# INTRODUCTION TO UNREAL ENGINE PART 1

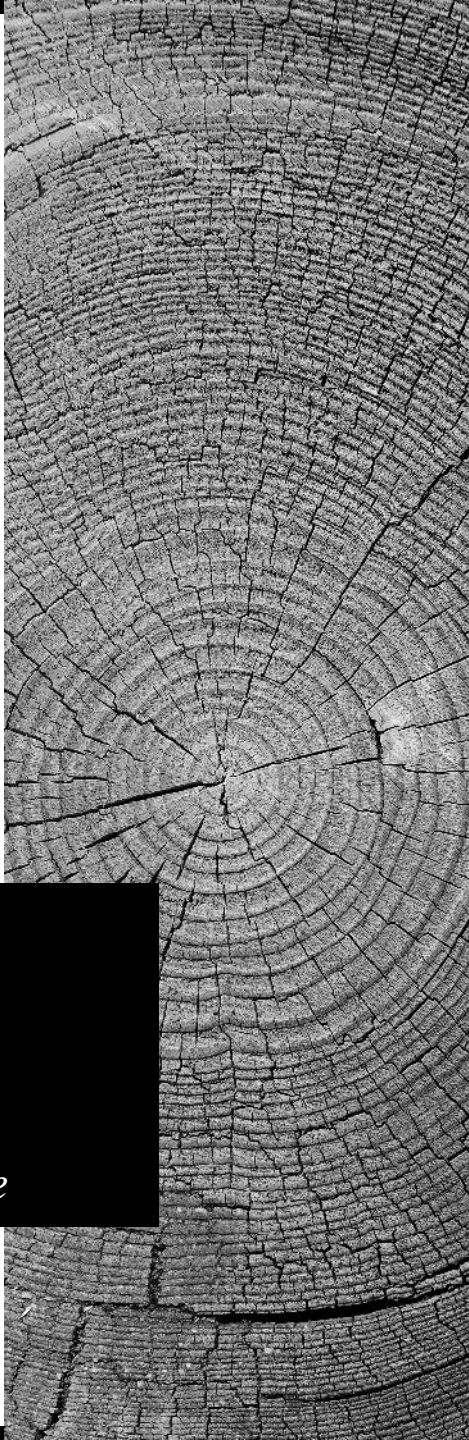
Unreal Engine 5.3.2

*By:*

*Natasha Mangan*

*Course:*

*Game Engine Architecture*



# ABOUT THIS INTRODUCTION

*It will be based on Unreal Engine version 5.3.2*

## It will include:

- C++ implementation
- Blueprint implementation (in part 2)

Some blueprints will be shown in part 1

# THE PROJECT

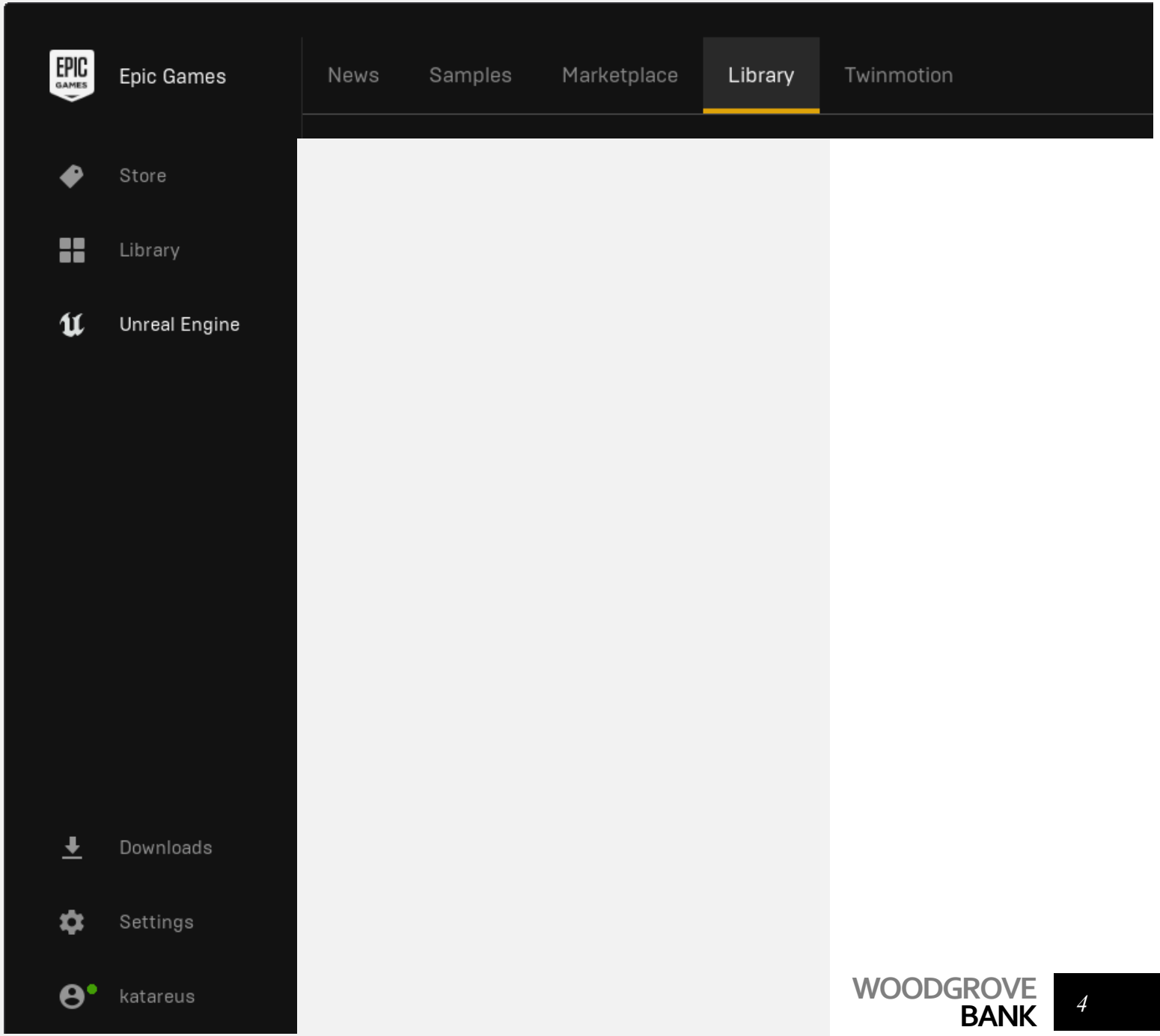
*Epic Games Store*

*Creating the Project*

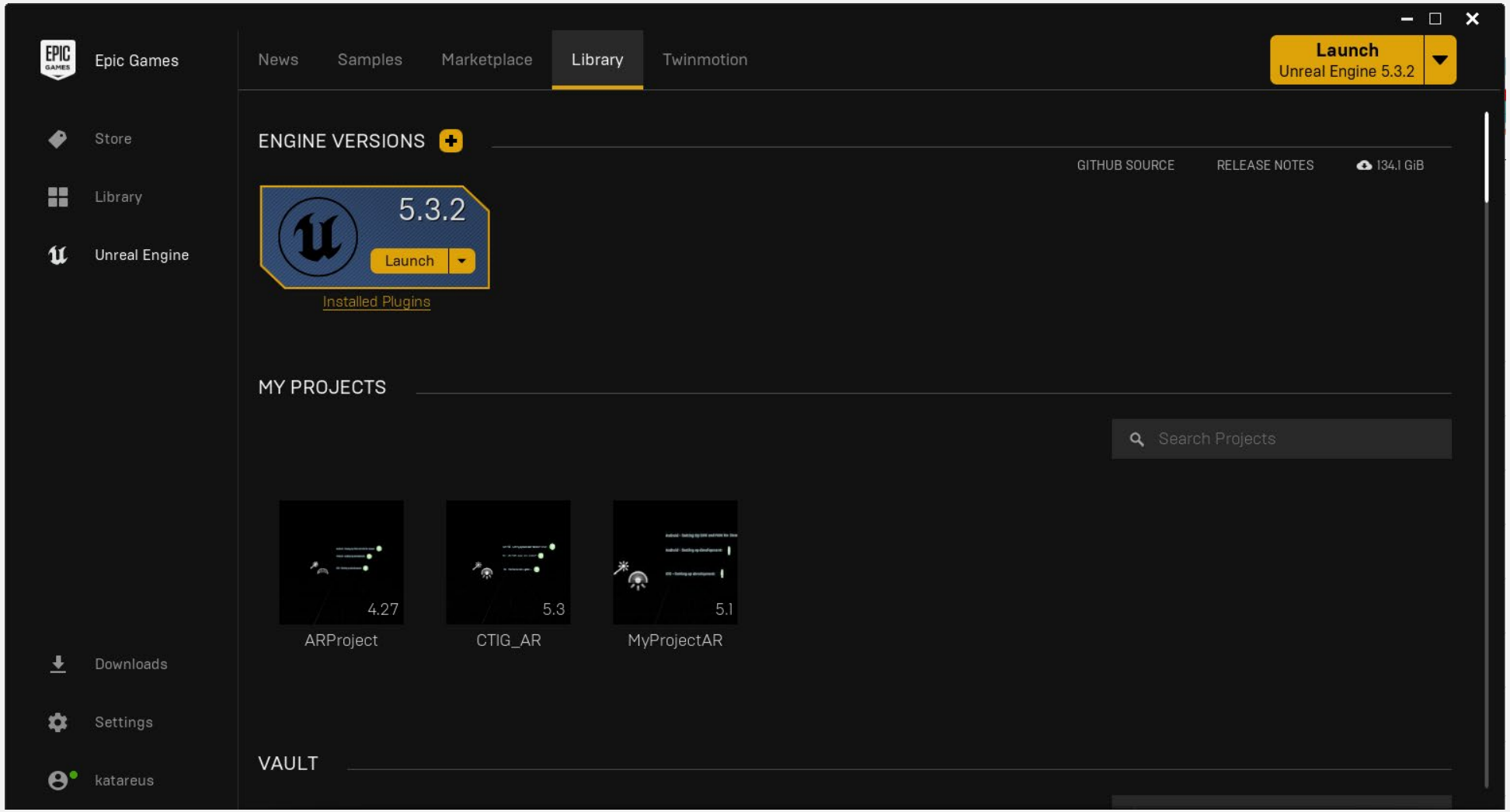
*Visual Studio*

# EPIC GAMES STORE

- Navigate to Unreal Engine on the left bar
- Then select Library on the top bar

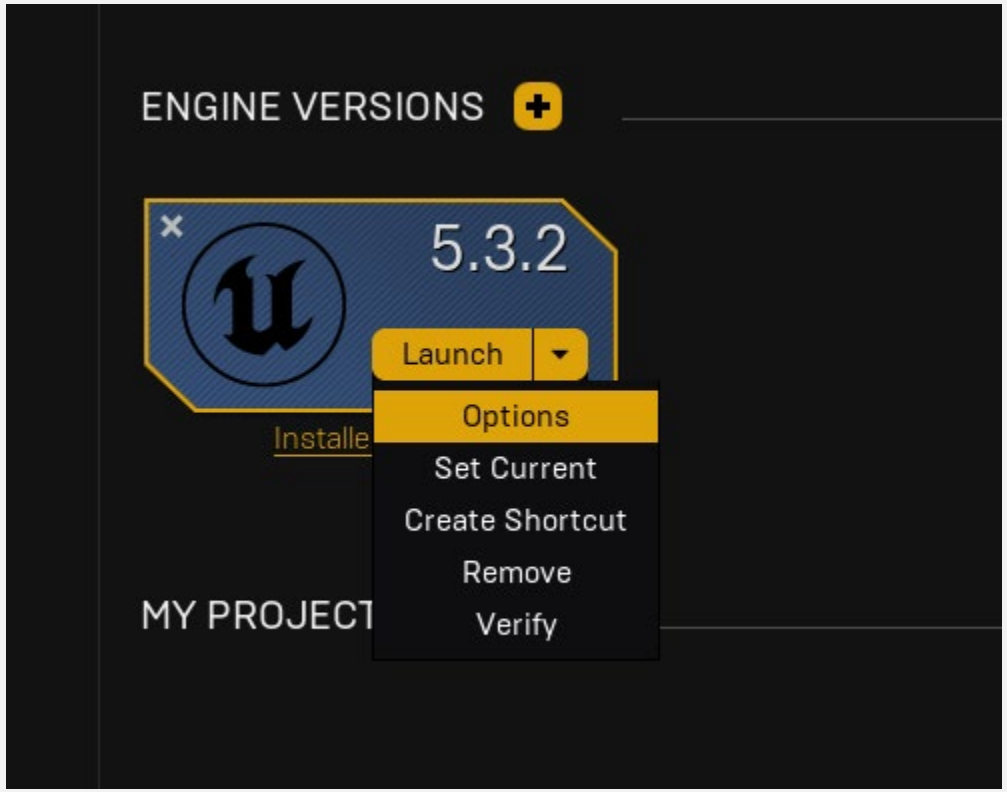


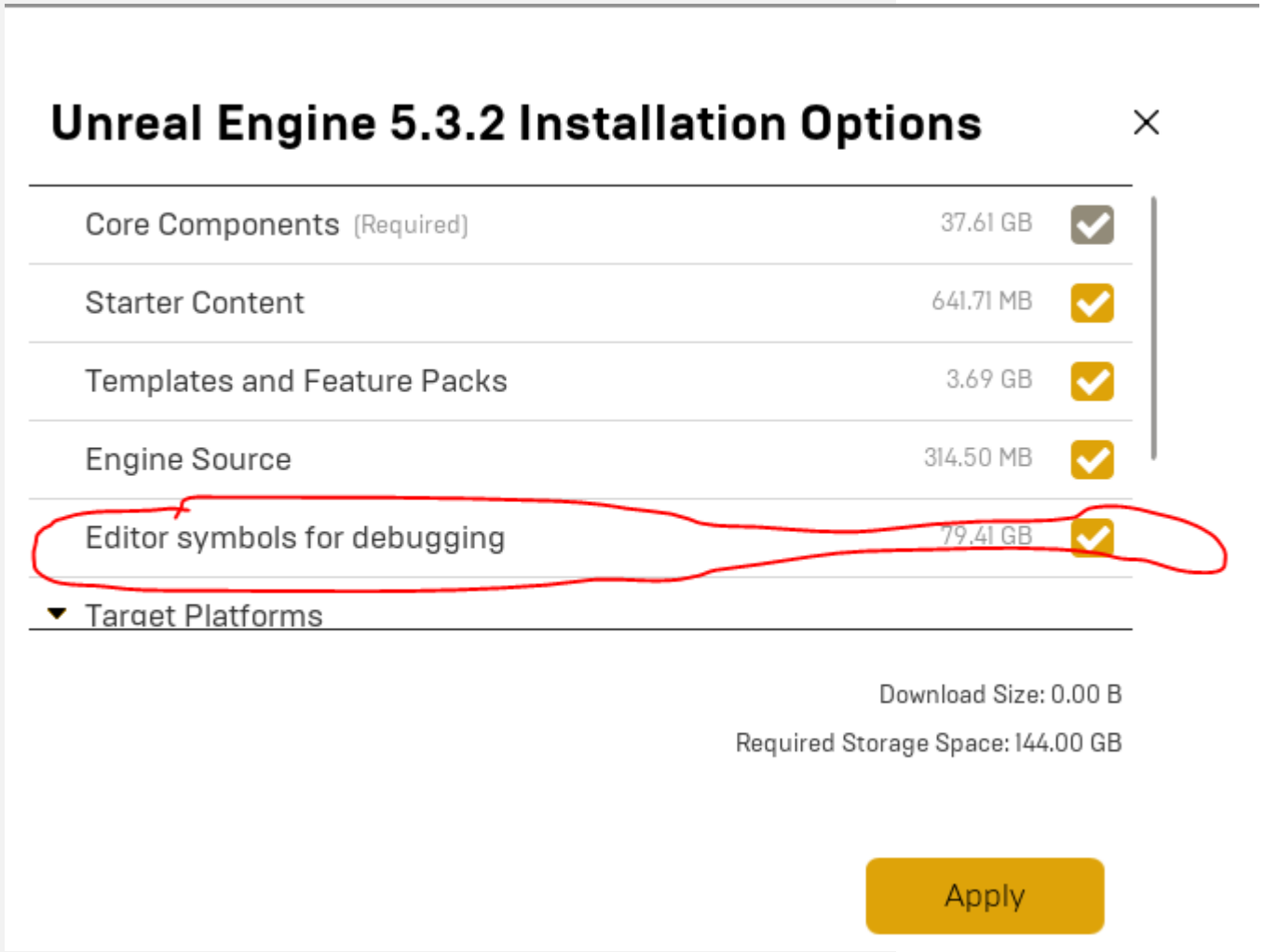
# EPIC GAMES STORE



# EPIC GAMES STORE

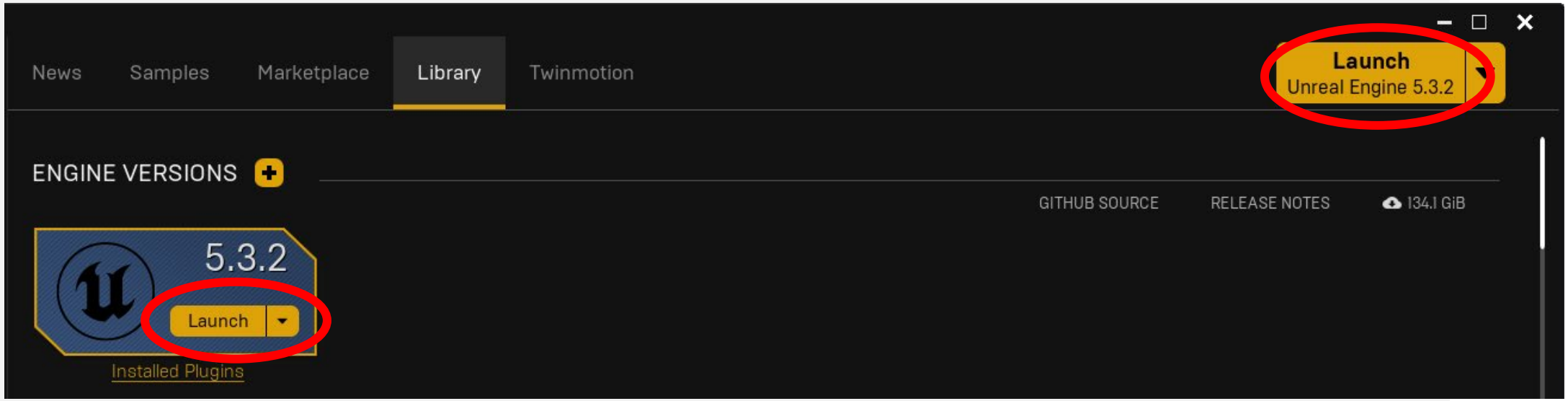
- For debugging with breakpoints in c++ you need to install an additional module



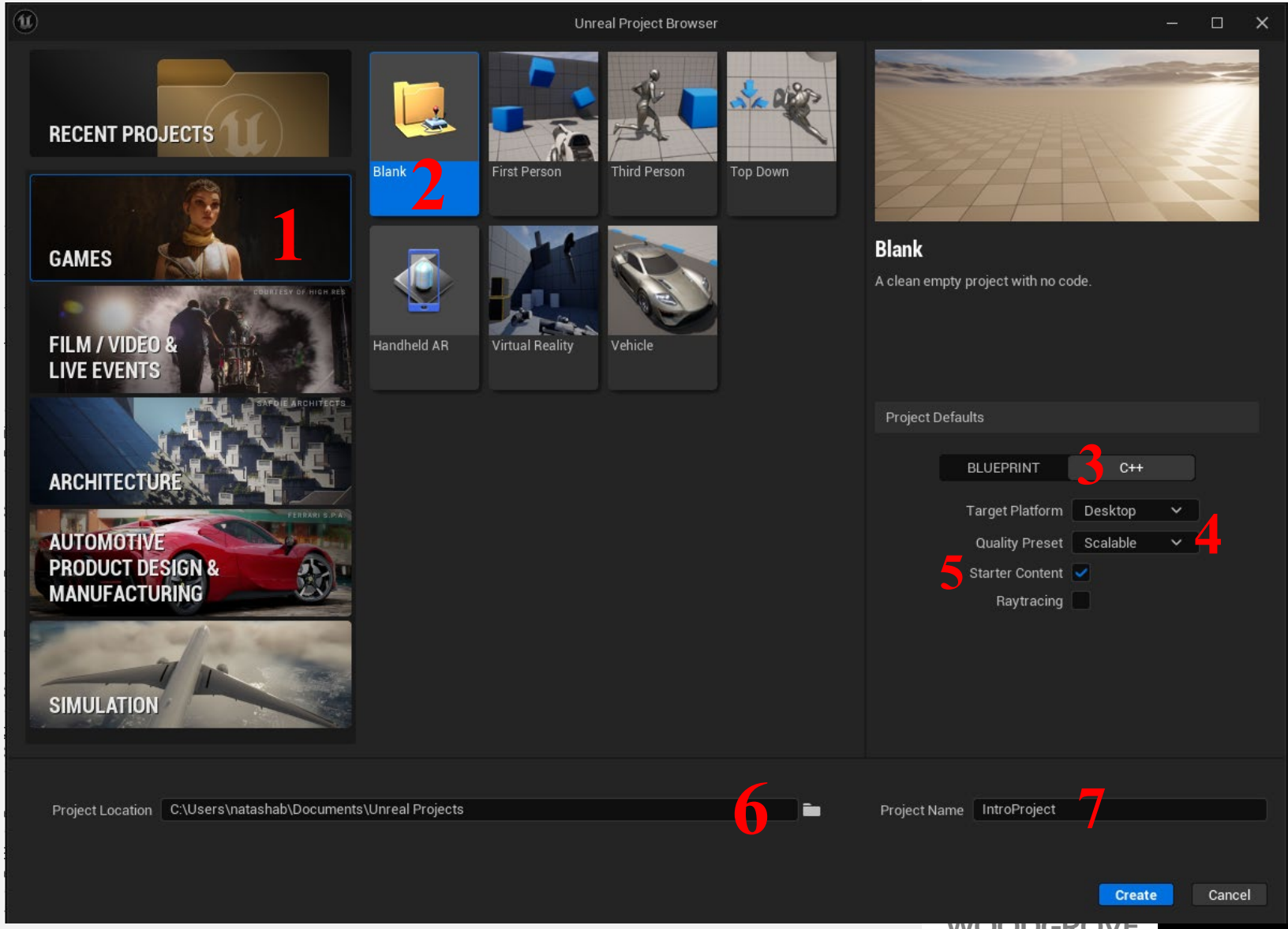
# CREATING THE PROJECT

- Click launch
  - On the engien version
  - Top right



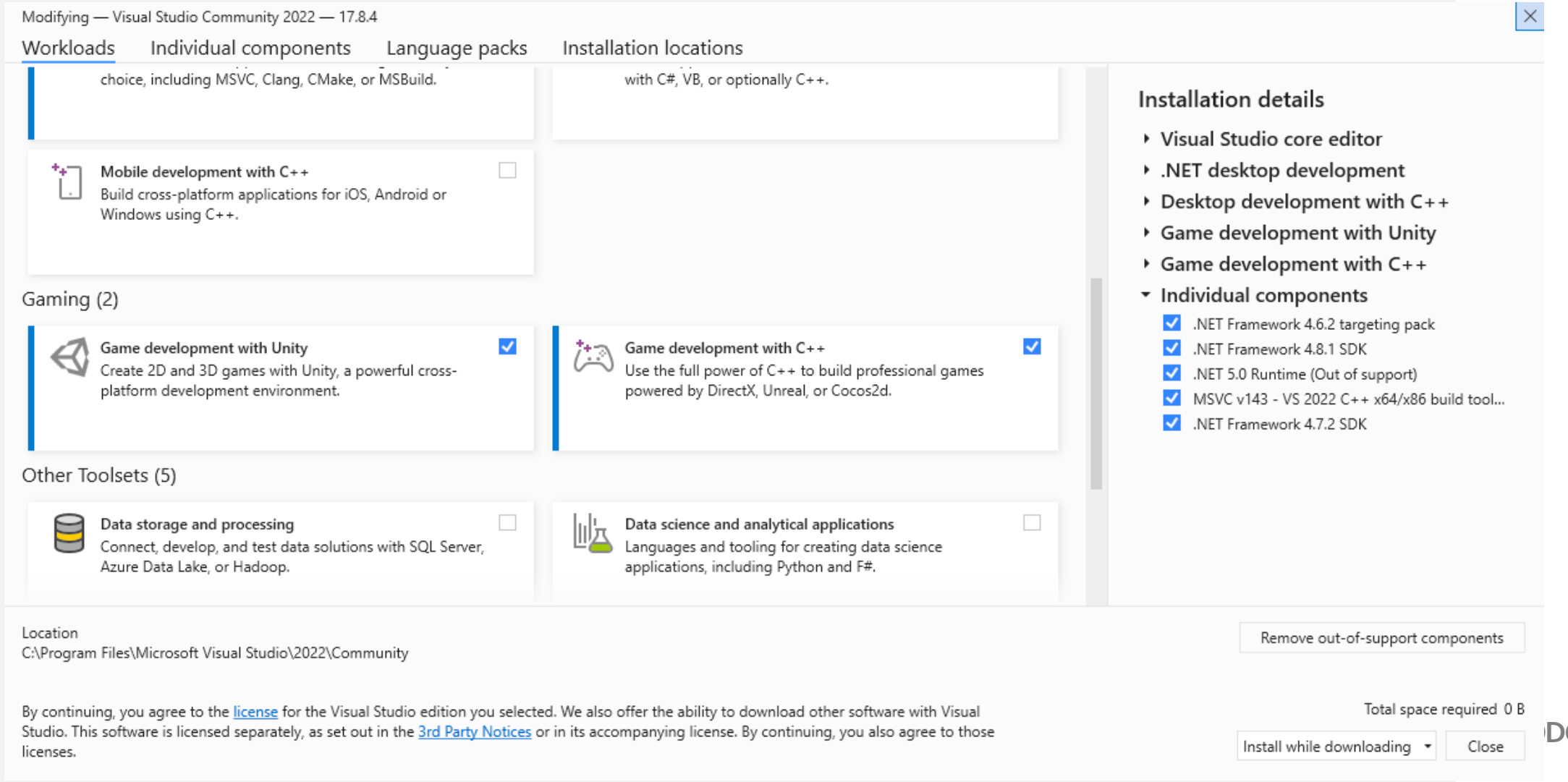
# CREATING THE PROJECT

1. Navigate to Games on the left panel
2. Click Blank Project
3. Select C++
   - You can change this later it is not set in stone
4. Set the quality preset to Scalable
   - Higher quality requires more RAM and higher specs
   - Scalable can be used on any device under any quality settings
5. Enable Starter Content
6. Set location of the project
7. Name the project
   - Do note that a folder will be created at the location based on project name

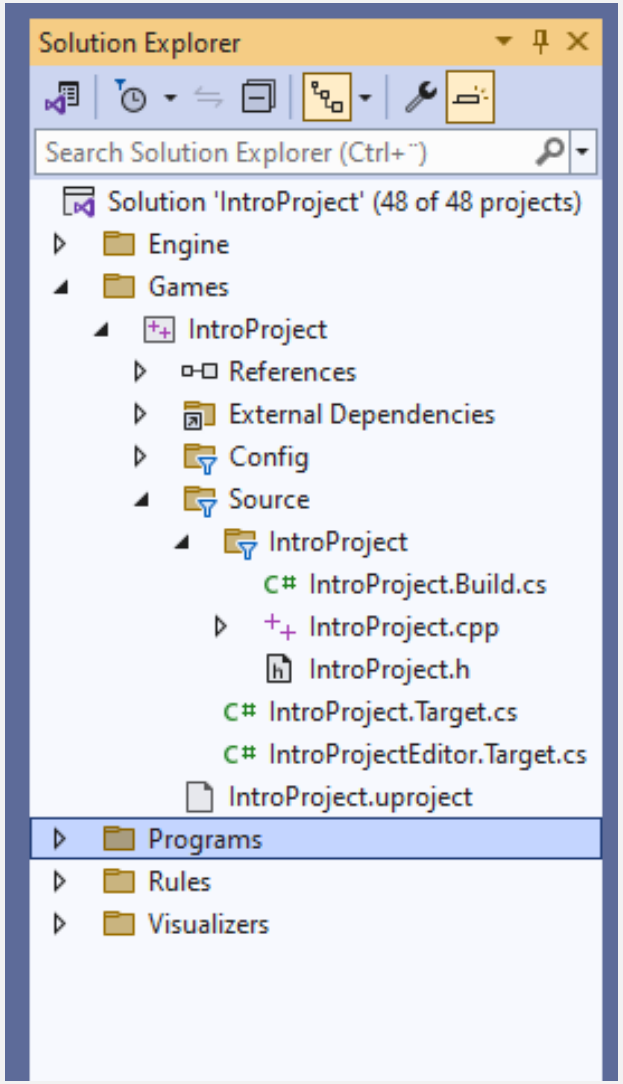


# VISUAL STUDIO INSTALLER

- Modify your visual studio
  - Add game development with c++ module and install



# VISUAL STUDIO – STRUCTURE OF PROJECT




In the solution explorer navigate:

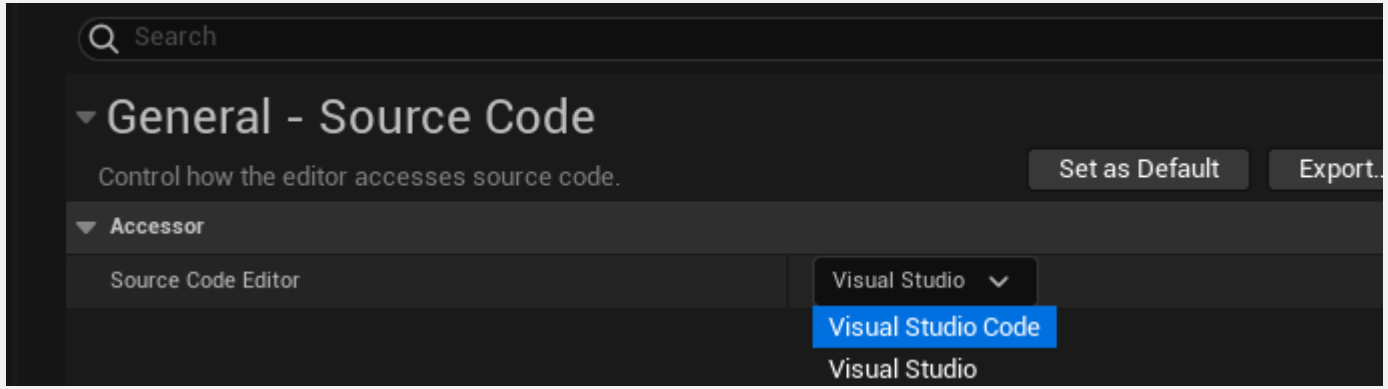
- To Games
- IntroProject
- Source
- IntroProject
  - Here are the c++ files for your game!

# VISUAL STUDIO CODE

You need to install VS Code and add the following modules:

- C / C++ Extension Pack : https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools-extension-pack
- C# for Visual Studio Code: https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csharp

In unreal Edit -> editor preferences menu:



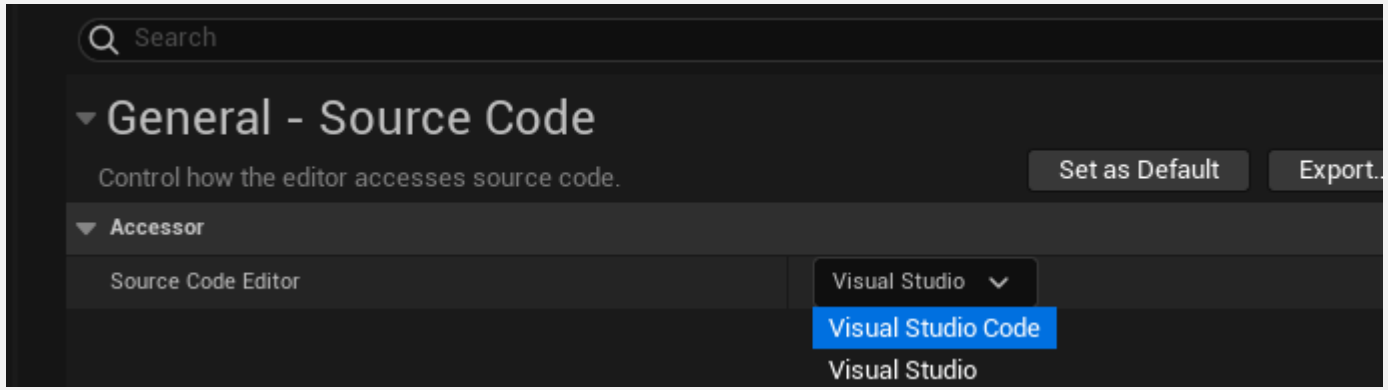

Next you need to generate your studio project files.

# VISUAL STUDIO CODE

The following Instructions can be found here: https://docs.unrealengine.com/5.0/en-US/setting-up-visual-studio-code-for-unreal-engine/

In your files

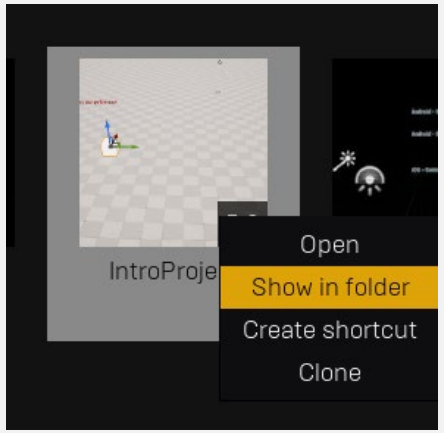
- C / C++ Extension Pack : https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools-extension-pack
- C# for Visual Studio Code: https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csharp



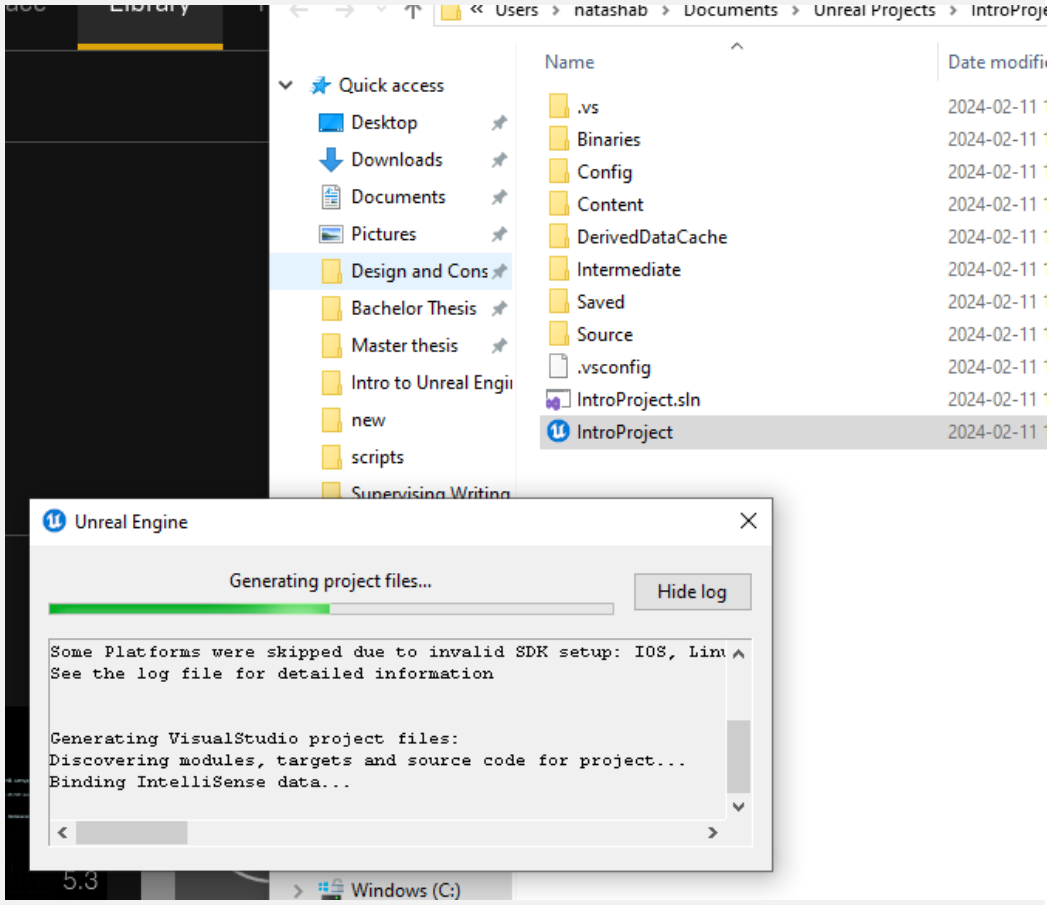

Next you need to generate your studio project files.

# VISUAL STUDIO CODE

Navigate to the project files (from epic games or manually search)




- Right-click on the Uproject file
- Select Generate Project Files.
  - Wait a bit after done, sometimes its still busy
    - If you cannot open the project after this!



# VISUAL STUDIO CODE

Now you need to install the build tools

VS Code uses the Microsoft Visual C++ compiler (MSVC) toolset on Windows.

1. Open the Microsoft Visual Studio Downloads page: https://visualstudio.microsoft.com/downloads/
2. Scroll down to the **All Downloads** section, then expand **Tools for Visual Studio**.
3. Click the **Download** button next to **Build Tools for Visual Studio 2022** and install it.

For MAC - Unreal Engine uses LLVM/Clang as its compiler toolset on MacOS. Refer to Microsoft's documentation on Using Clang for Visual Studio Code for full instructions on how to install and enable it.

For LinuX:

1. Open a Terminal and run the following command: sudo apt install clang
2. Run SetupToolchain.sh. This is located in your Engine's install directory under Build/BatchFiles/Linux
3. Run GenerateProjectFiles.sh to build your VS Code workspace.

# VISUAL STUDIO CODE

Now you need to install the build tools

VS Code uses the Microsoft Visual C++ compiler (MSVC) toolset on Windows.

1. Open the Microsoft Visual Studio Downloads page: https://visualstudio.microsoft.com/downloads/
2. Scroll down to the **All Downloads** section, then expand **Tools for Visual Studio**.
3. Click the **Download** button next to **Build Tools for Visual Studio 2022** and install it.

For MAC - Unreal Engine uses LLVM/Clang as its compiler toolset on MacOS. Refer to Microsoft's documentation on Using Clang for Visual Studio Code for full instructions on how to install and enable it.

For LinuX:

1. Open a Terminal and run the following command: sudo apt install clang
2. Run SetupToolchain.sh. This is located in your Engine's install directory under Build/BatchFiles/Linux
3. Run GenerateProjectFiles.sh to build your VS Code workspace.

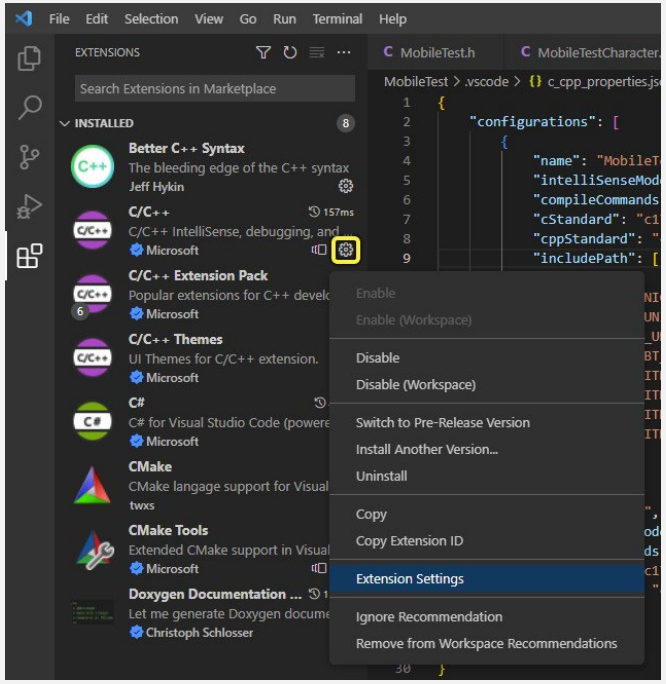
# VISUAL STUDIO CODE

In the Explorer, open .vscode/c_cpp_properties.json.

- Add the includePath property as follows:

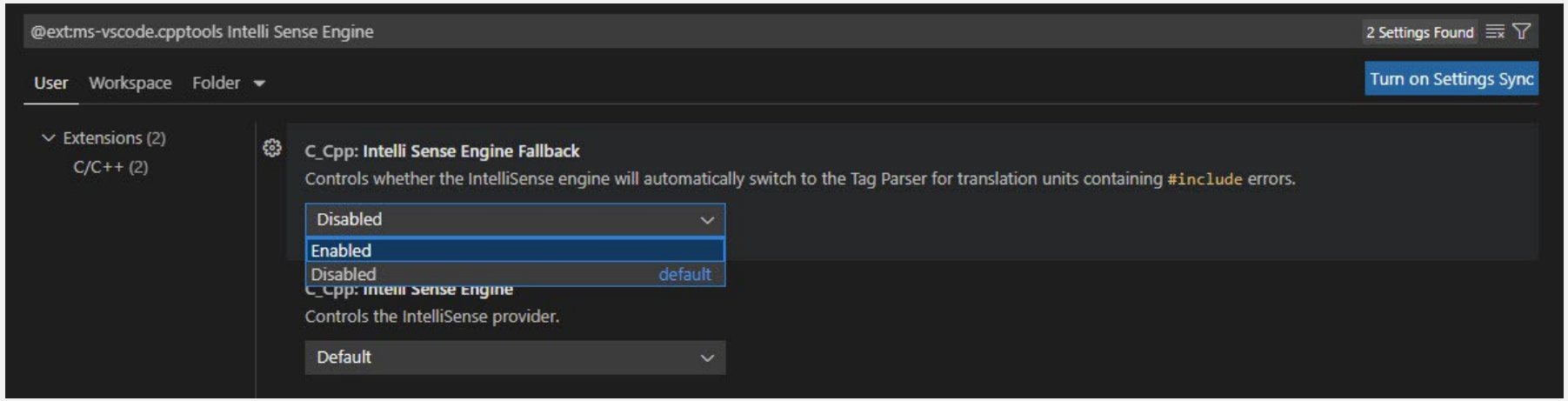
"includePath": [ "${workspaceFolder}\\Intermediate**", "[Workspace Folder]\\Plugins**", "[Workspace Folder]\\Source**" ],

Open the Extensions panel. Click the gear icon for the C/C++ extension, then click Extension Settings.



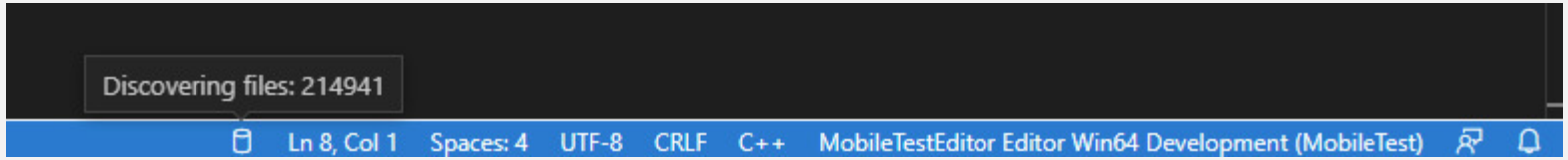
# VISUAL STUDIO CODE

Locate the entry for **C_Cpp: IntelliSense Engine Fallback**. Click the dropdown and set it to **Enabled**.




Set your configuration in the status tray to match the name of your configuration in c_cpp_properties.json.

You now see a small database icon in the VSCode status tray (blue bar) located in the lower-right side of the VS Code window. Mouse over this icon to see IntelliSense's progress parsing your project.



# VISUAL STUDIO CODE

Open the Run and Debug panel by clicking the play button tab on the left side of the window, or by pressing CTRL+SHIFT+D.

Click the dropdown next to RUN AND DEBUG at the top of the panel. Choose the Development Editor configuration for your project. You would choose Launch TestGameEditor (Development) (IntroProject).

Click the Play button or press F5 to start building your project in Editor mode. The project compiles, and when it finishes it opens Unreal Editor. Make sure you do not already have Unreal Editor open.

After you choose your Build Configuration, you can press F5 or click Run > Start Debugging in the toolbar to build and debug your project regardless of whether you are using the Run and Debug panel

# THE ENGINE

*The Inspector*

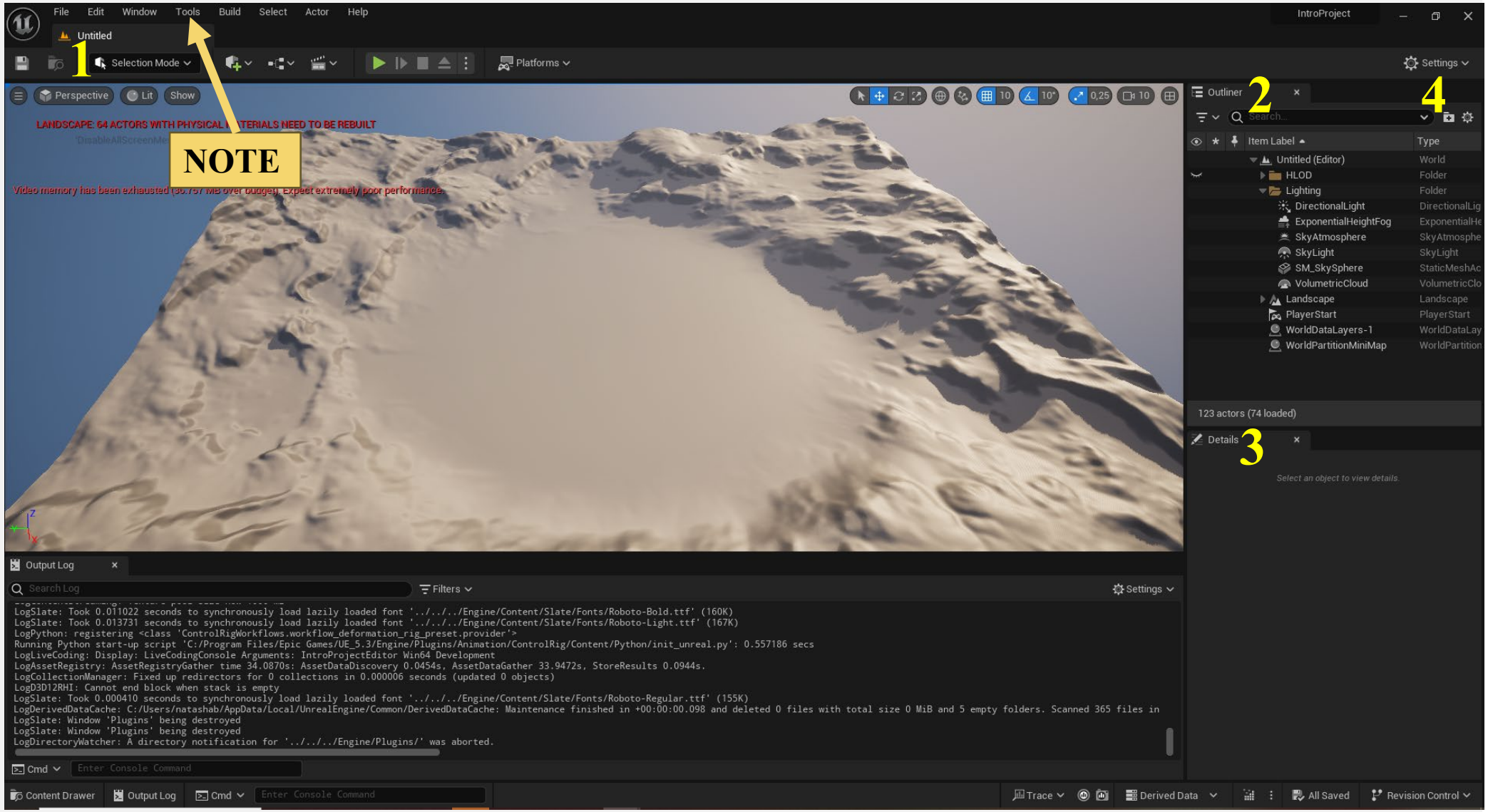
*Folder Structure*

*Building to PC*

*Map Settings*

*Game Mode Controls*

# THE INSPECTOR




1. Current map you are in
   a) UE calls the game levels maps
   b) Is “Untitled since it is not saved yet.
   c) By default, unreal opens an open world map
2. Outliner
   a) All the game objects currently in the map
   b) Can create folders for better structuring
3. Details
   a) Select a game object and this will show you the details of that object
   b) Transforms
   c) Script options etc.
4. Settings
   a) Quick menu – Plugins, Project Settings etc.
   b) Can also do snap settings

Note: Under tools you can enable world partition settings. It will allow you to control the open world map

# THE INSPECTOR


5. Play Controls
   a) Start game
   b) Stop game
   c) Pause game
   d) Cycle through breakpoints
6. Platforms
   a) Select how you want to package your game
7. Add game objects
   a) Any engine content like volumes, lights, etc.
8. World blueprint settings
   a. Set the game mode for this level
   b. Open this level's BP
9. Sequencer
   a. Movie/cutscene controls
10. Output log
    a) Open and close as you want.
    b) I have it pinned open.
11. Content Drawer
    a. Our assets in the project

# FOLDER STRUCTURE

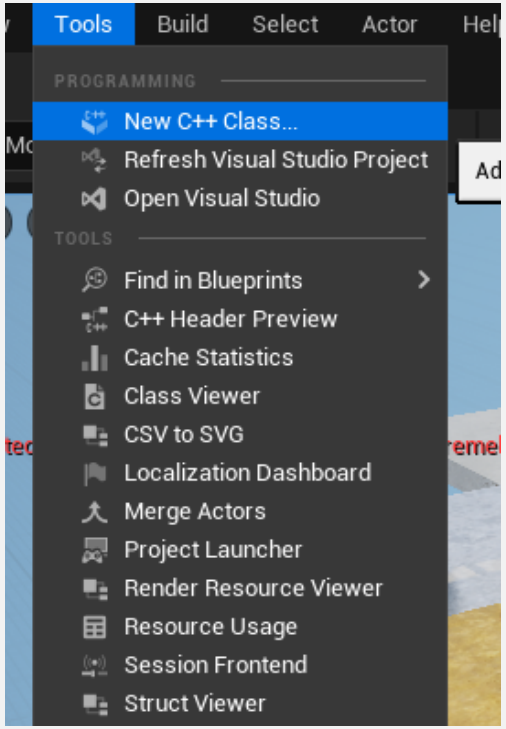
- The content drawer shows you all the available assets in the project


- Since we enabled starter content, there is a folder with all the starter content inside

# ADDING C++ TO A BP PROJECT

You can always change your project to include C++ files

- Since we already have C++ it looks like this:
  - Otherwise the ”refresh VS” and ”Open visual studio” is greyed out
  - To add we simple press ”New C++ Class” (may be called something else if it was BP before)
  - Then sit back and relax while it compiles the C++ files for you!

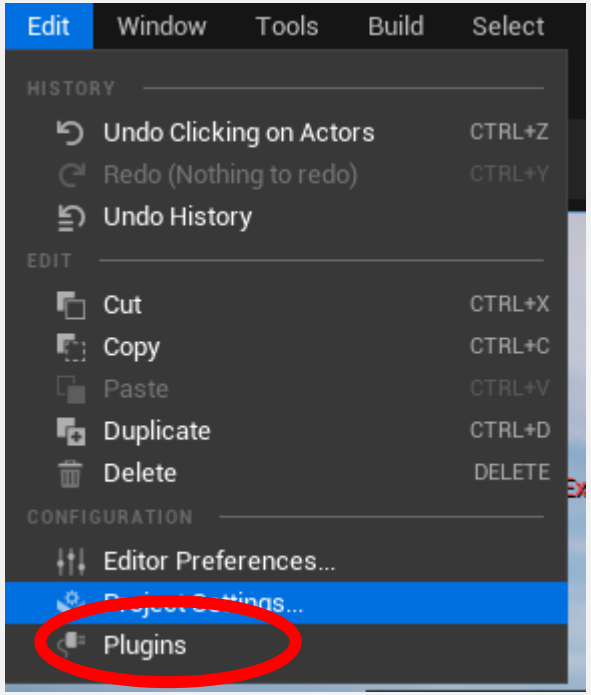


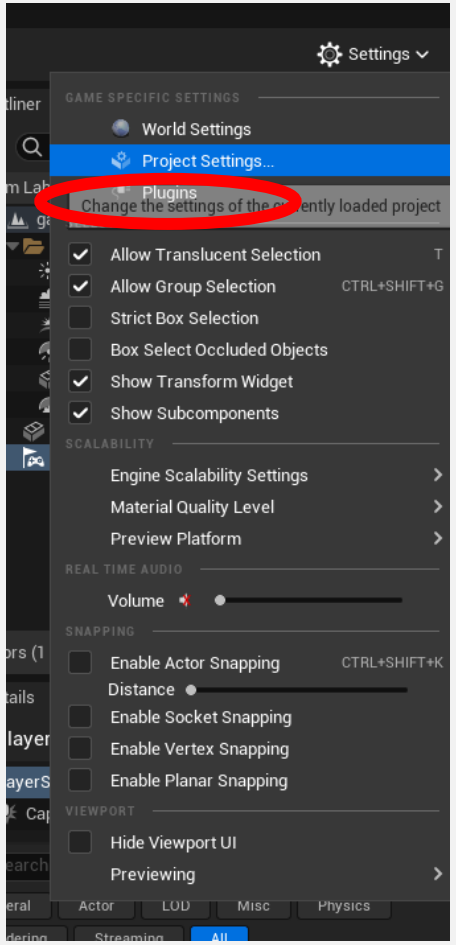
# MANAGE PLUGINS

We do need to setup Visual Studio, in our current version of the engine, this is already enabled

If it is not enabled:

- Select Edit -> Plugins
- OR on the right hand side you can select Settings -> Plugins

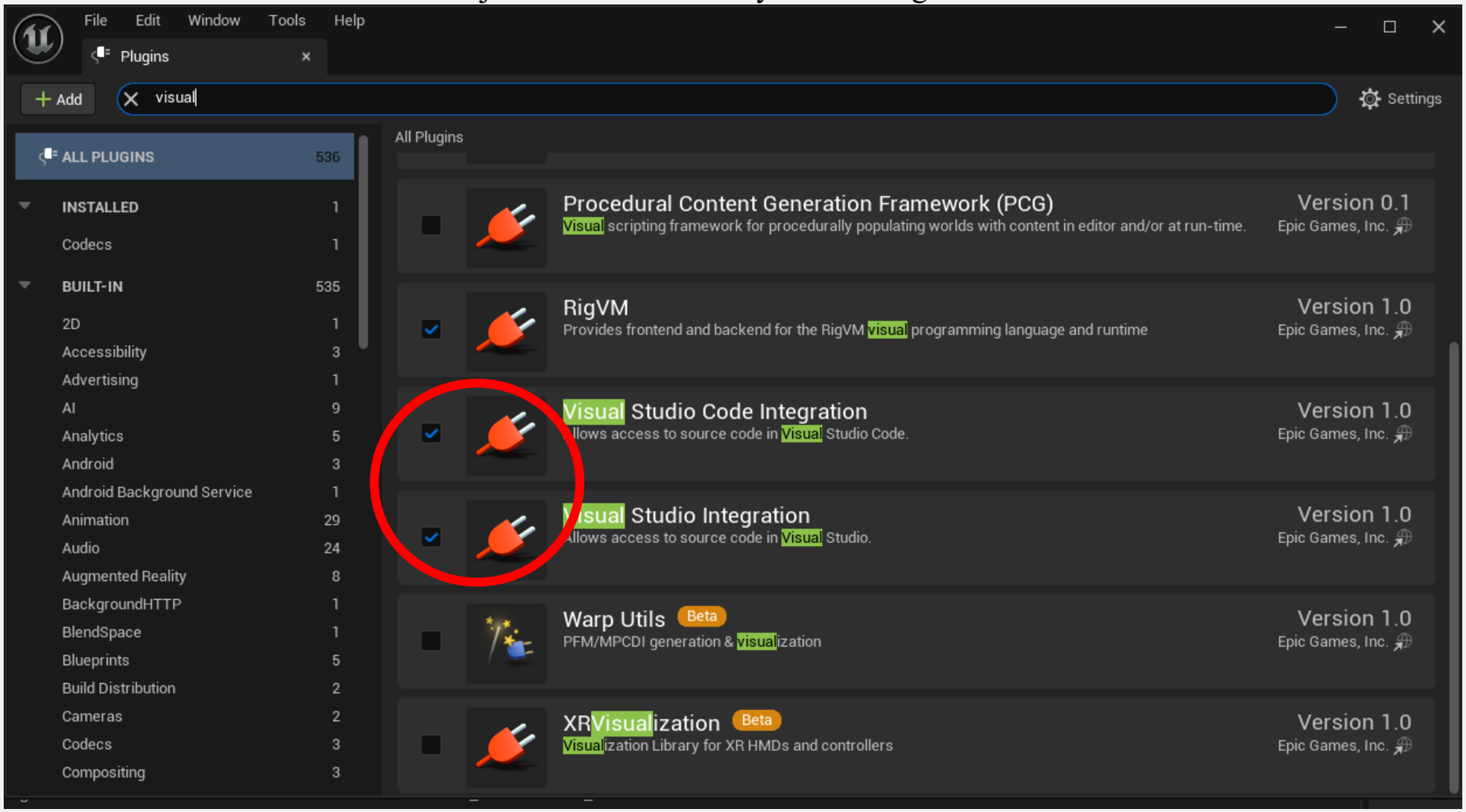


# MANAGE PLUGINS

Type in ”visual” in the search bar:

- Make sure both are enabled or just select which one you are using

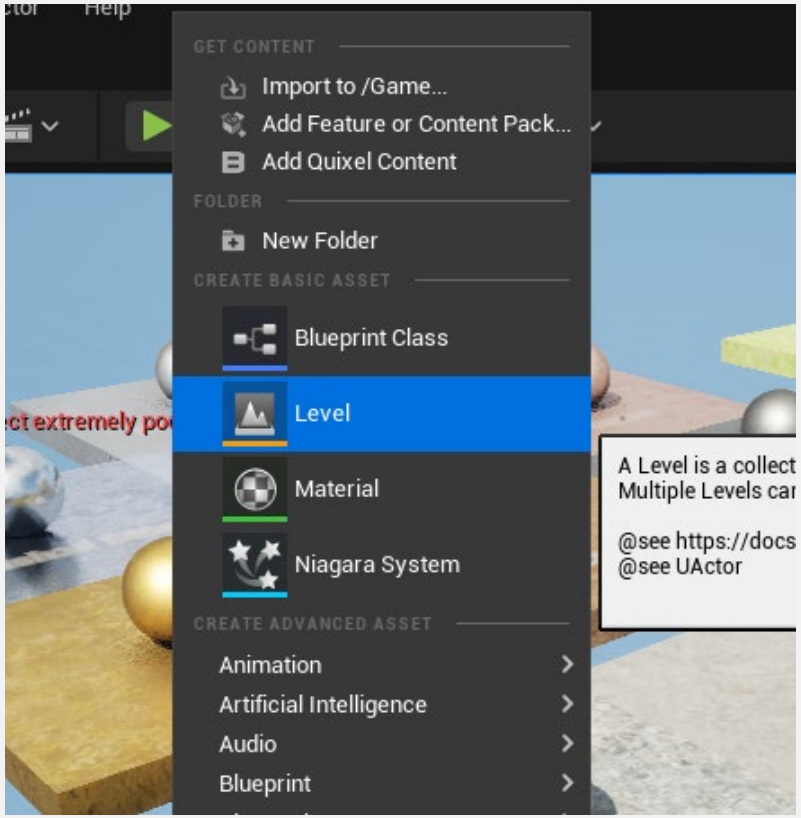


# CREATE A MAP

- To create a new level, there are two options A and B

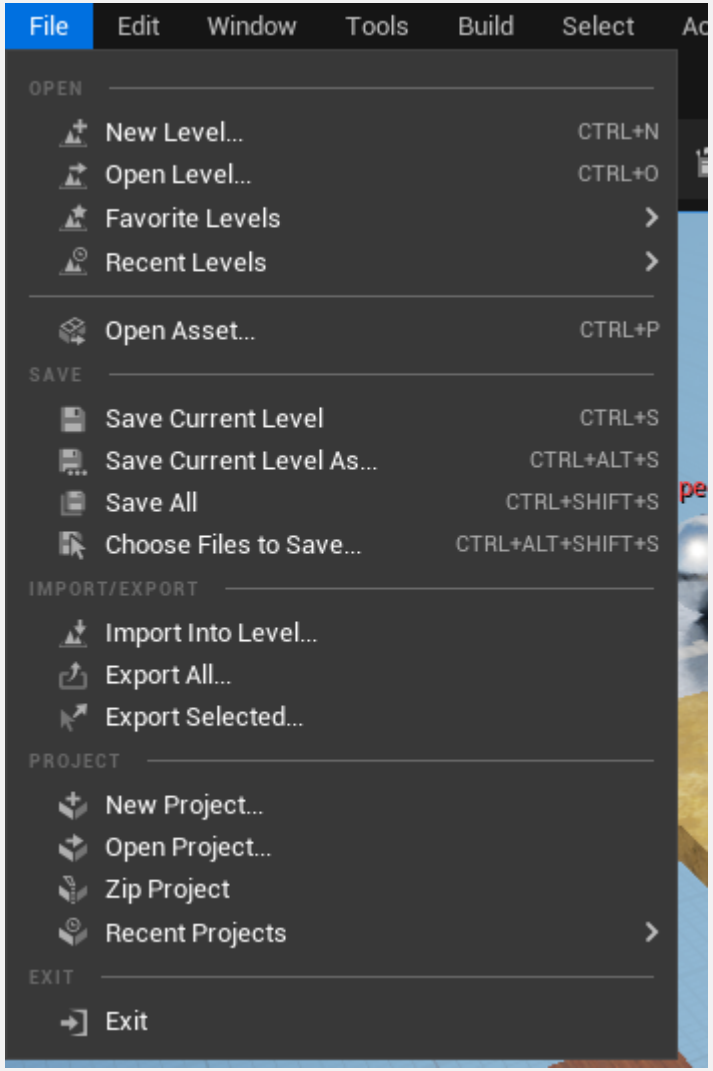
A – left click in the content drawer and select Level

- This option creates an Empty map (no content whatsoever)




B – Go to File -> New Level

- This option lets you choose from a template



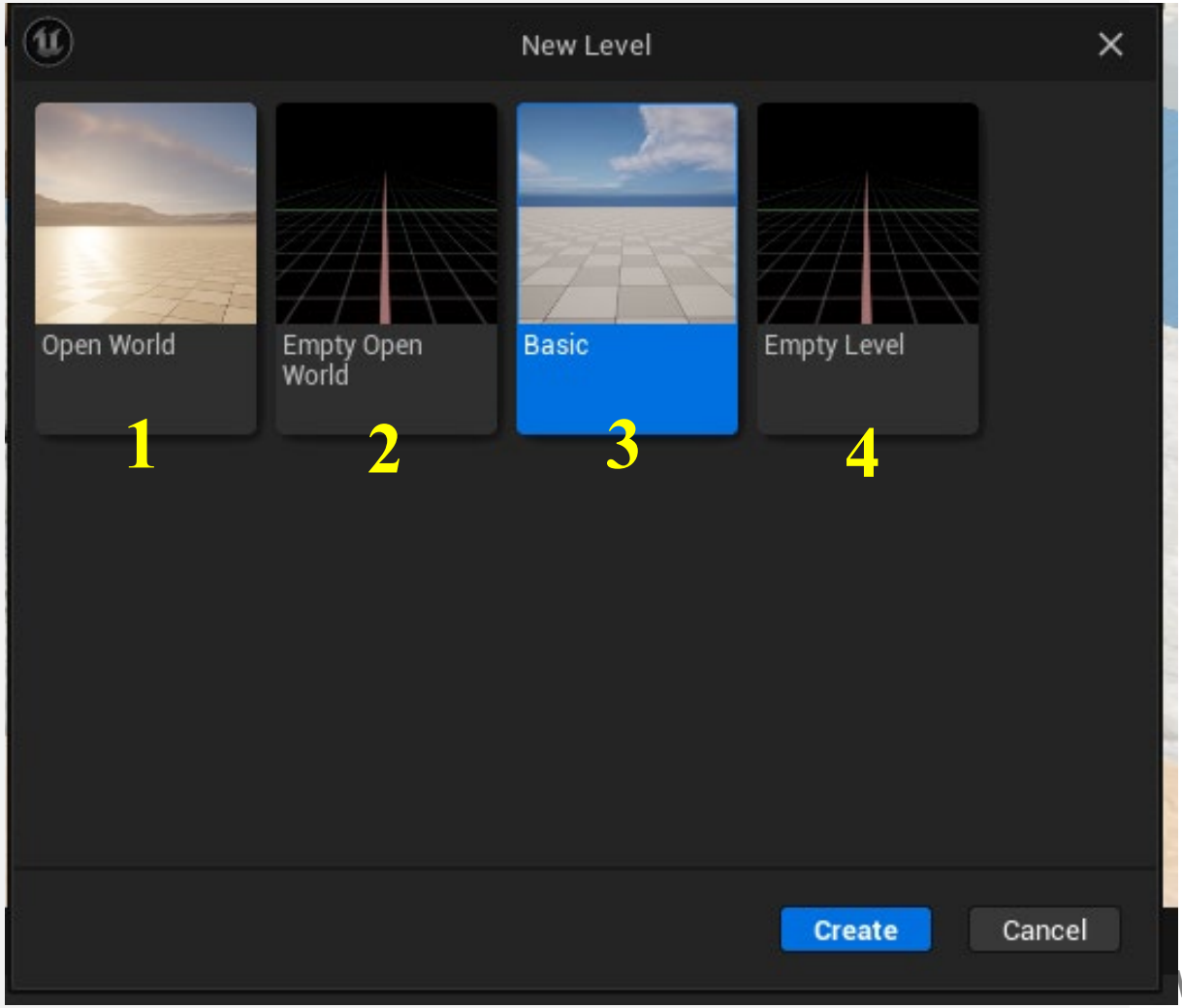

We will select option B

# CREATE A MAP

There are 4 template options

- You can create a template option yourself as well

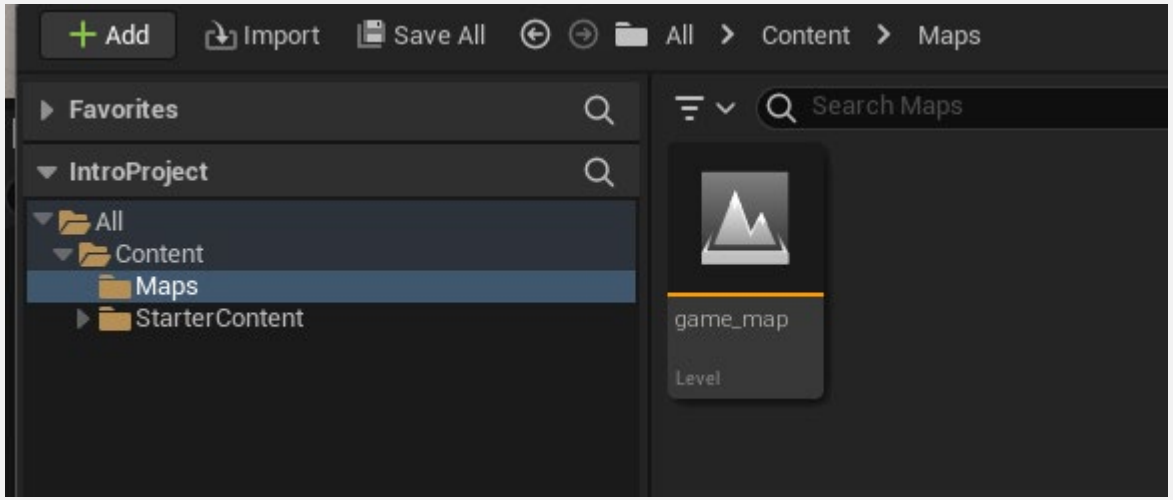
1. Open World
   a) What we saw earlier
   b) Default startup map
2. Empty Open World
   a) World partition system
   b) No Content
3. Basic
   a) Normal template
   b) Everything that Open World has
   c) But its a normal plane setup
   d) No world partition system
4. Empty Level
   a) No content whatsoever
   b) This was option A from earler




We will select No.3, Basic

# SAVE MAP

- First add a folder called Maps
  - Select the + Add button and select Folder from the menu
  - OR Left-click in the Content folder (the right hand side area) then select new folder from the menu

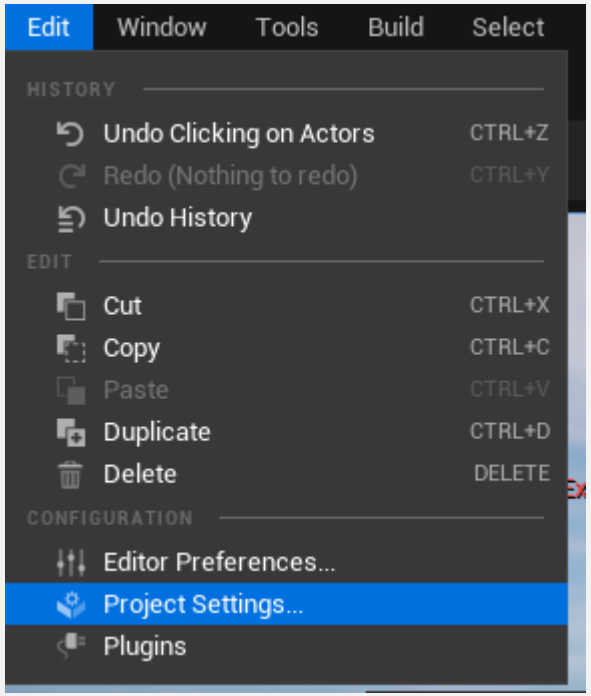


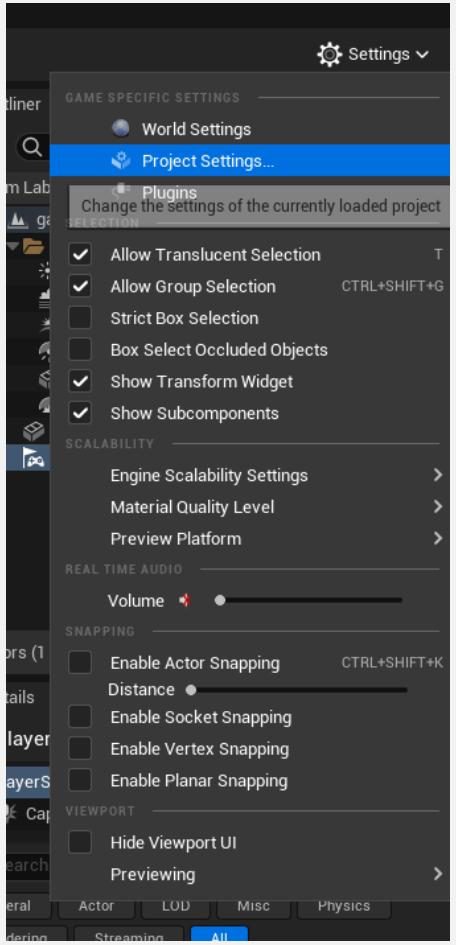

- Then either select the save icon near the ”untitled map” tab or CTRL + S.

# MAP SETTINGS

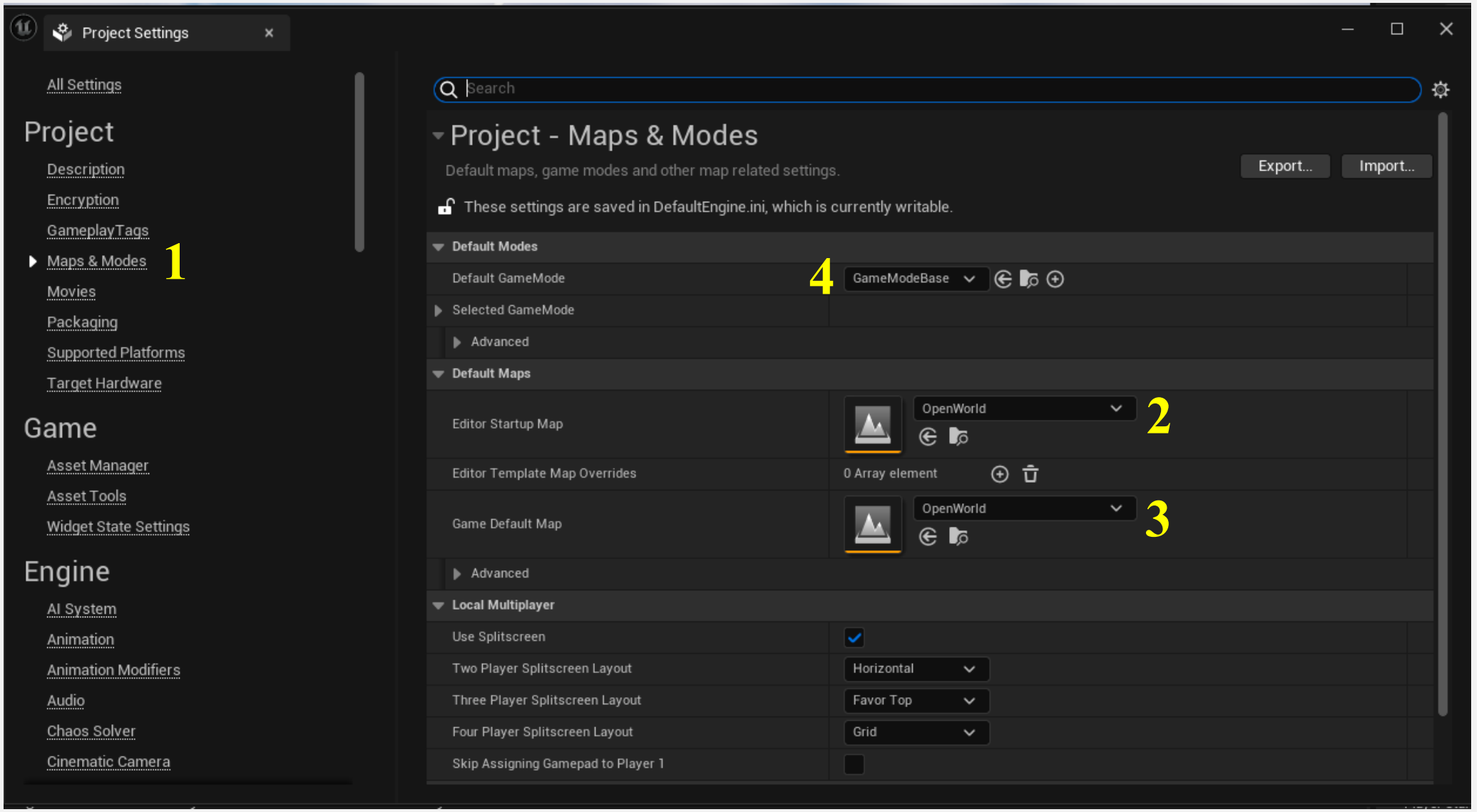
We need to then setup our project to reflect our new map.

- Select Edit -> Project Settings
- OR on the right hand side you can select Settings -> Project Settings



# MAP SETTINGS



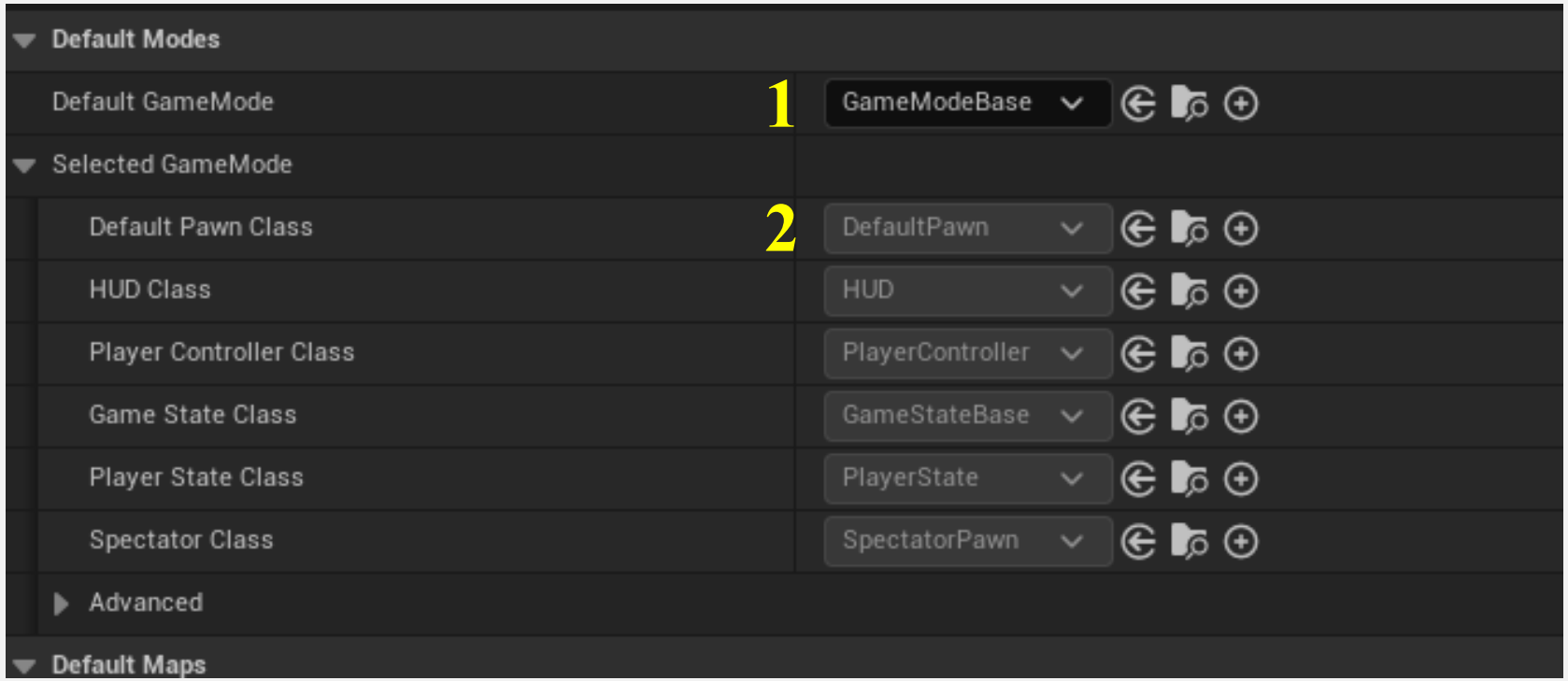

1. Navigate to Maps & Modes under PROJECT
2. Change Startup Map
   - This is the map with which the editor will open whenever you open the project
3. Change Game Default Map
   - This is the map the game will start with, when you build the game
   - Usually the menu_map
   - In advanced settings you can select which maps you want to package with the project
4. Select Game Mode
   - See next slide

# GAME MODE CONTROLS

In Game Mode, we can select what which classes and blueprints will be active when we play.

We will only care about the following:

1. Select your custom Game Mode (we will do this later)
2. Set Default Pawn – If we have a Player Start then this class/bp will be spawned and controlled. These are the controls spawned and enabled.



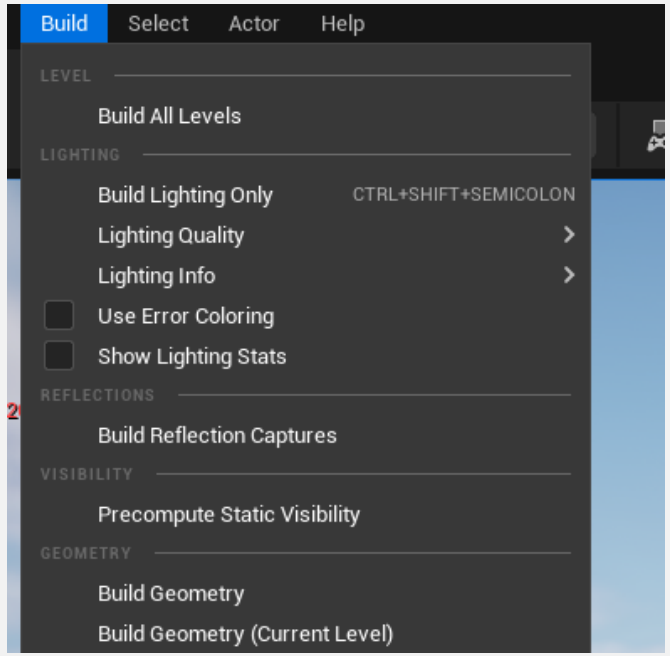
# BUILDING YOUR PROJECT

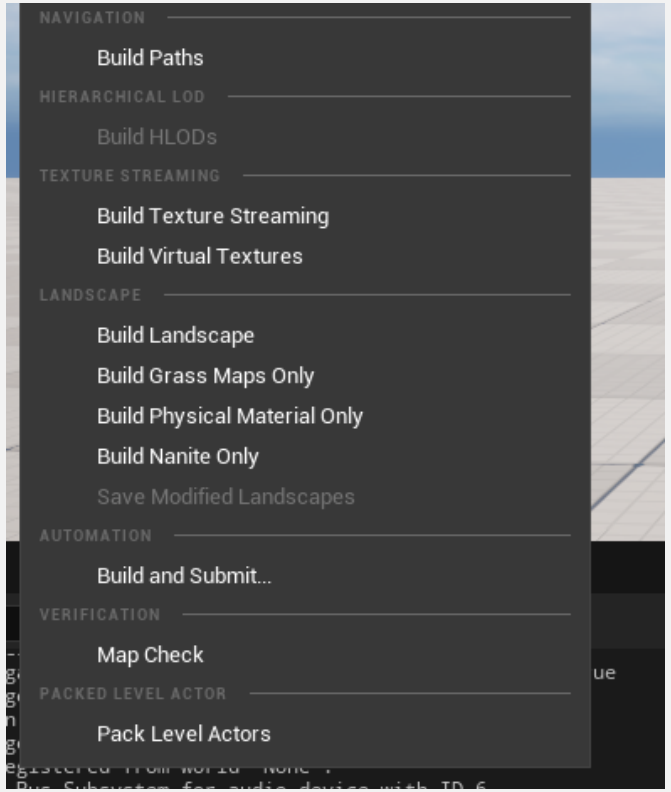
Now we are ready to build our project. (In ue we call this packaging)

But first, we need to build the levels

- Warns you in the top left of the viewport
- You can get away with just building the lighting
- If you package the project without building the lighting

you wont know what the scene looks like (could be complete darkness for all you know)

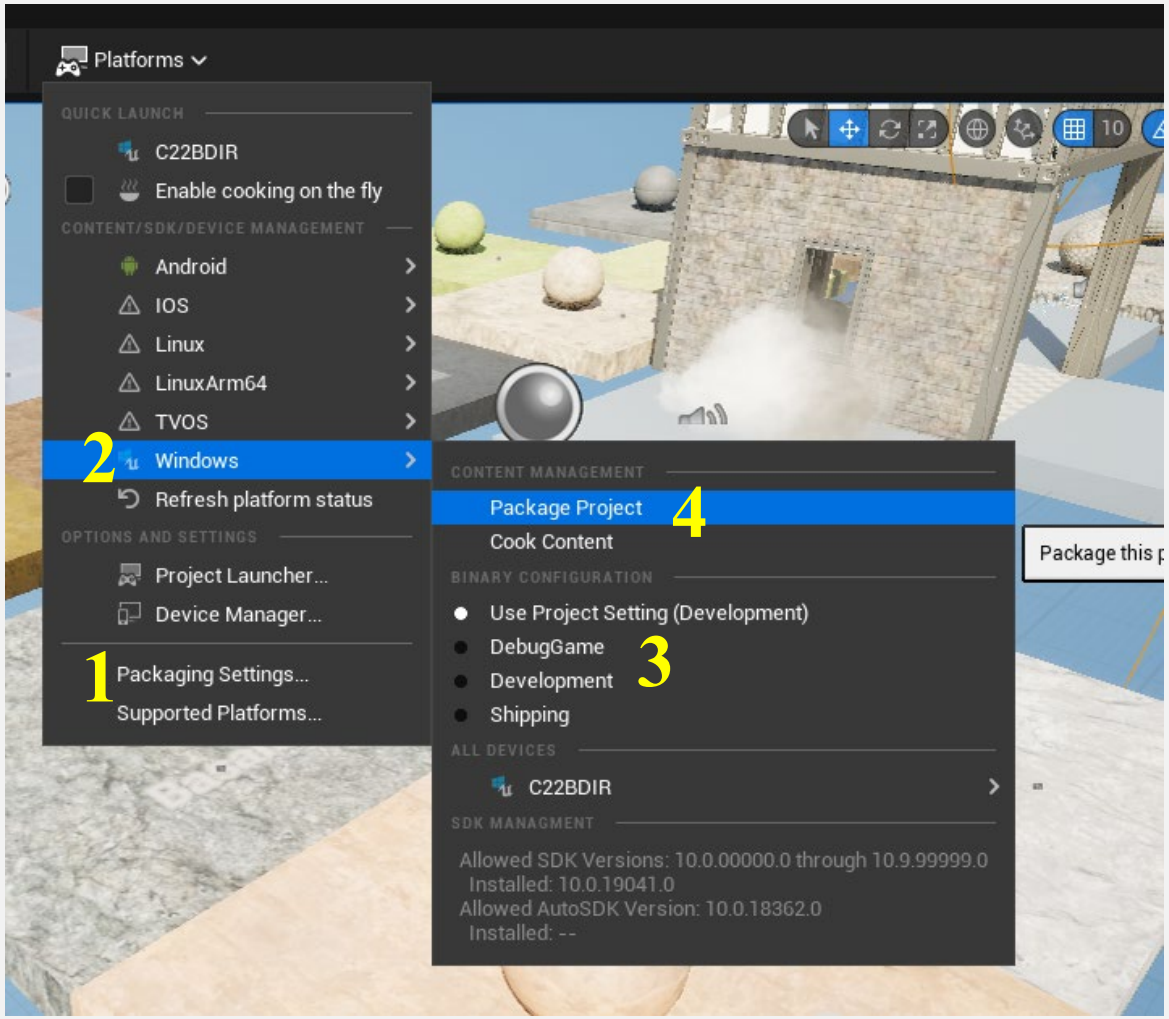


# BUILDING YOUR PROJECT

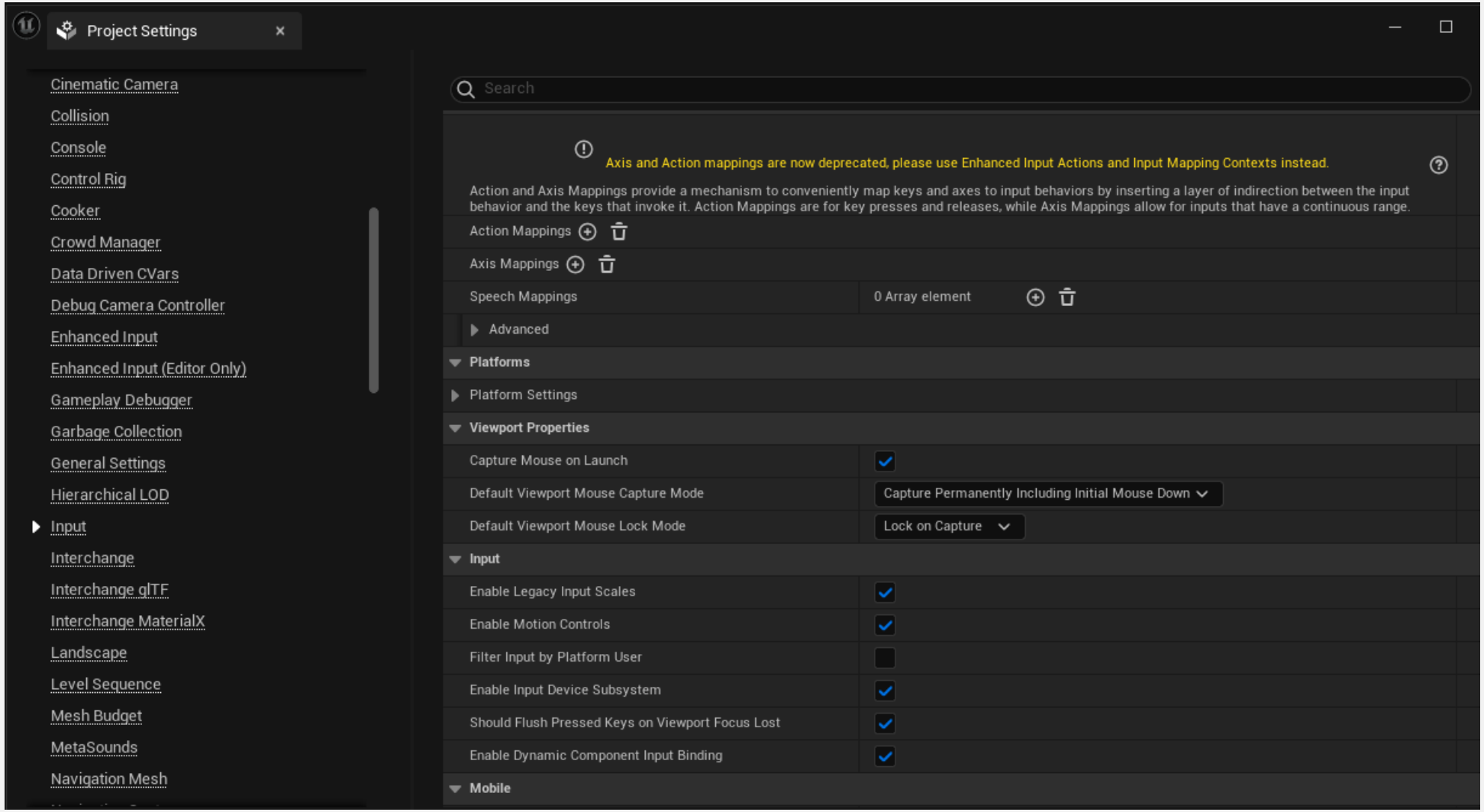
Select Platforms:

1. Packaging Settings – for those who want custom chunks etc.
2. Select Windows
3. Select Development
4. Select Package Project



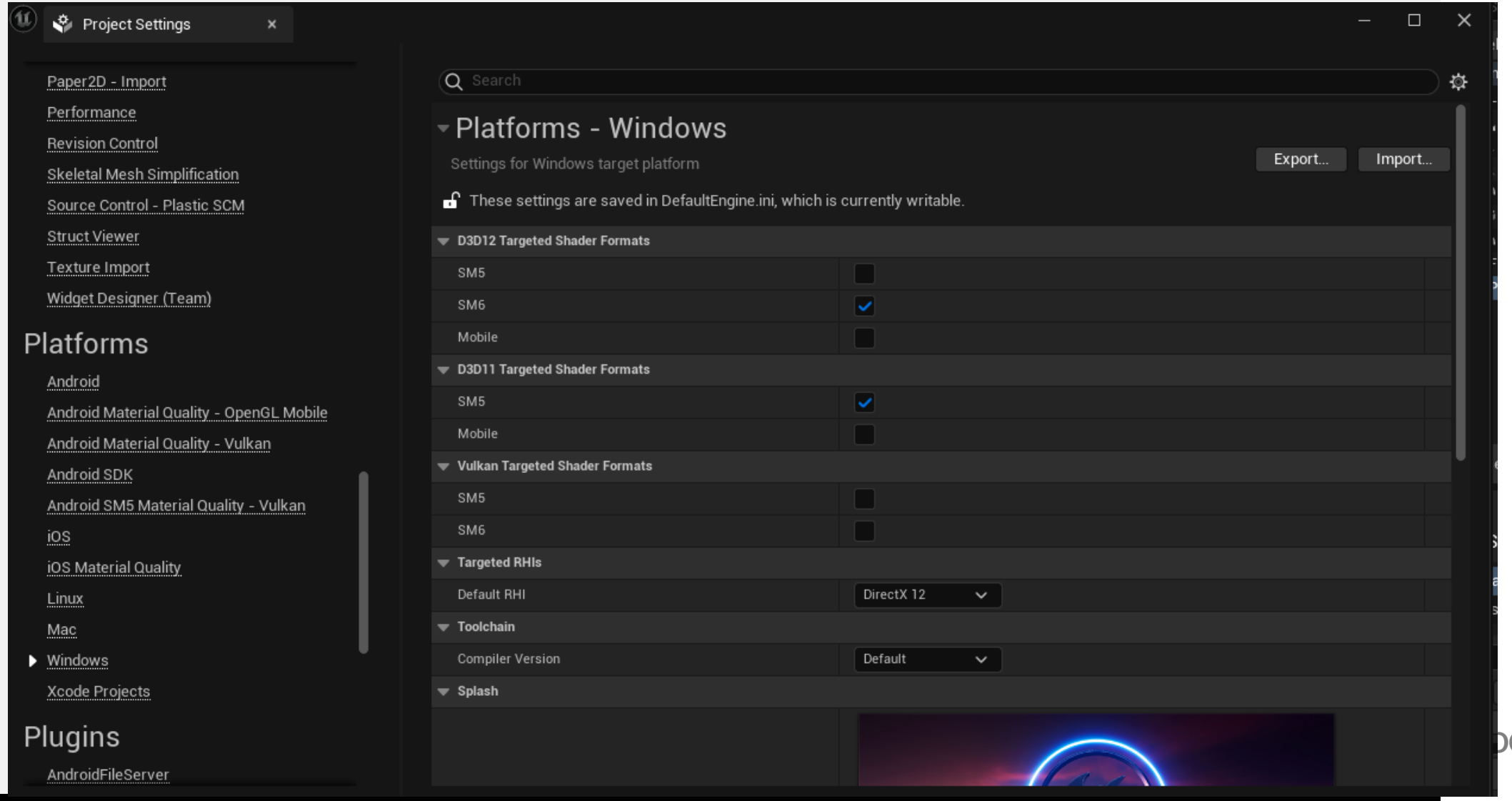
# INPUT SETTINGS

(Engine -> Input) Old Input System  - This is where Ue4 ( < 5.3) sets the Action Mappings (jump, interact) and Axis Mappings (MoveForward, MoveRight, LookUp, LookRight)



# WINDOWS PACKAGE SETTINGS

Here you can set the rendering pipelines, shader settings, game icons, splash screens etc.



# WORKING IN THE SCENE

*Toolbar*

*Movement*

*Shortcuts*

# TOOLBAR AND MOVEMENT

- WASD to move the camera around + hold right mouse button to rotate
- Right-click + Mouse wheel scroll to change the speed of the camera movement
- Mouse wheel scroll to snap the camera in zoom in – out.
- Left-click in the viewport to gain control (if this is not enabled by default in the BP/C++ already)
- Shift + F1 to unlock the cursor from the game viewport
- F8 to move around the scene and select game objects while the scene is running
- Select An object in Outliner and press F to instantly navigate the camera in the viewport to that object


From left to right

1. Viewport options – show stats, change view modes, field of view etc.
2. Perspective controls – Changes the camera view (Top, left, right, bottom, front, back)
3. Lit controls – Change the view in the game viewport (lit, unlit, wireframing etc)
4. Show – selecting which objects in the scene you want to see (turn off skeletal meshes, even post-processing and terrains)

# TOOLBAR AND MOVEMENT


From left to right

1. Shortcut Q - Selecting objects mode
2. Shortcut W – Move the current selected object
3. Shortcut E – Rotate the current selected object
4. Shortcut R – Scale the current selected object
5. Cycles gizmos between world and local coordinates
6. Control how objects snap to the surface
7. Grid snapping – currently in segments of 10
8. Rotation snapping – Move the current selected object
9. Scale snapping – Rotate the current selected object
10. Shortcut mouse wheel - Camera speed
11. Maximizes or restores this viewport – Move between seeing multiple camera views vs the singular view.

# TERRAIN SYSTEM

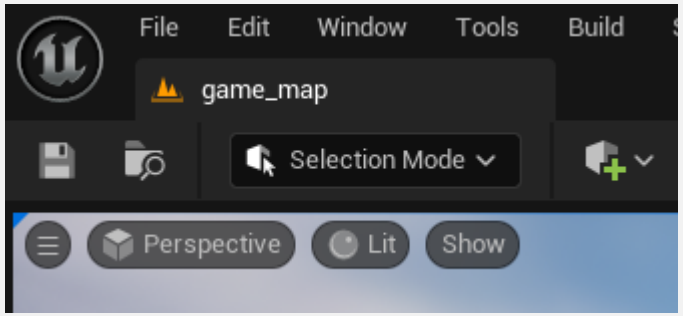
*Building*

*Deleting*

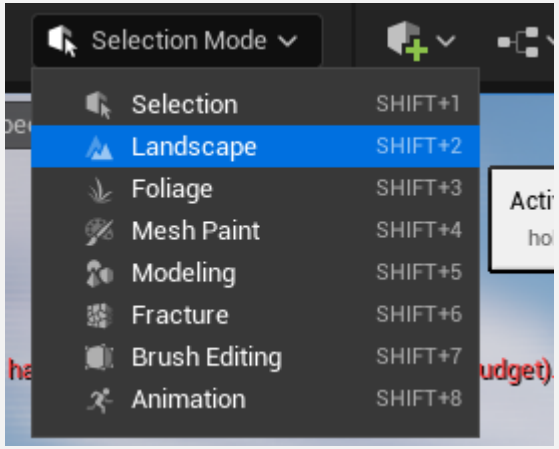
# CHANGE TO LANDSCAPE MODE

We ignored this little detail in the toolbar earlier

- Select the Selection Mode drop down



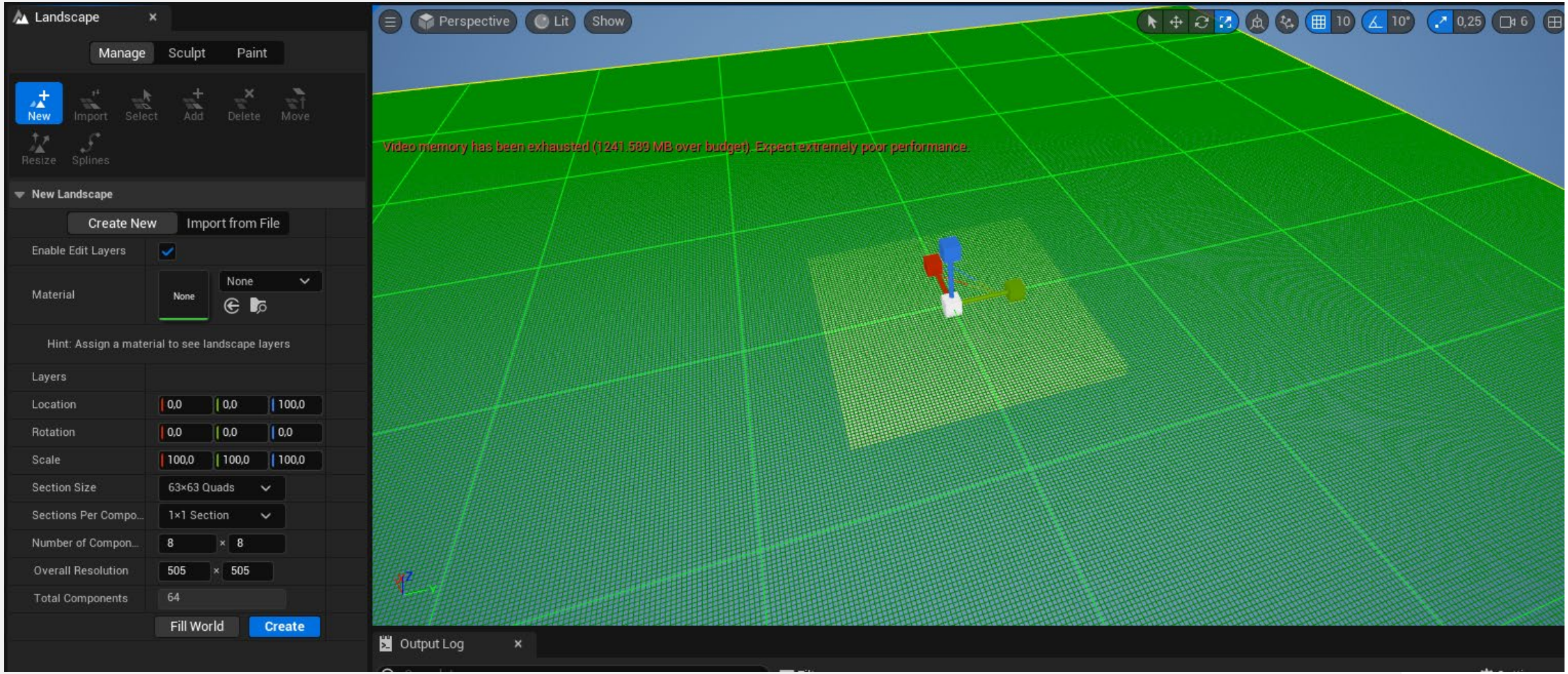

- Select Landscape



# CHANGE TO LANDSCAPE MODE

If we scroll out – using the scroll wheel

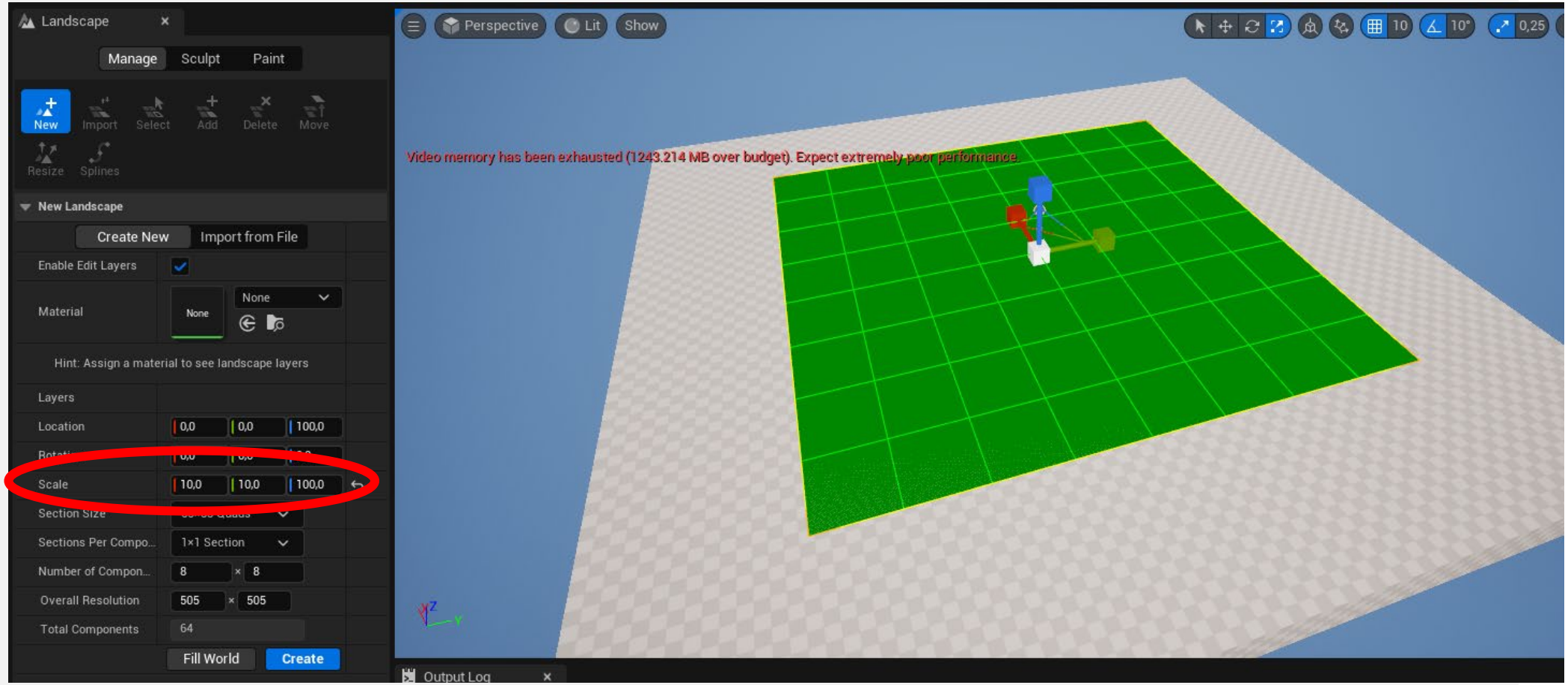
- You can see that our little square from before floating just under a possible terrain we can build.



# EDIT LANDSCAPE

Change the scale to 10 x 10 x 100 (Width x Length x Height)

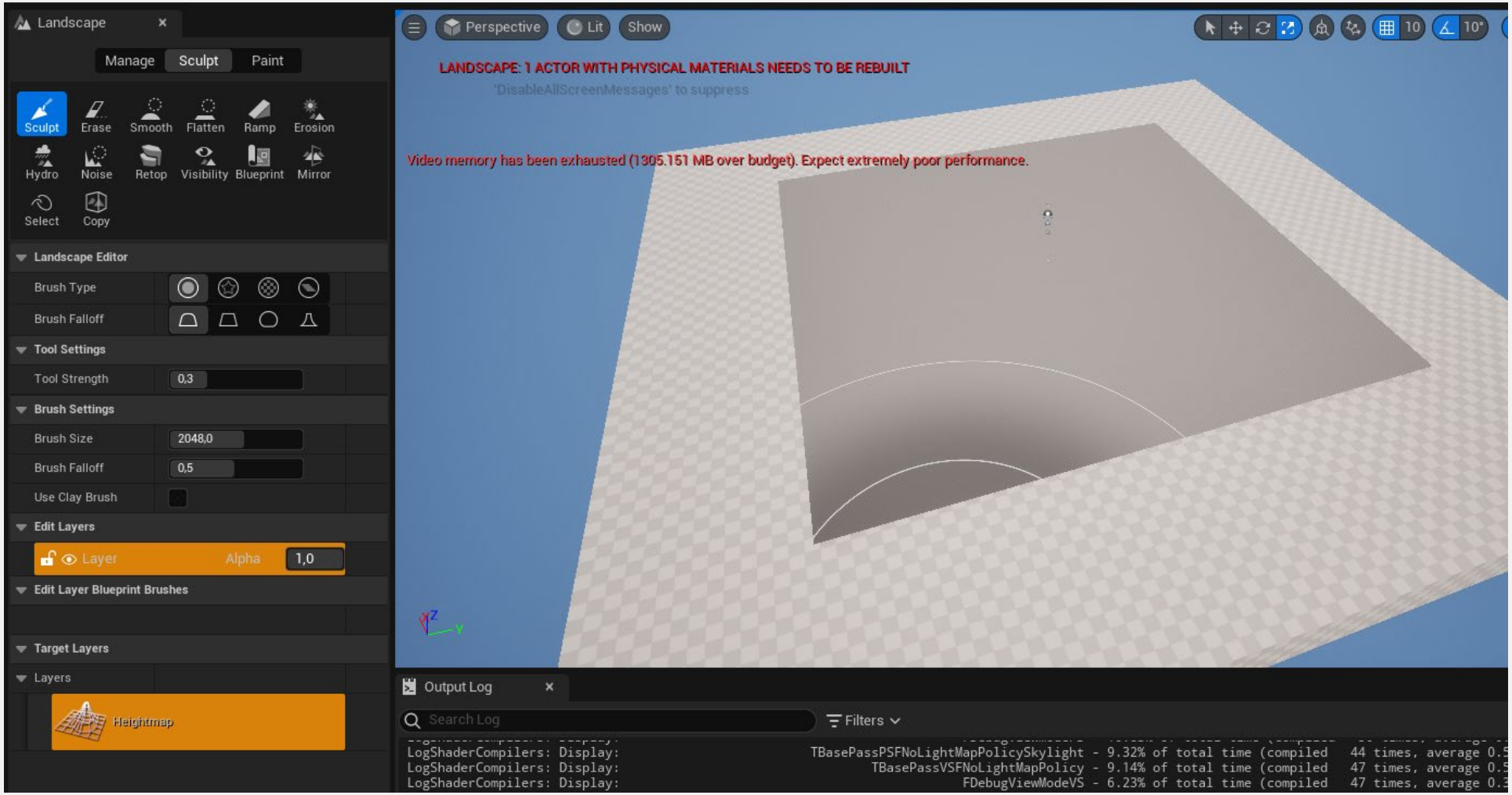
Keep the height/thickness at 100



# CREATE LANDSCAPE

Click Create!

Now we can manage, sculpt and paint the landscape.



# MORE ABOUT LANDSCAPES

In the documentation, there is a handy tutorial

https://docs.unrealengine.com/5.0/en-US/creating-landscapes-in-unreal-engine/

# GENERATED MESHES

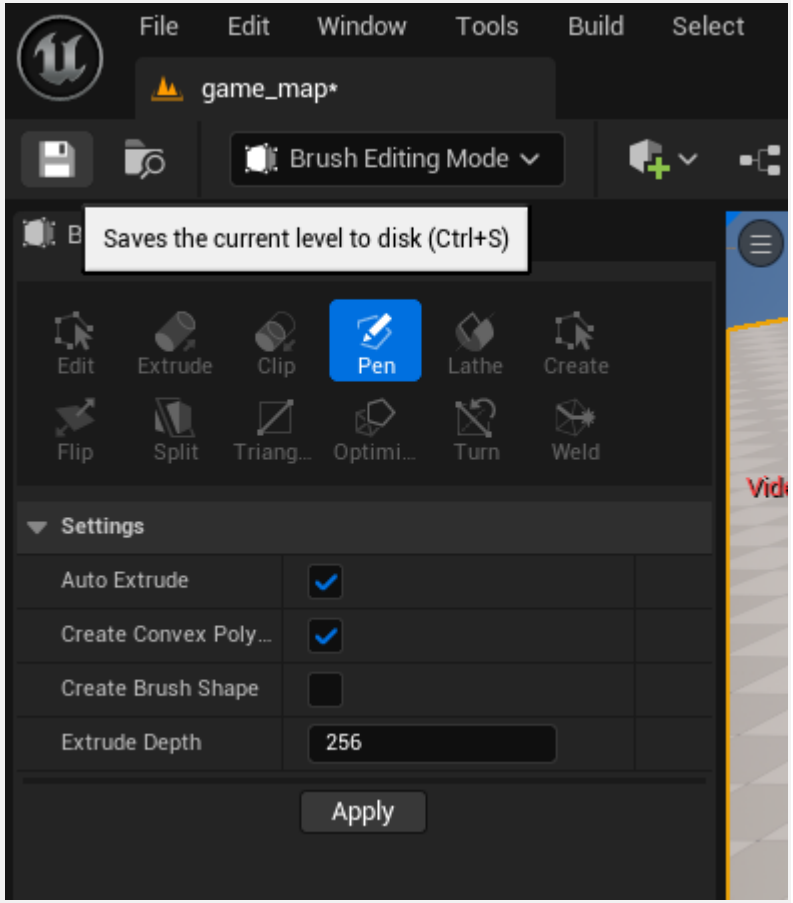
*BSP*

# BINARY SPACE PARTITIONING (BSP)

- The original brush shapes has been removed from ue5.
- This allowed you to manipulate brush shapes to layout a basic scene (level design).
  - Addition
  - Substraction
- You could also turn them into real static meshes later.

It has however changed to be a Selection Mode

- where you can create geometry and manipulate it yourself



# MENU SYSTEM

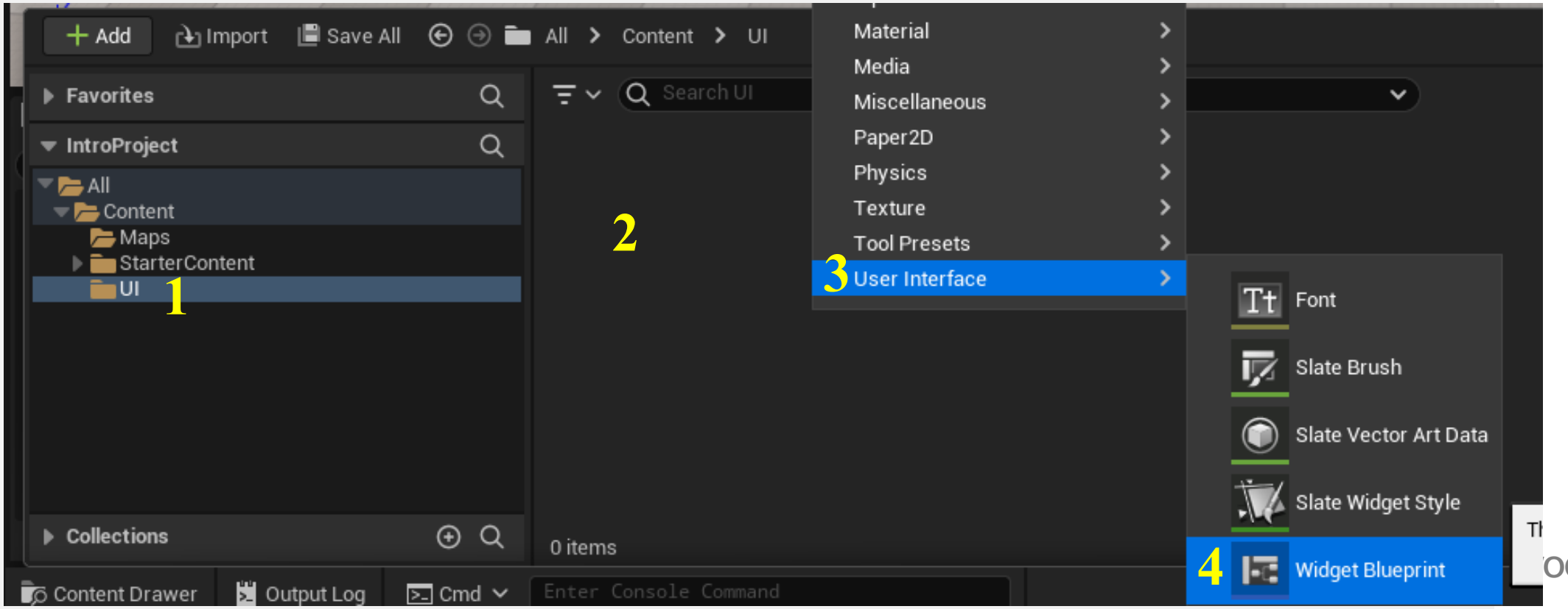
*HUD*

*Menu Screens*

# WIDGET BLUEPRINTS

These are how you create menus and HUDs in Unreal.

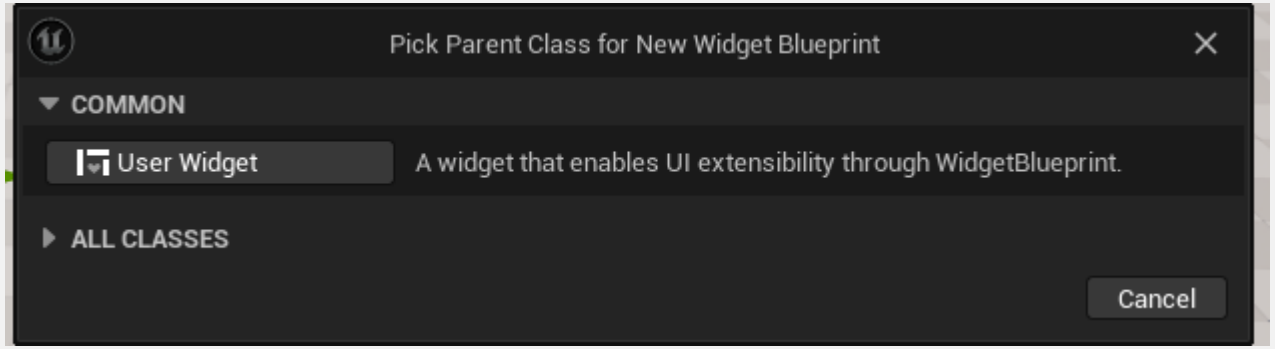
1. Create a new folder, ”UI”
   a) Make sure it’s a child to the content folder!
2. Right-click in the folder
3. Hover over User Interface from the menu
4. Select Widget Blueprint



# WIDGET BLUEPRINTS

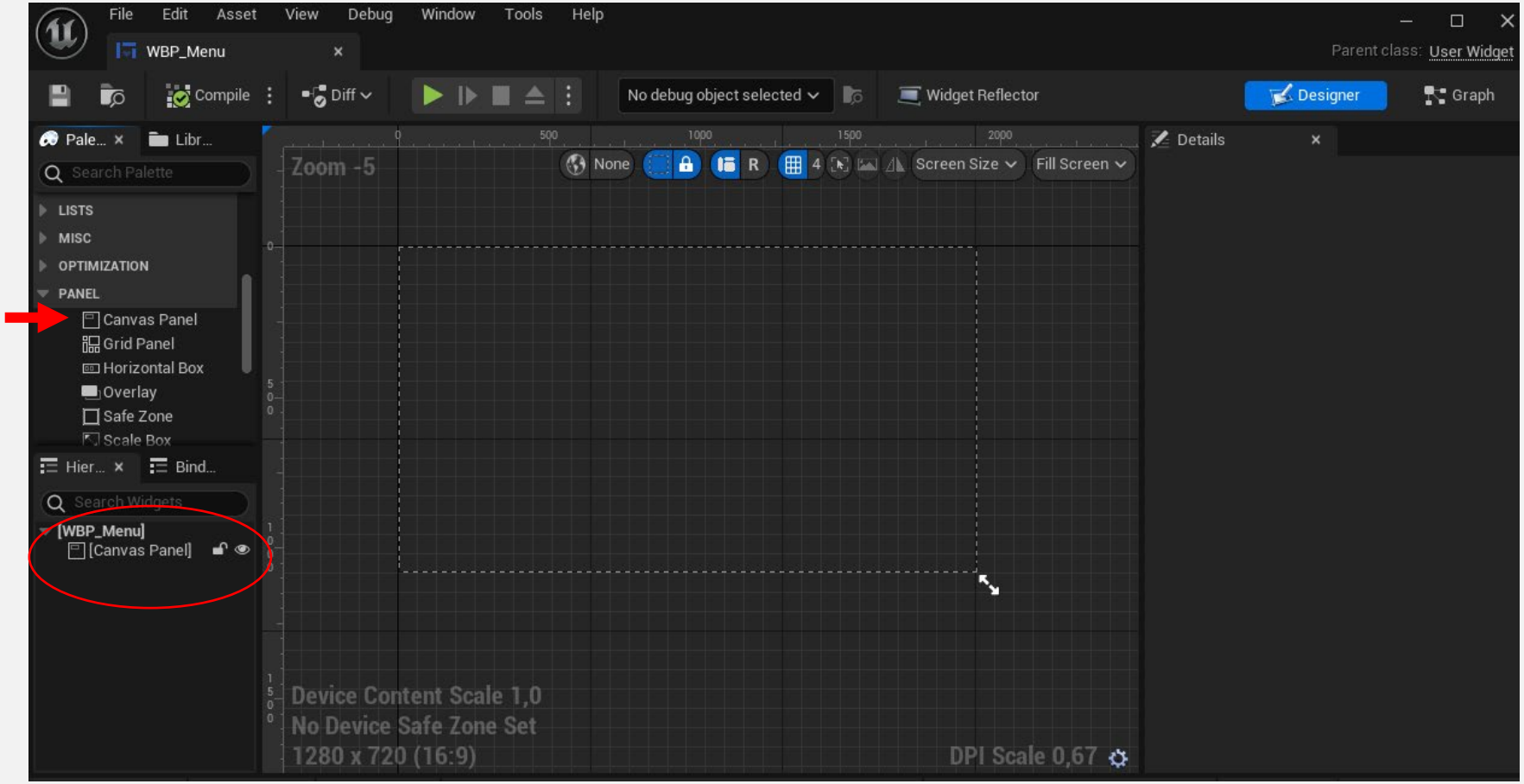
Select User Widget as the parent class

- If you had already created widgets before and want to use one of them as a parent or a c++ created one, then select all classes and search for your parent.

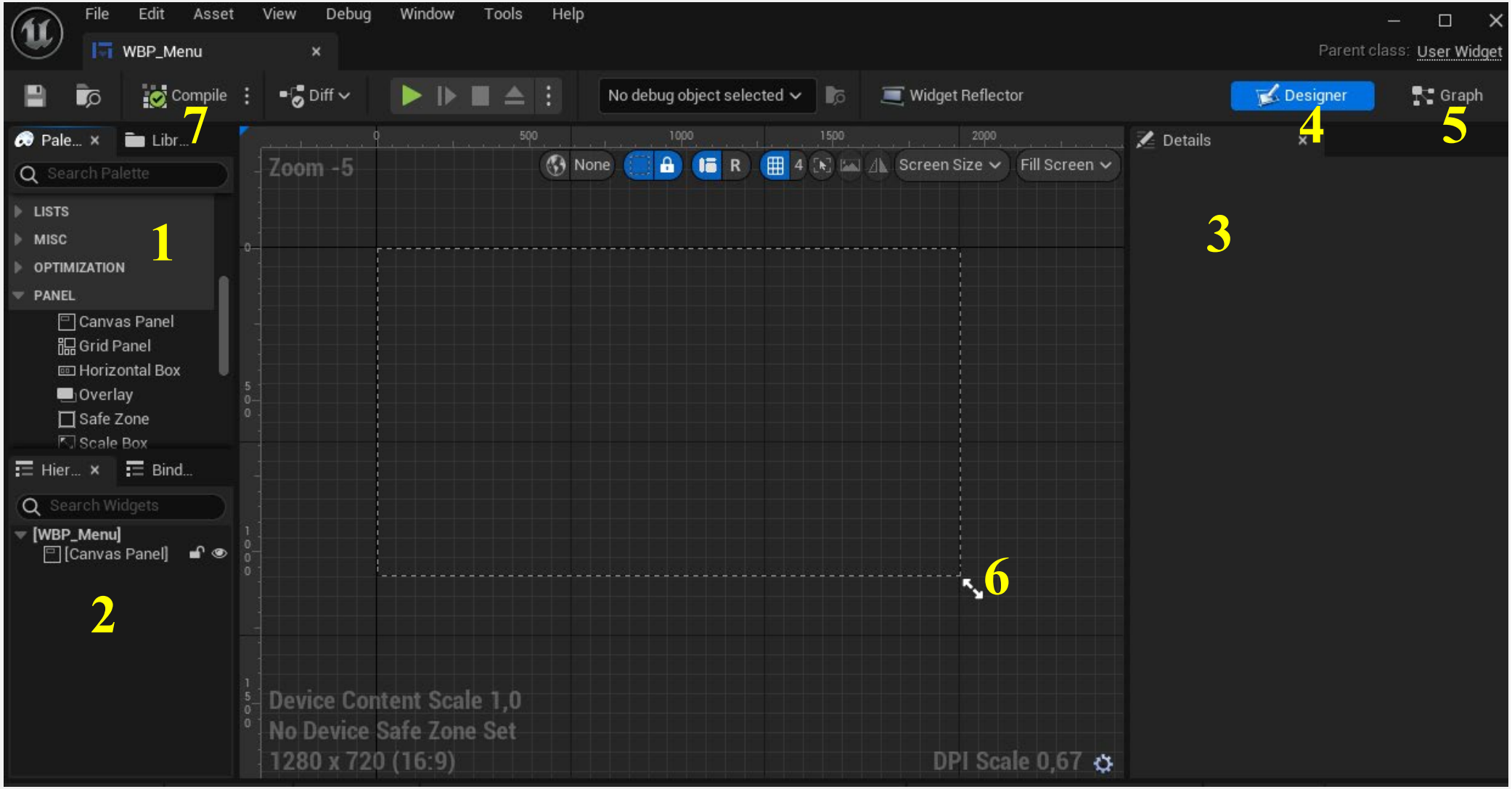


# WIDGET BLUEPRINTS

- Call this widget WBP_Menu. (WBP means widget blueprint)
- Add a canvas panel to the WBP_Menu Hierarchy
  - Drag and drop
  - Now we can do stuff!



# WIDGET BLUEPRINTS




1. Palette
   - UI tools you can use in the menu
2. Hierarchy
   - Same as Outliner for the viewport
   - It contains everything in the menu
3. Details
   - Select on object in hierarchy
   - See the details
   - change information/ settings
4. Designer
   - The view we are currently in
5. Graph
   - Blueprint where we create scripts and behaviours
6. View Screen Sizes
7. Compile + Save!
   - When your in a BP of any kind you need to coompile then save, when you change stuff

# MATERIALS

*Parent and Children*

*Self-Coded Shading*

# LIGHTING

# SOUND

*Sound Cues*

*Master Controls*

# ARTIFICIAL INTELLIGENCE

# WORKING WITH 2D

*Paper Sprites*

# C++

*How to call different objects!*

*Compiling Code*

# NAMING CONVENTIONS

Variables:

- bool for boolean values
- TCHAR for a character
- uint8 for unsigned bytes (1 byte).
- int8 for signed bytes (1 byte).
- uint16 for unsigned "shorts" (2 bytes).
- int16 for signed "shorts" (2 bytes).
- uint32 for unsigned ints (4 bytes).
- int32 for signed ints (4 bytes).
- uint64 for unsigned "quad words" (8 bytes).
- int64 for signed "quad words" (8 bytes).
- float for single precision floating point (4 bytes).
- double for double precision floating point (8 bytes).
- PTRINT for an integer that may hold a pointer

# NAMING CONVENTIONS

Class Names

- U – Objects
- A – Actors
- S – Slate (Widgets)
- I – Interface
- E – Enums
- T - Templates
- F – Anything else really, i.e., Structs / Strings / Vectors etc.

DO NOT:
Name your C++ files with A, U etc. It will be done for you. So just name it, i.e, Player!

Examples:

FName Player;

UClass* PlayerClass;

APawn* Comet;

ACharacter* PlayerOne;

FString PlayerText;

UTexture* WaterTexture;

Swidget* Menu;

# COMPILING CODE

Two Options:

- Live Coding
- VS Compile

Live Coding

- CTRL + S to save the script and then press CTRL + ALT + F11
- VS compile once every now (live coding is temporary until you do a proper compile later, i.e., if you close the project you will need to live code again when opened if you do not do this).

VS Compile

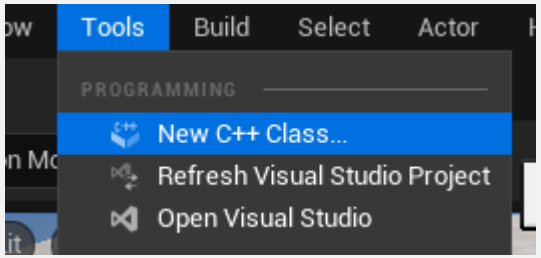
- You will need to close the Project (just the editor!)
- In VS, press Ctrl + B or Ctrl + Shift + B.

# PLAYABLE CHARACTERS

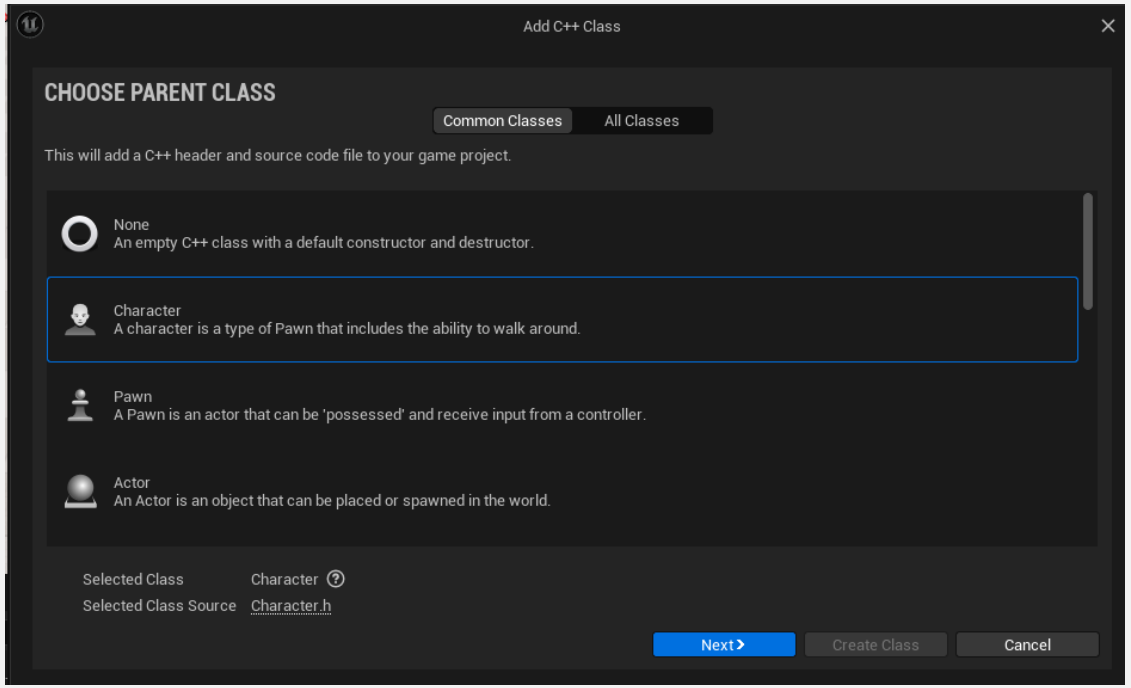
*Third Person*

# CREATE THE SCRIPT

- Since we just made this project, we need to create a new C++ file here

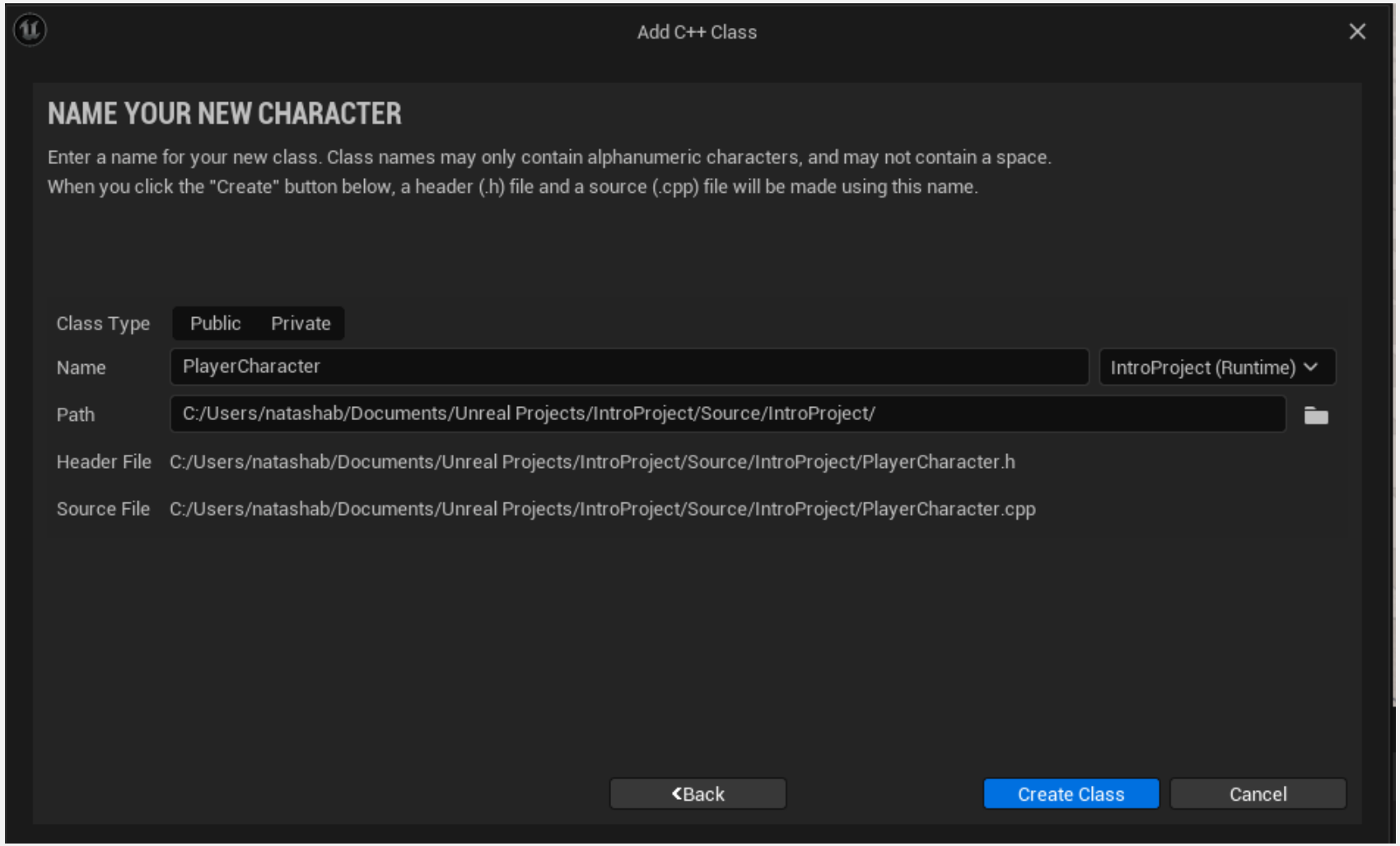



- Select Class Type: Character -> then click next



# CREATE THE SCRIPT

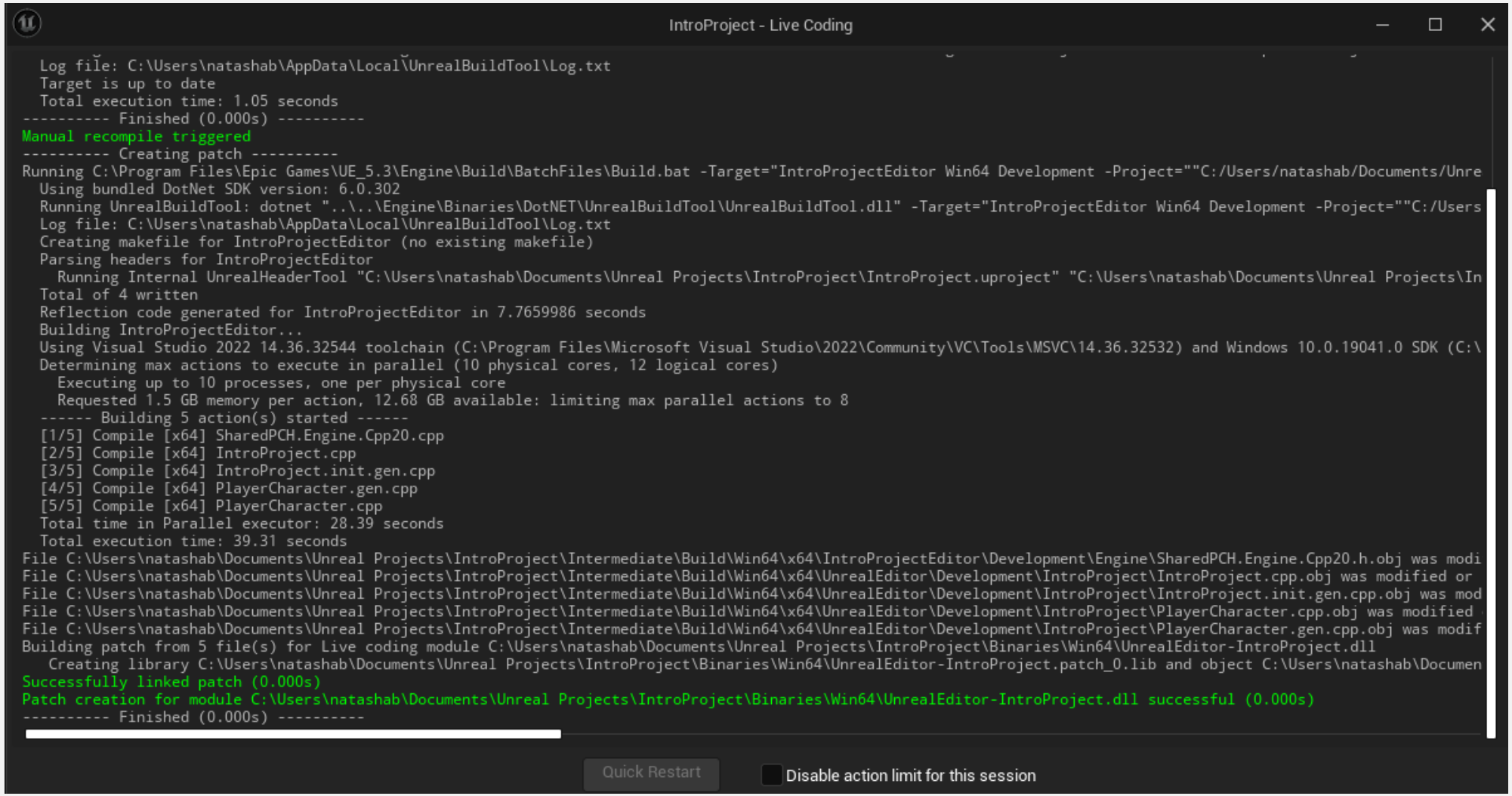
- Name the script ”PlayerCharacter”-> then click ”Create Class”



# CREATE THE SCRIPT

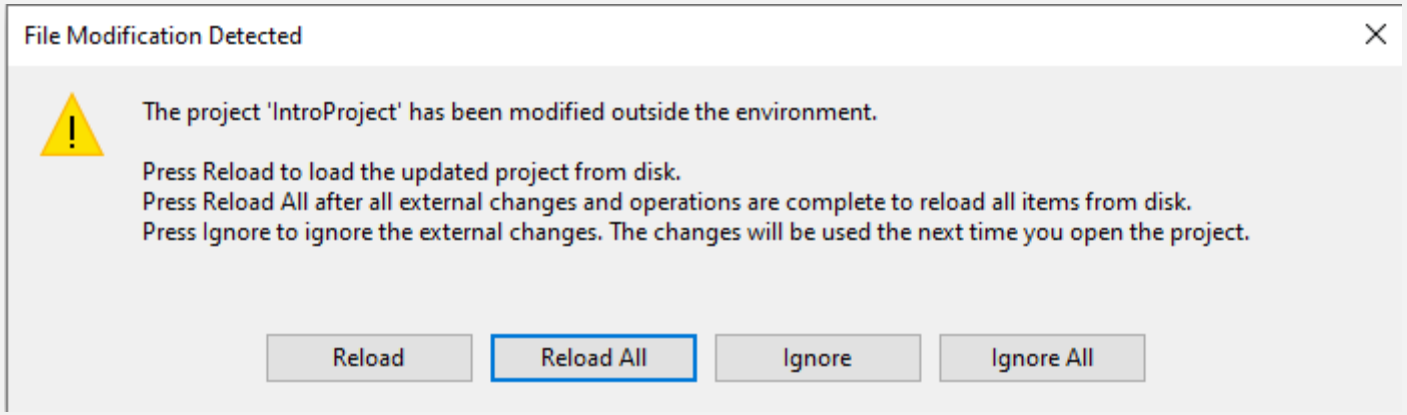
Once the script has finished being created

- A Live Coding Pop up opens and it begins compiling the scripts



# CREATE THE SCRIPT

In VS you may see this




Press ”Reload” NOT Reload All!!!!

# PLAYERCHARACTER.H

PlayerCharacter.h | PlayerCharacter.cpp | IntroProject.Build.cs

IntroProject | (Global Scope)

```
// Fill out your copyright notice in the Description page of Project Settings.

#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Character.h"
#include "PlayerCharacter.generated.h"

UCLASS()
class INTROPROJECT_API APlayerCharacter : public ACharacter
{
    GENERATED_BODY()

public:
    // Sets default values for this character's properties
    APlayerCharacter();

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

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

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

};
```

# ADJUST BUILD.CS

But we are not done! We need to add some things to the build.cs

PlayerCharacter.h | PlayerCharacter.cpp | IntroProject.Build.cs

IntroProjectModuleRules | Close (Ctrl+F4) | IntroProject

```
// Copyright Epic Games, Inc. All Rights Reserved.

using UnrealBuildTool;

1 reference
public class IntroProject : ModuleRules
{
    0 references
    public IntroProject(ReadOnlyTargetRules Target) : base(Target)
    {
        PCHUsage = PCHUsageMode.UseExplicitOrSharedPCHs;

        PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore" });

        PrivateDependencyModuleNames.AddRange(new string[] {  });

        // Uncomment if you are using Slate UI
        // PrivateDependencyModuleNames.AddRange(new string[] { "Slate", "SlateCore" });

        // Uncomment if you are using online features
        // PrivateDependencyModuleNames.Add("OnlineSubsystem");

        // To include OnlineSubsystemSteam, add it to the plugins section in your uproject file with the Enabled attribute set to tr
    }
}
```

# ADJUST BUILD.CS

- Add ”EnhancedInput”

```
public IntroProject(ReadOnlyTargetRules Target) : base(Target)
{
    PCHUsage = PCHUsageMode.Us
    PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore", "EnhancedInput" });

    PrivateDependencyModuleNames.AddRange(new string[] {  });

    // Uncomment if you are using Slate UI
```

class UnrealBuildTool.ReadOnlyTargetRules
Read-only wrapper around an existing TargetRules instance. This exposes target settings to modules without letting them to modify the

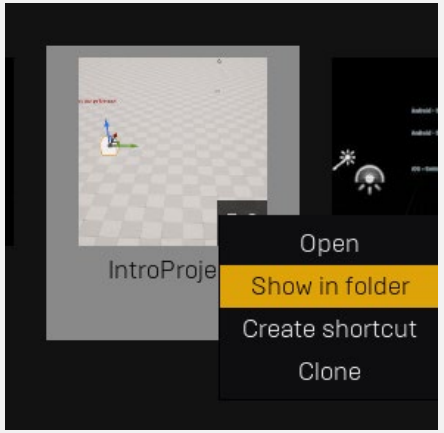
- Add ”UMG” as well, we will need this later

```
0 references
public IntroProject(ReadOnlyTargetRules Target) : base(Target)
{
    PCHUsage = PCHUsageMode.UseExplicitOrSharedPCHs;

    PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore", "EnhancedInput", "UMG" });

    PrivateDependencyModuleNames.AddRange(new string[] {  });

    // Uncomment if you are using Slate UI
```

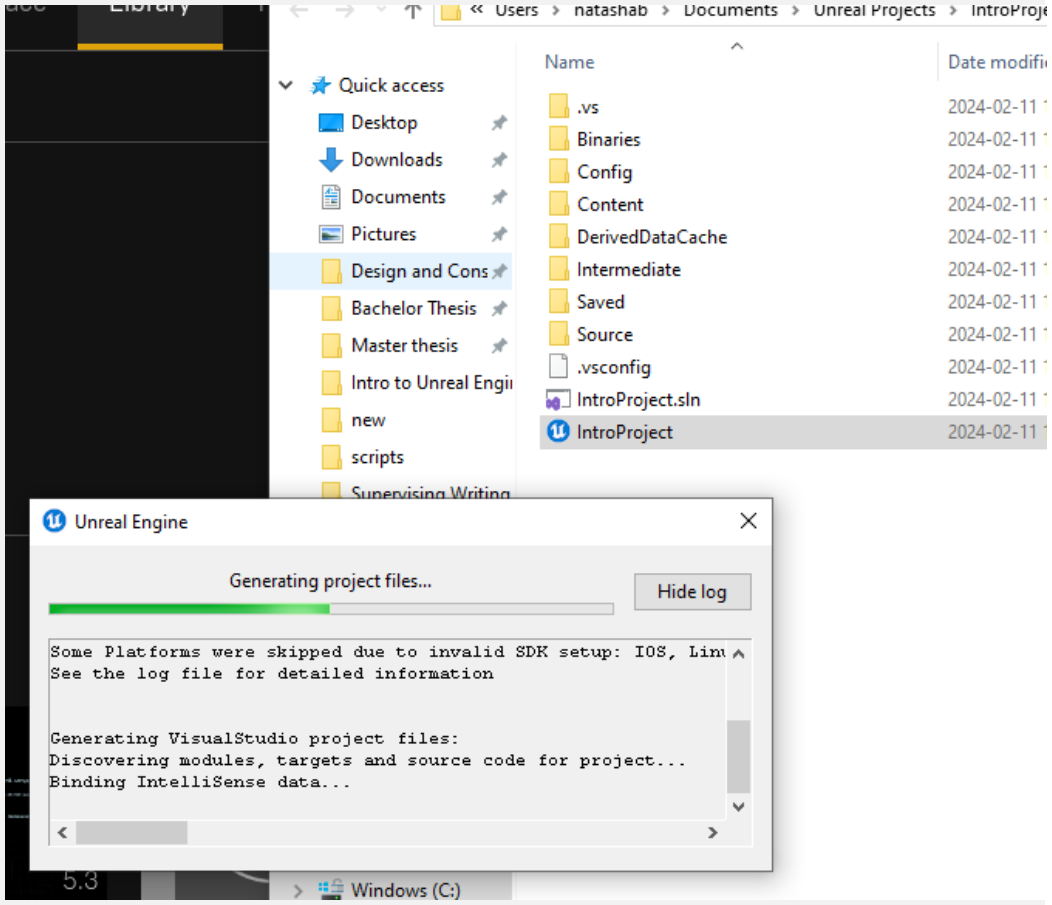
- You also need to Close the project and VS Compile (its a major file so its good to do!)

# IF THERE ARE ERRORS FROM BUILD

- Close everthing
- Navigate to the project files (from epic games or manually search)

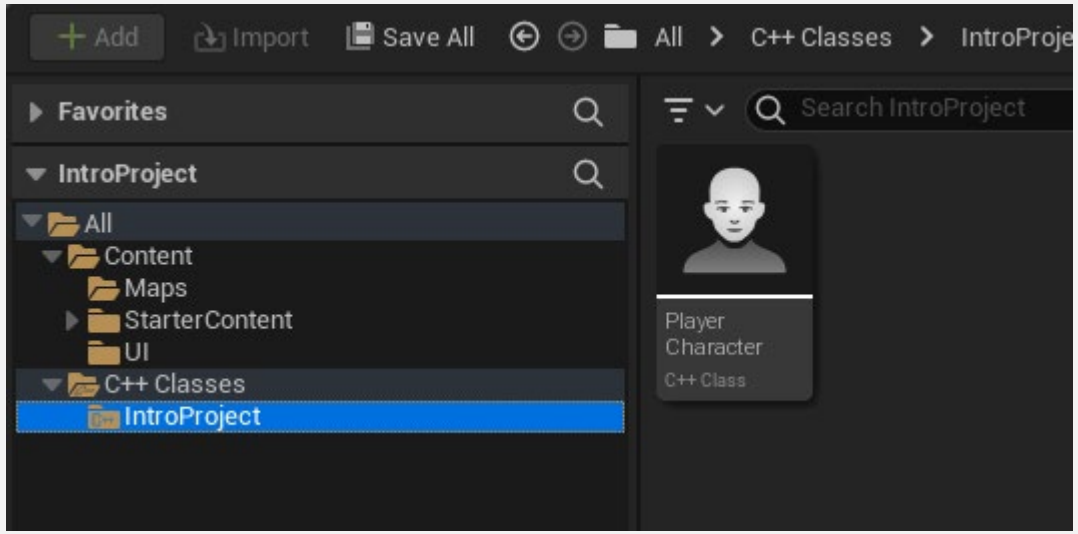



- Right-click on the Uproject file
- Select Generate Visual Studio Files.
  - Wait a bit after done, sometimes its still busy
    - If you cannot open the project after this!

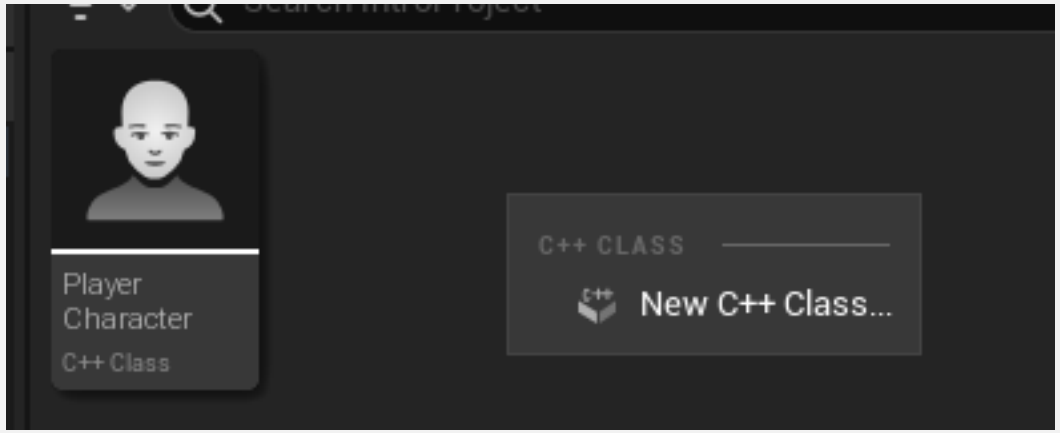


# NOW WE HAVE A C++ FOLDER

- Here all our scripts will be placed

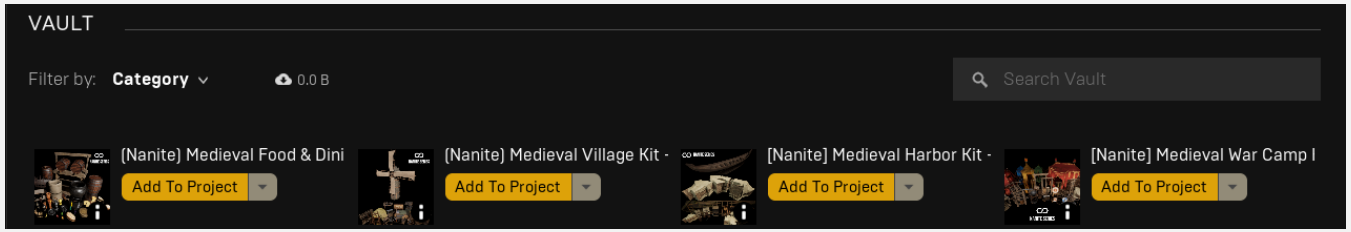


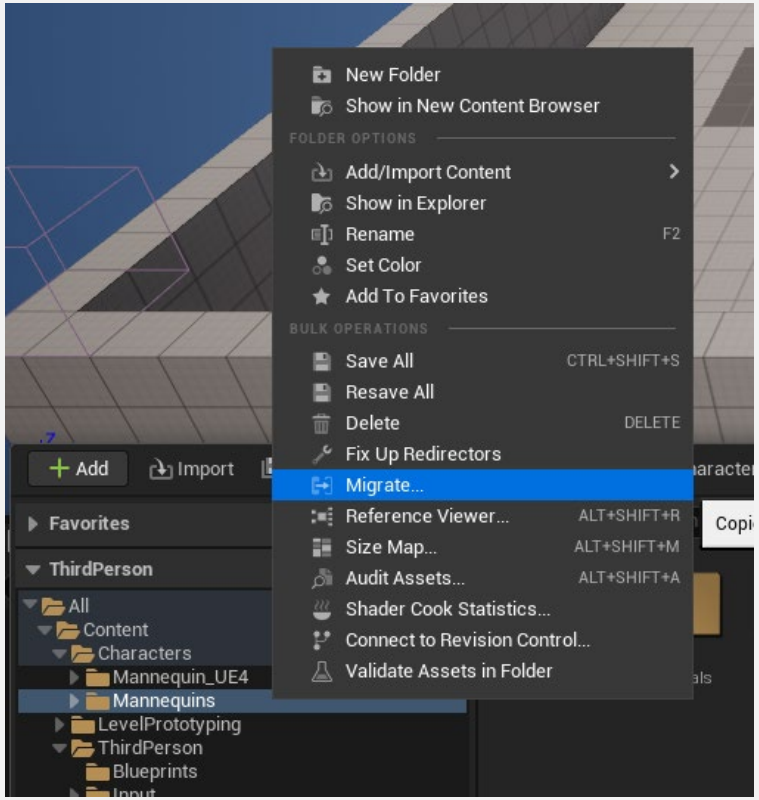

- We can now create c++ scripts from here (Right-click)



# NOW TO ADD CHARACTER ASSETS

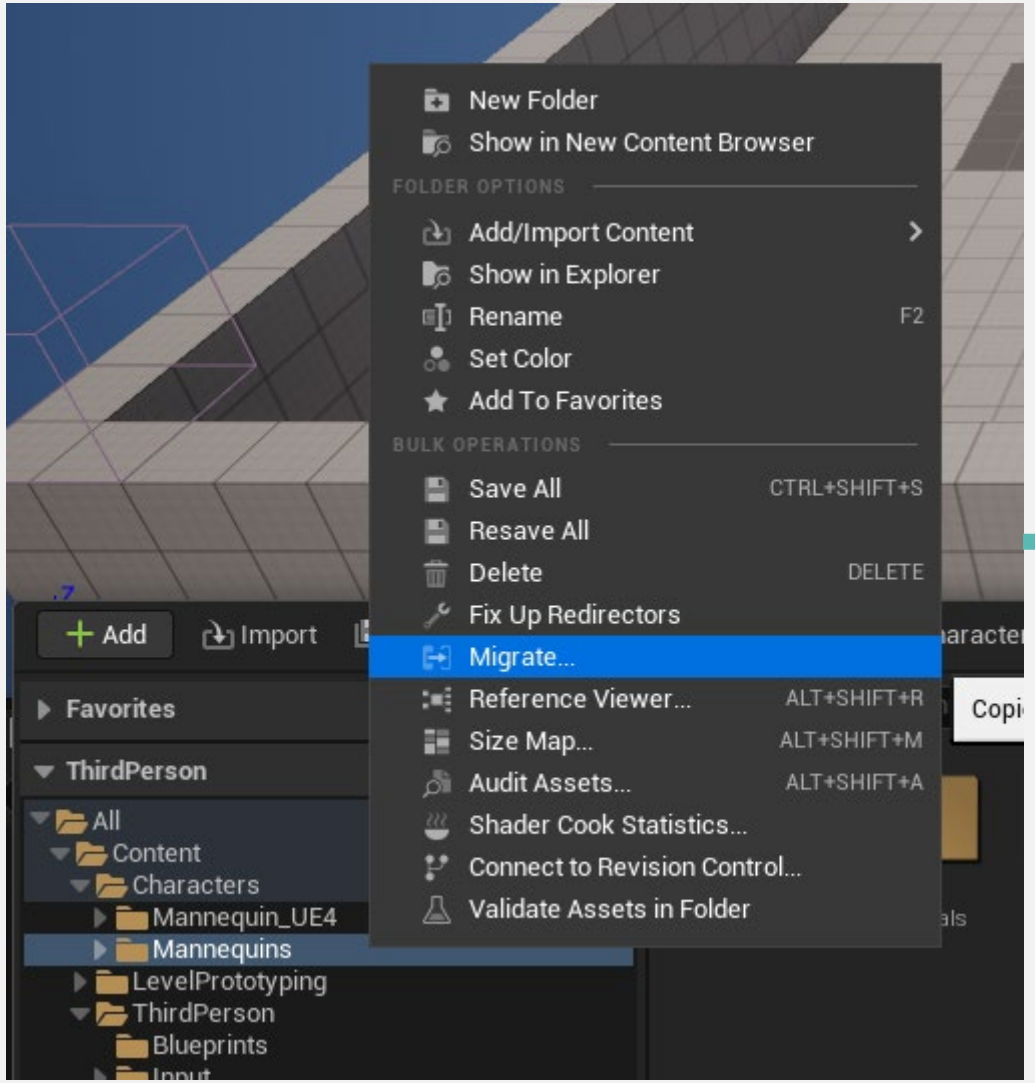
Two Options Migrate from one project to another or Add an asset from Epic Games Launcher

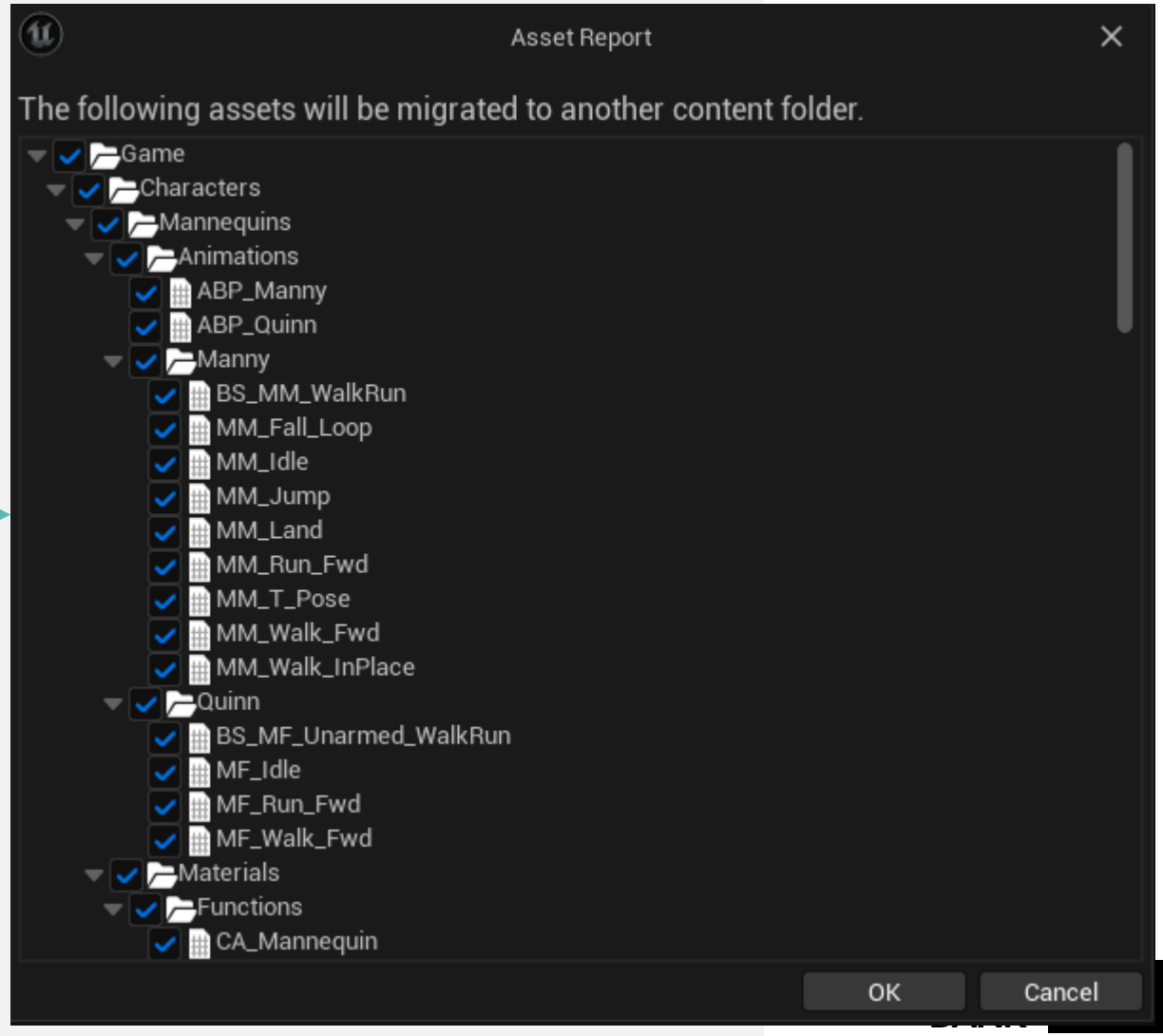


# NOW TO ADD CHARACTER ASSETS

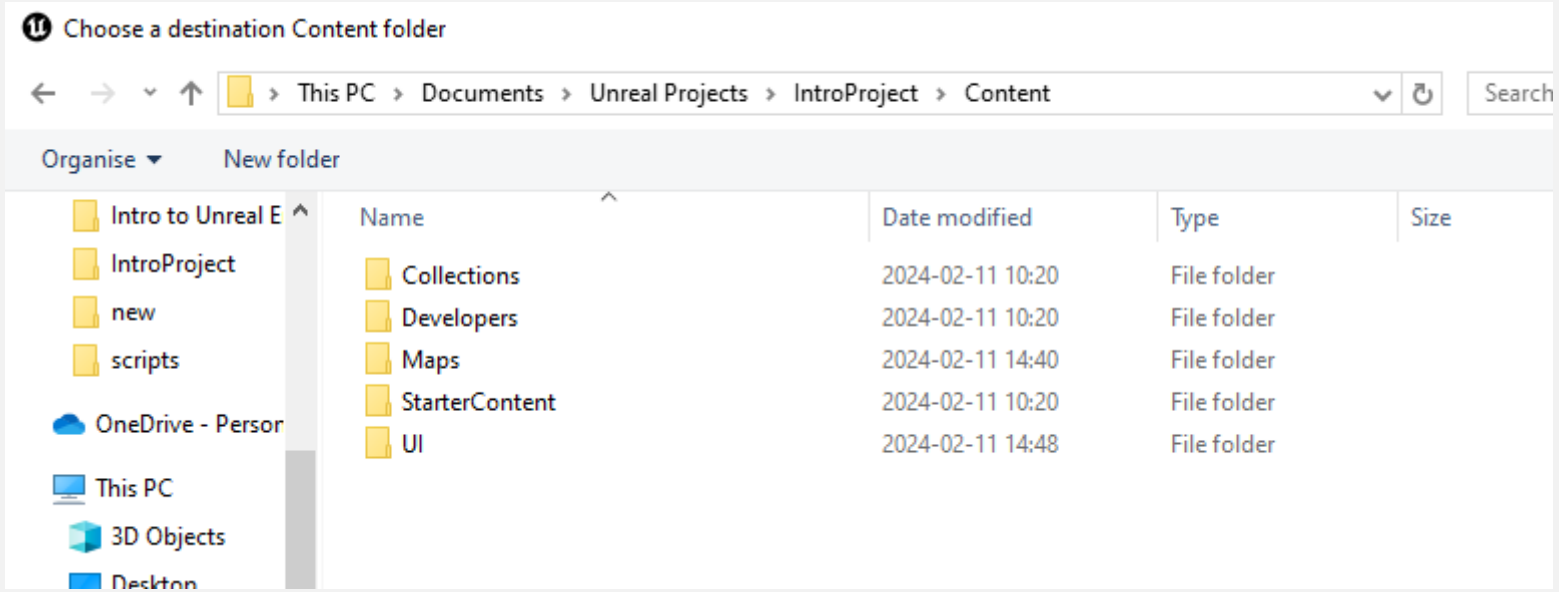
I will migrate from a Third Person Project. Right-Click on the folder you want to migrate -> Select Migrate and then check the assets to make sure your moving only the ones you want to move



# NOW TO ADD CHARACTER ASSETS

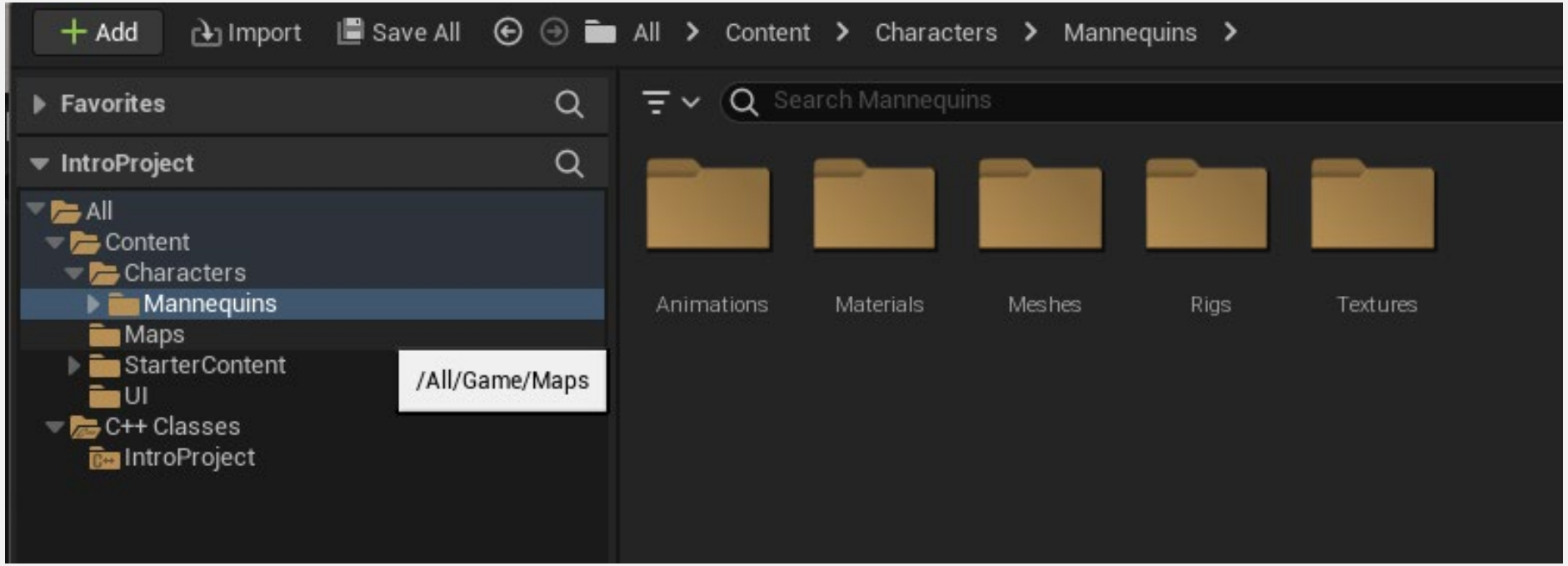
Now Navigate to the Content folder of your project and Select it in the pop-up window



# NOW TO ADD CHARACTER ASSETS

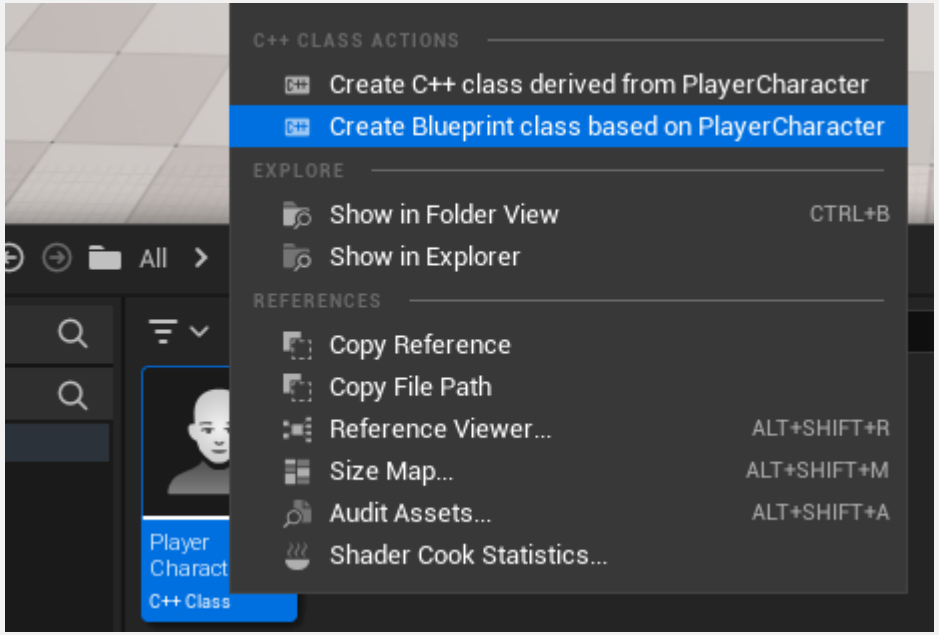
Works better if your projects are both open at the same time!

Now we have the Character we want to use. It comes with animations, a rig, skeletal mesh, textures and materials



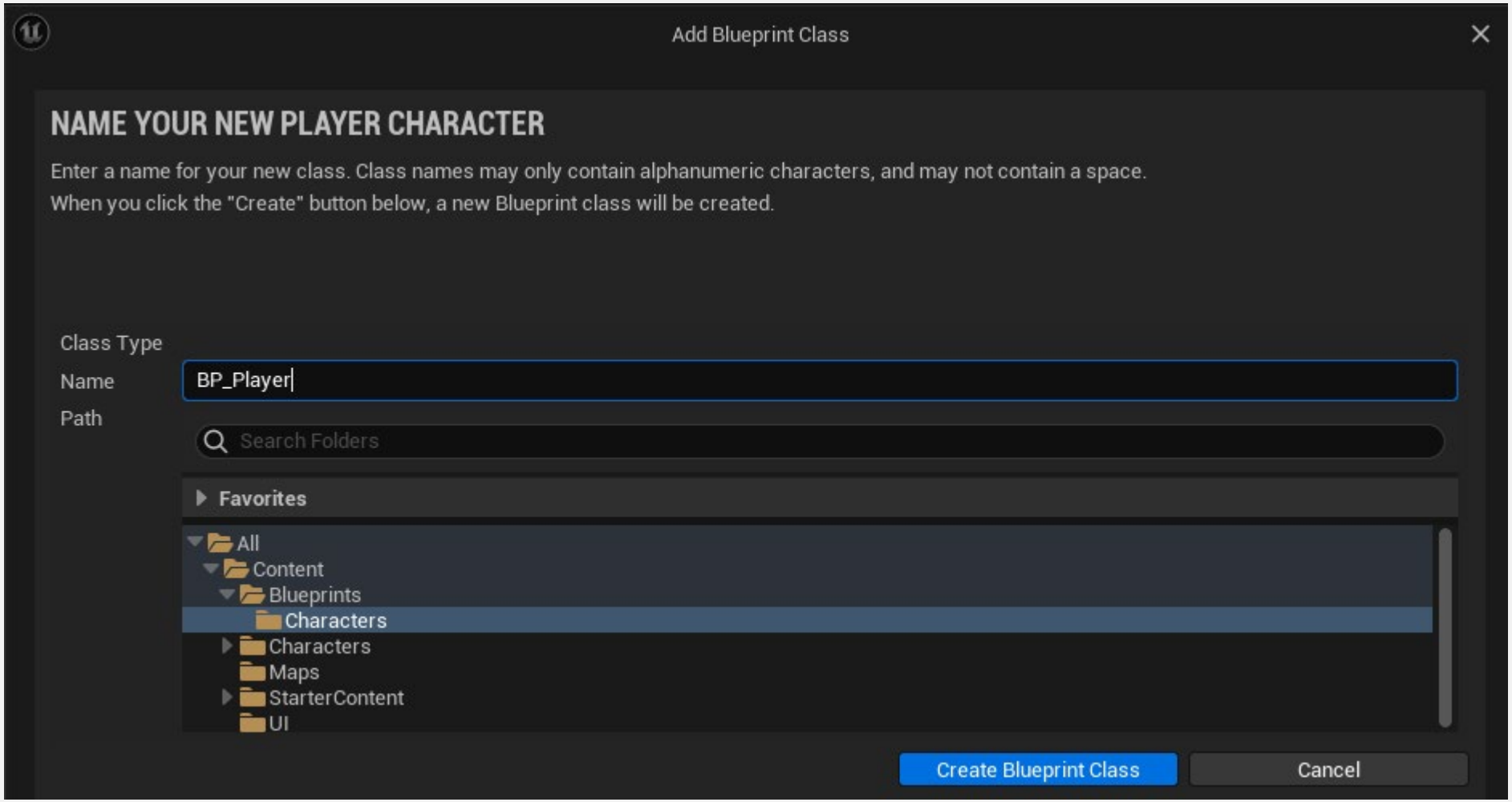
# MAKE A BLUEPRINT OF YOUR C++ CLASS

- Create a folder called Blueprints
- Create a folder called Characters inside it
- Right-click on the c++ class and select ”Create Blueprint class based on PlayerCharacter”



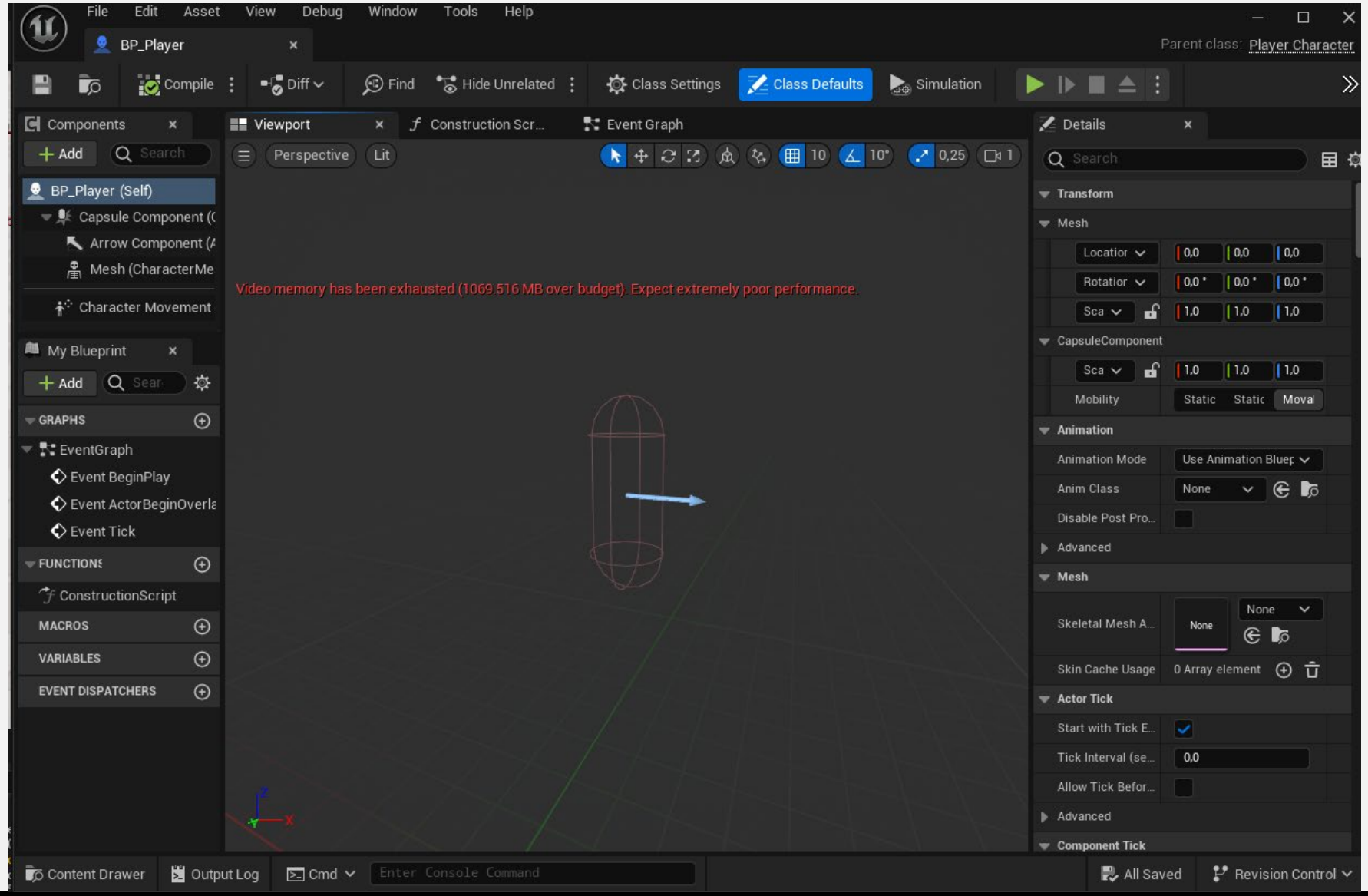
# MAKE A BLUEPRINT OF YOUR C++ CLASS

- Name this blueprint ”BP_Player”.



# MAKE A BLUEPRINT OF YOUR C++ CLASS

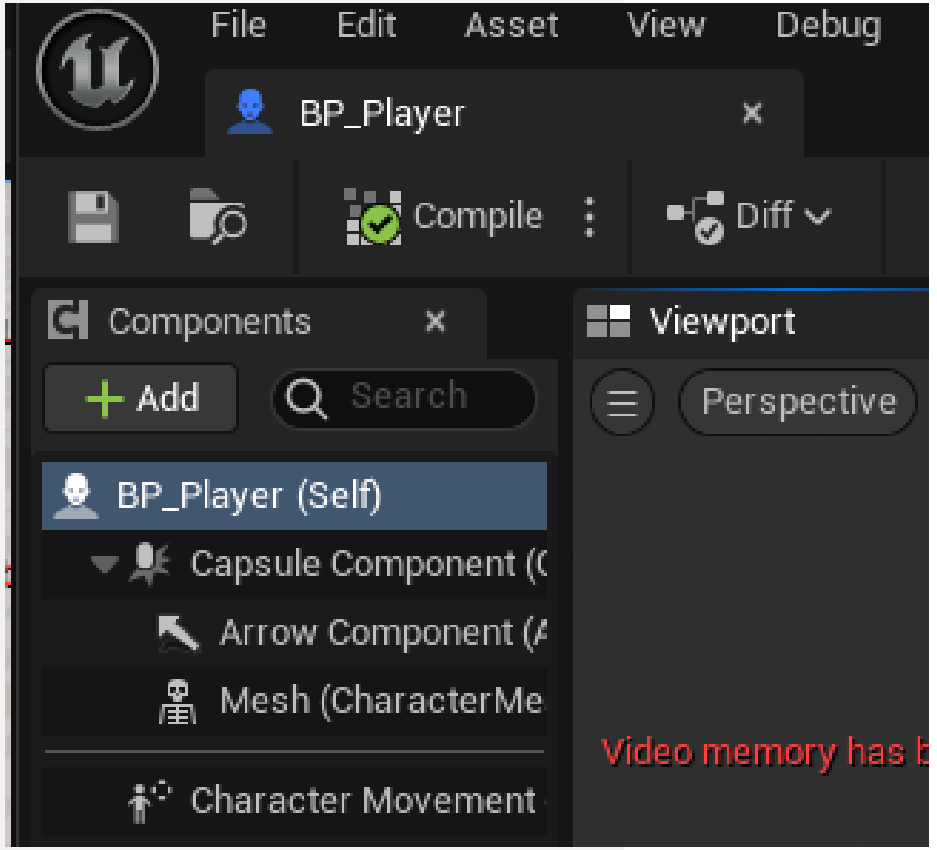
- Now it will auto-open once created



# MAKE A BLUEPRINT OF YOUR C++ CLASS

In the Components Menu, we can see all the current assets from this class

- Capsule Component
  - Used for collision detection and physics
  - This is the parent object
- Arrow Component
  - Indicated forward movement direction
- Mesh Component
  - The Skeletal Mesh – where we will put the mannequin we migrated over!
- Character Movement
  - This is from the parent class Character
  - We inherit these behaviours
  - Works like a component.



# WE WILL MAKE A THIRD PERSON CHARACTER

We need a few more components

- SpringArm Component
- Camera Component
- We will also need to add the variables for the input system

# C++ FUNCTIONS

UPROPERTY

- Variables and Components we want in the blueprints

Example:

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"))
        class USpringArmComponent* SprintArmComponent;
```

First Parameter:

- VisibleAnywhere - I can see it in BP, can adjust settings in the details panel
- EditAnywhere - I can change the values themselves in BP and adjust details!

Second Parameter

- BlueprintReadOnly – I can read values from BP edits
- BlueprintReadWrite – I can read and write values from BP edits

Third Parameter

- What folder structure I want to have. Will place under Default if you do not specify.
- Usually marked with ”Camera” (name) but since Camera is inherited we can simply call it

Fourth Parameter

- Optional
- For us we use it for defaults we want in private settings in c++

The word “class” is us forward declaring the USpringArmComponent. In the .cpp we need to #include all the relevant files for our character.

# C++ FUNCTIONS

UFUNCTION

- Functions we want to call in blueprints

Example:

```
UFUNCTION(BlueprintCallable)
            void GameOver(bool IsDead);
UFUNCTION(BlueprintPure)
            float GetHealth();
```

First Parameter:

- Lets us call the functions in blueprints
- Can also use
  - BlueprintNativeEvent
  - BlueprintImplementableEvent

Second Parameter

- BlueprintPure – creates a pure node in BP of the function
- BlueprintReadWrite – I can read and write values from BP edits
- For us we use it for defaults we want in private settings in c++

# SETTING UP THE CHARACTER.H

In the private section:

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"))
        class USpringArmComponent* SpringArmComponent;
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"))
        class UCameraComponent* Camera;
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess = "true"))
         class UInputMappingContext* DefaultMappingContext;
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess = "true"))
         class UInputAction* JumpAction;
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess = "true"))
         class UInputAction* MoveAction;
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess = "true"))
         class UInputAction* LookAction;
UPROPERTY(EditAnywhere)
        float RotationRate = 500.0f;
```

We also need to forward declare `struct FInputActionValue;` (we do this before the Uclass( ) Line)

This is for the two functions we need to create for the input system. These will be in the protected section:

```
void Look(const FInputActionValue& Value);
void Move(const FInputActionValue& Value);
```

# SETTING UP THE CHARACTER.CPP

In the constructor:

```
GetCapsuleComponent()->InitCapsuleSize(42.f, 96.0f);

bUseControllerRotationPitch = false;
bUseControllerRotationYaw = false;
bUseControllerRotationRoll = false;

GetCharacterMovement()->bOrientRotationToMovement = true;
GetCharacterMovement()->RotationRate = FRotator(0.0f, RotationRate, 0.0f);

SpringArm = CreateDefaultSubobject<USpringArmComponent>(TEXT(“SpringArm Component"));
SpringArm->SetupAttachment(RootComponent);
SpringArm->TargetArmLength = 400.0f;
SpringArm->bUsePawnControlRotation = true;

Camera = CreateDefaultSubobject<UCameraComponent>(TEXT(“Camera"));
Camera->SetupAttachment(SpringArm, USpringArmComponent::SocketName); //the socket is optional
Camera->bUsePawnControlRotation = false;
```

# SETTING UP THE CHARACTER

In BeginPlay:

```
//Add Input Mapping Context
if (APlayerController* PlayerController = Cast<APlayerController>(Controller))
{
    if (UEnhancedInputLocalPlayerSubsystem* Subsystem = ULocalPlayer::GetSubsystem<UEnhancedInputLocalPlayerSubsystem>(PlayerController->GetLocalPlayer()))
    {
        Subsystem->AddMappingContext(DefaultMappingContext, 0);
    }
}
```

In Player Input:

```
UEnhancedInputComponent* EnhancedInputComponent = Cast<UEnhancedInputComponent>(PlayerInputComponent);

if (EnhancedInputComponent == nullptr) { return; }


EnhancedInputComponent->BindAction(MoveAction, ETriggerEvent::Triggered, this,
&AThirdPersonCharacter::Move);

EnhancedInputComponent->BindAction(LookAction, ETriggerEvent::Triggered, this,
&AThirdPersonCharacter::Look);


//also add jump using Acharacter::Jump/StopJump

//these are triggered with Started and Completed
```

# SETTING UP THE CHARACTER

In Move:

```cpp
void AThirdPersonCharacter::Move(const FInputActionValue& Value)
{
    // input is a Vector2D
    FVector2D MovementVector = Value.Get<FVector2D>();

    if (Controller != nullptr)
    {
        // find out which way is forward
        const FRotator Rotation = Controller->GetControlRotation();
        const FRotator YawRotation(0, Rotation.Yaw, 0);

        // get forward vector
        const FVector ForwardDirection = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::X);

        // get right vector
        const FVector RightDirection = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::Y);

        // add movement
        AddMovementInput(ForwardDirection, MovementVector.Y);
        AddMovementInput(RightDirection, MovementVector.X);
    }
}
```

In Look:

```cpp
void AThirdPersonCharacter::Look(const FInputActionValue& Value)
{
    // input is a Vector2D
    FVector2D LookAxisVector = Value.Get<FVector2D>();

    if (Controller != nullptr)
    {
        // add yaw and pitch input to controller
        AddControllerYawInput(LookAxisVector.X);
        AddControllerPitchInput(LookAxisVector.Y);
    }
}
```

# INCLUDES

```
#include "Engine/LocalPlayer.h"
#include "Camera/CameraComponent.h"
#include "Components/CapsuleComponent.h"
#include "GameFramework/CharacterMovementComponent.h"
#include "GameFramework/SpringArmComponent.h"
#include "GameFramework/Controller.h"
#include "EnhancedInputComponent.h"
#include "EnhancedInputSubsystems.h"
#include "InputActionValue.h"
```

UE_LOG(LogTemp, Warning, TEXT("Hello World"));

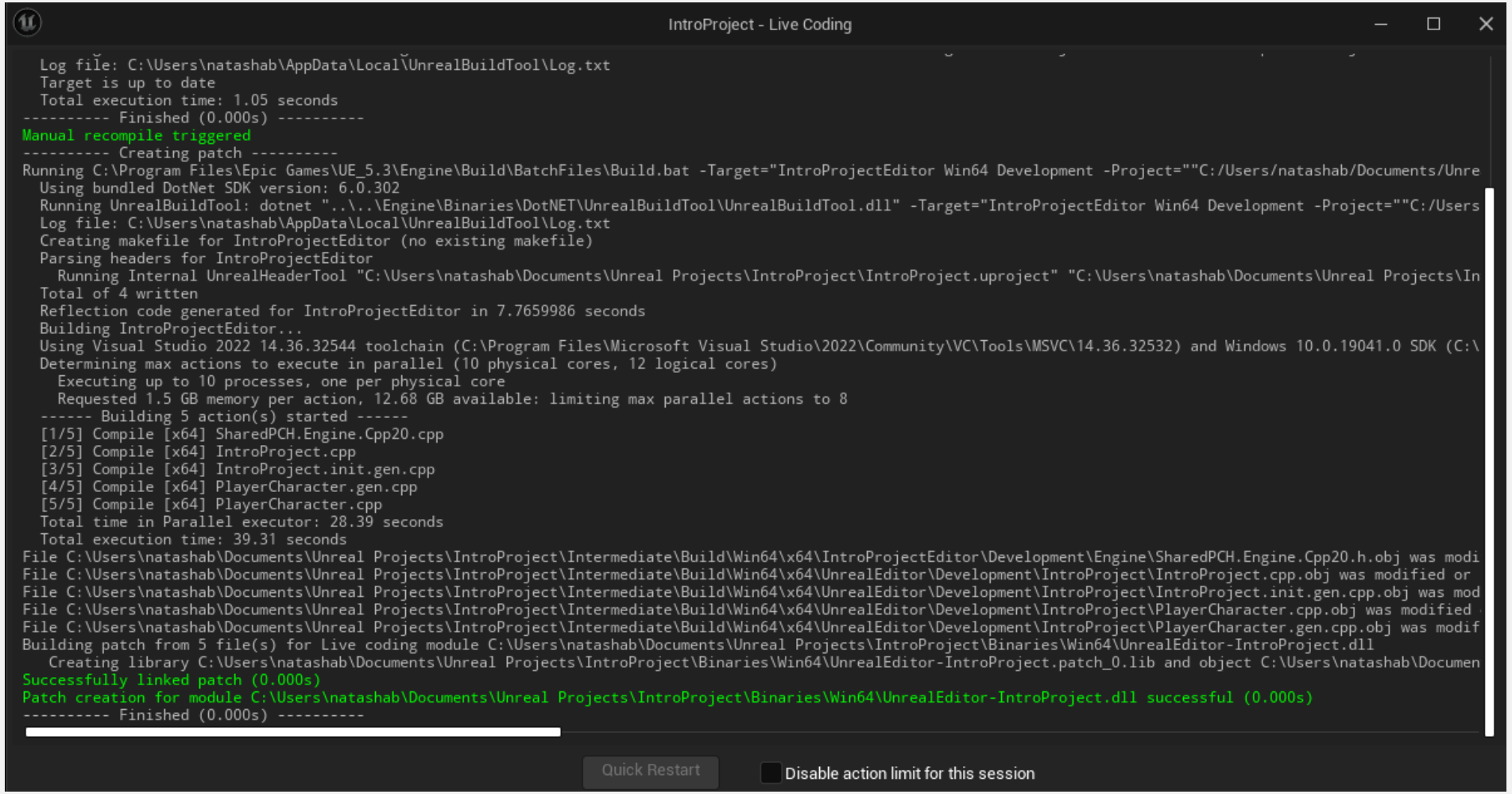
# UE_LOG

- Warning – yellow text in console and logfile
- Fatal – immediately crashes and prints red text in console and logfile
- Display – white text in console and logfile
- Error – red text in console and logfile
- Log – prints to logfile not console
- Verbose – same as Log
- VeryVerbose – same as Log

- `UE_LOG(LogTemp, Warning, TEXT( "Failed to find an Enhanced Input component!");`
- `UE_LOG(LogTemp, Warning, TEXT("Input value : %f"), value);`
- `UE_LOG(LogTemp, Warning, TEXT("Input value : %s"), *value.ToString());`

# COMPILE THE CODE

- Using VS Compile or Live Coding



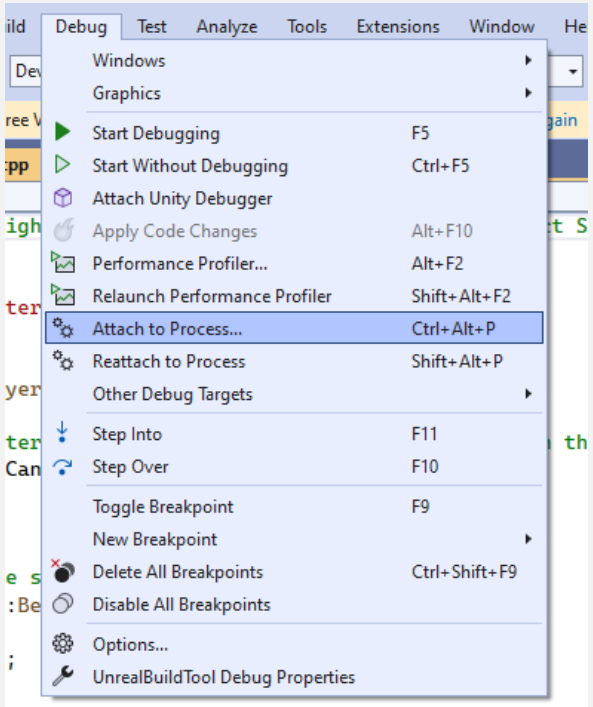
# DEBUGGING

# DEBUGGING

- You first need to set a breakpoint

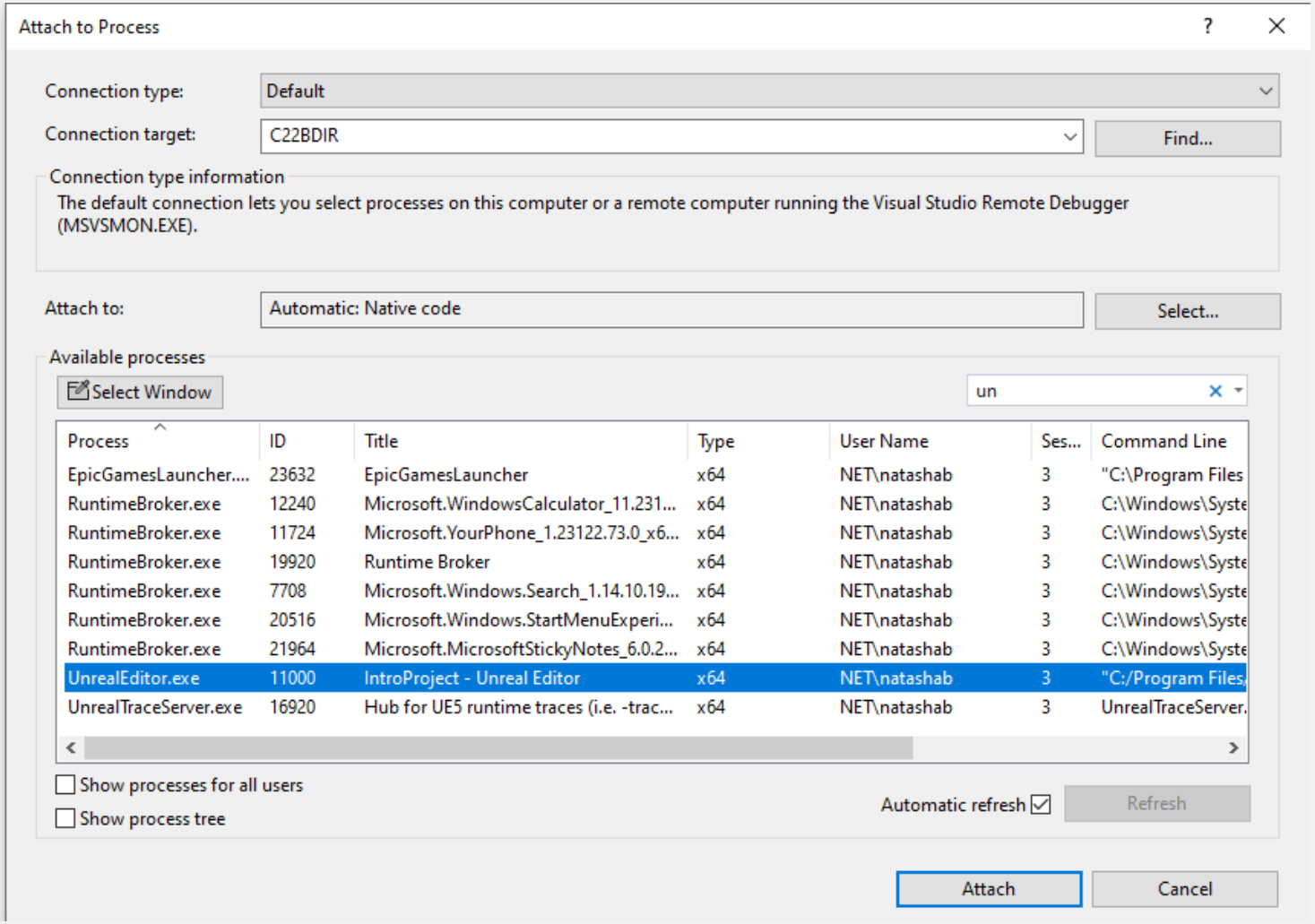
```
// Called every frame
void APlayerCharacter::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}
```

- Then attach to the process in VS -> Debug -> Attach to Process…



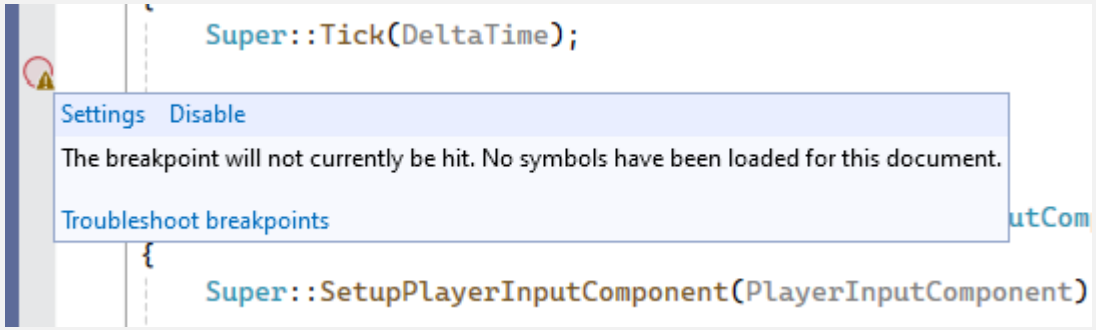
# DEBUGGING

- Then find your Unreal Engine exe and Select Attach



# DEBUGGING

If you get this notice:




Make sure you have downloaded the editor debugging symbols mentioned in the setup!

Then do the following:

- Press the Windows key + R to open the Run dialog box.
- Type “regedit” and press Enter to open the Registry Editor.
- In the Registry Editor, navigate to “HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager”.
- Right-click on the “Session Manager” key and select New → DWORD (32-bit) Value.
- Name the new value “DebuggerMaxModuleMsgs”.
- Double-click on the new value to open the Edit DWORD (32-bit) Value dialog box.
- In the Value data field, enter the value “2048” and click OK.
- Close the Registry Editor and restart your computer for the changes to take effect

# ENHANCED INPUT

# ENHANCED INPUT SYSTEM

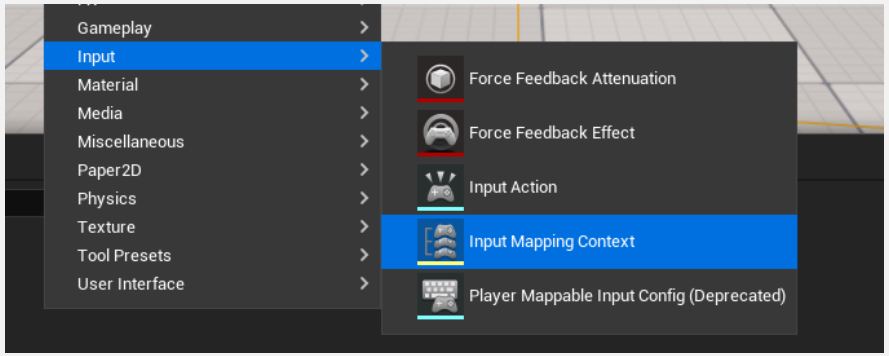
Need to setup the blueprint side of the input system

We need:

- Input Actions
- InputMappingContext

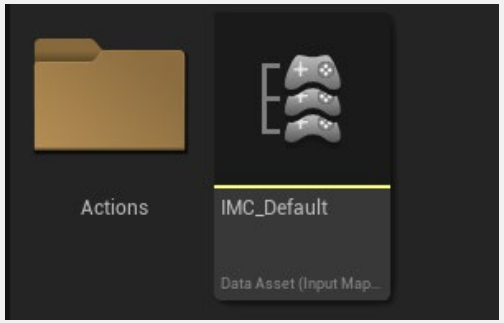
- First lets setup the InputMappingContext:
  1. Make a new folder in Blueprints called “Input”
  2. Inside “Inputs” make a new folder called “Actions”


  3. In the Input Folder, right-click -> hover over input -> select Input Mapping Context

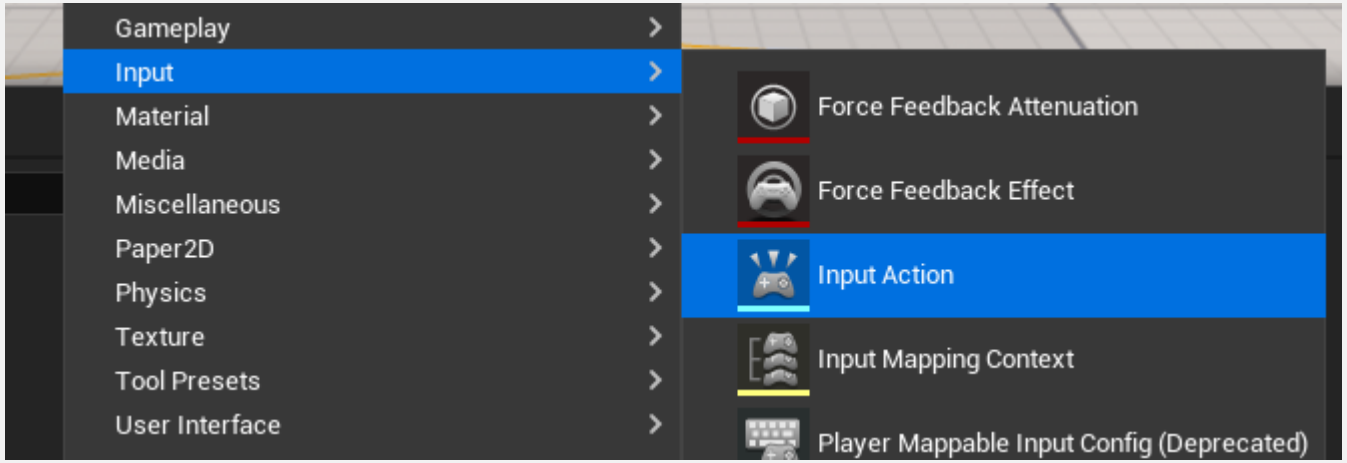


# ENHANCED INPUT SYSTEM

4. Name it “IMC_Default” (InputMappingContext_Default)



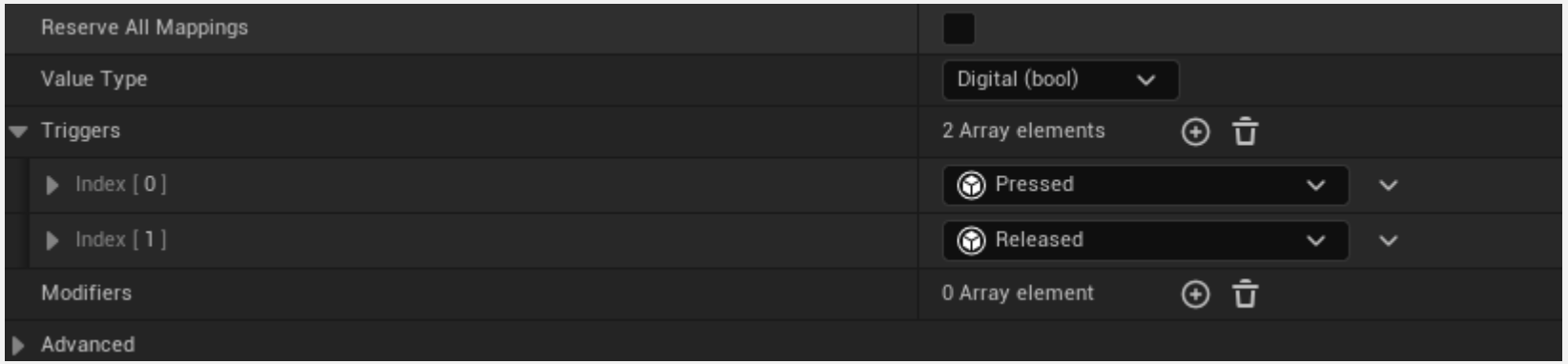

5. Open the Actions folder
6. Right-click -> Input -> Input Action



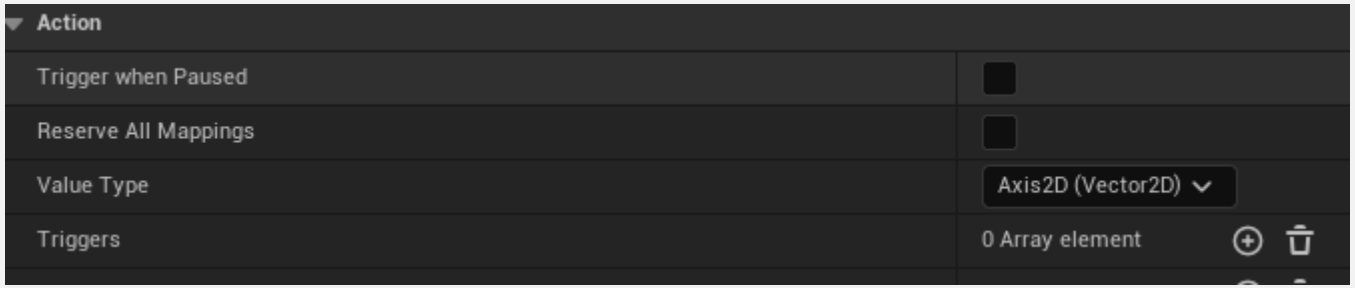

7. Name it “IA_Jump”, Repeat 6 and 7 for IA_Move and IA_Look

# ENHANCED INPUT SYSTEM

7. Open IA_Jump, under Action
   - Create 2 triggers (press the plus sign twice)
   - Set the first trigger to Pressed
   - Set the second trigger to Released



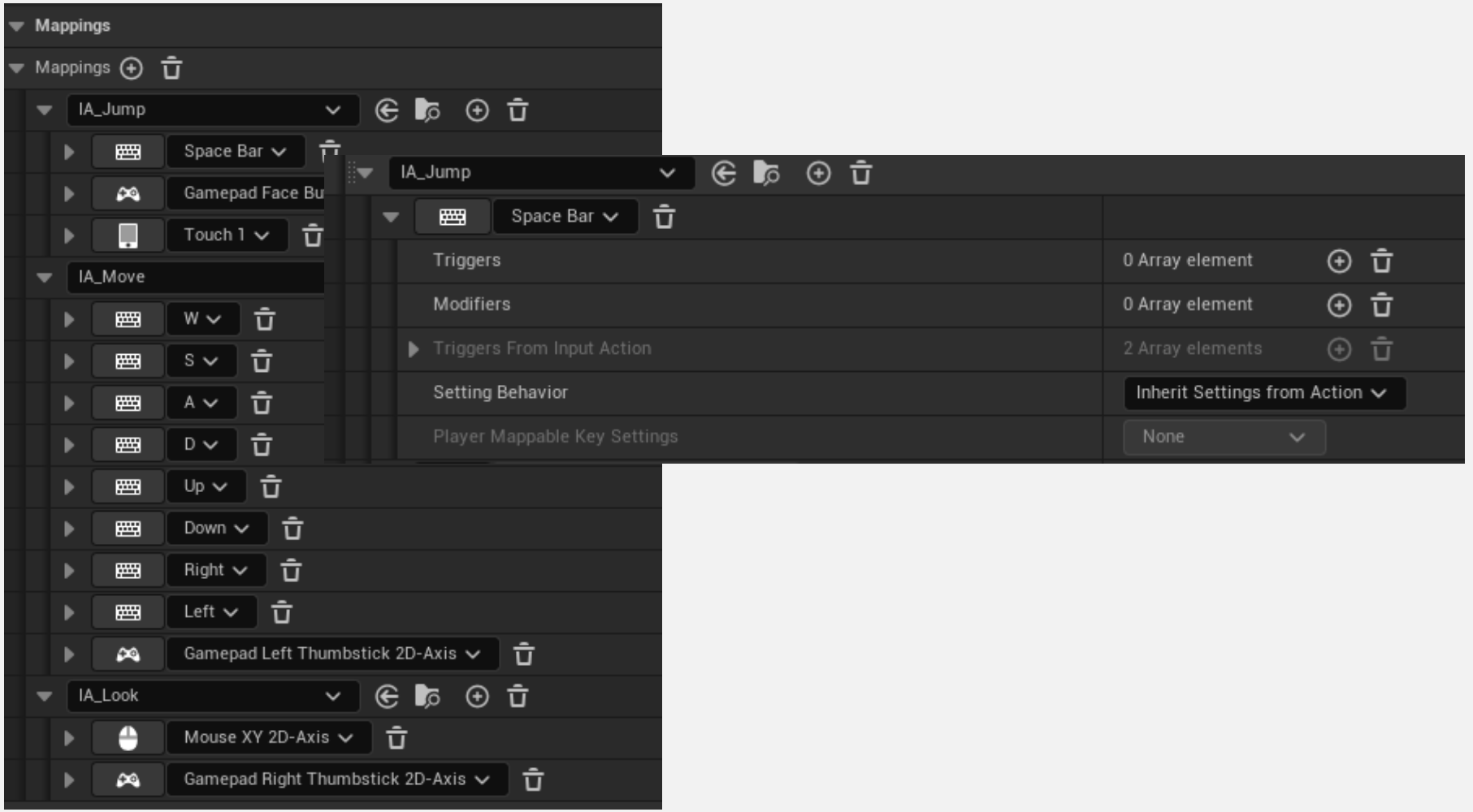

8. Open IA_Move, under Action
   - Change the value type to Axis2D(Vector2D)




9. Repeat step 8 for IA_Look

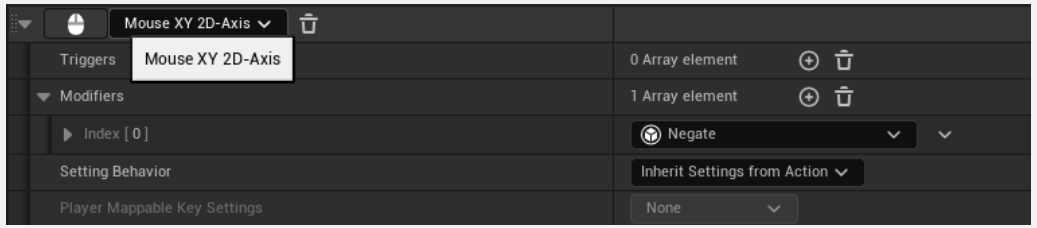
# ENHANCED INPUT SYSTEM

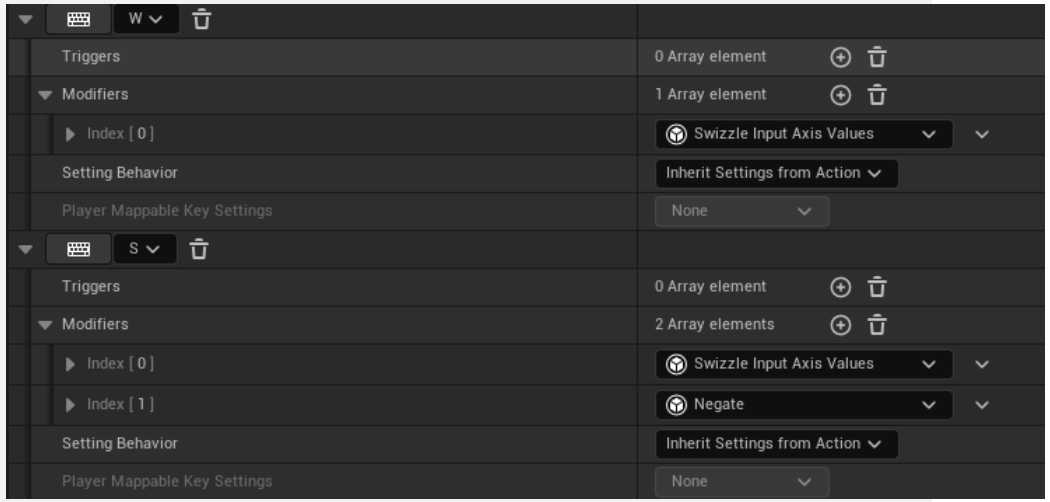
10. Open IMC_Default
11. Create 3 Mappings, for each IA Action



# ENHANCED INPUT SYSTEM

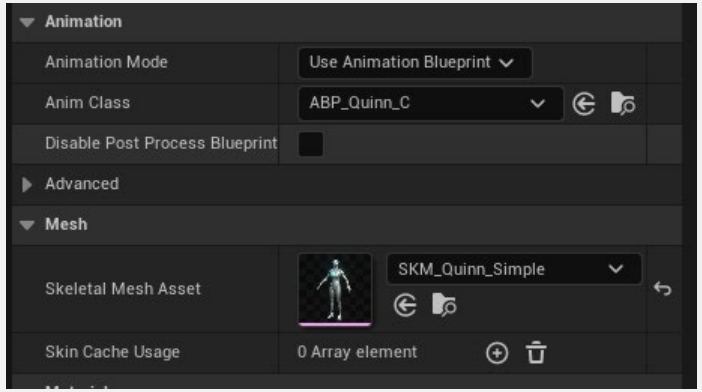
12. In each Control you set, you need to adjust the modifiers
    - Jump – No Modifiers, it is inherited from Action
    - Forward – 1 Modifier -> Swizzle Input Axis Values
    - Backwards – 2 Modifiers -> Swizzle Input Axis Values + Negate
    - Move Right – No Mods
    - Move Left – 1 Mod -> Negate
    - Gamepad Thumbstick L – 2 Mods -> Dead Zone + Scalar
    - Mouse XY – 1 Mod -> Negate
    - Gamepad Thumbstick R – 1 Mod -> Dead Zone

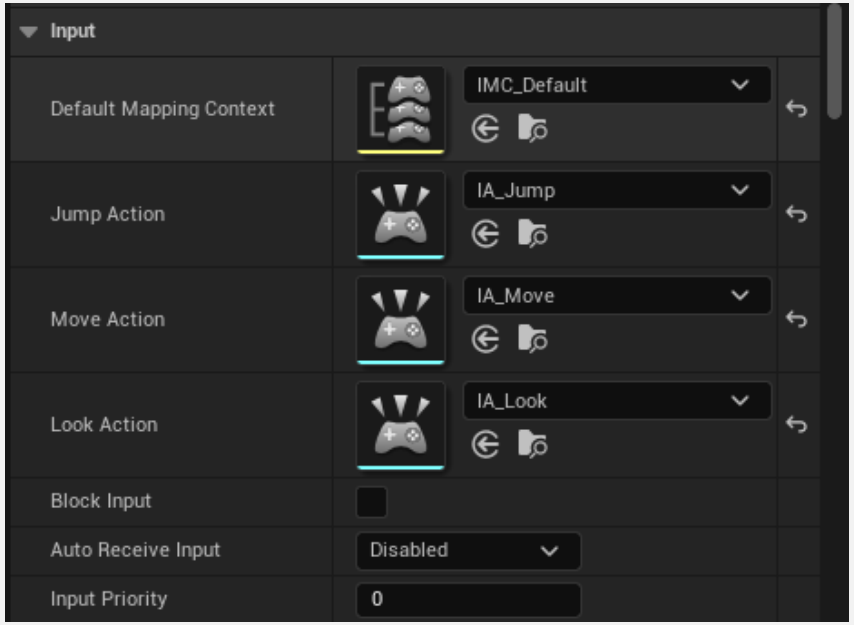



# FINISHING UP THE BP_PLAYER

- Assign all our variables for the Mesh



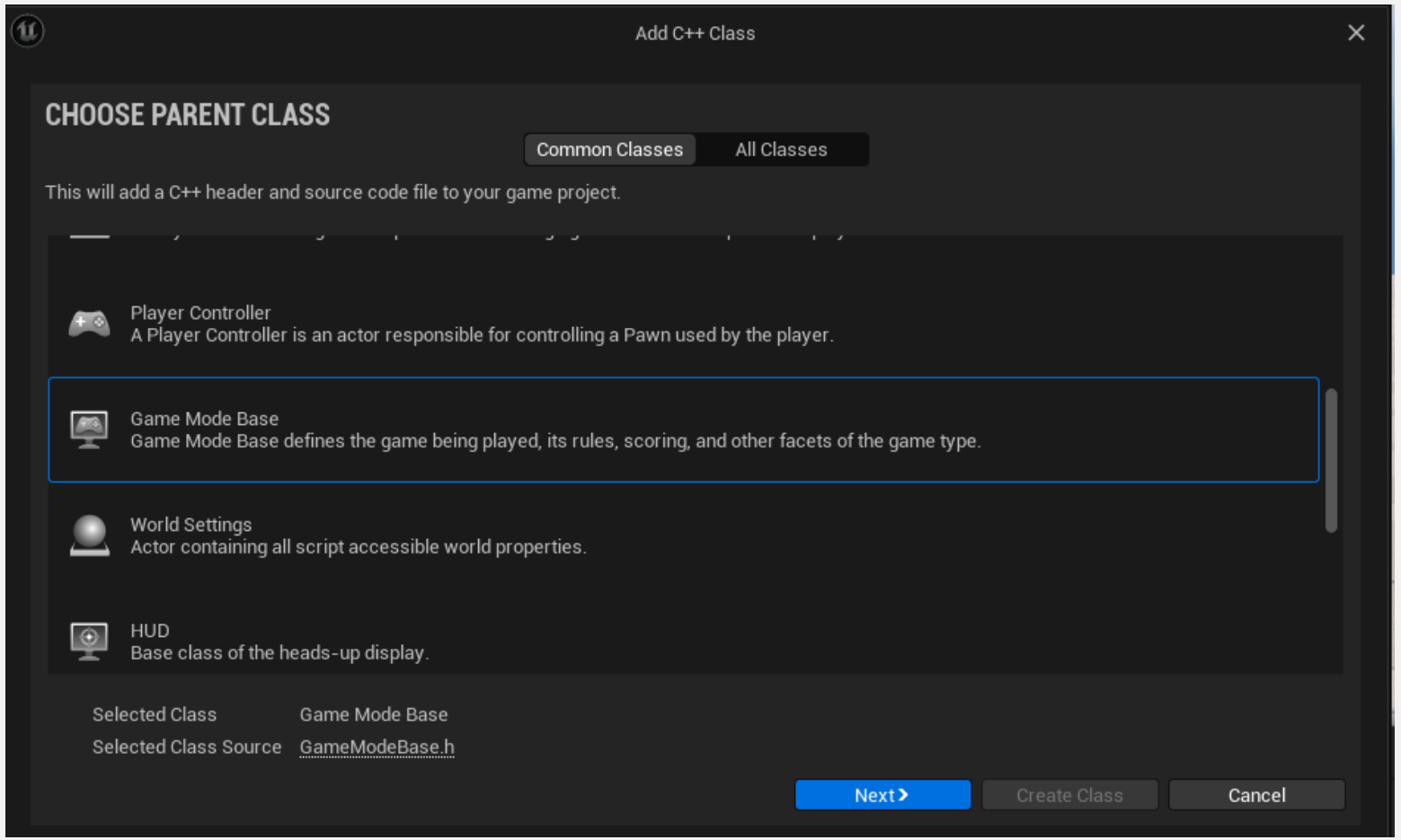

- Assign our input mapping context and Assign our input actions



# GAME MODE

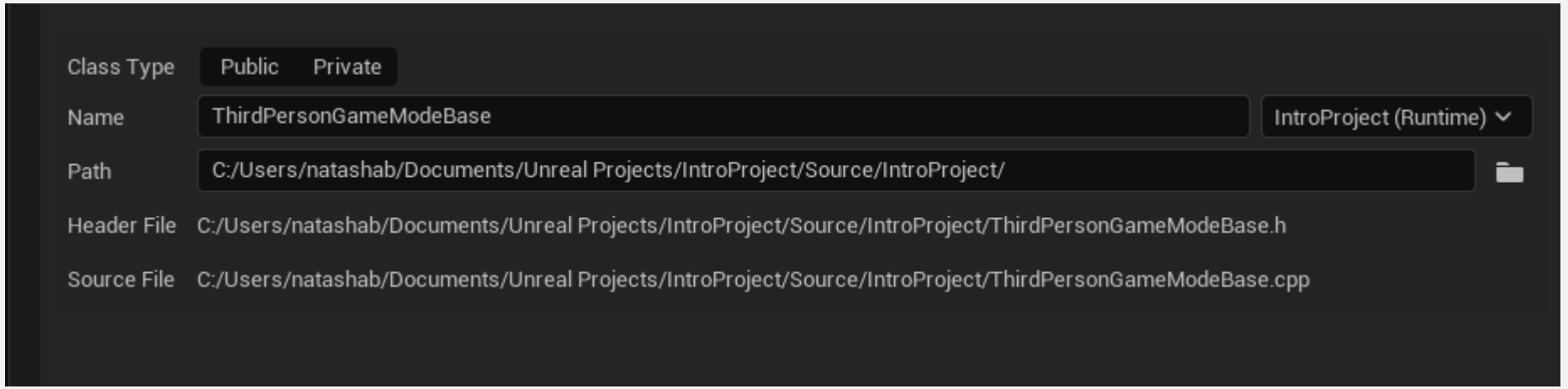
# CREATE GAMEMODE

- In the c++ scripts section right-click and select new c++
- Find the Game Mode Base Class and Click Next



# CREATE GAMEMODE

- Name the script ThirdPersonGameModeBase

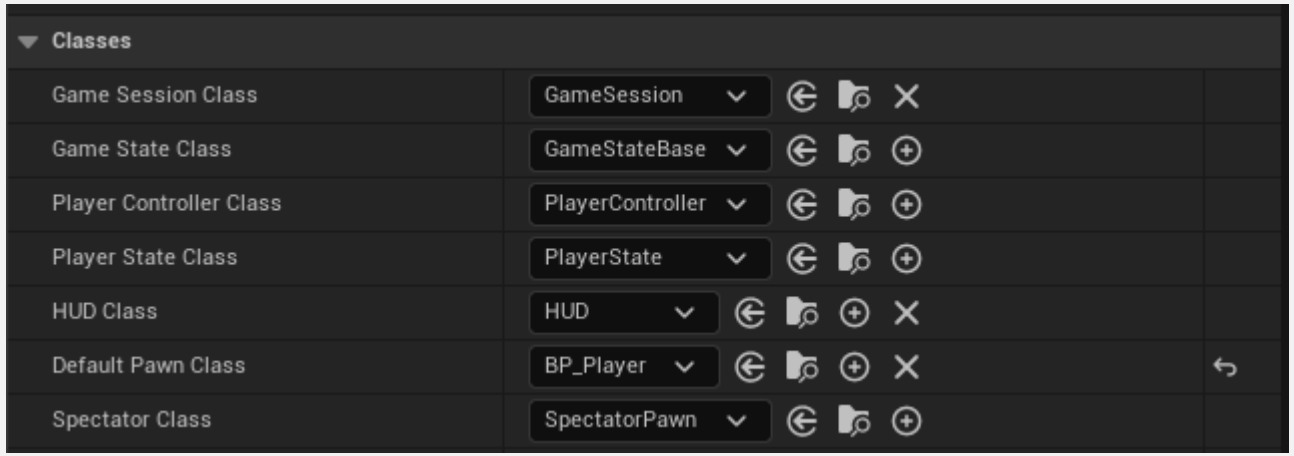



- Wait for the scripts to compile, when complete it should look like this:

```
#include "CoreMinimal.h"
#include "GameFramework/GameModeBase.h"
#include "ThirdPersonGameModeBase.generated.h"

/**
 *
 */
UCLASS()
class INTROPROJECT_API AThirdPersonGameModeBase : public AGameModeBase
{
    GENERATED_BODY()

};
```

# CREATE GAMEMODE

- Make a new folder in Blueprints called “GameModes”
- Create a blueprint from ThirdPersonGameModeBase
- Call it BP_ThirdPersonGameMode
- In the Class Defaults of the Blueprint change the default pawn to our BP_Player




- Do not forget to compile and save the BP!

# IN THE GAME SCENE

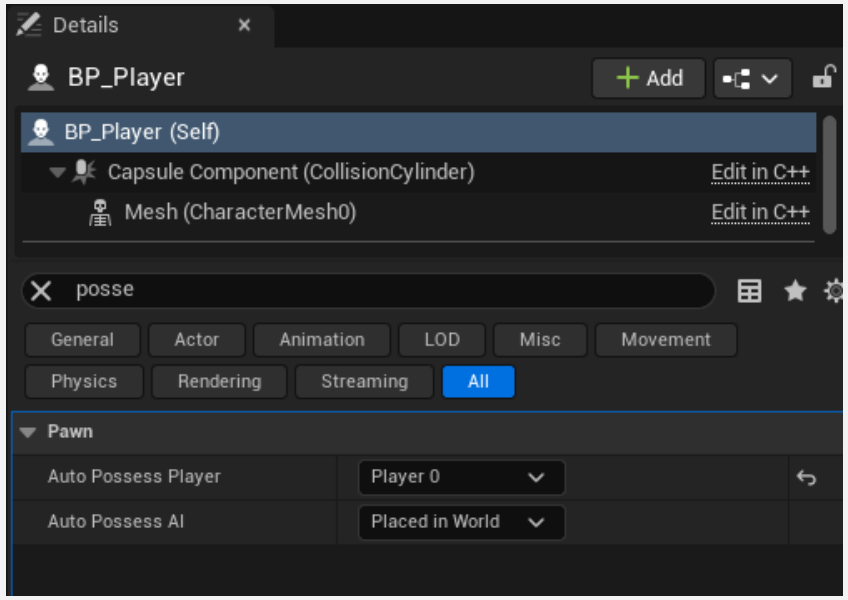
- Drag and drop the BP_Player in the scene

When we press play ….

Nothing happens! It is the same as before.

To “possess” our player we need to

1. Select the game object
2. Go to the details panel
3. Search for “Possess”
4. Change the Auto possess player to Player 0



# IN THE GAME SCENE

When we press play ….

We now possess the player! We can move around as our character!

BUT

This process of setting the auto possess it inefficient, we want to be able to press play without having to set up some stuff in the details for it to work.

- This is where the Game mode we created comes in to play.
- We set the default pawn to our BP_Player

This means the player will be spawned and controlled if we have a player start in the scene.

Now we just need to set the game mode, there are two methods:

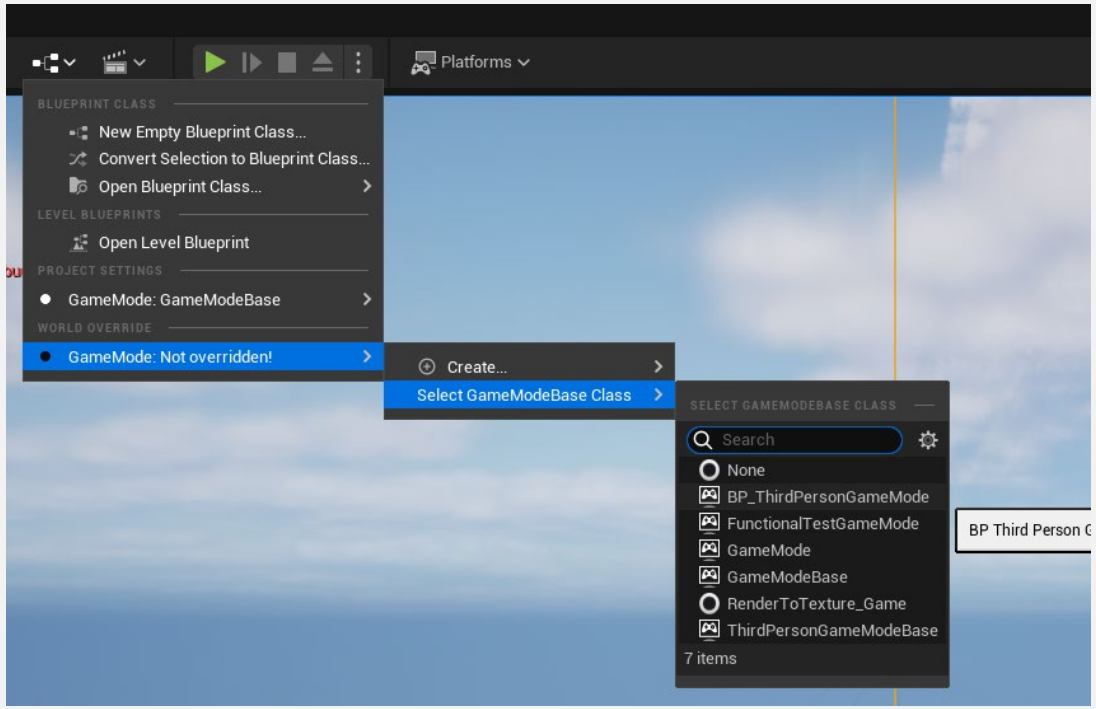
1 – We go into project settings and replace the game mode with our bp_thirdpersongamemode

2 – We select the level blueprint settings and change the override to our bp_thirdpersongamemode. This means only this map will use the third person game mode

Option 2 seems like the best option at this time.

# SET THE LEVEL GAME MODE OVERRIDE

- Go to world override -> select gamemodeabase class -> BP_ThirdPersonGameMode




- Remember to delete the BP_Player from the outliner before pressing play.
- Now our character spawns where the player start is (and facing direction)

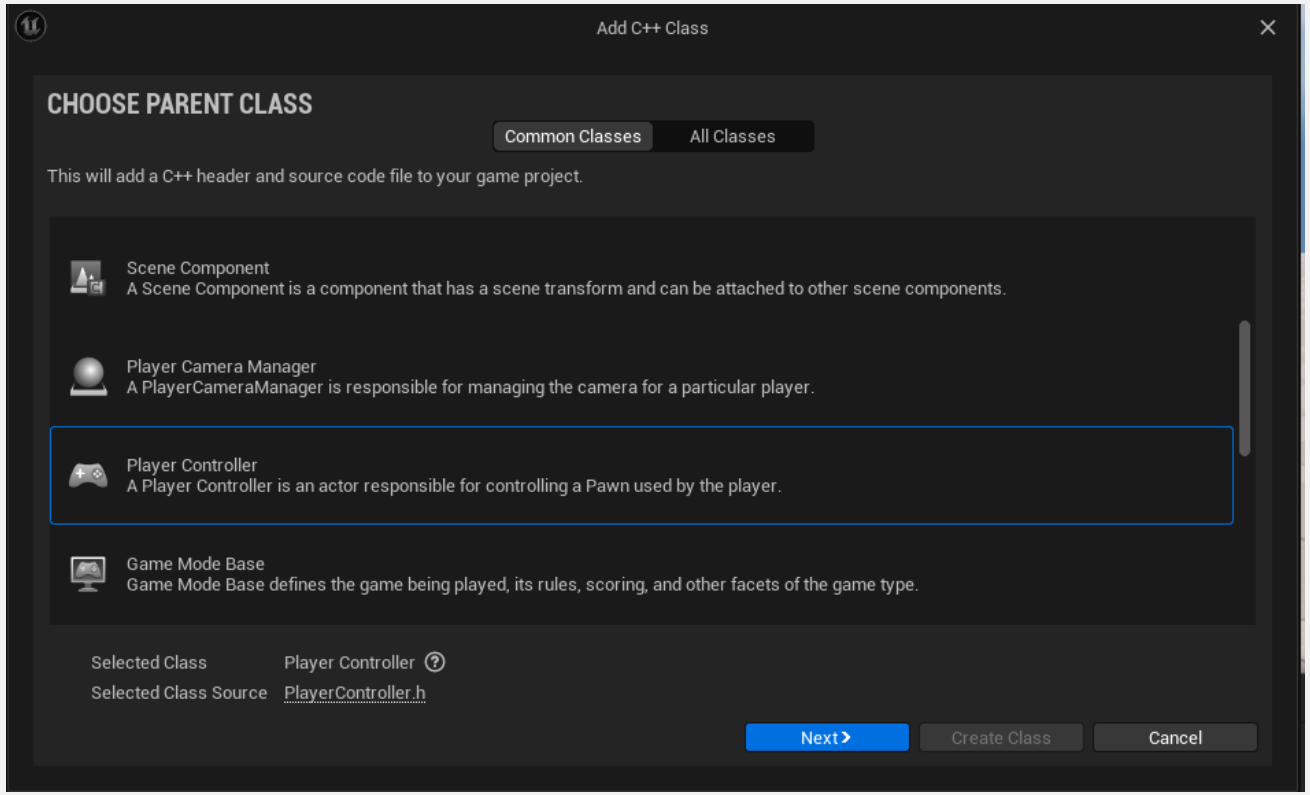
# PLAYER CONTROLLER

*Create and Setup the HUD*

# SETTING UP A PLAYER CONTROLLER

Add a new Script

- Right-click in c++ folder
- Select Player Controller and NEXT
- Name this script ”CharacterPlayerController”



# SETTING UP A PLAYER CONTROLLER

Once compiled the player controller header file should look like this

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/PlayerController.h"
#include "CharacterPlayerController.generated.h"

/**
 *
 */
UCLASS()
class INTROPROJECT_API ACharacterPlayerController : public APlayerController
{
    GENERATED_BODY()

};
```

Our purpose for this player controller is to setup the HUD

Which we will do in BeginPlay with the variables we will create in the header.

- Add the protected BeginPlay override (we inherit from player controller)
- Add the private UPROPERTY HUDClass which will hold the WBP information in the editor
- Add the private UPROPERTY HUD which holds the userwidget data in c++

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

private:
    UPROPERTY(EditAnywhere)
        TSubclassOf<class UUserWidget> HUDClass;

    UPROPERTY(VisibleAnywhere)
        UUserWidget* HUD;
```

# SETTING UP A PLAYER CONTROLLER

We need to include at the top of the .cpp

```
#include ”Blueprint/UserWidget.h”
```

Then in BeginPlay we add the following lines, after super::BeginPlay();

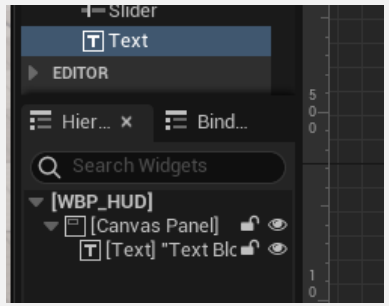
```
HUD = CreateWidget(this, HUDClass);
if (HUD)
{
        HUD->AddToViewport();
}
```

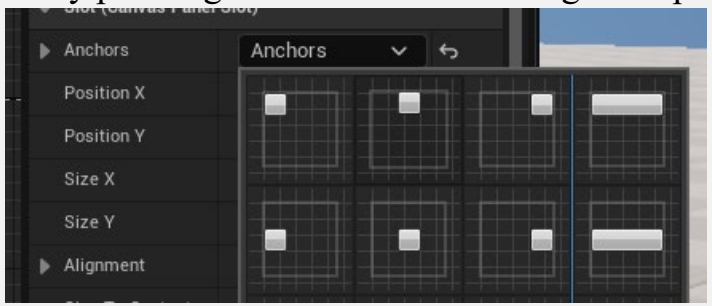
Save and Compile the Scripts!

# SETTING UP A PLAYER CONTROLLER

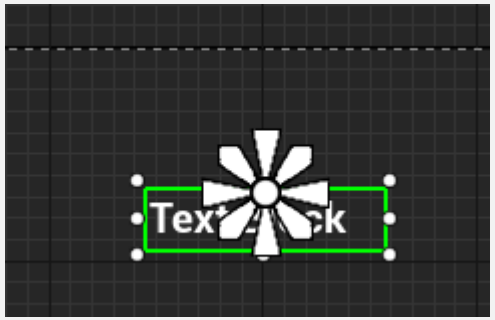
Lastly, after we compile the code

- Create a folder in Blueprints called ”Controllers”
- Create BP of player controller c++ class, call it ”BP_PlayerController”
- Create a new user widget in the UI folder called ”WBP_HUD”
- Open the user widget, add the canvas panel and then add a text component
  - Select the text and set the anchor by pressing shift + ctrl and clicking the top middle option



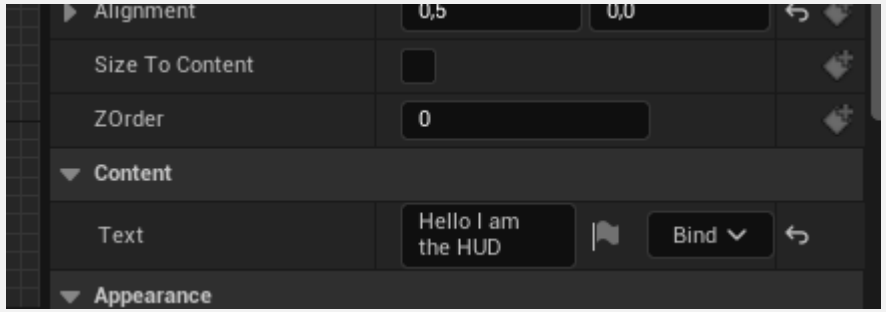

  - Widen the text rect box (until the text in the component is inside the boundary)
  - Hold ctrl + click the center of the anchor flower and drag it down a few units



  - This anchors our text to the top of the screen with a space of a few units in-between.

# SETTING UP A PLAYER CONTROLLER

- Change the text inside the component to ”Hello I am the HUD”




- You may need to adjust the bounding rect again. Otherwise you can enable the ”Size to Content”. Which should do this for you
- Compile and save the WBP!

- Last but not least
  - In the BP_PlayerController, set the HUDClass variable to our WBP_HUD.
  - In the BP_ThirdPersonGameMode, set the default player controller to our BP_PlayerController.
  - Now if we play the game we should see the text ”Hello I am the HUD” on our screen.

# ANIMATION CONTROLLER

# COLLISION HANDLING

*Enter*

*Stay*

*Exit*

# POSSESSION

*Switching Characters*

*Camera work*

# SEQUENCER

*Movies*

*Clips*

*Cutscenes*

# CLOSING NOTES

*Next Week*

# CLOSING OF PART 1

- Some slide headings were not discussed.
  - These will be in part 2
- Some slides were discussed a little
  - These will be in more detail for part 2

- Next week Wednesday is Part 2