

# **Game Programming with Open 3D Engine**

**Getting Started with 21.11+**

**Oleksandr Lozitskiy**

---

# **Game Programming with Open 3D Engine: Getting Started with 21.11+**

Oleksandr Lozitskiy

Publication date 2022

Copyright © 2022 Oleksandr Lozitskiy

---

# Table of Contents

Introduction .....	xi
Copyright .....	xi
My Mindset for this Book .....	xi
Intended Audience .....	xi
Source Code for this Book .....	xii
Use of PowerShell throughout the Book .....	xiv
I. Getting Started .....	1
1. Installing Open 3D Engine on Windows .....	3
2. Creating a New Project .....	6
II. Entities and Components .....	13
3. Introduction to Entities and Components .....	15
4. Writing Your Own Components .....	29
III. Introduction to Component Communication .....	41
5. What is FindComponent? .....	43
6. What is AZ::Interface? .....	50
7. What is an AZ::Event? .....	56
8. What is an AZ::EBus? .....	62
9. Using AZ::TickBus .....	70
IV. Introduction to Component Reflection and Prefabs .....	79
10. Configuring Components in the Editor .....	81
11. Introduction to Prefabs .....	88
V. Modularity in O3DE .....	99
12. What is a Gem? .....	101
13. Gem: Set Window Position .....	110
14. Enabling NvCloth Gem .....	115
VI. Unit Tests in O3DE .....	120
15. Writing Unit Tests for Components .....	122
16. Unit Tests with Mock Components .....	131
VII. Character Controller .....	140
17. Player Input .....	142
18. Character Movement .....	150
19. Turning using Mouse Input .....	159
VIII. Building Environment .....	167
20. Adding Physics to the World .....	169
21. Introduction to Materials and Lights .....	176
IX. Building Game Play .....	183
22. Introduction to User Interface .....	185
23. Interacting with UI in C++ .....	192
24. Kicking the Ball .....	204
25. Introduction to Animation .....	212
26. Animation State Machine .....	225
27. Script Canvas .....	231
X. Audio Effects (Wwise) .....	237
28. Wwise for O3DE .....	239
29. Importing Wwise Project .....	247
30. Introduction to Audio Components .....	249
XI. Multiplayer .....	254
31. Setting Up Multiplayer .....	257
32. Auto Components .....	263
33. Multiplayer in the Editor .....	271
34. Simple Player Spawner .....	274

35. Multiplayer Input Controls .....	282
36. Multiplayer Physics .....	296
37. Removal and Spawning .....	304
38. Multiplayer Animation .....	313
39. Team Spawner .....	319
40. Multiplayer Camera .....	327
Index .....	333

---

# List of Figures

2.1. Editor: new level .....	11
3.1. An Entity with Transform and Camera components .....	16
3.2. Editor: Entity Outliner .....	17
3.3. Editor: A sphere Entity in "Simple" level .....	18
3.4. Editor: Create (child) entity .....	18
3.5. Editor: child entity .....	19
3.6. Entity with Child Transform .....	19
3.7. Entity with Mesh component .....	20
3.8. Selecting Mesh Asset via Interactive Search .....	21
3.9. Editor: _Sphere_1x1 entity .....	21
3.10. Editor: Composite Object .....	22
3.11. Editor: Creating an Entity .....	22
3.12. Editor: Creating a child Entity .....	22
3.13. Cylinder 1 Entity .....	26
3.14. Cylinder 2 Entity .....	27
4.1. MyProject solution in Visual Studio .....	30
4.2. MyProject.Static library .....	31
4.3. MyProjectModule.cpp in Visual Studio .....	36
4.4. Visual Studio solution with MyComponent .....	38
6.1. Accessing Level entity and its components .....	50
6.2. Level entity in Entity Inspector .....	51
6.3. Using AZ::Interface .....	52
7.1. Setting up an AZ::Event .....	56
7.2. TransformComponent notifies using AZ::Event .....	58
8.1. Calling through an interface behind an EBus .....	64
8.2. TransformBus and TransformComponent .....	66
10.1. Basic Serialization .....	82
10.2. SerializeContext and EditContext .....	85
11.1. How to bring up Entity Outliner .....	89
11.2. How to bring up Asset Browser .....	89
12.1. Build structure of a gem .....	105
12.2. Gem Code Structure .....	106
12.3. mygem_files.cmake .....	109
13.1. IConsole snippet .....	111
13.2. Example of a console command: SetWindowXY .....	113
14.1. Configure Gems .....	116
14.2. Chicken from NvCloth gem .....	117
14.3. Instantiate Prefab .....	117
14.4. Pick Chicken_Actor.prefab .....	118
14.5. Entities in the Level .....	119
14.6. Chicken in O3DE .....	119
15.1. Location of unit tests in a gem .....	123
15.2. Location of unit tests in a gem .....	124
17.1. Picking third person movement input bindings .....	143
17.2. Input component with third person movement bindings .....	143
17.3. Asset Viewer .....	143
17.4. "Move forward" action .....	144
17.5. Design of Capturing Input .....	144
17.6. Input Binding for "move forward" in Asset Editor .....	145
17.7. Chicken Actor with Chicken Controller component .....	146
18.1. Design of Moving the Chicken .....	153

18.2. Chicken in game mode with <b>CTRL+G</b> .....	155
19.1. Turning using Mouse .....	160
20.1. Soccer Field .....	169
20.2. Soccer Field Entities .....	170
20.3. Components of Side Entity .....	170
20.4. Non-uniform Scale component .....	171
20.5. PhysX Collider Mesh Asset .....	171
20.6. Soccer Field parts as prefabs .....	171
20.7. Goal_End.prefab .....	171
20.8. Goal_End shapes .....	172
20.9. 3D text: O3DE .....	172
20.10. O3DE physical mesh .....	174
20.11. Ball components .....	174
20.12. Ball shape .....	175
20.13. Ball mesh .....	175
20.14. Soccer Ball .....	175
21.1. Entity Atom Default Environment with <b>Ground</b> child entity .....	176
21.2. Soccer Ball .....	176
21.3. Soccer Ball .....	177
21.4. Soccer Ball .....	177
21.5. `Add Material Component` button .....	178
21.6. Sun entity .....	178
21.7. Add Spot Light child entity .....	179
21.8. Light component with Spot type .....	179
21.9. Spot light on the goal line .....	179
21.10. Spot disk light .....	180
21.11. White material on a mesh .....	180
21.12. Spot disk light .....	180
21.13. Corner flood lamp .....	181
21.14. Default_grid material .....	181
21.15. Yellow base color .....	181
21.16. New level look .....	182
22.1. Hierarchy of elements .....	187
22.2. Root element .....	188
22.3. Panel Anchors .....	188
22.4. Scaling an element .....	189
22.5. Image's alpha value .....	189
22.6. Score Text element .....	189
22.7. Score Text element .....	190
23.1. PhysX Trigger Shape .....	192
23.2. Signing up for collider volume trigger notifications .....	194
23.3. Updating a Text field in User Interface .....	197
24.1. Chicken with a Trigger Shape .....	206
24.2. KickingComponent .....	206
24.3. Design of Kicking Component .....	207
25.1. First Blending of Two Animations .....	218
26.1. Animation State Machine .....	225
26.2. Adding a Transition between Blend Trees .....	227
26.3. Animation State Machine .....	229
26.4. List of Parameters .....	229
26.5. <b>Anim Graph</b> component with updated parameters .....	230
27.1. Two Script Canvas Nodes .....	234
32.1. Components and Controllers .....	264
32.2. Overview of Code Generation of Auto-components .....	265

33.1. A Network Entity Wrapped in a Prefab .....	273
35.1. Sending Player Input to the Server .....	285
35.2. Processing Input on the Server .....	287
35.3. Network Inputs .....	290
36.1. Network Property from the Server to a Client .....	299
36.2. Updated Goal Entity .....	300
37.1. Design of the New Ball Spawner .....	304
37.2. Ball Spawner with a Prefab .....	306
37.3. Ball Spawner Entity .....	308
38.1. Design of Multiplayer Chicken Animation Component .....	314
38.2. Updated Chicken.prefab .....	316
39.1. Team Spawner Design .....	319
39.2. Chicken Component with a Team Value .....	320
39.3. The Tale of Two Chickens .....	322
39.4. Blue Chicken Team .....	324
40.1. Design of Chicken Camera Component .....	328

---

# List of Examples

2.1. C:\Users\<user>\.o3de\Registry\bootstrap.setreg .....	9
2.2. The Asset Processor finished processing game assets .....	10
4.1. MyProject\Code\myproject_files.cmake .....	31
4.2. A default module file for a new project, MyProjectModule.cpp .....	32
4.3. The simplest component .....	33
4.4. The simplest MyComponent.cpp .....	34
4.5. Building the project from command line .....	36
4.6. MyComponent.cpp with Editor reflection .....	38
5.1. FindComponent in AzCore\Component\Entity.h .....	43
5.2. An Example of Calling GetWorldTM on Transform Component .....	43
5.3. An example of FindComponent .....	44
5.4. Definition of MyProject.Static in CMake .....	44
5.5. AzToolsFramework\ToolsComponents\TransformComponent.cpp .....	45
5.6. C:\git\o3de\Code\Framework\AzFramework\CMakeLists.txt .....	46
5.7. MyProject.Static linking against AZ::AzFramework library .....	46
5.8. MyFindComponent.h .....	48
5.9. MyFindComponent.cpp .....	48
6.1. Level entity with My Level Component .....	51
6.2. C:\git\book\MyProject\Code\Include\MyProject\MyInterface.h .....	52
6.3. MyLevelComponent with AZ::Interface .....	52
6.4. Using AZ::Interface in another component .....	53
6.5. MyLevelComponent.h .....	53
6.6. MyLevelComponent.cpp .....	54
6.7. MyInterfaceComponent.h .....	54
6.8. MyInterfaceComponent.cpp .....	55
7.1. MyEventComponent.h .....	59
7.2. MyEventComponent.cpp .....	60
8.1. TransformBus definition .....	65
8.2. TransformInterface snippet .....	65
8.3. Example of an EBus call .....	66
9.1. OscillatorComponent.h snippet .....	71
9.2. Example of handling AZ::TickBus::Handler::OnTick .....	74
9.3. OscillatorComponent.h full listing .....	76
9.4. OscillatorComponent.cpp full listing .....	77
10.1. AzCore\Serialization\SerializeContext.h .....	83
10.2. AzCore\Serialization>EditContext.h .....	84
10.3. Reflect() with Editor Configuration .....	85
10.4. AzCore\Serialization>EditContextConstants.inl .....	86
11.1. Complex_Object.prefab snippet with <b>Root Entity</b> entity .....	92
11.2. Complex_Object.prefab snippet with OscillatorComponent .....	92
11.3. Method SpawnAllEntities .....	94
11.4. MySpawnerComponent .....	94
11.5. Spawn entities at a location .....	95
11.6. MySpawnerComponent with Complex_Object.prefab .....	96
11.7. MySpawnerComponent.h .....	96
11.8. MySpawnerComponent.cpp .....	97
12.1. enabled_gems.cmake .....	104
12.2. MyGemBus.h .....	108
12.3. MyGemModuleInterface .....	109
13.1. AZ_CONSOLEFUNC usage .....	111
13.2. Use of AZ_CONSOLEFREEFUNC .....	111

13.3. WindowX full method .....	112
13.4. WindowConsoleCommand.cpp .....	113
13.5. windowposition_files.cmake .....	114
14.1. Snippet from o3de_manifest.json .....	116
14.2. enabled_gems.cmake with NvCloth gem .....	116
14.3. O3DE.fbx in the level .....	118
15.1. Add MyProject.Tests build target .....	123
15.2. myproject_test_files.cmake .....	124
15.3. MyProjectTest.cpp .....	125
15.4. Unit test output .....	125
15.5. MyProjectTest.cpp .....	128
15.6. OscillatorTest.cpp .....	128
16.1. MockTransformComponent definition .....	133
16.2. OscillatorMockTest.cpp .....	137
17.1. Empty Input component .....	142
17.2. Changes to MyGem\Code\CMakeLists.txt .....	145
17.3. Receiving Input .....	146
17.4. ChickenControllerComponent.h .....	147
17.5. ChickenControllerComponent.cpp .....	147
18.1. ChickenControllerComponent.h with character movement .....	155
18.2. ChickenControllerComponent.cpp with motion .....	156
19.1. ChickenControllerComponent.h .....	162
19.2. ChickenControllerComponent.cpp .....	163
22.1. Enable LyShine and its supporting gems .....	185
23.1. Moving a Rigid Body .....	198
23.2. GoalDetectorComponent.h .....	199
23.3. GoalDetectorComponent.cpp .....	199
23.4. UiScoreComponent.h .....	201
23.5. UiScoreComponent.cpp .....	202
24.1. KickingComponent.h .....	209
24.2. KickingComponent.cpp .....	210
25.1. Simple Motion component .....	212
25.2. Chicken Running at the Ball .....	222
25.3. ChickenBus.h .....	223
25.4. ChickenAnimationComponent.h .....	223
25.5. ChickenAnimationComponent.h .....	223
27.1. Set Named Parameter Bool .....	231
27.2. Chicken_Actor entity with Script Canvas component .....	235
27.3. New code in GoalDetectorComponent.cpp .....	235
27.4. UiScoreBus.h .....	235
30.1. Playing the Trigger from Script Canvas .....	251
30.2. AudioTriggerComponentBus.h .....	251
30.3. Playing a Sound from C++ .....	252
31.1. A lot more is required .....	257
31.2. mygem_autogen_files.cmake .....	257
31.3. Completed MyGem.Static with multiplayer enabled .....	258
31.4. Changes to MyGemModuleInterface.h .....	259
31.5. Changes to MyGemSystemcomponent.cpp .....	259
31.6. MyFirstNetworkComponent.AutoComponent.xml .....	260
31.7. Changes to Gems\MyGem\Code\mygem_files.cmake .....	260
32.1. An auto-component .....	263
32.2. mygem_files.cmake with an auto-component .....	266
32.3. Provided Stub of MyFirstNetworkComponent.h .....	267
32.4. Provided Stub of MyFirstNetworkComponent.cpp .....	267

32.5. Accessing the Entity from a Controller .....	268
32.6. MyFirstNetworkComponent.h with a component .....	268
32.7. MyFirstNetworkComponent.cpp with a component .....	269
33.1. Turn on CLTR+G multiplayer with editor.cfg .....	271
33.2. Editor server log .....	273
34.1. ChickenBus.h .....	275
34.2. FirstChickenComponent.AutoComplete.xml .....	276
34.3. ChickenComponent.h .....	276
34.4. ChickenComponent.cpp .....	276
34.5. Snippet from ChickenSpawnComponent.h .....	277
34.6. Snippet from ChickenSpawnComponent.cpp .....	278
34.7. Full code for ChickenSpawnComponent.h .....	280
34.8. Full code for ChickenSpawnComponent.cpp .....	280
35.1. Additions to Gems\MyGem\Code\mygem_files.cmake .....	282
35.2. LocalPredictionPlayerInputComponent.AutoComplete.xml .....	285
35.3. Single-player create input logic .....	286
35.4. Multiplayer create input logic .....	286
35.5. Snippet from ChickenMovementComponent.AutoComplete.h .....	287
35.6. Single-player Process Input .....	288
35.7. Multiplayer Process Input .....	288
35.8. ChickenMovementComponent.h .....	291
35.9. ChickenMovementComponent.cpp .....	292
36.1. Complete GoalDetectorComponent.AutoComplete.xml .....	297
36.2. GoalDetectorComponent.AutoComplete.h .....	300
36.3. GoalDetectorComponent.h .....	301
36.4. GoalDetectorComponent.cpp .....	302
37.1. BallComponent.AutoComplete.xml .....	305
37.2. RespawnBall method .....	307
37.3. BallComponent.AutoComplete.h .....	308
37.4. BallBus.h .....	309
37.5. BallComponent.h .....	309
37.6. BallComponent.cpp .....	309
37.7. BallSpawnerComponent.AutoComplete.h .....	310
37.8. BallSpawnerBus.h .....	310
37.9. BallSpawnerComponent.h .....	310
37.10. BallSpawnerComponent.cpp .....	311
38.1. ChickenAnimationComponent.AutoComplete.h .....	316
38.2. ChickenAnimationComponent.h .....	317
38.3. ChickenAnimationComponent.cpp .....	317
39.1. ChickenSpawnComponent.h .....	324
39.2. ChickenSpawnComponent.cpp .....	325
40.1. Camera behind a Chicken .....	330
40.2. ChickenCameraComponent.AutoComplete.xml .....	330
40.3. ChickenCameraComponent.h .....	330
40.4. ChickenCameraComponent.cpp .....	331

---

# Introduction

## Copyright

To illustrate how to use Open 3D Engine, screen shots, images, models, textures and short snippets of source code belonging to and copyrighted by Open 3D Foundation and the Linux Foundation are reproduced in this book. Open 3D Engine (O3DE) is an Apache 2.0-licensed multi-platform 3D engine.

### Tip

You should refer to Open 3D Engine website and the Linux Foundation websites for details on their licenses:

- <https://o3de.org/>
- <https://github.com/o3de/o3de/blob/development/LICENSE.txt>
- <https://o3d.foundation/>

## My Mindset for this Book

Those who teach a topic they know well often commit an honest mistake. They forget all the steps they had to take to reach their depth of knowledge on the topic. Any complicated topic involves inter-connected expertise and knowledge. What is obvious to an expert is a mystery to a novice. And yet it is an expert that ought to be able to take a step back and consider the whole context of a given topic and find the easiest way to present it to a novice.

While writing this book, I was careful to always take several steps back from my own knowledge and consider what I had to know to become comfortable with a task or a subsystem of O3DE. Whenever I thought a certain idea was essential to comprehension, I would take another step back and see if there were other important preceding concepts and details that I had come to take for granted, or implicitly assumed to be there in my mind. And then I would carefully present each idea in the proper order.

I chose to error on the side of moving slowly and providing too much information rather than omit a detail that would stop someone from learning O3DE from scratch.

## Intended Audience

If you are new to O3DE and wish to learn the basics of the engine, then this is the book for you.

It will help a lot if you are a C++ developer already. O3DE is written primarily in C++. It does support scripting languages, such as Lua and a visual language Script Canvas. However, one gets the most benefit from O3DE knowing its fundamental layers in C++.

No one book can make a person an expert of O3DE. If I were to enumerate all the details and knowledge necessary to achieve that level, one would get bored reading all the material. A human mind does not work that way. Instead, this book goes over the essential sub-systems of the engine in moderate depth and shows how to use them. What is a component? What is Behavior Context? How does one add a sound effect? How do you build a server-authoritative multiplayer game in O3DE?

The end goal of this book is to provide a wide overview of the engine. After that you will be able to learn O3DE on your own beyond the scope of this book.

# Source Code for this Book

You will find a lot C++ source code in the book itself. I have included enough source code in the book that I believe you will be able to both grasp the material and use this book as a reference when implementing various game features and sub-systems in your future. I kept the coding examples as small as possible while still showing enough to explain the material.

Some programming books present code examples where a big portion of the code is hidden behind their custom helper frameworks. You will find no such helper functions in this book. I believe it is better to show the entire code and avoid hiding details behind methods whose only purpose is to make examples smaller.

If the source code printed in this book is not enough, the accompanying source code and assets in their entirety can be found on GitHub:

<https://github.com/AMZN-Olex/O3DEBookCode2111>

A unique Git branch is assigned to each chapter of this book. The main branch is empty except for an introductory file by design. Each following branch builds on the previous one. Each chapter includes a link to the correct GitHub branch from the above repository.

For example, the following GitHub links:

- [https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch15\\_unittests](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch15_unittests)
- [https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch16\\_unittests\\_mock](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch16_unittests_mock)

Correspond to Chapter 15, *Writing Unit Tests for Components* and Chapter 16, *Unit Tests with Mock Components*.

The format for branch names is: **ch{chapter number}\_{chapter name}**.

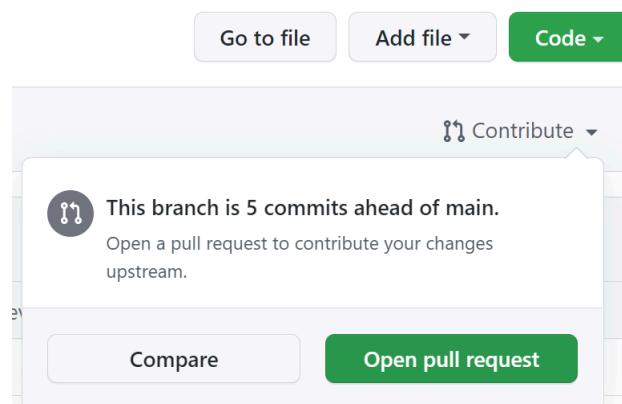
## Comparing GitHub Branches

It will be useful for you to compare two branches together to see what new changes were made in any given chapter. If you are not familiar with GitHub interface, here is how you can generate the difference between two Git branches.

1. Go to the branch associated with the chapter you are reading. For example:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch04\\_creating\\_components](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch04_creating_components)

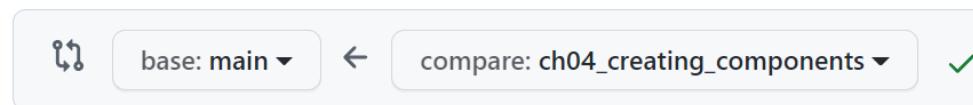
2. Select *Contribute* → *Open pull request*.



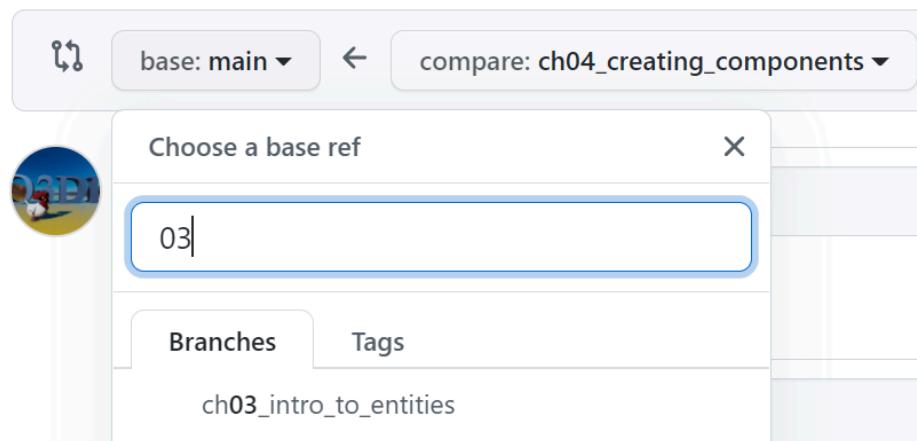
3. That will take you to **Open a pull request** page.

## Open a pull request

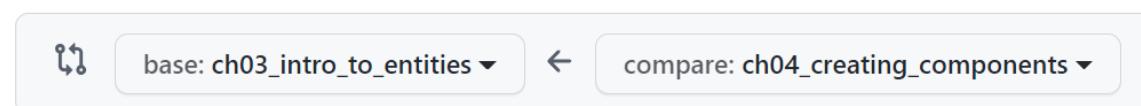
Create a new pull request by comparing changes across two branches. |



4. Select an earlier chapter as a base branch.



5. That will create a pull request configuration that will show all the differences between the two branches.



6. Scroll down a bit and you will see the changes.

 Showing 5 changed files with 55 additions and 0 deletions.

> 23  MyProject/Code/Source/MyComponent.cpp 

> 20  MyProject/Code/Source/MyComponent.h 

> ⏷ 2  MyProject/Code/Source/MyProjectModule.cpp 

> ⏷ 3  MyProject/Code/myproject\_files.cmake 

> ⏷ 7  MyProject/Levels/MyLevel/MyLevel.prefab 

## Use of PowerShell throughout the Book

I find it useful to write scripts to save myself the effort of typing out tedious commands. My personal choice for scripting on Windows is PowerShell. If you choose to use it as well then to run various scripts from the associated GitHub repository you will need to configure your system to allow the execution of PowerShell scripts. To do that you will need to change the execution policy to allow you to execute your custom PowerShell scripts.

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned
```

### Note

If you are curious what this command does, as you should, here is the official reference for it:

<https://go.microsoft.com/fwlink/?LinkID=135170>

If you are at all unsure of what this does to your system, then do not run it. You will not need it to understand the content of this book. You just will not be able to run some helpful script from various chapters, but you can always make your own in your favorite scripting language. You can find more documentation on PowerShell here:

<https://docs.microsoft.com/en-us/powershell/>

### Tip

You can always revert the changes to execution policy by executing:

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy Default
```

## Making PowerShell Your Development Console

I use PowerShell console as my build and configuration tool. Here is how you can set PowerShell to automatically start in your work directory and load helper scripts of your choice.

```
notepad $PROFILE
```

\$PROFILE is your personal configuration file for PowerShell that will execute commands of your choice every time a new PowerShell console opens. Therefore, we can modify this script to help us with O3DE development by adding helpful console shortcuts.

```
Set-Location "C:\git\book"

function Run-AP() {
    & "C:\O3DE\21.11.2\bin\Windows\profile\Default\AssetProcessor.exe"
}

function Run-Editor() {
    & "C:\O3DE\21.11.2\bin\Windows\profile\Default\Editor.exe"
}

Write-Output "Custom O3DE Tools loaded!"
```

The next time you open a PowerShell console, you will start in your work folder and load up any custom scripts that you write in \$profile.

```
C:\git\book> Run-Editor
```

And now onto learning Open 3D Engine! The game of Chicken Ball awaits!



---

## **Part I. Getting Started**

---

# Table of Contents

1. Installing Open 3D Engine on Windows .....	3
Introduction .....	3
Preparing Visual Studio .....	3
Acquiring O3DE .....	4
Running the Installer .....	4
2. Creating a New Project .....	6
Introduction .....	6
O3DE Command Line Interface .....	6
Linking Projects and O3DE .....	7
Building with CMake and Visual Studio .....	7
Asset Processor .....	8
Create a Level .....	11
Summary .....	11

---

# Chapter 1. Installing Open 3D Engine on Windows

## Introduction

This chapter will cover the following topics:

- Preparing Visual Studio 2019 (or 2022) for O3DE.
- Getting O3DE.
- Installing O3DE. (At the time of writing this book, O3DE 21.11.2 release was available.)

## Preparing Visual Studio

### Note

You can find the official guide on installing O3DE here:

<https://www.o3de.org/docs/welcome-guide/setup/>

O3DE is a C++ game engine, so it requires a few native Software Development Kits and various tools from Visual Studio (if you are building on Windows OS). By default, Visual Studio might not have some of these requirements, so I will go over the components that you do need to install.

### Note

Make sure to refer to the official reference on this topic:

<https://www.o3de.org/docs/welcome-guide/requirements/>

### Important

The supported version of Visual Studio by O3DE as of 21.11 is 2019, however, I have not found any issues with using Visual Studio 2022 and O3DE 21.11 together. The installation steps for either Visual Studio versions are the same.

Here are the steps:

1. Install Visual Studio 2019, Community or other editions.
2. Modify its installation to add *Desktop development with C++* and *Game development with C++*.
3. Install CMake 3.20.5 or later from <https://cmake.org/download/>. During installation, select one of the options that adds CMake to the system PATH.
4. Confirm that CMake is accessible from the command line.

```
PS C:\> cmake --version
cmake version 3.22.2
```

5. Install Git from your favorite location or <https://git-scm.com/download/win>.

6. Confirm that Git is accessible from the command line.

```
PS C:\> git --version
git version 2.35.1.windows.2
```

With these tools installed you are ready to take on O3DE.

## Acquiring O3DE

O3DE is available in two different ways. One source is a direct stand-alone installer and the other is O3DE GitHub repository. I will start by using the installer. You can download the installer from here:

<https://www.o3de.org/download/#windows>

### Note

I present the steps for building your own installations of O3DE from its GitHub repository in Chapter 28, *Wwise for O3DE* and Chapter 31, *Setting Up Multiplayer*.

## Running the Installer

### Note

The instructions here are applicable to PC Windows platform.

Once you download the installer and launch it, you will see its starting window.



O3DE stand-alone installer

You can find where the engine will be installed by looking at *Options*.

## Setup Options

Install location:

C:\O3DE\21.11.2

[Browse](#)

Default installation location

### ⚠ Important

Prior to launching the installer, you have to install CMake first as the installer needs to perform a number of initial steps that uses CMake, such as downloading various dependencies.

Once the installation has completed, confirm that the installed location contains the engine. By default, the location will be C:\O3DE\21.11.2. For the remainder of the book, I will reference this path as the engine path. If you choose to install it elsewhere, adjust the instructions accordingly.

If the engine installed successfully, you will be able to launch *O3DE Project Manager* from the installer by clicking *Launch* button or directly from C:\O3DE\21.11.2\bin\Windows\profile\Default\o3de.exe.

The next step is to create our first O3DE project.

---

# Chapter 2. Creating a New Project

## Introduction

### Note

The source and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch02\\_new\\_project](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch02_new_project)

This chapter will cover:

- How to create a new project.
- Building the project.
- What is Asset Processor?
- Opening a level in O3DE Editor.

### Tip

You can find the official getting started guide at:

<https://www.o3de.org/docs/welcome-guide/create/creating-projects-using-project-manager/>

## O3DE Command Line Interface

Even though one can create a new project using *O3DE Project Manager*, I will show you how to do it using command line tools instead, as I believe it will give you more power in the future. This CLI can be found at C:\O3DE\21.11.2\scripts\o3de.bat

```
PS C:\git\book> C:\O3DE\21.11.2\scripts\o3de.bat -h
usage: o3de.py [-h]
```

Here is the command line to create a new project:

```
o3de.bat create-project -pp C:\git\book\MyProject
```

This will create a new project at C:\git\book\MyProject.

### Note

The parameter **-pp** stands for project path. The script will deduce the project name to be *MyProject*.

There are more options and commands available for o3de.bat but this is enough for us to get started.

Confirm that C:\git\book\MyProject was created and contains various project files.

### Tip

I advise you to put your project under Git source control right away. As a reminder, I am creating the project under c:\git\book folder. If you are following along the book, you can clone the accompanying source code with the following commands:

```
mkdir c:\git\  
cd c:\git  
git clone https://github.com/AMZN-Olex/O3DEBookCode2111 book  
cd book  
git checkout origin/ch02_new_project
```

As you progress through the book, switch to the appropriate branches.

## Linking Projects and O3DE

If you are starting with a fresh O3DE installation and created a new project using the steps above, then the project will be already configured to use the installed O3DE engine at C:\O3DE\21.11.2. However, if you have multiple copies of O3DE installed or you are switching between different O3DE versions, then it is valuable to know how projects declare which engine they use.

There are two important files involved. The first one is *project.json* that is found in your project at C:\git\book\MyProject\project.json. The second one is C:\Users\<user>\.o3de\o3de\_manifest.json.

*project.json* specifies the name of O3DE engine installation to use with *engine* property.

```
{  
    "project_name": "MyProject",  
    ...  
    "engine}
```

*o3de\_manifest.json* specifies all your engine installations and their corresponding names.

```
"engines_path": {  
    "o3de-sdk}
```

If you ever move the engine installation or wish to try out a new engine, you will need to update these entries accordingly.

### Tip

The location of engines and projects is entirely up to you, so long as the two files above agree.

## Building with CMake and Visual Studio

Now that we have a new project, we can configure and build it. The first step is to create a build folder. The location is up to you, but I recommend to keep it short as build errors can show up if the maximum file path is exceeded due to a long build folder path on Windows.

I have chosen C:\git\book\build for the remainder of the book. Once inside the folder, execute the following CMake instruction to configure the project:

```
C:\git\book\build> cmake -S C:\git\book\MyProject\ -B .
```

Parameter **-S** specifies the source location of our project, while **-B** specifies the build folder location. The build folder is set to be the current folder which is C:\git\book\build.

### Tip

One other parameter I often use is **-DLY\_UNITY\_BUILD=OFF**. That turns off CMake unity files that compiles several source files together as one. While that can speed up your first initial compilation, I find that on incremental small changes inside a project, it is faster to work with unity builds disabled.

This step can take a while depending on your Internet download speed. A lot of third party packages will be downloaded and installed into `C:/Users/<user>/ .o3de/3rdParty`.

Once the configuration has completed, Visual Studio solution will be generated at `C:\git\book\build\MyProject.sln`. From this point on, you can either build your project from Visual Studio or from the command line, such as:

```
PS C:\git\book\build> cmake --build . --config profile
```

Project binaries will be placed under `C:\git\book\build\bin\profile`.

## Asset Processor

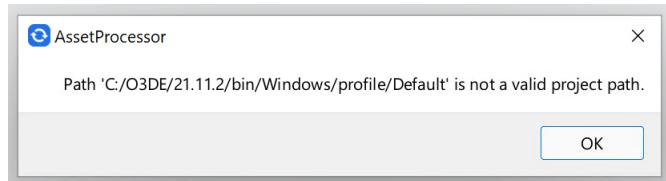
Before we start looking into launching our game project, we need to understand the main four executables involved in designing and running a game in O3DE: the Editor, the Asset Processor, game and server launchers.

1. `Editor.exe` is located inside O3DE's installed binary folder, `C:\O3DE\21.11.2\bin\Windows\profile\Default\Editor.exe`. It is used to design your game and game assets. It is also capable of running the game inside the Editor. It is provided by O3DE installation.
2. `AssetProcessor.exe` is located inside O3DE's installation with `Editor.exe`. Its purpose is to process and provide game assets to the Editor and game launchers.
3. `MyProject.GameLauncher.exe` is located inside project's build folder. It is a stand-alone launcher that runs the game without the Editor. It is compiled by your project's Visual Studio solution.
4. `MyProject.ServerLauncher.exe` is a dedicated server launcher for the project.

Whenever you launch the Editor, the Asset Processor will be launched in the background if it is not running yet. I recommend running the Asset Processor on its own at first to confirm that the project setup is good and all the assets are processing.

```
C:\O3DE\21.11.2\bin\Windows\profile\Default\AssetProcessor.exe
```

Before you launch the Asset Processor you need to configure the default project for O3DE. Otherwise, you will see the following error.



Asset Processor fails without a default project set

The Asset Processor tried to read the default project but did not find one specified in `C:/Users/<user>/ .o3de/o3de_manifest.json`. You should either pass the default project to the Asset Processor

with **--project-path C:\git\book\MyProject** or by setting the default project using O3DE Command Line Interface:

```
C:\O3DE\21.11.2\scripts\o3de.bat set-global-project  
-pp C:\git\book\MyProject
```

## ⓘ Tip

You can confirm that the above operation succeeded by looking at `C:\Users\<user>\.o3de\Registry\bootstrap.setreg`. It should have `project_path` set to `C:\git\book\MyProject`. You can either modify this file directly or use `o3de.bat`.

### Example 2.1. C:\Users\<user>\.o3de\Registry\bootstrap.setreg

```
{  
    "Amazon": {  
        "AzCore": {  
            "Bootstrap": {  
                "project_path": "C:/git/book/MyProject"  
            }  
        }  
    }  
}
```

With this configuration, the Asset Processor can be launched from the command line without specifying the project path.

## ! Important

Only one Asset Processor can be launched for your project. If you attempt to launch another one it will throw an error "Another instance of the Asset Processor may already be running on port 45643". If another instance is already running, then there is no need to launch another one. The Editor and game launchers will connect to the Asset Processor if one is already running.

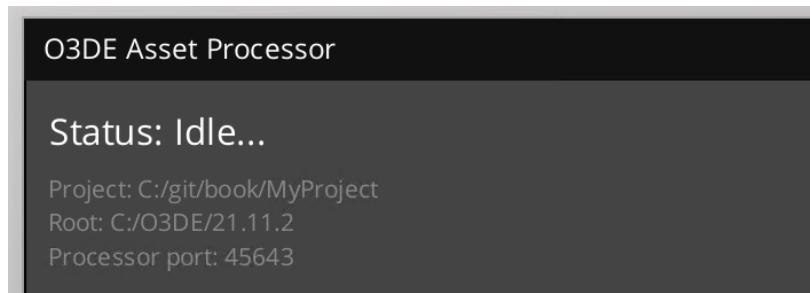
## ⓘ Tip

The Asset Processor can be sneaky. If it is launched by the Editor or a game launcher, it will start in the background, and you will have to find it in the Windows system tray. However, if you launch it directly from the command line, it will open in the foreground.

Once Asset Processor processes all the assets in the top left corner of its window it should show the following:

- **Status: Idle...** - otherwise it is still processing assets and you should wait until all the assets are done.
- **Project: C:/git/book/MyProject** - confirm that it is running for the right project.
- **Root: C:/O3DE/21.11.2** - confirm that the Asset Processor is using the expected engine.

### Example 2.2. The Asset Processor finished processing game assets

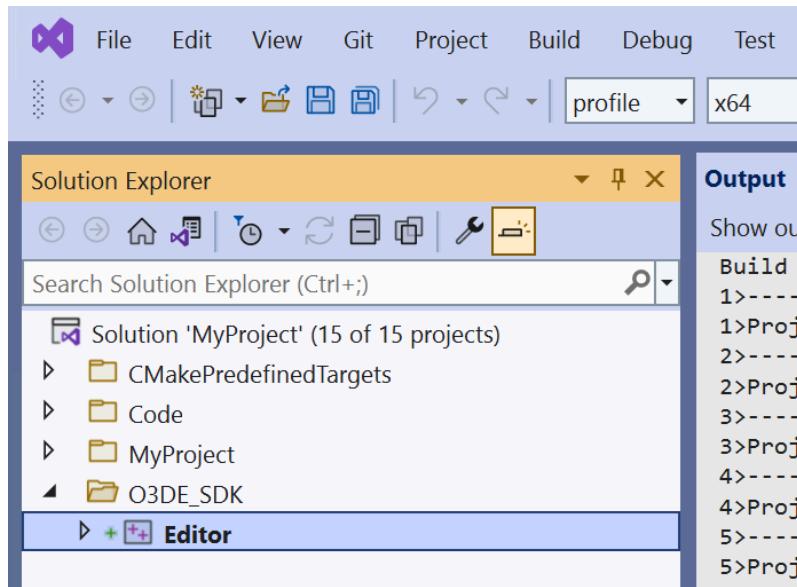


Asset Processor places processed game assets under `C:\git\book\MyProject\Cache` to be used at runtime.

You can launch the Editor with the following command:

```
C:\O3DE\21.11.2\bin\Windows\profile\Default\Editor.exe
```

Or from Visual Studio solution by building and running the *Editor* project that is in *O3DE\_SDK* solution folder.



#### Important

Do not close Asset Processor while working with O3DE. It is needed for many O3DE tools, including the Editor. Instead, minimize the Asset Processor and let it idle in the Windows system tray.

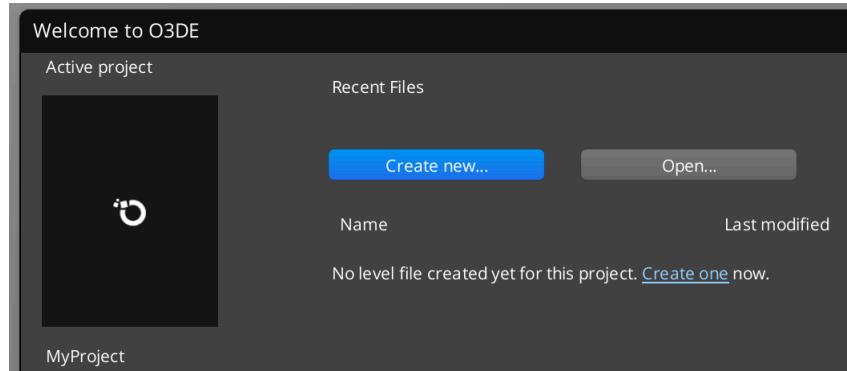
#### Tip

One of most common pitfalls is getting build errors due to the Asset Processor holding a lock on your game binaries. The way to solve that is to stop the Asset Processor every time before re-compiling your project. However, that gets tiresome. When you close the Editor, the Asset Processor is still running in the background. One way to simplify your life is to tell the Asset Processor to shutdown itself whenever the Editor or the last game launcher quits. You can do so by adding this switch to `C:\git\book\MyProject\game.cfg`.

```
ap_tether_lifet ime=1
```

## Create a Level

The new project we created comes without a level, but we can create a new one from an existing O3DE template. Once the Editor launches, you will see a **Welcome to O3DE** dialog. Create a new level using **Create new...** button.



Choose 'Create new...'

In this book, I am going to call it *MyLevel*.

## Summary

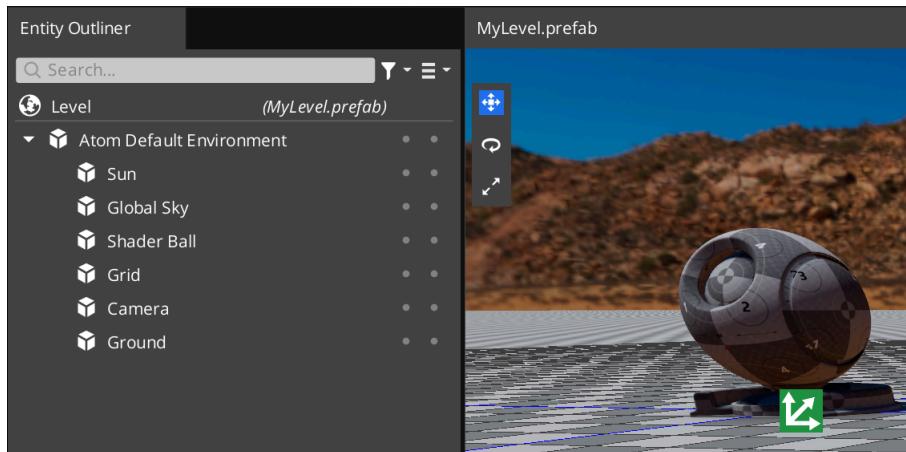
### Note

The source and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch02\\_new\\_project](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch02_new_project)

At this point, you should see the level in the Editor.

**Figure 2.1. Editor: new level**



Our next topic will be Entities. You can see them in **Entity Outliner**. If not visible, you can bring it up from Editor's menu: **Tools**→**Entity Outliner**.

Now we are ready to start looking into the most fundamental aspects of O3DE: *entities* and their *components*. That will be the focus of Chapter 3, *Introduction to Entities and Components*.

---

## **Part II. Entities and Components**

---

# Table of Contents

3. Introduction to Entities and Components .....	15
Introduction .....	15
O3DE Components .....	17
Composing Objects with Entities .....	21
Summary .....	27
4. Writing Your Own Components .....	29
Introduction to Game Project Files .....	29
Creating Your Own Component .....	33

---

# Chapter 3. Introduction to Entities and Components

To exist is to be something, as distinguished from the nothing of nonexistence, it is to be an entity of a specific nature made of specific attributes.

—Ayn Rand

## Introduction

### Note

We are picking up where we left off in Chapter 2, *Creating a New Project* where we created "MyLevel" in the Editor. The assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch03\\_intro\\_to\\_entities](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch03_intro_to_entities)

This chapter covers the following:

- What is an Entity?
- What is a Component?
- Transform component.
- Mesh component.
- Creating composite objects.

In O3DE the game world is constructed out of Entities (AZ::Entity in C++) that may contain a number of Components (AZ::Component). An entity is an independent group of components. Conceptually, a component is a particular attribute of an entity. For example, for an entity to be possess a location in space, it must have a transform component, TransformComponent. Similarly, if you wish to create an entity that is a movable camera, then it must have a transform and a camera component.

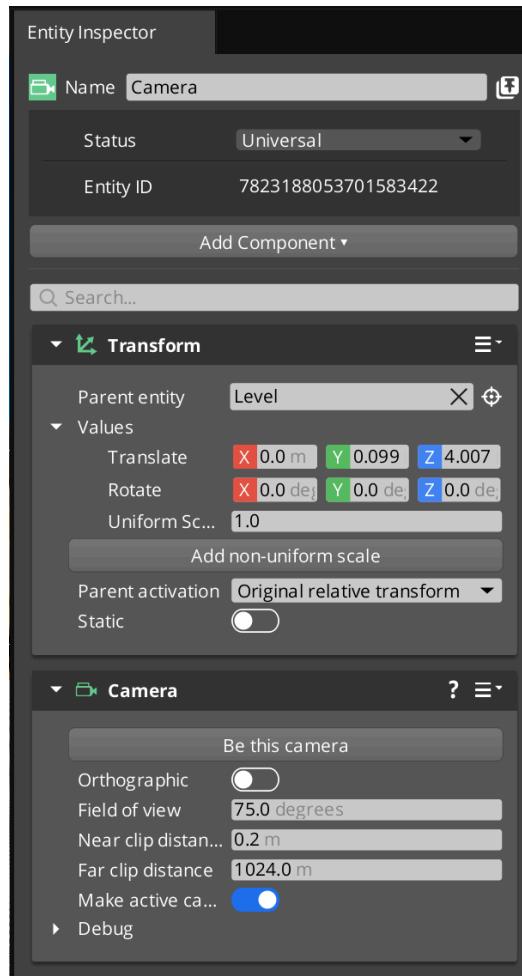
**An Entity by itself is an empty container of components.** It has little to itself aside from the attached components. You can find Entity source header at: C:\O3DE\21.11.2\Code\Framework\AzCore\AzCore\Component\Entity.h.

```
///! An addressable container for a group of components.  
///! An entity creates, initializes, activates,  
///! and deactivates its components.  
///! An entity has an ID and, optionally, a name.  
class Entity
```

An important aspect of entities is how light they are. All they really have is their identifier, AZ::EntityId. The rest of their interface has to do with interacting and managing the components that an entity has. All the behavior of an entity comes from the components it has. The components describe behavior and attributes of an entity. You are free to compose entities out of any combination of components.

Here is how it would look like in O3DE Editor.

**Figure 3.1. An Entity with Transform and Camera components**



Here, we are viewing an entity in **Entity Inspector** that has **Transform** and **Camera** components attached.

### Note

You can find the official reference on entities and components at:

<https://www.o3de.org/docs/welcome-guide/key-concepts/#the-component-entity-system>

You can find the header for `AZ::Component` in `C:\O3DE\21.11.2\Code\Framework\AzCore\AzCore\Component\Component.h`.

```
/**  
 * Base class for all components.  
 */  
class Component
```

Before we start digging into C++ code and interface of `AZ::Component`, let us go through the workflow and the common way of using entities and components to build game objects in the Editor.

# O3DE Components

A great way to get an idea about components is by taking a look at components that O3DE comes with. The first few components you are bound to run into are `TransformComponent` and `MeshComponent`.

## Note

A great starting guide to O3DE Editor on how to create entities and components can be found at <https://www.o3de.org/docs/learning-guide/tutorials/first-project/>.

You should go through the tutorial and tinker on your own in the Editor to get a sense for the interface. The rest of the chapter will go through the workflow of adding and modifying entities.

## Transform Component

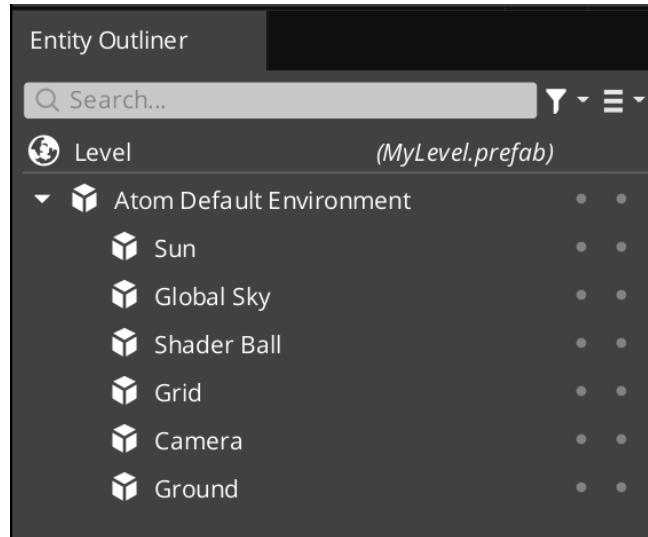
All entities created inside the Editor get a **Transform** component automatically added to it.

**Why does an entity get a Transform component by default?** If you were to create an Entity at runtime in game code, then the entity would be entirely empty. It will not even have a Transform component unless you add it yourself at runtime. However, when you create an entity in the Editor that implicitly places the entity somewhere in the level. Somewhere implies a location, therefore it must have a **Transform** component to be somewhere on the level.

## Tip

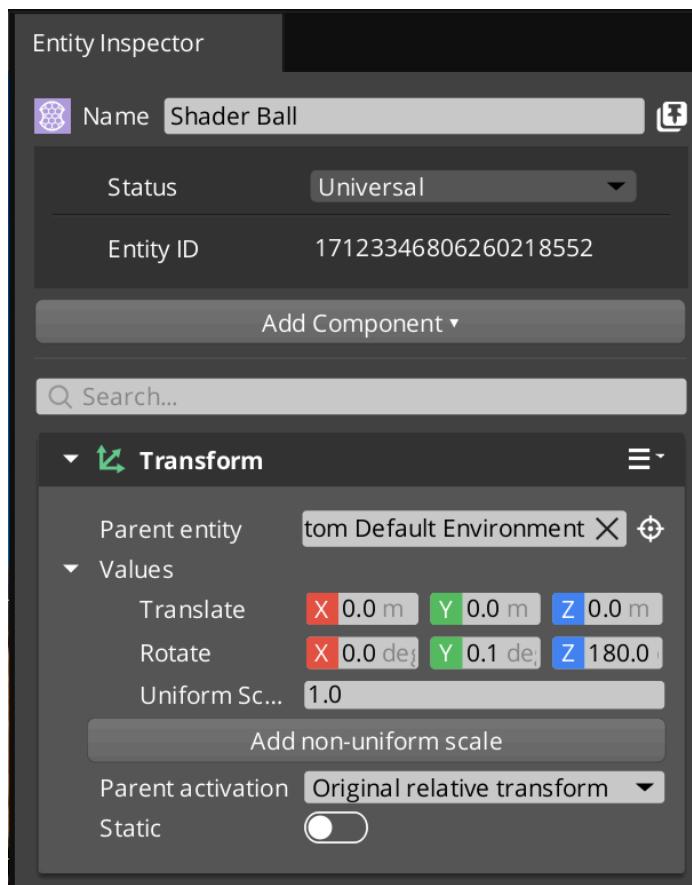
One exception to that rule is a special Level entity and its Level components. We take a look at them in Chapter 6, *What is AZ::Interface?*

**Figure 3.2. Editor: Entity Outliner**



"MyLevel" contains a few entities already. Here is one, **Shader Ball**, viewed in **Entity Inspector**.

**Figure 3.3. Editor: A sphere Entity in "Simple" level**



### Note

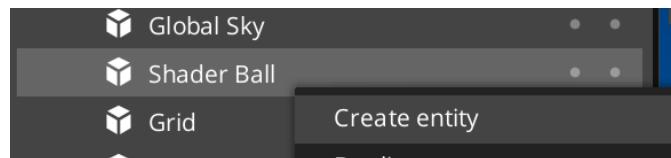
The official reference for **Transform** component can be found at: <https://www.o3de.org/docs/user-guide/components/reference/transform/> alongside with many other components.

**What is "Parent entity" field?** Components can introduce their own concepts and relationships. This field associates one entity to another entity in terms of their spatial relationship in the world. A child's position, orientation and scale will depend on the values of its parent entity.

## "Child" Entities

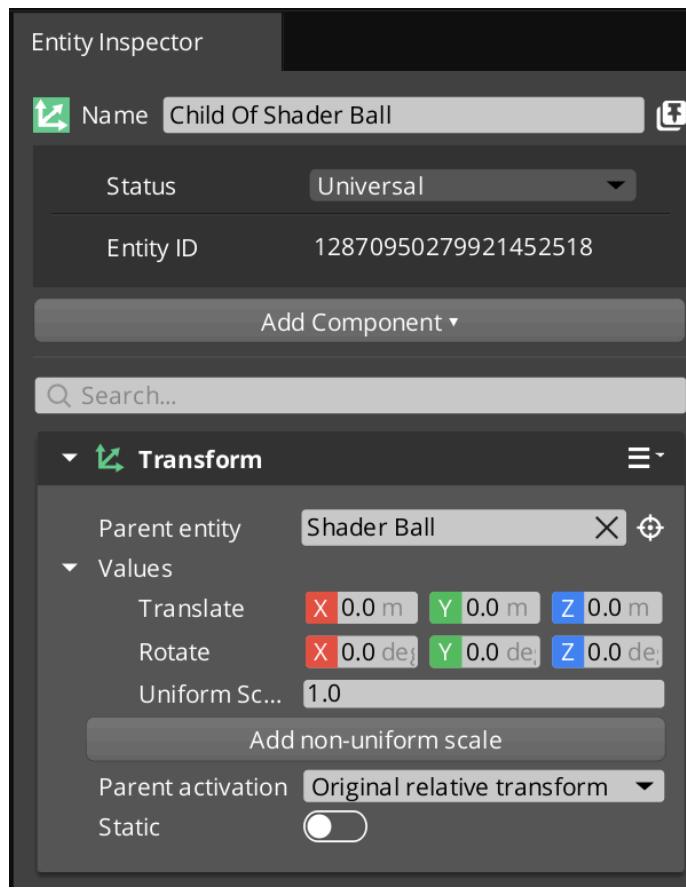
You can create a "child" Entity in **Entity Outliner** by right clicking on an existing entity and choosing **Create entity**. This is how it would look like in the Editor.

**Figure 3.4. Editor: Create (child) entity**



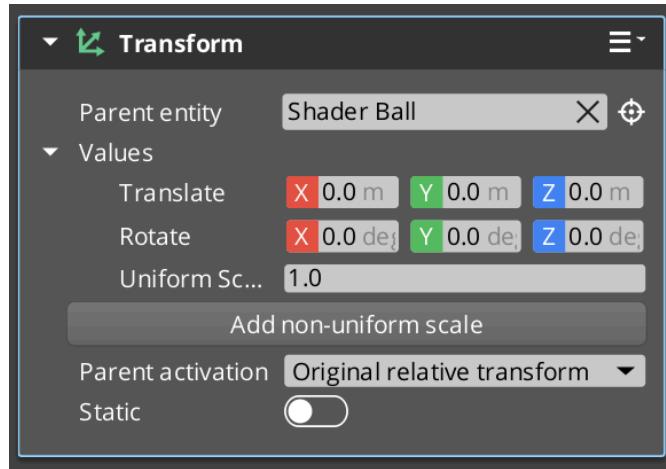
After the entity is created, select it in *Entity Outliner*, and rename to "Child of Shader Ball" in *Entity Inspector*.

Figure 3.5. Editor: child entity



The entity name is optional but it reminds us here that this entity is the child of *Shader Ball* entity, as expressed in the property *Parent entity* of Transform component.

Figure 3.6. Entity with Child Transform



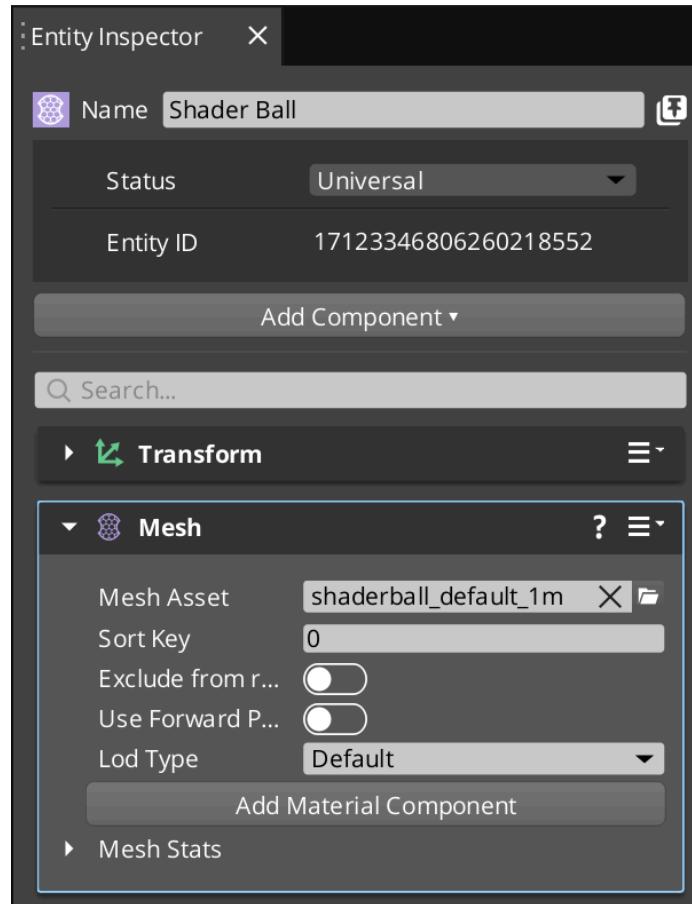
**Parent entity** points to **Shader Ball**. That means the values of **Translate**, **Rotate** and **Uniform Scale** become local values that depend on the parent's values, which means **Shader Ball**'s Transform component values.

The next common component we are going to look at is **Mesh** component.

## Mesh Component

A **Mesh** component allows you to add a graphical object to an entity. You can add it to an entity using **Add Component** menu in **Entity Inspector**.

**Figure 3.7. Entity with Mesh component**



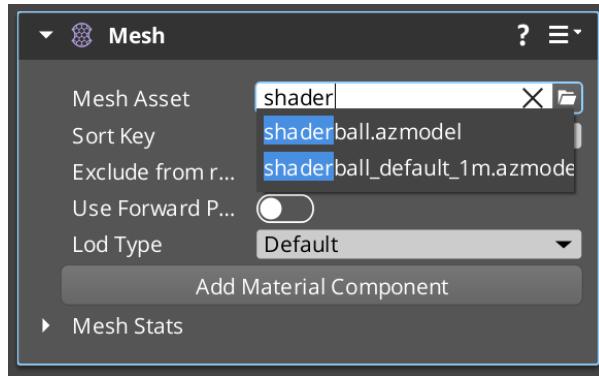
### Tip

I have collapsed **Transform** component to keep it out of the way but it is still there. You can toggle the collapsed state using the little triangle next to the name of the component.

In this case, **Mesh asset** has already been set, but you can always change it using a dialog that open when you click on folder button to the right of *Mesh Asset* property.

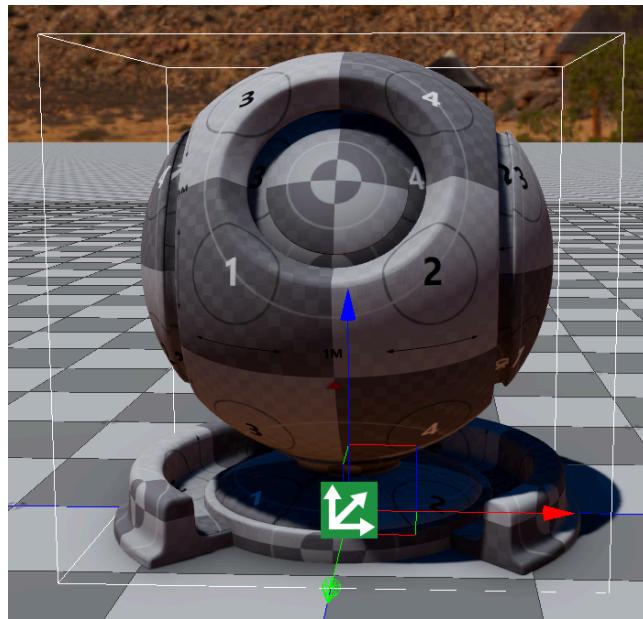
Another way to specify mesh asset is by typing directly in *Mesh Asset* text field. As you type a drop down list will appear the matches your text entry.

**Figure 3.8. Selecting Mesh Asset via Interactive Search**



**Add Material Component** button allows you to assign a custom material for the mesh. In this case, the entity does not have a material component, so a default material of the mesh is used.

**Figure 3.9. Editor: \_Sphere\_1x1 entity**

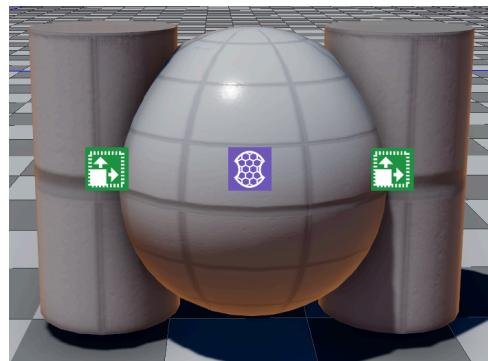


**Notice that Mesh component does not have any offset or position properties.** Mesh component does not provide any properties to specify how the mesh is to be positioned. Implicitly, the mesh is placed at the center of the entity, the local position of (0, 0, 0). Imagine for a moment that the mesh asset was centered at some non-zero local point. Then we would need to do some additional work to center it properly. We will tackle that in the next section.

## Composing Objects with Entities

We have already seen that we can relate one entity to another with a Transform component. In some cases, a single independent entity will do its job but in practice you often need to create complicated composite objects.

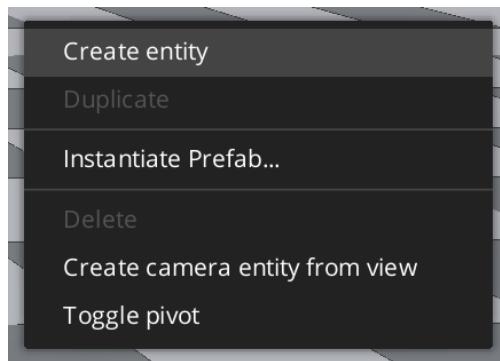
**Figure 3.10. Editor: Composite Object**



We will start with an empty "root" entity that is going to be the parent for all other entities we will create to represent the object. This is often useful, so that we can move the entire group together in the Editor by moving the "root" entity. It is still an entity like any other but Transform component parent entity field will keep them all together.

An entity can be created in the Editor by right clicking in the empty space in the main viewport and selecting **Create entity**.

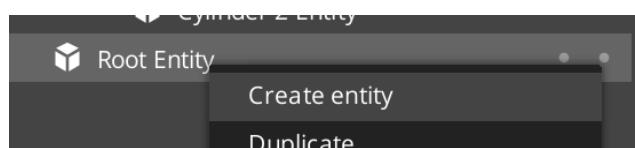
**Figure 3.11. Editor: Creating an Entity**



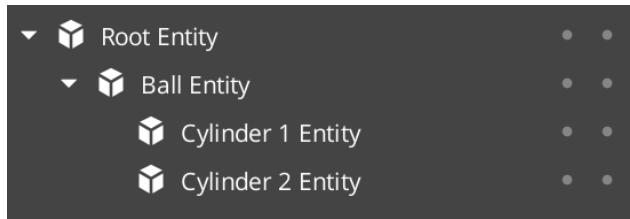
The Editor will assign some auto-generated name for it, such as **Entity11**, but we can rename it in **Entity Inspector**. I'll choose **Root Entity**.

Create a child entity of **Root Entity** for the ball in the middle of the composite object.

**Figure 3.12. Editor: Creating a child Entity**

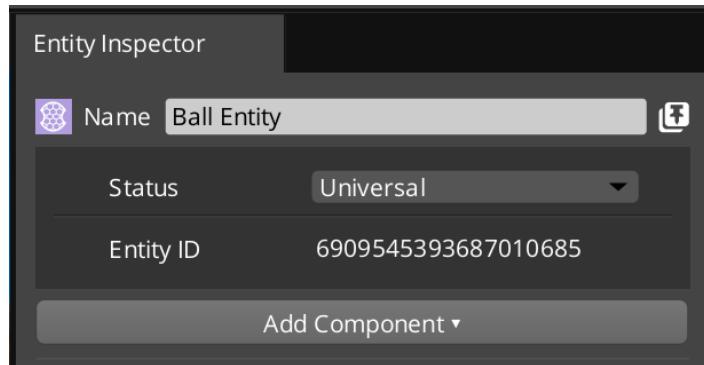


Create two cylinder meshes by creating two more child entities. This time the parent will be **Ball Entity**. Name child entities **Cylinder 1 Entity** and **Cylinder 2 Entity**. The outline will show their structure.



Structure of the entities

Now we can assign various mesh components to ball and cylinder entities. Here is how you can add a mesh component to **Ball Entity**. Select **Ball Entity**. In **Entity Inspector** click on **Add Component**.

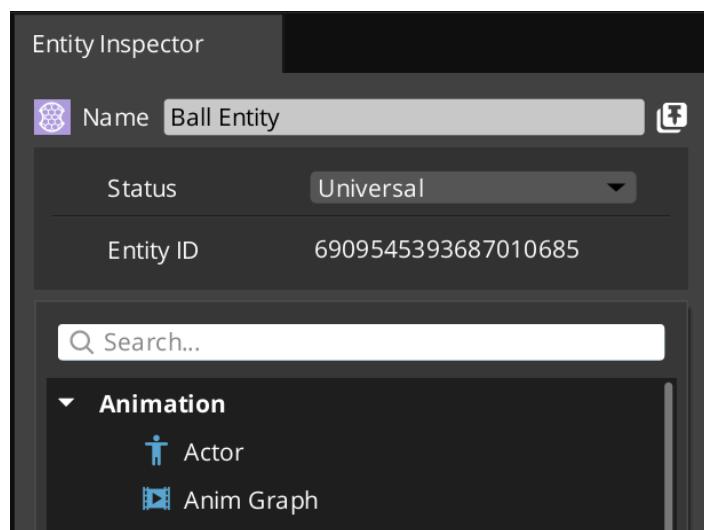


Add Component menu

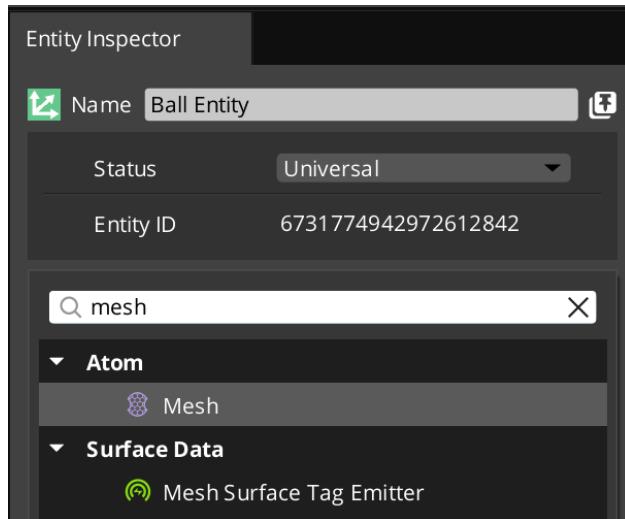
That will list all available components given your current project configuration.

### Tip

I find it easier to enter the name of the component in the **Search...** field to quickly find the component I am looking to add.



Search field when adding a component

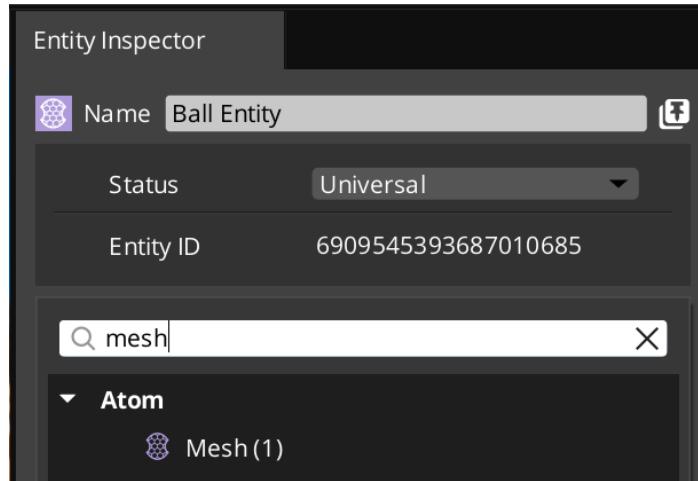


Looking for Mesh component

**Atom/Mesh** is the component we want. Clicking on it will add it to the entity. And then we can specify **Mesh asset** to be **\_Sphere\_1x1** just like the other entities we have looked at earlier.

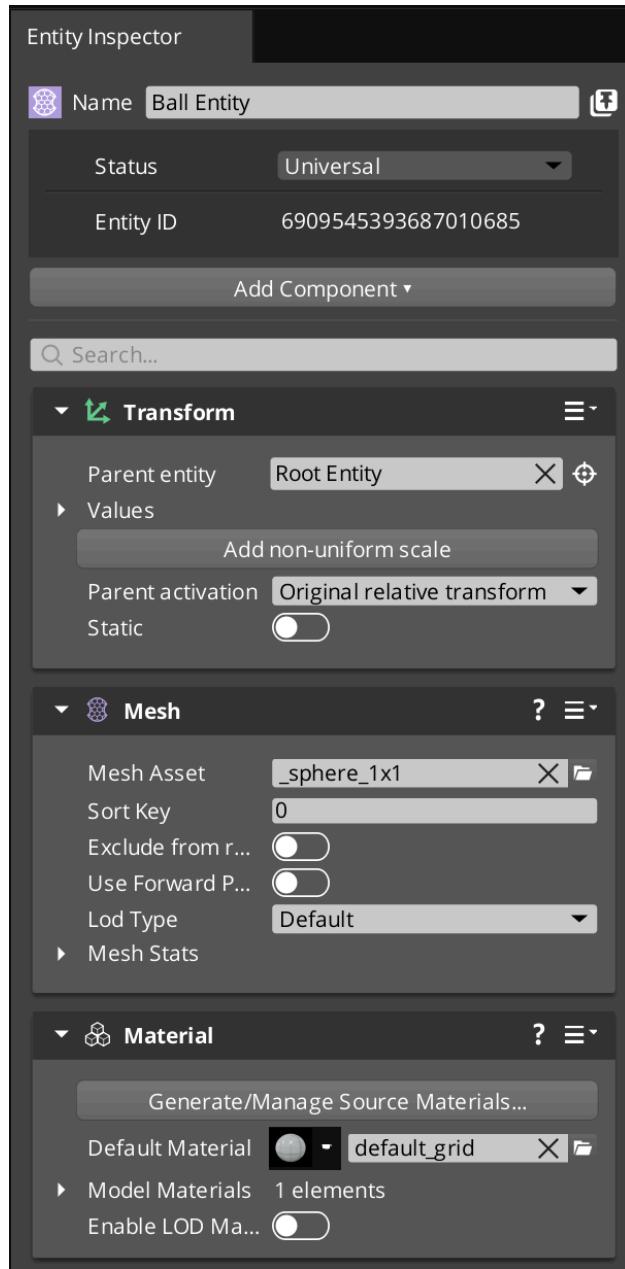
### Note

Not all components can be added more than once to the same entity. You can specify that restriction yourself when you write your components with static C++ method `GetIncompatibleServices`, which is covered in Chapter 9, *Using AZ::TickBus*. Here is how **Add Component** menu would look like if you were considering adding a second **Mesh** component to the same entity.



The Editor automatically adds **(1)** to the name of the component. That indicates that there is already a **Mesh** component on your entity. If a component was configured to limit itself to a single instance per entity, then you would no longer be able to find it in **Add Component** menu.

Here is completed **Ball Entity**.



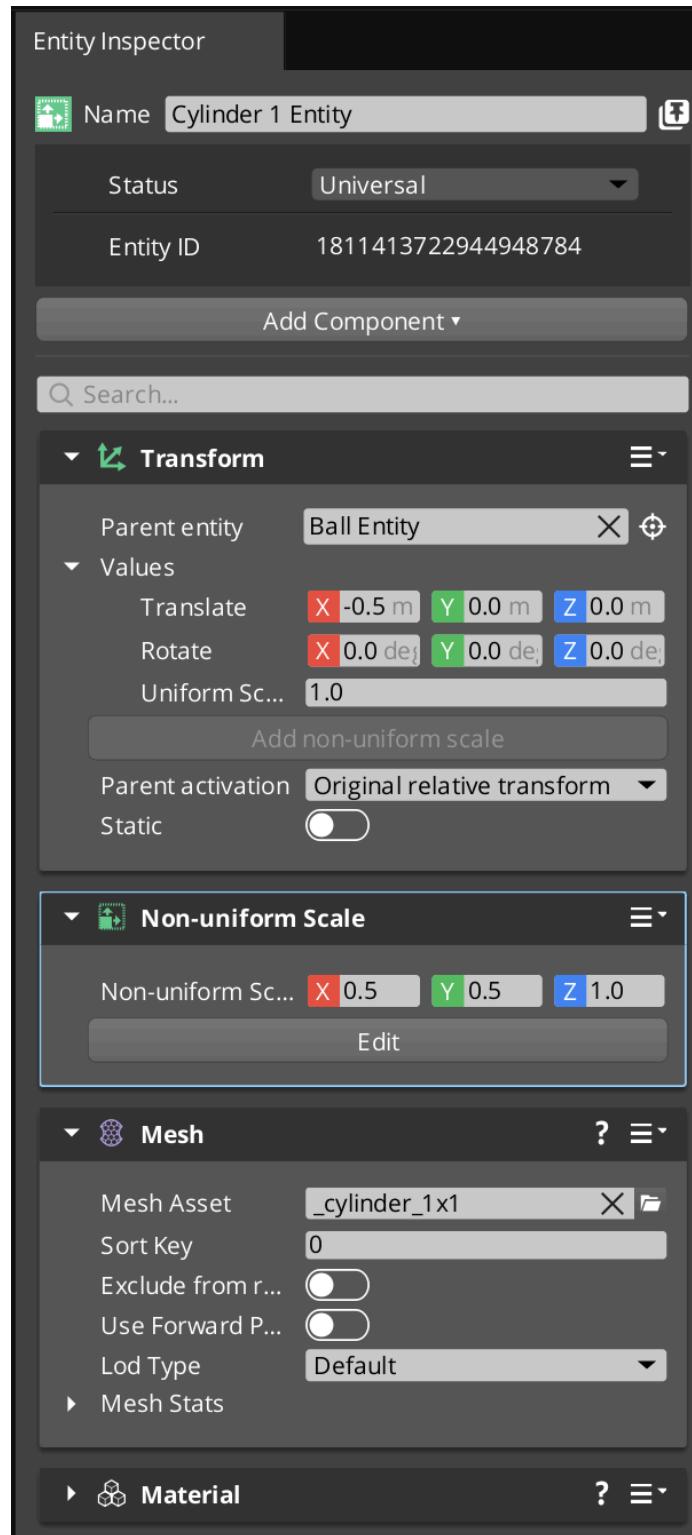
Entity with Mesh component

Cylinder entities follow the same structure but with a different **Mesh asset** and custom values in Transform component in order to move them to the side and change their scale in X and Y dimensions. Specifically, **Ball Entity** has a Translate value of (0, 0, 0) which is the center of the object. But cylinders' **Translate** values are (0.5, 0, 0) and (-0.5, 0, 0).

### Note

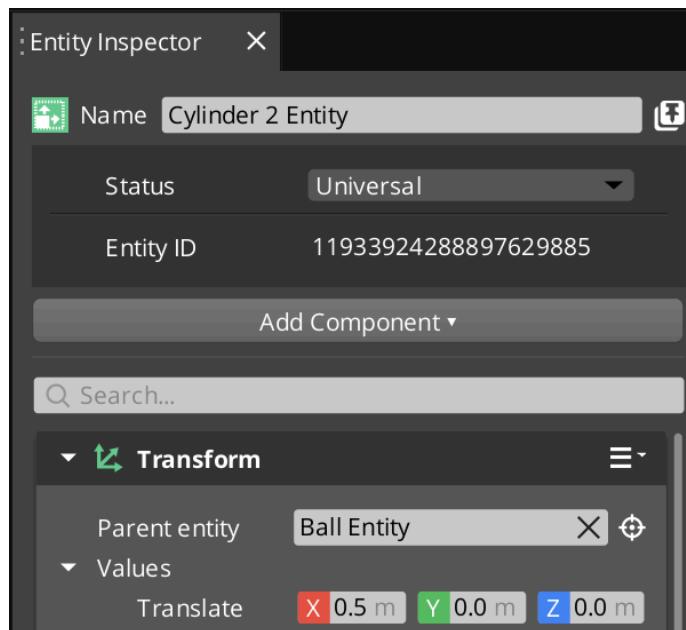
**Non-uniform Scale** is a special component that allows you to configure a non-uniform scale. Otherwise the entity is scaled uniformly in all directions.

Figure 3.13. Cylinder 1 Entity



An entity with a cylinder mesh

Figure 3.14. Cylinder 2 Entity



## Summary

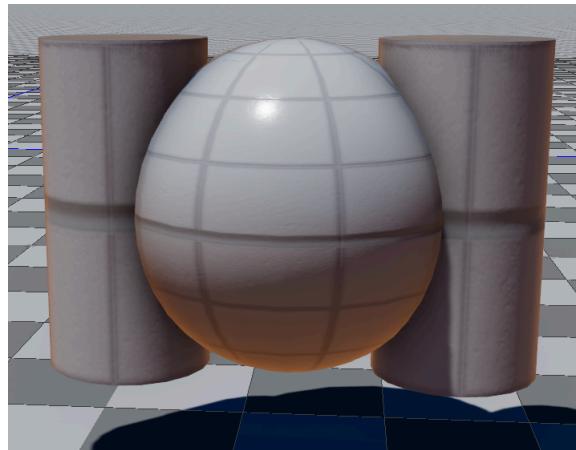
### Note

The assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch03\\_intro\\_to\\_entities](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch03_intro_to_entities)

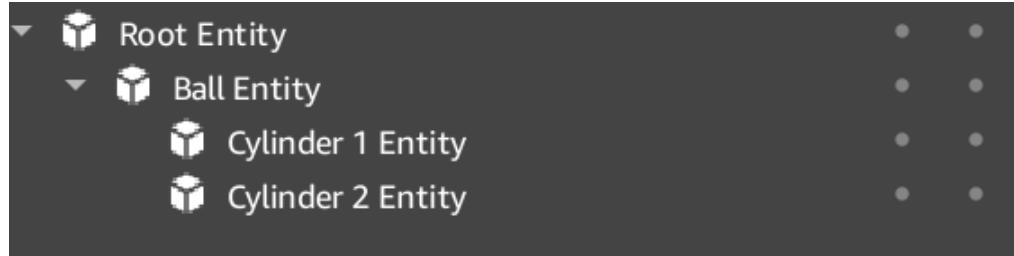
You can move the object by its **Root Entity** into the view of the camera entity. Once you are done with the changes, you need to save the level. The shortcut for that is **CTRL+S** or from the Editor's main menu **File → Save**.

You can then press **CTRL+G** to test the level in the editor. The composite object we have made in this chapter would look like this:



An object composed of multiple entities parented together

This chapter looked at the general idea of entities and components in O3DE. The real power lies in creating your components. In order to create custom components, one has to write them in C++ code. The next chapter will start by looking at the overall structure of a O3DE C++ project and how brand new components can be written from scratch in C++.



New entities we created in this chapter

---

# Chapter 4. Writing Your Own Components

Official reference how to create O3DE game projects:

<https://www.o3de.org/docs/user-guide/project-config/project-manager/>

## Introduction to Game Project Files

We created a new project in Chapter 2, *Creating a New Project*. In this chapter we are going to take a look at the files involved and add a new component to the project.

### Note

In later chapters, we will re-visit the structure of a game project in O3DE once we go over more concepts, such as EBus, gems, prefabs, etc. For now, I am going to cover just the basics to get us started writing new components.

This chapter covers the following topics:

- Basic project structure.
- Adding a new C++ component.
- Finding the new C++ component in the Editor.

## Basic Structure of a New Project

### Note

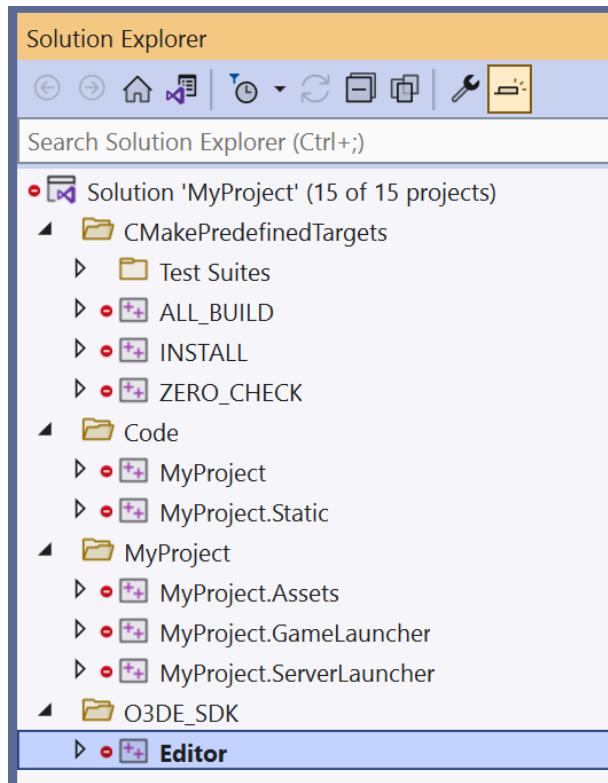
The source code for this project can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch04\\_creating\\_components](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch04_creating_components)

Previously we created a new project and placed it at `C:\git\book\MyProject`. We generated the build folder at `C:\git\book\build` that contains the Visual Studio solution `MyProject.sln`.

There are a lot of projects and sub-folders in here but we are going to focus on the essentials.

- **ZERO\_CHECK** is a special CMake target that rebuilds Visual Studio from any changes to CMake build files. When we add a new component we will need to build this project and re-load the solution with the updates.
- **Editor** is the target that runs O3DE Editor with our project. It does not actually build any of the Editor code. The Editor binary is provided by the O3DE installation in `C:\O3DE\21.11.2`.
- **MyProject.Static** is the static library for adding new components.

**Figure 4.1. MyProject solution in Visual Studio**

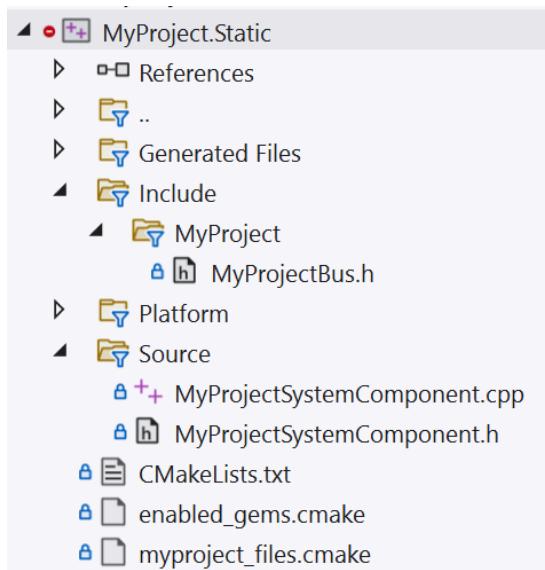
- **MyProject** is the dynamic library that links **MyProject.Static** and provides the component information to the Editor and game launcher. In other words, **MyProject.Static** defines the components and **MyProject** registers them for your project.

At the moment the only component we have in the project is the default system component.

- C:\git\book\MyProject\Code\Source\MyProjectSystemComponent.h
- C:\git\book\MyProject\Code\Source\MyProjectSystemComponent.cpp

What are the elements of **MyProject.Static** library?

- **Include/MyProject** contains various public interfaces. I will cover various options available in O3DE in later chapters when we tackle communication between components, starting with Chapter 5, *What is FindComponent?*
- **Source** contains source and header files of the components and any other classes and objects you might write.
- **CMakeLists.txt** is the build script where **MyProject.Static** is defined.
- **enabled\_gems.cmake** deals with the gem system that we will tackle in Chapter 12, *What is a Gem?*
- **myproject\_files.cmake** defines the list of source and other files to be included in the build and in Visual Studio solution.

**Figure 4.2. MyProject.Static library**

Generally, any new component you would write would be placed under `MyProject\Code\Source`, either directly in `Source` folder or under a sub-folder of your choice.

If you were to search for references to `MyProjectSystemComponent` under `C:\git\book\MyProject`, you would find two more files where it is referenced.

## **myproject\_files.cmake**

`*_files.cmake` are O3DE project files that list the files to be included and compiled for a project.

### **Example 4.1. MyProject\Code\myproject\_files.cmake**

```
set(FILES
    Include/MyProject/MyProjectBus.h
    Source/MyProjectSystemComponent.cpp
    Source/MyProjectSystemComponent.h
    enabled_gems.cmake
)
```

#### **Tip**

You can also include files other than the source code, such as various configuration files, shaders files, if you wish to have easier access them inside Visual Studio. The build system will exclude them from compilation.

You can see references to source and header files of `MyProjectSystemComponent`. File paths must be local to the project's Code folder: `C:\git\book\MyProject\Code`.

#### **Important**

In order to add new files to the build, you must reference them in this file.

## >Note

O3DE uses CMake as its build system. This chapter will only cover the essentials of CMake in order to add a new component. Each following chapter will introduce just enough of CMake knowledge to accomplish each task. As you progress through the chapters you will acquire all the necessary basics to feel comfortable with O3DE's use of CMake.

### MyProjectModule.cpp

And the other reference to `MyProjectSystemComponent` is in the module file. A module source file is the root file of a project that declares the project and, most importantly for this chapter, registers all the components.

```
MyProjectModule()
    : AZ::Module()
{
    // Push results of [MyComponent]::CreateDescriptor()
    // into m_descriptors here.
    m_descriptors.insert(m_descriptors.end(), {
        MyProjectSystemComponent::CreateDescriptor(),
    });
}
```

Any components added to `m_descriptors` in the code snippet above would be available for use in your project and, if properly configured, in the Editor. Here is the entire module file for reference.

#### Example 4.2. A default module file for a new project, `MyProjectModule.cpp`

```
#include <AzCore/Memory/SystemAllocator.h>
#include <AzCore/Module/Module.h>

#include "MyProjectSystemComponent.h"

namespace MyProject
{
    class MyProjectModule
        : public AZ::Module
    {
public:
    AZ_RTTI(MyProjectModule,
        "{4b67609b-b22c-4a71-9afe-3f9d10bcf5ac}", AZ::Module);
    AZ_CLASS_ALLOCATOR(MyProjectModule, AZ::SystemAllocator, 0);

    MyProjectModule()
        : AZ::Module()
    {
        m_descriptors.insert(m_descriptors.end(), {
            MyProjectSystemComponent::CreateDescriptor(),
        });
    }

    /**
     * Add required SystemComponents to the SystemEntity.

```

```
*/  
AZ::ComponentTypeList  
GetRequiredSystemComponents() const override  
{  
    return AZ::ComponentTypeList{  
        azrtti_typeid<MyProjectSystemComponent>(),  
    };  
}  
};  
}// namespace MyProject  
  
AZ_DECLARE_MODULE_CLASS(Gem_MyProject, MyProject::MyProjectModule)
```

### Note

Note that `GetRequiredSystemComponents()` is a special method to register components that are marked as *system* components. System components are placed on a special system entity that is activated as soon as the engine starts and persists outside of game levels until shutdown.

## Creating Your Own Component

### Note

The official reference for creating a C++ components can be found at:

<https://www.o3de.org/docs/user-guide/programming/components/create-component/>

You cannot create your own custom `AZ::Entity` in O3DE but you can create your own custom component derived from `AZ::Component`. This chapter will show how to create a simple component that shows up in the Editor. We will create a dummy component called, `MyComponent`. In general, whenever you add a new component to a project, you have to do the following steps:

- Create its header file `MyComponent.h` at `MyProject\Gem\Code\Source`.
- Create its source file `MyComponent.cpp` in the same folder.
- Update `myproject_files.cmake` with references to the above two new files.
- Register `MyComponent` in `MyProjectModule.cpp` in `MyProjectModule`'s constructor.
- Build the project.

I will now go through each step.

### `MyComponent.h`

The most basic and simplest component that does nothing aside from existing is as follows:

#### **Example 4.3. The simplest component**

```
#pragma once  
#include <AzCore/Component/Component.h>
```

```
namespace MyProject
{
    // An example of the simplest O3DE component
    class MyComponent : public AZ::Component
    {
        public:
            AZ_COMPONENT(MyComponent,
                "{4b589f6b-79f3-47b6-b730-aad0871d5f8f}");

            // AZ::Component overrides
            void Activate() override {}
            void Deactivate() override {}

            // Provide runtime reflection, if any
            static void Reflect(AZ::ReflectContext* reflection);
    };
}
```

**MyComponent** must derive from **AZ::Component**. All O3DE game components must be derived from **AZ::Component** either directly or through another class that derives from it.

**Macro AZ\_COMPONENT.** This macro marks the component type with a unique guid string. The first parameter is the class name. The second parameter is a unique identifier for this class in a guid form. You are responsible for creating a unique guid with curly brackets around it.

## Note

Visual Studio IDE has a built-in guid generator in the menu: **Tools → Create GUID**. You can also generate a guid using Visual Code plugins or even a PowerShell command:

```
PS C:\work\book\MyProject> (New-Guid).ToString("B")
{dcbbe9e2-5630-4e4b-ad7a-308c7abe5abf}
```

**AZ::Component interface implementation.** `Activate()` and `Deactivate()` are pure virtual methods from `AZ::Component`. They do not have to perform any work but they must be overridden. `AZ::Component` is defined in `C:\O3DE\21.11.2\Code\Framework\AzCore\AzCore\Component\Component.h`. We will discuss these methods later, for now they are not necessary to have `MyComponent` show up in the Editor.

**Reflect() method:** provides a runtime serialization of this component to the O3DE engine. In the next section, I will show the minimum work involved in reflecting a component to the Editor.

## MyComponent.cpp

This is where the description and the behavior of the component will be. However, for now we are going to only provide a simple stub.

### Example 4.4. The simplest MyComponent.cpp

```
#include "MyComponent.h"
#include <AzCore/Serialization/EditContext.h>

using namespace MyProject;
```

```
void MyComponent::Reflect(AZ::ReflectContext* reflection)
{
    AZ_UNUSED(reflection); // TODO provide reflection
}
```

## Tip

**AZ\_UNUSED** is macro that does nothing but pretends to use a parameter in order to silence some compiler warnings. In modern C++ you can also use `[ [maybe_unused] ]` on the parameter declaration.

This is enough to compile the project. Once we have everything in place, we will return to Reflect method. It will be useful to try different implementation of Reflect and see the consequences. So let us finish the rest of the C++ code changes that we need.

## myproject\_files.cmake

The file list for the project is located at `C:\git\book\MyProject\Code\myproject_files.cmake`. Since we only added one component, the following changes are enough:

```
set(FILES
    Include/MyProject/MyProjectBus.h
    Source/MyProjectSystemComponent.cpp
    Source/MyProjectSystemComponent.h
    enabled_gems.cmake

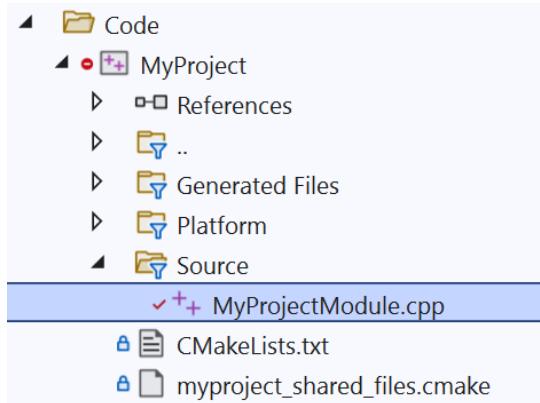
    Source/MyComponent.cpp # new
    Source/MyComponent.h    # new
)
```

## MyProjectModule.cpp

And lastly, the project needs to register the new component by adding its descriptor to the project module.

```
...
#include "MyComponent.h"
...
MyProjectModule()
: AZ::Module()
{
    m_descriptors.insert(m_descriptors.end(), {
        ...
        MyComponent::CreateDescriptor(),
    });
}
```

You can find this module file under *MyProject* build target in Visual Studio.

**Figure 4.3. MyProjectModule.cpp in Visual Studio**

`CreateDescriptor` is a method of `AZ::Component`. For simple tasks, you do not need to worry about what it does at this point. Just be aware that it provides a description of the component to the engine.

## Re-compile the Project

Now that we have gone over the basic C++ changes for a new empty component, let us discuss the workflow involved in updating the project.

### Tip

Close the Editor and the Asset Processor if they are running. Otherwise, the Asset Processor or the Editor might hold a lock on one of your project binaries. In such a case you will get a build error:

```
9>LINK : fatal error LNK1104: cannot open file  
'C:\git\book\build\bin\profile\MyProject.dll'
```

We have two ways of compiling the project on Windows: either Visual Studio solution or through the CMake command line interface.

## Building from Command Line

In some ways, this is the easiest method to compile the project. The benefit of building from a command line is that it allows you to use any editor you wish. Execute:

### Example 4.5. Building the project from command line

```
cmake --build C:\git\book\build\ --config profile
```

### Note

"`--build`" is a switch that tells CMake where the binary folder is for our project. Early on we created it at `C:\git\book\build`. `--config profile` tells CMake to build profile flavor of our project.

### Tip

If you ever need to debug your components, instead of compiling the project in debug build, you can add the following lines instead and still use profile binaries.

```
#pragma optimize( "", off )
/* unoptimized code section */
#pragma optimize( "", on )
```

For more details see your compiler documentation, for example:

<https://docs.microsoft.com/en-us/cpp/preprocessor/optimize?view=msvc-170>

## Building from Visual Studio

A more user-friendly way is to use Visual Studio.

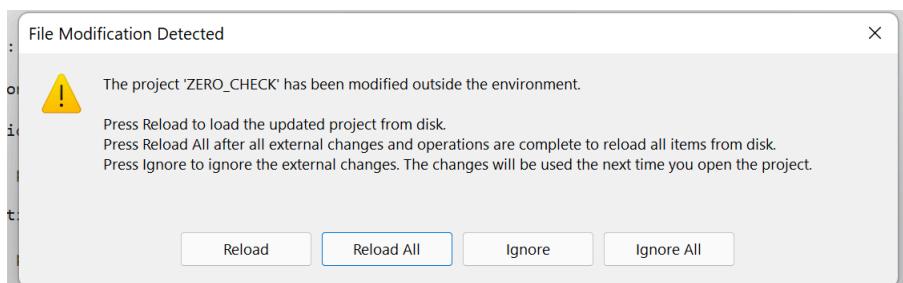
Our Visual Studio solution is located the build folder: C:\git\book\build\MyProject.sln. In this chapter we modified one of the build files, myproject\_files.cmake, and as you compile the solution, you will see the following build log.

```
Build started...
1>----- Build started: Project: ZERO_CHECK
1>Checking Build System
1>CMake is re-running because ...
1>  the file 'C:/git/book/MyProject/Code/myproject_files.cmake'
1>  is newer than ...
```

### Tip

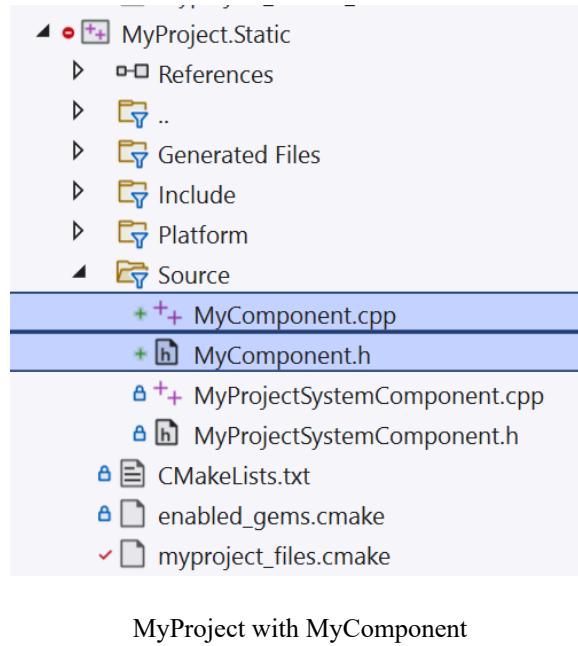
CMake detects changes based on file modification time. *ZERO\_CHECK* should only run when one of the build files have changed in your project. If you are seeing it run on every build, then inspect the build log and see which file CMake thinks has changed since the last build.

Once the build completes, Visual Studio will ask you if you want to reload the solution from disk, select **Reload All**.



Reload VS solution

Here is how the project now looks like in Visual Studio with MyComponent files.

**Figure 4.4. Visual Studio solution with MyComponent**

## Summary

If you were to re-launch the Editor, you would not find the component in **Add Component** menu in *Entity Inspector*. This is because `MyComponent::Reflect()` was left empty, therefore it did not tell the Editor about `MyComponent`. Let us change that to finish up this chapter.

### Note

Reflecting a component to the Editor is a design choice. Not all components need to be visible in the Editor. We will see various reasons for both sides in later chapters.

### Example 4.6. `MyComponent.cpp` with Editor reflection

```
#include "MyComponent.h"
#include <AzCore/Serialization/EditContext.h>

using namespace MyProject;

void MyComponent::Reflect(AZ::ReflectContext* reflection)
{
    auto sc = azrtti_cast<AZ::SerializeContext*>(reflection);
    if (!sc) return;

    sc->Class<MyComponent, Component>()
        ->Version(1);

    AZ::EditContext* ec = sc->GetEditContext();
    if (!ec) return;

    using namespace AZ::Edit::Attributes;
```

```
// reflection of this component for O3DE Editor
ec->Class<MyComponent>("My Component", "[my description]")
    ->ClassElement(AZ::Edit::ClassElements::EditorData, "")
        ->Attribute(AppearsInAddComponentMenu, AZ_CRC("Game"))
        ->Attribute(Category, "My Project");
}
```

Reflection in O3DE is broken down into several parts. For now, it is enough to know that the above code describes the following:

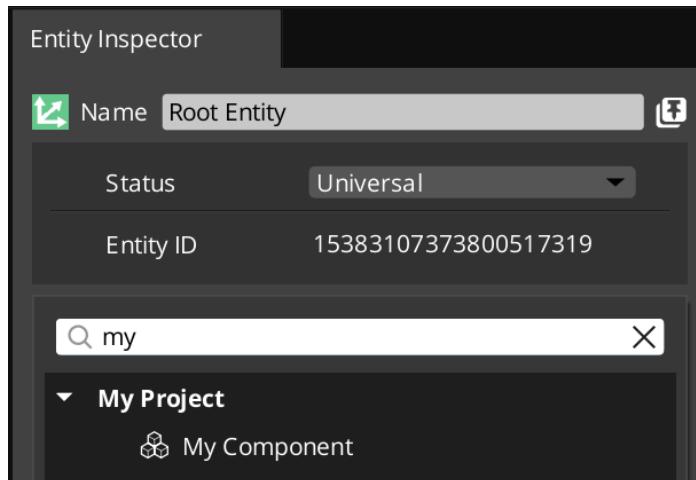
- MyComponent is an empty component with no properties and its version is one (1).

```
sc->Class<MyComponent, Component>()
    ->Version(1);
```

- MyComponent is a component that is available for use in game (and in the Editor) with name "My Component" under the category "My Project".

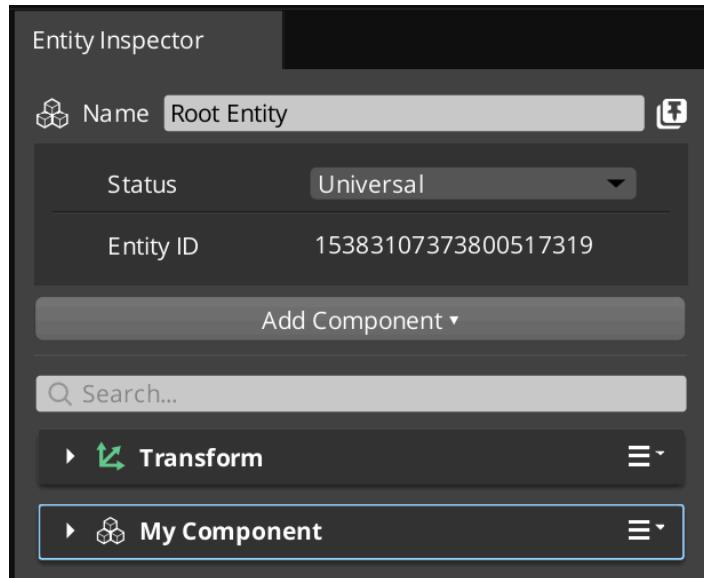
```
ec->Class<MyComponent>("My Component", "[my description]")
    ->ClassElement(AZ::Edit::ClassElements::EditorData, "")
        ->Attribute(AppearsInAddComponentMenu, AZ_CRC("Game"))
        ->Attribute(Category, "My Project");
```

Now **Add Component** menu in the Editor will list this component.



MyComponent in the Editor

And you can add it to any entity, for example to **Root Entity** of the object we built in Chapter 3, *Introduction to Entities and Components*.



## >Note

The source code for this project can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch04\\_creating\\_components](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch04_creating_components)

---

# **Part III. Introduction to Component Communication**

Now that we know how to create components, we need to learn how to build logic out of multiple components. The next milestone that will allow you to do that is some form of inter-component communication. You need various components talking to other components.

In O3DE there are several ways for components to communicate:

- A direct access via `FindComponent Entity` method.
- `AZ::Interface` - a singleton-like communication.
- `AZ::Event` - a notification mechanism.
- `AZ::EBus` - the most generalized and powerful form of communication in O3DE.

---

# Table of Contents

5. What is FindComponent? .....	43
Introduction .....	43
An Example .....	43
Adding a Library Reference .....	44
Declaring Dependencies between Components .....	47
Summary .....	47
6. What is AZ::Interface? .....	50
Introduction .....	50
Level Components .....	50
Defining an AZ::Interface .....	51
Summary .....	53
7. What is an AZ::Event? .....	56
Introduction .....	56
Example .....	57
Summary .....	59
8. What is an AZ::EBus? .....	62
Introduction .....	62
The Basic Idea of an EBus .....	63
Example of an EBus: TransformBus .....	64
What is an AZ::EntityId? .....	67
EBus Event versus EventResult .....	69
9. Using AZ::TickBus .....	70
Introduction to AZ::TickBus .....	70
OscillatorComponent .....	71
Implementing Oscillator Component .....	73
Summary .....	75

# Chapter 5. What is FindComponent?

## Introduction

The easiest and most direct form of communication between components is to get the pointer to the other component and then directly invoke its methods. With a raw pointer to another component, you get the fastest form of communication. There are some drawbacks to this approach but very often it is worth it. It is also the easiest one to understand, which is why I am starting the discussion with the use of `FindComponent()`.

### Note

You can find the code and project changes for this chapter on GitHub:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch05\\_find\\_component](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch05_find_component)

## An Example

Any component that derives from `AZ::Component` can invoke the following:

```
GetEntity() -> FindComponent<T>()
```

That enumerates over the components of an entity until a component of a given type `T` is found. You can find it in the engine code here:

### **Example 5.1. FindComponent in AzCore\Component\Entity.h**

```
///! Finds the first component of the requested component type.  
///! @return A pointer to the first component of the requested type.  
///! Returns a null pointer if a component of the type cannot be found.  
template<class ComponentType>  
inline ComponentType* FindComponent() const
```

In previous chapters I covered `Transform` component. In this chapter, I will show you how to get its pointer and get world translation (position) of the entity using `TransformComponent`'s public interface.

### **Example 5.2. An Example of Calling GetWorldTM on Transform Component**

```
AZ::Entity* e = GetEntity();  
  
using namespace AzFramework;  
TransformComponent* tc = e->FindComponent<TransformComponent>();  
if (tc)  
{  
    AZ::Transform t = tc->GetWorldTM();  
    AZ::Vector3 p = t.GetTranslation();
```

It is as simple as that but there are some things to consider here.

- We are referencing a class from `AzFramework` library, so we will need to update our build script to include `AzFramework` for `MyProject.Static` library.

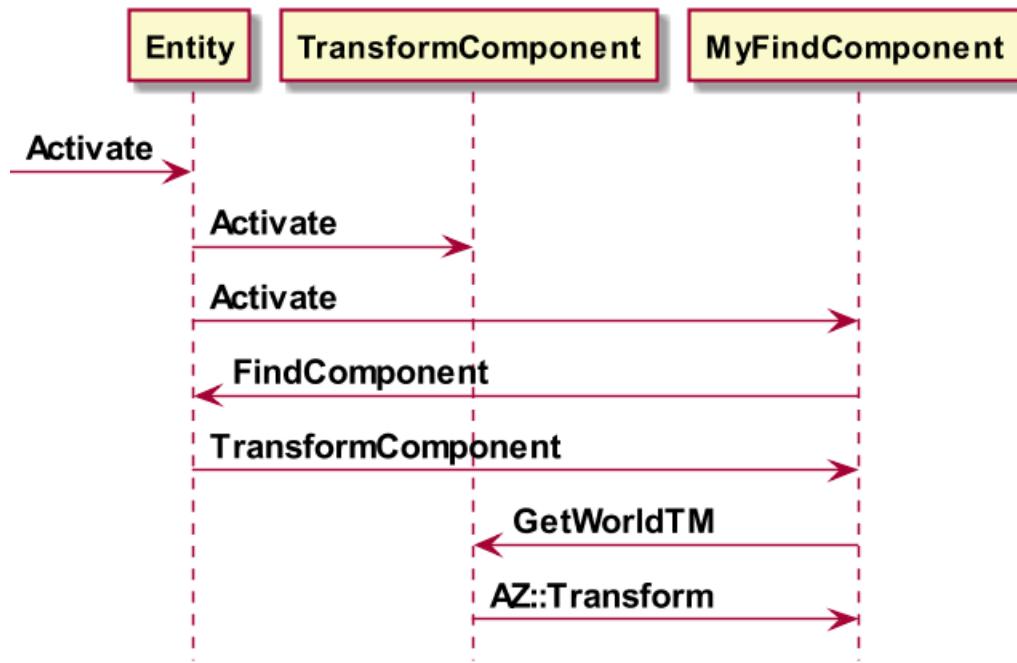
- We are adding a linking cost to the project, as building *MyProject.Static* from now on has to include the entire AzFramework static library. Over time that can grow to a large amount of time linking for sizable game projects.

### Note

AzFramework library is included just about everywhere, so it will not be an issue but the general idea applies. Keep an eye on the linking time of your project.

- We have added a dependence on another component. It means we have to consider what to do if the component is not found. If we place this code at component activation, such as inside `MyFindComponent::Activate()`, what happens if `TransformComponent` activates after `MyFindComponent`? The component may not be ready for us to communicate.

### Example 5.3. An example of FindComponent



### Important

Activation of Transform component must occur before activation of MyFindComponent for our logic to work.

Let us go over how to address these concern.

## Adding a Library Reference

`MyProject.Static` library requires a reference to `AzFramework` library. You can declare that inside `C:\git\book\MyProject\Code\CMakeLists.txt`, which is the build file for our game project. In this case, we only care about the definition of `MyProject.Static` library, which is as follows:

### Example 5.4. Definition of `MyProject.Static` in CMake

```
my_add_target(
```

```

NAME MyProject.Static STATIC
NAMESPACE Gem
FILES_CMAKE
    myproject_files.cmake
INCLUDE_DIRECTORIES
    PUBLIC
        Include
BUILD_DEPENDENCIES
    PRIVATE
        AZ::AzGameFramework
        Gem::Atom_AtomBridge.Static
)

```

The list of included libraries for MyProject.Static is under BUILD\_DEPENDENCIES. It has two sections: private and public. Private portion tells the build system which libraries to include for this build target only. Public portion tells the build system to include the libraries for the build target and for any targets that include MyProject.Static as well.

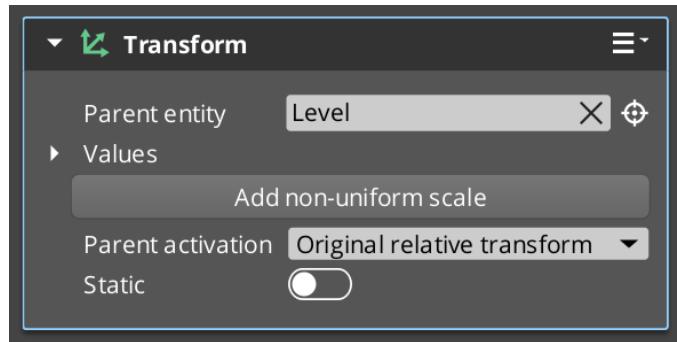
At the moment, we are not interested in this difference and PRIVATE section will work.

```

BUILD_DEPENDENCIES
PRIVATE
    AZ::AzFramework

```

**How do you know which library to include to get access to TransformComponent?** For this, we have to do some spelunking. We will go on a journey of going through source and build files to find out where TransformComponent lives. In the beginning we might only know about a component through the Editor.



The component is called *Transform* in the Entity Inspector. As I showed in the previous chapter, a component declares its name by specifying in its Reflect method. If you search long enough you can find the following declaration:

### Example 5.5. AzToolsFramework\ToolsComponents\TransformComponent.cpp

```

TransformComponent::Reflect(...)
{
    ptrEdit->Class<TransformComponent>("Transform",
        "Controls the placement of the entity in the world in 3d")
}

```

#### Note

Here I am looking at the engine code directly. You should clone a copy of the engine if for nothing else then for reference.

```
cd c:\git
git clone https://github.com/o3de/o3de
```

Then the above mentioned file will be at C:\git\o3de\Code\Framework\AzToolsFramework\AzToolsFramework\ToolsComponents\TransformComponent.cpp.

This reflection belongs to AzToolsFramework::Components::TransformComponent. It is the editor flavor of the component, whereas AzFramework::TransformComponent is the game component that you will interact in runtime when your game is running.

```
namespace AzToolsFramework
{
    namespace Components
    {
        class TransformComponent
        : public EditorComponentBase
```

For now, just know that if a component you are investigating is inheriting from EditorComponentBase then it is an editor component that is temporary in nature and only exists at design time. When a level is spawned at game time, these editor components are converted to game components.

## Tip

In fact, you can see which game component is created by looking at BuildGameEntity method of Transform Component.

```
void TransformComponent::BuildGameEntity(AZ::Entity* gameEntity)
{
    ...
    gameEntity->CreateComponent<
        AzFramework::TransformComponent>()
```

Now that we know which component we will interact with at runtime we can find the library that includes it. The reference to TransformComponent.h is found in C:\git\o3de\Code\Framework\AzFramework\AzFramework\azframework\_files.cmake. Well, what is using this list of files? The build script for AzFramework static library does.

### **Example 5.6. C:\git\o3de\Code\Framework\AzFramework\CMakeLists.txt**

```
ly_add_target(
    NAME AzFramework STATIC
    NAMESPACE AZ
    FILES_CMAKE
        AzFramework/azframework_files.cmake
    ...
)
```

Now we can link against this static library by using its name and namespace - AZ::AzFramework.

### **Example 5.7. MyProject.Static linking against AZ::AzFramework library**

```
ly_add_target(
    NAME MyProject.Static STATIC
```

```

    NAMESPACE Gem
    ...
    BUILD_DEPENDENCIES
    PRIVATE
        AZ::AzFramework
    ...
)

```

## Declaring Dependencies between Components

The next concern we should to tackle is getting rid of a situation where we do not get the component we want due to activation order of components.

```
TransformComponent* tc = e->FindComponent<TransformComponent>();
```

We can ensure that we find the component we are looking for declaring a game design dependency of our component on TransformComponent. A component can declare services it provides and services it requires. For example, the editor version of Transform component declares the following service:

```

void TransformComponent::GetProvidedServices(
    AZ::ComponentDescriptor::DependencyArrayType& provided)
{
    provided.push_back(AZ_CRC("TransformService", 0x8ee22c50));
}

```

Our example component, MyFindComponent, will then declare a dependency in static GetRequiredServices method:

```

static void GetRequiredServices(
    AZ::ComponentDescriptor::DependencyArrayType& required)
{
    required.push_back(AZ_CRC_CE("TransformService"));
}

```

### Note

Both AZ\_CRC and AZ\_CRC\_CE macros serve the same function. They convert a string into a hashed integer. The only difference is that AZ\_CRC\_CE computes the value at compile time.

When you are building entities in the Editor, the components will be saved in such a way that TransformComponent will always come before MyFindComponent.

### Important

There is one gotcha with these dependencies and the Editor, when you add or change a dependency rule for a component that is already on the entity, you will have to modify the entity's components for the rule to be re-applied.

## Summary

### Note

You can find the code and project changes for this chapter on GitHub:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch05\\_find\\_component](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch05_find_component)

Here is the full source code for our new component.

### Example 5.8. MyFindComponent.h

```
#pragma once
#include <AzCore/Component/Component.h>

namespace MyProject
{
    // An example of the simplest O3DE component
    class MyFindComponent : public AZ::Component
    {
        public:
            AZ_COMPONENT(MyFindComponent,
                         "{FE4F2E82-8E03-48EC-A967-559705597040}");

            static void GetRequiredServices(
                AZ::ComponentDescriptor::DependencyArrayType& required)
            {
                required.push_back(AZ_CRC_CE("TransformService"));
            }

            // AZ::Component overrides
            void Activate() override;
            void Deactivate() override {}

            // Provide runtime reflection, if any
            static void Reflect(AZ::ReflectContext* rc);
    };
}
```

### Example 5.9. MyFindComponent.cpp

```
#include "MyFindComponent.h"
#include <AzCore/Component/Entity.h>
#include <AzCore/Serialization/EditContext.h>
#include <AzFramework/Components/TransformComponent.h>

using namespace MyProject;

void MyFindComponent::Activate()
{
    AZ::Entity* e = GetEntity();

    using namespace AzFramework;
    TransformComponent* tc = e->FindComponent<TransformComponent>();
    if (tc)
    {
        AZ::Transform t = tc->GetWorldTM();
        AZ::Vector3 p = t.GetTranslation();
        AZ_Printf("MyFind", "activated at %f %f %f",

```

```
        p.GetX(), p.GetY(), p.GetZ() );
    }

void MyFindComponent::Reflect(AZ::ReflectContext* rc)
{
    auto sc = azrtti_cast<AZ::SerializeContext*>(rc);
    if (!sc) return;

    sc->Class<MyFindComponent, Component>()
        ->Version(1);

    AZ::EditContext* ec = sc->GetEditContext();
    if (!ec) return;

    using namespace AZ::Edit::Attributes;
    // reflection of this component for O3DE Editor
    ec->Class<MyFindComponent>("Find Component Example",
        [Communicates using FindComponent])
        ->ClassElement(AZ::Edit::ClassElements::EditorData, " ")
            ->Attribute(AppearsInAddComponentMenu, AZ_CRC("Game"))
            ->Attribute(Category, "My Project");
}
```

# Chapter 6. What is AZ::Interface?

## Introduction

The majority of component communication in O3DE occurs using AZ::EBus. It is the most powerful, the most generalized and the most feature rich inter-component communication system in the engine. I could even say that once you understand AZ::EBus you will understand how to work with the engine. However, I will start by showing you the simpler models first - AZ::Interface in this chapter and then AZ::Event in the next chapter. By studying these two first you will be ready to understand AZ::EBus.

### Note

You can find the code and project changes for this chapter on GitHub:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch06\\_az\\_interface](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch06_az_interface)

AZ::Interface in O3DE is a fancy singleton that works across module boundaries. It allows you declare an interface and then implement it in one of your objects. However, since AZ::Interface is, essentially, a singleton, a component that acts as an interface handler has to be the only component that handles that interface.

### Note

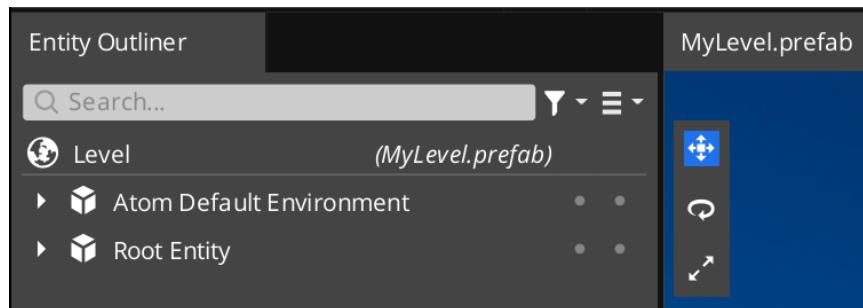
You can find the documentation on AZ::Interface online at

<https://www.o3de.org/docs/user-guide/programming/az-interface/>

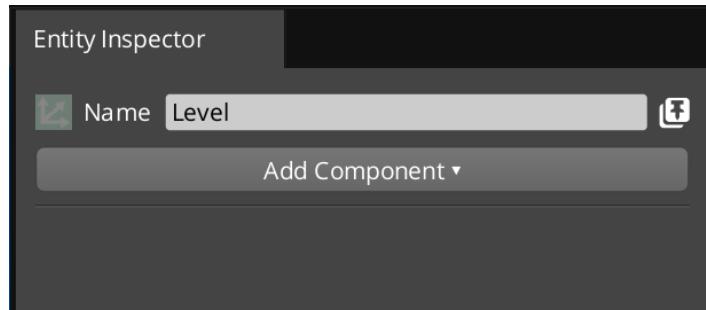
## Level Components

That brings us to a component type that is naturally unique and would serve as a great candidate to handle an AZ::Interface: *Level* components. We have already seen regular *game* components that can be added to entities in the Editor. There is another type of entity hidden in plain sight in *Entity Outliner*.

**Figure 6.1. Accessing Level entity and its components**



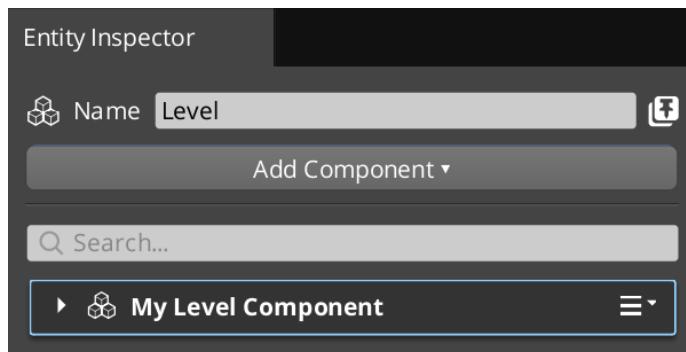
If you click on **Level** title, Entity Inspector will show you the Level entity, which by default has no components on it.

**Figure 6.2. Level entity in Entity Inspector**

Only a component marked as a *Level* component in their `Reflect` method can be added to Level entity.

```
ec->Class<MyLevelComponent>("My Level Component",
    "[Communicates using AZ::Interface]")
->ClassElement(AZ::Edit::ClassElements::EditorData, "")
->Attribute(AppearsInAddComponentMenu, AZ_CRC_CE("Level"))
```

### Example 6.1. Level entity with My Level Component



#### Tip

Whenever you find the Editor or the Asset Processor failing to start after you just added a new component, check that your new components have unique guids. It is easy to copy and paste the new component files and forget to update the guids. The debug log in Visual Studio will show the following error in such a scenario:

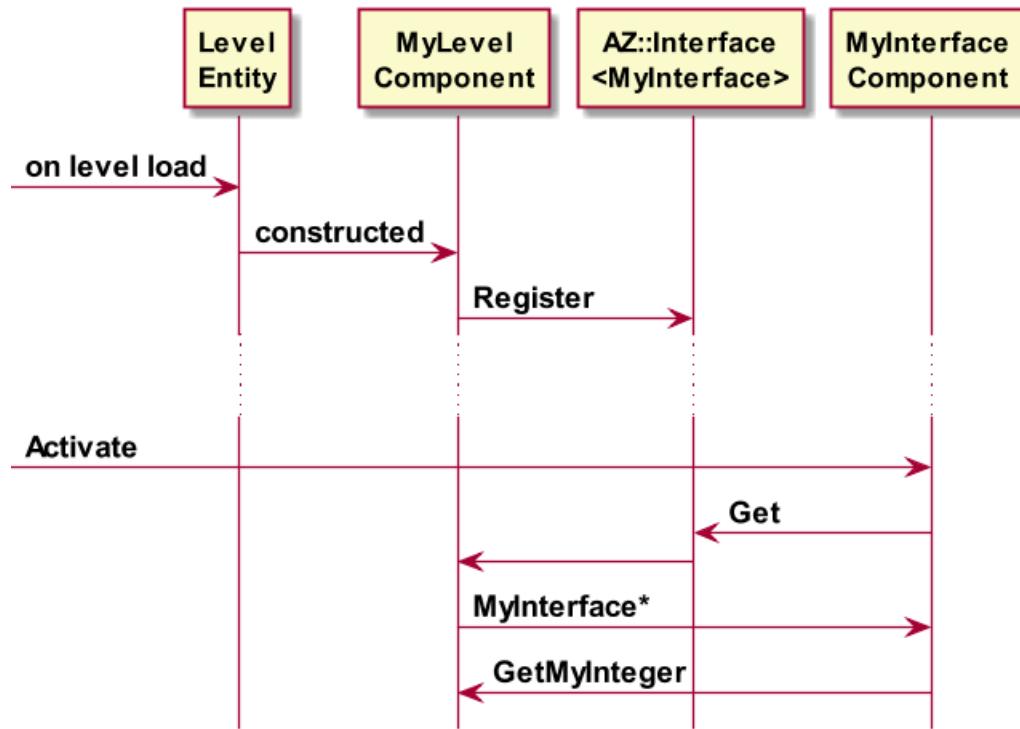
```
Component: Two different components have the same UUID
({FE4F2E82-8E03-48EC-A967-559705597040}), which is not allowed.
Change the UUID on one of them.
```

## Defining an AZ::Interface

In order to define and use your own AZ::Interface, there are three steps involved:

- Define a pure virtual class as the interface in a public header, such as `MyInterface`.
- Assign a handler for the interface using `AZ::Interface<MyInterface>::Registrar`
- Access the interface from another component using `AZ::Interface<MyInterface>::Get()`

**Figure 6.3. Using AZ::Interface**



Here is the interface definition.

#### Example 6.2. C:\git\book\MyProject\Code\Include\MyProject\MyInterface.h

```

#pragma once
#include <AzCore/RTTI/RTTI.h>

namespace MyProject
{
    class MyInterface
    {
        public:
            AZ_RTTI(MyInterface, "{D477F807-6795-4A1B-A475-AFBCF0584A08}");
            virtual ~MyInterface() = default;

            virtual int GetMyInteger() = 0;
    };
}
  
```

Here is an example of a component that acts as a handler for any requests placed on the interface.

#### Example 6.3. MyLevelComponent with AZ::Interface

```

// An example of a level component with AZ::Interface
class MyLevelComponent
    : public AZ::Component
    , public AZ::Interface<MyInterface>::Registrar
  
```

AZ::Interface<MyInterface>::Registrar will register the class as the handler for the interface at construction time of MyLevelComponent and unregister at destruction time.

## Tip

You can also manually register and unregister from the interface using Register and Unregister methods.

```
AZ::Interface<MyInterface>::Register(this);
// ...
AZ::Interface<MyInterface>::Unregister(this);
```

Here is an example of a component that uses the interface.

### Example 6.4. Using AZ::Interface in another component

```
void MyInterfaceComponent::Activate()
{
    if (MyInterface* myInterface = AZ::Interface<MyInterface>::Get())
    {
        AZ_Printf("Example", "%d", myInterface->GetMyInteger());
    }
}
```

## Summary

In this chapter I covered the use of AZ::Interface as a singleton that works across module boundaries. There were two components created. MyLevelComponent implemented MyInterface and was assigned to the level entity. MyInterfaceComponent accessed MyInterface. We had two components communicate with each despite being assigned on an entirely different kind of entities.

Here is the source code for MyLevelComponent.

### Example 6.5. MyLevelComponent.h

```
#pragma once
#include <AzCore/Component/Component.h>
#include <AzCore/Interface/Interface.h>
#include <MyProject/MyInterface.h>

namespace MyProject
{
    // An example of a level component with AZ::Interface
    class MyLevelComponent
        : public AZ::Component
        , public AZ::Interface<MyInterface>::Registrar
    {
    public:
        AZ_COMPONENT(MyLevelComponent,
                     "{69C64F9C-4C35-4612-9B3B-85FEBCC06FDB}");

        // AZ::Component overrides
        void Activate() override {}
        void Deactivate() override {}
    };
}
```

```

    // Provide runtime reflection, if any
    static void Reflect(AZ::ReflectContext* rc);

    // MyInterface
    int GetMyInteger() override { return 42; }
};

}

```

**Example 6.6. MyLevelComponent.cpp**

```

#include "MyLevelComponent.h"
#include <AzCore/Serialization/EditContext.h>

using namespace MyProject;

void MyLevelComponent::Reflect(AZ::ReflectContext* rc)
{
    auto sc = azrtti_cast<AZ::SerializeContext*>(rc);
    if (!sc) return;

    sc->Class<MyLevelComponent, Component>()
        ->Version(1);

    AZ::EditContext* ec = sc->GetEditContext();
    if (!ec) return;

    using namespace AZ::Edit::Attributes;
    // reflection of this component for O3DE Editor
    ec->Class<MyLevelComponent>("My Level Component",
        "[Communicates using AZ::Interface]")
        ->ClassElement(AZ::Edit::ClassElements::EditorData, "")
            ->Attribute(AppearsInAddComponentMenu, AZ_CRC_CE("Level"))
            ->Attribute(Category, "My Project");
}

```

Here is the source code for MyInterfaceComponent.

**Example 6.7. MyInterfaceComponent.h**

```

#pragma once
#include <AzCore/Component/Component.h>

namespace MyProject
{
    // An example of using AZ::Interface
    class MyInterfaceComponent : public AZ::Component
    {
    public:
        AZ_COMPONENT(MyInterfaceComponent,
            "{F73AB7B7-4F19-42FD-9651-13167FD222A6}");

        // AZ::Component overrides
        void Activate() override;
    };
}

```

```

void Deactivate() override {}

// Provide runtime reflection, if any
static void Reflect(AZ::ReflectContext* rc);
};

}

```

### Example 6.8. MyInterfaceComponent.cpp

```

#include "MyInterfaceComponent.h"
#include <AzCore/Serialization/EditContext.h>
#include <MyProject/MyInterface.h>

using namespace MyProject;

void MyInterfaceComponent::Activate()
{
    if (MyInterface* myInterface = AZ::Interface<MyInterface>::Get())
    {
        AZ_Printf("Example", "%d", myInterface->GetMyInteger());
    }
}

void MyInterfaceComponent::Reflect(AZ::ReflectContext* rc)
{
    auto sc = azrtti_cast<AZ::SerializeContext*>(rc);
    if (!sc) return;

    sc->Class<MyInterfaceComponent, Component>()
        ->Version(1);

    AZ::EditContext* ec = sc->GetEditContext();
    if (!ec) return;

    using namespace AZ::Edit::Attributes;
    // reflection of this component for O3DE Editor
    ec->Class<MyInterfaceComponent>("Using Interface Example",
        "[Communicates using AZ::Interface]")
        ->ClassElement(AZ::Edit::ClassElements::EditorData, "")
            ->Attribute(AppearsInAddComponentMenu, AZ_CRC("Game"))
            ->Attribute(Category, "My Project");
}

```

# Chapter 7. What is an AZ::Event?

## Introduction

### Note

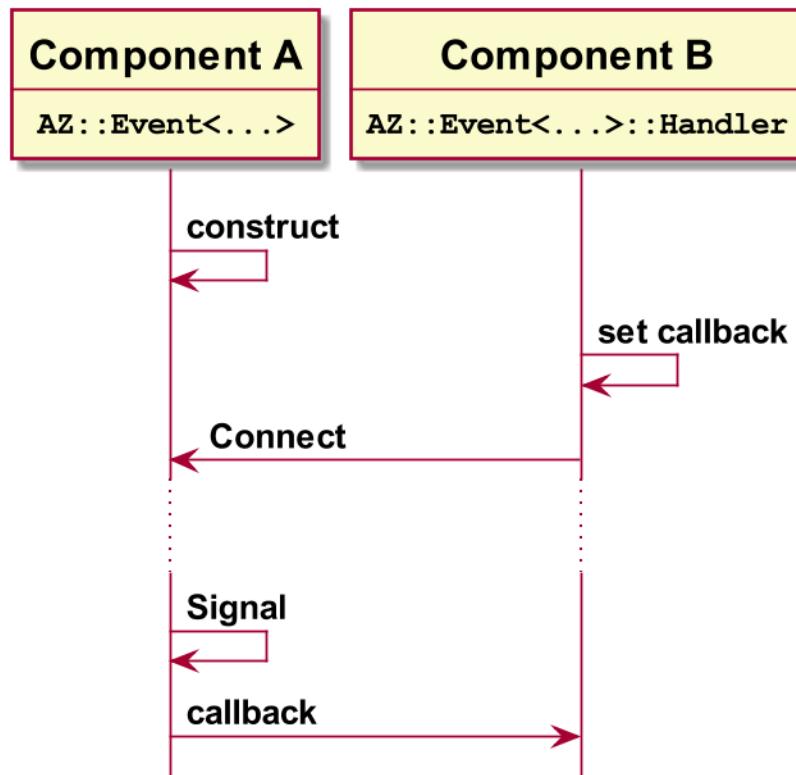
The official reference for AZ::Event can be found here:

<https://www.o3de.org/docs/user-guide/programming/az-event/>

AZ::Event is a great tool for implementing notification systems between components, where one component acts a publisher of events and others act as subscribers. There can be many subscribers on the same publisher. AZ::Event is not suited as a getter interface but is excellent for notification calls, such as a Transform component notifying any listening components that a change in world transform has occurred.

Here are C++ building blocks to create an AZ::Event.

**Figure 7.1. Setting up an AZ::Event**



1. Define an event using `AZ::Event<>`.

```
AZ::Event<int> myEvent;
```

2. Define a handler using `AZ::Event<>::Handler` with a callback.

```
AZ::Event<int>::Handler myHandler(  
    [](int value)
```

```
{
    /*process*/
} );
```

3. Connect the handler to the event.

```
myHandler.Connect(myEvent);
```

4. Send notifications from the event.

```
myEvent.Signal(42);
```

### Tip

You can define none or multiple parameters in the template arguments of AZ::Event<T>. The handler must match those types. Signal must use the same type of parameters as well.

For example, TransformComponent sends notifications out whenever its entity moves or changes its rotation or scale. It does so by defining the following event in C:\git\o3de\Code\Framework\AzCore\AzCore\Component\TransformBus.h:

```
using TransformChangedEvent = AZ::Event<
    const Transform&, const Transform&>;
```

There has to be a way to connect handlers to this event, which is done by calling TransformComponent::BindTransformChangedEventHandler:

```
void TransformComponent::BindTransformChangedEventHandler(
    AZ::TransformChangedEvent::Handler& handler)
{
    handler.Connect(m_transformChangedEvent);
}
```

If you look over the code of TransformComponent.cpp you will see that it signals the event at the appropriate time with:

```
m_transformChangedEvent.Signal(m_localTM, m_worldTM);
```

## Example

With the introductory details out of the way, we can build a new component that will print a debug message whenever an entity moves. There are four steps here.

1. Declare the right type of handler and optionally a callback method for the notification.

```
AZ::TransformChangedEvent::Handler m_movementHandler;
void OnWorldTransformChanged(const AZ::Transform& world);
```

2. In the constructor of the component define the lambda invoking our callback method OnWorldTransformChanged.

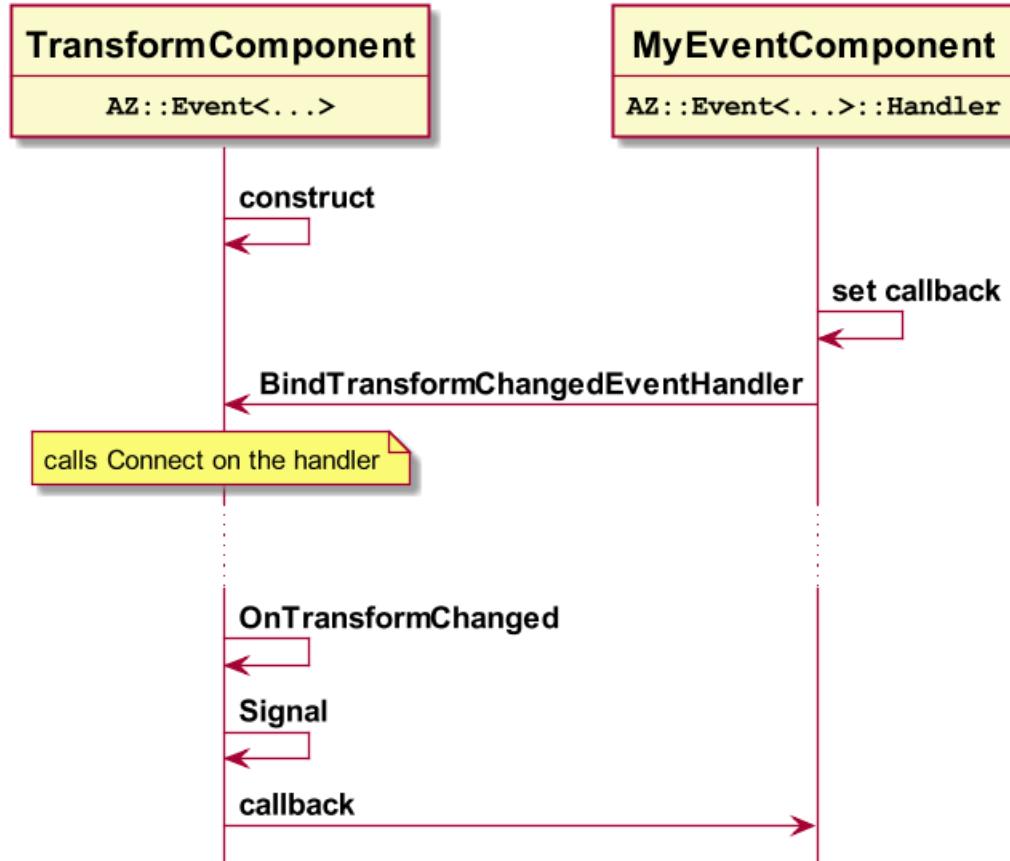
```
MyEventComponent::MyEventComponent()
: m_movementHandler(
    [this](
        const AZ::Transform& /*local*/,
```

```

const AZ::Transform& world)
{
    OnWorldTransformChanged(world);
}
{
}

```

Figure 7.2. TransformComponent notifies using AZ::Event



3. Implement the callback method to print the message.

```

void MyEventComponent::OnWorldTransformChanged(
    const AZ::Transform& world)
{
    AZ_Printf("MyEvent", "now at %f %f %f",
        world.GetTranslation().GetX(),
        world.GetTranslation().GetY(),
        world.GetTranslation().GetZ());
}

```

### Note

One could also just do all the work inside the lambda.

4. Lastly, the handler needs to be connected to the event via **BindTransformChangedEventHandler**.

```
void MyEventComponent::Activate()
{
    AZ::Entity* e = GetEntity();
    TransformComponent* tc = e->FindComponent<TransformComponent>();
    // track the movement of the entity
    tc->BindTransformChangedEventHandler(m_movementHandler);
}
```

### Tip

Different components may provide different ways to connect your handlers to their events but often the method name is of a similar name schema: BindSomethingEventHandler.

## Summary

### Note

You can find the code and project changes for this chapter on GitHub:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch06\\_az\\_interface](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch06_az_interface)

Here is the entire source for the new component, MyEventComponent.

#### Example 7.1. MyEventComponent.h

```
#pragma once
#include <AzCore/Component/Component.h>
#include <AzCore/Component/TransformBus.h>

namespace MyProject
{
    // An example of listening to movement events
    // of TransformComponent using an AZ::Event
    class MyEventComponent : public AZ::Component
    {
        public:
            AZ_COMPONENT(MyEventComponent,
                         "{934BF061-204B-4695-944A-23A1BA7433CB}");

            static void GetRequiredServices(
                AZ::ComponentDescriptor::DependencyArrayType& required)
            {
                required.push_back(AZ_CRC_CE("TransformService"));
            }

        MyEventComponent();

        // AZ::Component overrides
        void Activate() override;
        void Deactivate() override {}

        // Provide runtime reflection, if any
    };
}
```

```

    static void Reflect(AZ::ReflectContext* rc);

private:
    AZ::TransformChangedEvent::Handler m_movementHandler;
    void OnWorldTransformChanged(const AZ::Transform& world);
};

}


```

### Example 7.2. MyEventComponent.cpp

```

#include "MyEventComponent.h"
#include <AzCore/Component/Entity.h>
#include <AzCore/Serialization/EditContext.h>
#include <AzFramework/Components/TransformComponent.h>

using namespace MyProject;

MyEventComponent::MyEventComponent()
: m_movementHandler(
    [this]{
        const AZ::Transform& /*local*/,
        const AZ::Transform& world)
{
    OnWorldTransformChanged(world);
})
{
}

void MyEventComponent::OnWorldTransformChanged(const AZ::Transform& world)
{
    AZ_Printf("MyEvent", "now at %f %f %f",
        world.GetTranslation().GetX(),
        world.GetTranslation().GetY(),
        world.GetTranslation().GetZ());
}

void MyEventComponent::Activate()
{
    AZ::Entity* e = GetEntity();

    using namespace AzFramework;
    TransformComponent* tc = e->FindComponent<TransformComponent>();
    // track the movement of the entity
    tc->BindTransformChangedEventHandler(m_movementHandler);
}

void MyEventComponent::Reflect(AZ::ReflectContext* rc)
{
    auto sc = azrtti_cast<AZ::SerializeContext*>(rc);
    if (!sc) return;

    sc->Class<MyEventComponent, Component>()
        ->Version(1);
}

```

```
AZ::EditContext* ec = sc->GetEditContext();
if (!ec) return;

using namespace AZ::Edit::Attributes;
ec->Class<MyEventComponent>("My Event Component Example",
    [Communicates using AZ::Event])
->ClassElement(AZ::Edit::ClassElements::EditorData, " ")
    ->Attribute(AppearsInAddComponentMenu, AZ_CRC("Game"))
    ->Attribute(Category, "My Project");
}
```

---

# Chapter 8. What is an AZ::EBus?

## Introduction

A great way to explain something complex is by contrasting it to something similar but slightly different.

**What are common ways for objects or components in a game engine to communicate with each other?** Most often the method is by a direct call. That is you get a pointer or a reference to an instance of a component, and then invoke a method on it. Sometimes it might be done through a direct type pointer and sometimes it is done through a base class pointer.

```
// pseudo-code ahead
{
    MyComponent* a;
    a->DoSomething();

    // OR

    MyComponent* a;
    MyBase* b = a;
    b->DoSomething();
}
```

**In programming, whatever seems simple on small scale is often completely broken and terrible on large scale.** The above design certainly works. However, in large systems it often leads to everything becoming intertwined and turning into spaghetti code. Here is my theory on what is causing that in the context of a large scale product code with direct component interaction. Decoupling and abstraction is often only thought about member variables and base class pure interfaces. But there is more to decoupling than just decoupling a component's interface.

**Abstracting away the invocation of a component is just as important as abstracting away the details of the implementation of a component.** The fundamental principles of Object-Oriented Programming are well-known. One of such principles is abstraction. One abstracts away the details of a class, so that one can provide various runtime behaviors without binding the caller to a specific interface. This way A can call either B or C the same way, so long as both inherit from the same base class. However, there is another element here that can be abstracted away.

Consider that whenever you invoke a method on any instance, you had to get a pointer or a reference to that instance. That is a form of coupling between the caller and the callee. After all, however you got that pointer or a reference is how you will have to invoke the object.

**Now imagine that you abstracted away the details of how you call an object.** Now you decouple both what method you call on an object and how you call the method. That means you do not need worry about the receiving side at all. Maybe it is there, maybe it is not. Since you no longer hold on to a pointer, you do not have to commit yourself to a specific instance that you are invoking. One moment you might be talking to one instance, and the next moment you may be talking to a different instance. Or you might be calling a method on different number of objects at different times.

That is the real power of a design behind EBuses. Let us now take a look of what this theory means in practice.

# The Basic Idea of an EBus

Event buses (EBuses) are a general-purpose communication system that O3DE uses to dispatch notifications and receive requests. EBuses are configurable and support many different use cases.

Here is a common direct pointer approach through a virtual interface:

```
// pseudo-code ahead
Caller A;
Callee B;

// A somehow gets a pointer or a reference to B
BaseCallee* bb = &B;

A::Do()
{
    bb->DoSomething();
}
```

Now consider what you would need to do replace what `bb` is pointing to. For example, let us say that you were writing a unit test and wished to test the behavior of `A` towards `B`. If `A` had saved the pointer `bb` internally, which is common, you would have to make sure there is a way to modify that pointer and be sure that it is safe to change it.

Have you ever seen the pattern of `DEBUG_SetVariable()` in a public class before? Perhaps, it was even defined out in release builds, using preprocessor techniques with `#ifdef`? It is often created just for this purpose. Somebody had to write custom code to poke into an object to make it talk to a test object (sometimes known as a mock object).

```
// pseudo-code ahead
class A{
public:
    void DoSomething();
#ifndef _RELEASE
    void DEBUG_ChangePointerToB(BaseCallee*); // YUCK!
#endif
};
```

Or perhaps, this was solved by passing the object at construction. However, that couples the two objects at construction time, which has its own issues, because then you have to pass the right object at construction time. Long story short, while all such approaches work to one degree or another, they all are caused by not abstracting away the communication between objects.

Now, imagine a pseudo-code that does not couple the method of invocation:

```
// pseudo-code ahead
Caller A;
Callee B;
B.connect_on(42);

A::CallAnother()
{
    call_on(42, &DoSomething); // where DoSomething is a method
}
```

42 is only meant to be an arbitrary identifier. Its purpose is to figure out who will get `DoSomething()`. Taking this logic one step further, you can imagine B connecting and disconnecting on 42 whenever it pleases.

```
// pseudo-code ahead
B.connect_on(42);

// somebody calls
call_on(42, &DoSomething);

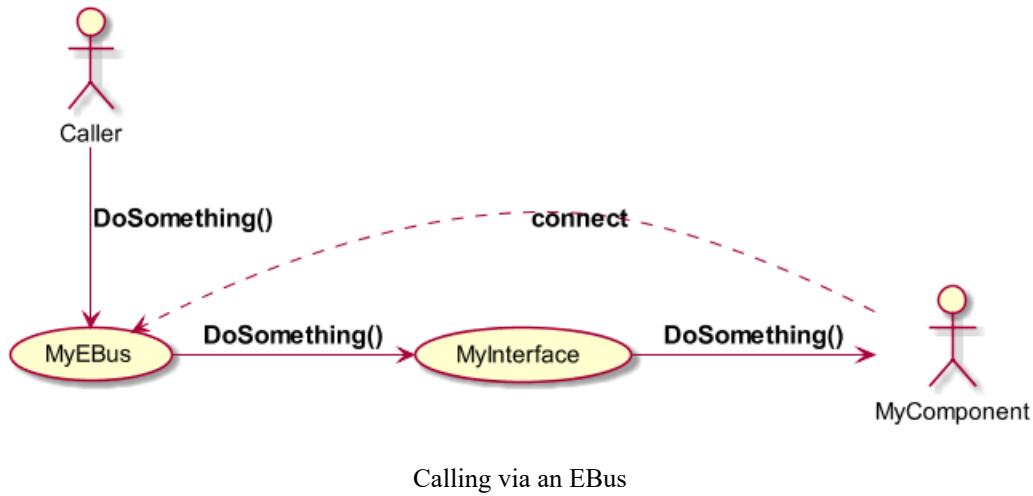
D.connect_on(42);
// now this method will be called on two objects
call_on(42, &DoSomething);

// or both disconnect
B.disconnect(42);
D.disconnect(42);

// Now nobody gets the call
call_on(42, &DoSomething);
```

Here is a graphical form of this form of communication, using an EBus design.

**Figure 8.1. Calling through an interface behind an EBus**



Now, you could re-route `DoSomething` from the Caller to another destination without changing the Caller at all. Instead, it would be up to `MyComponent` or other objects that act like it, such as `MyMockComponent`.

### ⚠ Important

Incidentally, this makes unit testing O3DE components a breeze. We will take a look at how to write unit tests for components in Chapter 15, *Writing Unit Tests for Components*.

## Example of an EBus: TransformBus

Well, that is enough theory for now. Let us take a look at a real example of an EBus and how one would invoke it. One of the most common EBuses in all of O3DE is `TransformBus`, which is implement-

ed by TransformComponent. Whenever you wish to move an AZ::Entity you would have to use TransformBus. Its definition can be found at C:\O3DE\21.11.2\Code\Framework\AzCore\AzCore\Component\TransformBus.h:

### Example 8.1. TransformBus definition

```
/***
 * The EBus for requests to position and parent an entity.
 * The events are defined in the AZ::TransformInterface class.
 */
typedef AZ::EBus<TransformInterface> TransformBus;
```

This is a declaration of an EBus. It declares a new type of AZ::EBus, TransformBus, with an interface provided by a pure virtual class TransformInterface. You can find it in the same file:

### Example 8.2. TransformInterface snippet

```
///! Interface for AZ::TransformBus, which is an EBus that
///! receives requests to translate (position), rotate, and
///! scale an entity in 3D space.
class TransformInterface
    : public ComponentBus
{
public:
...
    ///! Sets the entity's world space translation, which
    ///! represents how to move the entity to a new position
    ///! within the world.
    virtual void SetWorldTranslation(
        [[maybe_unused]] const AZ::Vector3& newPosition) {}

    ///! Gets the entity's world space translation.
    ///! @return A three-dimensional translation vector.
    virtual AZ::Vector3 GetWorldTranslation() {
        return AZ::Vector3(FLOAT_MAX); }
```

TransformComponent can be found at C:\O3DE\21.11.2\Code\Framework\AzFramework\AzFramework\Components\TransformComponent.h:

```
class TransformComponent
    : public AZ::Component
    , public AZ::TransformBus::Handler
...
```

As you can see it implements Handler sub-class of the EBus. In other words, that declares that TransformComponent will *handle* calls for TransformBus. And lastly, TransformComponent has to connect to TransformBus. You can see that in the corresponding source file, C:\git\o3de\Code\Framework\AzFramework\AzFramework\Components\TransformComponent.cpp:

```
void TransformComponent::Activate()
{
    AZ::TransformBus::Handler::BusConnect(m_entity->GetId());
...
}
```

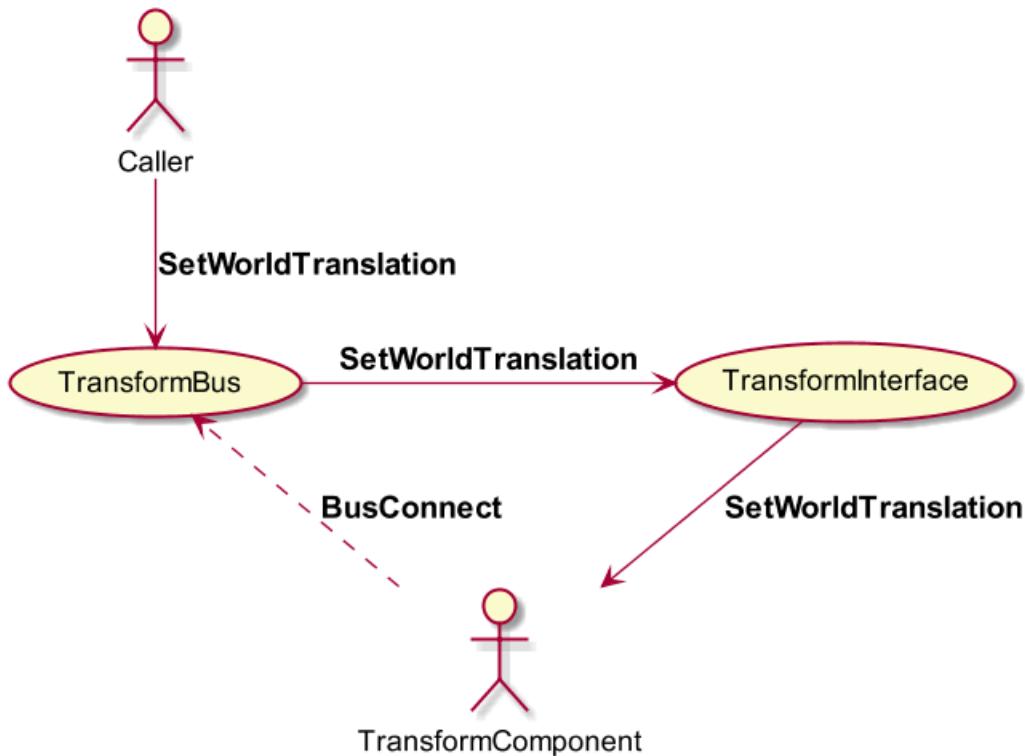
## Note

C:\git\o3de is my location where I cloned GitHub repository of O3DE:

<https://github.com/o3de/o3de>

We will come back to the meaning of `Activate()` and `m_entity->GetId()` in a moment. For now, let us glance at the whole picture.

**Figure 8.2. TransformBus and TransformComponent**



And here is how one would invoke `SetWorldTranslation` through `TransformBus`:

### **Example 8.3. Example of an EBus call**

```

AZ::TransformBus::Event(
    // where is it going?
    entity.GetId(),
    // what method will be invoked?
    &AZ::TransformBus::Events::SetWorldTranslation,
    // input parameter(s)?
    position);
  
```

This means that the caller is calling somebody through `TransformBus` whose identifier is `entity.GetId()` on the method `SetWorldTranslation` with a single input parameter, `position`. Who will receive this call? Whoever connected to `TransformBus` with the same entity id.

```
AZ::TransformBus::Handler::BusConnect(m_entity->GetId());
```

**Note**

This is not the only way to call an EBus. There are many different ways to suite your needs. Later chapters will go over common cases.

## What is an AZ::EntityId?

**In a world of Entities and Components attached to Entities, what would you expect to be the common way of addressing a specific component in the world?** The most common communication in O3DE is of a form: "Hey, you, the component of type X attached to entity Y, do this!" For example, if you wish to move an entity, then you would need to tell its attached **Transform** component to move. Thus, we have to know how to uniquely identify entities.

**Note**

In the Editor, one can set Entity names, however, those names are not unique and are only meant to be used for debugging purposes.

Entities are uniquely identified by an AZ::EntityId. Its definition can be found at C:\O3DE\21.11.2\Code\Framework\AzCore\AzCore\Component\EntityId.h

```
/**
 * Entity ID type.
 * Entity IDs are used to uniquely identify entities. Each
 * component that is attached to an entity is tagged with
 * the entity's ID, and component buses are typically
 * addressed by entity ID.
 */
class EntityId
{
    ...
    /**
     * Entity ID.
     */
    u64 m_id;
}
```

Under the hood, EntityId is a 64-bit unsigned integer, which is an AZ::u64 type in O3DE.

There is a number of ways of communicating with another component using AZ::EntityId:

1. Talk to a component on the same entity.
2. Talk to a component on the parent entity.
3. Talk to a component on a child entity.
4. Talk to a component on an unrelated entity.

## EBus on the Same Entity

This is one of the most common ways for components to communicate.

In general, a component that is attached to an entity has its entity id assigned:

```

class Component
{
...
/**
 * Returns the entity ID if the component is attached to
 * an entity. If the component is not attached to any
 * entity, this function asserts. As a safeguard, make
 * sure that GetEntity() != nullptr.
 * @return The ID of the entity that contains the component.
 */
EntityId GetEntityId() const;

```

So, we can rewrite the earlier example of calling SetWorldTranslation if it was called within the same entity:

```

AZ::TransformBus::Event(
    // to the same entity
    GetEntityId(),
    &AZ::TransformBus::Events::SetWorldTranslation,
    position);

```

## EBus to the Parent Entity

Often one would attach entities to other entities in order to compose complex objects such as in Chapter 3, *Introduction to Entities and Components*. Child-parent relationships in O3DE are expressed in TransformComponent and can be accessed through its EBus, TransformBus. So if you want to move the parent entity:

```

AZ::EntityId parentId;
// Get parent entity id of the current entity
AZ::TransformBus::EventResult(parentId, GetEntityId(),
    &AZ::TransformBus::Events::GetParentId);

```

The key is GetParentId method from AZ::TransformBus:

```

class TransformInterface {
    //! Returns the entityId of the entity's parent.
    //! @return The entityId of the parent. The entityId is
    //! invalid if the entity does not have a parent.
    virtual EntityId GetParentId() { return EntityId(); }

```

Afterwards it would follow with an EBus call to SetWorldTranslation, for example:

```

AZ::TransformBus::Event(
    // to the parent entity
    parentId,
    &AZ::TransformBus::Events::SetWorldTranslation,
    position);

```

## EBus to a child Entity

TransformInterface also has a way to get all of its child entities: GetChildren.

```

class TransformInterface {
    //! Returns the entityIds of the entity's immediate children.

```

```
virtual AZStd::vector<AZ::EntityId> GetChildren()
```

Here is how it could be used:

```
AZStd::vector<AZ::EntityId> children;
AZ::TransformBus::EventResult(children, GetEntityId(),
    &AZ::TransformBus::Events::GetChildren);

// iterate over all or just call one of children
for (AZ::EntityId id: children)
{
    AZ::TransformBus::Event(
        // to a child entity
        id,
        &AZ::TransformBus::Events::SetWorldTranslation,
        position);
}
```

## EBus Event versus EventResult

You may have noticed in the examples above we used AZ::TransformBus::Event and AZ::TransformBus::EventResult. The only difference is that EventResult returns a result. The first parameter in EventResult is the return value. In a general form both calls are:

```
// EBus without a return value
{BusType}::Event({BusId}, &{Method}, {any input parameters});

// EBus with a return value
{BusType}::EventResult({return_value},
    {BusId}, &{Method}, {any input parameters});
```

### Note

BusId in the above examples were AZ::EntityId but it could be any other type as well.

---

# Chapter 9. Using AZ::TickBus

Tick: You're not going crazy. You're going sane in a crazy world!

—The Tick (TV Series 1994-1997)

## Introduction to AZ::TickBus

### Note

You can find the code for this chapter on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch09\\_oscillator](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch09_oscillator)

This chapter covers the following:

- How to use TickBus.
- Write OscillatorComponent using TickBus.
- Deeper look into Activate and Deactivate methods.
- Declaring dependencies between components of the same entity.

Aside from calling an EBus on another component, such as invoking TransformBus, we also need to know how to sign up for an EBus. A common EBus to sign up for is AZ::TickBus. It allows one to perform an action on every game frame, otherwise known as a game tick in O3DE.

A great way to learn how to use an EBus is to implement a component that uses one. This chapter will write a new C++ component: Oscillator component. It will oscillate the position of the entity it is attached to using TransformBus and TickBus.

Looking at the definition of AZ::TickBus in C:\O3DE\21.11.2\Code\Framework\AzCore\AzCore\Component\TickBus.h:

```
/**  
 * The EBus for tick notification events.  
 * The events are defined in the AZ::TickEvents class.  
 */  
typedef AZ::EBus<TickEvents>      TickBus;
```

Looking at AZ::TickEvents as mentioned in the comment above:

```
class TickEvents  
    : public AZ::EBusTraits  
{  
    ...  
    /**  
     * Signals that the application has issued a tick.  
     * @param deltaTime The delta (in seconds) from  
     *                  the previous tick and the current time.  
     * @param time The current time.  
     */  
    virtual void OnTick(float deltaTime, ScriptTimePoint time) = 0;
```

OnTick is the callback we override to get tick events. Let us take a look how one would create a component that handles it.

## OscillatorComponent

We are going to create a new component, OscillatorComponent. Here is a portion of the header file to get started.

### >Note

See previous chapters how to create a new component.

#### Example 9.1. OscillatorComponent.h snippet

```
// An example of singing up to an Ebus, TickBus in this case
class OscillatorComponent
    : public AZ::Component
    , public AZ::TickBus::Handler // for ticking events
{
public:
    // be sure this guid is unique, avoid copy-paste errors!
    AZ_COMPONENT(OscillatorComponent,
        "{302AE5A0-F7C4-4319-8023-B1ADF53E1E72}");

protected:
    // AZ::Component overrides
    void Activate() override;
    void Deactivate() override;

    // AZ::TickBus overrides
    void OnTick(float dt, AZ::ScriptTimePoint) override;

    // what other components does this component require?
    static void GetRequiredServices(
        AZ::ComponentDescriptor::DependencyArrayType& req);
    ...
};
```

### >Note

This is why I like EBuses - you can make all methods protected and nobody can talk to your component unless they go through an approved interface, its EBus. It is like having a lawyer!

## Inheriting from an EBus Handler

If you wish to sign up for an EBus events, then you have to inherit from its handler, for example: AZ::TickBus::Handler.

## AZ::Component Overrides

This is a component, thus it has to inherit from AZ::Component. Activate and Deactivate are pure virtual methods and have to be overridden and implemented. We had skimmed over these methods before, so let us spend a moment to take a deeper look at them.

## >Note

AZ::Component is defined at C:\O3DE\21.11.2\Code\Framework\AzCore\AzCore\Component\Component.h

Source code comment for `Activate()` states: “[`Activate()`] puts the component into an active state. The system calls this function once during activation of each entity that owns the component. You must override this function. The system calls a component’s `Activate()` function only if all services and components that the component depends on are present and active. Use `GetProvidedServices` and `GetDependentServices` to specify these dependencies.”

And for `Deactivate()` method: “Deactivates the component. The system calls this function when the owning entity is being deactivated. You must override this function. As the best practice, ensure that this function returns the component to a minimal footprint. The order of deactivation is the reverse of activation, so your component is deactivated before the components it depends on. The system always calls the component’s `Deactivate()` function before destroying the component. However, deactivation is not always followed by the destruction of the component. An entity and its components can be deactivated and reactivated without being destroyed. Ensure that your `Deactivate()` implementation can handle this scenario.”

In other words, `Activate` is called by O3DE when your component comes to life and is expected to do its work. Whereas `Deactivate` tells your component that it must stop its work. It may be deleted afterwards or remain unused for some time and be activated later again.

## Override EBus Events

In order to receive `OnTick` events, one has to override them. `OnTick` is a virtual method.

## Dependency Declaration

`GetRequiredServices` is a special static method out of four (4) static methods that any component may declare to define its relation to other components on the same entity. In this case, `OscillatorComponent` will express that it requires `TransformComponent` to be present on the entity. Its implementation will be as follows:

```
void OscillatorComponent::GetRequiredServices(
    AZ::ComponentDescriptor::DependencyArrayType& req)
{
    // OscillatorComponent requires TransformComponent
    req.push_back(AZ_CRC_CE("TransformService"));
}
```

**How we do know to use "TransformService" value?** The only way to know that is to look at `TransformComponent::GetProvidedServices`:

```
void TransformComponent::GetProvidedServices(
    AZ::ComponentDescriptor::DependencyArrayType& provided)
{
    provided.push_back(AZ_CRC_CE("TransformService"));
}
```

In general, a component has four different ways to express various relationships with other components by providing: `GetRequiredServices`, `GetProvidedServices`, `GetDependentServices` and `GetIncompatibleServices`. The comments in O3DE code are great at explaining what these methods do, so here they are:

- GetRequiredServices
- ```
/**  
 * Specifies the services that the component requires.  
 * The system activates the required services before it activates  
 * this component. It also deactivates the required services  
 * after it deactivates this component. If a required service is  
 * missing before this component is activated, the system returns  
 * an error and does not activate this component.  
 */
```
- GetProvidedServices
- ```
/**  
 * Specifies the services that the component provides.  
 * The system uses this information to determine when to  
 * create the component.  
 */
```
- GetDependentServices
- ```
/**  
 * Specifies the services that the component depends on, but does  
 * not require. The system activates the dependent services before  
 * it activates this component. It also deactivates the dependent  
 * services after it deactivates this component. If a dependent  
 * service is missing before this component is activated,  
 * the system does not return an error and still activates this  
 * component.  
 */
```
- GetIncompatibleServices
- ```
/**  
 * Specifies the services that the component cannot operate with.  
 * For example, if two components provide a similar service and  
 * the system cannot use the services simultaneously, each of  
 * those components would specify the other component as an  
 * incompatible service.  
 */
```

### Note

All 4 of these static methods have the same input parameter:

AZ::ComponentDescriptor::DependencyArrayType&.

## Implementing Oscillator Component

### The Design of OscillatorComponent

Usually, an oscillation means using a sin or cosine mathematical functions. However, this chapter will take a different approach. The reason is the following: in order to force an entity along the output of a

trigonometric function, we would have to save the origin and force the position of the entity on every tick. That can work for many cases.

However, imagine you wanted to be able to affect the entity's position while it was oscillating. For example, you wrote another component, ShoveComponent, which moves the entity sidewise. Would it not be cool if both hypothetical ShoveComponent and OscillatorComponent could work together? I think so, therefore the design of OscillatorComponent calculates the change in position needed for each tick and does not save the starting position.

## Activation of OscillatorComponent

Compared to an earlier example component, MyComponent, the main new feature in this component is a custom implementation of Activate and Deactivate methods.

```
void OscillatorComponent::Activate()
{
    // We must connect, otherwise OnTick() will never be called.
    // Forgetting this call is a common error in O3DE!
    AZ::TickBus::Handler::BusConnect();
}

void OscillatorComponent::Deactivate()
{
    // good practice on cleanup to disconnect
    AZ::TickBus::Handler::BusDisconnect();
}
```

MyComponent was an empty component that did nothing, so it had nothing to do when it was activated. OscillatorComponent needs to sign up to AZ::TickBus. Inheriting will not connect to the bus by default. You have to do so manually when your component is ready. The soonest that can be is in Activate or later.

### ⚠ Important

The most frequent mistake for a beginner O3DE developer is forgetting to connect to the bus! If something does not work, check if you have connected to the buses you are handling events for.

### 📝 Note

OnTick will not be called until you call BusConnect. Once you call BusDisconnect, OnTick will no longer be called.

## Handling OnTick Event

### Example 9.2. Example of handling AZ::TickBus::Handler::OnTick

```
void OscillatorComponent::OnTick(float dt, AZ::ScriptTimePoint)
{
    m_currentTime += dt;

    // get current position
    AZ::Vector3 position;
```

```
AZ::TransformBus::EventResult(position, GetEntityId(),
    &AZ::TransformBus::Events::GetWorldTranslation);

// the amount of change per tick
const float change = (dt / m_period) * m_amplitude;

// move up during the first half of the period
if (m_currentTime < m_period / 2)
{
    position.SetZ(position.GetZ() + change);
    AZ::TransformBus::Event(GetEntityId(),
        &AZ::TransformBus::Events::SetWorldTranslation,
        position);
}
// move down during the second half of the period
else if (m_currentTime < m_period)
{
    position.SetZ(position.GetZ() - change);
    AZ::TransformBus::Event(GetEntityId(),
        &AZ::TransformBus::Events::SetWorldTranslation,
        position);
}
else // reset the time to start the next cycle
{
    m_currentTime = 0;
}
}
```

We have already seen the use of `SetWorldTranslation` and `GetWorldTranslation` in prior chapters. This is their use in something practical. This is a common pattern of moving an entity. First, you acquire its current position. Second, you set the position to the desired value.

## Summary

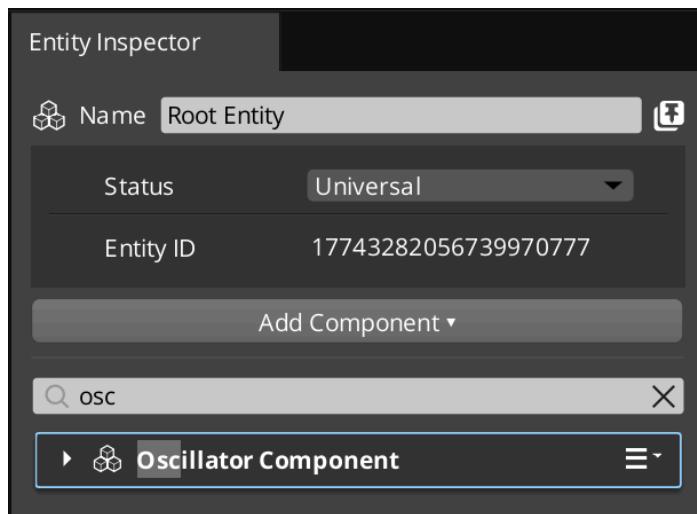
### Note

You can find the code for this chapter on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch09\\_oscillator](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch09_oscillator)

This chapter showed an example how to connect to an EBus, `AZ::TickBus`, and use it in conjunction with `TransformBus` in order to move an entity.

In order to use this component attached it to **Root Entity**.



Oscillator component attached to an entity

### Example 9.3. OscillatorComponent.h full listing

```
#pragma once
#include <AzCore/Component/Component.h>
#include <AzCore/Component/TickBus.h>

namespace MyProject
{
    // An example of singing up to an Ebus, TickBus in this case
    class OscillatorComponent
        : public AZ::Component
        , public AZ::TickBus::Handler // for ticking events
    {
    public:
        // be sure this guid is unique, avoid copy-paste errors!
        AZ_COMPONENT(OscillatorComponent,
                     "{302AE5A0-F7C4-4319-8023-B1ADF53E1E72}");

    protected:
        // AZ::Component overrides
        void Activate() override;
        void Deactivate() override;

        // AZ::TickBus overrides
        void OnTick(float dt, AZ::ScriptTimePoint) override;

        // Provide runtime reflection, if any
        static void Reflect(AZ::ReflectContext* reflection);

        // what other components does this component require?
        static void GetRequiredServices(
            AZ::ComponentDescriptor::DependencyArrayType& req);

    private:
        float m_period = 3.f;
    };
}
```

```
    float m_currentTime = 0.f;
    float m_amplitude = 10.f;
}
}
```

#### Example 9.4. OscillatorComponent.cpp full listing

```
#include "OscillatorComponent.h"
#include <AzCore/Serialization/EditContext.h>
#include <AzCore/Component/TransformBus.h>

using namespace MyProject;

void OscillatorComponent::Activate()
{
    // We must connect, otherwise OnTick() will never be called.
    // Forgetting this call is the common error in O3DE!
    AZ::TickBus::Handler::BusConnect();
}

void OscillatorComponent::Deactivate()
{
    // good practice on cleanup to disconnect
    AZ::TickBus::Handler::BusDisconnect();
}

void OscillatorComponent::OnTick(float dt, AZ::ScriptTimePoint)
{
    m_currentTime += dt;

    // get current position
    AZ::Vector3 position;
    AZ::TransformBus::EventResult(position, GetEntityId(),
        &AZ::TransformBus::Events::GetWorldTranslation);

    // the amount of change per tick
    const float change = (dt / m_period) * m_amplitude;

    // move up during the first half of the period
    if (m_currentTime < m_period / 2)
    {
        position.SetZ(position.GetZ() + change);
        AZ::TransformBus::Event(GetEntityId(),
            &AZ::TransformBus::Events::SetWorldTranslation,
            position);
    }
    // move down during the second half of the period
    else if (m_currentTime < m_period)
    {
        position.SetZ(position.GetZ() - change);
        AZ::TransformBus::Event(GetEntityId(),
            &AZ::TransformBus::Events::SetWorldTranslation,
            position);
    }
}
```

```
    else // reset the time to start the next cycle
    {
        m_currentTime = 0;
    }
}

void OscillatorComponent::Reflect(AZ::ReflectContext* reflection)
{
    auto sc = azrtti_cast<AZ::SerializeContext*>(reflection);
    if (!sc) return;
    sc->Class<OscillatorComponent, Component>()
        ->Version(1);

    AZ::EditContext* ec = sc->GetEditContext();
    if (!ec) return;
    using namespace AZ::Edit::Attributes;
    // reflection of this component for O3DE Editor
    ec->Class<OscillatorComponent>("Oscillator Component",
        "[oscillates the entity]")
        ->ClassElement(AZ::Edit::ClassElements::EditorData, " ")
            ->Attribute(AppearsInAddComponentMenu, AZ_CRC_CE("Game"))
            ->Attribute(Category, "My Project");
}

void OscillatorComponent::GetRequiredServices(
    AZ::ComponentDescriptor::DependencyArrayType& req)
{
    // OscillatorComponent requires TransformComponent
    req.push_back(AZ_CRC_CE("TransformService"));
}
```

---

## **Part IV. Introduction to Component Reflection and Prefabs**

---

## Table of Contents

10. Configuring Components in the Editor .....	81
Introduction .....	81
Component Properties in the Editor .....	81
Layers of Reflection .....	82
Summary .....	87
11. Introduction to Prefabs .....	88
Introduction .....	88
What are Prefabs? .....	88
Creating a Prefab .....	89
Editing Prefabs .....	91
Prefab is also a File .....	92
Spawning Prefabs .....	92
Summary .....	96

---

# Chapter 10. Configuring Components in the Editor

## Note

The source code for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch10\\_reflection](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch10_reflection)

## Introduction

This chapter will cover the following topics:

- How to provide editor configuration for a component.
- What is Serialization Context?
- What is Edit Context?

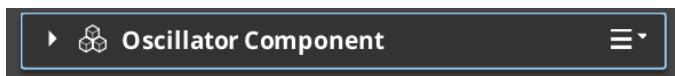
Chapter 9, *Using AZ::TickBus* implemented `OscillatorComponent`. It certainly accomplished the task. However, once you begin to test it in the editor with **CTRL+G** you will notice one pain point: if you did not like the behavior and wanted to modify it, you would have to go back to C++, make the change, re-compile the code, re-launch the Editor to finally see if your change achieved what you were looking for.

That is far too much effort and time to change one variable. Specifically, `OscillatorComponent` has two variables that would make sense to expose to the Editor: period and amplitude.

```
class OscillatorComponent
{
...
    float m_period = 3.f;
    float m_amplitude = 10.f;
};
```

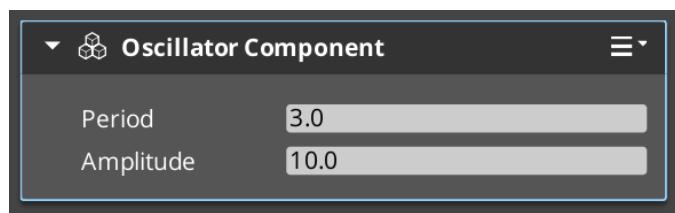
## Component Properties in the Editor

So far the component we have implemented exposed nothing to the Editor.



A component with no editor parameters

We can modify its reflection, so that we can change some of its behavior in the Editor. By exposing period and amplitude values to the Editor, Entity Inspector will show the component with two modifiable fields.



This way you would be able to enter a new value and test it immediately without re-compiling and restarting the Editor. This is possible to accomplish with O3DE reflection in `Reflect` method of a component.

## Layers of Reflection

The magic lies in `OscillatorComponent::Reflect`. As of last chapter, it provided no extra configuration for the Editor beyond the basic registration.

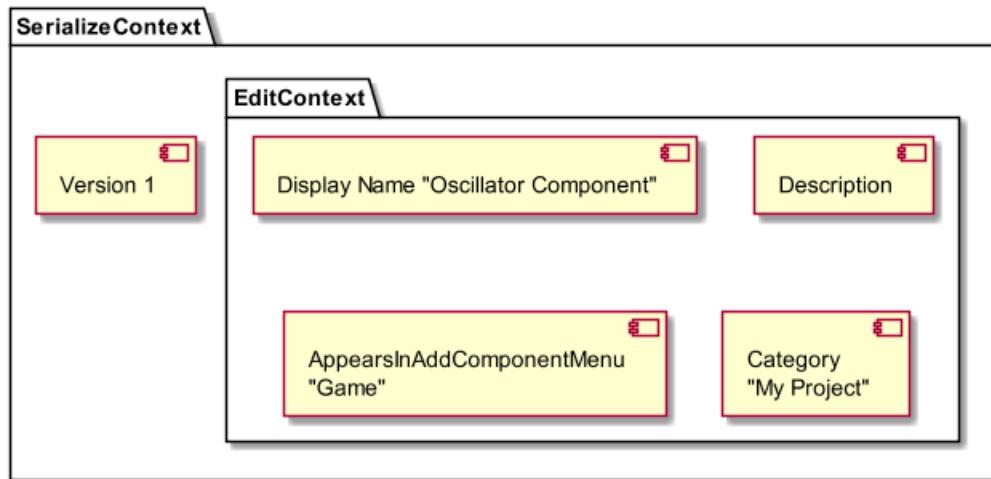
```
void OscillatorComponent::Reflect(AZ::ReflectContext* reflection)
{
    auto sc = azrtti_cast<AZ::SerializeContext*>(reflection);
    if (!sc) return;
    sc->Class<OscillatorComponent, Component>()
        ->Version(1);

    AZ::EditContext* ec = sc->GetEditContext();
    if (!ec) return;
    using namespace AZ::Edit::Attributes;
    // reflection of this component for O3DE Editor
    ec->Class<OscillatorComponent>("Oscillator Component",
        "[oscillates the entity]")
        ->ClassElement(AZ::Edit::ClassElements::EditorData, "")
            ->Attribute(AppearsInAddComponentMenu, AZ_CRC("Game"))
            ->Attribute(Category, "My Project");
}
```

Reflection in O3DE is described in layers, called contexts. The first and main layer is `SerializeContext`. There are two layers built on top of it: `EditContext` (for the Editor) and `BehaviorContext` (for scripting).

At the moment, the reflection data of `OscillatorComponent` can be represented like this:

**Figure 10.1. Basic Serialization**



## SerializeContext

### Example 10.1. AzCore\Serialization\SerializeContext.h

```
/***
 * Serialize context is a class that manages information
 * about all reflected data structures. You will use it
 * for all related information when you declare your data
 * for serialization.
 * In addition it will handle data version control.
 */
class SerializeContext
```

SerializeContext is a global storage of reflected information of data structures. In this chapter, we are going to limit ourselves to the discussion of reflecting custom components and their data into it. The starting point of reflecting a component is to declare it as a class within SerializeContext:

```
sc->Class<OscillatorComponent, Component>()
```

Class follows a programming method, called Builder pattern, where you can continuously invoke methods on their return values, because their return values are always ClassBuilder.

```
/***
 * Internal structure to maintain class information while
 * we are describing a class. User should call variety of
 * functions to describe class features and data.
 */
class ClassBuilder
```

ClassBuilder has a lot of methods but we will focus on the ones that we used in OscillatorComponent. Those methods are also the most common.

#### ClassBuilder: Version.

```
sc->Class<OscillatorComponent>()
    ->Version(1);
```

This specifies the version of your component in the context of reflection. You can use it as a form of version control. Whenever you make significant changes to a component during development, it is advisable to increase the version.

#### Note

Here is Version() declaration:

```
/// Add version control to the structure with optional converter.
/// If not controlled a version of 0 is assigned.
ClassBuilder* Version(
    unsigned int version,
    VersionConverter converter = nullptr);
```

## EditContext

### Example 10.2. AzCore\Serialization>EditContext.h

```
/**  
 * EditContext is bound to serialize context. It uses it for data  
 * manipulation. Its role is to be an abstract way to generate and  
 * describe how a class should be edited.  
 */  
class EditContext
```

#### ! Important

Before you can describe a class in `EditContext` you must describe it in `SerializeContext`. A similar class builder pattern is used in `EditContext` as well.

##### EditContext: Display Name and Description.

```
ec->Class<OscillatorComponent>("Oscillator Component",  
                                  "[oscillates the entity]")
```

`Class` is a templated method that declares a component for the Editor. The first parameter is the name as it will appear in **Add Component** menu. It does not need to match the exact class name and it may include spaces. The second is the description that will show up as a tooltip when you hover over its UI element in the Editor.

#### ! Note

You can find declaration for `Class<T>()` in `EditContext.h`:

```
template<class T>  
EditContext::ClassBuilder  
EditContext::Class(const char* displayName,  
                  const char* description)
```

Next up are various Editor specific attributes of a class. But before you can specify `Attribute`, you have to declare its `EditorData`:

```
->ClassElement(AZ::Edit::ClassElements::EditorData, "")
```

That tells the serialization system that we will describe editor specific data.

#### ! Important

Without the following `Attribute` the component will NOT show up the Editor!

```
->Attribute(AppearsInAddComponentMenu, AZ_CRC("Game"))
```

The value "Game" is a bit unfortunate as it really means that this component will appear in the Editor's **Add Component** menu. If you wish to hide your component from the Editor, you can omit this attribute completely.

#### ! Note

You may wonder if there are any other **Add Component** menus beside the Editor. Yep, there are a few more types, such as System components ("System"), Level components ("Level") and User Interface components ("UI") and Canvas User Interface components ("CanvasUI"). For example, a system component can be specified with:

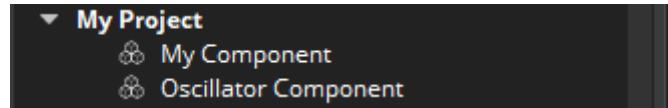
```
->Attribute(AppearsInAddComponentMenu, AZ_CRC("System"))
```

These components will be discussed in later chapters.

Another useful attribute is **Category**. It allows you to group your components together in **Add Component menu**.

```
->Attribute(Category, "My Project")
```

This is the attribute that makes OscillatorComponent show up under **My Project** in the Editor.

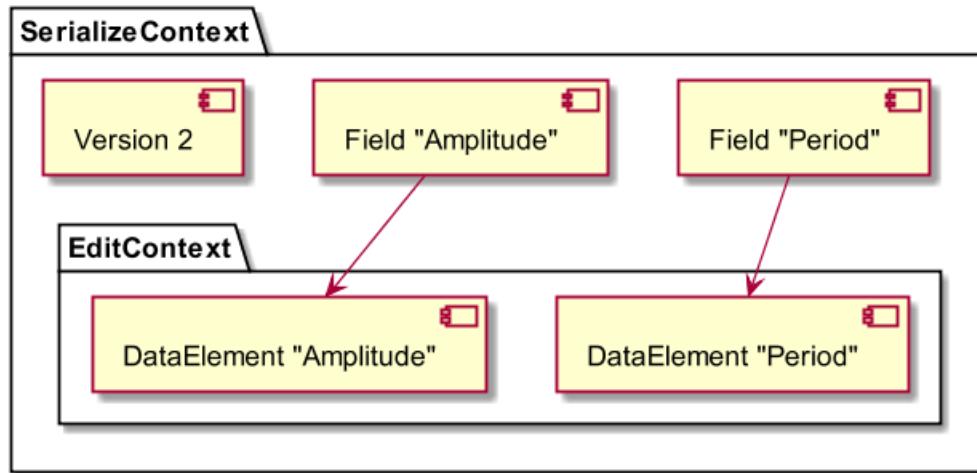


Oscillator component in Add Component menu

## Configurable Fields in the Editor

This chapter extends the reflection of OscillatorComponent to include its two member variables: `m_amplitude` and `m_period`. We are also incrementing the version from 1 to 2 for version control purposes.

**Figure 10.2. SerializeContext and EditContext**



### Important

Before you can declare a configurable field in the Editor, you have to define it in `SerializeContext`. Also notice that the names have to be the same. In this case, different contexts provide additional descriptions of the same field. In a sense, `SerializeContext` declares and creates a field and `EditContext` expands that definition with Editor configuration.

### Example 10.3. Reflect() with Editor Configuration

```
void OscillatorComponent::Reflect(AZ::ReflectContext* reflection)
{
    auto sc = azrtti_cast<AZ::SerializeContext*>(reflection);
```

```
if (!sc) return;
sc->Class<OscillatorComponent, Component>()
    // serialize m_period
    ->Field("Period", &OscillatorComponent::m_period)
    // serialize m_amplitude
    ->Field("Amplitude", &OscillatorComponent::m_amplitude)
    ->Version(2);

AZ::EditContext* ec = sc->GetEditContext();
if (!ec) return;
using namespace AZ::Edit::Attributes;
// reflection of this component for O3DE Editor
ec->Class<OscillatorComponent>("Oscillator Component",
    "[oscillates the entity]")
->ClassElement(AZ::Edit::ClassElements::EditorData, "")
    ->Attribute(AppearsInAddComponentMenu, AZ_CRC("Game"))
    ->Attribute(Category, "My Project")
    // expose the setting to the editor
    ->DataElement(nullptr, &OscillatorComponent::m_period,
        "Period", "[the period of oscillation]")
    // expose the setting to the editor
    ->DataElement(nullptr, &OscillatorComponent::m_amplitude,
        "Amplitude", "[the height of oscillation]");
}
```

Compared to the previous Reflect implementation, there are two new changes: Field and DataElement.

## Field

Recall that OscillatorComponent has a member variable: m\_period. Given that, it can be declared to be a serialized field with:

```
->Field("Period", &OscillatorComponent::m_period)
```

The first parameter for Field is a const char\* and the second is a point to the member variable.

## DataElement

The counterpart for the Editor (and Project Configurator) is DataElement:

```
->DataElement(Default, &OscillatorComponent::m_period,
    "Period", "[the period of oscillation"])
```

The first element is a custom UI identifier. Generally, a AZ::Edit::UIHandlers::Default (or simply nullptr) works for most use cases. That covers regular single values, such as integer, floats, strings and vectors (AZ::Vector3 and so on). If you have to explicitly specify the default UI handler then you can refer to the available list of options under AZ::Edit::UIHandlers.

### Example 10.4. AzCore\Serialization>EditContextConstants.inl

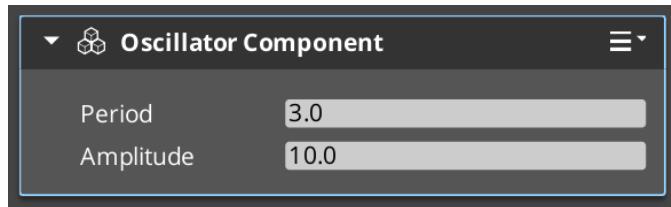
```
namespace UIHandlers
{
    /**
     * Helper for explicitly designating the default UI handler.
```

```
* i.e. DataElement(DefaultHandler, field, ...)  
*/  
const static AZ::Crc32 Default = 0;  
  
const static AZ::Crc32 Button = AZ_CRC("Button", 0x3a06ac3d);  
const static AZ::Crc32 Color = AZ_CRC("Color", 0x665648e9);  
...
```

The second parameter to DataElement is a pointer to a member variable of the component. The third parameter is the name that was assigned to the member variable with Field earlier. The last parameter is the description.

## Summary

With the changes in this chapter, the component is now configurable in the Editor.



Configurable options in the Editor

### Note

The source code for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch10\\_reflection](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch10_reflection)

You can quickly test the changes by modifying the values and using the shortcut **CTRL+G** to test the behavior right away in the Editor. (**ESC** exits the play in the Editor.)

---

# Chapter 11. Introduction to Prefabs

## Introduction

### Note

You can find the code and assets for this chapter on GitHub at:

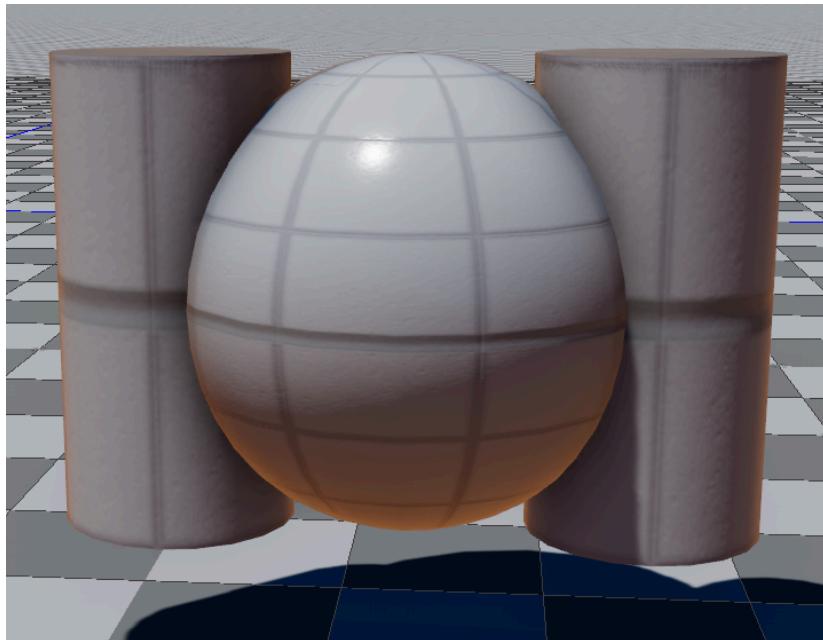
[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch11\\_prefabs](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch11_prefabs)

This chapter covers:

- What are prefabs?
- What are spawnables?
- Spawning prefabs.
- Spawning prefabs from C++.

## What are Prefabs?

A prefab is a group of entities. There is more to it but that is the essence. Think back to the composite object we made from 4 entities in chapter "Chapter 3, *Introduction to Entities and Components*".



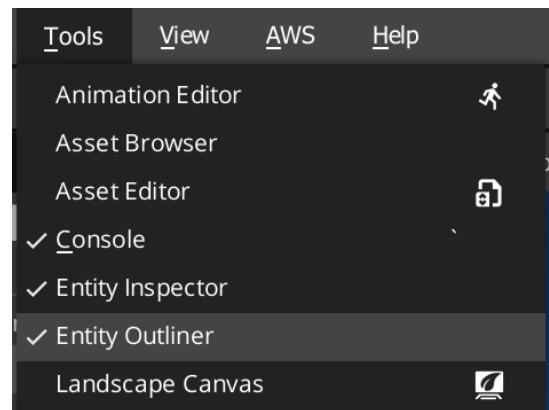
A group of entities

It would be tedious and error prone to manually re-create copies of it, if we wanted to have many of such objects. But if we turn this group of entities into a prefab, then we can easily duplicate it over and over again.

# Creating a Prefab

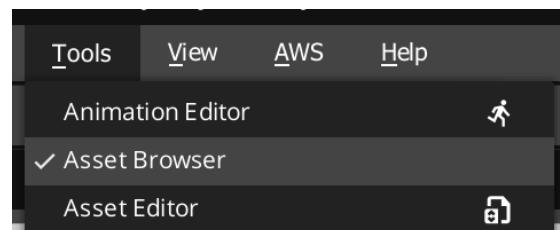
This section will cover how to create a prefab in the Editor. Before we get started we need two Editor panels that are useful when working with prefabs: **Entity Outliner** and **Asset Browser**. You can launch both from **Tools** menu of the Editor.

**Figure 11.1. How to bring up Entity Outliner**



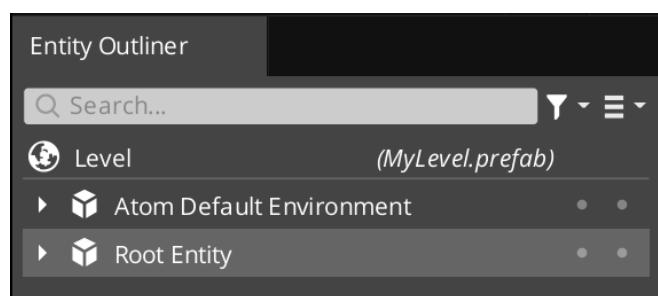
Tools→ Entity Outliner

**Figure 11.2. How to bring up Asset Browser**



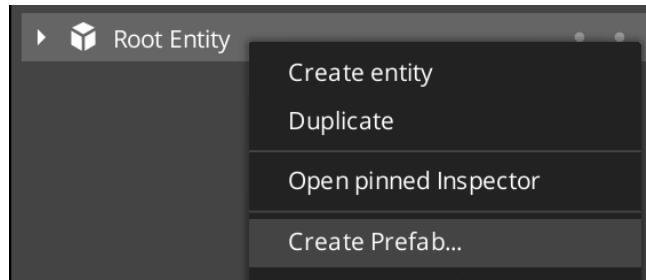
Tools→ Asset Browser

Previous chapter already created a complex object with **Root Entity** as its parent entity.



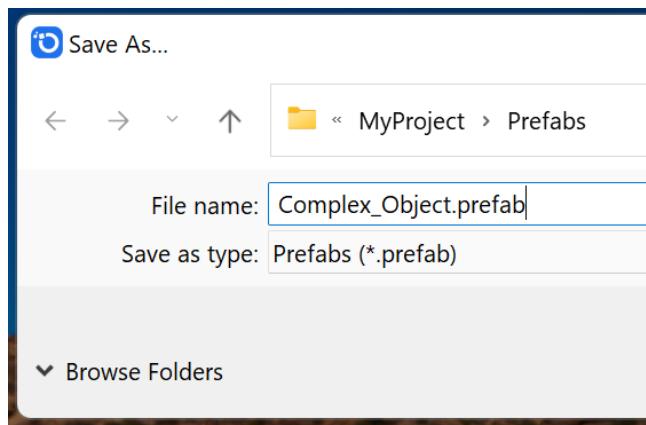
Current scene

Right click on **Root Entity** and choose **Create Prefab...**



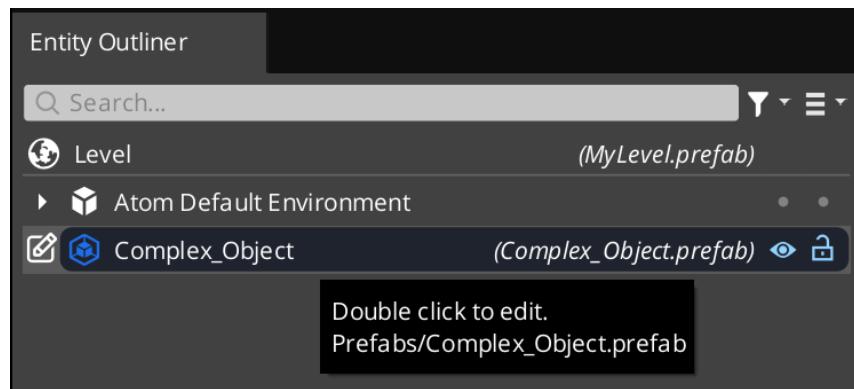
Right click on Root Entity → **Create Prefab**

That will bring up a **Save As...** dialog where you can choose the file name of the prefab.



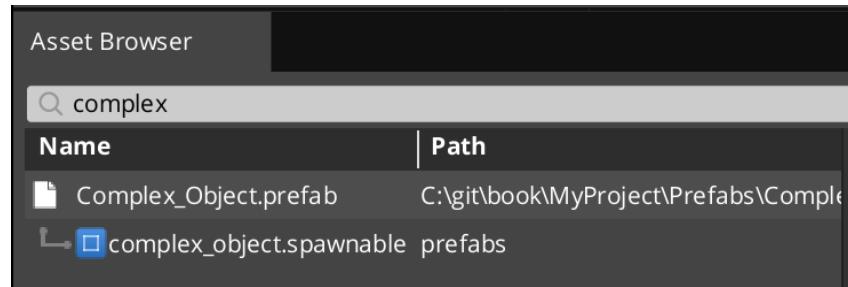
Save as Complex\_Object.prefab

The prefab should be located at C:\git\book\MyProject\Prefabs\Complex\_Object.prefab. **Entity Outliner** will change from "Root Entity" to the name of the prefab and turn the icon blue. The tooltip will state that it is now part of a prefab.



A prefab instance

The prefab will also appear in Asset Browser panel.



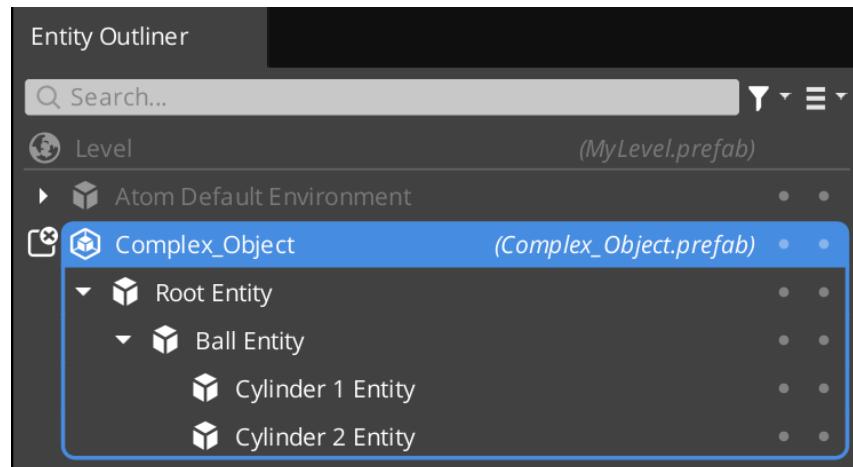
A prefab in Asset Browser

### Note

A *spawnable* is a derivative product of a prefab that can be spawned at runtime, whereas a *prefab* is a design time product. We will look at spawnables later in this chapter.

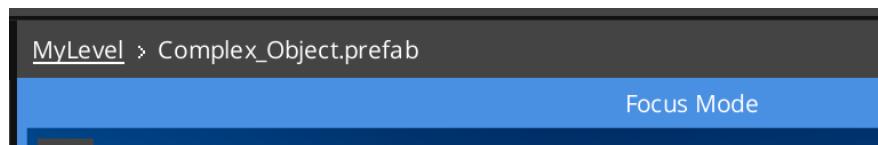
## Editing Prefabs

Notice that the entity structure within a prefab is now hidden. You must enter the edit mode of the prefab to see its entities. You can do that by double clicking on it or clicking the icon to the left of the prefab.



Editing a prefab

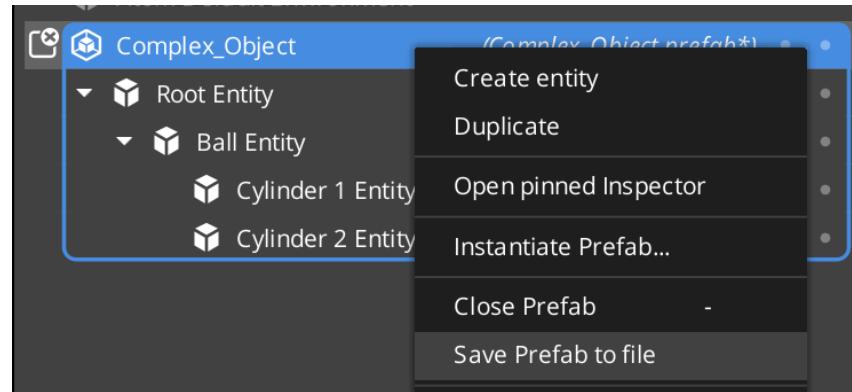
When you are editing a prefab, the Editor viewport turns to focus mode, where only changes to prefab entities are allowed until you exit prefab editing mode.



Once you make a change to prefab, its name will be marked with \* in Entity Outliner.



To save the prefab, right click on it in Entity Outliner and choose *Save Prefab to file*.



### Tip

You can also save the entire level and all the prefabs in it with **CTRL+S**.

You can exit the prefab edit mode with **ESC** or click on the icon to the left of the prefab title.

## Prefab is also a File

### Note

JSON serialization of a prefab is an internal matter of O3DE engine. You should not need to worry about the way a prefab is written to `Complex_Object.prefab` but occasionally when things go wrong for an unexplained reason it is useful to know the basics, so you can peak at the structure. Sometimes that will help you solve a bug.

A prefab is saved to disk as a JSON file. If you were to open it up you could search its contents for one of the components we created and attached to **Root Entity**, `OscillatorComponent`.

#### Example 11.1. Complex\_Object.prefab snippet with Root Entity entity

```
"Entities": {  
    "Entity_[621262670860)": {  
        "Id": "Entity_[621262670860]",  
        "Name": "Root Entity",
```

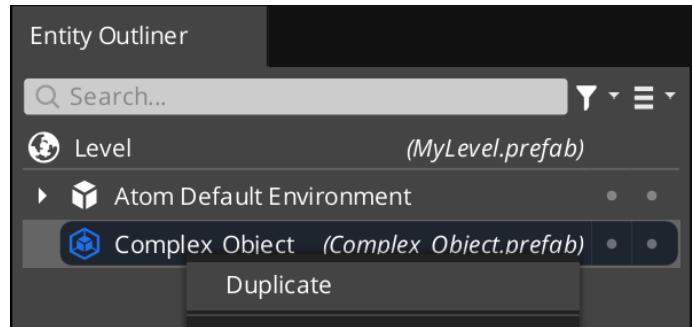
Under this element, there will be a reference to `OscillatorComponent`.

#### Example 11.2. Complex\_Object.prefab snippet with OscillatorComponent

```
"Component_[12076214170871786334)": {  
    "$type": "GenericComponentWrapper",  
    "Id": 12076214170871786334,  
    "m_template": {  
        "$type": "OscillatorComponent"  
    }  
},
```

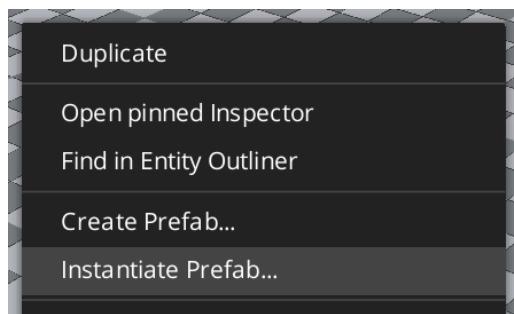
## Spawning Prefabs

In the Editor, you can add more prefab instances by duplicating existing prefabs in the level.



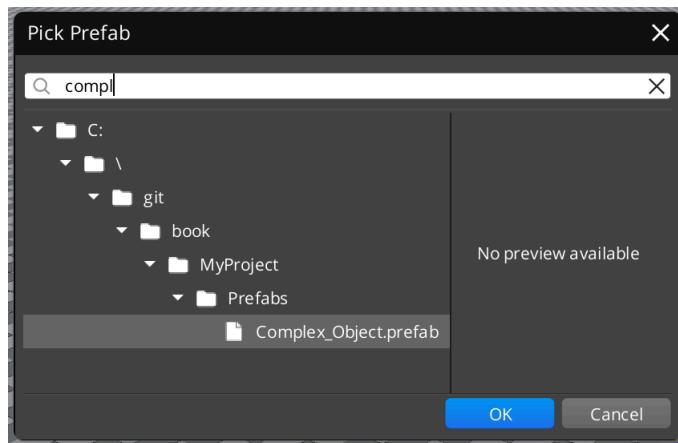
Duplicate prefab with Right Click → Duplicate

Or by instantiating one in the level with right click and then "Instantiate Prefab..."



Instantiate Prefab

And then selecting the prefab of your choice.



Select a prefab

## Spawning Prefabs from C++

If you want more power and control, then we must turn to C++ (or scripting). The API are the spawn methods from `AzFramework::SpawnableEntitiesInterface::Get()`, which is defined in `AzFramework/Spawnable/SpawnableEntitiesInterface.h`. Here is how to spawn all entities from a spawnable product of a prefab:

```
AzFramework::SpawnableEntitiesInterface::Get() -> SpawnAllEntities(...);
```

### Example 11.3. Method `SpawnAllEntities`

```
///! Spawn instances of all entities in the spawnable.  
///! @param ticket Stores the results of the call. Use this ticket  
///! to spawn additional entities or to despawn them.  
///! @param optionalArgs Optional additional arguments,  
///! see SpawnAllEntitiesOptionalArgs.  
virtual void SpawnAllEntities(  
    EntitySpawnTicket& ticket,  
    SpawnAllEntitiesOptionalArgs optionalArgs = {} ) = 0;
```

**What is a spawnable?** A spawnable is a runtime product of a prefab. A spawnable is the product that you can spawn in your game at runtime. You can see that each prefab has a corresponding spawnable in **Asset Browser**.



For `Complex_Object.prefab` there is `complex_object.spawnable` that is generated for you by the Asset Processor.

## MySpawnerComponent

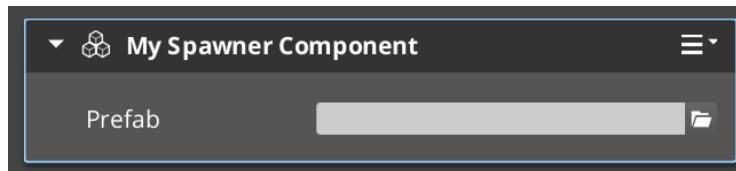
### Note

The source code and the level for this section can be found at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch11\\_prefabs](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch11_prefabs)

Here is the complete example of how to spawn a prefab from a component. We are going to build a component that will have a field that can be populated in the Editor and will spawn one instance of it when the entity is activated.

### Example 11.4. MySpawnerComponent



API of `SpawnAllEntities` requires two additional concepts to understand.

1. `AzFramework::Spawnable` is the product to spawn by `SpawnAllEntities`.
2. However, it needs to be wrapped with `AzFramework::EntitySpawnTicket`. This ticket will also hold the reference to the entities that will be spawned by spawn methods of `AzFramework::ISpawnableEntitiesInterface`. When the ticket goes out of scope, those entities are deactivated and destroyed.

With that in mind, here are the steps to spawn entities from a spawnable:

1. Create an asset of a spawnable. We will assign it from the Editor.

```
AZ::Data::Asset<AzFramework::Spawnable> m_spawnableAsset;
```

2. Create a ticket by passing the spawnable into it.

```
AzFramework::EntitySpawnTicket m_ticket;
```

```
m_ticket = AzFramework::EntitySpawnTicket(m_spawnableAsset);
```

3. Now if we pass this ticket into `SpawnAllEntities` it will spawn the entities from that spawnable. But there is one gotcha, *where* will those entities be spawned at? Unless, we specify the location, they will be spawned relative to the origin. `SpawnAllEntities` does not specify a way to specify the location, so how do we do that? The answer is in `AzFramework::SpawnAllEntitiesOptionalArgs`. It provides a way to modify the entities before they are activated via the member `m_preInsertionCallback`. Here is how to place the root entity where you want it.

### Example 11.5. Spawn entities at a location

```
using namespace AzFramework;
auto preSpawnCallback = [world](
    EntitySpawnTicket::Id /*ticketId*/,
    SpawnableEntityContainerView view)
{
    const AZ::Entity* e = *view.begin();
    if (auto* tc = e->FindComponent<TransformComponent>())
    {
        tc->SetWorldTM(world);
    }
};

SpawnAllEntitiesOptionalArgs optionalArgs;
optionalArgs.m_preInsertionCallback = AZStd::move(preSpawnCallback);
```

`AzFramework::SpawnableEntityContainerView` contains the entities that are about to be spawned and activated. Callback `m_preInsertionCallback` is called before the entities are activated.

4. With this configuration, we are ready to request the entities to be spawned.

```
AzFramework::SpawnableEntitiesInterface::Get() ->
    SpawnAllEntities(m_ticket, AZStd::move(optionalArgs));
```

### Note

The entities may not spawn immediately, so be sure that your code is ready to received the callback and the entities at the next or later game tick.

### Tip

If your prefab is built with an entity that contains all other entities as we did in Chapter 3, *Introduction to Entities and Components*, then you only need to assign the location of the root entity. Other child entities will position themselves relative to their parent's transform.

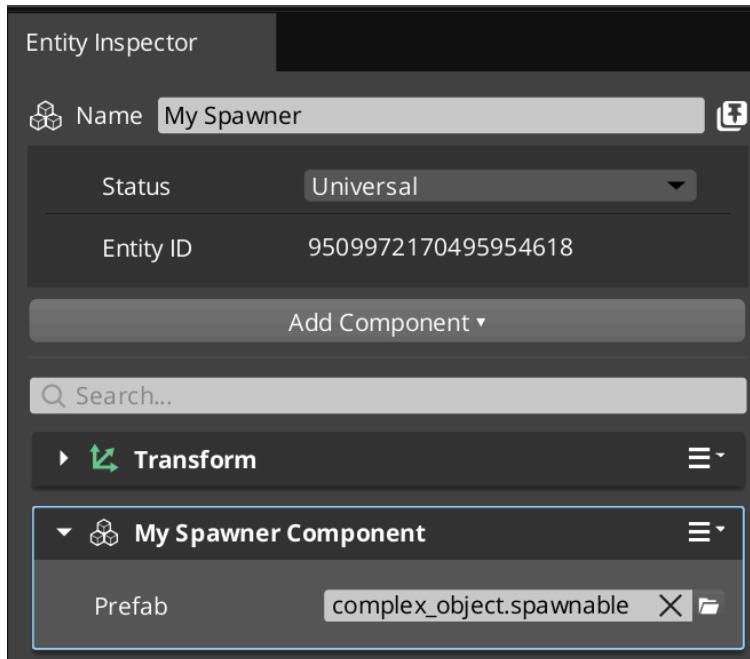
# Summary

## Note

You can find the code and assets for this chapter on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch11\\_prefabs](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch11_prefabs)

### Example 11.6. MySpawnerComponent with Complex\_Object.prefab



MySpawnerComponent reflects AZ::Data::Asset<AzFramework::Spawnable> to the Editor. Assign it Complex\_Object.spawnable. Here is the entire source code.

### Example 11.7. MySpawnerComponent.h

```
#pragma once
#include <AzCore/Component/Component.h>
#include <AzFramework/Spawnable/SpawnableEntitiesInterface.h>

namespace MyProject
{
    // An example of spawning prefab from C++.
    class MySpawnerComponent : public AZ::Component
    {
        public:
            AZ_COMPONENT(MySpawnerComponent,
                         "{F75160EB-CB82-4A41-8AB0-68AD43B9625B}");

            // AZ::Component overrides
            void Activate() override;
            void Deactivate() override {};
    };
}
```

```
// Provide runtime reflection, if any
static void Reflect(AZ::ReflectContext* reflection);

private:
    AZ::Data::Asset<AzFramework::Spawnable> m_spawnableAsset;
    AzFramework::EntitySpawnTicket m_ticket;
};

}
```

### Example 11.8. MySpawnerComponent.cpp

```
#include "MySpawnerComponent.h"
#include <AzCore/Asset/AssetSerializer.h>
#include <AzCore/Serialization/EditContext.h>

using namespace MyProject;

void MySpawnerComponent::Activate()
{
    using namespace AzFramework;
    AZ::Transform world = GetEntity()->GetTransform()->GetWorldTM();
    m_ticket = EntitySpawnTicket(m_spawnableAsset);

    auto cb = [world](
        EntitySpawnTicket::Id /*ticketId*/,
        SpawnableEntityContainerView view)
    {
        const AZ::Entity* e = *view.begin();
        if (auto* tc = e->FindComponent<TransformComponent>())
        {
            tc->SetWorldTM(world);
        }
    };
}

if (m_ticket.IsValid())
{
    SpawnAllEntitiesOptionalArgs optionalArgs;
    optionalArgs.m_preInsertionCallback = AZStd::move(cb);
    SpawnableEntitiesInterface::Get()->SpawnAllEntities(
        m_ticket, AZStd::move(optionalArgs));
}

void MySpawnerComponent::Reflect(AZ::ReflectContext* reflection)
{
    auto sc = azrtti_cast<AZ::SerializeContext*>(reflection);
    if (!sc) return;

    sc->Class<MySpawnerComponent, Component>()
        ->Field("Prefab", &MySpawnerComponent::m_spawnableAsset)
        ->Version(1);

    AZ::EditContext* ec = sc->GetEditContext();
```

```
if (!ec) return;

using namespace AZ::Edit::Attributes;
ec->Class<MySpawnerComponent>("My Spawner Component",
    "[An example of spawning prefab from C++]")
->ClassElement(AZ::Edit::ClassElements::EditorData, "")
    ->Attribute(AppearsInAddComponentMenu, AZ_CRC("Game"))
    ->Attribute(Category, "My Project")
    ->DataElement(nullptr, &MySpawnerComponent::m_spawnableAsset,
        "Prefab",
        "Spawn this prefab once when this entity activates")
;
}
```

---

## **Part V. Modularity in O3DE**

---

## Table of Contents

12. What is a Gem? .....	101
Introduction .....	101
Creating a New Gem .....	101
Adding Gem to the Project .....	102
Where Does a Project List the Gems It Uses? .....	104
File Structure of a Gem .....	104
Summary .....	109
13. Gem: Set Window Position .....	110
Introduction and Motivation .....	110
Create WindowPosition Gem .....	110
Console Subsystem .....	110
Console Command Callback .....	111
Conclusion and Source Code .....	113
14. Enabling NvCloth Gem .....	115
Introduction .....	115
Adding NvCloth Gem .....	115
Using NvCloth Gem .....	117
Adding Your Assets to Gems .....	118
Summary .....	119

---

# Chapter 12. What is a Gem?

## Introduction

This chapter covers the following topics:

- What is a Gem?
- Creating a new Gem.
- Gem's structure.

The gems system was developed to make it easy to share code between projects. Gems are reusable packages of module code and/or assets which can be easily added to or removed from an O3DE project.

O3DE engine was built to be modular. That is achieved by using gems. A gem is a collection of code and assets (or only assets without any native C++ code). The intention of a gem system is that a developer would have a collection of gems to pick from, as well as be able to write their own gems that they may use in their projects. A gem is a stand-alone piece of code with assets and can be re-used in multiple projects.

For example, if one were to write a gem that positions the game window at launch time, then you could re-use this functionality in different projects without having to copy paste the code. In fact, Chapter 13, *Gem: Set Window Position* will write such a gem!

## Creating a New Gem

### Note

The code for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch12\\_new\\_gem](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch12_new_gem)

O3DE comes with a lot of gems. They are all located under your O3DE install path or a GitHub clone. Here is my install location: C:\O3DE\21.11.2\Gems. For example, there are EMotionFX, Multiplayer, NvCloth gems and many others. These are core O3DE gems. If you were to write your own gem, I suggest placing it alongside with your project. So far we have created a project at C:\git\book\MyProject. A good option is to place my gems under C:\git\book\Gems. For example, in this chapter I will create a gem called MyGem and place it at C:\git\book\Gems\MyGem. The specific location of gems is entirely up to you. I will show you how to configure your project to see a gem from anywhere on your system.

### Note

The official O3DE documentation for creating a new gem can be found here:

<https://www.o3de.org/docs/user-guide/programming/gems/>

Let us create a brand new gem and see what we get out of the box and its overall structure. To create a new gem, I will use the command line interface of scripts\o3de.bat:

```
C:\O3DE\21.11.2\scripts\o3de.bat create-gem -gp C:\git\book\Gems\MyGem
```

## Tip

The script is smart enough to figure out that the name of the game should be MyGem. However, you can also specify the name yourself. Run the command with **-h** switch to view all available options.

```
C:\O3DE\21.11.2\scripts\o3de.bat create-gem -h
```

Gem name can be optionally specified with **-gn**.

# Adding Gem to the Project

You can use O3DE Project Manager to add and remove gems. You can find it in your O3DE installation, for example: C:\O3DE\21.11.2\bin\Windows\profile\Default\o3de.exe. The official guide on using O3DE user interface to add and remove gems can be found here:

<https://www.o3de.org/docs/user-guide/project-config/add-remove-gems/>

However, it is useful to understand what is going on behind the scenes. There are two configuration files that need to be modified to add a new gem to a project.

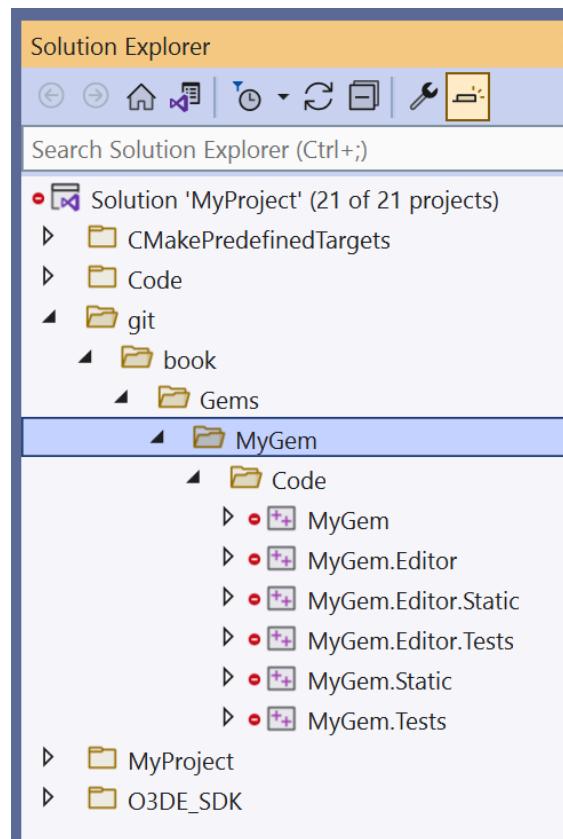
1. First file is the project configuration file at C:\git\book\MyProject\project.json.

```
{
    "project_name": "MyProject",
    "origin": "...",
    "license": "...",
    "display_name": "MyProject",
    "summary": "A short description of MyProject.",
    "canonical_tags": [
        "Project"
    ],
    "user_tags": [
        "MyProject"
    ],
    "icon_path": "preview.png",
    "engine": "o3de-sdk",
    "external_subdirectories": [
    ]
}
```

Property array *external\_subdirectories* is responsible for making a gem visible to the project. The following example adds the path to *MyGem* to the project.

```
"external_subdirectories": [
    "C:/git/book/Gems/MyGem"
]
```

Now if you build **ZERO\_TARGET** in Visual Studio, this gem will appear in the solution.



## ⚠️ Important

This does not enable the gem just yet! The gem is now visible by the project and will be built but one more step remains.

2. The second file enables the gem by its name in the project file: C:\git\book\MyProject\Code\enabled\_gems.cmake.

```
set(ENABLED_GEMS
    MyProject
    ...
    MyGem      # new
)
```

Where does the name *MyGem* come from? It comes from the gem's json configuration file "gem\_name" property in C:\git\book\Gems\MyGem\gem.json.

```
{
    "gem_name": "MyGem",
    ...
}
```

With these changes the new gem will be compiled as part of the project and will be enabled at runtime.

## Tip

You can confirm that the gem is enabled by placing a breakpoint in `C:\git\book\Gems\MyGem\Code\Source\MyGemSystemComponent.cpp` at `MyGemSystemComponent::Activate`. If you launch the Editor and the breakpoint gets hit then the gem has been successfully enabled.

# Where Does a Project List the Gems It Uses?

The entire list of enabled gem for our project is in `C:\git\book\MyProject\Code\enabled_gems.cmake`.

### Example 12.1. `enabled_gems.cmake`

```
set(ENABLED_GEMS
    MyProject
    Atom
    AudioSystem
    AWSCore
    CameraFramework
    DebugDraw
    EditorPythonBindings
    EMotionFX
    GameState
    ImGui
    LandscapeCanvas
    LyShine
    Multiplayer
    PhysX
    PrimitiveAssets
    SaveData
    ScriptCanvasPhysics
    ScriptEvents
    StartingPointInput
    TextureAtlas
    WhiteBox

    MyGem      # new
)
```

These are the default gems that are enabled when you create a project with default settings. **MyProject** gem is the project gem, which includes the project code at `C:\git\book\MyProject\Code`. Then there is a list of various core O3DE gems. In this chapter we just added *MyGem* in the last line.

# File Structure of a Gem

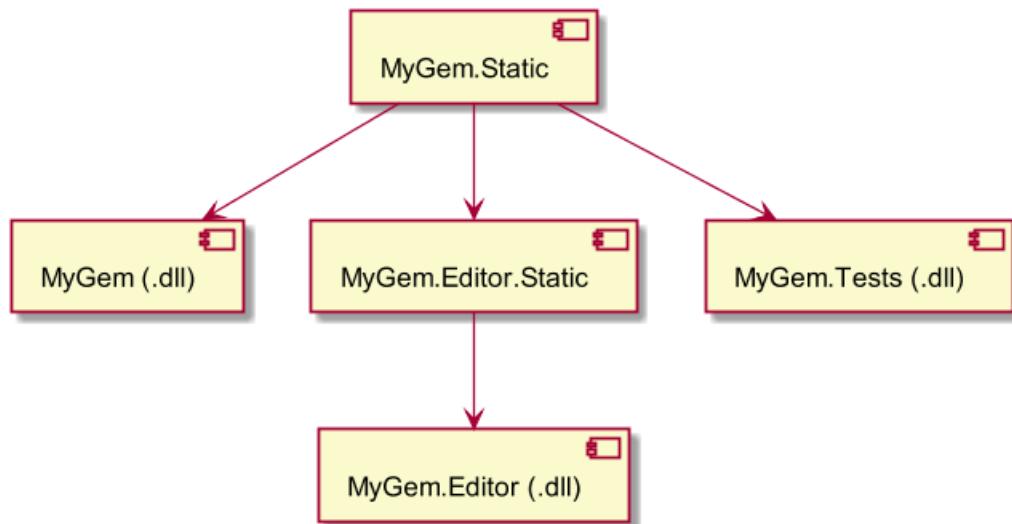
Let us go over the file structure of the gem we had created at `C:\git\book\Gems\MyGem`.

- **Assets** folder
  - A gem may provide assets for your project. If you had any, you would place them here. That could be scripts, prefabs and other types of assets. Since this is a new gem, this folder is empty.
- **Code** folder

- This is where C++ source code for your gem will reside. We will go over the details in just a moment.
- **preview.png** icon file
  - This is a default icon for your gem. This icon is the icon that O3DE Project Manager shows next to your gem. You may change it and customize it to your taste.
- **CMakeLists.txt** is the entry build file for the gem. We do not need to worry about it. The important build file for the gem is under Code folder, at `C:\git\book\Gems\MyGem\Code\CMakeLists.txt`.
- **gem.json** configuration file. The only importance of this file for us the name of the gem under `gem_name` property.

## Gem's C++ Code

**Figure 12.1. Build structure of a gem**



At a high level, a gem is made up of two essential build targets:

1. `MyGem.Static` is the static library containing your components.
2. `MyGem` module which links `MyGem.Static` library to create `C:\git\book\build\bin\profile\MyGem.dll` on Windows. This binary is loaded by the Editor, the Asset Processor and game launchers if this gem is enabled by the project.

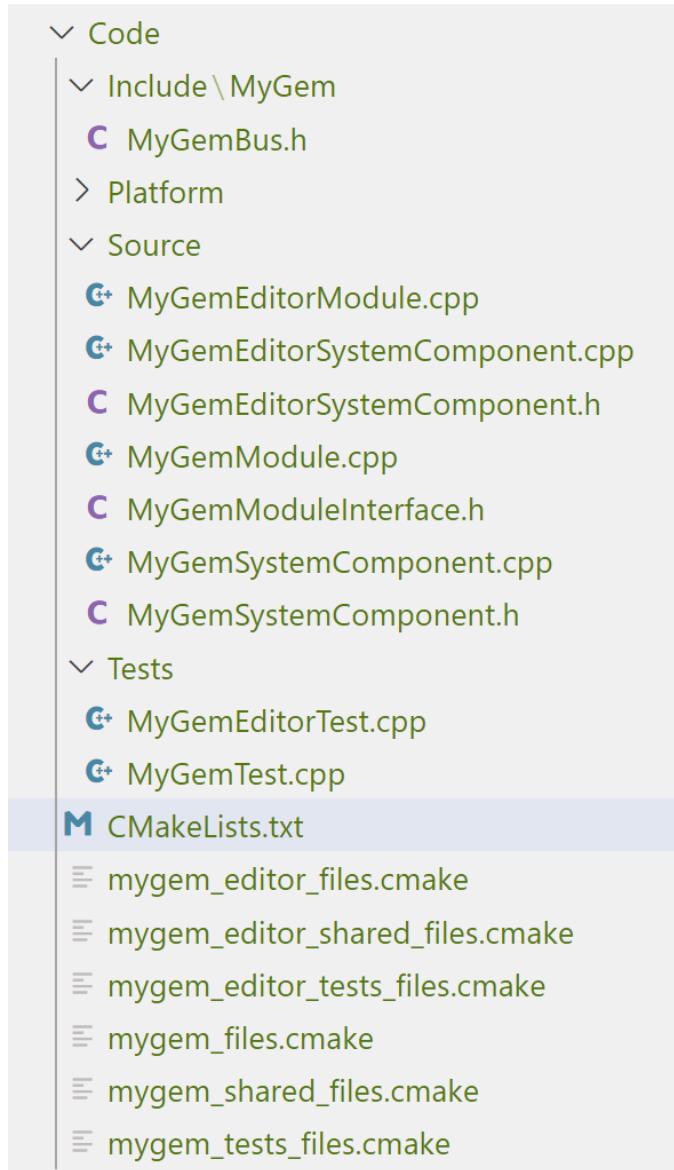
### ⚠ Important

This is the only required build target for a gem. You could actually merge the static library into the gem module but it is useful to put the component into a static library as it will be used in a few other build targets as well.

3. (Optional) `MyGem.Tests` module links against `MyGem.Static` and include any unit tests you might add.

4. (Optional) `MyGem.Editor.Static`, `MyGem.Editor.Tests` and `MyGem.Editor` are optional if you want to build Editor-specific design logic. This chapter will not cover them here as they are not necessary to get started with writing a game with O3DE.

**Figure 12.2. Gem Code Structure**



Let us dive deeper into the code structure of a gem that is created by default for us by `scripts/o3de.bat` at `C:\git\book\Gems\MyGem\Code`.

- `Include\MyGem` is the place for public C++ API of a gem, which should primarily be composed of header files, such as EBus headers and simple flat data structures.
- `Platform` folder contains various build and source files for specific platforms, such as Windows, Linux and so on.
- `Source` folder is the location for internal gem code, such as custom components.
- `Tests` folder contains basic unit tests for you to start with to test gem's code.

- `CMakeLists.txt` is the main build file for the gem. It defines all the build targets, such as gem's static library, gem's main module, unit tests build targets, etc.
- `mygem_editor_files.cmake`, `mygem_editor_shared_files.cmake`, `mygem_editor_tests_files.cmake` are list of files for Editor specific build targets, which I will not cover in this chapter.

### Note

Here is a quick summary on Editor specific targets. O3DE allows you to separate your Editor specific logic versus game runtime specific code. For example, you might want to create some design tools in the Editor that save their output to be used in the game. In such case, you might decide to separate that design code from your game code. Then the design components will go into Editor build targets, while final game components will go into regular components. This is optional, however. It is important to first understand how the regular non-Editor build targets work.

- `mygem_files.cmake` lists the files to go into the static library `MyGem.Static`.

```
set(FILES
    Include/MyGem/MyGemBus.h
    Source/MyGemModuleInterface.h
    Source/MyGemSystemComponent.cpp
    Source/MyGemSystemComponent.h
)
```

### Note

This lists both header and source files. In Visual Studio it will automatically put them into different filters.

- `mygem_shared_files.cmake` lists the files specific to building a shared library of `MyGem`.

```
set(FILES
    Source/MyGemModule.cpp
)
```

As expected, it only contains the module source file. All other component classes will come from the static library `MyGem.Static`.

- `mygem_tests_files.cmake` are the unit tests files.

```
set(FILES
    Tests/MyGemTest.cpp
)
```

### Tip

If you are only adding new components, you only need to worry about updating `mygem_files.cmake` to add your new component files. The content of other files will rarely change.

## Public Interface of a Gem

Project Manager script generated an empty EBus with an `AZ::Interface` for us at `C:\git\book\Gems\MyGem\Code\Include\MyGem\MyGemBus.h`. It is a starting point for a public interface for the gem.

## Note

AZ::Interface was covered in Chapter 6, *What is AZ::Interface?* EBus is a unique system within O3DE, for details see Chapter 8, *What is an AZ::EBus?*

You can have any number of EBuses in a gem. You may rename MyGemBus.h or delete it altogether.

### Example 12.2. MyGemBus.h

```
#pragma once

#include <AzCore/EBus/EBus.h>
#include <AzCore/Interface/Interface.h>

namespace MyGem
{
    class MyGemRequests
    {
        public:
            AZ_RTTI(MyGemRequests,
                    "{45ec313a-31a7-41ca-9e19-0f3f9351e328}");
            virtual ~MyGemRequests() = default;
            // Put your public methods here
    };

    class MyGemBusTraits
        : public AZ::EBusTraits
    {
        public:
            // EBusTraits overrides
            static constexpr AZ::EBusHandlerPolicy HandlerPolicy =
                AZ::EBusHandlerPolicy::Single;
            static constexpr AZ::EBusAddressPolicy AddressPolicy =
                AZ::EBusAddressPolicy::Single;
    };

    using MyGemRequestBus = AZ::EBus<MyGemRequests, MyGemBusTraits>;
    using MyGemInterface = AZ::Interface<MyGemRequests>;
}

// namespace MyGem
```

## Internal Gem Code

All internal implementation of the gem should go under MyGem\Code\Source. Here are the files that came from the gem template.

```
PS C:\git\book\Gems\MyGem\Code\Source> ls -Name

MyGemEditorModule.cpp
MyGemEditorSystemComponent.cpp
MyGemEditorSystemComponent.h
MyGemModule.cpp
MyGemModuleInterface.h
MyGemSystemComponent.cpp
```

---

`MyGemSystemComponent.h`

`MyGemEditorModule.cpp`, `MyGemEditorSystemComponent.cpp` and `MyGemEditorSystemComponent.h` are the basics to get started with building custom code for the Editor mode only, which you do not want to expose to the game runtime for either space or security concerns.

**MyGemModule.cpp** is the main class that declares the gem. At the moment it only registers a system component.

**MyGemModuleInterface.h** is the common base for both Editor and Game flavor of our gem. It lists the components that are currently registered with the engine.

### Example 12.3. MyGemModuleInterface

```
MyGemModuleInterface()
{
    m_descriptors.insert(m_descriptors.end(), {
        MyGemSystemComponent::CreateDescriptor(),
        ... // Add other components here
    });
}
```

**MyGemSystemComponent** is an empty system component that was generated for you. It is not necessary for a gem to have a system component. You may delete if you wish.

#### Note

A system component is a special component that is activated when the engine is initialized. It does not depend on any level and persists until the game launcher or the Editor quits. It is attached to the unique system entity. System components are great for handling system wide services that are not level dependent.

If you rename any of these files or add new files, be sure to update `mygem_files.cmake`.

#### Figure 12.3. mygem\_files.cmake

```
set(FILES
    Include/MyGem/MyGemBus.h
    Source/MyGemModuleInterface.h
    Source/MyGemSystemComponent.cpp
    Source/MyGemSystemComponent.h
)
```

## Summary

We have gone over the basics of a gem system and what we get when O3DE Project Manager script creates a new gem for us. In later chapters we will take a look at some examples of gems. That will allow us to understand how gems work and how they can be used to help you structure your game code.

#### Note

You can find the code for this chapter on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch12\\_new\\_gem](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch12_new_gem)

---

# Chapter 13. Gem: Set Window Position

## Note

You can find the code for this chapter on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch13\\_gem\\_window\\_position](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch13_gem_window_position)

## Introduction and Motivation

In this chapter we are going to implement our first gem. It will not introduce any new components but it will register a new console command that will allow the user to set the window position of the game application or the Editor. This is often useful in multiplayer testing if you need to launch a server and a few clients without them being overlapping each other. By default, O3DE engine would put both client windows in the same position. At some point, you may get tired of moving those windows manually and begin to wonder if you can automate that process somehow. Yes, you can. This chapter will show you how.

With this gem, you will be able to set the window position by passing the commands to the game launcher starting parameters, such as:

```
MyProject.GameLauncher.exe +SetWindowXY 100 200
```

**SetWindowXY** will expect two parameters: x and y distance from the top left corner of the screen to place the top left corner of the O3DE application.

## Create WindowPosition Gem

As shown in the previous chapter, we start by creating a new gem that will house the console command.

```
C:\O3DE\21.11.2\scripts\o3de.bat create-gem  
-gp C:\git\book\Gems\WindowPosition
```

Then register its location with the `project.json`.

```
"external_subdirectories": [  
    "C:/git/book/Gems/MyGem",  
    "C:/git/book/Gems/WindowPosition"  
]
```

And enable it for the project by its name.

```
set(ENABLED_GEMS  
...  
    MyGem  
    WindowPosition # new  
)
```

## Console Subsystem

O3DE allows you to add console commands using `AZ::Console` macros from `AzCore/Console/IConsole.h`.

**Figure 13.1. IConsole snippet**

```
///! @class IConsole
///! A simple console class for providing text
///! based variable and process interaction.
class IConsole
```

There is a number of useful macros that register new console commands and variables.

- AZ\_CONSOLEFREEFUNC is the one I will use in this chapter to register a free function as a callback when an AZ::Console command is invoked.
- There is also AZ\_CONSOLEFUNC that implements the callback as a class member function.

**Example 13.1. AZ\_CONSOLEFUNC usage**

```
class ConsoleTests
{
...
    void TestClassFunc(
        const AZ::ConsoleCommandContainer& someStrings);

AZ_CONSOLEFUNC(ConsoleTests, TestClassFunc,
    AZ::ConsoleFunctorFlags::Null, "");
```

- And AZ\_CVAR to create a console variable.

```
AZ_CVAR(int32_t, cloth_DebugDraw, 0, nullptr,
    AZ::ConsoleFunctorFlags::Null,
    "Draw cloth wireframe mesh:\n"
    " 0 - Disabled\n"
    " 1 - Cloth wireframe and particle weights");
```

Which can be used in the same source file by its name cloth\_DebugDraw.

```
if (cloth_DebugDraw == 1) { /*...*/ }
```

In source files outside of AZ\_CVAR definition, you have to first refer to it by AZ\_CVAR\_EXTERRED.

```
AZ_CVAR_EXTERRED(int32_t, cloth_DebugDraw);
```

**Example 13.2. Use of AZ\_CONSOLEFREEFUNC**

```
AZ_CONSOLEFREEFUNC(SetWindowXY, AZ::ConsoleFunctorFlags::Null,
    "Moves the window to x,y coordinates");
```

SetWindowXY is both the callback method for the console command and is the name to call in the when passing the command to the game launcher or invoking from an in-game console.

# Console Command Callback

All console callback methods are of the form:

```
void SetWindowXY(const AZ::ConsoleCommandContainer& args);
```

`AZ::ConsoleCommandContainer` is a fixed vector of strings. You can extract the parameters one at a time by converting them to the type you expect them to be.

Here is an example of a `WindowXY` callback for the console command `windowxy`:

```
void SetWindowXY(const AZ::ConsoleCommandContainer& args)
{
    if (args.size() != 2)
    {
        return;
    }

    // Grab first argument.
    AZ::CVarFixedString argX{ args.front() };
    const int x = atoi(argX.c_str());

    // Grab second argument.
    AZ::CVarFixedString argY{ *(args.begin() + 1) };
    const int y = atoi(argY.c_str());
    // ...
}
```

The rest of the implementation is Windows specific. It acquires a window handle and sets the position. It is outside of scope of this book to discuss Windows specific API, so we are going to just breeze through it by showing the final result.

## Note

If you are curious where one can read up the documentation of `GetActiveWindow`, `GetWindowInfo` and `SetWindowPos`, you should refer to public MSDN at:

<https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowpos>

### Example 13.3. `WindowX` full method

```
void SetWindowXY(const AZ::ConsoleCommandContainer& args)
{
    if (args.size() != 2)
    {
        return;
    }

    // Grab first argument.
    AZ::CVarFixedString argX{ args.front() };
    const int x = atoi(argX.c_str());

    // Grab second argument.
    AZ::CVarFixedString argY{ *(args.begin() + 1) };
    const int y = atoi(argY.c_str());

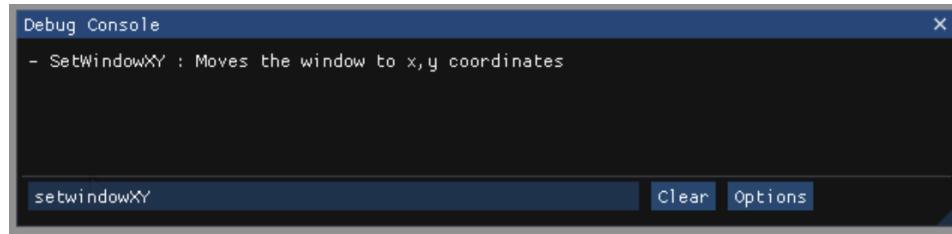
    HWND handle = GetActiveWindow();
    if (!handle)
    {
        // Try the console window.
        handle = GetConsoleWindow();
    }
}
```

```
if (handle)
{
    SetWindowPos(handle, nullptr,
                 x, y,
                 0, 0, SWP_NOOWNERZORDER | SWP_NOSIZE);
}
```

## Conclusion and Source Code

Once you have the new gem compiled and enabled for your project, you can bring up ImGui debug console in a game launcher with *tilde* key. and type a partial spelling of the command and hit TAB. That will trigger suggestions for auto-completion.

**Figure 13.2. Example of a console command: SetWindowXY**



### Note

You can find the code for this chapter on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch13\\_gem\\_window\\_position](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch13_gem_window_position)

## WindowConsoleCommand.cpp

**Example 13.4. WindowConsoleCommand.cpp**

```
#include <AzCore/PlatformIncl.h>
#include <AzCore/Console/IConsole.h>

namespace WindowPosition
{
    void SetWindowXY(const AZ::ConsoleCommandContainer& args)
    {
        if (args.size() != 2)
        {
            return;
        }

        // Grab first argument.
        AZ::CVarFixedString argX{ args.front() };
        const int x = atoi(argX.c_str());

        // Grab second argument.
    }
}
```

```
AZ::CVarFixedString argY{ *(args.begin() + 1) };
const int y = atoi(argY.c_str());

HWND handle = GetActiveWindow();
if (!handle)
{
    // Try the console window.
    handle = GetConsoleWindow();
}

if (handle)
{
    SetWindowPos(handle, nullptr,
        x, y,
        0, 0, SWP_NOOWNERZORDER | SWP_NOSIZE);
}
}

AZ_CONSOLEFREEFUNC(SetWindowXY, AZ::ConsoleFunctorFlags::Null,
    "Moves the window to x,y coordinates");
} // namespace WindowPosition
```

Notice that this time this file is not part of any component. It is a stand-alone source file but it still needs to be added to `windowposition_files.cmake`.

### Example 13.5. `windowposition_files.cmake`

```
set(FILES
    Include/WindowPosition/WindowPositionBus.h
    Source/WindowPositionModuleInterface.h
    Source/WindowPositionSystemComponent.cpp
    Source/WindowPositionSystemComponent.h

    Source/WindowConsoleCommand.cpp      # new
)
```

# Chapter 14. Enabling NvCloth Gem

## Introduction

O3DE engine has a lot of gems that are ready to be used. This chapter will go over the steps involved in adding one of core gems to our project.

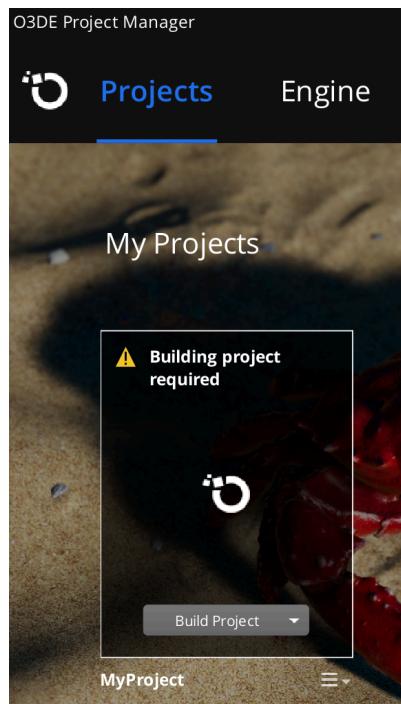
### Note

The changes in this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch14\\_enable\\_nvcloth](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch14_enable_nvcloth)

## Adding NvCloth Gem

In order to investigate available gems, we can use O3DE Project Manager by launching it from your O3DE installation at C:\O3DE\21.11.2\bin\Windows\profile\Default\o3de.exe. You should see the project on its front page.



'MyProject' in O3DE Project Manager

### Tip

If a project does not appear there, then needs to be added to C:\Users\<user>\.o3de\o3de\_manifest.json. This can be done either using the script:

```
C:\O3DE\21.11.2\scripts\o3de.bat register
```

```
--project-path C:\git\book\MyProject
```

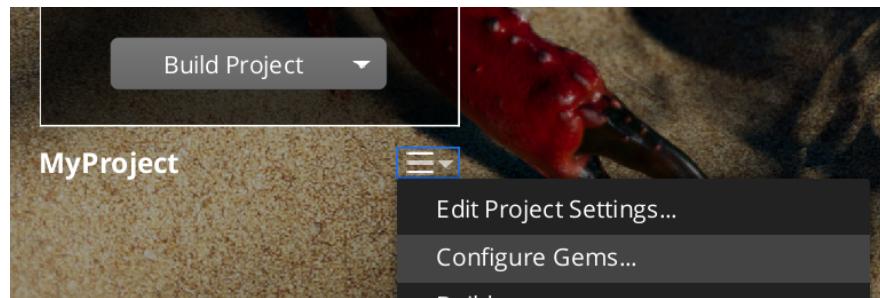
Or modifying `o3de_manifest.json`'s **projects** property directly and adding the full path to the project.

### Example 14.1. Snippet from `o3de_manifest.json`

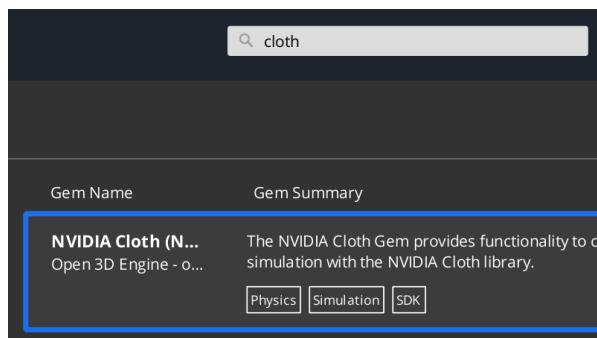
```
{  
    "o3de_manifest_name": "<user>" ,  
    "origin": "C:/Users/<user>/ .o3de" ,  
    ...  
    "projects": [  
        "C:/git/book/MyProject"  
    ],
```

You can add, remove and explore gems by choosing **Configure Gem** option from the drop down list next the project.

**Figure 14.1. Configure Gems**



Our main interest is **NvCloth** gem.



NVIDIA Cloth gem

You can enable the gem here and save the changes. Verify that the gem was added in `C:\git\book\MyProject\Code\enabled_gems.cmake`.

### Example 14.2. `enabled_gems.cmake` with NvCloth gem

```
set(ENABLED_GEMS  
...)
```

```
MyGem
WindowPosition
NvCloth    # new
)
```

## Using NvCloth Gem

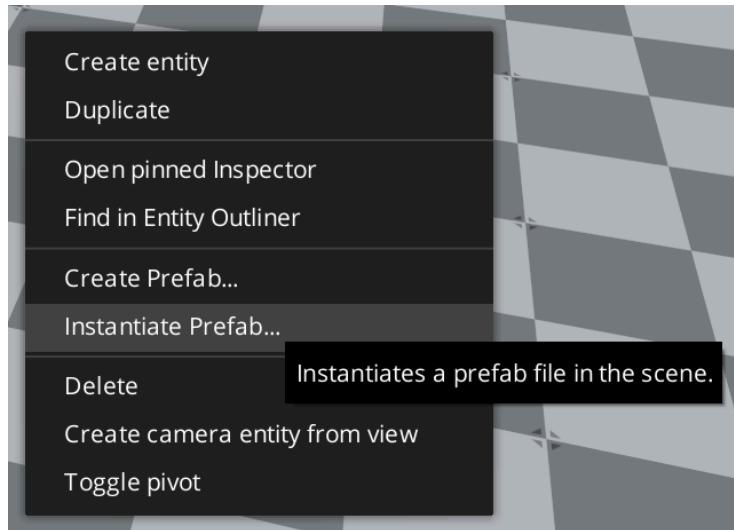
Once you re-build the project, the gem and its assets will become available to the project. We are going to use one of its assets: a chicken.

**Figure 14.2. Chicken from NvCloth gem**

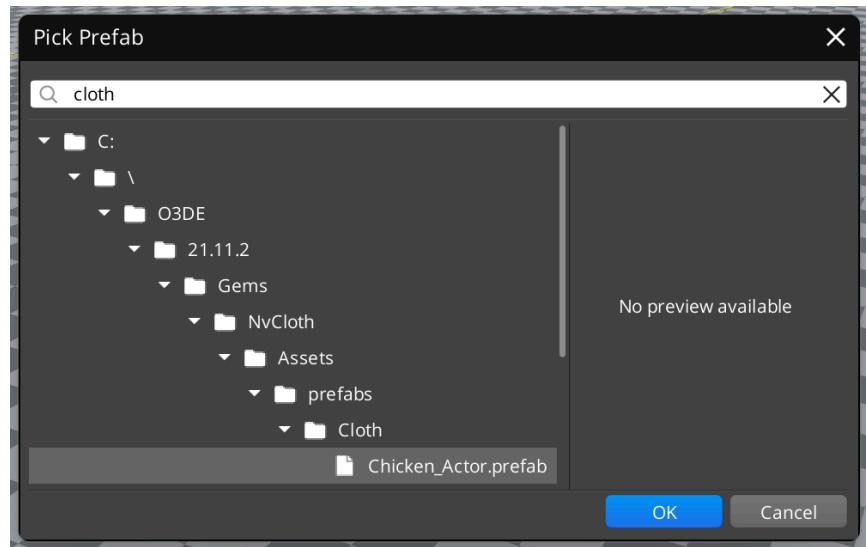


Spawn the chicken by right clicking in the viewport and choosing **Instantiate Prefab**.

**Figure 14.3. Instantiate Prefab**



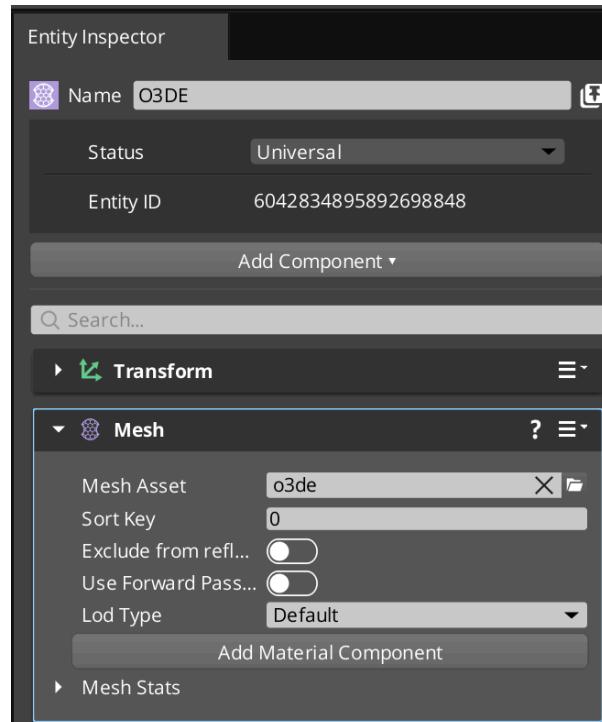
Then navigate to the NvCloth location at C:\O3DE\21.11.2\Gems\NvCloth\Assets\prefabs\Cloth\Chicken\_Actor.prefab or search for **chicken**.

**Figure 14.4. Pick Chicken\_Actor.prefab**

## Adding Your Assets to Gems

Aside from adding assets directly to the project, for example under `C:\git\book\MyProject\Assets`. You can also add assets under gems. For this chapter, I have created a 3D text model in Blender, exported it to fbx format and then placed it under `C:\git\book\Gems\MyGem\Assets\o3de.fbx`. You can then place it into the level using Mesh component.

### Example 14.3. O3DE.fbx in the level



# Summary

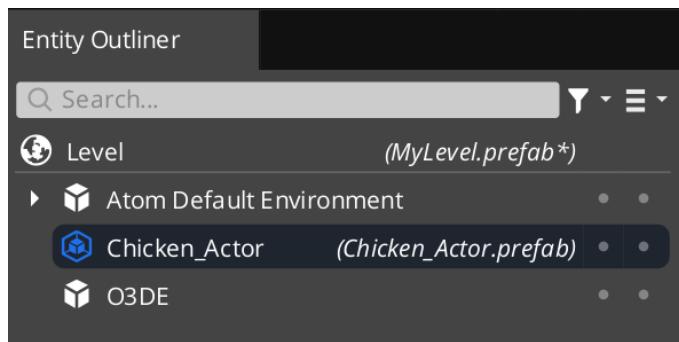
## Note

The changes in this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch14\\_enable\\_nvcloth](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch14_enable_nvcloth)

In this chapter, we have enabled NvCloth gem and added a new FBX asset under MyGem. Here is the look of the level with these changes.

**Figure 14.5. Entities in the Level**



**Figure 14.6. Chicken in O3DE**



---

## **Part VI. Unit Tests in O3DE**

---

## Table of Contents

15. Writing Unit Tests for Components .....	122
Introduction .....	122
Unit Test Build Target .....	122
Unit Tests .....	124
16. Unit Tests with Mock Components .....	131
Testing with Mocks .....	131
Mocking TransformComponent .....	131
Summary .....	137

---

# Chapter 15. Writing Unit Tests for Components

The system should first be made to run, even though it does nothing useful except call the proper set of dummy subprograms. Then, bit-by-bit it is fleshed out, with the subprograms in turn being developed into actions or calls to empty stubs in the level below.

—Frederick P. Brooks Jr. "No Silver Bullet - Essence and Accident in Software Engineering", 1986

## Introduction

### Note

The source code for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch15\\_unittests](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch15_unittests)

An often overlooked development process in game development is unit testing. O3DE engine comes with a powerful Google Test and Google Mock frameworks included.

### Tip

The presentation of Google Test and Google Mock framework is outside of the scope of this book, but you can find great tutorials for them online:

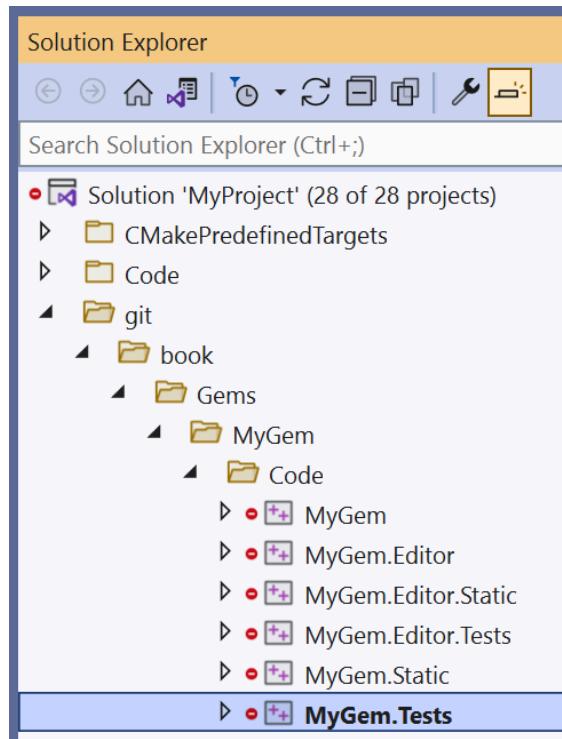
- <https://github.com/google/googletest>
- <https://github.com/google/googletest/blob/main/docs/primer.md>
- [https://github.com/google/googletest/blob/main/docs/gmock\\_for\\_dummies.md](https://github.com/google/googletest/blob/main/docs/gmock_for_dummies.md)
- [https://github.com/google/googletest/blob/main/docs/gmock\\_cook\\_book.md](https://github.com/google/googletest/blob/main/docs/gmock_cook_book.md)

This chapter will show you how to write a simple unit test for `OscillatorComponent` that we implemented in Chapter 9, *Using AZ::TickBus*.

## Unit Test Build Target

Whenever you create a new Gem in O3DE, you get a default unit test build target, such as **MyGem.Tests**. In a Visual Studio solution, you can find it alongside with other build targets.

However, when we generated **MyProject** it did not receive a unit test build target. We need *MyProject.Tests* build target in order to test `OscillatorComponent`, since the component is inside *MyProject* and not inside a gem. While it is not at all necessary to place components inside a project, it will serve as a good example of what it takes to add unit tests from scratch.

**Figure 15.1. Location of unit tests in a gem**

Since we are adding a new build target to the project, we have to modify C:\git\book\MyProject\Code\CMakeLists.txt. Here is the change to add to the end of CMakeLists.txt.

### Example 15.1. Add MyProject.Tests build target

```
# See if globally, tests are supported
if(PAL_TRAIT_BUILD_TESTS_SUPPORTED)
    ly_add_target(
        NAME MyProject.Tests ${PAL_TRAIT_TEST_TARGET_TYPE}
        NAMESPACE Gem
        FILES_CMAKE
            myproject_test_files.cmake
        INCLUDE_DIRECTORIES
            PRIVATE
                Tests
                Source
        BUILD_DEPENDENCIES
            PRIVATE
                AZ::AzTest
                AZ::AzFramework
                Gem::MyProject.Static
    )

    # Add MyProject.Tests to googletest
    ly_add_googletest(
        NAME Gem::MyProject.Tests
    )
endif()
```

The important elements here are:

- Mark **MyProject.Tests** target as a unit test build target.

```
ly_add_gtest(
    NAME Gem::MyProject.Tests
)
```

This way you can launch this target as a Google Unittest from Visual Studio.

- Target **MyProject.Tests** links against **MyProject.Static**, which has the object code with `OscillatorComponent`.

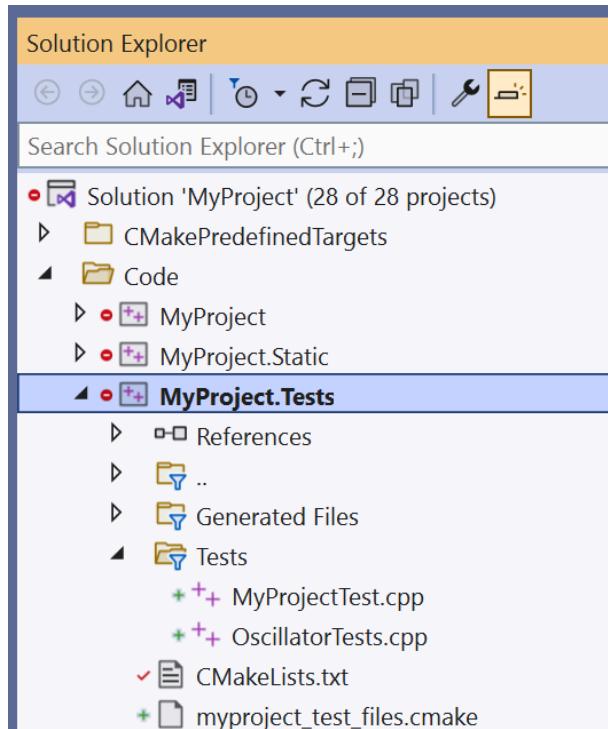
```
BUILD_DEPENDENCIES
PRIVATE
    Gem::MyProject.Static
```

- `myproject_test_files.cmake` contains the files to be built for unit tests.

### Example 15.2. `myproject_test_files.cmake`

```
set(FILES
    Tests/MyProjectTest.cpp
    Tests/OscillatorTests.cpp
)
```

**Figure 15.2. Location of unit tests in a gem**



## Unit Tests

`MyProjectTest.cpp` will serve as the unit test setup.

### Example 15.3. MyProjectTest.cpp

```
#include <AzTest/AzTest.h>
AZ_UNIT_TEST_HOOK(DEFAULT_UNIT_TEST_ENV);
```

## Running Unit Tests

Before we go into modifying the unit tests, let us go over how to run these tests. In Visual Studio you can set **MyProject.Tests** as the startup project by right clicking on the project and selecting **Set as StartUp Project**. Running the project will execute the unit tests.

### Example 15.4. Unit test output

```
[=====] Running 1 test from 1 test case.
[-----] Global test environment set-up.
[-----] 1 test from OscillatorTest
[ RUN    ] OscillatorTest.EntityMovingUp
[      OK ] OscillatorTest.EntityMovingUp (7 ms)
[-----] 1 test from OscillatorTest (8 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test case ran. (9 ms total)
[ PASSED ] 1 test.
```

### Tip

If you take a peak at the property pages for **MyProject.Tests** you can find the steps needed to run the unit tests from the command line:

```
C:/O3DE/21.11.2/bin/Windows/profile/Default/AzTestRunner.exe
C:/git/book/build/bin/profile/MyProject.Tests.dll AzRunUnitTests
```

## Structure of a Unit Test

### Note

The code for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch15\\_unittests](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch15_unittests)

The majority of functionality of a game in O3DE is done in components. So it only makes to write unit tests for components. A common approach is as follows:

1. Create an AZ::Entity.
2. Add the component you are testing.
3. Add another component that is involved in the interaction you are testing.
4. Invoke APIs (such as EBuses) that trigger or advance the interaction.
5. Test the results.

For example, here is the structure of a unit test we will write for OscillatorComponent (from Chapter 9, *Using AZ::TickBus*):

1. Create an `AZ::Entity` as mentioned. Nothing special is required for that. You may create one on the stack or on a heap.
2. Add `OscillatorComponent` to the entity.
3. Add `AzFramework::TransformComponent`.
4. Invoke `TickBus` since `OscillatorComponent` moves the entity in `OnTick` method.
5. Test that `TransformComponent` was moved up after one tick.

I will go through each part of the unit test source code file and present the entire code at the end.

## Memory Allocator Setup

The challenging part of O3DE unit test is getting the memory allocators right. You can avoid the headache by using a pre-made unit test base class, `::UnitTest::AllocatorsFixture` from `AzCore/UnitTest/TestTypes.h`. It has `SetUp` and `TearDown` virtual methods that it overrides from GoogleTest base class, `::testing::Test`.

```
#include <AzCore/UnitTest/TestTypes.h>
...
class OscillatorTest
    : public ::UnitTest::AllocatorsFixture
{
...
    void SetUp() override
    {
        ::UnitTest::AllocatorsFixture::SetUp();
    }
...
```

With that, O3DE memory allocators will be correctly set up and you will not need to deal with errors, such as:

```
'SystemAllocator' NOT ready for use! Call Create first!
```

If you ever attempted to write your own O3DE unit test, you have likely seen such runtime crashes. `AllocatorsFixture` solves that problem.

## Registering Components

A unit test runs in a slim environment. There are no gems loaded, even the project's components are not automatically registered.

### Tip

If you attempt to create a component in a unit test without first registering it with O3DE, you would hit a runtime assert, such as:

```
o3de\Code\Framework\AzCore\AzCore/UnitTest/UnitTest.h(155):
error: Entity '6688814788' [6688814788] cannot be activated.
No descriptor registered for Component class 'OscillatorComponent'.
```

That is solved by creating an instance of `AZ::SerializeContext` and reflecting the components you wish to use.

```
AZStd::unique_ptr<AZ::SerializeContext> m_sc;
AZStd::unique_ptr<AZ::ComponentDescriptor> m_td;
AZStd::unique_ptr<AZ::ComponentDescriptor> m_od;

...
    // register components involved in testing
    m_sc = AZStd::make_unique<AZ::SerializeContext>();
    m_td.reset(TransformComponent::CreateDescriptor());
    m_td->Reflect(m_sc.get());
    m_od.reset(OscillatorComponent::CreateDescriptor());
    m_od->Reflect(m_sc.get());
```

## Adding Components to an Entity

Once you have created an entity, which is as easy as:

```
Entity e;
```

You can add the components we have registered earlier:

```
...
    // helper method
void PopulateEntity(Entity& e)
{
    // OscillatorComponent is the component we are testing
    e.CreateComponent<OscillatorComponent>();
    // And how it interacts with
    e.CreateComponent<AzFramework::TransformComponent>();
...
}
```

## Activate the Entity

Recall that components are not active until their `Activate` methods are called. You can do that by initializing and activating the entity.

```
// Bring the entity online
e.Init();
e.Activate();
```

## Test Body

As described earlier, this test will trigger one tick event and test the results on `TransformComponent`.

```
TEST_F(OscillatorTest, EntityMovingUp)
{
    Entity e;
    PopulateEntity(e);

    // Move entity to (0,0,0)
    TransformBus::Event(e.GetId(),
        &TransformBus::Events::SetWorldTranslation,
        Vector3::CreateZero());

    // tick once
```

```
    TickBus::Broadcast(&TickBus::Events::OnTick, 0.1f,
                        ScriptTimePoint()));

    // Get entity's position
    Vector3 change;
    TransformBus::EventResult(change, e.GetId(),
                              &TransformBus::Events::GetWorldTranslation);

    // check that it moved up, by any amount
    ASSERT_TRUE(change.GetZ() > 0);
}
```

### >Note

OscillatorTest is the unit test fixture that this test body inherits from. That is why we can use PopulateEntity directly. EntityMovingUp is the name of the test.

## Unit Test Hook

O3DE has a macro that wraps GoogleTest. You only need to call it once from one of your test files.

```
#include <AzTest/AzTest.h>
AZ_UNIT_TEST_HOOK(DEFAULT_UNIT_TEST_ENV);
```

## Output

If all went well, you should see the following output in the command line upon running the unit tests.

```
[-----] 1 test from OscillatorTest
[ RUN      ] OscillatorTest.EntityMovingUp
[       OK ] OscillatorTest.EntityMovingUp (1 ms)
[-----] 1 test from OscillatorTest (2 ms total)
```

## Source Code

### Note

The source code for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch15\\_unittests](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch15_unittests)

#### Example 15.5. MyProjectTest.cpp

```
#include <AzTest/AzTest.h>
AZ_UNIT_TEST_HOOK(DEFAULT_UNIT_TEST_ENV);
```

#### Example 15.6. OscillatorTest.cpp

```
#include <OscillatorComponent.h>
#include <AzCore/Component/Entity.h>
#include <AzCore/Component/TransformBus.h>
#include <AzCore/Serialization/SerializeContext.h>
#include <AzCore/std/smart_ptr/unique_ptr.h>
```

```
#include <AzCore/UnitTest/TestTypes.h>
#include <AzFramework/Components/TransformComponent.h>
#include <AzTest/AzTest.h>

namespace MyProject
{
    class OscillatorTest
        : public ::UnitTest::AllocatorsFixture
    {
        AZStd::unique_ptr<AZ::SerializeContext> m_sc;
        AZStd::unique_ptr<AZ::ComponentDescriptor> m_td;
        AZStd::unique_ptr<AZ::ComponentDescriptor> m_od;
    protected:
        void SetUp() override
        {
            ::UnitTest::AllocatorsFixture::SetUp();

            // register components involved in testing
            m_sc = AZStd::make_unique<AZ::SerializeContext>();
            m_td.reset(AzFramework::TransformComponent
                ::CreateDescriptor());
            m_td->Reflect(m_sc.get());
            m_od.reset(OscillatorComponent::CreateDescriptor());
            m_od->Reflect(m_sc.get());
        }

        void TearDown() override
        {
            m_td.reset();
            m_od.reset();
            m_sc.reset();

            ::UnitTest::AllocatorsFixture::TearDown();
        }

        // helper method
        void PopulateEntity(AZ::Entity& e)
        {
            // OscillatorComponent is the component we are testing
            e.CreateComponent<OscillatorComponent>();
            // And how it interacts with
            e.CreateComponent<AzFramework::TransformComponent>();

            // Bring the entity online
            e.Init();
            e.Activate();
        }
    };

TEST_F(OscillatorTest, EntityMovingUp)
{
    using namespace AZ;
    Entity e;
    PopulateEntity(e);
```

```
// Move entity to (0,0,0)
TransformBus::Event(e.GetId(),
    &TransformBus::Events::SetWorldTranslation,
    Vector3::CreateZero());

// tick once
TickBus::Broadcast(&TickBus::Events::OnTick, 0.1f,
    ScriptTimePoint());

// Get entity's position
Vector3 change = AZ::Vector3::CreateZero();
TransformBus::EventResult(change, e.GetId(),
    &TransformBus::Events::GetWorldTranslation);

// check that it moved up, by any amount
ASSERT_TRUE(change.GetZ() > 0);
}
```

---

# Chapter 16. Unit Tests with Mock Components

## Note

The source code for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch16\\_unittests\\_mock](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch16_unittests_mock)

## Testing with Mocks

Chapter 15, *Writing Unit Tests for Components*, wrote a unit test that involved two real components. Another common pattern is to write a test that involves one real component and one mock component that mimics another. In this case, the real component will be `OscillatorComponent` and the mock will be `MockTransformComponent`.

The purpose of such a test is to check that `OscillatorComponent` invokes a particular EBus call. This is different from checking if some variable has the expected value. Instead, this type of unit test checks if a particular interface was invoked. So let us write a unit test that will check that `OscillatorComponent` calls `SetWorldTranslation` on a `Transform`-like component.

A "mock" component is a fake test component. It is created to test the interaction between the mock and some real component. It is a powerful testing tool. O3DE ships with Google Mock which provides a powerful C++ mock framework.

## Note

You can familiarize yourself with Google Mock on its extensive documentation online:

- <https://github.com/google/googletest/blob/master/gmock/README.md>
- [https://github.com/google/googletest/blob/main/docs/gmock\\_for\\_dummies.md](https://github.com/google/googletest/blob/main/docs/gmock_for_dummies.md)
- [https://github.com/google/googletest/blob/main/docs/gmock\\_cook\\_book.md](https://github.com/google/googletest/blob/main/docs/gmock_cook_book.md)

## Mocking TransformComponent

Conceptually, a component in O3DE is identified by EBus handlers it implements. You could choose to have a mock object inherit all of such handlers that `TransformComponent` implements. However, in the case of `OscillatorComponent` interacting with `TransformComponent`, we know that only `TransformBus` is involved, thus it is sufficient to only handle `TransformBus::Handler`.

```
#include <AzCore/Component/TransformBus.h>
...
/**
 * \brief Pretend to be a TransformComponent
 */
class MockTransformComponent
: public AZ::Component
, public AZ::TransformBus::Handler
```

{

Of course, it still has to be a O3DE component, so it has to inherit from AZ::Component and implement the most basic component interface.

```
...
    // be sure this guid is unique, avoid copy-paste errors!
    AZ_COMPONENT(MockTransformComponent,
        "{7E8087BD-46DA-4708-ADB0-08D7812CA49F}");

    // Just a mock object, no reflection necessary
    static void Reflect(ReflectContext*) {}

    // Mocking out pure virtual methods
    void Activate() override
    {
        AZ::TransformBus::Handler::BusConnect(GetEntityId());
    }
    void Deactivate() override
    {
        AZ::TransformBus::Handler::BusDisconnect();
    }
...
```

Since MockTransformComponent needs to handle TransformBus interface it has to connect to the bus.

## Mocking Virtual Methods

Now we get to the first Google Mock feature: mocking virtual methods.

```
class MockTransformComponent
...
    // OscillatorComponent will be calling these methods
    MOCK_METHOD1(SetWorldTranslation, void (const AZ::Vector3&));
    MOCK_METHOD0(GetWorldTranslation, AZ::Vector3 ());
...
```

These override virtual methods of TransformBus:

```
class TransformInterface {
...
    virtual void SetWorldTranslation(const AZ::Vector3&);
    virtual AZ::Vector3 GetWorldTranslation()
```

MOCK\_METHOD0, MOCK\_METHOD1 and so on provide stub implementations of these methods with many additional features. In this chapter, SetWorldTranslation and GetWorldTranslation will be used. A few other pure virtual methods from TransformBus will not be. Those still need to be mocked out, so that we can instantiate the object:

```
// Unused methods but they are pure virtual in TransformBus
MOCK_METHOD0(IsStaticTransform, bool ());
MOCK_METHOD0(IsPositionInterpolated, bool ());
MOCK_METHOD0(IsRotationInterpolated, bool ());
MOCK_METHOD0(GetLocalTM, const Transform&());
```

```
MOCK_METHOD0(GetWorldTM, const Transform& ());
MOCK_METHOD1(BindTransformChangedEventHandler,
             void(TransformChangedEvent::Handler&));
MOCK_METHOD1(BindParentChangedEventHandler,
             void(ParentChangedEvent::Handler&));
MOCK_METHOD1(BindChildChangedEventHandler,
             void(ChildChangedEvent::Handler&));
MOCK_METHOD2(NotifyChildChangedEvent,
             void(ChildChangeType, EntityId));
```

So the entire mock definition is as follows:

### Example 16.1. MockTransformComponent definition

```
/**
 * \brief Pretend to be a TransformComponent
 */
class MockTransformComponent
    : public AZ::Component
    , public AZ::TransformBus::Handler
{
public:
    // be sure this guid is unique, avoid copy-paste errors!
    AZ_COMPONENT(MockTransformComponent,
                 "{7E8087BD-46DA-4708-ABD0-08D7812CA49F}");

    // Just a mock object, no reflection necessary
    static void Reflect(ReflectContext*) {}

    // Mocking out pure virtual methods
    void Activate() override
    {
        AZ::TransformBus::Handler::BusConnect(GetEntityId());
    }
    void Deactivate() override
    {
        AZ::TransformBus::Handler::BusDisconnect();
    }

    // OscillatorComponent will be calling these methods
    MOCK_METHOD1(SetWorldTranslation, void (const AZ::Vector3&));
    MOCK_METHOD0(GetWorldTranslation, AZ::Vector3 ());

    // Unused methods but they are pure virtual in TransformBus
    MOCK_METHOD0(IsStaticTransform, bool ());
    MOCK_METHOD0(IsPositionInterpolated, bool ());
    MOCK_METHOD0(IsRotationInterpolated, bool ());
    MOCK_METHOD0(GetLocalTM, const Transform& ());
    MOCK_METHOD0(GetWorldTM, const Transform& ());
    MOCK_METHOD1(BindTransformChangedEventHandler,
                void(TransformChangedEvent::Handler&));
    MOCK_METHOD1(BindParentChangedEventHandler,
                void(ParentChangedEvent::Handler&));
    MOCK_METHOD1(BindChildChangedEventHandler,
```

```
        void(ChildChangedEvent::Handler&));
MOCK_METHOD2(NotifyChildChangedEvent,
    void(ChildChangeType, EntityId));
};
```

## Setting Up Entity with Mock Components

Much like in the previous chapter, we will create a re-usable test fixture to handle memory allocators and registration of the components.

```
class OscillatorMockTest
    : public ::UnitTest::AllocatorsFixture
{
    AZStd::unique_ptr<AZ::SerializeContext> m_sc;
    AZStd::unique_ptr<AZ::ComponentDescriptor> m_md;
    AZStd::unique_ptr<AZ::ComponentDescriptor> m_od;

protected:
    void SetUp() override
    {
        ::UnitTest::AllocatorsFixture::SetUp();

        // register components involved in testing
        m_sc = AZStd::make_unique<AZ::SerializeContext>();
        m_md.reset(MockTransformComponent::CreateDescriptor());
        m_md->Reflect(m_sc.get());
        m_od.reset(OscillatorComponent::CreateDescriptor());
        m_od->Reflect(m_sc.get());
    }
    ...
    ...
    // helper method
    void PopulateEntity(Entity& e)
    {
        // OscillatorComponent is the component we are testing
        e.CreateComponent<OscillatorComponent>();
        // We can mock out Transform and test the interaction
        mock = new MockTransformComponent();
        e.AddComponent(mock);

        // Bring the entity online
        e.Init();
        e.Activate();
    }

    MockTransformComponent* mock = nullptr;
};
```

Now, we have a helper method to create and activate the entity while keeping a pointer to mock.

### >Note

`e.AddComponent(mock)` passes the memory ownership of the component instance to the entity, so do not try to delete mock pointer in the test body. Entity destructor will take care of that.

## Setting Expectations on Interface Calls

Now we are ready to write the test body. It starts with preparing the entity.

```
TEST_F(OscillatorMockTest, Calls_SetWorldTranslation)
{
    Entity e;
    PopulateEntity(e);
```

The one feature we are going to use in this chapter is setting expectation that a particular method will be called. For example, we will test that `SetWorldTranslation` is called once.

```
// expect SetWorldTranslation() to be called
EXPECT_CALL(*mock, SetWorldTranslation(_)).Times(1);
```

### >Note

The underscore in `SetWorldTranslation(_)` is a special Google Mock object `::testing:::_` that matches any argument. So this statement is: "expect `SetWorldTranslation` to be called once with any parameter."

Recall that `OscillatorComponent` calls `GetWorldTranslation` to get the current position. We mocked it out in `MockTransformComponent` as:

```
MOCK_METHOD0(GetWorldTranslation, AZ::Vector3());
```

So it will not return any value by default. If your test were to run and call this method without any extra preparation, it would error out at runtime with an error:

```
[ RUN      ] OscillatorMockTest.Calls_SetWorldTranslation
unknown file: error: C++ exception with description
  "Uninteresting mock function call - returning default value.
Function call: GetWorldTranslation()
The mock function has no default action set, and its return
type has no default value set." thrown in the test body.
```

Google Mock provides a way to customize the behavior of `GetWorldTranslation` without having to modify `MockTransformComponent`. Here is how we can set it to return vector of (0,0,0):

```
// setup a return value for GetWorldTranslation()
ON_CALL(*mock, GetWorldTranslation()).WillByDefault(
    Return(AZ::Vector3::CreateZero()));
```

In plain English, this means "whenever `GetWorldTranslation` is called return `Vector3::CreateZero`".

With this, the test body is short and sweet:

```
TEST_F(OscillatorMockTest, Calls_SetWorldTranslation)
```

```
{  
    Entity e;  
    PopulateEntity(e);  
  
    // setup a return value for GetWorldTranslation()  
    ON_CALL(*mock, GetWorldTranslation()).WillByDefault(  
        Return(AZ::Vector3::CreateZero()));  
  
    // expect SetWorldTranslation() to be called  
    EXPECT_CALL(*mock, SetWorldTranslation(_)).Times(1);  
  
    TickBus::Broadcast(&TickBus::Events::OnTick, 0.1f,  
        ScriptTimePoint());  
}
```

It does the following actions:

1. Prepares the entity for the test.
2. Sets up default return value of GetWorldTranslation.
3. Sets up expectation what will be called by OscillatorComponent on TransformBus.
4. Ticks once.
5. At the end of the scope, Google Mock will do the work for us to ensure the expectation was met.

## Tip

If you were to run such a test you see a warning by Google Mock:

```
GMOCK WARNING:  
Uninteresting mock function call - taking default action specified  
MyProject\Gem\Code\Tests\MyProjectTest.cpp(105):  
    Function call: GetWorldTranslation()  
        Returns: 16-byte object <00-00 ... 00-00>  
NOTE: You can safely ignore the above warning unless this call  
should not happen. Do not suppress it by blindly adding  
an EXPECT_CALL() if you don't mean to enforce the call.
```

Google Mock is telling us in this warning that GetWorldTranslation was called as we would expect OscillatorComponent to do but that the test body did not specify any expectation regarding that. A good rule of thumb as the warning suggests is to only test for what really matters. In this case, SetWorldTranslation is the call that matters and the expectation for it was set.

Once you are sure that the test good, you can get rid of this warning by changing the type of mock object on creation from:

```
// We can mock out Transform and test the interaction  
mock = new MockTransformComponent();  
e.AddComponent(mock);
```

To:

```
// We can mock out Transform and test the interaction  
mock = new NiceMock<MockTransformComponent>();
```

```
e.AddComponent(mock);
```

NiceMock is a Google Mock class that silently ignores unexpected calls into a mock object.

# Summary

## Note

The source code for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch16\\_unittests\\_mock](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch16_unittests_mock)

If all goes well, you should see the following output for this chapter's unit test:

```
[-----] 1 test from OscillatorMockTest
[ RUN      ] OscillatorMockTest.Calls_SetWorldTranslation
[       OK ] OscillatorMockTest.Calls_SetWorldTranslation (2 ms)
[-----] 1 test from OscillatorMockTest (3 ms total)
```

Here is the entire source code for completeness.

### Example 16.2. `OscillatorMockTest.cpp`

```
#include <OscillatorComponent.h>
#include <AzCore/Component/ComponentApplication.h>
#include <AzCore/Component/Entity.h>
#include <AzCore/Component/TransformBus.h>
#include <AzCore/std/smart_ptr/unique_ptr.h>
#include <AzCore/UnitTest/TestTypes.h>
#include <AzTest/AzTest.h>

using namespace AZ;
using namespace AZStd;
using namespace MyProject;
using namespace ::testing;

/**
 * \brief Pretend to be a TransformComponent
 */
class MockTransformComponent
    : public AZ::Component
    , public AZ::TransformBus::Handler
{
public:
    // be sure this guid is unique, avoid copy-paste errors!
    AZ_COMPONENT(MockTransformComponent,
        "{7E8087BD-46DA-4708-ADB0-08D7812CA49F}");

    // Just a mock object, no reflection necessary
    static void Reflect(ReflectContext* ) {}

    // Mimic Transform component service
    static void GetProvidedServices(
```

```
AZ::ComponentDescriptor::DependencyArrayType& req)
{
    req.push_back(AZ_CRC("TransformService"));
}

// Mocking out pure virtual methods
void Activate() override
{
    AZ::TransformBus::Handler::BusConnect(GetEntityId());
}
void Deactivate() override
{
    AZ::TransformBus::Handler::BusDisconnect();
}

// OscillatorComponent will be calling these methods
MOCK_METHOD1(SetWorldTranslation, void (const AZ::Vector3&));
MOCK_METHOD0(GetWorldTranslation, AZ::Vector3 ());

// Unused methods but they are pure virtual in TransformBus
MOCK_METHOD0(IsStaticTransform, bool ());
MOCK_METHOD0(IsPositionInterpolated, bool ());
MOCK_METHOD0(IsRotationInterpolated, bool ());
MOCK_METHOD0(GetLocalTM, const Transform& ());
MOCK_METHOD0(GetWorldTM, const Transform& ());
MOCK_METHOD1(BindTransformChangedEventHandler,
    void(TransformChangedEvent::Handler&));
MOCK_METHOD1(BindParentChangedEventHandler,
    void(ParentChangedEvent::Handler&));
MOCK_METHOD1(BindChildChangedEventHandler,
    void(ChildChangedEvent::Handler&));
MOCK_METHOD2(NotifyChildChangedEvent,
    void(ChildChangeType, EntityId));
};

class OscillatorMockTest
    : public ::UnitTest::AllocatorsFixture
{
    AZStd::unique_ptr<AZ::SerializeContext> m_sc;
    AZStd::unique_ptr<AZ::ComponentDescriptor> m_md;
    AZStd::unique_ptr<AZ::ComponentDescriptor> m_od;

protected:
    void SetUp() override
    {
        ::UnitTest::AllocatorsFixture::SetUp();

        // register components involved in testing
        m_sc = AZStd::make_unique<AZ::SerializeContext>();
        m_md.reset(MockTransformComponent::CreateDescriptor());
        m_md->Reflect(m_sc.get());
        m_od.reset(OscillatorComponent::CreateDescriptor());
        m_od->Reflect(m_sc.get());
    }
}
```

```
void TearDown() override
{
    m_md.reset();
    m_od.reset();
    m_sc.reset();

    ::UnitTest::AllocatorsFixture::TearDown();
}

// helper method
void PopulateEntity(Entity& e)
{
    // OscillatorComponent is the component we are testing
    e.CreateComponent<OscillatorComponent>();
    // We can mock out Transform and test the interaction
    mock = new NiceMock<MockTransformComponent>();
    e.AddComponent(mock);

    // Bring the entity online
    e.Init();
    e.Activate();
}

MockTransformComponent* mock = nullptr;
};

TEST_F(OscillatorMockTest, Calls_SetWorldTranslation)
{
    Entity e;
    PopulateEntity(e);

    // setup a return value for GetWorldTranslation()
    ON_CALL(*mock, GetWorldTranslation()).WillByDefault(
        Return(AZ::Vector3::CreateZero()));

    // expect SetWorldTranslation() to be called
    EXPECT_CALL(*mock, SetWorldTranslation(_)).Times(1);

    TickBus::Broadcast(&TickBus::Events::OnTick, 0.1f,
                      ScriptTimePoint());
}
```

---

## **Part VII. Character Controller**

---

## Table of Contents

17. Player Input .....	142
Introduction .....	142
Starting Point Input Gem .....	142
Adding Input Mapping .....	142
Linking Against Starting Point Input Gem .....	144
Receiving Inputs in a C++ Component .....	145
Source Code .....	147
18. Character Movement .....	150
Introduction .....	150
PhysX Character Controller .....	150
Moving the Character .....	152
Source Code .....	155
19. Turning using Mouse Input .....	159
Introduction .....	159
Customizing Input Event Groups .....	159
Mouse Input .....	160
Source Code .....	162

---

# Chapter 17. Player Input

## Introduction

### Note

The accompanying source code for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch17\\_input](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch17_input)

One of the common elements in a game is a player controlled character. This part of the book will build a player controlled chicken that we added in Chapter 14, *Enabling NvCloth Gem*. But before we get to moving our chicken, I will show you how to capture player's input in O3DE.

## Starting Point Input Gem

O3DE comes with a gem that sets up the basic of input mapping - **Starting Point Input**. It is included in the default project template, so we have already have it enabled in `MyProject\Code\enabled_gems.cmake`.

```
set(ENABLED_GEMS
...
    StartingPointInput
...
)
```

## Adding Input Mapping

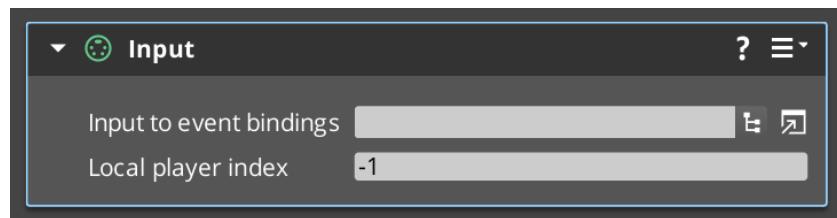
Starting Point Input gem provides us with **Input** component that maps input values to game actions.

### Note

The official document on Input component can be found here:

<https://www.o3de.org/docs/user-guide/interactivity/input/using-player-input/>

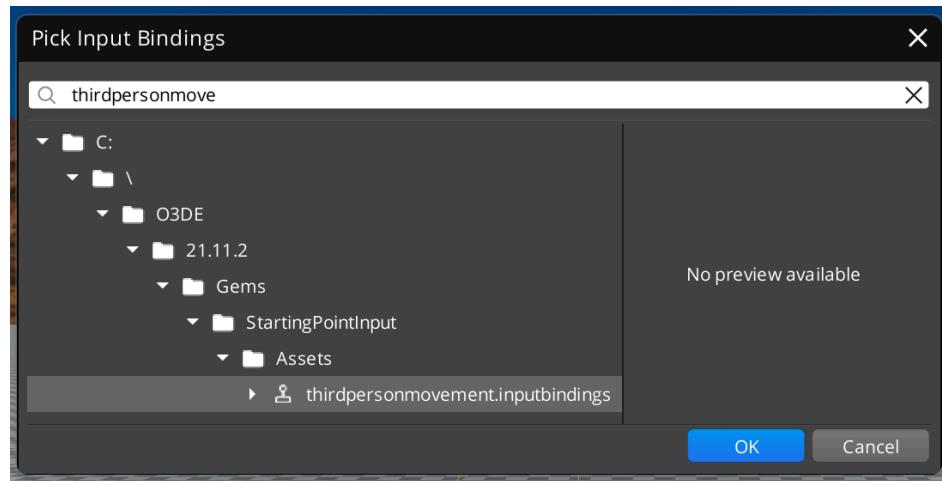
### Example 17.1. Empty Input component



1. Create a new entity in the level, **Input Map**.
2. Add **Input** component. **Local player index** is the value to point to which local player will receive the inputs. For now, keeping it at value "-1" means that the input will be sent to all local player controllers.

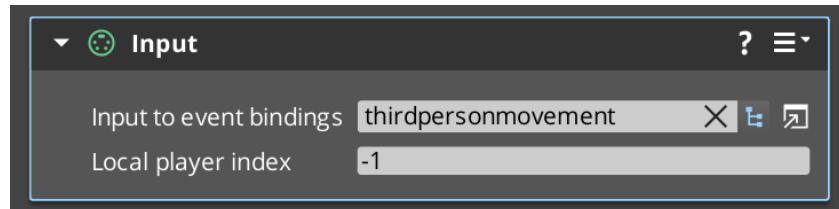
3. At the moment, there is no input bindings assigned. Assign one for **Input to event bindings** property. We can start with a template from Starting Point Input gem. It is located at C:\O3DE\21.11.2\Gems\StartingPointInput\Assets\thirdpersonmovement.inputbindings.

**Figure 17.1. Picking third person movement input bindings**



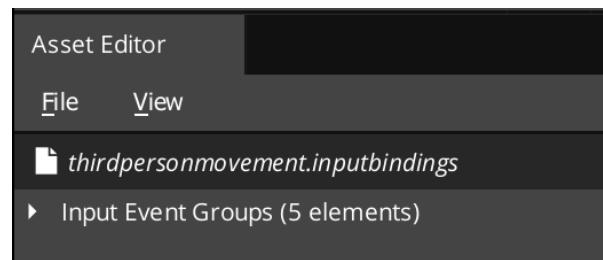
4. Input component should have thirdpersonmovement map assigned.

**Figure 17.2. Input component with third person movement bindings**

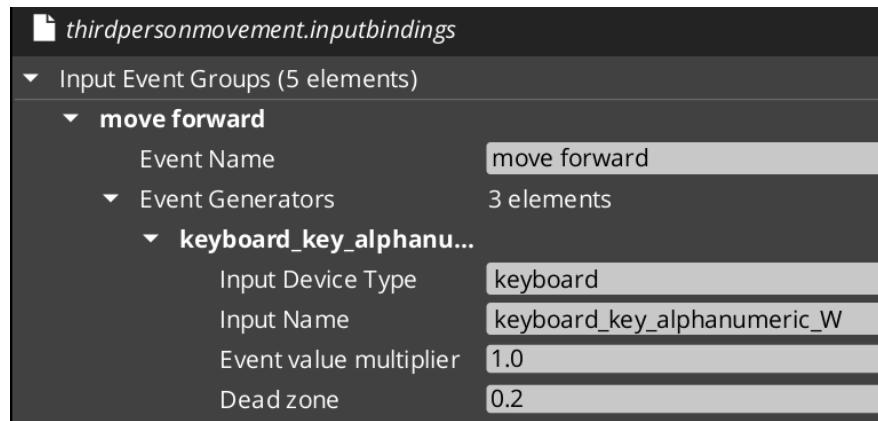


5. At this point, you can inspect the bindings using the right most icon next to **Input to event bindings**, which will open **Asset Viewer**.

**Figure 17.3. Asset Viewer**

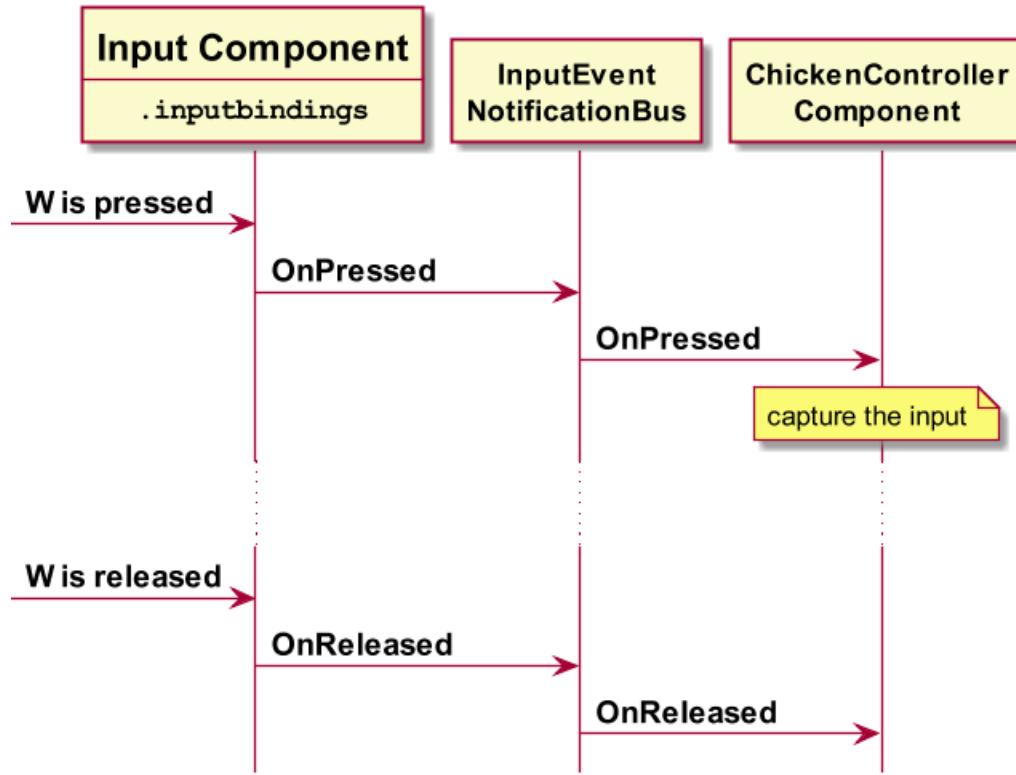


6. Open **Input Event Groups** and inspect *move forward* action.

**Figure 17.4. "Move forward" action**

7. We have arrived at the definition of *move forward* action, which is assigned keyboard key **W**.

Now that we have a clear path of how an action is triggered from a given input, let us take a look at how to capture it in a C++ component.

**Figure 17.5. Design of Capturing Input**

## Linking Against Starting Point Input Gem

We will be using one of Starting Point Input gem public interfaces, `InputEventNotificationBus`, to create a new component `ChickenControllerComponent`. In order to have access to the

header file that contains the ebus, we will need to link against the static library StartingPointInput.Static.

### Example 17.2. Changes to MyGem\Code\CMakeLists.txt

```
    ly_add_target(
        NAME MyGem.Static STATIC
        ...
        BUILD_DEPENDENCIES
        PUBLIC
            Gem::StartingPointInput.Static
        ...
    )
```

## Receiving Inputs in a C++ Component

1. Inherit from StartingPointInput::InputEventNotificationBus::MultiHandler.

```
class ChickenControllerComponent
: public AZ::Component
, public StartingPointInput::InputEventNotificationBus
    ::MultiHandler
```

2. Override OnPressed and OnReleased virtual methods.

```
// AZ::InputEventNotificationBus interface
void OnPressed(float value) override;
void OnReleased(float value) override;
```

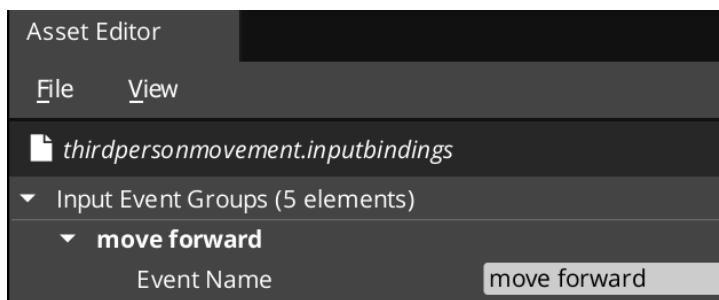
3. Define the action we are looking for.

```
const StartingPointInput::InputEventNotificationId
    MoveFwdEventId("move forward");
```

### ! Important

The value of "move forward" has to match the value of the appropriate **Event Name** in Input component binding.

**Figure 17.6. Input Binding for "move forward" in Asset Editor**



4. Receive the event in both OnPressed and OnRelease methods.

```

void ChickenControllerComponent::OnPressed(float value)
{
    const InputEventNotificationId* inputId =
        InputEventNotificationBus::GetCurrentBusId();
    if (inputId == nullptr)
    {
        return;
    }

    if (*inputId == MoveFwdEventId)
    {
        AZ_Printf("Chicken", "forward axis %f", value);
    }
}

```

5. Add ChickenControllerComponent to *Chicken\_Actor* entity in the level.

With these changes, you can press **W** and **S** to trigger movement input action.

### Example 17.3. Receiving Input

```

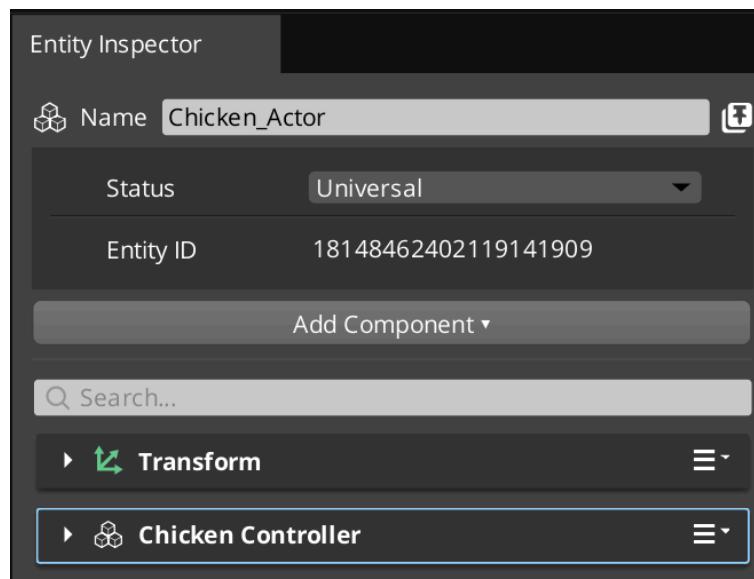
(Chicken) - forward axis -1.000000
(Chicken) - forward axis -0.000000
(Chicken) - forward axis -1.000000
(Chicken) - forward axis -0.000000
(Chicken) - forward axis 1.000000
(Chicken) - forward axis 0.000000

```

#### Note

Negative values are coming from pressing **S** key, which is mapped to the backward motion by sending *move forward* with a negative value.

**Figure 17.7. Chicken Actor with Chicken Controller component**



# Source Code

## Note

The accompanying source code for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch17\\_input](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch17_input)

This chapter added a new component, ChickenControllerComponent.

### Example 17.4. ChickenControllerComponent.h

```
#pragma once
#include <AzCore/Component/Component.h>
#include <StartingPointInput/InputEventNotificationBus.h>

namespace MyGem
{
    const StartingPointInput::InputEventNotificationId
        MoveFwdEventId("move forward");

    class ChickenControllerComponent
        : public AZ::Component
        , public StartingPointInput::
            InputEventNotificationBus::MultiHandler
    {
    public:
        AZ_COMPONENT(ChickenControllerComponent,
                     "{fe639d60-75c0-4e16-aa1a-0d44dbe6d339}");

        static void Reflect(AZ::ReflectContext* context);

        // AZ::Component interface implementation
        void Activate() override;
        void Deactivate() override;

        // AZ::InputEventNotificationBus interface
        void OnPressed(float value) override;
        void OnReleased(float value) override;
    };
} // namespace MyGem
```

### Example 17.5. ChickenControllerComponent.cpp

```
#include <ChickenControllerComponent.h>
#include <AzCore/Serialization/EditContext.h>

namespace MyGem
{
    using namespace StartingPointInput;

    void ChickenControllerComponent::Reflect(AZ::ReflectContext* rc)
```

```
{  
    if (auto sc = azrtti_cast<AZ::SerializeContext*>(rc))  
    {  
        sc->Class<ChickenControllerComponent, AZ::Component>()  
            ->Version(1);  
  
        if (AZ::EditContext* ec = sc->GetEditContext())  
        {  
            using namespace AZ::Edit;  
            ec->Class<ChickenControllerComponent>(  
                "Chicken Controller",  
                "[Player controlled chicken]")  
                ->ClassElement(ClassElements::EditorData, "")  
                ->Attribute(  
                    Attributes::AppearsInAddComponentMenu,  
                    AZ_CRC("Game"));  
        }  
    }  
  
    void ChickenControllerComponent::Activate()  
    {  
        InputEventNotificationBus::  
            MultiHandler::BusConnect(MoveFwdEventId);  
    }  
  
    void ChickenControllerComponent::Deactivate()  
    {  
        InputEventNotificationBus::MultiHandler::BusDisconnect();  
    }  
  
    void ChickenControllerComponent::OnPressed(float value)  
    {  
        const InputEventNotificationId* inputId =  
            InputEventNotificationBus::GetCurrentBusId();  
        if (inputId == nullptr)  
        {  
            return;  
        }  
  
        if (*inputId == MoveFwdEventId)  
        {  
            AZ_Printf("Chicken", "forward axis %f", value);  
        }  
    }  
  
    void ChickenControllerComponent::OnReleased(float value)  
    {  
        const InputEventNotificationId* inputId =  
            InputEventNotificationBus::GetCurrentBusId();  
        if (inputId == nullptr)  
        {  
            return;  
        }  
    }  
}
```

```
    if (*inputId == MoveFwdEventId)
    {
        AZ_Printf("Chicken", "forward axis %f", value);
    }
} // namespace MyGem
```

---

# Chapter 18. Character Movement

## Introduction

### Note

The accompanying source code for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch18\\_character\\_movement](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch18_character_movement)

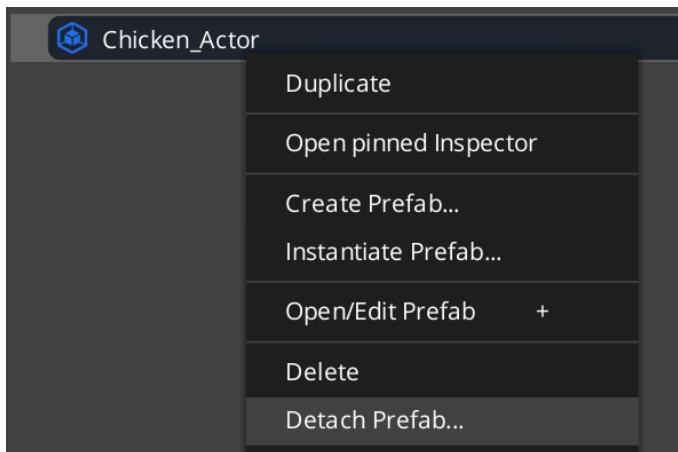
Previous chapter captured the input. This chapter will move the chicken based on that input by updating the code of Chicken Controller component. Pressing W or S will move the chicken forward and back, while pressing A or D will move it sideways left and right.

## PhysX Character Controller

In order to move the chicken we are going to use **PhysX Character Controller** component. Our project already enabled PhysX, so the component will show up. If it does not, check that PhysX gem is listed in `MyProject\Code\enabled_gems.cmake`:

```
set(ENABLED_GEMS
...
    PhysX
)
```

1. Detach chicken entities from the prefab. We are going to build a new chicken prefab.

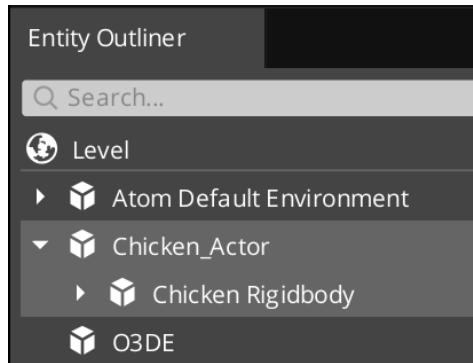


Right Click → Detach Prefab

2. This puts the entities out of the prefab and onto the level directly.

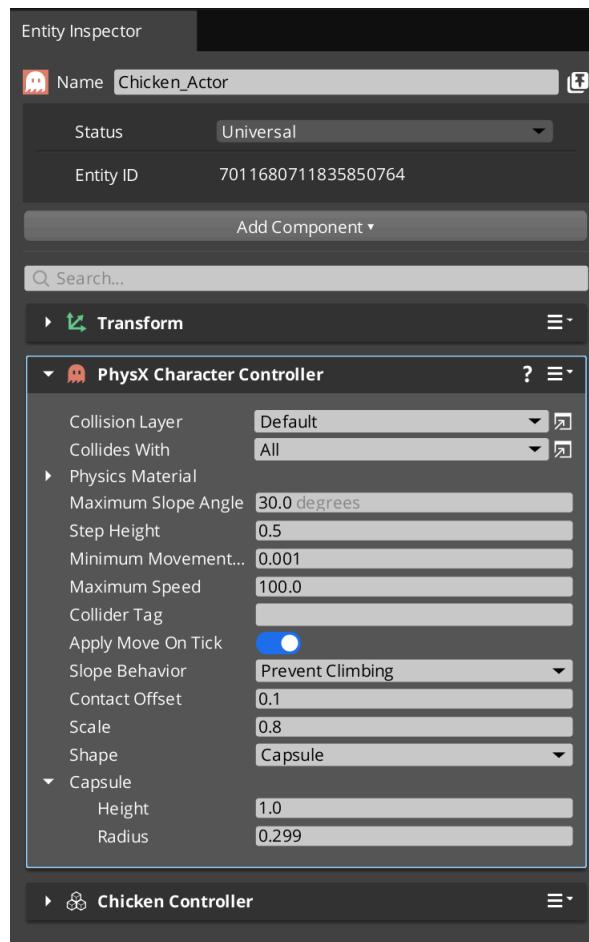
### Note

When you detach a prefab, the Editor will create a new entity with the name of the prefab and place all prefab entities under it. The prefab instance is then deleted, leaving the entities behind.



Detached Entities from `Chicken_Actor.prefab`

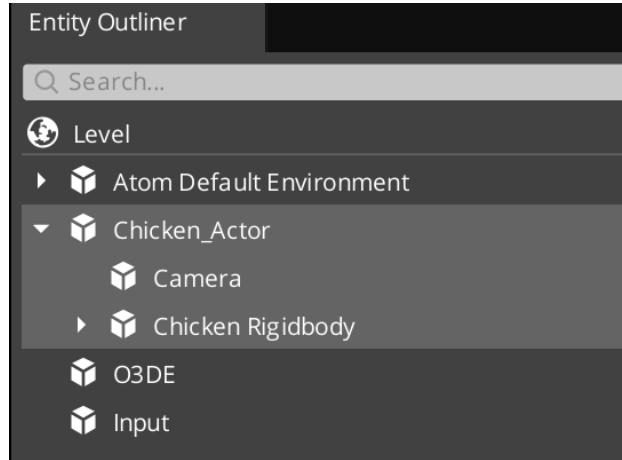
3. Add **PhysX Character Controller** component to **Chicken\_Actor** entity.



PhysX Character Controller component

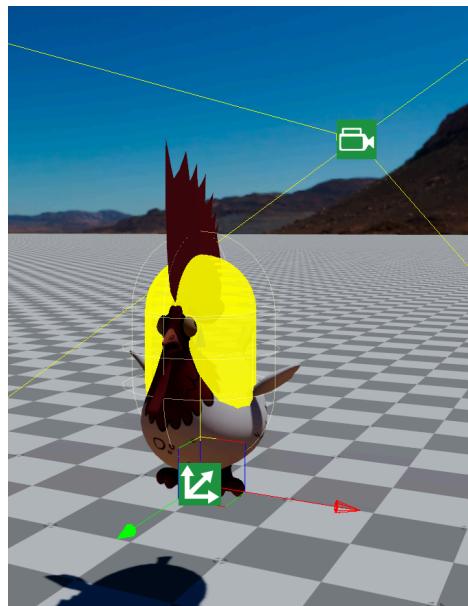
4. You have a few options for the shape of the controller on **Shape** property. I went with a capsule and modified the capsule shape to match the chicken, using **Height** and **Radius** properties from **Capsule** section.
5. Delete the camera entity from under **Atom Environment** entity.

6. Create an entity with a camera under **Chicken\_Action** entity. This way the camera will follow the chicken as it moves around the level.



Chicken with a Camera

7. Move the child entity *Camera* to some distance away from the chicken to create a comfortable third-person view.



Chicken with a Character Component

Now that the chicken character is setup with entities, we can improve `ChickenControllerComponent` to respond to player's input.

## Moving the Character

You can move the component by invoking the public API of PhysX Character Controller component. Its public interface can be found at `AzFramework\Physics\CharacterBus.h`. In this chapter, we are going to use `AddVelocity`:

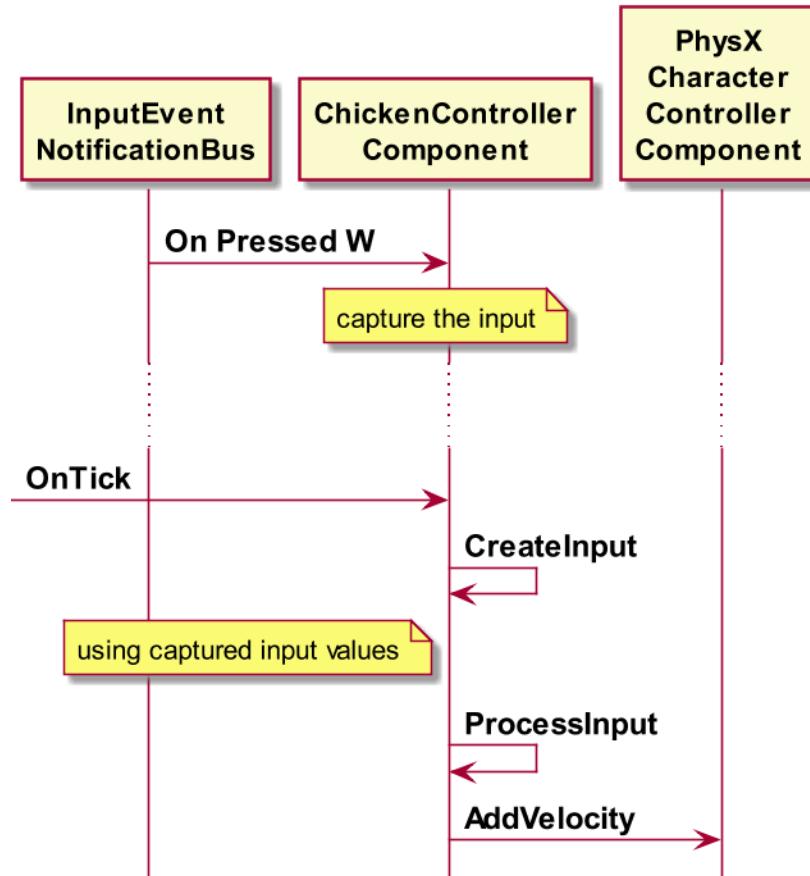
```
virtual void AddVelocity(const AZ::Vector3& velocity) = 0;
```

That adds some amount of velocity to the next frame to be simulated by PhysX, thus moving the character in the physical world.

### Tip

You can call `AddVelocity` multiple times per frame. The character will accumulate them together, such as adding player movement and gravity separately.

**Figure 18.1. Design of Moving the Chicken**



Here are the steps to implement forward and backward motion.

1. `ChickenControllerComponent` will sign up for game tick events.

```
class ChickenControllerComponent
: public AZ::Component
, public AZ::TickBus::Handler
```

2. On each tick, we will collect the input and process it.

```
void ChickenControllerComponent::OnTick(
    float, AZ::ScriptTimePoint)
{
    const ChickenInput input = CreateInput();
```

```
        ProcessInput(input);
    }
```

3. The input is initially saved in OnPressed and OnReleased methods.

```
void ChickenControllerComponent::OnPressed(float value)
{
    const InputEventNotificationId* inputId =
        InputEventNotificationBus::GetCurrentBusId();
    if (inputId == nullptr)
    {
        return;
    }

    if (*inputId == MoveFwdEventId)
    {
        m_forward = value;
    }
}
```

4. On tick, the input value is copied into a structure, ChickenInput.

```
ChickenInput ChickenControllerComponent::CreateInput()
{
    ChickenInput input;
    input.m_forwardAxis = m_forward;

    return input;
}
```

5. ChickenInput is processed and applied to the character using **PhysX Character Controller** component API.

```
void ChickenControllerComponent::ProcessInput(
    const ChickenInput& input)
{
    UpdateVelocity(input);

    Physics::CharacterRequestBus::Event(GetEntityId(),
        &Physics::CharacterRequestBus::Events::AddVelocity,
        m_velocity);
}
```

6. UpdateVelocity calculates the direction of the character's movement and saves the result into m\_velocity of type AZ::Vector3.

```
void ChickenControllerComponent::UpdateVelocity(
    const ChickenInput& input)
{
    float currentHeading = GetEntity()->GetTransform()->
        GetWorldRotationQuaternion().GetEulerRadians().GetZ();
    AZ::Vector3 fwd = AZ::Vector3::CreateAxisY(
        input.m_forwardAxis);
    m_velocity = AZ::Quaternion::CreateRotationZ(currentHeading) *
        TransformVector(fwd) * m_speed;
```

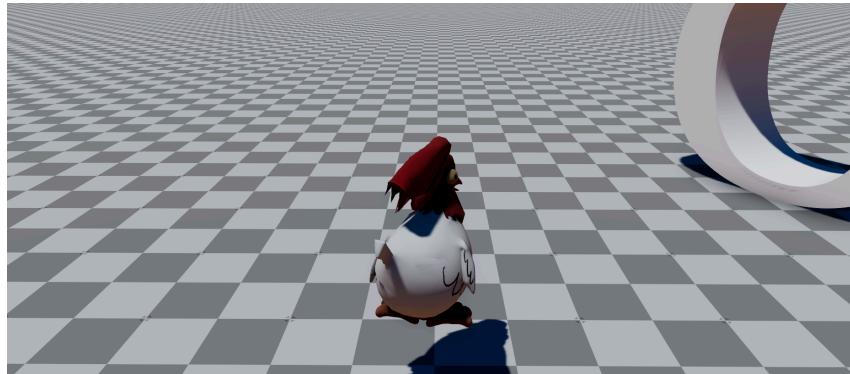
}

### >Note

The math here gets the direction the character is facing and creates a vector in that direction using player input value and speed of the chicken.

With these changes, the chicken will move forward when you press **W** and backward when you press **S**. The camera will follow along, since the camera component was attached to a child entity of the chicken.

**Figure 18.2. Chicken in game mode with CTRL+G**



We will add gravity to the chicken in Chapter 20, *Adding Physics to the World*.

## Source Code

### >Note

The accompanying source code for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch18\\_character\\_movement](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch18_character_movement)

Here is the updated code for `ChickenControllerComponent`. You can add strafing and other movement using the same approach by enhancing `CreateInput` and `ProcessInput`.

### Example 18.1. `ChickenControllerComponent.h` with character movement

```
#pragma once
#include <AzCore/Component/Component.h>
#include <AzCore/Component/TickBus.h>
#include <AzCore/Math/Vector3.h>
#include <StartingPointInput/InputEventNotificationBus.h>

namespace MyGem
{
    const StartingPointInput::InputEventNotificationId
        MoveFwdEventId("move forward");

    class ChickenInput
    {
        public:
```

```
    float m_forwardAxis = 0;
}

class ChickenControllerComponent
: public AZ::Component
, public AZ::TickBus::Handler
, public StartingPointInput::InputEventNotificationBus::MultiHandler
{
public:
    AZ_COMPONENT(ChickenControllerComponent,
        "{fe639d60-75c0-4e16-aala-0d44dbe6d339}");

    static void Reflect(AZ::ReflectContext* context);

    // AZ::Component interface implementation
    void Activate() override;
    void Deactivate() override;

    // AZ::InputEventNotificationBus interface
    void OnPressed(float value) override;
    void OnReleased(float value) override;

    // TickBus interface
    void OnTick(float deltaTime, AZ::ScriptTimePoint) override;

private:
    ChickenInput CreateInput();
    void ProcessInput(const ChickenInput& input);

    void UpdateVelocity(const ChickenInput& input);
    AZ::Vector3 m_velocity = AZ::Vector3::CreateZero();

    float m_speed = 6.f;
    float m_forward = 0.f;
};

} // namespace MyGem
```

### Example 18.2. ChickenControllerComponent.cpp with motion

```
#include <ChickenControllerComponent.h>
#include <AzCore/Component/Entity.h>
#include <AzCore/Component/TransformBus.h>
#include <AzCore/Serialization/EditContext.h>
#include <AzFramework/Physics/CharacterBus.h>

namespace MyGem
{
    using namespace StartingPointInput;

    void ChickenControllerComponent::Reflect(AZ::ReflectContext* rc)
    {
        if (auto sc = azrtti_cast<AZ::SerializeContext*>(rc))
```

```
{  
    sc->Class<ChickenControllerComponent, AZ::Component>()  
        ->Field("Speed", &ChickenControllerComponent::m_speed)  
        ->Version(1);  
  
    if (AZ::EditContext* ec = sc->GetEditContext())  
    {  
        using namespace AZ::Edit;  
        ec->Class<ChickenControllerComponent>(  
            "Chicken Controller",  
            "[Player controlled chicken]")  
            ->ClassElement(ClassElements::EditorData, "")  
            ->Attribute(  
                Attributes::AppearsInAddComponentMenu,  
                AZ_CRC_CE("Game"))  
            ->DataElement(nullptr,  
                &ChickenControllerComponent::m_speed,  
                "Speed", "Chicken's speed");  
    }  
}  
  
void ChickenControllerComponent::Activate()  
{  
    InputEventNotificationBus::MultiHandler::BusConnect(  
        MoveFwdEventId);  
    AZ::TickBus::Handler::BusConnect();  
}  
  
void ChickenControllerComponent::Deactivate()  
{  
    AZ::TickBus::Handler::BusDisconnect();  
    InputEventNotificationBus::MultiHandler::BusDisconnect();  
}  
  
void ChickenControllerComponent::OnPressed(float value)  
{  
    const InputEventNotificationId* inputId =  
        InputEventNotificationBus::GetCurrentBusId();  
    if (inputId == nullptr)  
    {  
        return;  
    }  
  
    if (*inputId == MoveFwdEventId)  
    {  
        m_forward = value;  
    }  
}  
  
void ChickenControllerComponent::OnReleased(float)  
{  
    const InputEventNotificationId* inputId =  
        InputEventNotificationBus::GetCurrentBusId();  
}
```

```
    if (inputId == nullptr)
    {
        return;
    }

    if (*inputId == MoveFwdEventId)
    {
        m_forward = 0.f;
    }
}

void ChickenControllerComponent::OnTick(float,
                                         AZ::ScriptTimePoint)
{
    const ChickenInput input = CreateInput();
    ProcessInput(input);
}

ChickenInput ChickenControllerComponent::CreateInput()
{
    ChickenInput input;
    input.m_forwardAxis = m_forward;

    return input;
}

void ChickenControllerComponent::UpdateVelocity(
    const ChickenInput& input)
{
    const float currentHeading = GetEntity()->GetTransform()->
        GetWorldRotationQuaternion().GetEulerRadians().GetZ();
    const AZ::Vector3 fwd = AZ::Vector3::CreateAxisY(
        input.m_forwardAxis);
    m_velocity = AZ::Quaternion::CreateRotationZ(currentHeading).
        TransformVector(fwd) * m_speed;
}

void ChickenControllerComponent::ProcessInput(
    const ChickenInput& input)
{
    UpdateVelocity(input);

    Physics::CharacterRequestBus::Event(GetEntityId(),
        &Physics::CharacterRequestBus::Events::AddVelocity,
        m_velocity);
}
} // namespace MyGem
```

---

# Chapter 19. Turning using Mouse Input

## Introduction

In Chapter 18, *Character Movement* we implemented character movement using **W** and **S** for moving forward and backward. In this chapter, I will cover using mouse motion to turn the character entity, which will turn the camera as well.

### Note

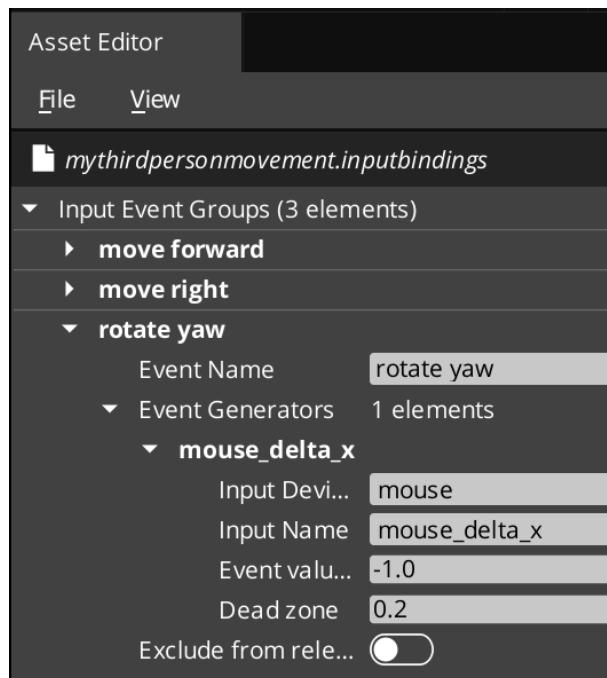
The accompanying source code for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch19\\_character\\_turning](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch19_character_turning)

## Customizing Input Event Groups

In Chapter 18, *Character Movement*, we used input binding file from O3DE installation at `C:\O3DE\21.11.2\Gems\StartingPointInput\Assets\thirdpersonmovement.inputbindings`. It would be a good idea to copy it, make it our own, and place it under our source control.

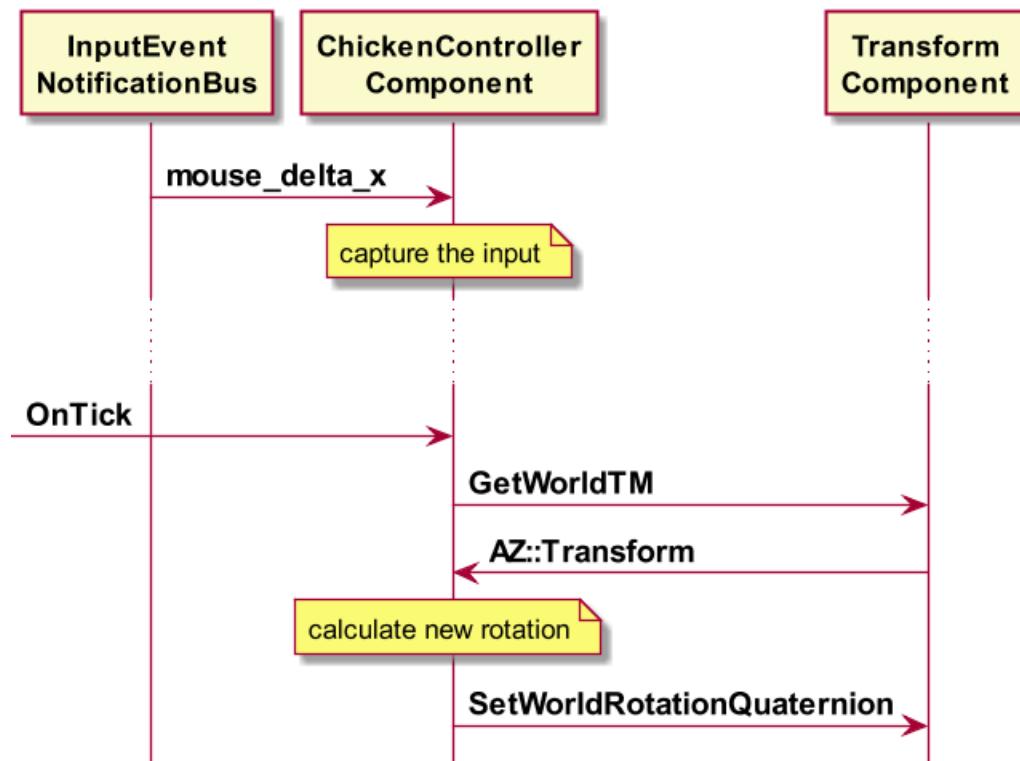
1. Copy `thirdpersonmovement.inputbindings` from `C:\O3DE\21.11.2\Gems\StartingPointInput\Assets\` to `C:\git\book\Gems\MyGem\Assets\` as `mythirdpersonmovement.inputbindings`.
2. Change **Input** component to point at `mythirdpersonmovement.inputbindings`.
3. Open Asset Editor for `mythirdpersonmovement.inputbindings`.
4. Remove all bindings except for *move forward*, *move right* and *rotate yaw*.



# Mouse Input

Input Event Groups allows you to specify the device and device input for each event. Here we have mouse device and **mouse\_delta\_x** input assigned to **rotate yaw** event. That means sideways motion of the mouse will be sent to the assigned event. Here is how to capture this mouse input and apply it.

**Figure 19.1. Turning using Mouse**



1. Create a variable to store the current state of **mouse\_delta\_x**.

```

class ChickenControllerComponent
{
...
    float m_yaw = 0.f;
};
  
```

2. Create an **InputEventNotificationId** for "rotate yaw".

```

const StartingPointInput::InputEventNotificationId
    RotateYawEventId("rotate yaw");
  
```

3. Sign up for the event with **InputEventNotificationBus**.

```

void ChickenControllerComponent::Activate()
{
...
    InputEventNotificationBus::MultiHandler::BusConnect(
        RotateYawEventId);
  
```

```
}
```

## ❗ Important

This step is critical. Otherwise you will not receive the mouse events at all for "rotate yaw".

4. Override virtual method `OnHeld` to capture mouse input.

```
void ChickenControllerComponent::OnHeld(float value)
{
    const InputEventNotificationId* inputId =
        InputEventNotificationBus::GetCurrentBusId();
    if (inputId == nullptr)
    {
        return;
    }

    if (*inputId == RotateYawEventId)
    {
        m_yaw = value;
    }
}
```

5. Update `CreateInput` to copy `m_yaw` value for processing.

```
ChickenInput ChickenControllerComponent::CreateInput()
{
    ChickenInput input;
    ...
    input.m_viewYaw = m_yaw;

    return input;
}
```

6. Update `ProcessInput` to update rotation of the chicken.

```
void ChickenControllerComponent::UpdateRotation(
    const ChickenInput& input)
{
    AZ::TransformInterface* t = GetEntity()->GetTransform();

    float currentHeading = t->GetWorldRotationQuaternion().
        GetEulerRadians().GetZ();
    currentHeading += input.m_viewYaw * m_turnSpeed;
    AZ::Quaternion q =
        AZ::Quaternion::CreateRotationZ(currentHeading);

    t->SetWorldRotationQuaternion(q);
}
```

With these changes, the chicken will turn based on mouse moving side to side. The camera will always stay behind the chicken, since it is locked to that orientation.

# Source Code

## >Note

The accompanying source code for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch19\\_character\\_turning](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch19_character_turning)

As a bonus, this source code also implements strafing. It is very similar in its implementation to moving forward and backward.

### Example 19.1. ChickenControllerComponent.h

```
#pragma once

#include <AzCore/Component/Component.h>
#include <AzCore/Component/TickBus.h>
#include <AzCore/Math/Vector3.h>
#include <StartingPointInput/InputEventNotificationBus.h>

namespace MyGem
{
    const StartingPointInput::InputEventNotificationId
        MoveFwdEventId("move forward");
    const StartingPointInput::InputEventNotificationId
        MoveRightEventId("move right");
    const StartingPointInput::InputEventNotificationId
        RotateYawEventId("rotate yaw");

    class ChickenInput
    {
    public:
        float m_forwardAxis = 0;
        float m_strafeAxis = 0;
        float m_viewYaw = 0;
    };

    class ChickenControllerComponent
        : public AZ::Component
        , public AZ::TickBus::Handler
        , public StartingPointInput::
            InputEventNotificationBus::MultiHandler
    {
    public:
        AZ_COMPONENT(ChickenControllerComponent,
                     "{fe639d60-75c0-4e16-aa1a-0d44dbe6d339}");

        static void Reflect(AZ::ReflectContext* rc);

        // AZ::Component interface implementation
        void Activate() override;
        void Deactivate() override;
    };
}
```

```
// AZ::InputEventNotificationBus interface
void OnPressed(float value) override;
void OnReleased(float value) override;
void OnHeld(float value) override;

// TickBus interface
void OnTick(float deltaTime, AZ::ScriptTimePoint) override;

private:
    ChickenInput CreateInput();
    void ProcessInput(const ChickenInput& input);

    void UpdateRotation(const ChickenInput& input);
    void UpdateVelocity(const ChickenInput& input);
    AZ::Vector3 m_velocity = AZ::Vector3::CreateZero();

    float m_speed = 10.f;
    float m_turnSpeed = 1.f;

    float m_forward = 0.f;
    float m_strafe = 0.f;
    float m_yaw = 0.f;
};

} // namespace MyGem
```

### Example 19.2. ChickenControllerComponent.cpp

```
#include <ChickenControllerComponent.h>
#include <AzCore/Component/Entity.h>
#include <AzCore/Component/TransformBus.h>
#include <AzCore/Serialization/EditContext.h>
#include <AzFramework/Physics/CharacterBus.h>

namespace MyGem
{
    using namespace StartingPointInput;

    void ChickenControllerComponent::Reflect(AZ::ReflectContext* rc)
    {
        if (auto sc = azrtti_cast<AZ::SerializeContext*>(rc))
        {
            sc->Class<ChickenControllerComponent, AZ::Component>()
                ->Field("Speed", &ChickenControllerComponent::m_speed)
                ->Field("Turn Speed",
                        &ChickenControllerComponent::m_turnSpeed)
                ->Version(2);

            if (AZ::EditContext* ec = sc->GetEditContext())
            {
                using namespace AZ::Edit;
                ec->Class<ChickenControllerComponent>(
                    "Chicken Controller",
                    "[Player controlled chicken]")
            }
        }
    }
}
```

```
    ->ClassElement(ClassElements::EditorData, "")
    ->Attribute(
        Attributes::AppearsInAddComponentMenu,
        AZ_CRC_CE("Game"))
    ->DataElement(nullptr,
        &ChickenControllerComponent::m_turnSpeed,
        "Turn Speed", "Chicken's turning speed")
    ->DataElement(nullptr,
        &ChickenControllerComponent::m_speed,
        "Speed", "Chicken's speed");
}
}

void ChickenControllerComponent::Activate()
{
    InputEventNotificationBus::MultiHandler::BusConnect(
        MoveFwdEventId);
    InputEventNotificationBus::MultiHandler::BusConnect(
        MoveRightEventId);
    InputEventNotificationBus::MultiHandler::BusConnect(
        RotateYawEventId);
    AZ::TickBus::Handler::BusConnect();
}

void ChickenControllerComponent::Deactivate()
{
    AZ::TickBus::Handler::BusDisconnect();
    InputEventNotificationBus::MultiHandler::BusDisconnect();
}

void ChickenControllerComponent::OnPressed(float value)
{
    const InputEventNotificationId* inputId =
        InputEventNotificationBus::GetCurrentBusId();
    if (inputId == nullptr)
    {
        return;
    }

    if (*inputId == MoveFwdEventId)
    {
        m_forward = value;
    }
    else if (*inputId == MoveRightEventId)
    {
        m_strafe = value;
    }
}

void ChickenControllerComponent::OnReleased(float value)
{
    const InputEventNotificationId* inputId =
        InputEventNotificationBus::GetCurrentBusId();
```

```
    if (inputId == nullptr)
    {
        return;
    }

    if (*inputId == MoveFwdEventId)
    {
        m_forward = value;
    }
    else if (*inputId == MoveRightEventId)
    {
        m_strafe = value;
    }
}

void ChickenControllerComponent::OnHeld(float value)
{
    const InputEventNotificationId* inputId =
        InputEventNotificationBus::GetCurrentBusId();
    if (inputId == nullptr)
    {
        return;
    }

    if (*inputId == RotateYawEventId)
    {
        m_yaw = value;
    }
}

void ChickenControllerComponent::OnTick(float,
                                         AZ::ScriptTimePoint)
{
    const ChickenInput input = CreateInput();
    ProcessInput(input);
}

ChickenInput ChickenControllerComponent::CreateInput()
{
    ChickenInput input;
    input.m_forwardAxis = m_forward;
    input.m_strafeAxis = m_strafe;
    input.m_viewYaw = m_yaw;

    return input;
}

void ChickenControllerComponent::UpdateRotation(
    const ChickenInput& input)
{
    AZ::TransformInterface* t = GetEntity()->GetTransform();

    float currentHeading = t->GetWorldRotationQuaternion().
        GetEulerRadians().GetZ();
```

```
currentHeading += input.m_viewYaw * m_turnSpeed;
AZ::Quaternion q =
    AZ::Quaternion::CreateRotationZ(currentHeading);

t->SetWorldRotationQuaternion(q);
}

void ChickenControllerComponent::UpdateVelocity(
    const ChickenInput& input)
{
    const float currentHeading = GetEntity()->GetTransform()->
        GetWorldRotationQuaternion().GetEulerRadians().GetZ();
    const AZ::Vector3 fwd = AZ::Vector3::CreateAxisY(
        input.m_forwardAxis);
    const AZ::Vector3 strafe = AZ::Vector3::CreateAxisX(
        input.m_strafeAxis);
    const AZ::Vector3 combined = (fwd + strafe).GetNormalized();
    m_velocity = AZ::Quaternion::CreateRotationZ(currentHeading).
        TransformVector(combined) * m_speed;
}

void ChickenControllerComponent::ProcessInput(
    const ChickenInput& input)
{
    UpdateRotation(input);
    UpdateVelocity(input);

    Physics::CharacterRequestBus::Event(GetEntityId(),
        &Physics::CharacterRequestBus::Events::AddVelocity,
        m_velocity);
}
} // namespace MyGem
```

---

## **Part VIII. Building Environment**

---

## Table of Contents

20. Adding Physics to the World .....	169
Introduction .....	169
Create a Soccer Field .....	169
Add a Static Mesh Collider .....	172
Add a Soccer Ball .....	174
Summary .....	175
21. Introduction to Materials and Lights .....	176
Introduction .....	176
Materials .....	176
Spot Lights .....	178
Disk Lights and Emissive Materials .....	179
Changing Base Color of a Material .....	181
Summary .....	182

---

# Chapter 20. Adding Physics to the World

## Introduction

Previous chapters have created a chicken controller. Our chicken can be made to run around and turn but there. There is one glaring defect. The chicken does not collide with any objects and is not affected by gravity. While we could easily add gravity using `AddVelocity` with the following example, it wouldn't be enough.

```
Physics::CharacterRequestBus::Event(GetEntityId(),  
    &Physics::CharacterRequestBus::Events::AddVelocity, m_gravity);
```

Our level does not have any physical object in it, so there is nothing for the chicken to bump into. It would just fall through the floor into oblivion. This chapter will address that by doing the following:

- Build a soccer field with static physical colliders.
- Add a static mesh collider to the 3D text of "O3DE" model.
- Build a soccer ball using a rigid physical body.
- Turn on gravity for the chicken.

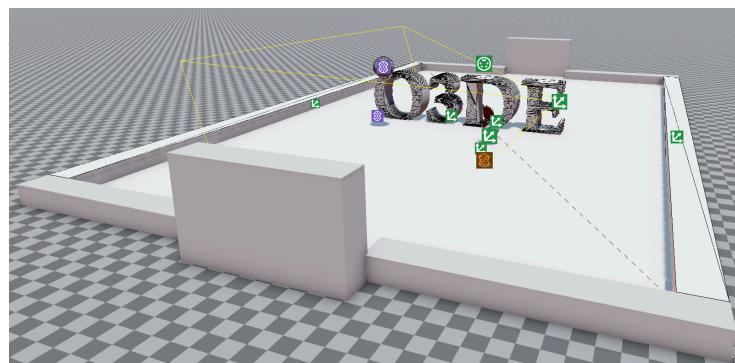
### Note

The accompanying assets for this chapter can be found on GitHub at:

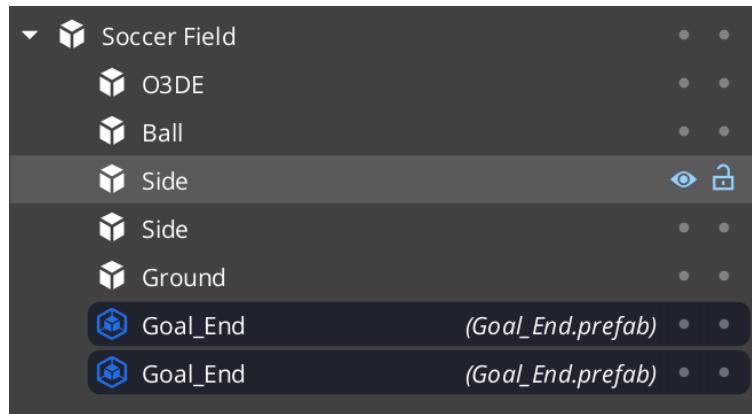
[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch20\\_physical\\_world](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch20_physical_world)

## Create a Soccer Field

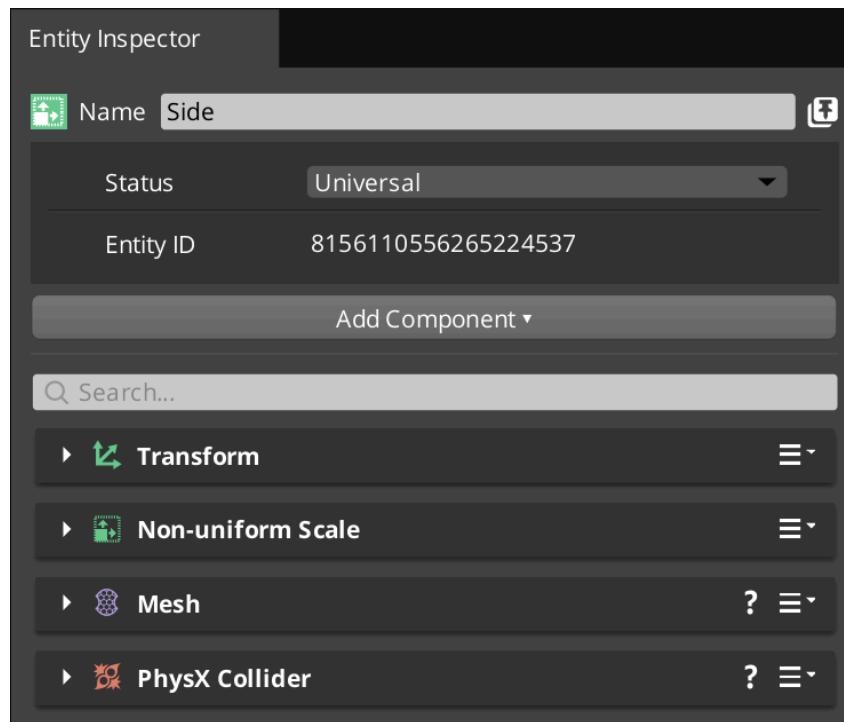
**Figure 20.1. Soccer Field**



We can build a soccer field out of boxes by resizing them and adding static physical colliders on them. Here is the entity outline of the soccer field.

**Figure 20.2. Soccer Field Entities**

**Side** and **Ground** entities have the same structure but with a different scale.

**Figure 20.3. Components of Side Entity**

They have the following components:

- **Transform** component to position the part.
- **Non-uniform Scale** component to size the part to the required dimensions.

**Figure 20.4. Non-uniform Scale component**

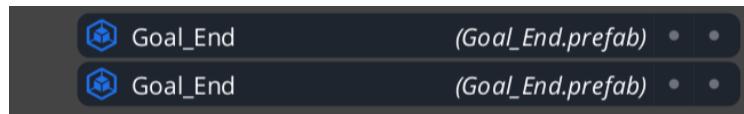
- **Mesh** component with a box model that is being resized by Non-uniform Scale component.
- **PhysX Collider** component to add a static immovable physical shape to match the scaled mesh.

**Figure 20.5. PhysX Collider Mesh Asset**

### Note

There is no need to scale the value of the PhysX Collider, as they will be scaled by **Non-uniform Scale** component.

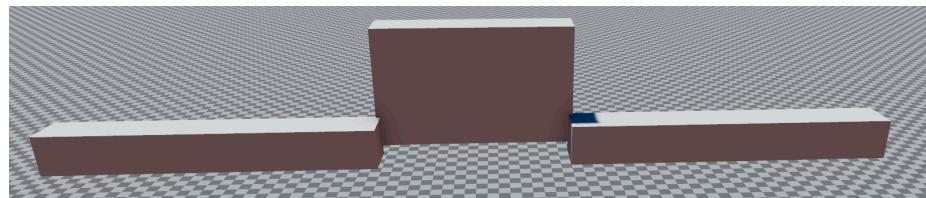
As I was building the soccer field, I noticed that there was a pattern of entities of building the goal line on each side. So in order to save myself the effort, I created a prefab out of one side, duplicated and moved it to the opposite end of the soccer field.

**Figure 20.6. Soccer Field parts as prefabs**

Inside them there are more entities of a similar design as **Side** and **Ground** entities.

**Figure 20.7. Goal\_End.prefab**

There are three scaled boxes to create the goal line and edges of the field.

**Figure 20.8. Goal\_End shapes**

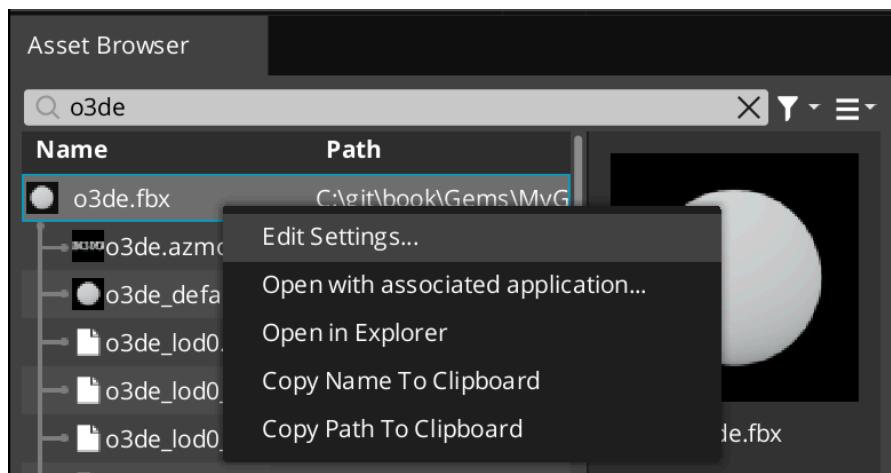
## Add a Static Mesh Collider

A more interesting question is how to add a physical collider to 3D text of "O3DE".

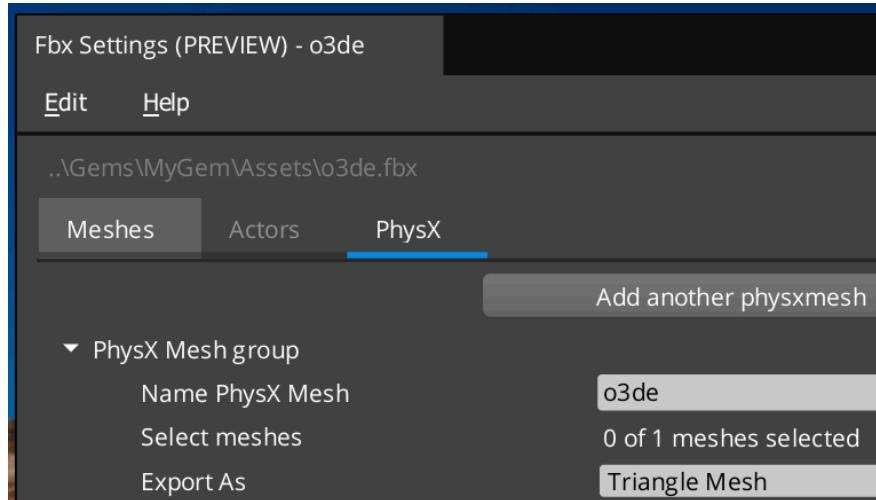
**Figure 20.9. 3D text: O3DE**

I created this model in Blender and exported it as .FBX file to the project but that does not automatically provide a physical shape for us. We can generate one using the following tools.

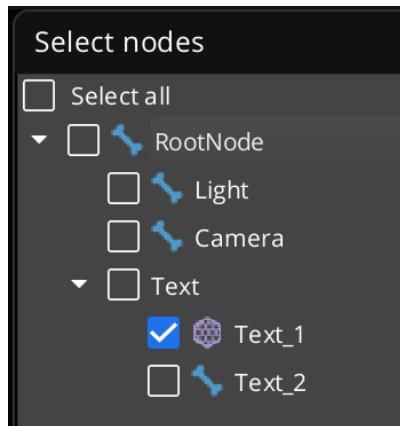
1. Open Asset Browser.
2. Find o3de.fbx asset.
3. Right click on o3de.fbx and choose **Edit Settings**.



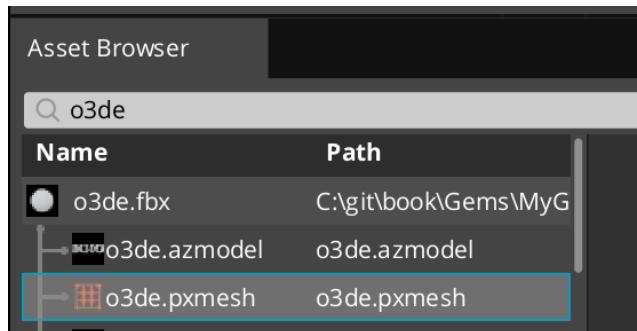
4. That will open **Fbx Settings** dialog, select **PhysX** tab.



5. Under **PhysX Mesh group**, open **Select meshes**.
6. Enable the mesh node, **Text\_1**. Click **Select** to save the selection.

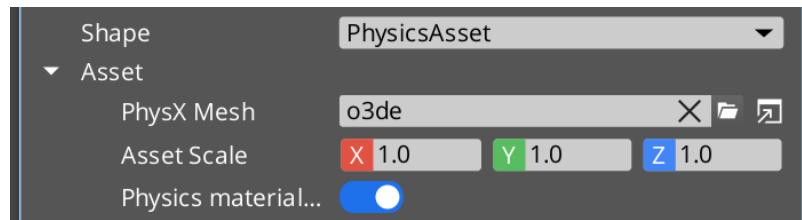


7. Click **Update** to produce a PhysX shape using **Text\_1** mesh node.



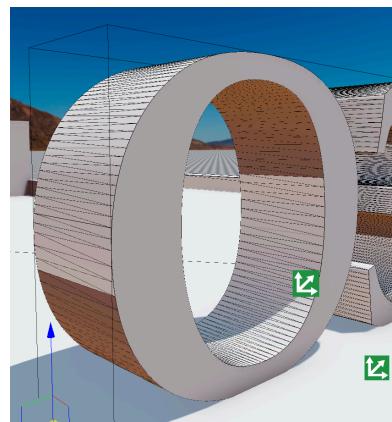
8. Asset Browser should show **o3de.pxmesh** under **o3de.fbx**.
9. Go back to the entity with O3DE 3D text.
10. On its PhysX Collider component, assign **Shape** type to **PhysicsAsset**.

11. Assign **PhysX Mesh** to o3de.pxmesh.



With these changes, you should see triangles form over the 3D text. That is the outline of the physical mesh over the text. The text has become physical.

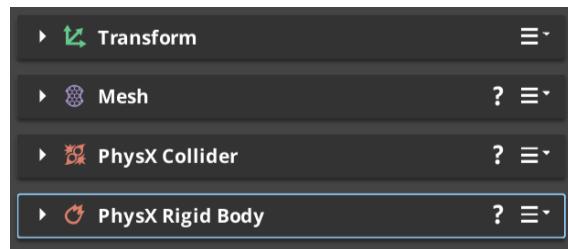
**Figure 20.10. O3DE physical mesh**



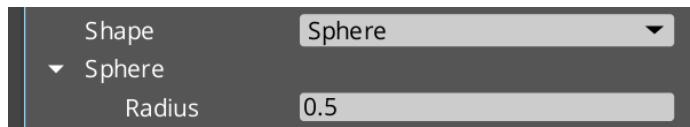
## Add a Soccer Ball

So far all the shapes we have added for the soccer field have been immovable static shapes but a soccer ball needs to be dynamic to move around as the chicken will be kicking it around. For that, we need **Physx Rigid Body** component.

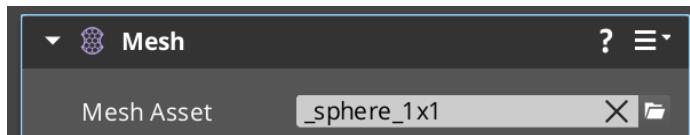
**Figure 20.11. Ball components**



Default values on **PhysX Rigid Body** component will work as is. Set **PhysX Collider** to the shape of Sphere with a radius of 0.5.

**Figure 20.12. Ball shape**

That will match size of `_spheres_1x1` mesh that comes with the engine.

**Figure 20.13. Ball mesh**

If you place this **Ball** entity on top of the soccer field, you will see it fall, collide with static shapes and roll around.

**Figure 20.14. Soccer Ball**

### Note

You might notice that running the chicken into the ball does not move the ball. The chicken is moved using PhysX character controller. We will need to do some extra work with physical triggers to detect collisions and apply impulse to the soccer ball. That will be done in Chapter 24, *Kicking the Ball*.

## Summary

We created a physical world for the chicken. Now we can apply gravity to the chicken by updating `ProcessInput` method with an additional call to `AddVelocity`.

```
void ChickenControllerComponent::ProcessInput(
    const ChickenInput& input)
{
    ...
    Physics::CharacterRequestBus::Event(GetEntityId(),
        &Physics::CharacterRequestBus::Events::AddVelocity,
        AZ::Vector3::CreateAxisZ(m_gravity));
}
```

### Note

The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch20\\_physical\\_world](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch20_physical_world)

# Chapter 21. Introduction to Materials and Lights

## Introduction

It is time to spice up our level. In previous chapter we built the soccer field using boxes with a default white material. We can add a lot of color and lights to the level using only the basic engine materials and lights.

### Note

The accompanying assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch21\\_materials\\_lights](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch21_materials_lights)

## Materials

### Note

You will find a reference for O3DE materials online at:

<https://www.o3de.org/docs/atom-guide/look-dev/materials/>

We will start by modifying the material of the ground mesh that came with Atom Default Environment entity.

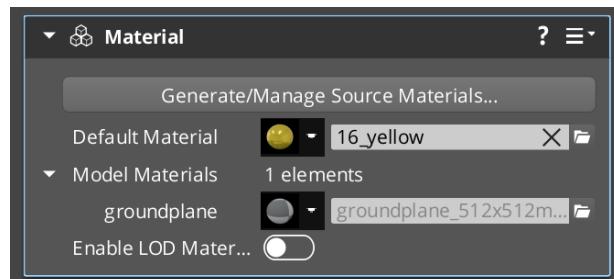
1. Select Atom Default Environment, then select Ground entity under it.

**Figure 21.1. Entity Atom Default Environment with Ground child entity**



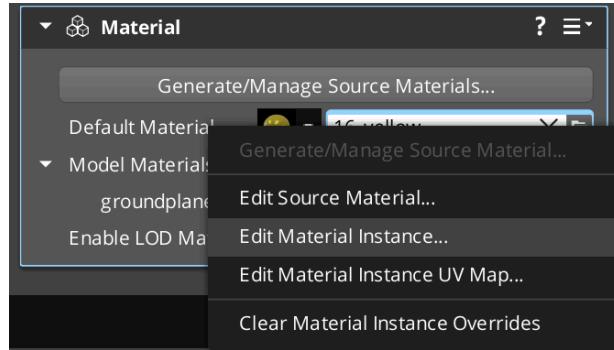
2. Find **Material** component.
3. Change **Default Material** to "16\_yellow". (This material comes with O3DE inside Atom renderer gems.)

**Figure 21.2. Soccer Ball**



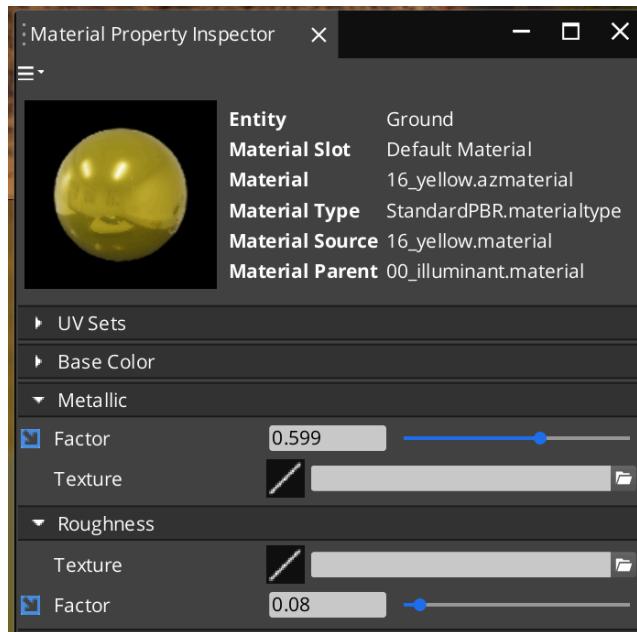
- Click on the sphere icon next to Default Material and then select **Edit Material Instance**.

**Figure 21.3. Soccer Ball**



- This will open **Material Property Inspector** where you can change various properties of this particular material on this specific entity. This way you are modifying the material only for this entity without affecting other entities that may use the same source material.
- I chose to set **Metallic** and **Roughness** factors to my liking.

**Figure 21.4. Soccer Ball**



### Tip

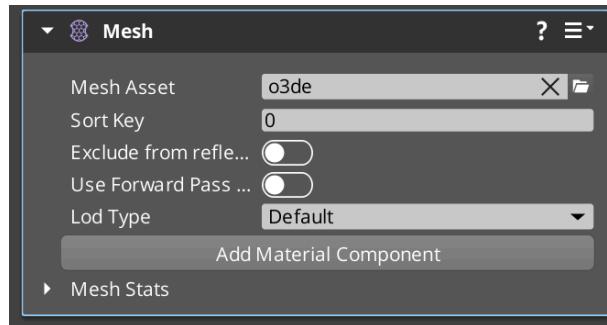
O3DE comes with Atom renderer that supports PBR textures (Physically Based Rendering). You can find or make your own PBR textures and assign their component textures, such as Base Color, Metallic, Roughness textures and so on. In this chapter, I am working with simple color textures only.

In the similar fashion as above, I have assigned O3DE mesh *13\_blue* PBR material, *09\_moderate\_red* to the sides of the soccer field and *14\_green\_tex* to the soccer ground mesh.

## ⚠ Important

Adding a Mesh component does not add Material component by default. You have to add Material component yourself to an entity, otherwise the default material for the mesh is used. The easiest way is to use the button **Add Material Component** on Mesh component.

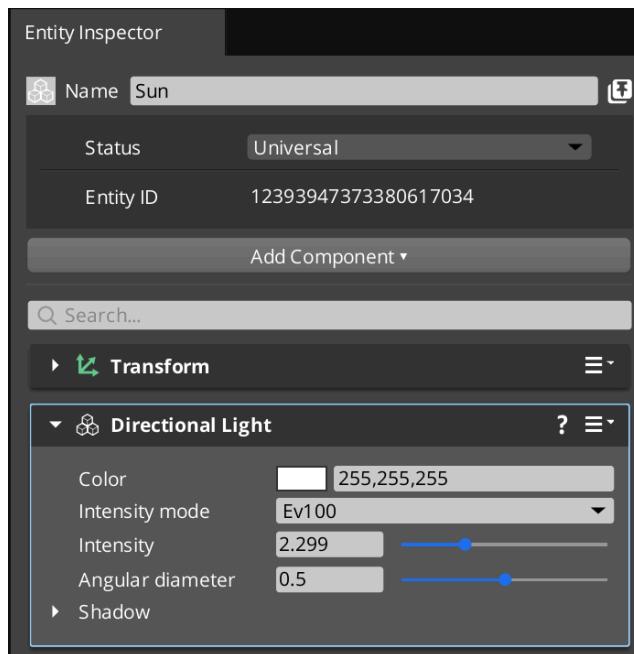
**Figure 21.5. `Add Material Component` button**



## Spot Lights

With some basic material assigned, we can turn towards adding a few fancy lights. Our level already came with a skybox, skylight and a direction light. You can find them under **Atom Default Environment** entity. For example, **Sun** child entity has a **Directional Light** component.

**Figure 21.6. Sun entity**



What I am interested is in adding a spot light to each goal line.

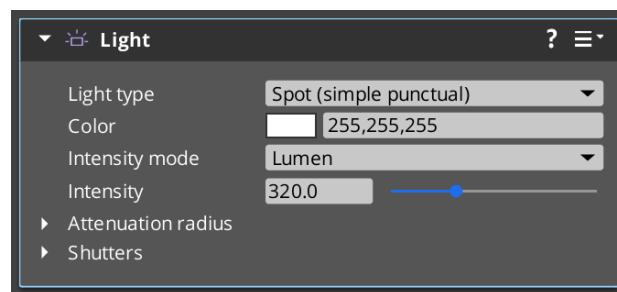
1. Open **Goal\_End** prefab.
2. Add **Spot Light** child entity to **Back** entity.

**Figure 21.7. Add Spot Light child entity**



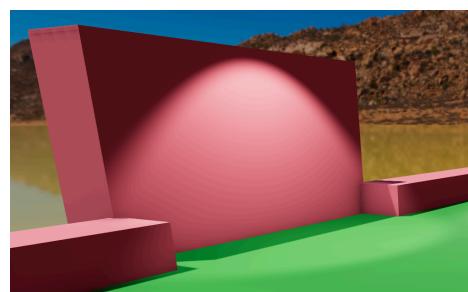
3. Add a **Light** component to Spot Light entity.
4. For **Light type**, choose **Spot (simple punctual)**.

**Figure 21.8. Light component with Spot type**



5. Use transform tools to point the spot light at the goal line.

**Figure 21.9. Spot light on the goal line**

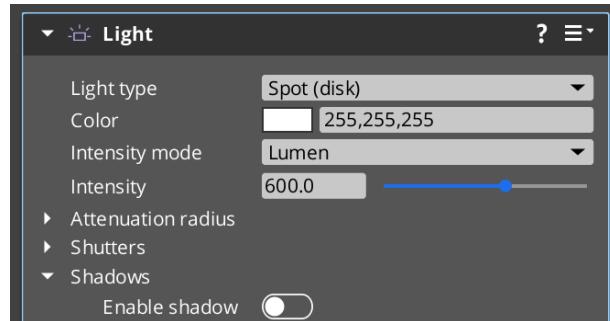


6. Save the prefab.

## Disk Lights and Emissive Materials

Here is a fun way to combine lights with materials to make a lamp.

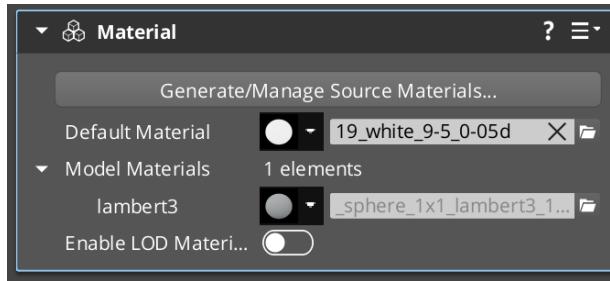
1. Using the same steps as above, create a **Spot (disk)** light.

**Figure 21.10. Spot disk light**

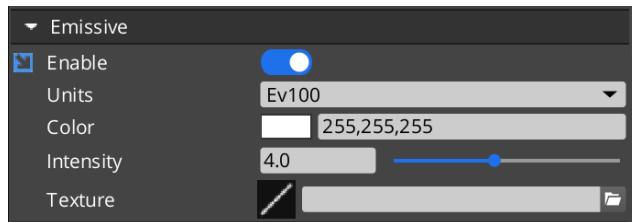
### Tip

You can selectively enable shadow support for each Light source to control the performance and look of your game. Use **Enable shadow** on Light component.

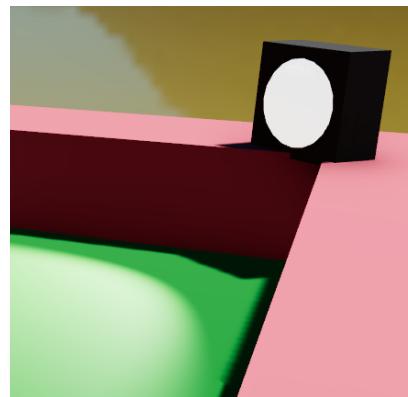
2. Create a mesh with a sphere that is squished along one axis, so it looks like a lamp source.
3. Choose a material for this mesh, for example *19\_white*.

**Figure 21.11. White material on a mesh**

4. Enable **Emissive** property on the material instance.

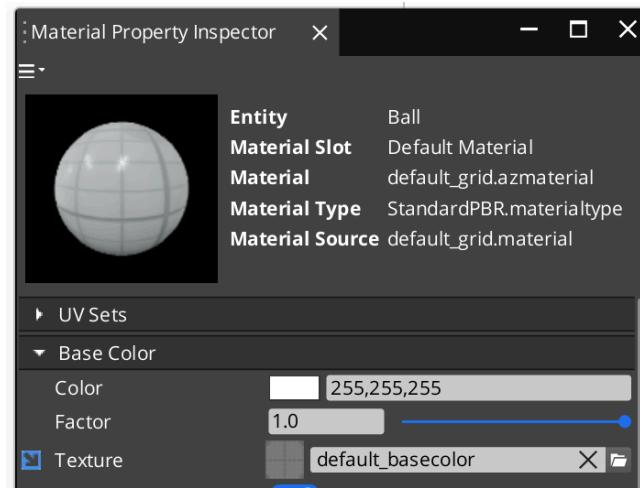
**Figure 21.12. Spot disk light**

5. Add a back mesh of your choice and style to create a flood lamp for the soccer field.

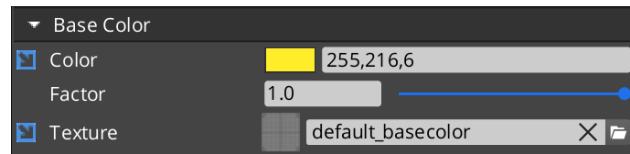
**Figure 21.13. Corner flood lamp**

## Changing Base Color of a Material

For the soccer ball, I do not want to use a solid color, since I want to see how the ball spins. Luckily, even the default project has access to **default\_grid** material, which will work for the chicken soccer ball.

**Figure 21.14. Default\_grid material**

I am not interested in a grey ball, though. So I will use **Base Color** section to change **Color** to yellow.

**Figure 21.15. Yellow base color**

# Summary

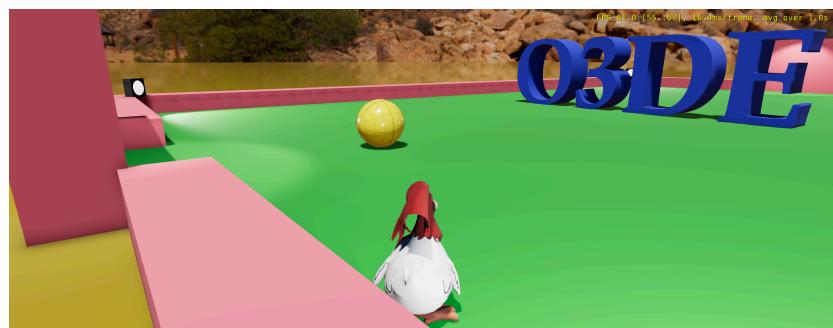
## >Note

The accompanying assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch21\\_materials\\_lights](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch21_materials_lights)

Our very brief introduction to materials and lights is over. There is a lot more to explore in these topics but this is enough to get us started. Here is how the level looks like a few tweaks to materials and addition of a few lights.

**Figure 21.16. New level look**



## >Note

The online documentation on materials can be found here:

<https://www.o3de.org/docs/atom-guide/look-dev/materials/>

The documentation on **Light** component is here:

<https://www.o3de.org/docs/user-guide/components/reference/atom/light/>

---

## **Part IX. Building Game Play**

---

# Table of Contents

22.	Introduction to User Interface .....	185
	Introduction .....	185
	Loading UI .....	186
	Creating a Canvas in UI Editor .....	186
	Drawing UI Canvas .....	190
	Summary .....	191
23.	Interacting with UI in C++ .....	192
	Introduction .....	192
	Physical Triggers .....	192
	Sign up for Trigger Events .....	194
	Receiving Updates in UI .....	197
	Moving a Rigid Body .....	198
	Summary .....	199
24.	Kicking the Ball .....	204
	Introduction .....	204
	The Root Cause of the Issue .....	204
	Trigger Shape on the Chicken .....	204
	Kicking Component .....	206
	Source Code .....	209
25.	Introduction to Animation .....	212
	Introduction to EMotionFX .....	212
	Simple Motion Component .....	212
	Anim Graph Component .....	213
	Building an Animation Graph .....	213
	Animation Blending .....	218
	Assigning the Anim Graph .....	221
	Driving Speed Parameter .....	221
	Source Code .....	222
26.	Animation State Machine .....	225
	Introduction .....	225
	Adding Flapping Motion .....	225
	Adding Celebration Blend Tree .....	225
	Summary .....	230
27.	Script Canvas .....	231
	Introduction .....	231
	Behavior Context .....	231
	Building a Canvas .....	233
	Summary .....	235

---

# Chapter 22. Introduction to User Interface

## Note

The level and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch22\\_basic\\_ui](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch22_basic_ui)

## Introduction

O3DE comes with its own User Interface subsystem, LyShine. The core of the system is implemented in **LyShine** gem (which is already enabled in MyProject) but it is also useful to enable two more supporting gems for LyShine.

### Example 22.1. Enable LyShine and its supporting gems

```
set(ENABLED_GEMS
...
    LyShine
    LyShineExamples      # new
    UiBasics            # new
)
```

## Note

Official reference for LyShine can be found at

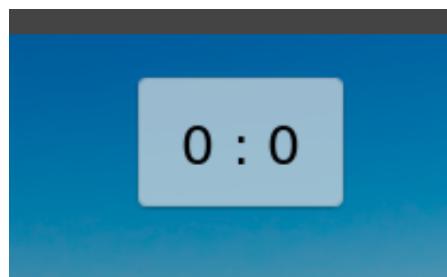
<https://www.o3de.org/docs/user-guide/interactivity/user-interface/>

LyShineExamples and UiBasics gems provide additional canvas examples and UI elements. Their assets are inside O3DE installation at Gems\UiBasics\Assets\UI\Slices\Library and Gems\LyShineExamples\Assets\UI\Canvases\LyShineExamples.

## Important

After you enable these gems, re-build the project in Visual Studio or with CMake, close the Asset Process and open it again. It should have a few hundred new assets to process. Otherwise, you will not see new UI elements.

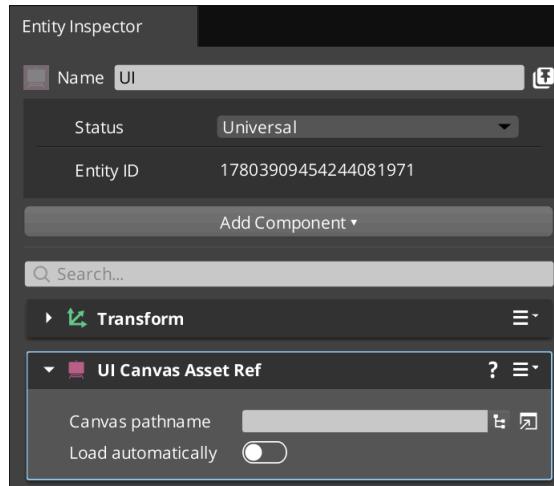
In this chapter, we will create an overlay to display two numbers. This overlay will be used to keep score of the soccer match.



# Loading UI

User Interface work begins by using **UI Canvas Asset Ref** component to load a UI asset.

1. Create a new entity in the level.
2. Add **UI Canvas Asset Ref** component.



3. Assign the canvas or create a new one.

## Tip

You can open UI editor from this component by clicking on the right most icon button next to **Canvas pathname** property.

4. Flip on **Load automatically** or load the canvas using C++ API.

```
UiCanvasAssetRefBus::Event(entityId,
    &UiCanvasAssetRefBus::Events::LoadCanvas);
```

# Creating a Canvas in UI Editor

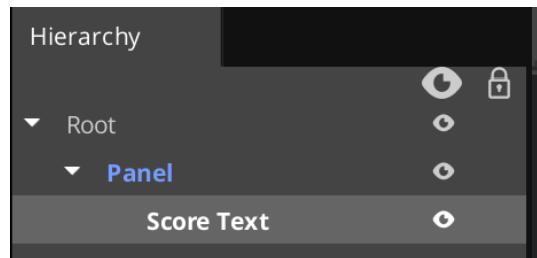
UI Editor is one the tools of O3DE Editor. You can bring it up via main menu with **Tools→UI Editor**.

That will opens UI Editor with an empty canvas. A canvas is a container for UI elements that are drawn together. Before we get started, let us save the new empty canvas to our project location. We will save it to `C:\git\book\MyProject\Assets\UI\hud.uicanvas`.

## Note

The location is optional within a project and its gems. The Asset Processor will find the asset regardless of where you put it under your project or under one of your gems but it is a good practice to follow some standard.

**Hierarchy** panel on the left lists the hierarchy of UI entities. The underlying system uses O3DE entities and components. **Hierarchy** panel is essentially another version of **Entity Outliner**. Let us create a canvas with elements placed at that top middle portion of the canvas.

**Figure 22.1. Hierarchy of elements**

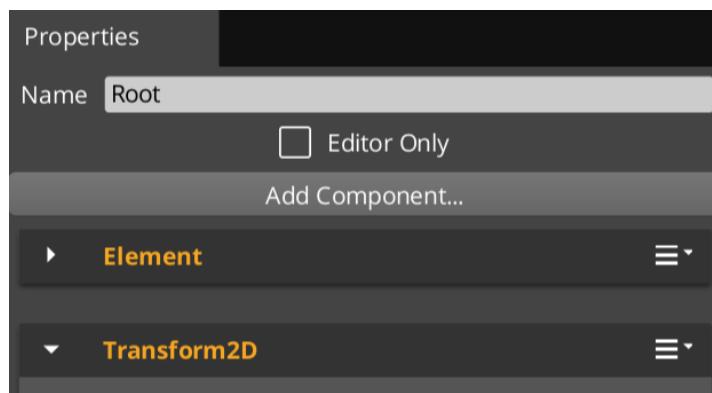
1. Right click on empty space in **Hierarchy**, then choose **New** and **Empty element**.
2. Name this element *Root* by double clicking on it. It will serve as the parent of all other UI elements.
3. Right click on Root element, then **New** and **Element from Slice Browser**.
4. Search for "panel" and select "Panel.slice."
5. Right click on Panel element, select "New", "Element from Slice Library." From group "UiBasics\Assets\UT\Slices\Library" select **Text**.
6. Rename Text element to "Score Text."

### Tip

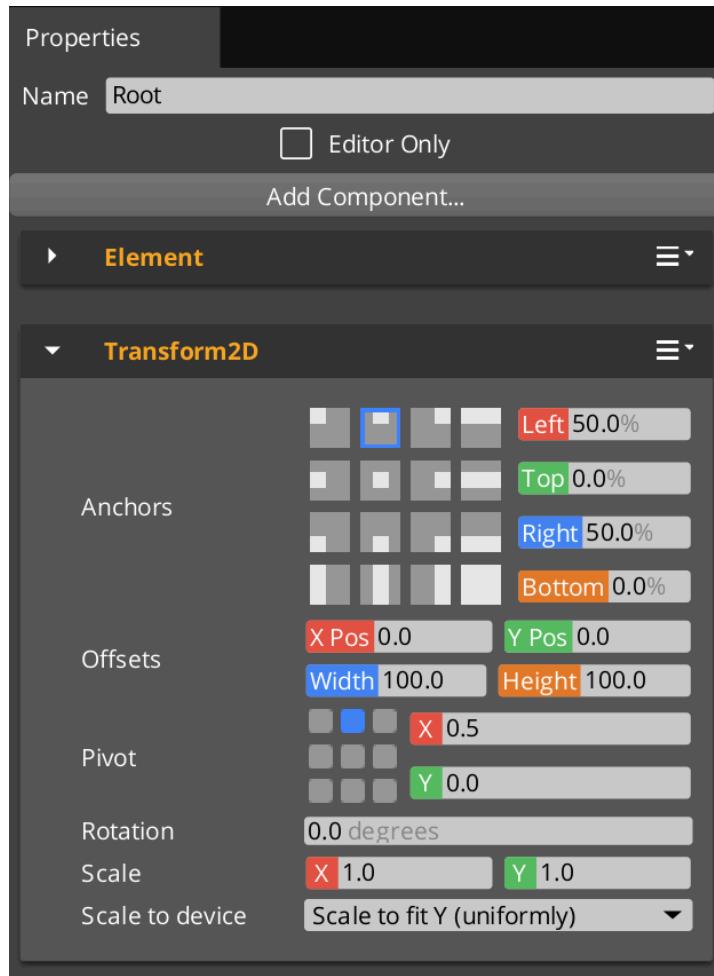
The order of the elements determines which one is drawn first. The root element is drawn first with leaf elements drawn last.

We have the outline of the elements we need but the panel is too big and the text is in the center. The way to move and modify elements is by using their **Transform2D** component and through Mode toolbar.

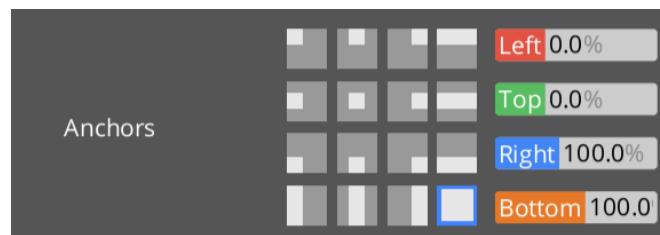
1. Select "Root" element.
2. Head over to properties and find **Transform 2D** component.



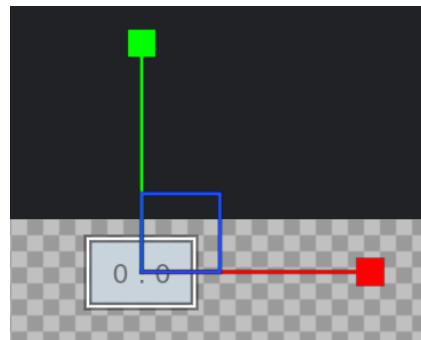
3. At the bottom of Transform 2D component, find **Scale to device** property, set it to "Scale to fit Y (uniformly)". That will fit the contents to the size of the viewport screen. That means you do not have to worry about setting pixel values and can think of any elements underneath as having relative proportional values to the actual viewport screen size.
4. Then choose top middle anchor. It is a small icon under Anchors section with a white box inside a larger gray box.

**Figure 22.2. Root element**

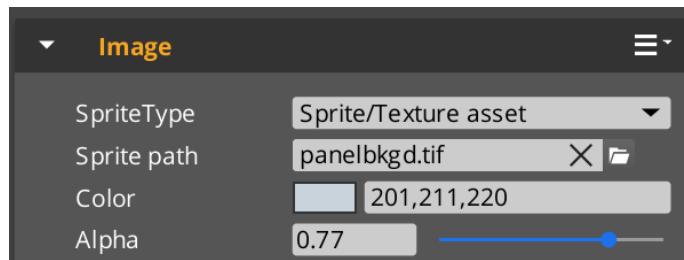
5. Select "Panel" element under "Root" element.
6. Select the anchor that fills the entire available space. This anchor icon looks like a full white square.

**Figure 22.3. Panel Anchors**

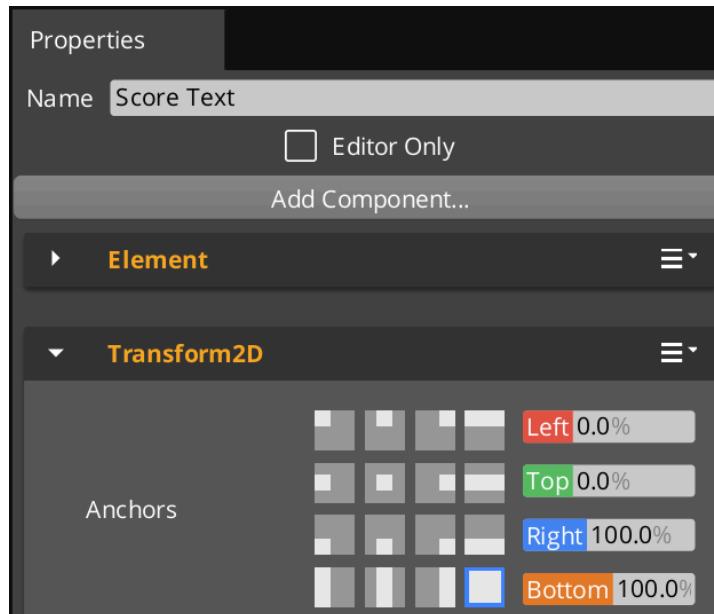
7. With "Panel" element selected, in Mode toolbar, select scaling tool and reduce the vertical size of the element by dragging the green tool line with a square at the top.

**Figure 22.4. Scaling an element**

- With "Panel" element selected, in **Image** component, lower Alpha value to less than 1.0. (0.77 was my choice.) This will make the panel semi-transparent.

**Figure 22.5. Image's alpha value**

- In Hierarchy panel select "Score Text" element, then apply the anchor that fills all available space. (It is the bottom right anchor icon.)

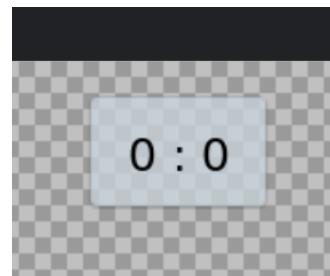
**Figure 22.6. Score Text element**

10. With "Score Text" element selected, go to its **Text** component and modify Text value to "0 : 0". You can change font and font color as well.

**Figure 22.7. Score Text element**

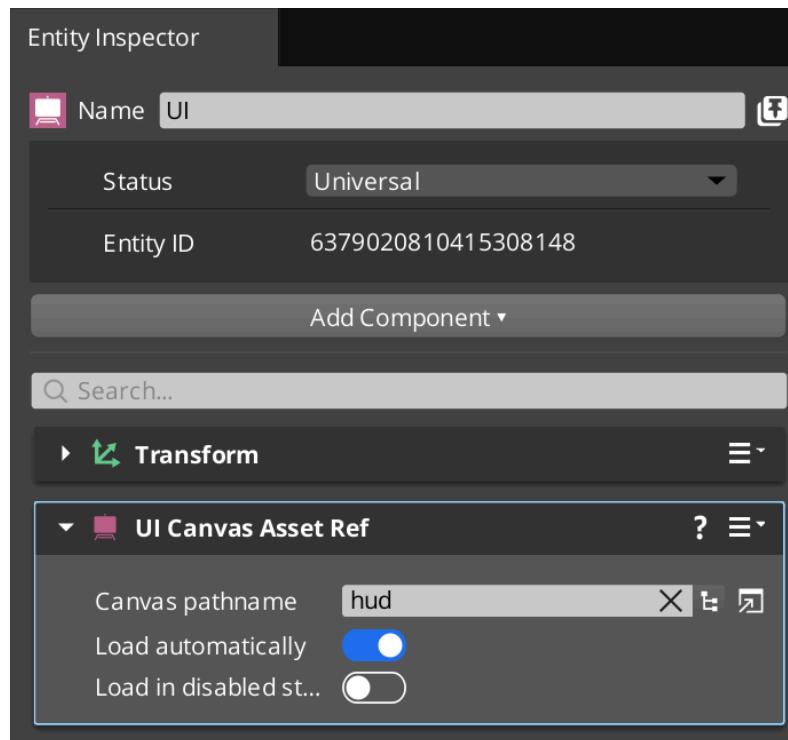


11. Save the canvas. It should look similar to this image where the panel with text appears at the top middle portion of the canvas.



## Drawing UI Canvas

In order to draw this new UI canvas over the entire screen, set **Canvas pathname** to the canvas we created.



Set hud.uicanvas

Since we have no programmatic interaction with this canvas yet, load it automatically by enabling **Load automatically** property.

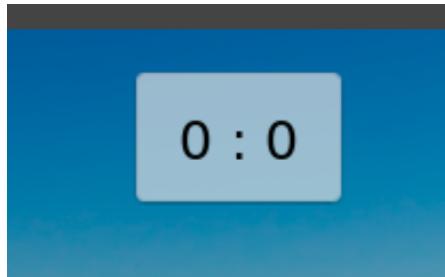
## Summary

### Note

The level and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch22\\_basic\\_ui](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch22_basic_ui)

We are now done! The canvas we created will not show up in Editor viewport until you enter game mode with **CTRL+G**. When you do, you should see score UI at the top of your screen.



UI Canvas in Game Mode

The next chapter will programmatically update this UI.

---

# Chapter 23. Interacting with UI in C++

## Note

The code and assets for this chapter can be found on GitHub at:

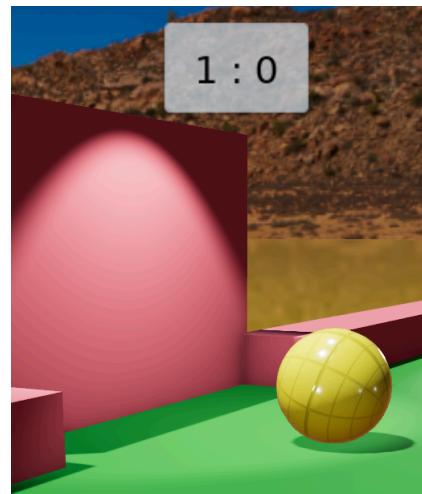
[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch23\\_ui\\_gameplay](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch23_ui_gameplay)

## Introduction

In this chapter we will build game logic to update UI scores when the ball entity hits one of the goals.

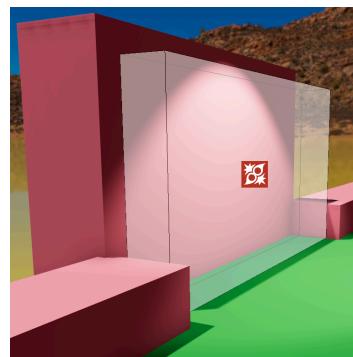
In order to achieve that we will need the following elements:

- A physical trigger at the goal lines.
- Sign up for trigger notifications from physical triggers whenever a physical body enters the trigger volume.
- Create a new User Interface component.
- Add the UI component to the canvas to update score text value.

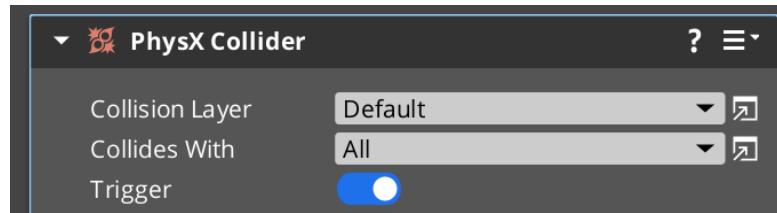


## Physical Triggers

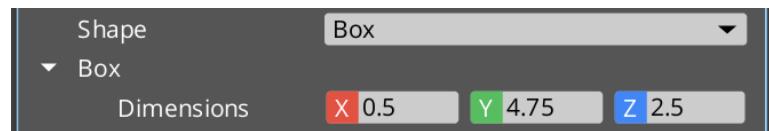
**Figure 23.1. PhysX Trigger Shape**



1. Create new entity **Goal 1** and **Goal 2**.
2. Add **PhysX Collider** component to each.
3. Enable **Trigger** switch. That turns the collider from a static collider into a trigger volume.

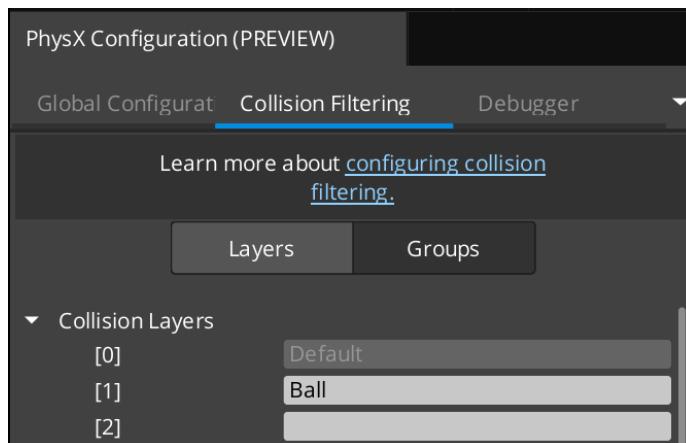


4. Choose box shape and match it to the goal space.

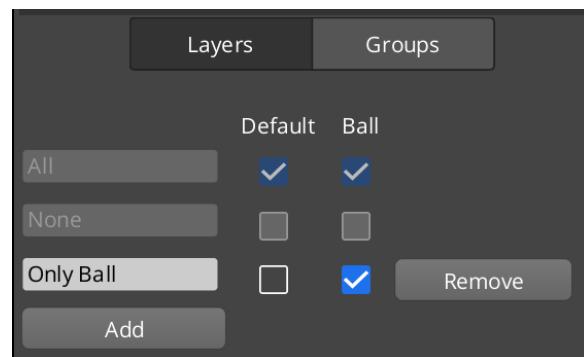


5. In order to count goals correctly our triggers should only detect the ball and no other physical objects. We will venture into collision filtering to achieve that. Click the icon to the right of **Colliders With** or from the main menu Tools → PhysX Configuration and then select Collision Filtering tab.

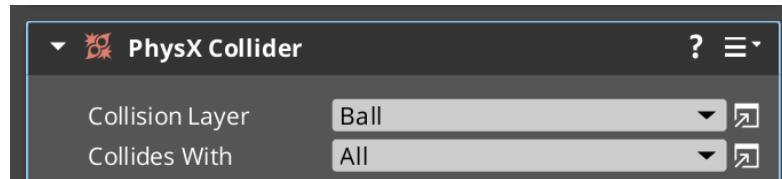
6. Under **Layers**, create *Ball* layer.



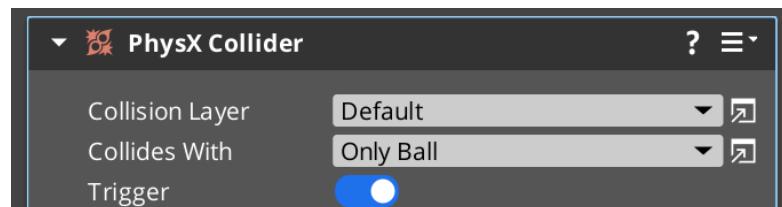
7. Under **Groups**, create *Only Ball* group with only "Ball" layer enabled.



8. Now we can use these groups and layers. Go to **Ball** entity, select its **PhysX Collider** component and set "Collision Layer" to "Ball", and "Collides With" to "All". This way the ball will collide with all objects but will be marked by a different collision layer that will give us ability to detect the ball versus other objects in the level.



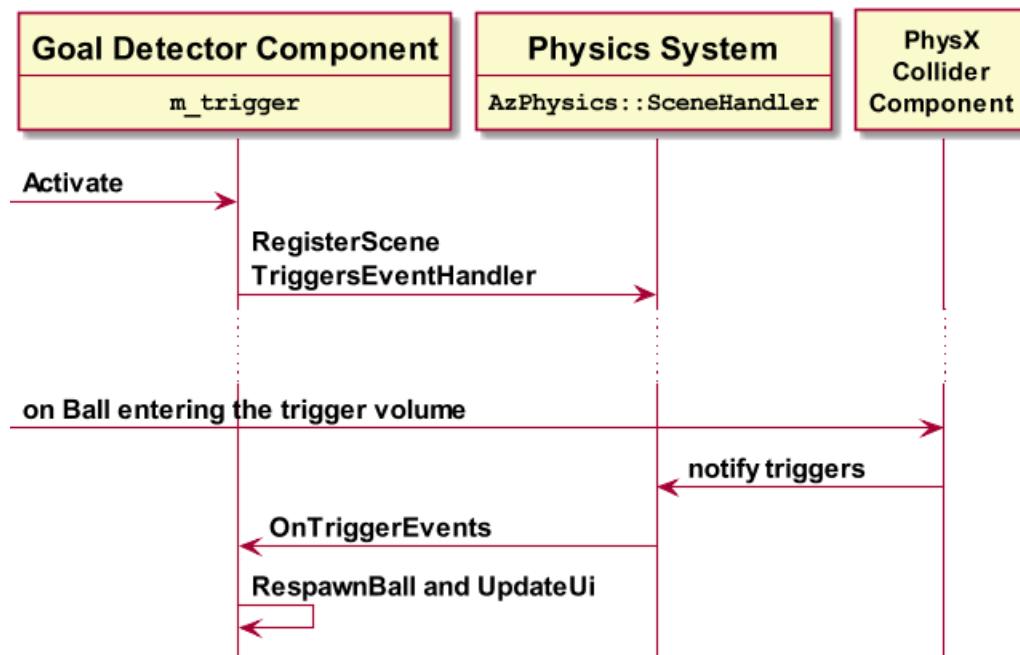
9. For **Goal 1** and **Goal 2** entities, their PhysX Collider components should have "Collision Layer" set to "Default" and "Collides With" set to "Only Ball".



## Sign up for Trigger Events

We will listen for the collision notifications in C++, using `GoalDetectorComponent`.

**Figure 23.2.** Signing up for collider volume trigger notifications



1. Create a trigger event handler.

```
AzPhysics::SceneEvents::OnSceneTriggersEvent::Handler m_trigger;
```

### Tip

We are using `AZ::Event` to sign up for notifications.

2. Assign a lambda to the handler.

```
GoalDetectorComponent::GoalDetectorComponent()
: m_trigger([this]()
    AzPhysics::SceneHandle,
    const AzPhysics::TriggerEventList& tel)
{
    OnTriggerEvents(tel);
}
{}
```

3. Register the handler with PhysX system.

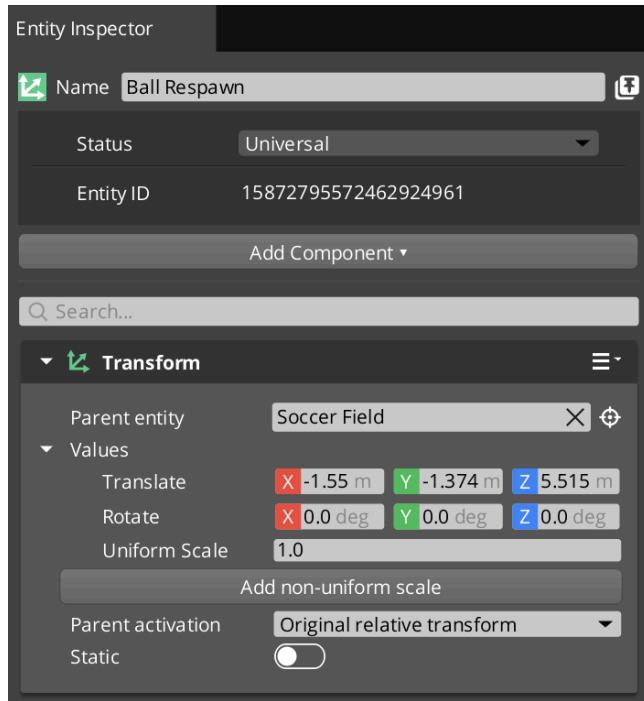
```
void GoalDetectorComponent::Activate()
{
    auto* si = AZ::Interface<AzPhysics::SceneInterface>::Get();
    if (si != nullptr)
    {
        AzPhysics::SceneHandle sh = si->GetSceneHandle(
            AzPhysics::DefaultPhysicsSceneName);
        si->RegisterSceneTriggersEventHandler(sh, m_trigger);
    }
}
```

4. Now we can process the trigger events in `OnTriggerEvents` method. It needs to do the following:

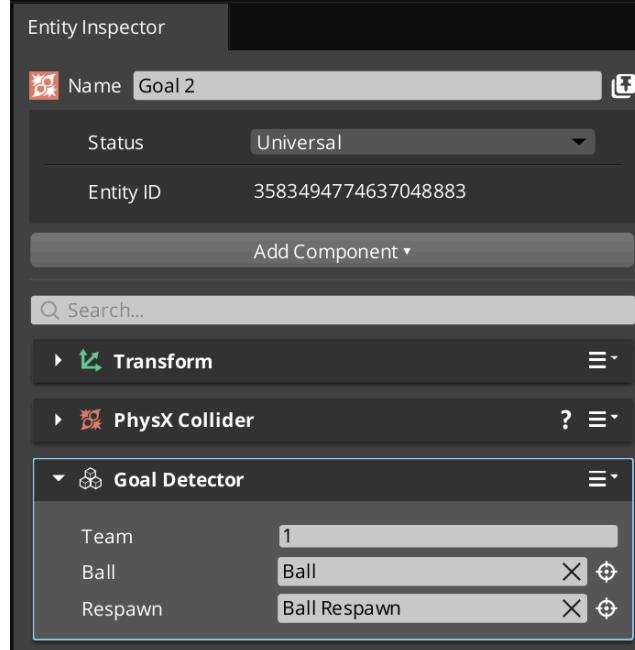
- Iterate over all occurred trigger events
- Find the trigger that matches the component we are on.
- Check that the event type was `Enter`, otherwise will also count Exit event as well and miscount the score.
- If everything checks out, then place the ball to the starting point and update the UI score.

```
void GoalDetectorComponent::OnTriggerEvents(
    const AzPhysics::TriggerEventList& tel)
{
    using namespace AzPhysics;
    for (const TriggerEvent& te : tel)
    {
        if (te.m_triggerBody &&
            te.m_triggerBody->GetEntityId() == GetEntityId())
        {
            if (te.m_type == TriggerEvent::Type::Enter)
            {
                AZ::Vector3 v = GetRespawnPoint();
                RespawnBall(v);
                UpdateUi();
            }
        }
    }
}
```

5. Respawn point is set by placing an entity at the respawn location and then passing it to `GoalDetectorComponent`. This way we can get the respawn location by querying the entity's location instead of messing around with world coordinates. In the level that we can easily adjust the respawn entity by moving it around and seeing visually in the level where the respawn point is.



6. Ball entity id is saved in the Editor on GoalDetectorComponent as well, so that Goal Detector component can move the ball back to the respawn location.



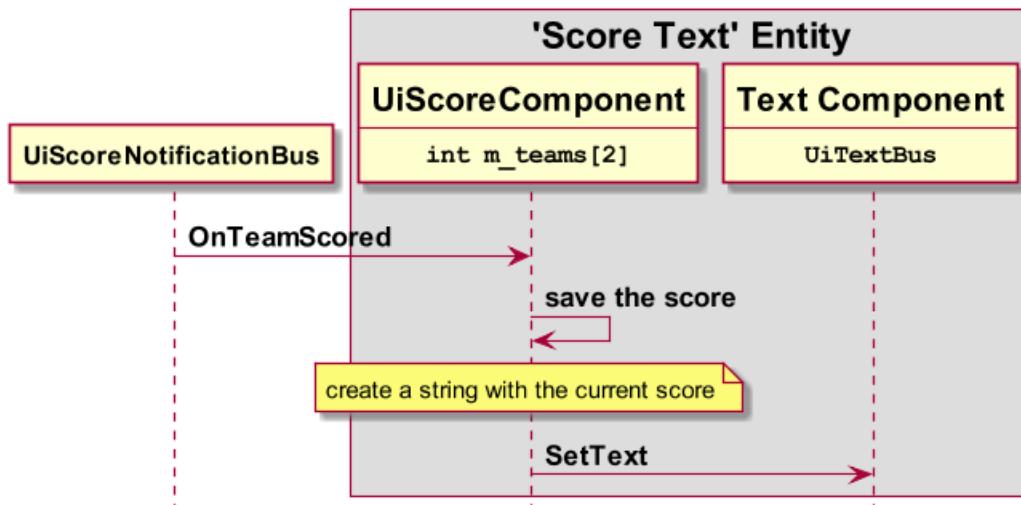
7. Team property will be used in updating UI score for the appropriate team. **Goal 1** entity has Team set to zero, while **Goal 2** entity has Team set to 1. It is just an internal identifier.
8. UpdateUI method sends a notification event using a new EBus that we will create: UiScoreNotificationBus.

```
void GoalDetectorComponent::UpdateUi()
{
    UiScoreNotificationBus::Broadcast(
        &UiScoreNotificationBus::Events::OnTeamScored, m_team);
}
```

## Receiving Updates in UI

In order to receive notifications on `UiScoreNotificationBus`, we are going to create `UiScoreComponent` that will receive them.

Figure 23.3. Updating a Text field in User Interface



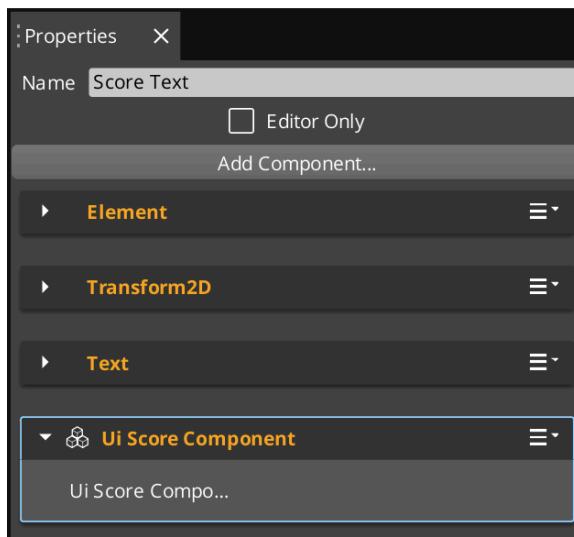
1. Create `UiScoreComponent` that inherits from the handler of the bus.

```
class UiScoreComponent
: public AZ::Component
, public UiScoreNotificationBus::Handler
```

2. Implement the notification callback that updates the text value in UI.

```
void UiScoreComponent::OnTeamScored(int team)
{
    if (team >= 0 && team <= 1)
    {
        m_teams[team]++;
        char buffer[10];
        azsnprintf(buffer, 10,
                    "%d : %d", m_teams[0], m_teams[1]);
        UiTextBus::Event(GetEntityId(),
                        &UiTextBus::Events::SetText, AZStd::string(buffer));
    }
}
```

3. Notice that it invokes `UiTextBus` on the same entity. Add `UiScoreComponent` in UI Editor to the entity **Score Text**, so that `UiScoreComponent` can communicate with `Text component`.



UiTextBus is a public interface of UI Text component.

## ⚠️ Important

A component can be added to a UI entity if it is marked as "UI" in its Reflect method.

```
Attribute(AppearsInAddComponentMenu, AZ_CRC_CE("UIT"))
```

# Moving a Rigid Body

One gotcha that may trip you up in this chapter is moving a rigid physical body. If you attempt to move it by only calling AZ::TransformBus::Event::SetWorldTranslation, you might find that the ball does not move at all. The cause is that while the translation is modified, it is overridden by the physical simulation.

One easy way to correct this is to disable physics of the ball, move it and then enable physics back on.

### Example 23.1. Moving a Rigid Body

```
void GoalDetectorComponent::RespawnBall(const AZ::Vector3& v)
{
    Physics::RigidBodyRequestBus::Event(m_ball,
        &Physics::RigidBodyRequestBus::Events::DisablePhysics);

    AZ::TransformBus::Event(m_ball,
        &AZ::TransformBus::Events::SetWorldTranslation, v);

    Physics::RigidBodyRequestBus::Event(m_ball,
        &Physics::RigidBodyRequestBus::Events::EnablePhysics);
}
```

# Summary

## Note

The code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch23\\_ui\\_gameplay](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch23_ui_gameplay)

At this point, the game updates score UI when the ball hits the goal trigger on each side of the soccer field.

### Example 23.2. GoalDetectorComponent.h

```
#pragma once
#include <AzCore/Component/Component.h>
#include <AzFramework/Physics/Common/PhysicsEvents.h>

namespace MyGem
{
    class GoalDetectorComponent
        : public AZ::Component
    {
    public:
        AZ_COMPONENT(GoalDetectorComponent,
                     "{eaf6ae0a-7444-47fb-a759-8d7b8a6f3356}");

        static void Reflect(AZ::ReflectContext* rc);

        GoalDetectorComponent();

        // AZ::Component interface implementation
        void Activate() override;
        void Deactivate() override {}

    private:
        AzPhysics::SceneEvents::
            OnSceneTriggersEvent::Handler m_trigger;
        void OnTriggerEvents(
            const AzPhysics::TriggerEventList& tel);

        AZ::EntityId m_ball;
        AZ::EntityId m_reset;
        int m_team = 0;

        AZ::Vector3 GetRespawnPoint() const;
        void RespawnBall(const AZ::Vector3& v);
        void UpdateUi();
    };
} // namespace MyGem
```

### Example 23.3. GoalDetectorComponent.cpp

```
#include <GoalDetectorComponent.h>
#include <AzCore/Component/TransformBus.h>
```

```
#include <AzCore/Interface/Interface.h>
#include <AzCore/Serialization/EditContext.h>
#include <AzFramework/Physics/PhysicsScene.h>
#include <AzFramework/Physics/RigidBodyBus.h>
#include <MyGem/UiScoreBus.h>

namespace MyGem
{
    void GoalDetectorComponent::Reflect(AZ::ReflectContext* rc)
    {
        if (auto sc = azrtti_cast<AZ::SerializeContext*>(rc))
        {
            sc->Class<GoalDetectorComponent, AZ::Component>()
                ->Field("Team", &GoalDetectorComponent::m_team)
                ->Field("Ball", &GoalDetectorComponent::m_ball)
                ->Field("Respawn", &GoalDetectorComponent::m_reset)
                ->Version(1);

            if (AZ::EditContext* ec = sc->GetEditContext())
            {
                using namespace AZ::Edit;
                ec->Class<GoalDetectorComponent>(
                    "Goal Detector",
                    "[Detects when a goal is scored]")
                    ->ClassElement(ClassElements::EditorData, "")
                    ->Attribute(
                        Attributes::AppearsInAddComponentMenu,
                        AZ_CRC_CE("Game"))
                    ->DataElement(0, &GoalDetectorComponent::m_team,
                        "Team", "Which team is this goal line for?")
                    ->DataElement(0, &GoalDetectorComponent::m_ball,
                        "Ball", "Ball entity")
                    ->DataElement(0, &GoalDetectorComponent::m_reset,
                        "Respawn", "where to put the ball after")
                ;
            }
        }
    }

    GoalDetectorComponent::GoalDetectorComponent()
        : m_trigger([this]{
            AzPhysics::SceneHandle,
            const AzPhysics::TriggerEventList& tel)
    {
        OnTriggerEvents(tel);
    })
    {

    }

    void GoalDetectorComponent::Activate()
    {
        auto* si = AZ::Interface<AzPhysics::SceneInterface>::Get();
        if (si != nullptr)
        {

```

```
AzPhysics::SceneHandle sh = si->GetSceneHandle(
    AzPhysics::DefaultPhysicsSceneName);
si->RegisterSceneTriggersEventHandler(sh, m_trigger);
}

void GoalDetectorComponent::OnTriggerEvents(
    const AzPhysics::TriggerEventList& tel)
{
    using namespace AzPhysics;
    for (const TriggerEvent& te : tel)
    {
        if (te.m_triggerBody &&
            te.m_triggerBody->GetEntityId() == GetEntityId())
        {
            if (te.m_type == TriggerEvent::Type::Enter)
            {
                AZ::Vector3 respawnLocation = GetRespawnPoint();
                RespawnBall(respawnLocation);
                UpdateUi();
            }
        }
    }
}

AZ::Vector3 GoalDetectorComponent::GetRespawnPoint() const
{
    AZ::Vector3 respawnLocation = AZ::Vector3::CreateZero();
    AZ::TransformBus::EventResult(respawnLocation, m_reset,
        &AZ::TransformBus::Events::GetWorldTranslation);
    return respawnLocation;
}

void GoalDetectorComponent::RespawnBall(const AZ::Vector3& v)
{
    Physics::RigidBodyRequestBus::Event(m_ball,
        &Physics::RigidBodyRequestBus::Events::DisablePhysics);
    AZ::TransformBus::Event(m_ball,
        &AZ::TransformBus::Events::SetWorldTranslation, v);
    Physics::RigidBodyRequestBus::Event(m_ball,
        &Physics::RigidBodyRequestBus::Events::EnablePhysics);
}

void GoalDetectorComponent::UpdateUi()
{
    UiScoreNotificationBus::Broadcast(
        &UiScoreNotificationBus::Events::OnTeamScored, m_team);
}
} // namespace MyGem
```

#### Example 23.4. **UiScoreComponent.h**

```
#pragma once
#include <AzCore/Component/Component.h>
```

```
#include <MyGem/UiScoreBus.h>

namespace MyGem
{
    class UiScoreComponent
        : public AZ::Component
        , public UiScoreNotificationBus::Handler
    {
public:
    AZ_COMPONENT(UiScoreComponent,
        "{49b2e5e8-e028-48b1-bc69-82c73b32422b}");

    static void Reflect(AZ::ReflectContext* rc);

    // AZ::Component interface implementation
    void Activate() override;
    void Deactivate() override;

    // UiScoreNotificationBus interface
    void OnTeamScored(int team) override;

private:
    int m_teams[2] = { 0, 0 };
};

} // namespace MyGem
```

### Example 23.5. UiScoreComponent.cpp

```
#include <UiScoreComponent.h>
#include <AzCore/Serialization/EditContext.h>
#include <LyShine/Bus/UiTextBus.h>

namespace MyGem
{
    void UiScoreComponent::Reflect(AZ::ReflectContext* rc)
    {
        if (auto sc = azrtti_cast<AZ::SerializeContext*>(rc))
        {
            sc->Class<UiScoreComponent, AZ::Component>()
                ->Version(1);

            if (AZ::EditContext* ec = sc->GetEditContext())
            {
                using namespace AZ::Edit;
                ec->Class<UiScoreComponent>(
                    "Ui Score Component",
                    "[Updates score text]")
                    ->ClassElement(ClassElements::EditorData, "")
                    ->Attribute(
                        Attributes::AppearsInAddComponentMenu,
                        AZ_CRC_CE("UI"));
            }
        }
    }
}
```

```
void UiScoreComponent::Activate()
{
    UiScoreNotificationBus::Handler::BusConnect(GetEntityId());
}

void UiScoreComponent::Deactivate()
{
    UiScoreNotificationBus::Handler::BusDisconnect();
}

void UiScoreComponent::OnTeamScored(int team)
{
    if (team >= 0 && team <= 1)
    {
        m_teams[team]++;
        char buffer[10];
        azsnprintf(buffer, 10,
                    "%d : %d", m_teams[0], m_teams[1]);
        UiTextBus::Event(GetEntityId(),
                         &UiTextBus::Events::SetText, AZStd::string(buffer));
    }
}
} // namespace MyGem
```

# Chapter 24. Kicking the Ball

## Introduction

The next glaring gameplay issue for me is that the chicken does not push the ball all that well. Sometimes it does and sometimes the ball ignores the chicken entirely.

This chapter will fix this issue by doing the following:

- Add a physical trigger on the chicken.
- Detect when a chicken collides with the ball using the trigger.
- Calculate the direction of the impulse from the chicken towards the ball.
- Apply the impulse to the ball to "kick" it.



### Note

The code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch24\\_kicking\\_ball](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch24_kicking_ball)

## The Root Cause of the Issue

The reason the character does not push the ball well, if at all, is actually a design decision by the physical simulation. O3DE uses PhysX as the simulation engine behind collider and rigid body components you have seen so far. Here is a quote from PhysX SDK page on the interaction between characters and dynamic actors.

It is tempting to let the physics engine push dynamic objects by applying forces at contact points. However it is often not a very convincing solution. The bounding volumes around characters are artificial (boxes, capsules, etc) and invisible, so the forces computed by the physics engine between a bounding volume and its surrounding objects will not be realistic anyway. They will not properly model the interaction between an actual character and these objects.

—PhysX 4.1 SDK Guide, Character Controllers

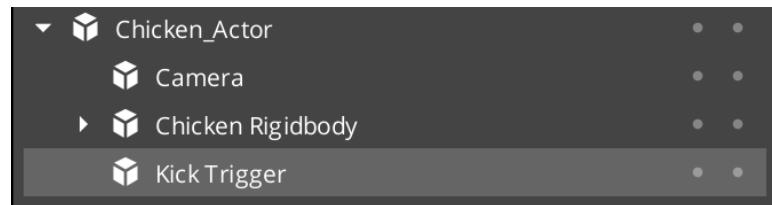
Long story short, it is up to us implement the interaction between the chicken character and the ball. That is not a problem. We can use the knowledge from our previous chapters to solve it.

## Trigger Shape on the Chicken

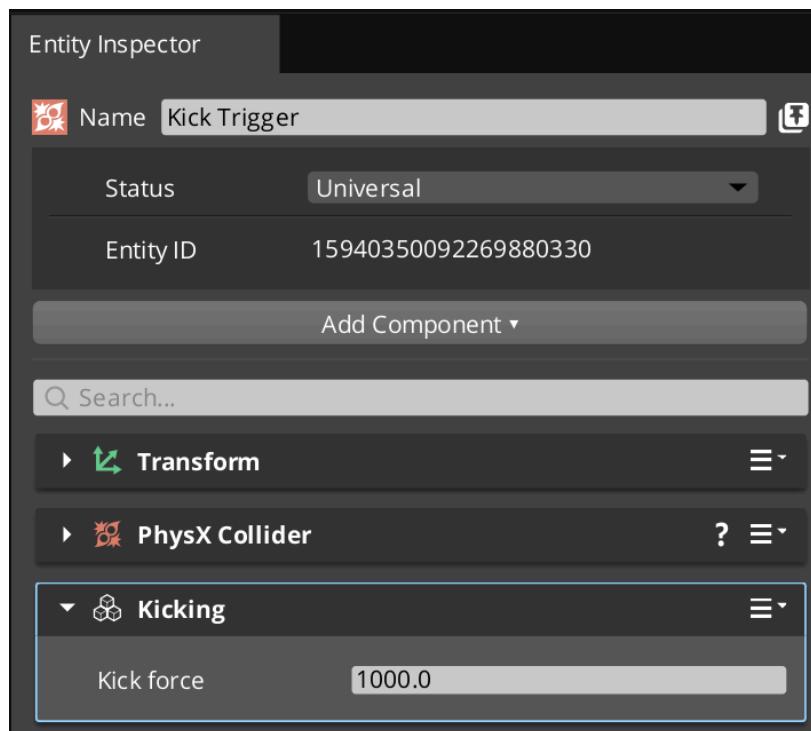
Here are the steps to add a trigger shape on the chicken.

1. Create a child entity under **Chicken\_Actor**.

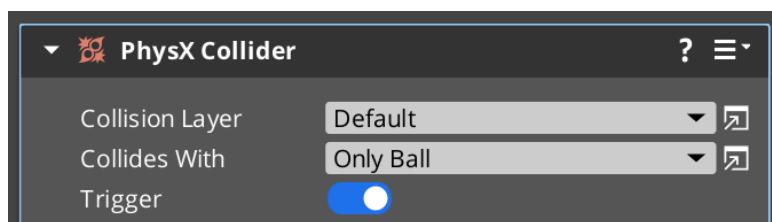
- Call the new entity **Kick Trigger**.



- Add **PhysX Collider** component to the new entity.
- We will also create a new component, **KickingComponent**, and add it to this entity.



- In Chapter 23, *Interacting with UI in C++*, we created a new collision layer and a group. We can filter collisions with the kicking trigger volume to the ball only. Set the **Collision Layer** to "Default".
- Set **Collides With** to "Only Ball".
- Turn on Trigger property on the collider.



- Set the shape of the collider to a sphere. You may need to modify size and offset of the sphere to match the chicken body.



The steps above should create the following trigger shape on the chicken.

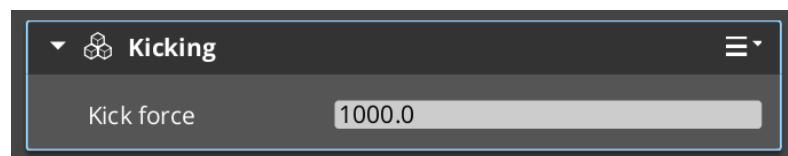
**Figure 24.1. Chicken with a Trigger Shape**



## Kicking Component

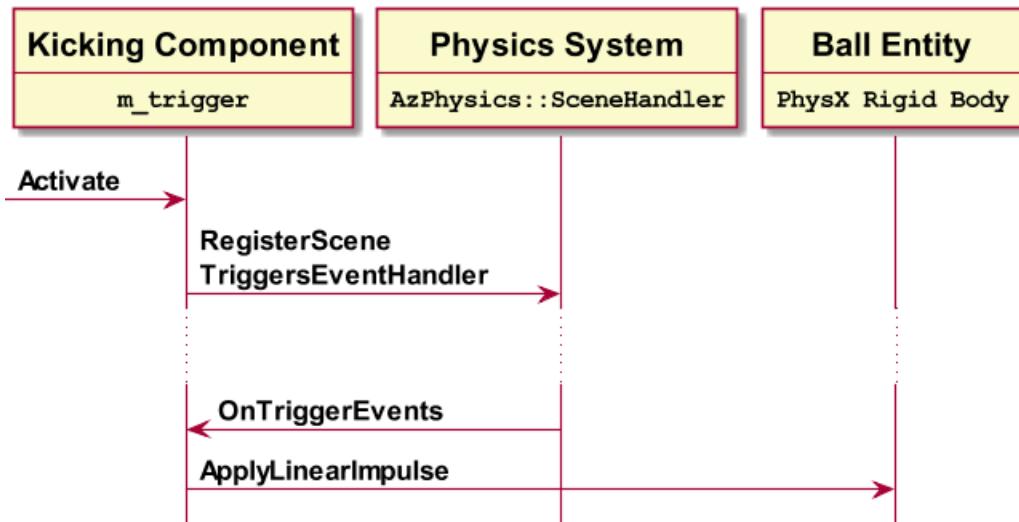
The trigger shape will detect when a ball is right next to the chicken. We need to process trigger events and apply impulse to the ball.

**Figure 24.2. KickingComponent**



Kicking component will have a configurable amount of impulse to apply to the ball. The logic of signing up for collision notifications is very similar to GoalDetectorComponent from Chapter 23, *Interacting with UI in C++*.

**Figure 24.3. Design of Kicking Component**



1. Create a trigger event handler.

```
AzPhysics::SceneEvents::OnSceneTriggersEvent::Handler m_trigger;
```

2. Assign a lambda to pass handling of the event to a member function.

```

KickingComponent::KickingComponent()
: m_trigger([this])
    AzPhysics::SceneHandle,
    const AzPhysics::TriggerEventList& tel)
{
    OnTriggerEvents(tel);
}
{
}
```

3. Register the event handler with PhysX sub-system.

```

void KickingComponent::Activate()
{
    auto* si = AZ::Interface<AzPhysics::SceneInterface>::Get();
    if (si != nullptr)
    {
        AzPhysics::SceneHandle sh = si->GetSceneHandle(
            AzPhysics::DefaultPhysicsSceneName);
        si->RegisterSceneTriggersEventHandler(sh, m_trigger);
    }
}
```

4. When the trigger event occurs, check that the ball entered the trigger volume and if so, kick it.

```
void KickingComponent::OnTriggerEvents(
```

```
const AzPhysics::TriggerEventList& tel)
{
    using namespace AzPhysics;
    for (const TriggerEvent& te : tel)
    {
        if (te.m_triggerBody &&
            te.m_triggerBody->GetEntityId() == GetEntityId())
        {
            if (te.m_type == TriggerEvent::Type::Enter)
            {
                KickBall(te.m_otherBody->GetEntityId());
            }
        }
    }
}
```

5. Kicking the ball is done by calculating the direction from the chicken to the ball, creating a vector along that direction, and using `ApplyLinearImpulse` on the ball's rigid body component.

```
void KickingComponent::KickBall(AZ::EntityId b)
{
    AZ::Vector3 impulse = GetBallPosition(b) - GetSelfPosition();

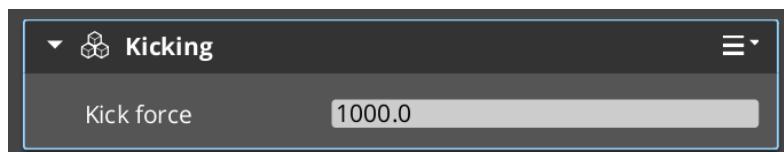
    impulse.Normalize();
    impulse *= m_kickForce;

    AddImpulseToBall(impulse, b);
}

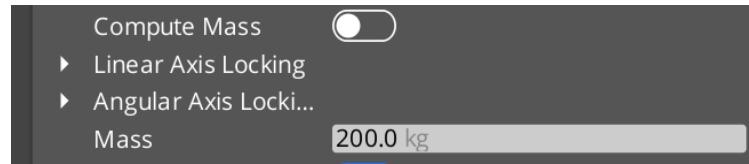
void KickingComponent::AddImpulseToBall(
    AZ::Vector3 v, AZ::EntityId ball)
{
    Physics::RigidBodyRequestBus::Event(ball,
        &Physics::RigidBodyRequestBus::Events::ApplyLinearImpulse,
        v);
}
```

## Mass versus Impulse

You will have to play around with how much impulse to apply to the ball versus how heavy the ball is. The impulse can be modified on our `KickingComponent`.



The mass of the ball can be modified on **Physx Rigid Body** component by turning off automatic mass calculation by disabling **Compute Mass** and specifying **Mass** value manually.



I found the ball mass of two hundred (200) and impulse value of a thousand (1000) works well for our game.

## Source Code

### Note

The code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch24\\_kicking\\_ball](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch24_kicking_ball)

In this chapter, we added one new component, `KickingComponent`, to have the chicken kick the ball in a consistent way that is suitable for the gameplay.

#### Example 24.1. `KickingComponent.h`

```
#pragma once
#include <AzCore/Component/Component.h>
#include <AzFramework/Physics/Common/PhysicsEvents.h>

namespace MyGem
{
    class KickingComponent
        : public AZ::Component
    {
public:
    AZ_COMPONENT(KickingComponent,
                  "{73A60188-9BFB-4168-A733-8A06BC500ECB}");

    static void Reflect(AZ::ReflectContext* rc);

    KickingComponent();

    // AZ::Component interface implementation
    void Activate() override;
    void Deactivate() override {}

private:
    AzPhysics::SceneEvents::
        OnSceneTriggersEvent::Handler m_trigger;
    void OnTriggerEvents(
        const AzPhysics::TriggerEventList& tel);

    float m_kickForce = 1000.f;

    void KickBall(AZ::EntityId ball);

    AZ::Vector3 GetBallPosition(AZ::EntityId ball);
}
```

```
    AZ::Vector3 GetSelfPosition();
    void AddImpulseToBall(AZ::Vector3 v, AZ::EntityId ball);
}
} // namespace MyGem
```

### Example 24.2. KickingComponent.cpp

```
#include <KickingComponent.h>
#include <AzCore/Component/Entity.h>
#include <AzCore/Component/TransformBus.h>
#include <AzCore/Interface/Interface.h>
#include <AzCore/Serialization/EditContext.h>
#include <AzFramework/Physics/PhysicsScene.h>
#include <AzFramework/Physics/RigidBodyBus.h>

namespace MyGem
{
    void KickingComponent::Reflect(AZ::ReflectContext* rc)
    {
        if (auto sc = azrtti_cast<AZ::SerializeContext*>(rc))
        {
            sc->Class<KickingComponent, AZ::Component>()
                ->Field("Kick force", &KickingComponent::m_kickForce)
                ->Version(1);

            if (AZ::EditContext* ec = sc->GetEditContext())
            {
                using namespace AZ::Edit;
                ec->Class<KickingComponent>(
                    "Kicking",
                    "[Kicking the ball when it comes close]")
                    ->ClassElement(ClassElements::EditorData, "")
                    ->Attribute(
                        Attributes::AppearsInAddComponentMenu,
                        AZ_CRC_CE("Game"))
                    ->DataElement(0, &KickingComponent::m_kickForce,
                        "Kick force", "impulse strength");
            }
        }
    }

    KickingComponent::KickingComponent()
        : m_trigger([this](
            AzPhysics::SceneHandle,
            const AzPhysics::TriggerEventList& tel)
    {
        OnTriggerEvents(tel);
    }) {}

    void KickingComponent::Activate()
    {
        auto* si = AZ::Interface<AzPhysics::SceneInterface>::Get();
        if (si != nullptr)
        {
```

```
AzPhysics::SceneHandle sh = si->GetSceneHandle(
    AzPhysics::DefaultPhysicsSceneName);
si->RegisterSceneTriggersEventHandler(sh, m_trigger);
}

void KickingComponent::OnTriggerEvents(
    const AzPhysics::TriggerEventList& tel)
{
    using namespace AzPhysics;
    for (const TriggerEvent& te : tel)
    {
        if (te.m_triggerBody &&
            te.m_triggerBody->GetEntityId() == GetEntityId())
        {
            if (te.m_type == TriggerEvent::Type::Enter)
            {
                KickBall(te.m_otherBody->GetEntityId());
            }
        }
    }
}

void KickingComponent::KickBall(AZ::EntityId b)
{
    AZ::Vector3 impulse = GetBallPosition(b) - GetSelfPosition();
    impulse.Normalize();
    impulse *= m_kickForce;
    AddImpulseToBall(impulse, b);
}

AZ::Vector3 KickingComponent::GetBallPosition(AZ::EntityId ball)
{
    AZ::Vector3 ballPosition = AZ::Vector3::CreateZero();
    AZ::TransformBus::EventResult(ballPosition, ball,
        &AZ::TransformBus::Events::GetWorldTranslation);
    return ballPosition;
}

AZ::Vector3 KickingComponent::GetSelfPosition()
{
    return GetEntity()->GetTransform()->
        GetWorldTM().GetTranslation();
}

void KickingComponent::AddImpulseToBall(
    AZ::Vector3 v, AZ::EntityId ball)
{
    Physics::RigidBodyRequestBus::Event(ball,
        &Physics::RigidBodyRequestBus::Events::ApplyLinearImpulse,
        v);
}
} // namespace MyGem
```

# Chapter 25. Introduction to Animation

## Introduction to EMotionFX

At this point in our game development, the chicken is stuck in idle animation. The next improvement we are going to make is to play walk animation when the chicken is moving and play idle animation when it is not moving.

O3DE comes with an animation gem, EMotionFX. In this chapter, I am going to show you how to build a state machine using EMotionFX to blend between idle and walk animations.

### Note

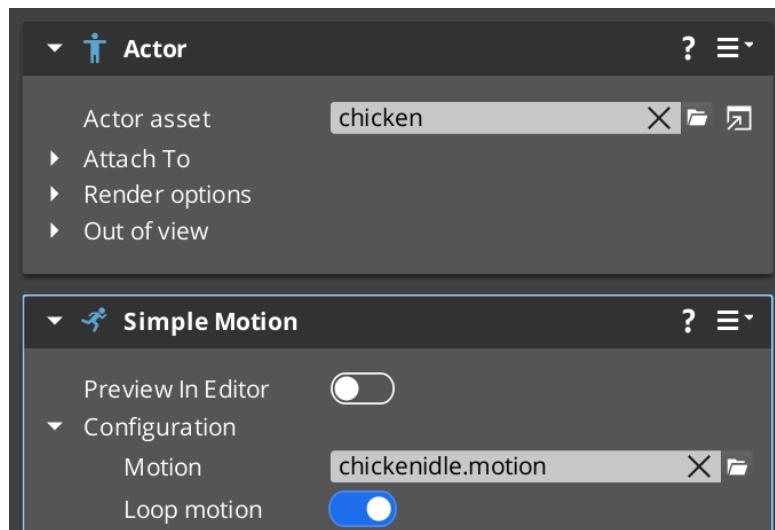
The level and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch25\\_animation](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch25_animation)

## Simple Motion Component

If you do not need to blend animations or other complex animation interaction, you can use **Simple Motion** component that can play one animation.

### Example 25.1. Simple Motion component



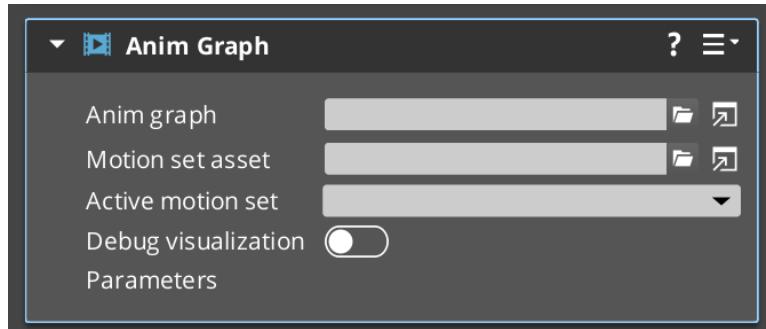
This component has various configuration options but we are going to remove it from **Chicken\_Actor** entity and replace it with **Anim Graph** component.

### Important

Both Simple Motion and Animation Graph components require an **Actor** component. **Chicken\_Actor** entity already has one assigned from C:\O3DE\21.11.2\Gems\NvCloth\Assets\Objects\cloth\Chicken\Actor\chicken.fbx. Actor components provide actor mesh and bone information for animations.

# Anim Graph Component

Animation Graph component will allow us to assign an animation graph to the actor and allow us to control the animation using parameters of our choice.



When you first add this component, its fields are empty. First, we have to create an **Anim graph** asset. Open Animation Editor either through the main menu **Tools** → **Animation Editor** or by clicking on the right most icon next to **Anim graph** property on the component.

## Building an Animation Graph

### Tip

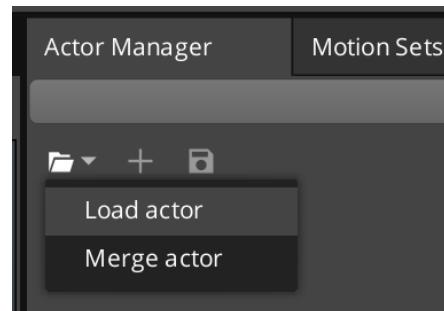
Animation Editor should open with an empty workspace. If not, reset it with **File** → **New Workspace**.

An animation graph is a collection of various nodes that define the logic and behavior of animations and their interactions, such as blending multiple animations together. At a high level, we are going to build a state machine that blends between idle motion from `chickenidle.fbx` and walking motion from `chickenwalking.fbx`. Both files come with NvCloth gem in O3DE installation at `C:\O3DE\21.11.2\Gems\NvCloth\Assets\Objects\cloth\Chicken\Motions`.

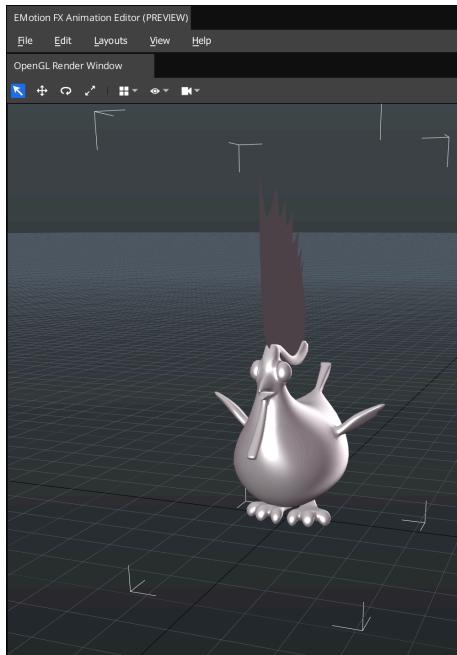
## Assign Preview Actor

To give us a visual guidance as we work on an animation graph, load an actor into Animation Editor.

1. Go to **Actor Manager** tab. (If it is not open, use the main menu: **View** → **Actor Manager**.)
2. Click on the folder icon.
3. Choose **Load actor**.



- Choose `chicken.actor` from NvCloth gem's location. That will load the chicken actor into **OpenGL Render Window**. It will serve as a testing ground for the animation graph.



## Test Motions

Before we spend time building out the graph, test that the motion works with chicken.

- Go to Motions tab, **View → Motions**.
- Click on plus "+" icon.
- Pick **Emotion FX Motion** dialog will open, select `chickenidle.motion`.

Actor Manager		Motion Sets	Motions	Anim Graph
+	✖	Search...		
Name		Duration	Joints	Morphs
chickenidle		5.67 ...	35	0

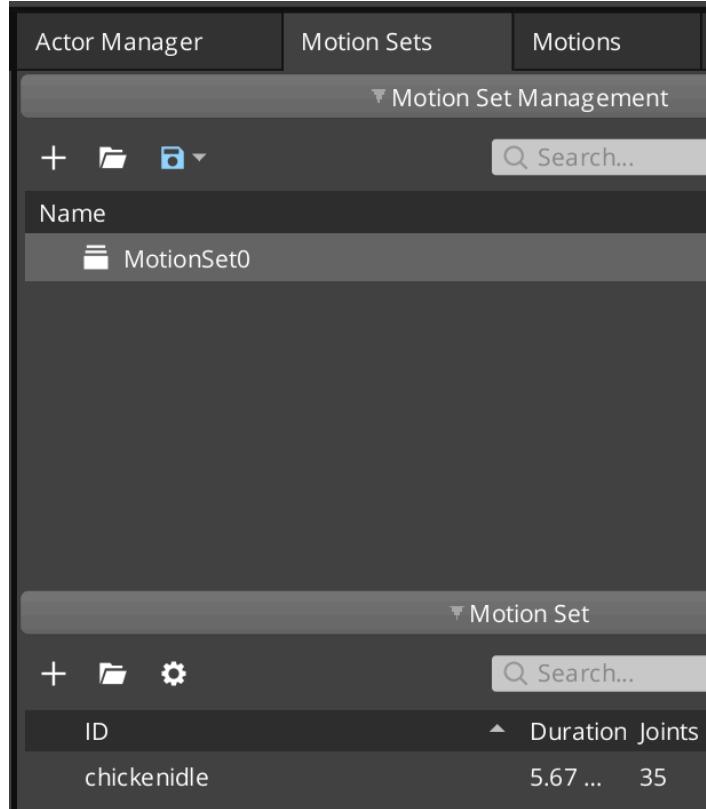
- In the list of motions, double click on **chickenidle** to test the motion.
- Repeat the steps for `chickenwalking.motion`.

That will play the animations in OpenGL Render Window to let you see how the animation look and if they match the actor.

## Create a Motion Set

Before we get into building the animation graph, we need to create a motion set to hold the collection of the motions we are going to use.

1. Go to Motion Sets tab, if it is not open use View → Motions Sets.
2. Click on plus "+" icon to create a new motion set, **MotionSet0**.
3. In **Motion Set** section, there is folder icon, click it and select `chickenidle.motion`



### Tip

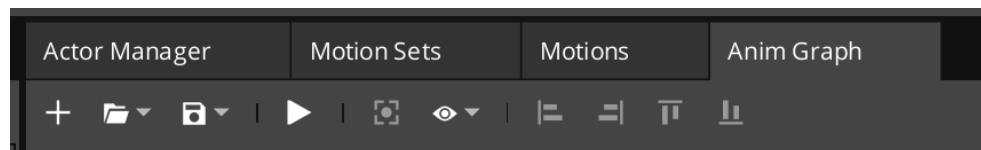
You can double click the motion **chickenidle** to test it in the render window.

4. Repeat the steps for `chickenwalking.motion`.
5. Save the motion set using floppy disk icon. For example, I saved it to `C:\git\book\Gems\MyGem\Assets\Animation\chicken.motionset`.

## Anim Graph

We are ready to start building an animation graph that will blend walking and idle motions.

1. Go to **Anim Graph** tab, if it is not open use View → Anim Graph.

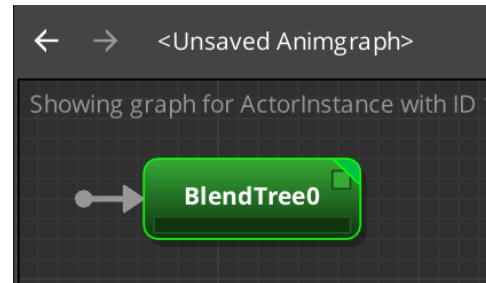


2. Click on plus "+" icon to create a new empty Animgraph.

- In empty work space of Anim Graph, right click and choose Create Node → Sources → BlendTree.

 **Tip**

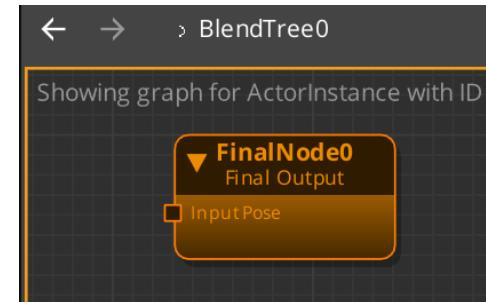
You can also drag and drop BlendTree node from **Anim Graph Palette**'s Sources group.



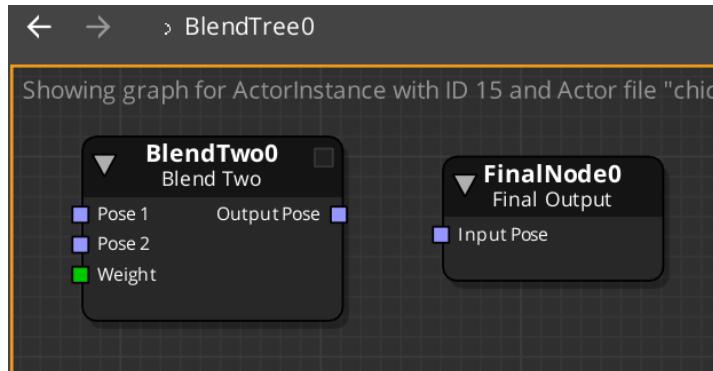
- Double click on the **BlendTree0** node to see its configuration. By default, a blend tree has a final node and nothing else. This node will receive the blend of two motions.

 **Tip**

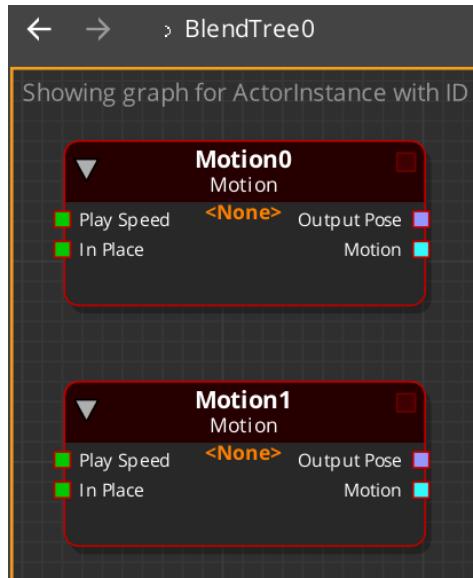
You can rename node names in **Attributes** panel.



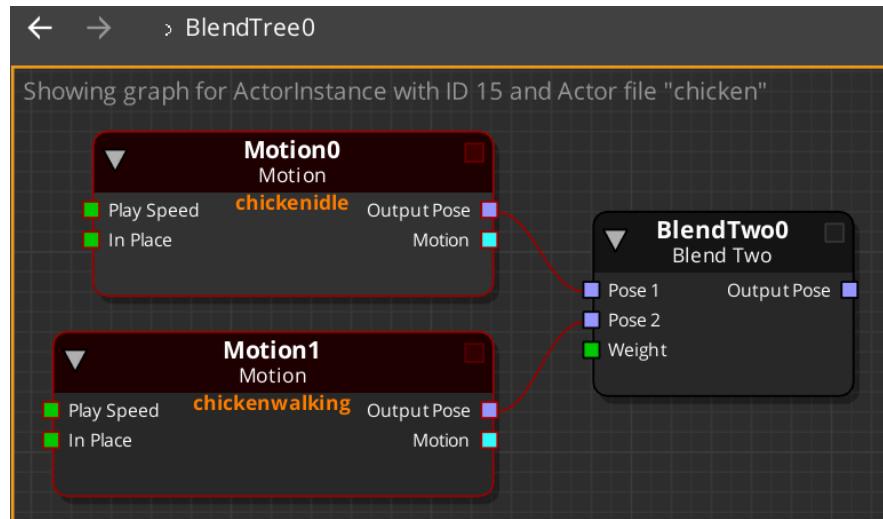
- Add Blend Two node to the left of **FinalNode0** with right click → Create Node → Blending → **BlendTwo**. This node will blend two motions together and assign the result into **FinalNode0**.



- To the left of **BlendTwo0**, create two Motion nodes, with Create Node → Sources → Motion.
- Select Motion0 node.
- Go to Attributes panel.
- To the right of Motions property, click on plus "+" button.
- Motion Selection Window will open, select **chickenidle** motion from MotionSet0.
- For Motion1 node, select **chickenwalking** motion in the same manner.



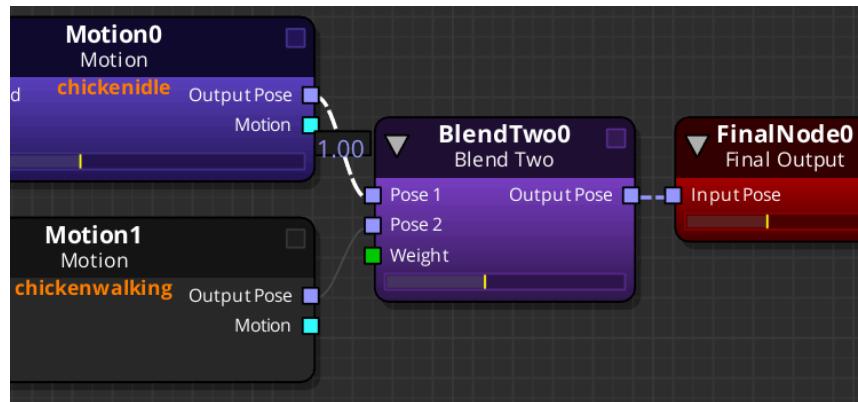
12. On Motion nodes, there is **Output Pose** output. Drag a line from the square next to Output Pose to **Pose 1** input on **BlendTwo0** node.



13. Connect Outpost Pose of BlendTwo0 node with FinalNode0's Input Pose. The animation graph should start to play. If not, there is a white arrow play button in Anim Graph's toolbar.

# Animation Blending

**Figure 25.1. First Blending of Two Animations**

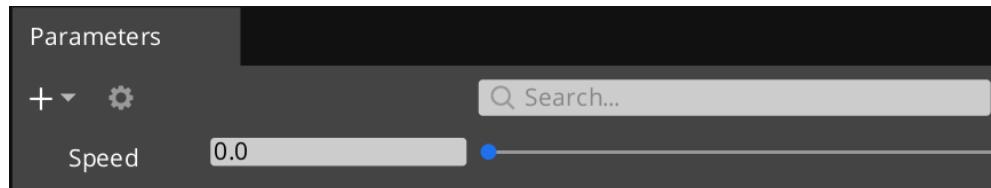


Let us take a look at what is going on. We created a blend node. Inside the node, we declared two motion sources that are blended together with Blend Two node, which outputs the result into final node. At the moment, the only motion that is actually playing is the idle motion. This is because **Weight** input of BlendTwo0 node determines the amount of blending between the two motions.

Since we are blending between idle and walking motions, it would make sense to blend it based on the speed of the chicken. If the speed is zero, it should play idle motion. As the chicken starts to move the anim graph should blend between the two motions until playing only the walking motion.

## Parameters

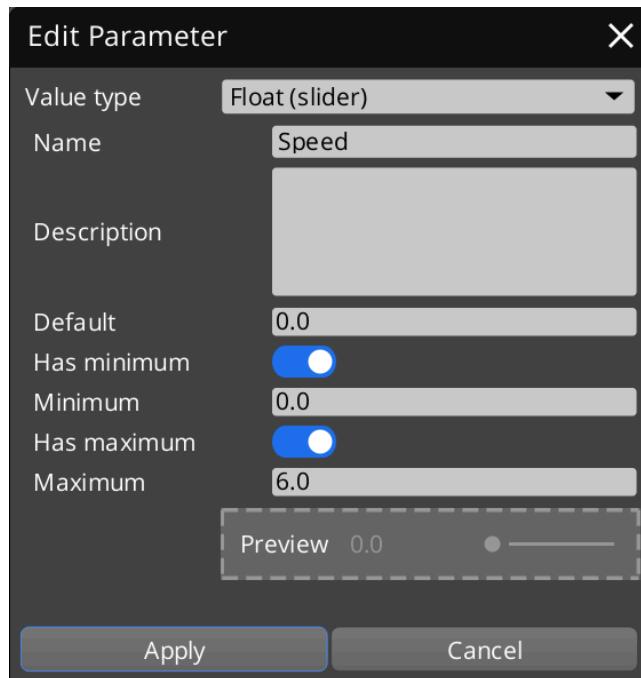
To control the blending, we will add a parameter that we can specify from outside of the animation graph.



There is a C++ API to do so from a component but before we get to that, we will create a parameter and test it inside the Animation Editor.

1. Go to Parameters window, using View → Parameter Window if it is not open.
2. Click on plus "+" icon.
3. Choose **Add parameter**.
4. For **Name**, enter "Speed".
5. For **Maximum**, enter six (6.0), since the maximum speed of the chicken was set to 6 in Chapter 18, *Character Movement*.

6. Click "Create" to create the parameter.



Speed parameter

7. In Anim graph space, choose Create Node → Sources → Parameters.
8. With Parameters0 node selected, go to Attributes panel and click on **Select parameters**.
9. Choose Speed parameter.

### Note

Our speed parameter range was set to be between zero and six. But the Weight input value of BlendTwo0 takes a range from zero to one. We have to remap the range to match Weight requirements using Range Remapper node.

10. Create Node → Math → Smoothing.

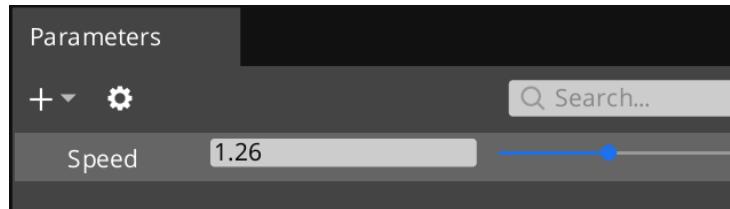
### Tip

Smoothing node is optional but it lets us smoothly raise the value of a parameter and thus blend between the motions over time without doing this work in a C++ component. However, if you need precise control over the animation output, you will have to drive the Speed parameter yourself.

11. Create Node → Math → Range Remapper.
12. With RangeRemapper0 node selected, go to Attributes panel and change Input Max to six (6) to match the range of Speed parameter.
13. Connect Parameters0 to Smoothing0, Smoothing0 to RangeRemapper0 and RangeRemapper0 to Weight input of BlendTwo0.



14. Now you can test the animation graph by changing Speed parameter value.

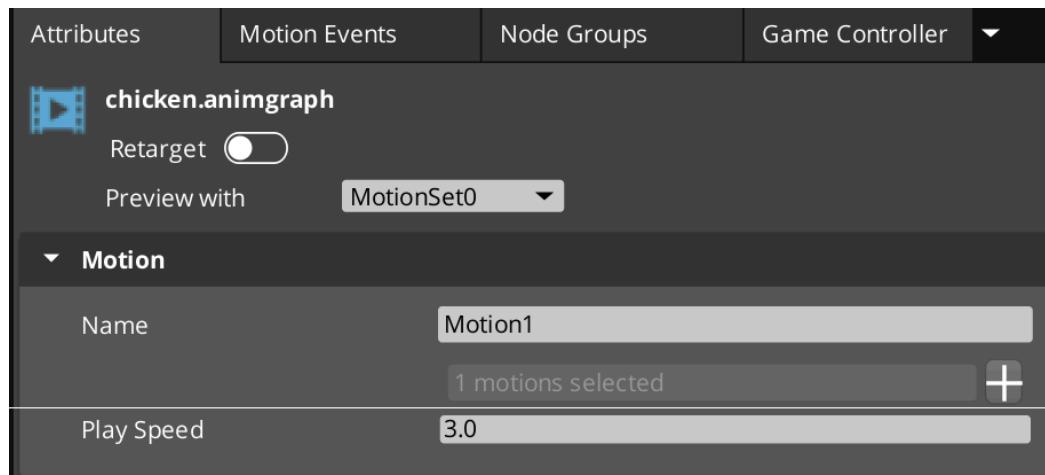


15. Save the anim graph as C:\git\book\Gems\MyGem\Assets\Animation\chicken.animgraph.
16. Save the workspace as C:\git\book\Gems\MyGem\Assets\Animation\chicken.emxworkspace.

## Motion Adjustments

We have a walking animation for the chicken but it is too slow for our gameplay. We can fix that up speeding up the play speed of the walking animation.

1. Go to **Motion1** node that has chickenwalking.motion assigned.
2. In Attributes panel, change Play Speed to three (3).
3. Test the look of the animation by modifying Speed parameter.

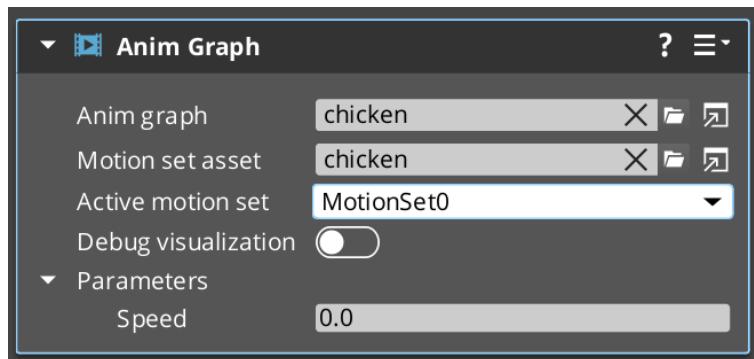


4. Save the animation graph.

# Assigning the Anim Graph

At this point, we are done with Animation Editor, going back to Editor, we can assign the new graph in Anim Graph component.

1. Set Anim graph to chicken.animgraph.
2. Set Motion set asset to chicken.motionset.
3. "Action motion set" property will update to MotinoSet0 on its own after you set the motion set.



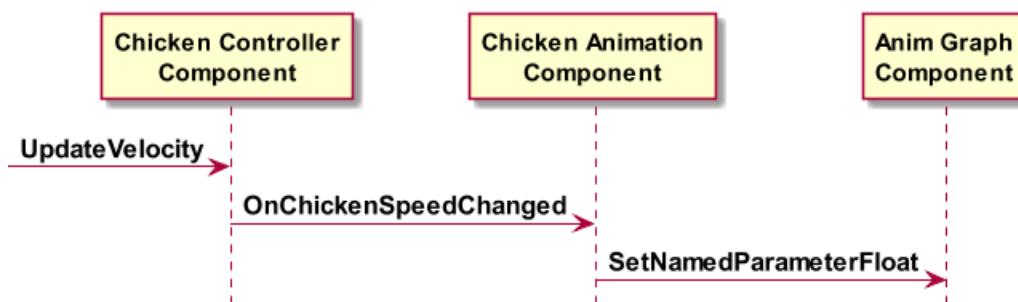
## Note

**Anim Graph** component reads the animation graph we assigned, finds Speed parameter and reflects it to the Editor. You can test the animation again by manually entering the value on the component and playing the level with **CTRL+G**.

# Driving Speed Parameter

The only work left for us is to set Speed parameter based on the chicken's speed. This can be done by calculating the speed of our chicken and then calling `SetNamedParameterFloat`.

```
using namespace EMotionFX::Integration;
AnimGraphComponentRequestBus::Event(m_actor,
    &AnimGraphComponentRequests::SetNamedParameterFloat,
    "Speed", s);
```



In order to separate concerns from other components we have written so far, I have created a new component, `ChickenAnimationComponent` that receives speed notifications via `ChickenNotificationBus` from `ChickenControllerComponent`.

```
void ChickenControllerComponent::UpdateVelocity(
    const ChickenInput& input)
{
    ...
    ChickenNotificationBus::Event(GetEntityId(),
        &ChickenNotificationBus::Events::OnChickenSpeedChanged,
        m_velocity.GetLength());
}
```

OnChickenSpeedChanged will assign the Speed parameter and finish the work for this chapter.

```
void ChickenAnimationComponent::OnChickenSpeedChanged(float s)
{
    using namespace EMotionFX::Integration;
    AnimGraphComponentRequestBus::Event(m_actor,
        &AnimGraphComponentRequests::SetNamedParameterFloat,
        "Speed", s);
}
```

### Example 25.2. Chicken Running at the Ball



## Source Code

### Note

The level and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch25\\_animation](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch25_animation)

**Example 25.3. ChickenBus.h**

```
#pragma once
#include <AzCore/Component/ComponentBus.h>

namespace MyGem
{
    class ChickenNotifications
        : public AZ::ComponentBus
    {
    public:
        virtual void OnChickenSpeedChanged(float speed) = 0;
    };

    using ChickenNotificationBus = AZ::EBus<ChickenNotifications>;
}
```

**Example 25.4. ChickenAnimationComponent.h**

```
#pragma once
#include <AzCore/Component/Component.h>
#include <MyGem/ChickenBus.h>

namespace MyGem
{
    class ChickenAnimationComponent
        : public AZ::Component
        , public ChickenNotificationBus::Handler
    {
    public:
        AZ_COMPONENT(ChickenAnimationComponent,
                     "{ED8B6A79-AA47-44B2-91B6-64A78439B037}");

        static void Reflect(AZ::ReflectContext* rc);

        // AZ::Component interface implementation
        void Activate() override;
        void Deactivate() override;

        // ChickenNotificationBus interface
        void OnChickenSpeedChanged(float s) override;

    private:
        AZ::EntityId m_actor;
    };
} // namespace MyGem
```

**Example 25.5. ChickenAnimationComponent.h**

```
#include <ChickenAnimationComponent.h>
#include <AzCore/Serialization/EditContext.h>
#include <Integration/AnimGraphComponentBus.h>

namespace MyGem
```

```
{  
    void ChickenAnimationComponent::Reflect(AZ::ReflectContext* rc)  
    {  
        if (auto sc = azrtti_cast<AZ::SerializeContext*>(rc))  
        {  
            sc->Class<ChickenAnimationComponent, AZ::Component>()  
                ->Field("Actor", &ChickenAnimationComponent::m_actor)  
                ->Version(1);  
  
            if (AZ::EditContext* ec = sc->GetEditContext())  
            {  
                using namespace AZ::Edit;  
                ec->Class<ChickenAnimationComponent>(  
                    "Chicken Animation",  
                    "[Player controlled chicken]")  
                    ->ClassElement(ClassElements::EditorData, "")  
                    ->Attribute(  
                        Attributes::AppearsInAddComponentMenu,  
                        AZ_CRC_CE("Game"))  
                    ->DataElement(nullptr,  
                        &ChickenAnimationComponent::m_actor,  
                        "Actor", "Entity with chicken actor.");  
            }  
        }  
    }  
  
    void ChickenAnimationComponent::Activate()  
    {  
        ChickenNotificationBus::Handler::BusConnect(GetEntityId());  
    }  
  
    void ChickenAnimationComponent::Deactivate()  
    {  
        ChickenNotificationBus::Handler::BusDisconnect();  
    }  
  
    void ChickenAnimationComponent::OnChickenSpeedChanged(float s)  
    {  
        using namespace EMotionFX::Integration;  
        AnimGraphComponentRequestBus::Event(m_actor,  
            &AnimGraphComponentRequests::SetNamedParameterFloat,  
            "Speed", s);  
    }  
} // namespace MyGem
```

# Chapter 26. Animation State Machine

## Introduction

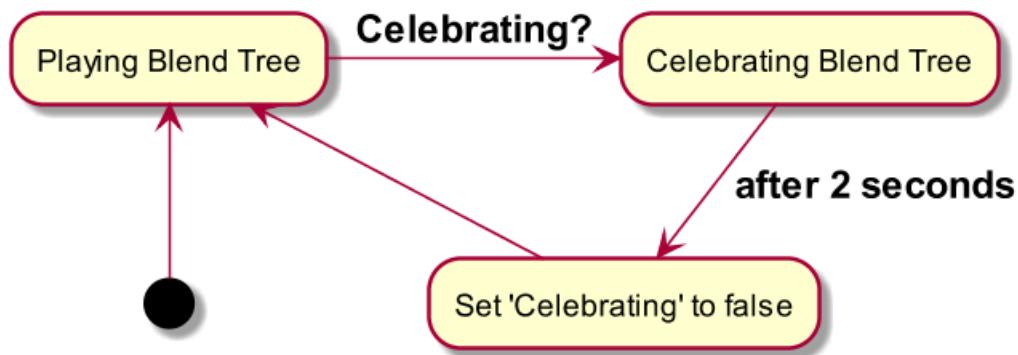
### Note

The level and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch26\\_animation\\_statemachine](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch26_animation_statemachine)

Building on Chapter 25, *Introduction to Animation*, we are going to add a chicken animation to celebrate scoring a goal using chicken flapping motion. This will involve building a state machine inside Animation Editor, defining transitions and parameter actions.

**Figure 26.1. Animation State Machine**

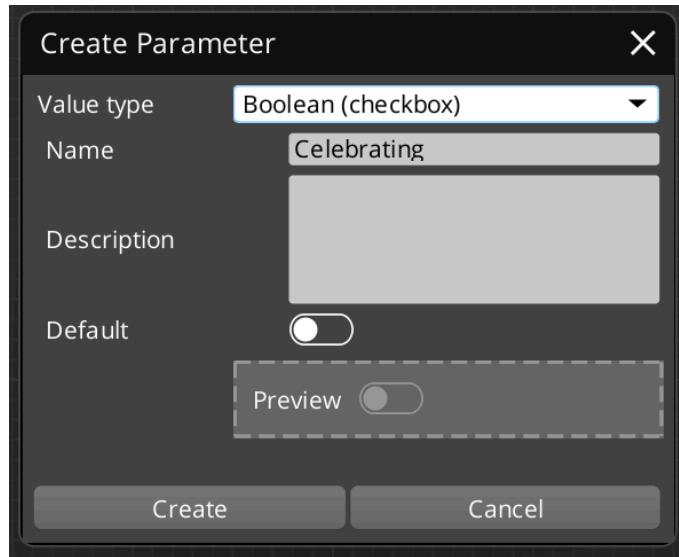


## Adding Flapping Motion

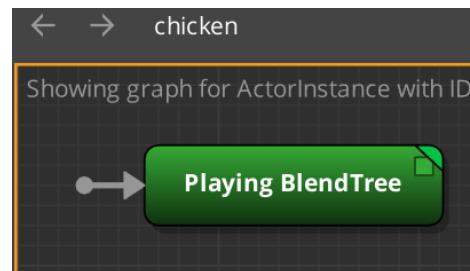
1. Open Animation Editor from Anim Graph component with `chicken.animgraph`.
2. Go to Motions Sets.
3. For MotionSet0, add `chickenflapping.motion` from `C:\O3DE\21.11.2\Gems\NvCloth\Assets\Objects\clothes\Chicken\Motions`.
4. Save the motion set.

## Adding Celebration Blend Tree

1. Add a **Boolean (checkbox)** parameter and name it "Celebrating". When the value of this parameter is set to true, the chicken will play a flapping motion to celebrate scoring a goal.
2. Set the default value of the parameter to false.



3. There is only one blend tree in the animation graph. Rename it to Playing BlendTree, so that we can distinguish it from the second new blend tree node that we are about to add.

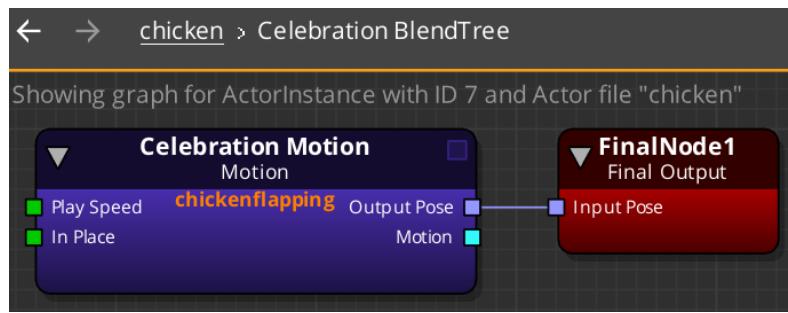


4. Create Node → Sources → Blend Tree.
5. Rename it to "Celebration BlendTree".
6. Double click Celebration Blend Tree.
7. Create Node → Sources → Motion.

### Note

When we were blending idle and walking motion we used a Blend Two node. However, for celebration motion we do not need to do any blending. We just want the chicken to play the flapping motion.

8. Rename the new motion node to Celebration Motion.
9. With the motion node selected, in Attributes panel, under Motions, assign **chickenflapping** motion from C:\O3DE\21.11.2\Gems\NvCloth\Assets\Objects\cloth\Chicken\Motions.
10. Connect Output Pose of Celebration Motion with Input Pose of FinalNode1.



11. Go back up a level.
12. Left click and drag an arrow from Playing Blend Tree to Celebration Blend Tree.

### Tip

In order to draw an arrow, left click closer to the edge of a node away from its name. If you hover your mouse over the node and slowly move it up, you will see the mouse pointer change its look that will indicate when the location will select the node or start creating an arrow.

**Figure 26.2. Adding a Transition between Blend Trees**

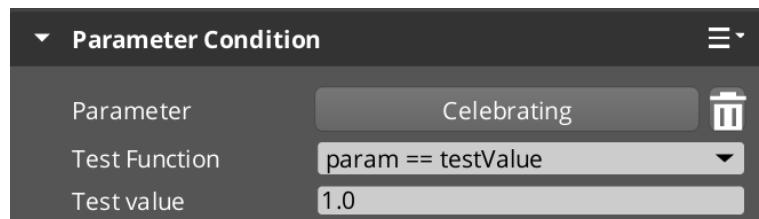


13. Select the arrow.

### Note

We are now going to add a condition when one blend tree will transition to the other one.

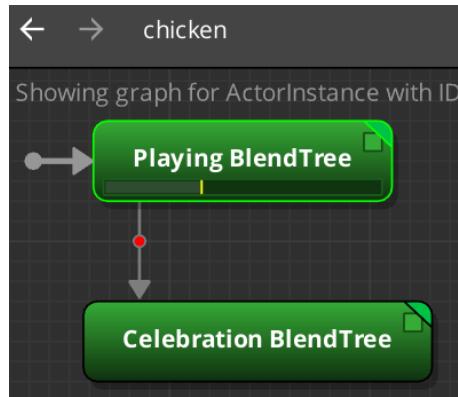
14. With the arrow selected, in Attributes panel, choose **Add condition** and then **Parameter Condition**.
15. Using "Select parameter" button, select **Celebrating** parameter.
16. For "Test Function", select equality condition.
17. For "Test value", set the value of one (1.0).



### >Note

A boolean value of "true" is treated as the value of one (1.0) by Animation Editor in parameter conditions.

18. That creates a condition to transition from "Playing Blend Tree" to "Celebration Blend Tree" when Celebrating parameter is set to true.



### >Note

Next step is to define when we exit the celebration state and reset Celebrating parameter back to false.

19. Left click and drag an arrow from Celebration Blend Tree to Playing Blend Tree.
20. Select the new arrow.
21. In Attributes panel, select Add condition, choose State Condition. This will add State Condition to the Attributes panel.
22. Under State, user Select node button to choose **Celebration Motion** node.
23. For Test Function, select "Has Reached Specified Playtime."
24. For Play Time, enter 2.0.

### >Note

The units for Play Time is seconds. If you take a look at "chicken flapping" motion under Motions tab, you will find that it is 2.47 seconds long. However, it has some idle time at the end of it, so I chose to transition away before the end of the motion.



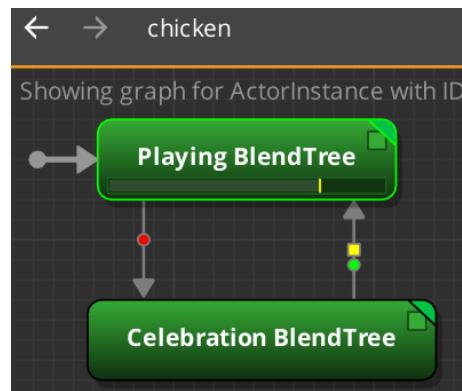
25. Now choose **Add action** in the same Attributes panel with the arrow still selected.

26. Choose Parameter Action.
27. Keep Trigger Mode as "On Enter."
28. For Parameter, select Celebration parameter.
29. For Trigger Value, keep 0.0. This creates an action of setting Celebration parameter value to zero (false for boolean parameters) when the selected transition occurs.

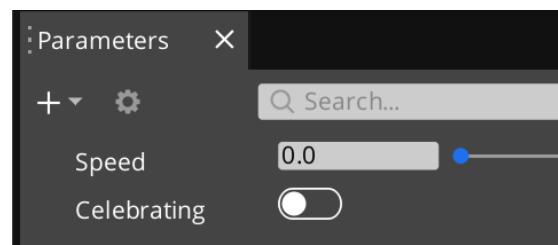


Change Parameter Value

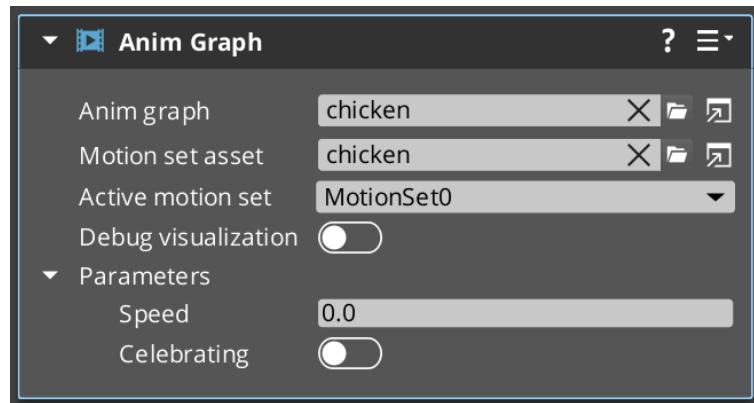
With these changes, we now have a state machine that switches between celebrating motion and blending between idle and walking motion.

**Figure 26.3. Animation State Machine**

You can test this logic by setting **Celebrating** parameter to enabled and then watch chicken celebration motion play for two seconds and then switch back to **Player Blend Tree** while setting Celebrating value back to false.

**Figure 26.4. List of Parameters**

Once you save the changes in the animation graph, **Anim Graph** component will update the list of parameters to reveal Celebrating boolean parameter.

**Figure 26.5. Anim Graph component with updated parameters**

## Summary

In this chapter, we enhanced the animation graph to add celebration motion and tested it in the Animation Editor. We know it works but there is no game logic to trigger it. We can do it by adding logic to a C++ component as we did in the previous chapter. However, we have already done that before, so in the next chapter I will show you how to do it using Script Canvas, the visual scripting in O3DE.

### Note

The assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch26\\_animation\\_statemachine](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch26_animation_statemachine)

# Chapter 27. Script Canvas

## Introduction

### Note

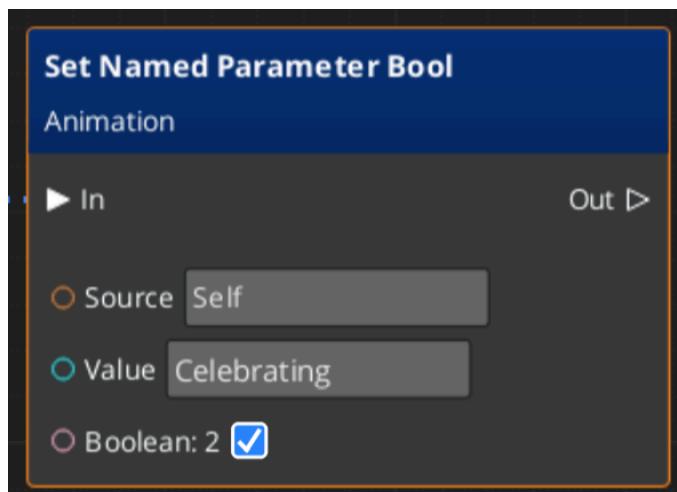
The assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch27\\_scriptcanvas](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch27_scriptcanvas)

Our goal for this chapter is to have the chicken celebrate each goal with a flapping motion. We created the animation graph to support that animation. We need to set 'Celebrating' animation parameter to true whenever a goal is scored. We could do it with C++ but this is a good opportunity to show you how to do the same work using visual scripting.

Script Canvas is a visual scripting tool in O3DE. For example, here is a visual node that is equivalent to the C++ method of `SetNamedParameterBool`, a similar method to what we used in earlier chapters.

### Example 27.1. Set Named Parameter Bool



As a reminder, `GoalDetectorComponent` sends out notifications when a goal is scored.

```
void GoalDetectorComponent::UpdateUi()
{
    UiScoreNotificationBus::Broadcast(
        &UiScoreNotificationBus::Events::OnTeamScored, m_team);
}
```

We need to sign up to receive this event and set *Celebration* parameter on **Anim Graph** component to true.

## Behavior Context

In order for Script Canvas to provide you with a node for a new custom notification EBus, we have to reflect that EBus to Behavior Context. Here are the steps to reflect a notification bus.

1. Here is the bus we are reflecting.

```
class UiScoreNotifications
    : public AZ::ComponentBus
{
public:
    virtual void OnTeamScored(int team) = 0;
};

using UiScoreNotificationBus = AZ::EBus<UiScoreNotifications>;
```

2. Create a behavior handler for it.

```
class ScoreNotificationHandler
```

3. Inherit from the EBus you are looking to reflect to scripting and from AZ::BehaviorEBusHandler.

```
class ScoreNotificationHandler
    : public UiScoreNotificationBus::Handler
    , public AZ::BehaviorEBusHandler
```

4. List all the methods you are looking to reflect with AZ\_EBUS\_BEHAVIOR\_BINDER macro. In this case, there is only OnTeamScored.

```
AZ_EBUS_BEHAVIOR_BINDER(ScoreNotificationHandler,
    "{33B5BC25-622B-4DF0-92CF-987CC6108C31}",
    AZ::SystemAllocator,
    OnTeamScored);
```

## Tip

If there were more methods to reflect you would list them after one at a time at the end of the macro.

```
AZ_EBUS_BEHAVIOR_BINDER(ScoreNotificationHandler,
    "{33B5BC25-622B-4DF0-92CF-987CC6108C31}",
    AZ::SystemAllocator,
    Method1,
    Method2,
    Method3);
```

5. Override each method to call FN\_<method name> methods that are generated by AZ\_EBUS\_BEHAVIOR\_BINDER. For example, there is FN\_OnTeamScored for OnTeamScored .

```
void OnTeamScored(int team) override
{
    Call(FN_OnTeamScored, team);
}
```

6. Register this handler with BehaviorContext. It can be done in any Reflect method. For example, in GoalDetectorComponent. It is a good spot, since it invokes an event on it. Another good option is to put it inside a system component of your gem or project.

```
void GoalDetectorComponent::Reflect(AZ::ReflectContext* rc)
{
```

```
//...
if (auto bc = azrtti_cast<AZ::BehaviorContext*>(rc))
{
    bc->EBus<UiScoreNotificationBus>("ScoreNotificationBus")
        ->Handler<ScoreNotificationHandler>();
}
}
```

### Tip

If you are reflecting a method to call from Script Canvas into C++, it takes just a few lines of code in C++. For example, here is the reflection for `SetNamedParameterBool` that we saw in the beginning of this chapter.

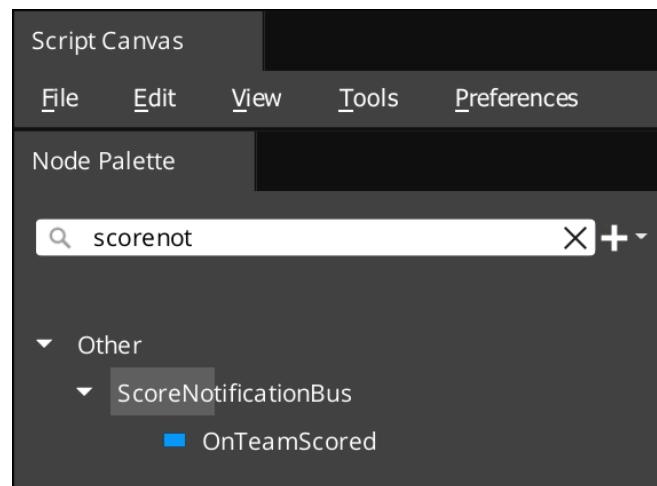
```
behaviorContext->EBus<AnimGraphComponentRequestBus>(
    "AnimGraphComponentRequestBus")
>Event("SetNamedParameterBool",
    &AnimGraphComponentRequestBus::Events::SetNamedParameterBool)
```

You can see this definition here: `C:\git\o3de\Gems\EMotionFX\Code\Source\Integration\Components\AnimGraphComponent.cpp`.

## Building a Canvas

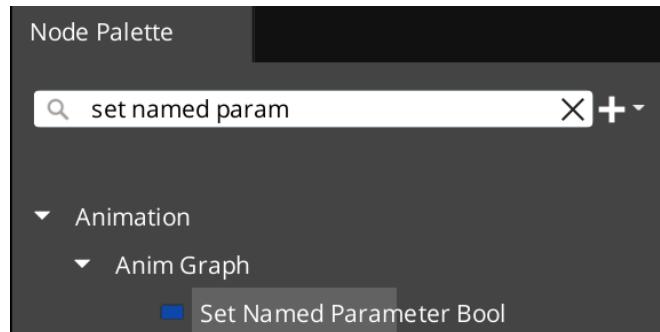
Once the project is compiled, you can re-open the Editor and create a new script canvas. Here are the steps to set **Celebration** parameter from Script Canvas.

1. Go to **Chicken\_Actor** entity. (It has **Anim Graph** component on it.)
2. Add Script Canvas component to the entity.
3. Open Script Canvas editor using Tools → Script Canvas or from the icon on Script Canvas component.
4. Once inside Script Canvas editor, start a new script with File → New Script.
5. In Node Pallete, search for "Score Notification Bus."



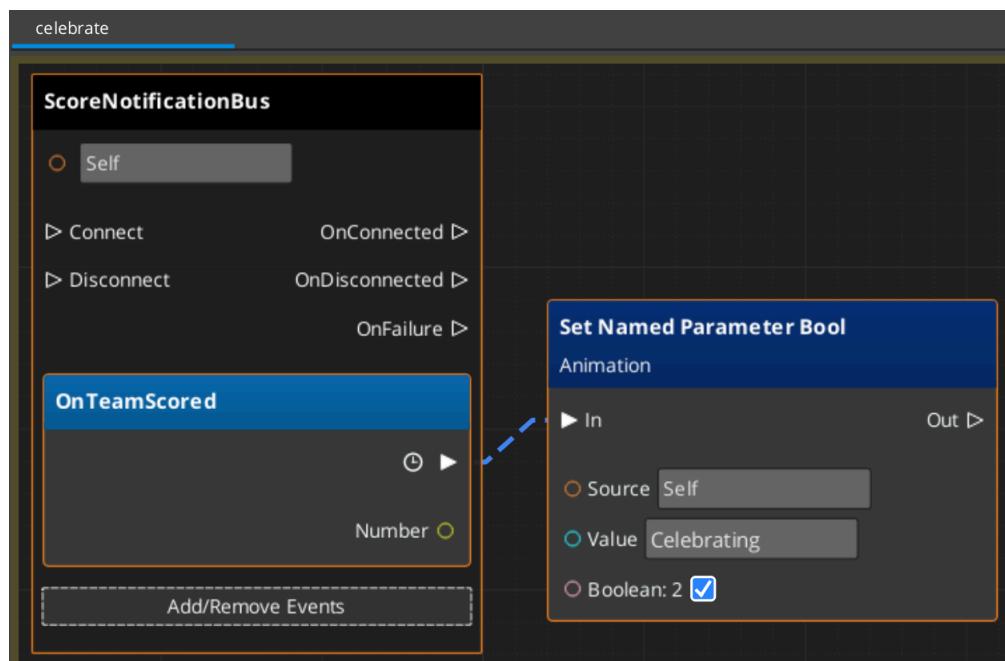
6. Drag and drop **OnTeamScored** into empty canvas space.

7. Search for "Set Named Parameter Bool" in Node Palette. Drag and drop it into the canvas as well.



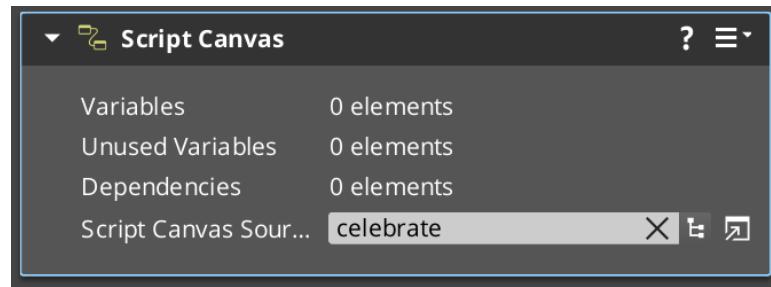
8. Connect the execution line from OnTeamScored to the **In** execution connector on "Set Named Parameter Bool" node. The notification node will start the execution of the script when an event is invoked. The execution will be passed to "Set Named Parameter Bool" node. Afterwards the **Out** connector will pass the execution to the next node if one is connected.

**Figure 27.1. Two Script Canvas Nodes**



9. On "Set Named Parameter Bool" node, set Value to Celebrating.
10. Set Boolean value to checked.
11. Save the canvas under the project or one of the gems, for example at C:\git\book\MyProject\scriptcanvas\celebrate.scriptcanvas.
12. Go back to Chicken\_Actor entity.
13. On Script Canvas component, assign the new canvas to Script Canvas Source File property.

### Example 27.2. Chicken\_Actor entity with Script Canvas component



With this component, anytime a goal is scored the chicken will play celebration flapping motion for two seconds.

## Summary

### Note

The source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch27\\_scriptcanvas](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch27_scriptcanvas)

Here are C++ code changes in this chapter.

### Example 27.3. New code in GoalDetectorComponent.cpp

```
void GoalDetectorComponent::Reflect(AZ::ReflectContext* rc)
{
    //...
    if (auto bc = azrtti_cast<AZ::BehaviorContext*>(rc))
    {
        bc->EBus<UiScoreNotificationBus>("ScoreNotificationBus")
            ->Handler<ScoreNotificationHandler>();
    }
}
```

### Example 27.4. UiScoreBus.h

```
#pragma once
#include <AzCore/Component/ComponentBus.h>
#include <AzCore/RTTI/BehaviorContext.h>

namespace MyGem
{
    class UiScoreNotifications
        : public AZ::ComponentBus
    {
    public:
        virtual void OnTeamScored(int team) = 0;
    };

    using UiScoreNotificationBus = AZ::EBus<UiScoreNotifications>;
}
```

```
/// NEW
class ScoreNotificationHandler
    : public UiScoreNotificationBus::Handler
    , public AZ::BehaviorEBusHandler
{
public:
    AZ_EBUS_BEHAVIOR_BINDER(ScoreNotificationHandler,
        "{33B5BC25-622B-4DF0-92CF-987CC6108C31}",
        AZ::SystemAllocator, OnTeamScored);

    void OnTeamScored(int team) override
    {
        Call(FN_OnTeamScored, team);
    }
};
```

## Note

You can find documentation on how to reflect various classes, structures and interfaces to Behavior Context at: <https://docs.o3de.org/docs/user-guide/programming/components/reflection/behavior-context/>

---

## **Part X. Audio Effects (Wwise)**

---

## Table of Contents

28. Wwise for O3DE .....	239
Introduction .....	239
Installing Wwise .....	239
Building O3DE Installer with Wwise .....	240
Setting up a Wwise Project .....	241
Generating a Sound Bank .....	245
Summary .....	246
29. Importing Wwise Project .....	247
Import to O3DE .....	247
Summary .....	248
30. Introduction to Audio Components .....	249
Introduction .....	249
Audio Trigger Component .....	249
Audio Proxy Component .....	252
Audio Listener Component .....	252
Summary .....	253

---

# Chapter 28. Wwise for O3DE

You'd better wise up, man!

—The Outsiders, 1983

## Introduction

### Note

Source code for this chapter can be found on GitHub:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch28\\_wwise\\_setup](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch28_wwise_setup)

This chapter will take a look at adding sound effects to our project. O3DE comes with an integration of Audiokinetic Wwise. However, the installer does not include it, since Wwise is not open-source.

### Important

Wwise is not a product of O3DE. It is stand-alone product. See its website for full conditions, licensing and details.

<http://www.audiokinetic.com/pricing>

In this chapter we will built a new installation of O3DE with Wwise integration enabled.

## Installing Wwise

1. Navigate to <https://www.audiokinetic.com/download/>
2. Download the installer for Windows.
3. Install and open Wwise Launcher.
4. Install Wwise 2021 version with "Authoring" and "SDK (C++)" packages.

### Note

At the time of writing this book, Wwise 2021.1.7.7796 was available and used.

### Important

If you can see environment variable WWISEROOT on the console, then the installation was successful. You may need to re-open a command console (and maybe even reboot your system) to see it after the installation.

```
C:\git\book\build>set | findstr WWISEROOT  
WWISEROOT=D:\Program Files (x86)\  
Audiokinetic\Wwise 2021.1.7.7796
```

During CMake configure step, you should see a message such as:

```
1>-- Using Wwise SDK at D:\Program Files (x86)\
```

```
Audiokinetic\Wwise 2021.1.7.7796
```

You can also test it by adding the following to your gem or project to confirm that CMake sees Wwise installation.

```
BUILD_DEPENDENCIES
PUBLIC
3rdParty::Wwise
```

## Building O3DE Installer with Wwise

### ⚠ Important

O3DE pre-built installer that we have used so far does not include Wwise gem.

We are going to build a new local O3DE installer with Wwise gem enabled.

1. Clone the repository of O3DE with tag **2111.2** to match the installer we have used so far in the book.

```
git clone -b 2111.2 https://github.com/o3de/o3de
```

2. I will assume that the repository was cloned at `c:\git\o3de`.
3. Create a build folder at `c:\git\o3de\build`.
4. From the build folder, configure CMake. This should pick up Wwise installation.

```
C:\git\o3de\build> cmake -S .. -B .
```

5. Open Visual Studio solution at `c:\git\o3de\build\O3DE.sln`.
6. Check that there is **Gems\AudioEngineWwise** present in the solution.
7. Build **INSTALL** target.

### 💡 Note

This step will take a while, since it builds the entire engine, all the gems and creates a local installation.

8. The new engine installation will be placed at `c:\git\o3de\install`.
9. Open `C:\git\o3de\install\engine.json`.
10. Modify **engine\_name** and **restricted\_name** to "my-o3de-2111".
11. Configure python in the new engine.

```
C:\git\o3de\install> .\python\get_python.bat
```

12. Register the engine.

```
C:\git\o3de\install> .\scripts\o3de.bat register --this-engine
```

13. Open `C:\Users\<user>\.o3de\o3de_manifest.json`.

14. Verify that this engine was registered. You should see the following lines.

```
{  
...  
    "engines": [  
        "C:/O3DE/21.11.2",  
        "C:/git/o3de/install",  
    ...  
    ],  
    "engines_path": {  
        "o3de-sdk": "C:/O3DE/21.11.2",  
        "my-o3de-2111": "C:/git/o3de/install",  
    ...  
}
```

15. Switch MyProject to use this new engine by modifying C:\git\book\MyProject\project.json. Change "engine" from "o3de-sdk" to "my-o3de-2111".

```
    "engine": "my-o3de-2111",
```

16. Re-compile the project.

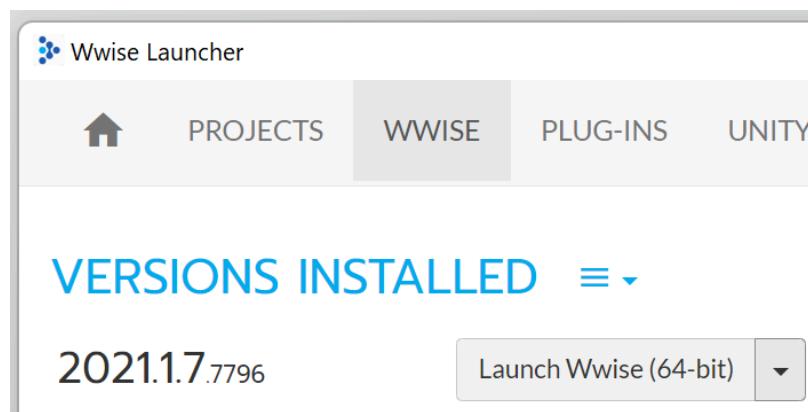
17. Re-run Asset Processor. (It is going to re-process all the assets.)

### Tip

Avoid starting the Editor before all the assets are processed. Crashes may occur before all the critical assets are ready.

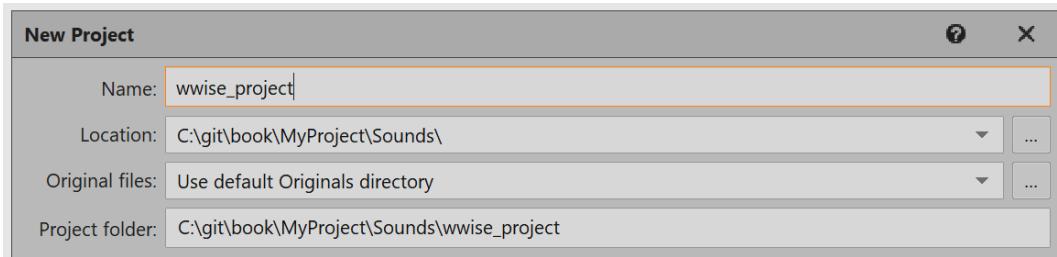
## Setting up a Wwise Project

O3DE pulls audio resources from a collection of audio assets that are called sound banks. These sound banks are generated by a Wwise project. We will create a brand new Wwise project by using Wwise Launcher.



Go to WWISE tab and launch Wwise

Initially, you would not have any Wwise projects, so you will need to create a new one.



In "New Project" dialog set the following values:

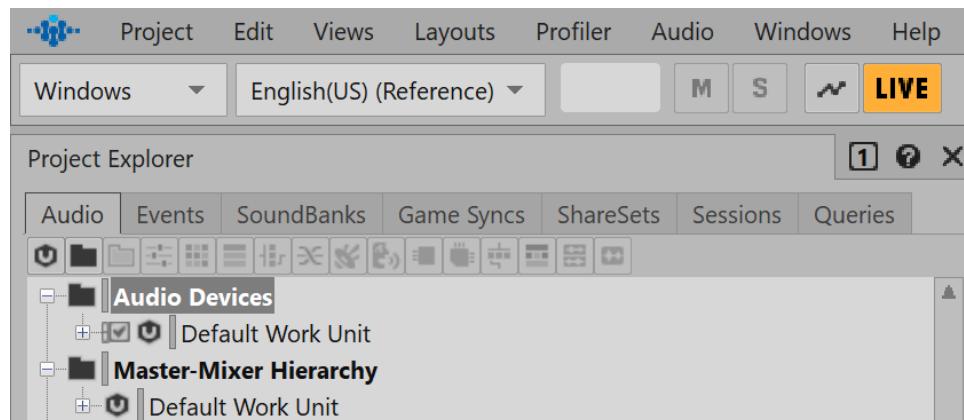
- For Name, enter: `wwise_project`
- Location: `C:\git\book\MyProject\Sounds\`
- Original files: "Use default Originals directory"
- Project folder: `C:\git\book\MyProject\Sounds\wwise_project`

## ⚠ Important

There are hidden requirements that O3DE places on Wwise projects.

- The name of the project has to be **`wwise_project`**. Always.
- It has to be placed at `MyProject\Sounds\wwise_project` where `MyProject` could be any of your projects.

Once you create the project you will be greeted with **Wwise Project Explorer**.

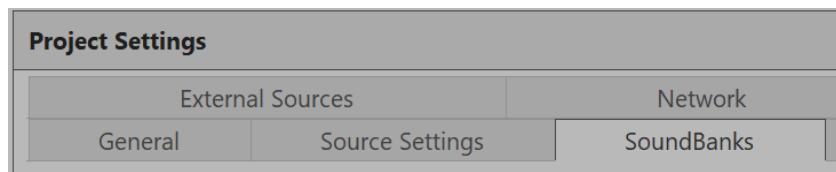


Wwise application

## ⚠ Important

Wwise's job is to generate sound banks for O3DE. However, O3DE expects the banks to be at a specific location, which is different from the default sound bank path set by Wwise. So before we get to adding audio files to the project, configure an important project setting that is vital to get audio working with O3DE. From the main menu, select **Project→Project Settings...**

Switch to **SoundBanks** tab.

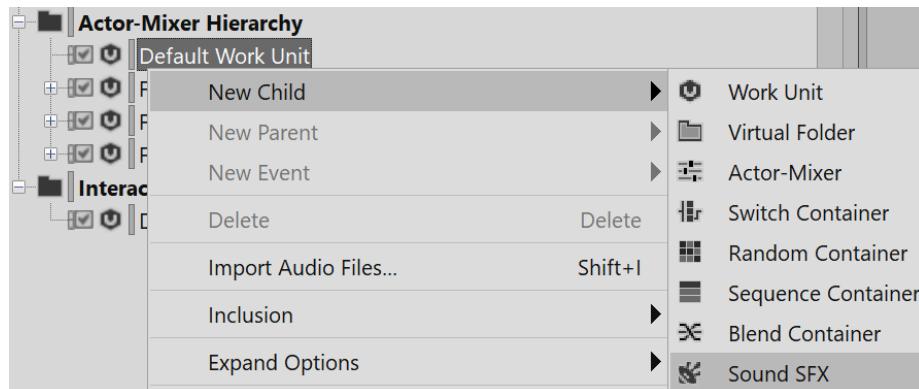


We need to change **SoundBack Folder** path from GeneratedSoundBanks\Generic\ to MyProject\Sounds\wwise. (If wwise folder is not there yet, create it.) When you do that Wwise will adjust the path to make it a relative path. So do not be surprised when it changes to .. \wwise. This is as it ought to be.



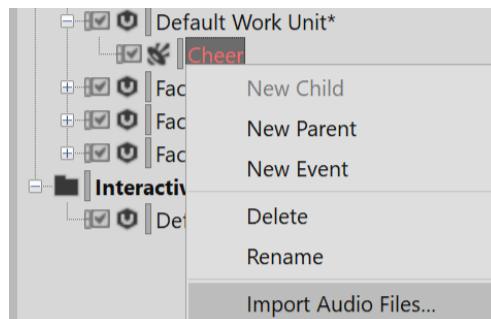
SoundBank path for O3DE

Create a new audio element under Actor-Mixer Hierarchy. Right click on **Default Work Unit** → **New Child** → **Sound SFX**.

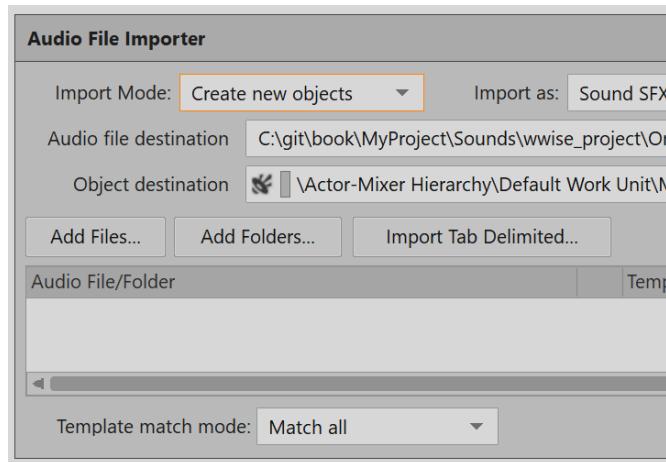


New Child → Sound SFX

**Sound SFX** is the simplest way to get started with adding audio to a Wwise project. It represents a single audio effect. Rename new sound sfx to **Cheer**. The next step is to assign it an audio asset by right clicking on **Cheer** → **Import Audio Files...**



The importer dialog will show up. Select **Add Files...** and navigate to applause.wav at C:\git\book\MyProject\Assets\Source\_Audio\. Click on **Import** in Audio File Importer dialog.



Audio file importer dialog

That will assign the wave file to **Cheer**. You should see that in Contents Editor.

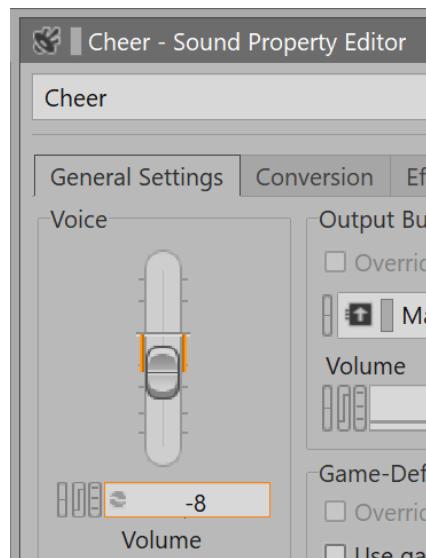


Imported music asset

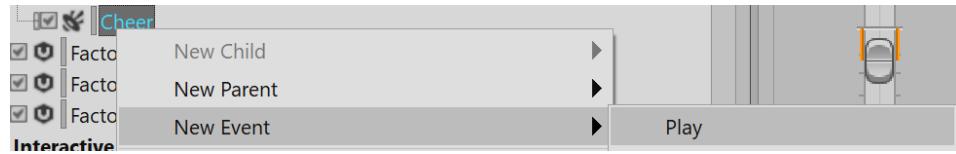
### Tip

You can press SPACE while **Cheer** is selected to test if the sound is the one you wanted and that it actually plays, at least inside Wwise application.

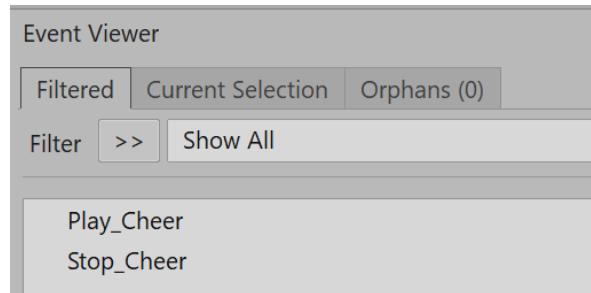
Personally, I found that the applause sound effect was too loud for my taste. So I modified the volume of this sound element to -8 in Sound Property Editor.



Each audio object in Wwise should have two events: play and stop. O3DE will manipulate the object through these events. You have to explicitly create them first in Wwise by right clicking on **Cheer** → **New Event** → **Play**.



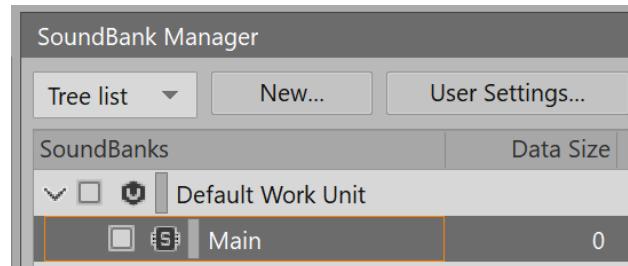
"Stop" event requires a similar procedure: **Cheer** → **New Event** → **Stop** → **Stop**. Now, under Event Viewer with **Cheer** selected, you should see new events.



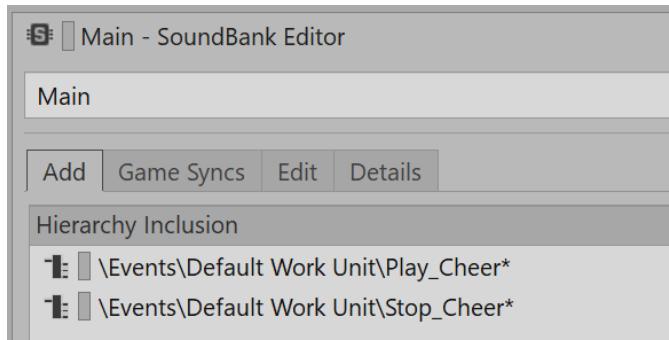
Play and stop events

## Generating a Sound Bank

1. From the main menu. Switch to Layouts → Soundbank.
2. Under SoundBankManager, click **New...** This will create a new sound bank for us.
3. In "New SoundBank" dialog, under Name enter "Main". Click OK.



4. From Event Viewer, drag and drop Play\_Cheer and Stop\_Cheer events into Main sound bank in SoundBank Manager.
5. SoundBank Editor window should now list under the events under **Hierarchy Inclusion**.



6. Back from SoundBank Manager, click **Generate All** button.

## Summary

One last step is to enable AudioSystem and AudioEngineWwise gems in `C:\git\book\MyProject\Code\enabled_gems.cmake`

```
set(ENABLED_GEMS
...
    AudioSystem
AudioEngineWwise
)
```

That finishes the setup of Wwise for the project.

### Important

You should see `Main.bnk` inside `C:\git\book\MyProject\Sounds\wwise` folder. It should have been generated by Wwise project.

### Note

Source code for this chapter can be found on GitHub:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch28\\_wwise\\_setup](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch28_wwise_setup)

The next step is to import these assets into O3DE. The next chapter will show you how to do that.

# Chapter 29. Importing Wwise Project

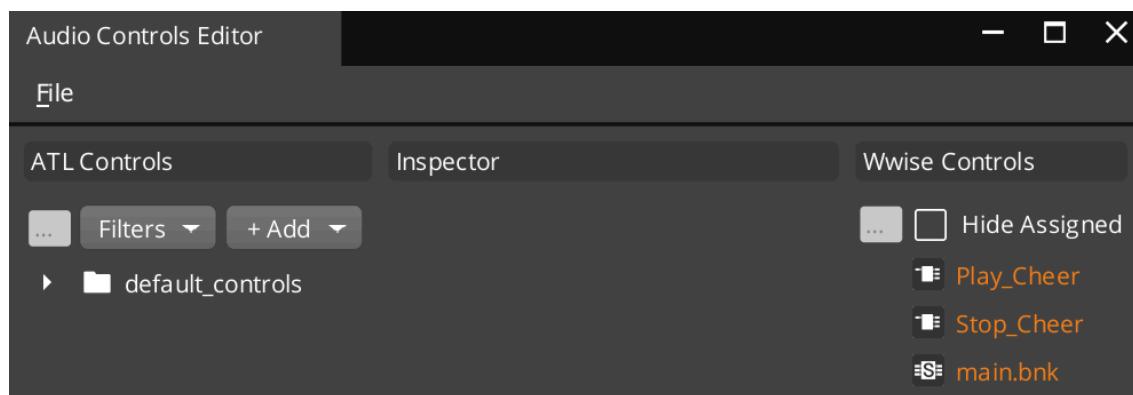
## Import to O3DE

### Note

You can find the assets for this chapter on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch29\\_wwise\\_import](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch29_wwise_import)

The sound banks generated by Wwise in the previous chapter are not immediately usable in O3DE. One has to create sound controls for them. They have to be manually created in via Audio Controls Editor from the main menu: **Tools→ Other→ Audio Controls Editor**.



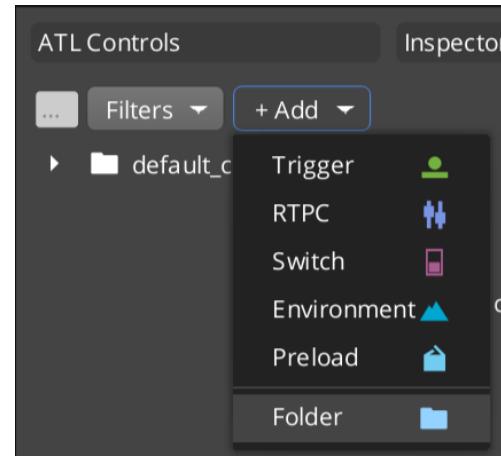
Audio Controls Editor

If everything went right so far, you should see the new events Play\_Cheer, Stop\_Cheer and the sound bank main.bnk on the right panel, which is Wwise Controls. That is the data and events that came from Wwise project.

The left panel "ATL Controls" lists the sound events and banks imported to the project.

The task in this editor is to create sound controls on the left from Wwise controls on the right. The first step is to create a new **Folder** on the left for organizational purposes.

The **default\_controls** folder is the legacy content that we can safely ignore.



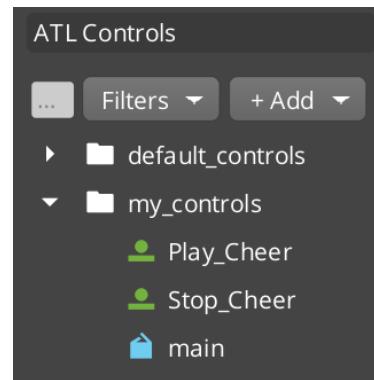
Add Folder

I have chosen to name the folder "my\_controls." Drag Play\_Cheer and Stop\_Cheer and main.bnk from Wwise Controls on the right into the new folder.

Play\_Cheer is the event to start playing the cheer sound effect.

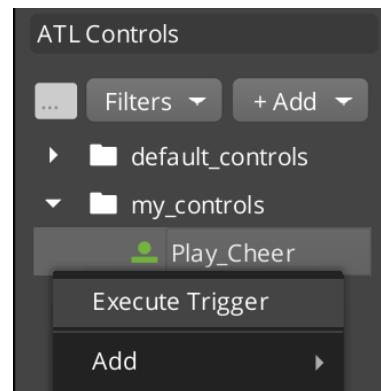
Stop\_Cheer is the event to stop playing the cheer sound effect.

The "main" bank contains the cheer sound data.

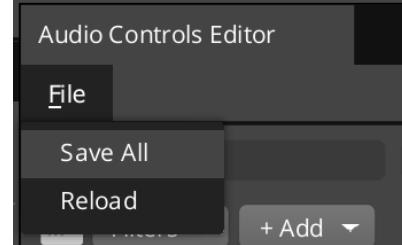


At this point, you can test that the sound effect by right clicking on **Play\_Cheer** and choosing **Execute Trigger** or pressing **SPACE** with the event selected in Auto Controls Editor.

If you can hear the sound effect, then everything works!



Save the changes with **File→ Save All**.



## Summary

We went over how to add sound effects to Wwise project and import them to O3DE projects. We are ready to start controlling and playing them in the game.

### Note

You can find the assets for this chapter on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch29\\_wwise\\_import](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch29_wwise_import)

# Chapter 30. Introduction to Audio Components

## Introduction

So far we created a Wwise project, added a cheer sound effect and imported sound events that control it into the project. This chapter will take a look at playing sounds. We will need to do the following:

- Add audio components: an Audio Trigger, an Audio Proxy and, optionally, an Audio Listener.
- Populate them with events controlling "cheer" sound effect.
- Play "cheer" sound effect using Script Canvas when a goal is scored.

### Note

You can find the code and the assets for this chapter on GitHub:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch30\\_audio\\_components](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch30_audio_components)

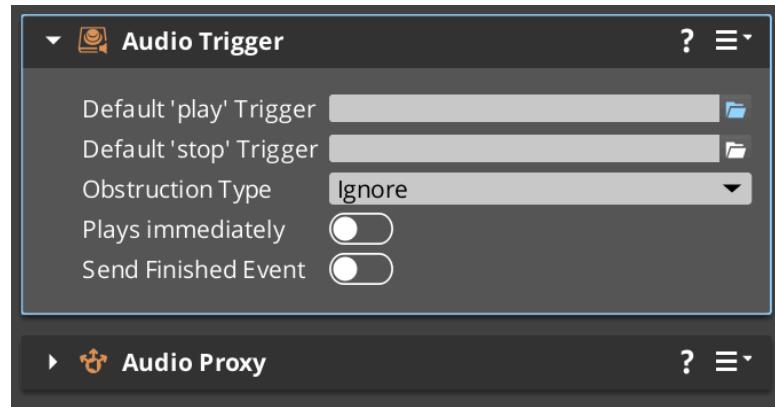
## Audio Trigger Component

Create a new entity on the level, for example **Goal Cheering**. Add an **Audio Trigger** component. It is used to trigger playing and stopping a sound effect.



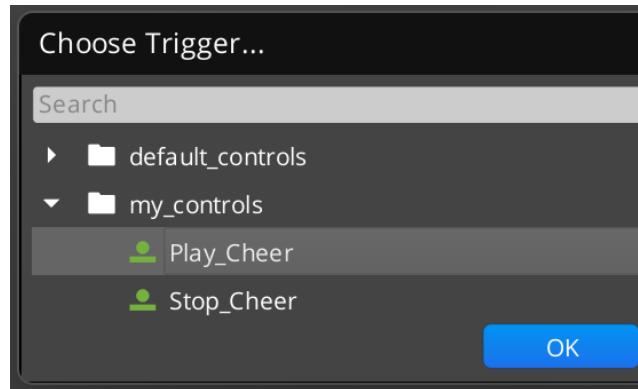
Missing dependency of Audio Trigger

An entity with an audio trigger requires an audio proxy component.

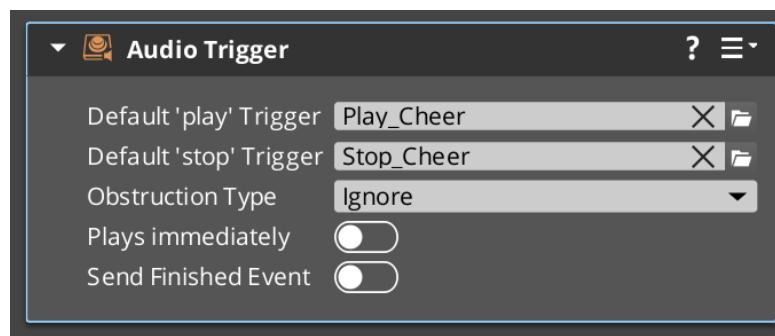


Empty audio component

Default 'play' and 'stop' triggers are the control events. For our cheering sound effect they need to be Play\_Cheer and Stop\_Cheer events that we added in Audio Controls Editor.



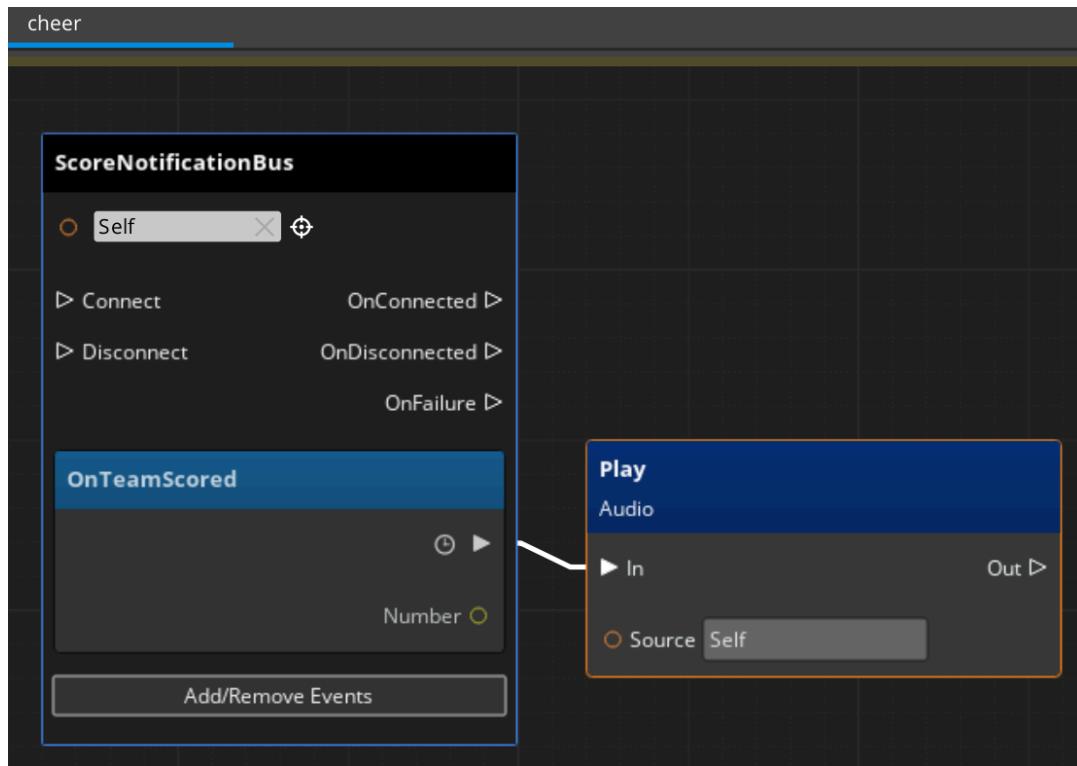
Once both triggers are assigned, we are done with configuring the Audio Trigger component.



Completed Audio Trigger component

Audio Trigger components can be controlled via its EBus with `AudioTriggerComponentRequest-Bus`. In Script Canvas you can invoke the trigger by connecting "On Team Scored" node with Audio Play node, in a similar way we used it in Chapter 27, *Script Canvas*.

### Example 30.1. Playing the Trigger from Script Canvas



- Save the script as MyProject\scriptcanvas\cheer.scriptcanvas
- On "Goal Cheering" entity, add Script Canvas component.
- Assign the new script.

Now every time a goal is scored, the cheer sound effect will play.

## AudioTriggerComponentRequestBus

We can also play a sound effect from C++ using EBus AudioTriggerComponentRequestBus. You can find it at LmbrCentral\Audio\AudioTriggerComponentBus.h. Here is the interesting part for this chapter.

### Example 30.2. AudioTriggerComponentBus.h

```
class AudioTriggerComponentRequests
    : public AZ::ComponentBus
{
public:
    //! Executes the play trigger if set.
    virtual void Play() = 0;

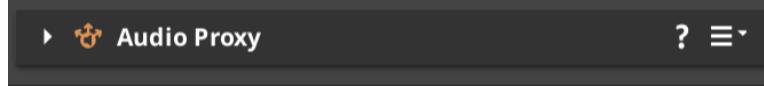
    //! Kills the play trigger if no stop trigger is set.
    //! Otherwise Executes the stop trigger.
    virtual void Stop() = 0;
```

So if you had a C++ component on the entity with an Audio Trigger component, then you could call it this way:

### Example 30.3. Playing a Sound from C++

```
AudioTriggerComponentRequestBus::Event(GetEntityId(),
    &AudioTriggerComponentRequestBus::Events::Play);
```

## Audio Proxy Component



While Audio Trigger component controls *playing* a sound effect, **Audio Proxy** component is the sound effect holder. It determines the sound location among other things. The sound effect location follows the entity's position by default. Audio Proxy's EBus is `AudioProxyComponentRequestBus` from "LmbrCentral" gem at `LmbrCentral\Audio\AudioProxyComponentBus.h`. For example, you can set the position manually or change movement following behavior.

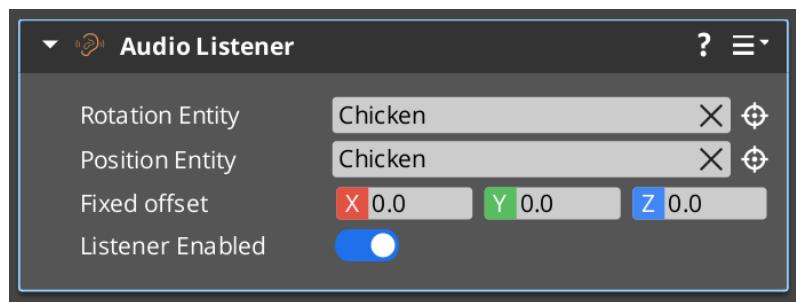
```
class AudioProxyComponentRequests
    : public AZ::ComponentBus
{
public:
    //...

    virtual void SetMovesWithEntity(bool shouldTrackEntity) = 0;

    /// Set the position of the audio proxy directly.
    virtual void SetPosition(const Audio::SATLWorldPosition& p) = 0;
};
```

## Audio Listener Component

The default sound listener is located at (0,0,0) coordinates. In order to specify the location, use Audio Listener component. A common location for an audio listener component is the camera entity or the character entity.



If you do not specify "Rotation Entity" and "Position Entity" on Audio Listener component, the entity's position will be used as the location of the audio listener. Or you can specify Chicken entity.

# Summary

There was no C++ code in this chapter, since we used Script Canvas to connect "On Team Scored" notification event with playing a sound effect.

## Note

You can find the code and the assets for this chapter on GitHub:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch30\\_audio\\_components](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch30_audio_components)

---

# Part XI. Multiplayer

You have made it! Welcome to multiplayer section of the book! In preceding chapters we created a single player game where a single chicken can kick the ball around and score goals on either side of the field. In this section, we will convert the game to multiplayer where each player can control a chicken from a different team.

**O3DE Multiplayer is a server-authoritative system with support for local prediction of player's input.** We will write our game logic in such a way that the server determines what goes in the game, while keeping player controls responsive. There is a lot to go through but I will break up the material into small consumable chapters.

---

# Table of Contents

31. Setting Up Multiplayer .....	257
Introduction .....	257
Enabling Multiplayer Code Generation .....	257
Building Installer from Development Branch of O3DE .....	260
Summary .....	262
32. Auto Components .....	263
Introduction .....	263
Code Generation .....	263
Components and Controllers .....	264
Summary .....	270
33. Multiplayer in the Editor .....	271
Introduction .....	271
Editor Server .....	271
Multiplayer Components in the Level .....	272
Summary .....	273
34. Simple Player Spawner .....	274
Introduction .....	274
Design .....	274
Summary .....	279
35. Multiplayer Input Controls .....	282
Introduction .....	282
Chicken Movement Component .....	282
Create Input .....	284
Process Input .....	287
Summary .....	290
36. Multiplayer Physics .....	296
Introduction .....	296
Network Ball .....	296
Network Property .....	296
Multiplayer Goal Detector .....	297
Summary .....	300
37. Removal and Spawning .....	304
Introduction .....	304
Ball Component .....	305
Ball Spawner Component .....	305
Entity Changes .....	308
Summary .....	308
38. Multiplayer Animation .....	313
Introduction .....	313
Changes to Chicken Movement Component .....	313
Chicken Animation Component .....	314
Entity Changes .....	315
Summary .....	316
39. Team Spawner .....	319
Introduction .....	319
Adding Team Value .....	320
Adding More Chickens .....	320
Changes to Chicken Component .....	322
Changes to Chicken Spawn Component .....	323
Summary .....	323
40. Multiplayer Camera .....	327

Introduction .....	327
Autonomous Controllers .....	327
Chicken Camera Component .....	327
Entity Changes .....	329
Summary .....	329

---

# Chapter 31. Setting Up Multiplayer

## Introduction

### Note

The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch31\\_enable\\_multiplayer\\_gem](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch31_enable_multiplayer_gem)

Our project has Multiplayer gem enabled, however, more setup is required before we can start writing multiplayer components.

### Example 31.1. A lot more is required

```
set(ENABLED_GEMS
    ...
    Multiplayer
)
```

In O3DE writing multiplayer components involves code generation, so we have to add code generation support to gems and projects that will be generating and compiling multiplayer components.

## Enabling Multiplayer Code Generation

In this chapter, we will enable multiplayer component support for MyGem that we have at C:\git\book\Gems\MyGem.

### Important

For any gems that include multiplayer components, the same steps will be required as well.

1. Add code generation files to MyGem.Static build target.

### Example 31.2. mygem\_autogen\_files.cmake

```
set(MPGEMPATH ${LY_ROOT_FOLDER}/Gems/Multiplayer/Code/Include)
set(AUTOPATH ${MPGEMPATH}/Multiplayer/AutoGen)

set(FILES
    ${AUTOPATH}/AutoComponent_Common.jinja
    ${AUTOPATH}/AutoComponent_Header.jinja
    ${AUTOPATH}/AutoComponent_Source.jinja
    ${AUTOPATH}/AutoComponentTypes_Header.jinja
    ${AUTOPATH}/AutoComponentTypes_Source.jinja
)
```

2. Add this list to MyGem.Static CMake target.

```
ly_add_target(
    NAME MyGem.Static STATIC
    FILES_CMAKE
```

```
mygem_autogen_files.cmake
...

```

3. Add code generation rules to MyGem.Static:

```
ly_add_target(
    NAME MyGem.Static STATIC
    ...
    AUTOGEN_RULES
        *.AutoComponent.xml,AutoComponent_Header.jinja,
        $path/$fileprefix.AutoComponent.h
        *.AutoComponent.xml,AutoComponent_Source.jinja,
        $path/$fileprefix.AutoComponent.cpp
        *.AutoComponent.xml,AutoComponentTypes_Header.jinja,
        $path/AutoComponentTypes.h
        *.AutoComponent.xml,AutoComponentTypes_Source.jinja,
        $path/AutoComponentTypes.cpp
)

```

## ⚠ Important

Each line of AUTOGEN\_RULES should have no line breaks in it. The book format forces line breaks but this portion should actually look like this in your CMakeLists.txt:

```
AUTOGEN_RULES
    *.AutoComponent.xml,Auto...ileprefix.AutoComponent.h
    *.AutoComponent.xml,Auto...ileprefix.AutoComponent.cpp
    *.AutoComponent.xml,Auto...ComponentTypes.h
    *.AutoComponent.xml,Auto...ComponentTypes.cpp
```

Otherwise, you will see odd CMake errors.

4. Add dependency on Multiplayer.Static for MyGem.Static. It is needed by multiplayer components.

```
ly_add_target(
    NAME MyGem.Static STATIC
    BUILD_DEPENDENCIES
        PUBLIC
            # For autogen components
            Gem::Multiplayer.Static
    ...
)
```

5. At this point, CMake build target for MyGem.static should look as follows.

### Example 31.3. Completed MyGem.Static with multiplayer enabled

```
ly_add_target(
    NAME MyGem.Static STATIC
    NAMESPACE Gem
    FILES_CMAKE
        mygem_files.cmake
        mygem_autogen_files.cmake
    INCLUDE_DIRECTORIES
        PUBLIC
            Include
```

```
PRIVATE
    Source
BUILD_DEPENDENCIES
PUBLIC
    AZ::AzCore
    AZ::AzFramework
    Gem::StartingPointInput.Static
    Gem::LyShine.Static
    Gem::PhysX.Static
    Gem::EMotionFXStaticLib
    # For autogen components
    Gem::Multiplayer.Static
AUTOGEN_RULES
    *.AutoComponent.xml,Au...refix.AutoComponent.h
    *.AutoComponent.xml,Au...refix.AutoComponent.cpp
    *.AutoComponent.xml,Au...ComponentTypes.h
    *.AutoComponent.xml,Au...ComponentTypes.cpp
)
```

6. Add multiplayer component descriptors to the gem. You need one such call for all multiplayer components in a gem.

#### Example 31.4. Changes to MyGemModuleInterface.h

```
#include <Source/AutoGen/AutoComponentTypes.h>
MyGemModuleInterface()
{
    m_descriptors.insert(m_descriptors.end(), {
        ...
    });
    ...
    //< Register multiplayer components
    CreateComponentDescriptors(m_descriptors);
}
```

#### Note

Some gems do not have a ModuleInterface and instead register component descriptors inside their Module constructors. Either way, you should add `CreateComponentDescriptors` to where you add component descriptors to AZ::Module's `m_descriptors`.

7. Register components with the Multiplayer system.

#### Example 31.5. Changes to MyGemSystemcomponent.cpp

```
#include <Source/AutoGen/AutoComponentTypes.h>
void MyGemSystemComponent::Activate()
{
    //< Register multiplayer components
    RegisterMultiplayerComponents();
}
```

8. Create multiplayer component XML definitions at `C:\git\book\Gems\MyGem\Code\Source\AutoGen`. Here is a starting multiplayer component with no C++ code required.

### Example 31.6. MyFirstNetworkComponent.AutoComplete.xml

```
<?xml version="1.0"?>
<Component
    Name="MyFirstNetworkComponent"
    Namespace="MyGem"
    OverrideComponent="false"
    OverrideController="false"
    OverrideInclude=""
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
</Component>
```

9. Include `MyFirstNetworkComponent.AutoComplete.xml` in the build.

### Example 31.7. Changes to Gems\MyGem\Code\mygem\_files.cmake

```
set(FILES
...
    Source/AutoGen/MyFirstNetworkComponent.AutoComplete.xml # new
)
```

10. Re-compile the gem and the project. If you see a CMake log line similar to the following snippet, you have successfully configured the gem to support multiplayer components.

```
2>Generating C:\git\book\build\External\MyGem-409da4e6\
    Code\Azcg\Generated\Source\AutoGen\
    MyFirstNetworkComponent.AutoComplete.h using
    template C:/git/o3de/install/Gems/Multiplayer/
    Code/Include/Multiplayer/AutoGen/AutoComponent_Header.jinja
    and inputs C:\git\book\Gems\MyGem\Code\Source\
    AutoGen\MyFirstNetworkComponent.AutoComplete.xml
```

## Building Installer from Development Branch of O3DE

### Important

If you are using O3DE 22.05.0 or later, then you can skip this section.

In order to get the best experience with multiplayer features of O3DE, we have to get features that were implemented since the release of O3DE Stable 21.11.2. Here are the steps to create a new installer from scratch from <https://github.com/o3de/o3de/tree/development>.

### Tip

You can get the latest nightly pre-built installer online at: <https://o3debinaries.org/download/windows.html>, look for "Nightly Development Build."

1. Clone O3DE repository.

```
git clone https://github.com/o3de/o3de
```

## >Note

At the time of writing this book, I tested the rest of the book with development branch at commit **b4b7e5a** from March 26th, 2022. You can reset to that point with the following git command.

```
git reset --hard b4b7e5a
```

2. I will assume that the repository was cloned at `c:\git\o3de`.
3. Create a build folder at `c:\git\o3de\build`.
4. From the build folder, configure CMake.

```
C:\git\o3de\build> cmake -S .. -B .
```

5. Open Visual Studio solution at `c:\git\o3de\build\O3DE.sln`.
6. Build **INSTALL** target.

## >Note

This step will take a while, since it builds the entire engine and all the gems.

7. The new engine installation will be at `c:\git\o3de\install`.
8. Open `C:\git\o3de\install\engine.json`.
9. Modify `engine_name` and `restricted_name` to "o3de-install".

10. Configure python in the new engine.

```
C:\git\o3de\install> .\python\get_python.bat
```

11. Register the engine.

```
C:\git\o3de\install> .\scripts\o3de.bat register --this-engine
```

12. Open `C:\Users\<user>\.o3de\o3de_manifest.json`.

13. Verify that this engine was registered. You should see the following lines.

```
{  
  ...  
  "engines": [  
    "C:/O3DE/21.11.2",  
    "C:/git/o3de/install",  
    ...  
  ],  
  "engines_path": {  
    "o3de-sdk": "C:/O3DE/21.11.2",  
    "o3de-install": "C:/git/o3de/install",  
    ...  
  }  
}
```

14. Switch MyProject to use this new engine by modifying `C:\git\book\MyProject\project.json`. Change "engine" from "o3de-sdk" to "my-o3de-2111".

```
"engine": "o3de-install"
```

15. Re-compile the project.
16. Re-run Asset Processor. It is going to re-process all the assets.

```
C:\git\o3de\install\bin\Windows\profile\Default\AssetProcessor.exe  
--project-path C:\git\book\MyProject\
```

Now we can use the latest and greatest features of multiplayer in O3DE!

## Summary

### Note

The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch31\\_enable\\_multiplayer\\_gem](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch31_enable_multiplayer_gem)

We have enabled code generation of multiplayer components for MyGem. Now we are free to start looking at what is the format of \*.AutoComponent.xml files, what they generate and what you can do with them in the next chapters.

---

# Chapter 32. Auto Components

## Introduction

### Note

The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch32\\_auto\\_components](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch32_auto_components)

In O3DE multiplayer components are code generated from their XML definitions, such as `MyFirstNetworkComponent.AutoComplete.xml` in the previous chapter. An auto-component is a code generated multiplayer component. This chapter will cover enough theory of auto-components to get us through the rest of multiplayer chapters.

- How does one create a new multiplayer component?
- What is a Controller?
- What is a (network) Component?
- Where are code generated classes created?
- Overriding base classes of controllers and components.

### Note

In this section of the book, *multiplayer* and *network* terms are used interchangeably.

## Code Generation

### Example 32.1. An auto-component

```
<?xml version="1.0"?>
<Component
    Name="MyFirstNetworkComponent"
    Namespace="MyGem"
    OverrideComponent="false"
    OverrideController="false"
    OverrideInclude=""
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
</Component>
```

Given the above XML definition the following files will be produced for you:

- `MyFirstNetworkComponent.AutoComplete.h`
- `MyFirstNetworkComponent.AutoComplete.cpp`

These files are placed inside the build code-generation section, such as `C:\git\book\build\External\MyGem-409da4e6\Code\Azcg\Generated\Source\AutoGen\MyFirstNetworkComponent.AutoComplete.h`.

### Tip

Your specific build path might be different but if you search within your build folder for these components you will find them.

In general, you should not pay too much attention to these files but you should know that they provide the backbone of multiplayer functionality of your components as well as various Editor and script reflection.

Code generated headers (\*.AutoComponent.h) declare two important classes: a Component and a Controller.

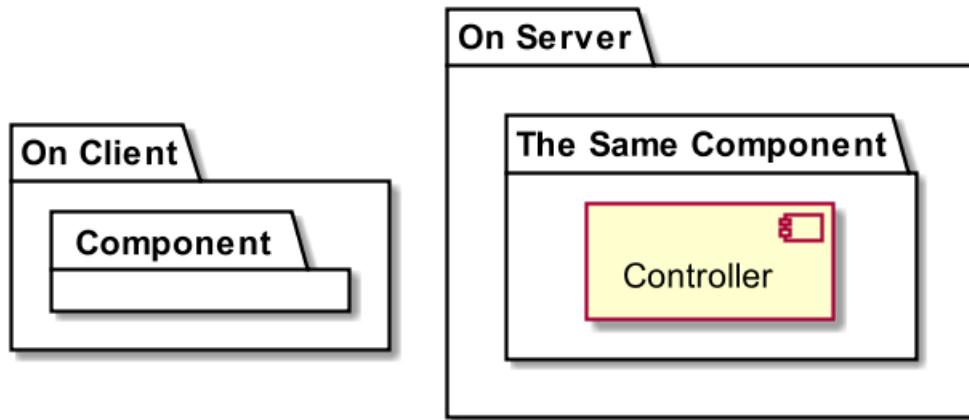
## Components and Controllers

In the context of multiplayer components, their Controllers have write access to component's state, while their Components have read-only access. On clients, a component will be without a controller and thus unable to modify its data directly. Changes can be made on the server where a controller *is* available.

### Note

There is one exception for autonomous entities, such as player entities, that need to be able to locally predict themselves. In such a case, there is a controller on a client as well but only for the player entity that the client controls. I cover this unique case in Chapter 40, *Multiplayer Camera*.

**Figure 32.1. Components and Controllers**



**Where do these controllers and components come from?** They are created by the code generator we have configured in previous chapter. An XML definition is the root source that defines the data and basic structure of a multiplayer component.

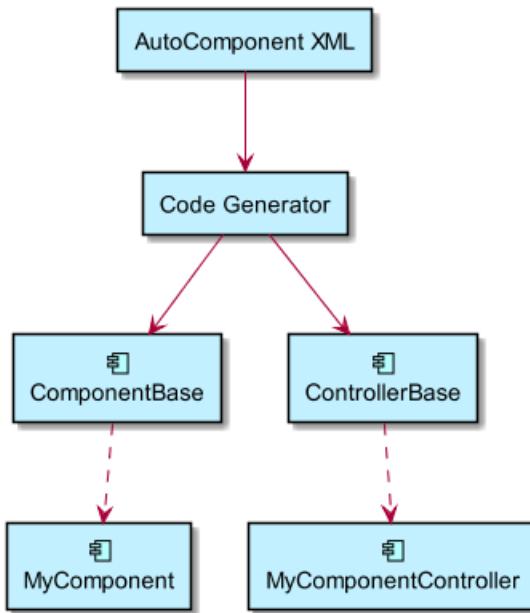
Look at the top of MyFirstNetworkComponent.AutoComponent.h for a useful comment describing the current configuration of the component and your options.

```
// No component roles have been overridden. You can modify
// MyFirstNetworkComponent.AutoComponent.xml to specify derived
// classes for your desired network role.
// Once your modifications are complete, build the game. Your build
// will fail, but this comment will be replaced with a stub that
```

---

```
// can be used as the basis for your derived component roles.
```

**Figure 32.2. Overview of Code Generation of Auto-components**



## Overriding Components and Controllers

As the comment from the code generated header file stated, you can override your component and controller. A good rule of thumb is that if you wish to provide server-specific logic, then you will want to override the controller using the following XML property:

```
OverrideController="true"
```

If you wish to provide client-specific logic, then override the component with:

```
OverrideComponent="true"
```

Here is example of a component that overrides both the component and the controller.

```

<?xml version="1.0"?>
<Component
    Name="MyFirstNetworkComponent"
    Namespace="MyGem"
    OverrideComponent="true"
    OverrideController="true"
    OverrideInclude="Source/MyFirstNetworkComponent.h"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
</Component>
  
```

## Creating Your Own Controllers

Just about any new network component you create will need three (3) files in your project regardless of what code generation magic does in the build folder. Here are these files:

1. XML auto-component definition, MyFirstNetworkComponent.AutoComplete.xml

2. Header file, `MyFirstNetworkComponent.h`.
3. Source file, `MyFirstNetworkComponent.cpp`.

**Here are the steps on writing network component code once you have an XML definition.**

1. Create an XML definition that overrides either the component or the controller or both.

`OverrideController="true"`

2. Specify **OverrideInclude** with the path to the header file where your own game logic will be written. This path is relative to the gem's location at `C:\git\book\Gems\MyGem\Code\Source`.

`OverrideInclude="Multiplayer\MyFirstNetworkComponent.h"`

This will expect to find a file at `C:\git\book\Gems\MyGem\Code\Source\Multiplayer\MyFirstNetworkComponent.h`.

### Note

Adding `Multiplayer` folder is optional. You can create your own folder structure or no folder at all and place your source code directly under `Source`, in which case **OverrideInclude** would be:

`OverrideInclude="MyFirstNetworkComponent.h"`

3. Create an empty `MyFirstNetworkComponent.h` and `MyFirstNetworkComponent.cpp`.
4. Add references to these files to the file list.

### **Example 32.2. `mygem_files.cmake` with an auto-component**

```
set(FILES
...
    # new
    Source/AutoGen/MyFirstNetworkComponent.AutoComponent.xml
    Source/Multiplayer/MyFirstNetworkComponent.h
    Source/Multiplayer/MyFirstNetworkComponent.cpp
)
```

5. Build `MyGem.static` target.

### Note

If neither a component nor a controller was overridden then you can skip the rest of the steps. The component will appear in the Editor but will not be particularly useful, since it will not possess any game logic.

6. You will get a lot of build errors but look for the errors that mention `MyFirstNetworkComponent.AutoComponent.h`. In Visual Studio you can double click on the error and jump to the generated header. Or navigate there by looking up the path in the build log.
7. Open `MyFirstNetworkComponent.AutoComponent.h`.
8. Look for the first big comment block at the top of the header file. The comment block will start with "*You may use the classes below as a basis for your new derived classes.*" The rest of the comment block will provide you with stubs for the header and source file.

9. Copy the code from "/// Place in your .h" to "/// Place in your .cpp" into MyFirstNetworkComponent.h.
10. Copy the rest of the comment to MyFirstNetworkComponent.cpp.
11. Compile the project again. Everything should compile cleanly.

### Example 32.3. Provided Stub of MyFirstNetworkComponent.h

```
#pragma once
#include <Source/AutoGen/MyFirstNetworkComponent.AutoComponent.h>

namespace MyGem
{
    class MyFirstNetworkComponentController
        : public MyFirstNetworkComponentControllerBase
    {
public:
    MyFirstNetworkComponentController(
        MyFirstNetworkComponent& parent);

    void OnActivate(Multiplayer::EntityIsMigrating) override;
    void OnDeactivate(Multiplayer::EntityIsMigrating) override;

protected:
    };
}
```

MyFirstNetworkComponentController is a controller but its interface is very similar to AZ::Component.

- OnActivate plays the same role as Activate does on a regular component.
- OnDeactivate plays the same role as Deactivate does on a regular component.

### Example 32.4. Provided Stub of MyFirstNetworkComponent.cpp

```
#include <Multiplayer/MyFirstNetworkComponent.h>
#include <AzCore/Serialization/SerializeContext.h>

namespace MyGem
{
    MyFirstNetworkComponentController::
        MyFirstNetworkComponentController(MyFirstNetworkComponent& p)
            : MyFirstNetworkComponentControllerBase(p)
    {

void MyFirstNetworkComponentController::OnActivate(
    Multiplayer::EntityIsMigrating)
{
}

void MyFirstNetworkComponentController::OnDeactivate(

```

```
    Multiplayer::EntityIsMigrating)
{
}
}
```

## Note

There is a reference here to `MyFirstNetworkComponent`. This component was generated for us by Multiplayer gem code generator. This component is already reflected and registered with the engine.

Even though the code above does not look like much, it comes with a lot power underneath by deriving from `MyFirstNetworkComponentControllerBase`. For example, the controller can at any time access its component by `GetParent()` or get the entity pointer the controller is on with `GetEntity()`.

### Example 32.5. Accessing the Entity from a Controller

```
void MyFirstNetworkComponentController::OnActivate(
    Multiplayer::EntityIsMigrating)
{
    AZ_Printf(__FUNCTION__, "we are on entity %s",
              GetEntity()->GetName().c_str());
}
```

We will explore multiplayer functionality of these classes when we start writing multiplayer game logic in the next chapters.

## Overriding Components

Using the same steps as with controllers, you can also override the component to perform various logic that does not require write access to the component's data.

### Tip

An example of such logic is receiving a notification when a goal was scored on the server in order to update client's UI we created in Chapter 23, *Interacting with UI in C++*.

1. Modify `MyFirstNetworkComponent.AutoComplete.xml` and enable **OverrideComponent**.
2. Build `MyGem.Static`.
3. There will be build errors, since we did not provide the component in `MyFirstNetworkComponent` but by visiting `MyFirstNetworkComponent.AutoComplete.h` in the build folder, you will find that the stub has been updated to include both the controller and the component for you to start with.
4. Copy over the stubs to `MyFirstNetworkComponent.h` and `MyFirstNetworkComponent.cpp`.
5. Build the project. Everything should compile cleanly now.

### Example 32.6. `MyFirstNetworkComponent.h` with a component

```
#pragma once
```

```
#include <Source/AutoGen/MyFirstNetworkComponent.AutoComponent.h>

namespace MyGem
{
    class MyFirstNetworkComponent
        : public MyFirstNetworkComponentBase
    {
public:
    AZ_MULTIPLAYER_COMPONENT(MyGem::MyFirstNetworkComponent,
        s_myFirstNetworkComponentConcreteUuid,
        MyGem::MyFirstNetworkComponentBase);

    static void Reflect(AZ::ReflectContext* context);

    void OnInit() override;
    void OnActivate(Multiplayer::EntityIsMigrating) override;
    void OnDeactivate(Multiplayer::EntityIsMigrating) override;

protected:
};

...
}
```

My First Network Component is a regular AZ::Component but derived through MyFirstNetworkComponentBase. Much like the controller, OnActivate and OnDeactivate are to be used instead of using regular Activate and Deactivate AZ::Component methods.

### Example 32.7. MyFirstNetworkComponent.cpp with a component

```
#include <Multiplayer/MyFirstNetworkComponent.h>
#include <AzCore/Serialization/SerializeContext.h>

namespace MyGem
{
    void MyFirstNetworkComponent::Reflect(AZ::ReflectContext* rc)
    {
        auto sc = azrtti_cast<AZ::SerializeContext*>(rc);
        if (sc)
        {
            sc->Class<MyFirstNetworkComponent,
                MyFirstNetworkComponentBase>()
                ->Version(1);
        }
        MyFirstNetworkComponentBase::Reflect(rc);
    }

    void MyFirstNetworkComponent::OnInit()
    {
    }

    void MyFirstNetworkComponent::OnActivate(
        Multiplayer::EntityIsMigrating)
    {
```

```
}

void MyFirstNetworkComponent::OnDeactivate(
    Multiplayer::EntityIsMigrating)
{
}
...
}
```

Reflect is a much simpler method in auto-components because all the reflection is done by code generated base classes, which can handle reflecting properties to the Editor and to scripting context.

## Summary

### Note

The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch32\\_auto\\_components](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch32_auto_components)

We have gone over enough aspects of network components to get started with writing our first multiplayer game logic. The important ideas to remember are the following:

- O3DE Multiplayer gem provides a server-authoritative multiplayer system.
- XML definitions are used to generate network component base classes.
- Multiplayer *Controllers* have write access to component data.
- Multiplayer *Components* provide read-only access to component data.

---

# Chapter 33. Multiplayer in the Editor

## Introduction

### Note

The accompanying assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch33\\_multiplayer\\_in\\_editor](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch33_multiplayer_in_editor)

There are two ways to iterate on multiplayer work, either by launching game launchers directly or using the Editor. `MyProject.ServerLauncher` with "`+host +loadlevel MyLevel`" parameters starts the server. `MyProject.GameLauncher` with "`+connect`" parameter starts and connects a client to the local server. Using the Editor is far easier and faster. This chapter will present the way to use the Editor in the most efficient and fastest method to iterate on a multiplayer game in O3DE.

## Editor Server

By default, when you press **CTRL+G** in the Editor multiplayer support is disabled. To enable the multiplayer mode, you have to set a CVar variable `editorsv_enabled` to true. By default, it is set to false, so in order to avoid having to enable it in the Editor console each time you launch the Editor, we can create a console command file that is loaded automatically by the Editor on launch: `C:\git\book\MyProject\editor.cfg`.

### Example 33.1. Turn on CLTR+G multiplayer with `editor.cfg`

```
editorsv_enabled=true  
editorsv_hidden=true  
editorsv_rhi_override=null
```

`editorsv_enabled` enables the editor server, `editorsv_hidden` launches the server without a window in the background and `editorsv_rhi_override` as "null" disables the rendering sub-system. (RHI stands for rendering hardware interface.) Together these are the fastest settings for you to iterate on multiplayer.

### Tip

If you need to see the server window while working with the Editor, then set `editorsv_rhi_override` to either "dx12" or "vulkan", depending on your platform, or leave it blank and have it be picked by the Atom renderer.

### Procedure 33.1. How to run a multiplayer game in the Editor.

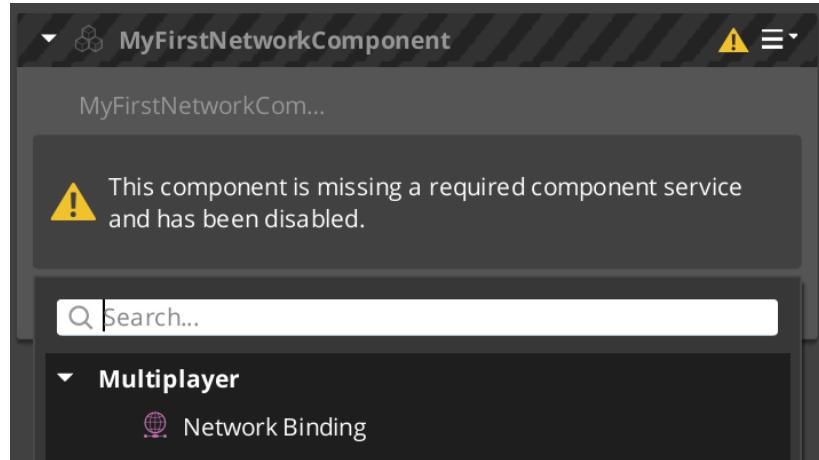
1. Create `editor.cfg` as described above.
2. Launch the Editor.
3. Enter game mode with **CTRL+G**.
4. Wait a few seconds for `MyProject.ServerLauncher` to launch hidden in the background.
5. In a few seconds the Editor will connect as a client to the server in the background.

# Multiplayer Components in the Level

Now that we know how to test multiplayer in the Editor, where the Editor acts a client, we need to know two important points.

## ⚠ Important

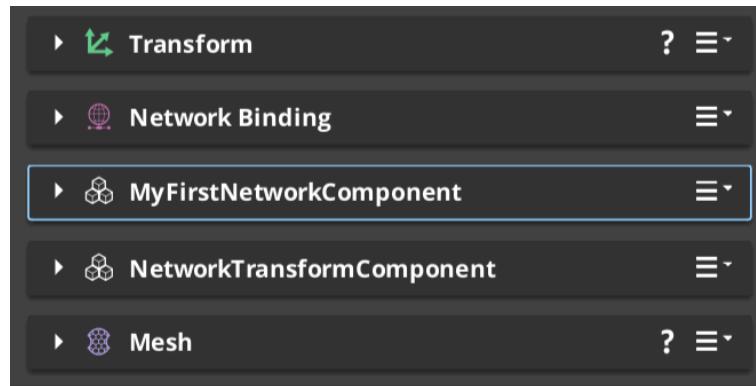
All multiplayer components require Net Binding component.



## ⚠ Important

Multiplayer components must *not* be placed directly into the level. They must be wrapped with a prefab, and then the prefab can be placed into the level.

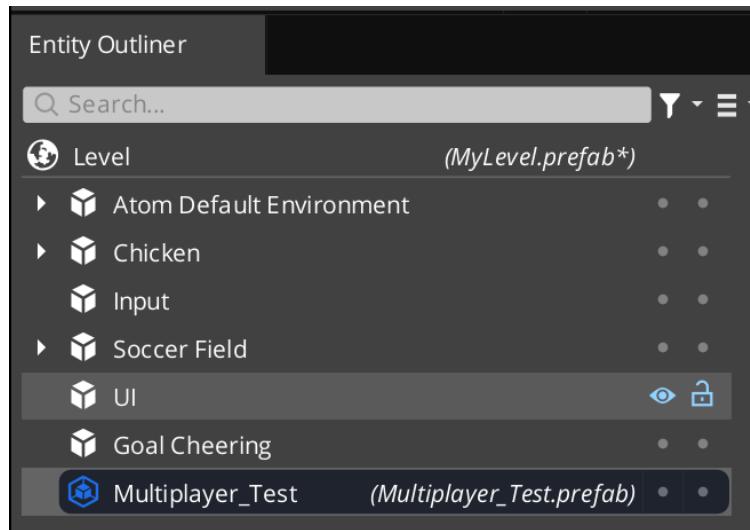
For example, we can create our first multiplayer entity with Transform, Network Binding, My First Network Component, Network Transform Component and a Mesh component.



Then in Entity Outliner create a prefab from this entity.

## ⚠ Important

Without Network Transform Component, the entity will not appear on clients.

**Figure 33.1. A Network Entity Wrapped in a Prefab**

This temporarily adds a white box in the middle of our level (`Multiplayer_Test.prefab`) and when I enter game mode with multiplayer, the white box immediately disappears. This happens because a level prefab is split into two portions: regular and network portion. Regular portion can be spawned immediately but then network portion has to wait until the Editor connects as a client to the server. Then the server will tell the client about all the network entities on the level.

Meanwhile you can confirm that this entity does spawn in the server by looking at the Editor console where it prints the debug lines we added in the previous chapter.

### **Example 33.2. Editor server log**

```
(EditorServer) - MyGem::MyFirstNetworkComponent::OnActivate:
    we are on entity Multiplayer Test

MyGem::MyFirstNetworkComponentController::OnActivate:
    we are on entity Multiplayer Test
```

## **Summary**

### **Note**

The accompanying assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch33\\_multiplayer\\_in\\_editor](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch33_multiplayer_in_editor)

In this chapter we learned how to play a multiplayer game in the Editor and how to place multiplayer entities in the level using prefabs. However, at the moment, none of multiplayer entities appear when entering game mode because our client setup is missing a critical piece without which a network connection between the server and the Editor cannot be established.

**We are missing a player object.** Once we tell the multiplayer system which entity is our player entity, the connection will be properly established and the development can begin. The next chapter will tackle creation of the simplest player spawner that I could come up with.

# Chapter 34. Simple Player Spawner

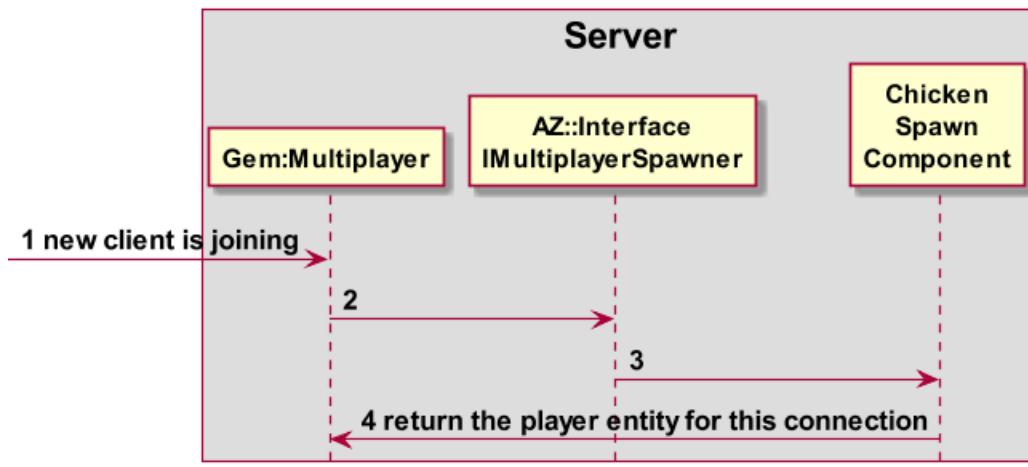
## Introduction

### Note

The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch34\\_player\\_spawner](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch34_player_spawner)

Gem Multiplayer expects us to assign a player entity for each incoming connection on the server.



IMultiplayerSpawner is an abstract interface defined at C:\git\o3de\Gems\Multiplayer\Code\Include\Multiplayer\IMultiplayerSpawner.h. It is up to us to implement it and supply it as an AZ::Interface. Its essential method is OnPlayerJoin:

```
Multiplayer::NetworkEntityHandle OnPlayerJoin(  
    uint64_t userId,  
    const Multiplayer::MultiplayerAgentDatum& agentDatum)
```

OnPlayerJoin expects back a NetworkEntityHandle which is a reference handle to a network entity. You can create a NetworkEntityHandle by passing a pointer to an AZ::Entity that has a Network Binding component.

```
NetworkEntityHandle(AZ::Entity*)
```

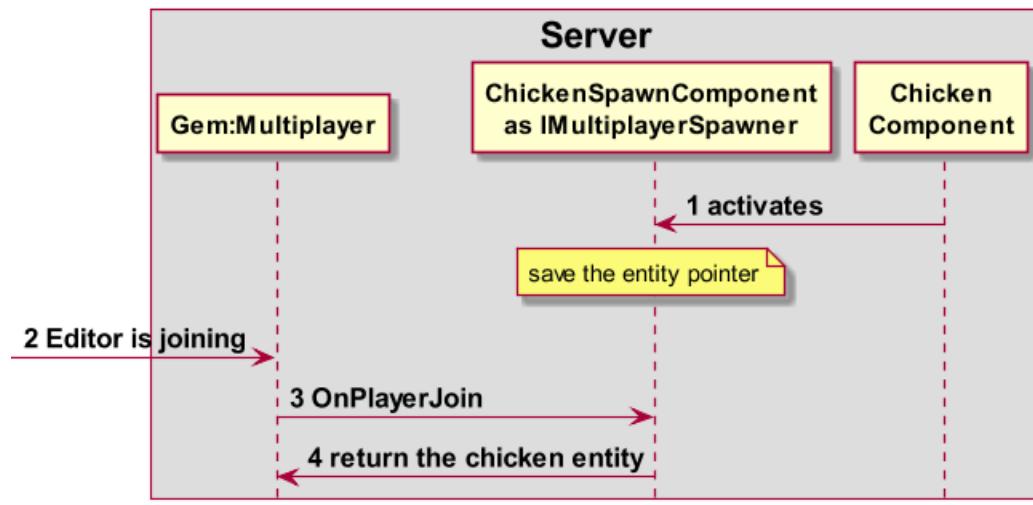
### Tip

Even though the name of the interface is a multiplayer spawner there is nothing in its API that forces our design choice of how to create our player entities. We can certainly spawn on demand, however, in this chapter I am going to keep it simple by pre-creating the player entity as a prefab on the level and when the Editor joins as a client, I will grab that prefab and return the entity from it.

## Design

I am going to create two new components: Chicken Spawn Component and Chicken Component. Chicken Spawn Component will be responsible for inheriting from IMultiplayerSpawner and implementing On-

PlayerJoin. Chicken Component will be added to the main chicken entity and will report itself to Chicken Spawn Component on activation on the server. Chicken Spawn Component will then assign this entity to the player connection.



## Chicken Notification Bus

Chicken component and Chicken Spawner component will communicate using a notification bus, ChickenNotificationBus.

### Example 34.1. ChickenBus.h

```

#pragma once
#include <AzCore/Component/ComponentBus.h>

namespace MyGem
{
    class ChickenNotifications
        : public AZ::ComponentBus
    {
    public:
        ...
        virtual void OnChickenCreated(
            [[maybe_unused]] AZ::Entity* e) {}
    };

    using ChickenNotificationBus = AZ::EBus<ChickenNotifications>;
}
  
```

When a chicken's prefab is spawned in the level, the chicken's entity will call `OnChickenCreated` that Chicken Spawner component will be expecting.

## Chicken Component

This is going to be our first multiplayer controller implementation. We want the chicken component to register itself with Chicken Spawn Component only on the server as clients are not involved in player entity creation. We are server-authoritative after all!

The first step is to create the XML definition of an auto-component that overrides the controller.

### Example 34.2. First ChickenComponent.AutoComponent.xml

```
<?xml version="1.0"?>
<Component
    Name="ChickenComponent"
    Namespace="MyGem"
    OverrideComponent="false"
    OverrideController="true"
    OverrideInclude="Source/Multiplayer/ChickenComponent.h"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
</Component>
```

Next we follow the steps from Chapter 32, *Auto Components* to create `ChickenComponent.h` and `ChickenComponent.cpp`.

### Example 34.3. ChickenComponent.h

```
#pragma once
#include <Source/AutoGen/ChickenComponent.AutoComponent.h>

namespace MyGem
{
    class ChickenComponentController
        : public ChickenComponentControllerBase
    {
public:
    ChickenComponentController(ChickenComponent& parent);

    void OnActivate(Multiplayer::EntityIsMigrating) override;
    void OnDeactivate(Multiplayer::EntityIsMigrating) override {}
};

}
```

### Example 34.4. ChickenComponent.cpp

```
#include <Multiplayer/ChickenComponent.h>
#include <MyGem/ChickenBus.h>

namespace MyGem
{
    ChickenComponentController::
        ChickenComponentController(ChickenComponent& p)
        : ChickenComponentControllerBase(p) {}

    void ChickenComponentController::OnActivate(
        Multiplayer::EntityIsMigrating)
    {
        if (!IsAuthority()) return;

        ChickenNotificationBus::Broadcast(
            &ChickenNotificationBus::Events::OnChickenCreated,
            GetEntity());
    }
}
```

```
    }  
}
```

**What is IsAuthority?** This is a method from the base class `MultiplayerController` that is only true when the controller is on the authoritative server.

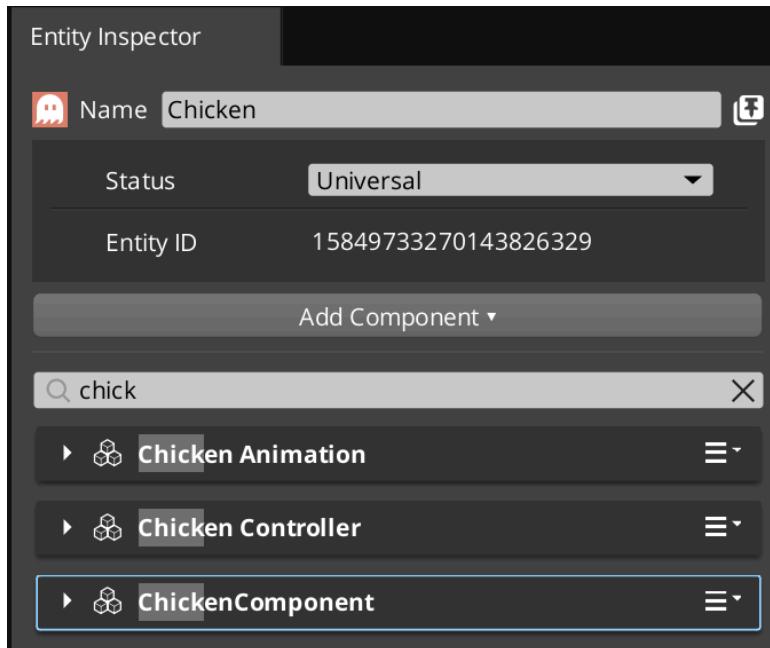
```
///! Returns true if this controller has authority.  
///! @return boolean true if this controller has authority  
bool IsAuthority() const;
```

If the controller has write access to the component, then it is considered to have an authority over it. As a quick side, there are only two options for a controller. A controller can either have an authority, if it is on the server, or it can be an autonomous controller on the client that controls the chicken and can locally predict the movement of the chicken. Otherwise, no controller would be created for a component. We will tackle autonomous controllers when we implement multiplayer chicken movement.

## Tip

As you can see writing network components takes a lot less lines of code than a fully written `AZ::Component`. A lot of boilerplate is handled by the code generator.

Chicken component goes on the root entity of the chicken structure we have in the level.



## Chicken Spawn Component

Now that we have a chicken entity reporting its presence upon activation on the server, we need `Chicken Spawn Component` to receive it.

### Example 34.5. Snippet from `ChickenSpawnComponent.h`

```
class ChickenSpawnComponent
```

```
: public AZ::Component
, public Multiplayer::IMultiplayerSpawner
, public ChickenNotificationBus::Handler
{
public:
...
    // ChickenNotificationBus
    void OnChickenCreated(AZ::Entity* e) override;

    // IMultiplayerSpawner overrides
    Multiplayer::NetworkEntityHandle OnPlayerJoin(
        uint64_t userId,
        const Multiplayer::MultiplayerAgentDatum&) override;
...
private:
    AZ::Entity* m_chicken = nullptr;
```

ChickenSpawnComponent inherits from IMultiplayerSpawner and implements OnPlayerJoin. OnChickenCreated saves the entity pointer and returns it via OnPlayerJoin when a client joins.

#### Example 34.6. Snippet from `ChickenSpawnComponent.cpp`

```
void ChickenSpawnComponent::OnChickenCreated(AZ::Entity* e)
{
    m_chicken = e;
}

NetworkEntityHandle ChickenSpawnComponent::OnPlayerJoin(
    [[maybe_unused]] uint64_t userId,
    const Multiplayer::MultiplayerAgentDatum&)
{
    return NetworkEntityHandle{ m_chicken };
}
```

Chicken Spawn component is not a multiplayer component, so it can be placed directly on the level, for example under a new entity, **Chicken Spawn**.

## Creating Player Prefab

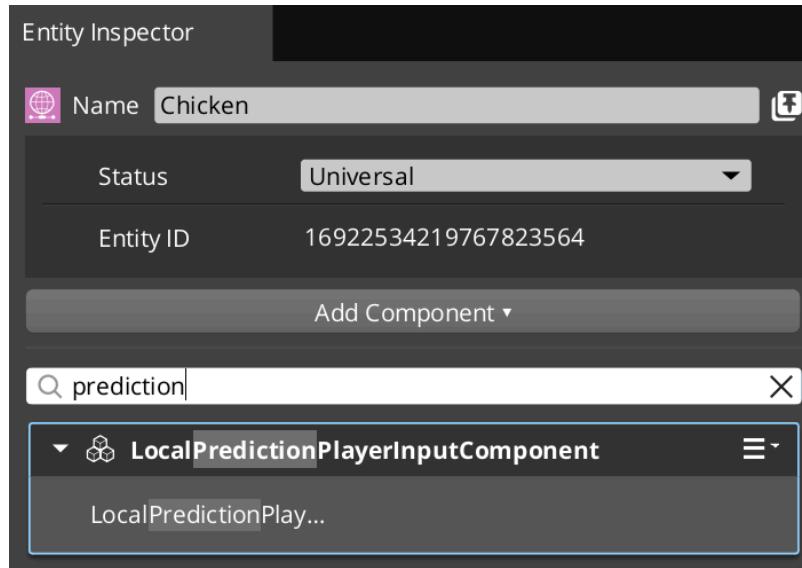
A player entity has to be a network entity, thus it goes inside a prefab, `Chicken.prefab`. Here are the steps to convert our current chicken into a multiplayer prefab.

1. For each chicken entity, add Net Binding and Network Transform components.

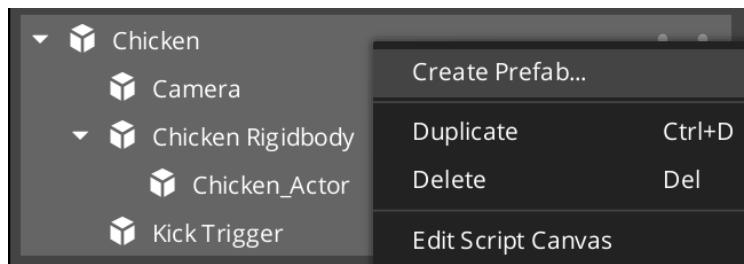
### Important

If you do not add both of these components, then those entities will not show up on clients at all. Net Binding component marks the entity as a network entity. Network Transform component specifies where to spawn the entity on clients and will update client position if the server moves the entity.

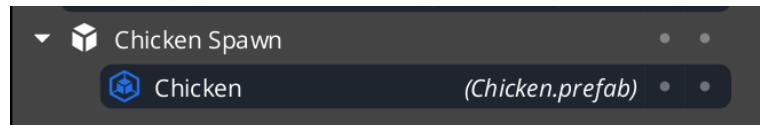
2. Add Local Prediction Player Input Component to the parent entity, **Chicken**. In the next chapter, it will be used to process player's input.



3. Select all the chicken entities and create a prefab out of them.



4. For better organization, I moved the prefab under Chicken Spawn entity.



## Summary

### Note

The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch34\\_player\\_spawner](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch34_player_spawner)

With these changes, when you press **CTRL+G**, you will finally see your camera follow a chicken. Also you will see multiplayer entities show up when the Editor connects to the server.

### Note

It takes a few seconds for the server to start and the Editor to connect. If you can see your camera switch to follow a chicken, then everything is working as intended. Server waits for a client to join and then assigns it an entity to control that is specified by Chicken Spawn component.

## ⚠ Important

The implementation in this chapter is very limited. It supports only one player on the server. In Chapter 39, *Team Spawner*, we will enhance Chicken Spawn Component to support multiple clients and have them join on different teams.

### Example 34.7. Full code for `ChickenSpawnComponent.h`

```
#pragma once
#include <AzCore/Component/Component.h>
#include <Multiplayer/IMultiplayerSpawner.h>
#include <MyGem/ChickenBus.h>

namespace MyGem
{
    class ChickenSpawnComponent
        : public AZ::Component
        , public Multiplayer::IMultiplayerSpawner
        , public ChickenNotificationBus::Handler
    {
public:
    AZ_COMPONENT(ChickenSpawnComponent,
                 "{814BAF21-10E4-4BE9-8380-C23B0EC27205}");

    static void Reflect(AZ::ReflectContext* rc);

    // AZ::Component interface implementation
    void Activate() override;
    void Deactivate() override;

    // ChickenNotificationBus
    void OnChickenCreated(AZ::Entity* e) override;

    // IMultiplayerSpawner overrides
    Multiplayer::NetworkEntityHandle OnPlayerJoin(
        uint64_t userId,
        const Multiplayer::MultiplayerAgentDatum&) override;
    void OnPlayerLeave(
        Multiplayer::ConstNetworkEntityHandle entityHandle,
        const Multiplayer::ReplicationSet& replicationSet,
        AzNetworking::DisconnectReason reason) override {}

private:
    AZ::Entity* m_chicken = nullptr;
};

} // namespace MyGem
```

### Example 34.8. Full code for `ChickenSpawnComponent.cpp`

```
#include <ChickenSpawnComponent.h>
#include <AzCore/Component/Entity.h>
#include <AzCore/Interface/Interface.h>
#include <AzCore/Serialization/EditContext.h>
#include <Multiplayer/IMultiplayer.h>
```

```
namespace MyGem
{
    using namespace Multiplayer;

    void ChickenSpawnComponent::Reflect(AZ::ReflectContext* rc)
    {
        if (auto sc = azrtti_cast<AZ::SerializeContext*>(rc))
        {
            sc->Class<ChickenSpawnComponent, AZ::Component>()
                ->Version(1);

            if (AZ::EditContext* ec = sc->GetEditContext())
            {
                using namespace AZ::Edit;
                ec->Class<ChickenSpawnComponent>(
                    "Chicken Spawn",
                    "[Player controlled chickens]")
                    ->ClassElement(ClassElements::EditorData, "")
                    ->Attribute(
                        Attributes::AppearsInAddComponentMenu,
                        AZ_CRC_CE("Game"));
            }
        }
    }

    void ChickenSpawnComponent::Activate()
    {
        AZ::Interface<IMultiplayerSpawner>::Register(this);
        ChickenNotificationBus::Handler::BusConnect(GetEntityId());
    }

    void ChickenSpawnComponent::Deactivate()
    {
        ChickenNotificationBus::Handler::BusDisconnect();
        AZ::Interface<IMultiplayerSpawner>::Unregister(this);
    }

    void ChickenSpawnComponent::OnChickenCreated(AZ::Entity* e)
    {
        m_chicken = e;
    }

    NetworkEntityHandle ChickenSpawnComponent::OnPlayerJoin(
        [[maybe_unused]] uint64_t userId,
        const Multiplayer::MultiplayerAgentDatum&)
    {
        return NetworkEntityHandle{ m_chicken };
    }
} // namespace MyGem
```

---

# Chapter 35. Multiplayer Input Controls

## Introduction

### Note

The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch35\\_multiplayer\\_input](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch35_multiplayer_input)

It may appear that our chicken is already moving on its own. After all, it does move in the Editor. However, if you were to launch a standalone server, you would notice that the chicken entity is only moving on the client. We are still using single player input component that is directly controlling the position of the entity. So far the server has no idea that you are issuing input.

### Warning

Out of all multiplayer topics, this chapter will be the most complicated as it involves re-writing the old single player input logic into a system that supports local prediction, correction and server side roll back.

The good news is that the actual logic is very similar to Chapter 18, *Character Movement*. It just needs to be placed under different methods. As I go through in this chapter, I will present single player and multiplayer implementations side by side.

## Chicken Movement Component

This chapter will create a new component, `ChickenMovementComponent`, to replace `ChickenControllerComponent`. It will be a multiplayer component that can capture and process input in such a manner that Multiplayer gem can locally predict the movement of the chicken without us doing any extra work.

As it is with multiplayer components, we will need three (3) new files.

### Example 35.1. Additions to `Gems\MyGem\Code\mygem_files.cmake`

```
set(FILES
...
# new
Source/AutoGen/ChickenMovementComponent.AutoComponent.xml
Source/Multiplayer/ChickenMovementComponent.h
Source/Multiplayer/ChickenMovementComponent.cpp
)
```

## XML Definition

XML definition of `ChickenMovementComponent` will have a number of new elements. This section will cover each new type:

- Component Relation

- Archetype Property
- Network Input

## Component Relation

A multiplayer controller can declare a dependency on another component's controller on the same entity. This will code generate a getter method. For example, we will need to use the controller of Network Character component in order to move our multiplayer chicken. The API for that is **NetworkCharacterComponentController::TryMoveWithVelocity**. Instead of trying to get the entity, then Network Character component and then fight the API to get to controller of another entity, we will declare a *Component Relation*.

```
<ComponentRelation Constraint="Required" HasController="true"
    Name="NetworkCharacterComponent" Namespace="Multiplayer"
    Include="Multiplayer/Components/NetworkCharacterComponent.h"/>
```

That will allow us to call **TryMoveWithVelocity** from Chicken Movement Component Controller directly.

```
GetNetworkCharacterComponentController() ->
    TryMoveWithVelocity(m_velocity, deltaTime);
```

### Note

This is similar to providing `GetRequiredServices()` method on an `AZ::Component` but with an extra benefit of receiving a helpful getter method.

## Archetype Property

In Chapter 18, *Character Movement*, `ChickenControllerComponent` exposed Speed, Turn Speed and Gravity settings to the Editor. We had to reflect it ourselves.

```
class ChickenControllerComponent
{
    float m_speed = 6.f;
    float m_turnSpeed = 0.005.f;
    float m_gravity = -9.8f;
    //...
    ->DataElement(nullptr,
        &ChickenControllerComponent::m_turnSpeed,
        "Turn Speed", "Chicken's turning speed")
    /// and so on
```

With **Archetype Property** we can save the effort and use code generator instead.

```
<ArchetypeProperty Type="float" Name="WalkSpeed"
    Init="6.0f" ExposeToEditor="true" />
<ArchetypeProperty Type="float" Name="TurnSpeed"
    Init="0.005f" ExposeToEditor="true" />
<ArchetypeProperty Type="float" Name="Gravity"
    Init="-9.8f" ExposeToEditor="true" />
```

**Archetype Property** will do all the work of reflecting the details to the Editor for us.

## Network Input

In Chapter 18, *Character Movement*, `ChickenControllerComponent` was responsible for processing player input and was storing it in `ChickenInput` structure.

```
class ChickenInput
{
public:
    float m_forwardAxis = 0;
    float m_strafeAxis = 0;
    float m_viewYaw = 0;
};
```

With multiplayer input, we declare these as `NetworkInput`'s.

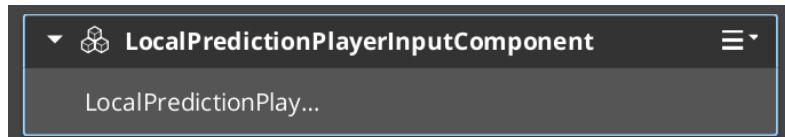
```
<NetworkInput Type="float" Name="ForwardAxis" Init="0.0f" />
<NetworkInput Type="float" Name="StrafeAxis" Init="0.0f" />
<NetworkInput Type="float" Name="ViewYaw" Init="0.0f" />
<NetworkInput Type="uint8_t" Name="ResetCount" Init="0" />
```

### Note

**Reset Count** is a special variable that is incremented any time the entity moves in an usual way, such as teleporting to a new location or stepping on a launch pad. Without this counter, local prediction will have issues with sudden movement and state changes.

### Important

Multiplayer components with a Network Input require Local Prediction Player Input component on the entity.



## Create Input

When you add a Network Input to a multiplayer component, code generator will add two (2) new methods for you to implement.

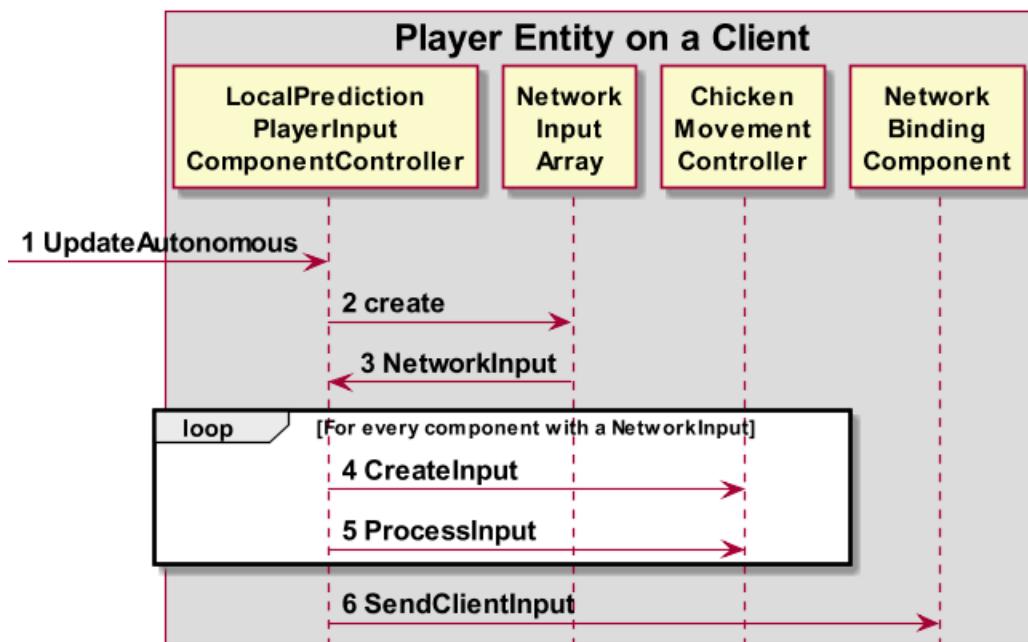
```
///! Common input creation logic for the NetworkInput.
///! Fill out the input struct and the MultiplayerInputDriver
///! will send the input data over the network to ensure
///! it's processed.
///! @param input input structure which to store input data
///!             for sending to the authority
///! @param deltaTime amount of time to integrate
///!                   the provided inputs over
void CreateInput(
    Multiplayer::NetworkInput& input,
    float deltaTime) override;
```

```

//! Common input processing logic for the NetworkInput.
//! @param input input structure to process
//! @param deltaTime amount of time to integrate the
//! provided inputs over
void ProcessInput(
    Multiplayer::NetworkInput& input,
    float deltaTime) override;
```

Creation of input occurs on clients on player owned entities, otherwise known as *Autonomous* entities. Their task is to collect player's input and send it to the server.

**Figure 35.1. Sending Player Input to the Server**



1. `UpdateAutonomous` gets called at a specified client input rate. You can control this rate with a Multiplayer gem variable, `cl_InputRateMs`. The default is to collect input every 33 milliseconds.

```

AZ_CVAR(AZ::TimeMs, cl_InputRateMs, AZ::TimeMs{ 33 }, nullptr,
        AZ::ConsoleFunctorFlags::Null,
        "Rate at which to sample and process client inputs");
```

2. `NetworkInputArray` is a container with the current input and previous seven (7) inputs. Our job is to collect the current input values. Local Prediction Player Input component will populate older entries on our behalf from the local input history.

**Why are we sending seven (7) older inputs each time?** This is because input is sent using an unreliable remote procedure call. Here is a snippet from XML definition of `LocalPredictionPlayerInputComponent` from Multiplayer gem, where `IsReliable` is set to false.

### Example 35.2. LocalPredictionPlayerInputComponent.AutoComplete.xml

```

<RemoteProcedure Name="SendClientInput" InvokeFrom="Autonomous"
    HandleOn="Authority" IsPublic="true" IsReliable="false"
    GenerateEventBindings="false"
```

```
Description="Client to server move / input RPC">

<Param Type="Multiplayer::NetworkInputArray" Name="inputArray" />
<Param Type="AZ::HashValue32" Name="stateHash" />
</RemoteProcedure>
```

Multiplayer replication uses UDP<sup>1</sup> protocol, which is unreliable by design to achieve faster delivery and reduce stalls. Multiplayer gem overcomes this by sending multiple inputs. This means we can lose up to seven (7) inputs and still be able to recover. Given client input rate of 33 milliseconds that allows for almost a quarter of a second recovery time (231 milliseconds) before the server fails to receive all input. This is sufficient for majority of real world network scenarios.

3. From Network Input Array, the multiplayer system will grab the first item (at zeroth index) as NetworkInput and pass it to all the relevant multiplayer components.
4. CreateInput's job is to collect the input for the last input time period.
5. ProcessInput is applied on the client immediately. This is the local prediction step, where the client guesses where the input will take it. If the server disagrees, the client will be corrected later.
6. The input is then sent to the server.

## ❗ Important

This was a behind-the-scenes look at the input processing logic. Users only need to worry about implementing CreateInput and ProcessInput methods. The rest will be done by the Multiplayer gem.

Here is side by side comparison between single player create input and multiplayer create input methods.

### Example 35.3. Single-player create input logic

```
ChickenInput ChickenControllerComponent::CreateInput()
{
    ChickenInput input;
    input.m_forwardAxis = m_forward;
    input.m_strafeAxis = m_strafe;
    input.m_viewYaw = m_yaw;

    return input;
}
```

And here is the multiplayer one.

### Example 35.4. Multiplayer create input logic

```
void ChickenMovementComponentController::CreateInput(
    Multiplayer::NetworkInput& input,
    [[maybe_unused]] float deltaTime)
{
    auto chickenInput = input.FindComponentInput<
        ChickenMovementComponentNetworkInput>();
```

---

<sup>1</sup>UDP stands for User Datagram Protocol. It is an unreliable form of network protocol, as opposed to TCP protocol. See [https://en.wikipedia.org/wiki/User\\_Datagram\\_Protocol](https://en.wikipedia.org/wiki/User_Datagram_Protocol) for an introduction.

```

chickenInput->m_forwardAxis = m_forward;
chickenInput->m_strafeAxis = m_strafe;
chickenInput->m_viewYaw = m_yaw;

chickenInput->m_resetCount =
    GetNetworkTransformComponentController()->GetResetCount();
}

```

**ChickenMovementComponentNetworkInput** was generated for us based on the XML definition that listed all the network inputs.

### Example 35.5. Snippet from ChickenMovementComponent.AutoComplete.h

```

class ChickenMovementComponentNetworkInput
    : public Multiplayer::IMultiplayerComponentInput
{
public:
...
    float m_forwardAxis = float(0.0f);
    float m_strafeAxis = float(0.0f);
    float m_viewYaw = float(0.0f);
    uint8_t m_resetCount = uint8_t(0);
};

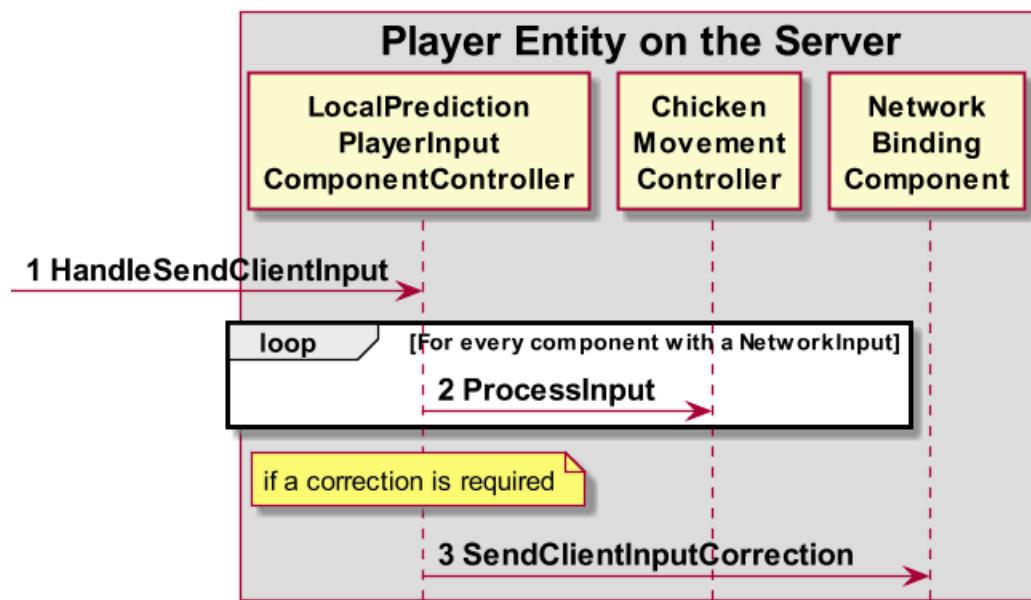
```

Overall, the only difference from the single-player input is that we have a special input reset counter to handle special movement cases but otherwise they are same.

## Process Input

Once the input arrives to the server, it will be applied using logic written in `ProcessInput`.

**Figure 35.2. Processing Input on the Server**



1. HandleSendClientInput is the handle for the remote procedure.
2. There is no creation of an input, since this is happening on the server. ProcessInput gets called for each relevant multiplayer component that creates input on clients.
3. If a mismatch is found between client and server results, then a correction is sent back to the client using an unreliable remote procedure from Local Prediction Player Input component.

```
<RemoteProcedure Name="SendClientInputCorrection"
    InvokeFrom="Authority" HandleOn="Autonomous" IsPublic="true"
    IsReliable="false" GenerateEventBindings="false"
    Description="Autonomous proxy correction RPC">
    <Param Type="Multiplayer::ClientInputId" Name="inputId" />
    <Param Type="AzNetworking::PacketEncodingBuffer"
        Name="correction" />
</RemoteProcedure>
```

## ⚠ Important

The above work is done for us by Multiplayer gem. As users, we implement ProcessInput and let the multiplayer system handle local prediction and correction.

For comparison, here are the single-player and multiplayer methods that process player's input.

### Example 35.6. Single-player Process Input

```
void ChickenControllerComponent::ProcessInput(
    const ChickenInput& input)
{
    UpdateRotation(input);
    UpdateVelocity(input);

    Physics::CharacterRequestBus::Event(GetEntityId(),
        &Physics::CharacterRequestBus::Events::AddVelocity,
        m_velocity);
    Physics::CharacterRequestBus::Event(GetEntityId(),
        &Physics::CharacterRequestBus::Events::AddVelocity,
        AZ::Vector3::CreateAxisZ(m_gravity));
}
```

### Example 35.7. Multiplayer Process Input

```
void ChickenMovementComponentController::ProcessInput(
    Multiplayer::NetworkInput& input,
    [[maybe_unused]] float deltaTime)
{
    auto chickenInput = input.FindComponentInput<
        ChickenMovementComponentNetworkInput>();
    if (chickenInput->m_resetCount != GetNetworkTransformComponentController()->GetResetCount())
    {
        return;
    }

    UpdateRotation(chickenInput);
```

```
    UpdateVelocity(chickenInput);

    GetNetworkCharacterComponentController()->
        TryMoveWithVelocity(m_velocity, deltaTime);
}
```

UpdateRotation and UpdateVelocity methods did not change. We use different ways to apply velocity at the end of each method but otherwise the logic is essentially the same.

### Note

This is where **Reset Count** property comes into play. The idea here is as follows. If the movement count was reset, it means we should not try to locally predict based on input with an old count.

Imagine our character stepped on a jump pad and was now sailing through the air. We do not want our local prediction to make a mistake of not stepping on the launch pad. It is a type of mistake we cannot afford with local prediction.

```
if (chickenInput->m_resetCount !=  
    GetNetworkTransformComponentController()->GetResetCount())  
{  
    return;  
}
```

This logic will skip over input that is too old and should not be re-applied.

CreateInput collects input into a structure and ProcessInput applies it. Behind the scenes, the input is stored and sent from the client to the server, where the input is processed using the same logic of Process Input. In this manner, both the client and the server should arrive to the same result. If there are any differences, the server will send a correction to the client and the input will be re-applied using ProcessInput.

It is almost magical how local prediction of player's input is handled for us by the Multiplayer gem via the generated code.

## What is Find Component Input?

You may have noticed that we get our input data class in an usual way. It was acquired by using method FindComponentInput of NetworkInput.

```
void ChickenMovementComponentController::ProcessInput(  
    Multiplayer::NetworkInput& input, float)  
{  
    auto chickenInput = input.FindComponentInput<  
        ChickenMovementComponentNetworkInput>();
```

Behind the scenes, any network entity that has components with network inputs gets an associated Multiplayer::NetworkInput container of component input structures, one for each such multiplayer component.

These input structures are generated from XML definition of each multiplayer component as needed. ChickenMovementComponentNetworkInput can be found at ChickenMovementComponent.AutoComponent.h.

```
class ChickenMovementComponentNetworkInput  
: public Multiplayer::IMultiplayerComponentInput
```

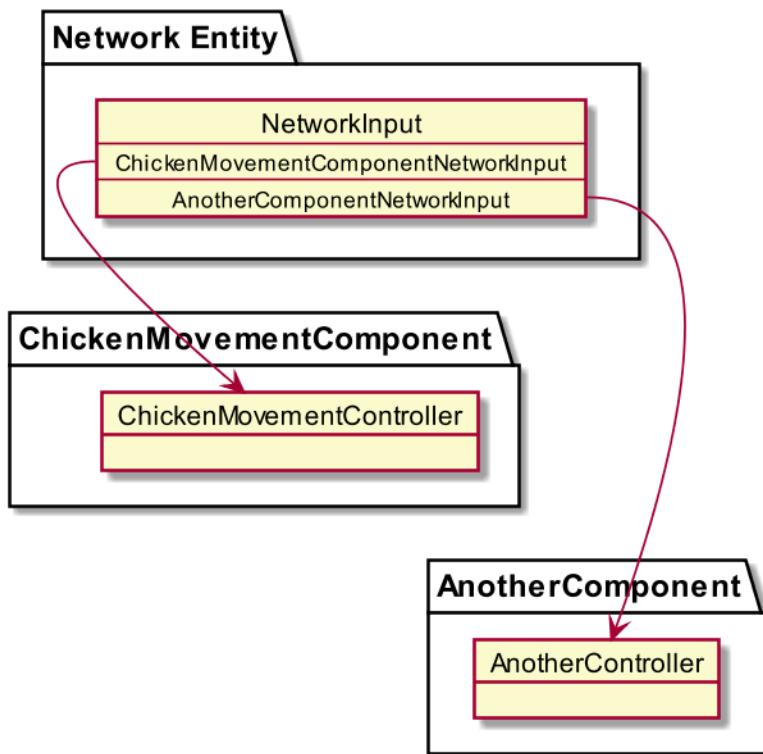
```

{
public:
    bool Serialize(AzNetworking::ISerializer& serializer);

    float m_forwardAxis = float(0.0f);
    float m_strafeAxis = float(0.0f);
    float m_viewYaw = float(0.0f);
    uint8_t m_resetCount = uint8_t(0);

    //...
};


```

**Figure 35.3. Network Inputs**

Network Binding and Local Prediction Player Input components work together to create a network input structure for each of multiplayer component with inputs. These structures are then provided for you in `NetworkInput` that you can query with `FindComponentInput`.

## Summary

### Note

The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch35\\_multiplayer\\_input](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch35_multiplayer_input)

The rest of the component does exactly the same work as `ChickenControllerComponent` did in Chapter 18, *Character Movement*, which is to sign up for events on `InputEventNotificationBus` to record key presses and mouse movements.

## ⚠ Important

You need to modify Chicken entity to remove the old Chicken Controller component and replace it with Chicken Movement component.

You should verify that the chicken does move with the server in the same way it moves on the client. Here are the commands to launch the server and a client.

```
MyProject.ServerLauncher.exe +host +loadlevel mylevel  
MyProject.GameLauncher.exe +connect
```

### Example 35.8. `ChickenMovementComponent.h`

```
#pragma once  
  
#include <Source/AutoGen/ChickenMovementComponent.AutoComponent.h>  
#include <StartingPointInput/InputEventNotificationBus.h>  
  
namespace MyGem  
{  
    const StartingPointInput::InputEventNotificationId  
        MoveFwdEventId("move forward");  
    const StartingPointInput::InputEventNotificationId  
        MoveRightEventId("move right");  
    const StartingPointInput::InputEventNotificationId  
        RotateYawEventId("rotate yaw");  
  
    class ChickenMovementComponentController  
        : public ChickenMovementComponentControllerBase  
        , public StartingPointInput::  
            InputEventNotificationBus::MultiHandler  
    {  
        public:  
            ChickenMovementComponentController(  
                ChickenMovementComponent& parent);  
  
            void OnActivate(Multiplayer::EntityIsMigrating) override;  
            void OnDeactivate(Multiplayer::EntityIsMigrating) override;  
  
            ///! Common input creation logic for the NetworkInput.  
            ///! Fill out the input struct and the MultiplayerInputDriver  
            ///! will send the input data over the network to ensure  
            ///! it's processed.  
            ///! @param input input structure which to store input data  
            ///!                 for sending to the authority  
            ///! @param deltaTime amount of time to integrate  
            ///!                 the provided inputs over  
            void CreateInput(  
                Multiplayer::NetworkInput& input,  
                float deltaTime) override;  
  
            ///! Common input processing logic for the NetworkInput.  
            ///! @param input input structure to process  
            ///! @param deltaTime amount of time to integrate the  
            ///!                 provided inputs over
```

```
    void ProcessInput(
        Multiplayer::NetworkInput& input,
        float deltaTime) override;

    // AZ::InputEventNotificationBus interface
    void OnPressed(float value) override;
    void OnReleased(float value) override;
    void OnHeld(float value) override;

protected:
    void UpdateRotation(
        const ChickenMovementComponentNetworkInput* input);
    void UpdateVelocity(
        const ChickenMovementComponentNetworkInput* input);

    float m_forward = 0;
    float m_strafe = 0;
    float m_yaw = 0;

    AZ::Vector3 m_velocity = AZ::Vector3::CreateZero();
};

} // namespace MyGem
```

### Example 35.9. ChickenMovementComponent.cpp

```
#include <AzCore/Component/Entity.h>
#include <AzCore/Component/TransformBus.h>
#include <Multiplayer/ChickenMovementComponent.h>
#include <Multiplayer/Components/NetworkCharacterComponent.h>
#include <Multiplayer/Components/NetworkTransformComponent.h>

namespace MyGem
{
    using namespace StartingPointInput;

    ChickenMovementComponentController::
        ChickenMovementComponentController(ChickenMovementComponent& p)
        : ChickenMovementComponentControllerBase(p) {}

    void ChickenMovementComponentController::OnActivate(
        Multiplayer::EntityIsMigrating)
    {
        InputEventNotificationBus::MultiHandler::BusConnect(
            MoveFwdEventId);
        InputEventNotificationBus::MultiHandler::BusConnect(
            MoveRightEventId);
        InputEventNotificationBus::MultiHandler::BusConnect(
            RotateYawEventId);
    }

    void ChickenMovementComponentController::OnDeactivate(
        Multiplayer::EntityIsMigrating)
    {
        InputEventNotificationBus::MultiHandler::BusDisconnect();
    }
}
```

```
}

void ChickenMovementComponentController::CreateInput(
    Multiplayer::NetworkInput& input,
    [[maybe_unused]] float deltaTime)
{
    auto chickenInput = input.FindComponentInput<
        ChickenMovementComponentNetworkInput>();

    chickenInput->m_forwardAxis = m_forward;
    chickenInput->m_strafeAxis = m_strafe;
    chickenInput->m_viewYaw = m_yaw;

    chickenInput->m_resetCount =
        GetNetworkTransformComponentController()->GetResetCount();
}

void ChickenMovementComponentController::ProcessInput(
    Multiplayer::NetworkInput& input,
    [[maybe_unused]] float deltaTime)
{
    auto chickenInput = input.FindComponentInput<
        ChickenMovementComponentNetworkInput>();
    if (chickenInput->m_resetCount !=
        GetNetworkTransformComponentController()->GetResetCount())
    {
        return;
    }

    UpdateRotation(chickenInput);
    UpdateVelocity(chickenInput);

    GetNetworkCharacterComponentController()->
        TryMoveWithVelocity(m_velocity, deltaTime);
}

void ChickenMovementComponentController::OnPressed(float value)
{
    const InputEventNotificationId* inputId =
        InputEventNotificationBus::GetCurrentBusId();
    if (inputId == nullptr)
    {
        return;
    }

    if (*inputId == MoveFwdEventId)
    {
        m_forward = value;
    }
    else if (*inputId == MoveRightEventId)
    {
        m_strafe = value;
    }
    else if (*inputId == RotateYawEventId)
```

```
        {
            m_yaw = value;
        }
    }

void ChickenMovementComponentController::OnHeld(float value)
{
    const InputEventNotificationId* inputId =
        InputEventNotificationBus::GetCurrentBusId();
    if (inputId == nullptr)
    {
        return;
    }

    if (*inputId == RotateYawEventId)
    {
        m_yaw = value;
    }
}

void ChickenMovementComponentController::OnReleased(float value)
{
    const InputEventNotificationId* inputId =
        InputEventNotificationBus::GetCurrentBusId();
    if (inputId == nullptr)
    {
        return;
    }

    if (*inputId == MoveFwdEventId)
    {
        m_forward = value;
    }
    else if (*inputId == MoveRightEventId)
    {
        m_strafe = value;
    }
    else if (*inputId == RotateYawEventId)
    {
        m_yaw = value;
    }
}

void ChickenMovementComponentController::UpdateRotation(
    const ChickenMovementComponentNetworkInput* input)
{
    AZ::TransformInterface* t = GetEntity()->GetTransform();

    float currentHeading = t->GetWorldRotationQuaternion().
        GetEulerRadians().GetZ();
    currentHeading += input->m_viewYaw * GetTurnSpeed();
    AZ::Quaternion q =
        AZ::Quaternion::CreateRotationZ(currentHeading);
```

```
t->SetWorldRotationQuaternion(q);
}

void ChickenMovementComponentController::UpdateVelocity(
    const ChickenMovementComponentNetworkInput* input)
{
    const float currentHeading = GetEntity()->GetTransform()->
        GetWorldRotationQuaternion().GetEulerRadians().GetZ();

    const AZ::Vector3 fwd = AZ::Vector3::CreateAxisY(
        input->m_forwardAxis);
    const AZ::Vector3 strafe = AZ::Vector3::CreateAxisX(
        input->m_strafeAxis);
    AZ::Vector3 combined = fwd + strafe;
    if (combined.GetLength() > 1.f)
    {
        combined.Normalize();
    }
    m_velocity = AZ::Quaternion::CreateRotationZ(currentHeading).
        TransformVector(combined) * GetWalkSpeed() +
        AZ::Vector3::CreateAxisZ(GetGravity());
}
} // namespace MyGem
```

---

# Chapter 36. Multiplayer Physics

## Introduction

### Note

The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch36\\_multiplayer\\_physics](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch36_multiplayer_physics)

With multiplayer player movement implemented, the next priority is fixing the ball. It is a rigid body that does not respect the server. Each instance of a game or the Editor simulates independently and so does the goal detector.

Our first step is to convert the soccer ball entity into a network rigid body and then to implement server-authoritative goal detection, while still delivering score updates to client's user interface.

## Network Ball

In order to convert the ball into a network rigid body, we have to do the following steps:

1. Add a Network Binding component to mark the entity as a network entity.
2. Add a Network Transform component to synchronize its position.
3. Add a Network Rigid Body component to let the server drive the simulation while disabling physics for this entity on clients.
4. Turn the Ball entity into a prefab. Save the prefab as `Network_Ball.prefab`.



### Important

We have to wrap a network entity with a prefab, otherwise the level will fail to load as network entities are not supported directly in the level.

With these changes the ball has become a server authoritative entity but the goal detector logic is now broken, since the client is no longer running physical simulation for the ball. Physical shape trigger is no longer triggering on clients against a ball that has physics disabled by Network Rigid Body component. The solution is to send score notifications from the server to clients.

### Note

Network Rigid Body component still runs physical simulation on the server but disables it for the same entity on clients. This way client entities follow server simulation.

## Network Property

Since we are building a server authoritative game, the server has to decide when a goal is scored. To that end, we are going to convert our Goal Detector component into a multiplayer component. When a goal

is scored the score needs to be replicated to clients so that they are aware when a goal is scored. That requires that we add game logic on the server and clients, which means the new Goal Detector multiplayer component will override both its generated component and its controller.

**What is a Network Property?** It is a property on a multiplayer component that is commonly controlled on the server and replicated to clients. You can define such properties in XML definition of multiplayer components with a Network Property element.

```
<NetworkProperty Type="int" Name="Score" Init="0"
    ReplicateFrom="Authority" ReplicateTo="Client"
```

This is an example of a network property that keeps a game score that the authoritative server can modify and replicate to clients, as defined by Replicate From and Replicate To fields. We will see it in action in this chapter as we build a multiplayer goal detector.

## Multiplayer Goal Detector

### ⚠ Important

When you convert a component to a multiplayer component, remove it from the list of manually registered components in AZ::Module. Code generation portion will register it for us.

Here are the differences between single player and multiplayer versions of Goal Detector component.

## Team as an Archetype Property

Single-player Goal Detector component reflected the team identifier.

```
->DataElement(0, &GoalDetectorComponent::m_team,
    "Team", "Which team is this goal line for?")
```

We will do the same work using an Archetype Property.

```
<ArchetypeProperty Type="int" Name="Team" Init="0"
    ExposeToEditor="true" />
```

One can access this value using generated GetTeam() method.

```
void GoalDetectorComponent::OnScoreChanged(int newScore)
{
    UiScoreNotificationBus::Broadcast(
        &UiScoreNotificationBus::Events::OnTeamScoreChanged,
        GetTeam(), newScore);
}
```

## Score

The old goal detector kept the score value in a UI component but that will not work with the server-authoritative design as UI is not even loaded in the server launcher. So we will move this value to the controller of new Goal Detector component. This way the value will be controlled by the server, with clients receiving updates to the score value when a goal is scored.

### Example 36.1. Complete GoalDetectorComponent.AutoComponent.xml

```
<?xml version="1.0"?>
```

```
<Component
    Name="GoalDetectorComponent"
    Namespace="MyGem"
    OverrideComponent="true"
    OverrideController="true"
    OverrideInclude="Multiplayer/GoalDetectorComponent.h"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >

    <NetworkProperty Type="int" Name="Score" Init="0"
        ReplicateFrom="Authority" ReplicateTo="Client"
        IsRewindable="false" IsPredictable="false" IsPublic="true"
        Container="Object" ExposeToEditor="false"
        ExposeToScript="true" GenerateEventBindings="true"
        Description="Current score on this side" />

    <ArchetypeProperty Type="int" Name="Team" Init="0"
        ExposeToEditor="true" />
</Component>
```

This will give the controller class a `ModifyScore()` method. The server side logic of detecting a goal remains the same as the single-player version of Goal Detector component.

```
void GoalDetectorComponentController::OnTriggerEvents(
    const AzPhysics::TriggerEventList& tel)
{
    const AZ::EntityId me = GetEntity()->GetId();
    using namespace AzPhysics;
    for (const TriggerEvent& te : tel)
    {
        if (te.m_triggerBody &&
            te.m_triggerBody->GetEntityId() == me)
        {
            if (te.m_type == TriggerEvent::Type::Enter)
            {
                // TODO respawn the ball
                ModifyScore()++;
                break;
            }
        }
    }
}
```

## >Note

We will add respawn logic to the soccer ball in the next chapter.

## Generate Event Bindings

Generate Event Bindings field of a Network Property adds a way to sign up for notifications of changes.

```
<NetworkProperty Name="Score" ... GenerateEventBindings="true" />
```

The sign up method's name is the name of the property with "AddEvent" suffix.

```
void ScoreAddEvent(AZ::Event<int>::Handler& handler);
```

On clients, ScoreAddEvent will allow us to sign up for change notifications. Here is how.

1. Create an event handler.

```
AZ::Event<int>::Handler m_scoreChanged;
```

2. Create a callback to handle the change.

```
void OnScoreChanged(int newScore);
```

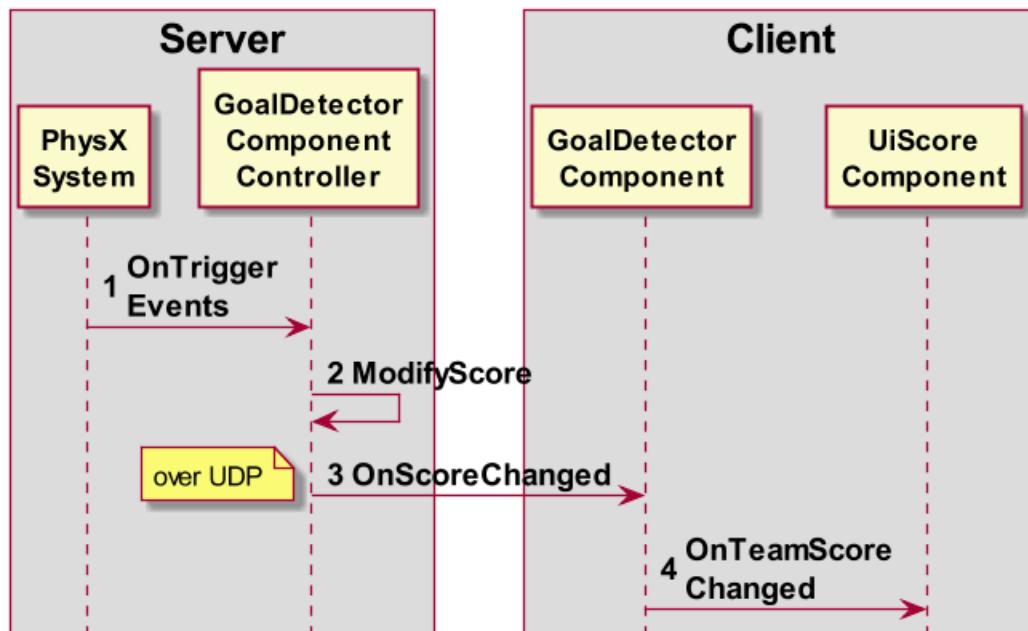
3. Assign the callback to the handler in the constructor of the component.

```
GoalDetectorComponent::GoalDetectorComponent()
: m_scoreChanged([this](int newScore)
{
    OnScoreChanged(newScore);
}) {}
```

4. Connect the handler to the event on activation of the component.

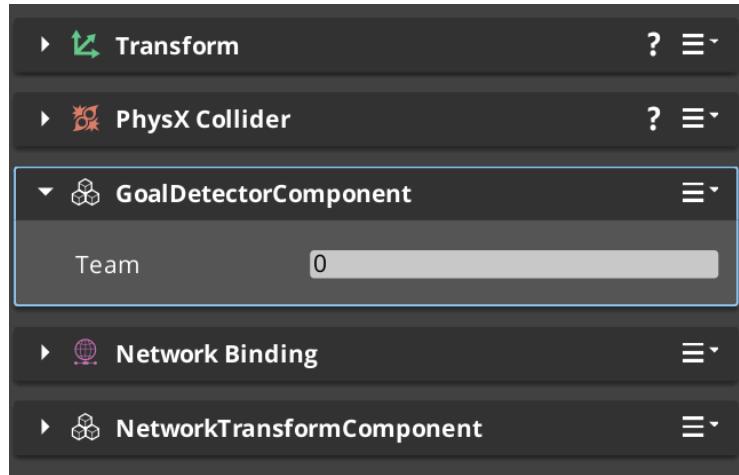
```
void GoalDetectorComponent::OnActivate(
    Multiplayer::EntityIsMigrating)
{
    ScoreAddEvent(m_scoreChanged);
}
```

**Figure 36.1. Network Property from the Server to a Client**

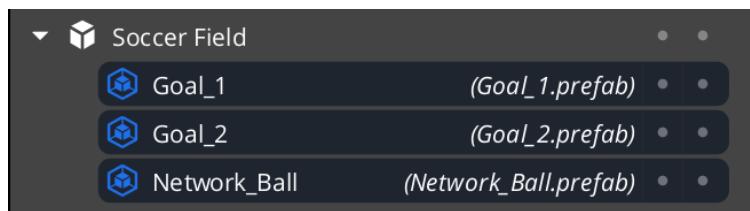


## Entity Changes

As a network entity, each goal entity will need a Network Binding and a Network Transform component.

**Figure 36.2. Updated Goal Entity**

Since goal entities are now network entities, we need to wrap them in a prefab. In fact, we will need two prefabs, one for each side of the soccer field. The only difference will be in the team value assigned to Goal Detector components. One of them will have a team value of zero (0) and the other a value of one (1).



## Summary

### Note

The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch36\\_multiplayer\\_physics](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch36_multiplayer_physics)

Detection of a goal has been successfully moved to the server. Goals are being scored again in the Editor game mode. UI is updated. However, the soccer ball is no longer being moved to the center of the field after a goal is scored. In the next chapter, I will show you a different approach of restarting the ball position by deleting the old ball and spawning a brand new one.

Until then, here is the source code for multiplayer Goal Detector component.

### **Example 36.2. GoalDetectorComponent.AutoComponent.h**

```
<?xml version="1.0"?>
<Component
    Name="GoalDetectorComponent"
    Namespace="MyGem"
    OverrideComponent="true"
    OverrideController="true"
    OverrideInclude="Multiplayer/GoalDetectorComponent.h"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<NetworkProperty Type="int" Name="Score" Init="0"
    ReplicateFrom="Authority" ReplicateTo="Client"
    IsRewindable="false" IsPredictable="false" IsPublic="true"
    Container="Object" ExposeToEditor="false"
    ExposeToScript="true" GenerateEventBindings="true"
    Description="Current score on this side"/>

<ArchetypeProperty Type="int" Name="Team" Init="0"
    ExposeToEditor="true" />
</Component>
```

### Example 36.3. GoalDetectorComponent.h

```
#pragma once
#include <AzCore/Component/Component.h>
#include <AzFramework/Physics/Common/PhysicsEvents.h>
#include <Source/AutoGen/GoalDetectorComponent.AutoComponent.h>

namespace MyGem
{
    class GoalDetectorComponent
        : public GoalDetectorComponentBase
    {
public:
    AZ_MULTIPLAYER_COMPONENT(MyGem::GoalDetectorComponent,
        s_goalDetectorComponentConcreteUuid,
        MyGem::GoalDetectorComponentBase);

    GoalDetectorComponent();
    static void Reflect(AZ::ReflectContext* context);

    void OnInit() override {}
    void OnActivate(Multiplayer::EntityIsMigrating) override;
    void OnDeactivate(Multiplayer::EntityIsMigrating) override {}

private:
    AZ::Event<int>::Handler m_scoreChanged;
    void OnScoreChanged(int newScore);
};

class GoalDetectorComponentController
    : public GoalDetectorComponentControllerBase
{
public:
    GoalDetectorComponentController(GoalDetectorComponent& p);

    void OnActivate(Multiplayer::EntityIsMigrating) override;
    void OnDeactivate(Multiplayer::EntityIsMigrating) override {}

private:
    AzPhysics::SceneEvents::
        OnSceneTriggersEvent::Handler m_trigger;
};
```

```
    void OnTriggerEvents(
        const AzPhysics::TriggerEventList& tel);
}
} // namespace MyGem
```

### Example 36.4. GoalDetectorComponent.cpp

```
#include <AzCore/Component/TransformBus.h>
#include <AzCore/Interface/Interface.h>
#include <AzCore/Serialization/EditContext.h>
#include <AzFramework/Physics/PhysicsScene.h>
#include <Multiplayer/GoalDetectorComponent.h>
#include <MyGem/UiScoreBus.h>

namespace MyGem
{
    GoalDetectorComponent::GoalDetectorComponent()
        : m_scoreChanged([this](int newScore)
    {
        OnScoreChanged(newScore);
    }) {}

    void GoalDetectorComponent::OnScoreChanged(int newScore)
    {
        UiScoreNotificationBus::Broadcast(
            &UiScoreNotificationBus::Events::OnTeamScoreChanged,
            GetTeam(), newScore);
    }

    void GoalDetectorComponent::Reflect(AZ::ReflectContext* rc)
    {
        auto sc = azrtti_cast<AZ::SerializeContext*>(rc);
        if (sc)
        {
            sc->Class<GoalDetectorComponent,
                GoalDetectorComponentBase>()->Version(1);
        }
        GoalDetectorComponentBase::Reflect(rc);

        if (auto bc = azrtti_cast<AZ::BehaviorContext*>(rc))
        {
            bc->EBus<UiScoreNotificationBus>("ScoreNotificationBus")
                ->Handler<ScoreNotificationHandler>();
        }
    }

    void GoalDetectorComponent::OnActivate(
        Multiplayer::EntityIsMigrating)
    {
        ScoreAddEvent(m_scoreChanged);
    }

    // Controller
    GoalDetectorComponentController::GoalDetectorComponentController()
```

```
GoalDetectorComponent& parent)
: GoalDetectorComponentControllerBase(parent)
, m_trigger([this]{
    AzPhysics::SceneHandle,
    const AzPhysics::TriggerEventList& tel)
{
    OnTriggerEvents(tel);
}) {}

void GoalDetectorComponentController::OnActivate(
    Multiplayer::EntityIsMigrating)
{
    auto* si = AZ::Interface<AzPhysics::SceneInterface>::Get();
    if (si != nullptr)
    {
        AzPhysics::SceneHandle sh = si->GetSceneHandle(
            AzPhysics::DefaultPhysicsSceneName);
        si->RegisterSceneTriggersEventHandler(sh, m_trigger);
    }
}

void GoalDetectorComponentController::OnTriggerEvents(
    const AzPhysics::TriggerEventList& tel)
{
    const AZ::EntityId me = GetEntity()->GetId();
    using namespace AzPhysics;
    for (const TriggerEvent& te : tel)
    {
        if (te.m_triggerBody &&
            te.m_triggerBody->GetEntityId() == me)
        {
            if (te.m_type == TriggerEvent::Type::Enter)
            {
                // TODO respawn the ball
                ModifyScore()++;
                break;
            }
        }
    }
} // namespace MyGem
```

# Chapter 37. Removal and Spawning

## Introduction

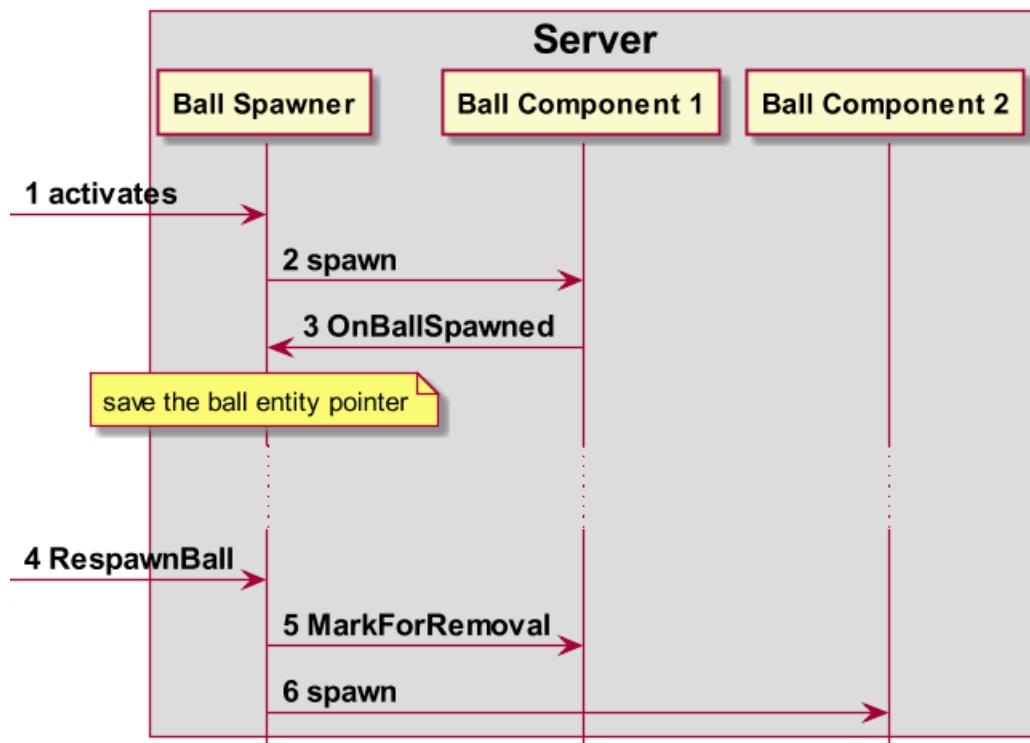
### Note

The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch37\\_removal\\_and\\_spawning](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch37_removal_and_spawning)

Chapter 36, *Multiplayer Physics*, re-implemented goal detection to use a server-authoritative mechanism. However, the soccer ball is not being placed back in the middle of the soccer field. In this chapter, I will show you to do that by deleting the old ball and spawning a new one.

**Figure 37.1. Design of the New Ball Spawner**



1. Previously, the soccer ball was placed on the level directly in the Editor. This time a new entity with a new Ball Spawner component will do that on entity activation.
2. I covered how to spawn a prefab in Chapter 11, *Introduction to Prefabs*.
3. Once `Network_Ball.prefab` has spawned, Ball component will call back to the spawner with `OnBallSpawned`. The spawner will save the entity pointer to manage its lifetime.
4. When it is necessary to respawn a ball after a goal is scored, a new EBus request will ask the spawner to respawn the ball with `RespawnBall`.
5. `MarkForRemoval` is a new Multiplayer gem API presented in this chapter. It is capable of deleting network entities.

## ⚠ Important

One must not deactivate and delete network entities directly. One has to use `MarkForRemoval` method from Network Entity Manager as shown later in this chapter.

6. The game cycle continues by spawning a new ball.

# Ball Component

By creating a Ball component and placing it on the Ball entity, we can send an event when a new Ball activates in the level after spawning. Since the spawning and removal is initiated on the server, this multiplayer component needs a controller. When a new network entity is created on the server, the multiplayer system will take care of creating the entity on clients. The same is true when a network entity is removed on the server.

### Example 37.1. BallComponent.AutoComplete.xml

```
<?xml version="1.0"?>
<Component>
    Name="BallComponent"
    Namespace="MyGem"
    OverrideComponent="false"
    OverrideController="true"
    OverrideInclude="Multiplayer/BallComponent.h"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
</Component>
```

The controller of Ball component will send out a notification when its entity activates.

```
void BallComponentController::OnActivate(
    Multiplayer::EntityIsMigrating)
{
    BallNotificationBus::Broadcast(
        &BallNotificationBus::Events::OnBallSpawned,
        GetEntity());
}
```

That is all the work that Ball Component does.

# Ball Spawner Component

We have already implemented a spawner in Chapter 11, *Introduction to Prefabs*. This time we will do it in a multiplayer component that deletes older entities and spawns new ones.

In order to cycle through the reference of the soccer ball entities, Ball Spawner component will keep their pointers in a small ring buffer.

```
AZStd::ring_buffer<AZ::Entity*> m_balls;
m_balls.set_capacity(2);
```

## 💡 Tip

A ring buffer is a circular container of a configurable constant size. New entries will overwrite the oldest elements first.

# Removing Network Entities

## ⚠ Important

Removing a network entity must not be done by directly deactivating them. Network entities have a number of systems behind-the-scenes handling their states and updates. Their removal must be requested from `INetworkEntityManager` interface with `MarkForRemoval`.

```
/// Marks the specified entity for removal and deletion.  
/// @param entityHandle the entity to remove and delete  
void MarkForRemoval(const ConstNetworkEntityHandle& h);
```

`Multiplayer::ConstNetworkEntityHandle` is a wrapper that refers to a network entity. It can be constructed by giving it a pointer to a network entity.

## >Note

A network entity is an entity that has a Network Binding component on it.

Here is how to remove an old ball using a pointer to its entity.

```
AZ::Entity* previousBall = m_balls.back();  
Multiplayer::ConstNetworkEntityHandle oldBall(previousBall);  
AZ::Interface<IMultiplayer>::Get()->  
    GetNetworkEntityManager()->MarkForRemoval(oldBall);
```

## ⚠ Important

Marked entity will not go away immediately, it may take another game tick for the entity and various network resources to be cleaned up and deleted.

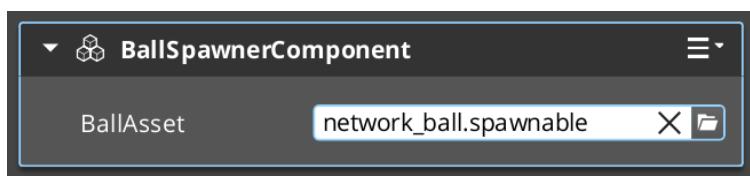
In order to disable any unwanted interaction with an old ball, we will deactivate physics, so that an old ball no longer activates goal triggers.

```
using RigidBus = Physics::RigidBodyRequestBus;  
RigidBus::Event(previousBall->GetId(),  
    &RigidBus::Events::DisablePhysics);
```

# Spawning Network Entities

Spawning network entities uses the same method as in Chapter 11, *Introduction to Prefabs*, but with one interesting detail. We are going to specify a Spawnable field for the ball prefab to spawn.

**Figure 37.2. Ball Spawner with a Prefab**



In Chapter 11, *Introduction to Prefabs*, we reflected `AZ::Data::Asset<AzFramework::Spawnable>` ourselves but we can also do that using Archetype Property in the auto-component XML definition.

```
<ArchetypeProperty  
    Type="AZ::Data::Asset<AzFramework::Spawnable>"  
    Name="BallAsset" Init="" ExposeToEditor="true" />
```

We will be able to get this asset by its generated getter method, `GetBallAsset()`.

### Note

XML does not allow less than ("<") and greater than (">") symbols in element property values, so we have to escape them with "&lt;" and "&gt;"

Additionally, for special types such as `AzFramework::Spawnable` you have to provide their include files with `Include` element, so that the generated base classes can compile.

```
<Include File="AzCore/Asset/AssetSerializer.h"/>  
<Include File="AzFramework/Spawnable/Spawnable.h"/>
```

We are going to spawn balls where the Ball Spawner component is located at. We can help ourselves by asking the auto-component generator to provide us with a getter for the transform component.

```
<ComponentRelation Constraint="Weak" HasController="false"  
    Name="TransformComponent" Namespace="AzFramework"  
    Include="AzFramework/Components/TransformComponent.h"/>
```

That will give us ability to get the world transform by calling `GetParent().GetTransformComponent() -> GetWorldTM()`. Here is the entire `RespawnBall` method.

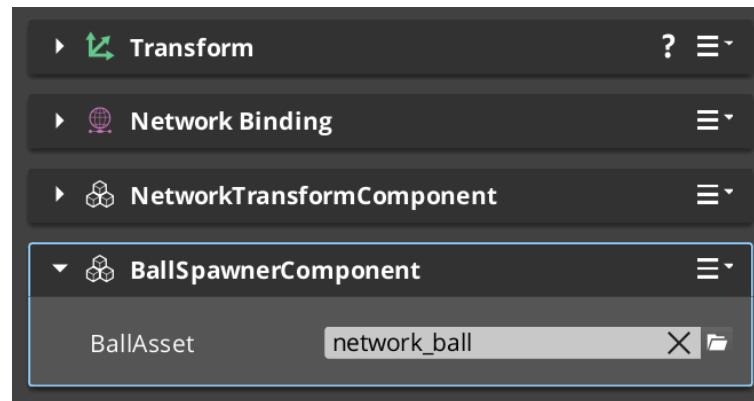
### Example 37.2. RespawnBall method

```
void BallSpawnerComponentController::RespawnBall()  
{  
    RemoveOldBall();  
  
    using namespace AzFramework;  
    AZ::Transform world = GetParent().GetTransformComponent() ->  
        GetWorldTM();  
    m_ticket = AzFramework::EntitySpawnTicket{ GetBallAsset() };  
  
    // place the ball  
    auto cb = [world](<  
        EntitySpawnTicket::Id /*ticketId*/,  
        SpawnableEntityContainerView view)</br>  
    {  
        const AZ::Entity* e = *view.begin();  
        if (auto* tc = e->FindComponent<TransformComponent>())  
        {  
            tc->SetWorldTM(world);  
        }  
    };  
  
    SpawnAllEntitiesOptionalArgs optionalArgs;  
    optionalArgs.m_preInsertionCallback = AZStd::move(cb);  
    SpawnableEntitiesInterface::Get()->SpawnAllEntities(  
        m_ticket, AZStd::move(optionalArgs));  
}
```

# Entity Changes

1. Modify Network\_Ball.prefab by adding Ball Component to the Ball entity.
2. Since we are going to spawn balls, remove Network\_Ball.prefab from the level.
3. Create a new entity, called Ball Spawner.
4. Add to it a Network Binding, a Network Transform and a Ball Spawner component.
5. Assign Network\_Ball to the Ball Asset field of Ball Spawner component.

**Figure 37.3. Ball Spawner Entity**



6. Create a prefab out of it, called Ball\_Spawner.prefab.



# Summary

## Note

The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch37\\_removal\\_and\\_spawning](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch37_removal_and_spawning)

In this chapter we added a respawn logic for the soccer balls. The moment a goal is scored, the current ball will be marked for removal, which removes it within one game frame or so, and we spawn a new ball in the middle of the field.

Here is the full source code for Ball and Ball Spawner components.

### Example 37.3. BallComponent.AutoComplete.h

```
<?xml version="1.0"?>
<Component
    Name="BallComponent"
```

```
Namespace="MyGem"
OverrideComponent="false"
OverrideController="true"
OverrideInclude="Multiplayer/BallComponent.h"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
</Component>
```

#### Example 37.4. BallBus.h

```
#pragma once
#include <AzCore/Component/ComponentBus.h>
namespace MyGem
{
    class BallNotifications
        : public AZ::ComponentBus
    {
    public:
        virtual void OnBallSpawned(AZ::Entity* ballEntity) = 0;
    };

    using BallNotificationBus = AZ::EBus<BallNotifications>;
}
```

#### Example 37.5. BallComponent.h

```
#pragma once
#include <Source/AutoGen/BallComponent.AutoComponent.h>

namespace MyGem
{
    class BallComponentController
        : public BallComponentControllerBase
    {
    public:
        BallComponentController(BallComponent& parent);
        void OnActivate(Multiplayer::EntityIsMigrating) override;
        void OnDeactivate(Multiplayer::EntityIsMigrating) override{};
    };
}
```

#### Example 37.6. BallComponent.cpp

```
#include <Multiplayer/BallComponent.h>
#include <MyGem/BallBus.h>

namespace MyGem
{
    BallComponentController::BallComponentController(
        BallComponent& parent)
        : BallComponentControllerBase(parent) {}

    void BallComponentController::OnActivate(
        Multiplayer::EntityIsMigrating)
    {
```

```
    BallNotificationBus::Broadcast(
        &BallNotificationBus::Events::OnBallSpawned,
        GetEntity()));
}
```

### Example 37.7. BallSpawnerComponent.AutoComponent.h

```
<?xml version="1.0"?>
<Component
    Name="BallSpawnerComponent"
    Namespace="MyGem"
    OverrideComponent="false"
    OverrideController="true"
    OverrideInclude="Multiplayer/BallSpawnerComponent.h"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <ComponentRelation Constraint="Weak" HasController="false"
        Name="TransformComponent" Namespace="AzFramework"
        Include="AzFramework/Components/TransformComponent.h"/>

    <Include File="AzCore/Asset/AssetSerializer.h"/>
    <Include File="AzFramework/Spawnable/Spawnable.h"/>
    <ArchetypeProperty
        Type="AZ::Data::Asset<AzFramework::Spawnable>;"
        Name="BallAsset" Init="" ExposeToEditor="true" />
</Component>
```

### Example 37.8. BallSpawnerBus.h

```
#pragma once
#include <AzCore/Component/ComponentBus.h>
namespace MyGem
{
    class BallSpawnerRequests
        : public AZ::ComponentBus
    {
    public:
        virtual void RespawnBall() = 0;
    };

    using BallSpawnerRequestBus = AZ::EBus<BallSpawnerRequests>;
}
```

### Example 37.9. BallSpawnerComponent.h

```
#pragma once
#include <AzCore/std/containers/ring_buffer.h>
#include <MyGem/BallBus.h>
#include <MyGem/BallSpawnerBus.h>
#include <Source/AutoGen/BallSpawnerComponent.AutoComponent.h>

namespace MyGem
{
```

```
class BallSpawnerComponentController
: public BallSpawnerComponentControllerBase
, public BallSpawnerRequestBus::Handler
, public BallNotificationBus::Handler
{
public:
    BallSpawnerComponentController(BallSpawnerComponent& parent);

    void OnActivate(Multiplayer::EntityIsMigrating) override;
    void OnDeactivate(Multiplayer::EntityIsMigrating) override;

    // BallRequestBus
    void RespawnBall() override;

    // BallNotificationBus
    void OnBallSpawned(AZ::Entity* e) override;

private:
    AzFramework::EntitySpawnTicket m_ticket;
    AZStd::ring_buffer<AZ::Entity*> m_balls;

    void RemoveOldBall();
};

}
```

### Example 37.10. BallSpawnerComponent.cpp

```
#include <AzFramework/Components/TransformComponent.h>
#include <AzFramework/Physics/RigidBodyBus.h>
#include <Multiplayer/BallSpawnerComponent.h>
#include <Multiplayer/Components/NetworkTransformComponent.h>
#include <MyGem/BallSpawnerBus.h>

namespace MyGem
{
    BallSpawnerComponentController::BallSpawnerComponentController(
        BallSpawnerComponent& parent)
        : BallSpawnerComponentControllerBase(parent){}

    void BallSpawnerComponentController::OnActivate(
        Multiplayer::EntityIsMigrating)
    {
        const AZ::EntityId me = GetEntity()->GetId();
        BallSpawnerRequestBus::Handler::BusConnect(me);
        BallNotificationBus::Handler::BusConnect(me);
        m_balls.set_capacity(2);
        RespawnBall();
    }

    void BallSpawnerComponentController::OnDeactivate(
        Multiplayer::EntityIsMigrating)
    {
        BallSpawnerRequestBus::Handler::BusDisconnect();
        BallNotificationBus::Handler::BusDisconnect();
    }
}
```

```
}

void BallSpawnerComponentController::RespawnBall()
{
    RemoveOldBall();

    using namespace AzFramework;
    AZ::Transform world = GetParent().GetTransformComponent() ->
        GetWorldTM();
    m_ticket = AzFramework::EntitySpawnTicket{ GetBallAsset() };

    auto cb = [world](
        EntitySpawnTicket::Id /*ticketId*/,
        SpawnableEntityContainerView view)
    {
        const AZ::Entity* e = *view.begin();
        if (auto* tc = e->FindComponent<TransformComponent>())
        {
            tc->SetWorldTM(world);
        }
    };

    SpawnAllEntitiesOptionalArgs optionalArgs;
    optionalArgs.m_preInsertionCallback = AZStd::move(cb);
    SpawnableEntitiesInterface::Get()->SpawnAllEntities(
        m_ticket, AZStd::move(optionalArgs));
}

void BallSpawnerComponentController::OnBallSpawned(AZ::Entity* e)
{
    m_balls.push_back(e);
}

void BallSpawnerComponentController::RemoveOldBall()
{
    if (m_balls.empty() == false)
    {
        AZ::Entity* previousBall = m_balls.back();

        using RigidBus = Physics::RigidBodyRequestBus;
        RigidBus::Event(previousBall->GetId(),
                        &RigidBus::Events::DisablePhysics);

        using namespace Multiplayer;
        const ConstNetworkEntityHandle oldBall(previousBall);

        AZ::Interface<IMultiplayer>::Get()->
            GetNetworkEntityManager()->MarkForRemoval(oldBall);
    }
}
```

---

# Chapter 38. Multiplayer Animation

## Introduction

### Note

The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch38\\_multiplayer\\_animation](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch38_multiplayer_animation)

We have made great strides in converting most of the game logic to multiplayer. The next task is to synchronize the animation of chickens. In single-player, a chicken played a running animation when it moved. This was broken as we moved to multiplayer, because the client animation component no longer receives any updates of its velocity.

We could fix it by computing velocity of the chicken on a client by signing up to AZ::TransformBus movement notifications but that can lead to issues. For example, a chicken may receive a correction when its player input results disagree with the server. In such a case the chicken might pop back or forward depending on the error, and our velocity calculation will be wrong.

A better approach would be for the server to relay back the velocity to all clients.

## Changes to Chicken Movement Component

We replicated a server value to clients in Chapter 36, *Multiplayer Physics*, when we needed score values on clients. Now we are going to replicate a velocity field of type AZ::Vector3.

```
<NetworkProperty Type="AZ::Vector3" Name="Velocity"
    ReplicateFrom="Authority" ReplicateTo="Client"
    ...
}
```

Previously we had m\_velocity as a member of ChickenMovementComponentController. We are going to replace it with a code generated field, which will give us GetVelocity and SetVelocity methods. Modify the controller's ProcessInput method.

```
void ChickenMovementComponentController::ProcessInput( . . . )
{
    . . .
    GetNetworkCharacterComponentController() ->
        TryMoveWithVelocity(GetVelocity(), deltaTime);
}
```

Modify the way we save velocity changes.

```
void ChickenMovementComponentController::UpdateVelocity(
    const ChickenMovementComponentNetworkInput* input)
{
    . . .
    SetVelocity(AZ::Quaternion::CreateRotationZ(currentHeading) *
        TransformVector(combined) * GetWalkSpeed() +
        AZ::Vector3::CreateAxisZ(GetGravity()));
}
```

Enable Generate Event Bindings property.

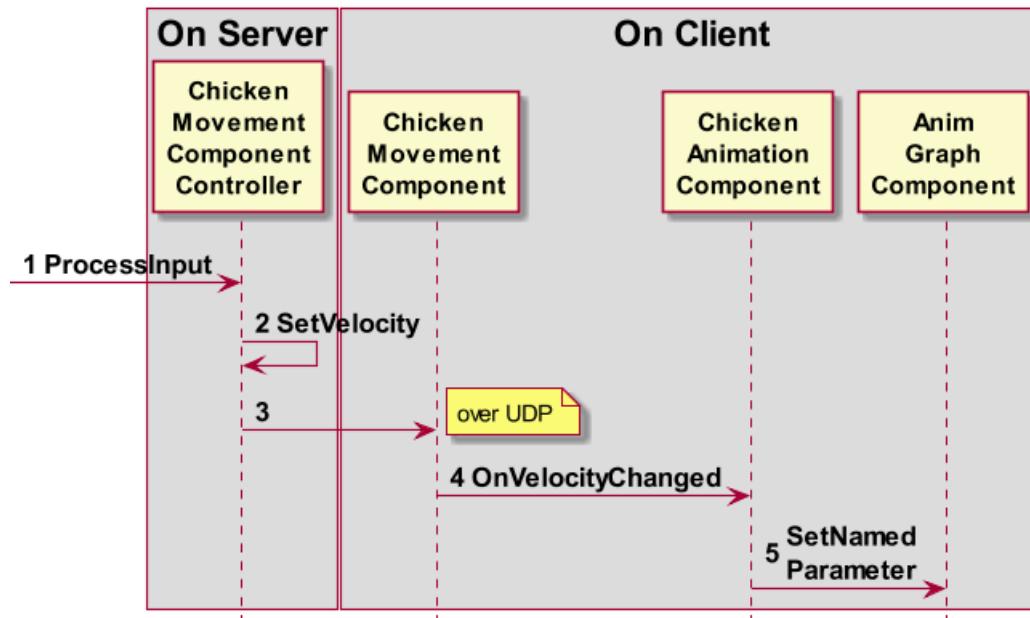
```
<NetworkProperty Type="AZ::Vector3" Name="Velocity"
    GenerateEventBindings="true"
```

This will generate **VelocityAddEvent** method, which serves as a way to sign up for velocity changes.

## Chicken Animation Component

In this chapter, we are converting Chicken Animation component to a multiplayer component that will listen for velocity changes from Chicken *Movement* component on clients where it will apply velocity change to the animation graph.

**Figure 38.1. Design of Multiplayer Chicken Animation Component**



1. The context starts at the server where processing of input has not changed.
2. Velocity network property is updated during processing input logic.
3. The multiplayer system replicates velocity change over the network to clients.
4. This is the first new change in this chapter. Chicken Animation component signs up and receives a notification that a velocity has changed.
5. The velocity value is applied to the animation graph the same way as was done in Chapter 26, *Animation State Machine*.

## Listening for Velocity Changes

### Tip

So far we listened for network property changes on the same component but we can also listen for the changes from another component on the same entity or even from a different entity.

Here are the steps to sign up for velocity changes on a client.

1. Declare a dependency of Chicken Animation component on Chicken Movement component.

```
<Component  
    Name="ChickenAnimationComponent" ...>  
  
    <ComponentRelation Constraint="Required" HasController="false"  
        Name="ChickenMovementComponent" Namespace="MyGem"  
        Include="Multiplayer/ChickenMovementComponent.h"/>
```

2. Add an event handler and a callback.

```
AZ::Event<AZ::Vector3>::Handler m_velocityChangedEvent;  
void OnVelocityChanged(AZ::Vector3 velocity);
```

3. Assign the callback to the handler in the component's constructor.

```
ChickenAnimationComponent::ChickenAnimationComponent()  
    : m_velocityChangedEvent([this](AZ::Vector3 velocity)  
    {  
        OnVelocityChanged(velocity);  
    })  
{}
```

4. Connect the handler to the event from Chicken Movement component.

```
void ChickenAnimationComponent::OnActivate(  
    Multiplayer::EntityIsMigrating)  
{  
    GetChickenMovementComponent()->VelocityAddEvent(  
        m_velocityChangedEvent);  
}
```

5. Drive animation graph parameters from the callback.

```
void ChickenAnimationComponent::OnVelocityChanged(  
    AZ::Vector3 velocity)  
{  
    velocity.SetZ(0);  
  
    using namespace EMotionFX::Integration;  
    AnimGraphComponentRequestBus::Event(GetEntityId(),  
        &AnimGraphComponentRequests::SetNamedParameterFloat,  
        "Speed", velocity.GetLength());  
}
```

### Note

I am setting the Z component of the velocity to zero since the running animation should not depend on the gravity portion that is pointing down the negative Z axis.

## Entity Changes

In this chapter we converted Chicken Animation component from a single-player to a multiplayer variant. Replace the component on the root entity of Chicken.prefab.

Additionally, we need to make some structural changes to `Chicken.prefab`, because Chicken Animation component now expects to find Animation Graph component on the same entity.



1. Remove old Chicken Animation component from Chicken entity.
2. Add new Chicken Animation component to top level Chicken entity.
3. Move all the components from Chicken\_Actor entity to the root Chicken entity. (Except Network Binding and Network Transform component, they are already there on Chicken entity.)
4. Delete Chicken\_Actor entity.
5. Delete Chicken\_Rigidbody entity. These entities are now empty and serve no purpose.

**Figure 38.2. Updated Chicken.prefab**



## Summary

### Note

The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch38\\_multiplayer\\_animation](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch38_multiplayer_animation)

In this chapter, we replicated animation states from the server to clients using velocity as the primary source of animation state.

### **Example 38.1. ChickenAnimationComponent.AutoComplete.h**

```
<?xml version="1.0"?>
<Component
    Name="ChickenAnimationComponent"
    Namespace="MyGem"
    OverrideComponent="true"
    OverrideController="false"
    OverrideInclude="Multiplayer/ChickenAnimationComponent.h"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<ComponentRelation Constraint="Required" HasController="false"
    Name="ChickenMovementComponent" Namespace="MyGem"
    Include="Multiplayer/ChickenMovementComponent.h"/>
</Component>
```

### Example 38.2. ChickenAnimationComponent.h

```
#pragma once
#include <Source/AutoGen/ChickenAnimationComponent.AutoComponent.h>

namespace MyGem
{
    class ChickenAnimationComponent
        : public ChickenAnimationComponentBase
    {
public:
    AZ_MULTIPLAYER_COMPONENT(MyGem::ChickenAnimationComponent,
        s_chickenAnimationComponentConcreteUuid,
        MyGem::ChickenAnimationComponentBase);

    static void Reflect(AZ::ReflectContext* rc);
    ChickenAnimationComponent();

    void OnInit() override {}
    void OnActivate(Multiplayer::EntityIsMigrating) override;
    void OnDeactivate(Multiplayer::EntityIsMigrating) override {}

private:
    AZ::Event<AZ::Vector3>::Handler m_velocityChangedEvent;
    void OnVelocityChanged(AZ::Vector3 velocity);
    };
}
```

### Example 38.3. ChickenAnimationComponent.cpp

```
#include <AzCore/Serialization/SerializeContext.h>
#include <Integration/AnimGraphComponentBus.h>
#include <Multiplayer/ChickenAnimationComponent.h>
#include <Source/AutoGen/ChickenMovementComponent.AutoComponent.h>

namespace MyGem
{
    void ChickenAnimationComponent::Reflect(AZ::ReflectContext* rc)
    {
        auto sc = azrtti_cast<AZ::SerializeContext*>(rc);
        if (sc)
        {
            sc->Class<ChickenAnimationComponent,
                ChickenAnimationComponentBase>()->Version(1);
        }
        ChickenAnimationComponentBase::Reflect(rc);
    }
}
```

```
ChickenAnimationComponent::ChickenAnimationComponent()
: m_velocityChangedEvent([this](AZ::Vector3 velocity)
{
    OnVelocityChanged(velocity);
})
{ }

void ChickenAnimationComponent::OnActivate(
    Multiplayer::EntityIsMigrating)
{
    GetChickenMovementComponent()->VelocityAddEvent(
        m_velocityChangedEvent);
}

void ChickenAnimationComponent::OnVelocityChanged(
    AZ::Vector3 velocity)
{
    velocity.SetZ(0);

    using namespace EMotionFX::Integration;
    AnimGraphComponentRequestBus::Event(GetEntityId(),
        &AnimGraphComponentRequests::SetNamedParameterFloat,
        "Speed", velocity.GetLength());
}
}
```

# Chapter 39. Team Spawner

## Introduction

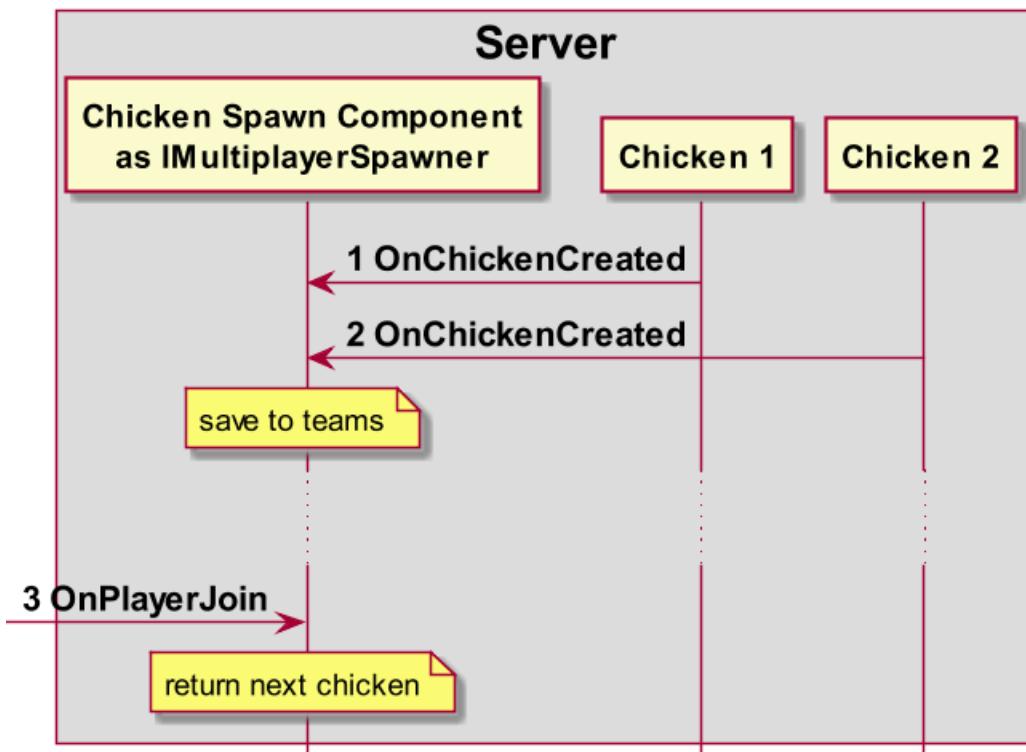
### Note

The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch39\\_team\\_spawner](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch39_team_spawner)

We have converted most of the game logic to multiplayer but we are not yet capable of supporting multiple players. If more than one player joins the server, they end up competing for controlling the same chicken. That cannot go well. This chapter will build on Chapter 34, *Simple Player Spawner*, by adding more chickens and giving each new player a different chicken from among two different teams.

**Figure 39.1. Team Spawner Design**



1. There will be a number of new chickens on the level.
2. Each chicken will report its activation and the team it is assigned by using `ChickenNotificationBus` that we already have in our game.
3. When the multiplayer system asks for a player entity for an incoming client, we will pick a chicken entity from one of the teams.

### >Note

If we just add more chicken prefabs to the level, we will uncover a defect in our design. We will have too many cameras on the level. There is a Camera component in `Chicken.prefab`, which means that for each new chicken we create in the level, another Camera component will be added. Instead of dealing with multiple cameras, this chapter will detach the camera from the chicken prefab and move it to the level. In the next chapter we will create Chicken Camera component to deal with the camera properly.

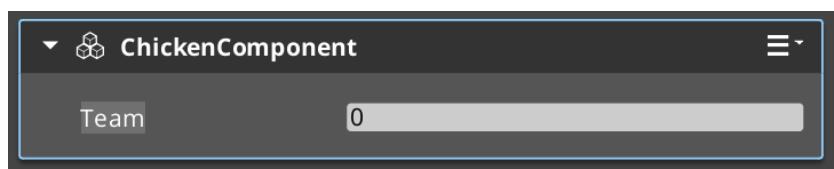
## Adding Team Value

In order to keep chickens separate between different teams, Chicken component XML definition will add a non-network property for the team number.

```
<Component Name="ChickenComponent" >
    <ArchetypeProperty Type="int" Name="Team" Init="0"
        ExposeToEditor="true" />
```

There will be two teams. We will set the team value in the Editor to either zero or one.

**Figure 39.2. Chicken Component with a Team Value**



## Adding More Chickens

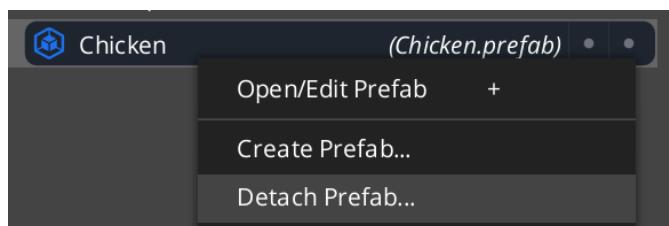
EAT MOR CHIKIN

—A popular food commercial

We will give up on `Chicken.prefab` and create two new chicken prefabs: `Chicken_Team_0.prefab` for the first chicken team and `Chicken_Team_1.prefab` for the second team.

### Tip

You can break apart a prefab with **Detach Prefab** command. That will remove the prefab from the level but leave behind all of its entities.



## Create a Team Chicken Prefab

1. Move Camera component out of `Chicken.prefab`.

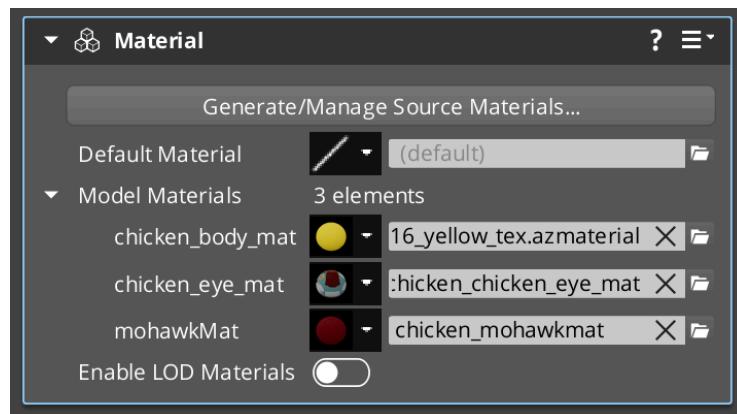
2. Use the remaining entities to create Chicken\_Team\_0.prefab and Chicken\_Team\_1.prefab.



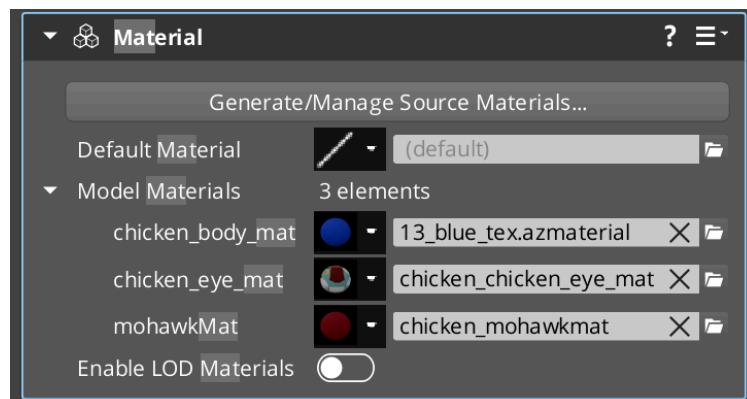
3. On Chicken entity, find Chicken component and set team value to zero (0).

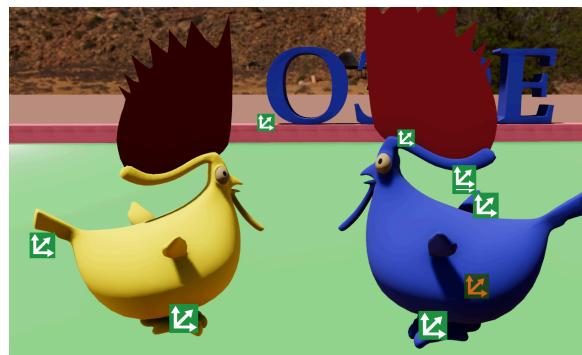


4. Set team value to one (1) for Chicken\_Team\_1.prefab.
5. In Chicken\_Team\_0.prefab, on Chicken entity, find Material component and change "chicken\_body\_mat" to a yellow material, such as 16\_yellow\_tex.azmaterial from Atom gems.



6. In Chicken\_Team\_1.prefab, on Chicken entity, find Material component and change "chicken\_body\_mat" to a blue material, such as 13\_blue\_tex.azmaterial from Atom gems.



**Figure 39.3. The Tale of Two Chickens**

This will give us two chicken types that will form opposite teams.

Under Chicken Spawn entity, create some number of chickens for each team. I created four for each team.



### Note

The camera no longer belongs to chicken prefabs.

## Changes to Chicken Component

On activation, chickens will report their team using an existing EBus.

```
virtual void OnChickenCreated(
    [[maybe_unused]] AZ::Entity* e,
    [[maybe_unused]] int team) {}
```

We already have a Chicken Component Controller. The only change is to use GetTeam() for the newly added Team network property.

```
void ChickenComponentController::OnActivate(
    Multiplayer::EntityIsMigrating)
{
    if (!IsAuthority()) return;

    ChickenNotificationBus::Broadcast(
```

```
    &ChickenNotificationBus::Events::OnChickenCreated,
        GetEntity(), GetTeam());
}
```

## Changes to Chicken Spawn Component

1. Chicken Spawn Component will store two containers of chicken entities, instead of just one chicken entity pointer as in Chapter 34, *Simple Player Spawner*.

```
AZStd::vector<AZ::Entity*> m_teams[2] = {};
AZ::Entity* GetNextChicken();
```

2. As chicken activation notifications come in, they will be stored into appropriate team container of entities.

```
void ChickenSpawnComponent::OnChickenCreated(
    AZ::Entity* e, int team)
{
    if (team >= 0 && team <= 1)
    {
        m_teams[team].push_back(e);
    }
}
```

3. GetNextChicken() method will pick from the team container that has more chickens remaining to keep the teams even.

```
AZ::Entity* ChickenSpawnComponent::GetNextChicken()
{
    AZStd::vector<AZ::Entity*>& team =
        m_teams[0].size() > m_teams[1].size() ?
            m_teams[0] : m_teams[1];
    if (team.empty()) return nullptr;

    AZ::Entity* newChicken = team.back();
    team.pop_back();

    return newChicken;
}
```

4. OnPlayerJoin is an existing method that the multiplayer system calls and expects to receive a network entity.

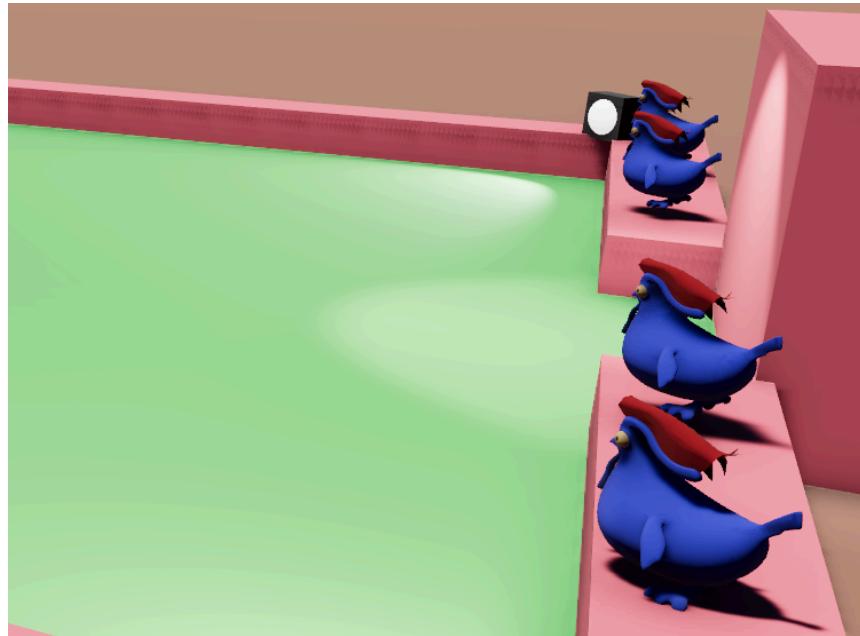
```
NetworkEntityHandle ChickenSpawnComponent::OnPlayerJoin(
    [[maybe_unused]] uint64_t userId,
    const Multiplayer::MultiplayerAgentDatum&)
{
    return NetworkEntityHandle{ GetNextChicken() };
}
```

## Summary

This chapter upgraded Chicken Spawner Component to support multiple players by placing multiple chickens on the level ahead of time. When a new player joins, they will control one of the chickens in the level.

You could spawn player entities but this approach allowed me to set up chickens on each side to give the soccer field a more welcoming look.

**Figure 39.4. Blue Chicken Team**



There is one last defect to take care of for the multiplayer portion of our game. The camera does not follow the right chicken. In fact, the camera does not follow any chicken at all right now. The next chapter will address this issue.

### Note

The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch39\\_team\\_spawner](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch39_team_spawner)

Chicken Spawn component is not a multiplayer component, so it has no XML definition.

### **Example 39.1. ChickenSpawnComponent.h**

```
#pragma once
#include <AzCore/Component/Component.h>
#include <Multiplayer/IMultiplayerSpawner.h>
#include <MyGem/ChickenBus.h>

namespace MyGem
{
    class ChickenSpawnComponent
        : public AZ::Component
        , public Multiplayer::IMultiplayerSpawner
        , public ChickenNotificationBus::Handler
    {
        public:
            AZ_COMPONENT(ChickenSpawnComponent,
```

```
"{814BAF21-10E4-4BE9-8380-C23B0EC27205}");  
  
static void Reflect(AZ::ReflectContext* rc);  
  
// AZ::Component interface implementation  
void Activate() override;  
void Deactivate() override;  
  
// ChickenNotificationBus  
void OnChickenCreated(AZ::Entity* e, int team) override;  
  
// IMultiplayerSpawner overrides  
Multiplayer::NetworkEntityHandle OnPlayerJoin(  
    uint64_t userId,  
    const Multiplayer::MultiplayerAgentDatum&) override;  
void OnPlayerLeave(  
    Multiplayer::ConstNetworkEntityHandle,  
    const Multiplayer::ReplicationSet&,  
    AzNetworking::DisconnectReason) override {}  
  
private:  
    AZStd::vector<AZ::Entity*> m_teams[2] = {};  
    AZ::Entity* GetNextChicken();  
};  
} // namespace MyGem
```

### Example 39.2. ChickenSpawnComponent.cpp

```
#include <ChickenSpawnComponent.h>  
#include <AzCore/Component/Entity.h>  
#include <AzCore/Interface/Interface.h>  
#include <AzCore/Serialization/EditContext.h>  
#include <Multiplayer/IMultiplayer.h>  
  
namespace MyGem  
{  
    using namespace Multiplayer;  
  
    void ChickenSpawnComponent::Reflect(AZ::ReflectContext* rc)  
    {  
        if (auto sc = azrtti_cast<AZ::SerializeContext*>(rc))  
        {  
            sc->Class<ChickenSpawnComponent, AZ::Component>()  
                ->Version(1);  
  
            if (AZ::EditContext* ec = sc->GetEditContext())  
            {  
                using namespace AZ::Edit;  
                ec->Class<ChickenSpawnComponent>(  
                    "Chicken Spawn",  
                    "[Player controlled chickens]")  
                    ->ClassElement(ClassElements::EditorData, "")  
                    ->Attribute(  
                        Attributes::AppearsInAddComponentMenu,
```

```
        AZ_CRC_CE( "Game" ) );
    }
}

void ChickenSpawnComponent::Activate()
{
    AZ::Interface<IMultiplayerSpawner>::Register(this);
    ChickenNotificationBus::Handler::BusConnect(GetEntityId());
}

void ChickenSpawnComponent::Deactivate()
{
    ChickenNotificationBus::Handler::BusDisconnect();
    AZ::Interface<IMultiplayerSpawner>::Unregister(this);
}

void ChickenSpawnComponent::OnChickenCreated(
    AZ::Entity* e, int team)
{
    if (team >= 0 && team <= 1)
    {
        m_teams[team].push_back(e);
    }
}

NetworkEntityHandle ChickenSpawnComponent::OnPlayerJoin(
    [[maybe_unused]] uint64_t userId,
    const Multiplayer::MultiplayerAgentDatum&)
{
    return NetworkEntityHandle{ GetNextChicken() };
}

AZ::Entity* ChickenSpawnComponent::GetNextChicken()
{
    AZStd::vector<AZ::Entity*>& team =
        m_teams[0].size() > m_teams[1].size() ?
            m_teams[0] : m_teams[1];
    if (team.empty()) return nullptr;

    AZ::Entity* newChicken = team.back();
    team.pop_back();

    return newChicken;
}
} // namespace MyGem
```

---

# Chapter 40. Multiplayer Camera

## Introduction

### Note

The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch40\\_multiplayer\\_camera](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch40_multiplayer_camera)

Chapter 39, *Team Spawner*, created two teams of chickens but disconnected the camera from following a chicken. This chapter will attach the camera to the player controlled chicken. There are eight (8) chickens on the level, so we will need some extra information to figure out which is the right chicken for each client.

## Autonomous Controllers

During the course of developing a multiplayer game, you will often have a need to execute some game logic that should only run on the local player controlled entity and nowhere else, not on the server and not on entities that are controlled by other clients. For example, in this chapter, we want the camera to follow only the chicken that the local player controls. The other chickens in the level must not grab the camera. This is where autonomous role comes in.

An *autonomous* role is a special role that is only granted to a client entity that was specifically assigned by a spawner, as we did in Chapter 34, *Simple Player Spawner*, and Chapter 39, *Team Spawner*.

A regular client component does not get a controller but the autonomous entity does get a controller and only for that autonomous entity. Only the autonomous entity is allowed to send input to the server. You can tell if your controller is autonomous by using **MultiplayerController::IsAutonomous** method.

```
void ChickenCameraComponentController::OnActivate(...)
{
    if (IsAutonomous())
    {
        // ...
    }
}
```

If you a client entity has an autonomous controller, then you can safely assume that this is the entity the client is controlling, which makes it the right anchor for the camera.

## Chicken Camera Component

Chicken Camera component will be a multiplayer component that overrides its controller, so that we can place our camera follow logic on the player controlled chicken entity. It will let the user specify its camera offset, so that we can place the camera behind the chicken.

```
<ArchetypeProperty Type="AZ::Vector3" Name="CameraOffset"
    ExposeToEditor="true" />
```

In the constructor, we will sign up for game tick events but only if the controller is autonomous.

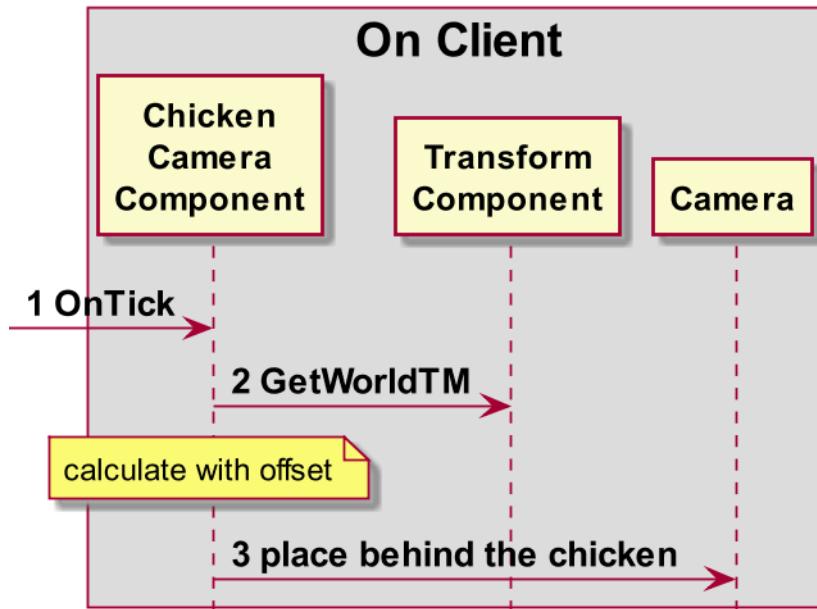
```
void ChickenCameraComponentController::OnActivate(
    Multiplayer::EntityIsMigrating)
{
```

```

if (IsAutonomous())
{
    AZ::TickBus::Handler::BusConnect();
}
}

```

**Figure 40.1. Design of Chicken Camera Component**



1. On each game tick grab the active camera. CameraSystemRequestBus is an EBus that will give you the camera.

```

AZ::Entity* ChickenCameraComponentController::GetActiveCamera()
{
    using namespace AZ;
    using namespace Camera;
    EntityId activeCameraId;
    CameraSystemRequestBus::BroadcastResult(
        activeCameraId,
        &CameraSystemRequestBus::Events::GetActiveCamera);

    auto ca = Interface<ComponentApplicationRequests>::Get();
    return ca->FindEntity(activeCameraId);
}

```

2. Query the current position and orientation of the player controlled chicken.

```

AZ::Transform chicken = GetParent().
    GetTransformComponent()->GetWorldTM();

```

3. Offset the position by GetCameraOffset() and move the camera behind the chicken.

```

void ChickenCameraComponentController::OnTick(
    float, AZ::ScriptTimePoint)
{
}

```

```
...
    AZ::Transform chicken =
        GetParent().GetTransformComponent()->GetWorldTM();
    AZ::Vector3 camera = chicken.GetTranslation() +
        chicken.GetRotation().TransformVector(GetCameraOffset());
    ...
    chicken.SetTranslation(camera);
    m_activeCameraEntity->GetTransform()->SetWorldTM(chicken);
}
```

## Entity Changes

Place Chicken Camera component on Chicken entity in both Chicken\_Team\_0.prefab and Chicken\_Team\_1.prefab.



### Tip

Once you add Chicken Camera component to the prefab and save it, you can modify the component values without the Editor.

1. Open the prefab at `MyProject\Prefabs\Chicken_Team_0.prefab`
2. Search for "ChickenCamera".
3. That will take you to JSON formatted configuration of the component.

```
"m_template": {
    "$type": "MyGem::ChickenCameraComponent",
    "CameraOffset": [
        0.0,
        -3.0,
        1.5
    ]
}
```

4. Make the change you want and save the prefab file.

## Summary

### Note

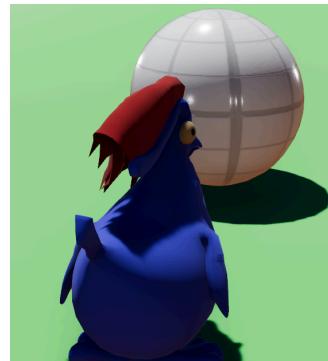
The accompanying source code and assets for this chapter can be found on GitHub at:

[https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch40\\_multiplayer\\_camera](https://github.com/AMZN-Olex/O3DEBookCode2111/tree/ch40_multiplayer_camera)

In this chapter we created a component that keeps the camera behind the chicken a player controls on a client. This allows us to keep a single camera component on the level. On the server, the camera will never

move since the controllers on the server are authoritative and not autonomous. On each client, the camera will be moved only by one of Chicken Camera components.

### Example 40.1. Camera behind a Chicken



### Example 40.2. ChickenCameraComponent.AutoComplete.xml

```
<?xml version="1.0"?>
<Component
    Name="ChickenCameraComponent"
    Namespace="MyGem"
    OverrideComponent="false"
    OverrideController="true"
    OverrideInclude="Multiplayer/ChickenCameraComponent.h"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <ComponentRelation Constraint="Weak" HasController="false"
        Name="TransformComponent" Namespace="AzFramework"
        Include="AzFramework/Components/TransformComponent.h"/>

    <ArchetypeProperty Type="AZ::Vector3" Name="CameraOffset"
        ExposeToEditor="true" />
</Component>
```

### Example 40.3. ChickenCameraComponent.h

```
#pragma once
#include <AzCore/Component/TickBus.h>
#include <Source/AutoGen/ChickenCameraComponent.AutoComplete.h>

namespace MyGem
{
    class ChickenCameraComponentController
        : public ChickenCameraComponentControllerBase
        , public AZ::TickBus::Handler
    {
        public:
            ChickenCameraComponentController(ChickenCameraComponent& p);

            void OnActivate(Multiplayer::EntityIsMigrating) override;
            void OnDeactivate(Multiplayer::EntityIsMigrating) override;
    };
}
```

```
// TickBus
void OnTick(float deltaTime, AZ::ScriptTimePoint) override;

private:
    AZ::Entity* m_activeCameraEntity = nullptr;
    AZ::Entity* GetActiveCamera();
};

}
```

#### Example 40.4. ChickenCameraComponent.cpp

```
#include <AzCore/Component/ComponentApplicationBus.h>
#include <AzFramework/Components/CameraBus.h>
#include <AzFramework/Components/TransformComponent.h>
#include <Multiplayer/ChickenCameraComponent.h>

namespace MyGem
{
    ChickenCameraComponentController::
        ChickenCameraComponentController(ChickenCameraComponent& p)
        : ChickenCameraComponentControllerBase(p) {}

    void ChickenCameraComponentController::OnActivate(
        Multiplayer::EntityIsMigrating)
    {
        if (IsAutonomous())
        {
            AZ::TickBus::Handler::BusConnect();
        }
    }

    void ChickenCameraComponentController::OnDeactivate(
        Multiplayer::EntityIsMigrating)
    {
        AZ::TickBus::Handler::BusDisconnect();
    }

    void ChickenCameraComponentController::OnTick(
        float, AZ::ScriptTimePoint)
    {
        if (!m_activeCameraEntity)
        {
            m_activeCameraEntity = GetActiveCamera();
            return;
        }

        AZ::Transform chicken =
            GetParent().GetTransformComponent()->GetWorldTM();
        AZ::Vector3 camera = chicken.GetTranslation() +
            chicken.GetRotation().TransformVector(GetCameraOffset());

        chicken.SetTranslation(camera);
        m_activeCameraEntity->GetTransform()->SetWorldTM(chicken);
    }
}
```

```
AZ::Entity* ChickenCameraComponentController::GetActiveCamera( )
{
    using namespace AZ;
    using namespace Camera;
    EntityId activeCameraId;
    CameraSystemRequestBus::BroadcastResult( activeCameraId,
        &CameraSystemRequestBus::Events::GetActiveCamera);

    auto ca = Interface<ComponentApplicationRequests>::Get();
    return ca->FindEntity(activeCameraId);
}
```

---

# Index

## A

Animation

- Blend Tree Node, 215
- Blend Two Node, 216
- Final Node, 218
- Motion Node, 218
- Parameter Action, 229
- Parameter Condition, 227
- Parameters, 218
- Play Speed, 220
- SetNamedParameterBool, 233
- State Condition, 228

AnimGraphComponentRequestBus, 222, 233, 315

AppearsInAddComponentMenu, 84

ApplyLinearImpulse, 208

Archetype Property, 283, 297

ArchetypeProperty, 320

Asset Processor, 8, 10

AudioTriggerComponentRequests, 251

AZ::Event, 56

AZ::Interface, 50

AZ::TransformBus, 313

AZ\_COMPONENT, 34

AZ\_CONSOLEFREEFUNC, 111

AZ\_CONSOLEFUNC, 111

AZ\_CVAR\_EXTERRED, 111

AZ\_UNIT\_TEST\_HOOK, 128

AZ\_UNUSED, 35

azmodel, 173

azrtti\_cast, 82

## B

Behavior Context, 231

BindTransformChangedEventHandler, 58

bootstrap.setreg, 9

## C

CameraSystemRequestBus, 328

CharacterBus, 152

CharacterRequestBus, 169

ClassBuilder, 83

CMake

- enabled\_gems.cmake, 103
- ZERO\_CHECK, 37
- ZERO\_TARGET, 102

Component, 16

- Actor, 212

- Anim Graph, 213

- Audio Listener, 252

- Audio Proxy, 249

- Audio Trigger, 249
- Directional Light, 178
- Image, 189
- Input, 142
- Local Prediction Player Input, 278
- Material, 21, 176, 321
- Mesh, 20
- Network Binding, 272
- Network Character, 283
- Network Transform, 272
- PhysX Collider, 192
- Simple Motion, 212
- Text, 190
- Transform, 17
- Transform 2D, 187
- UI Canvas Asset Ref, 186

Component Dependency, 73

- GetDependentServices, 73
- GetIncompatibleServices, 73
- GetProvidedServices, 73
- GetRequiredServices, 73

ComponentRelation, 283, 315

Console command

- Invoking, 113

ConstNetworkEntityHandle, 306

Controllers, 264

CreateComponentDescriptors, 259

## D

DependencyArrayType, 72

DisablePhysics, 198

## E

EBus, 62

- ComponentBus, 65

- TickBus, 70

- TransformBus, 65

EditContext, 82, 83

Editor, 8

- UI Editor, 186

editorsv\_enabled, 271

editorsv\_rhi\_override, 271

EnablePhysics, 198

engine\_name, 261

Entity, 15

- Entity Inspector, 12

- Entity Outliner, 89

- EntityId, 67

EXPECT\_CALL, 135

external\_subdirectories, 102

## F

Fbx Settings, 172

Find Component Input, 289

FindComponent, 43

## G

Gems, 101

EMotionFX, 212

LyShine, 185

LyShineExamples, 185

Multiplayer, 257

NvCloth, 115

StartingPointInput, 142

UiBasics, 185

Generate Event Bindings, 298, 313

GetActiveCamera, 328

GetEntityId, 67

GetParentId, 68

Google Mock, 131

Google Test, 122

## I

IConsole, 110

Import Audio Files, 243

IMultiplayerSpawner, 275

INetworkEntityManager, 306

Input Bindings, 159

Input Event Groups, 159

IsAuthority, 277

IsAutonomous, 327

## M

MarkForRemoval, 306

Material

Base Color, 181

Material Instance, 177

MOCK\_METHOD, 132

Mouse Input, 160

## N

Network Input, 284

Network Property, 297

NetworkEntityHandle, 274

NiceMock, 137

## O

o3de.bat

create-gem, 102

create-project, 6

ON\_CALL, 135

OnActivate, 269

OnPlayerJoin, 274, 323

OnSceneTriggersEvent, 194

OverrideComponent, 268

OverrideController, 276

## P

PhysX

Collision Filtering, 193

Compute Mass, 208

Mass, 208

Trigger, 205

Prefabs

Creating, 89

JSON, 329

Spawning, 92

pxmesh, 173

## R

RegisterMultiplayerComponents, 259

RegisterSceneTriggersEventHandler, 195

Remote Procedure, 285

RigidBodyRequestBus, 198, 208

Ring buffer, 305

## S

Script Canvas, 233

SendClientInputCorrection, 288

Serialization

DataElement, 86

Field, 86

SerializeContext, 83

SetWorldTranslation, 66

Slice, 88

Soundbank, 245

Spawnable, 306

SpawnAllEntities, 307

Spot Lights, 178

## T

TEST\_F, 135

TriggerEvent, 195

TryMoveWithVelocity, 283, 289

## U

UiCanvasAssetRefBus, 186

UIHandlers, 86

UiTextBus, 197

Unity builds, 8

User Interface, 185

Anchors, 187

Scale to device, 187

## W

Wwise, 239