Banishment - Game in Unreal Engine, C++ and Blueprints

Evan Smith

B.Sc.(Hons) in Software Development

April 10, 2022

Final Year Project

Advised by: Dr Martin Hynes

Department of Computer Science and Applied Physics
Galway-Mayo Institute of Technology (GMIT)

Contents

1 Introduction **5**
- 1.1 Programming Language . . . 5
- 1.2 Game Engine . . . 6

2 Context **7**
- 2.1 Github Repository . . . 7
- 2.2 Chapter Outline . . . 7
  - 2.2.1 Methodology . . . 7
  - 2.2.2 Technology Review . . . 8
  - 2.2.3 System Design . . . 8
  - 2.2.4 System Evaluation . . . 8

3 Methodology **9**
- 3.1 Approach to Development . . . 9
- 3.2 Testing . . . 9
- 3.3 Development Tools . . . 10
- 3.4 Research . . . 10

4 Technology Review **11**
- 4.1 The Game Engine . . . 11
  - 4.1.1 A brief history . . . 11
  - 4.1.2 Expected properties of a game engine . . . 12
  - 4.1.3 Choosing a game engine . . . 13
  - 4.1.4 Unreal Engine . . . 13
  - 4.1.5 Unreal Engine Blueprint . . . 14
  - 4.1.6 Unity . . . 15
  - 4.1.7 CryEngine . . . 16
- 4.2 C++ for Game Development . . . 17

5 System Design **19**
- 5.1 Player Character . . . 19

5.2 Player Input 20
5.3 Player Movement 21
5.4 Player Movement Animation 23
5.5 Weapons 25
5.6 Extra Player Functionality 26
5.6.1 Equipping a Weapon 27
5.6.2 Jumping 29
5.6.3 Sprinting 30
5.6.4 Attacking 31
5.7 Enemy Class 32
5.7.1 Enemy AI 33
5.7.2 Enemy Animation 34
5.7.3 Enemy Combat 36
5.8 Player Status: Health and Stamina 37
5.9 Dealing Damage 38
5.10 Enemy Health Bar 40
5.11 World Elements: Collectables and Hazards 40
5.12 Save and Load System 41

6 System Evaluation **45**
6.1 Testing 45
6.2 Limitations and Opportunities 45

7 Conclusion **47**

About this project

Abstract Action RPG games are one of the most popular game genres in today's world according to the Steam marketplace. So what makes them so popular and what is the best way to create them? In this paper, I look into the best way to create your own Action RPG game and what tools you can use to enhance your game development experience. I compare the pros and cons of a handful of the most powerful game engines in the market and discuss which one best suits your needs as a developer. Similarly, I take an in-depth look at the leading game development languages at your disposal.

I discuss my approach to the development lifecycle as well as go into detail on how I implemented the key features of an Action RPG. I show how to create a player character from the ground up including everything from basic movement to combat, animation triggering to health and stamina systems. Alongside the player, I also delve deep into intelligent enemy AI and their necessary features such as states, combat and animation. Lastly is something I feel is extremely necessary for single-player games, a save and load game system. I show how to create save game instances and reload all the features of the game down to the specifics such as players rotation.

Authors Evan Smith, 4th Year Software Development student, ATU (Atlantic Technological University)

Chapter 1

Introduction

My experience of programming during my academic studies at GMIT was very heavily focused around Java however as an avid gamer in my spare time I was extremely interested in exploring game development. Having previously studied game development using the Unity game engine alongside the C# language, I wanted to develop my skills further and focus my project on something new to me. I choose to study C++ as a game development language and use it along side Unreal Engine for my project. Consequently I began my journey by reading literature on the language and reading into the game engine's API references.

1.1 Programming Language

Settling on my language of choice was very easy. Having already studied Java, Python and C#, C++ was an obvious next step for me. C++ was first created in 1979 and continues to be used in abundance to this day. So, why is C++ used for game development? A big advantage that C++ as a game development language over others is its performance and efficiency. Gamers expect high performance alongside little to no delay when it comes to their favourite games. C++ complies much closer to the systems hardware than its counterparts which results in a much faster compile speed allowing the game engines to perform much better. The faster compile speed is not only a benefit for the player but also the developer. "The ability to iterate quickly on gameplay and get immediate feedback on design changes is paramount." [1]

A Key feature I found while using C++ is that the language does not rely on garbage collection, the developer is essentially the master of construction

and destruction. Being in control of when memory is allocated and freed means that you completely avoid any unpredictable glitches or problems that could arise when using an automated garbage collection system. For the likes of larger projects with a lot of scripting code this could have a significant negative impact.

C++ does not leave all the work up to you. The language is filled with open-source libraries that can take some of the workload off of you if used correctly. The libraries span a range of many different aspects of game development including but not limited to audio, debugging and graphic design. Alongside this is C++'s flexibility. The language allows you to easily develop games for an abundance of different platforms and operating systems.

1.2 Game Engine

Before finalizing my game engine choice I first downloaded a handful of different ones to try out, a few of which being CryEngine and Unity. Both of these engines proved to be fantastic in their own way. I tested them out by making a simple world with some simple character functionality but ultimately Unreal Engine was the best choice for me. Being one of the worlds leading game development engines I was eager to get start and learn everything the engine had to offer. It quickly became clear why the engine was so popular. Similar to C++, Unreal began its life in the mid 1990s. It quickly became popular due to its host of common functionality needed in the world of entertainment. Some of its key offerings include artificial intelligence support, 3D graphics rendering, networking infrastructure and input management which I found particularly helpful while developing my game.

Getting started in Unreal Engine proved to be quite difficult. The user interface, although clean and presentable, is jam packed full of features and can be easy to get lost in. One key feature that had Unreal Engine stand out above the rest is the beginner friendly tutorial you are greeted with upon startup. This tutorial walked me through all the main main features of the engine and showed me where all the most important features are kept. After testing out the engine for a few days, creating a simple level with basic functionality, I felt confident enough to being with my project.

Chapter 2

Context

From the start, I had a pretty clear concept of what I wanted to try to achieve with my project. Being a big fan of From Software's "Dark Souls" series I knew I wanted to create something similar. The key elements of a 'souls-like' game that I wanted to implement were;

- Smooth player movement.
- Fluid combat.
- A challenging game difficulty.
- An intelligent enemy AI.
- Complex player health and stamina systems.
- Variety in level landscape and the enemies that occupy them.

2.1 Github Repository

https://github.com/Smiithyy/AppliedProjectAndMinorDissertation

2.2 Chapter Outline

2.2.1 Methodology

In this section, I talk through my approach to my project's development cycle, how I went about testing my game as well as the research I did into the tools I used to create my game.

2.2.2 Technology Review

This section was all about the research I did into the game development tools available to developers. I compare a variety of different game engines, weighing the pros and cons of each to find the one that suited my needs. I also go into depth about the various programming languages that a developer can choose from when beginning their game development journey.

2.2.3 System Design

System design is all about the how of my game. How I went about implementing each feature of my game. I go into depth on how I created the player character alongside an intelligent enemy AI and how the two can perform combat together. I also discuss the creation of a complex player health and stamina system as well as various world elements such as collectables and hazards before finishing off with a standalone save/load game feature.

2.2.4 System Evaluation

In this final section, I reflect on the journey taken to create my game. I go in-depth into the testing phase of my game and discuss the limitations and opportunities that my game has.

Chapter 3

Methodology

In this section i introduce my approach to the development of my project, what kind of approach i took and how it effected my work flow.

3.1 Approach to Development

For my project, I decided to take an incremental approach to development. By breaking the workload down into smaller manageable pieces the development process went much smoother. Each increment either added to the previously worked on section or built off of it into the various features I had planned for the game. Panning for the game began with the major scope. What did I want my final product to achieve? Once I had my main objective set in place I worked backwards to form a plan and subsequent workflow chart. Using an online workflow management software called 'Monday.com' I was able to efficiently plan out each step of my development process. Alongside using 'Monday.com' I met with my supervisor, Dr Martin Hynes, weekly. The weekly meetings keep the development process flowing and added structure to the project. During these meetings, I had the opportunity to get valuable feedback on each part of my project while also discussing ideas and possible changes with Martin.

3.2 Testing

Being new to Unreal Engine, Blueprints and wanting to further my knowledge in C++, I decided to take on the task of personally testing and refining every feature of my game. This proved to be extremely beneficial. Play testing the features of my game allowed me to learn about how the code works alongside

blueprints in the engine thus I began to work more efficiently as I learned the best way to use the two tools in tandem.

3.3 Development Tools

A host of tools were used to complete this project.

- Unreal Engine 4.27 as the game engine.
- C++ as the programming language.
- Unreal Engine's build in visual scripting, Blueprint.
- Rider as the IDE.
- Unreal Engine 4.27 Documentation [2]
- Monday.com for online workflow management. [3]

3.4 Research

For the research portion of my project, I decided to take an empirical approach. I found that there was no better way of researching a game and its development process than to gather that information myself. The main question I needed to answer with this research was 'What do I want my game to do?'. Simply put, I wanted my game to give me the same feeling that my favourite games do, namely From Software's Dark Souls sires, so that's exactly where my research began. Having put thousands of hours into these games in the past I took a new approach while playing them for my research. I paid extra attention to the details taking notes as I went. "What features of the games appealed to me the most?" and "what features could I take inspiration from and improve on?". These were 2 very important questions I posed during my research. Being only one person I knew there was only so much research I could do so I took to steaming such as Twitch.tv and YouTube.com to watch and ask other gamers their opinions on the games. This proved to be very insightful as I received plenty of feedback on the game's most popular features as well as what they would like to see improved.

Chapter 4

Technology Review

4.1 The Game Engine

4.1.1 A brief history

What is a game engine? In its simplest form a game engine is a reuse-able piece of software that provides you with the tools and programs to help you build and customize a game; giving you a head-start in making your own games.

Before game engines, games were written as singular entities. The games had to be designed from the ground up to make optimal use of the desired platforms hardware. Thus most games through the 1980s were designed through a hard-coded ruleset and contained a small number of levels and graphics data. The term "game engine" was coined around the mid-1990s as hardware began to evolve and the need for quicker game development cycle became more prevalent. "A game engine is a reusable software layer allowing the separation of common game concepts from the game assets". [4] From Unreal Tournament, BioShock, Batman, Arkham, Final Fantasy VII Remake, to Fortnite, Unreal Engine (UE) has been the backbone of so many of our favourite games. Throughout its history, the engine has maintained itself as one of the most popular development platforms in the games industry. Its popularity is largely due to its extensive customizability, multi-platform capabilities and the ability to create AAA quality games with it. "After we began showing an early version of Unreal, our two first licensees – Legend Entertainment and Microprose – called us up and asked us about the possibility of using our engine in their games." - Sweeney

4.1.2 Expected properties of a game engine

A lot is expected from a game engine in order for it to be worth while using. A number of requirements need to be met;

- A rendering engine, for 2D and/or 3D graphics.
- A scene graph for managing graphical elements.
- Input handling, for keyboard & mouse, game-pad, touch devices or other hardware etc.
- A game loop for recalculating game events every frame.
- A physics engine with collision detection and response.
- A sound engine.
- Animation.
- Memory management.
- Process threading for managing multiple, parallel processes.

Additional functionalities may include;

- Scripting.
- Support for artificial intelligence.
- Networking support.
- Multi-platform publishing.

By in large, these extra features are supported by the engine, either by default or through third-party libraries. Bringing all these features together under one piece of software is ultimately the game engine developer's goal.

"A major trend for some game engines is a common goal to enable cross-platform publishing while keeping the same code base."[4] Therefore, most engines have the capability of not only publishing for desktop but also most of the major operating systems and gaming platforms.

4.1.3 Choosing a game engine

If you have ever googled the phrase "game engine" then you have witnessed, first-hand, the sheer volume of engines there are on offer. Thus raising the question, which engine is the right one for me? Certain engines have grown so broad and powerful that they can be used to develop any kind of game, while others have narrowed the scope and kept the engine simpler to make it more efficient at developing a certain type of game, appealing to a specific audience. Some engines do not even require any programming knowledge and are extremely friendly to newcomers. It is important to compare a variety of engines and look into their pros and cons for the type of project you aim to use them for. Now I will compare several different engines, surveying their pros and cons, target audiences and overall charm.

4.1.4 Unreal Engine

Epic Games released their fifth iteration of Unreal Engine (UE5) on April 5th 2022 however for the sake of this comparison section I shall be focusing on Unreal Engine 4, specifically version 4.27.

Unreal Engine 4 (UE4), being the fourth generation of UE, was released in 2014. As a game engine, it provides all the tools a developer would need to implement the common functionality needed in entertainment software. These features include but are not limited to; a 3D rendering system, real-time audio, input management, networking infrastructure and artificial intelligence support. UE4 became popular for its wide range of platform support including games consoles, desktop and mobile operating systems as well as virtual reality. Alongside being used with these platforms, UE4 is steadily becoming more popular for use in film as well as architecture. "Although initially developed to support the Unreal first-person shooter, and arguably not as rich-featured as Unity, the Unreal Engine has grown to be a very powerful engine capable of supporting any game genre (including 2D environments)." [4]

UE4 can be very daunting for new users. The sheer volume of menus and tools can have you feeling lost quickly, however, UE4 recognises newcomers and greets them with a fantastic tutorial of the engine. There is also a fantastic community surrounding UE who are always more than happy to help a developer in any area they may be struggling in.

Scripting within the engine is done using the C++ language. C++ is a fantastic language to use alongside the engine, it provides the user with great efficiency and performance as it compiles much closer to the system's hardware allowing testing of minor code tweaks to happen much quicker than

its counterpart languages. I will touch on C++ more a little later.

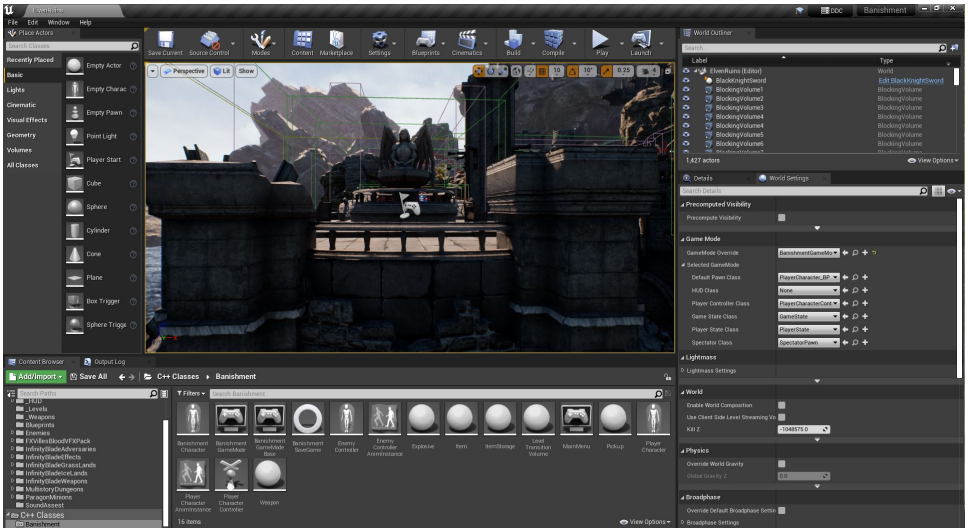

Figure 4.1: Unreal Engine Editor

4.1.5 Unreal Engine Blueprint

Scripting for UE4 does not have to be done solely in C++, or at all for that matter. Alongside the release of Unreal Engine 4 in 2014 Epic Games also premiered their brand new visual scripting system known as Blueprint. Blueprint is a node-based visual scripting system that allowed the developer to create game-play elements from within the editor itself. Blueprint has the ability to allow the developer to quickly prototype and visually experiment with their concept before translating it into C++ code. This advantage does not only apply to developers but also to any member of a team that may have work to do within the engine but have no knowledge of programming themselves, allowing them to test their work before it gets handed over to the developers. "This tool is described as being so powerful that entire games can be built using it exclusively" [4]

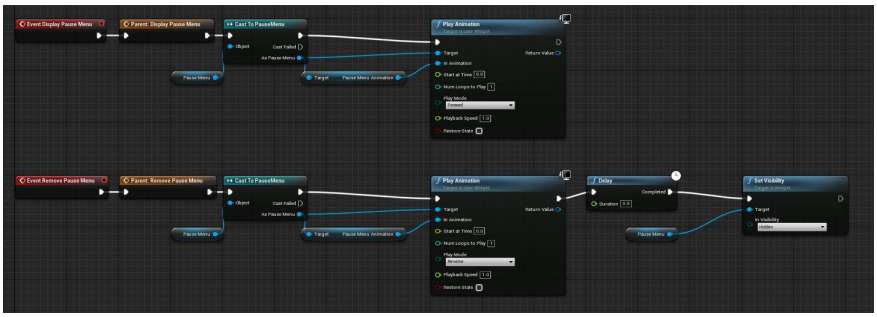

Figure 4.2: Blueprint Example - My Games Pause Menu Functionality

4.1.6 Unity

Unity was first premiered by Apple at their Worldwide Developers Conference in 2005. Since its release, it has gained huge traction in the game development world, building itself a massively dedicated fan base and community.

One of the major advantages of working with Unity is its platform support, currently, it can export to over 20 platforms at the time of writing this. Unity is made up of a visual editor and an IDE which allows the developer to quickly prototype their projects and even create simple scenes without a single line of code. Like UE4, Unity comes packed with all of the necessary features a developer would want when starting their project including 2D and 3D rendering, audio, animation and a scene graph among many others. One area in which Unity falls short of UE4 is its lack of input manager. All game-play inputs in Unity must be done via scripting which is written in C#.

As mentioned, scripting for Unity is done via C# however Unity also has its own scripting language largely based on Java-Script type syntax. By in large, the engine uses a lot of middle-ware to support its features, ”namely ’Mono’ for scripting, ’DirectX’ and ’OpenGL’ for rendering, ’Beäst’ and ’Substance’ for graphics, ’fmod’ for sound, ’PhysX’ as physics engine, ’RakNet’ for networking etc.” [4]

Alongside Unity’s dedicated community is its vast asset store. The store is filled with free and paid assets and components that can be used to enhance your game, speed up the development process or serve as learning material for new users to practice on. Unity has been used to create a plethora of both smaller scale mobile games such as ”Temple Run” or AAA industry level projects such as ”Kerbal Space Program” and ”Escape from Tarkov”. All in all Unity is a fantastic engine for new comes and experienced users alike.

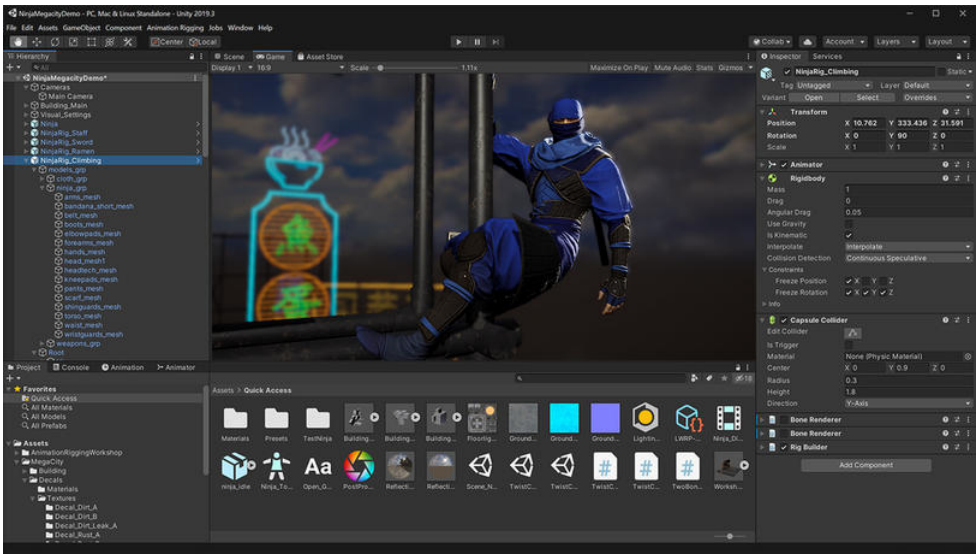

Figure 4.3: Unity Editor

4.1.7 CryEngine

CryEngine was another game engine that I touched on while deciding which engine would work best for my project. CryEngine (CE) was first released in 2002 by the German game developer Crytek. Since its initial release CE has gone through numerous iterations with its latest version, version 5.6.7, having been released in July of 2020.

CryEngine is an industry-level game engine that Crytek developed to support all of their games starting with 'Far Cry' and all its sequels. In comparison to the previously discussed engines, CE is certainly the least friendly to new users. Its editor's interface has a steep learning curve being packed with buttons and features making it extremely hard to use if you don't know your way around it. On top of the confusing layout to the editor, CE doesn't have a very big community behind it to seek help from. This coupled with the extremely small asset database makes the engine very hard for new or even experienced developers to use.

Nevertheless, CryEngine has been used to create countless amazing games in its time! From the 'Far Cry' and 'Crysis' series' to all the 'Sniper: Ghost Worrier' games, CE has produced some of the most influential games of our time.

Last but certainly not least is CryEngines graphics. It is no secret that the engine is capable of producing some of the best graphics known to video games. If the engine is in the right hands it can be extremely powerful as seen with Crytek's 'Ryse: Son of Rome' which won the SIGGRAPH award

for 'Best Real-Time Graphics' back in 2014.

In terms of scripting, CE is similar to UE4. While working mainly off of the C++ language the engine also has the capability to work with the Lua languages. Like Unreal, CryEngine has its own version of visual scripting that they have called 'FlowGraph'. Although not as easy to use as Unreal's Blueprint, FlowGraph is a very powerful visual scripting system that, once again, can provide the developer with fantastic support during their project's development life cycle if they put the hours in to learn how it works.

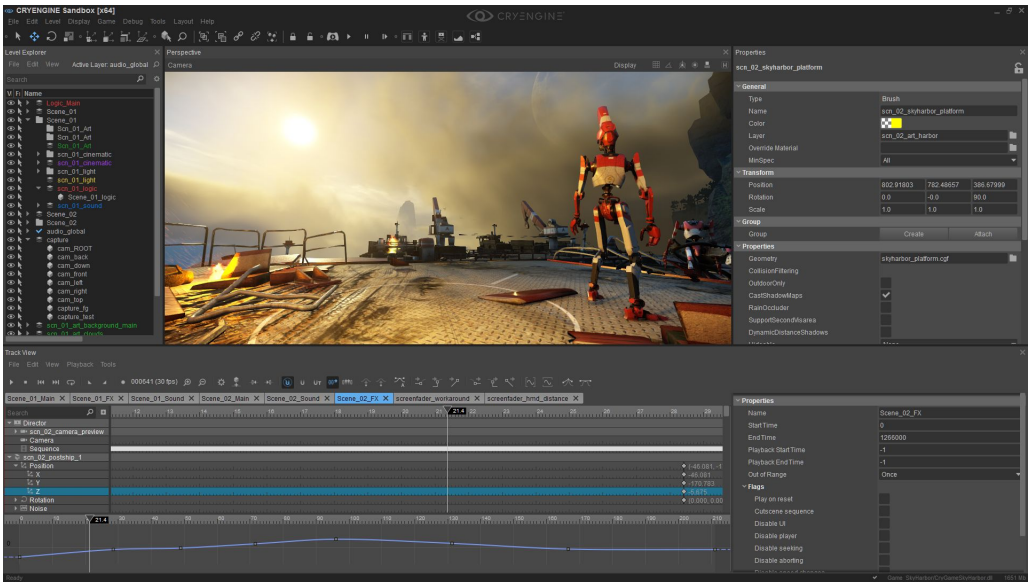

Figure 4.4: CryEngine Editor

4.2 C++ for Game Development

When it comes to games and game development, run-time efficiency is vastly more important that iteration time thus making C++ the obvious choice of programming language. Its power and efficiency has the language standing tall above its counterparts. For a developer, the ability to get almost instant feedback on design changes is key meaning C++'s ability to compile quickly without the need to restart or reload the game is perfect.

While some companies choose to create their games in C, the vast majority of developers and studios lean towards C++ for their projects. As games grow exponentially in size and complexity, developers needed the best way to deal with the vast amount of code. "With C++ you can more easily manage and maintain huge amounts of code than with C." [1]

So what makes C++ the so popular among developers? Well for one, the ability to directly allocate raw memory is a huge advantage. C++, unlike its counterparts, does not rely on garbage collection and instead allows the developer to be in complete control of their codes construction and destruction by using pointers and encapsulation in classes. This has major benefits when it comes to large game projects with lots of scripting code. Relying on a garbage collector to do its job correctly all of the time could have catastrophic effects on the project however all of this is avoided when the developer is 100% control over what code is created and destroyed. C++ as we know is an Object Orientated Programming (OOP) language. "Programmuing styles, such as OOP, help to organize code in such a way that it becomes easier to maintain and modify." [1]

Ultimately C++'s popularity comes down to its vast array of open-source libraries for the developer to pick from. The libraries allow the developer to call commonly used methods and functions from their own classes thus cutting down on code length, keeping the code tidier and avoiding any mistakes when attempting to program the method yourself. The open-source libraries span a wide range of programming aspects, a few of which being audio, debugging and graphic design.

C++ example - Function for equipping a weapon, taken from my project:

```
void APlayerCharacter::WeaponEquipDown()
{
        bWeaponEquipDown = true;

        if (MovementStatus == EMovementStatus::EMS_Dead) return;

        if (ActiveOverlappingItem)
        {
                AWeapon* Weapon = Cast<AWeapon>(ActiveOverlappingItem);
                if (Weapon)
                {
                        Weapon->Equip(this);
                        SetActiveOverlappingItem(nullptr);
                }
        }
}
```

Chapter 5

System Design

An overview of my game's design and how each element was implemented.

5.1 Player Character

The first step was finding the model I wanted to use for my player character. For this, I used a website called 'Mixamo.com'. Once I had chosen my model it was a simple download and import into Unreal Engine before I could start using it. Once I had my character model imported I made a C++ script called 'PlayerCharacter' (PC) which I would use for all the player scripting.

In the PC script, I began by setting up the camera boom, follow camera and collision capsule for the player. See Figure 5.1

```
// Create camera boom (zooms in on player if there is a collision)
CameraBoom = CreateDefaultSubobject<USpringArmComponent>(TEXT("CameraBoom"));
CameraBoom->SetupAttachment(InParent: GetRootComponent());
CameraBoom->TargetArmLength = 600.f; // Camera follows at this distance
CameraBoom->bUsePawnControlRotation = true; // Rotate arm based on controller

// Set size for collision capsule
GetCapsuleComponent()->SetCapsuleSize(InRadius: 26.f, InHalfHeight: 85.f);

// Create follow camera
FollowCamera = CreateDefaultSubobject<UCameraComponent>(TEXT("FollowCamera"));
// Attach camera to end of the boom
FollowCamera->SetupAttachment(CameraBoom, USpringArmComponent::SocketName);
FollowCamera->bUsePawnControlRotation = false; // CameraBoom already rotates matching controller orientation
```

Figure 5.1: Camera Scripting

Lastly, I created a blueprint based on the PC C++ script and set the mesh as the player model I imported earlier. See Figure 5.2 and 5.3

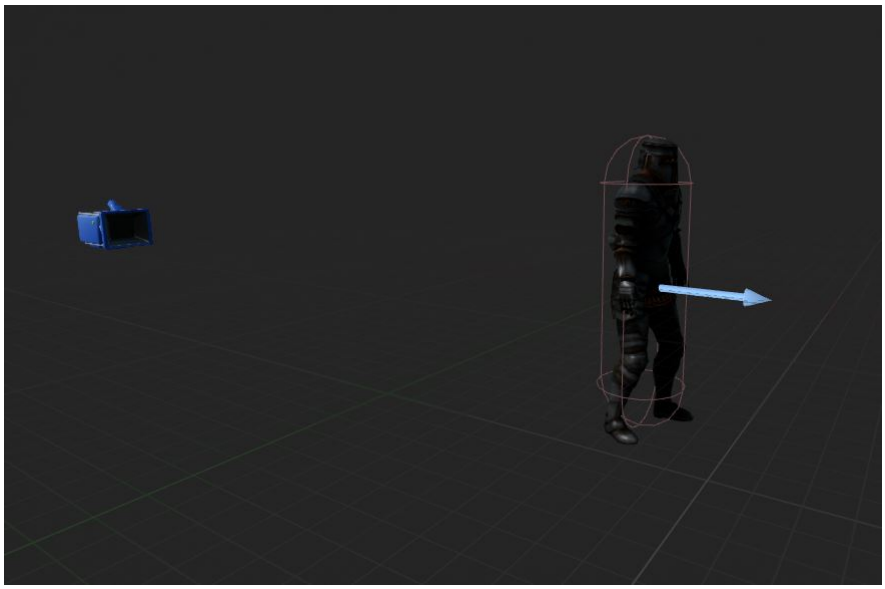

Figure 5.2: Player Blueprint

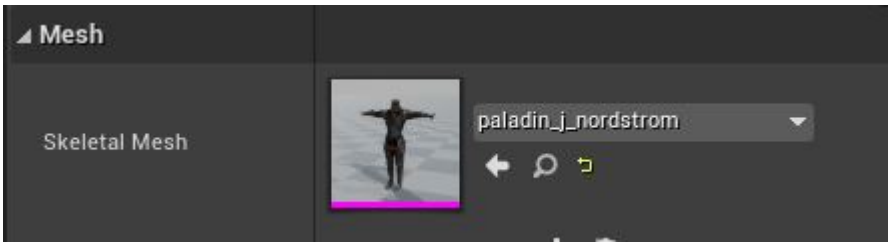

Figure 5.3: Player Mesh

5.2 Player Input

Thanks to Unreal Engine's input manager, setting up my games inputs was extremely easy as seen in Figure 5.4.

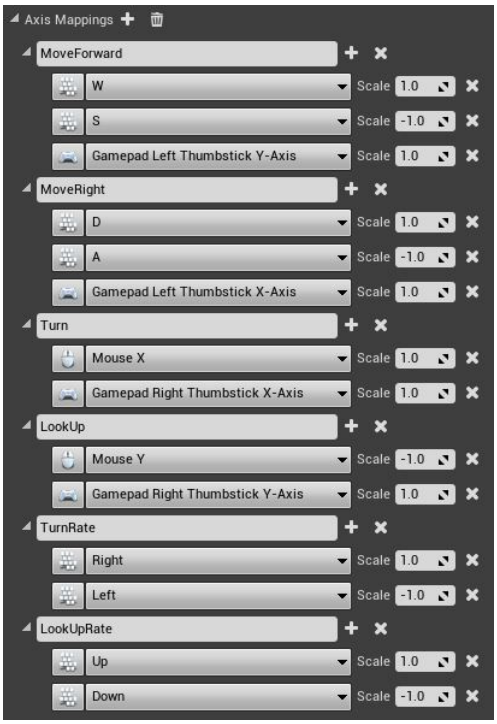

Figure 5.4: Movement Input Mapping

Setting up the input mapping in the C++ script was very easy thanks to C++'s open-source libraries. A single line for each input was all that was required to have the scrip linked to the input mapping in the engine. See Figure 5.5

```
PlayerInputComponent->BindAxis("MoveForward", Object: this, &APlayerCharacter::MoveForward);
PlayerInputComponent->BindAxis("MoveRight", Object: this, &APlayerCharacter::MoveRight);

PlayerInputComponent->BindAxis("Turn", Object: this, &APawn::AddControllerYawInput);
PlayerInputComponent->BindAxis("LookUp", Object: this, &APawn::AddControllerPitchInput);
PlayerInputComponent->BindAxis("TurnRate", Object: this, &APlayerCharacter::TurnAtRate);
PlayerInputComponent->BindAxis("LookUpRate", Object: this, &APlayerCharacter::LookUpAtRate);
```

Figure 5.5: Movement Input Binding

5.3 Player Movement

Next up was the player movement. I did this by taking advantage of the Controller library. I started by using the boolean 'bOrientRotationToMovement' to rotate the character in line with the direction of movement. I also set up

the character's rotation and movement speeds as well as some booleans for checking if the player is moving or sprinting here. See Figure 5.6

```
// Configure character movement
GetCharacterMovement()->bOrientRotationToMovement = true; // Character moves in direction of input...
GetCharacterMovement()->RotationRate = FRotator( InPitch: 0.0f,  InYaw: 740.f,  InRoll: 0.0f); // ...at this rotation rate

RunningSpeed = 650.f;
SprintingSpeed = 950.f;

bMovingForward = false;
bMovingRight = false;
bShiftKeyDown = false;
```

Figure 5.6: Movement Scripting

The movement functions were straightforward. After a few safety checks to make sure the player should be allowed to move I got the rotation and direction of the input and had the character move about those values. See Figure 5.7

```
// Forward/backward movement
void APlayerCharacter::MoveForward(float Value)
{
    bMovingForward = false;

    if ((Controller != nullptr) && (Value != 0.0f) && (!bAttacking) && (MovementStatus != EMovementStatus::EMS_Dead))
    {
        // Find out which way is forward
        const FRotator Rotation = Controller->GetControlRotation();
        const FRotator YawRotation( InPitch: 0.f, Rotation.Yaw,  InRoll: 0.f);

        const FVector Direction = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::X);
        AddMovementInput(Direction, Value);

        bMovingForward = true;
    }
}

// Side to side movement
void APlayerCharacter::MoveRight(float Value)
{
    bMovingRight = false;

    if ((Controller != nullptr) && (Value != 0.0f) && (!bAttacking) && (MovementStatus != EMovementStatus::EMS_Dead))
    {
        // Find out which way is forward
        const FRotator Rotation = Controller->GetControlRotation();
        const FRotator YawRotation( InPitch: 0.f, Rotation.Yaw,  InRoll: 0.f);

        const FVector Direction = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::Y);
        AddMovementInput(Direction, Value);

        bMovingRight = true;
    }
}
```

Figure 5.7: Movement Execution

Lastly, I set up the turning and lookup rate for the character by simply giving some variables values and inputting them into yaw and pitch methods. See Figure 5.8 and 5.9

```
//Set turn rates for input
BaseTurnRate = 65.f;
BaseLookUpRate = 65.f;
```

Figure 5.8: Variables for turnAt and lookUp rates

```
// Turning player
void APlayerCharacter::TurnAtRate(float Rate)
{
    AddControllerYawInput(Val: Rate * BaseTurnRate * GetWorld()->GetDeltaSeconds());
}

// Player look up/down
void APlayerCharacter::LookUpAtRate(float Rate)
{
    AddControllerPitchInput(Val: Rate * BaseLookUpRate * GetWorld()->GetDeltaSeconds());
}
```

Figure 5.9: Methods for turnAt and lookUp rates

5.4 Player Movement Animation

Next on the list was the animation for movement, once again I used Mixamo.com to source them. Mixamo is a fantastic tool for character meshes and animations, it allows you to preview your chosen character mesh with the animation before deciding to download them. See Figure 5.10 I started off by finding three basic animations for my player; Idle, Walking and Running.

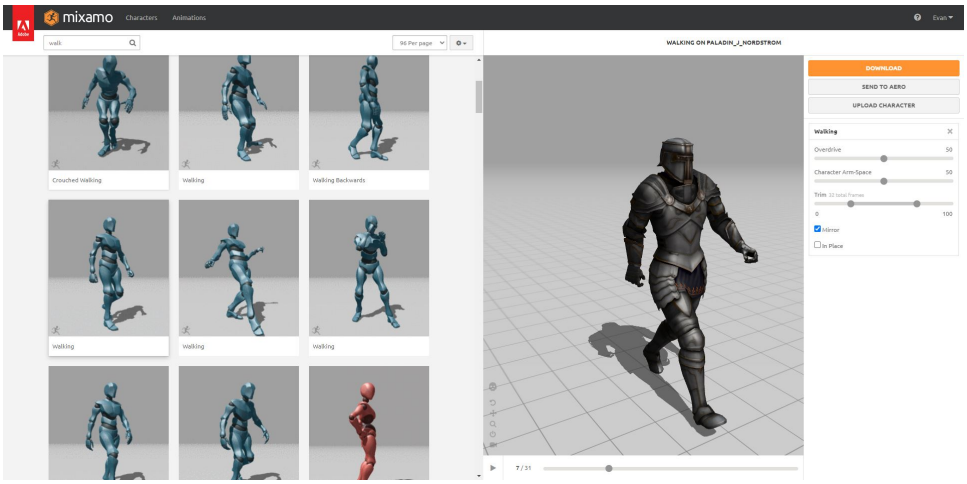

Figure 5.10: A look at previewing animations with chosen mesh

Once I had imported my animations I started by making a blend space for them and adding them to the timeline. See Figure 5.11

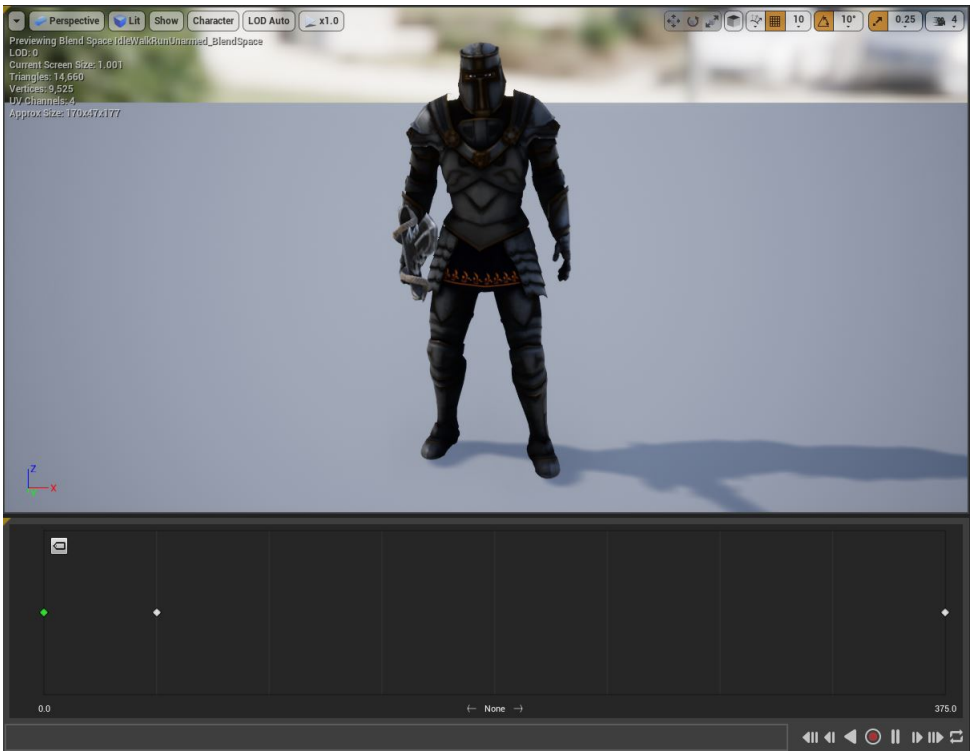

Figure 5.11: Blend space used for Idle/Walk/Run animation

Once the blend space was created I then needed an animation blueprint

for my character. Setting up the animation was extremely straight forward thanks to UE's blueprint. After setting up a state machine a simple click and drag had my animation in. By using the character's movement speed as the input it allowed the animation blend space to change between the states which was then outputted to the animation post. See Figure 5.12

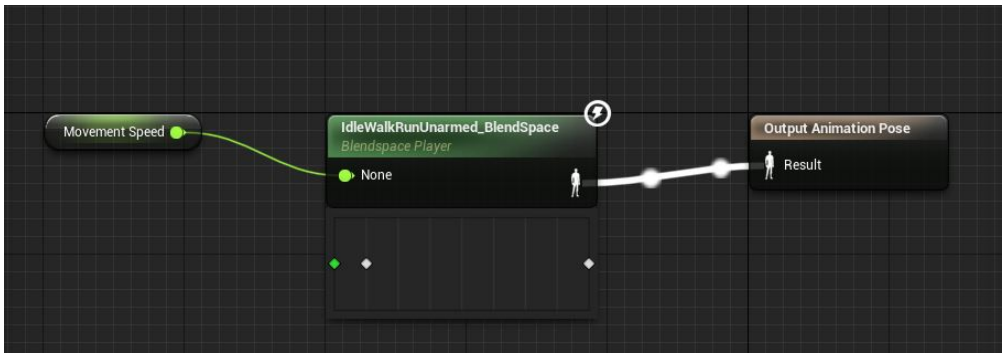

Figure 5.12: AnimBP for Idle/Walk/Run

5.5 Weapons

What kind of a game would this be without weapons? For this, I took to the UE marketplace to find some assets. The marketplace is packed with free assets for developers to use so finding some weapons for my game was very easy. I settled on using a pack called 'Infinity Blade: Weapons' created by Epic Games themselves to support their 'Infinity Blade' games. See Figure 5.13

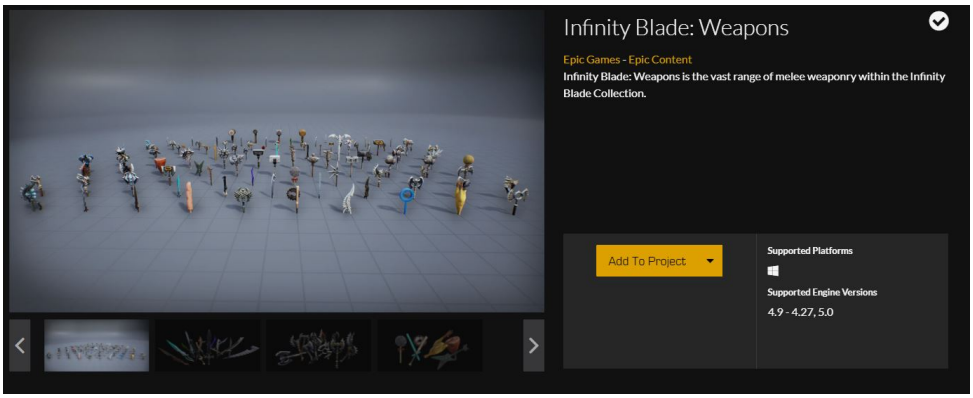

Figure 5.13: Infinity Blade: Weapons

For the weapons, I started similarly to the character, by creating a C++

script, called 'Weapon', and implementing all the basic features for the item such as its collision, damage value and skeletal mesh. See Figure 5.14

```
AWeapon::AWeapon()
{
    SkeletalMesh = CreateDefaultSubobject<USkeletalMeshComponent>(TEXT("SkeletalMesh"));
    SkeletalMesh->SetupAttachment( InParent: GetRootComponent());

    CombatCollision = CreateDefaultSubobject<UBoxComponent>(TEXT("CombatCollision"));
    CombatCollision->SetupAttachment( InParent: GetRootComponent());

    WeaponState = EWeaponState::EWS_Pickup;

    Damage = 25.f;
}
```

Figure 5.14: Weapon C++ Script

Lastly, I created a blueprint based on this 'Item' C++ class and selected the weapon mesh I wanted to use. See Figure 5.15

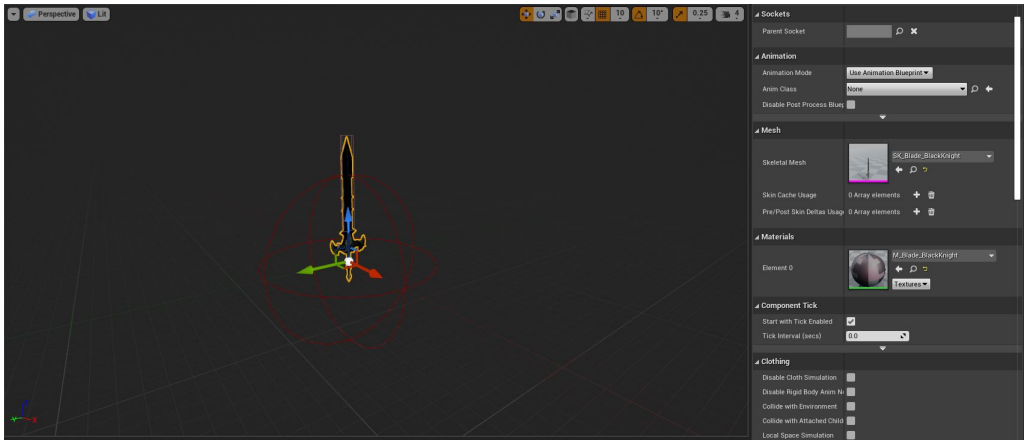

Figure 5.15: Blueprint for a Weapon

5.6 Extra Player Functionality

Next up I wanted to add extra functionality to the player. Namely, the ability to pick up weapons, attack, sprint and jump. For this I needed to set up some more action mapping, this was done in the same way as the movement inputs. See Figure 5.16 and Figure 5.17

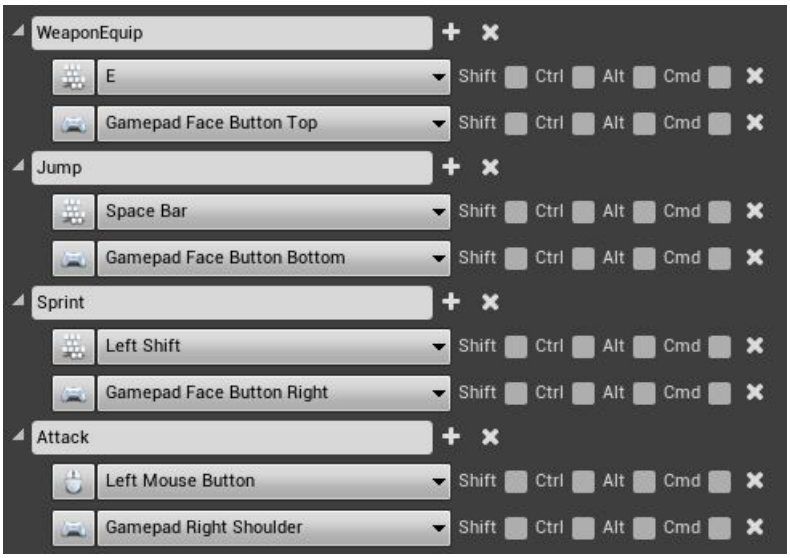

Figure 5.16: Player Action Mapping

```
PlayerInputComponent->BindAction("Jump", KeyEvent: IE_Pressed, Object: this, &APlayerCharacter::Jump);
PlayerInputComponent->BindAction("Jump", KeyEvent: IE_Released, Object: this, &ACharacter::StopJumping);

PlayerInputComponent->BindAction("Sprint", KeyEvent: IE_Pressed, Object: this, &APlayerCharacter::ShiftKeyDown);
PlayerInputComponent->BindAction("Sprint", KeyEvent: IE_Released, Object: this, &APlayerCharacter::ShiftKeyUp);

PlayerInputComponent->BindAction("WeaponEquip", KeyEvent: IE_Pressed, Object: this, &APlayerCharacter::WeaponEquipDown);
PlayerInputComponent->BindAction("WeaponEquip", KeyEvent: IE_Released, Object: this, &APlayerCharacter::WeaponEquipUp);

PlayerInputComponent->BindAction("Attack", KeyEvent: IE_Pressed, Object: this, &APlayerCharacter::LMBDown);
```

Figure 5.17: Player Action Binding

5.6.1 Equipping a Weapon

To be able to equip a weapon the player skeleton needs a socket to which the weapon could be attached on pick up. I also added a sword to the socket as 'Preview only' to correctly align the sword in the socket. See Figure 5.18

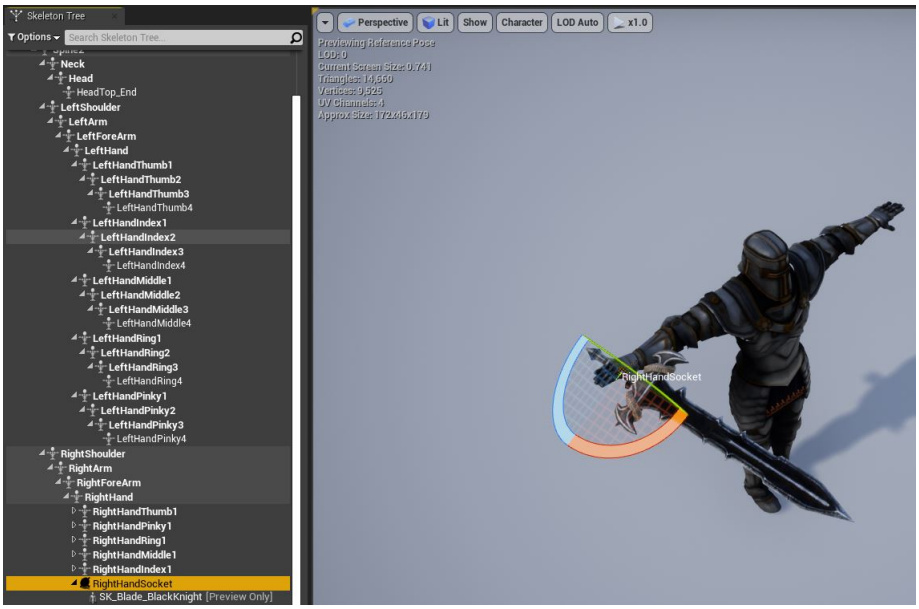

Figure 5.18: Weapon Socket on Player Skeleton

Next, I implemented an equip method that could be called by the player script when the 'WeaponEquip' input was pressed while overlapping with the weapon's collision volume. See Figure 5.19 and Figure 5.20

```
void APlayerCharacter::WeaponEquipDown()
{
    bWeaponEquipDown = true;

    if (MovementStatus == EMovementStatus::EMS_Dead) return;

    if (ActiveOverlappingItem)
    {
        AWeapon* Weapon = Cast<AWeapon>(ActiveOverlappingItem);
        if (Weapon)
        {
            Weapon->Equip(Char: this);
            SetActiveOverlappingItem(nullptr);
        }
    }
}
```

Figure 5.19: Function in Player Script Calling Equip from Weapon Script

```
void AWeapon::Equip(APlayerCharacter* Char)
{
    if (Char)
    {
        SetInstigator(Inst: Char->GetController());

        SkeletalMesh->SetCollisionResponseToChannel(ECollisionChannel::ECC_Camera, ECollisionResponse::ECR_Ignore);
        SkeletalMesh->SetCollisionResponseToChannel(ECollisionChannel::ECC_Pawn, ECollisionResponse::ECR_Ignore);

        SkeletalMesh->SetSimulatePhysics(bEnabled: false);

        const USkeletalMeshSocket* RightHandSocket = Char->GetMesh()->GetSocketByName("RightHandSocket");

        if (RightHandSocket)
        {
            RightHandSocket->AttachActor(this, SkelComp: Char->GetMesh());
            bRotate = false;

            Char->SetEquippedWeapon(this);
            Char->SetActiveOverlappingItem(nullptr);

        }
        if (OnEquipSound) UGameplayStatics::PlaySound2D(WorldContextObject: this, OnEquipSound);
    }
}
```

Figure 5.20: Weapon Equip Function

5.6.2 Jumping

As mentioned many times before, C++'s open-source libraries are a developer's best friend, making your player jump is a prime example of this! All that is required is to set some values to variables, see Figure 5.21, and to call the jump function from the super class, see Figure 5.22

```
GetCharacterMovement()->JumpZVelocity = 550.f;
GetCharacterMovement()->AirControl = 0.3f;
```

Figure 5.21: Setting Variable Values for Jumping

```
void APlayerCharacter::Jump()
{
    if (MovementStatus != EMovementStatus::EMS_Dead)
    {
        Super::Jump();
    }
}
```

Figure 5.22: Calling Jump Function from Super Class

5.6.3 Sprinting

For sprinting I decided to use an enum class, see Figure 5.23 to keep track of what state the player is in. Going about it this way had many implementations on my player class, not only allowing me to trigger the sprint feature but to all use it as checks before allowing certain code to execute e.g. See Figure 5.19, the player cannot equip a weapon if in the 'Dead' state.

```
enum class EMovementStatus : uint8
{
    EMS_Normal UMETA(DisplayName = "Normal"),
    EMS_Sprinting UMETA(DisplayName = "Sprinting"),
    EMS_Dead UMETA(DisplayName = "Dead"),

    EMS_MAX UMETA(DisplayName = "DefaultMAX")
};
```

Figure 5.23: enum Class for Player Status

Triggering sprinting was done through blueprints in the animationBP. Once the player was holding the sprint button, the player state would be changed to sprinting allowing the animation to play, see Figure 5.24 and Figure 5.25, and the movement speed to be increased.

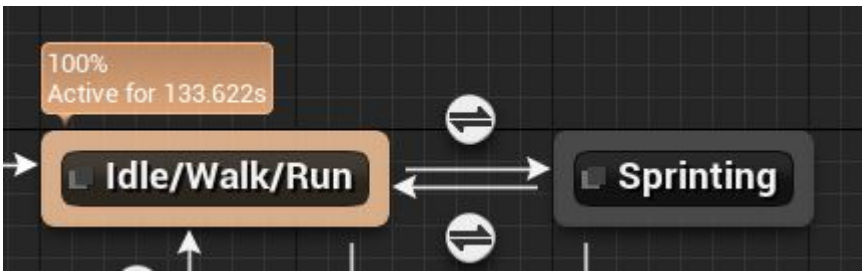

Figure 5.24: Idle/Walk/Run Relation to Sprinting

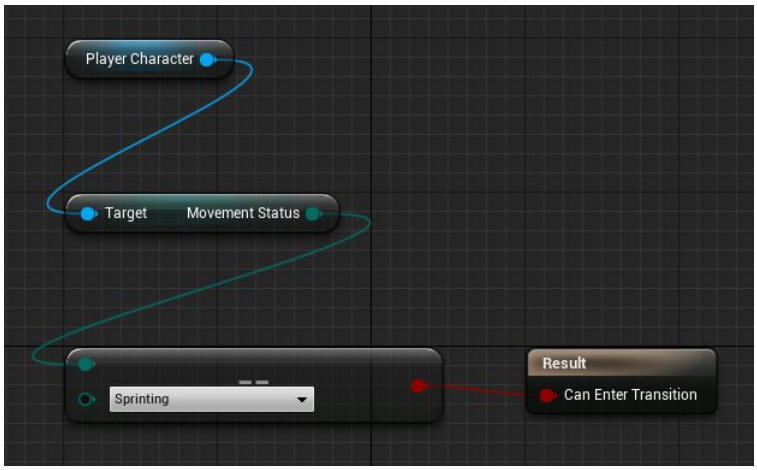

Figure 5.25: Requirements for Transitioning from Running to Sprinting

5.6.4 Attacking

For the player's attack and death animations, I used an animation montage. The montage allowed me to add multiple animations to the group and call upon them individually using section. See Figure 5.26

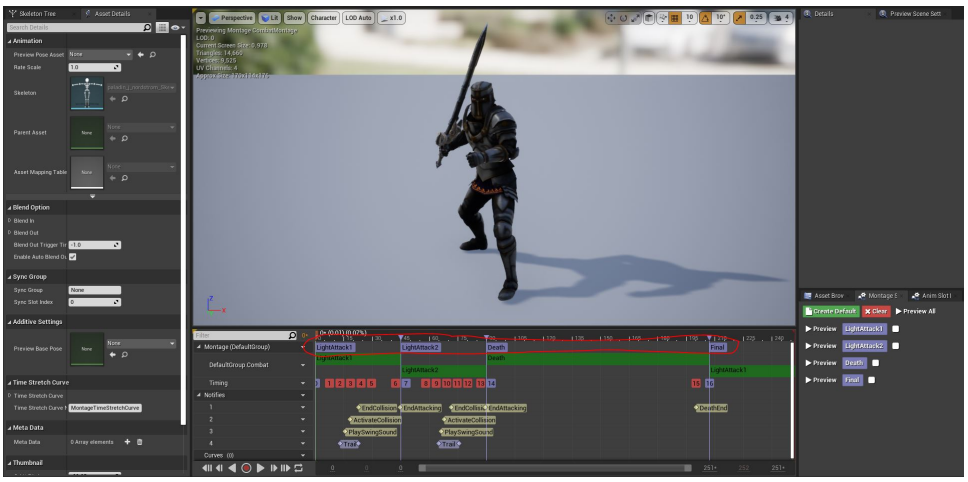

Figure 5.26: Circled in Red are the Montage Sections used to call each Animation

The animation sections can be called from the C++ script. I used a switch statement determined by a 'RandRange' math function to give the attack animation a bit of variety. See Figure 5.27

```cpp
void APlayerCharacter::Attack()
{
    if (!bAttacking && MovementStatus != EMovementStatus::EMS_Dead)
    {
        bAttacking = true;
        SetInterpToEnemy(true);

        UAnimInstance* AnimInstance = GetMesh()->GetAnimInstance();
        if(AnimInstance && CombatMontage)
        {
            int32 Selection = FMath::RandRange( Min: 0, Max: 1);
            switch(Selection)
            {
            case 0:
                AnimInstance->Montage_Play(CombatMontage, InPlayRate: 1.3f);
                AnimInstance->Montage_JumpToSection(FName("LightAttack1"), CombatMontage);

                break;

            case 1:
                AnimInstance->Montage_Play(CombatMontage, InPlayRate: 1.3f);
                AnimInstance->Montage_JumpToSection(FName("LightAttack2"), CombatMontage);

                break;

            default:
                ;
            }
        }
    }
}
```

Figure 5.27: C++ Script to call Attack Animations

5.7 Enemy Class

The enemy class started much the same as the player. I began by creating a C++ script called 'EnemyController' in which I set the various collision spheres needed to allow the AI to work. See Figure 5.28

```cpp
AEnemyController::AEnemyController()
{
    // Set this character to call Tick() every frame.  You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    AgroSphere = CreateDefaultSubobject<USphereComponent>(TEXT("AgroSphere"));
    AgroSphere->SetupAttachment( InParent: GetRootComponent());
    AgroSphere->SetCollisionResponseToChannel(ECollisionChannel::ECC_WorldDynamic, ECollisionResponse::ECR_Ignore);
    AgroSphere->InitSphereRadius(500.f);

    CombatSphere = CreateDefaultSubobject<USphereComponent>(TEXT("CombatSphere"));
    CombatSphere->SetupAttachment( InParent: GetRootComponent());
    CombatSphere->InitSphereRadius(125.f);
```

Figure 5.28: Script for Enemy Collision Spheres

Next, I created a Blueprint class for the enemy based on the EnemyController script. Inside this class, I adjusted the radius for the collision spheres

as well as set the skeletal mesh, see Figure 5.29, for the enemy which I got from another asset pack called 'Infinity Blade: Adversaries'. See Figure 5.30

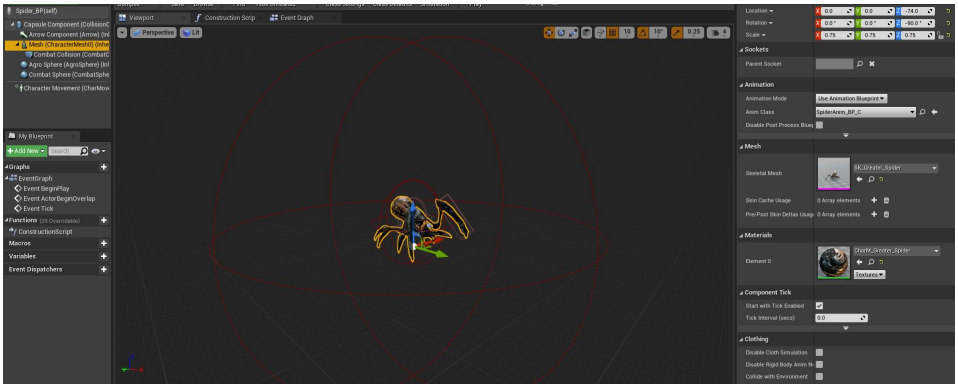

Figure 5.29: Enemy Blueprint Class

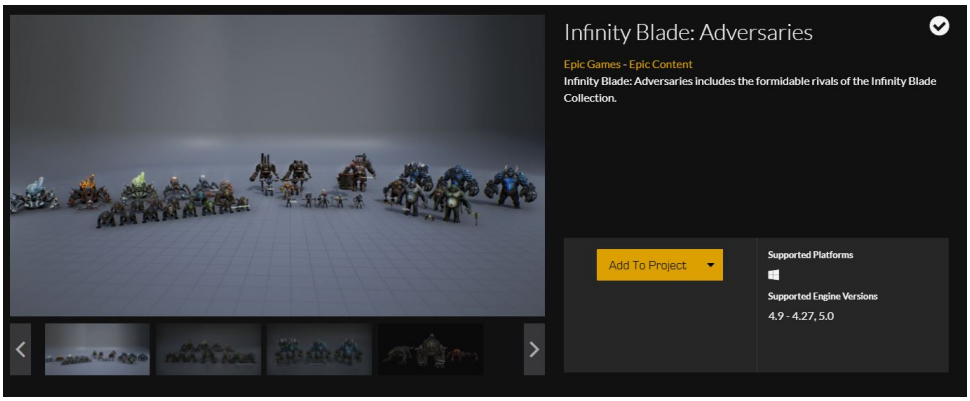

Figure 5.30: Infinity Blade: Adversaries Asset Pack

5.7.1 Enemy AI

For the enemy AI, I decided to use another enum class. The enum class allowed me to determine what state the enemy was in and thus execute a certain block of code. See Figure 5.31

```
UENUM(BlueprintType)
enum class EEnemyMovementStatus :uint8
{
    EMS_Idle            UMETA(DisplayName = "Idle"),
    EMS_MoveToTarget    UMETA(DisplayName = "MoveToTarget"),
    EMS_Attacking       UMETA(DisplayName = "Attacking"),
    EMS_Dead            UMETA(DisplayName = "Dead"),

    EMS_MAX             UMETA(DisplayName = "DefaultMax")
};
```

Figure 5.31: Enemy enum Class for Determining Enemy Status

Using a collision sphere I named 'AgroSphere' i was able to keep track of whether or not the player was inside the sphere. if they were, the enemy would agro and follow the player, if the player was outside the sphere the enemy would no longer be agro and return to its idle state. See Figure 5.32

```
void AEnemyController::AgroSphereOnOverLapBegin(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResults)
{
    if (OtherActor && Alive())
    {
        APlayerCharacter* PlayerCharacter = Cast<APlayerCharacter>(OtherActor);
        if (PlayerCharacter)
        {
            MoveToTarget(PlayerCharacter);
        }
    }
}

void AEnemyController::AgroSphereOnOverLapEnd(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex)
{
    if (OtherActor)
    {
        APlayerCharacter* PlayerCharacter = Cast<APlayerCharacter>(OtherActor);
        if (PlayerCharacter)
        {
            bHasValidTarget = false;
            if (PlayerCharacter->CombatTarget == this)
            {
                PlayerCharacter->SetCombatTarget(nullptr);
            }
            PlayerCharacter->SetHasCombatTarget(false);
            PlayerCharacter->UpdateCombatTarget();

            SetEnemyMovementStatus(EEnemyMovementStatus::EMS_Idle);
            if (AIController)
            {
                AIController->StopMovement();
            }
        }
    }
}
```

Figure 5.32: Enemy Agro Sphere Code for Moving to the Player

5.7.2 Enemy Animation

The enemy animation was done the same way as the player's animation. I started by making an animation blend space comprised of an idle and walking animation, see Figure 5.33, I then used an animation Blueprint to alter the blend space animation based on the enemy's movement speed thus playing the appropriate animation. See Figure 5.34

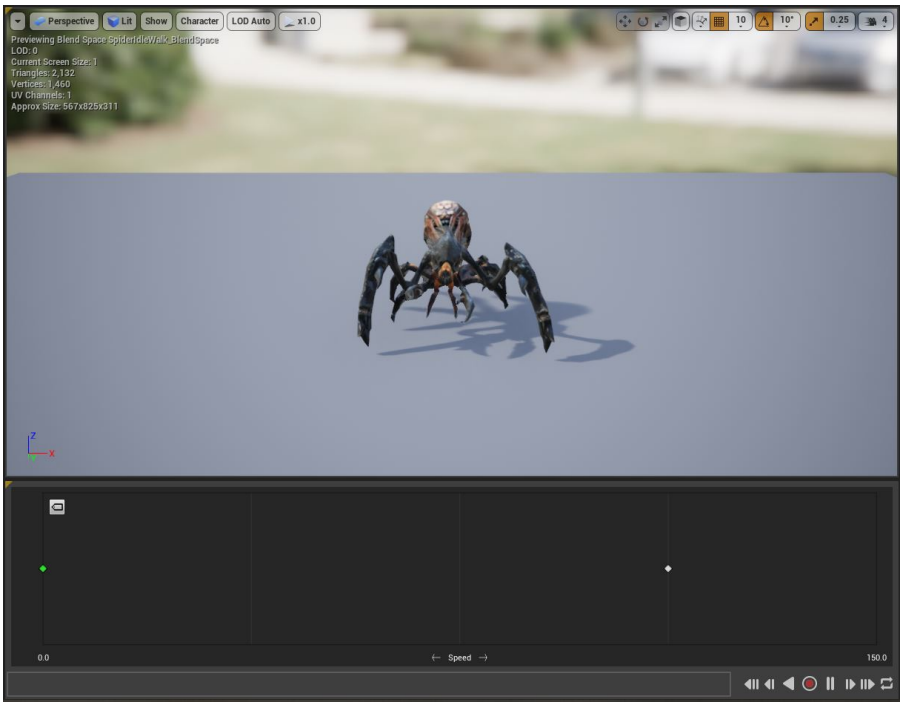

Figure 5.33: Enemy Animation Blendspace

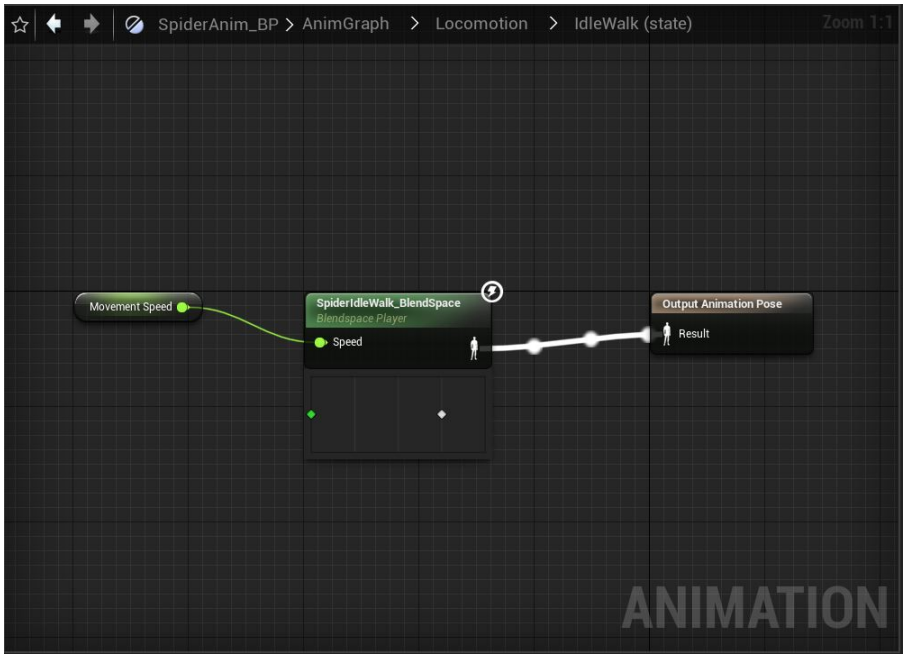

Figure 5.34: Enemy Animation Blueprint Determining Animation

5.7.3 Enemy Combat

The enemy's combat was done in a similar way to the agro/movement. I used a smaller sphere named 'CombatSphere' that was capable of detecting if the player was inside it. See Figure 5.35. If the player was inside the sphere the attack function would be called triggering the attack animation from the enemies animation montage. See Figure 5.36

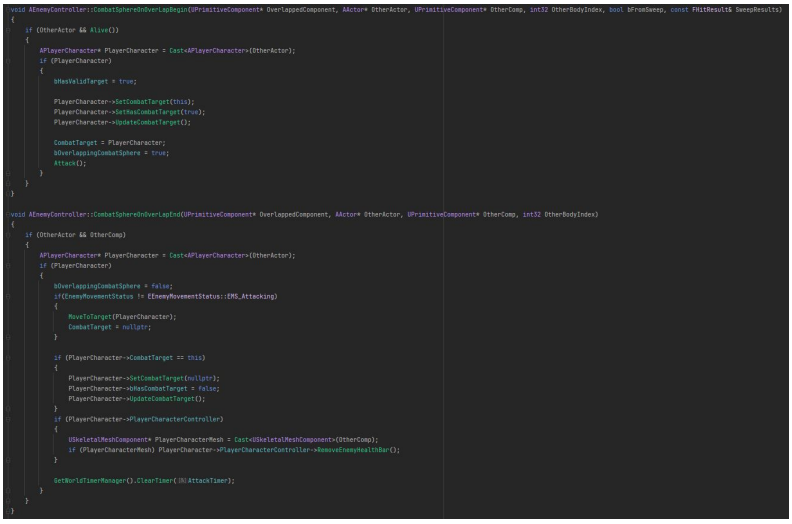

Figure 5.35: Enemy Combat Sphere Scripting

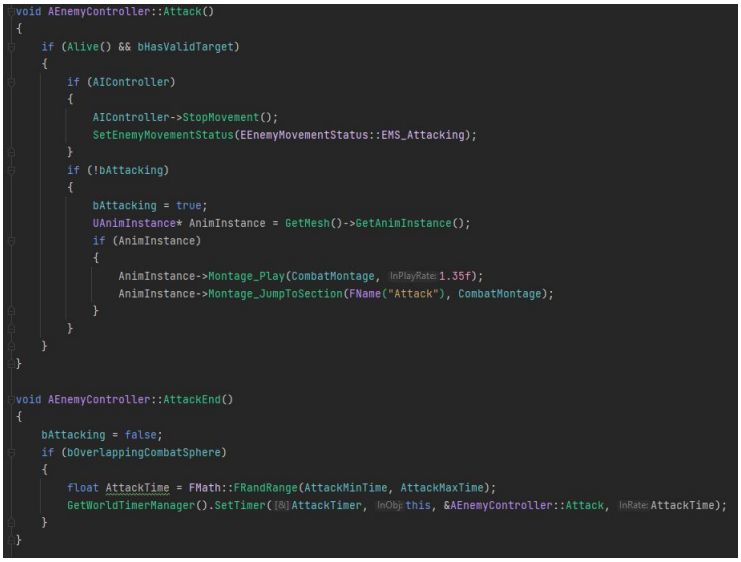

Figure 5.36: Enemy Attack Scripting

5.8 Player Status: Health and Stamina

I started by declaring the player's base stats as variables. See Figure 5.37

```
// Default player stat values
MaxHealth = 100.f;
Health = 65.f;
MaxStamina = 150.f;
Stamina = 120.f;

StaminaDrainRate = 25.f;
MinSprintStamina = 50.f;
```

Figure 5.37: Player Base Stats

Next, I created a HUD for the user's interface which contained their health and stamina bar. See Figure 5.38

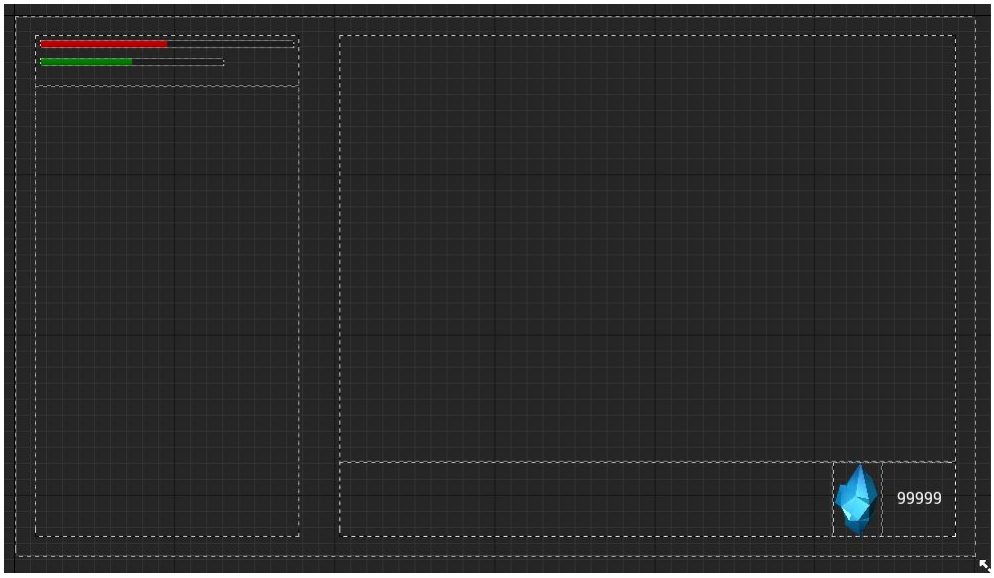

Figure 5.38: Player HUD

I then linked the individual health and stamina bar to their variables within the player's C++ script and altered the state of the bars depending on the variable's values. See Figure 5.39 and Figure **??**

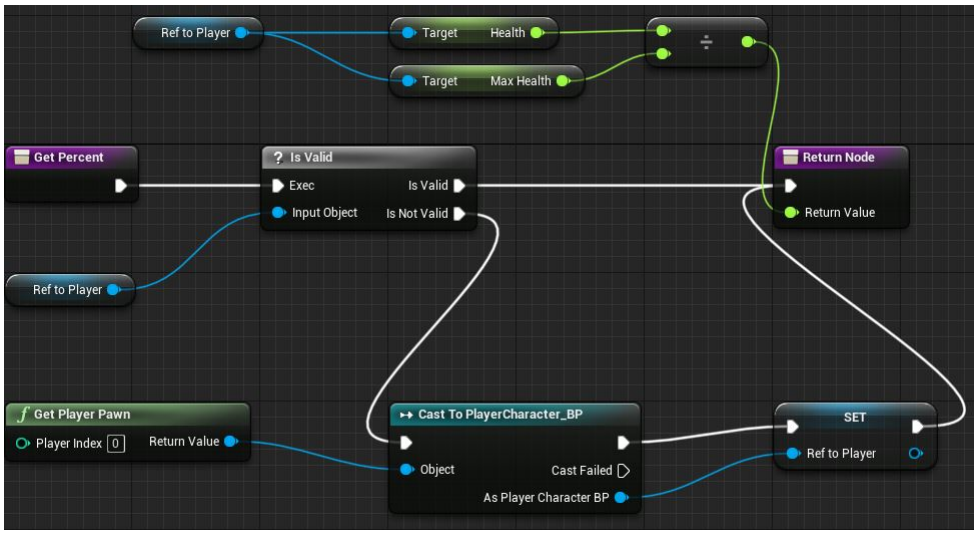

Figure 5.39: Player Health Bar

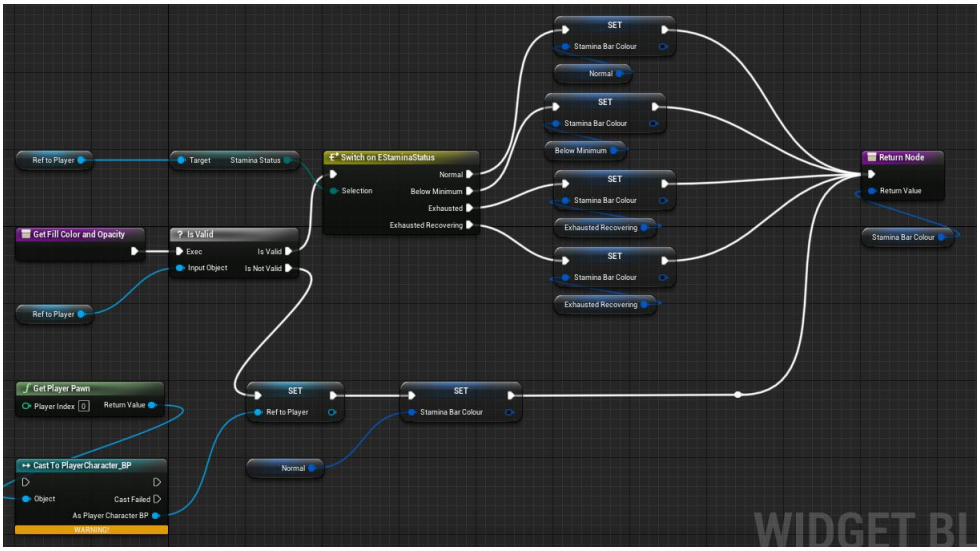

Figure 5.40: Player Stamina Bar

5.9 Dealing Damage

Dealing damage in either the player or enemy's case was easily done. I used collision boxes and the attacking limbs of the skeletons, see Figure 5.41 and Figure 5.42, to detect if they hit an object. Then if the object collided with

the opposing skeleton a function is called to deal damage i.e. remove values from the health pools.

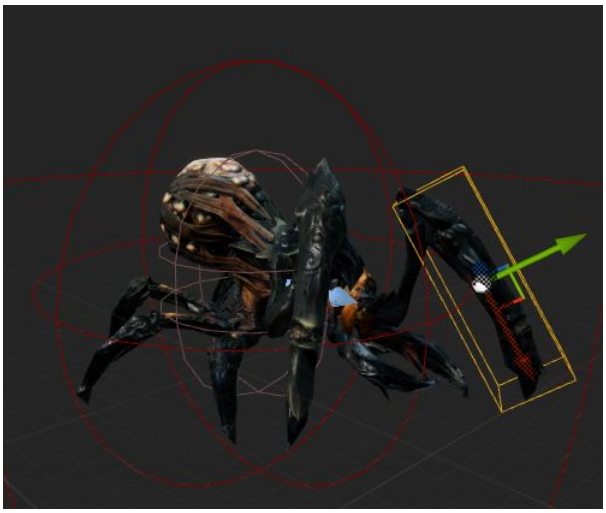

Figure 5.41: Enemy Combat Collision

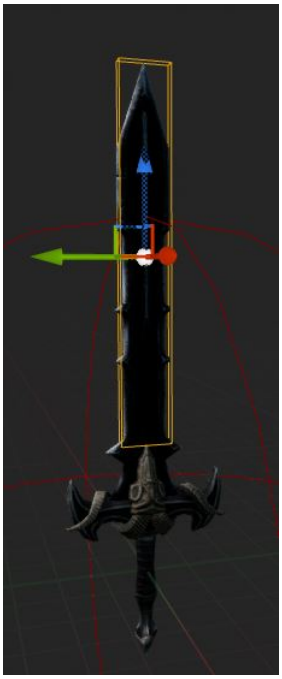

Figure 5.42: Sword Combat Collision

5.10 Enemy Health Bar

The enemy health bar was done in the same way as the player's. However this time instead of being displayed as part of the player's UI it was displayed above the enemy's mesh within the world. See Figure 5.43

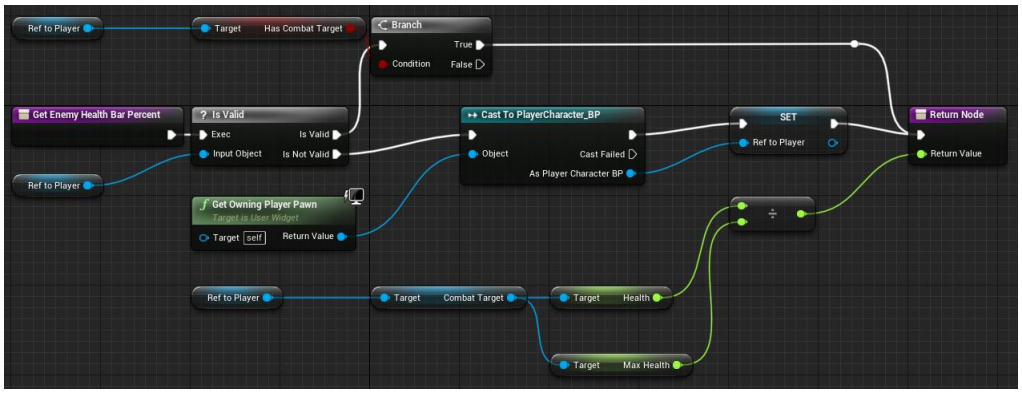

Figure 5.43: Enemy Health Bar

5.11 World Elements: Collectables and Hazards

I decided to add some collectables and hazards to the world. These came in the form of a currency I named 'Shards', health potions and explosive bombs that would explode if the player or enemy collided with them dealing damage.

These items were created in the same way, using a C++ class I created called 'Item'. See Figure 5.44

```
// Sets default values
AItem::AItem()
{
    // Set this actor to call Tick() every frame.  You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    CollisionVolume = CreateDefaultSubobject<USphereComponent>(TEXT("CollisionVolume"));
    RootComponent = CollisionVolume;

    Mesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("Mesh"));
    Mesh->SetupAttachment(InParent: GetRootComponent());

    IdleParticleComponent = CreateDefaultSubobject<UParticleSystemComponent>(TEXT("IdleParticleComponent"));
    IdleParticleComponent->SetupAttachment(InParent: GetRootComponent());

    bRotate = false;
    RotationRate = 45.f;
}
```

Figure 5.44: Item Script

Their individual features were then implemented in their own Blueprint classes. See Figure 5.45 and Figure 5.46

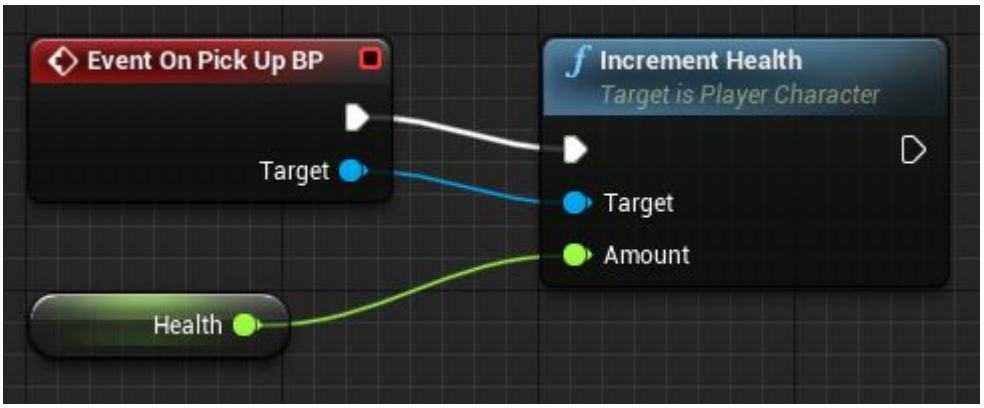

Figure 5.45: Health Potion

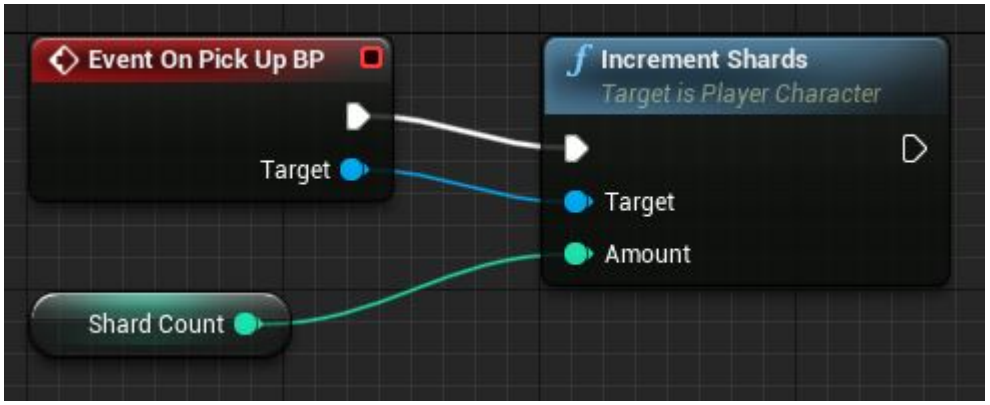

Figure 5.46: Shards

5.12 Save and Load System

The last feature I wanted to implement into my game was the ability to save and load game states. I did this within the player's C++ script. See Figure 5.47

```cpp
void APlayerCharacter::SaveGame()
{
    UBanishmentSaveGame* SaveGameInstance = Cast<UBanishmentSaveGame>(Src: UGameplayStatics::CreateSaveGameObject(UBanishmentSaveGame::StaticClass()));

    SaveGameInstance->CharacterStats.Health = Health;
    SaveGameInstance->CharacterStats.MaxHealth = MaxHealth;
    SaveGameInstance->CharacterStats.Stamina = Stamina;
    SaveGameInstance->CharacterStats.MaxStamina = MaxStamina;
    SaveGameInstance->CharacterStats.Shards = Shards;

    FString MapName = GetWorld()->GetMapName();
    MapName.RemoveFromStart(GetWorld()->StreamingLevelsPrefix);

    SaveGameInstance->CharacterStats.LevelName = MapName;

    if (EquippedWeapon)
    {
        SaveGameInstance->CharacterStats.WeaponName = EquippedWeapon->Name;
    }

    SaveGameInstance->CharacterStats.Location = GetActorLocation();
    SaveGameInstance->CharacterStats.Rotation = GetActorRotation();

    UGameplayStatics::SaveGameToSlot(SaveGameInstance, SaveGameInstance->PlayerName, SaveGameInstance->UserIndex);
}
```

Figure 5.47: Save Game Function

Calling on the save game function would save each aspect of the game into a save game instance. The instance also can save the player's location and rotation and the weapon the player currently had at the point of saving.

Loading the game was a case of reverse engineering the save function. See Figure 5.48

```cpp
void APlayerCharacter::LoadGame(bool SetPosition)
{
    UBanishmentSaveGame* LoadGameInstance = Cast<UBanishmentSaveGame>(Src: UGameplayStatics::CreateSaveGameObject(UBanishmentSaveGame::StaticClass()));

    LoadGameInstance = Cast<UBanishmentSaveGame>(Src: UGameplayStatics::LoadGameFromSlot(LoadGameInstance->PlayerName, LoadGameInstance->UserIndex));

    Health = LoadGameInstance->CharacterStats.Health;
    MaxHealth = LoadGameInstance->CharacterStats.MaxHealth;
    Stamina = LoadGameInstance->CharacterStats.Stamina;
    MaxStamina = LoadGameInstance->CharacterStats.MaxStamina;
    Shards = LoadGameInstance->CharacterStats.Shards;

    if (WeaponStorage)
    {
        AItemStorage* Weapons = GetWorld()->SpawnActor<AItemStorage>(WeaponStorage);

        if (Weapons)
        {
            FString WeaponName = LoadGameInstance->CharacterStats.WeaponName;

            if (WeaponName != TEXT(""))
            {
                if (Weapons->WeaponMap.Contains(WeaponName))
                {
                    AWeapon* WeaponToEquip = GetWorld()->SpawnActor<AWeapon>(Weapons->WeaponMap[WeaponName]);
                    WeaponToEquip->Equip(Char: this);
                }
            }
        }
    }

    if (SetPosition)
    {
        SetActorLocation(LoadGameInstance->CharacterStats.Location);
        SetActorRotation(LoadGameInstance->CharacterStats.Rotation);
    }

    SetMovementStatus(EMovementStatus::EMS_Normal);

    GetMesh()->bPauseAnims = false;
    GetMesh()->bNoSkeletonUpdate = false;

    if (LoadGameInstance->CharacterStats.LevelName != TEXT(""))
    {
        FName LevelName(*LoadGameInstance->CharacterStats.LevelName);

        SwitchLevel(LevelName);
    }
}
```

Figure 5.48: Load Game Function

The load function also can detect which level the player is in, thus avoiding

setting the player's location and rotation if they are on a different map to the one the save was made on. See Figure 5.49

```
void APlayerCharacter::LoadGameNoSwitch()
{
    UBanishmentSaveGame* LoadGameInstance = Cast<UBanishmentSaveGame>(Src: UGameplayStatics::CreateSaveGameObject(UBanishmentSaveGame::StaticClass()));

    LoadGameInstance = Cast<UBanishmentSaveGame>(Src: UGameplayStatics::LoadGameFromSlot(LoadGameInstance->PlayerName, LoadGameInstance->UserIndex));

    Health = LoadGameInstance->CharacterStats.Health;
    MaxHealth = LoadGameInstance->CharacterStats.MaxHealth;
    Stamina = LoadGameInstance->CharacterStats.Stamina;
    MaxStamina = LoadGameInstance->CharacterStats.MaxStamina;
    Shards = LoadGameInstance->CharacterStats.Shards;

    if (WeaponStorage)
    {
        AItemStorage* Weapons = GetWorld()->SpawnActor<AItemStorage>(WeaponStorage);

        if (Weapons)
        {
            FString WeaponName = LoadGameInstance->CharacterStats.WeaponName;

            if (WeaponName != TEXT(""))
            {
                if (Weapons->WeaponMap.Contains(WeaponName))
                {
                    AWeapon* WeaponToEquip = GetWorld()->SpawnActor<AWeapon>(Weapons->WeaponMap[WeaponName]);
                    WeaponToEquip->Equip(Char: this);
                }
            }
        }
    }

    SetMovementStatus(EMovementStatus::EMS_Normal);

    GetMesh()->bPauseAnims = false;
    GetMesh()->bNoSkeletonUpdate = false;
}
```

Figure 5.49: Load Game Function

All of this functionality was then nicely packaged up into a menu system that used buttons to call the save and load functions as well as a quit game feature. See Figure 5.50 and Figure 5.51

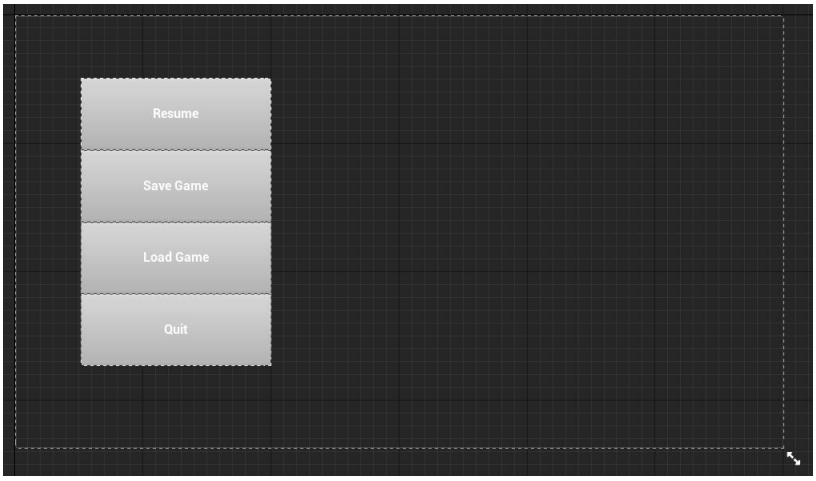

Figure 5.50: Pause Menu

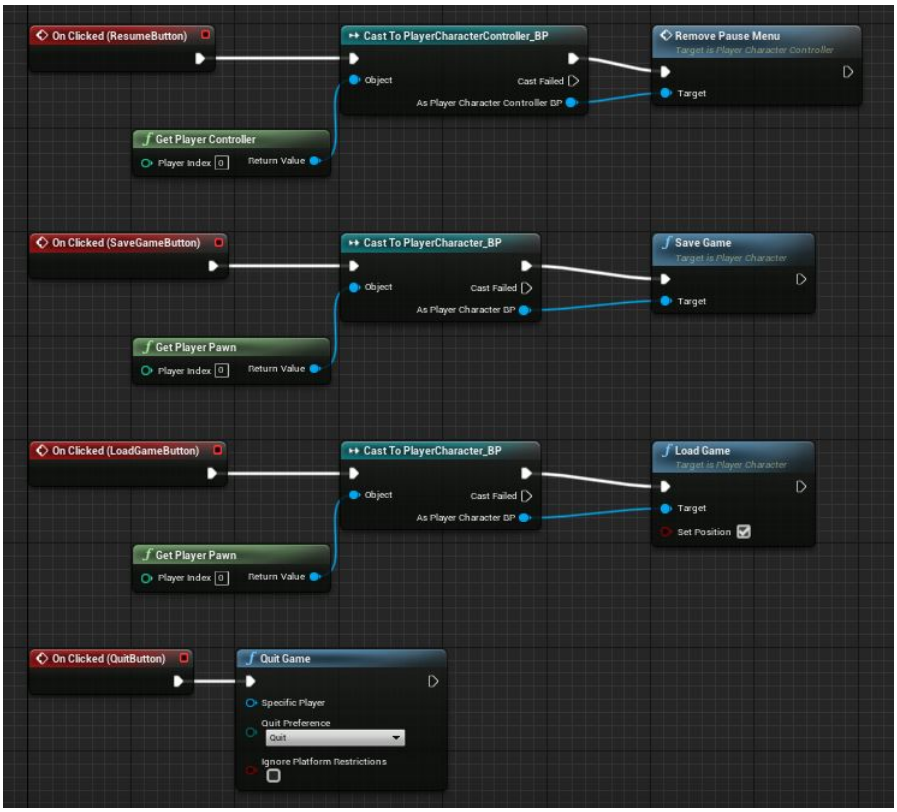

Figure 5.51: Pause Menu Functionality

Chapter 6

System Evaluation

6.1 Testing

A very cool feature of Unreal Engine 4 is the ability to set up automated tests that you can leave running while you do something else. I plan to use these in the future however for my project I decided to do all the testing myself. The main reason for this was to enhance my knowledge of the engine and language and to learn how the two work in tandem.

An element of the project that required a lot of testing and refining was the enemy AI. Being hands-on with the testing allowed me to pinpoint where the issues were arising and helped me to more knowledgeably fix them. Bug-fixing in this way thought me alot about how the C++ language works. Its gave me a new perspective on how to write the language and helped me to enhance and refine my code.

6.2 Limitations and Opportunities

My project, in the grand scheme of things, is quite basic when compared to the likes of the 'Dark Souls' Series by From Software. However, I do believe that my gameplay mechanics are a fantastic framework to be built upon. The combat mechanics in particular have so much room for expansion and refinement. In the future, I would like to build upon them by adding extra features such as more variety in the attacks, the ability to string attacks together in a fluid combo as well as having the enemies get stunned if they are hit with specifically high damage dealing attacks.

One feature of my game I feel performs well but is held back by its limitations is the save game feature. At the moment it is only capable of creating and storing a single save game state. In the future, I would like to

have the ability to have multiple save files stored as persistent data on the local machine, with the ability to call and play them from the main menu.

Overall I am very happy with how my project finished. I firmly believe that it could be used as skeleton code for developers to build upon when undertaking their Action RPG development adventure.

Chapter 7

Conclusion

I set out on this project with a huge personal goal; to improve my game development knowledge and skill set, if feel I have done just that.

I wanted to explore the various game engines on offer to the developer and compare all their pros and cons. Why are there so many game engines out there? What are each engine's areas of strength? Which programming language is the most powerful for the game I wanted to create? There are questions I needed the answers to.

My research suggested that Unreal Engine with its Blueprint feature alongside the C++ programming language was the best fit for what I was trying to achieve with my project.

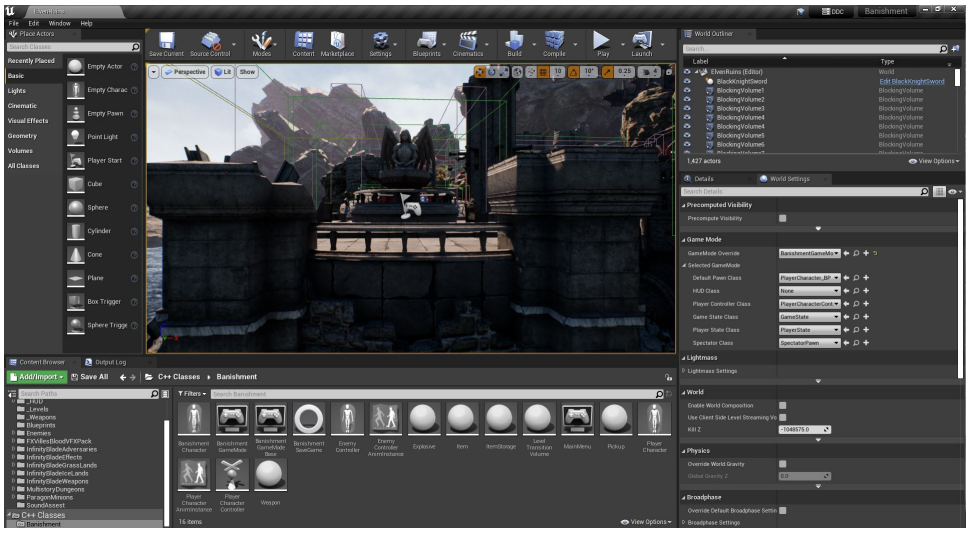

Figure 7.1: Unreal Engine Editor

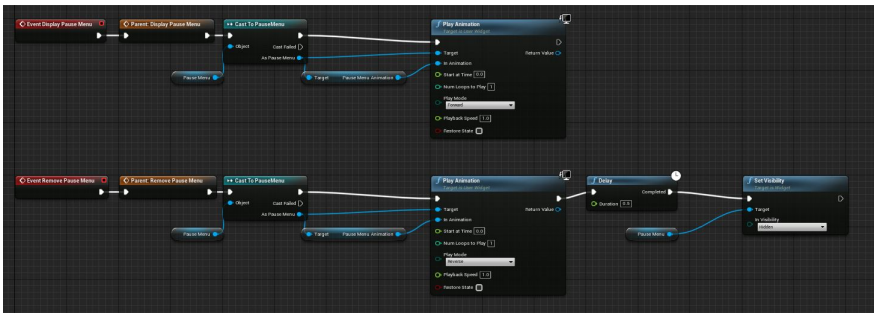

Figure 7.2: Unreal Engine Blueprint

I feel that the game produced from this project is a very strong framework for anyone wishing to develop their own Action RPG game. It provides a solid base for all the necessary player functionality including movement, combat along with a health and stamina system. Also included is intelligent enemy AI that can be used for any type of enemy with little to no tweaking of the code.

If you have ever played a game such as 'Dark Souls' then you know the frustration that comes along with the play-through. The countless deaths and near-death experiences, the bonds formed with the strangest of NPCs and the incredible feeling of finally defeating the boss that you have been stuck on for days! Some players are driven mad when playing these games, I was driven mad enough to create my own.

Figure 7.3: If you know, you know

Bibliography

[1] D. S. Nanadath, “Game development using c++,” 2014.

[2] “Unreal engine 4 documentation.”

[3] “Monday.com.”

[4] A. Andrade, “Game engines: a survey,” pp. 1–6, 2015.