Tri Hai Ha

GAME DEVELOPMENT WITH UNREAL ENGINE

Bachelor's thesis

Bachelor of Engineering

Information Technology

2022

South-Eastern Finland
University of Applied Sciences

South-Eastern Finland University of Applied Sciences

Degree title	Bachelor of Engineering
Author(s)	Tri Hai Ha
Thesis title	Game development with Unreal Engine
Commissioned by	
Year	2022
Pages	52 pages, 2 pages of appendices
Supervisor(s)	Reijo Vuohelainen

ABSTRACT

Nowadays, Unity and Unreal Engine has been the top two software that been used in the game industry to create, teach, and learn by many huge companies or individuals to create games. However, not many people know which program to learn or where to pick up when they want to get into the industry and become a game programmer.

The thesis provided a brief understanding of the definition of Unreal Engine, why it is a powerful tool in creating a game project, the possibility of Unreal Engine in the game industry and the reason why we should learn it. Along with Unreal Engine, we also get to know about Unreal Engine Blueprint Visual Scripting and C++, the programming language behind the Engine to unlock its full potential.

The goal of the thesis was all about how approach to Unreal Engine by providing all the information and knowledge, showing the unexperienced developer through the game project that created by Unreal and C++ in the thesis, so people can understand, learn, and create games by using the engine.

Keywords: Game development, C++ programming language, Unreal Engine

CONTENTS

1 INTRODUCTION

From the first piece of video game code written after 3 years since World War 2 ended, in 1948, Alan Turing and David Champ wrote a theoretical chess simulation called “Turo Champ”, which was actually the first piece of video game code that was ever written. Nowadays, with technology significantly evolving every moment, the game industry has also taken many big steps to become one of the top industries worldwide, providing entertainment for people with almost 20 games releases every month by independent developers, small indie game companies to triple A game companies that deliver masterpieces.

Along with the growth of the industry, the technology also needs to be evolved so people can use to create magnificent product. The two major game development programs at this current time are Unity and Unreal Engine. Both are incredible pieces of software and have earned their spot in the game industry by making popular games like “Cuphead” by Studio MDHR, “Hollow Knight” by Team Cherry for Unity, and “Fortnite” by Epic Games, “Dragon Ball FighterZ” by Arc System Work and Bandai Namco Entertainment for Unreal Engine. For a person who has the passion for game developing, wants to become a game programmer but don’t really know where to pick up, which engine should you put your time on and learn? As mentioned, both game engines are different and they are amazing, but after spending my time learning about both of them, I saw the potential of Unreal Engine and the reason why it earns more success and shines more than Unity in the game industry.

The goal of my thesis is to introduce and explain Unreal Engine throughout the process of creating a game project. We will go through the basic of Unreal from understanding the engine, Unreal Engine Editor interfaces and other elements in creating a game to learn coding with C++, the games industry standard language. C++ is a more complex compared to C# but once you understand it, using Unreal Engine will be a lot more proficient since you have unlocked more useful tools by learning the language. Creating game project from the most basic terminal games will help you understand deeply about

coding in C++ from simple to complex and understand more about Unreal Engine Framework.

2 INTRODUCTION ON GAME ENGINES

Game engine is specifically developed to create video games in different platform, such as computer and mobile phone. Currently there are many game engines exist in the game industries, but the two most popular game engines that most companies and studios are using are Unity and Unreal Engine.

2.1 Unity

Unity is a cross-platform game engine developed by Unity Technologies that supports 2D and 3D graphics, and uses C# for scripting (Unity (game engine), Wikipedia). Launched in 2005, Unity has been a people's choice engine for indie game developers to create video games and simulations on computers, consoles, and mobile devices. Besides that, Unity can also be suitable to create virtual-reality (VR) and augmented-reality (AR) games and has earned huge success through it. Since then, new content and functions got updated every year, and creators can develop and sell their created assets to other game makers through Unity Asset Store. Alongside the huge community, Unity Asset Store filled with free and pay-to-use assets for developers to share and use in their projects. By 2018, there had been approximately 40 million downloads through the digital store. Overall, Unity is an easy to interact and comfortable engine to learn and start with.

2.2 Unreal Engine

Unreal Engine is a game engine developed by Epic Games, first showcased in the 1998 first-person shooter game "Unreal". Initially developed for PC first-person shooter, it has been successfully used in a variety of other genres, including platformers, fighting games, MMORPGs and other RPGs as well. Written in C++, Unreal Engine features a high degree of portability, supporting a wide range of platforms (Unreal Engine, Wikipedia). Unreal Engine is a complete suite of creation tools for developing from independent hits to blockbuster franchises. Unreal Engine delivers high quality which makes many

studios choosing it to develop fantastic projects in video games or even movies and films because of its proven performance. Over the course of 2 decades with many system updates, Unreal Engine has become one of the two most trusted and reliable engines in the world.

2.3 The comparison and potential of Unreal

There are many topics and discussions about which engine is better between Unity and Unreal. Each engine has its own differences, and they are all incredible software to use in developing games. After doing research and through personal experience on both engines, I have gathered useful information to compare Unity and Unreal, and to answer the reason why I suggest we should start learning Unreal Engine.

Let's start with the similarities between the two engines. Both tools can produce triple-A quality graphics, have great bridges between most of the industry-standard software and provide an extensive toolbox including editor, animation, physics simulation, VR support and more (Simran Kaur Arora, 2021).

The differences between Unity and Unreal come from many aspects:

- Definition: Unreal is a source available game engine, and Unity is a cross-platform game engine.
- Programming language: Unreal uses C++ for development, and Unity uses C#.
- Graphics: Both tools have good graphics but Unreal is preferred over Unity because of the AAA quality graphics.
- Source code: Unreal has open source making the development process easier. Unity on the other hand does not provide open-source code, however it can be bought.
- Rendering: Unreal supports faster rendering making post-processing even faster. Rendering is slow in the case of Unity thus processing of projects is also slow.

According to Buvesa Game Development (2021), Coding language could be the determining factor: Unity with C#, and Unreal with C++. Generally, C++ is

considered to be the more difficult and more complex to learn, but Unreal Engine has visual scripting called Blueprint, a great alternative to coding allows you to do the same thing yet with no coding require, develop logic to the game. Unity has Bolt but Blueprint is better and more developed, and also be able to combine with C++, so you can prototype the game faster.

Graphics is an important part in creating games and a game engine that limits your graphically is going to limit your potential. For both Unity and Unreal, they are capable of creating amazing graphical fidelity for games. Most users found that Unreal has a slight edge over Unity in the quality of its visual effects. It can create photorealistic visualizations that immerse gamers and allow them to travel freely in a stunning new world and incorporate high-quality assets from a variety of sources. Lighting in Unreal looks more accurate and smoother compared to Unity.

Nowadays the difference between them is negligible since Unity has been slowly closing the gap with Unreal on graphic fidelity but requires more work. Unreal has Quixel Megascan, which is a library containing photorealistic materials available for free. Unreal has always had an edge on Unity because of its higher potential for photorealistic graphics. Animation is also a thing where Unreal Engine shines with its powerful rendering capabilities and top-notch visual effects.

Scripting tools for both platforms allow you to script your game from end-to-end. Either platform will provide you the functionality you need to quickly write out your game. When talking about built-in tools meant for development, Unreal has so many useful built-in tools and they make sure that all their stuff works with all their platform, and you don’t have to worry about third-party problem.

For multiplayer games, Unreal Engine already has its background from creating big multiplayer games like Unreal Tournament to Fortnite. Creating a successful Multiplayer game is challenging, Unreal Engine makes the process much more beginner friendly and archival with the built-in options. However, to create this type of game, we need to learn a lot but it feels much more archivable.

Though Unity is more like an all-rounded Engine for any developers and offers a better entry into the industry, Unreal has always been top choice for triple A studios in making games and has earned huge success through games with outstanding graphics. For fine-tuned graphics, fast render speeds, Unreal Engine is the choice for you to make enterprise-level game or even for indie game developers who want to make that extra quality into their game. Even though the learning curve will be more challenging but learning Unreal Engine rewards you with mind-blowing graphics capabilities and the possibility in the future is unimaginable.

Through the comparison and pointing out the highlights of Unreal Engine, we have seen the potential and possibility that the engine can do with game developing. Where exactly should we start and how can we prevent ourselves from being overwhelmed with the amount of knowledge inside Unreal? By learning and understanding about C++, the foundation we earn through it will help us in preparing our first step into Unreal Engine.

3 PROGRAMMING LANGUAGE C++

C++ is a general-purpose programming language created in 1979 by Bjarne Stroustrup – a Danish computer scientist as an extension of the C programming language, also known as “C with Classes” (Wikipedia). C++ is considered a middle-level language, a combination of all the characteristics and features of the low-level and high-level languages. C++ can be used for embedded programming, system programming and game programming. C++ is one of the most popular coding languages all around the world. According to TIOBE and PYPL (worldwide) Index, C++ still remains in top 5 of the most used programming languages along with Python, C, C# and JavaScript. Even though it has been existed for 35 years, but C++ is never outdated.

Why should we learn C++? Before learning we should have a broad view of what can the language do and build, then learn ways to start learning C++ and know the kinds of opportunities that C++ can provides to you. C++ was used to create many popular games like WoW, Diablo, StarCraft series, Doom, develop engines like Unreal, coscos2dx and also create graphics application

such as Adobe Premiere, Photoshop, Illustrator. Database also an example for the use of C++, for example Mysql. Finally, C++ is very important in Operating system for MAC OS and Microsoft Window. C++ is a great language to use whenever you have large buffer and in cases where you have high concurrency and need minimum latency. This applies to server application and games. According to Erin Schaffer (2022), the reason why C++ remains one of the most popular programming languages because it has many solid features and advantages, such as:

- Exception handling is built into C++. It's a tool that separates code, detects and handles exceptional circumstances arise while running programs.
- Function overloading is the process of having two or more functions with the same name but with different parameters. This feature allows you to define more than one definition for a function name or an operator in the same scope.
- C++ supports dynamic memory allocation, which helps free up and allocate memory.
- C++ standard template library (STL) is filled with templates of ready-to-use libraries for various data structures, arithmetic operations, and algorithms.
- Object oriented programming concepts allow you to treat data as objects and classes.
- C++ is a multi-paradigm language. This allows you to choose a single approach or mix aspects of different programming paradigms, such as generic, imperative, and object-oriented.
- C++ is versatile and has a large job market.
- C++ is great for resource-intensive application because of its scalability and performance capabilities.

Overall, C++ is a great language to learn if you want to have a deep understanding of how computers work. It lets you to get hands-on with low-level programming concepts and helps you to understand how computer operates. From the aspect of a game programmer, you will have many different job opportunities since Unreal Engine create such a huge impact on the game industries at the moment.

So, how to start learning C++? If you're completely new to programming, you will need to take time to familiarize with the fundamental of programming concepts. You first begin with C++ basic such as arrays, construction, and iterators. After knowing the basics, you can learn and explore more concepts like pointers, vectors and many more concepts. Now, let's start with the very first program for beginners in C++, the Hello World Program.

```
#include <iostream>
using namespace std;

int main()
{
    std::cout << "Hello World!";
    return 0;
}
```

Figure 1. The Hello World Program

By running this program in Figure 1. The Hello World Program, Hello World will be printed. In this very basic program, we have 2 parts: header file and the main function. The first part of the program is the header file. Header files are generally used to import features into the program. Using namespace std (standard) meaning that using all the things inside the standard library. The second part of the program is the main function. This is the function where the execution of the program starts. Inside the main function of the example printing out Hello World by using the "cout", which is used to print the output, and return 0 on the next line will indicate the program that nothing will return and the program will be executed successfully in Figure 2. Testing the debug console.

Figure 2. Testing the debug console

After understanding the first basic program, we will continue to the basics of C++.

Types and variables: in C++ there are different types of data defined by their keywords which is responsible for defining variables, such as Boolean used when we have 2 values either true or false and these values are used when there are conditions or Integers are used for values and they can be both positive and negative. There's also characters and float, etc.

Variables are used with the datatype to store values. This is the syntax for variables:

Datatype VariableName;

Following the syntax to define variables, we will need to mention the datatype, and then the variable name of our choice.

There are six different types of variables:

- **Int** to stores integers
- **Float** to stores decimals floating point number
- **Double** to defines floating point number
- **Char** to stores single characters
- **Bool** to stores Boolean, value with a true or false state
- **String** to stores strings of text

Data types are the classifications for different kinds of data you can use in a program. Data types tell our variables what data they can store. There are three data types in C++:

- Primitive data types are the built-in data that you can use to declare variables. They include **integer**, **character**, **Boolean**, **floating point**, **double floating point**, **void**, and **wide character**.
- Derived data types are derived from the primitive data types. They are **function**, **reference**, **array,** and **pointer**.
- User-Defined data types are defined by the programmer.

Arrays are one of the most important and widely used concepts in C++. Arrays can be defined as a group of similar kinds of elements and they are stored in contiguous memory locations one after another. Arrays make it possible to store multiple values of the same datatype into a single variable.

Syntax to define Array:

Datatype Array_name [number of elements]

Strings are defined as a group of characters used to represent text in the program and we can perform operations on strings to manipulate it. The syntax of string contains a collection of character surrounded by double quote.

Datatype string_name = “value”

There are two ways to create a string in C++. In a C-style string, the collection of characters is stored in the form of arrays, basically they’re arrays of type character. By using string object, we can create string object to hold a string. They’re implemented in the standard library which we must include in the program by using #include.

In Figure 3. If-else statement, **If-else** are two conditional statements used when we want to run the code based on conditions. A block of code inside If statement will run, only if the condition of If statement is true, otherwise it will run the block of the Else statement.

```
if (number == 5)
{
    std::cout << "correct";
}
else
{
    std::cout << "incorrect";
}
```

Figure 3. If-else statement

Loops has 2 types. For loops are used in C++ to repeat the execution of code, instead of repeating the code multiple times. The syntax has 3 parts

For (initialization; condition; updation)

{

body

}

Initialization is used to initialize the loop, condition is to determine when to end the loop and updation is used to update the loop variables.

While loops are used when we don’t know the exact number of times the loop should repeat. It will repeat the statement till the given condition is true. Once the condition becomes false, the control passes outside the loop.

Syntax of while loop

While (condition)

{

body

}

Operators are symbols that manipulate our data and perform operations. There are four kinds of operators in C++:

- Arithmetic Operators are used for mathematical operations.
- Assignment Operators are for assigning values to our variables.
- Comparison Operators compare two values.
- Logical Operators determine the logic between values.

Object is a collection of data that we can act upon. An Object in C++ has an attribute and method. We can construct objects using a class. To create a class, we use the “class” keyword, and we must define an access specifier, such as **public**, **private**, or **protected**. Once we define our class, we can define the attributes and objects.

Pointers are memory addresses. We can notice when we are using pointers by basically have a star next to a type in our code. It’s not as in multiplication but when it’s next to a type or a variable, it’s going to be a pointer. So why would we use pointer? The benefit is to save us moving things around in memory, like references. For example, we have complex game objects, and we do not want to move or copy it from one place in memory to another, so pointers save us from doing that. We simply refer to things by reference rather than the actual value, and we can mostly point any object. The only drawbacks of pointers that we can lose control of our data, lose track of the data or the object itself. There are potential disadvantages but there are also ways of managing it as well. An example of Pointer’s syntax:

FType* NameOfPointer

In all cases, the type of object pointed to is FType.

Finally, **functions** in C++ are a group of statements designed to perform special tasks. It allows us to write down codes inside the function and then we can use the code every time we need it.

These are the absolute basics of C++ in general. C++ is a difficult language, even for those who has a lot of experience in other languages. However, once you get more familiar with its advance capabilities, you'll unlock the potential of both the language and the game engine itself in your way to become an Unreal Engine game developer.

4 UNREAL ENGINE BLUEPRINT

Along with C++, Blueprint Visual Scripting system in Unreal Engine is a complete gameplay scripting system based on the concept of using a node-based interface to create gameplay elements from within Unreal Editor (Unreal Engine Documentation).

4.1 Blueprint Visual Scripting

As with many common scripting languages, Blueprint is used to define object-oriented classes or objects in the engine. As you use UE4, you will often find that objects defined using Blueprint are colloquially referred to as just "Blueprints". This system is extremely flexible and powerful as it provides the ability for designers to use virtually full range of concepts and tools generally only available to programmers. In addition, Blueprint-specific markup available in Unreal Engine's C++ implementation enables programmers to create baseline systems that can be extended by designers. The biggest advantage of Blueprints is that they allow very quick prototyping, highly accessible and self-explanatory.

4.2 Blueprint vs C++

According to Alex Forsythe (2021), between C++ and Blueprints in Unreal Engine, there are differences that make one better than the other. With the information provided on Unreal Engine 4 Documentation, we can point out the

advantage on both C++ and Blueprint Visual Scripting. On the advantage of Blueprints, they are better when handling assets and visual effects. When editing a Blueprint, we can have speculation of all the assets in our project and can use them in a Blueprint to see exactly how they make things look and sound immediately. When Blueprint references an asset, it creates an asset-to-asset dependency, which the engine can control naturally instead of going back from the start to manually update the source and recompile whenever an asset changes. Blueprints take over when it comes to scripted sequences. We can take full advantage of events and functions to write a synchronous code that is straightforward and intuitive when working in an Event Graph, which is limited and harder using C++. Blueprints also allow us to test and iterate quickly due to its fast iteration speed. It is faster to modify Blueprint logic and preview inside the editor than it is to recompile the game. Blueprints are also discoverable, everything is integrated. Types and function we use are all visible for us on Blueprint Editor, quick iteration and visual feedback also makes it easier for testing in general. Whether you have experience with C++, things we learn when using Blueprints can be directly transfer to C++.

On C++ advantage, it can give you maximal runtime performance. C++ codes can be fully optimized at compile-time for the platform it is going to run on. It also has explicit design, when exposing variables or functions from C++, we have more control overexposing precisely what we want, so we can protect specific functions/variables and build a formal "API" for our class. This allows you to avoid creating overly large and hard to follow Blueprints. Also, with broaden access, functions and variables defined in C++, it can be accessed from all other systems, for that passing information between different systems become easier. External libraries are one of the most powerful advantages of C++. If there is a C or a C++ library that we want to incorporate into our game, we can build it as a static or shared library for the supported platform, update module build rules to include the library and use the code we want in our project or plugin. Finally, C++ is easier to diff and merge, the data and code is stored as text, which makes working on multiple branches simultaneously easier.

Unreal Engine give us multiple options for programming games. With C++, we will write codes using general-purpose, text-based programming language,

and with Blueprint we write code by stringing graph nodes that represent events, control structures, function calls, define data and interfaces through in-editor dialogs. Unreal is designed in a way that C++ and Blueprints are very complementary. So, where does it make sense to use C++ and where does it make sense to use Blueprints? C++ is a programming language, while Blueprint is a scripting system. C++ is naturally better-suited for implementing low-level game system, and blueprint is better-suited for defining high-level behaviour and interaction. Typically, C++ and Blueprint will be used along those lines, but there's a fundamental about Unreal Engine that we should understand is that it does not assign that a certain class or problem in the domain of programming or scripting and require us to use specifically C++ or Blueprint to solve them. Unreal is explicitly designed to offer the flexibility:

- There is no separate "Scripting API". Whether we are using C++ or Blueprint, we are making use of the same engine systems the same way.
- C++ and Blueprint are integrated to allow for easy interoperability.
- Unreal extends C++ with UnrealHeaderTool code generation tied into UObject system, so it is easier to move from Blueprint to implementing higher-level functionality in C++.

With the information provide above, we can have a clearer vision on how C++ and Blueprint fits together and by benefiting both of C++ and Blueprints, take time with them and get an understanding of where their respective strengths lie, we can make our way of making games a lot more interesting.

5 UNREAL ENGINE

This chapter is about Unreal Engine interface and getting to know the engine better before we start with our project.

In Figure 4. Unreal Engine categories, Unreal Project categories provide you with many options since it is a multiple purpose engine that can create games, movies, engineering models and constructions.

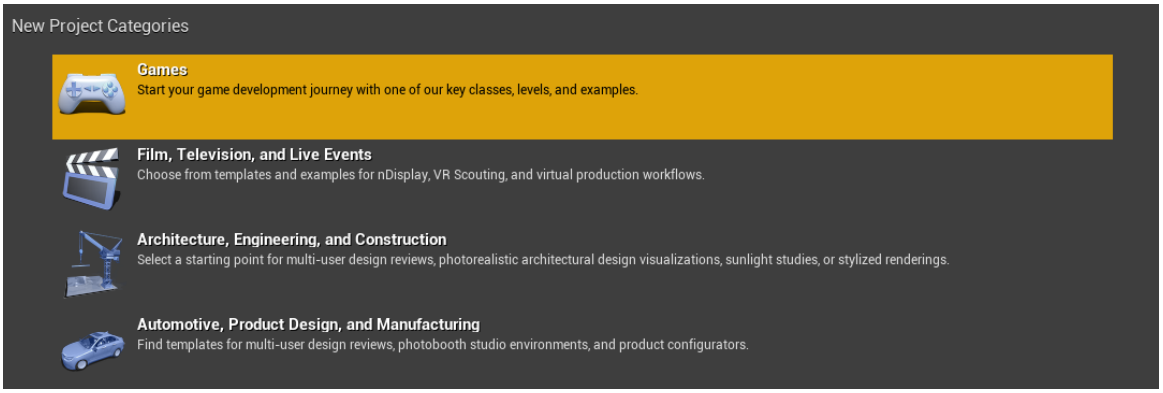

Figure 4. Unreal Engine categories

For creating games, one of the cores of Unreal, the Unreal Engine Project Categories menu also comes different templates for you to freely use to make your game in your own vision (Figure 5. Unreal Engine game templates). You can also have the option to create your game by using Unreal Engine Blueprint or with C++ from scratch.

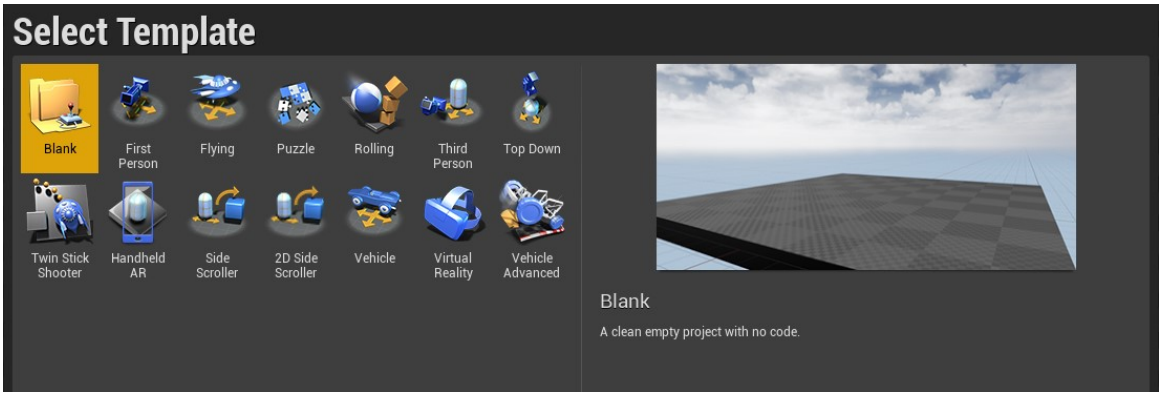

Figure 5. Unreal Engine game templates

After finishing creating the project, we will have a look around Unreal Engine interface (Figure 6. Unreal Engine interface).

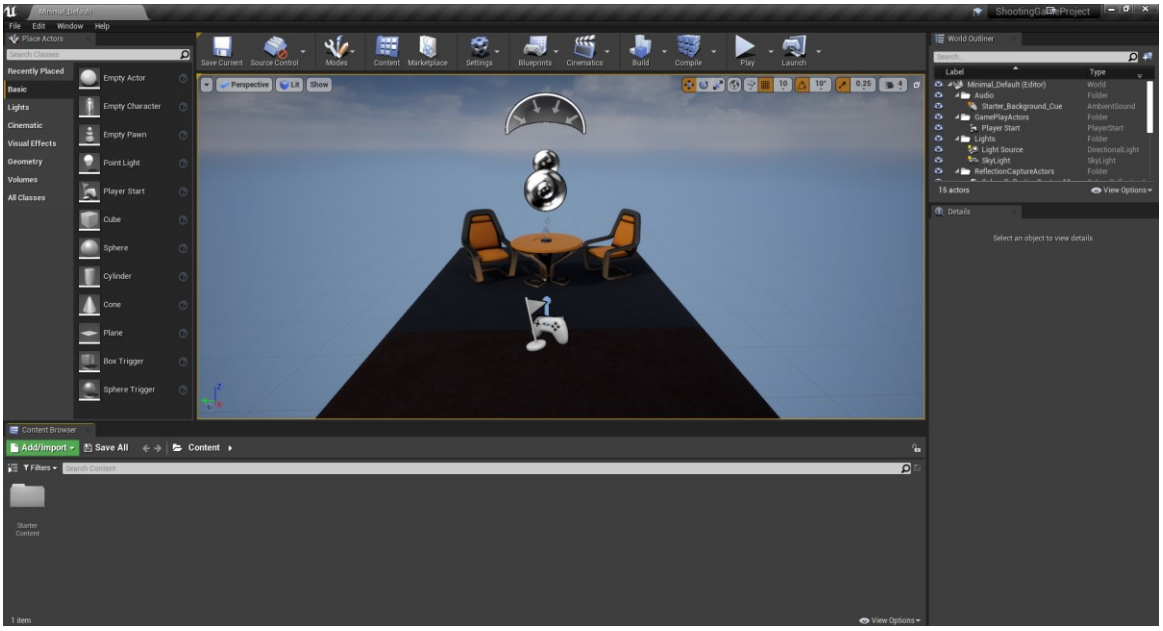

Figure 6. Unreal Engine interface

The main screen in front of us with the in-game terminal is our main viewport. We will mostly be working in it when we create levels, modify, or test the game. On the viewport's top right corner, there are buttons to manipulate Actors in our levels to move, rotate and scale them. Each tool can be set to snap at intervals and set to local space or world space. The button on the top left is where we use to change the way the Viewport renders. We can have the viewport show or hide types of Actors, change which render buffer is shown, enable wire frame, disable lighting and more.

On top of the main viewport is the Toolbar that provides quick access to commonly used tools and operations, includes the Modes Panel (Figure 7. Modes button in Toolbar) and contains a selection of various tool modes for the editor.

Figure 7. Modes button in Toolbar

These change the primary behaviour of the Level Editor for specialized tasks, such as placing new items into the world, creating geometry and volumes, painting on meshes, generating foliage, and sculpting landscapes.

Under the main viewport is the Content Browser, the primary area for creating, importing, organizing, viewing, and modifying content assets within Unreal Editor. It also provides the ability to manage content folders and other useful operations on asset such as renaming, moving, copying, and viewing references. It can also search for and interact with all assets in the game project.

On the right side of the viewport there are the Detail Panel and the World Outliner. The Detail Panel contains information, utilities, and functions specific to the current selection. It contains transform edit boxes for moving, rotating, and scaling Actors, displays all of the editable properties for the selected Actors, provides quick access to additional editing functionality depending on the types of Actors selected in the viewport.

For the World Outliner in Figure 8. World Outliner window, it displays all of the Actors within the scene in a hierarchical tree view. Actors can be selected and modified directly from here.

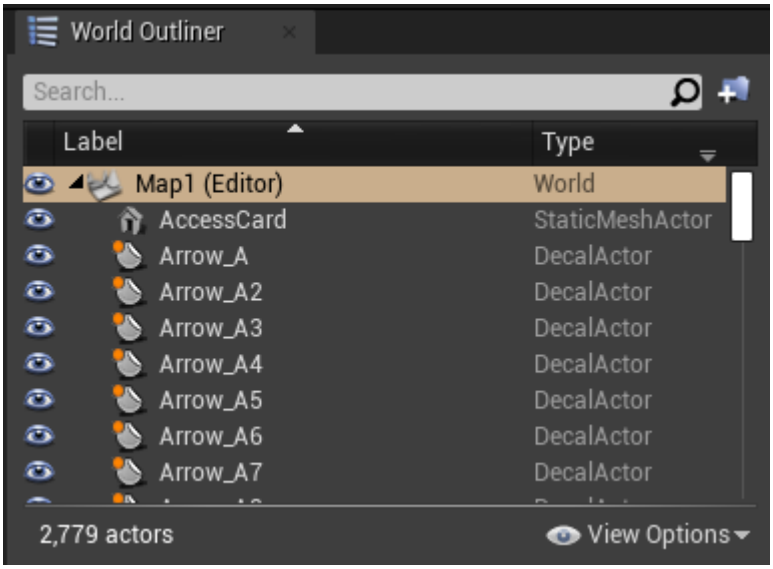

Figure 8. World Outliner window

On the left side we have the Mode's window loads with Actors default object that we can put on our scene, which is useful for setting up placeholder assets or tweaking things for example like the lights.

Finally, there is the Menu Bar. It is similar to the menu bar found in many computer applications, and it provides access to general tools and commands that are used when working with levels in the editor.

Naturally, the Unreal Engine interface is also flexible and customizable to do whatever you want. You can change the place of the tabs to wherever you want, collapse it, or make these windows float depending on the type of workflow you like.

Continue to Actors and Component. An Actor is considering a container that can have many components attach to it. There are also many different types of components that we can add to our actors like particles physics to lighting effects. We can even create our own types of components as well, depending on what we want to use them for. The use of it is to make the Actor in our scene more alive and give them presence within our level.

Inheritance is for “is a” relationship. For example, a character in Unreal is a pawn and a pawn is an actor. In this Inheritance, all the features that an actor has, a pawn will have by default and anything that a pawn has by default, a character will also inherit. Unreal makes extensive use of inheritance, it’s a powerful tool if used properly, it can be inflexible and hard to refactor.

6 GAME PROJECT

The project we are going to make throughout this thesis chapter is one of the most iconic game genres that make Unreal Engine stand out - third person shooting game. We will play as a cyborg created in a laboratory, own a highly evolve AI and learn that this place is weaponizing robots to create war. To stop this plan, the cyborg must fight, destroy all of the other robots and escape.

We want to have the player experience the core of shooting game so our game should have:

- The thrill of shooting battle with hype music theme
- Game mechanic 3D third person shooting

To create the game, we need to have a specific project plan for us to follow and know what we need to do to make the game properly. Our project plan should include:

- Create playing field
- Character and movement
- Animation
- Shooting function
- Health, Alive and Death system
- Enemy AI
- Win/lose conditions

First of we need to create our playing field. Then we will make our player and implement in movements for our character be able to move around the playing field that we create, then taking a bit deeper into animation to make our third person character looks more alive. Then we are going to create some shooting architecture and gun architecture discussions to allow us to do the shooting. Next, we are going to make a health and death system that apply for both our character and the AI. We are going to implement an enemy AI using behaviour trees and make our own behaviour tree nodes in C++. Finally, we are going to create a win/lose condition using the game mode and access to the level to detect whether all the enemies are dead for the win condition and whether we are dead for the lose condition. That is a rough outline of the project plan that we just create to follow, so let's dive in and get our project set up.

6.1 Environment

Before creating any element into our game, first we need to have a game environment - a playing field for our player/character to freely move around. Unreal Engine has UE Marketplace, an asset store where it provides both paid and free asset to download and use in our projects such as environments, characters, materials, plugins, megascans, blueprints, etc. It can be a very useful tools for individual developers to create games on their own, even for a team because of the diversity of content on the marketplace.

When you download an asset from Marketplace, it will directly navigate you to the Epic Game interface, where you have an option to add the asset into your project (Figure 9. Asset from UE Marketplace).

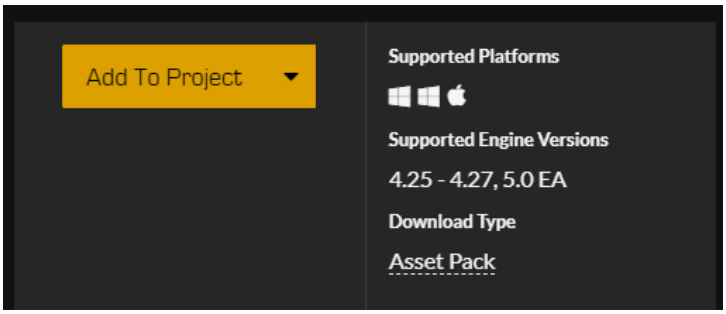

Figure 9. Asset from UE Marketplace

Another example, when have other assets that existed in another project and you want to transfer it into our own, we can just run the project, right click on the asset pack file > Migrate. We can have Unreal Engine selects all of the folders underneath the root that you are migrating, then we navigate the content folders and import them into the project that we want (Figure 10. Migrate progress).

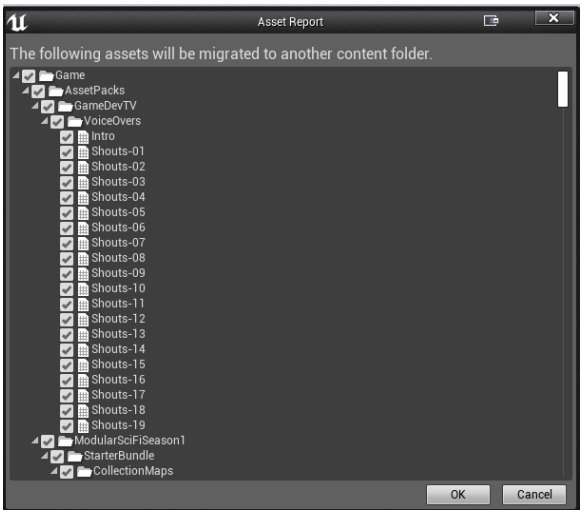

Figure 10. Migrate progress

In Figure 11. Map asset type information, when the assets are inside our project and ready to use, we can just press on an asset which has a Primary Asset Type as **Map**, as the example in the figure below, to load it into our scene.

Figure 11. Map asset type information

We are able to have the selected map as default by go over Settings>Projects>Maps & Modes and change the Editor Startup Map and Game Default Map to the one that we wanted.

6.2 Movement

Finishing up with the environment, the next thing we want to tackle is the player's movement, and to do this we will need to create a character class. Let's talk more about character and pawn. There is a distinction between them, the Pawn is anything that we can possess as a player, and the Character is a pawn, it directly inherits from the pawn class so it can do everything that the pawn can do. Overall, a character is a specialization of a pawn. It adds character-like features like movement, and Nav mesh movement, able the character to move around, avoid obstacles…

To see the differences between the two, we can have a look at the blueprint's children to see the components. In the content browser, right-click to create blueprint class and choose the class to inherit by search it through the search bar inside all classes section. In our case will be the pawn class and the character class we created to compare (Figure 12. Creating blueprint class).

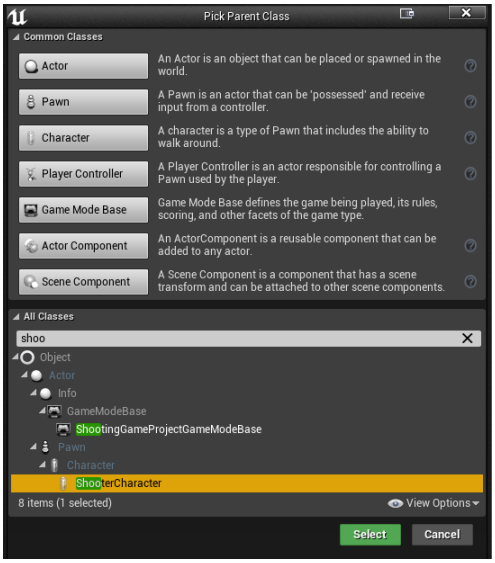

Figure 12. Creating blueprint class

So inside of the pawn class' blueprint, the component view will be empty. It has a default scene route and nothing else. If we go to the character blueprint, however, we got a capsule component, arrow component, mesh and character movement that already inherited inside of the character classes. For the capsule component, it is for doing rough physics and manipulation or collision with the terrain, along with the humanoid character that expecting to fit inside of the capsule. The arrow component which is basically tells us which way is forward for the character. Mesh is where we can put the mesh we want to use inside for the Skeletal mesh, in our project would be the robot shooter character. And the last thing that we need to look through is the character movement component. It is entirely responsible for giving the character-like movement, to being able to walk, run. We can see fully in the details panel with all of the movement setting (Figure 13. Blueprint class detail panel).

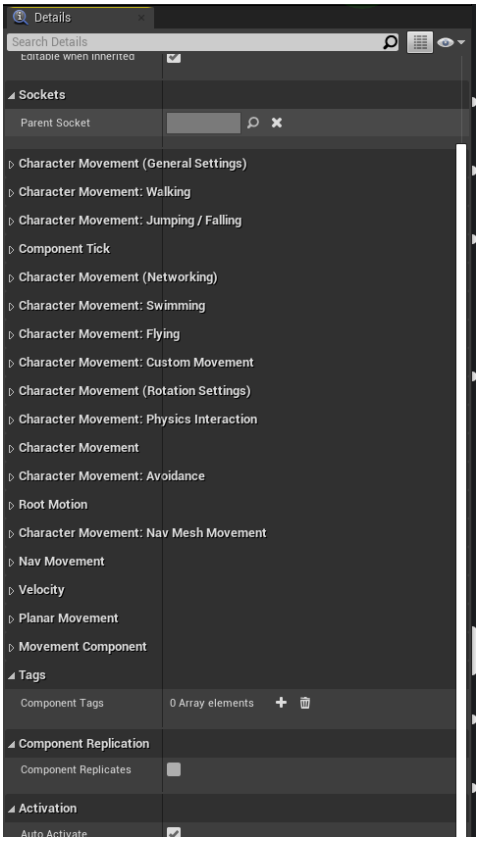

Figure 13. Blueprint class detail panel

When both the playing field and the character has been roughly created, we will setup the Game Mode Base Blueprint, which will define how the game be played, its rules, scoring, and other facets of the game type. So, by creating and add-in our field game mode blueprint, and character to be the default pawn class of the blueprint, we will be able to make sure the character is spawned in our map (Figure 14).

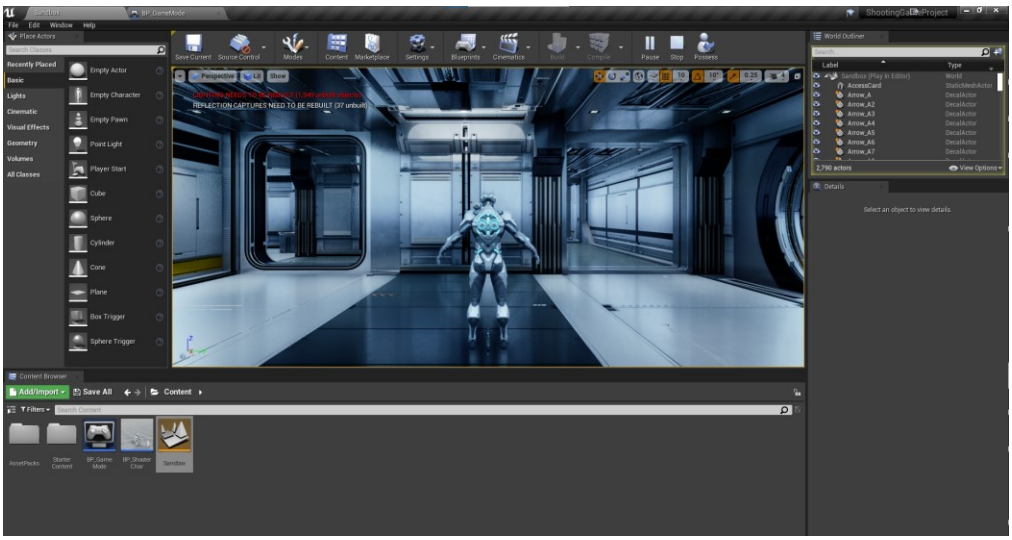

Figure 14. Unreal Engine overview.

Continuing with creating our character movement to move and look around the world that fits the movement of a first person or third person shooter game. Unreal Engine is an engine that optimized for making things like shooter games so when it comes to characters, there is a lot of simplifying functions for us to use. For the movement functions, we need to have a AddMovementInput(), which basically takes a vector and tells us which direction we are going to move in. For example if it is a forward vector, we move forward, if it is a backward vector, we move backward and so on. The next thing is the ControllerPitchInput() for looking up and down by moving our mouse. Then there is the ControllerYawInput() to look left and right and finally the Jump() function for the character to jump when we hit a key. With this list of function, what we need is to create bindings in our character actor class that bind inputs on our keyboard, mouse, or controller for these functions.

When you open up Visual Studio Code, inside ShooterCharacter.h file, we will see that we have already got the setup player input component function (Figure 15).

```
// Called to bind functionality to input
    virtual void SetupPlayerInputComponent(class UInputComponent*
PlayerInputComponent) override;
```

Figure 15. SetupPlayerInputComponent syntax

This is where we need to bind our input into. By pressing Alt+O, we will be redirect to the cpp file and find where we have got that set up, and the first thing is it gives us the input component that we want to bind with. Before we go further into the code, we need to take a step back and setup the axis in our project setting. Inside Project setting>Engine>Input, we can setup action mapping and axis mapping. The first mappings we are going to have are move forward and look up. We do not have a move backward and a look down because we can just do that with negative scale (Figure 16).

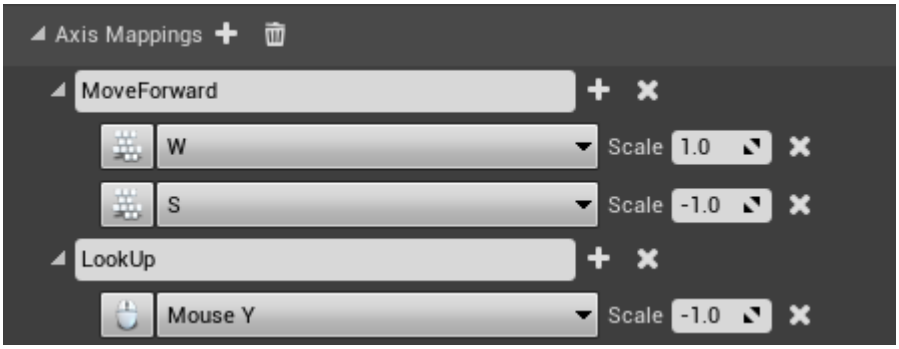

Figure 16. Project setting input

Once we have got the axis name, we can bind it using text macro and the text of the axis name, then we need to say who we are going to call the function on and what function we are going to call. We are going to create a new private function in the header file for the move forward and with the help of an extender in Visual Studio Code called C++ Helper, we can create an implementation for the cpp file by key combination Ctrl+Shift+P and type “Create Implementation”. With the function created, we can bind the PlayerInputComponent to the function, and by passing in the access value to the function, we will be able to get the forward vector of the actor and multiply by the axis value to give us the right result. Then we will have the forward vector multiplied by the axis value if this one will go forward. If it is minus one, a forward vector becomes a backward vector and we are going backward with this one function.

For the Look Up function, if we repeat the same function as we did before with Move Forward, the function will basically do nothing because it is just calling some other function with the same parameter. We can just call the AddControllerPitchInput by calling it directly from class APawn. To create a function to move left and right we will need to do things differently, not by just calling straight add movement input. We need to get the ActorRightVector instead of getting the ActorForwardVector. For the LookRight input we will use AddControllerYawInput instead of AddControllerPitchInput. For Jump action, we will use Action Mapping, not Axis Mapping. For the function, we use BindAction instead of BindAxis, and we need to use the syntax EInputEvent, which is an enum basically can take a few different options. We will use it to make the character jump on press and input the jump function that defined inside ACharacter class (Figure 17).

```
// Called to bind functionality to input
void AShooterCharacter::SetupPlayerInputComponent(UInputComponent*
PlayerInputComponent)
{
    //input player movement
    Super::SetupPlayerInputComponent(PlayerInputComponent);

    PlayerInputComponent->BindAxis(TEXT("MoveForward"), this,
&AShooterCharacter::MoveForward);
    PlayerInputComponent->BindAxis(TEXT("LookUp"), this,
&APawn::AddControllerPitchInput);
    PlayerInputComponent->BindAxis(TEXT("MoveRight"), this,
&AShooterCharacter::MoveRight);
    PlayerInputComponent->BindAxis(TEXT("LookRight"), this,
&APawn::AddControllerYawInput);
    PlayerInputComponent->BindAction(TEXT("Jump"),
EInputEvent::IE_Pressed, this, &ACharacter::Jump);

}

//forward and backward movement
void AShooterCharacter::MoveForward(float AxisValue)
{
    AddMovementInput(GetActorForwardVector() * AxisValue);
}

//left and right movement
void AShooterCharacter::MoveRight(float AxisValue)
{
    AddMovementInput(GetActorRightVector() * AxisValue);
}
```

Figure 17. Movement controle code

With the finished function, the character now can be able to move freely in the play area.

6.3 Camera

To setup a third person camera, we need to add a camera inside the blueprint configuration. By opening the blueprint editor, go to the component section and add in a Camera Component. With the camera on its own, the control rotation will not work entirely and if we back into a wall, the camera will go right through. By adding in the Spring Arm Component and make the Camera as the child component of it, we can keep the character within view and can remain in the playing field without go right through the wall to make the third person camera at its finest.

6.4 Animation

For character animation, we need to look through how Skeletal Mesh Animations work. The mesh we are using has a skeleton and we want to manipulate it though bones. Skeleton exists within the modelling program that was creating the mesh, but a skeleton in Unreal has a slightly different function is to tie together the meshes and animations. We can have multiple meshes which all use the same skeleton asset in Unreal and set multiple animations that use the same skeleton. With it we can share those animation assets between different meshes. With AnimGraph of Animation Blueprint, we can blend animations together using a series of variables, by changing the variables we can also change the blend of animation. To create an Animation Blueprint, press Add/Import>Animation> Animation Blueprint. Like the skeletal meshes, it relies on a skeleton, which means it can use all the animations of that skeleton and can be used on any of the meshes that related to that skeleton. The AnimGraph is like a blueprint, except it has no execution node, all it has is the data that flows through the graph. We want to get to the output pose – the output animation pose that we modeled our character into. By connecting different nodes to the result node and flowing through, we end up getting our ouput data. To add in a Blend node, right click on the AnimGraph, it will show a tab with all action for the blueprint, and we can search for the action that we want. Inside of the node, we have a value, two input poses and an output pose (Figure 18).

Figure 18. Blend Node

With this node, we can change between animations by changing the value and the data will flow to the output. But it is not ideal that we need to keep recompiling to change the value. By adding in a float variable, the game is

going to drive these variables in the animation blueprint to get the correct animation at any point during the play. With locomotion animation, which uses data from animation to move the character, blend nodes are not the optimal way. 2D Blend Space allow us to smoothly change different types of movement when we want to blend between things in a two-dimensional space. With Blend Space we will be able to move between these blend spaces and able to choose an animation and blend between it (Figure 19).

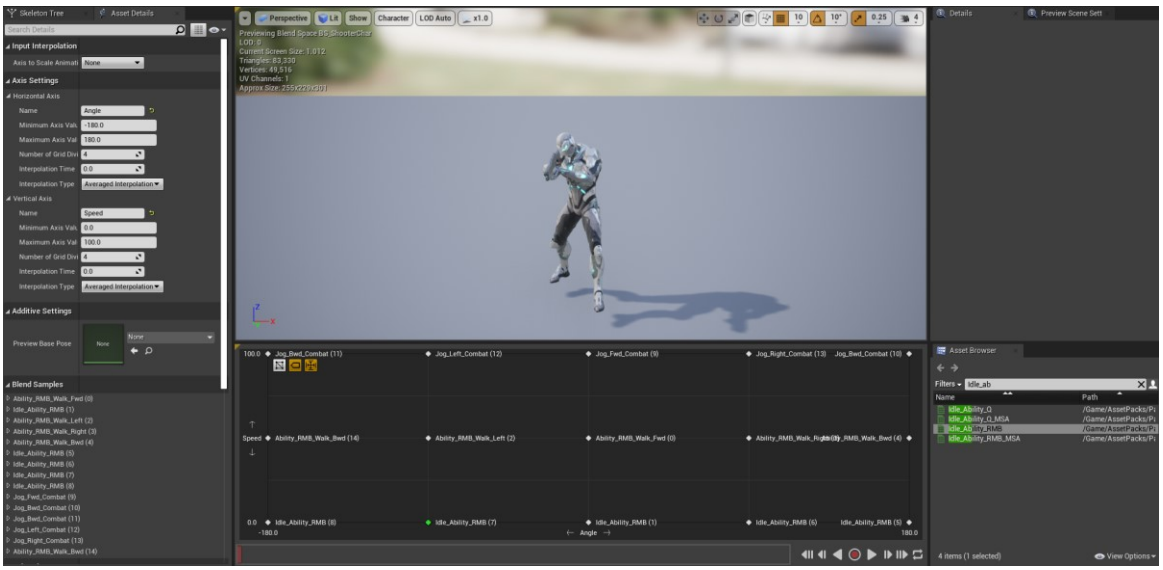

Figure 19. Blend Space overview

To connect our animation to our gameplay, we will add-in our locomotion into the AnimGraph and create two new variables for Angle and Speed. When the variables change, the blend space will redirect the pointer location so it will know what animation to blend between.

Inside this animation blueprint has an event graph that contains some event nodes, and for the Event Blueprint Update Animation, we want to update and take the information from the pawn about its movement. We will calculate and setup the speed. For example, we want to get the velocity of our pawn owner and the length of the vector. To set the speed using the vector length, we will have the speed float variable set in, the execution pin we will get from the blueprint update animation and the data pin, which is going to be the speed comes from the vector length of getting the velocity (Figure 20).

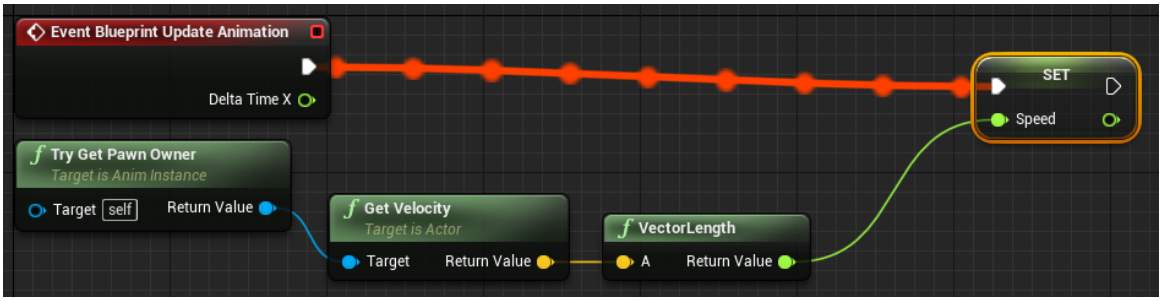

Figure 20. ShooterCharacter Event Graph A

Before we want to be able to work with the angle, we need to understand more about transforms and the definition of local and global terms. When you look at any of the components in the hierarchy, you can see a transform tab that includes location, rotation, and scale properties. That is basically what transform is, to change the properties and make the components transform. But there is a part that we are not seeing here, which is also a part of this transform, is that these same components are in a hierarchy, meaning that the transform also has a parent (Figure 21).

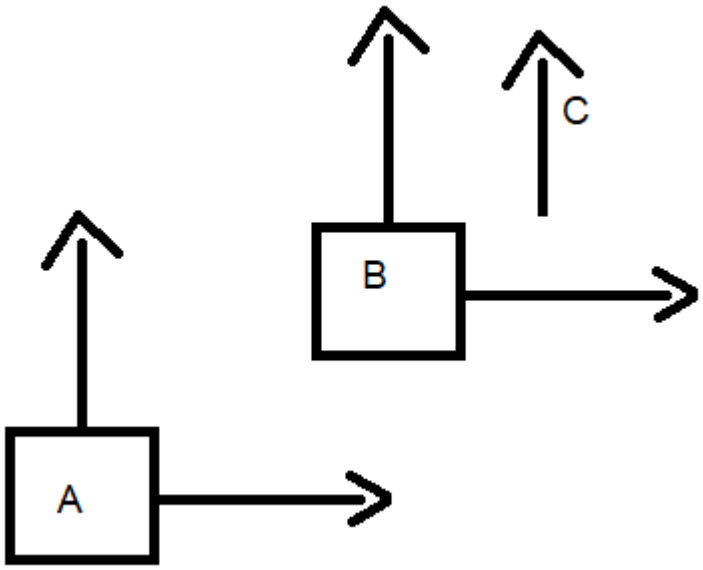

Figure 21. Example on Local and Global transformation

Base on the example we have above, A is the base, the world space. B is a component that got a position in the world, relative to A. And inside component B has C so C has a position relative to B. The way this hierarchy works is the parents of C is B and the parent of B is A, the base of the world

space. We also have coordinate system within each of them. For A, the world coordinate system is the global system, and the coordinate system for component B is local to its position and C is positioned relative to B. It means that if B has a change in its position, C moves along with it because the position of C relative to B has remained the same. Because B has moved, its position in the world has moved, it also applies for rotation as well.

For a different aspect, we have global space and local space. Global space has the base at the center of the world and the component moves. For local space, on the other hand, sees things opposite. The component is the center and the base moves. It all depends on our point of view. For the angle in our project, we are interested in a local angle relative to where the player is looking and transforming velocity use direction vector transform rather than position vector transforms because velocity does not tell us where we should be, it tells us the direction we should be going and how fast we should be going in that direction. We know the velocity is in the global space and we want to convert it to local space, we will use what is called an inverse transform direction (Figure 22).

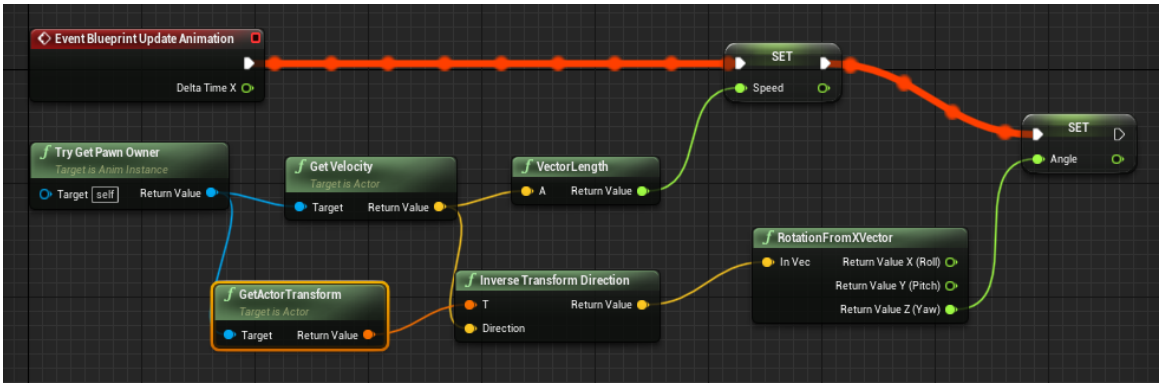

Figure 22. ShooterCharacter Event Graph B

We will get the transform of the pawn itself of the character, and we need to get the rotation of the vector by using rotation from X vector because it will set the yaw and the pitch vector, we are going to split struct pin to be able to get the yaw return value and set up the Angle variable. Based on the calculated and setting, our character now can move around forward, backward, left and right with different angle with animations smoothly. For aiming animation, we will have a different kind of animation which is known as **Additive animation**. The difference is it can be added on top of the other animations, for example,

it can add the information about where the player is aiming on top of the running animation.

6.5 Shooting function

The easiest way to create functions for guns in shooting games is to architect it as a separate thing to the character so we can be able to do various functions around it, such as switch to different type of weapons or have different characters with different weapons, so the gun itself can be responsible for how it shoots. First, we create a C++ gun actor because we do not need any behavior to possess it, and we create a blueprint subclass and add components, it means that we are going to have access to those components from within the C++ code to manipulate and write our code there. We want to setup root component and have a mesh component to be the child for root, because we want to be able to change the position of the mesh relative to the root of the gun for placing the handle in different location for different measures. We use UPROPERTY(VisibleAnywhere) for the components to be visible anywhere because they are pointers, and we do not want to be able to edit the pointers, but we do want them to be able to see the properties of those pointers and be able to edit those.

For spawning the gun as a child of our current character, we will need to spawn it at runtime from our character class in C++. We add in UPROPERTY(EditDefaultsOnly) to allow us to configure the class that we should be pointing to in order to spawn an actor, which blueprint class should be spawning from the shooter character. We use EditDefaultsOnly because we do not want to be able to edit this at runtime because it would give a false impression that we can change what type of gun we are using at runtime, where BeginPlay has already been called, that gun is already been spawned and it would not update anything. We use TSubclassOf<AGun> for this property, allow blueprints to restrict the classes that we can select from, to only the classes that are subclasses of the gun C++ class. This is not a pointer because we are not pointing to an instance of the gun, we are getting a blueprint class in this variable. We are going to use another UPROPERTY to store the gun itself. In ShooterCharacter blueprint, we setup the Gun blueprint in the gun class and compile.

Before we want to attach the gun actor to the character's hand, we have to hide the actual gun mesh that is currently there, hide the bone that has the gun mesh in C++ script. First, we need to get hold of the mesh components by calling GetMesh when we are on the character subclass. With it, we want to hide the bone by using HideBoneByName provided in by this line of code (Figure 23).

```
HideBoneByName( FName BoneName, EPhysBodyOp PhysBodyOption );
```

Figure 23. HideBoneByName syntax

BoneName is the name of bone to hide and **PhysBodyOption** is the option for physics bodies that attach to the bones to be hidden. And with this will be the function we use in our script (Figure 24).

```
GetMesh()->HideBoneByName(TEXT("weapon_r"), EPhysBodyOp::PBO_None);
```

Figure 24. function application

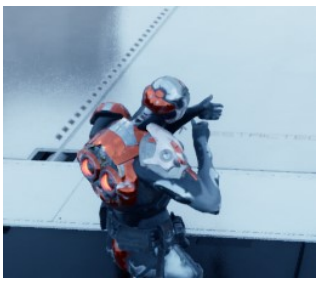

Figure 25. HideBoneByName applied

After the mesh is hidden (Figure 25), we need to add in a socket in place to put a different gun to the character's hand. To add a socket, we will go to the Skeleton-> right click on the bone that we want to attach the socket to-> Add socket. And we will use AttachToComponent to do the attachment of the component. This function will attach the RootComponent of this Actor to the supplied component, optionally at a named socket. **Parent** will be the parent to attach to, **AttachmentRules** defines how to handle transforms and welding when attaching the component, **SocketName** is the optional socket to attach to on the parent (Figure 26).

```
AttachToComponent(USceneComponent* Parent, const
FAttachmentTransformRules& AttachmentRules, FName SocketName =
NAME_None);
```

Figure 26. AttachToComponent syntax

Our function (Figure 27):

```
Gun->AttachToComponent(GetMesh(),
FAttachmentTransformRules::KeepRelativeTransform, TEXT("WeaponSocket"));
```

Figure 27. Function applied

We also need to setup ownership for the gun to allows it to know who is the owning character and it can use to get references to the owning character.

For a gun to function correctly, we need to create the shooting architecture so that whenever we press the mouse button, functions in the architecture will run. The reason to choose the gun as the core of the shooting architecture is that it can be able to support different types of shooting methods, it could be change depending on the implementation of the gun itself. We want to create a public function to the Gun header file; it is going to allow us to essentially active the gun from the shooter character from wherever we are binding the input and put in new input binding, call the shooting in the function on the character to create a shoot action and add up some particle effects when the shooting triggers with SpawnEmitter function.

To setup third person shooting, we need to set viewpoint to use for line tracing. We need to have ray cast from the camera so wherever the camera is looking at is going to be the thing that it hits so that the camera will point in exactly the right direction from where we are looking. First, we will get hold of the players viewport, each time we trigger the fire button, we will draw a debug camera to show the direction that the camera is looking (Figure 28).

Figure 28. Debug camera

Setting up Line Tracing, we will be using **LineTraceSingleByChannel()** function. It will trace a ray against the world using a specific channel and return the first blocking hit (Figure 29).

```
LineTraceSingleByChannel(struct FHitResult& OutHit,const FVector&
Start,const FVector& End,ECollisionChannel TraceChannel,const
FCollisionQueryParams& Params =
FCollisionQueryParams::DefaultQueryParam, const
FCollisionResponseParams& ResponseParam =
FCollisionResponseParams::DefaultResponseParam) const;
```

Figure 29. LineTraceSingleByChannel syntax

OutHit will define the first blocking hit found, **Start** and **End** is the starting and ending location of the ray, **TraceChannel** is the channel that the ray is in, used to determine which components to hit. **Params** is the additional parameters used for the trace, **ResponseParam** is the ResponseContainer to be used for this trace. If a blocking hit is found return TRUE. Channel function is better because it allows us to define our own custom channels and then define what sorts of objects are transparent to that channel and which one blocks it.

In Project Setting>Engine>Collision, we add a new trace channel to setup the bullet, the default response is block, which means that all object type is going to block unless we specify. We can edit the profile of objects in Preset list, calculate the end point for the line trace and impact effect when hits (Figure 30).

```
//End point calculation
    FVector End = Location + Rotation.Vector() * MaxRange;

    //Line trace



    FHitResult Hit;
    bool bSuccess = GetWorld()->LineTraceSingleByChannel(Hit, Location,
End, ECollisionChannel::ECC_GameTraceChannel1);
    if (bSuccess)
    {
        FVector ShotDirection = -Rotation.Vector();
        UGameplayStatics::SpawnEmitterAtLocation(GetWorld(),
ImpactEffect , Hit.Location, ShotDirection.Rotation());

    }
```

Figure 30. Creating shooting function

With the shooting function finished, we will move on and create the alive/death system

6.6 Health, Alive and Death system

Doing damage is one of the important elements in a shooting game. That is how we define how the health system works and knowing how much health a character has. Doing damage is a two-part process, sending the damage and receiving damage. The function we are going to use for this process is TakeDamage(). **DamageAmount** defines how much damage to apply, **DamageEvent** is the data package that fully describes the damage received, **EventInstigator** is the controller responsible for the damage, **DamageCauser** is the Actor that directly caused the damage, and it returns the amount of damage actually applied (Figure 31).

```
float TakeDamage(float DamageAmount, struct FDamageEvent const&
DamageEvent, class AController* EventInstigator, AActor* DamageCauser);
```

Figure 31. TakeDamage syntax

There are two main subtypes for the damage event define the main types of damage in Unreal Engine is **FPointDamageEvent** and **FRadialDamageParams**. Radial Damage are the parameters used to compute radial damage, Point Damage is a damage subclass that handles damage with a single impact location and source direction. In Figure 32, this is the FPointDamage Event construction:

```
FPointDamageEvent(float InDamage, struct FHitResult const& InHitInfo,
FVector const& InShotDirection, TSubclassOf<class UDamageType>
InDamageTypeClass)
```

Figure 32. FPointDamageEvent syntax

Apply into our code for the character to do damage (Figure 33):

```
AActor* HitActor = Hit.GetActor();
        if (HitActor != nullptr)
        {
            FPointDamageEvent DamageEvent(Damage, Hit, ShotDirection,
nullptr);

            HitActor->TakeDamage(Damage, DamageEvent, OwnerController,
this);
        }
```

Figure 33. Dealing damage applied

To deal damage to the actor, we are going to overriding TakeDamage() on the actor. We will get hold of the information about how much health the character have and create an implementation for taking the damage (Figure 34).

```
float AShooterCharacter::TakeDamage(float DamageAmount, struct
FDamageEvent const& DamageEvent, class AController* EventInstigator,
AActor* DamageCauser)
{
    float DamageApplied = Super::TakeDamage(DamageAmount, DamageEvent,
EventInstigator, DamageCauser);
    DamageApplied = FMath::Min(Health, DamageApplied);
    Health -= DamageApplied;

    return DamageApplied;

}
```

Figure 34. Taking damage applied

When a character runs out of health, we want to have a death animation whenever the health goes to 0. By using Blend Poses by bool, we can hook up the movement animation and a death animation to the nodes, if the character is alive, we can have the movement animation active. When the health reduces to 0, the pose will be come true, the character is dead, and it will automatically turn to true and active the death animation. We will create a public **UFunction(BlueprintPure)** with a Boolean as a const member function. We want it to be const because when we call the function, we do not want it to change any state of the ShooterCharacter, and BlueprintPure will make the node in the EventGraph as a pure node. Pure node does not have an execution pin and it always output the same result, it does not change anything in the ShooterCharacter, global or anywhere, the only effect it has is in the outputs that it produces.

6.7 AI system

In order to make the shooter game more alive, we will move on to create an AI for our game. For every single player game, AI have a huge role to make the gameplay shines. It can interact with the players in many ways depends on

how we program it. In our shooter game we will program our AI to have movements and can aim and shoot when it interacts with the player. We will setup the AI aiming by using built-in functions in Unreal Engine are **SetFocus()** and **ClearFocus().** Using the functions, SetFocus() can tell the AI that it should be trying to focus either on a particular actor, which we will use on our player, and if the focusing actor moves, the AI will keep on focusing on the actor. ClearFocus() will do the opposite, which tells the AI that it should stop focusing on the actor that we targeted. We also have **InPriority** variable that will handle the focus actor for the given priority. We can see the list of the priority by going to the definition of the enum (Figure 35).

```
virtual void SetFocus(AActor* NewFocus, EAIFocusPriority::Type
InPriority = EAIFocusPriority::Gameplay);

virtual void ClearFocus(EAIFocusPriority::Type InPriority);
```

Figure 35. SetFocus and ClearFocus syntax

We will also need to setup the movement for the AI so it can navigate around, avoid obstacles, move in the playing field and it also need to have pathfinding function where it can plan its route to come around the obstacles and come towards the player. In Unreal, we will generate a mesh which tells us where is walkable within a level and use algorithms includes pathfinding, so we can navigate on the mesh and create a route plan and the AI will use the route plan to move along. Inside Windows>Place Actors we have **Nav Mesh Bounds Volume**, this creates a volume, everything within it is going to be checked for how navigable it is, whether it is flat or steep. Supporting it inside the AI blueprint, there is a component called PathFollowingComponent. That component is responsible for finding nav mesh and allowing us to create paths to follow and move along them automatically. With the volume available around the map and the AI is able to move around, we will need to setup the AI **LineOfSight()**, so the AI can detect the player within its sight, and setup the movement so when it detects the player, the AI can follow the player around the map (Figure 36).

```
void AShooterAI::Tick(float DeltaSeconds)
{
    Super::Tick(DeltaSeconds);
    APawn* PlayerPawn = UGameplayStatics::GetPlayerPawn(GetWorld(), 0);
    //if LineOfSight
    if(LineOfSightTo(PlayerPawn))
```

```
    {
        //MoveTo
        MoveToActor(PlayerPawn, AcceptanceRadius);
        //SetFocus
        SetFocus(PlayerPawn);
    }
    // Else
    else
    {
        //ClearFocus
        ClearFocus(EAIFocusPriority::Gameplay);
        //StopMovement
        StopMovement();
    }
```

Figure 36. AI detection setting

To make a more complicated AI, we will have to use a more efficient tools to create the AI behavior than using the previous function. Fortunately, Unreal Engine provides us with **Behavior Tree** and **Black Board**, allowing us to set complex behaviors, actions and make our work on creating a complicate AI easier. Behavior Tree allow us to create trees of behavior, a great way of putting together natural behaviors into our game. Black Board is like the memory of the AI, it is where we store variables that we input them from the gameplay. By using the ShooterAI controller, we can reference in the blueprint the behavior tree we want it to run and make the controller entirely responsible for running the behaviors.

The first example will be using the **Sequence** node in the Behavior Tree, it will execute the children of the node from left to right, and it will stop executing when one of their children fails. Instead of making the AI follow the player in the area we will use the Behavior Tree to set up a route for the AI to follow. To setup for the AI to patrols in a specific route, we will set the keys in the Blackboard for the start location and the location we want it to go, mark the tasks as children of the sequence node and modify the execution options. The sequence will go through doing each task until completion or failure. If it fails, the sequence node will fail, and if it completes, the route will start it again (Figure 37).

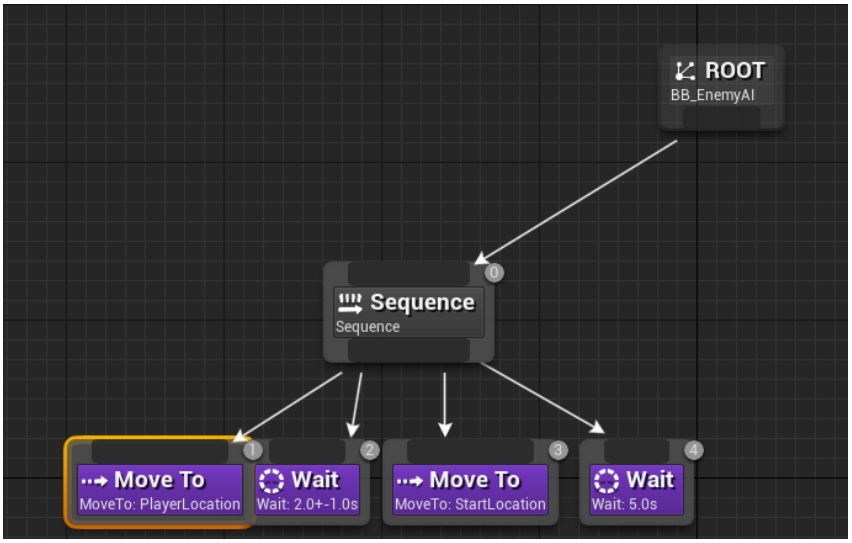

Figure 37. Behavior Tree example

Next is an example using the **Selector** node, it will execute the children of the node from left to right and will stop executing its children when one of their children succeeds. It is different from Sequence because in this scenario we want it to choose the first option that viable. To make it works, we will run Sequence beneath Selector instead of running tasks along with decoration which can modify the behaviour of the sequence. Using Selector node, we can have the AI choose between two behaviour that we setup in the Behaviour Tree. We can also create custom Behavior Tree task that has function based on C++ codes to create a behavior for the AI called **BTTask**. Inside the BTTaskNode header file, it has the information about the class and the virtual functions that we can use to implement.

- **ExecuteTask** is called when the task starts to execute
- **AbortTask** is when it is decided that we need to stop executing
- **TickTask** happens all the time that the task is executing after the first execute task calls.

We will setup the AI to chase when it saw the player, if it loses sight, then the AI will choose to investigate and goes over to the last known location, clear the value with our Clear Blackboard Value node created with C++, wait for 5 seconds, and move back to the starting location. We will also create a C++ node for the AI to shoot when it reaches the player certain radius. The AI will get hold to the pawn which is the ShooterCharacter so we can use the AIController to be able to call the Shoot function.

In Unreal's Behavior Tree, it has **BTService** which can help us with persistent running behaviors. It will execute at their defined frequency as long as their branch is being executed. These are often used to make checks and to update the Blackboard. BTService can also be custom created with C++. Along with Default focus service that set the AI to focus on the player's location, we create two custom service that can update our last know player location while chasing and update the player location if we have line of sight, clear the location if we do not.

Overall, **Behavior Trees** can be linked together to create choice, sequence, and put lots of tasks together to make more complex behaviors easily (Figure 38).

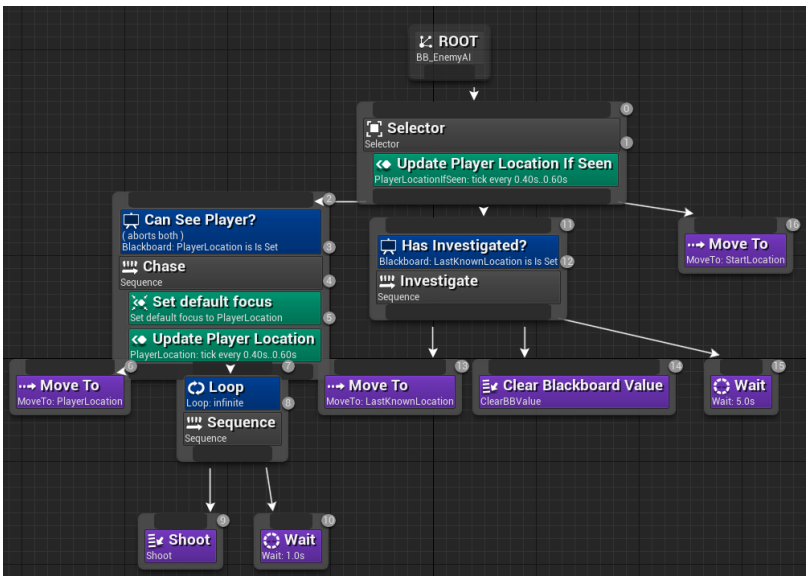

Figure 38. AI Behavior Tree

Finished setting up the Behavior Tree, the AI now will have a set of action to interact with the player in the playing field.

6.8 Ending the game

Finally, we will create the game controller that will receive the information about the end of the game, to make win and lose condition for the game to have more purpose, along with making the UI element to finalize our game project. To have the game ends, we will need to go into the state where they player lost, the game will inform the player and give the option to restart or quit, and the involved classes are: Game Modes - which are responsible for deciding rules about the game.

- ShooterPawn: have a virtual method which will be called to give the information about the player, or the AI has been killed.
- Player Controllers: have virtual methods to call and created to do the UI, restarting.

The ShooterPawn will find the ShootingGameMode, calls the virtual function PawnKilled. We will create a KillAllGameMode to inherits from the ShootingGameMode and it overrides the PawnKilled function to count up and decide differently on the rules of the game in case our game might have different modes for the gameplay. So that depending on the implementation we have in the KillAllGameMode, it will go through to the implementation via the override. And when we decided that the game is over, we will call from the GameMode the function on the root PlayerController and we can implement in our own Controller to do UI or restarting the game. This allows us to have the flexibility in mixing different gameplays and levels.

We need to create a controller that will receive the information that the game has ended, from it we will have a restarting behavior that will reload the level after the player has died. With the built in function **GameHasEnded()** inside Unreal Engine, we can create the function that ends the game when the player health is down to 0. Currently we are calling the GameHasEnded() function on the player controller, but we want it to be the function on all controllers, whether AI or player controller, to indicate win or lose condition when it ends. With **TActorRange** from **EngineUtils.h**, it is going to return us a range object, which is like a list that goes over all of the controllers in the level, and we can use it in a for loop to decide whether or not we should be passing in true or false to the GameHasEnded() function to know that the controller is the player or the AI, and the player is the winner or not. With TActorRange we also can use it to calculate the win condition. When PawnKilled() is called and they pawn is not the player controller but an AI, we could subtract from a variable which has the number of AI in the level, and when the number reduce to 0, the game will end and they player wins (Figure 39).

```
void AKillAllGameMode::PawnKilled(APawn* PawnKilled)
{
    Super::PawnKilled(PawnKilled);

    APlayerController* PlayerController =
Cast<APlayerController>(PawnKilled->GetController());
```

```
    if(PlayerController != nullptr)
    {
        EndGame(false);
    }

    //for loop over ShooterAI in World
    for(AShooterAI* Controller : TActorRange<AShooterAI>(GetWorld()))
    {
        if(!Controller->IsDead())
        {
            return;
        }
    }

    EndGame(true);
}
```

Figure 39. Application of game condition

We also have **RestartLevel()**, with this function we can set up a timer to restart the level after an amount of time. Overall, the game has win and lose condition, the player will lose when the health reduce to 0 and win when the player destroy all the AI in the level. The game will end and restart after an amount of time. With the finished function, we will create the UI to indicates the information for the player to display on the playing screen with **WidgetBlueprint** and display it through C++ in our ShooterPlayerController() script to show the win/lose screen to the viewport when the game ended. WidgetBlueprint also can be used to create player crosshair and can be used to create Health bars to display the player's health considering a compulsory UI for shooter game. The player Health Bar will be indicated by Binding function on the progress bar that every time it returns a different value, it is going to update the UI (Figure 40).

Figure 40. Gameplay UI

Sound effects is one of the core elements to make the game more entertaining. **Ambient Sound** is an actor that can use to play sound everywhere on the playing field, which we can use to play the background music. We can add sound for gun shot by **SpawnSoundAttached()** and **PlaySoundAtLocation()** for bullet impact. Along with Unreal Engine built in asset **Sound Cue**, we can set up random sound to play from a list of different sounds and have a modulator that defines a random volume and pitch modification when a sound starts. **Attenuation Distance** and **Spatialization** are definitions of shooting games, where Attenuation Distance will define volumes for sounds that have different distance from the player, such as which sound that closer will sound louder than a further one. Spatialization will gives us an illusion that a sound is coming from a particular direction. It applies when you have a headphone or two speakers, depending on where these sound sources are relative to where the player is looking, if it is on the left, the audio will play more on the left speaker/headphone.

With the project plan has been finished and all the compulsory functions has been created, we will create a playable level and build our game into a playable software. In Project Setting>Project>Packaging, we will have the options to build the game inside **Build Configuration**. **Development** mode will build our game, but we will still have testing options to configure your game and also can use console command. **Shipping** is what most games are build, it is a fully finished build with no console client. Supported Platforms is the list of platforms you want to build your game in, and Targeted Hardware will be choosing which class of hardware for our game. After finished with the setting, we will build our game by pressing build on the Toolbar and when building is finish, package the project by go to File>Package Project and our game now is a playable software.

The project's Visual Studio Codes Script files are available on Github.

7 UNREAL ENGINE 5

After 8 years since Epic Games shipped Unreal Engine 4, and 7 years since Unreal 4 become free for everybody to use. Recently, Epic announced that Unreal Engine 5 has officially released. Unreal 5 was revealed on May 13, 2020, supporting all existing systems including the next-generation consoles PlayStation 5 and Xbox Series X/S. On May 26,2020, it was released in early

access, and officially launched for developers on April 5,2022 (Wikipedia). With the official release of Unreal Engine 5, game developers and creators will have the chance to experience the next generation of computer graphics, along with many new features that makes Unreal Engine 5 become a game changer.

- **Lumen** is the brand-new lighting system that has been added into Unreal 5. Lumen is a big step for Unreal because it is true real time global illumination. Since the start of computer graphics, developers have been trying to simulate global illumination, essentially it is bounce lighting, which is extra light created from different rays bouncing off different surfaces. With Lumen now has true real time global illumination, developers no need to tie with time-consuming through light baking process.
- **Nanite** is one of Unreal Engine 5's major features. It is an engine that allows for high-detailed photographic source material to be imported into games. Objects in video games are comprised of flat planes of geometry called polygons. The more polygons in the world, the longer it takes for computers to render, which can slow down video games. **Level Of Detail** (LOD) has been used by video games to handle the problem, the further away an object is from the camera, the less polygons it will use to render that object, and it will make the game run faster since there is less polygons to be rendered, but it is a time-consuming process. Nanite allows for dynamic level of detail, meaning that we can have an object with millions of polygons without any LODs because those polygons dynamically deform, lower the total amount on the object, allow assets made for movies that has higher number of polygons to be used in video games.
- **Megascans asset library** stores over 16000 assets for us to populate objects into our world with its great features mention above. These assets include 3D objects, material objects, modular objects, plants and customizable tree has been added recently. All of the objects are photorealistic since they have been scanned from real life using a program called Reality Capture, converted into 3D object and imported into Unreal.

- **MetaHuman** is amazing in creating humans because they are always the hardest assets to create for games, especially realistic humans. Now Metahuman can be easily add-in to Unreal Engine 5 with Quixel Bridge, download the character and add into our game, and we are able to animate everything about the metahumans that we have added in.

With Ureal Engine 5 release, Unreal Engine 4 projects can be easily converted into Unreal 5 projects. Running Unreal Engine 5, we can make use of all the new features such as Lumen and Nanite, and converting projects can be a lot better than the previous builds.

8 CONCLUSION

The purpose of this thesis was to introduce about game developing with Unreal Engine and helped programmers who were interested to take their first step to become an Unreal Engine game developer. Along with the introduction, the thesis provided the information on the background of the history of games, how it evolved through time and introduction to Unreal Engine – one of the most popular game engines in the game industry. It also provided details on Unity Game Engine and had a comparison between Unity and Unreal to have a clear vision on the difference and understand about the possibility that Unreal Engine can do.

The first chapter introduced to the reader the idea of how the game industry had evolved over the pass years and had the basic understanding about engines advantages to take those first steps in the game industry and in game developing. Continue to the next chapter, readers could understand about programming in Unreal Engine by learning the fundamental of C++ programming language and Blueprints Visual Scripting. Throughout the chapter, readers could have the basic understanding of Unreal programming and continue to the practical part in creating an Unreal game project. In the third chapter, the tutorial on making the project was very detailed, helped readers to understand about the concept and logic on creating and programming the game. With these knowledge combining with practical training, you could gain experience on game developing and the journey from beginner to become a junior developer could be easily archived. Unreal also

came with useful documentations, detail explanations on guiding readers through understanding basics, functions, building, creating, optimizing, and sharing many useful information. The community of Unreal Creators could also be reachable to get supports along the way.

Unreal is a great engine to learn. With the release of Unreal Engine 5, many companies, studios big and small had started working on the engine. Having the knowledge of Unreal Engine, you will open many doors ahead to dive into the game industry. Unreal is also partnering with ambitious trainers and training centers across the globe, creating events to connect and grows. You can have more information by visiting Unreal Engine Community.

In conclusion, with this research on game developing with Unreal Engine, I can share and contribute my personal experience and knowledge to inspire people who are interested in the game industry and can help them to archive their goal in different way. Through this research, I also have a chance to learn and develop myself more to be better. From my point of view, being a game developer is challenging because creating a game that can make others enjoy and feel the excitement while playing it is a task that not many could archive easily. The road ahead might be rough, but the potential of the game industry is limitless. With passion and hard work, we will be able to do anything.

REFERENCES

About Unreal Engine. Available at: https://en.wikipedia.org/wiki/Unreal_Engine [Accessed February 2022]

About Unity Engine. Available at: https://en.wikipedia.org/wiki/Unity_(game_engine) [Accessed February 2022]

Simran Kaur Arora, 2021. Unity vs Unreal Engine: Which Game Engine Should You Choose? Available at: https://hackr.io/blog/unity-vs-unreal-engine#:~:text=Unity%20has%20a%20wide%20range,making%20the%20development%20process%20easier. [Accessed February 2022]

Evelyn Trainor-Fogleman, 2021. Unity vs Unreal Engine: Game engine comparison guide for 2021. Available at: https://www.evercast.us/blog/unity-vs-unreal-engine [Accessed February 2022]

educba.com. Difference between Unreal Engine and Unity. Available at: https://www.educba.com/unreal-engine-vs-unity/ [Accessed February 2022]

Buvesa Game Development, 2021. Why I changed from Unity to Unreal Engine. Available at: https://www.youtube.com/watch?v=FvT64rNdHuo [Accessed February 2022]

About C++. Available at: https://en.wikipedia.org/wiki/C%2B%2B [Accessed March 2022]

Erin Schaffer, 2022. Is C++ still a good language to learn for 2022. Available at: https://www.educative.io/blog/learn-cpp-for-2022 [Accessed March 2022]

Amanda Fawcett, originally published in 2020, update in 2021. Learn C++ from scratch: The complete guide for beginners. Available at: https://www.educative.io/blog/how-to-learn-cpp-the-guide-for-beginners [Accessed March 2022]

Ryan Thelin, 2020. Intermediate C++ tutorial: strings, maps, memory, and more. Available at: https://www.educative.io/blog/intermediate-cpp-tutorial [Accessed March 2022]

Unreal Engine 4.27 Documentation. Unreal Engine Blueprint overview. Available at: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/Overview/#:~:text=The%20Blueprint%20Visual%20Scripting%20system,or%20objects%20in%20the%20engine. [Accessed March 2022]

Unreal Engine 4.27 Documentation. Balancing Blueprint and C++. Available at:
https://docs.unrealengine.com/4.27/en-US/Resources/SampleGames/ARPG/BalancingBlueprintAndCPP/ [Accessed March 2022]

Alex Forsythe, 2021. Blueprints vs. C++: How They Fit Together and Why You Should Use Both. Available at:
https://www.youtube.com/watch?v=VMZftEVDuCE [Accessed March 2022]

Unreal Engine 4.27 Documentation. Documentation on Unreal Engine Interface. Available at:
Viewports
https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LevelEditor/Viewports/

Modes
https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LevelEditor/Modes/

UI
https://docs.unrealengine.com/4.27/en-US/Basics/ContentBrowser/UI/

Details Panels
https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LevelEditor/Details/

World Outliners
https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LevelEditor/SceneOutliner/

Toolbar
https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LevelEditor/Toolbar/

Unreal Engine free asset store. Available at:
https://www.unrealengine.com/marketplace/en-US/free?count=20&sortBy=effectiveDate&sortDir=DESC&start=0
[Accessed March 2022]

Information about Unreal Engine 5. Available at:

https://en.wikipedia.org/wiki/Unreal_Engine#Unreal_Engine_5 [Accessed April 2022]

Unreal Sensei, 2022. 5 Reasons Unreal Engine 5 is a Big Deal. Available at:

https://www.youtube.com/watch?v=cRLnR4Kot2M [Accessed April 2022]

Unreal Engine 5.0 Release Notes. Available at:

https://docs.unrealengine.com/5.0/en-US/unreal-engine-5-0-release-notes/
[Accessed April 2022]

LIST OF FIGURES