Amazon Lumberyard

Welcome Guide

Version 1.26

Amazon Lumberyard: Welcome Guide

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

Table of Contents

Welcome to Amazon Lumberyard 1
- Amazon Lumberyard's creative capabilities without compromise 1
- Welcome Guide contents 2

Lumberyard features 3
- Here are just some of Lumberyard's features: 3
- Supported platforms 4

How Amazon Lumberyard works 5
- Overview of the Lumberyard framework 5
- Working with Gems 7
- Messaging between systems with EBus 7
- The Component Entity system 9
- The Lumberyard Asset Pipeline 9
- Scripting for the Lumberyard framework 10
- Further learning 11

Amazon Lumberyard set up 12
- System requirements 12
  - System requirements 12
  - Prerequisites 13
  - Visual C++ redistributable packages 13
- Install 14
  - Run the Lumberyard installer 14
  - Lumberyard's executables 14
  - Lumberyard's directory structure 15
- Setup Assistant 15
  - Lumberyard Setup Assistant custom installation 16
- Project Configurator 18
  - Select a project 19
  - Create a new project 20

Create with Amazon Lumberyard 24
- How do I create a game with Lumberyard? 24
- Work as an artist 26
- Work as a designer 27
- Work as a game engineer 28
- Lumberyard Editor tour 29
  - The Lumberyard Editor default layout 30
  - Navigating the Lumberyard Perspective viewport 32
  - Movement preferences 33
  - Save Perspective locations 34
- Additional tools 34
  - Animation Editor 35
  - FBX Settings 36
  - Texture Settings Editor 38
  - Asset Editor 40
  - Level Inspector 41
  - Material Editor 42
  - Particle Editor 43
  - PhysX Configuration 44
  - Script Canvas 45
  - Terrain Editor 46
  - Terrain Tool 47
  - Track View 47
  - UI Editor 48
  - Audio Controls Editor 48
  - Console Variables Editor 49

Lens Flare Editor 50
Sun Trajectory Tool 51
Terrain Texture Layers 51
Time of Day Editor 52
Get started 53
Prerequisites 53
Assets 54
About the tutorial game 54
Chapters 55
Levels 55
Entities 59
Characters 64
PhysX 70
Input and Scripting 74
Cameras 79
Terrain 83
Slices 89
Release 93
Additional Lumberyard resources 94
Legal information 95
Lumberyard redistributables 95
Alternate web services 96

Welcome to Amazon Lumberyard

Amazon Lumberyard is a *free* high-performance *3D engine* with a professional suite of tools, editors, and libraries that enable you to create captivating real-time graphics, immersive experiences, awe-inspiring virtual worlds, and dynamic visualizations. Lumberyard brings the capabilities of Amazon Web Services and the community of Twitch to connect your ideas to players, customers, and viewers around the world.

What does "free" mean?

Lumberyard is free. There are no seat licenses, no site licenses, no subscriptions, no support contracts, no revenue share. Just free. If you choose to use AWS services in your project, you'll be responsible for fees associated with those services. Amazon Lumberyard is always free to use and you can customize the tools and source code to suit your needs.

Amazon Lumberyard's creative capabilities without compromise

A world-class engine

At its core, Lumberyard is a high-performance real-time 3D engine that produces incredible visual fidelity. Lumberyard includes all the capabilities that professional designers, artists, and developers expect, with familiar user experience patterns that enable fast adoption and development. Its robust toolset enables rapid iteration to the highest quality. Built on a modular architecture and extensible through the *Gems* framework, Lumberyard makes it easy to add new features, APIs, and assets.

Deeply integrated with Amazon Web Services

Lumberyard's online capabilities, backed by Amazon Web Services, offer staggering possibilities to developers. Common connected elements like dedicated servers, dynamic content, online economies, and real-time stats are easily implemented and scaled, so you can focus on the creative aspects of your project. With Lumberyard's crossplay capabilities and Twitch integration, you can find new and novel ways to engage people socially. The limitless potential of AWS can deliver experiences that reach far beyond the capabilities of any single device.

Whether you're a student, hobbyist, independent developer, or major studio, Lumberyard provides the same growing toolbox to realize your creative vision.

Interested? Read on to learn what's in the box.

Welcome Guide contents

Lumberyard features (p. 3) - Learn about Lumberyard's feature set.

How Amazon Lumberyard works (p. 5) - A high-level overview of the technology behind Lumberyard.

Setting up Amazon Lumberyard (p. 12) - Download, install, and create a project with Lumberyard.

Create with Amazon Lumberyard (p. 24) - A quick introduction to Lumberyard's workflows and core tools.

Get started with Amazon Lumberyard tutorials (p. 53) - Step-by-step tutorials to get you hands-on experience with Lumberyard's workflows and core tools.

Additional Lumberyard Resources (p. 94) - Links to video tutorials, documentation, and Lumberyard online communities.

Lumberyard features

Lumberyard is packed with features that enable you to create beautiful worlds, engaging characters, and dynamic simulations. With Lumberyard's deep integration to Amazon Web Services and Twitch, you can add cloud-connected features and connect players and customers around the world.

Here are just some of Lumberyard's features:

Engage fans with Twitch

Add new levels of interactivity between Twitch streamers and viewers with Twitch ChatPlay. Give Twitch streamers dynamic real-time broadcast customization options with Twitch Metastream. Enable direct engagement between Twitch streamers and viewers with Twitch JoinIn.

Build robust multiplayer features, fast

Easily deploy, operate, and scale multiplayer game servers with Amazon GameLift. Use the Cloud Gems Framework to add dynamic content, leaderboards, and daily messages.

Create incredible worlds

Lumberyard Editor is a powerful, customizable world-building tool. Sculpt and paint expansive, multi-layered terrain. Decorate your landscape with dense, lush vegetation. Design panoramic skies with dynamic time-of-day and atmospheric effects. Create interactive infinite oceans. Populate your world with life-like entities using simple but powerful components.

Tell stories with emotive characters

Create believable, immersive characters with *EMotion FX*'s data-driven tools. Lumberyard's Animation Editor provides a visual, node-based interface for bringing characters to life with blend trees and blend spaces, visual state machines, and linear skinning. Tell engaging stories through cinematics with Lumberyard's cinematic sequence editor, Track View.

Spectacular visual effects with NVIDIA PhysX and particles

NVIDIA PhysX and NVIDIA Cloth enable you to build dynamic, real-time physical interactions with static colliders, rigid bodies, and realistic cloth simulations. With Lumberyard's Particle Editor, you can create stunning visual effects.

Develop engaging experiences with ease

You don't need to be an experienced programmer to use the power of Lumberyard. Design captivating systems and experiences with Script Canvas, a node-based visual script editor that enables easy access to Lumberyard's underlying functionality. Lumberyard also supports Lua for designers who prefer the rapid iteration that scripting languages provide.

Leverage the power of Amazon Web Services

With Lumberyard, you can incorporate AWS services such as Amazon DynamoDB, AWS Lambda, Amazon S3, Amazon Cognito, Amazon SNS, and Amazon SQS with Cloud Canvas tools and solutions. The Cloud Gem Framework Gem provides C++ classes that can execute any AWS SDK for C++ call, bringing the vast compute and storage of the cloud to your projects.

Supported platforms

You can use Lumberyard to develop games for the following platforms:

- Microsoft Windows PC
- Android
- iOS
- macOS
- PlayStation
- Xbox
- Linux (dedicated server only)

Some platforms have additional requirements.

- For console support, see the Xbox for game developers and PlayStation Partners portals.
- For mobile devices, see Creating Android and iOS projects in the User guide.
- For macOS, see Creating macOS projects in the User guide.
- For Linux dedicated servers, see Creating Lumberyard Executables for Linux in the User guide.

How Amazon Lumberyard works

Amazon Lumberyard provides a complete, end-to-end environment for developing and delivering games and simulations, and supports a wide variety of platforms, including consoles, mobile devices, virtual reality, and Microsoft Windows PCs. Since it's such a large development environment with so many different features and tools, it can be intimidating to new users, especially for those who don't have a traditional developer background. This topic covers the various parts of Lumberyard at a high level and the common ways you can work with it depending on your role or task as a new or experienced game developer or designer.

Lumberyard has several tools, editors, and systems that work together to help you assemble a game. Central to this is the "Lumberyard game engine", which provides the following:

- Graphics rendering and output
- Game logic execution
- Messaging across game systems and components
- Memory and resource abstraction and management
- Multi-threading support for input, audio, physics simulation, user interfaces (UI), artificial intelligence (AI), networking and multiplayer, and other common game features
- Asset management and packaging

The engine itself is really just a collection of components and modules, called *Gems*. When developing, you use the Lumberyard Editor and tools to assemble the engine for your game, choosing and adding these gems and modules as you develop it. You also use the Lumberyard tools to add your assets—including textures, meshes, sounds, music, and scripts—to construct your game's unique experience and gameplay.

Think of Lumberyard as a collection of discrete elements: code, scripts, various GUI-based editors, and command line tools. When you compile a game project, Lumberyard's build scripts pull in all the pieces specified in your project's configuration to build your game. The parts of the engine that go into your game are only those you've configured your project to use, and there's very little functionality included in the final compiled code that you didn't ask to have in it. Likewise, the asset bundling and management tools make sure you only ship with the assets you actually use in your game.

You can build a game in Lumberyard just using the Lumberyard Editor, but you will be constrained to the Gems and tools provided (along with the assets and scripts you create). If your ambitions are greater—if you want to evolve Lumberyard to support features and systems we haven't provided in the box—read on.

Overview of the Lumberyard framework

The Lumberyard installation provides a sandbox of different bits to combine, including over 100 gems and modules for you to use in your game. As a high-level concept, think of Lumberyard as a framework—a conceptual structure from which you can add new features or remove anything you don't plan to use in your game. When you need to change or extend the behavior of something, you don't need to go to some massively over-architected set of code files and hack or refactor a feature in to your game's code. Rather, you just work with only the gem or module that contains the functionality you want to change.

By focusing on modularity, you can safely experiment with different feature changes without risking progress on the entire game with an unintended regression.

Central to Lumberyard is the **AzFramework** library, which relates all of the systems, modules, and Gems into the infrastructure for your game. The Event Bus (EBus) system provides request and notification messaging across the DLLs for these systems, modules, and Gems. As a developer, you write C++ code to implement methods defined in C++ API headers that define the functionality you need.

The default Lumberyard installation is to `{drive-letter}:\Amazon\Lumberyard{lumberyard-version-number}\` on your PC. There are two folders under it: `\dev` and `\3rdParty`. We'll examine the most important contents of \dev, which is where AzFramework and all of the various Gem and module interfaces are defined, as well as the EBus interfaces. It also contains scripts, samples, and Editor assets.

- `\dev\Bin64vc142` (or, in versions prior to 1.24, `\dev\devBin64vc141`) contains the asset builder libraries, the Lumberyard editor modules (plugins) and libraries (under `\EditorPlugins`), among other things. Build and configuration logs are under `\Logs`.
- `\dev\Code` contains the C++ API headers that you'll include in any code you create to extend the core functionality of Lumberyard. They are organized under folders for each of the systems or features that you want to extend. Most of them are virtual interfaces that you'll implement so you connect that functionality to the relevant Lumberyard system or feature through EBus, or contain the expected type and template definitions to use. These include:
  - `\Code\CryEngine` contains API headers and C++ code files for CryEngine systems and features. If you need to tweak the behaviors of CryEngine, including the 3D renderer, the legacy physics engine, the GridMate networking engine, or other components of the CryEngine, start with the source files in this directory.
  - `\Code\Framework` contains the AzFramework APIs. AzFramework is the core Lumberyard framework that defines all the systems, including Gems, systems engines, and EBus.
    - `\Code\Framework\AzCore\AzCore`: Contains all of the APIs for interoperating with Lumberyard's systems, modules, and Gems. If you are extending Lumberyard's default functionality, this is where you'll find all the APIs you need.
    - `\Code\Framework\GridMate`: Contains the GridMate networking APIs.
- `\Editor` contains all the assets, scripts, and configuration files for the Lumberyard Editor as well as some of the various editors it manages, such as the Materials Editor, the UI Canvas Editor, and the Particle Editor.
- `\Engine` contains all the default asset binaries, script files (`.lua`), entity configuration files (`.ent`), material definition files (`.mtl`), shader extensions (`.ext`), and engine configuration files for Lumberyard.
- `\Gems` contains all the source and build files for the Gems that ship with Lumberyard. When you create a new Gem, you'll add the code and build sources here, and then enable the Gem through the Project Configurator.
- `\ProjectTemplates` contains the game project templates you can choose from when initially configuring Lumberyard for your game's development.
- `\SamplesProject` and `\StarterGame` contain samples and a complete game level for you to review.
- `\Tools` contains a number of tools and SDKs you'll use throughout the development of your game, including:
  - `\Tools\AWSNativeSDK`: Contains scripts for obtaining various platform-specific SDKs for working with Amazon's cloud services in their native languages.
  - `\Tools\AWSPythonSDK`: Contains Python SDKs for working with Amazon's cloud services.
  - `\Tools\AzCodeGenerator`: Contains the binaries and configuration files for AZ Code Generator, which is a command-line utility that generates source code (or any data or text) from specially tagged source code. For more details, see Automating Boilerplate with Az Code Generator.
  - `\Tools\build`: Contains scripts and tools for building your game project with Waf including the `lmbr_waf` command-line build utility.

- `\Tools\CrySCompileServer`: Contains the executable and libraries for the CryEngine shader compiler service.
- `\Tools\LuaRemoteDebugger`: Contains the executable and libraries for the Lua script remote debugging tool.
- `\Tools\Python`: Contains all basic Python libraries and the interpreter. Lumberyard supports a minimum version of 3.7.5.
- `\Tools\Redistributables`: Contains common packages and files that you can ship with your game, including DirectX and Visual Studio DLLs.
- `\Tools\RemoteConsole`: Contains the Lumberyard remote console application.

Lumberyard also provides two command-line tools, `Lmbr.exe` and `Lmbr_waf.exe`, which you use to manage your game project overall and configure the details of your game project during development.

- `Lmbr.exe`--Provides a set of commands for managing and tracking Gems, capabilities, and 3rd party tools and packages.
- `Lmbe_waf.exe`--Provides a set of commands for automating the building and packaging of game projects with the Waf build automation framework.

All of these parts—plus some not listed here—define Lumberyard, and can be used to construct your game. As you incorporate Lumberyard systems and develop your own, you will want to communicate across them. For that, we have EBus.

Working with Gems

Many of Lumberyard's capabilities are implemented through *Gems.* A Gem is a packaged extension that can be enabled in a project to add functionality or features. Gems might contain module code, assets, scripts, supporting files and references to other Gems. Using Gems enables you to choose the features you need for your project and exclude those that aren't necessary. By keeping everything modular and only using what you need, the Gem system lets you iterate faster. Asset collections, code samples, components, libraries, tools, and even entire game projects can be distributed as Gems.

You enable Gems when you create your project in Project Configurator. You can return to Project Configurator at any time to enable additional Gems and disable unused Gems. When you create your first project, be sure to click the **Enable Gems** option in Project Configurator to see the list of Gems available. Some Gems are core systems and required for all Lumberyard projects. Other Gems are extensions for existing Gems and require their dependencies be enabled for your project. Gems you enable are detected and built automatically when you build your project.

You can create your own Gems and easily reuse and distribute your own code and assets. To get a better idea what goes into creating a Gem, have a look at the Gems directory of your Lumberyard installation and examine the included Gems. The process for creating your own Gem is through Project Configurator, and is very similar to creating a project.

Messaging between systems with EBus

All of Lumberyard's Gems and systems, as well as the components in your projects need a way to communicate with each other. Lumberyard uses a general-purpose communication system called Event Bus (EBus for short).

As discussed earlier, Gems and systems are typically implemented as DLLs. EBus is used to communicate between them—and specifically, to invoke functions in one Gem or system from another. EBus provides

both request and publish/subscribe event interfaces (buses) that allow calls across those DLLs. For example, if you've created a Gem for custom physics behaviors and you'd like to provide data to the CryEngine renderer, you'd do so by implementing an EBus interface in your Gem.

We provide the interfaces for the Gems and systems DLLs we ship as headers in the default installation. To use the functionality in these DLLs, you use the interfaces in these headers to register for a single cast (Event) or broadcast (Broadcast) event, or through supplying a data request functor to a Request Bus handler.

Likewise, to expose functionality from your own gems and provide data to another system, you must inherit the virtual interface declared in the corresponding header file and implement the handlers on that interface in your Gem's classes, and then register that handler with the EBus system. Specifically, you'll register a handler you create with EBus, which will pass a pointer to your class method to the targeted system or post a notification to the systems that are subscribed to it.

Inside of your Gem code, you also manage the connection and disconnection of your implemented handler for the EBus. EBus is just a list of handlers that calls all the functors (function pointers) registered with it..

For singleton handlers where you only need one interface to communicate across DLLs, consider using AZ::Interface and AZ:Event directly, without EBus.

There are two types of EBus:

- Request bus: This EBus type registers a handler for a method that can be called by other systems.
- Notification bus: This EBus type provides a messaging interface for notifications that systems can publish or subscribe to.

EBuses have many advantages over traditional polling methods:

- Abstraction – Minimize hard dependencies between systems.
- Event-driven programming – Eliminate polling patterns for more scalable and higher performance 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.

EBuses are configurable and support many different use cases:

- As a direct global function call.
- Dispatch processing to multiple handlers.
- Queue all calls, acting like a command buffer.
- As an addressable mailbox.
- For imperative delivery.
- For queued delivery.
- Automatic marshaling of a function call into a network message or other command buffer.

For details on using EBus, read:

- Working with the Event Bus (EBus) System
- Event Buses in Depth

The Component Entity system

But what about the parts of your game that are actually in the game proper? Lumberyard has a model for that as well, called the component entity system.

Understanding the component entity system is fundamental to using Lumberyard. It's conceptually simple: Every in-game object you create for your project is an *entity*, with a unique ID and container. Each entity contains *components* that provide functionality. The functionality components provide is broad and includes primitive shapes, meshes, audio, artificial intelligence behaviors, animation, visual effects, physics, scripts, and so much more. There are even components that provide tool and debugging functionality.

As an example, suppose you want to create a door entity that can be opened and closed. You made a mesh and a couple audio files. Now, consider the functionality your door must have.

- Display the door model.
- Play back the audio files when the door opens and closes.
- Prevent passing through the door when it's closed.
- Animate the door open and close.
- Trigger the door open and close by some mechanism.

Knowing the functionality your door entity needs, you add components to the entity for each aspect of the door, including its presence in the game world and the ways a player can interact with it. A *Mesh* component visually represents the door in the game world. *Audio Trigger* components provide the audio when it opens or closes. A *PhysX Collider* component prevents a player from passing through the door when it's closed. *Script Canvas* components define the behaviors, including animation and sound playback, when the door is opened or closed. Whatever behavior you need to model, each entity is going to have a collection of components to support it. The only component common to all entities is the *Transform* component, which provides the position, orientation and scale of your entity in the game world.

Entities are easy to grasp and create, but can become complex. If the entity requires a lot of functionality, the list of components grows quickly. What if you want to add a latch with its own animation and audio to the door? What if you want to add a breakable glass pane to the door? Suddenly, the entity goes from having five components to dozens of components. This is where *slices* come in.

A slice, like an entity, is a container with a transform behavior. Instead of containing components, a slice contains one or more configured entities and might contain other slices as well. To create a more complex door, you could have the initial door entity, a second entity for the latch and its components, and a third entity for the breakable glass pane and its components. These three small entities are collected into a slice that provides a reusable, fully functional door asset. You may have heard similar concepts referred to as *prefabs* in other software. In Lumberyard, a slice is a collection of entities and/or other slices in a single reusable asset.

Behaviors applied to a slice can potentially cascade down to all of the entities it contains, and then down to the components of that entity. However, the reverse is not true, as it would make no sense for a window shattering to apply to a door latch.

Once you've internalized the slice > entity > component hierarchy, consider how you would use these concepts to develop the various elements potentially populating your game levels and world.

The Lumberyard Asset Pipeline

We've discussed what's in the Lumberyard installation, the basic framework of Lumberyard and how its various parts communicate with one another, and how you represent objects in your game. But how

do you create and manage assets for your game? After all, a door doesn't truly exist in your game if it doesn't have some form of representation.

For the purposes of Lumberyard, we'll define an asset as a resource file, saved on disk, consumed by your project in some way. An asset might be a font for your user interface, a bitmap file that contains a grassy terrain texture, a rock mesh you sculpted, animations for a character, etc. Some assets might be created in Lumberyard. Specialized files called *inputbindings* that map buttons from a game pad to input events for your project and *physicsmaterials* that describe the physical properties of surfaces, for example, are both created in Lumberyard's Asset Editor.

For many reasons, the primary being performance considerations, these different assets cannot be consumed by Lumberyard without being converted to operating system specific, *game ready* data. This process, going from a source asset file on disk to game ready data is the Asset Pipeline. The processing is performed automatically by Asset Processor.

Asset Processor is a background process (you'll see its icon in the task tray when it's running) that constantly scans directories in your project for changes in files. When changes are detected, Asset Processor uses configurable rules to determine how to handle the changes. The objective is to have game ready versions of all assets for each OS and each game directory in a location called the *asset cache*. The asset cache is kept separate from your asset directories and can be automatically rebuilt entirely from your source assets by the Asset Processor.

The asset cache contains a full image of all files (except executables and related files) needed to run your project. Asset Processor keeps the image up to date, ensuring that new files are ready to use in the project runtime and Lumberyard Editor as soon as possible. Your project runtime will only load assets from the asset cache and never directly from your asset source directories.

Projects like modern games can have thousands of assets that need to be monitored and processed for multiple target operating systems. To manage this complexity, the Asset Pipeline is completely configurable. Here are just some of the configuration options available:

- Specify what directories should be monitored for changes.
- Specify target operating systems and tailor the Asset Pipeline's behavior per target operating system.
- Set the number of concurrent processing tasks.
- Use metadata information to associate file types and process side-by-side assets.
- Add your own asset types to the Asset Pipeline.
- Batch process assets on a build server.

When you're preparing to ship, you'll need to package the assets your project uses. Even small projects can have hundreds of assets, including multiple versions of assets, many of them not required in your final project. Manually tracking and determining which assets you need to ship can be tedious, time consuming an error prone. Asset Bundler solves this for you.

Asset Bundler makes shipping the specific assets used for the release of your game more reliable and repeatable. Reliability is based on an underlying dependency system. If you make changes to your project and add, remove, or update assets, Asset Bundler uses the dependencies to automatically determine which assets to include. Repeatability is based on underlying configuration files that provide consistency each time you run Asset Bundler.

Scripting for the Lumberyard framework

Now, you've created assets and defined them in your game using slices, entities, and components. You have an empty level or world to roam, but where's the game? That's where scripting comes in.

Lumberyard includes two scripting technologies for creating logic and behaviors: Script Canvas and Lua.

Script Canvas is a visual scripting environment. In the Script Canvas editor, you create, connect, and rearrange graphical nodes that provide a visual representation of the logic flow. Script Canvas offers an approachable and easy-to-read environment to author behaviors using the same framework as Lua and C ++. You can use Script Canvas to create scripts without needing to know how to code.

To enable Script Canvas for Lumberyard, you must enable the Script Canvas Gem.

Lua is powerful, fast, lightweight, embeddable scripting language. Lua facilitates quick iteration in your project because you can run your changes immediately without needing to recompile your source code.

Lumberyard's functionality is exposed to Script Canvas and Lua by the behavior context. The behavior context reflects runtime code and makes it accessible to scripts by providing bindings to C++ classes, methods, properties, constants, and enums. The behavior context also provides bindings for Lumberyard's EBus so you can dispatch and handle events through Script Canvas and Lua.

Functionality for both Script Canvas and Lua is added to entities through components. You can have multiple script components and mix and match between Lua and Script Canvas within your entities. This approach enables you to create small, manageable modules of logic and behavior that can be reused throughout your projects.

Further learning

As you've probably suspected, there's a lot more to Lumberyard than this, but we hope this gives you a good sense of where to start your searches for greater detail in our docs and code. Here's a few additional links to help you explore further:

For some great videos on getting started with Lumberyard, check out our Getting Started video tutorial series.

For some docs to get you started, check out the following topics:

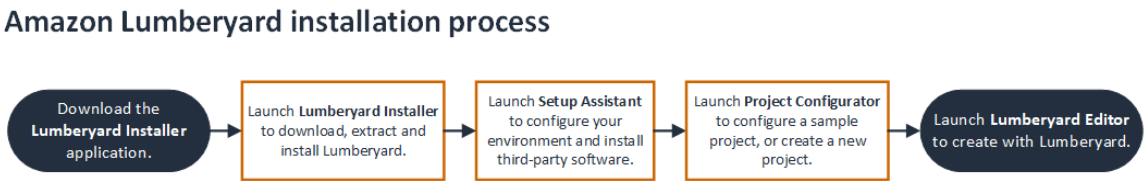
- Lumberyard Editor
- Programming Concepts
- Gems
- Component Entity System
- Component Reference
- Programmer's Guide to Entities and Components
- Emotion FX Animation Editor
- Script Canvas
- Lua Editor

Setting up Amazon Lumberyard

Amazon Lumberyard has a two-part installation process:

1. The installer application downloads, extracts, and installs Lumberyard.
2. **Setup Assistant** configures and installs additional software based on your development needs. Jump right in to the Lumberyard Editor with an **Express Install** or customize Lumberyard's features for your development needs with a **Custom Install**.

After Lumberyard is installed, use **Project Configurator** to select a project or create a new project. The following sections detail the minimum requirements for Lumberyard and guide you through the installation process.



Topics

- Amazon Lumberyard system requirements (p. 12)
- Installing Amazon Lumberyard (p. 14)
- Configuring your Amazon Lumberyard environment with Setup Assistant (p. 15)
- Manage Lumberyard projects with Project Configurator (p. 18)

Amazon Lumberyard system requirements

Lumberyard has a minimum set of system requirements for development, as outlined in the following sections. Disk space and RAM requirements are dependent on the options that you choose during installation.

System requirements

If your system is capable of running a modern real-time 3D game with good performance you should be set, however, review these detailed requirements to be certain.

Lumberyard requires Windows 10.

- 3 GHz quad-core processor
- 8 GB RAM
- 2 GB VRAM DirectX 11 or later compatible video card
  - NVIDIA GeForce GTX 660 Ti with driver version 368.81 or later
  - AMD Radeon HD 8730M with driver version 16.15.2211 or later

- 60 GB of free disk space

Note
If you select options to build the engine, editor, or tools in **Setup Assistant**, 14 GB RAM is required for compilation.
Some advanced graphics features require a DirectX 12 or later compatible video card.
Required free disk space is dependent on the options that you select when installing Lumberyard.

Prerequisites

You can use the Lumberyard Editor and tools without installing additional software. To create new projects or use advanced development features in Lumberyard, you need a developer environment that includes Microsoft Visual Studio. The following versions are supported:

- Microsoft Visual Studio 2019 version **16.2.4** thru version **16.7.x**.
- Microsoft Visual Studio 2017 version **15.9.14** thru version **15.9.x**.

Microsoft offers Visual Studio Community edition free to individual developers. For more information and to download and install Visual Studio Community, visit the Visual Studio Community portal.

Visual Studio 2017 and 2019 required features

The default Visual Studio installation might not include all of the features that are required by Lumberyard. Ensure that the following Visual Studio features are enabled:

1. Launch the **Visual Studio Installer** from your download directory or the **Start Menu** if you've already installed Visual Studio.
2. If you've installed Visual Studio, choose **More - Modify** on the version of Visual Studio you'll use with Lumberyard.
3. On the **Workloads** tab:
   - Select **Game development with C++**.
     - In the **Installation details** panel on the right, select at least one **Windows 10 SDK**.
   - Select **Desktop development with C++**.
4. On the **Individual components** tab, under **Compilers, build tools, and runtime**, make sure that a **VC++ toolset** that corresponds to the installed version of Visual Studio has been selected:
   - If using **Visual Studio 2017**: Select at least one version of the **VC++ 2017 toolset**.
   - If using **Visual Studio 2019**: Select at least one version of the **MSVC v142 - VS 2019 C++ x64/x86 build tool**.
     - (Optional) To build with the Visual Studio 2017 toolset in Visual Studio 2019, select **MSVC v141 - VS 2017 C++ x64/x86 build tools**.

Visual C++ redistributable packages

Lumberyard Installer will attempt to download and install the required Visual C++ Redistributable packages during the installation process. Under some circumstances, the installation of these redistributable packages may fail. If you receive a missing Visual C++ runtime `.dll` error while running Lumberyard Installer, Lumberyard Setup Assistant, or Project Configurator, do the following:

1. Check that the Visual C++ redistributable installers for Visual Studio 2012 and Visual Studio 2019 have been successfully downloaded. The installers are located in the corresponding **Visual Studio** directories in the `lumberyard_version\dev\Tools\Redistributables\` directory.

2. If the Visual C++ redistributable installers for Visual Studio 2012 and Visual Studio 2019 have not been downloaded by Lumberyard Installer, manually download the installers from Microsoft.
   - Visual C++ Redistributable for Visual Studio 2012
   - Visual C++ Redistributable for Visual Studio 2019
3. Run both Visual C++ redistributable installers.
4. Retry the Lumberyard installation after the Visual Studio redistributables have successfully installed.

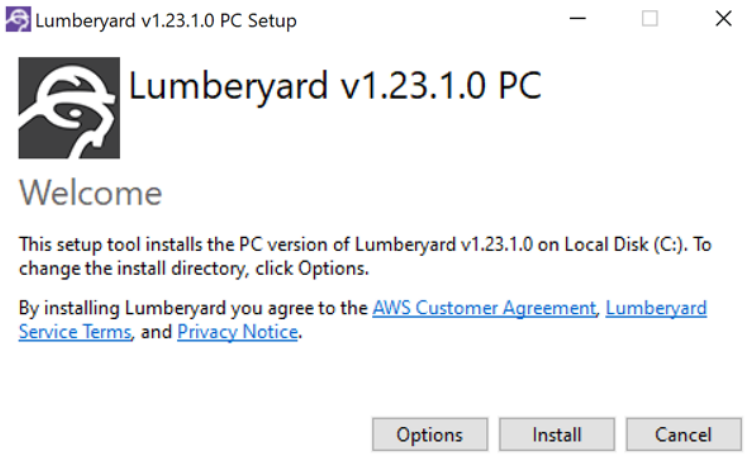
> **Note**
> The Visual C++ Redistributable for Visual Studio 2019 also contains redistributables for Visual Studio 2015 and 2017.

Installing Amazon Lumberyard

Download the latest Lumberyard Installer.

Run the Lumberyard installer

Navigate to your `Downloads` directory and run `LumberyardInstaller.exe` to download, extract, and install Lumberyard.



The default Lumberyard installation path is `C:\Amazon\Lumberyard\`. To set a different installation path, choose the **Options** button. Choose the **Install** button to begin installation. The process can take some time, depending on your internet connection speed.

> **Note**
> In some situations, the Lumberyard Installer can hang and refuse to progress. The only known remedy at this time is to forcibly terminate the installer through the Windows Task Manager and reboot, then retry the installation.

The installer displays an **Installation Successfully Completed** message. Click **Launch Lumberyard Setup Assistant** to continue with setup.

Lumberyard's executables

`LumberyardInstaller.exe` creates shortcuts on the desktop and in the Start Menu for three applications:

Setup Assistant (p. 15)

- Setup Assistant configures Lumberyard's environment according to your development needs, and downloads and installs additional software and SDKS. You can use Setup Assistant at any time to add development features to your Lumberyard environment.

  SetupAssistant.exe is located in `lumberyard_version\dev\Tools\LmbrSetup\Win`.

Project Configurator (p. 18)

- With Project Configurator, you create, configure, set, and build projects. When you run Project Configurator for the first time, you see several sample projects that are available to help you learn Lumberyard's features.

  ProjectConfigurator.exe is located in `lumberyard_version \dev\Bin64vc141_or_vc142`.

Lumberyard Editor (p. 29)

- Lumberyard Editor is Lumberyard's core application. In Lumberyard Editor, you create levels, assets, and interactions for your projects.

  Editor.exe is located in `lumberyard_version \dev\Bin64vc141_or_vc142`.

Lumberyard's directory structure

The default Lumberyard installation location is `C:\Amazon\Lumberyard\lumberyard_version\`. The root directory contains the following directories and files:

- `dev`
  - `_WAF_`: Waf build system files.
  - `Bin64`: Binaries and configuration files for the Resource Compiler.
  - `Bin64vc141`: Binaries and configuration files for Visual Studio 2017.
  - `Bin64vc142`: Binaries and configuration files for Visual Studio 2019.
  - `Code`: Source files and solution files for the Lumberyard engine and tools.
  - `Editor`: Editor assets.
  - `Engine`: Engine assets.
  - `Gems`: Modular components and assets.
  - `MultiplayerSample`: Multiplayer sample project that demonstrates how to build a multiplayer game with the component entity system.
  - `ProjectTemplates`: Configuration files, libraries, and scripts for the empty template.
  - `SamplesProject`: Sample project.
  - `StarterGame`: A full example game with 3D environments, event scripting, and basic enemy AI.
  - `Tools`: Third-party tools and plugins.
  - `engineroot.txt`: System file required by Lumberyard Setup Assistant to verify the directory.
- `3rdParty`
  - Third-party software required to use or compile Lumberyard.
  - `3rdParty.txt`: System file used by other third-party tools to verify the directory.

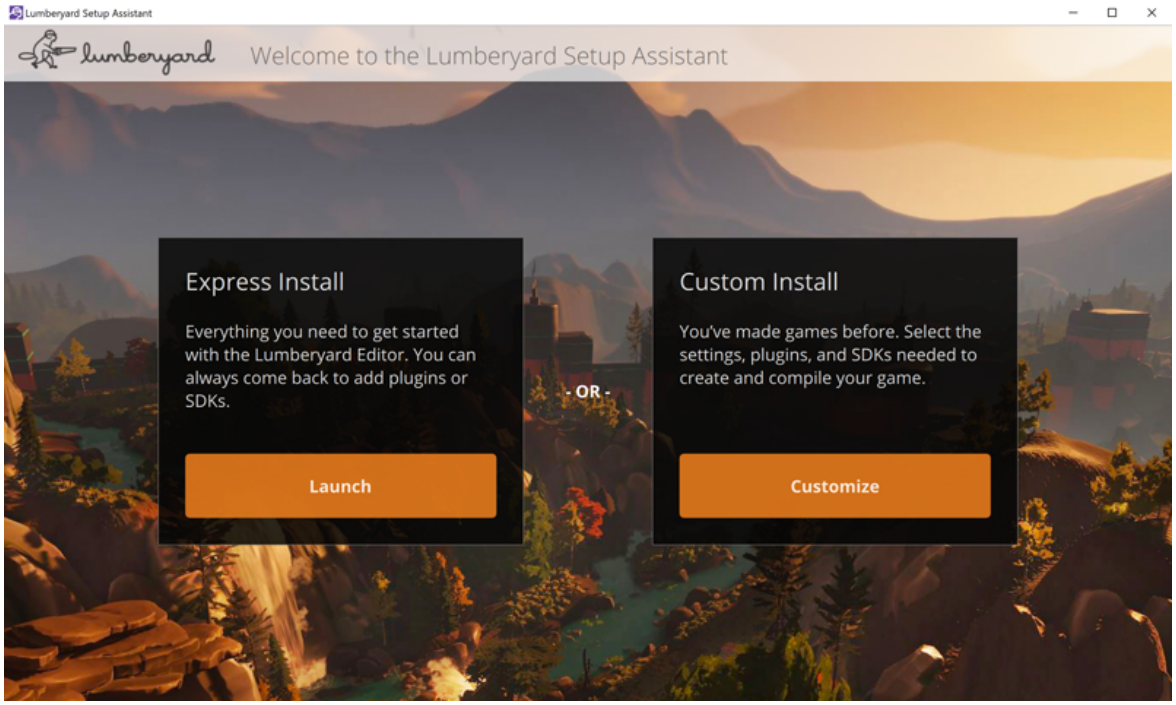
Configuring your Amazon Lumberyard environment with Setup Assistant

Lumberyard Setup Assistant configures and maintains your Lumberyard environment based on your development needs. If you're continuing from Installing Amazon Lumberyard (p. 14), Setup Assistant

should be running on your desktop. Setup Assistant can also be launched from the Start Menu or desktop shortcuts.

Lumberyard Setup Assistant performs several important functions:

- Ensures that you have the required runtime software.
- Ensures that you have the required SDKs.
- Provides plugins for detected content applications such as Photoshop and Maya.
- Validates registry settings, paths, and libraries.



On first run, Lumberyard Setup Assistant presents two options, **Express Install** and **Custom Install**. If you're a content creator, such as a designer or artist, and won't be compiling code, or if you want to jump right in and use the Lumberyard Editor and tools, select **Express Install**. You can always revisit Setup Assistant to add development features. If you'd like to set up Lumberyard for code development, select **Custom Install**.

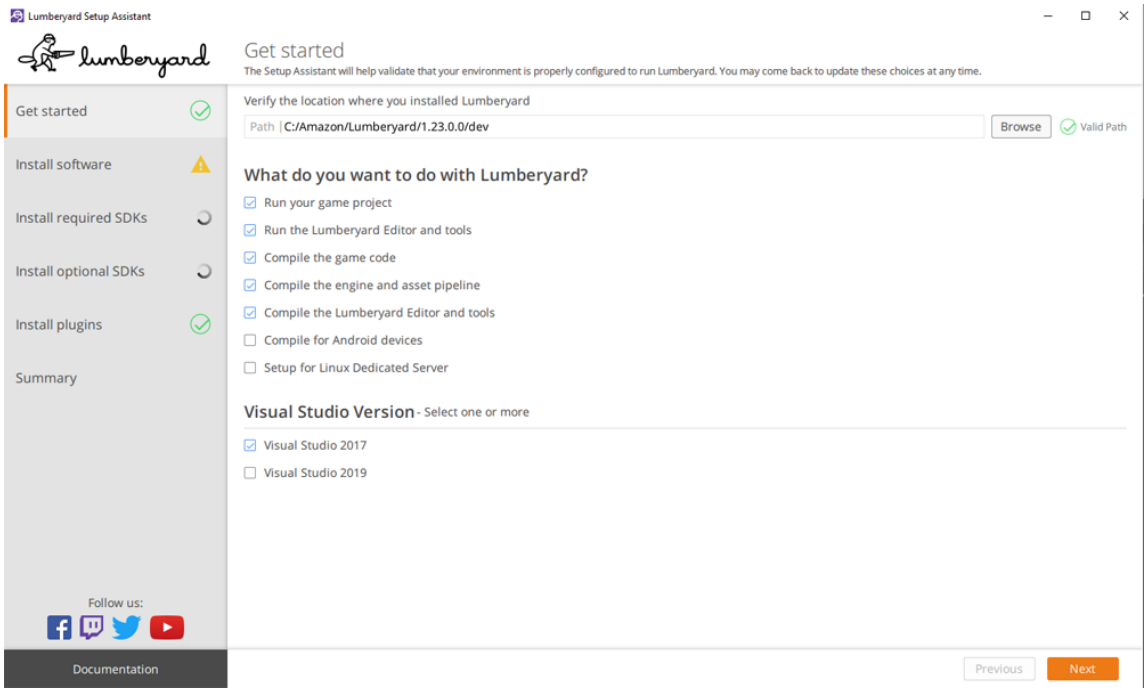
You should run Setup Assistant periodically, especially after you make changes to your environment, to validate and repair settings and paths. You can also customize Setup Assistant with a configuration file to easily integrate your specific directory structure.

> **Note**
> Some Setup Assistant options require the installation of third-party software and licenses, so make sure that you consult the terms of service before installing third-party software.

Lumberyard Setup Assistant custom installation

With **Custom Install**, you specify how you would like to use Lumberyard. Setup Assistant downloads third-party software and validates that the environment is properly configured based on your choices.

1. Custom installation begins with specifying how you intend to use Lumberyard.



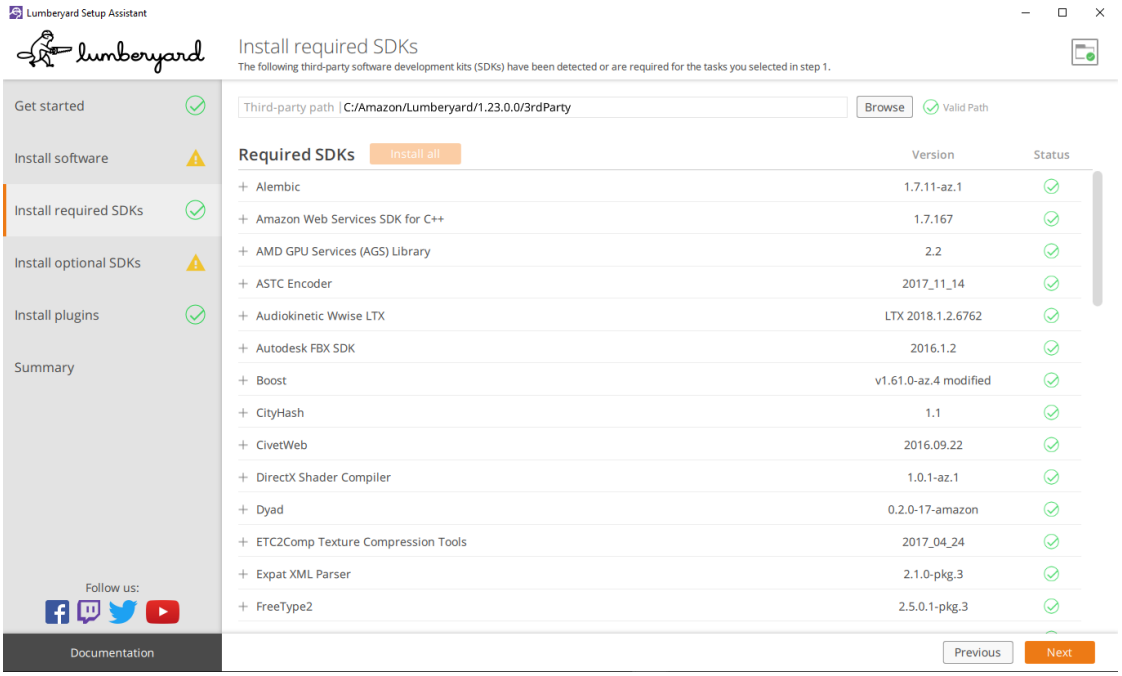
2. Verify the **Path** is correct for your Lumberyard installation.
3. Select options based on your development needs:
   - **Run your game project**
   - **Run the Lumberyard Editor and tools**
   - **Compile the game code*** - Select this option if you intend to create new projects with Lumberyard.
   - **Compile the engine and asset pipeline*** - Select this option if you intend to make changes to the engine or asset pipeline.
   - **Compile the Lumberyard Editor and tools*** - Select this option if you intend to make changes to Lumberyard Editor or other tools.
   - **Compile for Android devices***
   - **Setup for Linux Dedicated Server***

   Note
   If you select any of these options, you might need to perform additional tasks, such as installing Microsoft Visual Studio and installing additional required SDKs. These tasks display on the **Install software** and **Install required SDKs** pages.

Select **Visual Studio 2017**, **Visual Studio 2019**, or both.

Important
The version(s) of Visual Studio selected here will be enabled as build platforms, and Visual Studio Solutions will be generated.

- Once you are satisfied with your selections, choose **Next** to install required software and SDKs.
  1. The **Install software** and **Install required SDKs** pages display a red icon if a requirement cannot be found and a green checkmark for installed requirements. Missing optional software and SDKs display a yellow icon. Follow the instructions on each page to install the required software and SDKS.



Note
Ensure the **Third-party path** on the **Install required SDKs** page is correct for your Lumberyard installation.

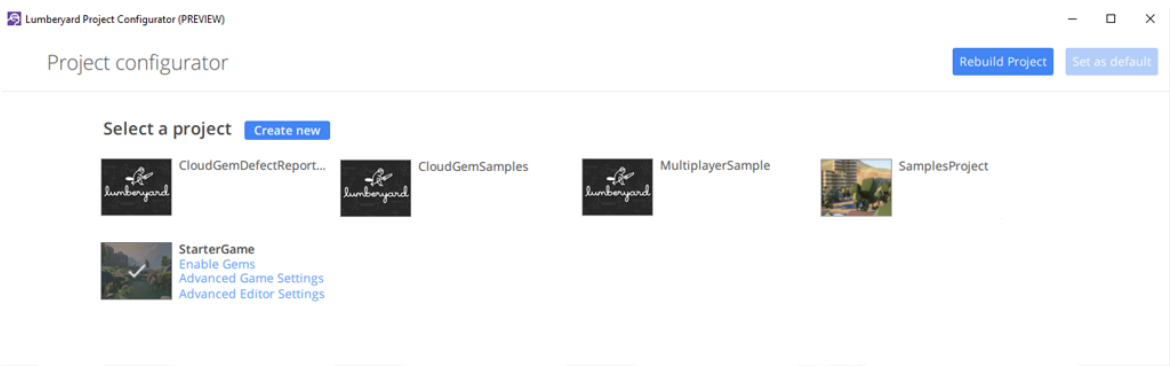
2. Install desired *optional* software and SDKs on the **Install software**, **Install optional SDKs**, and **Install plugins** pages by following the instructions on each page.
3. When you have completed installing software and SDKs, the **Summary** page displays information about your Lumberyard environment. From the **Summary** page, you can launch the Lumberyard Editor by choosing **Launch Editor**. If you'd like to choose an existing project or create a new project, choose **Configure project** to launch Project Configurator.

Manage Lumberyard projects with Project Configurator

Project Configurator configures and manages your Lumberyard projects. If you're continuing from Configuring your Amazon Lumberyard environment with Setup Assistant (p. 15), Project Configurator should be running on your desktop. Project Configurator can also be launched from the Start Menu or desktop shortcuts.

With Project Configurator, you can:

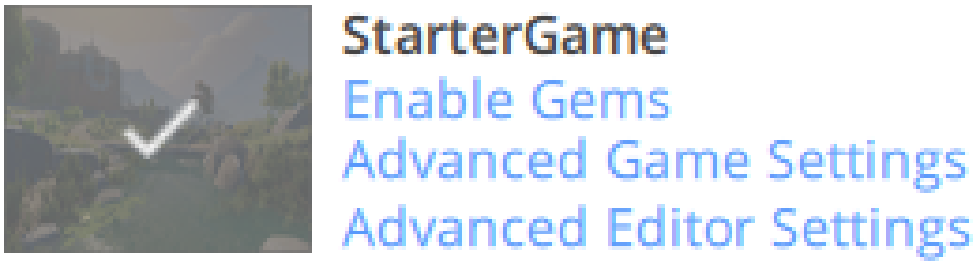
- Set a default project.
- Create new projects.
- Add code or asset features to your project by enabling Gems.
- Create new Gems that can be added to projects.
- Build projects.
- Set advanced game and editor settings per project.



Select a project

Lumberyard has several example projects that you can work with to learn Lumberyard's features. To select a project, choose the project's icon. The selected project is highlighted with a check mark.

> **Important**
> If you're new to Lumberyard and would like to explore its tools and features quickly, we strongly recommend using the default **StarterGame** project. Changing or modifying the default project, or creating a new project will require rebuilding the Lumberyard game and engine. The initial build for a Lumberyard project can take two hours or more depending on your system specifications.

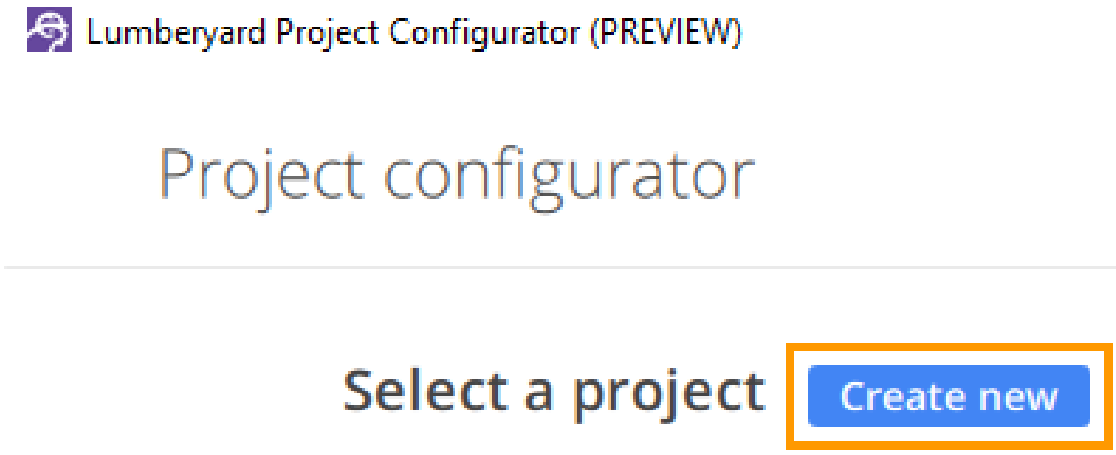


To load a project in Lumberyard Editor, the project must be set to default. While the project is selected, choose the **Set as default** button in the upper right-hand corner of the Project Configurator window. If the **Set as default** button is inactive, the selected project is already the default project.

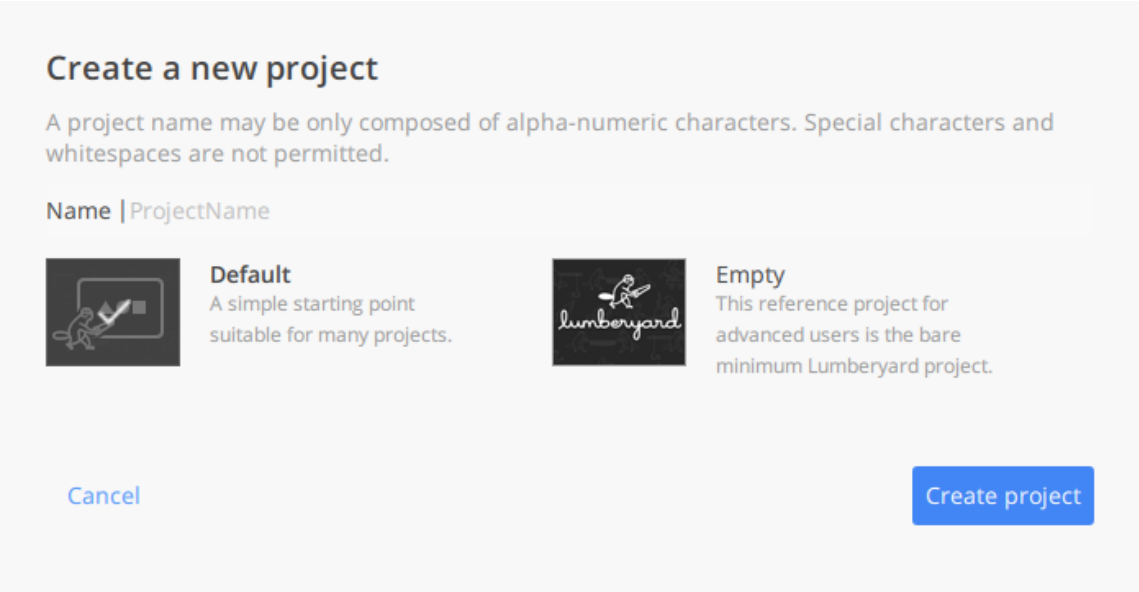
Create a new project

When you create a new project Project Configurator creates a new directory in `lumberyard_version\dev\` with your project's name. Any project-specific files like configuration data, assets, levels, and scripts live within this project directory. Throughout this process, Project Configurator creates the necessary files and Visual Studio solutions, then builds your project. This process can take some time.

1. Choose the **Create new** button in the upper left of the Project Configurator to bring up the **Create a new project** window.

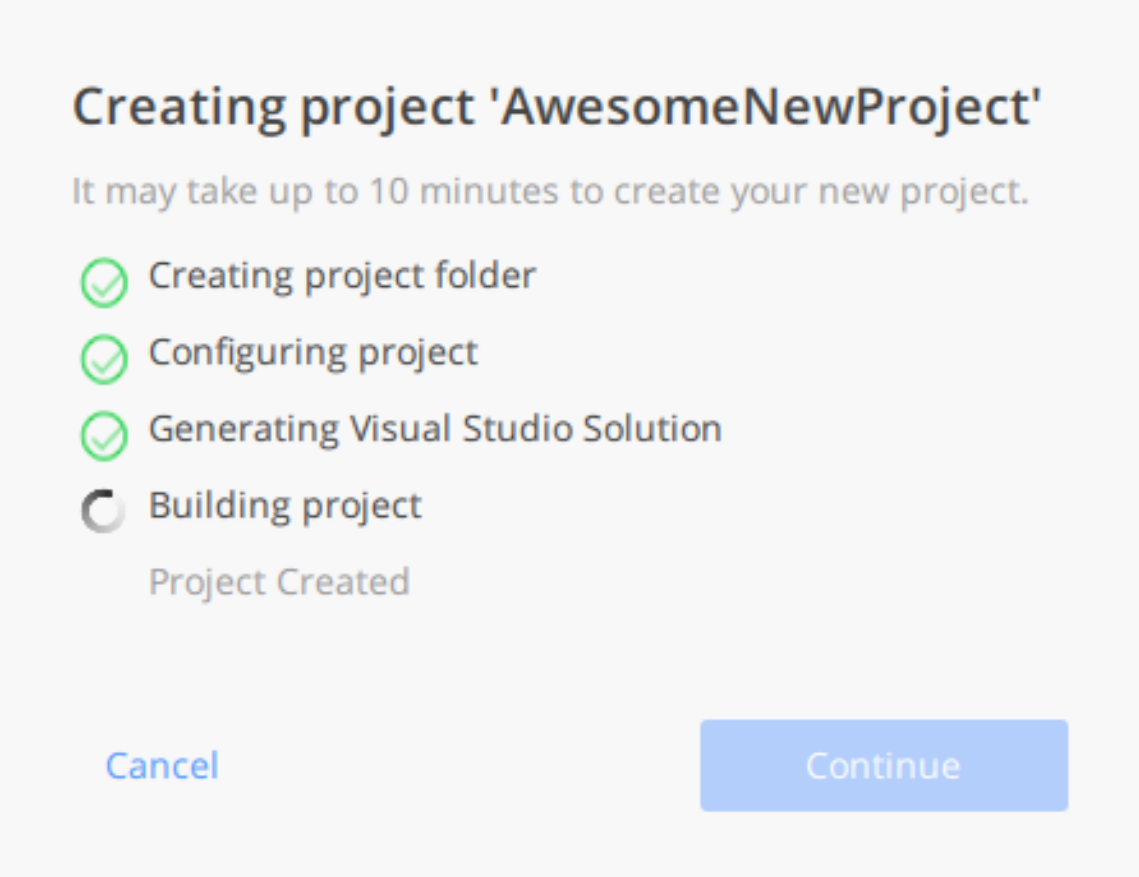


2. In **Create a new project** give your project a name in the upper left and select either the **Default** or **Empty** project template.



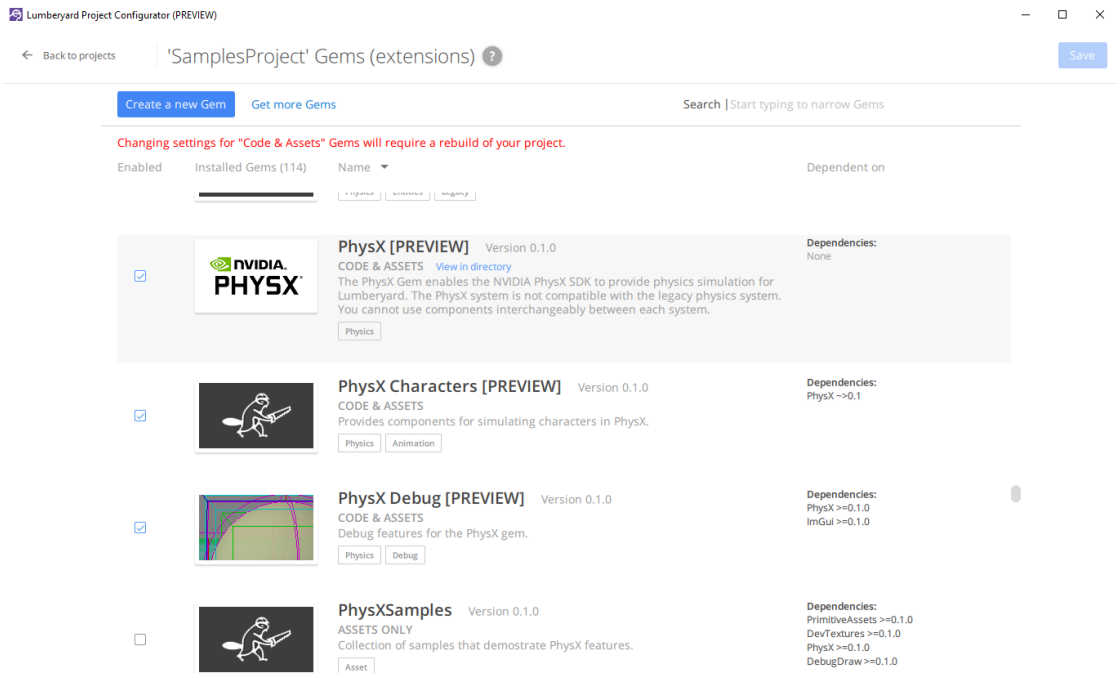
Note
Many features in Lumberyard are implemented in packaged extensions called Gems. The **Default** project template contains a basic set of commonly used Gems. The **Empty** project template contains a minimal set of required Gems. Once you have created a project, you can add or remove Gems.

3. Choose **Create project**. A new window displays your progress as your project is created and built. The build process can take some time based on your system.



4. When your new project has been built, choose **Continue** to return to the main Project Configurator interface. Your new project should be selected. If not, choose the project icon to select it. Choose the **Set as default** button to make your project the default project.
5. Your project is now ready. You can move on to Lumberyard Editor or take the time to explore some of the advanced features of the Project Configurator before moving on. Next to your new project icon are three links:
   - **Enable Gems**

     Choose **Enable Gems** to open the Gems editor. Gems are packaged extensions that add new features and assets to your project. You can create your own Gems here, similarly to how you created a new project. Scrolling down the list of available Gems, you'll notice that some gems are enabled (their box is checked). To add or remove Gems, check or uncheck the box next to the Gem.



Note
Adding or removing Gems might require rebuilding your project, which can take some time. To rebuild your project, choose the **Rebuild** button on the main Project Configurator page.

- **Advanced Game Settings**

  Choose **Advanced Game Settings** to open the game settings view. In this view, you can modify your project's memory allocation and other settings that are exposed by Lumberyard, as well as the Gems that have been added to your project.
- **Advanced Editor Settings**

  Choose **Advanced Editor Settings** to open the settings view. In this view, you can modify Lumberyard Editor settings for your project.

Create with Amazon Lumberyard

Previously in this guide, we covered How Amazon Lumberyard works (p. 5) and Setting up Amazon Lumberyard (p. 12).

Now, let's get started creating your game! In this topic, we provide an overview of creating a game with Lumberyard, and help you identify where to focus your learning, based on your role in game development. If you're proficient in all roles, read everything!

How do I create a game with Lumberyard?

Keeping it Modular

Game development in Lumberyard is best understood if you keep a key Lumberyard design philosophy in mind as you learn: the concept of **modularity**. The Lumberyard game engine, its systems, and its environment are built as a collection of C++ modules. Choosing the right modules based on the combination of your game's design and your workflows will keep your game development process focused and likewise keep the overhead of managing your game project simpler.

Even the Lumberyard Editor – the most common tool associated with Lumberyard – uses these modules, which we call **Gems**. Lumberyard ships with well over 100 Gems as part of the installation, and you can acquire other Gems from third parties, or write new ones yourself. When you build your game, the functionality of these Gems is combined to create the systems of your game. This means that you don't have to hard-code the systems and features of your game. Instead, you can either obtain or create a Gem with the functionality you need – and even maintain the code for it independent of the game! This isolation means that a game developer can work on game-specific AI functionality in a Gem. When the Gem is updated, level designers can access the new functionality after a rebuild of the game project, and without impacting any other Gem. Gems can also add new features to the Lumberyard Editor and provide new asset processing behaviors.

You'll notice this philosophy of modularity in the individual entities you create, up through the most complex functionality you add.

For example, consider the concept of an entity in Lumberyard. An entity can represent just about anything in your game under Lumberyard's component entity system. By giving the entity components, you begin to shape their utility in your game. The components specify entity behaviors and properties.

You might have an entity that you want to see and interact with in your game world that uses all of the following components:

- The default transform component to define its position
- A mesh component to define its visual geometry
- NVIDIA PhysX components to define collision characteristics and other aspects for a realistic, rigid body simulation
- An input component to reference an input event-binding definition
- A script component to automate some sort of behavior, or process input events from the player

- A camera component to allow the entity to be used as a camera

...and many more!

Other entities that you create will be invisible to players. These entities might exist to implement triggers, spawn environmental effects, or reference assets created with tools, such as the game UI that you created in UI Editor.

A major part of assembling a game in Lumberyard revolves around using the Lumberyard Editor to do the following:

- Place and group entities
- Add components to these entities
- Configure properties on these components
- Use tools associated with the components

Tools you might use that are associated to specific components include:

- Script Canvas to create and edit scripts using a visual scripting system, then reference from the Script Canvas component.
- Emotion FX Animation Editor to animate characters, then reference from the **Anim Graph** component.
- Asset Editor to create input bindings that bind raw player input, such as keystrokes, to events, then reference from the **Input** component.
- Audio Controls Editor to setup sound effects that map to Wwise controls, then reference from the **Audio Trigger** and **Audio Switch** components.

...and many more.

Some tools can be opened directly from their associated component. Others require you to open them from the **Tools** menu in Lumberyard Editor. For a tour of Lumberyard Editor, see Introduction to the Lumberyard Editor (p. 29). For an overview of the tools provided with Lumberyard, see the following topic on Tools available in Amazon Lumberyard (p. 34).

The modular nature of Lumberyard means there are additional assets, components, and tools that you can add by enabling Gems in Project Configurator. Lumberyard comes with a library of Gems. Gems can include new code, new assets, or both! You can even write your own. In fact, this is what a game's C++ programmers will often spend their time doing, to help create the gameplay that makes your game unique.

Starting the Journey

Are you starting a new project, maybe to start a prototype of your game? Or to just play around with Lumberyard – take it for a spin and learn what it can do? You'll probably need to know how to get the minimum systems going, which typically include:

- Camera
- Rendering
- Physics
- Input

You'll use the Project Configurator tool to set the project that you're working on. You can use one of the projects that ship with Lumberyard, such as the Samples Project, if you want to learn or play around with something premade. Or, if you want to start something new, this is also where you create a brand new project, based on the template of your choice.

Then, launch the Lumberyard Editor, open one of the sample levels in the Sample Project, and start exploring! We suggest following along with one of these written or video tutorials:

- Learn Lumberyard in 20 Minutes
- Set Up Amazon Lumberyard series
- Creating a Controllable Entity
- Basics of Motion series

Joining a Team

Depending on your role in your game development team and the scope of the game project that you're working on, you might not encounter all of the tools and technologies that Lumberyard provides. In the next section, we'll take a look at how you might want to focus your learning path.

Work as an artist

Lumberyard provides all of the basic tools that you need to import, assemble, and blend animations in a AAA game, creating new worlds or environments for the player to visualize, interact with, and experience.

While much of your work as an artist might involve using tools outside Lumberyard, you will use Lumberyard tools when you want to control your character animations, and manipulate the look and feel of your assets or the environment. You might be working with designers to create the game environment.

As an example, let's take a look at the workflow for setting up an actor in your game.

1. Outside of Lumberyard, you create a character model, materials, textures, and rig for the character.
2. Lumberyard's Asset Processor automatically processes source assets into platform-specific game assets, ensuring new or modified files are ready to use in Lumberyard as soon as possible. Use FBX Settings if you want to modify the processor settings.
3. You import your actor file in Animation Editor and create a motion set to specify the motions that you want for your character.
4. Next, you create an animation graph using nodes.
5. Then, you build a blend tree to blend the animations together.
6. When you've built and previewed the animations and are ready to try them out in a game environment, you can switch over to Lumberyard Editor.
7. In the Lumberyard Editor, you create or open an existing test level.
8. To see your animated character, you need an entity with:
   - an **Actor** component to create a controllable character with the actor file from Animation Editor and a material linked to your actor asset.
   - an **AnimGraph** component to use the animation graph and motion set assets that you created in the Animation Editor.
9. To control your character in your level, you might want to work with a gameplay designer or programmer at this point to add an **Input** component, **PhysX** components, and script components so you can run through and playtest all of your character's animations in your game's specific environment.

We suggest that you begin your learning path by browsing the following set of Lumberyard tools and technologies, and then focusing on the ones that apply to your needs:

- Lumberyard Editor
- Asset Pipeline
- Component Entity System
- Component Reference
- Emotion FX Animation Editor
- FBX Settings Tool
- Gem Library
- Cinematics and the Track View Editor
- Shaders and Materials
- Terrain and Environment
- Vegetation Editor

Some specific tutorials you might want to look at include:

- Import Assets into Lumberyard
- Animation System and Emotion FX
- Lighting

Work as a designer

Lumberyard Editor is an important, core tool for game designers and level designers. It's where you create your levels, populate them with entities, and assign components to those entities. It also provides access to important tools such as the UI Editor for UI designers, the Audio Controls Editor for sound designers, and Script Canvas for all designers who will be working with the visual scripting system in Lumberyard.

Here's how you might start out:

1. When you start Lumberyard Editor for the first time, you can create a new level, or you might prefer to open one of the existing levels in the Samples Project to play around with. If the Samples Project is not your current project, use Project Configurator first, and set it as the default project.
2. You start populating your level in the viewport by creating entities. In Lumberyard, an entity can be just about anything, from the static objects you see, to the triggers you script, to the placeholder objects that reference game UI.
3. You add components to your entities through the Entity Inspector tool.
4. To provide player control for your entities, you'll need an input binding. Using the Asset Editor tool, you bind raw player input from keyboard, mouse, and game controllers to events that you create. Then you can listen for and respond to these events using one of the scripting tools.
5. Script Canvas and Lua are common scripting tools that are used in Lumberyard. Script Canvas provides a visual, node-based scripting system for scripting your gameplay logic, while Lua provides a more traditional scripting environment based on the Lua API. Add a Script Canvas component or Lua Script component to an entity. Develop a script in one of the script editors. Then you can add that script to the component to control that entity at runtime. For example, using a script, you can control an entity by responding to player input events, spawning dynamic entities at runtime, producing visual effects, and much more.
6. As you populate your world with entities, you'll learn that a great way to save time is to use the slice system. Slices are a type of prefab system that allows you to group and nest component entities together, save the group as a slice, and then create multiple instances of that slice throughout the

levels of your project. In each instance of the slice, you have the ability to make changes to that specific instance. This change is referred to as a slice override. You can also choose to save the override to the original slice, which then pushes that change to every other instance, too.

7. When you're ready to author game UI, you use the UI Editor, where you can establish a UI canvas and layout your interface and script your workflows.
8. When you're ready to add sound to your game, you establish audio events and triggers in the Audio Controls Editor. You can then add this audio to entities through components, and script its playback using one of the scripting systems.

To get the most out of Lumberyard, browse the Components library and the Gems library to see what Lumberyard has to offer. Then talk to your programmers to see what additional components need to be authored for your game.

We suggest that you begin your learning path by browsing the following set of Lumberyard tools and technologies, and then focusing on the ones that apply to your needs:

- Lumberyard Editor
- Asset Pipeline
- Component Entity System
- Component Reference
- Gem Library
- Script Canvas
- Lua Editor
- AI Navigation
- Audio Controls Editor
- Emotion FX Animation Editor
- Cloud Canvas
- UI Editor

Some specific tutorials you might want to look at include:

- Creating a Controllable Entity
- Script Canvas series
- Basics of Motion series
- Working with Slices
- Simulate Physics
- Modify Terrain
- Import Assets into Lumberyard
- Animation System and Emotion FX

Work as a game engineer

As a Lumberyard game engineer, you will likely need to learn both how to support the designers and artists in the design of the game and environment, and how to author individual components. These new components can then be added to entities in the Lumberyard Editor to create custom gameplay. You might also learn how to develop new Script Canvas nodes. These new nodes can then be used by designers in the Script Canvas editor to handle new events that you created, or change the properties of

your new components. On a larger scale, when you need to work on a system that can be distributed as a shareable container of code and assets, you can learn how to create a Gem.

We suggest that your learning path looks like this:

1. Follow the intro tutorials to learn about the component entity system and the existing library of components.
2. Browse the Gem library to see examples of the larger functionality they can add compared to individual components.
3. Learn about authoring your own components and Gems, where you will also learn about working with EBuses, Lumberyard's event bus and general-purpose messaging system; AZ Modules, a collection of C ++ code built as a static or dynamic library; and more.

Here is a basic set of Lumberyard tools and technologies to focus on:

- Lumberyard Editor
- Programming Concepts
- Gems
- Component Entity System
- Component Reference
- Programmer's Guide to Entities and Components
- Emotion FX Animation Editor
- Script Canvas
- Lua Editor

Some specific tutorials you might want to look at include:

- Creating a Controllable Entity
- Import Assets into Lumberyard
- Script Canvas series
- Working with Slices
- Working with Gems
- Simulate Physics
- Animation System and Emotion FX

Introduction to the Lumberyard Editor

Lumberyard Editor is your primary workspace. From here, you access all of the tools to design, create, test, play, and deploy your project. If you have used other professional engines or 3D animation packages, you'll find the user experience familiar and adapt to Lumberyard Editor quickly.

Lumberyard Editor can be launched from the start menu or the Lumberyard Editor desktop icon. When Lumberyard Editor launches, you're given the option to create a new level or load an existing level. If you're using one of the example Lumberyard projects, you will find example levels in the **Levels** directory of the project. If you're working with a new project, you must create a new level.

For a 20-minute crash course on navigating the **Perspective** viewport, customizing the Lumberyard Editor layout, creating entities, and working with components in Lumberyard Editor, see the following video tutorial.

Note
Amazon Lumberyard version 1.25 introduced an updated interface. You will notice differences in the visual interface and user experience viewing this video while using Lumberyard 1.25 or later; however, the information in the video tutorial is still applicable to the fundamentals of using Lumberyard Editor.

The Lumberyard Editor default layout

The default layout of Lumberyard Editor contains the most commonly used tools in a configuration, similar to other content creation applications. The core workflow of Lumberyard is to create and place *entities* in a level, so the default layout contains a menu bar, two toolbars, and five tool panes focused on entity creation and placement.

You can customize the layout through drag and drop, and save to a custom layout through the **Layouts** option in the **View** menu of the main menu bar. Drag the separator bars between panes to resize the panes. Drag the title bar of a pane to tear off the pane. The pane can be dropped anywhere in the layout or dropped outside of Lumberyard Editor as its own window. To restore the default layout, select **Default Layout** from the **Layouts** option in the **View** menu of the main menu bar.

1. **Perspective** – This 3D viewport is a real-time view of your level. In **Perspective**, you create and place entities, and view and play your project.

   Right-click **Perspective** in the title bar of the pane to open the perspective menu. From the perspective menu, you can toggle visibility for various helpers such as the construction plane, icons, and guides. You can also select an aspect ratio, view through various cameras placed in the level, create new cameras from the current view, and split the **Perspective** pane into multiple views.

   Right-click in **Perspective** to open the context menu to create entities and *slices*, which are reusable assets that contain multiple entities. From the context menu, you can also create *layers* that you can use to organize entities and slices in your level.

2. **Entity Outliner** – The **Entity Outliner** displays a list of entities, slices, and layers in the current level.

   Right-click in the **Entity Outliner** to open the context menu to create entities, slices, and layers. Much of the functionality of the **Entity Outliner** context menu is shared with the **Perspective** context menu. The **Entity Outliner** context menu also has options to find selected entities and slices in **Perspective**, organize the list in the **Entity Outliner** and find slices in the **Asset Browser**.

3. **Entity Inspector** – The **Entity Inspector** displays the components of the currently selected entity. At the top of the **Entity Inspector** is a field for the entity **Name** and an **Add Component** button. The **Add Component** button opens a list of components, sorted by type, that can be added to the entity. Each component has its own set of properties that are displayed in the **Entity Inspector**. All entities contain a transform component that sets the position, rotation, and scale of the entity in the level.

4. **Asset Browser** – The **Asset Browser** browses your project's on-disk assets. Assets such as meshes, animations, and textures are created in third-party applications. Assets such as materials, scripts, and slices are created within Lumberyard Editor. The assets that you create are stored in your project directory. You can also browse default assets that are included with Gems that have been added to your project.

The left of the pane of the **Asset Browser** displays a directory structure that you can browse for available assets. When an asset is selected, the preview pane on the right displays a thumbnail preview and information about the asset, if available.

5. **Editor Console** – The **Editor Console** shows command and process output from Lumberyard Editor and your project. When you load a level, for example, the console displays messages about assets and configuration files as they load, and might display warnings and errors if issues are encountered. You can enter console commands such as setting console variables in the entry field at the bottom of the console. Click the **{x}** button in the lower left of the **Editor Console** to open the **Console Variables Editor**, which provides a simple interface for setting console variables.
6. **Toolbar** – The **Toolbar** provides easy access to various editor tools and features. The toolbar is docked at the top of the editor by default, but you can also dock it vertically on the edges of the editor. To customize the toolbar, right-click anywhere on the toolbar and select **Customize** from the context menu. You can choose which toolbars, views, or modes to include. You can also add commands to a toolbar.
7. **Perspective Toolbar** – The **Perspective Toolbar** at the bottom of the **Perspective** pane displays position information for selected objects. You can also adjust navigation speed; mute audio; go to a specific position; toggle camera collision; run simulations, and enable VR preview mode.

Navigating the Lumberyard Perspective viewport

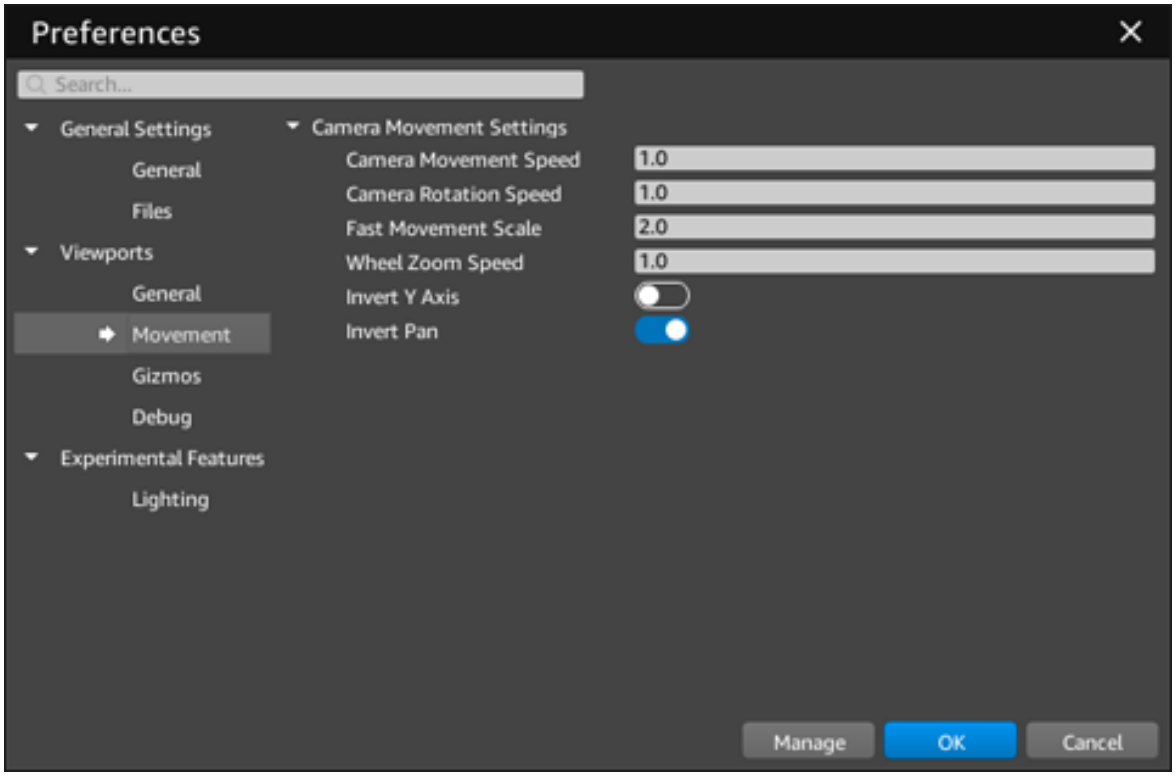
Lumberyard's interaction model will be familiar to anyone who has played a first-person PC game, with a few minor tweaks and additions. Movement is handled by keyboard input, and view is handled by pointer device input.

- **W** – Move forward.
- **S** – Move backward.
- **A** – Move left.
- **D** – Move right.
- **Q** – Move down.
- **E** – Move up.
- **Z** – Focus on selected.
- **Right mouse + drag** – Rotate view, known as *mouselook* in most games.

- **Mouse wheel scroll** – Zoom view.
- **Middle mouse + drag** – Pan view.
- **Left mouse** – Select entity.
- **Left mouse + drag** – Area select entities.

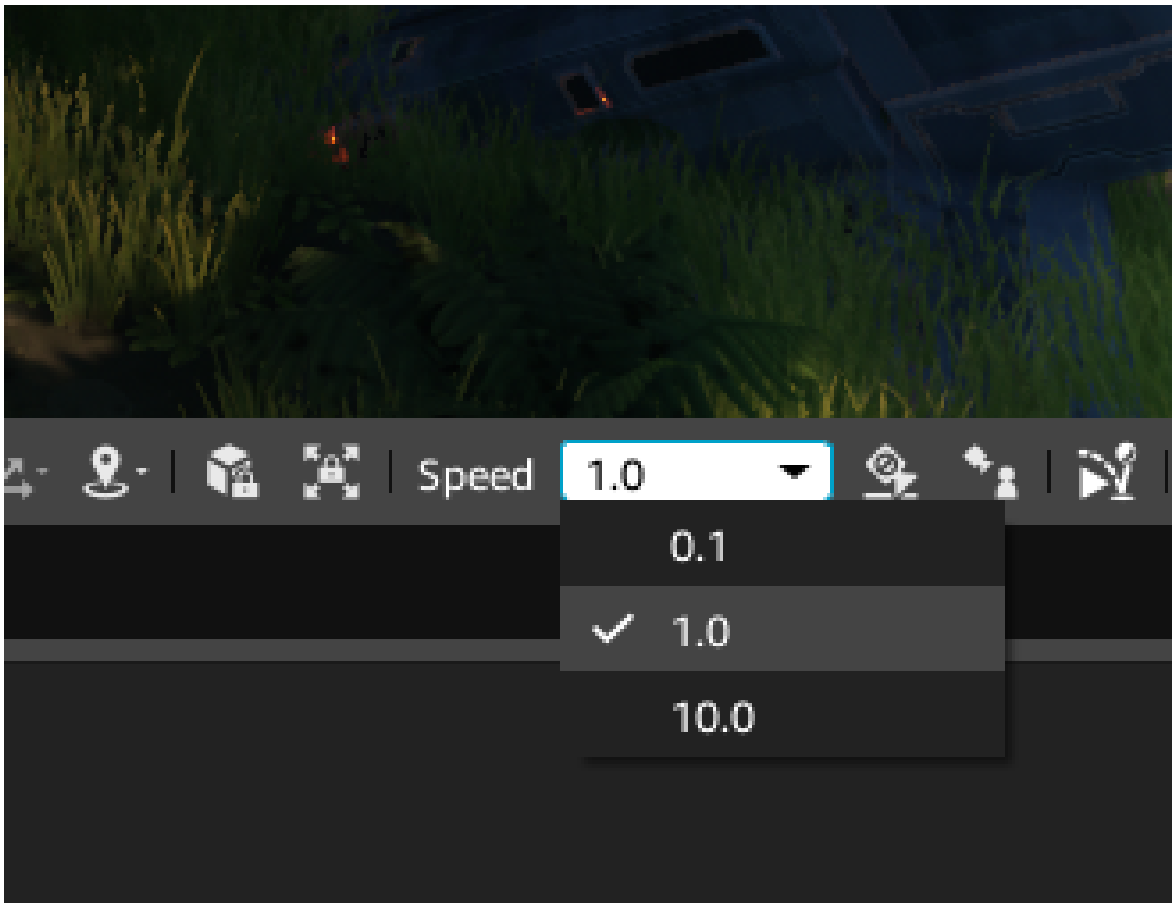
Movement preferences

You might prefer that your editor camera controls behave like a flight simulator. Or you might want to speed up or slow down the default movement or rotation of the editor camera. You can adjust the default editor camera control behavior by setting the **Movement** properties in the **Global Preferences** editor.



Choose **Global Preferences** from the **Editor Settings** group in the **Edit** menu. Select **Movement** under the **Viewports** list on the left. Here, you can invert either mouse axis and adjust the movement speed of the editor camera.

When you have your movement preferences set to your liking, you might find at times that the editor camera movement is too fast or too slow in certain situations. You can adjust the movement speed in the **Perspective Toolbar** at the bottom of the **Perspective** pane.



Enter a floating point value in the **Speed** property to set movement speed. You can also click the arrow to the right of the **Speed** property to set the movement speed to **0.1**, **1.0**, or **10.0**.

Save Perspective locations

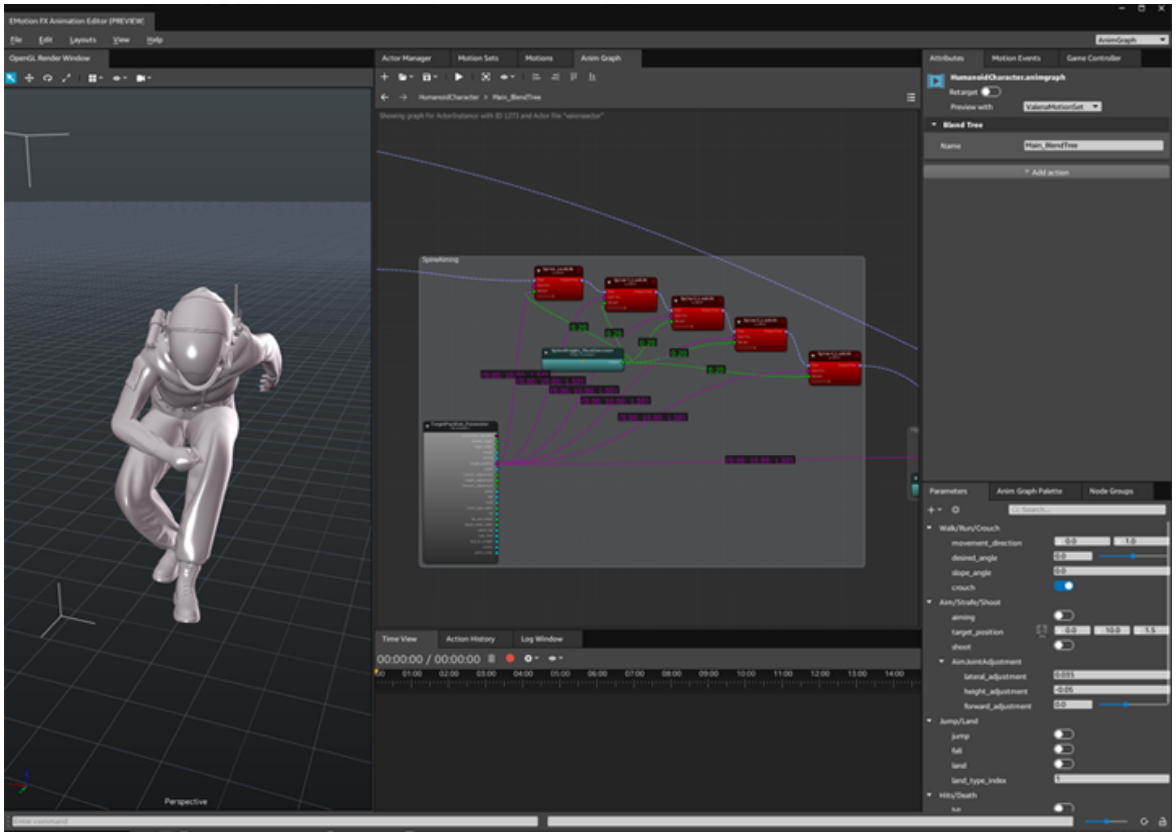
As you build a level, you might find that it's helpful to have preset **Perspective** views saved for later use. You can save the current editor camera view, assigning it to a **[Function]** key. To save a **Perspective** location, press **[Control + Function(1-12)]** To set the **Perspective** view to a saved location, press **[Shift + Function(1-12)]**.

Tools available in Amazon Lumberyard

The default **Lumberyard Editor** layout contains tools common to most development workflows, but there are a lot of additional tools available. Depending on your development role, these may not always be the tools you need - or even the right tools for your job. The **Lumberyard Editor** supports custom layout options so that you can add all of the tools you need to use most frequently right where you need them. Some tools that have multiple panes, such as the **Animation Editor**, can have their own custom and preset layouts.

These tools are available in the **Tools** menu of the **Lumberyard Editor**.

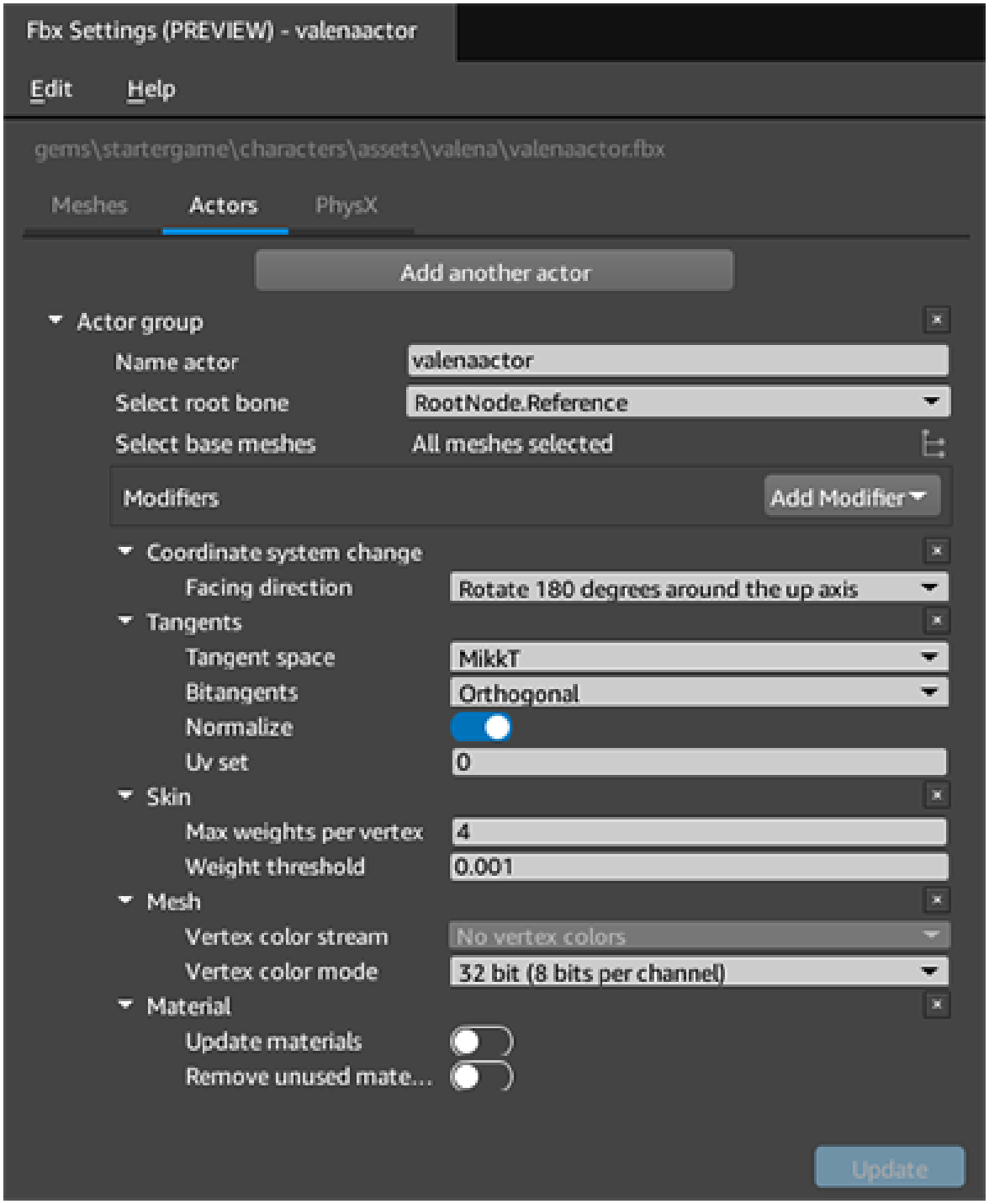
Animation Editor



With **Animation Editor**, build animated *behaviors* for your *actors*.

Use the Animation Editor to build animation loops and set up smooth transitions between them. To create behaviors, begin by loading an actor, which is a skinned mesh, and its corresponding animations. The animations are added to *motion sets*. You create *motion graphs* using motion sets that can blend animation based on states and events. The motion graphs that you create define your actor's behaviors.

FBX Settings



The **FBX Settings** tool converts static `.fbx` meshes, actors, PhysX meshes, and motions into Lumberyard assets.

When you export or copy `.fbx` files to a directory in your current game project, Asset Processor detects the files. Using these files as input, Lumberyard calculates default settings. These settings specify how Asset Processor converts the files into the appropriate mesh, materials, or animation assets.

To customize `.fbx` settings, find the asset in **Asset Browser**. Right-click the asset and select **Edit Settings** to open **FBX Settings**. The available options in **FBX Settings** vary depending on the contents of the `.fbx` file.

Texture Settings Editor

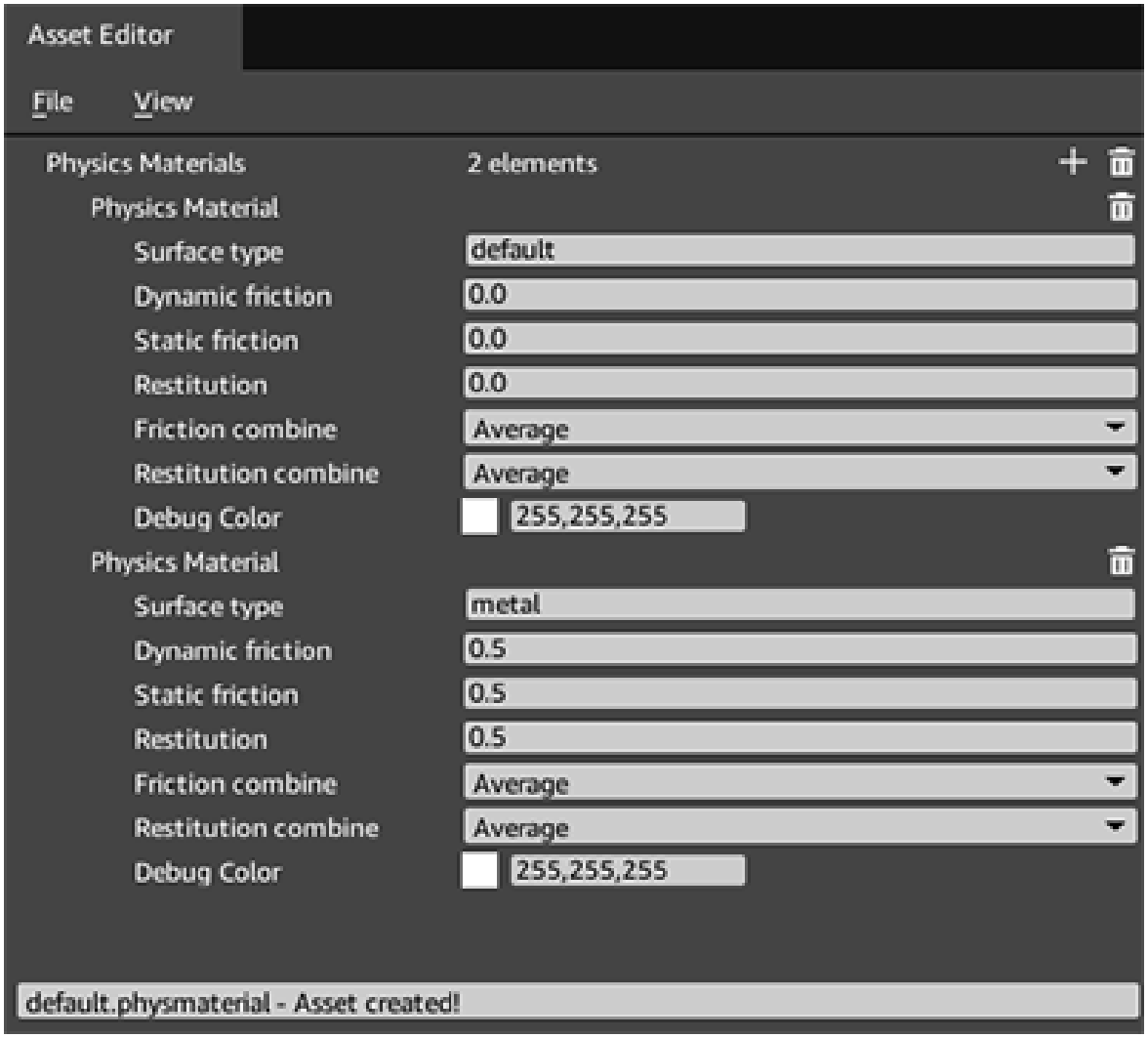
Texture Settings Editor converts image files into Lumberyard assets.

When you export or copy image files to a directory in your current game project, Asset Processor detects the files. Using these files as input, Lumberyard calculates default settings. These settings specify how Asset Processor converts the files into the appropriate textures.

To customize the texture settings, find the image asset in **Asset Browser**. Right-click the asset and select **Edit Image Settings** to open **Texture Settings Editor**. In **Texture Settings Editor**, you can choose from presets based on the texture type and specify settings for various platforms. **Texture Settings Editor** displays a preview image of the texture so you can view the settings results.

- `.bmp`
- `.gif`
- `.jpg`
- `.jpeg`
- `.png`
- `.tga`
- `.tif`
- `.tiff`

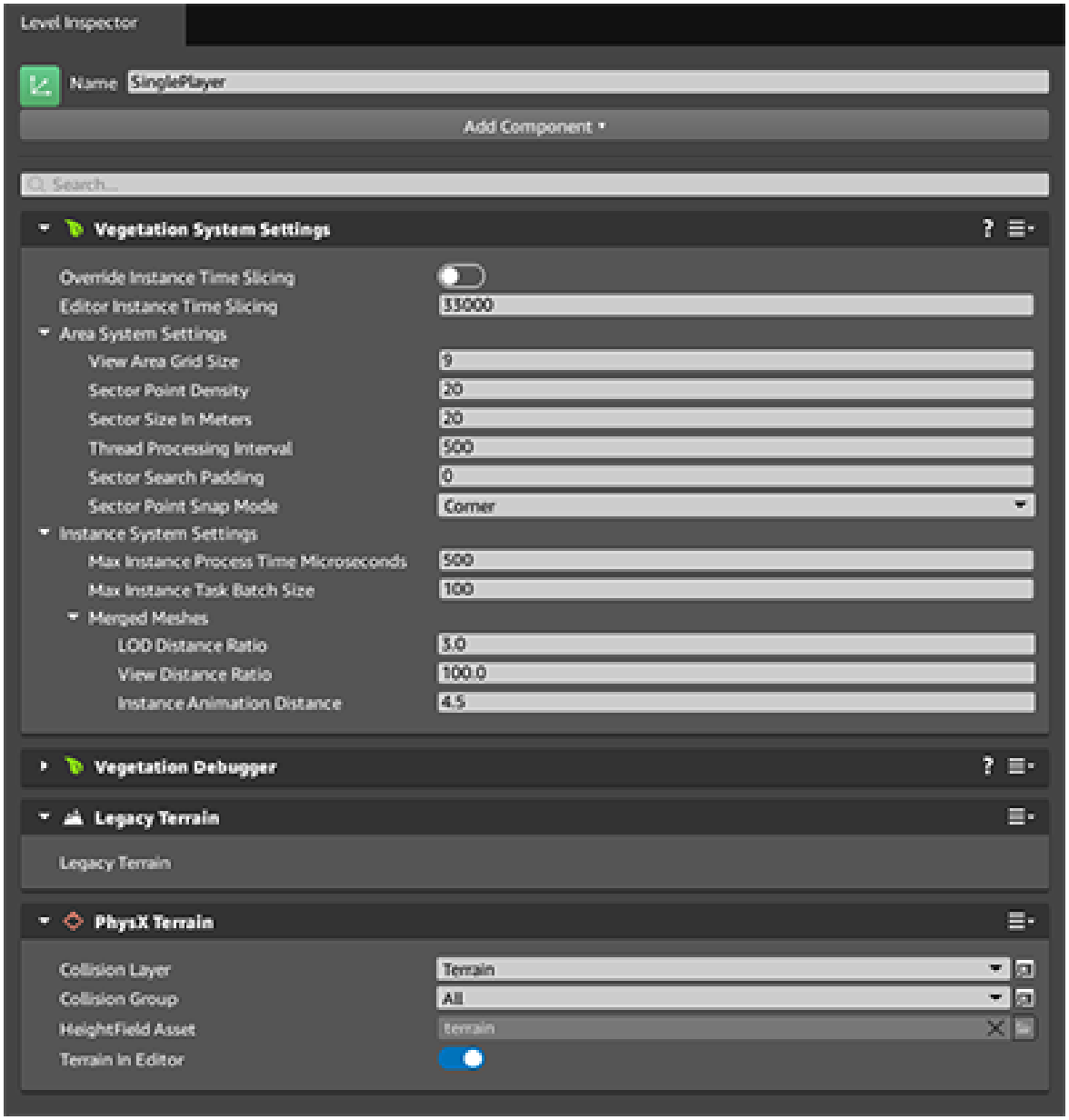
Asset Editor



Create and edit Lumberyard-specific assets in **Asset Editor**.

Lumberyard has a small number of specialized assets such as *script events* that allow scripts to communicate with each other, *physics materials* that give surfaces physical properties like friction, and *input bindings* that bind input to events. Create and edit these specialized asset types with the **Asset Editor**.

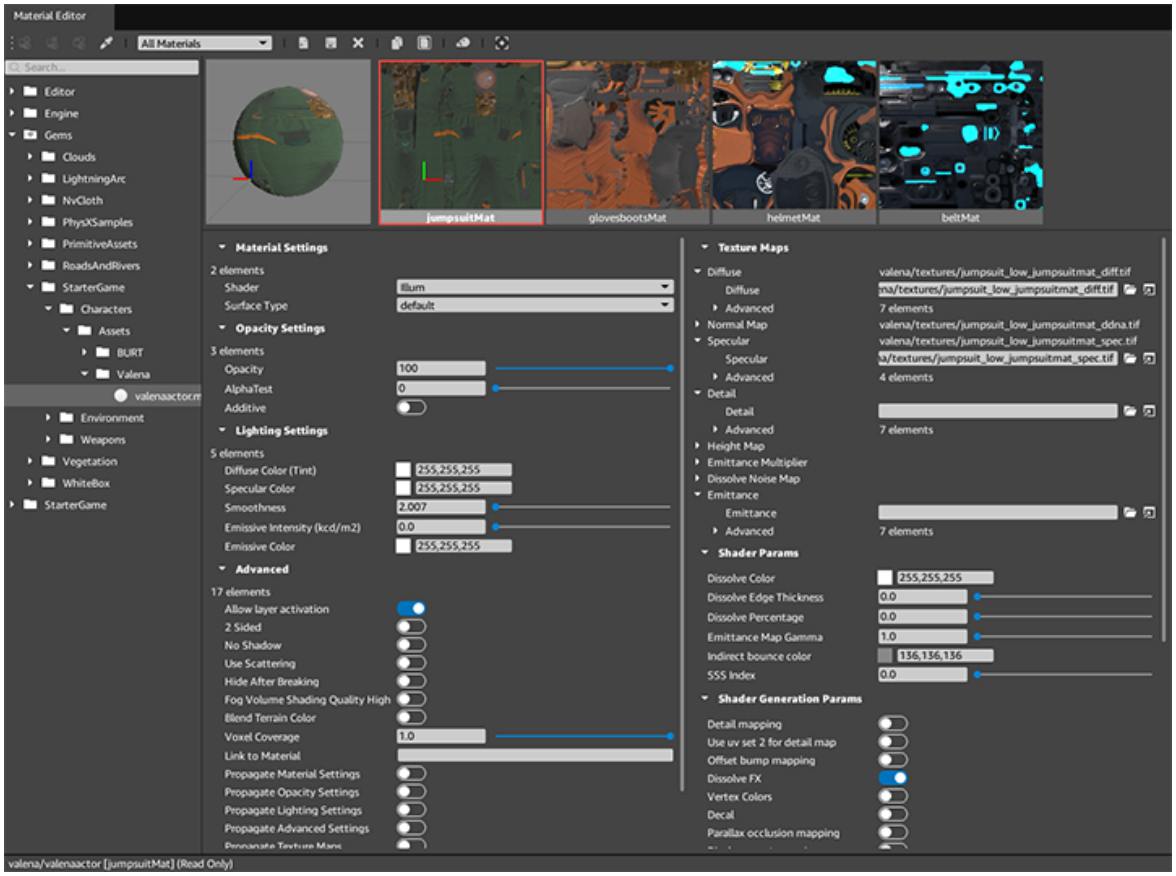
Level Inspector



Add and modify level components in **Level Inspector**.

Level Inspector allows you to add and modify level components, similar to **Entity Inspector**. In **Level Inspector**, you can add terrain components and NVIDIA PhysX terrain components for terrain collisions.

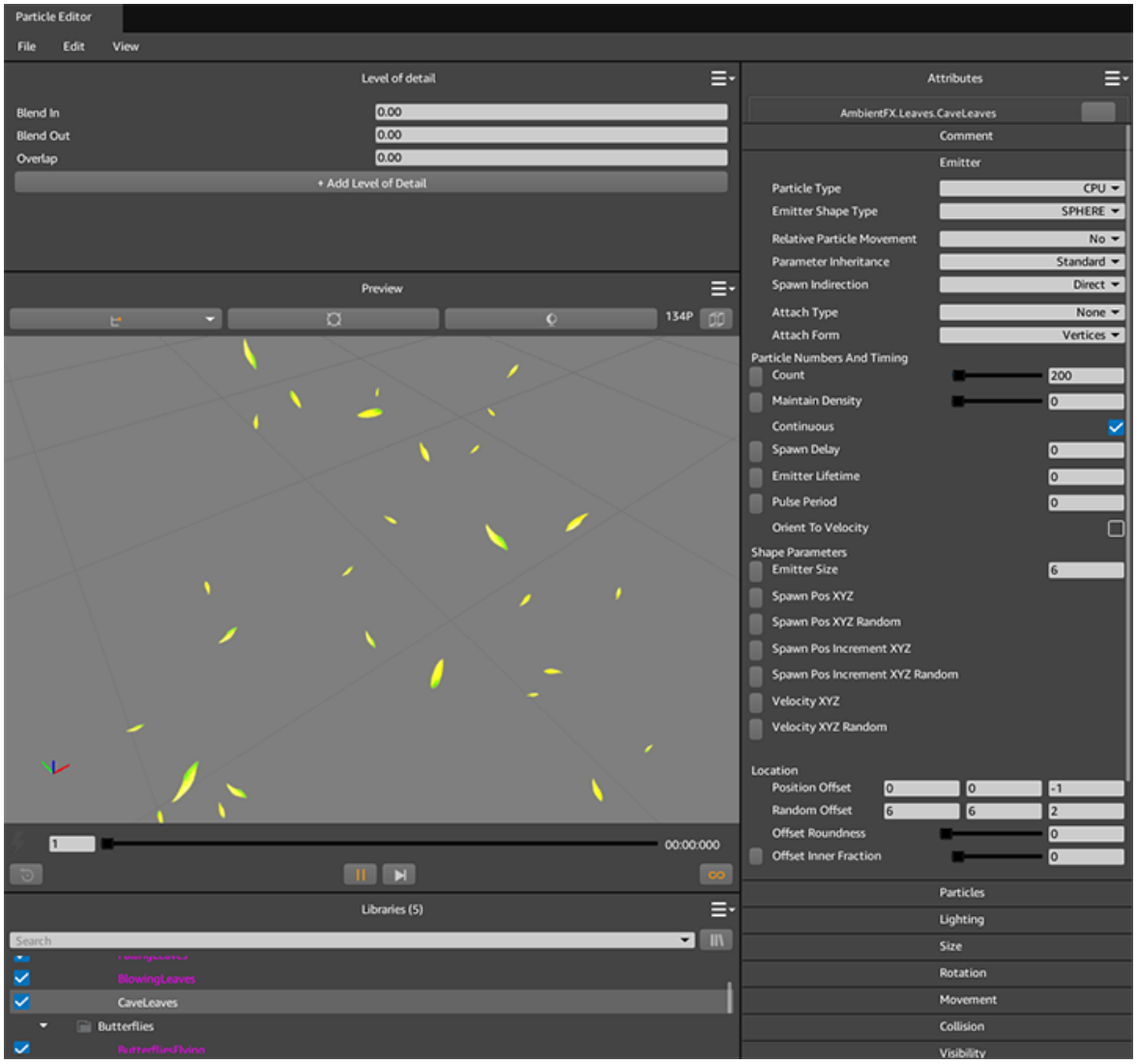
Material Editor



Create and edit the appearance of the entities in your project with **Material Editor**.

A material has a set of properties that determines how its surface reacts to physical actions, other materials, and its environment. **Material Editor** is the primary tool used to create materials and map textures, and lets you configure texture properties like opacity, lighitng effects, shader parameters, and more.

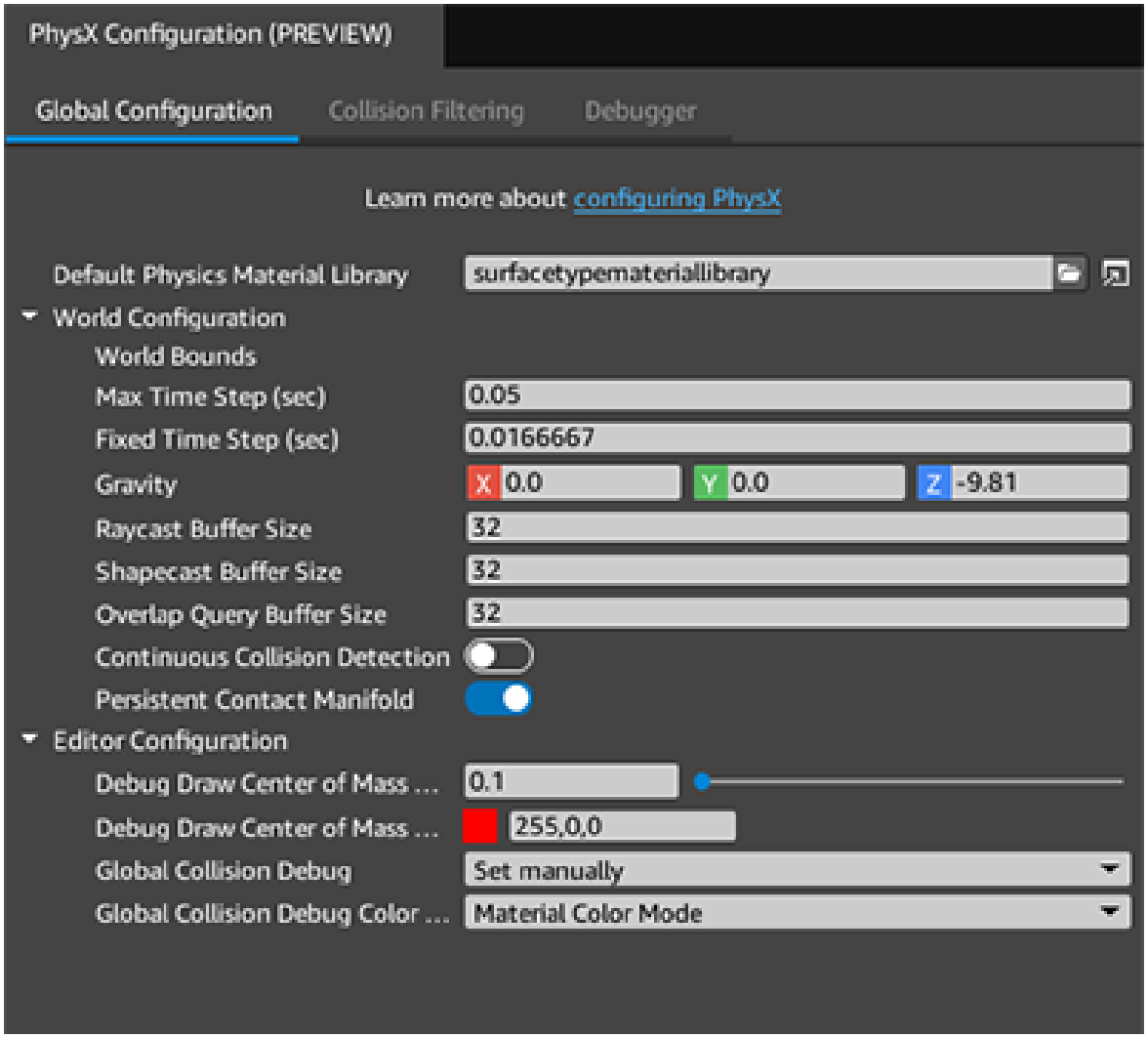
Particle Editor



Build layered, dynamic visual effects with **Particle Editor**.

Lumberyard includes an advanced particle effects system that you can use to simulate environment effects like fire and sparks, or weather effects like fog, snow, or rain. Use **Particle Editor** to create and manage libraries of particle effects in your project.

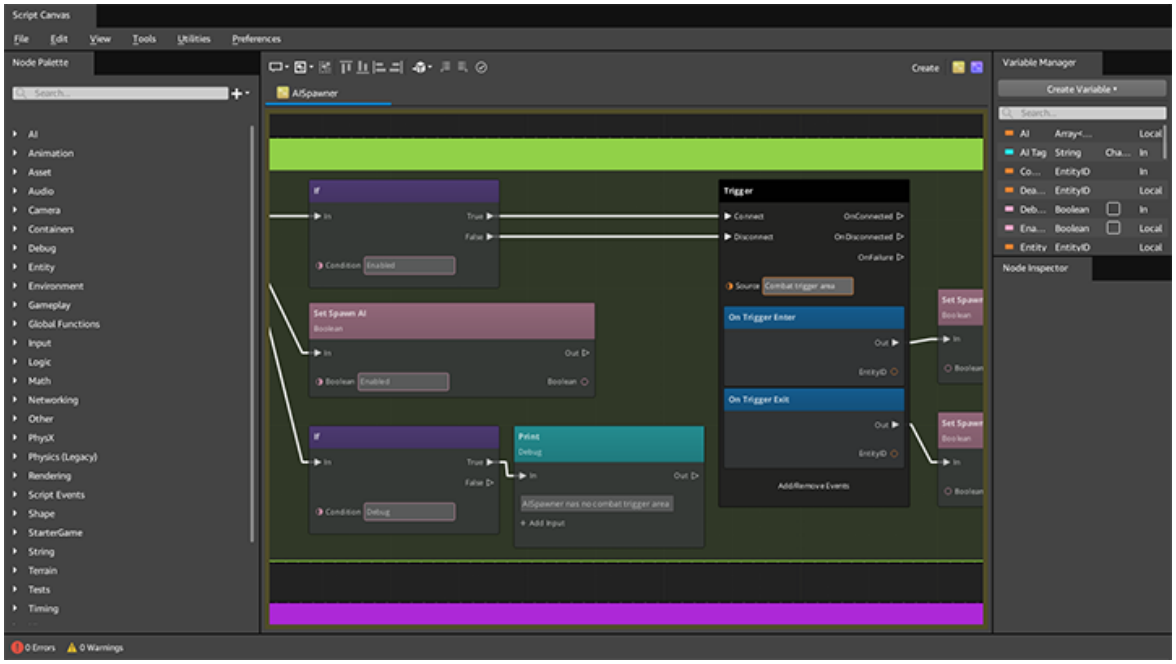
PhysX Configuration



Set the global PhysX properties for your project with **PhysX Configuration**.

Lumberyard integrates NVIDIA PhysX for real-time physics simulation. With **PhysX Configuration**, you can set global properties for NVIDIA PhysX such as gravity, balance simulation performance and accuracy, create filters with collision layers and groups, and set up a PhysX visual debugger.

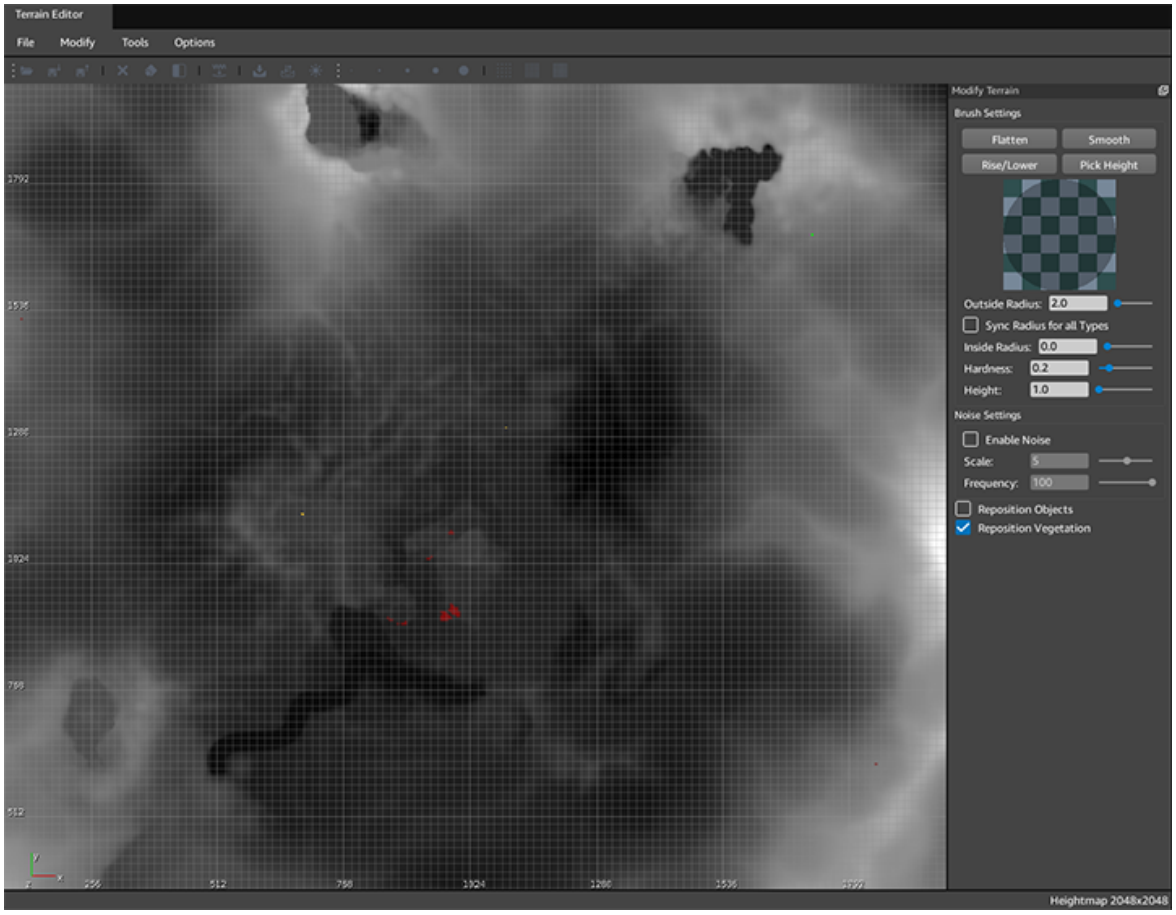
Script Canvas



Program logic and behaviors visually with **Script Canvas**.

Script Canvas is one of Lumberyard's most powerful tools. With **Script Canvas**, you can create behaviors, functions, and logic in a visual programming environment. **Script Canvas** is designed to use Lumberyard's serialization, reflection, modularization, and EBus messaging systems. It's tightly integrated with Lumberyard's component entity system and built on the `AzCore` library. This means that you can create event-driven logic and behaviors without programming experience.

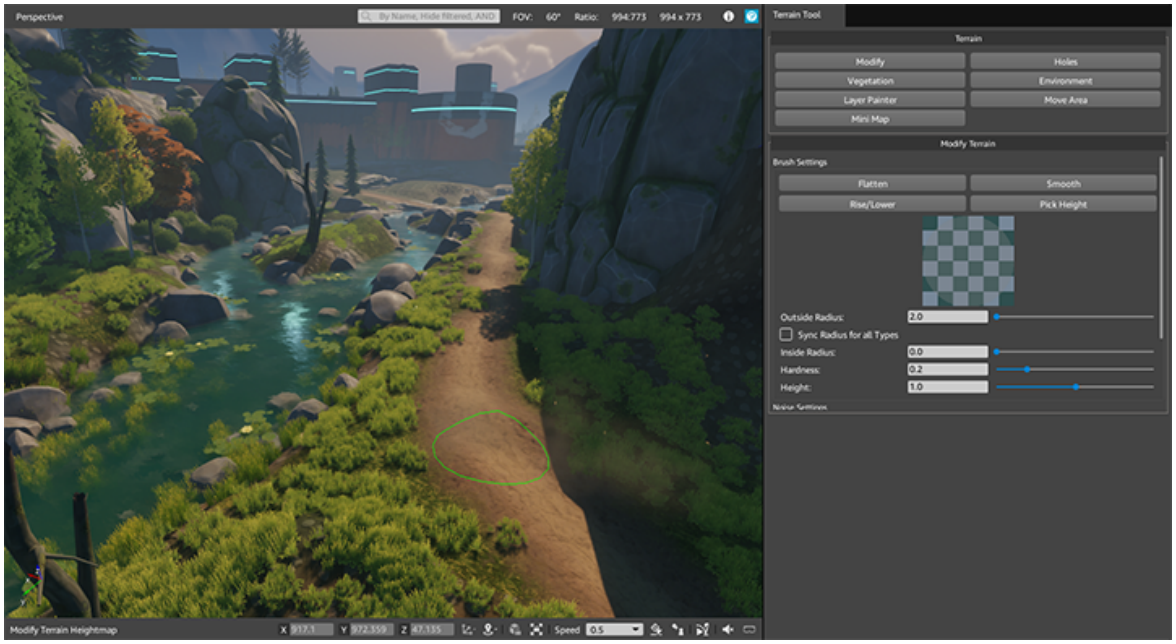
Terrain Editor



Create terrain with **Terrain Editor**.

With **Terrain Editor**, you can paint heightmaps to create peaks and valleys in your terrain and add megaterrain textures for sweeping vistas.

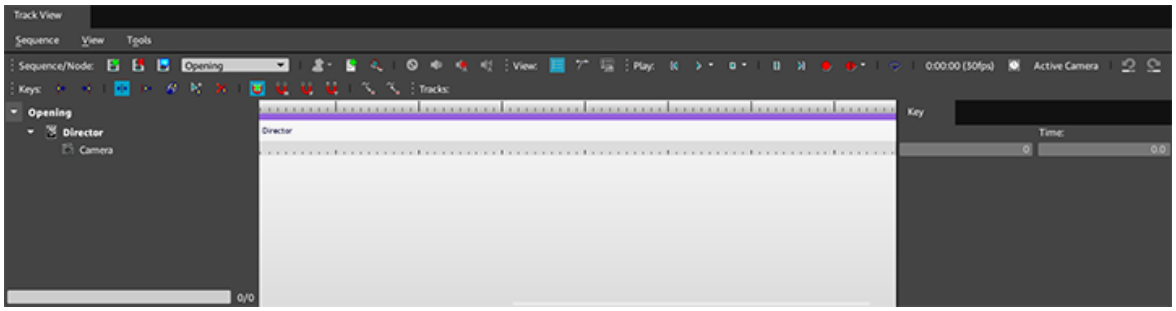
Terrain Tool



Add fine detail to terrain with **Terrain Tool**.

With **Terrain Tool**, you can sculpt accurate details on your terrain and paint texture layers live in the **Perspective** viewport.

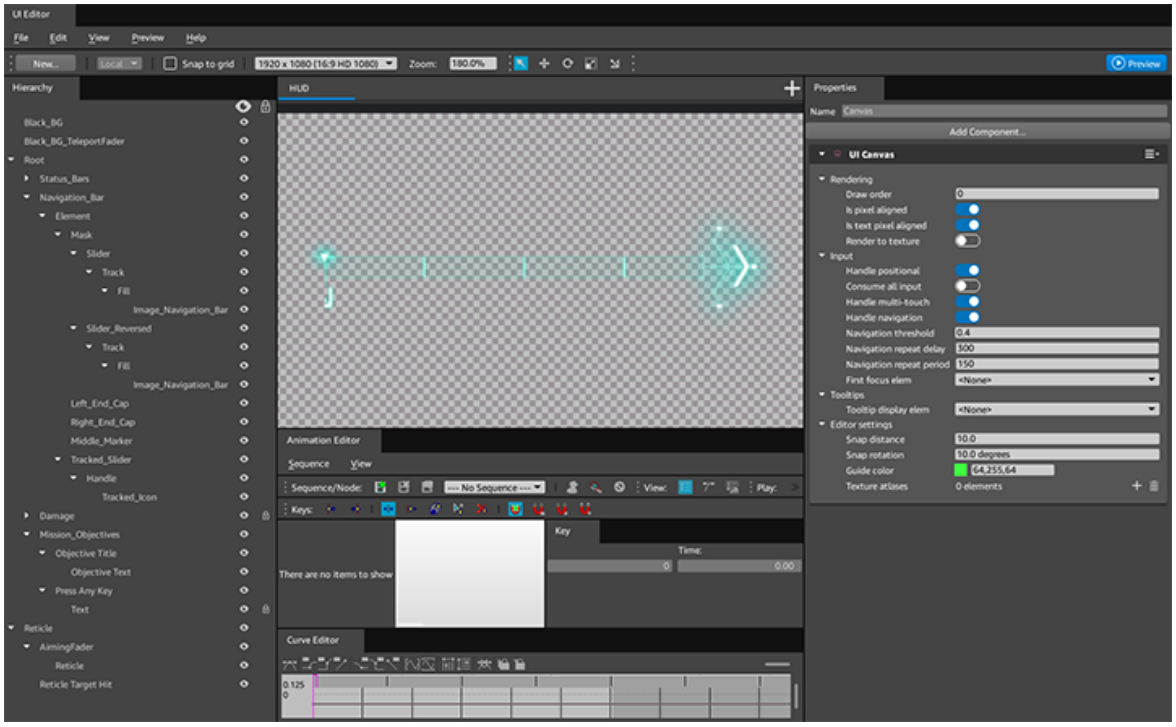
Track View



Create cinematic sequences with **Track View**.

Track View is the primary tool to create and manage cinematic sequences like cuscenes or scripted animation events.

UI Editor



Design dynamic user interfaces with **UI Editor**.

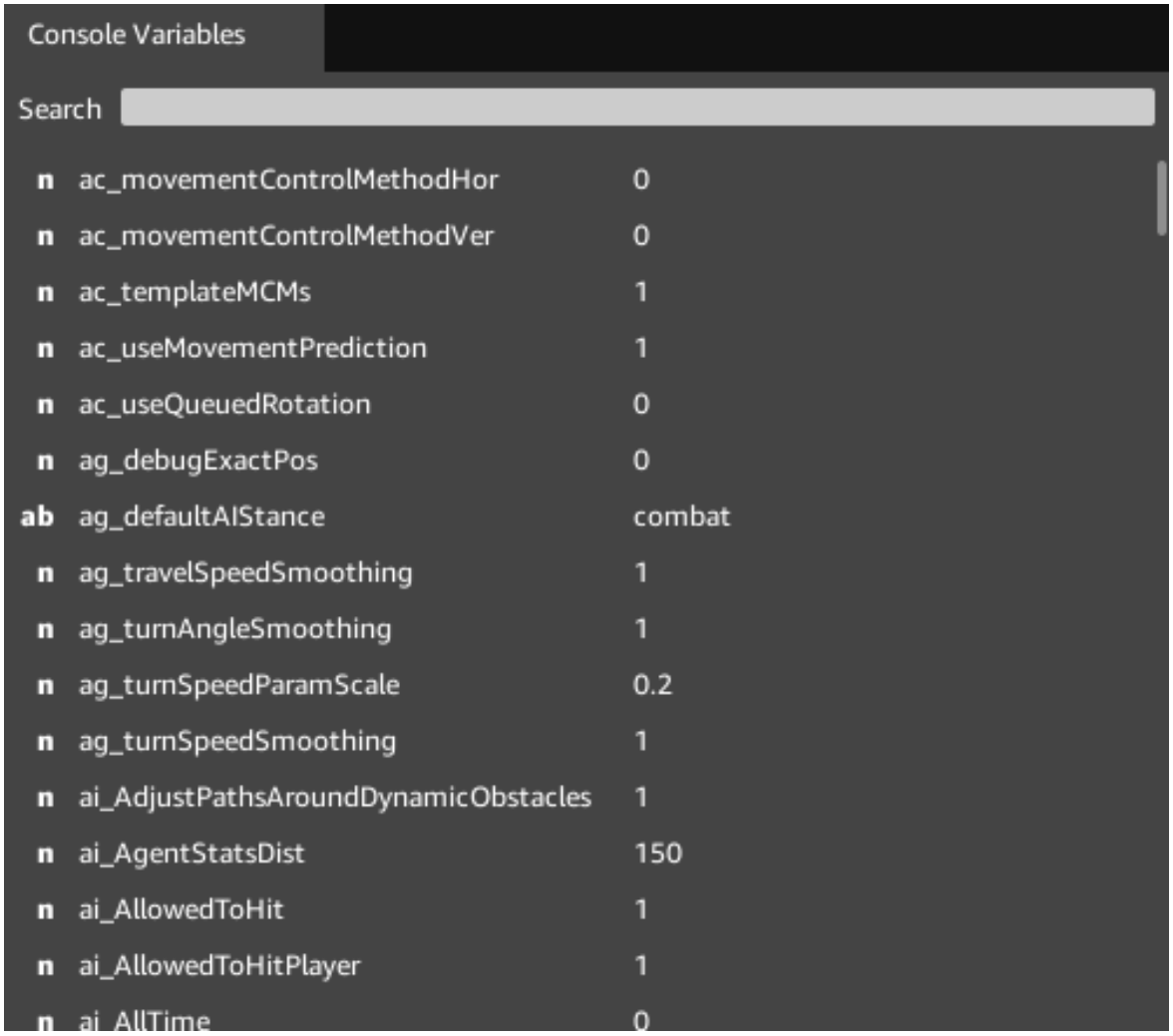
You can use **UI Editor** to create, customize, and animate various game user interface elements and components such as menus, buttons, and heads-up displays.

Audio Controls Editor

Map audio controls in **Audio Controls Editor**.

Your project communicates all actions, events, and parameters to the audio system with Audio Translation Layer (ATL) controls. These ATL controls are mapped to one or more controls inside your selected middleware (Wwise or Wwise LTX). With **Audio Controls Editor**, you can create controls and make connections between the ATL controls and the middleware controls.

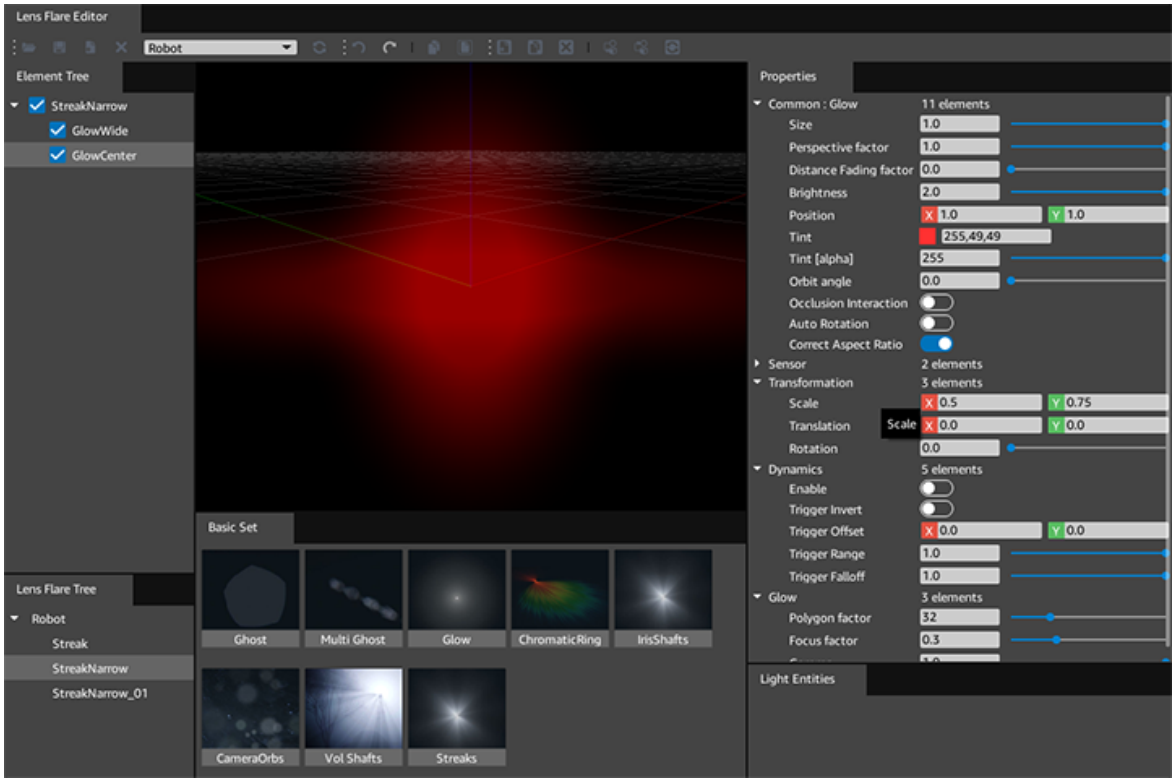
Console Variables Editor



Find and set console variables in **Console Variables Editor**.

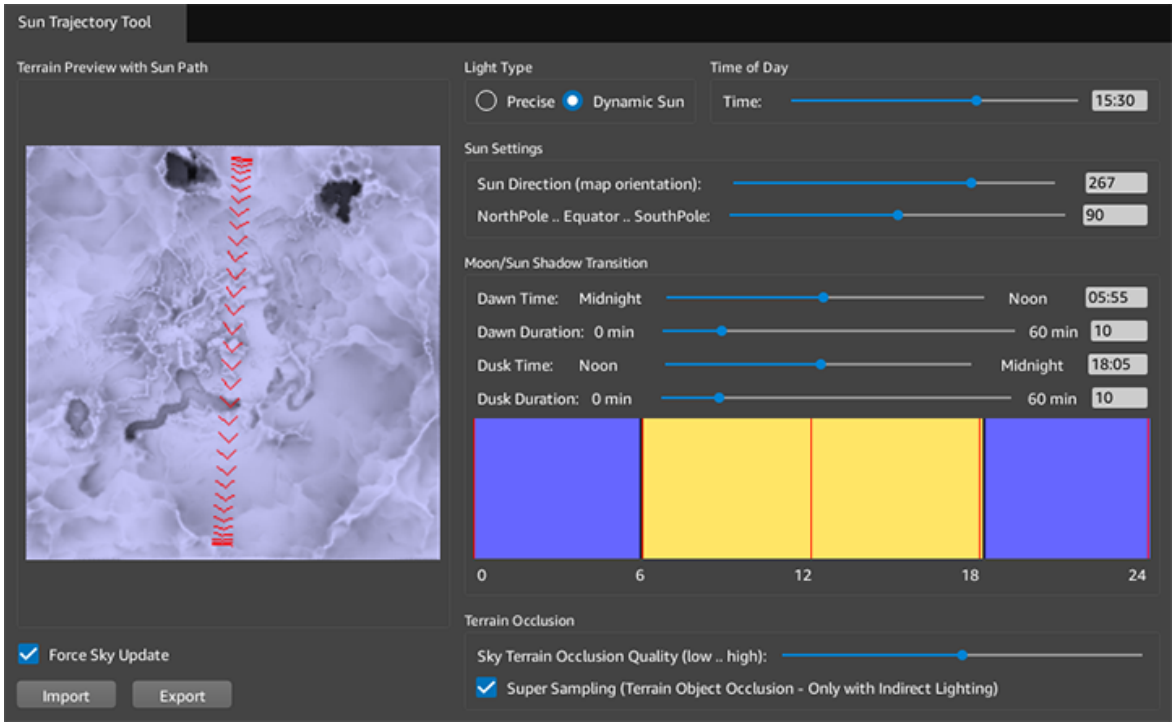
Lumberyard has many console variables,or *CVARs*, that control all aspects of the editor and your project. CVARs can enable and disable debug features, set output verbosity, modify system performance, and much more. **Console Variables Editor** presents a searchable list of available CVARs so you can modify their values.

Lens Flare Editor



Design unique lens flare effects for specific lights in **Lens Flare Editor**.

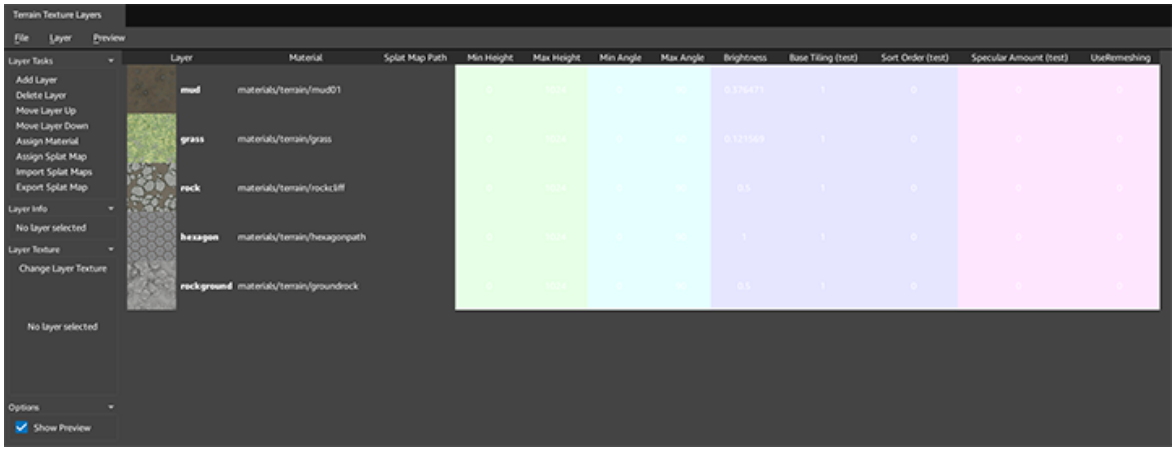
Sun Trajectory Tool



Create dynamic time-of-day lighting with the **Sun Trajectory Tool**.

With **Sun Trajectory Tool**, you define the current time, sunrise and sunset, and sun direction to create dynamic dawn, daylight, dusk, and night skies.

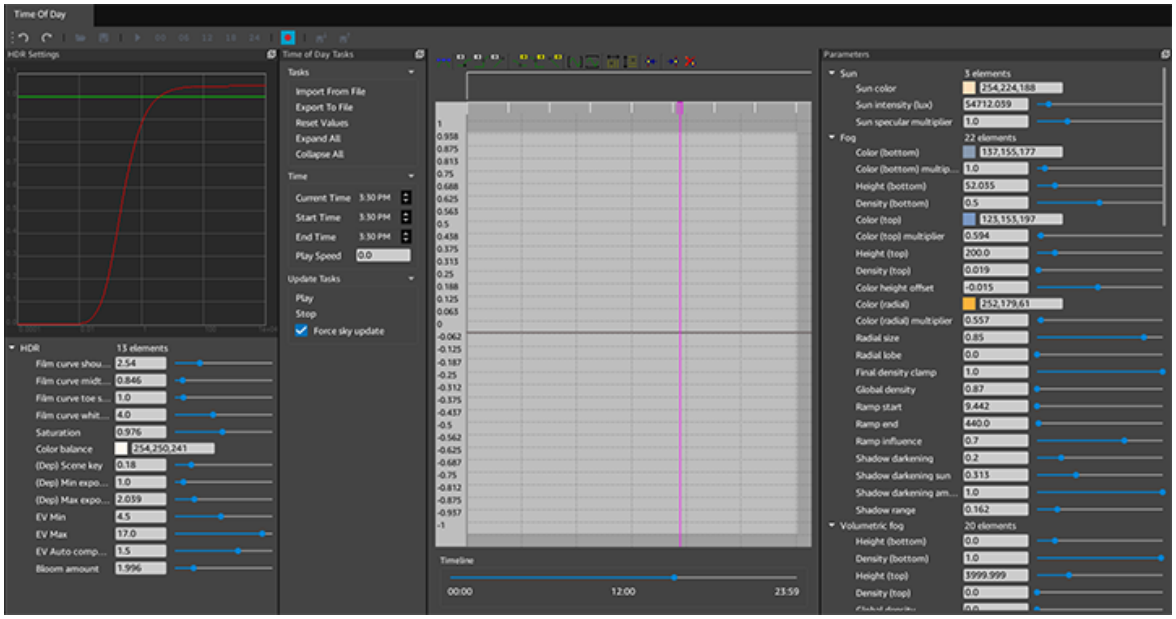
Terrain Texture Layers



Build realistic natural terrain with **Terrain Texture Layers**.

Texture layers are used to create natural-looking terrain that transitions between surface types such as sand, dirt, mud, grass, and stone. With **Terrain Texture Layers** you create and order terrain layers, and apply materials and textures that create transitions between terrain surface types.

Time of Day Editor



Create natural lighting and atmosphere transitions over time with **Time of Day Editor**.

Time of Day Editor configures changes to environment parameters over time to mimic a day-night lighting cycle. The **Time of Day Editor** uses a 24-hour time line graph and a recording function to store changing environment parameter values.

Get started with Amazon Lumberyard

This **Get started** guide is a work in progress and is undergoing continual improvement. Please leave feedback using the **Feedback** link at the top of the web page. More tutorials are coming soon!

Often times, the best way to learn is *to do*. Amazon Lumberyard was designed and created to help you make games, not read docs, so let's make your first game!

We've designed a very simple game that can be built in a few hours with Amazon Lumberyard. As you work through the tutorials in order, you'll learn about the different features and workflows in Lumberyard, and build comfort with the Lumberyard Editor interface.

At the end, you'll have a simple packaged version of a game you can share. It won't win any awards, but you'll understand the myriad features of Lumberyard, from slices and components to physics to scripting to input and audio support.

Note

- If you don't want to read at all, check out the video playlist for this tutorial series. We're not judging. Words are inefficient!

Prerequisites

Have you installed and configured Amazon Lumberyard yet? Nope?

To get started "getting started", you will need the following on your local computer:

- Amazon Lumberyard 1.26 or greater
- Visual Studio 2017 (version 15.9.27+) or 2019 (version 16.7.3+), including the Community Edition
- The **WelcomeGuideTutorials** game project.
- A basic understanding of the concepts used in developing 3D games

Important
We've provided a Lumberyard project containing the assets required to complete this tutorial series. You can download the project here: WelcomeGuideTutorials-v1.0.zip

Need some help setting up Lumberyard for the first time? Start here! (p. 14)

Or, you can watch a video on setup made specifically for this tutorial series:

Are you good? If so, let's go over the basic bits we provide you to get started.

Assets

We've provided a small project game containing assets for this tutorial series. Assets include meshes, textures, scripts, and complete levels for the end of each tutorial chapter.

The asset package WelcomeGuideTutorials-v1.0.zip contains the following directories (among others):

- \WelcomeGuideTutorials (root)
  - \Actors — Contains the `.fbx` and `.mtl` files for the "chicken" asset. The `.fbx` files contain meshes and a skeleton. The `.mtl` files are material files containing surface properties that define the look of the chicken.
  - \InputBindings — Contains the input configuration files for the tutorial game.
  - \Levels — Contains the data for each chapter's level configuration.
  - \ScriptCanvas — Contains the scripts you'll work with for each chapter.
  - \Slices — Contains the assembled slices (prefabs) for the chicken and other assets used in the tutorials.
  - \Textures — Contains the texture images used in the tutorials.

Open the ZIP file and place the folder and its contents under the \dev directory in your Amazon Lumberyard installation path. (For example, C:\Program Files\Amazon Lumberyard\dev.) We provided levels for each "chapter" of the larger tutorial series, so you can just pick a chapter you're interested in, load the *final* level for the preceding chapter, and start there. Or, you can start from chapter 1 and build each level on your own. Your time; your call!

About the tutorial game

In this series of tutorials, you'll develop a game we called "Flyin' the Coop". (You can name it whatever you like! Tech writers only *think* they're clever.) This is a simple game where you control a nameless, lonely chicken as they attempt to escape the coop that imprisons them. Across each chapter, you'll develop a little experience with the different features of Lumberyard as they related to specific game

development tasks, in an order that builds upon prior learning. Each chapter is relatively short and is designed to help you get over that first hurdle when working with a new tool: getting comfortable with the different features, tools, terms, and workflows.

Chapters

Get started with Amazon Lumberyard tutorial chapters

Chapter	Description
the section called “Levels” (p. 55)	Start here if you’re brand new to Lumberyard. Here, you’ll learn how to go from launching Lumberyard to creating a basic level.
the section called “Entities” (p. 59)	Learn how to create an entity, work with components, and sketch meshes with **White Box**.
the section called “Characters” (p. 64)	Learn how to import assets from `.fbx` files, and build a player character *slice*.
the section called “PhysX” (p. 70)	Learn how to add PhysX colliders to **White Box** entities. You’ll also learn to use the **PhysX Player Controller** to add a simple collider to a player entity, and use the **Input** and **Script Canvas** components to support player input.
the section called “Input and Scripting” (p. 74)	Learn how to create your own input events, and build a **Script Canvas** network to respond to input.
the section called “Cameras” (p. 79)	Learn how to create a simple 3rd person follow camera for the player character.
the section called “Terrain” (p. 83)	Learn how to create more detailed terrain in your level.
the section called “Slices” (p. 89)	Learn how to create props for you level with White Box meshes and *dynamic slices*.
the section called “Release” (p. 93)	Learn how to export your level and create a stand-alone package of your game.

Tutorial One: Create a level

In this first tutorial, you will create a simple level with a flat terrain in Lumberyard. At the beginning of each of the following tutorials, you will have the option to continue using the level you’ve been working on, or start fresh with a level that we created using the instructions up to that point. Each tutorial expects you to have successfully completed all the steps from the previous tutorials, so if you have any difficulty along the way, or wish to use one of our completed levels for any reason, you can locate them in the `Levels` directory of the **WelcomeGuideTutorials** project folder. The filename begins with a chapter number, and ends with the word `final`. And if you *do* have trouble completing any steps, we ask that you please leave us feedback using the **Feedback** link on this page, so we can continue to improve these tutorials.

Tip
If you like, you can follow this chapter in video (4:53 minutes) form:

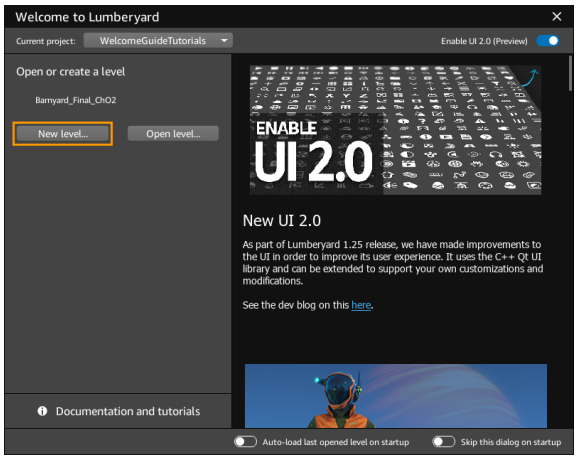
Before you proceed, you should already have completed the following steps:

- Downloaded and unzip the WelcomeGuideTutorials project into your Lumberyard dev folder.
- Set the **WelcomeGuideTutorials** project as your default project in **Project Configurator**.
- Configure the project using `lmbr_waf configure` from the command line in the `lumberyard\dev` directory.
- Build the project using a build command such as `lmbr_waf build_win_x64_vs2019_profile -p game_and_engine` from the command line in the `lumberyard\dev` directory. Depending on your hardware, the build may take a while to complete.

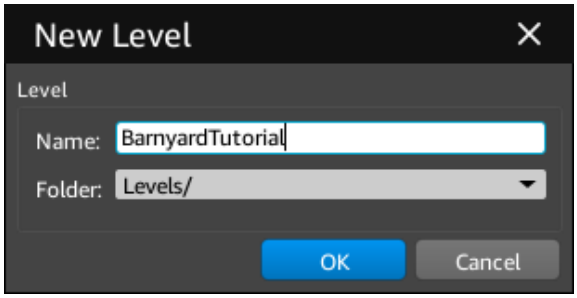
> **Note**
> See Building Lumberyard projects for information on command line configuration and build. Alternatively, you can configure and build the project from **Project Configurator**. See the section called "Project Configurator" (p. 18) for details on using **Project Configurator**.

When the basic project has been built, you are ready to continue. On to your first level!

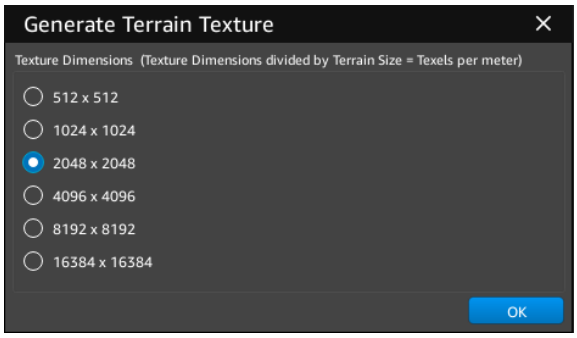
1. Open **Lumberyard Editor** by choosing the Lumberyard Editor icon from your desktop or Start Menu.
2. In the **Welcome to Lumberyard** dialog box, choose **New level...**.



3. In the **New Level** dialog, give your new level a name. Ensure the base **Levels/** directory is specified for the **Folder** option, and choose **OK**. This saves the new level in the existing **Levels/** directory.

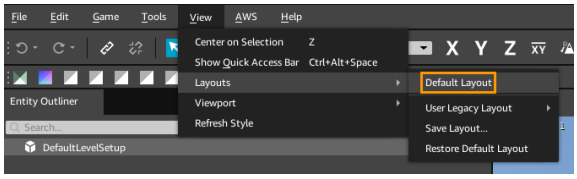


4. If the **Generate Terrain Texture** dialog appears, select **2048 x 2048** and choose **OK**. (If this dialog does not appear, skip to the next step.)

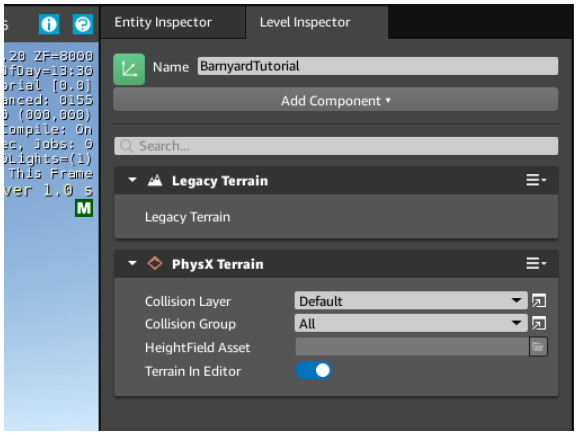


Lumberyard Editor displays your new level in the **Perspective** pane with a default sky, an infinite ocean, and a flat terrain plane with a default grid texture.

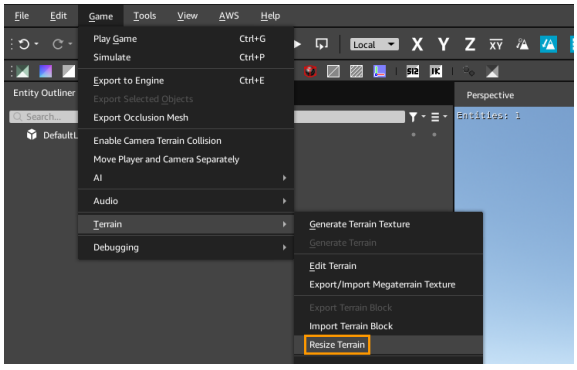
5. This tutorial is written for the default **Lumberyard Editor** layout, so make sure this is the layout that you're using. To set the layout, access the menu bar and select **View**, **Layouts**, and choose **Default Layout**.



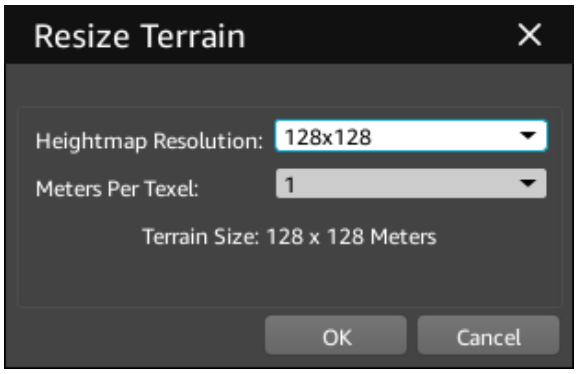
6. This tutorial uses **NVIDIA PhysX** to simulate physics interactions. To enable collision with the terrain, add a **PhysX Terrain** level component. Choose the **Level Inspector** tab in the upper right of the editor interface, choose **Add Component**, and select **PhysX Terrain** from the list of level components.



7. The terrain is quite large. You won't need all this space. You can resize the terrain to make it a bit more manageable. From the **Main Menu** bar, select **Game**, **Terrain** and choose **Resize Terrain**.



8. Complete the fields in the **Resize Terrain** dialog. For **Heightmap Resolution**, select **128 x 128**. For **Meters Per Texel**, select **1** and choose **OK**.

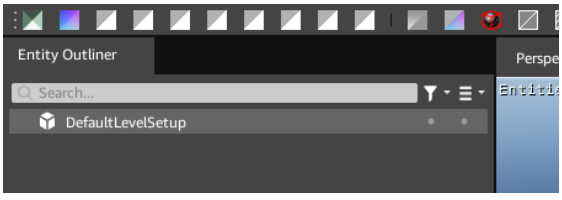


9. Because the terrain has been resized, the terrain texture must be regenerated. The **Generate Terrain Texture** dialog automatically opens after the previous step. Select **2048 x 2048** and choose **OK**.

10. The resized terrain might have disappeared in the **Perspective** pane. You can use the camera controls to find the terrain and bring it back into view. You can hold the right mouse button and move the mouse to aim the camera, and use the W, A, S, and D keys to fly forward, left, back and right, respectively. You might want to set the camera speed to `10.0` for this part. You can change the camera speed in the lower right of **Perspective**, as shown in the following animated GIF. If you're having trouble moving the camera with the **W**, **A**, **S**, and **D** keys, try clicking anywhere in the viewport, which will deselect any entity you have selected. For more information, see the the section called "Lumberyard Editor tour" (p. 29).

Another method to set your perspective is to focus on a selected entity. Because the **DefaultLevelSetup** entity was set up before you resized the terrain, you must move the entity to center it in the new terrain size. Use the following steps to move the entity.

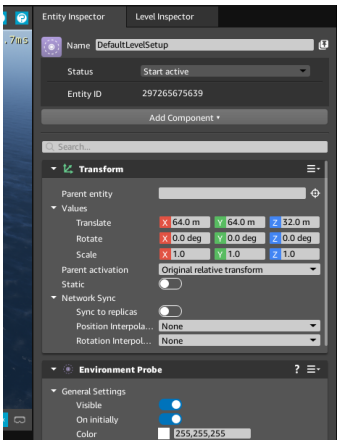
a. In the **Entity Outliner** on the left of the editor, click the **DefaultLevelSetup** entity to select it.



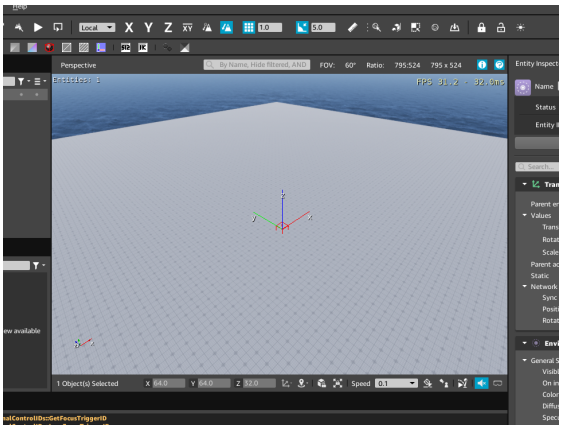
b. Choose the **Entity Inspector** tab on the right of the editor to view the components of the **DefaultLevelSetup** entity.

c. The **DefaultLevelSetup** entity contains two components, an **Environment Probe**, and a **Transform**. Every entity has a **Transform** component that defines the position, rotation and scale of the entity in the level. You can use the **Translate** property of the **Transform** component to center the entity on the resized terrain.

The terrain is a 128 x 128 meter plane. The lower left corner of the terrain is positioned at `0.0` in the X and Y axes in world space, `32.0` meters above the infinite ocean surface (Z axis). In the the **Transform** component, set the **Translate** X and Y properties to `64.0` (half the size of the terrain), and set the **Translate** Z property to `32.0`. This places the **DefaultLevelSetup** entity at the center of the terrain.



d. With the **DefaultLevelSetup** entity still selected, press Z to focus the **Perspective** pane on the **DefaultLevelSetup** entity and the surrounding terrain. You can also use the camera controls that were explained previously to make fine adjustments to the camera position.



11. When you have the perspective view set up to your liking, you can bookmark the location by pressing **Control** + any **Function** key. Press **Control + F1** to bookmark the view. You can return to the view at any time by pressing **Shift + F1**.
12. Save the level. Press **Control + S** or select **Save** from the **File** menu.

When you're ready, try the section called "Entities" (p. 59).

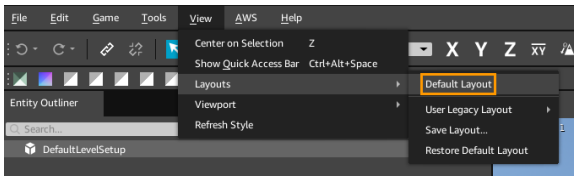
Tutorial Two: Create an entity with White Box

In this tutorial, you will create a simple environment prop using White Box, a component tool that you can use to quickly sketch 3D models in Lumberyard. You will also learn the basics of the component entity system, including creating entities and adding components.

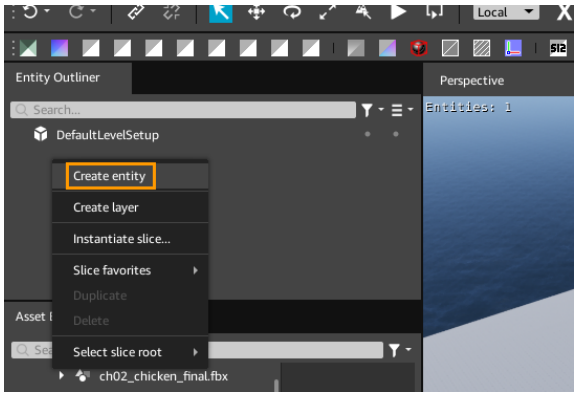
Tip
If you like, you can follow this chapter in video (4:43 minutes) form:

Begin this tutorial either with the level you created in the section called “Levels” (p. 55), or by opening `ch01_barnyard_final` from the `Levels` directory of the **WelcomeGuideTutorials** project. To open a level in Lumberyard, choose **Open Level...** from the **File** menu in the main menu bar, or press **Control + O**.

1. This tutorial is written for the default **Lumberyard Editor** layout, so make sure this is the layout that you're using. To set the layout, access the menu bar and select **View**, **Layouts**, and choose **Default Layout**.

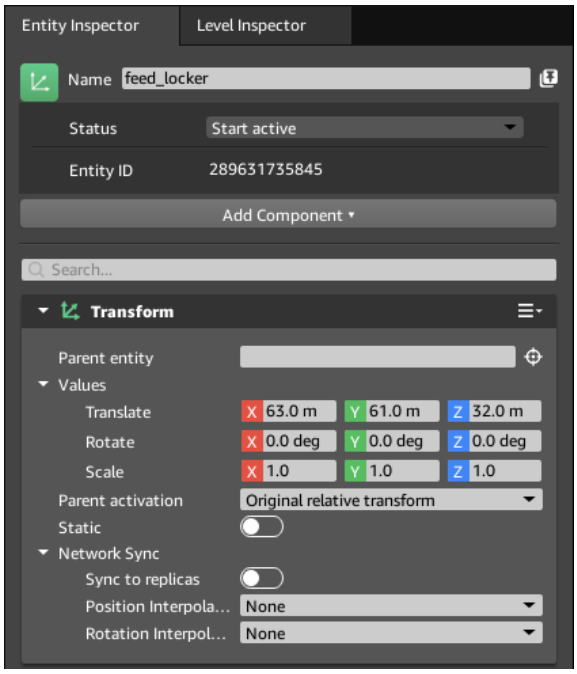


2. Create an entity by right-clicking in **Entity Outliner** on the left side of the editor and choose **Create entity** from the context menu. Click on the newly created entity to select it.



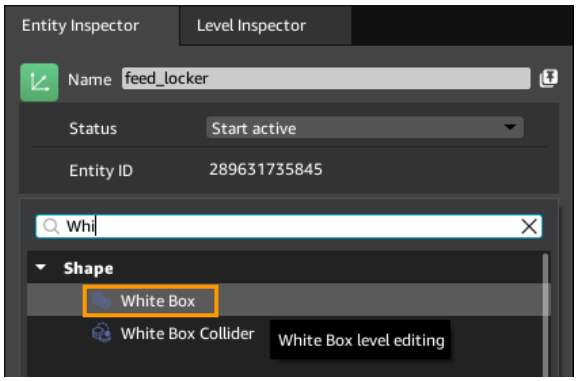
3. With the entity selected, review the **Entity Inspector** on the right of the editor. You can see the entity has a **Transform** component, a default **Name**, and a unique **Entity ID**. All entities have a **Transform** component that defines the position, orientation, and scale of the entity in the level. Also notice the transform *gizmo* (the three-axis object labeled X, Y, and Z) in the **Perspective** view. If you don't see the gizmo, with the entity selected press **Z** to focus on it. The gizmo is the representation of your new entity's transform in the level.

   Change the name of the entity. In **Entity Inspector**, click in the **Name** property and enter `feed_locker` as the name for the entity.

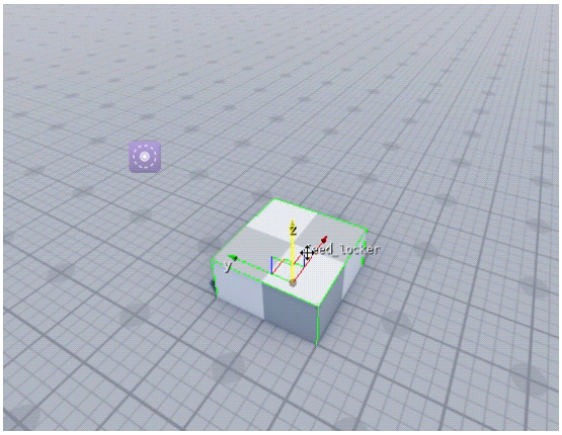


4. Entities are collections of components that define a *thing* for your project. An entity can be a character, a prop for the environment, a visual effect, and more. Components add functionality to an entity such as a renderable mesh, a PhysX collider, or a script that responds to user input (for example). In the following steps use the White Box component tool to quickly sketch out the prop. This way, you can further develop your project before investing considerable time and resources in building the final asset.

   With the **feed_locker** selected, in **Entity Inspector**, choose **Add Component**. Begin typing `whi` in the search field, and select **White Box** from the filtered list.

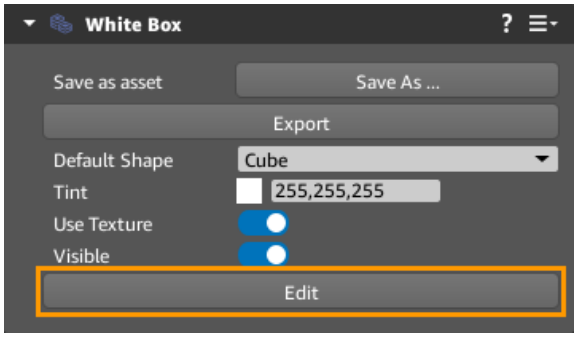


5. The entity now has a cube with a checkerboard texture, but it might be partly beneath the terrain.
   a. To move the cube above the terrain, with the **feed_locker** entity selected, press the **2** key. You can also choose the **Move** icon in the top toolbar to enable the move tool.
   b. Hover your mouse cursor above the Z axis of the transform gizmo in **Perspective** view (the axis turns yellow), hold the left mouse button, and drag up until the box is above the terrain.



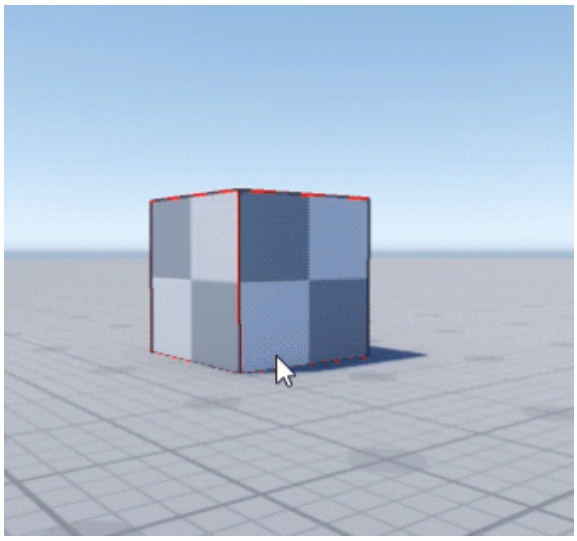
Note
If **Snap to grid** is enabled, the feed locker entity will automatically snap to a position above the terrain. To freely position the entity, press **G** or click the **Snap to grid** icon in the top menu bar to disable **Snap to grid**.

6. You'll use a few modeling operations to modify the White Box cube into the rough shape of a feed locker. To modify the White Box, you must be in White Box edit mode. Choose **Edit** in the White Box component to enter edit mode.

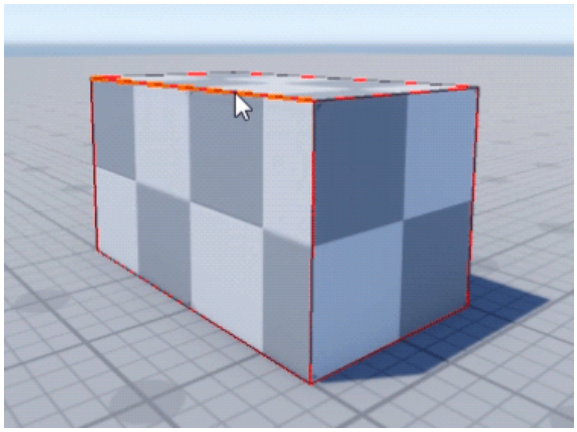


Note
A complete list of White Box edit mode operations can be found in the White Box component documentation.

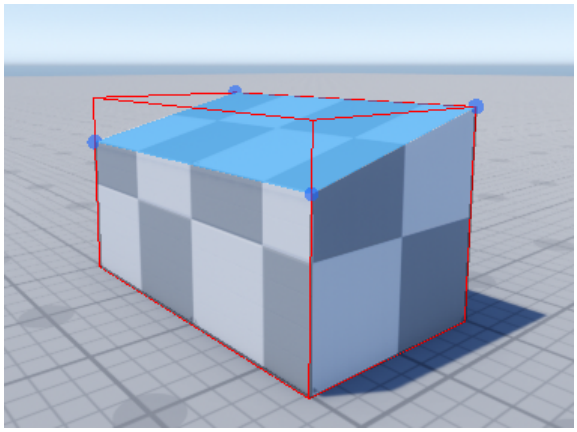
7. Hover over a side of the box. Hold the **left mouse button** and drag to pull the side of the box, increasing its width.



8. Hover over the top front edge of the box. Hold the **left mouse button** and drag to pull the edge down a bit, creating a slope on the top of the box.

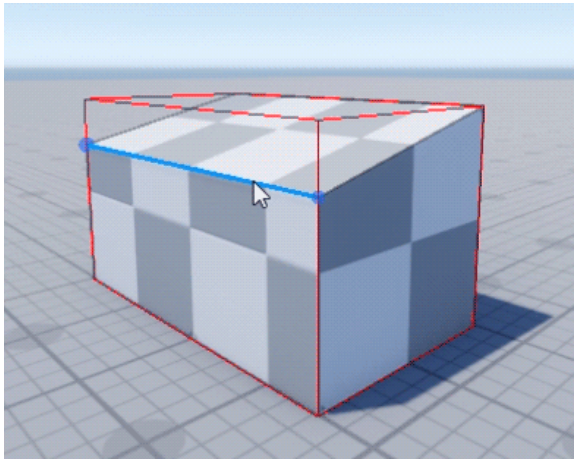


9. Next you'll perform a bit more complex of an operation to create the lid for the locker. Start by clicking the top of the box to select the top face. The face highlights blue and the vertices of the face are displayed.

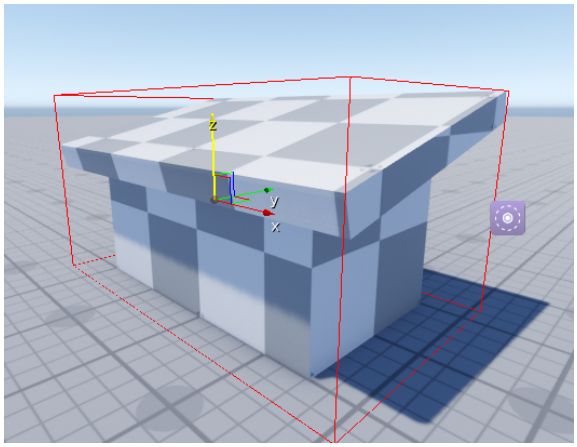


10. Perform an extrude scale operation. Hold the **Control** key, hover over one of the vertices of the selected face, and hold the **left mouse button** as you drag the mouse outward from the box.

11. Perform an extrude operation. Hold the **Control** key, hover over the selected face, and hold the **left mouse button** as you drag the mouse upward from the box.



12. Choose **Done** in the White Box component to exit White Box edit mode. The completed feed_locker White Box model should resemble the image below.



13. Save the level. Press **Control + S** or select **Save** from the **File** menu.

For extra credit, try using the steps above to create an new entity, add a White Box component, and use some of the additional editing features of the White Box component to create a more complex environment asset.

When you're ready, try the section called "Characters" (p. 64).

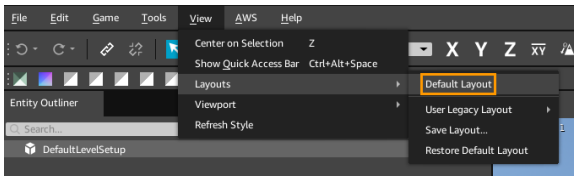
Tutorial Three: Build a player character

In this tutorial, you will build a player character from various components and create a slice for your player character. You will also learn about Lumberyard's asset pipeline.

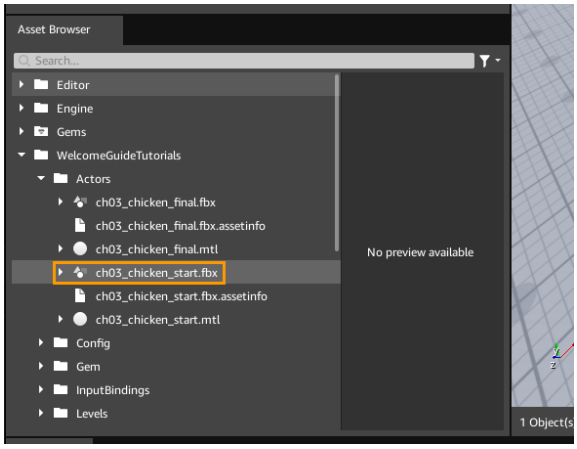
Tip
If you like, you can follow this chapter in video (2:48 minutes) form:

Begin this tutorial either with the level you created in the section called "Entities" (p. 59), or by opening `ch02_barnyard_final` from the `Levels` directory of the **WelcomeGuideTutorials** project. To open a level in Lumberyard, choose **Open Level...** from the **File** menu in the main menu bar.

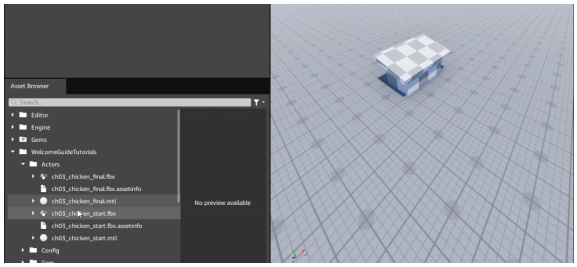
1. This tutorial is written for the default **Lumberyard Editor** layout, so make sure this is the layout that you're using. To set the layout, access the menu bar and select **View**, **Layouts**, and choose **Default Layout**.



2. Create a player character entity. In **Asset Browser** on the left side of the editor, find `ch03_chicken_start.fbx` in the **Actors** directory. This asset was created in a third-party application and saved as a `.fbx` file for import into Lumberyard.



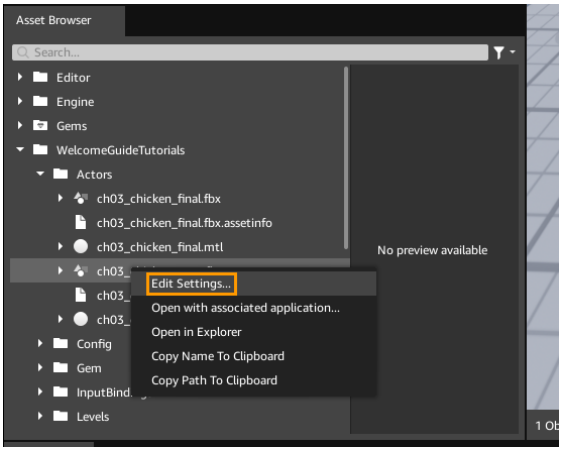
3. Click and drag `ch03_chicken_start.fbx` into the **Perspective** pane. Drop it somewhere in front of the **feed_locker** entity. This creates an entity for the chicken player character from the `ch03_chicken_start.actor` asset.



4. Press the **Z** key to focus on the chicken entity, or use the mouse and keyboard navigation controls to get a good view of the chicken asset.
5. Modify the FBX Settings for the chicken asset to import her mohawk as a cloth object. Models, rigged actors, and animations created in third-party applications are imported into Lumberyard through `.fbx` files. The chicken `.fbx` file contains a mesh asset for the chicken's mohawk that has not been imported. You can add the mohawk to the chicken's **Actor group** in **FBX Settings** to import it, and add a modifier that will add cloth data to the mohawk so that it behaves like a cloth object at run-time.

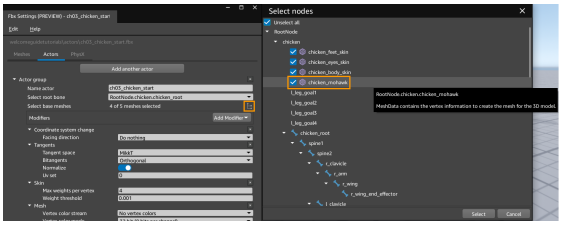
   a. Right-click `ch03_chicken_start.fbx` in **Asset Browser** and choose **Edit Settings** to view the **FBX Settings** for the chicken.

      FBX Settings should open to the **Actors** tab displaying one **Actor group**. This group will output one actor asset from the `.fbx`. The `.actor` asset contains meshes and a skeleton that are selected in the actor group. The actor group is mostly set up. You must make a couple of edits to import the chicken's mohawk as a cloth object.

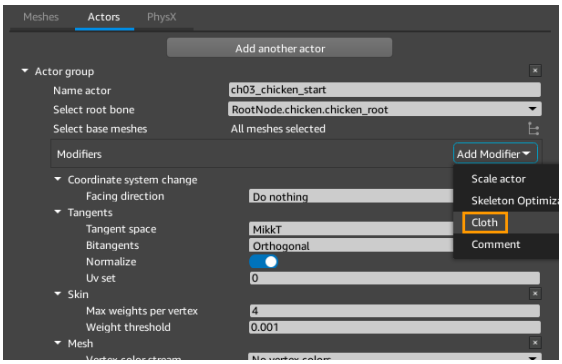


b. In the `ch03_chicken_start` actor group, choose the **Hierarchy** button to the right of **Select base meshes**.

c. To import the mohawk mesh, in the **Select nodes** window, choose **chicken_mohawk** from the list to add the mohawk mesh to the mesh group. Choose the **Select** button to close the window.

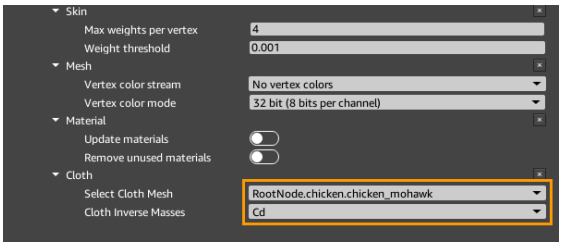


d. To add cloth data to the mohawk, in **FBX Settings** choose **Add Modifier** and select **Cloth** from the modifier list.



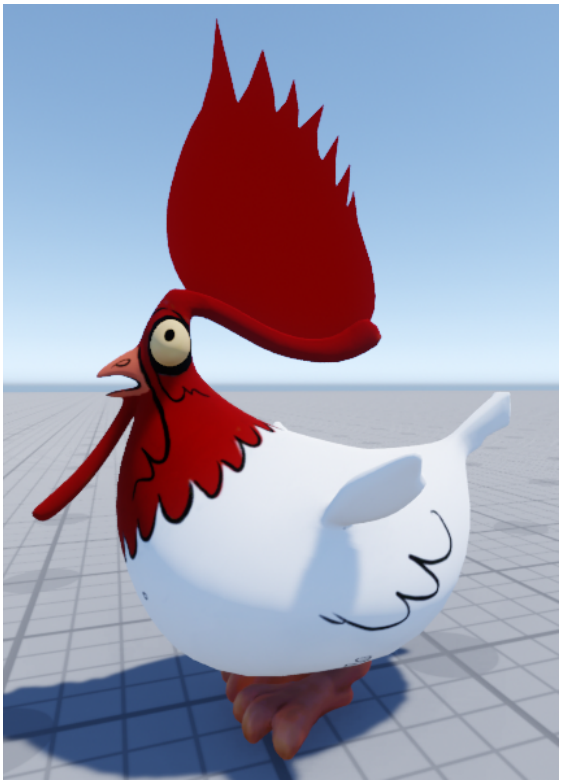
e. In the cloth modifier, for **Select Cloth Mesh**, choose `RootNode.chicken.chicken_mohawk` to add the chicken's mohawk as a cloth object.

f. In the cloth modifier, for **Cloth Inverse Masses** choose `Cd` from the list to use chicken_mohawk's vertex color stream to set the mass values for the cloth object. The vertex colors applied to the chicken's mohawk were pre-painted in a 3rd-party application. Mass will be generated per vertex on the mohawk based on the red value contained in the `Cd` vertex color stream. for more information on Cloth, see Cloth Simulation.



g. Choose **Update** at the bottom of the **FBX Settings** window.

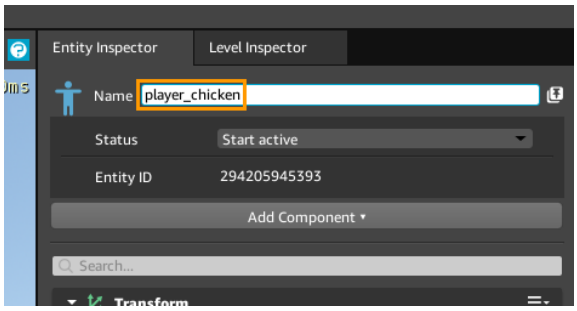
h. In the **Progress** window, choose **Ok**. The window closes and the `.fbx` file is quickly processed with the new settings. The mohawk mesh appears in **Perspective** on the chicken entity.



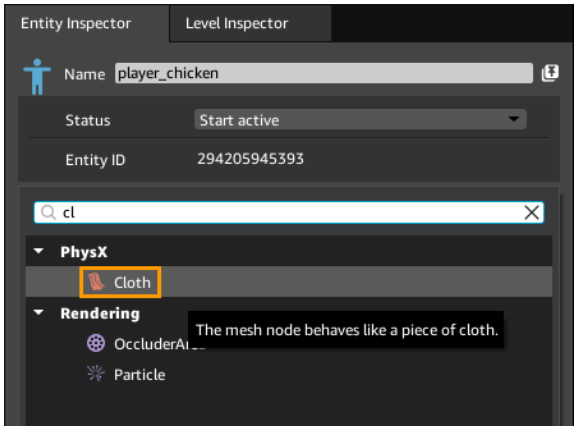
i. Close the **FBX Settings** window.

> **Note**
> When you close the **FBX Settings** window, you might see a window stating there are unsaved changes even though the changes have been automatically saved. This is a known issue. Choose **Discard** to close the **FBX Settings** tool.

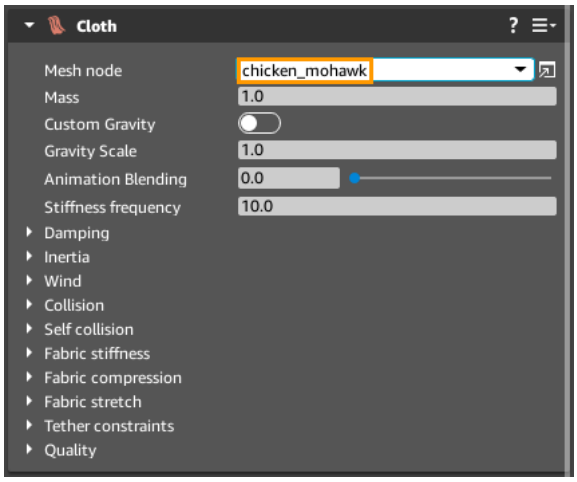
6. In **Entity Outliner**, click the `ch03_chicken_start` entity to select it.
7. In **Entity Inspector** on the right of the editor, rename the entity. Click in the field next to **Name** and enter `player_chicken`.



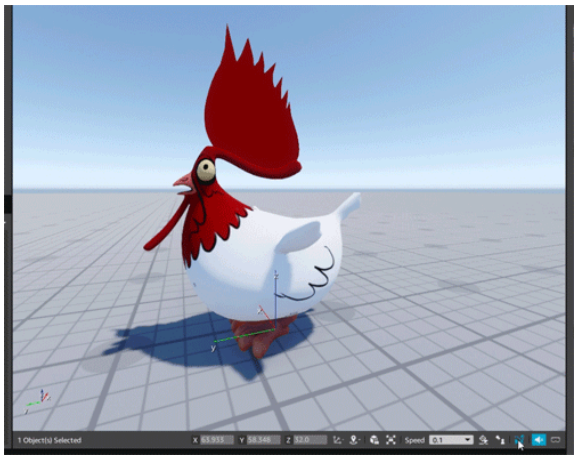
8. The mohawk has been imported and has cloth data applied, but to get the mohawk to simulate as a cloth object at run-time, you need to add a **Cloth** component to the `player_chicken` entity. In **Entity Inspector**, choose **Add Component** to open the component list. Enter `cl` in the search field to filter the list by name and choose **Cloth** from the filtered results to add a **Cloth** component.



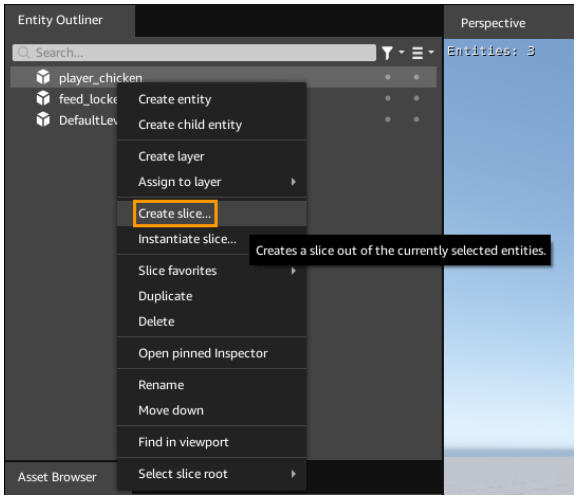
9. In the **Cloth** component that has been added to the entity, for **Mesh node**, choose `chicken_mohawk` from the list.



10. Click the **Simulate** button in the bottom toolbar of the **Perspective** pane, or press **Control + P** to test the cloth simulation. The chicken's mohawk should flop over like cloth. Press **Control + P** to exit simulation mode.



11. Create a slice from the `player_chicken` entity. Right-click the **player_chicken** entity in **Entity Outliner**, and choose **Create slice...**. Save the entity as a `.slice` in the `Slices` directory as `player_chicken.slice`. Slices are reusable and can be instanced, each instance having unique property settings. You can create complex slices using collections of simple entities.



For extra credit, play with the settings in the **Cloth** component and test the simulation in **Perspective** to fine-tune the mohawk behavior. See the Cloth component documentation for more info. Save your changes to the `player_chicken.slice`.

The cloth simulation requires collision primitives to prevent the mohawk cloth mesh from penetrating the chicken geometry when simulated. You can see how these colliders are set up in **Animation Editor**.

1. Open **Animation Editor** from the **Tools** menu.
2. Open the chicken actor from the **File** menu in **Animation Editor**.
3. Enable the **Physics** layout from the **Layouts** menu in **Animation Editor**.

You will see spheres and capsules attached to the chicken's skeleton. These are the cloth colliders. You can locate their nodes in the **Skeleton Outliner** by the shirt icon to the right of the node name. Like the **Cloth** modifier you added in **FBX Settings**, these colliders and their properties are saved in the `.assetinfo` file so that Asset Processor can include them in the runtime asset.

When you're ready, try the section called "PhysX" (p. 70).

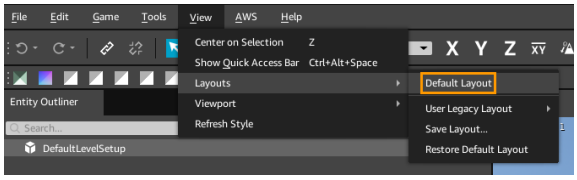
Tutorial Four: Create PhysX colliders

In this tutorial, you add a **White Box Collider** to the feed locker and a **Character Controller** to the chicken to detect collisions with PhysX. In the first tutorial, you added a **PhysX Terrain** level component which creates a collider for terrain. The chicken and the feed locker colliders, in combination with the **PhysX Terrain** level component, allow the chicken to move and jump around the level without falling through the terrain or the feed locker.

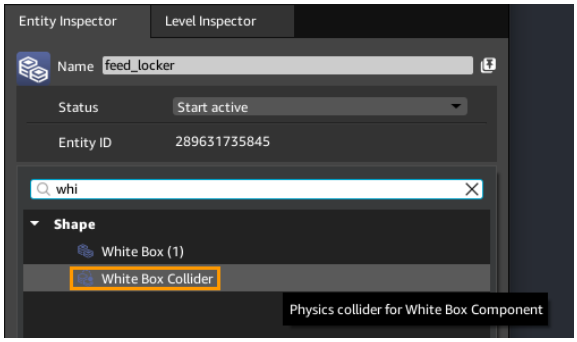
Tip
If you like, you can follow this chapter in video (4:28 minutes) form:

Begin this tutorial either with the level you created in the section called "Characters" (p. 64), or by opening `ch03_barnyard_final` from the `Levels` directory of the **WelcomeGuideTutorials** project. To open a level in Lumberyard, choose **Open Level...** from the **File** menu in the main menu bar.

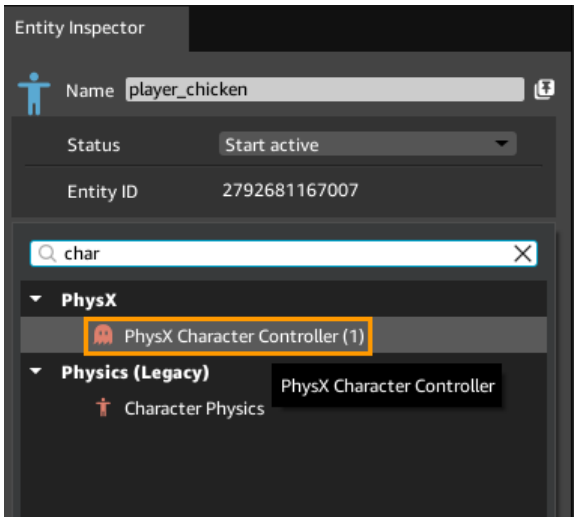
1. This tutorial is written for the default **Lumberyard Editor** layout, so make sure this is the layout that you're using. To set the layout, access the menu bar and select **View**, **Layouts**, and choose **Default Layout**.



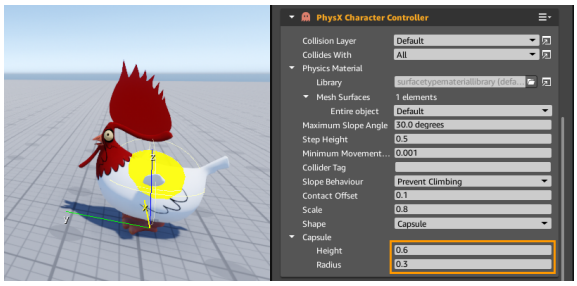
2. Add a collider to the the feed locker. Select the `feed_locker` entity either by clicking its name in **Entity Outliner**, or by clicking on it in **Perspective**. In **Entity Inspector**, click **Add Component** to open the component list. Enter `whi` in the search field to filter the list by name and choose **White Box Collider** from the filtered results to add a **White Box Collider** component. For this tutorial the default property settings on the component are fine.



3. Add a collider to the chicken. Select the chicken slice by clicking its name in **Entity Outliner** or by clicking on it in **Perspective**. In **Entity Inspector**, click **Add Component** to open the component list. Enter `char` in the search field to filter the list by name and choose **PhysX Character Controller** from the filtered results to add a **PhysX Character Controller** component.

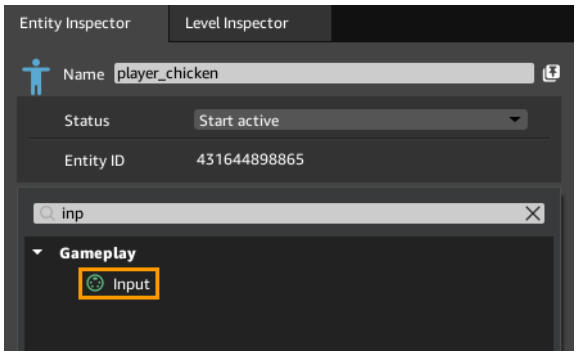


4. The default collider shape for the **PhysX Character Controller** is a capsule. Change its Height to `0.6` and its **Radius** to `0.3`, so that the collider more closely matches the shape of the chicken's body.



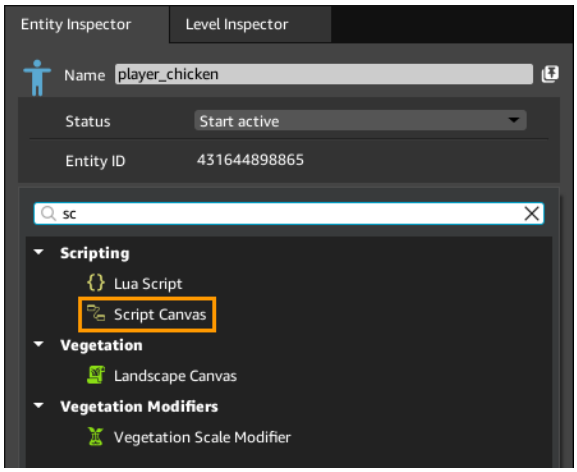
5. To test the colliders, you need to be able to move the chicken. To do this, you add input bindings that link input from keys and buttons to events in Lumberyard. You also add a script canvas that handles the input events. The next chapter will go into detail about input bindings and **Script Canvas**. For now, add an already created set of input bindings and a script canvas to provide basic movement for the chicken.

   With **player_chicken** selected, in **Entity Inspector**, choose **Add Component** to open the component list. Enter `inp` in the search field to filter the list by name and choose **Input** from the filtered results to add an **Input** component.

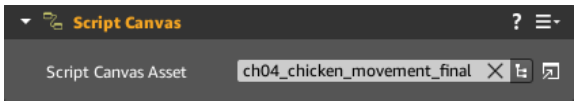


6. In the **Input** component, for **Input to event bindings**, choose the **Hierarchy** button to the right. Expand the `InputBindings` directory and open `ch04_chicken_controls_final.inputbindings`. The `.inputbindings` file is an asset created in Lumberyard that creates event groups with event generators. A button on a gamepad is an event generator and a `.inputbindings` asset links the button to a named event you can handle through script.

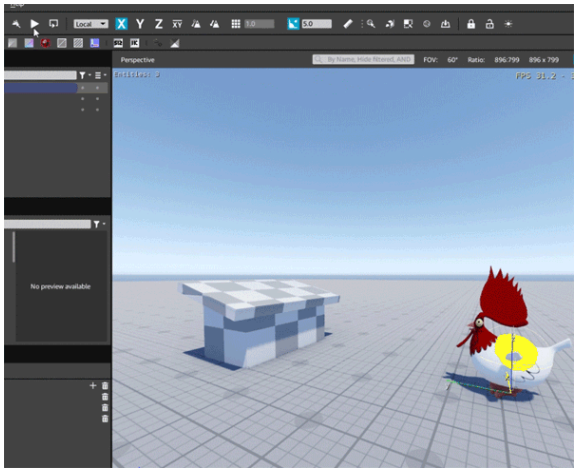
7. With **player_chicken** selected, in **Entity Inspector**, choose **Add Component** to open the component list. Enter `sc` in the search field to filter the list by name and choose **Script Canvas** from the filtered results to add a **Script Canvas** component.



8. In the **Script Canvas** component, for **Script Canvas Asset**, choose the **Hierarchy** button to the right. Expand the `ScriptCanvas` directory and open `ch04_chicken_movement_final`. The `ch04_chicken_movement_final.scriptcanvas` asset contains a few script networks that respond to events generated from the `ch04_chicken_controls_final.inputbindings` asset, making the chicken move in response.



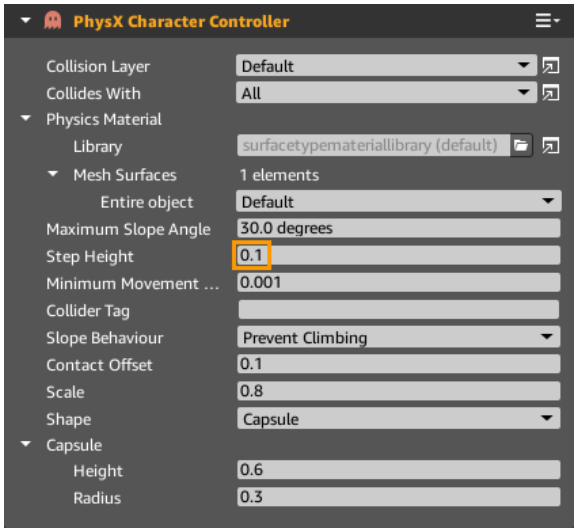
9. In the top toolbar, click the **Play** button or press **Control + G** to enter play mode. Use the **W**, **A**, **S**, and **D** keys or, preferably, the **Left thumbstick** on a game controller to move the chicken. The **Space** key, or the **Right bumper** button on the controller makes the chicken jump. Press **Escape** to exit play mode.



10. You might have noticed that when the chicken collides with the feed box, she will pop up on top of the feed box if you continue to press forward on the thumbstick. This is a feature of the **PhysX**

Character Controller. The **PhysX Character Controller** has a **Step Height** property that allows the character to walk up objects smaller than a set height. The property is set too high by default for this scenario.

With **player_chicken** selected, in **Entity Inspector**, in the **PhysX Character Controller** component, set **Step Height** to `0.1`. Test the chicken input again. The chicken no longer pops up on the feedbox after colliding with it.



11 Save the changes to the `player_chicken` slice. Right-click the player_chicken slice in Entity Outliner to open the context menu, and choose `player_chicken.slice` in the **Save Slice Overrides** list to save the changes you made to `player_chicken`.

For extra credit, open Script Canvas to view the `ch04_chicken_movement_final.scriptcanvas` asset.

1. From **Tools** in the main menu, choose **Script Canvas** to open the Script Canvas editor.

   In **Script Canvas**, expand the **File** menu and choose **Open** or press **Control + O**. In the `ScriptCanvas` directory, open `ch04_chicken_movement_final.scriptcanvas`. This script canvas creates event notifications for the events generated by `ch04_chicken_controls_final.inputbindings`, and then moves the chicken entity in response to the events. There are comments and groups in the canvas that explain what various parts of the graph do.

2. Notice the **Variable Manager** in the upper right of the **Script Canvas** window. The variables control various aspects of the chicken's movement. The variables that are intialized to some value other than `0.0` set limits on properties such as how fast the chicken can move, how high it can jump, and how long it hangs in the air at the apex of its jump. Try changing one of the variables, such as **jumpMaxHeight**, and test the changes to see the effect.

   Tip
   Save your changes to a new script canvas. Assign the new script canvas to the chicken's **Script Canvas** component to preserve the original `ch04_chicken_movement_final.scriptcanvas`.

When you're ready, try the section called "Input and Scripting" (p. 74).

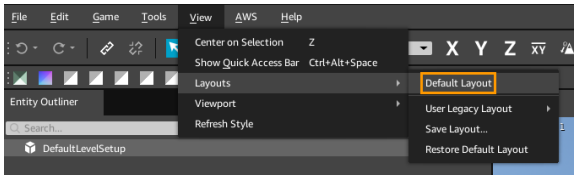
Tutorial Five: Handling player input through Script Canvas

In this tutorial, you will add new input event generators and build a network in **Script Canvas** to turn the chicken when the user presses the right thumbstick or the **L** and **;** keys. This tutorial goes into detail about handling input events, working with variables, and finding, placing, and connecting nodes in **Script Canvas**.

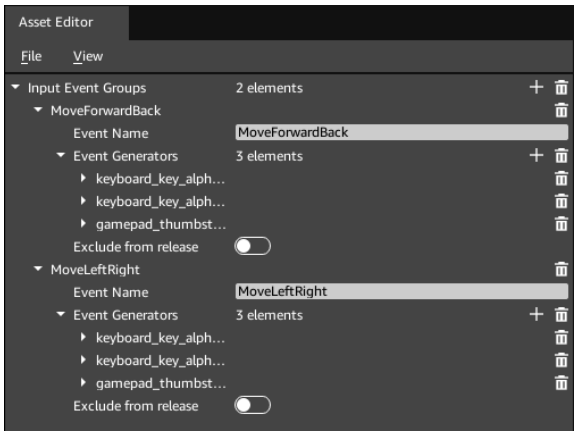
> **Tip**
> If you like, you can follow this chapter in video (7:33 minutes) form:

Begin this tutorial either with the level you created in the section called "PhysX" (p. 70), or by opening `ch04_barnyard_final` from the `Levels` directory of the **WelcomeGuideTutorials** project. To open a level in Lumberyard, choose **Open Level...** from the **File** menu in the main menu bar.

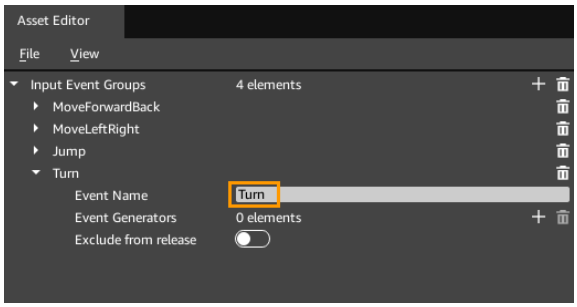
1. This tutorial is written for the default **Lumberyard Editor** layout, so make sure this is the layout that you're using. To set the layout, access the menu bar and select **View**, **Layouts**, and choose **Default Layout**.



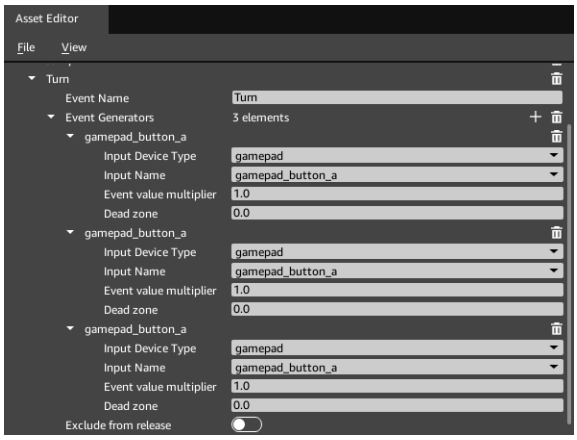
2. Edit the `.inputbindings` asset to create a new event group and event generators for turn input. From the **Tools** menu, choose **Asset Editor**.
3. Locate the **Asset Editor** tab on the left below **Asset Browser**. In **Asset Editor**, expand the **File** menu and choose **Open**. In the **InputBindings** folder, open `ch05_chicken_controls.inputbindings`. The `.inputbindings` file binds user input to events in Lumberyard that you can handle through script. This `.inputbindings` asset has been set up with three events: forward and back movement, left and right movement, and jump. These events are bound to keyboard and gamepad input.



4. In **Asset Editor**, in the `ch05_chicken_controls.inputbindings` asset, click the **+** button to the right of **Input Event Groups** to create a new input event group. Expand the group by clicking the arrow in the left. Enter `Turn` for **Event Name** in the new group.



5. Add **three** event generators. Event generators are mapped to the keys and buttons on your input devices. Click the **+** button to the right of **Event Generators** in the **Turn** input event group to create an new event generator. In the **Class to create** dialog for each new generator, accept the default choice **Input**.

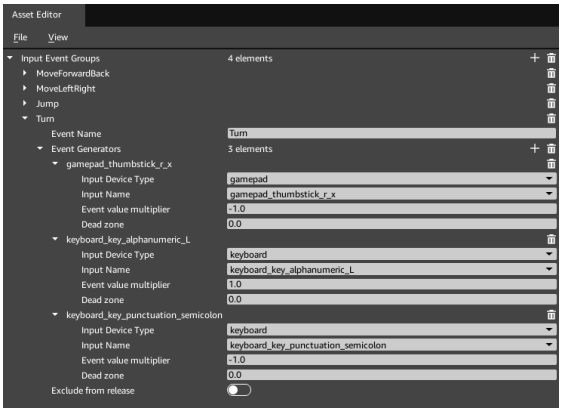


6. The following steps assign the first input generator to the right thumbstick, and the second and third generators to the **L** and **;** keys so the player can turn the chicken left and right.

   a. In the first input generator, set **Input Device Type** to `gamepad`, set **Input Name** to `gamepad_thumbstick_r_x`, and set **Event value multiplier** to `-1.0`. Setting **Event value multiplier** to `-1.0` will invert the output from the left to right axis of the right thumbstick. This is necessary to rotate the chicken to the right when the thumbstick is pressed right, and left when the thumbstick is pressed left.

   b. In the second input generator, set **Input Device Type** to `keyboard`, set **Input Name** to `keyboard_key_alphanumeric_L`, and set **Event value multiplier** to `1.0`.

   c. In the third input generator, set **Input Device Type** to `keyboard`, set **Input Name** to `keyboard_key_punctuation_semicolon`, and set **Event value multiplier** to `-1.0`.



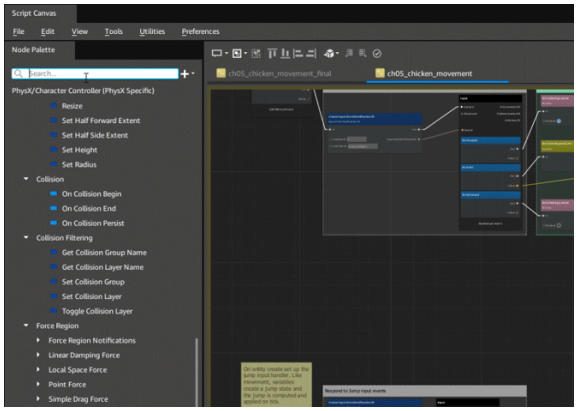
7. In **Asset Editor** expand the **File** menu and choose **Save** to save the new input bindings.

Important
The chicken entity is set to use a different `.inputbindings` than the one you created. Make sure to update the **Input** component on `player_chicken` to use the `ch05_chicken_controls.inputbindings` asset you edited.

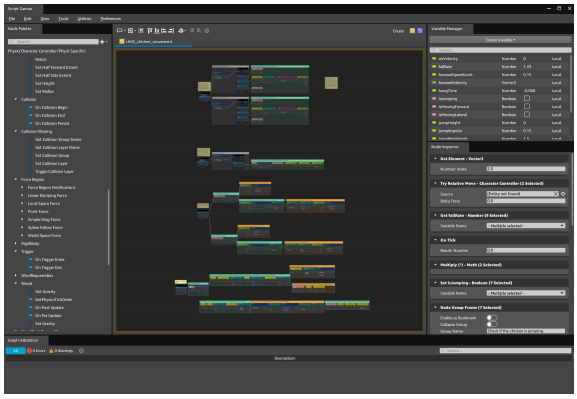
8. You must create a script to handle the input events the `Turn` input bindings will generate. From **Tools** in the main menu, choose **Script Canvas** to open the Script Canvas editor.

   Script Canvas is a visual script editor. In Script Canvas, rather than write code, you create scripts visually by placing and connecting nodes. The nodes in **Script Canvas** are functions that use the same framework as Lua and the Lumberyard C++ API. You can create complex functionality in Script Canvas without knowing how to program.

   On the left of **Script Canvas** is the **Node Pallette**. You can filter the node list by typing terms in the search bar at the top of the pallette. To place a node, click and drag node names from the pallette into the graph. Hold the right mouse button and drag the mouse to pan the graph. Scroll the mouse wheel to zoom the graph. Take a moment to examine the contents of the network and try navigating the graph.

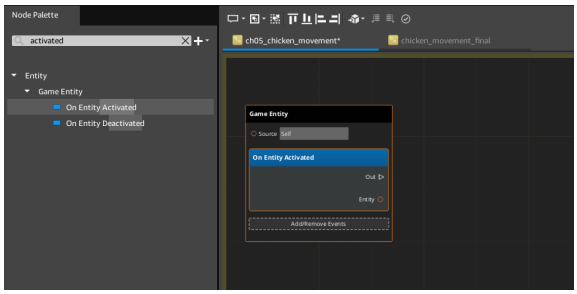


9. In **Script Canvas**, expand the **File** menu and choose **Open** or press **Control + O**. In the `ScriptCanvas` directory, open `ch05_chicken_movement.scriptcanvas`. This script canvas creates event notifications for the input events generated by `ch05_chicken_controls.inputbindings`, and then moves the `player_chicken` entity based on those input events.

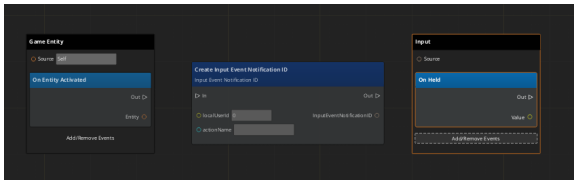


10. To create a network that handles the `Turn` input event, begin by panning and zooming the graph to find an empty spot to build a new node network.
11. Type `activated` in the search bar at the top of the **Node Pallette** to filter the list. Click and drag the **On Entity Activated** node into the graph. Every entity can generate an activated event. In this scenario, the chicken entity generates an activated event when gameplay starts. The network connected to this node will execute when the chicken is activated.

Note the **On Entity Activated** node is in a container called **Game Entity**. This is because game entities can also be deactivated. In some circumstances you will want access to both events in the same network.



12. Using the same method as above, place a **Create Input Event Notification ID** node to the right of the **On Entity Activated** node in the graph.
13. Place an **On Held** node to the right of **Create Input Event Notification ID** in the graph. The **On Held** node is in an **Input** container. The **Input** container may also have on pressed and on released events in the same network.

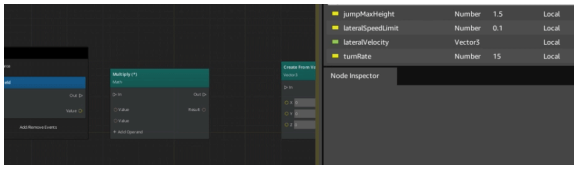


14. Click the **Create Input Event Notification ID** node to select it. Set the **actionName** property to `Turn` either by typing in the field on the node, or in the **Node Inspector** to the right. This sets the script to respond to the `Turn` event you created in the input bindings.
15. Click the **Input** node to select it. In the node inspector to the right, enable the **Display Connection Controls** property. This exposes some input connection pins on the **Input** node, and enables the node to be wired into a network. Most often, event nodes like **On Held** begin a network, so their input connections pins are hidden by default.
16. Wire up the following pins by clicking and dragging from the *first pin* to the *second pin*:
    a. From **On Entity Activated** *Out* to **Create Input Event Notification ID** *In*
    b. From **Create Input Event Notification ID** *Out* to **Input** *Connect*
    c. From **Create Input Event Notification ID** *InputEventNotificationID* to **Input** *Source*

    These three nodes are the basis for handling player input. The network that connects to these nodes will be executed every time the player presses the right thumbstick, or the **L** or **;** keys.
17. The functionality to rotate the chicken entity requires three nodes. Add a **Multiply ()** node to the right of the **On Held** node.
18. Add a (Vector3) **Create From Values** node to the right of the **Multiply ()** node.
19. Add a **Rotate** node to the right of the **Create From Values** node.
20. The value **On Held** provides must be multiplied by a value that controls how fast the chicken rotates when the right thumbstick is held. The **turnRate** variable controls rotation speed. You can add the

turnRate variable to the **Multiply ()** node as a reference using drag and drop. Locate the **turnRate** variable in the **Variable Manager** to the right.

21. Click and drag the **turnRate** variable into the graph and drop it on the second **Number** pin of the **Multiply ()** node to reference the **turnRate** variable.



22. Wire up the following pins by clicking and dragging from the *first pin* to the *second pin*:

    a. From **On Held** *Out* to **Multiply ()** *In*

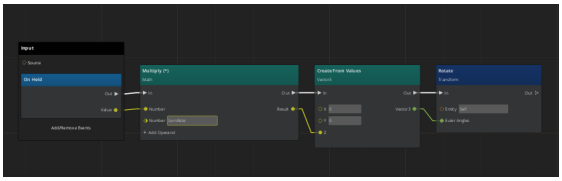
    b. From **On Held** *Value* to **Multiply** *Number*(1)

    c. From **Multiply** *Out* to **Create From Values** *In*

    d. From **Multiply** *Result* to **Create From Values** *Z*

    e. From **Create From Values** *Out* to **Rotate** *In*

    f. From **Create From Values** *Vector3* to **Rotate** *Euler Angles*

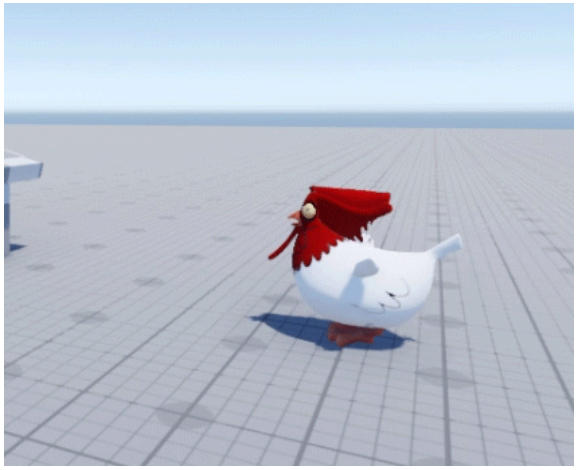


    The **On Held** node provides input values when the user holds a button or key on a device. In this scenario, the right thumbstick is an analog control and provides a value between `-1.0` and `1.0` depending how far left or right it is pressed. The value from the input is multiplied by the turn rate. The result will determine how quickly the chicken turns when the user holds the right thumbstick left or right. The result is plugged into the Z element of a Vector3, Z being the axis we want to rotate the chicken around. The resulting Vector3 is used by the **Rotate** node to apply a rotation to the chicken entity.

23. Save the `ch05_chicken_movement` Script Canvas by pressing **Control + S** or choosing **Save** from the **File** menu in **Script Canvas**.

    Important
    The chicken entity's **Script Canvas** component is not referencing the `ch05_chicken_movement` script canvas you just edited. Make sure the chicken entity's **Script Canvas** component is referencing the correct script canvas, otherwise the turn functionality you just added will not work in the next step.
    Select the chicken by clicking it in **Perspective**. In **Entity Inspector**, in the **Script Canvas** component, click the **Hierarchy** icon to the right of **Script Canvas Asset**, expand the **ScriptCanvas** directory and choose `ch05_chicken_movement.scriptcanvas`.

24. In the top toolbar, click the **Play** button or press **Control + G** to enter play mode. Use the **Left thumbstick** or **W**, **A**, **S**, and **D** keys to move the chicken. Use the **Right thumbstick** or **L** and **;** keys to turn the chicken. Tap the **Space** key or the **Right bumper** button on a gamepad repeatedly to make the chicken jump. Press **Escape** to exit play mode.



Note
This tutorial implements a very simple approach to handling turn input. The result of this tutorial is not framerate independent. The chicken will turn slower or faster depending on framerate. In `ch05_chicken_movement_final.scriptcanvas` you can examine a better implementation that multiplies the input by a `turnMaxVelocity` and puts the result in a `turnVelocity` variable. The turn is computed **On Tick** using `Delta` (the elapsed time between ticks) to make the rate the chicken turns consistent regardless of framerate.

For extra credit, try adding a new event generator to the input bindings. Create a small graph in Script Canvas that uses a **DrawTextOnScreen** node to display a message, variable, or value on screen when a key is pressed. Hint: use a **BuildString** Script Canvas node to create a string from a numerical value.

When you're ready, try the section called "Cameras" (p. 79).

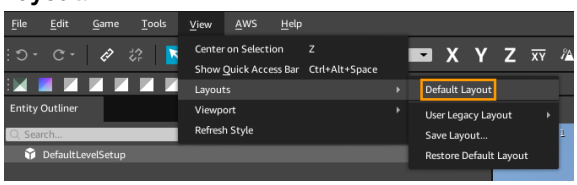
Tutorial Six: Add a camera

In this tutorial, you will create a camera entity and use **Script Canvas** to make a third-person camera follow the player. This tutorial introduces the **On Tick** event, and explains how to request data from other entities in **Script Canvas**.

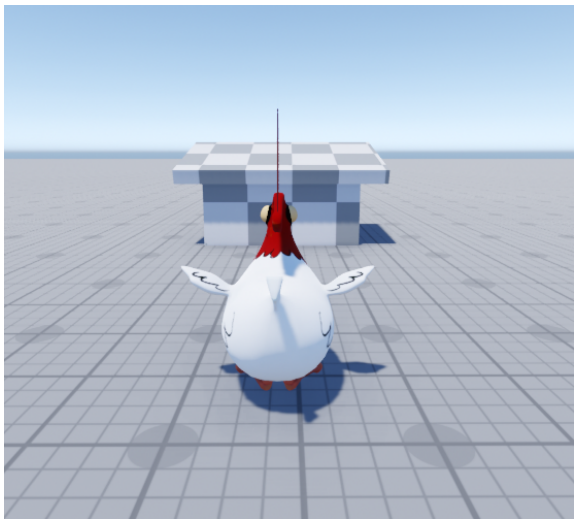
Tip
If you like, you can follow this chapter in video (4:41 minutes) form:

Begin this tutorial either with the level you created in the section called "Input and Scripting" (p. 74), or by opening `ch05_barnyard_final` from the `Levels` directory of the **WelcomeGuideTutorials** project. To open a level in Lumberyard, choose **Open Level...** from the **File** menu in the main menu bar.

1. This tutorial is written for the default **Lumberyard Editor** layout, so make sure this is the layout that you're using. To set the layout, access the menu bar and select **View**, **Layouts**, and choose **Default Layout**.

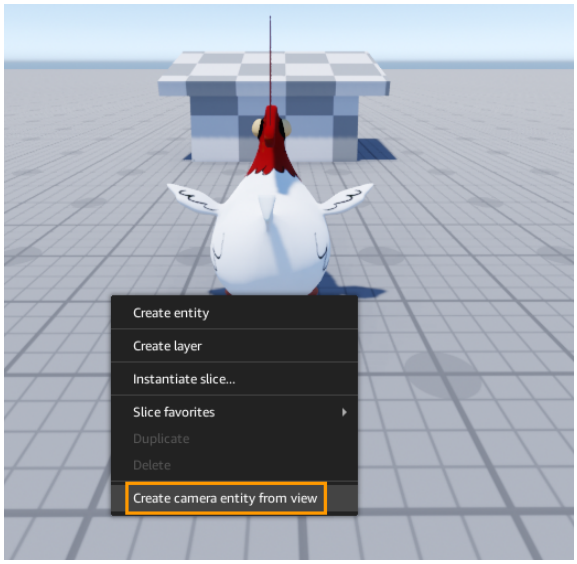


2. Use the navigation controls to line up the **Perspective** view roughly behind and above the chicken. The exact view is not important. The camera you create in this tutorial automatically provides a consistent view.



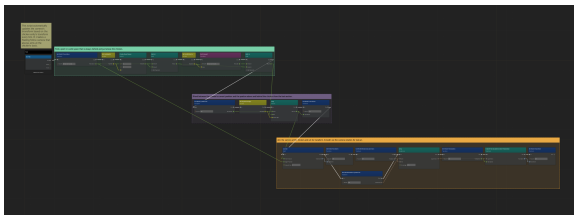
3. Right-click in the viewport to open the context menu and select **Create camera entity from view**. A new entity is added to the level containing a transform and a **Camera** component. The default settings for the camera component are suitable for this tutorial.

   Zoom out and adjust your view with the **S** key and right mouse button to get a view of the camera entity. The camera entity displays the camera's frustum as a yellow wire frame. The frustum represents the field of view of the camera. Enter play mode and **Perspective** snaps to the camera entity's view. Press Escape to exit play mode.



4. Rename the camera entity. Select the camera entity by clicking its name in **Entity Outliner** or by clicking the camera in **Perspective**. Name the entity `player_camera`.
5. Add a `.scriptcanvas` to the `player_camera` entity. The provided `.scriptcanvas` updates the camera entity's position and rotation in response to the chicken's movement creating a floating follow camera. In **Entity Inspector**, choose **Add Component** and add a **Script Canvas** component to the `player_camera` entity.
6. Open **Script Canvas** by expanding the **Tools** menu and choosing **Script Canvas**.

7. In **Script Canvas**, expand the **File** menu, and choose **Open**. In the `ScriptCanvas` directory, select `ch06_chicken_camera.scriptcanvas`.



Before continuing, take a moment to examine the script and understand its function.

The entry node (the node furthest to the left) is an **On Tick** event. Every entity has an **On Tick** event. Any networks connected to this event node will execute for every tick of the game clock, generally once per frame, 30, 60, or more times per second. You should use **On Tick** whenever your entity needs to process something at regular intervals. It's best to keep the processes that use **On Tick** simple. Complicated and computationally expensive processes that run every tick can result in poor performance.

The first node group (the green box) finds the position and orientation of the chicken in world space, and uses the **camDistance** and **camHeight** variables to create a position behind and slightly above the chicken.

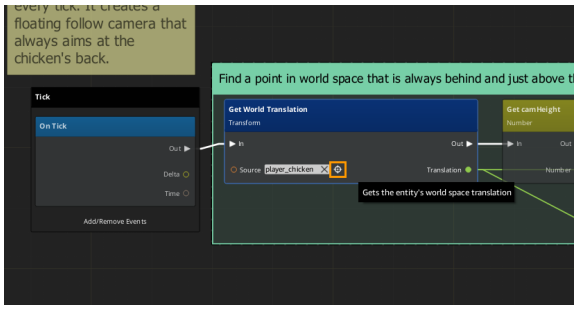
The second node group (the purple box) takes the position created in the first node group and the position of the camera at the current tick, and *lerps* the two positions based on a **lerpPercentage** variable. Lerp is shorthand for *linear interpolation*, which is a function that estimates a new position between the two given positions, weighted by a value (**lerpPercentage** in this scenario). The position created by the **Lerp** node is somewhere between the camera's current position and the position calculated in the first node group. The result is that the camera follows the chicken with a slight lag that softens the camera movement. If you use the position generated in the first node group without this **Lerp** function, the resulting camera movement is very sharp and jarring. You can adjust the camera movement lag by changing the **lerpPercentage** variable. Use values between `0.0` and `1.0`.

At the end of the second node group, the camera is positioned behind the chicken, facing the same direction as the chicken, but not actually looking at the chicken. In the third node group (the orange box), the **Look At** node produces a transform (a matrix representing a position, orientation, and scale) that ensures the camera is positioned behind and above the chicken, and rotated to look at the chicken. The slight rotations added to the camera by **Look At** can be jarring when the chicken jumps. The final group of nodes smooth out the camera rotation.

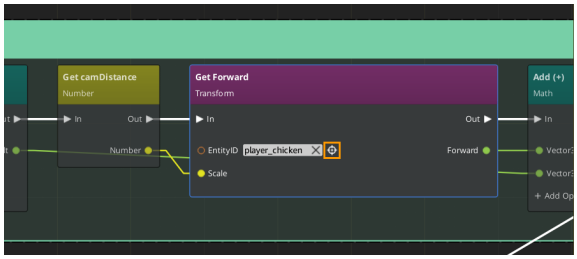
The script gets the world rotation of the camera at the current tick and the world rotation of the camera after the **Look At** has been applied. These world rotations are formatted as *quaternions*. A quaternion represents a series of *Euler* rotations around the X, Y, and Z axes as a single rotation around a new axis. The primary reason for using quaternions is to avoid *gimbal lock*, which is a common issue with Euler rotations. If this camera smoothing functionality had been implemented with Euler rotations, many more nodes would be required. In an Euler implementation, in certain circumstances, the camera might flip and roll unpredictably due to gimbal lock. With quaternions, the rotations are lerped, producing a camera rotation somewhere between the current camera rotation and the **Look At** rotation. The result is put in a transform and applied to the camera entity, softening the rotation of the camera.

8. The `ch06_chicken_camera.scriptcanvas` is functionally complete, but needs to reference the `player_chicken` entity to get the location and orientation of the chicken. You might notice that some nodes, such as the **Get World Translation** and **Get Forward** nodes in the first node group, have input pins labeled **Source** and **Entity ID**. These inputs are set to `Self` when they are placed in a graph, which means they will get the required data from the entity that the script is attached to. These inputs can reference other entities in the level, however.

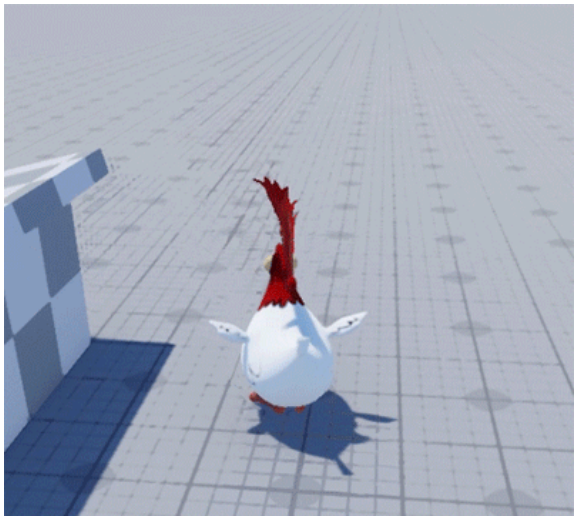
Find the **Get World Translation** node in the first node group, click the target button to the right of **Source**, then click the `player_chicken` entity in **Perspective**. This references the chicken entity in the **Get World Translation** node and allows the camera script to get the chicken's position in the world.



9. Find the **Get Forward** node in the first node group, click the target button to the right of **EntityID**, then click the `player_chicken` entity in **Perspective**. This allows the camera script to get the chicken's orientation.



10. Press **Control + S** to save the `ch06_chicken_camera.scriptcanvas`, and close the **Script Canvas** window.
11. Assign the `ch06_chicken_camera.scriptcanvas` to the **Script Canvas** component you added to the `player_camera` entity. Select the camera by clicking it in **Perspective**. In **Entity Inspector**, in the **Script Canvas** component, click the **Hierarchy** icon to the right of **Script Canvas Asset**, expand the **ScriptCanvas** directory and choose `ch06_chicken_camera.scriptcanvas`.
12. Test the camera. Press the **Play** button or **Control + G** and move the chicken. The camera should follow behind the chicken.



For extra credit, try changing the **Field of View** of the camera component to see its effect. Try to change the camera script canvas to create a side-scrolling like camera view.

When you're ready, try the section called "Terrain" (p. 83).

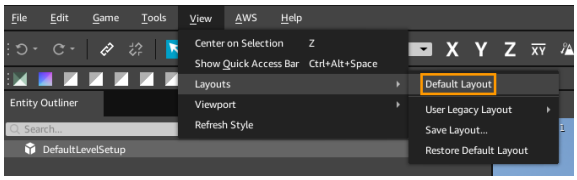
Tutorial Seven: Create terrain

In this tutorial, you will use the **Terrain Editor**, **Terrain Tool**, **Texture Layers**, and **Material Editor** to create a grassy island. You use **Terrain Editor** to create heightmaps that shape the terrain. In **Texture Layers**, you create layers and assign materials from **Material Editor** that can be blended across the terrain. Finally, you use **Terrain Tool** to paint materials onto the terrain and generate a terrain texture.

Tip
If you like, you can follow this chapter in video (6:45 minutes) form:

Begin this tutorial either with the level you created in the section called "Cameras" (p. 79), or by opening `ch06_barnyard_final` from the `Levels` directory of the **WelcomeGuideTutorials** project. To open a level in Lumberyard, choose **Open Level...** from the **File** menu in the main menu bar.

1. This tutorial is written for the default **Lumberyard Editor** layout, so make sure this is the layout that you're using. To set the layout, access the menu bar and select **View**, **Layouts**, and choose **Default Layout**.

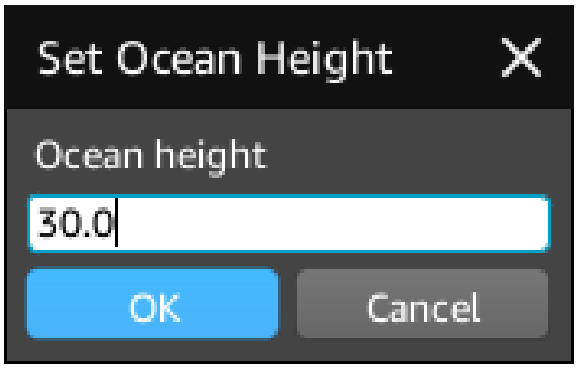


2. Open **Terrain Editor** to modify the terrain. Expand the **Tools** menu and choose **Terrain Editor**. With **Terrain Editor**, you can paint and modify your terrain's *heightmap*.

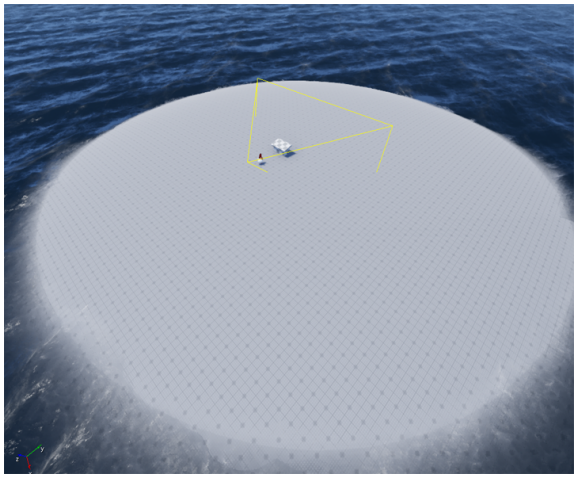
   A heightmap is a grayscale image where each pixel defines the height of one square meter of terrain. Heightmaps in Lumberyard are 32-bit. Each pixel in the heightmap contains a value between 0.0 and 1.0. Black (0.0) pixels are low areas. White (1.0) pixels are high areas. The height in world space of a white pixel is defined by the **Max Height** property.
3. Set the terrain's **Max Height**. In **Terrain Editor**, expand the **Modify** menu and choose **Set Terrain Max Height**.
4. You might recall from the first tutorial that the terrain is at 32.0 on the Z axis in world space. In the **Set Terrain Max Height** dialog, enter `32.0` and choose **OK**.

5. The infinite ocean is currently at 16.0 on the Z axis in world space. Move it closer to the terrain. In **Terrain Editor**, expand the **Modify** menu and choose **Set Ocean Height**.

6. In the **Set Ocean Height** dialog, enter `30.0` to place the ocean 2 meters below the terrain and choose **OK**.



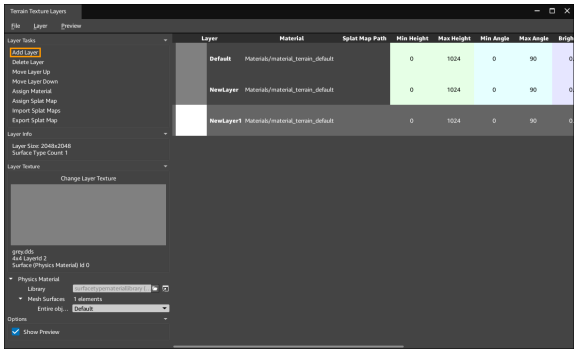
7. **Terrain Editor** has painting tools that you can use to edit heightmaps in the **Modify Terrain** panel to the right. **Terrain Editor** can also automatically generate heightmaps as a starting point. In **Terrain Editor**, expand the **Modify** menu, and choose **Make Isle**. **Terrain Editor** generates a smooth heightmap leaving the center of the terrain at 32.0 in Z, and the edges of the terrain gradually fall off beneath the surface of the ocean.



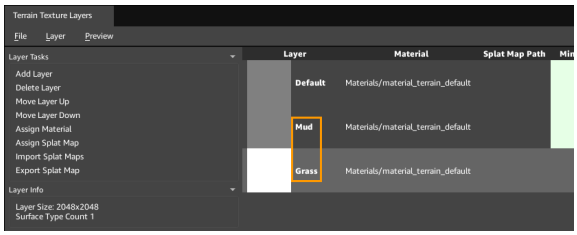
8. To make the terrain more visually interesting, you first must create texture layers for the terrain. In **Terrain Editor**, expand the **Tools** menu, and choose **Terrain Texture Layers...**.

   In **Terrain Texture Layers**, you can create layers for the terrain and assign materials to the layers. You then use **Terrain Tool** to paint a terrain texture that blends between the layers. Note there is currently one layer containing the default grid material displayed on the terrain.

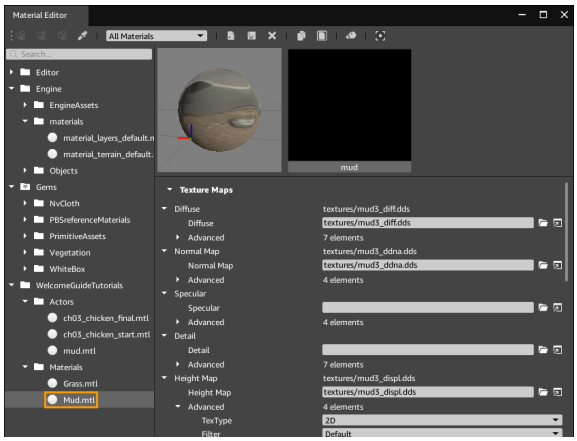
9. In **Terrain Texture Layers**, add two layers to the terrain by selecting **Add Layer** for the **Layer Tasks** panel on the right.



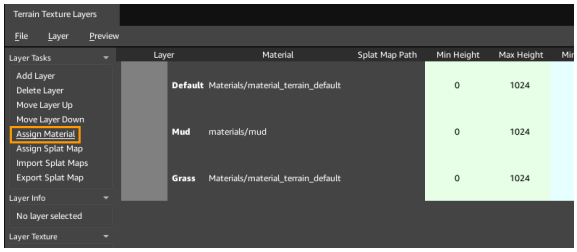
10. Rename the new layers. Double-click in the **NewLayer** cell and enter `Mud`.
11. Double-click in the **NewLayer1** cell and enter `Grass`.



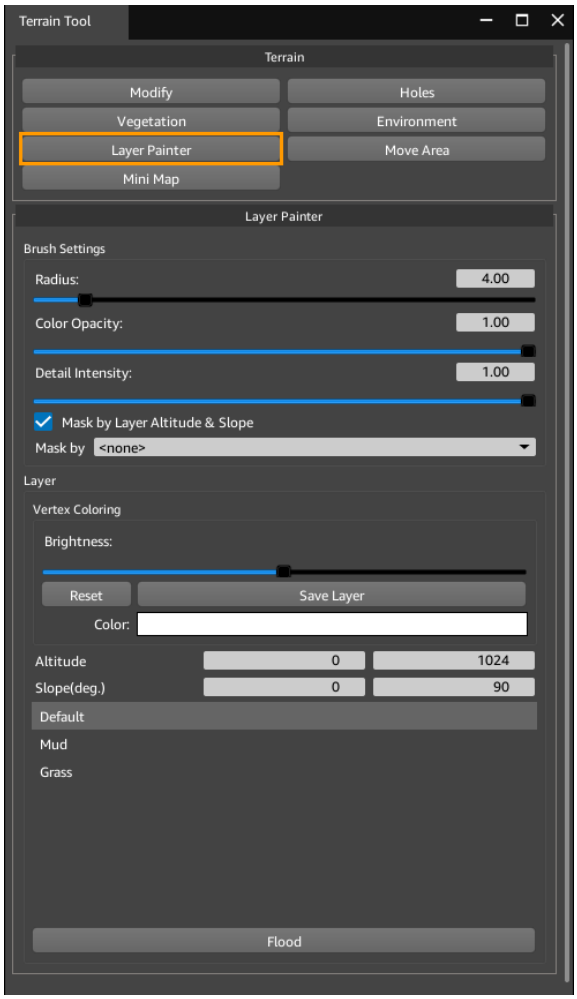
12. Materials are a combination of texture maps, shaders, and properties that create a look for a surface. Grass and Mud terrain layer materials are provided. Set a material for the mud layer. In **Terrain Texture Layers**, click the material name link in the **Material** cell of the `Mud` layer to open **Material Editor**.
13. **Material Editor** opens with the default terrain material selected. In **Material Editor**, in the directory tree on the left, expand the **WelcomeGuideTutorials** directory, expand the **Materials** directory, and click **Mud.mtl** to select the Mud material.



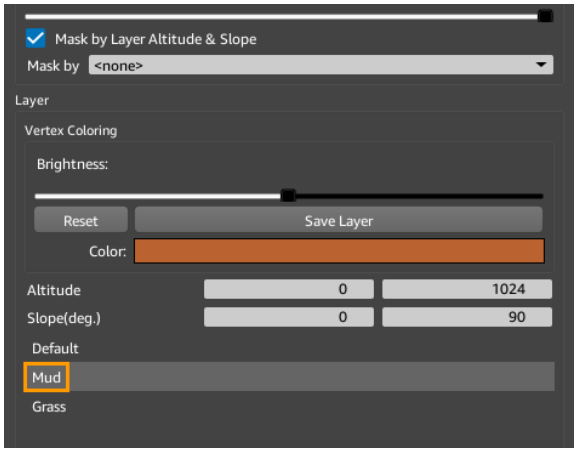
14. With the mud material selected, return to **Terrain Texture Layers**. In **Terrain Texture Layers**, in the **Layer Tasks** panel, click **Assign Material** to assign Mud.mtl to the Mud terrain layer.



15. Repeat the preceding two steps, this time assigning Grass.mtl to the Grass terrain layer.
16. Close **Terrain Texture Layers**, **Material Editor**, and **Terrain Editor**. Return to the Lumberyard editor.
17. In the Lumberyard editor, expand the **Tools** menu and choose **Terrain Tool**. The **Terrain Tool** tab hosts tools that contain some of the same functionality as **Terrain Editor**. In **Terrain Editor** you can paint heightmaps in a 2D view. With **Terrain Tool**, you can paint terrain maps and sculpt terrain in **Perspective**.
18. In **Terrain Tool**, choose **Layer Painter** to access the terrain layer painting tools.



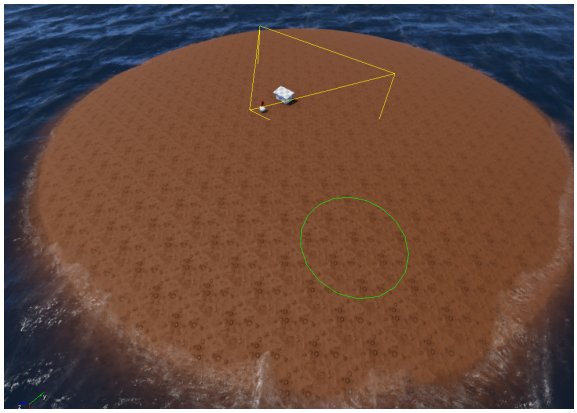
19. In **Layer Painter**, you can select a terrain layer, and paint the layer in **Perspective**. The layers you paint are tinted by a vertex color. In **Layer Painter**, in the lower **Layer** section, click **Mud** to select the mud layer.



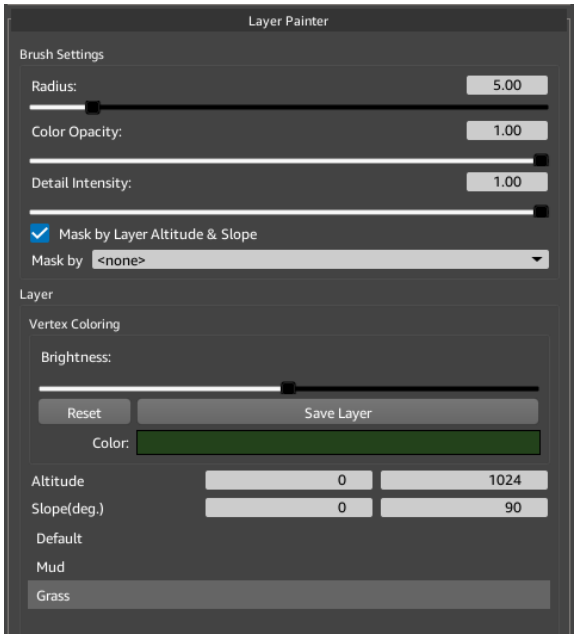
20. Choose a vertex color for the Mud layer. In **Layer Painter**, click the color swatch next to the **Color** property and set the vertex color to a desaturated orange using these values: **R** `187`, **G** `99`, **B** `48`.

21. Fill the terrain with the mud layer by clicking the **Flood** button at the bottom of **Layer Painter**. You might have to scroll down in **Terrain Tool** to locate the **Flood** button. The Mud layer is applied to the entirety of the terrain.

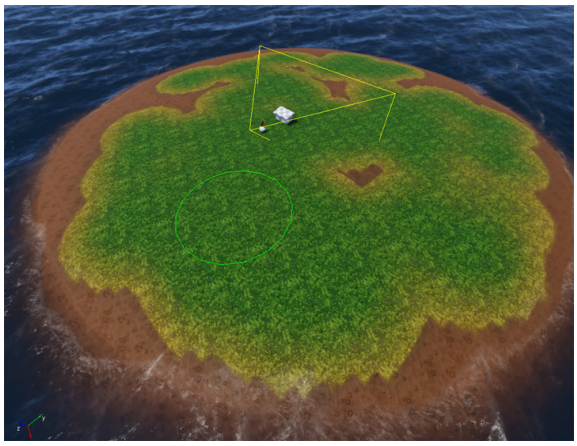
    Important
    The changes that you make in **Layer Painter** are made in memory. You must save the changes that you make to each layer as you work, and generate a new terrain texture to commit the changes to disk. Because the changes are stored in memory, **Perspective** might fail to display the most current changes. To force the changes to update, enter game mode by pressing the **Play** button in the tool bar or **Control + G**.



22. In **Layer Painter**, click the **Save Layer** button to save the changes you made to the Mud layer.
23. Paint some grass over the mud layer. In **Layer Painter**, in the lower **Layer** section, click **Grass** to select the Grass layer.
24. Choose a vertex color for the Grass layer. In **Layer Painter**, click the color swatch next to the **Color** property and set the vertex color to a desaturated green using these values: **R** `35`, **G** `66`, **B** `27`.
25. In **Brush Settings**, set **Radius** to 5.0. Because the terrain is set to `1` meter per texel, this radius setting creates a circular brush that covers about 10 meters of terrain.



26. In **Perspective**, click and drag on the terrain to paint the **Grass** layer over the **Mud** layer. Leave the **Mud** layer visible around the edges of the island, and leave some mud exposed in patches of grass.



27. In **Layer Painter**, click the **Save Layer** button to save the changes you made to the Grass layer.
28. Enter play mode by pressing the **Play** button or pressing **Control + G**. Walk around the island and see how the terrain texture layers blend. Also, notice that you can walk into the ocean. Press **Escape** to return to the editor.

29. The terrain texture must be committed to disk. In the Lumberyard editor, Expand the **Game** menu, choose the **Terrain** sub-menu, and select **Generate Terrain Texture**.
30. In the **Generate Terrain Texture** dialog, select `4096 x 4096` and choose **OK** to generate a terrain texture to disk.
31. You changed the heightmap for the terrain in this tutorial. For terrain collisions to be properly detected in the stand-alone version of your game, the **PhysX Terrain** level component must reference the new heightmap. To make the connection between the heightmap and the **PhysX Terrain** level component, delete the existing **PhysX Terrain** level component and add a new one. The heightmap will be automatically associated with the new **PhysX Terrain** level component.
    a. Choose the **Level Inspector** tab on the right side of the editor.
    b. Click the **PhysX Terrain** level component to select it.
    c. Press the **Delete** key to delete the **PhysX Terrain** level component.
    d. Choose **Add Component**, and select **PhysX Terrain** from the list of level components.

For extra credit, go to **Layer Painter** and experiment with changing the **Radius**, **Color Opacity**, and **Color** settings. Try using different vertex colors in the same layer and blending the results. Remember to save layer changes and regenerate the terrain texture.

Explore the other tools in **Terrain Tool**. Try the **Modify** tool and sculpt the terrain in **Perspective**.

When you're ready, try the section called "Slices" (p. 89).

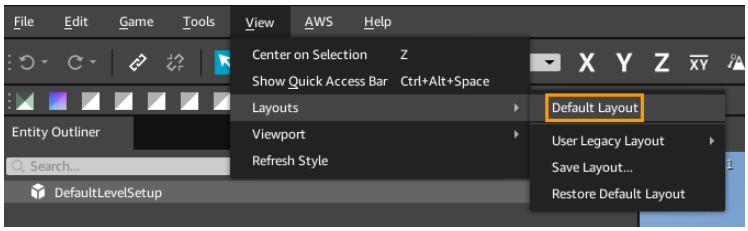
Tutorial Eight: Create environment props with White Box and slices

In this tutorial, you will learn how to make entities that can be instanced with unique properties by using *dynamic slices*. You will also learn how White Box meshes can be shared across entities.

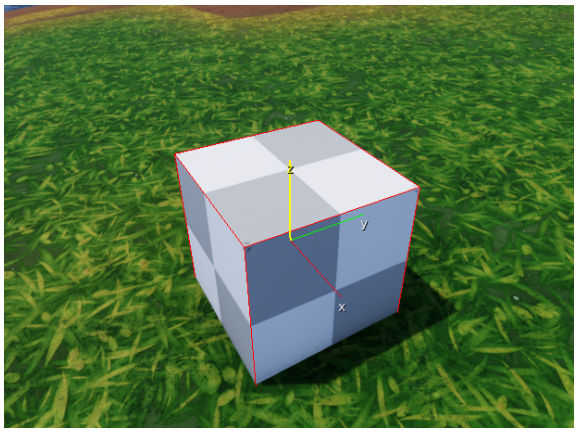
Tip
If you like, you can follow this chapter in video (6:33 minutes) form:

Begin this tutorial either with the level you created in the section called “Terrain” (p. 83), or by opening `ch07_barnyard_final` from the `Levels` directory of the **WelcomeGuideTutorials** project. To open a level in Lumberyard, choose **Open Level...** from the **File** menu in the main menu bar.

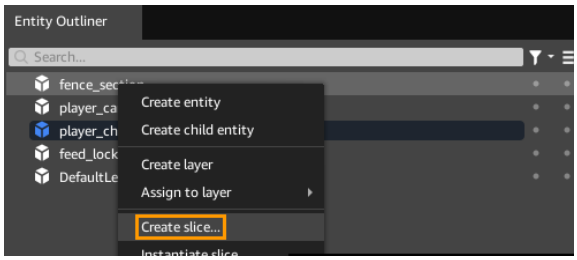
1. This tutorial is written for the default **Lumberyard Editor** layout, so make sure this is the layout that you're using. To set the layout, access the menu bar and select **View**, **Layouts**, and choose **Default Layout**.



2. Create a new entity for a section of fence. In **Perspective**, right-click and select **Create Entity** form the context menu.
3. With the new entity selected, in **Entity Inspector**, set its **Name** property to `fence_section1`.
4. Press the **Z** key to focus on the `fence_section1` entity in **Perspective**.
5. Add a white box component. With the `fence_section1` entity selected, in **Entity Inspector**, choose **Add Component**. Begin typing `whi` in the search field, and select **White Box** from the filtered list.
6. Add a collider. In **Entity Inspector**, choose **Add Component**. Begin typing `whi` in the search field, and select **White Box Collider** from the filtered list.
7. Press the **2** key or select the **Move** tool from the toolbar to move the entity. Move the `fence_section1` entity so that it's roughly sitting on the terrain.

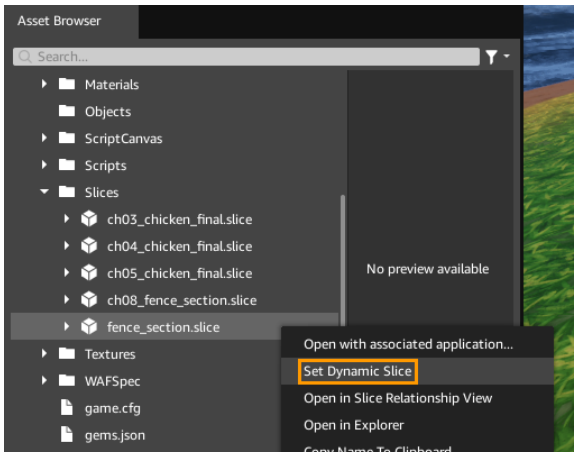


8. Create a slice from the `fence_section1` entity. In **Entity Outliner**, right-click the `fence_section1` entity and select **Create Slice** from the context menu.

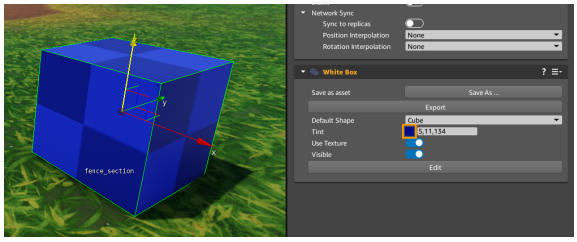


9. Name the slice `fence_section1.slice` and ensure the file browser is in the **Slices** directory, and choose **Save**.
10. In **Asset Browser**, expand the **Slices** directory, right-click `fence_section1.slice`, and select **Set Dynamic Slice** from the context menu.

    Dynamic slices are optimized versions of slices that have their editor dependencies removed and can be instantiated at run-time.



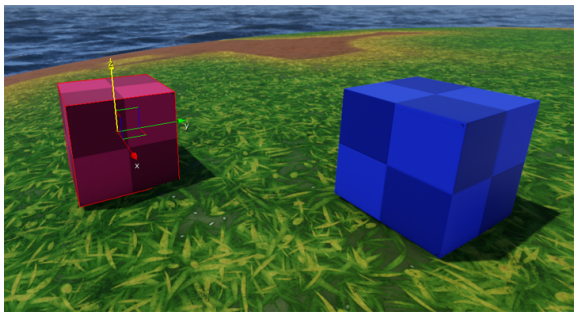
11. Change the color of the `fence_section1` dynamic slice. Click the `fence_section1` dynamic slice in **Perspective** to select it. In **Entity Inspector**, in the **White Box** component, click the color swatch to the right of **Tint**, and use the color picker to change the color of the fence section to blue.



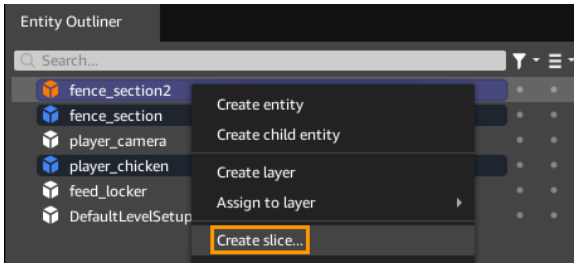
12. Save the mesh created by the **White Box** component to disk. With the `fence_section1` dynamic slice still selected, in **Entity Inspector**, in the **White Box** component, choose **Save As...** to save the cube as a white box mesh.
13. In the **Save As Asset...** dialog, expand the Objects directory, set `fence_section_whitebox.wbm` as the file name, and choose **Save**.

    The cube has now been saved to disk as a white box mesh (`.wbm`). White box meshes can be loaded into **White Box** components and referenced across entities. Any further edits are automatically saved and propagated to entities that use the white mesh.
14. Load the saved mesh in the **White Box** component. In **Entity Inspector**, in the **White Box** component, set **Default Shape** to `Custom Mesh Asset`. Choose the **Directory** button to the right of **Mesh Asset** and select the 'fence_section_whitebox.wbm' from the **Objects** directory.

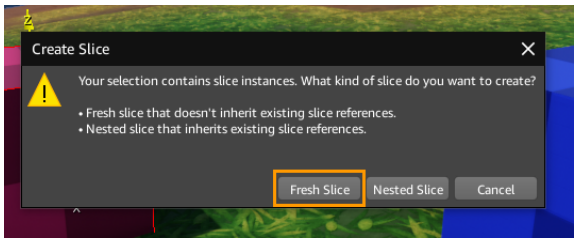
15. In **Entity Outliner**, right-click the `fence_section1` dynamic slice, choose **Save Slice Overrides** and select `fence_section1.slice` to quick save the changes to the dynamic slice.
16. To demonstrate the capabilities of the white box mesh, create a duplicate of the `fence_section1` dynamic slice. In **Asset Browser**, locate `fence_section1.slice` in the **Slices** directory. Click and drag `fence_section1.slice` into **Perspective** and place it near the original fence section.
17. With the new dynamic slice selected, in **Entity Inspector**, set its **Name** property to `fence_section2`.
18. Set a different color for `fence_section2` In **Entity Inspector**, in the **White Box** component of the `fence_section2` slice instance, click the color swatch to the right of **Tint**, and use the color picker to change the color of the fence section to red.



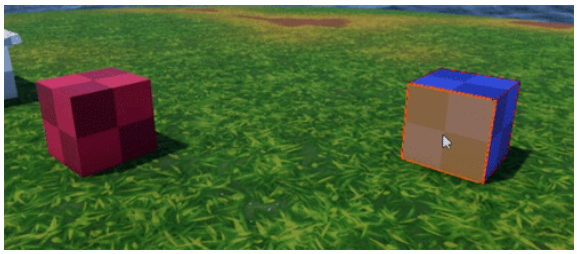
19. Create a new slice from `fence_section2`. In **Entity Outliner**, right-click `fence_section2` and select **Create Slice**.



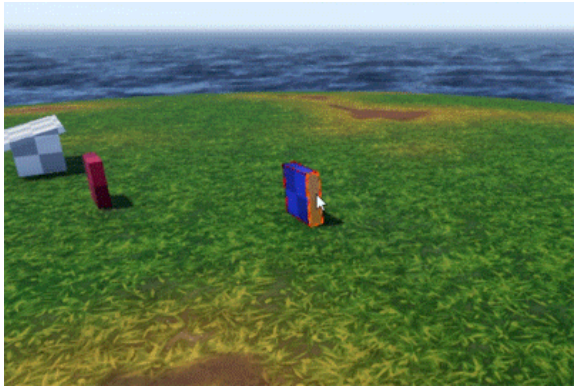
20. A warning appears in the **Create Slice** dialog because `fence_section2` is an instance of an existing slice. Choose **Fresh Slice** to create a new slice.



21. Name the slice `fence_section2.slice` and ensure the file browser is in the **Slices** directory, and choose **Save**.
22. In **Asset Browser**, expand the **Slices** directory, right-click `fence_section2.slice`, and select **Set Dynamic Slice** from the context menu.
23. Now edit the mesh of the `fence_section1` dynamic slice. Click the `fence_section1` slice in **Perspective** to select it.
24. In **Entity Inspector**, in the **White Box** component, choose **Edit** to enter White Box edit mode.
25. Click and drag on the front face of the box, to make the box thinner. Notice the `fence_section2` slice updates in real time because it is using the same white box mesh.

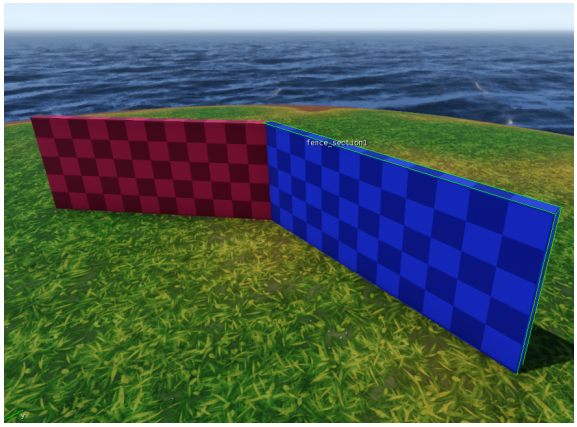


26. Click and drag on the top and side of the `fence_section1` slice to create a wall that is 3 units high and 5 units long. The checkerboard texture can be used to size the white box mesh. Each square in the texture is a half unit.



Tip
To make more precise modifications, enable the **Snap To Grid** button in the **Lumberyard Editor** and set the snap grid value to 1.0. Then enter White Box edit mode and extrude by 1 unit measurements.

1. In **Entity Inspector**, in the **White Box** component, choose **Done** to exit White Box edit mode.
2. Select either fence section slice and use the move tool (**2** key) and the rotate tool (**3** key) to position the fence sections at a slight angle and overlapping each other a bit.



If you press the **Play** button, or **Control + G**, to enter play mode, the chicken will be able to jump over the fence if you press jump repeatedly. For extra credit, see if you can limit the number of times the player can press jump to prevent the chicken from clearing the fence. Hint: use the **Get Tick Delta Time** node to create a timer.

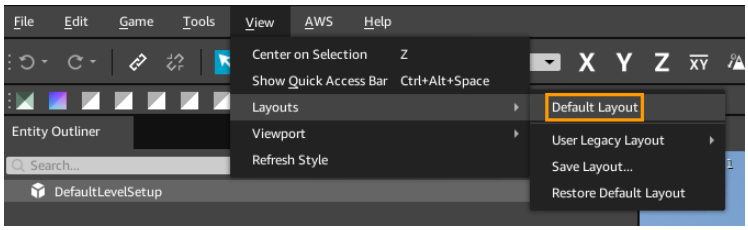
Tutorial Nine: Package and build a release

In this tutorial, you will learn how to export your level for the engine, set a default level, and test your level in a standalone game. Finally, you'll learn how to bundle assets and create a stand-alone release build of your project.

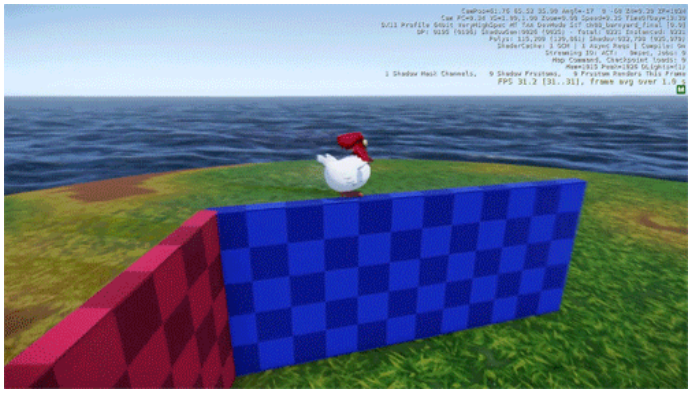
Tip
If you like, you can follow this chapter in video (2:51 minutes) form:

Begin this tutorial either with the level you created in the section called "Slices" (p. 89), or by opening `ch08_barnyard_final` from the `Levels` directory of the **WelcomeGuideTutorials** project. To open a level in Lumberyard, choose **Open Level...** from the **File** menu in the main menu bar.

1. This tutorial is written for the default **Lumberyard Editor** layout, so make sure this is the layout that you're using. To set the layout, access the menu bar and select **View**, **Layouts**, and choose **Default Layout**.



2. The level needs to be exported to a run-time asset for Lumberyard. To export your level, expand the **Game** menu in top menu bar and choose **Export to Engine** or press **Control + E**. A message appears reporting success. Choose **OK** to close the message box.
3. There are several levels in the **Levels** directory. You must specify one level to be the start-up level that will load when the game begins. This is done by creating an `autoexec.cfg` file in the root of the **WelcomeGuideTutorials** project folder that specifies the level to load. An `autoexec.cfg` has been provided for the project. The file contain a single line: `map ch08_barnyard_final`. If your working level has been saved under a different name, use a text editor to replace `ch08_barnyard_final` with the name of your level.
4. To test your game as a stand-alone before committing to bundling the assets, and building a release, find the **WelcomeGameTutorialsLauncher.exe** in your build folder located in `lumberyard_root\dev\Bin64vc141` (if you are using Visual Studio 2017) or located in `lumberyard_root\dev\Bin64vc142` (if you are using Visual Studio 2019). Run **WelcomeGameTutorialsLauncher.exe**. The project runs in a stand-alone window and automatically loads the level. Test all the functionality of the keyboard and gamepad. When you are satisfied everything works, press **Escape** to exit.



Additional Lumberyard resources

After setting up Amazon Lumberyard, learn how to use the platform with our Getting Started video series. This series covers all the basics of working with Lumberyard and is a terrific foundation for moving on to intermediate and advanced topics.

- Getting Started with Amazon Lumberyard
  - Lumberyard also has an ever-growing YouTube channel that's loaded with additional tutorial videos and presentations. Subscribe and enable notifications so you always know when new videos are released.
- Amazon Lumberyard on YouTube
  - Dive deeper into the features and usage of Lumberyard with the Amazon Lumberyard documentation.
- Amazon Lumberyard documentation home
  - Join the online community in the Lumberyard forums to learn from Lumberyard staff and users, and share your projects.
- Amazon Lumberyard forums
  - Lumberyard users have created an unofficial Discord. Join the conversation to share tips and get help from Lumberyard users.
- Amazon Lumberyard on Discord

Legal information

The Amazon Lumberyard engine, integrated development environment, and related assets and tools are licensed as "Lumberyard Materials" under the terms and conditions of the AWS Customer Agreement and the Lumberyard Service Terms. Please see these terms and conditions for details.

Topics

- Lumberyard redistributables (p. 95)
- Alternate web services (p. 96)

Lumberyard redistributables

For purposes of the Lumberyard Service Terms, the Lumberyard materials in the directories listed below are designated as "Lumberyard Redistributables." Unless subdirectories of a directory are specified, all files in the directory listed are deemed Lumberyard Redistributables.

> **Note**
> Restrictions on use and distribution of the Lumberyard materials, including in source code form, are specified in the Service Terms.

- `\3rdParty\GameLift`
- `\dev_WAF_`
- `\dev\Bin64`
- `\dev\CloudGemSamples`
- `\dev\Code\CloudGemSamples`
- `\dev\Code\CryEngine`
- `\dev\Code\Framework`
- `\dev\Code\Launcher`
- `\dev\Code\MultiplayerProject`
- `\dev\Code\SamplesProject`
- `\dev\Code\Sandbox`
- `\dev\Code\Tools`
- `\dev\Code\Tools\AssetTagging`
- `\dev\Code\Tools\ClangReflect`
- `\dev\Code\Tools\CryCommonTools`
- `\dev\Code\Tools\CryD3DCompilerStub`
- `\dev\Code\Tools\CrySCompilerServer`
- `\dev\Code\Tools\CryXML`
- `\dev\Code\Tools\DBAPI`
- `\dev\Code\Tools\GemRegistry`
- `\dev\Code\Tools\HLSLCrossCompiler`
- `\dev\Code\Tools\LUARemoteDebugger`
- `\dev\Code\Tools\PRT`
- `\dev\Code\Tools\RC`
- `\dev\Code\Tools\ShaderCacheGen`

- `\dev\Code\Tools\SphericalHarmonics`
- `\dev\Code\Tools\AssetProcessor`
- `\dev\Editor`
- `\dev\Engine`
- `\dev\FeatureTests`
- `\dev\Gems`
- `\dev\MultiplayerProject`
- `\dev\ProjectTemplates`
- `\dev\SamplesProject`
- `\dev\Tools\Build\waf-1.7.13`
- `\dev\Tools\lmbr_aws\AWSResourceManager\default-project-content`
- `\dev\AssetProcessorPlatformConfig.ini`
- `\dev\bootstrap.cfg`
- `\dev\editor.cfg`
- `\dev\engineroot.txt`
- `\dev\lmbr_aws.cmd`
- `\dev\lmbr_waf.bat`
- `\dev\lmbr_waf.exe`
- `\dev\SetupAssistantConfig.json`
- `\dev\system_BuildShaderPak_DX11.cfg`
- `\dev\system_BuildShaderPak_GL4.cfg`
- `\dev\system_windows_pc.cfg`
- `\dev\waf_branch_spec.py`
- `\dev\wscript`
- All directories

Alternate web services

For purposes of the Lumberyard Service Terms, "Alternate web service" means any non-AWS compute, database, storage, or container service that is similar to or can act as a replacement for the following services: Amazon EC2, Amazon Lambda, Amazon DynamoDB, Amazon RDS, Amazon S3, Amazon EBS, Amazon EC2 Container Service, or Amazon GameLift.