Game Development with Unreal Engine 5

Learn the Basics of Game Development in Unreal Engine 5

Mitchell Lynn

Cliff Sharif

Game Development with Unreal Engine 5

Learn the Basics of Game Development in Unreal Engine 5

Mitchell Lynn

Cliff Sharif

www.bpbonline.com

FIRST EDITION 2023

Copyright © BPB Publications, India

ISBN: 978-93-55513-441

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

Distributors:

BPB PUBLICATIONS
20, Ansari Road, Darya Ganj
New Delhi-110002
Ph: 23254990/23254991

DECCAN AGENCIES
4-3-329, Bank Street,
Hyderabad-500195
Ph: 24756967/24756400

MICRO MEDIA
Shop No. 5, Mahendra Chambers,
150 DN Rd. Next to Capital Cinema,
V.T. (C.S.T.) Station, MUMBAI-400 001
Ph: 22078296/22078297

BPB BOOK CENTRE
376 Old Lajpat Rai Market,
Delhi-110006
Ph: 23861747

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Manipal Technologies, Manipal

www.bpbonline.com

Dedicated to

Game Programmers, Application Developers and Game Designers, who like to make the "Unreal" with "Unreal Engine 5"

About the Authors

- **Cliff Sharif** (aka Emperor Katax) is programmer and digital artist. His first touch to digital content was with Commodore 64 and Amiga 500. He practiced early steps of programming multimedia and real-time application with these platforms and also touched first edges of digital art belongs to that era. Then he began his carrier with programming and digital art in 1998, by using Windows 98.

  Cliff was an Unreal Tournament pro gamer and was involved with WCG international tournaments as team organizer back in 2004. Then after publishing his first book "Unreal Engine Physics Essentials", he switched his main carrier on develop application with Unreal Engine. He was involved in develop, manage and design industry standard applications of Virtual Reality, Architectural Visualizations, Real-time midi-controlled visuals, and recently Virtual Production by using Unreal Engine with Blueprint and C++. He certified as "Unreal Authorized Instructor – UAI" by Epic Games in 2020 and after that his main focus went for teaching game programming and design with Unreal Engine to students around the world.

- **Mitchell Lynn** is a programmer who has been playing videogames since before he could walk, he started out playing classics like DOOM and Quake on his Grandfather's old hardware, and has continued playing through every gaming generation since. This experience has been extremely helpful to his progression and capability as a programmer.

  After completing secondary education, Mitch decided to enter into the industry of creating the games that had occupied so much of his life. After a couple of years studying under Cliff to become a competent programmer, he entered the industry working on Virtual Reality simulation software, and then started developing games, both of which using Unreal Engine. He specializes in data management-based programming, and is always happy to assist those interested in becoming a programmer themselves.

Preface

The Unreal Engine 5 is the latest game development engine released by Epic Games, and this book will cover the basic setup and usage of the engine, as well as provide and explain examples of how to create fundamental objects of a game.

After a quick review of mathematics used in game design, we go through Unreal Editor which is the main environment for debug and develop the application by using Unreal Engine 5. Then users will learn how to use programming skills in develop application by using unreal engine 5 templates and Blueprint visual programming language. This part will establish a base to discover and learn other parts of the engine in future chapters.

Users will learn create Actors which are fundamental game objects in the engine. Then we learn how to use data structures, im0plement event and event dispatchers, using interface, and handling users input data. We learn about shaders, Niagara particle system, Metasound and packaging the project by making practical examples with step-by-step instructions.

By the end of this book, user has an intermediate to advanced knowledge of how to use engine templates, and make standalone executable prototypes of games or interactive application using Unreal Engine 5.

Chapter 1 This chapter is for absolute beginners in game development. We learn about "Game Engine" and will focus on the Unreal Engine as an example of a popular and high-quality game engine which is the focus of this book. Then we go through the installation process and describe how to customize the engine based on the user requirements. Also, we will learn how to use the Epic Games Launcher which is basically an interface to access Epic Games applications, of which Unreal Engine 5 is the latest one.

Chapter 2 Game developers need proper math skills for solving complex problems. An example could be calculation movement of 3D object in space, or calculating a target angle between 2 objects when they are pointing towards the player. Solving each of these problems requires knowledge of math which we learn in this chapter. Covering each aspect of the mathematical approach used in the digital world is not our goal in this chapter, but the user will go through the basics of math and follow the learning path to permit understanding more complex topics.

Chapter 3 This chapter will cover using the Epic Games launcher to open, update, or repair the engine, as well as how to use the marketplace. It will also cover the basic standard layout of the editor, how to modify the layout of the editor, where the important menus are located, their purpose, and how to use them, as well as explain useful hotkeys (and Reroute nodes), and skim through the project settings and plugins menus.

Chapter 4 Unreal Engine 5 uses a visual scripting language for making games and applications known as "Blueprint". In this chapter users will learn the basics of blueprints, how to make and populate them in the scene and learn how to import mesh components into the game application and assign them in the blueprint. Also, users will learn how to use game engine resources by adding blueprint components to their blueprint object and basics of using physics. At the end, users learned basic knowledge of blueprint design before they learn coding in blueprints in the next chapters.

Chapter 5 By using previous knowledges from chapter 4, users will learn how to develop code in blueprints and make game objects. Also, users will learn about Unreal Engine 5 templates, and how to program user input to the game character which meet learning more about Project Setting in the editor. Also, user learn how to change graphic setting of game in cases of rendering and light.

Chapter 6 Levels within UE5 consist of objects, (of which there are many types), and knowing these objects and how they function within the application, is essential for efficient & effective "Object Oriented" programming. We learn about game objects and materials in this chapter. Numbers of actor components that create game object in the game, can use materials. Material changes the appearance of the game object and can be simple or very complex.

Chapter 7 This chapter will cover the basic types of data (Ones that see regular use) used within Unreal Engine 5 (And programming in general), what each type is exactly, additional information about them, as well as their uses and benefits, with use-case examples. It will also cover how they can interact with each other (Pin splitting and conversions)

Chapter 8 Previously in chapter 5, users learned to use collision events to handle the collision. Now they will learn an advanced level of programming blueprints which is "Event Handling" and "Interface". We learn the "core knowledge" of communication between game objects in the engine which is essential for any scenario of applications development with Unreal Engine 5.

Chapter 9, There are a number of more complicated types of variables that aren't used as much as the simpler ones, but are arguably more important, as they are generally used for creating larger, core systems that the simpler variables are used within. We also cover how to use data for changing pos, in an animation Instance game object.

Chapter 10, This chapter will cover the more advanced aspects of game objects, including some other components with their functionalities/events (Projectile components, actor components, projectile components, movement components, etc.). It will also cover saving data to the system that game is running on, for future usage (Serialization), which is important for keeping track of player progress and their settings.

Chapter 11, In this chapter, users learn how to use the Unreal Engine 5 audio engine and develop audio code for the game. This feature is new in Unreal Engine 5 compare to previous versions of the engine. Also, users learn the basics of using Niagara which is a particle system simulator inside the engine to generate VFX effects. We explain a simple pipeline by a practical example, for using these features.

Chapter 12, Packaging a project is the last milestone of making any application by using Unreal Engine. There are numbers of options, with layers of details which are designed for packaging the application for different target machines. It basically takes time and research, to learn the best options for packaging your application, but we cover the main concept behind packaging and how the editor provides tools and resources on this. Also, users will learn how to avoid possible issues on packaging data.

Appendix, The "Enhanced Input" is a new feature to implement complex input handling which will be a replacement of default input system in future versions including 5.1. This appendix explains the logic and features of "Enhanced Input" system, and will go through a practical example of how to implement it with a third person player character.

Coloured Images

Please follow the link to download the
Coloured Images of the book:

https://rebrand.ly/0m8xq5c

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Table of Contents

Chapter 1
What is Unreal Engine?

A long time ago, in order to make a video game, creative developers had to work hard and solve complex problems. They had to develop their own *Game Engine*, making assets, debug the application and release it with limited hardware variations in the market at that time. The *Game Engine* is an application for creating, debugging, and publishing a video game for target machines like PC, VR, mobile, and so on. After the release of Windows 95 by Microsoft, developers got standard framework and network tools for developing high-quality and multiplayer games, which led them to create more powerful game engines. At the same time, the game industry got massive support from hardware and software manufacturers, which not only increased audio and visual quality but also supported new gaming platforms and AR/VR gaming.

Today, a team of developers and designers with a creative mind have a number of game engines and massive resources from industry to begin their project. We will learn how to use *Unreal Engine 5*, which is one of the most successful game engines in the industry and has massive support from the community of game developers.

Structure

In this chapter, we will discuss the following topics:

- History of Unreal Engine
- Installing Epic Games Launcher

Objectives

After studying this chapter, you should be able to install Epic Game Launcher and Unreal Engine 5, and you should know how to customize your installation. In this chapter, we will also go through a brief history of Epic Games and Unreal Engine and review some important titles there.

History of Unreal Engine

Everything began back in the late ’80s with *Tim Sweeney* from Maryland, when he returned home from work for holiday and turned on his IBM computer to play a video game, like he did when he was younger. He soon noticed that he is passionate about making computer games as a career, but it was an unusual decision and risky at the time. The game industry was so poor in terms of hardware and software and more importantly, lack of technical resources for making games was a serious problem for developing video games; however, none of these problems changed his mind.

Tim is one of the pioneers of the game industry like *John Carmack* from Id software (as you can see in *Figure 1.1*), who developed their own game engine from the beginning of their career. At the very beginning, he learned how to develop applications by using object-oriented programming methods, and then he developed his first game, known as ZZT, released for MS-DOS by 1991. It was a puzzle game with the *modding* ability. After that, he entered the game industry by opening his own company, *Epic Game.*

***Figure 1.1**: John Carmack and Tim Sweeney in GDCA 2017*

In 1998, Epic Games released their first video game in the **first-person shooter** (**FPS**) genre called *Unreal*, and it got a lot of attention due to the quality of visuals in game, skeleton mesh animations, rendering techniques, audio effects, and rich game story. For making *Unreal*, they developed their own game engine with an editor to debug code and design game objects. By 1999, Epic Games released *Unreal Tournament*, which was a multiplayer first person shooter (FPS) game, and it was developed with same engine as *Unreal* (refer to *Figure 1.2*):

Unreal
"The best looking game of all time."
"The future of gaming."
"R.I.P. Quake II"
WINDOWS 95 NT 98 CD-ROM
GT Interactive Software
Unreal
TOURNAMENT
ALL-NEW TEAM COMBAT FOR 1-16 PLAYERS
DOES NOT REQUIRE INTERNET ACCESS OR A 3D CARD

***Figure 1.2**: First series of FPS shooter games which released by Epic Games and developed by Unreal Engine*

After the millennium, Epic Games put a massive push on the quality and flexibility of their engine and, they worked on supporting multiple platforms by the engine. That led studios and game developers to making several big titles in the industry, like *Bioshock* and *Borderland*, which brought massive marketing and support by the community of gamers and game designers.

The latest version of Epic Games game engine is known as **Unreal Engine 5**, and developers and designers have free access to the game engine for creating, designing, debugging, and publishing stand-alone game and non-game applications on multiple platforms and target machines like PC, Xbox, mobile, augmented reality, and virtual reality.

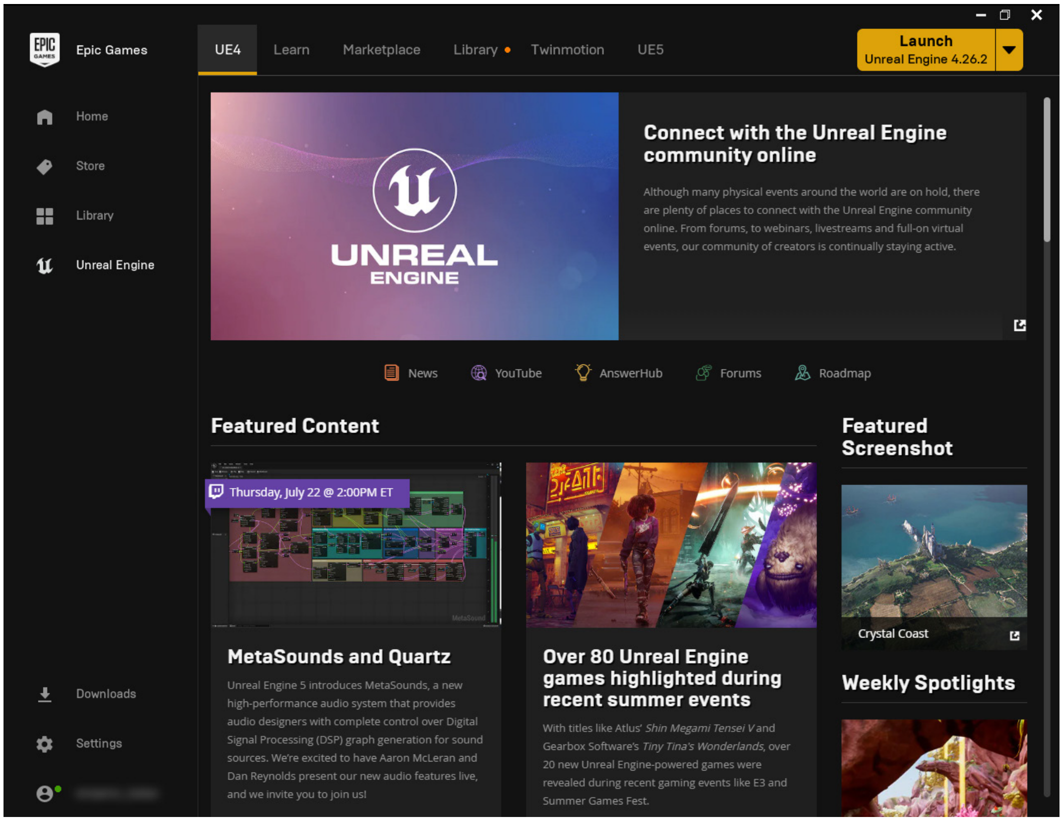
Now, let's install and use Unreal Engine 5.

Installing Epic Games Launcher

Before using Unreal Engine, you must download and install an application, known as **Epic Game Launcher**. This application is designed to organize your game development environment, access to standard game assets, latest updates, and community news when you use Unreal Engine 5.

Let's look at the steps for installing EPIC Games Launcher:

1. First, you need to make an account and profile in **https://www.epicgames.com/**, and then download and install the launcher. Then, open it and sign in to your Epic Games account from the launcher. You will have full access to Epic Games products, as shown in *Figure 1.3*, with Epic Games Launcher:



Connect with the Unreal Engine community online

Although many physical events around the world are on hold, there are plenty of places to connect with the Unreal Engine community online. From forums, to webinars, livestreams and full-on virtual events, our community of creators is continually staying active.

MetaSounds and Quartz

Unreal Engine 5 introduces MetaSounds, a new high-performance audio system that provides audio designers with complete control over Digital Signal Processing (DSP) graph generation for sound sources. We're excited to have Aaron McLeran and Dan Reynolds present our new audio features live, and we invite you to join us!

Over 80 Unreal Engine games highlighted during recent summer events

With titles like Atlus' *Shin Megami Tensei V* and Gearbox Software's *Tiny Tina's Wonderlands*, over 20 new Unreal Engine-powered games were revealed during recent gaming events like E3 and Summer Games Fest.

***Figure 1.3**: Epic Games Launcher*

2. To install Unreal Engine 5, click on the **`Unreal Engine`** tab in the launcher and then click on the **`Library`** tab on top; then, locate the **`ENGINE VERSIONS`** title and click on the plus-shaped drop box, as shown in the following figure:

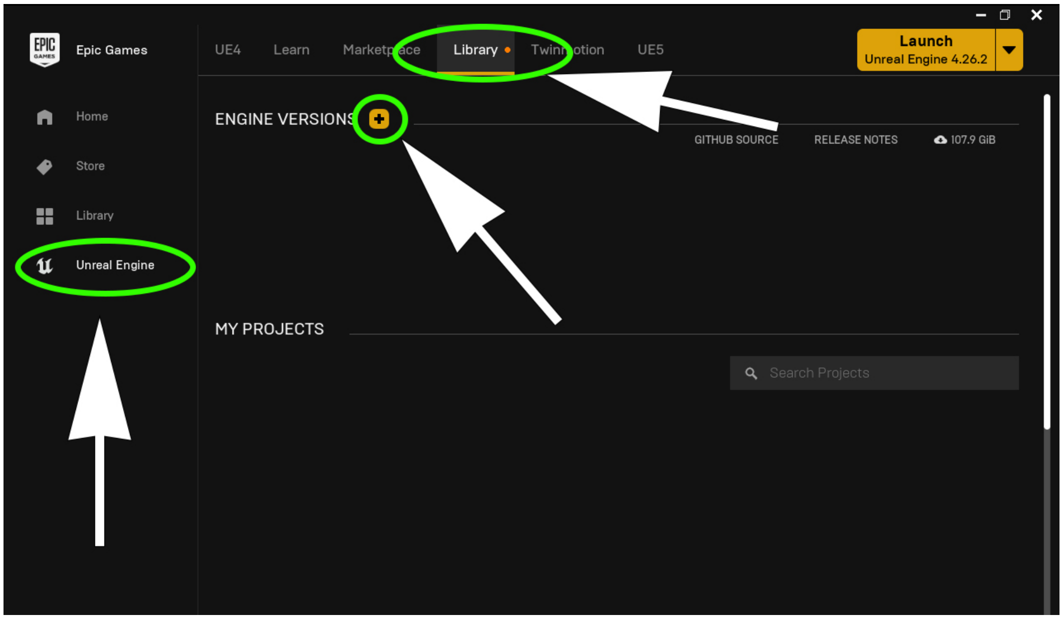


***Figure 1.4**: Navigate to Unreal Engine tab, click on library and add new engine version by clicking on the dropdown menu*

3. The launcher will automatically add an icon that represents your engine version. You can add and install multiple versions of Unreal Engine on your system by customizing the engine's version, as shown in *Figure 1.5*:

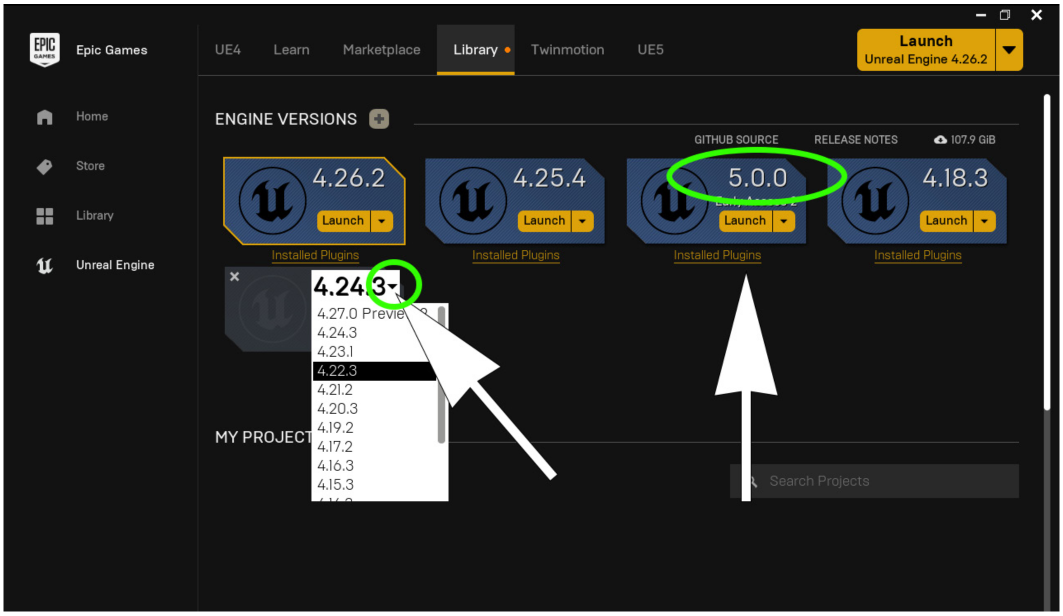


***Figure 1.5**: Select engine version from engine panel and install Unreal Engine 5*

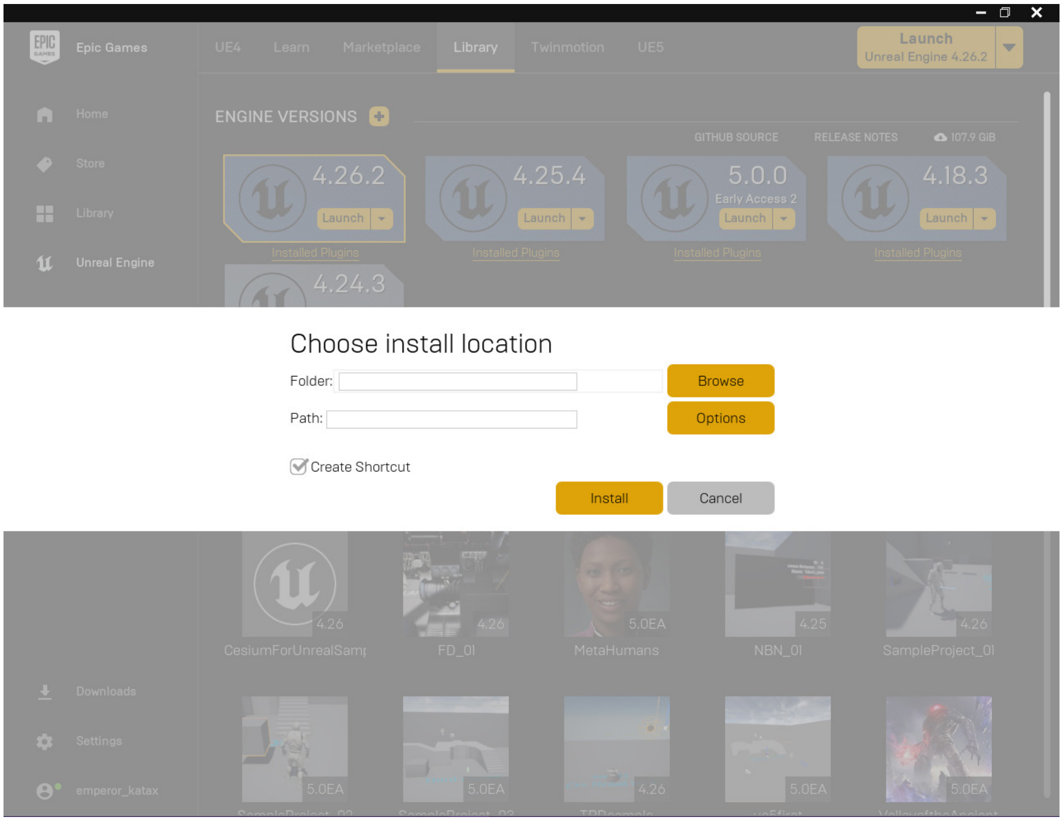
Here's the main reason behind this: imagine that you find a game asset or plugin from a marketplace or other source, which is not compatible with newer versions of the engine. In these cases, you can download the older version of the engine, which is compatible with your asset/plugin, and then migrate it to the new version and fix the possible issues. We will go through this in detail in the following chapters.

Keep in mind that Unreal Engine 5 needs quality hardware features on your target machine to perform rendering and functionalities; so, double-check your system features before installing the engine. It is highly recommended to have more than 200-400 gigabit free space on local hard drives to avoid performance issues when you debug and run your game, when using assets, and when working with git technologies.

Here is list of recommendations and minimum hardware requirement for running Unreal Engine 5 and its standard sample projects on PC target machine:

- Recommended hardware
    - **Operating system:** Windows 10 64-bit
    - **Processor:** 12-core Intel or AMD, 3.4 GHz or faster
    - **Memory:** 64 GB RAM
    - **Video card/DirectX version:** NVIDIA RTX 2080 or AMD Radeon 5700 XT
- Minimum software requirements for running the engine or editor
    - **Operating system:** Windows 10 64-bit
    - **Memory:** 32 GB RAM
    - **Video card/DirectX version:** 8 GB of VRAM like NVIDIA GTX 1080 or AMD RX Vega 64

After you press **`Install`** for any version of engine, you will be prompted to customize the folder and path of your installation, as shown in *Figure 1.6*:



Choose install location

Folder:

Path:

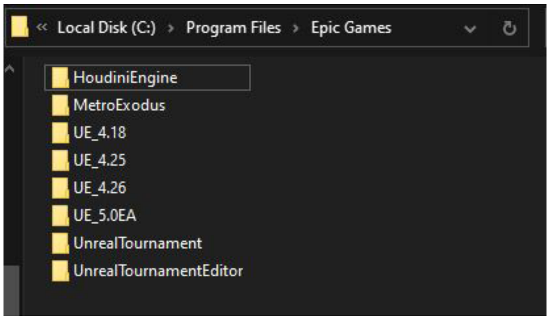
Create Shortcut

Browse | Options | Install | Cancel

Figure 1.6: After pressing "Install", you will be navigated to the install page

Keep in mind that the **Folder** field represents a common folder that belongs to Epic Games applications. The **Path** field represents the folder of your selected engine version.

For example, *Figure 1.7*, as you see, has four different versions of engine plus Houdini and Unreal Tournament requirements, all located in the **Epic Games** folder, which is present in the **Folder** field on the installation page:

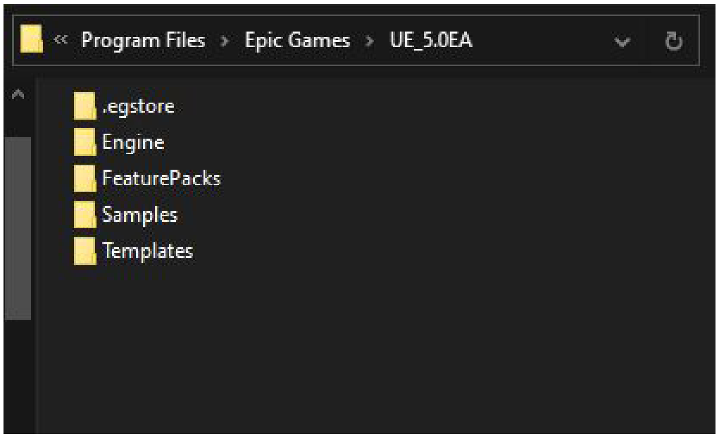


Local Disk (C:) › Program Files › Epic Games

HoudiniEngine
MetroExodus
UE_4.18
UE_4.25
UE_4.26
UE_5.0EA
UnrealTournament
UnrealTournamentEditor

Figure 1.7: Example of having different version of engine inside "Epic Games" folder

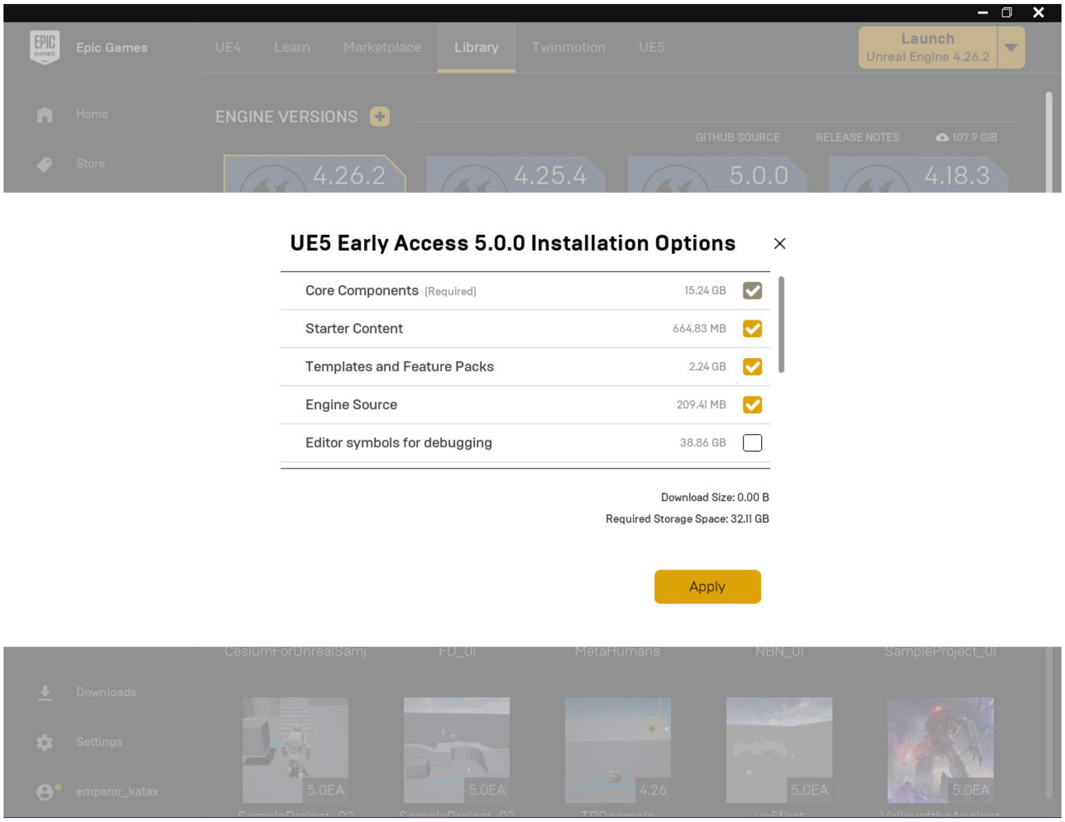
Each engine folder represents the **Source** of a specific version of Unreal Engine, and when you make a new project, the engine makes an exact copy of necessary files from this folder and makes a new folder and project for work in your selected path. We will go through this process in the following chapters, but keep in mind that any custom change in this folder and its contents may bring risk and instabilities to your project, so don't touch the contents of engine folders. *Figure 1.8* illustrates the default folder structure of Unreal Engine 5:


« Program Files › Epic Games › UE_5.0EA

.egstore
Engine
FeaturePacks
Samples
Templates

Figure 1.8: *Unreal Engine 5 default folder structure*

After you set the installation path, click on the **Option** button to navigate to the option page. In the page shown in *Figure 1.9*, you can customize your engine installation based on your project features. For example, if we have an Android device like a mobile phone, and our project is mobile game development, then the engine has to use a number of extra resources to compile and debug your game during development. So, you have to tick **Android** in the **Target Platform** section to get essential support for the engine. On the other hand, imagine that you have a simple PC project, and it doesn't need Android support. Here, you can deactivate android support and save local drive space.



UE5 Early Access 5.0.0 Installation Options

Core Components (Required)	15.24 GB	✓
Starter Content	664.83 MB	✓
Templates and Feature Packs	2.24 GB	✓
Engine Source	209.41 MB	✓
Editor symbols for debugging	38.86 GB	☐

Download Size: 0.00 B
Required Storage Space: 32.11 GB

Figure 1.9: Unreal Engine 5 option page

There is a list of options on this page, which is as follows:

- **Core components**: Engine native libraries, editor tools and essential components
- **Starter content**: Premade multiple assets to help development and design prototypes
- **Template and feature packs**: Provide numbers of premade templates based on your project like first person shooter or VR; we will grab and use the `ThirdPerson` template during this book to learn and debug our C++ code
- **Engine source**: Complete source code of engine written in C++ programming language and accessible via Visual Studio or any C++ compatible IDE
- **Editor symbols for debugging**: Add extra libraries and symbols for debugging
- **Target platforms**: List of drivers of target machines

When you finish installing a version of the engine, you can always customize it at any time. As you see in *Figure 1.10*, you can click on the small arrow located on each engine panel and then click on **`Options`** from the dropdown menu:

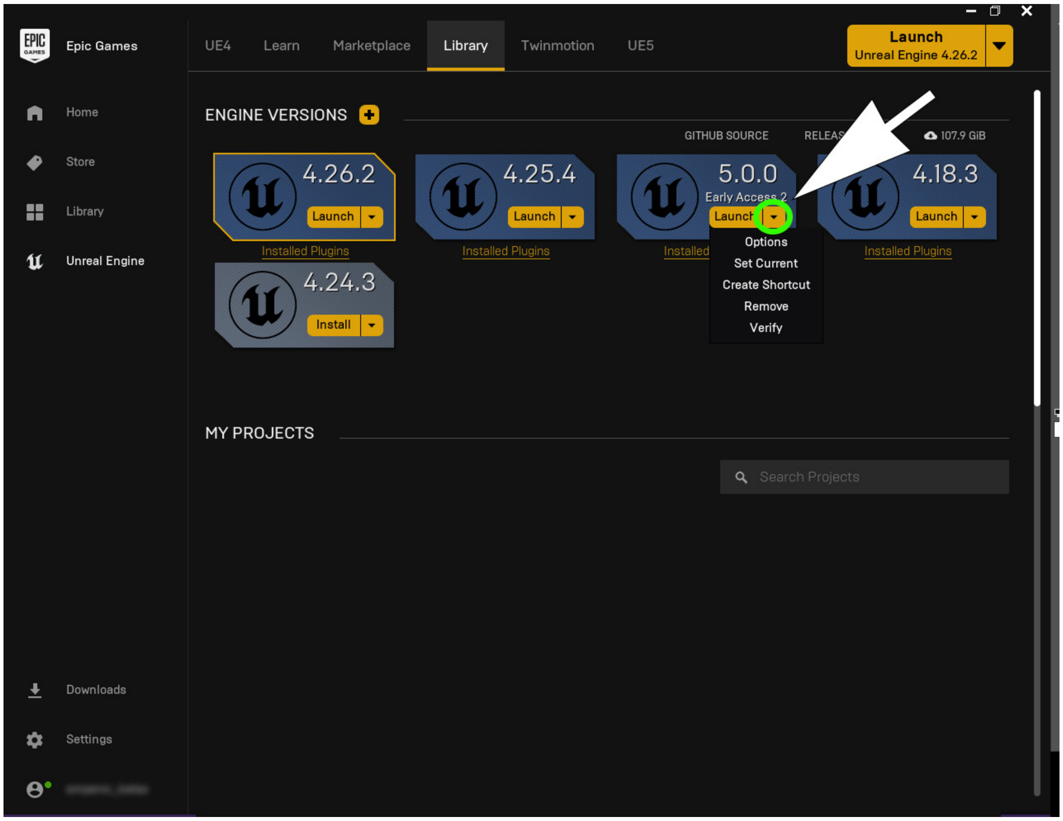


Figure 1.10*: Customize game options after install engine*

For example, imagine that you have to make a new project that supports HoloLens, and when you install the engine, you untick the HoloLens option there. In this case, you can tick the HoloLens in **`Target Platforms`**, and the launcher will automatically install the related resources.

The drop-down menu also has following options:

- **`Set Current`**: Select this version of engine as the default version to launch, which will be represented at the top-right side of your launcher screen, as shown in *Figure 1.11*:

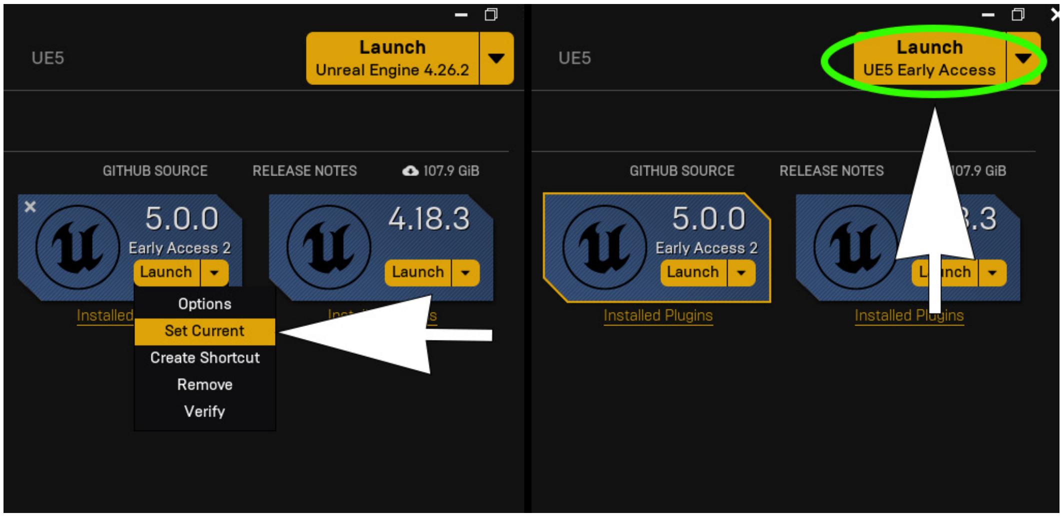


***Figure 1.11**: Set engine version as default for launch*

- **`Create Shortcut`**: Make a shortcut of this version and add it to your desktop for a quick launch.
- **`Remove`**: Uninstall the engine with all its components.
- **`Verify`**: After installation, you can verify your installed file in case of custom changes to the local source or missing files.

Conclusion

During the past decades, Epic Games had a massive impact on the industry in the case of designing and developing video games on multiplatform. They created their own game engine, known as *Unreal Engine*, and released it as a free application to download and use by the public. The qualities and multi-platform approaches of their game engine resulted in brilliant video games and huge support by game designers and programmers after the millennium. The *Unreal Engine 5* is the latest version of Unreal Engine, which has a major improvement on mesh processing, dynamic lights, and audio programming. For using it, check your system preferences, make an account in Epic Games online portal **https://store.epicgames.com/en-US/**, and then install Epic Game Launcher. This application is designed to organize and support your game design projects with Unreal Engine. From the launcher, you can install a number of versions of the engine and also customize their options.

We will go through making projects and using game templates in the next chapter.

Points to remember

- By default, you can use Epic Game Launcher to install multiple versions of Unreal Engine.
- You can customize and remove each version of Unreal Engine by using launcher menus.

Multiple choice questions

1. **How can you add or remove versions of Unreal Engine on my computer?**
    a. Using Epic Game Launcher and navigating to the Library tab
    b. Using Epic Game Launcher and navigating to the Marketplace tab
    c. Using Visual Studio Installer and navigating to the Workflow tab
    d. Using Epic Game Launcher and navigating to the Unreal Engine tab

Answer

1. **a**

Questions

1. Is that possible to install multiple versions of Unreal Engine and use them on the same machine?
2. As a default installation, how much space do you need on your hard drive to install Visual Studio and Unreal Engine together?

Key term

- **Epic Game Launcher** is the application that installs and customizes engine versions and gives access to other Epic Game online services like Marketplace. You have to make an account and profile through Epic Game home page, i.e., **https://www.epicgames.com/**, before using this application.

CHAPTER 2 Math for Game Design

Making computer games meet using programming skills invents a harmony of visuals and sound effects, which interact with players based on their inputs. Players can interact with games via keyboard and mouse or any other controllers like VR set or console. Everyone knows that player interactions with game is a human factor, so as a beginner, they may have numbers of mistakes, like randomly/ accidently clicking on buttons or pushing the wrong button. After a player becomes an expert in a game, their interaction with the game gets more mature and accurate on time. So, how we can make a bridge between human input as a player and the video game we designed in order to make a good gaming experience for player either as a beginner or an expert? What is the secret of making a harmony of visuals and assets that supports the *Fun* of gaming?

The answer is hidden in two main factors of the game: the quality of design and the efficiency level of the code. In this chapter, we will go through essential knowledge of *mathematics*, which any game developer needs to develop code for game. Mathematical approaches are an important part of any game development. They can support game mechanics, tune shaders, and increase performance. Mathematical functions and procedures are involved in creating material, playing animation, running special effects, supporting logics in inventory system, simplifying user interactions, calculating delays, and much more. Having advanced knowledge of mathematics and numeric in programming will always give accurate results and efficient qualities in any video game project.

Structure

In this chapter, we will discuss the following topics:

- Basic mathematics
- Using mathematics in programming game
    - Vector
- Mathematical functions

Objectives

After studying this unit, you will learn the basics of mathematics in game development by going through numeric and Boolean data types and learn *how* and *where* to use mathematical operations on these basic data types. After this, you will learn how to use Vector and its operations and formulas, which is great knowledge on each game development scenario. At this stage, you are ready to learn more complex mathematical topics and as planned, you will learn essential knowledge of Trigonometry and functions related to it. Unfortunately, there is not enough space in this chapter to cover all aspects and details of Trigonometry, but we will cover a good base for your future research and studies on this important topic in math and game development.

We will also cover distance and normalize at the end of this chapter.

Basic mathematics

Each data in computer needs space in the memory to store its value, and it must be connected to a variable in your code in order to accessed it`s values by your application. On each game application, you will find numbers of variables that are responsible to store various kind of data or better say, *Data Type* in memory. There is a long list of data types in Unreal Engine 5, and we will cover most of them and their relation to variables over the following chapters, but for now, we like to get an overview on mathematical procedures, so let’s focus on numeric and Boolean data types that are commonly used for making variables in order to perform mathematical operations and functions in game.

The *Numbers* are represented by numeric data types, and the *Booleans* are represented by Boolean data types. A variable made from a numeric data type is simply a *signed number*, and you can use it for making health bar of player, damage of player weapon, enemy distance from others or the color values of friendly unit’s armor. All these examples use a *signed number* to present data to player.

A variable made from Boolean data type represents *True* and *False* and that's all! These two values (true and false) are commonly used to present binary logic of game, like decision-making, or switch between different tasks, for example, turn on night vision view from the main menu in a shooter game or defuse bomb in counter strike (as shown in *Figure 2.1*) and win the round. All these are examples of using a Boolean variable to switch or activate relative functions in the game:

***Figure 2.1**: Defuse bomb in Counter Strike is basically a Boolean procedure that affects the state of the game*

As we mentioned, numeric data types are simply numbers. They are implemented by three main data types in Unreal Engine 5: **`Integer`**, **`Float`**, and **`Byte`**. Integer variables are **`whole`** numbers and can be positive, negative or 0; examples are 12, -7, and 4000. Float numbers are like integers but also have an attached decimal value; examples are 0.025, 7.44, and -5.4. Numbers like 2.0 and -4.0 represent float numbers because they have an attached decimal value, which is **`.0`**. Byte variables can hold a positive whole number between 0 to 255; examples are 0, 64, 110, so numbers like 400 or -2 can **`NOT`** be implemented and hold by Byte data type.

All integer, float and byte data types have basic operators for add, subtract, multiply, divide, and modulo, which are described in *Table 2.1*:

Name	Sign	Output result
Add	+	Add two values with each other
Subtract	-	Subtract one value from another
Multiply	*	Multiply two values
Divide	/	Divide one value by another
Modulo	%	Return the remainder or signed remainder of a division

***Table 2.1**: Basic Numeric operators*

Boolean data types are simply accepting **`True`** and **`False`** as value, and they are implemented by just one type known as **`Boolean`**. Sometimes these values are referred to **`0`** (stands for **`False`**) and **`1`** (stands for **`True`**), which are numeric representations of a Boolean data type. Boolean data types have a number of operators, which are shown in *Table 2.2*. These operators are easy to learn, but as a beginner, you have to understand *when* and *how* they can be used to simulate the logic of your game. We will go through simple examples of basic Boolean operators used in Unreal Engine 5:

Name	Output result
NOT	Returns the logical complement of the Boolean value
AND	Returns the logical AND of two values
OR	Returns the logical OR of two values
XOR	Returns the logical eXclusive OR of two values

***Table 2.2**: Basic Boolean operators*

Boolean operator AND

This operator returns **`True`** if both comparisons are **`True`**, as shown in *Table 2.3*:

Variable A	Variable B	Output Result
True	True	True
True	False	False
False	True	False
False	False	False

***Table 2.3**: Boolean AND operator*

Imagine that you are designing logic for a space simulation game. As you know, each human logically needs **`Air`** to breath and **`Water`** to drink to stay alive. So, how can we develop such a logic in our game by using Boolean? First, imagine that **`Air`** and **`Water`** are two Boolean variables like **`A`** and **`B`**, which can hold **`True`** or **`False`** as value. Now, if **`A`** is **`True`**, it means we have **`Air`**, and if it is **`False`**, it means we don't have **`Air`**, and it is the same for **`Water`** with variable **`B`**. Now, let's make some logical rules:

- No Air and no Water = Player cannot survive
- No Air and enough Water = Player cannot survive
- Enough Air and no Water = Player cannot survive
- Enough Air and enough Water = Player can survive

Now, let us replace Air and Water with variables **`A`** and **`B`**, and simplify our logic:

- A is False and B is False = Cannot survive
- A is False and B is True = Cannot survive
- A is True and B is False = Cannot survive
- A is True and B is True = Can survive

As you see, the **`AND`** operator is the best match to simulate this logic in the game.

Boolean operator OR

This operator returns **`True`** if at least one comparison is **`True`**, as shown in *Table 2.4*:

Variable A	Variable B	Output Result
True	True	True
True	False	True
False	True	True
False	False	False

Table 2.4: *Boolean OR operator*

Imagine that you are designing the logic for a city-building strategy game. Each city has a unique building that supplies electricity. This building needs either water or gas as the main resource to generate electricity. In case of having no water and gas together, the building cannot supply electricity. So, again, how can we develop such a logic in our game by using Boolean? First, imagine that **`Gas`** and **`Water`** are two Boolean variables like **`A`** and **`B`** that can hold **`True`** or **`False`** as value. Now, if **`A`** is **`True`**, it means we have enough **`Gas`**, and if it is **`False`**, it means we don't have **`Gas`**, and it is the same for **`Water`**, which is represented by variable **`B`**. Now, let us write some logical rules:

- No Gas and no Water = Building cannot produce electricity
- No Gas and enough Water = Building can supply electricity
- Enough Gas and no Water = Building can supply electricity
- Enough Gas and enough Water = Building can supply electricity

Now, let us replace **`Air`** and **`Water`** with variables **`A`** and **`B`**, and simplify our logic:

- A is False and B is False = No electricity
- A is False and B is True = Having electricity
- A is True and B is False = Having electricity
- A is True and B is True = Having electricity

As you see, the **OR** operator can simulate this logic for the building in the game.

Boolean operator XOR

This operator returns **True** if at least one comparison is **True**, as shown in *Table 2.5*:

Variable A	Variable B	Output Result
True	True	False
True	False	True
False	True	True
False	False	False

Table 2.5*: Boolean OR operator*

A very simple example of using **XOR** to simulate logic is to check the possibility of pregnancy between two individuals based on gender type. Assume that variable A and B are two individuals; here is the logic:

- A is male and B is male = pregnancy is NOT possible
- A is female and B is male = pregnancy is possible
- A is male and B is female = pregnancy is possible
- A is female and B is female = pregnancy is NOT possible

So, simply like the previous example, you can assign **True** and **False** to represent gender and then use **XOR** to simulate pregnancy logic based on gender in any application.

Boolean operator NOT

This operator returns the inversed value or opposite response of a Boolean variable. This means if a Boolean variable is **True**, the **NOT** operator will change it to **False** and if it is **False**, this operator will switch it to **True**, as shown in *Table 2.6*:

Variable A	Operator	Output Result
True	NOT	False
False	NOT	True

Table 2.6*: Boolean NOT operator*

Using mathematics in programming game

As you know, each video game has a number of 2D or 3D assets in the scene. These assets follow logical or physical rules to move, rotate, and affect other objects in the scene when the player runs the application and plays the game. Mathematics is a vital part of this experience because by using it, developers can address physical properties of each asset and control logical procedures during game play. So far, we learned about numeric and Boolean variables, and the operators belong to each group. These variables are the fundamental elements of special kinds of objects and functions that we like to describe and review in this section. By using these functions and objects, you can simulate geometric calculations, physics, and mechanics in your game. These objects and functions are common in all game development technology, so using Unreal Engine, Unity or CryEngine, you can use similar objects and math functions to develop your game.

An example of mathematical object is matrices, which is a rectangular arrangement of numeric or Boolean data types into rows and columns. Matrices are useful to solve linear equations and geometric conversions. Another example is a very common mathematical object known as Vector, which is made by numeric data types and wildly used in game development. *Figure 2.2* represents 3 x 3 matrices called **A** on the left, which has 9 members, and there is a vector on right side, called B, that looks like an arrow:

$$A = \begin{bmatrix} 1.414 & 1.732 & 2.236 \\ 0 & 1 & 0.783 \\ -1 & 1.618 & 2.718 \end{bmatrix}$$

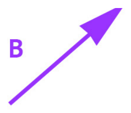


***Figure 2.2**: Left- representation of Matrices (Right -representation of Vector)*

Vector

Vector is one of the most important mathematical objects used in each game application to commonly represent location, direction, or velocity of an object in the game scene. Each vector always has two informative factors: *direction* and *magnitude or size*. You can imagine a vector as a directed line with one start point, and an end point with an arrow, as shown in *Figure 2.3*:

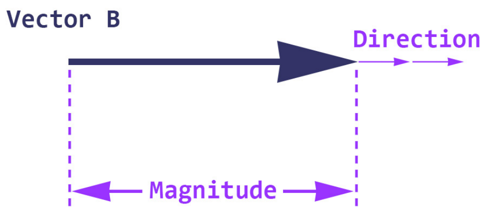


***Figure 2.3**: Vector B magnitude and direction*

In nutshell, vector is like an array made of two or more numeric data types, which are known as **Vector's Elements**. Each vector element can be defined as a variable of **`Int`**, **`Float`**, or **`Byte`** numeric data type. Unreal Engine 5, supports vector variables in three variations, which are shown in *Table 2.7*:

Vector type	No. elements	Representation	Magnitude
Vector 3	3	(x,y,z)	$\sqrt{}(x^2 + y^2 + z^2)$
Vector 2	2	(x,y)	$\sqrt{}(x^2 + y^2)$
Vector 4	4	(x,y,z,w) or (r,g,b,alpha)	$\sqrt{}(x^2 + y^2 + z^2 + w^2)$

Table 2.7*: Vector variables support in Unreal Engine 5*

Vector 2 has two elements and is used in 2D calculations to present numeric values of X and Y coordination's of an object in the scene. *Figure 2.4* shows an example of using vector 2 to present location of 2D game objects:

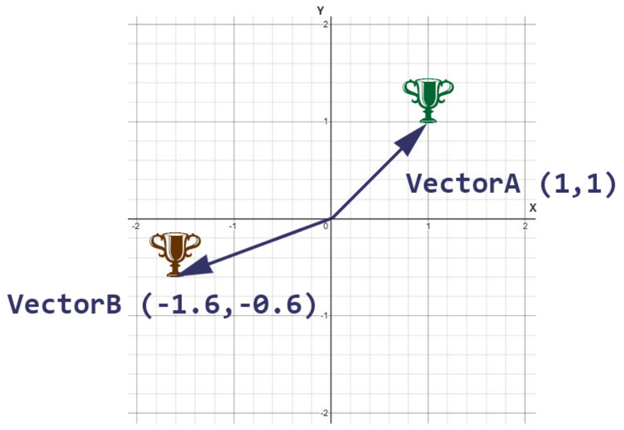


Figure 2.4*: Green cup is located at X = 1 and Y=1 and Brown cup is located at X=-1.6 and Y=-0.6 in a 2D map. Vector A uses these values as its elements to presents 2D location of the object. Follow the same rules, Vector B (-1.6, -0.6) shows the location of Brown cup at X=-1.6 and Y=-0.6.*

Vector 3 has three elements and is used to present 3D coordination of geometrics objects in the scene. *Figure 2.5* shows a 3D representation of a point in the space:

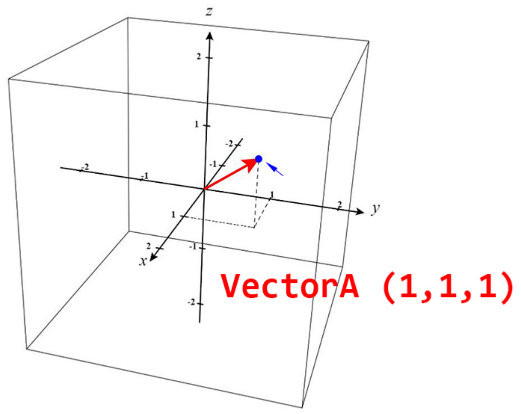
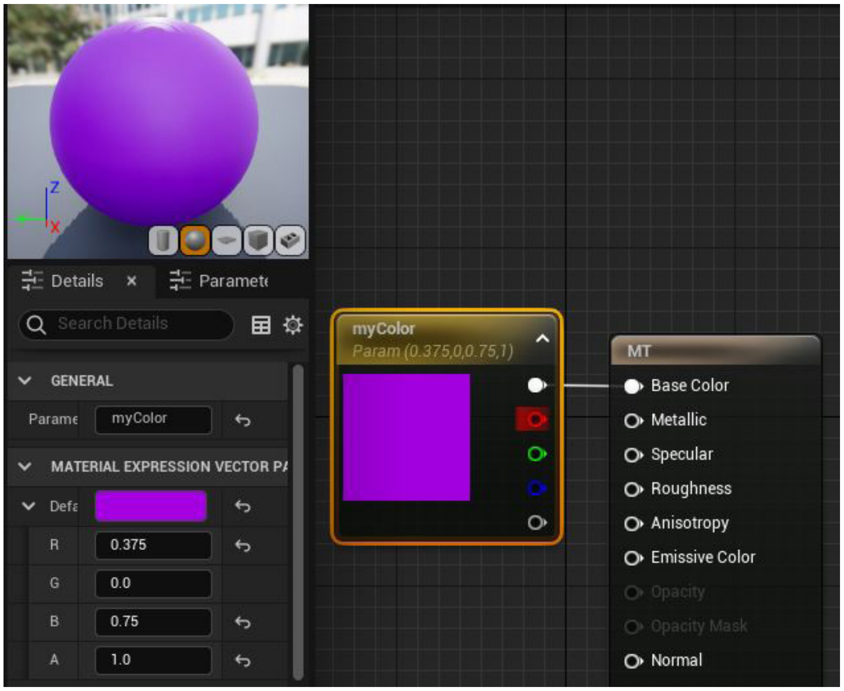


Figure 2.5: Blue dot is located at X = 1 and Y=1 and Z=1. Vector A uses these values as its elements to presents 3D location of the object. So, Vector A (1, 1,1) shows the location of the Blue dot at X=1 and Y=1 and Z=1.

Vector 4 is heavily used in Shader programming. Each element of this vector represents a color: **R** for Red, **G** for Green, and **B** for Blue. The last element of Vector 4 represents alpha or transparency. *Figure 2.6* is an example of using vector 4 to create a color in material in Unreal Engine 5; we will cover this in detail in the following chapters:



Parame	myColor
Defa	🟪
R	0.375
G	0.0
B	0.75
A	1.0

Figure 2.6: Using a Vector 4 variable (myColor) with (0.375, 0.0, 0.75, 1.0) as element, which are addressing Red, Green, Blue, and Alpha channels to create material color in Unreal Engine 5.

You can use math operators on vectors like numeric data type. All types of vectors support operators add and subtract like numeric data types, and you can use operator multiply and divide on a vector to multiply or divide it by scalar value. In addition, vectors have one unique operator known as **Dot Product**, which is commonly used in shader code and creating VFX functions. Now, let's go through each operator and explain how they work.

Vector operator add

For adding two vectors, you need to add similar vector elements with each other. Assume that vector A (2, -4) and vector B (8, 6) represent *x* and *y* of two locations in the scene. Now, if you like add these two vectors, you need to add X elements with each other and the same for Y elements, so A + B = (2 + 8, -4 + 6) = (10, 2).

Vector operator subtract

For subtracting two vectors, you need to subtract similar vector elements from each other. Let's use the previous example: consider vector A (2, -4) and vector B (8, 6) of two locations in the scene. Now, if you subtract these two vectors, you need to subtract X elements from each other, and the same for Y elements, so A - B = (2 - 8, -4 - 6) = (-6, -10).

Figure 2.7 shows a visual representation of add and subtract two vectors in a 2-dimensional map:

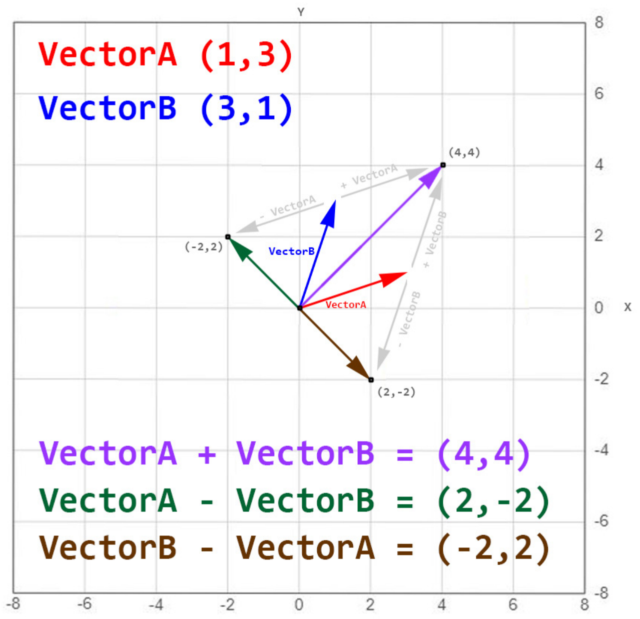


***Figure 2.7**: Visual representation of add and subtract operation on two Vector 2*

Vector operators multiply and divide by scalar

For multiplying a scalar value to vector, you need to multiply it to each element of vector. Imagine that *Vectors A (2, -4, .5)* represent *x*, *y* and *z* of a 3D object in the scene. This object has a distance from origin of map (remember that vectors have magnitude and direction), now we like to *move* it in the same direction. So, you can multiply or divide each element of vector by a scalar number and change the magnitude of vector. For example, if we multiply *Vector A* by 10, it means we move the object to a new location far from first one, and the new location will be as follows:

$$10 \times \text{Vector } A = (10 \times 2, 10 \times -4, 10 \times .5) = (20, -40, 5) \text{ new coordination.}$$

Now, to move the object closer to the origin, you can divide it by a scalar, and it works like multiplying. For example, if we like to bring the object 100 times closer to the origin, we can divide each element of vector by this number:

$$\text{Vector } A / 100 = (2/100, -4/100, .5/100) = (.2, -0.040, .005) \text{ new coordination}$$

Vector operator Dot Product

The *Dot Product* is a scalar value that is simply a number. Assume that *Vector A* and *Vector B* are locations of two items in a 2D map, as shown in *Figure 2.8*. Now, we can calculate *Dot Product* by multiplying similar elements of each vector and adding them to each other with the following formula:

$$A \cdot B = A_x \times B_x + A_y \times B_y = 2 \times 6 + 6 \times 4 = 36$$

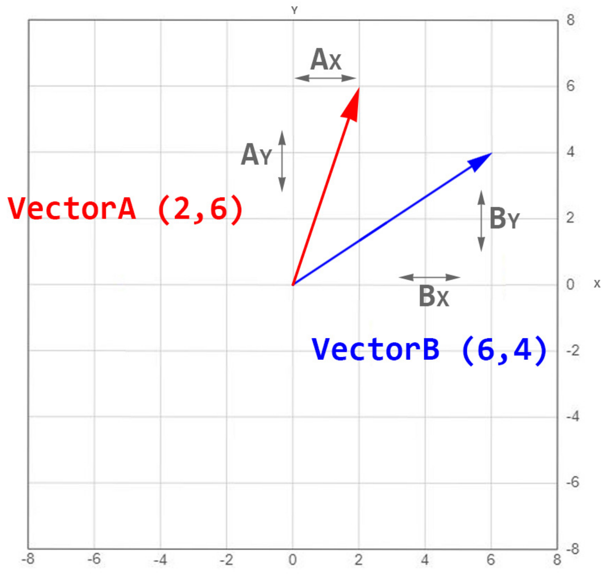


***Figure 2.8**: Visual representation of elements of Vector A and B*

You may be surprised at why such a number needs special operator to extract. This Dot Product shows features of vectors **A** and **B** and the angle between them:

- When vectors *A* and *B* have a 90-degree angle between them, the Dot Product is always **zero**.
- When vectors *A* and *B* have a 0-degree angle between them, the Dot Product is equal to multiply magnitude of each vector to other.
- To calculate the Cosine of angle between vectors *A* and *B*, you need to calculate their Dot Product first, and then divide that by the value of multiplying the magnitude of each vector with the following formula:

  $A \,.\, B = \textit{magnitude of } A \times \textit{magnitude of } B \times \cos\,(\textit{angle between } A \textit{ and } B)$

Figure 2.9 shows an example of finding angles between vectors by using Dot product:

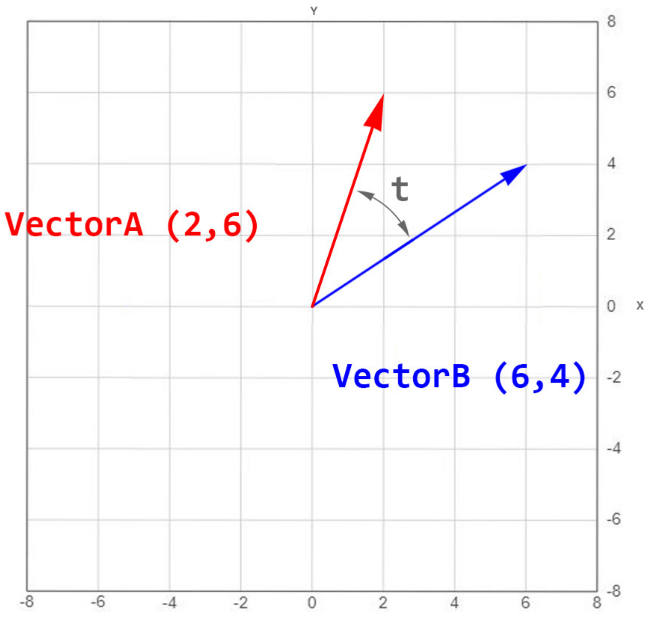


***Figure 2.9**: Vector A and B and the angle "t" between them*

By using formula for vector, A (2, 6) and vector B (6, 4):

$$A \cdot B = \textit{magnitude of } A \times \textit{magnitude of } B \times \cos\,(\textit{angle between } A \textit{ and } B)$$

Then we have:

$$36 = 6.32456 \times 7.2111 \times \cos(t) \textit{ which means } \cos(t) = 0.789$$

The angle *t* will be 37.8 degree.

Mathematical functions

Alongside mathematical objects like Vector, you need to use mathematical functions to simulate complex scenarios. For example, if you want to calculate power of a number or normalize a Vector, mathematical functions like Power and Normalize will be used. Unreal Engine 5 has in-depth collections of functions to express and use math for geometric, physical, and shader simulations. Here, we go through essential math functions that you must know in any game development scenario. In the following chapters, you will learn how to implement these functions into your code, but before that, you must know what they do and where to use them.

Math function: power

This function multiplies your base number by itself based on your multiplier value. For example, let's use 4 as base number and 2 as multiplier, so $4^2 = 4 \times 4 = 16$ and $3^4 = 3 \times 3 \times 3 \times 3 = 81$.

Math function: Pi

This function returns constant value for Pi number as 3.141592.

For example, the area of a circle can be found with the formula $Area = \pi \times radius^2$, surface area of sphere can be calculated by $Area = 4 \times \pi \times radius^2$, and the formula for converting degree to radian uses Pi as well, which is $degree° = \pi \times 180 \times radians$. There are several other formulas in geometry and physics that meet using Pi as a constant parameter.

Math function: ABS

This function returns the *Absolute* value of input by removing its negative sign. In case of having positive numbers as input, it returns them with no change. For example, imagine that you are a UI programmer of a city building strategy game and need to show the level of air pollution on players' monitors. You already have a variable, and if it goes below -40, an alarm message should come up and give the player information. So, by using ABS, you can simply change *-40* to *40* and print a message like *The pollution is 40% below Standard level!*

Math function: floor

Using this function will *remove* the fraction from float numbers, so it returns *0* for *0.2* and *4* for *4.2*. So, easy to remember, Floor only remove fraction from float numbers.

Math function: cell

Using this function, like floor, will *remove* fraction from float numbers, but it *rounds up* the number in output, so it returns *1* for *0.2* and *5* for *4.2*. In short, cell *remove* and *rounds up* fraction from float numbers.

Math function: clamp

This function is used to clamp the input value to the specified minimum and maximum range as output. Suppose you need to develop code to show the positions of all players in a mini map of each player's UI. Inside the game, distance between players can be hundreds of meters, but on the monitor, you only have an area of 240 x 240 pixels. So, a good solution will be using *Clamp to Range* function which converts values of player position, between minimum and maximum values of mini map`s size. That is how magic became easy: by using math functions!

Math functions: sine and cosine

Mathematics is not just Numeric and Booleans, as we discussed for game programming; it has a number of branches. One of the most important and ancient branches of mathematics that has been used for thousands of years is known as **trigonometry**. Trigonometry talks about triangles and their relations. This branch is extremely useful for game design and unfortunately, it isn't in the scope of this book to go through all the details of trigonometry, but as basics, let's learn the right triangle rules.

Any triangle with a 90-degree angle is called a **right triangle**. Each side of a right triangle has a title, as shown in *Figure 2.10*:

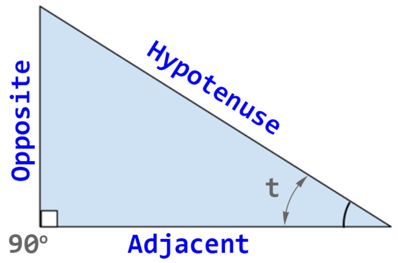


***Figure 2.10**: Right triangle and its side names related to angle "t"*

Here:

- **Hypotenuse**: The side opposite of the right angle, which is 90°
- **Adjacent**: The side next to the angle *t*
- **Opposite**: The side on opposite of the angle *t*

As the rules of trigonometry proved, we can calculate the size of each side of right-angled triangle, by using the ratios and the angle between pair of sides. Assume that we have a right triangle and the size of its hypotenuse is equal to one unit of measurement (for example 1 meter.). We can draw this triangle in a circle, as shown in *Figure 2.11*:

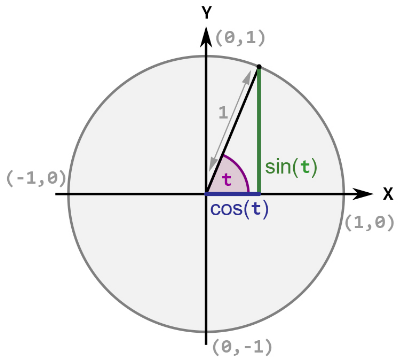


Figure 2.11*: Calculate Sin and Cos of angle "t" based on the right triangle with a hypotenuse of length is 1 unit of measurement.*

The mathematical relation between angle *t* and other sides of a right triangle is presented by sine and cosine functions which also known as ratios:

$$sin\ (t) = Opposite \ / \ Hypotenuse$$

$$cos\ (t) = Adjacent \ / \ Hypotenuse$$

Sine and cosine are functions of angle and represent a ratio between the sides of a right triangle. Each angle returns a numeric float value for sine and cosine, and as the preceding image shows, the value is between the range of 1 and -1. Sine and cosine values are commonly used in engineering, physics and electronics, and you can find the sine and cosine buttons on each calculator. Now, use values between 0 – 360 for angle and draw the output graph of sine and cosine, probably by using a calculator or a drawing program (or even a table of sine and cosine from 100 years old book). You will end up with the same image as *Figure 2.12*:

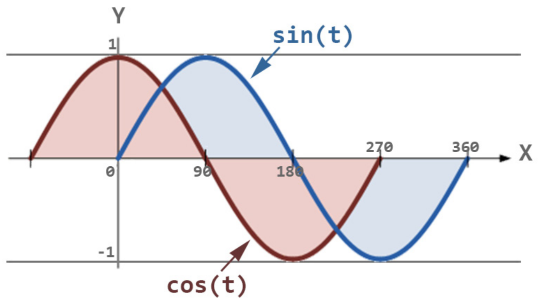


Figure 2.12*: Visual representation of Sin(t) and Cos(t) where "t" is between 0 to 360*

As the image shows, the values on the X axis represent the angle *t*, and they are between 0 and 360, and values on the Y axis represent the output of the sine and cosine functions, and the rate of change is between 1 and -1. This behavior in output makes sine and cosine ideal functions to simulate movement and vibration. When we go through shader code in the following chapters, we will use these values to make a blinking light visual effect in game by using "Material Editor" of the engine.

Math function: distance

Suppose you need to calculate the distance between Black cup and Green cup objects in the scene shown in *Figure 2.13*. For this, we need to use Vector as the position of each object and use the following formula:

$$\textit{Distance 2D between Vector A and B} = \sqrt{(B_x - A_x)^2 + (B_y - A_y)^2}$$

Now, let's use numbers and calculate the distance:

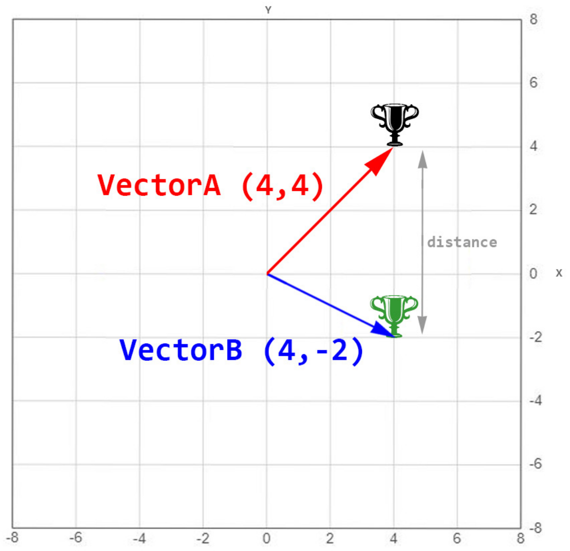


***Figure 2.13**: The graph shows that distance between objects is 6*

$$\textit{Distance between Vector A and B} = \sqrt{(4 - 4)^2 + (-2 - 4)^2} = \sqrt{0 + 36} = 6$$

As you see, the result from formula is an exact match to the preceding image. We can use this formula to calculate distance in 3D scenes:

$$\textit{Distance 3D between Vector A and B} = \sqrt{(B_x - A_x)^2 + (B_y - A_y)^2 + (B_z - A_z)^2}$$

Also, by increasing the elements, you can use it for checking the distance between 4-dimensional vectors, but such a geometric coordination is unusual; perhaps you can assign the fourth element in these vectors to represent time:

Distance 4D between Vector A and B $= \sqrt{(B_x - A_x)^2 + (B_y - A_y)^2 + (B_z - A_z)^2 + (B_t - A_t)^2}$

Math function: normalized vectors

When a vector gets normalized, it's magnitude will be changed to 1, and it saves its direction. You can calculate the normalization of a vector by dividing each element to the magnitude of vector. For example, for getting normalize of a 2D and 3D vector, we can use these formulas:

Normalize 2D Vector A $= (A_x / \sqrt{A_x^2 + A_y^2},\ A_y / \sqrt{A_x^2 + A_y^2})$

Normalize 3D Vector B $= (B_x / \sqrt{B_x^2 + B_y^2 + B_z^2},\ B_y / \sqrt{B_x^2 + B_y^2 + B_z^2},\ B_z / \sqrt{B_x^2 + B_y^2 + B_z^2})$

Conclusion

Programming computer games is heavily dependent on mathematical operations and logics. Depending on the complexity of a game scenario, programmers use different types of mathematics. For example, for calculating the location and rotation of object in the scene, either 2D or 3D, you have to know and use vector operations and functions. Also, in the case of dynamics and physical behaviors, you may need to use functions and expressions related to trigonometry. Other examples are using logical variables to switch between game states like switching level or change animations type.

The *Unreal Engine 5* provides a series of mathematical functions and procedures to use with game mechanics and also shaders. In the next chapter, we will use math to acauline built the game, and this chapter will be a reference for our scenarios and solutions while developing a game.

Points to remember

- Numeric data types are numbers like **`Int`**, **`Float`**, and **`Byte`**.
- Boolean data type is a bool type variable with only 2 values: True and False.
- You can customize and remove each version of Unreal Engine by using launcher menus.

Multiple choice questions

1. **What is Dot Product?**
    a. Dot Product is a Vector operation that results in a number.
    b. Cosine of angle between Vector A and Vector B is equal to Dot Product of A and B, divided by the result of magnitude of A multiplied to the magnitude of B.
    c. When Dot Product results in zero, it means that vectors have a 90-degree angle between them.
    d. Answers a, b and c are correct.

2. **Which sentence is correct about a right triangle?**
    a. Right triangle is a triangle that has a 90-degree angle.
    b. Cosine of angle in right triangle is equal to the size of adjacent divided by hypotenuse, and sine of angle is equal to the size of opposite divided by hypotenuse.
    c. Sine and cosine return value between -1 and 1 for angles between 0 and 360 degrees.
    d. Answers a, b and c are correct.

3. **How do you calculate the distance between two objects in the scene?**
    a. By using the formula $\sqrt{((B_x - A_x)^2 + (B_y - A_y)^2)}$, where A and B represent locations of object as vector, we can calculate the distance between objects in 2D map.
    b. By using the formula $\sqrt{((B_x - A_x)^2 + (B_y - A_y)^2)}$, where A and B represent locations of object as vector, we can calculate the distance between objects in 3D map.
    c. By using the formula $\sqrt{((B_x - A_x)^2 + (B_y - A_y)^2 + (B_z - A_z)^2)}$, where A and B represent locations of object as vector, we can calculate the distance between objects in 3D map.
    d. Answers a and c are correct.

Answers

1. **d**
2. **d**
3. **d**

Questions

1. Refer to **www.mathsisfun.com/algebra/vectors-cross-product.html**; what is the main difference between "Dot Product" and "Cross Product" when we deal with vectors?
2. How can you calculate the tangent of an angle in a right triangle?
3. What is logarithm in mathematics?
4. What is a rotation matrix?

Key term

- **Epic Game Launcher** is the application that installs and customizes the engine version and gives access to other Epic Game online services like Marketplace. You have to make an account and profile through the Epic Game home page, that is, **https://www.epicgames.com/**, before using this application.

Chapter 3
Editor Basics and Epic Launcher

This chapter will cover using the Epic Games launcher to open, update, or repair the engine, and it will walk you through how to use the asset Marketplace it provides. It will also cover the basic standard layout of the editor, how to modify the layout of the editor, where the important menus are located, their purpose, and how to use them. It will also explain useful hotkeys (and reroute nodes) and skim through the project settings and plugins menus.

Structure

In this chapter, we will discuss the following topics:

- Epic Launcher layout
- Editor layout
- Common hotkeys
- Important menu/window
- Project settings
- Plugins

Objectives

After studying this chapter, you should be able to check different sections of Epic Game Launcher related to create project or grab assets, learn the common menus and their functionalities in "Unreal Editor." You will also be able to check and understand usage of default panels inside the editor. This chapter will equip you with Using "Project Setting" panel, and you will be able to check, add or remove plugins to the engine.

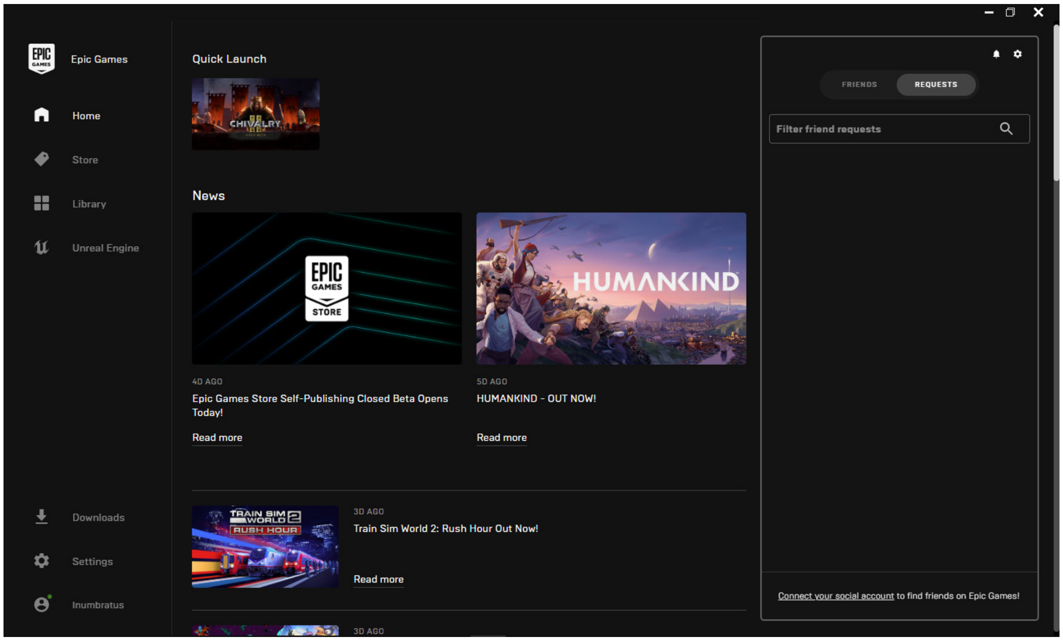
Epic Launcher layout

To start using UE5, you need to install it using the **Epic Games Launcher** (**EGL**), which was covered in *Chapter 1: What Is Unreal Engine?*

Inside the EGL there is a list of tabs in the top left of the Interface. These tabs are discussed in the following sections.

Home

The **Home** page contains news and deals from Epic Games that they want their users to see, as shown in *Figure 3.1*:



4D AGO
Epic Games Store Self-Publishing Closed Beta Opens Today!
Read more

5D AGO
HUMANKIND - OUT NOW!
Read more

3D AGO
Train Sim World 2: Rush Hour Out Now!
Read more

3D AGO

Connect your social account to find friends on Epic Games!

***Figure 3.1**: Epic Games Launcher Home Page*

This page doesn't contain news regarding Unreal Engine; that is displayed elsewhere in the launcher.

Store

The **`Store`** page (shown in *Figure 3.2*) is where you can purchase games to play and manage through the Epic Games Launcher:

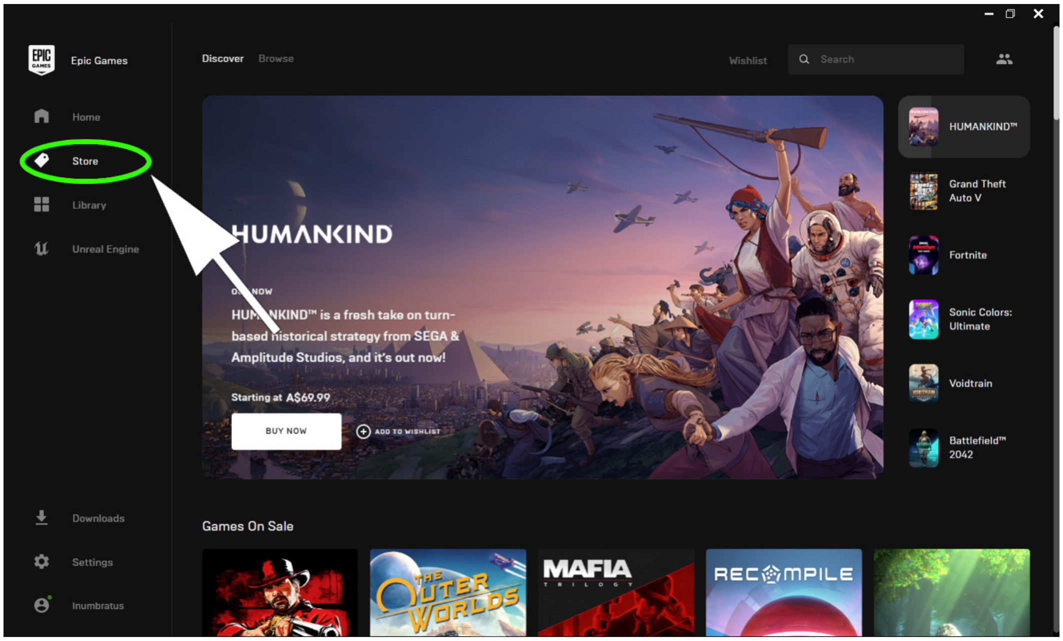


***Figure 3.2**: Epic Games Launcher Store Page*

Epic Games frequently allows users to obtain games (to keep) for free and sell titles that are exclusive to the Epic Games Launcher and are unobtainable through other online game vendors (usually only temporarily before they're available elsewhere).

Library

The **`Library`** page (shown in *Figure 3.3*) is where you can manage games purchased through the Epic Games Launcher. On this page, you can install, uninstall, repair, and launch the games you have access to:

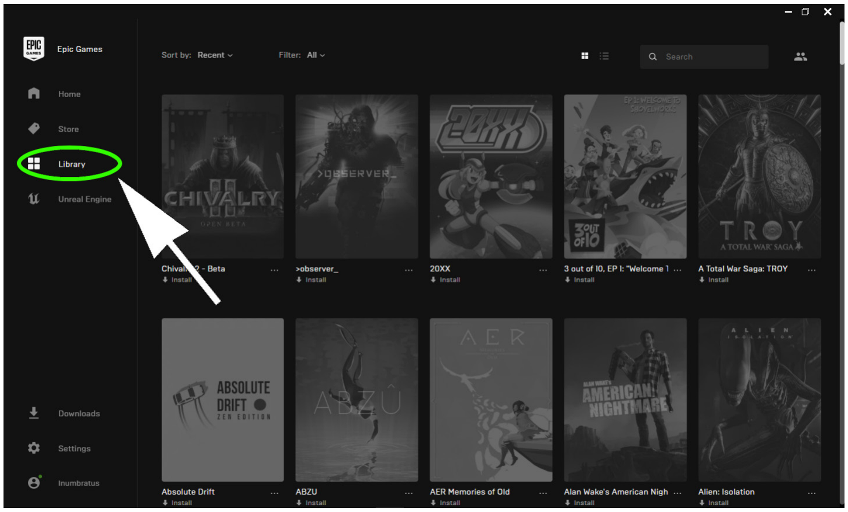
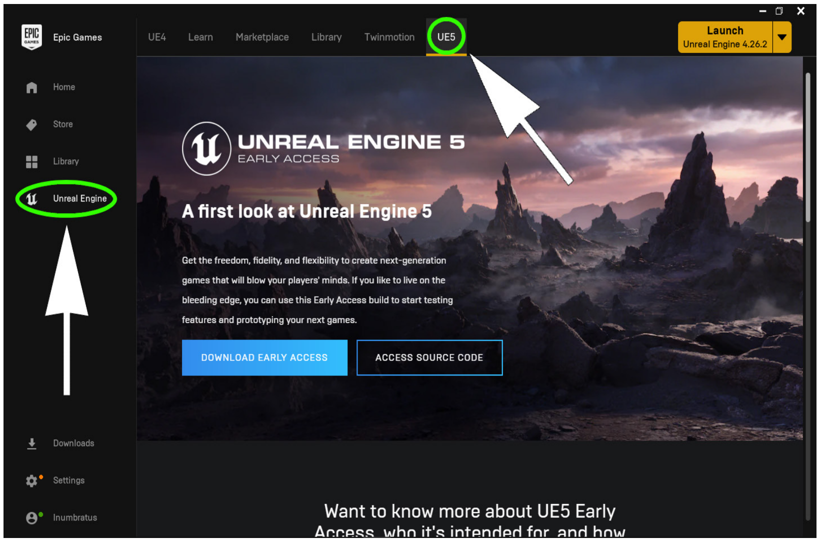


Figure 3.3: *Epic Games Launcher Library Page*

Unreal Engine

This is a page that contains sub-pages related to the Unreal Engine; it is shown in *Figure 3.4*:



A first look at Unreal Engine 5

Get the freedom, fidelity, and flexibility to create next-generation games that will blow your players' minds. If you like to live on the bleeding edge, you can use this Early Access build to start testing features and prototyping your next games.

Want to know more about UE5 Early Access, who it's intended for, and how

Figure 3.4: *Epic Games Launcher Unreal Engine Page, with the UE5 page selected*

It has its own **`Home`** (One for **`UE4`** and one for **`UE5`**), **`Learn`**, **`Marketplace`**, **`Library`**, and **`Twinmotion`** pages. The **`Home`** pages display featured content (related to unreal engine) that Epic Games want their users to see. The **`Learn`** page only contains content pertaining to Unreal Engine 4 currently, and the **`Twinmotion`** page is unimportant to the purpose of this book, so they will not be covered.

Marketplace

The Unreal Engine Marketplace (shown in *Figure 3.5*) contains a vast amount of content that is available to be purchased, downloaded, and used:

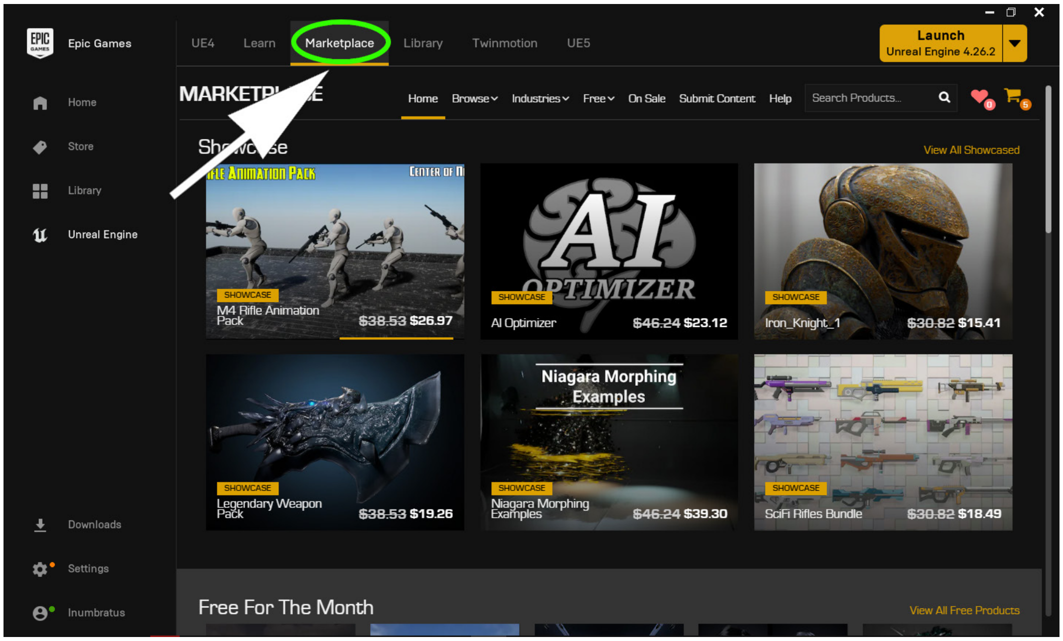


***Figure 3.5**: Unreal Engine Marketplace*

There is free content available as well; Epic Games very generously makes certain content free to keep each month. When using the marketplace, remember to make sure the content you are obtaining is compatible with the engine version(s) you are using.

Library

The Unreal Engine Library contains access to all projects associated with the launcher, all content purchased from the marketplace, and access to modify (update, install, verify, or uninstall) versions of the engine.

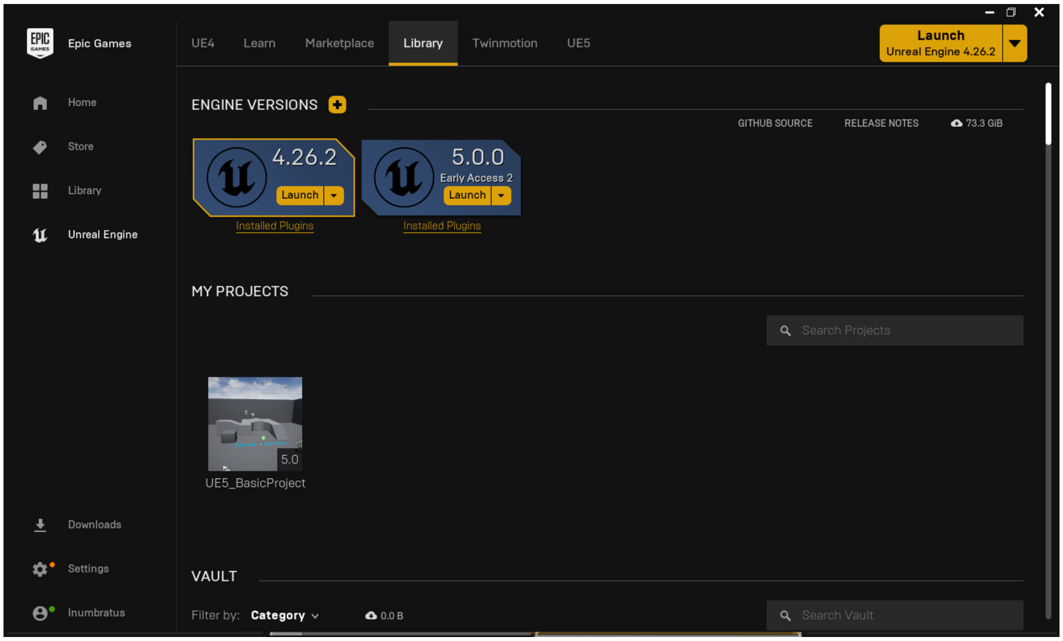
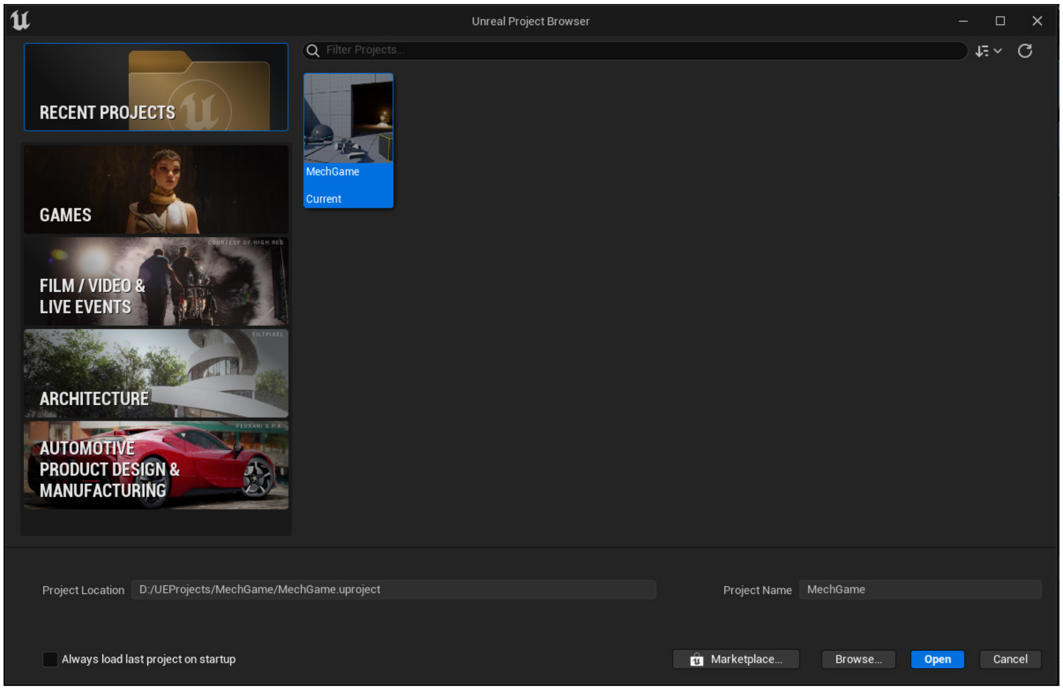


Figure 3.6: *Unreal Engine Library*

Project Browser

Once you open Unreal Engine from the Epic Games Launcher, you will be prompted to create a new project or open an existing one through the project browser, which is shown in *Figure 3.7*:



Project Location: D:/UEProjects/MechGame/MechGame.uproject
Project Name: MechGame
Always load last project on startup

Figure 3.7: *Unreal Engine 5 Project Browser*

Once a project has been made or opened, the editor will load and then open. In the project browser, templates used to create new projects are situated on the left side of the interface, while existing projects are shown on the right.

Editor layout

The Editor's interface is made up of a series of *Panels*, each of which provide specific functionalities. New windows can be opened to edit objects, most of which are also made up of panels.

Standard layout

This is what you will see after opening Unreal Engine 5 for the first time. This is the standard/default layout of the editor, as shown in *Figure 3.8*:

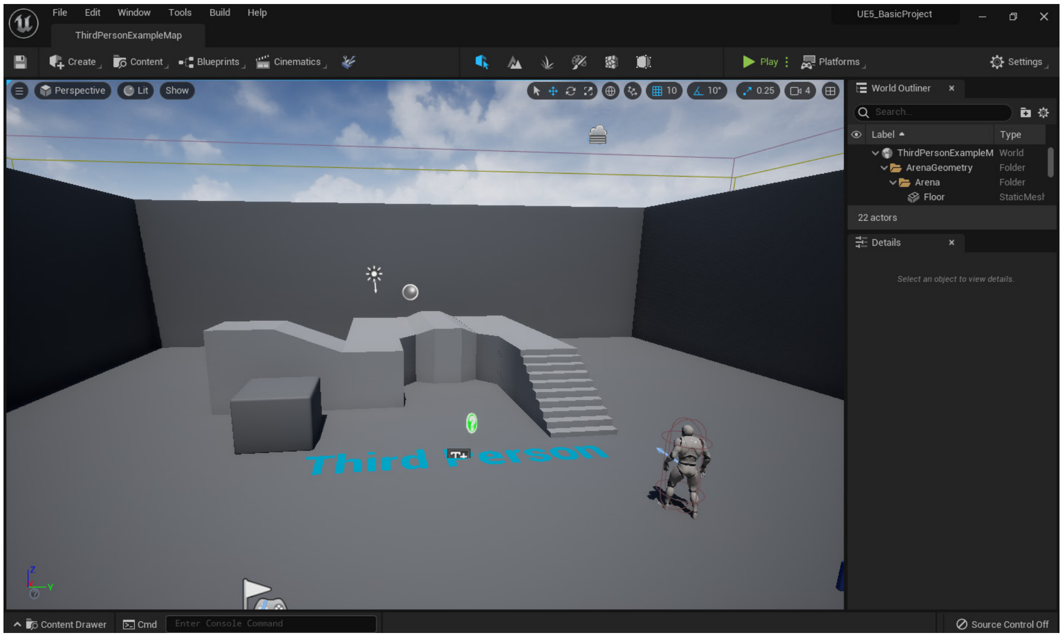


Figure 3.8: *Unreal Engine 5 Editor (Default Layout)*

It has the most commonly used editor panels, a single viewport allowing you to view the world that is currently loaded, and a toolbar providing the ability to simulate using/playing the project, and some means of modifying the current level and the objects within it.

Editing the layout

The layout of the editor can be changed at will by dragging and dropping panels as you please, for example (see *Figure 3.9*):

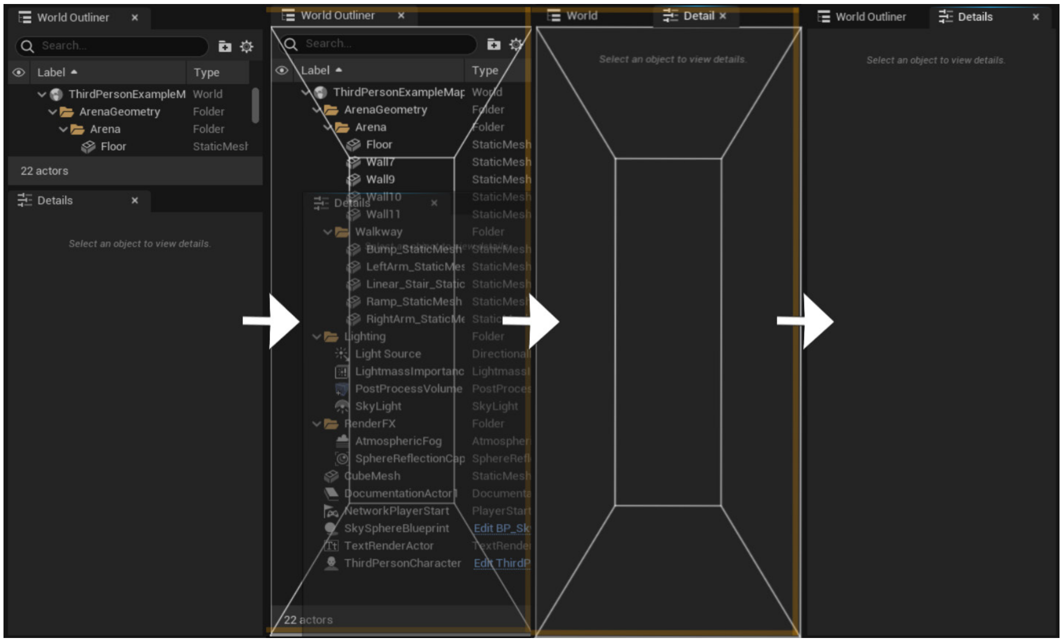


Figure 3.9: *Dragging and dropping panels*

Layout menu

You have a great deal of control over the layout of the editor, so set it up as you see fit. If you would like to return to the standard layout, you can do so at any time by going to the layout menu, as shown in *Figure 3.10*. You can also save and apply custom layouts. So, if you like to change the layout depending on what exactly you're using the editor for, that can very easily be done / managed:

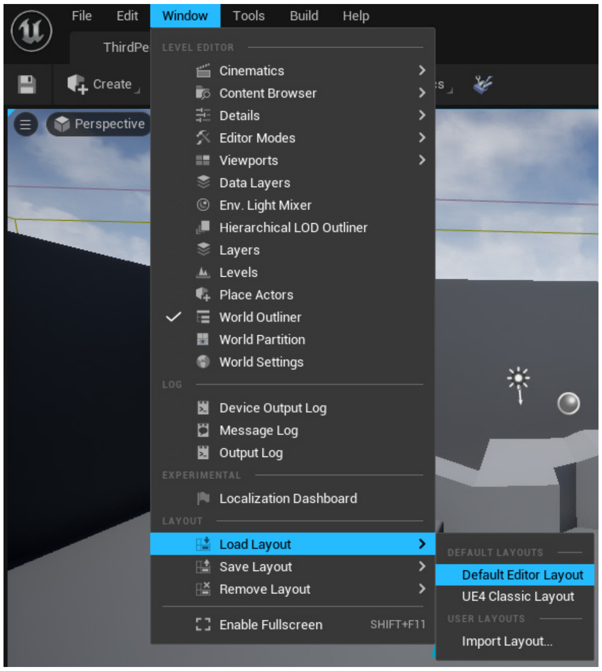


***Figure 3.10**: Layout menu*

Common hotkeys

Hotkeys are combinations of keys on the keyboard that when pressed together, will act as a shortcut to performing a specific function. *Table 3.1* lists the most commonly used hotkeys in UE5:

Function	Hotkey
Undo	*Ctrl + Z*
Redo	*Ctrl + Y*
Cut	*Ctrl + X*
Copy	*Ctrl + C*
Paste	*Ctrl + V*
Duplicate	*Ctrl + W*
Rename Selected	*F2*
Toggle Full-screen mode	*Shift + F11*

***Table 3.1**: Common UE5 Hotkeys*

Important menu/window locations

Inside the editor, there are a great deal of tabs, menus, and panels, all with their own unique purpose. Knowing which one is where and what each one's purpose will save a developer a great amount of time and effort. This chapter will cover the default/standard layout of the UE5 editor, though some elements may differ if you've modified the editor's layout. These tabs and menus can be found on the left side of the editor's title bar, as shown in *Figure 3.11*:

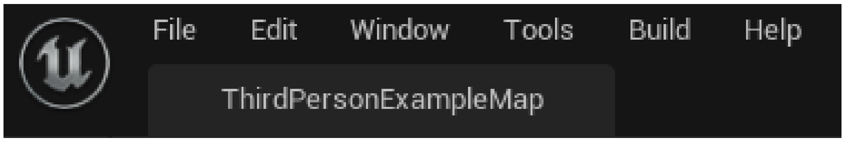


***Figure 3.11**: Title Bar tabs*

File tab

The **`File`** tab contains a series of options regarding file and editor management. From here, you can save your work, open, or start a new project, change or create levels, open an asset to modify, compress the current project for easy transferal, or close the editor.

Edit tab

The **`Edit`** tab (shown in *Figure 3.12*) in the top left of the editor's interface contains a number of options, most of which will be used via the hotkeys provided next to them (undo, cut, copy, paste, and so on):

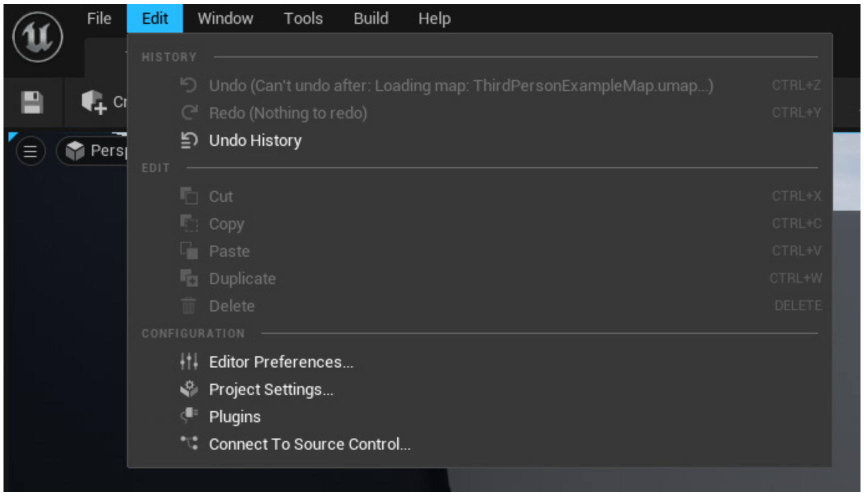


***Figure 3.12**: Configuration menus*

It also contains four very important configuration menus at the bottom of the list. In order, these are as follows:

- **`Editor Preferences`**: This menu is used for making changes to the editor's appearance and modifying some of its functions; changes made in this menu apply to the editor, so their effects will be kept regardless of the project you are working on (while using the same system, of course).
- **`Project Settings`**: This menu is for changing settings purely for the current project; it includes the ability to modify metadata for the project (version numbers, project name, etc.), AI/navigation settings, physics settings, platform settings (UE5 can be used to create apps for Windows, Android, Mac, iOS, and Linux, although some platforms require additional set up.). This menu also includes settings pages for active plugins.
- **`Plugins`**: The plugins menu is a list that allows the user to activate and deactivate plugins, which provide added functionality to the engine/editor (depending on the plugin). The list contains many plugins that are provided with the engine, with many of them disabled by default.
- **`Connect to source control`**: UE5 has integration with the common forms of source control (Subversion, Plastic SCM, Perforce, and Git). This is mostly useful when working on a project collaboratively, and it allows you to manage your source control from within the editor.

Window tab

The **`Window`** tab (shown in *Figure 3.13*) contains a long list of editor panels that aren't opened by default and the option to open multiples of certain panels. Most of these panels are used for more advanced actions within the editor, so a brief explanation will be provided:

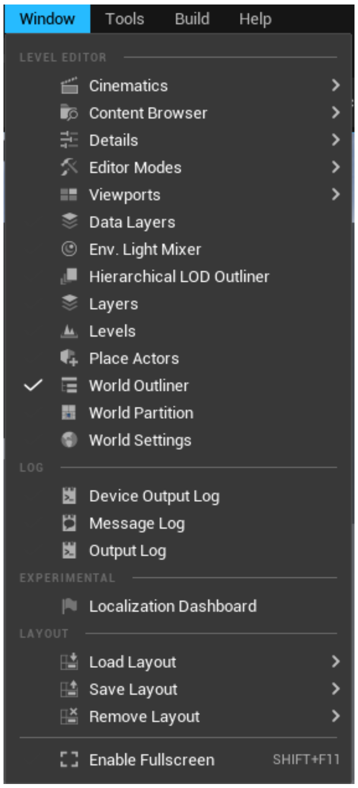


***Figure 3.13**: Window tab*

Level editor

Panels in this section of the list are used for editing the composition of the level and its settings:

- **Cinematics**: Contains access to four panels used for making cinematics with the editor (cutscenes, movies, and so on)
- **Content browser**: Allows creating multiples of the content browser panel (at the bottom of the interface by default)
- **Details**: Allows creating multiples of the `Details` panel (at the right of the interface by default)
- **Editor modes**: Opens toolbars for using the editor to edit level geometry and terrain (these toolbars are already open by default)
- **Viewports**: Allows creating multiples of the `Viewport` (the `Camera` previewing the level, meaning you can get different angles or view different parts of the level at once)

- **Data layers**: Opens a panel used to manage `Layers` within the level (a means of grouping map elements and a replacement for the `Layers` option further down the list)
- **Env. light mixer**: Opens a panel providing access to lighting details for light sources within the map, which greatly simplifies setting up a level's lighting
- **Hierarchical LOD outliner**: Allows tweaking of hierarchical LODs if they are enabled (they aren't by default), which is a means of grouping together certain map element's LODs, primarily for optimization of a project
- **Layers**: Opens a panel used to manage `Layers` within the level (a means of grouping map elements, and precursor to the `Data Layers` option covered previously)
- **Levels**: Creates a panel for managing and modifying sub-levels used within the current one; it is very useful for setting up level streaming (enabling/disabling parts of the map depending on the player's location)
- **Place actors**: Creates a panel used to spawn actors into the level, which can also be done using the content browser panel
- **World outliner**: Opens the World Outliner panel, which provides a list of all actors placed within the currently opened level (the world), this panel is open by default (Right side of the editor)
- **World partition**: Provides access to a panel used to manage sub-levels made using world partitioning (an automated process of dividing a large level into smaller sub-levels)
- **World settings**: Creates a panel providing access to settings for the current level (a `Details` panel for the level)

Logs

Logs are a text output of the results of the engine's various processes:

- **Device output log**: Opens an output log for an attached device
- **Message log**: Opens the message log, which reports compiler errors and warnings, among other things
- **Output log**: Opens the output log, which reports almost everything the engine does

Experimental

The `Experimental` tab is for features of the engine that are still being tested, and as such may not be entirely stable or functional.

Localization dashboard is an interface to assist translating a project into multiple languages (referred to as *Localization*).

Layout

The **`Layout`** tab contains panels used to modify the appearance of the editor:

- **Load layout**: Allows loading the default UE5 or UE4 layouts, as well as any custom layouts
- **Save layout**: Used to save the current layout for future usage
- **Remove layout**: Used to delete custom layouts that are no longer needed

Tools tab

The **`Tools`** tab allows access to a group of tools used for validating data within the project, finding references, viewing information about the project/editor and modifying/creating objects.

Build tab

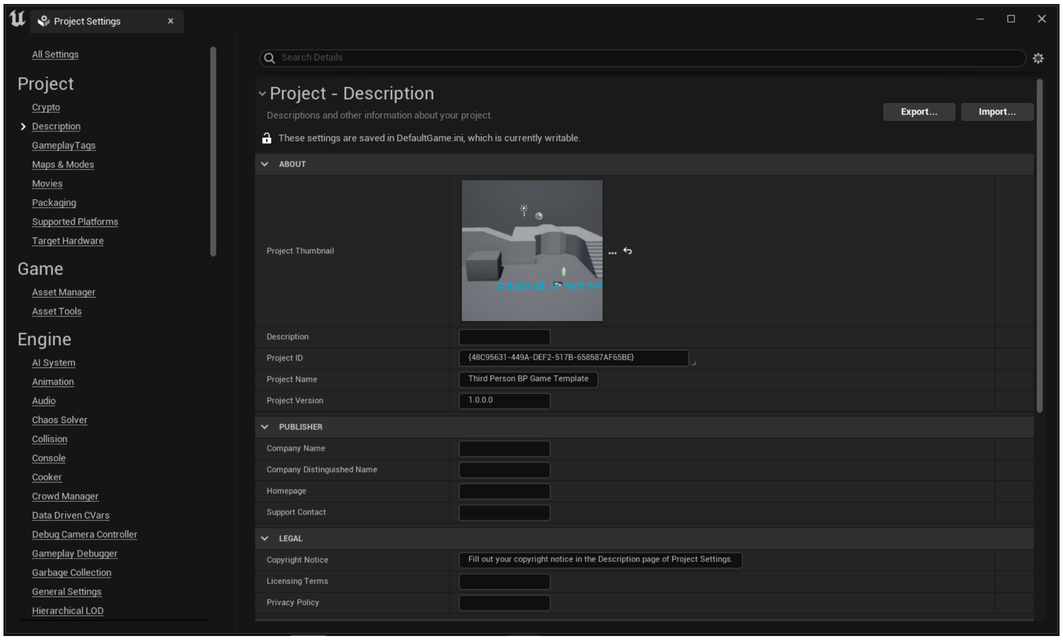
The **`Build`** tab contains a list of functions used to compile/build aspects of the project (lighting, AI navigation), allowing the user to get a better idea of how the project will look/function once it is packaged out as its own application.

Help tab

The **`Help`** tab contains access to documentation regarding the engine and its uses, and a series of basic tutorials. It also contains information regarding the editor and its development, and a bug reporter.

Project settings

The **`Project Settings`** menu (shown in *Figure 3.14*) has a list of categories on the left side of its interface, which allows you to view all settings associated with them:



Project - Description

Descriptions and other information about your project.

These settings are saved in DefaultGame.ini, which is currently writable.

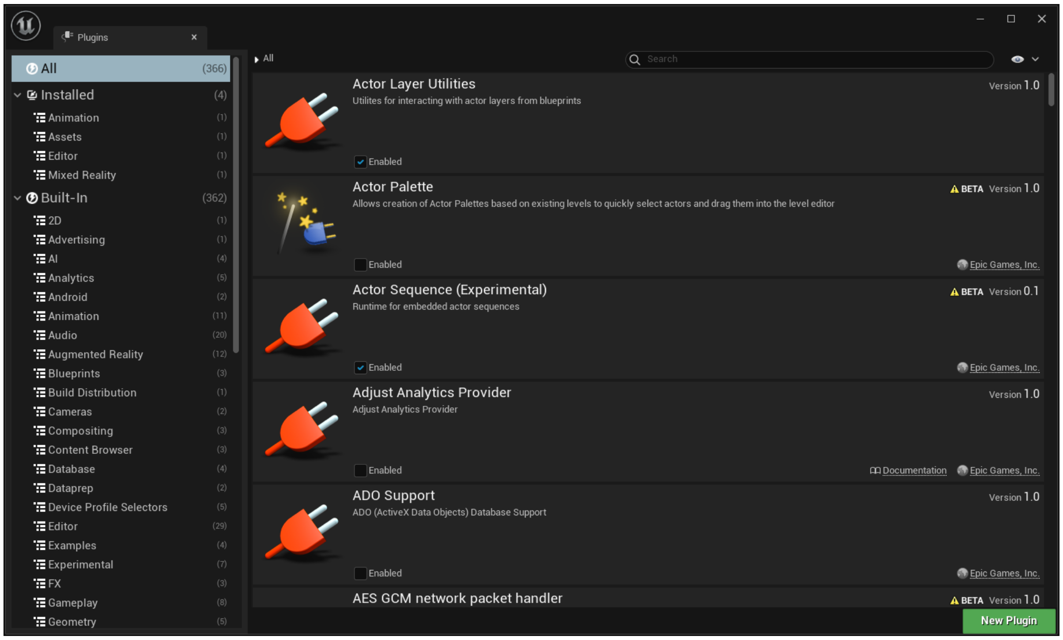
Project Thumbnail	
Description	
Project ID	{48C95631-449A-DEF2-517B-658587AF65BE}
Project Name	Third Person BP Game Template
Project Version	1.0.0.0
Company Name	
Company Distinguished Name	
Homepage	
Support Contact	
Copyright Notice	Fill out your copyright notice in the Description page of Project Settings.
Licensing Terms	
Privacy Policy	

***Figure 3.14**: Project Settings Menu*

Modifying the settings in this menu will affect things in/about the project. This menu can be used to modify the project's meta-data (project version, name, and so on.), AI Navigation, physics settings, collision profiles, and much more.

Plugins

The **`Plugins`** menu (shown in *Figure 3.15*), just like the project settings menu has a list of categories on the left side of its' interface. Selecting one will filter the plugin list on the right side of the interface to only contain plugins associated with the selected category:



Actor Layer Utilities — Utilites for interacting with actor layers from blueprints — Version 1.0 — ✓ Enabled

Actor Palette — Allows creation of Actor Palettes based on existing levels to quickly select actors and drag them into the level editor — BETA Version 1.0 — Enabled — Epic Games, Inc.

Actor Sequence (Experimental) — Runtime for embedded actor sequences — BETA Version 0.1 — ✓ Enabled — Epic Games, Inc.

Adjust Analytics Provider — Adjust Analytics Provider — Version 1.0 — Enabled — Documentation — Epic Games, Inc.

ADO Support — ADO (ActiveX Data Objects) Database Support — Version 1.0 — Enabled — Epic Games, Inc.

AES GCM network packet handler — BETA Version 1.0

***Figure 3.15**: Plugins menu*

Unreal Engine 5 comes with many plugins, some of which are enabled by default, while some are not.

Plugins are packages that provide additional functionality and/or content to the project. If there is something you wish to do with the editor that you can't, there's a good chance there's a plugin in this menu that can assist you if not outright provide that functionality. Failing that, you may be able to find a custom plugin on the internet or potentially, make one yourself.

Conclusion

Unreal Engine 5 is extremely expansive. It has capabilities for just about anything you could imagine; knowing your way around the editor and what it is capable of is very important when trying to use it effectively and efficiently. You can change the way the editor is set up to match your preferences, and you can save and use different layouts depending on what exactly you are using the editor for. The locations of some elements of the editor cannot be changed (the title bar tabs). UE5 contains its own tutorial and links to documentation regarding itself and its usage. If you cannot figure something out or are curious as to what its exact purpose or usage is, chances are you can find out with a little research.

In the next chapter, we will go over the usage of Unreal Engine's visual scripting language *blueprint* and actor components.

Points to remember

- Make sure whatever you get from the marketplace is compatible with the engine version you want to use it for.
- Remember the hotkeys, as they will save you a great deal of time; learn the hotkeys for more advanced actions that were not included in this chapter.
- Remember the purpose of the tabs in the title bar and the options within them, some of them will likely see common usage.
- Research things you do not know about.
- It is worth doing the tutorial included with UE5 (located in the **`Help`** tab).
- The Epic Games Launcher is the home of Unreal Engine, but it is not its sole purpose.
- There are many plugins provided with UE5; knowing what's available may save you a great deal of time when developing a project.
- The **`Log`** options within the **`Window`** tab is extremely useful for debugging (finding the cause of errors).

Multiple choice questions

1. **Which tab contains the layout menu?**
    a. File
    b. Edit
    c. Window
    d. Help

2. **What is the purpose of the world outliner?**
    a. A list of all the objects in the world
    b. Closing the editor
    c. Modifying sub levels and level streaming
    d. Building geometry

Answers

1. **c**
2. **a**

Questions

1. Where do you get assistance with using the editor?
2. Explain what a plugin is.
3. What can the project settings menu be used to change?

Key terms

- **Editor**: The Unreal Engine 5 program with a project loaded
- **Panel**: An element within the editor with a specific purpose that can be moved around the editor
- **Log**: A text output containing information regarding the current/previous running instance(s) of the project
- **Level**: An environment to be used within the project
- **World**: The space that contains all the currently open levels

CHAPTER 4
Using Blueprints

Each game engine has its own method and patterns to facilitate design and develop game assets. For example, having a 3D editor like *Unity* to edit and design level by user, or having in-depth physic simulation functions and procedure to simulate physics like *Source engine* are basically some tools provided by game engine to help users make game materials. The Unreal Engine has its own method known as **blueprint** to create game assets, organize them on the scene, control them with game mechanics rules, and optimize them during runtime for player depending on machine features.

In this chapter, we will learn how to create and design blueprint game objects, which is a fundamental learning step for both designer and programmers to use Unreal Engine. And then by going through step-by-step practical examples, we will learn how to use blueprint components. Then we learn how to add static mesh and physical behaviour to blueprint game object.

Structure

In this chapter, we will discuss the following topics:

- What is a blueprint?
- Design with blueprint
- Blueprint components

- Adding Mesh to blueprint
- Adding physics (basic)

Objectives

After studying this unit, you should be able to create blueprint objects inside Unreal Engine. The blueprint object supports different functionalities and purpose within your game application, so we will go through making an actor blueprint and components attached to it, and we will learn hierarchical relation between these components in order to support game scenario and procedures of game.

What is a blueprint?

To answer this question, let us assume you are the following:

- A designer with an experience of using 3D design applications for business, game design or art, and also having 2D design experience with applications like Photoshop and Illustrator.
- A programmer with experience of coding simple functions and class.
- New to game design and programming and would like to discover your talent in Unreal Engine 5.

Now, let us explain *blueprint* for each of these groups.

As a 3D designer, you know that users can **`Create`** objects and also **`Edit`** these objects in each 3D application. Also, the user has details of size, rotations, and coordination of the objects somewhere on their user interface. A good example of these tools is

Blender. As *Figure 4.1* shows, by opening Blender, the user has a basic 3D object with its coordination on right and some tools to edit on the left:

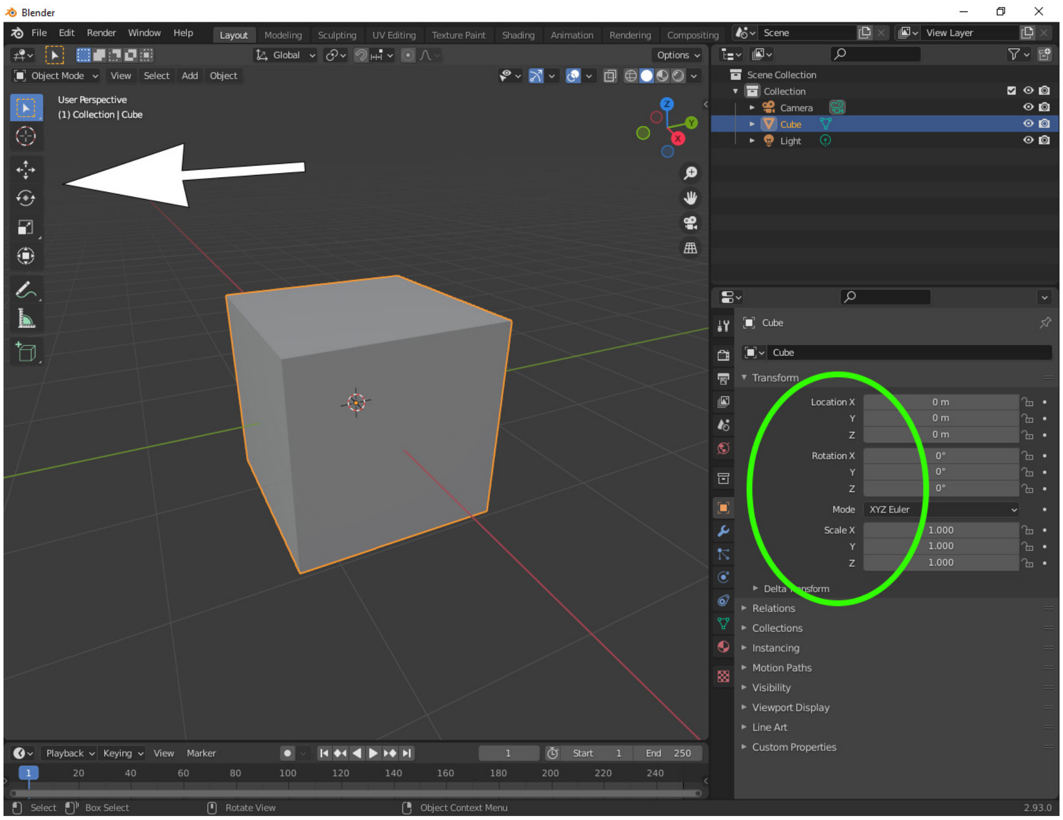


***Figure 4.1**: Visual*

As a 2D designer, you already know about color pallet, image size in pixel, and transparency between image layers when they overlap. This is basic knowledge of

2D design that, for example Photoshop, provides a massive toolset as *Figure 4.02* shows,which are designed for working with a 2 dimensional images. :

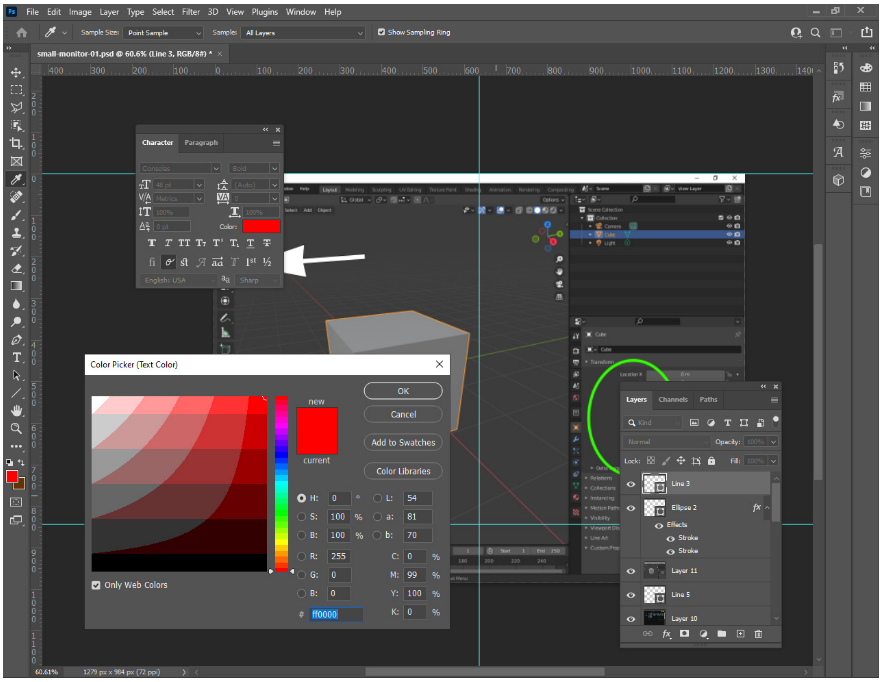


Figure 4.2: *Visual*

Figure 4.3 is a screenshot of Unreal Tournament. This game is **first-person shooter (FPS)**, which is completely designed with Unreal Engine:

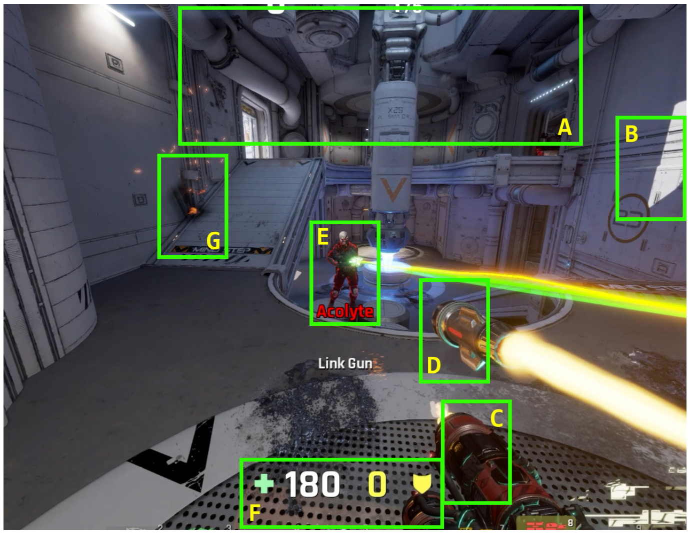


Figure 4.3: *Screenshot of Unreal Tournament with marked game elements*

As you can see in *Figure 4.3*, there are a number of *game elements*; some have 3D locations in the scene, and some are render data for the user, like material and UI. As examples of blueprint, we marked some game elements like 3D details and materials of level (green box **A**), lighting effects in environment (green box **B**), the weapon object (green box **C**), the actual rocket round that the player shoots (green box **D**), the player`s user interface UI (green box **F**), and finally, the enemy unit (green box **E**). Each of these elements uses a blueprint object to present its own contents, render materials, and change their own state based on player interaction in game. For example, on impact of rocket to the walls, the material blueprint on that location automatically renders a *decal effect*, which is marked by green box **G**, or the fire behind the rocket in green box **D** is switched on by rocket blueprint, and it is a particle system, which again, is a blueprint by itself.

Similar to what you experienced in 2D, and 3D design, a blueprint in Unreal Engine is like an **Object**. This **Object**, or let's say **Blueprint**, has many types, and each one is designed to perform specific **Action** in the game, like providing a visual effect, presenting user interface, grabbing player inputs, generating time-based cinematics, playing audio, and so on.

Designers can use blueprints to implement physics, create special effects like particles (like fog and fire), simulate day and night lighting in map, provide game assets (like weapons), and for animation and cinematography. Blueprints have different types, and each type has its own properties and features. For now, we will focus on the basics of design and automation in blueprint. Good design of blueprint always saves time and increased performance in massive scale projects.

As a programmer, assume that you are already aware of logic and methods in a programming language to develop the main code, make function and declare class, and also know about the **order of execution** in code, which means the sequence of running each *line* of your code after each other. In Unreal Engine, blueprint works like *code*, but instead of executing *lines of code*, the engine will execute *blocks* of *procedures* and *functions* in a sequence similar to the *order of execution*.

Refer to *Figure 4.3*; there are a number of blueprint objects, like the weapon marked in green box **C**. Each time a player presses fire, a code inside the weapon`s Blueprint is executed and visual effect of shooting is activated; then, it plays animation on weapon and spawns a projectile in the scene, which is basically a Blueprint of a rocket round (marked by green box **D**). The user interface marked in green box **F** is also a series of Blueprint objects which grab data from player`s states and show them to the user on screen.

Another way to imagine Blueprint as a developer is to assume that each **`Blueprint`** is like a **`Class`** with properties, methods, and functions. As you know, there can be different types of class in any application to perform different tasks. In Unreal Engine as well, there are different types of **`Blueprints`** to perform specific functionalities in game. For example, **`Character Blueprint`** can accept AI navigations, **`HUD Blueprint`** is designed for user interface interactions, **`Animation Blueprint`** supports a number of methods and functions for running animations, and so on. We will cover blueprint programming in the following chapters, and we will cover these **`Blueprint`** objects in depth.

Now, assume that you are not a programmer, and designing is not your concern. This means you never seriously touched any 3D application, you don't have Photoshop experience, and your knowledge about coding is zero. Well, the good news is, there is no problem in this situation; using blueprint in Unreal Engine is a mix of design and code, which is an opportunity for beginners to discover their own talent, either in coding or design (or maybe both!), for making game assets. So, let's take a deep breath and start.

The word *Blueprint* by its own meaning: it is a photographic presentation and guide of an early *plan* for a *visual structure* or *interactive system*.

Imagine that you have an empty magic *Paper Box*, as shown in *Figure 4.4*. Now, because it's a magical item, we can attach anything to it easily, like flashlight, wheels, maybe weapon, smoke, or camera. Now, assume that we like to make a simple vehicle, so add four wheels to it and an electrical engine to move the wheels. Also, add a camera to see where it goes:

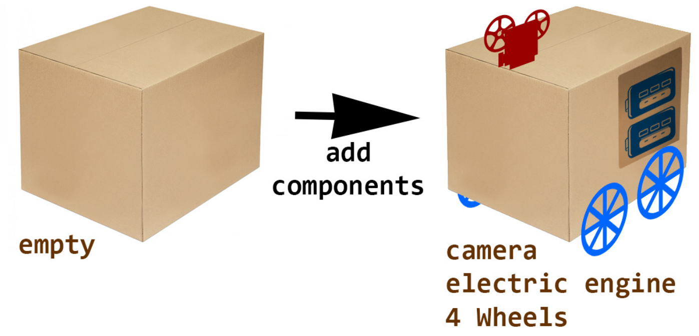


***Figure 4.4**: Example of making a very basic Blueprint of a vehicle with 4 wheels and an electric engine and camera*

At this stage, having an engine, wheels and a camera is NOT enough, we need something to *Control* it and get it to move and stop. So, by using magic, we simply add a remote receiver and some mechanics that can get *events* from receiver and apply them to the engine. Sounds simple, doesn't it? So, what we made by magic is a *Blueprint* of a remote car. This approach is extremely useful at the beginning of any Blueprint development.

The blueprint explains features of design and address events and explains the result of each event as visual presentation or informative data. For example, refer to *Figure 4.3*: the weapon's blueprint already has weapon mesh with a number of materials, and it also has code to grab the **`click`** event from mouse and then execute an event for play weapon-shoot animation (same as the remote vehicle explained in *Figure 4.4*: when you press run in remote control, the engine moves the wheels and the box starts moving). When the player shoots the rocket in game, the weapon's blueprint will spawn smoke and flash on screen by an event, and a projectile will spawn in the scene and move to the target. Each of these events, (mouse input, animation, and visual effect) occurs in a **`sequence`** and uses a **`component`** to show the result. For example, weapon-shoot animation is played on weapon 3D mesh object **`component`**, and smoke and fire are made by particle system **`component`**, and mouse detection is using Player Controller **`component`** to work with. This is the same as the example in *Figure 4.4*: each **`component`** is responsible for a unique task (wheels for movement, engine for power, and so on).

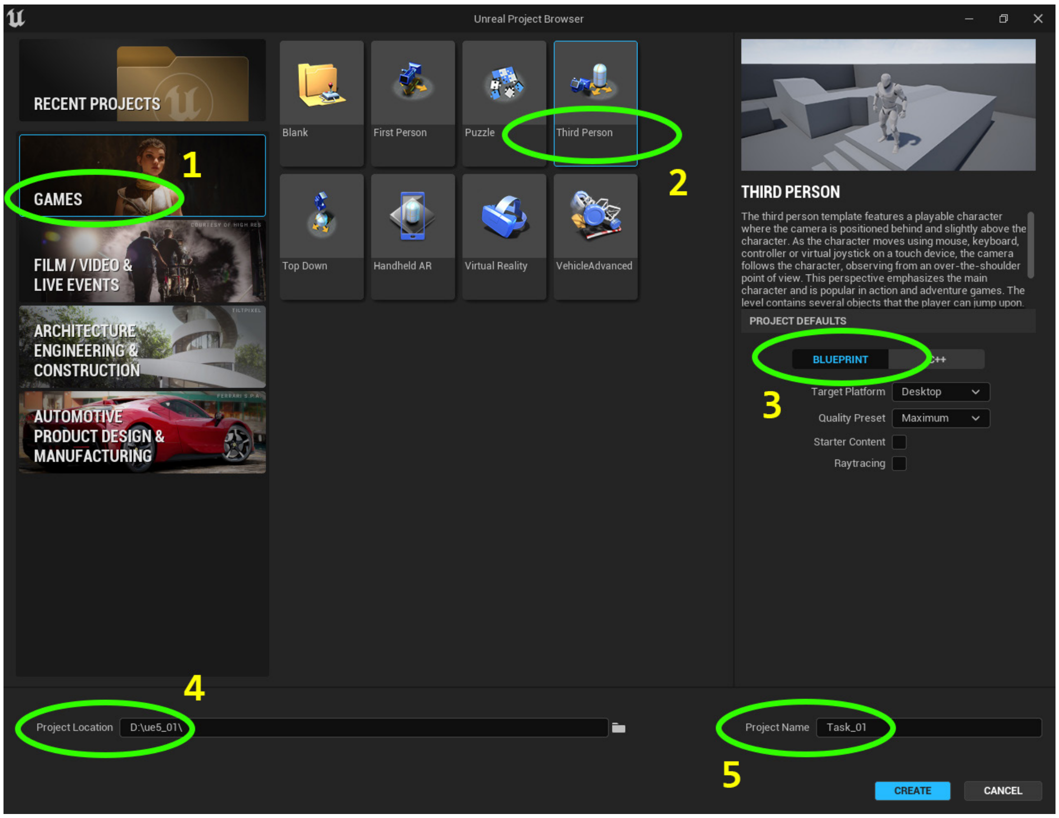
Each Blueprint in Unreal Engine has an editor known as **Blueprint Editor**. Blueprint Editor prepares tools to create and customize visual elements of a game object that we will go through in this chapter. At the same time, it has an object-oriented node-base visual scripting language for programming game object functionalities. This language is designed to be easy to learn and has massive support by C++ libraries developed for the engine. We will cover blueprint programming in detail in a later chapter.

It sounds complicated, but be patient, Blueprint is magically designed to be *simple*; by the end of this book, you will be able to make your own weapon blueprint and use it in your own game.

Design with blueprint

The Unreal Engine has a series of tools developed for use with blueprints known as engine's **Blueprint Components**. These components provide support to design game elements (like the weapon in the earlier example), develop functionality like AI, spawn particle effect, play audio, and animation. They cover 90% of making a basic-advanced asset for putting in a game scene. In this chapter, we will learn the basics of design game assets with Blueprint components.

Revisit the previous chapter, open Unreal Engine 5, and quickly make a new project by using the **`Third Person`** template from the **`Game template`** category, as shown in *Figure 4.5*:



THIRD PERSON

The third person template features a playable character where the camera is positioned behind and slightly above the character. As the character moves using mouse, keyboard, controller or virtual joystick on a touch device, the camera follows the character, observing from an over-the-shoulder point of view. This perspective emphasizes the main character and is popular in action and adventure games. The level contains several objects that the player can jump upon.

PROJECT DEFAULTS

[✓ BLUEPRINT] C++

Target Platform: Desktop

Quality Preset: Maximum

Starter Content: ☐

Raytracing: ☐

Project Location: D:\ue5_01\

Project Name: Task_01

Figure 4.5*: Select the GAMES category (1), then choose Third Persson template (2), then select BLUEPRINT in the project defaults (3), and then set project location (4) and project name (5) and click on the "CREATE" button.*

Open the project, select the **`ThirdPersonBP`** folder in content browser and then select the **`Blueprints`** folder. At the same time, on top of content browser, you can find the folder path that you just navigate (this area is useful when you navigate between folders). Now, right-click in the content browser and click on **`Blueprint Class`** from the list, as shown in *Figure 4.6*:

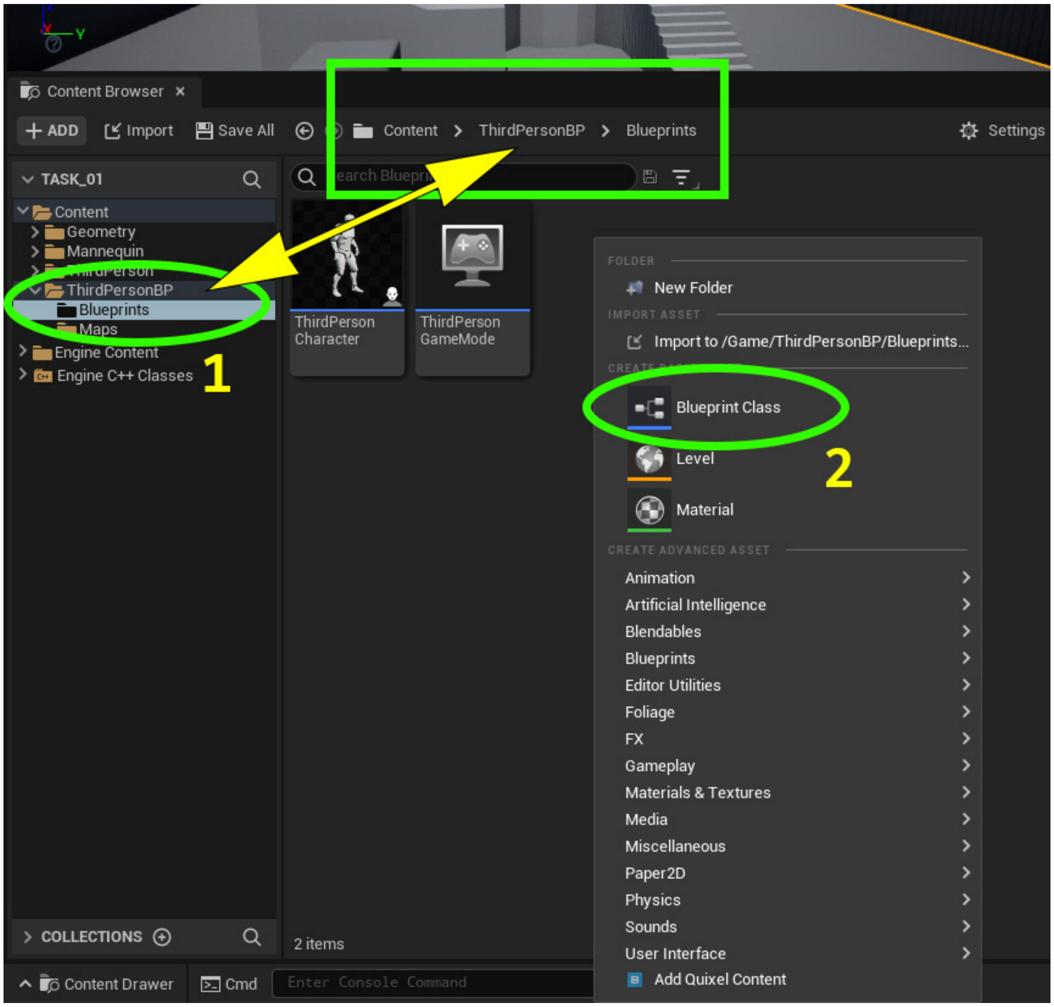
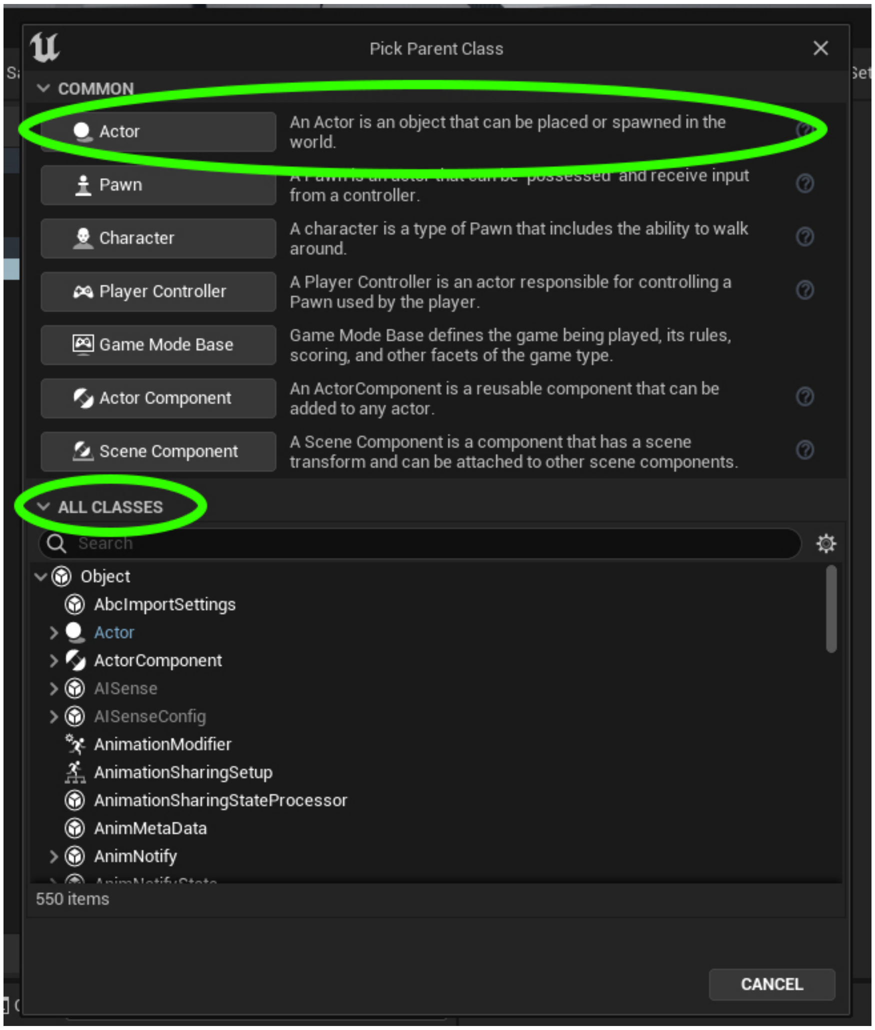


Figure 4.6: *Make new blueprint by choose a folder in content browser (1) and then right click in there and select "Blueprint Class"(2)*

The engine will automatically open the **`Pick Parent Class`** window with a list of **`Blueprint`** objects to select, as shown in *Figure 4.7*. As this image shows, this window contains a number of blueprint classes, and by clicking on the **`ALL CLASSES`** drop-down list, you will find a complete list of available blueprint classes. We will cover the essential blueprint classes shown here by the end of this book:



Pick Parent Class

COMMON

Actor	An Actor is an object that can be placed or spawned in the world.
Pawn	A Pawn is an actor that can be 'possessed' and receive input from a controller.
Character	A character is a type of Pawn that includes the ability to walk around.
Player Controller	A Player Controller is an actor responsible for controlling a Pawn used by the player.
Game Mode Base	Game Mode Base defines the game being played, its rules, scoring, and other facets of the game type.
Actor Component	An ActorComponent is a reusable component that can be added to any actor.
Scene Component	A Scene Component is a component that has a scene transform and can be attached to other scene components.

ALL CLASSES

Search

- Object
  - AbcImportSettings
  - Actor
  - ActorComponent
  - AISense
  - AISenseConfig
  - AnimationModifier
  - AnimationSharingSetup
  - AnimationSharingStateProcessor
  - AnimMetaData
  - AnimNotify

550 items

CANCEL

Figure 4.7: Select blueprint class type from the "Pick Parent Class" window. Click on "ALL CLASSES" to expand the list

For now, click on **`Actor`**, and the engine will automatically make a new object in your content browser, which is your new **`Blueprint`**. Now, rename this blueprint in content browser to **`BP_Item_01`** by right-clicking on the object and selecting rename or by pressing *F2*, as shown in *Figure 4.8*. Using **`BP_`** as the prefix for naming a blueprint is an industry standard that we will follow in this book:

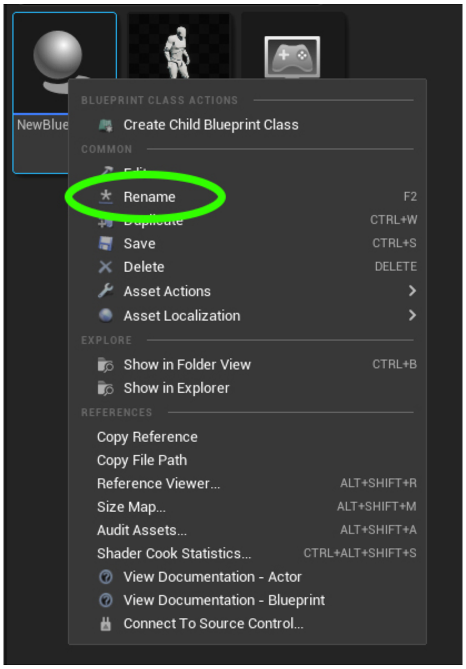

NewBlue
BLUEPRINT CLASS ACTIONS
Create Child Blueprint Class
COMMON
Edit
Rename F2
Duplicate CTRL+W
Save CTRL+S
Delete DELETE
Asset Actions
Asset Localization
EXPLORE
Show in Folder View CTRL+B
Show in Explorer
REFERENCES
Copy Reference
Copy File Path
Reference Viewer... ALT+SHIFT+R
Size Map... ALT+SHIFT+M
Audit Assets... ALT+SHIFT+A
Shader Cook Statistics... CTRL+ALT+SHIFT+S
View Documentation - Actor
View Documentation - Blueprint
Connect To Source Control...

***Figure 4.8**: Right-click on blueprint and select Rename from the list*

The new object is an **`Actor Blueprint`** object. It is a basic game asset that can be placed on stage, spawned in game, and can receive user interactions. Double-clicks on **`BP_Item_01`** blueprint in your content browser, and you will open a blueprint editor, as shown in *Figure 4.9*. First, click on the **`Viewport`** panel to activate it. Then, on the left, locate the **`Component`** panel, click on **`+ADD`** button, and select **`Cube`** from drop-down list:

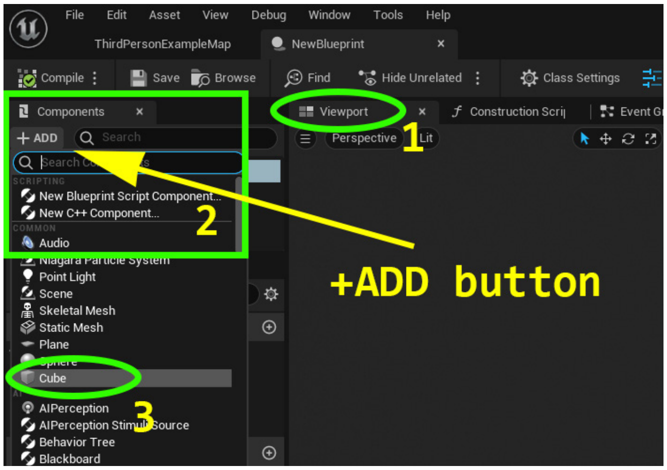
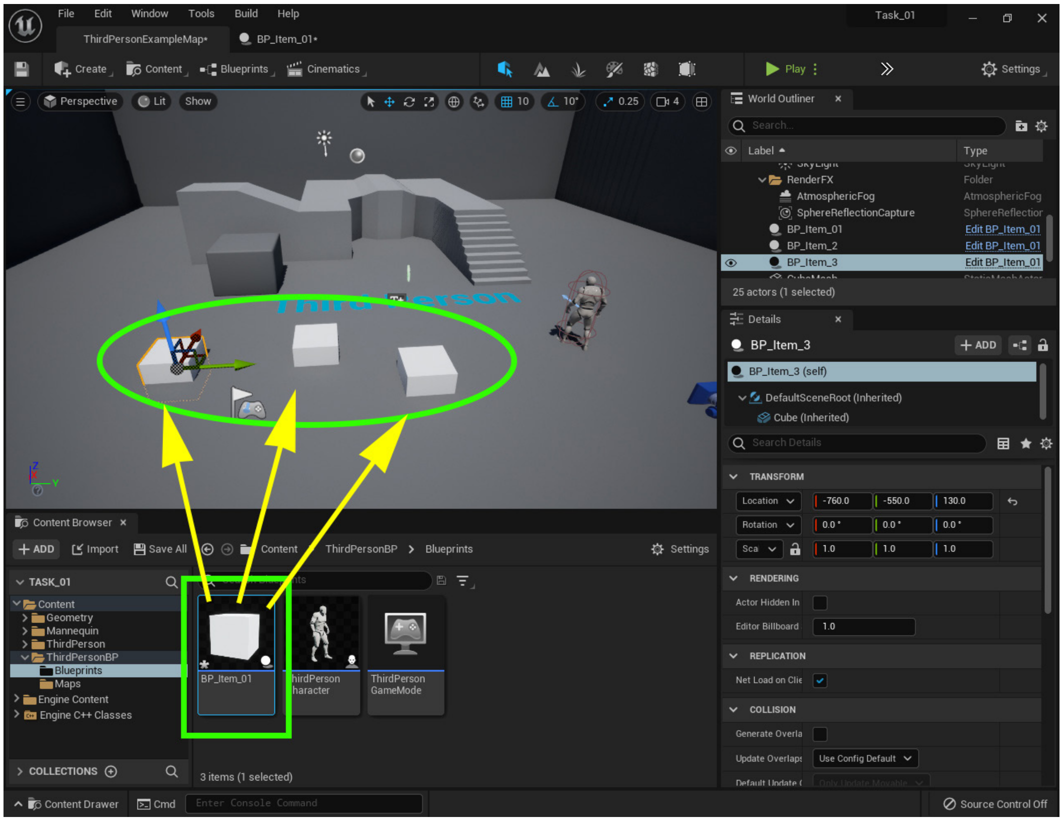


***Figure 4.9**: Click on Viewport panel (1) and then click on the +ADD button in the Components panel (2) and select Cube from the drop-down list (3)*

Now, navigate to the main editor; as you can see, the **BP_Item_01** blueprint has changed and now it shows a cube. Now, drag and drop three instances of **BP_Item_01** to stage by using mouse, as shown in *Figure 4.10*:



Label	Type
RenderFX	Folder
AtmosphericFog	AtmosphericFog
SphereReflectionCapture	SphereReflection
BP_Item_01	Edit BP_Item_01
BP_Item_2	Edit BP_Item_01
BP_Item_3	Edit BP_Item_01

25 actors (1 selected)

BP_Item_3
BP_Item_3 (self)
DefaultSceneRoot (Inherited)
Cube (Inherited)

TRANSFORM

Location	-760.0	-550.0	130.0
Rotation	0.0 °	0.0 °	0.0 °
Sca	1.0	1.0	1.0

RENDERING
Actor Hidden In ☐
Editor Billboard 1.0
REPLICATION
Net Load on Clie ✓
COLLISION
Generate Overla ☐
Update Overlaps Use Config Default

Figure 4.10: Drag and drop three instances of BP_Item_01 to the scene

Now, click on each **BP_Item_01** blueprint instance and use transform tools (located on top of viewport) to change location, rotate, and rescale each object on the stage, as shown in *Figure 4.11*. When you use transition tools, the values of each property (location, rotation, and scale) are shown in the **Details** panel in the **Transform** section. At the same time, you have a list of your objects and blueprints in the scene in the **World Outliner** panel:

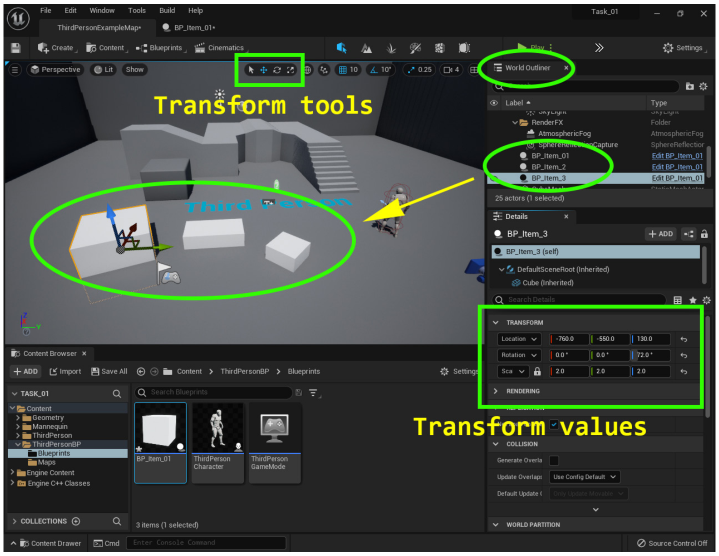


Figure 4.11: *You can customize transform values (Location, rotation, scale) in the Details panel*

You can directly click on each item in the **`World Outliner`** panel and check properties that belong to them in the **`Details`** panel. To delete items from the scene, press *Delete* on keyboard or right-click on the object/blueprint anywhere and then select **`Delete`** from the **`Edit`** section. Here, you have options for **`Cut`**, **`Copy`**, **`Paste`**, **`Duplicate`**, **`Delete`**, and **`Rename`**, as shown in *Figure 4.12*:

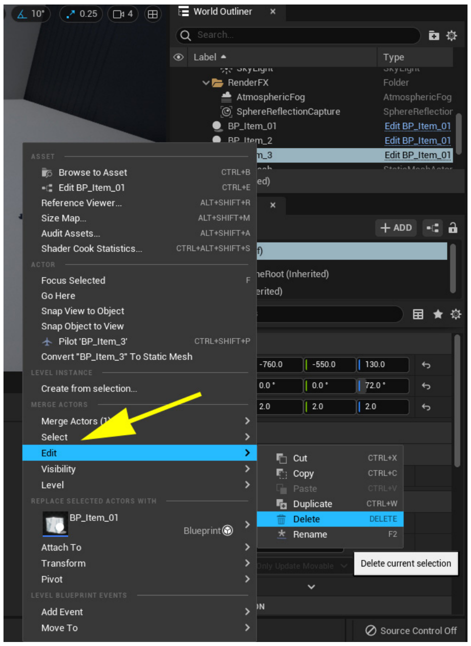


Figure 4.12: *Edit options for blueprint instances on the scene*

Now, follow the same procedure and create **`BP_Item_02`**. Then, from the **`Component`** panel, use the **`+ADD`** button to add sphere static mesh component. Then, drag and drop two instances of them into the stage and change their transform values, as shown in *Figure 4.13*:

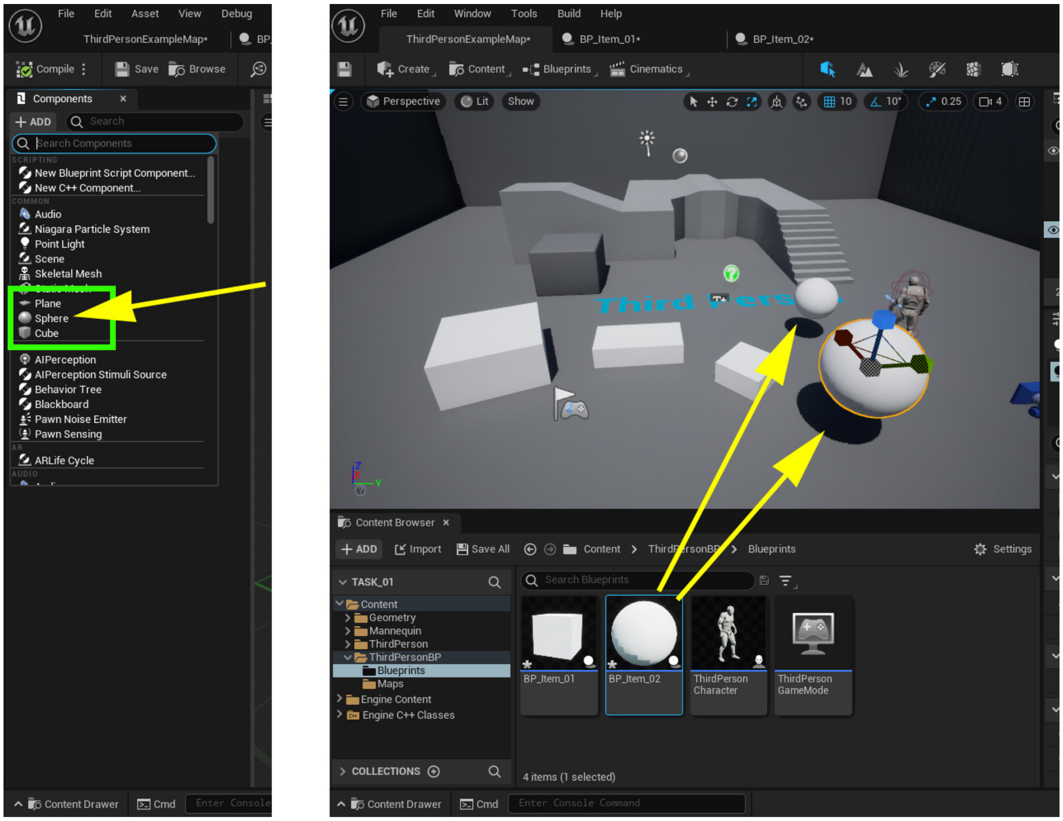


Figure 4.13: *Left: Add Sphere static mesh component in Blueprint editor to BP_Item_02 and drop two instances of it into the scene*

Now, press the **`Play`** button and test the blueprints you made in the scene, which will be similar to the ones in Figure 4.14. This is the magic behind blueprint in Unreal Engine, which enhanced making game assets and simplified populating them on the scene:

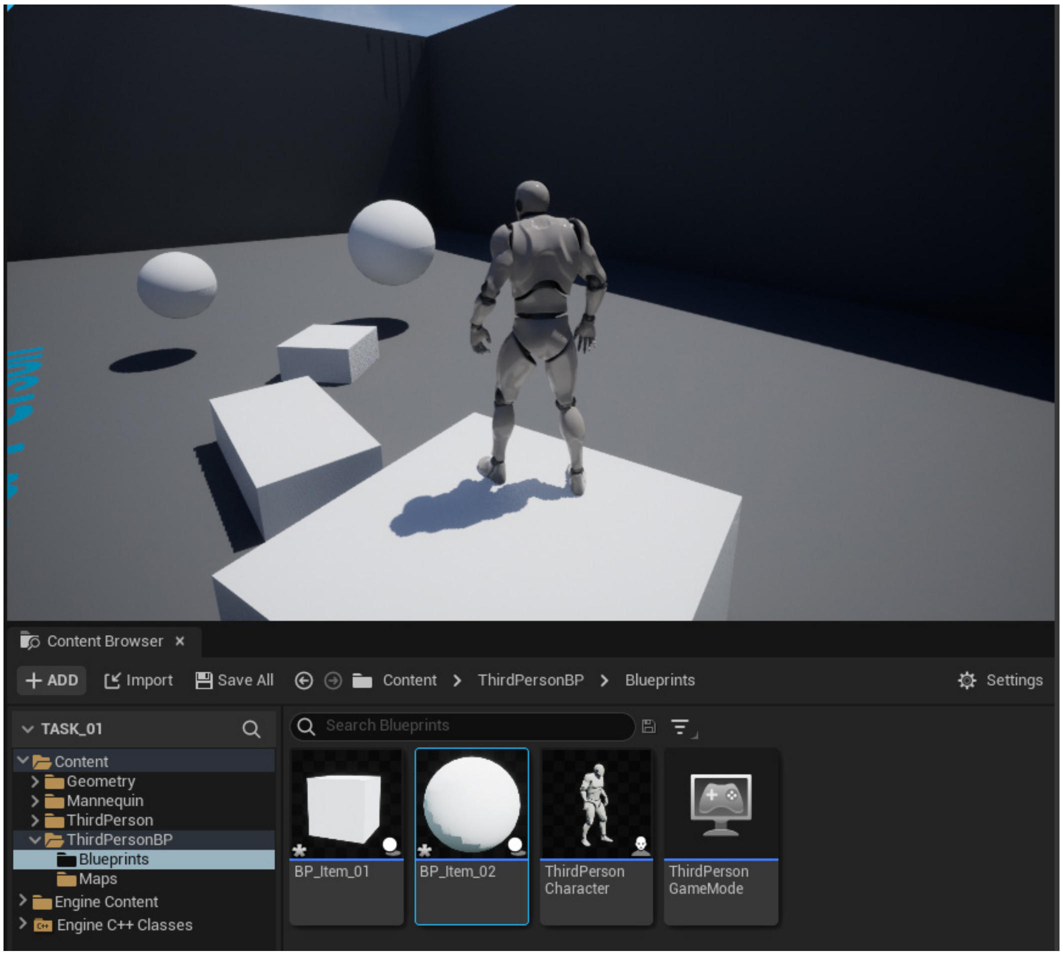


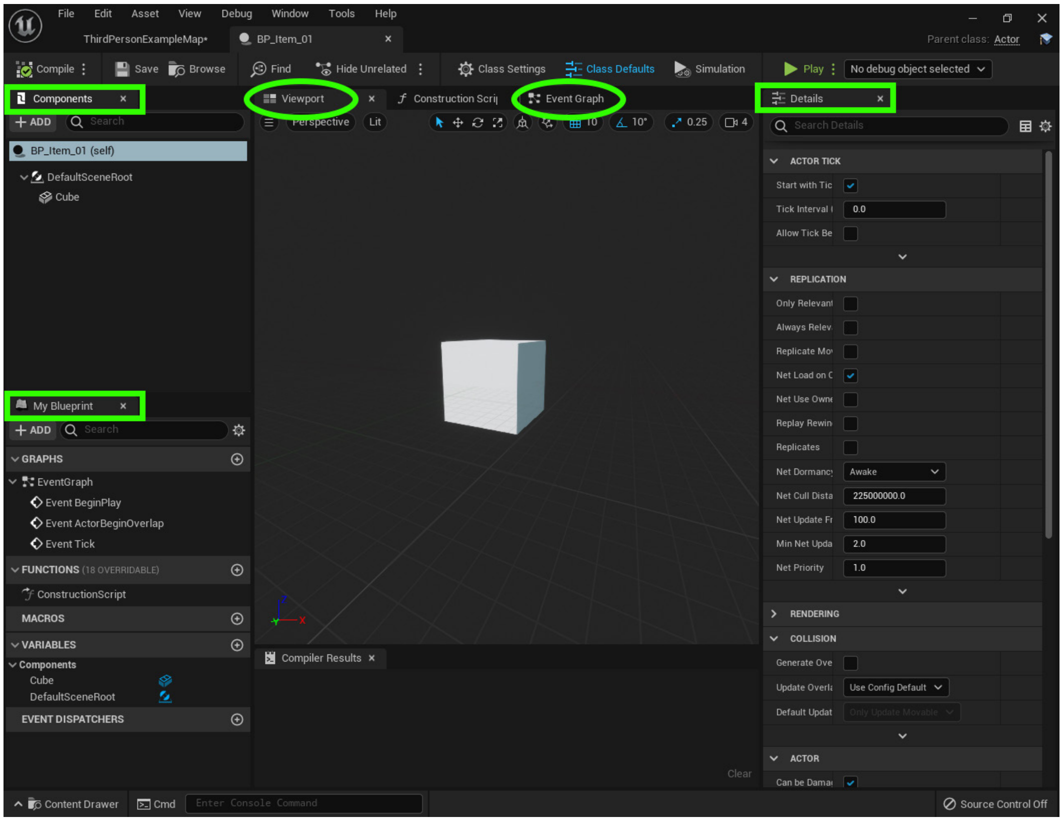
***Figure 4.14**: Testing new actor blueprints in the scene*

With this done, it's time to take a closer look at the blueprint editor and components.

Blueprint components

In Unreal Engine, blueprints are designed to accelerate design and coding process to make game assets. At any level of skill in game design, when you use Unreal Engine, you must make a number of blueprint components to make a game asset. These components are either used in designs like cubes and spheres, which we used before or are used for expressing a functionality like character movement or play audio file. To work with components, you must know the basics of blueprint Editor, which is similar to main editor in case of viewport tools and navigations, except that the blueprint editor is customized to develop and design blueprint objects.

Double-click on the **BP_Item_01** blueprint to open blueprint editor. As *Figure 4.15* shows, there are various panels and buttons in blueprint editor that look complicated, but the majority of your work (at the beginning) are with these panels:



Start with Tic ✓
Tick Interval 0.0
Allow Tick Be
Only Relevant
Always Relev
Replicate Mo
Net Load on ✓
Net Use Own
Replay Rewin
Replicates
Net Dormanc Awake
Net Cull Dista 225000000.0
Net Update Fr 100.0
Min Net Upda 2.0
Net Priority 1.0
Generate Ove
Update Overl Use Config Default
Default Updat Only Update Movable
Can be Dama ✓

Figure 4.15: *Most in use Blueprint Editor panels: Components (1), My Blueprint (2), Details (3), Viewport (4) and EventGraph (5)*

- **Components**: This panel is for adding, selecting, and editing blueprint components; also, the hierarchy of components are editable from this panel. You can attach components to each other by simply dragging and dropping them over each other with the mouse.

- **My Blueprint**: This panel is designed to show properties, procedures, and inherited variables of the actual blueprint you work with. We will go through more details on this panel when we begin coding in a later chapter.

- **Details**: When you select an item from previous panels, it will automatically refresh the **Details** panel and give the properties of that specific item. For example, as shown in *Figure 4.16*, if you choose **Cube static mesh** in the **Components** panel, the **Detail** panel will show the series of properties related to Cube, like transforms and material, which can directly change and customize the **Cube** component.

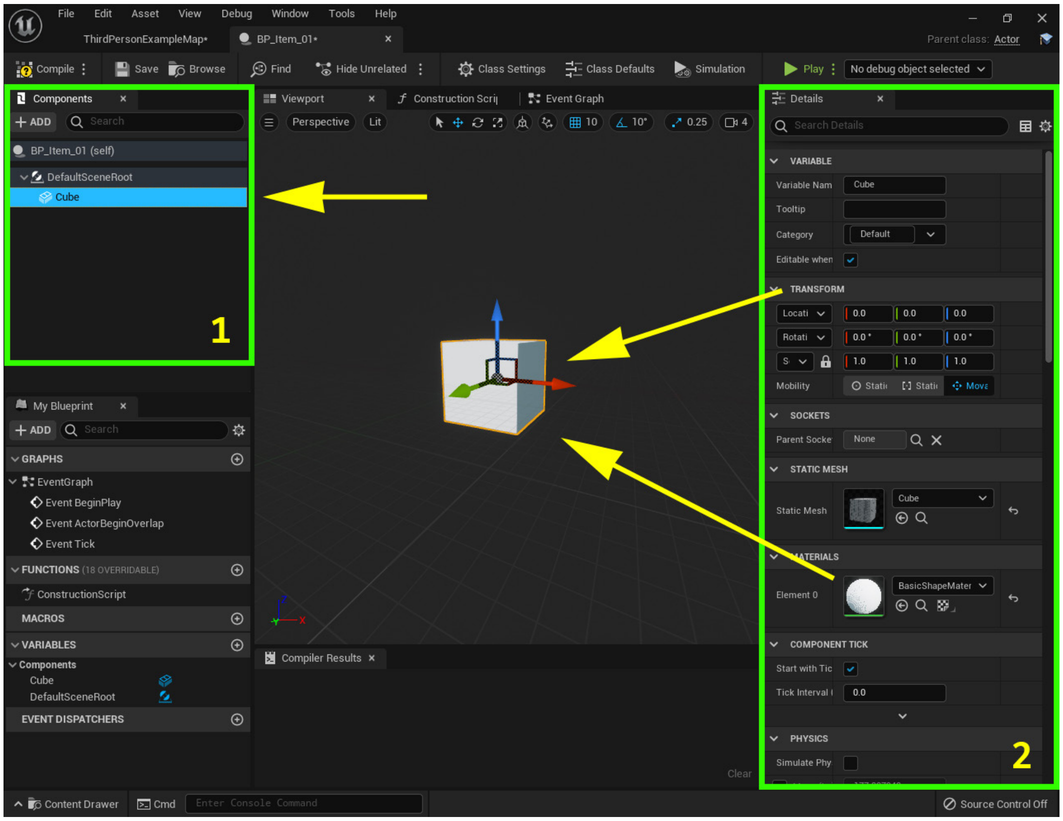


Figure 4.16: *After clicking on Cube in Components panel (1), Detail panel will show properties related to Cube (2) or any component chosen from Components panel*

- **Viewport**: The **Viewport** panel is the main area of designing a blueprint, and it works like main editor viewport but with less options. As *Figure 4.17* shows, on top of this panel, user has options to change perspective and rendering mode, select/rotate/scale tool and handy snapping option for each and camera speed:

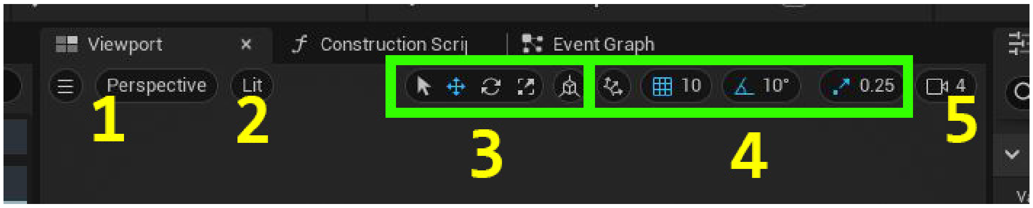


Figure 4.17: *Tool set on top of Viewport panel: Viewport mode (1), rendering options (2), editing tools (3), grid and snap setup (4) and camera speed setting (5)*

- **`Event Graph`**: This is the editor for *coding* game functions and procedures by using the Blueprint Visual Scripting system in Unreal Engine. This is a gameplay scripting system supported by C++ and is based on the concept of using a node-based interface. Users can drag and drop items to this panel from the **`Components`** panel and the **`My Blueprint`** panel and use them as reference in code. Also, by right-clicking inside this panel, as *Figure 4.18* shows, user has access to a series of blueprint node, which we will cover in the later chapters when we dig into coding with blueprint:

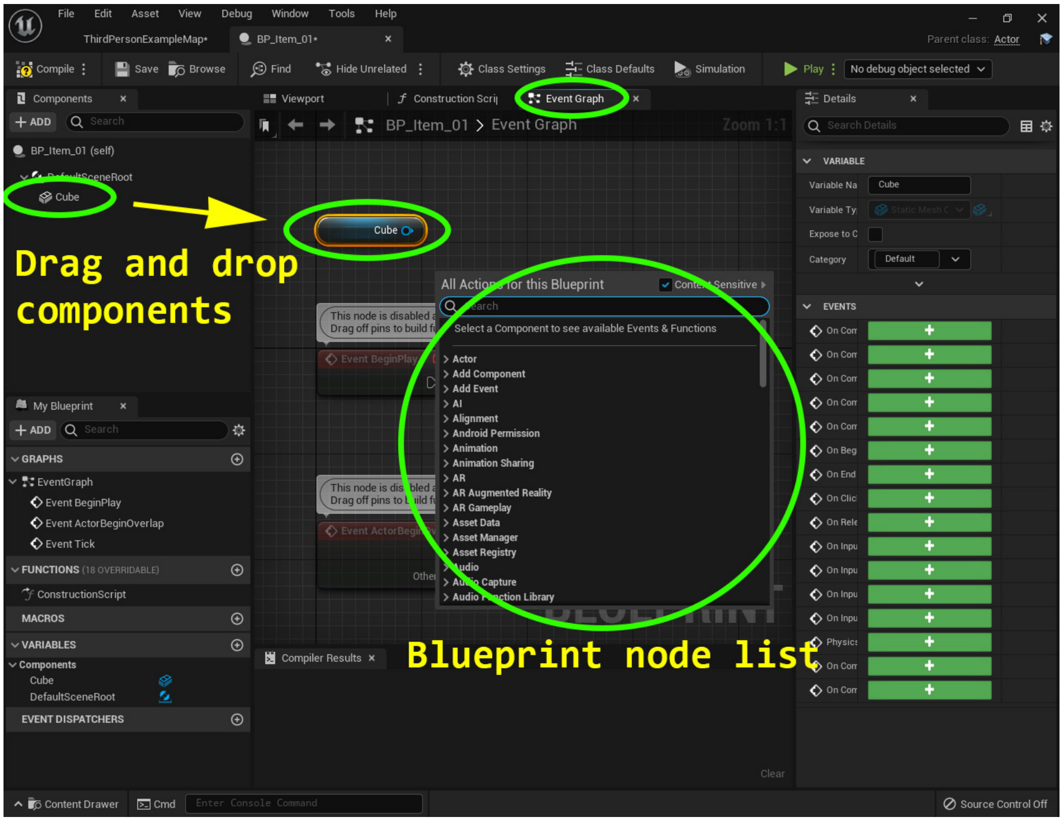


Figure 4.18: *EventGraph with a reference to Cube static mesh and right-click to get blueprint node list*

In case a user accidently closes any of these panels, the **`Components`**, **`My Blueprint`**, **`Details`** and **`Viewport`** panels have shortcuts in the **`Windows`** menu, as shown in Figure 4.19; however, to open **`EventGraph`**, the user will have to double-click on the **`EventGraph`** title in the **`My Blueprint`** panel in the **`Graphs`** section:

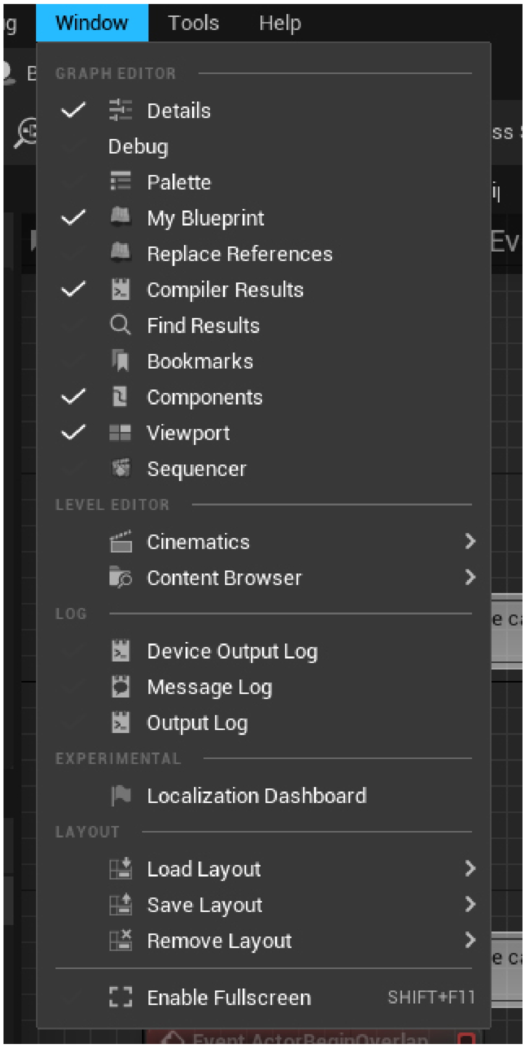


Figure 4.19*: Windows menu in Blueprint editor*

Now, let's focus on the **`Component`** and **`Viewport`** panels. There is a hierarchical relation between components in the **`Component`** panel. Each object inherits the properties of its parents. You can simply *grab* a component in the **`Component`** panel with a left-click and then drag it onto other components there and create a parent and child hierarchy. For example, as Figure 4.20 shows, the *Hierarchical* relationship between components in the component's panel in which the **`Sphere`** static mesh component is the child of **`Cube3`** static mesh component, which means *any* change on **`Cube3`** will be applied on **`Sphere`** as well at the same time, so if we move **`Cube3`**, then **Sphere** will move as well. Also, any change in its rotation and size will affect the child's objects:

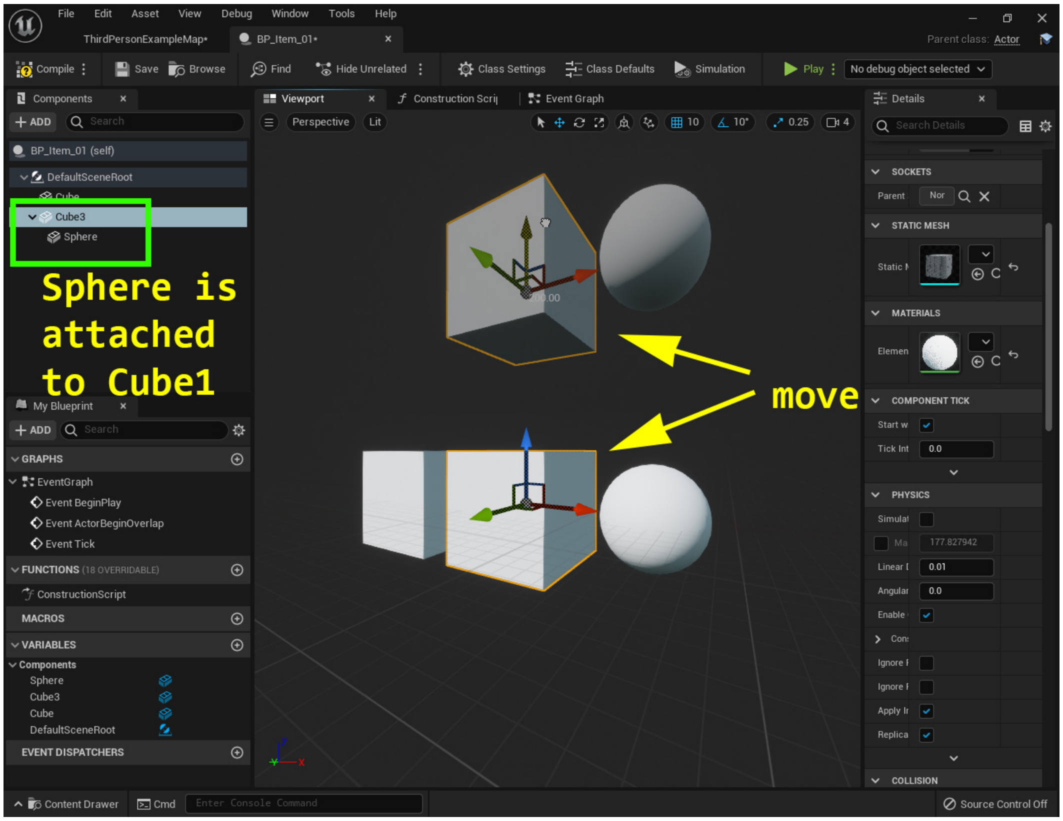


Figure 4.20: Sphere static mesh component is a "child" of Cube3 static mesh component and at the same time, Cube3 static mesh component, is the parent of Sphere static mesh component

Hierarchical relationship between components is critical in the process of designing a game asset when using many components, like static meshes, light sources, and audio components. A wise design of parent and children's components can beautifully perform game scenarios when the blueprint object is in the scene. Suppose you need to make a character like assassin creed's enemy units that shoot arrows of fire on the player. The blueprint object has a static mesh component for the character, another static mesh component for arrow, a **Point Light** component for light of fire, a Niagara Particle System component for the fire itself, and finally, the arrow static mesh component. So, in case of shooting the arrow, the arrow static mesh component must be *parent* of light and particle system component because when the arrow moves, all children will move at the same time.

Also, the hierarchy rule is valid for the blueprints that are in the scene; you can simply click on each blueprint in the **World Outliner** panel and drag them onto other ones to make parent and child hierarchy. This is very useful when relocating or rotating groups of objects together and making complex assets. *Figure 4.21* shows an example of making an asset by using only one blueprint:

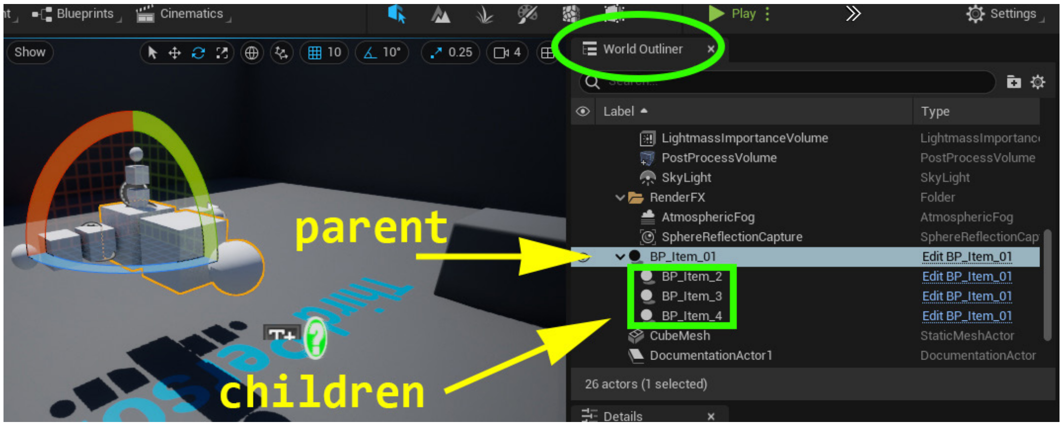


***Figure 4.21**: The BP_Item_01 is used as parent for 3 other BP_Item_01 blueprints with different sizes*

Similar to the blueprint editor, a user can add blueprint components to blueprint on the stage in the main editor and customize them. Let's try one of them; as shown in *Figure 4.22*, choose the blueprint object, click on the **+ADD** button in the **Details** panel, and then select the **Rotating Movement** component. This component is designed to rotate the assigned blueprint object in runtime:

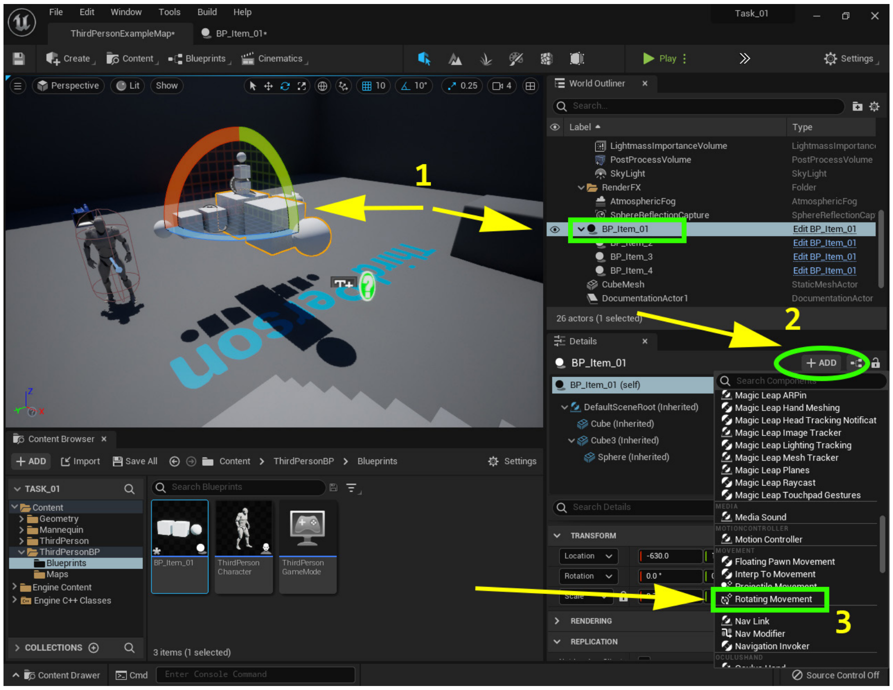
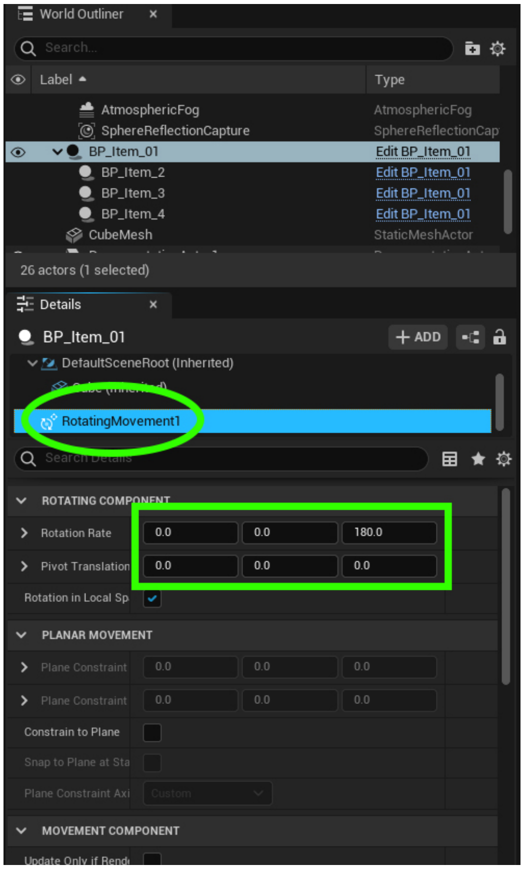


***Figure 4.22**: Add Rotating Movement component to blueprint object in scene*

When you press **Play** at the top, you will find that the object rotates around Z axis in the scene. Now, press *Esc* to return to the editor and click on the **Rotating Movement** component in your blueprint details panel, as shown in *Figure 4.23*:



World Outliner

Label	Type
AtmosphericFog	AtmosphericFog
SphereReflectionCapture	SphereReflectionCap
BP_Item_01	Edit BP_Item_01
BP_Item_2	Edit BP_Item_01
BP_Item_3	Edit BP_Item_01
BP_Item_4	Edit BP_Item_01
CubeMesh	StaticMeshActor

26 actors (1 selected)

Details

BP_Item_01

DefaultSceneRoot (Inherited)

Cube (Inherited)

RotatingMovement1

ROTATING COMPONENT

Rotation Rate	0.0	0.0	180.0
Pivot Translation	0.0	0.0	0.0
Rotation in Local Sp	✓		

PLANAR MOVEMENT

Plane Constraint	0.0	0.0	0.0
Plane Constraint	0.0	0.0	0.0
Constrain to Plane			
Snap to Plane at Sta			
Plane Constraint Axi	Custom		

MOVEMENT COMPONENT

Update Only if Rend

Figure 4.23: Rotating Movement component properties in Detail panel

As you see, there are a number of properties there to change rotation rat, pivot, and so on. There is a similar component called **InterpToMovement**, which results in the movement of blueprint with the same properties. As a practice, remove the **Rotating Movement** component and add **InterpToMovement**; then, add two points in the **Control Points** property and test the result.

Adding mesh to blueprint

Making assets with **Cube**, **Sphere**, and **Cone** static mesh component is interesting, but how about importing your own mesh into the engine with materials and animations? Suppose we want to import and replace **Cube** static mesh component in **BP_Item_01** with a FBX mesh that we already designed in a 3D application. We use the **`Abandoned Cottage House`** 3D model from **https://free3d.com/**, which is free to download and use.

Now, switch to main editor and navigate to the **`ThirdPersonBP`** folder in content browser. When you select a folder in content browser, it automatically shows the path of that folder on top of the panel. Now, right-click in the content browser panel and select **`New Folder`**, as shown in *Figure 4.24*, to create a new folder and then rename it to **`Mesh`**:

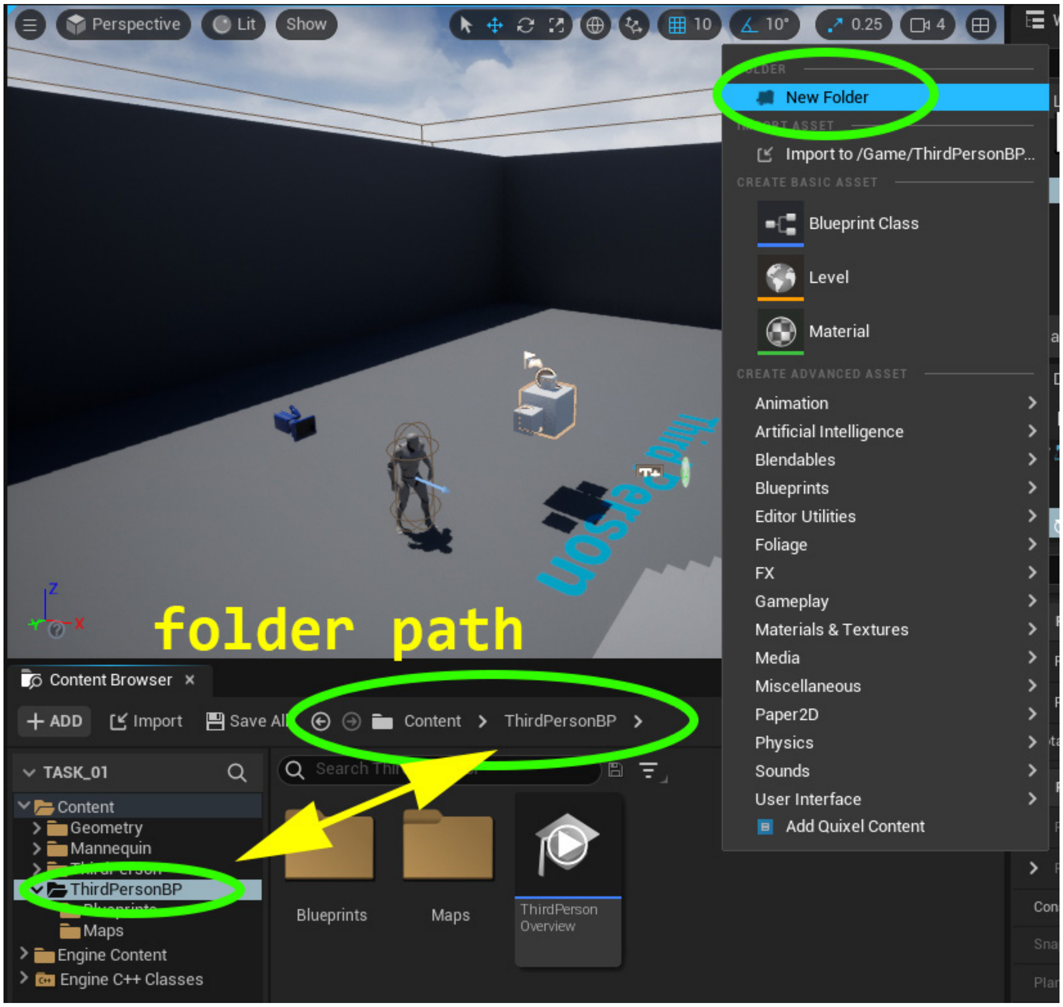


Figure 4.24*: Folder path is always shown on top of content browser panel when you select each folder in there*

Then, select the **`Mesh`** folder, and then select import, as shown in *Figure 4.25*. Next, navigate to the location of the FBX model, select it, and click on **`Import`**. You can drag and drop the FBX model from windows into the Unreal Engine as well:

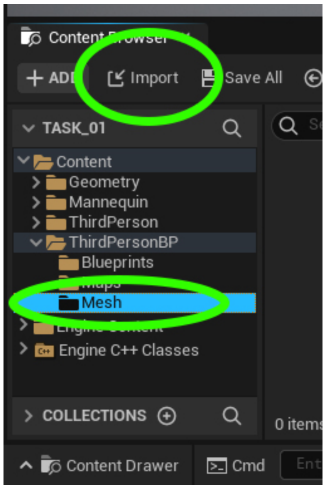
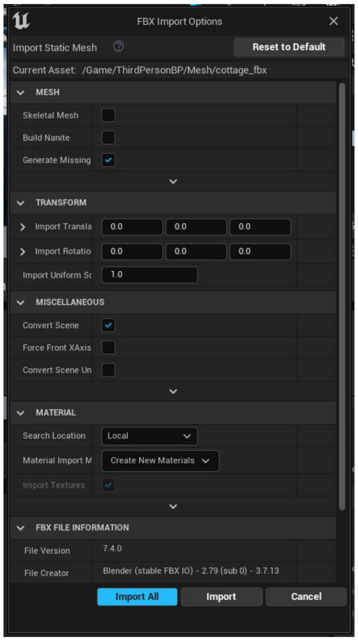


***Figure 4.25**: To add extra assets to your project, choose the desired folder in content browser as the destination of files and then click on import*

The engine will open the **`FBX Importer Options`** windows, as shown in *Figure 4.26*, with options to import 3D models known as **`Static Mesh`** and **`Skeletan mesh`** objects and animation objects:


FBX Import Options

Import Static Mesh — Reset to Default

Current Asset: /Game/ThirdPersonBP/Mesh/cottage_fbx

MESH
- Skeletal Mesh ☐
- Build Nanite ☐
- Generate Missing ✓

TRANSFORM
- Import Transla 0.0 0.0 0.0
- Import Rotatio 0.0 0.0 0.0
- Import Uniform Sc 1.0

MISCELLANEOUS
- Convert Scene ✓
- Force Front XAxis ☐
- Convert Scene Un ☐

MATERIAL
- Search Location: Local
- Material Import M: Create New Materials
- Import Textures ✓

FBX FILE INFORMATION
- File Version: 7.4.0
- File Creator: Blender (stable FBX IO) - 2.79 (sub 0) - 3.7.13

Import All | Import | Cancel

***Figure 4.26**: FBX Import Option windows*

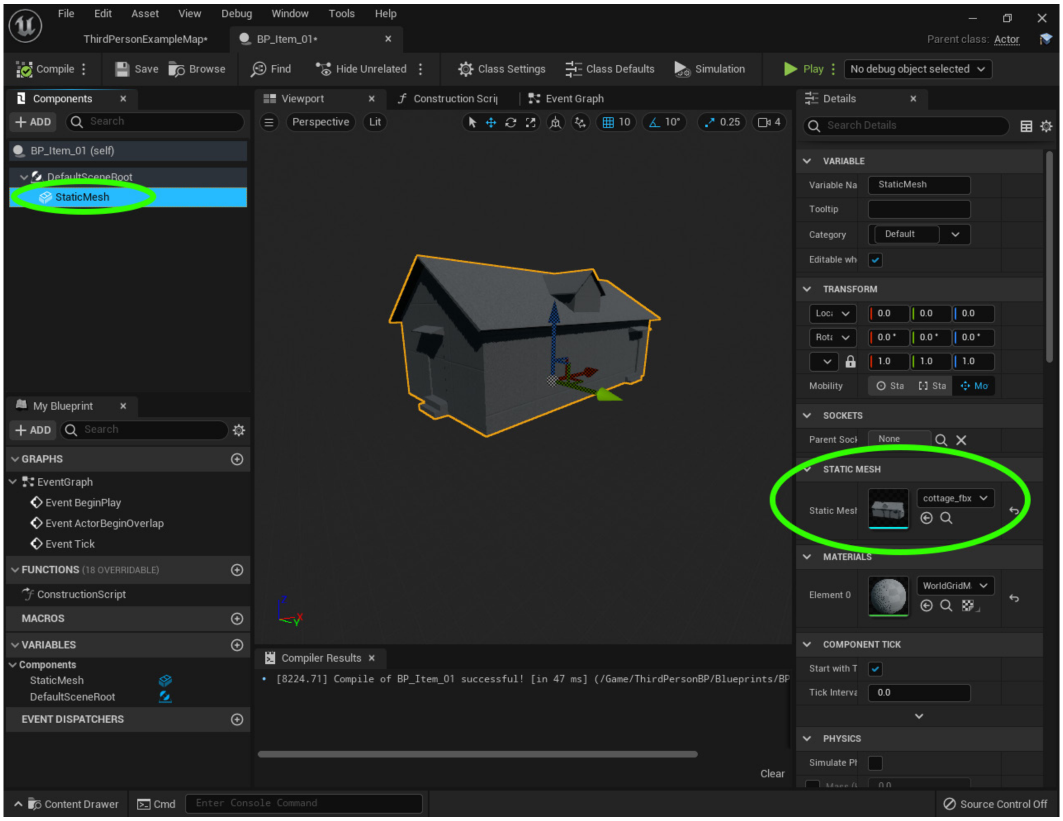
Here are the options for importing **`Static Mesh`** object into Unreal Engine 5:

- **`MESH>Skeletal Mesh`**: The engine will attempt to import the FBX file as a **`Skeletal Mesh`**, and the import options will switch to handle importing the FBX file as a **`Skeletal Mesh`**.
- **`MESH>Build Nanite`**: Nanite is a virtualized geometry system that uses an engine's internal mesh format and rendering technology to render pixel scale detail and high object counts. Nanite's data format is highly compressed and supports fine-grained streaming with automatic level of detail. Engine. By clicking on this option, you can activate Nanite procedures to render you mesh in the scene.
- **`MESH>Generate Missing`**: The engine will automatically generate collision on your mesh.
- **`TRANSFORM`**: By default, your static mesh will inherit transform data from the original 3D design application. Sometimes, you may need to alter the setting and for example, rotate your import object 180 on *Y* axis to match the game scene. These options can change values of original location, rotation, and uniform scale of the imported static mesh.
- **`MATERIAL>Search Location`**: It defines the location to search material.
- **`MATERIAL>Import method`**: It defines the actual material file that will be applied to static mesh after importing it. This option will affect Import texture, which allows you to import texture files into engine.
- **`MATERIAL>Invert Normal Maps`**: This option with invert the Normal map applied to the object in 3D modeler application.
- **`MATERIAL>Reorder Material to FBX Order`**: The 3D model sometimes has more than one material involve. This option reorder materials to support flexibility.
- **`FBX FILE INFORMATION`**: It shows built data of 3D model before import.

Keep in mind that after you import any FBX object into the engine, like **`cottage_fbx.fbx`** in our example, it will be translated to a different file format known as **`.uasset`**, so if you check content folder form a windows browser, you will find **`cottage_fbx.uasset`**, and this file can't be import to any other 3D application. The **`.uasset`** is just valid with the engine.

Now, double-click on **`BP_Item_01`**, and in blueprint editor in component panel, right-click on **`Cube`** and select **`delete`**. This way, the component will be deleted from the **`Blueprint`** and will disappear from **`Viewport`** panel. Now, add the **`Static`**

`Mesh` component and then select **`cottage_fbx`** in the **`StaticMesh`** section from the **`Detail`** panel, as shown in *Figure 4.27*:



```
[8224.71] Compile of BP_Item_01 successful! [in 47 ms] (/Game/ThirdPersonBP/Blueprints/BP
```

Figure 4.27: Add Rotating Movement component to blueprint object in scene

When you switch back to editor, you will find that all instances of **`BP_Item_01`** are replaced with imported static mesh.

Adding physics

When players play games like *Counter Strike*, the realism features of actions and reactions from or with environment is a critical feature of game. These features are calculated and provided by a *Physic* simulator program, which has processing support from graphic card and CPU by using special libraries in the operating system, which mostly are part of graphic card drivers and update. Some physical simulations meet heavy processing with both CPU and GPU, which serve scientific, artistic, and business scenarios. The good news is Unreal Engine has one of the most complete set of tools for simulating physics on collisions, materials, character's mesh, physical force, wind, clothing, ray casting, damping and frictions, destruction geometries and physical constraint. We don't have enough space in this book to cover all of these, but as an introduction, let's check *physics constraints* and use them in an example.

Referring to what we covered earlier in the chapter, make a new actor blueprint called **`BP_Item_02`** and add a **`Sphere`** static mesh component to it; finally, place two instances of this blueprint in the scene with different scales, as shown in *Figure 4.28*:

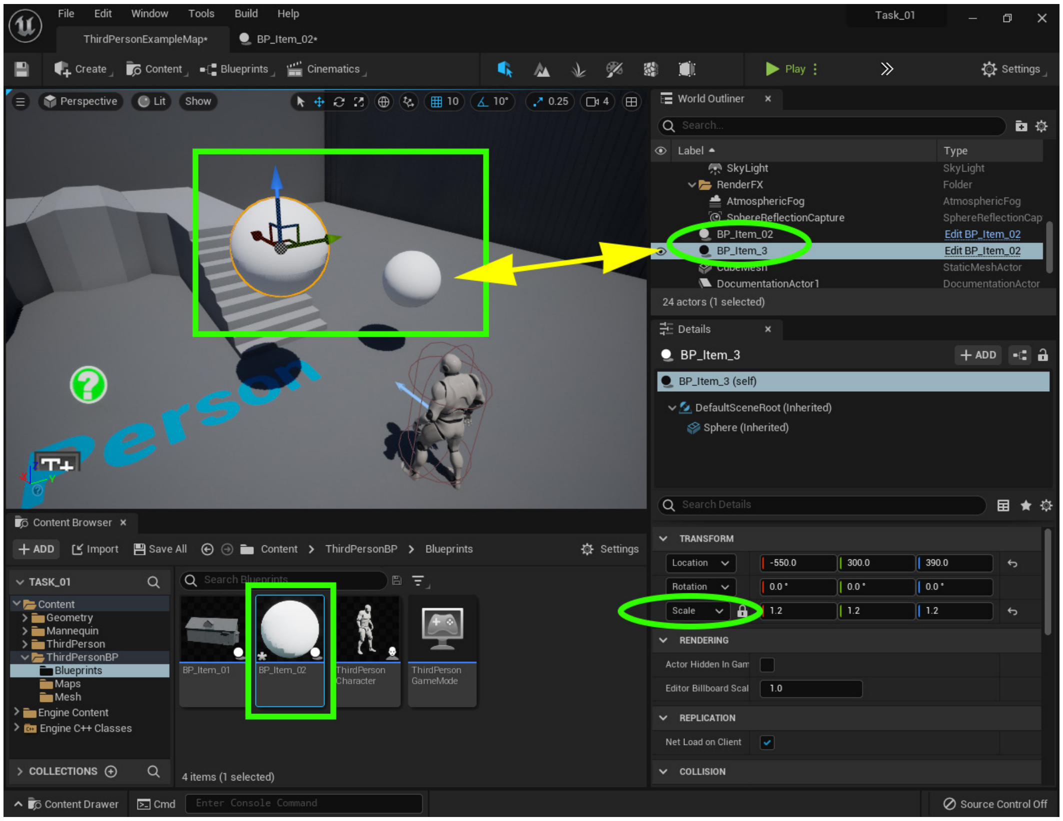


***Figure 4.28**: Making new actor blueprint "BP_Item_02"and placing two instances on the stage*

Now, if you press the **Play** button, the **BP_Item_02** blueprints will stay at their locations without any change, as shown in *Figure 4.29*:

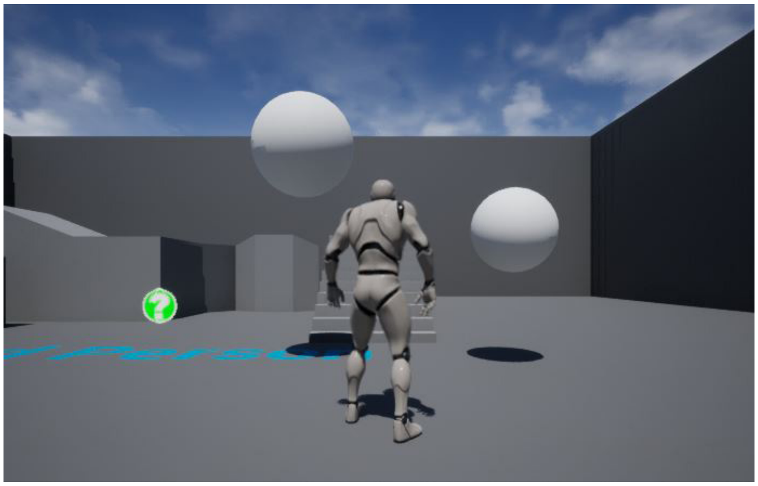
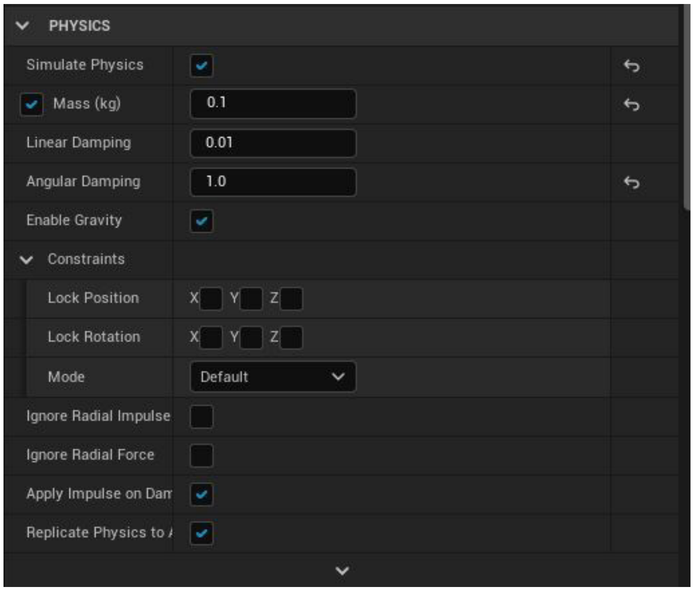


***Figure 4.29**: Scene with two instances of VP_Item_02*

Return to the main editor by pressing escape and open the **BP_Item_02** blueprint editor. Select the **Sphere** static mesh component in the **Components** panel and check **Simulate Physics** in the **PHYSICS** properties of the **Details** panel, as shown in *Figure 4.30*. This will activate an accurate physical-based simulation of gravity by using the engine`s tools designed for this feature. User can tick **Mass** and change the object weight by kilograms:



PHYSICS

Simulate Physics	✓
✓ Mass (kg)	0.1
Linear Damping	0.01
Angular Damping	1.0
Enable Gravity	✓
Constraints	
Lock Position	X Y Z
Lock Rotation	X Y Z
Mode	Default
Ignore Radial Impulse	
Ignore Radial Force	
Apply Impulse on Dam	✓
Replicate Physics to /	✓

***Figure 4.30**: PHYSICS properties in Details panel with customized Mass as 0.1 Kg*

When you press **`Play`**, the **`Spheres`** *fall dawn* to the scene, which means they accept engine's simulated gravity force and respond by falling towards the surface of the scene, as shown in *Figure 4.31*:

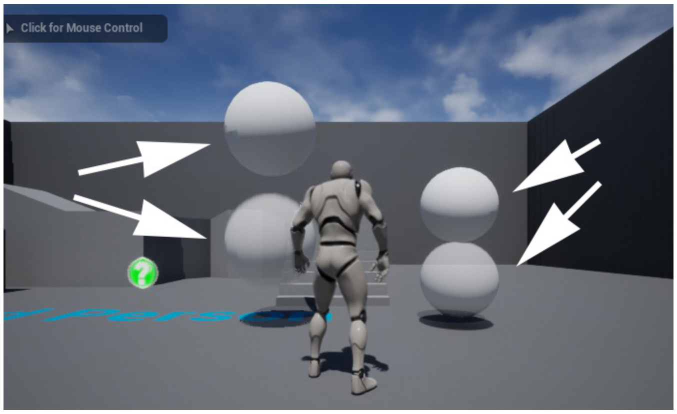
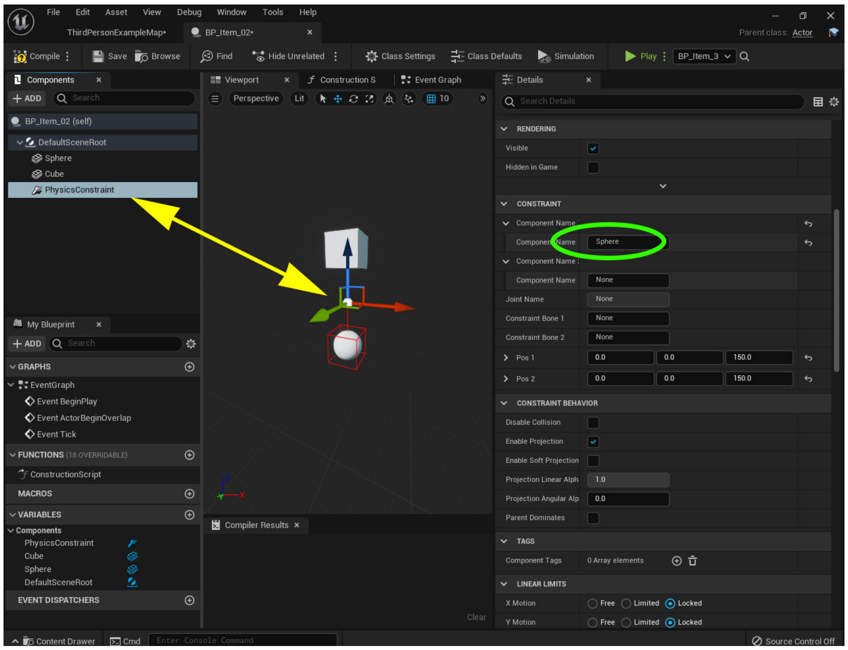


Figure 4.31*: Simulate gravity, by activate Simulate Physics check box in Details panel for each Sphere*

Now, switch back to **`BP_item_02`** blueprint editor, add a **`Cube`** static mesh component to this blueprint, and move it above the **`Sphere`**. Then, add one **`PhysicsConstraint`** component to your blueprint, select it, and place it between **`Cube`** and **`Sphere`** static mesh components. Then, in the **`Details`** panel, put **`Sphere`** in one of the **`Component Name`** text boxes, as shown in *Figure 4.32*:



RENDERING
Visible ✓
Hidden in Game ☐

CONSTRAINT
Component Name 1
Component Name: Sphere
Component Name 2
Component Name: None
Joint Name: None
Constraint Bone 1: None
Constraint Bone 2: None
Pos 1: 0.0 0.0 150.0
Pos 2: 0.0 0.0 150.0

CONSTRAINT BEHAVIOR
Disable Collision ☐
Enable Projection ✓
Enable Soft Projection ☐
Projection Linear Alph: 1.0
Projection Angular Alp: 0.0
Parent Dominates ☐

TAGS
Component Tags: 0 Array elements

LINEAR LIMITS
X Motion: Free Limited [✓ Locked]
Y Motion: Free Limited [✓ Locked]

Figure 4.32*: Add PhysicsConstraint component and place it between Sphere and Cube in BP_Item_02 blueprint actor*

Play the game and push **Sphere** static mesh in the scene with third person character of the game. As you see, it looks like **PhysicsConstraint** is grabbing the **Sphere** and forcing it to remain there and rotate around the **PhysicsConstraint**, as shown in *Figure 4.32*. This component works like a *joint* for any physics enabled (*Figure 4.30*) component in a blueprint object:

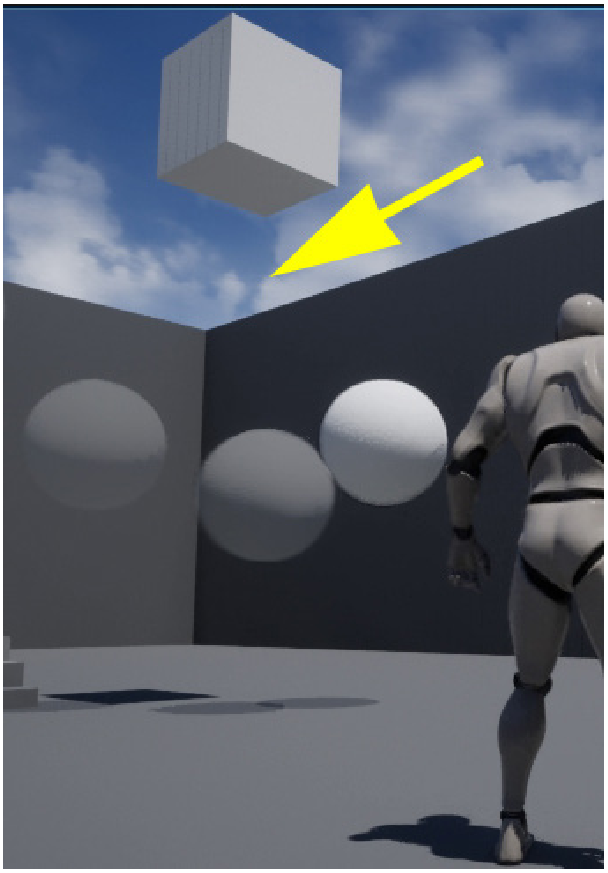


***Figure 4.33**: Location of PhysicsConstraint component and the effect on Sphere in the scene*

There are a lot of other details about physical simulations in Unreal Engine. A good source for beginning is Epic Games online documentations on Physics and its example projects.

Conclusion

The Blueprints are basic elements of each Unreal Engine application. Any designer or developer must learn how to create blueprint object, add component to it, and populate it on the scene. Blueprint components support a wide range of game elements and procedures like static mesh and physics. There is a hierarchical relationship between components of blueprint and blueprint itself in the actual scene. Some of these components can be customized with external materials, like static mesh components that a user can import and assign custom made mesh by using them. Also, blueprints can handle scripting language and perform functionalities in runtime, and we will go through the basics of blueprint scripting language in the later chapters.

Points to remember

- A good *design* of blueprint will always save time and increased performance in massive projects.
- Using **`BP_`** as the prefix for naming blueprint is an industry standard.
- The *Actor* Blueprint object is a game asset that can be placed on stage, spawned in game and can receive user interactions.
- Each change to *parents* in a hierarchical relationship will immediately affect all children.

Multiple choice questions

1. **What is the hierarchical relationship of components in Unreal Engine?**
    a. Each child is immediately affected if the parent is affected.
    b. Parent is immediately affected if any child is affected.
    c. Parents are immediately affected if all children are affected.
    d. Changes in any will immediately change others.
2. **The `PhysicsConstraint` is which of the following?**
    a. An independent object that works unlike blueprints.
    b. Designed to disable physics in game.
    c. Working like a *joint* for any physics enabled component in a blueprint object.
    d. Change the physical behavior of containing object.
3. **How many blueprint objects can we create in Unreal Engine?**
    a. There are limitations on making blueprints by default.
    b. Some blueprints are just working with special components, so we can't make them until we make those components.

c. It depends on your system features, but there are no limitations on making them.

d. Only a minimum number of blueprints are allowed in Unreal Engine.

Answers

1. **a**
2. **c**
3. **c**

Questions

1. Refer to **https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Physics**; what is PhAT?
2. How can you calculate the tangent of an angle in a right triangle?
3. How can you use linear limits in the **`PhysicsConstraint`** component to filter motion based on axis?
4. What are angular limits in **`PhysicsConstraint`** component?

Key terms

- **Blueprint editor –** Each blueprint type has a compatible editor inside the engine to develop, design and customize blueprint game objects of that type. Example are animation blueprint editor used for animation game objects, Niagara blueprint editor used for particle system game objects, Material blueprint editor used for shader programming, and so on.
- **Blueprint component –** Standalone game objects that are attached to a blueprint object as child and perform extra functionalities for the application.
- **Component hierarchy –** The parent and child relationship within blueprint components which can be customized by user or dynamically changed in runtime.
- **Blueprint node list –** The list of blueprint functions which user can access by a right click inside the Event Graph panel.

Chapter 5
Project Templates and User Interaction

Making basic functionalities in a game application is directly related to the scenario and type of that game. For example, basic functionalities in shooter games are like moving around, shooting, and changing camera view. On the other hand, in a VR game, basic functionalities are directly dependent on VR controllers and the way the VR headset is working. Unreal Engine provides a number of templates with premade functionalities that match the template scenario. This can save a lot of time on prototyping and making sample projects from scratch.

In this chapter, we will go through default templates of the engine and review their details. Then we develop user input functionalities by learning basics of blueprint programming which is based on blueprint implementations which we learned from previous chapter. Then we will create a simple game asset with collision detection by learning more about blueprint code.

Structure

In this chapter, we will discuss the following topics:

- Using templates in Unreal Engine 5
- Adding user interaction to blueprint
- Developing code in blueprint

- Simple collision detection
- Making simple game assets

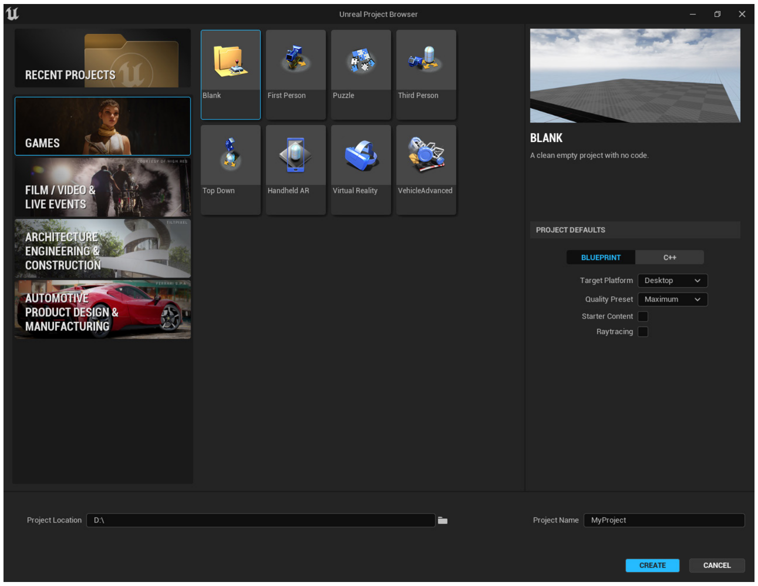
Objectives

In this chapter, we will learn about the templates that exist in Unreal Engine 5 and explain their usage on making game and applications. Then, we will learn how to develop our first code in blueprint and handle user interactions with application. A user can press keys on the keyboard, controller, or mouse to make changes inside their application, based on how they program those key events.

By the end of this chapter, you will be able to create and program a blueprint asset inside the engine and use it based on your preferences and code.

Using Templates in Unreal Engine 5

As we mentioned in *Chapter 4, Using Blueprints*, by making a new project in Unreal Engine, the user must choose one **`Project Template`** at the beginning. The engine provides four categories of project templates for users, and each one contains several project templates. Each **`Project template`** is a game/application prototype with basic functionalities and default setting for making a standalone game or application. They have predefined assets, customized rendering options, default map, and tools based on the category which been selected from. Press the **`Launch`** button in the **`Library`** tab of Unreal Engine 5, and you will navigate to **`Unreal Project Browser`** windows, as shown in *Figure 5.1*:



BLANK
A clean empty project with no code.

Target Platform: Desktop
Quality Preset: Maximum
Starter Content
Raytracing

Project Location: D:\
Project Name: MyProject

***Figure 5.1**: Unreal Project Browser windows in Unreal Engine 5*

After the user clicks on each category, the engine shows a number of project templates available for that category. Here is a list of categories:

- **`GAMES`**: These project templates shown in *Figure 5.2* are designed to support specific game types, including some customizable options and basic games assets. A very common template in this section is the **`Third Person`** template, which we will use in this chapter:

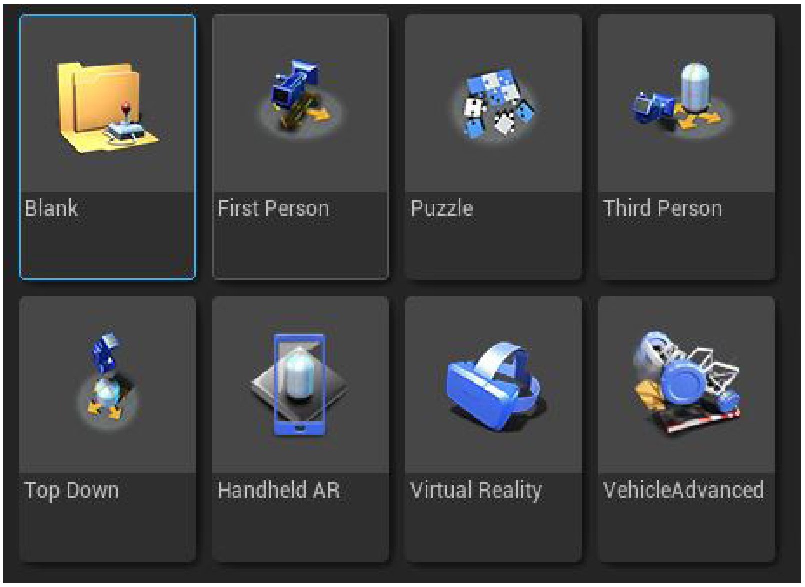


***Figure 5.2**: List of "Games" templates in Unreal Engine 5*

- **`FILM, TELEVISION/VIDEO, AND LIVE EVENTS`**: Project templates arranged and designed for **`Virtual Production`** and cinematography template are located in this category; it is shown in *Figure 5.3*:

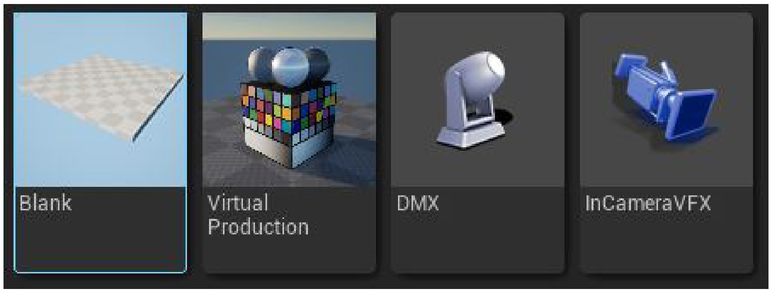


***Figure 5.3**: List of "Film, Television/Video, and Live Events" templates in Unreal Engine 5*

- **`ARCHITECTURE, ENGINEERING AND CONSTRUCTION`**: Project templates related to architecture, engineering, and Microsoft HoloLens are available in this category; it is shown in *Figure 5.4*. These templates have some special plugins like *Datasmith*, which are designed to work with *AutoDesk* applications for collaborative industrial or architectural projects:

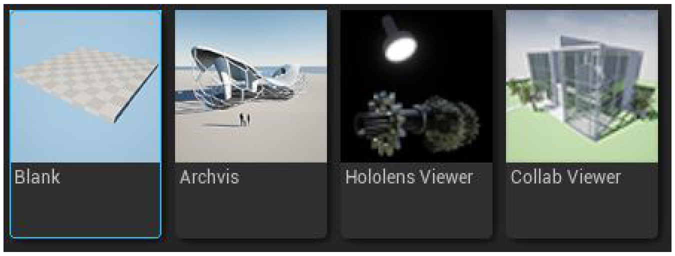


Figure 5.4*: List of "Architecture, Engineering and Construction" templates in Unreal Engine 5*

- **`AUTOMOTIVE, PRODUCT DESIGN, AND MANUFACTURING`**: These project templates support high-quality, realistic, real-time rendering, which is a brilliant feature in Unreal Engine 5. These project templates are designed for advertisement and product presentations scenarios, as shown in *Figure 5.5*:

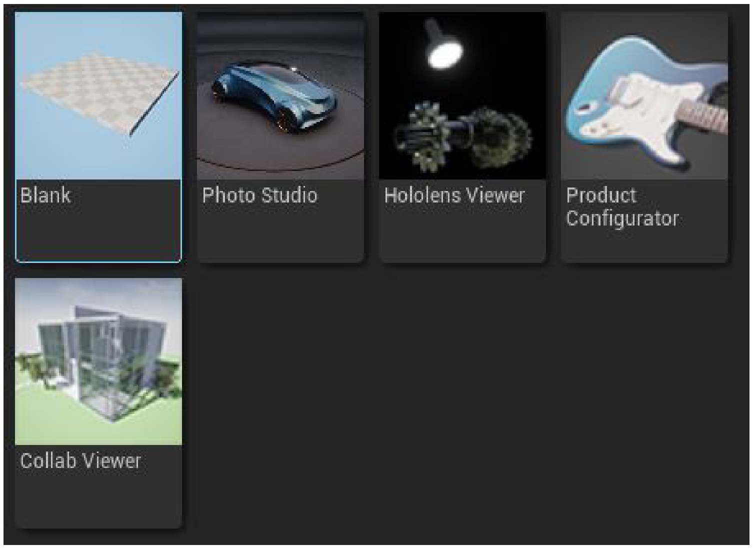


Figure 5.5*: List of "Automotive, Product Design, and Manufacturing" templates in Unreal Engine 5*

Each category has a **`Blank`** project template, which is an empty project using default rendering configurations related to chosen category and an empty map/level. The remaining project templates of each category have special features to support different scenarios, which is out of the scope of this chapter.

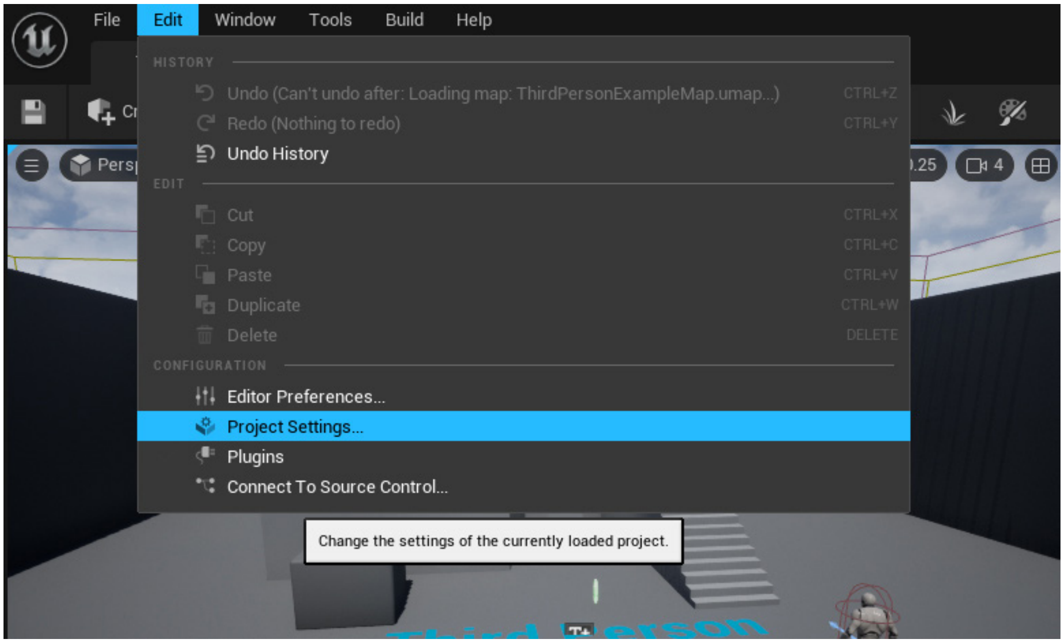
For this book, we will use the **`Third Person`** template from the **`GAMES`** category. This template is designed for developing a third person game like Tomb Raider and Assassin Creed; it has predefined features that we will use to learn basics of user interactions in Unreal Engine 5.

Click on the **`GAMES`** category and then select the **`Third Person`** template from the options; then, change the name and project path, uncheck **`Starter Contents`**, and then click on the **`CREATE`** button. The engine will build a standalone project with selected template totally in C++ programming language. Then, the engine will run **`Unreal Editor`** to open the project. The Unreal Editor is an application that provides basic and advanced tools to edit and create an application with Unreal Engine.

When your project comes up with unreal editor, press the **`Play`** button on top of the editor. The Unreal Engine will run a third person game prototype, in which user can run and jump by using keyboard and change camera view by moving the mouse. You can press the *Esc* key on your keyboard or click on the **`Stop`** button to stop the game. As a practice on what you learned from the previous chapter, make some actor blueprints, add them to the scene, and make a simple third person game.

Adding user interaction to blueprint

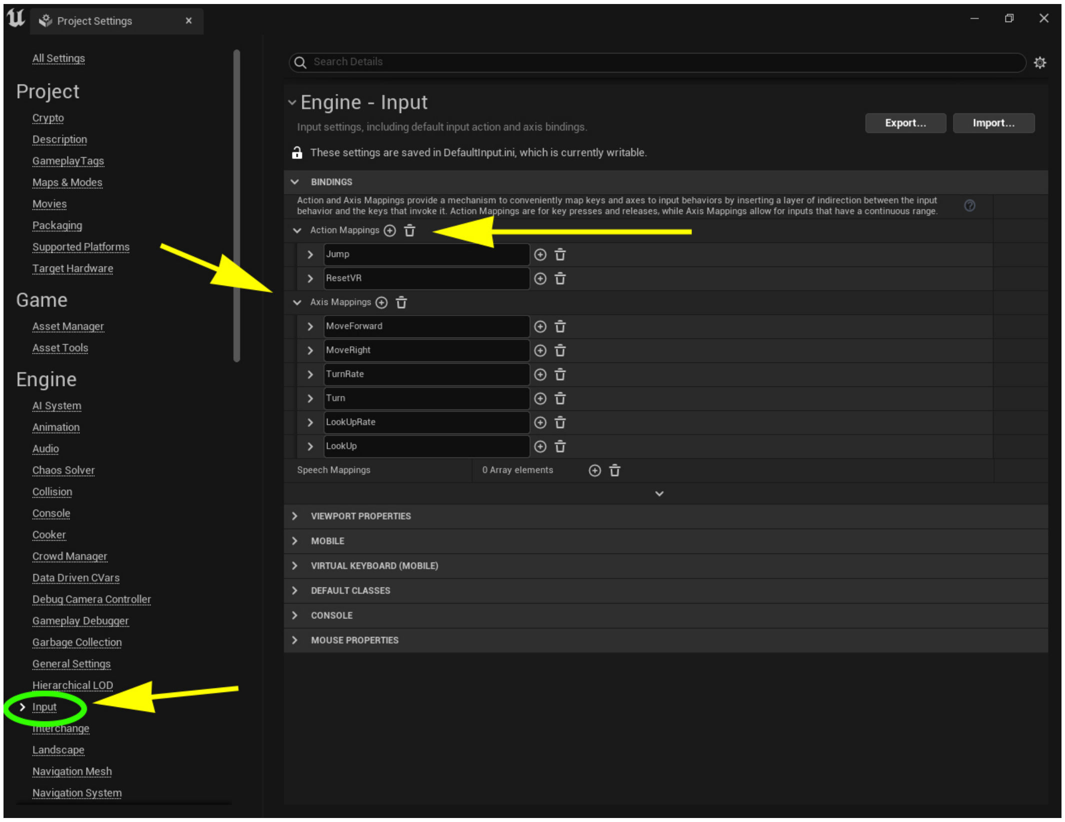
In the previous chapter, we learned how to *design* a blueprint by adding blueprint components like static mesh. Now, we would like to do some basic coding in blueprint, which is essential for everyone who likes to use unreal engines. A very good point to start with is to define and change the user interface. Click on **`Project Setting...`** from the **`Edit`** menu, as shown in *Figure 5.6*:



Change the settings of the currently loaded project.

Figure 5.6*: Project Setting in Edit menu*

The engine will open the **`Project Settings`** window, which is designed to customize features and functionalities of a project. For this chapter, we would like to add new input keys to project, so click on **`Input`**, which is located at **`All Settings`** list on the left, and then click on and expand **`Action Mappings`** and **`Axis Mappings`** drop-down list, as shown in *Figure 5.7*:



Engine - Input

Input settings, including default input action and axis bindings.

These settings are saved in DefaultInput.ini, which is currently writable.

BINDINGS

Action and Axis Mappings provide a mechanism to conveniently map keys and axes to input behaviors by inserting a layer of indirection between the input behavior and the keys that invoke it. Action Mappings are for key presses and releases, while Axis Mappings allow for inputs that have a continuous range.

Action Mappings
- Jump
- ResetVR

Axis Mappings
- MoveForward
- MoveRight
- TurnRate
- Turn
- LookUpRate
- LookUp

Speech Mappings 0 Array elements

Figure 5.7: *Access to Input setting in Unreal Engine 5*

The **`Action Mappings`** dropdown is responsible for your direct actions in game, and in this template (**`Third Person`**), actions **`Jump`** and **`Reset VR`** are already assigned by default. Click on **`Jump`** and expand its dropdown; as you see in *Figure 5.8*, the *space bar* of keyboard, gamepad face button, and a number of other machines are listed here. You can add new ones by clicking on plus with a circle button and delete them by clicking on the **`bin`** icon:

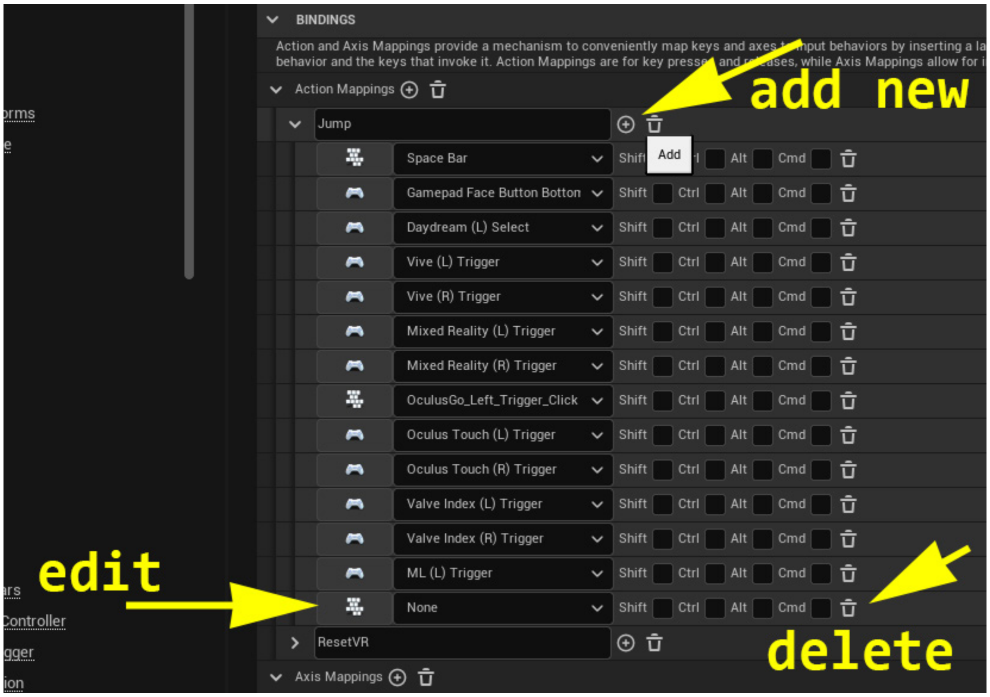

BINDINGS
Action and Axis Mappings provide a mechanism to conveniently map keys and axes to input behaviors by inserting a la
behavior and the keys that invoke it. Action Mappings are for key presses and releases, while Axis Mappings allow for i
Action Mappings
add new
Jump
Add
Space Bar
Gamepad Face Button Botton
Daydream (L) Select
Vive (L) Trigger
Vive (R) Trigger
Mixed Reality (L) Trigger
Mixed Reality (R) Trigger
OculusGo_Left_Trigger_Click
Oculus Touch (L) Trigger
Oculus Touch (R) Trigger
Valve Index (L) Trigger
Valve Index (R) Trigger
ML (L) Trigger
None
Shift
Ctrl
Alt
Cmd
edit
delete
ResetVR
Axis Mappings
Controller

Figure 5.8: Add new input for "Jump" action

To edit a key, click on **None**, and as you see in *Figure 5.9*, the engine automatically shows a list of available target machines. Select **Keyboard** and then from the list of keys, select **Backspace** just for testing. You can always return here and change the key or remove it:

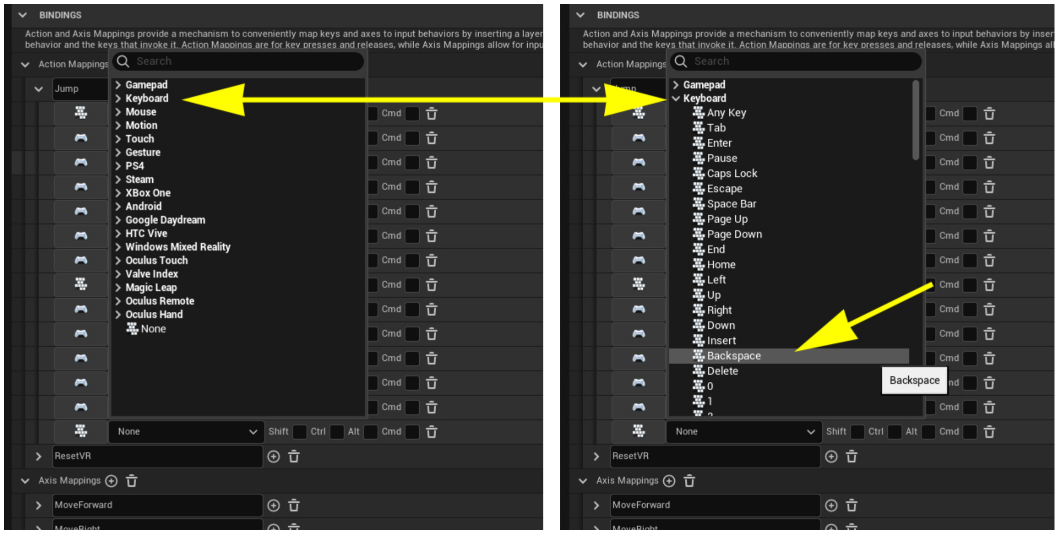
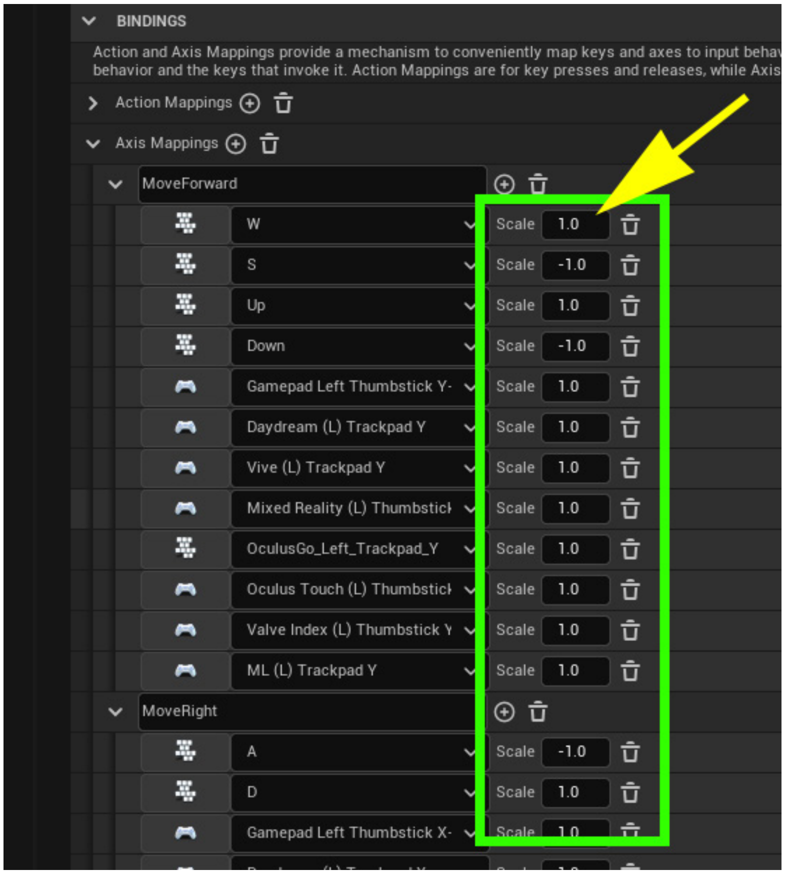


Figure 5.9: Add new input for "Jump" action

Close these windows, and the engine will automatically save your changes. Go back to the editor and press play and then press *Backspace*. As you see, the third person player character will **`Jump`** with this key as well as with the *Spacebar*. Users can assign other target machines like gamepad and test them with same result.

Now, let's go back to **`Edit`** | **`Project Settings…`** | **`Input`** and check **`Axis Mappings`**. As you can see in *Figure 5.10*, we have *W* and *A* to move forward and *S* and *D* keys to move right. The difference between **`Axis Mappings`** and **`Action Mappings`** is important to know. **`Action Mappings`** are basically support **`on and off`** interactions like **`Jump`**. When you press space bar, the character moves up, and after a delay, it moves down; you can press **`Jump`** again. The **`Axis Mappings`**, however, are best to support **`on and off`** interaction, but it is designed to send values. For example, if you press W to walk forward, as *Figure 5.10* shows, the engine will use a scale value, which is **`1`**, for walking forward. Now if you like to move backward, you have to press *S*, and as you see in *Figure 5.10*, the engine uses a scale value of **`-1`** to move the character in opposite direction of moving forward, i.e., backward. The same logic is working for the *A* and *D* buttons, which move characters left and right:



BINDINGS

Action and Axis Mappings provide a mechanism to conveniently map keys and axes to input behav
behavior and the keys that invoke it. Action Mappings are for key presses and releases, while Axis

Action Mappings

Axis Mappings

MoveForward

W	Scale	1.0
S	Scale	-1.0
Up	Scale	1.0
Down	Scale	-1.0
Gamepad Left Thumbstick Y-	Scale	1.0
Daydream (L) Trackpad Y	Scale	1.0
Vive (L) Trackpad Y	Scale	1.0
Mixed Reality (L) Thumbstick	Scale	1.0
OculusGo_Left_Trackpad_Y	Scale	1.0
Oculus Touch (L) Thumbstick	Scale	1.0
Valve Index (L) Thumbstick Y	Scale	1.0
ML (L) Trackpad Y	Scale	1.0

MoveRight

A	Scale	-1.0
D	Scale	1.0
Gamepad Left Thumbstick X-	Scale	1.0

***Figure 5.10**: Axis Mappings" need Scale value to send user interaction to engine*

Developing code in Blueprint

Now, let us assign a keyboard key and learn how to print a message on screen. First, add a new input to **`Action Mapping`**, and then assign **`My Action`** as its name; then, select **`Keyboard`** from the drop-down list of target machines and then select **`0`**, as shown in *Figure 5.11* and then close the project setting windows. The engine will automatically save your changes:

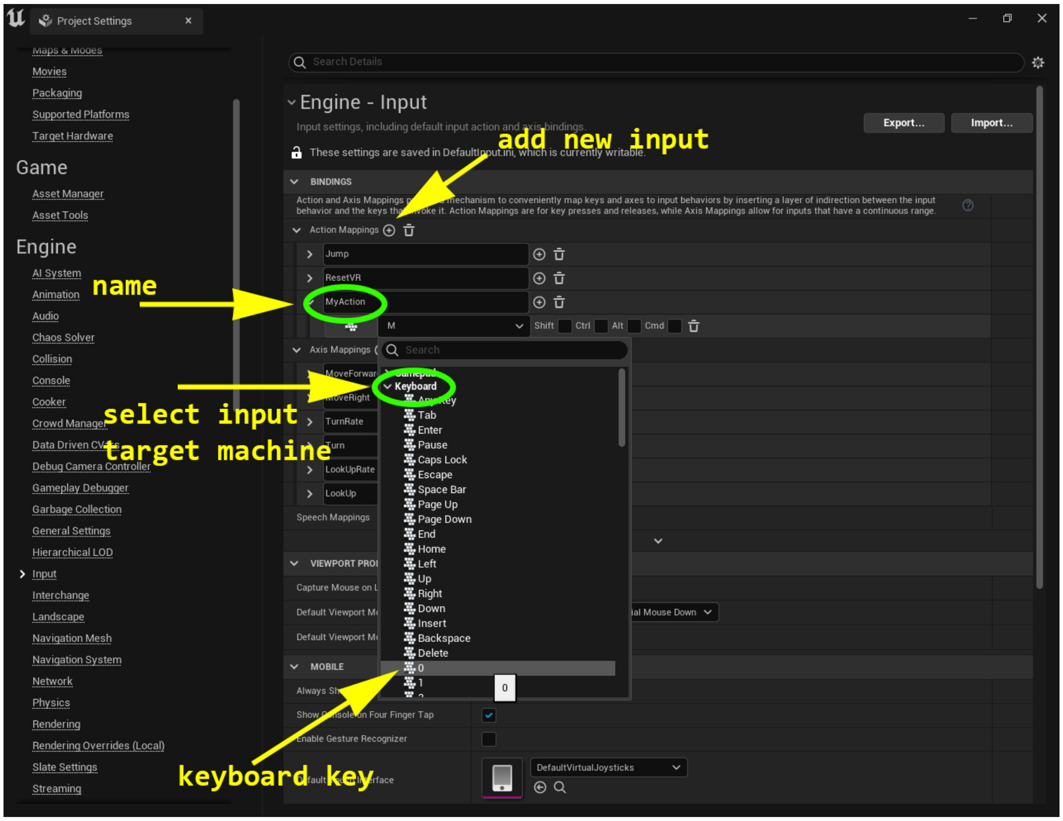


Figure 5.11*: Make new input key for keyboard*

At this stage, the engine knows that we like to use key 0 as an input. Now, we need to add code to our game to *implement* this action. Locate th ethird person blueprint in **`Content/ThirdPersonBP/Blueprints/`**and double-click on it to open. After the blueprint editor opens, click on the **Event Graph** tab, and in an empty area there, right-click and type **My Action**, as shown in *Figure 5.12*:

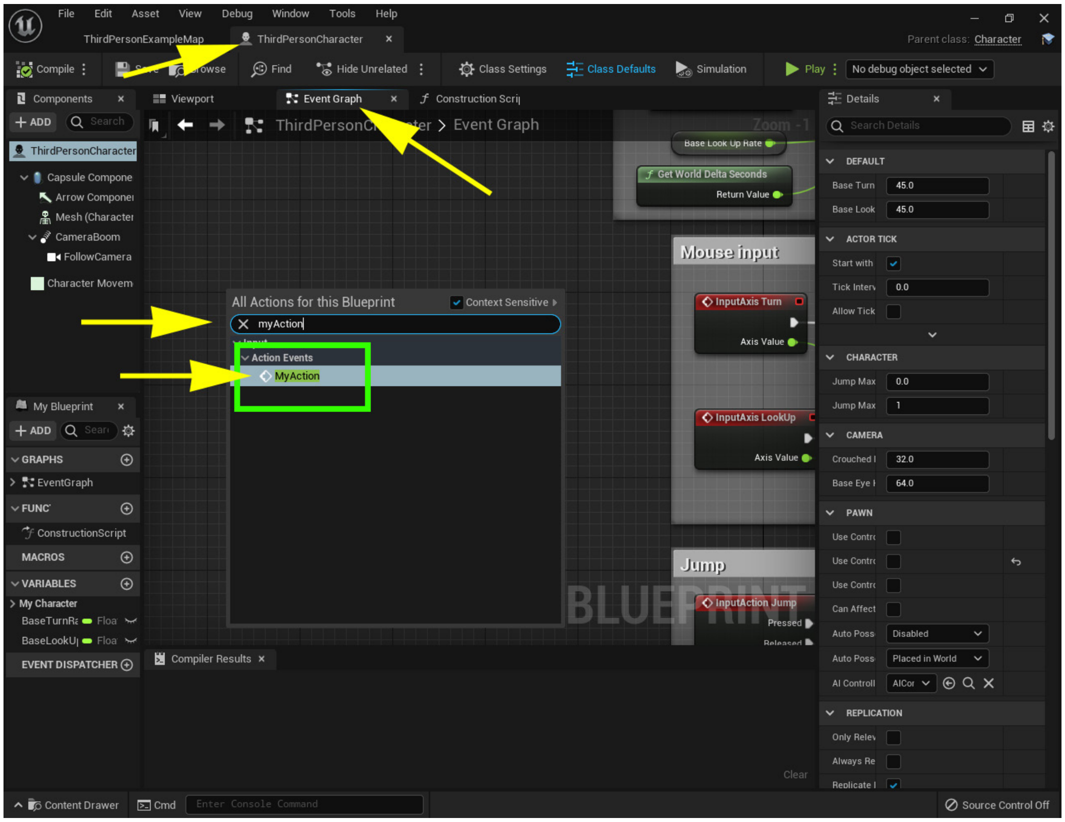


***Figure 5.12**: Navigate to Event Graph in Third Person blueprint, and then right-click and type "My Action" and click on "MyAction" in the list*

The engine will show a drop-down list with the name of input event that you already made. Now, click on it in drop-down list, and the engine automatically will add a red titled block to the **Event Graph** with the title **Input Action My Action**, as shown in *Figure 5.13*. This is a **Blueprint** block of code that is responsible for detecting press and release of key **0** from keyboard by user:

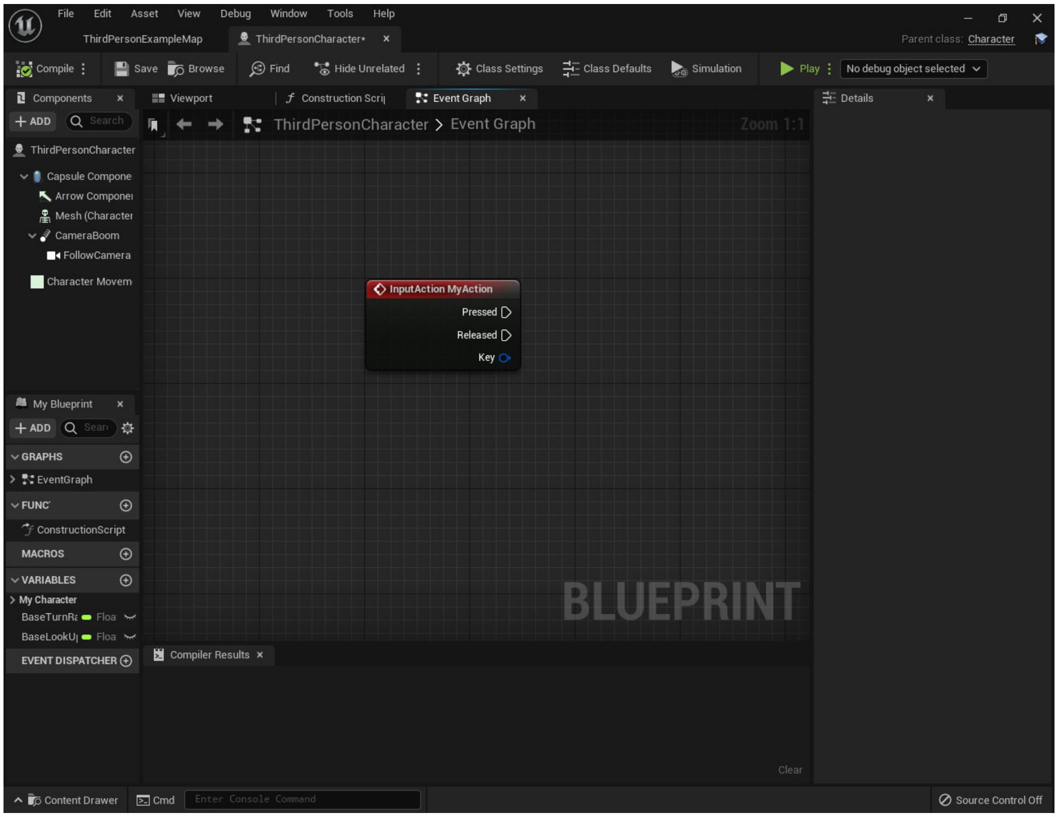


Figure 5.13: "My Action" key event after implementation in blueprint

Right-click again in front of your event block and type **`Print`** and again, the engine will automatically show a proper match for what you typed in; then, click on **`Print String`**. Now you have a new block with the title **`Print String`**, as shown in *Figure*

5.14. This is a blueprint block of code and is responsible for printing the message on the screen:

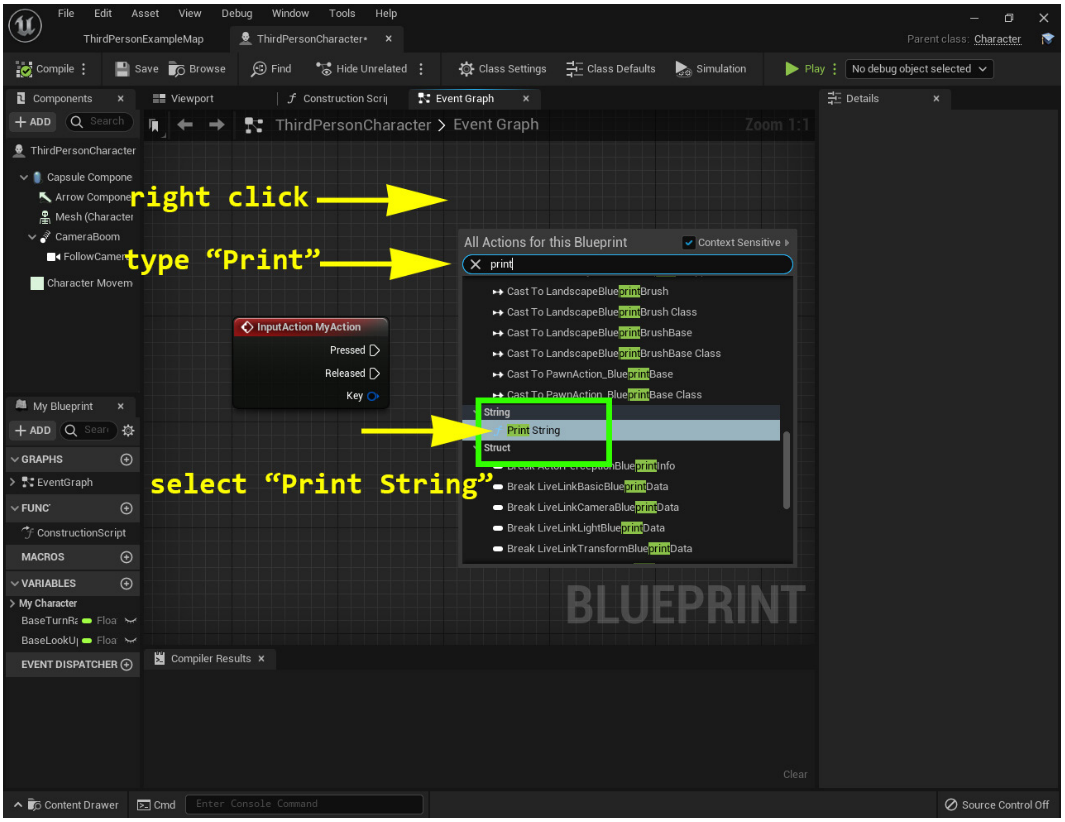

right click
type "Print"
select "Print String"
All Actions for this Blueprint
Context Sensitive
print
Cast To LandscapeBlueprintBrush
Cast To LandscapeBlueprintBrush Class
Cast To LandscapeBlueprintBrushBase
Cast To LandscapeBlueprintBrushBase Class
Cast To PawnAction_BlueprintBase
Cast To PawnAction_BlueprintBase Class
String
Print String
Struct
Break LiveLinkBasicBlueprintData
Break LiveLinkCameraBlueprintData
Break LiveLinkLightBlueprintData
Break LiveLinkTransformBlueprintData
InputAction MyAction
Pressed
Released
Key
ThirdPersonCharacter > Event Graph
BLUEPRINT

Figure 5.14: *Right-click in an empty space and then type "print" and select "Print String"*

Now we need to make an *execution line* between these two blocks. Select **`Input Action My Action`** and then click on and drag (don't release mouse button) the small triangle in front of **`Pressed`**; immediately, the engine will draw a white curvy line from that triangle to the location of the mouse pointer. Now, move the mouse over the right triangle in **`Print String`** (without releasing the mouse button). When a small green tick shape appears, release the mouse button, and the white line will automatically connect these two blocks. After this, click on and change **`Hello`** in the **`In String`** text area to **`The key is Pressed`**, as shown *Figure 5.15*. You already made your first blueprint code! Run the game and press key *0*. You will see **`The key is Pressed`** message on the screen:

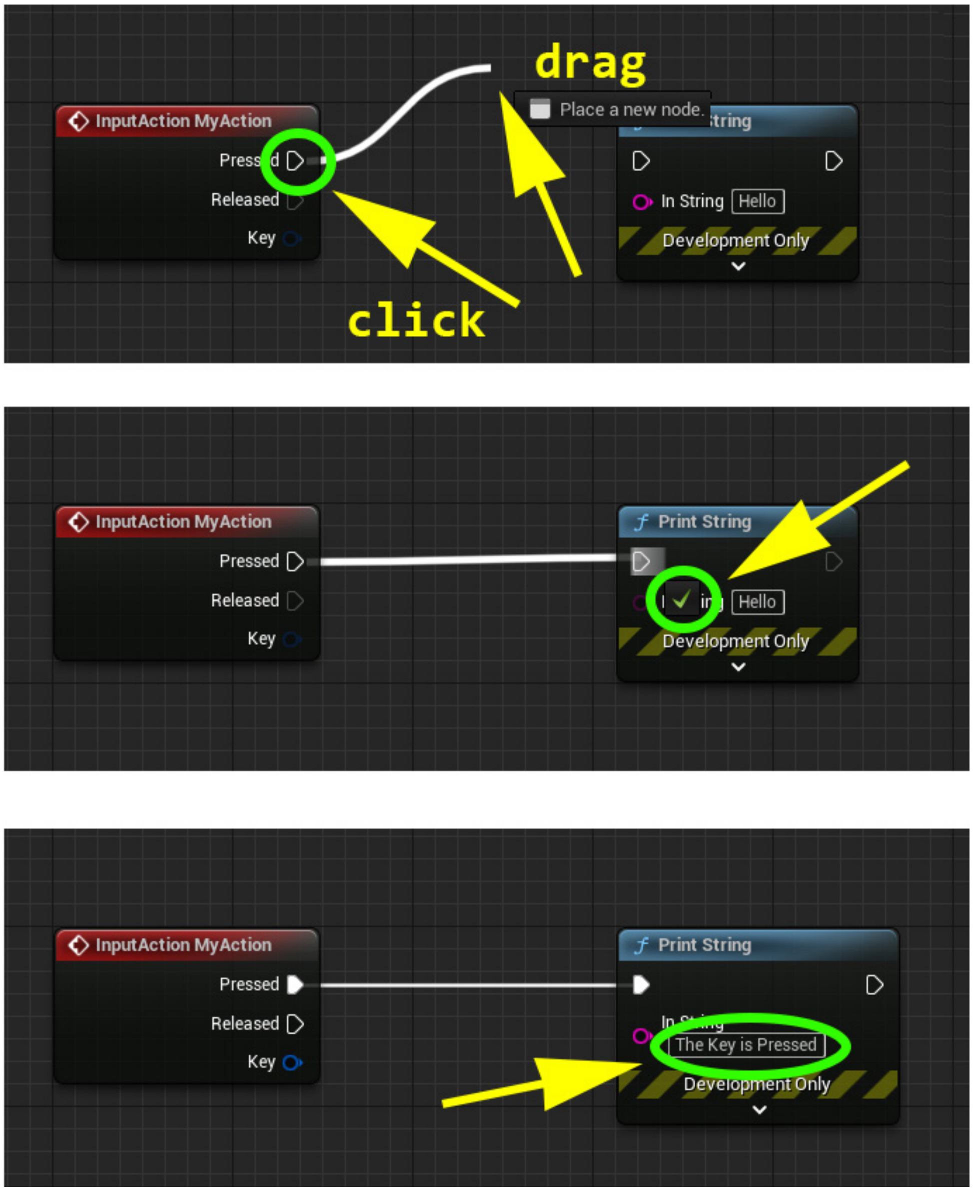
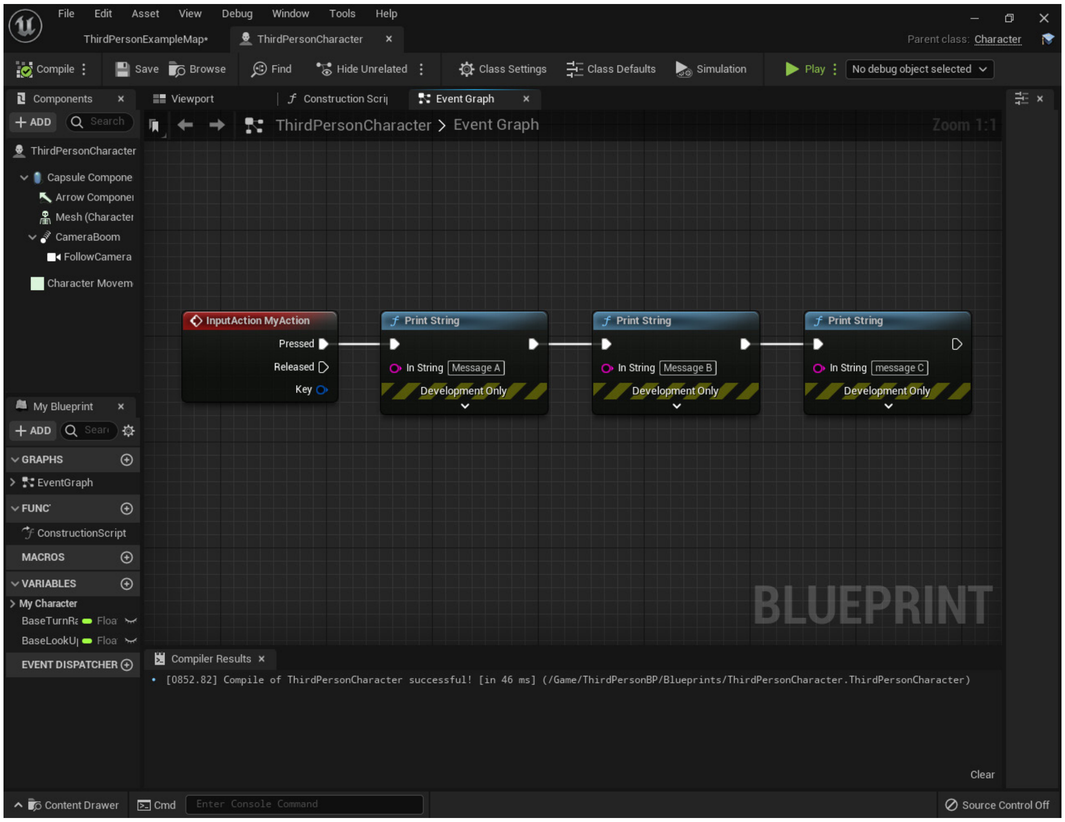


***Figure 5.15**: Click on the small triangle that has the "Pressed" title and drag a white curvy line from it; then, without releasing mouse, move it over the "left" triangle of Print String block. Then release the mouse, and block will get automatically connected by one white line. Then in the text box of Print String, type a message. When you run the game and press key 0, the message will be printed on the screen by engine.*

The white curvy line is known as **execution line**; it executes each blueprint block from left to right when you run the game. To understand this order, add two more print string blocks and connect them to the previous block, as shown in *Figure 5.16*, and run the game. User can select any one or a group of blueprint blocks and copy/

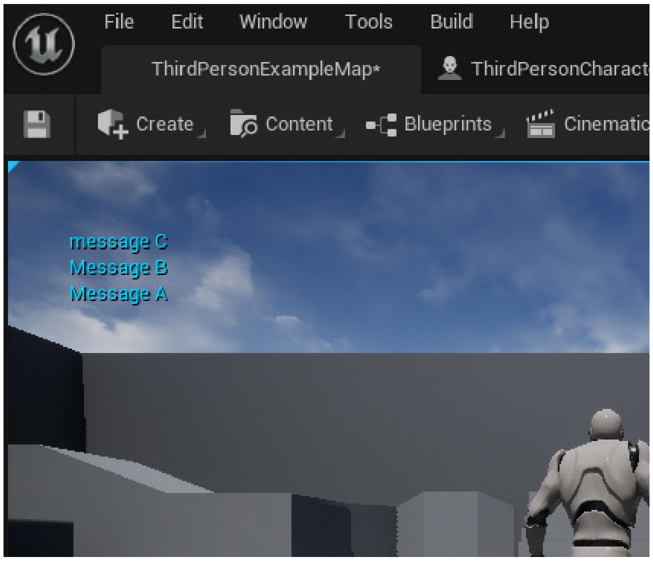
paste them by using the keyboard, or they can right-click and type **`Print`** and select **`Print String`**, as we did earlier:



```
• [0852.82] Compile of ThirdPersonCharacter successful! [in 46 ms] (/Game/ThirdPersonBP/Blueprints/ThirdPersonCharacter.ThirdPersonCharacter)
```

Figure 5.16*: Add two more Print String blueprint blocks with different message*

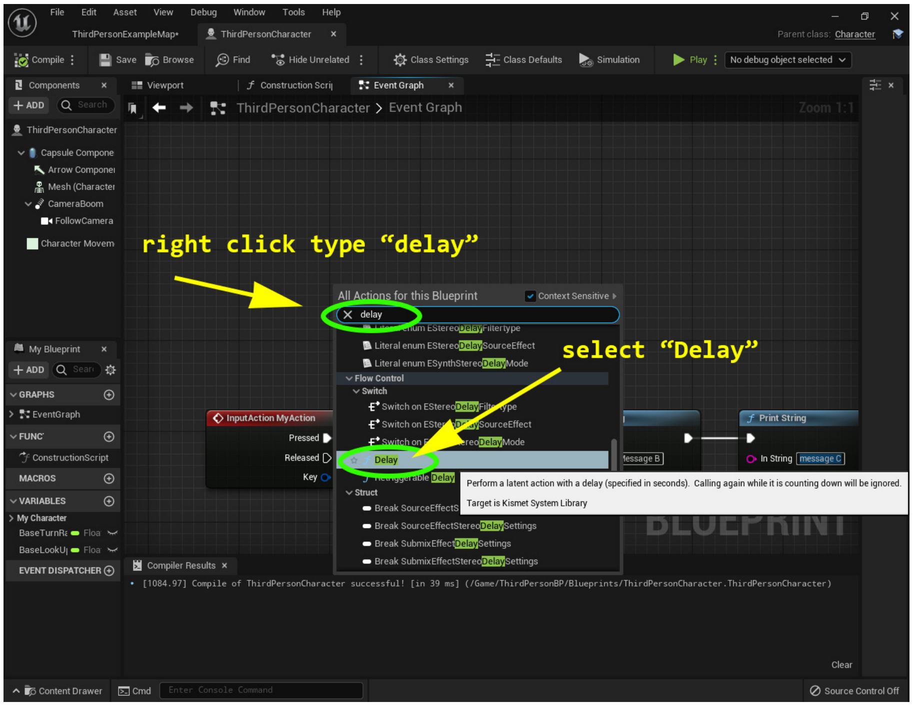
As per the order of the three messages shown in *Figure 5.17*, the execution line run prints blocks from *Left* to *Right*, so **`Message A`**, then **`Message B`**, and finally, **`message C`** get printed on the screen:



```
message C
Message B
Message A
```

***Figure 5.17**: Messages print in order after running the game*

Now, let's use another blueprint block known as **`Delay`** and check the execution order again. Right-click on **`Event Graph`**, type **`Delay`**, and then select the **`Delay`** command from blueprint list, as shown in *Figure 5.18*:

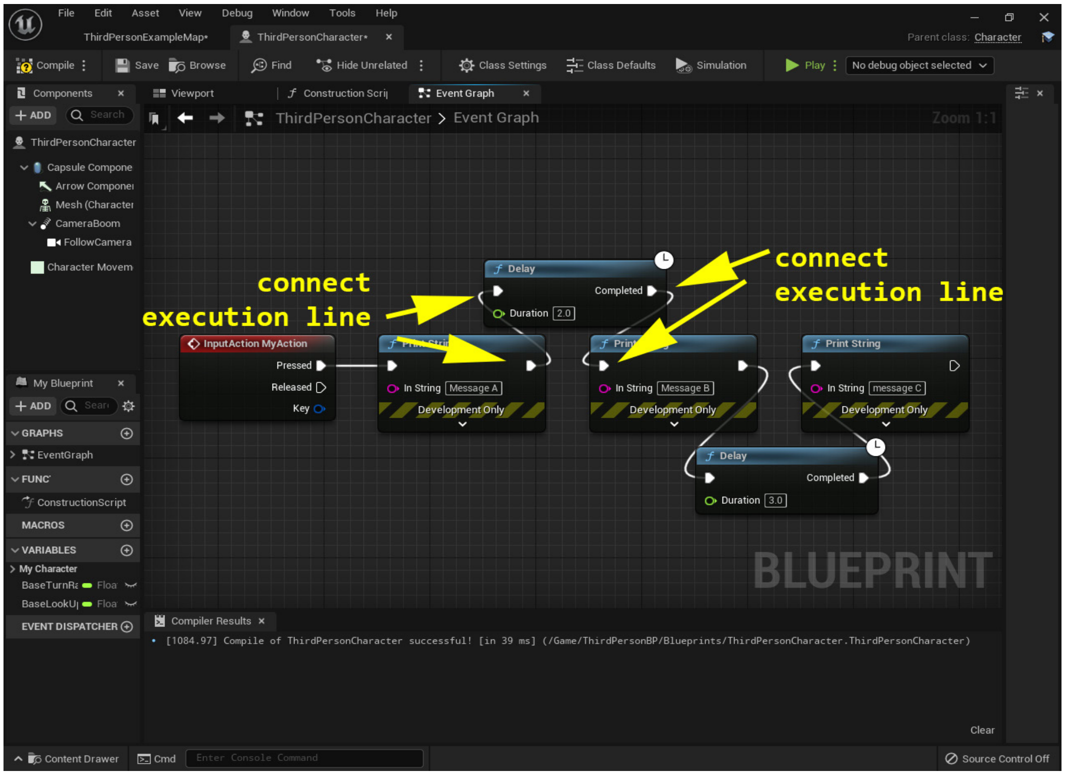


Perform a latent action with a delay (specified in seconds). Calling again while it is counting down will be ignored.
Target is Kismet System Library

```
• [1084.97] Compile of ThirdPersonCharacter successful! [in 39 ms] (/Game/ThirdPersonBP/Blueprints/ThirdPersonCharacter.ThirdPersonCharacter)
```

***Figure 5.18**: Right-click in an empty space, type "delay", then select Delay command from the list*

On the **Delay** block, locate **Duration** and change it to 2; add it between **Message A** and **Message B** by attaching execution line from the **Print String** blocks to the **Delay** block (simply click and drag execution line from each of print string blocks and attach them to delay). Now add another delay between **Message B** and **message C** with **Duration** equal to 3, as shown in *Figure 5.19*:



```
• [1084.97] Compile of ThirdPersonCharacter successful! [in 39 ms] (/Game/ThirdPersonBP/Blueprints/ThirdPersonCharacter.ThirdPersonCharacter)
```

***Figure 5.19**: After selecting Delay from list, engine will add A Delay block to event graph. Drag and Drop execution lines between previous and next blocks of print string*

When you run this code, the engine follows execution line from left to right; first, it runs the first print message, then the delay after it, which leads to a 2-second pause, and then it runs the next print message, followed by the **Delay** after it, which is 3 seconds. Finally, it runs the last print string block.

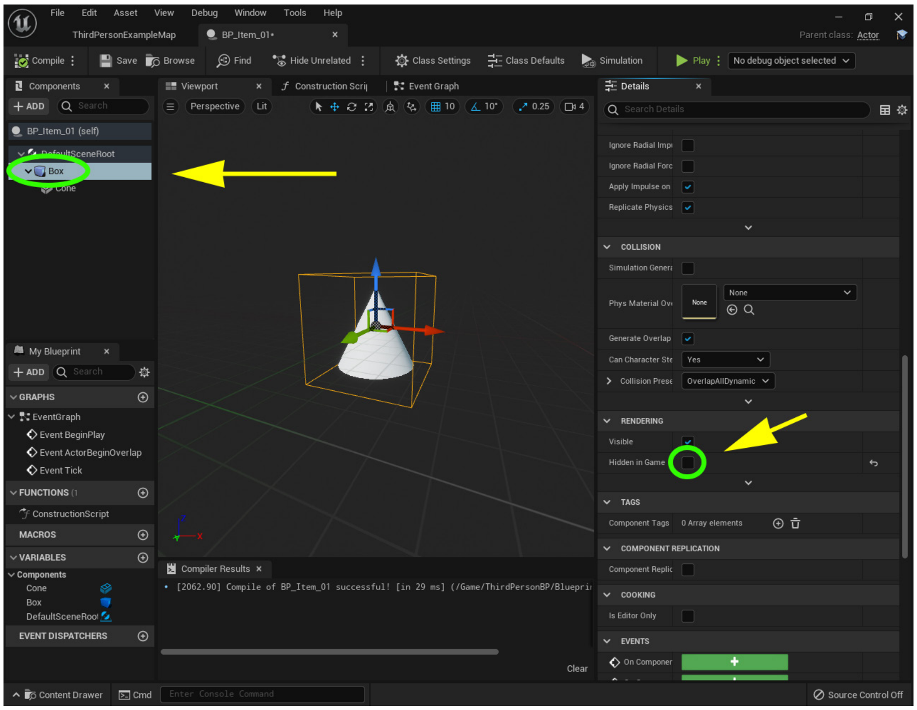
This is how you *develop* code in blueprint. Developers can add different blueprint blocks from the blueprint list and receive different results after running the code; we will cover this in further detail in the later chapters. If you are familiar with coding in a programming language like C++, Python, or JavaScript, you just need to understand two simple rules in blueprint programming: each blueprint block (like **Delay** and **Print String**, which we used) is like **A Function**, and as you already experienced, the compiler **Run** each *line* of code in traditional programming, *After each other*. In blueprint programming, the execution line (white curvy line) is *Running* each *block* of blueprint *After each other from Left to Right*. So, by using these two simple rules on your logic, you can easily adapt and use blueprint programming in Unreal Engine 5 for developing game and applications.

Simple collision detection

Collision detection is one the most important topics in game development. When (at least) two game objects collide with each other, then there is a *game event*, called **Collision**. There are methods and functions inside the engine to help developers to detect it. We touch and learn the simplest way to detect collision in this part to expand basic knowledge on blueprint programming.

Follow the given steps to do so:

1. Refer to *Chapter 4: Using Blueprints*, make an Actor blueprint, and call it **`BP_Item_01`**.
2. From the blueprint component list, add **`Box Collision`** to actor and untick the **`Hidden in Game`** properties from the **`Details`** panel located in the **`REBDERING`** section for it.
3. Add a **`Cone`** static mesh and change its scale to **`(.5,.5,.5)`**. The result should look like *Figure 5.20*:



```
• [2062.90] Compile of BP_Item_01 successful! [in 29 ms] (/Game/ThirdPersonBP/Bluepri
```

Figure 5.20: Actor with Box Collision and Cone static mesh. The Box Collision is visible when player play the game

4. Switch to **Event Graph**, click on the **Box Collision** component, and then locate **EVENTS** drop-down list. In this list, the engine already provides some blueprint events for developers to use, so click on the plus (**+**) button for event known as **On Component Begin Overlap**, as shown in *Figure 5.21*:

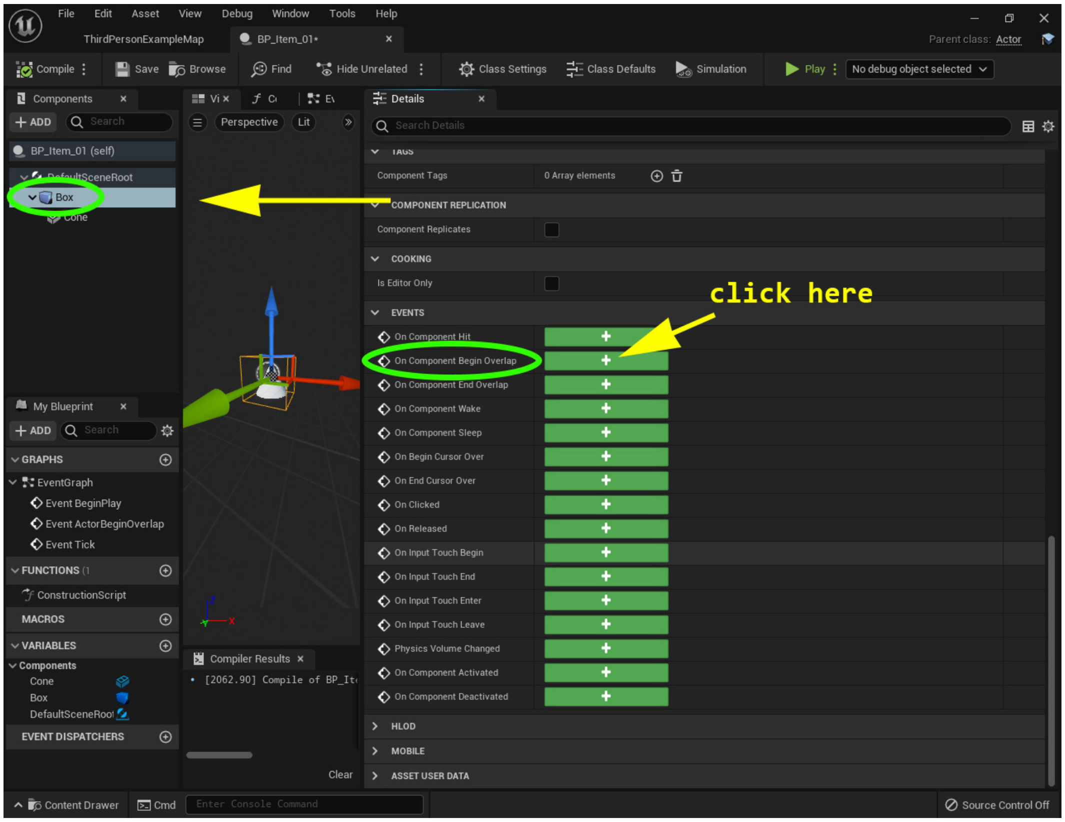


Figure 5.21: Select Box Collision and locate "On Component Being Overlap" event from event list

5. After you click on the plus button, the engine automatically navigates to **Event Graph** and adds a new event for you, as shown in *Figure 5.22*; this is just like what we made before for detect key *0* in keyboard. This event will get fired when the **Box Collision** component collides with another game object. Let's practice this by printing some messages, as we did for keyboard before:

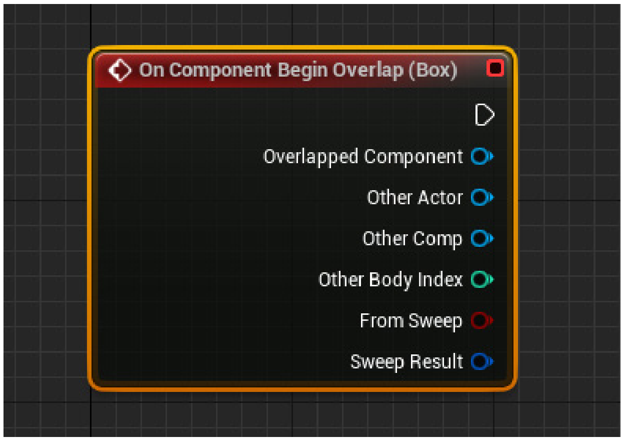
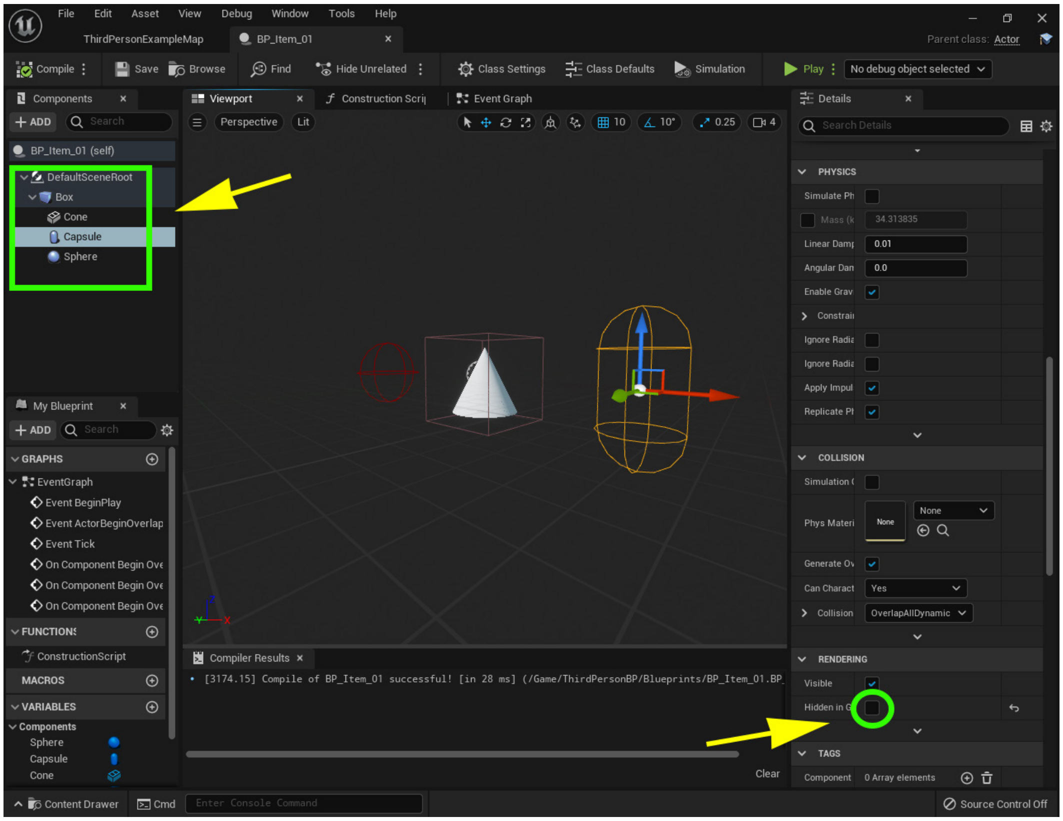


Figure 5.22: *Default collision detection blueprint event for Box Collision component*

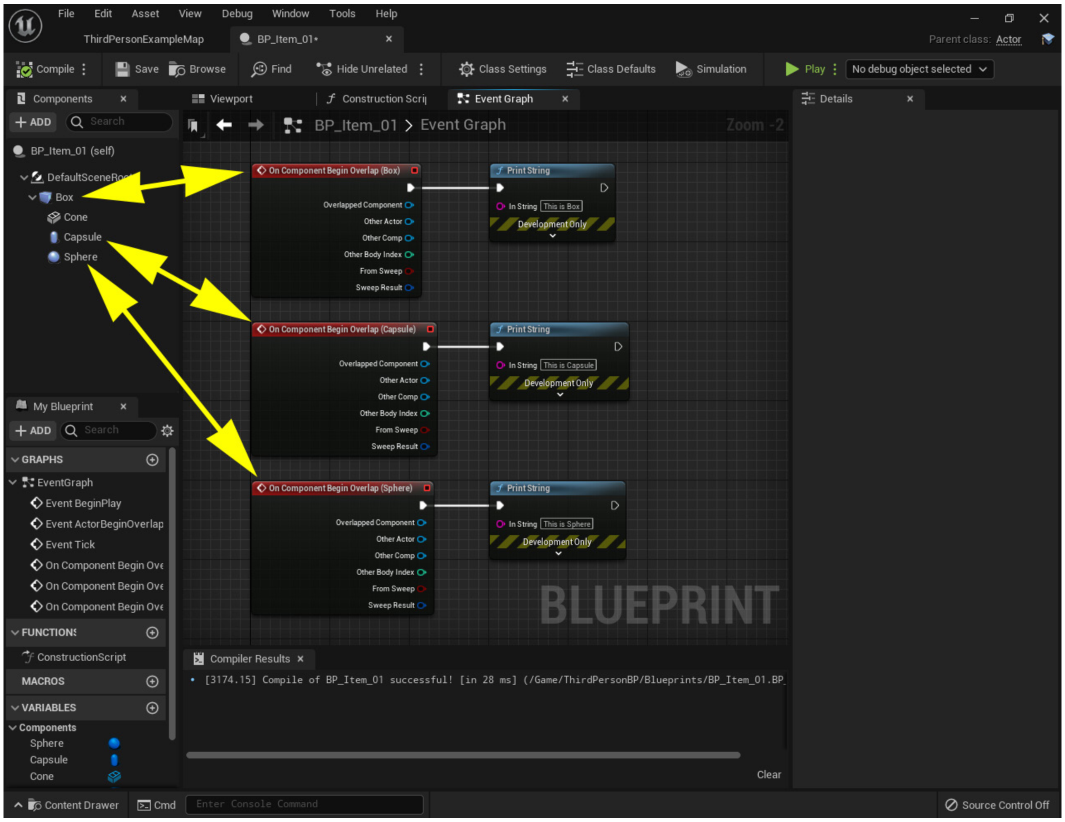
6. Add the **`Sphere Collision`** component and the **`Capsule Collision`** component to your actor and untick the **`Hidden in Game`** option for all in **`RENDERING`**, as shown in *Figure 5.23*:



```
• [3174.15] Compile of BP_Item_01 successful! [in 28 ms] (/Game/ThirdPersonBP/Blueprints/BP_Item_01.BP_
```

Figure 5.23: *Box, Capsule and Sphere collision components which are visible in game*

7. Now make collision events for new components, as we did for **`Box Collision`** component, and add the **`Print String`** command to each of them for showing a message when they collide, as shown in *Figure 5.24*:


On Component Begin Overlap (Box)
On Component Begin Overlap (Capsule)
On Component Begin Overlap (Sphere)
Print String
In String This is Box
In String This is Capsule
In String This is Sphere
Development Only
BP_Item_01 > Event Graph
Box
Cone
Capsule
Sphere
BLUEPRINT

```
• [3174.15] Compile of BP_Item_01 successful! [in 28 ms] (/Game/ThirdPersonBP/Blueprints/BP_Item_01.BP_
```

***Figure 5.24**: Add Print String to each collision event*

8. As you can see in *Figure 5.25*, after the game is running, each time the **`Third Person`** character gets into box, sphere, and capsule collision component, the engine will print out the message you developed:

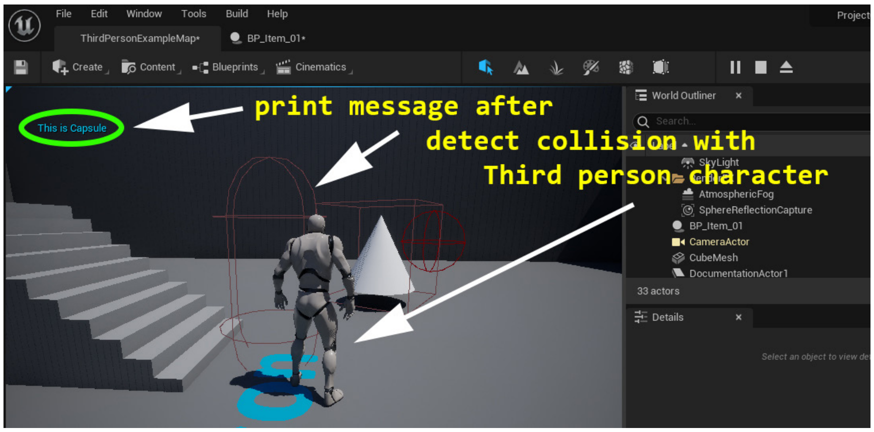


Figure 5.25: Each collision component gets active after colliding with third person character

Making simple game assets

Now, let's make a simple game asset from the blueprint actor we already have, which works with collision events. Refer to the previous chapter, replace `Cone` with a `Cube` static mesh component, and resize it to `(2.75, 0.25, 0.25)`; then, add the `Rotating Movement` component to your actor, as shown in *Figure 5.26*:

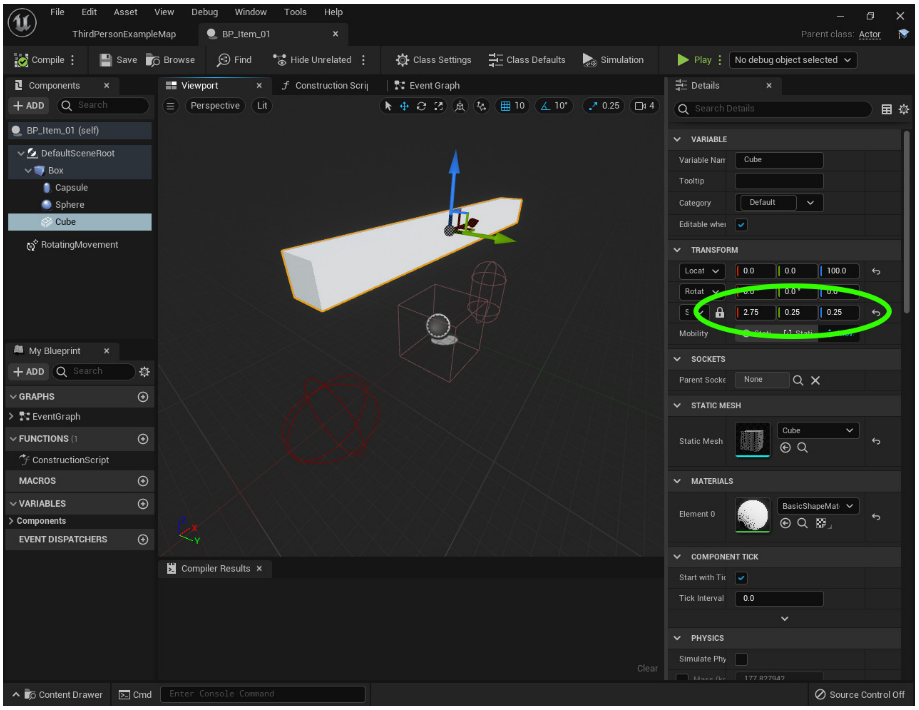


Figure 5.26: Add Cube static mesh and Rotating Movement component

If you run the game now, all components begin to rotate together, which is not what we would like to happen. Navigate to event graph and locate **`Event Begin Play`**; then, drag and drop one instance form **`Cube`** and **`Rotation Movement`** components close to it. Then, click on the **`Rotation Movement`** blue pin on the right side of its block and drag the mouse and release. Engine straightaway shows a list of *compatible* blueprint blocks for this component. Type **`Set Updated Component`** and select the **`Blueprint`** block that matches the title, as shown in *Figure 5.27*:

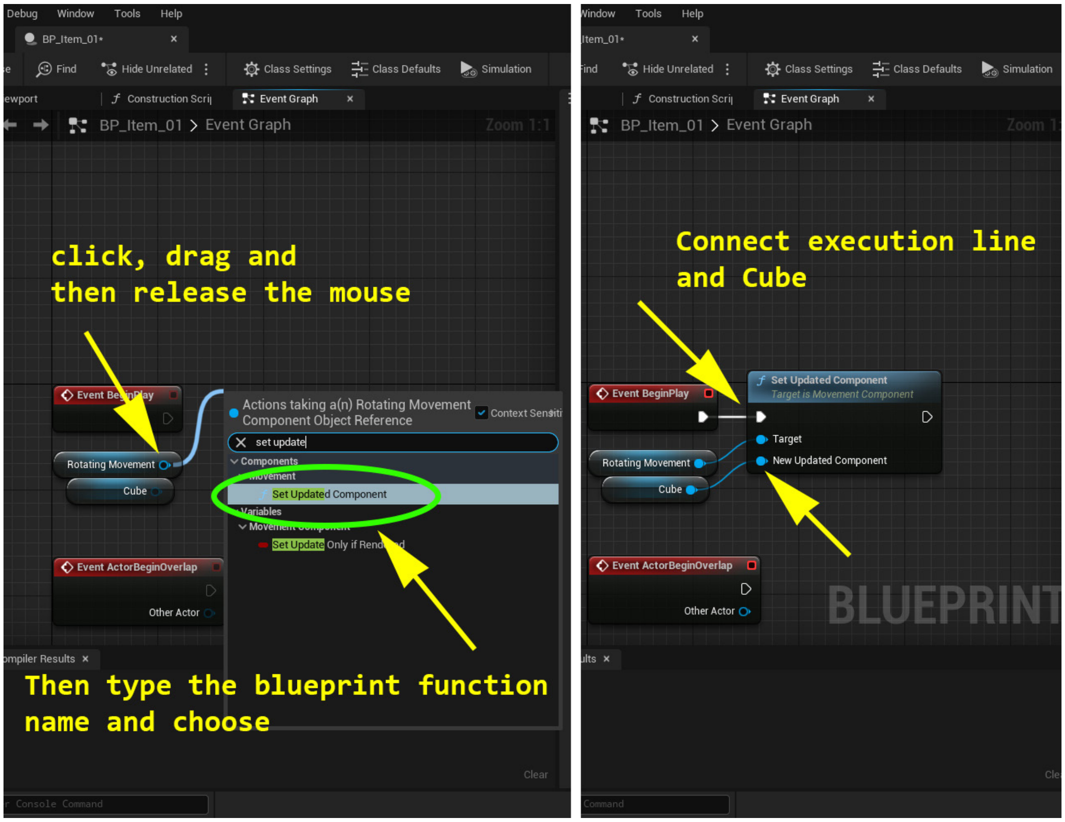


Figure 5.27: Add Set Update Component blueprint block to the code to assign Cube as target of rotation

The engine will show a new blueprint block that accepts execution lines (like **`print string`** and **`delay`**), and the **`Rotating Movement Component`** is connected to its **`Target`** pin. The **`Target`** pin represents the reference of the blueprint block that the code needs to be activated for; here, it is **`Rotating Movement Component`**. Now, connect the cube to the **`New Updated Component`** pin; this way, you assign the static mesh component that you like to rotate *only*. Now, connect this block with the execution line to **`Begin Play`** event, as shown in *Figure 5.27*. We will talk more about the **`Begin Play`** event in the next chapter, but in short, this event gets run (called) exactly after the *beginning* of the game. So, when you run the game, you ask **`Rotating Movement Component`** to assign only the **`Cube`** static mesh component for rotation at the beginning of game and then game is running, which happens after you run the code.

Now, locate collision events that you already made for the **`Capsule`** and **`Sphere`** collision components, and drag another instance of **`Rotation Movement`** in front of **`Capsule`** collision, where the **`Print String`** ends. Then, click and drag a blue curvy line form **`Rotation Movement`** and release the mouse button; the engine will automatically open a drop-down list with the **`blueprint`** command, and then type **`Deactivate`** and locate the **`Deactivate`** blueprint command in the list shown in *Figure 5.28*:

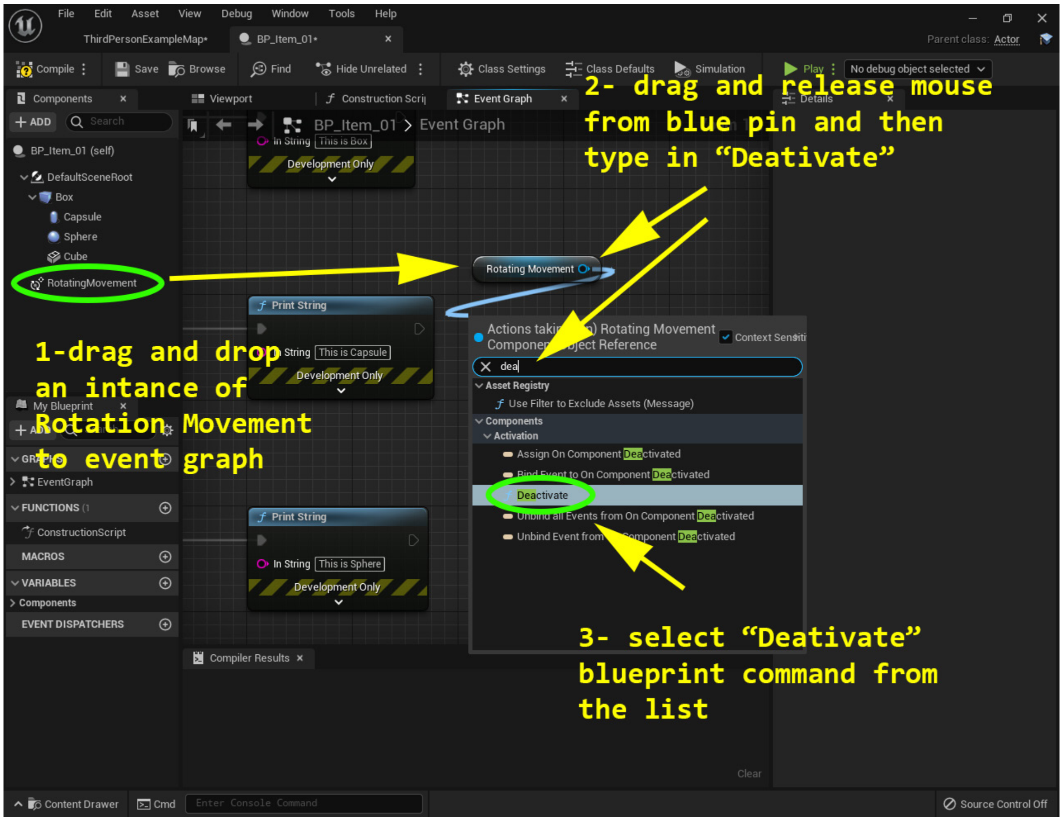

1-drag and drop an intance of Rotation Movement to event graph
2- drag and release mouse from blue pin and then type in "Deativate"
3- select "Deativate" blueprint command from the list
Components
RotatingMovement
Rotating Movement
Print String
In String This is Capsule
In String This is Sphere
Development Only
Actions taking a(n) Rotating Movement Component Object Reference
Context Sensitive
dea
Asset Registry
Use Filter to Exclude Assets (Message)
Components
Activation
Assign On Component Deactivated
Bind Event to On Component Deactivated
Deactivate
Unbind all Events from On Component Deactivated
Unbind Event from On Component Deactivated
BP_Item_01 > Event Graph

***Figure 5.28**: Locate "Deactivate" blueprint command for Rotation Movement component*

After you click on **Deactivate**, engine automatically adds **Deactivate** blueprint block to event graph. Now, connect the execution line from **print string** to this block, as shown in *Figure 5.29*:

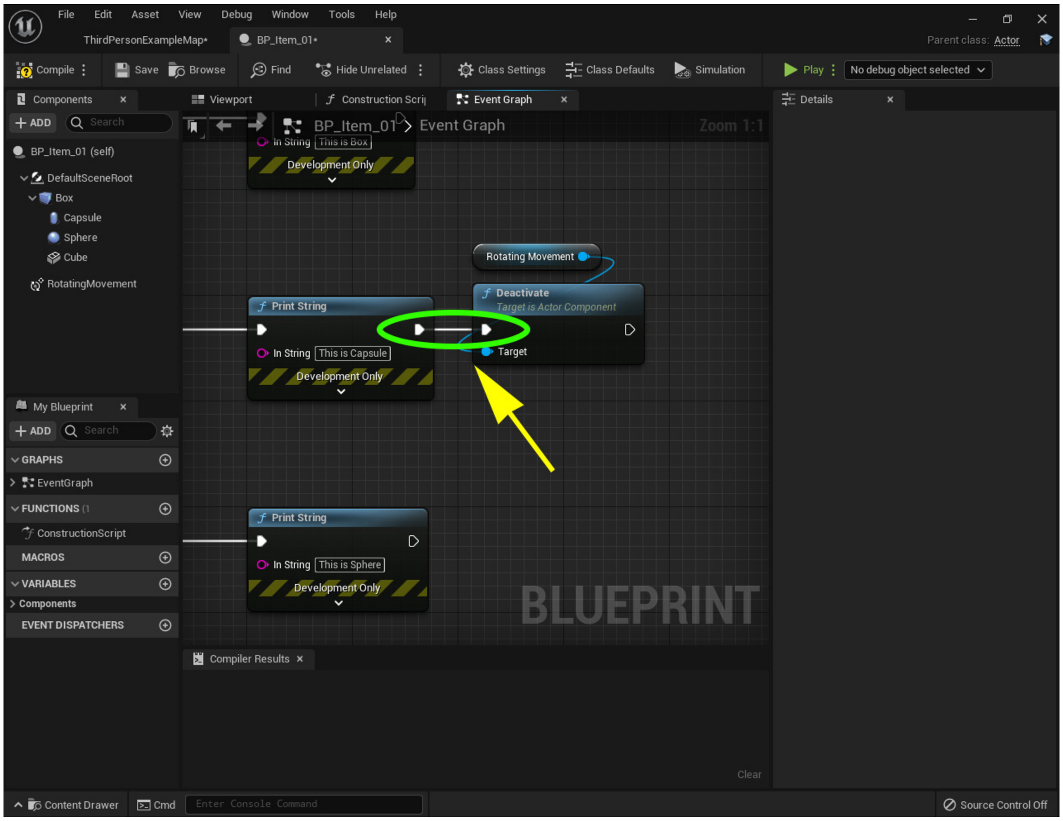


***Figure 5.29**: Connect "Deactivate" blueprint command for Rotation Movement component to the end of Capsule collision blueprint code*

When you run the game, on each collision with the **Capsule** collision component, the rotation movement gets deactivated by this code. Press *Esc* and add **Activate** blueprint code to the **Rotation Movement** component by following the same method (drag line from blueprint pin, release, type the **blueprint** command in the list, and select it). Then, connect it to the **Sphere** collision code, as shown in *Figure 5.30*:

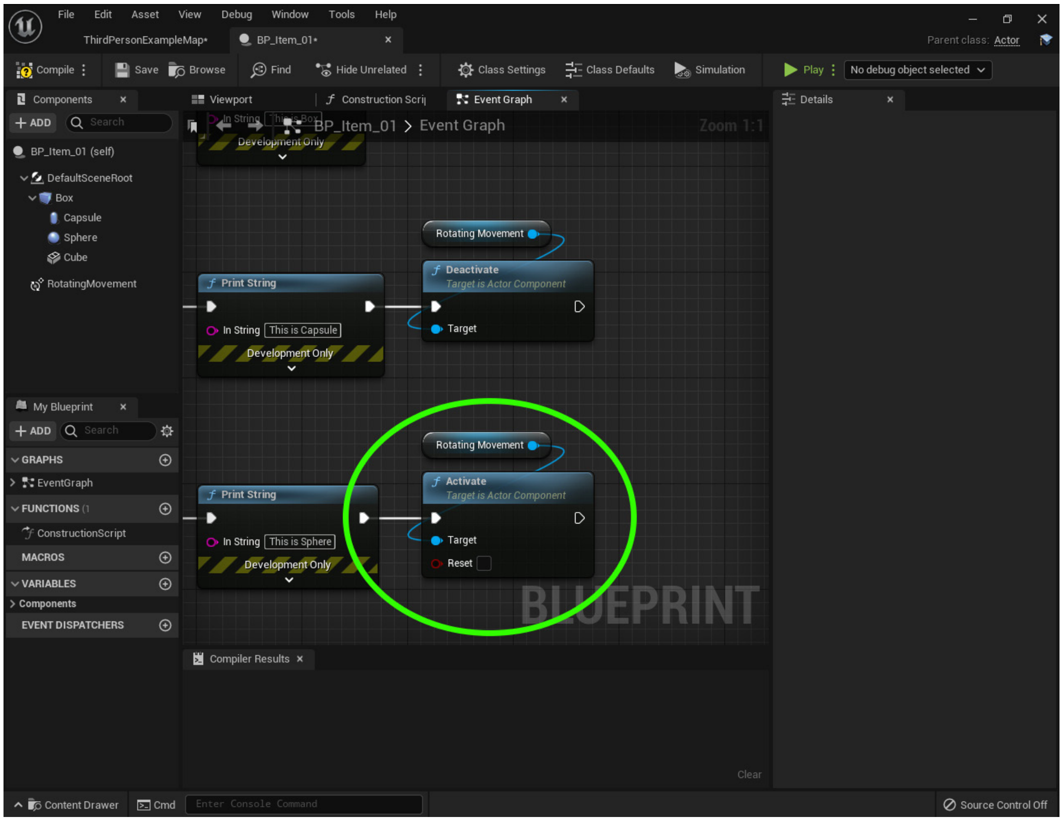


***Figure 5.30**: Connect "Activate" blueprint command for Rotation Movement component to the end of Sphere collision blueprint code*

When the game is running, each time a player collides with **`Capsule`**, the rotation will stop, and each time a player collides with **`Sphere`**, the rotation begins.

Congratulations! You've *designed and developed* your first Unreal Engine 5`s blueprint game assets by using two collision components: one static mesh component and one **`Rotation Movement`** component in an actor blueprint.

Conclusion

You have made your first Unreal Engine interactive game assets that works based on collision detection. We started from project templates, where a user can choose different types of environments to begin with, and we chose the **`Third Person`** template to use over this chapter. Then, we learned where to find target machines and user inputs in engine setting.

We developed our first blueprint code by printing messages on the screen by pressing keyboard buttons during game play, and then we made a blueprint actor with collision events and used these events to print messages. At the end, by using another blueprint component, we developed code for an interactive simple game asset.

This is how you *design and develop* blueprint actors for game. Each blueprint component has a number of blueprint commands that can mix with others and run by execution line to present game activities. We will dig into more details on how to make more complex objects and learn blueprint programming method, but for now, try to use the **`First Person`** project template, and make same game assets that are sensitive to weapon projectile, which already exists in there. All assets are ready to use in this project template, and this practice increased your skills on design and develop blueprint in Unreal Engine 5.

It's time to practice!

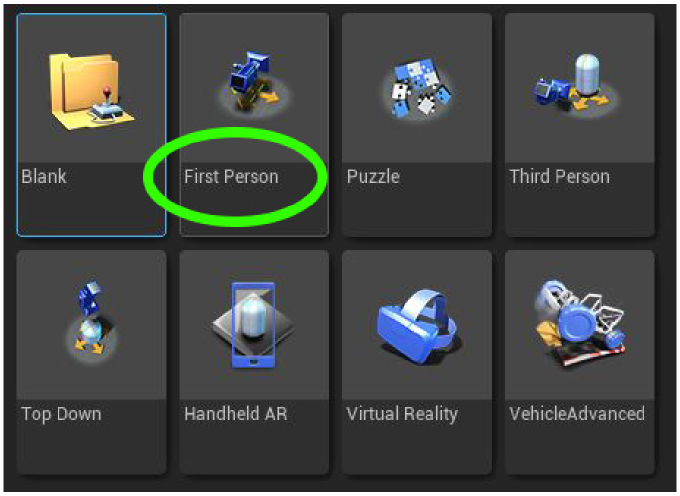


***Figure 5.31**: List of "Games" templates in Unreal Engine 5*

In the next chapter, we will learn more about game objects and how to add materials to our assets and make more proper assets for our applications.

Points to remember

- To change input keys, open **`Edit | Project Settings`** and select **`Input`**.
- The **`Action Mappings`** is like Boolean; engine will generate true or false on user interactions. On the other hand, **`Axis Mappings`** are making float values, depending on user interactions.
- The white blueprint execution line executes blueprints blocks from *left* to *right*.

Multiple choice questions

1. **When user moves to a different location by one click, this kind of movement is known as Teleport. What is the best mapping method to simulate *teleport* action by user?**
    a. Action mappings
    b. Both action mappings and axis mappings are required
    c. Axis mappings
    d. Teleport is not available in Unreal Engine

2. **When a collision collider gets triggered, which events are called inside blueprint?**
    a. Nothing is set automatically, and users need to create new custom events to support this functionality
    b. Event begin play
    c. Event on component begin & end overlap
    d. Event tick

3. **What is execution line?**
    a. The graphical white line that connects execution pin between blueprint node in event graph
    b. A special type of template
    c. An engine internal procedure to detect collision
    d. Located in project setting, it defines the input values by user interface

Answers

1. **a**
2. **c**
3. **a**

Questions

1. How can you swap mouse axis from project setting?
2. How can you swap mouse axis by blueprint code?

3. Which blueprint component can play sound, and can you switch it on and off with blueprint code?
4. Third person character blueprint has an inherited blueprint component known as `Character Movement`, which has a number of functions and properties involved with the player character on the scene. How can the user change player speed from *walk* to *run* by pressing and holding the *Shift* button on the keyboard?

Key terms

- **Blueprint Block –** Visual presentation of each blueprint function inside blueprint editor at Event Graph panel.
- **Input Setting –** The engine input setting located in Project Setting.
- **Project Setting –** An editor windows which shows most of the settings related to application and game engine.
- **Blueprint Command –** A blueprint block which performs a procedure in the application.
- **Action Mapping –** Design and address keys, on an input device like keyboard or controller, and so on, in order to interact with application with "press" and "release" methods.
- **Axis Mapping -** Design and address keys, on an input device like keyboard or controller, and so on, in order to interact with application based on a continuous range of input.
- **Event Graph (editor) –** Each blueprint editor has a panel to edit and develop blueprint code. This panel is known as "Event Graph".
- **Execution Line –**The execution line connects blueprint blocks together and execute each from "left to right" in order to perform procedures and functions of those blueprint blocks. It is a white color line which can be connected or disconnected from the execution pin of each blueprint block.

Chapter 6
Game Objects and Materials

Levels within UE5 consist of objects, (of which there are many types), and being aware of these objects and how they function is essential to programming or designing in an efficient and effective (object-oriented) manner. Many of the potential components that make up an object can be assigned materials, which change the appearance of the object. Materials can be simple, but they can become very complex.

Structure

In this chapter, we will discuss the following topics:

- Game objects
    - Classes
    - Inheritance
    - Components
    - Variables
    - Transforms
    - Components
- Events
    - Construction

    - Beginplay
    - Tick
    - Endplay
    - Custom events
    - Event overrides

- Material editor
    - Material nodes
    - Material instances

- Material data
    - Material Parameter Collections
    - Dynamic materials

Objectives

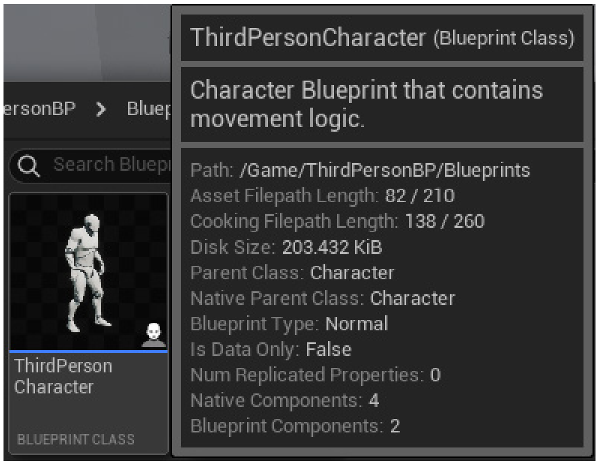
After studying this chapter, you should understand the composition of objects within Unreal Engine, object inheritance and classes, common events, and the basic usage of Materials, Material Parameters, and the Material Editor.

Game objects

Every entity in UE5 is, at its core, an object. The **`UObjectBase`** class is at the top of every class inheritance hierarchy; everything is based on it. Although the object base class is usually used sparingly as an input for functions/events/macros that can accept any item (such as the **`GetClass`** function shown in *Figure 6.2*), it is important to be aware of its existence because of its importance to the structure of the Unreal Engine.

Classes

Every object has a *class*, which is used to identify it among the many transmutations of the base object class. Parent classes (among other information) can be quickly seen by hovering over a class in the content browser, as shown in *Figure 6.1*:



ThirdPersonCharacter (Blueprint Class)

Character Blueprint that contains movement logic.

Path: /Game/ThirdPersonBP/Blueprints
Asset Filepath Length: 82 / 210
Cooking Filepath Length: 138 / 260
Disk Size: 203.432 KiB
Parent Class: Character
Native Parent Class: Character
Blueprint Type: Normal
Is Data Only: False
Num Replicated Properties: 0
Native Components: 4
Blueprint Components: 2

***Figure 6.1**: Blueprint Class in the content browser*

In blueprint, classes are useful for spawning objects, quickly tracking immediate inheritance, and checking to see if a provided object is of a specific type, as shown in *Figure 6.2*:

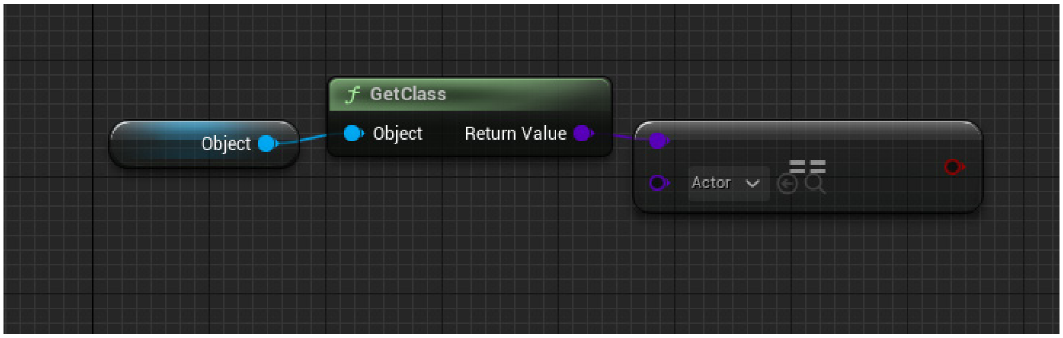
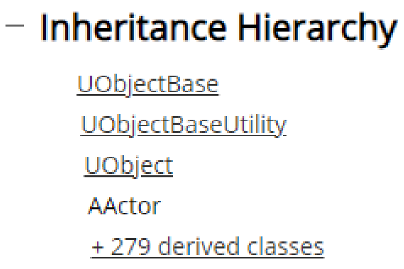


***Figure 6.2**: Checking the class of a provided object*

Inheritance

The **`Object`** is the basis of every class and following the chain of inheritance of any object will always lead back to the base **`Object`** class. For example, one of the most used classes in Unreal engine is the **`Actor`** class, and like most classes, it follows an inheritance hierarchy leading back to **`Object`** (**`UObjectBase`**), as illustrated in *Figure 6.3*. The class immediately above a given class in its inheritance hierarchy is its *parent* class, while the class below it is a *child* of the class. A class can have many children, but only one parent.



– Inheritance Hierarchy

UObjectBase
UObjectBaseUtility
UObject
AActor
+ 279 derived classes

Figure 6.3: *"Actor" Class inheritance Hierarchy*

When an object of a certain class inherits from another class (like the **`Actor`** class inherits from the **`UObject`** class, it inherits the variables, components, functions, and events of the *parent* class (the class directly above it in the inheritance hierarchy).

The *parent* class can be determined by hovering over it in the content browser and viewing its details, as is done in *Figure 6.1*, or within the blueprint editor, where the parent class is listed in the top-right corner, as shown in *Figure 6.4* (selecting the underlined parent class will open it):

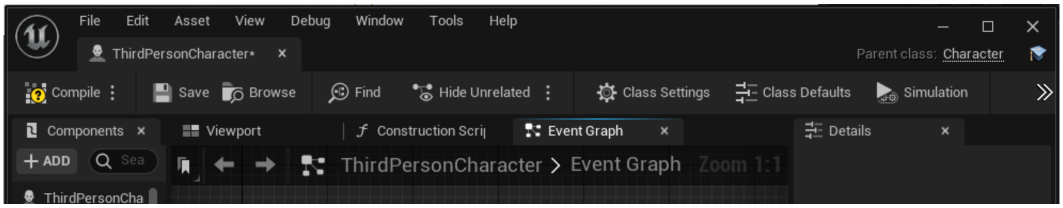
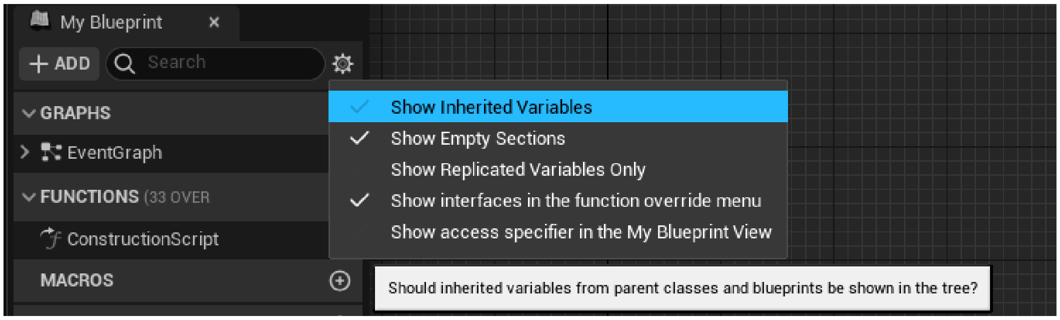


Figure 6.4: *Parent Class in the blueprint editor*

Variables

Variables are data stored in an object that contains either a reference or a value. There are many types of variables, as covered in *Chapter 2, Math for Game Design*. Child classes inherit all the variables of the parent class, although they are hidden in the blueprint editor by default; to make them visible, click on the **`gear`** icon in the **`My Blueprint`** panel and enable **`Show Inherited Variables`**, as shown in *Figure 6.5*:



✓ Show Inherited Variables
✓ Show Empty Sections
Show Replicated Variables Only
✓ Show interfaces in the function override menu
Show access specifier in the My Blueprint View

Should inherited variables from parent classes and blueprints be shown in the tree?

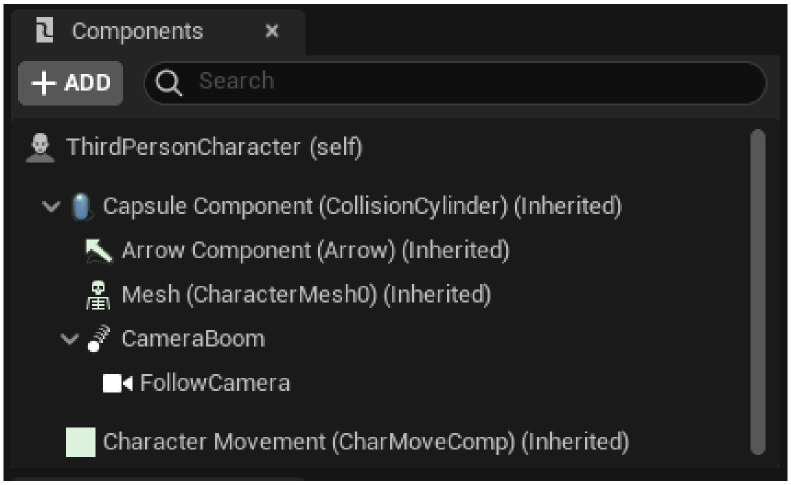
***Figure 6.5**: Showing Inherited Variables*

Components

Components are the parts that make up a game object by adding visual presentation, and/or functionalities to it. The types of components are as follows:

- **Primitive components**: Provide a visual form for the object, have a transform, and can have a collision (For interacting physically with other objects).
- **Scene components**: Provide a transform and a means of attaching other components neatly.
- **Actor components**: Give an object additional behavior or provide data management. Actor components can contain events and functions, which can be used within the object they are attached to.

The component panel of the blueprint editor shows all the components in a class; this is shown in *Figure 6.6*:

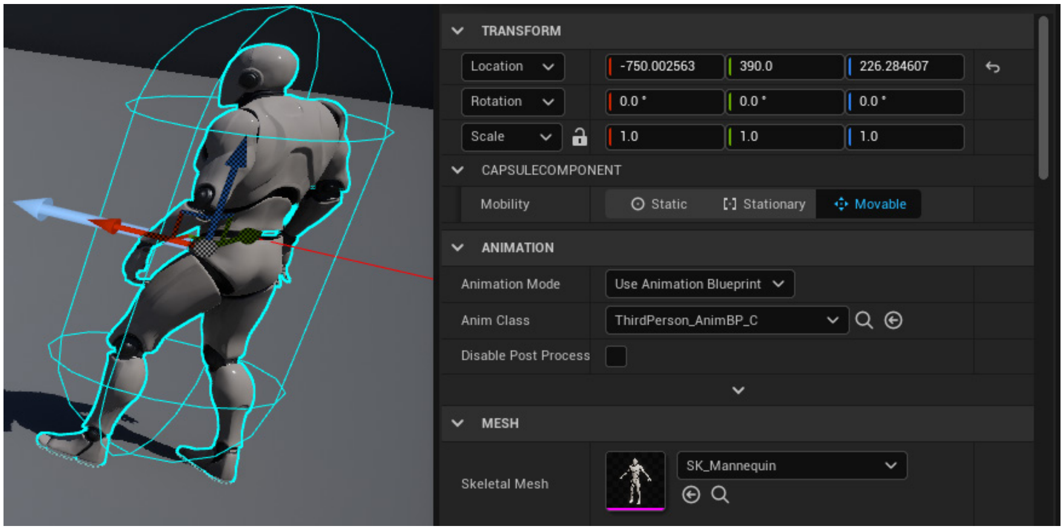


ThirdPersonCharacter (self)
Capsule Component (CollisionCylinder) (Inherited)
Arrow Component (Arrow) (Inherited)
Mesh (CharacterMesh0) (Inherited)
CameraBoom
FollowCamera
Character Movement (CharMoveComp) (Inherited)

***Figure 6.6**: Component Panel*

Transforms

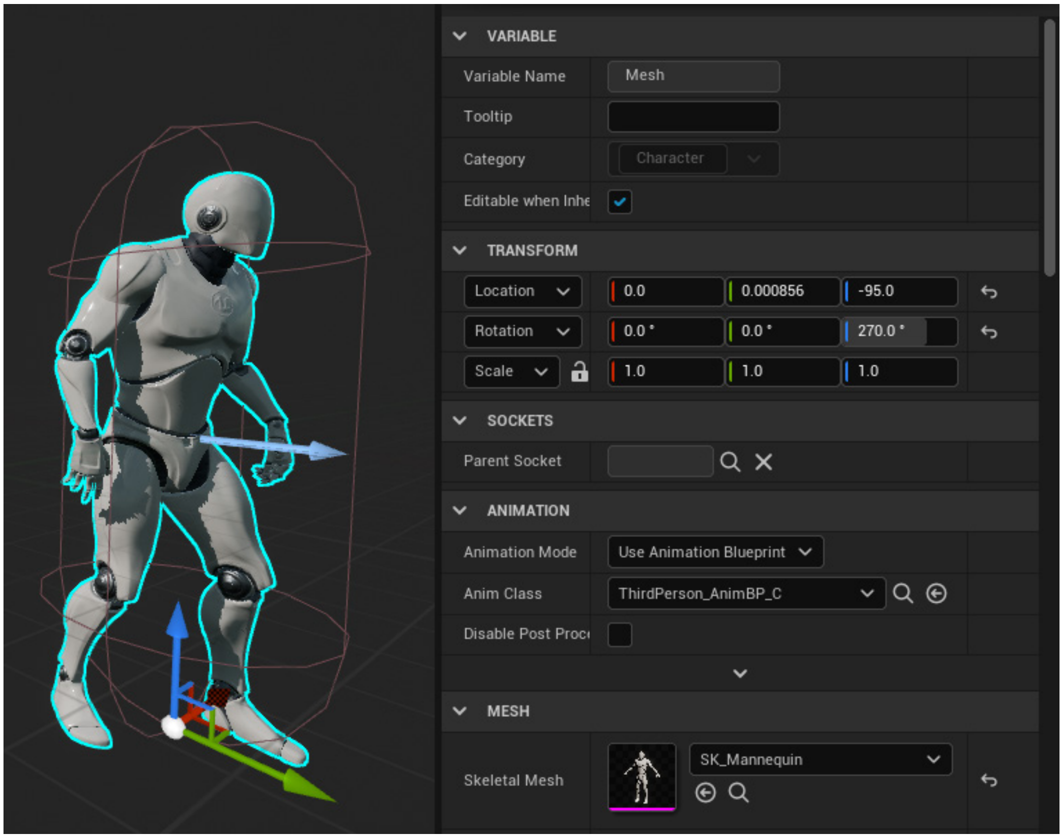
Transforms are the combination of **`Location`**, **`Rotation`**, and **`Scale`**. The transform of an object or component can be relative to either the root/origin of the world (coordinates 0,0,0), which is a *World* transform, or relative to the transform of it's parent (component or object), which is a *Relative* transform. The transform of an object can be edited from the **`TRANSFORM`** section of its details panel, as depicted in *Figure 6.7*:



TRANSFORM
Location: -750.002563 | 390.0 | 226.284607
Rotation: 0.0° | 0.0° | 0.0°
Scale: 1.0 | 1.0 | 1.0
CAPSULECOMPONENT
Mobility: Static | Stationary | [✓ Movable]
ANIMATION
Animation Mode: Use Animation Blueprint
Anim Class: ThirdPerson_AnimBP_C
Disable Post Process: ☐
MESH
Skeletal Mesh: SK_Mannequin

***Figure 6.7**: World Transform of an Object*

All component transforms are relative to the parent component or object within the editor, but there are functions to get the world or relative transform of components or objects if needed (**`Get World Transform`** and **`Get Relative Transform`**). Component transforms can be edited in the "**`details`**" panel of the component within the blueprint editor, as shown in *Figure 6.8*:



VARIABLE
Variable Name: Mesh
Tooltip:
Category: Character
Editable when Inhe: ✓

TRANSFORM
Location: 0.0 | 0.000856 | -95.0
Rotation: 0.0 ° | 0.0 ° | 270.0 °
Scale: 1.0 | 1.0 | 1.0

SOCKETS
Parent Socket:

ANIMATION
Animation Mode: Use Animation Blueprint
Anim Class: ThirdPerson_AnimBP_C
Disable Post Proc:

MESH
Skeletal Mesh: SK_Mannequin

***Figure 6.8**: Relative Transform of a component*

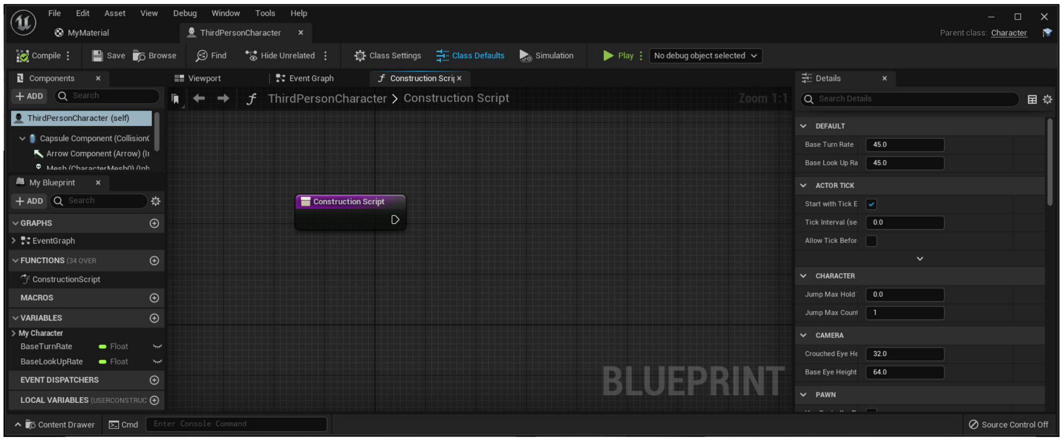
Events

Events are nodes that are executed at specific times (when the game begins, when it closes, on every tick, or when the editor is compiled, among others), by player input, or from other events or functions. They are the starting point of execution lines, which is how functionality for the game is made to occur.

For further information regarding events, visit this link: **https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Events/**

Construction script

This is a special event/function that is separate from all the others, because it only runs when compiling the editor. Construction script is used for performing advanced behavior within an object that cannot be achieved without an execution line before running the game:



DEFAULT
Base Turn Rate 45.0
Base Look Up Ra 45.0
ACTOR TICK
Start with Tick E ✓
Tick Interval (se 0.0
Allow Tick Befor
CHARACTER
Jump Max Hold 0.0
Jump Max Count 1
CAMERA
Crouched Eye He 32.0
Base Eye Height 64.0
PAWN

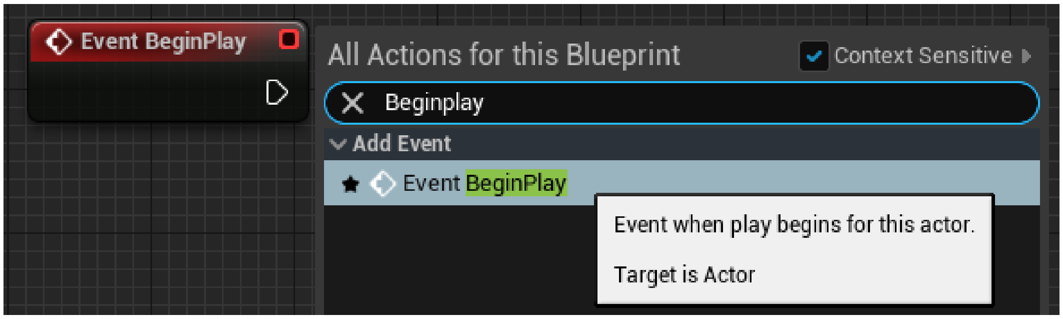
Figure 6.9: Construction Script

This can be extremely useful for generating or randomizing map elements or utilizing public variables to modify the object. There are restrictions to the construction script however, such as nodes that occur over time being unusable.

Beginplay

Beginplay is an event that runs as soon as an object is loaded, if it exists when the game or level is first loaded (if the object is placed in the world/level manually before pressing **Play**). In this situation, the **Beginplay** event will be executed as soon as the game is ready to play.

If the object is created afterward (the object is spawned into the world after **Play** is pressed), then the **Beginplay** event will run as soon as the object is spawned. The **Beginplay** event can be seen in *Figure 6.10*:



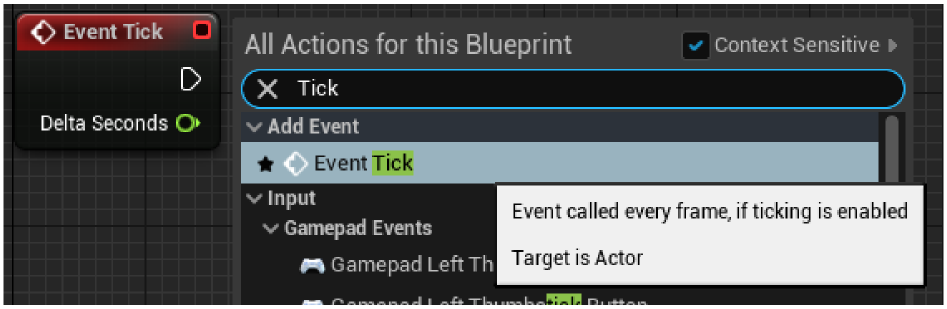
Event when play begins for this actor.

Target is Actor

Figure 6.10: BeginPlay Event (In graph and search)

Tick

The **`Tick`** event is (by default) executed once per tick of the game. A **`Tick`** in this context is when the object updates its state, which normally happens many times every second, depending on the frame rate and settings:



Event called every frame, if ticking is enabled

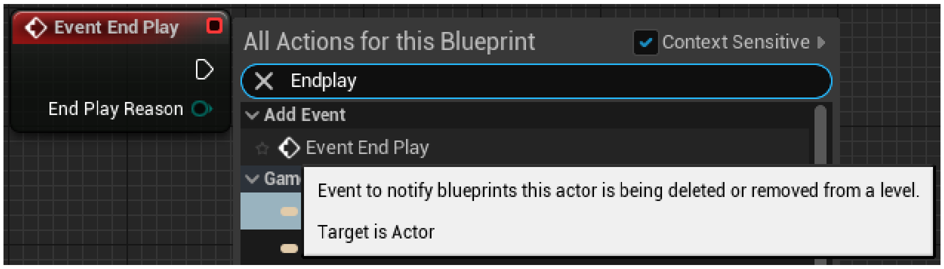
Target is Actor

Figure 6.11*: Tick Event (In Graph and Search)*

Tick has a number of settings, which allow its rate to be slowed, stopped altogether, or set to only occur in certain situations. The **`Delta Seconds`** output on the event is the last recorded amount of time between frames, which is very useful for nodes with time inputs that you want to run continuously, such as **`Print String`** nodes.

Endplay

The **`Endplay`** event runs when an object is destroyed. This event will be executed for every object that exists when the game ends; the game will end once these events are complete. It is very useful for serialization (saving player data/progress) on objects that can only be deleted when the game ends (such as the *Game Instance*) and for the final functionality for an object before it is removed from play:



Event to notify blueprints this actor is being deleted or removed from a level.

Target is Actor

Figure 6.12*: EndPlay Event (In graph and search)*

Custom Events

Custom Events are developer created events that need to be manually called. They can be called from external objects, components, and other events within the same object, although a reference to the object to run the event is required if it is to be called externally (from another object):

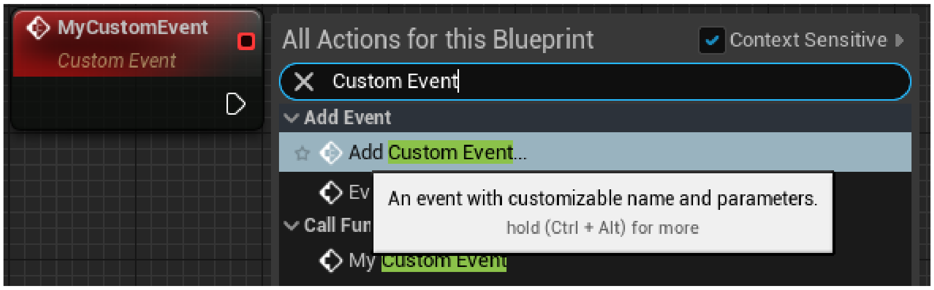


Figure 6.13*: Custom Event (In graph and search)*

Like functions, custom events can have custom inputs, which can then be utilized within the event.

Material Editor

Materials are responsible for the surface appearance of a primitive component or UI Element. Unreal Engine has an editor specifically for materials, the Material editor, which has several similarities to the Blueprint Editor and many key differences. The material editor can be seen in *Figure 6.14*:

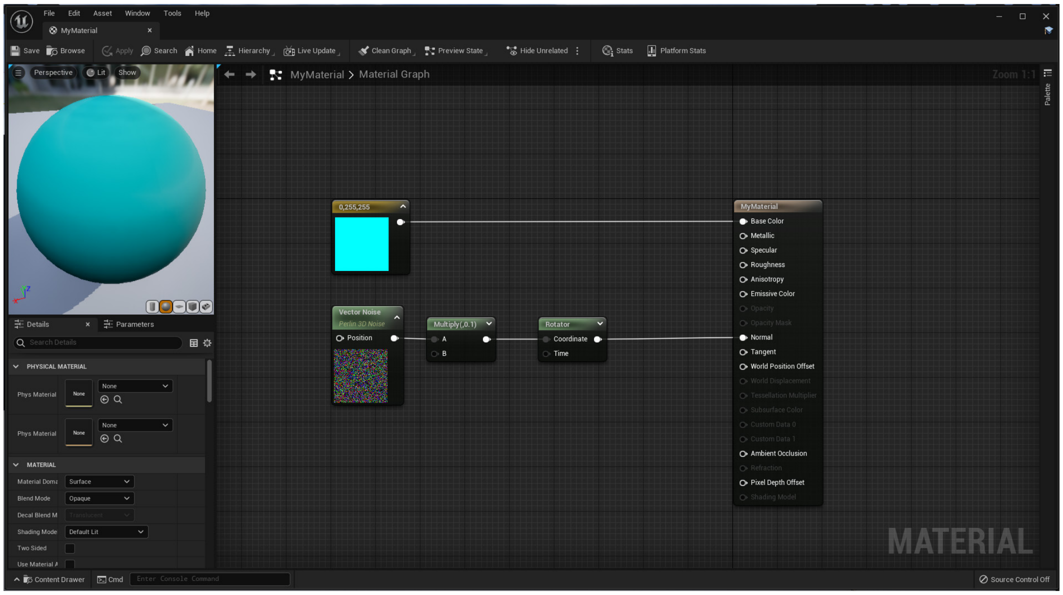


Figure 6.14*: Material Editor*

There are many uses for materials, such as decals (think posters or blood stains on walls within a level), as a *skin* for primitive components, and as light sources for example. However, in order for materials to be used properly for these purposes, they need to be set up correctly in the editor. You can change the type of the material you are editing from the **`Material Domain`** selection in the **`Details`** panel, as shown in *Figure 6.15*. This defines where the material will be used and alters the final output node to match the type of material selected:

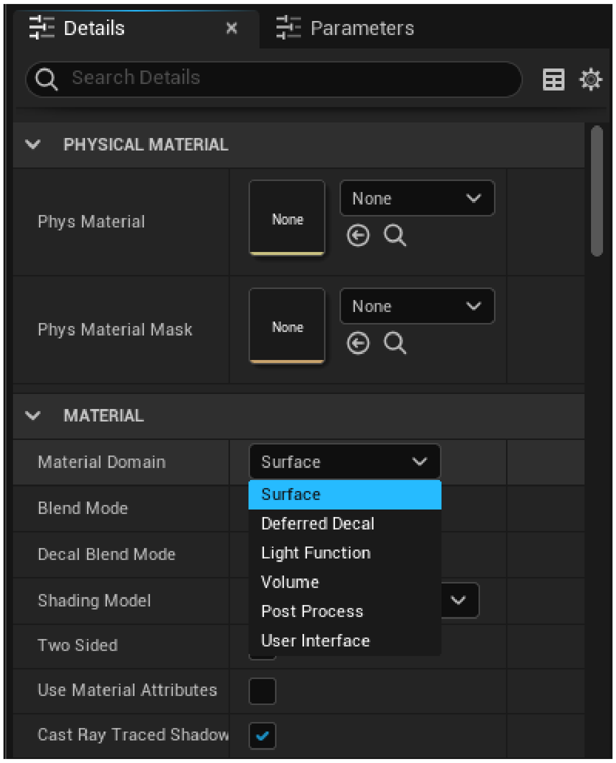


***Figure 6.15**: Material Domain Selection*

Material nodes

Like the Blueprint editor, the material editor uses nodes, but the focus is placed a lot more on the inputs/outputs of each node. This is because depending on the type of material you are editing, it can only have so many features (as shown in *Figure 6.16*), and there are no execution lines, only the order that nodes are applied to an input:

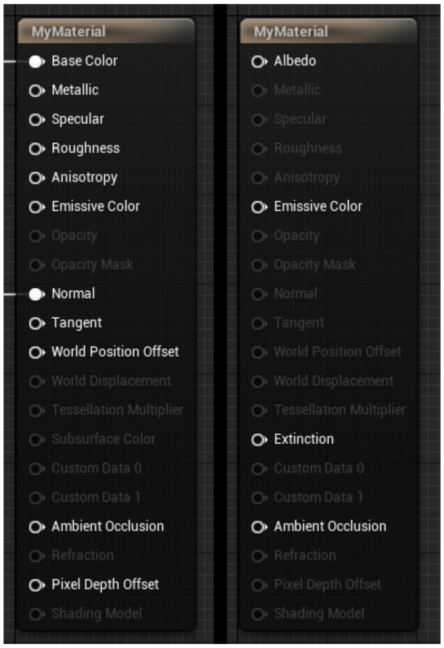


Figure 6.16: *Material outputs differ for each material Domain (Surface Material on the left, Volume material on the right)*

Most material editor nodes take a texture or vector input (consisting of three-color channels (red, green, and blue) and an **`Alpha`** Channel, which is the opacity/transparency of the material), modify it in a specific way, and output it with those changes applied, which can then be further modified or applied to the final output node. Some nodes take different inputs, like coordinates or vectors. Some basic material nodes are shown in *Figure 6.17*:

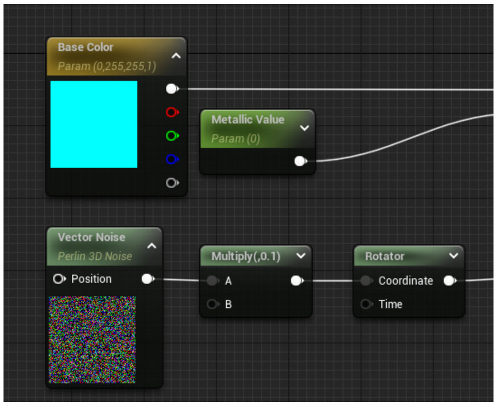


Figure 6.17: *Material Nodes*

Material instances

Material instances are children of a *master* material that can have values used by the material changed specific to each instance, allowing for several variations of the same base material that the instances are based upon. These values can be Vectors (colors) or Scalars (floats). The creation of a material instance (left) and the Material instance editor (right) is illustrated in *Figure 6.18*:

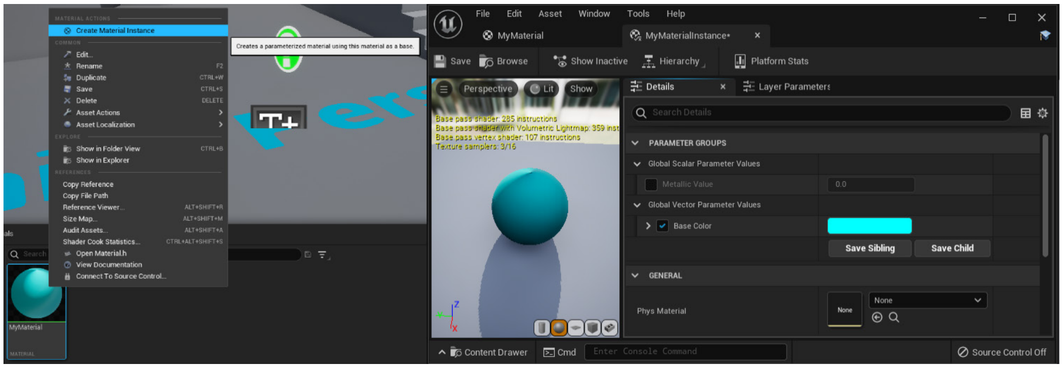


Figure 6.18: *Creating a Material Instance & the Material Instance Editor*

In order to have these parameters you can change for each instance; they need to be set up in the master material. You can see a Vector (yellow) and Scalar (green) parameter in use in the master material in *Figure 6.19*:

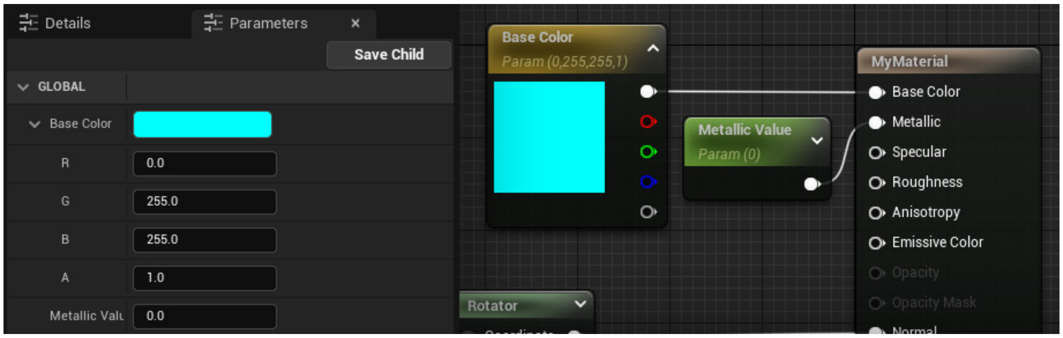


Figure 6.19: *Vector and Scalar Material Parameters*

These parameters will appear in the **`Parameters`** panel of the Material Editor (shown in *Figure 6.18*), where you can preview the effects of modifying these values and confirm that they will be accessible when modifying instances of this material.

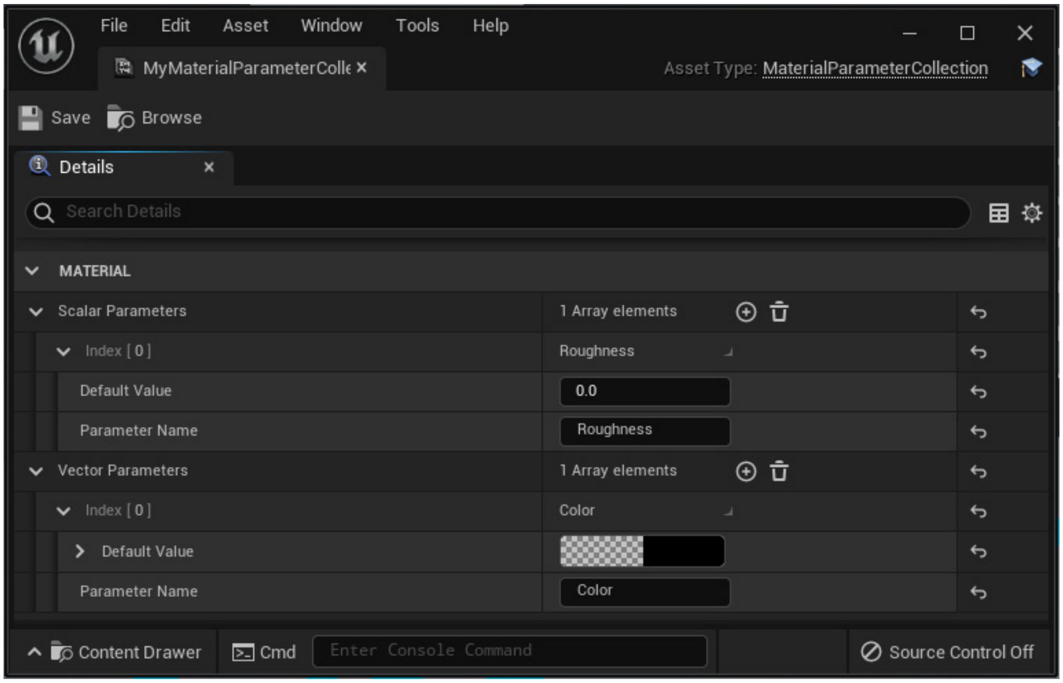
Material data

Materials can have dynamic elements, meaning that they can be altered by modifying a value that they are tied to while the game is running. This can be done by modifying a value in a Material Parameter collection, which is global and can be accessed by many materials or by modifying a value in a specific Dynamic Material.

Material Parameter Collections

Material Parameter Collections (**MPCs**) are a set of global variables that can be altered while the project is running, and these variables can be utilized by materials via special nodes within the Material Editor. MPCs are under the **`Materials & Textures`** section when creating a new asset via the content browser.

MPCs can contain **`Vector Parameters`** and **`Scalar Parameters`**. Changing the value of a parameter in an MPC will alter every material that uses that parameter accordingly. The MPC Editor can be seen in *Figure 6.20*:



MATERIAL

Scalar Parameters	1 Array elements
Index [0]	Roughness
Default Value	0.0
Parameter Name	Roughness
Vector Parameters	1 Array elements
Index [0]	Color
Default Value	▨
Parameter Name	Color

***Figure 6.20**: Material Parameter Collection One Scalar and One Vector Parameter*

To use a MPC parameter within a material, you need to use a **`Collection Parameter`** node (as illustrated in *Figure 6.21*):

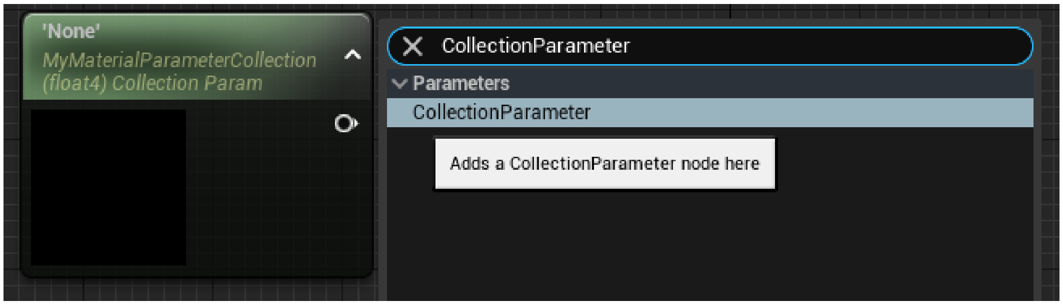

'None'
MyMaterialParameterCollection
(float4) Collection Param
CollectionParameter
Parameters
CollectionParameter
Adds a CollectionParameter node here

***Figure 6.21**: Collection Parameter Node creation*

Then, select the **`MPC`** and specific parameter you would like the node to represent in the **`Details`** panel while the node is selected, as shown in *Figure 6.22*:

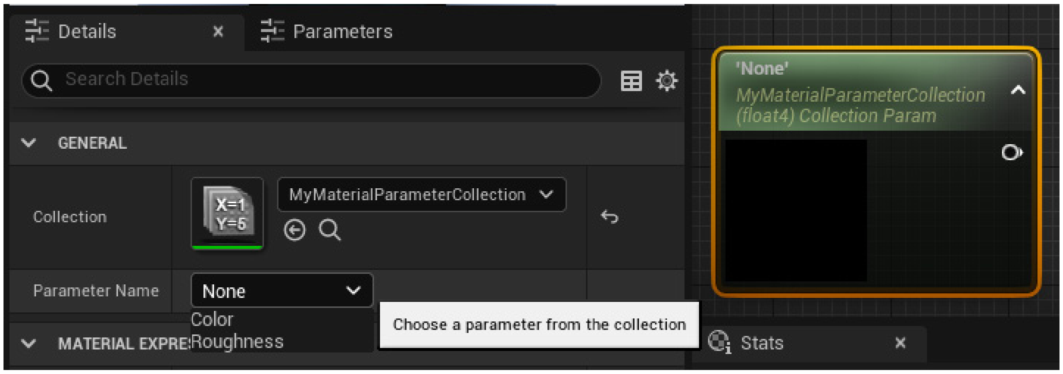

Details
Parameters
Search Details
GENERAL
Collection
MyMaterialParameterCollection
Parameter Name
None
Color
Roughness
MATERIAL EXPRE
Choose a parameter from the collection
'None'
MyMaterialParameterCollection
(float4) Collection Param
Stats

***Figure 6.22**: Collection Parameter Node setup*

Once the Node is set up and used within the material, the **`Parameter`** within the collection can be modified through blueprint via the **`Set Scalar Parameter Value`** and **`Set Vector Parameter Value`** nodes. These nodes and their inputs are shown in *Figure 6.23*:

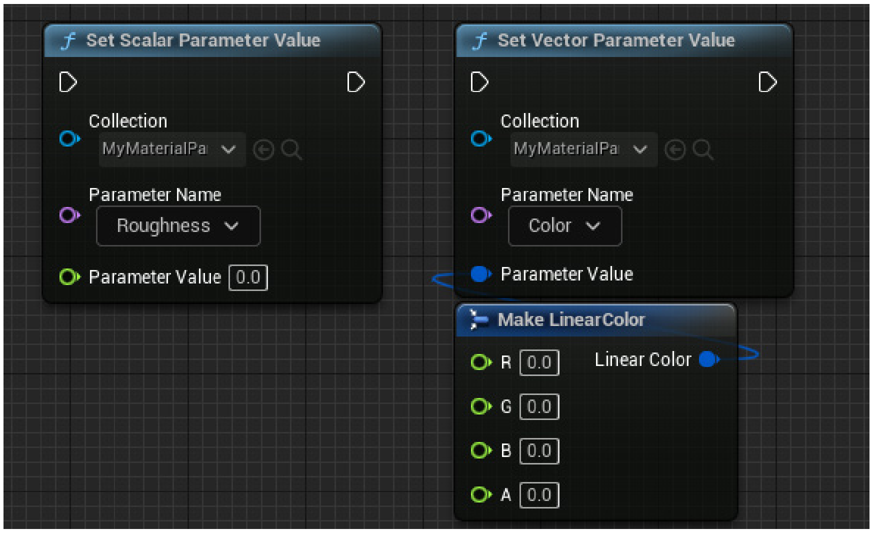


Figure 6.23: *Set Scalar and Vector Parameter value blueprint nodes*

Dynamic Materials

A Dynamic Material is one that has been initialized and is stored as a reference within a blueprint; the reference to it permits altering parameters used by the material. A Dynamic Material is set up using the **`Create Dynamic Material Instance`** node, which outputs a reference to the Dynamic Material to be stored, as shown in *Figure 6.24*:

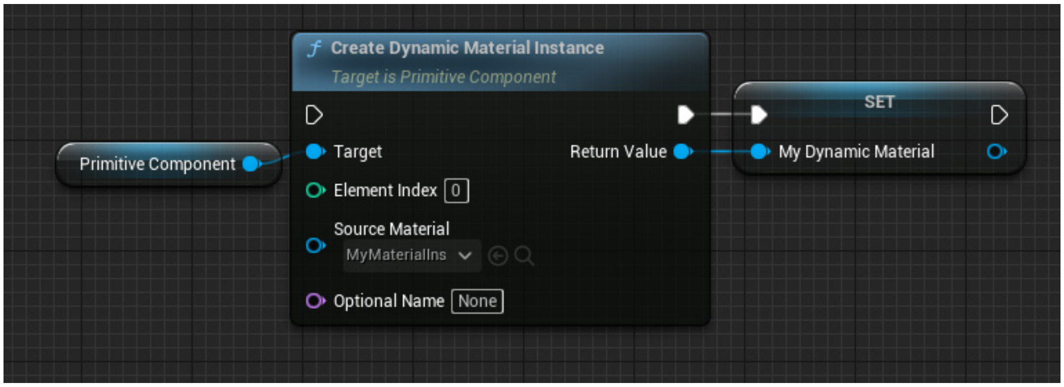


Figure 6.24: *Creating a Dynamic Material*

Once a Dynamic Material is set up, the reference to it can be used to alter parameters within the material using targeted versions of the **`Set Scalar Parameter Value`** and **`Set Vector Parameter Value`** nodes, as shown in *Figure 6.25*:

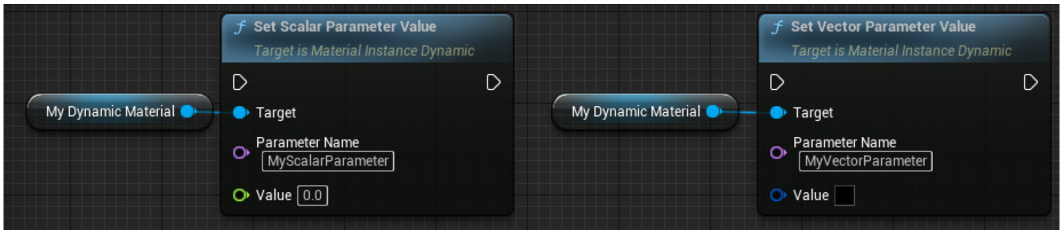


Figure 6.25*: Changing a Dynamic Material Parameter*

Dynamic Materials work in a manner similar to Material Parameter Collections, with the main difference being that Dynamic Materials' parameters are specific to each Dynamic Material Instance; the parameters are not shared globally like Material Parameter Collection parameters.

Conclusion

The structure and inheritance of Game Objects is very important to understanding how to use UE5 effectively and creating functionality for your games in the appropriate places. Inheritance is very useful for creating variations of an object that each branch out into different behaviors while maintaining (at least) elements of the original. The components that make up objects each have their uses. Specific events are used to perform functionality at the appropriate times. Materials are an incredibly useful way of getting the visuals of a game to look exactly as desired, and they are not static; they can be changed throughout the game as desired, although Materials need to be set up correctly to use this behavior. Variables are a very complex topic and were briefly covered regarding inheritance in this chapter.

The next chapter will go further into detail about variables, including the types of variables and their uses.

Points to remember

- Everything in UE5 is an object in addition to its derivation of the base object class.
- Every class in UE5 inherits the functions, events, components and variables of its parent.
- Functionality in UE5 is performed mostly through events.

- The Material Editor has many similarities to the Blueprint editor, and it also has many differences.
- Material Parameter Collection parameters are global and affect every material that uses them; Dynamic Material Parameters do not.
- The Beginplay event runs when an object is initialized.
- Tick occurs once every frame (by default).
- Custom events can have custom input variables.
- Actor components do not have valid transforms.

Multiple choice questions

1. **What types of parameters do Material Parameter Collections use?**
    a. Vector and Transform
    b. Float and Rotator
    c. Enumerator and Scalar
    d. Scalar and Vector
2. **When does the Endplay event run?**
    a. When an object initializes
    b. When an object is deleted
    c. Once every frame
    d. Every 10 seconds
3. **What do you call the class immediately above another in the inheritance hierarchy?**
    a. The Parent Class
    b. The Uncle Class
    c. The Child Class
    d. The Base Class

Answers

1. **d**
2. **b**
3. **a**

Questions

1. What can you not do within the construction script?
2. How do you set up a Dynamic Material?
3. What are the Material Domains?

Key terms

- **Blueprint Editor**: The window used to edit a blueprint class
- **Material Editor**: The window used to edit a material
- **Parameter**: A variable used within a material

Chapter 7 Simple Data

This chapter will cover the basic types of data (ones that see regular use) used in UE5 (and programming in general), explain what each type is exactly, provide additional information about each type, and also discuss their uses and benefits, along with use cases. It will also cover how they can interact with each other (pin splitting and conversions).

Structure

In this chapter, we will discuss the following topics:

- Variables in blueprint
  - Integers
    - Integer 64
  - Floats
    - Doubles
  - Booleans
  - Strings
    - Name
    - Text

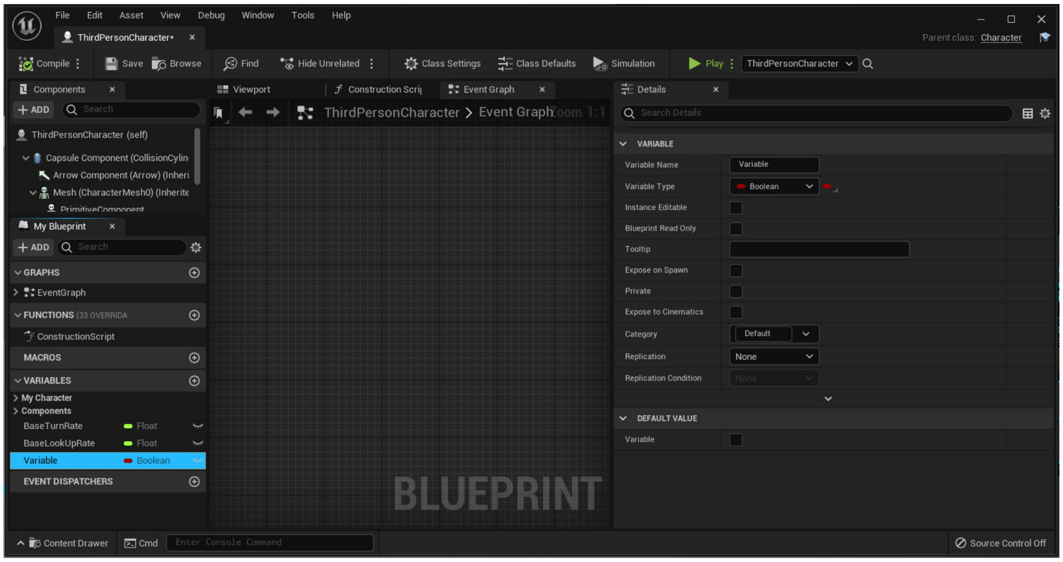
- Vectors
    - Axes
- Rotators
- Pin splitting
- Conversions

Objectives

After studying this chapter, you should know all the variables commonly used in UE5, how to set up variables within blueprints, and how these variables can be converted and broken down.

Variables in blueprint

Blueprint objects can contain variables within themselves. They are displayed/accessible in the **`Variables`** section of the **`My Blueprint`** panel in the blueprint editor (in the bottom left by default), as shown in *Figure 7.1*:



VARIABLE

Variable Name	Variable
Variable Type	Boolean
Instance Editable	☐
Blueprint Read Only	☐
Tooltip	
Expose on Spawn	☐
Private	☐
Expose to Cinematics	☐
Category	Default
Replication	None
Replication Condition	None

DEFAULT VALUE

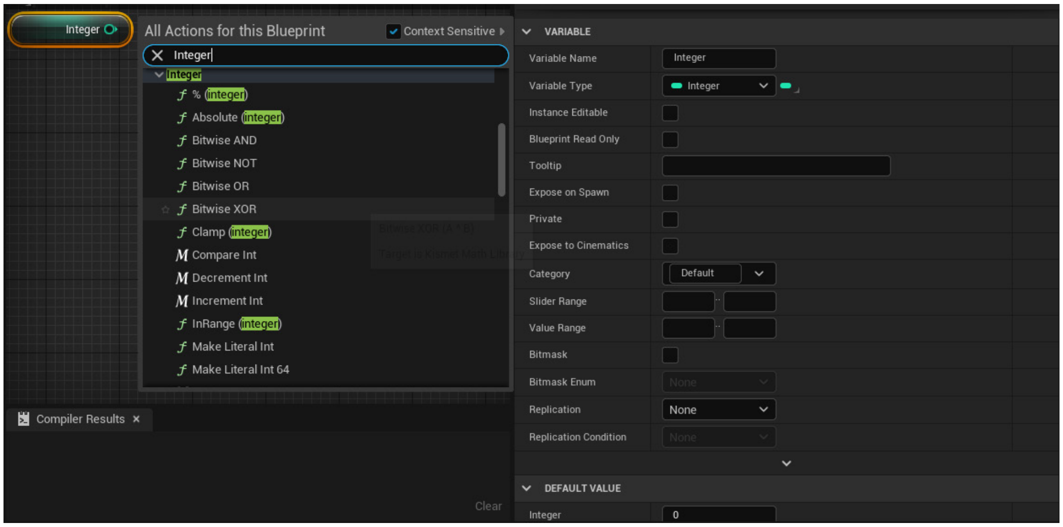
Variable	☐

Figure 7.1: Blueprint Editor with a variable selected

Once a variable is selected, it can be modified in the "**`Details`**" Panel. Things that you can change include the name of the variable, the type of variable it is, whether it is public or private, whether or not it can be changed through blueprints, and its default value.

Integers (Ints)

An integer is a variable with a positive or negative whole numerical value. Integers are useful for situations where amounts (of whole numbers) are to be denoted. There are several mathematical operators for altering a provided integer, some of which are shown in *Figure 7.2*:



Integer
- Integer
  - % (integer)
  - Absolute (integer)
  - Bitwise AND
  - Bitwise NOT
  - Bitwise OR
  - Bitwise XOR
  - Clamp (integer)
  - Compare Int
  - Decrement Int
  - Increment Int
  - InRange (integer)
  - Make Literal Int
  - Make Literal Int 64

Bitwise XOR (A ^ B)
Target is Kismet Math Library

VARIABLE

Variable Name	Integer
Variable Type	Integer
Instance Editable	☐
Blueprint Read Only	☐
Tooltip	
Expose on Spawn	☐
Private	☐
Expose to Cinematics	☐
Category	Default
Slider Range	
Value Range	
Bitmask	☐
Bitmask Enum	None
Replication	None
Replication Condition	None

DEFAULT VALUE

Integer	0

***Figure 7.2**: Integer Variable and some operators*

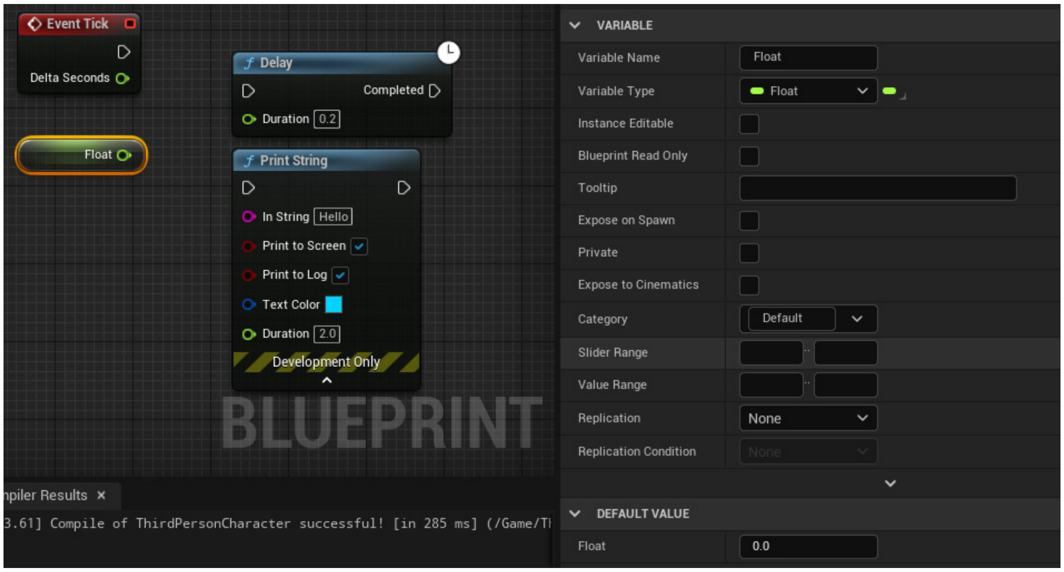
Integer 64

When utilizing integers, you may notice another type of variable called **Integer 64**, which is rarely needed but is good to be aware of. Integer 64 is the same as a standard integer, with two key differences: integer 64 takes up more memory space than a standard integer but has a much larger maximum value (both positive and negative).

You likely won't use Integer 64 variables often, but you may need to when you are dealing with incredibly high values. The additional memory cost mentioned is negligible with modern hardware, but the majority of UE5's nodes still use Integers instead of Integer 64s (where applicable), so using an integer means that a conversion is not needed to use these nodes, which keeps things neater.

Floats

A float is another numerical variable, which, like integers, can be positive or negative. However, unlike integers, floats can contain decimal values. They are very commonly used as inputs in many nodes within UE5, particularly where time is involved. *Figure 7.3* shows a float variable and two of the most common nodes that utilize the float variable type for inputs:



```
3.61] Compile of ThirdPersonCharacter successful! [in 285 ms] (/Game/Tl
```

VARIABLE

Variable Name	Float
Variable Type	Float
Instance Editable	☐
Blueprint Read Only	☐
Tooltip	
Expose on Spawn	☐
Private	☐
Expose to Cinematics	☐
Category	Default
Slider Range	
Value Range	
Replication	None
Replication Condition	None

DEFAULT VALUE

Float	0.0

Figure 7.3: *Float Variable and some nodes that use it*

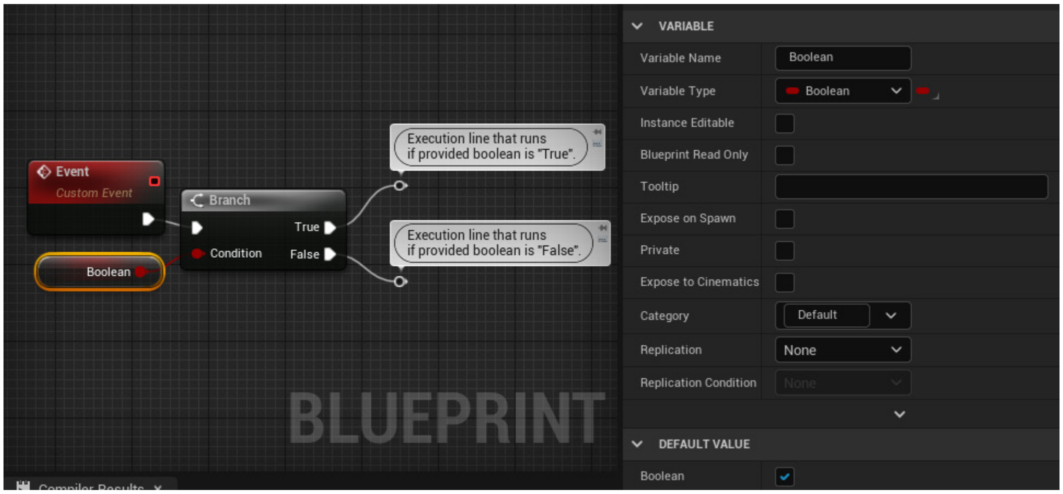
Doubles

Like integers, floats have a variable that is almost identical with a few exceptions: the **`Double`**. Floats can contain up to 7 digits, whereas a double can contain up to 15, meaning that Doubles are more accurate than Floats are, but at a slightly increased performance/memory cost.

As with Integers and Integer 64s, Floats are used much more commonly than Doubles, but Doubles may be required when dealing with incredibly small/accurate numbers. As with Integer 64s, the additional memory cost mentioned is negligible with modern hardware, but the majority of UE5's nodes still use floats instead of Doubles (where applicable), so using a float means that a conversion is not needed to use these nodes.

Booleans (Bools)

A Boolean variable can only contain one of two values: True and False. Booleans are very useful for having functionality branch into different potential outcomes based on the state of the game, and an example of this logic is shown in *Figure 7.4*:


Event
Custom Event
Branch
True
Condition
False
Boolean
Execution line that runs if provided boolean is "True".
Execution line that runs if provided boolean is "False".
BLUEPRINT
Compiler Results

VARIABLE

Variable Name	Boolean
Variable Type	Boolean
Instance Editable	☐
Blueprint Read Only	☐
Tooltip	
Expose on Spawn	☐
Private	☐
Expose to Cinematics	☐
Category	Default
Replication	None
Replication Condition	None

DEFAULT VALUE

Boolean	✓

Figure 7.4: *Bool Variable used with a branch node*

Using Boolean operators covered in *Chapter 2, Math For Game Design*, Boolean conditions can become rather complex to accommodate multiple conditions within a game, as is done in *Figure 7.5*:

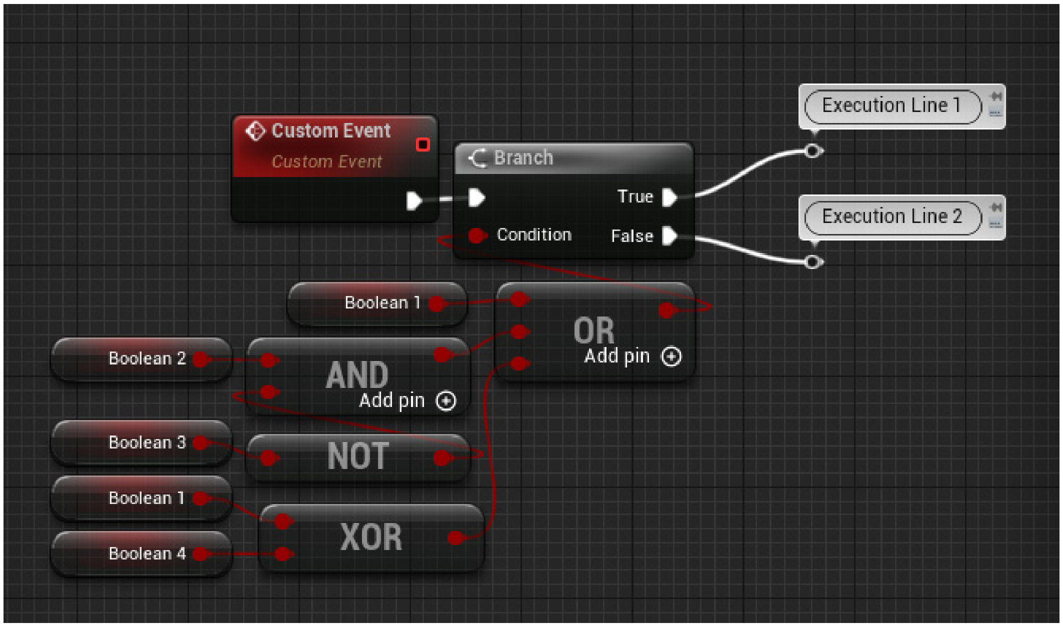
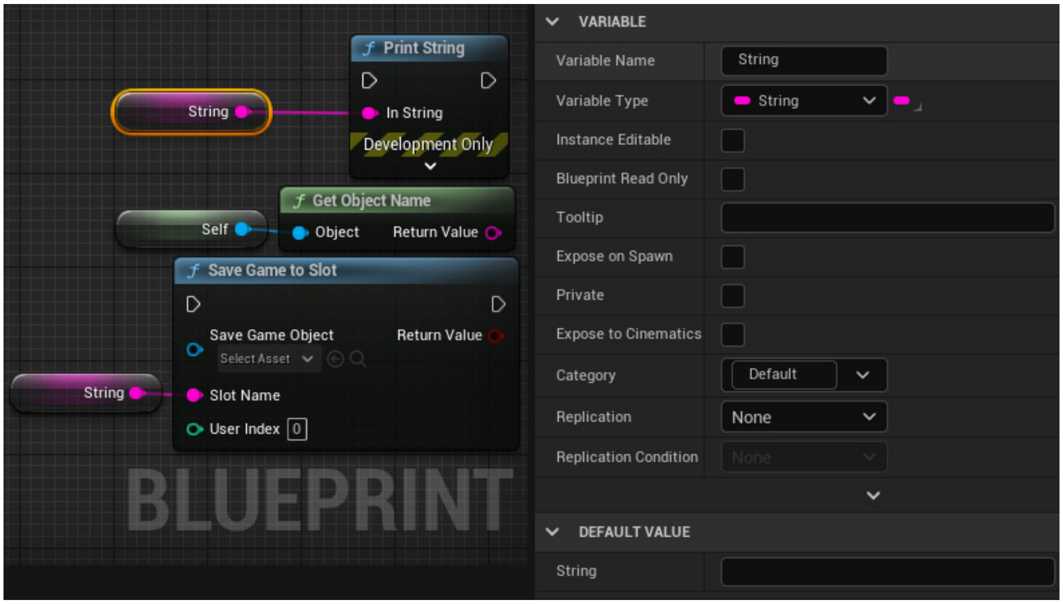


Figure 7.5: *Bool Variables and some operators in use*

Strings

A string is a variable that can contain any combination of characters. Strings are most commonly used for printing information out to an output log and/or viewports via the **`Print String`** node, getting certain data regarding provided objects, and dealing with serialization, all of which can be seen in *Figure 7.6*:



VARIABLE

Variable Name	String
Variable Type	String
Instance Editable	☐
Blueprint Read Only	☐
Tooltip	
Expose on Spawn	☐
Private	☐
Expose to Cinematics	☐
Category	Default
Replication	None
Replication Condition	None

DEFAULT VALUE

String	

***Figure 7.6**: String variable and its common uses*

While strings are the standard for text-based variables, there are two other similar variables, i.e., **`Name`** and **`Text`**, which have more specialized uses; they are mentioned below.

Name

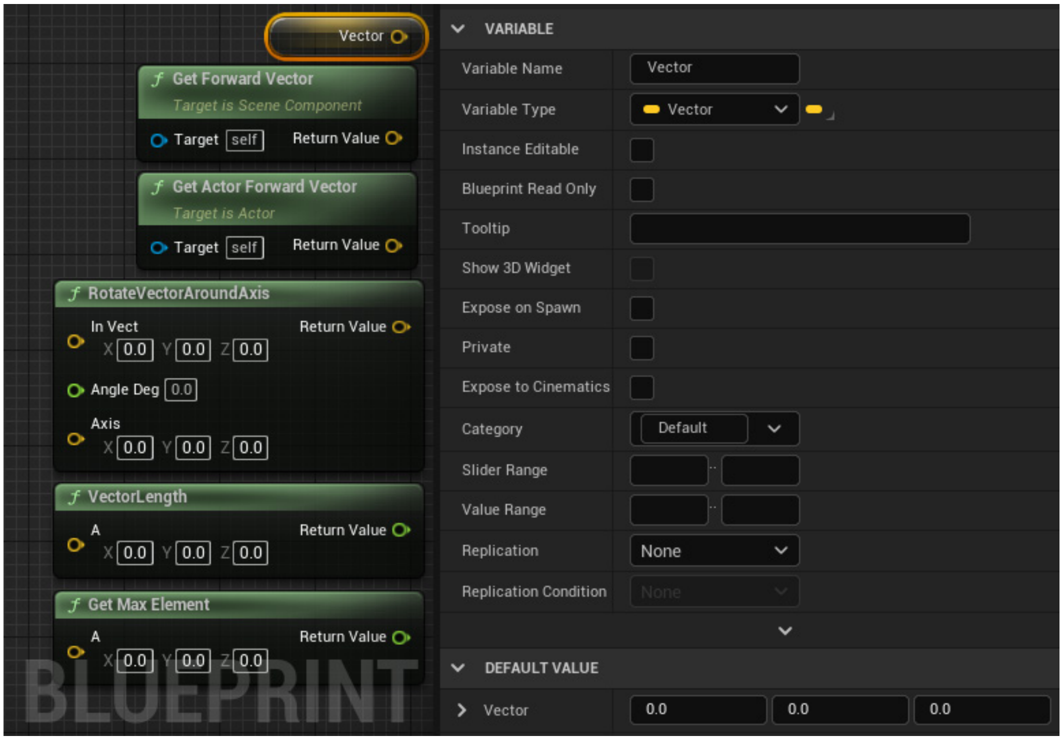
Name variables are mostly used as descriptors within assets, such as bone or socket names in a skeleton asset, but they are also used in nodes directly involved with those assets, such as the *Get all Socket names* node.

Text

Text is similar to string, but **`Text`** variables are set up for localization, meaning it is very easy to have **`Text`** variables differ based on the user's selected language.

Vector

A Vector is a variable consisting of three float values, each with an assigned axis (X, Y, or Z), which represents a location in a 3D space, but is sometimes used to represent directions, such as with the *Get Forward Vector* node:



VARIABLE

Variable Name	Vector
Variable Type	Vector
Instance Editable	☐
Blueprint Read Only	☐
Tooltip	
Show 3D Widget	☐
Expose on Spawn	☐
Private	☐
Expose to Cinematics	☐
Category	Default
Slider Range	..
Value Range	..
Replication	None
Replication Condition	None

DEFAULT VALUE

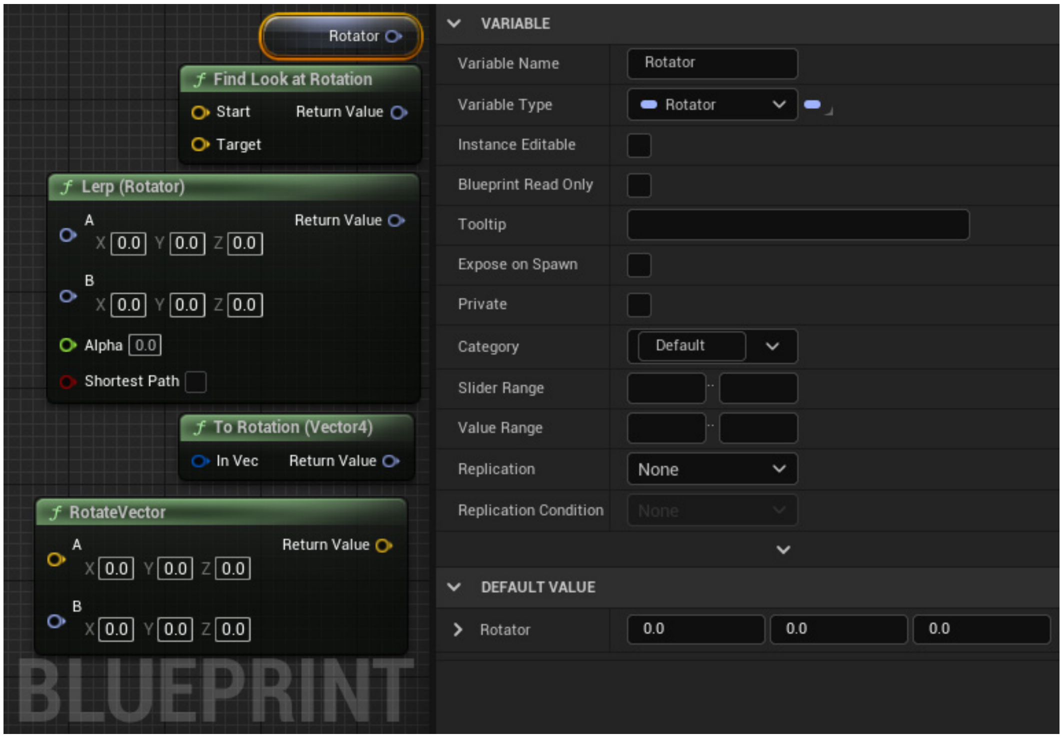
Vector	0.0	0.0	0.0

Figure 7.7: Vector variable and some nodes that use it

In addition to standard Vectors that have three dimensions, there are also two- and four-dimensional **`Vector`** variable types available for use, which may be more appropriate to use depending on the situation.

Rotator

A Rotator is a variable consisting of three float values, each with an assigned axis (X, Y, or Z), which represents a three-dimensional Rotation. The X-Axis is *Roll*, the Y-Axis is *Pitch*, and the Z-Axis is *Yaw*. Each axis has a value between -180 and 180 degrees. This can be seen in the blueprint editor in addition to some nodes utilizing rotators in *Figure 7.8*:



VARIABLE

Variable Name	Rotator
Variable Type	Rotator
Instance Editable	
Blueprint Read Only	
Tooltip	
Expose on Spawn	
Private	
Category	Default
Slider Range	
Value Range	
Replication	None
Replication Condition	None

DEFAULT VALUE

Rotator	0.0	0.0	0.0

Figure 7.8: Rotator variable and some nodes that use it

A visual representation of a rotator can be seen in *Figure 7.9*; **Blue** represents the Z-axis (Yaw), **Red** represents the X-axis (Roll), and **Green** represents the Y-axis (Pitch):

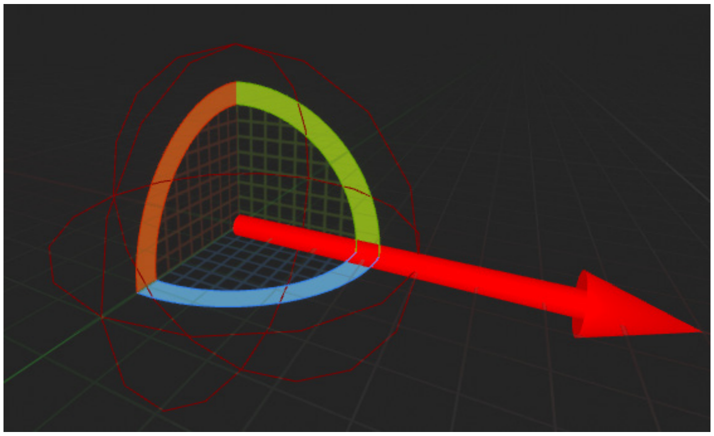


Figure 7.9: Rotator Axes (Blue= Z/Yaw, Red = X/Roll, Green = Y/Pitch)

Pin splitting

Variables that consist of multiple values can be split into the values that comprise it, allowing each one to be dealt with separately. This is done by right-clicking on the output of such a variable and selecting **`Split Struct Pin`**, as shown in *Figure 7.10*:

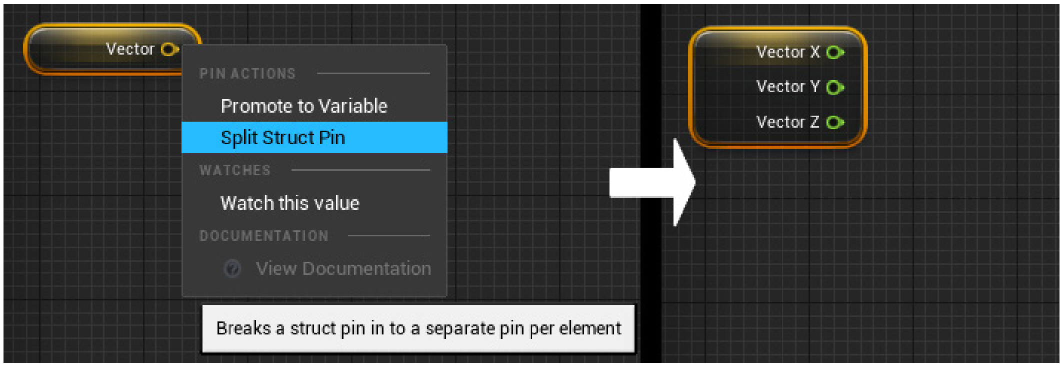


***Figure 7.10**: Split Struct Pin on a Vector*

Once a Struct pin has been split, they can be recombined by right-clicking on an element of the split Struct and selecting **`Recombine Struct Pin`**.

Conversions

Variable types that contain similar data, such as floats and ints, can be converted into each other by using a conversion node or by applying whatever mathematical processes are necessary to do so, such as rounding a float (*Removing its decimals*) to convert it to an int where necessary; both of these can be seen in *Figure 7.11*:

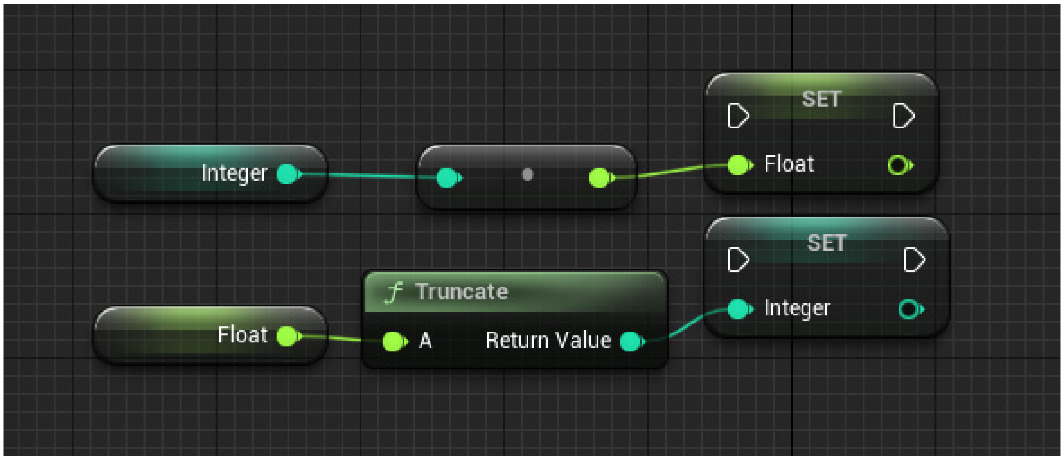


***Figure 7.11**: Converting Variables*

Conclusion

There are many types of variables available to use in Unreal Engine, and they can be made to interact with each other through math, conversions, and splitting. These variables have traits specific to them that need to be known to utilize them properly, and many have similar variable types that may be better suited to a specific situation. Programming relies on mathematical knowledge and aptitude because of the large number of different variable types and the functions required in order to use them to suit your needs.

Variables can be rather large, being composed of multiple sub-variables as in the case of vectors and rotators, or they can be simple single numeric values, as with integers. Knowing which specific variable type to use in a given situation is incredibly important for keeping a project neat and not overcomplicating any mathematics you do while programming. Mathematical errors and inefficiencies often lead to instability and poor performance, and thus, must be avoided.

The next chapter will cover interfaces and event handling, which is a method of communication between objects.

Points to remember

- Programming relies heavily on mathematics.
- There are several functions for each specific variable type.
- Integer 64s take up more memory than standard Integer but can have larger values.
- Doubles take up more memory space than floats but are more accurate.
- Many variable types have similar variable types for more specific uses.
- Some variables are composed of other variable types.
- Poor mathematical work when programming will cause issues.

Multiple choice questions

1. **How many digits can a double contain?**
    a. 7
    b. 17
    c. 14
    d. 15

2. **What variable type are the axes in a vector?**
    a. Integer
    b. Double
    c. Float
    d. Boolean

3. **What axis represents the *Yaw* of a rotator?**
    a. The X axis
    b. The Y axis
    c. The Z axis

Answers

1. **d**
2. **c**
3. **c**

Questions

1. What is a struct?
2. What is localization?
3. What might you use a four-dimensional vector for?

Key terms

- **Struct**: A collection of variables
- **Pin**: The input/output instance of a variable on a node
- **Degrees**: The unit used to measure rotations

CHAPTER 8
Interface and Event Handling

This chapter is all about understanding and using *events* in Unreal Engine 5. The *event handlining* knowledge and programming is a *skill* in game programming and directly works on the *Timing* and *Actions* that players observe when running game on their machine. A simple example of *event* is pressing space bar to jump in Tomb Raider. The *Jump* event is a code, which will run *after* a player presses the space bar and the character animation responds to it. You, (as an event programmer) need to define this code in Unreal Engine and run proper procedure after player fire it.

To learn this, we will go through the basics of event programming in engine and then learn ab advanced topic in event programming: **interface**.

At the end, we will go over the basics of *physical-based* events in the engine, which is one of the most magnificent features of Unreal Engine 5.

Structure

In this chapter, we will discuss the following topics:

- What is the event in Unreal Engine 5?
- Using event dispatchers
- Using interface
- Using physics (advanced)

Objectives

After studying this chapter, you should be able to make custom *event* in blueprint and directly call them with an instance of belonged actor class. You should also know how to use casting and call events on target objects by using *Line Trace* tool in the engine. Additionally, you should be familiar with the basics of interface implementations in blueprint by making examples. We will also go through using and apply physical impulse to spawned objects in this chapter, and we will learn how to make ice cubes out of cube objects in the scene. The last one is supported by physical material simulation in Unreal Engine 5.

To learn the topics covered in this chapter, make a new third-person template project as mentioned in the previous chapters. Then, we will expand this project step by step with coding and features involved in blueprint programming.

What is the event in Unreal Engine 5?

Let's say, *everything* is working with *events* in Unreal Engine. The event is like a *caller* or *beginner* that you can *assign and address* to a series of *Actions* in you game. Very simple example, you press mouse and it causes the gun shoot in Fortnight, or you press *Spacebar* and the player jumps in Tomb Raider. Both the click of a mouse and pressing the space bar are *firing* an *event* each inside the engine, and then the engine *Activates* the proper procedure, like shoot or jump.

Well, you may be surprised of how simple events like Collision or Begin Play and Tick can handle *everything* in engine, and you are right: the true power of a blueprint programmer is how to create new events inside the engine and address them to control the sequence of *Actions* in game during runtime. The **`Collision`** event and **`Tick`** and **`Begin Play`** are *Default* events on each blueprint, which are fired by default. The developer needs to add new events to blueprint object and design the sequence of running the scenario of game. We will practice this during the most part of this chapter by making simple interaction between blueprint objects.

To make a new event, right-click in event graph and type *custom event*; then, select the **`Custom Event`** block from the list, as shown in *Figure 8.1*:

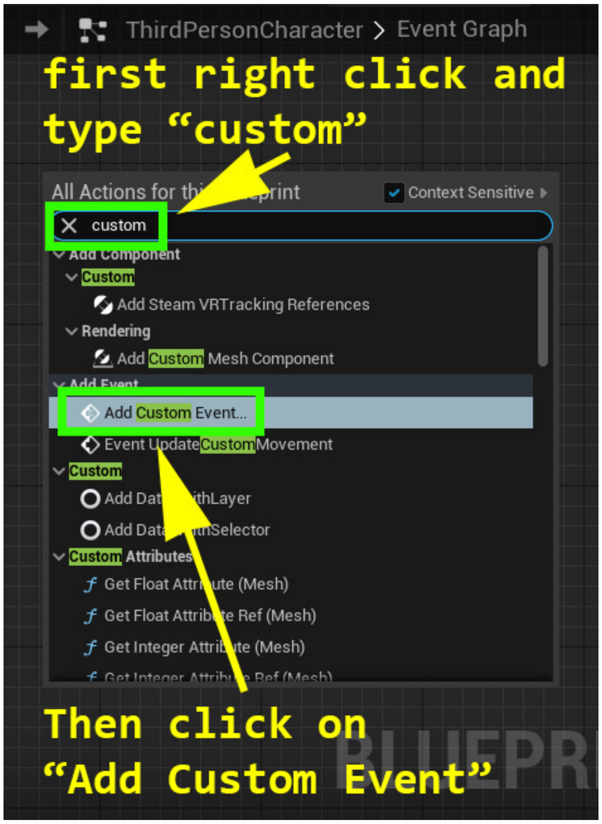

ThirdPersonCharacter > Event Graph
first right click and type "custom"
All Actions for this Blueprint
Context Sensitive
custom
Add Component
Custom
Add Steam VRTracking References
Rendering
Add Custom Mesh Component
Add Event
Add Custom Event...
Event UpdateCustomMovement
Custom
Add DataWithLayer
Add DataWithSelector
Custom Attributes
Get Float Attribute (Mesh)
Get Float Attribute Ref (Mesh)
Get Integer Attribute (Mesh)
Get Integer Attribute Ref (Mesh)
Then click on "Add Custom Event"

***Figure 8.1**: Make new custom event in Unreal Engine 5*

After you click on **`Custom Event`** in the list, the engine automatically makes a new event block and places it on event graph, as shown in *Figure 8.2*. Press the **`Compile`** button at the top:

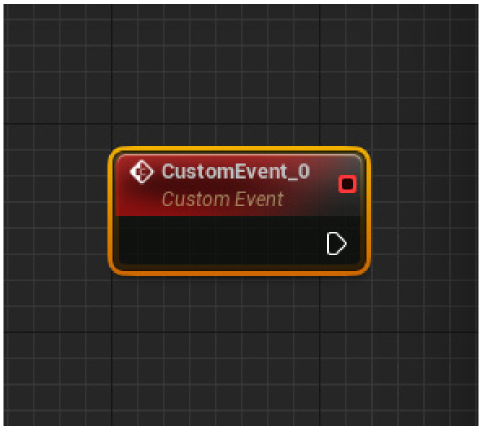
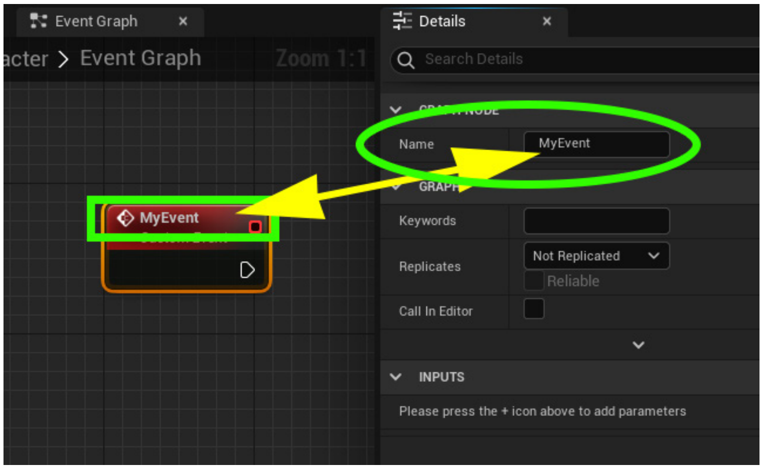


***Figure 8.2**: Default new custom event block which placed in event graph*

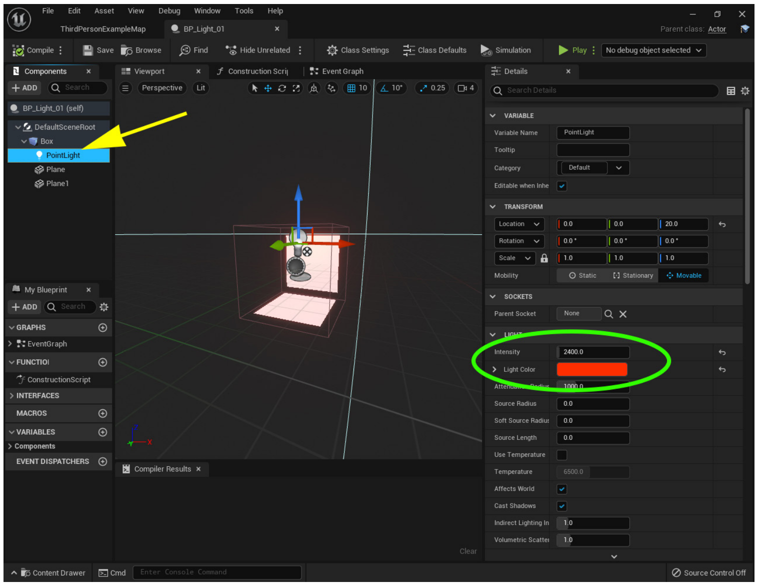
You can rename this event from the **Details** panel in the **Name** field, as shown in *Figure 8.3*:



Name: MyEvent
Keywords:
Replicates: Not Replicated
Reliable
Call In Editor
Please press the + icon above to add parameters

Figure 8.3: Rename custom event from "CustomEvent_0" to "MyEvent" by using "Name" field in Details panel

Now, let's try using events by creating two different game assets, a light source, and a switch to control it. Refer to *Chapter 5, Project Templates and User Interaction*; let's make a simple game asset as an actor blueprint that contains a light source, as illustrated in *Figure 8.4*. **BP_Light_01** is an actor blueprint and has a **PointLight** blueprint component with intensity 2400, which makes a source of red light in the scene, and it has two **Plane** static mesh components:



Variable Name: PointLight
Tooltip:
Category: Default
Editable when Inherited: ✓
Location: 0.0 0.0 20.0
Rotation: 0.0° 0.0° 0.0°
Scale: 1.0 1.0 1.0
Mobility: Static Stationary [✓ Movable]
Parent Socket: None
Intensity: 2400.0
Light Color: 🟥
Attenuation Radius: 1000.0
Source Radius: 0.0
Soft Source Radius: 0.0
Source Length: 0.0
Use Temperature:
Temperature: 6500.0
Affects World: ✓
Cast Shadows: ✓
Indirect Lighting Intensity: 1.0
Volumetric Scattering: 1.0

Figure 8.4: Point Light component properties in BP_Light_01 blueprint components

To support switching light source, as shown in *Figure 8.5*, we make two custom events **`LightSwitch_ON`** and **`LightSwitch_OFF`** in the event graph; these events change the intensity of **`PointLight`** from **`2400`** to **`0`** by using the **`Set Intensity`** blueprint command:

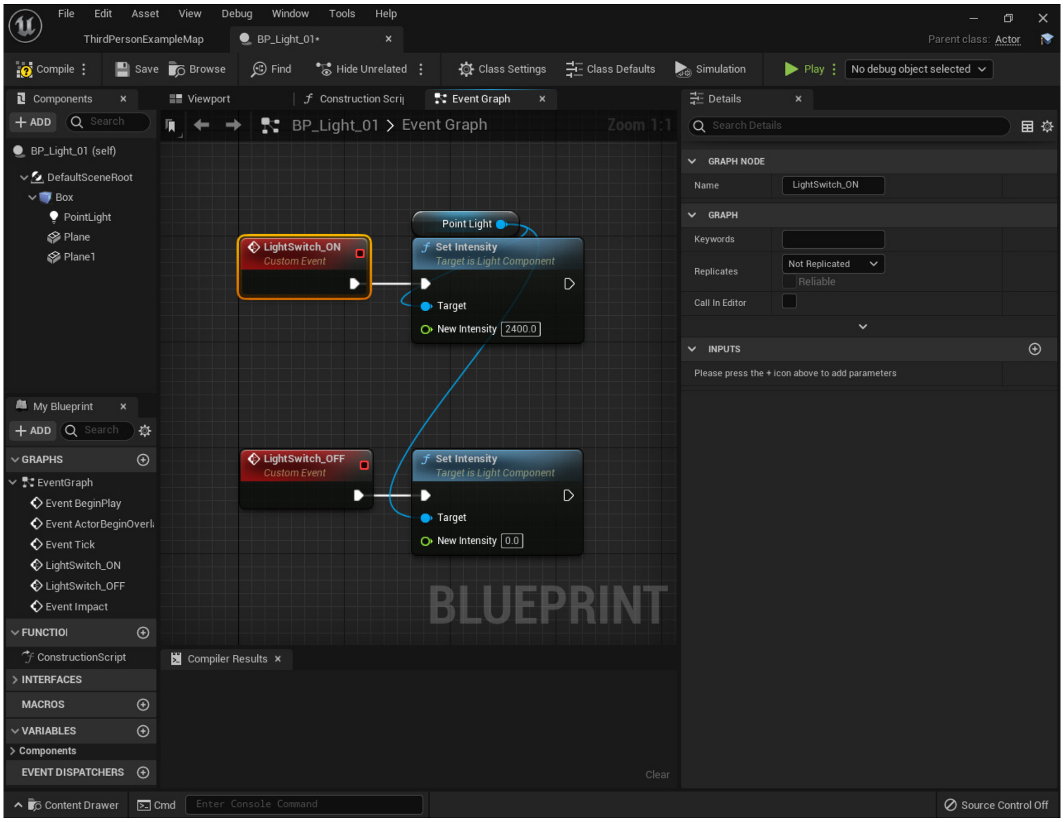
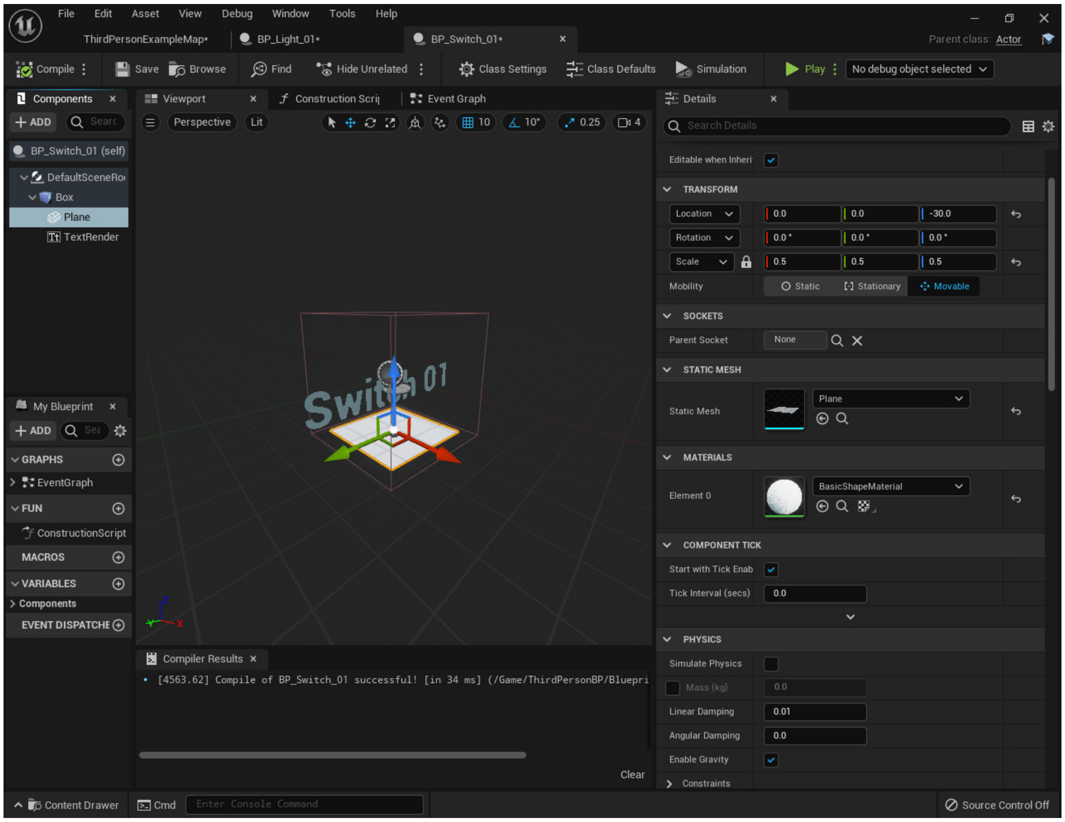


***Figure 8.5**: Using two custom events to switch point light component*

The next blueprint **`BP_Switch_01`** is a switch to control light source; as shown in *Figure 8.6*, it has a **`Text Render`** component to show its name and a **`Box Collision`** that covers a **`Plane`** static mesh component in which a user can walk in and out:



Editable when Inherited ✓

TRANSFORM
Location 0.0 0.0 -30.0
Rotation 0.0° 0.0° 0.0°
Scale 0.5 0.5 0.5
Mobility Static Stationary [✓ Movable]

SOCKETS
Parent Socket None

STATIC MESH
Static Mesh Plane

MATERIALS
Element 0 BasicShapeMaterial

COMPONENT TICK
Start with Tick Enabled ✓
Tick Interval (secs) 0.0

PHYSICS
Simulate Physics
Mass (kg) 0.0
Linear Damping 0.01
Angular Damping 0.0
Enable Gravity ✓
Constraints

```
• [4563.62] Compile of BP_Switch_01 successful! [in 34 ms] (/Game/ThirdPersonBP/Bluepri
```

Figure 8.6: *Actor "BP_Switch_01" components*

The event graph for this blueprint, as *Figure 8.7* shows, has two events from **`Box Collision`** for begin overlap and end overlap. When the user collides with box collision component in the game, each of these events will get fired. On each, we grab the **`BP_Light_01`** blueprint object from scene by using the **`Get All Actor from Class`** blueprint node, and then fire the proper event each to turn the light source on and off:

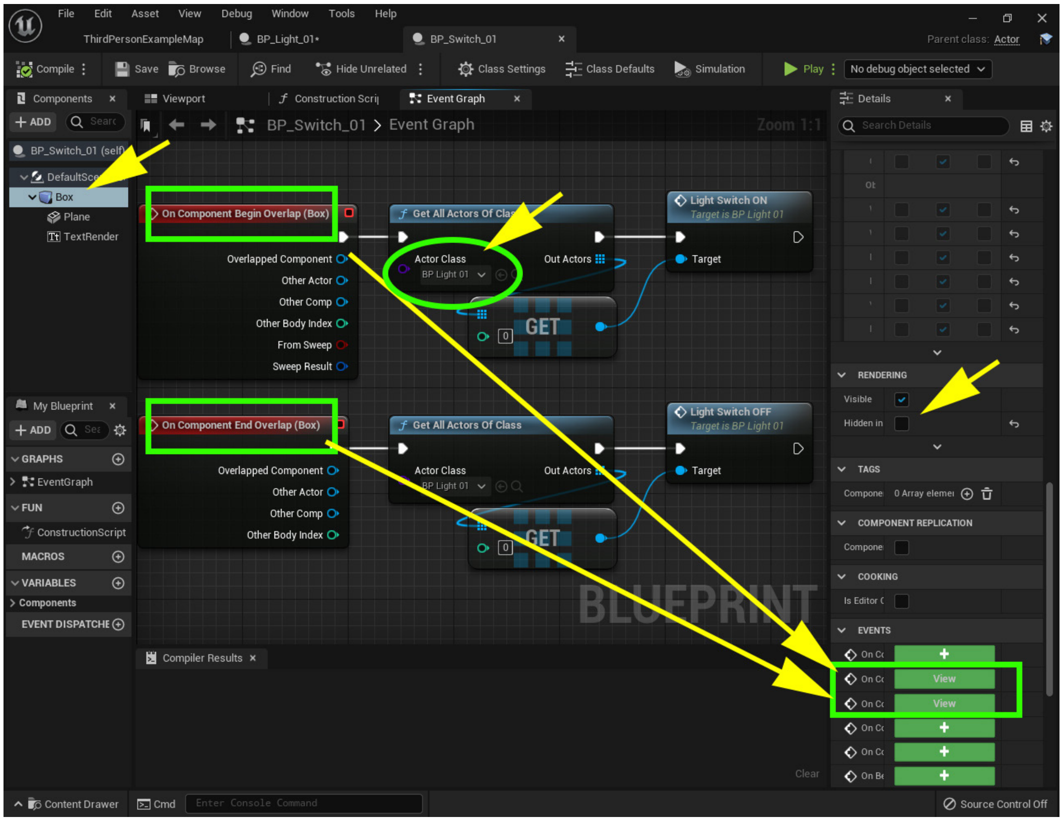


Figure 8.7: *Actor BP_Switch_01 event graph has two collision events from box collision. Each collision grabs an instance of "BP_Light_01" and activates custom events to switch the lights.*

This is the most direct way of calling events on other blueprints, but there is a much easier way to make it. Now add one instance of each actor in the scene and run the game. When you collide with switch actor, the light source actor will activate and deactivate light, as shown in *Figure 8.8*:

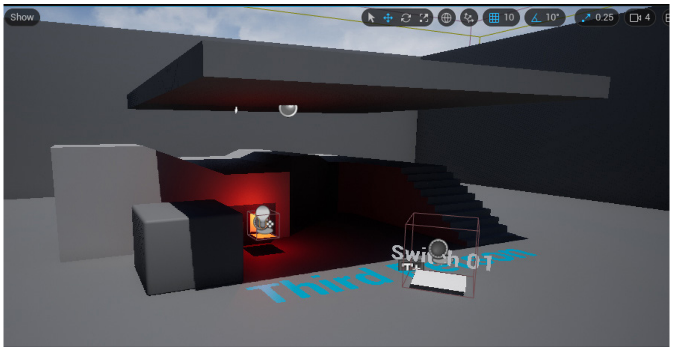
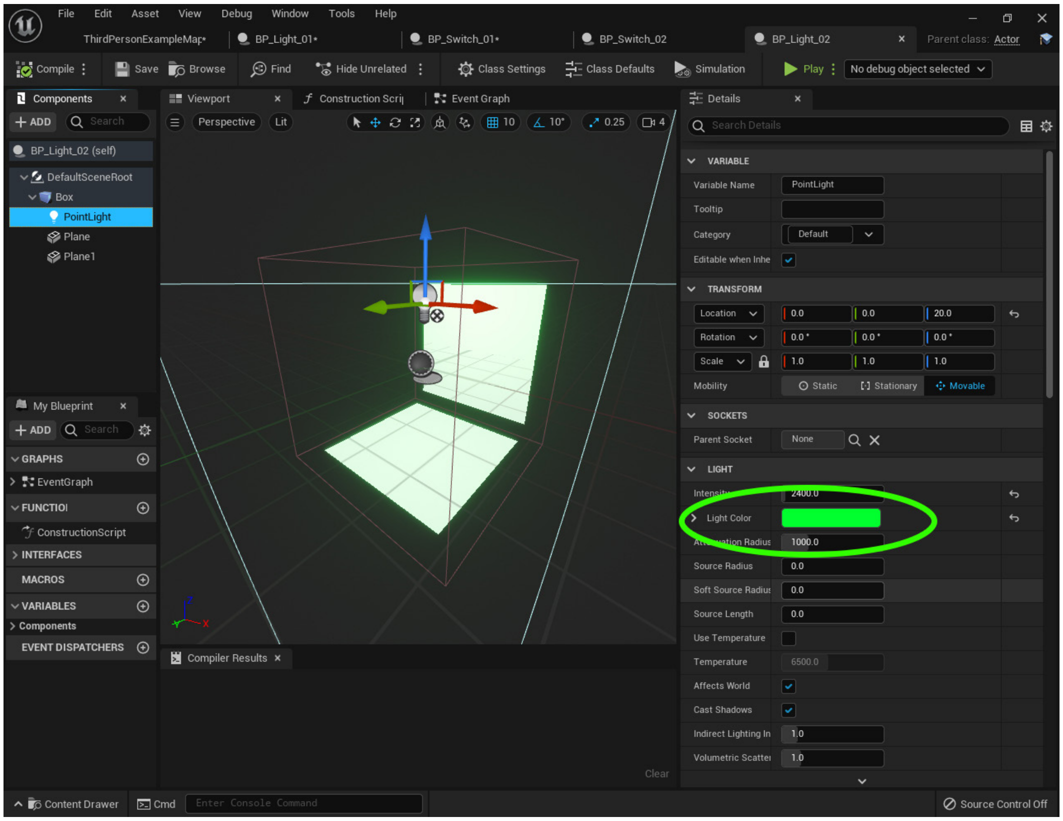


Figure 8.8: *Making a simple scene in third person template and add light source and switch actors to the scene*

Make another blueprint actor, **`BP_Light_02`**, like **`BP_Light_01`** and change the light color to green, as shown in *Figure 8.9*:



Search Details

VARIABLE

Variable Name: PointLight
Tooltip:
Category: Default
Editable when Inhe: ✓

TRANSFORM

Location: 0.0 | 0.0 | 20.0
Rotation: 0.0° | 0.0° | 0.0°
Scale: 1.0 | 1.0 | 1.0
Mobility: Static | Stationary | [✓ Movable]

SOCKETS

Parent Socket: None

LIGHT

Intensity: 2400.0
Light Color: 🟩
Attenuation Radius: 1000.0
Source Radius: 0.0
Soft Source Radius: 0.0
Source Length: 0.0
Use Temperature:
Temperature: 6500.0
Affects World: ✓
Cast Shadows: ✓
Indirect Lighting In: 1.0
Volumetric Scatter: 1.0

Figure 8.9*: Actor "BP_Light_02" viewport tab*

Now let's make a different switch to control this light source. This technique is useful for simplifying blueprint code and design *tools* for designers. Right-click and duplicate **`BP_Switch_01`** in the content browser and rename it as **`BP_Switch_02`**. Now, open its event graph and make a new public variable from **`BP_Light_02`** in it; name it **`LightSource`**. We like to save an instance of **`BP_Light_02`** in this variable. Keep in mind that you can make variables of all blueprint actors you made and use them in the blueprint code. Press the **`Compile`** button at the top, and then drag and drop an instance of this variable to event graph; as shown in *Figure 8.10*, you can directly call events from each one to control the light:

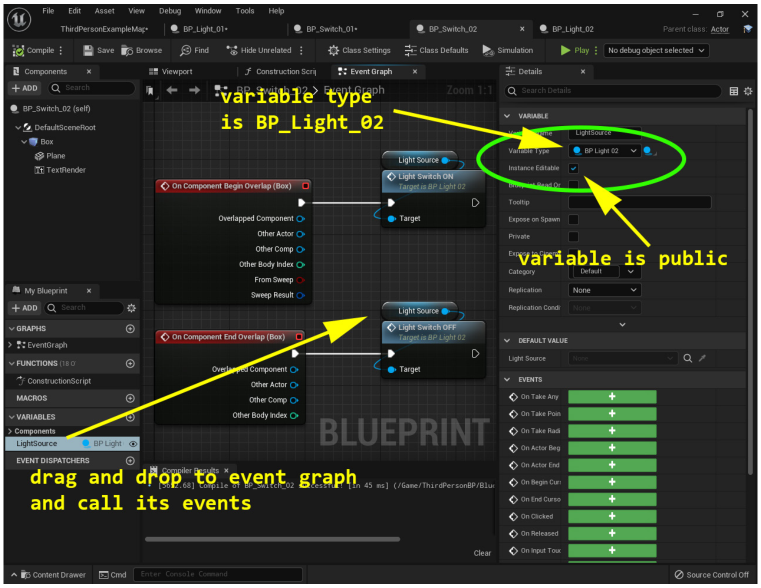


Figure 8.10: *Make a public variable in "BP_Switch_02" to hold an instance of BP_Light_02 actor blueprint. Then use it to call events which control light.*

After this, drag and drop **`BP_Switch_02`** to the scene; as shown in *Figure 8.11* and *Figure 8.12*, it has a public variable **`LightSource`**. Then, click on the **`Picker`** icon and then on **`BP_Light_02`**; now **`LightSource`** is assigned to **`BP_Light_02`** on the scene, so the code will switch the light source on **`Box Collision`**. Play the game and check the results:

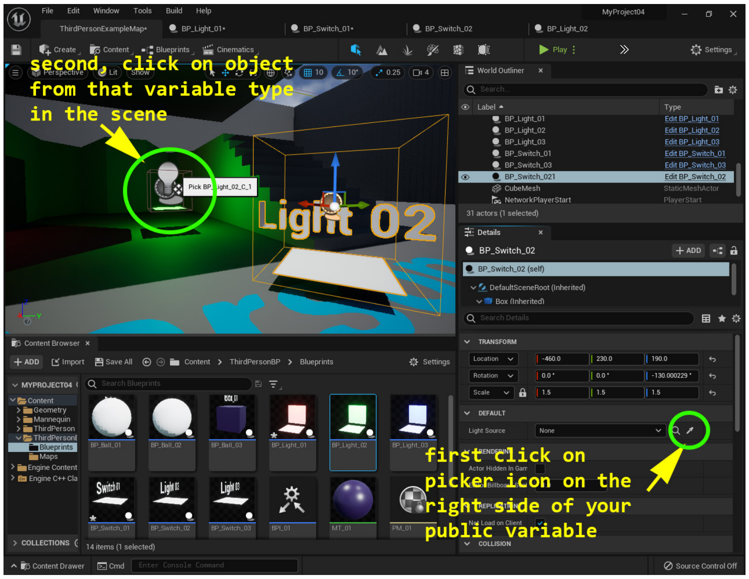


Figure 8.11: *Assign value to a public variable in the scene*

After clicking on object, double-check the name of picked object in your public variable **LightSource**:

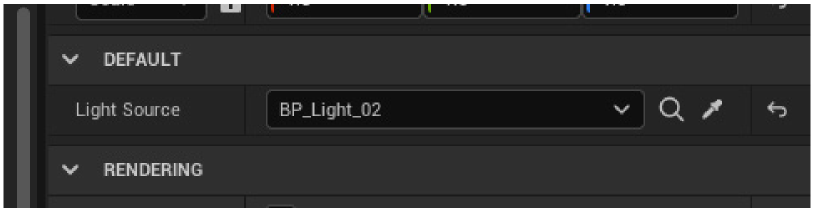
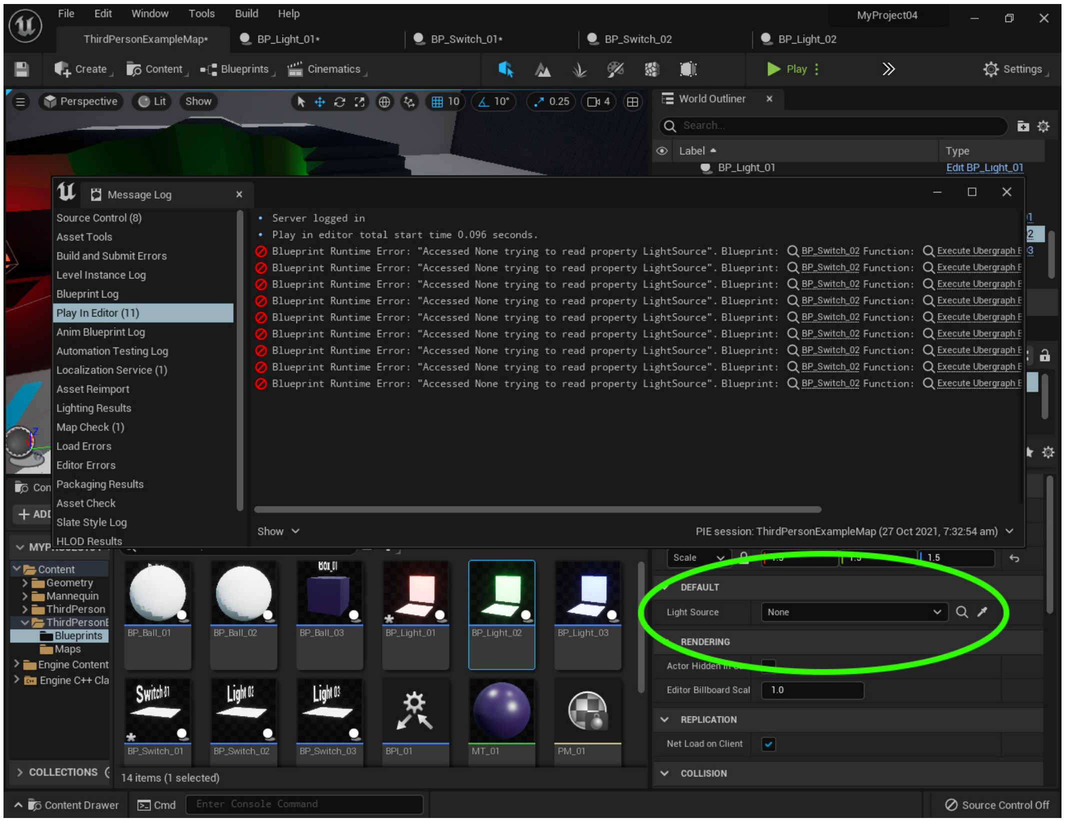


***Figure 8.12**: The public variable would show the object name if it assigned correctly*

If you forget to assign variable to **LightSource**, the engine will mostly throw an error message for **None** reference values, as shown in *Figure 8.13*, so keep in mind that you must "Always" address and assign your public blueprint variables in the scene or at begin play event inside that blueprint:



```
• Server logged in
• Play in editor total start time 0.096 seconds.
Blueprint Runtime Error: "Accessed None trying to read property LightSource". Blueprint: BP_Switch_02 Function: Execute Ubergraph B
Blueprint Runtime Error: "Accessed None trying to read property LightSource". Blueprint: BP_Switch_02 Function: Execute Ubergraph B
Blueprint Runtime Error: "Accessed None trying to read property LightSource". Blueprint: BP_Switch_02 Function: Execute Ubergraph B
Blueprint Runtime Error: "Accessed None trying to read property LightSource". Blueprint: BP_Switch_02 Function: Execute Ubergraph B
Blueprint Runtime Error: "Accessed None trying to read property LightSource". Blueprint: BP_Switch_02 Function: Execute Ubergraph B
Blueprint Runtime Error: "Accessed None trying to read property LightSource". Blueprint: BP_Switch_02 Function: Execute Ubergraph B
Blueprint Runtime Error: "Accessed None trying to read property LightSource". Blueprint: BP_Switch_02 Function: Execute Ubergraph B
Blueprint Runtime Error: "Accessed None trying to read property LightSource". Blueprint: BP_Switch_02 Function: Execute Ubergraph B
Blueprint Runtime Error: "Accessed None trying to read property LightSource". Blueprint: BP_Switch_02 Function: Execute Ubergraph B
PIE session: ThirdPersonExampleMap (27 Oct 2021, 7:32:54 am)
```

Light Source: None

Actor Hidden in Game

Net Load on Client: ✓

***Figure 8.13**: Error message when there is "None" assigned to public variable*

Using event dispatchers

So far, we have learned how to create and use events very similar to functions. Now, let's learn another method of using events in Unreal Engine known as **event dispatchers**. In this method, the engine will *Dispatch* (or run) an event in the background from the object who *broadcast* it. This event is handled by the engine as a C++ delegate event, which means the engine assigned processing and memory resources for running it by compiler in the background. At the *receiver* of this event, a user has to *assign* a custom event to detect broadcasting of the event and handle the actions required for it.

To practice this, duplicate **`BP_Switch_02`** in the content browser and rename it as **`BP_Switch_03`**. Then, open the event graph of this actor, locate **`EVENT DISPATCHERS`** in the **`My Blueprint`** tab, and as *Figure 8.14* shows, add two new event dispatchers **`ActivateLight`** and **`DeActivateLight`**. Press the **`Compile`** button at the top. Then, drag and drop each of event dispatchers to the event graph and select **`Call`** from list. As you can see in *Figure 8.14*, the event dispatcher has a special icon attached to blueprint box. This makes it easier for developers to find event dispatchers in the code. Connect these two event dispatchers to begin overlap and end overlap events of the **`Box Collision`** component. Now, on each collision event with the **`Box Collision`** component, the **`BP_Switch_03`** will dispatch a proper event:

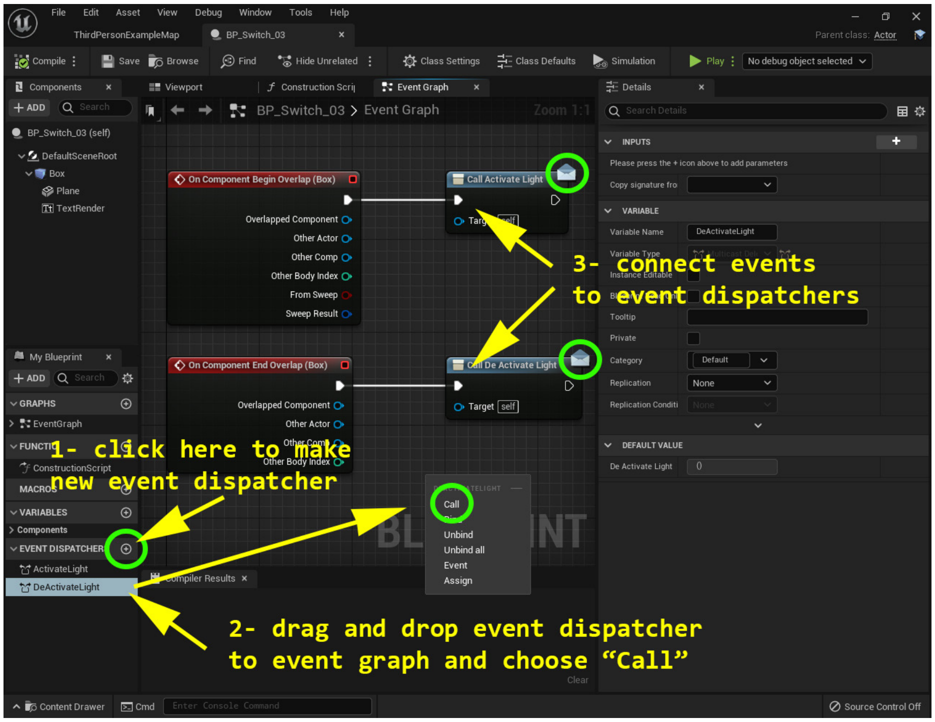

BP_Switch_03 > Event Graph
On Component Begin Overlap (Box)
Call Activate Light
On Component End Overlap (Box)
Call De Activate Light
1- click here to make new event dispatcher
2- drag and drop event dispatcher to event graph and choose "Call"
3- connect events to event dispatchers
EVENT DISPATCHERS
ActivateLight
DeActivateLight
Call
Unbind
Unbind all
Event
Assign
Variable Name DeActivateLight
Category Default
Replication None

Figure 8.14: Making new event dispatchers and "call" them on Box collision component events

Now, event dispatchers from **BP_Switch_03** need a little bit more work to activate light sources. Duplicate **BP_Light_02** in content browser and change its name to **BP_Light_03**. Open it and change the light source color to blue. Then, switch to event graph and grab an instance of **BP_Switch_03** by using the **Get All Actors Of Class** blueprint node in begin play and assign it to a variable named **Switch**, as shown in *Figure 8.15*:

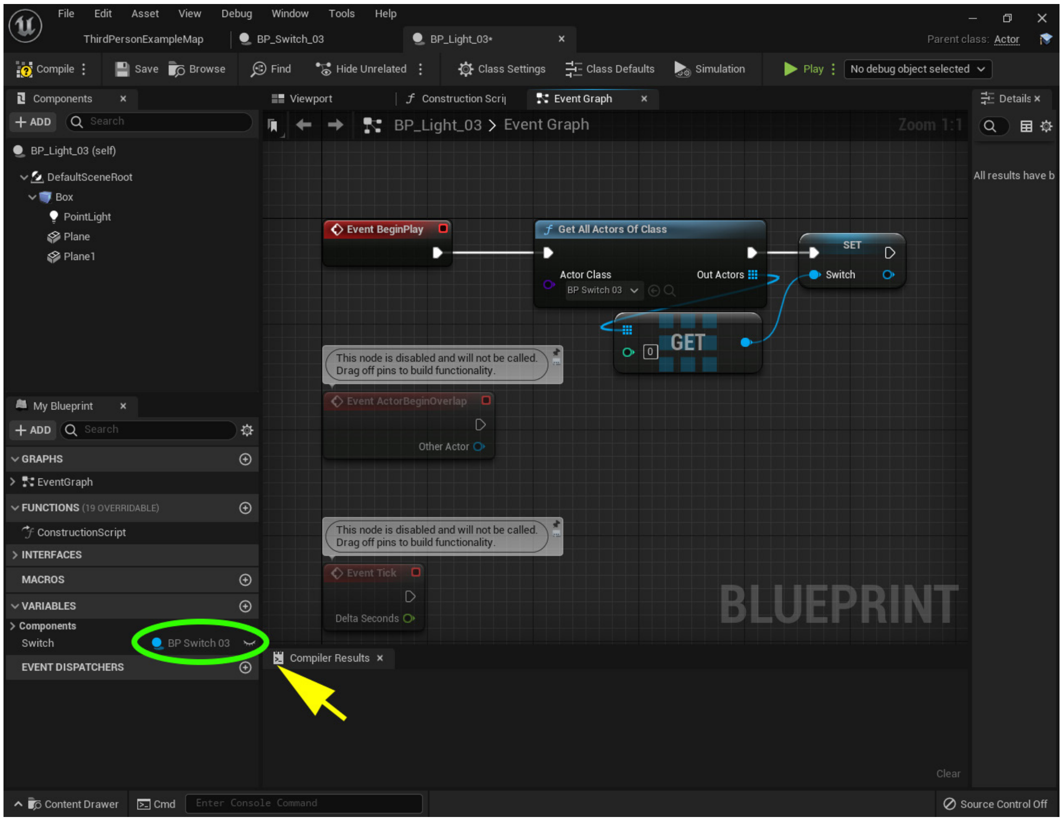


Figure 8.15: Make a new variable from BP_Switch_03 type and grab a reference to it at event begin play

Then, drag an instance of the **Switch** variable to event graph, click on the blue pin, drag the blue line and release the mouse, and then type **bind event to activate light**. The engine automatically shows the **Bind Event to Activate Light** command, as shown in *Figure 8.16*. Please do "Not" press the **Compile** button at the top at this stage. (The engine may show error title on blueprint block, and the game will not run):

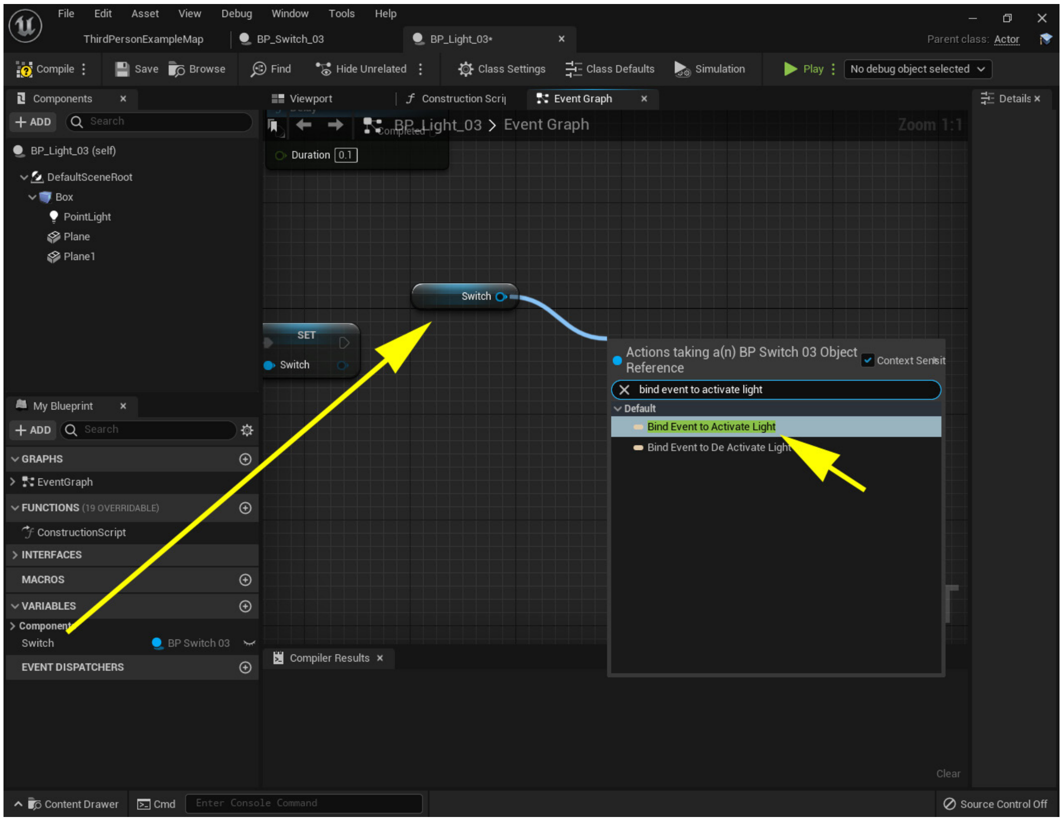


Figure 8.16: Drag and drop "Switch" variable to the event graph and bind an event to its event dispatchers

Select the **`Bind Event to ActivateLight`**, and the engine will automatically add the **`Bind Event to ActivateLight`** blueprint command to event graph. Then, click on and drag a red line from event pin, as shown in *Figure 8.17*, and then connect the red line to red pin of the **`LightSwitch_ON`** event. Now, press the **`Compile`** button at the top:

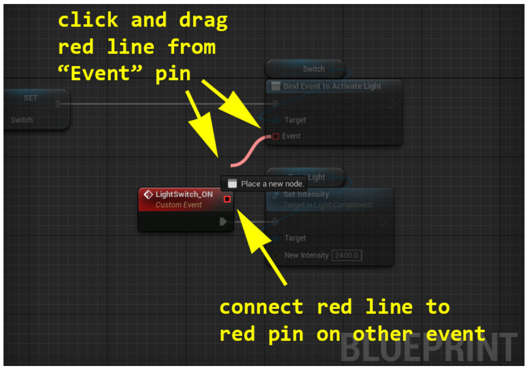
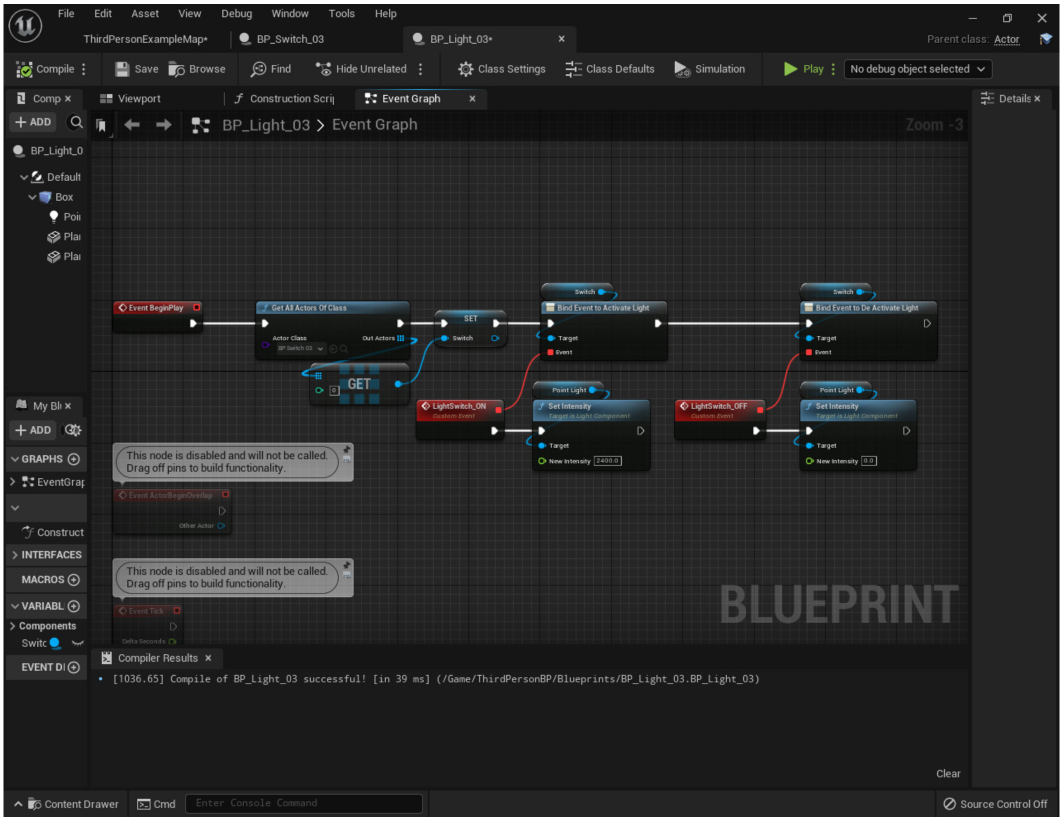


Figure 8.17: Drag and drop "Switch" variable to the event graph and bind an event to its event dispatchers

Follow the same procedure for **`Bind Event to DeActivateLight`** and **`Bind`** it (or connect events by using the red line) to the **`LightSwitch_OFF`** event. Your code should look like *Figure 8.18* at the end of this procedure. When you run the code, **`BP_Switch_03`** is dispatching an event on collision and the **`BP_Light_03`**, which already *bind* events to handle those event dispatchers from **`BP_Switch_03`**, will change the light source intensity:



This node is disabled and will not be called. Drag off pins to build functionality.

This node is disabled and will not be called. Drag off pins to build functionality.

```
• [1036.65] Compile of BP_Light_03 successful! [in 39 ms] (/Game/ThirdPersonBP/Blueprints/BP_Light_03.BP_Light_03)
```

Figure 8.18*: Actor “BP_Light_03” event graph*

Event dispatchers gave developers a nice way to broadcast events and data with all game objects in the scene at runtime. To communicate with the event dispatchers, you just need to bind your own events to the source of event dispatcher, as we did in the previous example. In the previous methods, used in **`BP_Switch_01`** and **`BP_Switch_02`**, the sender *directly* grabs light source object in the scene and runs an event inside it, which means the sender must have a *reference* of target. However, when event despatcher runs an event, it doesn’t matter *who the target is*, each target can grab a copy of sender, bind an event to *event dispatcher* of it, and respond properly. This is the most important difference between *direct call* of an event and *bind to event* in Unreal Engine.

Using interface

Before looking at this topic in depth, we need to talk about how to use line trace and what casting is in blueprint programming.

Line Trace in blueprint

Using *Line Tracing* in game development is a powerful tool and has many usages, depending on the scenario of the game and the nature of the problem the developer wants to solve. Imagine that you have a laser pointer in your hand, as shown in *Figure 8.19*. Each time a user turns it on, a laser light goes from your hand to the object that the laser pointer is pointed at:

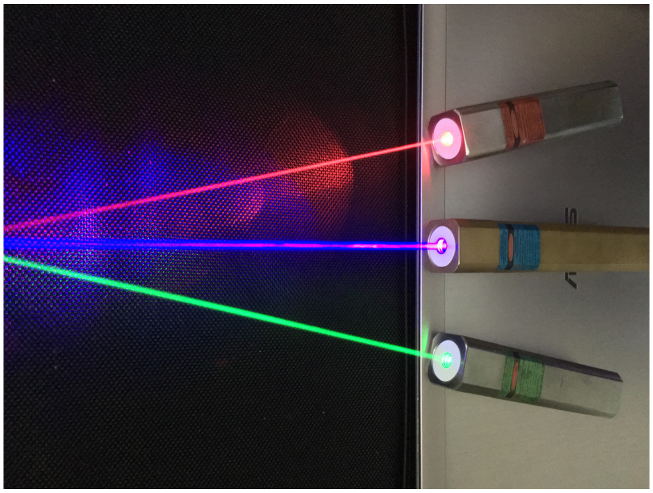
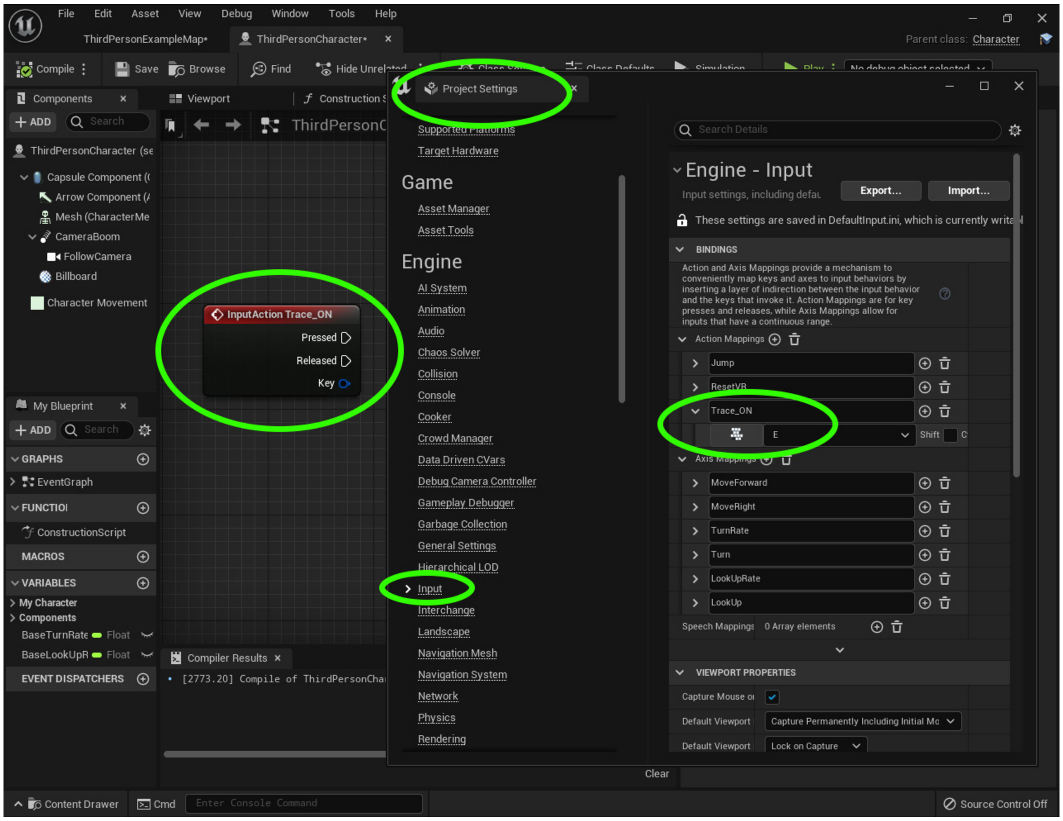


***Figure 8.19**: "Laser Pointers" devices with a power supply which generate laser beam of visible light.*

In most game engines, there are tools for developers to simulate and show such an effect inside the level. Unreal Engine 5 uses the **`Line Trace By Channel`** blueprint command, which performs a collision trace along a given line and returns the first object that the trace hits. Let's try this in the current project by using the third-person character's camera as shooting object for laser light. Refer to *Chapter 5, Project Templates and User Interaction*, and make a keyboard input action event for *E* key called **`Trace_ON`** in **`Third Person Character`** blueprint, as shown in *Figure*

8.20. We will use this key to directly call events on game objects in the stage during runtime:



Engine - Input

Input settings, including defau

These settings are saved in DefaultInput.ini, which is currently writa

BINDINGS

Action and Axis Mappings provide a mechanism to conveniently map keys and axes to input behaviors by inserting a layer of indirection between the input behavior and the keys that invoke it. Action Mappings are for key presses and releases, while Axis Mappings allow for inputs that have a continuous range.

Action Mappings: Jump, ResetVR, Trace_ON (E, Shift)

Axis Mappings: MoveForward, MoveRight, TurnRate, Turn, LookUpRate, LookUp

Speech Mappings: 0 Array elements

VIEWPORT PROPERTIES

Capture Mouse on: ✓

Default Viewport: Capture Permanently Including Initial Mo

Default Viewport: Lock on Capture

BaseTurnRate Float

BaseLookUpR Float

[2773.20] Compile of ThirdPersonCha

Figure 8.20: *Add new input mapping for "E" key on keyboard*

Now, right-click on the third-person character blueprint event graph and type **`line trace by channel`**; then, click on the **`LineTraceByChannel`** blueprint command and connect it by execution line to the **`Trace_ON`** event. As *Figure 8.21* shows, each **`LineTraceByChannel`** blueprint command needs some input values and has some options. Additionally, a user can click on the little arrow in the blueprint block to expand its properties:

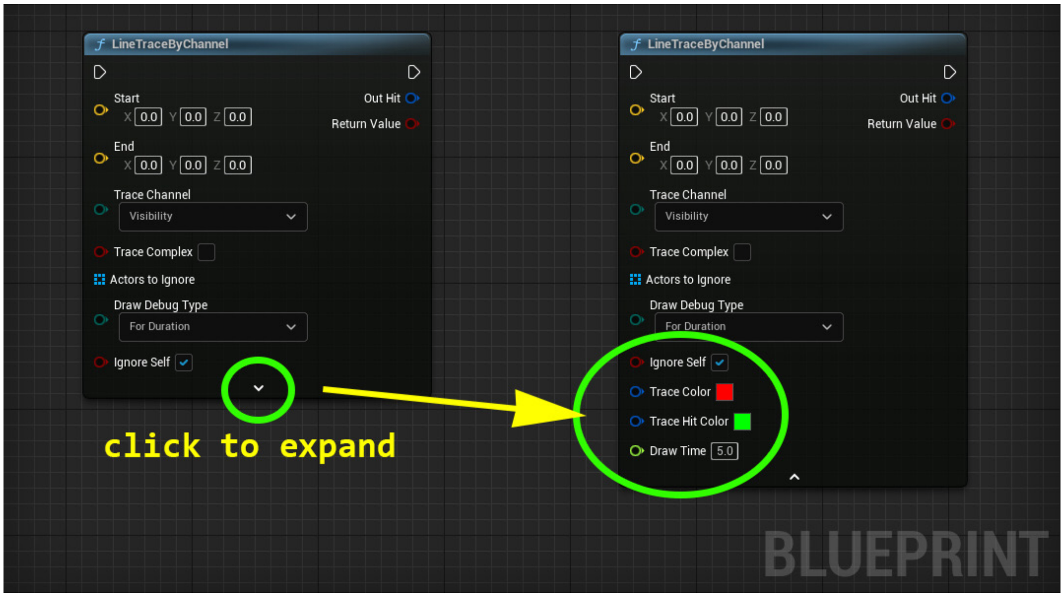
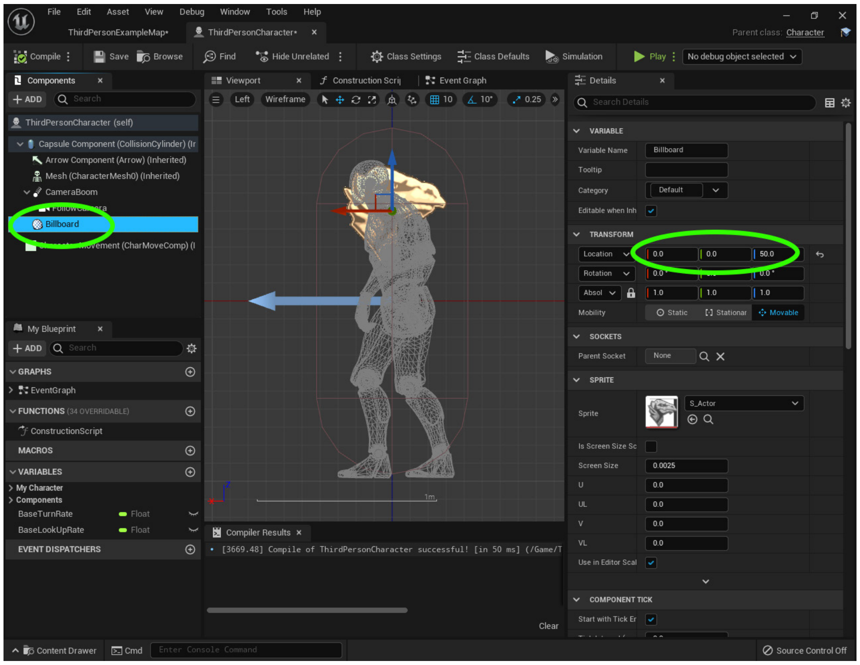


Figure 8.21: *"Line trace By Channel" blueprint block*

The **`Start`** and **`End`** pins represent the beginning and end points of the laser pointer; to make them, add a billboard blueprint component to the **`Capsule Component`** of the third-person character blueprint and relocate it to 0,0,50, as shown in *Figure 8.22*:



```
[3669.48] Compile of ThirdPersonCharacter successful! [in 50 ms] (/Game/T
```

Figure 8.22: *Use a billboard component to set start point of line trace*

Then, drag and drop the **`Billboard`** and **`Follow Camera`** components into the event graph. For the **`Start`** pin, which is a vector, we use Billboard world location by using the **`Get World Location`** blueprint command. For the **`End`** pin, we need to find **`Forward Vector`** of the camera; it is the direction we like to shoot the laser pointer in. After that, the **`Forward Vector`** must be multiplied by any number to increase the range. For doing this, use **`Get Forward Vector`** to get forward vector of follow camera component, and then use the **`Multiply`** blueprint operator to multiply a vector by a *float* value. After this, the value is ready to add to start point, which is the billboard location. Use the **`Add`** blueprint operator for vectors and connect it to the **`End`** pin. Finally, click on **`Draw Debug Type`** and change the option to **`For Duration`**. This shows the **`Line Trace`** on the screen for 5 seconds. To change the duration and color of the line trace, click on the little arrow in the **`LineTraceByChannel`** blueprint block and change the values. Your final code should look like *Figure 8.23*:

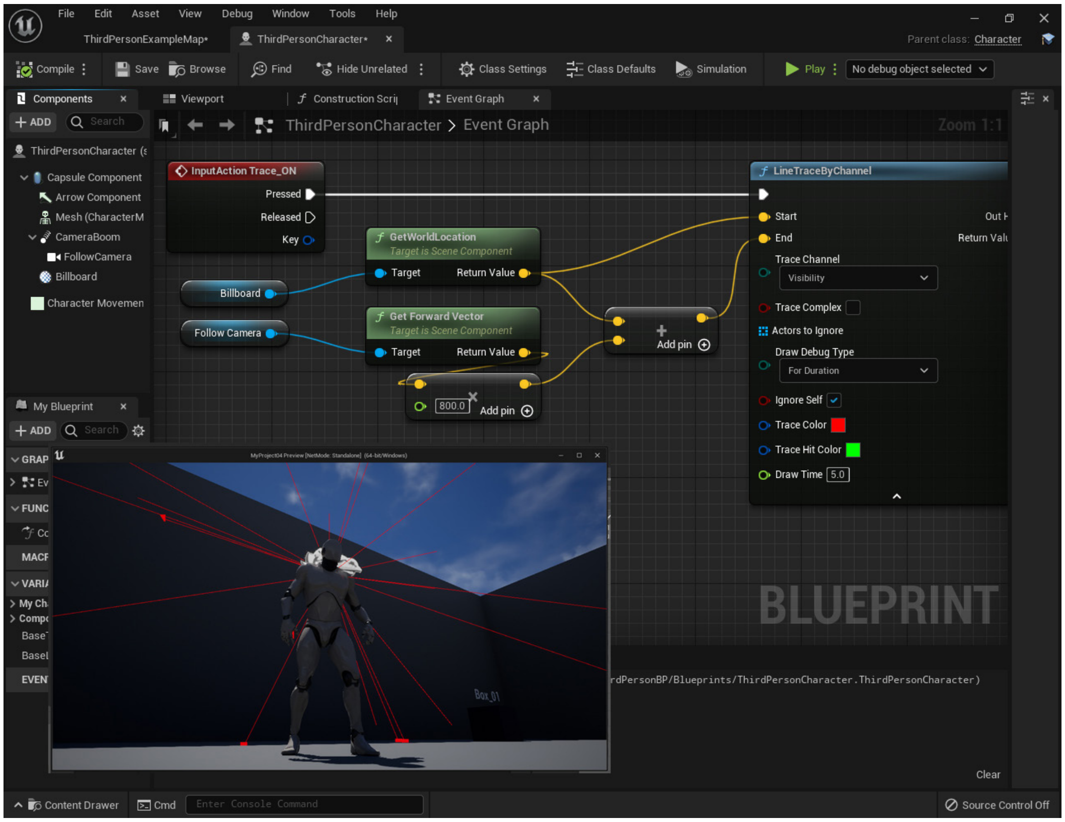


***Figure 8.23**: Final blueprint code and result on using E key, to activate line trace from third person player toward camera angle*

Now play the game and press *E* to activate line trace. You can change the location of **`Billboard`** to customize line trace start, and change multiplier value to change the laser bin range.

Now, how about we get rid of all switch actors and just use our laser light to control the light sources? This way, we don't need to use collision from switch actors at all, so we can remove them from stage.

For this, we need to learn *Casting* in blueprint first.

Casting in blueprint

So far, we have learned that line trace is a tool that draws a straight line between two points during runtime in a game based on begin and end point. Open the third-person blueprint and find the **`LineTraceByChannel`** blueprint command you already made. Then use mouse to drag an execution line form **`Out Hit`** pin and type **`break`**, and then select **`Break Hit Results`** to add this blueprint block to event graph and expand it by clicking on the small arrow on it, as shown in *Figure 8.24*:

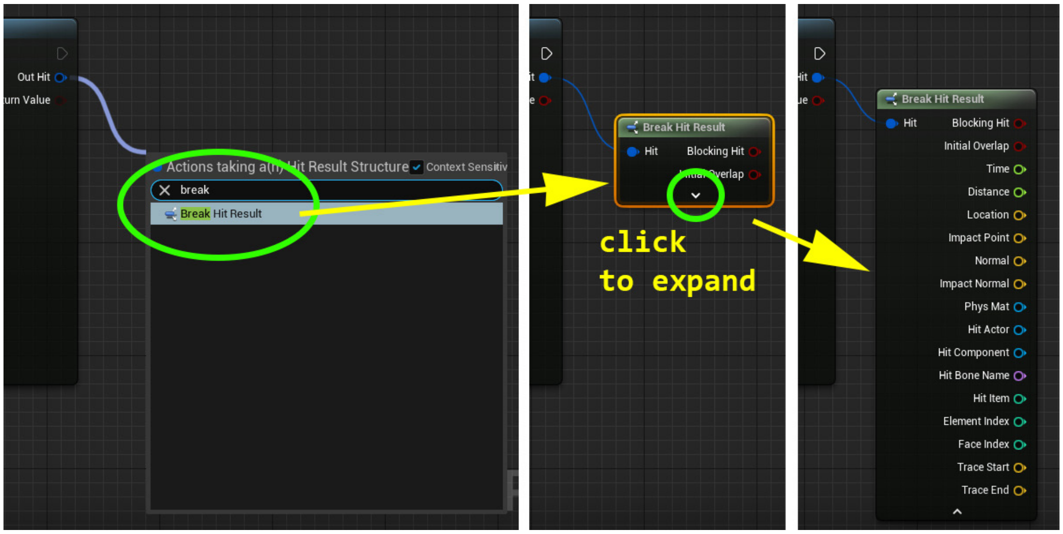
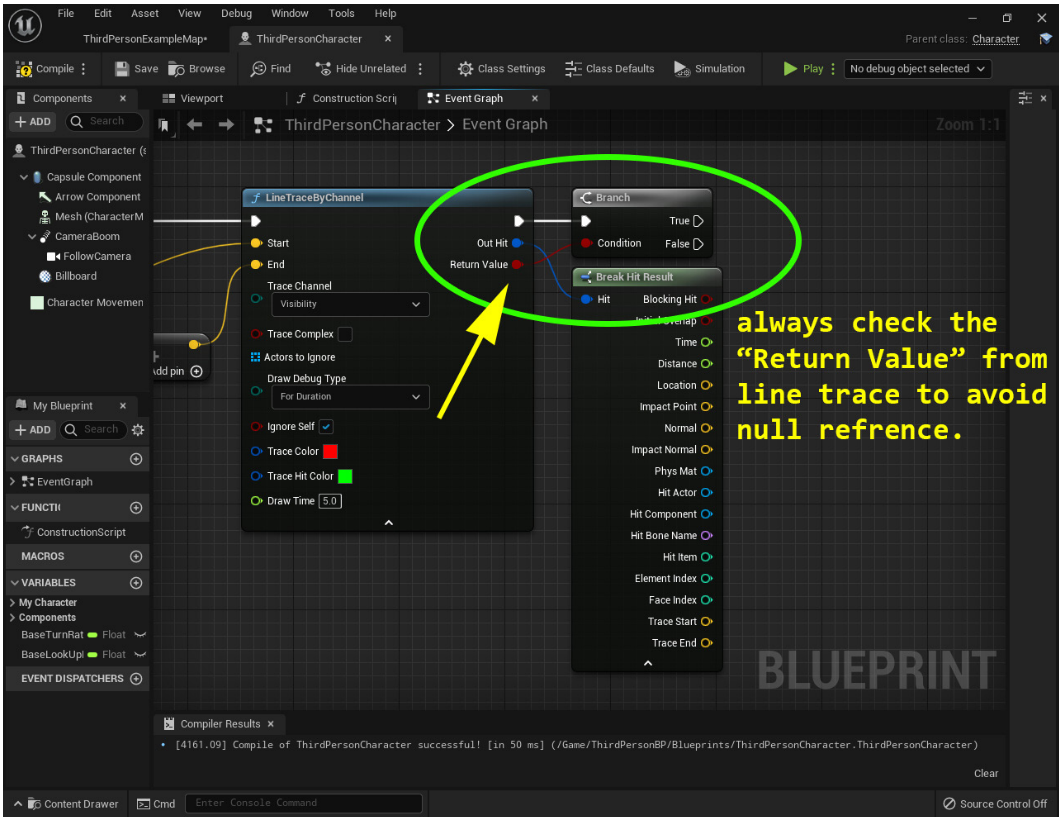


***Figure 8.24**: Make "Break Hit result/" blueprint node from line trace "Out Hit" pin*

As you can see, when we break the **`Out Hit`** pin, it returns several pieces of data. One of them is a reference to **`Hit Actor`**, which is the actor the laser light has impact with. This actor can be stage items, other blueprint actors or *nothing*, which means laser light did not impact with any object at the **`End`** of its tracing line. So, the first thing we have to do is check if the **`Hit Actor`** exists, which can be done by checking the

`Return Value` Boolean pin of the **`LineTraceByChannel`** block with an if statement, as shown in *Figure 8.25*:



```
• [4161.09] Compile of ThirdPersonCharacter successful! [in 50 ms] (/Game/ThirdPersonBP/Blueprints/ThirdPersonCharacter.ThirdPersonCharacter)
```

Figure 8.25: Sometimes line trace returns "null" reference, and you have to check it before using hit result values

The next step is to know *what* the **`Hit Actor`** is; if the **`Hit Actor`** is **`BP_Light_01`**, then we can run events on this actor and control light source. So, to discover the actor type, we need to use a method in programming, known as **casting**. Casting is a common communication method where you take a reference to an actor and try to convert it to a different class. If the conversion is successful, we can directly use this actor as a reference to access its properties and events (like we did in **`BP_Switch_01`** and **`BP_Switch_02`**).

Click and drag blueline from **`Hit Actor`**, then release and type **`BP_Light`**, and then choose the **`Cast to BP_Light_01`** blueprint block, as shown in *Figure 8.26*:

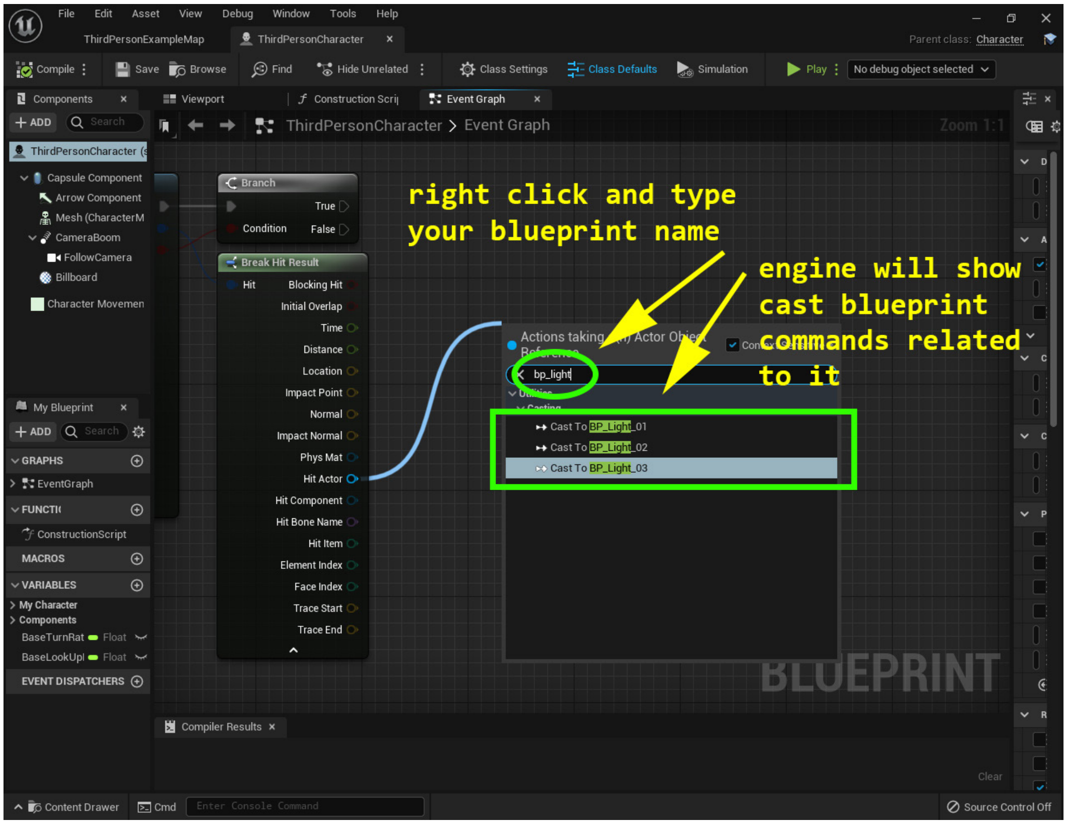

right click and type your blueprint name
engine will show cast blueprint commands related to it
bp_light
Cast To BP_Light_01
Cast To BP_Light_02
Cast To BP_Light_03
Hit Actor
Break Hit Result
Branch
ThirdPersonCharacter > Event Graph

***Figure 8.26**: Casting in blueprint*

Then, click on its **As BP Light 01** blue pin, drag blueline, release and type **light switch**, and then choose the **Light Switch OFF** blueprint event from the list, as shown in *Figure 8.27*. Press the **Compile** button at the top and run the game:

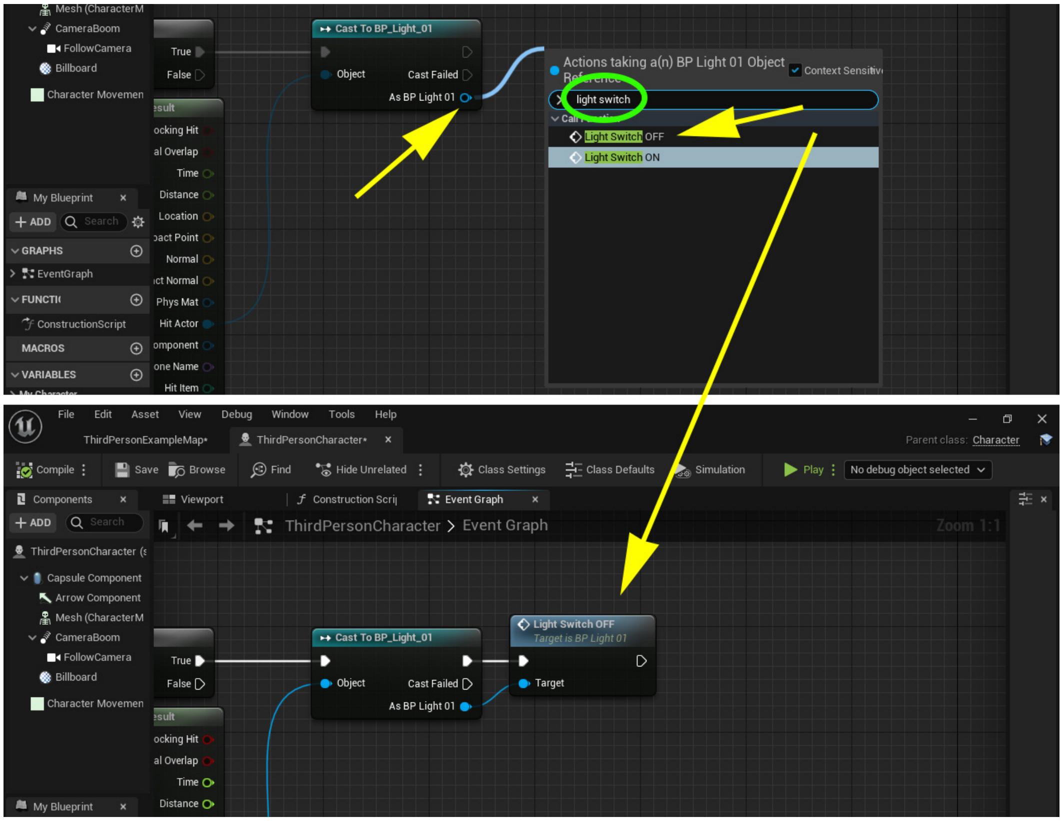
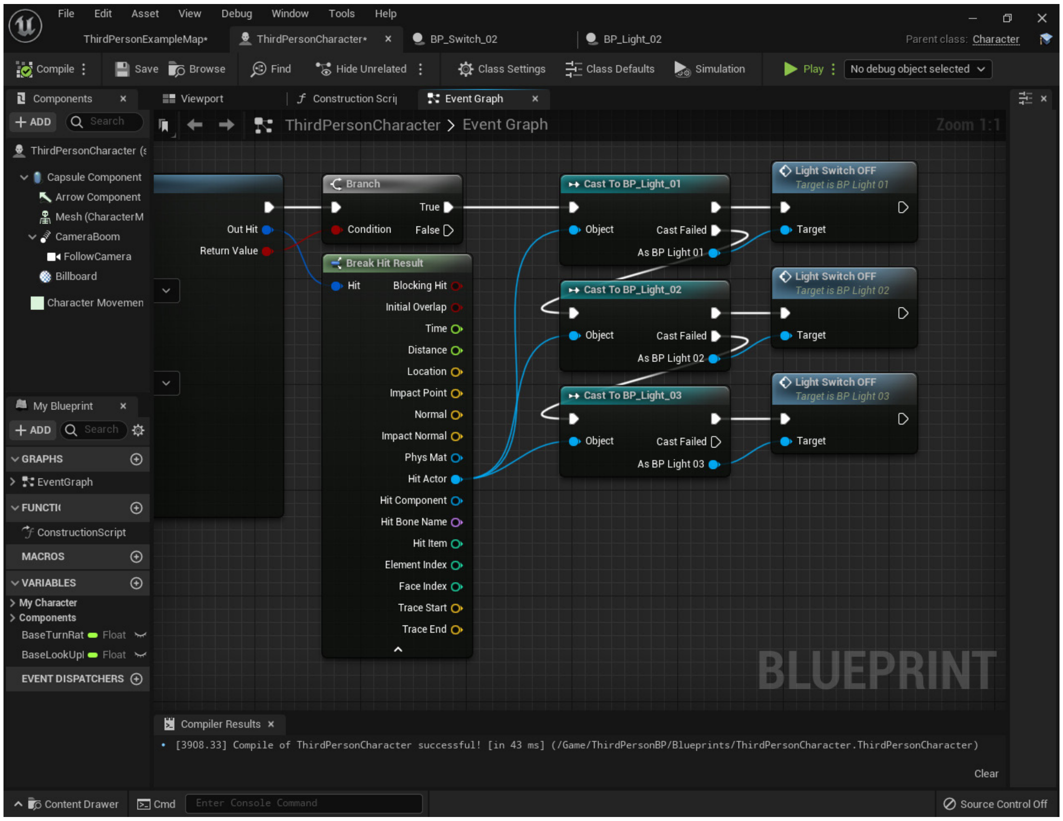


Figure 8.27: The "As BP Light 01" returns a reference to BP Light 01 actor on the stage. So, we can call "Light Switch OFF "event from this object to control light.

When the user presses *E* on the keyboard, a line trace in drawn by the engine inside the game; then, the code checks the **Return Value** of line trace. If the laser impacts with an object, then **Return Value** will be true, and then the code *tries to cast* the **Hit Actor** to the **BP_Light_01** blueprint actor. When we say *try to cast*, it means the engine grabs **Hit Actor** and compares it to the **BP_Light_01** source class in the content browser. If these two are *the same*, then the engine executes the top execution pin in the blueprint, which will run the **Light Switch OFF** event from light source actor. If the objects are not the same, the **Cast Failed** execution pin will be activated.

Now, follow the same procedure and add two more cast blueprint blocks to support the **Light Switch OFF** function on both **BP_Light_02** and **BP_Light_03**. As shown in *Figure 8.28*, we can use the **Cast Failed** execution pin to execute multiple casts. This way, the engine will run first cast, and if that fails, it will check for the next one; if both of those cast fails, it will check the last cast, and on each one, the **`Light Switch OFF`** function will be executed if the cast is successful:



```
• [3908.33] Compile of ThirdPersonCharacter successful! [in 43 ms] (/Game/ThirdPersonBP/Blueprints/ThirdPersonCharacter.ThirdPersonCharacter)
```

***Figure 8.28**: Using cast on three actors to activate an event*

When the game is running, the third-person player can switch off all light sources. But there's one problem: assume that we have 16 different light source actors on the scene in game. To switch off any of them, we have to do 16 cast(s) in third-person character and run similar event to **`light_switch_off`** on each. Imagine that the light source objects are huge blueprints with a number of elements and details, and when you are casting to each in runtime, the engine should load them into memory, compare them to what **`Line Trace`** returns as **`Hit Actor`**, and if the cast fails, free the memory used for loading and load the next object in memory; this will go on until it finds the right object to cast and then run the event on it. This is not only unprofessional but also abuses engine processing and system resources during running game.

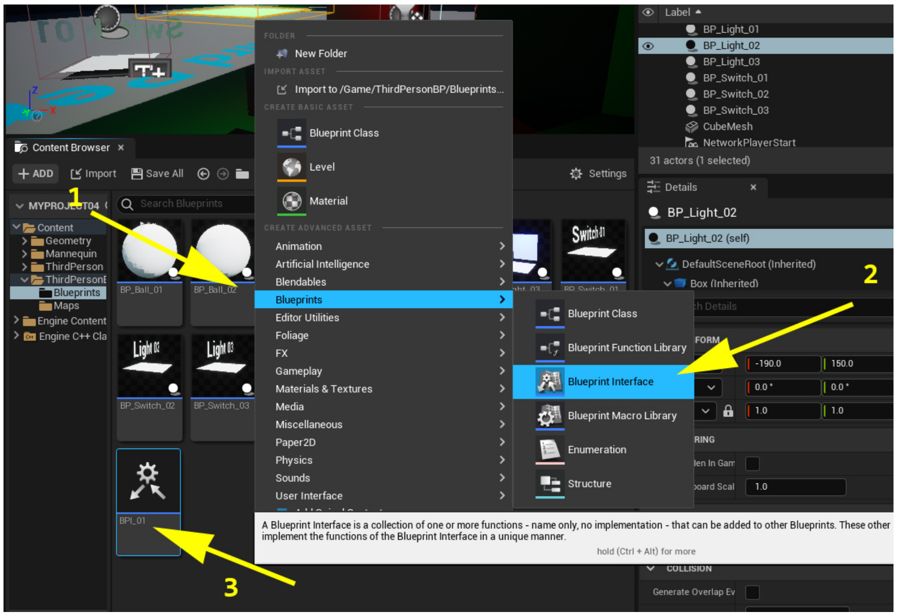
To solve this issue, we need to use a technology in programming known as **interface** implementation.

Interface implementation in blueprint

Using interface is an advanced knowledge in developing applications. It works like magic and simplifies the code and solutions. The interface itself is a programming syntax/structure that allows the computer to use certain properties and functions on an object/class. Imagine that you have a robot that can speak fluently in five different languages. This means your robot's brain (like a computer application) has five different input and output structures to *receive* and *send* data in each of those languages. Consider this example: it can *detect* the message **`Good Morning`** in five languages and *respond* properly in a specific language, as it's brain (like a computer application) has the *interface* of that language to receive it and respond to it. So, if you upgrade this robot with a new interface of a different language and plug it into its brain (add that to its computer application), your robot can communicate in that language as well.

Having *interface* is a very expensive technology not many programming languages can support to compile and implement. The Unreal Engine blueprint programming language provides in-depth support for development and use *Interface* in the code. We will describe the basics of *Interface development* in this section.

Right-click in the content browser and select **`Blueprint Interface`** from the **`Blueprint`** menu, as shown in *Figure 8.29*. Rename the blueprint interface as **`BPI_01`**. The **`BPI`** prefix stands for **`Blueprint Interface`**, and following this naming convention is an industry standard:



A Blueprint Interface is a collection of one or more functions - name only, no implementation - that can be added to other Blueprints. These other implement the functions of the Blueprint Interface in a unique manner.

hold (Ctrl + Alt) for more

Figure 8.29: 1- Right click and select "Blueprint" from content browser. 2- Locate "Blueprint Interface" and left click on it. 3- Rename new interface to "BPI_01".

Double-click on **BPI_01** in the content browser to open its editor. The blueprint interface editor, as shown in *Figure 8.30*, is very simple compared to previous blueprints we made. Click on the **+ADD** button, and the engine will add a new function in the **FUNCTIONS** section. Rename this function to **Impact** and save it:

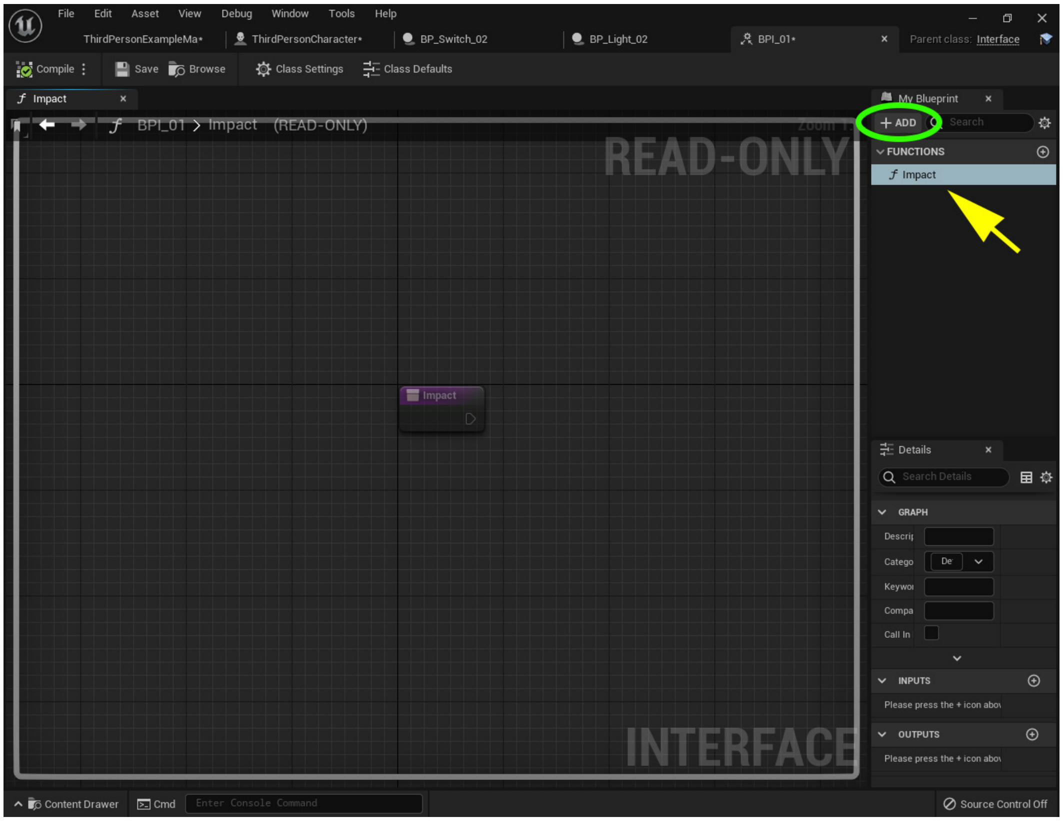


***Figure 8.30**: Click on +ADD to add new functions in blueprint interface editor*

Now we need to upgrade our game objects with this interface to control light. Open each light source blueprint, click on **Class Setting** and locate **INTERFACES** in the **Details** panel. Then, click on the **Add** drop-down button, as shown in *Figure 8.31*, and locate the **BPI_01** interface you made and click on it. Press the **Compile** button at the top, and the actor will be upgraded with interface and can use the **`BPI_01`** interface for communication:

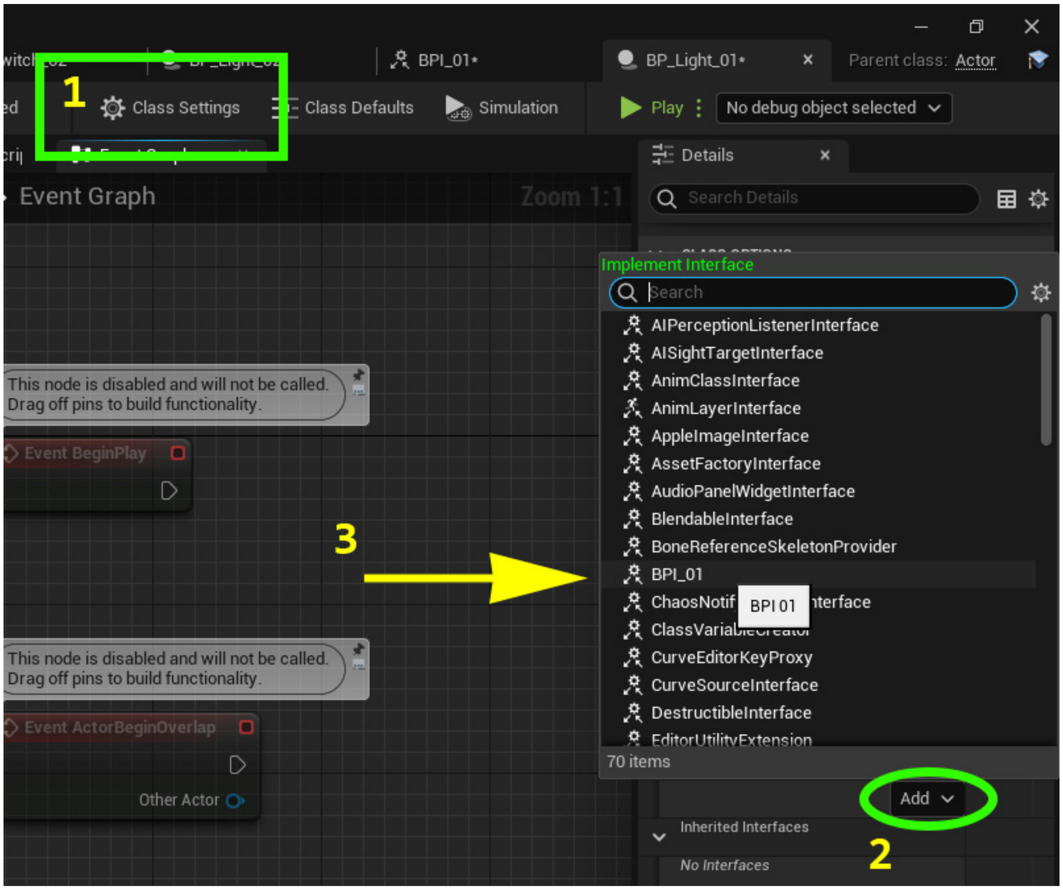


Figure 8.31*: 1- Locate and click on Class Setting tab on each light source actor, 2- Click +ADD in details panel. 3- Find BPI_01 interface and click on it.*

After assigning interface to all light sources, save the project, close it and open it again. Then, open the third-person character blueprint and locate the **`LineTraceByChannel`** blueprint command. Disconnect the casts and then click and drag blueline from **`Hit Actor`**; then, type **`Impact`**, which is the name of the function you already made in the **`BPI_01`** blueprint interface, as shown in *Figure 8.32*. Then, click on the execution line to this function. As you see, the interface's function has a different icon in its blueprint block, which makes it easier to detect during development code in event graph. Press the **`Compile`** button at the top:

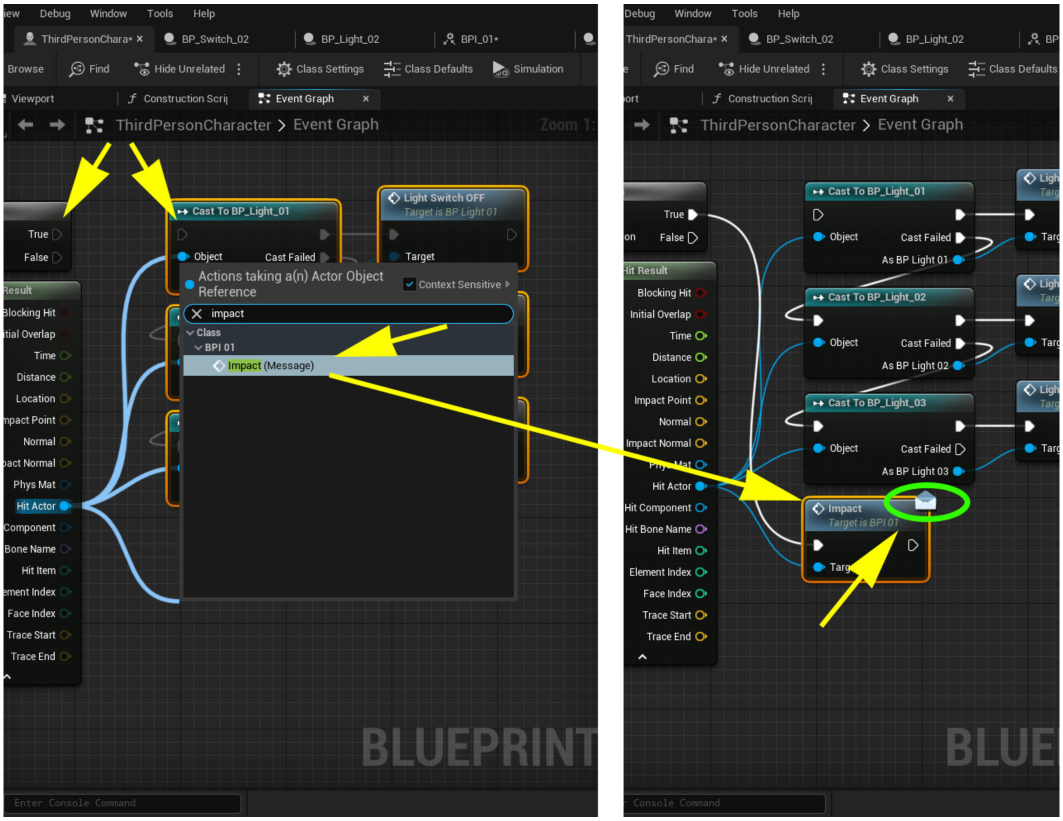


Figure 8.32: *1- Open the third-person character blueprint and disconnect cast commands. Then, drag the "Impact" function from Hit Actor and connect it to execution line. The blueprint interface functions have special icon on their blueprint block, like event dispatchers in switch actors. IF you don't have "Impact" name on the list, save project, close it and open it again and then try one more time.*

Now, open each light source, and right-click in event graph and type **`Impact`**; select **`Event Impact`** in the **`Add Event`** section in the list, as shown in *Figure 8.33*. (If you don't have **`Impact`** name on the list, press **`Compile`**, save project, close it, open it again and then try):

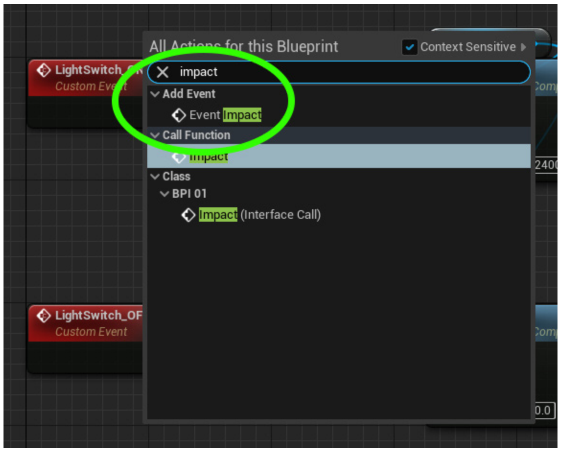


Figure 8.33: *Right-click in event graph and type "Impact". The interface function which we made will come in the list multiple times. Choose "Event Impact" from the list.*

The event belonging to interface has a special icon, which makes it easier to detect in event graph. Now, disconnect other events in blueprints, add a **`FlipFlop`** to interface event (**`Impact`**) and connect it to source light events, as shown in *Figure 8.34*. We also use a Flip Flop to switch light between on and off. Flip Flop node is switched execution line between **A** and **B** that leads to different events in the light source. Run the game and check the magic of using interface by pressing *E* on each light source:

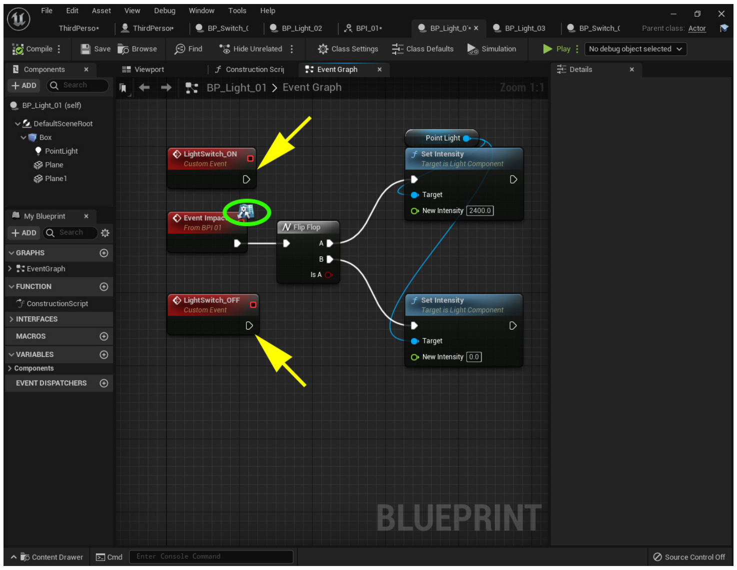


Figure 8.34: *Disconnect previous events and calling light source events by "Impact" interface event. The interface event has a special icon on its blueprint block.*

Each time a user presses the *E* key, the line trace hits an actor in the scene and then checks if the actor has the **`BPI_01`** interface; if it has the interface, the engine will run it immediately. In other languages, we don't need to cast anymore; we just check if an interface exists at the destination and then activate it. On each light source, the blueprint interface runs its event, which is implemented as **`Impact`** in the interface.

Using physics (advanced)

Unreal Engine uses PhysX by default to drive a physical simulation calculation to perform all collision calculations and physical rules. So far, we have learned about physical constrains and used collision events to track the third-person character; now, it's time to learn more about some physical tools in engine.

Using physical gravity

Make a new actor blueprint called **`BP_Ball_01`**, which has a **`Sphere`** collision component with **`Sphere`** static mesh component resized inside it and a text render component to show its title, as shown in *Figure 8.35*:

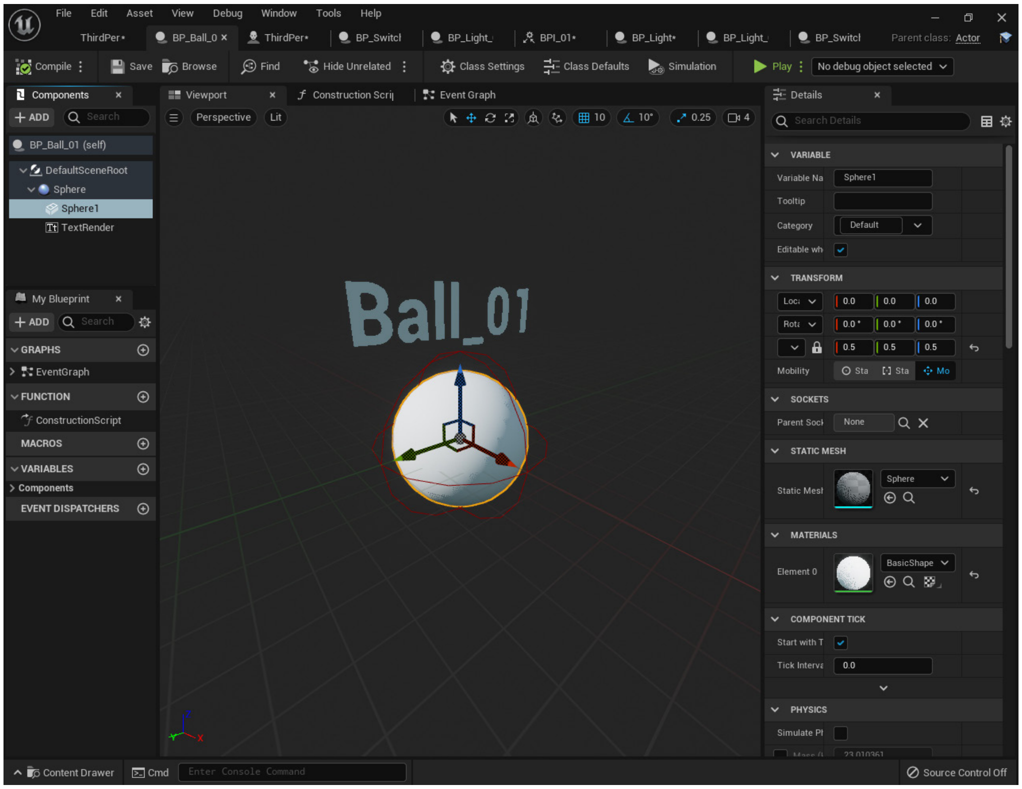


Figure 8.35: *Ball_01 actor with Sphere collision component, Sphere static mesh component and Text render component*

Now, switch to event graph and refer to the beginning of this chapter, make a new custom event and name it **`ActivateGravity`**, drag and drop a reference from the sphere collision component, and get the **`Set Simulate Physics`** blueprint block from it, check the **`Simulate`** pin to true and connect this block to the custom event, as shown in *Figure 8.36*. Press the **`Compile`** button at the top:

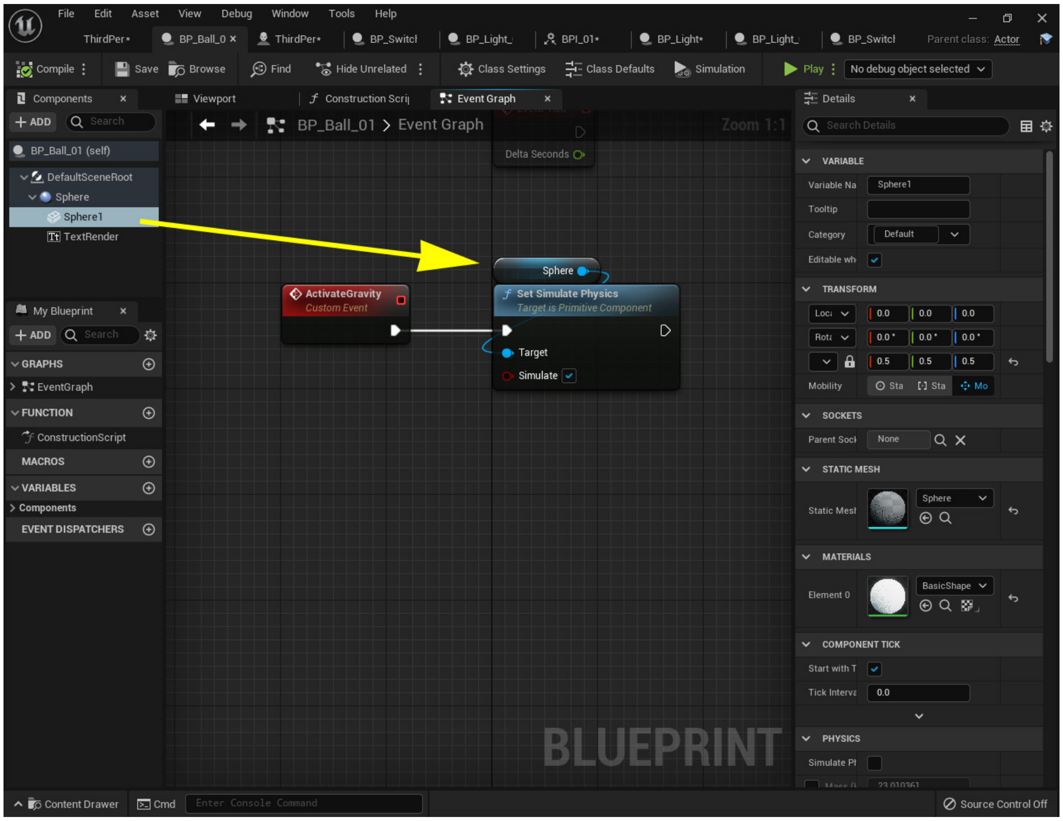


***Figure 8.36**: Ball_01 actor event graph*

Now, open the third-person character blueprint and execute the **`ActivateGravity`** event from this object with a simple cast in line trace, as shown in *Figure 8.37* (we can assign and use interface to **`ball_01`** for this, but for now, let's make it simple with cast):

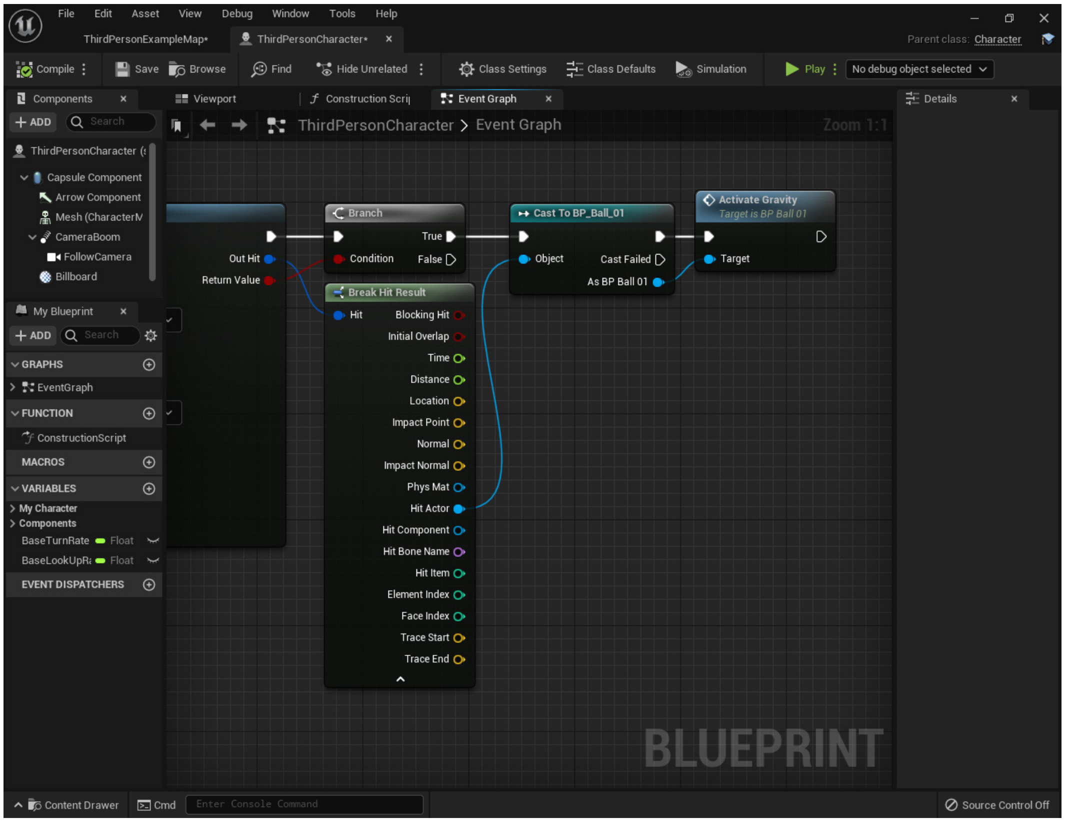


Figure 8.37: Remove previous code, and then Cast to ball_01 actor and run activate gravity event from it with line trace in third person character blueprint

Now we need to make the scene to test the physics. Drag and drop two instances of **`BP_Ball_01`** to the scene from content browser and locate them in the middle of air. When you run the game as presented in *Figure 8.38*, by shooting laser light (line trace) to each **BP_Ball_01**, it will fall to the ground, and you can move them by pushing these objects with the third-person character:

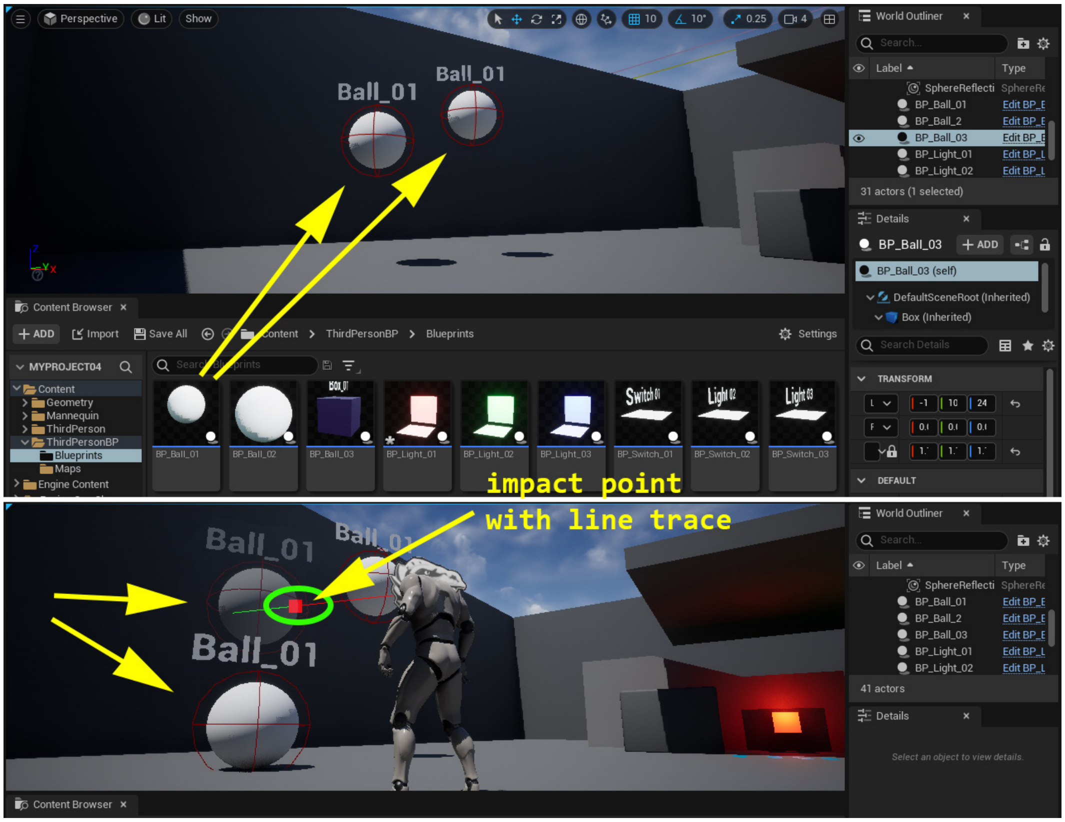


Figure 8.38: Top: Drag and drop to the scene some instances of Ball_01 and put them in the air.

Bottom: Run the game and shoot line trace to each.

Add physical force with impulse

Now, let's make *shooting object* functionality for this actor by pressing *E* on the keyboard (sounds exciting, isn't it!). For this action, we first need to *spawn* the ball actor in the world and then add *physical force* to move it in the direction we like. Duplicate **BP_Ball_01** in the content browser, rename it as **BP_Ball_02** blueprint and navigate to event graph. Make a new vector variable called **Direction** and then make a float variable called **Speed**. For each variable, make sure to tick **Expose on Spawn** in their details panel and make them public, as shown in *Figure 8.39*. Press the **Compile** button at the top. Now, drag and drop an instance from sphere collision component to the event graph, and grab the **Add Impulse** blueprint block out of it. As you see, this block accepts an input vector known as **Impulse** that represents a three-dimensional force value each on the *X*, *Y* and *Z* axes. We can increase force on each axis by using high values on that axis, so as a simple practice, we use **`Speed`** to represent force and **`Direction`** as input variables, and by multiply speed to direction, we can simply make an **`Impulse`** input.

Then, use **`Set Simulate Physics`** to activate physic simulation, and connect the **`Add Impulse`** node to **`Event Begin Play`** and press the **`Compile`** button at the top. Now, everything is ready; we just need to spawn this actor by shooting it from third-person character:

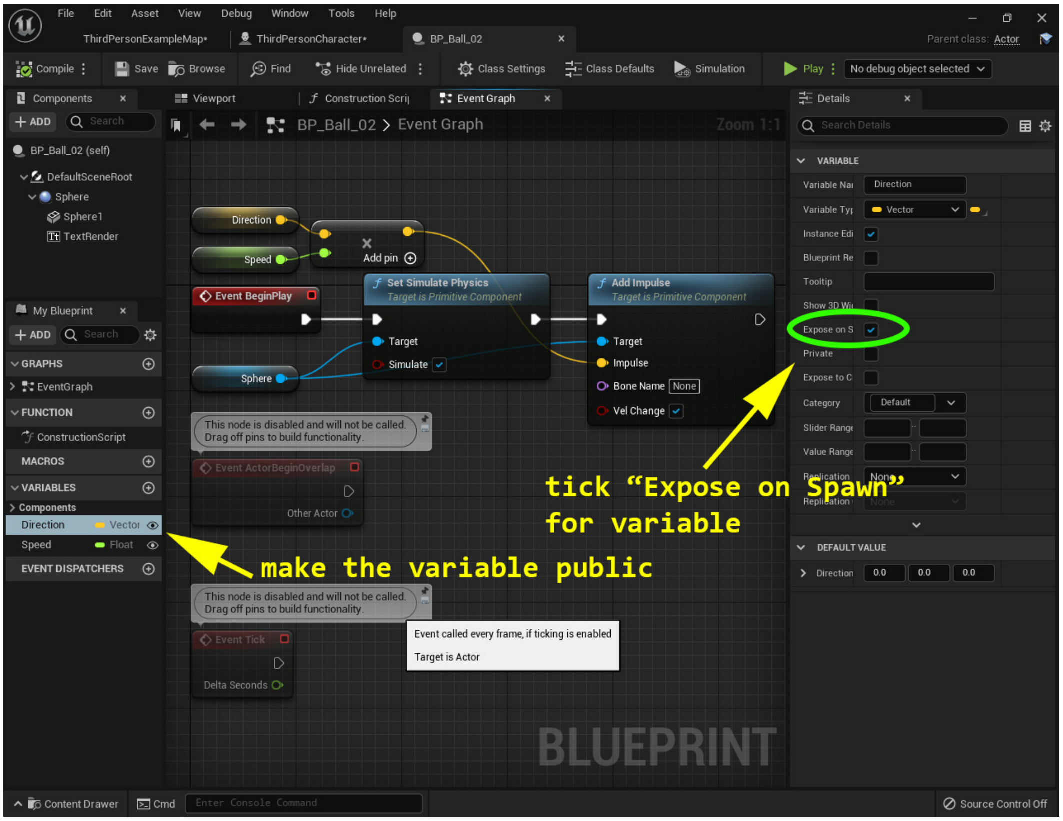


Figure 8.39: Ball_02 actor blueprint event graph

Open third-person character blueprint and locate your keyboard event. Then, right-click on the event graph, type *spawn Actor from class*, and then select the blueprint block, as shown in *Figure 8.40*:

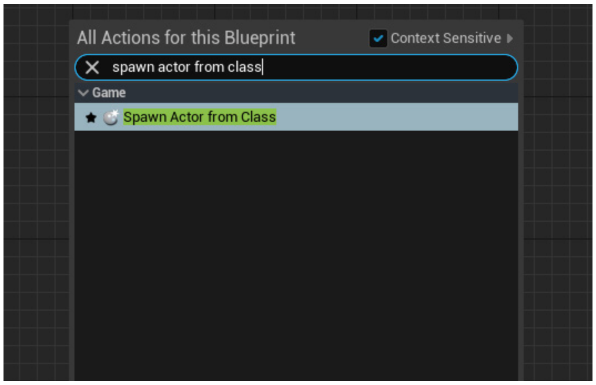


***Figure 8.40**: "Spawn Actor From Class" blueprint command*

The engine will automatically add the **`Spawn Actor`** blueprint block to the event graph; then, select **`BP_Ball_02`** from the **`Class`** drop-down menu and the blueprint block will automatically change and show your public variables **`Direction`** and **`Speed`**, as shown in *Figure 8.41*. Now, we need couple of input values in this block, like **`Direction`** and **`Speed`** variables, which are references to the public variables we made in the **`Ball_02`** blueprint object:

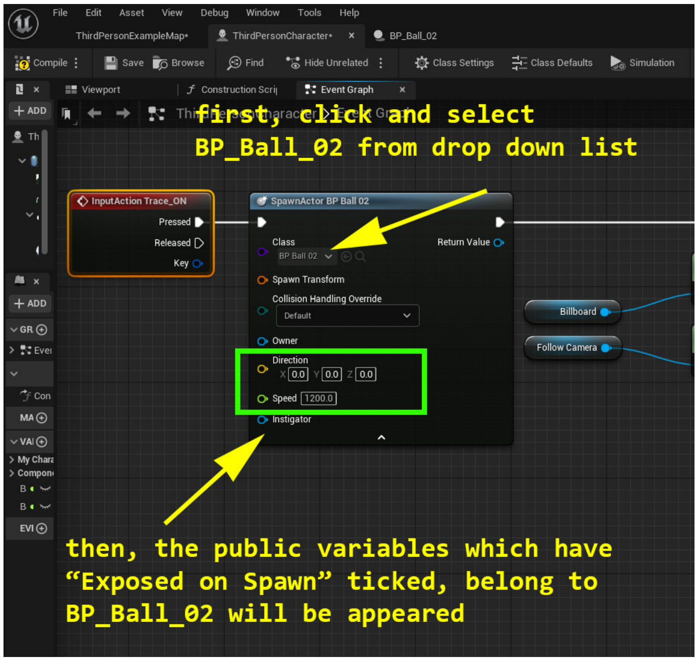


***Figure 8.41**: "Spawn Actor From Class" which the "Class" pin is set to "BP_Ball_01"*

To make a value for **`Spawn Transform`**, we use same data we used for line trace, by using **`Make Transform`** blueprint block. This block can also change the scale of the spawn object. For **`Direction`**, we can use follow camera component's forward vector, and it will give the exact directions, like a camera, to the spawned object; finally, for the speed, let's experience with 750. The final code should look like *Figure 8.42*:

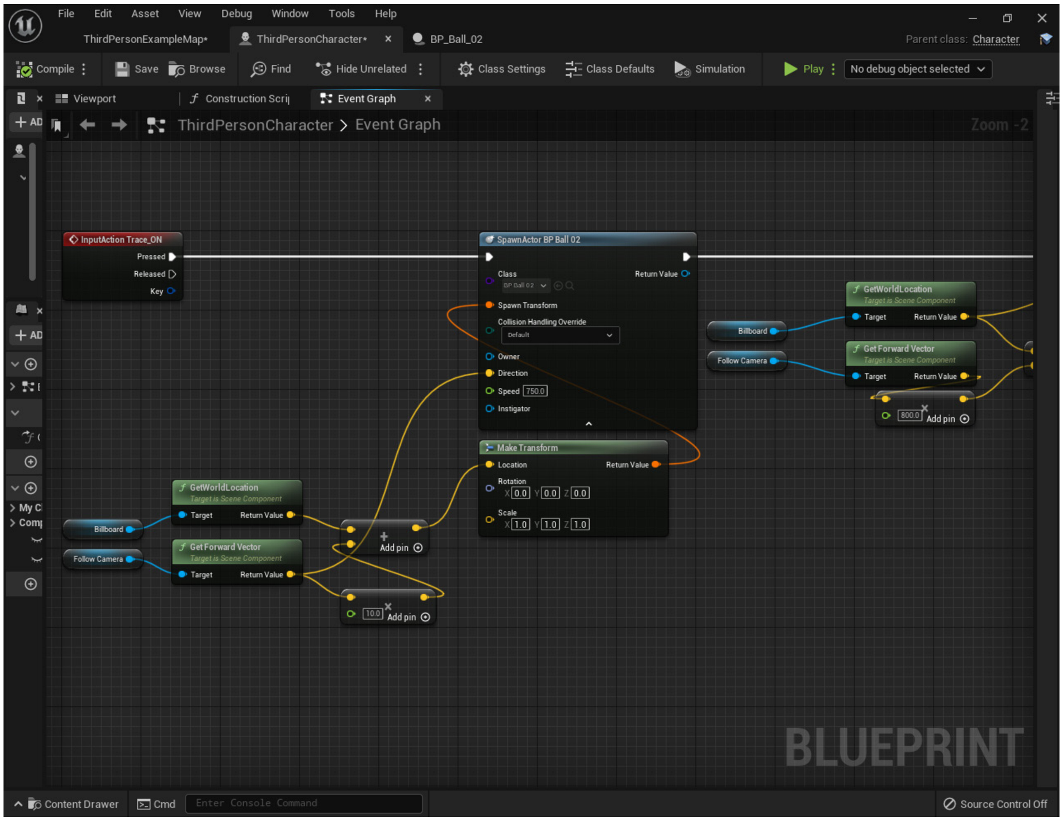


***Figure 8.42**: This code represents shooting functionality by using key event, camera and billboard components. The code will spawn actors from the blueprint class BP_Ball_02*

Now, run the code and press *E* in a different camera direction. Voila! The ball object is shooting in the camera direction on each key press, as shown in *Figure 8.43*. Now, press *Esc*, change **`Speed`** to **`200`** and check the results. As you see, the impulse force has decreased slightly on the shooting object:

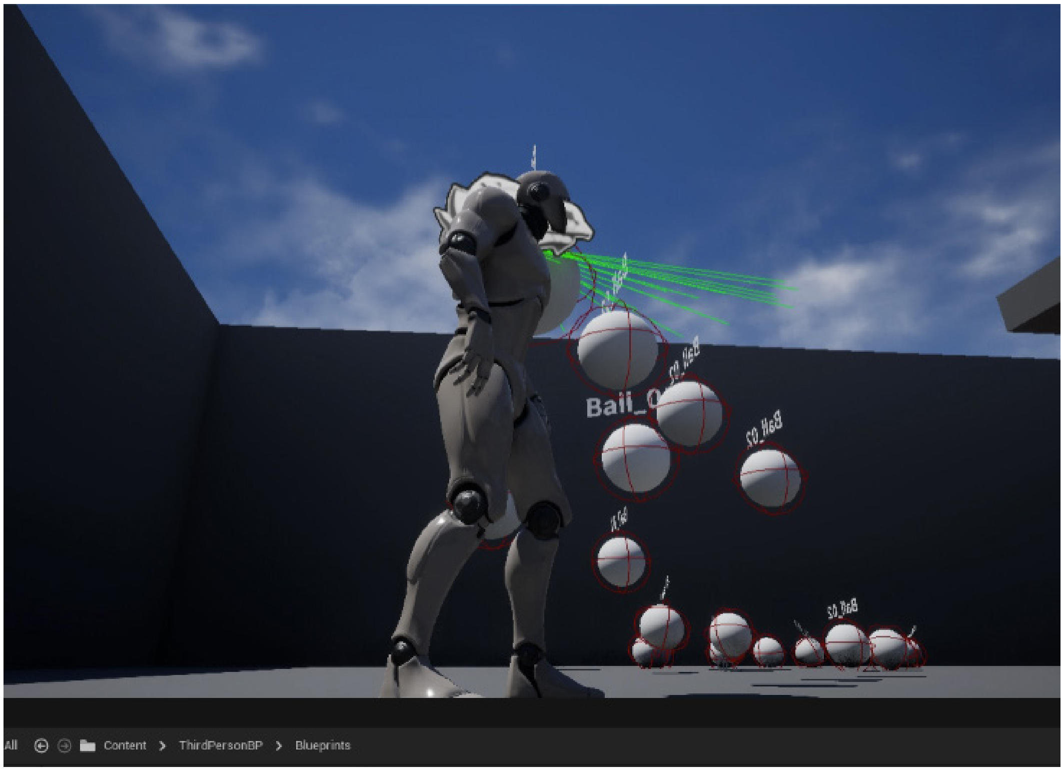


***Figure 8.43**: Shooting BP_Ball_02 from third person character with speed set to 750*

Make physical behavior as material

Unreal Engine also has an advanced system to simulate physics through materials. Right-click on content browser, click on **`Physics`** in the menu, and then select **`Physical Material`** from the list. The engine will show the **`Pick Physical Material Class`** windows; choose the **`Physical Material`** class in there, and then click on the **`Select`** button, as shown in *Figure 8.44*. Rename the new physical material as **`PM_01`**:

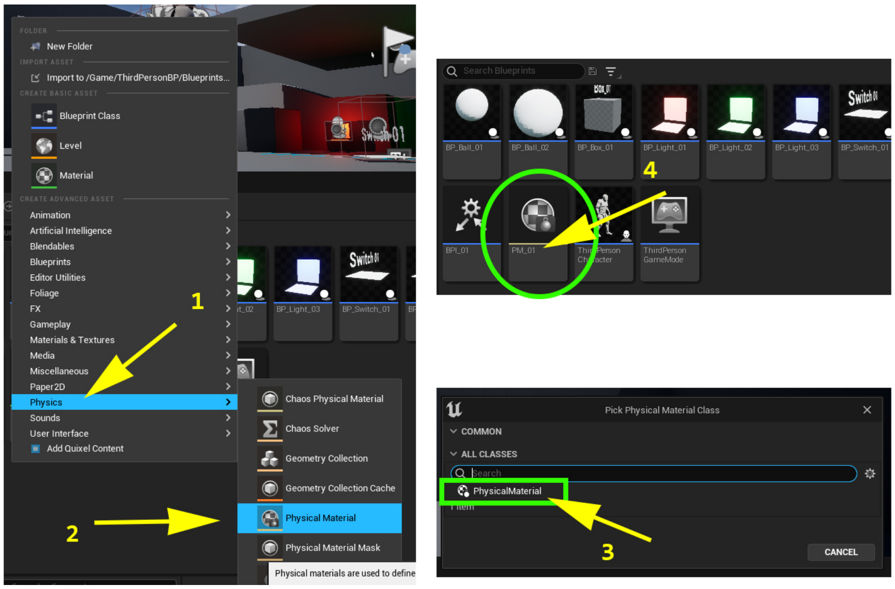


Figure 8.44: *Right-click in the content browser (1) and from Physics in the list (2) select Physical Material. Then (3) click on "PhysicalMaterial" from "Pick Physical Material Class" windows and (4) rename the new physical material to "PM_01"*

Now, we need a material to handle our physical material. Make a simple material called **`MT_01`** and assign the **`PM_01`** physical material that we made to it at the **`PHYSICAL MATERIAL`** section in its event graph, as shown in *Figure 8.45*:

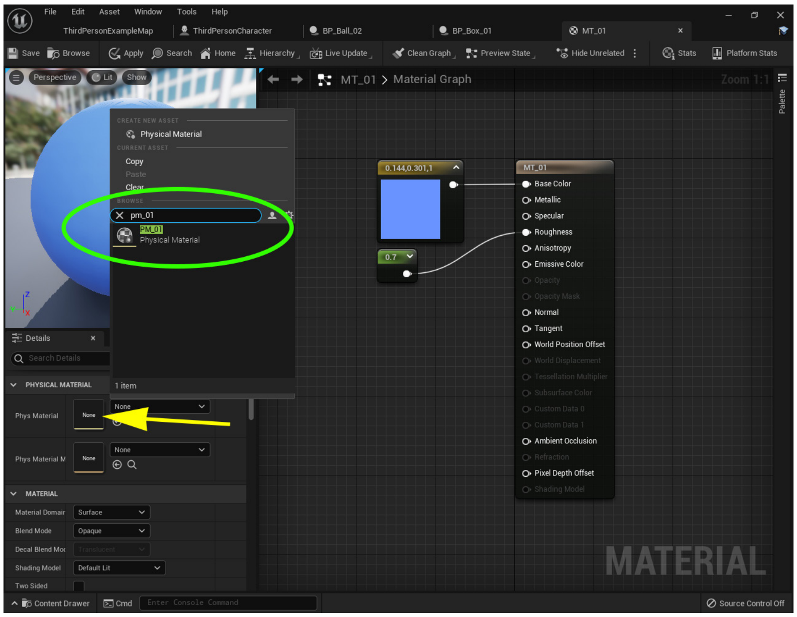
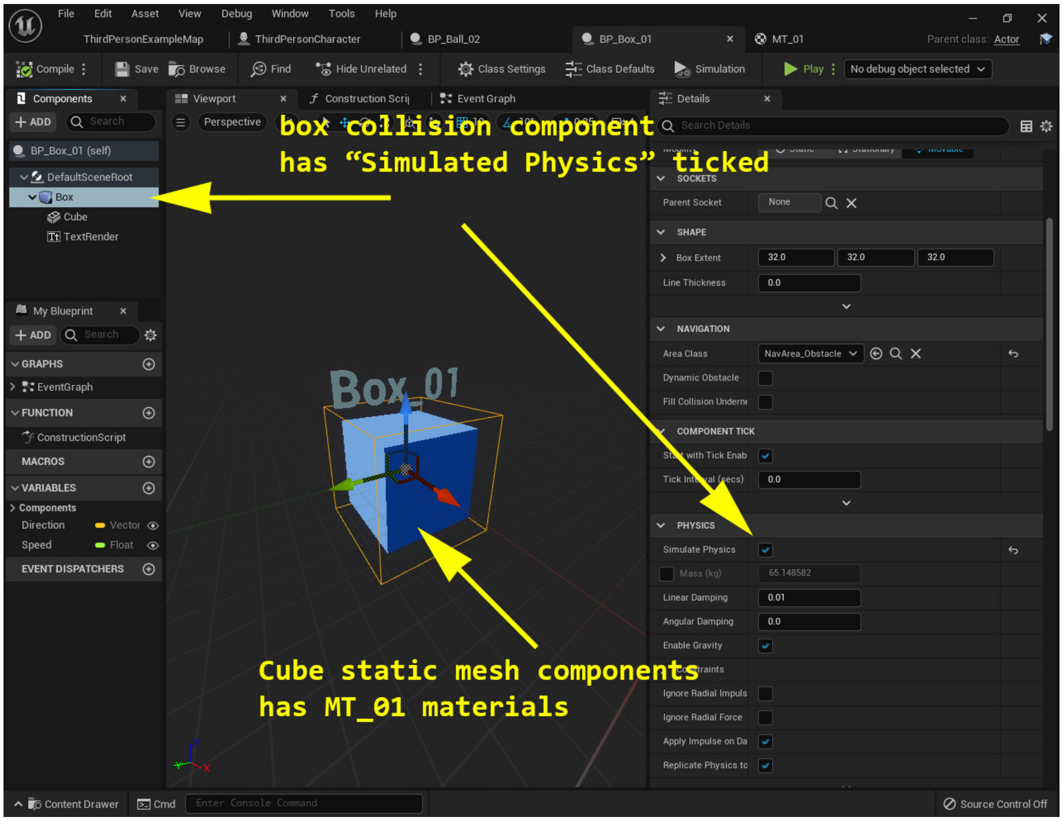


Figure 8.45: *Add physical material from materials editor in Unreal Engine 5*

Now, we need an actor to test this material. Make a new actor called **`BP_Box_01`** with a box collision component that has **`Simulate Physics`** ticked in its details panel, a **`Cube`** static mesh component and a Text render component, as shown in *Figure 8.46*. Assign the **`MT_01`** material to cube static mesh component:



Parent Socket	None
Box Extent	32.0 32.0 32.0
Line Thickness	0.0
Area Class	NavArea_Obstacle
Dynamic Obstacle	☐
Fill Collision Underne	☐
Start with Tick Enab	✓
Tick Interval (secs)	0.0
Simulate Physics	✓
Mass (kg)	65.148582
Linear Damping	0.01
Angular Damping	0.0
Enable Gravity	✓
Ignore Radial Impuls	☐
Ignore Radial Force	☐
Apply Impulse on Da	✓
Replicate Physics to	✓

***Figure 8.46**: BP_Box_01 blueprint actor*

Now, place an instance of this actor on stage. Then, press play and try to push the cube with the third person character, as shown in *Figure 8.47*. It moves, but not easy, is it?

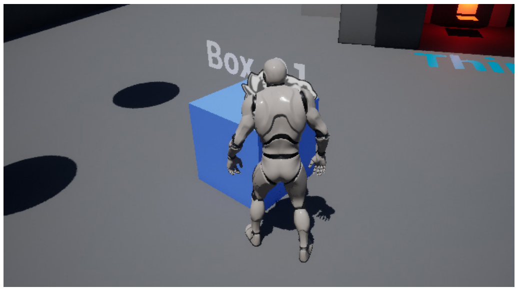
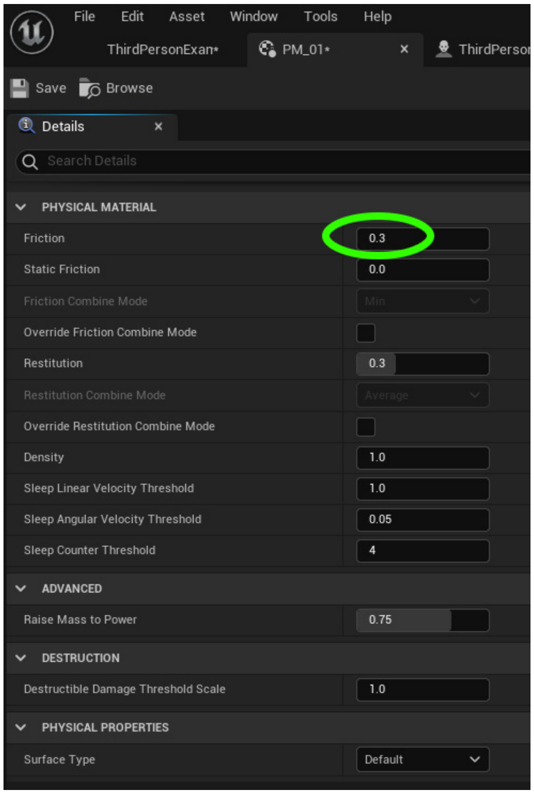


***Figure 8.47**: Add BP_Box_01 blueprint actor to stage and try to push it with the third person character*

Now, press *Esc* to exit the game, and then open **`PM_01`** by double-clicking on it in the content browser. Physical material, as shown in *Figure 8.48*, has a number of parameters that directly change the behavior of the static mesh object using this material. For now, change the **`Friction`** value from 0.3 to 0 and click on **`Save`**:



PHYSICAL MATERIAL

Friction	0.3
Static Friction	0.0
Friction Combine Mode	Min
Override Friction Combine Mode	☐
Restitution	0.3
Restitution Combine Mode	Average
Override Restitution Combine Mode	☐
Density	1.0
Sleep Linear Velocity Threshold	1.0
Sleep Angular Velocity Threshold	0.05
Sleep Counter Threshold	4

ADVANCED

Raise Mass to Power	0.75

DESTRUCTION

Destructible Damage Threshold Scale	1.0

PHYSICAL PROPERTIES

Surface Type	Default

***Figure 8.48**: Physical material parameters*

Now, run the game again and try to push the cube again. Voila! It sounds like the cube behaves like a big *ice cube* and moves with a simple push by the player character:

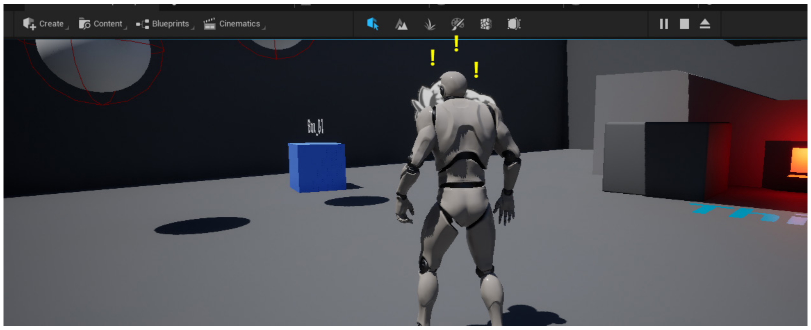


***Figure 8.49**: Looks like the BP_Box_01 behaves like a big block of ice after get pushed by third person character*

What you have on screen now is a physical material simulation, which runs by engine in real time. There are a number of situations you can simulate through physical materials in the engine; while that is out of scope of this section, keep in mind that this is one of the most powerful tools in the engine to simulate real physics over each static mesh object when the game is running.

Conclusion

In this chapter, we learned how to use event handling by making a number of game assets and connecting them to each other as a simple switch button for a source light. Event handling is an art of developing applications with Unreal Engine, so as much as you can experience and learn in this area, the quality of your final work will show bright features. At the same time, the engine has implemented interface support for blueprint programming. The interface, as we learn, works like a common *gateway* to send and receive events between objects in an extremely simple way.

At the end, we learned how to shoot physical objects based on physical impulse simulation and change the physical features of each material to make them behave like real-world objects.

For making much in depth custom functionalities for actors on the scene, developers need to use and process different data types at the same time. To do that, developers mostly use groups of variables which are organized in data containers. These data containers simplify the process and organized the code which is very useful in order to develop complex functionalities.

The Unreal Engine 5 has a complete support for a majority of data types that can be used to control game flow, animation, and game mechanics. Learning that is essential knowledge for each game programmer.

Points to remember

- When you make a new interface and assign it to your objects, it is better to save the project and open it again. The reason for this is technically complicated, but in simple words, the engine needs to grab the new interface code and compile it a little bit different from compiling normal blueprints. Sometimes, this process is not fully running after users make new interface blueprint in editor and then assign it to objects. So, after saving and restarting, the engine will fully compile all new game objects again, which, in turn, will mean that the interface will be applied and ready to use.
- Event and interface in Unreal Engine can accept variables to send and receive data.

- When Simulate Physics is ticked on an object, remember that you can tick *mass* and assign values as kilograms to your object and simulate physics based on it. This property is compatible with physical material features belonging to object.

Multiple choice questions

1. **For what do we use events in Unreal Engine 5?**
    a. To develop communication between blueprint game objects in the scene
    b. To handle collision and address user input
    c. To implement interface function support on blueprint objects
    d. Options a, b and c are correct

Answer

1. **d**

Questions

1. How can you send and receive a variable with an event?
2. How can you receive values from blueprint interface?
3. What is the difference between `LineTraceByChannel` and `LineTracebyObject` in blueprint programming?
4. Can you force engine's line trace to bypass an object in the scene? How?

Key terms

- **Blueprint interface -** Blueprint Interface is a collection of one or more functions, without implementations (name only) that can be added to other Blueprints for send and receive data.
- **Blueprint event -** Executive nodes in blueprint that are called during runtime for execution of an individual network of procedures, are known as blueprint event.
- **Line trace –** Traces functions, "shoot" out an invisible ray/line which can detect geometry between two points. In case that geometry is hit by this ray/line, the trace function (line trace) will return data related to hit point.

- **Cast -** Casting is a common programming method where you take a reference to an object and try to compare with or convert into a different class.
- **Blueprint collision –** Every object that can collide can use procedures(events/functions) to detect and manage its behavior based on collide type.
- **Physical material –** They are special type of materials used to define the behavior, respond and possible actions of a physical object when interacting dynamically with the world based on physical parameters.
- **Impulse -** An impulse is an "instantaneous physical force" which usually applied once to the object.
- **Simulate physics –** To activate physical behaviors on an object, find and check "Simulate Physics" property which is mostly located on detail panel of each actor blueprint.
- **Light source -** Unreal Engine has four default light types: Point, Directional, Spot and Sky. Directional lights are used as primary outdoor light or any light that needs to appear as extreme or located very far distances. Point lights are like classic "light bulb", emitting light in all directions from a single point. Spot lights emit light from a single point, but have their light limited by a set of cones. Sky lights capture the background of the game scene and apply it as lighting to meshes located in the scene.

CHAPTER 9

Data Processing (Enum, Struct, Map, Data Tables) and Animation Instances

There are several complicated types of variables that aren't used as much as the simpler ones but are arguably more important, as they are generally used for creating larger, core systems that the simpler variables are used within. This chapter will also cover animation instances (changing animations using data within blueprint).

Structure

In this chapter, we will discuss the following topics:

- Object based variables
- Enumerators (Enums)
  - Byte
  - Enumerator editor
- Structures (structs)
  - Nested structs
  - Structure editor
- Arrays
- Maps

- Data tables
    - Data table editor
    - CSV/JSON export
    - Data table usage in blueprint
- Animation instances
- Multi-variable type operators

Objectives

After studying this chapter, you should know of some of the more advanced variable types, their uses, how to set them up for use, some of the blueprint nodes that utilize them, the basic concept of animation instances, how to use them to animate skeletal meshes via blueprint, and how to use multi-variable math operators.

Object-based variables

Enumerators, structures, and data tables are predefined; although they are used within blueprints, their traits are defined outside of the blueprint editor within an object that stores this information:

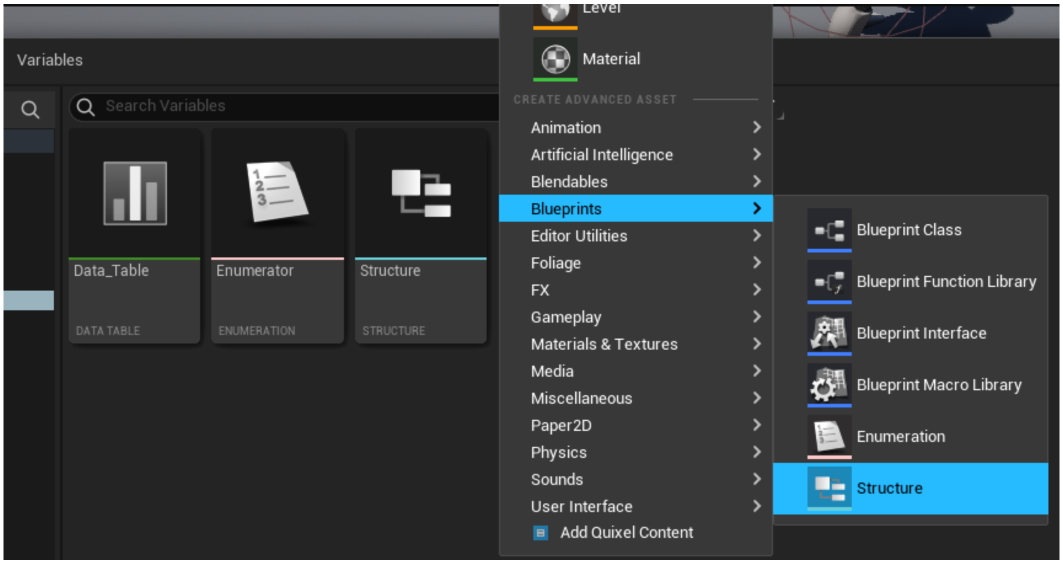
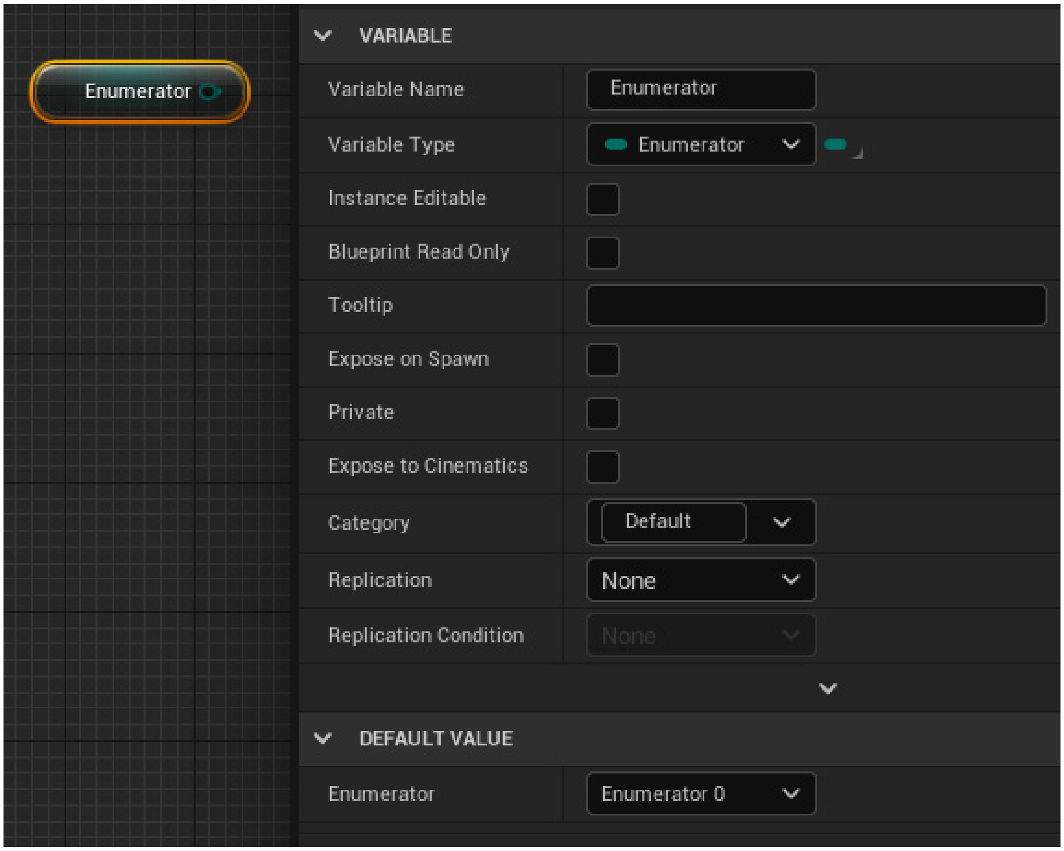


***Figure 9.1**: Object Based Variables in Content Browser*

Each of these variables have an editor specific to them, which is accessed by opening them through the content browser. Within these editors, the associated variable can be modified. Creating these assets is done by right-clicking in the content browser; structure and enumeration are found under the **`Blueprints`** category, and **`Data Tables`** are found under **`Miscellaneous`**.

Enumerators (Enums)

Enumerator variables contain a list of named values from a predefined list of possible values, which are defined within an enumerator object in the project files. :



VARIABLE

Variable Name	Enumerator
Variable Type	Enumerator
Instance Editable	☐
Blueprint Read Only	☐
Tooltip	
Expose on Spawn	☐
Private	☐
Expose to Cinematics	☐
Category	Default
Replication	None
Replication Condition	None

DEFAULT VALUE

Enumerator	Enumerator 0

***Figure 9.2**: Enumerator Variable*

Enumerators are used to hold a value from a set list of potential values, so they are useful for defining states, usually within (or accessed by) an animation blueprint as shown in image below.

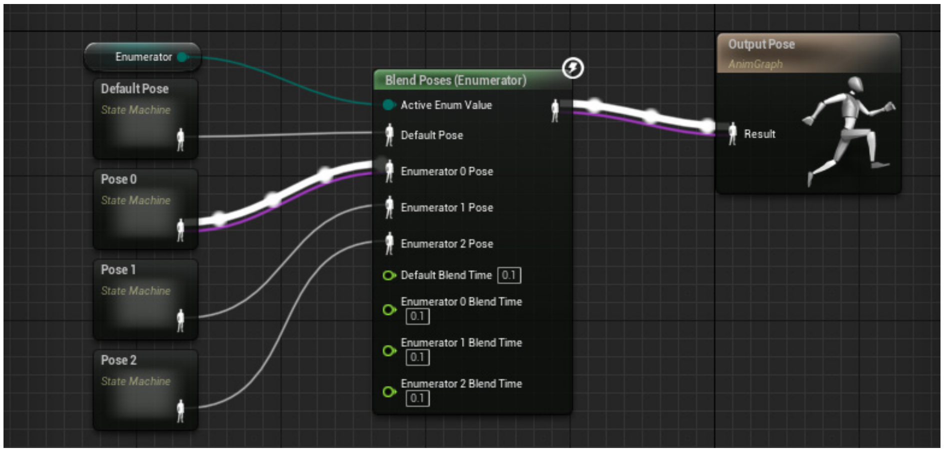
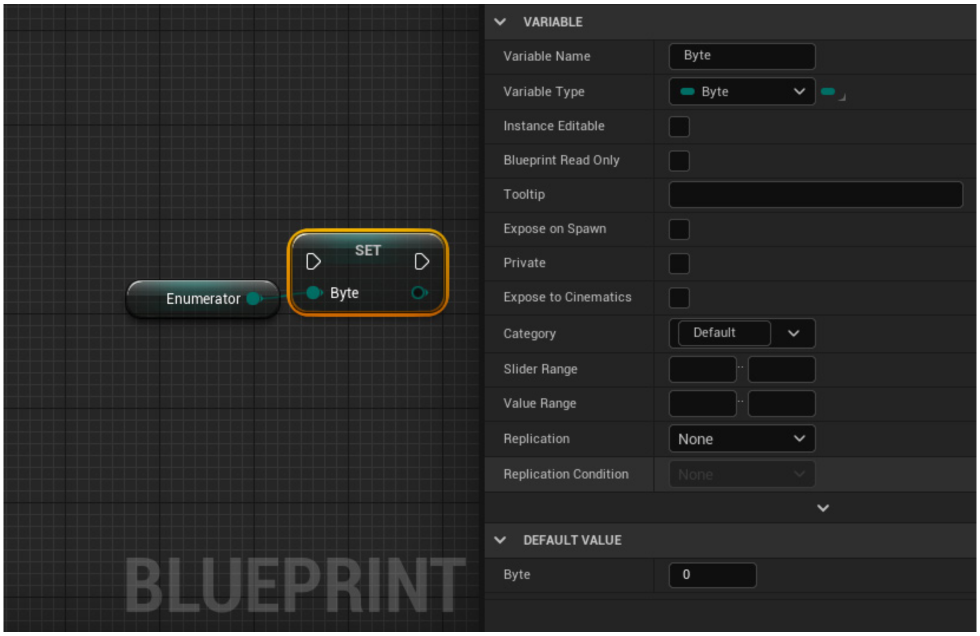


Figure 9.3: *Enumerator in Animation Graph*

Bytes

Enumerators can also be converted to and from **`Byte`** variables, which allows reference to enumerator values via the index value (place in the list of possible enumerator values):



VARIABLE

Variable Name	Byte
Variable Type	Byte
Instance Editable	
Blueprint Read Only	
Tooltip	
Expose on Spawn	
Private	
Expose to Cinematics	
Category	Default
Slider Range	
Value Range	
Replication	None
Replication Condition	None

DEFAULT VALUE

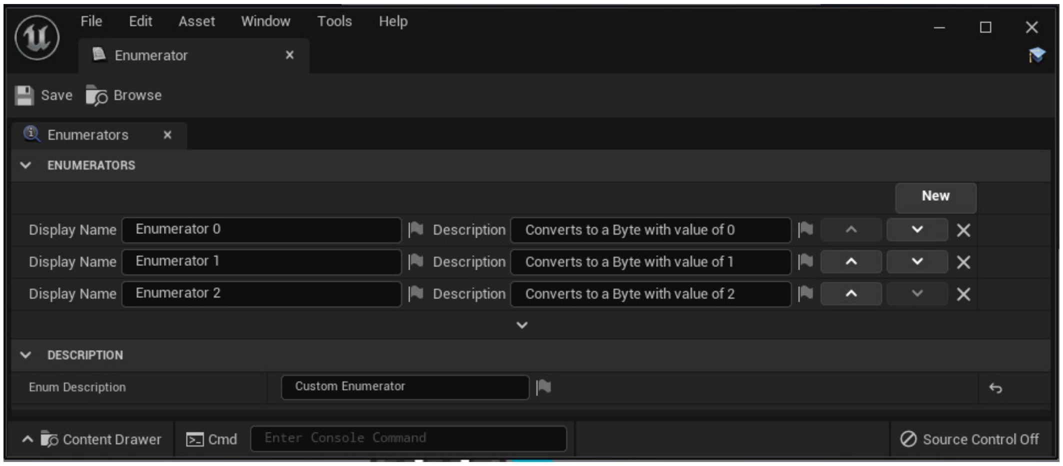
Byte	0

Figure 9.4: *Byte variable*

Bytes are similar to integers, but they can only contain whole numbers up to 255, making them perfect for usage alongside lists like enumerators because you can't have negative or partial entries.

Enumerator editor

Enumerators are modified via an Enumerator specific editor, where entries can be added, modified and defined:



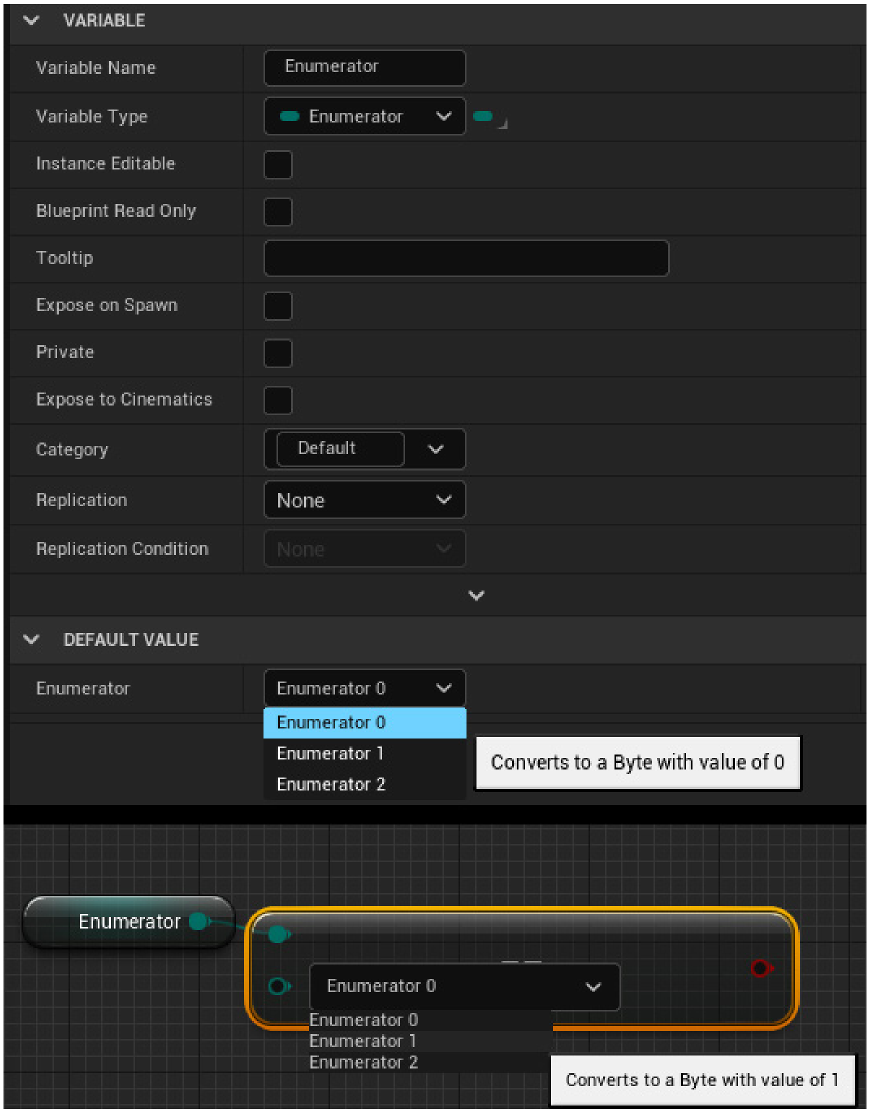
Display Name	Enumerator 0	Description	Converts to a Byte with value of 0
Display Name	Enumerator 1	Description	Converts to a Byte with value of 1
Display Name	Enumerator 2	Description	Converts to a Byte with value of 2

Enum Description: Custom Enumerator

***Figure 9.5**: Enumerator Variable Editor*

Each entry has a couple of text boxes; the one labeled **`Display Name`** allows you to set the name displayed for that particular entry when using the enumerator, and the **`Description`** box text will be displayed when hovering over that entry in a drop-down menu (**`Setting Default value`** or comparing values, as shown in *Figure*

9.6). Entries can also be rearranged, but be aware that changing the order of entries may cause issues in logic utilizing the enumerator:



VARIABLE

Variable Name	Enumerator
Variable Type	Enumerator
Instance Editable	
Blueprint Read Only	
Tooltip	
Expose on Spawn	
Private	
Expose to Cinematics	
Category	Default
Replication	None
Replication Condition	None

DEFAULT VALUE

Enumerator	Enumerator 0

Enumerator 0
Enumerator 1
Enumerator 2

Converts to a Byte with value of 0

Enumerator

Enumerator 0

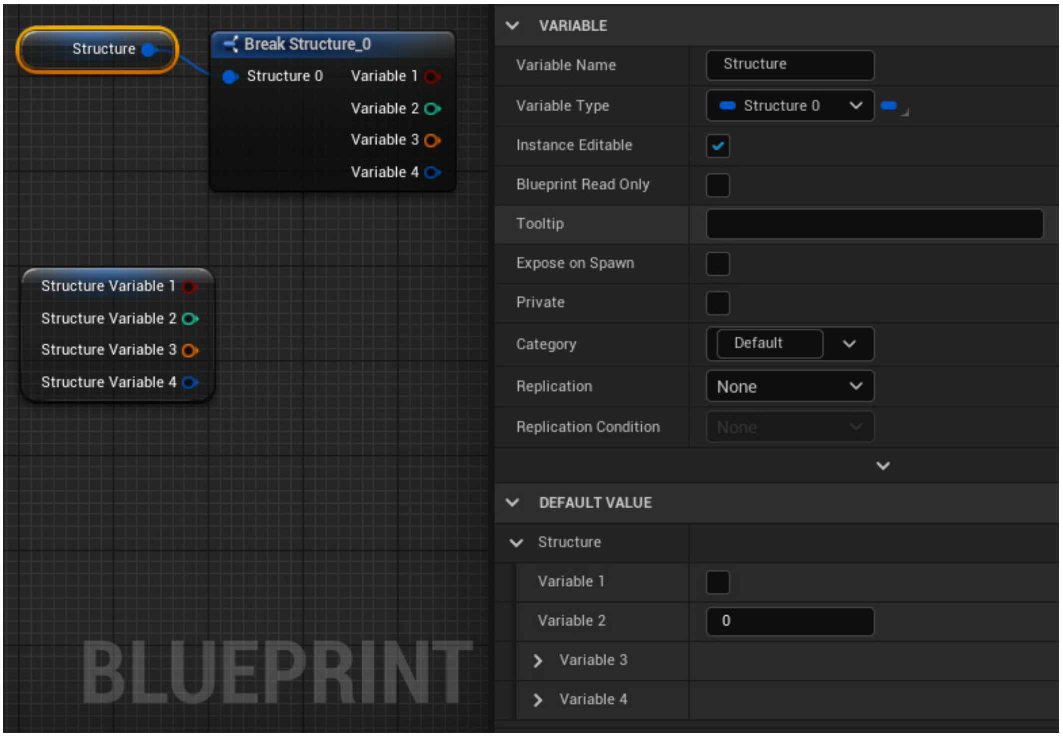
Enumerator 0
Enumerator 1
Enumerator 2

Converts to a Byte with value of 1

Figure 9.6: *Enumerator Descriptions*

Structures (structs)

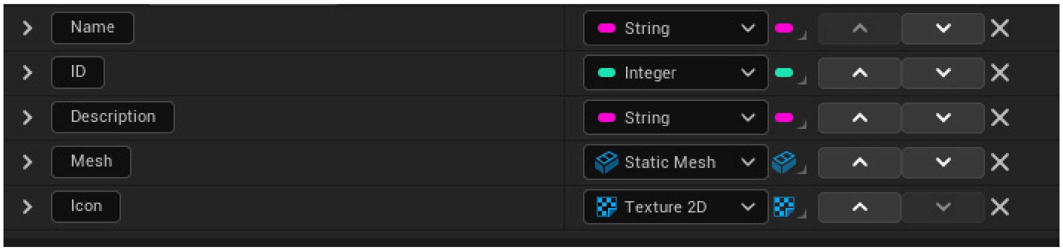
Structures are collections of variables, which can include other structs. This means they can be used to neatly contain a large amount of varied information.



Variable Name	Structure
Variable Type	Structure 0
Instance Editable	✓
Blueprint Read Only	
Tooltip	
Expose on Spawn	
Private	
Category	Default
Replication	None
Replication Condition	None
Structure	
Variable 1	
Variable 2	0
Variable 3	
Variable 4	

Figure 9.7: *Structure variable*

Structures are used for grouping together related data neatly into a single variable; for example, you could contain information about an item that can be placed within some type of inventory system by the player, likely containing a static mesh variable to be used when the item is placed in the world, an icon to display the item within the player's inventory, some type of item ID (**`Integer`**) for fast reference on what specific item it is, a name, and possibly also a description:



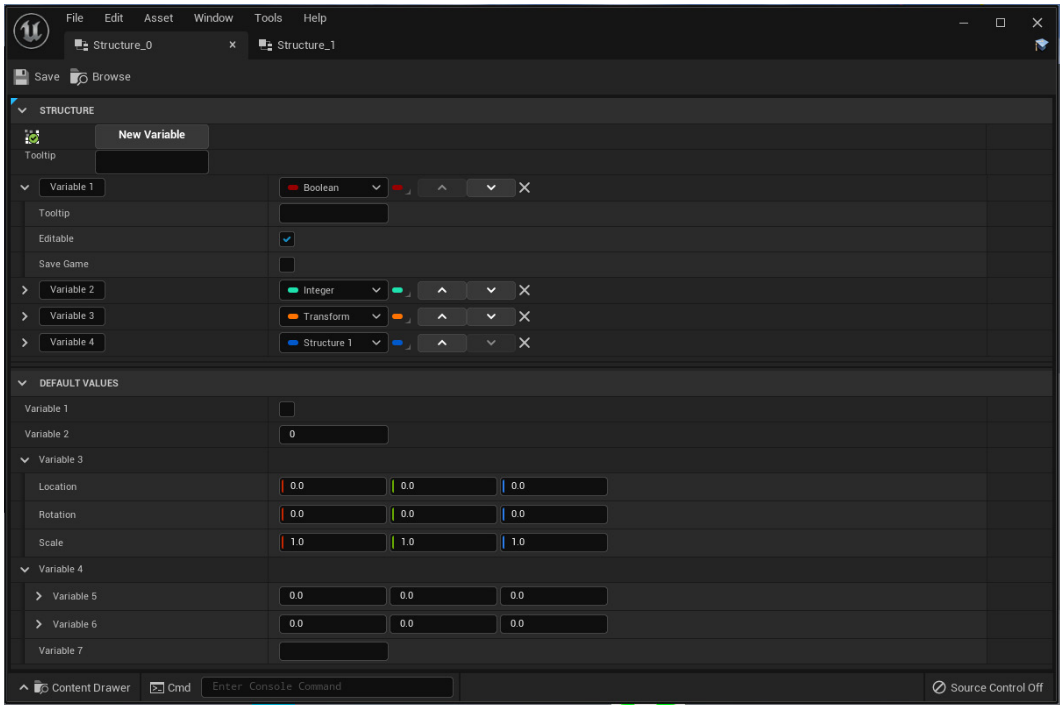
Name	String
ID	Integer
Description	String
Mesh	Static Mesh
Icon	Texture 2D

Figure 9.8: *Structure Example*

When using structs in blueprints, using pin splitting (as covered in *Chapter 7, Simple Data*) or break nodes (as shown in *Figure 9.7*) is a necessity to access the variables within the struct.

Structure editor

Structures are modified via a structure specific editor, where variables can be added, modified, and defined in the top half of the editor, and the default values for the variables used within the struct can be set in the bottom half of the editor:



STRUCTURE

Variable 1	Boolean
Tooltip	
Editable	✓
Save Game	
Variable 2	Integer
Variable 3	Transform
Variable 4	Structure 1

DEFAULT VALUES

Variable 1			
Variable 2	0		
Variable 3			
Location	0.0	0.0	0.0
Rotation	0.0	0.0	0.0
Scale	1.0	1.0	1.0
Variable 4			
Variable 5	0.0	0.0	0.0
Variable 6	0.0	0.0	0.0
Variable 7			

***Figure 9.9**: Structure Editor*

Each entry in the top section of the editor contains a text box for naming the variable, a drop-down menu for selecting the variable type, a few buttons for reordering the entries, and a cross for removing the entry.

Each entry can also be expanded by clicking on the arrow to the left of its name (as seen in *Figure 9.9*), which allows access to the tooltip for that entry. A Boolean to set whether that entry's variable can be modified on an instance of a blueprint using the struct, and a Boolean to set whether the variable will be serialized if the struct is used in a save file.

Nested structs

Structures can contain structures within themselves, although not a struct of the same type, so no infinite loops:

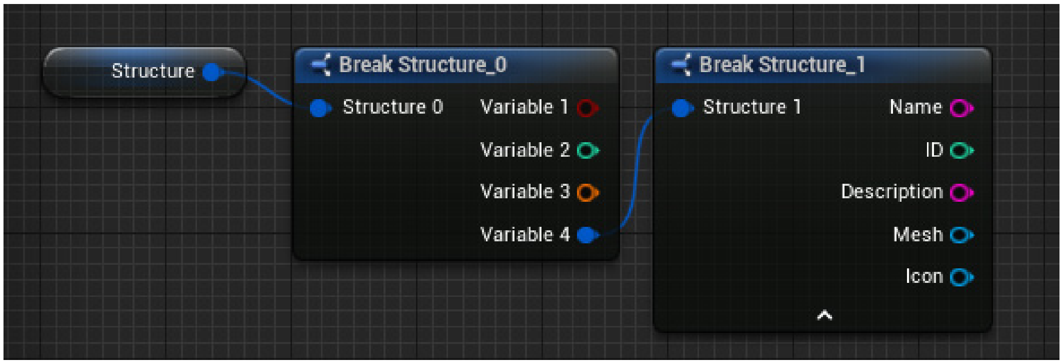
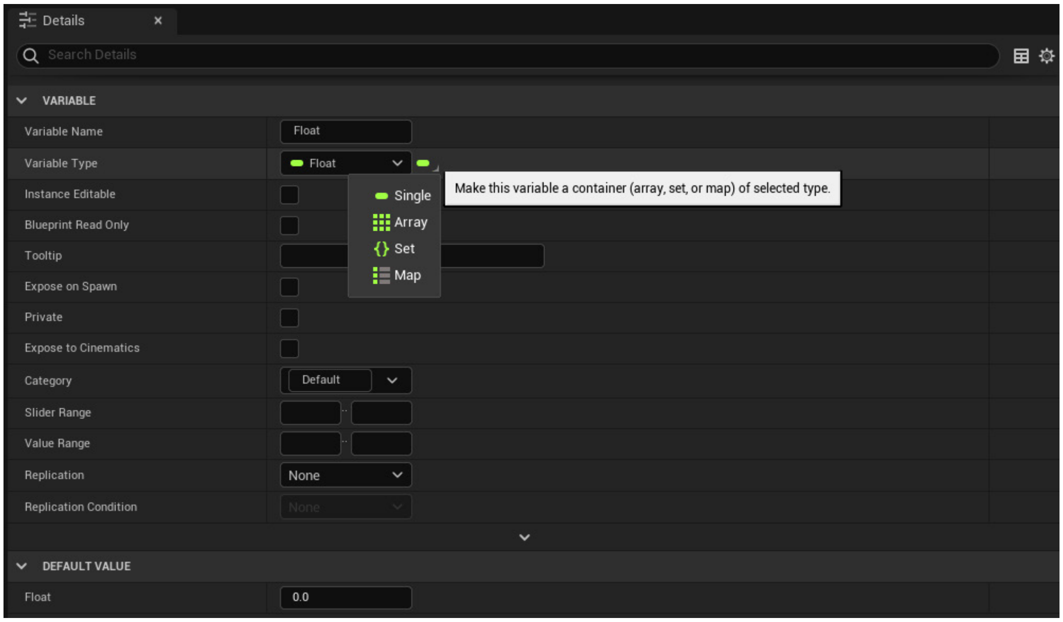


***Figure 9.10**: Nested Struct*

This allows for structs to contain subcategories within themselves, which is extremely useful for defining complex objects that require many variables.

Arrays

A list of values of a specific variable type is called an **array**. Arrays can be made by selecting an existing variable and changing it from a single value to an array, as shown in *Figure 9.11*:



VARIABLE

Variable Name	Float
Variable Type	Float
Instance Editable	
Blueprint Read Only	
Tooltip	
Expose on Spawn	
Private	
Expose to Cinematics	
Category	Default
Slider Range	
Value Range	
Replication	None
Replication Condition	None

Make this variable a container (array, set, or map) of selected type.

DEFAULT VALUE

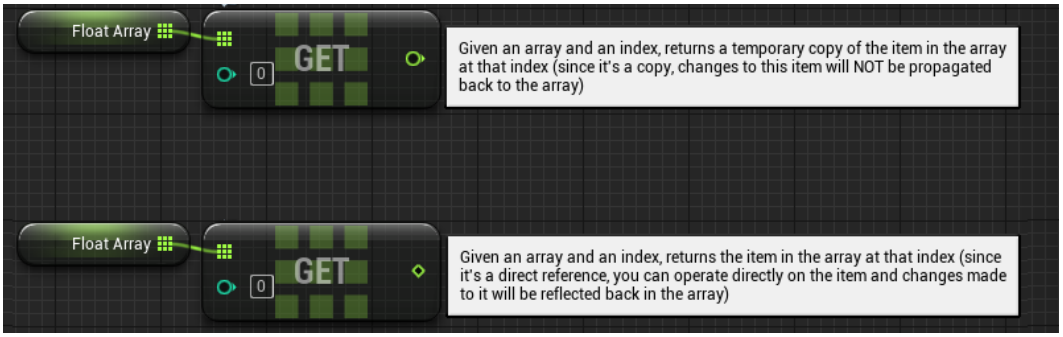
Float	0.0

***Figure 9.11**: Change container type*

Arrays have nodes specifically for them, including ones that allow getting values from a specific entry in an array, removing or altering entries, and more.

Get

In order to get a value for a specific entry in an array, one of two variations of the **`Get`** node will need to be used:



Given an array and an index, returns a temporary copy of the item in the array at that index (since it's a copy, changes to this item will NOT be propagated back to the array)

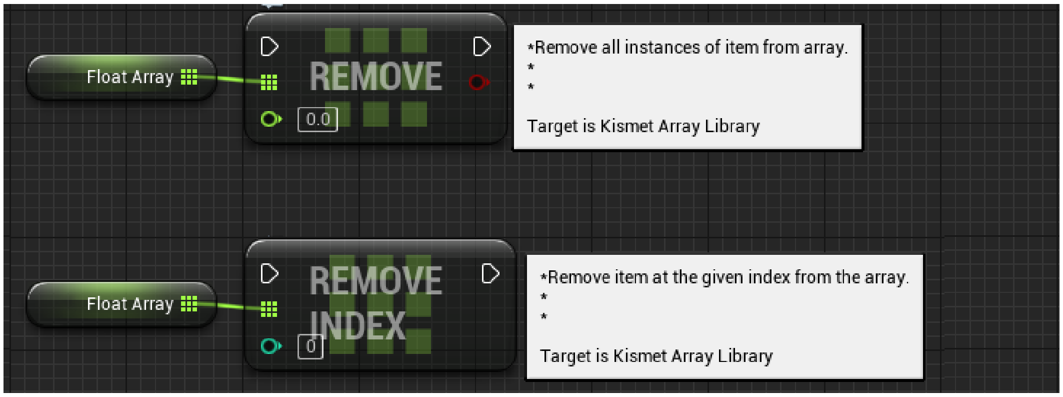
Given an array and an index, returns the item in the array at that index (since it's a direct reference, you can operate directly on the item and changes made to it will be reflected back in the array)

Figure 9.12*: Get Array Value*

When creating a **`Get`** node, you will have a choice between a node that gets a copy of the value and a node that gets a reference to the value. The key difference between the two is that the reference allows you to change the output value in the array by altering the output value, whereas the copy only gives you the value with no capacity for altering it.

Remove

In order to remove one or more values from an array, one of two remove nodes will need to be used. The **`Remove`** node will take a value and remove all entries matching that value from the array, while the **`Remove Index`** node will remove a specific entry:



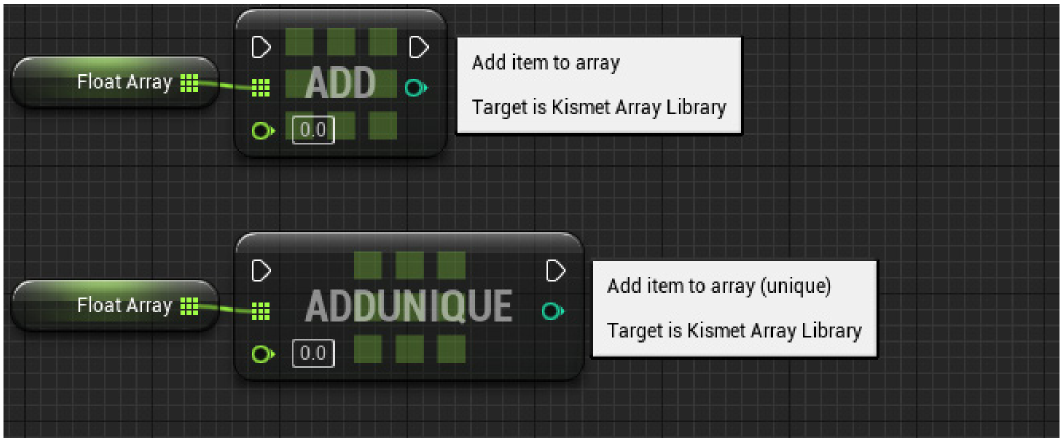
*Remove all instances of item from array.
*
*
Target is Kismet Array Library

*Remove item at the given index from the array.
*
*
Target is Kismet Array Library

Figure 9.13*: Remove from Array nodes*

Add

In order to add a value to an array, one of two **`Add`** nodes (or the **`Set Array Element`** node) will need to be used. The **`Add`** node will add any value to an existing array as the last entry, whereas **`AddUnique`** will only add values that are not already present in the array:



Add item to array

Target is Kismet Array Library

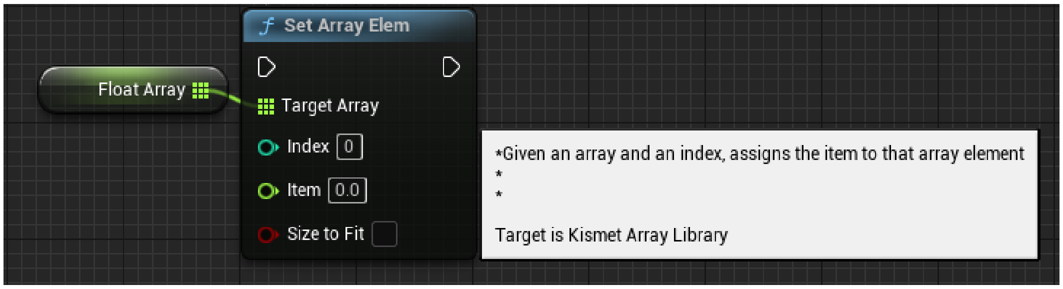
Add item to array (unique)

Target is Kismet Array Library

***Figure 9.14**: Add to Array nodes*

Set Array Element

This node allows you to either change an existing entry in an array or add one if the input index is not a currently existing entry in the array, and **`Size to Fit`** is set to true:



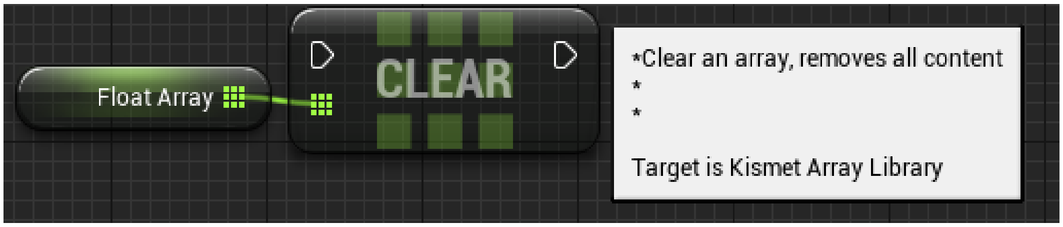
*Given an array and an index, assigns the item to that array element
*
*

Target is Kismet Array Library

***Figure 9.15**: Set Array Element nodes*

Clear

This node allows you to completely empty an array by removing all entries:



*Clear an array, removes all content
*
*

Target is Kismet Array Library

***Figure 9.16**: Clear Array node*

Length

This node allows you to get the number of entries in the provided array:

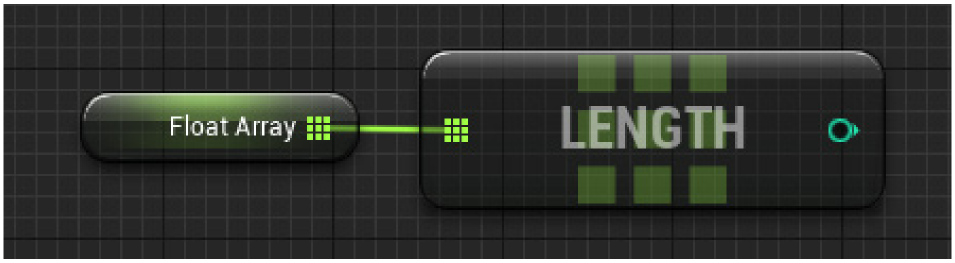
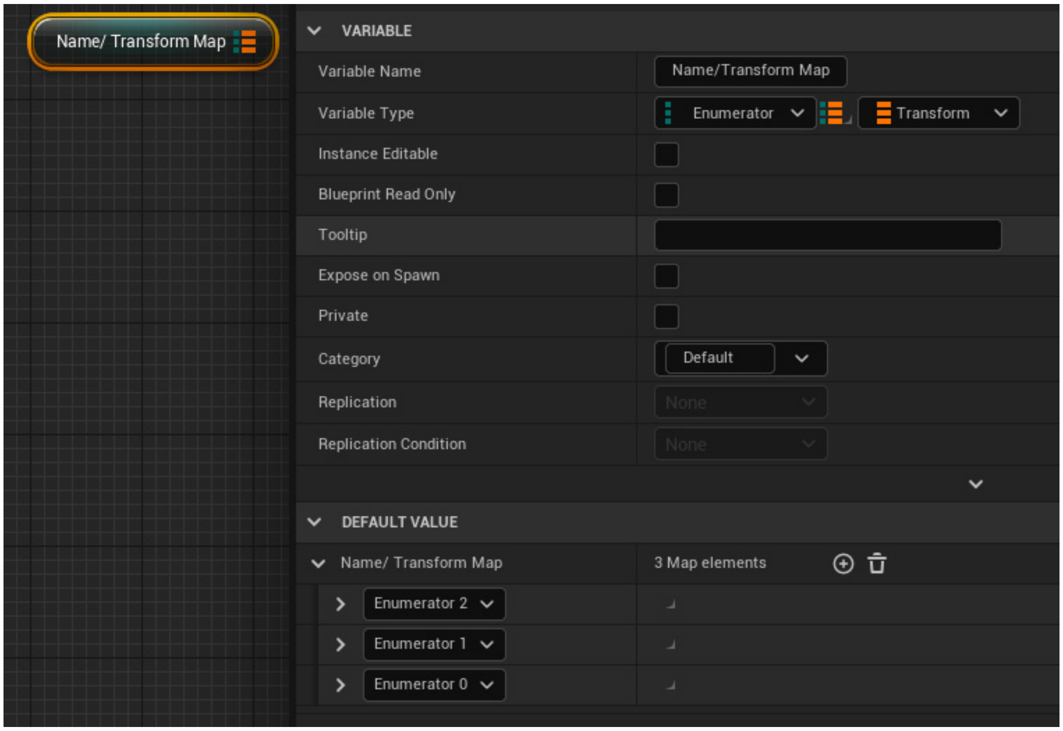


***Figure 9.17**: Length node*

Maps

Maps are similar to arrays in that they are a list, but instead of it being a number of entries with an index value corresponding to their place in the list, a map consists of a number of variable values and a corresponding value of an entirely different variable type. The left variable type in a **`Map`** entry is called a **`Key`**, and the corresponding value on the right is referred to as a **`Value`**, both of which can be seen in *Figure 9.14*. Maps are also created through the same method as arrays, but by choosing the **`Map`** option instead of the **`Array`** option, as seen in *Figure 9.11*:



VARIABLE

Variable Name	Name/Transform Map
Variable Type	Enumerator · Transform
Instance Editable	☐
Blueprint Read Only	☐
Tooltip	
Expose on Spawn	☐
Private	☐
Category	Default
Replication	None
Replication Condition	None

DEFAULT VALUE

Name/ Transform Map	3 Map elements
Enumerator 2	
Enumerator 1	
Enumerator 0	

Figure 9.18: Map Variable

Get keys and values

These nodes will allow you to get all the keys and values used in the map, respectively:

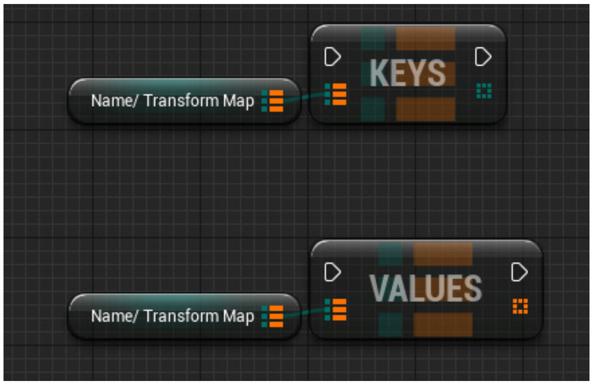


Figure 9.19: Get Keys and Values Nodes

Find

This node will search the provided map for the provided key, and if it finds it, it will return the corresponding value:

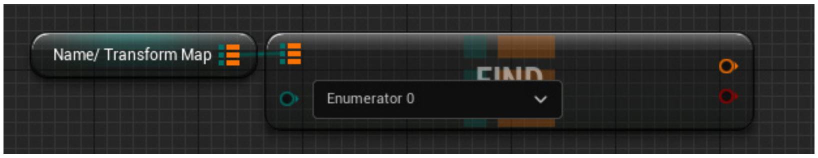


Figure 9.20: *Find Node*

Length

This node will search the provided map for the provided key, and if it finds it, it will return the corresponding value:

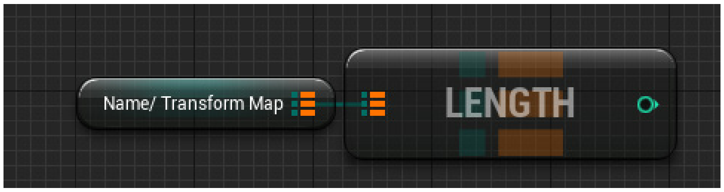


Figure 9.21: *Length Node*

Clear

This node will completely clear the map:

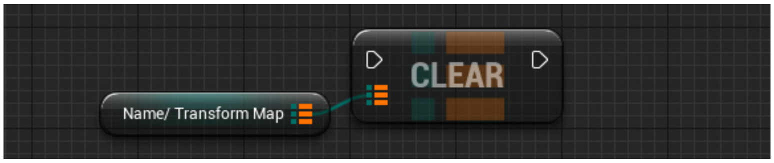


Figure 9.22: *Clear Node*

Contains

This node will return a Boolean stating whether the provided key can be found in the provided map:

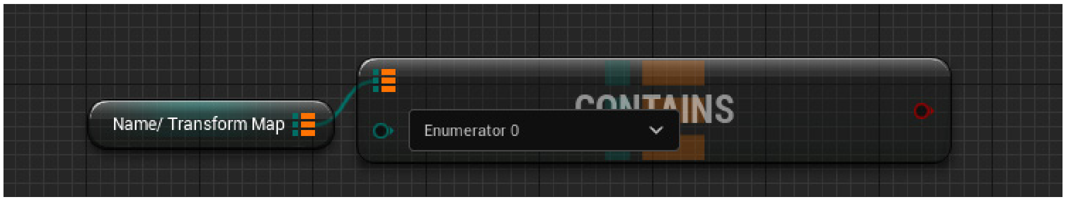


Figure 9.23: *Contains Node*

Is Empty and Is Not Empty

These nodes will return a Boolean stating whether the map contains any entries:

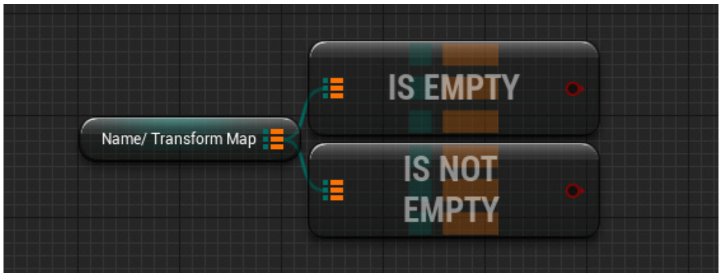
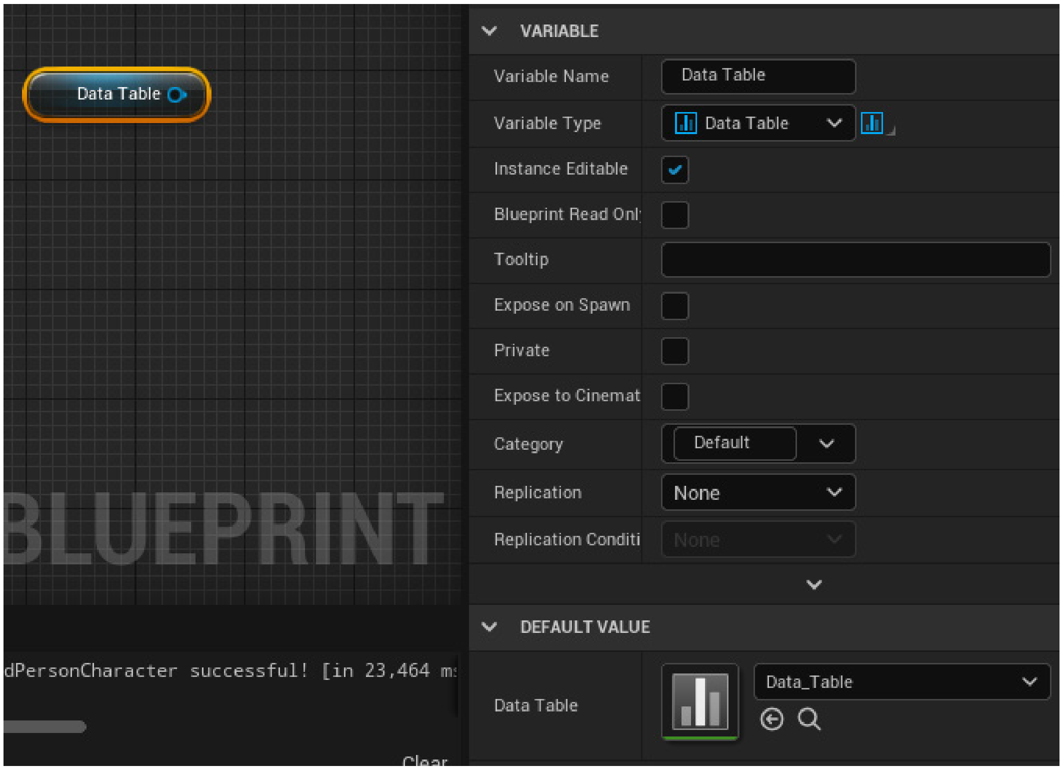


***Figure 9.24**: Is Empty and Is Not Empty Node*

Data tables

Data tables are a predefined list of entries using a structure to define entries. They are used in cases where a project contains a large number of similar objects that are defined by structs. A reference to a data table may be used as a variable within a blueprint:



```
dPersonCharacter successful! [in 23,464 ms
```

VARIABLE

Variable Name	Data Table
Variable Type	Data Table
Instance Editable	✓
Blueprint Read Onl	
Tooltip	
Expose on Spawn	
Private	
Expose to Cinemat	
Category	Default
Replication	None
Replication Conditi	None

DEFAULT VALUE

Data Table	Data_Table

***Figure 9.25**: Data Table*

Data tables are essentially spreadsheets. When creating a **Data Table**, you will be asked to choose which structure the **Data Table** will use for its rows, as seen in *Figure 9.26:*

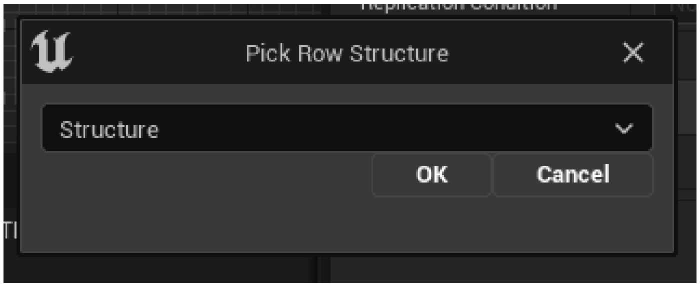
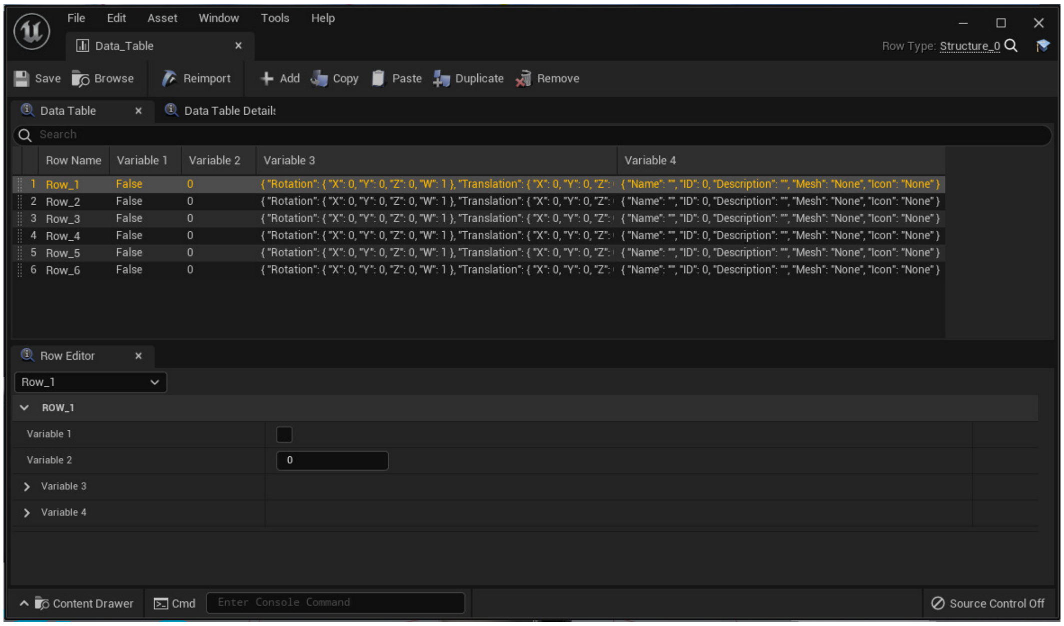


***Figure 9.26:** Data Table Creation*

Data table editor

Like enumerators and structures, data tables have their own editor. Rows can be added, modified, removed, and rearranged as needed. To add a new row, select the **Add** button at the top of the editor, or right-click on an entry and add a new row above or below the selected entry from the context menu. To modify a row, select it from the list of rows in the **Data Table** panel, and then alter its data in the **Row editor** panel, as follows:



Row Type: Structure_0

	Row Name	Variable 1	Variable 2	Variable 3	Variable 4
1	Row_1	False	0	{ "Rotation": { "X": 0, "Y": 0, "Z": 0, "W": 1 }, "Translation": { "X": 0, "Y": 0, "Z":	{ "Name": "", "ID": 0, "Description": "", "Mesh": "None", "Icon": "None" }
2	Row_2	False	0	{ "Rotation": { "X": 0, "Y": 0, "Z": 0, "W": 1 }, "Translation": { "X": 0, "Y": 0, "Z":	{ "Name": "", "ID": 0, "Description": "", "Mesh": "None", "Icon": "None" }
3	Row_3	False	0	{ "Rotation": { "X": 0, "Y": 0, "Z": 0, "W": 1 }, "Translation": { "X": 0, "Y": 0, "Z":	{ "Name": "", "ID": 0, "Description": "", "Mesh": "None", "Icon": "None" }
4	Row_4	False	0	{ "Rotation": { "X": 0, "Y": 0, "Z": 0, "W": 1 }, "Translation": { "X": 0, "Y": 0, "Z":	{ "Name": "", "ID": 0, "Description": "", "Mesh": "None", "Icon": "None" }
5	Row_5	False	0	{ "Rotation": { "X": 0, "Y": 0, "Z": 0, "W": 1 }, "Translation": { "X": 0, "Y": 0, "Z":	{ "Name": "", "ID": 0, "Description": "", "Mesh": "None", "Icon": "None" }
6	Row_6	False	0	{ "Rotation": { "X": 0, "Y": 0, "Z": 0, "W": 1 }, "Translation": { "X": 0, "Y": 0, "Z":	{ "Name": "", "ID": 0, "Description": "", "Mesh": "None", "Icon": "None" }

Row_1

ROW_1

Variable 1	☐
Variable 2	0
Variable 3	
Variable 4	

***Figure 9.27:** Data Table Editor*

The data table editor is very simple to use and is a great way to keep projects neat and organized.

CSV/JSON export

As data tables are spreadsheets, they can be exported/imported to/from CSV or JSON files for alteration or viewing in external spreadsheet editing software. To export a data table to a .CSV or a .JSON file, right-click on it within the content browser, and then select either **`Export as CSV`** or **`Export as JSON`**, as shown in *Figure 9.28*:

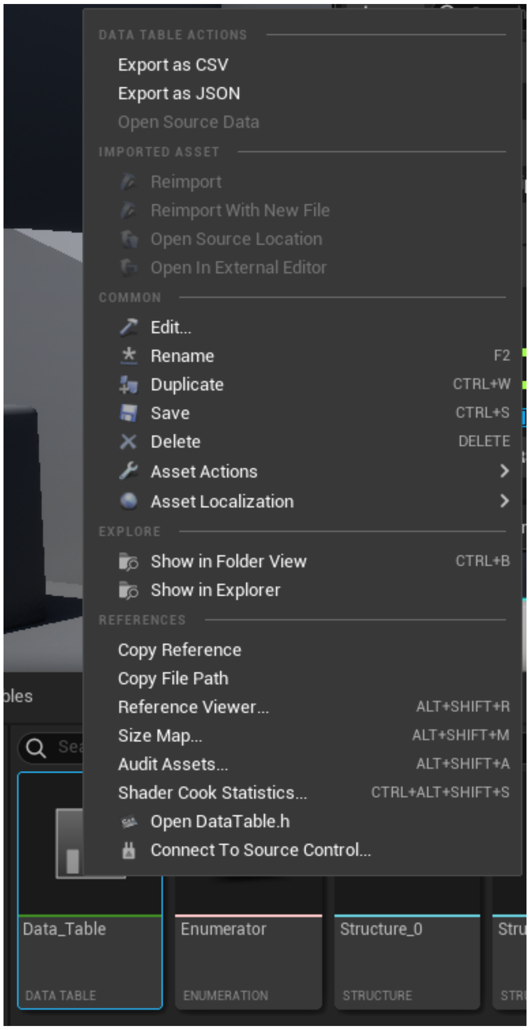


***Figure 9.28**: Data Table export options*

Data Table usage in blueprint

Data tables can be referenced using variables in blueprints, but specific nodes need to be used to read data from a blueprint. These nodes can be seen in *Figure 9.29*:

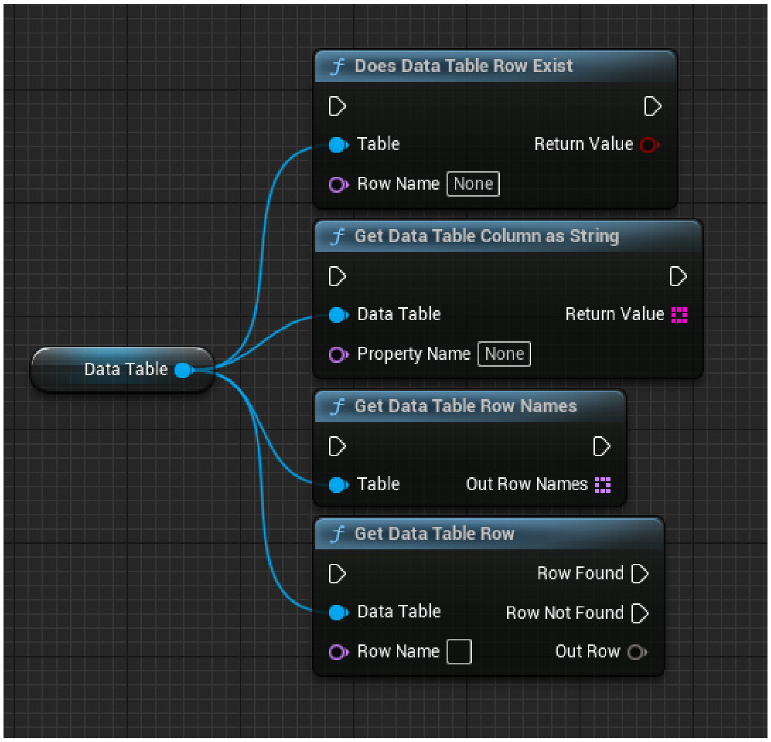


***Figure 9.29**: Data Table nodes*

These nodes each have specific uses that can be used well in conjunction with one another.

Does the data table row exist?

This node is used to check if there is a row in the input **`Data Table`** matching the input **`Row Name`**. This node serves as a **`IsValid`** node equivalent for data tables:

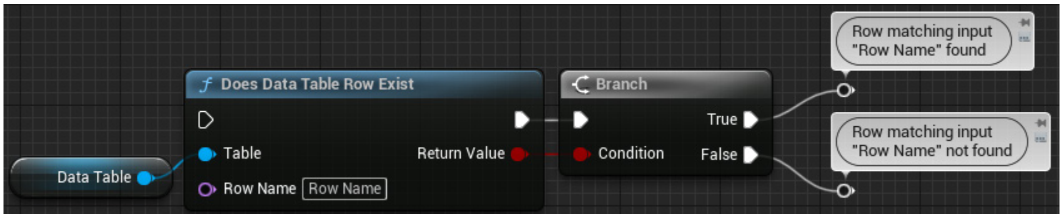

Row matching input "Row Name" found
Row matching input "Row Name" not found
Does Data Table Row Exist
Branch
Table
Return Value
Condition
True
False
Data Table
Row Name
Row Name

Figure 9.30: Does Data Table row exist

Get data table column as string

This node gets all the entries for a provided column/variable (the **`Property Name`** input) in a **`Data Table`** and returns them in an array of strings:

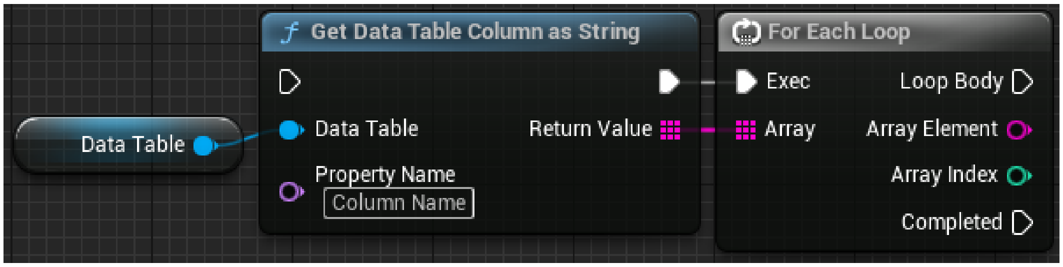


Figure 9.31: Get Data Table Column as String

Get data table row names

This node gets the row names for all the entries in a **`Data Table`**, which, when used in conjunction with **`Get Data Table Row`**, can allow searching or evaluating an entire **`Data Table`**.

Get data table row

This node gets the row under the provided **`Row Name`** (provided it exists) and returns the row as the struct the **`Data Table`** uses, with all the variable values that row contains in the data table:

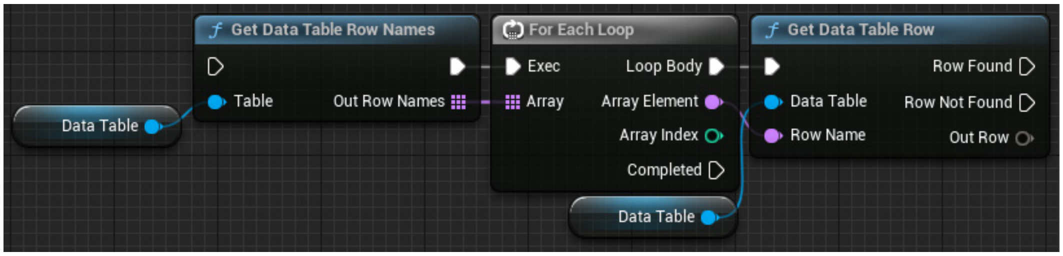


Figure 9.32: Get Data Table Row Names & Get Data Table Row

Animation instances

An animation instance is the current instance of whatever object is driving the animation of a skeletal mesh, which is usually an Animation Blueprint, but it can also be a singular animation asset or a custom method created by an advanced user.

Animation instance nodes

There are a handful of Blueprint nodes related to animation instances, used for altering, referencing, or obtaining information about the animation playing on a skeletal mesh.

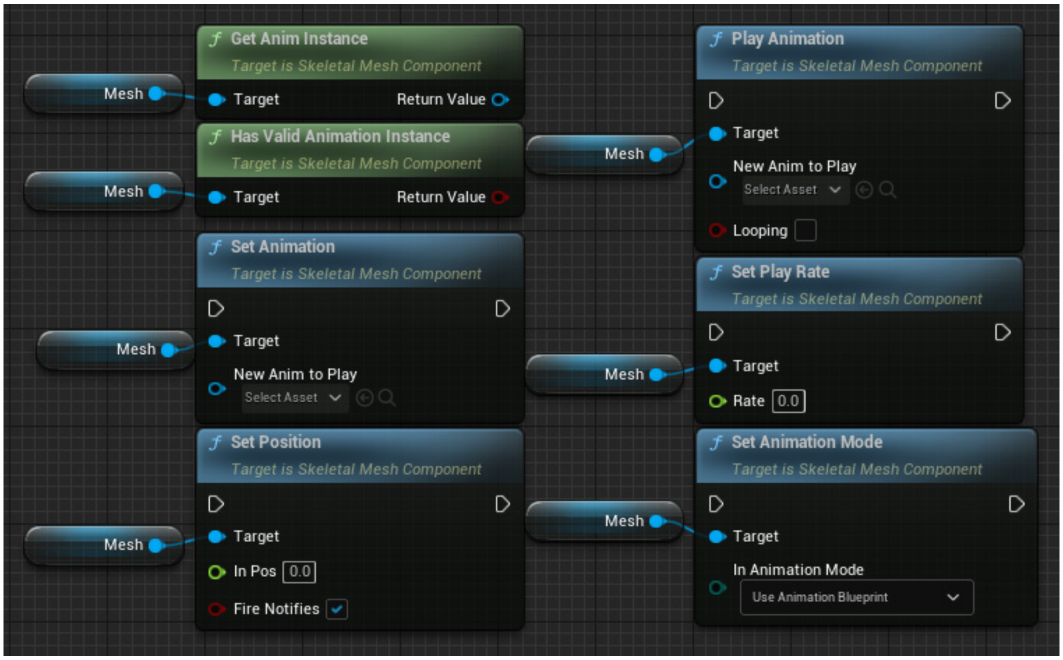
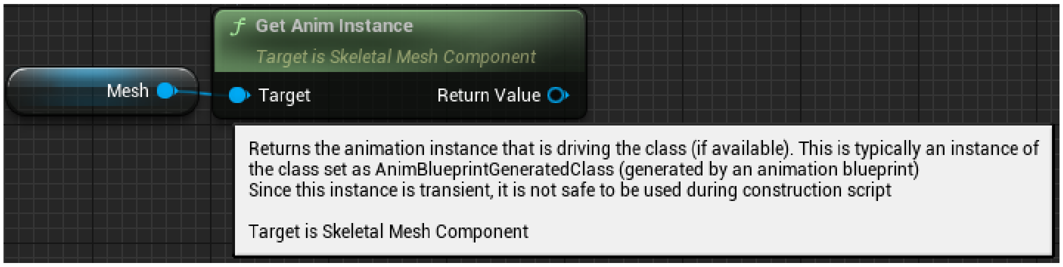


Figure 9.33*: Some Animation Instance Nodes*

Get Anim Instance

This node provides a reference to the animation instance for the provided mesh, which, in conjunction with a cast node, can be used to get a reference to a currently running animation blueprint (if the mesh is using one):



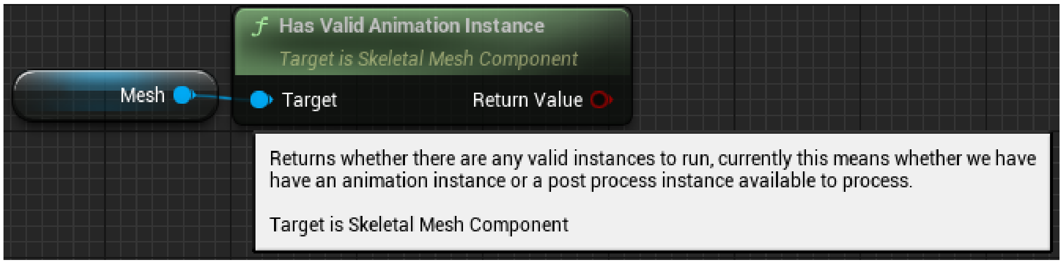
Returns the animation instance that is driving the class (if available). This is typically an instance of the class set as AnimBlueprintGeneratedClass (generated by an animation blueprint)
Since this instance is transient, it is not safe to be used during construction script

Target is Skeletal Mesh Component

***Figure 9.34**: Get Animation Instance Node*

Has valid animation instance

This node checks to see that the provided mesh has an animation instance, which can be used to avoid errors when trying to alter an Animation instance:


Has Valid Animation Instance
Target is Skeletal Mesh Component
Mesh
Target
Return Value

Returns whether there are any valid instances to run, currently this means whether we have have an animation instance or a post process instance available to process.

Target is Skeletal Mesh Component

***Figure 9.35**: Has Valid Animation Instance Node*

Play

This node plays the animation currently set on the provided skeletal mesh:

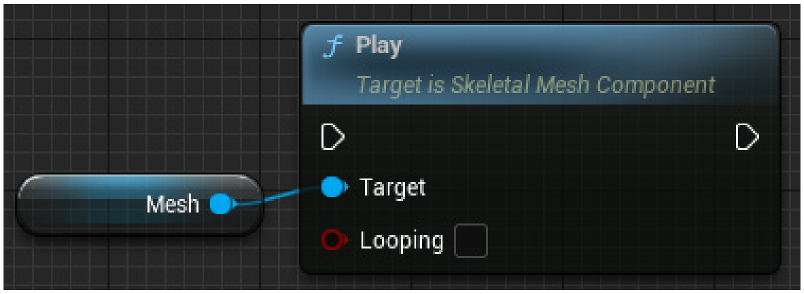


***Figure 9.36**: Play Node*

Stop

This node stops the animation currently playing on the provided skeletal mesh:

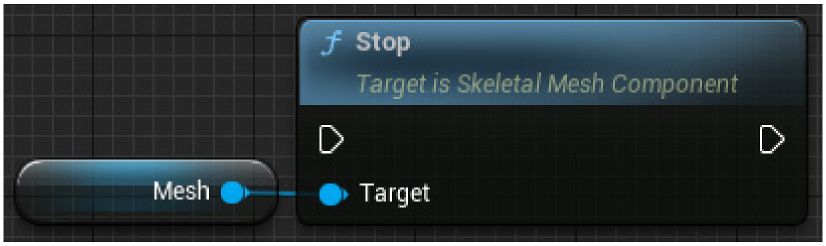


Figure 9.37: Stop Node

Set animation

This node changes the current animation for the provided mesh but doesn't automatically play the specified animation:

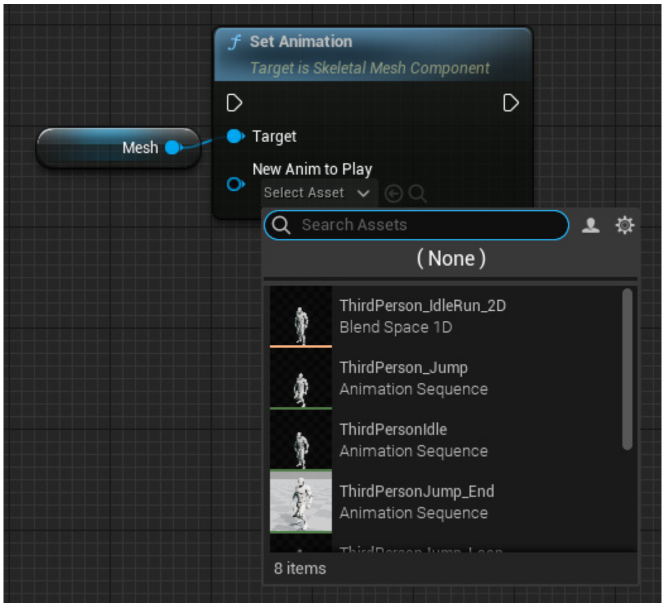


Figure 9.38: Set Animation Node

Set position

This node sets the current animation playing on the mesh to the provided time. The input **`In Pos`** defines (in seconds) the point in time to set the current animation to, and **`Fire Notifies`** sets whether any **`Animation Notifies`** between the position the animation is at before and after the node is executed are activated by this node:

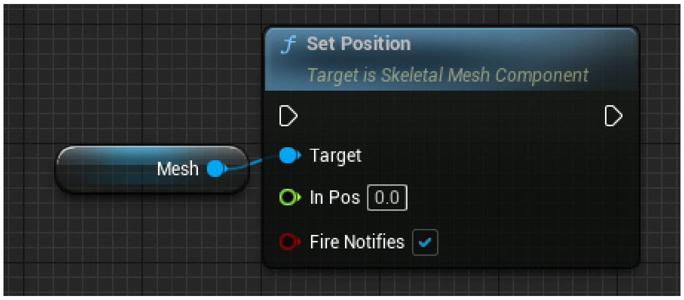


***Figure 9.39**: Set Position Node*

Get position

This node gets the current playback position of the animation of the provided mesh:

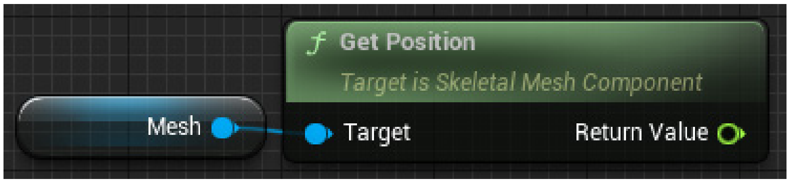


***Figure 9.40**: Get Position Node*

Is Playing

This node returns a Boolean stating whether the provided **`Skeletal Mesh`** is animating:

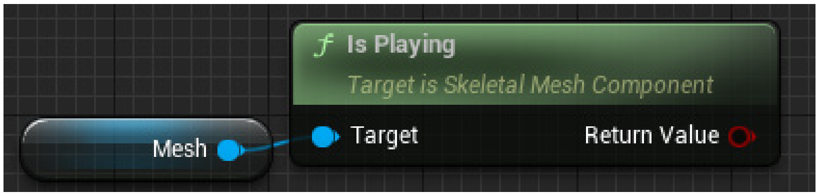


***Figure 9.41**: Is Playing Node*

Play animation

This node changes the current animation for the provided mesh and plays it. The input **`Looping`** sets whether the provided animation will replay after it completes. This node is essentially a combination of the **`Set Animation`** and the **`Play`** Nodes:

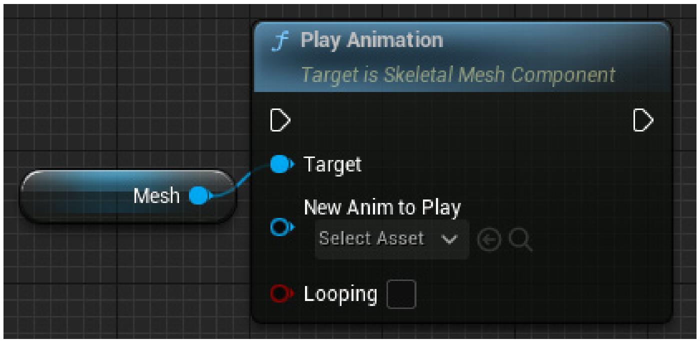


Figure 9.42: *Play Animation Node*

Set Play Rate

This node changes the speed that the current animation plays at. A value of one is standard speed, values lower than one are slower, and values higher than one are faster:

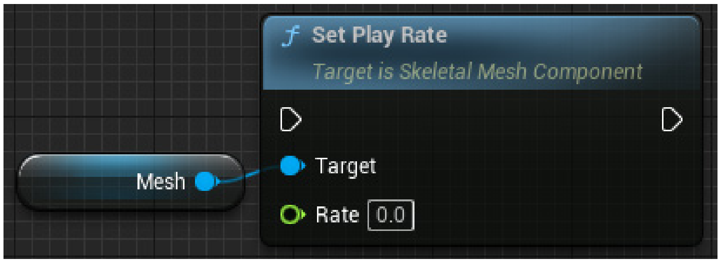


Figure 9.43: *Set Play Rate Node*

Get Play Rate

This node returns the speed that the current animation is playing at:

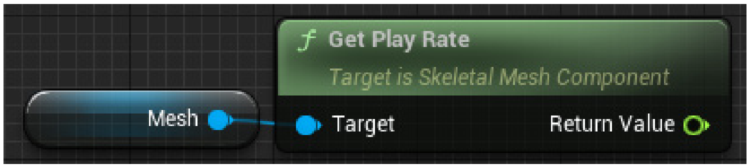


Figure 9.44: *Get Play Rate Node*

Set animation mode

This node changes the current animation mode for the provided mesh, which is used for switching between using singular animation, animation blueprints, and other user defined animation modes:

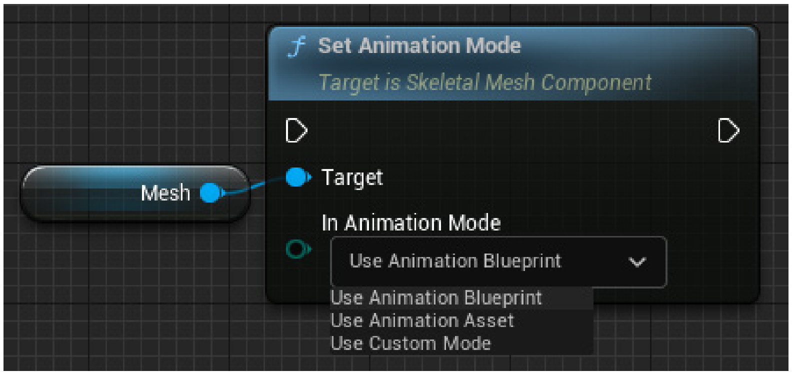


Figure 9.45: *Set Animation Mode Node*

Get animation mode

This node returns the current animation mode for the provided mesh:

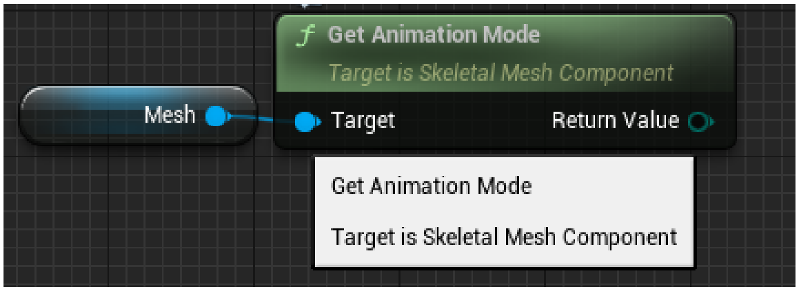

Get Animation Mode
Target is Skeletal Mesh Component
Mesh
Target
Return Value
Get Animation Mode
Target is Skeletal Mesh Component

Figure 9.46: *Set Animation Mode Node*

Override animation data

This node is a combination of most of the **`Animation Instance`** nodes previously covered. It is used for changing everything except the animation mode of a **`Mesh`** with a single node:

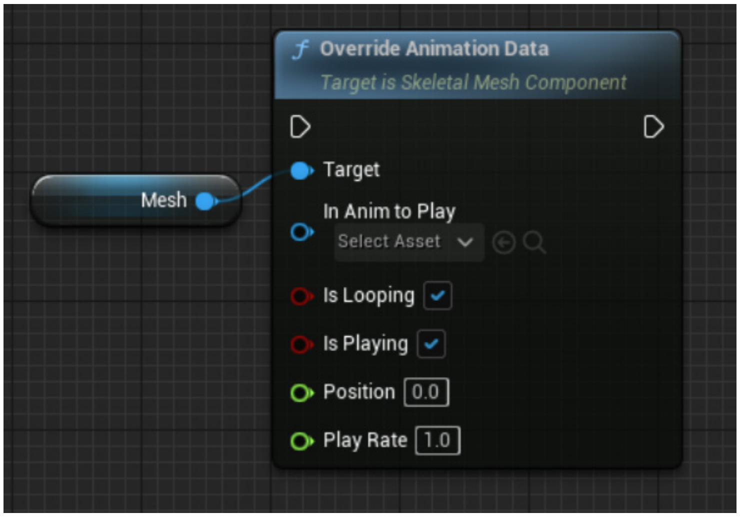
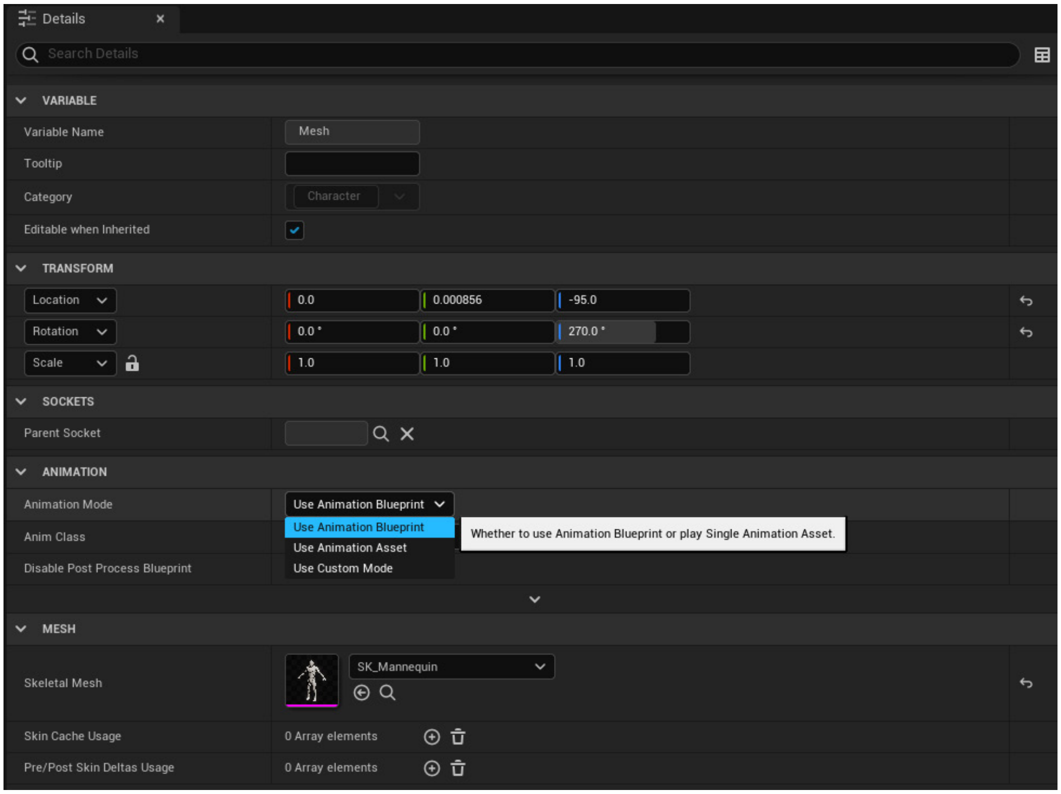


***Figure 9.47**: Override Animation Data Node*

It is better to use the other, more specialized nodes for setting specific parameters regarding the animation of a mesh, but when it all needs to change at once for whatever reason, this node is better in terms of performance and readability of your blueprint.

Animation modes

Skeletal meshes can only be animated by a single source at a time. The different methods of doing so are referred to as **`Animation Modes`**, and they include **`Animation Blueprints`**, **`Animation Assets`** and **`Custom Modes`**:


Details

Search Details

VARIABLE

Variable Name	Mesh
Tooltip	
Category	Character
Editable when Inherited	✓

TRANSFORM

Location	0.0	0.000856	-95.0
Rotation	0.0 °	0.0 °	270.0 °
Scale	1.0	1.0	1.0

SOCKETS

Parent Socket	

ANIMATION

Animation Mode	Use Animation Blueprint
Anim Class	
Disable Post Process Blueprint	

Whether to use Animation Blueprint or play Single Animation Asset.

MESH

Skeletal Mesh	SK_Mannequin
Skin Cache Usage	0 Array elements
Pre/Post Skin Deltas Usage	0 Array elements

***Figure 9.48**: Animation Mode*

Animation blueprints are specialized blueprints, specifically for providing skeletal meshes with the capacity for complex logic regarding animation, such as having two different animations play on different parts for a skeletal mesh, or having specific animations play under certain conditions. The custom animation mode is for advanced users who have created their own method for animating skeletal meshes.

Multi-variable type operators

A new feature in UE5 is that basic math operators can now natively handle variables of different types without the need for extra conversion or specific multi-variable math nodes. In order to make use of this, simply use a math operator and connect the different variables into the inputs:

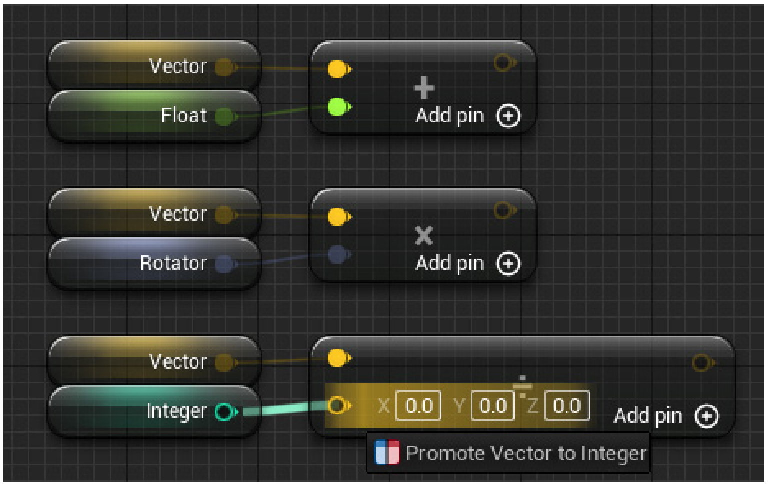


***Figure 9.49**: Multi-Variable Type Operators*

Note that only compatible variables can do this, and that the output value will be of the same type as the variable in the first pin slot.

Conclusion

There are more complicated variables in UE5, which have specialized aspects to them, making them harder to understand and use, but they can be used to great effect due to their complex nature. Variables are also capable of becoming more complex through the use of arrays and maps.

The animation of skeletal meshes can be managed through blueprint through the use of specialized nodes but can also be managed through Animation Blueprints when more complicated animation logic is required.

UE5 has made some improvements over its predecessor around basic math, allowing for easier management of mathematics involving different variable types.

By Next chapter, we learn more about game objects by going through class properties, settings of tick function, replications setting, and serialization data related to game objects.

Points to remember

- Properly used structures and data tables can make a project much easier to work on.

- Enumerators convert natively to bytes.
- Arrays and maps are similar and use some very similar nodes.
- Data tables can be exported in and out of UE5.
- Structures can contain structures, which are very useful for organization.
- Skeletal meshes can only be animated by a single source at a time.
- For loops and arrays are frequently used together.
- Structures cannot contain a variable of the same struct, preventing infinite loops.

Multiple choice questions

1. **What files can Data Tables be exported as?**
   a. JSON & CSV
   b. .JSON & PNG
   c. CSV & FBX
   d. JPG & PSD
2. **Can you add an entry to an array using the `Set Array Element` node?**
   a. Yes
   b. No
3. **What variable type does the output of a math operator use?**
   a. Integer
   b. The variable type of the first pin
   c. Vector

Answers

1. **a**
2. **a**
3. **b**

Questions

1. What is a break node?

2. What is an Animation Blueprint?
3. What is an operator?

Key terms

- **Array**: A collection of values in a list
- **Struct**: A collection of variables
- **Animation**: Movement over time
- **Export**: Outputting data from one program to use in another
- **Pin**: The input/output instance of a variable on a node

Chapter 10
Game Objects (Advanced) and Serialization

This chapter will cover the more advanced aspects of game objects, including more specialized settings for objects and an easy method of interaction between objects. It will also cover saving data to the system the game is running on for future usage (serialization), which is important for keeping track of player progress and their graphical and input settings.

Structure

In this chapter, we will discuss the following topics:

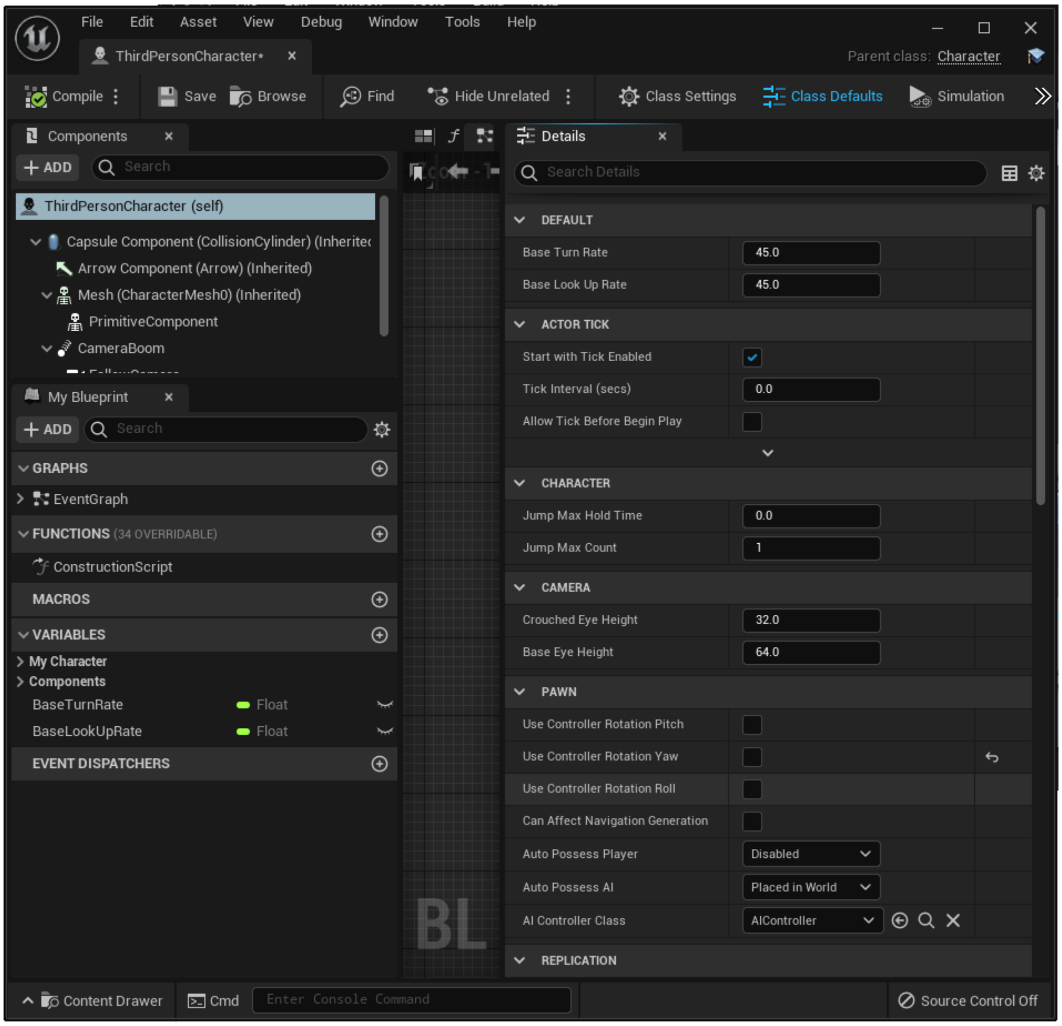
- Class defaults
- Class settings
  - Changing inheritance
  - Interfaces (setup, returns, and usage)
- Tick settings
- Replication settings
- Serialization
  - Save game objects
  - User settings (scalability and input)

Objectives

After studying this chapter, you should understand the core settings of game objects and how to alter them, in addition to being able to save data and alter the engine's settings from within the project.

Class defaults

At the top of the blueprint editor, there is a button to open up the default settings of the currently open blueprint; within this menu are options for changing the default values for variables within the blueprint, and for changing the values used by its components:



DEFAULT

Base Turn Rate	45.0
Base Look Up Rate	45.0

ACTOR TICK

Start with Tick Enabled	✓
Tick Interval (secs)	0.0
Allow Tick Before Begin Play	☐

CHARACTER

Jump Max Hold Time	0.0
Jump Max Count	1

CAMERA

Crouched Eye Height	32.0
Base Eye Height	64.0

PAWN

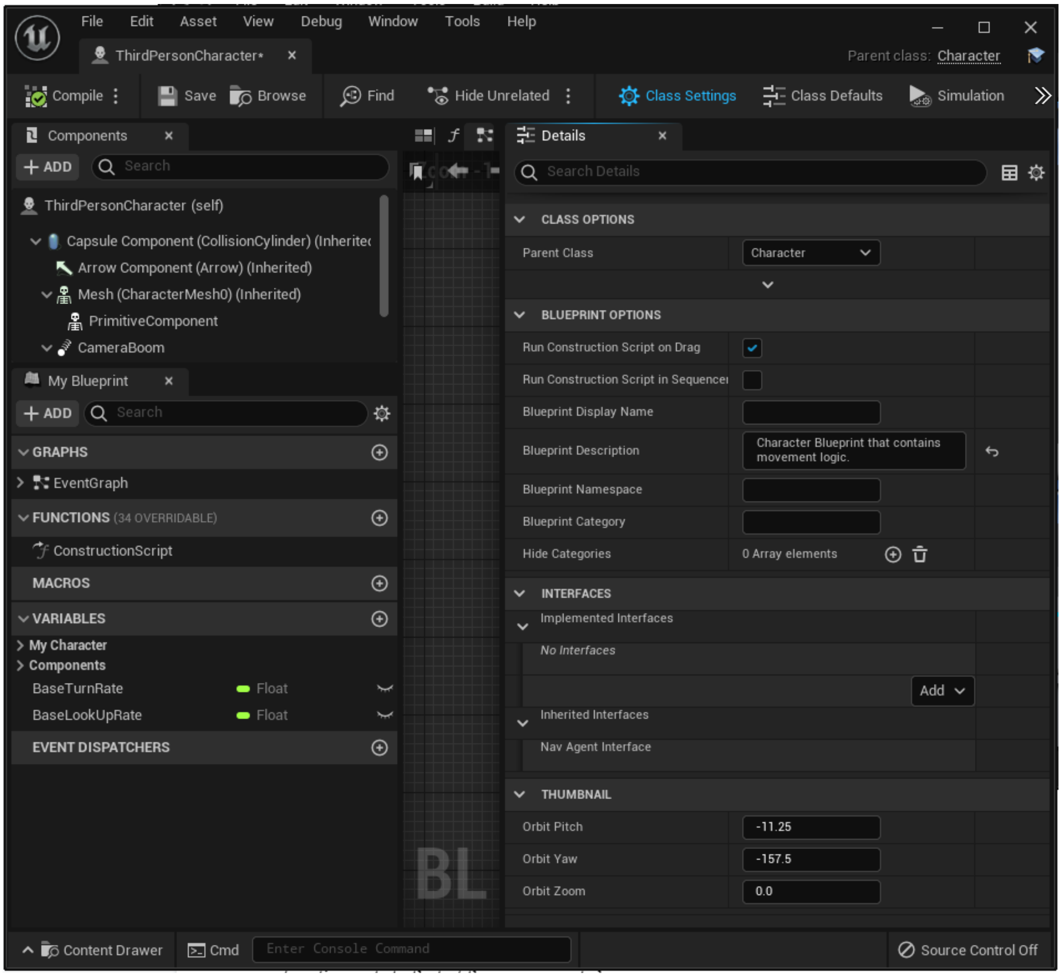
Use Controller Rotation Pitch	☐
Use Controller Rotation Yaw	☐
Use Controller Rotation Roll	☐
Can Affect Navigation Generation	☐
Auto Possess Player	Disabled
Auto Possess AI	Placed in World
AI Controller Class	AIController

REPLICATION

***Figure 10.1**: Class Defaults Settings*

Class settings

Next to the class defaults button, there is a button to open the class settings of the currently open blueprint; within this menu are options for changing the parent class, managing interfaces, modifying the thumbnail used for the blueprint in the content browser, changing the blueprint's metadata, and a couple of construction script settings:



CLASS OPTIONS

Parent Class	Character

BLUEPRINT OPTIONS

Run Construction Script on Drag	✓
Run Construction Script in Sequencer	
Blueprint Display Name	
Blueprint Description	Character Blueprint that contains movement logic.
Blueprint Namespace	
Blueprint Category	
Hide Categories	0 Array elements

INTERFACES

Implemented Interfaces

No Interfaces

Add

Inherited Interfaces

Nav Agent Interface

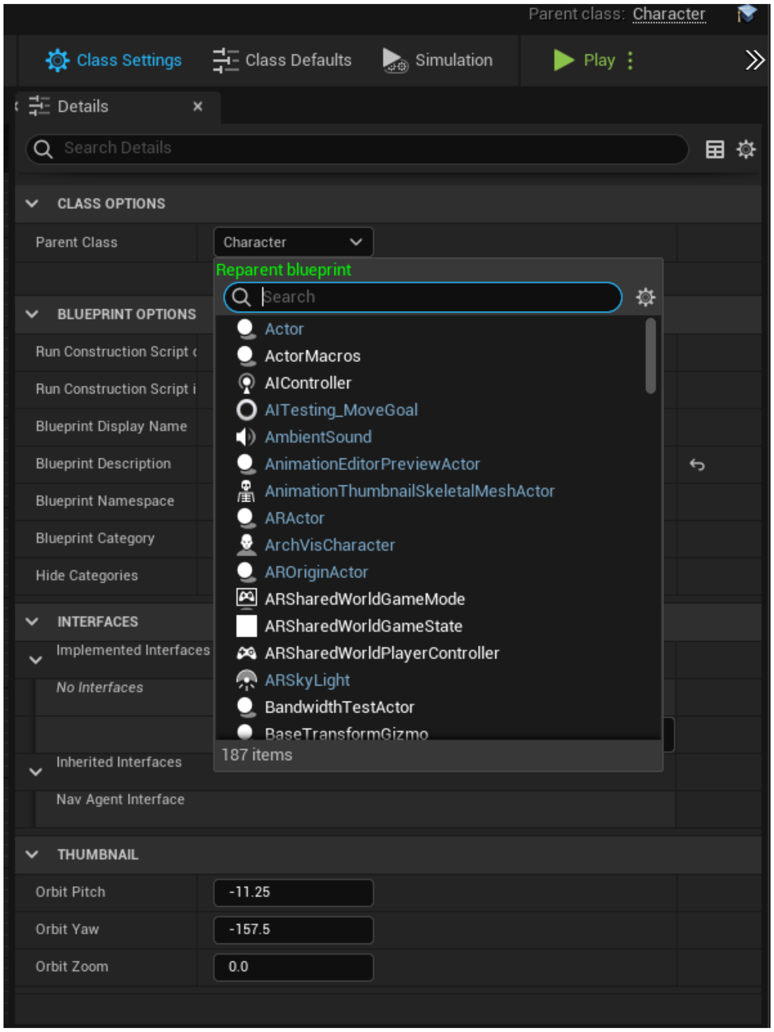
THUMBNAIL

Orbit Pitch	-11.25
Orbit Yaw	-157.5
Orbit Zoom	0.0

***Figure 10.2**: Class settings*

Changing inheritance

Inheritance can be changed from within the class settings of a blueprint; this allows you to change the parent of a blueprint, changing its inherited components, variables, functions, and so on to that of the new parent class:



Parent Class: Character

Orbit Pitch: -11.25

Orbit Yaw: -157.5

Orbit Zoom: 0.0

Figure 10.3*: Changing inheritance*

Interfaces

An interface is a way to get two objects to interact without requiring a reference to a specified class (*casting*), they are incredibly useful for having a single interaction from one object with numerous reactions from many different objects.

Setup

In order to use interfaces, they first need to be created; this can be done by right-clicking in the content browser and selecting the **`Blueprint Interface`** option under the **`Blueprints`** category:

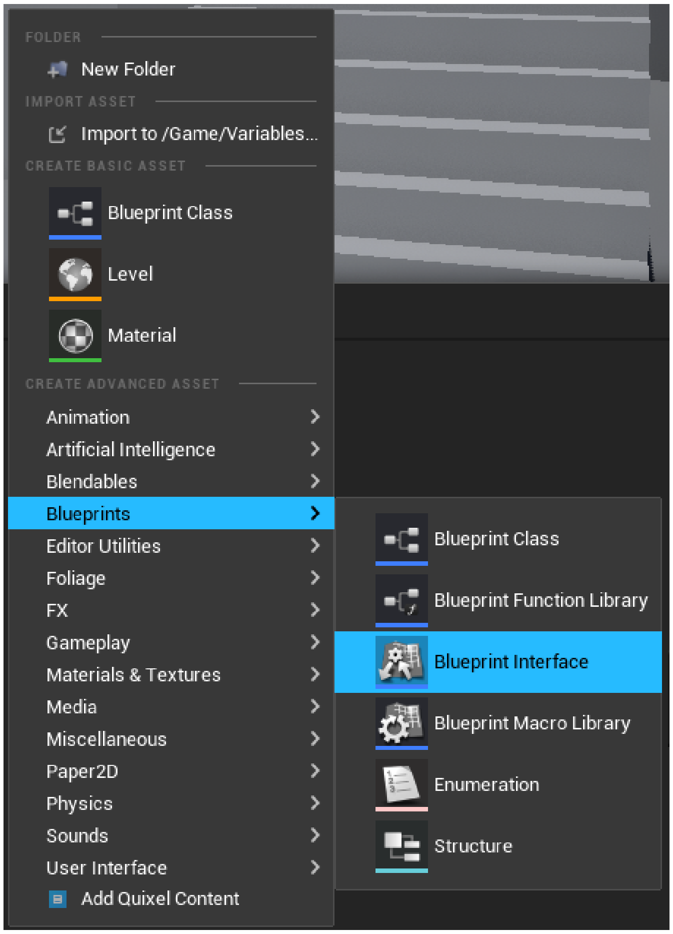


***Figure 10.4**: Creating interface*

Once created, interfaces need to be set to be used by each blueprint that needs to use it. This is done within the interface section of the **`Class Settings`** menu.

Returns

Like data tables and enumerators, interfaces have their own editor. Within this editor, you can add functions to be called through this interface and manage inputs and outputs to be used by it. You can't add code to interface's function inside the interface editor, but this can be done in each blueprint that uses that interface.

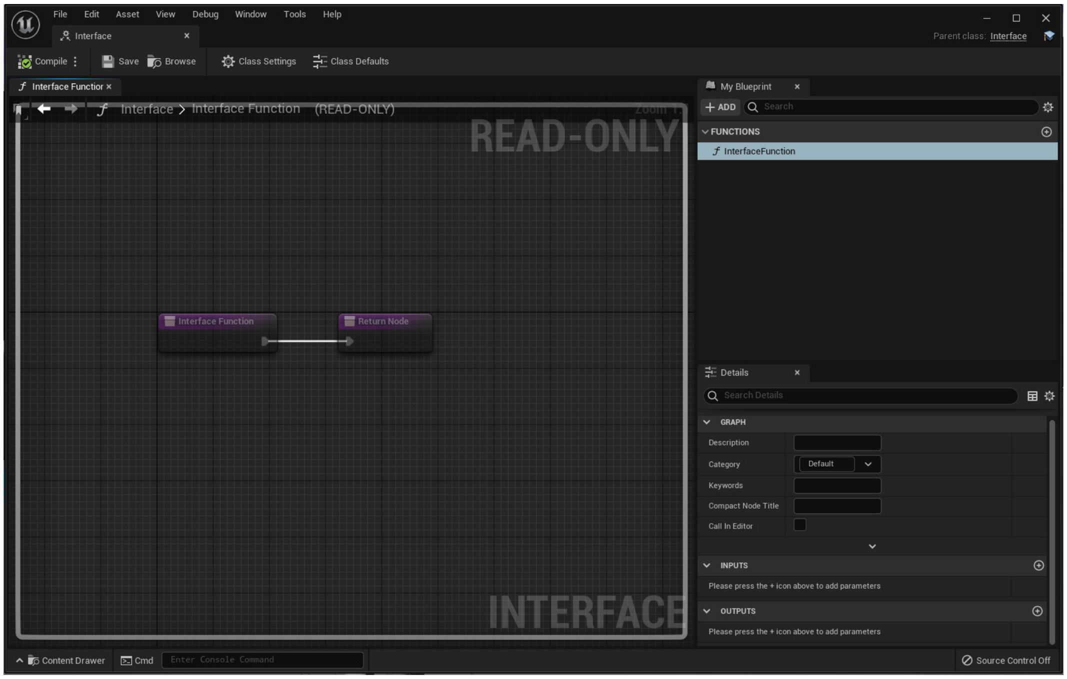


Figure 10.5: Interface Editor

Interfaces can contain multiple functions, which can be seen and managed in the **`Functions`** section of the **`My Blueprint`** panel within the interface editor, while the **`INPUTS`** and **`OUTPUTS`** of the currently selected function are located within the **`Details`** panel, which also contains metadata settings for the function (description, category, and so on).

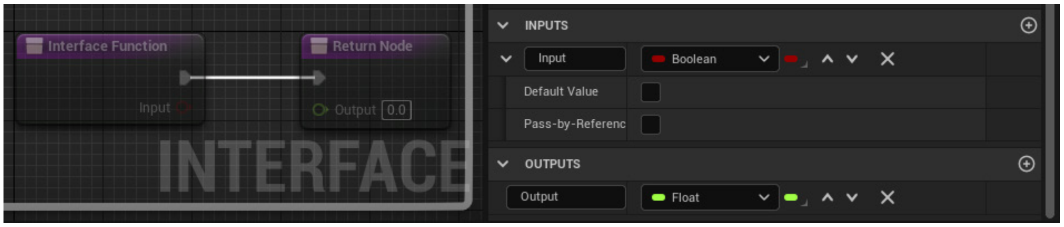


Figure 10.6: Interface function inputs and outputs

Usage

In order for interfaces to be utilized by a blueprint, they need to be added to the list of interfaces used by the blueprint from within the interface section of its **`Class`**

`Settings` (as seen in *Figure 10.7*), and then the function(s) desired to be used need to be defined within the blueprint:

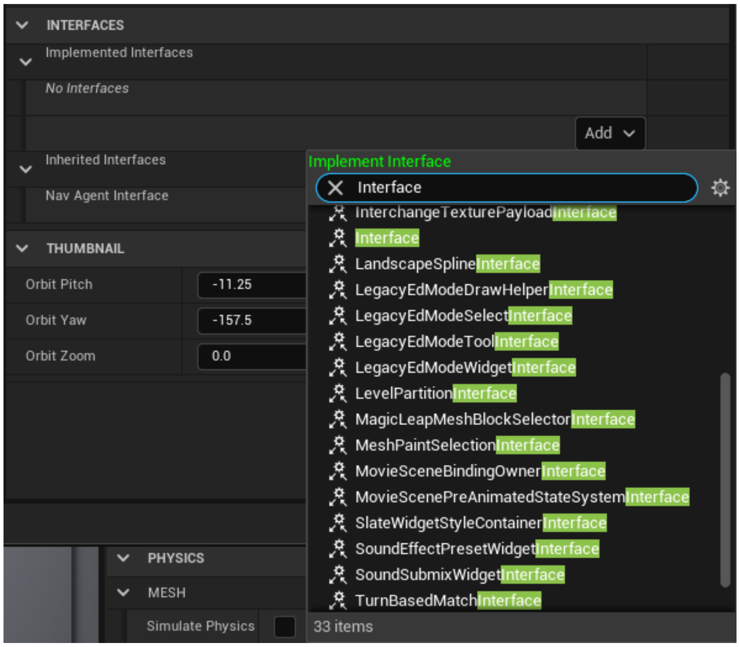

INTERFACES
Implemented Interfaces
No Interfaces
Add
Inherited Interfaces
Nav Agent Interface
THUMBNAIL
Orbit Pitch -11.25
Orbit Yaw -157.5
Orbit Zoom 0.0
PHYSICS
MESH
Simulate Physics
Implement Interface
Interface
InterchangeTexturePayloadInterface
Interface
LandscapeSplineInterface
LegacyEdModeDrawHelperInterface
LegacyEdModeSelectInterface
LegacyEdModeToolInterface
LegacyEdModeWidgetInterface
LevelPartitionInterface
MagicLeapMeshBlockSelectorInterface
MeshPaintSelectionInterface
MovieSceneBindingOwnerInterface
MovieScenePreAnimatedStateSystemInterface
SlateWidgetStyleContainerInterface
SoundEffectPresetWidgetInterface
SoundSubmixWidgetInterface
TurnBasedMatchInterface
33 items

***Figure 10.7**: Adding an interface to an Actor*

After an interface is used by an actor, the functions inside the interface can be defined within the blueprint. This is done by first implementing the function under the **`INTERFACES`** section of the **`My Blueprint`** panel within the blueprint editor, as shown in *Figure 10.8*:

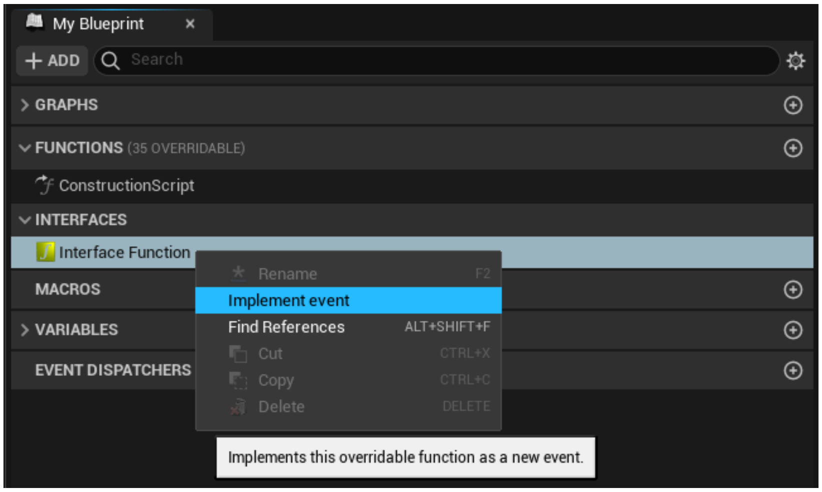


***Figure 10.8**: Implementing interface function*

Once the function has been implemented, it can be defined, as shown in *Figure 10.9*. If an interface function has outputs, it will be handled as a function within the blueprint; if it does not, it will be treated as an event:

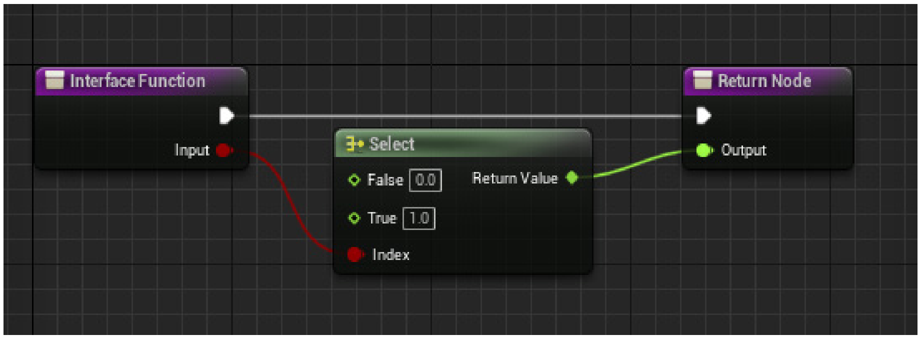


Figure 10.9*: Defined interface function*

After implementing an **`Interface`** function, it can be called. This is done by getting a reference to the object to execute the function and then executing it using a message node. It is also suggested to make use of a **`Does Implement Interface`** node to avoid errors, which can be seen in *Figure 10.10* alongside the message node:

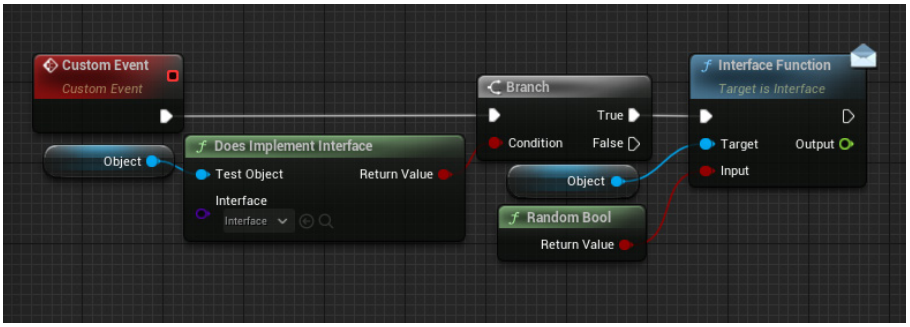
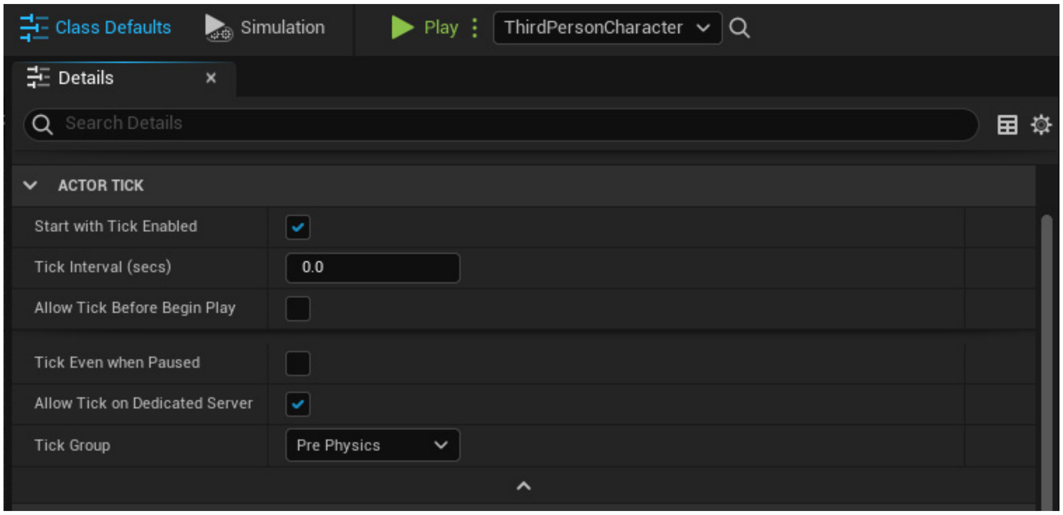


Figure 10.10*: Calling interface function*

Tick settings

Tick settings can be found under the **`ACTOR TICK`** section of a blueprint's **`CLASS DEFAULT`** settings. These settings directly influence the **`Tick`** event covered in *Chapter 6, Game Objects and Materials*, and when the Actor's physics are calculated:



Start with Tick Enabled: ✓
Tick Interval (secs): 0.0
Allow Tick Before Begin Play: ☐
Tick Even when Paused: ☐
Allow Tick on Dedicated Server: ✓
Tick Group: Pre Physics

Figure 10.11: *Tick Settings*

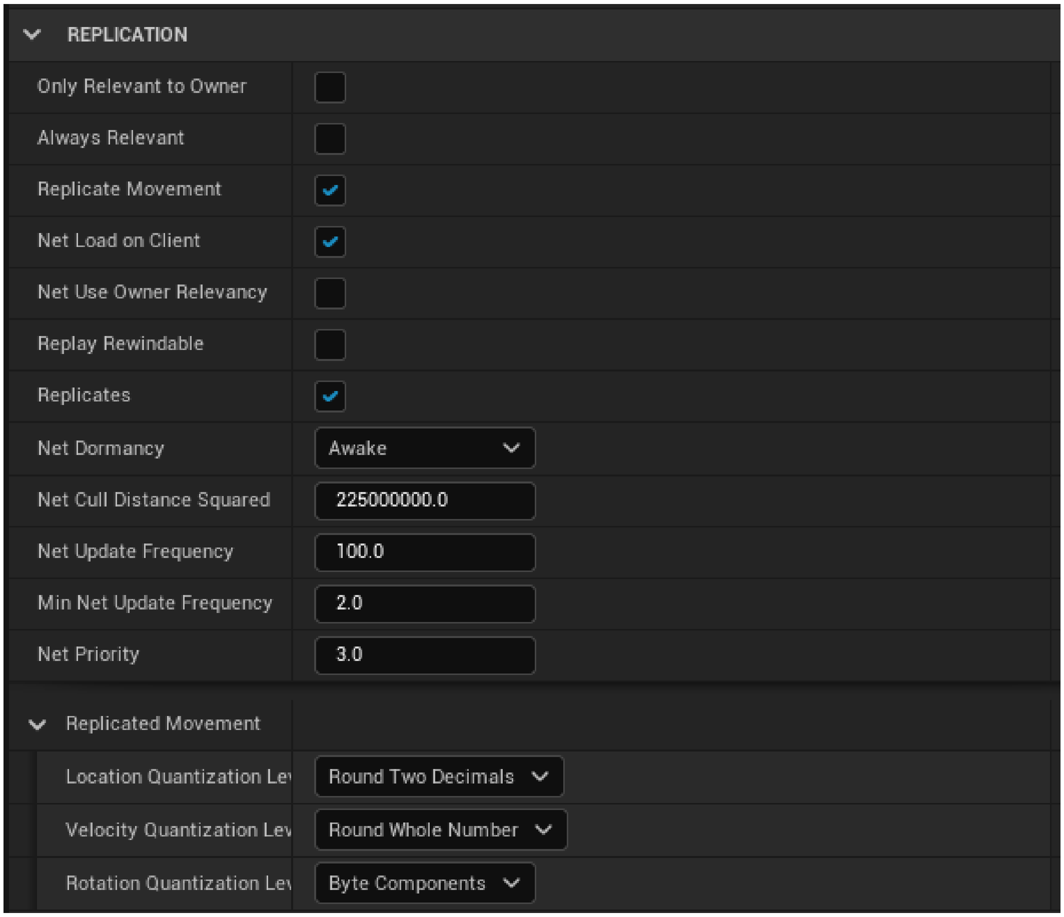
The tick settings are as follows:

- **`Start with Tick Enabled`**: Sets whether the Actor will be ticking by default when it is initialized. An Actor's Tick can be enabled or disabled during runtime if necessary.
- **`Tick Interval (secs)`**: Sets how long in seconds the actor is to wait between Ticks. This setting is very useful for optimization, as the **`Tick Event`** can be very performance intensive depending on what the event is used for, so delaying a tick will ease the load placed on the system, although the functionality dependent on the tick will not always be completely updated.
- **`Allow Tick Before Begin Play`**: Allows the actor to tick before it is initialized and the **`BeginPlay`** event is executed.
- **`Tick Even when Paused`**: Sets whether the Actor will be allowed to continue ticking when the project is paused, although the tick likely won't function properly if it does so, depending on what logic the event is executing.
- **`Allow Tick on Dedicated Server`**: Sets whether the Actor is allowed to tick on a server that is not also functioning as a client. This is a multiplayer setting that should be disabled for actors that are intended to be handled locally.

- **`Tick Group`**: This setting determines the point during the engine updating that this object's tick should execute. This is useful for making sure that the functionality of the **`Tick Event`** on multiple objects interacts properly. For example, if you have an actor that has **`Tick`** functionality depending on values set in another actor's Tick, you will want the **`Tick`** of the dependent object to execute after that of the actor setting the value, so the value is always up to date and valid.

Replication settings

Replication is a multiplayer term that refers to the transmission of data between a server and clients. A blueprint class's replication setting (as seen in *Figure 10.12*) can be found within its **`CLASS DEFAULTS`**:



REPLICATION

Only Relevant to Owner	☐
Always Relevant	☐
Replicate Movement	✓
Net Load on Client	✓
Net Use Owner Relevancy	☐
Replay Rewindable	☐
Replicates	✓
Net Dormancy	Awake
Net Cull Distance Squared	225000000.0
Net Update Frequency	100.0
Min Net Update Frequency	2.0
Net Priority	3.0

Replicated Movement

Location Quantization Lev	Round Two Decimals
Velocity Quantization Lev	Round Whole Number
Rotation Quantization Lev	Byte Components

Figure 10.12*: Replication settings*

The replication settings are as follows:

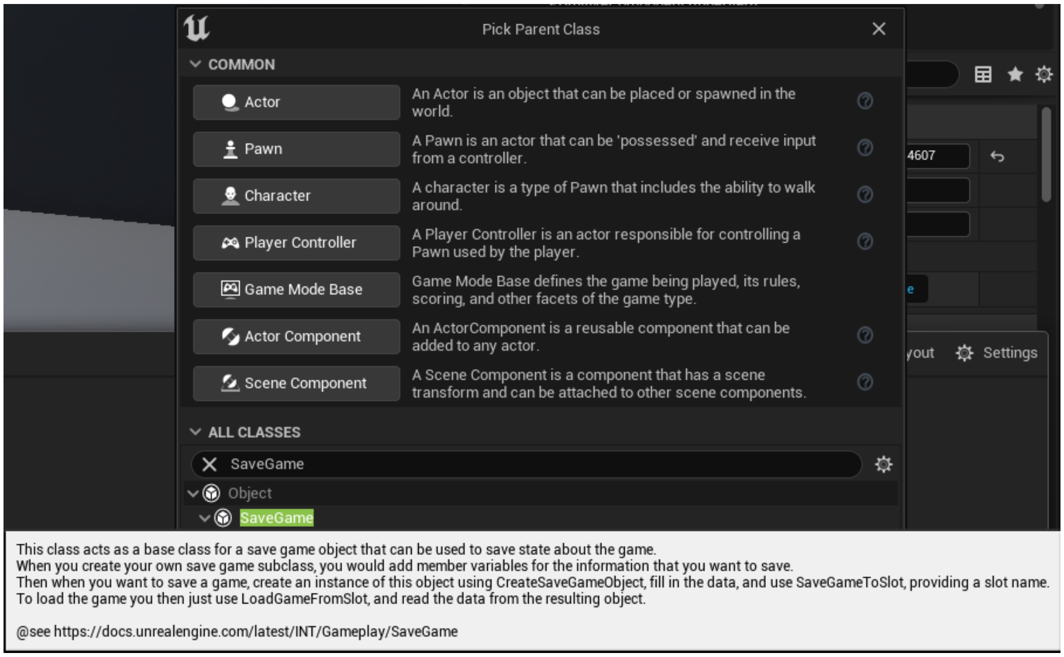
- **`Only Relevant to Owner`**: Relevancy is whether an object is deemed capable of influencing or being visible to a client. This setting specifically sets the object to only be relevant to the owning object, which for a pawn, would be its controller.
- **`Always Relevant`**: This sets the object to always be relevant.
- **`Replicate Movement`**: This sets the object's positioning and rotation to be replicated between the server and clients.
- **`Net Load on client`**: This forced actor to be loaded by clients loading the level.
- **`Net Use Owner Relevancy`**: This sets the actor to share relevancy with its owner.
- **`Replay Rewindable`**: This sets whether the object will be handled when using UE5's replay system.
- **`Replicates`**: This sets the object to be replicated to clients.
- **`Net Dormancy`**: A dormant object is one that is capable of being replicated but is temporarily set not to. This setting allows the user to disable dormancy entirely, to allow it but disable it by default, make the object always dormant, and make it partially or initially dormant.
- **`Net Cull Distance Squared`**: This is the squared value of the distance away from a client's viewpoint at which the object will stop being relevant and replicated.
- **`Net Update Frequency`**: This sets how many times per second the object's replicated data will be updated.
- **`Min Net Update Frequency`**: This sets how many times per second the object's replicated data will be updated if the object's replicated data is changed infrequently.
- **`Net Priority`**: This is the importance of replicating the actor; a higher value means the object is more likely to be replicated under a stressed network connection.
- **`Replicated Movement`**: This is used for fine-tuning the accuracy of the object's replicated positioning, rotation, and velocity.

Serialization

Serialization is storing data on the user's device for later usage. In the case of save game objects, this can be used for storing gameplay data, such as where the player character is located, what's in their inventory, their name, and so on. User settings, such as graphical and input settings, are handled separately by the engine.

Save game objects

In order to make use of save game objects, a **Save Game** class first needs to be created and defined. This can be done by selecting **Blueprint Class** within the context menu of **Content Browser**, and then, under **All Classes** in the menu that is created, selecting **SaveGame** as the parent class, as shown in *Figure 10.13*. The *save game class* is used as the template for the save game objects that will be created to store player data:



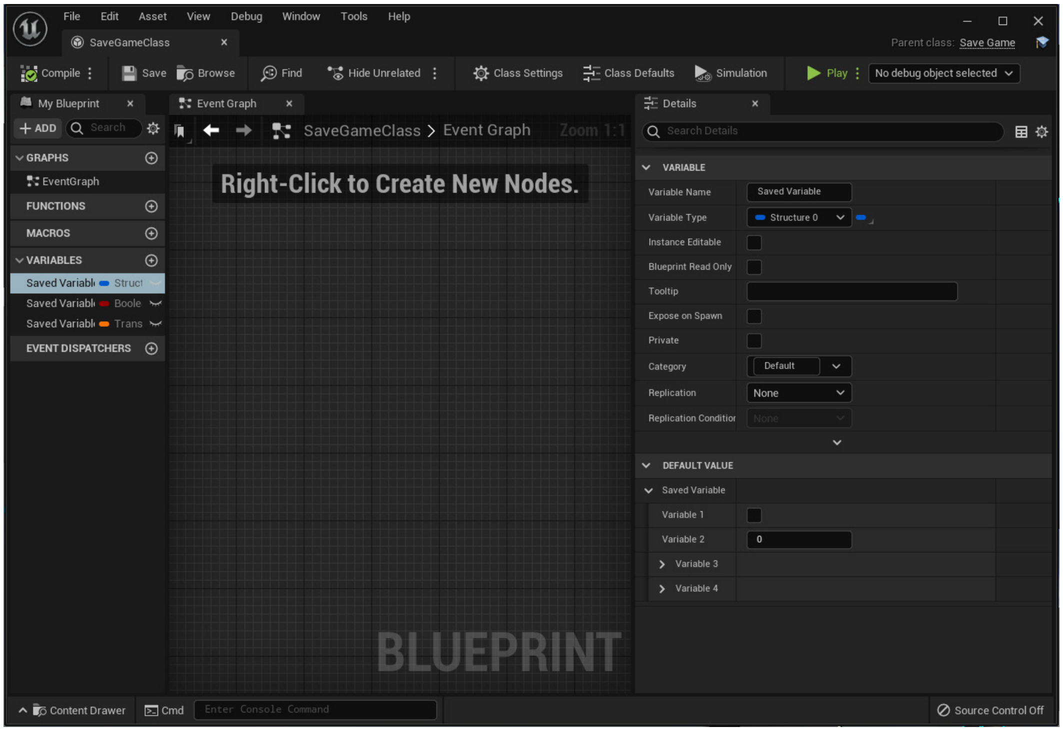
An Actor is an object that can be placed or spawned in the world.
A Pawn is an actor that can be 'possessed' and receive input from a controller.
A character is a type of Pawn that includes the ability to walk around.
A Player Controller is an actor responsible for controlling a Pawn used by the player.
Game Mode Base defines the game being played, its rules, scoring, and other facets of the game type.
An ActorComponent is a reusable component that can be added to any actor.
A Scene Component is a component that has a scene transform and can be attached to other scene components.

This class acts as a base class for a save game object that can be used to save state about the game.
When you create your own save game subclass, you would add member variables for the information that you want to save.
Then when you want to save a game, create an instance of this object using CreateSaveGameObject, fill in the data, and use SaveGameToSlot, providing a slot name.
To load the game you then just use LoadGameFromSlot, and read the data from the resulting object.

@see https://docs.unrealengine.com/latest/INT/Gameplay/SaveGame

Figure 10.13: *Creating Save Game Class*

Save game classes have their own editor very similar to the structure editor (as shown in *Figure 10.14*), in which the data that the save game object will store can be defined in the form of **VARIABLES**:



VARIABLE

Variable Name	Saved Variable
Variable Type	Structure 0
Instance Editable	☐
Blueprint Read Only	☐
Tooltip	
Expose on Spawn	☐
Private	☐
Category	Default
Replication	None
Replication Condition	None

DEFAULT VALUE

Saved Variable	
Variable 1	☐
Variable 2	0
Variable 3	
Variable 4	

***Figure 10.14**: Save Game Class Editor*

Once a **`Save Game`** class has been created and defined, it is ready to be used. This is done through blueprint, by creating A **`Save Game Object`**, setting the variables within it, and then saving it to the user device with a specific name and user index. This can be seen in *Figure 10.15*:

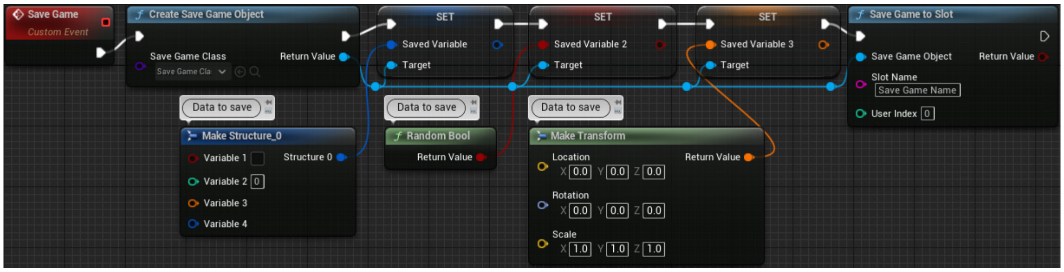


***Figure 10.15**: Saving*

In order to load a previously created **`Saved Game Object`**, a variant of the **`Load Game from Slot`** node needs to be used, with an input slot name that matches that of a previously created **`Save Game Object`**. Once this is done, the data from the **`Save Game Object`** can be obtained using a cast node, and then the variables can be used as needed. This entire process can be seen in *Figure 10.16*:

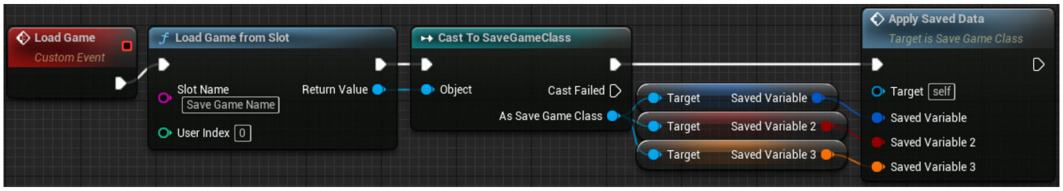


***Figure 10.16**: Loading*

User settings

User settings are the user's graphical and input settings that are stored and used by the engine. They are handled separately to **`Save Game Objects`**. These settings are accessed using the **`Get Game User Settings`** and **`Get Input Settings`** nodes shown in *Figure 10.17*:

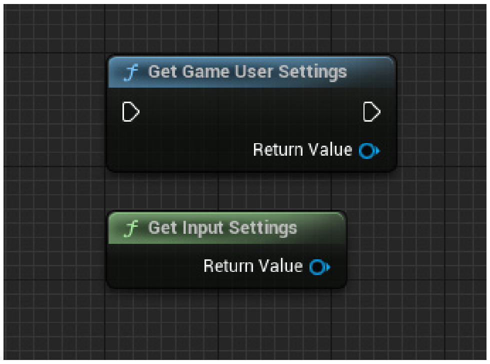


***Figure 10.17**: User Settings nodes*

Scalability

Scalability refers to the visual quality the engine is rendering the project at. This can be modified in the editor and at runtime through blueprint, with the functionality usually accessed by the player through UI elements, such as a settings menu. An example of the texture quality being changed can be seen in *Figure 10.18*:

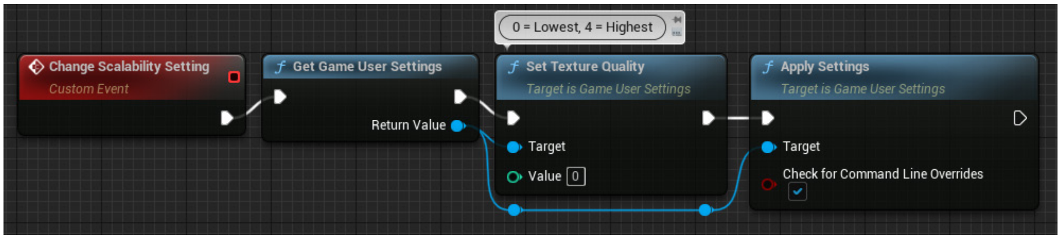


***Figure 10.18**: Changing Scalability*

The value of **`Scalability`** settings can also be obtained using the **`Get Game User Settings`** node, as shown in *Figure 10.19*. This is very useful for altering level elements based on the current settings the project is operating under, which is, in turn, incredibly useful for optimizing the project:

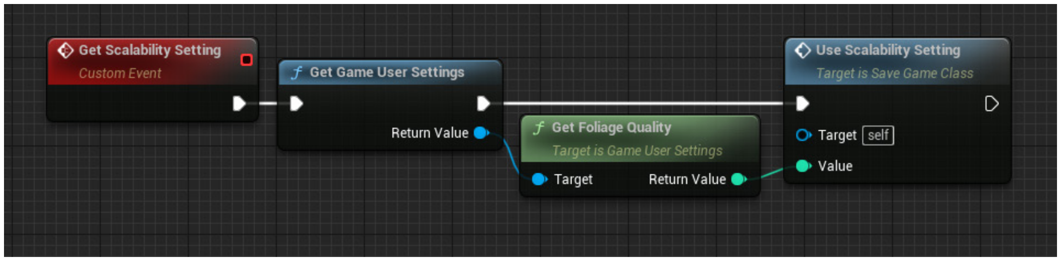


***Figure 10.19**: Getting Scalability*

In addition to specific graphics settings, the overall graphical settings of the engine can be altered via the **`Set Overall Scalability Level`** Node, as shown in *Figure 10.20*:

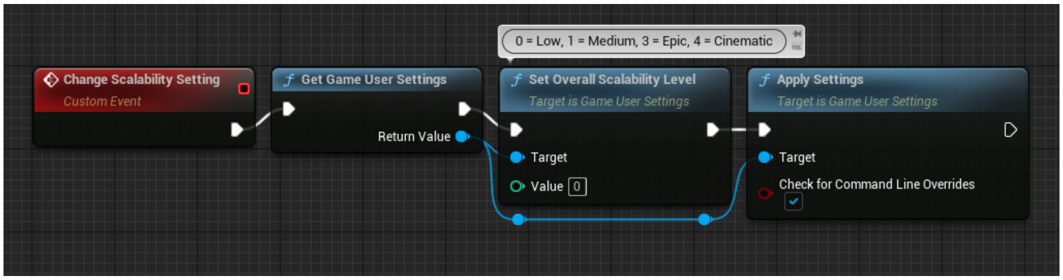

0 = Low, 1 = Medium, 3 = Epic, 4 = Cinematic
Change Scalability Setting
Custom Event
Get Game User Settings
Return Value
Set Overall Scalability Level
Target is Game User Settings
Target
Value 0
Apply Settings
Target is Game User Settings
Target
Check for Command Line Overrides

***Figure 10.20**: Setting Overall Scalability*

Input

Input is how the user interacts with the project; they use an input on their device, be it a keyboard, mouse or some form of controller, and the engine takes that input and executes necessary events tied to those inputs, as covered in *Chapter 5, Templates and*

User Interaction. Inputs can have mappings added using the "**`Add Action Mapping`**" Node, as shown in *Figure 10.21*:

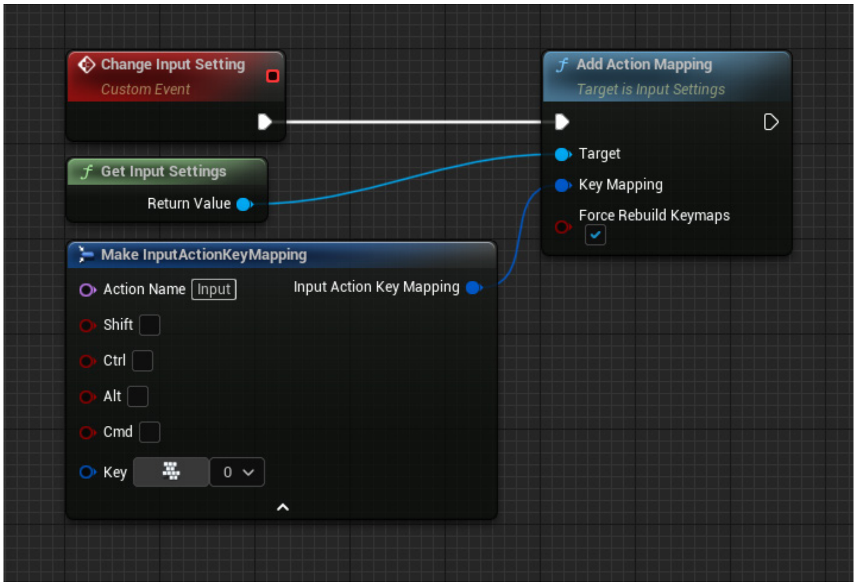


***Figure 10.21**: Adding Input Mapping*

Inputs can also have mappings removed using the **`Remove Action Mapping`** node, as shown in *Figure 10.22*:

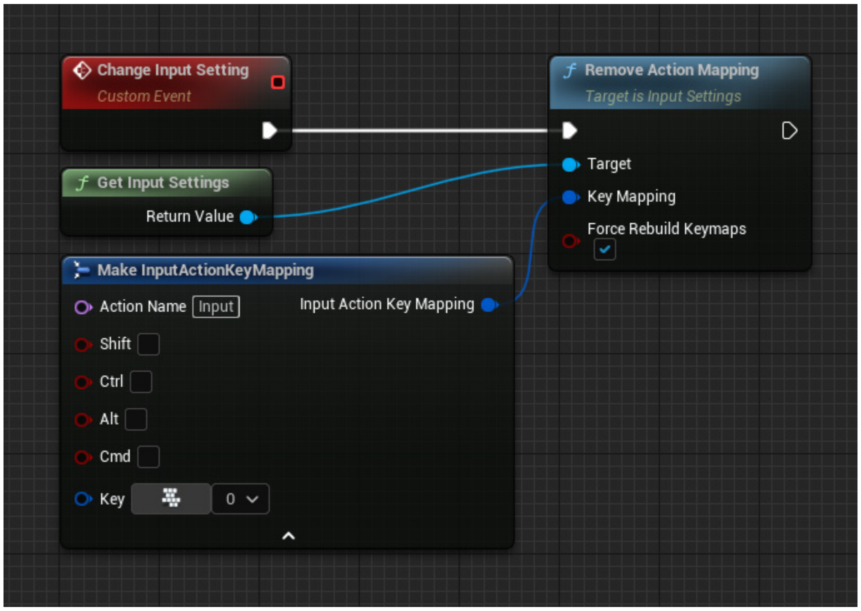


***Figure 10.22**: Removing Input Mapping*

Conclusion

Developers have a great deal of control over game objects, how they interact, and how the user interacts with and views the project. You can greatly modify how objects act by default and over the internet via multiplayer by changing their default and class settings and can even change the inheritance of an object easily. Most of these settings are rather advanced and will generally see minimal usage, but in the niche situations they apply to, it is incredibly useful to be aware of them and their purpose.

Interfaces are a great way to standardize interaction between objects neatly; they can be applied to almost any situation. Serialization is important not just for keeping track of player progress throughout a game but also for input and the visuals of the project.

By next chapter we learn how to add sound to game, by using a new feature known as *Metasound* in Unreal Engine 5. Indeed, we will learn using Niagara to add particle system to the scene and control it by blueprint code.

Points to remember

- Inputs can have multiple mappings.
- You can change the overall scalability as well as specific settings.
- Interfaces with outputs are treated as functions, and those without are treated as events.
- Objects inherit the functions, variables, events, and components of their parents.
- Ticks can be delayed to improve performance.
- Only relevant objects will be replicated.

Multiple choice questions

1. **Where are the Tick Settings located?**
    a. Class Defaults
    b. Content Browser
    c. Class Settings
    d. Interface Editor

2. **What node is used to get/set scalability settings?**
    a. Get User Game Settings
    b. Get Game User Settings
    c. Get Input Settings
3. **Is Tick limited when the game is paused?**
    a. Yes
    b. No

Answers

1. **a**
2. **B**
3. **A**

Questions

1. How many mappings can an input have?
2. What is the replay system?
3. What is the `User Index` of a saved game object?
4. What is culling?

Key terms

- **UI**: User interface
- **Mapping**: A hardware input matched to an input in the project settings
- **Replication**: The transferal of data over multiplayer

Chapter 11
Audio and Particles

Unreal Engine 5 uses a flexible VFX system to generate visual effects at runtime known, as Niagara, which we will go through in this chapter. Niagara has some default templates that provide a range of basic to complicated visual effects and are ideal for using as base-effect to make more complicated ones, and prototyping. A technical artist or VFX developer can use Niagara's functions and procedure to generate a flexible particle system and then programmers can use it and render / trigger it by using game parameters. Like shaders, Niagara system is running on each frame and can use both CPU and GPU for running the visual effect on screen.

At the same time, the engine has an audio system that provides complete control over a **Digital Signal Processing** (**DSP**) graph for the generation of sound sources. Audio programmers can customize the audio, reuse sound graph as feedback, import external audio files, and use a separate editor for sound design and programming. This sound system is known as MetaSound, and we will cover how to use and program this system in this chapter.

Structure

In this chapter, we will discuss the following topics:

- Using Niagara

- Using audio
    - o Using MetaSound in blueprint

Objectives

By the end of this chapter, you will have an understanding of how to use Niagara to create particle systems. We will go through simulating a rain effect and then connect it to an actor blueprint in the scene. We will also learn how to assign and use parameters to control a particle system from other objects. Additionally, we will mix two Niagara particle systems at the end to show how you can simplify them as one.

The next part of the chapter is dedicated to learning MetaSound, which is an Unreal Engine 5 feature to program and create audio effects in game. MetaSound provides advanced support compared to older versions of Unreal Engine, to use Wave audio files and dynamically change them based on user parameters. MetaSound has its own editor and uses blueprint to generate code like other actors, but the way audio code works is a bit different. We will go through a simple play and loop sound functionality and explain different approaches to make it. At the end, you will learn essential basic skills on how to invoke triggers in MetaSound and make internal feedback to control the audio.

Keep in mind that both VFX and Audio are huge and expensive topics in game design and programming with Unreal Engine; you may require more research and practice in this area. Making sample project of each learning curve is recommended when you deal with Niagara and MetaSound.

Using Niagara

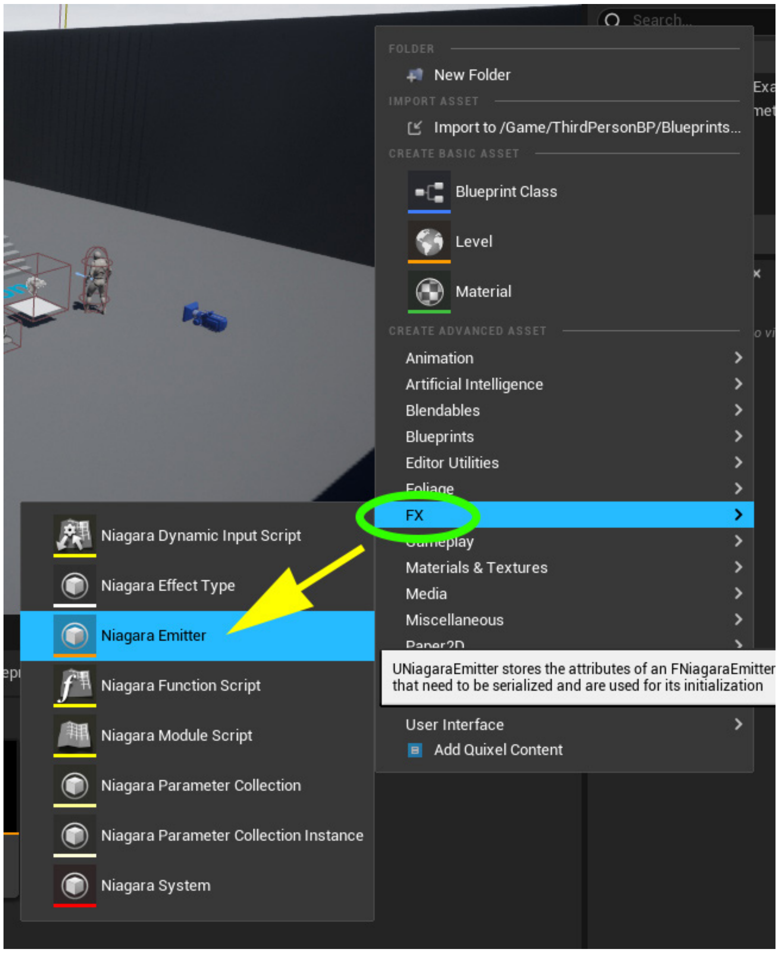
The Niagara is a particle system generator used by Unreal Engine to create/render and run VFX effects in game. Each particle system is a program that runs an infinite *loop*. Inside this loop, there are a number of tasks/modules that hire CPU and GPU (graphic card) to *calculate* and *render* the particle system's *effect* in the game world. This process is expensive for the engine in cases of memory usage and processing tasks. Imagine that you have 10 particle systems in the scene; this means you have 10 infinite loops that are running each frame to make all the particle systems up and running on each frame. This may work fine on a gaming PC machine, but your application will experience lag and even freeze in mobile phones with weaker graphic processing units and limited memory for system. There are a number of solutions to avoid this. User can deactivate particle systems that are not in player's view which leads to having less process and memory usage in the background. The engine also has a mechanism known as **Culling** which works similar, and it deactivates any process (like particle system) that are not in player field of view.

Another method is to combine a couple of effect as one particle system and changing it by passing its parameters. Also, dynamically deleting or disabling particle system components that are not in use will help increase performance.

Making Niagara emitter

The first step to establishing a particle system by using Niagara in Unreal Engine 5 is to simply make a Niagara emitter. Follow these steps to do so:

1. Navigate to the **`blueprint`** folder in content browser, right-click and click on **`Niagara Emitter`** from the VFX list, as shown in *Figure 11.1*:

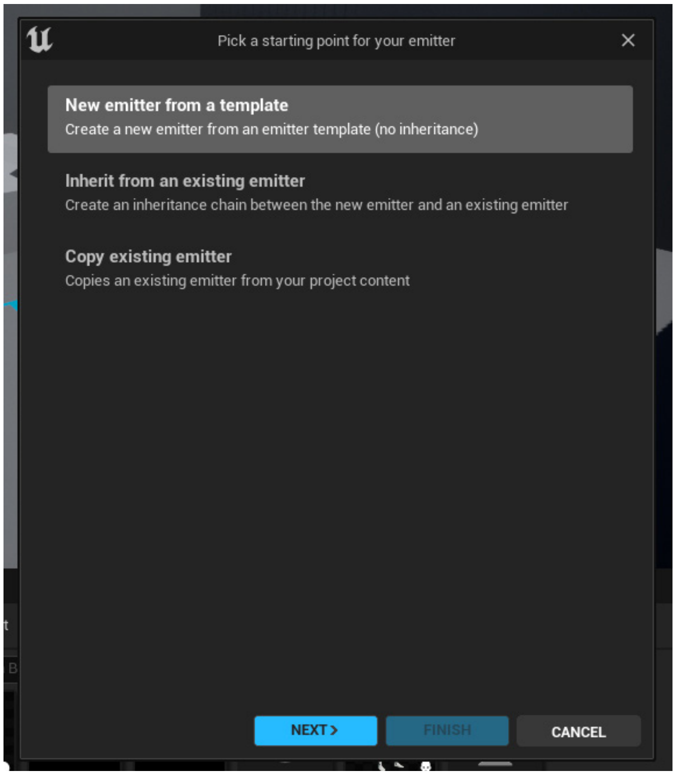


UNiagaraEmitter stores the attributes of an FNiagaraEmitter that need to be serialized and are used for its initialization

***Figure 11.1**: Make new Niagara Emitter particle system object in content browser*

2. The engine will ask the starting point of the emitter, as shown in *Figure 11.2*, select **`New emitter`**, and click on the **`Next`** button. The starting point of each

emitter is designed to use other emitters when the user likes to combine them or customize previous ones as a new emitter:



Pick a starting point for your emitter

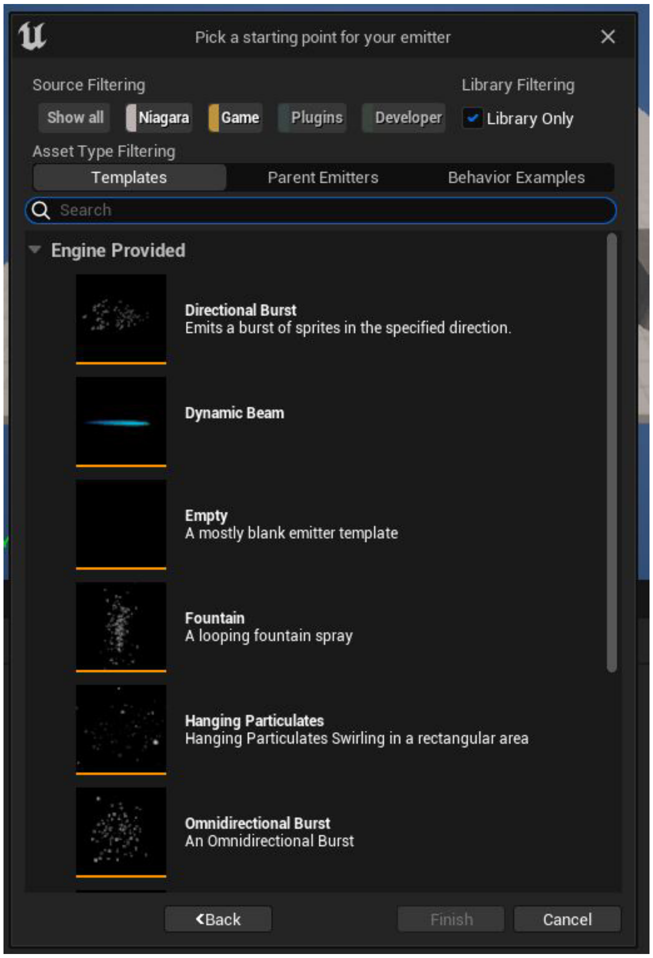
New emitter from a template
Create a new emitter from an emitter template (no inheritance)

Inherit from an existing emitter
Create an inheritance chain between the new emitter and an existing emitter

Copy existing emitter
Copies an existing emitter from your project content

***Figure 11.2**: Pick starting point windows*

3. Now, we need to choose a template for the emitter. From the **`Templates`** tab, select the **`Empty`** template, as shown in *Figure 11.3*. The **`Empty`** template contains basic preferences for making a Niagara particle system. The other tabs are advanced topics, and we will not dig into them in this chapter. Click on the **`Finish`** button and then rename it to **`NE_01`**:



Directional Burst
Emits a burst of sprites in the specified direction.

Dynamic Beam

Empty
A mostly blank emitter template

Fountain
A looping fountain spray

Hanging Particulates
Hanging Particulates Swirling in a rectangular area

Omnidirectional Burst
An Omnidirectional Burst

***Figure 11.3**: List of default templates of Niagara emitter*

4. The next step is to customize the Niagara emitter. Double-click on **`NE_01`** Niagara emitter, and the engine will navigate to Niagara editor, as shown in *Figure 11.4*. Locate the **`System Overview`** tab on top, and you will find an orange block named **`NE_01`**. This block, which is known as **`Niagara Overview Node`**, is responsible to run and render your particle system code (which is an infinite *loop*). We will go through a simple setup to simulate *Rain* effect by using it. The editor has a **`Preview`** tab that renders **`Niagara`**

Overview Node in real time and is the output of the block. The **Selection** tab shows the properties of selected items in **Niagara Overview Node**:

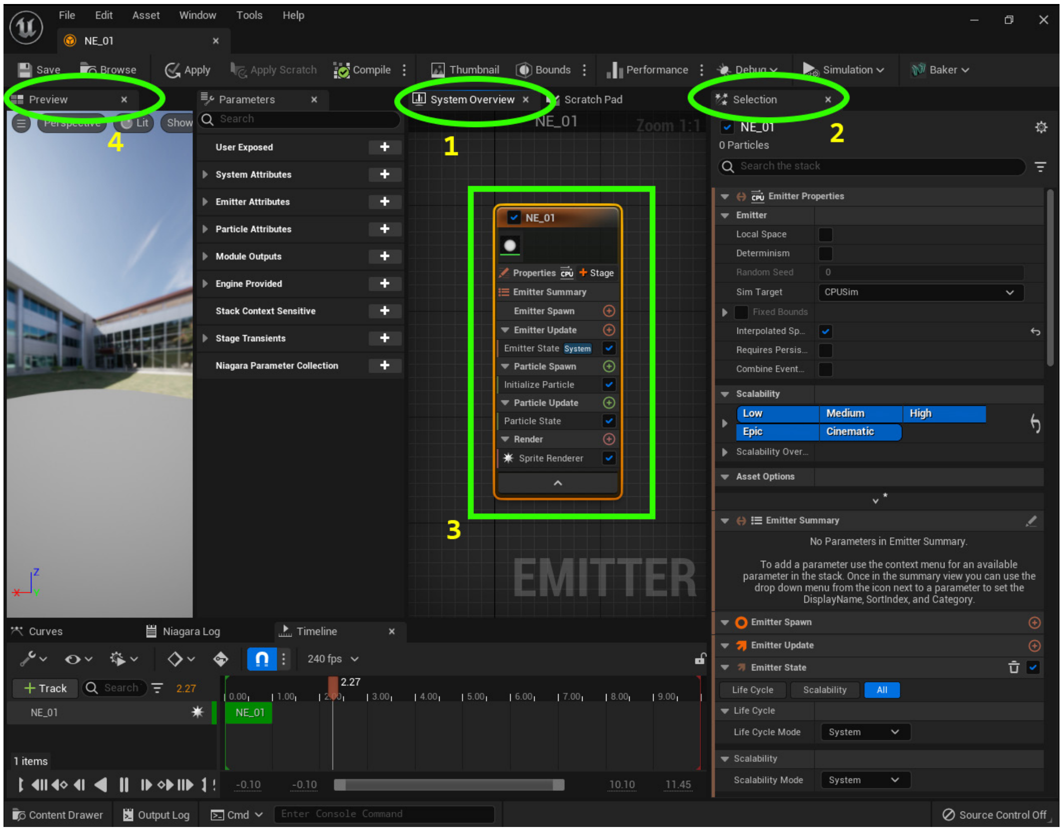


Figure 11.4: The Niagara editor 1- System Overview tab shows the "Niagara Overview Node" and its modules and renderers. 2- Selection tab is designed to show properties of each selected item in "Niagara Overview Node". 3- The actual "Niagara Overview Node" of "Empty" emitter template in Unreal Engine 5. 4- Preview tan renders the output of "Niagara Overview Node" in real-time.

5. The **NE_01** node has a number of main procedures that are executed from *top* to *down* on each frame. Each procedure can run a number of functions known as **Module**, **Renderer**, and **Event**, which are codes and functions, designed to provide services, effects, and communications with other game objects. Let's begin with **Emitter Update procedure**, as shown in *Figure 11.5*. Click on the small circular icon, type spawn, and select the **Spawn Rate** module from the list:

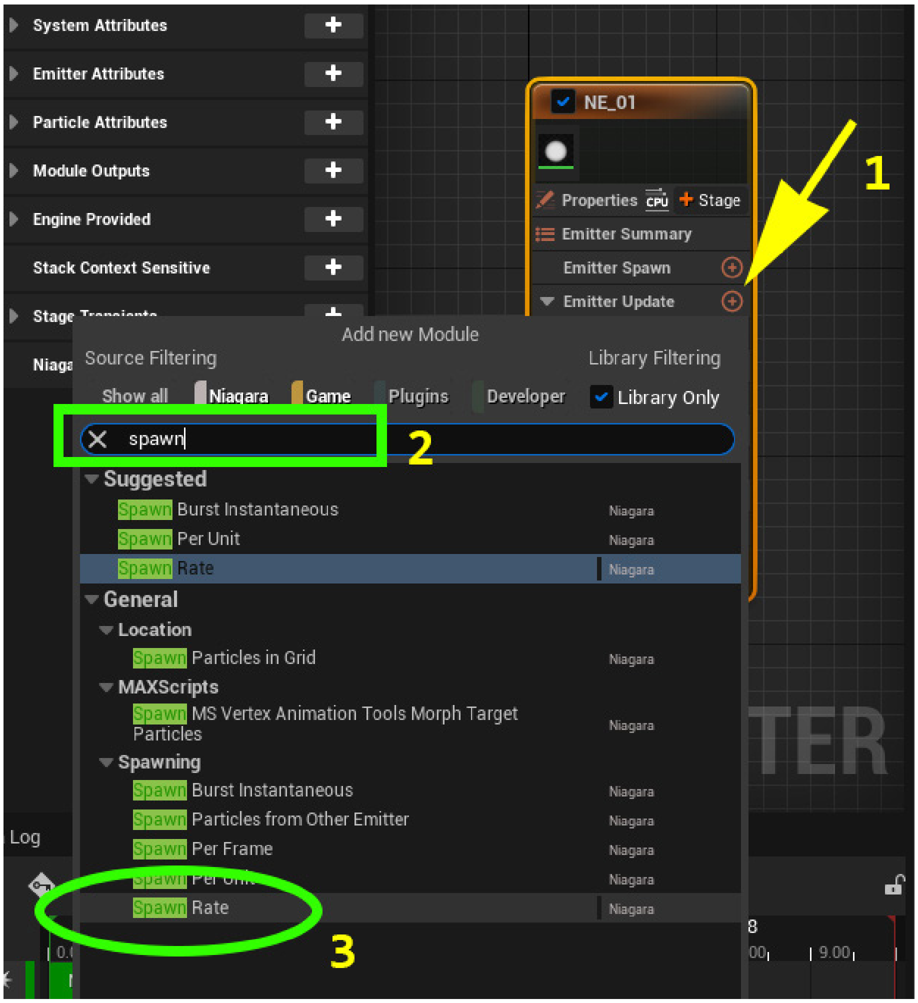


Figure 11.5: *First click on "Emitter Update" then: 1- Click on little circular icon. 2- Then immediately type "spawn", the engine automatically open and search for the keyword in a drop-down list of Niagara modules in the engine. 3- Select "Spawn Rate" module*

6. The module will be added as the last module on this **`Emitter Update`**. Click on it and then change **`Spawn Rate`** to **`64`**, as shown in *Figure 11.6*. The **`Preview`**

tab immediately updates with the latest change in **Niagara Overview Node**. We customize the total number of a sprite image to **64** by using **Spawn Rate**:

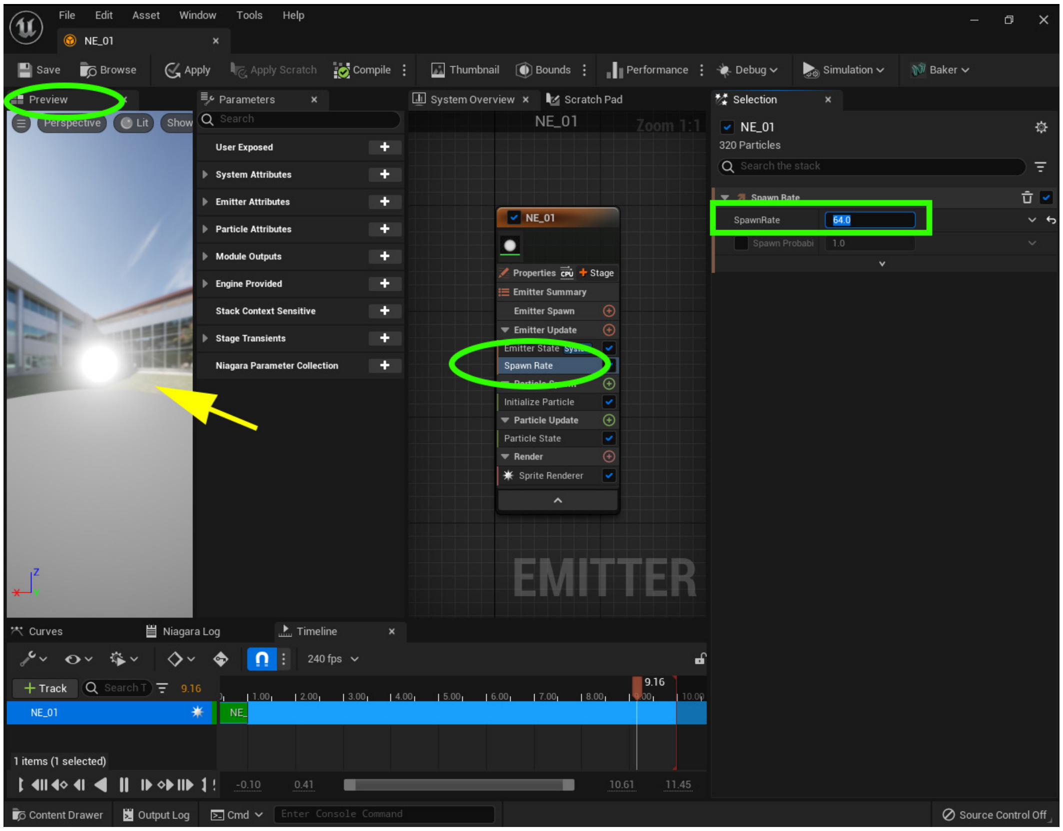
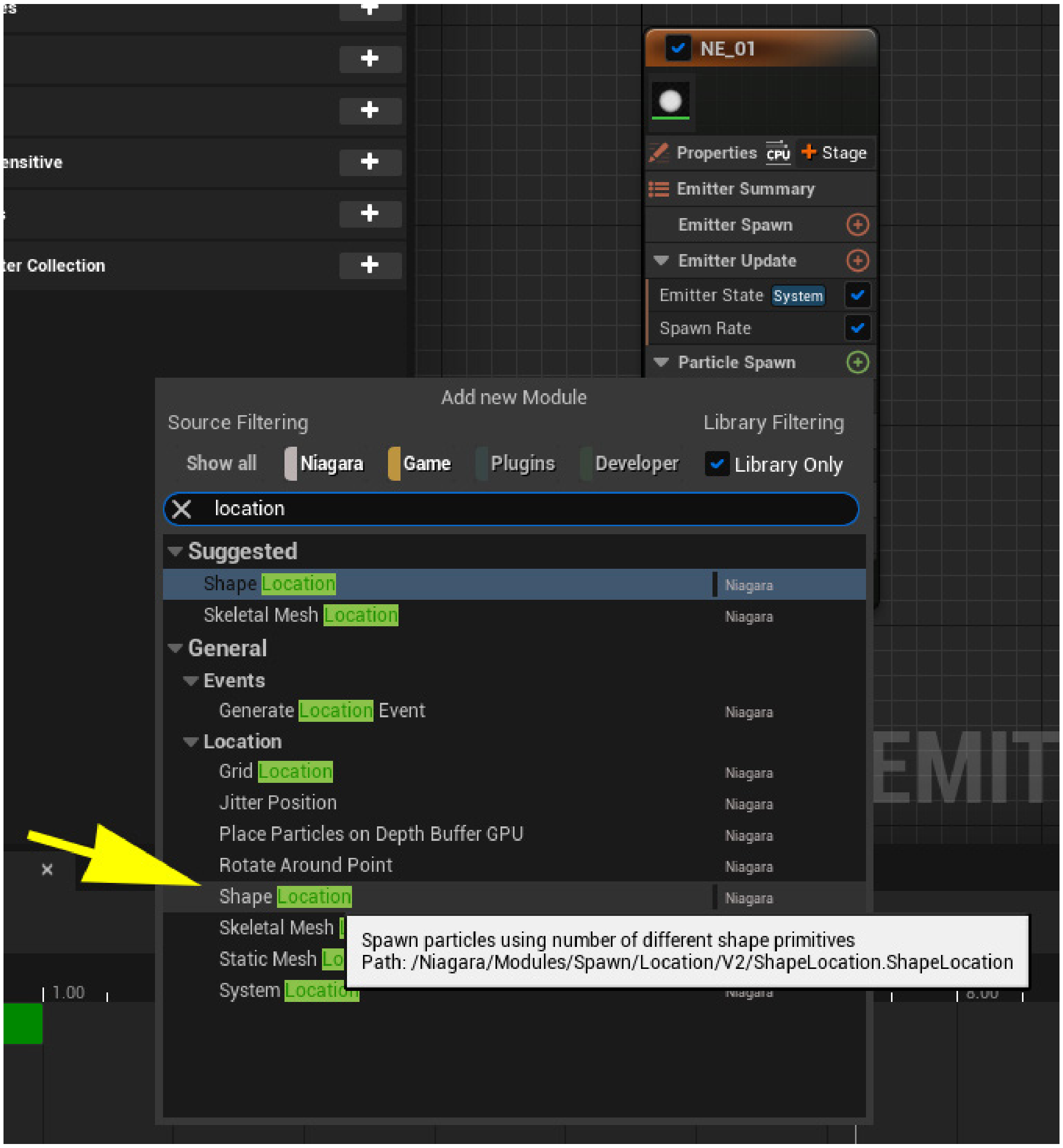


Figure 11.6: *The "Spawn Rate" module will customize number of spawned instances in a Niagara emitter. In this image, 64 instances of a sprite object are spawned at the same location, so the preview tab shows a shiny spot.*

7. Now, select **Particle Spawn** and then click on the small circular icon on it; then, type **location** and select **Shape Location**, as shown in *Figure 11.7*:

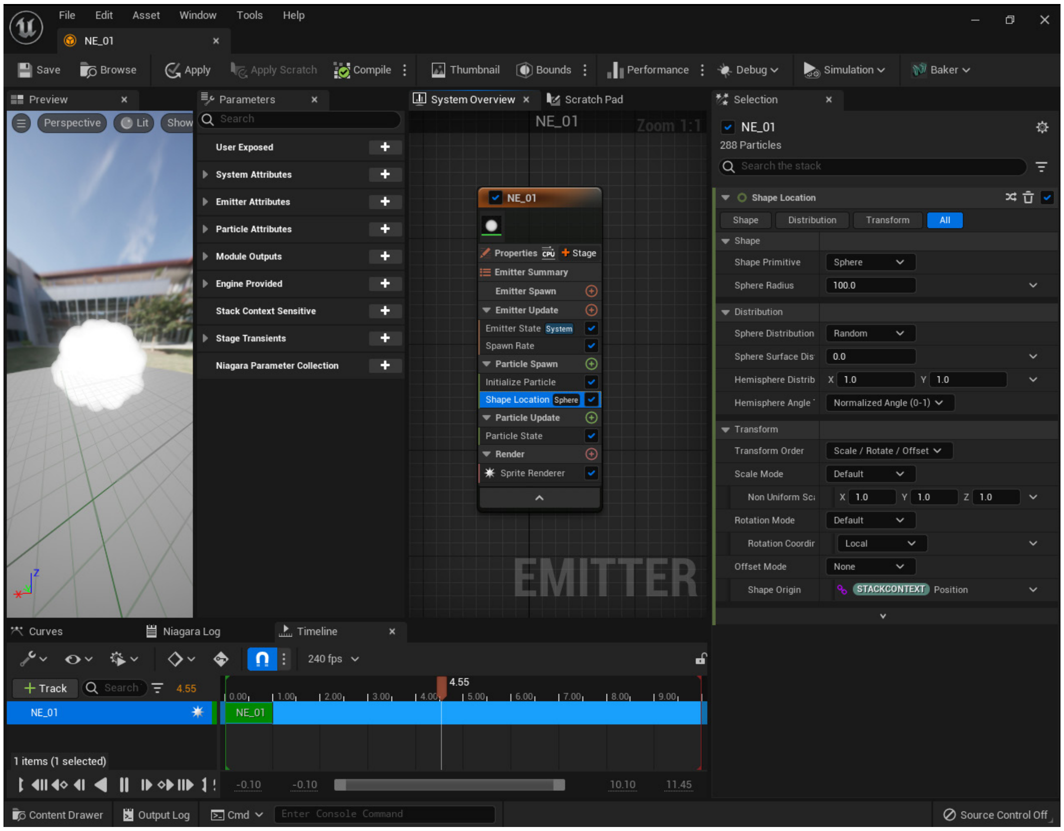


Spawn particles using number of different shape primitives
Path: /Niagara/Modules/Spawn/Location/V2/ShapeLocation.ShapeLocation

Figure 11.7: Remember the "Niagara Overview Node" executes each module form "Top" to "Down". The "Particle Spawn" has numbers of modules to set the location of spawned particle which are spawned by "Spawn Rate" module on top of them. Click on the little circular icon and immediately type "location" and select "Sphere location" from the list.

8. Then, the engine will immediately render 64 instances of a sprite image in a spherical 3D space in **`Preview`** tab, as shown in *Figure 11.8*. The Niagara particle system first runs **`Spawn Rate`** from **`Emitter Update`**, which spawn **`64`** instances of sprite to the scene; then, in the next step, relocate them by running Shape Location module from **`Particle Spawn`**, which results in

rendering all sprite images in a spherical 3D coordination that you can see in the preview tab. Keep in mind that the execution order of modules is from top to down:



NE_01
288 Particles

Shape Location

Shape Primitive	Sphere
Sphere Radius	100.0
Sphere Distribution	Random
Sphere Surface Dis	0.0
Hemisphere Distrib	X 1.0 Y 1.0
Hemisphere Angle	Normalized Angle (0-1)
Transform Order	Scale / Rotate / Offset
Scale Mode	Default
Non Uniform Sc	X 1.0 Y 1.0 Z 1.0
Rotation Mode	Default
Rotation Coordir	Local
Offset Mode	None
Shape Origin	STACKCONTEXT Position

Figure 11.8: A simple Niagara emitter with spherical location in preview tab

9. Now, let's make it move like rain. There are a number of methods here to move these sprites. Let's try gravity force, by following these steps:

    a. Click on **`Particle Update`**, which has a small circular icon.

    b. Type gravity and select the **`Gravity Force`** module from the list.

    c. Niagara may pop-up an error message, as shown in *Figure 11.9*.

    d. Press **`Fix issue`** and the engine will fix the conflict. (This option works for a majority of the issues on making simple emitters in Niagara.):

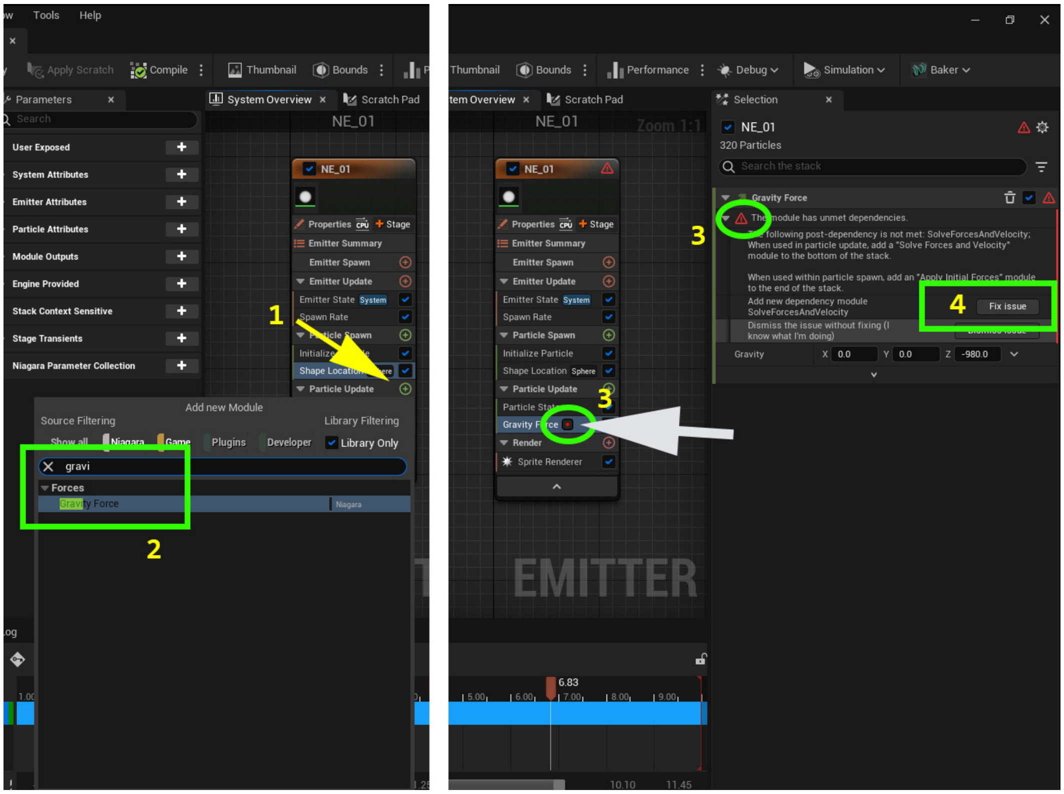

1
2
3
3
4
gravi
Forces
Gravity Force
Add new Module
Fix issue
The module has unmet dependencies.
Gravity X 0.0 Y 0.0 Z -980.0

***Figure 11.9**: Sometimes when you add a module in Niagara, an error message comes up. In this example, we like to add Gravity Force module to the NS_01 emitter.*

10. After fixing the issue, select the **`Gravity Force`** module and check the **`Selection`** tab. The engine automatically assigns **`-980`** on Z-axis as gravity value, which matches with earth gravity in the real world, and the result is falling sprites (like snow) in the **`Preview`** tab, as shown in *Figure 11.10:*

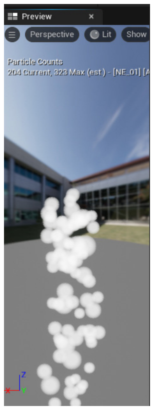


***Figure 11.10**: Falling effect on sprites by using Gravity Force module*

11. The final step is making this particle system *visually* similar to actual *rain* in nature. Click on and select **`Initialize Particle`** from **`Particle Spawn`** and change the **`Sprite Size`** property to **`Non-Uniform`** and change the value of X and Y to 4 and 54, as shown in Figure *11.11*:

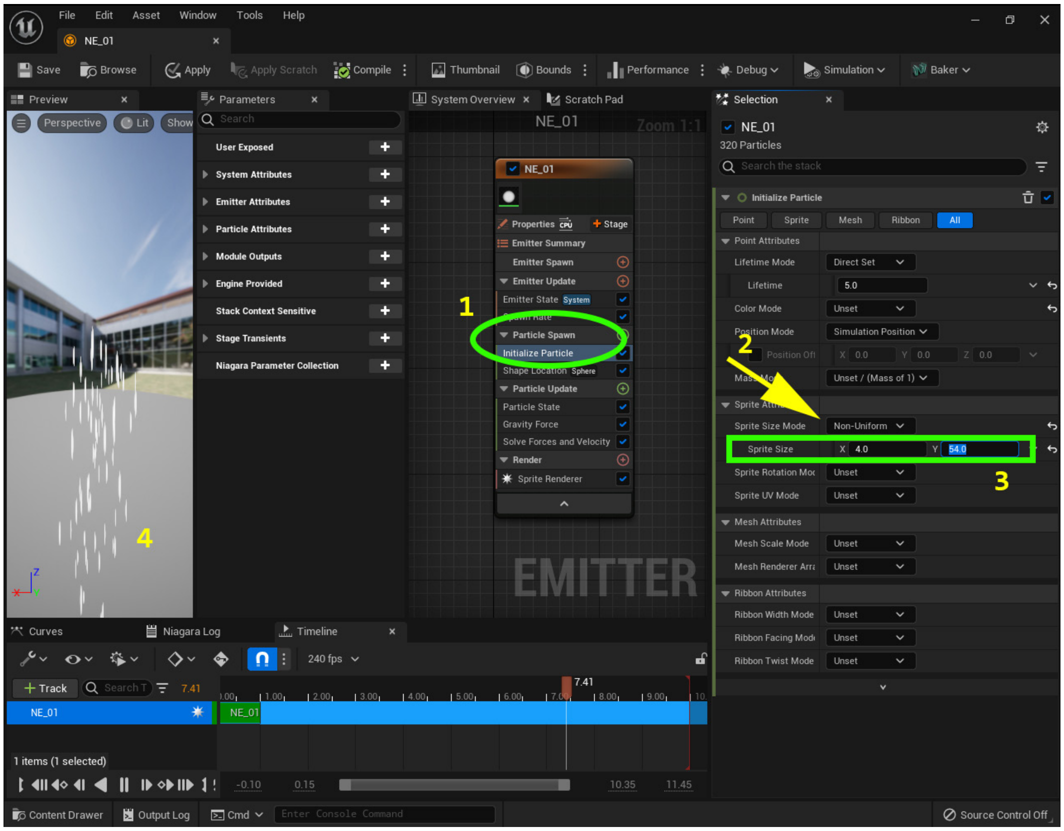


Figure 11.11*: Change particle shape to rain's drop. 1- Select Initialize Particle. 2-Locate Sprite Size Mode property in Selection tab and change it from "Unset" to "Non-Uniform". In general, "Non-Uniform "option allows users to customize values in the selected property. 3- Change dimensions of sprite by using Sprite Size property. 4- The engine automatically updates the result with your change in Preview tab.*

12. Now, select **`Sphere Location`** from **`Particle Spawn`** and change **`Sphere Radius`** to **`512`**. This way, you can increase the size of the sphere that the particles are spawned in. The result will look like light rain. Now, change it to **`64`** and the result looks different, which we like to use further in a blueprint actor. *Figure 11.12* shows the visual differences of using these values:

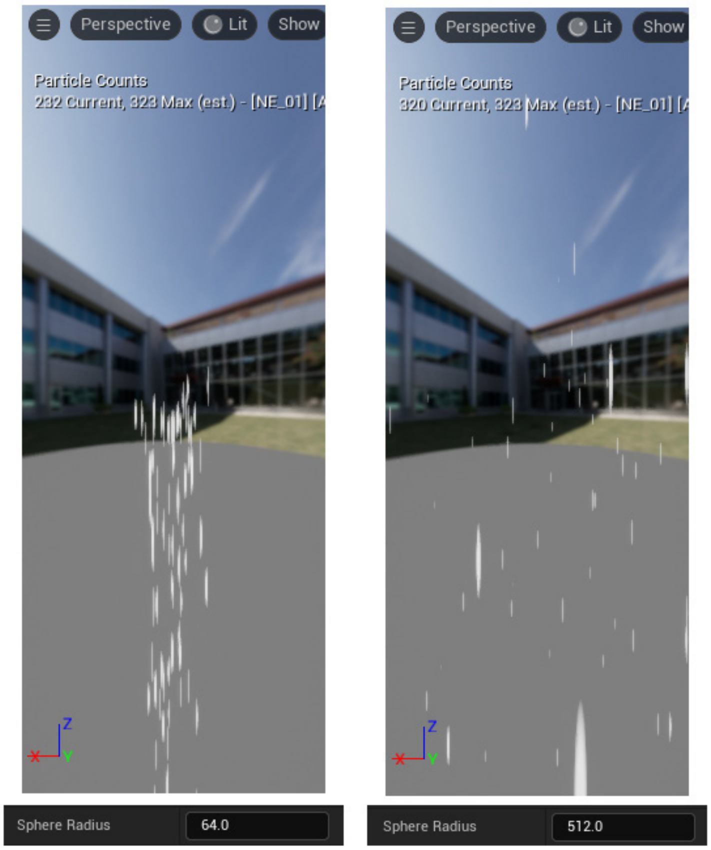


***Figure 11.12**: Different values on Sphere Radios property which is part of Sphere Location in Particle Spawn*

Don't forget to compile and save your emitter after any change (otherwise, you may experience unusual behavior in particle, especially when there are large number of modules involved in making particle).

Making Niagara system

At this level, we just made a simple Niagara emitter to simulate rain effect. To use this emitter inside game, we will make a **`Niagara System`** and add our **`Niagara Emitter`** to it, as shown in *Figure 11.13*:

1. Right-click in the **`Content browser`** and select **`Niagara System`** from FX:

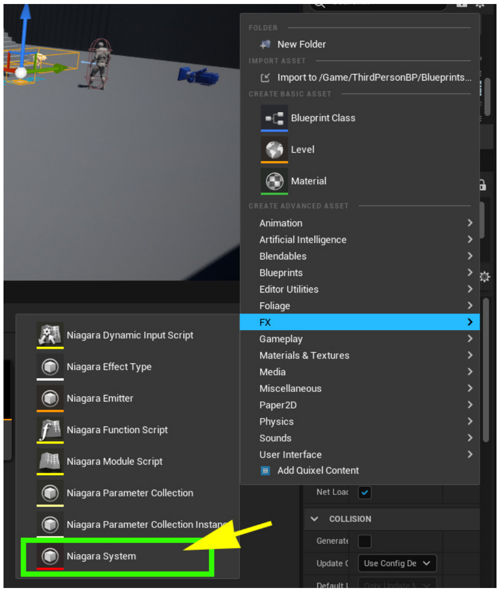
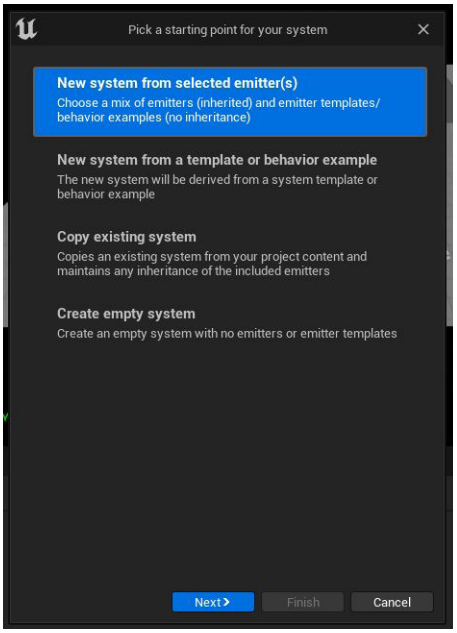


***Figure 11.13**: Create Niagara System from FX menu*

2. The engine automatically opens a dialog box, as shown in *Figure 11.14*; select **`New system from selected emitter(s)`** and press **`Next`**:


Pick a starting point for your system

New system from selected emitter(s)
Choose a mix of emitters (inherited) and emitter templates/ behavior examples (no inheritance)

New system from a template or behavior example
The new system will be derived from a system template or behavior example

Copy existing system
Copies an existing system from your project content and maintains any inheritance of the included emitters

Create empty system
Create an empty system with no emitters or emitter templates

***Figure 11.14**: Select starting point windows for Niagara System. Like Niagara Emitter, users are allowed to use previous systems to make custom systems or create new one.*

3. On the next page, first locate and select **`NE_01`** from the **`Parent Emitters`** panel, and then click on the **+** button, as shown in *Figure 11.15*. This way, the engine will add the **`NE_01`** emitter to our **`Niagara System`** and will update it on each frame. Next, click on the **`Finish`** button and rename the new **`Niagara System`** as **`NS_01`**.

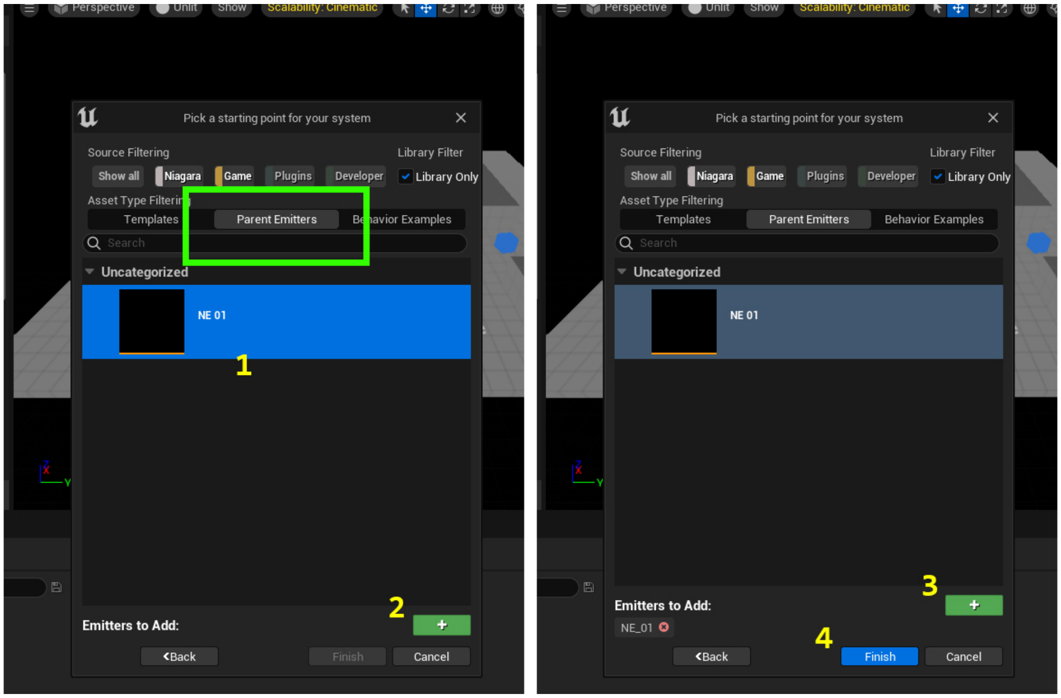


Figure 11.15: *To add Niagara Emitter template to Niagara System. 1- Select a Niagara Emitter template from the list in Parent Emitters panel. 2-Press + button. 3- The name of template should appear under "Emitters to Add:" title, then press Finish button*

4. Double-click on the **`NS_01`** Niagara system and the engine will automatically open its editor, as shown in *Figure 11.16*. In the editor, there is the **`NE_01 Niagara Overview`** node, which can get activated or deactivated by using the top-left tick box. Indeed, here we have a new box, which is responsible for managing dynamic features of this particle system. This box is always

active, and you can't switch it off like the **Niagara Overview** node. We will discuss this box, but let's add this particle system to the stage first:

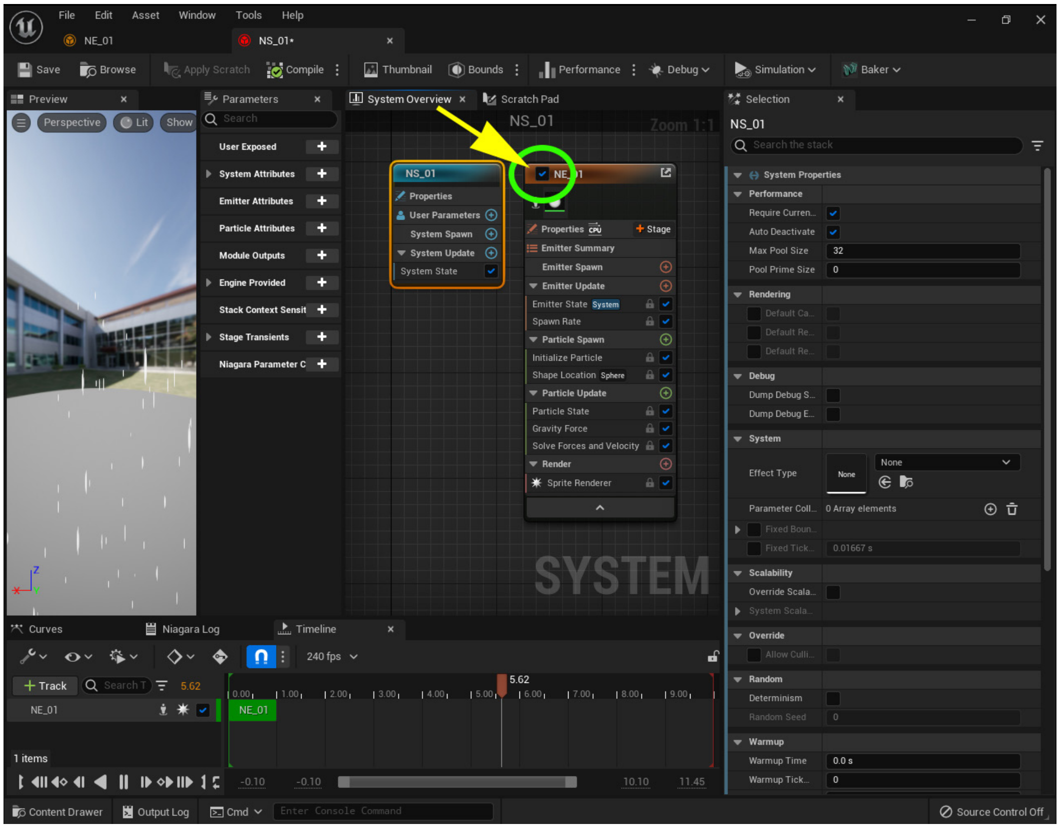
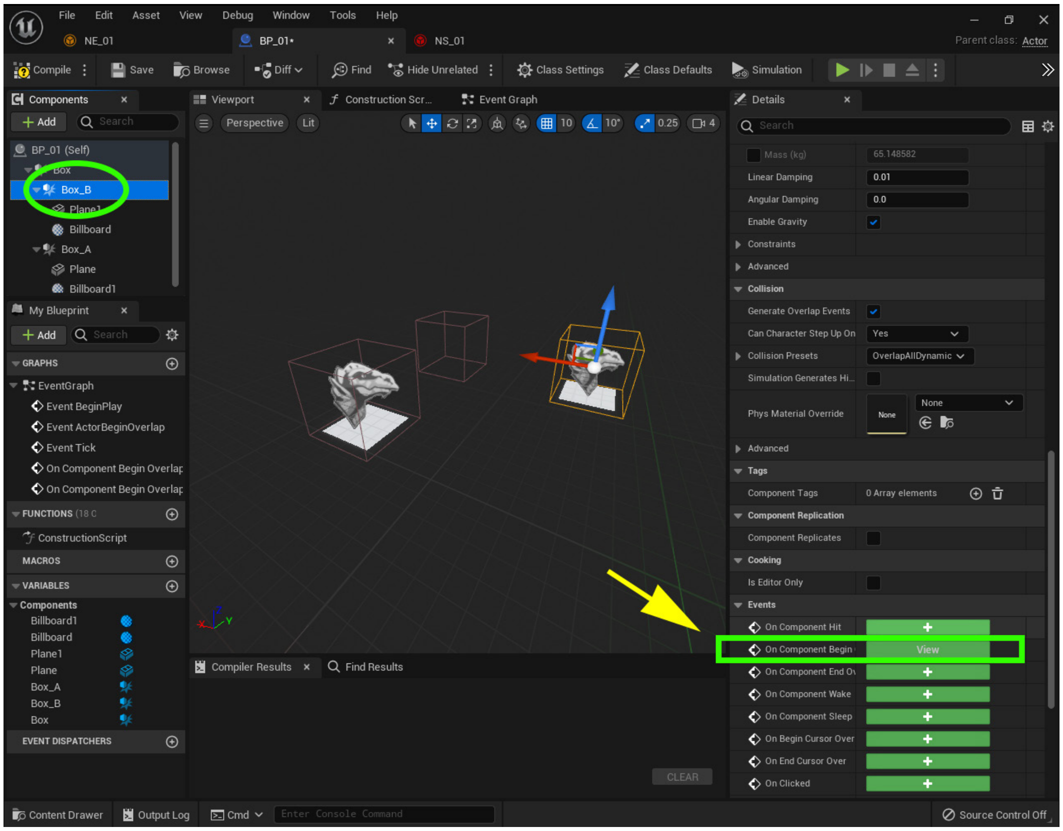


Figure 11.16: The Niagara System editor; each emitter has a tick box for activation

5. Referring to the previous chapters, make an actor blueprint called **BP_01** that has two collision boxes. Each collision box is parent of a **Plane** static mesh

component and a billboard component and is assigned to an overlapped event, as shown in *Figure 11.17*:



Mass (kg) 65.148582
Linear Damping 0.01
Angular Damping 0.0
Enable Gravity ✓
Constraints
Advanced
Collision
Generate Overlap Events ✓
Can Character Step Up On Yes
Collision Presets OverlapAllDynamic
Simulation Generates Hi...
Phys Material Override None
Advanced
Tags
Component Tags 0 Array elements
Component Replication
Component Replicates
Cooking
Is Editor Only

Figure 11.17: *An actor blueprint with 2 collision boxes; each event has assigned for overlapping*

6. Now, add a **`Niagara Particle System`** blueprint component to the root of your actor, as shown in *Figure 11.18*, and then assign **`NS_01`** to it. The component begins to play the particle system in the editor's **`Viewport`** tab:

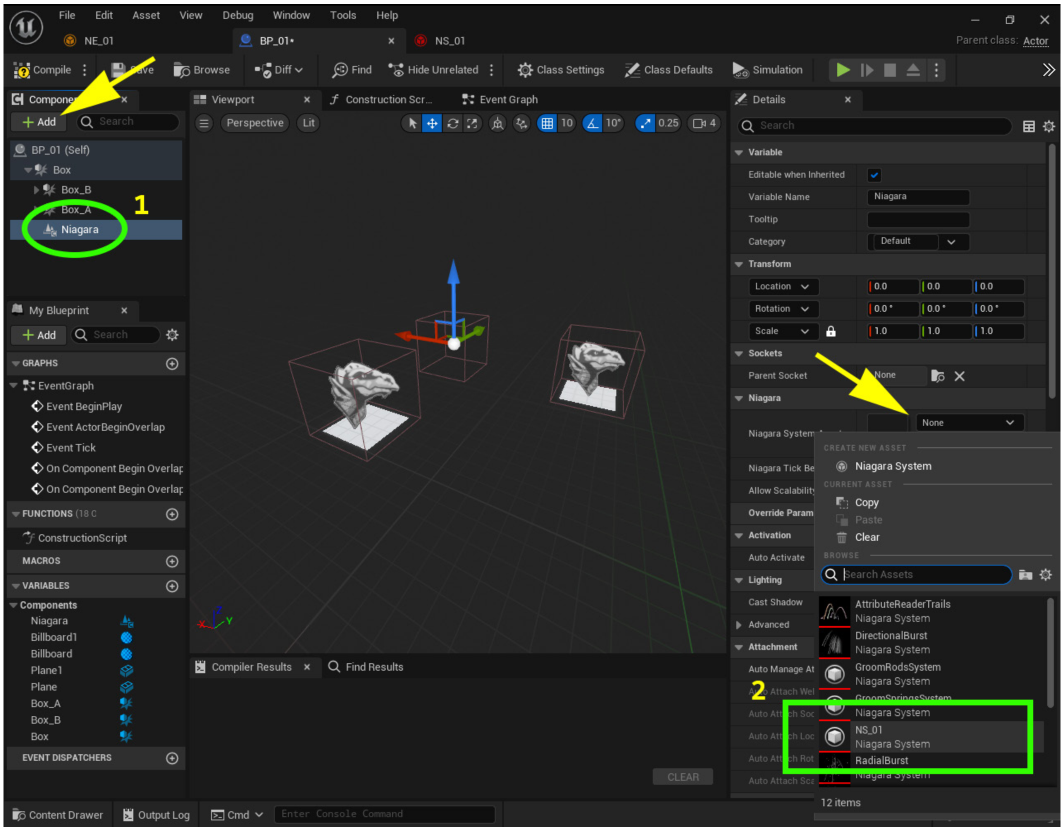
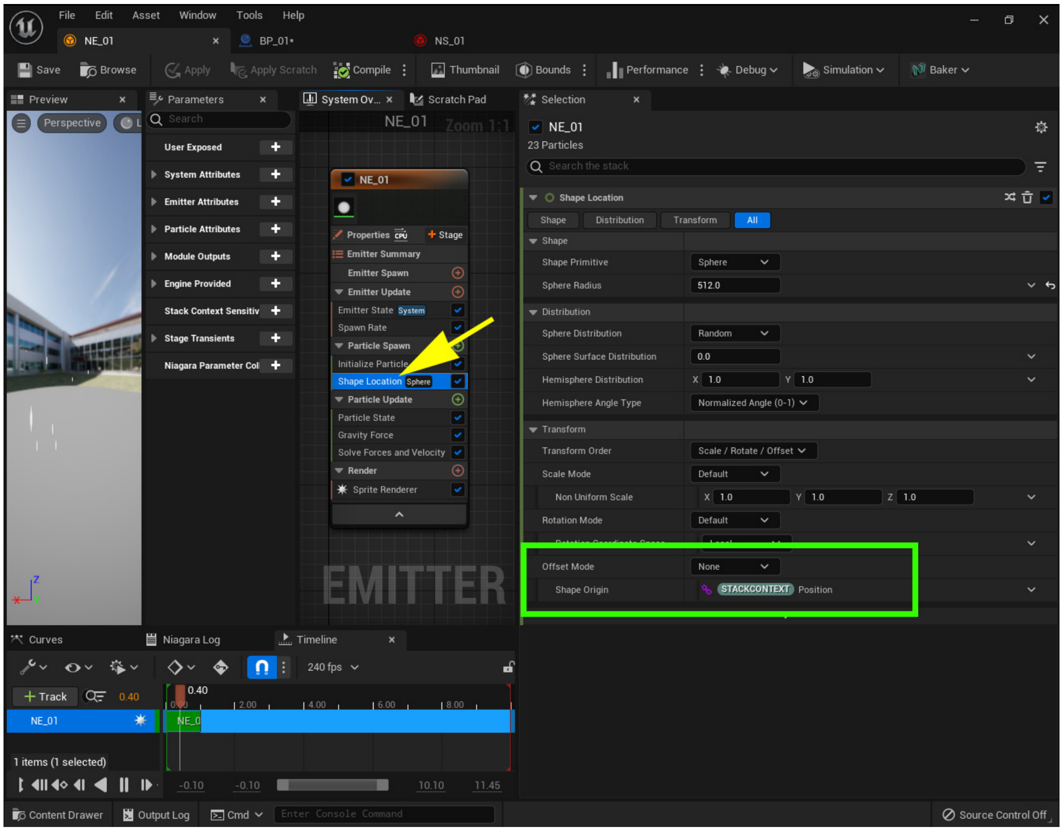


Figure 11.18: Click on the ADD button and select Niagara Particle System blueprint component from the drop-down list

Select the component you add, then locate Niagara System Asset property in Details panel, click on drop down menu and select "NS_01"

7. Now, imagine that we want to switch the location of this particle to collision boxes (in the **`BP_01`** actor) when we collide with them. For doing this, we must make a **`Parameter`** inside the **`Niagara System`** and use it as a location of our particle system. It sounds complicated, but Unreal Engine makes this very simple. First, open the **`NS_01`** editor by double-clicking on it and select **`Sphere Location`** in the **`NE_01`** box. As you see in selection tab, this module has a property known as **`Sphere Origin`**, as shown in *Figure 11.19*. This is a vector type property and will determine the location in which the **`Sphere Location`** module spawned the emitters:



NE_01
23 Particles

Shape Location

Shape Primitive	Sphere
Sphere Radius	512.0
Sphere Distribution	Random
Sphere Surface Distribution	0.0
Hemisphere Distribution	X 1.0 Y 1.0
Hemisphere Angle Type	Normalized Angle (0-1)
Transform Order	Scale / Rotate / Offset
Scale Mode	Default
Non Uniform Scale	X 1.0 Y 1.0 Z 1.0
Rotation Mode	Default
Offset Mode	None
Shape Origin	STACKCONTEXT Position

Figure 11.19: Click on Sphere Location module in Particle Spawn, and locate Sphere Origin in the Selection tab

8. We want to access this property from the blueprint **BP_01** in game and change it to control the location of emitter. For making this functionality, we must make and assign a parameter to this Niagara system and then control it from the **BP_01** actor blueprint. Click on the **+** button from the **User Exposed** dropdown box. Here, we need to make our parameter, so choose a Vector parameter from the **Common** category and name it **SPLocation**. Then, select **Shape Location** particle spawn procedure, click on the arrow icon in **Shape Origin**, and type **convert**. Now, select **Convert Vector** to **Position** from the **Dynamic Inputs** category. This function will add two parameters to your **Shape Origin** property. The **Input Position** property defines the received vector value to this function, so we need to connect it to our **SPLocation** parameterm which is also a vector. To do this, click on the little arrow and type user; as you see in *Figure 11.20*, your parameter name will appear in the

list, and by selecting it, this user defined parameter will be directly *connected* to the Niagara emitter:

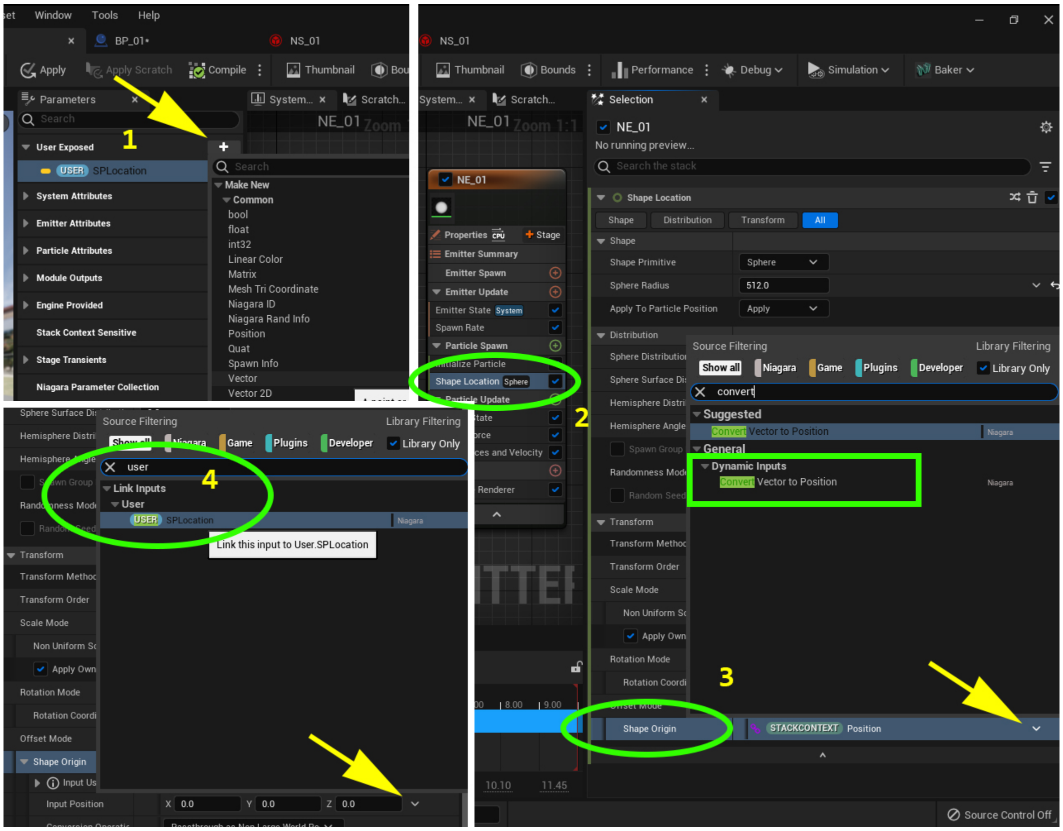


Figure 11.20: First step is assigning a parameter to the Niagara emitter. 1- Click on little + icon in User Exposed dropdown list and add a Vector parameter rwith name SPLocation. 2-Select Shape Location in NS_01 Emitte. 3-Then Locate Shape Origin property and assign Convert Vector to Position function to it. 4-Finaly, assign SPLocationas USER pinput to Input Position property.

9. Now, open **`NS_01`**, which is a **`Niagara System`** component, and click on **`User Parameters`** in the **`NS_01`** node. As *Figure 11.21* shows, you will have your custom parameter, **`SPLocation`**, listed here. If you do not have your parameter listed, you must compile and save your emitter (**`NE_01`**), then compile and save your **`Niagara System`** component (**`NS_01`**) and then open them again. If you have a very complex system and emitters, you may need to restart the editor to find your custom parameters listed in this property:

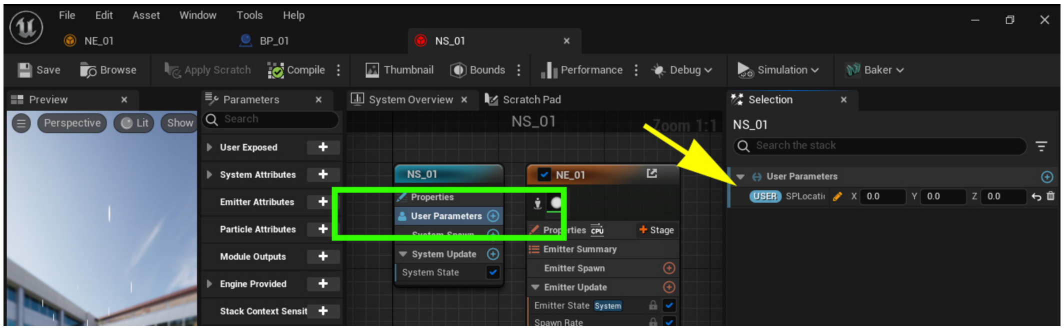


Figure 11.21: List of user-defined parameter in NS_01 node

10. The last step is to use the **`SPLocation`** parameter inside **`BP_01`** actor blueprint. Open **`BP_01`** to drag and drop an instance of Niagara particle system component into event graph. Then, assign the **`Set Vector Parameter`** blueprint function to the Niagara component, and in the **`Parameter Name`** field, add **`SPLocation`**, which is the name of the new parameter we just made. Keep in mind that the parameter name is case sensitive data and must be the same as what you edit in **`System Setting`** in the Niagara system. Now, assign world location of each collision box to **`Param`** field, as shown in *Figure 11.22*:

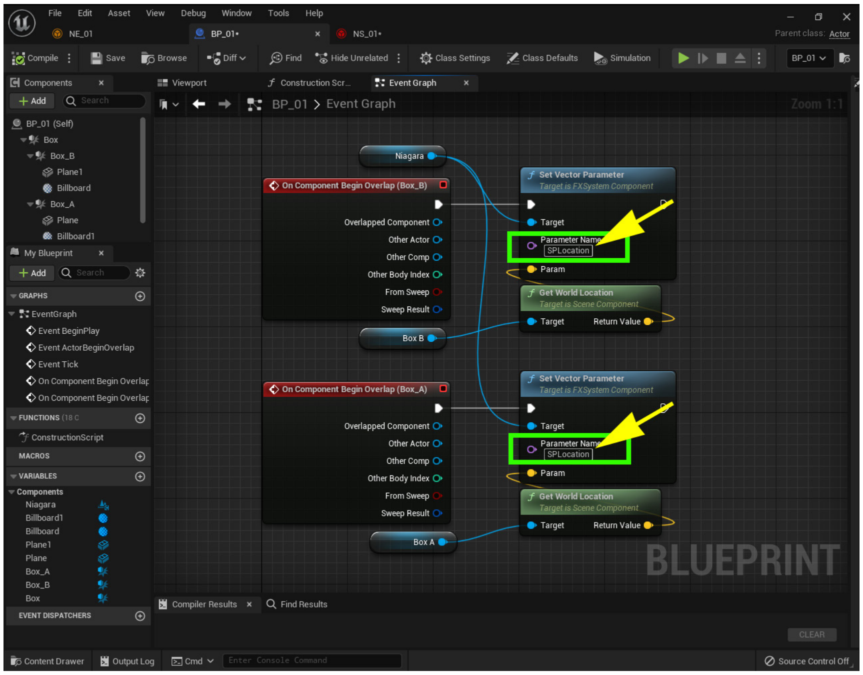


Figure 11.22: Using overlap event for collision box to relocate Niagara particle system. The particle system uses "SPLocation" parameter to communicate with this actor blueprint. Keep in mind the parameter's name is critical data and MUST be the same.

Place some **BP_01** actors on the scene, run the game, and walk into the collision boxes. As you see, on each collision, the particle system will relocate to that location. Now, make another Niagara Emitter called **NE_02** from an **Empty** template, as we did for **NE_01**, and follow these steps to customize it:

1. Open **NE_02** it and click on **Renderer**; add **Mesh Renderer** to it by using the little circular icon, as shown in *Figure 11.23*. You can activate and deactivate each module inside the **Niagara Emitter** by clicking on its check box, so deactivate **Sprite Renderer**:

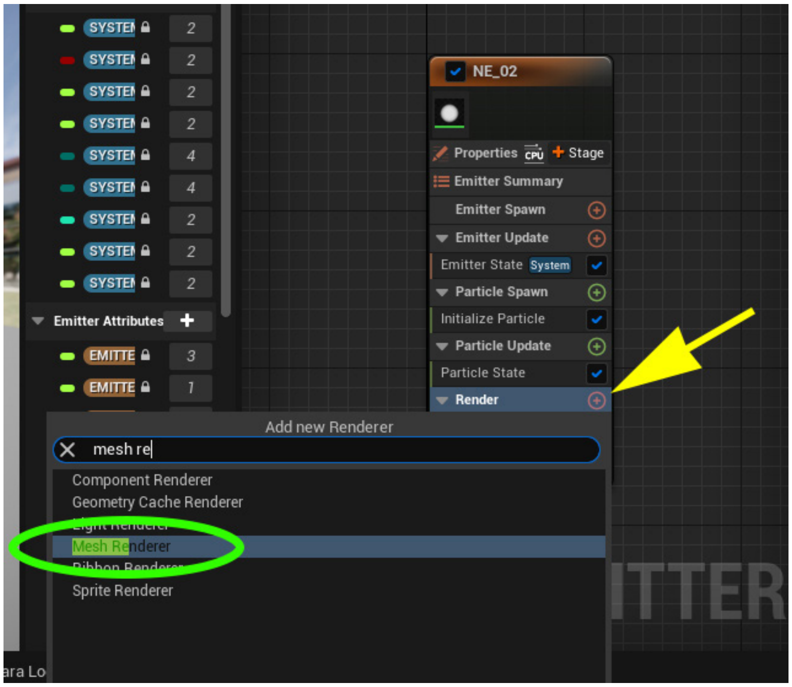


Figure 11.23*: Select Renderer, click on the small circular icon and then select Mesh renderer from the list.*

2. The Mesh Renderer renders mesh objects instead of sprite objects for the emitter, which is more expensive in case of processing, but has its own visual features. First, we need to assign a mesh to this renderer. Click on **Mesh Renderer**, then locate **Meshes** in the selection tab, press on the little circular icon there and then add **1M_Cube** static mesh to this module. Then, in the **Scale** field, change X and Y to .2 and Z to 4, as shown in Figure 11.24. You can also customize mesh materials in this module. Indeed, turn off the **Sprite Renderer** module because we don't like to render any sprite as we did for the **NS_01** emitter:

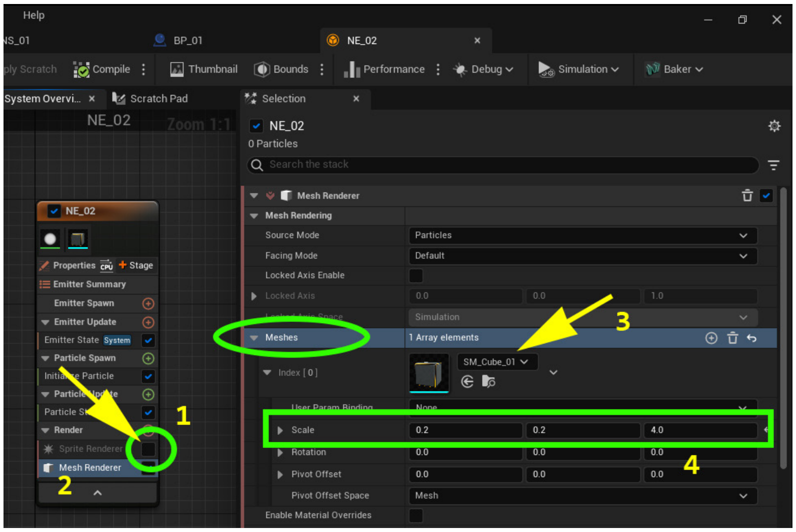


***Figure 11.24**: First, switch off the Sprite renderer (1) by unticking its module. Then, select the Mesh Renderer module (2) and from the list of its properties, locate Meshes. From here, choose "1M_Cube" (3) or any other static mesh there. Now, locate Scale property and change the scale values of x, y and z. (4).*

3. Now, like the previous particle, add the **`Spawn Rate`** module to **`Emitter Update`**, and set spawn rate property to **`64`**. Then, add the **`Shape Location`** module and set **`Shape`** to use **`Torus`**, as shown in *Figure 11.25*, and set its properties **`Large Radius`** to **`720`** and **`Handle Radius`** to **`64`**. This module spawns the emitter in different shapes:

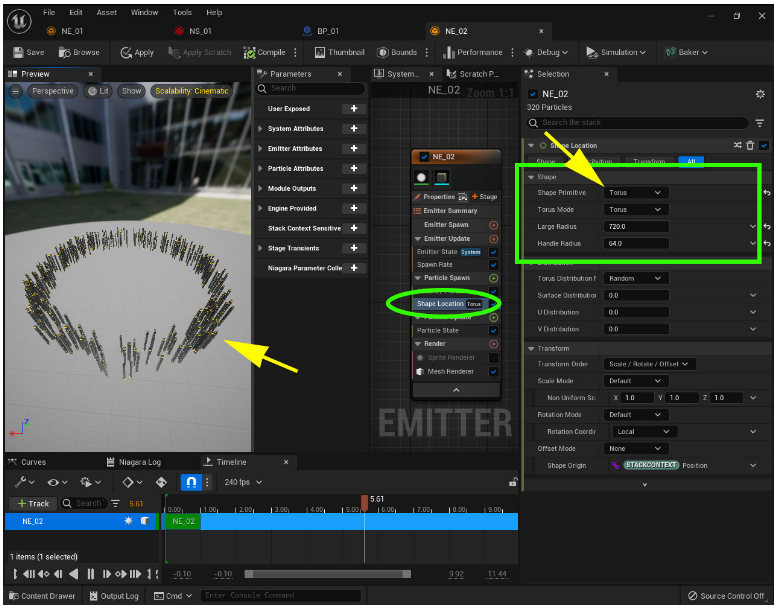


***Figure 11.25**: Using Torus on Shape Location module in Particle Spawn will shape a ring of spawned particle*

4. Now, select the **NS_01** Niagara system and open its editor by double-clicking on it. Right-click inside the **System Overview** tab, click on **Add Emitter** and choose **NE_02** from the list, as shown in *Figure 11.26*:

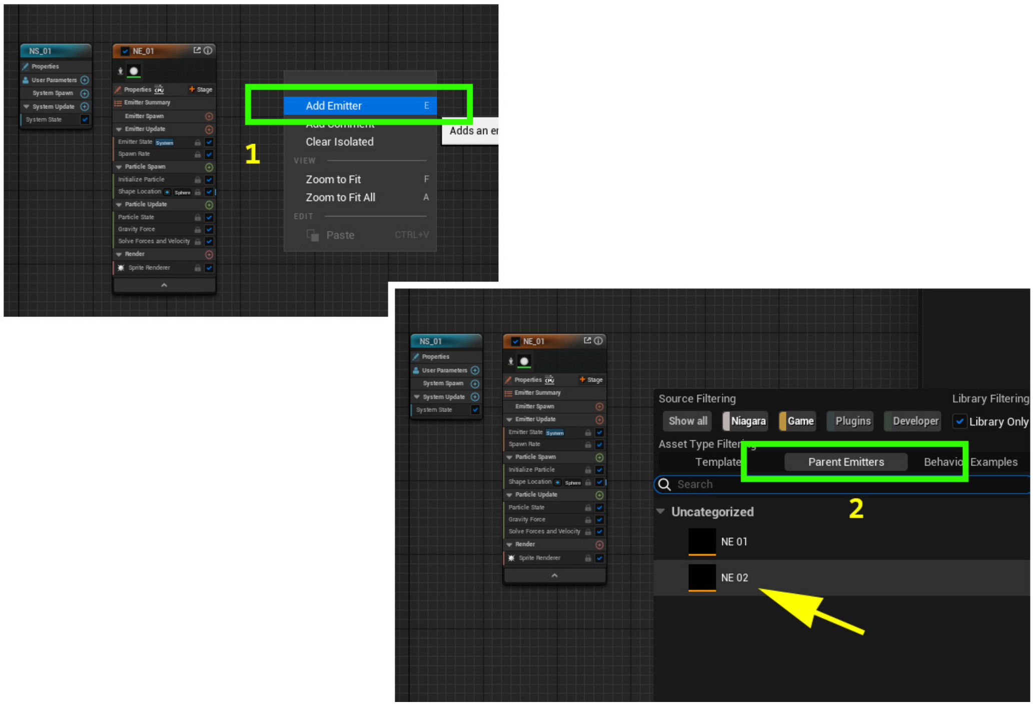


Figure 11.26*: Add new Niagara emitter to Niagara system*

5. The engine automatically adds the new emitter to the Niagara system and will show it on actor, as shown in *Figure 11.27*. This is how you combine a number of Niagara emitters with each other as one **Niagara System** particle system. Keep in mind to press the **Compile** and **Save** buttons after each update in your particle system objects when you edit them:

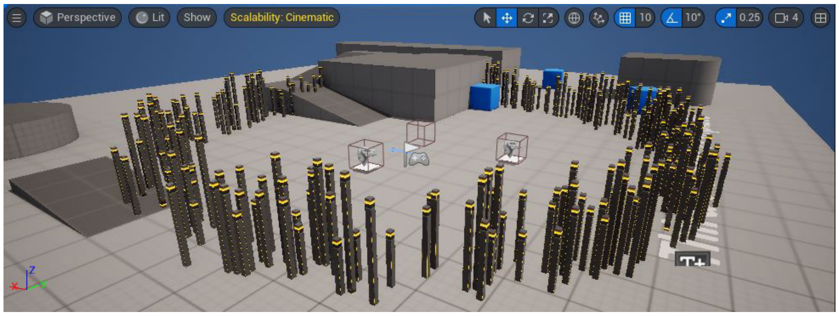


Figure 11.27*: Each Niagara System can combine and mix Niagara emitter as one particle system and presents the effect in the scene. In the example actor blueprint, NS_01 Niagara system, used NE_01 and NE_02 Niagara emitters to perform this VFX effect.*

Using audio

Having audio in any interactive application increases the quality of service that the application is developed for and satisfies the users when interacting with the application. Computer games are interactive applications and for some games, having a detailed and accurate audio in the game is essential. Games like VALORANT or Counter Strike are totally dependent on audio quality and accuracy when players experience game play.

The Unreal Engine 5 introduces an advanced method for audio programming, creates real time audio and processes audio data to develop applications like games. This audio system is known as **MetaSound**, and as compared to the previous versions of the engine, it revolutionized coding style for audio programming. The audio programmer has a complete list of audio functions and procedures to *render* and *play* audio in game. MetaSound provides audio designers with complete control over DSP graph generation for sound sources. They can use imported audio files or create sound in real-time by using internal signal generators of MetaSound to generate patterns and sequences of sound/music during game play.

There are a huge number of blueprints nodes to design and control DSP graph in real-time, which came with MetaSound. Audio designers and programmers can make a range of variable data types like integer, float, bool, and triggers as parameters inside the audio code. They can also make single or array of variables to save values or objects (like sounds) and monitor data inside MetaSound object during runtime. Here is a **`Print Log`** node that is extremely new and provides many useful features.

To import audio files into the engine, render your audio files in wave format **`.wav`** and then simply drag and drop them from your windows browser to content browser in engine. The engine automatically addresses the new audio asset, and you can use it anywhere in the code.

Making MetaSound

The MetaSound is a plugin that sometimes is not activated by default, so you need to activate MetaSound in Plugins and after restarting the engine, the MetaSound will be available for development. Then, similar to Niagara, create an audio system and assign variables and events to control it. Here are the steps to learn the details of this process:

1. Click on **Settings**, located on top-right of main editor, and then click on **Plugins** in the drop-down menu, as shown in *Figure 11.28*. The engine will automatically open the **Plugins** window.

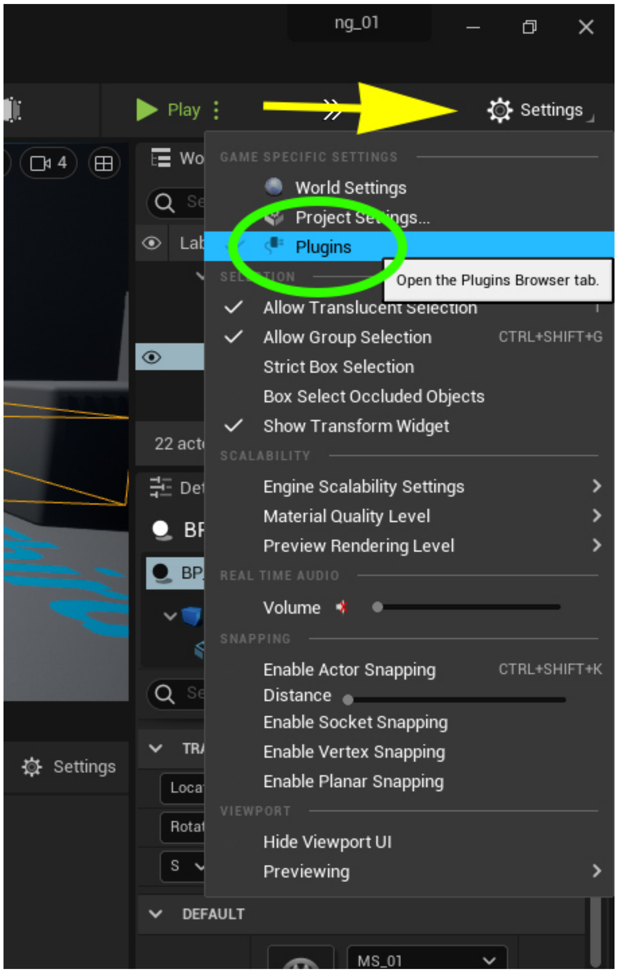
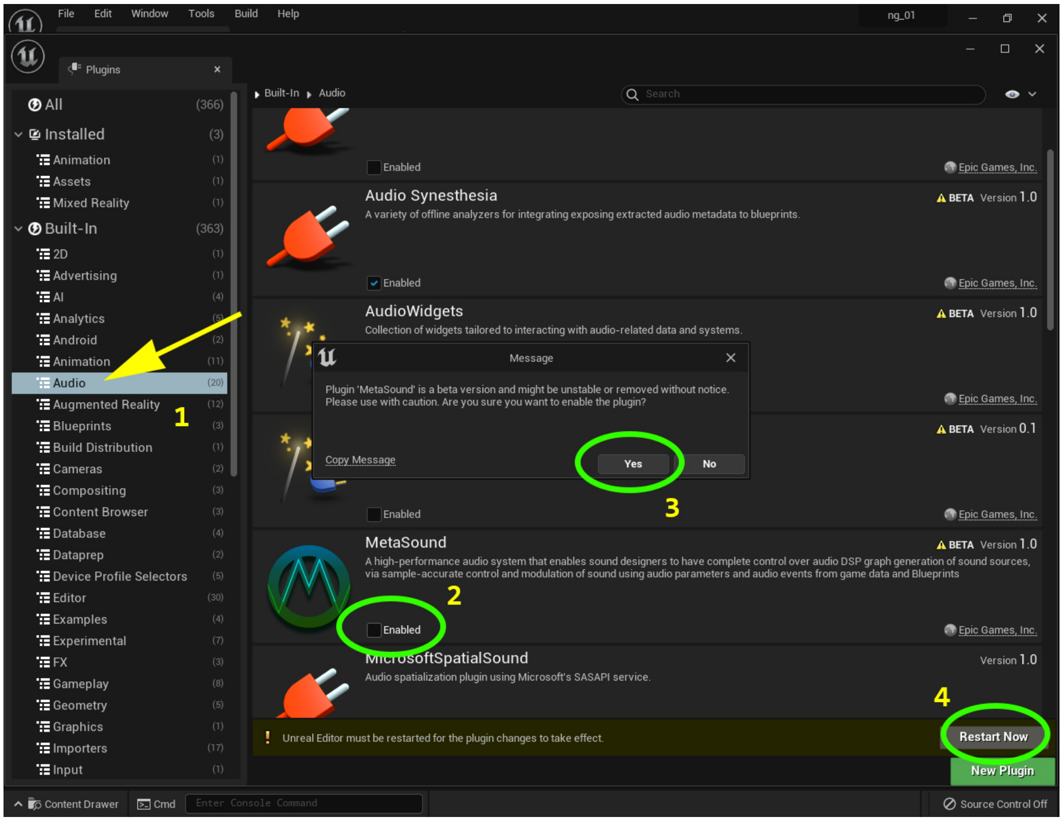


Figure 11.28*: In the main editor, click on "****Settings****" and then select on Plugins*

2. Locate **Audio** in the left bar of the plugin window and click on it; then, on the right, find the MetaSound plugin and click on **Enable** on its checkbox. The engine will open a message to confirm your selection, as shown in *Figure 11.29*. Click on **Yes**, and then restart your project:



Audio Synesthesia — A variety of offline analyzers for integrating exposing extracted audio metadata to blueprints.

AudioWidgets — Collection of widgets tailored to interacting with audio-related data and systems.

Message: Plugin 'MetaSound' is a beta version and might be unstable or removed without notice. Please use with caution. Are you sure you want to enable the plugin?

MetaSound — A high-performance audio system that enables sound designers to have complete control over audio DSP graph generation of sound sources, via sample-accurate control and modulation of sound using audio parameters and audio events from game data and Blueprints

MicrosoftSpatialSound — Audio spatialization plugin using Microsoft's SASAPI service.

Unreal Editor must be restarted for the plugin changes to take effect.

Figure 11.29*: To activate the MetaSound plugin, click on Audio category from left panel (1) and then find MetaSound and click on Enable (2). The engine will ask your confirmation from a Message window, click" Yes" button (3) and then click on "Restart Now" button (4). You must restart your project to acticate MetaSound.*

3. Now, inside content browser, right-click, and click on the **Sounds** category and select **MetaSound Source** from the blueprint list, as shown in *Figure*

11.30. The engine will make an instance of **`MetaSound`** and place it in content browser. Rename it to **`MS_01`** and then double-click on it to open its editor:

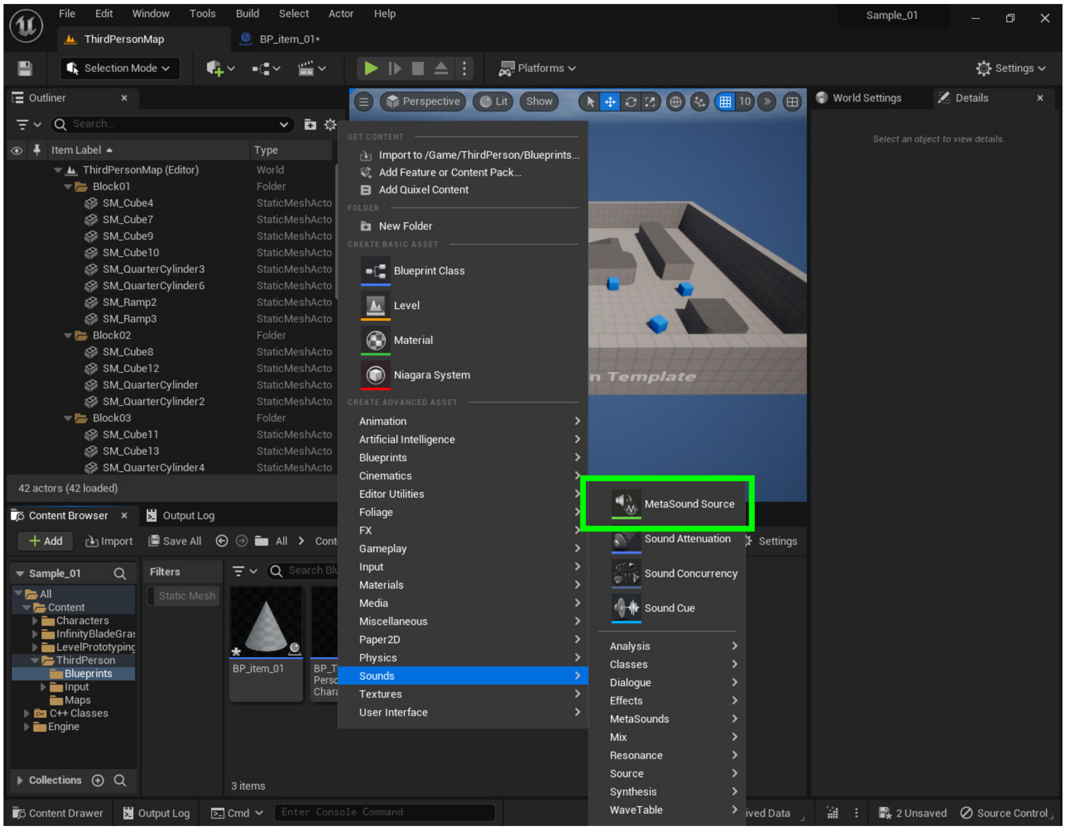


Figure 11.30: *"MetaSound Source" blueprint object is located in Sounds category*

4. The MetaSound editor is designed for audio programming in Unreal Engine 5. As you can see in *Figure 11.31*, there are three default nodes in the MetaSound editor known as Triggers. These nodes are listed in the left panels and are responsible to *activate* input and *send* output through MetaSound object to the game scene during runtime. To understand how these triggers operate, let's add a sound and play it. Right-click and type wave and then select **`Wave Player (Mono)`** from the list, as shown in *Figure 11.31*:

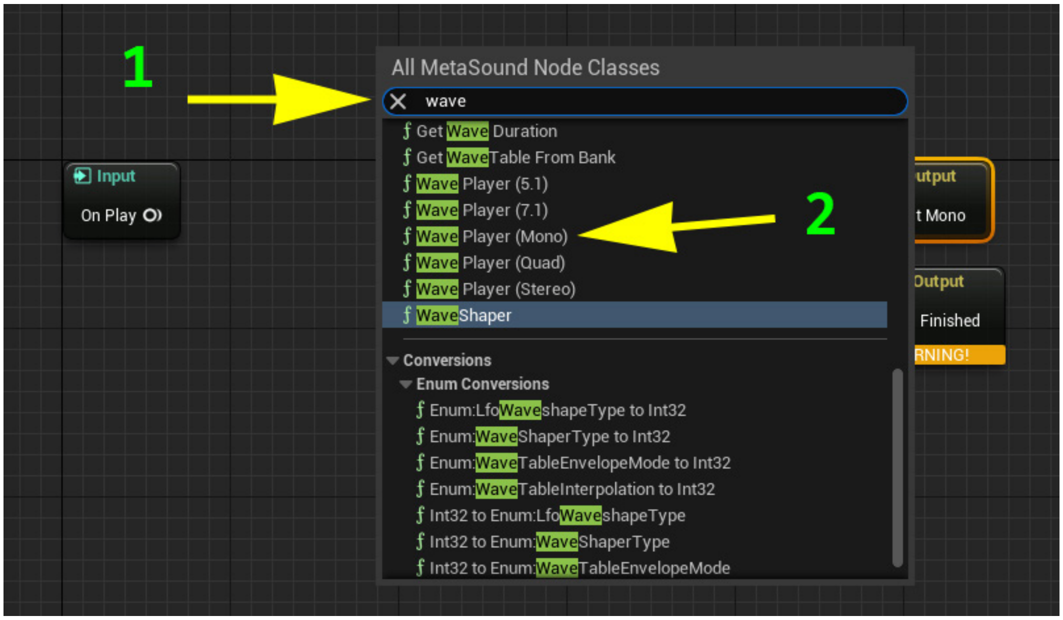

1
2
All MetaSound Node Classes
wave
Get Wave Duration
Get WaveTable From Bank
Wave Player (5.1)
Wave Player (7.1)
Wave Player (Mono)
Wave Player (Quad)
Wave Player (Stereo)
WaveShaper
Conversions
Enum Conversions
Enum:LfoWaveshapeType to Int32
Enum:WaveShaperType to Int32
Enum:WaveTableEnvelopeMode to Int32
Enum:WaveTableInterpolation to Int32
Int32 to Enum:LfoWaveshapeType
Int32 to Enum:WaveShaperType
Int32 to Enum:WaveTableEnvelopeMode
Input
On Play
utput
t Mono
Output
Finished
RNING!

Figure 11.31*: The "MS_01" MetaSound object blueprint editor. For playing a wave sound, right click and type "wave" (1) and then slelect "WavePlayer(Mono)" from list (2).*

5. The engine will add the **`Wave Player(Mono)`** blueprint block to the editor, which is responsible to play an audio file as mono. As you see there is a list of wave players, which provide different audio outputs, but for now we use mono output. Click on the **`Wave Asset`** option and choose an audio file (in this example, we use **`Click_On_Button`** sound wave asset, which is an engine default sound asset), as shown in *Figure 11.32*. Then, click on the **`OnPlay`** output pin in Input trigger and connect its execution line to **`Play`** pin in **`WavePlayer`**. Also, do the same for **`OutMono`** audio output in **`WavePlayer`** and connect it to the **`Out Mono`** pin in the **`Output`** trigger. The **`Output`** trigger

will send the audio to the scene. Finally, press **Play** on top and the sound will play:

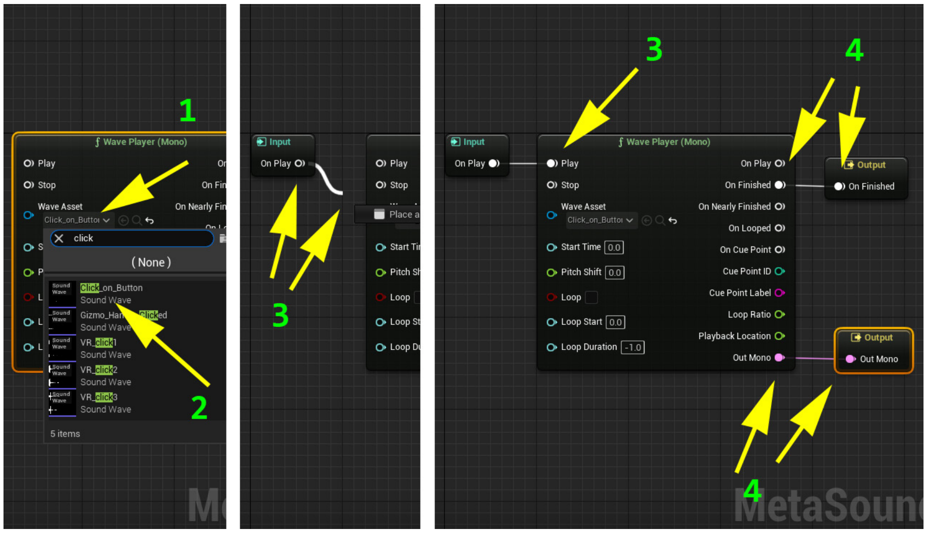


Figure 11.32: First select an audio asset to play in Wave Player by setting "Wave Asset" property (1) (2), then click and drag execution line from OnPlay pin in Input node and connect it to Play pin in WavePlayer node. (4) Do the same for OutMonot and OnFinished pins.

6. Now, let's attach this **MetaSound** object to a blueprint actor and play the sound on box collisions. Make a new actor blueprint with a collision box and call it **BP_item_01**. First bind events for box collision components in blueprint editor and then add an **Audio** blueprint component to this actor. Now select it, and locate **Sound** property in the **Details** panel and assign the **MS_01 MetaSound** object to it, as shown in *Figure 11.33*:

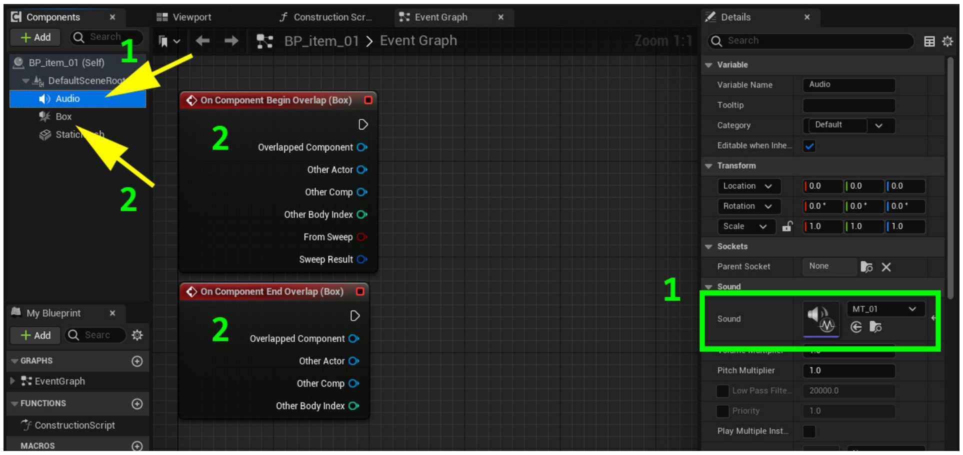
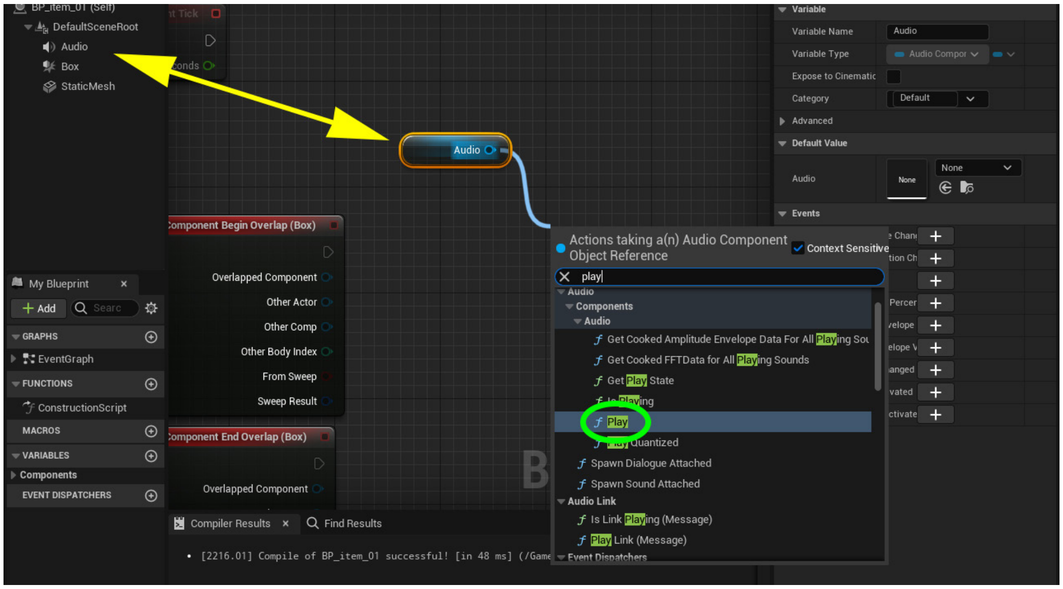


Figure 11.33: Select Audio component and assign "MS_01" MetaSound object for its Sound property (1), also bind collision events for box component(2).

7. Next, drag and drop the **`Audio`** component to the event graph and assign **`Play`** blueprint node to it, as shown in *Figure 11.34*:



```
• [2216.01] Compile of BP_item_01 successful! [in 48 ms] (/Game
```

Figure 11.34: *Drag and drop Audio component to the event graph and assign Play node to it from blueprint function list*

8. Assign **`Play`** from **`Audio`** to both collision event of collision box, as shown in *Figure 11.35*. Press **`Compile`** and test the game. You must hear the "**`Click_On_Button`**" sound on each collision with the box during runtime:

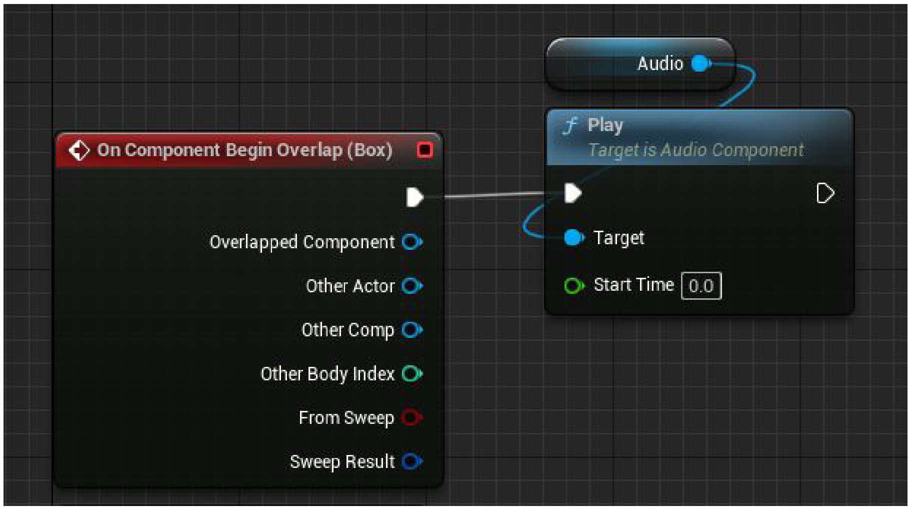


Figure 11.35: *Add play node to Audio component for box collision event.*

So far, we learn how to play a sound asset by using audio component and events in an actor. What if we need to loop the sound or in other word, "repeat" the sound?

For this functionality, open **`MS_01`** and check the **`Loop`** property in **`WavePlayer`**, as shown in *Figure 11.36*. When you run the game, the audio is repeated:

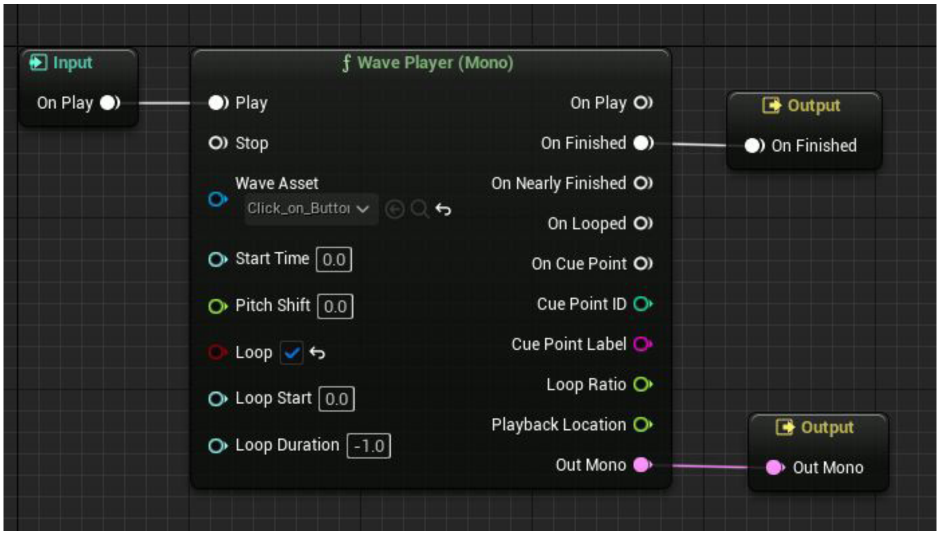
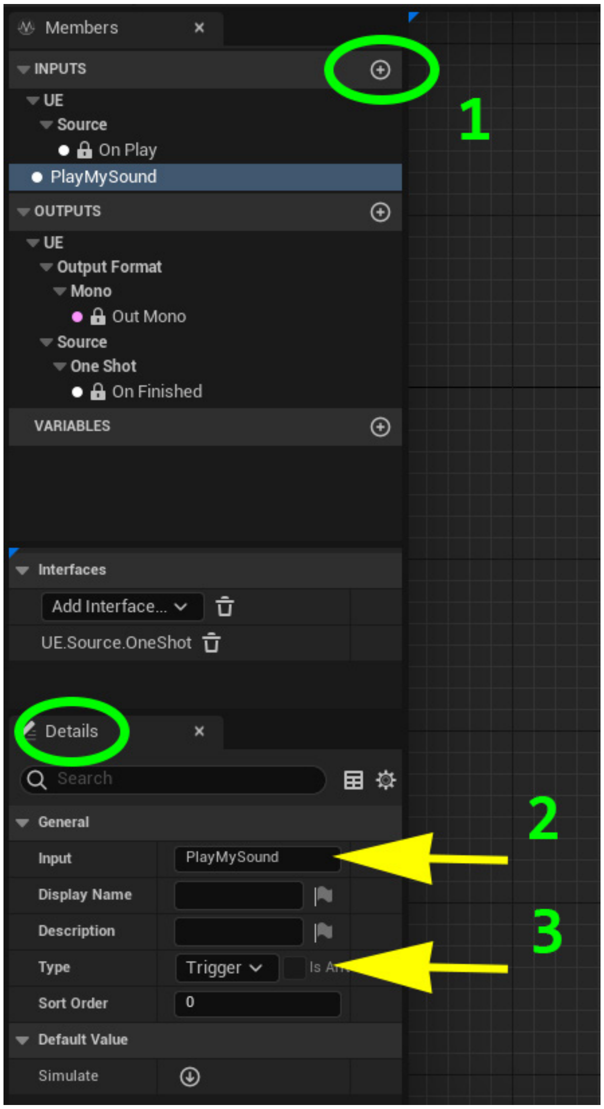


***Figure 11.36**: Check "Loop" check box in WavePlayer box to for looping the sound*

9. Now, imagine that we like to play the audio on loop when a player collides with box collision and stop the audio when a player leaves the box collision. There are a couple of ways to do this, and one of them is to assign *custom triggers* in the **`MetaSound`** object and switch the sound by using them from actor blueprint. "*custom triggers*" behave like events in actor blueprint but they are designed for MetaSound control. Open **`MS_01`**, click on the circular icon (with a plus sign inside) in the Inputs section located in the **`MetaSound`** panel. The engine will make a new input variable and show its properties in **`Details`** panel. Change its name to **`PlayMySound`** and its **`Type`** to **`Trigger`**, as shown in *Figure 11.37*. Make another input trigger and rename it to **`StopMySound`**. Now we made 2 input triggers to play and stop our sound.:



Input	PlayMySound
Display Name	
Description	
Type	Trigger
Sort Order	0

Figure 11.37: 1- Click on the circular icon (1) and add new input variable. Then change its name (2) and select "Trigger" for its "Type" property (3).

10. These input triggers will help us control the audio from blueprint. Drag and drop new triggers to the editor and connect them to **`Play`** and **`Stop`** pins of

`WavePlayer` as you can see in *Figure 11.38. Also disconnect* **`On Finished`** *pin from* **`WavePlayer`**:

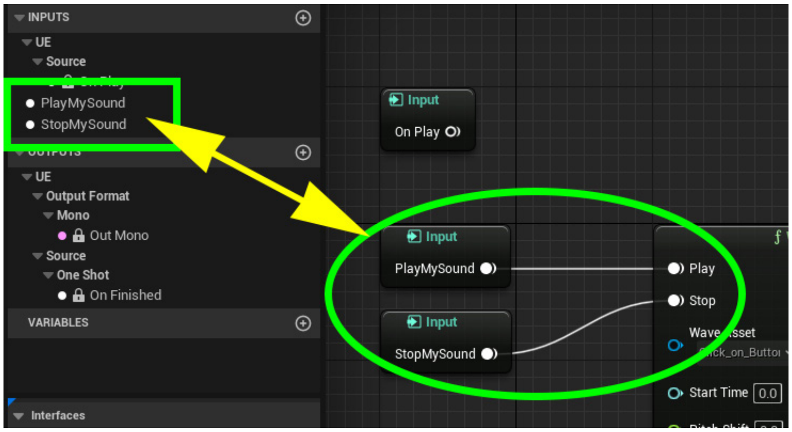


Figure 11.38*: Add Input triggers to the audio code*

11. Now, we need to *use* these input triggers in **`BP_item_01`** blueprint actor. Open it remove the **`play`** function for the **`Audio`** component. Now, assign the **`Execute Trigger Parameters`** blueprint node to each **`Audio`** component. This blueprint node works like as an interface between the **`MetaSound`** object and your actor objects. You can access to parameters inside the **`MetaSound`** object by using **`In Name`** property. Be very careful, the **`In Name`** property is case sensitive, so you must put *"exact"* parameter names there. In this example, we have **`PlayMySound`** and **`StopMySound`** as input names of our triggers. Put these names inside the **`In Name`** property, as shown in *Figure 11.39*. After you run game, the sound will be switched based on collision box events:

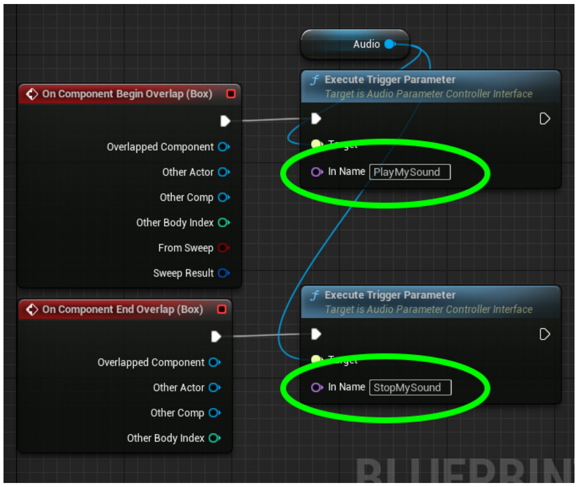

Audio
Execute Trigger Parameter
Target is Audio Parameter Controller Interface
On Component Begin Overlap (Box)
Overlapped Component
Other Actor
Other Comp
Other Body Index
From Sweep
Sweep Result
Target
In Name PlayMySound
On Component End Overlap (Box)
Execute Trigger Parameter
Target is Audio Parameter Controller Interface
Overlapped Component
Other Actor
Other Comp
Other Body Index
Target
In Name StopMySound
BLUEPRINT

Figure 11.39*: Sending parameter to MetaSound objects. Keep in mind the parameter name must be the same as MetaSound parameters*

12. Now, let's increase our knowledge by making this functionality with a different approach. Open **`MetaSound`**, make new trigger called **`SetMyLoop`** and set its **`Type`** as **`Bool`**. Then, add this trigger to the editor and connect it to the **`Loop`** pin in **`Wave Player`**, as shown in *Figure 11.40*. This way, we can directly change the loop functionality for wave player, so we don't need **`StopMySound`** anymore and can unpin it from **`Stop`** pin on **`WavePlayer`**:

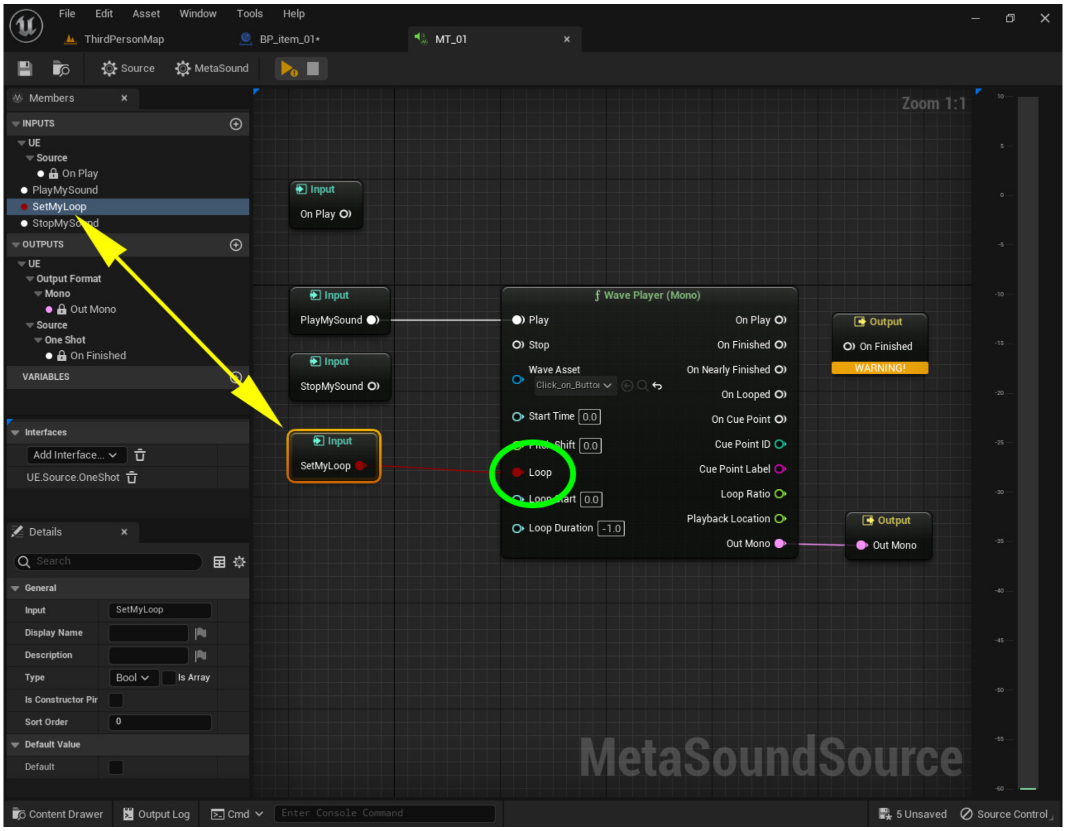


Figure 11.40*: Make new input trigger as "LoopCheck" and set its Type to bool. Drag and drop it to the editor and connect it to Loop pin on WavePlayer. Finaly, unpin the StopMySound trigger.*

13. Open **`BP_item_01`** again and assign **`Set Boolean Parameter`** blueprint node to the **`Audio`** component. This blueprint node will give you access to **`SetMyLoop`** bool parameter inside the MT_01 metasound object. Again, the **`In Name`** property is case sensitive, so you must put "*exact*" parameter names

there. Also, call **PlayMySound** trigger after switch on the loop as shown in *Figure 11.41*.

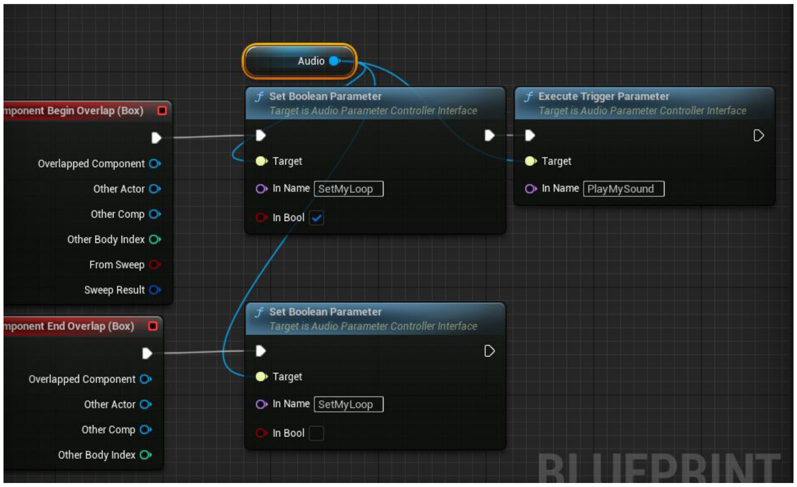


***Figure 11.41**: To access SetMyLoop bool parameter, click and add "Set Boolean Parameter" node and use exact name of bool parameter in the "InName" field.*

14. Now, let's add a delay to this looping sound. Open **MS_01** and unpin **Loop** and set it to false in **WavePlayer**, then and add the **Trigger Delay** node to the **On Play** pin in **WavePlayer**, as shown in *Figure 11.42*. This trigger gets activated (triggered) when the **WavePlaye**r playing sound and generate delay in second based on an integer value in **Delay Time** field. But there is one problem here; how play the sound *"again"* to make this delayed loop working?

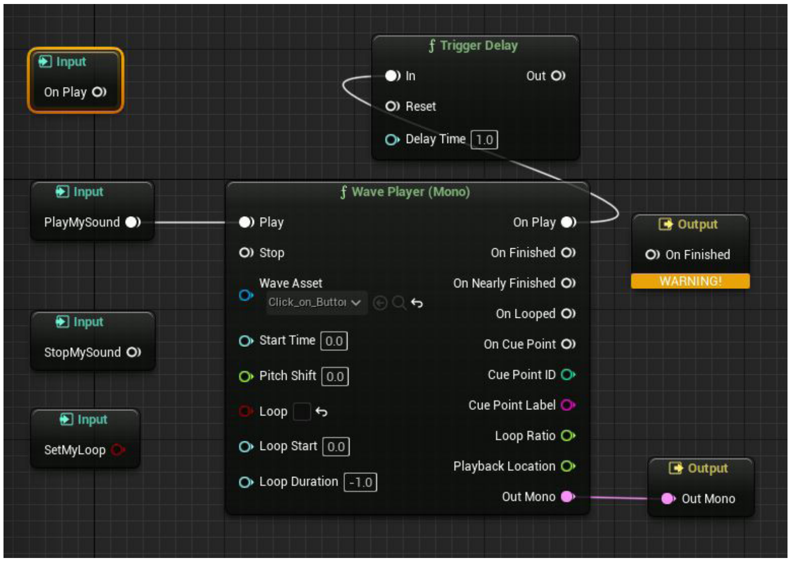


***Figure 11.42**: Add the Trigger Delay node to the On Play trigger pin and set Loop pin to false.*

15. MetaSound has patterns for coding audio loops. One of these patterns is implement "feedback" inside the audio code. You can create a variable and use it like a trigger anywhere in the code (similar to events in actor blueprint). For solving our problem here, we need to activate the **`Play`** pin in **`WavePlayer`** again. Click and drag a line from **`Out`** pin of **`Trigger Delay`** node and select **`Promote To Graph Variable`** option from the list, as shown in *Figure 11.43*:

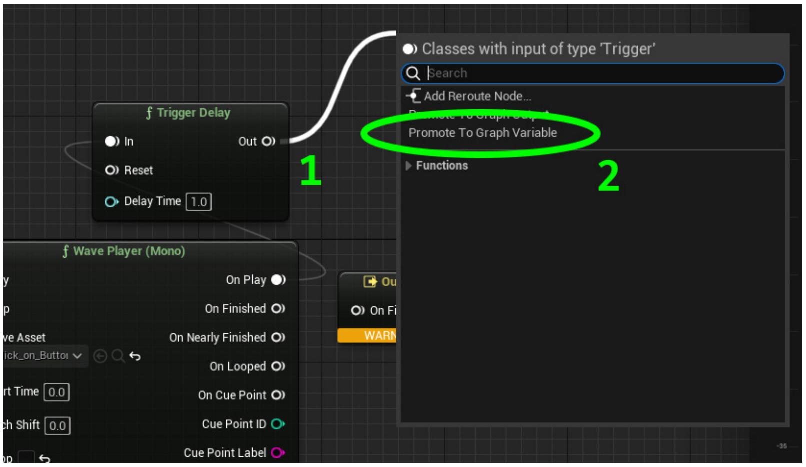


***Figure 11.43**: Click and drag from "Out" pin in "Trigger Delay" node (1) and select "Promote to Graph Variable" from the list (2).*

16. The metasound will automatically create a variable and add it to **`VARIABLES`** panel. Click on it and rename it to **`Out_01`** as shown in *Figure 11.44*:

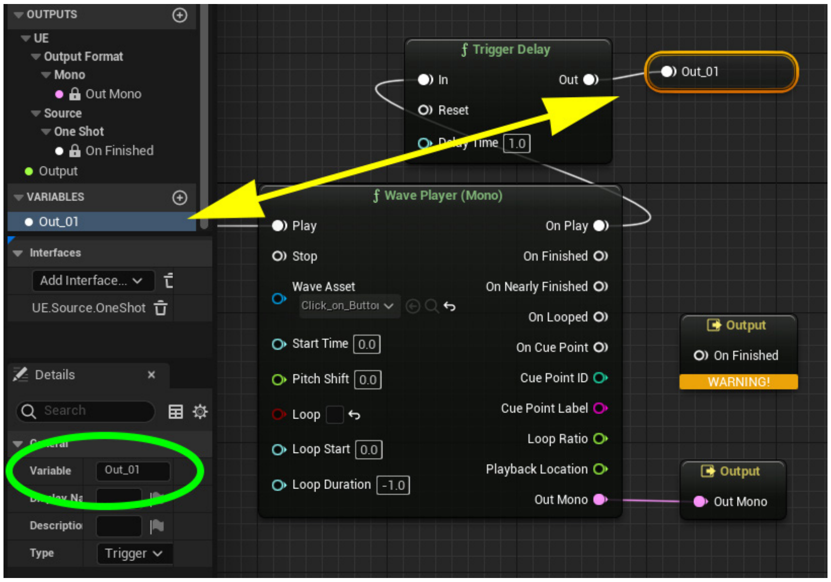


***Figure 11.44**: Click on new variable and rename it to "Out_01".*

17. Now, we need to use **`Out_01`** as a trigger on **`Play`** pin in **`WavePlayer`**, in order to repeat the sound. By using **`Trigger Any`** node, you can grab multiple triggers as input and activate an output trigger if any of them get triggered. As image Fig 11:45 shows, we add **`Trigger Any (2)`** node and connect our **`PlayMySound`** trigger to it as first input. Now anytime **`PlayMySound`** get triggered, the sound will be played.

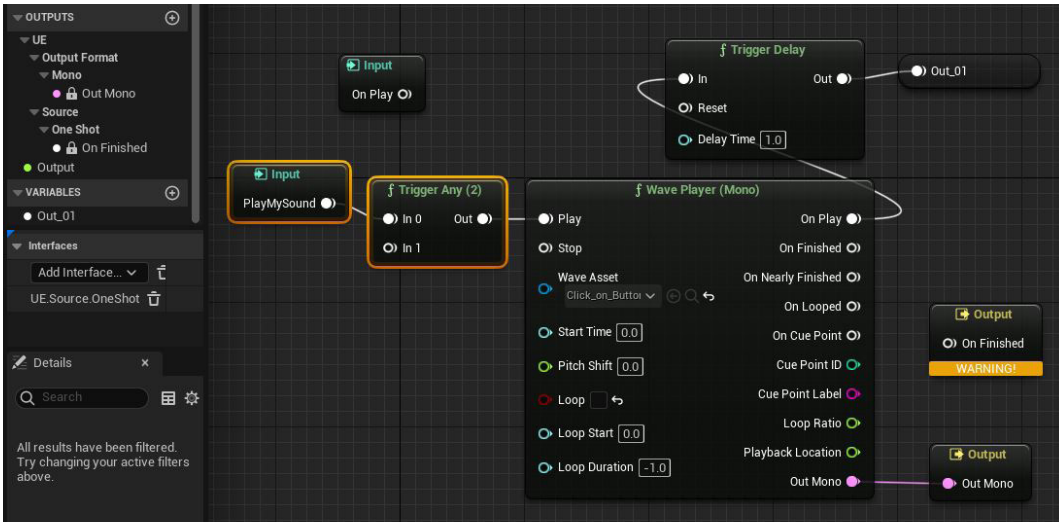


Figure 11.45: Click on new variable and rename it to "Out_01".

18. At this stage we already address **`PlayMySound`** trigger, and we need to add another trigger to **`Trigger Any (2)`** node in order to make the feedback working, and play the loop. We can use **`Out_01`** variable as a trigger. As you see in Fig 11:46, drag and drop **`Out_01`** variable in to the editor has numbers of extra options. Normal drag and drop makes a default reference to variable. Using Alt on keyboard will create a reference to delayed version of variable (which we will use). By holding Shift on keyboard and drag, we can "set" the trigger and is available only once per variable. By holding Ctrl on keyboard and drag, the editor will highlight all instances of that variable in the editor which is handy to find them in complex graph of code. We need to use delayed version of variable because this version will automatically be triggered by engine "after" very small delay. As you see, there is a clock like icon over delayed variable to make them separate from default ones.

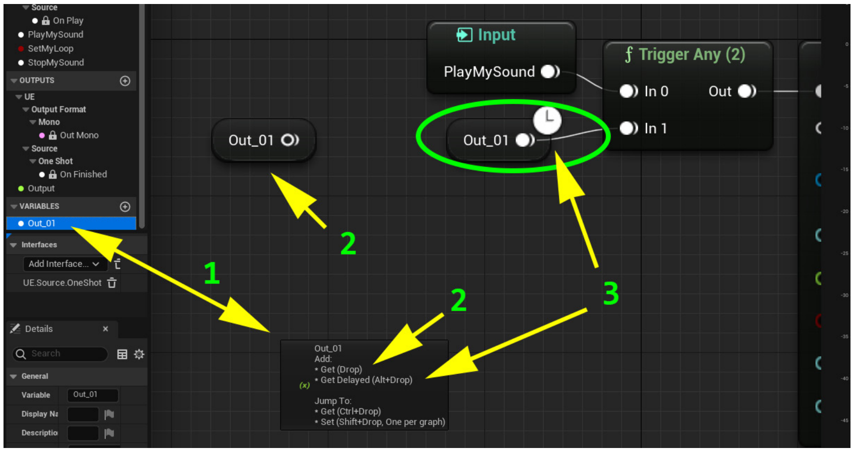


Figure 11.46: *Drag and drop the variable in the editor(1) by default creates a referenced to variable (2), but, if hold Alt key on keyboard and drag and drop to editor (3), then we have a reference to delayed variable.*

19. Now run the game and the sound should be heard as loop with 1 second delay. To stop the sound after player, leave collision box, we can use **`SetMyLoop`** bool value with a compare node.

    Right click in the editor and type “trigger compare”. The editor will give you 3 types of compare node for bool, float and int32 variables. We can use bool node for checking **`SetMyLoop`** trigger, so select **`Trigger Compare (Bool)`** node. This node has two input pin as **`A`** and **`B`**, and this node compare these two and trigger **`True`** and **`False`** output triggers as result. So as Fig 11.47 shows, we set pin **`A`** to true, and compare it with trigger **`SetMyLoop`** which is connected to pin **`B.`**

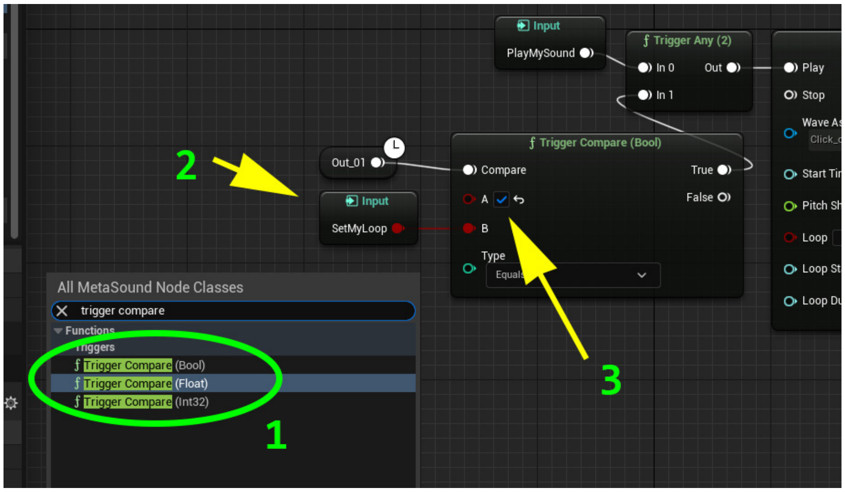


Figure 11.47: *Right click in the editor and type “trigger compare ”(1). Then add “Trigger Compare(Bool)” and connect “SetMyLoop” bool trigger to pin “B” on it (2) and set “A” pin to true (3).*

20. After running the game, when **`SetMyLoop`** is equal to true, the **`True`** output pin will be triggered and we have a looped sound. But in case if it is false, the compare node will activate the **`False`** trigger output and as previous image shows, it is not connected to any other node, so in result the looping sound will stop playing.

    Like actor blueprint, we have a node in metasound editor to print debug message in output log panel in main editor. Actor blueprints use **`Print String`** node, and metasound use **`Print Log`** node to print debug messages. This node has an input trigger to activate it, an input string text field known as **`Label`** to add extra note, and another input pin known as **`Value To Log`** which depends on type can be bool, float, int32 or string. Fig 11.48 demonstrates using **`Print Log(Bool)`** to print value of **`SetMyLoop`** on each loop.

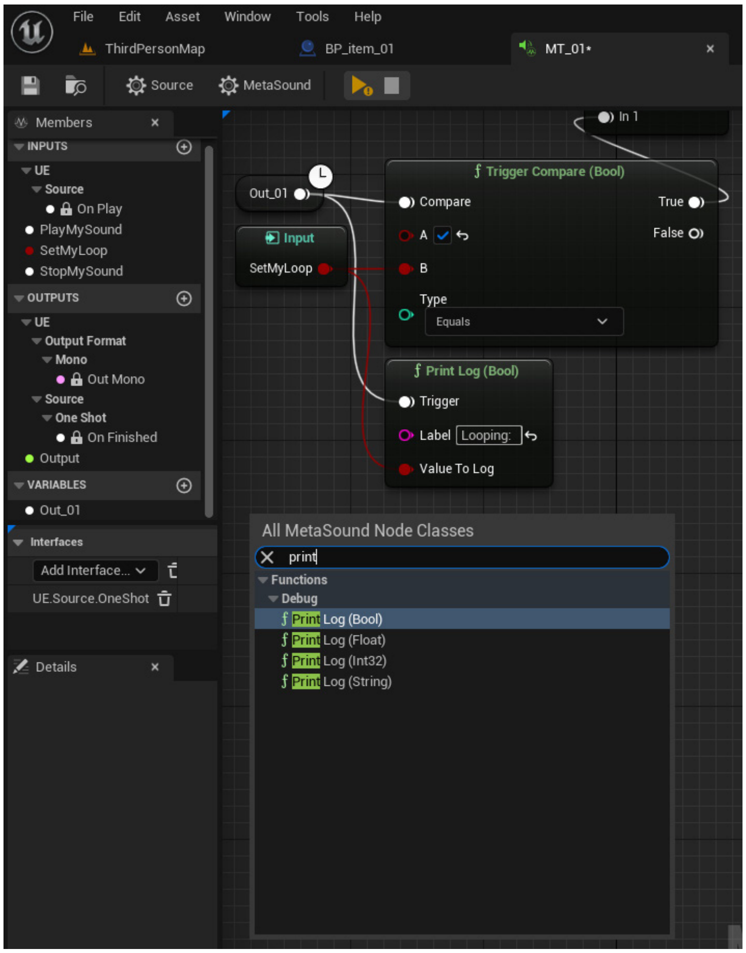
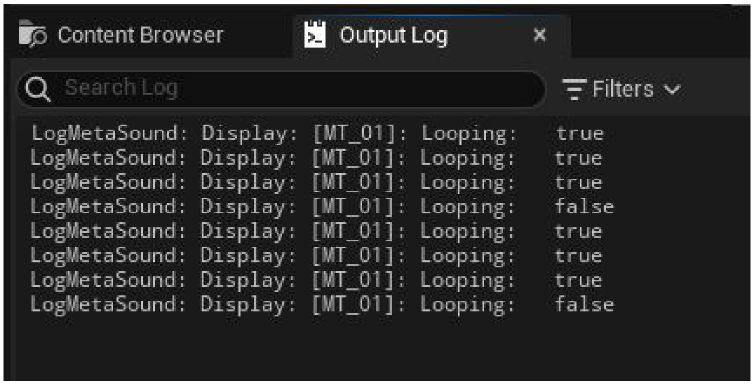


Figure 11.48: *Right click in editor and type print and then choose "PringLog(Bool)"and connect it to "Out_01" trigger and "SetMyLoop" bool trigger.*

21. You can check the output of **`Print Log(Bool)`** in the editor in output log panel as shown in Fig 11.49 by running the game and interact with **`BP_item_01`** collision box.



```
LogMetaSound: Display: [MT_01]: Looping:   true
LogMetaSound: Display: [MT_01]: Looping:   true
LogMetaSound: Display: [MT_01]: Looping:   true
LogMetaSound: Display: [MT_01]: Looping:   false
LogMetaSound: Display: [MT_01]: Looping:   true
LogMetaSound: Display: [MT_01]: Looping:   true
LogMetaSound: Display: [MT_01]: Looping:   true
LogMetaSound: Display: [MT_01]: Looping:   false
```

***Figure 11.49**: Printing log notes in "Output Log" panel in main editor with values from metasound object at runtime.*

There are more details about setting and coding patterns in MetaSound, which meet much dedicated book to this topic. At this stage, we have learned how to create MetaSound object, play an audio asset by using it, make triggers and parameters inside it, and use them to communicate with blueprint actors. Also, we have learned how to invoke triggers inside the MetaSound object by using delayed variable and derive logical procedures based on bool triggers. These are basic skills that will help you to research and practice more about MetaSound coding patterns in Unreal Engine 5.

Conclusion

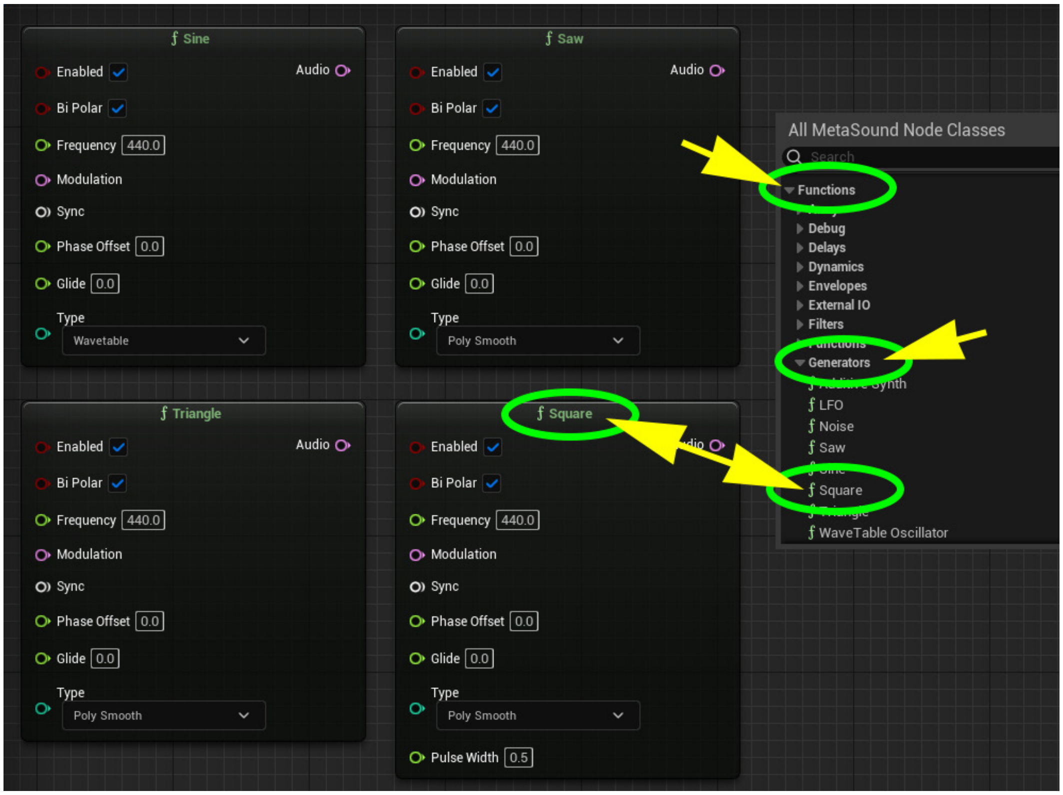
The harmony between visual effects and sound is always a memorable feature of any game. It doesn't matter if the game is simple or belongs to a franchise. We went through the basics of Niagara, which provides powerful tools to generate and address a complex particle system inside game scene. We learned how to make parameters to control Niagara System and mix emitters to make complex visual effects. At the end, we went through MetaSound, which a unique feature of Unreal Engine 5. We learned how to generate and control sound effects inside the game scene at runtime by learning the basics of audio programming and practice different approaches.

Both Niagara and MetaSounds looks complex environment, but since we know the basics, the rest is research and development by making sample projects.

In the next chapter, we will learn about *packaging* and learn how to make the final product for target platforms like PC and Android. There, you can test and compare the quality of your sample projects on PC and phone device and make it more professional.

Points to remember

- Use the **`.wav`** format for audio files before import to engine as audio asset.
- Having complex static mesh or expensive material for static mesh when using Niagara *Render* is risky. You may need to research on how to optimize your mesh *render* cost or in case of material, simplify your shader code as much as you can.
- MetaSound sometimes works unusually. The best way to fix this issue is by closing its editor, pressing **`Save All`** in content browser and then open it again.
- MetaSound comes with numbers of audio-rate generators: Saw, Sine, Square, and Triangle, as shown in *Figure 11.50* are classic examples. Each of these nodes supports controlling their frequency and modulating that frequency at audio rate (for FM synthesis). You can check other generators in "Function>Generators" from drop down blueprint node list in editor.



ƒ Sine: Enabled ✓, Bi Polar ✓, Frequency 440.0, Modulation, Sync, Phase Offset 0.0, Glide 0.0, Type Wavetable; output Audio

ƒ Saw: Enabled ✓, Bi Polar ✓, Frequency 440.0, Modulation, Sync, Phase Offset 0.0, Glide 0.0, Type Poly Smooth; output Audio

ƒ Triangle: Enabled ✓, Bi Polar ✓, Frequency 440.0, Modulation, Sync, Phase Offset 0.0, Glide 0.0, Type Poly Smooth; output Audio

ƒ Square: Enabled ✓, Bi Polar ✓, Frequency 440.0, Modulation, Sync, Phase Offset 0.0, Glide 0.0, Type Poly Smooth, Pulse Width 0.5; output Audio

***Figure 11.50**: Classic audio-rate generators in MetaSound: Saw, Sine, Square, and Triangle. You can find more generators in "Function>Generators" after right click in editor.*

Multiple choice questions

1. **Why do we use Niagara System in Unreal Engine?**
    a. To add Niagara Emitters and control them by parameters
    b. To make a particle system
    c. Implement real-time visual effects in game scene
    d. Options a, b and c are correct

2. **Which of the following sentences about `Trigger` in MetaSound is correct?**
    a. It works like event in actor blueprints.
    b. It handles receiving parameters of various data type from other objects.
    c. It has series of special nodes to select, combine, compare, etc. in case of using numbers of triggers together.
    d. Options a, b, c are correct.

Answers

1. **d**
2. **d**

Questions

1. What are your own system limitations (home or office) when you work with Niagara?
2. Why does Niagara Emitter's *Render* sometime not show any visual output in the **`Preview`** panel?
3. Some external audio cards provide a wider range of audio tools for creating and processing real-time sound. How can this help MetaSound to provide complex real-time audio experience with Unreal Engine 5 when you make sound by using *pure* audio-rate generators?
4. What is the difference between *Send* and *Receiver* when we use Triggers in MetaSound object?

Key terms

- **Niagara -** Niagara is Unreal Engine's VFX system to design and develop particle systems in application.

- **Renderer** – Rendering any visual effect from Niagara is the function of Renderer.
- **Emitter -** Niagara emitters are basically like a containers for other Niagara modules.
- **Modules -** Niagara modules are the base level of Niagara VFX which present unique behavior and visualizations.
- **User parameter -** An abstraction of data in a Niagara simulation which is defined by user.
- **MetaSound –** Unique feature in Unreal Engine 5 to design and develop high-performance audio system that provides audio designers with complete control over a **Digital Signal Processing** (**DSP**) graph for the generation of sound sources.
- **Trigger –** Trigger is a pin or series of pins on MetaSound`s blueprint blocks to receive and send data or execution lines.
- **Set Boolean Parameter:** Set a Boolean value in MetaSound by using its name.
- **Execute Trigger Parameter:** Activate a trigger in MetaSound by using its name.
- **Delayed Variable:** In MetaSound, developers can use a variable as a trigger. When they get triggered, the engine automatically trigger them as default and as a delayed version. The delayed version will triggered after a small value of time.

Chapter 12
Packaging

Each game engine has its own procedures and options to release a *Built* version of the project. The *Built* version of a project is an executable standalone application that users can run on their platform and observe its functionalities and interact with it during runtime. In Unreal Engine, the process of making and releasing the built version is known as *Packaging*. The engine will go through several processes that are customizable by the user to compile the source code, involve game assets, and make the executable version of the project at the end.

Structure

In this chapter, we will discuss the following topics:

- Platforms and licensing
- Plugins (Engine and Project Level)

Objectives

In this chapter, we will learn how to build and release a playable standalone game. We will learn the packaging method, review options and windows related to this process, and point out Android built and plugins options in project setting related to this topic.

Platforms and licensing

Platform in computer science is a digital environment that provides resources to run a software/app compatible with that platform. Each platform has its own features: some can provide cloud processing tools for the applications, some use virtual machine technologies to execute applications independently from local machines and some totally depend on it. For example, Microsoft Windows operating system family provide a platform for PC computers in which developers can create application and software and *execute* them by using Windows tools and resources on PC machines. Example of these applications are games, word processing, network management applications, graphic software, and so on.

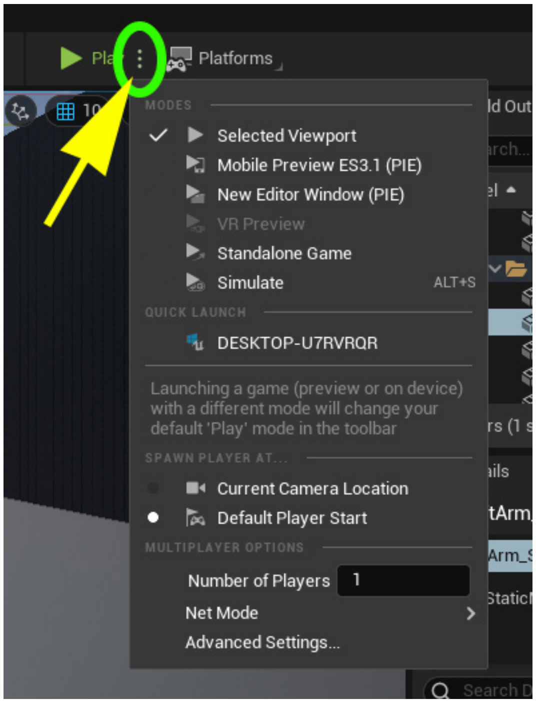
Unreal Engine 5 supports a number of platforms to develop and run applications. As *Figure 12.1* shows, by clicking on the vertical three-dots icon to the right of the **`Play`** button in the main editor, you will have access to a drop-down list of available platforms that you can run a preview of your game on.

By default, the engine has these options to preview game, which is listed in **`MODES`** of drop-down list:

- **Selected Viewport**: The engine runs game inside the editor. By default, the mouse cursor will disappear, and the keyboard/controller is active in the game. You can hold shift and press *F1* to release control of mouse and keyboard while the game is running.
- **Mobile preview (version & type)**: Mobile devices have less processing power and graphic qualities than a normal PC. When you have heavy shaders or complex assets and want to check the quality for example in a mobile phone, the engine allows to run a mobile rendered playable preview of game in separate windows when you click on this option.
- **New Editor Windows**: Instead of running the game inside the main editor, you can run it in a separate window by choosing this option.
- **VR Preview**: In case of having a VR headset connected to your machine, the engine will preview the game inside the VR headset. Keep in mind this is just a *preview*, and in order to have a real VR game, you have to make your project from a VR template (refer to chapter # engines template) or develop a new one from scratch.
- **Standalone Game**: The engine run *one* standard preview of game, based on your local machine features in separate window.
- **Simulate**: When the engine previews the game in Simulate mode, the game is running and all game objects receive *tick* event at runtime, but the user interaction with game mouse and keyboard will *not* be taken into account.

So, the user can click on each object and check its blueprint and monitor the execution line, variables, events, and so on inside the selected blueprint. At the same time, user is *not* allowed to edit any blueprint code because all event graphs are disabled. This is extremely important for debugging projects at runtime, and it's highly recommended to be skillful on using Simulate mode when you develop a serious project with Unreal Engine.

Now, let's see these **`Play`** options in the editor, as shown in *Figure 12.1*:



Launching a game (preview or on device) with a different mode will change your default 'Play' mode in the toolbar

Number of Players 1

***Figure 12.1**: Click on the vertical three-dots icon to the right of the Play button to expand play options*

Let's get back to the Platform topic at the beginning. As you see, by using different Play options, the engine can preview and run the game/application in different platforms, but how can a PC machine run other platform previews? The answer is hidden in a toolkit known as **Software Development Kit** (**SDK**). The SDK is a set of digital tools mostly designed and published by platform manufacturers. For example, this SDK can provide services and support for making applications with Android, which is a public platform, for a number of mobile phones, VR headsets, and tablets. In order to develop/create a new application that can run with an Android platform, developers need to use Android SDK on their machine. The Android SDK adds a series of codes and libraries to support development and debug on local machine. This sounds pretty complex, but we don't need to be worried, since the engine has methods to grab SDKs and use them for target platform. Follow these steps to do that:

1. Click on **`Platform Icon`** in main editor, as shown in *Figure 12.2*. The drop-down list shows a number of platforms and by default, your machine is listed and is available to launch and run the game:

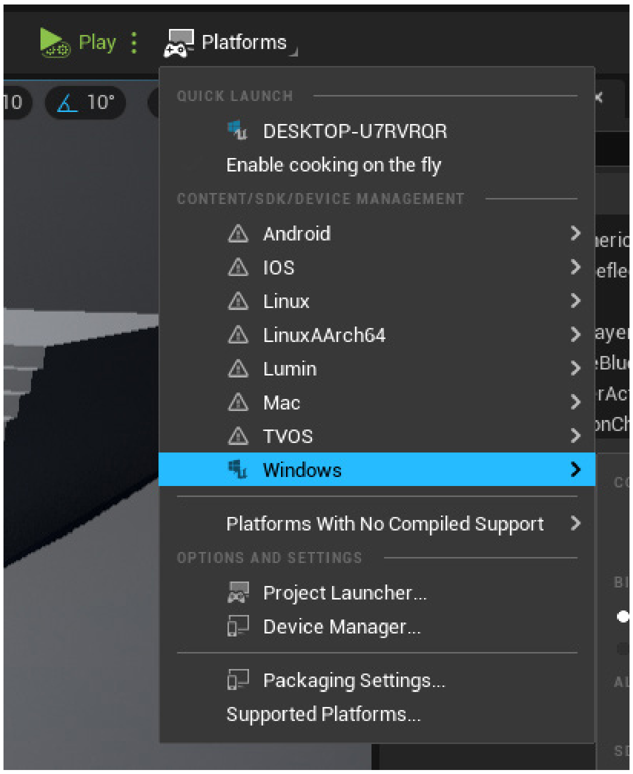


***Figure 12.2**: Platforms drop-down menu*

2. By clicking on the **`Project Launcher`** in the Platforms drop-down menu, the engine will automatically open project launcher windows. Here, users can assign projects to different platforms for testing, cook, debugging, and releasing shipping version.

3. As *Figure 12.3* shows, the project launcher has two main panels. The top one shows the available platforms, and by pressing **`Advanced`**, you can see the details of the device where you want to deploy project builds. The bottom panel with title **`Custom Launch Profiles`** is designed to customize project built. Users can select default engine profiles by clicking on the small arrow or add new profile and customize it by clicking on the **+** icon:

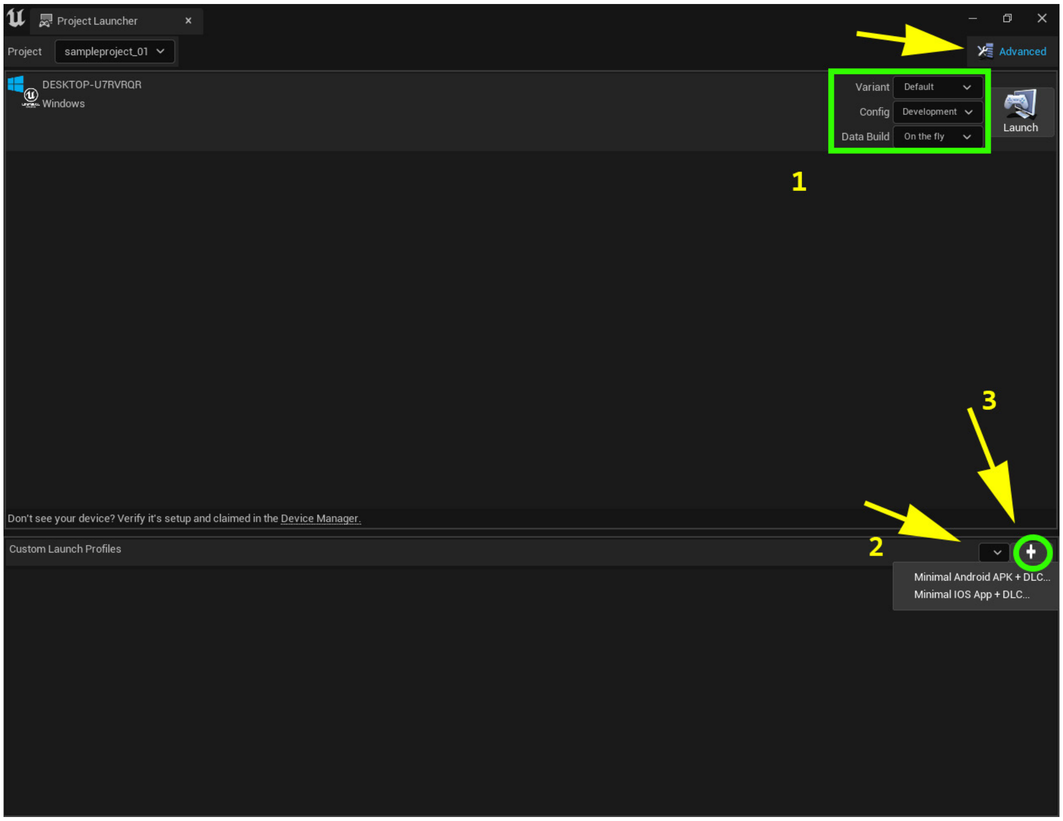
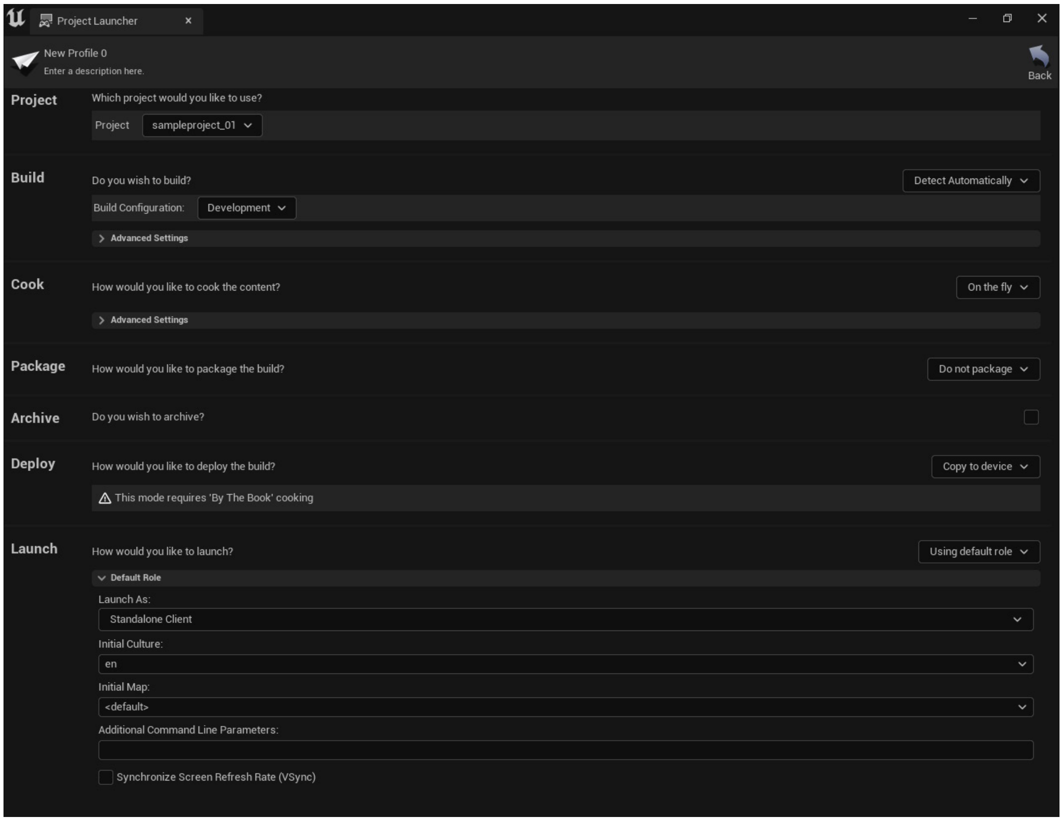


***Figure 12.3**: Click on Advanced button (1), then to assign engine's default launch profile click on the little arrow (2). Also, by pressing "+" icon (3), you can make a custom launch profile.*

4. When the user clicks on the **+** icon, the engine opens a new profile page, and we can customize the **`Build`**, **`Package`**, **`Archive`**, **`Deploy`** and **`Launch`** properties of the project, as shown in *Figure 12.4*:


Project Launcher

New Profile 0
Enter a description here.

Back

Project — Which project would you like to use?
Project: sampleproject_01

Build — Do you wish to build? Detect Automatically
Build Configuration: Development
Advanced Settings

Cook — How would you like to cook the content? On the fly
Advanced Settings

Package — How would you like to package the build? Do not package

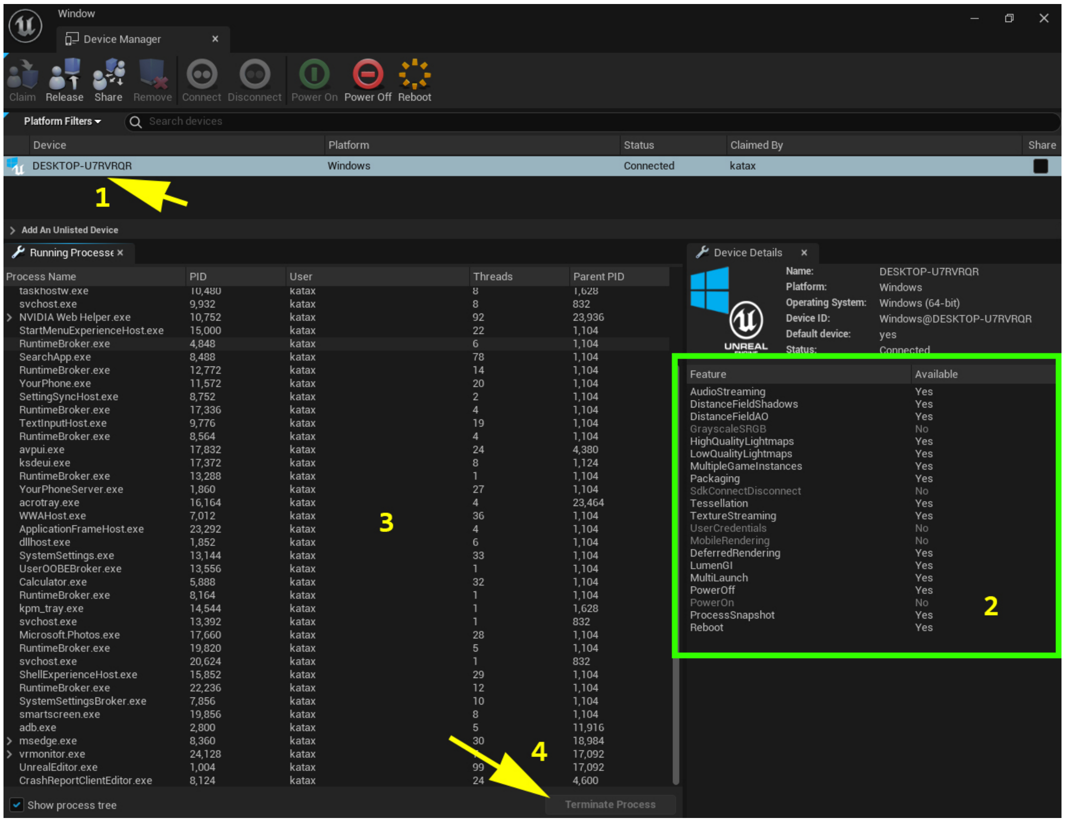
Archive — Do you wish to archive?

Deploy — How would you like to deploy the build? Copy to device
This mode requires 'By The Book' cooking

Launch — How would you like to launch? Using default role
Default Role
Launch As: Standalone Client
Initial Culture: en
Initial Map: <default>
Additional Command Line Parameters:
Synchronize Screen Refresh Rate (VSync)

***Figure 12.4**: Custom launch profile screen*

5. Click on **`Device Manager`** in the **`Platform`** drop-down menu, and the engine will open device manager window. By default, users have at least one device listed there (the machine who runs the engine). Click on the default device in **`Device Manager`**, as shown in *Figure 12.5*, and the engine automatically show details of that device plus a list of running processes that can be terminated by the user:



Device	Platform	Status	Claimed By	Share
DESKTOP-U7RVRQR	Windows	Connected	katax	

Process Name	PID	User	Threads	Parent PID
taskhostw.exe	10,480	katax	8	1,628
svchost.exe	9,932	katax	8	832
NVIDIA Web Helper.exe	10,752	katax	92	23,936
StartMenuExperienceHost.exe	15,000	katax	22	1,104
RuntimeBroker.exe	4,848	katax	6	1,104
SearchApp.exe	8,488	katax	78	1,104
RuntimeBroker.exe	12,772	katax	14	1,104
YourPhone.exe	11,572	katax	20	1,104
SettingSyncHost.exe	8,752	katax	2	1,104
RuntimeBroker.exe	17,336	katax	4	1,104
TextInputHost.exe	9,776	katax	19	1,104
RuntimeBroker.exe	8,564	katax	4	1,104
avpui.exe	17,832	katax	24	4,380
ksdeui.exe	17,372	katax	8	1,124
RuntimeBroker.exe	13,288	katax	1	1,104
YourPhoneServer.exe	1,860	katax	27	1,104
acrotray.exe	16,164	katax	4	23,464
WWAHost.exe	7,012	katax	36	1,104
ApplicationFrameHost.exe	23,292	katax	4	1,104
dllhost.exe	1,852	katax	6	1,104
SystemSettings.exe	13,144	katax	33	1,104
UserOOBEBroker.exe	13,556	katax	1	1,104
Calculator.exe	5,888	katax	32	1,104
RuntimeBroker.exe	8,164	katax	1	1,104
kpm_tray.exe	14,544	katax	1	1,628
svchost.exe	13,392	katax	1	832
Microsoft.Photos.exe	17,660	katax	28	1,104
RuntimeBroker.exe	19,820	katax	5	1,104
svchost.exe	20,624	katax	1	832
ShellExperienceHost.exe	15,852	katax	29	1,104
RuntimeBroker.exe	22,236	katax	12	1,104
SystemSettingsBroker.exe	7,856	katax	10	1,104
smartscreen.exe	19,856	katax	8	1,104
adb.exe	2,800	katax	5	11,916
msedge.exe	8,360	katax	30	18,984
vrmonitor.exe	24,128	katax	[illegible]	17,092
UnrealEditor.exe	1,004	katax	99	17,092
CrashReportClientEditor.exe	8,124	katax	24	4,600

Name:	DESKTOP-U7RVRQR
Platform:	Windows
Operating System:	Windows (64-bit)
Device ID:	Windows@DESKTOP-U7RVRQR
Default device:	yes
Status:	Connected

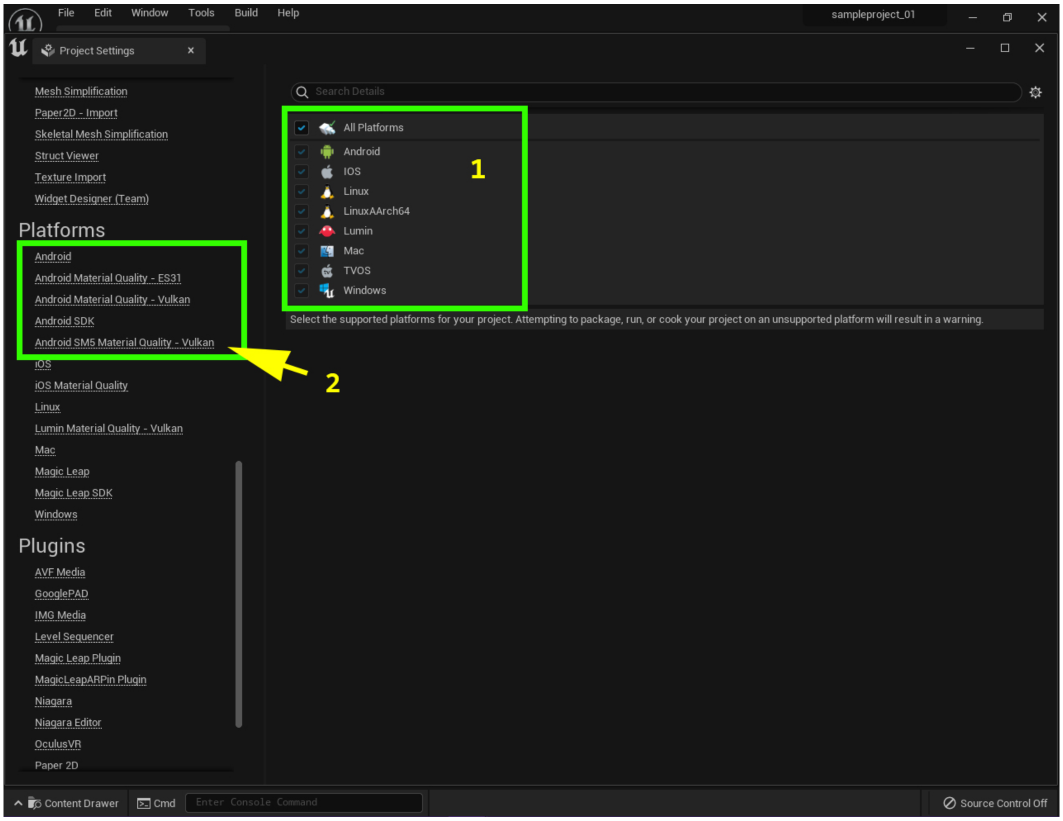
Feature	Available
AudioStreaming	Yes
DistanceFieldShadows	Yes
DistanceFieldAO	Yes
GrayscaleSRGB	No
HighQualityLightmaps	Yes
LowQualityLightmaps	Yes
MultipleGameInstances	Yes
Packaging	Yes
SdkConnectDisconnect	No
Tessellation	Yes
TextureStreaming	Yes
UserCredentials	No
MobileRendering	No
DeferredRendering	Yes
LumenGI	Yes
MultiLaunch	Yes
PowerOff	Yes
PowerOn	No
ProcessSnapshot	Yes
Reboot	Yes

***Figure 12.5**: Device Manager windows with list of devices (1), properties related selected platform (2), running processes related to it (3), and "Terminate Process" button (4) for terminating a process*

6. Click on **`Supported Platform`**, and the engine will open **`Project Setting`** (which is also available from **`Edit`** menu in main editor) and shows a list of available platforms for the project, as shown in *Figure 12.6*.

   For example, if your game/application is made for *only* PC users, it is recommended to untick other platforms here and leave **`Windows`** ticked to simplify the packaging process. Now, refer to the topic *Platform* at the beginning of this section; you can see that there are a number of platforms that Unreal Engine 5 is able to support to make an interactive application. Android is one of most important platforms in this list, which, as we mentioned before, is used in a number of digital smart devices. Locate the

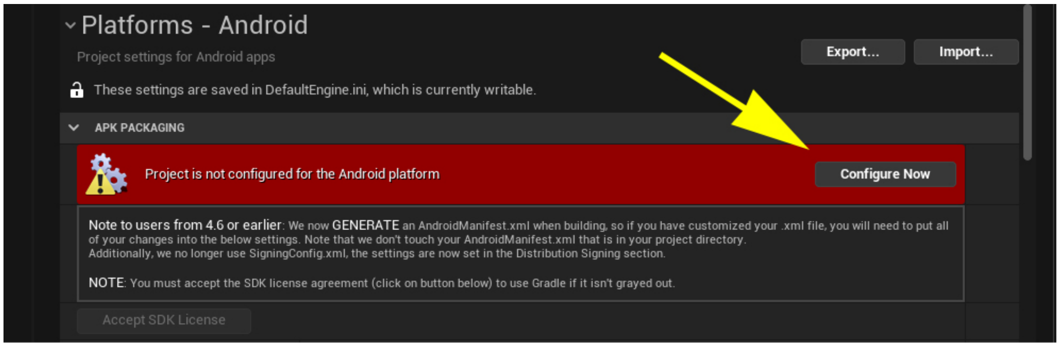
`Platforms` title on the left side of the **`Project Setting`** window, as shown in *Figure 12.6*:



Select the supported platforms for your project. Attempting to package, run, or cook your project on an unsupported platform will result in a warning.

Figure 12.6: *1- Supported Platform list in Project Settings windows*
2- Android SDK properties in Platforms

7. Now, click on Android and, as shown in *Figure 12.7*, there is a list of properties to customize the Android release of application. The first buttons you must click on are the **`Configure Now`** buttons, after which the engine will set the project for releasing in Android at background. The next one is **`Accept SDK License`**, which is grayed out by default:



Platforms - Android

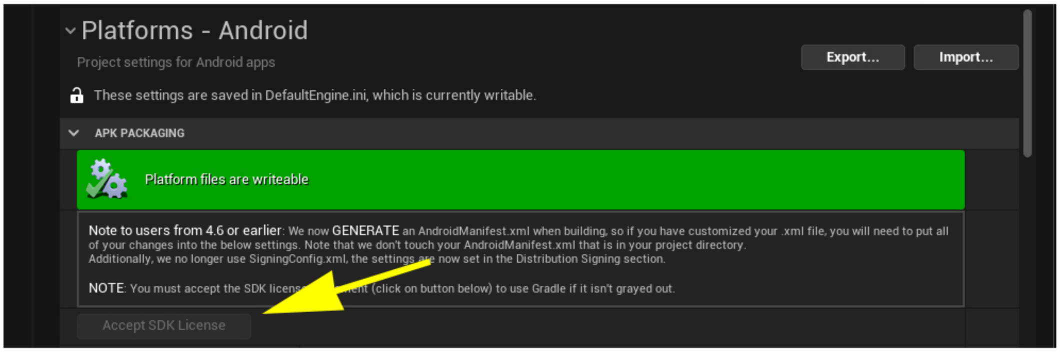
Project settings for Android apps

These settings are saved in DefaultEngine.ini, which is currently writable.

Project is not configured for the Android platform

Note to users from 4.6 or earlier: We now GENERATE an AndroidManifest.xml when building, so if you have customized your .xml file, you will need to put all of your changes into the below settings. Note that we don't touch your AndroidManifest.xml that is in your project directory.
Additionally, we no longer use SigningConfig.xml, the settings are now set in the Distribution Signing section.

NOTE: You must accept the SDK license agreement (click on button below) to use Gradle if it isn't grayed out.



Platforms - Android

Project settings for Android apps

These settings are saved in DefaultEngine.ini, which is currently writable.

Platform files are writeable

Note to users from 4.6 or earlier: We now GENERATE an AndroidManifest.xml when building, so if you have customized your .xml file, you will need to put all of your changes into the below settings. Note that we don't touch your AndroidManifest.xml that is in your project directory.
Additionally, we no longer use SigningConfig.xml, the settings are now set in the Distribution Signing section.

NOTE: You must accept the SDK license agreement (click on button below) to use Gradle if it isn't grayed out.

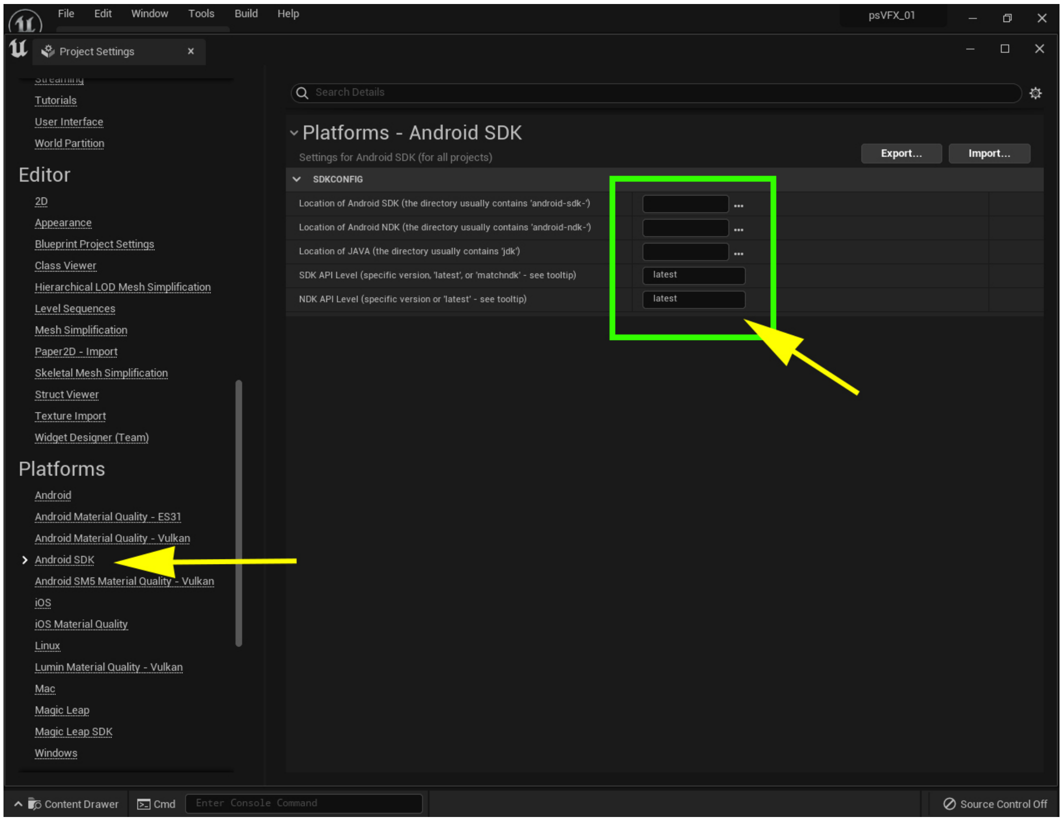
Figure 12.7*: Click on the "Configure Now" button to apply Android configuration to your project*

8. To activate this button, we need to do licensing first. Licensing is an agreement between SDK provider and users who like to use the SDK and develop applications with it. or like to download and use the application developed with that SDK.

   For example, when an application checks the licensing status with Google, their servers use a key that is uniquely associated with the application to support user and developers.

9. To use Android SDK, you must click on **`Android SDK`**, as shown in *Figure 12.8*, and then address the necessary resources from local drive. Then, switch

back to the **Android** section, as shown in Figure 12.8, and click on **Accept SDK License**:



Platforms - Android SDK
Settings for Android SDK (for all projects)

SDKCONFIG

Location of Android SDK (the directory usually contains 'android-sdk-')	
Location of Android NDK (the directory usually contains 'android-ndk-')	
Location of JAVA (the directory usually contains 'jdk')	
SDK API Level (specific version, 'latest', or 'matchndk' - see tooltip)	latest
NDK API Level (specific version or 'latest' - see tooltip)	latest

***Figure 12.8**: Android SDK configuration*

Plugins

Plugins are collections of data and code that users can easily enable or disable within the Editor on a per-project basis. They can modify built-in engine features (like MetaSound explained in *Chapter 11, Audio and Particles*), simplify and address complex instructions, create new file types, and add new menus or tool bar to editor to serve special order and more. You can get plugins from Marketplace or develop one by yourself. As *Figure 12.9* shows, clicking on **Settings** and then on **Plugins** will make the engine open the **Plugins** window. Indeed, you can open this window from the **Edit** menu in editor:

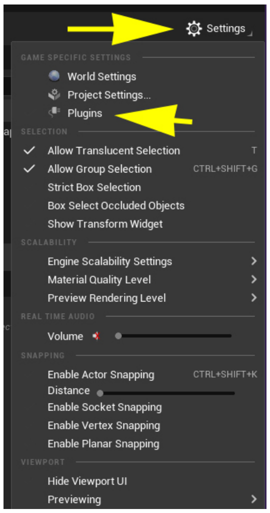
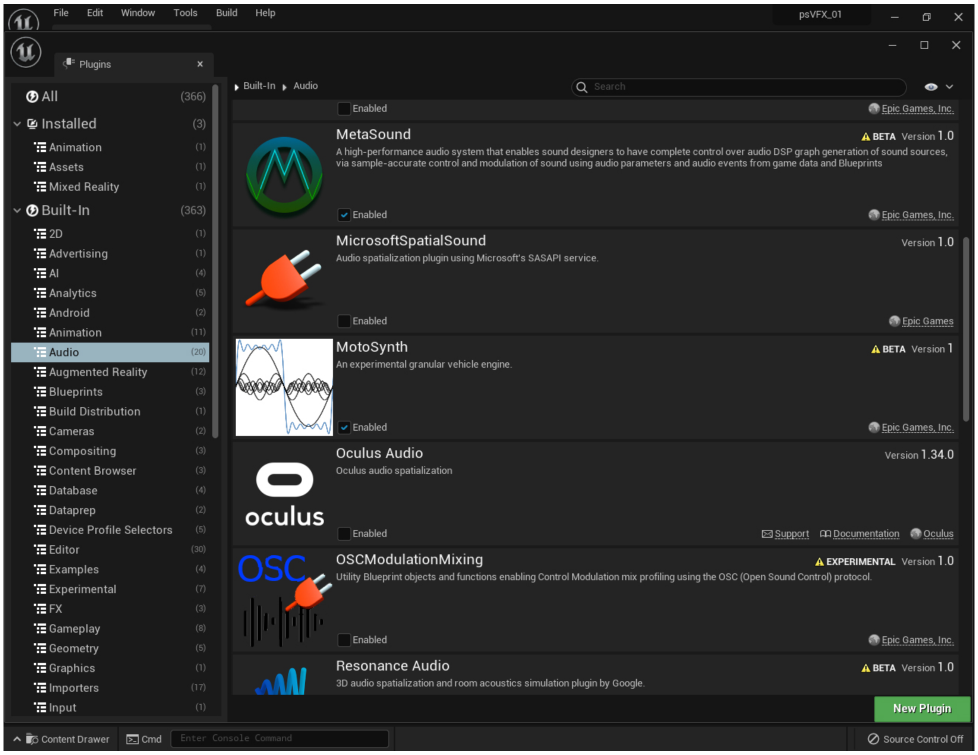


***Figure 12.9**: Open Plugins from Setting icon*

The **`Plugins`** window shows a list of **`Installed and Built-In`** plugins, as shown in *Figure 12.10*; a user can switch them by clicking on the **`Enabled`** tick box. After each switch, you must save and restart your project to activate or deactivate the plugin:



Enabled — Epic Games, Inc.

MetaSound — BETA Version 1.0
A high-performance audio system that enables sound designers to have complete control over audio DSP graph generation of sound sources, via sample-accurate control and modulation of sound using audio parameters and audio events from game data and Blueprints
✓ Enabled — Epic Games, Inc.

MicrosoftSpatialSound — Version 1.0
Audio spatialization plugin using Microsoft's SASAPI service.
Enabled — Epic Games

MotoSynth — BETA Version 1
An experimental granular vehicle engine.
✓ Enabled — Epic Games, Inc.

Oculus Audio — Version 1.34.0
Oculus audio spatialization
Enabled — Support — Documentation — Oculus

OSCModulationMixing — EXPERIMENTAL Version 1.0
Utility Blueprint objects and functions enabling Control Modulation mix profiling using the OSC (Open Sound Control) protocol.
Enabled — Epic Games, Inc.

Resonance Audio — BETA Version 1.0
3D audio spatialization and room acoustics simulation plugin by Google.

***Figure 12.10**: Plugins window*

In Unreal Engine, there are two types of plugins:

- The first one runs inside the engine and adds new functionality to it or improves existing ones, like the *Steam VR* plugin, which is designed to support Steam API communication with VR application.
- The second type are used with *game* and are mostly designed to add new contents or simplify existing processes, depending on the scenario. For example, imagine you have to monitor the health bar for huge number of units inside a strategy game to use by tester team.

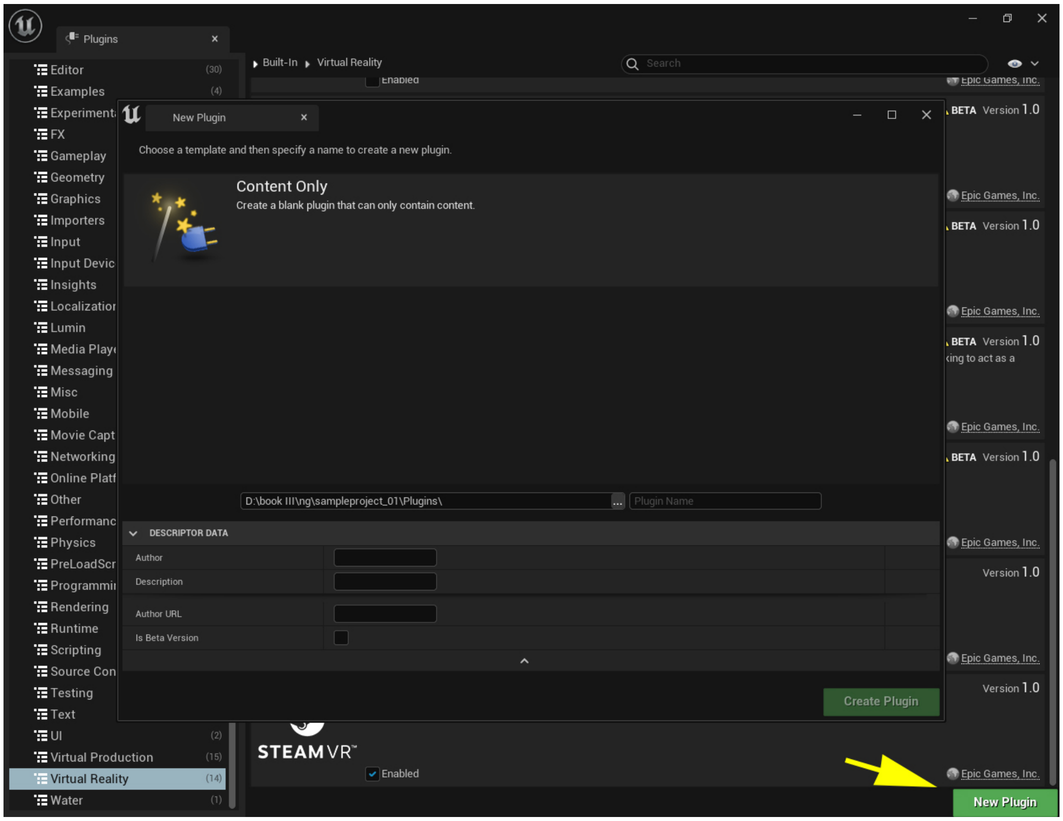
An example for first type is making a plugin to support new hardware like new VR headsets, new streaming data protocol, import and export data at runtime, using live cameras, etc. These plugins are designed to support communications between the engine and other data source which can be vital for the game or application developed by the Unreal Engine.

Example of second type can be creating a plugin to simplify the translucent materials using with static meshes in the scene. This plugin can increase the performance on devices like mobile phones which can't handle transparent material by their graphic card. So, it`s obvious that this plugin will be active in game and also in final version because it adds new functionality and increased performance of game. Another example of second type is developed a plugin that is designed to automatically get the date from game objects and show them with a UI inside the game. This plugin will be useful for tester, and it will also accelerate the testing process. This plugin is not necessary for game play and will be disabled in final version.

To make a new plugin, press **`New Plugin`** from the Plugins window; then, from the **`New Plugin`** window, as shown in *Figure 12.11*, you can customize your plugin. Plugin configuration files should be placed using the same convention as other configuration files:

- **Engine plugins**: **`[PluginName]/Config/Base[PluginName].ini`**
- **Game plugins**: **`[PluginName]/Config/Default[PluginName].ini`**

Each plugin needs an icon to display in the Editor's Plugin Browser. The image should be a 128x128 **`.png`** file called **`Icon128.png`** and should be kept in the Plugin's **`/Resources/`** directory. You can share or sell your plugins inside Marketplace as well:


Choose a template and then specify a name to create a new plugin.

Content Only

Create a blank plugin that can only contain content.

D:\book III\ng\sampleproject_01\Plugins\

***Figure 12.11**: Making new plugin by clicking on the New Plugin button*

Conclusion

In this chapter, we learned that Unreal Engine can support multiple platforms for developing game and applications. The engine will use platform SDKs as source and then adapt the packaging process based on the SDKs properties. The Device Manager and Project Launcher are useful tools to monitor and customize each platform's features. Also, users can customize SDK properties from **`Project Setting`**.

Now, it's time to sit and plan for a sample project or prototype for your portfolio. We know how to use the engine templates, customize them by creating new game objects, materials, and particle system, and then make them interactive and then establish a harmony of connected objects based on a Events, Interface and Enums. We also learned animation code and complex data processing with Unreal Engine.

It's time to make a plan and a schedule, and use Unreal Engine 5 to develop and build your own experience in the gaming industry.

Points to remember

- Disable unnecessary platform from final built sometimes slightly decreases built file size.
- Some plugins are not free or need special hardware on target machines. Sometimes, you may need to ask developers to program some code to support it. For example, automatically disable high-quality rendering by code because the platform graphic card is very low. At the end, you must make sure the final built can run for each target platform independently.
- Double-check local addresses of save and load of your data and projects files. Sometimes, missing path is a source of catastrophic crash in the release version.

Multiple choice question

1. `VR Preview` in `Play` options is responsible for __________.
    a. Converting an existing application to a virtual reality application
    b. Changing Project Setting automatically
    c. Previewing the game inside the headset by using current SDKs when the VR headset is connected to user`s local machine
    d. Simulating a standalone mobile phone application

Answer

1. c

Questions

1. What is the right setting for Android SDK?
2. How can you use Android Studio to install the latest version of Android SDK?
3. Which types of plugins are expensive to get enabled, in case of using graphic and processing resources from system?

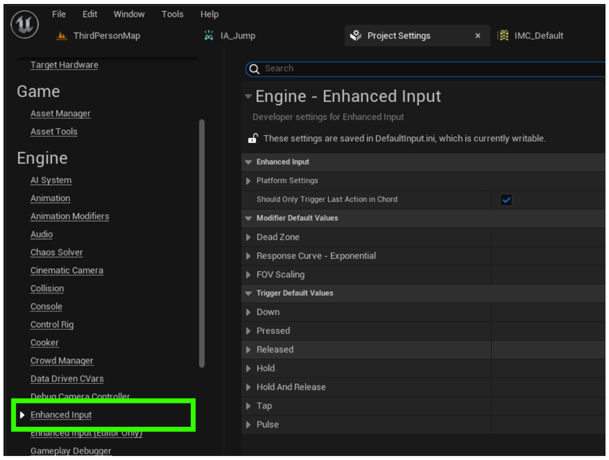
Key terms

- Device manager - The Device Manager is an editor tool that allows monitoring connected devices to engine and running processes, with abilities to switch them on/off.

- SDK – The SDK is stands for "software development kit" (also known as, devkit), and it is a set of software-building tools and libraries for a specific platform, including the built, framework, debuggers and libraries specific to an operating system.
- Android - Android is an **operating system** (**OS**) which mostly used for handle inputs and running applications on mobile devices, tablets, and headsets.
- Play modes - Unreal Editor allows users to instantly play preview and debug game from within the editor without waiting for files to save. Users can play preview in the actual editor as default output for play mode, or different outputs by switching play mode.
- Project launcher - The Project Launcher is an editor tool, used to make and release builds of projects to specified platforms.
- Plugins - Plugins are switchable collections of code and data, that can add runtime gameplay functionality, modify built-in Engine features (or add new ones), create new file types, and extend the capabilities of the Editor with new menus, tool bar commands, and sub-modes.

Appendix

The user input system in Unreal Engine works like an interface that receives user input from the input device like keyboard, mouse, VR controller, so on, and dispatch events by sending values to the code developed by users. This system is updated by Epic Game to a totally new approach with much more complexity and enhancements. The new input system, known as **`Enhanced Input`** is introduced with Unreal Engine 5.1. You can access its properties from **`Project Setting`** panel as shown in *Figure Appendix.1*:



These settings are saved in DefaultInput.ini, which is currently writable.

Should Only Trigger Last Action in Chord ✓

***Figure Appendix.1**: Properties of "Enhanced Input" in project setting panel*

At the same time, the old input system is currently functional with Unreal Engine 5, 5.1, and 4. You can check *Chapter 5, Project Templates and User Interaction* for details about its implementation. The old input system is functional in all mentioned versions, and developers can implement input procedures by using it.

Structure

In this appendix, we will discuss the following topics:

- Overview of using Enhanced Input
- Use correct Value Type
- Implement user interaction in Blueprint

Overview of using Enhanced Input

Users have variety of controllers for playing game or communicating with an Unreal Engine application. Each controller's input is basically a key or volume, which sends data to the target machine (like PC, VR or Xbox). For example, W, A, S, and D keys on keyboard are mostly used for movement action of player character. The mouse click simulates the shooting action in many games.

To implement an action, the engine provides an object known as **`Input Action`** which is located in the **`Input`** section of blueprint menu as shown in *Figure Appendix.2*:

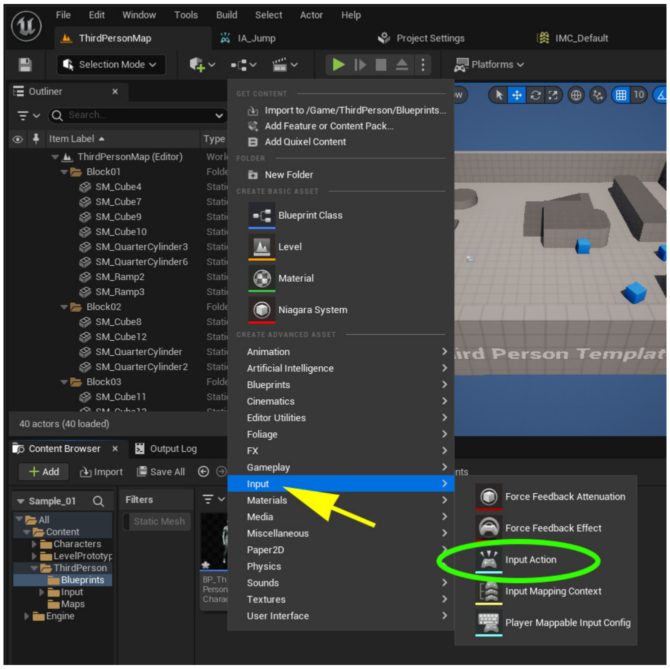


***Figure Appendix.2**: Locate Input Action object from Input in blueprint menu*

The **`Input Action`** object represents the title and the value of an interactive input to the engine. The engine will convert this object to an event, with some values involved with it. Then, the user can grab and use these objects to activate procedures inside game like shooting or movement, and so on. It looks straight forward but, having an input action object is not enough. We need another object to map or assign these actions to the actual key and volumes on the user`s controller.

For doing this, we need to use another object called **`Input Mapping Context`** which is located at the same location of **`Input`** Action as *Figure Appendix.3* shows:

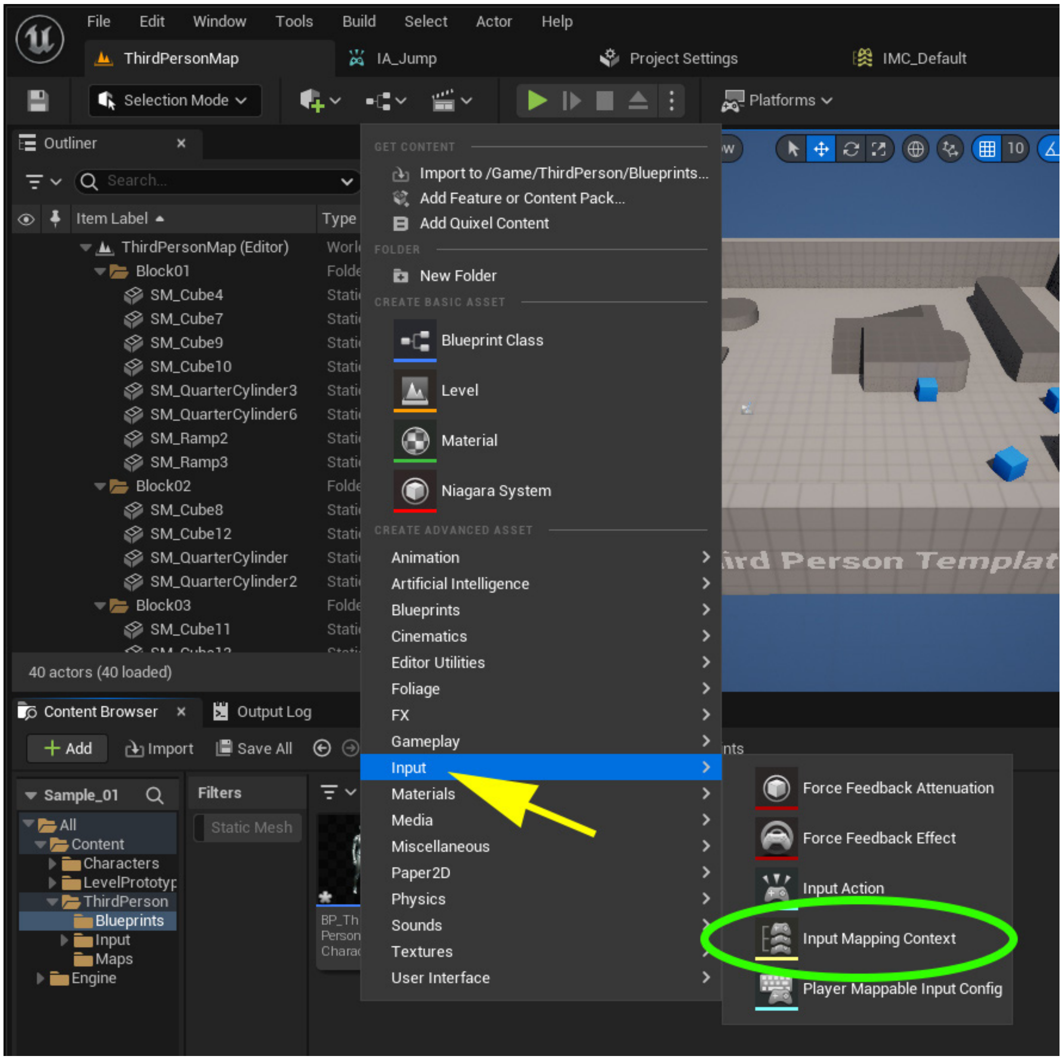


Figure Appendix.3*: Input Mapping Context in blueprint menu.*

This object is responsible to use **`Input Action`** objects and assign the actual keys/ volumes of controller to it from a list of default controllers provides by engine as default.

To get better understanding of relation between these objects, let us check the current input system in a third person template project. Make a project from third person template and from content browser, locate **`Input`** folder, and double click on **`IMC_Default`** object which is an **`Input`** Mapping Context object as shown in *Figure Appendix.4*:

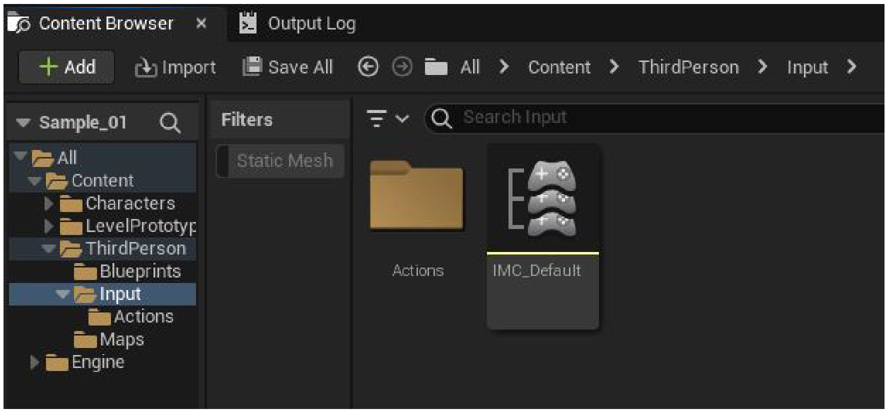


***Figure Appendix.4**: IMC_Default in Input folder.*

Then, the engine will automatically open the **`Details`** panel of this object as shown in *Figure Appendix.5*. As you see in the following figure, we have three input actions here by name **`IA_Jump`**, **`IA_Move,`** and **`IA_Look`** in **`Mappings`** dropdown panel. Each input action is assigned to a series of keys and volumes on keyboard, mouse, controller, and touch device which are basically the input devices for the third person template project by default. (Other template may have different mappings.) You can customize these input keys/volumes for each input action, remove them and even assign additional input keys/volume to mapping lists.

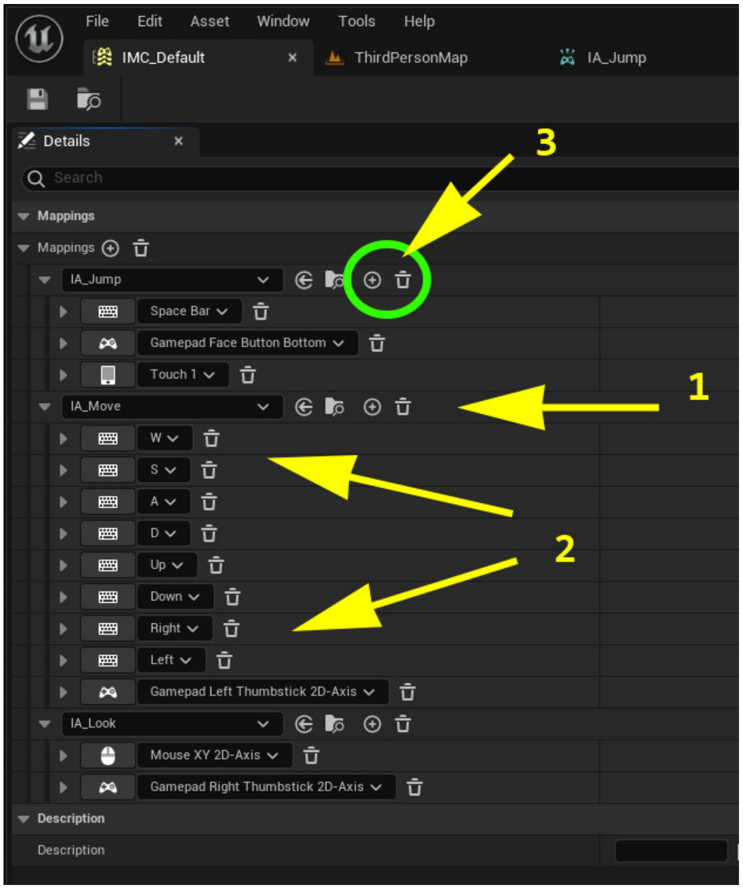


***Figure Appendix.5**: 1- The Input Action object dropdown list.*
2. Keys and volumes from listed devices. 3. Add or remove keys and volumes.

Now, let us see where these input actions (**`IA_Jump`**, **`IA_Move`** and **`IA_Look`**) came from. As the preceding figure shows, the **`IMC_Default`** object has three input actions which are made by Input Action object, located in **`Actions`** folder, inside **`Input`** folder in Content Browser as shown in *Figure Appendix.6*. You can create a new Input Action object here and then assign a key or volume of the listed controllers to it inside

`IMC_Default` | **`Input Mapping Context`** object to expand the user interactions with this project.

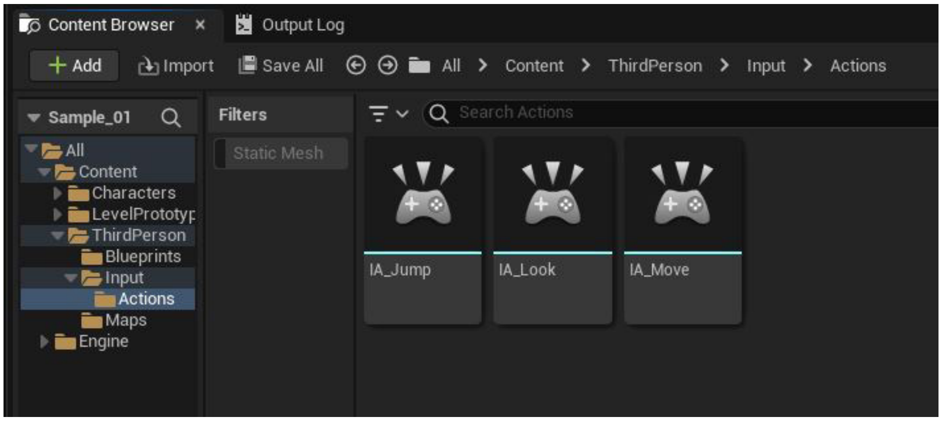


Figure Appendix.6*: Three input action object inside "Actions" folder.*

Using correct Value Type

The Value Type is a data which passes from input action object to input mapping context object, on each user interaction with controller. To understand the functionality of Value Type we will check the blueprint implementation of **`IA_Jump`** and **`IA_Move`** input action objects.

Double click on **`IA_Jump`** and **`IA_Move`** and the engine will open the **`Details`** panels of each. As you see the **`IA_Jump`** input action use a value type of "Digital(bool)" but the **`IA_Move`** uses **`Axis2D(Vector2D)`** as value type as shown in *Figure Appendix.7*:

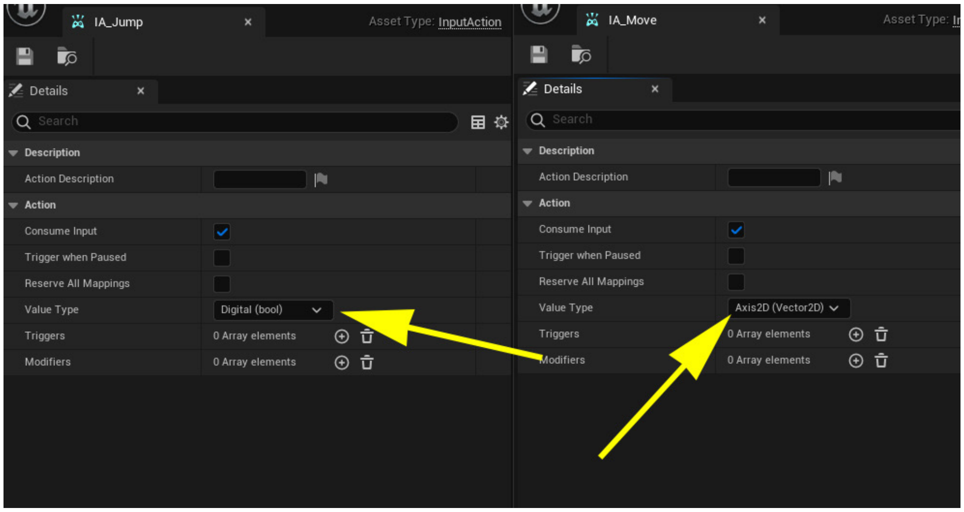


Figure Appendix.7*: "IA_Jump" and "IA_Move"details panel*

By selecting **`Digital(bool)`** as value type, the engine will generate a Boolean value of *true* or *false* on each interaction with input action object. For example, if you map a mouse button to it, on each click, it will send *true* to the engine and when you release the click, it will send *false*.

Now open **`BP_ThirdPersonCharacter`** blueprint and locate **`IA_Jump`** event as shown in *Figure Appendix.8*. As you can see, the **`IA_Jump`** event activates two functions which perform jump and stop jumping procedures. The **`IA_Jump`** object uses a Value Type as **`Digital(bool)`** as shown in *Figure Appendix.7*. In the blueprint code, **`IA_Jump`** object (which is implemented as an event), will provide a Boolean output named **`Action Value`** on each interaction by user. We add another **`IA_Jump`** event by right click in event graph and type **`IA_Jump`**, and then print out the **`Action Value`** on the screen from this event as shown in the following figure:

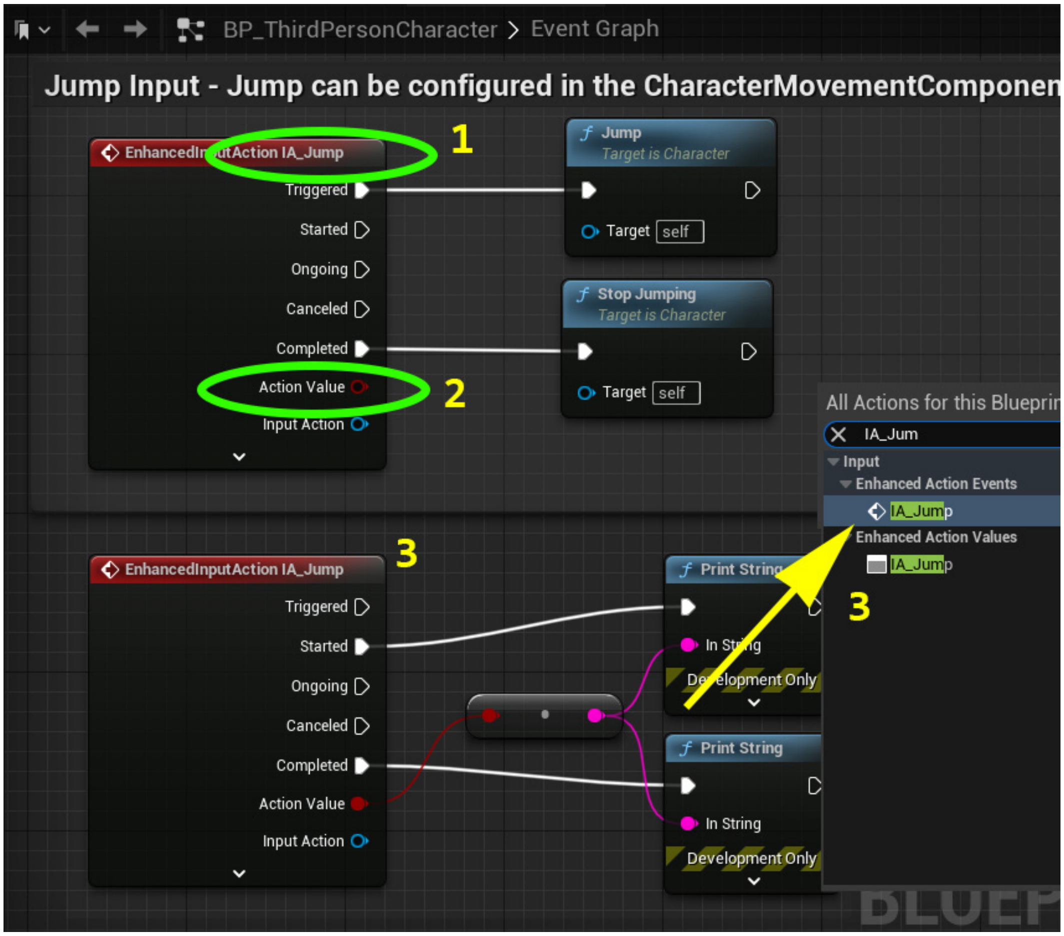
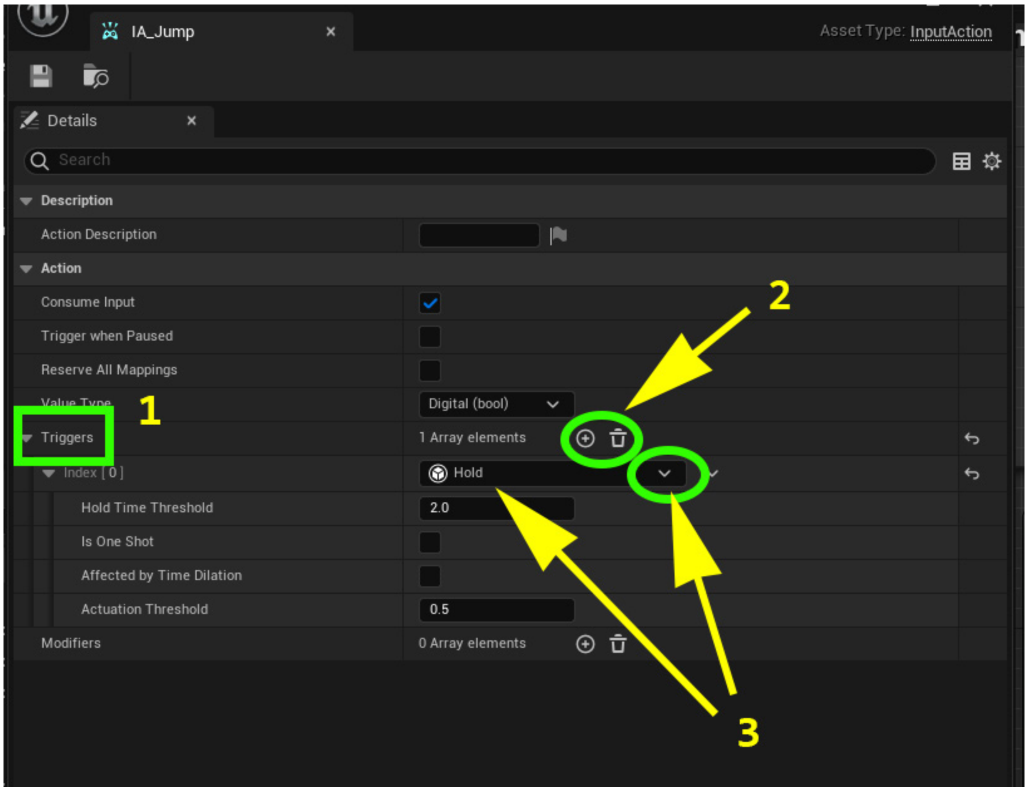


Figure Appendix.8: *1. Each "Input Action" object implement an event in blueprint. 2. Returned "Value Type" after the event get activated. 3. You can implement more than just one event from any "Input Action".*

As you see, each **`Input Action`** object can be implemented as event(s) inside blueprint code, and we can use the **`Action Value`** of each, to perform different scenarios by code. The **`Input Action`** objects can accept some procedures to enhanced using them without extra work on coding. Let us check one of them for **`IA_Jump`** action:

1. Open **`IA_Jump`** object and locate **`Triggers`**.
2. Then, add **`Hold`** trigger from the dropdown list to the array of triggers as shown in *Figure Appendix.9*.
3. Set **`Hold Time Threshold`** to 2 seconds as well.
4. Now, run game and press spacebar and hold your finger for more than 2 seconds. As you see, there is a delay before jump procedure get activated, and the reason behind that came from adding **`Hold`** trigger in input action object.



Description
Action Description
Action
Consume Input ✓
Trigger when Paused
Reserve All Mappings
Value Type: Digital (bool)
Triggers: 1 Array elements
Index [0]: Hold
Hold Time Threshold: 2.0
Is One Shot
Affected by Time Dilation
Actuation Threshold: 0.5
Modifiers: 0 Array elements

***Figure Appendix.9**: 1. Locate "Triggers" dropdown panel.*
2. Using these icons to add or remove elements. 3. Find and add "Hold" from dropdown menu.

Keep in mind that some triggers, will activate different execution pins in the event node of belonged input action in blueprint code. This means depending on your controller type, (mouse, touch screen, VR, and so on) you can wisely select/combine triggers to add more complexity and accuracy to your input actions when they get triggered by user`s controller.

There are another type of procedures which are used to modify the **`Value Type`** that send an input action object. These procedures can be assigned in **`Modifiers`** dropdown menu with the same method as we did for **`Triggers`**. To check this, let us multiply the value type of **`Axis2D(Vector2D)`** in **`IA_Move`** input action, by a scalar value and show them on the screen.

As you see in *Figure Appendix.10*, **`IA_Move`** event in third person character blueprint receives 2 values of x and y for action values, why? Since the **`IA_Move`** input action objects have **`Value Type`** as **`Axis2D(Vector2D)`**, it provides values as x and y on each interaction by user through its belonged event. Here, we modified the code by printing the values of X and Y on the screen. At this stage you will see 0, 1 and -1 on screen (we will explain these values shortly) as shown in the following figure:

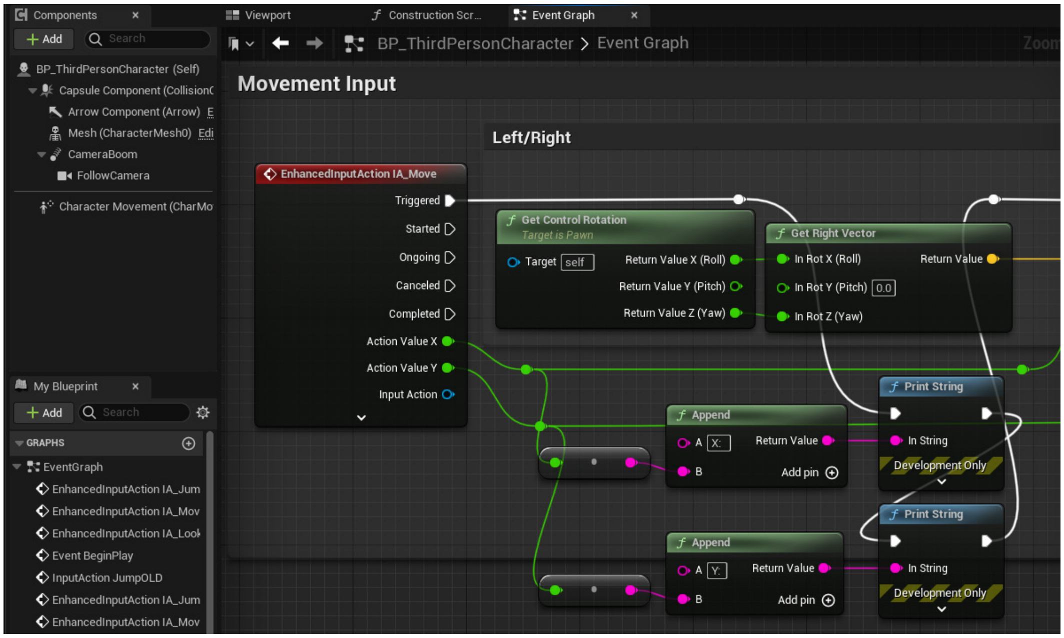
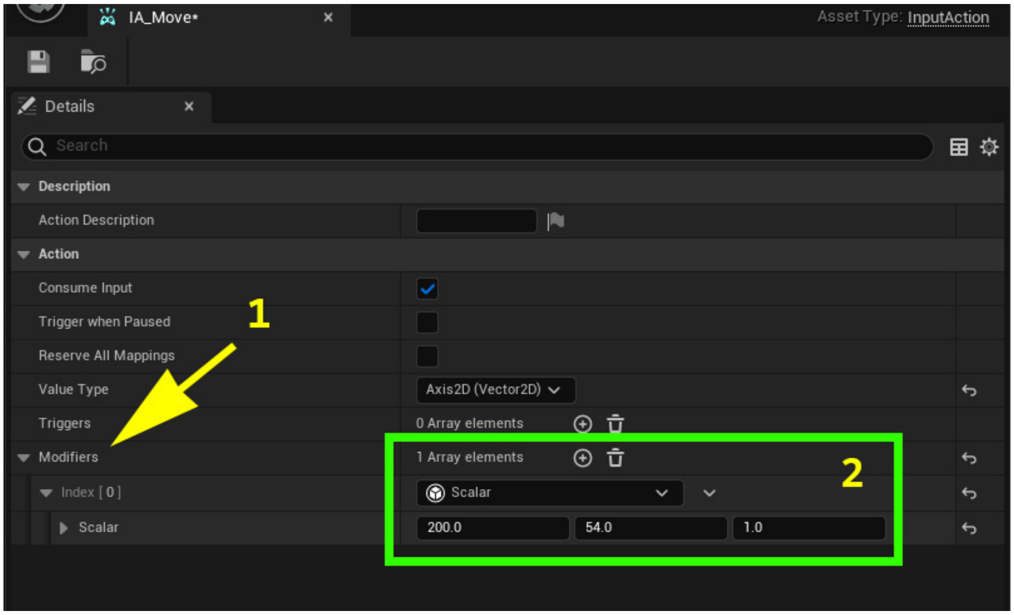


***Figure Appendix.10**: Showing action values of X and Y, from "IA_Move" event on the screen by adding Print String and Append node.*

Now open **`IA_Move`** input action objects, locate **`Modifiers`** and add a **`Scalar`** modifier to this action. Then set x and y values of this modifier to **`200`** and **`54`**. (these values are just examples and can be anything else) *Figure Appendix.11* shows these settings:



Description	
Action Description	
Action	
Consume Input	✓
Trigger when Paused	
Reserve All Mappings	
Value Type	Axis2D (Vector2D)
Triggers	0 Array elements
Modifiers	1 Array elements
Index [0]	Scalar
Scalar	200.0 / 54.0 / 1.0

Figure Appendix.11*: 1. Locate Modifiers dropdown panel 2. Add one modifier to the array, and select "Scalar" from dropdown list, then set value x and y to 200 and 54*

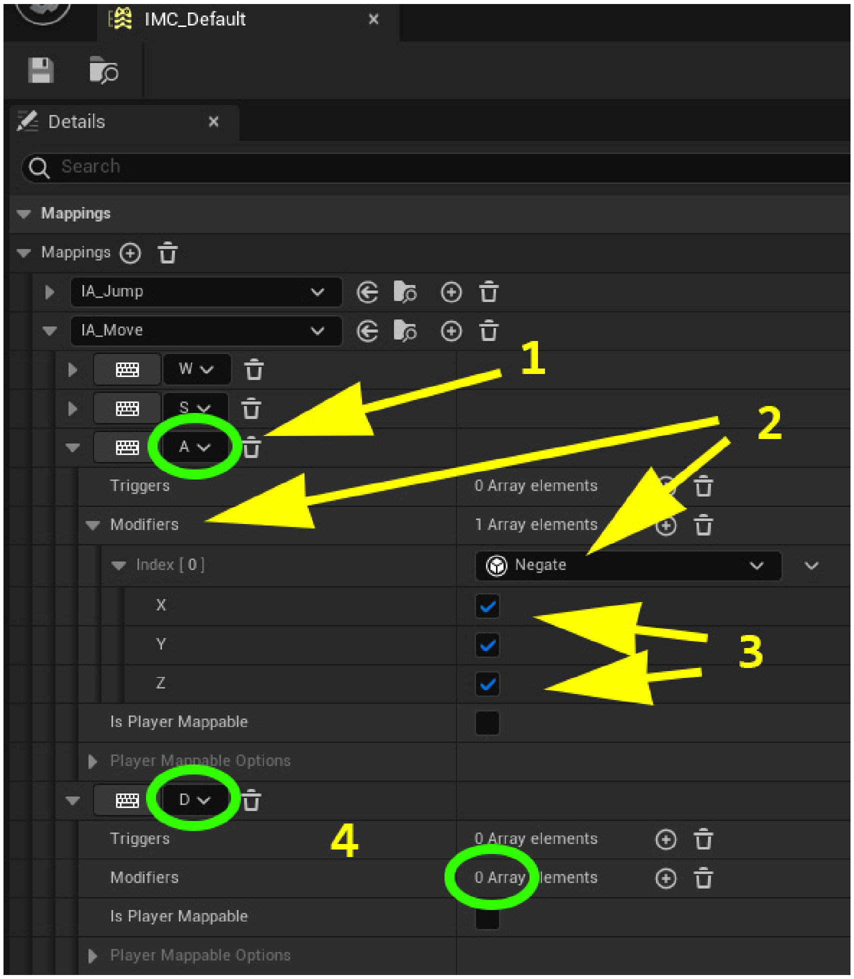
Now, to monitor the result of this modifier we need to print the values of the **`IA_Move`** action. As you see in Figure Appendix.10, it is already implemented.

The output will be 54.0 and -54.0 when you press W and S key which causes movement on Y axis. It will be -200.0 and 200.0 when you press A and D key which causes movement on X axis.

You may get surprised why we have an output of -200 and 200 when we press A or D keys. This caused by another modifier which you can assign to the value of your input action **`Value Type`**, but this one is handled by **`IMC_Default`** object itself.

In addition to assigning modifiers to an **`Input action`** object, you can assign modifier to each key/volume inside an **`Input Mapping Context`** object. This enhanced and supported more customizations over your input mapping when you deal with each *key* or *volume* on user's controller. Perhaps, you need to modify a **`Value Type`** for VR controller differently from using keyboard. Here, you can involve this difference on your input **`Value Type`** from an input mapping context object like **`IMC_Default`** in this project.

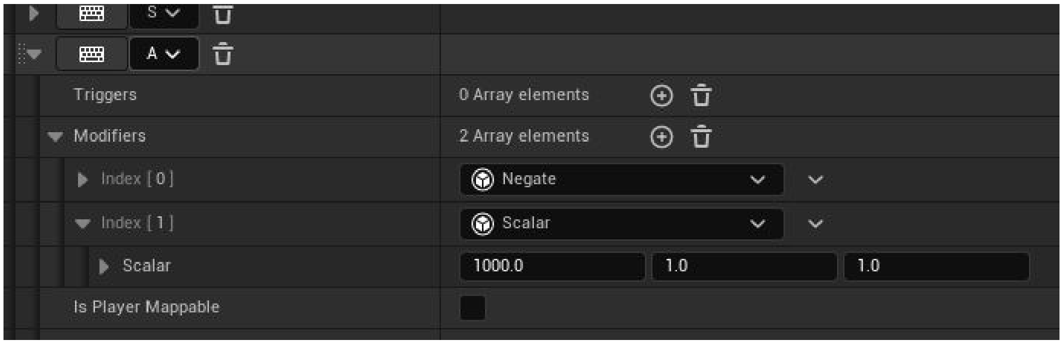
Open **`IMC_Default`** object and check the **`Modifiers`** drop down for A and D keys as shown in *Figure Appendix.12*. As you see in the figure, there is a modifier called **`Negate`**, assigned to A key which works on X and Y, and Z index of the "Value Type" passed by "IA_Move" input action object. The "Negate" modifier will multiply the input by -1 and that's why we have -200 when we press A key.



Triggers 0 Array elements
Modifiers 1 Array elements
Index [0] Negate
X ✓
Y ✓
Z ✓
Is Player Mappable
Triggers 0 Array elements
Modifiers 0 Array elements
Is Player Mappable

***Figure Appendix.12**: 1. Check key A dropdown panel. 2. The "Negate" modifier is assigned in Modifiers panel. 3. The values for X, Y and Z are ticked, means they will multiply by -1. 4. This modifier is NOT applied to key D.*

There are numbers of modifiers which you can apply with an **`Input Action`** object or **`Input Mapping Context`** object, and each affect the value type differently. You can also add more than one trigger or modifier to these objects. The engine will save them in an array and will execute all, by index order upon user interaction with controller. Try adding another modifier to A key as shown in *Figure Appendix.13* and see how the values are changing:



S

A

Triggers: 0 Array elements

Modifiers: 2 Array elements

Index [0]: Negate

Index [1]: Scalar

Scalar: 1000.0 | 1.0 | 1.0

Is Player Mappable

***Figure Appendix.13**: By adding a "Scalar" modifier to key A, the value from input action object (IA_Move), will get multiplied by 1000 and the result will send through the event`s "Action Value" by X index, when key A is pressed.*

As mentioned earlier, some modifiers work fine only with **`Axis2D(Vector2D)`** value type (or Digital(bool) or Axis3D(VectorD)). So, you must set the right modifier for value type, compatible with your input action and trigger that belong to it. This is the art of *user input programming*. It gets more serious when you try to program complex input scenarios like what we have in fighting games like *Mortal Combat* and *Tekken*. In those type of games, input from users through controller can have delay, use combinations of multiple keys, or send different values. You can wisely program such a complex scenario via enhanced input system in the engine, by using right value types, triggers, and modifiers.

Implement custom user interaction in Blueprint

Now, let us make a new input action, map it to keyboard with a new input mapping context and implement it in third person character blueprint code. As an example, we will boost the player speed to 1200 by pressing E or Q key on keyboard and reset it to default speed which is 500 when releasing these keys. To review previous contents, we implement this action with two different methods.

1. First create a new **`Input Mapping Context`** object at the same location as **`IMC_Default`** and rename it to **`IMC_01`**.

2. Then, open a third person character blueprint, locate **`Begin Play`** event and use **`Add Mapping Context`** blueprint node to add your new input mapping context object to the code as shown in *Figure Appendix.14*.

3. Do not forget to connect **`Target`** pin to **`Enhanced Input Local Player Subsystem`** blueprint node.

4. This is essential to activate your **`IMC_01`** mapping context object after running game.

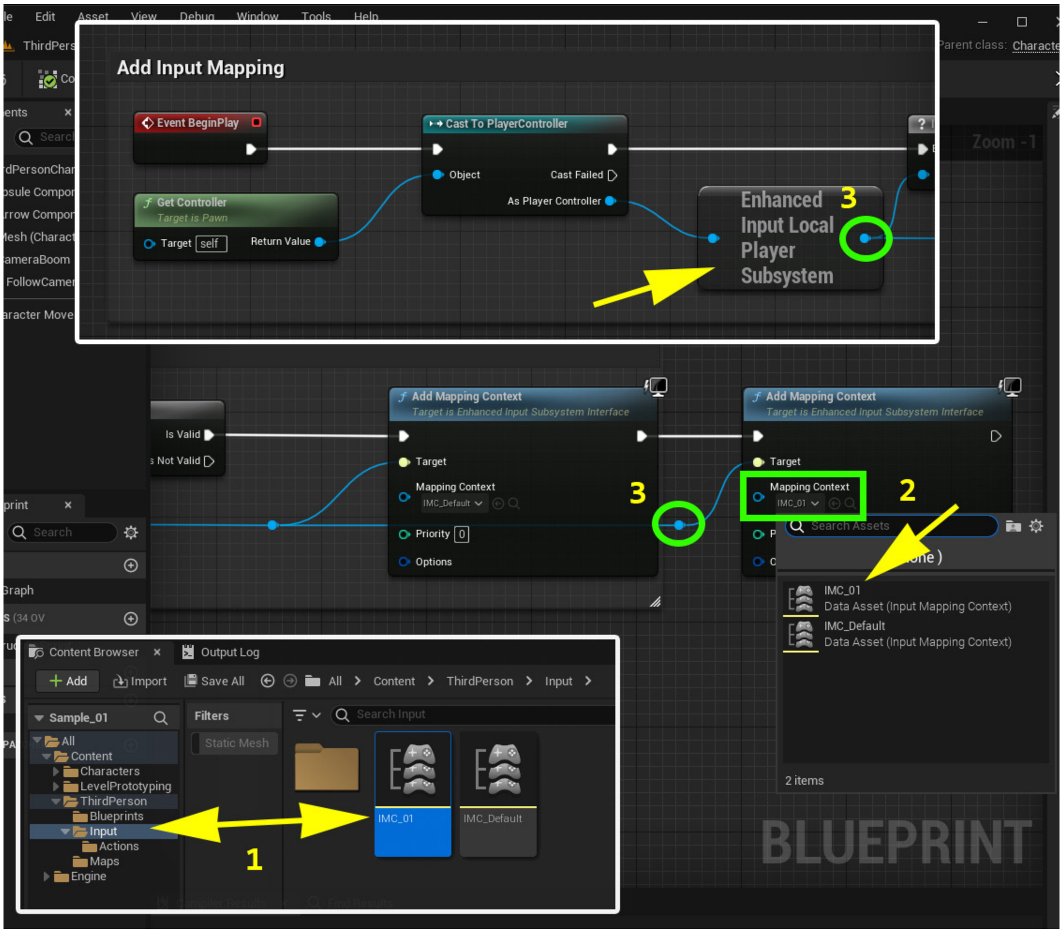
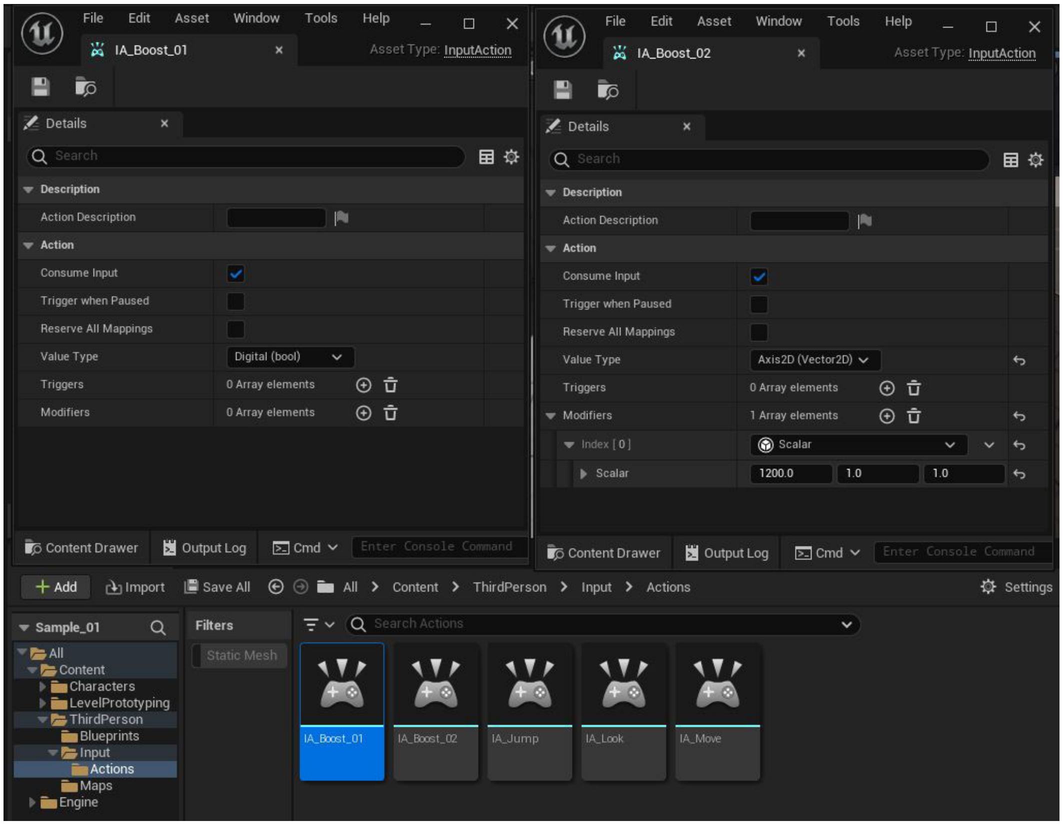


Figure Appendix.14: 1. Make new "Input Mapping Context" object in "Input" folder. 2. Add this object to the end of "Begin Play" procedures in blueprint by using "Add Mapping Context" node. 3. Connect Target pin to "Enhanced Input Local Player Subsystem" blueprint node.

5. Now, make two input action objects in **`Actions`** folder, and name them **`IA_Boost_01`** and **`IA_Boost_02`**.

6. Then, open each and set the **`Value Type`** to **`Digital(bool)`** for **`IA_Boost_01`** and **`Axis2D(Vector2D)`** for **`IA_Boost_02`**. Then, add a **`Scalar`** modifier to **`IA_Boost_02`** with value of 1200 for x. The input action objects should look like *Figure Appendix.15* at the end:



IA_Boost_01 — Asset Type: InputAction

Description	
Action Description	
Action	
Consume Input	✓
Trigger when Paused	
Reserve All Mappings	
Value Type	Digital (bool)
Triggers	0 Array elements
Modifiers	0 Array elements

IA_Boost_02 — Asset Type: InputAction

Description	
Action Description	
Action	
Consume Input	✓
Trigger when Paused	
Reserve All Mappings	
Value Type	Axis2D (Vector2D)
Triggers	0 Array elements
Modifiers	1 Array elements
Index [0]	Scalar
Scalar	1200.0 / 1.0 / 1.0

All > Content > ThirdPerson > Input > Actions

IA_Boost_01, IA_Boost_02, IA_Jump, IA_Look, IA_Move

Figure Appendix.15: *Making tow input action objects with different Value Types and Modifiers in "Actions" folder*

7. Now, we need to assign E and Q key to these two input actions. Open **`IMC_01`** input mapping context object, add **`IA_Boost_01`** and **`IA_Boost_02`**

to **`Mappings`** dropdown panel, and set E and Q as key for each one as shown in *Figure Appendix.16*. Now, the actions are mapped to the keys in keyboard:

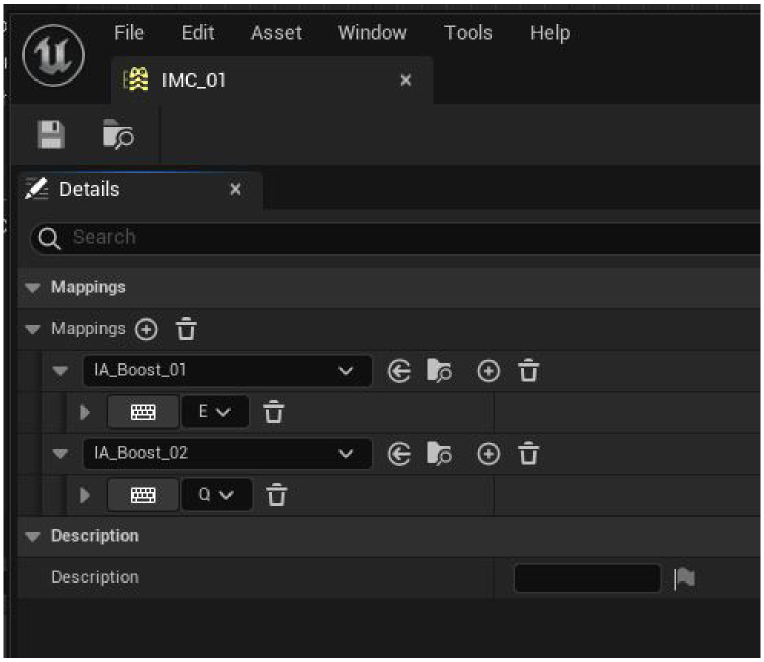


Figure Appendix.16*: Mapping previous input action objects to E and Q key in “IMC_01” input mapping context object*

8. Finally, open third person character blueprint again, right click in event graph, type input action names **`IA_Boost_01`** and **`IA_Boost_02`** and then add an event for each in the event graph.

9. As you see in *Figure Appendix.17*, we use Boolean **`Action Value`** from **`IA_Boost_01`** event for switch between two different speeds. Also, we use the x value of vector 2D **`Action Value`** from **`IA_Boost_02`** event to affect the movement speed.

10. Keep in mind, both **`IA_Boost_01`** and **`IA_Boost_02`** input actions, result in the same behavior in-game, but the value types, and implementations of each input actions, are different.

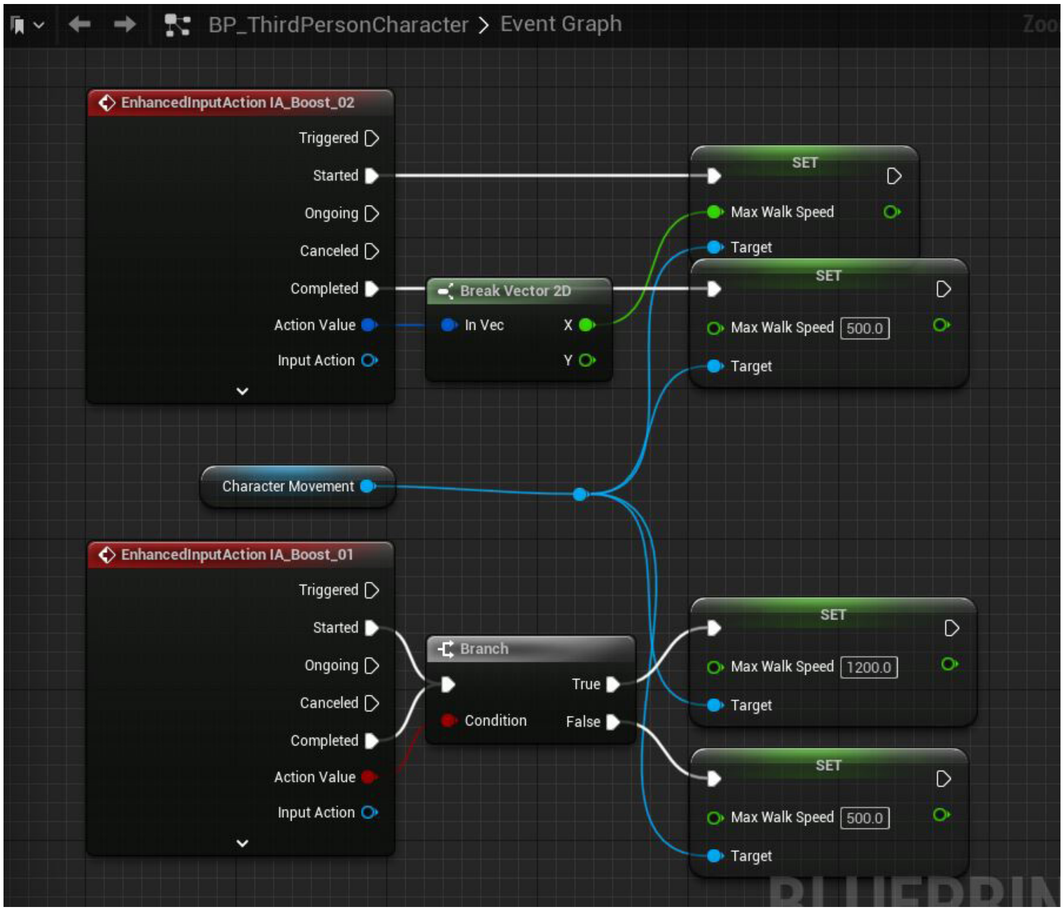


Figure Appendix.17*: Using two different input actions to produce same result on boost speed of the character movement*

Conclusion

The *Enhanced Input* in the engine expands the flexibilities and complexities of implementations and mapping user controllers into the engine code. Each controlling device meet different combination of modifiers and triggers. There are multiple method to implement same behavior with user input by using triggers, modifiers, and a combination of these. The details for each type of triggers, and modifiers is mostly related to user interface programming advanced topics which is out of focus of this book.

Key terms

- Enhanced Input
- Input Action
- Input Mapping Context
- Value Type
- Digital(bool)
- Axis2D(Vector2D)
- Axis3D(Vector)
- Action Value
- Triggers
- Modifiers
- Negate
- Scalar

Index