# Table of Contents

Lumberyard for Programmers .................................................................................................................. 1
AI .......................................................................................................................................................... 3

**AI System Concepts** .......................................................................................................................... 3
  - AI System Overview ......................................................................................................................... 4
  - Pathfinding Costs .............................................................................................................................. 7
  - Sensory Models .................................................................................................................................. 10
  - Flight ................................................................................................................................................ 14
  - AI C++ Class Hierarchy ...................................................................................................................... 15
  - 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 ................................................................................................................ 21
  - 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::Execute2D .................................................................. 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 ..................................................................................................................... 42
  - Refpoints .......................................................................................................................................... 86
  - Signals .............................................................................................................................................. 87

**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
Lumberyard Developer Guide

Step 3: Sign in as Your IAM User ................................................................. 146
Step 4: Enabling the Cloud Canvas Gem (extension) Package ...................... 147
Step 5: Add Administrator Credentials to Lumberyard ................................. 147
Step 6: Initializing Cloud Canvas from the Command Line ............................. 149
Step 7: Locating and Adding Resource Groups ............................................... 149
Step 8: Creating Deployments ....................................................................... 150
Step 9: Inspecting Your Resources in AWS ..................................................... 152
Step 10: Using IAM to Administer a Cloud Canvas Team ............................... 152
Step 11: Remove Cloud Canvas Functionality and AWS Resources ................ 154

Don't Die Sample Project ............................................................................. 154

Setup .............................................................................................................. 154
Viewing Lambda Code in Visual Studio ........................................................ 157
Deleting the AWS Project Stack ..................................................................... 159
AWS Services Used ........................................................................................ 159

Using Cloud Canvas ......................................................................................... 160
Cloud Canvas Tools in Lumberyard Editor .................................................... 160
Editing Cloud Canvas Files ........................................................................... 161
Initializing **Cloud Canvas Resource Manager** .......................................... 162
Managing Cloud Canvas Profiles .................................................................... 163
Understanding Resource Status Descriptions .............................................. 165
Using the Cloud Canvas Command Line ....................................................... 165
Viewing the Cloud Canvas Progress Log ....................................................... 179
Working with Deployments ........................................................................... 180
Working with JSON Files .............................................................................. 186
Working with Project Stacks ......................................................................... 187
Working with Resource Groups ..................................................................... 188

Cloud Canvas Flow Graph Node Reference .................................................. 193
Cloud Canvas Configuration Nodes ............................................................... 194
Cognito (Player Identity) Nodes ..................................................................... 197
DynamoDB (Database) Nodes ....................................................................... 198
Lambda (Cloud Functions) Node .................................................................... 205
S3 (Storage) Nodes ........................................................................................ 206
SNS (Notification Service) Nodes ................................................................. 208
S3 (Message Queuing Service) Nodes ........................................................... 211
Static Data (PROTOTYPE) Nodes ................................................................... 212

Resource Definitions ..................................................................................... 216
Resource Definition Location ........................................................................ 216
**project-settings.json** ............................................................................. 217
user-settings.json ........................................................................................ 219
project-template.json ................................................................................... 220
deployment-template.json .......................................................................... 225
deployment-access-template.json .............................................................. 227
The project-code Subdirectory ...................................................................... 234
resource-group/{resource-group} subdirectories ....................................... 234
resource-template.js.on ............................................................... 234
The lambda-function-code Subdirectory ...................................................... 238

Resource Deployments .................................................................................. 238
Configuration Bucket ................................................................................... 239

Resource Mappings ....................................................................................... 240
Using Mappings in AWS Flow Nodes ............................................................ 241
Using Mappings with the AWS C++ SDK ...................................................... 241
Using Mappings in Lambda Functions .......................................................... 242

Custom Resources ....................................................................................... 242
CognitoIdentityPool ....................................................................................... 242
EmptyDeployment ......................................................................................... 243
ResourceGroupConfiguration ........................................................................ 243
LambdaConfiguration .................................................................................... 244
PlayerAccess ................................................................................................. 245
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Bus (EBus)</td>
<td>343</td>
</tr>
<tr>
<td>Entity System</td>
<td>320</td>
</tr>
<tr>
<td>Demo and Video Capture</td>
<td>314</td>
</tr>
<tr>
<td>Capturing Video and Audio</td>
<td>314</td>
</tr>
<tr>
<td>Preparation</td>
<td>314</td>
</tr>
<tr>
<td>Video Settings</td>
<td>314</td>
</tr>
<tr>
<td>Starting and Ending the Video Recording</td>
<td>316</td>
</tr>
<tr>
<td>Audio Settings</td>
<td>316</td>
</tr>
<tr>
<td>Configuration Files</td>
<td>317</td>
</tr>
<tr>
<td>Recording Time Demos</td>
<td>318</td>
</tr>
<tr>
<td>Overview</td>
<td>318</td>
</tr>
<tr>
<td>Recording Controls</td>
<td>318</td>
</tr>
<tr>
<td>Related Console Variables</td>
<td>318</td>
</tr>
<tr>
<td>Entity Property Prefixes</td>
<td>320</td>
</tr>
<tr>
<td>Creating a New Entity Class</td>
<td>321</td>
</tr>
<tr>
<td>Entity Pool System</td>
<td>323</td>
</tr>
<tr>
<td>Editor Usage</td>
<td>324</td>
</tr>
<tr>
<td>Static versus Dynamic Entities</td>
<td>324</td>
</tr>
<tr>
<td>Entity Pool Definitions</td>
<td>324</td>
</tr>
<tr>
<td>Entity Pool Creation</td>
<td>326</td>
</tr>
<tr>
<td>Creating and Destroying Static Entities with Pools</td>
<td>327</td>
</tr>
<tr>
<td>Creating and Destroying Dynamic Entities with Pools</td>
<td>329</td>
</tr>
<tr>
<td>Serialization</td>
<td>330</td>
</tr>
<tr>
<td>Listener/Event Registration</td>
<td>331</td>
</tr>
<tr>
<td>Debugging Utilities</td>
<td>332</td>
</tr>
<tr>
<td>Entity ID Explained</td>
<td>332</td>
</tr>
<tr>
<td>Adding Usable Support on an Entity</td>
<td>333</td>
</tr>
<tr>
<td>Overview</td>
<td>333</td>
</tr>
<tr>
<td>Preparing the Script</td>
<td>333</td>
</tr>
<tr>
<td>Implementing IsUsable</td>
<td>334</td>
</tr>
<tr>
<td>Implementing OnUsed</td>
<td>334</td>
</tr>
<tr>
<td>Entity Scripting</td>
<td>334</td>
</tr>
<tr>
<td>Structure of a Script Entity</td>
<td>334</td>
</tr>
<tr>
<td>Using Entity State</td>
<td>337</td>
</tr>
<tr>
<td>Using Entity Slots</td>
<td>338</td>
</tr>
<tr>
<td>Linking Entities</td>
<td>339</td>
</tr>
<tr>
<td>Exposing an Entity to the Network</td>
<td>340</td>
</tr>
<tr>
<td>Event Bus (EBus)</td>
<td>343</td>
</tr>
<tr>
<td>Bus Configurations</td>
<td>343</td>
</tr>
<tr>
<td>Single Handler</td>
<td>343</td>
</tr>
<tr>
<td>Many Handlers</td>
<td>344</td>
</tr>
<tr>
<td>EBus with Addresses and a Single Handler</td>
<td>345</td>
</tr>
<tr>
<td>EBus with Addresses and Many Handlers</td>
<td>346</td>
</tr>
<tr>
<td>Synchronous vs. Asynchronous</td>
<td>348</td>
</tr>
<tr>
<td>Additional Features</td>
<td>348</td>
</tr>
<tr>
<td>Usage and Examples</td>
<td>349</td>
</tr>
<tr>
<td>Declaring an EBus</td>
<td>349</td>
</tr>
<tr>
<td>EBus Configuration Options</td>
<td>350</td>
</tr>
<tr>
<td>Implementing a Handler</td>
<td>351</td>
</tr>
<tr>
<td>Sending Messages to an EBus</td>
<td>352</td>
</tr>
<tr>
<td>Retrieving Return Values</td>
<td>352</td>
</tr>
<tr>
<td>Return Values from Multiple Handlers</td>
<td>352</td>
</tr>
<tr>
<td>Asynchronous/Queued Buses</td>
<td>353</td>
</tr>
<tr>
<td>File Access</td>
<td>354</td>
</tr>
<tr>
<td>CryPak File Archives</td>
<td>354</td>
</tr>
<tr>
<td>Features</td>
<td>354</td>
</tr>
<tr>
<td>Unicode and Absolute Path Handling</td>
<td>354</td>
</tr>
</tbody>
</table>
Channels and Message Priorities ................................................................. 411
Customizable Classes .................................................................................. 412
CarrierDesc ................................................................................................. 412
Synchronizing Game State Using Replicas .................................................. 414
Replicas ....................................................................................................... 414
Replica Chunks ............................................................................................ 418
Data Sets ...................................................................................................... 420
Remote Procedure Calls (RPCs) .................................................................... 422
Replica Manager ......................................................................................... 423
Marshaling and Unmarshaling ...................................................................... 425
Binding Game Objects to the Network Using Components ......................... 426
Using OpenSSL ........................................................................................... 431
Limitations .................................................................................................... 431
Implementation Support ............................................................................... 431
Cipher ........................................................................................................... 431
Building with Encryption ............................................................................. 432
Enabling Encryption ..................................................................................... 432
Controlling Bandwidth Usage ...................................................................... 435
Controlling the Send Rate ........................................................................... 435
Bandwidth Limiter ....................................................................................... 435
Controlling Burst Length ............................................................................. 436
Prioritization of Replica Updates ................................................................. 436
Tuning Bandwidth at Runtime ...................................................................... 437
Network Serialization and Aspects .............................................................. 437
Aspects ......................................................................................................... 437
Profiles ........................................................................................................ 438
RMI Functions ............................................................................................ 438
Ordering RMI Functions .............................................................................. 438
Ordering Rules ............................................................................................. 438
RMI Function Flags ..................................................................................... 439
Examples ..................................................................................................... 439
Physics .......................................................................................................... 441
Geometries ................................................................................................... 441
Geometry Management Functions ............................................................... 441
Physical Entities ............................................................................................ 442
Creating and managing entities ................................................................. 443
Functions for Entity Structures .................................................................... 445
Common Functions ...................................................................................... 446
Living Entity-Specific Functions ................................................................. 448
Particle Entity-Specific Functions ............................................................... 449
Articulated Entity-Specific Functions ......................................................... 450
Rope Entity-Specific Functions ................................................................... 452
Soft Entity-Specific Functions ..................................................................... 452
Collision Classes ........................................................................................ 453
Setup ............................................................................................................ 453
Code ............................................................................................................ 454
Types ........................................................................................................... 455
Filtering the collision ................................................................................... 455
Interface ...................................................................................................... 455
Functions for World Entities ......................................................................... 455
Advancing the Physical World Time State .................................................. 455
Returning Entities with Overlapping Bounding Boxes .............................. 456
Casting Rays in an Environment .................................................................. 457
Creating Explosions .................................................................................... 458
Profiler ......................................................................................................... 459
Profiler Tutorial .......................................................................................... 459
Registering Your Application ....................................................................... 460
Launching Profiler ....................................................................................... 460
Capturing Data .......................................................... 460
Inspecting Data .......................................................... 462
Playing Back Data ......................................................... 464
Exporting Data ............................................................ 469
Creating and Using Annotations ...................................... 469
Using Annotations ....................................................... 470
Creating Annotations ................................................... 471
Using Profiler for Networking ......................................... 473
Prerequisites ............................................................... 473
Carrier Profiler ......................................................... 473
Replica Activity Profiler ................................................ 474
Using the Profiler for CPU Usage ................................. 481
Understanding the Tree View ........................................ 482
Controlling the Display ................................................ 482
Using Profiler for VRAM .............................................. 484
Notes .......................................................................... 484
Understanding the Captured Data ................................. 485
Inspecting the Data ...................................................... 485
Using GridHub ............................................................. 487
Registering an Application in GridHub ......................... 487
Viewing and Configuring GridHub ............................... 487
Troubleshooting GridHub ............................................. 489
System ........................................................................ 490
Memory Handling ....................................................... 490
Hardware Memory Limitations .................................... 490
Choosing a Platform to Target ..................................... 490
Budgets ....................................................................... 491
Allocation Strategy with Multiple Modules and Threads .... 491
Caching Computational Data ........................................ 491
Compression ............................................................. 491
Disk Size ..................................................................... 491
Total Size ..................................................................... 491
Address Space ........................................................... 492
Bandwidth .................................................................... 492
Latency ........................................................................ 492
Alignment .................................................................... 492
Virtual Memory .......................................................... 493
Streaming .................................................................... 493
Streaming System ....................................................... 493
Low-level Streaming System ....................................... 493
Streaming and Levelcache Pak Files ............................. 499
Single Thread IO Access and Invalid File Access .......... 501
High Level Streaming Engine Usage .......................... 501
Text Localization and Unicode Support ......................... 503
Terminology ............................................................... 503
What encoding to use? ............................................... 504
How does this affect me when writing code? ................ 505
How does this affect me when dealing with text assets? .... 506
Utilities provided in CryCommon ................................. 507
Further reading ............................................................ 507
CryLog ....................................................................... 507
CryLog Logging Functionality ...................................... 507
Verbosity Level and Coloring ....................................... 507
Log Files ...................................................................... 508
Console Variables ...................................................... 508
CryConsole ................................................................. 509
Color coding .............................................................. 509
Dumping all console commands and variables ............... 509
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Console Variables</td>
<td>510</td>
</tr>
<tr>
<td>Adding New Console Commands</td>
<td>510</td>
</tr>
<tr>
<td>Console Variable Groups</td>
<td>511</td>
</tr>
<tr>
<td>Deferred execution of command line console commands</td>
<td>512</td>
</tr>
<tr>
<td>OVar Tutorial</td>
<td>513</td>
</tr>
</tbody>
</table>
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 Importer (Preview) (p. 115)
  A new FBX Importer enables you to bring single static FBX meshes and single materials into Lumberyard, and provides an abstraction layer that you can extend to accept other input formats.
• AZ Modules (Preview) (p. 120)
  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 initialization functions.
• Callback References (p. 130)
  Describes the callbacks for the Entity System (p. 320) and for GameRules scripts.
• Cloud Canvas (p. 135)
  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. 255)
  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. 320).
• Controller Devices and Game Input (p. 267)
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. 276)**

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. 314)**

Describes how to use Lumberyard Editor or the Lumberyard standalone Launcher to record benchmarking videos and capture audio.

- **Entity System (p. 320)**

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. 343)**

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. 354)**

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. 365)**

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. 374)**

Lua is Lumberyard's scripting language. This section contains a Lua scripting reference and provides topics on Lua script usage, Lua and C++ integration, the Lua remote debugger, and the Lua XML Loader.

- **Networking System (p. 402)**

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. 441)**

Describes the Lumberyard physics system and how to interact with the physics engine. This section shows how to create a physical world object, fill the world with geometries and physical entities, and control the entities with the functions described.

- **System (p. 490)**

Contains topics on memory handling, streaming, localization, logging, and console tools.
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. 14)
• AI C++ Class Hierarchy (p. 15)
• 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:

<table>
<thead>
<tr>
<th>Node type</th>
<th>Resistance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangular-no-water</td>
<td>0.5</td>
</tr>
<tr>
<td>Triangular-water</td>
<td>1.0</td>
</tr>
<tr>
<td>Waypoint</td>
<td>0.6</td>
</tr>
<tr>
<td>Road</td>
<td>0</td>
</tr>
<tr>
<td>Flight/Volume</td>
<td>0</td>
</tr>
</tbody>
</table>

**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:

\[
\text{Cost} = \text{Link.DestinationDanger} \times \text{Agent.DangerCost} + (1 + \text{Factor}) \times \text{Link.Length}
\]

where

\[
\text{Factor} = \text{Agent.[link-type]ResistanceFactor} \times \text{Link.[link-type]Resistance} + \text{Agent.ExposureFactor} \times \text{Link.Exposure}
\]

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 hard 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
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:

```javascript
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:
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:

```xml
<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:**

```cpp
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:

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

Lua Script

```lua
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.

```plaintext
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. 21)
- 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.

```cpp
// Check for shooter near cover using TPS
static const char *sQueryName = "CHitMiss::GetAccuracy";
ITacticalPointSystem *pTPS = gEnv->pAISystem->GetTacticalPointSystem();
```
```cpp
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, bIsVal idHidePos );
```

**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.

```cpp
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.

```cpp
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.

```cpp
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.

```cpp
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:

```lua
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' 'grid' | 'indoor' |
| BoolQuery ::= BoolProperty | (Test 'Glue' Object) |
| BoolProperty ::= 'coverSoft' | 'coverSuperior' | 'coverInferior' | 'currentlyUsedObject' |
| 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)

cchangeInDistance
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 produce 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 the 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 \(<\text{QueryTPS}>\) node can be used to call pre-defined TPS queries by name. The \(<\text{QueryTPS}>\) node will return either success or failure.

The most common usage pattern involving the \(<\text{QueryTPS}>\) node is to use it in conjunction with the \(<\text{Move}>\) node inside a \(<\text{Sequence}>\) to determine the status of a specified position. The example below illustrates a call to a pre-defined TPS query called \( \text{SDKGrunt_TargetPositionOnNavMesh} \), 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" avoid Dangers="0"/>
</Sequence>
```

The definition of the pre-defined query \( \text{SDKGrunt_TargetPositionOnNavMesh} \) 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 unit 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, \( t \rightarrow (x,y,z) \).
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:

```cpp
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 HMMWs 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:

```lua
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. 42)
- Refpoints (p. 86)
- Signals (p. 87)
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 `ChannelConfig.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

```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>
    <Channel name="Personal" priority="1" minSilence="2" actorMinSilence="3" type="personal"/>
  </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".

Example

```
<?xml version="1.0"?><Communications>
  <!--sound event example-->
  <Config name="Welcome">
    <Communication name="comm_welcome" finishMethod="sound" blocking="none">
      <Variation soundName="sounds/dialog:dialog:welcome" />
```

Version 1.4
38
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`

<table>
<thead>
<tr>
<th>Language</th>
<th>American English</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Path</td>
<td>languages/dialog/ai_npc_01/</td>
</tr>
<tr>
<td>Signal</td>
<td>Sound File</td>
</tr>
<tr>
<td><code>see</code></td>
<td><code>SDK NPC 01 Example</code></td>
</tr>
<tr>
<td>see_player_00</td>
<td>i see you</td>
</tr>
<tr>
<td>see_player_01</td>
<td>hey there you are</td>
</tr>
<tr>
<td>see_player_02</td>
<td>hey i have been looking for you</td>
</tr>
<tr>
<td>pain</td>
<td>pain_01</td>
</tr>
<tr>
<td>pain_01</td>
<td>ouch</td>
</tr>
</tbody>
</table>
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

```lua
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.

expiry (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" expiry="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" expiry="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

```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. 53).

**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.

![Animation](image1.png)  [Animate]

[Speak]

[Move]

**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.

```xml
<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.

```xml
<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).

```cpp
#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
    // 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.

```xml
<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.

  ```xml
  <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.

  ```xml
  <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.

  ```xml
  <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.
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.

```xml
<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" orNever BeenSet="1" />
      <Selector>
        <IfTime since="GroupMemberDied" isLessThan="10">
          <MoveCautiouslyTowardsTarget />
        </IfTime>
        <MoveConfidentlyTowardsTarget />
      </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.

```xml
<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.

```cpp
#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.

```xml
<Sequence>
    <QueryTPS name="CoverFromTarget" _startLog="Finding cover" _fail
ureLog="Failed to find cover" />
</Sequence>
```
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.

```c
#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.


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
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. 55)

- Loop (p. 55)
- LoopUntilSuccess (p. 55)
- Parallel (p. 55)
- Selector (p. 56)
- Sequence (p. 57)
- StateMachine (p. 57)
- State & Transitions (p. 57)
- SuppressFailure (p. 58)
- Timeout (p. 59)
- Wait (p. 59)

AI Nodes (p. 59)

- AdjustCoverStance (p. 59)
- Aim (p. 60)
- AimAroundWhileUsingAMachiningGun (p. 60)
- Animate (p. 61)
- AnimationTagWrapper (p. 61)
- AssertCondition (p. 61)
- AssertLua (p. 62)
- AssertTime (p. 62)
- Bubble (p. 63)
- 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. 67)
• Move (p. 68)
• Priority & Case (p. 69)
• PullDownThreatLevel (p. 70)
• QueryTPS (p. 70)
• RandomGate (p. 70)
• SendTransitionSignal (p. 71)
• SetAlertness (p. 71)
• Shoot (p. 72)
• ShootFromCover (p. 73)
• Signal (p. 73)
• SmartObjectStatesWrapper (p. 74)
• Stance (p. 74)
• StopMovement (p. 74)
• Teleport (p. 75)
• ThrowGrenade (p. 75)
• WaitUntilTime (p. 76)

**CryAction Nodes (p. 76)**

• AnimateFragment (p. 76)

**Game Nodes (p. 77)**

• InflateAgentCollisionRadiusUsingPhysicsTrick (p. 77)
• KeepTargetAtADistance (p. 77)
• Melee (p. 78)
• ScorcherDeploy (p. 79)
• SuppressHitReactions (p. 80)

**Flying Nodes (p. 81)**

• Hover (p. 81)
• FlyShoot (p. 81)
• WaitAlignedWithAttentionTarget (p. 81)
• Fly (p. 82)
• FlyForceAttentionTarget (p. 83)
• FlyAimAtCombatTarget (p. 83)
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

```xml
<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

```xml
<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

```xml
<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

```xml
<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

```xml
<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

```xml
<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
```xml
<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 (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 SUCCCEEDS once it has run the duration specified (plus random variation).

Example
```xml
<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

```xml
<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

```xml
<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.

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

- useReferencePointForInitialDirectionAndPivotPosition
  - Boolean.

Success/Failure

The node does not succeed or fail.
Example

```xml
<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

```xml
<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

```xml
<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

```xml
<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

```xml
<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

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

ClearTargets
Clears the AI agent's targets information.
**Parameters**

None.

**Success/Failure**

The node always SUCCEEDS.

**Example**

```xml
<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**

```xml
<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**

```xml
<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**

```xml
<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**

```xml
<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

```xml
<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

```xml
<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

```xml
<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 SUCCCEEDS 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.

turnTowardsMovementDirectionBeforeMoving
   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**

```xml
<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

```xml
<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

```xml
<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

```xml
<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

```xml
<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

```xml
<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

```xml
<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.

stanceToUseIfSlopesTooSteep
(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**

```xml
<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**

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

**Signal**

Sends a signal to the AI system. See Signals (p. 87).

**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**

```xml
<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

```xml
<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

```xml
<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**

```xml
<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**

```xml
<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**

```xml
<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**

```xml
<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**

```xml
<Melee target="AttentionTarget" cylinderRadius="1.5" hitType="hitTypeName" materialEffect="materialEffectName" />```
Lua table settings

The Lua table `melee` contains the following settings:

```lua
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**

```xml
<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**

```xml
<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

```xml
<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

```xml
<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 SUCCCEEDS 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 SUCCCEEDS 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.

<table>
<thead>
<tr>
<th>When</th>
<th>Lua variable</th>
<th>XML tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each node tick</td>
<td>Helicopter_Speed</td>
<td>desiredSpeed</td>
</tr>
<tr>
<td>Node activation</td>
<td>Helicopter_Loop</td>
<td>loopAlongPath</td>
</tr>
<tr>
<td>Node activation</td>
<td>Helicopter_StartFromClosestLocation</td>
<td>startPathFromClosestLocation</td>
</tr>
</tbody>
</table>

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_ForestedTargetId` 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**

```xml
<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**

```xml
<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**

```xml
<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

```xml
<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

```xml
<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

```xml
<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.

```lua
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, l, data.fValue); -- noncontinuous stick
      AI.PushGoal(g_StringTemp1, "signal", 0, 1, "VEHICLE_GOTO_DONE", SIGNAL_FILTER_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.

```lua
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:

```lua
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:

```lua
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:

```lua
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.

```lua
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:

```lua
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):

```lua
CoverIdle = {
    "CoverIdle",
    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 =
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:

```lua
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnNoTarget</td>
<td></td>
<td>Attention target is lost</td>
</tr>
</tbody>
</table>

#### Sound

Sound heard (no visible target).

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnSuspectedSound-Heard</td>
<td></td>
<td>Threat is AITHREAT_SUSPECT</td>
</tr>
<tr>
<td>OnInterestingSound-Heard</td>
<td></td>
<td>Threat is AITHREAT_INTERESTING</td>
</tr>
<tr>
<td>OnThreateningSound-Heard</td>
<td></td>
<td>Threat is AITHREAT_THREATENING</td>
</tr>
<tr>
<td>OnEnemyHeard</td>
<td></td>
<td>Threat is AITHREAT_AGGRESSIVE</td>
</tr>
</tbody>
</table>

#### Memory

The target is not visible and is in memory.

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnEnemyMemory</td>
<td></td>
<td>Threat is AITHREAT_THREATENING</td>
</tr>
<tr>
<td>OnLostSightOfTarget</td>
<td></td>
<td>Threat is AITHREAT_AGGRESSIVE</td>
</tr>
<tr>
<td>Name</td>
<td>Parameters</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>OnMemoryMoved</td>
<td></td>
<td>Threat is AITHREAT_AGGRESSIVE and its location or owner has changed</td>
</tr>
</tbody>
</table>

**Visual**

The target is visible.

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

**Awareness of Player**

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnPlayerLooking</td>
<td>sender, data</td>
<td>Player is looking at the AI for entity.Properties.awarenessOfPlayer seconds. data.iValue = player distance</td>
</tr>
<tr>
<td>OnPlayerSticking</td>
<td>sender</td>
<td>Player is staying close to the AI since &lt;entity.Properties.awarenessOfPlayer&gt; seconds</td>
</tr>
<tr>
<td>OnPlayerLookingAway</td>
<td>sender</td>
<td>Player has just stopped looking at the AI</td>
</tr>
<tr>
<td>OnPlayerGoingAway</td>
<td>sender</td>
<td>Player has just stopped staying close to the AI</td>
</tr>
</tbody>
</table>

**Awareness of Attention Target**

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnTargetApproaching</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnTargetFleeing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnNewAttentionTarget</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnAttentionTargetThreatChanged</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnNoTargetVisible</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Weapon Damage

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnBulletRain</td>
<td>sender</td>
<td>Enemy is shooting</td>
</tr>
<tr>
<td>OnDamage</td>
<td>sender, data</td>
<td>AI was damaged by another friendly/unknown AI. data.id = damaging AI's entity id</td>
</tr>
<tr>
<td>OnEnemyDamage</td>
<td>sender, data</td>
<td>AI was damaged by an enemy AI. data.id = damaging enemy’s entity id</td>
</tr>
</tbody>
</table>

### Proximity

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnCloseContact</td>
<td></td>
<td>enemy gets at a close distance to an AI (defined by Lua Property “damageRadius” of this AI)</td>
</tr>
<tr>
<td>OnCloseCollision</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Vehicles

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

### User-defined

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

```lua
AI.ResetRanges(entityID);
AI.AddRange(entityID, range, enterSignal, leaveSignal);
```
Weapon-Related Signals

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnLowAmmo</td>
<td></td>
<td>AI goes into automatic reload after its clip is empty</td>
</tr>
<tr>
<td>OnMeleeExecuted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnOutOfAmmo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnReload</td>
<td></td>
<td>AI has requested a path and the end of path is far from the desired destination</td>
</tr>
<tr>
<td>OnReloadDone</td>
<td></td>
<td>AI has requested a path which is not possible</td>
</tr>
<tr>
<td>OnReloaded</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Navigation Signals

Pathfinding

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnEndPathOffset</td>
<td>sender</td>
<td>AI has requested a path and the end of path is far from the desired destination</td>
</tr>
<tr>
<td>OnNoPathFound</td>
<td>sender</td>
<td>AI has requested a path which is not possible</td>
</tr>
<tr>
<td>OnPathFindAtStart</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnBackOffFailed</td>
<td>sender</td>
<td>AI tried to execute a &quot;backoff&quot; goal which failed</td>
</tr>
<tr>
<td>OnPathFound</td>
<td>sender</td>
<td>AI has requested a path and it's been computed successfully</td>
</tr>
</tbody>
</table>
### Steering

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnSteerFailed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Smart Objects

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnEnterNavSO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnLeaveNavSO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnUseSmartObject</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Navigation Shapes

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnShapeEnabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnShapeDisabled</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Tactics Signals

#### Tactical Point System

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnTPSDestNotFound</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnTPSDestFound</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnTPSDestReached</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Cover

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnHighCover</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnLowCover</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnMovingToCover</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnMovingInCover</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnEnterCover</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnLeaveCover</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnCoverCompromised</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Groups Signals

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnGroupChanged</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnGroupMemberMutated</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnGroupMemberDiedNearest</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Formation

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnNoFormationPoint</td>
<td>sender</td>
<td>AI couldn't find a formation point</td>
</tr>
<tr>
<td>OnFormationPointReached</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnGetToFormationPointFailed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Group Coordination

Group target is the most threatening target of the group.

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnGroupTargetNone</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnGroupTargetSound</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnGroupTargetMemory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnGroupTargetVisual</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PerformingRole</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Flow Graph Signals

These are signals sent by corresponding Flow Graph nodes when they are activated.

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACT_AIMAT</td>
<td></td>
<td>AI:AIShootAt</td>
</tr>
<tr>
<td>ACT_ALERTED</td>
<td></td>
<td>AI:AIAlertMe</td>
</tr>
<tr>
<td>ACT_ANIM</td>
<td></td>
<td>AI:AIAnim</td>
</tr>
<tr>
<td>ACT_ANIMEX</td>
<td></td>
<td>AI:AIAnimEx</td>
</tr>
<tr>
<td>ACT_CHASETARGET</td>
<td></td>
<td>Vehicle:ChaseTarget</td>
</tr>
<tr>
<td>ACT_DIALOG</td>
<td></td>
<td>AI:ReadabilityDialog (also sent by Dialog System)</td>
</tr>
</tbody>
</table>
### Signals

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACT_DIALOG_OVER</td>
<td></td>
<td>Sent by Dialog System</td>
</tr>
<tr>
<td>ACT_DUMMY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACT_DROP_OBJECT</td>
<td>AI:AI_dropObject</td>
<td></td>
</tr>
<tr>
<td>ACT_ENTERVEHICLE</td>
<td>Vehicle:Enter</td>
<td></td>
</tr>
<tr>
<td>ACT_EXECUTE</td>
<td>AI:AI_Execute</td>
<td></td>
</tr>
<tr>
<td>ACT_EXITVEHICLE</td>
<td>Vehicle:Exit, Vehicle:Unload</td>
<td></td>
</tr>
<tr>
<td>ACT_FOLLOW</td>
<td>AI:AI_Follow</td>
<td></td>
</tr>
<tr>
<td>ACT_FOLLOWPATH</td>
<td>AI:AI_FollowPath, AI:AI_FollowPath_speed_stance, Vehicle:FollowPath</td>
<td></td>
</tr>
<tr>
<td>ACT_GRAB_OBJECT</td>
<td>AI:AI_GrabObject</td>
<td></td>
</tr>
<tr>
<td>ACT_GOTO</td>
<td>AI:AI_Goto, AI:AI_Goto_speed_stance, and the AI Debugger when the user clicks the middle mouse button.</td>
<td></td>
</tr>
<tr>
<td>ACT_JOINFORMATION</td>
<td>AI:AI_FormationJoin</td>
<td></td>
</tr>
<tr>
<td>ACT_SHOOTAT</td>
<td>AI:AI_ShootAt</td>
<td></td>
</tr>
<tr>
<td>ACT_USEOBJECT</td>
<td>AI:AI_UseObject</td>
<td></td>
</tr>
<tr>
<td>ACT_VEHICLESTICKPATH</td>
<td>Vehicle:StickPath</td>
<td></td>
</tr>
<tr>
<td>ACT_WEAPONDRAW</td>
<td>AI:AI_WeaponDraw</td>
<td></td>
</tr>
<tr>
<td>ACT_WEAPONHOLSTER</td>
<td>AI:AI_WeaponHolster</td>
<td></td>
</tr>
<tr>
<td>ACT_WEAPONSELECT</td>
<td>AI:AI_WeaponSelect</td>
<td></td>
</tr>
</tbody>
</table>

### Other Signals

#### Forced Execute

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnForcedExecute</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnForcedExecuteComplete</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Animation

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnimationCanceled</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Game

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnFallAndPlay</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Vehicle-related

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnActorSitDown</td>
<td>Actor has entered a vehicle</td>
<td></td>
</tr>
</tbody>
</table>

### Squads

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnSomebodyDied</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnBodyFallSound</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnBodyFallSound</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnUnitDied</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnSquadmateDied</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnPlayerTeamKill</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnUnitBusy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnPlayerDied</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnFriendInWay</td>
<td>sender</td>
<td>AI is trying to fire and another friendly AI is on his line of fire</td>
</tr>
<tr>
<td>URPRISE_ACTION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnActionDone</td>
<td>data</td>
<td>AI action of this agent was finished. dataObjectName 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 &quot;the object&quot; of the AI action</td>
</tr>
</tbody>
</table>
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 predictable, 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:

```c
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:
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:

```c
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):

```c
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):

```c
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:

```c
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:

```c
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:

```c
gEnv->pCharacterManager->CAF_IsLoaded(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 TWeight: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 TWeight:1.00 seg:00 inmem:1
```

- **Layer index (decimal, zero-based)**
- **User token (decimal)**
- **Animation name (alias) of the currently playing animation, aim/look-pose or bspace**
- **ATime:0.84 (1.17s/1.40s)**
  - **ATime:XXX (YYYYs/ZZZZs)**
  - **XXX** = 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:

<table>
<thead>
<tr>
<th>char</th>
<th>flag</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>CA_FORCE_TRANSITION_TO_ANIM</td>
<td>0x008000</td>
</tr>
<tr>
<td>y</td>
<td>CA_FULL_ROOT_PRIORITY</td>
<td>0x004000</td>
</tr>
<tr>
<td>b</td>
<td>CA_REMOVE_FROM_PRO</td>
<td>0x002000</td>
</tr>
<tr>
<td>V</td>
<td>CA_TRACK_EXCLUSIVE</td>
<td>0x001000</td>
</tr>
<tr>
<td>F</td>
<td>CA_FORCE_SKELETON_UPDATE</td>
<td>0x000800</td>
</tr>
<tr>
<td>x</td>
<td>CA_DISABLE_MULTI_LAYER</td>
<td>0x000400</td>
</tr>
<tr>
<td>3</td>
<td>CA_KEYFRAME_SAMPLE_30Hz</td>
<td>0x000200</td>
</tr>
<tr>
<td>n</td>
<td>CA_ALLOW_ANIM_RESTART</td>
<td>0x000100</td>
</tr>
<tr>
<td>S</td>
<td>CA_MOVE2IDLE</td>
<td>0x000080</td>
</tr>
<tr>
<td>I</td>
<td>CA_IDLE2MOVE</td>
<td>0x000040</td>
</tr>
<tr>
<td>A</td>
<td>CA_START_AFTER</td>
<td>0x000020</td>
</tr>
<tr>
<td>K</td>
<td>CA_START_AT_KEY_TIME</td>
<td>0x000010</td>
</tr>
<tr>
<td>T</td>
<td>CA_TRANSITION_TIME-WARPING</td>
<td>0x000008</td>
</tr>
<tr>
<td>R</td>
<td>CA_REPEAT_LAST_KEY</td>
<td>0x000004</td>
</tr>
<tr>
<td>L</td>
<td>CA_LOOP_ANIMATION</td>
<td>0x000002</td>
</tr>
<tr>
<td>M</td>
<td>CA_MANUAL_UPDATE</td>
<td>0x000001</td>
</tr>
</tbody>
</table>
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
  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:

<table>
<thead>
<tr>
<th>Frame Index</th>
<th>Real Time (seconds)*</th>
<th>Animation Normalized Time**</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0 s</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>0.033.. s = 1/30 s</td>
<td>0.022.. = 1/45</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>30</td>
<td>1.0 s</td>
<td>0.666.. = 30/45</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>44</td>
<td>1.466.. s = 44/30 s</td>
<td>0.977.. = 44/45</td>
</tr>
</tbody>
</table>
* 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.

<table>
<thead>
<tr>
<th>Frame Index</th>
<th>Real Time</th>
<th>AnimEvents</th>
<th>(Animation) Normalized Time</th>
<th>Segment Index*</th>
<th>Segment Normalized Time**</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0 s</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Frame Index</td>
<td>Real Time</td>
<td>AnimEvents</td>
<td>(Animation) Normalized Time</td>
<td>Segment Index*</td>
<td>Segment NormalizedTime**</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
<td>------------</td>
<td>-----------------------------</td>
<td>----------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>1</td>
<td>0.033.. s</td>
<td></td>
<td>0.022..</td>
<td>0</td>
<td>0.033.. = 1/30</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>30</td>
<td>1.0 s</td>
<td>segment1</td>
<td>0.666..</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>44</td>
<td>1.466.. s</td>
<td></td>
<td>0.977..</td>
<td>1</td>
<td>0.933.. = 14/15</td>
</tr>
<tr>
<td>45</td>
<td>1.5 s</td>
<td></td>
<td>1.0</td>
<td>1</td>
<td>1.0</td>
</tr>
</tbody>
</table>

* 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::GetCurrentSegmentExpectedDurationSeconds()` 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.

<table>
<thead>
<tr>
<th>Frame Index</th>
<th>Real Time</th>
<th>I_CAF Ticks*</th>
<th>Keytime**</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0 s</td>
<td>0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Version 1.4  
113
**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.

<table>
<thead>
<tr>
<th>Frame Index</th>
<th>Real Time</th>
<th>I_CAF Ticks*</th>
<th>Keytime**</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.033.. s</td>
<td>160</td>
<td>1.0</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>30</td>
<td>1.0 s</td>
<td>4800</td>
<td>30.0</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>44</td>
<td>1.466.. s</td>
<td>7040</td>
<td>44.0</td>
</tr>
<tr>
<td>45</td>
<td>1.5 s</td>
<td>7200</td>
<td>45.0</td>
</tr>
</tbody>
</table>
Asset Importer (Preview) Technical Overview

The Lumberyard Asset Importer and the FBX Importer are in preview release and are subject to change.

This topic provides a high-level technical overview of the Lumberyard Asset Importer architecture and its core libraries. The preview release of the Asset Importer features:

- A new FBX Importer (built entirely from scratch) that makes it easier for you to bring single static FBX meshes and single materials into Lumberyard. For information on how to use the FBX Importer tool, available from Lumberyard Editor, see FBX Importer in the Amazon Lumberyard User Guide.
- An abstraction layer that you can extend to accept other input formats.

You can use the abstraction layer to add support for older formats like OBJ and DAE, or for custom data formats. A generic layer of data called the scene graph enables this extensibility, as explained in the following section.

Architecture and Concepts

Following are the key concepts and components of the Lumberyard Asset Importer.

**FBX** – File extension for Autodesk’s .fbx file format. This is a general use container file for many types of data commonly used in video game development. Most modeling packages have the ability to export the FBX file format natively. The preview release of the Lumberyard Asset Importer supports this format.

**Importer** – Code that takes input data creates a scene graph (Lumberyard’s generic tree-based representation of data) or scene manifest from it. For example, you might implement an FbxSceneGraphImporter that can construct a scene graph from an FBX file. Alternatively, you could have a JsonManifestImporter that constructs a scene manifest from a JSON file.

**Scene** – This is an abstraction that contains a scene graph tree data structure and a scene manifest. The scene manifest contains metadata about the scene graph.
Scene graph – A generic tree-based representation of data. The scene graph is a layer of data between an intermediate 3D asset (like FBX) and the Lumberyard engine. A scene graph is composed internally of scene nodes which optionally contain data in the form of a data object.

Scene node – A node within a scene graph that may contain data in the form of a data object, or exist solely for hierarchical reasons. A node's name is made unique in the scene graph by concatenating all its parent nodes and delimiting them with a '.'. For example, if a node named root has the two children child1 and child2, the full node names are root, root.child1, and root.child2. Even if another node has a child named child2, the full name is unique because of the hierarchy.

Data object – A generic container for data which supports RTTI (run-time type information) and casting through the AZ_RTTI system. This is the basic unit of storage used by both the SceneManifest key-value-pair dictionary and the SceneNode internal data payload. The DataObject class provides limited reflection support for editing of class data, but not serialization to file.

Scene manifest – This is a flat dictionary of metadata about a scene graph. This metadata will be mostly in the form of groups and rules, but supports generic and anonymous data provided by the user as well. All data is stored as key-value-pairs, with the key being a string, and the value being a DataObject.

Groups – These structures, managed by a scene manifest, contain rules that specify how to process scene graph data so that game data files can be generated. The current version of the Asset Importer supports mesh groups, which can generate CGF and MTL files.

Rules – These structures, contained in groups, specialize the handling of asset processing or display information like metadata or comments to the user. For example, you might create rules to change the vertex indexing bit count or provide positional or rotation offsets in the files that are output. The current version of the Asset Importer supports the rule types origin, advanced mesh, material, and comment.

Exporter – Code that takes data from a scene graph or scene manifest and changes it into another data format. For example, you might have a CgfSceneExporter that exports scene graph data using instructions from the scene manifest into the CContentCGF structure in memory. You could use CgfExporter, which allows the creation of static geometry files. You could also have a JsonManifestExporter that writes the data contained in a scene manifest into a JSON file on disk.

The following diagram illustrates the relationships among the components described.
Other Components

**Asset Processor** – The Asset Importer uses the Lumberyard Asset Processor. The asset processor runs in the background looking for changes in source data like FBX files or their metadata like `.scenesettings` files. When source or metadata files are added, removed, or changed, the asset processor automatically reprocesses the source data and outputs game data in a cache directory.

**Reflection** – The Lumberyard engine supports a robust and performant reflection system built on the AZ_RTTI system of AzCore. The Asset Importer and associated systems display data through the Lumberyard ReflectedPropertyGrid UI system that is part of the AzToolsFramework. The AzToolsFramework is a common UI framework for displaying and editing class data.

Core Libraries

Following are the core libraries of the Asset Importer. The name of each library is followed by its directory location in parentheses.

**SceneCore** (`\dev\Code\Tools\SceneAPI\SceneCore`) – This library contains common data storage structures, specifically: Scene, SceneGraph, SceneManifest, and DataObject. It also contains interfaces for currently supported Group, Rule, and SceneNode data types. It contains basic iterators that can be used to easily iterate through SceneGraph or SceneManifest data. Finally, it contains basic importers and exporters common to all use cases.

**FbxSceneBuilder** (`\dev\Code\Tools\SceneAPI\FbxSceneBuilder`) – This library enables the loading of FBX data and its conversion into SceneGraph data. For convenience, it includes the full source of the lightweight pass-through FBXSdkWrapper. The FBXSdkWrapper includes the importer for FBX data so that SceneCore does not require a dependency on the FBX SDK. This is the suggested model.
if you want to extend the SceneAPI codebase to support file formats that have dependencies on specialized libraries.

**SceneData** (\dev\Code\Tools\SceneAPI\SceneData) – This library contains the concrete implementations provided with the Lumberyard Asset Importer for Group, Rule, and SceneNode data types. These implementations are provided in a separate library that illustrates how you can extend or implement your own rule sets by using the SceneCore library. The SceneData library also contains factories for generating different types of data.

**EditorAssetImporter** (\dev\Code\Sandbox\Plugins\EditorAssetImporter) – This is the Lumberyard Editor plugin that allows the editing of scene manifests. It uses the FbxSceneBuilder and SceneCore libraries to read and write data defined in the SceneData library to disk when the user requests it. Currently, EditorAssetImporter supports only files that have the .fbx file extension. Other SceneBuilder implementations could support additional file types.

**ResourceCompilerScene** (\dev\Code\Tools\rc\ResourceCompilerScene) – The Resource Compiler (rc.exe) plugin uses the instructions from SceneManifest to process SceneGraph and generate game data. The Resource Compiler uses the FbxSceneBuilder and SceneCore libraries to read, write, and store data in a format defined by the SceneData library. The Resource Compiler is currently associated with the .fbx file extension. Other importers, if implemented, could translate data from non-FBX sources into SceneGraph data.

**FBXSdkWrapper** (\dev\Code\Tools\SceneAPI\FBXSdkWrapper) – As mentioned, this is a wrapper around the FBX SDK provided for convenience and testing. If you have an internal FBX serialization and storage model, you can use this library to replace it with the Autodesk FBX SDK.

**FBX Importer Example Workflow**

The following workflow illustrates how these libraries might work together in a production environment.

1. You open the Lumberyard Editor and launch the **FBX Importer**. The importer loads the EditorAssetImporter.dll Lumberyard Editor plugin. Note: For information on how to use the FBX Importer, see FBX Importer.

2. Using the FBX Importer file browser, you choose an FBX file that has been output from the modeling package. The FBX Importer (a specialized importer contained in FbxSceneBuilder) creates a SceneGraph that contains the equivalent of the data in the .fbx file.

3. If this is the first time you are importing the .fbx file, an empty SceneManifest is also created. If you have already imported the file, the JsonManifestImporter loads the corresponding .scenesettings file from the same directory and creates a new SceneManifest.

4. Using the Asset Importer UI, you create groups that define the output files that the Lumberyard Asset Importer will create. The groups specify names for the output files and the target nodes in the SceneGraph that will generate the data for the output files. You also create rules to change how the output files are processed.

5. When you have created all the desired groups and rules for the scene graph, you click **Import**. This creates or overwrites the scene manifest by saving it to a .scenesettings file in the same location as the .fbx file you loaded. For example, if character.fbx is loaded, a character.scenesettings file is created in the same directory. JsonManifestExporter performs this operation. If an MTL file does not exist, MaterialExporter generates a default MTL file.

6. The Asset Processor detects the creation of, or any change to, the .scenesettings file and calls the Resource Compiler (rc.exe) so that the .fbx file will be reprocessed.

7. The Resource Compiler detects the .fbx file type, recognizes that it has a ResourceCompilerScene plugin module that processes .fbx files, and runs the plugin.

8. The ResourceCompilerScene plugin creates a scene graph and scene manifest that contain data equivalent to the data in the .fbx file. The plugin parses the groups and rules in SceneManifest and...
calls specialized exporters that use these groups and rules to write the scene graph data to game data files.

9. When the Resource Compiler completes, the MTL and the CGF files for the models are in the correct location and ready for use in your game.

Possible Asset Importer Customizations

Because SceneGraph and SceneManifest are flexible and extensible constructs, you can use them to easily process additional data types. The DataObject structure, which allows RTTI-based handling of arbitrary data, is key to this flexibility. Following are some customization examples:

- **Input File Types**: To process file types other than FBX, you implement a new importer that generates a SceneGraph from an arbitrary data format.

- **Input File Data**: Because the SceneNode in SceneGraph accepts arbitrary data payloads in the form of RTTI enabled DataObject structures, you build a SceneGraph structure to handle specialized or anonymous data unique to your team.

- **Custom Asset Processing Instructions**: Because Group and Rule are implemented as pure virtual interfaces, you easily customize them with instructions specific to your project.
  - Because the SceneManifest supports anonymous data like strings and numbers, the code you require for prototyping is kept to a minimum.
  - You extend or replace the basic concrete implementation of the SceneData library to fulfill your team's specific asset processing requirements.
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.

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.

Lumberyard currently supports both legacy modules and AZ modules but going forward will focus on AZ modules. If you are a game developer who wants to get a head start on developing code for this new technology, consider building your code as an AZ module. At the present time, gems support only legacy modules, so creating an AZ module requires some extra steps. However, gem support for AZ modules is planned for the near future, so if you begin building your code as an AZ module now, you can move it into a gem in a future release.

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:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Legacy Modules</th>
<th>AZ Modules</th>
</tr>
</thead>
</table>

Lumberyard Developer Guide
Comparing AZ Modules to Legacy Modules

Version 1.4
120
## 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. 343), and the Component Entity system. These systems are fast, flexible, and work together in a cohesive manner. The AZ framework supports game development with Lumberyard tools.

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compatibility</td>
<td>Modules can be converted to AZ modules with no loss of functionality.</td>
<td>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.</td>
</tr>
<tr>
<td>Ease of adding services (singleton classes) to modules</td>
<td>Adding services usually requires editing files in CryCommon. A file for the singleton’s class interface must exist in the CryCommon folder, and a variable to hold the singleton in gEnv must exist.</td>
<td>Modules create components and attach them to the system entity. No editing of game engine files is required.</td>
</tr>
<tr>
<td>Ease of use for low-level application features</td>
<td>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.</td>
<td>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.</td>
</tr>
<tr>
<td>Exposure of properties</td>
<td>Modules have no uniform way to let users control settings for their service. Some services read settings from .xml files in the assets folder, which must be edited by hand.</td>
<td>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.</td>
</tr>
<tr>
<td>Game engine dependency</td>
<td>Modules must run in the game engine and are difficult to extend for use in tools which do not have game code.</td>
<td>Modules are not specific to the game engine and can be used outside them.</td>
</tr>
<tr>
<td>Initialization functions</td>
<td>Function parameters are specific to CryEngine.</td>
<td>Function parameters are specific to the AZ framework (for more information, see the following section).</td>
</tr>
<tr>
<td>Order of initialization</td>
<td>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.</td>
<td>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.</td>
</tr>
</tbody>
</table>
but can also be used outside it. For example, several tools that are part of Lumberyard, like the Asset Processor and the Component Entity system use the AZ Framework and AZ Modules, but contain no game code.

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 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, a tool is planned that will allow you to edit the properties of system components just as you edit the properties of in-game components.

Current Lumberyard AZ Modules

Lumberyard currently provides two AZ modules, but more are planned.

LmbrCentral

The LmbrCentral module ties together systems from CryEngine and systems from the AZ framework. LmbrCentral is used by game applications.

LmbrCentralEditor

The LmbrCentralEditor module is used by the Lumberyard Editor and is not for use in standalone game applications. LmbrCentralEditor contains Lumberyard Editor versions of certain components in addition to all the code from LmbrCentral.

Creating an AZ Module

AZ modules are in preview release and subject to change.

This tutorial guides you through the tasks of creating a sample AZ module named "HelloWorld," connecting the module to your project, and calling the service that the module provides.

The tasks and their subtasks can be summarized as follows:

A. Creating a Module
   I. Create an appropriate file layout for your module.
   II. Write the public interface for a service that your module provides.
   III. Write a system component that implements the service.
   IV. Write an AZ::Module class to initialize the service in your module.
   V. Write the build files for your module.
B. Connecting the Module to Your Project
   I. Build your module with Waf.
II. Configure your project to load the module.
III. Add the module’s public interfaces to your project's include paths.

C. Calling the Module's Service

Creating a Module

In this section, you create an appropriate file layout for your module. You then write a public interface for the service that your module provides, a system component to implement the service, and an \texttt{AZ::Module} class to initialize the service. Finally, you create the appropriate build files for your module.

1. Create an Appropriate File Layout for Your Module

Your module should have a directory and file structure similar to the following:

```
HelloWorld/
  include/
    HelloWorld/
      HelloWorldBus.h
source/
    HelloWorldSystemComponent.h
    HelloWorldModule.cpp
    HelloWorld.waf_files
    wscript
```

2. Write the Public Interface for the Service That Your Module Provides

The interface for this service is written as an EBus. The EBus interface enables code from any other module to make calls to the service that the module provides. For information about EBus authoring, see Event Bus (EBus) (p. 343).

The following example declares an EBus interface for the HelloWorld service.

```
#include <AzCore/EBus/EBus.h>
namespace HelloWorld
{
    // Declare the interface for a service provided by this module.
    // Use an EBus so that code from any other module can easily make calls
    // to this service.
    class HelloWorldRequests : public AZ::EBusTraits
    {
        public:
            // OVERRIDE EBusTraits to customize the behavior of an EBus.
            // 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.
```
3. Write a System Component that Implements the Service

The following example declares a component for the system entity. The component handles calls to the request bus that you created. The code provides a way to activate and deactivate the service, uses reflection to expose properties that can be customized, and states the service provided so that other components can declare a dependency on it if necessary.

HelloWorldSystemComponent.h

```cpp
#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
    {
        public:
            // Every component definition must contain the AZ_COMPONENT macro,
            // specifying the type name and a unique UUID.
            AZ_COMPONENT(HelloWorldSystemComponent, "{38605002-545D-4C6C-8453-F363C7540533}")
            // Calls to HelloWorldRequestBus::SayHello(...) are handled here.
            void SayHello(const char* name) override
            {
                AZ_Printf("HelloWorld", "Hello %s, you certainly look smashing to night.", name);
            }
        private:
            // Activate() is where the component "turns on".
            void Activate() override
            {
                // Begin handling calls to HelloWorldRequestBus
                HelloWorldRequestBus::Handler::BusConnect();
            }
            // Deactivate() is where the component "turns off".
            void Deactivate() override
            {
                // Stop handling calls to HelloWorldRequestBus
                HelloWorldRequestBus::Handler::BusDisconnect();
            }
            // Reflect properties that developers may want to customize.
```
4. Write an AZ::Module Class to Initialize the Service in Your Module

The following example creates a class that inherits from AZ::Module. This class reflects on other classes in the module and adds required system components to the system entity. The example also declares functions for initialization.

**HelloWorldModule.cpp**

```cpp
#include <AzCore/Module/Module.h>
#include <AzCore/Component/Entity.h>
#include "HelloWorldSystemComponent.h"
namespace HelloWorld
{
    // Each module should contain a class which inherits from AZ::Module.
    // This class performs tasks such as reflecting the classes within
    // the module and adding critical components to the system entity.
    class HelloWorldModule
    : public AZ::Module
    {
        public:
            HelloWorldModule()
            : AZ::Module()
            {
                // Push component descriptors into m_descriptors here.
                m_descriptors.push_back(HelloWorldSystemComponent::Create
```
5. Write Build Files for Your Module

Lumberyard uses the Waf build system to ensure you build only what is needed. A .waf_files file lists the source files to include in the build, and a wscript file defines the build configuration for your game. For more information, see Waf Build System.

HelloWorld.waf_files

The HelloWorld.waf_files file lists the include and source files to include in your build, as follows:

```json
{
    "auto":
    {
        "include":
        [
            "include/HelloWorld/HelloWorldBus.h"
        ],
        "source":
        [
            "source/HelloWorldModule.cpp",
            "source/HelloWorldSystemComponent.h"
        ]
    }
}
```

wscript

In the wscript file, target specifies the name of the module, vs_filter specifies the Visual Studio filter, and file_list specifies the name of the waf_files file, as follows:

```python
def build(bld):
    bld.CryEngineNonRCModule(
        target          = 'HelloWorld',
        vs_filter       = 'Game',  # visual studio filter path
        file_list       = 'HelloWorld.waf_files',
        use             = ['AzFramework'],
        includes        = ['include', 'source'],
    )
```

B. Connecting the Module to Your Project

Connecting the module to your project currently requires the manual steps described in this section.
1. Build Your Module with Waf

Now you’re ready to use Waf to build your module.

To build your module with Waf

1. Modify the \texttt{wscript} in a parent directory so that Waf recurses into your module’s directory, as in the following example:

```
parent wscript

# ...
SUBFOLDERS = [
    # ..., 
    'HelloWorld'
]
# ...
```

2. Add your module to the appropriate .\texttt{json} spec files in the Lumberyard dev\_WAF\_specs\ directory so that waf can build your module, as in the following example:

```
Example spec file

{
// ...
"modules":
{ // ...
    "HelloWorld"
} // ...
```

2. Configure Your Project to Load the Module

Next you need to configure your project to load the module you just created.

To configure your project to load your module

1. Add your module to your Lumberyard directory’s dev\LYConfig_Default.xml file. This addition causes nonmonolithic builds to load and initialize your module.

```
LYConfig_Default.xml

<ObjectStream version="1">
  <Class name="ComponentApplication::Descriptor" type="{70277A3E-2AF5-4309-9BBF-6161AFBDE792}">
    <!-- ... -->
  </Class>
</ObjectStream>
```

```
<Class name="AZStd::vector" field="modules" type="{2BADE35A-6F1B-4698-B2BC-3373D010020C}"/>
  <Class name="DynamicModuleDescriptor" field="element" type="{D2932FA3-282V-4..."}
```
2. Add your module to your Lumberyard directory's `dev\Code\Engine\LmbrCentral\source\StaticModules.cpp` file, as in the following example. This addition causes monolithic builds to load and initialize your module.

```
// ...
extern "C" AZ::Module* CreateModuleClass_LmbrCentral();
extern "C" AZ::Module* CreateModuleClass_HelloWorld();

extern "C" void CreateStaticModules(AZStd::vector<AZ::Module*> & modulesOut)
{
    modulesOut.push_back(CreateModuleClass_LmbrCentral());
    modulesOut.push_back(CreateModuleClass_HelloWorld());
}
// ...
```

3. If your module must be loaded by Lumberyard Editor, add your module to your Lumberyard directory's `dev\LyConfig_Editor.xml` file as follows:

```
<ObjectStream version="1">
  <Class name="ComponentApplication::Descriptor" type="{70277A3E-2AF5-4309-9BBF-6161AFBDE792}">
    <!-- ... -->
    <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="LmbrCentralEditor" type="{EF8FF807-DDEE-4EB0-B678-4CA3A2C490A4}"/>
      </Class>
      <Class name="DynamicModuleDescriptor" field="element" type="{D2932FA3-9942-4FD2-A703-2E750F57C003}">
        <Class name="AZStd::string" field="dynamicLibraryPath" value="HelloWorld" type="{EF8FF807-DDEE-4EB0-B678-4CA3A2C490A4}"/>
      </Class>
    </Class>
  </Class>
</ObjectStream>
```
3. Add the Module’s Public Interfaces to Your Project’s Include Paths

In your project’s wscript file, edit the includes statement to add your module’s public interfaces to the project’s include paths, as in the following example:

```
# ...
includes = [..., bld.Path('Code/Engine/HelloWorld/include')],
# ...
```

C. Calling the Module’s Service

Use EBus to invoke the SayHello function in your module at an appropriate point in your project code, as follows:

```
#include <HelloWorld/HelloWorldBus.h>

void InSomeFunctionSomewhere()
{
    // ...
    // Invoke the call through EBus.
    EBUS_EVENT(HelloWorld::HelloWorldRequestBus, SayHello, "Bruce");
    // ...
}
```
Callback References

This section provides script callback information for the Entity system and game rules.

This section includes the following topics:
- Entity System Script Callbacks (p. 130)
- Game Rules Script Callbacks (p. 132)

Entity System Script Callbacks

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

<table>
<thead>
<tr>
<th>Callback Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnSpawn</td>
<td>Called after an entity is created by the Entity system.</td>
</tr>
<tr>
<td>OnDestroy</td>
<td>Called when an entity is destroyed (like OnShutDown() gets called).</td>
</tr>
<tr>
<td>OnInit</td>
<td>Called when an entity gets initialized via ENTITY_EVENT_INIT, and when its ScriptProxy gets initialized.</td>
</tr>
<tr>
<td>OnShutDown</td>
<td>Called when an entity is destroyed (like OnDestroy() gets called).</td>
</tr>
<tr>
<td>OnReset</td>
<td>Usually called when an editor wants to reset the state.</td>
</tr>
<tr>
<td>OnPropertyChange</td>
<td>Called by Lumberyard Editor when the user changes one of the properties.</td>
</tr>
</tbody>
</table>
## Script State Functions

<table>
<thead>
<tr>
<th>Callback Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnBeginState</td>
<td>Called during Entity.GotoState() after the state has been changed (that is, after OnEndState() is called on the old state).</td>
</tr>
<tr>
<td>OnBind</td>
<td>Called when a child entity is attached to an entity. Parameters include:</td>
</tr>
<tr>
<td></td>
<td>• script table for the child entity</td>
</tr>
<tr>
<td>OnCollision</td>
<td>Called when a collision between an entity and something else occurs. Parameters include:</td>
</tr>
<tr>
<td></td>
<td>• script table with information about the collision</td>
</tr>
<tr>
<td>OnEndState</td>
<td>Called during Entity.GotoState() while the old state is still active and before OnBeginState() is called on the new state.</td>
</tr>
<tr>
<td>OnEnterArea</td>
<td>Called when an entity has fully entered an area or trigger. Parameters include:</td>
</tr>
<tr>
<td></td>
<td>• areald (int)</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td>OnEnterNearArea</td>
<td>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.</td>
</tr>
<tr>
<td>OnLeaveArea</td>
<td>Called when an entity has fully left an area or trigger. Parameters include:</td>
</tr>
<tr>
<td></td>
<td>• areald (int)</td>
</tr>
<tr>
<td></td>
<td>• fade fraction (float) This value is always 0.0f.</td>
</tr>
<tr>
<td>OnLeaveNearArea</td>
<td>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.</td>
</tr>
<tr>
<td>OnMove</td>
<td>Called whenever an entity moves through the world.</td>
</tr>
<tr>
<td>OnMoveNearArea</td>
<td>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.</td>
</tr>
<tr>
<td>OnProceedFadeArea</td>
<td>Called when an entity has recently entered an area and fading is still in progress. Parameters include:</td>
</tr>
<tr>
<td></td>
<td>• areald (int)</td>
</tr>
<tr>
<td></td>
<td>• fade fraction (float)</td>
</tr>
</tbody>
</table>
### Callback Function

<table>
<thead>
<tr>
<th>Callback Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| **OnSoundDone**          | Called when a sound stops. Parameters include:  
  - `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 timer, 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

This topic provides reference information on callbacks used with the GameRules scripts.

<table>
<thead>
<tr>
<th>Callback Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| **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` |
<table>
<thead>
<tr>
<th>Callback Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnClientEnteredGame</td>
<td>Called when a player enters the game and is part of the game world. Called on the server only.</td>
</tr>
<tr>
<td></td>
<td>• <strong>channelId</strong> – Channel identifier of the player.</td>
</tr>
<tr>
<td></td>
<td>• <strong>playerScriptTable</strong> – The player’s script table.</td>
</tr>
<tr>
<td></td>
<td>• <strong>bReset</strong> – Boolean indicating whether or not the channel is from the reset list.</td>
</tr>
<tr>
<td></td>
<td>• <strong>bLoadingSaveGame</strong> – Boolean indicating whether or not the call was made during a saved game loading.</td>
</tr>
<tr>
<td>OnDisconnect</td>
<td>Called when the player disconnects on the client. Called on the client only.</td>
</tr>
<tr>
<td></td>
<td>• <strong>cause</strong> – Integer identifying the disconnection cause. See EDisconnectionCause.</td>
</tr>
<tr>
<td></td>
<td>• <strong>description</strong> – Human readable description of the disconnection cause.</td>
</tr>
<tr>
<td>OnChangeSpectatorMode</td>
<td>Called when a player changes the spectator mode. Called on the server only.</td>
</tr>
<tr>
<td></td>
<td>• <strong>entityId</strong> – Player who made the change.</td>
</tr>
<tr>
<td></td>
<td>• <strong>mode</strong> – New spectator mode (1=fixed, 2=free, 3= follow).</td>
</tr>
<tr>
<td></td>
<td>• <strong>targetId</strong> – Possible target entity to spectate.</td>
</tr>
<tr>
<td></td>
<td>• <strong>resetAll</strong> – Boolean indicating whether or not to reset player-related things like the inventory.</td>
</tr>
<tr>
<td>OnChangeTeam</td>
<td>Called when a player switches teams. Called on the server only.</td>
</tr>
<tr>
<td></td>
<td>• <strong>entityId</strong> – Player who switched teams.</td>
</tr>
<tr>
<td></td>
<td>• <strong>teamId</strong> – New team identifier.</td>
</tr>
<tr>
<td>Callback Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>OnExplosion</strong></td>
<td>Called when an explosion is simulated. Called on the server and client.</td>
</tr>
<tr>
<td></td>
<td>• <strong>pos</strong> – Position of the explosion in the game world.</td>
</tr>
<tr>
<td></td>
<td>• <strong>dir</strong> – Direction of the explosion.</td>
</tr>
<tr>
<td></td>
<td>• <strong>shooterId</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>weaponId</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>shooter</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>weapon</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>materialId</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>damage</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>min_radius</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>radius</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>pressure</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>hole_size</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>effect</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>effectScale</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>effectClass</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>typeld</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>type</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>angle</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>impact</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>impact_velocity</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>impact_normal</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>impact_targetId</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>shakeMinR</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>shakeMaxR</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>shakeScale</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>shakeRnd</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>effect</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>impact_velocity</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>impact_normal</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>impact_targetId</strong></td>
</tr>
<tr>
<td></td>
<td><strong>AffectedEntities</strong> – Affected entities table.</td>
</tr>
<tr>
<td></td>
<td><strong>AffectedEntitiesObstruction</strong> – Affected entities obstruction table.</td>
</tr>
</tbody>
</table>
Cloud Canvas

Cloud Canvas is a visual scripting interface to AWS services. With Cloud Canvas you can build connected game features that use Amazon DynamoDB, AWS Lambda, Amazon Simple Storage Service (Amazon S3), Amazon Cognito, Amazon Simple Notification Service (Amazon SNS), and Amazon Simple Queue Service (Amazon SQS). With Cloud Canvas, you can create cloud-hosted features such as daily gifts, in-game messages, leaderboards, notifications, server-side combat resolution, and asynchronous multiplayer gameplay like card games, word games, and ghost racers. Cloud Canvas eliminates the need for you to acquire, configure, or operate host servers yourself, and reduces or eliminates the need to write server code for your connected gameplay features.

Features

Cloud Canvas offers a wide range of helpful components:

- 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

To see how Cloud Canvas uses AWS services in a sample project, see Don't Die Sample Project (p. 154). For a tutorial on Cloud Canvas, see Lumberyard Tutorials.

Tools

You can access Cloud Canvas functionality by using any of the following:

• **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. 193).
• **Cloud Canvas C++ APIs** – For software development.
• **Using the Cloud Canvas Command Line (p. 165)** – For managing resource groups, mappings, deployments, and projects.
• **Cloud Canvas Tools in Lumberyard Editor (p. 160)** – For managing AWS resources, deployments, and credentials, and for navigating directly to the AWS consoles supported by Cloud Canvas.

To see how AWS services used for the Don't Die sample project, see Don't Die Sample Project (p. 154).

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

• Pricing (p. 137)
• Cloud Canvas Core Concepts (p. 137)
• Cloud Canvas Resource Manager Overview (p. 141)
• Tutorial: Getting Started with Cloud Canvas (p. 143)
• Don't Die Sample Project (p. 154)
• Using Cloud Canvas (p. 160)
• Cloud Canvas Flow Graph Node Reference (p. 193)
• Resource Definitions (p. 216)
• Resource Deployments (p. 238)
• Resource Mappings (p. 240)
• Custom Resources (p. 242)
• Access Control and Player Identity (p. 246)
• AWS Client Configuration (p. 252)
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.

To see the AWS services used for the Don't Die sample project, see Don't Die Sample Project (p. 154).

Cloud Canvas Core Concepts

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
- How a team can collaborate on a game that has cloud-connected resource groups

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 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.

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. For more information, see Cloud Canvas Resource Manager Overview (p. 141).

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 Access Control and Player Identity (p. 246).

Cloud Canvas Resource Manager Overview

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.

![Diagram of resource management in CloudCanvas](image)

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 Access Control and Player Identity.

**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. 144).

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 in your account, creating an administrator group, and then adding the IAM user to the administrator group.

Create an IAM user

It's time to create your IAM 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 Create New Users.
4. Type a user name into box 1. This tutorial uses the name CloudCanvasAdmin.
5. Ensure that the Generate an access key for each user check box is checked.
6. Click Create. Your IAM user is created along with two important credentials: an access key and a secret access key. These credentials must be entered into Cloud Canvas in order to access the AWS resources in your project.
7. Click Show User Security Credentials to view your security credentials, or click Download Credentials to download them in a .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.

   **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.

8. Click Close. Next, you create a password for the user.
9. In the list of users, click the CloudCanvasAdmin user name (or whatever name you used).
10. Click the Security Credentials tab to create a password so that you can sign in as this user later.
11. Click Manage Password.
12. Click Assign an auto-generated password or Assign a custom password.
13. When you are finished, choose Apply.

You have now created an IAM user. Next, you create an administrator group to which you later add the CloudCanvasAdmin user.

Create a Group for Administrators

IAM groups simplify managing permissions for a number of users without having to manage each user individually.

To create a group for your Cloud Canvas administrative IAM user

1. In the left navigation pane of the IAM console, choose Groups.
2. Choose Create New Group.
3. Give the group a name, such as CloudCanvasAdministrators.
4. Click Next Step.
5. Select the check box next to the AdministratorAccess policy. 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.

6. Click **Next Step**.
7. Review the proposed group that you are about to create.
8. Click **Create Group**.

You have now created a group with administrator permissions.

Next, you add the CloudCanvasAdmin IAM user that you created earlier to the CloudCanvasAdministrators group that you just created.

**Add an IAM User to an Administrator Group**

Adding users to an administrator group is easier and more secure than attaching a policy with administrator permissions separately to each IAM user. In this tutorial, you add only one IAM user to the administrator group, but you can add more if required.

**To add the CloudCanvasAdmin IAM user to the CloudCanvasAdministrators group**

1. Click the name of the newly created CloudCanvasAdministrators group (not the check box).
2. In the **Users** tab, choose **Add Users to Group**.
3. Select the check box next to the CloudCanvasAdmin IAM user that you want to belong to the administrator group.
4. Click **Add Users**. The CloudCanvasAdmin user name now appears in the list of users for the CloudCanvasAdministrators group.
5. Sign out of the AWS Management Console so that you can sign in as your CloudCanvasAdmin user in the next step.

**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
   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.
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 use the SamplesProject, and how to enable the Cloud Canvas Gem package in a new project if you are not using the SamplesProject.

Cloud Canvas in the SamplesProject

The default SamplesProject is already configured to use the Cloud Canvas Gem package. If you are using the SamplesProject, no additional steps are needed. Go to Step 5: Add Administrator Credentials to Lumberyard (p. 147).

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. 147).
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. 144).

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. 144).

   ```bash
   lmbr_aws add-profile --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. If you are using SamplesProject and have checked Lumberyard into source control, ensure that the `<root>\SamplesProject\AWS\project-settings.json` file has been checked out and is writeable. If you are using a new project, this file will be created during the initialization process, along with other files in the project's AWS directory.
2. Open a command line window and change to your Lumberyard `dev` directory.
3. 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 (N. Virginia), which supports Amazon Cognito.

Type the following command:

```
lmbr_aws create-project-stack --region us-east-1
```

The command initializes the contents of the `<root>\<game>\AWS` directory 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.

4. You can see the resources created in your AWS account by typing the following command:

```
lmbr_aws list-resources
```

5. 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 locate the `DontDieAWS` resource group already defined for you by the SamplesProject. If you’re using a different project, it also shows you how to add a resource group and optionally add some example resources.
Locating the Resource Group Defined by SamplesProject

The SamplesProject defines a single resource group named "DontDieAWS". The resource definitions for this resource group are found in the
<root>\SamplesProject\AWS\resource-group\DontDieAWS\resource-template.json file. This file is an AWS CloudFormation template. It will be used to create the AWS resources required by SamplesProject in the next step of this tutorial.

You can see that the resource group is part of SamplesProject by typing the following at the command prompt, and then pressing Enter.

```bash
lmbr_aws list-resource-groups
```

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:

```bash
lmbr_aws add-resource-group --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:

```bash
lmbr_aws list-resource-groups
```

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. 179) provides details about the creation process.

7. To make the deployment the default, see **Making a Deployment the Default (p. 184)**.

**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 create-deployment --deployment TestDeployment
   ```

3. You can see that the deployment is now part of the Lumberyard project by typing the following command:
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. 146)).
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. 149). 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. 146)).
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. If you are using the SamplesProject, the name begins with SamplesProject-DontDieDeploymentAccess.
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. 185).

---

**Don't Die Sample Project**

This sample project shows how you can use the AWS Cloud Canvas Resource Management system and AWS Lambda for a game. Don’t Die also uses Project Configurator, which is a standalone application included with Lumberyard that you use to specify to the Waf Build system the game project and assets (Gems) that you want to include in a build.

AWS resources used in the Don’t Die project may be subject to separate charges and additional terms. There is a free tier for all the AWS services used in this project. See the end of this topic to learn more about AWS services used in this project.

### Setup

Setting up the sample project involves a few tasks.

### Creating the AWS Project Stack

All AWS resources associated with Don’t Die, such as DynamoDB tables, S3 buckets, and Lambda, are created through the Cloud Canvas Resource Management system. The cloud-connected features of Don’t Die, such as High Score and Daily Gift, will not work until you set up their associated AWS resources. Before you can set up the resources, you must:

- Have an AWS account with administrator permissions credentials.
- Make your AWS account available for use with Lumberyard Editor and the Cloud Canvas command line if you have not yet done so. To do this, perform one of the following:
  - Use the Lumberyard Editor Credentials Manager to add an AWS profile. For more information, see Managing Cloud Canvas Profiles (p. 163).
  - At a command line prompt on the Lumberyard \dev folder, type the following:
lmbr_aws add-profile --aws-access-key {accesskey} --aws-secret-key {secretkey} --profile {profilename} --make-default

Replace \{accesskey\} and \{secretkey\} keys with the corresponding keys from your AWS account. Replace \{profilename\} with a name of your choice for the profile. The --make-default argument optionally makes this set of credentials your default profile when you use the Cloud Canvas command line.

**Note**
When you use Credentials Manager, the profile that you specify becomes the default.

- Ensure that SamplesProject is selected in Project Configurator, which you can run from Lumberyard folder location \dev\Bin64\ProjectConfigurator.exe. This configuration points the AWS Resource Management System to \SamplesProject\AWS, which contains all the files, templates, and project code for creating the required AWS resources.
- In Project Configurator, ensure that the Cloud Canvas Gem and the Static Data Gem are enabled.
- Ensure that the SamplesProject\AWS\project-settings.json file is writable. During the resource setup process, the Amazon Resource Name (ARN) for the AWS CloudFormation stack is added into this file.
- Select the region where the AWS resources will live. *Don't Die* relies on a service called Amazon Cognito for player identity, which is currently only supported in the following regions: US East (N. Virginia) (us-east-1), EU (Ireland) (eu-west-1), and Asia Pacific (Tokyo) (ap-northeast-1). This example uses US East (N. Virginia).

### To create the project stack

1. From a command line prompt on the Lumberyard \dev folder, type the following (the example assumes that you are using the US East region):

```
lmbr_aws create-project-stack --region us-east-1
```

Creating the stack will take a few minutes.

Next, you will create a deployment, which is a complete set of your game's cloud resources. You can have separate deployments (p. 238) that correspond to the different stages of a release (for example, an internal deployment for developers, and a release deployment for players). To keep your project simple, you will create just one deployment.

2. From the same command line prompt, type the following to create a deployment called "DontDieDeployment":

```
lmbr_aws create-deployment --deployment DontDieDeployment
```

### Test the Game

At this point you can test the game and its AWS connected High Scores feature. You can run the game either from Lumberyard Editor or from the standalone SamplesProject launcher.

#### To run the game from Lumberyard Editor

1. Open Lumberyard Editor.
2. In the Welcome to the Lumberyard Editor dialog, click Open Level.
3. In the Open a Level dialog, choose Levels, Samples, Dont_Die, and then click Open.
4. To upload the resources for your game, do one of the following:

   • On the command line, type

     `lmbr_aws upload-resources`

   • In Lumberyard Editor, perform the following steps:

     1. Click AWS, Cloud Canvas, Cloud Canvas Resource Manager.

     2. In Resource Groups, click Upload all resources.
3. Exit Lumberyard Editor and restart it.

   **Note**
   This step and the next are workarounds for a bug that will be fixed in a later release.

4. Follow the previous steps to load the **Dont_Die** level in Lumberyard Editor.

5. To play the game in Lumberyard Editor, press `Ctrl-G`.

   Your high scores will be recorded in the cloud and will reappear if you play again.

You can also run the game from the SamplesProject standalone launcher, although a few more steps are required.

**To run the game from the standalone launcher**

1. In Lumberyard Editor, click **File, Export to Engine** or press `Ctrl-E` to export the level.

2. In Lumberyard Editor, click **AWS, Cloud Canvas, Cloud Canvas Resource Manager**.

3. In **Cloud Canvas Resource Manager** navigation tree, expand **Administration (advanced)**, and then select **project-settings.json**. You can use resource manager to edit the JSON text, or open the `dev\SamplesProject\AWS\project-settings.json` file and edit the file with a text editor.

4. Under the line

   "DefaultDeployment": "DontDieDeployment",

   add the line:

   "ReleaseDeployment": "DontDieDeployment",

5. Save the file.

6. At a command line prompt, run the command

   `lmbr_aws update-mappings --release`

7. Run the program `dev\Bin64\SamplesProjectLauncher.exe`.

8. Type `~` (the tilde character) to open the SamplesProject launcher console.

9. In the console, type

   `map dont_die`

   and then press **Enter**.

   The game opens directly in the SamplesProject launcher so that you can play it. Your high scores will be recorded in the cloud and will reappear if you play again.

### Viewing Lambda Code in Visual Studio

You can view the Lambda source code for the *Don't Die* project in Visual Studio 2013. However, you must install the following tools first:
Acquiring the Mappings File

All of the Lambda code in the Don't Die project refers to AWS resources by friendly names like "High Score Table." Lambda functions use a mappings file to translate friendly names to the physical names of the actual AWS resources in your account. This file is generated by the AWS Resource Management System but is only inserted in the Lambda function right before the system uploads the code to AWS. In order to run the Lambda functions from Visual Studio, copy the mappings file locally, as follows:

To acquire the mappings file
1. Open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. Filter for DontDie and click the linked name of one of the Lambda functions in the list (one function should contain the text DontDieMain). The detail page for the function appears.
3. Click Actions, and then click Download function code.
4. Download the .zip file to your computer.
5. Open the .zip file and extract the CloudCanvas directory together with the settings.js file in it to the \dev\SamplesProject\AWS\resource-group\DontDieAWS\lambda-function-code\ directory. This file enables Visual Studio to find the correct resources for the Lambda function.
6. To view the solution, open the dev\SamplesProject\AWS\SamplesProjectAWS.sln file in Visual Studio.

Lambda Code Overview

Within the Visual Studio project, under the apps folder, there is a folder called don't-die-main. This folder contains the dont-die-main Lambda function that is used throughout the game. The function executes commands such as start-game, end-game, and get-high-score-table that are sent to it from the client, and then it sends back the result. Commands can be batched together in an effort to minimize the number of calls to other AWS services. For example, if two commands that update the player table are sent together, the Lambda function combines the commands into a single DynamoDB update.

The don't-die-main folder contains three files: _sampleEvent.json, _testdriver.js, and app.js. The first two files are not uploaded to AWS (this is controlled by the .ignore file) but are used for local testing only. The _sampleEvent.json file contains test data that represents the data that the client sends. The _testdriver.js file executes the Lambda code locally, emulating the AWS service that calls the Lambda function. Both files are generated by the AWS toolkit, although they have been somewhat modified. The third file, app.js, contains the entry point for the exports.handler Lambda function in this sample.

To test the Lambda function locally, right-click the _testdriver.js file, select Set as Node.js Startup File, and click Debug > Start Debugging.

The general behavior of the Lambda function is as follows:

• Each time the dont-die-main Lambda function is executed, a DontDie object is created, and Start is called on it.
• The DontDie object waits for each game system in the systemModuleList array to initialize.
• If a game system has a set of static data that it relies on, it tries to load it. If AWS executes the Lambda function on a computer on which the Lambda function has been recently used, it uses the data already loaded from the previous execution.
• After all game systems are initialized, the commands sent by the client are executed. This step corresponds to the code in the app.js file.
• After all commands have executed serially, Finish is called on the DontDie object. This call gives each game system a chance to save any changes to state that have been made. For example, if player data has changed, it is saved to DynamoDB.

Deleting the AWS Project Stack

After ensuring that your Amazon S3 bucket is empty, you can delete your deployment, resources, and project stack.

Empty Your S3 Bucket

Don't Die creates one Amazon S3 bucket. Before you can delete the project stack, ensure this S3 bucket is empty. Otherwise, the AWS Resource Management system cannot delete the S3 bucket, which blocks it from deleting the project stack.

Note
The following steps are necessary only if you manually added files to the Don't Die mainbucket. If you didn’t add any files, you can skip this step.

To empty an S3 bucket

1. Sign in to the AWS Management Console and open the Amazon S3 console at https://console.aws.amazon.com/s3/.
2. Find the samplesproject bucket that has mainbucket somewhere in the title, and click that bucket.
3. Select each item in the bucket, and then click Actions, Delete.

After the Amazon S3 bucket is empty, you can delete the project stack.

Remove Your Deployment and its Resources

To remove your deployment, its resources, and your project stack, you can use either the Cloud Canvas Resource Manager or the Cloud Canvas command line to delete the DontDieDeployment. For steps, see Deleting Cloud Canvas Deployments and Their Resources (p. 185).

AWS Services Used

The Don't Die sample project implements game back-end features on AWS. By default, this project uses the services listed in the table below. When you use certain features in the sample project, you are using the AWS services that power them. You can add additional services by customizing the templates or writing your own templates.

When you initialize the Don't Die sample project, you are prompted to deploy AWS services to your account by using the included AWS CloudFormation templates.

There is no additional charge for using Cloud Canvas. AWS resources you use for Don't Die may be subject to separate charges and additional terms. You pay for AWS resources created using Cloud Canvas, such as Lambda functions, DynamoDB tables, and IAM in the same manner 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.
Using Cloud Canvas

Cloud Canvas provides a rich set of tools for Lumberyard Editor and for the command line that you can use to create and manage the cloud connected features of your game.

You can use these tools to define and manage the AWS resources in your Lumberyard project, including the following:

- **Initializing (p. 162)** a Lumberyard project with Cloud Canvas functionality
- **Managing the resource deployments (p. 238) and resource groups (p. 138) (connected features) for your project**
- **Updating the contents of your Lumberyard game project**
- **Previewing and editing (p.161) the AWS CloudFormation templates that define (p.216) the AWS resources for your game.**

**Topics**

- Cloud Canvas Tools in Lumberyard Editor (p. 160)
- Editing Cloud Canvas Files (p. 161)
- Initializing Cloud Canvas Resource Manager (p. 162)
- Managing Cloud Canvas Profiles (p. 163)
- Understanding Resource Status Descriptions (p. 165)
- Using the Cloud Canvas Command Line (p. 165)
- Viewing the Cloud Canvas Progress Log (p. 179)
- Working with Deployments (p. 180)
- Working with JSON Files (p. 186)
- Working with Project Stacks (p. 187)
- Working with Resource Groups (p. 188)

### Cloud Canvas Tools in Lumberyard Editor

Lumberyard Editor provides tools that make it easy for you to connect your game to AWS. To get started, click **AWS** in the Lumberyard Editor toolbar:
The AWS menu has the following options:

- **Credentials manager** – Select or manage one or more AWS profiles that provide credentials required to access your AWS account. For more information, see Managing Cloud Canvas Profiles (p. 163).
- **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.
- **Cloud Canvas** – The Cloud Canvas menu has 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. 183).
  - **Cloud Canvas Resource Manager** – Define and manage the AWS resources for your Lumberyard project. For a conceptual introduction to resource manager, see Cloud Canvas Resource Manager Overview (p. 141).
  - **Open an AWS Console** – Get quick access to the main AWS Management Console and to consoles for Amazon Cognito, DynamoDB, Amazon S3, and Lambda:

  These links use your currently active AWS profile to connect to AWS. You can use the Managing Cloud Canvas Profiles (p. 163) to select which profile is active.

## Editing Cloud Canvas 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. 186) 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:

![Open in script editor](image)

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.

### 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. 187) node is selected, and in the detail pane, the **Viewing the Cloud Canvas Progress Log** (p. 179) shows the progress of the initialization.

### 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.
  
  ![Credentials Manager](image)

- In Cloud Canvas Resource Manager, click the name of the current profile in the Resource Manager toolbar:

  ![Cloud Canvas Resource Manager](image)

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.

![Add Profile](image)

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 add-profile --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.

### 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 pending** – The resource is defined in the local configuration but doesn’t exist in AWS.
- **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 pending** – The resource is not defined in the local configuration but it does exist 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.

### 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.

**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. 167) 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-1.
- 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:

- --profile {profile} The AWS command line tool profile that is used.
- --aws-access-key {access-key} The AWS access key that is used.
- --aws-secret-key {secret-key} The AWS secret key that is used.

Command Summary

This topic discusses the following commands:

- add-login-provider (p. 167) – Add a player login provider to the Amazon Cognito identity pool configuration.
- add-profile (p. 168) – Add an AWS profile to the AWS command line tool configuration.
- add-resource-group (p. 168) – Add a group of related resources to the project.
- create-deployment (p. 169) – Create an independent copy of your project's resources.
- create-project-stack (p. 169) – Create the AWS resources needed for a Lumberyard project. If the {game}\AWS directory contains no resource definitions, default resource definitions are created.
- default-deployment (p. 170) – Show or set the default deployment.
- default-profile (p. 171) – Set, clear, or show the default profile from the AWS command line tool configuration.
- delete-deployment (p. 171) – Delete an independent copy of your project's resources.
- delete-project-stack (p. 171) – Delete a project stack. This command will not delete projects with deployments.
- import-resource (p. 172) – Import a resource to a resource group.
- list-deployments (p. 172) – List all deployments in the local project.
- list-importable-resources (p. 173) – List all supported resources currently existing on AWS.
- list-mappings (p. 173) – Show the logical to physical resource name mappings.
- list-profiles (p. 173) – List the AWS profiles that have been configured.
- list-resource-groups (p. 173) – List the resource groups for the project.
- list-resources (p. 174) – List all of the resources associated with the project.
Common Arguments

Most of the `lmbr_aws` commands accept the following arguments, in addition to their own individual arguments:

- `-h` or `--help` – Display help for the command.
- `--root-directory {root}` – Identifies the Lumberyard\dev directory. The default is the current working directory.
- `--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.
- `--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.

Commands

Following are details of the `lmbr_aws` commands.

add-login-provider

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 Access Control and Player Identity (p. 246).

In addition to the Common Arguments (p. 167), the add-login-provider subcommand 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 used 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 `lmbr_aws update-project-stack` after running this command so that the PlayerAccessIdentityPool Resource (p. 234) configuration is updated to reflect the change.

**add-profile**

Add an AWS profile to the AWS command line tool configuration.

In addition to the Common Arguments (p. 167), the `add-profile` subcommand 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** `{profilename}`
  Required. The name of the AWS profile to add.
• **--make-default**
  Optional. Make the new profile the default profile.

**add-resource-group**

Add a ResourceGroupConfiguration and AWS CloudFormation stack resources to your deployment-template.json file. The added resources will be similar to the HelloWorldConfiguration Resource (p. 227) and HelloWorld Resource (p. 227) in the example deployment-template.json (p. 225) file.

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. 167), the `add-resource-group` subcommand accepts the following argument:

• **--resource-group** `{resource-group-name}`
  Required. The name of the resource group to add.
• **--include-example-resources**
create-deployment

Create a complete and independent copy of all the resources needed by the Lumberyard project.

In addition to the Common Arguments (p. 167), the create-deployment subcommand accepts the following arguments:

* --deployment {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 parameter. Possible values include: CAPABILITY_IAM.

* --confirm-aws-usage
  Optional. Confirms that you know that the create-deployment command will create AWS resources for which you may be charged and that may perform actions that can affect permissions in your AWS account. If not specified, you are prompted for confirmation.

create-project-stack

Initialize Cloud Canvas resource management for a Lumberyard project. This includes creating a set of default Resource Definitions (p. 216) in the {root}/{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. 167), the create-project-stack subcommand 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
  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. Also disables the prompt for confirmation during the command's execution.

* --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 parameter. 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 option can be used only with the create-project-stack and list-importable-resources commands.

How create-project-stack works

1. The create-project-stack command creates the project's AWS CloudFormation stack using a bootstrap template that defines only the Configuration Bucket (p. 239) resource.
2. The project-template.json (p. 220) file and the zipped up contents of the project-code subdirectory (p. 234) are uploaded to the Configuration Bucket (p. 239).
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, create-project-stack creates the directory if necessary and copies the contents of the {root}\Tools\lmbr_aws\AWSResourceManager\default-project-content directory to that directory.
• create-project-stack 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 option to specify a different name for the project stack.
• create-project-stack fails if the {root}\(\{game\}\)AWS\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 update-project-stack command.

default-deployment

Set or show the default deployment.

In addition to the Common Arguments (p. 167), the default-deployment subcommand 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.

**default-profile**

Set, clear, or show the default profile in the AWS command line tool configuration.

In addition to the Common Arguments (p. 167), the `default-profile` subcommand 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.

**delete-deployment**

Delete a complete and independent copy of all the resources needed by the Lumberyard project.

In addition to the Common Arguments (p. 167), the `delete-deployment` subcommand accepts the following argument:

- `--deployment {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 parameter. 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.

**delete-project-stack**

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. 167), the `delete-project-stack` subcommand accepts the following argument:

- `--confirm-resource-deletion`

  Optional. Confirms with the user that this command will delete all the AWS resources used by the AWS CloudFormation resource manager to manage your game’s AWS resources. Also disables the default prompt for confirmation when the command is run.

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.

**import-resource**

Import a resource to a resource group.

In addition to the Common Arguments (p. 167), the `import-resource` subcommand accepts the following arguments:

- `--type {dynamodb,s3,lambda}`

  Optional. The type of the AWS resource to import.

- `--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}`

  Required. The name of the resource group to import.

- `--download`

  Optional. If specified, downloads the contents of the Amazon S3 bucket.

**list-deployments**

List all deployments in the local project.

Example output:

<table>
<thead>
<tr>
<th>Name</th>
<th>Status</th>
<th>Reason</th>
<th>Id</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnotherDeployment</td>
<td>CREATE_PENDING</td>
<td>Resource is defined in the local project template but does not exist in AWS.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Development</td>
<td>CREATE_COMPLETE</td>
<td>03/04/16 18:43:11</td>
<td>arn:aws:cloudformation:us-east-1:&lt;ACCOUNTID&gt;:stack/foo-hw-Development-ZDLXUB7FKR94/8e6492f0-e248-11e5-8e7e-50d5ca6e60ae</td>
<td></td>
</tr>
</tbody>
</table>
User Default Deployment: (none)
Project Default Deployment: Development
Release Deployment: (none)

**list-importable-resources**

List all supported resources currently existing on AWS.

In addition to the Common Arguments (p. 167), the `list-importable-resources` subcommand accepts the following arguments:

- `--type {dynamodb,s3,lambda}`
  
  Required. The type of the AWS resource to list.

- `--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 option can be used only with the `list-importable-resources` and `create-project-stack` commands.

**list-mappings**

Show the logical to physical resource name mappings.

Example output:

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloWorld.SayHello</td>
<td>AWS::Lambda::Function</td>
<td>foo-hw-Development-ZDLXUB7FKR94-HelloWo-SayHello-1FADMFNE5M1CO</td>
</tr>
<tr>
<td>PlayerAccessIdentityPool</td>
<td>Custom::CognitoIdentityPool</td>
<td>us-east-1:108f6d6a-f929-4212-9947-a03269b9582e</td>
</tr>
<tr>
<td>PlayerLoginIdentityPool</td>
<td>Custom::CognitoIdentityPool</td>
<td>us-east-1:3020e175-0ddd-4860-8dad-1db57162cbb2</td>
</tr>
<tr>
<td>ProjectPlayerAccessTokenExchangeHandler</td>
<td>AWS::Lambda::Function</td>
<td>foo-hw-</td>
</tr>
<tr>
<td>ProjectPlayerAccessTokenExchangeHandler-1BG6JJ94IZAUV</td>
<td>Configuration</td>
<td>&lt;ACCOUNTID&gt;</td>
</tr>
<tr>
<td>account_id</td>
<td>Configuration</td>
<td></td>
</tr>
<tr>
<td>region</td>
<td>Configuration</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**list-profiles**

List the AWS profiles that have been configured.

**list-resource-groups**

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. 167), the `list-resource-groups` subcommand accepts the following argument:

- `--deployment {deployment-name}`

Optional. The name of the deployment to list resource groups for. If not given, the default deployment is used.

Example output:

<table>
<thead>
<tr>
<th>Name</th>
<th>Status</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnotherResourceGroup</td>
<td>CREATE_PENDING</td>
<td>Resource is defined in the local deployment template but does not exist in AWS.</td>
</tr>
<tr>
<td>HelloWorld</td>
<td>CREATE_COMPLETE</td>
<td></td>
</tr>
</tbody>
</table>

```
```

`list-resources`

List all of the resources associated with the project.

In addition to the Common Arguments (p. 167), the `list-resources` subcommand accepts the following arguments:

- `--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` parameters.

- `--deployment {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}`

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.

Example output:

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Status</th>
<th>Timestamp</th>
<th>Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration</td>
<td>AWS::S3::Bucket</td>
<td>CREATE_COMPLETE</td>
<td>03/04/16 18:38:25</td>
<td>foo-hw-configuration-vxaq1g44s0ef</td>
</tr>
<tr>
<td>Development</td>
<td>AWS::CloudFormation::Stack</td>
<td>CREATE_COMPLETE</td>
<td>03/04/16 18:43:11</td>
<td>arn:aws:cloudformation:us-east-1:&lt;ACCOUNT ID&gt;:stack/foo-hw-Development-ZDLXUB7FKR94/8e6492f0-e238-11e5-8e7e-50d5ca6e60ae</td>
</tr>
</tbody>
</table>
```
Development.HelloWorld AWS::CloudFormation::Stack
Development.HelloWorld.Messages AWS::DynamoDB::Table
CREATE_COMPLETE 03/04/16 18:41:24 foo-hw-Development-2DLXUB7FKR94-HelloWorld-WSGZ15EUWX52-Messages-W8398CX6EB7C
Development.HelloWorld.PlayerAccess Custom::PlayerAccess
CREATE COMPLETE 03/04/16 18:42:54 CloudCanvas:PlayerAccess:foo-hw-Development-2DLXUB7FKR94-HelloWorld-WSGZ15EUWX52
Development.HelloWorld.SayHello AWS::Lambda::Function
CREATE COMPLETE 03/04/16 18:42:45 foo-hw-Development-2DLXUB7FKR94-HelloWorld-SayHello-1FADMFNE5MICO
Development.HelloWorld.SayHelloConfiguration Custom::LambdaConfiguration
CREATE COMPLETE 03/04/16 18:42:39 CloudCanvas:LambdaConfiguration:foo-hw-Development-2DLXUB7FKR94-HelloWorld-WSGZ15EUWX52:SayHello:6e3be3f1-933b-47b7-b3f6-21a045cbda7
Development.HelloWorldConfiguration Custom::ResourceGroupConfiguration
CREATE COMPLETE 03/04/16 18:40:39 CloudCanvas:DeploymentConfiguration:foo-hw-Development-2DLXUB7FKR94-HelloWorld
DevelopmentAccess AWS::CloudFormation::Stack
CREATE COMPLETE 03/04/16 18:44:58 arn:aws:cloudformation:us-east-1:<ACCOUNT ID>:stack/foo-hw-DevelopmentAccess-14RNG95501JMJ/f56ff7f0-e238-11e5-a77e-50d5cd148236
DevelopmentAccess.Owner AWS::IAM::Role
CREATE COMPLETE 03/04/16 18:44:38 foo-hw-DevelopmentAccess-14RNG95501JMJ-Owner-1H1MHLAZ0KElJ
DevelopmentAccess.OwnerPolicy AWS::IAM::ManagedPolicy
DevelopmentAccess.Player AWS::IAM::Role
CREATE COMPLETE 03/04/16 18:44:33 foo-hw-DevelopmentAccess-14RNG95501JMJ-Player-1JXYH5PP0434S
DevelopmentAccess.PlayerAccess Custom::PlayerAccess
CREATE COMPLETE 03/04/16 18:44:49 CloudCanvas:PlayerAccess:foo-hw-DevelopmentAccess-14RNG95501JMJ
DevelopmentAccess.PlayerAccessIdentityPool Custom::CognitoIdentityPool
CREATE COMPLETE 03/04/16 18:44:41 us-east-1:108f6d6a-f928-4212-9947-a032699582e
DevelopmentAccess.PlayerLoginIdentityPool Custom::CognitoIdentityPool
CREATE COMPLETE 03/04/16 18:44:43 us-east-1:3020e175-0ded-4860-8dad-1db57162bcb2
DevelopmentAccess.PlayerLoginRole AWS::IAM::Role
CREATE COMPLETE 03/04/16 18:44:33 foo-hw-DevelopmentAccess-14RNG95-PlayerLoginRole-70M8548KMBJL
DevelopmentConfiguration Custom::DeploymentConfiguration
CREATE COMPLETE 03/04/16 18:40:17 CloudCanvas:DeploymentConfiguration:foo-hw:Development
ProjectPlayerAccessTokenExchangeHandler AWS::Lambda::Function
CREATE COMPLETE 03/04/16 18:40:39 foo-hw-ProjectPlayerAccessTokenExchangeHandler-1BG6JJ84IZAUV
ProjectPlayerAccessTokenExchangeHandlerRole AWS::IAM::Role
CREATE COMPLETE 03/04/16 18:40:33 foo-hw-ProjectPlayerAccessTokenExchangeHandlerRole-ToE7MY1OB67N
ProjectResourceHandler AWS::Lambda::Function
CREATE COMPLETE 03/04/16 18:40:08 foo-hw-ProjectResourceHandler-XAP5CBAMQCYP
ProjectResourceHandlerExecution AWS::IAM::Role

Version 1.4

175
**remove-login-provider**

Remove a player login provider from the Amazon Cognito identity pool configuration.

In addition to the Common Arguments (p. 167), the `remove-login-provider` subcommand accepts the following argument:

- **--provider** `{provider-name}`
  
  Required. The name of the provider.

  The `remove-login-provider` 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 update-project-stack` after running this command so that the `PlayerAccessIdentityPool Resource (p. 234)` configuration is updated to reflect the change.

**remove-profile**

Remove an AWS profile from the AWS command line tool configuration.

In addition to the Common Arguments (p. 167), the `remove-profile` subcommand accepts the following argument:

- **--profile** `{profile-name}`
  
  Required. The name of the AWS profile to remove.

**remove-resource-group**

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. 167), the `remove-resource-group` subcommand accepts the following argument:

- **--resource-group** `{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.
rename-profile

Rename an AWS profile in the AWS command line tool configuration.

In addition to the Common Arguments (p. 167), the rename-profile subcommand 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.

update-login-provider

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 Access Control and Player Identity (p. 246).

In addition to the Common Arguments (p. 167), the update-login-provider subcommand 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 update-login-provider 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 update-project-stack` after running this command so that the PlayerAccessIdentityPool Resource (p. 234) configuration is updated to reflect the change.

update-mappings

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. 167), the `update-mappings` subcommand accepts the following argument:

• `--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. 234) 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` option is specified, the release deployment is queried.

update-profile

Update an AWS profile.

In addition to the Common Arguments (p. 167), the `update-profile` subcommand 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}`

Required. The name of the AWS profile to update.

Note
To make an existing profile the default profile, use the `default-profile (p. 171)` command.

update-project-stack

Update the project's AWS CloudFormation stack.

In addition to the Common Arguments (p. 167), the `update-project-stack` subcommand accepts the following arguments:

• `--confirm-aws-usage`

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. Also disables the prompt for confirmation done during the command's execution.
• --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 parameter. Possible values include CAPABILITY_IAM.

How update-project-stack works

1. The project-template.json (p. 220) file and the zipped up contents of the project-code subdirectory (p. 234)
are uploaded to the Configuration Bucket (p. 239).
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 update-project-stack command fails if the

(root)\{game\}\AWS\project-settings.json file does not exist or does not have a valid

ProjectStackId property.

upload-resources

Update a resource group's AWS CloudFormation stack in a specified deployment.

In addition to the Common Arguments (p. 167), the upload-resources subcommand accepts the following
arguments:

• --deployment

Required. The name of the deployment to update.

• --resource-group

Required. The name of the resource group to update.

• --confirm-aws-usage

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. It also
disables the default confirmation prompt that occurs during the command's execution.

• --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 parameter. Possible values include CAPABILITY_IAM.

The resource-template.json (p. 234) file and the zipped up contents of the lambda-function-code
subdirectory (p. 238) are uploaded to the Configuration Bucket (p. 239). An AWS CloudFormation stack
update operation is then performed by using the uploaded template file. The lambda-function-code.zip file is used when updating the Lambda function resources defined by the resource template.

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 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. 163).

**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. 179) 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. 188) in the Working with Project Stacks (p. 187) 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. 163).

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. 187) in the Working with Project Stacks (p. 187) 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. 185).

**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. 188) in the Working with Project Stacks (p. 187) section.

**Topics**
- Making a Cloud Canvas Deployment Active (p. 183)
- Deleting Cloud Canvas Deployments and Their Resources (p. 185)

**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. 188) node or an Individual Resource Group (p. 190) 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. 162) and Create Deployment (p. 180).

**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:
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:

```bash
lmbr_aws default-deployment --set <deployment name>
```
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 list-deployments
   ```

4. Type the following command for each of the deployments that you want to delete:

   ```
   lmbr_aws delete-deployment --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 delete-project-stack
```

This removes all AWS resources that are related to your Cloud Canvas project.

## 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 Cloud Canvas Files](#) (p. 161).

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 Access Control and Player Identity.

**user-settings.json**

The user-settings.json file contains user specific settings. For more information, see Resource Definitions.

**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. 163).
- 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. 162).

**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:
Status – The status of the AWS CloudFormation stack. See Understanding Resource Status Descriptions (p. 165) for a description of the values this column may have. To see additional status information, pause your mouse 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 mouse 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.

Stack Resources Table

The Stack resources table shows the status of the resources that your project is using.

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Type</th>
<th>Status</th>
<th>Timestamp</th>
<th>ID</th>
</tr>
</thead>
</table>
| Configuration | AWS::S3::Bucket | Create complete | Mon Jun 6 16:34:06 2016 | ...
| ProjectPlayerAccessTokenExchangeHandler | AWS::Lambda::Function | Create complete | Mon Jun 6 16:35:38 2016 | ...
| ProjectPlayerAccessTokenExchangeHandlerRole | AWS::IAM::Role | Create complete | Mon Jun 6 16:35:32 2016 | ...
| ProjectResourceHandler | AWS::Lambda::Function | Create complete | Mon Jun 6 16:35:42 2016 | ...
| ProjectResourceHandlerExecution | AWS::IAM::Role | Create complete | Mon Jun 6 16:35:37 2016 | ...

This table has the following columns:

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. 165). To see additional status information, pause your mouse 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 mouse on the truncated ID.

Working with Resource Groups

When you select Resource Groups in the Cloud Canvas Resource Manager Cloud Canvas configuration navigation pane, the detail pane shows the status of the resource groups that belong to the current deployment of your project. 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. 163).
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. 183).*

## Resource Groups

The **Resource Groups** detail pane lists the resource groups in your current deployment:

![Resource Groups detail pane](image)

The **Resource Groups** detail pane has the following options:

### Upload all resources

The **Upload all resources** option starts the process of modifying your resources in AWS as needed to match all of the definitions in all of your local resource groups. As the update proceeds, resource groups with the status of **Create pending** will change to **Create complete**. The update might take a few minutes.

![Upload all resources](image)

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 the administrator of the AWS account that you use. For more information, see *Initializing Cloud Canvas Resource Manager (p. 162).*

- After you have initialized the project, Lumberyard prompts you to create a deployment for it. A deployment creates all the AWS resources specified by your resource group definition. For more information, see *Create Deployment (p. 180).*

For information about the **Progress log**, see *Viewing the Cloud Canvas Progress Log (p. 179).*

### Add resource group

Use the **Add resource group** option to add 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**

The **Resource group status** table 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. 188) in the **Working with Project Stacks** (p. 187) section.

**Individual Resource Group**

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.

Note the following:

- If you select a resource group 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. 163).

- If you select a resource group 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. 183).

**Adding Resources in 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:
The definitions 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. The following image shows the status details for the **DontDieAWS** resource group:

The status pane for a resource group has the following options:
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 specified by the local resource definitions that you created with the **Add resource** option.

As the update proceeds, the resources with the status of **Create pending** will change to **Create complete**.

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 the administrator of the AWS account that you use. For more information, see **Initializing Cloud Canvas Resource Manager** (p. 162).
- If you do not yet have a deployment for your project, Lumberyard prompts you to create one. A deployment creates all the AWS resources specified by your resource group definition. For more information, see **Create Deployment** (p. 180).

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 using the **Remove resource group** option) 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:

![Delete group resources](image)

After you click **Yes**, the deletion operation may 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. 188) in the Working with Project Stacks (p. 187) section.

**resource-template.json**

For information about this node, see Working with JSON Files (p. 186).

**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 defined by your resource group. For more information, see `lambda-function-code` Directory. Also see related information for the **project-code** (p. 193) 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 Cloud Canvas Files (p. 161).

## 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. 194)
- Cognito (Player Identity) Nodes (p. 197)
- DynamoDB (Database) Nodes (p. 198)
- Lambda (Cloud Functions) Node (p. 205)
- S3 (Storage) Nodes (p. 206)
- SNS (Notification Service) Nodes (p. 208)
- SQS (Message Queuing Service) Nodes (p. 211)
- Static Data (PROTOTYPE) Nodes (p. 212)

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. 194)
- SetConfigurationVariable node (p. 194)
- ConfigureProxy node (p. 195)
- GetConfigurationVariableValue node (p. 195)
- SetDefaultRegion node (p. 196)

ApplyConfiguration node

Applies AWS configuration to all managed clients.

Inputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apply</td>
<td>Any</td>
<td>Applies the current AWS configuration to all managed clients</td>
</tr>
</tbody>
</table>

Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
</tbody>
</table>

SetConfigurationVariable node

Sets a configuration parameter value.

Inputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configure</td>
<td>Any</td>
<td>Sets the parameter value</td>
</tr>
<tr>
<td>Name</td>
<td>String</td>
<td>Name of the parameter to set</td>
</tr>
<tr>
<td>Value</td>
<td>String</td>
<td>Value to which the parameter will be set; may contain $param-name$ substrings</td>
</tr>
</tbody>
</table>
Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
</tbody>
</table>

**ConfigureProxy node**

Sets the proxy configuration used by all AWS clients.

```plaintext
ConfigureProxy
- Configure
- Host= [String]
- Port= [Integer]
- UserName= [String]
- Password= [String]
```

Inputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configure</td>
<td>Any</td>
<td>Sets the proxy configuration</td>
</tr>
<tr>
<td>Host</td>
<td>String</td>
<td>Proxy host</td>
</tr>
<tr>
<td>Port</td>
<td>Integer</td>
<td>Proxy port</td>
</tr>
<tr>
<td>UserName</td>
<td>String</td>
<td>Proxy user name</td>
</tr>
<tr>
<td>Password</td>
<td>String</td>
<td>Proxy password</td>
</tr>
</tbody>
</table>

Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
</tbody>
</table>

**GetConfigurationVariableValue node**

Inserts configuration value parameters into a string.

```plaintext
GetConfigurationVariableValue
- Expand
- Value= [String]
```

Version 1.4

195
### Inputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expand</td>
<td>Any</td>
<td>Expands parameter references</td>
</tr>
<tr>
<td>Value</td>
<td>String</td>
<td>Value containing <code>$param-name$</code> substrings</td>
</tr>
</tbody>
</table>

### Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
<tr>
<td>Value</td>
<td>String</td>
<td>Value with <code>$param-name$</code> substring replaced by parameter values</td>
</tr>
</tbody>
</table>

### SetDefaultRegion Node

Sets (overrides) the region for all AWS clients in the current project.

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. 194)** 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. 197)
• ConfigureAuthenticatedPlayer node (p. 197)

ConfigureAnonymousPlayer node

Creates an anonymous identity on the device in your AWS account.

### Inputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configure</td>
<td>Any</td>
<td>Configure your game to use Amazon Cognito for anonymous players</td>
</tr>
<tr>
<td>AWSAccountNumber</td>
<td>String</td>
<td>Your AWS account number. This is needed to access Amazon Cognito.</td>
</tr>
<tr>
<td>IdentityPoolID</td>
<td>String</td>
<td>The unique ID of your Amazon Cognito identity pool. To create an identity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pool ID, sign in to the AWS Management Console and use the Amazon Cognito</td>
</tr>
<tr>
<td></td>
<td></td>
<td>console at <a href="https://console.aws.amazon.com/cognito/">https://console.aws.amazon.com/cognito/</a>.</td>
</tr>
<tr>
<td>CachingFileLocationOverride</td>
<td>String</td>
<td>If specified, causes the Amazon Cognito ID to be cached to the path</td>
</tr>
<tr>
<td></td>
<td></td>
<td>specified instead of to <code>&lt;HOME_DIR&gt;/aws/.identities</code>.</td>
</tr>
</tbody>
</table>

### Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>error message</td>
</tr>
<tr>
<td>CognitoIdentityID</td>
<td>String</td>
<td>The unique ID of the user</td>
</tr>
</tbody>
</table>

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.
### DynamoDB (Database) Nodes

You can use these flow graph nodes to connect your game to Amazon DynamoDB.

**Topics**
- AtomicAdd node (p. 199)
- DeleteItem node (p. 199)
- GetItem node (p. 200)
- PutItem node (p. 201)
- Query node (p. 202)
- ScanTable node (p. 203)
- UpdateItem node (p. 203)
- GetStringSet node (p. 204)
**AtomicAdd node**

Add a number to an attribute in DynamoDB and return the number.

### Inputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Any</td>
<td>Writes the <code>val</code> specified in the <code>Value</code> port to DynamoDB</td>
</tr>
<tr>
<td>TableName</td>
<td>String</td>
<td>The name of the DynamoDB table to which to write</td>
</tr>
<tr>
<td>TableKeyName</td>
<td>String</td>
<td>The key name used in the table</td>
</tr>
<tr>
<td>Key</td>
<td>String</td>
<td>Specifies the key to which to write</td>
</tr>
<tr>
<td>Attribute</td>
<td>String</td>
<td>Specifies the attribute to which to write</td>
</tr>
<tr>
<td>Value</td>
<td>Integer</td>
<td>Specifies the value to write</td>
</tr>
</tbody>
</table>

### Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
<tr>
<td>NewValue</td>
<td>String</td>
<td>The value of the attribute after the addition.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeleteItem</td>
<td>Any</td>
<td>Deletes the specified item from DynamoDB.</td>
</tr>
<tr>
<td>TableName</td>
<td>String</td>
<td>The name of the DynamoDB table from which to delete</td>
</tr>
<tr>
<td>TableKeyName</td>
<td>String</td>
<td>The key name used in the table</td>
</tr>
<tr>
<td>KeyValue</td>
<td>String</td>
<td>Specifies the key to delete</td>
</tr>
</tbody>
</table>

### Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
<tr>
<td>DeletedItems</td>
<td>Any</td>
<td>Activated when matches were found to delete</td>
</tr>
<tr>
<td>NoResults</td>
<td>Any</td>
<td>No matching results were found</td>
</tr>
</tbody>
</table>

### GetItem node

Gets values from DynamoDB.

**Cloud Canvas (AWS): AWS Services: DynamoDB (Database): GetItem**

- GetItem
- TableName
- TableKeyName
- KeyValue
- AttributeToReturn

**Inputs**

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetItem</td>
<td>Any</td>
<td>Retrieves the item specified from DynamoDB</td>
</tr>
<tr>
<td>TableName</td>
<td>String</td>
<td>The name of the DynamoDB table from which to read</td>
</tr>
<tr>
<td>TableKeyName</td>
<td>String</td>
<td>The key name used in the table</td>
</tr>
<tr>
<td>KeyValue</td>
<td>String</td>
<td>Specifies the key to read</td>
</tr>
<tr>
<td>AttributeToReturn</td>
<td>String</td>
<td>Specifies the attribute to read</td>
</tr>
</tbody>
</table>
## Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
<tr>
<td>DataOut</td>
<td>String</td>
<td>String data that was read from DynamoDB</td>
</tr>
<tr>
<td>NumberOut</td>
<td>String</td>
<td>Number data that was read from DynamoDB</td>
</tr>
<tr>
<td>BoolOut</td>
<td>String</td>
<td>Boolean value that was read from DynamoDB</td>
</tr>
<tr>
<td>NoResults</td>
<td>Any</td>
<td>No matching results were found for the table, key, and attribute specified</td>
</tr>
</tbody>
</table>

## PutItem node

Writes values to DynamoDB.

```
Cloud Canvas (AWS):AWS Services:DynamoDB (Database):PutItem
• PutItem
• TableName=
• TableKeyName= hk
• KeyValue=
• AttributeToWrite=
• DataIn=
• DataType=string
• KeyMustNotExist=1
```

## Inputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PutItem</td>
<td>Any</td>
<td>Writes the item specified to DynamoDB</td>
</tr>
<tr>
<td>TableName</td>
<td>String</td>
<td>The name of the DynamoDB table to which to write</td>
</tr>
<tr>
<td>TableKeyName</td>
<td>String</td>
<td>The key name used in the table</td>
</tr>
<tr>
<td>KeyValue</td>
<td>String</td>
<td>Specifies the key to write</td>
</tr>
<tr>
<td>AttributeToWrite</td>
<td>String</td>
<td>Specifies the attribute to write</td>
</tr>
<tr>
<td>DataIn</td>
<td>String</td>
<td>The data to write</td>
</tr>
<tr>
<td>DataType</td>
<td>String</td>
<td>The data type that the data will be written as</td>
</tr>
<tr>
<td>KeyMustNotExist</td>
<td>Boolean</td>
<td>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.</td>
</tr>
</tbody>
</table>
## Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>of the port is the error message</td>
</tr>
<tr>
<td>KeyAlreadyExists</td>
<td>Any</td>
<td>The key already exists; no change was made</td>
</tr>
</tbody>
</table>

## Query node

Queries values in DynamoDB.

### Inputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query</td>
<td>Any</td>
<td>Queries table data in DynamoDB</td>
</tr>
<tr>
<td>TableName</td>
<td>String</td>
<td>The name of the DynamoDB table to query</td>
</tr>
<tr>
<td>TableKeyName</td>
<td>String</td>
<td>The name of the table key to query</td>
</tr>
<tr>
<td>KeyValue</td>
<td>String</td>
<td>The value of the key to query</td>
</tr>
<tr>
<td>AttributeToCheck</td>
<td>String</td>
<td>The attribute to query</td>
</tr>
<tr>
<td>AttributeComparison-</td>
<td>String</td>
<td>The comparison type to make against the attribute;</td>
</tr>
<tr>
<td>Type</td>
<td></td>
<td>the default is EQUALS. Other possible values are</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GREATER_THAN, GREATER_THAN_OR_EQUALS, LESS_THAN,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LESS_THAN_OR_EQUALS.</td>
</tr>
<tr>
<td>AttributeComparison-</td>
<td>String</td>
<td>The value to compare against the attribute</td>
</tr>
<tr>
<td>Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AttributeComparison-</td>
<td>String</td>
<td>The data type of AttributeComparisonValue (string, bool, or number); the default is string</td>
</tr>
<tr>
<td>ValueType</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>of the port is the error message</td>
</tr>
</tbody>
</table>
ScanTable node

Scans for entries which pass a comparison test in DynamoDB.

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatchFound</td>
<td>Any</td>
<td>A match was found</td>
</tr>
<tr>
<td>NoMatch</td>
<td>Any</td>
<td>No match was found</td>
</tr>
</tbody>
</table>

### Inputs

```
Scan
TableName=
Attribute=
AttributeComparisonType=EQUALS
AttributeComparisonValue=
AttributeComparisonValueType=string
```

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan</td>
<td>Any</td>
<td>Scans for matches in DynamoDB table data using the specified attributes</td>
</tr>
<tr>
<td>TableName</td>
<td>String</td>
<td>The name of the DynamoDB table to scan</td>
</tr>
<tr>
<td>Attribute</td>
<td>String</td>
<td>The attribute to query for</td>
</tr>
<tr>
<td>AttributeComparisonType</td>
<td>String</td>
<td>The comparison type to make against the attribute; this defaults to EQUALS.</td>
</tr>
<tr>
<td>AttributeComparisonValue</td>
<td>String</td>
<td>The value to compare against the attribute</td>
</tr>
<tr>
<td>AttributeComparisonValueType</td>
<td>String</td>
<td>The data type of AttributeComparisonValue (string, bool, or number); the default is string</td>
</tr>
</tbody>
</table>

### Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
<tr>
<td>Matches-Found</td>
<td>Any</td>
<td>The number of matches found on a successful scan</td>
</tr>
</tbody>
</table>

UpdateItem node

Updates attribute values of an existing item in DynamoDB.
## Inputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UpdateItem</td>
<td>Any</td>
<td>Updates an item in DynamoDB</td>
</tr>
<tr>
<td>TableName</td>
<td>String</td>
<td>The name of the DynamoDB table to use</td>
</tr>
<tr>
<td>TableKeyName</td>
<td>String</td>
<td>The name of the key in the table</td>
</tr>
<tr>
<td>KeyValue</td>
<td>String</td>
<td>The value of the key to write</td>
</tr>
<tr>
<td>AttributeToWrite</td>
<td>String</td>
<td>The attribute to write to</td>
</tr>
<tr>
<td>DataIn</td>
<td>String</td>
<td>The data to write</td>
</tr>
<tr>
<td>DataType</td>
<td>String</td>
<td>The data type to write the data as</td>
</tr>
<tr>
<td>KeyMustExist</td>
<td>Boolean</td>
<td>True if the key specified must already exist in the table; the default is true</td>
</tr>
<tr>
<td>AttributeMustExist</td>
<td>Boolean</td>
<td>True if the attribute must exist for the key specified; the default is true</td>
</tr>
</tbody>
</table>

## Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
<tr>
<td>ConditionsFailed</td>
<td>Any</td>
<td>Key or attribute not found (either the KeyMustExist or AttributeMustExist condition failed)</td>
</tr>
</tbody>
</table>

### GetStringSet node

Retrieves the members of a string set.
### Lambda (Cloud Functions) Node

You can use this flow graph node to invoke AWS Lambda functions.

#### Invoke node

![Invoke node diagram]

**Inputs**

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invoke</td>
<td>Any</td>
<td>Invokes a Lambda function, optionally providing JSON data as arguments through the <code>Args</code> port. For more information, see <a href="https://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html">AWS Lambda Invoke Request Syntax</a>.</td>
</tr>
<tr>
<td>FunctionName</td>
<td>String</td>
<td>The name of the Lambda function to call</td>
</tr>
<tr>
<td>Args</td>
<td>String</td>
<td>The input data that will be sent to the Lambda function call as arguments in JSON format. For more information, see <a href="https://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html">AWS Lambda Invoke Request Syntax</a>.</td>
</tr>
</tbody>
</table>
Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
<tr>
<td>Result</td>
<td>String</td>
<td>The data that was output by the Lambda function if no error occurred</td>
</tr>
</tbody>
</table>

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. 206)
- UploadFile node (p. 207)
- GeneratePublicUrl node (p. 207)

DownloadFile node

Downloads a file from Amazon S3.

Inputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DownloadFile</td>
<td>Any</td>
<td>Reads file data from an Amazon S3 bucket</td>
</tr>
<tr>
<td>BucketName</td>
<td>String</td>
<td>The name of the Amazon S3 bucket to use</td>
</tr>
<tr>
<td>KeyName</td>
<td>String</td>
<td>The name of the file to download from Amazon S3</td>
</tr>
<tr>
<td>FileName</td>
<td>String</td>
<td>The filename to use for the downloaded object</td>
</tr>
</tbody>
</table>

Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
</tbody>
</table>
UploadFile node

Uploads a file to Amazon S3.

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UploadFile</td>
<td>Any</td>
<td>Uploads a file to an Amazon S3 bucket</td>
</tr>
<tr>
<td>BucketName</td>
<td>String</td>
<td>The name of the Amazon S3 bucket to use</td>
</tr>
<tr>
<td>KeyName</td>
<td>String</td>
<td>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.</td>
</tr>
<tr>
<td>ContentType</td>
<td>String</td>
<td>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.</td>
</tr>
<tr>
<td>FileName</td>
<td>String</td>
<td>The name of the file to upload</td>
</tr>
</tbody>
</table>

Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
</tbody>
</table>

GeneratePublicUrl node

Generates a presigned URL that points to an Amazon S3 location that you specify.
## Inputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PresignUrl</td>
<td>Any</td>
<td>Generates a presigned URL for the Amazon S3 location specified</td>
</tr>
<tr>
<td>BucketName</td>
<td>String</td>
<td>The name of the Amazon S3 bucket to use</td>
</tr>
<tr>
<td>KeyName</td>
<td>String</td>
<td>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.</td>
</tr>
<tr>
<td>Http Request Method</td>
<td>String</td>
<td>The HTTP method against which to presign (DELETE, GET, POST, or PUT)</td>
</tr>
</tbody>
</table>

## Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
<tr>
<td>Url</td>
<td>String</td>
<td>The signed URL</td>
</tr>
</tbody>
</table>

## SNS (Notification Service) Nodes

You can use these flow graph nodes to process Amazon Simple Notification Service (Amazon SNS) messages.

### Topics
- ParseMessage node (p. 208)
- Notify node (p. 209)
- CheckArnSubscribed node (p. 209)
- SubscribeToTopic node (p. 210)

### ParseMessage node

You can use the ParseMessage node to extract the subject and body text in JSON format from an Amazon SNS message.

#### Inputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>Any</td>
<td>Extract the subject and body text in JSON format from an Amazon SNS message</td>
</tr>
<tr>
<td>Message</td>
<td>String</td>
<td>The JSON message to deserialize.</td>
</tr>
</tbody>
</table>
Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
<tr>
<td>Body</td>
<td>String</td>
<td>The message body</td>
</tr>
<tr>
<td>Subject</td>
<td>String</td>
<td>The message subject</td>
</tr>
</tbody>
</table>

Notify node

Publishes messages to an Amazon SNS topic.

Inputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notify</td>
<td>Any</td>
<td>Sends a notification to an Amazon SNS topic</td>
</tr>
<tr>
<td>Message</td>
<td>String</td>
<td>The message to send</td>
</tr>
<tr>
<td>Subject</td>
<td>String</td>
<td>The subject of the message</td>
</tr>
<tr>
<td>TopicARN</td>
<td>String</td>
<td>The Amazon Resource Name for your Amazon SNS topic</td>
</tr>
</tbody>
</table>

Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
</tbody>
</table>

CheckArnSubscribed node

Checks if an ARN is subscribed to an Amazon SNS topic.
Inputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check</td>
<td>Any</td>
<td>Checks if an ARN is subscribed to an Amazon SNS topic</td>
</tr>
<tr>
<td>TopicARN</td>
<td>String</td>
<td>The Amazon SNS topic ARN to check</td>
</tr>
<tr>
<td>Endpoint</td>
<td>String</td>
<td>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.</td>
</tr>
</tbody>
</table>

Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
<tr>
<td>True</td>
<td>Any</td>
<td>The ARN is subscribed to the Amazon SNS topic</td>
</tr>
<tr>
<td>False</td>
<td>Any</td>
<td>The ARN is not subscribed to the Amazon SNS topic</td>
</tr>
</tbody>
</table>

SubscribeToTopic node

Subscribes to an Amazon SNS topic.

```
Cloud Canvas (AWS):AWS Services:SNS (Notification service):SubscribeToTopic
  * Subscribe
  * Protocol
  * TopicARN
  * Endpoint
```

Inputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscribe</td>
<td>Any</td>
<td>Subscribes to a topic to receive messages published to that topic. For more information, see Subscribe to a Topic.</td>
</tr>
<tr>
<td>Protocol</td>
<td>String</td>
<td>The protocol of the endpoint to which to subscribe</td>
</tr>
<tr>
<td>TopicARN</td>
<td>String</td>
<td>The ARN of the Amazon SNS topic to which to subscribe</td>
</tr>
<tr>
<td>Endpoint</td>
<td>String</td>
<td>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.</td>
</tr>
</tbody>
</table>
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. 211)
- Push node (p. 211)

PollAndNotify node

Inputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>Any</td>
<td>Start polling an AWS queue</td>
</tr>
<tr>
<td>QueueName</td>
<td>String</td>
<td>The name of an AWS queue that has already been created</td>
</tr>
</tbody>
</table>

Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
<tr>
<td>OnMessageReceived</td>
<td>String</td>
<td>The most recent message on the stack</td>
</tr>
<tr>
<td>QueueArn</td>
<td>String</td>
<td>The ARN (Amazon Resource Name) of the queue</td>
</tr>
</tbody>
</table>

Push node

Pushes a message to an AWS queue
**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 at will and/or monitor them at regular intervals for changes.

**Topics**
- Add Monitored Bucket node (p. 212)
- Get Static Data node (p. 213)
- Load Static Data node (p. 214)
- Remove Monitored Bucket node (p. 214)
- Request Bucket node (p. 215)
- Set Update Frequency node (p. 215)

**Add Monitored Bucket node**

Adds an Amazon S3 bucket to monitor.

**Inputs**

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddBucket</td>
<td>Void</td>
<td>Adds a bucket to watch for updates</td>
</tr>
</tbody>
</table>
## Get Static Data node

Retrieves a field from a static data definition.

### Inputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get</td>
<td>Void</td>
<td>Retrieves a value from static data</td>
</tr>
<tr>
<td>StaticDataType</td>
<td>String</td>
<td>The type of the static data to retrieve</td>
</tr>
<tr>
<td>StaticDataId</td>
<td>String</td>
<td>The identifier for the static data definition in the table</td>
</tr>
<tr>
<td>StaticDataField</td>
<td>String</td>
<td>The field name of the data to retrieve</td>
</tr>
<tr>
<td>ActivateOnUpdate</td>
<td>Void</td>
<td>Fire the node again the next time an update of the data takes place</td>
</tr>
</tbody>
</table>

### Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
<tr>
<td>StringOut</td>
<td>String</td>
<td>The output of a string field</td>
</tr>
<tr>
<td>NumberOut</td>
<td>Integer</td>
<td>The output of a numeric field</td>
</tr>
<tr>
<td>BoolOut</td>
<td>Boolean</td>
<td>The output of a Boolean</td>
</tr>
<tr>
<td>Port</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------</td>
<td>------------------------------------------------------</td>
</tr>
<tr>
<td>FloatOut</td>
<td>Integer</td>
<td>The output of a floating point numeric field</td>
</tr>
</tbody>
</table>

**Load Static Data node**

Attempts to load static data of the type specified.

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Any</td>
<td>Load a type of static data</td>
</tr>
<tr>
<td>StaticDataType</td>
<td>String</td>
<td>The type of static data to load</td>
</tr>
</tbody>
</table>

**Outputs**

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
<tr>
<td>Finished</td>
<td>String</td>
<td>Finished attempting to load</td>
</tr>
</tbody>
</table>

**Remove Monitored Bucket node**

Removes a bucket name from the list of monitored buckets.

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove</td>
<td>Any</td>
<td>Removes a bucket from the list of monitored buckets</td>
</tr>
<tr>
<td>BucketName</td>
<td>String</td>
<td>The name of the bucket to remove</td>
</tr>
</tbody>
</table>
Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
<tr>
<td>Finished</td>
<td>String</td>
<td>Finished removing the bucket</td>
</tr>
</tbody>
</table>

Request Bucket node

Requests an update of a specific bucket, or of all monitored buckets.

Inputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RequestBucket</td>
<td>Any</td>
<td>Requests an update of a specific bucket or of all monitored buckets</td>
</tr>
<tr>
<td>BucketName</td>
<td>String</td>
<td>The name of the bucket for which to request an update. To request updates for all buckets, leave this value blank.</td>
</tr>
</tbody>
</table>

Outputs

<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>Any</td>
<td>Activated upon a successful operation</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Activated upon an error being detected; the value of the port is the error message</td>
</tr>
<tr>
<td>Finished</td>
<td>String</td>
<td>Finished sending the request</td>
</tr>
</tbody>
</table>

Set Update Frequency node

Sets or clears a recurring timer to poll monitored buckets.
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 `{root}\{game}\AWS` directory, where `{root}` is the Lumberyard installation `\dev` subdirectory and `{game}` is the directory identified by the `sys_game_folder` property in the `{root}\bootstrap.cfg` file. For example, if your game is the SamplesProject, 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 create-project-stack` (p. 169) command.

In addition, some user-specific configuration data is kept in the `{root}\Cache\{game}\pc\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.

```
{root}\{game}\AWS\  
  project-settings.json  
  project-template.json  
  deployment-template.json  
  deployment-access-template.json  
  project-code\  
    (Lambda function Code)
```
Each of these .json files is described in the following sections.

**project-settings.json**

The `project-settings.json` file contains project configuration data. The structure of this file is as follows:

```json
{
  "key": "value",
  "deployment": {
    "key": "value",
    "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.

An example `project-settings.json` settings file follows.

```json
{
  "ProjectStackId": "arn:aws:cloudformation:us-west-2:...",
  "DefaultDeployment": "Development",
  "ReleaseDeployment": "Release",
  "deployment": {
    "": {
      "resource-group": {
        "HelloWorld": {
          "parameter": {
            "WriteCapacityUnits": 1,
            "ReadCapacityUnits": 1,
            "Greeting": "Hi"
          }
        }
      }
    }
  }
}
```

DeploymentStackId Property

The DeploymentStackId property identifies the AWS CloudFormation stack for a deployment. The project's resource groups are children of these stacks. For more information, see Resource Deployments (p. 238).

The DeploymentStackId property is set by the create-deployment (p. 169) 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 `create-deployment (p. 169)` 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.

```json
{
  ...
  "parameter": {
    "{template-parameter-name-1}": {template-parameter-value-1},
    ...
    "{template-parameter-name-n}": {template-parameter-value-n}
  }
  ...
}
```

**user-settings.json**

The `user-settings.json` file contains user-specific configuration data.

**File Location**

The `user-settings.json` file is found at `{root}\Cache\{game}\pc\user\AWS\user-settings.json`. It is not in the `{root}\{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.

An example `user-settings.json` file follows.

```json
{
  "DefaultDeployment": "Test",
  "Mappings": {
    "HelloWorld.SayHello": {
      "ResourceType": "AWS::Lambda::Function",
      "PhysicalResourceId": "MyGame-Test-xxxxxxxxxxxxx-HelloWorld-yyyyyyyyyyyy-SayHello-zzzzzzzzzzzz"
    }
  }
}
```

**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` file overrides the property from the `project-settings.json` file. The project and user defaults can be set using the `lmbr_aws default-deployment (p. 170)` command.
Mappings Property

The Mappings property specifies the mapping of friendly names used in Lumberyard Editor to actual resource names. For example, the DailyGiftTable DynamoDB table would get mapped to a name like SamplesProject-DontDieDeployment-7BA1D0NO4N-DontDieAWS-11126XY7KU7F-DailyGiftTable-1G4G33K16D8ZS.

This property is updated automatically when the default deployment changes or when the default deployment is updated. It can be refreshed manually by using the `lmbr_aws update-mappings` command.

**project-template.json**

The `project-template.json` file is an AWS CloudFormation template that defines resources that support the Cloud Canvas resource management system.

An example `project-template.json` file follows.

```json
{
  "AWSTemplateFormatVersion": "2010-09-09",

  "Metadata": {
    "CloudCanvas": {
      "Id": "$Revision: #6 $"
    }
  },

  "Parameters": {
    "ConfigurationKey": {
      "Type": "String",
      "Description": "Location in the configuration bucket of configuration data."
    }
  },

  "Resources": {

    "Configuration": {
      "Type": "AWS::S3::Bucket",
      "DeletionPolicy": "Retain",
      "Properties": {
        "VersioningConfiguration": {
          "Status": "Enabled"
        },
        "LifecycleConfiguration": {
          "Rules": [
            {
              "Id": "DeleteOldVersions",
              "NoncurrentVersionExpirationInDays": "2",
              "Status": "Enabled"
            },
            {
              "Id": "DeleteUploads",
              "Prefix": "uploads",
              "ExpirationInDays": "2",
              "Status": "Enabled"
            }
          ]
        }
      }
    }
  }
}
```
"ProjectPlayerAccessTokenExchangeHandlerRole": {
  "Type": "AWS::IAM::Role",
  "Properties": {
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Action": "sts:AssumeRole",
          "Principal": {
            "Service": "lambda.amazonaws.com"
          }
        }
      ]
    }
  },
  "Policies": [
    {
      "PolicyName": "PlayerAccessTokenExchange",
      "PolicyDocument": {
        "Version": "2012-10-17",
        "Statement": [
          {
            "Sid": "WriteLogs",
            "Effect": "Allow",
            "Action": [
              "logs:CreateLogGroup",
              "logs:CreateLogStream",
              "logs:PutLogEvents"
            ],
            "Resource": "arn:aws:logs::*::*"
          },
          {
            "Sid": "GetAuthSettings",
            "Action": [
              "s3:GetObject",
              "s3:HeadObject"
            ],
            "Effect": "Allow",
            "Resource": [
              { "Fn::Join": [ "", [ "arn:aws:s3:::", { "Ref": "Configuration" }, "/player-access/auth-settings.json" ] ] }
            ]
          },
          {
            "Sid": "DescribeStacks",
            "Action": [
              "cloudformation:DescribeStackResources",
              "cloudformation:DescribeStackResource"
            ],
            "Effect": "Allow",
            "Resource": [ "*" ]
          }
        ]
      }
    }
  ]
}
"ProjectResourceHandlerExecution": {
    "Type": "AWS::IAM::Role",
    "Properties": {
        "AssumeRolePolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Action": "sts:AssumeRole",
                    "Principal": {
                        "Service": "lambda.amazonaws.com"
                    }
                }
            ]
        }
    },
    "Policies": [
        {
            "PolicyName": "ProjectAccess",
            "PolicyDocument": {
                "Version": "2012-10-17",
                "Statement": [
                    {
                        "Sid": "WriteLogs",
                        "Effect": "Allow",
                        "Action": [
                            "logs:CreateLogGroup",
                            "logs:CreateLogStream",
                            "logs:PutLogEvents"
                        ],
                        "Resource": "arn:aws:logs:*:*:*"
                    },
                    {
                        "Sid": "ReadAndWriteUploadedConfiguration",
                        "Effect": "Allow",
                        "Action": [
                            "s3:GetObject",
                            "s3:PutObject"
                        ],
                        "Resource": {
                            "Fn::Join": [
                                "",
                                [
                                    "arn:aws:s3:::", 
                                    {
                                        "Ref": "Configuration" 
                                    }, 
                                    "/upload/**"]
                            ]
                        }
                    },
                    {
                        "Sid": "DescribeStacksAndResources",
                        "Effect": "Allow",
                        "Action": [
                            "cloudformation:DescribeStackResources",
                            "cloudformation:DescribeStackResource",
                            "cloudformation:ListStacks",
                            "cloudformation:DescribeStackEventHistory" 
                        ],
                        "Resource": "arn:aws:cloudformation:*:*:*"
                    }
                ]
            }
        }
    ]
},
"Roles": [
    {
        "Type": "AWS::IAM::Role",
        "Properties": {
            "AssumeRolePolicyDocument": {
                "Version": "2012-10-17",
                "Statement": [
                    {
                        "Effect": "Allow",
                        "Action": "sts:AssumeRole",
                        "Principal": {
                            "Service": "lambda.amazonaws.com"
                        }
                    }
                ]
            }
        },
        "Policies": [
            {
                "PolicyName": "ProjectAccess",
                "PolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Sid": "WriteLogs",
                            "Effect": "Allow",
                            "Action": [
                                "logs:CreateLogGroup",
                                "logs:CreateLogStream",
                                "logs:PutLogEvents"
                            ],
                            "Resource": "arn:aws:logs:*:*:*"
                        },
                        {
                            "Sid": "ReadAndWriteUploadedConfiguration",
                            "Effect": "Allow",
                            "Action": [
                                "s3:GetObject",
                                "s3:PutObject"
                            ],
                            "Resource": {
                                "Fn::Join": [
                                    "",
                                    [
                                        "arn:aws:s3:::", 
                                        {
                                            "Ref": "Configuration" 
                                        }, 
                                        "/upload/**"]
                                ]
                            }
                        }
                    ]
                }
            }
        ]
    }
]
"cloudformation:DescribeStacks",
  "Resource": [ "*" ]
},
{
  "Sid": "ManagePlayerAndFunctionRoles",
  "Effect": "Allow",
  "Action": [ "iam:CreateRole",
              "iam:DeleteRole",
              "iam:GetRole",
              "iam:DeleteRolePolicy",
              "iam:PutRolePolicy"
           ],
  "Resource": { "Fn::Join": [ "", [ "arn:aws:iam::", { "Ref": "AWS::AccountId" }, ":role/", { "Ref": "AWS::StackName" }, "/*" ] ] }
}
{
  "Sid": "CreateUpdateCognitoIdentity",
  "Effect": "Allow",
  "Action": [ "cognito-identity:*" ],
  "Resource": { "Fn::Join": [ "", [ "arn:aws:cognito-identity:", { "Ref": "AWS::Region" }, ":", { "Ref": "AWS::AccountId" }, ":identitypool/*" ] ] }
}
{
  "Sid": "ReadPlayerAccessConfiguration",
  "Effect": "Allow",
  "Action": [ "s3:GetObject" ],
  "Resource": { "Fn::Join": [ "", [ "arn:aws:s3::", { "Ref": "Configuration" }, "/player-access/auth-settings.json" ] ] }
}
"ProjectResourceHandler": {
  "Type": "AWS::Lambda::Function",
  "Properties": {
    "Description": "Implements the custom resources used in this project's templates.",
    "Handler": "custom_resource.handler",
    "Role": { "Fn::GetAtt": [ "ProjectResourceHandlerExecution", "Arn" ] },
    "Runtime": "python2.7",
    "Timeout": 90,
"Code": {
    "S3Bucket": { "Ref": "Configuration" },
    "S3Key": { "Fn::Join": [ "/", [ { "Ref": "ConfigurationKey" }, "project-code.zip" ] ] }
}

"ProjectPlayerAccessTokenExchangeHandler": {
    "Type": "AWS::Lambda::Function",
    "Properties": {
        "Description": "Implements the token exchange for OAuth and openid used for player access.",
        "Handler": "auth_token_exchange.handler",
        "Role": { "Fn::GetAtt": [ "ProjectPlayerAccessTokenExchangeHandlerRole", "Arn" ] },
        "Runtime": "python2.7",
        "Code": {
            "S3Bucket": { "Ref": "Configuration" },
            "S3Key": { "Fn::Join": [ "/", [ { "Ref": "ConfigurationKey" }, "project-code.zip" ] ] }
        }
    }
}

ConfigurationKey Parameter

The ConfigurationKey parameter identifies the location of configuration data in the configuration bucket. The parameter value is set by Cloud Canvas when you use the template to update the AWS CloudFormation stack.

Configuration Resource

The Configuration resource describes the Amazon S3 bucket that is used to store project configuration data.

ProjectPlayerAccessTokenExchangeHandlerRole Resource

The ProjectPlayerAccessTokenExchangeHandlerRole resource describes the IAM role that is used to grant permissions to the ProjectPlayerAccessTokenExchangeHandler resource.

ProjectResourceHandlerExecution Resource

The ProjectResourceHandlerExecution resource describes the IAM role that is used to grant permissions to the ProjectResourceHandler Lambda function resource.

ProjectResourceHandler Resource

The ProjectResourceHandler resource describes the Lambda function that implements the AWS CloudFormation custom resource handler that implements the custom resources used in the project's
AWS CloudFormation templates. The code for this Lambda function is uploaded from the
(game)\AWS\project-code directory by the lmbr_aws create-project-stack (p. 169) and
update-project-stack (p. 178) commands. For more information, see Custom Resources (p. 242).

**ProjectPlayerAccessTokenExchangeHandler Resource**

The ProjectPlayerAccessTokenExchangeHandler resource describes the Lambda function that
implements the token exchange process for player access. The code for this Lambda function is uploaded
from the (game)\AWS\project-code directory by the lmbr_aws create-project-stack (p. 169) and
update-project-stack (p. 178) commands. For more information, see Access Control and Player
Identity (p. 246).

**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.

An example deployment-template.json file follows.

```json
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Metadata": {
        "CloudCanvas": {
            "Id": "$Revision: #4 $"
        }
    },
    "Parameters": {
        "ProjectResourceHandler": {
            "Type": "String",
            "Description": "Service token of the custom resource handler."
        },
        "ConfigurationBucket": {
            "Type": "String",
            "Description": "Bucket that contains configuration data."
        },
        "ConfigurationKey": {
            "Type": "String",
            "Description": "Location in the configuration bucket of configuration data."
        },
        "DeploymentName": {
            "Type": "String",
            "Description": "The name of the deployment associated with this stack."
        },
        "ProjectStackId": {
            "Type": "String",
            "Description": "The ARN of the project stack associated with this deployment."
        }
    },
    "Resources": {
```
"HelloWorldConfiguration" : {
    "Type": "Custom::ResourceGroupConfiguration",
    "Properties": {
        "ServiceToken": { "Ref": "ProjectResourceHandler" },
        "ConfigurationBucket": { "Ref": "ConfigurationBucket" },
        "ConfigurationKey": { "Ref": "ConfigurationKey" },
        "ResourceGroup": "HelloWorld"
    }
},

"HelloWorld": {
    "Type": "AWS::CloudFormation::Stack",
    "Properties": {
        "TemplateURL": { "Fn::GetAtt": [ "HelloWorldConfiguration", "TemplateURL" ] },
        "Parameters": {
            "ProjectResourceHandler": { "Ref": "ProjectResourceHandler" },
            "ConfigurationBucket": { "Fn::GetAtt": [ "HelloWorldConfiguration", "ConfigurationBucket" ] },
            "ConfigurationKey": { "Fn::GetAtt": [ "HelloWorldConfiguration", "ConfigurationKey" ] }
        }
    }
},

"Outputs": {
    "StackName": {
        "Description": "The deployment stack name.",
        "Value": {"Ref": "AWS::StackName"}
    }
}

Parameters

The deployment-template.json file has the following parameters. The parameter values are provided by Cloud Canvas when it uses the template to update an AWS CloudFormation stack.

ProjectResourceHandler Parameter

The ProjectResourceHandler parameter identifies the custom resource handler Lambda function used for the project.

ConfigurationBucket Parameter

The ConfigurationBucket parameter identifies the configuration bucket.

ConfigurationKey Parameter

The ConfigurationKey parameter identifies the location of configuration data in the configuration bucket.

DeploymentName Parameter

The DeploymentName parameter identifies the deployment name associated with this stack.
**ProjectStackId Parameter**

The `ProjectStackId` parameter identifies project stack associated with this deployment.

**Resources**

The `deployment-template.json` file defines two resources:

**HelloWorldConfiguration Resource**

The `HelloWorldConfiguration` resource describes a `ResourceGroupConfiguration (p. 243)` custom resource that is used to configure the `HelloWorld` resource.

The `deployment-template.json` file contains a similar `ResourceGroupConfiguration` resource for each of the project's resource groups.

**HelloWorld Resource**

The `HelloWorld` resource describes the AWS CloudFormation stack that implements the project's `HelloWorld` resource group.

The `deployment-template.json` file contains a similar AWS CloudFormation stack resource for each of the project's resource groups.

**Outputs**

The `Outputs` section of the template defines values that the template generates.

**deployment-access-template.json**

The `deployment-access-template.json` file is an AWS CloudFormation Template that defines the resources used to secure a deployment.

An example `deployment-access-template.json` file follows.

```json
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Metadata": {
        "CloudCanvas": {
            "Id": "$Revision: #6 $"
        }
    },
    "Parameters": {
        "ProjectResourceHandler": {
            "Type": "String",
            "Description": "The the project resource handler lambda ARN."
        },
        "ConfigurationBucket": {
            "Type": "String",
            "Description": "Bucket that contains configuration data."
        }
    }
}
```
"ConfigurationKey": {
    "Type": "String",
    "Description": "Key that contains the current upload location."
},

"ProjectPlayerAccessTokenExchangeHandler": {
    "Type": "String",
    "Description": "ARN for the lambda that the login cognito-identity pool needs access to."
},

"ProjectStack": {
    "Type": "String",
    "Description": "The name of the project stack."
},

"DeploymentName": {
    "Type": "String",
    "Description": "The name of the deployment."
},

"DeploymentStack": {
    "Type": "String",
    "Description": "The name of the deployment stack."
},

"DeploymentStackArn": {
    "Type": "String",
    "Description": "The ARN of the deployment stack."
}

},

"Resources": {

"OwnerPolicy": {
    "Type": "AWS::IAM::ManagedPolicy",
    "Properties": {
        "Description": "Policy that grants permissions to update a deployment stack, and all of it's resource group stacks."
    }
},

"Path": {
    "Fn::Join": [
        "", [
            { "Ref": "ProjectStack" }, 
            "/", { "Ref": "DeploymentName" }, 
            "/" ]
    }
},

"PolicyDocument": {
    "Version": "2012-10-17",
    "Statement": [{
        "Sid": "ReadProjectDeploymentAndResourceGroupStack State",
        "Effect": "Allow",
        "Action": {
            "cloudformation:DescribeStackResource",
            "cloudformation:DescribeStackResources",
            "cloudformation:DescribeStackEvents"
        },
        "Resource": {
            "Fn::Join": [
                "", [
                    "arn:aws:cloudformation",
                    ":", { "Ref": "AWS::Region" }, 
                    ":", { "Ref": "AWS::AccountId" }, 
                    ":stack/", { "Ref": "ProjectStack" }, 
                    "/**" ]
        }
    }
}


```json
{
  "Description": "Permissions for the execution of the cloudformation::CreateStack, cloudformation::UpdateStack, and cloudformation::DeleteStack actions.

Sid: "InvokeProjectResourceHandler",
  "Effect": "Allow",
  "Action": ["lambda:InvokeFunction"],
},
{
  "Description": "Permissions for creating and updating Stacksets.

Sid: "ReadAndWriteDeploymentAndResourceGroupConfiguration",
  "Effect": "Allow",
  "Action": ["s3:PutObject", "s3:GetObject"],
  "Resource": ["arn:aws:s3:::ConfigurationBucket/*/deployment/*"]
},
{
  "Description": "Permissions for creating, updating, and deleting Stacksets.

Sid: "UpdateDeploymentStack",
  "Effect": "Allow",
  "Action": ["cloudformation:UpdateStack"],
},
{
  "Description": "Permissions for creating, updating, and deleting Stacksets.

Sid: "CreateUpdateAndDeleteResourceGroupStacks",
  "Effect": "Allow",
  "Action": ["cloudformation:CreateStack", "cloudformation:UpdateStack", "cloudformation:DeleteStack"],
}
}```
{
  "Owner": {
    "Type": "AWS::IAM::Role",
    "Properties": {
      "Path": {
        "Fn::Join": [
          "/",
          {
            "Fn::Join": [
              "/",
              {
                "Fn::Join": [
                  "/",
                  {
                    "Ref": "ProjectStack"
                  }
                
              }
            ]
          }
        ]
      }
    }
  }
}
}
"Player": {
  "Type": "AWS::IAM::Role",
  "Properties": {
    "Path": {
      "Fn::Join": [
        "",
        [
          { "Ref": "ProjectStack" },
          "/",
          { "Ref": "DeploymentName" },
          "/"
        ]
      ]
    },
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Sid": "AccountUserAssumeRole",
          "Effect": "Allow",
          "Action": "sts:AssumeRole",
          "Principal": {
            "AWS": {
              "Ref": "AWS::AccountId"} } } ]
    },
    "ManagedPolicyArns": [
      { "Ref": "OwnerPolicy" }
    ]
  }
},

"PlayerAccess": {
  "Type": "Custom::PlayerAccess",
  "Properties": {
    "ServiceToken": {
      "Ref": "ProjectResourceHandler" },
    "ConfigurationBucket": {
      "Ref": "ConfigurationBucket" },
    "ConfigurationKey": {
      "Ref": "ConfigurationKey" },
    "DeploymentStack": {
      "Ref": "DeploymentStackArn" }
  },
  "DependsOn": [ "Player" ]
},

"PlayerLoginRole": {
  "Type": "AWS::IAM::Role",
  "Properties": {
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Action": "sts:AssumeRoleWithWebIdentity",
          "Principal": {
            "Federated": "cognito-identity.amazonaws.com" }
        } ]
    } }
"Principal": {
    "Federated": "cognito-identity.amazonaws.com"
  }
},
"Policies": [
  {
    "PolicyName": "ExchangeTokenAccess",
    "PolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Sid": "PlayerLoginExecution",
          "Effect": "Allow",
          "Action": [ "lambda:InvokeFunction" ],
          "Resource": { "Ref": "ProjectPlayerAccessTokenExchangeHandler" }
        }
      ]
    }
  }
],
"PlayerLoginIdentityPool": {
  "Type": "Custom::CognitoIdentityPool",
  "Properties": {
    "ServiceToken": { "Ref": "ProjectResourceHandler" },
    "AllowUnauthenticatedIdentities": "true",
    "UseAuthSettingsObject": "false",
    "ConfigurationBucket": { "Ref": "ConfigurationBucket" },
    "ConfigurationKey": { "Ref": "ConfigurationKey" },
    "IdentityPoolName": "PlayerLogin",
    "Roles": {
      "unauthenticated": { "Fn::GetAtt": [ "PlayerLoginRole", "Arn" ] }
    }
  }
},
"PlayerAccessIdentityPool": {
  "Type": "Custom::CognitoIdentityPool",
  "Properties": {
    "ServiceToken": { "Ref": "ProjectResourceHandler" },
    "AllowUnauthenticatedIdentities": "true",
    "UseAuthSettingsObject": "true",
    "ConfigurationBucket": { "Ref": "ConfigurationBucket" },
    "ConfigurationKey": { "Ref": "ConfigurationKey" },
    "IdentityPoolName": "PlayerAccess",
    "Roles": {
      "unauthenticated": { "Fn::GetAtt": [ "Player", "Arn" ] },
      "authenticated": { "Fn::GetAtt": [ "Player", "Arn" ] }
    }
  }
}
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 the resources defined in the example deployment-access-template.json file.

OwnerPolicy Resource

The OwnerPolicy resource describes an IAM Managed Policy that gives owner level access to the deployment. The AWS account administrator always has full access to the deployment, but may want to limit other users’ access to specific deployments. That can be done by attaching OwnerPolicy to an IAM User (or you can use the Owner role, which is also defined by the deployment access template).

Owner access includes the following:

• The ability to update the deployment and all of its resource groups.
• Full access to the group’s resources created for the deployment.

For more information, see Project Access Control (p. 246).

Owner Resource

The Owner resource describes an IAM role with the OwnerPolicy attached.

For more information, see Project Access Control (p. 246).

Player Resource

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. 234)). The role’s policies are updated by the PlayerAccess custom resources that appear in the deployment-access-template.json (p. 227) and in the resource-template.json (p. 234) files. The PlayerAccessIdentityPool Amazon Cognito identity pool resource allows players to assume this role.

For more information, see PlayerAccessIdentityPool Resource (p. 234) and Access Control and Player Identity (p. 246).

PlayerAccess Resource

The PlayerAccess resource describes a PlayerAccessIdentityPool Resource (p. 234). This resource is responsible for configuring the player role using the PlayerAccess metadata found on the resources to which the player should have access.
For more information, see Access Control and Player Identity (p. 246).

**PlayerLoginRole Resource**

The PlayerLoginRole resource describes the IAM role that is temporarily assumed by the player as part of the login process.

For more information, see Access Control and Player Identity (p. 246).

**PlayerLoginIdentityPool Resource**

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 Access Control and Player Identity (p. 246).

**PlayerAccessIdentityPool Resource**

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 Access Control and Player Identity (p. 246).

**The project-code Subdirectory**

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. 242).

It also contains the code that implements the token exchange step of the player login process. For more information, see Access Control and Player Identity (p. 246).

**resource-group\{resource-group\} subdirectories**

The AWS resources used by the game are organized into separate resource groups, as represented by individual \{resource-group\} subdirectories under the parent resource-group directory. The resource-group directory may contain any number of \{resource-group\} subdirectories, each typically named after your game project.

**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.

The example resource-template.json file that follows defines a SayHello Lambda function that is executed by the game to generate a greeting for a player. The generated message is stored in a DynamoDB table.

```json
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Metadata": {
        "CloudCanvas": {
            "Id": "$Revision: #3 $"
        }
    }
}```
"Parameters": {
  "ProjectResourceHandler": {
    "Type": "String",
    "Description": "Service token of the custom resource handler."
  },
  "ConfigurationBucket": {
    "Type": "String",
    "Description": "Bucket that contains configuration data."
  },
  "ConfigurationKey": {
    "Type": "String",
    "Description": "Location in the configuration bucket of configuration data."
  },
  "ReadCapacityUnits": {
    "Type": "Number",
    "Default": "1",
    "Description": "Number of game state reads per second."
  },
  "WriteCapacityUnits": {
    "Type": "Number",
    "Default": "1",
    "Description": "Number of game state writes per second."
  },
  "Greeting": {
    "Type": "String",
    "Default": "Hello",
    "Description": "Greeting used by the SayHello Lambda function."
  }
},

"Resources": {
  "Messages": {
    "Type": "AWS::DynamoDB::Table",
    "Properties": {
      "AttributeDefinitions": [
        {
          "AttributeName": "PlayerId",
          "AttributeType": "S"
        }
      ],
      "KeySchema": [
        {
          "AttributeName": "PlayerId",
          "KeyType": "HASH"
        }
      ],
      "ProvisionedThroughput": {
        "ReadCapacityUnits": "1",
        "WriteCapacityUnits": "1"
      }
    }
  }
}
"ReadCapacityUnits": { "Ref": "ReadCapacityUnits" },
"WriteCapacityUnits": { "Ref": "WriteCapacityUnits" }
},
"Metadata": {
  "CloudCanvas": {
    "FunctionAccess": [
      {
        "FunctionName": "SayHello",
        "Action": "dynamodb:PutItem"
      }
    ]
  }
},
"SayHelloConfiguration": {
  "Type": "Custom::LambdaConfiguration",
  "Properties": {
    "ServiceToken": { "Ref": "ProjectResourceHandler" },
    "ConfigurationBucket": { "Ref": "ConfigurationBucket" },
    "ConfigurationKey": { "Ref": "ConfigurationKey" },
    "FunctionName": "SayHello",
    "Runtime": "python2.7",
    "Settings": {
      "MessagesTable": { "Ref": "Messages" },
      "Greeting": { "Ref": "Greeting" }
    }
  }
},
"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": {
      "PlayerAccess": {
        "Action": "lambda:InvokeFunction"
      }
    }
  }
},
"PlayerAccess": {
Resource Template Parameters

This section describes the parameters defined in the example resource-template.json file. Parameter values are provided by Cloud Canvas when it uses the template to update an AWS CloudFormation stack.

**ProjectResourceHandler Parameter**

The ProjectResourceHandler parameter identifies the custom resource handler Lambda function used for the project.

**ConfigurationBucket Parameter**

The ConfigurationBucket parameter identifies the configuration bucket.

**ConfigurationKey Parameter**

The ConfigurationKey parameter identifies the location of configuration data in the configuration bucket.

**ReadCapacityUnits and WriteCapacityUnits Parameters**

The ReadCapacityUnits and WriteCapacityUnits parameters are used to configure the Messages resource defined by the template. Values for parameters such as these are typically provided by the project-settings.json (p. 217) and can be customized for each deployment.

Resource Template Resources

This section describes the resources defined in the example resource-template.json file.

**Messages Resource**

The Messages resource describes a DynamoDB Table. See AWS::DynamoDB::Table for a description of the AWS CloudFormation DynamoDB table resource definition format.

The Metadata.CloudCanvas.FunctionAccess property of the resource definition is used by the SayHelloConfiguration custom resource to grant the SayHello Lambda function resource permission to write data into the table. For more information, see Lambda Function Access Control (p. 247).

**SayHelloConfiguration Resource**

The SayHelloConfiguration resource describes a LambdaConfiguration (p. 244) resource that provides various configuration inputs for the SayHello Lambda function resource.
The **Settings** property for this resource is used to pass configuration data to the **SayHello** Lambda function. For more information, see **LambdaConfiguration** (p. 244).

### SayHello Resource

The **SayHello** resource describes a Lambda function resource that implements some of the game logic. See **AWS::Lambda::Function** for a description of the AWS CloudFormation Lambda function resource definition format.

The Lambda function’s **Execution Role**, which determines the AWS permissions the function has when it executes, is created by the **SayHelloConfiguration** resource, which uses the Metadata.CloudCanvas.FunctionAccess properties that appear on the resources that the function can access.

The Metadata.CloudCanvas.PlayerAccess property of the resource definition determines the access that players have to the **SayHello** resource. In this case, they can only invoke the lambda function.

### PlayerAccess Resource

The **PlayerAccess** resource in the resource template is a **PlayerAccess** (p. 245) custom resource. It grants players access to resources specified by the Metadata.CloudCanvas.PlayerAccess properties on the definitions of the resources to which they have access.

Note that the **PlayerAccess DependsOn** property lists the resources that define this metadata property. This ensures that AWS CloudFormation creates or updates the **PlayerAccess** resources after the resources with the metadata property have been created or updated.

### The lambda-function-code Subdirectory

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 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. 242).
- `{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 Access Control and Player Identity (p. 246).

• `{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-OwnerPolicy – An IAM managed policy that grants "owner" access to a deployment. See Project Access Control (p. 246).

  • `{project}`-{deployment}-Access-Owner – An IAM role that grants "owner" access to a deployment. See Project Access Control (p. 246).

  • `{project}`-{deployment}-Access-Player – An IAM role that grants "player" access to a deployment. See Access Control and Player Identity (p. 246).

  • `{project}`-{deployment}-Access-PlayerLoginRole – An IAM role that grants players temporary anonymous access used during the player login process. See Access Control and Player Identity (p. 246).

  • `{project}`-{deployment}-Access-PlayerLoginIdentityPool – An Amazon Cognito identity pool used for player identity. For more information, see Access Control and Player Identity (p. 246).

  • `{project}`-{deployment}-Access-PlayerAccessIdentityPool – An Amazon Cognito identity pool that provides the temporary player identity used during the player login process. For more information, see Access Control and Player Identity (p. 246).

• `{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-1A1WIH6M2KPRI. The tools provided by Lumberyard hide these actual resource names under most circumstances.

## Configuration Bucket

The configuration Amazon S3 bucket is used to store configuration data for the project. The tools provided with Cloud Canvas manage uploads to this bucket.

The configuration bucket contents are as follows.

```
/  
  upload/  
    {upload-id}/  
      project-template.json  
      project-code.zip  
    deployment/  
      {deployment}/  
        deployment-template.json  
      resource-group/  
        {resource-group}/  
          resource-template.json  
          lambda-function-code.zip  
          lambda-function-code.zip.{function-name}.configured
```
All the /upload/{upload-id}/.* objects in this bucket, except the *.configured objects, 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.

The lambda-function-code.zip.{function-name}.configured objects in this bucket are created by the LambdaConfiguration custom resources when settings are injected into the code. See LambdaConfiguration (p. 244) for more information.

The /player-access/auth-settings.json file stores the security credentials used to implement player login by using social networks such as Facebook or by using the player's Amazon credentials. This file is created and updated by the `lmbr_aws add-login-provider` (p. 167), `update-login-provider` (p. 177), and `remove-login-provider` (p. 176) commands.

### Resource Mappings

Resource mappings map the friendly names used in a game's Resource Definitions (p. 216) to the actual names of the resources created for one or more specific Resource Deployments (p. 238). For example, a DynamoDB table name like DailyGiftTable would get mapped to a name like SamplesProject-DontDieDeployment-78AIXR0NO04N-DontDieAWS-1I12C6Y07KU7F-DailyGiftTable-1G4G33K16D82S 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 78AIXR0NO04N, 1I12C6Y07KU7F and 1G4G33K16D82S 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 `{root}/game/AWS/project-settings.json` (p. 217) file and can be overridden for each user by the `{root}/Cache/game/pc/user/AWS/user-settings.json` (p. 219) file. You can change the default deployment by using the `lmbr_aws default-deployment` (p.170) command or by using the Cloud Canvas Resource Manager (p. 184).

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. 219) 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 update-mappings` (p.178) 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 update-mappings --release` (p. 178) 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. 217) file.

In both cases, the AWS Client Configuration (p. 252) system is configured with the mapping data. You can use this configuration in AWS flow nodes, with the AWS C++ SDK, and in Lambda functions.
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. The AWS flow nodes use the AWS Client Configuration (p. 252) system to look up the actual resource name by referring to the names in the {resource-group}.{resource} format. For detailed information on the Cloud Canvas flow graph nodes, see the Cloud Canvas Flow Graph Node Reference (p. 193).

Using Mappings with the AWS C++ SDK

The AWS Client Configuration (p. 252) system supports the configuration of service specific clients using arbitrary application defined names. The Lumberyard IDE uses names in the {resource-group}.{resource} format to define configurations that map from those friendly names to the actual resource name that must be provided when calling AWS C++ SDK APIs. This configuration can be retrieved and used by using C++ code like the following.

Code example: mapping resource names in C++

```cpp
#include <LmbrAWS\IAWSClientManager.h>
#include <LmbrAWS\ILmbrAWS.h>
#include <aws\lambda\LambdaClient.h>
#include <aws\lambda\model\InvokeRequest.h>
#include <aws\lambda\model\InvokeResult.h>
#include <aws\core\utils\Outcome.h>

void InvokeSayHello()
{
    LmbrAWS::IClientManager* clientManager = pEnv->pLmbrAWS->GetClientManager();

    // Use {resource-group}.{resource} format name to lookup the configured client.
    LmbrAWS::Lambda::FunctionClient client = clientManager->GetLambdaManager().CreateFunctionClient("HelloWorld.SayHello");

    // Get the Lambda function resource name using the client.GetFunctionName() method.
    Aws::Lambda::Model::InvokeRequest req;
    req.SetFunctionName(client.GetFunctionName());

    // Invoke the Lambda function asynchronously (async isn't required, but you should never
    // do network I/O in the main game thread).
    client->InvokeAsync(req,
    [](const Aws::Lambda::LambdaClient*, const Aws::Lambda::Model::InvokeRequest&, const Aws::Lambda::Model::InvokeOutcome& outcome, const std::shared_ptr<const Aws::Client::AsyncCallerContext>&)
    {
        if(outcome.IsSuccess())
        {
            const Aws::Lambda::Model::InvokeResult& res = outcome.GetResult();
            // Process lambda result
        }
    });
```

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. 244).

**Custom Resources**

Cloud Canvas provides a number of AWS CloudFormation custom resources that can be used in the project, deployment, and resource group AWS CloudFormation template files. These custom resources are implemented by the Lambda function code found in the `{root}\{game}\AWS\project-code` directory and the `ProjectResourceHandler` resource defined in the `{root}\{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. 242) – Manages Amazon Cognito identity pool resources.
- **EmptyDeployment** (p. 243) – Used in the `deployment-template.json` when there are no resource groups defined.
- **LambdaConfiguration** (p. 244) – Provides configuration data for Lambda function resources and maintains the Lambda function's execution role.
- **PlayerAccess** (p. 245) – Maintains the policies on the player role.

**CognitoIdentityPool**

The `Custom::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 created using 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.

### EmptyDeployment

The `Custom::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::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::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 up 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 Lambda Function Access Control (p. 247).

**PlayerAccess**

The Custom::PlayerAccess resource is used in resource-template.json files to update the player role so that players have the desired access to the resource group’s resources. It is also used in the deployment-access-template.json file to update the player role so that players have the desired access to the deployment’s 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 deployment is located in the configuration bucket. The value of this property isn’t actually used, however since the Cloud Canvas tools insure that the key is different for each AWS CloudFormation operation, the presences of this property has the effect of forcing the custom resource handler to be executed by AWS CloudFormation on for every operation.

• ResourceGroup

  Optional. The ID of the resource group for which the player role is updated.

• DeploymentStack

  Optional. The ID of the deployment for which the player role is updated.

  Only one of the ResourceGroup and DeploymentStack properties must be provided.

**Output Properties**

The PlayerAccess custom resource does not produce any output values.

**PlayerAccess Metadata Format**

This custom resource looks for Metadata.CloudCanvas.PlayerAccess properties on the project’s resource group definitions and constructs a policy which is attached to the player role. The policy allows the indicated actions on those resources. The Metadata.CloudCanvas.PlayerAccess property has the following form:

```json
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Resources": {
    "...": {
      "Type": "...",
      "Properties": {
```
The required `Action` property is the same as defined for an IAM policy and is described in detail in the IAM Policy Elements Reference. Note that a single value can be provided instead of a list of values.

### Access Control and Player Identity

Cloud Canvas helps you control access to your game's AWS resources in three ways:

- **Project Access Control** (p. 246)
- **Player Access Control** (p. 247)
- **Lambda Function Access Control** (p. 247)

### Project Access Control

It is often necessary to limit project team member access to the project's resources. This can help prevent different development teams from accidentally updating the resources being used by another development team. It is also necessary to prevent project team members from accessing the resources used by the released version of the game but in some cases to also prevent project team members from accessing player's personal information, such as e-mail addresses, which may be stored in those resources.

The default `deployment-access-template.json` (p. 227) file provided by Cloud Canvas defines an OwnerPolicy Resource (p. 233) IAM managed policy resource, which allows a deployment AWS CloudFormation stack to be updated, including creating, updating, and deleting the resources defined by the project's resource groups. This template also defines an Owner Resource (p. 233). IAM role resource that has the OwnerPolicy attached.

If desired, the OwnerPolicy resource definition in the `deployment-access-template.json` file can be modified or additional policies can be created. However, be sure that you really understand how IAM permissions work before doing so. Incorrectly using this resource definition can make your AWS account vulnerable to attack and abuse, which could result in unexpected AWS charges (in addition to any other repercussions).

### Authorize AWS Use in Lumberyard Editor

To authorize a group of developers to use AWS via Lumberyard Editor, perform the following steps.
To authorize AWS use in Lumberyard Editor

1. Create an IAM user for each developer.
2. Generate the access key and secret keys for each user.
3. Attach a policy to the IAM user that determines what that user is allowed to do. These policies are generated when a project is initialized, or you can apply your own.
4. Deliver the access key and secret key to the developer by a secure method.

   **Caution**
   
   You should not deliver access or secret keys by using email, or check them into source control. Such actions present a significant security risk.

5. In Lumberyard Editor, have each developer navigate to AWS, Cloud Canvas, Permissions and deployments.
6. Have the developer add a new profile that uses the access key and secret key that he or she has been provided.

**Player Access Control**

In order for the game to access AWS resources at runtime, it must use credentials that grant the necessary access when calling AWS APIs. This could be done by creating an IAM user with limited privileges and embedding that user’s credentials (both the AWS access key and the secret key) in the game code. But AWS Amazon Cognito identity pools provide a more powerful and secure solution for this problem.

How Cloud Canvas uses Amazon Cognito identity pools is described in the Player Identity (p. 249) section.

Ultimately player access is controlled by the player role defined in the default Cloud Canvas deployment-access-template.json (p. 227) file. The policies attached to this role are set by the PlayerAccess Resource (p. 238) custom resources that appear in the resource-template.json (p. 234) files.

**Lambda Function Access Control**

When an AWS Lambda function is executed, it assumes an IAM role that determines the access the function has to other AWS resources. Creating and configuring such roles requires IAM permissions that cannot safely be granted to all the project's team members; doing so would allow them to circumvent the security measures that limit their access to specific deployments.

To implement Lambda-function access control without requiring that the project team members be granted these IAM privileges, you use the Cloud Canvas LambdaConfiguration (p. 244) custom resource. Using the Metadata.CloudCanvas.FunctionAccess entries on each of the group resources to which a Lambda function requires access, the handler for the LambdaConfiguration resource creates and configures a role for each Lambda function that allows the function to perform the indicated actions on the resources it requires.

The Metadata.CloudCanvas.Function property has the following form:

```json
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Resources": {
    "...": {
      "Type": "...",
      "Properties": {
        "..."
      },
      "Metadata": {
```

Version 1.4
247
The required `Action` property is the same as defined for an IAM policy and is described in detail in the IAM Policy Elements Reference. Note that a single value can be provided instead of a list of values.

The optional `ResourceSuffix` property value is appended to the resource's ARN in the generated policy. This can be used to further restrict access to the resource. For example, for Amazon S3 buckets it can be used to restrict access to objects with matching names. For more information, see `Resource` in the IAM Policy Elements Reference.

The following diagram illustrates how different elements of Lumberyard access control work together.

In the diagrammed example, Lambda functions do things like submit a player's high scores to a DynamoDB database or retrieve the top ten scores from it.

The Player Access Control IAM policy allows the game to call Lambda functions on behalf of the player. In turn, the Function Access Control policy determines the AWS resources that Lambda functions can access (in the example, it's a DynamoDB database). This secure arrangement prevents the player from accessing the DynamoDB database directly and offers the following benefits:
• It enables you to validate the input from the client and remove insecure or unwanted inputs. For example, if a client self-reports an impossibly high or low score, you can reject the unwanted value before it can be written to the database.
• It prevents a customer from trying to access another customer’s data.
• It prevents malicious attacks.

To create (and later, update as required) the DynamoDB database, Lambda functions, and access control policies, AWS CloudFormation reads the AWS CloudFormation templates from the Amazon S3 configuration bucket and executes the instructions they contain. AWS CloudFormation reads the deployment-access-template.json file and creates a Deployment Access Control IAM policy, which determines which resources AWS CloudFormation can create or update for a particular deployment. This is key in keeping development, test, and live deployments separate and secure from one another.

The templates also use custom resources to implement functionality that AWS CloudFormation by itself cannot perform. In Lumberyard, custom resources are like library functions. For example, the deployment-access-template.json file calls the CognitoIdentityPool custom resource to create Amazon Cognito identity pools. To create the Function Access Control IAM policy for each Lambda function, the template calls the LambdaConfiguration custom resource. The custom resource reads the FunctionAccess metadata entries for the particular resources to which the Lambda function should have access and creates the Function Access Control policy that is needed for the current user and deployment.

Similarly, the resource-template.json template PlayerAccess custom resource is called to create the Player Access Control policy, which determines the Lambda functions and other resources that the game can call and use on behalf of the player.

**Player Identity**

As described in the preceding section Player Access Control (p. 247), the 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.

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. 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.

Amazon Cognito allows a user to start with an unauthenticated identity and then associate that identity with an external identity at a later point in time while preserving the Amazon Cognito-provided identity.

Cloud Canvas supports both anonymous (unauthenticated) and authenticated player identities, but authenticated identity support is more complex and requires additional setup and coding.

**Anonymous (Unauthenticated) Player Login**

The login process for anonymous (unauthenticated) players is shown in the diagram below:
This 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.

## 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, shown in the diagram that follows, is more complex than the anonymous player login process. 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.

The presence of the auth_token cvar triggers the Cloud Canvas authenticated player login flow. If the cvar is not set, the anonymous player login process is used. The value of the cvar must be a string of the form {provider}:{id}, where {provider} identifies an external identity provider that you have configured for your game (see Configuring External Identity Providers (p.251) in the section that follows) and {id} is the player's identity as returned by the login process for that provider.

When auth_token is set, Cloud Canvas will pass the provided {id} value to the ProjectPlayerAccessTokenExchangeHandler Lambda function. The Lambda function calls the external provider's API to with the specified ID and receives a value that is passed to Amazon Cognito to get the player's identity and credentials. The calls made by ProjectPlayerAccessTokenExchangeHandler use application IDs and the secret values you provide as part of the external identity provider configuration process.

As shown in the diagram above, Cloud Canvas uses one Amazon Cognito identity pool to get the credentials used to invoke the ProjectPlayerAccessTokenExchangeHandler and a different Amazon Cognito identity pool to get the credentials used to access the rest of your game's resources. This is required because access ProjectPlayerAccessTokenExchangeHandler is always anonymous.

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 External Identity Providers

Cloud Canvas does not automate the process of retrieving an auth code from an external identity provider and setting the auth_token cvar. This is your responsibility as a game developer. Following are some possible implementation methods:

- On a PC, you can have your identity provider redirect its URI to a static web page that redirects the user to a custom URI. You can use the custom URI to launch the game and pass the auth code as a command line argument (for example, yourGame.exe +auth_token=provider:code). Cloud Canvas detects this command line argument and logs the user into your game. This only has to be done once since the auth tokens are cached locally.
- You can have your game retrieve the auth code itself (but for many external identity providers, this may require using an embedded web browser). After retrieving the auth code, you can call gEnv->lmbrAWS->GetClientManager()->Login(providerName, code), or just set the cvar auth_token.
- If you have a launcher for your game, you can embed a web browser window in the launcher to allow the player to log in to the external identity provider. You can then retrieve the auth code and launch the game by using the +auth_token=provider:code parameter.

External identity providers are configured using the lmbr_aws add-login-provider (p.167), update-login-provider (p.177), and remove-login-provider (p.176) 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.
You must run `lmbr_aws update-project` after running `add-login-provider`, `update-login-provider`, or `remove-login-provider` so that the PlayerAccessIdentityPool Resource (p. 234) 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.

### AWS Client Configuration

Access to AWS services from C++ is controlled through AWS client objects. Cloud Canvas provides a layer of abstraction on top of the AWS clients given by the SDK. This enables you to easily apply configuration changes across client objects, including those being used internally by Cloud Canvas flow graph nodes. These changes can be made through C++ or with the provided flow graph nodes.

#### Configuring AWS Flow Graph Nodes

Lumberyard provides flow graph nodes that perform various AWS operations, such as invoking a Lambda function or reading data from a DynamoDB table. Each flow graph node has a port that identifies the AWS resource to operate on. For example, DynamoDB nodes have a `TableName` port that identifies a DynamoDB table. For detailed information on the Cloud Canvas flow graph nodes, see the Cloud Canvas Flow Graph Node Reference (p. 193).

By default, the AWS resource uses the value you specify for the name. You can use the default client settings to configure the client that accesses the resource.

**To vary the resource name based on game settings**

1. In the default client settings, add a parameter reference in the resource name.
2. Use the `Cloud Canvas (AWS):Configuration:SetConfigurationVariable` flow graph node to set the value that replaces the parameter reference in the resource name.

For example, if you aren’t using Resource Deployments but still want to manage resources in different stages, you can use "DailyGift_$stage$" for your table name and set the "$stage$" parameter value to "test" or "release" based on the conditions you choose.

If you want to use different client settings (for example, timeouts) for different resources, you can use the C++ API to set up named configurations. When the resource name provided to the flow graph node matches a named configuration, the configuration is used to set up the client to access the resource.

The configuration-related flows are as follows:

- `ConfigureProxy` - Configures the HTTP proxy host, port, user, and password used for AWS service calls.
- `SetConfigurationVariable` - Sets the parameter value for the AWS resource.
- `GetConfigurationVariableValue` - Replaces the "$param$" substrings in a string with the current parameter value or "" if there is no current value.
- `SetDefaultRegion` - Sets the region for all AWS clients in the current project.
• ApplyConfiguration – Applies AWS configuration settings set by the other configuration flow graph nodes. This allows you to apply multiple changes at once to prevent changes from being applied out of the desired order.

Configuring Using C++

The Client Manager implementation and the AWS flow graph node implementations are provided through the Cloud Canvas Gem. To use the Client Manager, add the Cloud Canvas Gem to your game project by using the Project Configurator. To get the Client Manager in code, call gEnv->pLmbrAWS->GetClientManager(). This will return an interface called IClientManager.

The Client Manager maintains a set of parameter name to parameter value mappings. You can access these mappings by using the object returned by the GetConfigurationParameters method. You can then insert parameter values into certain configuration settings by using "$param-name$" substrings. If the parameter value contains "$param-name$" substrings, these are also replaced with the corresponding parameter value.

AWS client configurations are based on the properties of the object returned by the Client Manager’s GetDefaultClientSettings method. You can specify named overrides for these settings by using the collection object returned by the GetClientSettingsOverridesCollection method. Overrides can aggregate other overrides by name, allowing common configuration settings to be specified once and then reused. These names can include "$param-name$" values, which are replaced as described above. If the resource name for an AWS flow graph node matches the name of a client settings overrides collection entry, those overrides are used by the client to access the resource.

Service-specific configurations are provided through the following Client Manager methods: GetCognitoIdentityManager, GetDynamoDBManager, GetIAMManager, GetLambdaManager, GetS3Manager, GetSNSManager, GetSQSManager, and GetSTSManager.

Each of these service-specific managers provide the following:

• A GetDefaultClientSettingsOverrides method that returns an object that can be used to define the default settings for all of a particular service's clients.
• Methods that allow resources to be configured by name.

For example, the DynamoDB manager has a GetTableClientSettingsCollection method that you can use to configure settings (the configuration name and resource name associated with the configuration) for a specific DynamoDB table. The resource name can include "$param-name$" values, which are replaced as described above.

In an AWS flow graph node, you can use the configuration name as the resource name in order to use that configuration. In this case, the configuration name acts as an alias for the actual resource name. For example, a node could specify "DailyGift" as the table name and the resource configuration "DailyGift" could specify "foo-bar-$stage$" as the resource name.

If you use the name "DailyGift_$stage$" in a flow graph node and do not define a resource configuration for the name "DailyGift_$stage$", a default configuration is created that uses "DailyGift_$stage$" for both the configuration name and resource name.

Using Configured Clients from C++

The Client Manager methods (GetCognitoIdentityManager, GetDynamoDBManager, GetIAMManager, GetLambdaManager, GetS3Manager, GetSNSManager, GetSQSManager, and GetSTSManager) return objects that provide methods for creating clients configured with the settings described above.
The `CreateManagedClient` method returns a client object that is configured using the client settings overrides identified by the provided name. This object can be passed around by value. The client's methods are accessed using the `->` operator.

You can use the resource-specific `CreateXClient` methods (for example, `CreateTableClient` for a DynamoDB table resource) to configure a client to access a specific resource. The returned object has a method for retrieving the resource name (such as `client.GetTableName`), while the client methods are accessed using the `->` operator (e.g. `client->GetItem(...)`).

**Important**

You can create clients in an unconfigured state. Flow graph nodes and other game components can create clients before the configuration for the clients is fully initialized. For example, the Amazon Cognito credentials provider and configuration parameters may need to be initialized by flow graph nodes that are triggered at game start, before other flow graph nodes or components can access AWS services. Because of this, you must trigger client configuration explicitly once the settings are confirmed complete.

You must call the Client Manager's `ApplyConfiguration` method at least once before clients are ready for use. After you make configuration changes, you must call the method again for those changes to take effect. You can use the `IsReady` method (`client.IsReady()`, not `client->IsReady()`) to determine if a client has been configured.
Component Entity System

The Component Entity System is currently in preview and is undergoing active development. It will replace the legacy Entity System (p. 320).

This guide provides engine and game programmers with examples and best practices for creating and reflecting Lumberyard components in C++.

**Topics**
- Creating a Component (p. 255)
- Reflecting a Component for Serialization and Editing (p. 257)
- Slices and Dynamic Slices (p. 263)

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.

**Component Example**

An example component class skeleton follows:

```cpp
class MyComponent
 : public AZ::Component
{
 public:
  AZ_COMPONENT(MyComponent, "{0C09F774-DECA-40C4-8B54-3A93033EC381}");

  // AZ::Component interface implementation
  void Init() override {}
};
```
Component Elements

The required and optional elements that comprise a component are as follows:

AZ::Component

Every component must include AZ::Component somewhere in its inheritance ancestry. Noneditor components generally inherit directly from AZ::Component, but you can create your own component class hierarchies, as in the following example:

class MyComponent
: public AZ::Component

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, so it is safe to cache pointers or references to other components on the entity in performance-critical situations.

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:

```cpp
// Required Reflect function.
static void Reflect(AZ::ReflectContext* context);
```

For more information, see Reflecting a Component for Serialization and Editing (p. 257).

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:

```cpp
// 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);
```

Reflecting a Component for Serialization and Editing

The Component Entity System is currently in preview and is undergoing active development. It will replace the legacy Entity System (p. 320).
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 runtime state. This is typical; components commonly contain 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. 262). 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 not add an entry in `EditContext`. An example follows:

```cpp
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:

- **SerializeContext** – Contains reflection data for serialization and construction of objects.
- **EditContext** – Contains reflection data for visual editing of objects.
- **BehaviorContext** – Contains reflection for runtime 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 APIs 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.
Serialization

Reflecting a class for serialization involves a builder pattern style markup in C++, as follows:

```cpp
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 your 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 ints (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:

```cpp
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.");
```
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:

```cpp
->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
  - `&SomeGlobalFunction`
- Member functions
Change Notification Callbacks

Another commonly used feature of the edit context is its ability to bind a change notification callback:

```cpp
->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.

```cpp
->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 re-evaluate them, which may 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`. If you want to modify the options available in the combo box for `Property A` with the values from another `Property B`, 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, as in this example:
...  
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;
}

Slices and Dynamic Slices

A slice is a collection of configured entities (p. 255) 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 cascading 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.

![Image of File Browser with Set Dynamic Flag highlighted]

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:

![Image of File Browser with a newly generated dynamic slice]

**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. 257) 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.

```csharp
// 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

This section provides insight into Lumberyard's support for input devices, including information on setting up controls and action maps.

This section includes the following topics:
- Action Maps (p. 267)
- CryInput (p. 268)
- Setting Up Controls and Action Maps (p. 270)

Action Maps

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:

```cpp
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.

Action Map Manager Events

If you have other systems in place that should know about action map events, you can subscribe to Action Map Manager events using the interface IActionMapEventListener.
The following events are available:

- `eActionMapManagerEvent_ActionMapsInitialized` — Action map definitions have been successfully initialized.
- `eActionMapManagerEvent_DefaultActionEntityChanged` — Default action entity has been changed (Manager will automatically assign new action maps to this entity).
- `eActionMapManagerEvent_FilterStatusChanged` — An existing filter has been enabled/disabled.
- `eActionMapManagerEvent_ActionMapStatusChanged` — An existing action map has been enabled/disabled.

## 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:

```cpp
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.

### InputEventListener

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 joystick, 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:**

```c++
// 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
```
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. 273) 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:

```xml
<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="thirdperson" onPress="1" noModifiers="1" keyboard="f1" />
<!-- debug keys – end -->
</actionmap>
```

**Versioning**

```xml
<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_AAlways

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 now also define action filters directly in your defaultProfile.xml. The following attributes are available:

- name – How the filter will be identified
- type (actionFail, actionPass) – Specifies whether the filter will let an action fail or pass

```xml
<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:

```xml
<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. 270) and *Action Maps* (p. 267).

<table>
<thead>
<tr>
<th>Player Action</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Player Movement</strong></td>
<td>W, A, S, D</td>
</tr>
<tr>
<td><strong>Player Aim</strong></td>
<td>Mouse XY</td>
</tr>
<tr>
<td><strong>Jump</strong></td>
<td>Spacebar</td>
</tr>
<tr>
<td><strong>Sprint</strong></td>
<td>Shift</td>
</tr>
<tr>
<td><strong>Crouch</strong></td>
<td>C</td>
</tr>
<tr>
<td><strong>Slide (when sprinting)</strong></td>
<td>C</td>
</tr>
<tr>
<td><strong>Fire</strong></td>
<td>Mouse 1</td>
</tr>
<tr>
<td><strong>Zoom</strong></td>
<td>Mouse 2</td>
</tr>
<tr>
<td><strong>Melee</strong></td>
<td>V</td>
</tr>
<tr>
<td><strong>Fire Mode</strong></td>
<td>2</td>
</tr>
<tr>
<td><strong>Reload</strong></td>
<td>R</td>
</tr>
<tr>
<td><strong>Use</strong></td>
<td>F</td>
</tr>
<tr>
<td><strong>Toggle Weapon</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>Toggle Explosive</strong></td>
<td>3</td>
</tr>
<tr>
<td><strong>Toggle Binoculars</strong></td>
<td>B</td>
</tr>
<tr>
<td><strong>Toggle Light (attachment)</strong></td>
<td>L</td>
</tr>
<tr>
<td><strong>Third Person Camera</strong></td>
<td>F1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vehicle Action</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Accelerate</strong></td>
<td>W</td>
</tr>
<tr>
<td><strong>Boost</strong></td>
<td>Shift</td>
</tr>
<tr>
<td><strong>Brake/Reverse</strong></td>
<td>S</td>
</tr>
<tr>
<td><strong>Handbrake</strong></td>
<td>Spacebar</td>
</tr>
<tr>
<td><strong>Steer</strong></td>
<td>A/D</td>
</tr>
<tr>
<td><strong>Look</strong></td>
<td>Mouse XY</td>
</tr>
</tbody>
</table>
## Key Naming Conventions

This page lists some of the name conventions used for action maps.

### Key Gestures

<table>
<thead>
<tr>
<th>Letters</th>
<th>&quot;a&quot; - &quot;z&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbers</td>
<td>&quot;1&quot; - &quot;0&quot;</td>
</tr>
<tr>
<td>Arrows</td>
<td>&quot;up&quot;, &quot;down&quot;, &quot;left&quot;, &quot;right&quot;</td>
</tr>
<tr>
<td>Function keys</td>
<td>&quot;f1&quot; - &quot;f15&quot;</td>
</tr>
<tr>
<td>Numpad</td>
<td>&quot;np_1&quot; - &quot;np_0&quot;, &quot;numlock&quot;, &quot;np_divide&quot;, &quot;np_multiply&quot;, &quot;np_subtract&quot;, &quot;np_add&quot;, &quot;np_enter&quot;, &quot;np_period&quot;</td>
</tr>
<tr>
<td>Esc</td>
<td>&quot;escape&quot;</td>
</tr>
<tr>
<td>~</td>
<td>&quot;tilde&quot;</td>
</tr>
<tr>
<td>Tab</td>
<td>&quot;tab&quot;</td>
</tr>
<tr>
<td>CapsLock</td>
<td>&quot;capslock&quot;</td>
</tr>
<tr>
<td>Shift</td>
<td>&quot;lshift&quot;, &quot;rshift&quot;</td>
</tr>
</tbody>
</table>
### Key Naming Conventions

<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl</td>
<td>&quot;lctrl&quot;, &quot;rctrl&quot;</td>
</tr>
<tr>
<td>Alt</td>
<td>&quot;lalt&quot;, &quot;ralt&quot;</td>
</tr>
<tr>
<td>spacebar</td>
<td>&quot;space&quot;</td>
</tr>
<tr>
<td>-</td>
<td>&quot;minus&quot;</td>
</tr>
<tr>
<td>=</td>
<td>&quot;equals&quot;</td>
</tr>
<tr>
<td>Backspace</td>
<td>&quot;backspace&quot;</td>
</tr>
<tr>
<td>[ ]</td>
<td>&quot;lbracket&quot;, &quot;rbracket&quot;</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;backslash&quot;</td>
</tr>
<tr>
<td>;</td>
<td>&quot;semicolon&quot;</td>
</tr>
<tr>
<td>'</td>
<td>&quot;apostrophe&quot;</td>
</tr>
<tr>
<td>Enter</td>
<td>&quot;enter&quot;</td>
</tr>
<tr>
<td>,</td>
<td>&quot;comma&quot;</td>
</tr>
<tr>
<td>-</td>
<td>&quot;period&quot;</td>
</tr>
<tr>
<td>/</td>
<td>&quot;slash&quot;</td>
</tr>
<tr>
<td>Home</td>
<td>&quot;home&quot;</td>
</tr>
<tr>
<td>End</td>
<td>&quot;end&quot;</td>
</tr>
<tr>
<td>Delete</td>
<td>&quot;delete&quot;</td>
</tr>
<tr>
<td>PageUp</td>
<td>&quot;pgup&quot;</td>
</tr>
<tr>
<td>PageDown</td>
<td>&quot;pgdn&quot;</td>
</tr>
<tr>
<td>Insert</td>
<td>&quot;insert&quot;</td>
</tr>
<tr>
<td>ScrollLock</td>
<td>&quot;scrolllock&quot;</td>
</tr>
<tr>
<td>PrintScreen</td>
<td>&quot;print&quot;</td>
</tr>
<tr>
<td>Pause/Break</td>
<td>&quot;pause&quot;</td>
</tr>
</tbody>
</table>

### Mouse Gestures

<table>
<thead>
<tr>
<th>Gesture</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left/primary mouse button</td>
<td>&quot;mouse1&quot;</td>
</tr>
<tr>
<td>Right/secondary mouse button</td>
<td>&quot;mouse2&quot;</td>
</tr>
<tr>
<td>Mouse wheel up</td>
<td>&quot;mwheel_up&quot;</td>
</tr>
<tr>
<td>Mouse wheel down</td>
<td>&quot;mwheel_down&quot;</td>
</tr>
<tr>
<td>New position along x-axis</td>
<td>&quot;maxis_x&quot;</td>
</tr>
<tr>
<td>New position along y-axis</td>
<td>&quot;maxis_y&quot;</td>
</tr>
</tbody>
</table>
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. 276)
- CryString (p. 301)
- ICrySizer (p. 302)
- Serialization Library (p. 302)

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. 282) 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. 291). 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. 293).

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.

```csharp
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.

```csharp
Public Module CryGUIDGenModule

    Sub GenerateCryGUID()
        Dim newGuid As System.Guid
        newGuid = System.Guid.NewGuid()
    End Sub
```

Version 1.4

277
Dim guidStr As String

    guidStr = newGuid.ToString("N")
    guidStr = guidStr.Insert(16, ", 0x")
    guidStr = guidStr.Insert(0, "0x")

    REM guidStr = guidStr + vbCrLf
    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. 299).

ICryUnknown is declared as follows.

```plaintext
struct ICryUnknown
{
    CRYINTERFACE_DECLARE(ICryUnknown, 0x100000010001000, 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. 280). 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. 281). 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.

```cpp
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.

```c++
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.

```c++
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. 277).

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.

```cpp
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.

```cpp
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.

```cpp
IMyExtensionPtr pMyExtension = ...;
ICryUnknownPtr pCompUnk = crycomposite_query(pMyExtension, "foo");
IFooPtr pComposite = cryinterface_cast<IFoo>(pCompUnk);
if (pComposite)
{
```
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.

```cpp
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
    }
}
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. 277).

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. 291).

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.

Caution
   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)
Constructs a CryGUID. Useful when searching the registry for extensions by class ID.

Parameters
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

```cpp
/////////////////////////////////////////////////////////////////////////
// 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()
```
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.
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.

```cpp
// 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.

```cpp
///////////////////////////////////////////
// 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"
```
The following example shows how the extension class `MyExtension` can be customized and expanded to implement two more interfaces, `IFoo` and `IBar`.

```cpp
CRYREGISTER_CLASS(CMyExtension)

CMyExtension::CMyExtension()
{
}

CMyExtension::~CMyExtension()
{
}

void CMyExtension::CallMe()
{
    printf("Inside CMyExtension::CallMe()...\n");
}

The following example shows how the extension class `MyExtension` can be customized and expanded to implement two more interfaces, `IFoo` and `IBar`.

```cpp
// 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)
}```
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. 278) and ICryFactory (p. 279).

### 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. 278) and ICryFactory (p. 279).
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, 0x88ff91b6b7be2afull};
return iid;
}

public:
virtual void TellMeWhyIDontLikeMacros() = 0;

typedef boost::shared_ptr<INoMacros> INoMacrosPtr;

// 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;
umIIDs = 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.

```cpp
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, 0x9df6c6c0d1c6d0)
    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,
                    0x9287dcd7e26a3a5)

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.

```cpp
///////////////////////////////////////////
// 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.

(Xml code)

#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

#include <IMyExtension.h>
#include <CryExtension/Impl/ClassWeaver.h>

class CMyExtension : public IMyExtension
{
    ...
};

#endif // #if defined(INCLUDE_MYEXTENSION)

// MyExtension.cpp
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. 288).

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.

```cpp
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

```cpp
#include "Serialization/IArchive.h"
#include "Serialization/STL.h"

enum AttachmentType
{
    ATTACHMENT_SKIN,
    ATTACHMENT_BONE
};
```
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.

```cpp
#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`.

```cpp
#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.

```cpp
#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.

```cpp
#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:

```cpp
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:

```cpp
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:

```cpp
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.
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 the 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.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Read-only field</td>
<td>Prevents the user from changing the value of the property. The effect is non-recursive.</td>
</tr>
</tbody>
</table>
### Prefixes and Roles

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Inline</td>
<td>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.</td>
</tr>
<tr>
<td>^^</td>
<td>Inline in front of a name</td>
<td>Places the property name before the name of the parent structure. Useful to add check boxes before a name.</td>
</tr>
<tr>
<td>&lt;</td>
<td>Expand value field</td>
<td>Expand the value part of the property to occupy all available space.</td>
</tr>
<tr>
<td>&gt;</td>
<td>Contract value field</td>
<td>Reduces the width of the value field to the minimum. Useful to restrict the width of inline fields.</td>
</tr>
<tr>
<td>&gt;N&gt;</td>
<td>Limit field width to N pixels</td>
<td>Useful for finer control over the UI. Not recommended for use outside of the editor.</td>
</tr>
<tr>
<td>+</td>
<td>Expand row by default.</td>
<td>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.</td>
</tr>
</tbody>
</table>

### Combining control characters

Multiple control characters can be put together to combine their effects, as in the following example.

```cpp
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.

<table>
<thead>
<tr>
<th>Decorator</th>
<th>Purpose</th>
<th>Defined for types</th>
<th>Context needed</th>
</tr>
</thead>
</table>
| AnimationPath | Selection UI for full animation path. | Any string-like type, like:
  - `std::string`
  - `string (CryStringT)`
  - `SCRCRef`
  - `CCryName` |                |
<table>
<thead>
<tr>
<th>Decorator</th>
<th>Purpose</th>
<th>Defined for types</th>
<th>Context needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>CharacterPath</td>
<td>UI: browse for character path (cdf)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CharacterRigPath</td>
<td>UI: browse for .rig files.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SkeletonPath</td>
<td>UI: browse for .chr or .skel files.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JointName</td>
<td>UI: list of character joints</td>
<td>ICharacter-Instance*</td>
<td></td>
</tr>
<tr>
<td>AttachmentName</td>
<td>UI: list of character attachments</td>
<td>ICharacter-Instance*</td>
<td></td>
</tr>
<tr>
<td>SoundName</td>
<td>UI: list of sounds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParticleName</td>
<td>UI: particle effect selection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Serialization/Decorators/Math.h</td>
<td>Edit or store radians as degrees</td>
<td>float, Vec3</td>
<td></td>
</tr>
<tr>
<td>RadiansAsDeg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Serialization/Decorators/Range.h</td>
<td>Sets soft or hard limits for numeric values and provides a slider UI.</td>
<td>Numeric types</td>
<td></td>
</tr>
<tr>
<td>Range</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Serialization/Call-back.h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Callback</td>
<td>Provides per-property callback function. See Adding callbacks to the PropertyTree (p. 310).</td>
<td>All types apart from compound ones (structs and containers)</td>
<td></td>
</tr>
</tbody>
</table>

**Decorator example**

The following example uses the `Range` and `CharacterPath` decorators.

```cpp
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.

```cpp
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.

```cpp
// CContextList m_contextList;
QPropertyTree* 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 a 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.**

```cpp
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.**

```cpp
#include <Serialization/Callback.h>  // located in Editor/Include
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.

```cpp
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.

```cpp
#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.

```cpp
void IArchive::Doc(const char*)
```

```cpp
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.

```cpp
template<class T> void IArchive::Warning(T& instance, const char* format, ...)
template<class T> void IArchive::Error(T& instance, const char* format, ...)
```

```cpp
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 message](image)

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. 302) 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.

```cpp
// 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.

```cpp
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.
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. 314)
- Recording Time Demos (p. 318)

**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:

<table>
<thead>
<tr>
<th>r_CustomResPreview</th>
<th>Preview status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No preview</td>
</tr>
<tr>
<td>1</td>
<td>Scaled to match the size of the viewport</td>
</tr>
<tr>
<td>2</td>
<td>Cropped to the size of the viewport</td>
</tr>
</tbody>
</table>

### 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

\[
\text{step} = \frac{1 \text{ second}}{\text{number of frames}}
\]

A table of common time step values follows.

<table>
<thead>
<tr>
<th>FPS</th>
<th>Time Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 (PAL)</td>
<td>0.04</td>
</tr>
<tr>
<td>30</td>
<td>0.033333333</td>
</tr>
<tr>
<td>60</td>
<td>0.016666667</td>
</tr>
</tbody>
</table>

### 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.

**Caution**

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:

```csharp
#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:

```csharp
#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:

```plaintext
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:

```csharp
#Sound.DeactivateAudioDevice()
s_OutputConfig 3
#Sound.ActivateAudioDevice()
Capture_frames 1
```

- To stop recording, use a config file like the following:

```csharp
#Sound.DeactivateAudioDevice()
s_OutputConfig 0
#Sound.ActivateAudioDevice()
Capture_frames 0
```
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 been 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

<table>
<thead>
<tr>
<th>Command</th>
<th>Keystroke</th>
<th>Console Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Recording</td>
<td><code>Ctrl + PrintScreen</code></td>
<td>record</td>
</tr>
<tr>
<td>End Recording</td>
<td><code>Ctrl + Break</code></td>
<td>stoprecording</td>
</tr>
<tr>
<td>Start Playback</td>
<td><code>Shift + PrintScreen</code></td>
<td>demo</td>
</tr>
<tr>
<td>Stop Playback</td>
<td><code>Ctrl + Break</code></td>
<td>stopdemo</td>
</tr>
</tbody>
</table>

Related Console Variables

- `stopdemo` - Stop playing a time demo.
• demo `demoname` - Play the time demo from the file specified
• demo `fixed_timestep` - Specify the number of updates per second.
• demo `panoramic` - Use a panoramic view when playing back the demo.
• demo `restart_level N` - Restart the level after each loop. Possible values for N: 0 = Off; 1 = use quicksave on first playback; 2 = load level start.
• demo `ai` - Enable or disable AI during the demo.
• demo `savestats` - Save level stats at the end of the loop.
• demo `max_frames` - Specify the maximum number of frames to save.
• demo `screenshot_frame N` - Make 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` - Quit the game after the demo run finishes.
• demo `noinfo` - Disable the information display during demo playback.
• demo `scroll_pause` - Enable the use of the ScrollLock key to pause demo play and record.
• demo `num_runs` - Specify the number of times to loop the time demo.
• demo `profile` - Enable demo profiling.
• demo `file` - Specify the time demo filename.
The Entity system is currently on a path to deprecation in favor of the Lumberyard Component Entity System (p. 255).

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. 320)
- Creating a New Entity Class (p. 321)
- Entity Pool System (p. 323)
- Entity ID Explained (p. 332)
- Adding Usable Support on an Entity (p. 333)
- Entity Scripting (p. 334)

## 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_POSITION },
{ "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 },
```
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.

```xml
<Entity
  Name="Fan"
  Version 1.4
  ...
  
  [ "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 ],
```
• Create a new Lua script file, for example GameSDK\Entities\Scripts\Fan.lua. The Lua file will define the entity logic.

```lua
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_para mTypes 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
```

Lumberyard Developer Guide
Creating a New Entity Class
self.currrotspeed = self.currrotspeed - System.GetFrameTime() * self.acceleration;
if (self.currrotspeed < self.minrotspeed) then
    self.currrotspeed = self.minrotspeed;
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. 324)
- Entity Pool Creation (p. 326)
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\EntityPoolDefinitions.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:

```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.

   ```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. 326).

### 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. 332) 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. 327)). 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. 327)).

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.

```cpp
bool CActor::GetEntityPoolSignature( TSerialize signature )
{
    signature.BeginGroup("Actor");
    signature.EndGroup();
    return true;
}
```

The section Debugging Utilities (p. 332) 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. 330).

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**.

```cpp
CEntityLoadManager::ParseEntities

SEntityLoadParams loadParams;
if (ExtractEntityLoadParams(entityNode, loadParams))
{
    if (bEnablePoolUse && loadParams.spawnParams.bCreatedThroughPool)
    {
        CEntityPoolManager *pPoolManager = m_pEntitySystem->GetEntityPoolManager();
        bSuccess = (pPoolManager && pPoolManager->AddPoolBookmark(loadParams));
    }
}
```
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: 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:

```cpp
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. 327)).
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 it's 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:

```lua
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. 334)
- Using Entity State (p. 337)
- Using Entity Slots (p. 338)
- Linking Entities (p. 339)
- Exposing an Entity to the Network (p. 340)

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:

```xml
<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).

```lua
LivingEntity = {
  Properties = {
    soclasses_SmartObjectClass = "",
    bMissionCritical = 0,
    bCanTriggerAreas = 1,
    object_Model = "objects/default/primitive_capsule.cgf",
    bExcludeCover=0,

    Physics = {
      bPhysicalize = 1, -- True if object should be physicalized at all.
      bPushableByPlayers = 1,
    },

    MultiplayerOptions = {
      bNetworked  = 0,
    },

    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). More information on Archetypes can be found in the 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**

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>boolean</td>
</tr>
<tr>
<td>f</td>
<td>float</td>
</tr>
<tr>
<td>i</td>
<td>integer</td>
</tr>
<tr>
<td>s</td>
<td>string</td>
</tr>
<tr>
<td>clr</td>
<td>color</td>
</tr>
<tr>
<td>object_</td>
<td>object compatible with Lumberyard (CFG, CGA, CHR or CDF file)</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>CGF model to be rendered over an entity instance.</td>
</tr>
<tr>
<td>ShowBounds</td>
<td>Flag indicating whether or not a bounding box is drawn around an entity instance when selected.</td>
</tr>
<tr>
<td>AbsoluteRadius</td>
<td></td>
</tr>
<tr>
<td>Icon</td>
<td>BMP icon to be drawn over an entity instance.</td>
</tr>
<tr>
<td>IconOnTop</td>
<td>Flag indicating whether or not the icon is drawn over or under an entity instance.</td>
</tr>
<tr>
<td>DisplayArrow</td>
<td></td>
</tr>
<tr>
<td>Links</td>
<td></td>
</tr>
</tbody>
</table>
**Functions**

A script entity can include several callback functions called by the engine or game system. See Entity System Script Callbacks (p. 130) 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. 130))

**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.

```lua
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:

```lua
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:**

---

Lumberyard Developer Guide
Using Entity State

---

Version 1.4
337
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.

```lua
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.

```lua
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.

```lua
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.

<table>
<thead>
<tr>
<th>Lumberyard resource</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>static geometry</td>
<td><code>LoadObject()</code> or <code>LoadSubObject()</code></td>
</tr>
<tr>
<td>animated character</td>
<td><code>LoadCharacter()</code></td>
</tr>
<tr>
<td>particle emitter</td>
<td><code>LoadParticleEffect()</code></td>
</tr>
<tr>
<td>light</td>
<td><code>LoadLight()</code></td>
</tr>
<tr>
<td>cloud</td>
<td><code>LoadCloud()</code></td>
</tr>
<tr>
<td>fog</td>
<td><code>LoadFogVolume()</code></td>
</tr>
<tr>
<td>volume</td>
<td><code>LoadVolumeObject()</code></td>
</tr>
</tbody>
</table>
Modifying Slot Parameters

Each of these resources 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.

```lua
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".

```lua
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);
            end
        end
        i=i+1;
    end
end
```
return true;
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.

```lua
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 },
  },
}
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.

<table>
<thead>
<tr>
<th>RMI attach flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_ATTACH</td>
<td>No special control (preferred)</td>
</tr>
<tr>
<td>PRE_ATTACH</td>
<td>Call occurs before data serialized</td>
</tr>
<tr>
<td>POST_ATTACH</td>
<td>Call occurs after the data serialized</td>
</tr>
</tbody>
</table>

The following example shows a function declaration:

```lua
function DeathMatch.Client:ClClientConnect(name, reconnect)
```

The following examples illustrate a function call:

```lua
self.allClients:ClVictory( winningPlayerId );

self.otherClients:ClClientConnect( channelId, player:GetName(), reconnect );

self.onClient:ClClientConnect( channelId, player:GetName(), reconnect );
```

See RMI Functions (p. 438) 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:

```cpp
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:

```cpp
CryAction.ForceGameObjectUpdate(self.id, true);
```

The script entity receive the `OnUpdate()` function callback of its Server table.

```lua
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 be 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. 343)
- Many Handlers (p. 344)
- EBus with Addresses and a Single Handler (p. 345)
- EBus with Addresses and Many Handlers (p. 346)

**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.

```cpp
// 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:

```cpp
// 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:
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 without an ID arrive at each address in no particular order.

```cpp
// 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.

```cpp
// 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`:

```cpp
// 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`. 
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. 353) 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. 349)
- EBus Configuration Options (p. 350)
- Implementing a Handler (p. 351)
- Sending Messages to an EBus (p. 352)
- Retrieving Return Values (p. 352)
- Return Values from Multiple Handlers (p. 352)
- Asynchronous/Queued Buses (p. 353)

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.

```cpp
class ExampleInterface : public AZ::EBusTraits
{
public:
    // ------------------ EBus Configuration -------------------
    // These override the defaults in EBusTraits.
    static const AZ::EBusHandlerPolicy HandlerPolicy = AZ::EBusHandlerPolicy::Single;
    static const AZ::EBusAddressPolicy AddressPolicy = AZ::EBusAddressPolicy::Single;

    // One handler per address is supported.
    ~ExampleInterface() override { }

    // ------------------------ Other -------------------------

    // Handlers inherit from ExampleInterfaceBus::Handler
    virtual void DoSomething() = 0;
    virtual void SomeMessage() { }

    // 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() { }
};
```
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. 343):

```c++
// 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 specifies only a single address. An ID is not required.

```c++
// The EBus contains a single address.
static const AZ::EBusAddressPolicy AddressPolicy = AZ::EBusAddressPolicy::Single;
```

A common use for a single address policy is a singleton pattern in which various systems post messages or requests to a single system elsewhere in the codebase.

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. 343) or many handlers (p. 344).
- **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. 343) or an EBus with addresses and a single handler (p. 345).
- **Multiple** – Any number of handlers are supported. Ordering is ignored. Uses include many handlers (p. 344) or an EBus with addresses and many handlers (p. 346).
- **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. 349). This means that the handler can be derived from ExampleInterfaceBus::Handler.

```cpp
#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 ReturnedValue(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 EBuses and start receiving events, your handler must call BusConnect():

```cpp
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 an ID.

```cpp
// 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:

```cpp
#include "ExampleInterface.h"
// note: We don't need to include MyHandler.h
...
...
EBUS_EVENT(ExampleInterfaceBus, DoSomething);

// calls the EBus without reading the result, packs 5 as the first parameter.
EBUS_EVENT(ExampleInterfaceBus, ReturnedValue, 5);
```

If your EBus is addressed, you can send events to a specific address ID. Events broadcast globally are received at all addresses.

```cpp
// broadcasts to ALL HANDLERS on the EBus regardless of address, even if the
// EBus has addresses
EBUS_EVENT(ExampleAddressBus, Test);

// broadcasts only to handlers connected to address 5.
EBUS_EVENT_ID(5, ExampleAddressBus, Test)
```

Retrieving Return Values

If you make a synchronous call (EBUS_EVENT), you can also supply a variable in which to place the result:

```cpp
// ALWAYS INITIALIZE YOUR RESULT!!!
// Since there may be nobody connected to the EBus, your result may not be populated.
bool result = false;
EBUS_EVENT_RESULT(result, ExampleInterfaceBus, ReturnedValue, 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:

```cpp
// Counterexample: returnValue contains only the result of the final handler.
bool returnValue = false;
EBUS_EVENT_RESULT(returnValue, SomeInterface::Bus, 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:

```cpp
#include <AZCore/EBus/Results.h>
...
AZ::EBusAggregateResults<bool> results;
EBUS_EVENT_RESULT(results, SomeInterfaceBus, DoesAnyoneObject);
// results now contains a vector of all results from all handlers.

// alternative:
AZ::EBusLogicalResult<bool, AZStd::logical_or<bool>> response(false);
EBUS_EVENT_RESULT(response, SomeInterfaceBus, DoesAnyoneObject);
// response now contains each result, using logical OR operation. So all responses are OR'd with each other.
```

**Note**
Additional building blocks (for example, arithmetic results) are available inside `results.h`.

### Asynchronous/Queued Buses

To declare an EBus on which events can be queued and sent asynchronously, add the following to the EBus declaration:

```cpp
static const bool EnableEventQueue = true;
```

You can use `EBUS_QUEUE_EVENT` and its variants to enqueue events on a 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:

```cpp
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. 354)
- Tracking File Access (p. 363)

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. 356)
- Reading Files with CryPak (p. 357)
- Writing to File System Files With CryPak (p. 360)
- Modifying Paks With CryArchive (p. 361)
- CryPak Details (p. 362)

### 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:
   
   This sample was read from the `.pak` archive

3. Save the file.
4. Inside the `GameSDK` directory, create a subfolder called `Examples`.
5. Add the `ExampleText.txt` file to the `Examples` folder so that the path looks like this:

   `<root>\GameSDK\Examples\ExampleText.txt`

6. 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.

   ```c
   char* fileContent = NULL;
   if (!ReadFromExampleFile(&fileContent))
   {
       CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "Read FromExampleFile() 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.

   ```c
   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(VAVALIDATOR_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 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()**

```c
char* fileContent = NULL;
if (!ReadFromExampleFile(&fileContent))
{
    CryWarning(VALIDATOR_MODULE_SYSTEM, VALIDATOR_WARNING, "ReadFromExampleFile() failed");
} else
{
    CryLogAlways("ExampleText contains %s", fileContent);
    delete[] fileContent;
}
```

**ReadFromExampleFile() implementation**

```c
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;
}
```
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. 361).

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.

   ```cpp
   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.

   ```cpp
   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;
       }
   ```
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(V

```

* 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::METH_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.

• 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.
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 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

```cpp
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 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_InvalidFileAccess 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. 365)
- TrueType Font Rendering (p. 369)
- Generating Stars DAT File (p. 370)
- Anti-Aliasing and Supersampling (p. 371)

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.

   ```cpp
   struct IPrismRenderNode : public IRenderNode
   {
   ...
   }
   ```

2. Add a new enum to the list of already defined render nodes in
   CryEngine/CryCommon/IEntityRenderState.h.

   ```cpp
eenum EERType
   {
   ...
   eERType_PrismObject,
   ...
   }
   ```

3. Add **PrismObjectRenderNode.h** to Cry3DEngine.

   ```cpp
   #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();
   ```
#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_Pris mObject);
    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 SRendering PassInfo &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, SRenderItemSorter(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
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.

```xml
<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>
    </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 \(<\text{pass}>\) element without child elements means that the pass is rendered with the default settings. The \(<\text{pos}>\) tag is used to offset the font, while the \(<\text{color}>\) 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 \( 0 \ldots 9 \) to set one of the 10 available colors. Use \$\$ to print the \$ symbol, and \$0 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.

- \texttt{r\_DebugFontRendering}  
  Provides information on various font rendering features, useful for verifying function and documenting usage.
  - 0=off
  - 1=display

- \texttt{r\_DumpFontNames}  
  Logs a list of fonts currently loaded.

- \texttt{r\_DumpFontTexture}  
  Dumps the texture of a specified font to a bitmap file. You can use \texttt{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 \texttt{Build\Engine\EngineAssets\Sky\stars.dat}. This data is loaded in the function \texttt{CStars::LoadData}, implemented in the file \texttt{CRSky.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)

<table>
<thead>
<tr>
<th>Name</th>
<th>Offset</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>0</td>
<td>uint32</td>
<td>0x52415453 (ASCII: STAR)</td>
</tr>
<tr>
<td>Version</td>
<td>4</td>
<td>uint32</td>
<td>0x00010001</td>
</tr>
<tr>
<td>NumStars</td>
<td>8</td>
<td>uint32</td>
<td>Number of star entries in the file</td>
</tr>
</tbody>
</table>

Entry (12 bytes)

<table>
<thead>
<tr>
<th>Name</th>
<th>Offset</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RightAscension</td>
<td>0</td>
<td>float32</td>
<td>in radians</td>
</tr>
<tr>
<td>Declination</td>
<td>4</td>
<td>float32</td>
<td>in radians</td>
</tr>
<tr>
<td>Red</td>
<td>8</td>
<td>uint8</td>
<td>star color, red channel</td>
</tr>
<tr>
<td>Green</td>
<td>9</td>
<td>uint8</td>
<td>star color, green channel</td>
</tr>
<tr>
<td>Blue</td>
<td>10</td>
<td>uint8</td>
<td>star color, blue channel</td>
</tr>
<tr>
<td>Magnitude</td>
<td>11</td>
<td>uint8</td>
<td>brightness, normalized range</td>
</tr>
</tbody>
</table>

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`.

<table>
<thead>
<tr>
<th>Mode</th>
<th>CVar Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No anti-aliasing</td>
<td>0</td>
<td>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.</td>
</tr>
<tr>
<td>SMAA_Low (1X)</td>
<td>1</td>
<td>Enables sub-pixel morphological anti-aliasing (SMAA), which removes jaggies (staircase artifacts) on polygon edges. This mode does not address sub pixel details.</td>
</tr>
<tr>
<td>SMAA_Med (1TX)</td>
<td>2</td>
<td>Enables SMAA with basic temporal re-projection to reduce pixel crawling.</td>
</tr>
<tr>
<td>SMAA_High (2TX)</td>
<td>3</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.
Lua Scripting

This section provides reference information and help with scripting in Lua. It also covers how to use tools including the Lua debugger and XML loader.

This section includes the following topics:

- Lua Scripting Reference (p. 374)
- Integrating Lua and C++ (p. 394)
- Lua Script Usage (p. 395)
- Using the Lua Remote Debugger (p. 396)
- Using the Lua XML Loader (p. 398)
- Recommended Reading (p. 401)

Lua Scripting Reference

This section contains reference information on Lua scripting functions. Functions are organized here based on the system they are defined in.

Node Index

Common Lua Globals and Functions (p. 376)

- Globals (p. 376)
- AIDebugToggle() (p. 377)
- ShowTime() (p. 377)
- count() (p. 377)
- new() (p. 378)
- merge() (p. 378)
- mergef() (p. 378)
- Vec2Str() (p. 378)
- LogError() (p. 378)
- LogWarning() (p. 379)
- Log() (p. 379)
- dump() (p. 379)
- EmptyString() (p. 379)
- NumberToBool() (p. 380)
- EntityName() (p. 380)
- EntityNamed() (p. 380)
- SafeTableGet() (p. 380)

**EntityUtils Lua Functions (p. 380)**

- DumpEntities() (p. 380)
- CompareEntitiesByName() (p. 381)
- CompareEntitiesByDistanceFromPoint() (p. 381)
- BroadcastEvent() (p. 381)
- MakeDerivedEntity() (p. 382)
- MakeDerivedEntityOverride() (p. 382)
- MakeUsable() (p. 382)
- MakePickable() (p. 383)
- MakeSpawnable() (p. 383)
- EntityCommon.PhysicalizeRigid() (p. 383)

**Math Lua Globals and Functions (p. 383)**

- Global Vectors (p. 383)
- Constants (p. 384)

**Vector functions**

- IsNullVector() (p. 384)
- IsNotNullVector() (p. 385)
- LengthSqVector() (p. 385)
- LengthVector() (p. 385)
- DistanceSqVectors() (p. 385)
- DistanceSqVectors2d() (p. 385)
- DistanceVectors() (p. 386)
- dotproduct3d() (p. 386)
- dotproduct2d() (p. 386)
- LogVec() (p. 386)
- ZeroVector() (p. 387)
- CopyVector() (p. 387)
- SumVectors() (p. 387)
- NegVector() (p. 387)
- SubVectors() (p. 387)
- FastSumVectors() (p. 388)
- DifferenceVectors() (p. 388)
- FastDifferenceVectors() (p. 388)
- ProductVectors() (p. 388)
• FastProductVectors() (p. 388)
• ScaleVector() (p. 389)
• ScaleVectorInPlace(a,b) (p. 389)
• NormalizeVector() (p. 389)
• VecRotate90_Z() (p. 390)
• VecRotateMinus90_Z() (p. 390)
• crossproduct3d() (p. 390)
• RotateVectorAroundR() (p. 390)
• ProjectVector() (p. 390)

Utility Functions
• LerpColors() (p. 391)
• Lerp() (p. 391)
• __max() (p. 391)
• __min() (p. 392)
• clamp() (p. 392)
• Interpolate() (p. 392)
• sgn() (p. 392)
• sgnnz() (p. 392)
• sqr() (p. 393)
• randomF() (p. 393)
• iff() (p. 393)
• DistanceLineAndPoint() (p. 391)

Physics Lua Functions (p. 393)
• Physics.RegisterExplosionShape() (p. 393)
• Physics.RegisterExplosionCrack() (p. 394)

Common LuaGlobals and Functions

Common Lua Globals and Functions

• File location: Game/Scripts/common.lua
• Loaded from: Game/Scripts/main.lua

Globals

Use the following globals to avoid temporary Lua memory allocations:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>g_SignalData_point</td>
<td>Basic 3D vector value used by g_SignalData.</td>
</tr>
<tr>
<td>g_SignalData_point2</td>
<td>Basic 3D vector value used by g_SignalData.</td>
</tr>
<tr>
<td>g_SignalData</td>
<td>Used to pass signal data in AI behavior scripts (see: Signals (p. 87)).</td>
</tr>
<tr>
<td>g_StringTemp1</td>
<td>Commonly used for temporary strings inside Lua functions.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>g_HitTable</td>
<td>Commonly used by the Physics.Raycast function.</td>
</tr>
</tbody>
</table>

A **g_HitTable** used with `Physics.Raycast` can contain the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pos</td>
<td>3D vector world coordinates of the ray hit.</td>
</tr>
<tr>
<td>normal</td>
<td>3D normal vector of the ray hit.</td>
</tr>
<tr>
<td>dist</td>
<td>Distance of the ray hit.</td>
</tr>
<tr>
<td>surface</td>
<td>Type of surface hit.</td>
</tr>
<tr>
<td>entity</td>
<td>Script table of entity hit (if one was hit).</td>
</tr>
<tr>
<td>renderNode</td>
<td>Script handle to a foliage or static render node.</td>
</tr>
</tbody>
</table>

A **g_SignalData** table can contain the following parameter types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vec3</td>
<td>3D vector.</td>
</tr>
<tr>
<td>ScriptHandle</td>
<td>Normally used to pass along an entity ID.</td>
</tr>
<tr>
<td>Floating Point</td>
<td>Floating point value.</td>
</tr>
<tr>
<td>Integer</td>
<td>Integer or number value.</td>
</tr>
<tr>
<td>String</td>
<td>String value.</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_tbl</td>
<td>Table to retrieve the number of key-value pairs from.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_obj</td>
<td>Existing table you want to create a new one from.</td>
</tr>
<tr>
<td>norecurse</td>
<td>Flag indicating whether or not to recursively recreate all sub-tables. If set to TRUE, sub-tables will not be recreated.</td>
</tr>
</tbody>
</table>

merge()

Merges two tables without merging functions from the source table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dst</td>
<td>Destination table to merge source table information into.</td>
</tr>
<tr>
<td>src</td>
<td>Source table to merge table information from.</td>
</tr>
<tr>
<td>recurse</td>
<td>Flag indicating whether or not to recursively merge all sub-tables.</td>
</tr>
</tbody>
</table>

mergef()

Merges two tables including merging functions from the source table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dst</td>
<td>Destination table to merge source table information into.</td>
</tr>
<tr>
<td>src</td>
<td>Source table to merge table information from.</td>
</tr>
<tr>
<td>recursive</td>
<td>Flag indicating whether or not to recursively merge all sub-tables.</td>
</tr>
</tbody>
</table>

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).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec</td>
<td>3D vector table to convert. Example: (x=1,y=1,z=1).</td>
</tr>
</tbody>
</table>

LogError()

Logs an error message to the console and the log file. Message appears in red text in the console.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmt</td>
<td>Formatted message string.</td>
</tr>
</tbody>
</table>
LogWarning()

Logs a warning message to the console and the log file. Message appears in yellow text in the console.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmt</td>
<td>Formatted message string.</td>
</tr>
<tr>
<td>...</td>
<td>Optional argument list. For example: LogWarning(&quot;MyError: %f&quot;, math.pi);</td>
</tr>
</tbody>
</table>

Log()

Logs a message to the console and the log file. Commonly used for debugging purposes.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmt</td>
<td>Formatted message string.</td>
</tr>
<tr>
<td>...</td>
<td>Optional argument list. For example: Log(&quot;MyLog: %f&quot;, math.pi);</td>
</tr>
</tbody>
</table>

dump()

Dumps information from a specified table to the console.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_class</td>
<td>Table to dump to console. For example: g_localActor</td>
</tr>
<tr>
<td>no_func</td>
<td>Flag indicating whether or not to dump the table functions.</td>
</tr>
<tr>
<td>depth</td>
<td>Depth of the tables tree dump information from.</td>
</tr>
</tbody>
</table>

EmptyString()

Checks whether or not a given string is set and its length is greater than zero. Returns TRUE or FALSE.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>String to check for.</td>
</tr>
</tbody>
</table>
**NumberToBool()**

Checks whether or not a number value is true (non-zero) or false (zero).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>Number to check for.</td>
</tr>
</tbody>
</table>

**EntityName()**

Retrieves the name of a specified entity ID or entity table. If the entity doesn’t exist, this function returns an empty string.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entity</td>
<td>Entity table or entity ID to return a name for.</td>
</tr>
</tbody>
</table>

**EntityNamed()**

Checks whether or not an entity with the specified name exists in the entity system. Returns **true** or **false**. Commonly used for debugging.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of entity to check for.</td>
</tr>
</tbody>
</table>

**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**.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>Table to check for the existence of a sub-table.</td>
</tr>
<tr>
<td>name</td>
<td>Sub-table name to check for.</td>
</tr>
</tbody>
</table>

**EntityUtils Lua Functions**

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:
CompareEntitiesByName()

Compares two entities identified by name. This function is commonly used when sorting tables.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ent1</td>
<td>Name of first entity table.</td>
</tr>
<tr>
<td>ent2</td>
<td>Name of second entity table.</td>
</tr>
</tbody>
</table>

Example

```lua
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**.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ent1</td>
<td>Entity 1 table</td>
</tr>
<tr>
<td>ent2</td>
<td>Entity 2 table</td>
</tr>
<tr>
<td>point</td>
<td>3D position vector identifying the point to measure distance to.</td>
</tr>
</tbody>
</table>

Example

```lua
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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sender</td>
<td>Entity that sent the event.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>event</td>
<td>String based entity event to process.</td>
</tr>
</tbody>
</table>

**Example**

```lua
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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_DerivedClass</td>
<td>Derived class table.</td>
</tr>
<tr>
<td>_Parent</td>
<td>Parent or base class table.</td>
</tr>
</tbody>
</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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_DerivedClass</td>
<td>Derived class table.</td>
</tr>
<tr>
<td>_Parent</td>
<td>Parent or base class table.</td>
</tr>
</tbody>
</table>

**MakeUsable()**

Adds usable functionality, such as an **OnUsed** event, to a specified entity.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entity</td>
<td>Entity table to make usable.</td>
</tr>
</tbody>
</table>

**Example**

```lua
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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entity</td>
<td>Entity table to make pickable.</td>
</tr>
</tbody>
</table>

**MakeSpawnable()**

Adds spawn functionality to a specified entity. Commonly used for AI actors during creation.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entity</td>
<td>Entity table to make spawnable.</td>
</tr>
</tbody>
</table>

**EntityCommon.PhysicalizeRigid()**

Physicalizes an entity based on the specified entity slot and its physics properties.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entity</td>
<td>Entity table to physicalize.</td>
</tr>
<tr>
<td>nSlot</td>
<td>Entity slot to physicalize.</td>
</tr>
<tr>
<td>Properties</td>
<td>Physics properties table</td>
</tr>
<tr>
<td>bActive</td>
<td>Not used.</td>
</tr>
</tbody>
</table>

**Math Lua Globals and Functions**

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:

<table>
<thead>
<tr>
<th>Global Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>g_Vectors.v000</td>
<td>Basic zero vector.</td>
</tr>
<tr>
<td>g_Vectors.v001</td>
<td>Positive z-axis direction vector.</td>
</tr>
<tr>
<td>g_Vectors.v010</td>
<td>Positive y-axis direction vector.</td>
</tr>
<tr>
<td>g_Vectors.v100</td>
<td>Positive x-axis direction vector.</td>
</tr>
</tbody>
</table>

---

Lumberyard Developer Guide
Math Lua Globals and Functions
## Global Name

<table>
<thead>
<tr>
<th>Global Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>g_Vectors.v101</td>
<td>The x and z-axis direction vector.</td>
</tr>
<tr>
<td>g_Vectors.v110</td>
<td>The x and y-axis direction vector.</td>
</tr>
<tr>
<td>g_Vectors.v111</td>
<td>The x, y and z-axis vector.</td>
</tr>
<tr>
<td>g_Vectors.up</td>
<td>Positive z-axis direction vector.</td>
</tr>
<tr>
<td>g_Vectors.down</td>
<td>Negative z-axis direction vector.</td>
</tr>
<tr>
<td>g_Vectors.temp</td>
<td>Temporary zero vector.</td>
</tr>
<tr>
<td>g_Vectors.tempColor</td>
<td>Temporary zero vector. Commonly used for passing rgb color values.</td>
</tr>
<tr>
<td>g_Vectors.temp_v1</td>
<td>Temporary zero vector.</td>
</tr>
<tr>
<td>g_Vectors.temp_v2</td>
<td>Temporary zero vector.</td>
</tr>
<tr>
<td>g_Vectors.temp_v3</td>
<td>Temporary zero vector.</td>
</tr>
<tr>
<td>g_Vectors.temp_v4</td>
<td>Temporary zero vector.</td>
</tr>
<tr>
<td>g_Vectors.vec-MathTemp1</td>
<td>Temporary zero vector.</td>
</tr>
<tr>
<td>g_Vectors.vec-MathTemp2</td>
<td>Temporary zero vector.</td>
</tr>
</tbody>
</table>

## Constants

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>g_Rad2Deg</td>
<td>Basic radian-to-degree conversion value.</td>
</tr>
<tr>
<td>g_Deg2Rad</td>
<td>Basic degree-to-radian conversion value.</td>
</tr>
<tr>
<td>g_Pi</td>
<td>Basic Pi constant based on math.pi.</td>
</tr>
<tr>
<td>g_2Pi</td>
<td>Basic double-Pi constant based on math.pi.</td>
</tr>
<tr>
<td>g_Pi2</td>
<td>Basic half-Pi constant based on math.pi.</td>
</tr>
</tbody>
</table>

## IsNullVector()

Checks whether or not all components of a specified vector are null.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Vector to check.</td>
</tr>
</tbody>
</table>
**IsNotNullVector()**

Checks whether or not any components of a specified vector is not null.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Vector to check.</td>
</tr>
</tbody>
</table>

**LengthSqVector()**

Retrieves the squared length of a specified vector.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Vector to retrieve length for.</td>
</tr>
</tbody>
</table>

**LengthVector()**

Retrieves the length of a specified vector.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Vector to retrieve length for.</td>
</tr>
</tbody>
</table>

**DistanceSqVectors()**

Retrieves the squared distance between two vectors.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>First vector.</td>
</tr>
<tr>
<td>b</td>
<td>Second vector.</td>
</tr>
</tbody>
</table>

**DistanceSqVectors2d()**

Retrieves the squared distance between two vectors in 2D space (without z-component).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>First vector.</td>
</tr>
<tr>
<td>b</td>
<td>Second vector.</td>
</tr>
</tbody>
</table>
DistanceVectors()
Retrieves the distance between two vectors.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>First vector.</td>
</tr>
<tr>
<td>b</td>
<td>Second vector.</td>
</tr>
</tbody>
</table>

dotproduct3d()
Retrieves the dot product between two vectors.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>First vector.</td>
</tr>
<tr>
<td>b</td>
<td>Second vector.</td>
</tr>
</tbody>
</table>

dotproduct2d()
Retrieves the dot product between two vectors in 2D space (without z-component).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>First vector.</td>
</tr>
<tr>
<td>b</td>
<td>Second vector.</td>
</tr>
</tbody>
</table>

LogVec()
Logs a specified vector to console.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Descriptive name of the vector.</td>
</tr>
<tr>
<td>v</td>
<td>Vector to log.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
<td>Vector to zero out.</td>
</tr>
</tbody>
</table>

**CopyVector()**

Copies the components of one vector to another.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
<td>Destination vector.</td>
</tr>
<tr>
<td>src</td>
<td>Source vector.</td>
</tr>
</tbody>
</table>

**SumVectors()**

Adds up the components of two vectors.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>First vector.</td>
</tr>
<tr>
<td>b</td>
<td>Second vector.</td>
</tr>
</tbody>
</table>

**NegVector()**

Negates all components of a specified vector.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Vector to negate.</td>
</tr>
</tbody>
</table>

**SubVectors()**

Copies the componentwise subtraction of two vectors to a destination vector.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
<td>Destination vector.</td>
</tr>
<tr>
<td>a</td>
<td>First vector.</td>
</tr>
<tr>
<td>b</td>
<td>Second vector.</td>
</tr>
</tbody>
</table>
**FastSumVectors()**
Copies the componentwise addition of two vectors to a destination vector.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
<td>Destination vector.</td>
</tr>
<tr>
<td>a</td>
<td>First vector.</td>
</tr>
<tr>
<td>b</td>
<td>Second vector.</td>
</tr>
</tbody>
</table>

**DifferenceVectors()**
Retrieves the difference between two vectors.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>First vector.</td>
</tr>
<tr>
<td>b</td>
<td>Second vector.</td>
</tr>
</tbody>
</table>

**FastDifferenceVectors()**
Copies the componentwise difference between two vectors to a destination vector.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
<td>Destination vector.</td>
</tr>
<tr>
<td>a</td>
<td>First vector.</td>
</tr>
<tr>
<td>b</td>
<td>Second vector.</td>
</tr>
</tbody>
</table>

**ProductVectors()**
Retrieves the product of two vectors.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>First vector.</td>
</tr>
<tr>
<td>b</td>
<td>Second vector.</td>
</tr>
</tbody>
</table>

**FastProductVectors()**
Copies the product of two vectors to a destination vector.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
<td>Destination vector.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>a</td>
<td>First vector.</td>
</tr>
<tr>
<td>b</td>
<td>Second vector.</td>
</tr>
</tbody>
</table>

**ScaleVector()**

Scales a specified vector \( a \) by a factor of \( b \).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Vector.</td>
</tr>
<tr>
<td>b</td>
<td>Scalar.</td>
</tr>
</tbody>
</table>

**ScaleVectorInPlace(a,b)**

Retrieves a new vector based on a copy of vector \( a \) scaled by a factor \( b \).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>First vector.</td>
</tr>
<tr>
<td>b</td>
<td>Scalar.</td>
</tr>
</tbody>
</table>

**ScaleVectorInPlace(dest,a,b)**

Copies vector \( a \) scaled by the factor of \( b \) to a destination vector.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
<td>Destination vector.</td>
</tr>
<tr>
<td>a</td>
<td>First vector.</td>
</tr>
<tr>
<td>b</td>
<td>Scalar.</td>
</tr>
</tbody>
</table>

**NormalizeVector()**

Normalizes a specified vector.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Vector to normalize.</td>
</tr>
</tbody>
</table>
VecRotate90_Z()
Rotates a specified vector by 90 degree around the z-axis.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>Vector to rotate.</td>
</tr>
</tbody>
</table>

VecRotateMinus90_Z()
Rotates a specified vector by -90 degree around the z-axis.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>Vector to rotate.</td>
</tr>
</tbody>
</table>

crossproduct3d()
Copies the result of the cross product between two vectors to a destination vector.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
<td>Destination vector.</td>
</tr>
<tr>
<td>p</td>
<td>First vector.</td>
</tr>
<tr>
<td>q</td>
<td>Second vector.</td>
</tr>
</tbody>
</table>

RotateVectorAroundR()
Copies to a destination vector the result of the vector rotation of vector p around vector r by a specified angle.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
<td>Destination vector.</td>
</tr>
<tr>
<td>p</td>
<td>First vector.</td>
</tr>
<tr>
<td>r</td>
<td>Second vector.</td>
</tr>
<tr>
<td>angle</td>
<td>Rotation angle.</td>
</tr>
</tbody>
</table>

ProjectVector()
Copies to a destination vector the result of the vector projection of vector P to the surface with a specified normal N.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
<td>Destination vector.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>P</td>
<td>Vector to project.</td>
</tr>
<tr>
<td>N</td>
<td>Surface normal.</td>
</tr>
</tbody>
</table>

**DistanceLineAndPoint()**

Retrieves the distance between point \( a \) and the line between \( p \) and \( q \).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Point to measure from.</td>
</tr>
<tr>
<td>p</td>
<td>Vector ( p ).</td>
</tr>
<tr>
<td>q</td>
<td>Vector ( q ).</td>
</tr>
</tbody>
</table>

**LerpColors()**

Performs linear interpolation between two color/vectors with a factor of \( k \).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Color/vector ( a ).</td>
</tr>
<tr>
<td>b</td>
<td>Color/vector ( b ).</td>
</tr>
<tr>
<td>k</td>
<td>Factor ( k ).</td>
</tr>
</tbody>
</table>

**Lerp()**

Performs linear interpolation between two scalars with a factor of \( k \).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Scalar ( a ).</td>
</tr>
<tr>
<td>b</td>
<td>Scalar ( b ).</td>
</tr>
<tr>
<td>k</td>
<td>Factor ( k ).</td>
</tr>
</tbody>
</table>

**__max()**

Retrieves the maximum of two scalars.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Scalar ( a ).</td>
</tr>
<tr>
<td>b</td>
<td>Scalar ( b ).</td>
</tr>
</tbody>
</table>
**__min()**
Retrieves the minimum of two scalars.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Scalar a.</td>
</tr>
<tr>
<td>b</td>
<td>Scalar b.</td>
</tr>
</tbody>
</table>

**clamp()**
Clamps a specified number between minimum and maximum.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_n</td>
<td>Number to clamp.</td>
</tr>
<tr>
<td>_min</td>
<td>Lower limit.</td>
</tr>
<tr>
<td>_max</td>
<td>Upper limit.</td>
</tr>
</tbody>
</table>

**Interpolate()**
Interpolates a number to a specified goal by a specified speed.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual</td>
<td>Number to interpolate.</td>
</tr>
<tr>
<td>goal</td>
<td>Goal.</td>
</tr>
<tr>
<td>speed</td>
<td>Interpolation speed.</td>
</tr>
</tbody>
</table>

**sgn()**
Retrieves the sign of a specified number (0 returns 0).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Number to get sign for.</td>
</tr>
</tbody>
</table>

**sgnnz()**
Retrieves the sign of a specified number (0 returns 1).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Number to get sign for.</td>
</tr>
</tbody>
</table>
sqr()

Retrieves the square of a specified number.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Number to square.</td>
</tr>
</tbody>
</table>

randomF()

Retrieves a random float value between two specified numbers.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>First number.</td>
</tr>
<tr>
<td>b</td>
<td>Second number.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Test value.</td>
</tr>
<tr>
<td>a</td>
<td>Return value if test value is not nil.</td>
</tr>
<tr>
<td>b</td>
<td>Return value if test value is nil.</td>
</tr>
</tbody>
</table>

Physics Lua Functions

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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sGeometryFile</td>
<td>Name of a boolean shape cfg file.</td>
</tr>
<tr>
<td>fSize</td>
<td>Shape's characteristic size.</td>
</tr>
<tr>
<td>BreakId</td>
<td>Breakability index (0-based) used to identify the breakable material.</td>
</tr>
</tbody>
</table>
### Integrating Lua and C++

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](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:

---

### Physics.RegisterExplosionCrack()

Registers a new explosion crack for breakable objects in the physics engine.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sGeometryFile</td>
<td>Name of a crack shape cgf file. This type of file must have three helpers to mark the corners, named “1”, “2” and “3”.</td>
</tr>
<tr>
<td>BreakId</td>
<td>Breakability index (0-based) used to identify the breakable material.</td>
</tr>
</tbody>
</table>

---

### Shape’s relative probability

Shape's relative probability; when several shapes with the same size appear as candidates for carving, these relative probabilities are used to select one.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fProbability</td>
<td>Name of a splinters cgf file, used for trees to add splinters at the breakage location.</td>
</tr>
<tr>
<td>fSplintersOffset</td>
<td>Size offset for the splinters.</td>
</tr>
<tr>
<td>sSplintersCloudEffect</td>
<td>Name of splinters particle fx; this effect is played when a splinters-based constraint breaks and splinters disappear.</td>
</tr>
</tbody>
</table>
Exposing Functions

To expose C++ functions to scripts, implement a new class derives from CScriptableBase, as shown in the following example.

class CScriptBind_Game :
    public CScriptableBase
{
public:
    CScriptBind_Game( ISystem* pSystem );
    virtual ~CScriptBind_Game() {}
    int GameLog(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");

Lua Script Usage

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++.
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. 394). Another option is to create a script entity, as described in Entity Scripting (p. 334).

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:

```lua
#Script.ReloadScript("Scripts\EntityCommon.lua")
```

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 127.0.0.1 (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:

![Scripts folder dialog]

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:

<table>
<thead>
<tr>
<th>To do this</th>
<th>Do this</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open a script file</td>
<td>Double click the script file in the navigation pane, or press Ctrl+O to open the Find File dialog.</td>
</tr>
<tr>
<td>Set a break point</td>
<td>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 program execution stops on a break point, the Call Stack and Locals tabs populate.</td>
</tr>
<tr>
<td>Remove a break point</td>
<td>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.</td>
</tr>
<tr>
<td>Use the Breakpoints tab</td>
<td>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.</td>
</tr>
<tr>
<td>To watch (inspect) variable values</td>
<td>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.</td>
</tr>
<tr>
<td>Pause execution</td>
<td>Click the pause (break) button on the toolbar or press Ctrl+Alt+Pause.</td>
</tr>
<tr>
<td>Resume execution</td>
<td>Click the play button on the toolbar or press F5.</td>
</tr>
<tr>
<td>Step over a procedure</td>
<td>Click the toolbar icon or press F10.</td>
</tr>
</tbody>
</table>
To do this | Do this
---|---
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

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:

```lua
numbers = {0,1,2,3,4,5,6,7,8,9}
```

In the XML data file, the array would look like this:

```xml
<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:

```xml
<Array name="numbers" type="int" elementName="number"/>
```

The second array type is an array of tables. In Lua:

```lua
wheels = {
  {size=3, weight=10},
  {size=2, weight=1},
  {size=4, weight=20},
}
```

In the XML data file:

```xml
<wheels>
  <wheel size="3" weight="10"/>
  <wheel size="2" weight="1"/>
  <wheel size="4" weight="20"/>
</wheels>
```

The XML definition file:

```xml
<Array name="wheels" elementName="wheel" />
  <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:
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:

```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:

```xml
(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:

```xml
<Definition root="Data">
  <Property name="version" type="string"/>
  <Table name="test">
    <Property name="a" type="string"/>
    <Property name="b" type="int" optional="1"/>
  </Table>
</Definition>
```
Corresponding XML data file:

```xml
<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>
```

**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
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 Project 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. 459).

Topics
- Getting Started with Multiplayer (p. 402)
- Session Service (p. 405)
- Carrier (p. 411)
- Synchronizing Game State Using Replicas (p. 414)
- Using OpenSSL (p. 431)
- Controlling Bandwidth Usage (p. 435)
- Network Serialization and Aspects (p. 437)
- RMI Functions (p. 438)

Getting Started with Multiplayer

This section describes how to set up a multiplayer server, host a game on it, connect to the game, and use console commands to control the server.

Setting up a Multiplayer Server

After you create and set up a multiplayer level and export the level to Lumberyard, you can run a server to host your new level.

Before you can host your new level, you must do the following:

- Ensure the same SDK build is installed on all the computers you will use to play the level.
• Copy the multiplayer level directory to each SDK installation where you will play.

Example: `<root>\Game\Levels`

• Identify the IP address of your host server, which will be used to allow other players to join your game.

Hosting a Level

Hosting a level on your server takes just a few steps.

To host a level on the server

1. On your host server, run the game executable.
2. From the console, run the command(s) for the task(s) you want to perform.
   • `mpstart <local_port>` — Initializes the multiplayer service. Replace `<local_port>` with the local UDP port you want to use to initialize the socket (optional). Port 64090 is used by default.
   • `mphost` — Hosts a session. The server listens for incoming connections on the port that you specified in `mpstart`.
   • `map <map_name>` — Launches the specified map. Replace `<map_name>` with the name of the map you want to use.

Connecting to the Server

After the server is running, you can connect to the hosted game from each of the computers you want to use to play the multiplayer game.

To connect to the server

1. On your computer, run the game executable.
2. From the console, run the commands for the tasks that you want to perform.
   • `mpstart <local_port>` — Initializes the multiplayer service. Replace `<local_port>` with the local UDP port you want to use to initialize the socket (optional). Port 64090 is used by default. To connect to a server on the same computer, replace `local_port` with 0 to use an ephemeral port.
   • `mpjoin <server_addr> <server_port>` — Join a game instance. The `<server_addr>` and `<server_port>` settings are optional. Localhost:64090 is used by default.

   Note
   When running a client on the same computer as the server, set the client to use an ephemeral port. Call `mpstart 0` in order to prevent port conflicts.

Console Command Summary

Use the following commands when working with a network server.

```
gm_debugdraw debug_draw_level
```

Set the debug draw level. Accepts as a parameter a number whose 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:

```cpp
class DebugDrawBits
{
  enum{Bit0   = BIT(0),
        Bit1   = BIT(1),
        Bit2   = BIT(2),
        Bit3   = BIT(3),
        Bit4   = BIT(4),
        Bit5   = BIT(5),

        Full   = Bit0 | Bit1 | Bit2 | Bit3 | Bit4,
        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>]**
  Initialize the network system and optionally set 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.
Sample Project

For an example of how to set up a multiplayer project, see Multiplayer Sample Project in the Amazon Lumberyard User Guide.

Session Service

GridMate’s session service provides session connectivity and management. Both hub-and-spoke (client/server) and P2P full-mesh topologies are supported, and multiple sessions can be created 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.

Starting and Stopping the Session Service

To start the session service in GridMate, call:

```cpp
GridMate::StartGridMateService<LANSessionService>(gridmate, GridMate::SessionServiceDesc());
```

To stop the service, call:

```cpp
IGridMate::UnregisterService();
```

This terminates all current sessions and searches.

Start session service example

The following LAN multiplayer service example starts the online service if it has not already been started.

```cpp
IGridMate* gridmate = gEnv->pNetwork->GetGridMate();
if (gridmate)
{
    GridMate::StartGridMateService<LANSessionService>(gridmate, GridMate::SessionServiceDesc());
}
```

Hosting a Session

To host a session, call `SessionService::HostSession()`. The function takes two arguments: `SessionParams` and `CarrierDesc`. The function returns a `GridSession` pointer.

`SessionParams` is used to configure the sessions topology, the number of slots, and the matchmaking properties. The local user to which to bind the current session is also set using `SessionParams`.

`CarrierDesc` is used to set the carrier parameters used for the current session. For `CarrierDesc` parameter descriptions, see CarrierDesc Parameters (p. 412).

The pointer returned by `SessionService::HostSession()` will remain valid until the `GridSessionCallbacks::OnSessionDelete()` event occurs.
Host session example

The following is an example component that creates a GridMate session and listens for clients that connect to the session.

```cpp
// GridMateHost
#include <AZCore/Component/Component.h>
#include <AZCore/Component/TickBus.h>
#include <GridMate/GridMate.h>
#include <GridMate/Session/Session.h>
#include <GridMate/Session/LANSession.h>

class HostComponent : public AZ::Component,
   public AZ::TickBus::Handler,
   public GridMate::SessionEventBus::Handler
{
    GridMate::IGridMate* mGridMate;
    GridMate::LANSessionService* mMultiplayerService;
    GridMate::GridSession* mSession;

public:
    AZ_COMPONENT(HostComponent, "{41900198-1122-4392-A579-CDDC85A23277}");

    HostComponent() : mSession(nullptr), mGridMate(nullptr), mMultiplayerService(nullptr) {};

    // Events from GridMate::SessionEventBus
    void OnSessionCreated(GridMate::GridSession* session) override {};  
    void OnSessionError(GridMate::GridSession* session, const GridMate::string& error) override {}; 
    void OnMemberJoined(GridMate::GridSession* session, GridMate::GridMember* member) {}; 
    void OnMemberLeaving(GridMate::GridSession* session, GridMate::GridMember* member) {};

    void Init() override {};
    void Activate() override {
        // The GridMate specific memory allocators must be created before setting up GridMate.
        // If multiple GridMate instances are to be created (e.g. one per component) this needs to be called at that time that the ComponentApplication is initialized.
        AZ::AllocatorInstance<GridMate::GridMateAllocator>::Create();
        AZ::AllocatorInstance<GridMate::GridMateAllocatorMP>::Create();

        // The port that all game data will be transmitted.
        static const AZ::s32 GAME_SEARCH_PORT = 40000;
        // The port that is listening for clients searching for games on the LAN.
        static const AZ::s32 GAME_PORT = 20000;

        // Initialize GridMate and services systems.
        mGridMate = GridMate::GridMateCreate(GridMate::GridMateDesc());
        mMultiplayerService = GridMate::StartGridMateService<LANSessionSer
```
Searching for Sessions

To search for available sessions, call `SessionService::StartGridSearch()`. This function returns a `GridSearch` pointer, which on search completion contains a list of found sessions. Search completion events are sent via `GridSessionCallbacks::OnGridSearchComplete()` with the `GridSearch` pointer returned by `SessionService::StartGridSearch()`.
To cancel a search, use `GridSearch::AbortSearch()`.

The `GridSearch` pointer remains valid until the `GridSessionCallbacks::OnGridSearchRelease()` event occurs.

### Joining a Session

Sessions can be joined by calling `SessionService::JoinSession()` with a `SearchInfo` pointer returned by a search. The function returns a `GridSession` pointer. When a client has successfully joined a session, an event is sent via `GridSessionCallbacks::OnSessionJoined()`.

The pointer returned by `SessionService::JoinSession()` remains valid until the `GridSessionCallbacks::OnSessionDelete()` event occurs.

### Join session example

The following example component searches for any GridMate session and joins the first one that it finds.

```cpp
// GridMateJoin
#include <AZCore/Component/Component.h>
#include <AZCore/Component/TickBus.h>
#include <GridMate/GridMate.h>
#include <GridMate/Session/LANSession.h>
#include <GridMate/Session/Session.h>

class JoinComponent
    : public AZ::Component
    , public AZ::TickBus::Handler
    , public GridMate::SessionEventBus::Handler
{
    GridMate::IGridMate* mGridMate;
    GridMate::LANSessionService* mMultiplayerService;
    GridMate::GridSession* mSession;
    GridMate::GridSearch* mSearch;

public:
    AZ_COMPONENT(JoinComponent, "{41900198-1122-4392-A579-CDDC85A23277}");

    JoinComponent() : mSession(nullptr), mGridMate(nullptr), mMultiplayerService(nullptr) {}

    // Events from GridMate::SessionEventBus
    void OnSessionJoined(GridMate::GridSession* session) override {};
    void OnSessionError(GridMate::GridSession* session, const GridMate::string& error) override {};
    void OnMemberJoined(GridMate::GridSession* session, GridMate::GridMember* member) {};
    void OnMemberLeaving(GridMate::GridSession* session, GridMate::GridMember* member) {};
    void OnGridSearchComplete(GridMate::GridSearch* search)
    {
        if(search->GetNumResults() > 0)
        {
            const GridMate::SearchInfo* result = search->GetResult(0);
            GridMate::CarrierDesc;
            carrierDesc.m_enableDisconnectDetection = true;
        }
    }
```
carrierDesc.m_connectionTimeoutMS = 10000;
carrierDesc.m_threadUpdateTimeMS = 30;
carrierDesc.m_version = 1;
carrierDesc.m_port = 0;

// Join the session.
// This operation is asynchronous and is only considered complete
when
// OnSessionJoined() or OnSessionError() events are called.
mSession = mMultiplayerService->JoinSession(result, GridMate::JoinParams(), carrierDesc);
}

void Init() override {

void Activate() override {
    
    // The GridMate specific memory allocators must be created before setting
    up GridMate.
    // If multiple GridMate instances are to be created (e.g. one per com
    ponent) this needs
    // to be called at that time that the ComponentApplication is initial
    ized.
    AZ::AllocatorInstance<GridMate::GridMateAllocator>::Create();
    AZ::AllocatorInstance<GridMate::GridMateAllocatorMP>::Create();

    // The port that all game data will be transmitted.
    static const AZ::s32 GAME_SEARCH_PORT = 40000;
    // The port that is listening for clients searching for games on the
    LAN.
    static const AZ::s32 GAME_PORT = 20000;

    // Initialize GridMate and services systems.
    mGridMate = GridMate::GridMateCreate(GridMate::GridMateDesc());

    // Attach this component to the event bus.
    AZ::TickBus::Handler::BusConnect();
    GridMate::SessionEventBus::Handler::BusConnect(mGridMate);

    // Search for games on the LAN.
    GridMate::LANSearchParams params;
    params.m_serverPort = GAME_SEARCH_PORT;
    mSearch = mMultiplayerService->StartGridSearch(&params);
}

void Deactivate() override {
    if(mSession)
        mSession->Leave(0);
    GridMate::SessionEventBus::Handler::BusDisconnect(mGridMate);
    AZ::TickBus::Handler::BusDisconnect();
    GridMate::GridMateDestroy(mGridMate);
    mMultiplayerService = nullptr;
    mGridMate = nullptr;
}

void OnTick(float dateTime, AZ::ScriptTimePoint t) {
    
    // Update the GridMate system each tick.
}
Reacting to Session Events

GridMate session events are sent via `GridMate::SessionEventBus`. To listen for session events, perform the following steps.

**To listen for session events**

1. Derive a listener from `SessionEventBus::Listener`.
2. Implement the `GridSessionCallbacks` interface.
3. Use the corresponding `GridMate` pointer to connect the listener.
4. To distinguish between the `GridSession` and `GridSearch` instances, pass the `GridSession` or `GridSearch` pointer together with the event.

**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.

  **Caution**
  - 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 will be left.

  **Caution**
  - 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.

Cloud Hosting with GameLift

Lumberyard supports cloud hosting of dedicated servers through GameLift.

To use GameLift in your project, enable the GameLift gem. For information about gems, see Gems in
the Amazon Lumberyard User Guide.

Developers can access GameLift directly through the C++ API, or by using the flow graph nodes provided
by the GameLift gem. For more information about GameLift, see in the Amazon GameLift Developer
Guide and the Amazon GameLift API Reference.

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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_address</td>
<td>const char *</td>
<td>Specifies the local communication address to which the driver will bind. A value of 0 specifies any address. The default is nullptr.</td>
</tr>
<tr>
<td>m_connectionEvaluationThreshold</td>
<td>float</td>
<td>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.</td>
</tr>
<tr>
<td>m_connectionTimeoutMS</td>
<td>unsigned int</td>
<td>Determines the time to allow for a connection attempt. The default is 5000 milliseconds.</td>
</tr>
<tr>
<td>m_disconnectDetectionPacketLossThreshold</td>
<td>float</td>
<td>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.</td>
</tr>
<tr>
<td>m_disconnectDetectionRttThreshold</td>
<td>float</td>
<td>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.</td>
</tr>
<tr>
<td>m_driver</td>
<td>class Driver *</td>
<td>Specifies a custom driver implementation. The default is nullptr.</td>
</tr>
</tbody>
</table>

Version 1.4

412
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_driverIsCrossPlatform</td>
<td>bool</td>
<td>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.</td>
</tr>
<tr>
<td>m_driverIsFullPackets</td>
<td>bool</td>
<td>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.</td>
</tr>
<tr>
<td>m_driverReceiveBufferSize</td>
<td>unsigned int</td>
<td>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.</td>
</tr>
<tr>
<td>m_driverSendBufferSize</td>
<td>unsigned int</td>
<td>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.</td>
</tr>
</tbody>
</table>
| 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. |
| 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. |
### Synchronizing Game State Using Replicas

Game sessions use replicas to synchronize the state of the session. To use a replica, you simply declare the states that need to be synchronized and the remote procedure calls (RPCs) that are supported. After you bind the replica object to the network, the engine does the heavy lifting. 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, and corresponding remote proxy replica objects are created on the remote nodes. Only the owner of the replica is allowed to change state, 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**

- Replicas (p. 414)
- Replica Chunks (p. 418)
- Data Sets (p. 420)
- Remote Procedure Calls (RPCs) (p. 422)
- Replica Manager (p. 423)
- Marshaling and Unmarshaling (p. 425)
- Binding Game Objects to the Network Using Components (p. 426)

### Replicas

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. 423).

---

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_threadPriority</td>
<td>int</td>
<td>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.</td>
</tr>
<tr>
<td>m_threadUpdateTimeMS</td>
<td>int</td>
<td>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.</td>
</tr>
<tr>
<td>m_trafficControl</td>
<td>class TrafficControl *</td>
<td>Specifies a custom traffic control implementation that controls traffic flow to all connections and that handles issues like network congestion.</td>
</tr>
<tr>
<td>m_version</td>
<td>Version-Type</td>
<td>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.</td>
</tr>
</tbody>
</table>
Replica Chunks

Every replica holds a collection of user-defined ReplicaChunk (p. 418) objects that are synchronized with all the peers in the current session. A replica chunk is a container for user-defined DataSet (p. 420) objects and Remote Procedure Calls (RPCs) (p. 422). 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. 418).

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. 405), and 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, to change the chunks' data sets 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. 418).

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:

```cpp
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:

```cpp
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 will not get 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:

```cpp
replica->Destroy();
```

Sample Code

This example creates a user-defined chunk, creates a replica, attaches the chunk to the replica, and binds the replica to the session replica manager.
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
    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
    // 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
// from
replica manager instance
    // 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 will be 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. 414), 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 Data Sets (p. 420) and/or Remote Procedure Calls (RPCs) (p. 422). 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`:

```cpp
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::Data
SetHandler> 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. 423).

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<MyReplicaChunk>({...});

Where {...} is forwarded to the MyReplicaChunk 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<MyReplicaChunk>(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.

Data Sets

You can use DataSet objects to synchronize the state of a session across the network. When a value
in the data set changes, the updates are propagated automatically. Data sets can be of any type, but
they must support the assignment and comparison operators. Your DataSet declaration can specify a
custom marshell. If you do not specify a marshell, the your 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 data set constructor.

The following example declares a **ReplicaChunk** object that has two **DataSet** objects of type **float**. One data set uses the default marshaler. The other data set uses a custom marshaler called MyCustomMarshaler.

```cpp
class MyChunkType : public GridMate::ReplicaChunk
{
    public:
        MyChunkType()
            : m_synchedFloat("SynchedFloat")
            , m_synchedHalf("SynchedHalf")
        {
        }
        GridMate::DataSet<float> m_synchedFloat;
        GridMate::DataSet<float, MyCustomMarshaler> m_synchedHalf;
};
```

Data sets can be optionally bound to a callback on the chunk interface so that the callback is called when new data arrives.

```cpp
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 data sets. Normally, data sets 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 data set.
2. The new value is broadcast to the remote peers.
3. The data set 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**:

```cpp
... GridMate::DataSet<Vector3> m_pos;
...
Remote Procedure Calls (RPCs)

Remote procedure calls (RPCs) allow games to send events or requests through replicas. RPCs are best suited for infrequent or one-off events or for directing messages to a specific node.

In GridMate, 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 do not receive RPCs that were requested before the clients joined.

Declaring RPCs

Like data sets, 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 it processes the request. If the handler function returns true when called on the master replica, the RPC is propagated to all the proxy replicas. The return value is ignored when called on proxies.

```cpp
class MyChunkType : public GridMate::ReplicaChunk
{
public:
  // Rpc1 takes two arguments: a float marshaled using default marshaler, and
  // another float marshaled using CustomFloatMarshaler.
  // Requests are handled by MyChunkType::HandleRpc1().
  GridMate::Rpc<GridMate::RpcArg<float>, GridMate::RpcArg<float, CustomFloatMarshaler>>::BindInterface<MyChunkType, &HandleRpc1> Rpc1;

  // Handler function bound to Rpc1.
  // Verifies the RPC and if accepted, returns true so it is propagated to all proxies.
  bool HandleRpc1(float v1, float v2, const GridMate::RpcContext& rpcContext)
  {
    // The RPC handler can perform additional checks to verify the request.
    if (VerifyRpcRequest())
    {
      printf("Execute Rpc1...
");

      // Return true to propagate to all proxies.
      return true;
    }
    // Return false to stop propagation to proxies.
    return false;
  }
};
```
Rpc2 has no arguments.
// Requests are handled by MyChunkType::HandleRpc2().
GridMate::Rpc<>::BindInterface<MyChunkType, &HandleRpc2, GridMate::RpcUnreliable> Rpc2;

// Handler function bound to Rpc2.
bool HandleRpc2(const GridMate::RpcContext& rpcContext)
{
    printf("Execute Rpc2...\n");
    // Never propagate Rpc2 to proxies.
    return false;
}

void Foo(MyChunkType* myChunkInstance)
{
    // Call Rpc1.
    myChunkInstance->Rpc1(1.f, 2.f);

    // CallRpc2.
    myChunkInstance->Rpc2();
}

Replica Manager

The replica manager is a sub-system 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 Lifetimes

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

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:

```cpp
GridMate::ReplicaManager* replicaManager = session->GetReplicaMgr(); // Get replica manager from the current session
```
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 until the user calls Destroy() on the replica, or until 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:

```cpp
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 and 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, including of its data sets (p. 420), RPCs (p. 422), 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 Marshaling and Unmarshaling (p. 425).
**Update From Replica: Updating proxy replicas**

A change in a custom `ReplicaChunk (p. 418)` 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 state.

**Marshaling and Unmarshaling**

As noted in Marshaling: Sending data to other peers (p. 424), marshalling is the sending of changes in replica data to every peer instance in the GridMate session. Data is written to the network using `WriteBuffer`, and data received is read using `ReadBuffer`. Each buffer specifies the endianness used.

Marshalers are responsible for reading and writing data to and from the buffers. Default marshalers are provided for most built-in data types, and other commonly used data types. Custom marshalers can be implemented to support custom types or custom compression techniques. Default marshalers are implemented by C++ template specialization:

```cpp
namespace GridMate
{
    template<typename T>
    class Marshaler
    {
        public:
            void Marshal(WriteBuffer& wb, const T& value);
            void Unmarshal(T& value, ReadBuffer& rb);
    
    }
}
```

If a marshaler 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`:

```cpp
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;
```
Binding Game Objects to the Network Using Components

Any `ComponentEntity` that shares data on the network must include the `NetBindingComponent`. The `NetBindingComponent` creates a replica (p. 414) for the `ComponentEntity` and can bind any replica chunk (p. 418) that a component creates to the replica. You can bind base Lumberyard components to the network by just adding the `NetBindingComponent`, but some extra steps are required if you are using custom components.

Custom Components

Use the following steps to enable networking on a custom component.

To enable networking on a custom component

1. Inherit the component from `AzFramework::NetBindable`:

   ```cpp
   #include <AzFramework/Network/NetBindable.h>
   class TransformComponent
   : public Component
   , public AzFramework::NetBindable
   ```

2. Modify the `AZ_COMPONENT` definition to include `AzFramework::NetBindable`:

   ```cpp
   AZ_COMPONENT(TransformComponent,"{22B10178-39B6-4C12-BB37-77DB45FDD3B6}",
   AzFramework::NetBindable);
   ```

   If `AzFramework::NetBindable` is missing, the `NetBindingComponent` will not discover the custom component when it checks for components that have chunks to be added to the replica.

   **Note**
   Changing this definition requires that you reexport the affected level.

3. Modify `SerializeContext` to include `AzFramework::NetBindable`:

   ```cpp
   serializeContext->Class<TransformComponent, AZ::Component, AzFramework::NetBindable>()
   ```

   If `AzFramework::NetBindable` is missing from the serialization definition, the master replica chunk will function correctly, but the proxy replica will not receive its proxy chunk.

   **Note**
   Changing this definition requires that you reexport the affected level.

4. Implement the following `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;

**Network Binding Function Details**

A number of functions are available for working with component entities on the network.

**GetNetworkBinding**

Call the `GetNetworkBinding` function only on the master `ComponentEntity`. This function returns a replica chunk and allows the component to initialize any state that the component needs to synchronize across the session. After the chunk is returned, the chunk is automatically attached to the appropriate replica.

**SetNetworkBinding**

Call the `SetNetworkBinding` function on all proxy `ComponentEntity` objects. The function passes a `ReplicaChunk` object to the component so the component can initialize its internal data to match that of the replica chunk received. The received chunk is already bound to the appropriate replica.

**UnbindFromNetwork**

Call the `UnbindFromNetwork` function to stop the component from reacting to data updates from the network—for example, when the master no longer exists, has been deactivated, or has relinquished control to the local source.

**Creating a Chunk**

Before you can fully enable the interface on the component, you must create a replica chunk to store whatever state the component has to share.

class TransformReplicaChunk : public GridMate::ReplicaChunk
{
public:
    AZ_CLASS_ALLOCATOR(TransformReplicaChunk, AZ::SystemAllocator, 0);
    static const char* GetChunkName() { return "TransformReplicaChunk"; }
    TransformReplicaChunk()
    : m_localTransform("LocalTransformData")
    , m_entityId() 
    { }

    void SetEntityId(EntityId entityId)
    { 
        m_entityId = entityId; 
    }
};
In order for the component to react to changes in the Dataset object, either the replica chunk must signal to the component when this change occurs (as is done here using the BindInterface extension to Dataset), or the component must poll the replica chunk and check the Dataset object for changes.

Notes

- You must register the new ReplicaChunk object. For more information, see Registering Chunk Type (p. 420).
- The ReplicaChunk object requires an empty constructor so that it can be unmarshalled.

Example: Filling out the AzFramework::NetBindable Interface

The examples below illustrate the use of GetNetworkBinding, SetNetworkBinding and UnbindFromNetwork.

GetNetworkBinding

```cpp
GridMate::ReplicaChunkPtr TransformComponent::GetNetworkBinding()
```
TransformReplicaChunk* replicaChunk = GridMate::CreateReplicaChunk<TransformReplicaChunk>();
replicaChunk->SetEntityId(GetEntityId());
m_replicaChunk = replicaChunk;
UpdateReplicaChunk();
return m_replicaChunk;
}

SetNetworkBinding

void TransformComponent::SetNetworkBinding(GridMate::ReplicaChunkPtr replicaChunk)
{
    AZ_Assert(m_replicaChunk == nullptr, "ERROR: Being bound to two replica chunks");
    bool isTransformChunk = replicaChunk != nullptr;
    AZ_Assert(isTransformChunk, "ERROR: Being bound to invalid chunk type");
    if (isTransformChunk)
    {
        m_replicaChunk = replicaChunk;
        TransformReplicaChunk* transformReplicaChunk = static_cast<TransformReplicaChunk*>(m_replicaChunk.get());
        transformReplicaChunk->SetEntityId(GetEntityId());
        UpdateFromReplicaChunk();
    }
}

UnbindFromNetwork

void TransformComponent::UnbindFromNetwork()
{
    m_replicaChunk = nullptr;
}

Maintaining State

The last step is to make sure that any local modifications to the desired networkable state do not overwrite the networked state, and to update the replica chunk when the component is in control of the state and the local state changes.

// Don't allow external setting of the Transform if it's network controlled.
void TransformComponent::SetLocalTM(const Transform& tm)
{
    if (!IsNetworkControlled())
    {
        ForceLocalTM(tm);
    }
}

// Helper function to let the replica bypass the previous safety check.
// Also signals the need to update the replica chunk with the new state.
void TransformComponent::ForceLocalTM(const Transform& tm)
{
    m_localTM = tm;
ComputeWorldTM(); // We can use dirty flags and compute it later on demand

UpdateReplicaChunk();
}

// Don't manipulate the local TM if it's network controlled.
// And in the case of the master, update the replica with
// the new information that changed here.
void TransformComponent::ComputeLocalTM()
{
    if (IsNetworkControlled())
    {
        return;
    }

    if (m_parentTM)
    {
        m_localTM = m_parentTM->GetWorldTM().GetInverseFull() * m_worldTM;
    }
    else
    {
        m_localTM = m_worldTM;
    }
    EBUS_EVENT_PTR(m_notificationBus, TransformNotificationBus, OnTransformChanged, m_localTM, m_worldTM);
    UpdateReplicaChunk();
}

// Helper method
bool TransformComponent::IsNetworkControlled() const
{
    return m_replicaChunk != nullptr && !m_replicaChunk->IsMaster();
}

// Helper method that updates the replica chunk's transform from the local
// transform.
void TransformComponent::UpdateReplicaChunk()
{
    if (m_replicaChunk && m_replicaChunk->IsMaster())
    {
        TransformReplicaChunk* transformReplicaChunk = static_cast<TransformReplicaChunk*>(m_replicaChunk.get());
        transformReplicaChunk->SetLocalTransform(GetLocalTM());
    }
}

// Helper method that updates the local transform from the replica chunk
void TransformComponent::UpdateFromReplicaChunk()
{
    if (m_replicaChunk && !m_replicaChunk->IsMaster())
    {
        TransformReplicaChunk* transformReplicaChunk = static_cast<TransformReplicaChunk*>(m_replicaChunk.get());
        ForceLocalTM(transformReplicaChunk->GetLocalTransform());
    }
}
Using OpenSSL

GridMate uses the OpenSSL implementation of Datagram Transport Layer Security (DTLS) to support encryption of all UDP traffic sent between clients and servers.

Topics
- Limitations (p. 431)
- Implementation Support (p. 431)
- Cipher (p. 431)
- Building with Encryption (p. 432)
- Enabling Encryption (p. 432)

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:

<table>
<thead>
<tr>
<th>Technology</th>
<th>Role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECDHE</td>
<td>Master key exchange</td>
<td>Ephemeral Elliptic Curve Diffie-Hellman anonymous key agreement protocol</td>
</tr>
<tr>
<td>RSA</td>
<td>Peer authentication</td>
<td>RSA algorithm used to authenticate client and server</td>
</tr>
<tr>
<td>AES256</td>
<td>Symmetric encryption cipher</td>
<td>Advanced Encryption Standard that uses a 256-bit master key</td>
</tr>
<tr>
<td>GCM</td>
<td>Block cipher mode of operation</td>
<td>Galois/Counter Mode authenticated encryption algorithm</td>
</tr>
<tr>
<td>SHA384</td>
<td>Hashing algorithm</td>
<td>SHA-2 with a 384-bit digest size</td>
</tr>
</tbody>
</table>
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']
```
To enable encryption in a GridMate session

1. To set the encryption parameters, create an instance of $\text{SecureSocketDesc}$. The parameters are described in $\text{SecureSocketDesc}$ (p. 433).

2. Create an instance of $\text{SecureSocketDriver}$ that passes in the instance of $\text{SecureSocketDesc}$. The instance of $\text{SecureSocketDesc}$ must be available for the duration of the GridMate session.

3. Before hosting or joining a GridMate session, define $\text{CarrierDesc}$ by setting the $\text{CarrierDesc::m\_driver}$ property to the instance of $\text{SecureSocketDriver}$. If no instance of $\text{SecureSocketDriver}$ is provided, an unencrypted driver is used that provides plaintext communication.

4. You can delete the $\text{SecureSocketDriver}$ instance at the end of the GridMate session, ideally in the $\text{OnSessionDelete}$ event on the $\text{SessionEventBus}$.

The GridMate Session Encryption Example (p. 434) at the end of this topic has sample code for these steps.

SecureSocketDesc

The constructor for $\text{SecureSocketDriver}$ requires a $\text{SecureSocketDesc}$ object that provides all encryption configuration required for the secure connection. The configuration parameters are described in the following table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{m_privateKeyPEM}$</td>
<td>Base-64 encoded string PEM private key.</td>
</tr>
<tr>
<td>$\text{m_certificatePEM}$</td>
<td>Base-64 encoded string PEM public certificate.</td>
</tr>
<tr>
<td>$\text{m_certificateAuthorityPEM}$</td>
<td>Base-64 encoded string PEM certificate authority.</td>
</tr>
<tr>
<td>$\text{m_authenticateClient}$</td>
<td>If set to 1, the client is expected to provide a signed certificate for authentication. To implement this, $\text{m_certificatePEM}$ must be set on the client, and the server needs to set up $\text{m_certificateAuthorityPEM}$. The default setting is 0.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Role</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>$\text{m_certificateAuthorityPEM}$</td>
</tr>
<tr>
<td>Server</td>
<td>$\text{m_privateKeyPEM, m_certificatePEM, m_certificateAuthorityPEM}$</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Role</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td><code>m_privateKeyPEM, m_certificatePEM, m_certificateAuthorityPEM</code></td>
</tr>
<tr>
<td>Server</td>
<td><code>m_privateKeyPEM, m_certificatePEM, m_certificateAuthorityPEM</code></td>
</tr>
</tbody>
</table>

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.

```cpp
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
Controlling Bandwidth Usage

GridMate (p. 402) provides several ways to control the bandwidth that your game uses, including bandwidth throttling and the prioritization of replica (p. 414) 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:

```cpp
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:

```cpp
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:

```cpp
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. 418) 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:

```cpp
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:

```cpp
// A replica chunk with this priority will be sent before all the chunks with Normal priority, but after all the chunks with High priority:
```
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:

```cpp
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):

<table>
<thead>
<tr>
<th>CVar</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gm_replicas-SendTime</td>
<td>The time, in milliseconds, between replica transmissions. A value of 0 binds the interval to the GridMate tick rate.</td>
</tr>
<tr>
<td>gm_replicas-SendLimit</td>
<td>The limit, in bytes, of the amount of replica data that can be sent per second. A value of 0 disables the limit.</td>
</tr>
<tr>
<td>gm_burstTime-Limit</td>
<td>The time, in seconds, that bursts in bandwidth are allowed. Bursts are allowed only if the bandwidth is not capped when the burst occurs.</td>
</tr>
</tbody>
</table>

### 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.
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).

RMI Functions

To send remote method invocations (RMIs), use the `InvokeRMI` function, which has the following syntax:

```cpp
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. 439) 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. 439) 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:

![Diagram showing PREATTACH, ENTITY ASPECT DATA, and POSTATTACH]

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 RMI

- **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 RMI

- **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`.

```cpp
#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`:

```cpp
Class CInventory : public CGameObjectExtensionHelper<CInventory, 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:

```cpp
pInventory->GetGameObject()->InvokeRMI(CInventory::Cl_SetAmmoCount(), TRMIInventory_Ammo("Pistol", 10), eRMI_ToAllClients);
```

The following line will invoke the function:

```cpp
pInventory->GetGameObject()->InvokeRMI(CInventory::Cl_SetAmmoCount(), TRMIInventory_Ammo("Pistol", 10), eRMI_ToAllClients);
```
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**
- Geometries (p. 441)
- Physical Entities (p. 442)
- Functions for Entity Structures (p. 445)
- Collision Classes (p. 453)
- Functions for World Entities (p. 455)

**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 manager, 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_multicontact flags give some hints on whether multiple contacts are possible. Specifying that multiple contacts are unlikely (mesh_multicontact0) 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_multicontact2 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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE_STATIC</td>
<td>An immovable entity. An immovable entity can still be moved manually by setting positions from outside, in order to ensure proper interactions with simulated objects, it is better to use PE_RIGID entity with infinite mass.</td>
</tr>
<tr>
<td>PE_RIGID</td>
<td>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;</td>
</tr>
<tr>
<td>PE_ARTICULATED</td>
<td>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 will not be used.</td>
</tr>
<tr>
<td>PE_WHEELEDVEHICLE</td>
<td>A wheeled vehicle. Internally it is built on top of a rigid body, with added vehicle functionality (wheels, suspensions, engine, brakes). 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.</td>
</tr>
<tr>
<td>PE_LIVING</td>
<td>A special entity type to represent player characters that can move through the physical world and interact with it.</td>
</tr>
<tr>
<td>PE_PARTICLE</td>
<td>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.</td>
</tr>
<tr>
<td>PE_ROPE</td>
<td>A rope object. It can either hang freely or connect two purely physical entities.</td>
</tr>
<tr>
<td>PE_SOFT</td>
<td>A system of non-rigidly connected vertices that can interact with the environment. A typical usage is cloth objects.</td>
</tr>
</tbody>
</table>

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 \( \text{mass} = \text{density} \times \text{volume} \); 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.
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_dynamics

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 \( q_{ext} \). 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 `bIneritVel` 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**

```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**

```lua
Player = {
   ...
   physicsParams = {
      collisionClass=collision_class_player,
   },
   ...
}
```
BasicAI.lua

BasicAI = {  
  ...,  
  physicsParams =  
  {  
    collisionClass=collision_class_ai,  
  },  
  ...,  
}

The `type` identifies which entity the collision classes belong to.

Some collision classes like the following are defined in `CryPhysics`:

- collision_classTerrain
- collision_classWheeled
- collision_classLiving
- collision_classArticulated
- collision_classSoft
- collision_classRoped
- collision_classParticle

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:

```cpp
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:

```cpp
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:

```cpp
ai7.ignore = AI_1
```

If you want object1 to ignore all living physical entities, set its ignore flag like this:

```cpp
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. 444).

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. 444) and optional sorting of the output list by entity mass in ascending order.

#### Syntax

```cpp
virtual int GetEntitiesInBox(Vec3 ptmin, Vec3 ptmax, IPhysicalEntity **pList, int objtypes, int szListPrealloc=0) = 0;
```

#### Example call

```cpp
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptmin</td>
<td>Minimum point in the space that defines the desired box volume.</td>
</tr>
<tr>
<td>ptmax</td>
<td>Maximum point in the space that defines the desired box volume.</td>
</tr>
<tr>
<td>pList</td>
<td>Pointer to a list of objects that the function populates.</td>
</tr>
<tr>
<td>objtypes</td>
<td>Types of objects that need to be considered in the query.</td>
</tr>
<tr>
<td>szListPrealloc</td>
<td>If specified, the maximum number of objects contained in the pList array.</td>
</tr>
</tbody>
</table>

---

Version 1.4

456
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:

<table>
<thead>
<tr>
<th>Entity type flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ent_static</code></td>
<td>Static entities</td>
</tr>
<tr>
<td><code>ent_terrain</code></td>
<td>Terrain</td>
</tr>
<tr>
<td><code>ent_sleeping_rigid</code></td>
<td>Sleeping rigid bodies</td>
</tr>
<tr>
<td><code>ent_rigid</code></td>
<td>Rigid bodies</td>
</tr>
</tbody>
</table>

After the function completes, you can easily iterate through the entity list to perform desired operations, as in the following code outline:

```cpp
for (int i = 0; i < entityCount; ++i) {
    IPhysicalEntity* entity = entityList[i];
    [...]
    if (entity->GetType() == PE_RIGID) {
        [...]
    }
    [...]
}
```

If `ent_allocate_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/distance^2. 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 use 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. 487).

Topics
• Profiler Tutorial (p. 459)
• Creating and Using Annotations (p. 469)
• Using Profiler for Networking (p. 473)
• Using the Profiler for CPU Usage (p. 481)
• Using Profiler for VRAM (p. 484)
• Using GridHub (p. 487)

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. 460)
• Launching Profiler (p. 460)
• Capturing Data (p. 460)
• Inspecting Data (p. 462)
• Playing Back Data (p. 464)
• Exporting Data (p. 469)
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**:

   ![Profiler interface](image)

   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.

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:
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, 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. 473), Using the Profiler for CPU Usage (p. 481), and Using Profiler for VRAM (p. 484).

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:

![Save to CSV icon](image)

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 surfacing per-frame log information from the data captured from your application. The first section shows you how annotations are used in Profiler; the next section shows you how you can modify your application so that they appear in Profiler.

Topics
- Using Annotations (p. 470)
- Creating Annotations (p. 471)
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 appears:

The dialog box contains a list of available annotations and their display colors. For information on creating annotations for your application, see Creating Annotations (p. 471).

2. When you select an annotation in the dialog box, a marker and line of the same color appears in the channel display (you might have to scroll horizontally to find the marker):

3. To display details for the annotations that occurred on a frame, pause your mouse on an annotation marker. In the example image, IP addresses have been redacted out.
Creating Annotations

In order 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:

```cpp
AZ_TracePrintf("GridMate","Connection %s \rightarrow %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 \rightarrow <IP_Address> 57455 (Client) (Connections=1)!
```

The text displays in Profiler like this:

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 of 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 on "GridMate", shows the message specified by the line of code that was added to the application:

```
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.
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. 402). For an introduction to the Profiler tool, see Profiler (p. 459).

Topics

• Carrier Profiler (p. 473)
• Replica Activity Profiler (p. 474)

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:

![Chunk Type View](image)

**To inspect chunk type details**

- Click the details icon for the chunk type:

![Chunk Type Details](image)

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:

<table>
<thead>
<tr>
<th>All Threads</th>
<th>Incl. Time</th>
<th>Excl. Time</th>
<th>Calls</th>
<th>Acc. Time</th>
<th>Acc. Calls</th>
</tr>
</thead>
</table>

**All Threads**
Use the thread selector to control which threads are shown in the tree view and in the graph:

- **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. 484)
- Understanding the Captured Data (p. 485)
- Inspecting the Data (p. 485)

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:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Category</td>
<td>Number of Allocations</td>
<td>Memory Usage</td>
</tr>
<tr>
<td>2</td>
<td>Texture</td>
<td>449</td>
<td>440342073</td>
</tr>
<tr>
<td>3</td>
<td>Buffer</td>
<td>1034</td>
<td>14778088</td>
</tr>
<tr>
<td>4</td>
<td>Resource Name</td>
<td>VRAM Allocation Size</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>SRT_ShadowPool</td>
<td>67108864</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>CachedShadowMap_0</td>
<td>35515392</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>$AutoDownload_4</td>
<td>10777216</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>$AutoDownload_3</td>
<td>16777216</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>$AutoDownload_5</td>
<td>19777216</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>HeightMapAO_Depth_1</td>
<td>11134800</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>HeightMapAO_Depth_0</td>
<td>8388608</td>
<td></td>
</tr>
</tbody>
</table>

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_00000000002CS9248, 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. 459).

**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. 487)
- Viewing and Configuring GridHub (p. 487)
- Troubleshooting GridHub (p. 489)

**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**:

   ![Show](image)

   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:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Connection ID</th>
<th>IsHost</th>
<th>IsLocal</th>
<th>IsReady</th>
</tr>
</thead>
<tbody>
<tr>
<td>d491d7</td>
<td>GridHub_copyapp</td>
<td>--</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3d15d6</td>
<td>GridHub_Headless</td>
<td>144424500</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

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. 490)
- Streaming System (p. 493)
- Text Localization and Unicode Support (p. 503)
- CryLog (p. 507)
- CryConsole (p. 509)

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 amount 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

```c
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
   eStreamTaskTypeShader     = 8,  // Shader combination data
   eStreamTaskTypeSound      = 7,  // Sound
   eStreamTaskTypeMusic      = 6,  // Music
   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

```cpp
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

```cpp
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 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.

```cpp
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;
}
```
```c
m_nTimeGroup = (uint64)(gEnv->pTimer->GetCurrTime() / max(1, g_cvars.sys_streaming_requests_grouping_time_period));
uint64 nPriority = m_ePriority;

int64 nDiskOffsetKB = m_nDiskOffset >> 10; // KB
m_nSortKey = (nDiskOffsetKB) | (((uint64)m_nTimeGroup) << 30) | (nPriority << 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

```c
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

```c
struct SRequestTypeInfo
{
    int nOpenRequestCount;
    int nPendingReadBytes;

    // stats collected during the last second
    uint32 nCurrentReadBandwidth;
};
```
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`.

### 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 management 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).
- **cfg_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 three 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**

```cpp
#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(eStreamTask
TypeNewType, 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.

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>character</td>
<td>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].</td>
</tr>
<tr>
<td>Unicode</td>
<td>A standard maintained by the Unicode Consortium that deals with text and language standardization.</td>
</tr>
<tr>
<td>UCS</td>
<td>Universal Character Set, the standardized set of characters in the Unicode standard (also, ISO-10646)</td>
</tr>
<tr>
<td>(UCS) code-point</td>
<td>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</td>
</tr>
<tr>
<td>(text) encoding</td>
<td>A method of mapping (a subset of) UCS to a sequence of code-units, or the process of applying an encoding.</td>
</tr>
<tr>
<td>code-unit</td>
<td>An encoding-specific unit integral identifier used to encode code-points. Many code-units may be used to represent a single code-point.</td>
</tr>
<tr>
<td>ASCII</td>
<td>A standardized encoding that covers the first 128 code-points of the UCS space using 7- or 8-bit code-units.</td>
</tr>
<tr>
<td>(ANSI) code-page</td>
<td>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.</td>
</tr>
<tr>
<td>UTF</td>
<td>UCS Transformation Format, a standardized encoding that covers the entire UCS space.</td>
</tr>
<tr>
<td>UTF-8</td>
<td>A specific instance of UTF, using 8-bit code-units. Each code-point can take 1 to 4 (inclusive) code-units.</td>
</tr>
<tr>
<td>UTF-16</td>
<td>A specific instance of UTF, using 16-bit code-units. Each code-point can take 1 or 2 code-units.</td>
</tr>
<tr>
<td>UTF-32</td>
<td>A specific instance of UTF, using 32-bit code-units. Each code-point is directly mapped to a single code-unit.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Code-unit size</th>
<th>Code-point size</th>
<th>Maps the entire UCS space</th>
<th>Trivial to encode/decode</th>
<th>Immune to byte-order differences</th>
<th>Major users</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>7 bits</td>
<td>1 byte</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>Many English-only apps</td>
</tr>
<tr>
<td>(ANSI) code-page</td>
<td>8 bits</td>
<td>varies, usually 1 byte</td>
<td>no</td>
<td>varies, usually yes</td>
<td>yes</td>
<td>Older OS functions</td>
</tr>
<tr>
<td>UTF-8</td>
<td>8 bits</td>
<td>1 to 4 bytes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>Most text on the internet, XML</td>
</tr>
<tr>
<td>UTF-16</td>
<td>16 bits</td>
<td>2 to 4 bytes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>Windows &quot;wide&quot; API, Qt</td>
</tr>
<tr>
<td>UCS-2</td>
<td>16 bits</td>
<td>2 bytes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>None (replaced with UTF-16)</td>
</tr>
<tr>
<td>UTF-32 UCS-4</td>
<td>32 bits</td>
<td>4 bytes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>Linux &quot;wide&quot; API</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Text data type</th>
<th>Encoding</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code</td>
<td>ASCII</td>
<td>We write our code in English, which means ASCII is sufficient.</td>
</tr>
</tbody>
</table>
Text data type | Encoding | Reason
--- | --- | ---
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 `std::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 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.

```cpp
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.

<table>
<thead>
<tr>
<th>Message</th>
<th>verbosity 0</th>
<th>verbosity 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>eAlways</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>eErrorAlways</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Version 1.4
507
<table>
<thead>
<tr>
<th>Message</th>
<th>verbosity 0</th>
<th>verbosity 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>eWarningAlways</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>eInput</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>eInputResponse</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>eError</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>eWarning</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>eMessage</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>eComment</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Source</th>
<th>Log file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lumberyard Editor</td>
<td>Editor.log</td>
</tr>
<tr>
<td>Game</td>
<td>game.log (default)</td>
</tr>
<tr>
<td>Error messages</td>
<td>Error.log</td>
</tr>
</tbody>
</table>

**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
### 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 IVar 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:

```cpp
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:

```cpp
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.

```plaintext
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.

```plaintext
sys_spec_Particles
```

Console variable group to apply settings to multiple variables

```plaintext
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
```
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

<table>
<thead>
<tr>
<th>Command</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait_frames</td>
<td>num:</td>
<td>Wait for num frames before the execution of the list is resumed.</td>
</tr>
<tr>
<td>wait_seconds</td>
<td>sec:</td>
<td>Wait for sec seconds before the execution of the list is resumed.</td>
</tr>
</tbody>
</table>

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\GameD11\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.

```c
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.

   ```cpp
   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 preceding 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.

   ```cpp
   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.

```cpp
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.

```plaintext
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.