicsUpdate`	Making a request and reliably querying its impact	Time-consuming tasks	`PhysicsSystemEventBus`

Physics on the Main Thread

Running the physics simulation in the main thread makes gameplay scripting simpler and more reliable. All requests are processed immediately and all queries represent the latest state. However, a physics-intensive game is more likely to impact the frame rate.

The following diagram illustrates a physics simulation on the main thread.



To enable physics in the main thread, type the following in the `game.cfg` file:

```
sys_physics_CPU=0
```

Physics Scripting Tips and Tricks

Following are a few troubleshooting tips for physics scripting.

To turn on debug rendering of physics proxies

Do one of the following:

- In the console, type the following:

```
p_draw_helpers=1
```

- In Lumberyard Editor,in the **Rollup Bar**, click the **Display Settings** tab. Under **Render Settings** in the **Profile Options** section, select **Show Proxy**.

Correcting Objects That Float Away

If an entity floats away like a balloon, check the following common causes and solutions:

- The entity has gone beyond the borders of the terrain. The solution is to stay on the terrain.
- The entity has a low mass, and CryPhysics has determined that it is less dense than air. The solution is to adjust the mass or density.

Scripting with Physics Example

The following code example uses `BehaviorContext` syntax to script the physics for a rocket.

```
local rocket =
{
    Properties =
    {
        InitialVelocity = {default=Vector3(), description="The initial speed and direction
 of the entity"},
        Impulse = {default=Vector3(), description="The impulse to add every frame"},
    },
}

function rocket:OnActivate()
    PhysicsComponentRequestBus.Event.SetVelocity(self.entityId,
 self.Properties.InitialVelocity)
    self.TickHandler = TickBus.CreateHandler(self)
end
```

```
function rocket:OnTick(dt, scriptTime)
    PhysicsComponentRequestBus.Event.AddImpulse(self.Properties.Impulse)
end

function rocket:OnDeactivate()
    self.TickHandler:Disconnect()
end

return rocket
```

Geometries

Geometries are first created as independent objects so that they can be used alone via the `IGeometry` interface which they expose and then they can be physicalized and added to physical entities. Geometry physicalization means computing physical properties (volume, inertia tensor) and storing them in a special internal structure. Pointers to these structures can then be passed to physical entities.

Each physicalized geometry has a reference counter which is set to 1 during creation, incremented every time the geometry is added to an entity and decremented every time the geometry is removed or the entity is deleted. When the counter reaches 0, the physical geometry structure is deleted and the corresponding `IGeometry` object is released. The `IGeometry` object is also reference counted.

Geometry Management Functions

Geometry management functions are accessible through the geometry manager, which is a part of physical world. To obtain a pointer to the geometry manger, call the `GetGeomManager()` function.

CreateMesh

The `CreateMesh` geometry management function creates a triangular mesh object from a set of vertices and indices (3 indices per triangle) and returns the corresponding IGeometry pointer.

- The engine uses triangle connectivity information in many places, so it is strongly recommended to have meshes closed and manifold. The function is able to recognize different vertices that represent the same point in space for connectivity calculations (there is no tolerance though, it checks only for exact duplicates). Open edges are ok only for geometries that will not be used as parts of dynamic physical entities and only if there will be little or no interaction with them.
- For collision detection the function can create either an OBB or a memory-optimized AABB or a single box tree. Selection is made by specifying the corresponding flag. If both AABB and OBB flags are specified, the function selects the tree that fits the geometry most tightly. Since an OBB tree is tighter in most cases, priority of AABBs can be boosted to save memory (also, AABB checks are slightly faster if the trees are equally tight). The engine can either copy the vertex/index data or use it directly from the pointers provided.
- The `mesh_multycontact flags` give some hints on whether multiple contacts are possible. Specifying that multiple contacts are unlikely (`mesh_multycontact0`) can improve performance a bit at the expense of missing multiple contacts if they do occur (note that it does not necessarily mean they will be missed, it is a hint for the algorithm to use some optimizations more aggressively). `mesh_multycontact2` disables this optimization and ..1 is a recommended default setting. Convex geometries are detected and some additional optimizations are used for them, although internally there is no separate class for convex objects (this may change in the future).
- Meshes can have per-face materials. Materials are used to look up friction, bounciness, and pierceability coefficients and can be queried by the game as a part of collision detection output.
- The `CreateMesh` function is able to detect meshes that represent primitives (with the specified tolerance) and returns primitive objects instead. In order to activate this detection, the corresponding

flags should be specified. Note that primitives can't store materials. They can only have one in the physical geometry structure, so this detection is not used when the material array has more than one material index in it.

CreatePrimitive

`CreatePrimitive`: Creates a primitive geometry explicitly. The corresponding primitive (cylinder, sphere, box, heightfield, or ray) structure should be filled and passed as a parameter, along with its ::type.

RegisterGeometry

`RegisterGeometry` physicalizes an `IGeometry` object by computing its physical properties and storing them in an auxiliary structure. Material index (`surfaceidx`) can be stored in it; it will be used if the geometry itself does not have any materials specified (such as if it is a primitive). `AddRefGeometry` and `UnregisterGeometry` comprise a reference "sandwich" for it. Note that the latter does not delete the object until its reference count becomes 0.

Geometries and physicalized geometries can be serialized. This saves time when computing OBB trees. That computation is not particularly slow, but serialization is faster.

Physical Entities

Physical entities can be created via calls to the `CreatePhysicalEntity` method of the physical world. `CreatePhysicalEntity` can create the types of entities noted in the following table:

Physical Entity Types

Type	Description
`PE_ARTICULATED`	An articulated structure, consisting of several rigid bodies connected with joints (a ragdoll, for instance). It is also possible to manually connect several `PE_RIGID` entities with joints, but in this case they will not know that they comprise a single object, and thus some useful optimizations cannot be used.
`PE_LIVING`	A special entity type to represent player characters that can move through the physical world and interact with it.
`PE_PARTICLE`	A simple entity that represents a small lightweight rigid body. It is simulated as a point with some thickness and supports flying, sliding and rolling modes. Recommended usage: rockets, grenades and small debris.
`PE_RIGID`	A single rigid body. Can have infinite mass (specified by setting mass to 0), in which case it will not be simulated but will interact properly with other simulated objects;
`PE_ROPE`	A rope object. It can either hang freely or connect two purely physical entities.
`PE_SOFT`	A system of non-rigidly connected vertices that can interact with the environment. A typical usage is cloth objects.
`PE_STATIC`	An immovable entity. An immovable entity can still be moved manually by setting positions from outside, but in order to ensure proper interactions with simulated objects, it is better to use `PE_RIGID` entity with infinite mass.
`PE_WHEELEDVEHICLE`	A wheeled vehicle. Internally it is built on top of a rigid body, with added vehicle functionality (wheels, suspensions, engine, brakes).

Note
`PE_RIGID`, `PE_ARTICULATED` and `PE_WHEELEDVEHICLE` are purely physical entities that comprise the core of the simulation engine. The other entities are processed independently.

Creating and managing entities

When creating and managing entities, keep in mind the following:

- Entities use a two-dimensional, regular grid to speed up broad phase collision detection. The grid should call the `SetupEntityGrid` function before physical entities are created.
- Entities can be created in permanent or on-demand mode and are specified by the parameter `lifeTime` (use 0 for permanent entities). For on-demand mode, the entity placeholders should be created first using `CreatePhysicalPlaceholder`. Physics will then call the outer system to create the full entity whenever an interaction is required in the bounding box for this placeholder.
- If an entity is not involved in any interactions for the specified lifetime, it will be destroyed, with the placeholder remaining. Placeholders require less memory than full entities (around 70 bytes versus 260 bytes). It is possible for an outer system to support hierarchical placeholders, such as meta-placeholders that create other placeholders upon request.
- A sector-based, on-demand physicalization is activated after `RegisterBBoxInPODGrid` is called. Entities are created and destroyed on a sector basis. The sector size is specified in `SetupEntityGrid`.
- You can use `SetHeightfieldData` to set up one special static terrain object in the physical world. You can also create unlimited terrain geometry manually and add it to an entity.

Destroying, suspending, and restoring entities

To destroy, suspend, or restore a physical entity, use `DestroyPhysicalEntity` and set the `mode` parameter to 0, 1, or 2, respectively. Suspending an entity clears all of its connections to other entities, including constraints, without actually deleting the entity. Restoring an entity after suspension will not restore all lost connections automatically. Deleted entities are not destroyed immediately; instead, they are put into a recycle bin. You might need to remove references to any one-way connections. The recycle bin is emptied at the end of each `TimeStep`. You can also call `PurgeDeletedEntities`.

Physical entity IDs

All physical entities have unique IDs that the physics engine generates automatically. You do not need to specify an ID during creation. You can also set a new ID later. Entities use these IDs during serialization to save dependency information. When reading the saved state, be sure that entities have the same IDs. IDs are mapped to entity pointers by use of an array, so using large ID numbers will result in allocation of an equally large array.

Associations with outside objects

To maintain associations with outside engine objects, physical entities store an additional void pointer and two 16-bit integers (`pForeignData`, `iForeignData`, and `iForeignFlags`) . These parameters are set from outside, not by the entities. Use `pForeignData` to store a pointer to the outside engine reference entity and `iForeignData` to store the entity type, if applicable.

For each material index, the physical world stores the friction coefficient, a bounciness (restitution) coefficient, and flags. When two surfaces contact, the contact's friction and bounciness are computed as an average of the values of both surfaces. The flags only affect raytracing.

Simulation type

Physical entities are grouped by their simulation type, in order of increasing "awareness". Certain interface functions, such as ray tracing and querying entities in an area, allow filtering of these entities by type.

- 0 (`bitmask ent_static`) – Static entities. Although terrain is considered static, it does not have a special simulation type. It can be filtered independently with the `ent_terrain` bitmask.
- 1 (`bitmask ent_sleeping_rigid`) – Deactivated, physical objects (rigid bodies and articulated bodies).
- 2 (`bitmask ent_rigid`) – Active, physical objects.
- 3 (`bitmask ent_living`) – Living entities.
- 4 (`bitmask ent_independent`) – Physical entities that are simulated independently from other entities (particles, ropes, and soft objects).
- 6 (`bitmask ent_triggers`) – Entities (or placeholders) that are not simulated and only issue callbacks when other entities enter their bounding box.
- 7 (`bitmask ent_deleted`) – Objects in the recycle bin. Do not use this directly.

Entities that have a lower simulation type are not aware of entities with higher simulation types (types 1 and 2 are considered as one for this purpose), so players (type 3) and particles (type 4) check collisions against physical entities (types 1 and 2) but physical entities do not know anything about them. Similarly, ropes (type 4) can check collisions against players but not the other way. However, entities of higher types can still affect entities with lower types by using impulses and constraints. Most entities expect a particular simulation type (and will automatically set to the proper value).

There are exceptions to the 'awareness hierarchy': for example, articulated entities can be simulated in types 1 and 2 as fully physicalized dead bodies, or in type 4 as skeletons that play impact animations without affecting the environment and being affected by it.

Functions for Physical Entities

Most interactions with physical entities will use the functions `AddGeometry`, `SetParams`, `GetParams`, `GetStatus`, and `Action`.

- `AddGeometry` – Adds multiple geometries (physicalized geometries) to entities. For more details, see the AddGeometry section that follows.
- `RemoveGeometry` – Removes geometries from entities.
- `SetParams` – Sets parameters.
- `GetParams` – Gets the simulation input parameters.
- `GetStatus` – Gets the simulation output parameters. `GetStatus` requests the values that an entity changes during simulation.
- `Action` – Makes an entity execute an action, such as adding an impulse.

These functions take structure pointers as parameters. When you want to issue a command, you can create a corresponding structure (for example, as a local variable) and specify only the fields you need. The constructor of each structure provides a special value for all fields that tells the physics engine that the field is unused. You can also do this explicitly by using the `MARK_UNUSED` macro and `is_unused` to verify that the field is unused.

AddGeometry

`AddGeometry` adds a physicalized geometry to an entity. Each geometry has the following properties:

- `id` – A unique part identifier within the bounds of the entity to which the geometry belongs. You can specify the ID or use `AddGeometry` to generate a value automatically. The ID doesn't change if the parts array changes (for example, if some parts from the middle are removed), but the internal parts index might change.
- `position`, `orientation`, and `uniform scaling` – Relative to the entity.
- `mass` – Used for non-static objects; static objects assume infinite mass in all interactions. You can specify the mass or density where the complementary value will be computed automatically (using

formula mass = density*volume; `volume` is stored in the physicalized geometry structure and scaled if the geometry is scaled).

- `surface_idx` – Used if neither IGeometry nor physicalized geometry have surface (material) identifiers.
- `flags` and `flagsCollider` – When an entity checks collisions against other objects, it checks only parts that have a flag mask that intersects its current part's `flagsCollider`. You can use 16-type bits (`geom_colltype`) to represent certain entity groups. Although not enforced, it is good practice to keep these relationships symmetrical. If collision checks are known to be one-sided (for example, entity A can check collisions against entity B but never in reverse), you can choose to not maintain this rule. Certain flags are reserved for special collision groups, such as `geom_colltype1 = geom_colltype_players` and `geom_colltype2 = geom_colltype_explosion` (when explosion pressure is calculated, only parts with this flag are considered). There are also special flags for raytracing and buoyancy calculations: `geom_colltype_ray` and `geom_floats`.
- `minContactDist` – The minimum distance between contacts the current part of the entity might have with another part of an entity. Contacts belonging to different parts are not checked for this. You can leave this unused so it will initialize with a default value based on geometry size. Each part can have both geometry and proxy geometry. Geometry is used exclusively for raytracing and proxy geometry. If no proxy geometry is specified, both geometries are set to be equal to allow the raytracing to test against high-poly meshes without needing to introduce changes to the part array layout.

Functions for Entity Structures

This section describes functions that control general and specific kinds of entity structures.

Topics

- Common Functions (p. 1025)
- Living Entity-Specific Functions (p. 1027)
- Particle Entity-Specific Functions (p. 1029)
- Articulated Entity-Specific Functions (p. 1029)
- Rope Entity-Specific Functions (p. 1031)
- Soft Entity-Specific Functions (p. 1031)

Common Functions

pe_params_pos

Sets the position and orientation of the entity. You can use offset/quaternion/scaling values directly or allow the physics to extract them from a 3x3 (orientation+scaling) or a 4x4 (orientation_scaling +offset) matrix. Physics use a right-to-left transformation order convention, with vectors being columns (`vector_in_world = Matrix_Entity * Matrix_Entity_Parts * vector_in_geometry`). All interface structures that support matrices can use either row-major or column-major matrix layout in memory (the latter is considered transposed; thus, the corresponding member has T at the end of its name).

There is no per-entity scaling; scaling is only present for parts. When a new scaling is set with `pe_params_pos`, it is copied into each part and overrides any previous individual scalings. This structure also allows you to set the simulation type manually. After changes are made, entity bounds are typically recalculated and the entity is re-registered in the collision hash grid; however, this can be postponed if `bRecalcBounds` is set to `0`.

pe_params_bbox

Sets an entity's bounding box to a particular value, or queries it when used with `GetParams`). The bounding box is recalculated automatically based on the entity's geometries, but you can set the

bounding box manually for entities without geometries (for example, triggers) or placeholders. If the entity has geometries, it might recalculate its bounding box later, overriding these values. Bounding boxes are axis-aligned and in the world coordinate system.

pe_params_outer_entity

Specifies an outer entity for an entity. When a box of interest (its center) is inside the entity with an outer entity, the outer entity is excluded from the set of potential colliders. This allows you to have a building exterior quickly culled away when the region of interest is inside the building's interior. Outer entities can be nested and an optional geometry to test for containment is supported.

pe_params_part

Sets or queries the entity part's properties. The part can be specified using an internal part index or its ID.

pe_simulation_params

Sets simulation parameters for entities that can accept these parameters (e.g. physical entities, ropes, and soft entities). `minEnergy` is equal to sleep speed squared. Damping and gravity can be specified independently for colliding and falling state, for example when there are no contacts.

pe_params_buoyancy

Sets the buoyancy properties of the object and the water plane. The physics engine does not have a list of water volumes, so the outer system must update water plane parameters when they change. The water surface is assumed to be a perfect plane, so you can simulate bobbing of the waves by disturbing the normal of this surface. `waterFlow` specifies the water movement velocity and affects the object based on its `waterResistance` property). A separate sleeping condition is used in the water (`waterEmin`).

pe_params_sensors

Attaches sensors to entities. Sensors are rays that the entity can shoot to sample the environment around it. It is more efficient to do it from inside the entity step than by calling the world's raytracing function for every ray from outside the entity step. Living entities support vertical-down sensors.

pe_action_impulse

Adds a one-time impulse to an entity. `impulse` is the impulse property (in N*sec; impulse P will change the object's velocity by P/[object mass]). `point` is a point in world space where the impulse is applied and used to calculate the rotational effects of the impulse. The `of point` momentum can be used to specify the rotational impulse explicitly. If neither the point nor momentum are specified, the impulse is applied to the center of the mass of the object. `iApplyTime` specifies the time when the impulse is applied. By default the value is 2 ("after the next step") to allow the solver an opportunity to reflect the impulse.

pe_action_add_constraint

Adds a constraint between two objects. Points specify the constraint positions in world space. If the second point is used and different from the first point, the solver will attempt to join them.

Relative positions are always fully constrained to be 0 (i.e. the points on the bodies will always be in one spot) and relative rotations can be constrained in twist and bend directions. These directions correspond to rotation around the x-axis and the remaining rotation around a line on the yz-plane (tilt of the x axis) of a relative transformation between the two constraint coordinate frames attached to the affected bodies.

The original position of the constraint frames are specified with `qframe` parameters in world or entity coordinate space (as indicated by the corresponding flag in `flags`). If one or both qframes are unused, they are considered to be an identity transformation in either the world or entity frame.

Rotation limits are specified with the `xlimits` and `yzlimits` parameters, with valid element values of 0 (minimum) and 1 (maximum). If the minimum is more than or equal to the maximum, the corresponding relative rotation is prohibited. `pConstraintEntity` specifies an entity that represents the constraint. When passed a pe_action_add_constraint pointer, `Action` returns a constraint identifier that can be used to remove the constraint. 0 indicates a failure.

pe_action_set_velocity

Sets the velocity of an object, which is useful for rigid bodies with infinite mass (represented as mass). `pe_action_set_velocity` informs the physics system about the body's velocity, which can help the solver ensure zero relative velocity with the objects contacted. If velocity is not set and only the position is changed, the engine relies solely on penetrations to enforce the contacts. Velocity will not be computed automatically if the position is set manually each frame. The body will continue moving with the specified velocity once it has been set.

pe_status_pos

Requests the current transformation (position, orientation, and scale) of an entity or its part. You can also use `pe_params_pos` with `GetParams`. If matrix pointers are set, the engine will provide data in the corresponding format. The `BBox` member in this structure is relative to the entity's position.

pe_status_dymamics

Retrieves an entity's movement parameters. Acceleration and angular acceleration are computed based on gravity and interactions with other objects. External impulses that might have been added to the entity are considered instantaneous. `submergedFraction` is a fraction of the entity's volume under water during the last frame (only parts with the `geom_float` flag are considered). `waterResistance` contains the maximum water resistance that the entity encountered in one frame since the status was last requested (the accumulated value is cleared when the status is returned). This value can be useful for generating splash effects.

Living Entity-Specific Functions

Living entities use cylinders or capsules as their bounding geometry. Normally the cylinders are hovering above the ground and the entity shoots a single ray down to detect if it is standing on something. This cylinder geometry always occupies the first part slot (it is created automatically). It is possible to add more geometries manually, but they will not be tested against the environment when the entity moves. However, other entities will process them when testing collisions against the entity.

Living entities never change their orientation themselves; this is always set from outside. Normally, living entities are expected to rotate only around the z-axis, but other orientations are supported. However, collisions against living entities always assume vertically oriented cylinders.

pe_player_dimensions (GetParams | SetParams)

Sets the dimensions of the living entity's bounding geometry.

`heightPivot` specifies the z-coordinate of a point in the entity frame that is considered to be at the feet level (usually 0).

`heightEye` is the z-coordinate of the camera attached to the entity. This camera does not affect entity movement, its sole purpose is to smooth out height changes that the entity undergoes (during walking

on a highly bumpy surface, such as stairs, after dimensions change and during landing after a period of flying). The camera position can be requested via the `pe_status_living` structure.

`sizeCollider` specifies the size of the cylinder (x is radius, z is half-height, y is unused).

`heightColliders` is the cylinder's center z-coordinate.

The head is an auxiliary sphere that is checked for collisions with objects above the cylinder. Head collisions don't affect movement but they make the camera position go down. `headRadius` is the radius of this sphere and `headHeight` is the z-coordinate of its center in the topmost state (that is, when it doesn't touch anything).

pe_player_dynamics (GetParams | SetParams)

Sets a living entity's movement parameters. Living entities have their 'desired' (also called 'requested') movement velocity (set with `pe_action_move`) and they attempt to reach it. How fast that happens depends on the `kInertia` setting. The greater this value is, the faster the velocity specified by `pe_action_move` is reached. The default is 8. 0 means that the desired velocity will be reached instantly.

`kAirControl` (0..1) specifies how strongly the requested velocity affects movement when the entity is flying (1 means that whenever a new requested velocity is set, it is copied to the actual movement velocity).

`kAirResistance` describes how fast velocity is damped during flying.

`nodSpeed` (default 60) sets the strength of camera reaction to landings.

`bSwimming` is a flag that tells that the entity is allowed to attempt to move in all directions (gravity might still pull it down though). If not set, the requested velocity will always be projected on the ground if the entity is not flying.

`minSlideAngle`, `maxClimbAngle`, `maxJumpAngle` and `minFallAngle` are threshold angles for living entities that specify maximum or minimum ground slopes for certain activities. Note that if an entity's bounding cylinder collides with a sloped ground, the behavior is not governed by these slopes only.

Setting `bNetwork` makes the entity allocate a much longer movement history array which might be required for synchronization (if not set, this array will be allocated the first time network-related actions are requested, such as performing a step back).

Setting `bActive` to 0 puts the living entity to a special 'inactive' state where it does not check collisions with the environment and only moves with the requested velocity (other entities can still collide with it, though; note that this applies only to the entities of the same or higher simulation classes).

pe_action_move

Requests a movement from a living entity. `dir` is the requested velocity the entity will try to reach. If `iJump` is not 0, this velocity will not be projected on the ground, and snapping to the ground will be turned off for a short period of time. If `iJump` is 1, the movement velocity is set to be equal to `dir` instantly. If `iJump` is 2, `dir` is added to it. `dt` is reserved for internal use.

pe_status_living

Returns the status of a living entity.

`vel` is the velocity that is averaged from the entity's position change over several frames.

`velUnconstrained` is the current movement velocity. It can be different from `vel` because in many cases when the entity bumps into an obstacle, it will restrict the actual movement but keep the movement velocity the same, so that if on the next frame the obstacle ends, no speed will be lost.

`groundHeight` and `groundSlope` contain the point's z coordinate and normal if the entity is standing on something; otherwise, `bFlying` is 1. Note that `pGroundCollider` is set only if the entity is standing on a non-static object.

`camOffset` contains the current camera offset as a 3d vector in the entity frame (although only z coordinates actually changes in it).

`bOnStairs` is a heuristic flag that indicates that the entity assumes that it is currently walking on stairs because of often and abrupt height changes.

Particle Entity-Specific Functions

pe_params_particle

Sets particle entity parameters.

During movement, particles trace rays along their paths with the length `size*0.5` (since `size` stands for 'diameter' rather than 'radius') to check if they hit something. When they lie or slide, they position themselves at a distance of `thickness*0.5` from the surface (thus thin objects like shards of glass can be simulated).

Particles can be set to have additional acceleration due to thrust of a lifting force (assuming that they have wings) with the parameters `accThrust` and `accLift` but these should never be used without specifying `kAirResistance`; otherwise, particles gain infinite velocity.

Particles can optionally spin when in the air (toggled with flag `particle_no_spin`). Spinning is independent from linear motion of particles and is changed only after impacts or falling from surfaces.

Particles can align themselves with the direction of the movement (toggled with `particle_no_path_alignment` flag) which is very useful for objects like rockets. That way, the y-axis of the entity is aligned with the heading and the z-axis is set to be orthogonal to y and to point upward ('up' direction is considered to be opposite to particle's gravity).

When moving along a surface, particles can either slide or roll. Rolling can be disabled with the flag `particle_no_roll` (it is automatically disabled on steep slopes). Note that rolling uses the particle material's friction as damping while rolling treats friction in a conventional way. When touching ground, particles align themselves so that their normal (defined in entity frame) is parallel to the surface normal.

Particles can always keep the initial orientation as well (`particle_constant_orientation`) and stop completely after the first contact (`particle_single_contact`). `minBounceVel` specifies the lower velocity threshold after which the particle will not bounce, even if the bounciness of the contact is more than 0.

Articulated Entity-Specific Functions

Articulated entities consist of linked, rigid bodies called structural units. Each structural unit has a joint that connects it to its parent. For the connection structure, you should use a tree with a single root. Linked loops are not allowed.

Articulated entities can simulate body effects without interactions with the environment by using featherstone mode, which you can tweak so that the entity tolerates strong impacts and so that complex body structures have stiff springs. Articulated entities use a common solver for interactive mode.

pe_params_joint

You can use `pe_params_joint` to:

- Create a joint between two bodies in an articulated entity
- Change the parameters of an existing articulated entity

- Query the parameters of an existing articulated entity, when used with `GetParams`

A joint is created between the two bodies specified in the `op` parameter at the pivot point (in the entity frame). When a geometry is added to an articulated entity, it uses `pe_articgeomparams` to specify which body the geometry belongs to (in `idbody`). `idbody` can be any unique number and each body can have several geometries. There are no restrictions on the order in which joints are created, but all bodies in an entity must be connected before the simulation starts.

Joints use Euler angles to define rotational limits. Flags that start with `angle0_` can be specified individually for each angle by shifting left by the 0-based angle index. For example, to lock the z-axis you can use OR the flags with `angle0_locked<<2`). The child body inherits the coordinate frame from the first entity (geometry) that was assigned to it.

Joint angular limits are defined in a relative frame between the bodies that the joint connects. Optionally the frame of the child body can be offset by specifying a child's orientation that corresponds to rotation angles (0,0,0), using q0, pMtx0, or pMtx0T. This can help to get limits that can be robustly represented using Euler angles.

A general rule for limits is to set upper and lower bounds at least 15 to 20 degrees apart (depending on simulation settings and the height of the joint's velocity) and to keep the y-axis limit in the -90..90 degrees range (preferably within safe margins from its ends).

Note
All angles are defined in radians in the parameter structure.

`pe_params_joint` uses 3D vectors to represent groups of three values that define properties for each angle. In addition to limits, each angle can have a spring that will pull the angle to 0 and a dashpot that will dampen the movement as the angle approaches its limit. Springs are specified in acceleration terms: stiffness and damping can stay the same for joints that connect bodies with different masses, and damping can be computed automatically to yield a critically damped spring by specifying `auto_kd` for the corresponding angle.

`joint_no_gravity` makes the joint unaffected by gravity, which is useful if you assume forces that hold the joint in its default position are enough to counter gravity). This flag is supported in featherstone mode.

`joint_isolated_accelerations` makes the joint use a special mode that treats springs like guidelines for acceleration, which is recommended for simulating effects on a skeleton. This flag is supported in featherstone mode.

Effective joint angles are always the sum of `q` and `qext`. If springs are activated, they attempt to drive `q` to 0. The allows you to set a pose from animation and then apply physical effects relative to it. In articulated entities, collisions are only checked for pairs that are explicitly specified in `pSelfCollidingParts` (this setting is per body or per joint, rather than per part).

pe_params_articulated_body

`pe_params_articulated_body` allows you to set and query articulated entity simulation mode parameters. Articulated entities can be attached to something or be free, and are set by the `bGrounded` flag. When grounded, the entity can:

- Fetch dynamic parameters from the entity it is attached to (if `bInreritVel` is set; the entity is specified in `pHost`)
- Be set using the `a`, `wa`, `w` and `v` parameters

`bCollisionResp` switches between featherstone mode (0) and constraint mode (1).

`bCheckCollisions` turns collision detection on and off. It is supported in constraint mode.

`iSimType` specifies a simulation type, which defines the way in which bodies that comprise the entity evolve. Valid values:

- `0` – joint pivots are enforced by projecting the movement of child bodies to a set of constrained directions |
- `1` – bodies evolve independently and rely on the solver to enforce the joints. The second mode is not supported in featherstone mode. In constraint mode, it is turned on automatically if bodies are moving fast enough.

  We recommend setting this value to `1` to make slow motion smoother.

Lying mode

Articulated entities support a lying mode that is enabled when the number of contacts is greater than a specified threshold (`nCollLyingMode`). Lying mode has a separate set of simulation parameters, such as gravity and damping. This feature was designed for ragdolls to help simulate the high damping of a human body in a simple way, for example by setting gravity to a lower value and damping to a higher than usual value.

Standard simulation versus freefall parameters

Standard simulation parameters can be different from freefall parameters. When using the constraint mode, articulated entities can attempt to represent hinge joints (rotational joints with only axis enabled) as two point-to-point constraints by setting the `bExpandHinges` parameter (this value propagates to `joint_expand_hinge` flags for all joints, so you do not need to manually set the value for joints).

Rope Entity-Specific Functions

Ropes are simulated as chains of connected equal-length sticks ("segments") with point masses. Each segment can individually collide with the environment. Ropes can tie two entities together. In this case ropes add a constraint to the entities when the ropes are fully strained and won't affect their movement.

In order to collide with other objects (pushing them if necessary) in a strained state, the rope must use dynamic subdivision mode (set by `rope_subdivide_segs` flag).

pe_params_rope

Specifies all the parameters a rope needs to be functional.

Rope entities do not require any geometry. If you do not specify initial point positions, the rope is assumed to be hanging down from its entity position. If you do specify initial point positions, segments should have equal length but within some error margin. Ropes use an explicit friction value (not materials) to specify friction.

If `pe_params_rope` is passed to `GetParams`, `pPoints` will be a pointer to the first vertex in an internal rope vertex structure, and `iStride` will contain the size of it.

Soft Entity-Specific Functions

There are two types of soft entities: *mesh-based* and *tetrahedral lattice-based*. Mesh based entities use a soft, constraint-like solver and are typically cloth objects. Tetrahedral lattice-based entities use a spring solver and are typically jelly-like objects.

The longest edges of all triangles can optionally be discarded with the `sef_skip_longest_edges` flag.

Collisions are handled at the vertex level only (although vertices have a customizable thickness) and work best against primitive geometries rather than meshes.

pe_params_softbody

This is the main structure to set up a working soft entity (another one is `pe_simulation_params`).

Thickness

The thickness of the soft entity is the collision size of vertices (they are therefore treated as spheres). If an edge differs from the original length by more than `maxSafeStep`, positional length enforcement occurs.

Damping

Spring damping is defined with `kdRatio` as a ratio to a critically damped value (overall damping from `pe_simulation_params` is also supported).

Wind

Soft entities react to wind if `airResistance` is not 0 (if wind is 0, having non-zero `airResistance` would mean that the entity will look like it is additionally damped - air resistance will attempt to even surface velocity with air velocity).

Water

Soft entities react to water in the same way that they react to wind, but the parameters specified in `pe_params_buoyancy` are used instead. Note that the Archimedean force that acts on vertices submerged in the water will depend on the entity's density which should be defined explicitly in `pe_simulation_params` (dependence will be same as for rigid bodies - the force will be 0 if `waterDensity` is equal to `density`). `collTypes` enables collisions with entities of a particular simulation type using `ent_` masks.

pe_action_attach_points

Can be used to attach some of a soft entity's vertices to another physical entity.

`piVtx` specifies vertex indices.

`points` specify attachment positions in world space. If `points` values are not specified, current vertex positions are the attachment points.

Collision Classes

Use collision classes to filter collisions between two physical entities. A collision class comprises two 32-bit uints, a `type`, and an `ignore`.

You can use collision classes to implement scenarios such as "player only collisions," which are objects passable by AI actors but not passable by players. This feature allows you to configure filtering of the collision between physical entities independently of their collision types.

Setup

Physical entities can have one or more collision classes and can ignore one or more collision classes. To have a physical entity ignore a collision, use the `ignore_collision` attribute of the `<Physics>` element in the `<SurfaceType>` definition, as shown in the following example:

`SurfaceTypes.xml`

```
<SurfaceType name="mat_nodraw_ai_passable">
```

```
  <Physics friction="0" elasticity="0"  pierceability="15"
 ignore_collision="collision_class_ai"/>
</SurfaceType>
```

All physical entity types such as `LivingEntity` and `ParticleEntity` are supplied with default collision classes like `collision_class_living` and `collision_class_particle`. Living entity uses one additional game specific collision class: either `collision_class_ai` for AI actors, or `collision_class_player` for players.

`Player.lua`

```
Player = {
...
  physicsParams =
  {
    collisionClass=collision_class_player,
  },
...
}
```

`BasicAI.lua`

```
BasicAI = {
...
  physicsParams =
  {
    collisionClass=collision_class_ai,
  },
...
}
```

Code

```
struct SCollisionClass
{
    uint32 type;     // collision_class flags to identify the entity
    uint32 ignore;   // another entity will be ignored if *any* of these bits are set in
 its type
};
```

The `type` identifies which entity the collision classes belong to.

Some collision classes like the following are defined in `CryPhysics`:

- `collision_class_terrain`
- `collision_class_wheeled`
- `collision_class_living`
- `collision_class_articulated`
- `collision_class_soft`
- `collision_class_roped`
- `collision_class_particle`

Other collision classes are defined in `GamePhysicsSettings.h`, starting from the `collision_class_game` bit:

```
#define GAME_COLLISION_CLASSES(f) \
```

```
    f( gcc_player_capsule,     collision_class_game << 0) \
    f( gcc_player_body,        collision_class_game << 1) \
    f( gcc_pinger_capsule,     collision_class_game << 2) \
    f( gcc_pinger_body,        collision_class_game << 3) \
    f( gcc_vehicle,            collision_class_game << 4) \
    f( gcc_large_kickable,     collision_class_game << 5) \
    f( gcc_ragdoll,            collision_class_game << 6) \
    f( gcc_rigid,              collision_class_game << 7) \
    f( gcc_alien_drop_pod,     collision_class_game << 8) \
    f( gcc_vtol,               collision_class_game << 9) \
```

All these classes are automatically exposed to Lua. Brushes and most objects have the collision classes available in the properties through the editor.

Types

For types, you can set many or zero bits.

In the following example, of the classes `LIVING`, `PLAYER`, `TEAM1`, `TEAM2`, `AI`, `AI_1`, and `AI_2`, `player1` belongs to the `LIVING` entity class, the `PLAYER` class, and the `TEAM1` class:

```
SCollisionClass player1(0,0), player2(0,0), ai1(0,0), ai7(0,0), object1(0,0);

player1.type = LIVING|PLAYER|TEAM1;
player2.type = LIVING|PLAYER|TEAM2;
ai1.type = LIVING|AI|AI_1;
ai7.type = LIVING|AI|AI_2;
object1.type = 0;
```

Filtering the collision

Filtering occurs by checking the `type` of one entity against the `ignore` of another entity.

This is done both ways, and if bits overlap, then the collision is ignored. For example:

```
bool ignoreCollision = (A->type & B->ignore) || (A->ignore & B->type);
```

If you want `ai7` to ignore collisions with anything that has `AI_1` set, then add `AI_1` to the `ignore` flags like this:

```
ai7.ignore = AI_1
```

If you want `object1` to ignore all living physical entities, set its `ignore` flag like this:

```
object1.ignore=LIVING
```

Interface

- For code, see `physinterface.h` and `GamePhysicsSettings.h`.
- To access and set the collision classes on the physical entity, use `*pe_collision_class struct SCollisionClass pe_params_collision_class`.
- For helpers that set additional ignore maps, see `GamePhysicsSettings.h`.
- In Lua, see `SetupCollisionFiltering` and `ApplyCollisionFiltering`. Lua script-binding is done through `SetPhysicParams(PHYSICPARAM_COLLISION_CLASS)`.

Functions for World Entities

Use the functions in this section to modify entities or a physical world environment.

Advancing the Physical World Time State

The `TimeStep` functions make the entities advance their state by the specified time interval.

If `timeGranularity` in the physical variables is set, the time interval will be snapped to an integer value with the specified granularity (for example, if `timeGranularity` is 0.001, the time interval will be snapped to a millisecond).

Entities that perform the step can be filtered with `ent_` flags in the `flags` parameter.

The `flags` parameter can contain `ent_` masks for Simulation type (p. 1023).

The `flags` parameter can also contain the `ent_flagged_only` flag. This flag causes entities to be updated only if the entities have the `pef_update` flag set.

Specifying the `ent_deleted` flag will allow the world to delete entities that have timed out if physics on demand is used.

Most entities have the maximum time step capped. To have larger timesteps, entities have to perform several substeps. The number of substeps can be limited with the physics variable `nMaxSubsteps`.

Returning Entities with Overlapping Bounding Boxes

The function `GetEntitiesInBox` uses the internal entity hash grid to return the number of entities whose bounding boxes overlap a specified box volume. The function supports filtering by Simulation type (p. 1023) and optional sorting of the output list by entity mass in ascending order.

Syntax

```
virtual int GetEntitiesInBox(Vec3 ptmin,Vec3 ptmax, IPhysicalEntity **&pList, int objtypes,
 int szListPrealloc=0) = 0;
```

Example call

```
IPhysicalEntity** entityList = 0;
int entityCount = gEnv->pPhysicalWorld->GetEntitiesInBox(m_volume.min, m_volume.max,
 entityList,
   ent_static | ent_terrain | ent_sleeping_rigid | ent_rigid);
```

Parameters

Parameter	Description	
`ptmin`	Minimum point in the space that defines the desired box volume.	
`ptmax`	Maximum point in the space that defines the desired box volume.	
`pList`	Pointer to a list of objects that the function poplulates.	
`objtypes`	Types of objects that need to be considered in the query.	

Parameter	Description	
`szListPrealloc`	If specified, the maximum number of objects contained in the `pList` array.	

The possible object types are described in the `physinterface.h` header file in the `entity_query_flags` enumerators. A few are listed in the following table:

Entity type flag	Description	
`ent_static`	Static entities	
`ent_terrain`	Terrain	
`ent_sleeping_rigid`	Sleeping rigid bodies	
`ent_rigid`	Rigid bodies	

After the function completes, you can easily iterate through the entity list to perform desired operations, as in the following code outline:

```
for (int i = 0; i < entityCount; \++i)
{

   IPhysicalEntity\* entity = entityList[i];

   [...]

   if (entity->GetType() == PE_RIGID)
   {
      [...]
   }

   [...]

}
```

If `ent_alloctate_list` is specified, the function allocates memory for the list (the memory can later be freed by a call to `pWorld->GetPhysUtils()->DeletePointer`). Otherwise, an internal pointer will be returned.

> **Note**
> Because the physics system uses this pointer in almost all operations that require forming an entity list, no such calls should be made when the list is in use. If such calls are required and memory allocation is undesired, copy the list to a local pre-allocated array before iterating over it.

Casting Rays in an Environment

The `RayWorldIntersection` physical world function casts rays into the environment.

Depending on the material that the ray hits and the ray properties, a hit can be pierceable or solid.

A pierceable hit is a hit that has a material pierceability higher than the ray's pierceability. Material pierceability and ray pierceability occupy the lowest 4 bits of `material` flags and `RayWorldIntersection` flags.

Pierceable hits don't stop the ray and are accumulated as a list sorted by hit distance. The caller provides the function with an array for the hits. A solid hit (if any) always takes the slot with index 0 and pierceable hits slots from 1 to the end.

Optionally, the function can separate between 'important' and 'unimportant' pierceable hits (importance is indicated by `sf_important` in material flags) and can make important hits have a higher priority (regardless of hit distance) than unimportant ones when competing for space in the array.

By default, `RayWorldIntersection` checks only entity parts with the `geom_colltype_ray` flag. You can specify another flag or combination of flags by setting `flags |= geom_colltype_mask<<rwi_colltype_bit`. In this case, all flags should be set in part so that the specified flag can be tested.

`RayTraceEntity` is a more low-level function and checks ray hits for one entity only. `RayTraceEntity` returns only the closest hit.

Alternatively, `CollideEntityWithBeam` can perform a sweep-check within a sphere of the specified radius. In order to detect collisions reliably, the sphere specified should be outside of the object. The `org` parameter corresponds to the sphere center.

Creating Explosions

The function `SimulateExplosion` is used to simulate explosions in a physical world.

The only effect of explosions inside the physics system are impulses that are added to the nearby objects. A single impulse is calculated by integrating impulsive pressure at an area fragment multiplied by this area and scaled by its orientation towards the epicenter.

Impulsive pressure has a falloff proportional to `1/distance2`. If `distance` is smaller than `rmin`, it is clamped to `rmin`.

`impulsive_pressure_at_r` is the impulsive pressure at distance `r`.

`SimulateExplosion` can optionally build an occlusion cubemap to find entities occluded from the explosion (`nOccRe_s` should be set to a non-zero cubemap resolution in one dimension in this case). First, static entities are drawn into the cubemap, and then dynamic entities of the types specified in `iTypes` are tested against the map. Thus, dynamic entities never occlude each other.

Passing -1 to `nOccRes` tells the function to reuse the cubemap from the last call and process only the dynamic entities that were not processed during the last call. This is useful when the code that creates the explosion decides to spawn new entities afterwards, such as debris or dead bodies, and wants to add explosion impulses to them without recomputing the occlusion map.

Due to the projective nature of the cubemap, small objects very close to the epicenter can occlude more than they normally would. To counter this, `rmin_occ` can specify linear dimensions of a small cube that is subtracted from the environment when building the occlusion map. This crops the smaller objects but can make explosions go through thin walls, so a compromise set of dimensions should be used.

`nGrow` specifies the number of occlusion cubemap cells (in one dimension) that dynamic entities are inflated with. This can help explosions to reach around corners in a controllable way. After a call has been made to `SimulateExplosion`, the physics system can return how much a particular entity was affected by by calling `IsAffectedByExplosion`.

`IsAffectedByExplosion` returns fraction from zero to one. The `IsAffectedByExplosion` function performs a lookup into a stored entity list; it does not recompute the cubemap. The explosion epicenter used for generating impulses can be made different from the one used to build a cubemap. For example, you can create explosions slightly below the ground to make things go up instead of sideways. Note that this function processes only parts with `geom_colltype_explosion`.

Profiler

Profiler is in preview release and is subject to change.

Profiler is a Lumberyard tool that can capture, save, and analyze network, CPU, and VRAM usage statistics. You can used the saved data to analyze network usage frame by frame, fix problems in the use of network bandwidth, and optimize the performance of your game.

To capture data, Profiler works with GridHub. When you launch Profiler, GridHub launches automatically as a supporting background process. For more information about GridHub, see Using GridHub (p. 1066).

Topics

- Profiler Tutorial (p. 1038)
- Creating and Using Annotations (p. 1048)
- Using Profiler for Networking (p. 1051)
- Using the Profiler for CPU Usage (p. 1060)
- Using Profiler for VRAM (p. 1063)
- Using GridHub (p. 1066)

Profiler Tutorial

Profiler is in preview release and is subject to change.

You can register an application in GridHub and use Profiler to capture, inspect, play back, and export the data that you collect.

Topics

- Registering Your Application (p. 1039)
- Launching Profiler (p. 1039)
- Capturing Data (p. 1039)

- Inspecting Data (p. 1041)
- Playing Back Data (p. 1043)
- Exporting Data (p. 1048)

Registering Your Application

To enable Profiler to capture information from your application, you must first register the application in GridHub. To do so, add `AzFramework::TargetManagementComponent` to the application's `SystemComponent`.

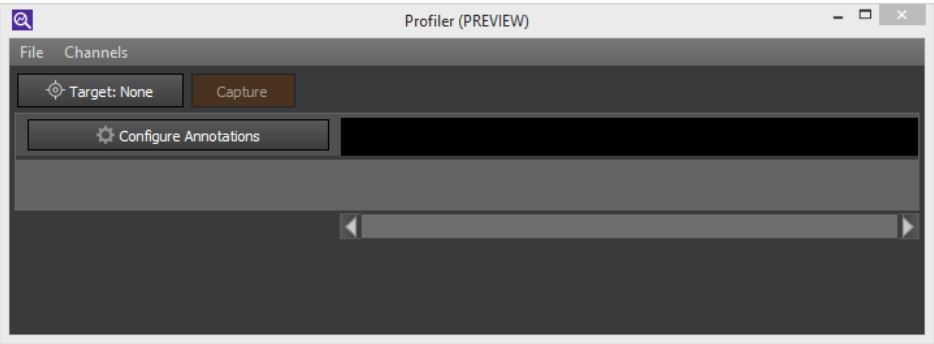
Note: Lumberyard's built-in applications already have this component added by default.

Launching Profiler

Unlike many Lumberyard utilities, you launch Profiler from its own executable file.

To launch profiler

- From the Lumberyard `dev\Bin64\` directory, run `Profiler.exe`.



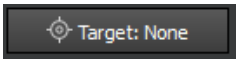
Capturing Data

Profiler has two main modes of use: *capture mode* and *inspection mode*.

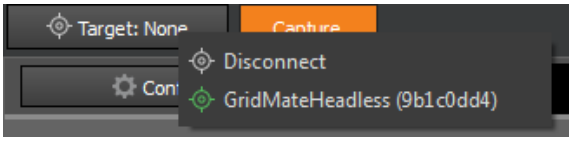
To use capture mode, perform the following steps.

To capture data

1. Click **Target**.

   Profiler shows you the applications that are available for profiling:

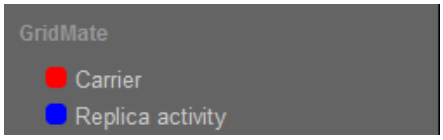
   

2. Select a target application.

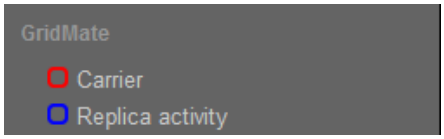
   After you have selected a target, the target selector shows the state of the connection to the target. The next time you launch Profiler, it automatically selects your target for you, if it's available.

The window is divided horizontally into channels that have associated Profiler instances. A channel is a collection of Profiler instances that relate to a specific system.

3. Each Profiler instance in a channel has a unique color. A Profiler instance is active when its color is solid:

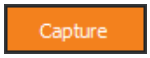


Click the color next to a Profiler instance. The color is no longer solid, showing that the Profiler instance is inactive:

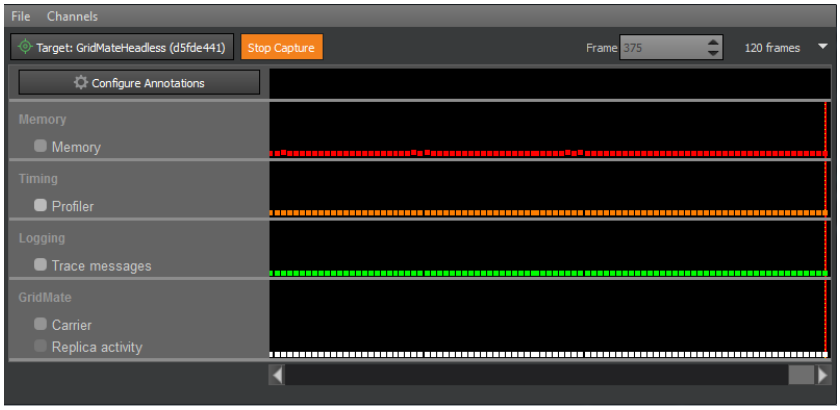


Click the color again to turn on the display and activate the instance again.

4. After you have selected a target and chosen the Profiler instances that you want to see, click **Capture**.



After the capture begins, data begins to populate the channels.



5. To stop the data capture, click **Stop Capture**.

6. When prompted, save the captured data to disk. Profiler saves the data in a binary format file with a `.drl` extension, reloads the data from disk, and switches to inspection mode.

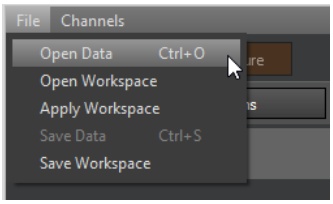
   Note
   If you do not save the data, it will be discarded.

Inspecting Data

You can use profiler to examine the data that you have captured.

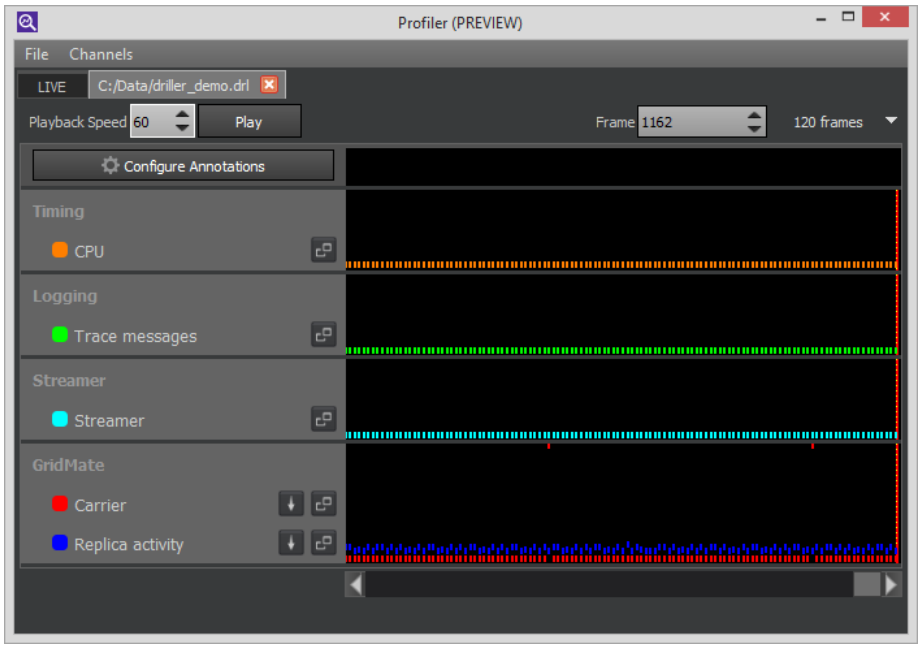
To inspect captured data

1. In Profiler, click **File**, **Open Data**, or press **Ctrl+O**:



2. Navigate to the `.drl` file that contains your saved data and open it.

   The main screen of the Profiler provides an overview of the channels of system information. This example uses a file that has 1162 frames of data:

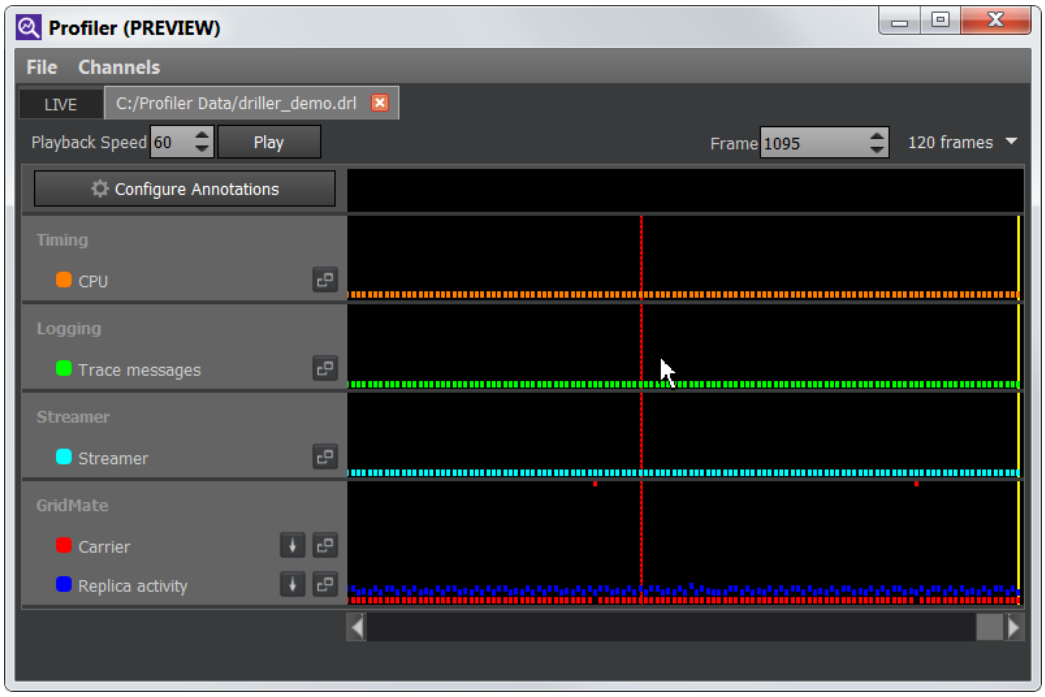


   You can use this main view to discover anomalies across channels, or to examine particular areas of interest at a high level.

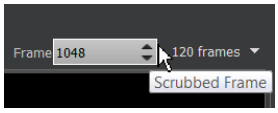
   When you open the main window, the scroll box at the bottom is on the right because the playback stopped at the end of the captured data.

   Notice the red vertical line on the right.

3. Click in the channels area of the window.

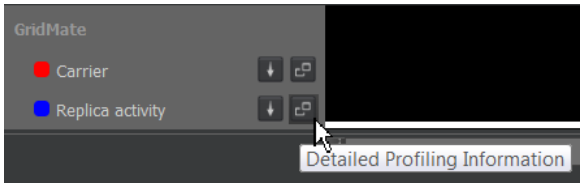


The red vertical line moves to where you clicked. The frame indicator shows the new position of the red line. You can place the red line, which is called the *scrubber*, on any frame that you want to examine in detail. For finer control over the position of the scrubber, you can enter a number in the **Frame** indicator.



The scrubber moves accordingly.

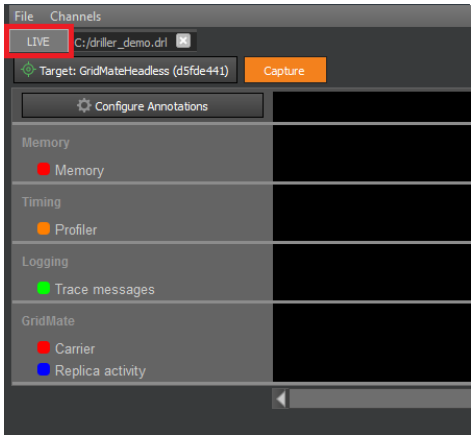
4. To view detailed information about a frame on which the scrubber rests, the click the **Detailed Profiling Information** icon next to the profiler instance whose data you would like to see:



Profiler instance information appears in a detail window.

Individual profilers present details in different ways, so their detail windows can look different. For information on system-specific detail windows in Profiler, see Using Profiler for Networking (p. 1051), Using the Profiler for CPU Usage (p. 1060), and Using Profiler for VRAM (p. 1063).

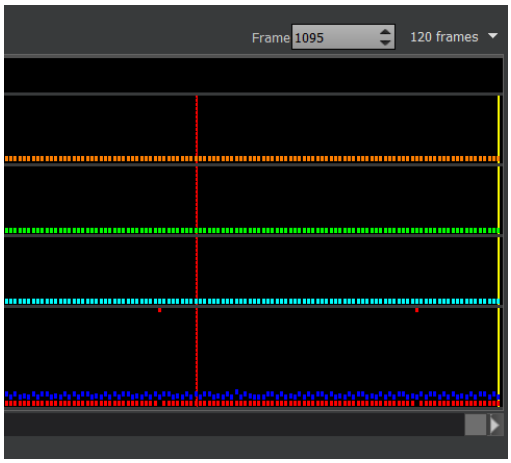
5. To return to capture mode from inspection mode, click the **LIVE** tab.



Playing Back Data

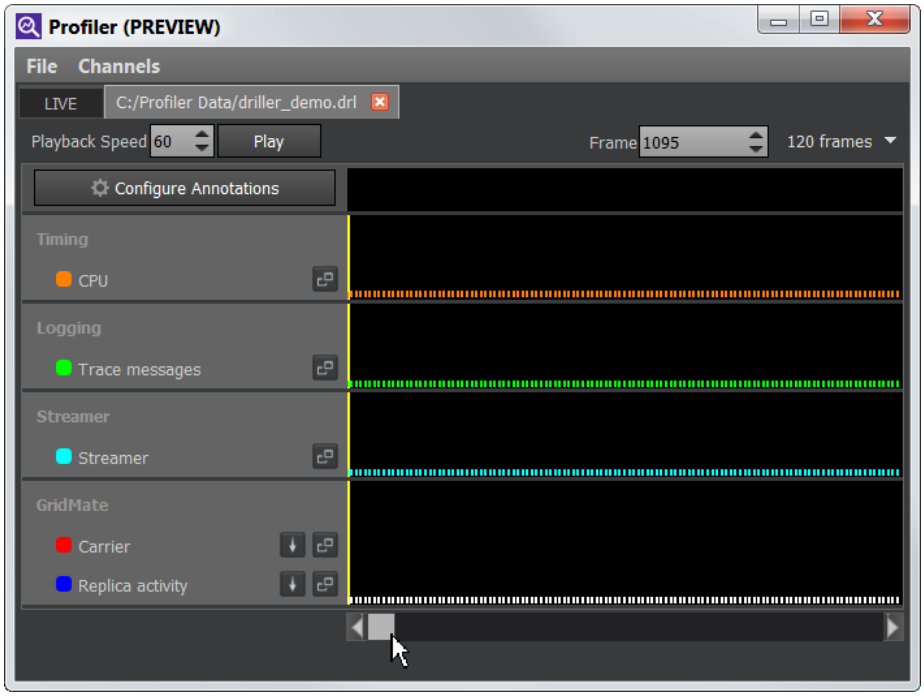
You can mark and play back a subset of your captured data.

Notice that after you moved the scrubber the first time, a yellow vertical line appeared on the right at the end of the data:



This yellow marker is movable and marks the end of your desired playback range. By default, it is at the end of the captured data but may be obscured by the red scrubber.

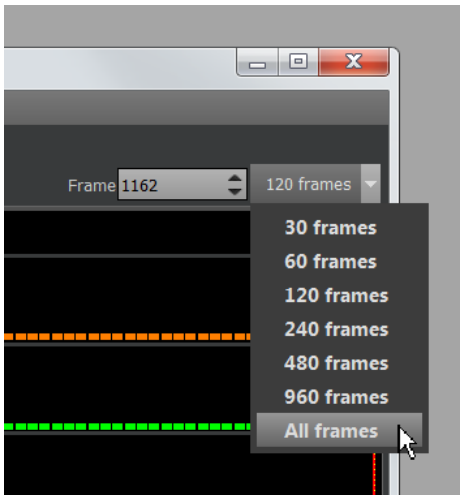
1. Scroll the window all the way to the left, to the beginning of the capture range.



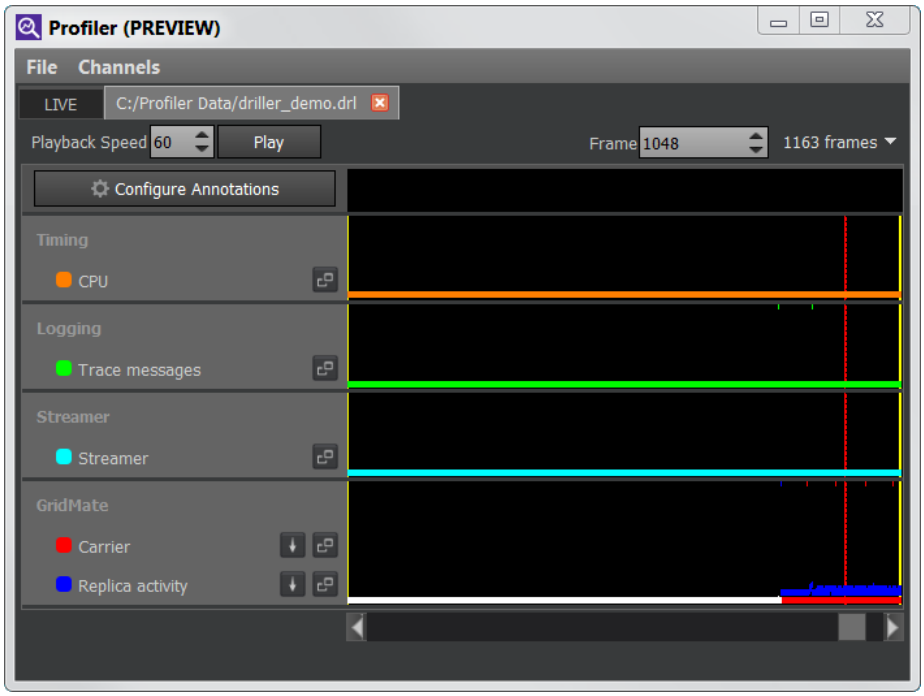
Now a yellow marker also appears at the beginning of the data. You can use these two yellow markers, which by default are at the beginning and end of the capture range, to restrict the range of playback to an area of data that you are interested in. You will use these shortly.

If you have many frames of data (as in this example), the initial view does not show you all frames by default.

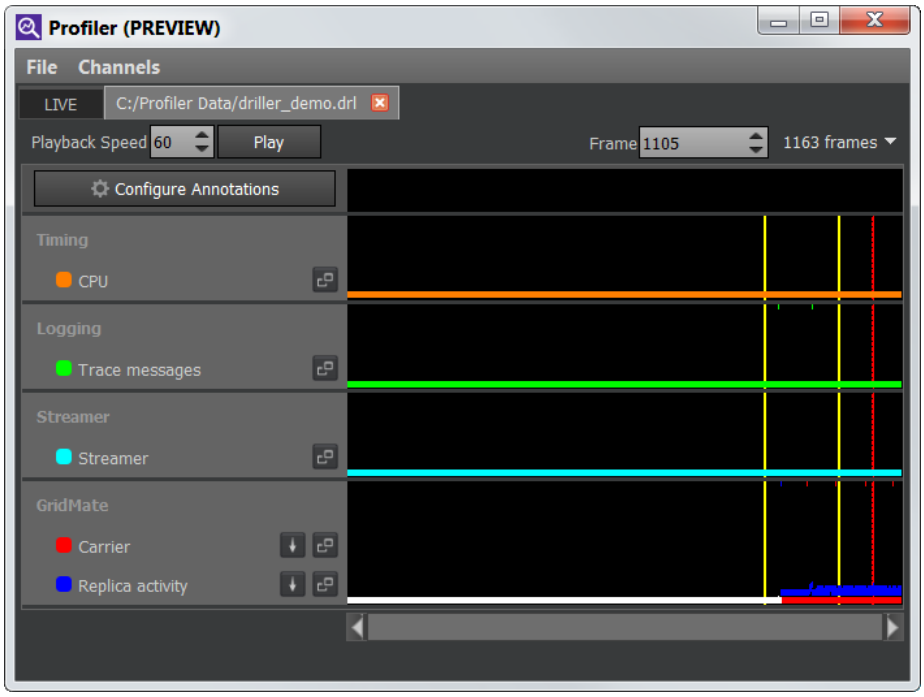
2. To see all frames at once, click the **Frame Count Selector**, which determines the number of frames visible, and choose **All frames**:



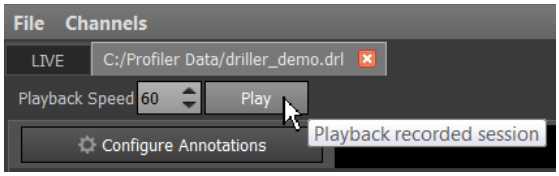
Now you can see the entire range of captured data, with the yellow markers at the beginning and at the end:



3. Drag the two yellow markers to an area of data that you want to play back. You can ignore the position of the scrubber for now.



4. Click **Play** to start the playback:



As the data plays back, the scrubber moves from the first yellow marker to the second, and then loops back to the first.

Here are some tips to keep in mind:

- If the playback speed is too fast (the default is 60), use the **Playback Speed** option to adjust it from 1 through 60.
- If you click a location in the playback window during playback, the playback stops and moves the scrubber to the location that you clicked.
- You can place the scrubber on a frame that you are interested in and click the detail button for a profiler instance to see the detail window for the frame.
- For greater convenience and visibility, leave the profiler instance detail window open to see the data change in the detail window as the scrubber loops between markers.

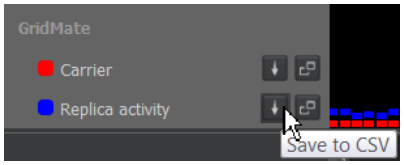
5. Click **Stop** to stop the playback.

Exporting Data

Some Profiler instances have an export option that you can use to save data to a `.csv` file.

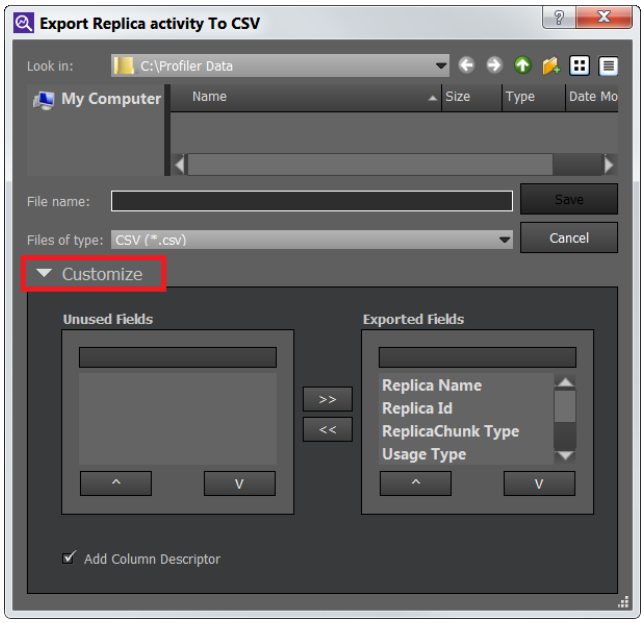
To export data from a Profiler instance to a `.csv` file

1. Click the **Save to CSV** icon for the Profiler instance whose data you want to save:



   Note
   Not all profilers have the data export option.

2. To choose the fields that you want to export, click **Customize** in the export dialog box.



Creating and Using Annotations

Profiler is in preview release and is subject to change.

In Profiler, annotations are a convenient way of highlighting per-frame log information from the data captured from your application. After you learn how annotations are used in Profiler, you can modify your application so that they appear in Profiler.

Topics

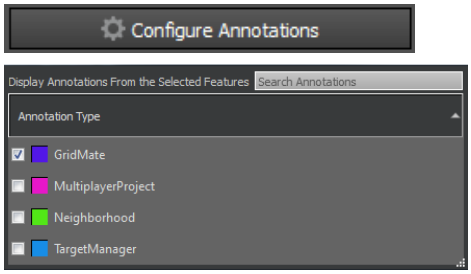
- Using Annotations (p. 1049)
- Creating Annotations (p. 1050)
- Viewing Annotations in Trace Messages Profiler (p. 1051)

Using Annotations

Annotations in the Lumberyard Profiler tool flag frames in your captured data that have associated log information. By default, annotations are turned off.

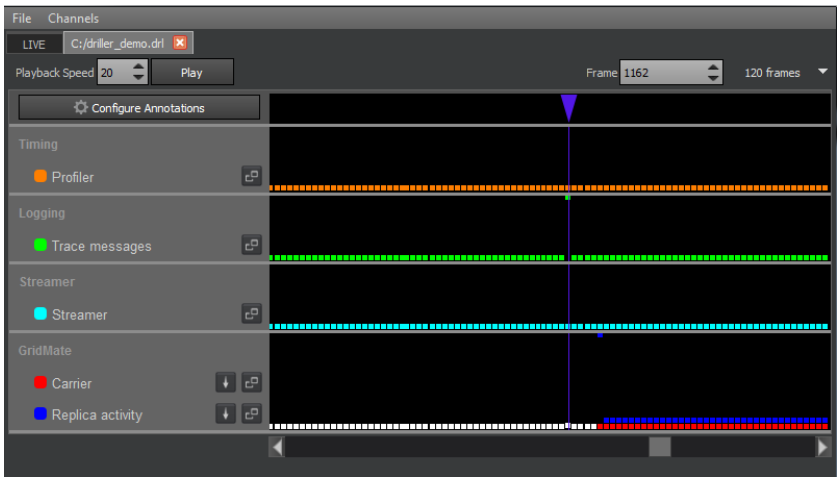
To use annotations

1. To turn on annotations in the Lumberyard Profiler tool, click **Configure Annotations**:

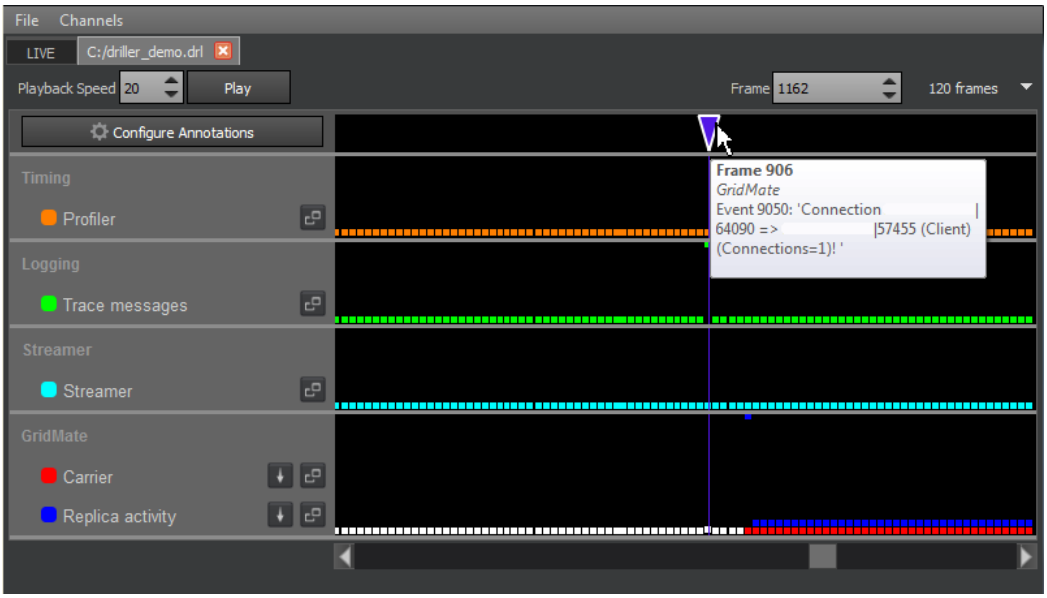


   The **Configure Annotations** dialog box contains a list of available annotations and their display colors. For information on creating annotations for your application, see Creating Annotations (p. 1050).

2. When you select an annotation in the dialog box, a marker and line of the same color appears in the channel display. Note that you might have to scroll horizontally to find the marker.



3. To display details for the annotations that occurred on a frame, pause your pointer on an annotation marker. In the example image, IP addresses have been redacted out.



Creating Annotations

To create an annotation, you add one or more lines of C++ logging code to your application. The added code instructs Lumberyard's logging system to include the logging information that you specify as a part of your capture. Lumberyard transforms the logged messages into annotations for you. Then, in Profiler, when you click **Configure Annotations**, you actually choose which system's annotations are displayed (for example, **GridMate** or **MultiplayerProject**).

To create an annotation, place a line of C++ code like the following in your application:

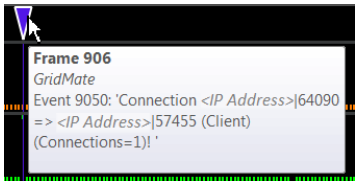
```
AZ_TracePrintf("GridMate","Connection %s => %s (%s) (Connections=%d!\n")
```

The first parameter is the window (that is, system) of the trace (in this case, `GridMate`), and the second is the content of the trace that will be shown as the annotation.

The example results in the following annotation text:

```
GridMate - Connection <IP_Address>|64090 => <IP_Address>|57455 (Client) (Connections=1)!
```

The text displays in Profiler like this:



Alternatives to AZ_TracePrintf

In your code, instead of using `AZ_TracePrintf`, you can use `AZ_Error` or `AZ_Warning`, depending on the degree of severity that you want. `AZ_TracePrintf` always logs a message, but is of the lowest concern from an inspection viewpoint.

The following example uses `AZ_Error`:

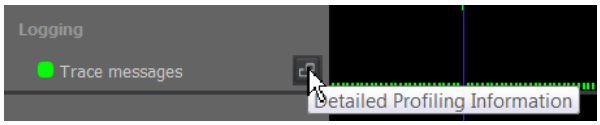
```
if (networkTableContext.ReadValue(elementIndex,forcedDataSetIndex))
{
    AZ_Error("ScriptComponent",forcedDataSetIndex >= 1 && forcedDataSetIndex <=
 ScriptComponentReplicaChunk::k_maxScriptableDataSets,"Trying to force Property (%s) to an
 invalid DataSetIndex(%i).",scriptProperty->m_name.c_str(),forcedDataSetIndex);
    if (forcedDataSetIndex >= 1 && forcedDataSetIndex <=
 ScriptComponentReplicaChunk::k_maxScriptableDataSets)
    {
        networkedTableValue.SetForcedDataSetIndex(forcedDataSetIndex);
    }
}
else
{
    AZ_Error("ScriptComponent",false,"Trying to force Property (%s) to unknown
 DataSetIndex. Ignoring field.", scriptProperty->m_name.c_str());
}
```

In the example, if either of the error conditions occur, an annotation is created.

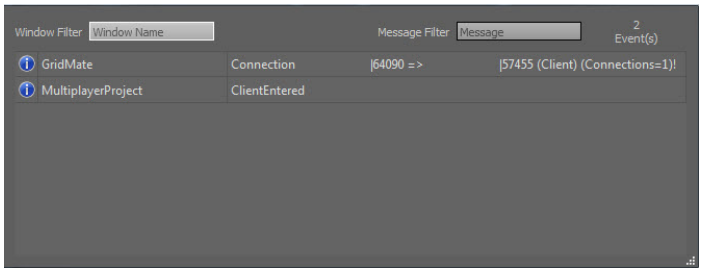
Viewing Annotations in Trace Messages Profiler

Another way to confirm that your annotations are in place is by using the Trace Messages profiler.

In the Profiler **Logging** channel, click the **Trace messages** profiler details icon to see the logging systems currently in place:



The **Trace messages** profiler instance shows all the trace messages that were generated from the start of the capture to the currently analyzed frame. Messages are shown with the oldest message at the top and the newest message at the bottom:



You can use the **Window Filter** to show the system and/or **Message Filter** to show the message text that you are interested in.

The following example, filtered by "GridMate", shows the message specified by the line of code that was added to the application:

Using Profiler for Networking

Profiler is in preview release and is subject to change.

You can use the Lumberyard Profiler tool to examine how your game uses network bandwidth, including its GridMate carrier connections and replica activity. You can use network-specific profilers to drill down further into the activity of specific replica chunks, RPCs, and data sets.

Prerequisites

This topic assumes familiarity with Lumberyard networking and the Lumberyard Profiler tool. For information on Lumberyard networking, see Networking System (p. 951). For an introduction to the Profiler tool, see Profiler (p. 1038).

Topics

- Carrier Profiler (p. 1052)
- Replica Activity Profiler (p. 1053)

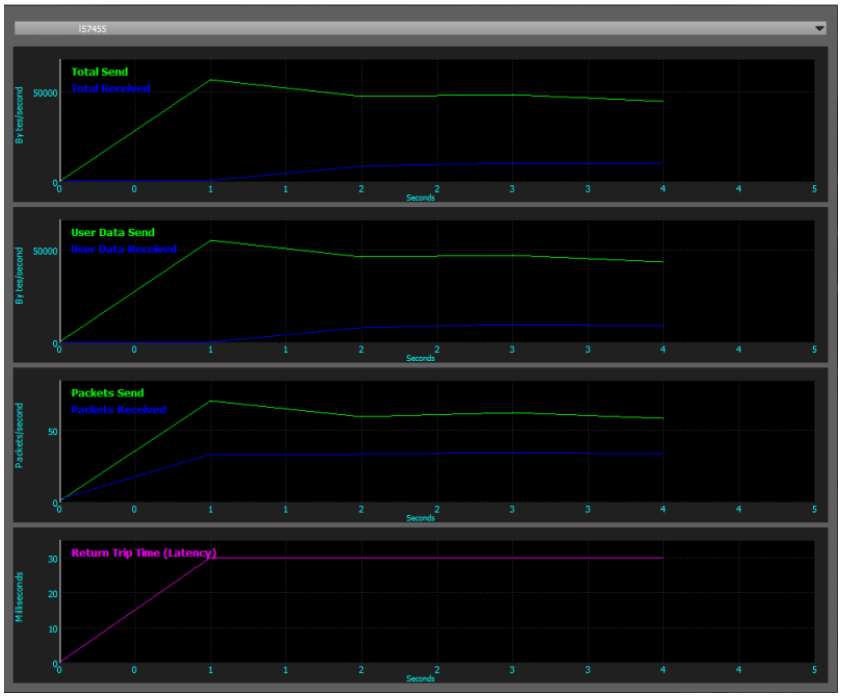
Carrier Profiler

The Profiler tool has a GridMate channel with **Carrier** and **Replica activity** profiler instances. You can use the Carrier profiler detail view to examine the bandwidth usage of a selected GridMate carrier connection.

To open the detail view for the Carrier profiler

- Click the **Detailed Profiling Information** icon for **Carrier** in the GridMate channel:

The Carrier profiler detail view resembles the following image:



This view uses all of the data supplied in the capture session to show an overview of the bandwidth usage through the GridMate carrier for the selected connection. It includes the following information:

- **Total Sent/Total Received** – The total number of bytes sent and the total number of bytes received on the selected connection.

- **User Data Sent/User Data Received** – The user data sent and the user data received on the selected connection. This data does not include the overhead associated with carrier or connection maintenance.

- **Packets Sent/Packets Received** – The number of packets sent and the number of packets received.

- **Return Trip Time (Latency)** – How many seconds the packets took to make a return trip.

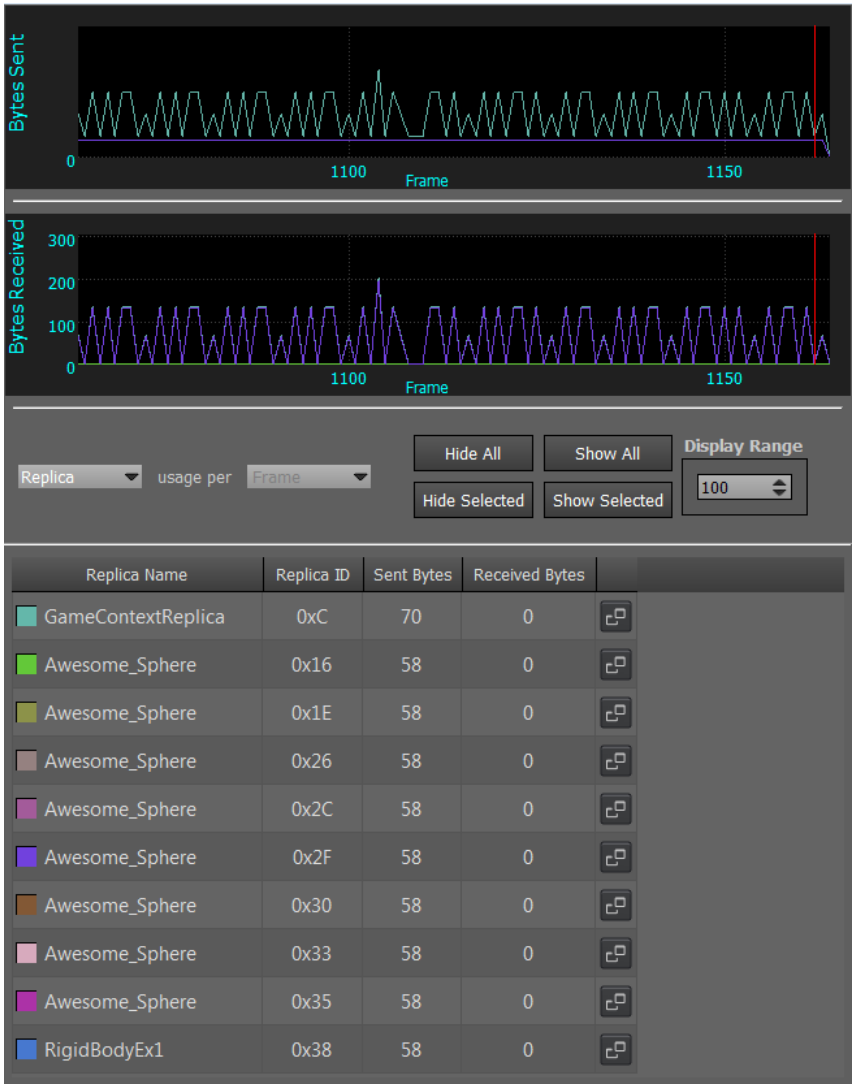
Replica Activity Profiler

You can use the Replica Activity profiler to see how much replica bandwidth your application is using.

To open the Replica Activity profiler

- Click the **Detailed Profiling Information** icon for **Replica activity**.

The Replica Activity profiler detail view has a pair of **Bytes Sent** and **Bytes Received** graphs at the top, a toolbar to control the display in the middle, and a table of replicas at the bottom:



This view is useful for discovering how much bandwidth a single entity is using, and for finding what information is synchronized in response to particular events for particular entities.

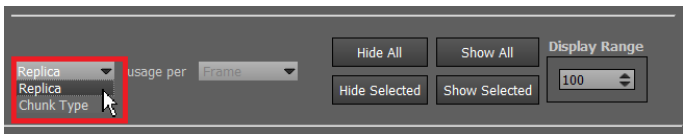
Two main detail views are available for replica activity: **Replica** and **Chunk Type**. The view defaults to **Replica**, but Profiler remembers your most recent choice and uses it the next time you view replica activity details.

Using Replica View

In replica view, the table shows how much data each replica used in a given frame.

To change the view to Replica

- In the toolbar, choose **Replica**.



Each replica is represented by its associated color in the graphs above the toolbar. Replica view includes the following information:

- **Bytes Sent** – Shows bandwidth usage in bytes sent by the object for a particular frame.
- **Bytes Received** – Shows bandwidth usage in bytes received by the object for a particular frame.

To display or hide an individual line in the graph

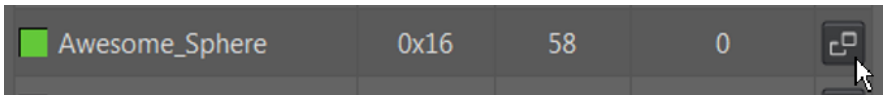
- Double-click the associated row in the tree.

  The toolbar also offers the following options:

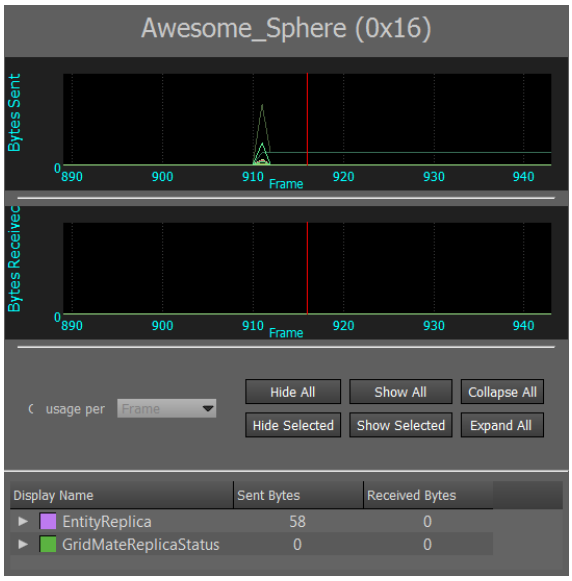
  - **Hide All** – Hides the line graphs of all replicas in the table.
  - **Show All** – Shows the line graphs for all replicas in the table.
  - **Hide Selected** and **Show Selected** – Use **Ctrl+click** to select individual replicas in the table, and then click **Hide Selected** or **Show Selected** to hide or show the graphs for the replicas that you selected.
  - **Display Range** – Determines the number of frames that are shown in the graph, with the currently selected frame in the center. You can use this option to zoom in or out on the data.

To display replica chunk details for a particular replica

- Click its details icon.

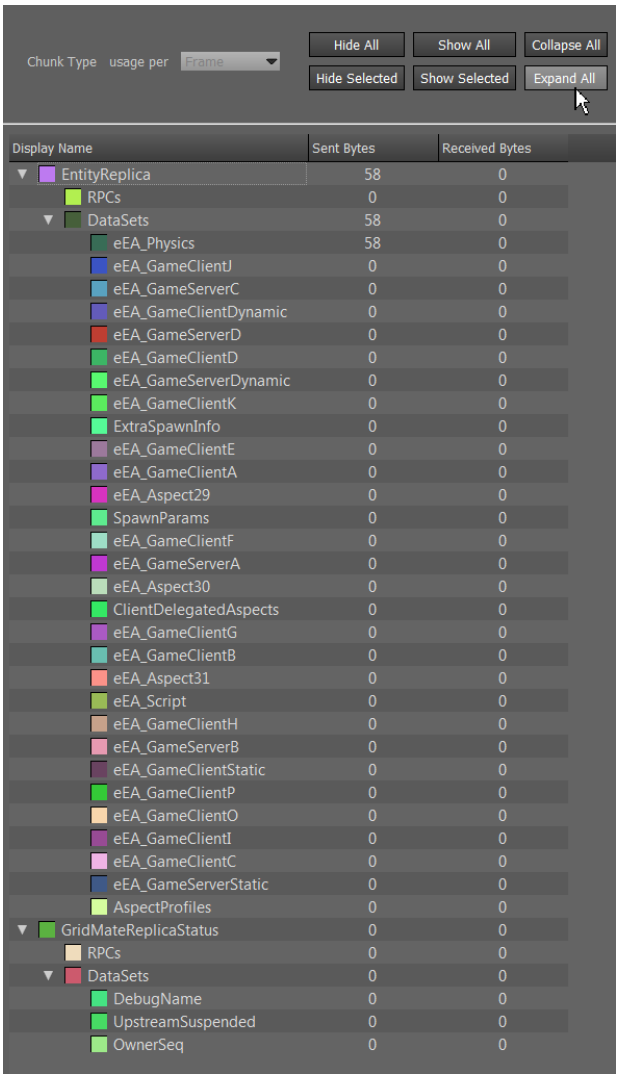


The graph shows the bytes sent and received for a replica chunk, data set, and RCP:



You can use this details view to see what replica chunk types a given replica is using, how much data each replica chunk type is using, and how much bandwidth individual data sets and RPCs are using.

Tip
Click **Expand All** to list all replica chunks in all replicas, and every data set and remote procedure call (RPC) in each replica chunk:



To use the Replica Activity profiler tree view

- Do either of the following:
  - Select a row to highlight its corresponding line in the graph.
  - Double-click a row to display or hide the graph for the row.

  The following information is available:

  - **Display Name** – The debug name associated with the corresponding row of the table.
  - **Sent Bytes** – The number of bytes sent for an item, including all information sent by children of the item.
  - **Received Bytes** – The number of bytes received by an item, including all information received by children of the item.

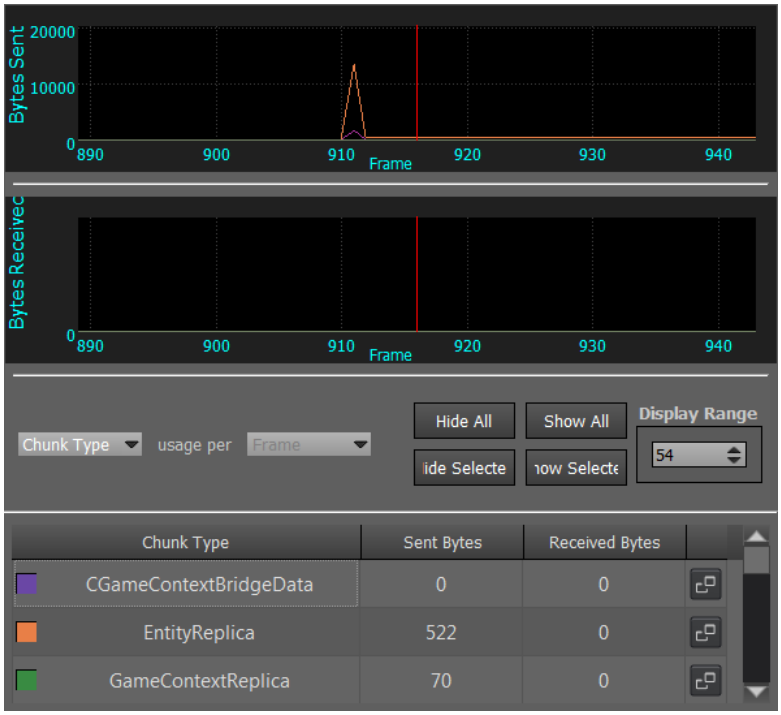
Chunk Type View

Chunk type view shows you how much data each chunk type used in a given frame. The view is useful for seeing how much information a particular system might be using across all entities.

To change the view to Chunk Type

- In the toolbar on the main detail page for **Replica activity**, choose **Chunk Type**.

The chunk type view shows how much data a particular replica chunk type is using in a given frame:

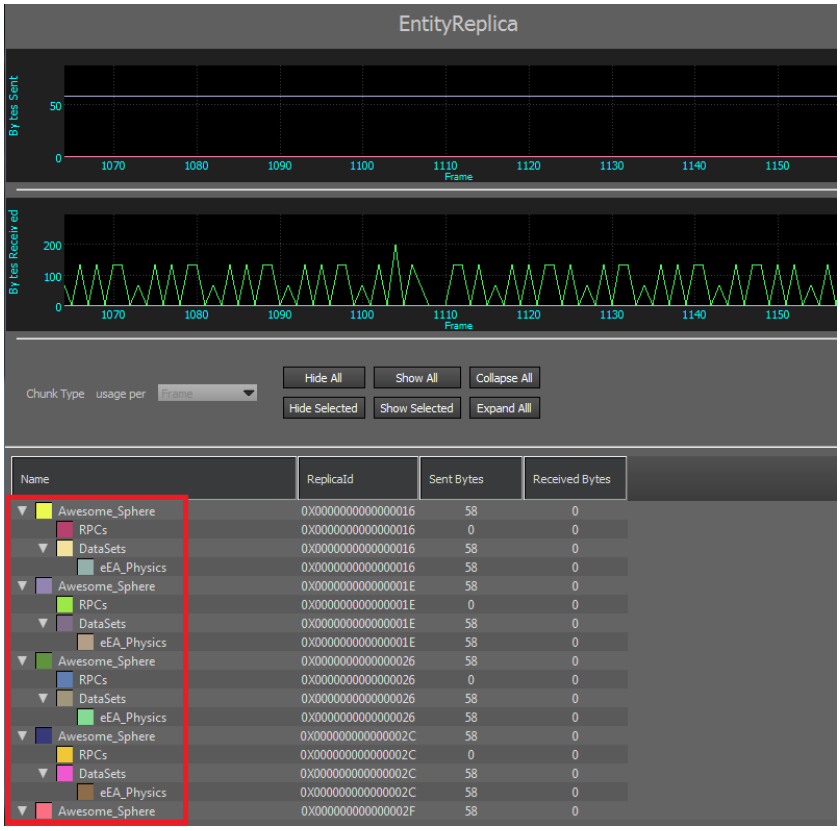


To inspect chunk type details

- Click the details icon for the chunk type:

The details window shows which replicas are using a chunk type's bandwidth, how much data they are using, and how much data the individual data sets and RPCs are using:

As before, you can expand the items in the tree to see detailed information about each:



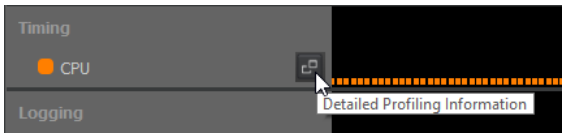
Using the Profiler for CPU Usage

Profiler is in preview release and is subject to change.

The CPU profiler gathers usage statistics about how long a function or method executed, how many times it was executed, who called it, and how much of a frame was spent on it. You can combine this information to get a systemwide view of usage, or isolate particular systems by filtering for specific threads.

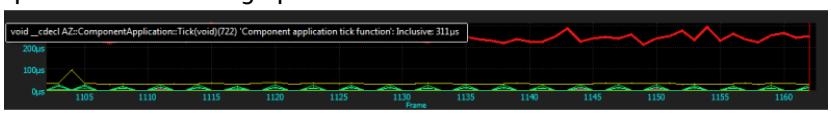
To use the CPU profiler

1. To open the detail view for the CPU profiler, click the **Detailed Profiling Information** icon for the **CPU** profiler instance.

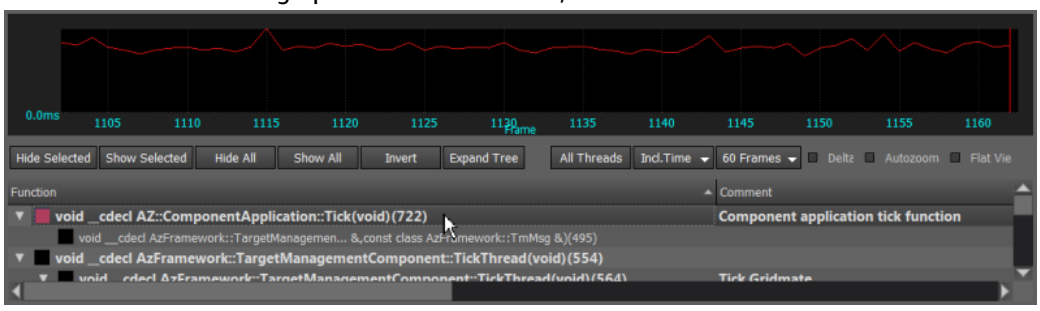


   The CPU details view has a graph of CPU usage, a toolbar, and a tree view of calls made in a frame. Each call in the tree view has the same color as its corresponding line in the graph:

2. Pause your mouse on a line in the graph to see the call that the line represents and to display the specific value for the graph at the area near the cursor.



3. To show or hide the line graph of a row in the tree, double-click the row.



Understanding the Tree View

The CPU profiler tree view represents a call hierarchy of profiler log points (called *hooks*). A profiler hook that is active while another call is active shows as a child of the first hook. The hooks act as a stack: The last hook that was pushed onto the stack is the parent of the hook that was pushed onto the stack before it. The tree view has the following information:

Function

The function declaration where the profiler data point was generated.

Comment

A user-defined message that distinguishes specific events in the same function.

Excl. Time (Micro)

(Exclusive time) The time, in microseconds, spent executing this function and no other functions called by this function.

Incl. Time (Micro)

(Inclusive time) The time, in microseconds, spent executing this function and other functions called by this function.

Excl. Pct

(Exclusive percent) Exclusive time represented as a percent of total run time.

Incl. Pct

(Inclusive percent) Inclusive time represented as a percent of total run time.

Calls

The number of calls to this function.

Child Time (Micro)

The time, in microseconds, that functions that were called by this function took to execute.

Total Time (Micro)

A running total of the time, in microseconds, that was spent inside of this function.

Child Calls

How many functions this function called.

Total Calls

The running total of how many times this function was called.

Thread ID

The thread on which this function was executed

Controlling the Display

You can use the toolbar to control how the captured CPU data is displayed:

Hide Selected

Hide the graph of the rows selected in tree view.

Show Selected

Show the graph of the rows selected in tree view.

Hide All

Hides the graph of all rows in the tree view.

Show All

Shows the graphs of all rows in the tree view.

Invert

Shows graphs for all rows in the tree view that are hidden; hides the graphs of all rows in the tree view that are showing.

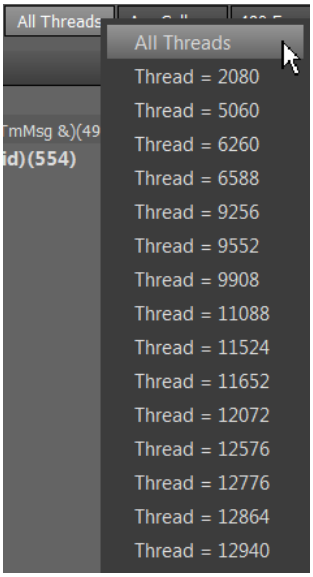
Expand Tree

Expands all rows in the tree view hierarchy.

The right side of the toolbar offers more options:

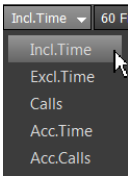
All Threads

Use the thread selector to control which threads are shown in the tree view and in the graph:



Incl. Time

Use this selector to choose the meaning of the time displayed.

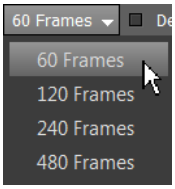


- **Incl. Time** – (Inclusive time) The time spent in this function inclusively.

- **Excl. Time** – (Exclusive time) The time spent in this function exclusively.
- **Calls** – The number of times this function was called in the frame.
- **Acc. Time** – (Accumulated time) The total amount of time spent in this function up to the frame being analyzed.
- **Acc. Calls** – (Accumulated calls) – The total number of times this function was called up to the frame being analyzed.

<*number*> Frames

Use this selector to choose how frames of history are displayed in the graph:



Delta

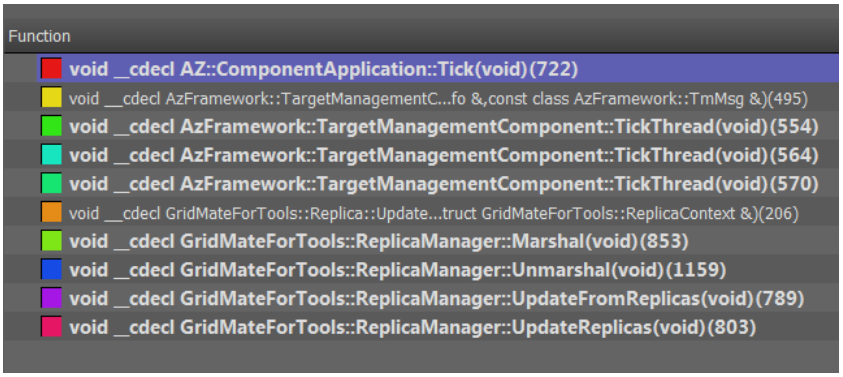
Unused option.

Autozoom

When selected, maintains the approximate zoom level (number of frames displayed) whenever the graph changes.

Flat View

Flattens the tree of function calls (removes the hierarchical indentation), as in the following image:



Using Profiler for VRAM

Profiler is in preview release and is subject to change.

You can use the video memory profiler (VRAM profiler) to determine which resources are contributing most to run-time VRAM usage in your game.

The VRAM profiler records the amount of video memory used by a game, including how many memory deallocations and allocations occurred during the capture. This latter information is useful in tracking down rendering performance bottlenecks.

You can also use the memory usage information from VRAM profiler to determine your game's minimum PC GPU (graphics processing unit) memory requirements, or to determine whether your game will run out of memory on a console or mobile device.

Topics

- Notes (p. 1064)
- Understanding the Captured Data (p. 1064)
- Inspecting the Data (p. 1065)

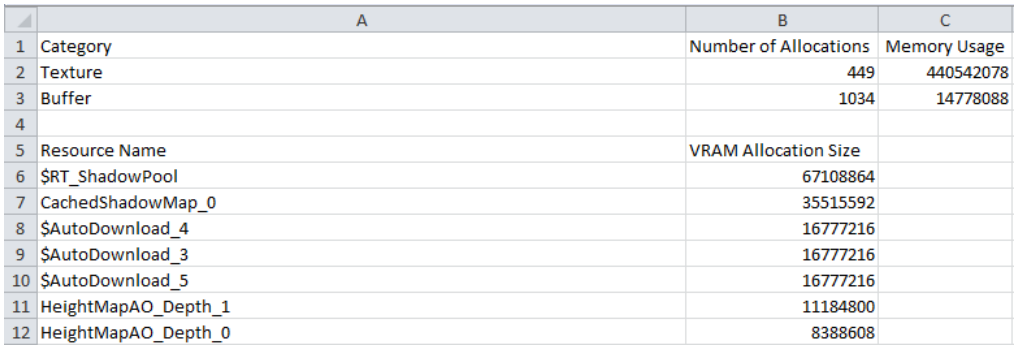
Notes

The VRAM profiler has the following attributes:

- The VRAM profiler has no graph view or tree view.
- The only supported export format is `.csv`. For steps on saving Profiler data to a `.csv` file, see Exporting Data.
- Lumberyard uses a variety of memory pooling schemes, so the actual allocated amount of VRAM is slightly more than what is reported.

Understanding the Captured Data

The following image shows how your saved `.csv` file appears in a spreadsheet application:



	A	B	C
1	Category	Number of Allocations	Memory Usage
2	Texture	449	440542078
3	Buffer	1034	14778088
4			
5	Resource Name	VRAM Allocation Size	
6	$RT_ShadowPool	67108864	
7	CachedShadowMap_0	35515592	
8	$AutoDownload_4	16777216	
9	$AutoDownload_3	16777216	
10	$AutoDownload_5	16777216	
11	HeightMapAO_Depth_1	11184800	
12	HeightMapAO_Depth_0	8388608	

The captured data contains essentially two tables of information: an overview of memory allocation and usage (divided between texture and buffer assets), and a list of resources with the amount of VRAM that was allocated for each during the capture.

Detailed information about each heading follows.

Category

Indicates the type of allocation:

- **Texture** – Includes texture assets, dynamically generated textures, and frame buffers.
- **Buffer** – Includes vertex and index buffers, constant buffers, and other run-time buffers.

Number of Allocations

The number of allocation events recorded. When the capture starts, all active allocations are sent to the profiler as a starting number. Any new allocations or deallocations will increase or decrease this number.

Memory Usage

The total size, in bytes, of VRAM used.

Resource Name

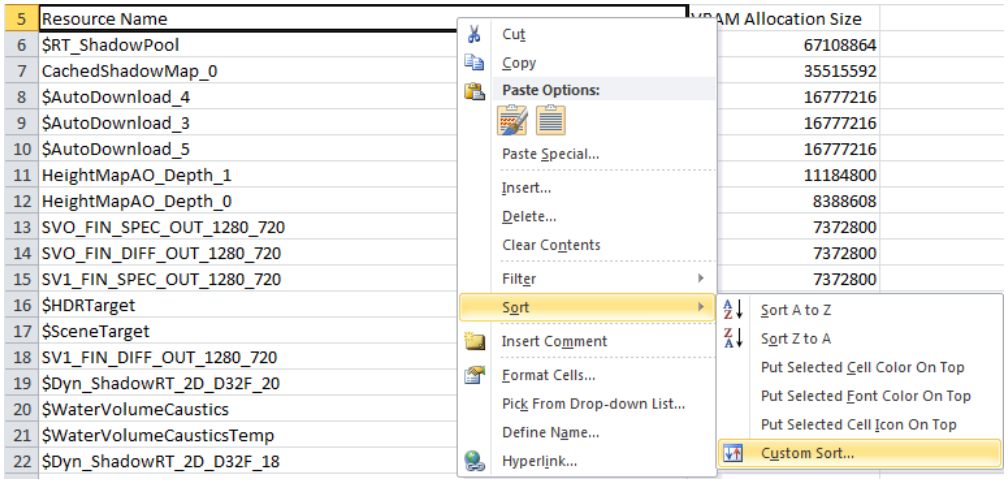
The name and full path of the allocated resource. A resource name without a path usually denotes a run-time engine resource.

VRAM Allocation Size

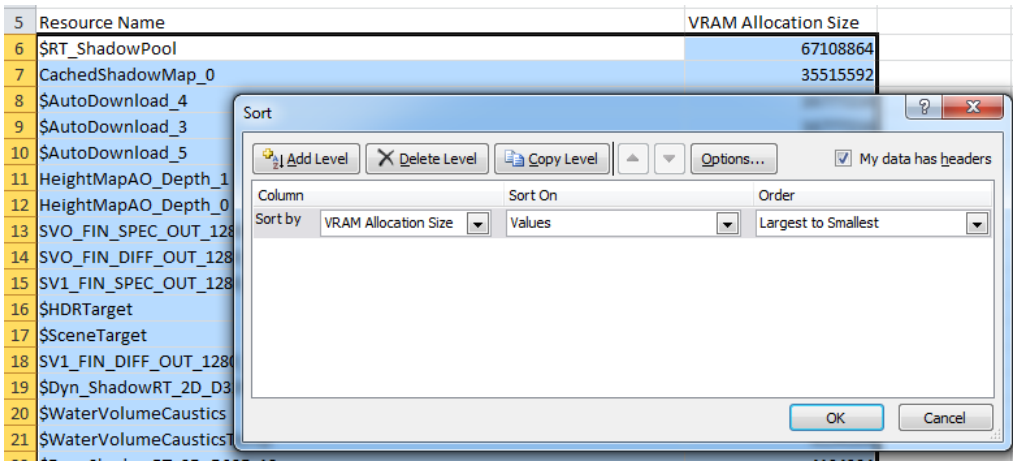
The size, in bytes, of the allocation.

Inspecting the Data

When you first open the spreadsheet, the data is unordered. To sort the data, you can use a spreadsheet application:

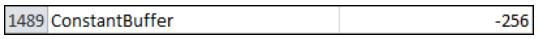


To quickly and easily identify the largest offending assets or run-time resources, sort by **VRAM Allocation Size** in descending order, or by **Resource Name** from A to Z:



Negative VRAM Allocation Sizes

Some fields may have a negative number for **VRAM Allocation Size**, as in the following image:



These important occurrences show that a VRAM deallocation event occurred during the capture. If you observe a large number of deallocation entries over a short time period, your game might be experiencing a significance decrease in performance. To improve your game's performance across all platforms, you should aim to have as few idle per-frame VRAM allocations and deallocations as possible.

Why Some Textures Are Not Reported in the .csv File

If you see a lot of allocations named `StreamingTexturePool` or entries like `$TexturePool_9_0000000002C59248`, this means the texture streaming system is active. The texture streaming system allocates all textures by default into a variety of cached texture pools. The VRAM profiler reports the size of the active streaming pools and not the names of the actual texture assets. To obtain the names and sizes of the allocated and loaded textures, set `r_TexturesStreaming=0` in your system configuration file, and then do another capture. This setting disables the texture streaming system and causes the true sizes of the texture allocations to be reported.

> **Note**
> In this situation, it is advisable to do two captures: one with `r_TexturesStreaming` enabled, and one with it disabled. When texture streaming is enabled, your VRAM usage is less because of texture eviction and the loading of lower resolution mipmap levels. The memory reporting is more accurate when texture streaming is enabled, but you get a much clearer view of your worst-case memory usage when texture streaming is disabled.

Using GridHub

GridHub is in preview release and is subject to change.

GridHub is Lumberyard's connection hub for debugging. GridHub acts as a central hub through which specified local clients connect with each other and exchange information. When you run the Lumberyard diagnostic and debugging tools `Profiler.exe` or `LuaIDE.exe` (located in the `\dev\Bin64` directory), GridHub launches as a background process in Windows and enables their functionality. For more information about Profiler, see Profiler (p. 1038).

> **Note**
> Because GridHub listens for connections on the loopback address (`127.0.0.1`), you must run GridHub on the same computer as the target application.

Topics

- Registering an Application in GridHub (p. 1066)
- Viewing and Configuring GridHub (p. 1067)
- Troubleshooting GridHub (p. 1068)

Registering an Application in GridHub

To register an application in GridHub so that Profiler can capture information from the application, add `AzFramework::TargetManagementComponent` to the application's `SystemComponent`.

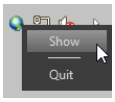
> **Note**
> Lumberyard's built-in applications already have this component added by default.

Viewing and Configuring GridHub

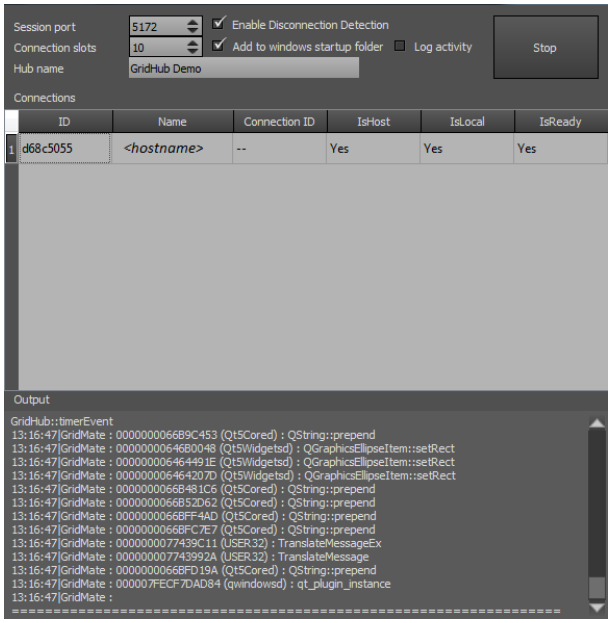
When you launch `Profiler.exe` or `LuaIDE.exe`, GridHub starts automatically and is represented by a globe icon in the Windows taskbar.

To view and configure GridHub

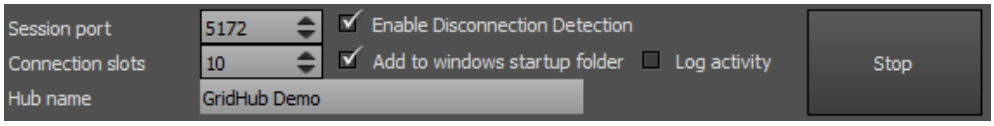
1. In the Windows taskbar, right-click the globe icon and choose **Show**:

   The GridHub window has a configuration bar, a connections pane, and pane for viewing log messages:

2. You can use the configuration toolbar to view or change GridHub configuration:

   The toolbar options are as follows:

   Session port – Specifies the port on which GridHub listens for discovery requests.

   Connection slots – Specifies the maximum number of applications that can be connected concurrently to GridHub.

   Hub name – The name of your hub. By default, this is the name of the local computer.

   > **Note**
   > The name of the hub must be the neighborhood name to which the `TargetManagementComponent` connects.

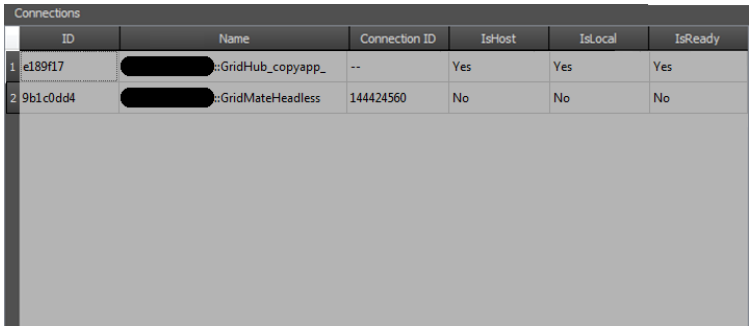
   Enable Disconnection Detection – Specifies whether the connection to GridHub is terminated when the source fails to respond.

Add to Windows startup folder – Specifies whether GridHub starts automatically when Windows starts.

Log activity – Starts or stops logging.

Start/Stop – Starts or stops GridHub. When GridHub is off, no connections are discovered or maintained.

3. When GridHub and your target application are active, your target application appears in the GridHub **Connections** list:



The columns in the **Connections** list provide the following information:

ID – The identifier of the connected application.

Name – The name of the connected application.

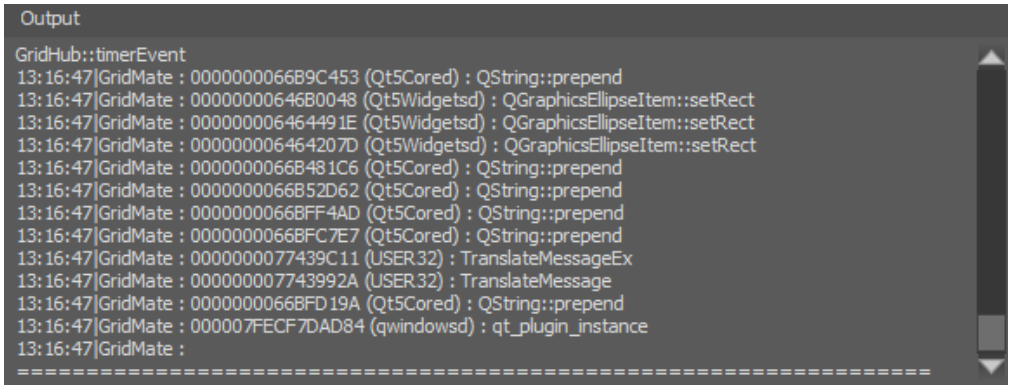
Connection ID – The identifier of the connection between GridHub and the application.

IsHost – Whether or not the connection is the connection host

IsLocal – Whether or not the connection is local.

IsReady – Whether or not the application is ready to handle further connections.

4. Use the **Output** window to see the log messages that GridHub generates as it manages connections:



When GridHub is terminated, the connections it established are also terminated.

Troubleshooting GridHub

If you experience difficulty using GridHub, check the following:

- Make sure that the neighborhood name in `TargetManagerComponent` is the same as the one in GridHub.

- Make sure that the port that GridHub is listening on is the same port as the one specified for `TargetManagementComponent`.
- Make sure that all applications are running on the same computer. The GridHub socket is bound to the loopback address `127.0.0.1`.

System

This section contains topics on general system issues, including memory handling, streaming, and localization. It also provides information on logging and console tools.

Topics

- Memory Handling (p. 1070)
- Streaming System (p. 1073)
- Text Localization and Unicode Support (p. 1082)
- CryLog (p. 1086)
- CryConsole (p. 1088)

Memory Handling

This article discusses some memory and storage considerations related to game development.

Hardware Memory Limitations

Developing for game consoles can be challenging due to memory limitations. From a production point of view, it is tempting to use less powerful hardware for consoles, but the expectations for console quality are usually higher in an increasingly competitive market.

Choosing a Platform to Target

It is often better to choose only one development platform, even if multiple platforms are targeted for production. Choosing a platform with lower memory requirements eases production in the long run, but it can degrade the quality on other platforms. Some global code adjustments (for example, TIF setting "globalreduce", TIF preset setting "don't use highest LOD") can help in reducing memory usage, but often more asset-specific adjustments are needed, like using the TIF "reduce" setting. If those adjustments are insufficient, completely different assets are required (for example, all LODs of some object are different for console and PC). This can be done through a CryPak feature. It is possible to bind multiple pak files to a path and have them behave as layered. This way it is possible to customize some platforms to use

different assets. Platforms that use multiple layers have more overhead (memory, performance, I/O), so it is better to use multiple layers on more powerful hardware.

Budgets

Budgets are mostly game specific because all kinds of memory (for example, video/system/disk) are shared across multiple assets, and each game utilizes memory differently. It's a wise decision to dedicate a certain amount of memory to similar types of assets. For example, if all weapons roughly cost the same amount of memory, the cost of a defined number of weapons is predictable, and with some careful planning in production, late and problematic cuts can be avoided.

Allocation Strategy with Multiple Modules and Threads

The Lumberyard memory manager tries to minimize fragmentation by grouping small allocations of similar size. This is done in order to save memory, allow fast allocations and deallocations and to minimize conflicts between multiple threads (synchronization primitives for each bucket). Bigger allocations run through the OS as that is quite efficient. It is possible to allocate memory in other than the main thread, but this can negatively impact the readability of the code. Memory allocated in one module should be deallocated in the same module. Violating this rule might work in some cases, but this breaks per module allocation statistics. The simple `Release()` method ensures objects are freed in the same module. The string class `(CryString)` has this behavior built in, which means the programmer doesn't need to decide where the memory should be released.

Caching Computational Data

In general, it is better to perform skinning (vertex transformation based on joints) of characters on the GPU. The GPU is generally faster in doing the required computations than the CPU. Caching the skinned result is still possible, but memory is often limited on graphics hardware, which tends to be stronger on computations. Under these conditions, it makes sense to recompute the data for every pass, eliminating the need to manage cache memory. This approach is advantageous because character counts can vary significantly in dynamic game scenes.

Compression

There are many lossy and lossless compression techniques that work efficiently for a certain kind of data. They differ in complexity, compression and decompression time and can be asymmetric. Compression can introduce more latency, and only few techniques can deal with broken data such as packet loss and bit-flips.

Disk Size

Installing modern games on a PC can be quite time consuming. Avoiding installation by running the game directly from a DVD is a tempting choice, but DVD performance is much worse than hard drive performance, especially for random access patterns. Consoles have restrictions on game startup times and often require a game to cope with a limited amount of disk memory, or no disk memory at all. If a game is too big to fit into memory, streaming is required.

Total Size

To keep the total size of a build small, the asset count and the asset quality should be reasonable. For production it can make sense to create all textures in double resolution and downsample the content

with the Resource Compiler. This can be useful for cross-platform development and allows later release of the content with higher quality. It also eases the workflow for artists as they often create the assets in higher resolutions anyway. Having the content available at higher resolutions also enables the engine to render cut-scenes with the highest quality if needed (for example, when creating videos).

Many media have a format that maximizes space, but using the larger format can cost more than using a smaller one (for example, using another layer on a DVD). Redundancy might be a good solution to minimize seek times (for example, storing all assets of the same level in one block).

Address Space

Some operating systems (OSes) are still 32-bit, which means that an address in main memory has 32-bits, which results in 4 GB of addressable memory. Unfortunately, to allow relative addressing, the top bit is lost, which leaves only 2 GB for the application. Some OSes can be instructed to drop this limitation by compiling applications with large address awareness, which frees up more memory. However, the full 4 GB cannot be used because the OS also maps things like GPU memory into the memory space. When managing that memory, another challenge appears. Even if a total of 1 GB of memory is free, a contiguous block of 200 MB may not be available in the virtual address space. In order to avoid this problem, memory should be managed carefully. Good practices are:

- Prefer memory from the stack with constant size (SPU stack size is small).
- Allocating from the stack with dynamic size by using `alloca()` is possible (even on SPU), but it can introduce bugs that can be hard to find.
- Allocate small objects in bigger chunks (flyweight design pattern).
- Avoid reallocations (for example, reserve and stick to maximum budgets).
- Avoid allocations during the frame (sometimes simple parameter passing can cause allocations).
- Ensure that after processing one level the memory is not fragmented more than necessary (test case: loading multiple levels one after another).

A 64-bit address space is a good solution for the problem. This requires a 64-bit OS and running the 64-bit version of the application. Running a 32-bit application on a 64-bit OS helps very little. Note that compiling for 64-bit can result in a bigger executable file size, which can in some cases be counterproductive.

Bandwidth

To reduce memory bandwidth usage, make use of caches, use a local memory access pattern, keep the right data nearby, or use smaller data structures. Another option is to avoid memory accesses all together by recomputing on demand instead of storing data and reading it later.

Latency

Different types of memory have different access performance characteristics. Careful planning of data storage location can help to improve performance. For example, blending animation for run animation needs to be accessible within a fraction of a frame, and must be accessible in memory. In contrast, cut-scene animations can be stored on disk. To overcome higher latencies, extra coding may be required. In some cases the benefit may not be worth the effort.

Alignment

Some CPUs require proper alignment for data access (for example, reading a float requires an address dividable by 4). Other CPUs perform slower when data is not aligned properly (misaligned data access). As caches operate on increasing sizes, there are benefits to aligning data to the new sizes. When new

features are created, these structure sizes must be taken into consideration. Otherwise, the feature might not perform well or not even work.

Virtual Memory

Most operating systems try to handle memory quite conservatively because they never know what memory requests will come next. Code or data that has not been used for a certain time can be paged out to the hard drive. In games, this paging can result in stalls that can occur randomly, so most consoles avoid swapping.

Streaming

Streaming enables a game to simulate a world that is larger than limited available memory would normally allow. A secondary (usually slower) storage medium is required, and the limited resource is used as a cache. This is possible because the set of assets tends to change slowly and only part of the content is required at any given time. The set of assets kept in memory must adhere to the limits of the hardware available. While memory usage can partly be determined by code, designer decisions regarding the placement, use, and reuse of assets, and the use of occlusion and streaming hints are also important in determining the amout of memory required. Latency of streaming can be an issue when large changes to the set of required assets are necessary. Seek times are faster on hard drives than on most other storage media like DVDs, Blue-Rays or CDs. Sorting assets and keeping redundant copies of assets can help to improve performance.

Split screen or general multi-camera support add further challenges for the streaming system. Tracking the required asset set becomes more difficult under these circumstances. Seek performance can get get worse as multiple sets now need to be supported by the same hardware. It is wise to limit gameplay so that the streaming system can perform well. A streaming system works best if it knows about the assets that will be needed beforehand. Game code that loads assets on demand without registering them first will not be capable of doing this. It is better to wrap all asset access with a handle and allow registration and creation of handles only during some startup phase. This makes it easier to create stripped down builds (minimal builds consisting only of required assets).

Streaming System

The Lumberyard streaming engine takes care of the streaming of meshes, textures, music, sounds, and animations.

Low-level Streaming System

CryCommon interfaces and structs

The file `IStreamEngine.h` in CryCommon contains all the important interfaces and structs used by the rest of the engine.

First of all there is the `IStreamEngine` itself. There is only one `IStreamingEngine` in the application and it controls all the possible I/O streams. Most of the following information comes directly from the documentation inside the code, so it's always good to read the actual code in `IStreamEngine.h` file for any missing information.

The most important function in `IStreamEngine` is the `StartRead` function which is used to start any streaming request.

IStreamEngine.h

```
UNIQUE_IFACE struct IStreamEngine
{
public:

    // Description:
    //   Starts asynchronous read from the specified file (the file may be on a
    //   virtual file system, in pak or zip file or wherever).
    //   Reads the file contents into the given buffer, up to the given size.
    //   Upon success, calls success callback. If the file is truncated or for other
    //   reason can not be read, calls error callback. The callback can be NULL
    //   (in this case, the client should poll the returned IReadStream object;
    //   the returned object must be locked for that)
    // NOTE: the error/success/progress callbacks can also be called from INSIDE
    //   this function
    // Return Value:
    //   IReadStream is reference-counted and will be automatically deleted if
    //   you don't refer to it; if you don't store it immediately in an auto-pointer,
    //   it may be deleted as soon as on the next line of code,
    //   because the read operation may complete immediately inside StartRead()
    //   and the object is self-disposed as soon as the callback is called.
    virtual IReadStreamPtr StartRead (const EStreamTaskType tSource, const char* szFile,
 IStreamCallback* pCallback = NULL, StreamReadParams* pParams = NULL) = 0;

};
```

The following are the currently supported streaming task types. This enum should be extended if you want to stream a new object type.

IStreamEngine.h

```
enum EStreamTaskType
{
    eStreamTaskTypeCount      = 13,
    eStreamTaskTypePak        = 12, // Pak file itself
    eStreamTaskTypeFlash      = 11, // Flash file object
    eStreamTaskTypeVideo      = 10, // Video data (when streamed)
    eStreamTaskTypeReadAhead  = 9,  // Read ahead data used for file reading prediction
    eStreamTaskTypeShader     = 8,  // Shader combination data
    eStreamTaskTypeSound      = 7,
    eStreamTaskTypeMusic      = 6,
    eStreamTaskTypeFSBCache   = 5,  // Complete FSB file
    eStreamTaskTypeAnimation  = 4,  // All the possible animations types (dba, caf, ..)
    eStreamTaskTypeTerrain    = 3,  // Partial terrain data
    eStreamTaskTypeGeometry   = 2,  // Mesh or mesh lods
    eStreamTaskTypeTexture    = 1,  // Texture mip maps (currently mip0 is not streamed)
};
```

A callback object can be provided to the `StartStream` function to be informed when the streaming request has finished. The callback object should implement the following `StreamAsyncOnComplete` and `StreamOnComplete` functions.

IStreamEngine.h

```
class IStreamCallback
{
public:
    // Description:
    //   Signals that reading the requested data has completed (with or without error).
    //   This callback is always called, whether an error occurs or not, and is called
    //   from the async callback thread of the streaming engine, which happens
    //   directly after the reading operation
    virtual void StreamAsyncOnComplete (IReadStream* pStream, unsigned nError) {}
```

```
    // Description:
    //   Same as the StreamAsyncOnComplete, but this function is called from the main
    //   thread and is always called after the StreamAsyncOnComplete function.
    virtual void StreamOnComplete (IReadStream* pStream, unsigned nError) = 0;
};
```

When starting a read request, you can also provide the optional parameters shown in the following code.

IStreamEngine.h

```
struct StreamReadParams
{
public:

    // The user data that'll be used to call the callback.
    DWORD_PTR dwUserData;

    // The priority of this read
    EStreamTaskPriority ePriority;

    // Description:
    //   The buffer into which to read the file or the file piece
    //   if this is NULL, the streaming engine will supply the buffer.
    // Notes:
    //   DO NOT USE THIS BUFFER during read operation! DO NOT READ from it, it can lead to
 memory corruption!
    void* pBuffer;

    // Description:
    //   Offset in the file to read; if this is not 0, then the file read
    //   occurs beginning with the specified offset in bytes.
    //   The callback interface receives the size of already read data as nSize
    //   and generally behaves as if the piece of file would be a file of its own.
    unsigned nOffset;

    // Description:
    //   Number of bytes to read; if this is 0, then the whole file is read,
    //   if nSize == 0 && nOffset != 0, then the file from the offset to the end is read.
    //   If nSize != 0, then the file piece from nOffset is read, at most nSize bytes
    //   (if less, an error is reported). So, from nOffset byte to nOffset + nSize - 1 byte
 in the file.
    unsigned nSize;

    // Description:
    //   The combination of one or several flags from the stream engine general purpose
 flags.
    // See also:
    //   IStreamEngine::EFlags
    unsigned nFlags;
};
```

The return value of the `StartRead` function is an `IReadStream` object which can be optionally stored on the client. The `IReadStream` object is refcounted internally. When the callback object can be destroyed before the reading operation is finished, the readstream should be stored separately, and the abort should be called on it. Doing this will clean up the entire read request internally and will also call the async and sync callback functions.

The `Wait` function can be used to perform a blocking reading requests on the streaming engine. This function can be used from an async reading thread that uses the Lumberyard streaming system to perform the actual reading.

IStreamEngine.h

```
class IReadStream : public CMultiThreadRefCount
{
public:
    virtual void Abort() = 0;
    virtual void Wait( int nMaxWaitMillis=-1 ) = 0;
};
```

Internal flow of a read request

The Lumberyard stream engine uses extra worker and IO threads internally. For every possible IO input, a different `StreamingIOThread` is created which can run independently from the others.

Currently the stream engine has the following IO threads:

- Optical – Streaming from the optical data drive.
- Hard disk drive (HDD) – Streaming from installed data on the hard disk drive (this could be a fully installed game, or shadow copied data).
- Memory – Streaming from packed in-memory files, which requires very little IO.

When a reading request is made on the streaming engine, it first checks which IO thread to use, and computes the sortkey. The request is then inserted into one of the `StreamingIOThread` objects.

After the reading operation is finished, the request is forwarded to one of the decompression threads if the data was compressed, and then into one of the async callback threads. The amount of async callback threads is dependent on the platform, and some async callback threads are reserved for specific streaming request types such as geometry and textures. After the async callback has been processed, the finished streaming request is added to the streaming engine to be processed on the main thread. The next update on the streaming engine from the main thread will then call the sync callback (`StreamOnComplete`) and clean up the temporary allocated memory if needed.

For information regarding the IO/WorkerThreads please check the `StreamingIOThread` and `StreamingWorkerThread` class.

Read request sorting

Requests to the streaming engine are **not** processed in a the same order as which they have been requested. The system tries to internally 'optimize' the order in which to read the data, to maximize the read bandwidth.

When reading data from an optical disc , it is very important to reduce the amount of seeks. (This is also true when reading from a hard disk drive, but to a lesser extent). A single seek can take over 100 milliseconds, while the actual read time might take only a few milliseconds. Some official statistics from the 360 XDK follow.

- Outer diameter throughput : 12x (approximately 15 MB per second).
- Inner diameter throughput : 5x (6.8 MB per second).
- Average seek (1/3rd stroke) time : 110 ms typical, 140 ms maximum.
- Full stroke seek time : 180 ms typical, 240 ms maximum.
- Layer switch time : 75 ms.

The internal sorting algorithm takes the following rules into account in the following order.

- **Priority of the request** – High priority requests always take precedence, but too many of them can introduce too many extra seeks.

- **Time grouping** – Requests made within a certain time are grouped together to create a continuous reading operation on the disc for every time group. The default value is 2 seconds, but can be changed using the following cvar: `sys_streaming_requests_grouping_time_period`. Time grouping has a huge impact on the average completion time of the requests. It increases the time of a few otherwise quick reading requests, but drastically reduces the overall completion time because most of the streaming requests are coming from random places on the disc.
- **Actual offset on disc** – The actual disc offset is computed and used during the sorting. Files which have a higher offset get a higher priority, so it is important to organize the layout of the disc to reflect the desired streaming order.

For information regarding sorting, please refer to the source code in `StreamAsyncFileRequest::ComputeSortKey()`. The essential sorting code follows.

CAsyncIOFileRequest::ComputeSortKey

```
void CAsyncIOFileRequest::ComputeSortKey(uint64 nCurrentKeyInProgress)
{
    .. compute the disc offset (can be requested using CryPak)

    // group items by priority, then by snapped request time, then sort by disk offset
    m_nDiskOffset += m_nRequestedOffset;
    m_nTimeGroup = (uint64)(gEnv->pTimer->GetCurrTime() / max(1,
 g_cvars.sys_streaming_requests_grouping_time_period));
    uint64 nPrioriry = m_ePriority;

    int64 nDiskOffsetKB = m_nDiskOffset >> 10; // KB
    m_nSortKey = (nDiskOffsetKB) | (((uint64)m_nTimeGroup) << 30) | (nPrioriry << 60);
}
```

Streaming statistics

The streaming engine can be polled for streaming statistics using the `GetStreamingStatistics()` function.

Most of the statistics are divided into two groups, one collected during the last second, and another from the last reset (which usually happens during level loading). Statistics can also be forcibly reset during the game.

The `SMediaTypeInfo` struct gives information per IO input system (hard disk drive, optical, memory).

IStreamEngine.h

```
struct SMediaTypeInfo
{
    // stats collected during the last second
    float fActiveDuringLastSecond;
    float fAverageActiveTime;
    uint32 nBytesRead;
    uint32 nRequestCount;
    uint64 nSeekOffsetLastSecond;
    uint32 nCurrentReadBandwidth;
    uint32 nActualReadBandwidth;    // only taking actual reading time into account

    // stats collected since last reset
    uint64 nTotalBytesRead;
    uint32 nTotalRequestCount;
    uint64 nAverageSeekOffset;
    uint32 nSessionReadBandwidth;
    uint32 nAverageActualReadBandwidth; // only taking actual read time into account
```

```
};
```

The `SRequestTypeInfo` struct gives information about each streaming request type, such as geometry, textures, and animations.

IStreamEngine.h

```
struct SRequestTypeInfo
{
    int nOpenRequestCount;
    int nPendingReadBytes;

    // stats collected during the last second
    uint32 nCurrentReadBandwidth;

    // stats collected since last reset
    uint32 nTotalStreamingRequestCount;
    uint64 nTotalReadBytes;     // compressed data
    uint64 nTotalRequestDataSize;   // uncompressed data
    uint32 nTotalRequestCount;
    uint32 nSessionReadBandwidth;

    float fAverageCompletionTime;   // Average time it takes to fully complete a request
    float fAverageRequestCount; // Average amount of requests made per second
};
```

The following example shows global statistics that contain all the gathered data.

IStreamEngine.h

```
struct SStatistics
{
    SMediaTypeInfo hddInfo;
    SMediaTypeInfo memoryInfo;
    SMediaTypeInfo opticalInfo;

    SRequestTypeInfo typeInfo[eStreamTaskTypeCount];

    uint32 nTotalSessionReadBandwidth;  // Average read bandwidth in total from reset -
 taking full time into account from reset
    uint32 nTotalCurrentReadBandwidth;  // Total bytes/sec over all types and systems.

    int nPendingReadBytes;      // How many bytes still need to be read
    float fAverageCompletionTime;   // Time in seconds on average takes to complete read
 request.
    float fAverageRequestCount; // Average requests per second being done to streaming
 engine
    int nOpenRequestCount;      // Amount of open requests

    uint64 nTotalBytesRead;     // Read bytes total from reset.
    uint32 nTotalRequestCount;  // Number of request from reset to the streaming engine.

    uint32 nDecompressBandwidth;    // Bytes/second for last second

    int nMaxTempMemory;     // Maximum temporary memory used by the streaming system
};
```

Streaming debug information

Different types of debug information can be requested using the following CVar: `sys_streaming_debug x`.

Streaming and Levelcache Pak Files

As mentioned earlier, it is very important to minimize the seeks and seek distances when reading from an optical media drive. For this reason, the build system is designed to optimize the internal data layout for streaming.

The easiest and fastest approach is to not do any IO at all, but read the data from compressed data in memory. For this, small paks for startup and each level are created. These are loaded into memory during level loading. Some paks remain in memory until the end of the level. Others are only used to speed up the level loading. All small files and small read requests should ideally be diverted to these paks.

A special `RC_Job` build file is used to generate these paks: `Bin32/rc/RCJob_PerLevelCache.xml`. These paks are generated during a normal build pipeline. The internal managment in the engine is done by the CResourceManager class, which uses the global `SystemEvents` to preload or unload the paks.

Currently, the following paks are loaded into memory during level loading (`sys_PakLoadCache`).

- **level.pak** – Contains all actual level data, and should not be touched after level loading anymore.
- **xml.pak**
- **dds0.pak** – Contains all lowest mips of all the textures in the level.
- **cgf.pak** and **cga.pak** – Only load when CGF streaming is enabled.

The following paks are cached into memory during the level load process (`sys_PakStreamCache`).

- **dds_cache.pak** - Contains all dds files smaller than 6 KB (except for dds.0 files).
- **cgf_cache.pak** - Contains all cgf files smaller than 32 KB (only when CGF streaming is enabled).

> **Important**
> Be sure that these paks are available. Without them, level loading can take up to a few minutes, and streaming performance is greatly reduced.

The information regarding all the resources of a level are stored in the `resourcelist.txt` and `auto_resourcelist.txt`. These files are generated by an automatic testing system which loads each level and executes a prerecorded playthrough on it. These `resourcelist` files are used during the build phase to generate the level paks.

All data not in these in memory paks is handled through IO on the optical drive or hard disk drive, and it is also best to reduce the amount of seeks here. This optimization phase is also performed during the build process using the resource compiler.

All the data which can be streamed is extracted from all the resource lists from all levels, and is removed from the default pak files (for example, `objects.pak`, `textures.pak`, `animations.pak`) and put into new optimized paks for streaming inside a streaming folder.

The creating of the streaming paks uses the following rules:

- Split by **extension**: Different extension files are put into different paks (for example, dds, caf, dba, cgf) so that files of the same type can be put close to each other. This enables them to be read in bursts. The paks are also used to increase the priority of certain file types during request sorting by using the disc offset.
- Split by DDS **type**: Different dds types are sorted differently to increase the priority of different types (for example, diffuse maps get higher priority than normal maps). The actual distance in the pak is used during the sorting of the request.

- Split by DDS **mip**: The highest mips are put into a separate pak file. They usually take more than 60% of the size of all the smaller mips and can then be streamed with a lower priority. This greatly reduces the average seek time required to read the smaller textures. The texture streaming system internally optimizes the reads to reflect these split texture data.
- Sort **alphabetically**: Default alphabetical sorting is required because some of the data (such as CGF's during MP level loading), are loaded in alphabetical order. Changing this sort order can have a severe impact on the loading times.

The actual sorting code is hardcoded in the resource compiler, and can be found at: `Code\Tools\RC\ResourceCompiler\PakHelpers.cpp`.

> **Important**
> If you make changes to the sorting operator in the resource compiler, be sure to make the same changes to the texture streaming and streaming engine sorting operators.

Single Thread IO Access and Invalid File Access

It is very important that only a single thread access a particular IO device at one time. If multiple threads read from the same IO device concurrently, then the reading speed is more than halved, and it may take a number of seconds to read just a few kilobytes. This occurs because the IO reading head will partially read a few kilobytes for one thread, and then read another few kilobytes for another thread while always performing expensive seeks in between.

The solution is to exclusively read from `StreamingIOThreads` during gameplay. Lumberyard will by default show an **Invalid File Access** warning in the top left corner when reading data from the wrong thread, and will stall deliberately for threed seconds to emulate the actual stall when reading from an optical drive.

High Level Streaming Engine Usage

It is very easy to extend the current streaming functionality using the streaming engine. In this section, a small example class is presented that shows how to add a new streaming type.

First, create a class which derives from the `IStreamCallback` interface, which informs about streaming completion, and add some basic functionality to read a file. The file can either be read directly or use the streaming engine. When the data is read directly, it calls the `ProcessData` function to parse the loaded data. The function is also called from the `async` callback. Some processing can be performed here on the data if needed because it does not run on the main thread.

The default parameters are used when starting a reading request on the streaming engine. It is also possible to specify the final data storage to help reduce the number of dynamic allocations performed by the streaming engine.

The class also stores the read stream object in order to get information about the streaming request or to be able to cancel the request when the callback object is destroyed. The pointer is reset in the sync callback because after the call it will no longer be referenced by the streaming engine.

CNewStreamingType

```
#include
class CNewStreamingType : public IStreamCallback
{
public:
    CNewStreamingType() : m_pReadStream(0), m_bIsLoaded(false) {}
    ~CNewStreamingType()
    {
```

```
        if (m_pReadStream)
            m_pReadStream->Abort();
    }

    // Start reading some data
    bool ReadFile(const char* acFilename, bool bUseStreamingEngine)
    {
        if (bUseStreamingEngine)
        {
            StreamReadParams params;
            params.dwUserData = eLoadFullData;
            params.ePriority = estpNormal;
            params.nSize = 0; // read the full file
            params.pBuffer = NULL; // don't provide any buffer, but copy data when
 streaming is done

            m_pReadStream = g_pISystem->GetStreamEngine()-
>StartRead(eStreamTaskTypeNewType, acFilename, this, &params);
        }
        else
        {
            // old way of reading file in a sync way (blocking call!)
            const char* acData = 0;
            size_t stSize = 0;

            .. read file directly using CryPak or fopen/fread

            ProcessData(acData, stSize);
            m_bIsLoaded = true;
        }

        return m_bIsLoaded;
    }

    // Check if the data is ready and loaded
    bool IsLoaded() const { return m_bIsLoaded; }

protected:

    // implement the IStreamCallback function
    void StreamAsyncOnComplete(IReadStream* pStream, unsigned nError)
    {
        if(nError)
        {
            return;
        }

        const char* acData = (char*)pStream->GetBuffer();
        size_t stSize= pStream->GetBytesRead();

        ProcessData(acData, stSize);
        m_bIsLoaded = true;
    }

    void StreamOnComplete (IReadStream* pStream, unsigned nError)
    {
        m_pReadStream = 0;
    }

    // process the actual loaded data
    void ProcessData(const char* acData, size_t stSize);

    // store the stream callback object to be sure it can be canceled when the object is
 destroyed
    IReadStreamPtr m_pReadStream;
    // Extra flag used to check if the data is ready
```

```
    bool m_bIsLoaded;
}
```

Text Localization and Unicode Support

Because games are typically localized to various languages, your game might have to use text data for many languages.

This document provides programming-related information regarding localization, including localization information specific to Lumberyard.

Terminology

The following table provides brief descriptions of some important terms related to localization and text processing.

Term	Description
character	A unit of textual data. A character can be a glyph or formatting indicator. Note that a glyph does not necessarily form a single visible unit. For example, a diacritical mark [´] and the letter [a] are separate glyphs (and characters), but can be overlaid to form the character [á].
Unicode	A standard maintained by the Unicode Consortium that deals with text and language standardization.
UCS	Universal Character Set, the standardized set of characters in the Unicode standard (also, ISO-10646)
(UCS) code-point	An integral identifier for a single character in the UCS defined range, typically displayed with the U prefix followed by hexadecimal, for example: U+12AB
(text) encoding	A method of mapping (a subset of) UCS to a sequence of code-units, or the process of applying an encoding.
code-unit	An encoding-specific unit integral identifier used to encode code-points. Many code-units may be used to represent a single code-point.
ASCII	A standardized encoding that covers the first 128 code-points of the UCS space using 7- or 8-bit code-units.
(ANSI) code-page	A standardized encoding that extends ASCII by assigning additional meaning to the higher 128 values when using 8-bit code-units There are many hundreds of code-pages, some of which use multi-byte sequences to encode code-points.
UTF	UCS Transformation Format, a standardized encoding that covers the entire UCS space.
UTF-8	A specific instance of UTF, using 8-bit code-units. Each code-point can take 1 to 4 (inclusive) code-units.
UTF-16	A specific instance of UTF, using 16-bit code-units. Each code-point can take 1 or 2 code-units.
UTF-32	A specific instance of UTF, using 32-bit code-units. Each code-point is directly mapped to a single code-unit.

Term	Description
byte-order	How a CPU treats a sequence of bytes when interpreting multi-byte values. A byte-orderTypically either little-endian or big-endian format
encoding error	A sequence of code-units that does not form a code-point (or an invalid code-point, as defined by the Unicode standard)

What encoding to use?

Since there are many methods of encoding text, the question that should be asked when dealing with even the smallest amount of text is, "In what encoding is this stored?" This is an important question because decoding a sequence of code-units in the wrong way will lead to encoding errors, or even worse, to valid decoding that yields the wrong content.

The following table describes some common encodings.

Encoding	Code-unit size	Code-point size	Maps the entire UCS space	Trivial to encode/ decode	Immune to byte-order differences	Major users
ASCII	7 bits	1 byte	no	yes	yes	Many English-only apps
(ANSI) code-page	8 bits	varies, usually 1 byte	no	varies, usually yes	yes	Older OS functions
UTF-8	8 bits	1 to 4 bytes	yes	no	yes	Most text on the internet, XML
UTF-16	16 bits	2 to 4 bytes	yes	yes	no	Windows "wide" API, Qt
UCS-2	16 bits	2 bytes	no	yes	no	None (replaced with UTF-16)
UTF-32 UCS-4	32 bits	4 bytes	yes	yes	no	Linux "wide" API

Because there is no single "best" encoding, you should always consider the scenario in which it will be used when choosing one.

Historically, different operating systems and software packages have chosen different sets of supported encodings. Even C++ follows different conventions on different platforms. For example, the "wide character" `wchar_t` is 16-bits on Windows, but 32-bits on Linux.

Because Lumberyard products can be used on many platforms and in many languages, full UCS coverage is desirable. The follow table presents some conventions used in Lumberyard:

Text data type	Encoding	Reason
Source code	ASCII	We write our code in English, which means ASCII is sufficient.
Text assets	UTF-8	Assets can be transferred between machines with potentially differing byte-order, and may contain text in many languages.
Run-time variables	UTF-8	Since transforming text data from or to UTF-8 is not free, we keep data in UTF-8 as much as possible. Exceptions must be made when interacting with libraries or operating systems that require another encoding. In these cases all transformations should be done at the call-site.
File and path names	ASCII	File names are a special case with regards to case-sensitivity, as defined by the file system. Unicode defines 3 cases, and conversions between them are locale-specific. In addition, the normalization formats are typically not (all) accounted for in file-systems and their APIs. Some specialized file-systems only accept ASCII. This combination means that using the most basic and portable sub-set should be preferred, with UTF-8 being used only as required.

General principles

- Avoid using non-ASCII characters in source code. Consider using escape sequences if a non-ASCII literal is required.
- Avoid using absolute paths. Only paths that are under developer control should be entered. If possible, use relative ASCII paths for the game folder, root folder, and user folder. When this is not possible, carefully consider non-ASCII contents that may be under a user's control, such as those in the installation folder.

How does this affect me when writing code?

Since single-byte code-units are common (even in languages that also use double-byte code-units), single-byte string types can be used almost universally. In addition, since Lumberyard does not use ANSI code-pages, all text must be either ASCII or UTF-8.

The following properties hold for both ASCII and UTF-8.

- The NULL-byte (integral value 0) only occurs when a NULL-byte is intended (UTF-8 never generates a NULL-byte as part of multi-byte sequences). This means that C-style null-terminated strings act the same, and CRT functions like `strlen` will work as expected, except that it counts code-units, not characters.
- Code-points in the ASCII range have the same encoded value in UTF-8. This means that you can type English string literals in code and treat them as UTF-8 without conversion. Also, you can compare characters in the ASCII range directly against UTF-8 content (that is, when looking for an English or ASCII symbol sub-string).
- UTF-8 sequences (containing zero or more entire code-points) do not carry context. This means they are safe to append to each other without changing the contents of the text.

The difference between position and length in code-units (as reported through `string::length()`, `strlen()`, and similar functions) and their matching position and length in code-points is largely irrelevant. This is because the meaning of the sequence is typically abstract, and the meaning of the bytes matters only when the text is interpreted or displayed. However, keep in mind the following caveats.

- **Splitting strings** – When splitting a string, it's important to do one of the following.
  1. Recombine the parts in the same order after splitting, without interpreting the splitted parts as text (that is, without chunking for transmission).
  2. Perform the split at a boundary between code-points. The positions just before and just after any ASCII character are always safe.
- **API boundaries** – When an API accepts or returns strings, it's s important to know what encoding the API uses. If the API doesn't treat strings as opaque (that is, interprets the text), passing UTF-8 may be problematic for APIs that accept byte-strings and interpret them as ASCII or ANSI. If no UTF-8 API is available, prefer any other Unicode API instead (UTF-16 or UTF-32). As a last resort, convert to ASCII, but understand that the conversion is lossy and cannot be recovered from the converted string. Always read the documentation of the API to see what text encoding it expects and perform any required conversion. All UTF encodings can be losslessly converted in both directions, so finding any API that accepts a UTF format gives you a way to use UTF encoding.
- **Identifiers** – When using strings as a "key" in a collection or for comparison, avoid using non-ASCII sequences as keys, as the concept of "equality" of UTF is complex due to normalization forms and locale-dependent rules. However, comparing UTF-8 strings byte-by-byte is safe if you only care about equality in terms of code-points (since code-point to code-unit mapping is 1:1).
- **Sorting** – When using strings for sorting, keep in mind that locale-specific rules for the order of text are complex. It's fine to let the UI deal with this in many cases. In general, make no assumptions of how a set of strings will be sorted. However, sorting UTF-8 strings as if they were ASCII will actually sort them by code-point. This is fine if you only require an arbitrary fixed order for `std::map` look-up, but displaying contents in the UI in this order may be confusing for end-users that expect another ordering.

In general, avoid interpreting text if at all possible. Otherwise, try to operate on the ASCII subset and treat all other text parts as opaque indivisible sequences. When dealing with the concept of "length" or "size", try to consider using in code-units instead of code-points, since those operations are computationally cheaper. In fact, the concept of the "length" of Unicode sequences is complex, and there is a many-to-many mapping between code-points and what is actually displayed.

How does this affect me when dealing with text assets?

In general, always:

- Store text assets with UTF-8 encoding.
- Store with Unicode NFC (Normalization Form C). This is the most common form of storage in text editing tools, so it's best to use this form unless you have a good reason to do otherwise.
- Store text in the correct case (that is, the one that will be displayed). Case-conversion is a complex topic in many languages and is best avoided.

Utilities provided in CryCommon

Lumberyard provides some utilities to make it easy to losslessly and safely convert text between Unicode encodings. In-depth technical details are provided in the header files that expose the `UnicodeFunctions.h` and `UnicodeIterator.h` utilities.

The most common use cases are as follows.

```
string utf8;
wstring wide;
Unicode::Convert(utf8, wide); // Convert contents of wide string and store into UTF-8
 string
```

```
Unicode::Convert(wide, utf8); // Convert contents of UTF-8 string to wide string
```

```
string ascii;
Unicode::Convert<Unicode::eEncoding_ASCII, Unicode::eEncoding_UTF8>(ascii, utf8); //
 Convert UTF-8 to ASCII (lossy!)
```

Important
The above functions assume that the input text is already validly encoded. To guard against malformed user input or potentially broken input, consider using the `Unicode::ConvertSafe` function.

Further reading

For an introduction to Unicode, see The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!).

For official information about Unicode, see The Unicode Consortium.

CryLog

CryLog Logging Functionality

You can log in Lumberyard by using the following global functions.

- `CryLog (eMessage)`
- `CryLogAlways (eAlways)`
- `CryError (eError)`
- `CryWarning (eWarning)`
- `CryComment (eComment)`

If more control is required, the `ILog` interface can be used directly by using the following syntax.

```
gEnv->pLog->LogToFile("value %d",iVal);
```

Verbosity Level and Coloring

You can control the verbosity of logging with the console variables `log_Verbosity` and `log_FileVerbosity`.

The following table shows the levels of verbosity and color convention. In the console, warnings appear in yellow, and errors appear in red.

Message	verbosity 0	verbosity 1	verbosity 2	verbosity 3	verbosity 4	Color in console
eAlways	X	X	X	X	X	
eErrorAlways	X	X	X	X	X	red
eWarningAlways	X	X	X	X	X	yellow

Message	verbosity 0	verbosity 1	verbosity 2	verbosity 3	verbosity 4	Color in console
eInput	?	?	?	?	?	
eInputResponse	?	?	?	?	?	
eError		X	X	X	X	red
eWarning	-	-	X	X	X	yellow
eMessage	-	-		X	X	
eComment	-	-	-	-	X	

Key

- X – the message type is logged to the console or file
- ? – some special logic is involved

> **Tip**
> Full logging (to console and file) can be enabled by using `log_Verbosity 4`.

Log Files

The following log file sources write to the log files indicated.

Source	Log file
Lumberyard Editor	`Editor.log`
Game	`game.log` (default)
Error messages	`Error.log`

Console Variables

The following console variables relate to logging.

`log_IncludeTime`

```
Toggles time stamping of log entries.
Usage: log_IncludeTime [0/1/2/3/4/5]
  0=off (default)
  1=current time
  2=relative time
  3=current+relative time
  4=absolute time in seconds since this mode was started
  5=current time+server time
```

`ai_LogFileVerbosity`

```
None = 0, progress/errors/warnings = 1, event = 2, comment = 3
```

```
log_Verbosity DUMPTODISK
```

```
defines the verbosity level for log messages written to console
-1=suppress all logs (including eAlways)
0=suppress all logs(except eAlways)
1=additional errors
2=additional warnings
3=additional messages
4=additional comments
```

CryConsole

The console is a user interface system which handles console commands and console variables. It also outputs log messages and stores the input and output history.

Color coding

The game console supports color coding by using the color indices 0..9 with a leading $ character. The code is hidden in the text outputted on the console. Simple log messages through the `ILog` interface can be used to send text to the console.

```
This is normal $1one$2two$3three and so on
```

In the preceding example, one renders in red, two in green, and three (and the remaining text) in blue.

Dumping all console commands and variables

All console commands and console variables can be logged to a file by using the command `DumpCommandsVars`. The default filename is `consolecommandsandvars.txt`.

To restrict the variables that should be dumped, a sub-string parameter can be passed. For example, the command

```
DumpCommandsVars i_
```

logs all commands and variables that begin with the sub-string "i_". (for example, `i_giveallitems` and `i_debug_projectiles`).

Console Variables

Console variables provide a convenient way to expose variables which can be modified easily by the user either by being entered in the console during runtime or by passing it as command-line argument before launching the application.

More information on how to use command-line arguments can be found in the Command Line Arguments article.

Console variables are commonly referred to as `CVar` in the code base.

Registering new console variables

For an integer or float based console variable, it is recommended to use the `IConsole::Register()` function to expose a C++ variable as a console variable.

To declare a new string console variable, use the `IConsole::RegisterString()` function.

Accessing console variables from C++

Console variables are exposed using the `ICVar` interface. To retrieve this interface, use the `IConsole::GetCVar()` function.

The most efficient way to read the console variable value is to access directly the C++ variable bound to the console variable proxy.

Adding New Console Commands

The console can easily be extended with new console commands. A new console command can be implemented in C++ as a static function which follows the `ConsoleCommandFunc` type. Arguments for this console command are passed using the `IConsoleCmdArgs` interface.

The following code shows the skeleton implementation of a console command:

```
static void RequestLoadMod(IConsoleCmdArgs* pCmdArgs);

void RequestLoadMod(IConsoleCmdArgs* pCmdArgs)
{
  if (pCmdArgs->GetArgCount() == 2)
  {
  const char* pName = pCmdArgs->GetArg(1);
  // ...
  }
  else
  {
  CryLog("Error, correct syntax is: g_loadMod modname");
  }
}
```

The following code will register the command with the console system:

```
IConsole* pConsole = gEnv->pSystem->GetIConsole();
pConsole->AddCommand("g_loadMod", RequestLoadMod);
```

Console Variable Groups

Console variable groups provide a convenient way to apply predefined settings to multiple console variables at once.

Console variables are commonly referred to as `CVarGroup` in the code base. Console variable groups can modify other console variables to build bigger hierarchies.

Warning
Cycles in the assignments are not detected and can cause crashes.

Registering a new variable group

To register a new variable group, add a new `.cfg` text file to the `GameSDK\config\CVarGroups` directory.

`sys_spec_Particles.cfg`

```
[default]
; default of this CVarGroup
= 4
e_particles_lod=1
```

```
e_particles_max_emitter_draw_screen=64

[1]
e_particles_lod=0.75
e_particles_max_emitter_draw_screen=1

[2]
e_particles_max_emitter_draw_screen=4

[3]
e_particles_max_emitter_draw_screen=16
```

This creates a new console variable group named `sys_spec_Particles` that behaves like an integer console variable. By default, this variable has the state `4` (set in the line following the comment in the example).

On changing the variable, the new state is applied. Console variables not specified in the `.cfg` file are not set. All console variables need to be part of the default section. An error message is output in case of violation of this rule.

If a console variable is not specified in a custom section, the value specified in the default section is applied.

Console variable group documentation

The documentation of the console variable group is generated automatically.

```
sys_spec_Particles
```

```
Console variable group to apply settings to multiple variables

sys_spec_Particles [1/2/3/4/x]:
 ... e_particles_lod = 0.75/1/1/1/1
 ... e_particles_max_screen_fill = 16/32/64/128/128
 ... e_particles_object_collisions = 0/1/1/1/1
 ... e_particles_quality = 1/2/3/4/4
 ... e_water_ocean_soft_particles = 0/1/1/1/1
 ... r_UseSoftParticles = 0/1/1/1/1
```

Checking if a console variable group value represents the state of the variables it controls

From the console

In the console you can type in the console variable group name and press tab. If the variable value is not represented, it will print the value of `RealState`.

```
sys_spec_Particles=2 [REQUIRE_NET_SYNC] RealState=3
sys_spec_Sound=1 [REQUIRE_NET_SYNC] RealState=CUSTOM
sys_spec_Texture=1 [REQUIRE_NET_SYNC]
```

By calling the console command `sys_RestoreSpec` you can check why the `sys_spec_` variables don't represent the right states.

From C++ code

From the code you can use the member function `GetRealIVal()` and compare its return value against the result of `GetIVal()` in `ICVar`.

Deferred execution of command line console commands

The commands that are passed via the command line by using the + prefix are stored in a separate list as opposed to the rest of the console commands.

This list allows the application to distribute the execution of those commands over several frames rather than executing everything at once.

Example

Consider the following example.

```
--- autotest.cfg --
hud_startPaused = "0"
wait_frames 100
screenshot autotestFrames
wait_seconds 5.0
screenshot autotestTime

-- console --
crysis.exe -devmode +map island +exec autotest +quit
```

In the example, the following operations were performed:

- Load the island map.
- Wait for 100 frames.
- Take a screenshot called autotestFrames.
- Wait for 5 seconds.
- Take a screenshot called autotestTime.
- Quit the application.

Details

Two categories of commands are defined: blocker and normal.

For each frame, the deferred command list is processed as a fifo. Elements of this list are consumed as long as normal commands are encountered.

When a blocker is consumed from the list and executed, the process is delayed until the next frame. For instance, commands like `map` and `screenshot` are blockers.

A console command (either command or variable) can be tagged as a blocker during its declaration using the `VF_BLOCKFRAME` flag.

The following synchronization commands are supported.

Optional Title

Command	Type	Description
`wait_frames` *num*:	<int>	Wait for *num* frames before the exeution of the list is resumed.

Command	Type	Description
`wait_seconds` *`sec`*:	<float>	Wait for *`sec`* seconds before the exeution of the list is resumed.

CVar Tutorial

This tutorial shows you how to modify existing and create console variables (CVars). CVars can be used to control many configurable behaviors in Lumberyard. You can also use them in your game.

Note
This brief tutorial is intended for programmers. Most of the content uses code.

Creating CVars

To create a console variable

1. In your code editor, open the `Code\GameSDK\GameDll\GameCVars.h` file, which declares all game-specific CVars.
2. Locate the `SCVars` struct. Inside the struct, declare a new variable, as in the following example.

```
struct SCVars
{
 int g_tutorialVar; //add this line

 //... pre-existing code ...
};
```

   The variable you added will be used to store the current value of the variable. If you need to store fractional numbers, you can also add a variable of the type `float`.

   Next, you will register the CVar with the game engine so that its value can be changed by using the console.
3. In the same `Code\GameSDK\GameDll\GameCVars.cpp` file, locate the `InitCVars` function.

```
void SCVars::InitCVars(IConsole *pConsole)
{
 m_releaseConstants.Init( pConsole );

 REGISTER_CVAR(g_tutorialVar, 42, VF_NULL, "This CVar was added using the tutorial on
 CVars"); //add this line

    //... pre-existing code ...
}
```

4. Specify a default value and help text for the variable. You can initialize the variable with any value that is valid for the type with which the variable was declared in the header file. The preceeding example specifies 42 as the default value and some help text that will be shown to users.
5. When your game unloads, be sure to un-register the variable. In the `Code\GameSDK\GameDll\GameCVars.cpp` file, locate and use the `ReleaseCVars` function, as shown in the following example.

```
void SCVars::ReleaseCVars()
{
 IConsole* pConsole = gEnv->pConsole;

 pConsole->UnregisterVariable("g_tutorialVar", true); //add this line
```

```
 //... pre-existing code ...
}
```

6. After you finish making changes, don't forget to compile your code.

Using the CVar

You can now change the value of the CVar that you created by using code, the console, and `.cfg` files.

From code

To access the value of the variable in your game code, use the `g_pGameCVars` pointer, as shown in the following example.

```
int myTutorialVar = g_pGameCVars->g_tutorialVar;
```

From the console

To change the value of the cvar from the console, use the syntax *`cvar_name`*=*`cvar_value`*. The following example sets the value of the `g_tutorialVar` console variable to 1337.

```
g_tutorialVar = 1337
```

From .cfg files

It's also possible to change the default CVar value from one of the `.cfg` files. Whenever a CVar is assigned a value, its previous value is discarded. Therefore, the last assignment is the one that is current.

The following list shows the order of initialization for console variables.

1. The value specified in the `GameCVars.cpp` file when `REGISTER_CVAR` is used. (A change here requires compiling.)
2. The value specified in the `system.cfg` file.
3. The value specified in the user's `user.cfg` file.
4. Any value assigned at game runtime.

> **Tip**
> To change the default value of an existing CVar without having to compile, add a line to `system.cfg` file to override the default.

Lumberyard Blog, Forums, and Feedback

As we continue to improve Lumberyard, we want to thank everyone in our developer community. Without your participation in the forums, your messages, and your bug reports, Lumberyard wouldn't be as strong as it is.

- Keep sending your feedback to `<lumberyard-feedback@amazon.com>`.
- If you haven't spoken up on the forums yet, we would love to have you.
- You can also keep up with new changes on our blog and leave comments to let us know what you think.