
# Archimedes' Maze

Project Engineering

Year 4

Josua Brink

Bachelor of Engineering (Honours) in Software and Electronic Engineering

Galway-Mayo Institute of Technology

2021/2022

# DECLARATION

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Galway-Mayo Institute of Technology.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

*Josua Brink*

# TABLE OF CONTENTS

# 1 SUMMARY

For my final year project, I've designed a game using the Unreal Engine 4. The reason for using the Unreal Engine was because of its use of C++ and that it allows for more detailed memory management.

To manage my project, I used several different tools to create a timeline and manage different versions of my game. Jira was used as my general planning tool which helped me keep track of different issues and manage timing with a detailed timeline generated. And Git was my primary version control application where I kept my code repository and pushed all my changes to.

My project contained several classes which I created myself with custom functionality written in C++ which includes:

- **Player Character** – The pawn which the player controls and uses to interact with the world around them
- **Projectile** – A projectile object which is thrown by the player when they press the left mouse button
- **Path Point** – These are created when the player is aiming by holding down the left mouse button before releasing. They indicate where the projectile will travel.
- **Big Red Button** – When this class is interacted with, it opens the entrance to the next stage of the level.
- **Entrance** – The separator between stages of the level. This is opened when the Big Red Button class is hit by the projectile.
- **Pressure Plate** – This class handles closing of the doors behind the player once the player passes through.
- **Character HUD** – This is the main UI class that's attached to the player character. It holds functionality related to the GUI and pause menu.
- **Moving Platforms** – These are obstacles which move along a sin wave pattern.
- **My Save Game** – This class serves as a template for creating save files. It contains variables which will be saved to this file.

There are also additional features which I added to this game. This includes several blueprint classes and a plugin which I installed to manage gravity directions.

- **Pause Menu** – This is a blueprint class create from the UMG Ui designer. It handles saving, loading, and quitting the game.
- **Main Menu** – This is another class created from the UI designer. This gets opened when the game first starts and has options to continue from a saved game, to start a new game, or to quit out of the game.
- **Directional Gravity** – This is the plugin I installed to allow me to change the gravity vector in the game engine which lets me set the direction the gravity points in.

# 2 POSTER


GMIT
INSTITIÚID TEICNEOLAÍOCHTA NA GAILLIMHE MAIGH EO
GALWAY-MAYO INSTITUTE OF TECHNOLOGY
ARCHIMEDES' MAZE
JOSUA BRINK
BENG (HONOURS) IN SOFTWARE AND ELECTRONIC ENGINEERING
Project Description
I have always had a passion for gaming and everything that goes into developing a game. With this in mind I've decided to base my final year project on developing a game using the Unreal Engine 4. The goal of this project is to develop the groundwork on which a full game can be created with reusable functions and classes.
Game Environment
Unreal Blueprints
Custom C++ Code
Unreal Engine
C++ Compiler
Operating System
Unreal Engine
A game engine is basically a software framework specialised in creating video games which include libraries and tools relevant for this purpose along with a 3D editor to place assets into the game world.
The reason I decided on using the unreal engine for my project was that it's based on C++. This allows developers to work more closely with the hardware that the game's running on, and allows you to manage aspects such as memory much more closely.
I also wanted to learn more about developing software using modern C++ and the unreal engine gives plenty of opportunity to delve deeper into this with its extensive use of classes and object orientated coding style.
Class Diagram
Github Page
Software & Languages
git
Jira
blender
UNREAL ENGINE
Throughout development I used many different tools and software to develop this project.
For project planning, I used Jira to develop a timeline of features that need to be delivered. This worked alongside Git which handled version control and held the code repository.
The project itself was built around the unreal engine, which acted as a framework for the project and provided a 3D editor for the game environment. I also used blender to develop some of my own custom assets for the game.
And finally C++ was used extensively to implement features into the game.
Conclusion & Results
In conclusion, I've developed a solid base from which I could develop a full game out of. I've added many features which I can reuse throughout developing a full game out this project.
The result of this project is an environment the player can interact with and explore. There's full player input and a projectile system. I've also added in a main menu and a pause screen which allows the player to save and load the game, and there are several components I've created which the player can interact with.

# 3 INTRODUCTION

I have always had a passion for gaming and everything that goes into developing a game. Everything from setting up the players input to creating custom 3D models to adding functionality to those 3D models using code, has all interested me.

With all this in mind I've decided to base my final year project on developing a game using the Unreal Engine 4. A game engine acts as a sort of framework for the game which includes functions and tools specialised in developing custom features.

The Unreal Engine is based on C++ and includes many features to smooth the process of game development. This includes a in game editor to design the in-game world. This is where you can place the 3D models which you've written code for.

Along with the editor comes many libraries and functions that can be used in C++. These are used to customise the classes to do pretty much anything in your game.

The goal of this project is to develop the groundwork on which a full game can be created with many of the functions and classes being able to be reused throughout the development process.

I have based the concept of the game I'm developing for this project on other physics-based puzzle games like the "Portal" game series and "Outer Wilds".

These games take an interesting approach to solving puzzles by involving the environment in the solution to solve those puzzles. This includes situations where the simulated gravity might be shifted, or you would need to use momentum gained from the gravity to traverse platforms.

For my own project I was inspired to attempt to create similar features.

# 4 THE UNREAL ENGINE

As mentioned previously, the unreal engine acts as a sort of framework for the game, with tools and libraries specialised in developing gameplay features, which also includes a 3D editor for configuring the game world.

## 4.1 THE UNREAL EDITOR

The game engine comes with an editor which is used to place the different components that are part of the engine or custom made, into the game world. This is configured using an X, Y and Z coordinate system to track where in the game world a component or mesh is.

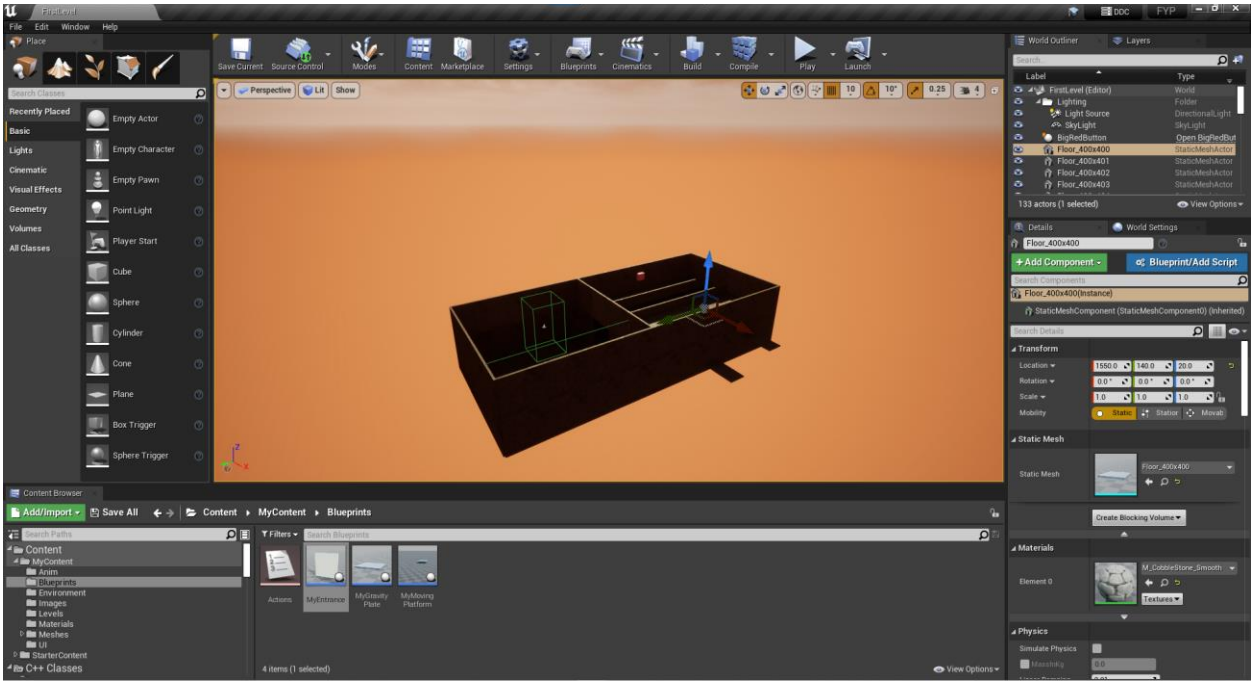


*Figure 1 - Unreal Editor*

The editor comes with a built-in content browser so that you can easily access assets to place in the game world and to configure them in either C++ or using the Unreal blueprints. The placed components can be seen on the right in the "World Outliner" panel. This lists out everything that's currently placed in the game.

Underneath that you can see the properties of the currently selected component. These can be defined using UProperties in your C++ files.

The game can be run using the play button on the top bar. This is used for debugging and testing the various features added.

## 4.2 Unreal Class Structure

When creating a new class using the unreal engine libraries, several functions are implemented, and the new class extends an abstract class in these libraries.

When creating a new class, 2 functions get implemented. The Tick function, and the BeginPlay function.

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "PressurePlate.generated.h"

UCLASS()
class FYP_API APressurePlate : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    APressurePlate();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

};
```

*Figure 2- Unreal Engine Class Structure*

When the game is running, every frame the Tick function gets called. This allows the class to execute its code in real time as the game runs. DeltaTime, the variable seen in the tick function refers to the time in between frames. This can be useful for timing-based applications.

The BeginPlay function acts similarly to the class's constructor, however it only gets called after the actor gets spawned in the game world, while the constructer gets called much earlier in its lifecycle.

The header guards standardly used for the Unreal Engine are "#pragma once". The unreal engine isn't compatible with the other form of header guard used "#ifndef"

And the last thing to note is the order in which the #includes are added. The generated.h file must always be included last in the order. Everything else must be added before it.

## 4.3 Garbage Collection

Because the unreal engine comes with its own garbage collector, using destructors isn't as necessary when designing classes in C++.

When an actors destroy function is called, or its deleted, it's added to a queue for the garbage collector. Then every 30-60 seconds [1] the garbage collector removes any references of the deleted objects from the current game session.

This then frees up whatever memory that object was using before being removed.

# 5 UNREAL BLUEPRINTS

Unreal Blueprints are a special type of object in the game engine which can be created directly in the editor or from existing C++ classes. It serves as a "visual scripting tool" [2] which calls functions and classes from withing the game engine.

It can be very useful for high level features such as for the User Interface and using animations.

## 5.1 EVENT CHART

Most of the scripting in blueprints is handled in the event graph within the class. It uses the same kind of logical statements as more conventional programming does, however it's done in a more visual flowchart style

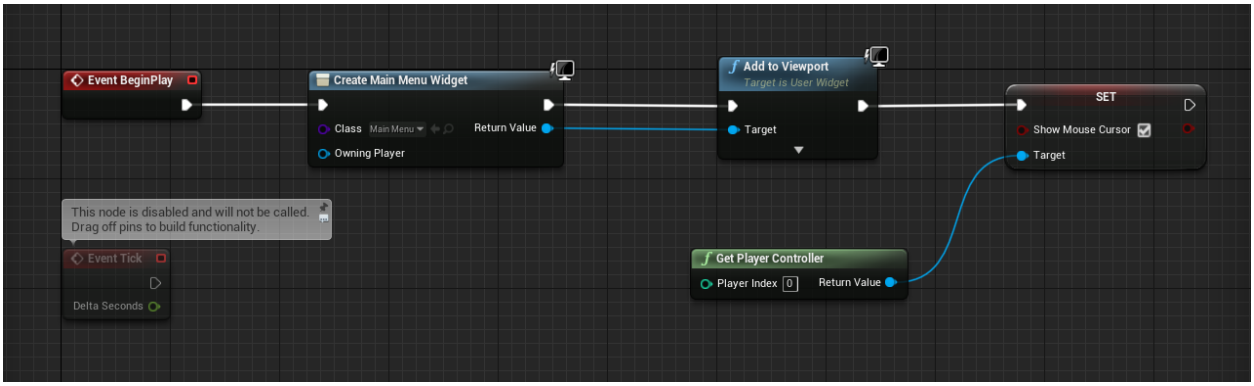



*Figure 3 - Event Graph*

When created from a C++ class, the blueprint still has access to all its functions, due to it extending the base class. This allows you to seamlessly integrate your C++ code into custom blueprints.

There are several advantages to creating blueprints out of C++ classes instead of directly using C++, one of which are the properties in each instance that can be changed simply through the component properties menu. This allows for these blueprint classes to affect different objects based on properties set. E.g. Passing an instance of Entrance to an instance of Button.

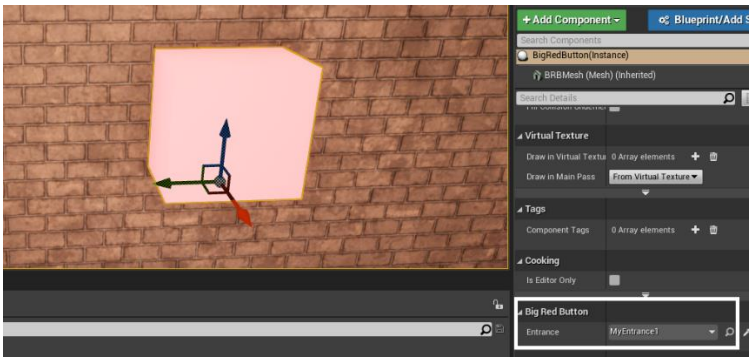



*Figure 4 - Button being passed specific entrance*

The downside however is that it can be rather messy with more complex logic and the performance cost is much larger compared to using C++. With all this in mind it can be situationally useful but C++ in a lot of cases gives better performance.

## 5.2 UPROPERTIES

UProperties are variables defined in C++ that are visible or editable in the Unreal Editor. This allows you to pass functionality you define in C++ directly to a blueprint class you create from the C++ version.

It allows for a lot of flexibility in how these classes are designed by allowing you to handle most of the logic in code while setting specific properties through the editor or blueprints.

A very useful way of using these properties is by a UProperty as a pointer to another object, and then setting that pointer through the editor, essentially connecting the two instances together. E.g. Figure 4.

To create these UProperties, you use the UPROPERTY call and define where these are visible/editable. And then you simply declare the variable like you would any other.

```
public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    UPROPERTY(VisibleDefaultsOnly, Category = Movement)
        UMaterialInstanceDynamic* BRBMaterial;

    UPROPERTY(VisibleAnywhere)
        UStaticMeshComponent* BRBMesh;

    UPROPERTY(EditAnywhere)
        AEntrance* entrance;
```

*Figure 5 - Uproperty Declaration*

In this case, it's editable anywhere so that I can set it in the editor later.

## 5.3 UMG UI Designer

The UMG UI Designer serves as a GUI designer similar to the Swing Designer used in IntelliJ for Java.

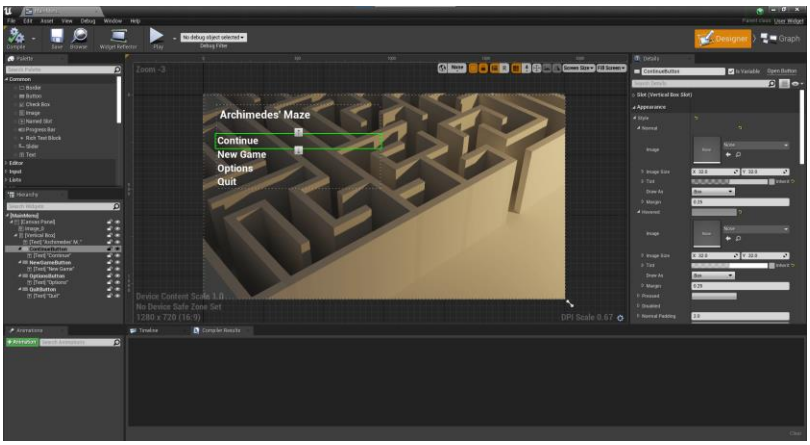



*Figure 6 - UI Editor*

GUI Components can be added from the Palette section and then customised through its properties. They come with several event calls which can be defined through blueprints or called through C++.


*Figure 7 - UI Events*

Any functionality defined in the On Clicked handler will be called once the button is pressed.

# 6 PROJECT ARCHITECTURE

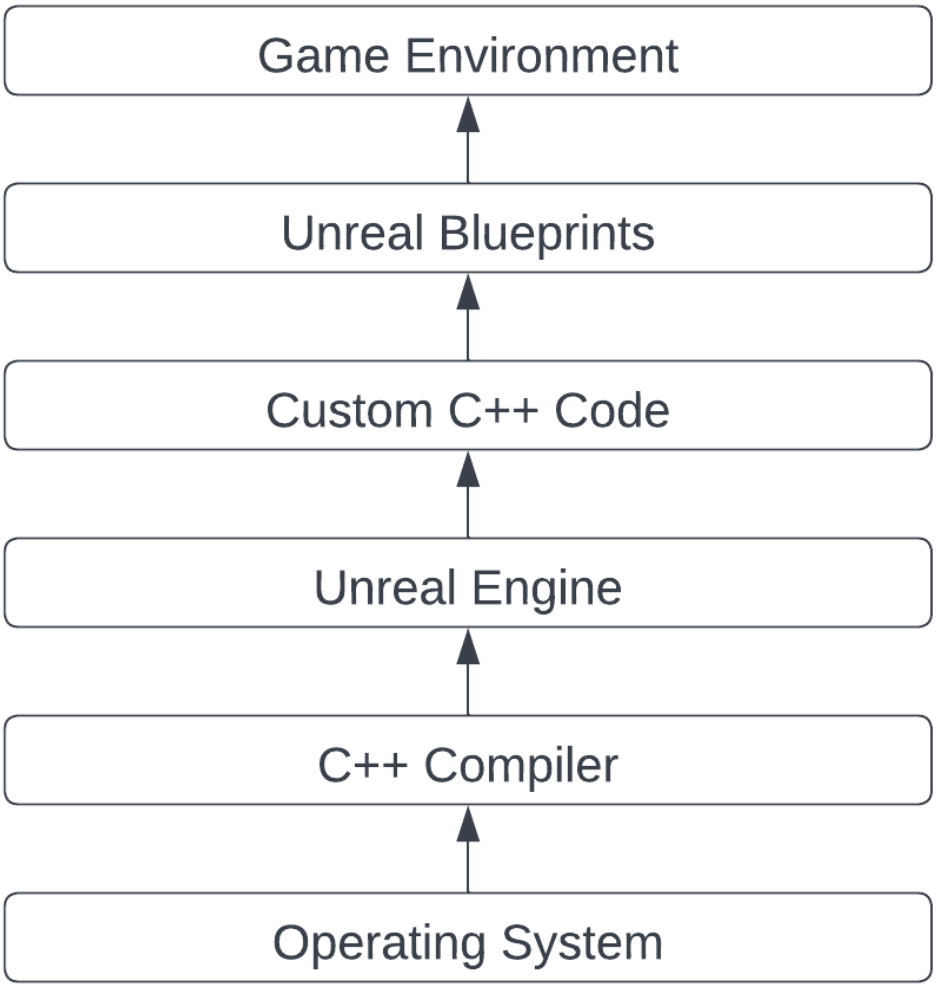



*Figure 8 - Project Architecture*

# 7 PROJECT PLAN

## 7.1 INITIAL TIMELINE


*Figure 9- Initial Predicted Timeline*

For my initial prediction, I had to make a lot of assumptions as I had no experience developing games and didn't really have much knowledge about what's involved in the game development process. It became clear very quickly that I wasn't going to be able to follow to the initial timeline closely.

My initial idea was to approach the project as a whole, developing each section to completion before moving on to the next. As I got further in development it was becoming increasingly clear that I had to re-evaluate my approach to game development.

## 7.2 Using Jira

Instead of dealing with each aspect of the game such as environmental code and player controls, I was instead working on features which I planned on adding to the game which dealt with many of these aspects at the same time.

This led me to the project development tool I used for the remainder of this project. Jira. It made planning out my next step much simpler by separating them into individual epics. These epics can then have sub-sections for tasks that need to be done within that feature

It also allowed me to keep track of the projects timeline more clearly by giving me a roadmap with dates for individual task completion.


*Figure 10- Jira Roadmap*

Jira also has a plugin which integrates the GitHub repository for this project into the Jira project. This allows it to keep track of commits and branches I push using Git as long as I add the task number to the branch name or commit message.

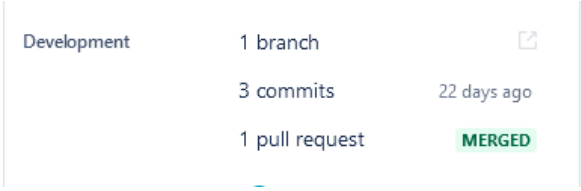



*Figure 11- Jira tracking commits*

## 7.3 GitHub Version Control

To manage my projects versions, I used GitHub. This is a commonly used and convenient way to manage different versions of any project. GitHub also serves as a storage location for all code related to this project in the master branch.

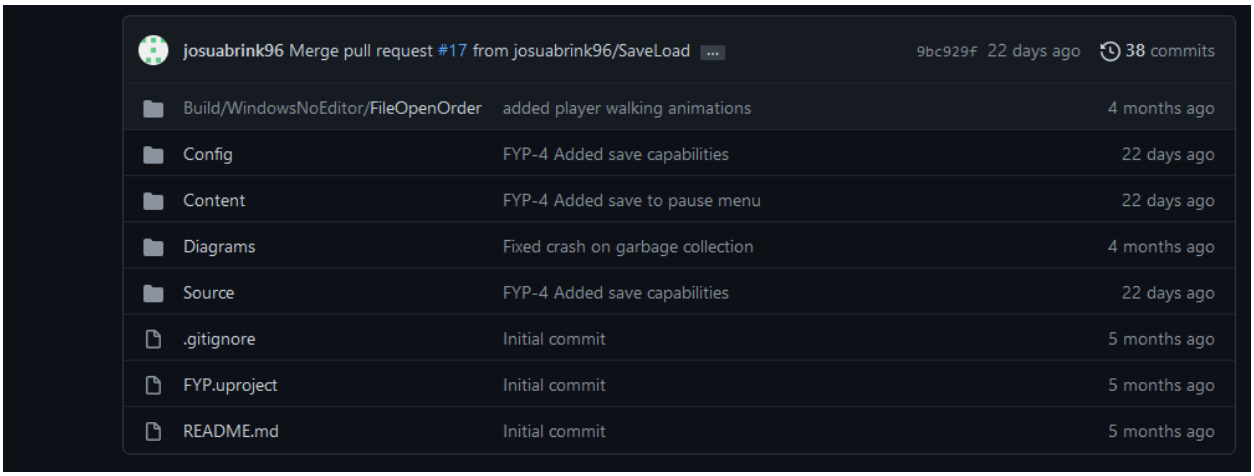



*Figure 12- GitHub Repository*

Any new additions which are pushed from the git bash terminal will be added to a separate branch that can be compared to master through the use of a pull request which checks if it can be merged into the master branch.


*Figure 13- Example Pull Request*

As I mentioned previously, this is integrated into Jira through a plugin so that whenever I push any work I've done to a branch, Jira is updated to reflect these changes that have been submitted.

# 8 C++ CLASS DESCRIPTION

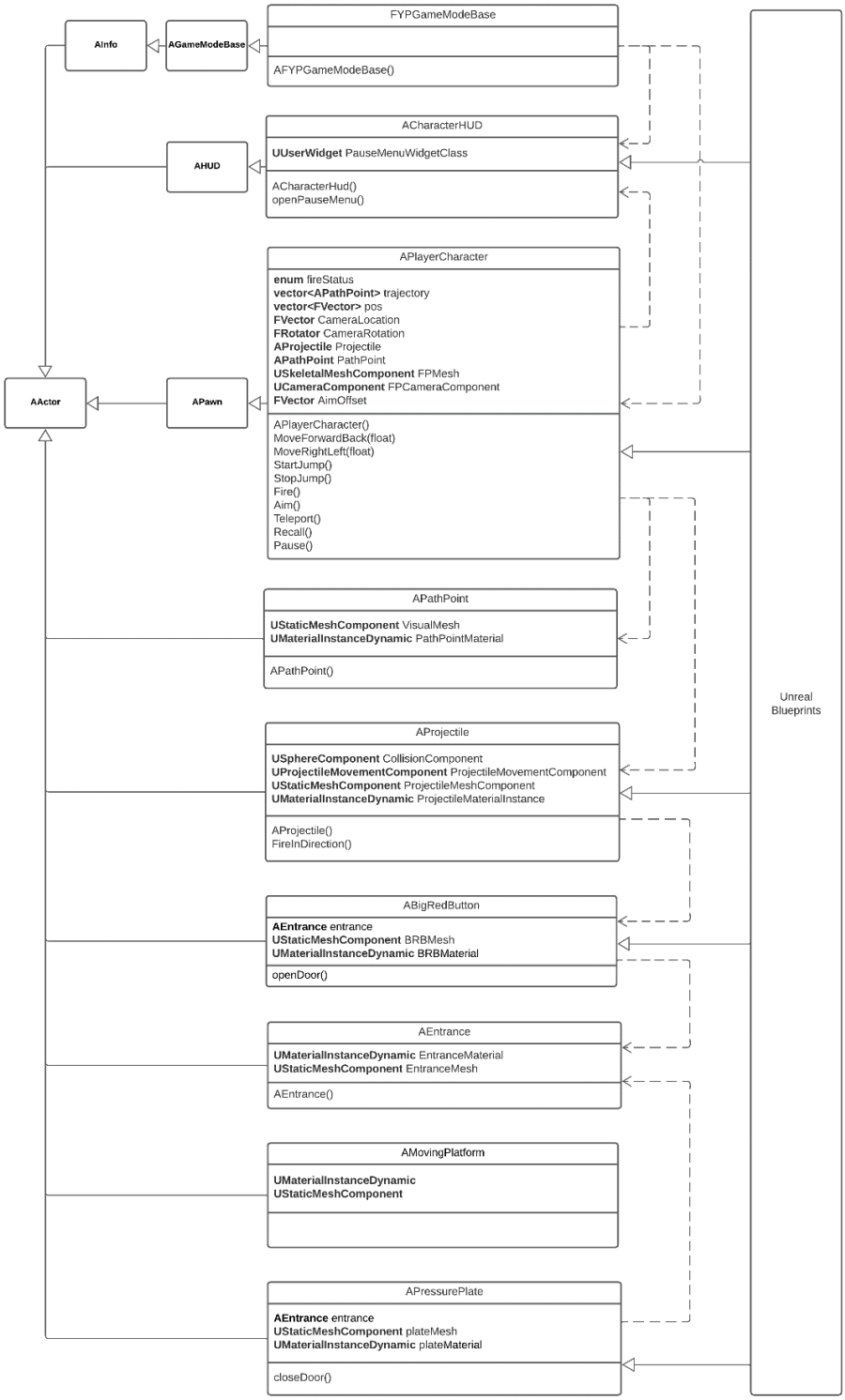



*Figure 14- Class diagram*

## 8.1 PLAYER CHARCTER

Most of the interactions that happen in the game are caused by the player character. This is the character in the game world that the player directly controls and uses to interact with the world around them.

However, before being able to do anything, the camera and inputs need to be set up. These configurations define the controls of the game and how the player can interact with other objects. [3]

### 8.1.1 Configuring the Camera and Mesh

The camera acts as the players perspective in the game. Whatever the camera component sees, the player can see.

Firstly, the camera component needs to be created using the "CreateDefaultSubobject" function call to create an instance of the "UCameraComponent" class. This is then attached to the player characters capsule component using the "SetupAttachment" function. [4]

This will allow the camera to follow the player. The capsule is simply the pill shaped area around the player characters mesh.

```
FPCameraComponent = CreateDefaultSubobject<UCameraComponent>(TEXT("FirstPersonCamera"));
check(FPCameraComponent != nullptr);
FPCameraComponent->SetupAttachment(CastChecked<USceneComponent, UCapsuleComponent>(GetCapsuleComponent()));
FPCameraComponent->SetRelativeLocation(FVector(0.0f, 0.0f, 50.0f + BaseEyeHeight));
FPCameraComponent->bUsePawnControlRotation = true;
```

*Figure 15 - Configuring Camera*

Then to allow the player to control the cameras rotation using the X axis that will be set up in the input configuration section, the Boolean value "bUsePawnControlRotation" needs to be set to true.

With the camera attached to the capsule component, the next thing I had to do was create a "USkeletalMeshComponent". This is an instance of the 3D mesh for the "PlayerCharacter" class. In this case I'm setting the first-person mesh as visible for the player while the 3rd person mesh is set to not visible. This will give the player a first-person perspective in the game.

```
FPMesh = CreateDefaultSubobject<USkeletalMeshComponent>(TEXT("FirstPersonMesh"));
check(FPMesh != nullptr);
FPMesh->SetOnlyOwnerSee(true);
FPMesh->SetupAttachment(FPCameraComponent);
FPMesh->bCastDynamicShadow = false;
FPMesh->CastShadow = false;
GetMesh()->SetOwnerNoSee(true);
```

*Figure 16 - Setting the Player Mesh*

### 8.1.2 Input Configuration

To configure inputs for the player, 2 areas of the project need to be changed. Firstly, I had to edit the project settings in the unreal editor, binding the key to an axis or function.

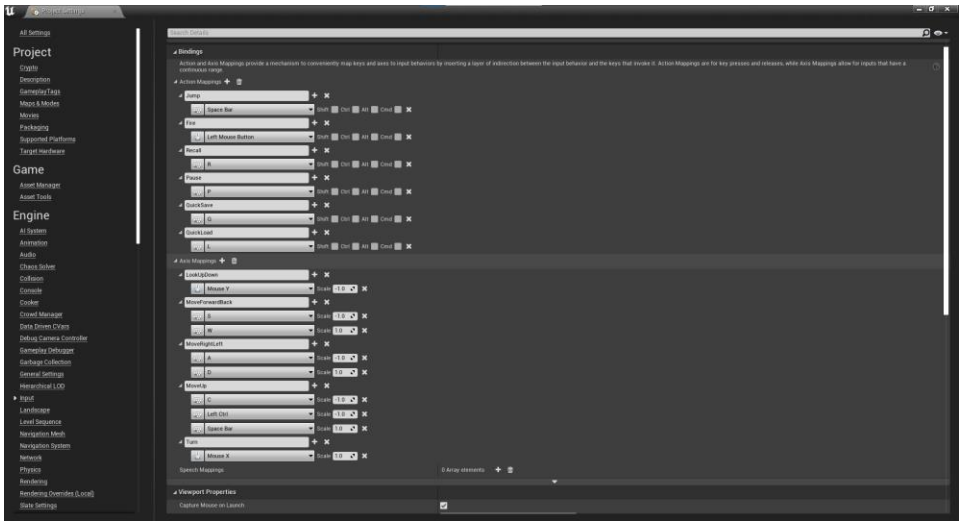


*Figure 17 - Project Setup Input Config*

This sets the corresponding Action/Axis mapping in the DefaultInputs.ini file, which handles key bindings and input settings for the player.

```
ActionMappings=(ActionName="Jump",bShift=False,bCtrl=False,bAlt=False,bCmd=False,Key=SpaceBar)
ActionMappings=(ActionName="Fire",bShift=False,bCtrl=False,bAlt=False,bCmd=False,Key=LeftMouseButton)
ActionMappings=(ActionName="Recall",bShift=False,bCtrl=False,bAlt=False,bCmd=False,Key=R)
ActionMappings=(ActionName="Pause",bShift=False,bCtrl=False,bAlt=False,bCmd=False,Key=P)
ActionMappings=(ActionName="QuickSave",bShift=False,bCtrl=False,bAlt=False,bCmd=False,Key=Q)
ActionMappings=(ActionName="QuickLoad",bShift=False,bCtrl=False,bAlt=False,bCmd=False,Key=L)
```

*Figure 18 - DefaultInputs.ini*

Now that the keys are mapped to an action name, they can be found when the "SetupPlayerInputComponent" is called once the character is created. Then I simply called the "BindAction" function from the "PlayerInputComponent" and connected the signal from the input to an axis or a function I defined in the PlayerCharacter class.

```
// Called to bind functionality to input
void APlayerCharacter::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
{
	Super::SetupPlayerInputComponent(PlayerInputComponent);
	PlayerInputComponent->BindAxis("MoveForwardBack", this, &APlayerCharacter::MoveForwardBack);
	PlayerInputComponent->BindAxis("MoveRightLeft", this, &APlayerCharacter::MoveRightLeft);

	PlayerInputComponent->BindAxis("Turn", this, &APlayerCharacter::AddControllerYawInput);
	PlayerInputComponent->BindAxis("LookUpDown", this, &APlayerCharacter::AddControllerPitchInput);

	PlayerInputComponent->BindAction("Jump", IE_Pressed, this, &APlayerCharacter::StartJump);
	PlayerInputComponent->BindAction("Jump", IE_Released, this, &APlayerCharacter::StopJump);

	PlayerInputComponent->BindAction("Fire", IE_Pressed, this, &APlayerCharacter::Aim);
	PlayerInputComponent->BindAction("Fire", IE_Released, this, &APlayerCharacter::Fire);

	PlayerInputComponent->BindAction("TestRotate", IE_Pressed, this, &APlayerCharacter::TestRotate);

	PlayerInputComponent->BindAction("Recall", IE_Pressed, this, &APlayerCharacter::Recall);

	PlayerInputComponent->BindAction("Pause", IE_Pressed, this, &APlayerCharacter::Pause);

	PlayerInputComponent->BindAction("QuickSave", IE_Pressed, this, &APlayerCharacter::QuickSave);
	PlayerInputComponent->BindAction("QuickLoad", IE_Pressed, this, &APlayerCharacter::QuickLoad);
}
```

*Figure 19 - SetupPlayerInputComponent Function*

### 8.1.3 Firing System

The firing system in my project started off as a simply projectile the character throws. I bound the left mouse button to a function which spawns the "Projectile" in front of the player.

Then I set its launch direction, i.e., the direction the player is facing, and then call the projectiles function to fire in that direction. [5]

To spawn the actor, you simply need to give it the object you wish to spawn, the location where it will spawn and the rotation of the actor once its spawned.

And finally, the spawn parameters. The parameters hold the owner of the object being spawned, in this case the player character and the instigator for OnHit events.

```
if (World)
{
        FActorSpawnParameters SpawnParams;
        SpawnParams.Owner = this;
        SpawnParams.Instigator = GetInstigator();

        // Spawn the projectile at the muzzle.
        Projectile = World->SpawnActor<AProjectile>(ProjectileClass, AimLocation, AimRotation, SpawnParams);
        if (Projectile)
        {
                // Set the projectile's initial trajectory.
                FVector LaunchDirection = AimRotation.Vector();
                Projectile->FireInDirection(LaunchDirection);
        }
}
```

*Figure 20 - Spawning the Projectile*

However, I wanted the player to only be allowed to throw 1 projectile at a time and before being allowed to throw another one, they'd need to recall that projectile. This is where the idea of using a state machine occurred to me.

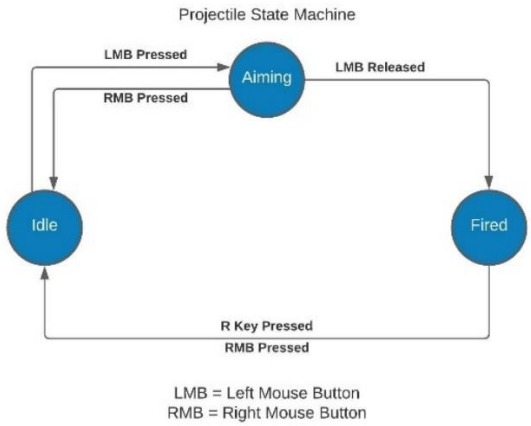



*Figure 21 - Firing System State Machine*

Instead of using countless if else statements to check Boolean values for each state, I instead defined an Enum which can be 1 of 3 values. "Idle", "Aiming", or "Fired". This corresponds to the mouse being clicked, held, and then released.

The release of the left mouse button was bound to a separate function call to handle the fired event.

```
enum fireStatus {
        Idle,
        Aiming,
        Fired,
};
```

*Figure 22 - Fire state Enum*

Once the left mouse button is clicked, the Aim function gets called. This sets the Enum value to Aiming and fills a vector with a series of trajectory points which are spawned into the game world.

```
void APlayerCharacter::Aim() {
    if (status == Idle) {
        status = Aiming;
        GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Red, TEXT("Aiming..."));

        UWorld* World = GetWorld();
        if (World) {
            for (auto& p : pos) {
                PathPoint = World->SpawnActor<APathPoint>(p, CameraRotation);
                trajectory.Add(PathPoint);
            }
        }
    }
```

*Figure 23 - Aim function*

Now that the status is set to Aiming, a trajectory line is spawned into the game to show the path the projectile will take. However, its position will need to be corrected each in game tick so with that in mind, I have check for status in the Tick function of this class to handle the positioning of these path points.

```
void APlayerCharacter::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    if (status == Aiming) {
        GetActorEyesViewPoint(CameraLocation, CameraRotation);
```

*Figure 24 - Aiming state*

Just like before, I needed to get the initial position of these path points using the camera rotation and position to calculate where the initial spawn location is and then applying an offset to it to correct where in the characters perspective it should show up.

```
FPredictProjectilePathParams params;
FPredictProjectilePathResult result;

AimOffset.Set(100.0f, 100.0f, 0.0f);
FVector AimLocation = CameraLocation + FTransform(CameraRotation).TransformVector(AimOffset);
FRotator AimRotation = CameraRotation;
AimRotation.Pitch += 10.0f;

params.StartLocation = AimLocation;
params.LaunchVelocity = AimRotation.Vector() * 2000.0f;
params.MaxSimTime = 2.0f;
```

*Figure 25 - Path point position*

With the initial position calculated, I'm calling the "PredictProjectilePath" function from the "UGameplayStatics" library in the Unreal Engine. This function basically simulates a projectile being thrown along the same path and then plots points along the arch which are then stored in a Vector. [6]

I am then iterating over the results of this function call and adding it to a vector holding coordinates. Then to correct the position of the trajectory path points, I'm iterating over the vector of my trajectory, which holds all the instances of my path point objects and setting its location to that position along the arch I just calculated.

```
UWorld* World = GetWorld();
if (World)
{
        UGameplayStatics::PredictProjectilePath(World, params, result);
        pos.Empty();
        for (const auto& p : result.PathData) {
                pos.Add(p.Location);
        }
        int i = 0;
        for (const auto& t : trajectory) {
                t->SetActorLocation(pos[i]);
                i++;
        }
}
```

*Figure 26 - Setting trajectory position*

This then results in a trajectory line that changes its angle based on where the player is looking, and it follows the players perspective.

Then on release of the left mouse button, the fire function is called, which spawns the projectile at the initial spawn location I calculated earlier and fires it in the direction the player is looking. [Figure 20 - Spawning the Projectile].

At this point the state machine is locked to the "Fired" state until the player recalls the projectile, which I've bound to the R key. Once that's pressed, the recall function gets called, the status is reset back to the "Idle" state, and the projectile is set to be destroyed.

```
if (status == Fired) {
        status = Idle;
        GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Red, TEXT("Ball recalled."));
        Projectile->Destroy();
}
```

*Figure 27 - Recall function*

### 8.1.4 Saving/Loading game

Saving and loading a game refers to storing variables and data you wish to persist between sessions and then retrieving it later and setting your local variables and properties to them, essentially storing the current state of your game, and then retrieving it later.

To handle saving and loading I've bound the Q key to the save function and the L key to the load function. [7]

To save a game, I needed to create an instance of a save game object [My Save Game]. Then I simply set the variables to the value I wish to save; in this case I'm just storing the players location and then called the "SaveGameToSlot" function from "UGameplayStatics" which creates a save game file with the stored variables. [8]

```
void APlayerCharacter::QuickSave() {
	UMySaveGame* SaveGameInstance = Cast<UMySaveGame>(UGameplayStatics::CreateSaveGameObject(UMySaveGame::StaticClass()));
	SaveGameInstance->PlayerLocation = this->GetActorLocation();
	UGameplayStatics::SaveGameToSlot(SaveGameInstance, TEXT("Quick Save"), 0);
	GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Red, TEXT("Game Saved."));
}
```

*Figure 28 - Quick Save function*

Then to retrieve the data from the save file I needed to create another instance of my save game class, and then cast the object returned by the "LoadGameFromSlot" function call to my own save object.

After that it's as simple as getting the location variable from the save game object and assigning it to my player location.

```
void APlayerCharacter::QuickLoad() {
	UMySaveGame* SaveGameInstance = Cast<UMySaveGame>(UGameplayStatics::CreateSaveGameObject(UMySaveGame::StaticClass()));
	SaveGameInstance = Cast<UMySaveGame>(UGameplayStatics::LoadGameFromSlot(TEXT("Quick Save"), 0));
	this->SetActorLocation(SaveGameInstance->PlayerLocation);
	GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Red, TEXT("Game Loaded."));
}
```

*Figure 29 - Quick Load function*

### 8.1.5 Pausing the Game

To pause the game, I bound the P key to a pause function. From the player character I'm simply finding the players HUD class by casting the object returned from the GetHUD function to the HUD class I created and then calling the "openPauseMenu" function. See [Opening Pause Menu] for pause functionality. [9]

```
void APlayerCharacter::Pause() {
	ACharacterHud* myHud = Cast<ACharacterHud>(UGameplayStatics::GetPlayerController(this, 0)->GetHUD());
	myHud->openPauseMenu();
}
```

### 8.1.6 Implementing Gravity plugin

To make directional gravity work for my project, I had to download a plugin to allow me to change the direction of the simulated gravity. The gravity property in the Unreal Engine is hard coded to only point towards the negative Z axis. [10]

This "Directional Gravity" plugin I downloaded created a custom movement component which overrides the default gravity direction and allows for the gravity FVector to be set in any direction, I.e. X, Y, or Z axis.

For this to work with my project however I had to implement the new gravity movement component in my player character. To do this I had to declare a function to get the new gravity movement component by casting the default movement component to the one defined in the plugin [11]

```
UGravityMovementComponent* APlayerCharacter::GetGravityMovementComponent()
{
        return Cast<UGravityMovementComponent>(GetMovementComponent());
}
```

*Figure 30 - Get Gravity Movement Component*

And then I needed to implement the gravity component as my default character movement component. This overrides the gravity direction defined in the character movement component.

```
APlayerCharacter(const FObjectInitializer& ObjectInitializer)
        : Super(ObjectInitializer.SetDefaultSubobjectClass<UGravityMovementComponent>(ACharacter::CharacterMovementComponentName)) {};
```

*Figure 31 - Implement Gravity Component*

For more details see [Gravity Plugin]

## 8.2 PROJECTILE

The projectile class is simply an actor like the other classes created in this project, with physics and collision properties applied to it.

### 8.2.1 Setting up projectile Physics

For the physics to work properly, a few settings first needed to be applied. To prevent it from passing through other objects a collision pre-set first needed to be defined in the project settings. [5]

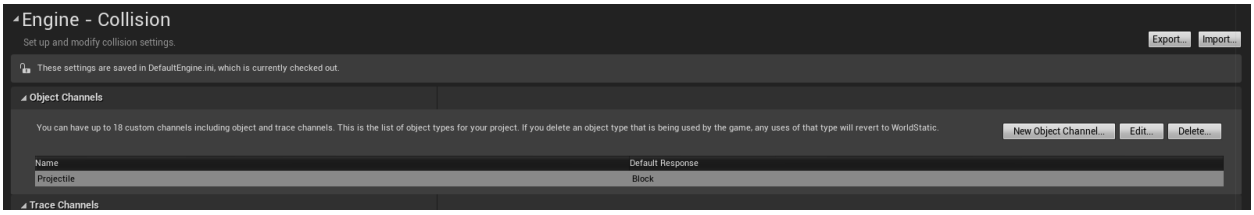



*Figure 32 - Projectile Object Channel*

The object channel allows me to apply these collision pre-sets to objects with that channel applied to them. Then I set up the collision pre-set, which defines exactly what the object should

do when interacting with other objects in the world. In my case I wanted to prevent it from passing through other objects, so I set it to block all other objects.

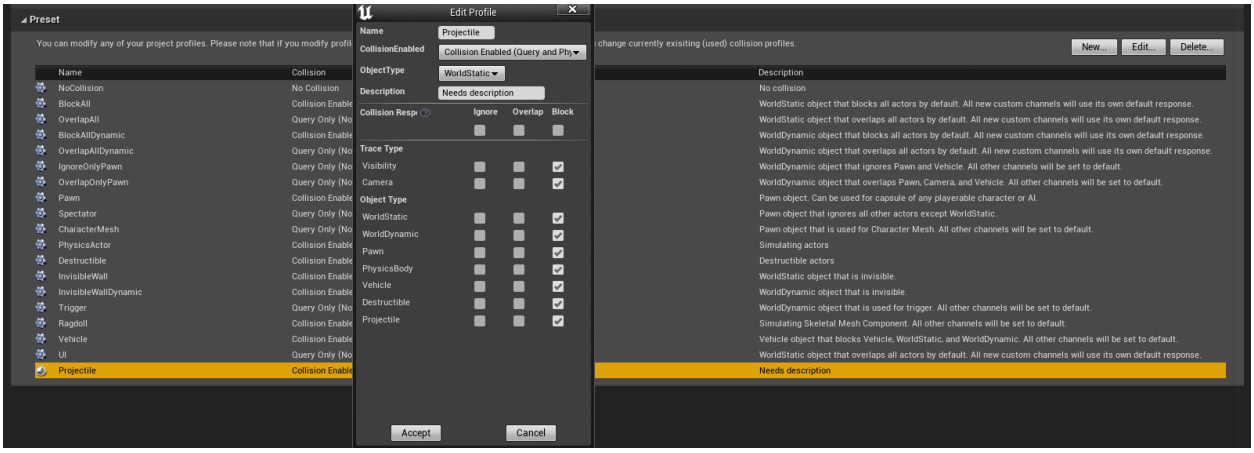



*Figure 33 – Collision Pre-set*

With the pre-set defined, I had to apply it to the class in C++. I got the collision component of the projectile class and set the profile to the projectile pre-set I defined previously. Then I added an OnHit handler to trigger when it collides with another object.

```
if (!CollisionComponent)
{
        CollisionComponent = CreateDefaultSubobject<USphereComponent>(TEXT("SphereComponent"));
        CollisionComponent->BodyInstance.SetCollisionProfileName("Projectile");
        CollisionComponent->OnComponentHit.AddDynamic(this, &AProjectile::OnHit);
        CollisionComponent->InitSphereRadius(15.0f);
        RootComponent = CollisionComponent;
}
```

*Figure 34 - Applying collision*

Then I created the movement component for the projectile class. The movement component in actors contains properties related to simulated physics such as velocity and gravity.

Here I'm setting some of the initial parameters related to the projectile's movement. I'm applying the initial speed, max speed, and gravity to the projectile. I'm also setting the bounce variable which allows the projectile to bounce back from the object it hits. [12]

```
if (!ProjectileMovementComponent)
{
        ProjectileMovementComponent = CreateDefaultSubobject<UProjectileMovementComponent>(TEXT("ProjectileMovementComponent"));
        ProjectileMovementComponent->SetUpdatedComponent(CollisionComponent);
        ProjectileMovementComponent->InitialSpeed = 2000.0f;
        ProjectileMovementComponent->MaxSpeed = 3000.0f;
        ProjectileMovementComponent->bRotationFollowsVelocity = true;
        ProjectileMovementComponent->bShouldBounce = true;
        ProjectileMovementComponent->Bounciness = 0.3f;
        ProjectileMovementComponent->ProjectileGravityScale = 1.0f;
}
```

*Figure 35 - Movement properties*

### 8.2.2 On Hit Event

The On Hit event is fired when the projectile collides with another object. It returns several parameters that can be used for any number of applications, in my case I'm mostly interested in using the OtherActor variable.

It refers to the object that the projectile is colliding with, however, only returns an abstract object of type AActor. To get access to the functionality of that class, I needed to check if it's the class I'm looking for by getting its name and then casting it to that class.

Here I'm using the on hit event to check if the projectile hit the button and if it has, it opens the door to the next stage of the level.

```
void AProjectile::OnHit(UPrimitiveComponent* HitComponent, AActor* OtherActor, UPrimitiveComponent* OtherComponent, FVector NormalImpulse, const FHitResult& Hit)
{
	if (OtherActor->GetClass()->GetName() == "BigRedButton") {
		Cast<ABigRedButton>(OtherActor)->openDoor();
	}
}
```

*Figure 36 - On Hit event*

## 8.3 Path Point

The Path Point class is a very simple place holder with just a Mesh and Material applied to it. It doesn't need to do much because it only serves as a visual indicator along the trajectory line. The meshes and textures were set up like any other class.

First finding the location of the Mesh and Material assets, and if they succeed setting them in the visual mesh variable. I'm also scaling the speres down to a $10^{th}$ of its size, so that they're small enough to not get in the way.

```
	static ConstructorHelpers::FObjectFinder<UStaticMesh> sphereAsset(TEXT("/Game/MyContent/Meshes/TrajectoryPoint.TrajectoryPoint"));
	if (sphereAsset.Succeeded())
	{
		VisualMesh->SetStaticMesh(sphereAsset.Object);
		VisualMesh->SetWorldScale3D(FVector(0.1f, 0.1f, 0.1f));
		VisualMesh->SetRelativeLocation(FVector(0.0f, 0.0f, 0.0f));
		VisualMesh->SetCollisionEnabled(ECollisionEnabled::NoCollision);
	}

	static ConstructorHelpers::FObjectFinder<UMaterial>Material(TEXT("/Game/MyContent/Materials/Trajectory.Trajectory"));
	if (Material.Succeeded())
	{
		PathPointMaterial = UMaterialInstanceDynamic::Create(Material.Object, VisualMesh);
		VisualMesh->SetMaterial(0, PathPointMaterial);
	}
```

*Figure 37 - Path Point visual properties*

## 8.4 Big Red Button

To move on to the next stage of the level I designed a button that would interact with the exit door. To trigger the door to open, the player needs to throw their projectile at the button which will then set the exit door to open.

To allow the button to directly reference that specific door, I had to declare a UProperty of the door object type in the button class declaration. This allows me to link this button to a specific door when I set that variable in the property's menu in the unreal editor.

```
UPROPERTY(EditAnywhere)
        AEntrance* entrance;
```

*Figure 38 - Door object*

After creating a blueprint of the button class, I can directly set this variable in its blueprint's properties.

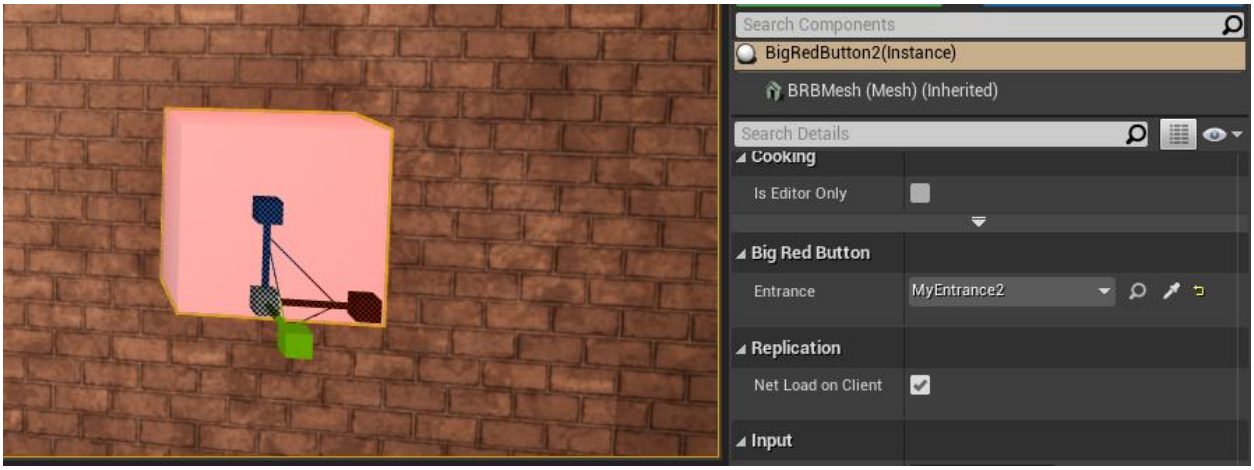



*Figure 39 - Setting entrance property*

With a pointer to the entrance, I can simply call the setOpen function from the entrance itself. The pointer is a direct reference to that instance of the door.

```
int ABigRedButton::openDoor()
{
        GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Red, TEXT("Opening Door."));
        if (entrance != nullptr) {
                entrance->setOpen(true);
        }
        return 0;
}
```

*Figure 40 - Opening door*

## 8.5 ENTRANCE

The entrance class is the door which is connected to the button class through the blueprint. This acts as a separator between stages in the level. The goal being to open up the door and pass on to the next stage.

The door waits on the button being hit which sets a Boolean value in the Entrance class to true. Once it's set the doors location Z value starts to decrement, which starts moving the entrance downwards. It keeps decreasing until the doors position moves past over its total height, placing it just below the ground.

When the "open" Boolean value gets set back to false after passing through, the doors Z position starts to increment again, until its position is back to where it started from.

```
void AEntrance::Tick(float DeltaTime)
{
	Super::Tick(DeltaTime);
	FVector doorLocation = GetActorLocation();

	if (open) {
		if ((positionCounter < (dimensions.Z * 2) + 10)) {
			positionCounter++;
			doorLocation.Z--;
			SetActorLocation(doorLocation);
		}
	}
	else {
		if (positionCounter > 0) {
			positionCounter--;
			doorLocation.Z++;
			SetActorLocation(doorLocation);
		}
	}
}
```

To account for different sized doors, I used the dimensions of the mesh rather than a hard coded value. This means that if I had a larger exit, the door would still move its position down below its height. [13]

```
if (entranceAsset.Succeeded()) {
	EntranceMesh->SetStaticMesh(entranceAsset.Object);
	dimensions = EntranceMesh->GetStaticMesh()->GetBounds().BoxExtent * EntranceMesh->GetRelativeScale3D();
}
```

*Figure 41 - Getting dimensions of door*

To open and close the door I created a setter function to set the Boolean value in the class to either true or false depending on the use.

```
int AEntrance::setOpen(const bool isOpen)
{
	open = isOpen;
	return 0;
}
```

*Figure 42 - Door setter*

## 8.6 Pressure Plates

While the button class was used to open the door, the pressure plate class is used to close it after the player has passed through. The pressure plate behaves very similar to the button and gets passed the entrance reference as well.

I check for the players position in the pressure plate class every tick and compare it to the position of the pressure plates origin and its end point. This will only be set to true once the player as stepped inside the pressure plates area.

```
void APressurePlate::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
    if (UGameplayStatics::GetPlayerController(GetWorld(), 0)->GetPawn() != nullptr) {
        APlayerController* controller = UGameplayStatics::GetPlayerController(GetWorld(), 0);
        if (controller->GetPawn()->GetActorLocation().Y >= this->GetActorLocation().Y && controller->GetPawn()->GetActorLocation().Y <= (this->GetActorLocation().Y + dimensions.Y)) {
            if (controller->GetPawn()->GetActorLocation().X >= this->GetActorLocation().X && controller->GetPawn()->GetActorLocation().X <= (this->GetActorLocation().X + dimensions.X)) {
                closeDoor();
            }
        }
    }
}
```

*Figure 43 - Checking for player location*

Once the check for players X and Y values are true, a local function to close the entrance gets called which simply sets the entrance "open" Boolean value to false, which causes the door to start closing.

```
int APressurePlate::closeDoor()
{
    GEngine->AddOnScreenDebugMessage(-1, 5.0f, FColor::Red, TEXT("Closing Door."));
    if (entrance != nullptr) {
        entrance->setOpen(false);
    }
    return 0;
}
```

*Figure 44 - Setting door to closed*

## 8.7 Character Hud

The character HUD class is the main UI component that's attached to the player character. This contains code related to GUI components the player can see and interact with. This is also the class from which menus can be opened from.

### 8.7.1 Setting up Crosshair

The Crosshair acts as an indicator in the centre of the players view to show what they're pointing at. To create the UI and the Crosshair, the "DrawHUD" function needs to be implemented from the abstract AHUD class in the unreal engine. [5]

Since GUI components are shown as 2D objects overlaying the players view, the positioning needs to be calculated in that 2D area.

Firstly, I'm getting the centre of the screen and then I'm calculating the size of the Crosshair texture to account for the width and height of the image. Then I'm simply drawing the texture on to the 2D canvas, which is the player's view.

```
void ACharacterHud::DrawHUD()
{
    Super::DrawHUD();

    if (CrosshairTexture)
    {
        // Find the center of our canvas.
        FVector2D Center(Canvas->ClipX * 0.5f, Canvas->ClipY * 0.5f);

        // Offset by half of the texture's dimensions so that the center of the texture aligns with the center of the Canvas.
        FVector2D CrossHairDrawPosition(Center.X - (CrosshairTexture->GetSurfaceWidth() * 0.5f), Center.Y - (CrosshairTexture->GetSurfaceHeight() * 0.5f));

        // Draw the crosshair at the centerpoint.
        FCanvasTileItem TileItem(CrossHairDrawPosition, CrosshairTexture->Resource, FLinearColor::White);
        TileItem.BlendMode = SE_BLEND_Translucent;
        Canvas->DrawItem(TileItem);
    }
}
```

*Figure 45 - Draw Crosshair*

### 8.7.2 Opening Pause Menu

The pause menu allows the player to pause the game whenever it's opened. This stops the tick components from running while the menu is open. I created this menu using the UMG UI Designer in the Unreal Engine. It's simply a transparent overlay with a set of buttons to Resume, Save, Load, and Quit the game. For more details on menu functionality see [Pause Blueprint]

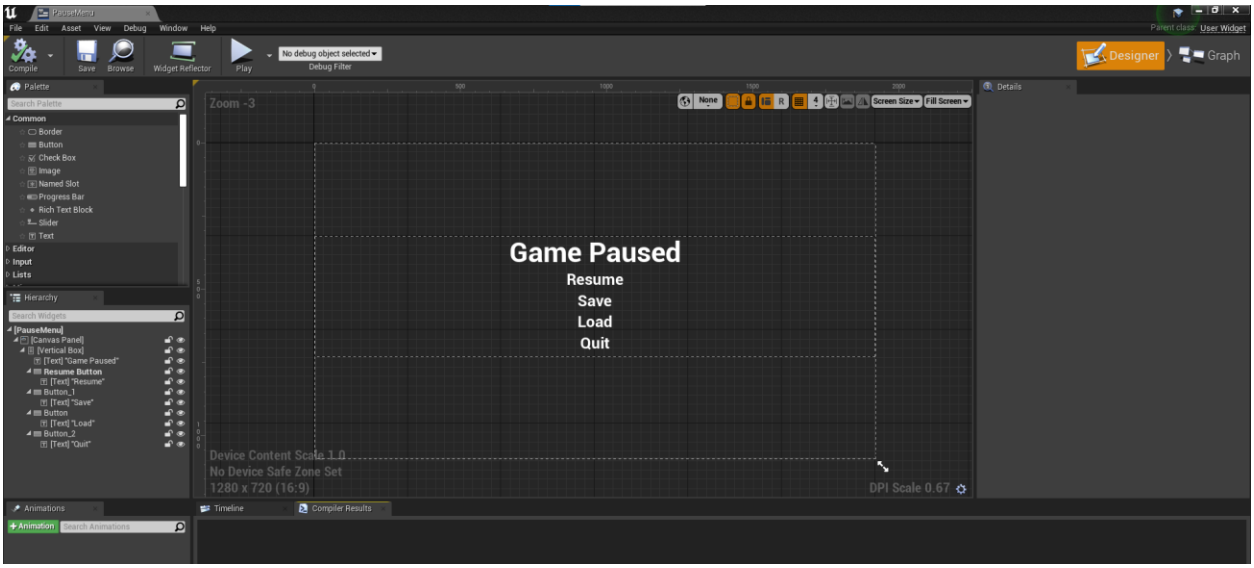



*Figure 46 - Pause menu*

Before I was able to open the menu however, I needed to find the "Widget" class. I used the FClassFinder function call to get my PauseMenu blueprint as a UUserWidget object. Only after finding the object and getting its class from that object can you use the UUserWidget functions. [14], [15]

```
ACharacterHud::ACharacterHud()
{
    static ConstructorHelpers::FClassFinder<UUserWidget> WidgetAsset(TEXT("WidgetBlueprint'/Game/MyContent/UI/PauseMenu.PauseMenu_C'"));
    if (WidgetAsset.Succeeded()) {
        PauseMenuWidgetClass = WidgetAsset.Class;
    }
```

*Figure 47 - Finding UUserWidget Class*

With the class found, I used the UGameplayStatics function "SetGamePaused" to stop the game from running while the pause menu is open. I then created the pause menu widget by using the class I found earlier of my blueprint, and I set it to the current world which is the level the player is currently in.

Then I removed player input from the game and set it so that the player can only interact with the pause menu. This stopped them from being able to move their character and then set the mouse cursor as visible. [16]

With all the configurations done I finally added the widget to the players viewport so that they can see and interact with it.

```
void ACharacterHud::openPauseMenu()
{
    if (PauseMenuWidgetClass) {
        APlayerController* controller = UGameplayStatics::GetPlayerController(GetWorld(), 0);
        UGameplayStatics::SetGamePaused(GetWorld(), true);
        UUserWidget* Widget = CreateWidget(GetWorld(), PauseMenuWidgetClass);
        if (Widget) {
            UWidgetBlueprintLibrary::SetInputMode_UIOnly(controller, Widget);
            controller->bShowMouseCursor = true;

            Widget->AddToViewport();
        }
    }
```

*Figure 48 - Opening the Pause Menu*

## 8.8 Moving Platforms

The moving platforms class acts as a sort of obstacle to the player. They're programmed to move along the X axis between positive and negative values.

To calculate the position of the platform I'm using the creation time of the object and passing it to a sin function along with the DeltaTime (Which is the time between frames). This then is multiplied by 2 to increase the steps it takes.

The location of the platform is then incremented by the result of the sin function. This makes the platform move smoothly along this waveform [17]

```
void AMovingPlatform::Tick(float DeltaTime)
{
	Super::Tick(DeltaTime);
	FVector location = GetActorLocation();
	float time = GetGameTimeSinceCreation();
	float posX = FMath::Sin(time + DeltaTime) * 2;

	if (thisPlatform) {
		location.X += posX;
	}
	else {
		location.X -= posX;
	}

	SetActorLocation(location);
}
```

*Figure 49 - Moving Platforms Sin*

## 8.9 My Save Game

The Save Game class is simply an implementation of the class defined in the Unreal Engine libraries. It contains placeholder variables which will be saved to the file when saving the game. Anything that needs to be saved will be set in the Save Game object.

In my case I'm saving the index of the save slot, the save slot name, and the player location. The only variable which will be used upon retrieving the save data is the Player Location. [8]

```
public:
	UPROPERTY(VisibleAnywhere, Category = Basic)
		FString SaveSlotName;

	UPROPERTY(VisibleAnywhere, Category = Basic)
		uint32 UserIndex;

	UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category = Basic)
		FVector PlayerLocation;

	UMySaveGame();
```

*Figure 50 - Save Game Class*

# 9 ADDITIONAL FEATURES

## 9.1 GRAVITY PLUGIN

When I was creating my directional gravity feature, I quickly discovered that the gravity direction in the Unreal Engine is hard coded to point towards the negative Z axis. To be able to implement the feature I wanted, I'm using a directional gravity plugin which overrides the default movement component with one that allows the gravity FVector to be set in other directions [10]

To use this new functionality, I first had to implement the new gravity movement component in the player character. See [Implementing Gravity plugin].

With the new gravity movement component implemented I can now call the "SetGravityDirection" function from a blueprint class. I simply get my player controller like I would in C++, and the call the Get Gravity Movement Component function from that. This then allows me to call the Set Gravity Direction function.

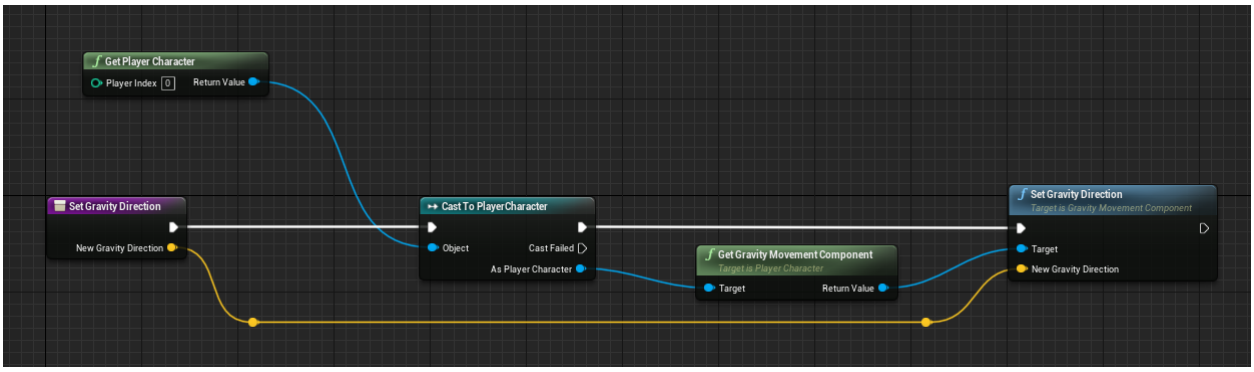



*Figure 51 - Set gravity direction*

The value that it's being passed can be seen here. The new gravity direction will be a force of 9.8 meters per second on the X axis to replicate the conditions on earth.

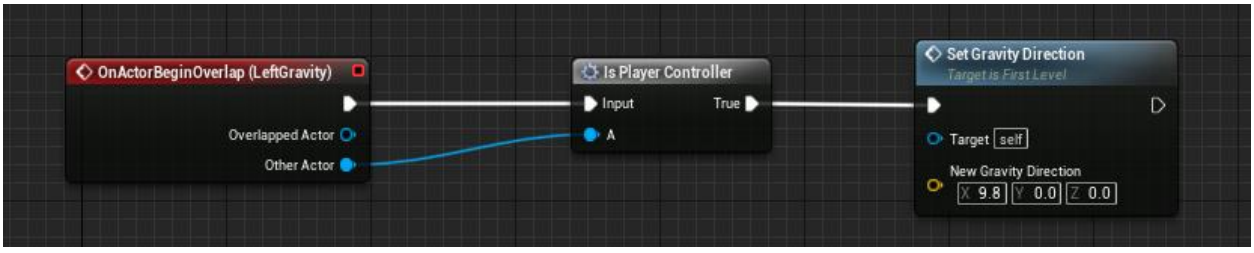



*Figure 52 - FVector passed to gravity*

## 9.2 TRIGGERS

A trigger is an object that can be placed through the editor which will generate an overlap event when another object passes through it. To set my new gravity direction, I'm using a trigger to generate an event when the player passes through it. See [Figure 52 - FVector passed to gravity]

This overlap event will continue to be active if the player remains in the trigger area. Upon exiting another event is fired, the OnActorEndOverlap event. Once this happens, I'm setting the gravity back to the default -9.8 M/s on the negative Z Axis.

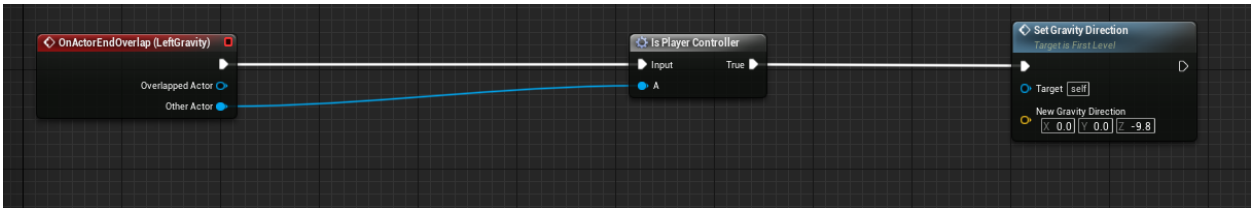



*Figure 53 - Resetting gravity direction*

## 9.3 Pause Blueprint

The pause menu logic is handled in 2 places. In C++ where the widget gets created and set to the players viewport, and in Blueprints where I define the save, quit, and resume button handlers.

### 9.3.1 Save Handler

The process for saving the game works the same in blueprints as it does in C++. Firstly the Save Game object gets created from the template class I defined in C++. The player location gets passed to the FVector variable inside the save game object, and then that object is saved to the slot, creating the save file.

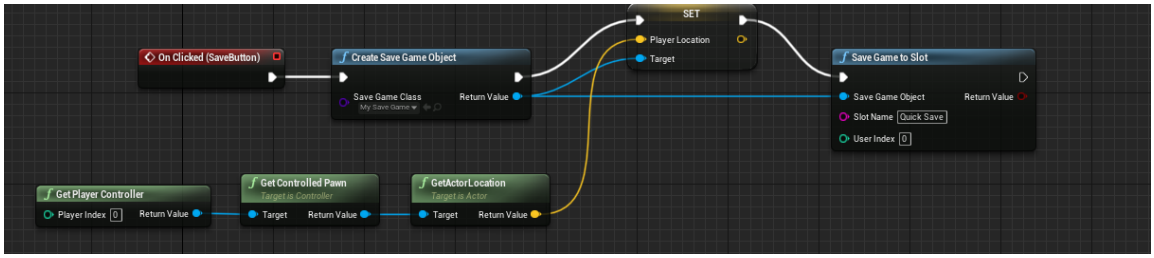



*Figure 54 - Save Handler*

### 9.3.2 Load Handler

To load a game from the pause menu, 2 steps need to be taken. Firstly the same logic as C++ is applied to the blueprint graph. The file is loaded and cast to the "MySaveGame" class. Then the location is retrieved from that object and the player location gets updated from it.

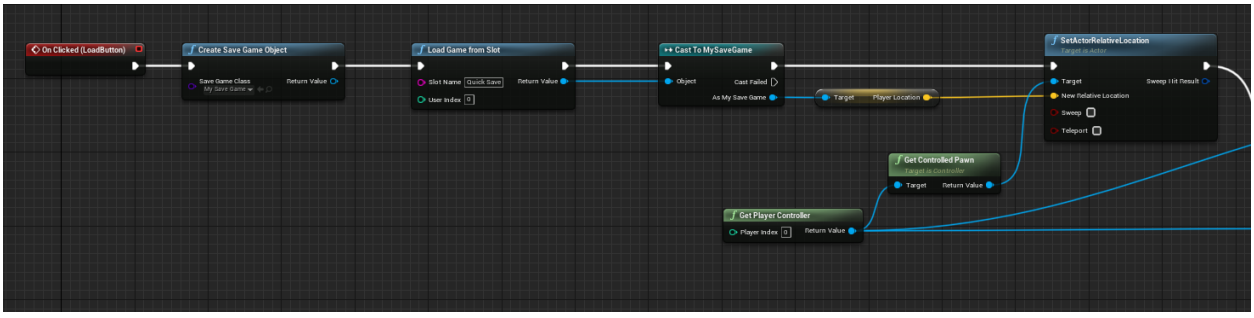



*Figure 55 - Load Handler*

However, there's still the problem with the menu being open and the game set to paused. So, the same logic as the resume handler needed to be applied here as well to close the menu. The input mode is set back to the game, the cursor is hidden, the widget is removed, and the game is set to un-paused again.

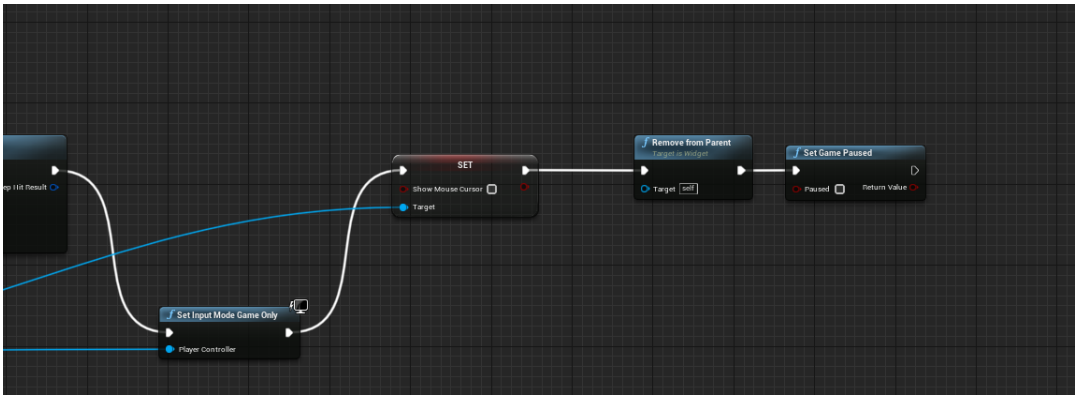



*Figure 56 - Unpausing Game*

### 9.3.3 Resume Handler

To resume a game from a paused menu, all the steps to pause a game need to be reversed. Firstly, the input mode needs to be changed back to the game, then the mouse cursor needs to be hidden again. The pause widget gets removed. And finally, the game is set to un-paused.

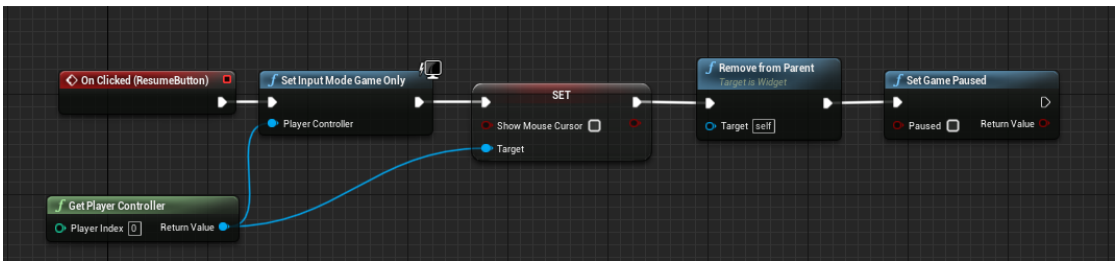



*Figure 57 - Resume handler*

### 9.3.4 Quit Handler

To quit the game from a blueprint is pretty simple. I simply call the Execute Console Command function and pass it the quit parameter which closes the current session.

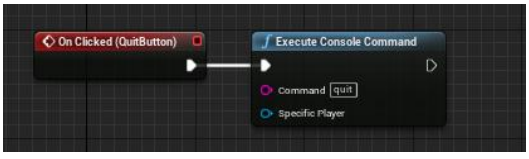



*Figure 58 - Quit game*

## 9.4 Main Menu Blueprint

To create a working main menu, I needed to create a blank level. This was to prevent any components from starting before the player clicked to start the game. The new blank level was simply and empty space where only the main menu would open.

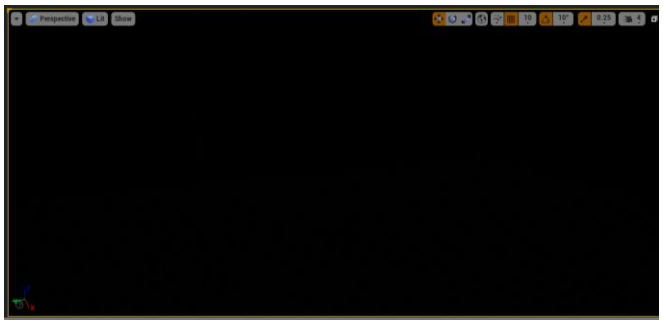
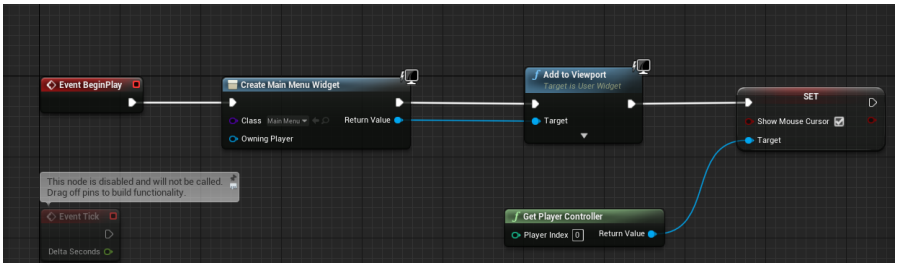


*Figure 59 - Blank Level*

This blank initial level had a simple blueprint script which would open the main menu widget, add it to the view port, and set the mouse cursor to visible.




The main menu has 3 buttons to click. Continue, New Game, and Quit. Each of these has its own handler in the Widgets blueprint.

### 9.4.1 Continue Handler

The continue handler does 2 things. Firstly, it opens the playable level similarly to the new game handler. Then it creates a save game object to load the save file and retrieve the player location from it.

It then updates the players location to the one retrieved from the save file.

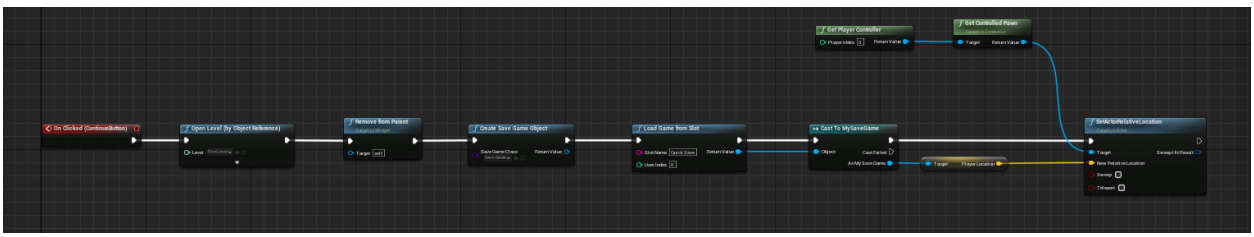
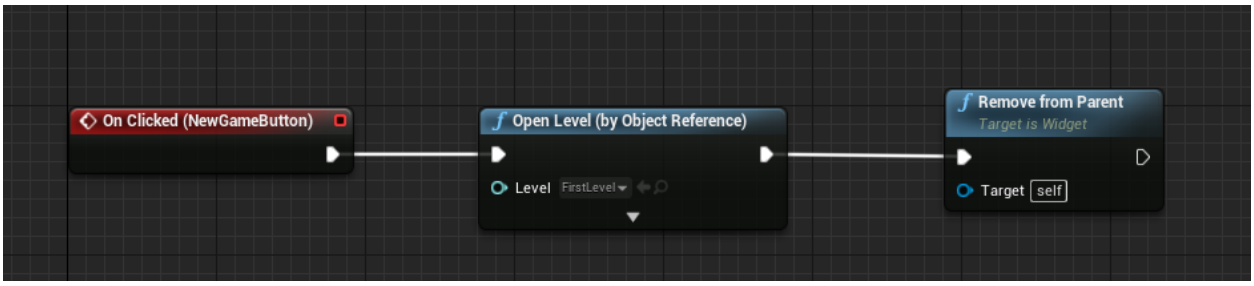


*Figure 60 - Load game from Main Menu*

### 9.4.2 New Game Handler

When the new game button is clicked, the widget opens the playable level and removes itself from the players viewport, closing the main menu.




### 9.4.3 Quit Handler

Identically to the pause menu quit handler, there's a single command executed from the blueprint, which runs the quit console command, closing the game.

# 10 CONCLUSION

In conclusion, I've developed a solid base from which I could develop a full game out of. I've added many features which I can reuse throughout developing a full game out this project.

The result of this project is an environment the player and interact with and explore.

There's a full set of player inputs which allows them walk and interact with the world around them using the W, A, S and D keys.

There's a working projectile and trajectory system which is used to affect the world around them, and other objects such as throwing it at a button to open a door.

A save and load system which stores the player location and updates it when the game is loaded from the save file that's created. This is also incorporated into a main menu and a pause menu which can save and load to and from that save file.

Several environmental components such as moving platforms, doors which can be opened and closed through the use of a button, and gravitational fields which sets the players gravity to a different direction.

I have created these components to be re-usable in different locations if I were to expand the game and add on to it. And hopefully in the future I'll be able to develop this game further into a full working game.

# 11 Appendix

## 11.1 Asset

An asset could be anything from a 3D model designed in a 3rd party software such as Blender to an audio file to create sound effects in game. An asset can be used in any number of ways to create gameplay features or effects when the game runs

## 11.2 Mesh/Model

A mesh or model is a specific type of 3D asset which is usually created in an application such as Blender. These can be assigned to several different inherited classes inside the unreal engine.

## 11.3 Skeleton/Skeletal Mesh

A skeleton or a skeletal mesh is a 3D model specifically designed to have parts which can rotate, allowing you to apply animations to that model. These are primarily used for Pawns and Characters in the unreal engine.

## 11.4 Texture/Material

A texture or material is a top level covering of a mesh. This can be a solid colour or any pattern that wraps around a 3D mesh.

## 11.5 HUD

HUD stands for Heads Up Display. It acts as a user interface in games to display important information to the player, for example, a players life bar or ammunition count in a shooter game.

## 11.6 FVector

An FVector is a 3-dimensional coordinate system which holds the X, Y and Z coordinates of whatever it's assigned to. It can also hold directional forces such as for gravity or velocity.

# 12 REFERENCES


[1] "Garbage Collection | Unreal Engine Community Wiki." https://unrealcommunity.wiki/garbage-collection-36d1da (accessed Apr. 19, 2022).

[2] Epic Games, "Introduction to Blueprints | Unreal Engine Documentation." https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/GettingStarted/ (accessed Apr. 19, 2022).

[3] "Setting Default Pawn In C++ - Programming & Scripting / C++ - Unreal Engine Forums." https://forums.unrealengine.com/t/setting-default-pawn-in-c/90489 (accessed Apr. 18, 2022).

[4] Epic Games, "First Person Shooter Tutorial | Unreal Engine Documentation." https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/CPPTutorials/FirstPersonShooter/ (accessed Apr. 18, 2022).

[5] Epic Games, "Implementing Projectiles | Unreal Engine Documentation." https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/CPPTutorials/FirstPersonShooter/3/#3.5-addingcrosshairstoyourviewport (accessed Apr. 18, 2022).

[6] Epic Games, "FPredictProjectilePathParams | Unreal Engine Documentation." https://docs.unrealengine.com/4.27/en-US/API/Runtime/Engine/Kismet/FPredictProjectilePathParams/ (accessed Apr. 20, 2022).

[7] Epic Games, "Saving and Loading Your Game | Unreal Engine Documentation." https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/SaveGame/ (accessed Apr. 21, 2022).

[8] reubs, "Unreal Engine C++ Tutorial - Creating Savegames - YouTube." https://www.youtube.com/watch?v=jkfbzt8etws (accessed Apr. 18, 2022).

[9] "UE4 C++ Getting a reference to HUDClass after it has been initialized - Stack Overflow." https://stackoverflow.com/questions/64251219/ue4-c-getting-a-reference-to-hudclass-after-it-has-been-initialized (accessed Apr. 18, 2022).

[10] Tefel Dev, "Directional & Planet Gravity in Code Plugins - UE Marketplace." https://www.unrealengine.com/marketplace/en-US/product/directional-planet-gravity?sessionInvalidated=true (accessed Apr. 20, 2022).

[11] Tefel Dev, "Free Directional gravity, gravitatation like Planetary Anihilation - YouTube." https://www.youtube.com/watch?v=bcTqFA23P9Y (accessed Apr. 20, 2022).

[12] "Scale and Measurement Inside Unreal Engine 4 – techarthub." https://www.techarthub.com/scale-and-measurement-inside-unreal-engine-4/ (accessed Apr. 20, 2022).

[13] "C++ get Mesh size - Unreal Engine / Programming & Scripting - Unreal Engine Forums." https://forums.unrealengine.com/t/c-get-mesh-size/391638 (accessed Apr. 18, 2022).

[14] Mike Stevanovic, "UE4 C++ - Endless Runner from scratch - Episode 14 - Pause Game - YouTube." https://www.youtube.com/watch?v=B387lB5ZFY0 (accessed Apr. 18, 2022).

[15] "Error creating a UUserWidget in C++ - Programming & Scripting / UI - Unreal Engine Forums." https://forums.unrealengine.com/t/error-creating-a-uuserwidget-in-c/368162 (accessed Apr. 18, 2022).

[16] "How to set input mode in C++ - Programming & Scripting / C++ - Unreal Engine Forums." https://forums.unrealengine.com/t/how-to-set-input-mode-in-c/82797 (accessed Apr. 18, 2022).

[17] "Moving an object up and down using sine - Epic Games Forums." https://forums.epicgames.com/udk/udk-development/udk-programming-and-unrealscript/230323-moving-an-object-up-and-down-using-sine (accessed Apr. 18, 2022).