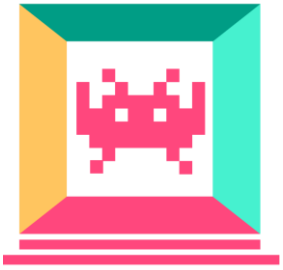


Unreal Engine Game Development for Beginners

By Daniel Buckley

Unreal Engine Developer and Teacher

Before diving into this eBook, why not check out some resources that will supercharge your coding skills:

Table of Contents

Beginner's Guide to Game Development with Unreal Engine

Introduction

Are you ready to start developing FPS games, RPGs, retro-style games, and more with the amazing Unreal Engine? You're in for quite a treat, then!

Unreal Engine 4 is one of the most popular engines out there for game development - powering everywhere from indie games like *Dead by Daylight* to AAA games like *Gears 5.* Its powerful graphical features allow developers to create visually stunning games, while the Unreal Engine blueprinting system creates an environment that makes it easy for programmers and non-programmers alike to create the game of their dreams. To boot, the engine is completely free until you earn $1 million with your product, making it a great choice for any aspiring developer. Before you get intimidated by the power of the engine, though, remember that everyone starts somewhere - and this tutorial is just the place to get started.

In this tutorial, we'll be guiding you through your initial steps as you create your first game inside of the Unreal Engine. This game will be a 3D platformer game with collectible gems and enemies to avoid. We'll be going many of Unreal's features such as level creation, blueprints, creating logic with nodes, and even creating a UI.

All in all, we'll provide you with the foundations to get started, so if you're ready to go, let's get started with creating our Unreal Engine platformer!

Project Files

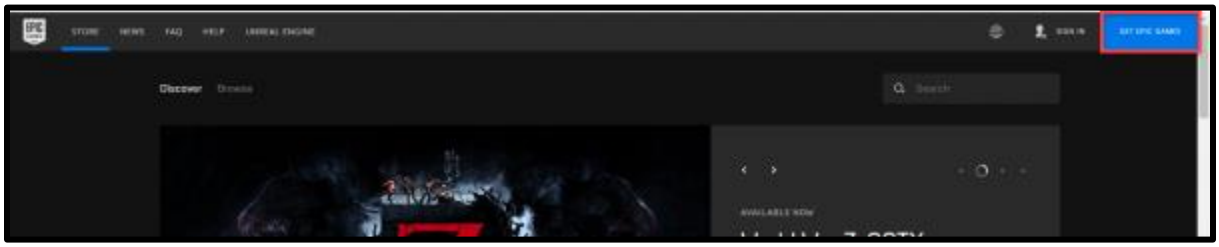
You can also choose to download the assets and complete project here.

What is the Unreal Engine?

The Unreal Engine is a game engine created by Epic Games. With the focus on powerful 3D graphics, this engine has been used to create many popular AAA games, including: Fortnite, Bioshock, Rocket League, and many more.

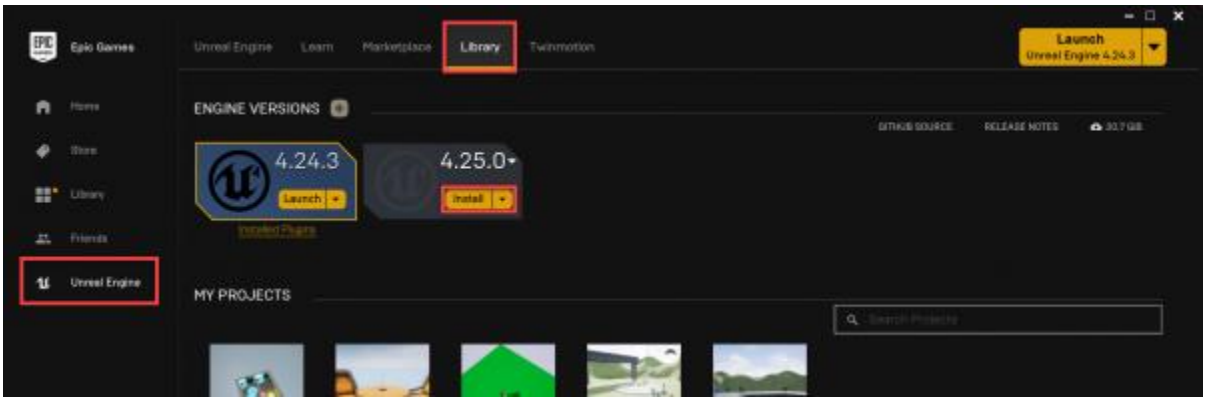
Installation

To install the Unreal Engine, we need to download the Epic Games launcher. You may already have this, but if you don't just go to the Epic Games website, and click on the **Get Epic Games** button in the top-left.



Go through the installation process and when it prompts you to login, do so. This can be done by creating an Epic Games account, or by logging in with your Google account.

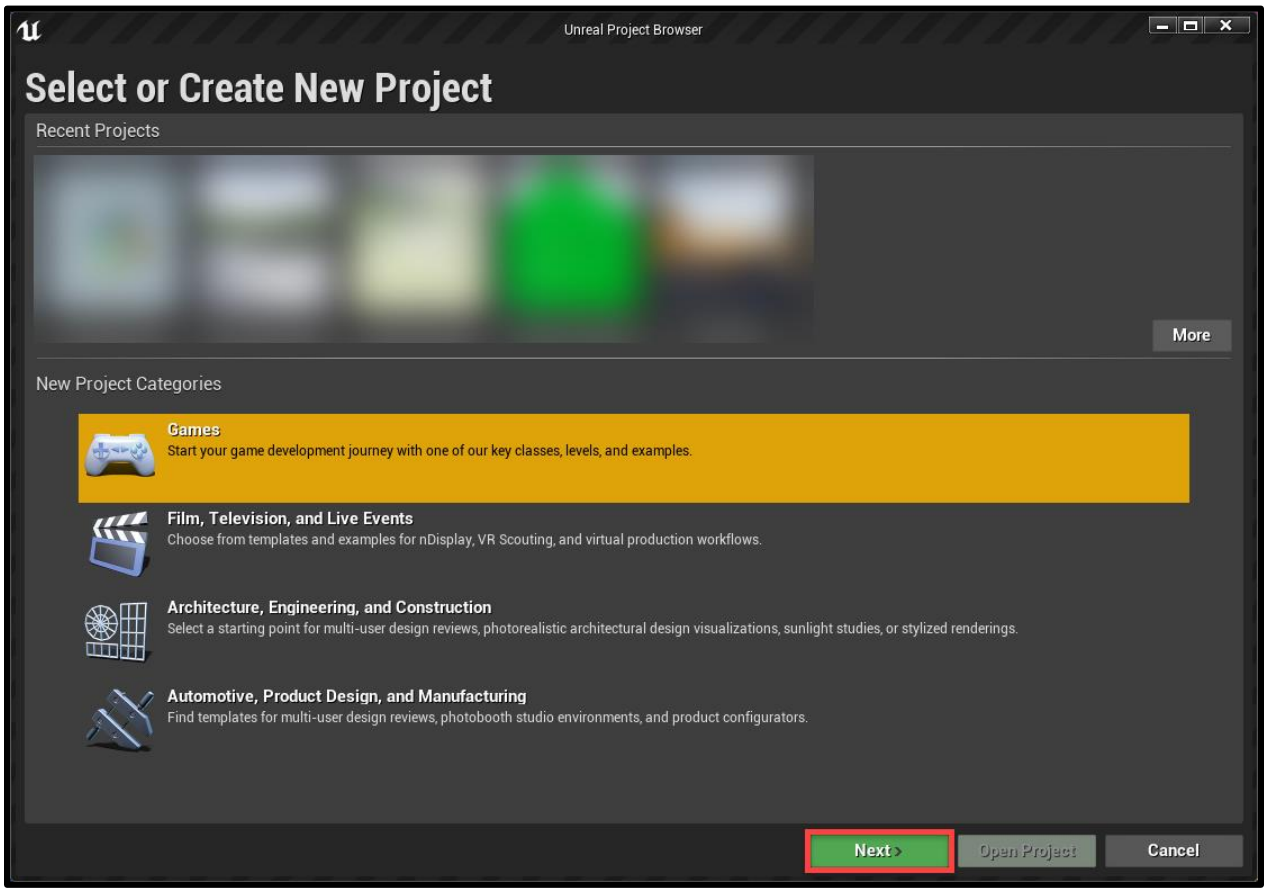
Once you've logged in and have the launcher open, navigate over to the **Unreal Engine** page. Here, you want to go to the **Library** screen and click the install button. I've already installed the engine, so you'll just see one version appearing.



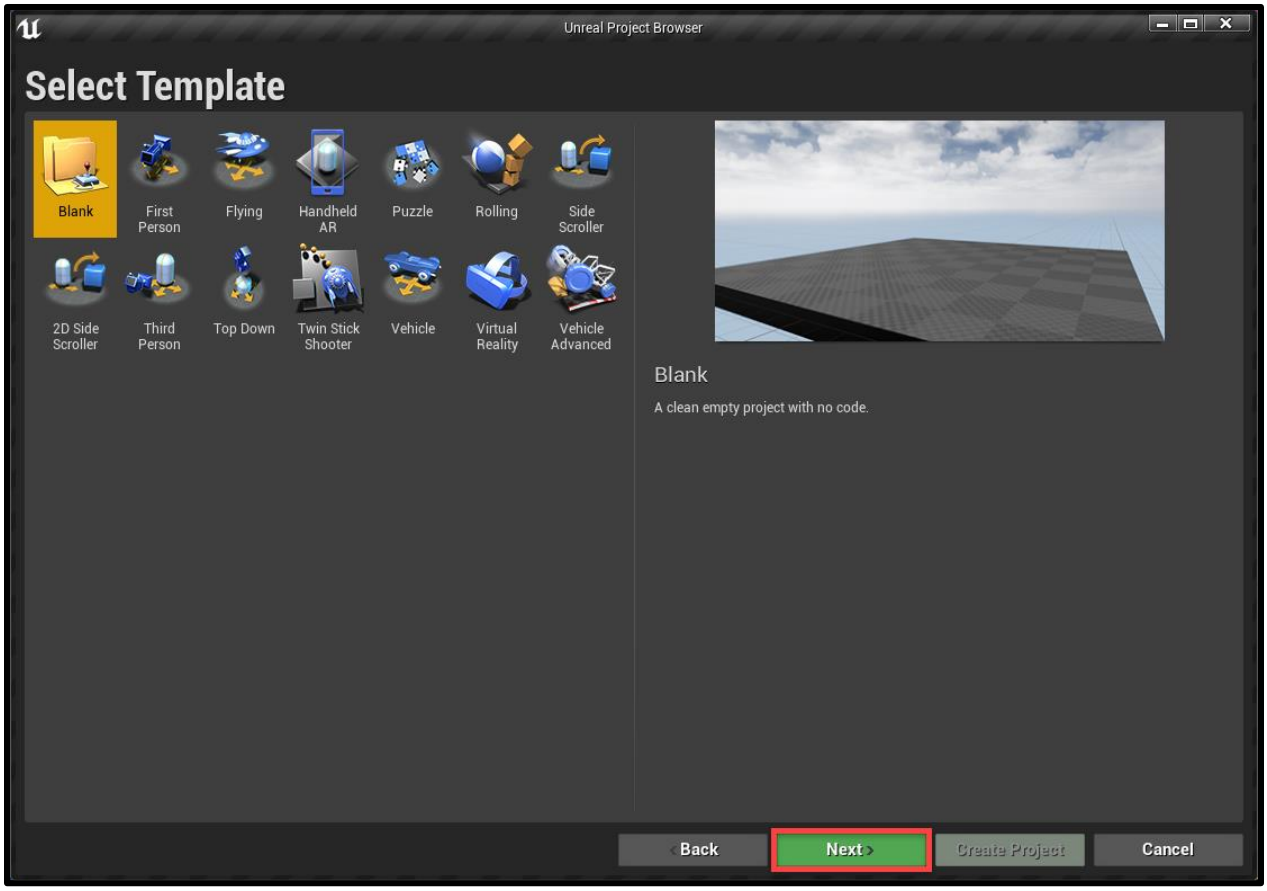
This may take some time as the engine can be around **15 GB** in size.

Creating a New Project

Once that's complete, click on the yellow **Launch** button to open it up. Here we can choose what we want to create. Select **Games** then click the **Next** button.



The next screen allows us to select a template. These are a really good learning resource to see how certain systems can be created in the engine. For now though, select **Blank** and hit next.



Finally, we have the project settings. There's a few things we need to make sure we have selected.

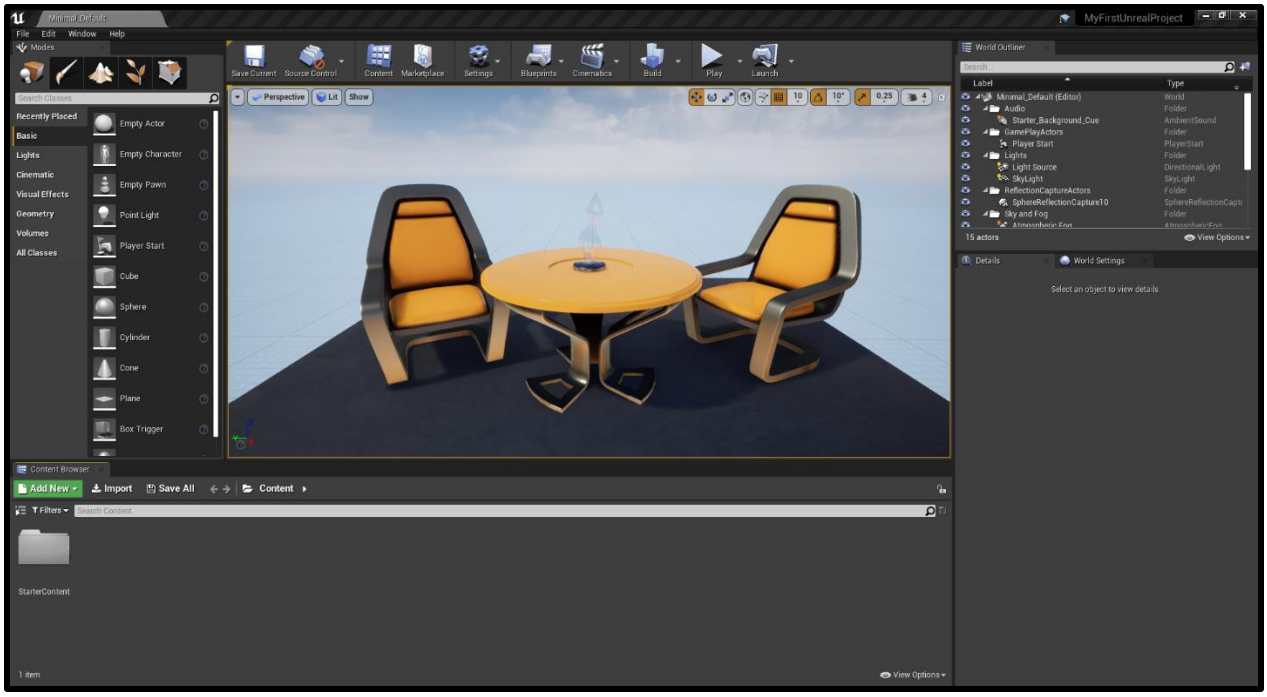
1. Make sure **Blueprint** is selected rather than C++.
2. Make sure **With Starter Content** is selected rather than No Starter Content.

Down at the bottom, we can choose the location to store our project (it will be around 800 MB) and the name. Once you've done all that, click on the **Create Project** button to open up the editor.

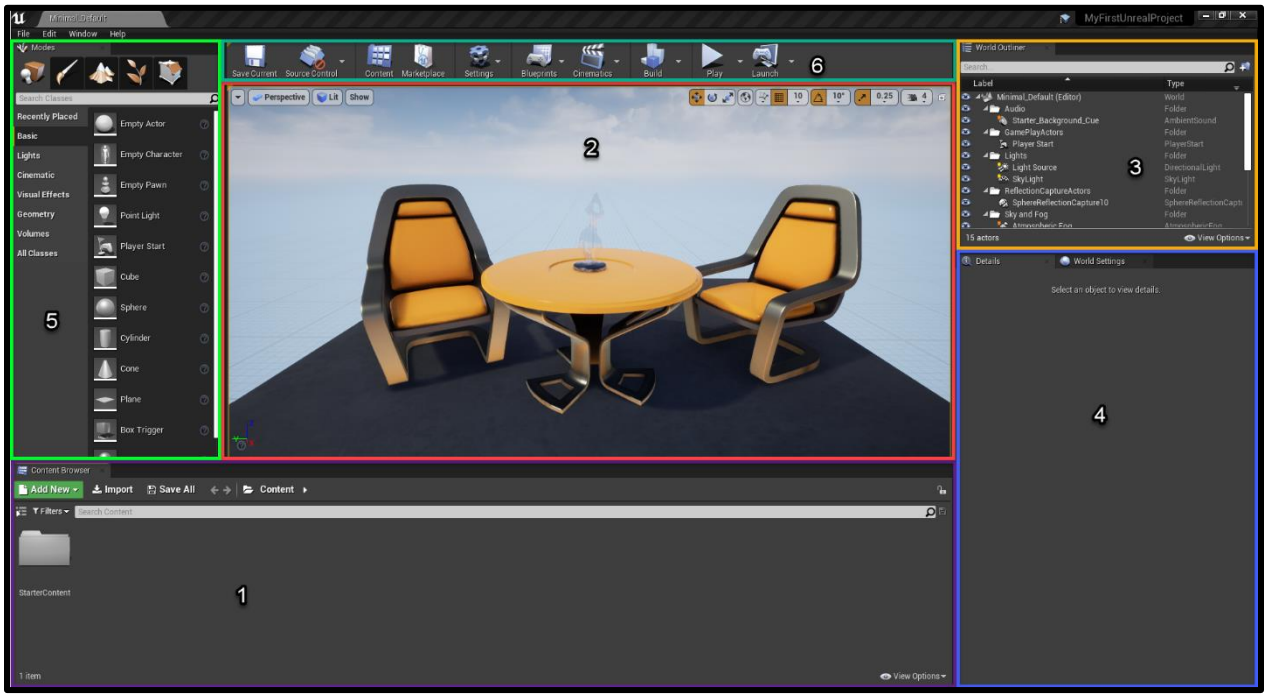
Quick Note: When the editor opens you may see a Steam VR window popup. We can just close that since we're not working in VR.

Editor Overview

Welcome to the editor. This is where we'll be creating our game! It may look daunting at first with all the windows, buttons and options - but let's go ahead and try make some sense out of it all.



This is the level editor and the main screen that appears when opening Unreal. The engine is split up into a number of different editors (material editor, blueprint editor, animation editor, etc). Each serve a specific purpose to help us create games. The level editor here is what we use to build our levels. Place down objects, setup the lighting and overall manage the project.

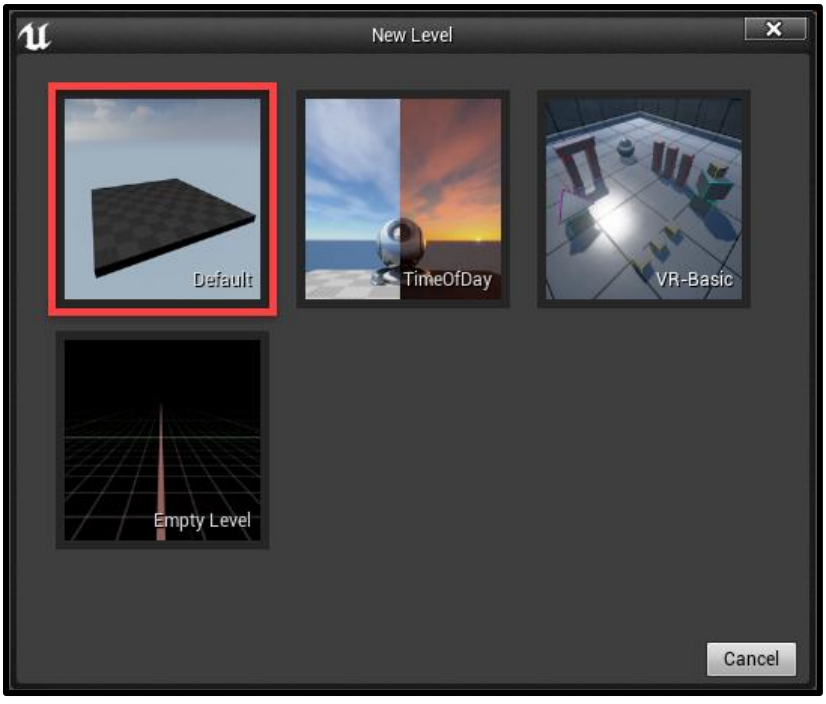


1. First, we have the **Content Browser**. This is like a file explorer and shows us all of the assets (models, sound effects, textures, materials, etc) for our game and the folder structure.
2. Next, we have the **Viewport** screen. Here we can see into our level, move the camera around, select and move around objects.
3. This is the **World Outliner** and it shows us all of the objects in our current level.
4. The **Details** panel shows us the inner workings of a selected object. Try selecting an object in the viewport window and you'll see that various properties such as the position, rotation, scale, rendering, etc appears for that object.
5. This is the **Modes** panel. Here we can create primative 3D models, create lights, triggers, VFX, etc.
6. Finally, the **Toolbar**. This is where we can test out the game, build it, save the game, etc.
7. We'll be going over more about each panel as we get there in the tutorial.

Creating a New Level

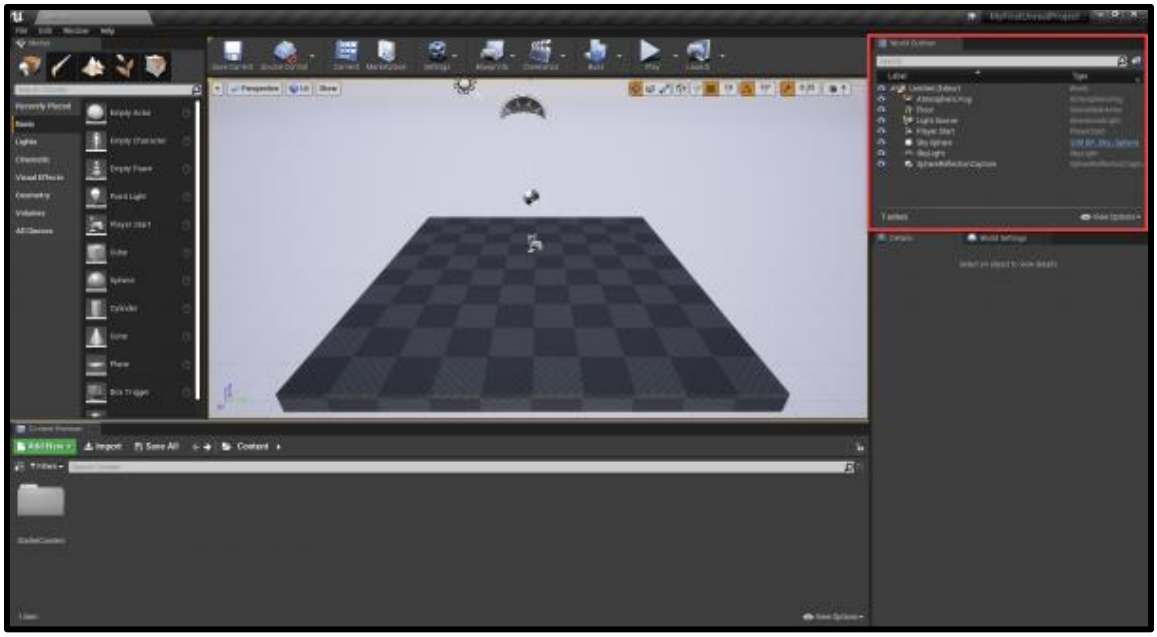
In Unreal Engine, we have the concept of **levels.** Levels are what splits a game up. Different stages, areas, etc. Right now, we're in one of the starter levels that comes with the starter content. Let's now go ahead and create a new one.

1. Go **File > New Level**
2. Select the **Default** level



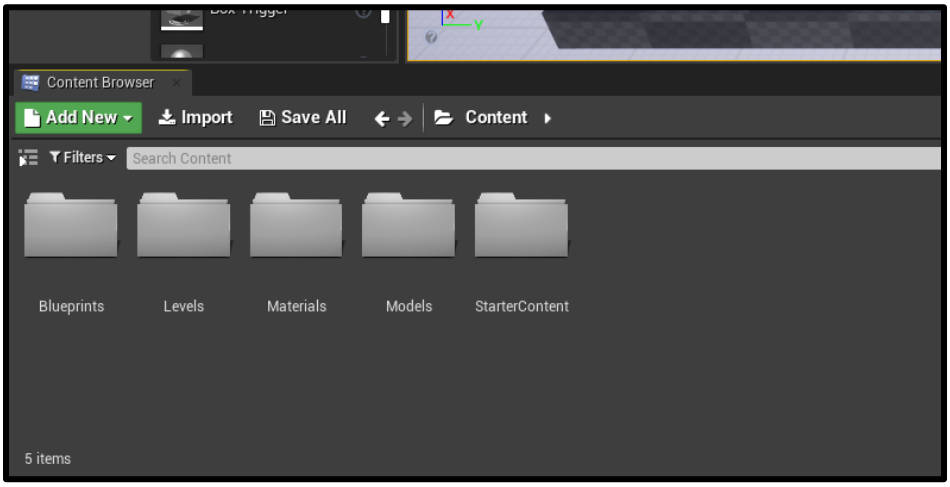
This will create us a brand new level with a few things already in it.

Over in the **World Outliner** you'll see that we have a floor model, a light, a player start and a few skybox objects. These help build the scene we see here.

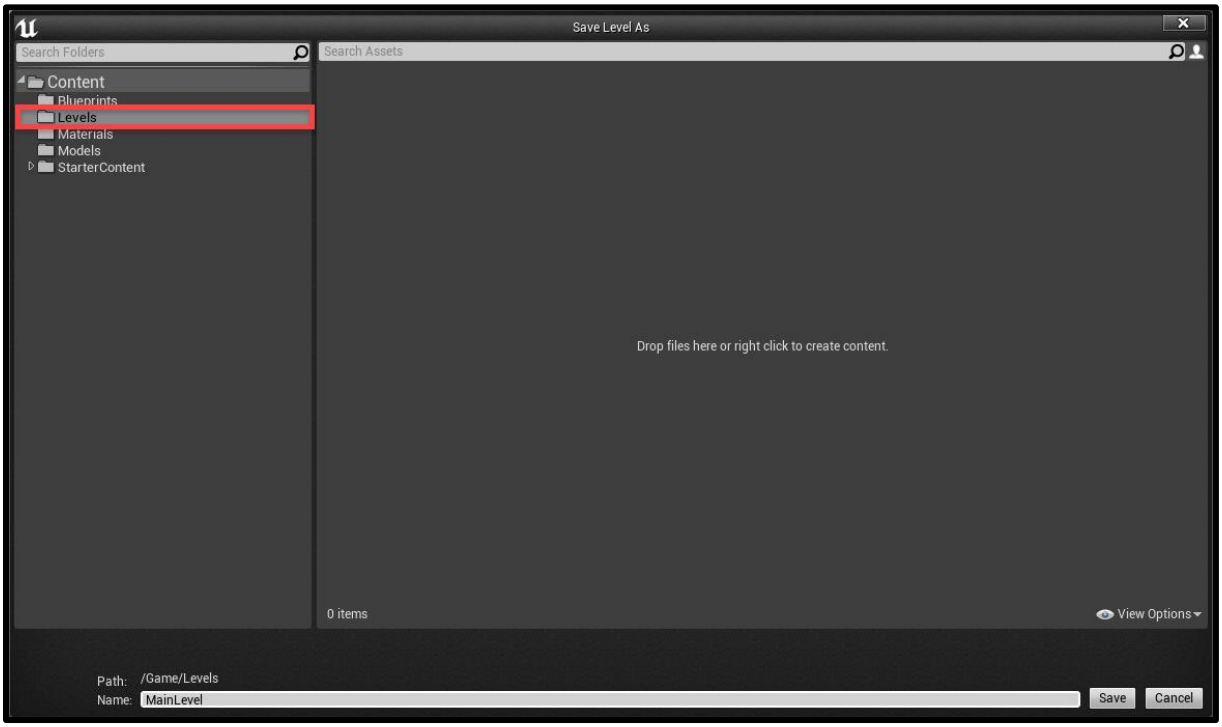


Before we continue, let's go down to the **Content Browser** and create 4 new folders to store our future assets (right click and select *New Folder*).

- Blueprints (to store our blueprints such as the player, gems, enemies, etc).
- Levels
- Materials
- Models

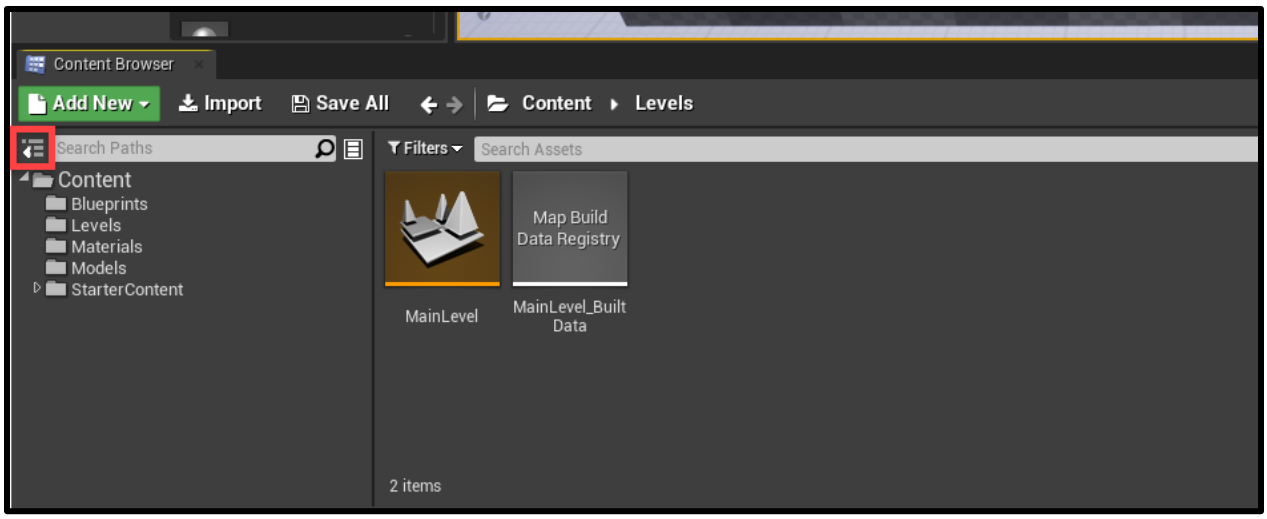


We can now save our level (CTRL + S) to the Levels folder. Give it the name of **MainLevel**.



If we open the levels folder, you'll see that we have the **MainLevel** asset there. If we ever want to open the level, we can just double click on it.

Quick Note: To make navigation easier, click on the *Show Sources Panel* button. This will give us an overview of the folder structure.



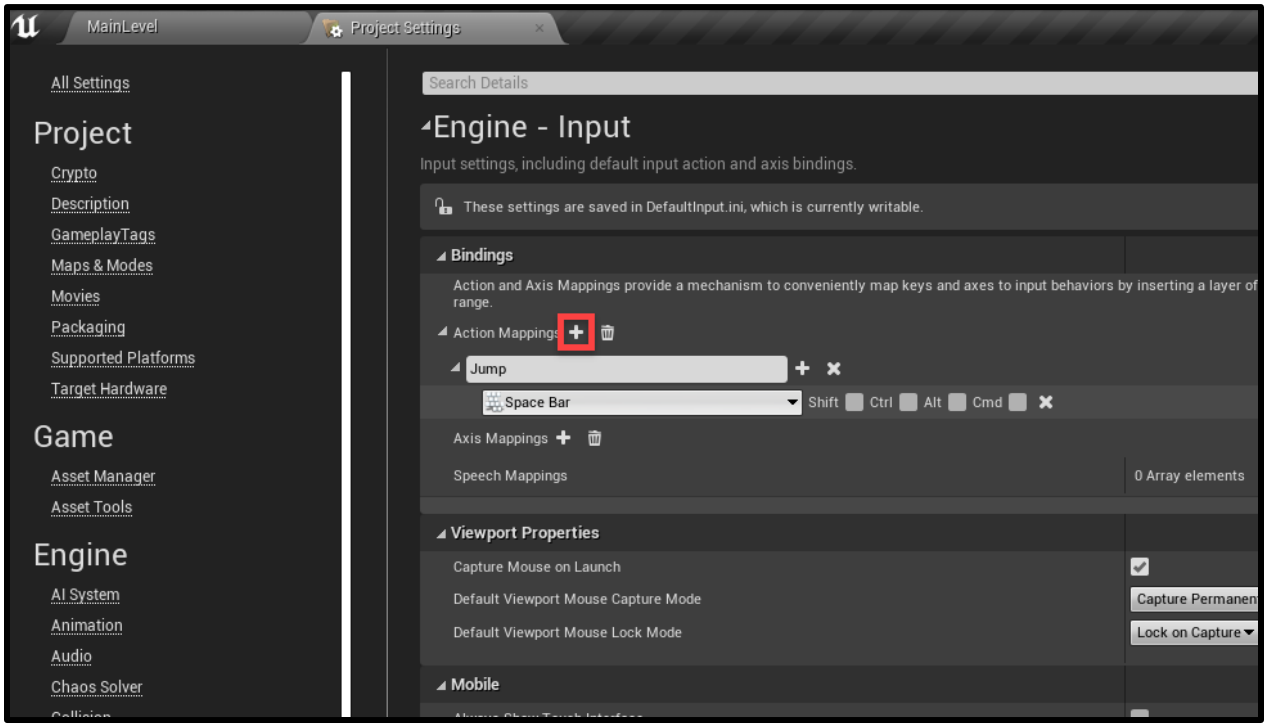
While we're here, let's also import our intro assets. These are a few 3D models and materials we'll be using and can be downloaded [zva-file-link id=11843 text="here"][/zva-file-link]. Inside of the .ZIP file, there will be a *Models* and *Materials* folder. Place the contents in their respective counterparts in the Unreal editor content browser. It may pop-up with some import options, just click import on all of them.

Creating the Player

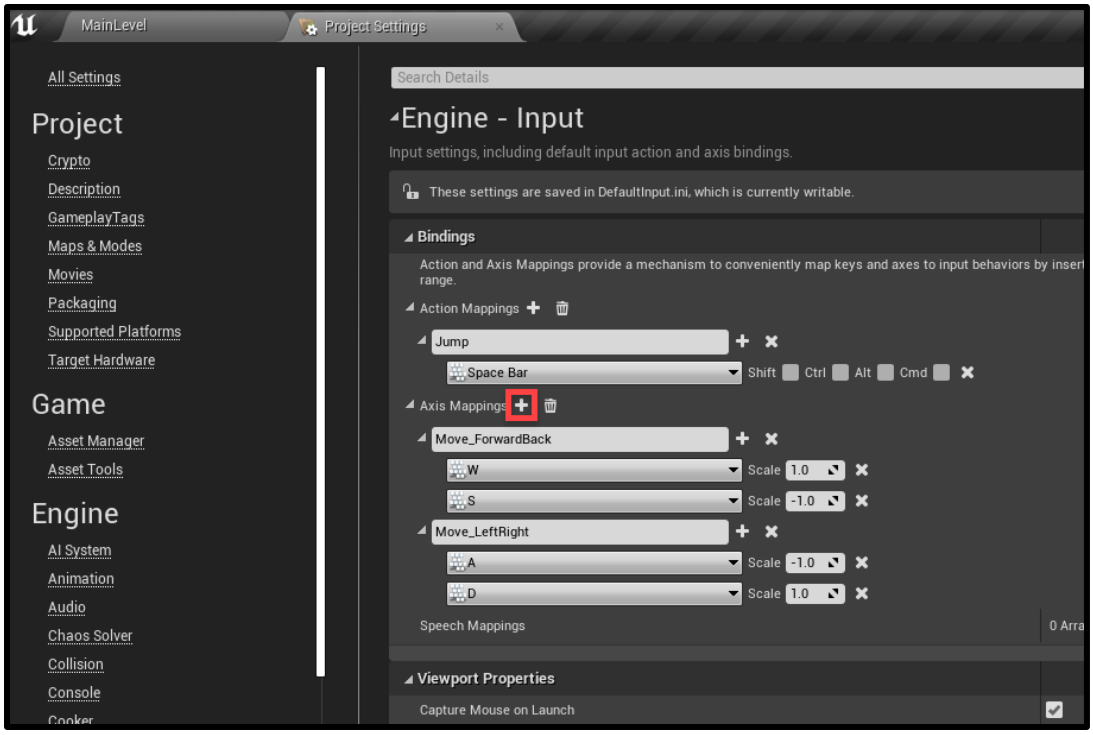
Now it's time to create our player! We'll be able to move around and jump on platforms. But before we jump in, let's set up the controls, because we need to know which buttons are for moving and which are for jumping.

Open up the **Project Settings** window (*Edit > Project Settings*). On the left hand side list, select **Input** to go to the input options. We have *actions* and *axis*'. Actions are single button inputs (i.e. press a button and something happens). Then we have axis inputs which change a value based on the given input (i.e. a controller joystick).

First, let's create a new action called *Jump*. Set the key to be the space bar.

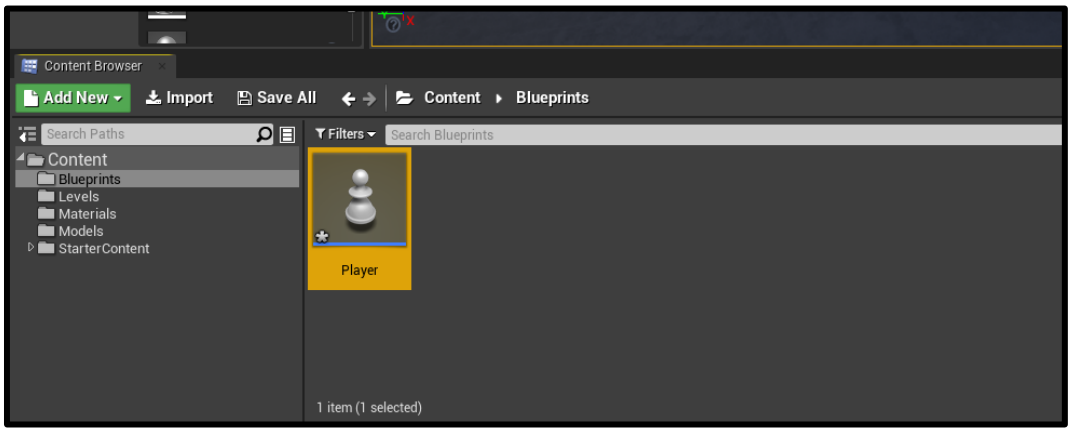


Next, let's add two new axis mappings. An axis for moving forward and back as well as an axis for moving left and right. Make sure to set their scale as seen in the image.



Now that we've got our inputs setup, let's go back to the main level window (tab at the top of the screen) and create our player **blueprint**. In the Unreal Engine, blueprints are objects that we can have in our games which each have their own properties, scripting and various other things. Think of the player, enemies, gems, etc. These are all blueprints. They have their own logic and they can be created/destroyed. To create a blueprint, let's go to our *Bluerints* folder, right click and select **Blueprint Class**.

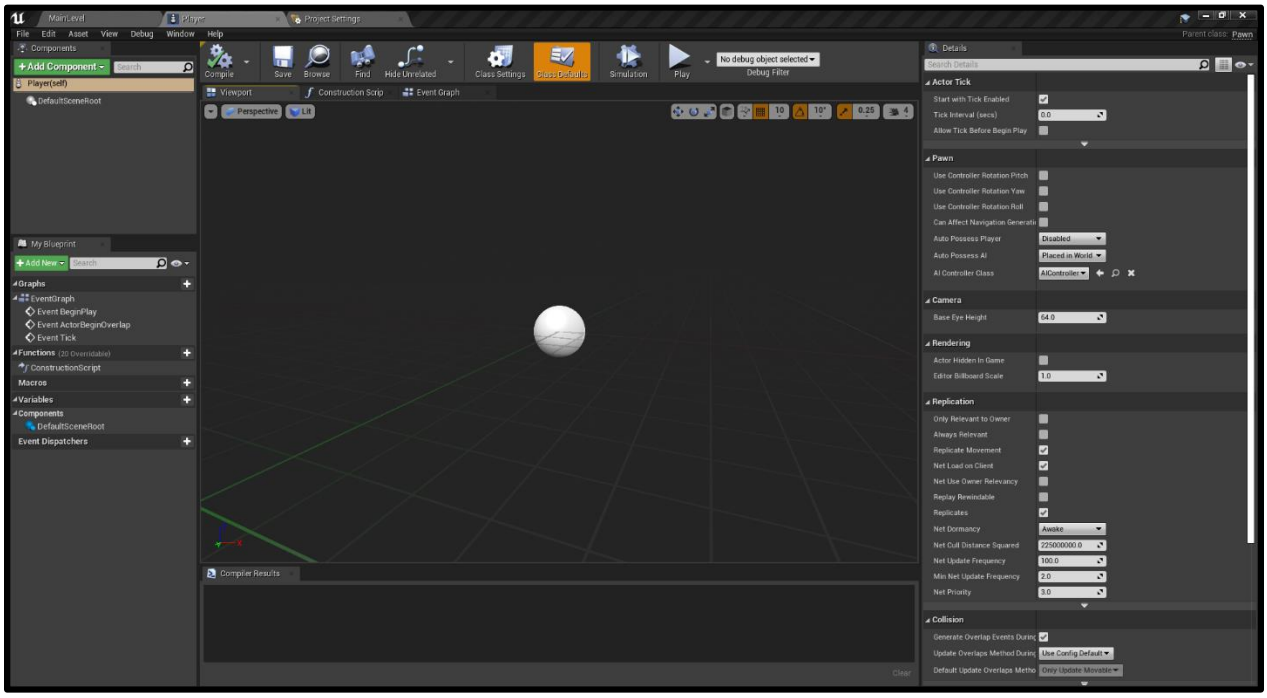
It will ask us to choose a parent class. In Unreal, we've got a number of base classes we can expand upon depending on what we want to create. Select the **Pawn**. As the player we can possess a pawn so that it will receive our inputs. Call the blueprint: *Player*.



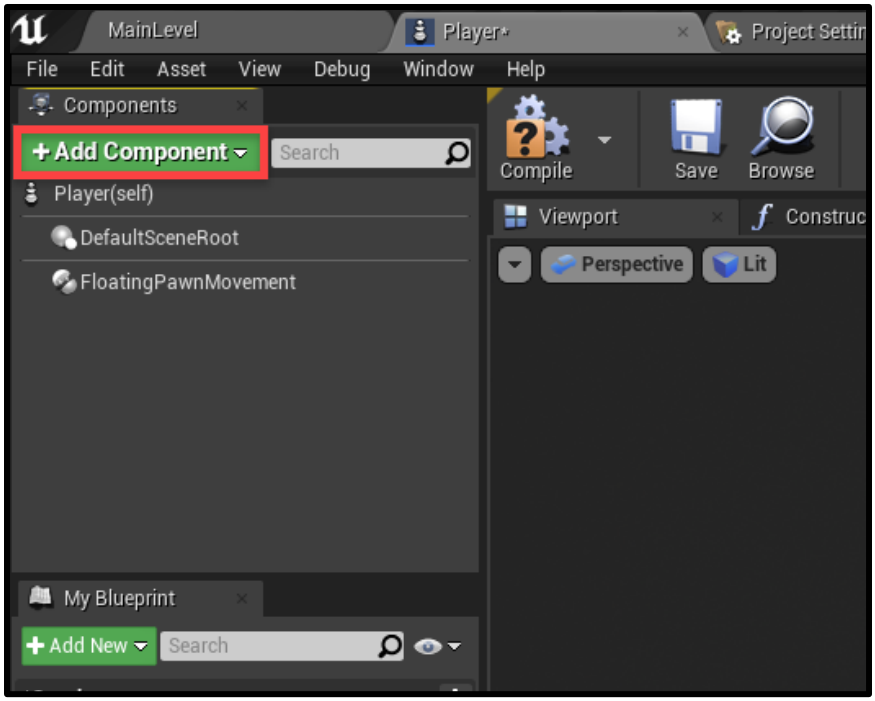
Now double click on the blueprint to open it up in the Blueprint Editor. It may prompt you to open it in the full editor. If so, click the button.

So here we have the Blueprint Editor. Here, we can create our player object by doing many things.

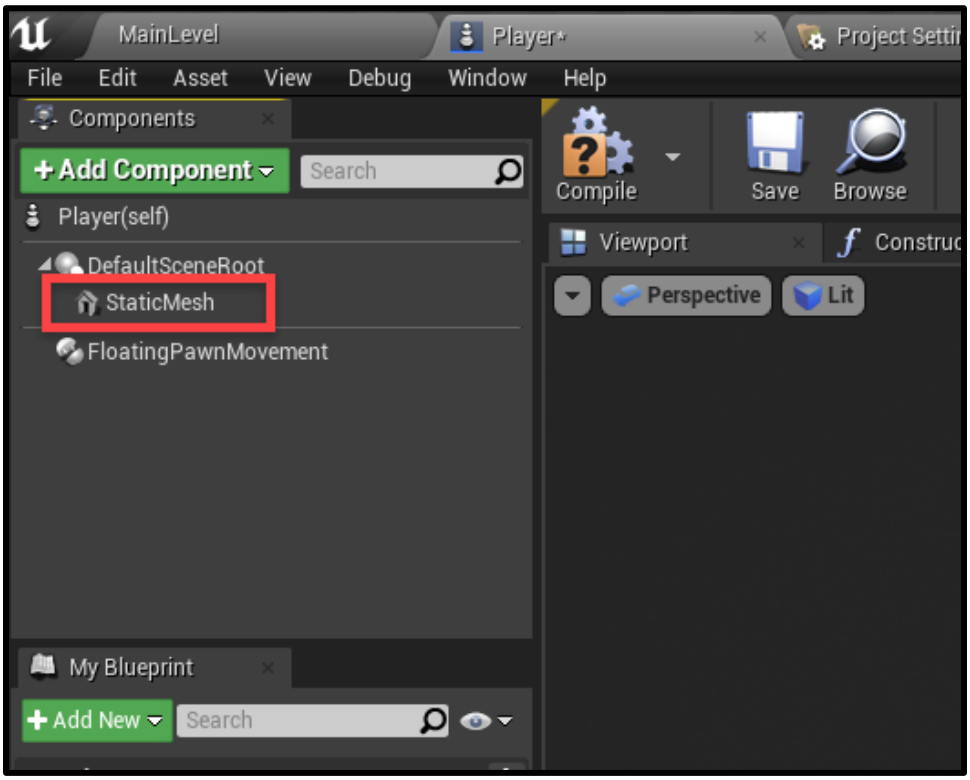
- We can give the player models, lights, effects and other various objects to build them up.
- We can use the blueprint graph to create logic for the player.
- We can tweak the player's properties to our liking.



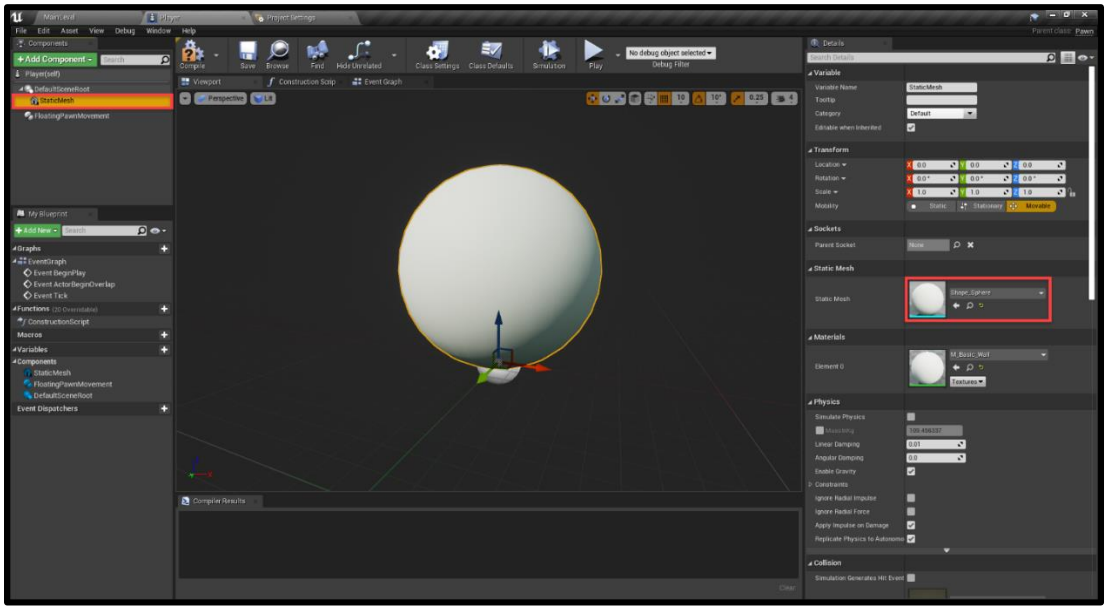
First thing we need to do is give our pawn (the type of class the player is) the ability to move. Or give it the knowledge of how to do so. For this, we can click on the **Add Component** button and search for *FloatingPawnMovement*. This will add the component to our component list.



Next, we need to give the player some physical presence. Click **Add Component** again and search for *StaticMesh*. This is basically a 3D model.

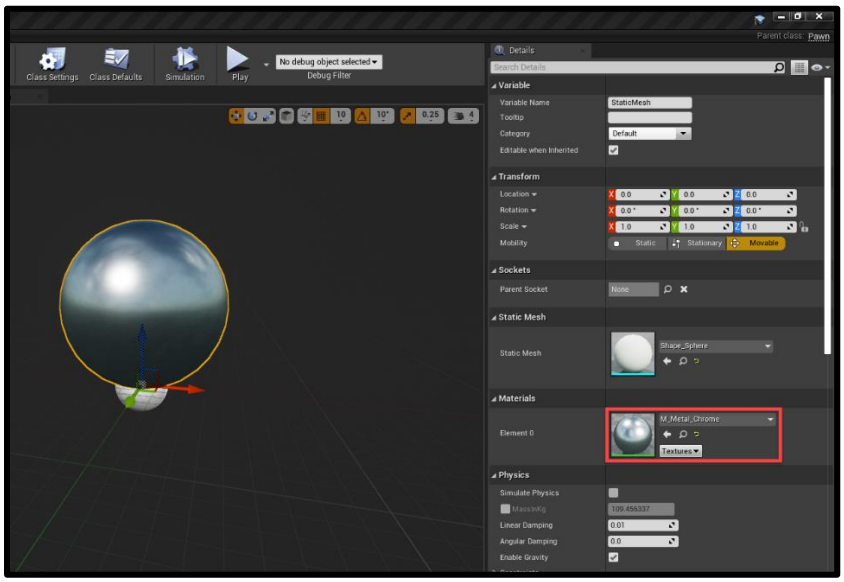


To assign the model, select the *StaticMesh* and over in the **Details** panel, set the *Static Mesh* field to be *Shape_Sphere*.



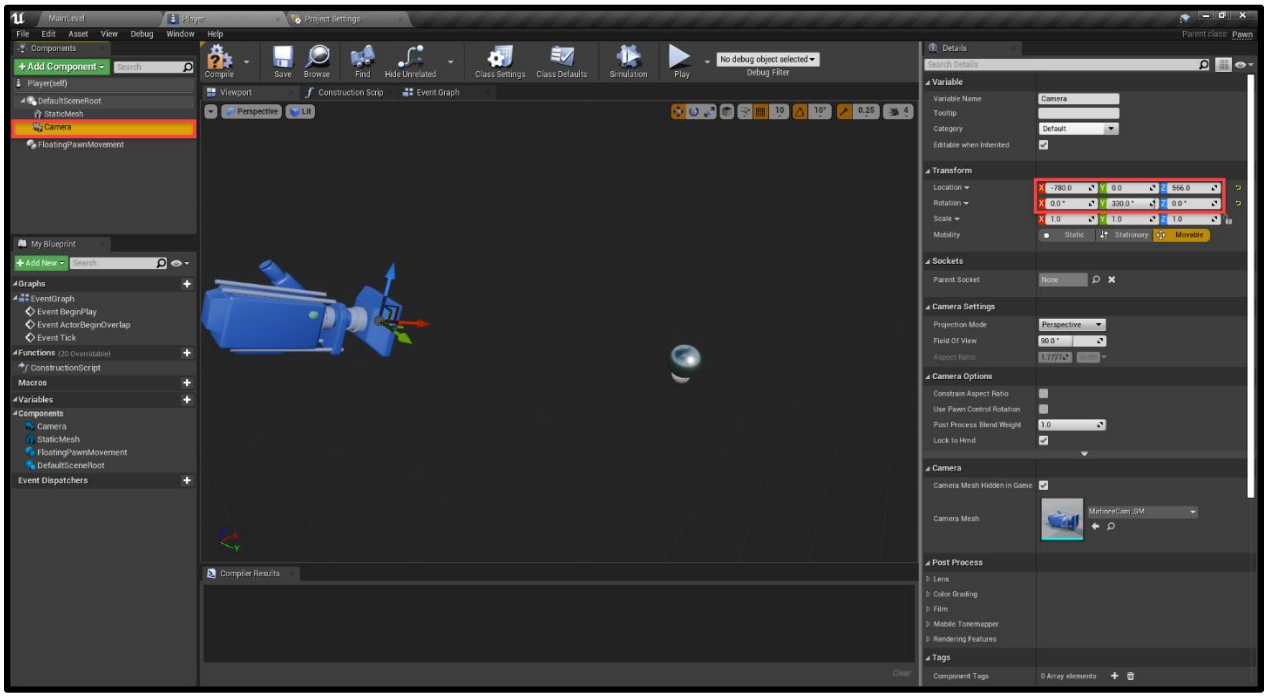
Quick Note: You can move the camera around in the center viewport. Hold *right-click* and use WASD to fly around.

A plain white sphere doesn't look that good, so let's also change the *Material* to *M_Metal_Chrome*. When we set up lighting later on this will reflect it all and look pretty good.

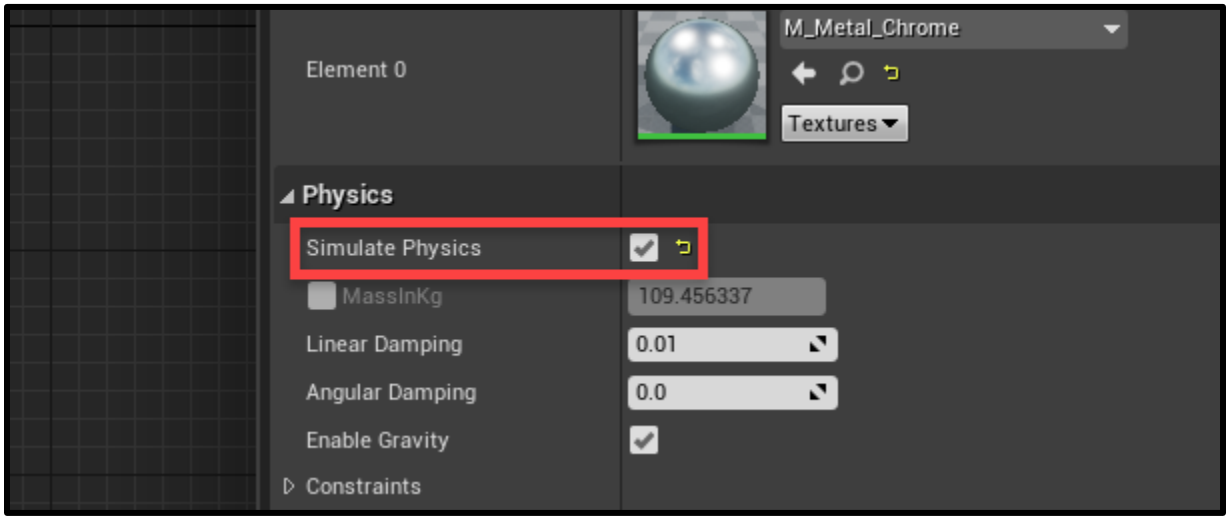


The final component is going to be a camera. Because we need a way of looking into the world and following the player. Add a new *Camera* component. In the **Details** panel...

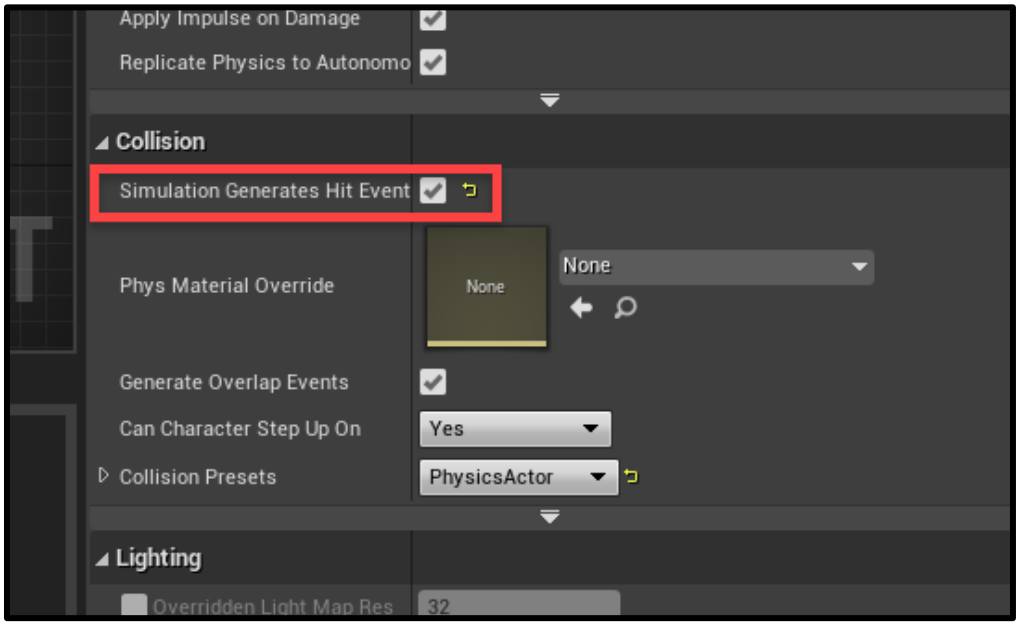
- Set the **Location** to *-780, 0, 566*
- Set the **Rotation** to *0, 330, 0*



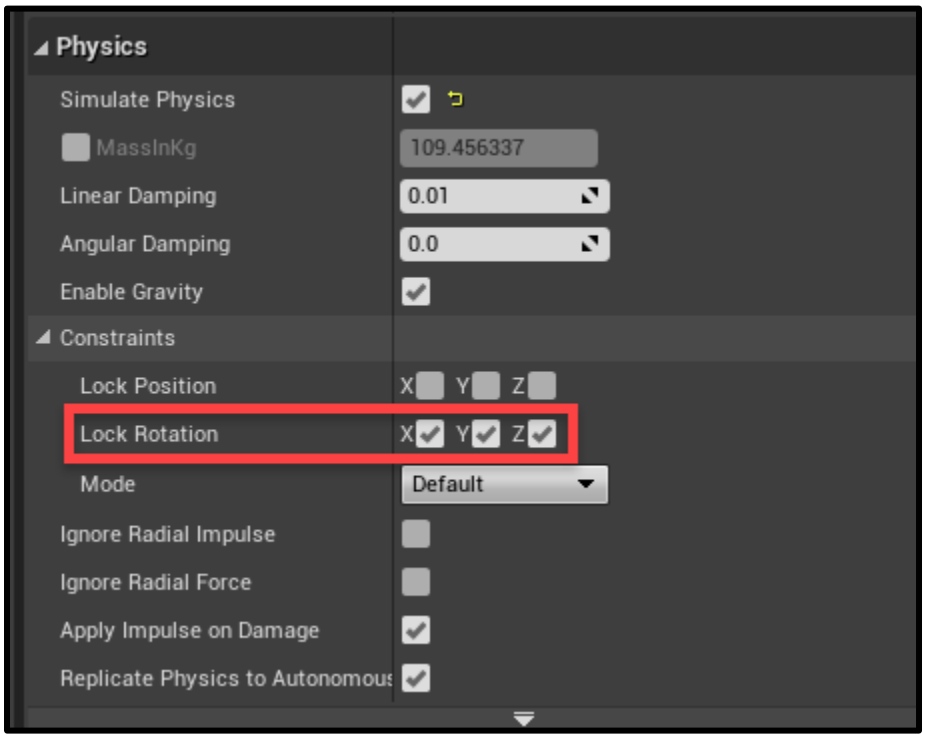
Select the **StaticMesh** and enable *Simulate Physics*. This will give us gravity and the ability to collide with things.



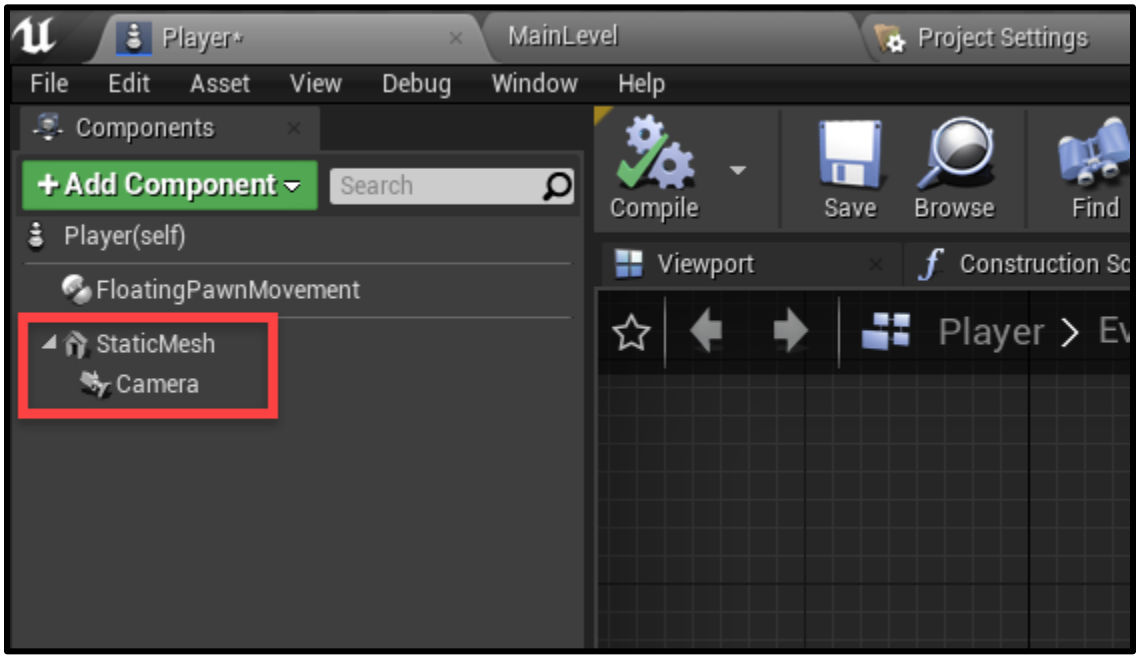
We also need to enable **Simulation Generates Hit Events**. This will allow us to detect collisions.



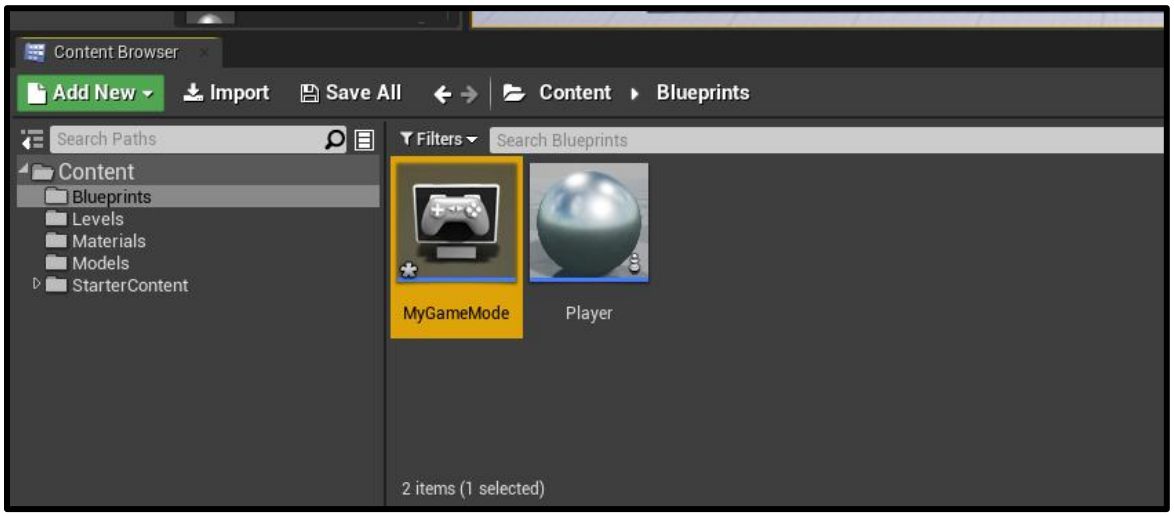
To prevent the player from rolling on its own, let's open the **Constraints** tab and enable **Lock Rotation** on the X, Y and Z axis'.



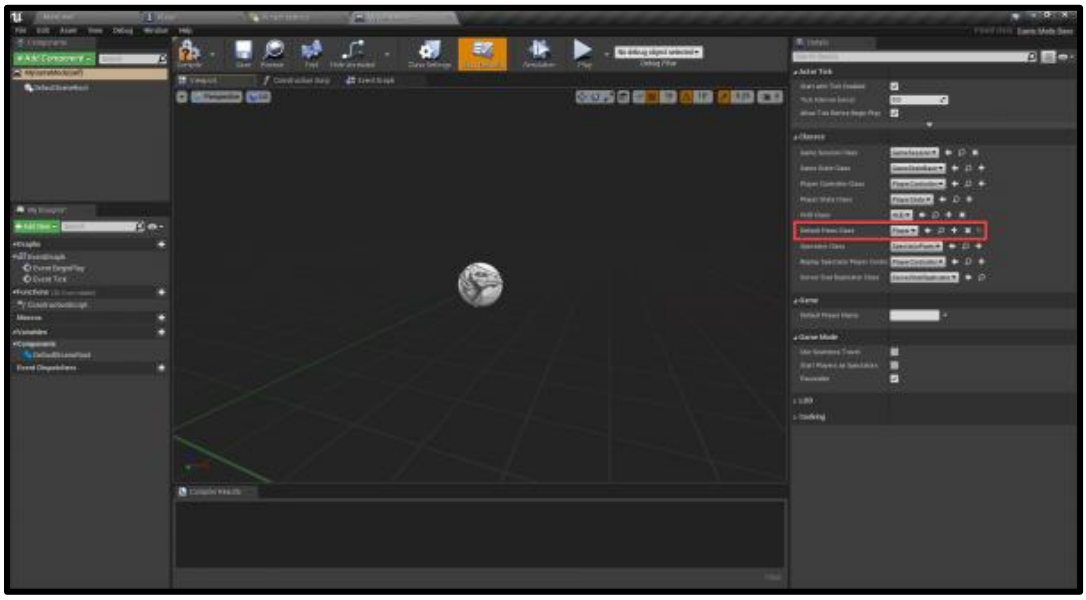
Finally, let's make the static mesh the root node. To do this, click and drag the static mesh on top of the *DefaultSceneRoot*. This will replace it and make the camera a child.



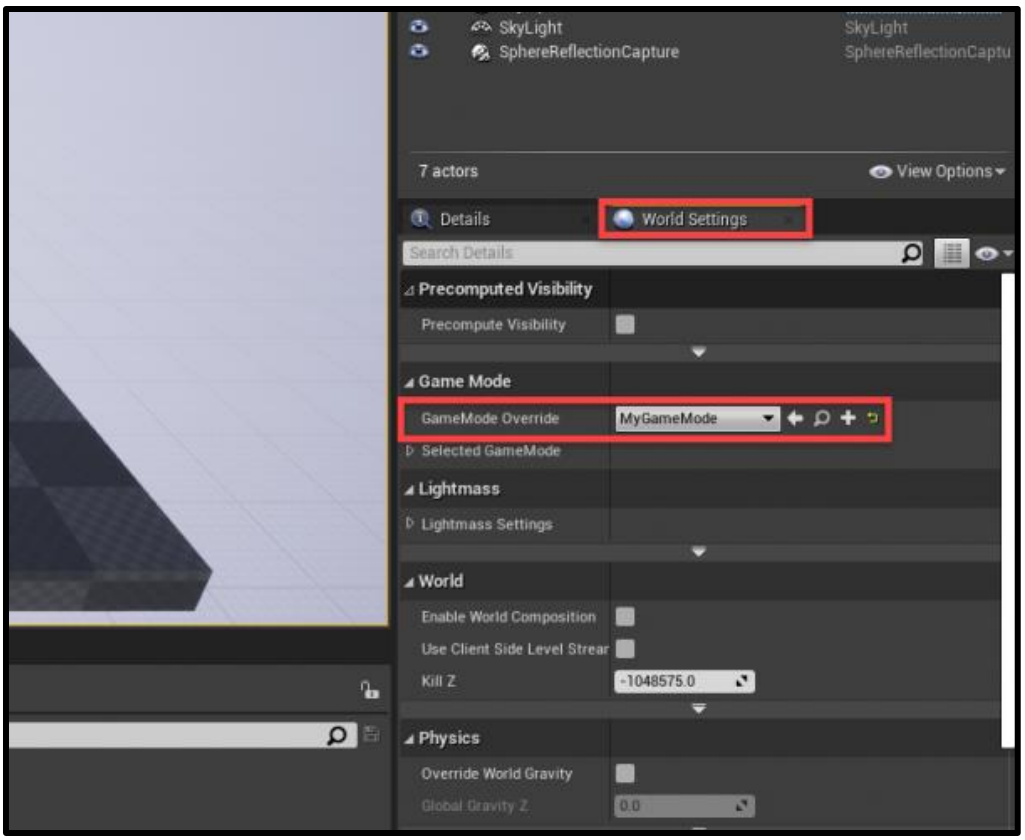
Let's now test this out in our level. Back in the level editor (click on the MainLevel tab), we need to first create a game mode blueprint. This will tell the game what object the player is, etc. In the **Blueprints** folder, create a new blueprint of type *Game Mode Base* and call it *MyGameMode*.



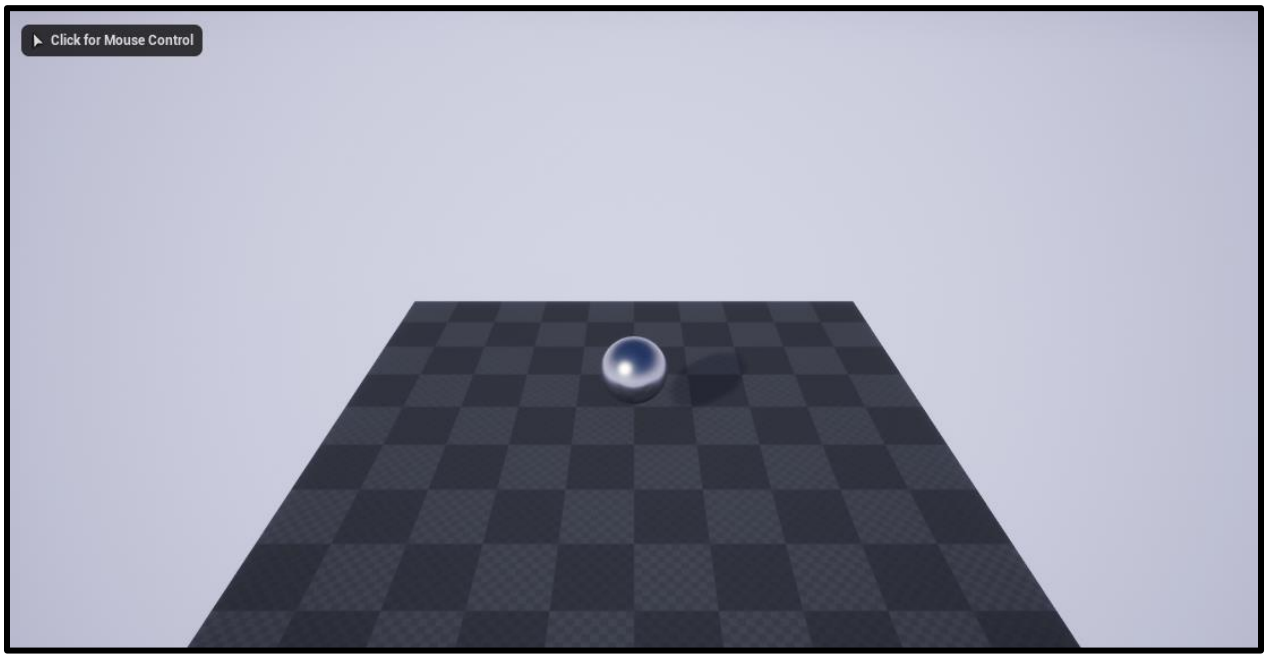
Double click on it to open up the blueprint editor. In the **Details** panel, we just want to set the **Default Pawn Class** to be our *Player*.



Save that, then return to the level editor. In the **Details** panel, click on the **World Settings** tab and set the *GameMode Override* to be our new *MyGameMode*.



Now if you click the **Play** button, you'll see that the center viewport turns into the game and our player spawns in!

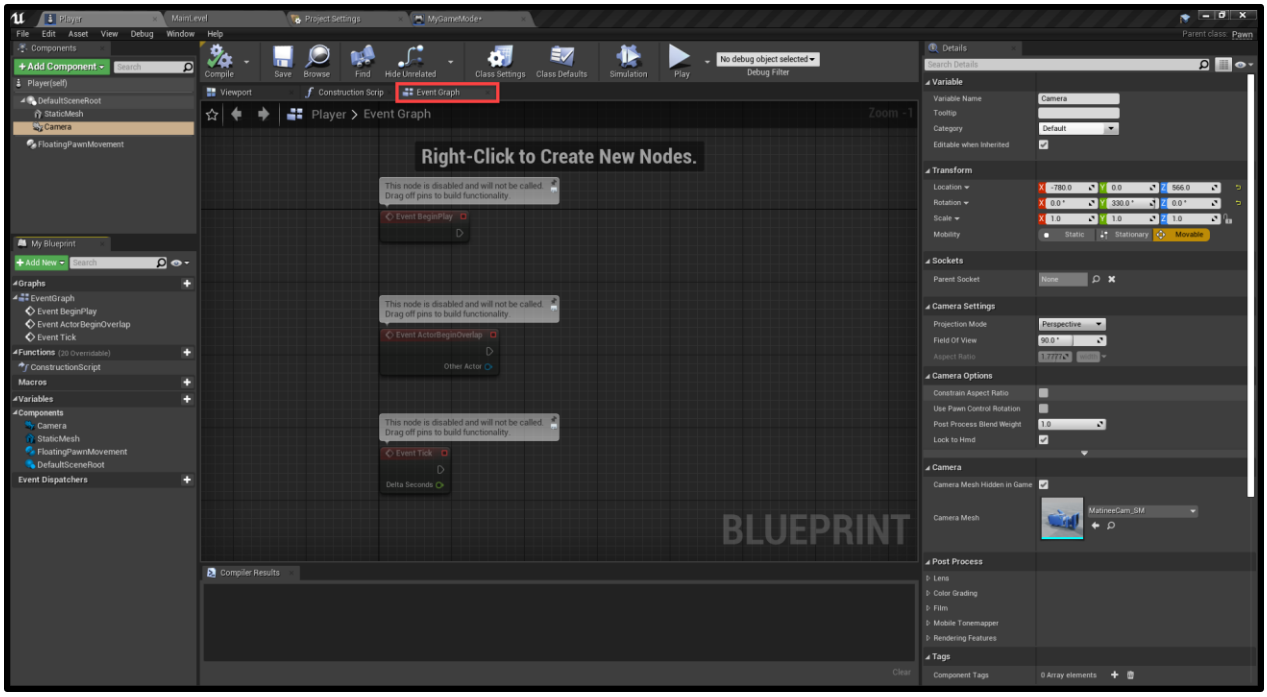


Assigning Logic to our Player

Now that we've got all of the components for our player, let's begin by giving them logic. In the *Player* blueprint, click on the **Event Graph** tab. This will take us to the node graph. In Unreal, we can create logic by creating these graphs. They basically consist of nodes with connections. Certain connections will trigger when certain things happen. We can also have conditions, so when something happens, we can check if it was *a* or *b* then do something based on that.

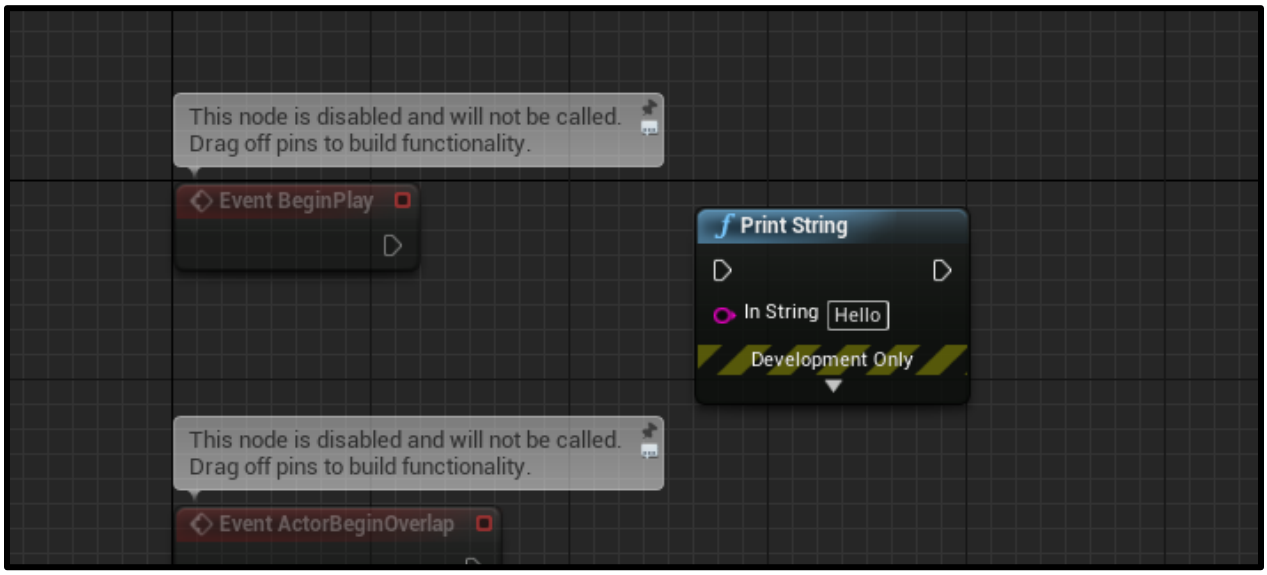
You'll see that there are three nodes already there. These are event nodes (the red ones) and trigger when something happens.

- **BeginPlay** triggers once right at the start of the game
- **ActorBeginOverlap** triggers when we enter the collider of another object
- **Tick** gets triggered every frame

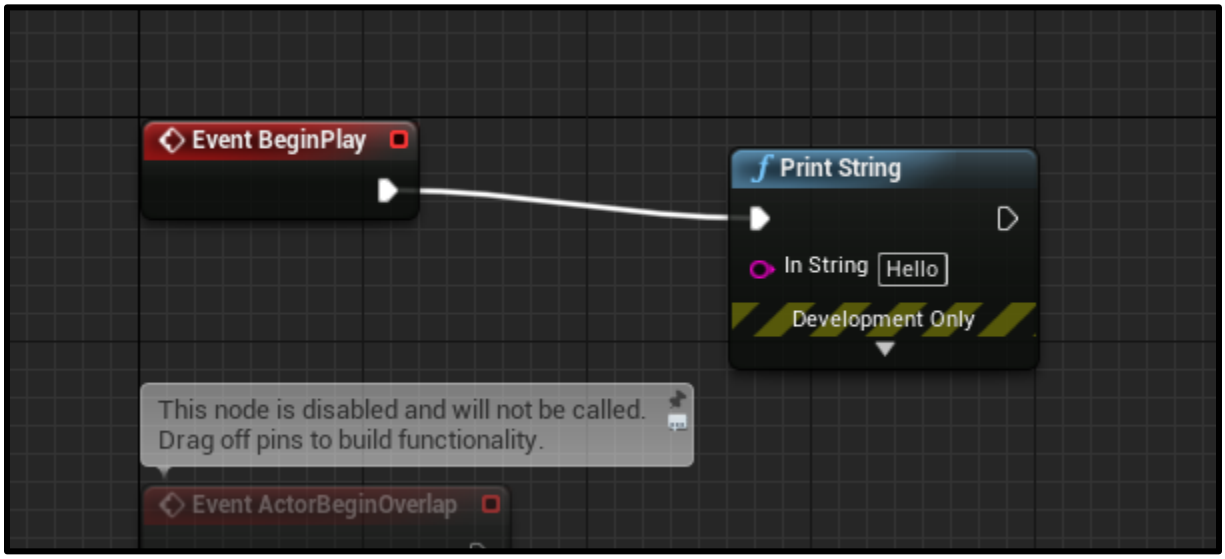


Let's test this out. Right click and search for a node called **Print**. When triggered, this node can print something to the screen.

You'll see that each node has these white arrows. These are how we connect nodes and *power* them. Some such as print have an input and output, white begin play only has an output. This means that when print is powered, it can then move onto another node. Click and hold the **BeginPlay** node's output and drag that to **Print**'s input.

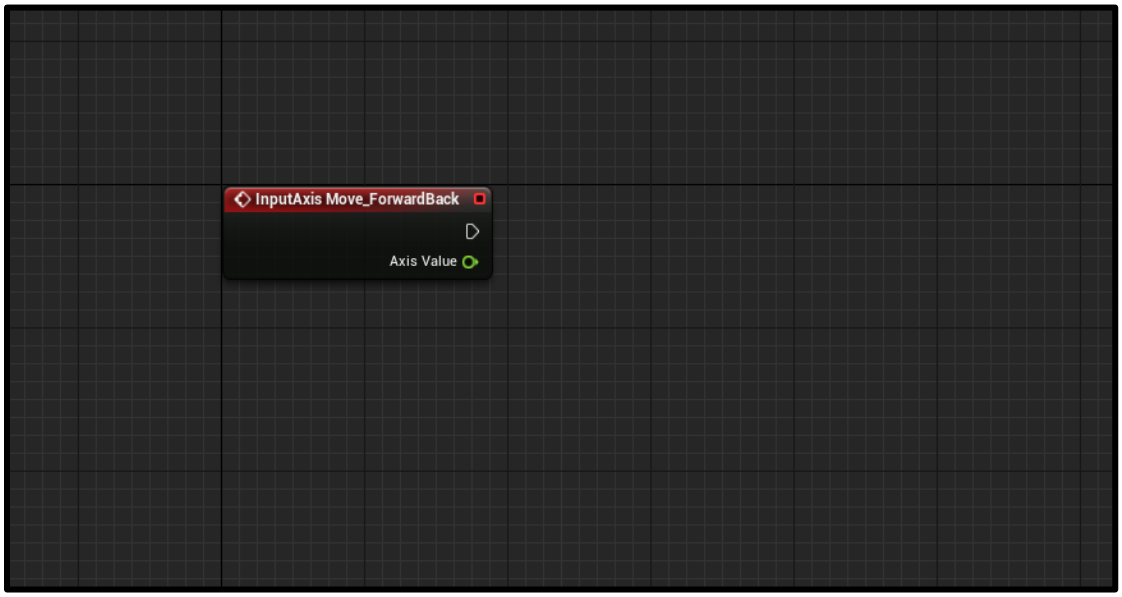


So now when **BeginPlay** is triggered (once at the start of the game), the print node will trigger - printing *hello* to the screen. You can go back to the level editor and test it out.

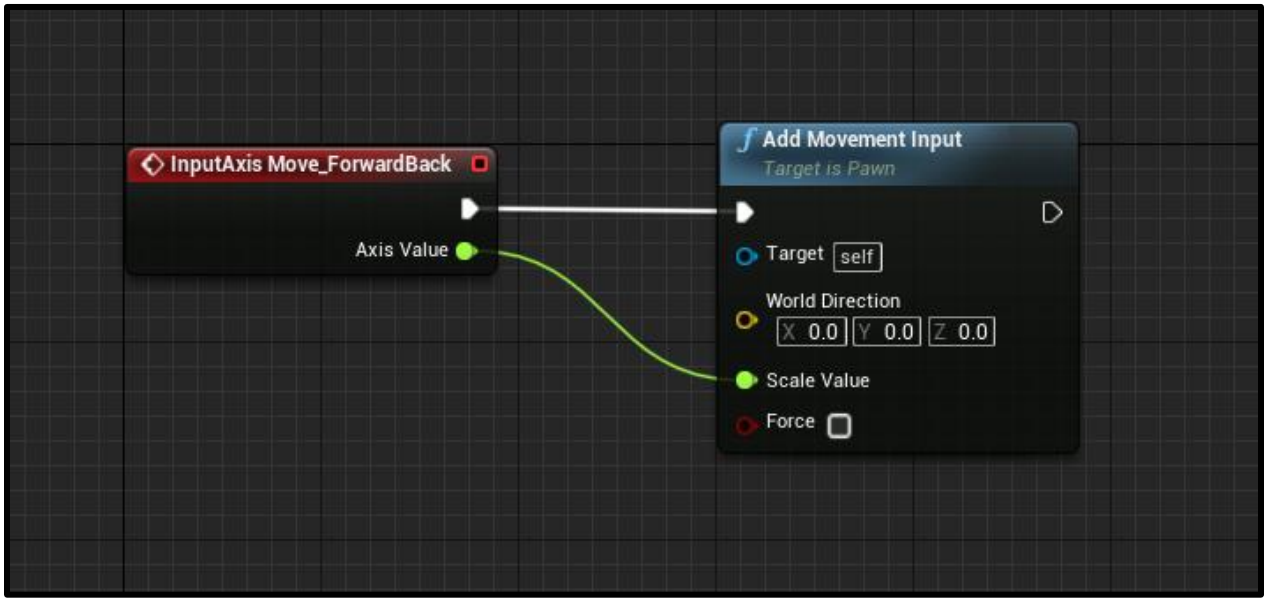


Now let's get started on moving our player around. Select all of the nodes and delete them (delete key). You can also move around with right mouse button and zoom with the scroll wheel.

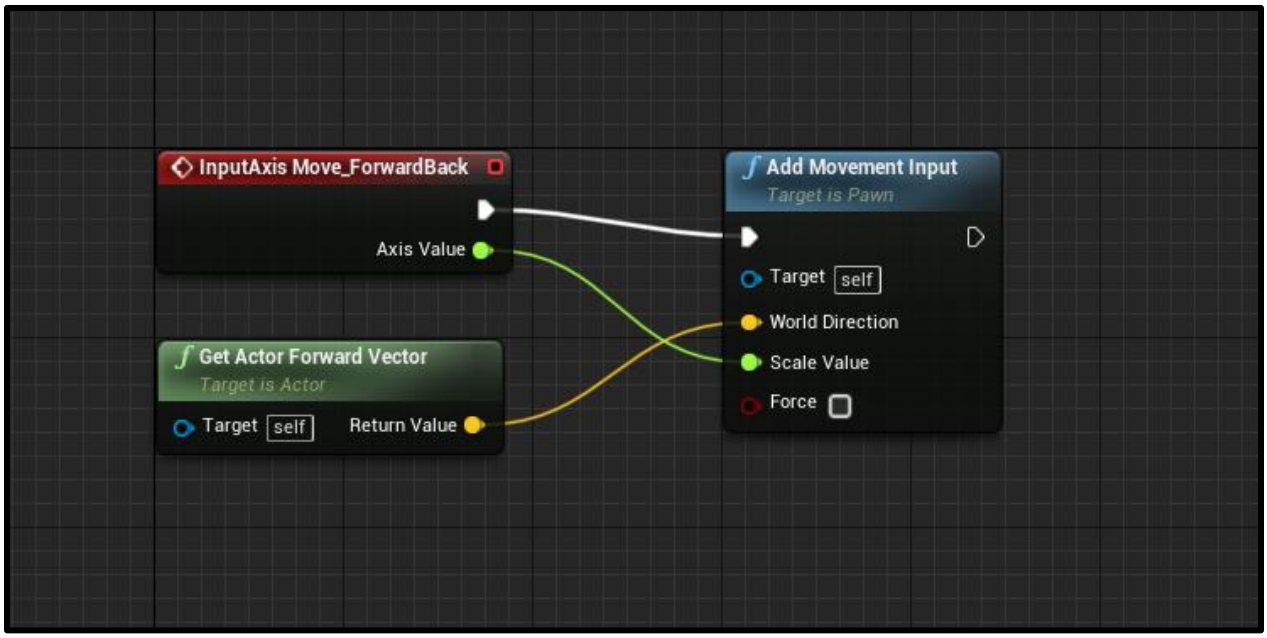
Right click and create a new node called **InputAxis Move_ForwardBack**. You'll notice that this is one of the inputs we created. So when *W* or *S* is pressed, this will trigger. It also has this green **Axis Value** thing outputting as well. Nodes can have both input and output values. For us, the axis value is just the scale value from -1 to 1.



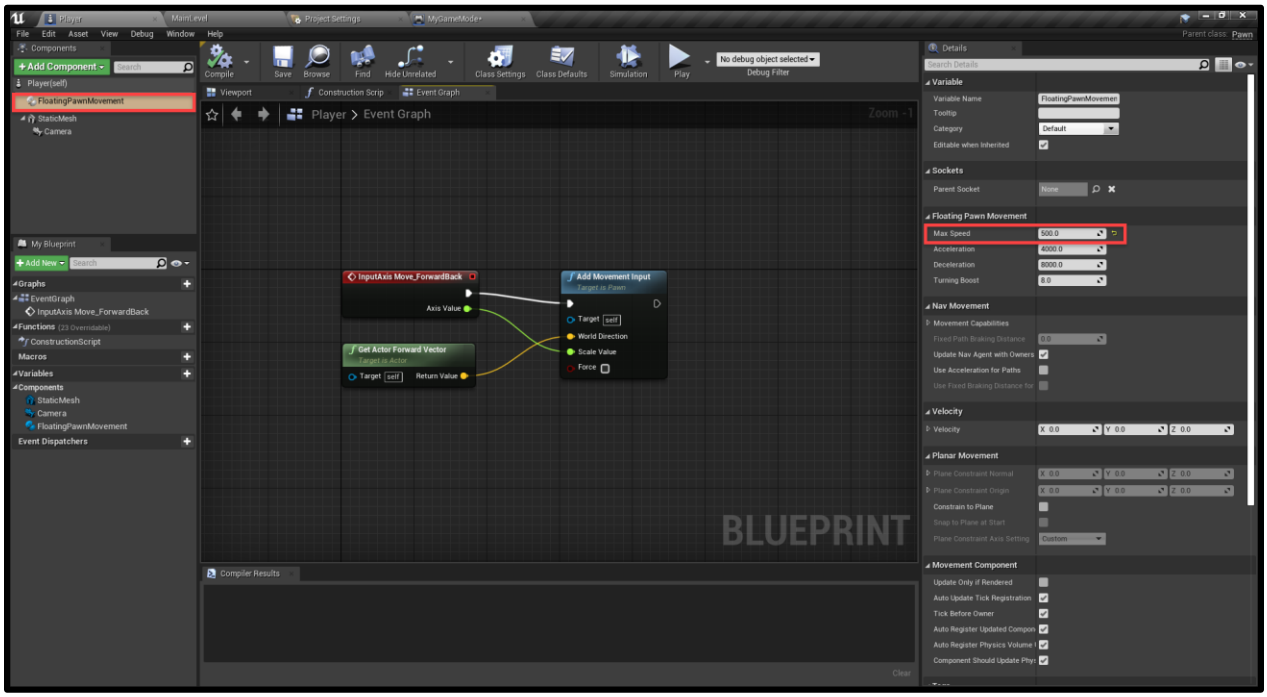
Drag out the node's **flow output** (the white arrow) and release it. We can then search for the **Add Movement Input** node and it will automatically connect. This node does all the hard work of actually moving the player, with a few given inputs. Since we've already got the scale value, drag our *Axis Value* output into the *Scale Value* input.



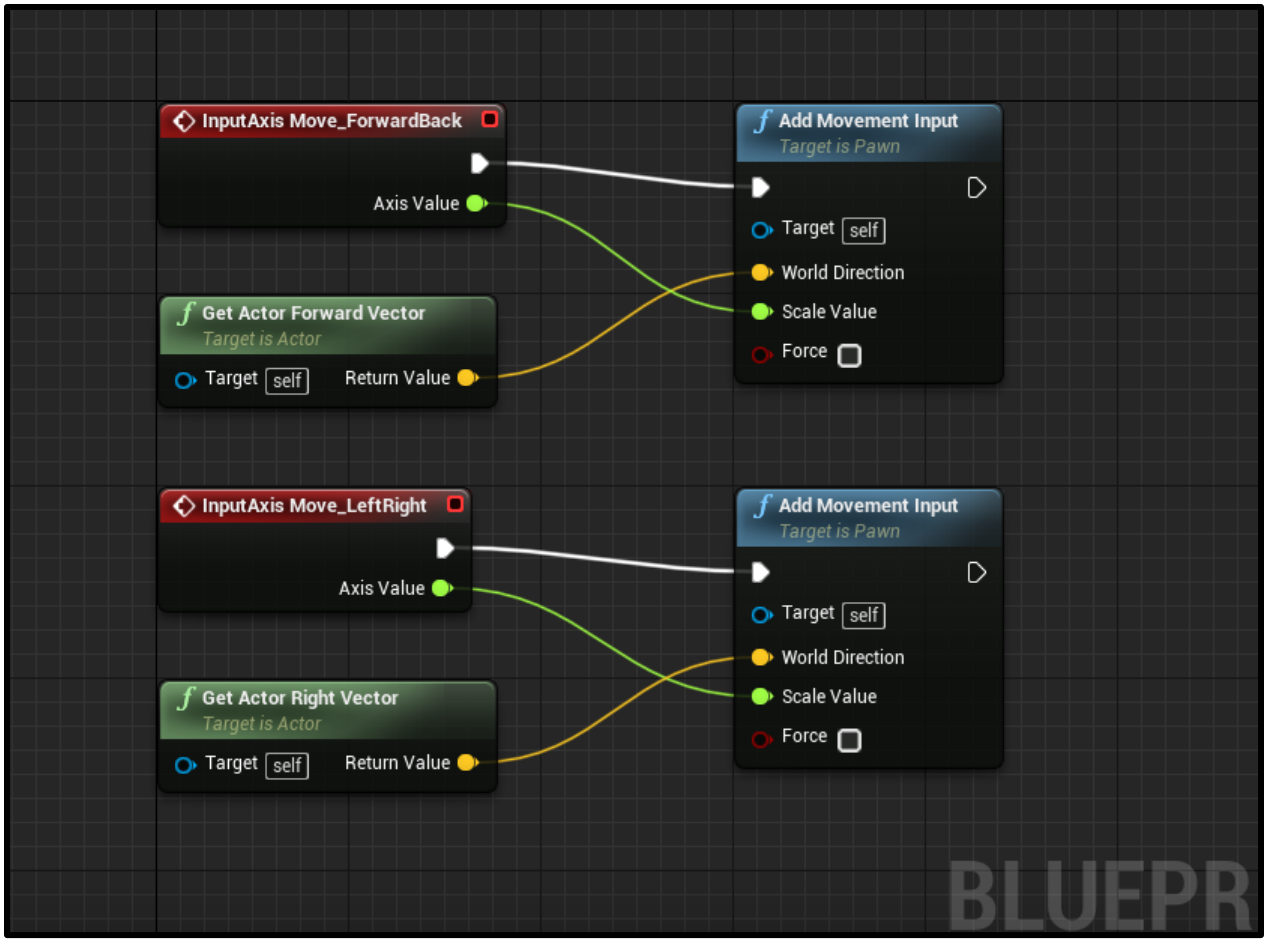
Finally, we just need to give it the world direction that it will move in. For this, right click and create a new node called **Get Actor Forward Vector** and plug that into the *World Direction* input.



At the top of the screen we can now press the **Play** button and test it out! You may notice the player is a bit fast. Select the *FloatingPawnMovement* component and set the **Max Speed** to *500*.

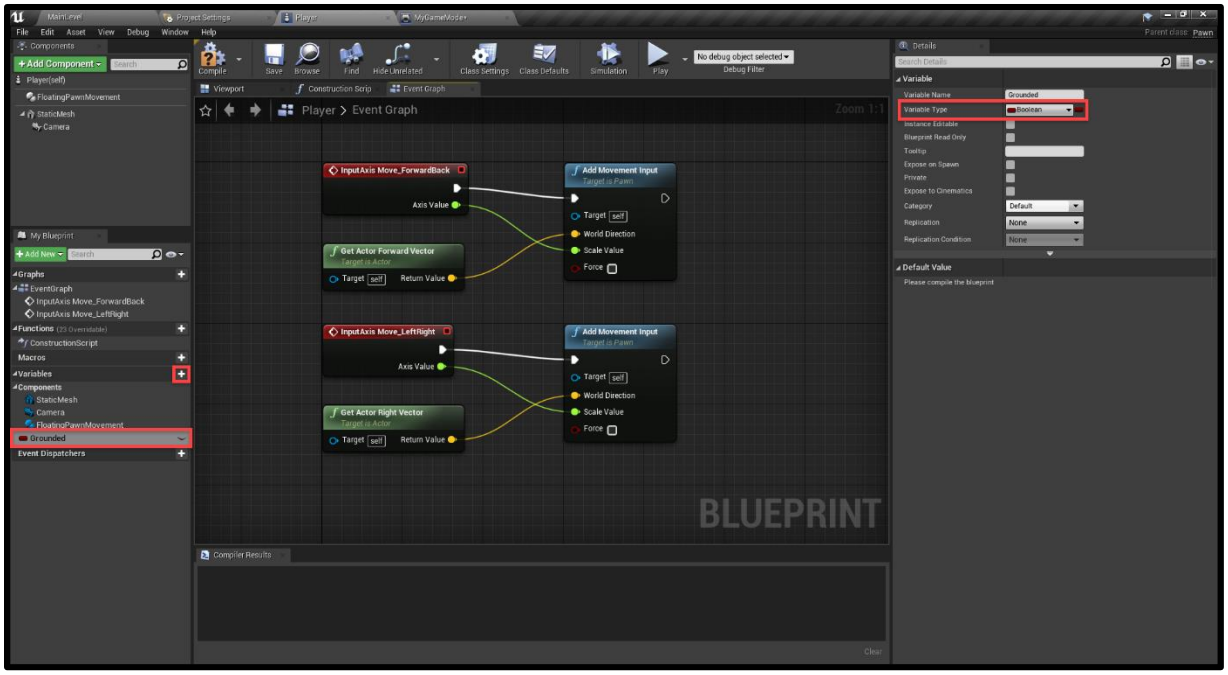


Now we can setup the horizontal movement. This time with the **InputAxis Move_LeftRight** node which is triggered with the *A* and *D* keys. For the world direction, we want to use the actor right direction.

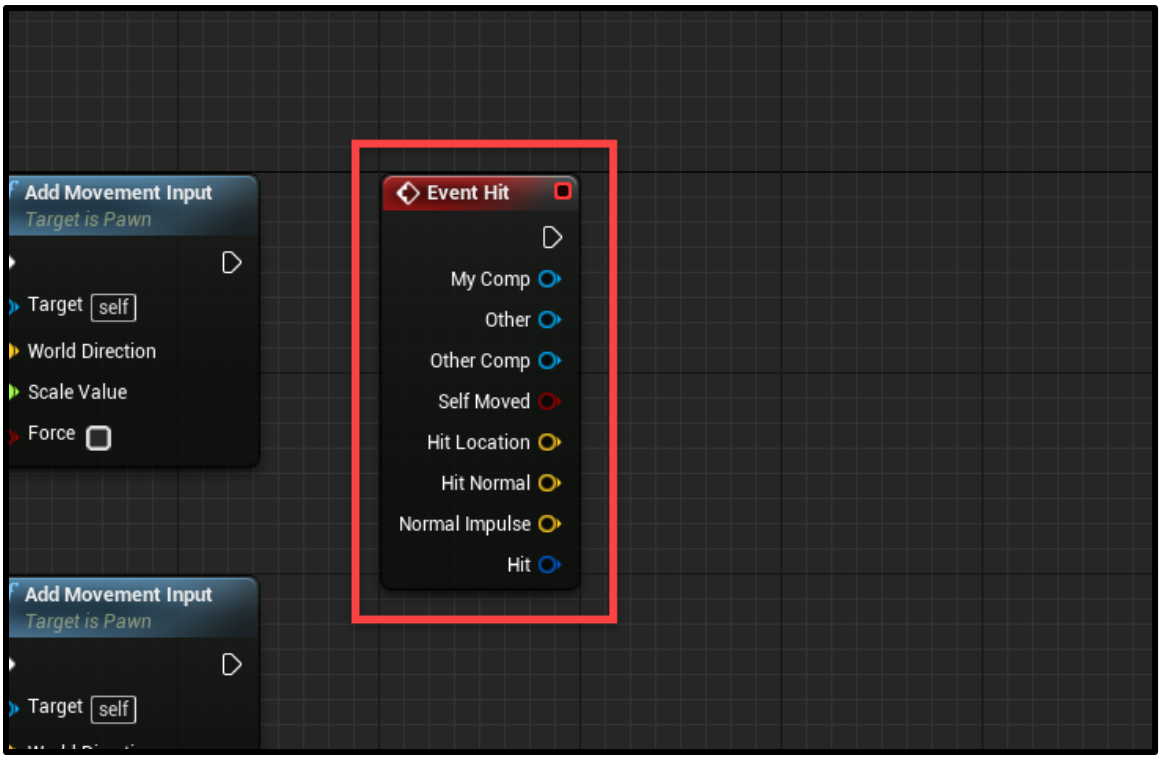


Now you should be able to play the game with the WASD keys.

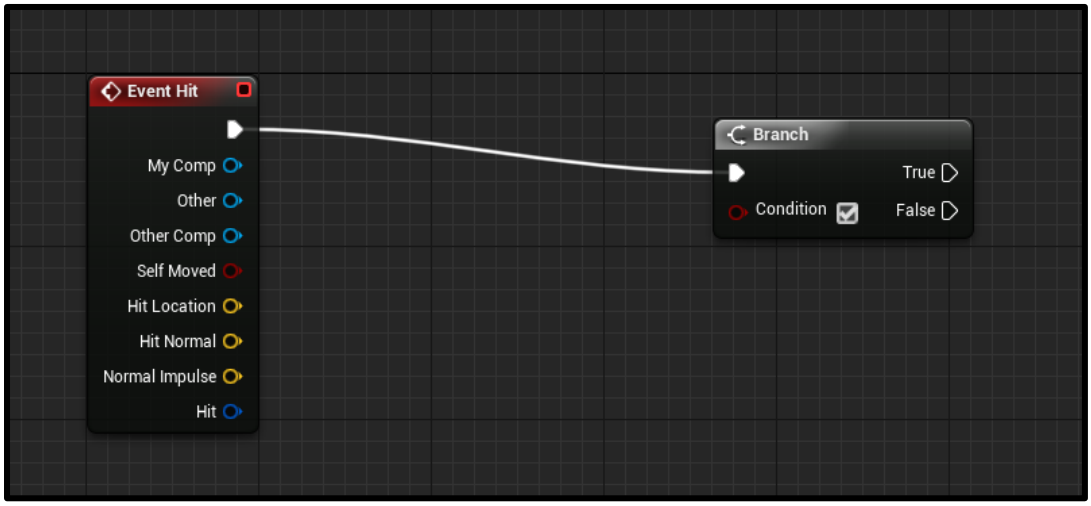
Next is jumping. For this though, we need to know when we're standing on the ground. Let's begin by creating a new variable. In the **My Blueprint** panel, create a new variable called **Grounded**. Over in the **Details** panel, set the **Variable Type** to *Boolean*. A boolean variable can be true or false.



In our graph, create a new node called **Event Hit**. This gets triggered whenever we enter the collider of another object (i.e. the ground). There are many outputs, but we're just going to use one.

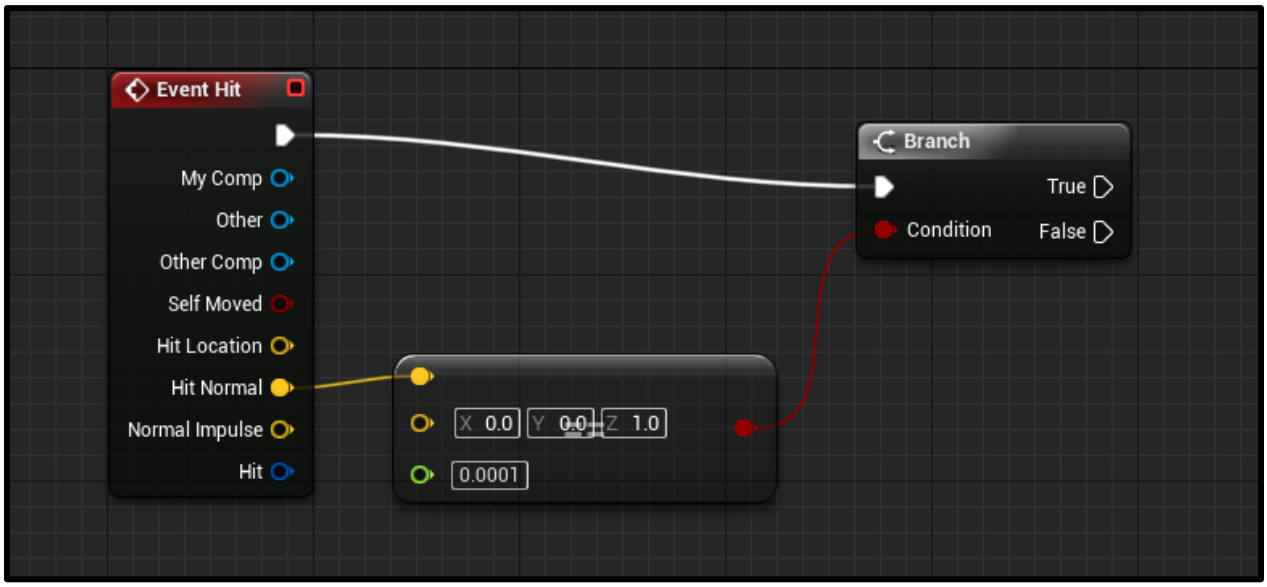


Connect the output flow to a new node called **Branch**. This is a very special node which allows us to split up the flow depending on a given condition. When we collide with another object, we don't know if it's above, down or to the side. So we only want to be grounded when we're colliding with an object below us.

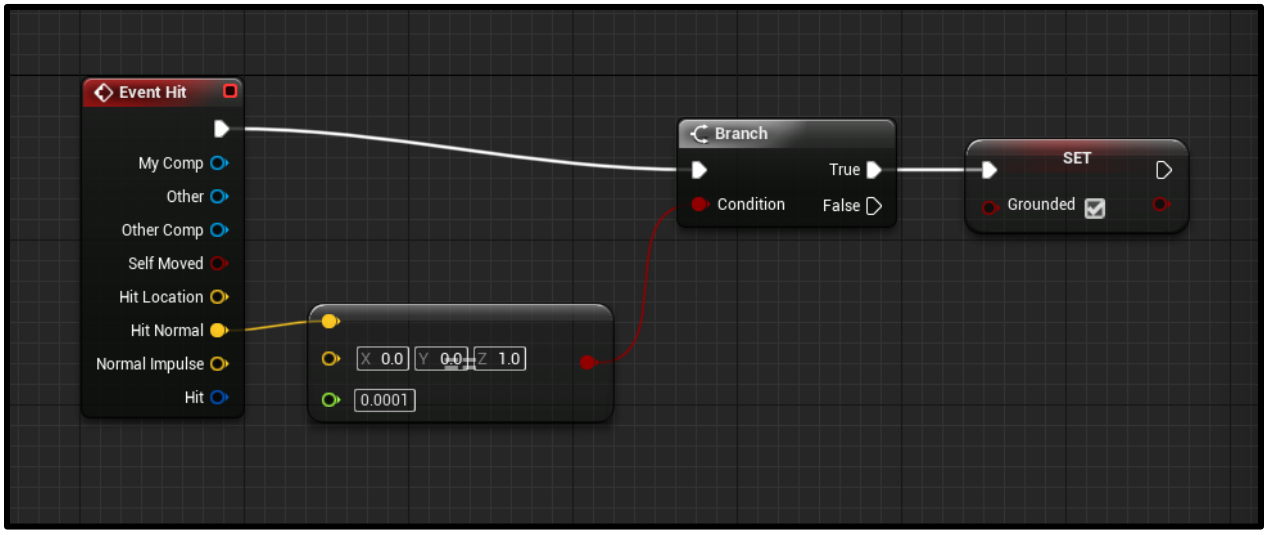


To do this, we're going to create a new node called **Equal (Vector)**. This node takes in two vectors (property containing a X, Y and Z) and compares them. If they're the same, it outputs true, otherwise false.

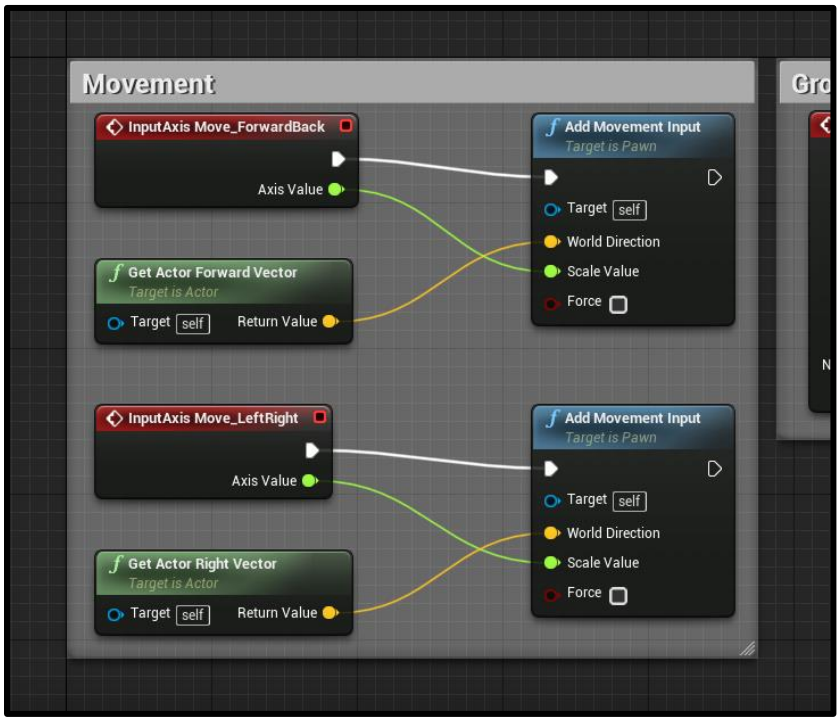
Connect the Event Hit's *Hit Normal* output to the node, and set the other input to *0, 0, 1*. What we're checking here is if the object we hit, has a collision normal (direction the collision point is facing) of up. If so, this means we're standing on something. Connect the node to the Branch.



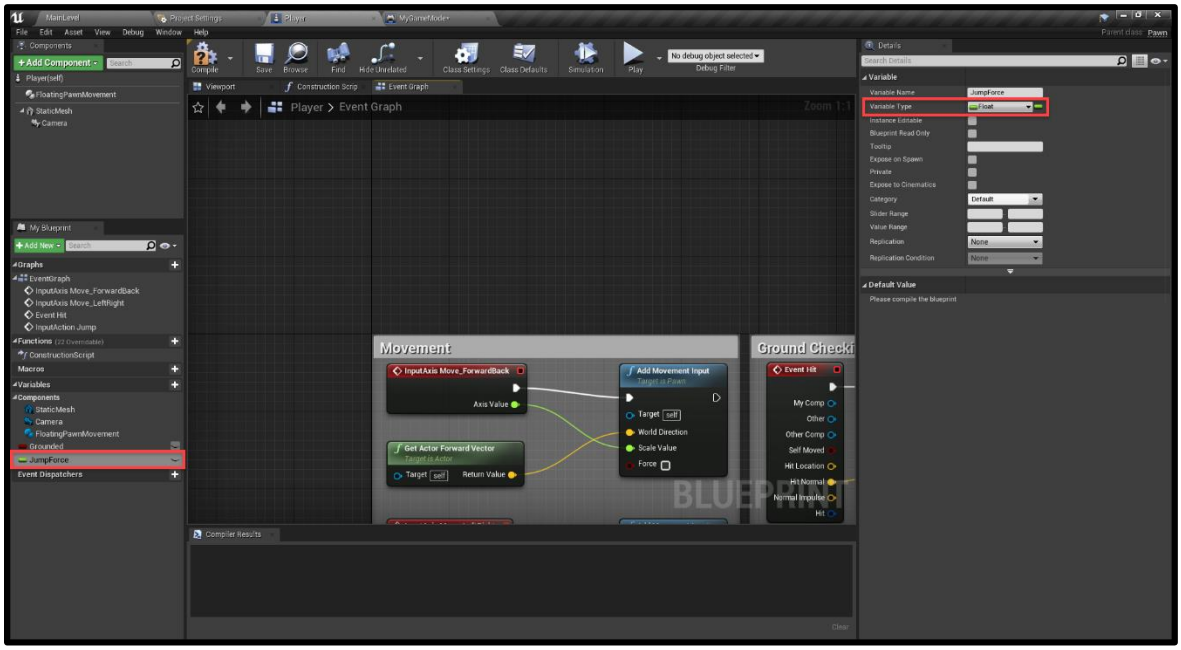
Next, on the branch node we have two flow outputs. One for if the condition is true and one for if it's false. Create a new node called **Set Grounded** and connect that to the branch's true output. Make sure to also tick the grounded input. This means that when we collide with an object, we'll check to see if we're standing on something. If so, set the *Grounded* variable to true.



You may soon notice that the graph can look quite convoluted. A way around this is by categorising different parts. So select all of the movement related nodes, right click and select **Create Comment From Selection**. You can then give it name.

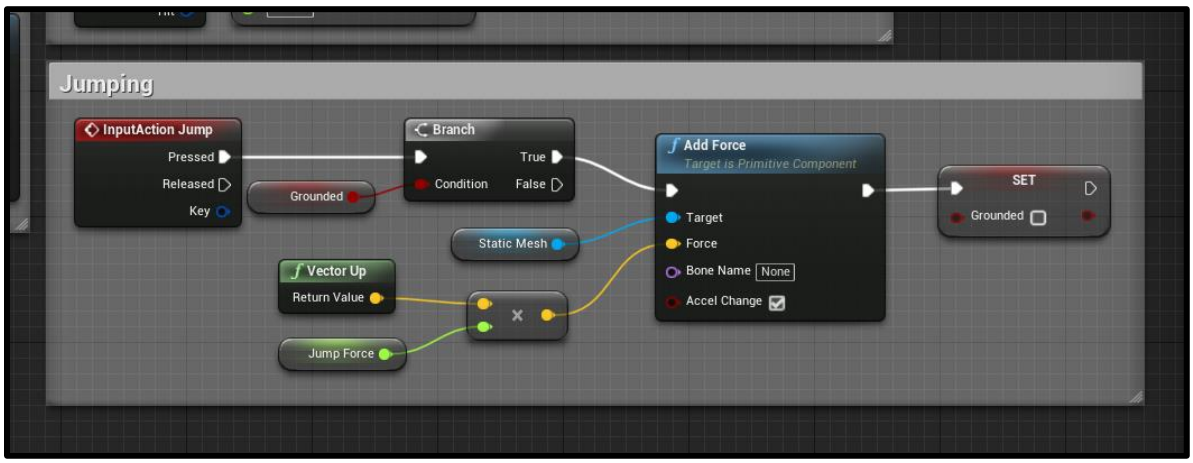


Now that we can detect the ground, let's implement jumping. To begin, we can create a new variable called **JumpForce**. Set the variable type to *Float*. This is a floating point number (i.e. decimal number like 5.2, 100.1, etc).

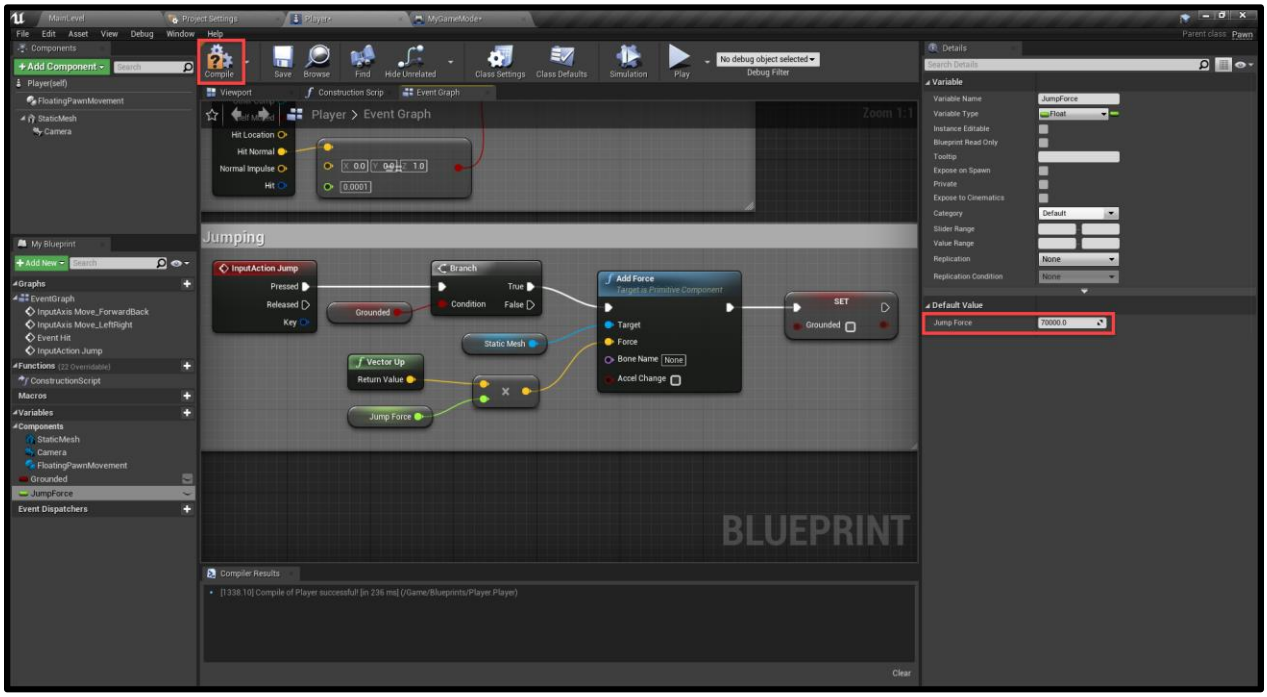


This is what the jumping will look like.

- The **InputAction Jump** node gets triggered when we press the *Space* button.
- This goes into a branch where we're checking the condition of the *Grounded* variable.
- If we are grounded, we're going to add force to the static mesh.
    - Enable *Accel Change*.
- The force, is going to be Vector up multiplied by the jump force.
- Then finally, we're setting *Grounded* to false.



Next, click the **Compile** button to compile the graph. This will allow us to set the default value for the jump force variable. Set it to *70,000*. We can then press play and test out the jumping.

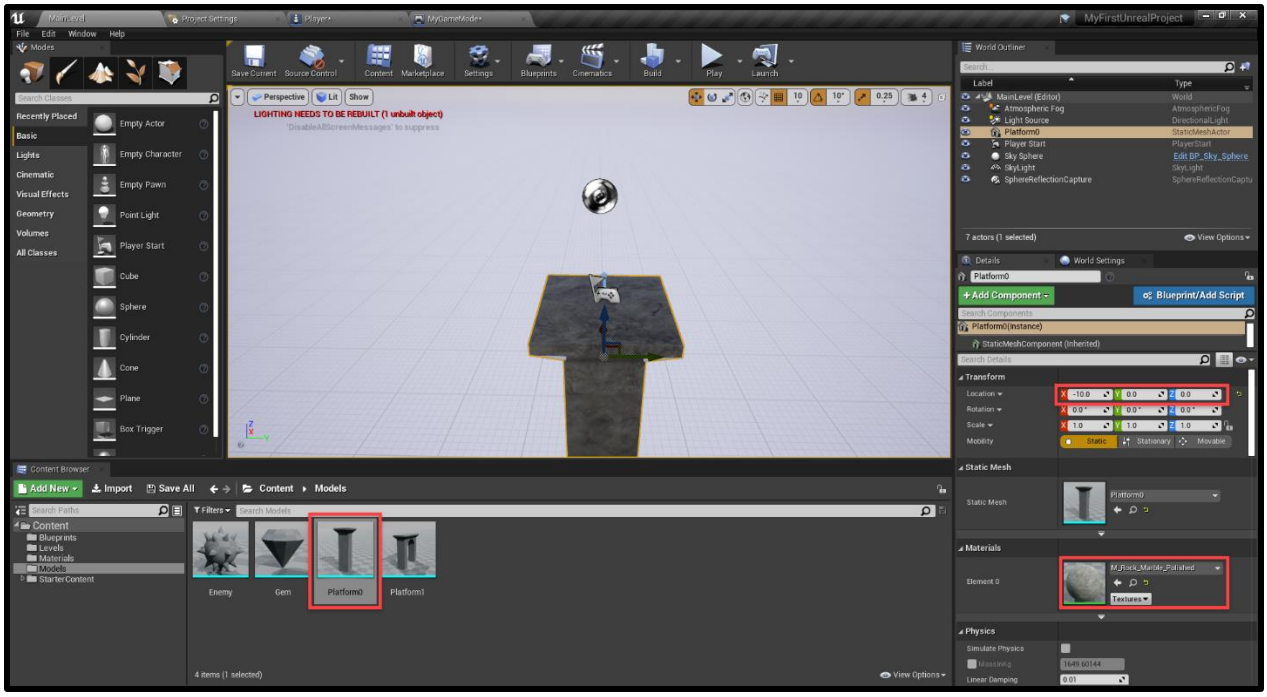


Building the Level

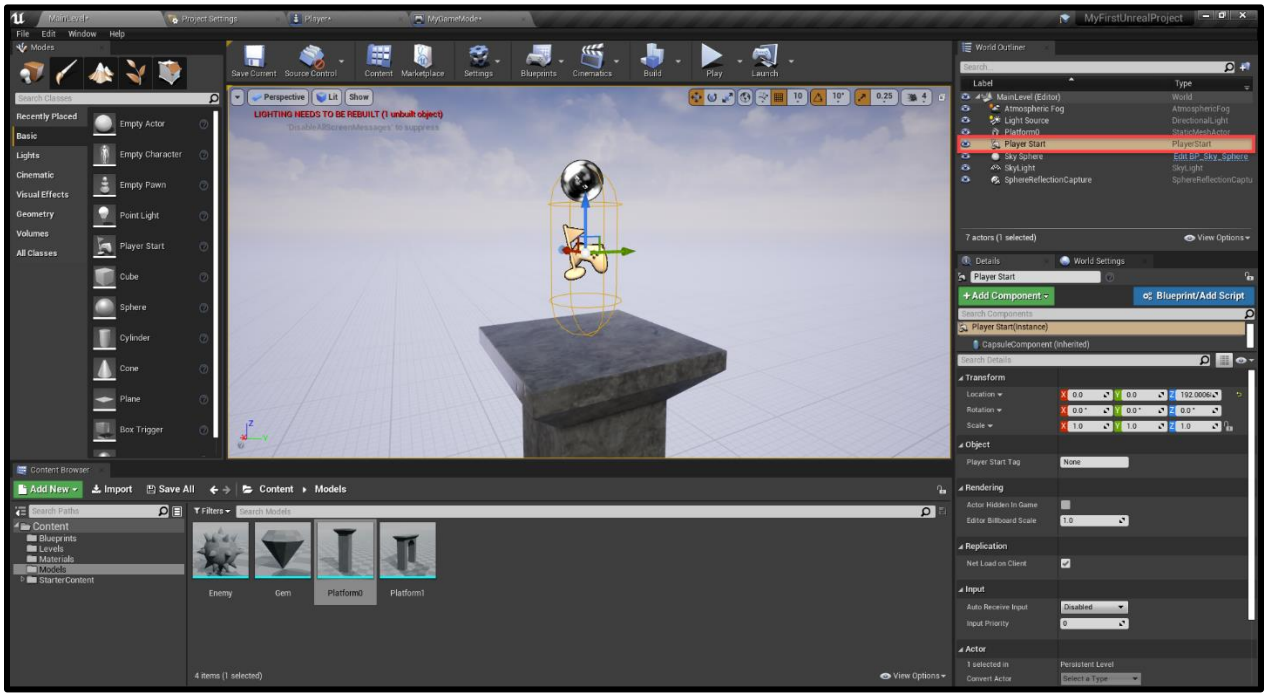
Back in the level editor, our game looks quite bland. Start by selecting then deleting the existing ground. In the **Models** folder, drag in the *Platform0* model.

- Set the **Location** to *-10, 0, 0*
- Set the **Material** to *M_Rock_Marble_Polished*

Quick Note: You can use the move gizmo (red, blue and green arrows) to move around a selected object. You can toggle between the rotate (press E), scale (press R) and move (press W) gizmos.



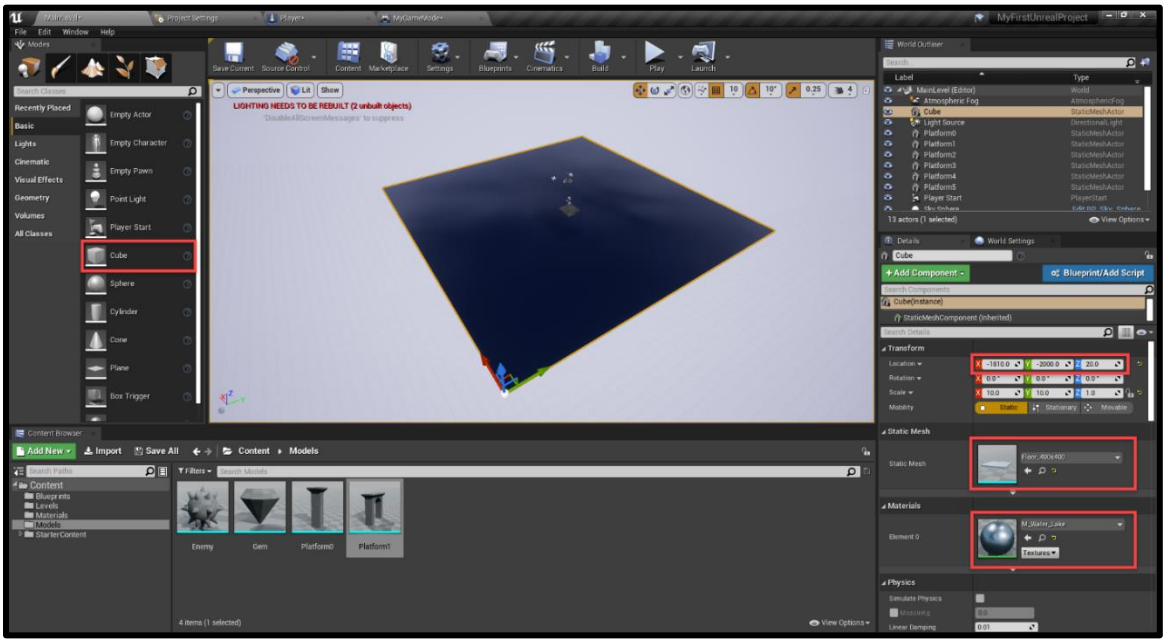
We also need to move the Player Start object up a bit so it's standing on the platform.



Next, we'll add in some water which will kill the player if they touch it.

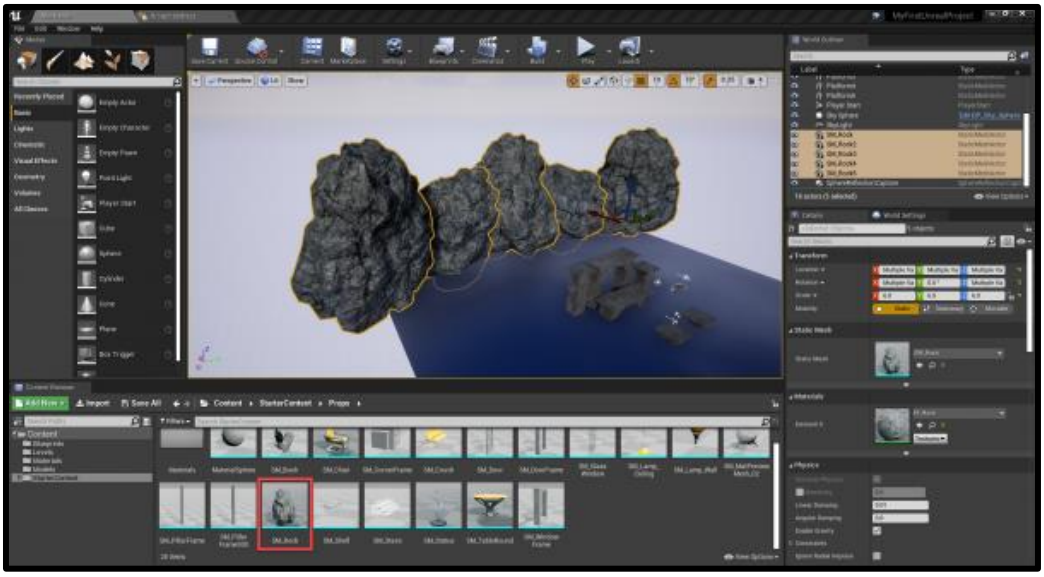
In the **Modes** panel, drag the cube object.

- Set the **Location** to *-1800, -2000, 20*
- Set the **Scale** to *10, 10, 1*
- Set the **Static Mesh** to *Floor_400x400*
- Set the **Material** to *M_Water_Lake*



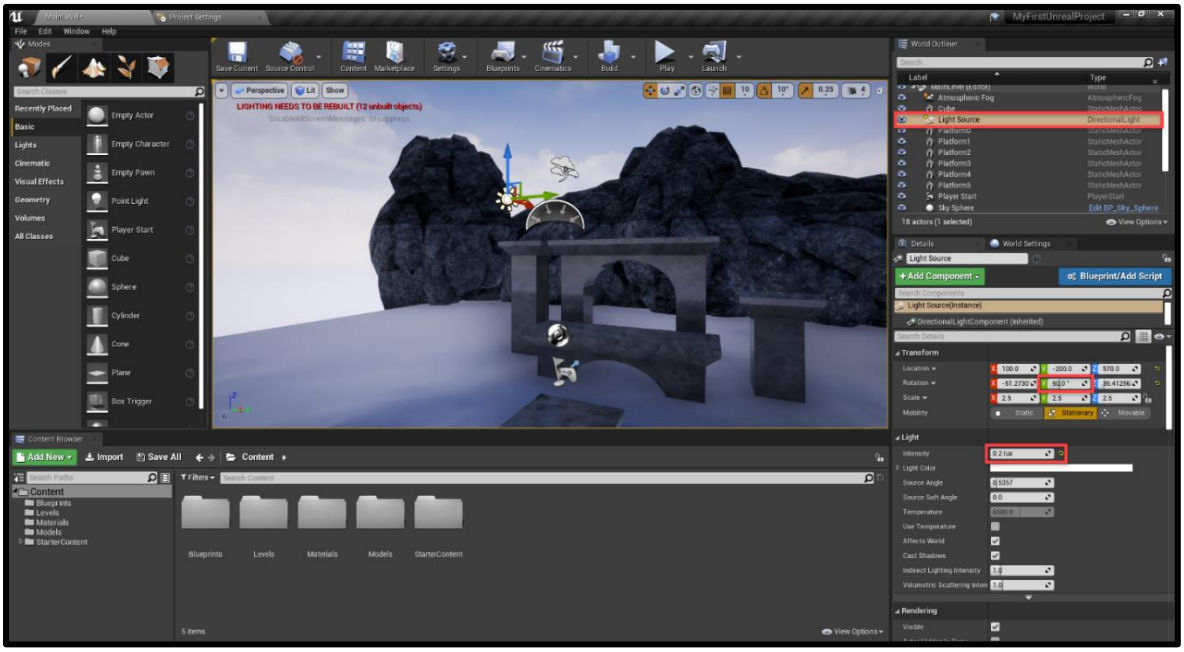
After this, we can go through and add in some more platforms.

Inside of the *StarterContent/Props* folder, we have a **SM_Rock** model. Drag that into the scene, position/scale/rotate it. We can then copy and paste it and move it around to look like the image below. This will be a good background for the level.



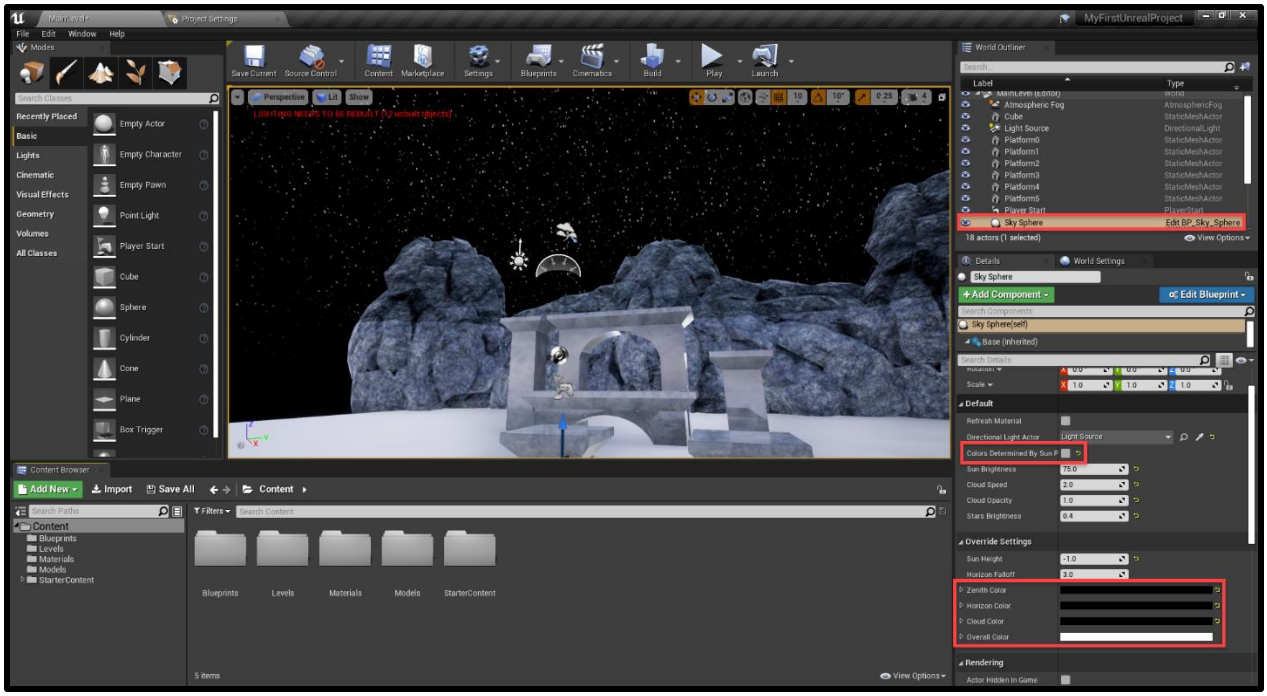
Now let's make it night time. Select the **Light Source**.

- Set the **Y Rotation** to *90*
- Set the **Intensity** to *0.2*

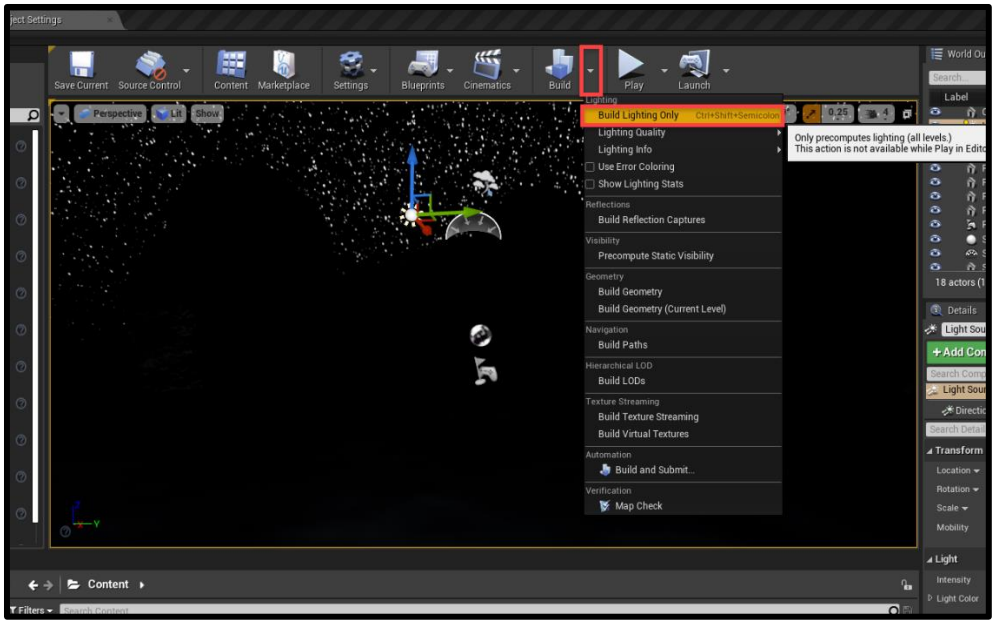


Select the **Sky Sphere**.

- Disable **Colors Determined By Sun Position**
- Set the **Zenith Color, Horizon Color** and **Cloud Color** to *Black*
- Set the **Overall Color** to *White*
- Set the **Stars Brightness** to *0.4*

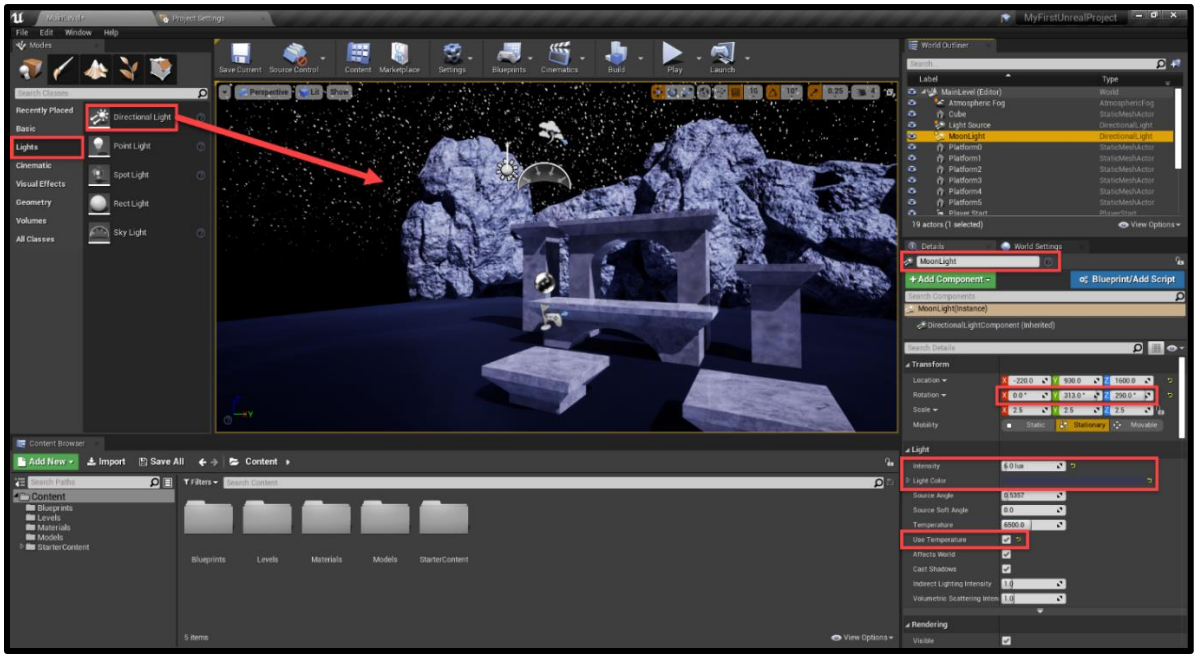


It's not dark though. To fix this, we can click on the arrow near the **Build** button and select **Build Lighting Only**. This will build our lighting, causing the scene to become dark.

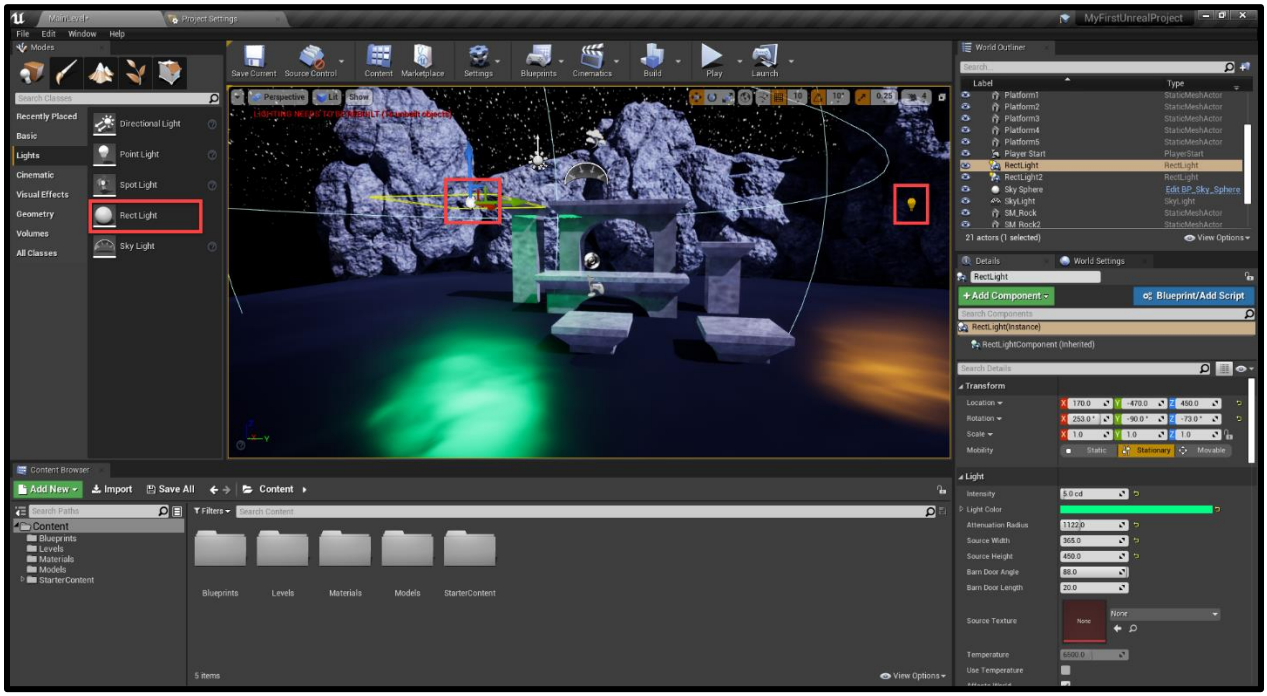


We don't want the scene to be totally dark though. In the **Modes** panel, go to the **Lights** tab and drag in a **Directional Light**.

- Set the **Name** to *MoonLight*
- Set the **Rotation** to *0, 313, 290*
- Set the **Intensity** to *6*
- Set the **Light Color** to a dark blue (Hex Decimal: 0A0B15FF)
- Enable **Use Temperature**

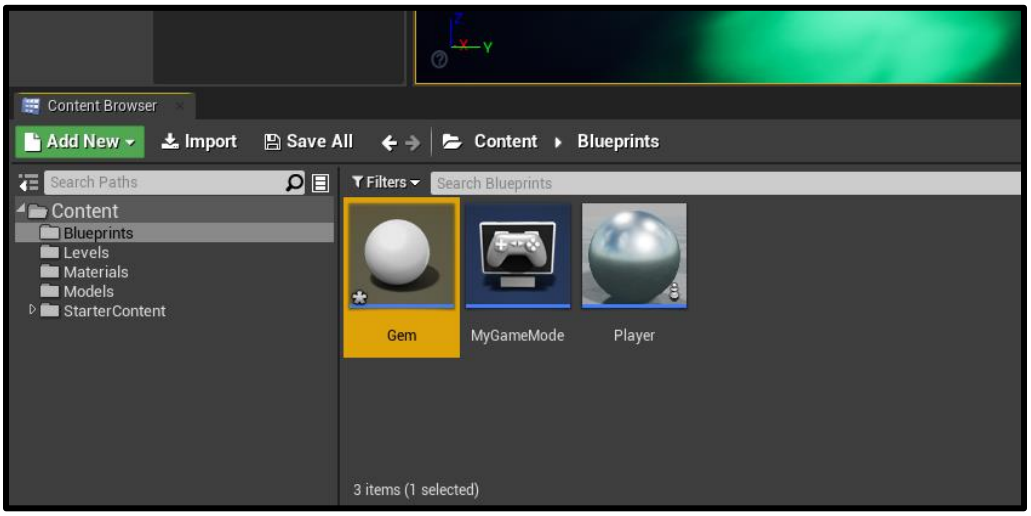


Optionally, we can also create some **Rect Lights** of different colors to add some color. But don't go overboard, as our enemies and gems will also be emitting light.



Collectable Gems

To guide the player through the level, we're going to have some gems for them to collect. In the **Blueprints** folder, create a new blueprint with a parent class of **Actor**. Call this blueprint: **Gem**.

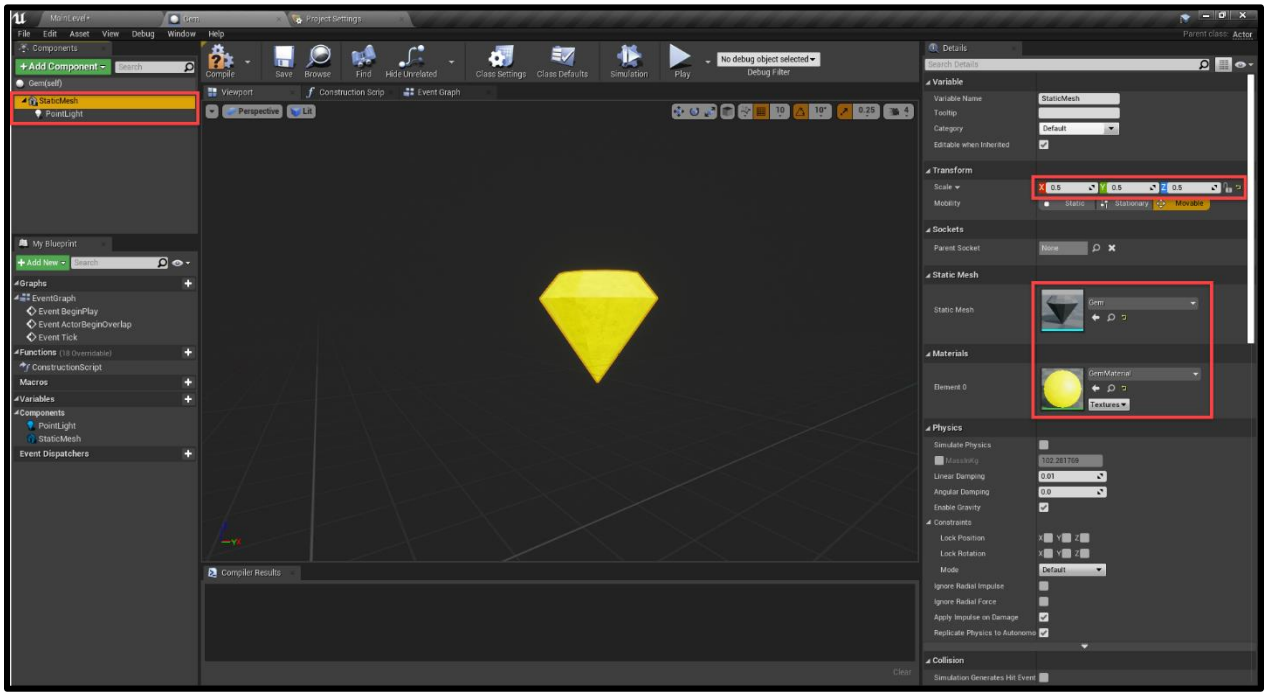


Double click on the blueprint to open up the editor. First, we need to add a **Static Mesh** component and drag it onto the scene root to make it the main component.

- Set the **Scale** to *0.5, 0.5, 0.5*
- Set the **Static Mesh** to *Gem*
- Set the **Material** to *GemMaterial*

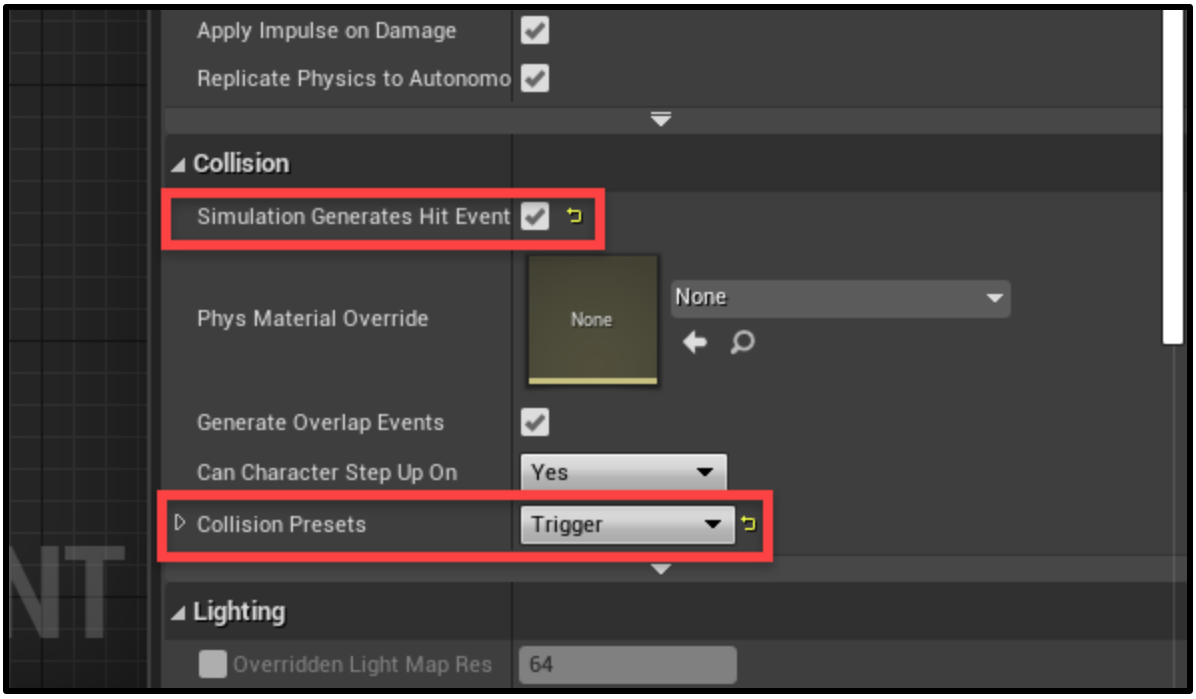
As a child of the mesh, add a new **Point Light** component.

- Set the **Light Color** to *Yellow*



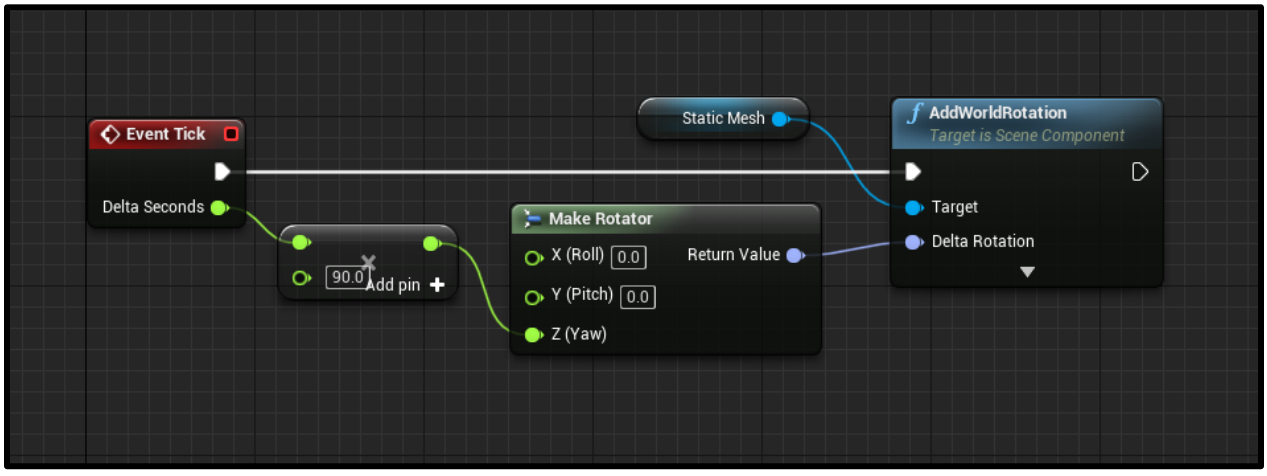
Selecting the static mesh, let's go to the **Details** panel.

- Enable **Simulation Generates Hit Events**
- Set the **Collision Presets** to *Trigger*
    - This means the gem can go through the player and detect a collision with them

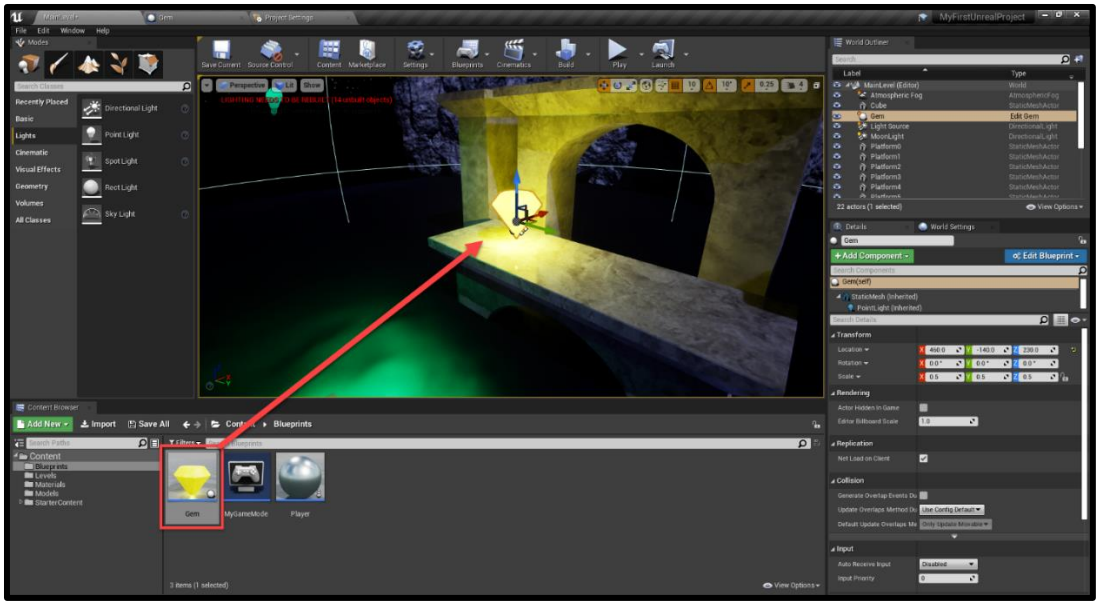


Now let's go over to the **Event Graph**. Here, we want to setup the ability for the gem to rotate over time.

- **Event Tick** gets called every frame
- It triggers the **AddWorldRotation** node which will add a rotation to our existing one
- The rotation is going to be along the Z (vertical) axis, 90 degrees a second

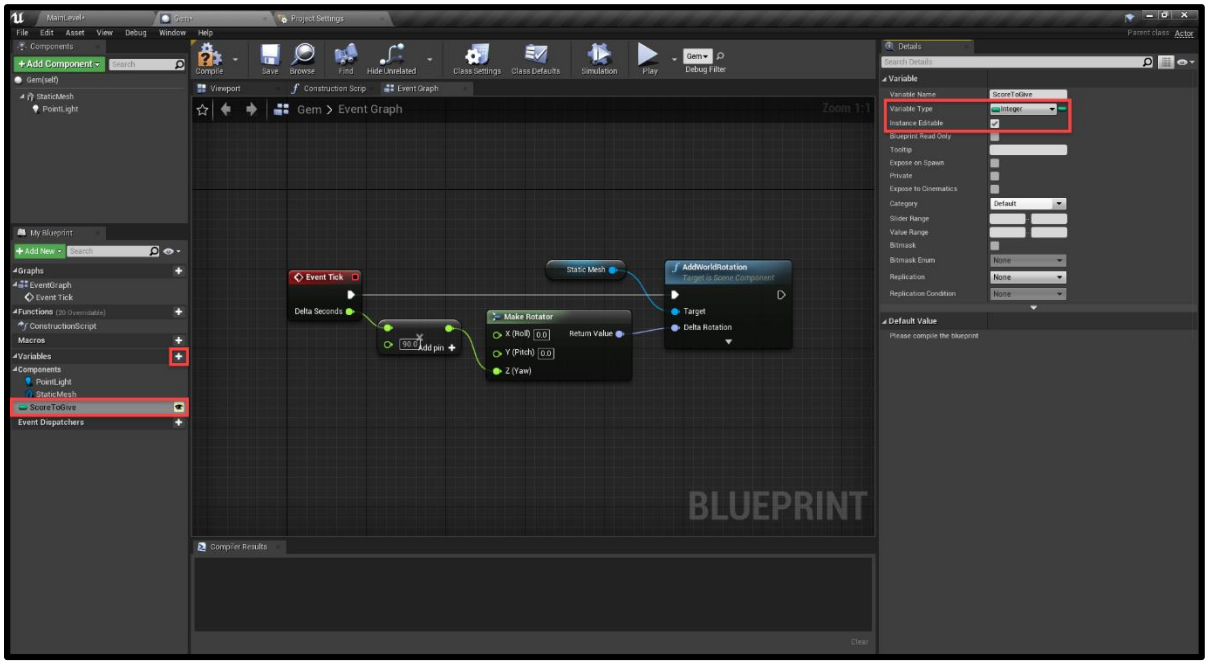


Save that, then go back to the level editor. Here, we can drag in the blueprint to create a new object. Place the gem on top of the platform like this.

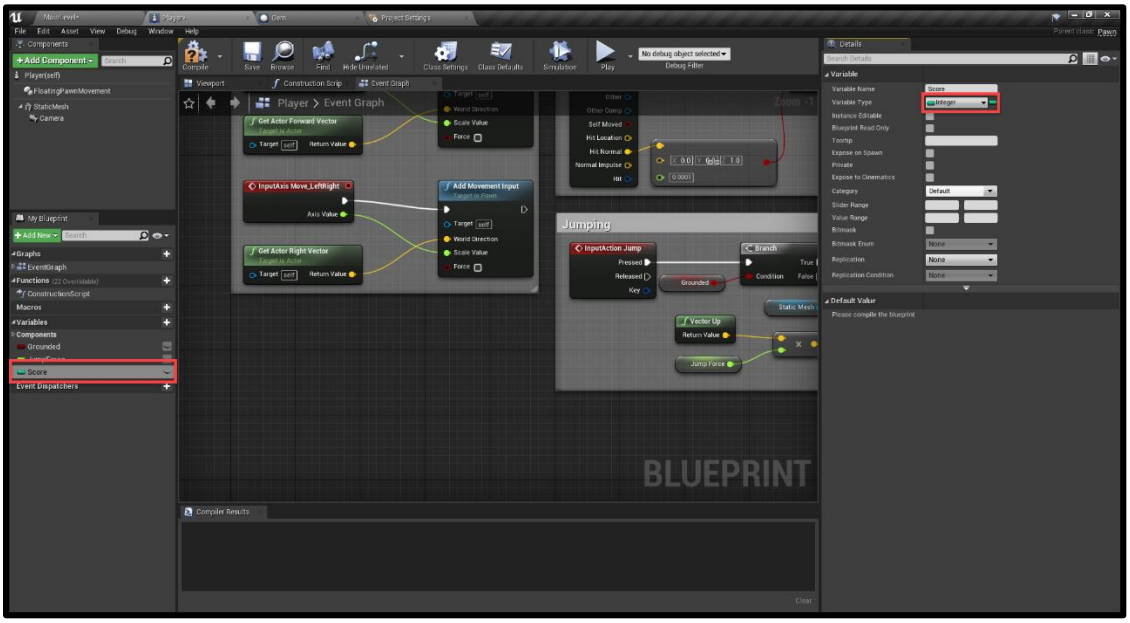


We can then press the **Play** button and see that it rotates!

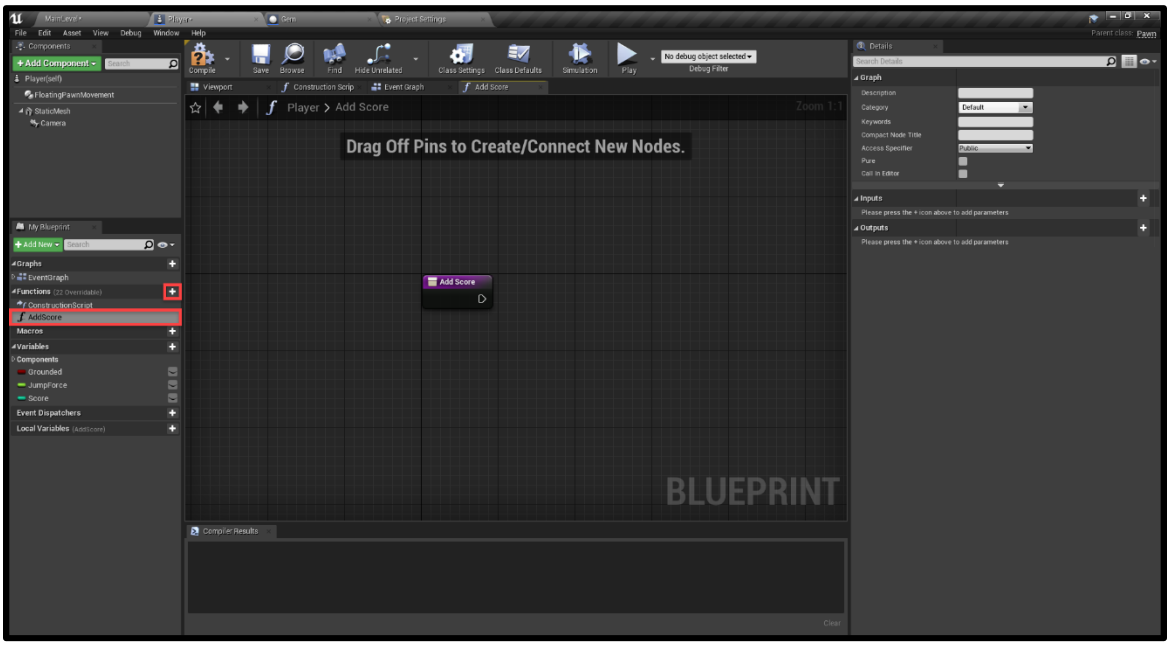
Now we need the gems to give the player some sort of score. Inside of the gem's blueprint graph, let's create a new variable called **ScoreToGive** of type *Integer* (whole number). Also enable **Instance Editable**. This means that when we create one of these gems in the level, we can set the score to give individually. Each gem can have a different score from the rest. Click the **Compile** button to make these changes appear in the level editor.



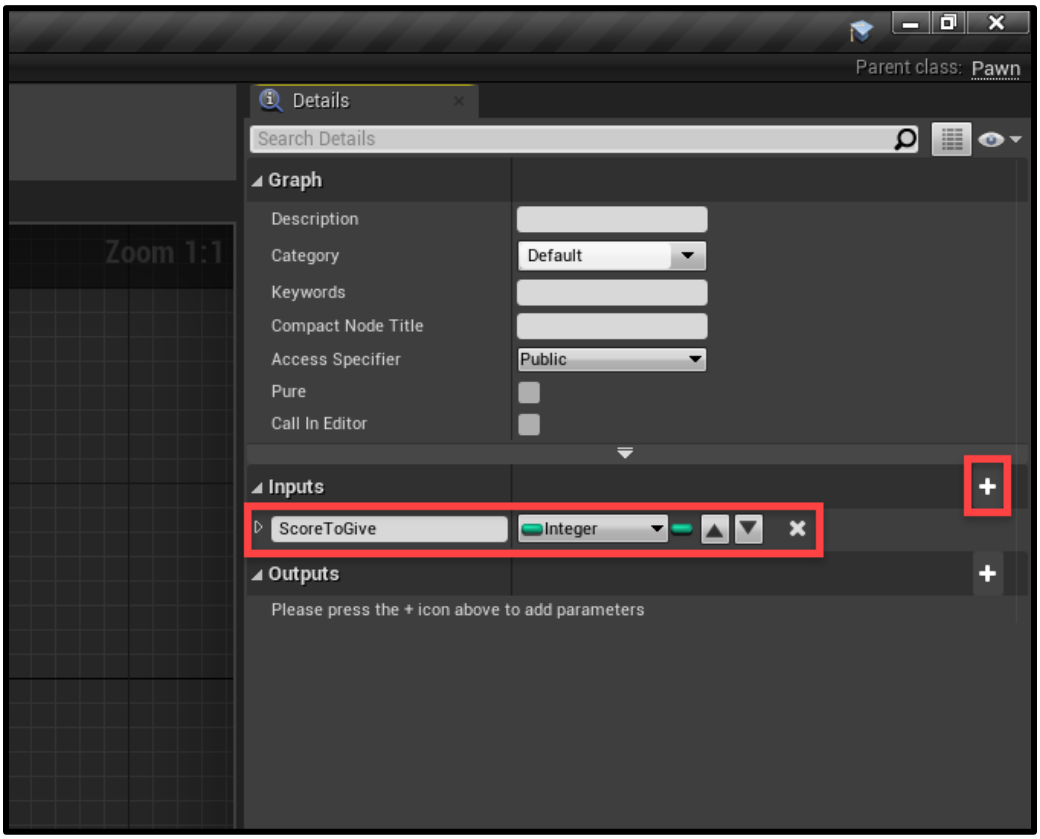
We don't have any score to add to yet. So let's go to the **Player** blueprint and create a new integer variable called **Score**.



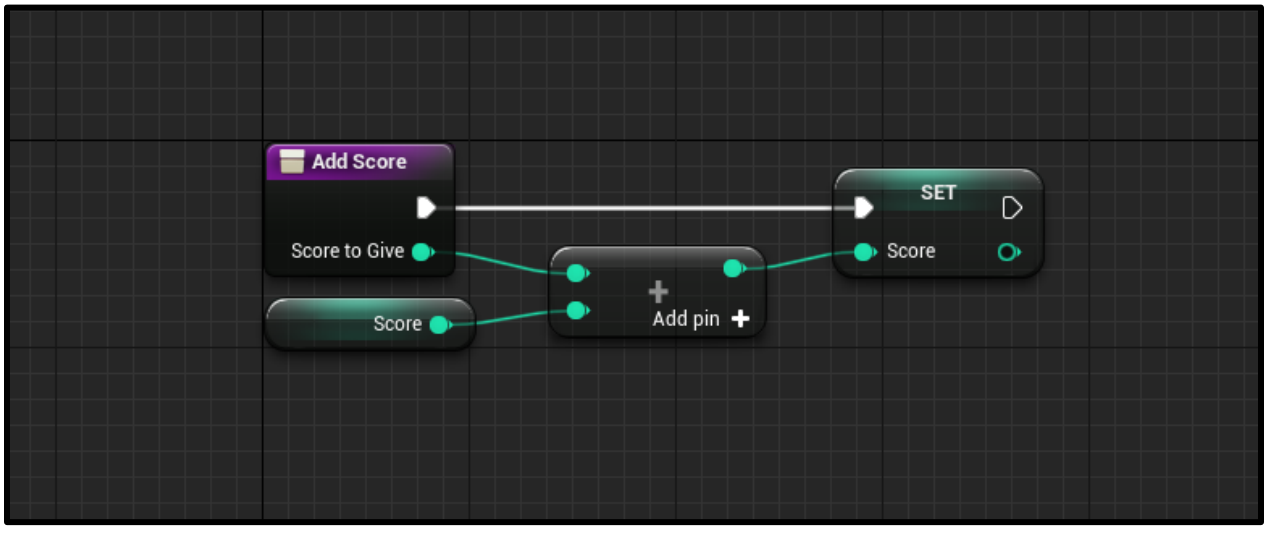
Next, let's create a new function (click on the **+** next to *Functions* in the **My Blueprint** panel) and call it **AddScore**. A function is basically a bunch of logic that can be triggered whenever we want. So from the gem script, we could call the **AddScore** function, and the logic in that blueprint will trigger.



Functions can have inputs and outputs. For add score, we want an input on the amount of score to give to the player.



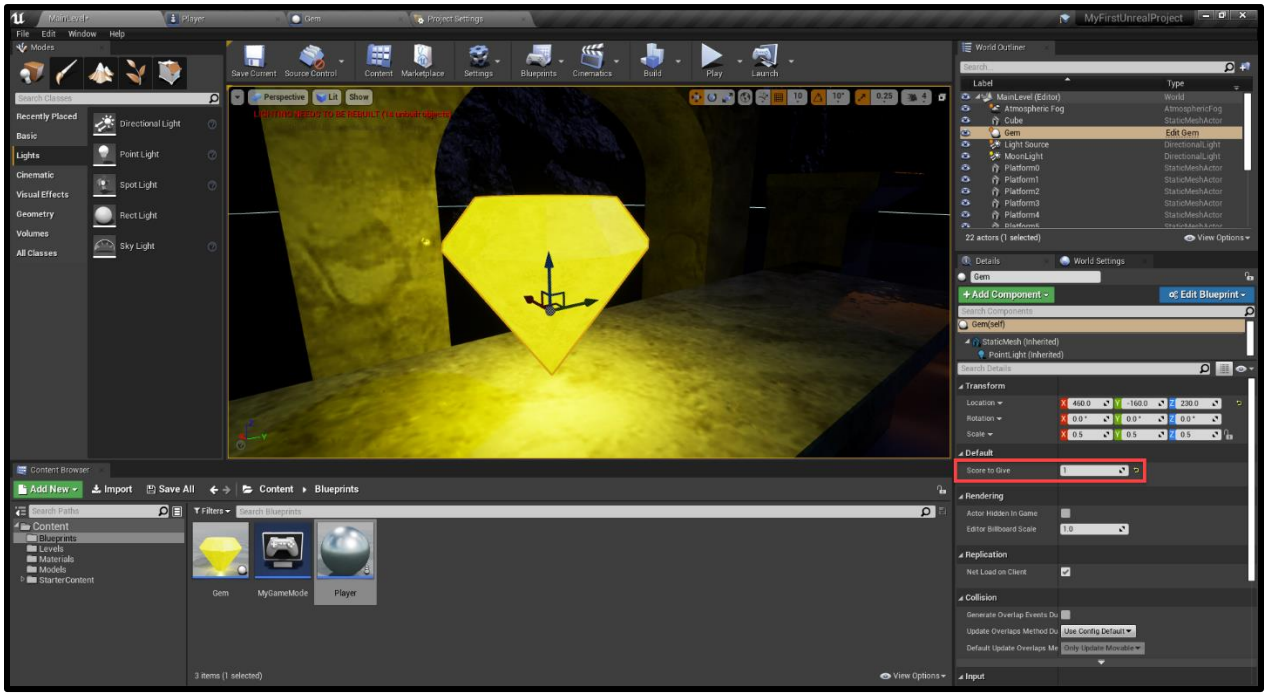
What we want to do in this function is add the *Score to Give* to our *Score*.



Back in the **Gem** blueprint, let's implement the ability to collect it.

- The **Event ActorBeginOverlap** node gets triggered when another object has entered our collider.
- We then take the object that collided with us and try to cast it to the *Player* class.
    - This basically converts the reference to a player blueprint so we can access the player specific things.
- If the cast was successful, we're going to call the **AddScore** function on the player.
- Then the **DestroyActor** node will destroy the gem.

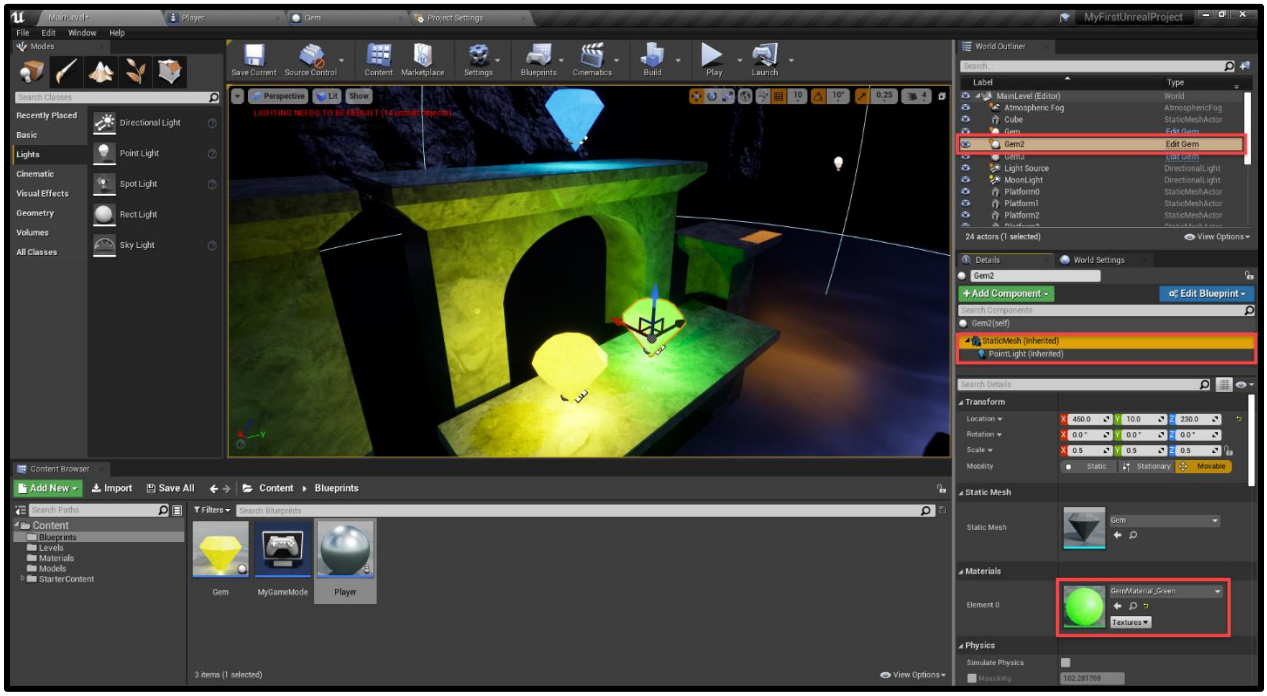
Let's now go back to the level editor. Select the gem and you should see a **Score to Give** property in the details panel. Set that to *1*.



Let's also create multiple gems of different values. Copy and paste a gem...

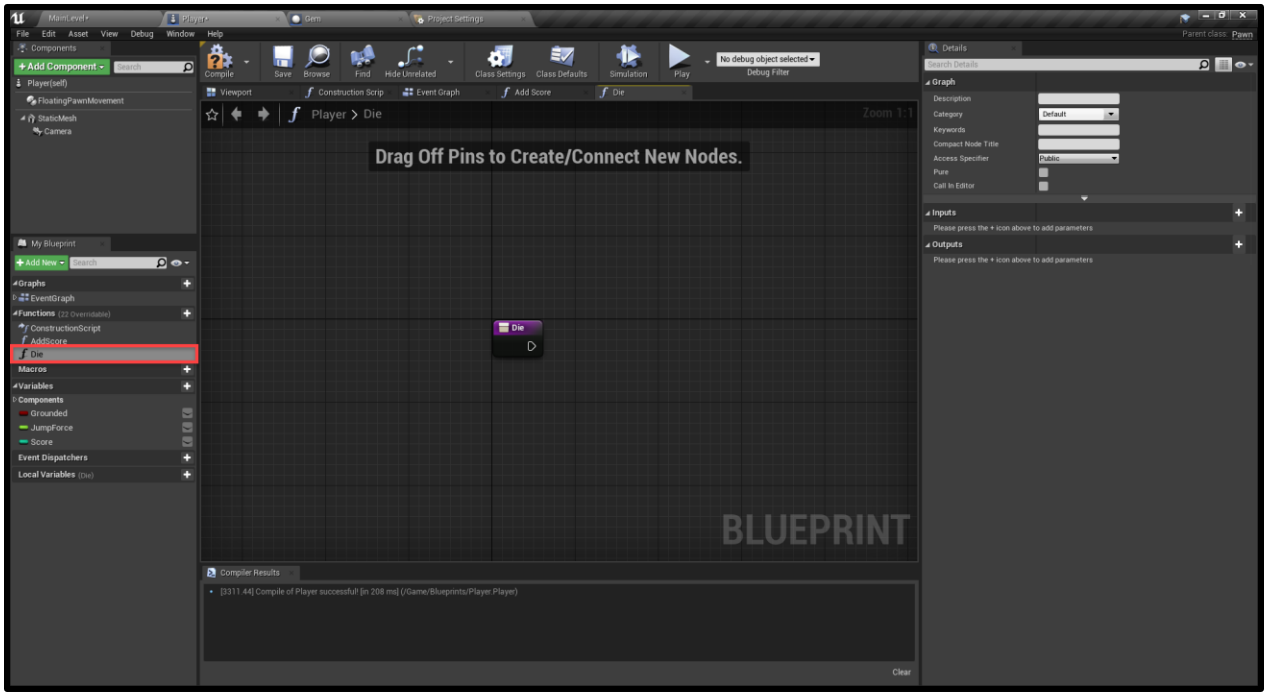
- In the **Details** panel, select the **Static Mesh** component
    - Set the **Material** to *GemMaterial_Green*
- In the **Details** panel, select the **Point Light** component
    - Set the **Light Color** to *green*

We can also change the score to give value.



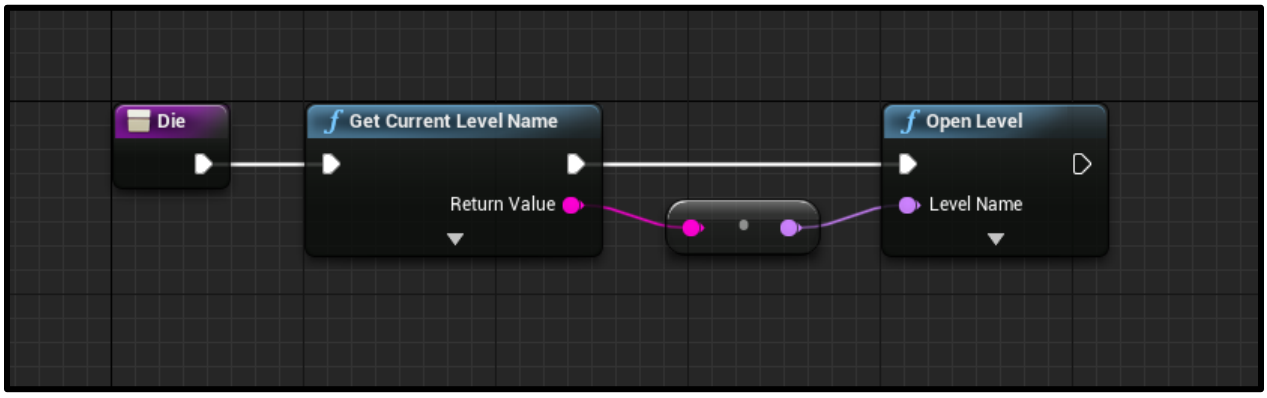
Resetting the Player

Right now when the player hits the water, nothing happens. What we want to do, is reset the level when the player "dies". So in the **Player** blueprint, create a new function called **Die**.



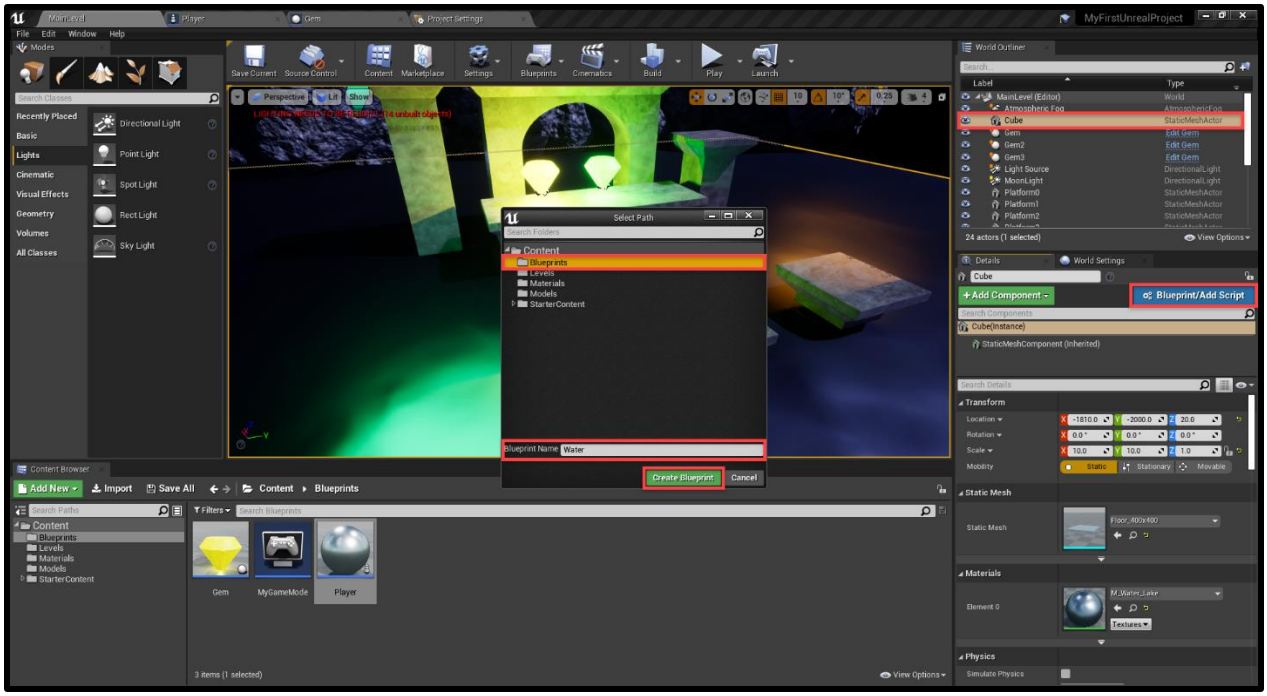
This function will reset the current level.

- We first get the name of the current level.
- Then we trigger the **Open Level** node, inputting the level name.



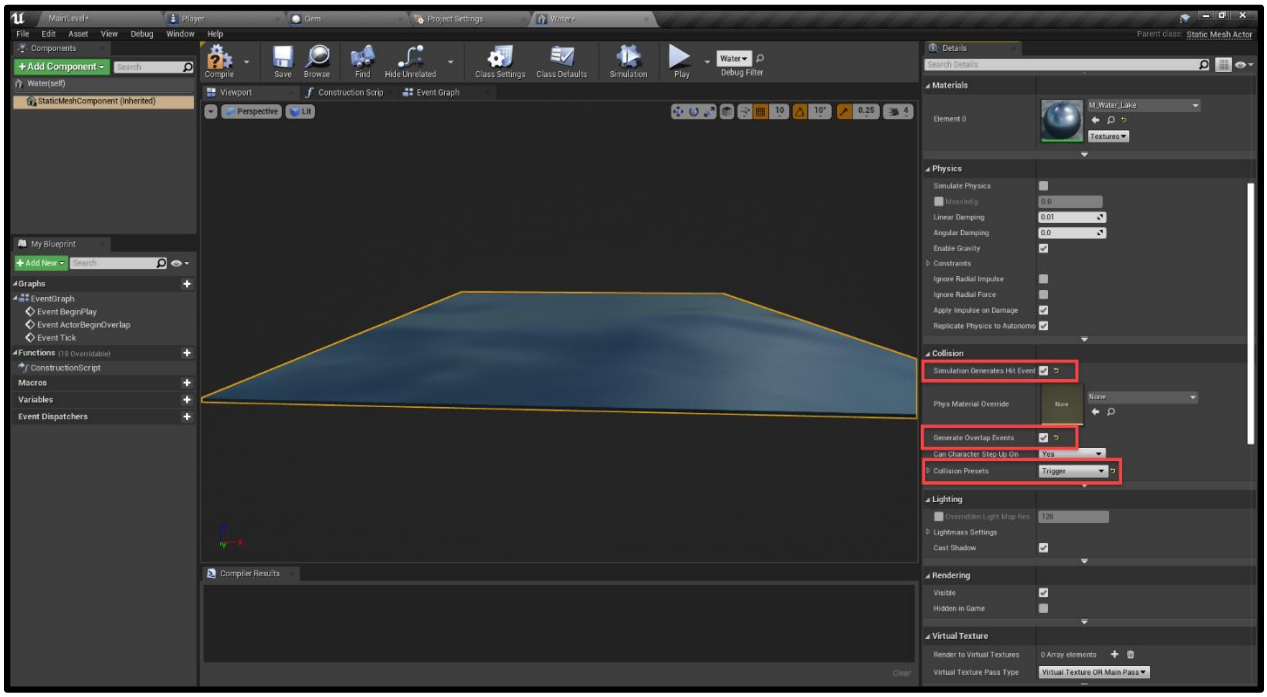
Back in the level editor, let's call this function when the player hits the water.

1. Select the water object
2. In the **Details** panel, click **Blueprint/Add Script**
3. A window will popup, select the *Blueprints* folder, rename the blueprint and click **Create Blueprint**



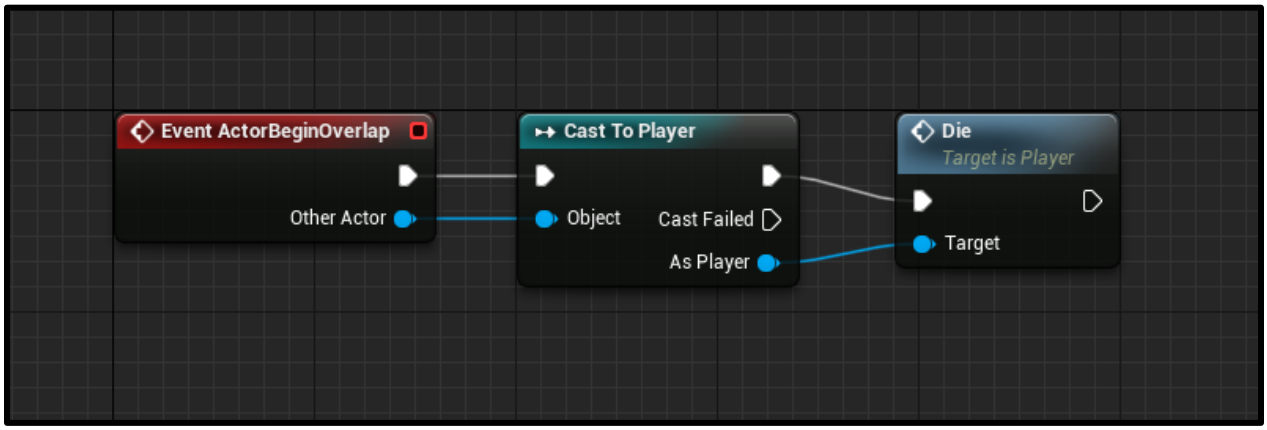
This will open up a new blueprint editor. Select the static mesh component.

- Enable **Simulation Generates Hit Events**
- Enable **Generate Overlap Events**
- Set the **Collision Preset** to *Trigger*



Click on the **Event Graph** tab as we're going to setup a few nodes.

- **Event ActorBeginOverlap** gets triggered when an object enters the water collider
- We're then going to cast the colliding object to the *Player* class
- If it was the player, then call the **Die** function



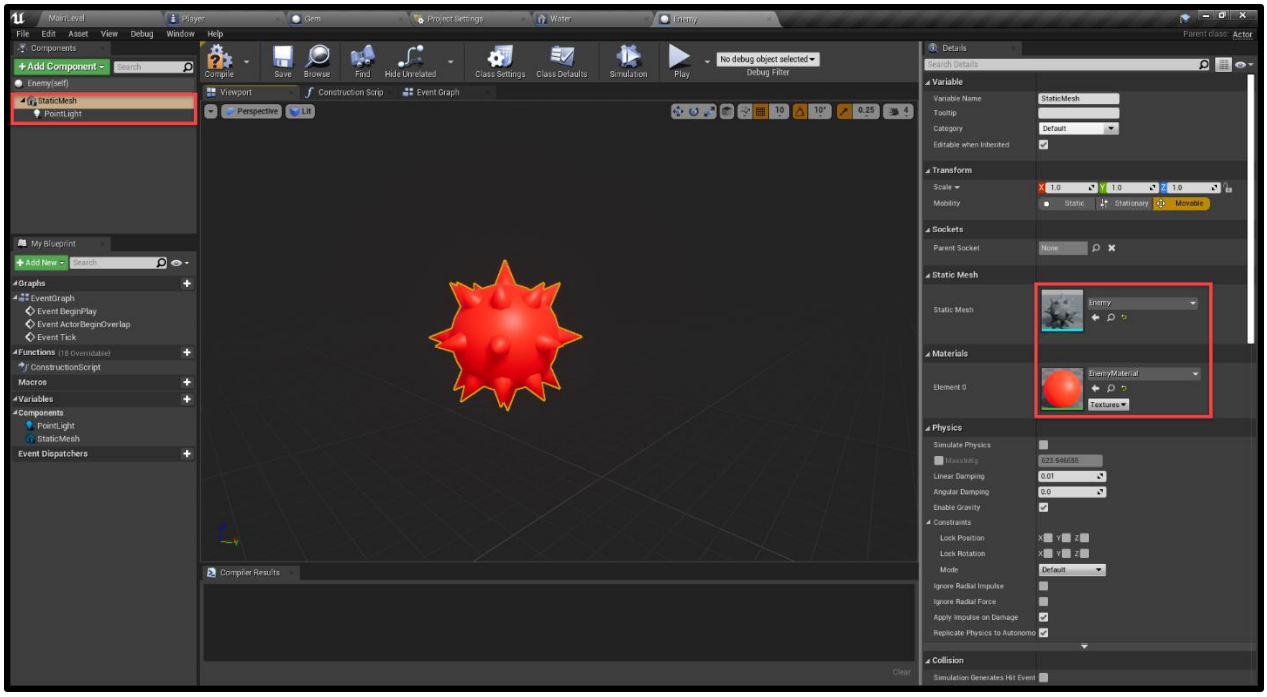
Back in the level editor we can press play and test it out!

Enemy Blueprint

Now it's time to create the enemy blueprint. This enemy will move back and forth between two positions. If the enemy hits the player, the Die function will be called. So to begin, let's create a new actor blueprint called **Enemy**.

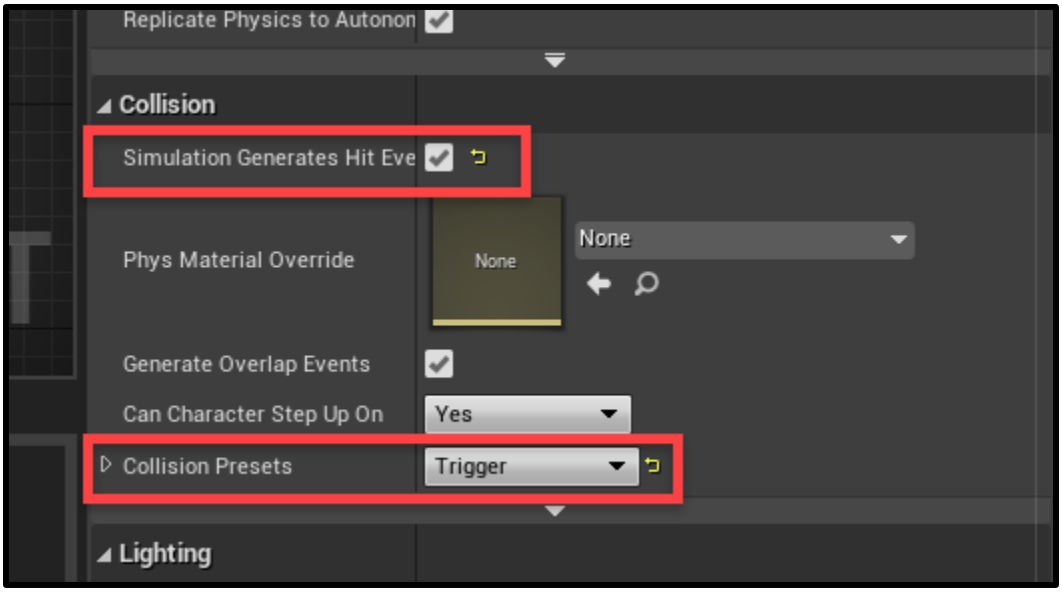
1. Create a new **Static Mesh** component and make that the parent component.
    a. Set the **Static Mesh** to *Enemy*
    b. Set the **Material** to *EnemyMaterial*
2. Create a new **Point Light** component and make it the child of static mesh
    a. Set the **Color** to *red*
    b. Set the **Intensity** to *3000*

The component layout is very similar to the gem.



Select the **Static Mesh** component.

- Enable **Simulation Generates Hit Events**
- Set the **Collision Presets** to *Trigger*

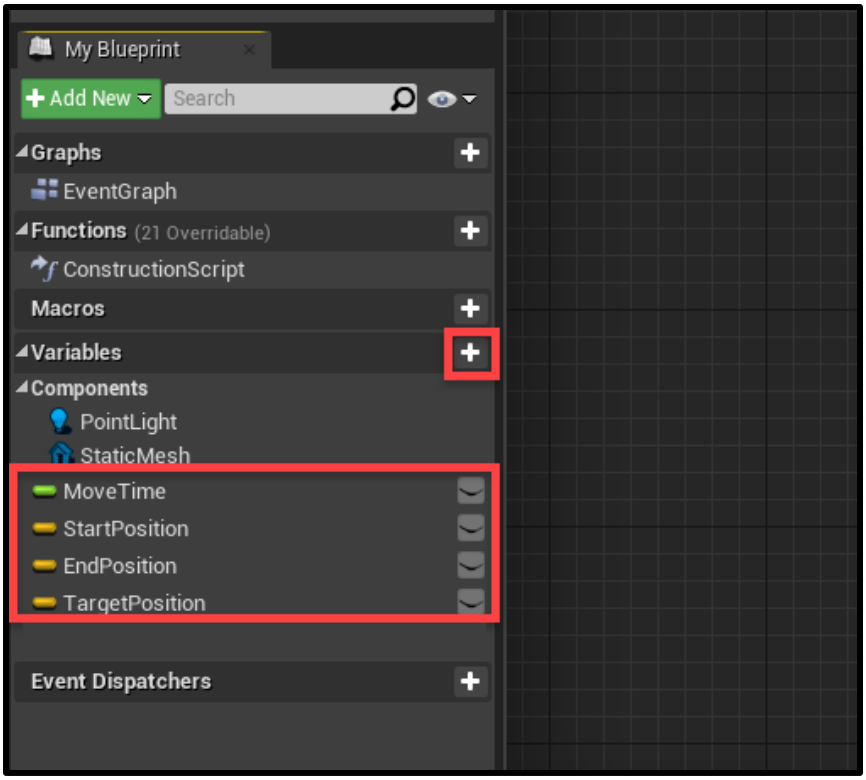


Next, we can head over to the **Event Graph** and begin to create the logic. We'll start with the variables.

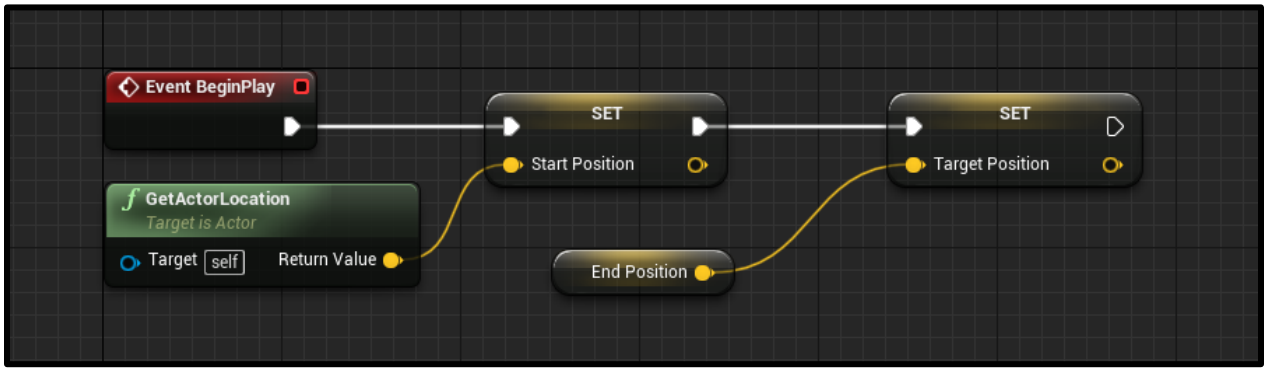
- **MoveTime** (Float) the time it takes to move to a point.
- **StartPosition** (Vector) starting position of the enemy.
- **EndPosition** (Vector) position to move to.
- **TargetPosition** (Vector) position the enemy is currently moving to.

Click the **Compile** button so we can set some variable properties in the **Details** panel.

- **MoveTime** - enable *Instance Editable*
- **EndPosition** - enable *Instance Editable*

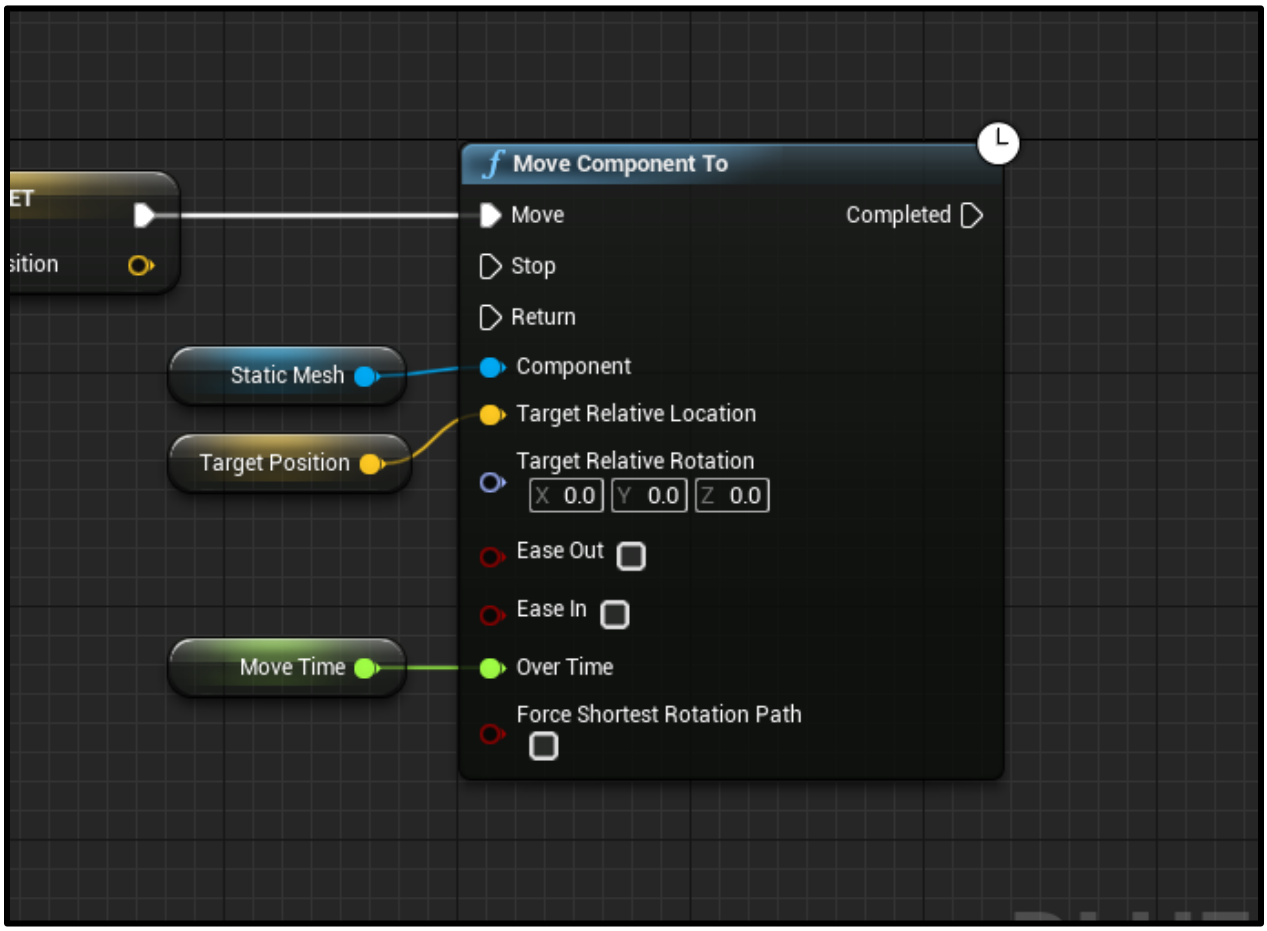


To begin, we'll set the initial values for the **Start Position** and **Target Position**.

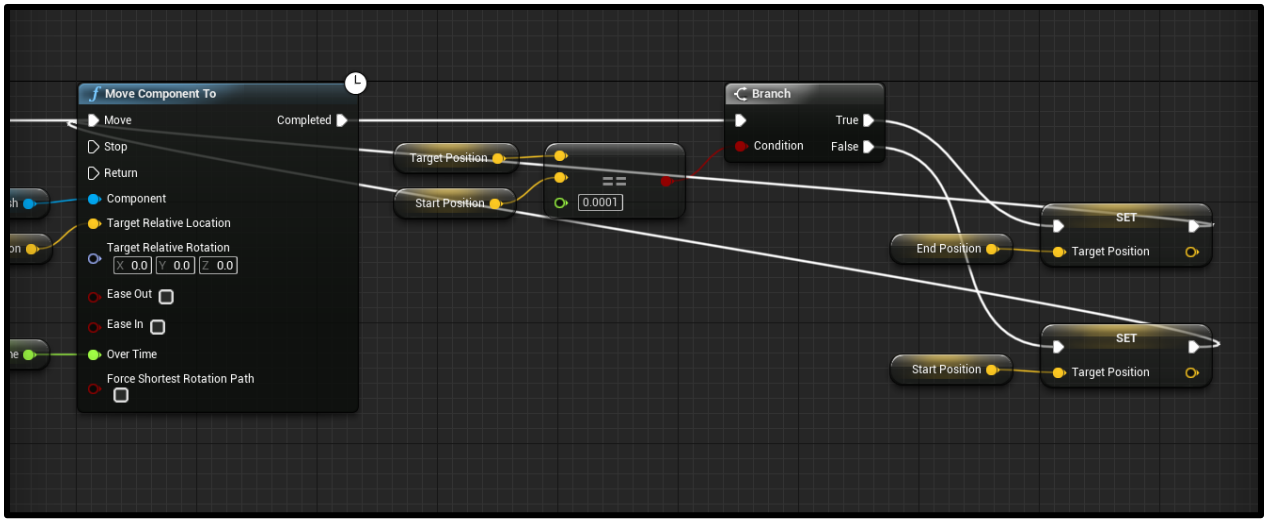


Continue the flow into a **Move Component To** node. This node takes in a few inputs and moves the object towards a certain location.

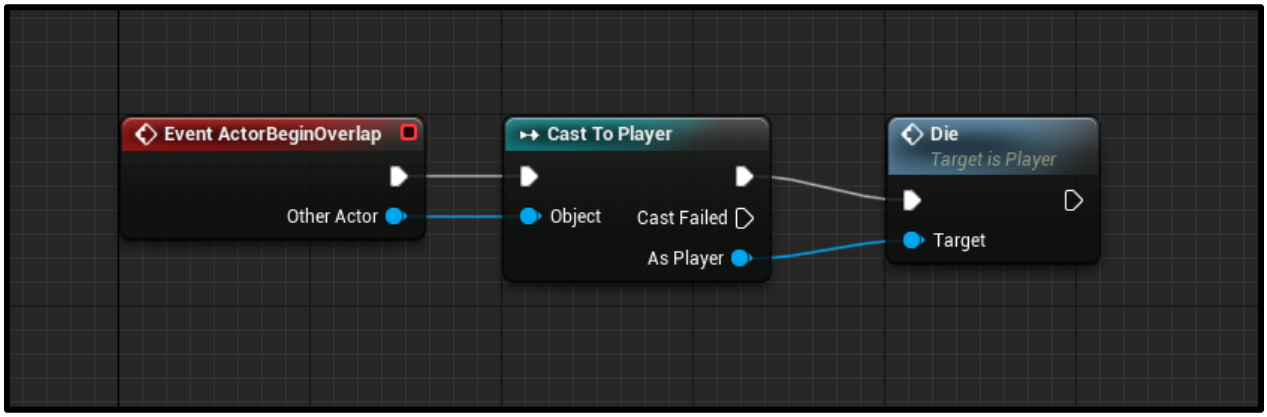
- Set the **Component** input to our *Static Mesh* component
- Set the **Target Relative Location** input to our *TargetPosition* variable
- Set the **Over Time** input to our *MoveTime* variable



When we've arrived at the target position, the *Completed* output flow will trigger. Here, we want to switch the **TargetPosition** variable to the start position and vice versa so that it bounces between the two.



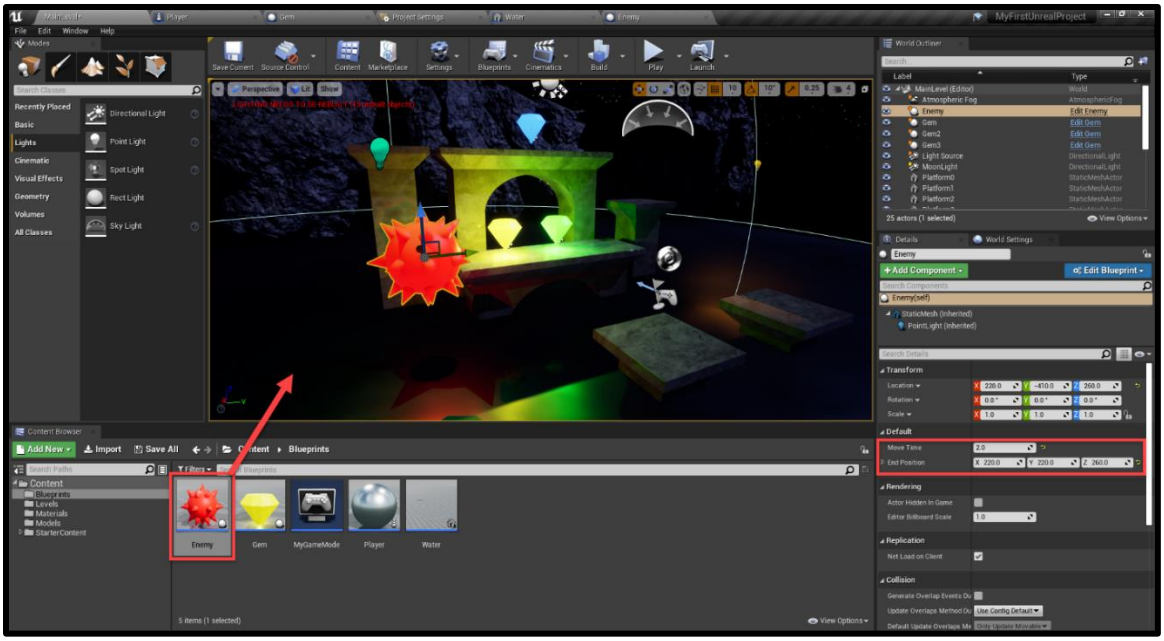
The movement is finished! Now let's work on colliding with the player. This is basically the same as the water blueprint.



Hit the **Compile** button. Then back in the level editor, we can drag in the **Enemy** blueprint.

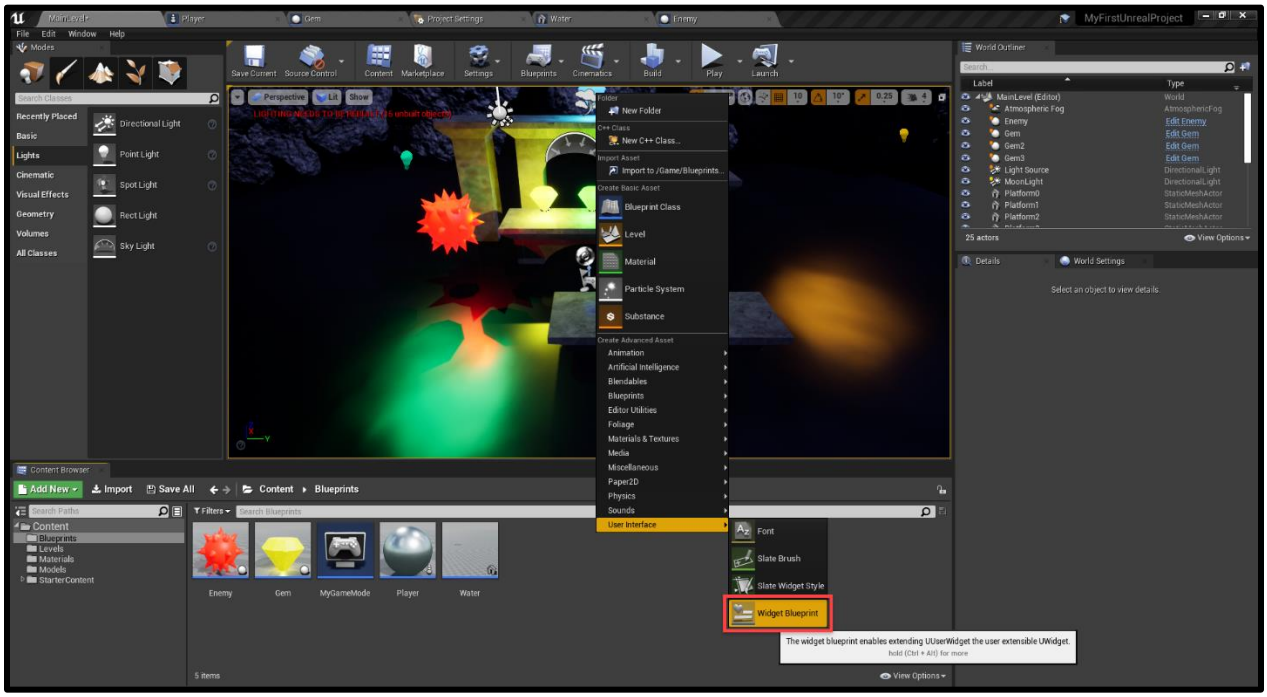
- Set the **Move Time** to *2*
- Set the **End Position** to the location where you want the enemy to move to

If you press play, you'll see that it will bounce between the two points! And if you hit it, the scene will reset.



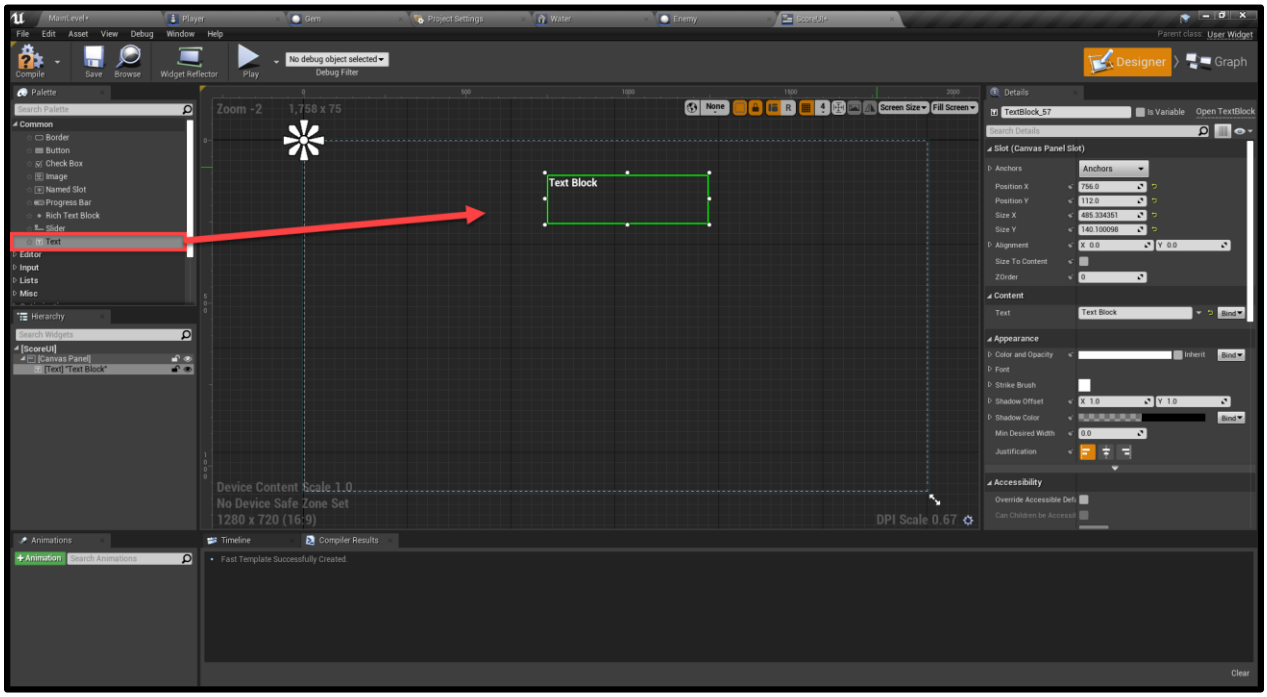
Score UI

Finally, let's setup our user interface (UI). This is going to be text on-screen which will display the player's score. In the **Blueprints** folder, right click, hover over *User Interface* and select **Widget Blueprint**. Call this **ScoreUI**.



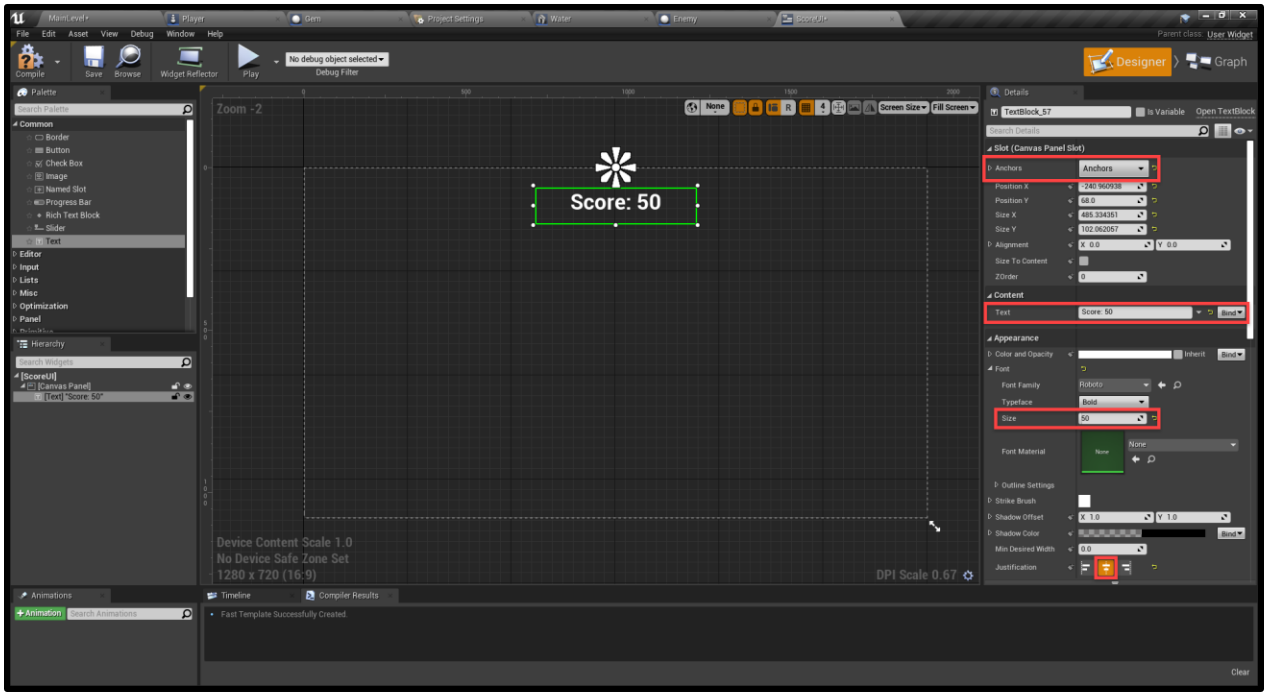
When you double click on it, it will open up in widget editor. Here, we can setup the UI elements and apply some logic to them.

In the **Palette** panel, we can see all the different UI components we can use. Drag in the **Text** component. You can click and drag to move it around and resize it by clicking and dragging on the white circles.

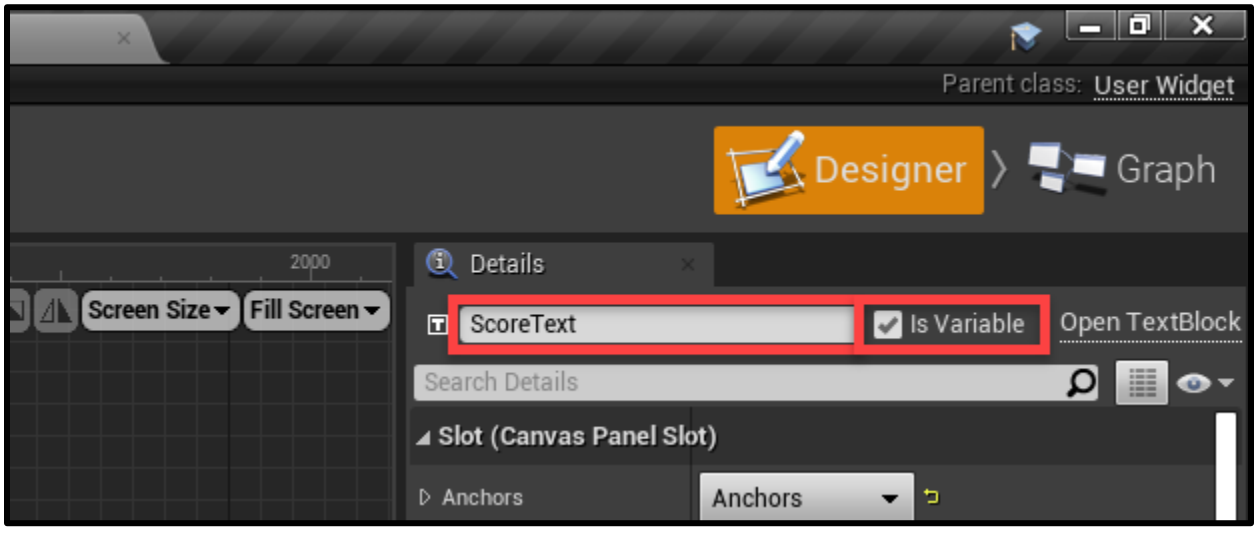


In the **Details** panel...

- Set the **Anchors** to *top-center*
    - Anchors are there for attaching the UI component to a point on the screen. This means when we resize the screen, the text will always be at the top center.
- Set the **Text** to *Score: 0* ***(different from image below)***
- Set the **Font - Size** to *50*
- Set the **Justification** to *Center*



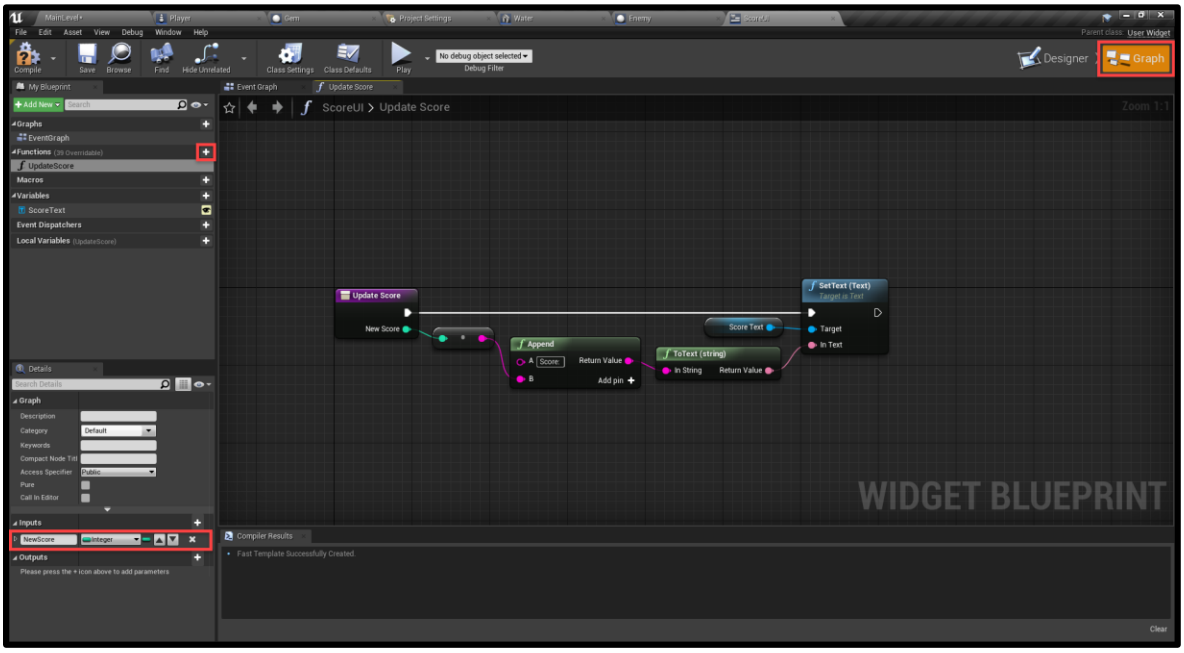
Also, change the name to *ScoreText* and enable **Is Variable**. This will allow us to modify it with nodes.



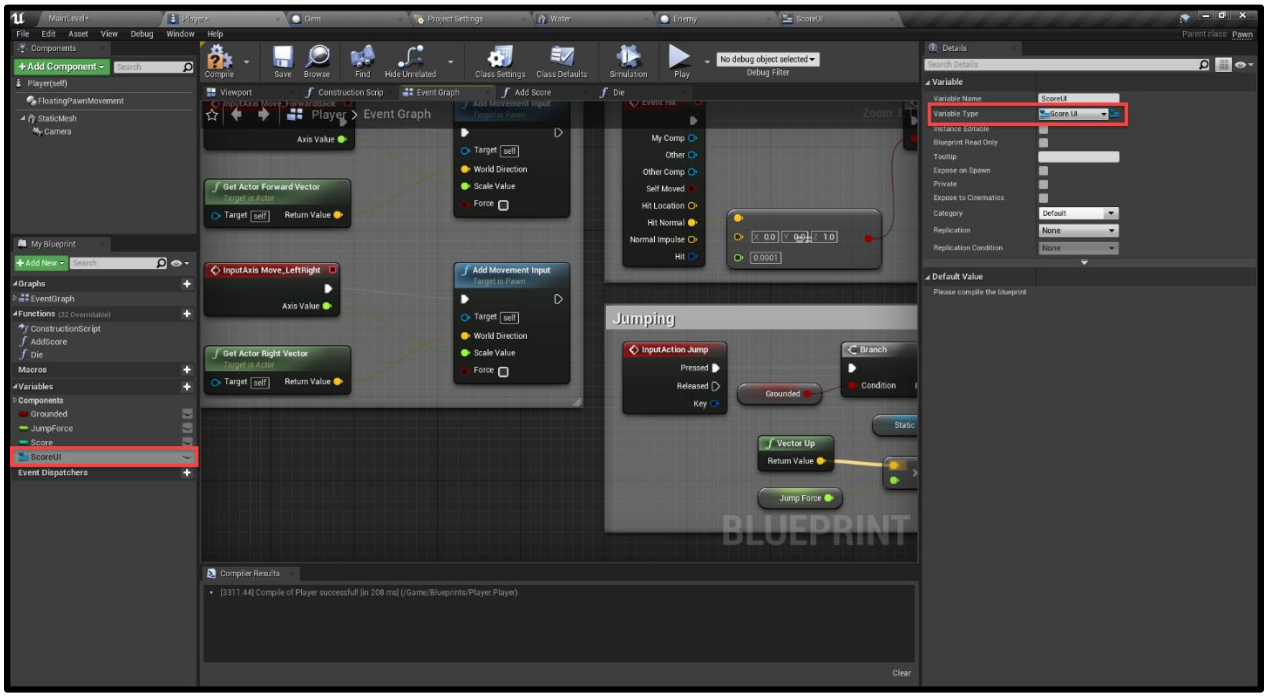
At the top right of the screen, click on the **Graph** button. This will switch us over to the event graph.

1. Create a new function called **UpdateScore**
2. Give it an integer input called **NewScore**
3. Layout the graph like in the image below

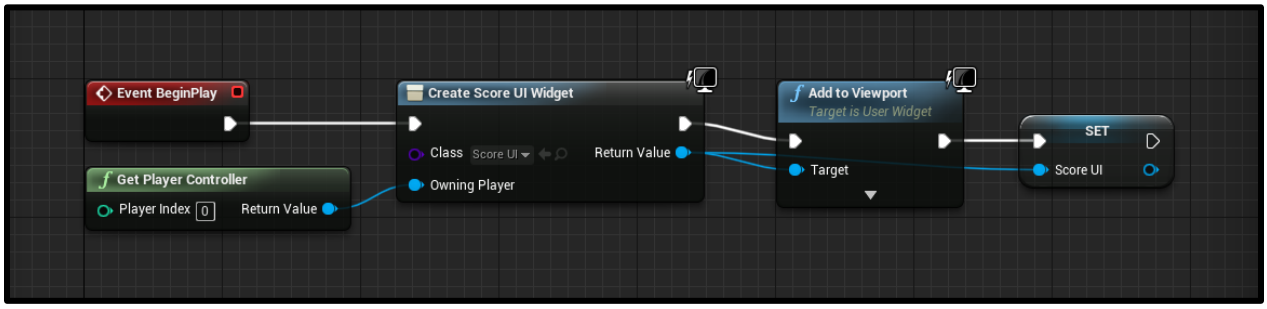
What we're doing here, is creating a new string (text) with two elements. First a "Score: " string combined with the new score. Then the score text is set to this new text.



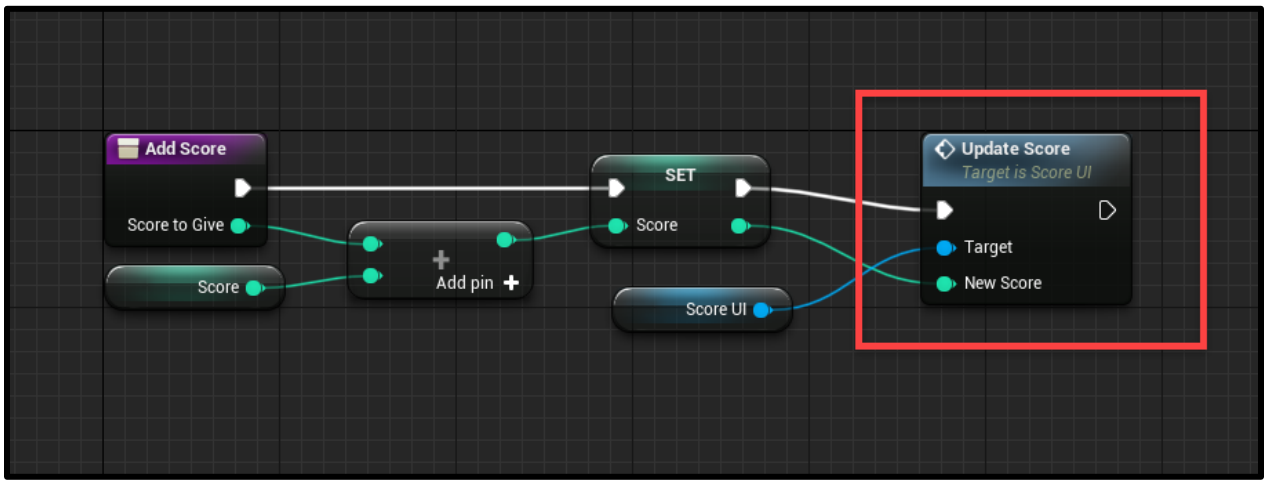
Now we need to hook this up to the player. In the **Player** blueprint, create a new variable of type **ScoreUI** called **ScoreUI**.



Let's then create the **Event BeginPlay** node which gets called when the game starts. We're going to create a new widget, add it to the viewport and then set that as our **ScoreUI** variable.



In the **AddScore** function, let's call the UI's **UpdateScore** text function.



You can now press play and see that when we collect gems, our score goes up!

Conclusion

There we go! We've completed our first game in the Unreal Engine - just like that.

Over the course of this Unreal Engine tutorial, we've covered a lot. Not only did we learn how to make and add a variety of objects to compose our 3D platformer level, but we learned extensively how to work with Unreal Engine's blueprinting system to compose every bit of logic of our game (without manually coding anything!).

From here you can choose to expand the existing game - adding in new features or upgrading the graphics. Or you could choose to take your new skill set and apply it to a totally new game. There are many different features and editors available in the Unreal Engine, so have a look at

them all. There are also many online resources that go over each aspect of the engine, what each node does, and how to do specific things that you may want. Either way, the foundations you've developed here will provide a great starting point for you to become an expert Unreal Engine developer!

We wish you the best of luck with your future games!

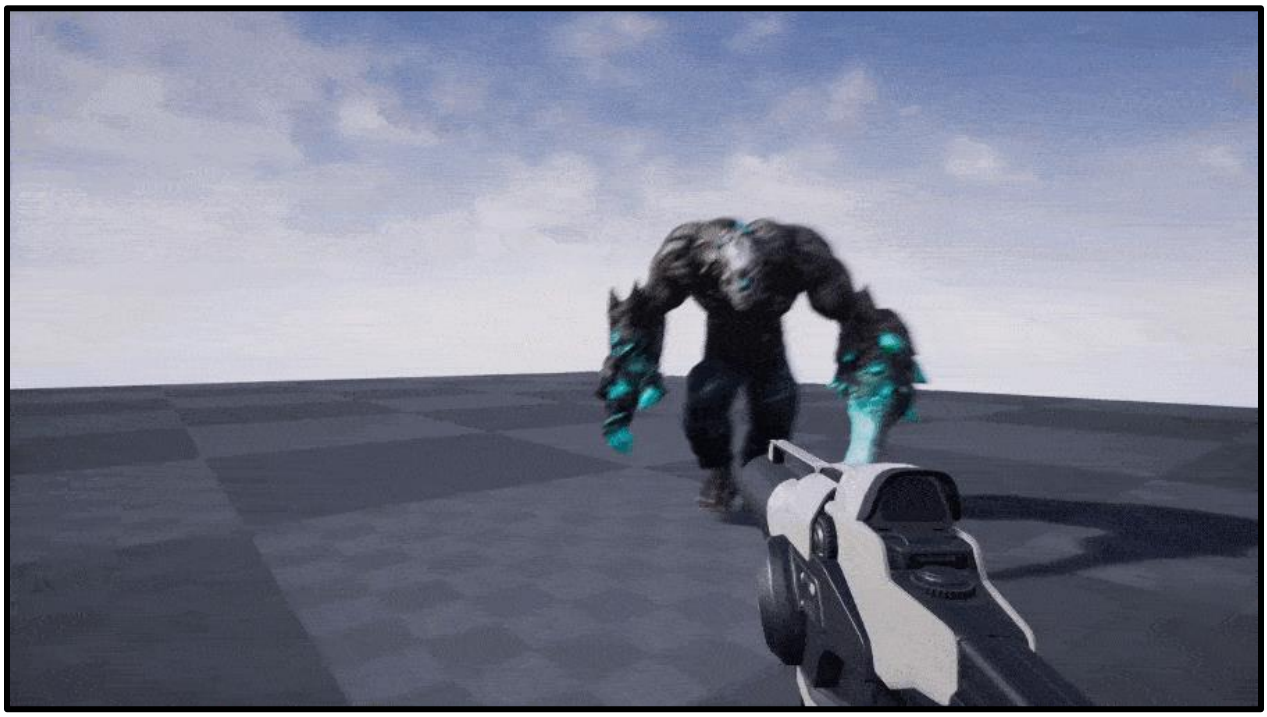
How to Create a First-Person Shooter in the Unreal Engine

Introduction

Whether you're a fan or not, there's no denying that first-person shooters are a popular game genre. Thus, it can be beneficial to know how to make one, whether it's just to round out your own skills or to create that FPS game idea you've had haunting your dreams. It can be even more beneficial to learn how to do this with Unreal Engine, the popular and graphically stunning engine behind a ton of popular games.

In this tutorial, we're going to show you how to create a first-person shooter game inside of the Unreal Engine. The game will feature a first-person player controller, enemy AI with animations, and various shooting mechanics. We will also cover the Unreal Engine blueprinting system for the logic, so no other coding knowledge is needed to jump in!

Before we start though, it's important to know that this tutorial won't be going over the basics of Unreal Engine. If this is your first time working with the engine, we recommend you follow our intro tutorial.



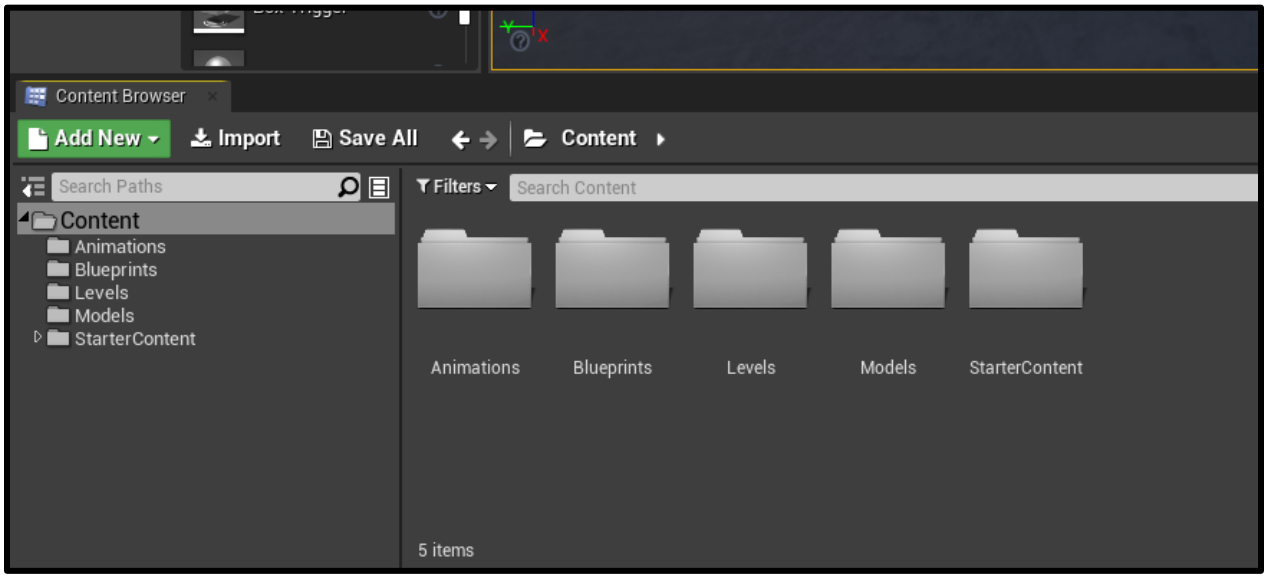
Project Files

There are a few assets we'll be needing for this project such as models and animations. You can also download the complete Unreal project via the same link here.

Project Setup

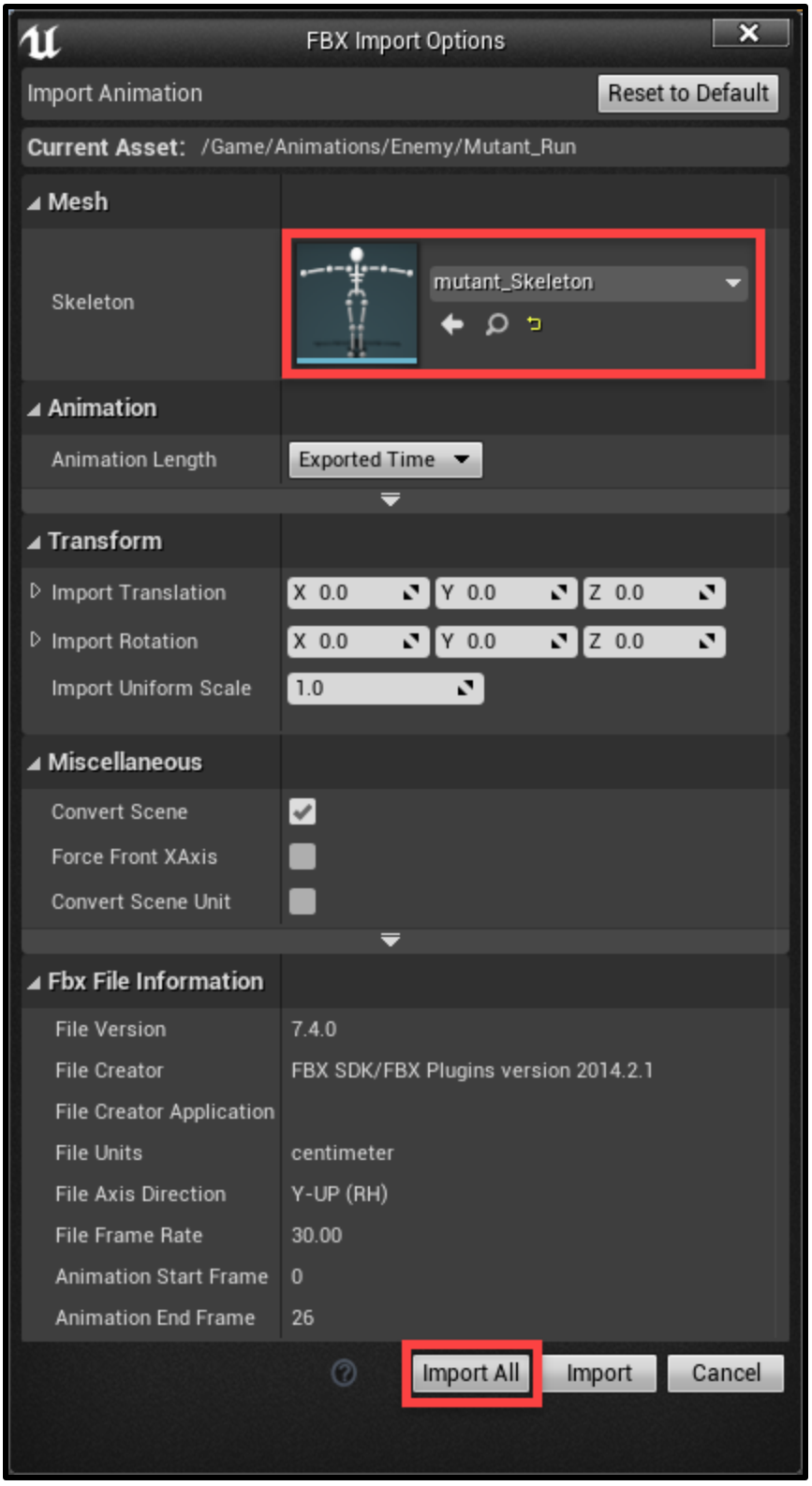
To begin, create a new project making sure to include the **Starter Content**. This should then open up the Unreal Engine editor. Down in the **Content Browser**, create four new folders.

- Animations
- Blueprints
- Levels
- Models

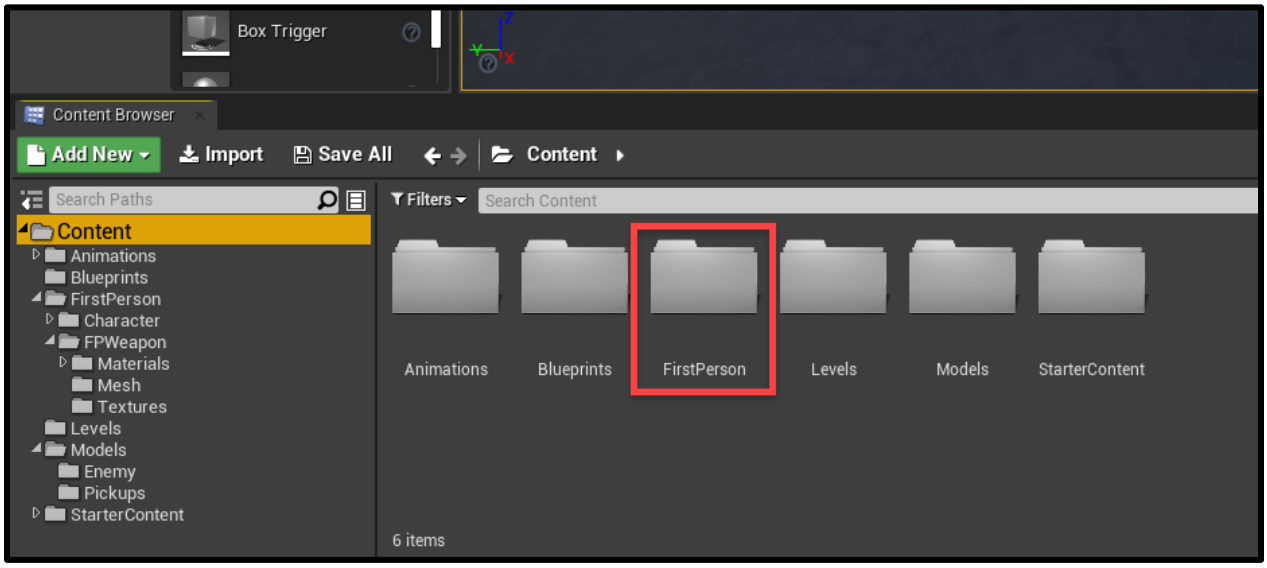


Next, download the required assets (link at the start of tutorial). Inside the .ZIP file are three folders. Begin by opening the **Models Folder Contents** folder and drag all the folders inside of that into the **Models** folder in the **Content Browser**. Import all of that.

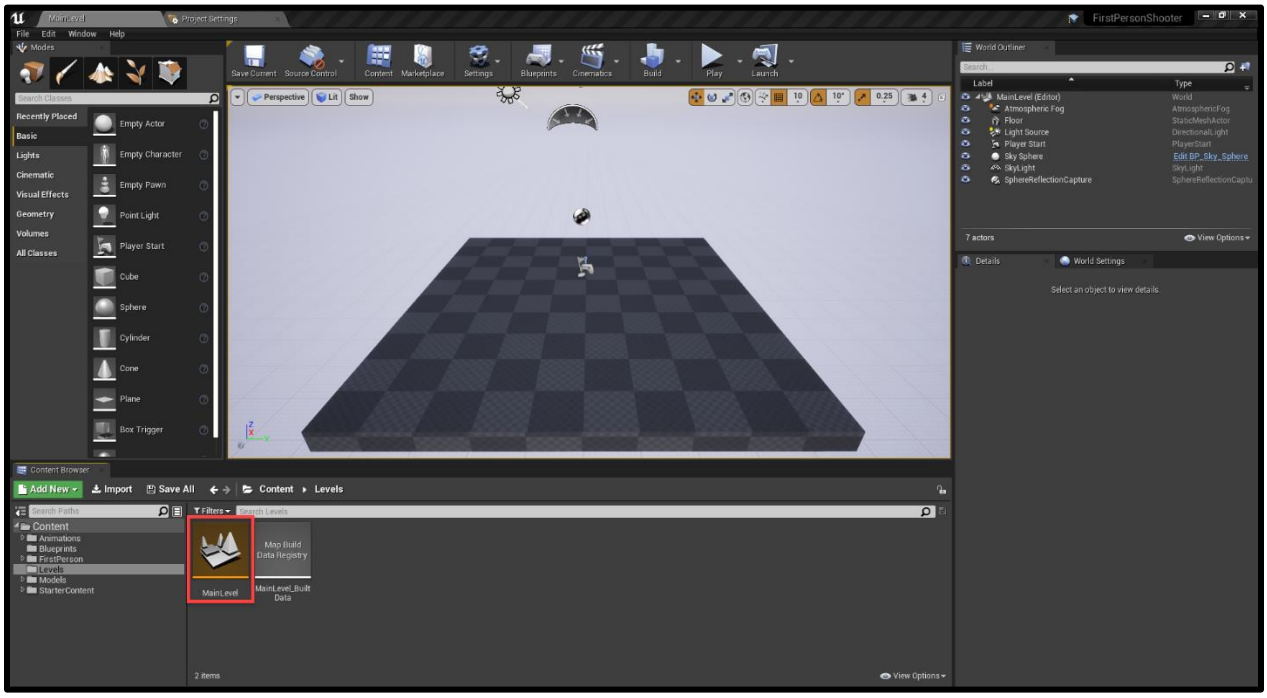
Then do the same for the **Animations Folder Contents**. When those prompt you to import, set the **Skeleton** to *mutant_Skeleton*. The mutant model and animations are free from Mixamo.



Finally, drag the **FirstPerson** folder into the editor's **Content** folder. This a is a gun asset that comes from one of the Unreal template projects.

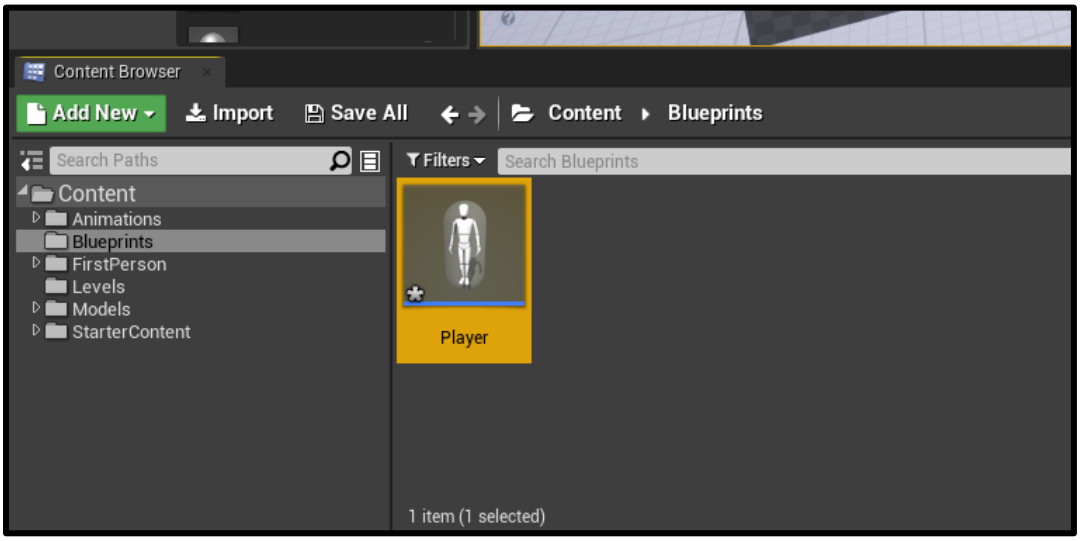


Now we can create a new level (*File > New Level*) and select the **Default** level template. When the new level opens, save it to the **Levels** folder as *MainLevel*.



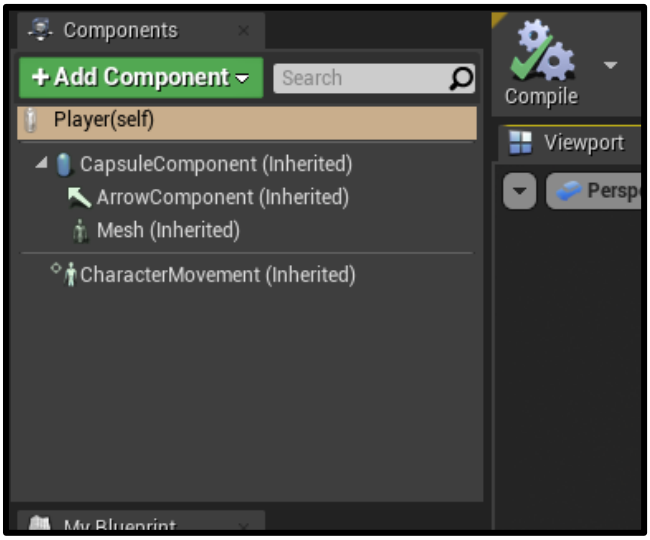
Setting Up the Player

Let's now create the first-person player controller. In the **Blueprints** folder, create a new blueprint with the parent class of **Character**. Call it *Player*. The character parent includes many useful things for a first-person controller.



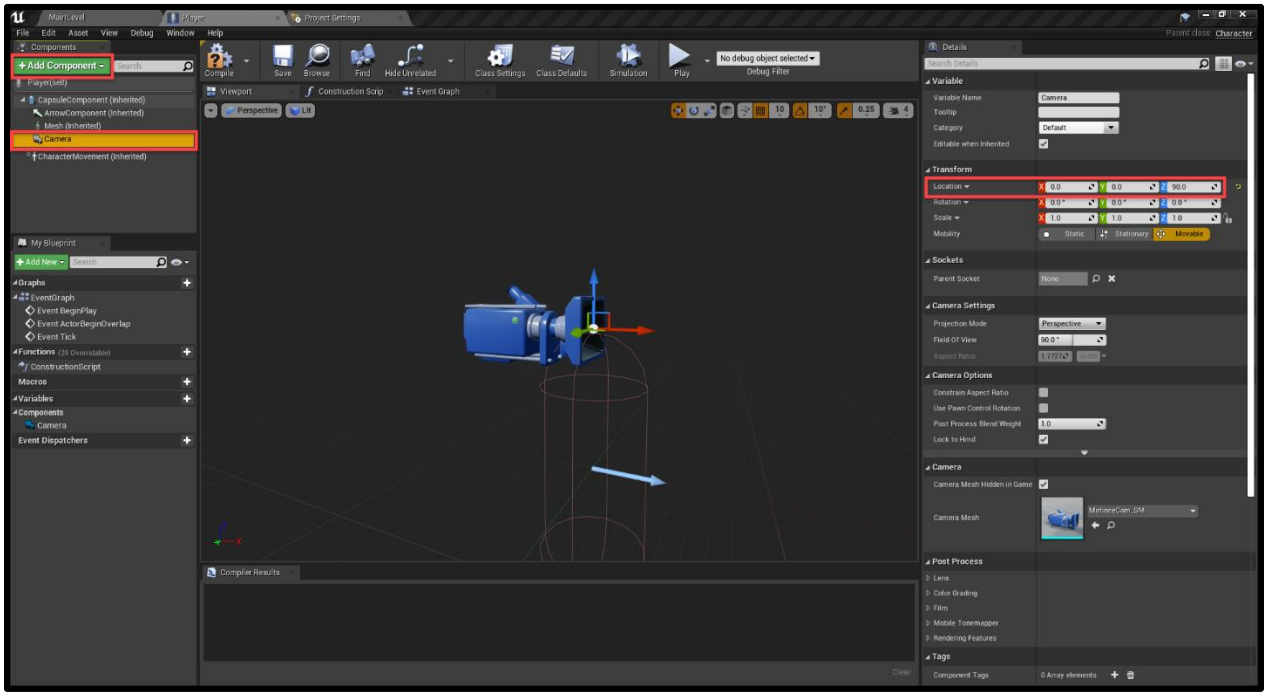
Double click it to open the player up in the blueprint editor. You'll see we have a few components already there.

- CapsuleComponent = our collider
    - ArrowComponent = forward direction
    - Mesh = skeletal character mesh
- CharacterMovement = movement, jumping, etc.



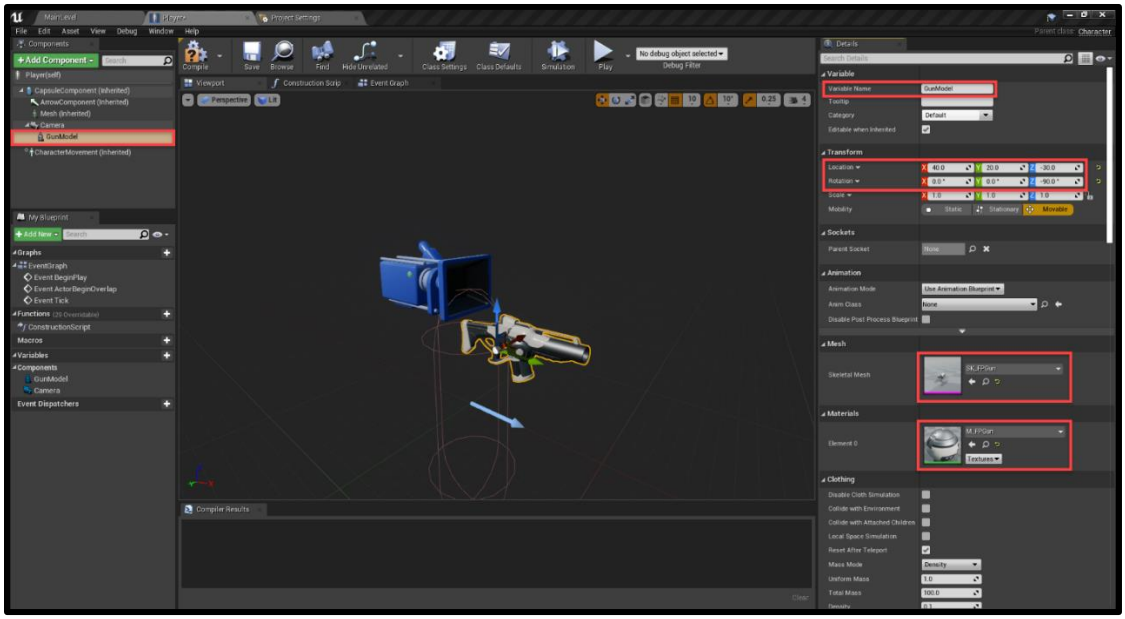
We can start by creating a new **Camera** component. This allows us to see through our player's eyes into the world.

- Set the **Location** to *0, 0, 90* so that the camera is at the top of the collider.



For the gun, create a **Skeletal Mesh** component and drag it in as a child of the camera.

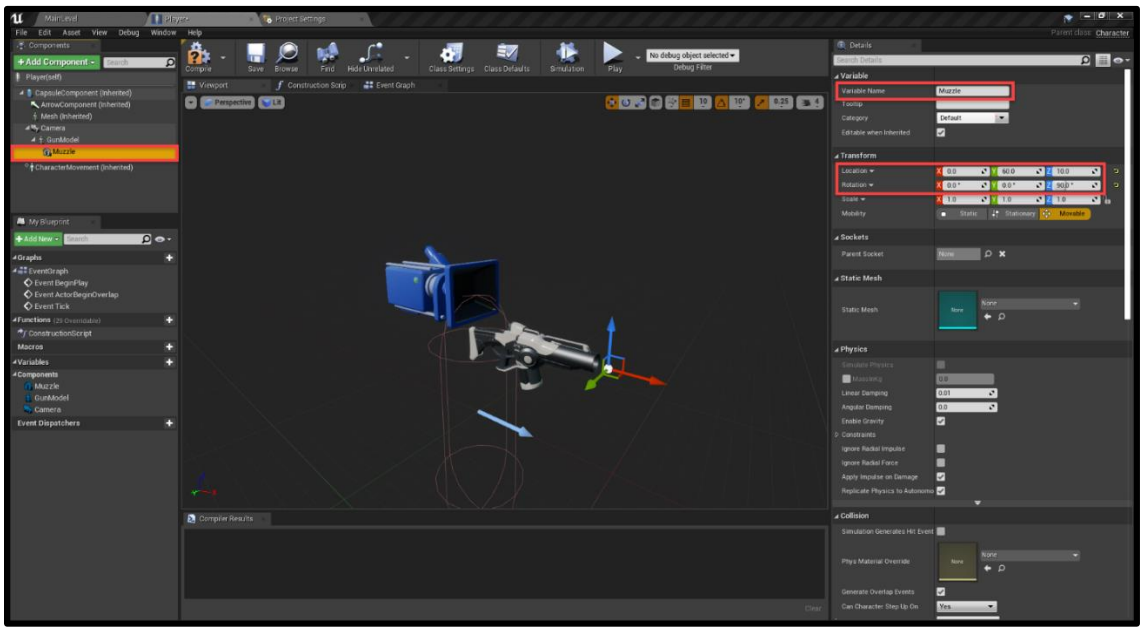
- Set the **Variable Name** to *GunModel*
- Set the **Location** to *40, 0, -90*
- Set the **Rotation** to *0, 0, -90*
- Set the **Skeletal Mesh** to *SK_FPGun*
- Set the **Material** to *M_FPGun*



Finally, we can create the muzzle. This is a **Static Mesh** component which is the child of the gun model.

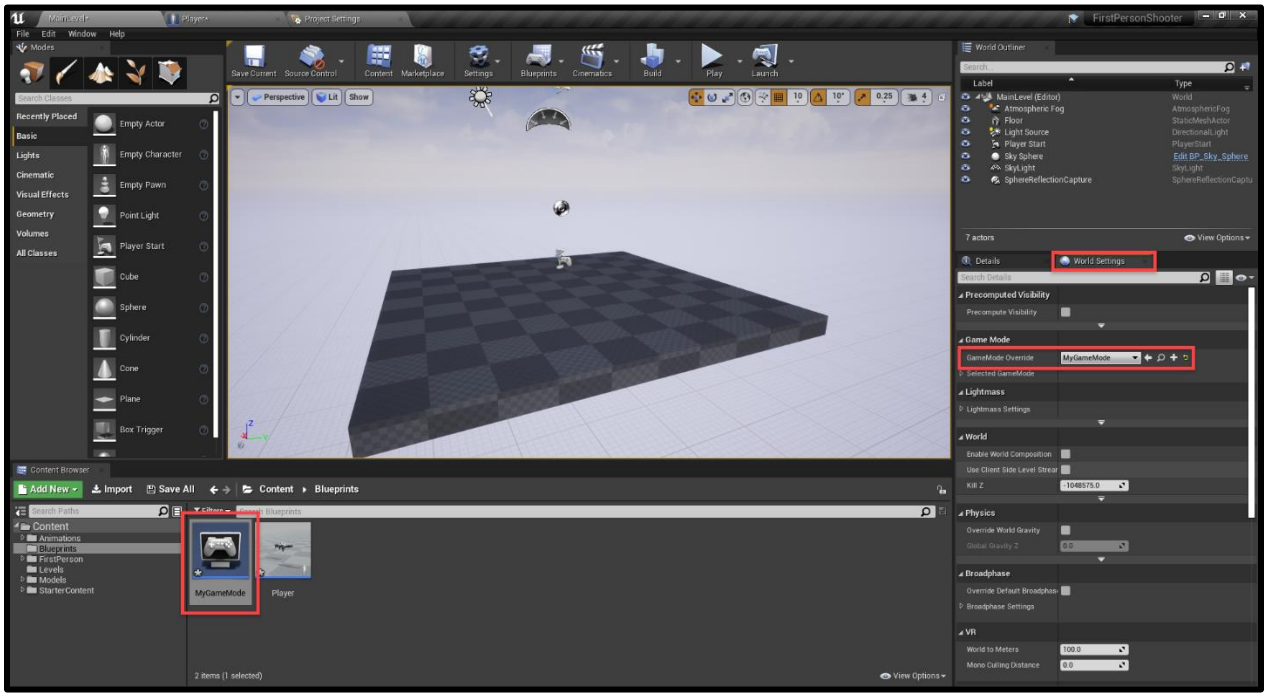
- Set the **Variable Name** to *Muzzle*
- Set the **Location** to *0, 60, 10*
- Set the **Rotation** to *0, 0, 90*

We're using this as an empty point in space, so no model is needed.

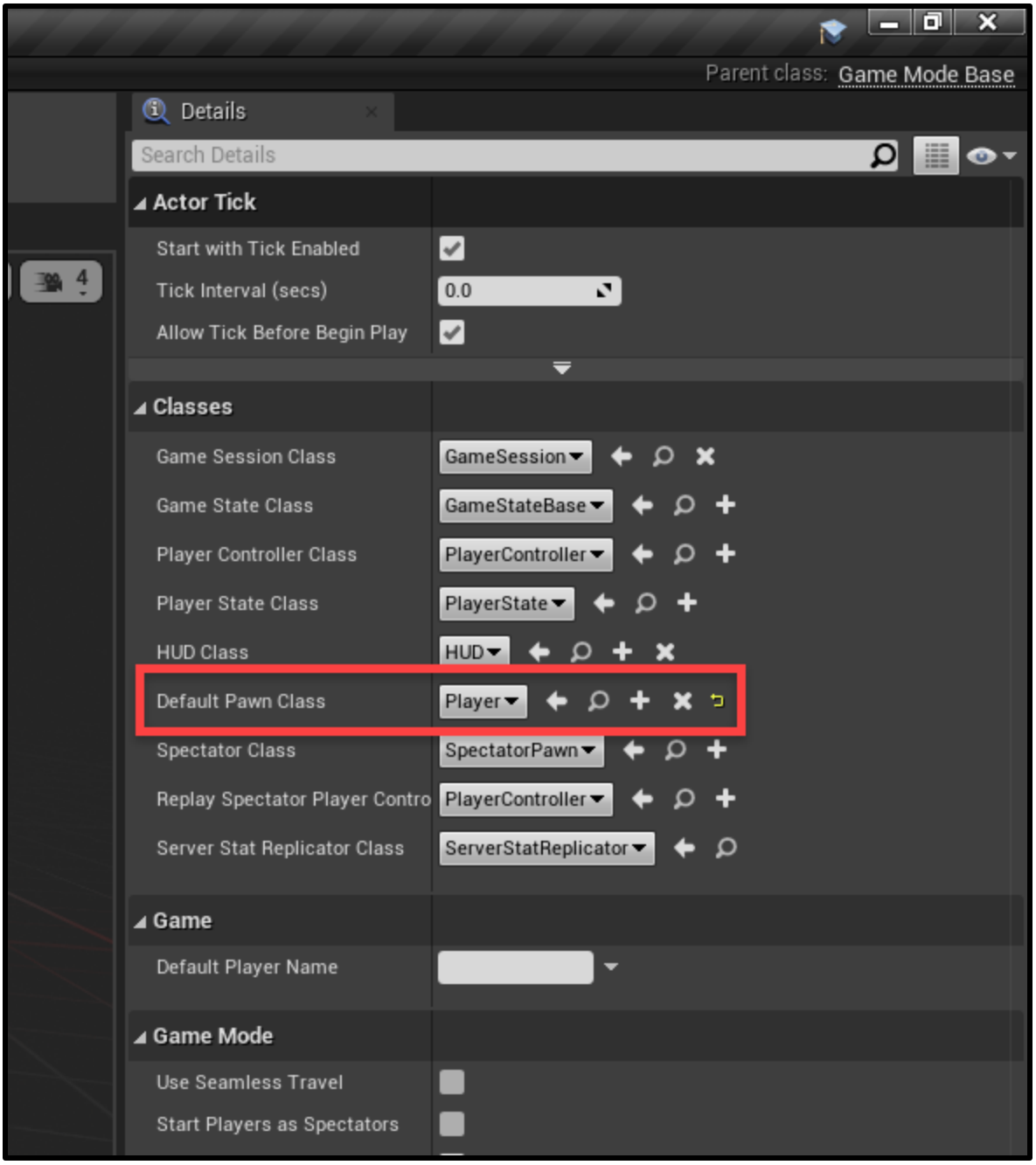


That's it for the player's components! In order to test this out in-game, let's go back to the level editor and create a new blueprint. With a parent class of **Game Mode Base**, call it **MyGameMode**. This blueprint will tell the game what we want the player to be, etc.

In the **Details** panel, click on the **World Settings** tab and set the **GameMode Override** to be our new game mode blueprint.



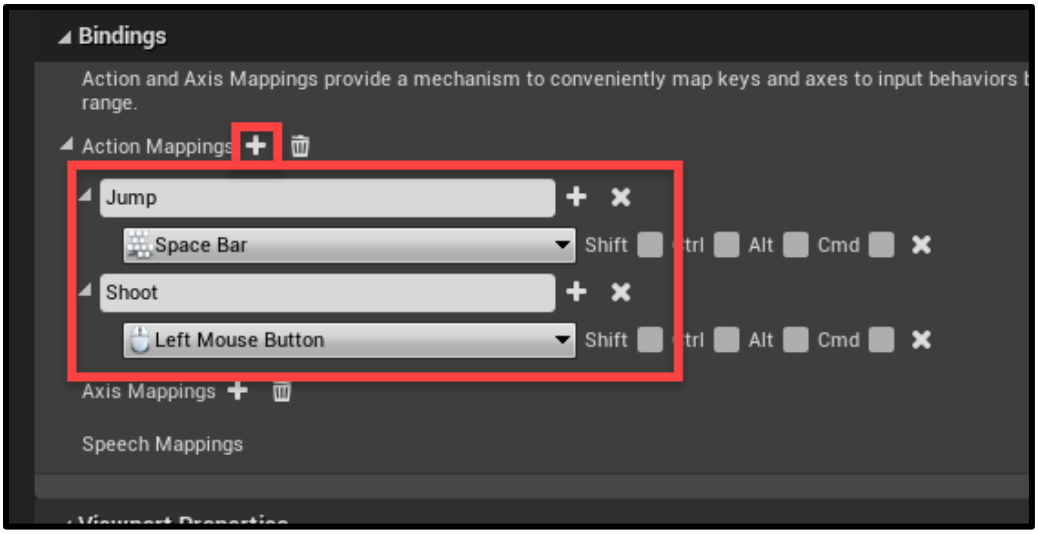
Next, double-click on the blueprint to open it up. All we want to do here is set the **Default Pawn Class** to be our player blueprint. Once that's done: save, compile, then go back to the level editor.



One last thing before we start creating logic for the player, and that is the key bindings. Navigate to the **Project Settings** window (*Edit > Project Settings...*) and click on the **Input** tab.

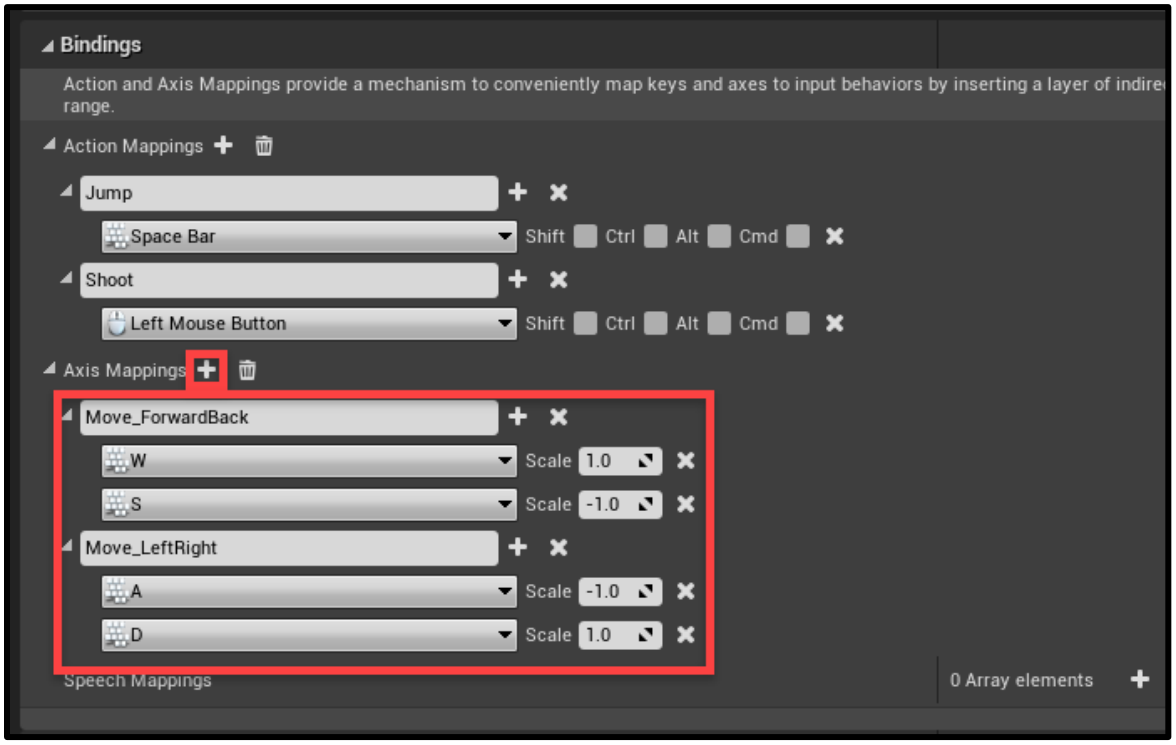
Create two new *Action Mappings*.

- Jump = space bar
- Shoot = left mouse button



Then we want to create two *Axis Mappings*.

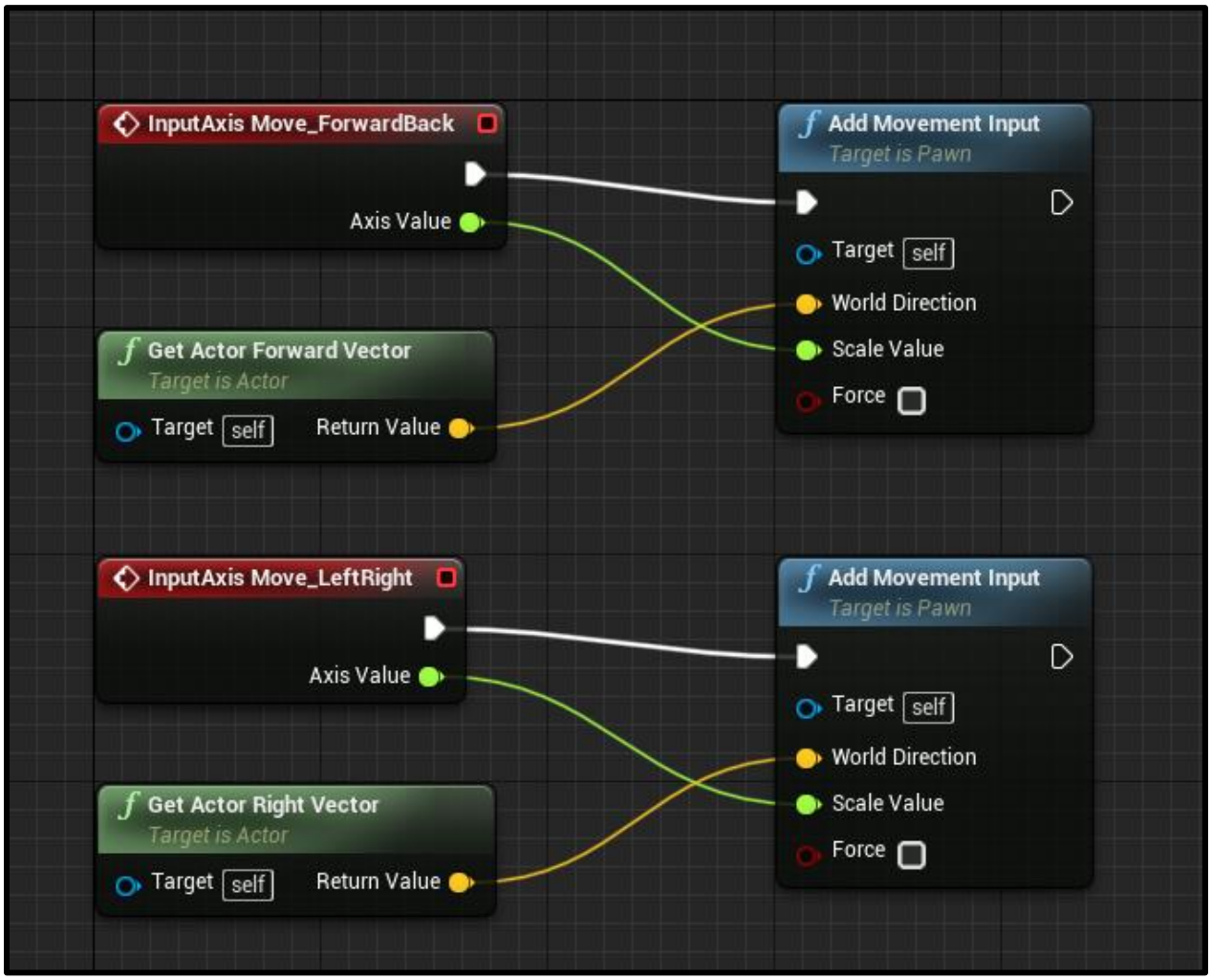
- Move_ForwardBack
  - *W* with a scale of 1.0
  - *S* with a scale of -1.0
- Move_LeftRight
  - *A* with a scale of -1.0
  - *D* with a scale of 1.0



Player Movement

Back in our **Player** blueprint, we can implement the movement, mouse look and jumping. Navigate to the **Event Graph** tab to begin.

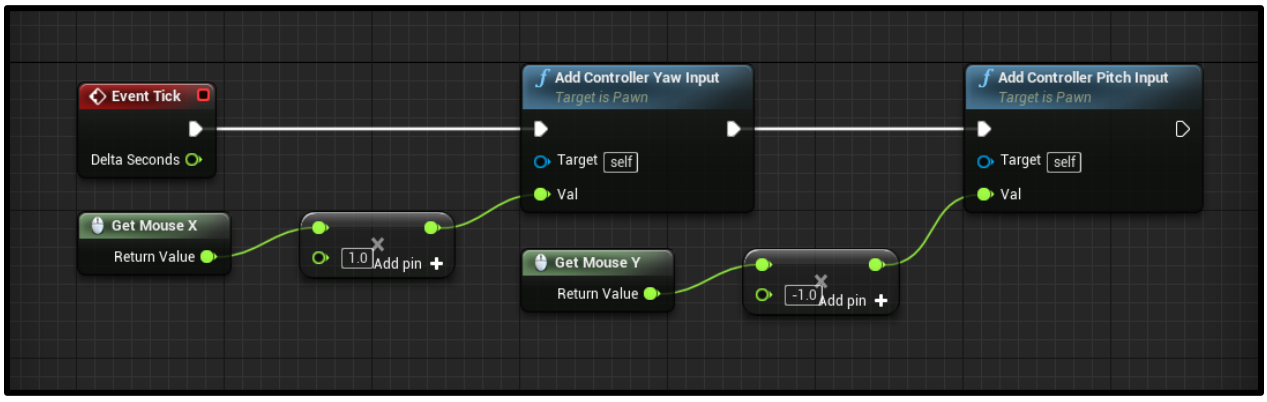
First we have the movement. The two axis input event nodes will plug into an **Add Movement Input** node, which will move our player.



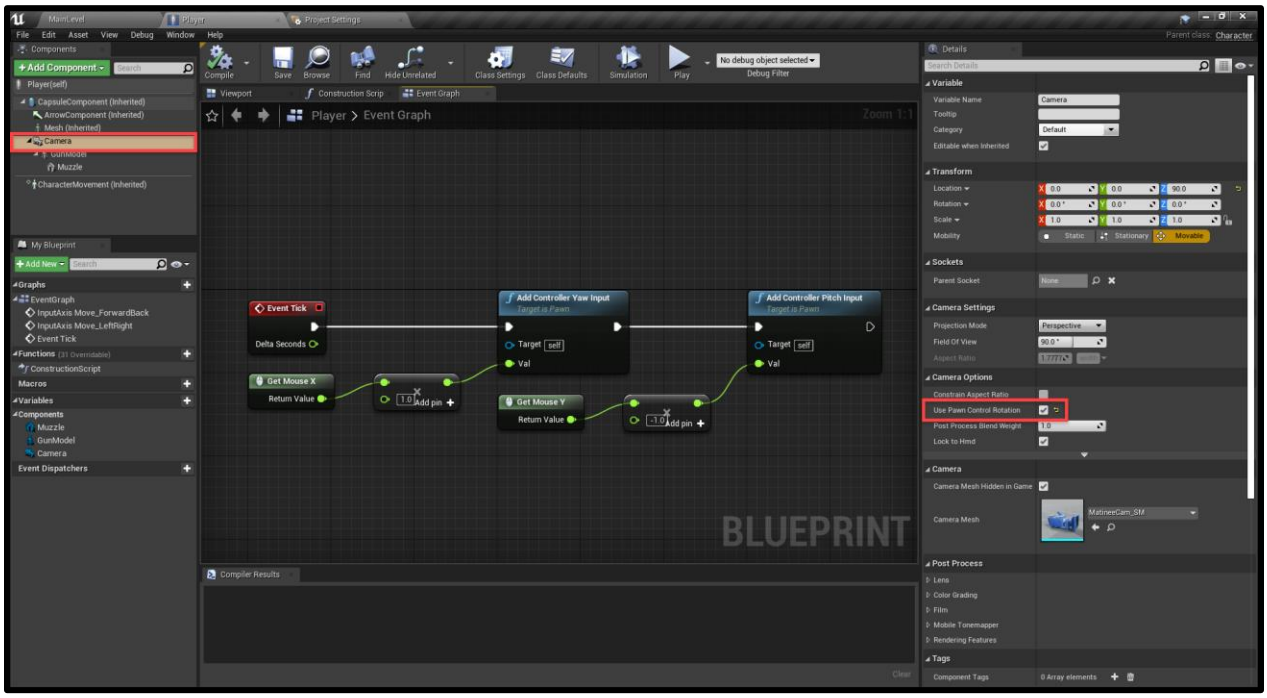
We can then press **Play** to test it out.

Next up, is the mouse look. We've got our camera and we want the ability to rotate it based on our mouse movement. We want this to be triggered every frame so start out with an **Event Tick** node.

This then plugs into an **AddControllerYawInput** and an **AddControllerPitchInput**. These nodes will rotate the camera along the Z and Y axis'. The amount will be based on the mouse movement multiplied by 1 and -1. -1 because the mouse Y value inverts the camera movement.



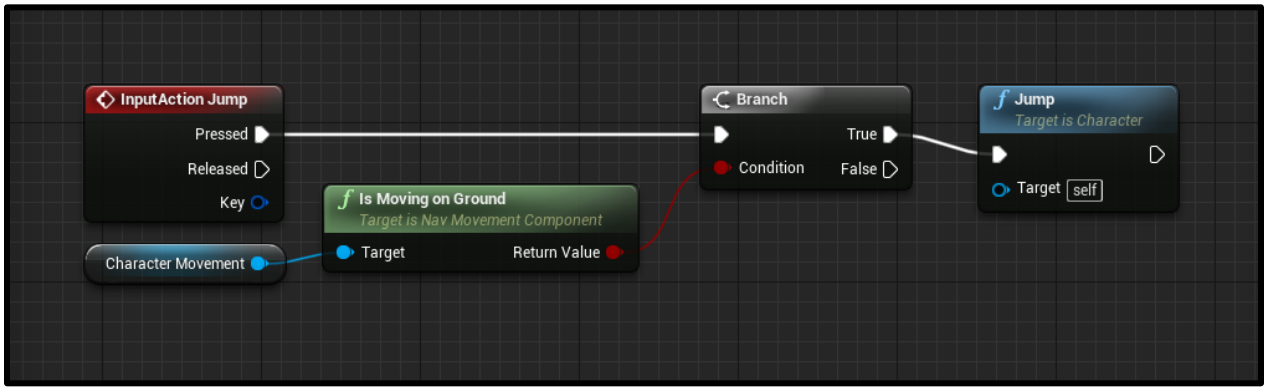
If you press play, you'll see that you can look left to right, but not up and down. To fix this, we need to tell the blueprint that we want to the camera to use the pawn's control rotation. Select the **Camera** and enable **Use Pawn Control Rotation**.



Now when you press play, you should be able to move and look around!

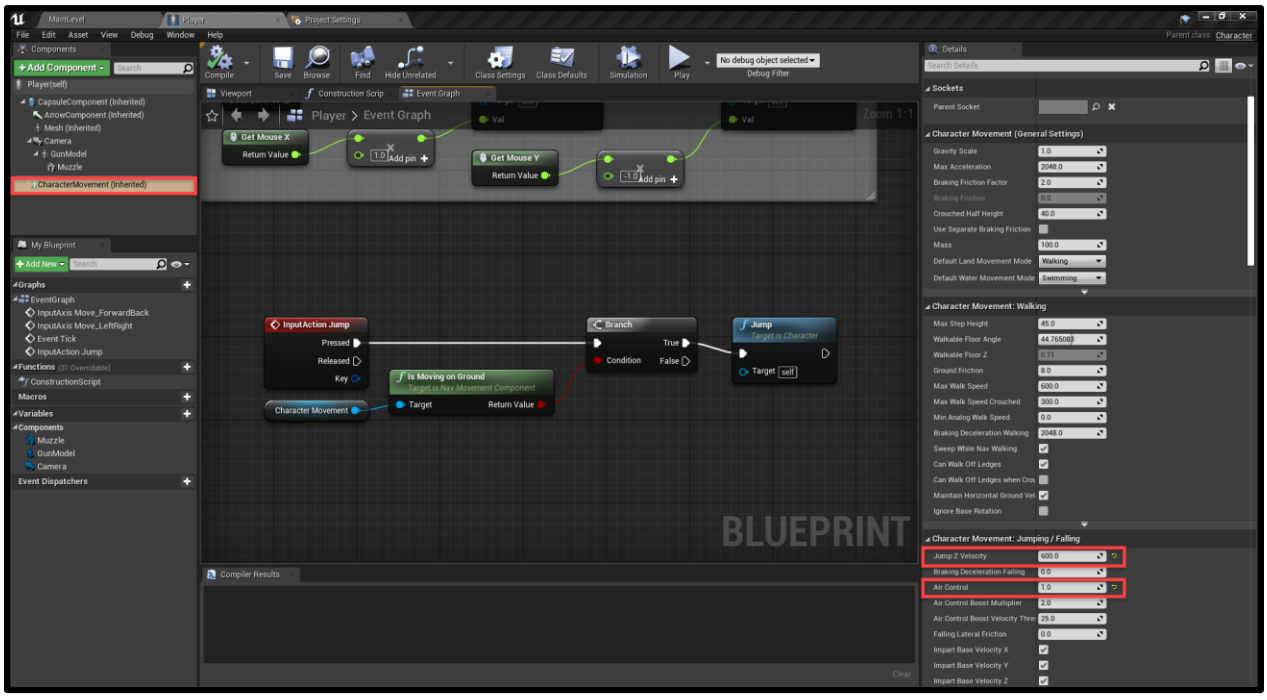
Jumping is relatively easy since we don't need to manually calculate whether or not we're standing on the ground. The **CharacterMovement** component has many build in nodes for us to do this easily.

Create the **InputAction Jump** node which is an event that gets triggered when we press the jump button. We're then checking if we're moving on ground (basically, are we standing on the ground). If so, jump!



We can change some of the jump properties. Select the **CharacterMovement** component.

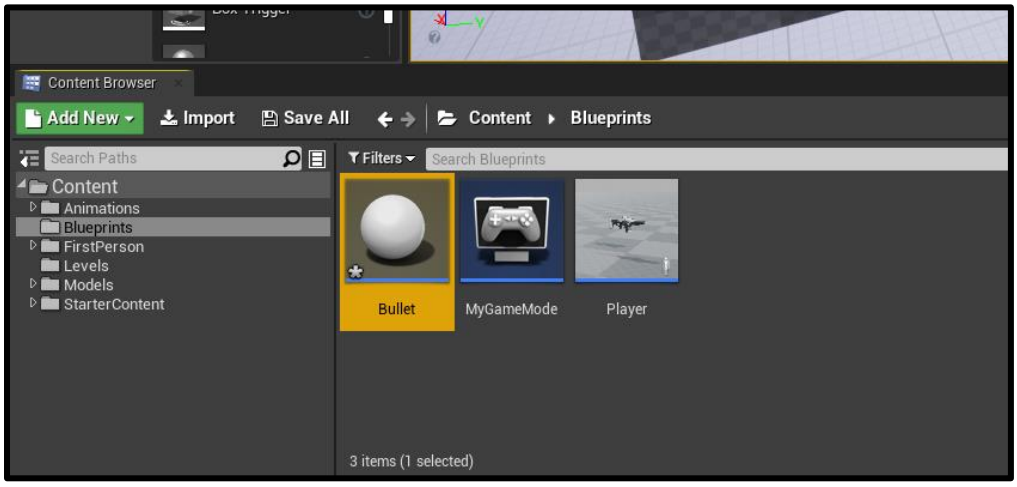
- Set the **Jump Z Velocity** to *600*
- Set the **Air Control** to *1.0*



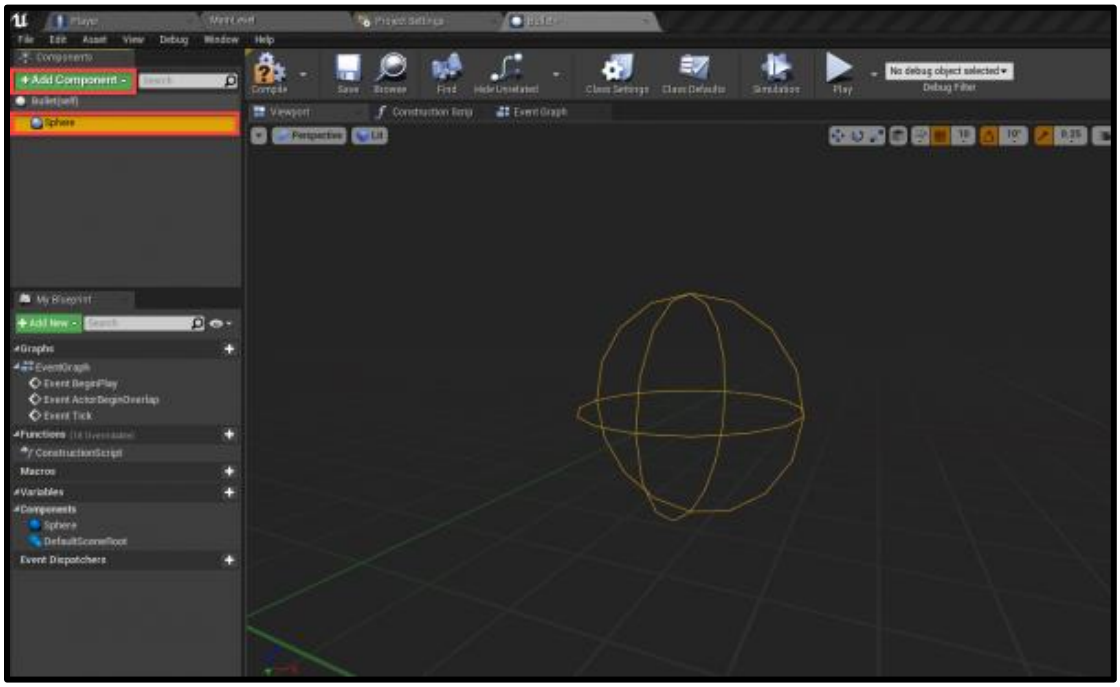
You can tweak the many settings on the component to fine tune your player controller.

Creating a Bullet Blueprint

Now it's time to start on our shooting system. Let's create a new blueprint with a parent type of **Actor**. Call it **Bullet**.



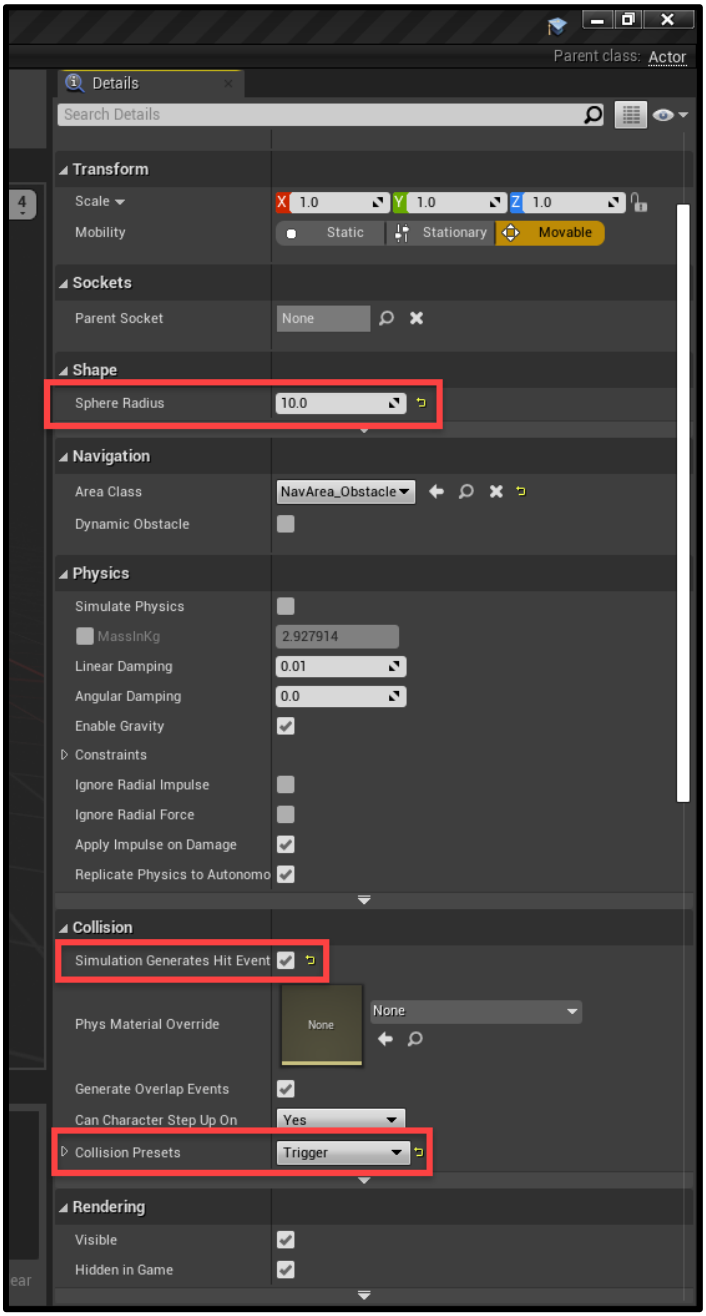
Open it up in the blueprint editor. First, let's create a new **Sphere Collision** component and drag it over the root node to make it the parent.



In the **Details** panel...

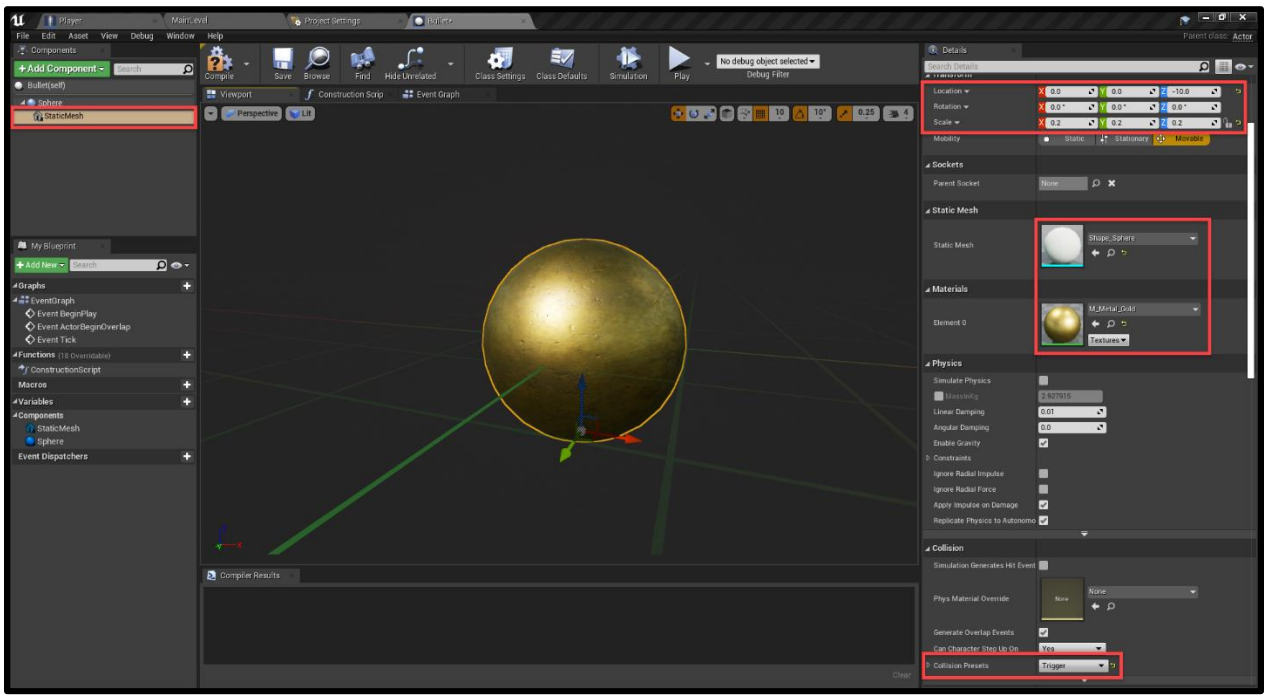
- Set the **Sphere Radius** to *10*
- Enable **Simulation Generates Hit Events**
- Set the **Collision Presets** to *Trigger*

Now we have the ability to detect collisions when the collider enters another object.



Next, create a new component of type **Static Mesh**.

- Set the **Location** to *0, 0, -10*
- Set the **Scale** to *0.2, 0.2, 0.2*
- Set the **Static Mesh** to *Shape_Sphere*
- Set the **Material** to *M_Metal_Gold*
- Set the **Collision Presets** to *Trigger*

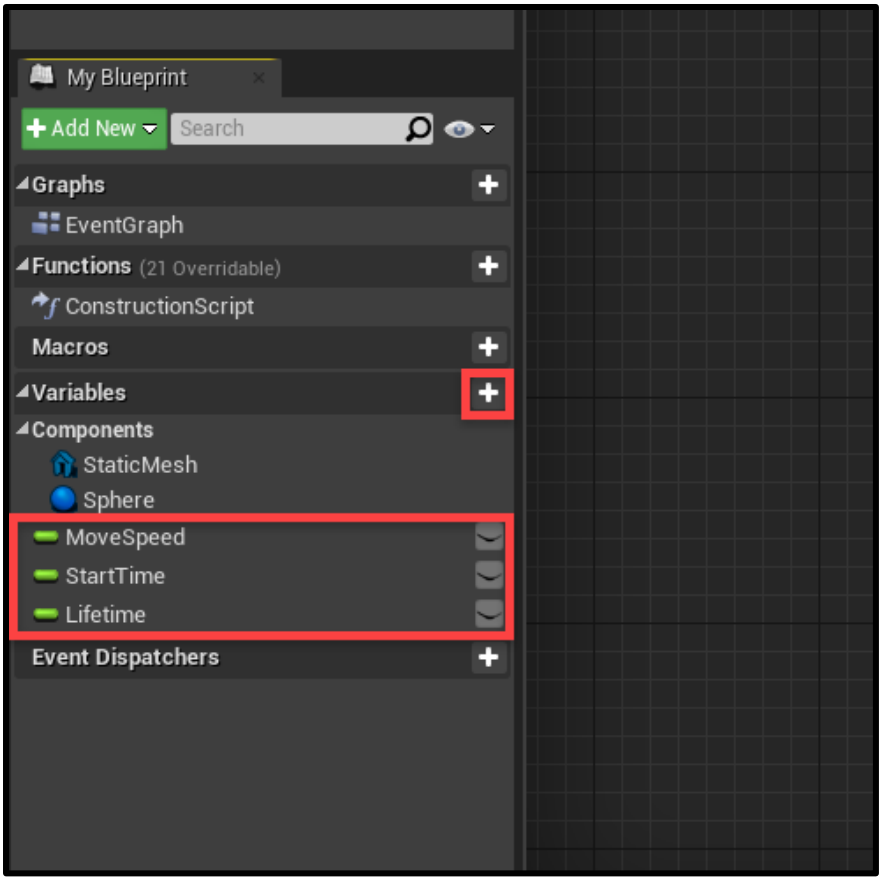


Go over to the **Event Graph** tab and we can begin with out variables.

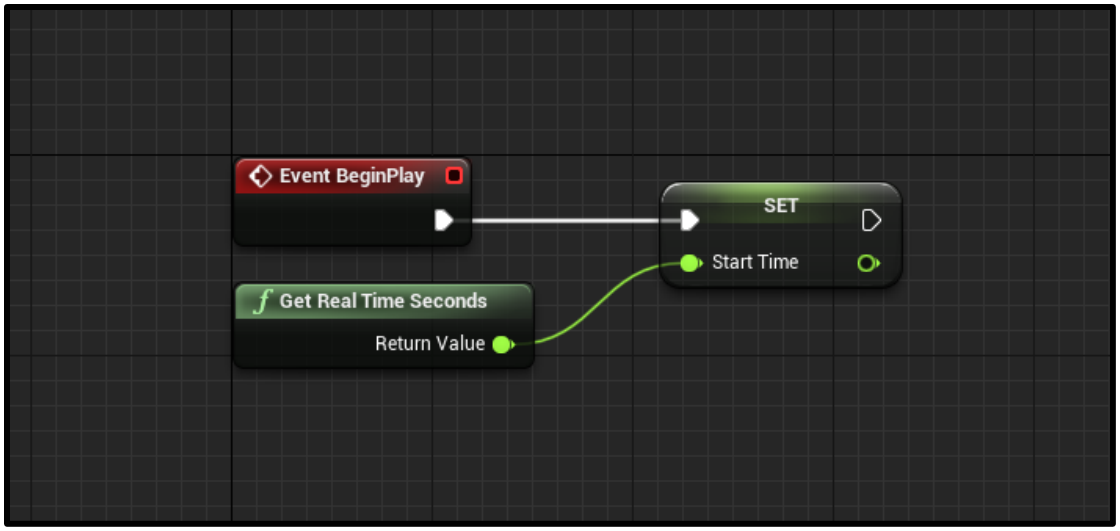
- MoveSpeed (Float)
- StartTime (Float)
- Lifetime (Float)

Then click the **Compile** button. This will allow us to now set some default values.

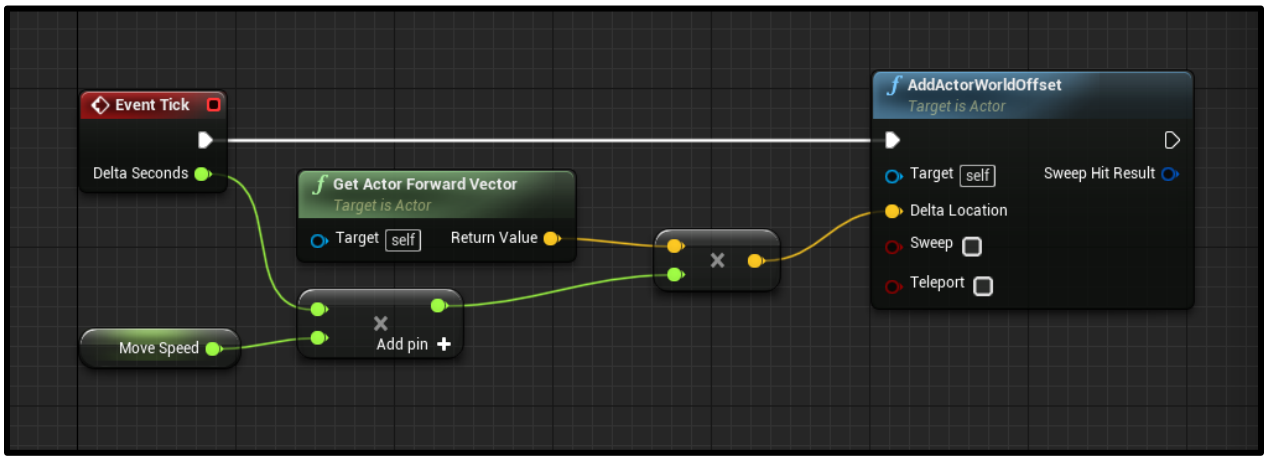
- MoveSpeed = 5000.0
- Lifetime = 2.0



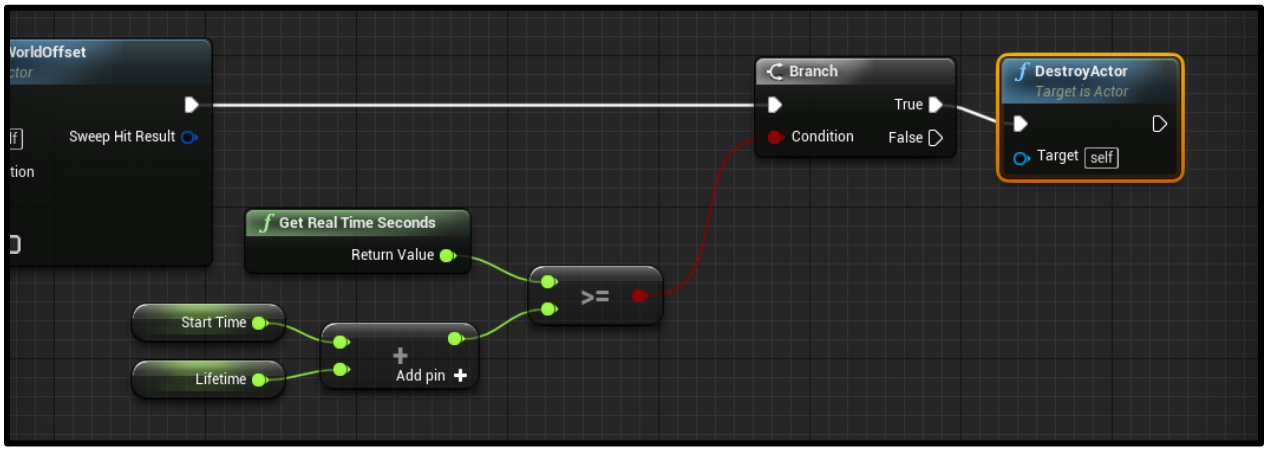
We want our bullet to destroy itself after a set amount of time so that if we shoot the sky, it won't go on forever. So our first set of nodes is going to set the start time variable at the start of the game.



Over time, we want to move the bullet forward. So we'll have the event tick node plug into an **AddActorWorldOffset** node. This adds a vector to our current location, moving us. The delta location is going to be our forward direction, multiplied by our move speed. You'll see that we're also multiplying by the event tick's *Delta Seconds*. This makes it so the bullet will move at the same speed, no matter the frame rate.



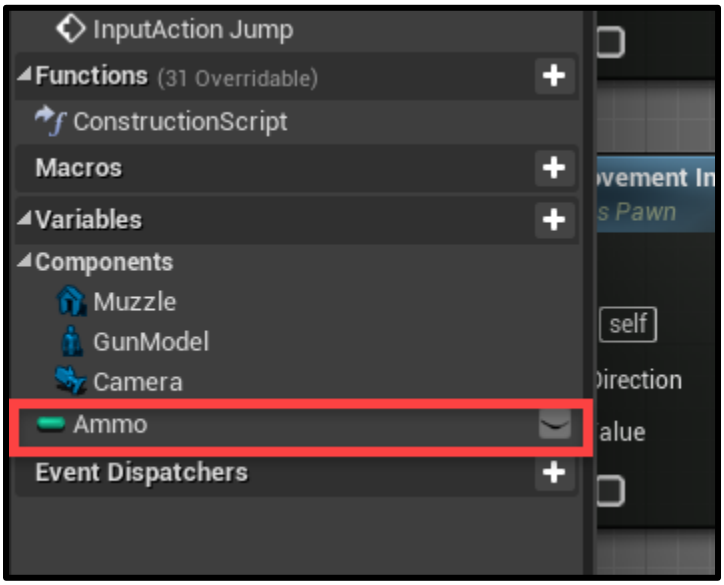
Connect the flow to a **Branch** node. We'll be checking each frame if the bullet has exceeded its lifetime. If so, we'll destroy it.



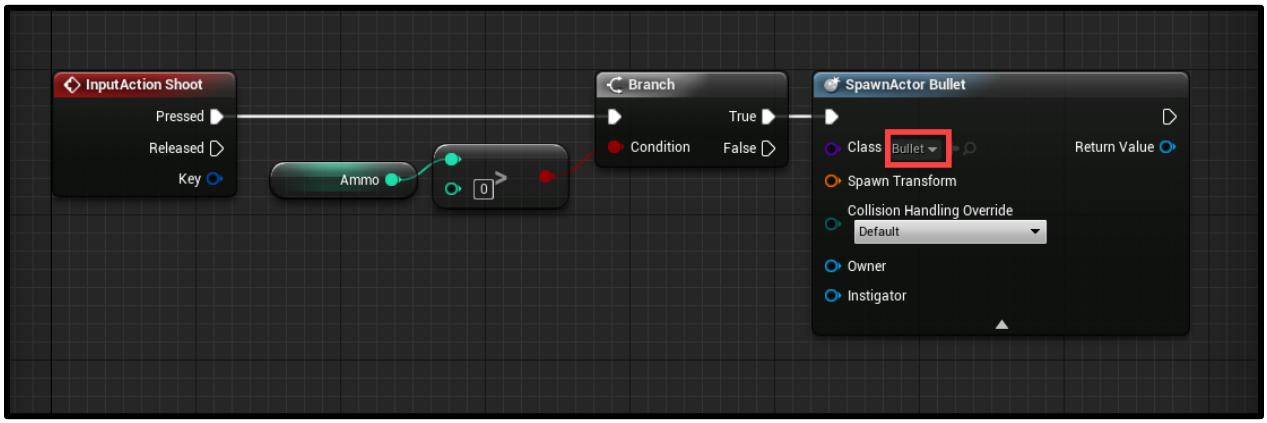
All the bullet needs now, is the ability to hit enemies. But we'll work on that once enemies are implemented.

Shooting Bullets

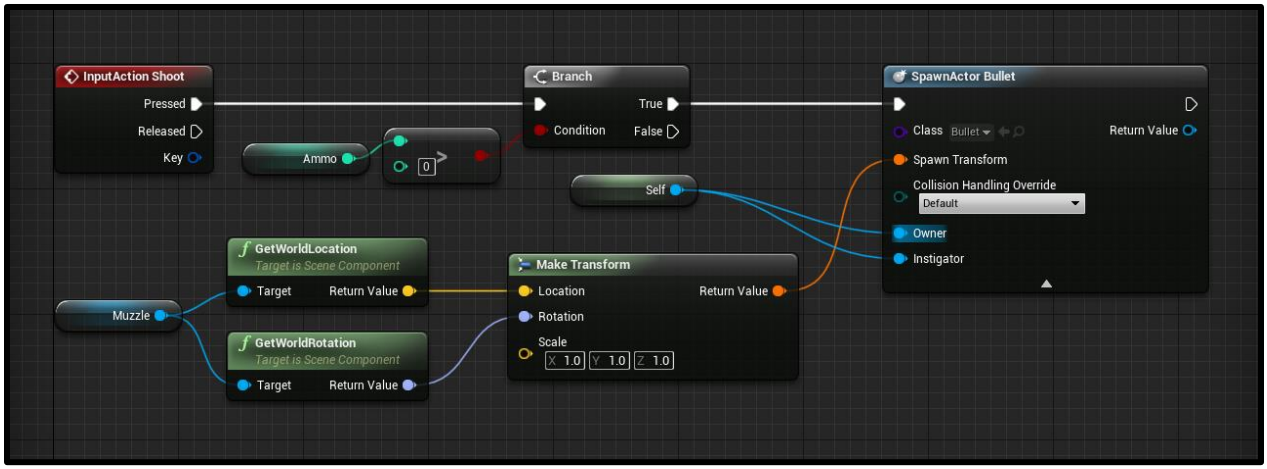
Back in our **Player** blueprint, let's implement shooting. First, we need to create an integer variable called **Ammo**. Compile, and set the default value to 20.



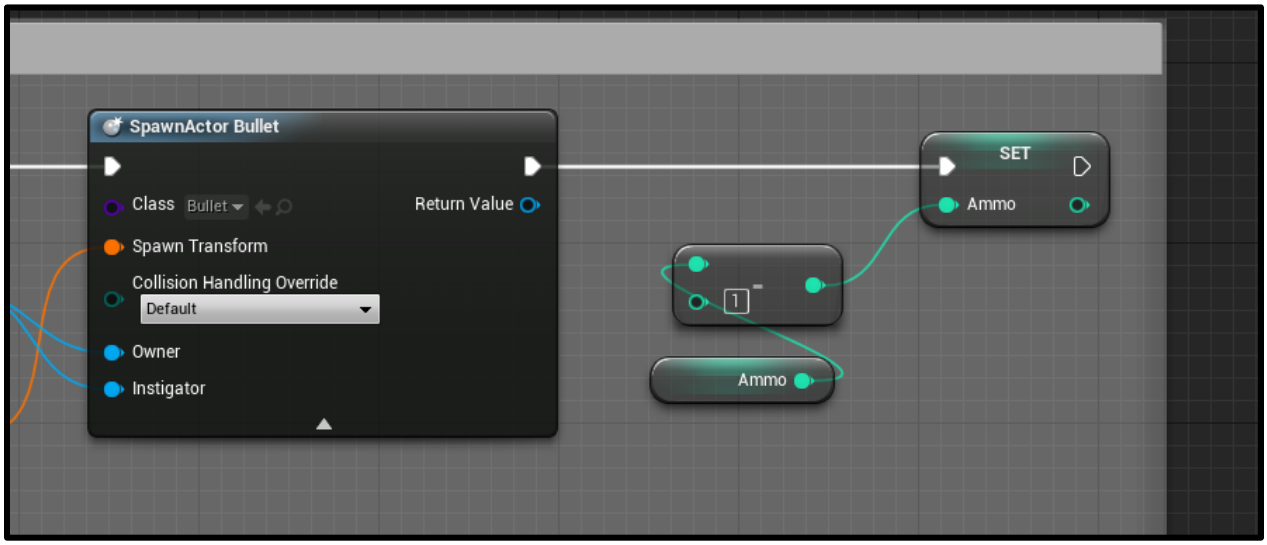
To begin, let's add in the **InputAction Shoot** node. This gets triggered when we press the left mouse button. Then we're going to check if we've got any ammo. If so, the **SpawnActor** node will create a new object. Set the *Class* to **Bullet**.



Next, we need to give the **SpawnActor** blueprint a spawn transform, owner and instigator. For the transform, we'll make a new one. The location is going to be the muzzle and the rotation is going to be the muzzle's rotation. For the owner and instigator, create a **Get a Reference to Self** node.



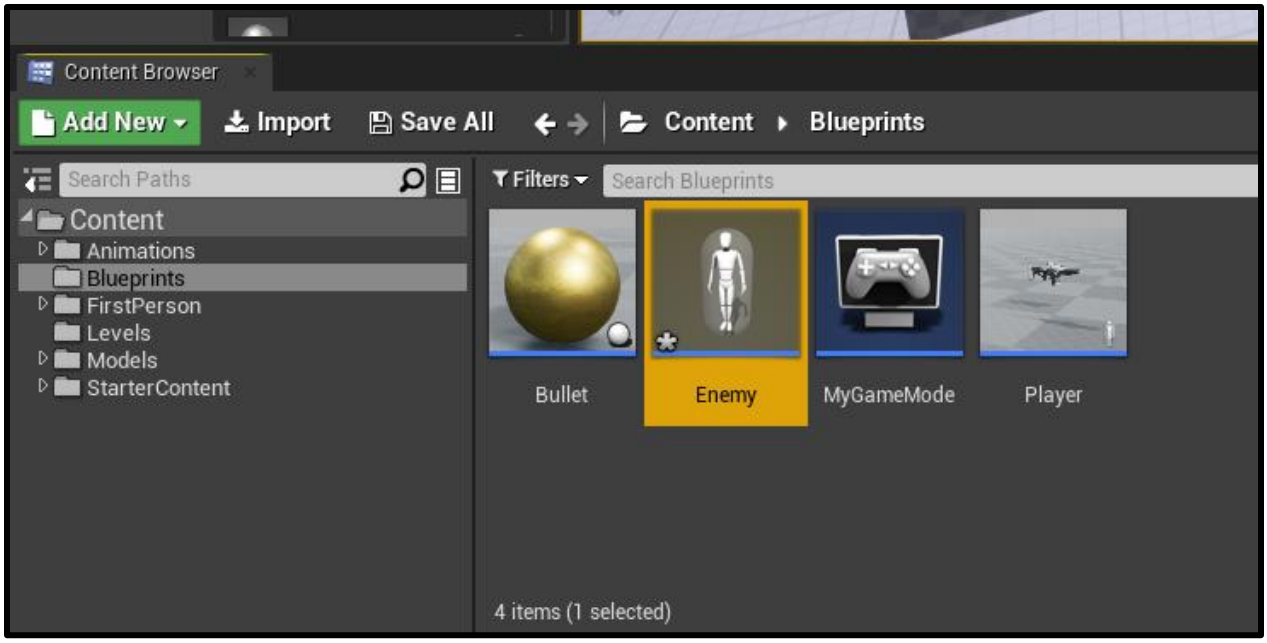
Finally, we can subtract 1 from our ammo count.



We can now press play and see that the gun can shoot!

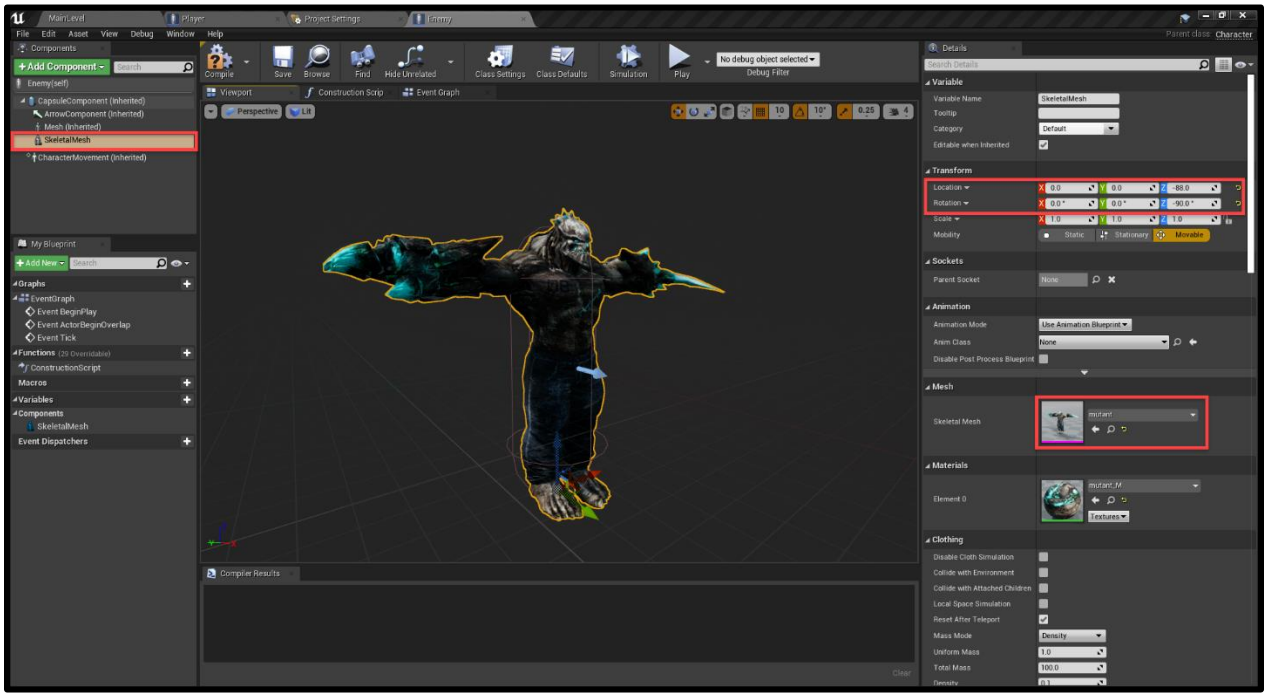
Creating the Enemy

Now it's time to create the enemy and its AI. Create a new blueprint with the parent class of **Character**. Call it **Enemy**.

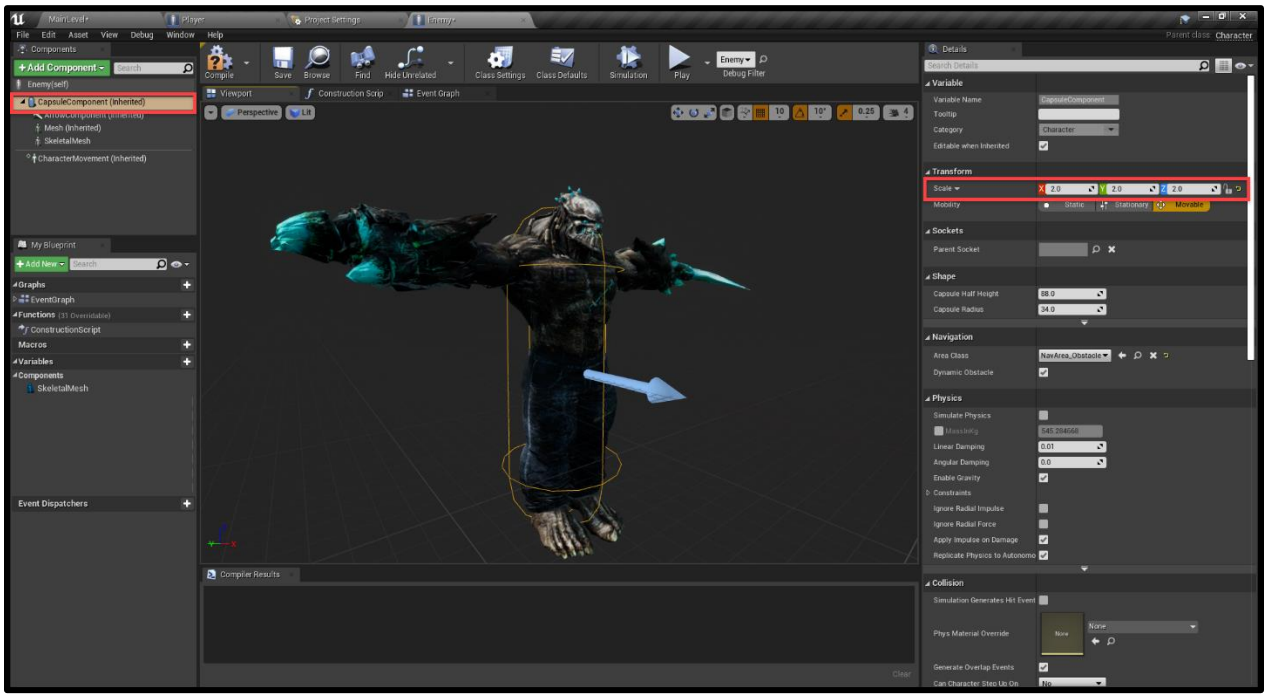


Inside of the blueprint, create a new **Skeletal Mesh** component.

- Set the **Location** to *0, 0, -88*
- Set the **Rotation** to *0, 0, -90*
- Set the **Skeletal Mesh** to *mutant*



Select the **CapsuleComponent** and set the **Scale** to *2*. This will make the enemy bigger than the player.

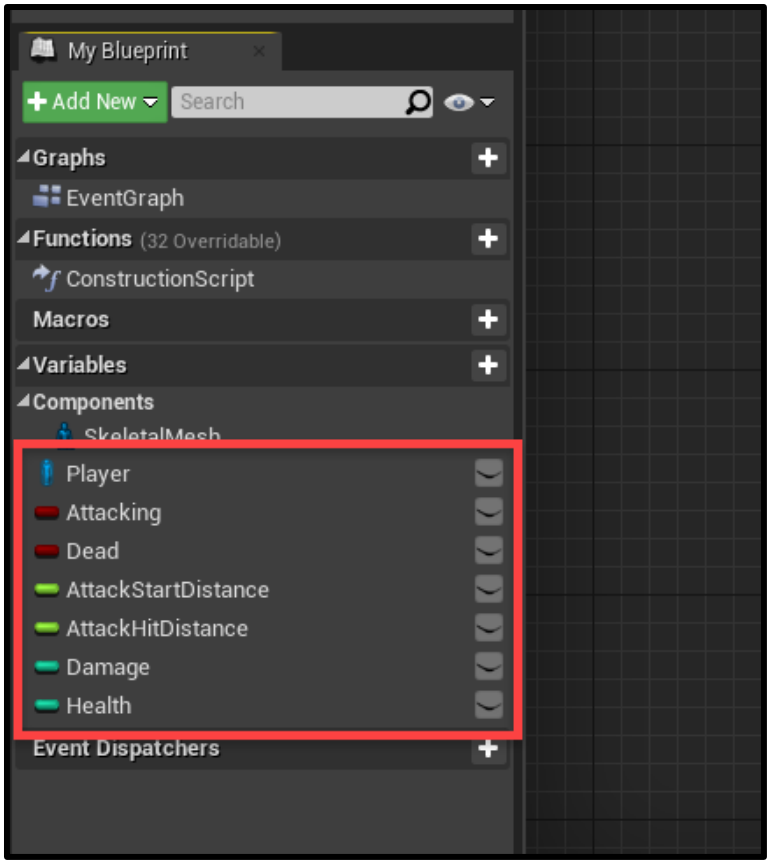


Next, let's go over to the **Event Graph** and create our variables.

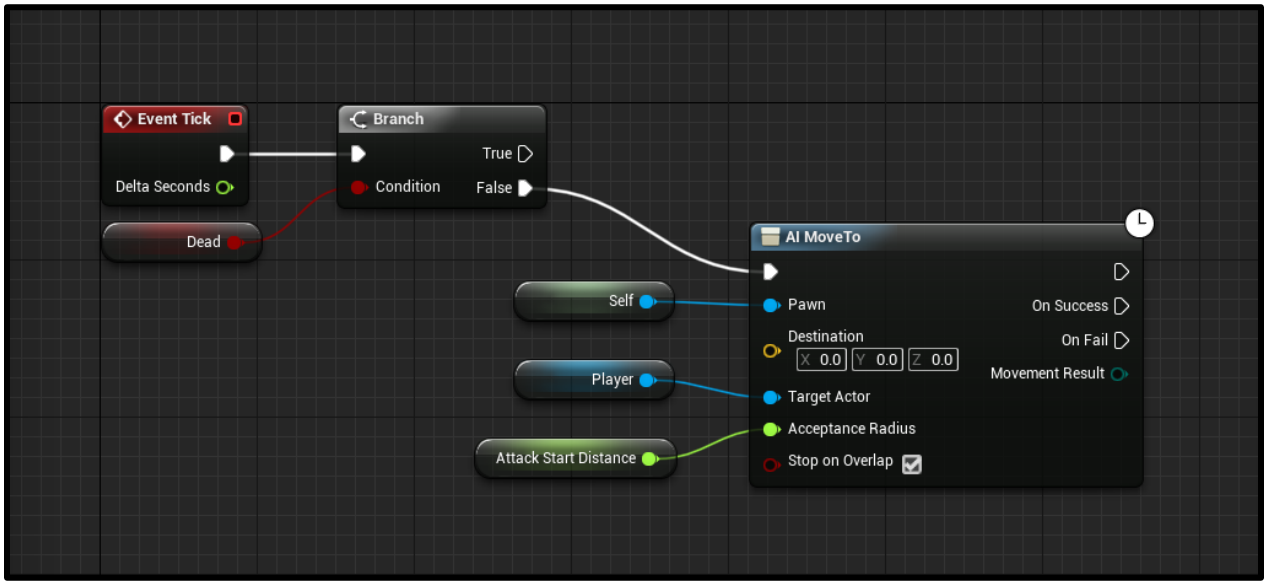
- Player (Player)
- Attacking (Boolean)
- Dead (Boolean)
- AttackStartDistance (Float)
- AttackHitDistance (Float)
- Damage (Integer)
- Health (Integer)

Hit the **Compile** button, then we can set some default values.

- AttackStartDistance = 300.0
- AttackHitDistance = 500.0
- Health = 10



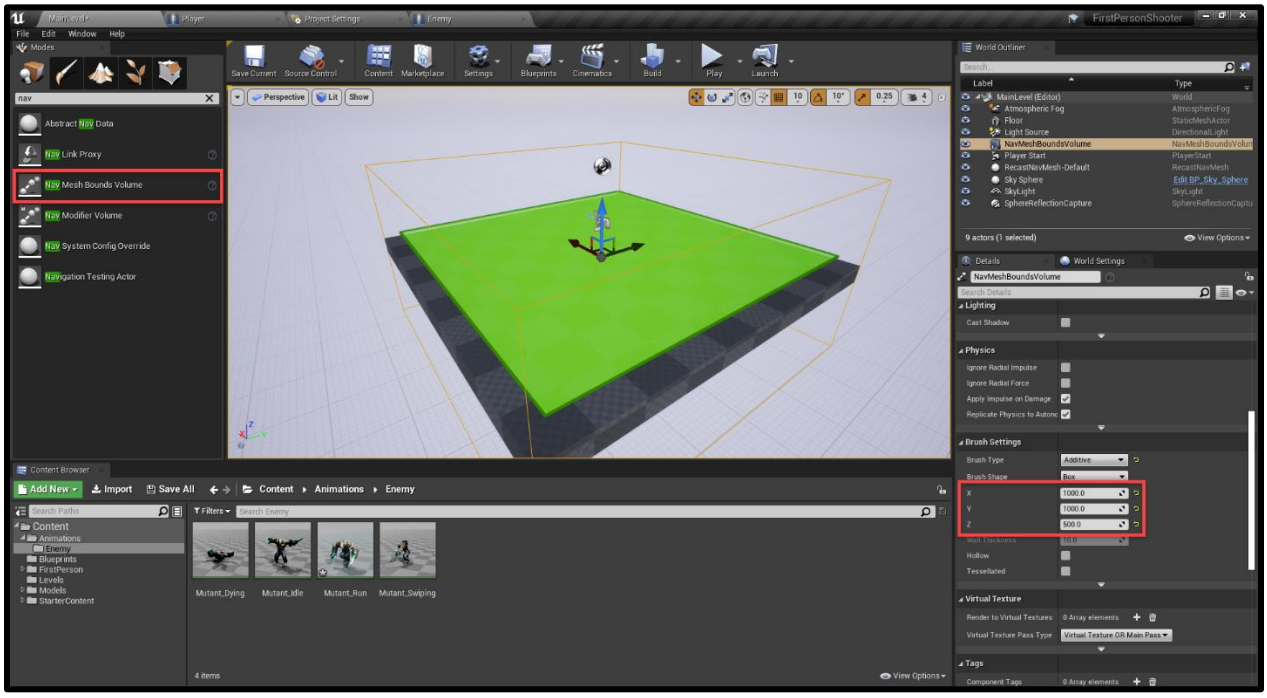
Let's get started. First, we want to be using the Event Tick node which triggers every frame. If we're not dead, then we can move towards the player. Fill in the properties as seen below.



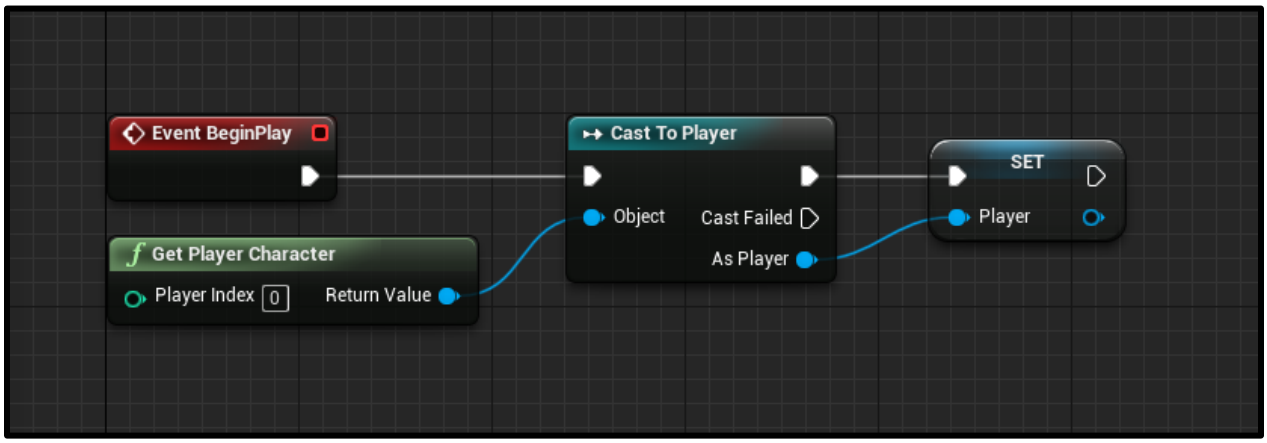
Back in the level editor, we need to generate a nav-mesh for the enemy to move along.

1. In the **Modes** panel, search for the *Nav Mesh Bounds Volume* and drag it into the scene
2. Set the **X** and **Y** size to *1000*
3. Set the **Z** size to *500*

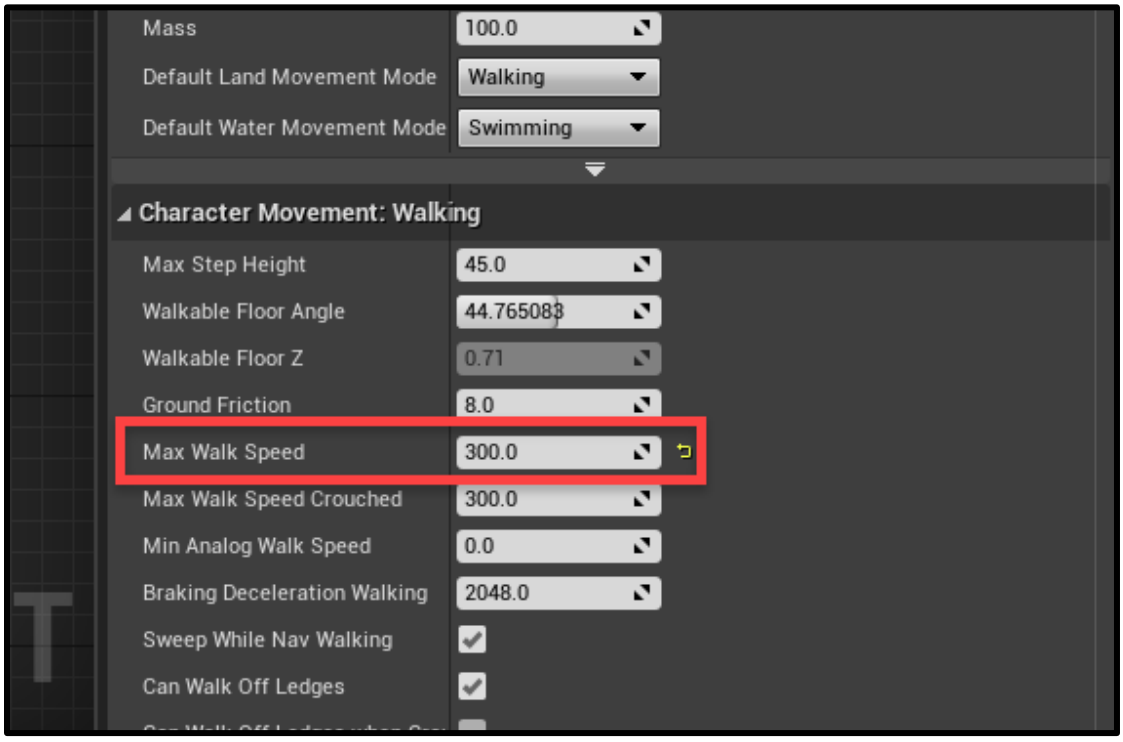
You can press **P** to toggle the nav-mesh visualizer on or off.



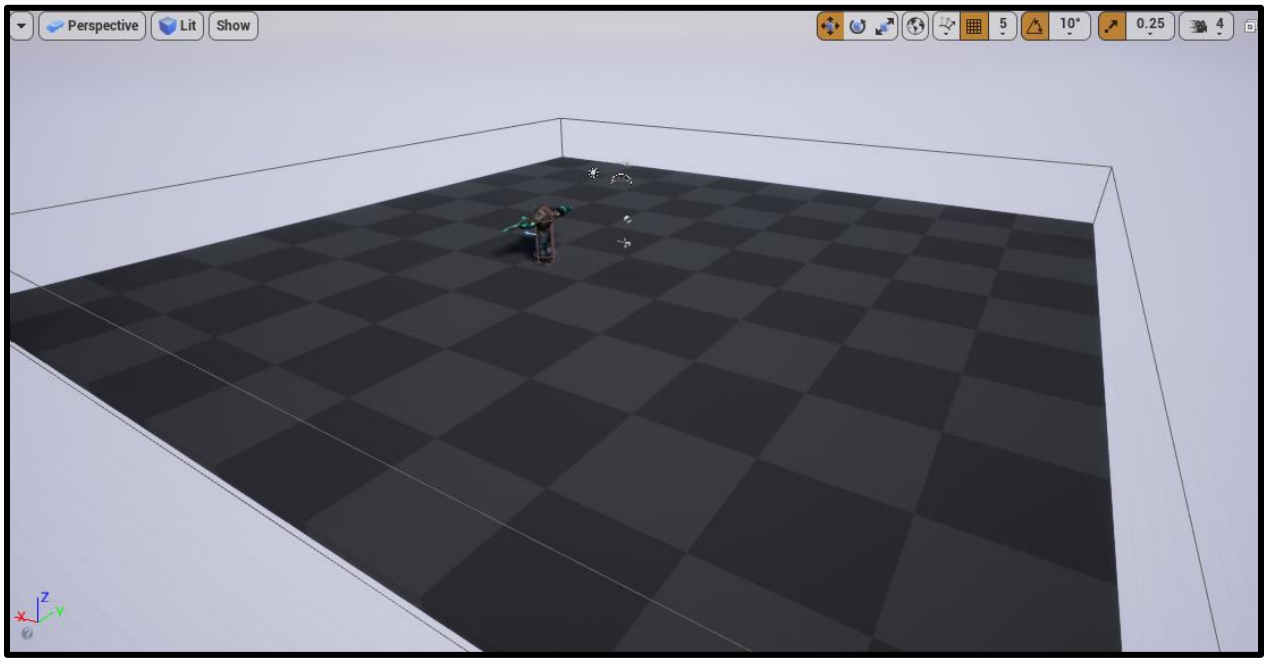
You can also increase the size of the platform and nav mesh bound. Back in the enemy blueprint, let's make it so when the game starts, we set our player variable.



Finally, let's select the **CharacterMovement** component and change the **Max Walk Speed** to *300*.



Back in the level editor, let's increase the size of the ground and nav mesh bounds.

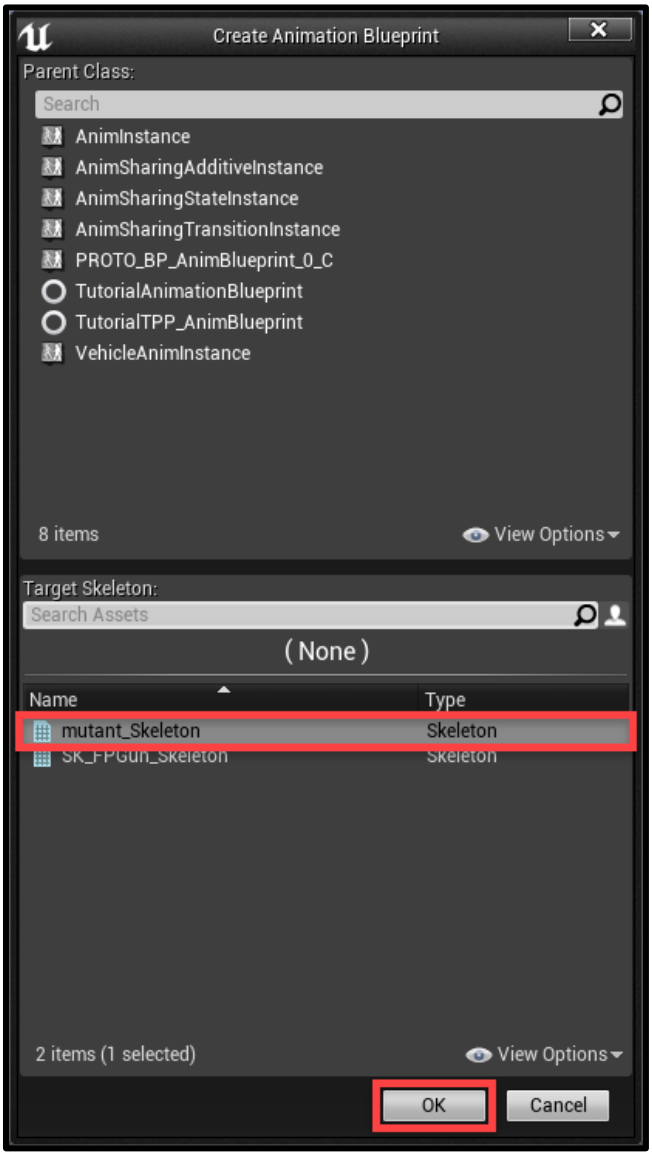


Let's set some default values for the variables.

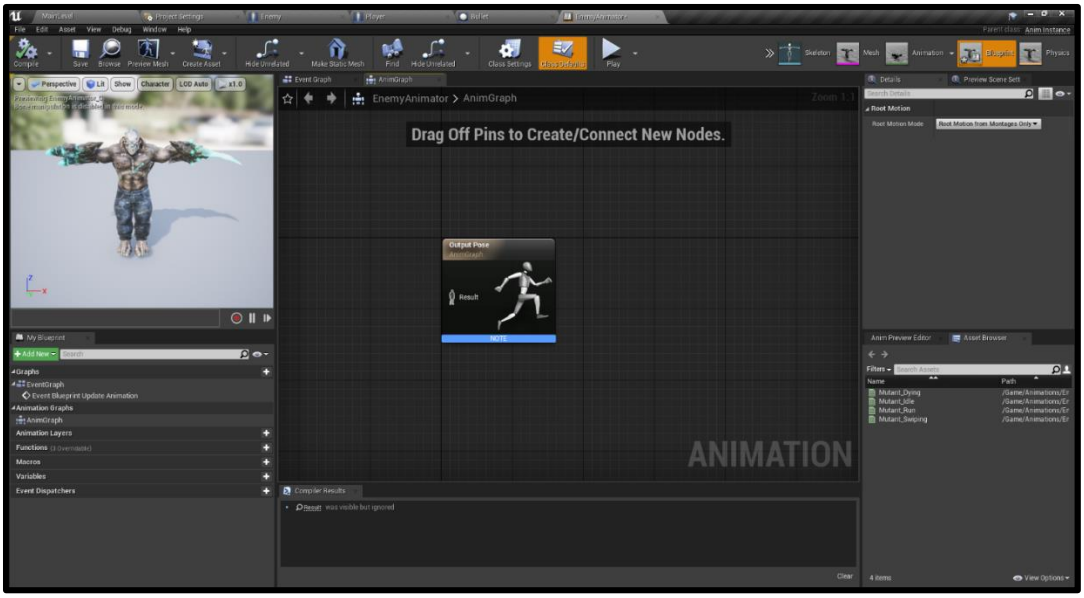
- AttackStartDistance = 300.0
- AttackHitDistance = 500.0
- Damage = 1
- Health = 10

Enemy Animations

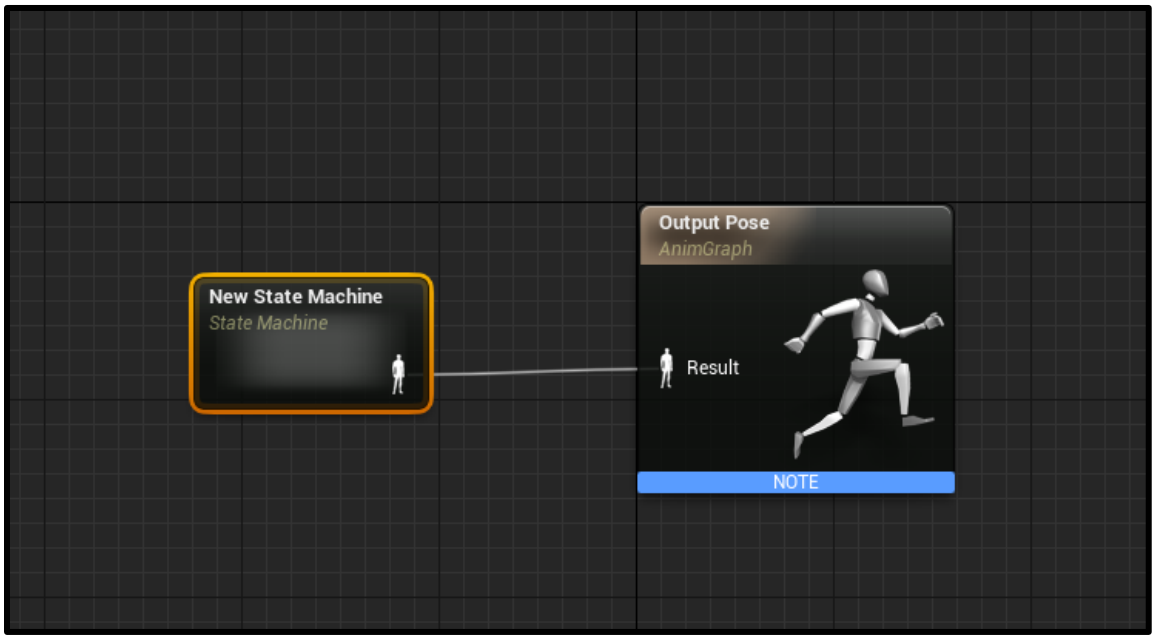
Next, is to create the enemy animations and have those working. In the *Blueprints* folder, right click, select *Animation* > *Animation Blueprint*. Select the *mutant* for our skeleton, then click **Ok**. Call this blueprint: **EnemyAnimator**.



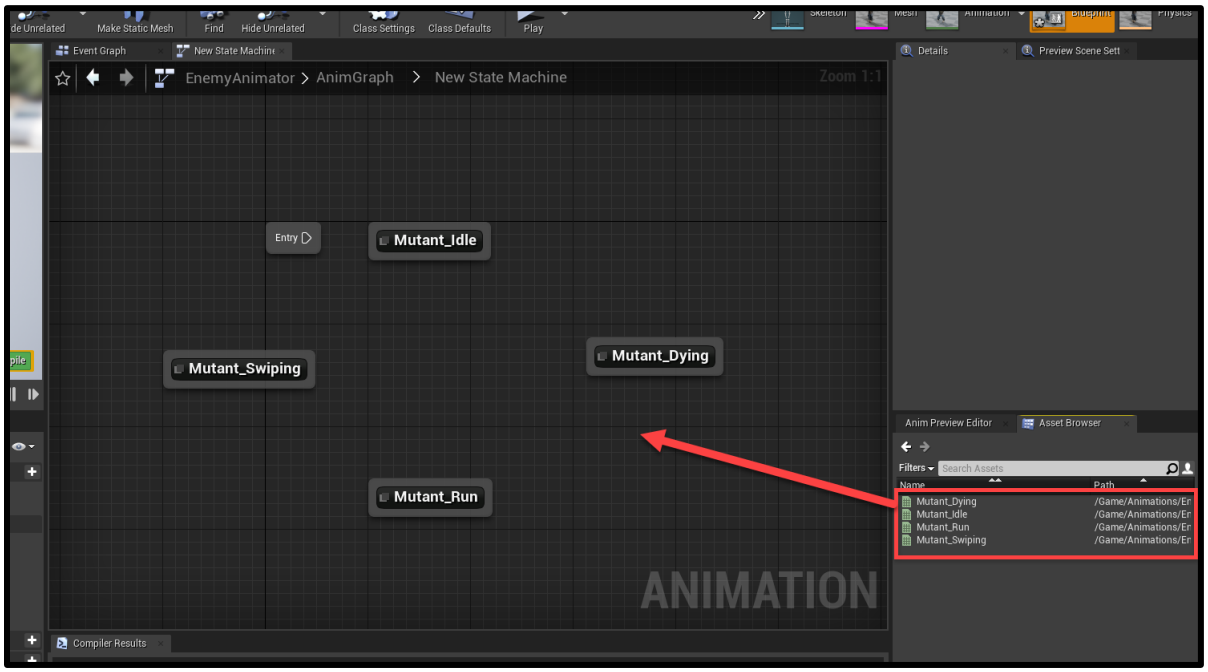
Double-clicking on it will open up the animation editor. On the top-left, we have a preview of our skeleton. At the bottom-left, we have a list of our 4 animations. You can open them up to see a preview in action.



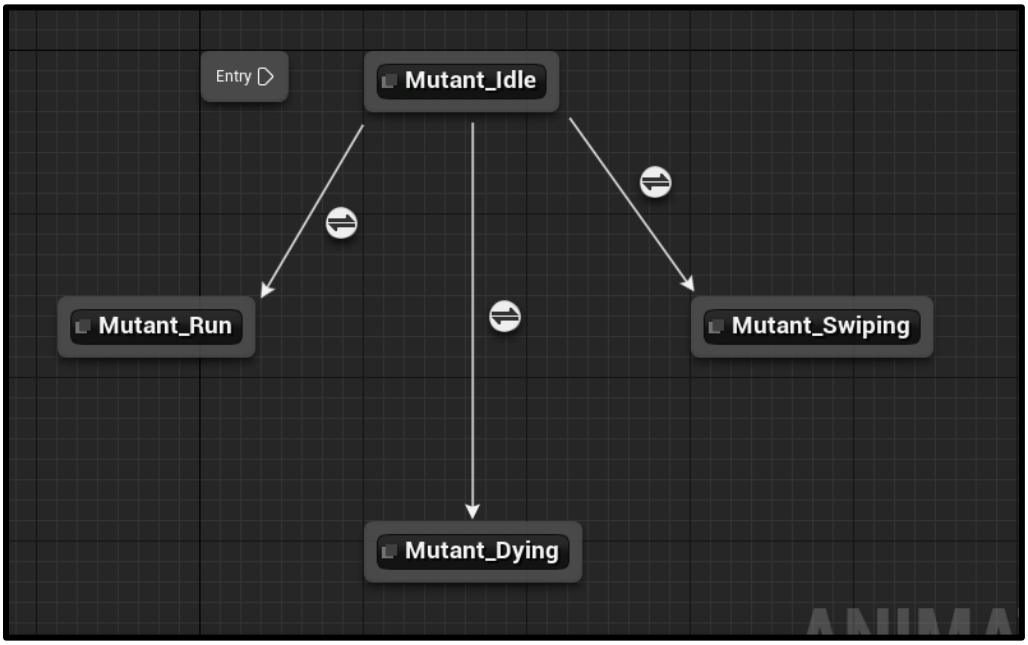
What we want to do now, is in the center graph, right click and create a new *state machine*. A state machine is basically a collection of logic to determine what animation we currently need to play.



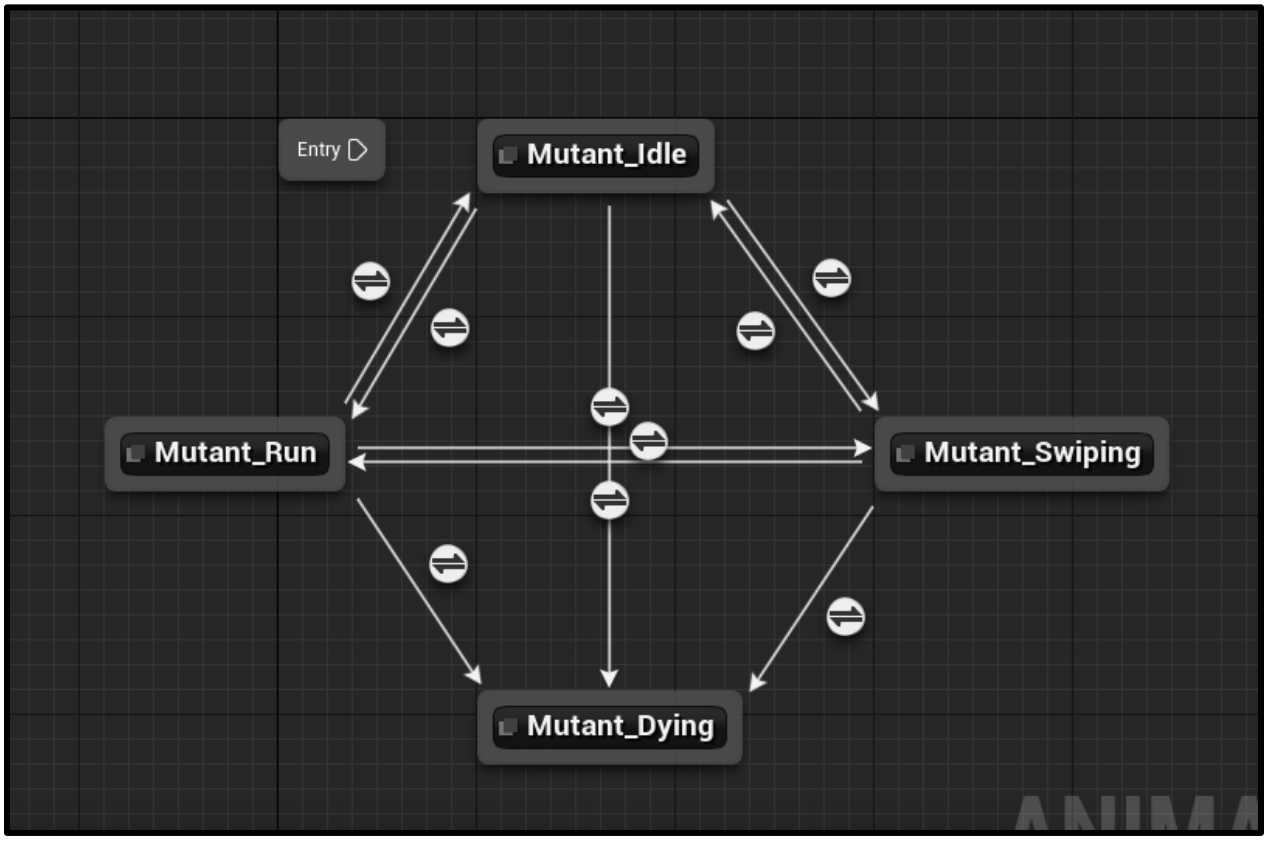
You can then double-click on the state machine to go inside of it. This is where we're going to connect our animations so that the animator can move between them in-game. Begin by dragging in each of the 4 animations.



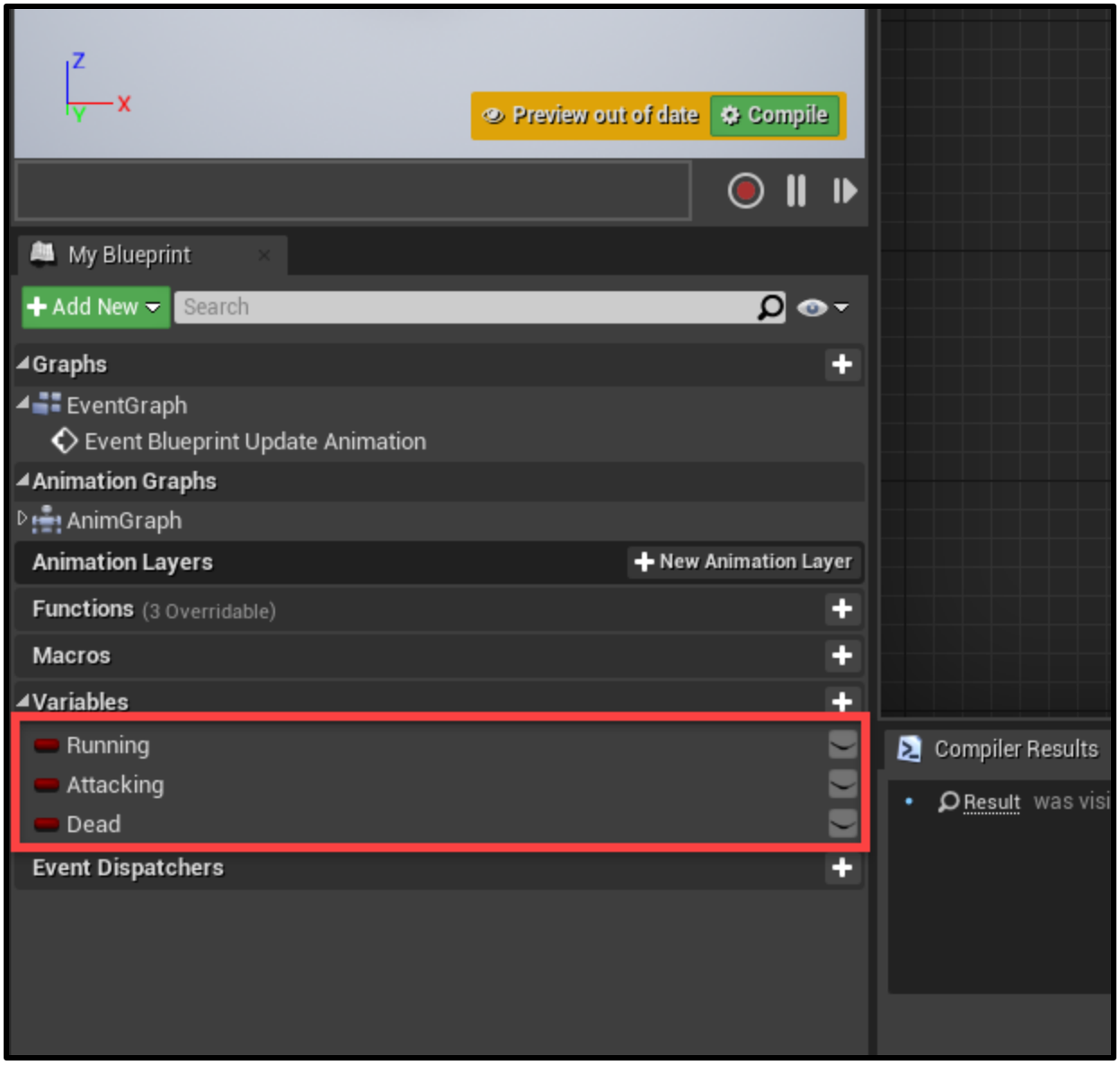
Then we need to think... If we're idle, what animations can we transition to? All of them, so on the idle state, click and drag on the border to each of the three others.



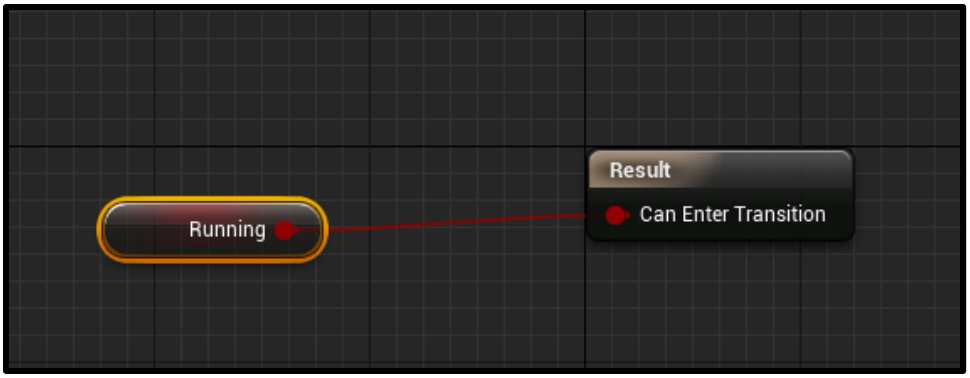
We then need to do the same with the other states. Make the connections look like this:



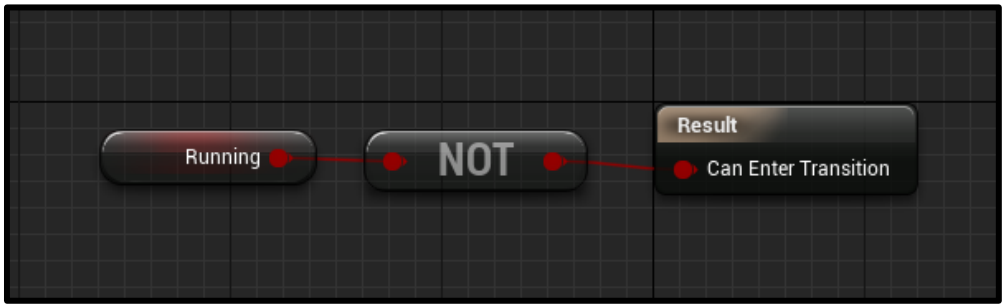
In order to apply logic, we need to create three new Boolean variables. Running, Attacking and Dead.



You'll see that next to the state transition lines is a circle button. Double-click on the one for idle -> run. This will open up a new screen where we can define the logic of moving along this transition. Drag in the **Running** variable and plug that into the *Can Enter Transform* node. So basically if running is true, then the transition will happen.

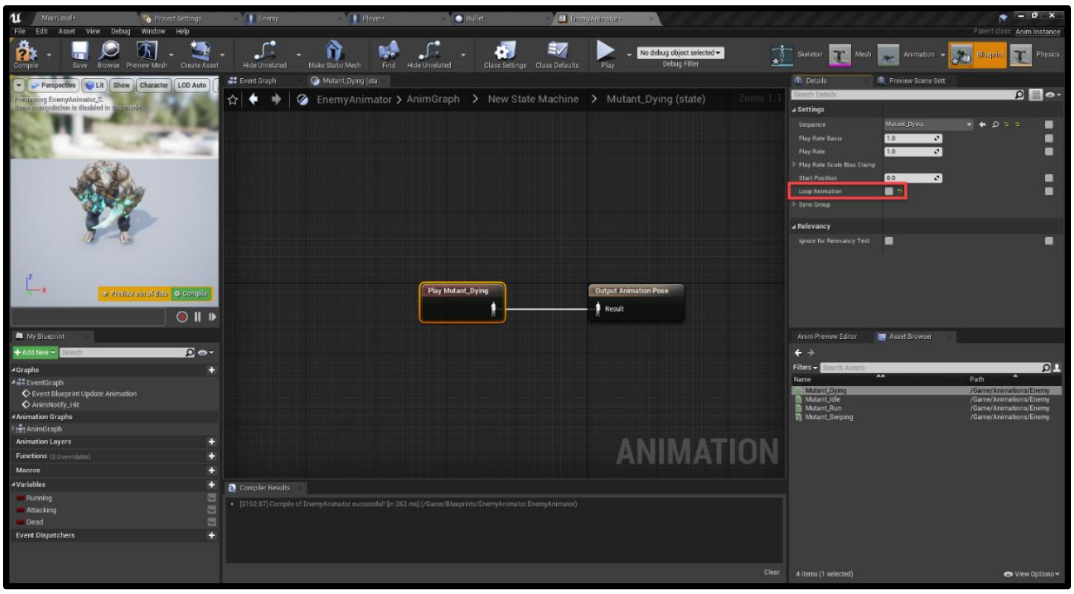


Back in the state machine open the run -> idle transition. Here, we want to do the opposite.

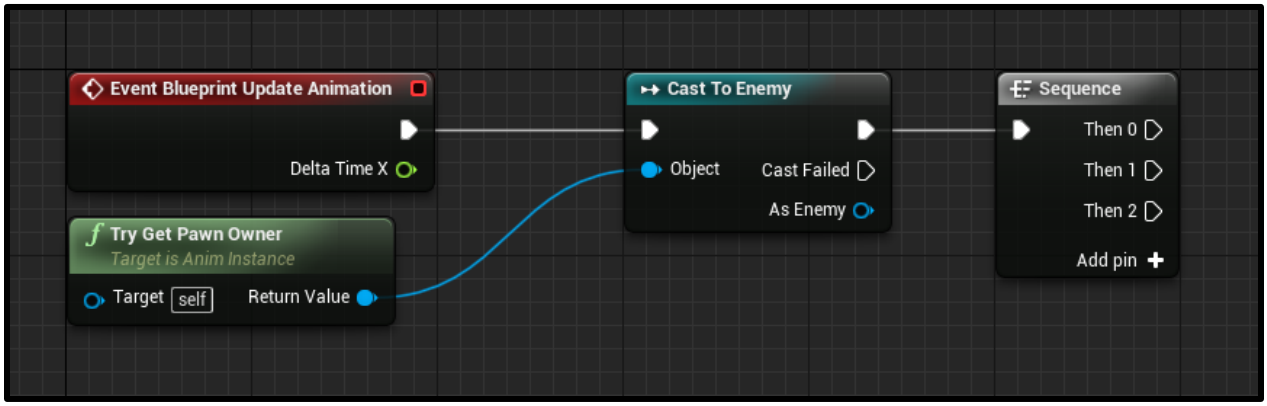


Go through and fill in the respective booleans for all of the transitions.

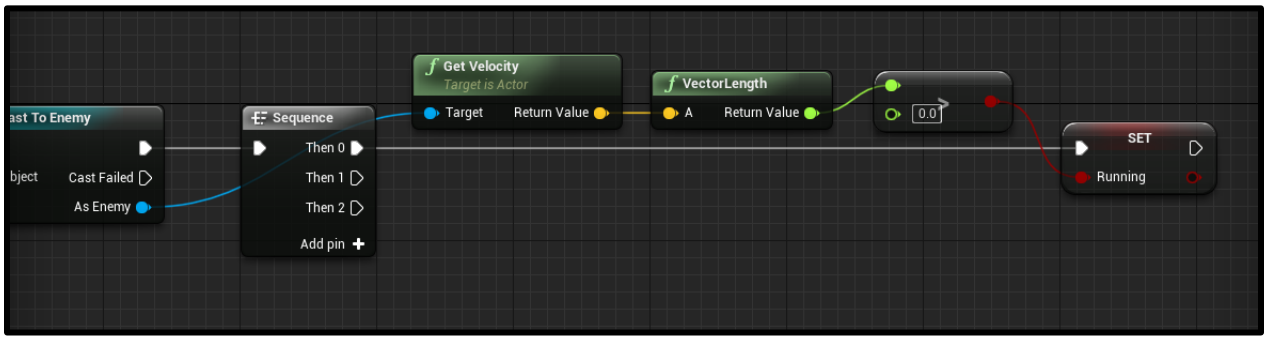
While we're at it, let's double click on the **Mutant_Dying** state, select the state and disable **Loop Animation** so it only plays once.



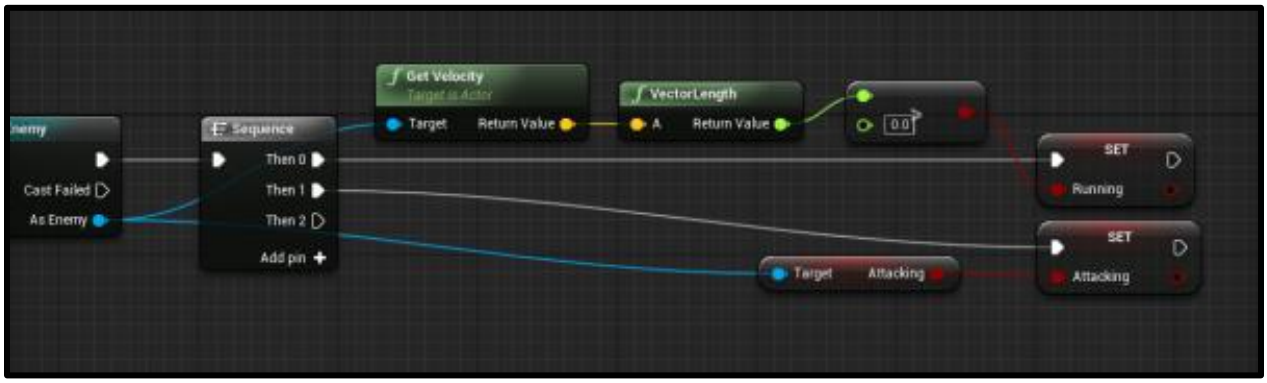
At the top of the state screen is an **Event Graph** tab. Click that to go to the graph where we can hook our three variables up to the enemy blueprint. First, we need to cast the owner to an enemy, then connect that to a **Sequence** node.



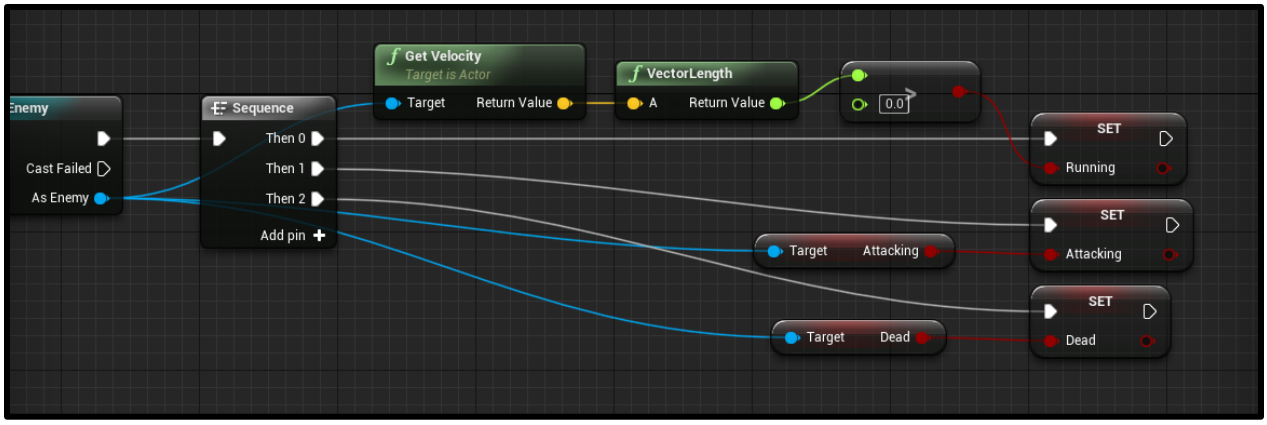
To set the running variable, we're going to be checking the enemy's velocity.



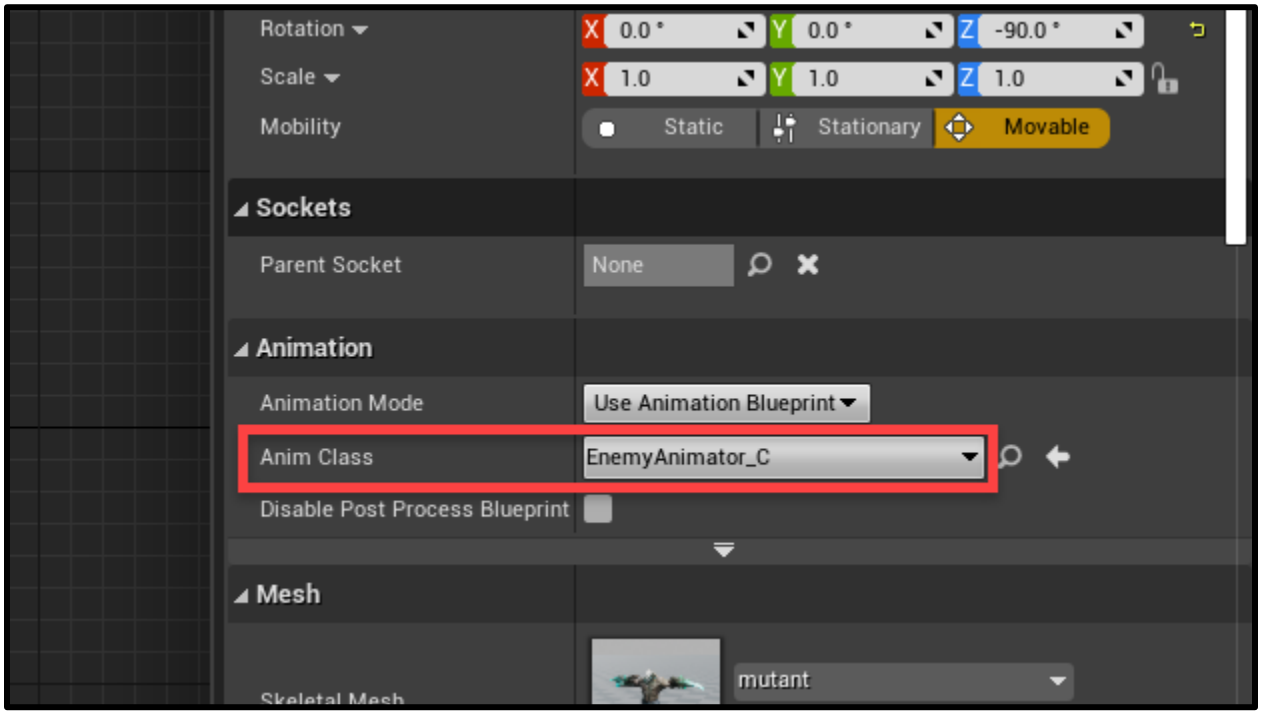
For attacking, we just want to get the enemy's attacking variable.



The same for dead.



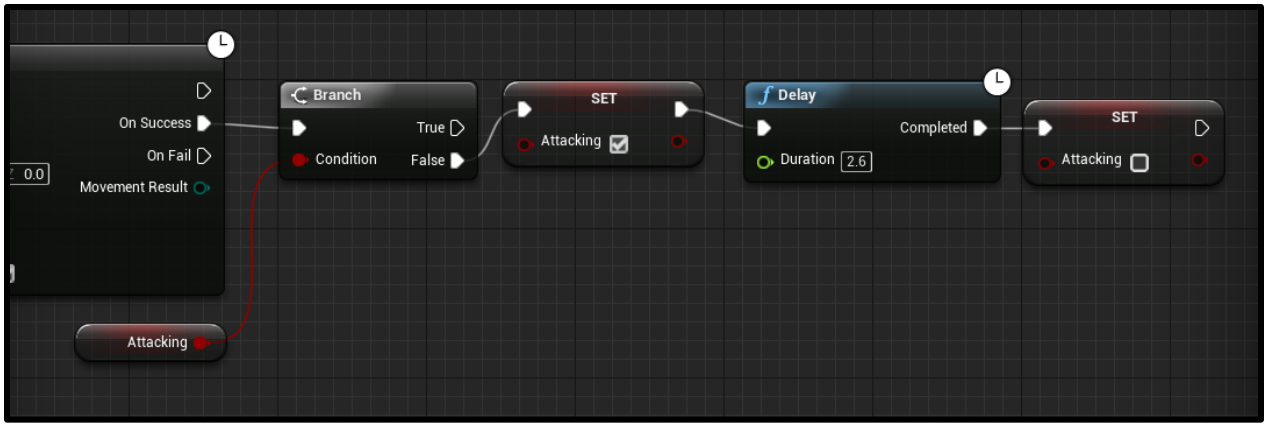
Back in the **Enemy** blueprint, select the *SkeletalMesh* component and set the **Anim Class** to be our new *EnemyAnimator*.



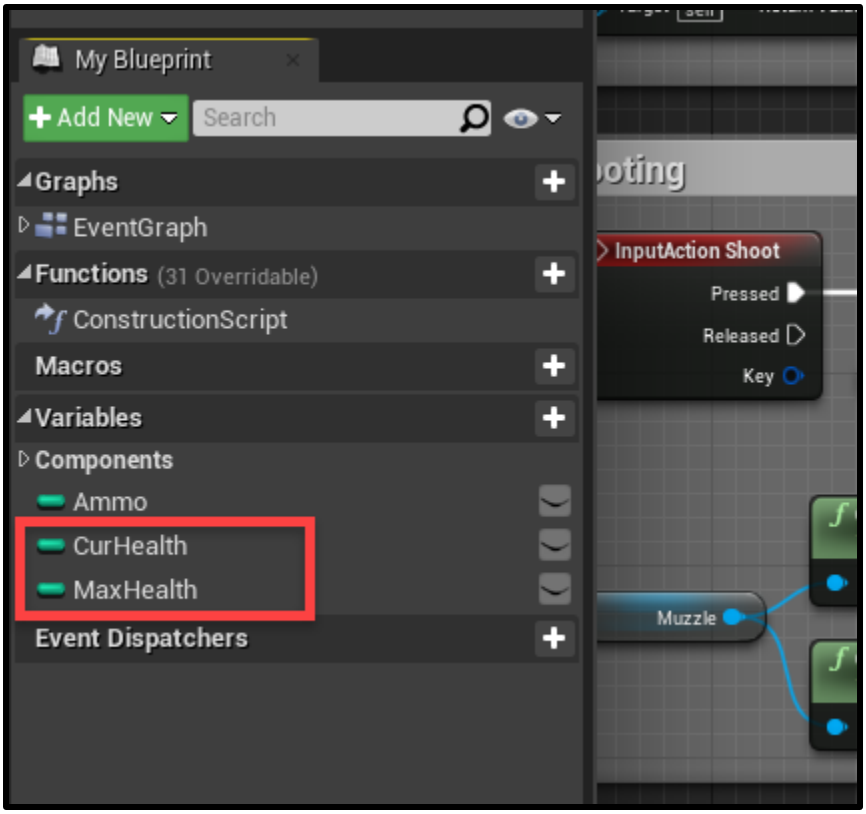
Press play and see the results!

Attacking the Player

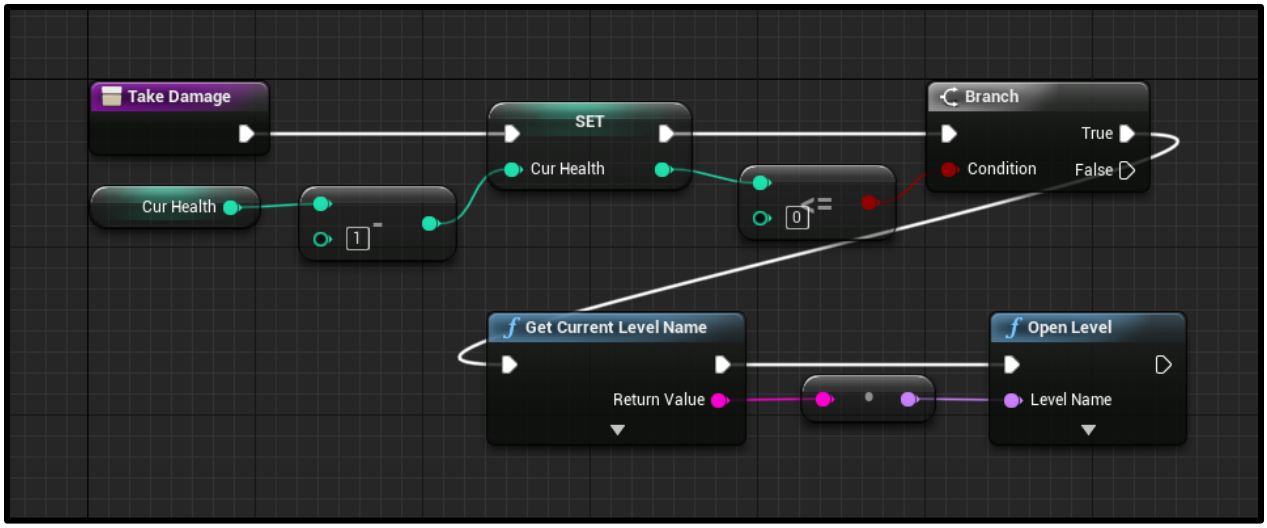
In the **Enemy** blueprint, let's continue the flow from the *AIMoveTo* component's On Success output. We basically want to check if we're not current attacking. If so, then set attacking to true, wait 2.6 seconds (attack animation duration), then set attacking to false.



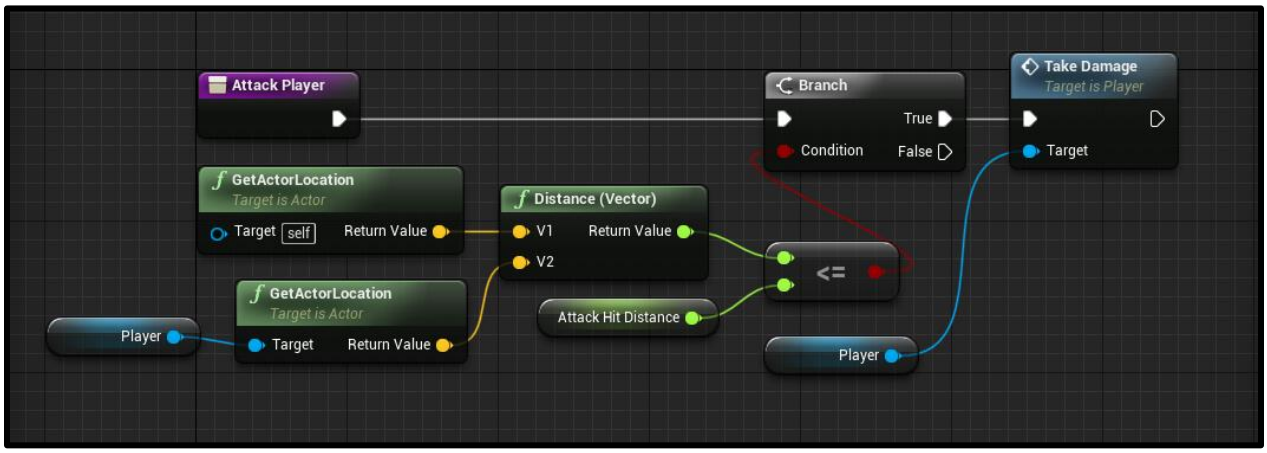
If you press play, you should see that the enemy runs after you, then attacks. For it to do damage though, we'll first need to go over to the **Player** blueprint and create two new variables. Compile and set both default values to *10*.



Next, create a new function called **TakeDamage**.

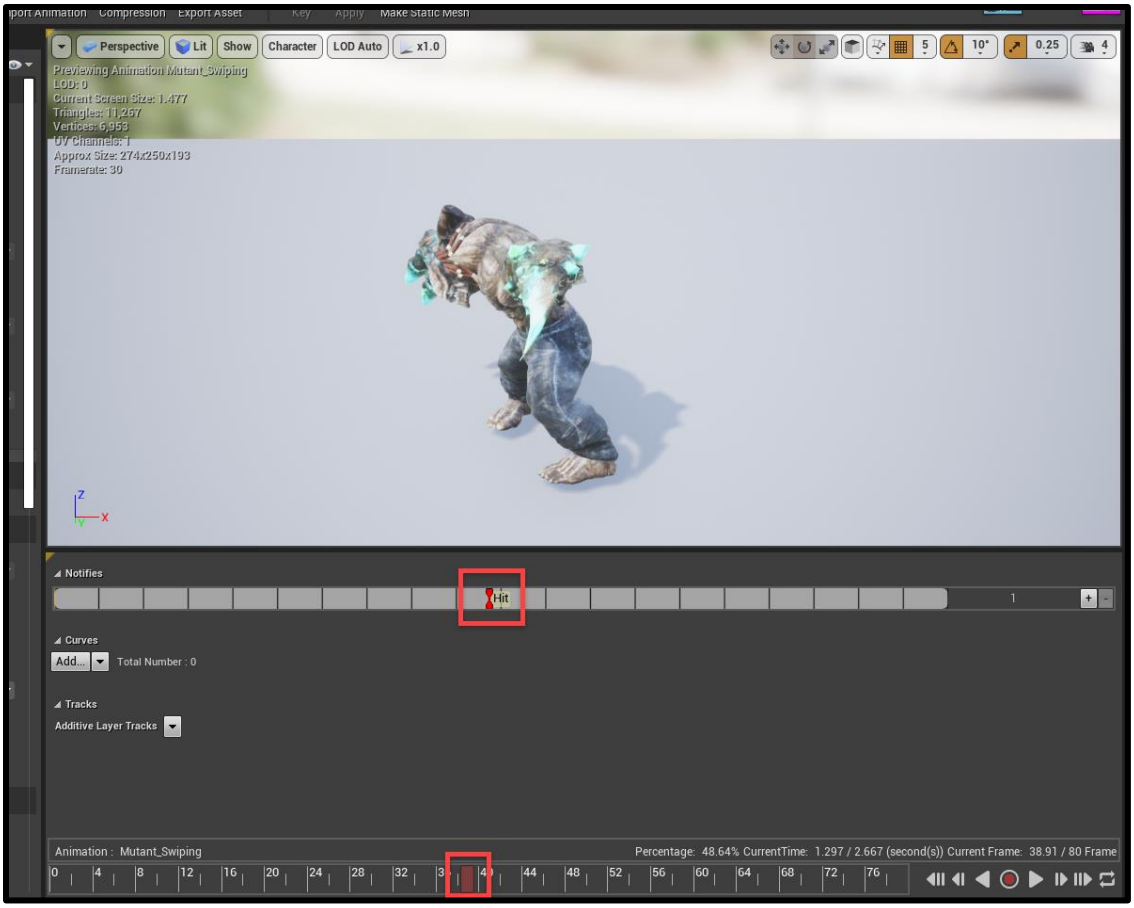


Back in the **Enemy** blueprint, create a new function called **AttackPlayer**.

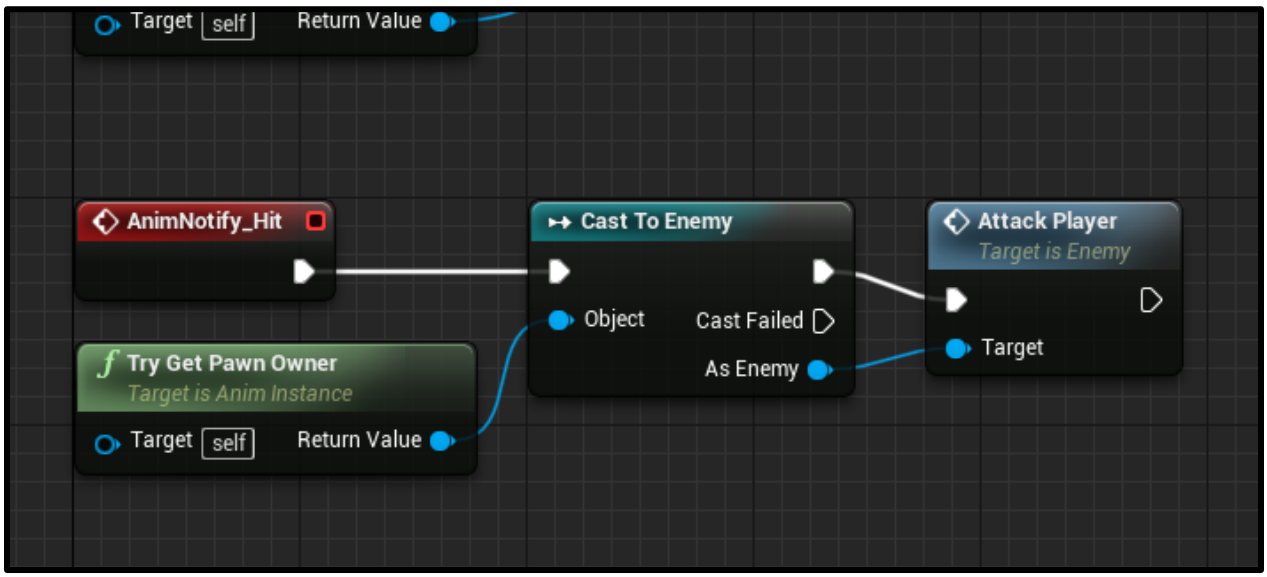


To trigger this function, we need to go to the enemy animation editor and double-click on the swiping animation in the bottom-left. Here, we want to drag the playhead (bottom of the screen) to the frame where we want to attack the player. Right click and select *Add Notify... > New Notify...*

Call it **Hit**.



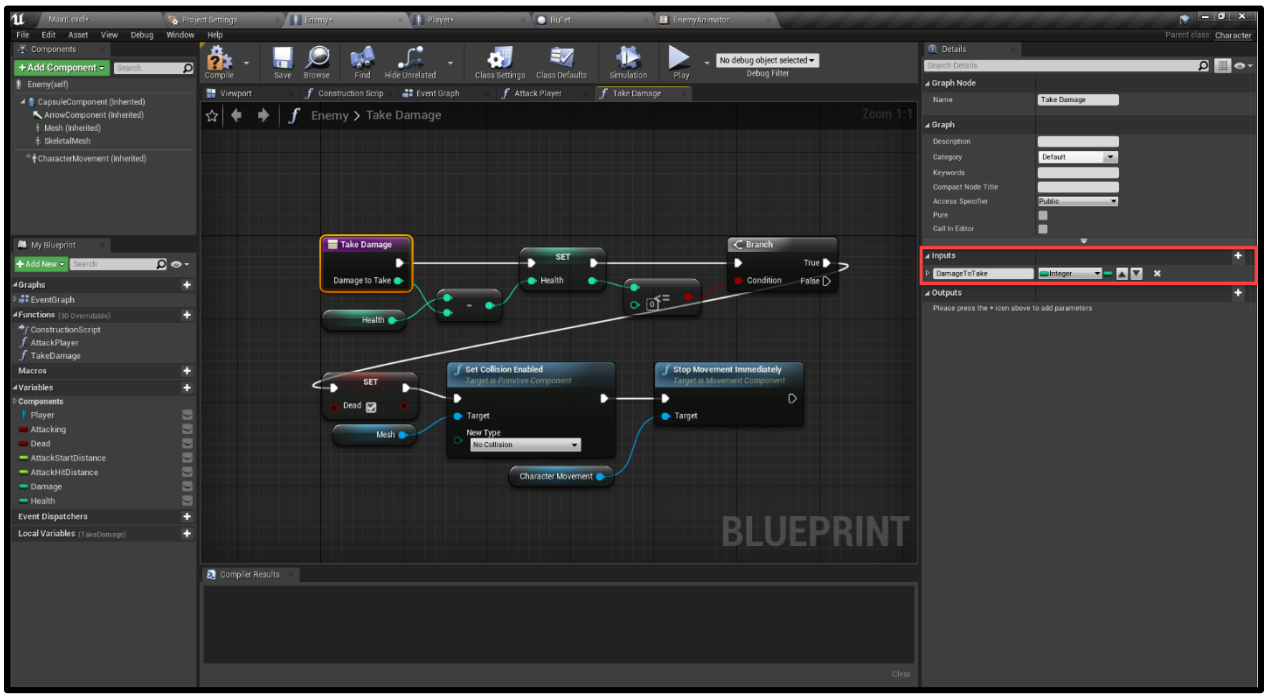
Then back in the animator event graph, we can create the **AnimNotify_Hit** event. This gets triggered by that notify.



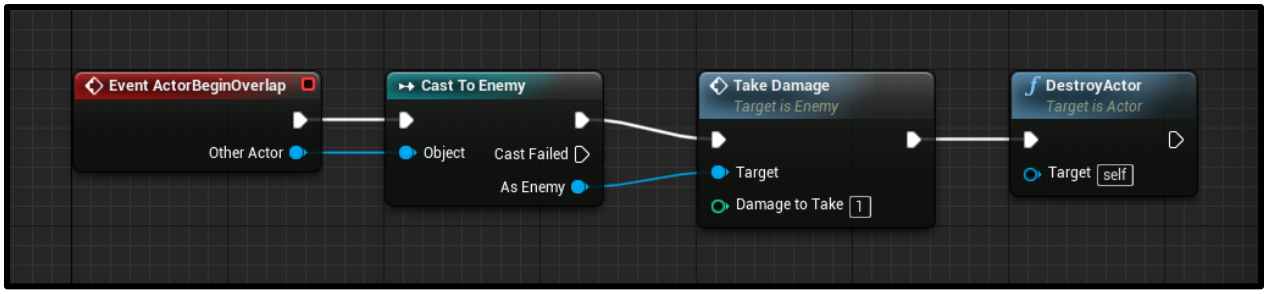
Now you can press play and see that after 10 hits, the level will reset.

Shooting the Enemy

Finally, we can implement the ability for the player to kill the enemy. In the Enemy blueprint, create a new function called **TakeDamage**. Give it an input of **DamageToTake** as an integer.



Then in the **Bullet** blueprint, we can check for a collider overlap and deal damage to the enemy.



If you press play, you should be able to shoot the enemy and after a few bullets, they should die.

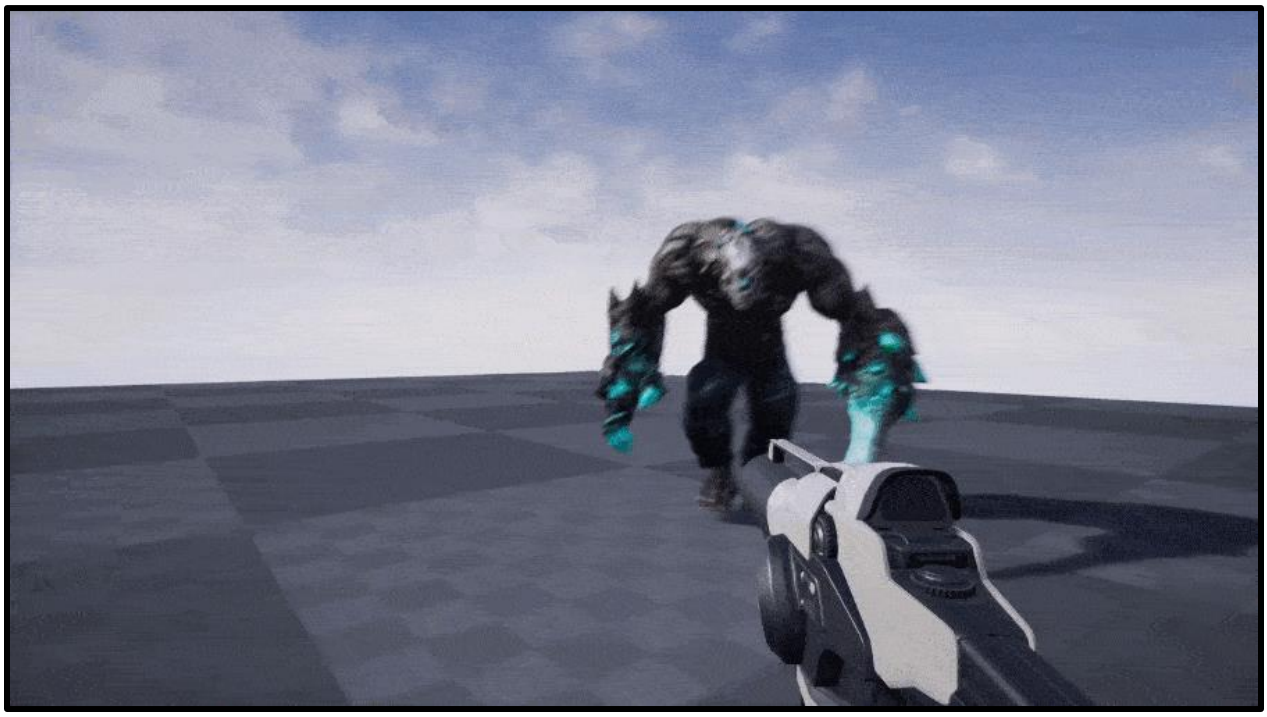
Conclusion

And there we are!

Thank you for following along with the tutorial! You should now have a functional first-person shooter with a player controller and enemy AI - complete with animations and dynamic state machine! Not only will your players be able to dash about the stage jumping and shooting at their leisure, but the health mechanics create a challenge to stay alive. You've accomplished this and more with just Unreal Engine's blueprinting system as well, learning how to control nodes to provide a variety of functionality.

From here, you can expand the game or create an entirely new one. Perhaps you want to add more enemies, or create an exciting 3D level where enemies might be just around that shadow-y corner! The sky is the limit, but with these foundations you have all you need to get started with robust FPS creation in Unreal Engine.

We wish you luck with your future games!



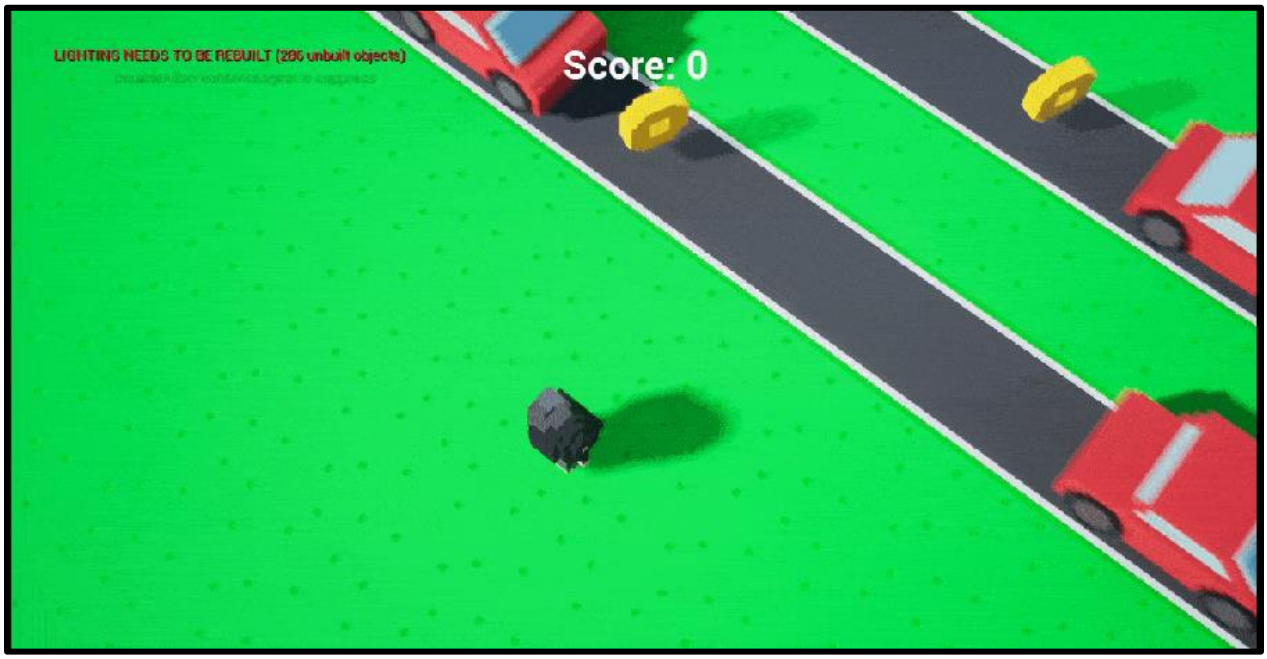
Creating an Arcade-Style Game in the Unreal Engine

Introduction

Complex games can be a lot of fun, but sometimes all players want to do is sit back and relive the retro days where simple mechanics and goals were the reigning champs. As a developer, not only might you want to create your own retro, arcade-style types of games (with some modern flair), but it can also be a great beginner's project to start your game development skills!

As such, in this tutorial, we'll be creating a road-crossing arcade game in the Unreal Engine. The game will feature a player controller, coins, cars, and an end goal - capturing the sorts of straight-forward mechanics you'd expect. Additionally, we'll be working heavily with Unreal Engine's unique Blueprinting system, so no coding from scratch is necessary!

If this is your first time using the Unreal Engine, we recommend you follow our intro to Unreal Engine tutorial first, as this one won't be going over the basics of the engine itself.



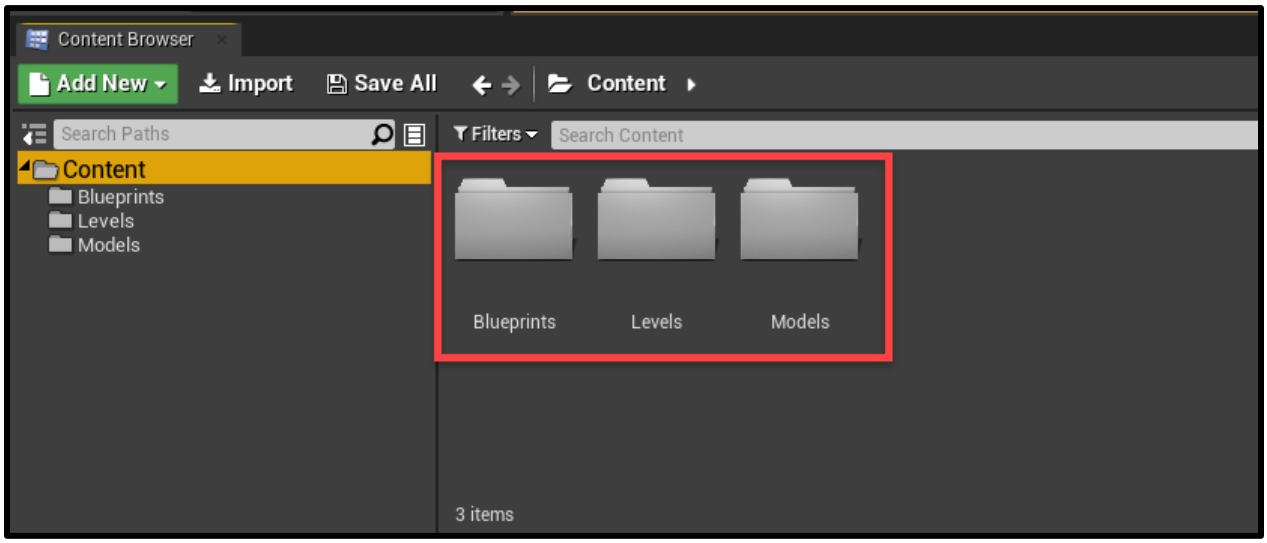
Project Files

For this project, we'll be needing some 3D models made in MagicaVoxel. You can also download the complete project with all the blueprints, levels, settings and models here.

Creating the Project

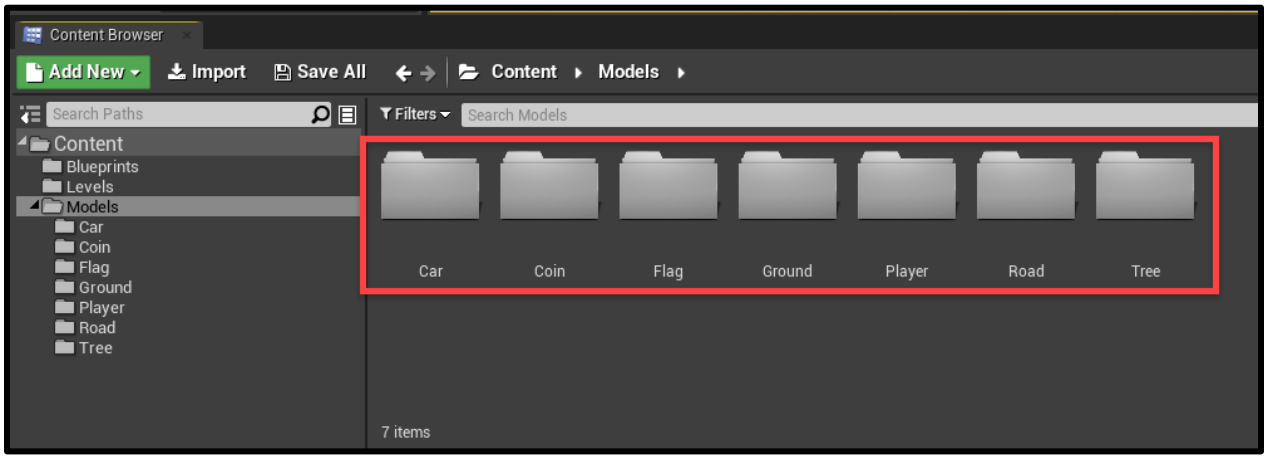
To begin, create a new project (no starter content needed). Then we're going to create three new folders in the **Content Browser**.

- Blueprints
- Levels
- Models



Next, save the open level to the *Levels* folder as **MainLevel**.

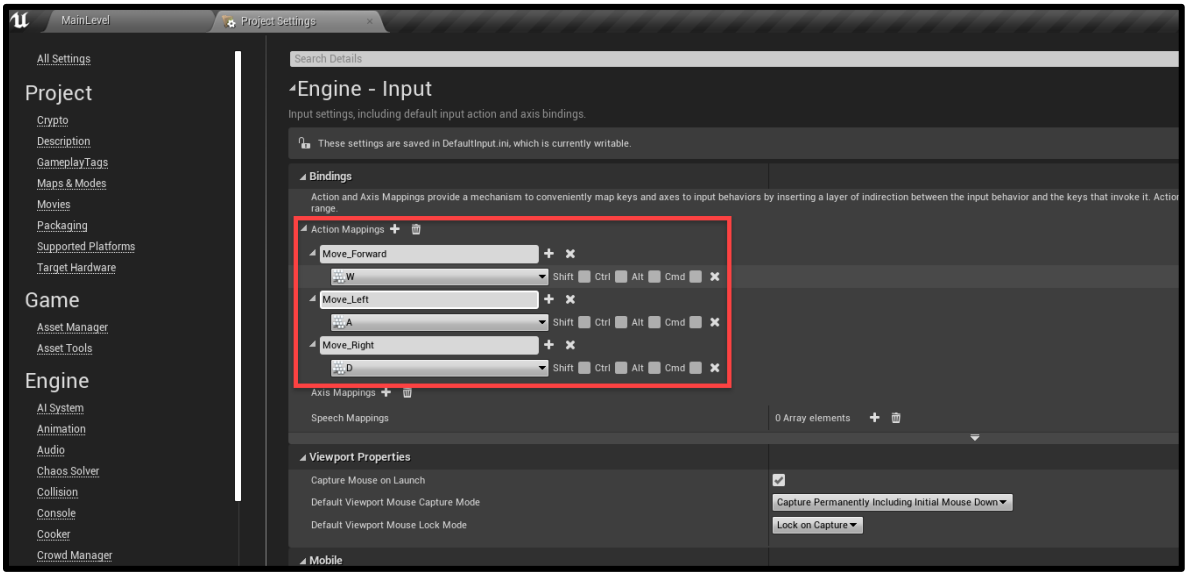
We'll be using some pre-made 3D models from MagicaVoxel, so download the included assets (top of the tutorial) and drag the contents inside of the *Models* folder, into the **Content Browser** in our *Models* folder. Like so:



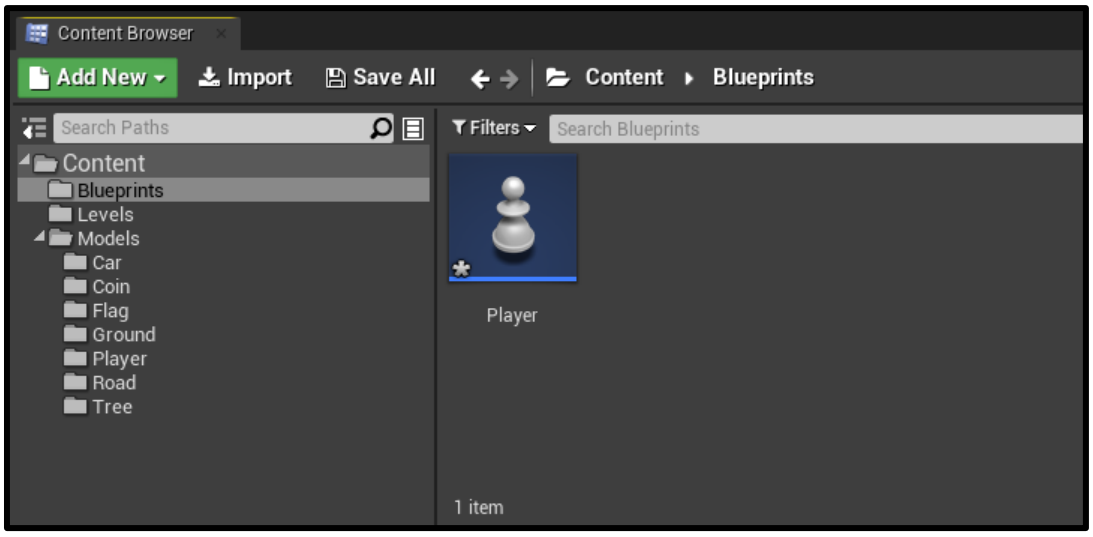
Creating the Player

Now we can create our player. This player will be able to move forwards, left and right. To begin, we need to setup our key bindings. Go to the **Project Settings** window (*Edit* > *Project Settings*...) and navigate to the **Input** tab. We want to create three new action mappings.

- Move_Forward = W
- Move_Left = A
- Move_Right = D



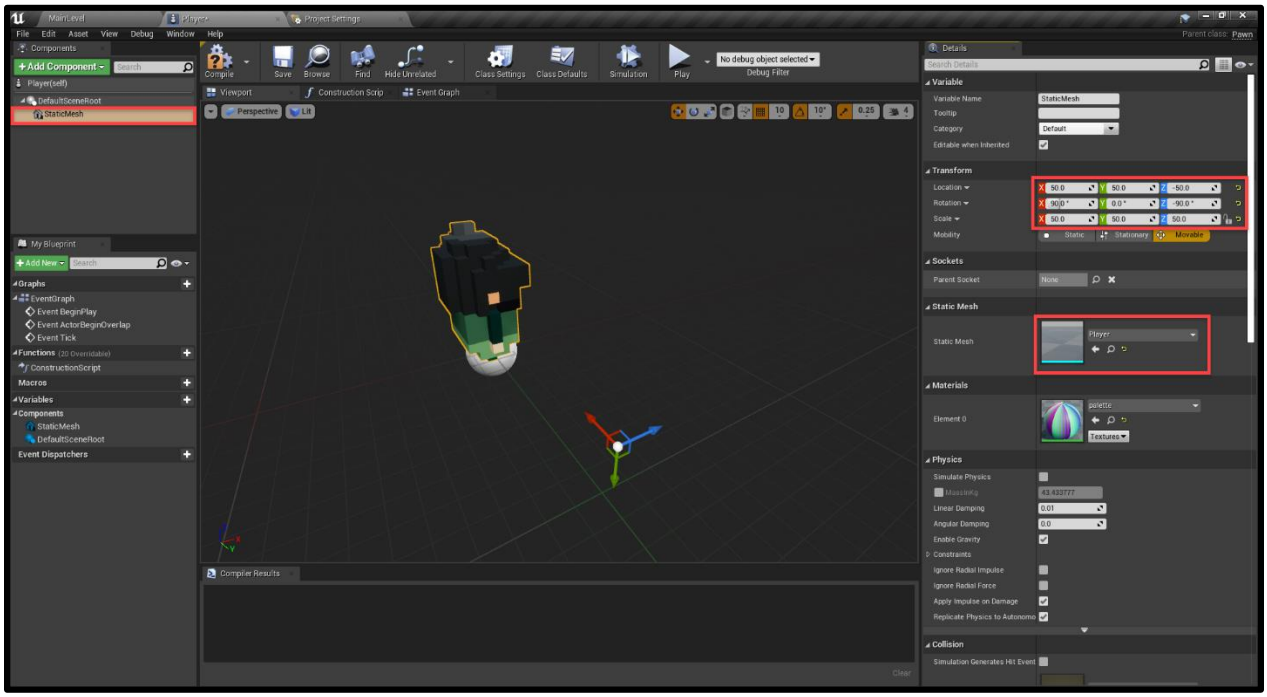
Back in the level editor, we can navigate to our *Blueprints* folder and create a new blueprint for our player. Make it's parent class a *Pawn* and call it **Player**.



Open it up and we can begin.

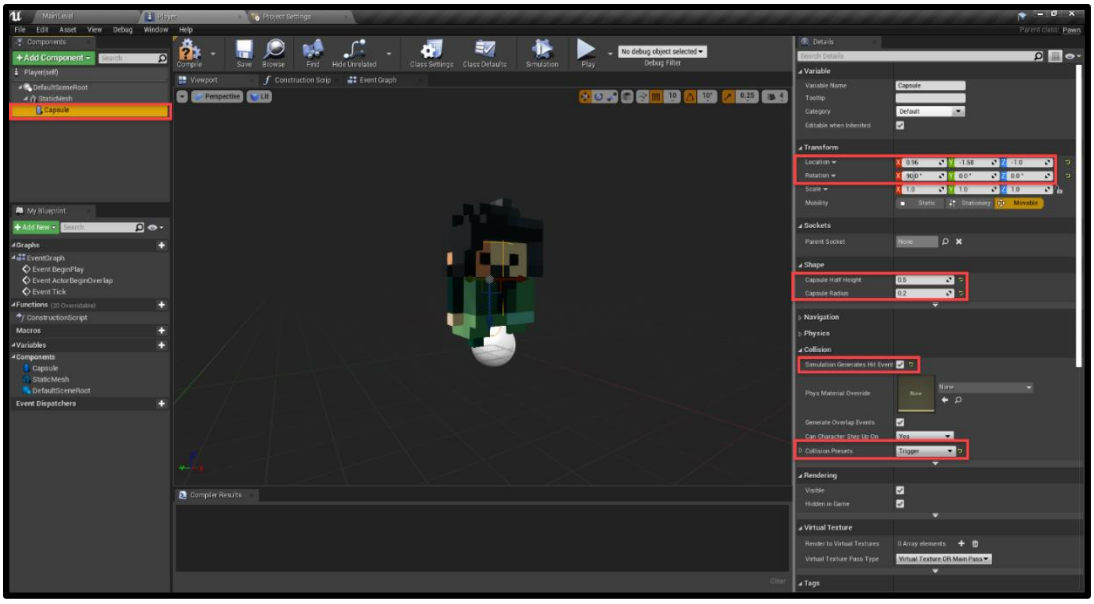
First, create a new **Static Mesh** component.

- Set the **Static Mesh** to *Player*
- Set the **Location** to *50, 50, -50*
- Set the **Rotation** to *90, 0, -90*
- Set the **Scale** to *50, 50, 50*
- (not pictured) Disable **Generate Overlap Events**



Next, create a **Capsule Collider** component as a child of the static mesh.

- Set the **Location** to *0.96, -1.58, -1.0*
- Set the **Rotation** to *90, 0, 0*
- Set the **Capsule Half Height** to *0.5*
- Set the **Capsule Radius** to *0.2*
- Enable **Simulation Generates Hit Events**
- Set the **Collision Presets** to *Trigger*

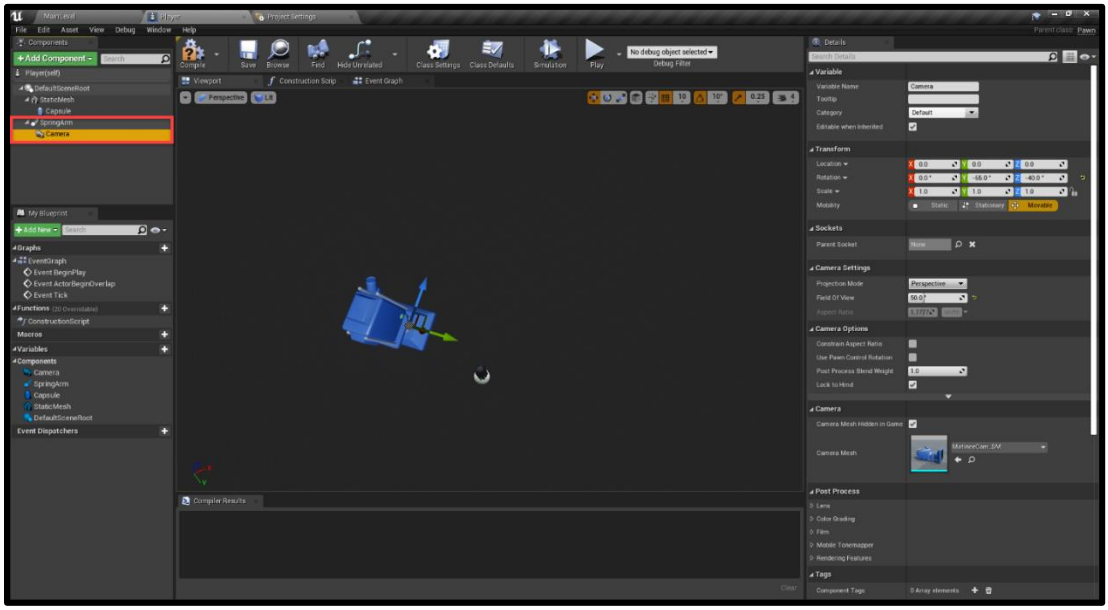


Finally, we have the camera setup. Create a new **Spring Arm** component as the child of static mesh. This component will hold our camera in place, even if we're rotating.

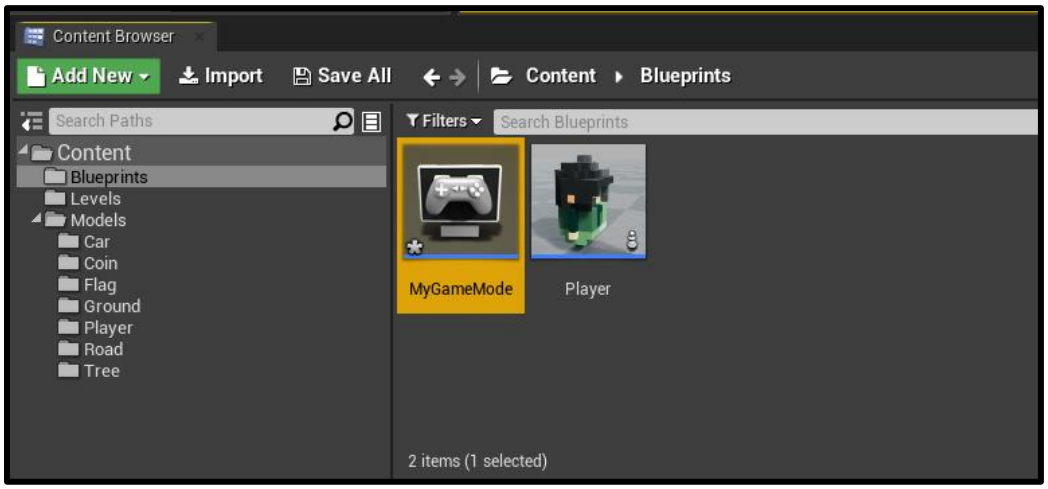
- Set the **Socket Offset** to *100, 0, 0*
- Set the **Target Offset** to *0, 227, 820*

Then, create a **Camera** component as a child.

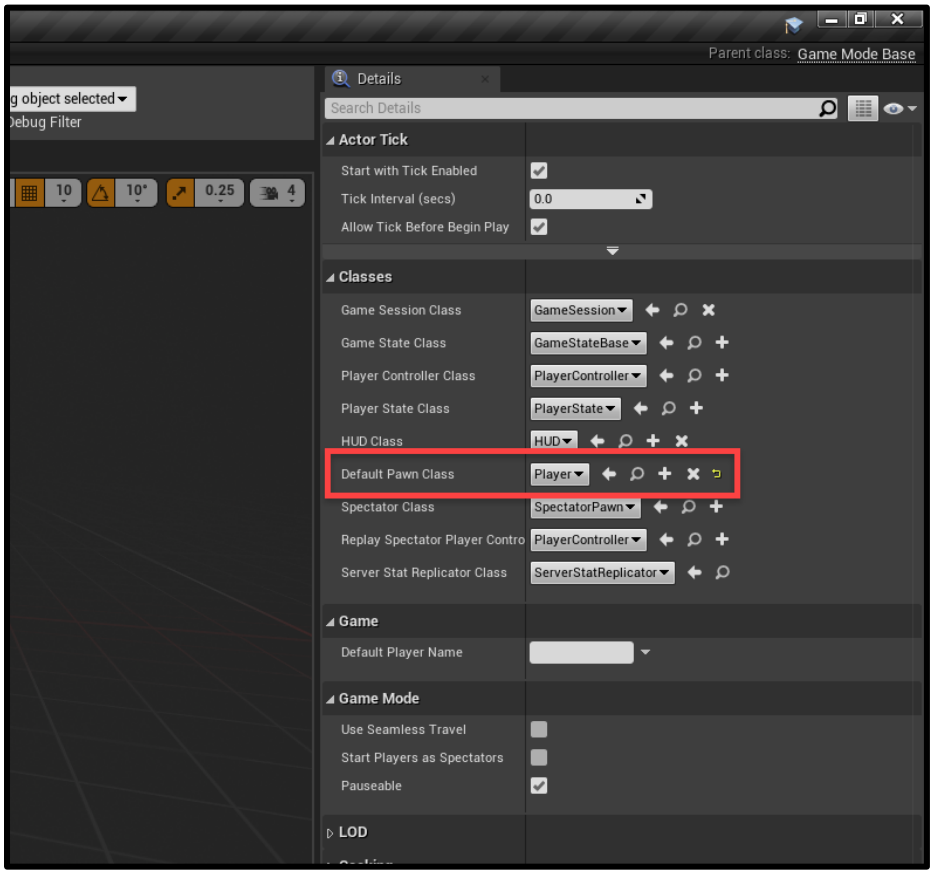
- Set the **Rotation** to *0, -65, -40*
- Set the **Field Of View** to *50*



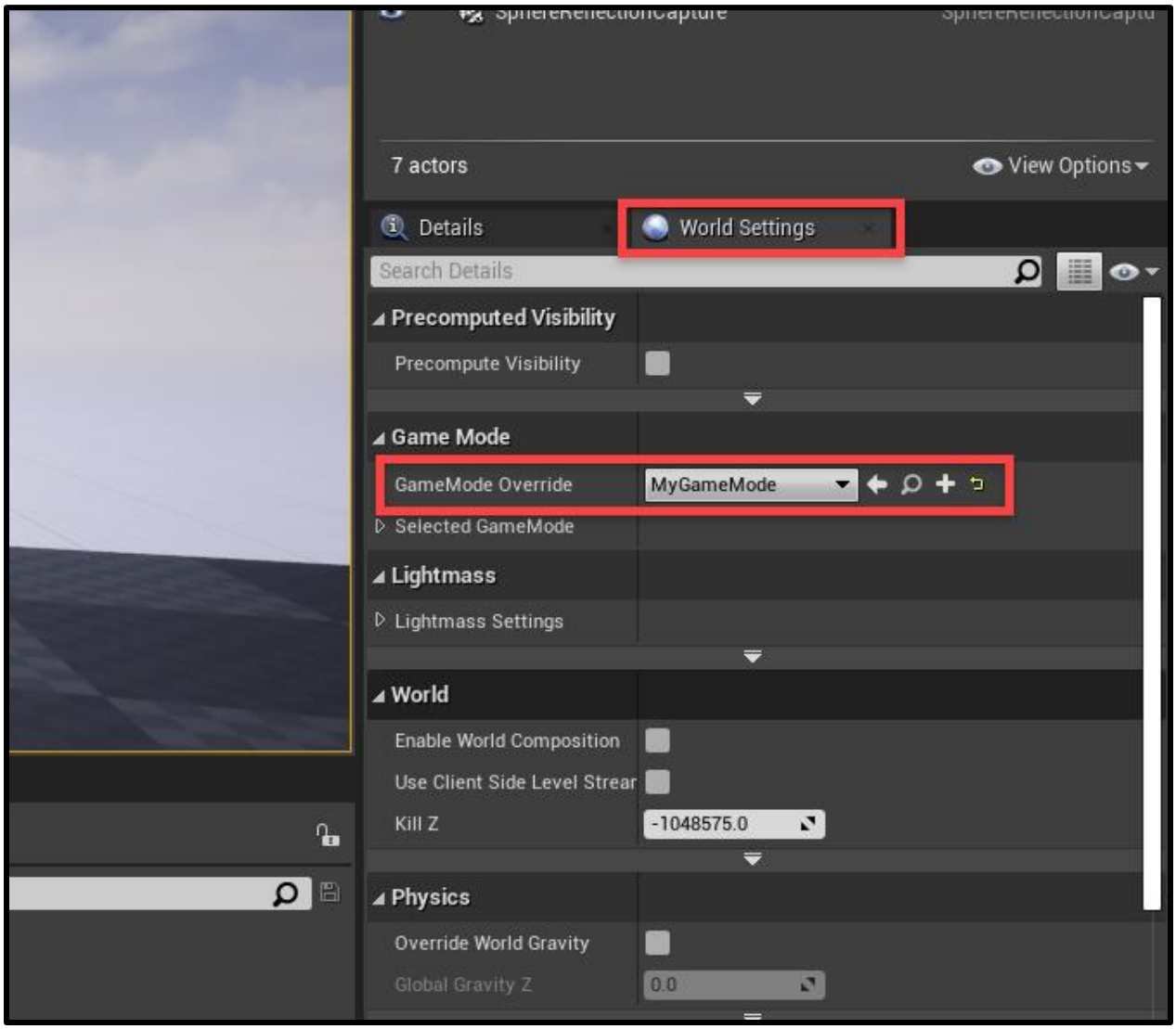
Let's now go back to the level editor and in order to play as the player, we need to create a game mode blueprint.



Open it up and all we need to do, is set the **Default Pawn Class** to our *Player* blueprint.



Save and compile it. Back in the level editor, go to the **World Settings** panel and set the **GameMode Override** to our new game mode.



Now if you press play, you'll see the player appearing with the camera above them.

Moving the Player

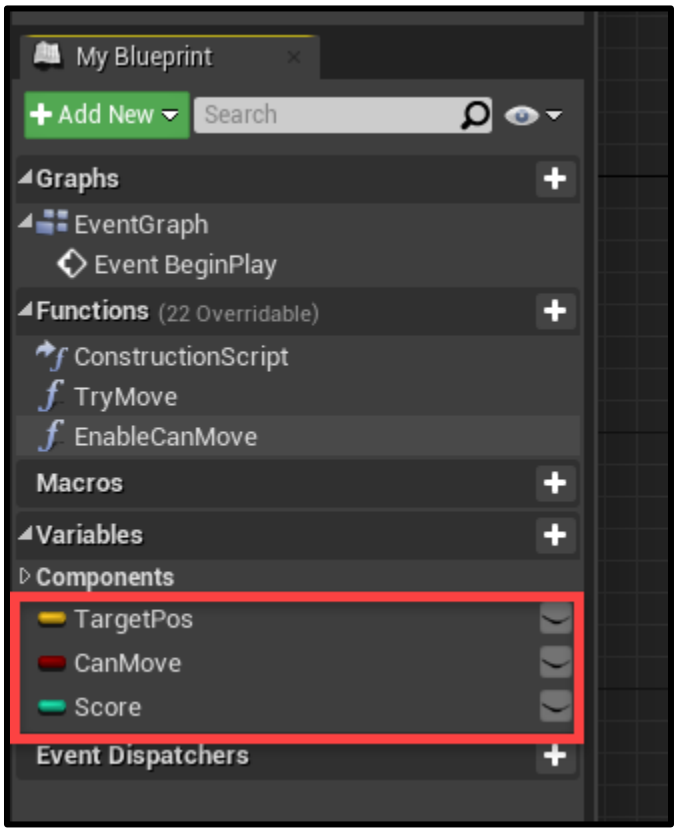
Back in the **Player** blueprint, we can begin to implement the movement. First, we'll need to create the variables.

- TargetPos (Vector)

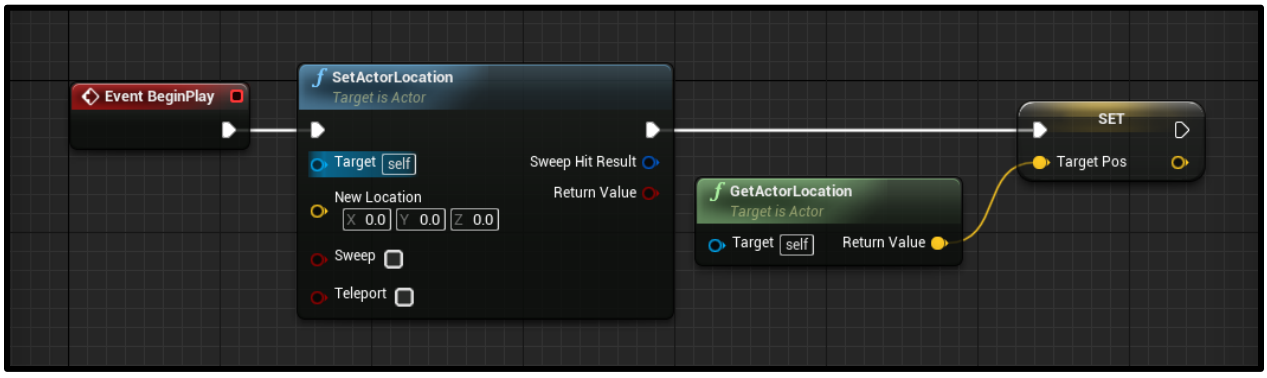
- CanMove (Boolean)
- Score (Integer)

Click **Compile** and set some default values.

- CanMove = true

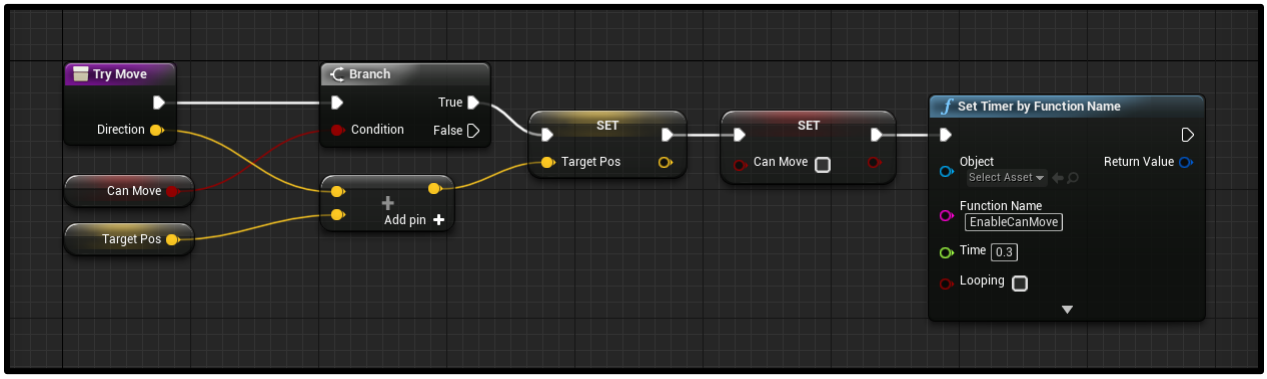


In the **Event Graph**, we can begin by resetting the player position. The *Target Pos* variable is where the player is moving to, so we'll set it to our position first.

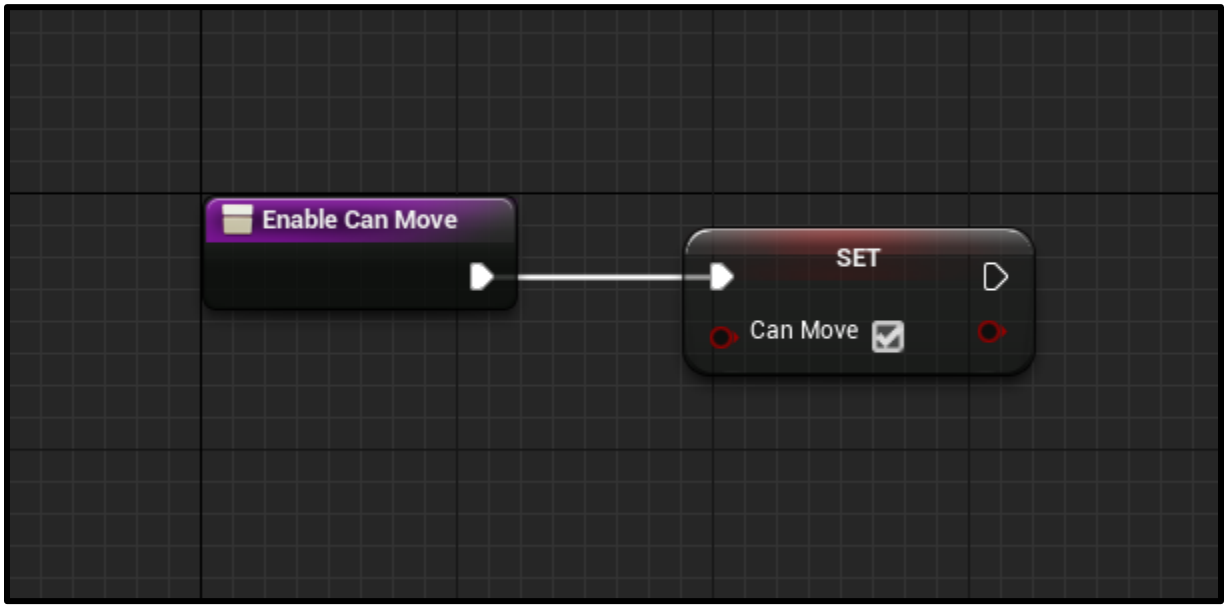


Next, we'll need to do pretty much the same thing for moving forward, left and right. To bypass just having three copies of the same code, we're going to create a new function called **TryMove**. Create a new input for the function called **Direction** and make that of type *Vector*.

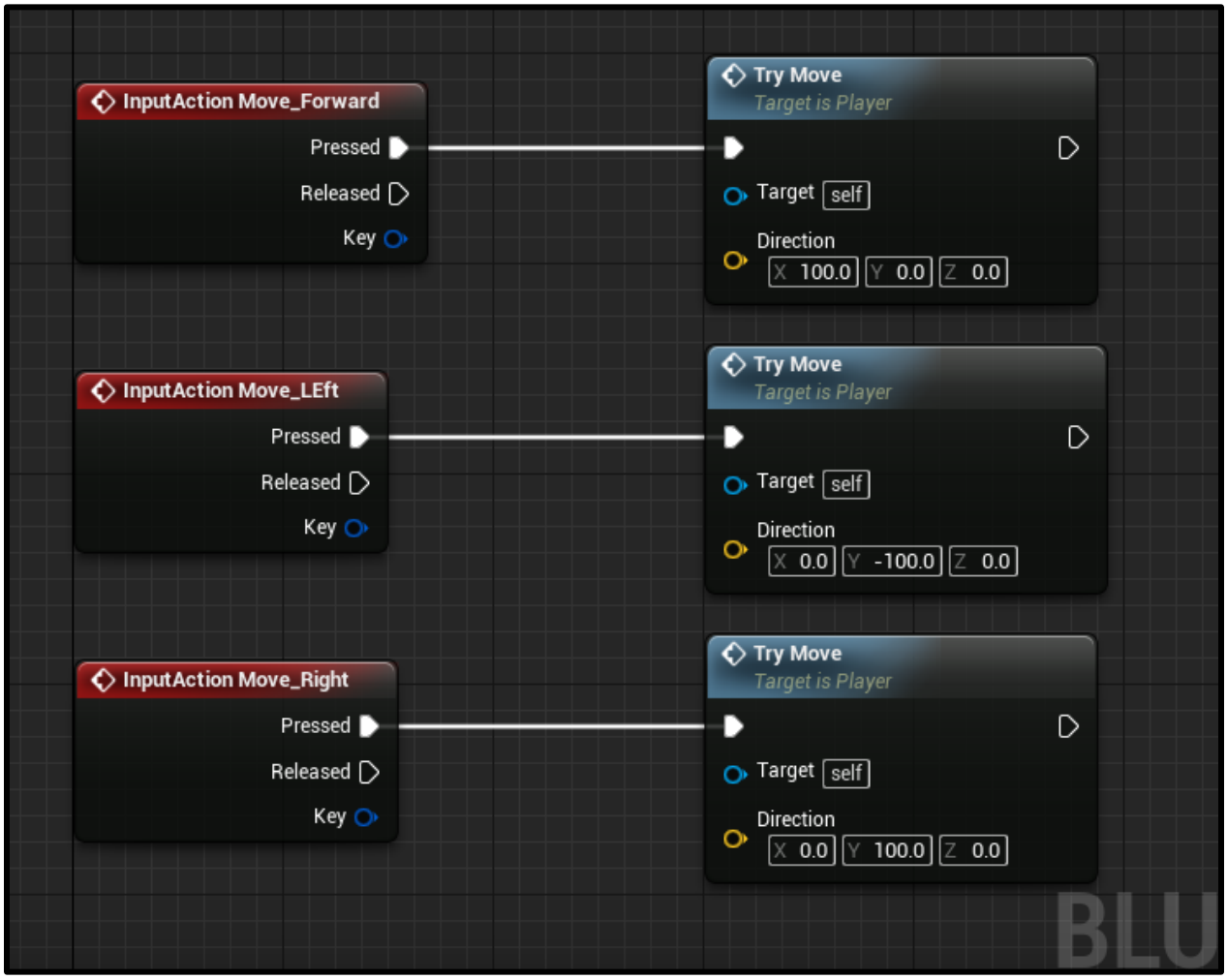
What we're doing here, is checking if we can move. If so, add the direction to our *TargetPos* and then disable can move. But we want to be able to move eventually, so we're going to call a function with a delay. The *EnableCanMove* function is what we'll call after 0.3 seconds.



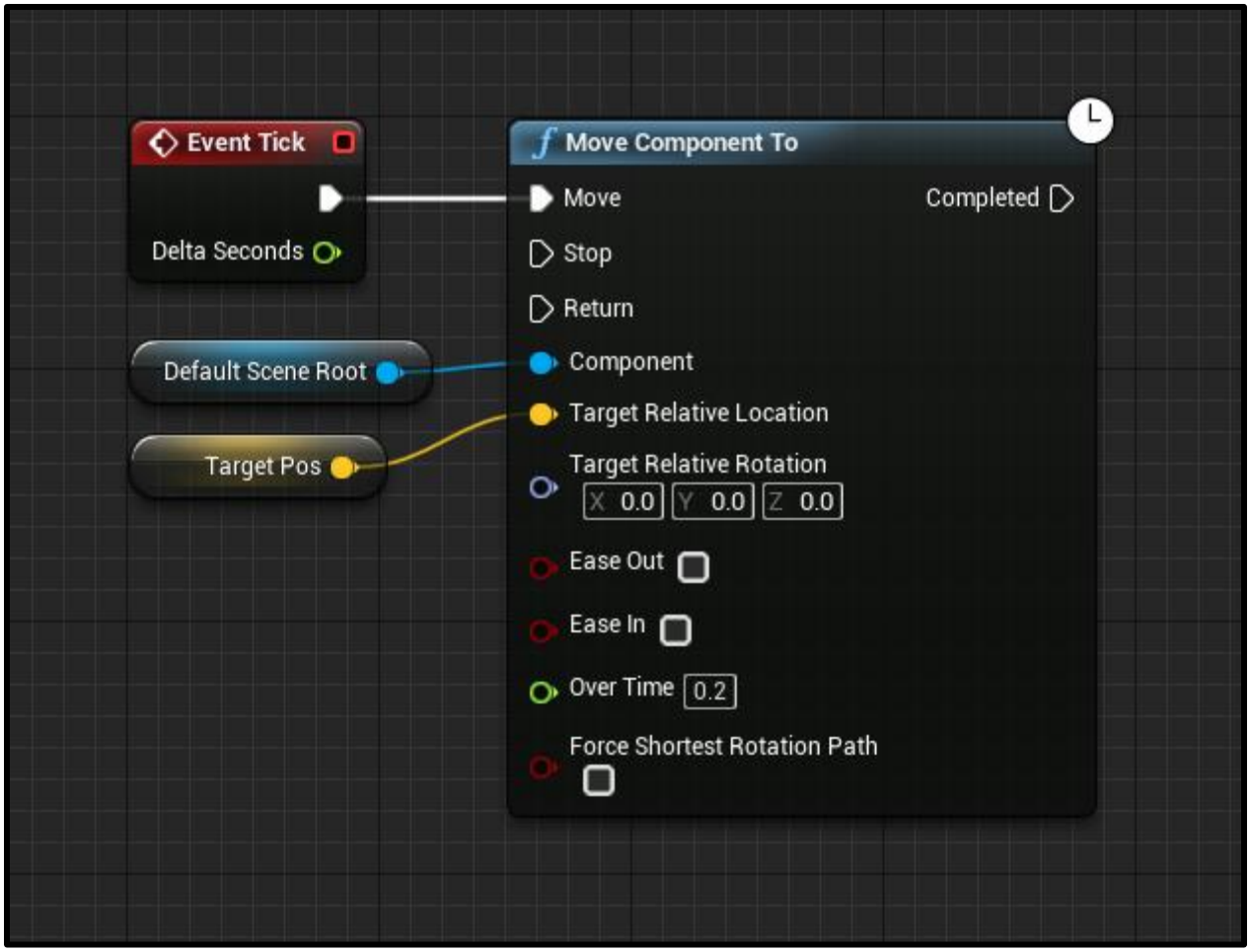
Let's now create that **EnableCanMove** function. All we're doing here is enabling the can move function.



Back in the main event graph, let's now setup our movement input to call the TryMove function. Make sure to fill in the respective *Direction* inputs.



Next, we need to actually move the player. So every frame (event tick node), we're going to trigger the **Move Component To** node which moves a component to the requested position.



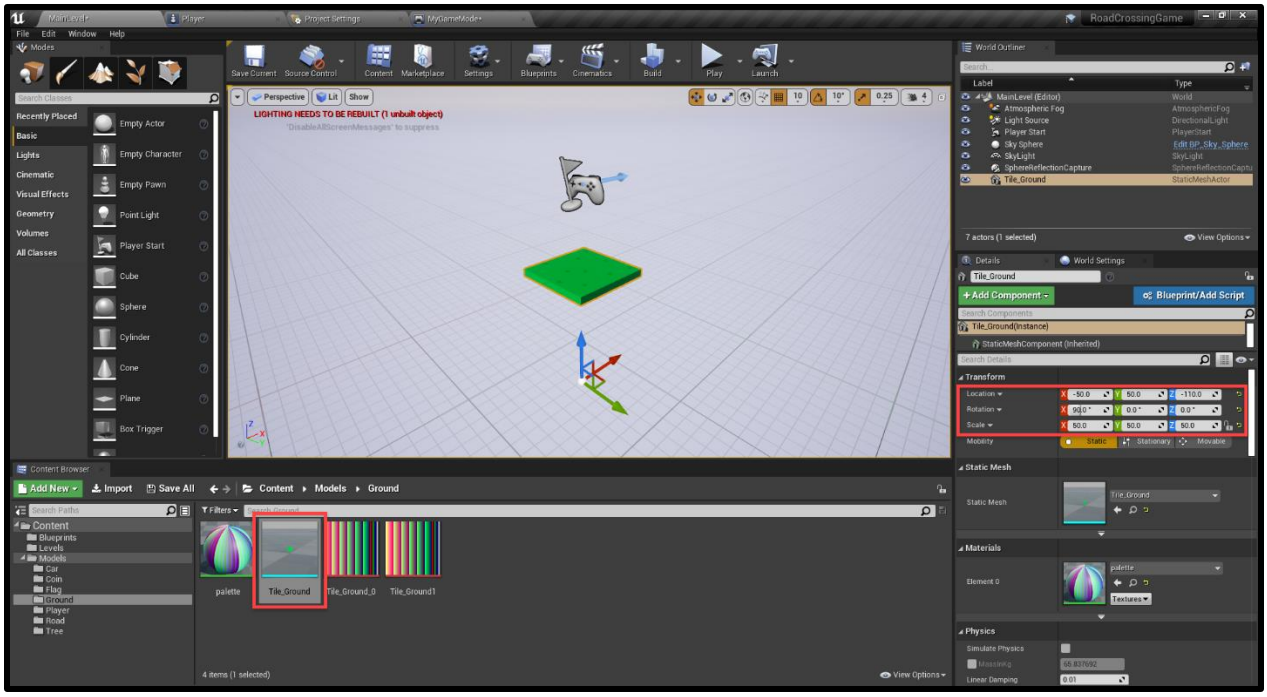
You can now press play and test it out!

Level Layout

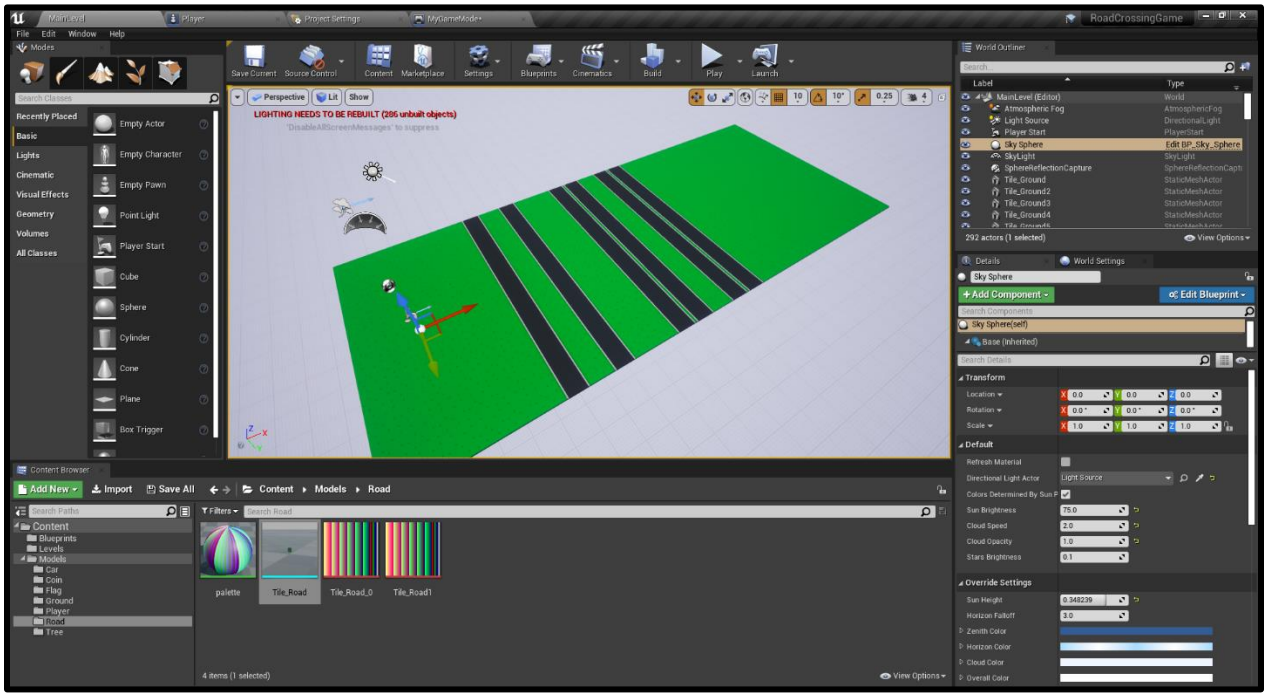
Before we continue with the rest of the game's systems, let's create our level. So in the level editor begin by deleting the floor object.

Then, in the *Models/Ground* folder, drag in the **Tile_Ground** model.

- Set the **Location** to *-50, 50, -110*
- Set the **Rotation** to *90, 0, 0*
- Set the **Scale** to *50, 50, 50*

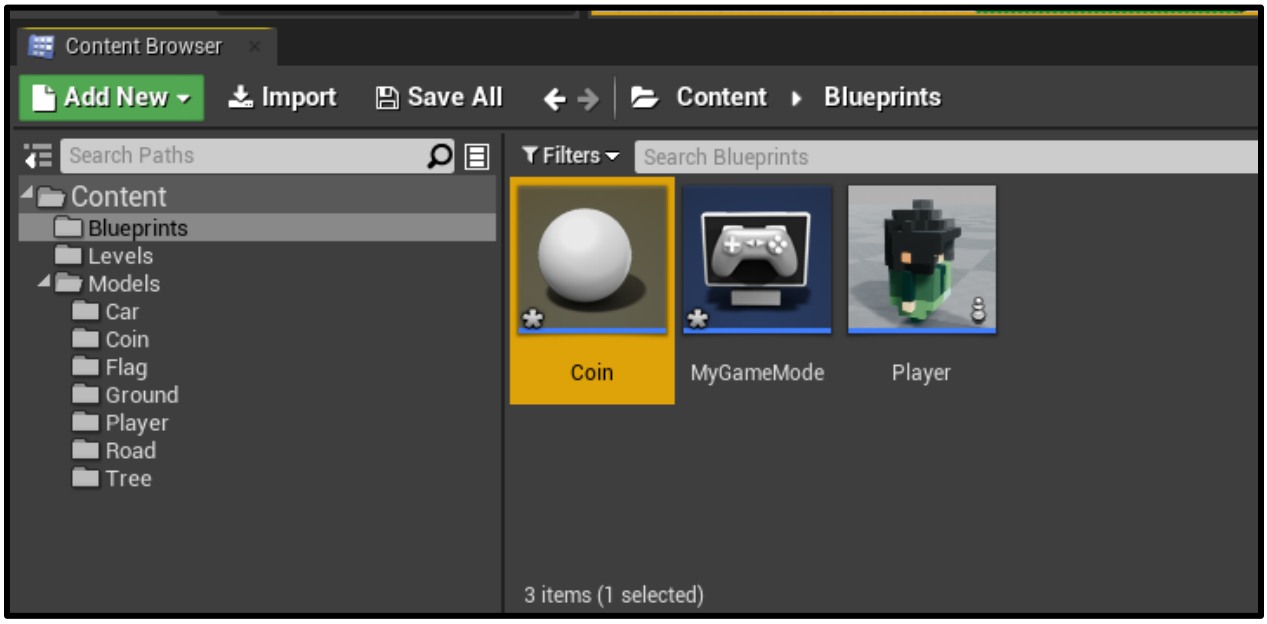


Combining the ground and road tiles, create a layout like the one below.



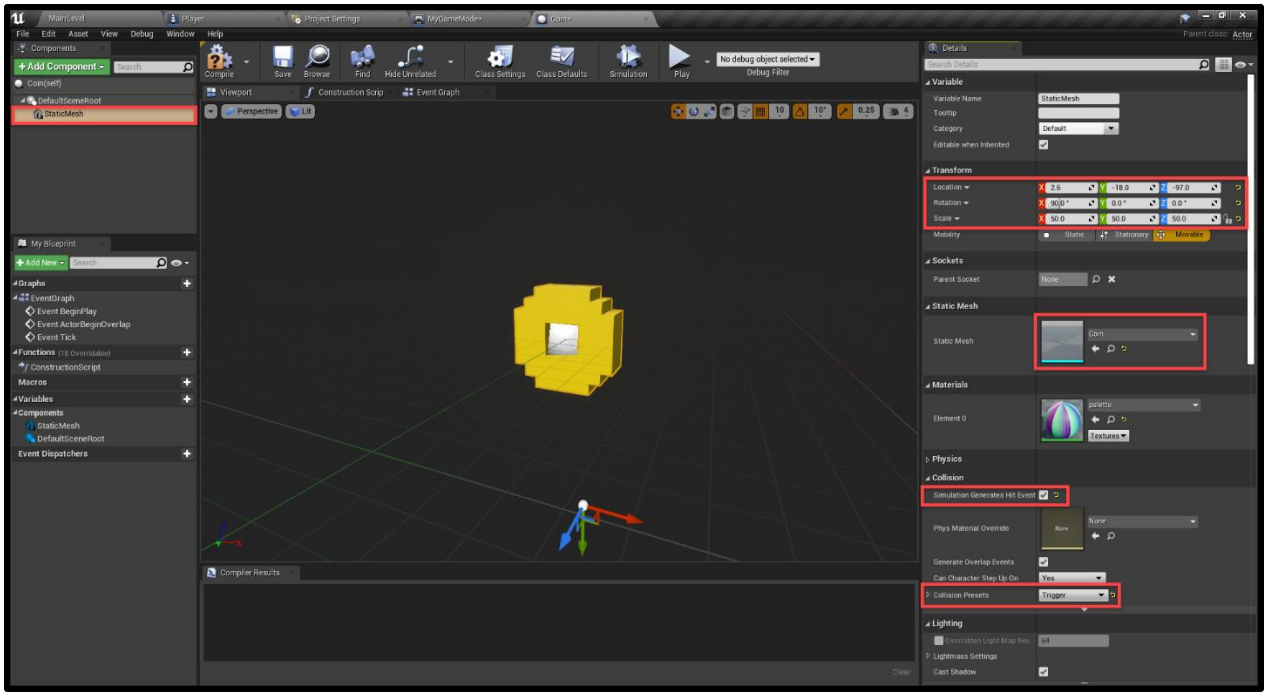
Collectable Coins

When playing the game, players can collect coins to increase their score. Create a new blueprint (parent class of *Actor*) and call it **Coin**.

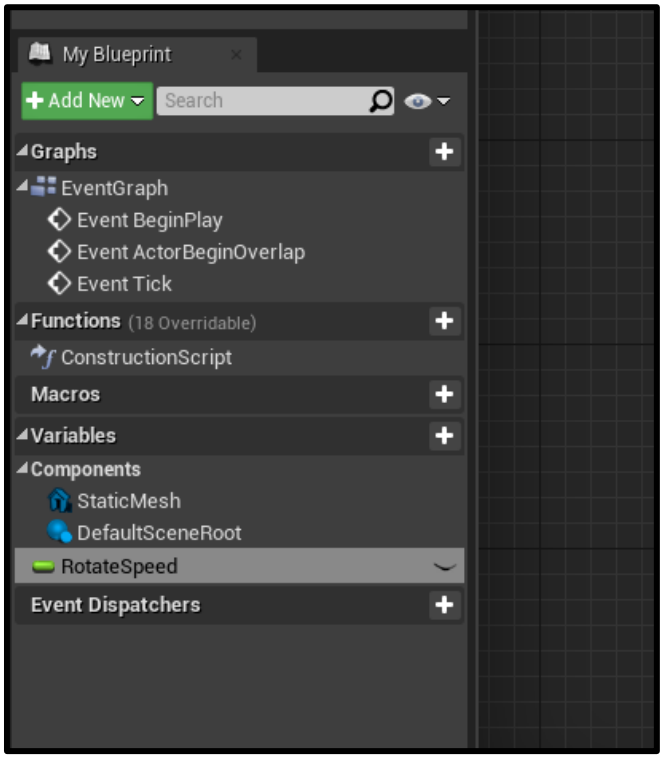


All we're going to do component wise, is create a new **Static Mesh** component.

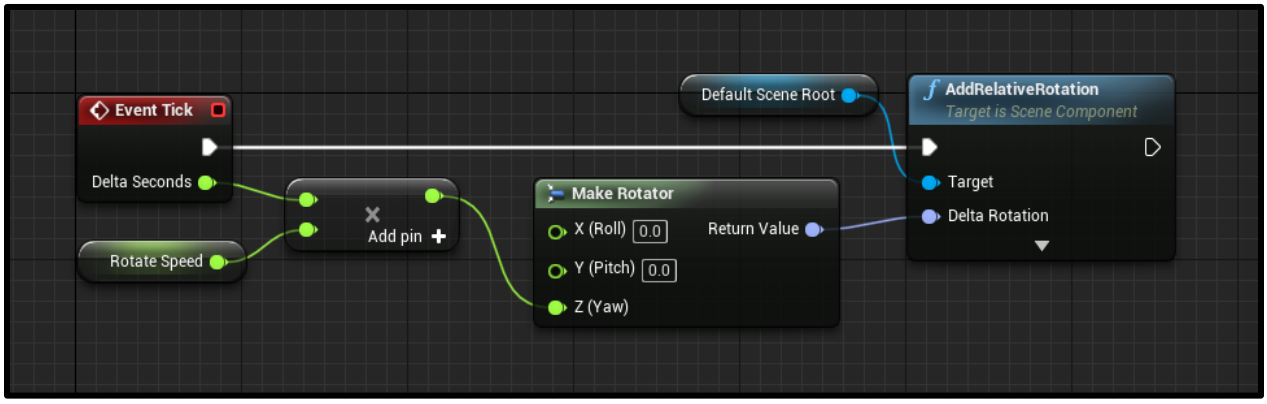
- Set the **Static Mesh** to *Coin*
- Set the **Location** to *2.6, -18.0, -97.0*
- Set the **Rotation** to *90, 0, 0*
- Set the **Scale** to *50, 50, 50*
- Enable **Simulation Generates Hit Events**
- Set the **Collision Presets** to *Trigger*



Create a new variable of type *Float* and call it **RotateSpeed**. Compile and set the default value to *90*.



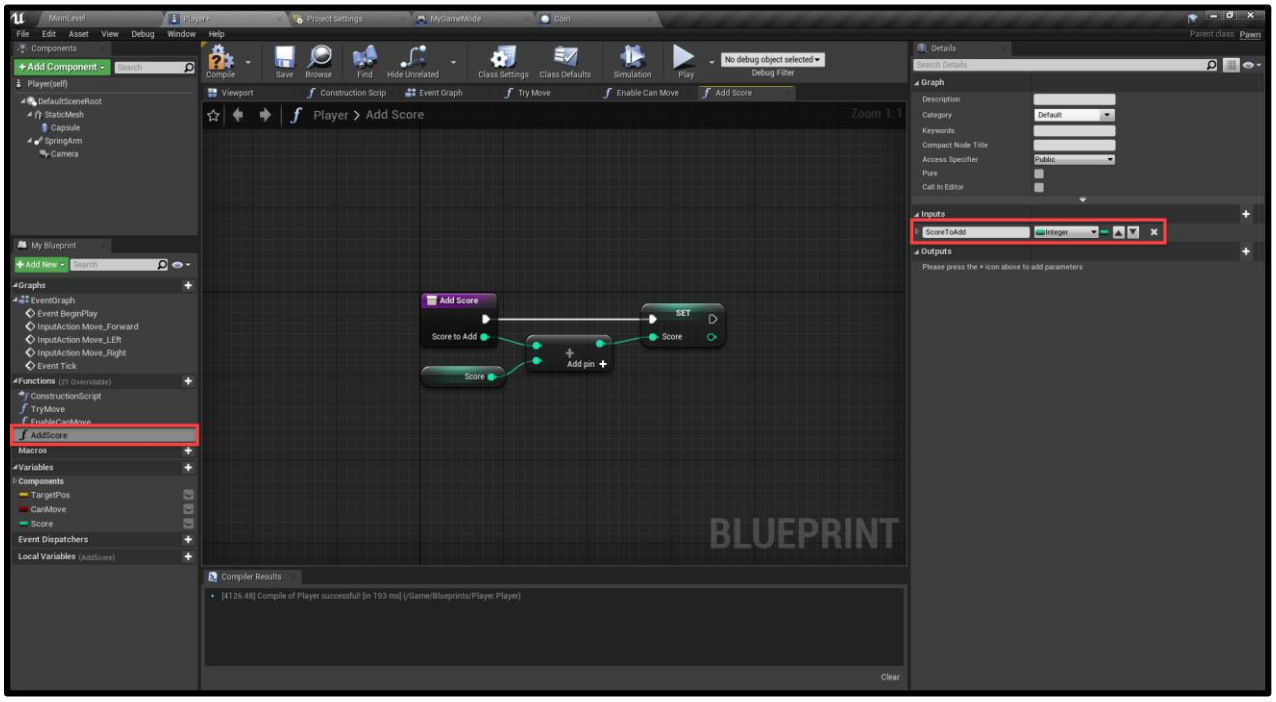
In the event graph, we're first going to set it up so the coin rotates over time. We're multiplying by the delta seconds so that the coin rotates in terms of degrees per second and not degrees per frame. This will make it the same across any frame rate.



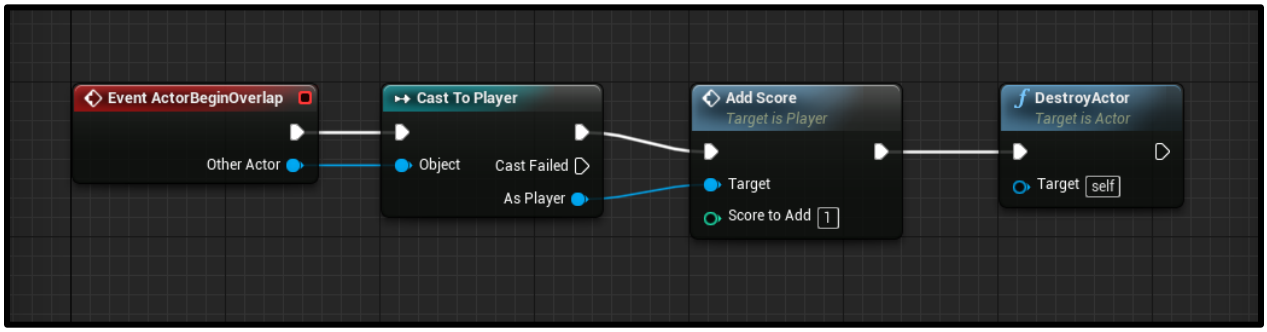
Next, we need to add the score to the player when they collect the coin. First, go to the **Player** blueprint and create a new function called **AddScore**.

- Create an input for the function called **ScoreToAdd** of type *Integer*

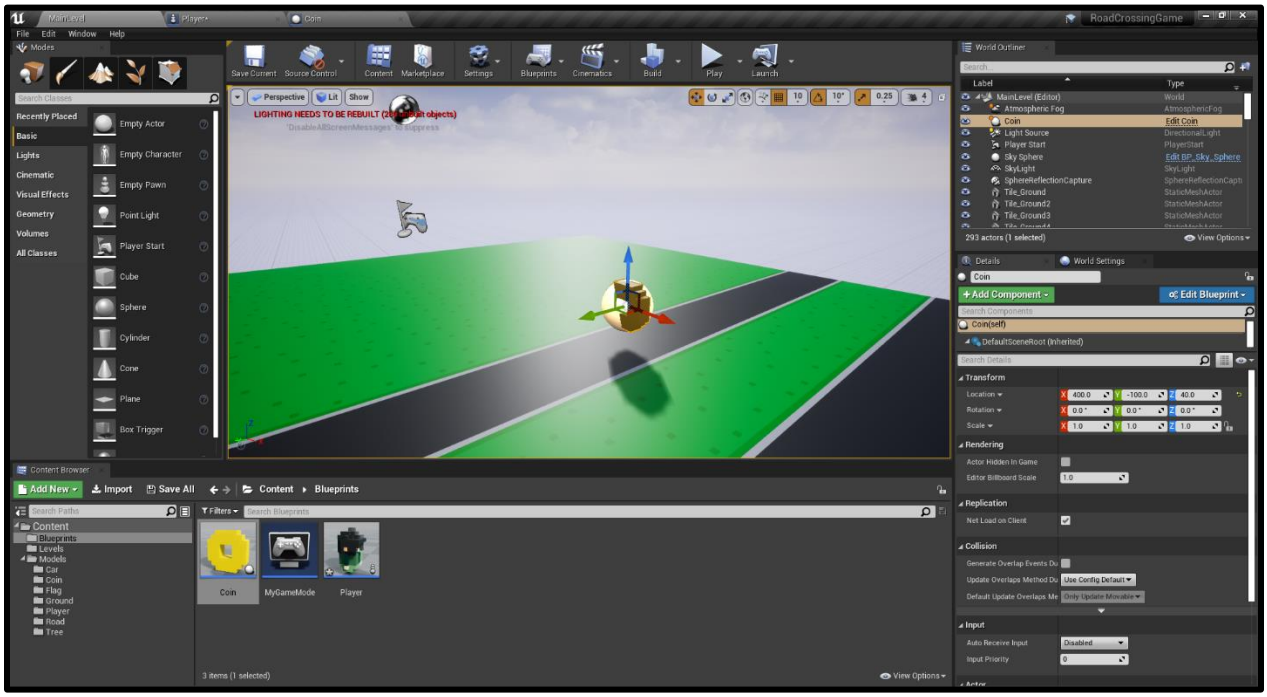
This function will just add to the existing score.



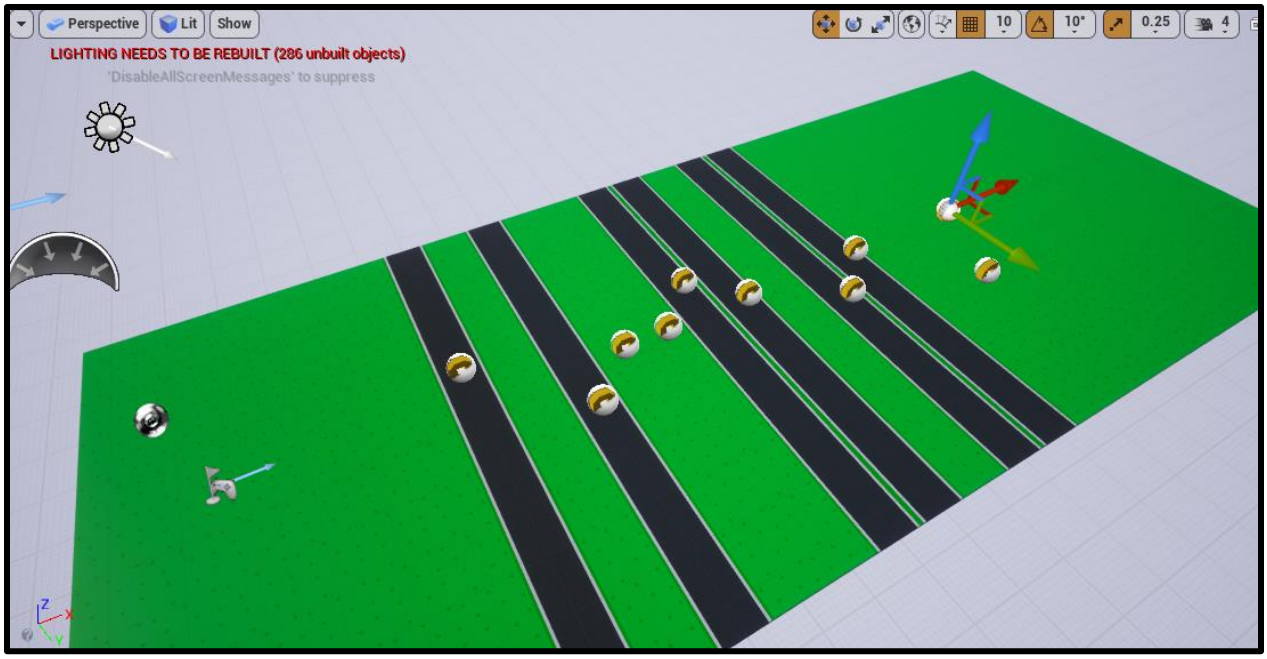
Back in the **Coin** blueprint, let's create the logic so that when the player collides with the coin, we add to their score and get destroyed.



In the level editor now, we can drag in the coin blueprint and position it where we want. If you press play and collect the coin, you should see that it gets destroyed.



We can also then add in the rest of the coins.

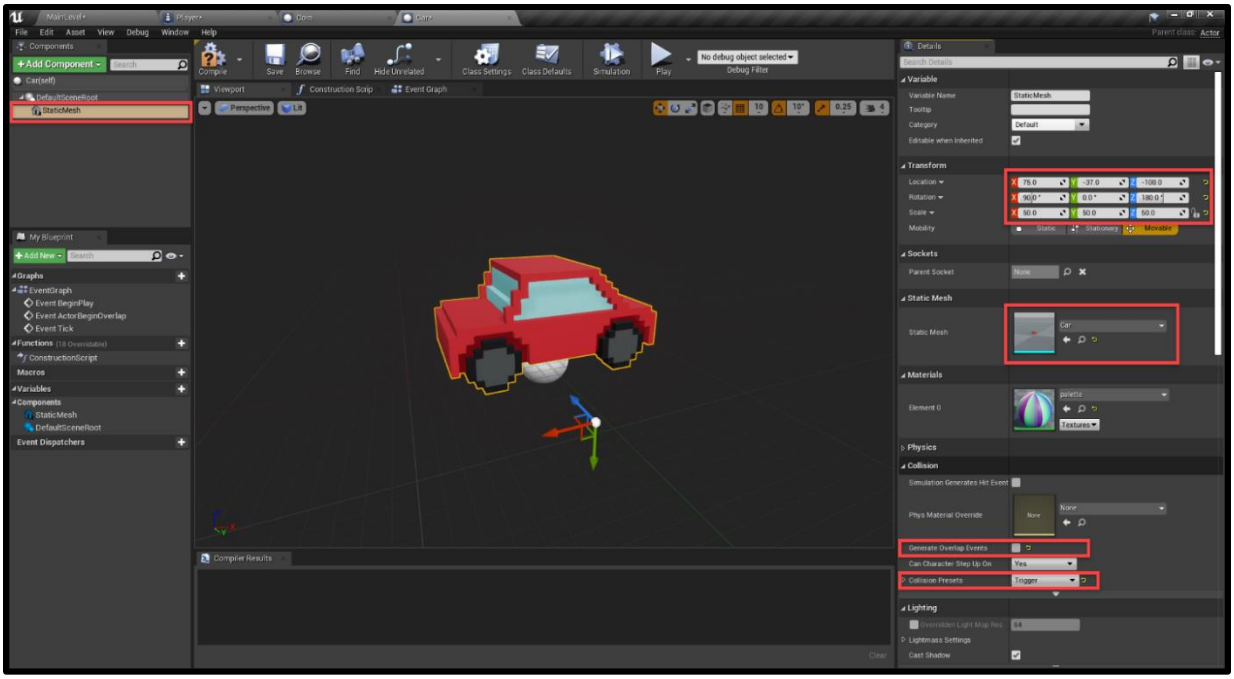


Creating the Cars

The cars are going to be the player's main obstacle. Hitting them will cause the game to restart.

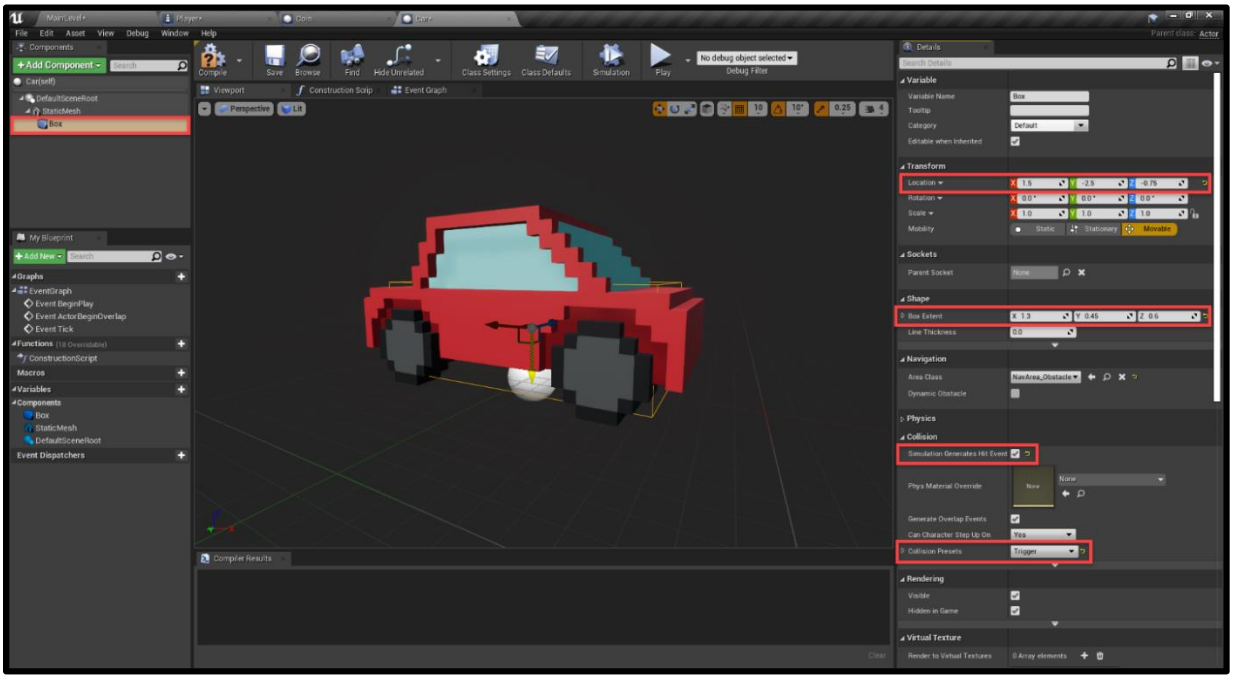
Create a new blueprint (parent class of *Actor*) and call it **Car**. Open it up and we'll start by creating a static mesh component.

- Set the **Static Mesh** to *Car*
- Set the **Location** to *75, -37, -100*
- Set the **Rotation** to *90, 0, 180*
- Set the **Scale** to *50, 50, 50*
- Disable **Generate Overlap Events**
- Set the **Collision Presets** to *Trigger*



Next, create a **Box Collider** component.

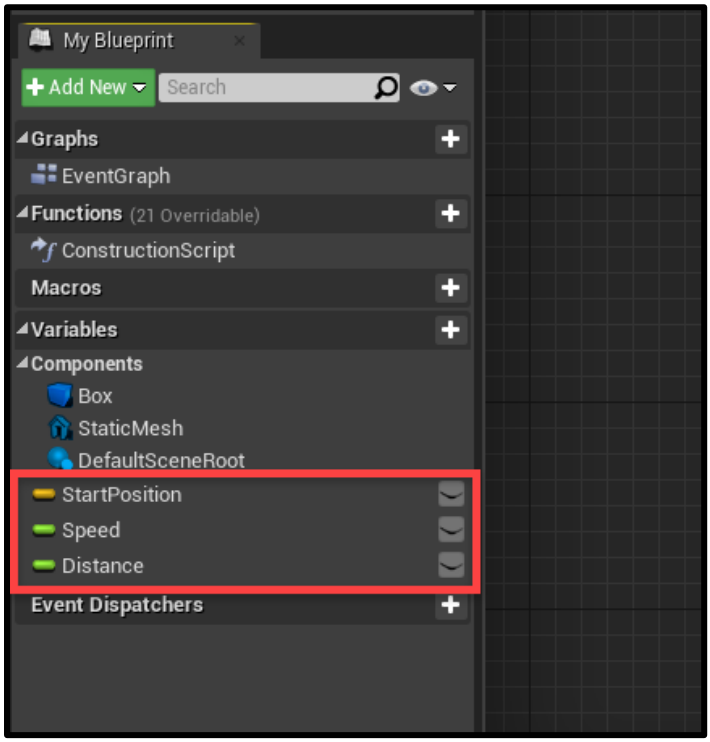
- Set the **Location** to *1.5, -2.5, -0.75*
- Set the **Box Extents** to *1.3, 0.45, 0.6*
- Enable **Simulation Generates Hit Events**
- Set the **Collision Presets** to *Trigger*



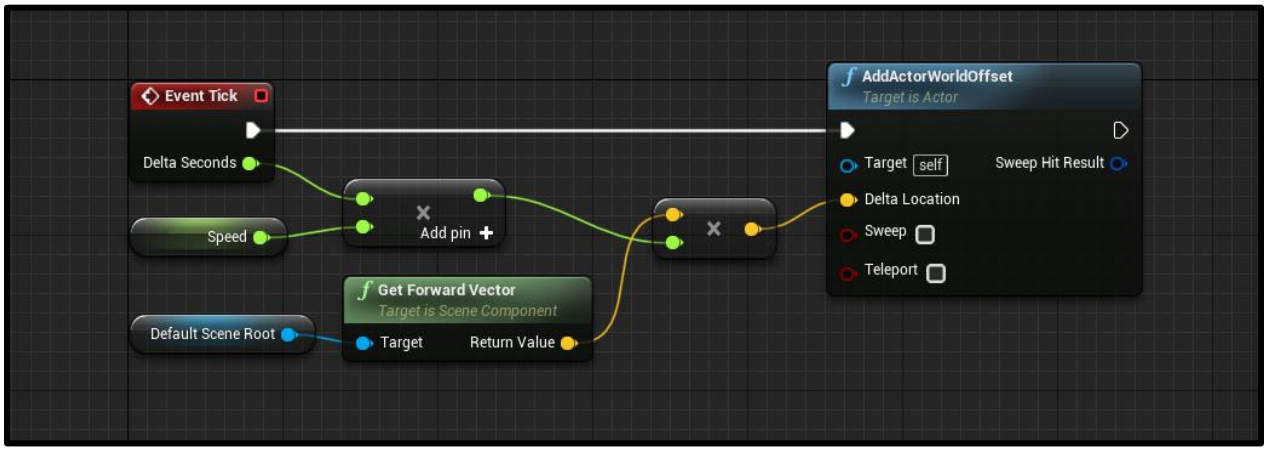
Let's now head over to the **Event Graph** where we'll begin by creating some variables.

- StartPosition (Vector)
- Speed (Float)
- Distance (Float)

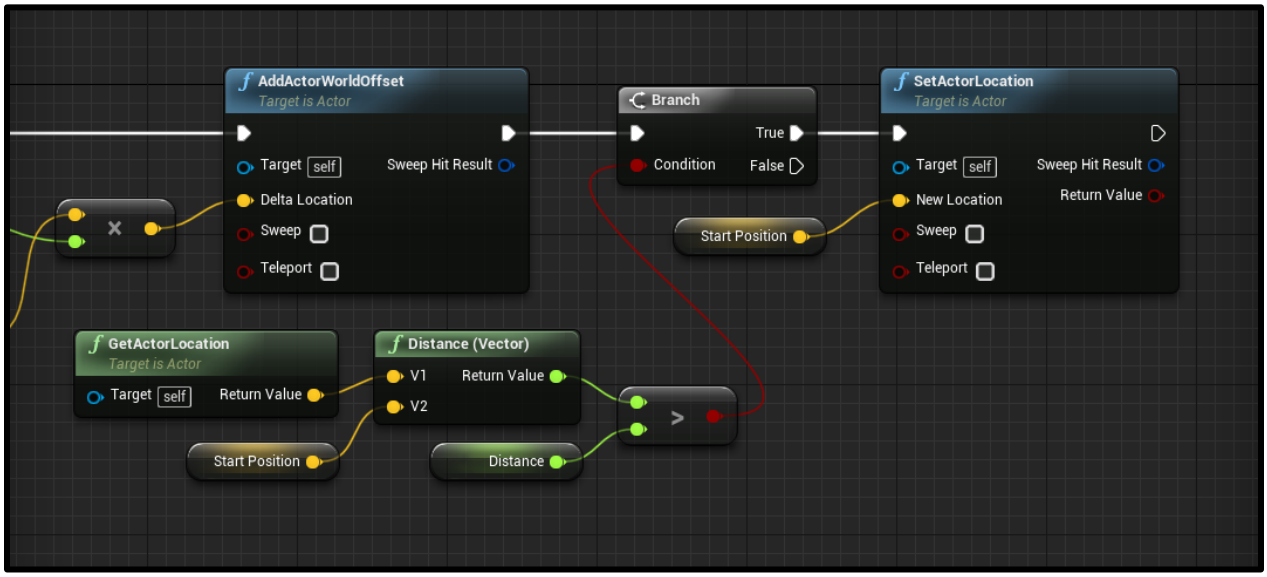
Compile, then for each variable, enable **Instance Editable** so we can change the values in the level editor. But for the speed, let's set that to a default value of *400*.



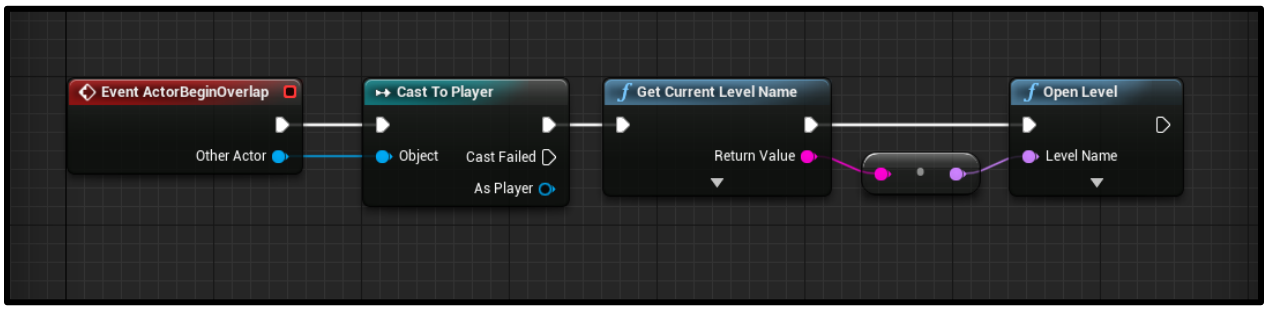
Now for the nodes. Each frame, we're going to be moving in our forward direction.



We're also wanting to check each frame if we've driven as far as the distance variable. If so, then reset our position to the start position.



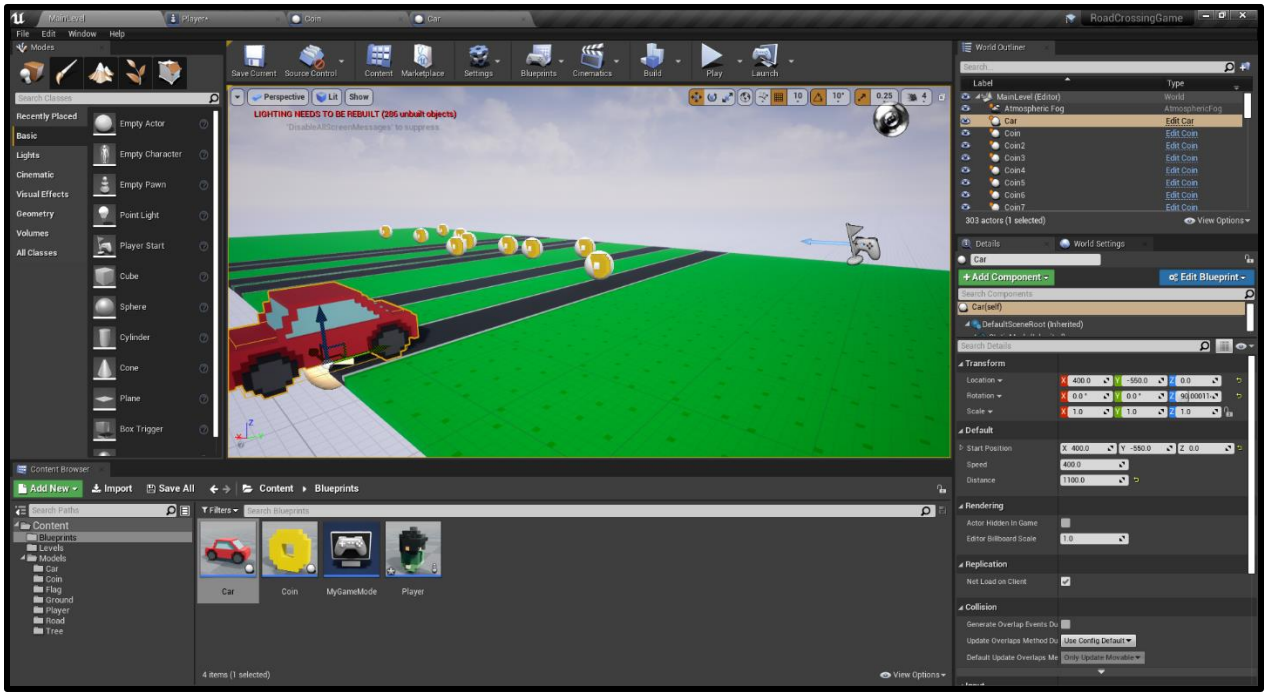
Finally, we're going to check for an overlap of the player collider. If so, we'll restart the scene.



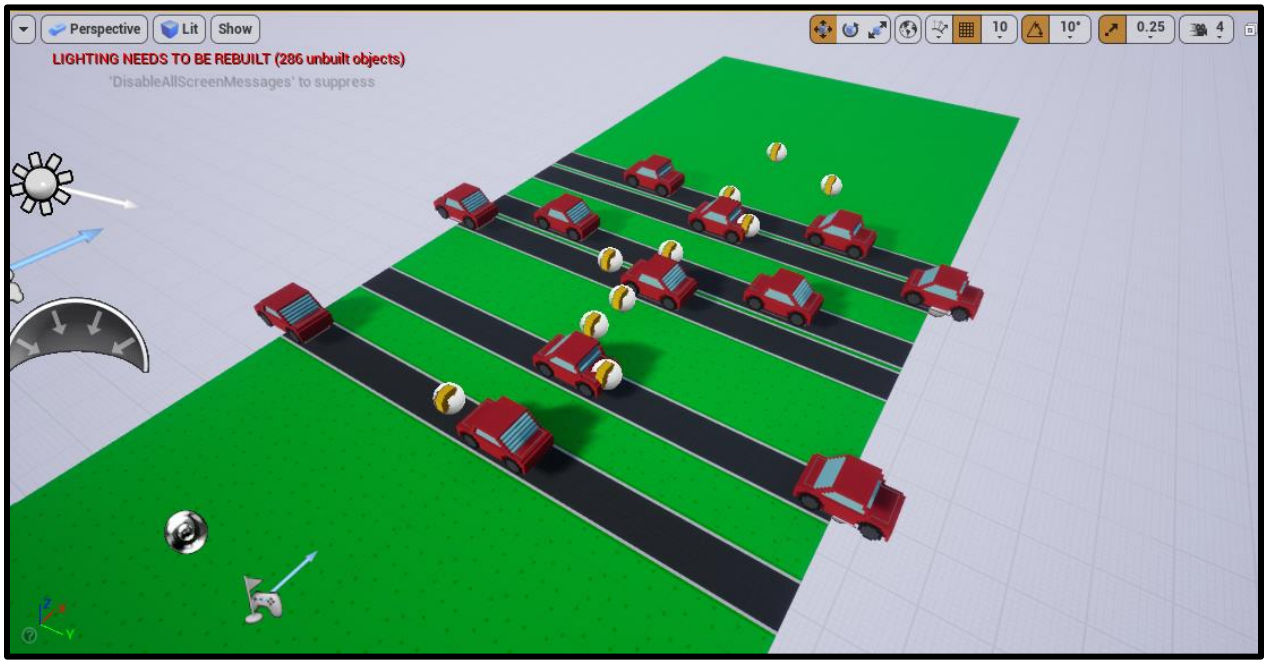
Back in the level editor, let's place down our first car.

- Set the **Location** to *400, -550, 0*
- Set the **Rotation** to *0, 0, 90*
- Set the **Start Position** to *400, -550, 0*
- Set the **Distance** to *1100*

Now if you press play, you should see the car move down the road, then reset its position once it reaches the end.



Go ahead and create many more cars to fill in the roads.

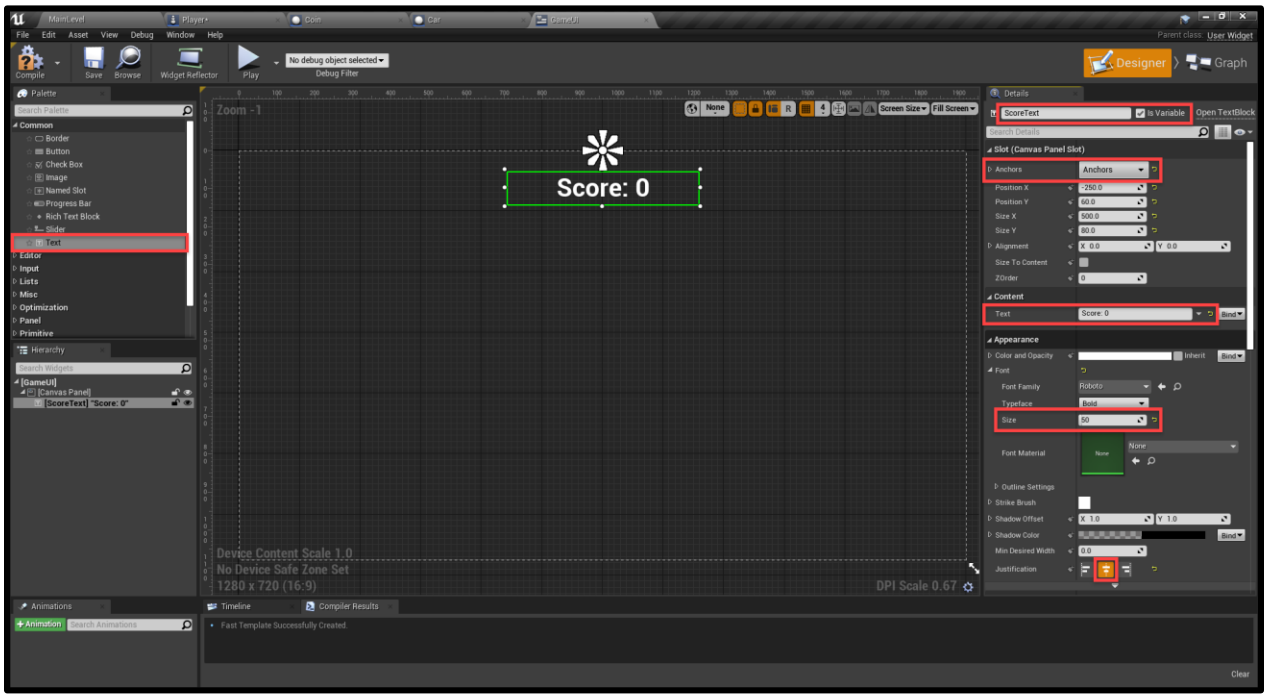


Game UI

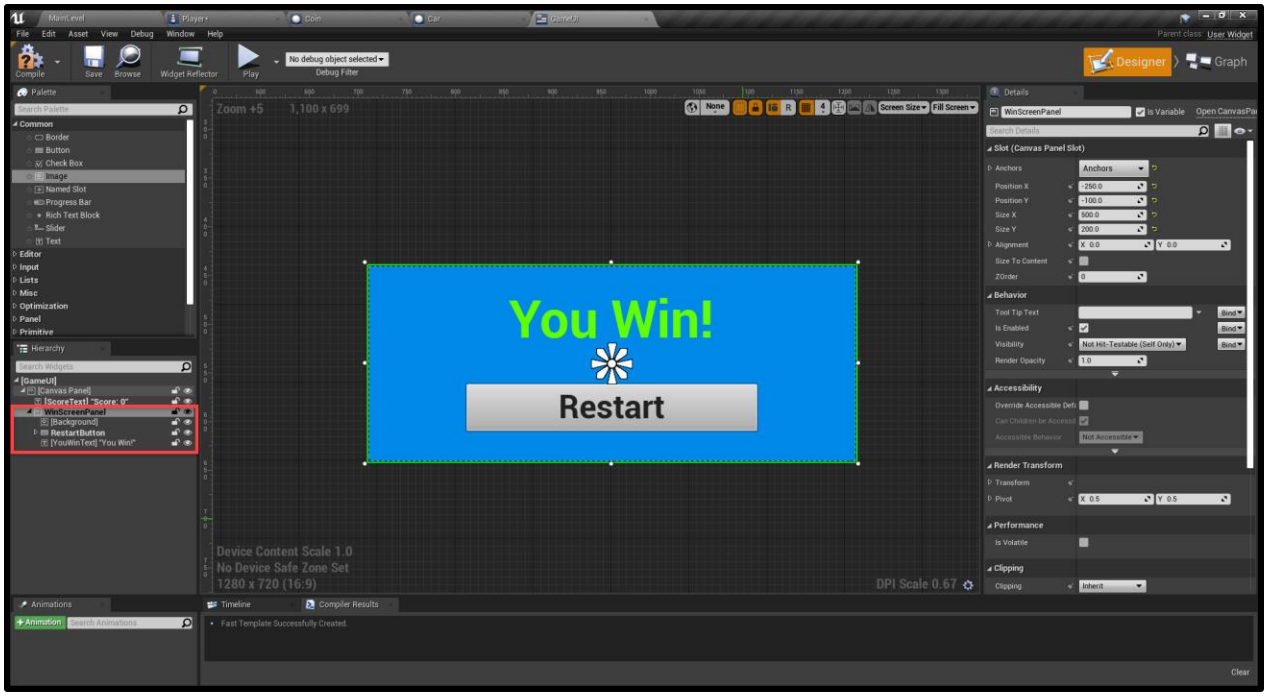
Now we're going to create some UI for the game. We're going to have two elements. A win screen and a score text so the player can see their current score. In the *Blueprints* folder, right click and select *User Interface > Widget Blueprint*. Call it **GameUI**. Double-click to open the UI editor window.

First, drag in a new **Text** component.

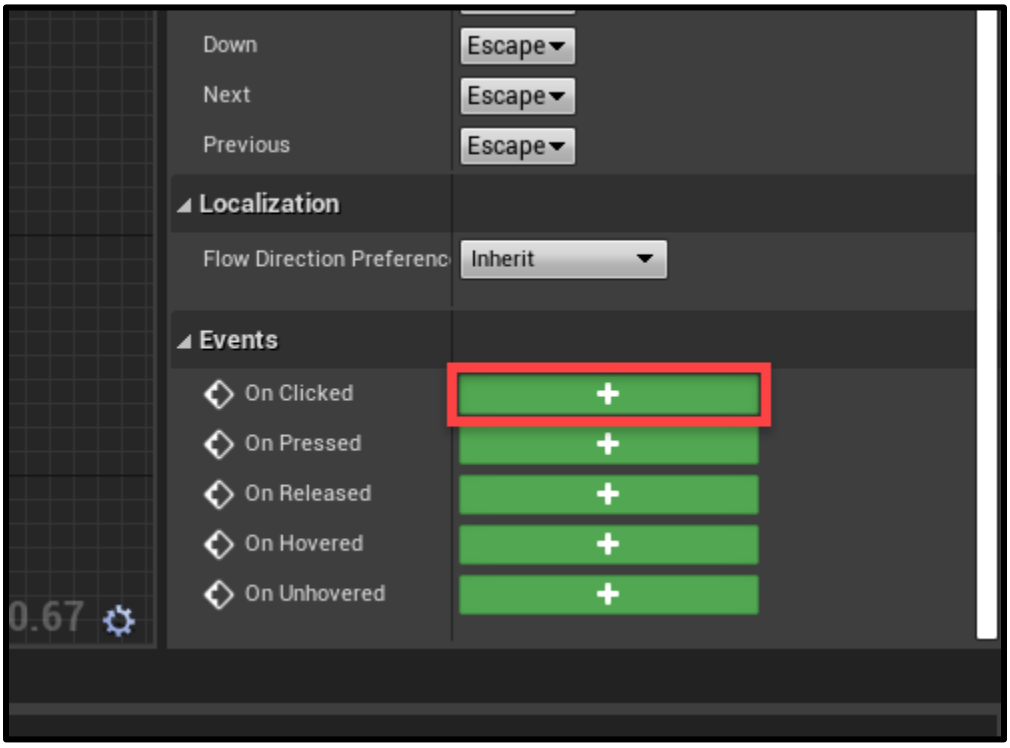
- Set the **Name** to *ScoreText*
- Enable **Is Variable**
- Set the **Anchors** to *Top-Center*
- Set the **Position X** to *-250*
- Set the **Position Y** to *60*
- Set the **Size X** to *500*
- Set the **Size Y** to *80*
- Set the **Text** to *Score: 0*
- Set the **Size** to *50*
- Set the **Justification** to *Center*



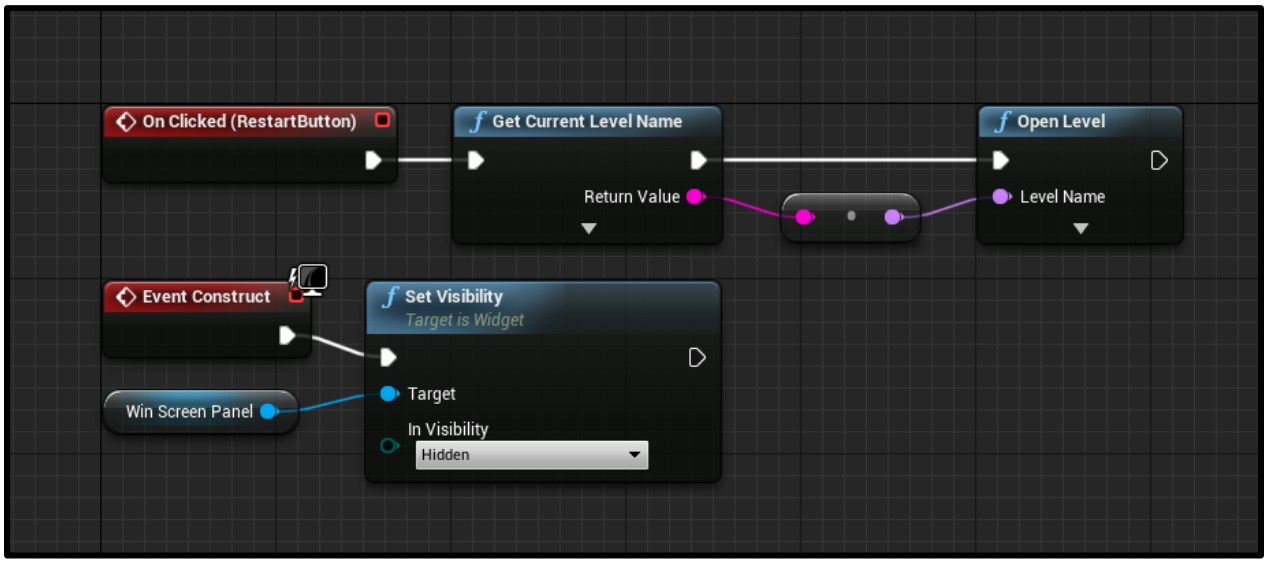
Next, we want to create the win screen. Drag in a canvas panel component. This will be the contents of the win screen. Populate it with a header text and button to restart the game. Make sure that the canvas panel has **Is Variable** enabled.



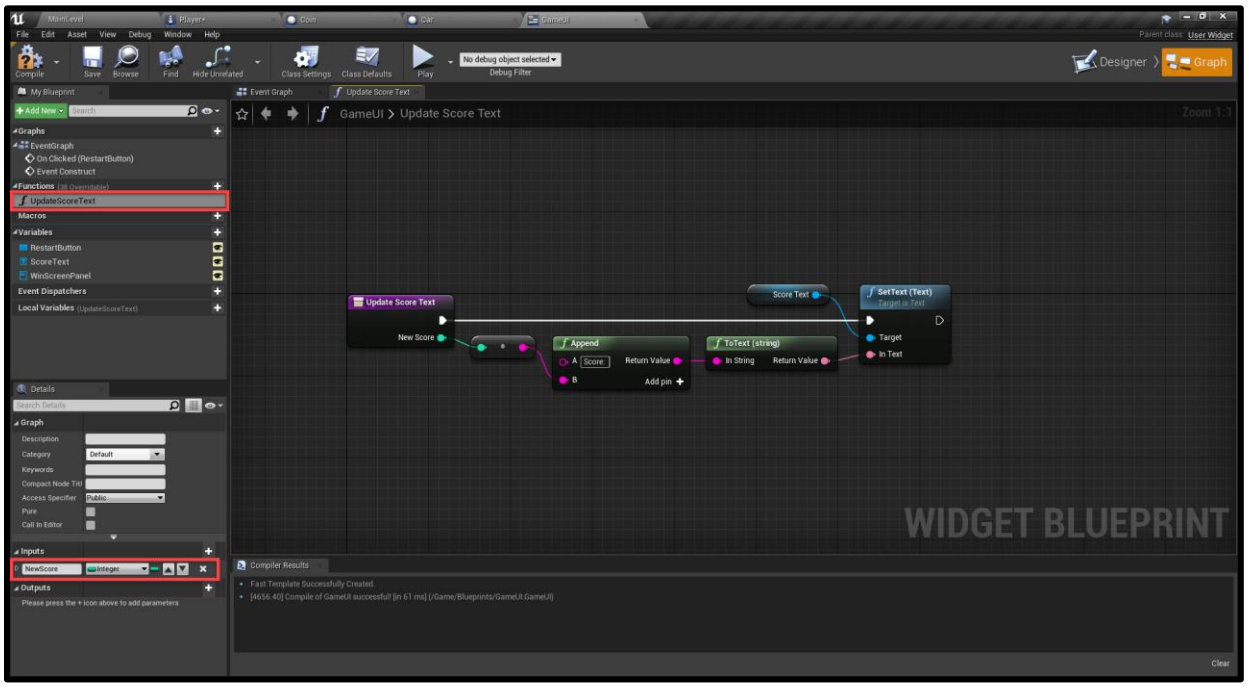
We then want to select our restart button, and in the **Details** panel create an *On Clicked* event.



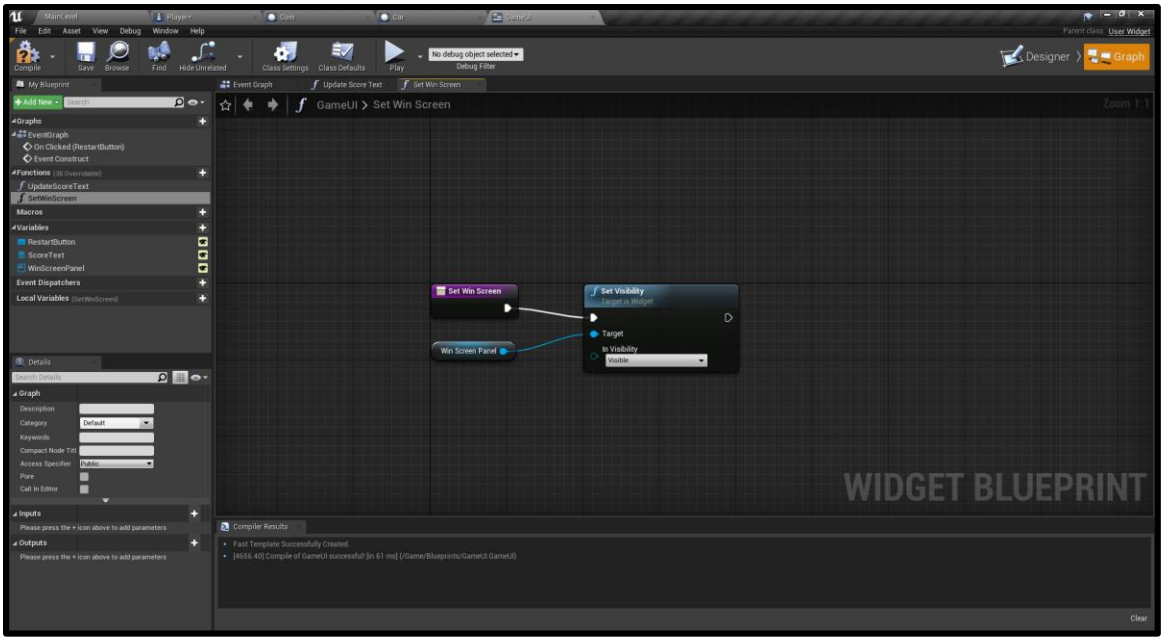
This will create an OnClicked node in the graph. All we want to do with this, is when triggered - restart the current level. Also, we'll add the EventConstruct node which gets called when the UI is created. All we're doing here, is disabling the win screen panel.



Next, we'll create the **UpdateScoreText** function. This gets called when we collect a coin. Add an input of type *Integer* and call it **NewScore**.

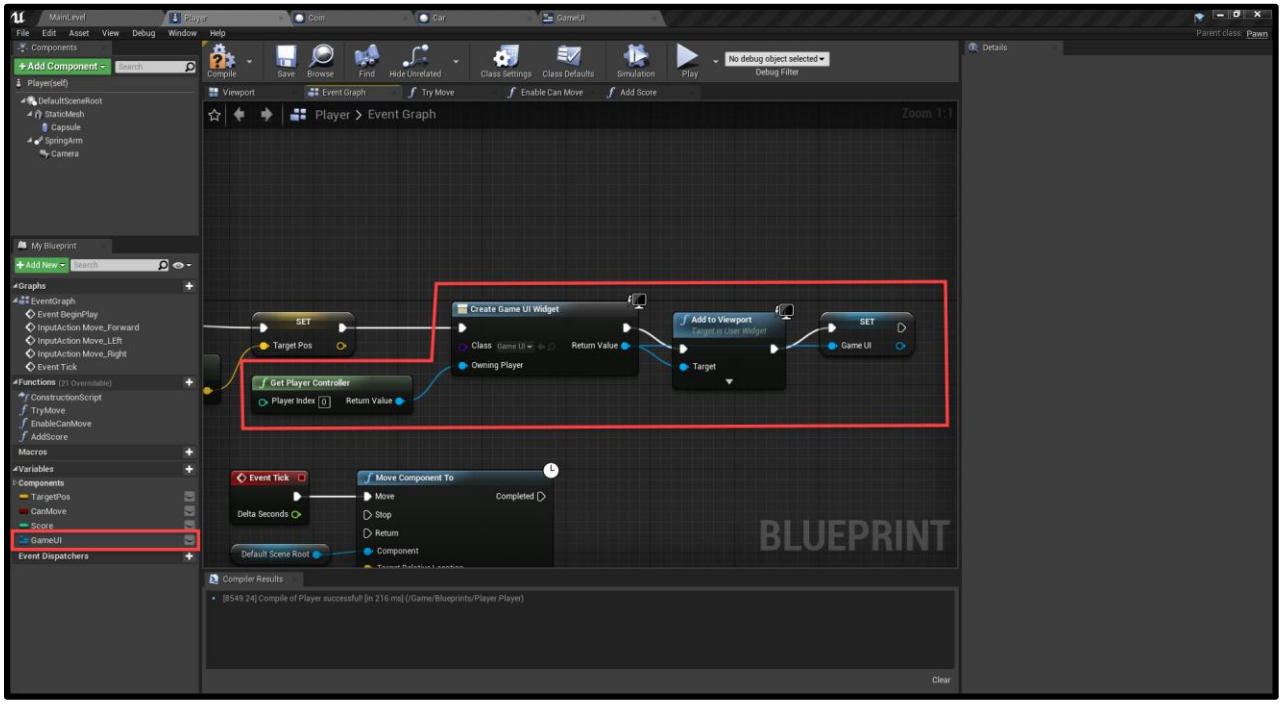


Then we have the **SetWinScreen** function which just sets the win screen panel to be visible.

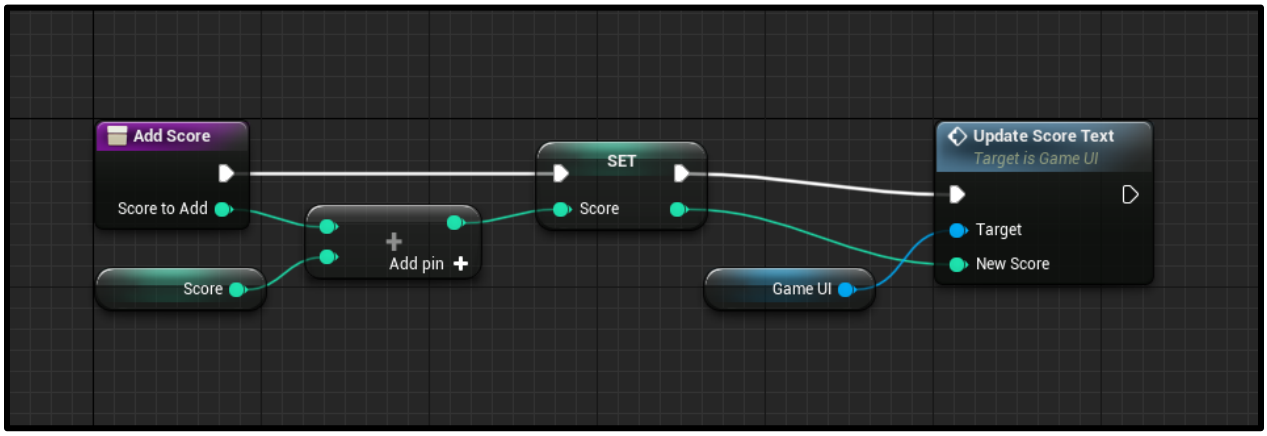


Back in the **Player** blueprint, we can initialize the UI. First, create a new variable of type *GameUI* called **GameUI**.

Then continuing on the EventBeginPlay flow, create and initialize the UI widget.



So inside of the **AddScore** function, we can update the UI.



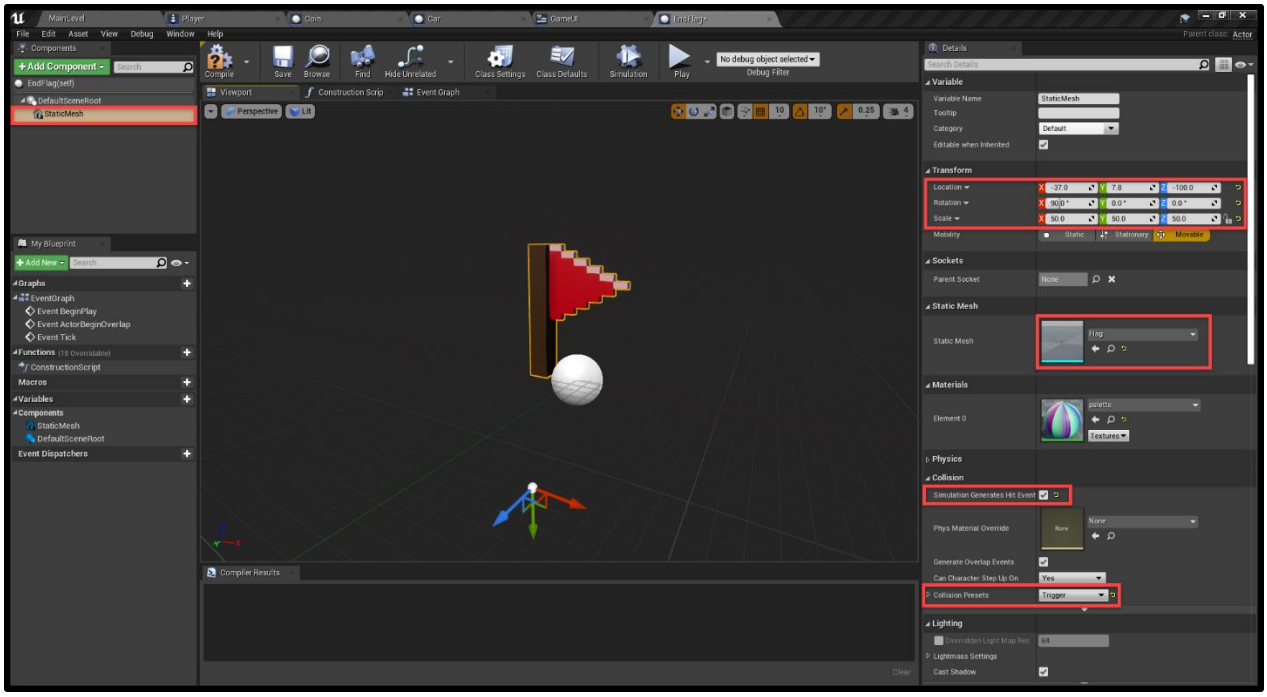
If you press play, you'll see it in action!

End Goal

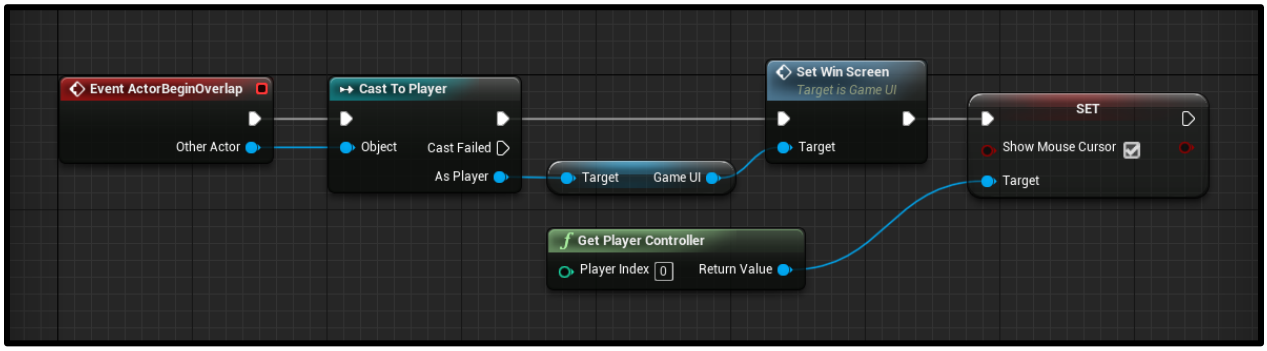
To finish the project off, we're going to create a flag at the end of the level that the player needs to reach. This will enable the win screen, prompting them to restart the game. So in the *Blueprints* folder, create a new blueprint of type *Actor* called **EndFlag**.

All we're going to do component wise is add in a static mesh.

- Set the **Static Mesh** to *Flag*
- Set the **Location** to *-37.0, 7.8, -100.0*
- Set the **Rotation** to *90, 0, 0*
- Set the **Scale** to *50, 50, 50*
- Enable **Simulation Generates Hit Events**
- Set the **Collision Presets** to *Trigger*

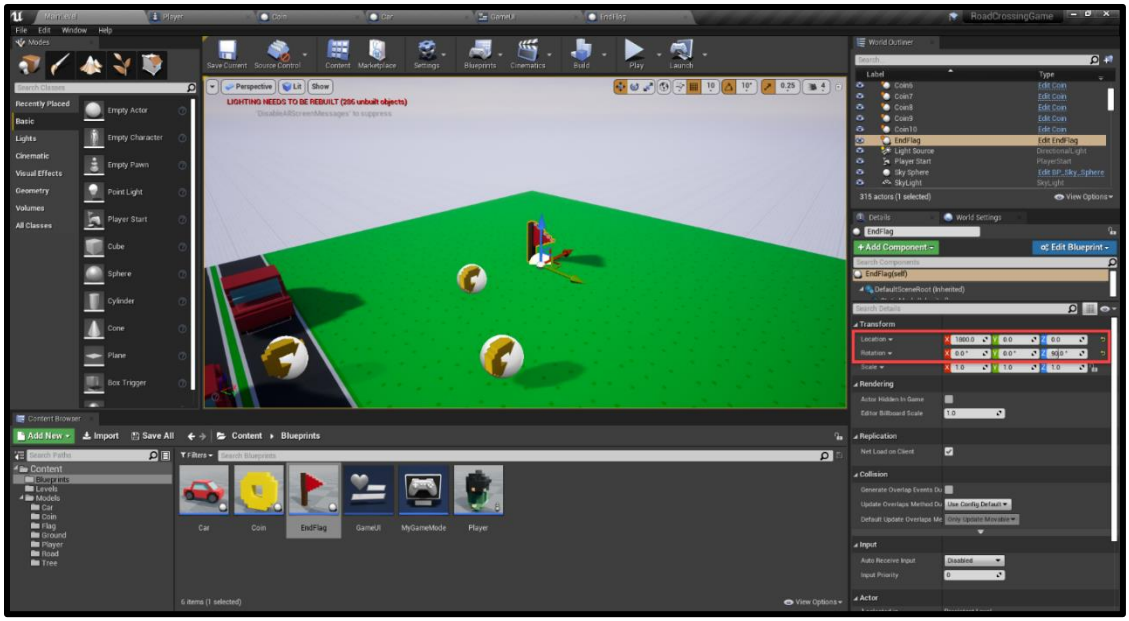


Over in the **Event Graph**, we just want to check when the flag has collided with something. If it's the player, enable the win screen. Also enable the cursor.



Back in the level editor, let's drag in the flag and position it at the end of the level.

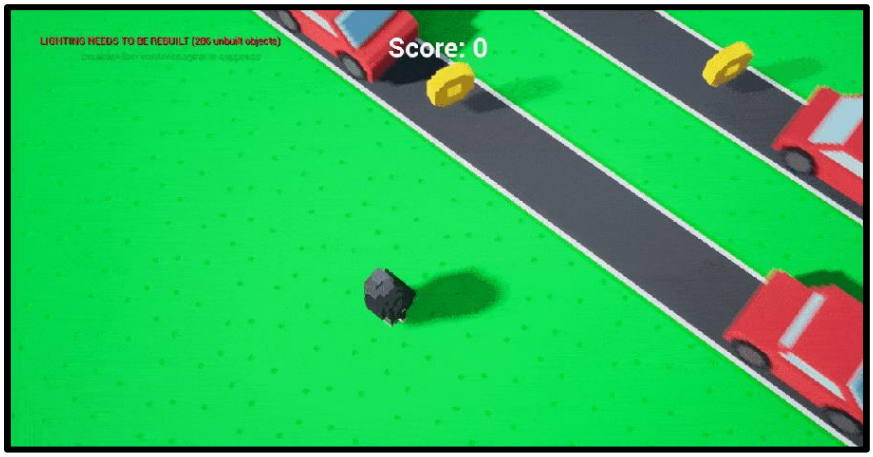
- Set the **Location** to *1800, 0, 0*
- Set the **Rotation** to *0, 0, 90*



Conclusion

There you have it! You now have a complete road-crossing, arcade game made with Unreal Engine! Over the course of this tutorial, we've learned to set up everything from coin collection to the constant flow of forward-moving cars to create obstacles for the player. All of this was accomplished with Unreal Engine's Blueprinting system, which handles all our game logic - including win conditions!

From here, you could expand the game by adding in more obstacles, collectibles, or even more unique levels. You can also use these fundamentals to create an entirely new game with your new-found knowledge. Whatever you decide, good luck, and we hope you've learned a lot about building games with Unreal Engine!



How to Create an Action RPG in the Unreal Engine

Introduction

Welcome everyone! No matter how many years pass, RPGs remain a popular genre. Many developers dream of making their own, whether to experiment with never-before-seen mechanics or a simple desire to tell a deep and interactive story. Given Unreal Engine's amazing graphical capabilities, it is also a top engine choice for many pursuing this in order to capture the unique aesthetic style they're after.

In this tutorial, we'll be taking the first steps on this path and show you how to create an action RPG with Unreal Engine. It will feature a third-person player controller, who can move, jump, attack and block, along with an enemy who will chase after the player and attack them. This tutorial will also cover the basics of setting animation transitions for that as well. The project shown here can be a great basis for a much larger game, or just a good way to learn many of the systems in the Unreal Engine. Regardless, if you're ready to start making your own RPGs, let's dive in.

If this is your first time using the engine, though, then we recommend you view our Unreal Engine Beginner's tutorial first.

Project Files

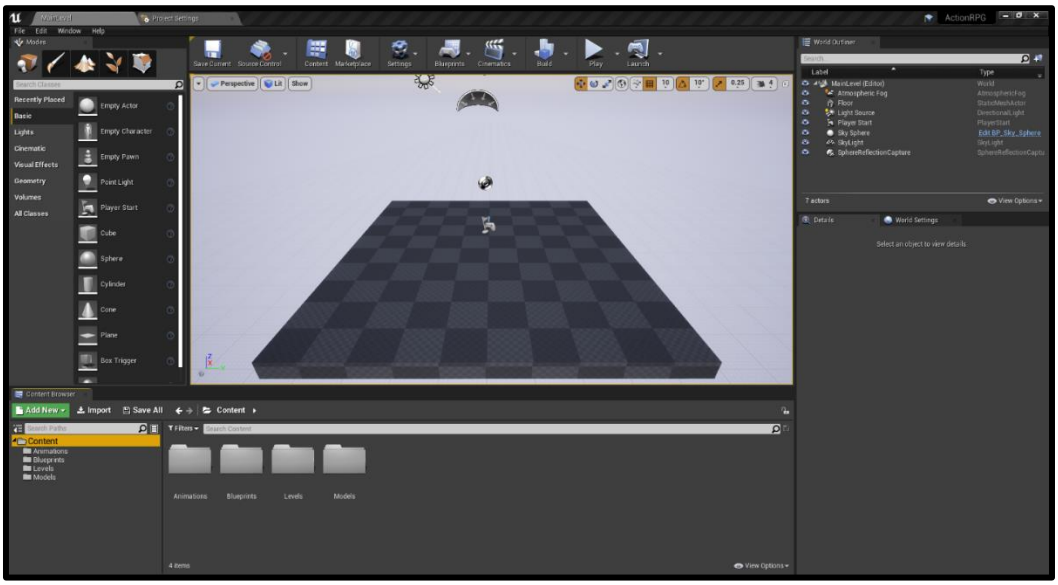
This tutorial will be using a few models and animations from Mixamo. You can choose to get your own, or download the ones we'll be using for the project. You can download the complete project to see how all the levels, models and blueprints interact here.

Creating the Project

To begin, let's create a new project (you don't need to include the starter content). When the editor opens up, let's create four new folders.

- Animations
- Blueprints
- Levels
- Models

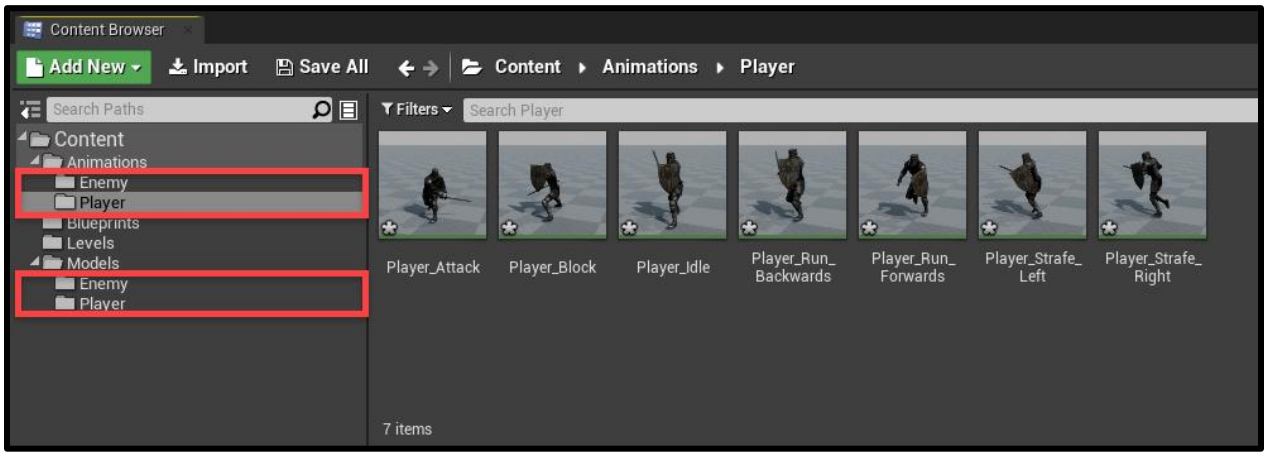
Save the current level to the *Levels* folder as **MainLevel**.



Next, download the assets from the beginning of the tutorial and extract them anywhere on your computer.

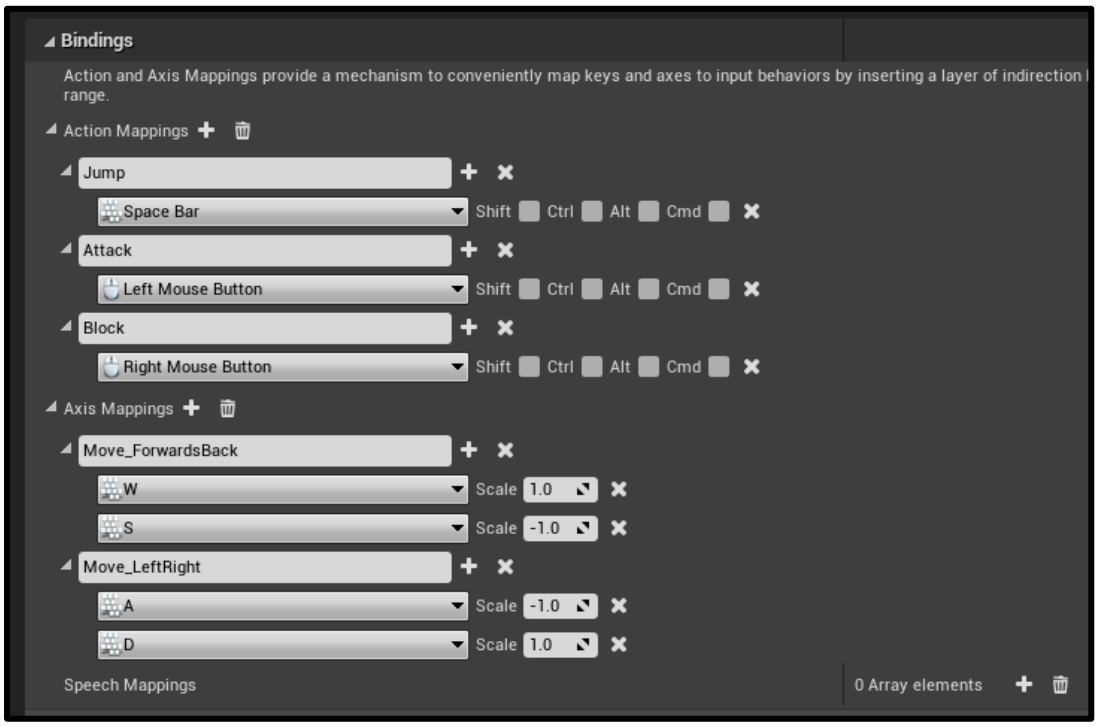
1. First, open the *Models Contents Folder* folder and drag the contents into our project's **Models** folder.
    a. Import all those assets when promoted to.
2. Then do the same for the *Animations Contents Folder*.
    a. When asked to select a skeleton for the animation, choose either the player of enemy one depending on the animation.

You should now have both models and all the enemy and player animations needed for the tutorial.



Before we start creating the player, we'll need to setup the control inputs. In the **Project Settings** window (*Edit > Project Settings...*) go to the **Input** screen.

Here, you want to create three new *Action Mappings* and two new *Axis Mappings*. Fill them in as seen in the image below:



Creating the Player

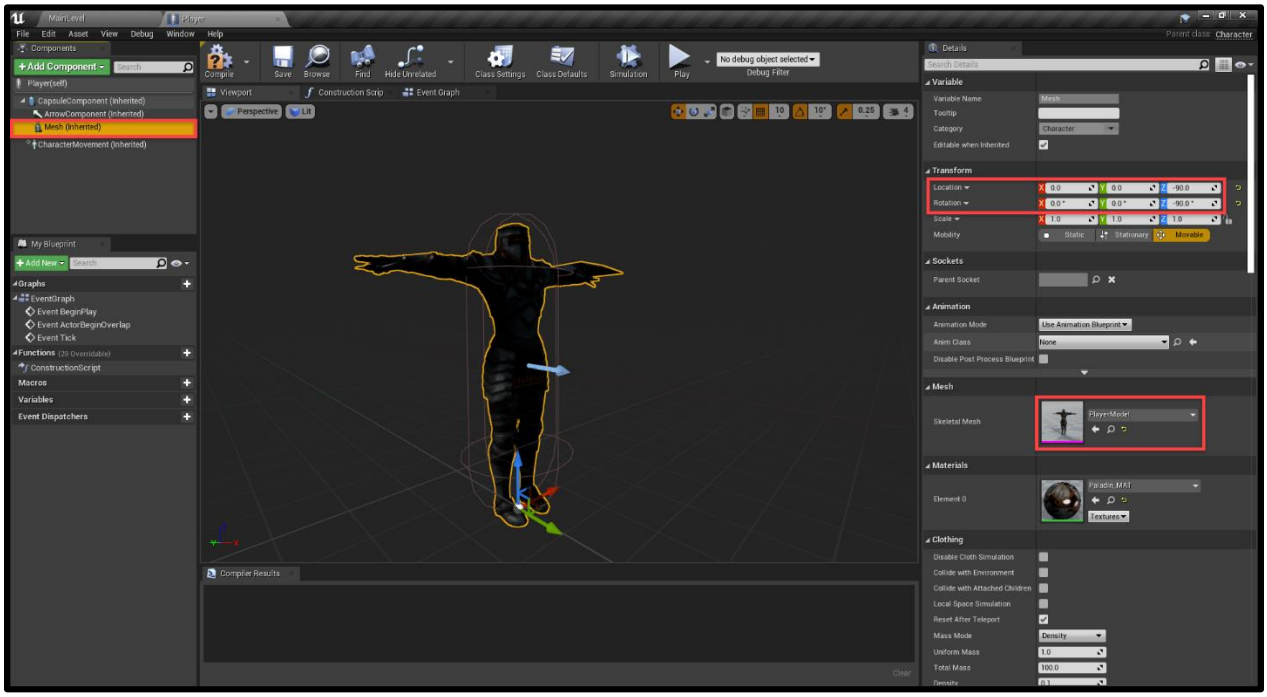
Time to create our player. In the **Blueprints** folder, create a new blueprint (parent class of *Character*) called **Player**. Double-click it to open the blueprint editor.

We have a few components already created for us.

- CapsuleComponent (collider)
    - ArrowComponent (defines the forward direction)
    - Mesh (our player's skeletal mesh)
- CharacterMovement (movement, jumping, etc)

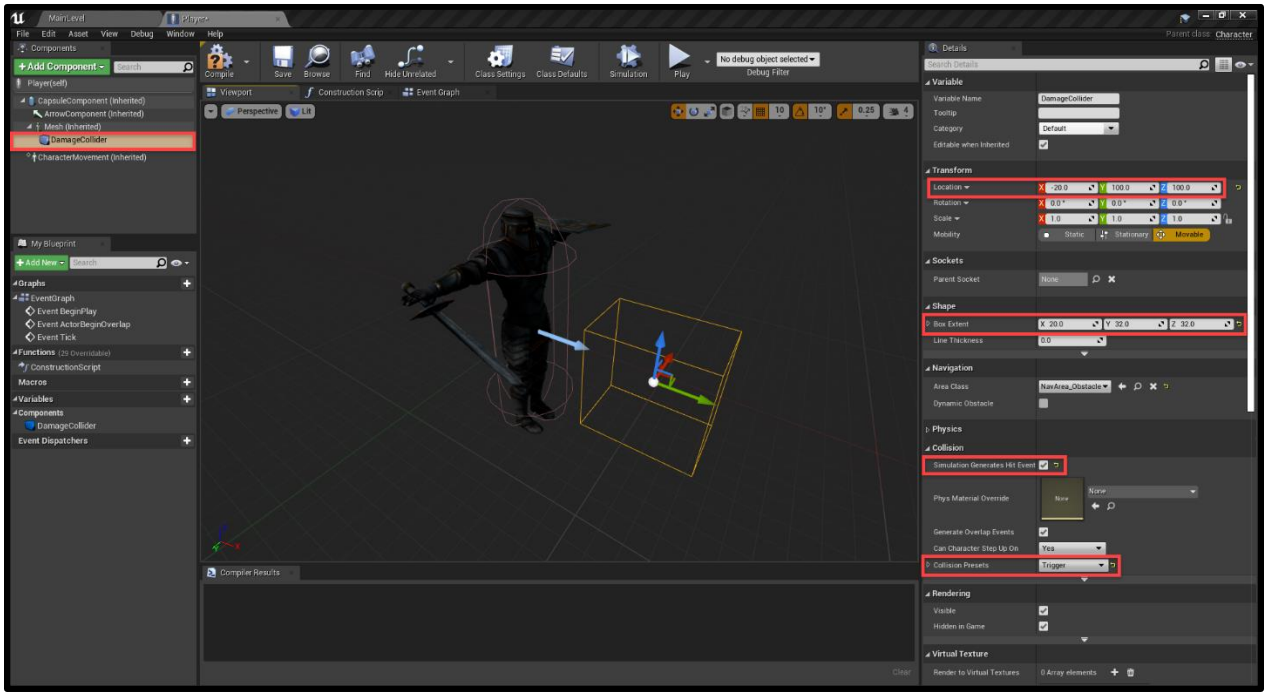
First, select the **Mesh**.

- Set the **Skeletal Mesh** to *PlayerModel*
- Set the **Location** to *0, 0, -90*
- Set the **Rotation** to *0, 0, -90*



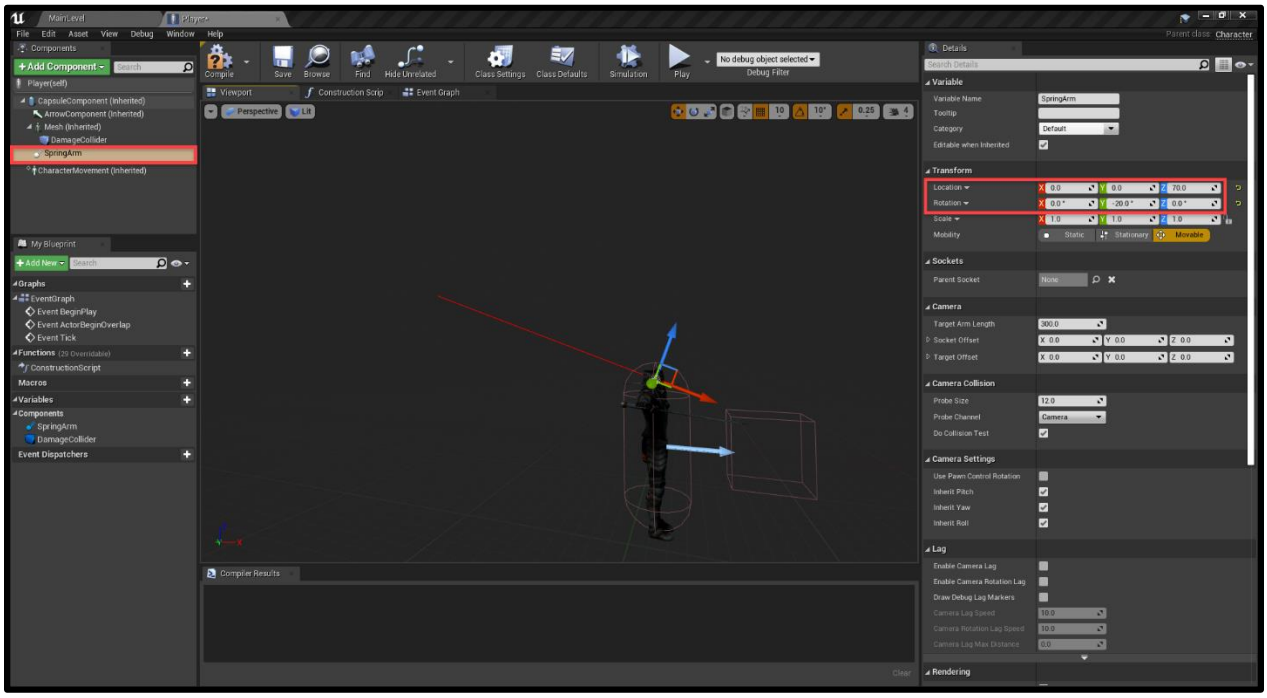
When attacking, we'll need to check if we're actually hitting anything. This will be done with a damage collider. Create a new **Box Collision** component and make it a child of the mesh.

- Set the **Location** to *-20, 100, 100*
- Set the **Box Extents** to *20, 32, 32*
- Enable **Simulation Generates Hit Events**
- Set the **Collision Presets** to *Trigger*



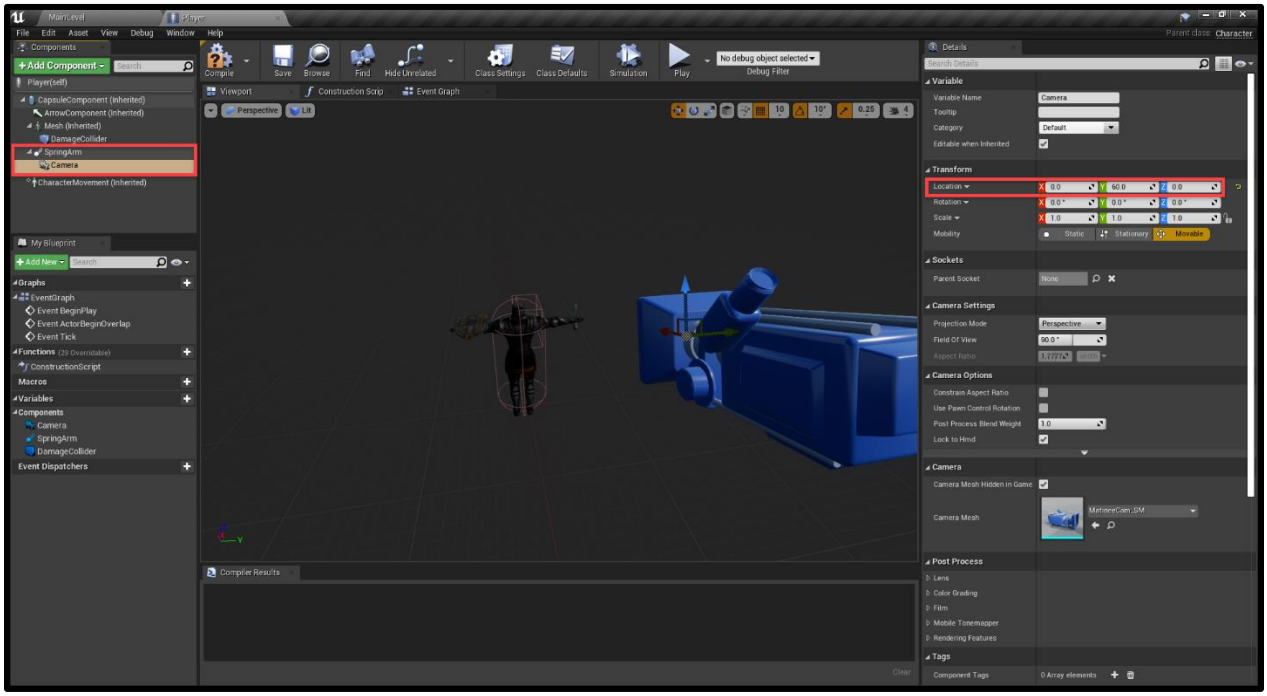
Next, we need to setup the camera. In order to have it orbit around our player, we'll first create a **Spring Arm** component. This holds its children at a distance.

- Set the **Location** to *0, 0, 70*
- Set the **Rotation** to *0, -20, 0*



As a child of the spring arm, create a **Camera** component.

- Set the **Location** to *0, 60, 0*

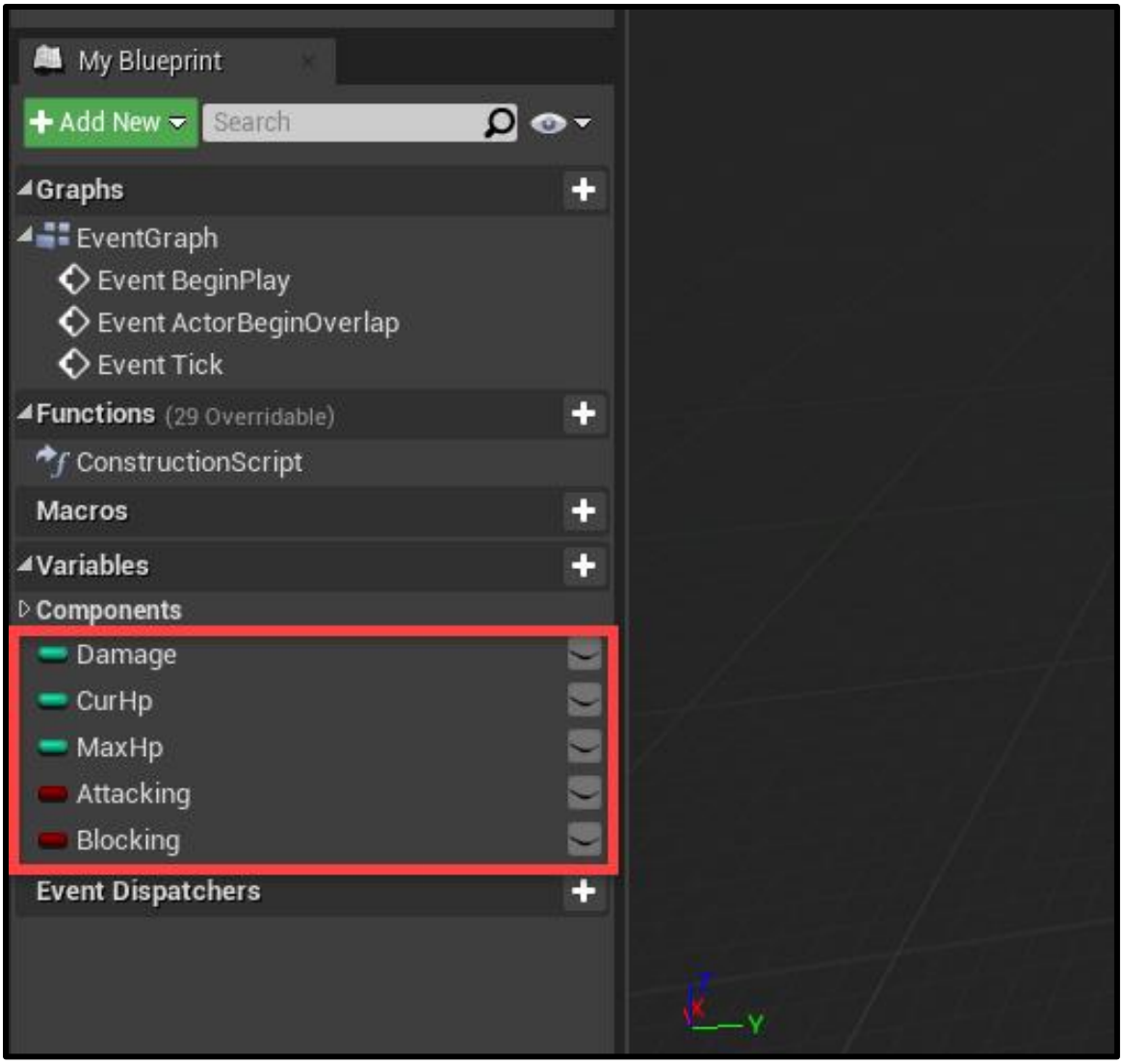


Now that we've got all of our components setup, we can begin with our variables.

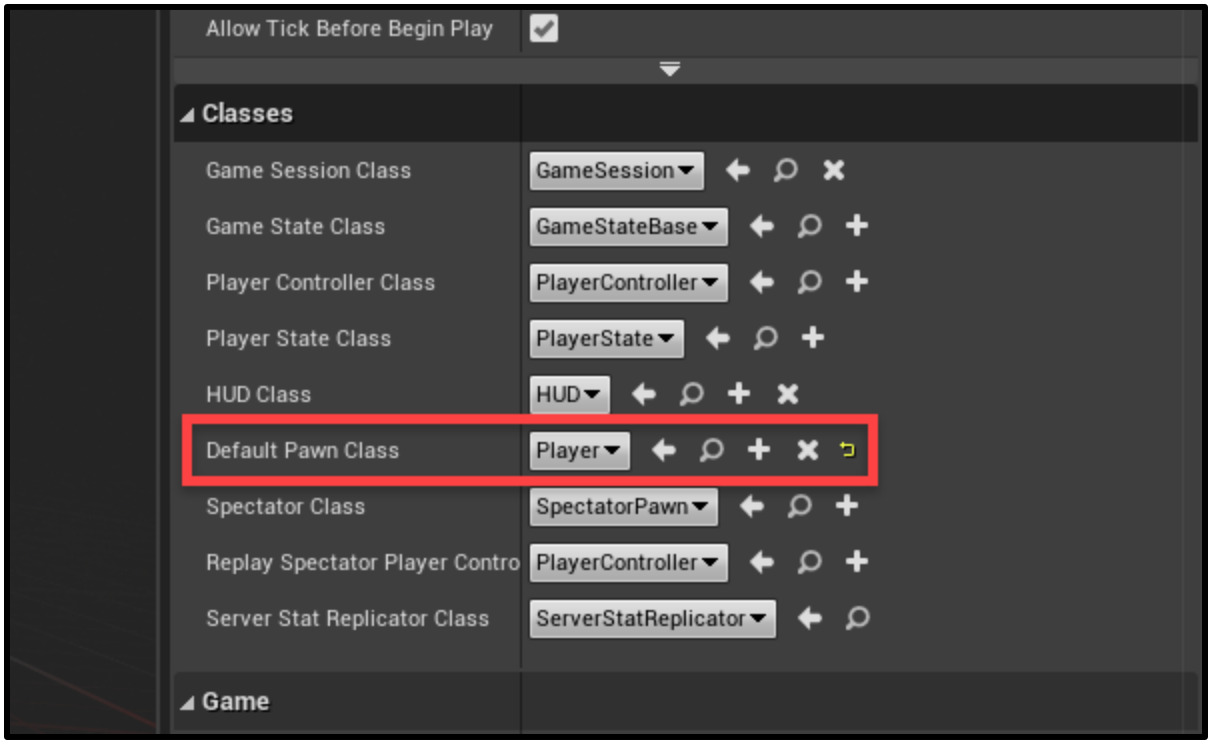
- Damage (Integer)
- CurHp (Integer)
- MaxHp (Integer)
- Attacking (Boolean)
- Blocking (Boolean)

You can then click the **Compile** button and fill in some default values.

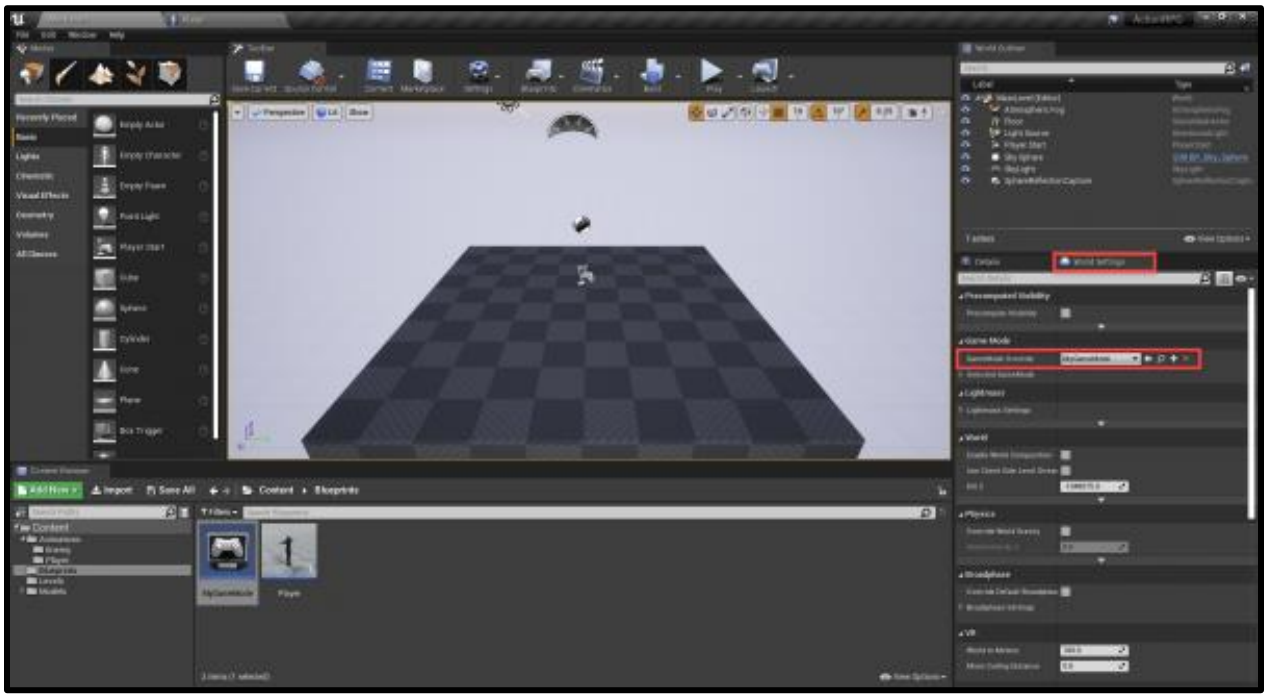
- Damage = 1
- CurHp = 5
- MaxHp = 5



Next, let's go back to the level editor and create a new blueprint. This is going to be of parent class *GameModeBase* and it's going to be called **MyGameMode**. Open it up, and all we're going to do here is set the **Default Pawn Class** to *Player*.



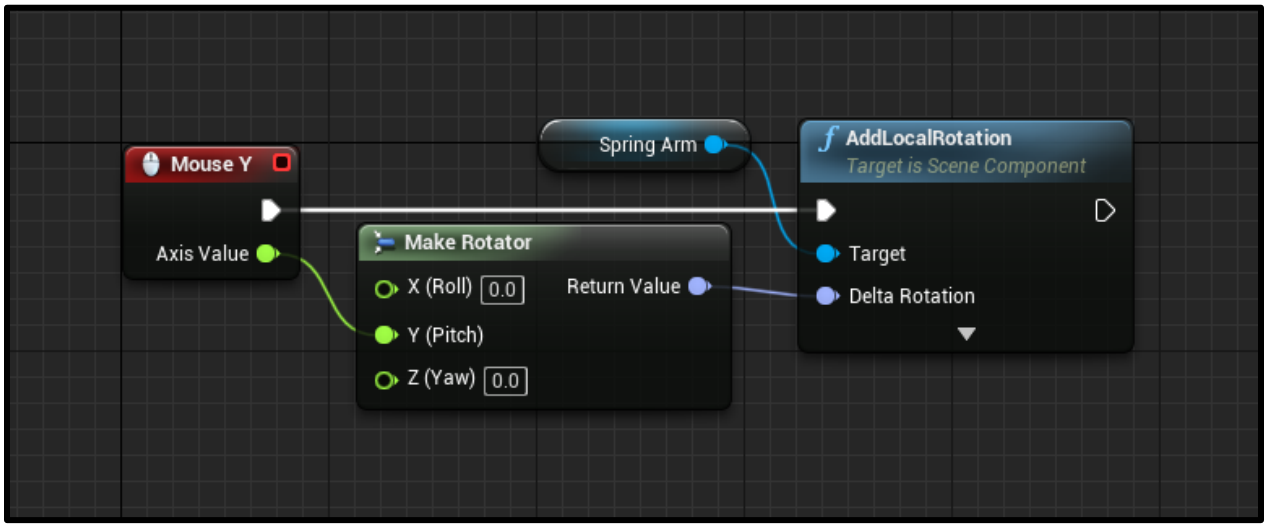
Back in the level editor, let's go to the **World Settings** panel and set the **GameMode Override** to our *MyGameMode*. Now when we press play, the player should spawn.



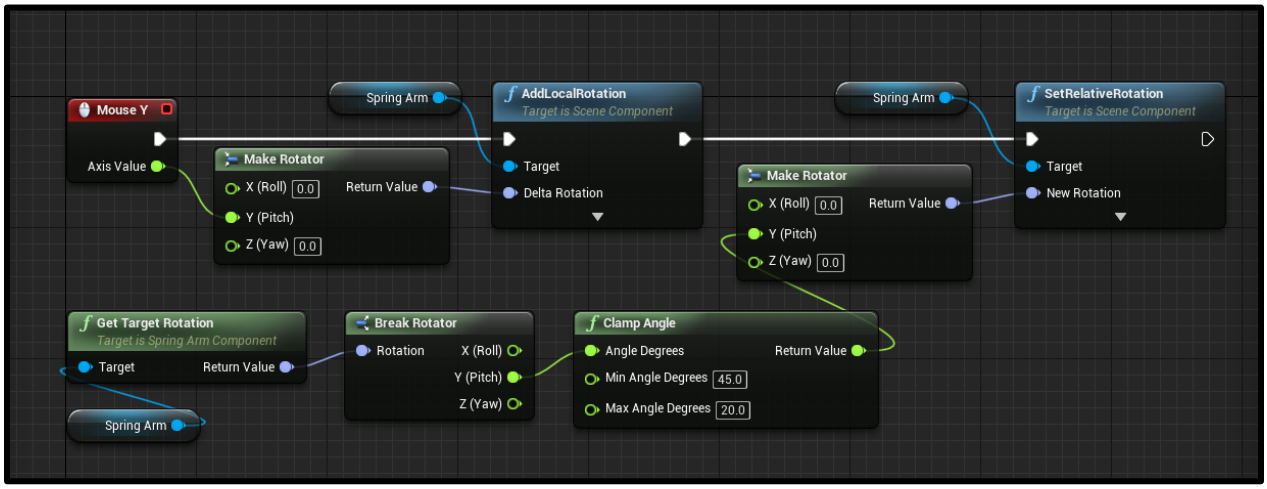
Camera Orbit Logic

Let's go to our player's event graph and begin by implementing the camera orbit functionality. This will allow us to move the mouse around, causing the camera to move around the player like in a third-person game.

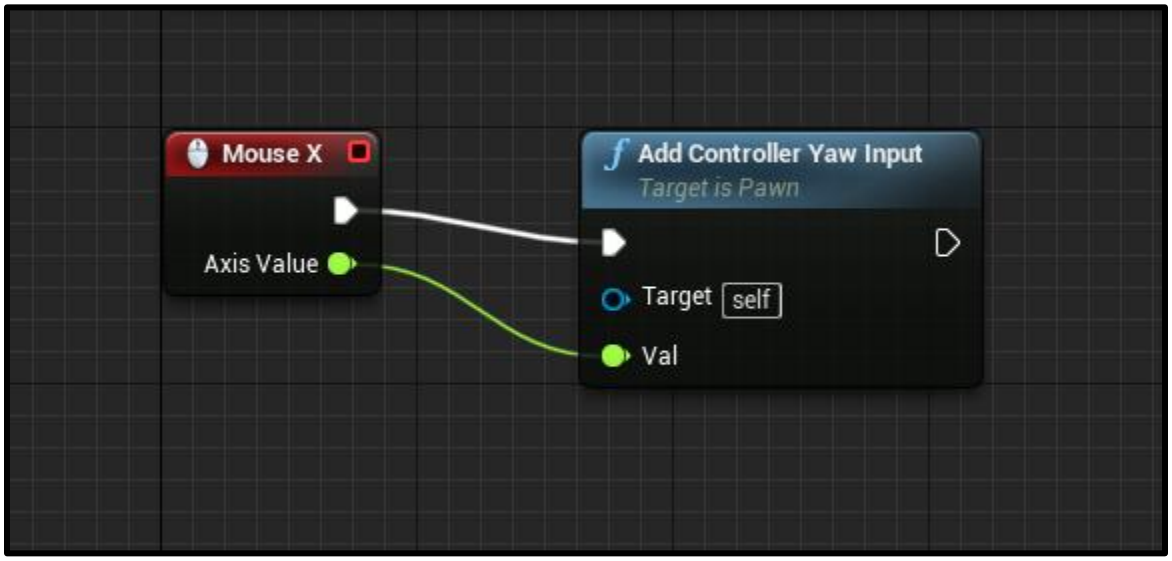
The **Event Mouse Y** node gets triggered when we move the mouse up and down. We're creating a rotation along the Y axis and adding that to the spring arm. If you press play, you should see it in work.



One problem though, is that we can rotate all the way around. What we need to do, is clamp the rotation. Prevent it from moving too far up and too far down. We'll be using the **Clamp Angle** node to clamp the Y rotation axis. Now if you press play, you'll see that there's a limit to where you can rotate the camera vertically.



For the horizontal rotation, we can just use the **Event Mouse X** node and plug that into the **Add Controller Yaw Input** node. This will automatically rotate the player vertically.



We now have a fully working camera orbit.

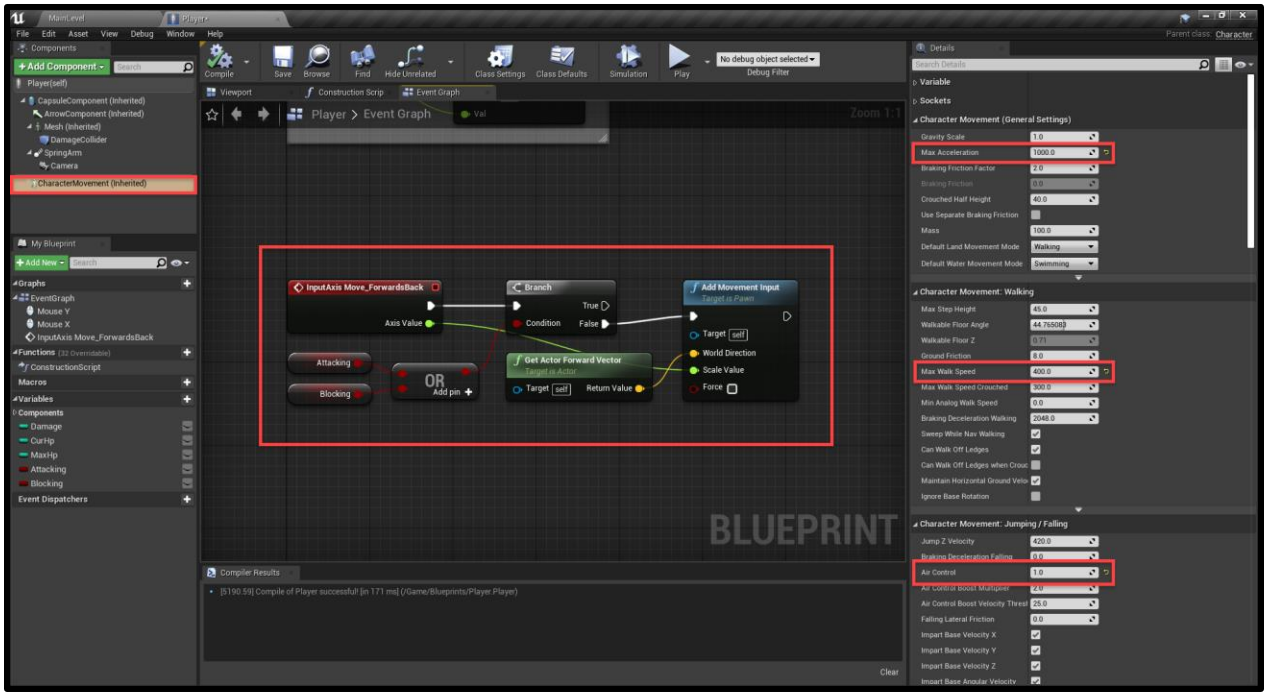
Moving the Player Around

For the player movement, we're first going to be checking for when the player is using the *Move_ForwardsBack* input axis. We'll then check if they're currently attacking or blocking. If not, then we'll move them in the respective direction.

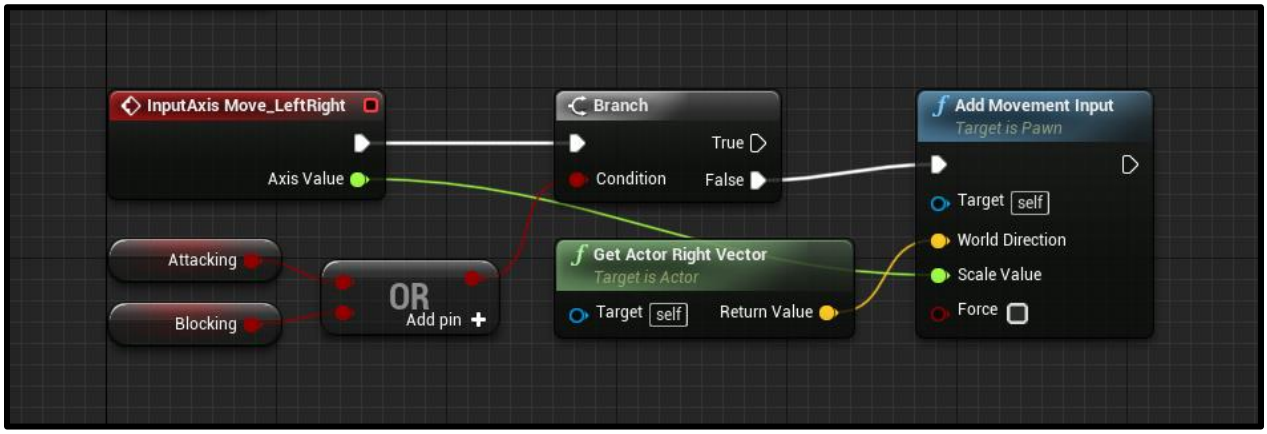
We can also select the **CharacterMovement** component to change some of the movement properties.

- Set the **Max Acceleration** to *1000*
- Set the **Max Walk Speed** to *400*
- Set the **Air Control** to *1.0*

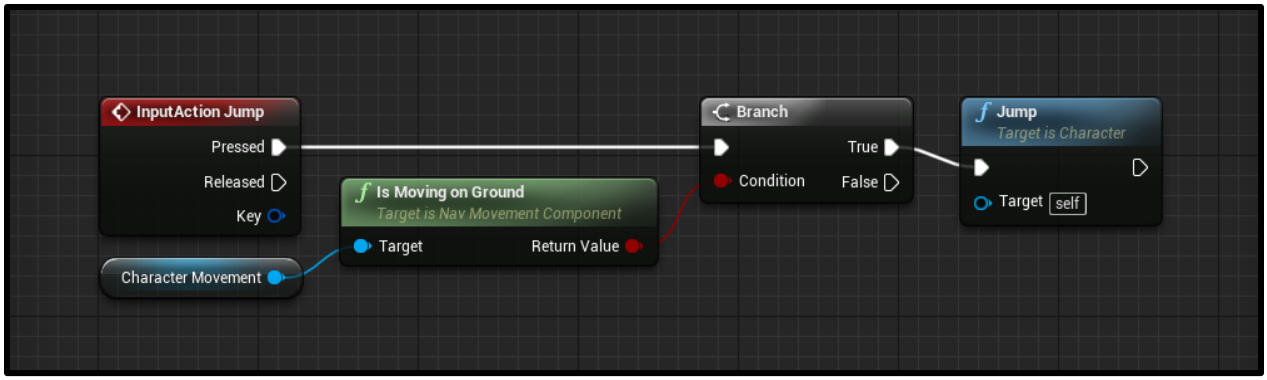
Now if you press play, you should be able to move forwards and back relative to where you're facing.



Horizontal movement is basically the same, but we're checking for the *Move_LeftRight* input and setting the world direction to the player's right vector.

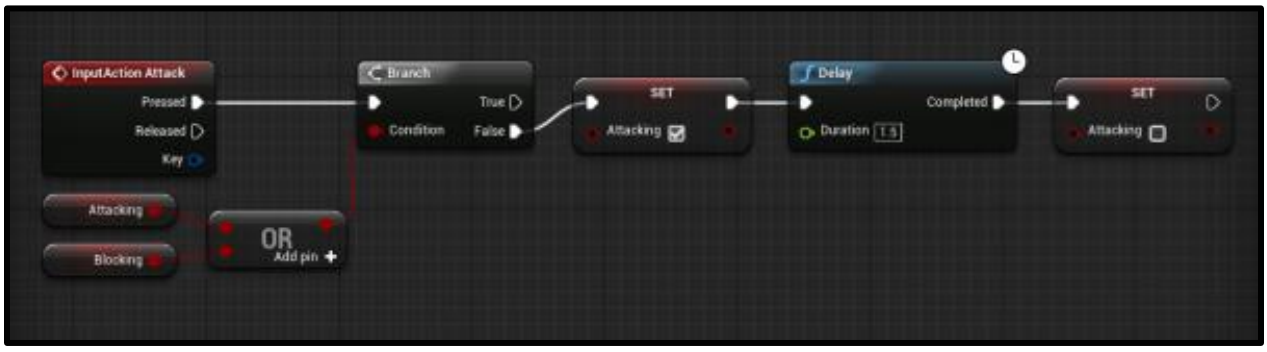


If you press play, you should see that we can now fully move around. Since we're using the character parent class, it comes with a bunch of pre-made things such as jumping. To implement jumping, we just need to check if we're on the ground, then trigger the jump node.

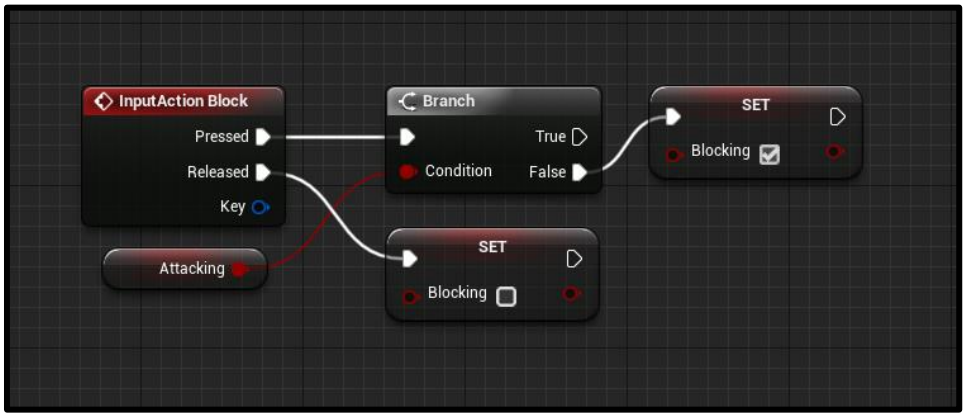


Attacking and Blocking

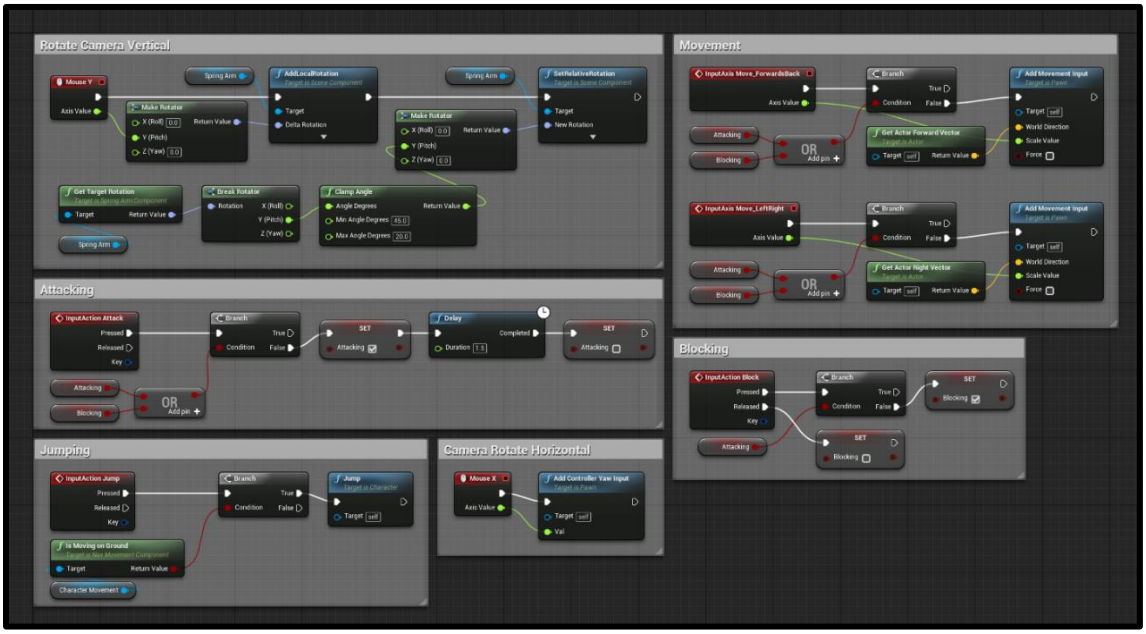
The player can attack and block incoming attacks. First, for attacking we'll check for the appropriate input and make sure we're not already attacking or blocking. Then we'll set the attacking variable to true, wait 1.5 seconds (duration of the attack animation) and set it back to false.



It's similar for blocking. We'll check for the input, make sure we're not attacking then set the blocking variable to true. When we let go of the button, we'll set it to false.

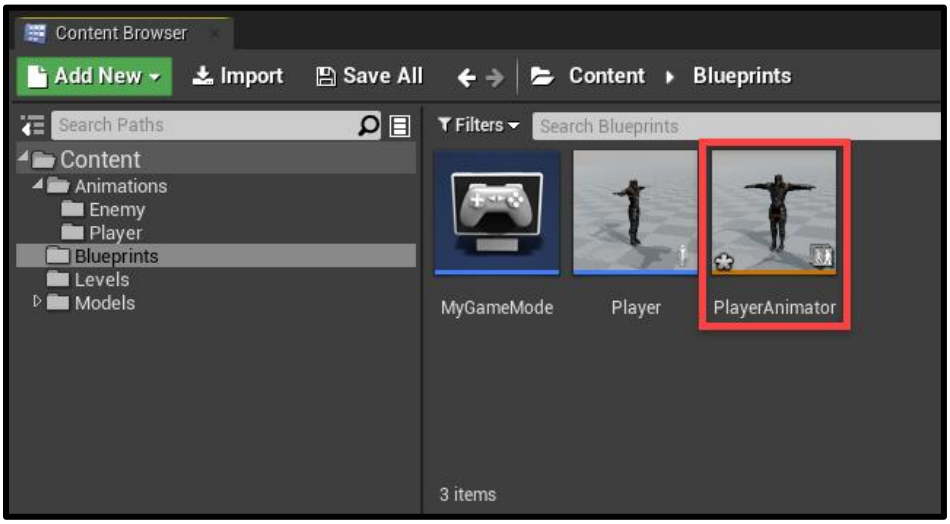


Here's an overview of everything you should have in the graph so far. I've commented the different sections to make it easier to read. You can do this by selecting a group of node, then right clicking them and clicking **Create Comment From Selection**.

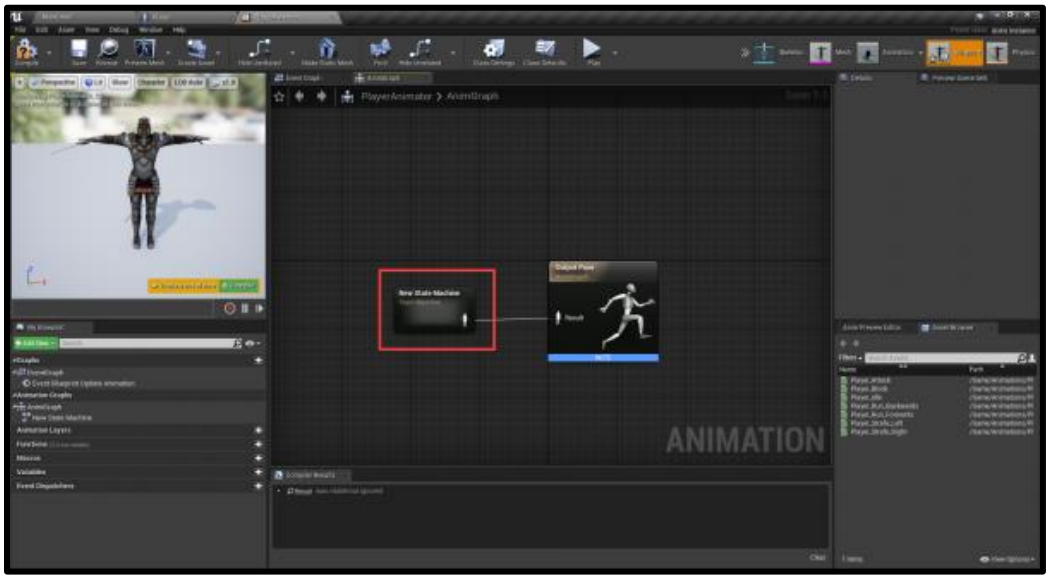


Player Animations

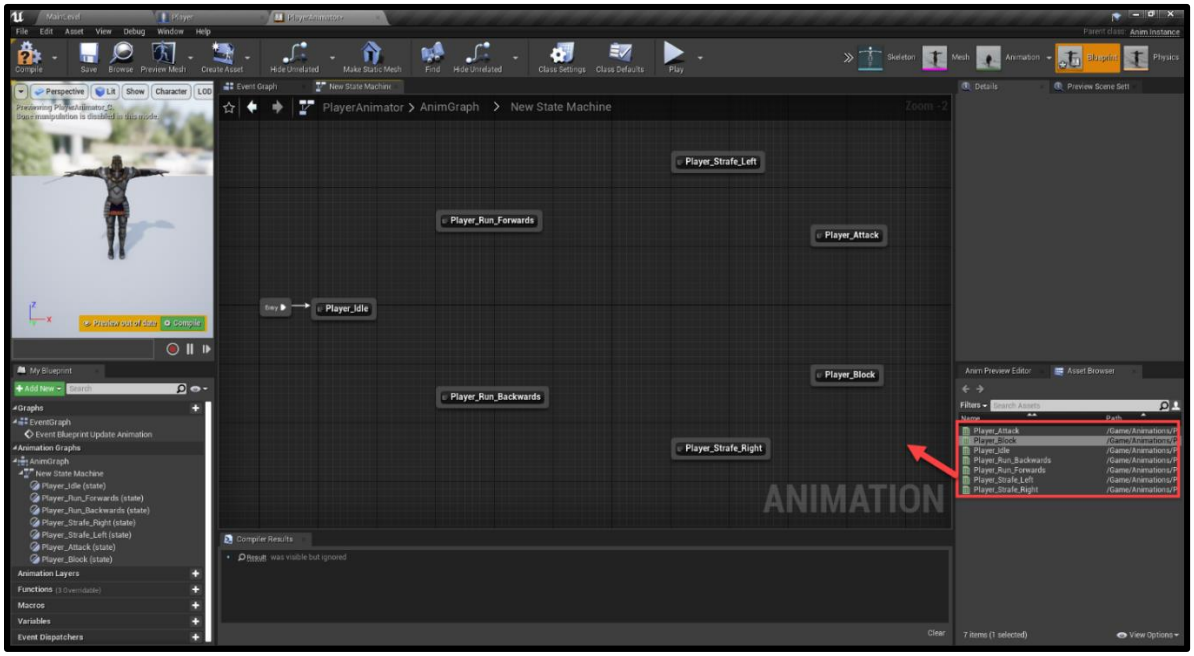
We've got our player setup but the model is static. To fix this, let's go to the *Blueprints* folder and create a new *Animation > Animation Blueprint*. When it asks for a skeleton, choose the player model. Call the blueprint **PlayerAnimator**.



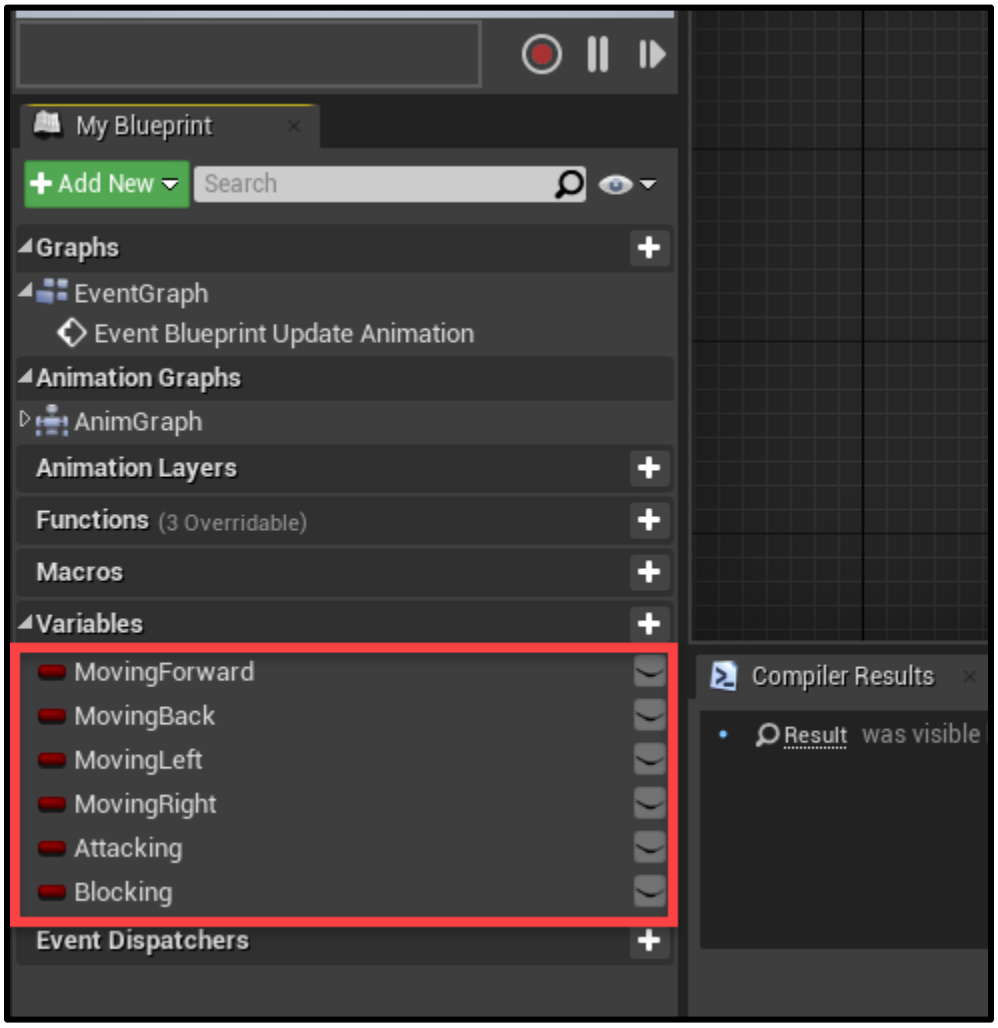
Double-click to open it up. First, we're going to create a new state machine (right click and create state machine node). Plug this into the **Output Pose**. A state machine basically determines what animation to play based on given inputs (are we moving? attacking? blocking?).



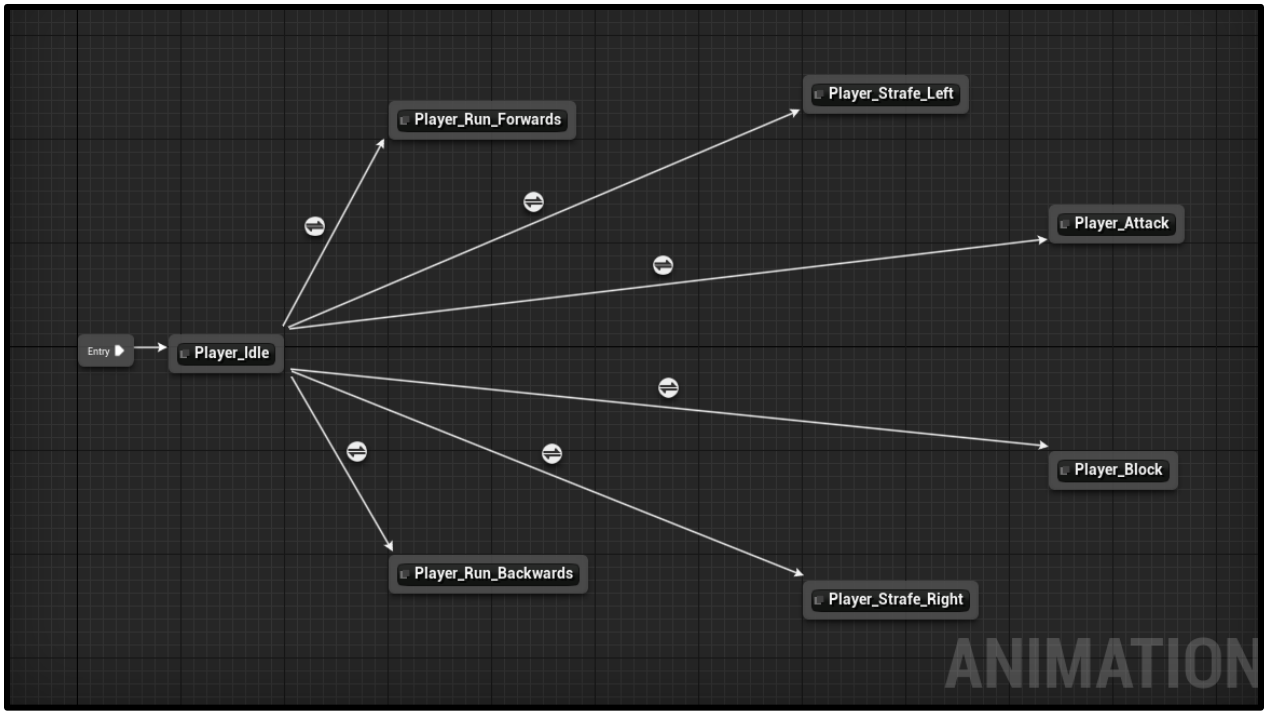
Double-click on the state machine to open it up. This is where we're going to connect the 7 animations we have. Drag them in like so, and connect the entry node to the idle animation.



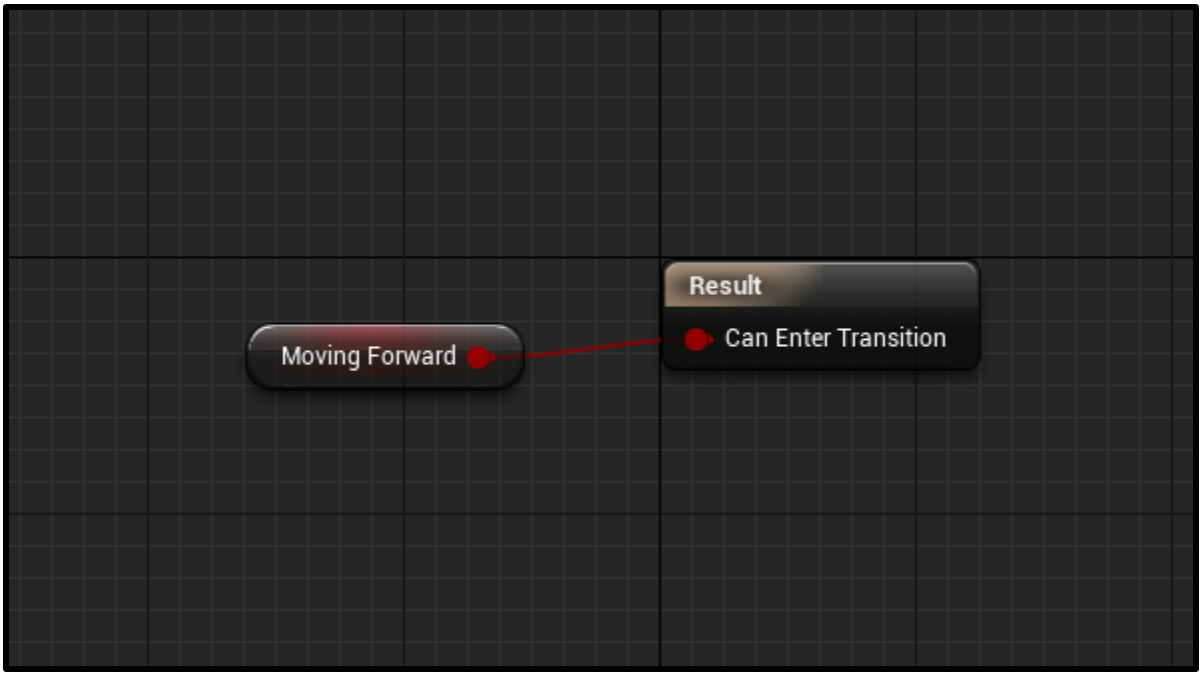
At the bottom left we can create the variables we'll be using. All these are booleans.



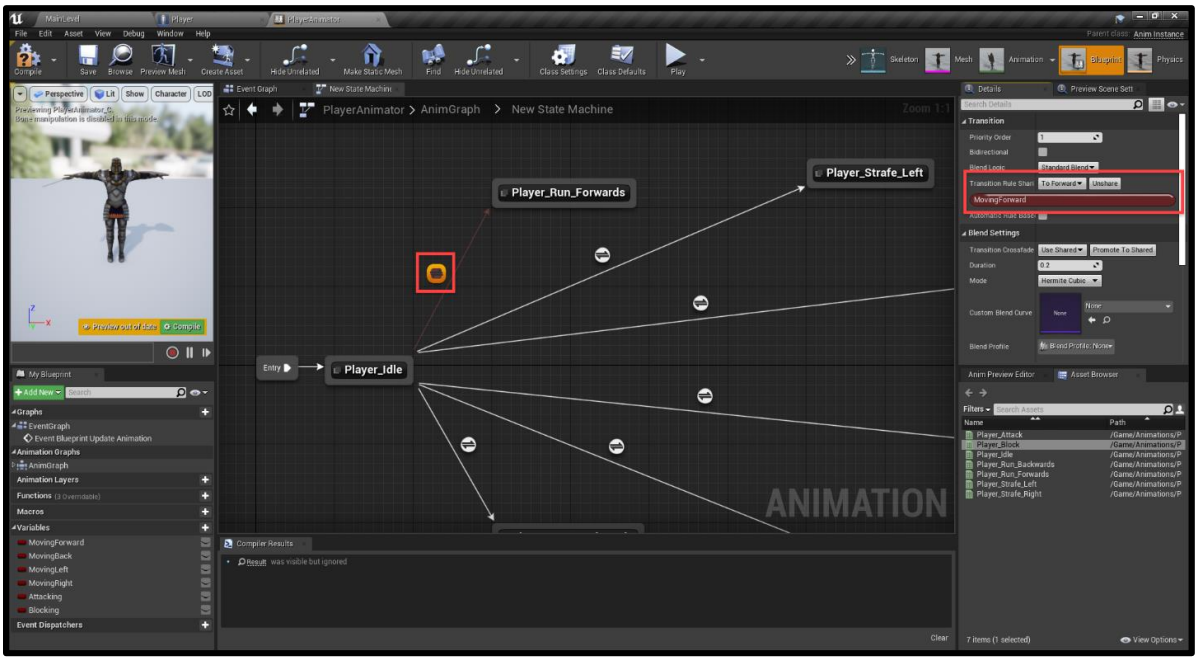
Let's start with the idle animation. Connect that to all other animations which we can transition to.



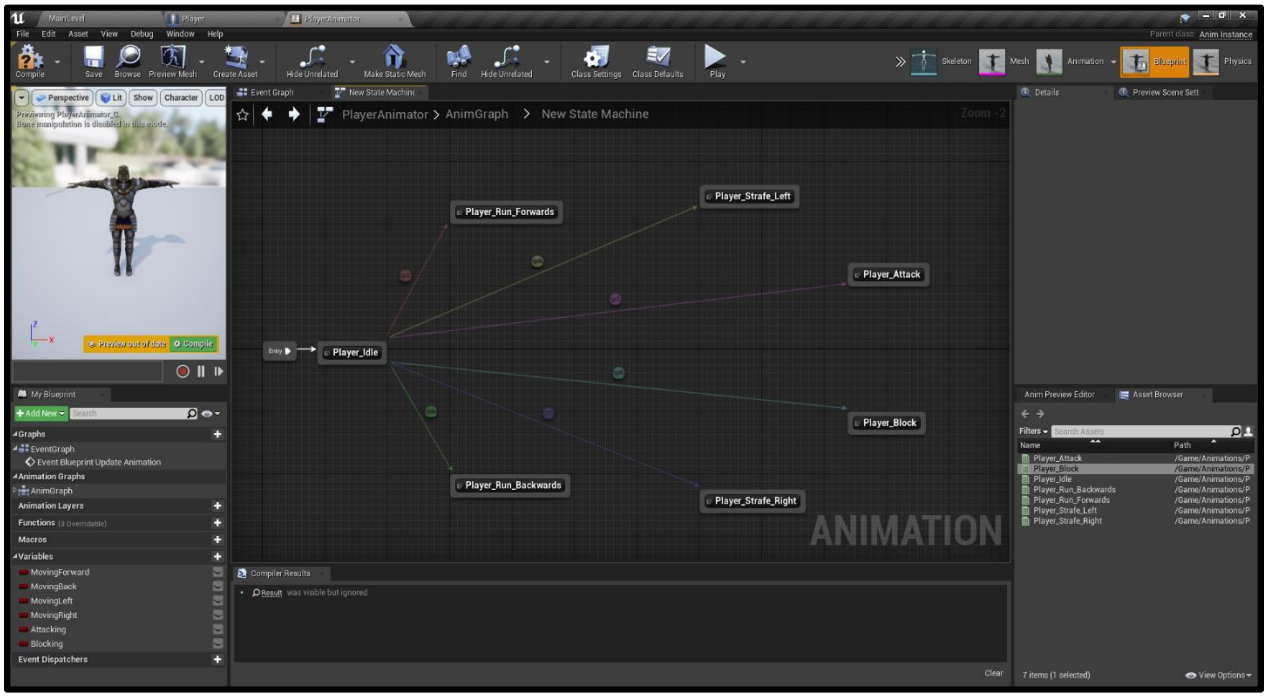
You'll see that each connection has a white circle button. Double-click on the run forwards transition, it will take us to a graph where we can setup a condition. This is going to return true or false for whether or not we can enter the transition.



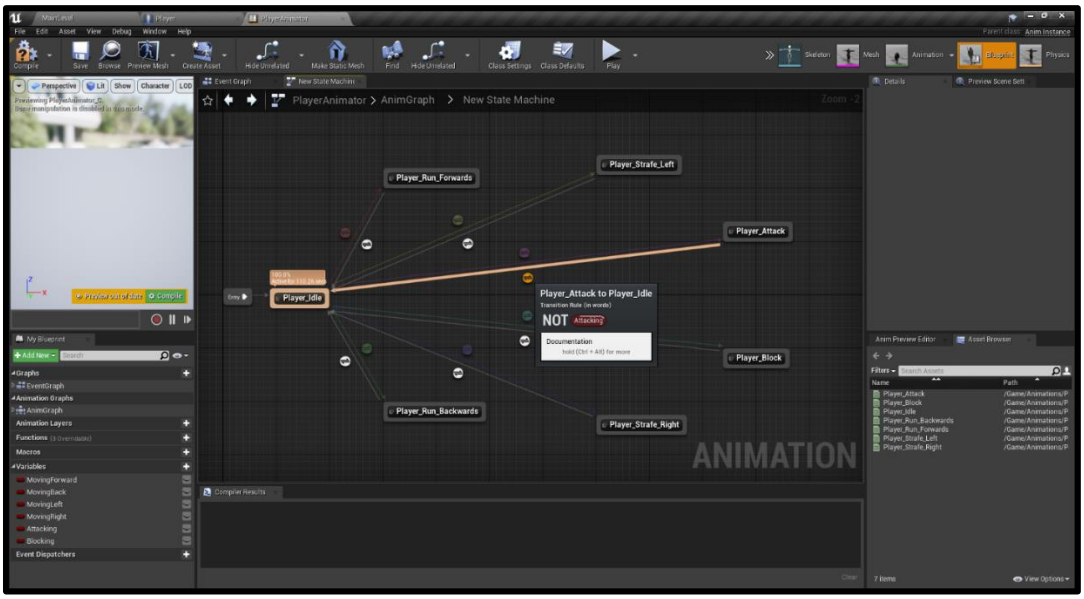
We can then go back to the state machine. Select the transition we just made and click **Promote to Shared**. Call it **To Forward**. This is basically a saved transition rule we can use again without needing to go back into the same graph.



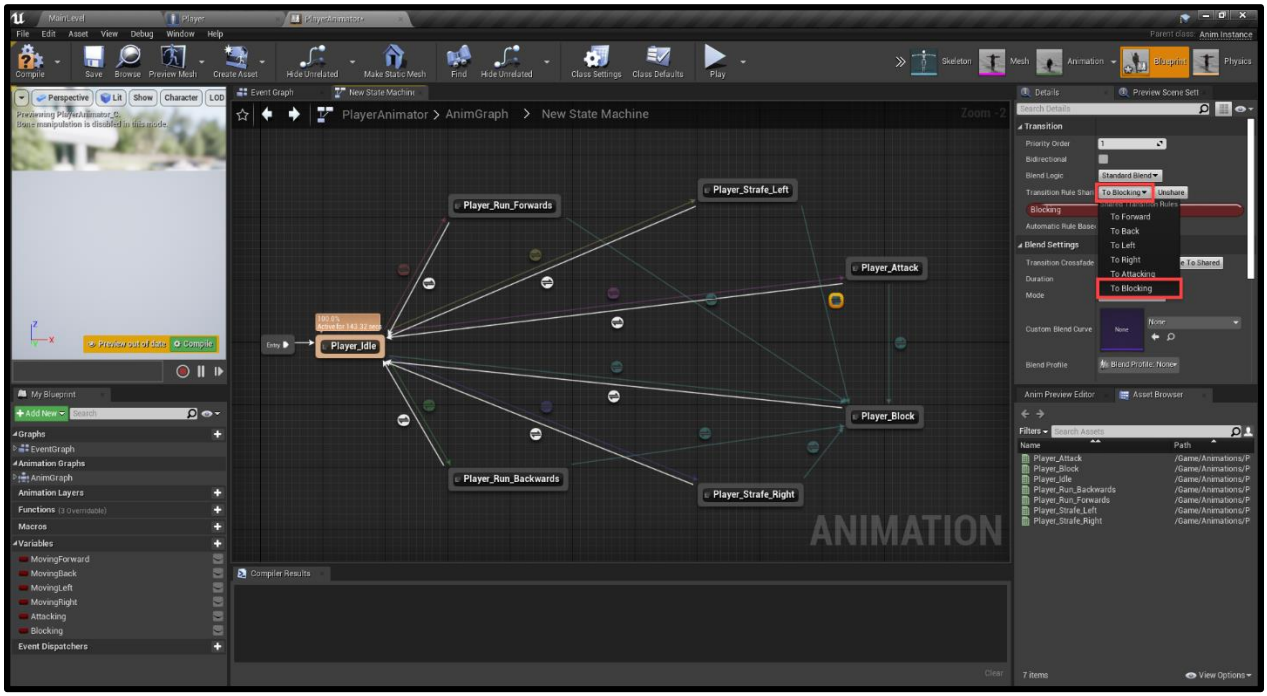
Let's go ahead and fill in the rest for the appropriate transitions.



We've got transitions from the idle animation, but what about back to it? Create a transition from each animation, back to idle. The condition is going to be the same as the one going to it, but with a not node in-between. Basically the opposite.

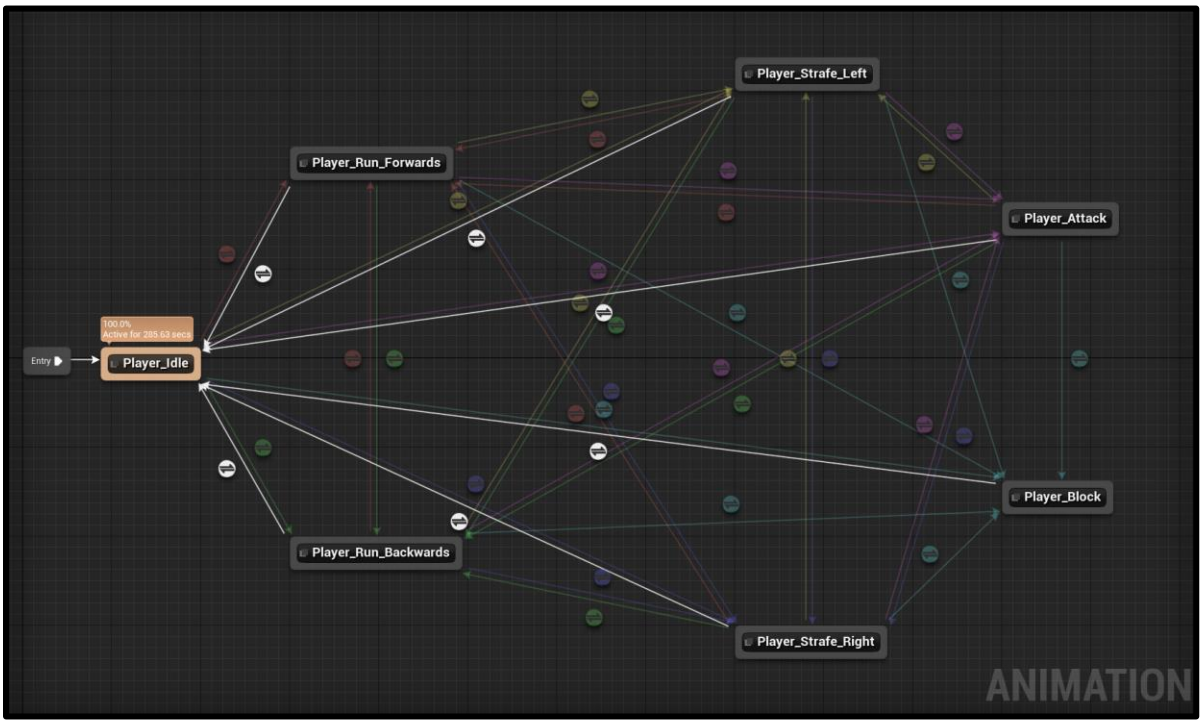


From here, all the animations are pretty much the same in the way they're connected, except for the **Player_Block** one. For that, we can only transition back to idle. So create a transition from all the other animations, using the **To Blocking** transition rule.

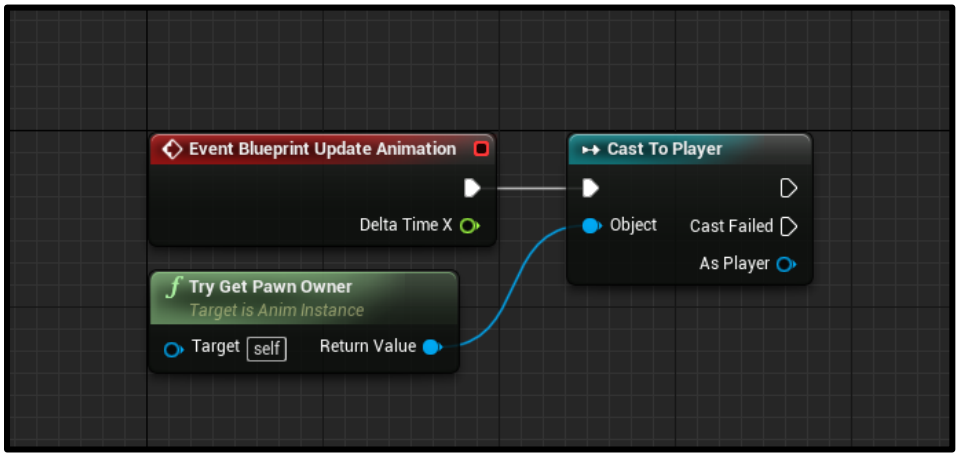


For the rest of the animations, we want to create a transition to and from every other animation like we already have with the idle animation. Just make sure that there's no return transition from the block animation.

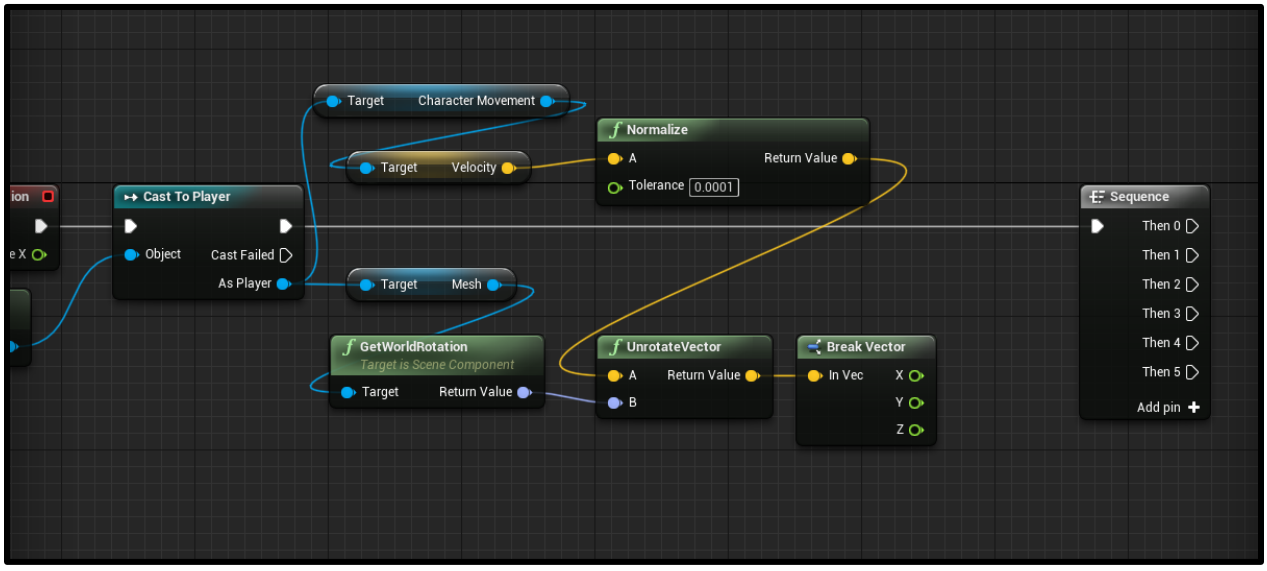
Here's what the final state machine should look like. Basically for each animation - think what animations can it transition to.



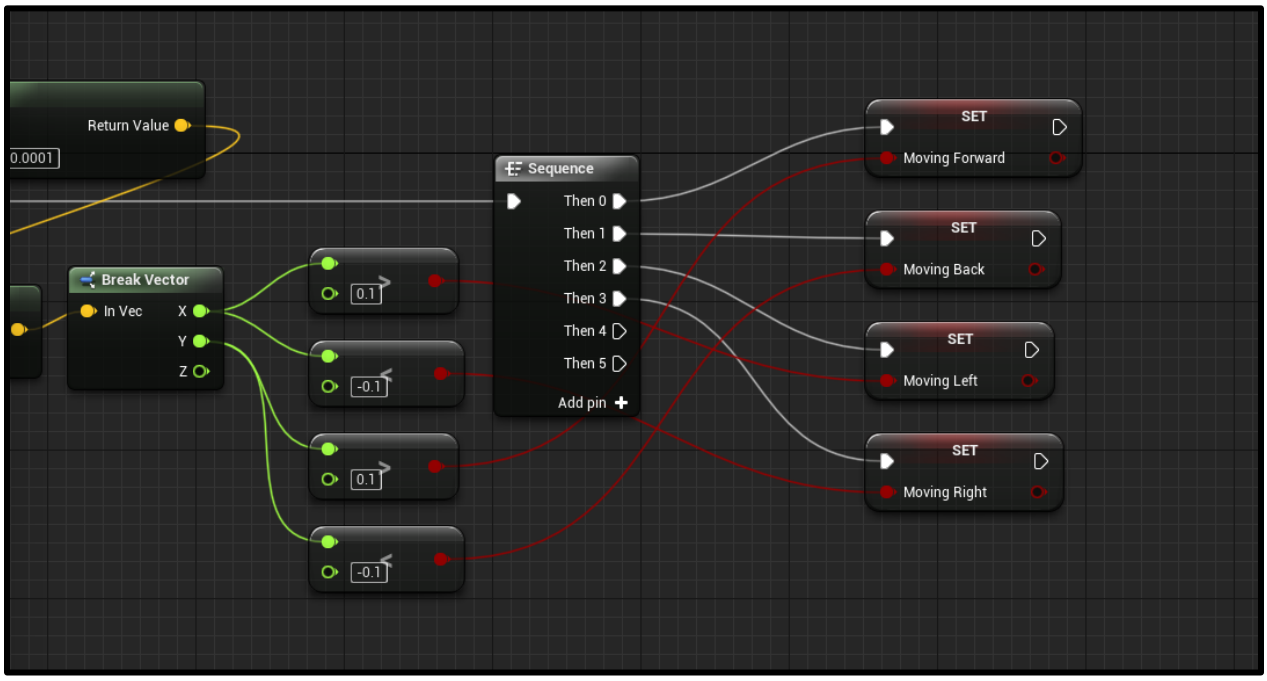
We've got the animations setup, but there's no logic behind setting the variables yet. Go to the **Event Graph** tab, and there are two nodes by default. Each frame, we want to cast the pawn owner to a player class so we can access its properties.



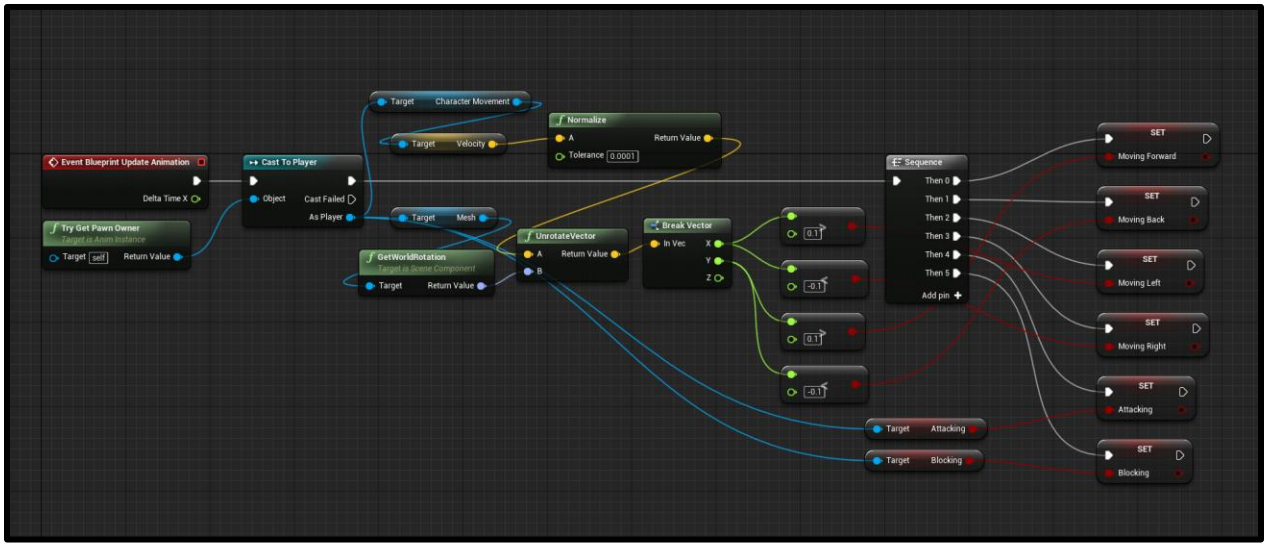
What we want to do is get the player's velocity. We'll use this to determine which direction we're moving in, so we can decide which animation to play. The sequence node can trigger a number of different nodes sequentially. Click **Add pin** so we have 5 outputs.



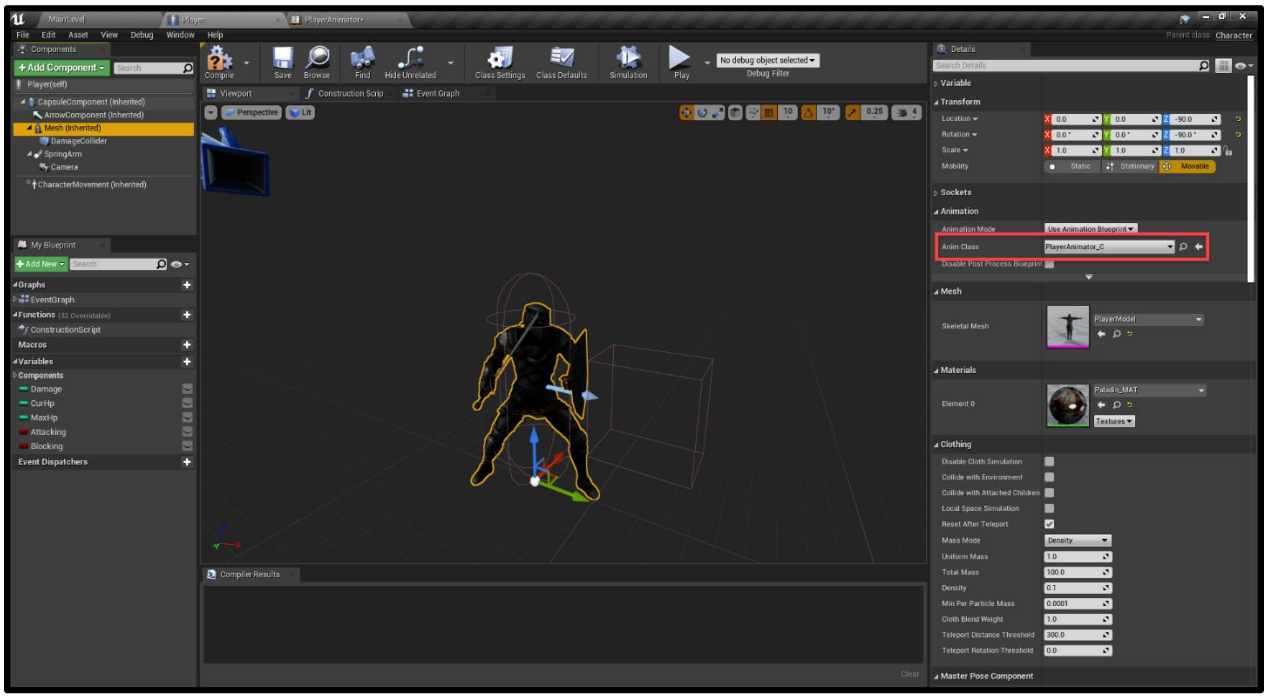
Here's how we want to set the moving booleans.



The attacking and blocking booleans will just be based on our player's respective variables.



Finally, back in the **Player** blueprint select the **Mesh** component and set the **Anim Class** to *PlayerAnimator*.



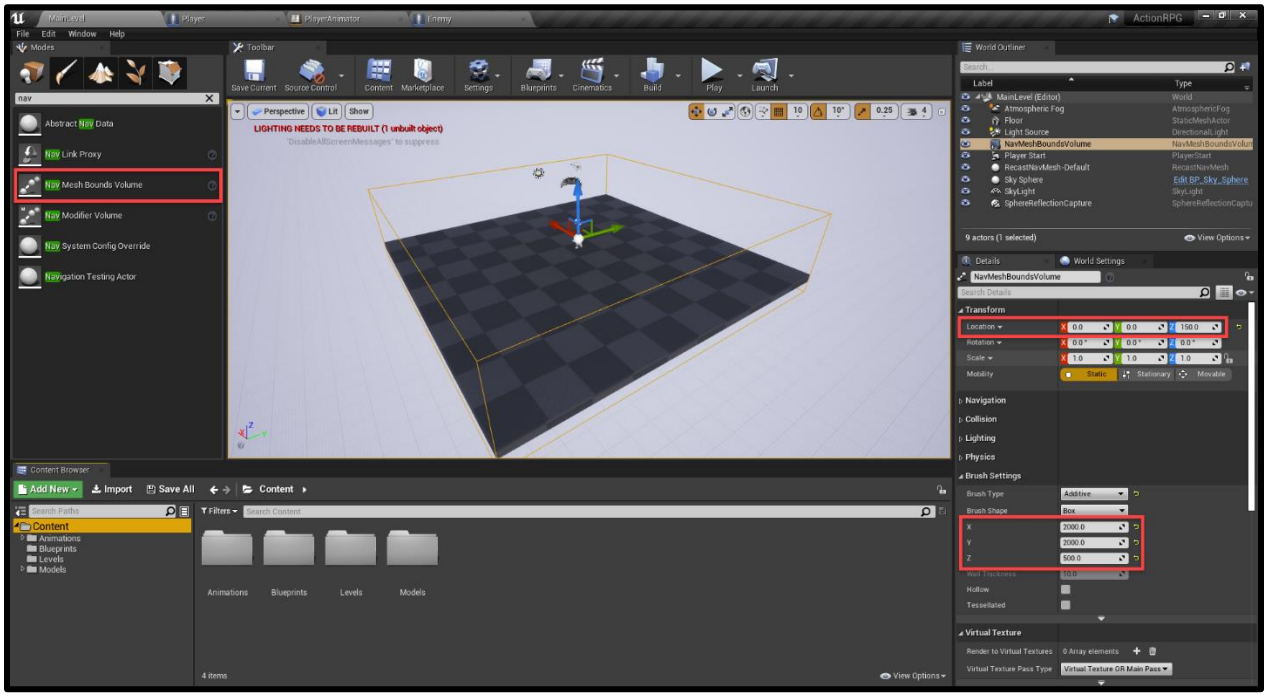
You should now be able to see the animations playing in-game!

Navigation Volume

Before we create the enemy, we'll need to setup the nav mesh. This allows an AI to move freely through an environment, navigate around and over obstacles. In the **Modes** panel, search for the **Nav Mesh Bounds Volume** object and drag that in.

- Set the **Location** to *0, 0, 150*
- Set the **X** and **Y** to *2000*
- Set the **Z** to *500*

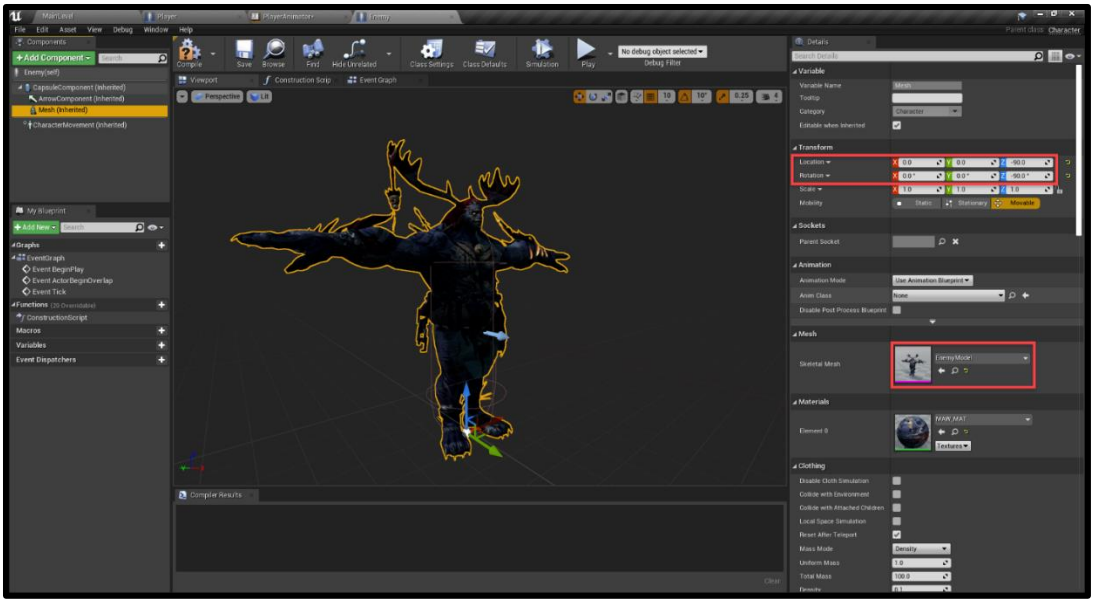
I also increased the size of the ground. You can press **P** to toggle the nav mesh visibility.



Creating the Enemy

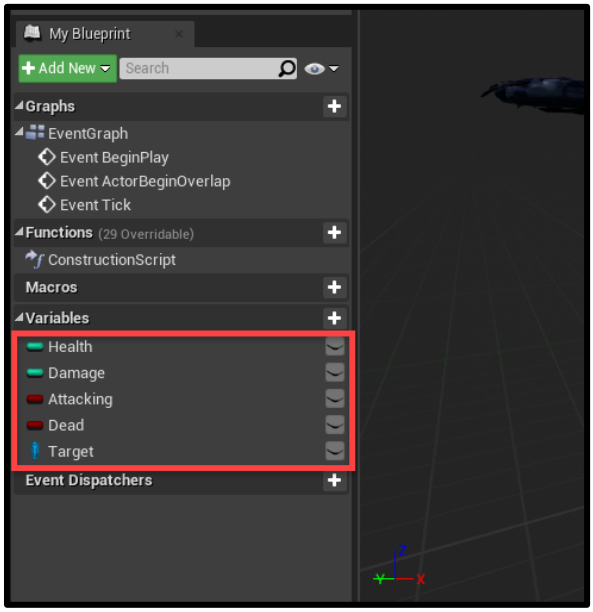
In the **Blueprints** folder, create a new blueprint of type *Character*. Call it **Enemy**. In the blueprint, we're just going to modify the mesh component.

- Set the **Static Mesh** to *EnemyModel*
- Set the **Location** to *0, 0, -90*
- Set the **Rotation** to *0, 0, -90*

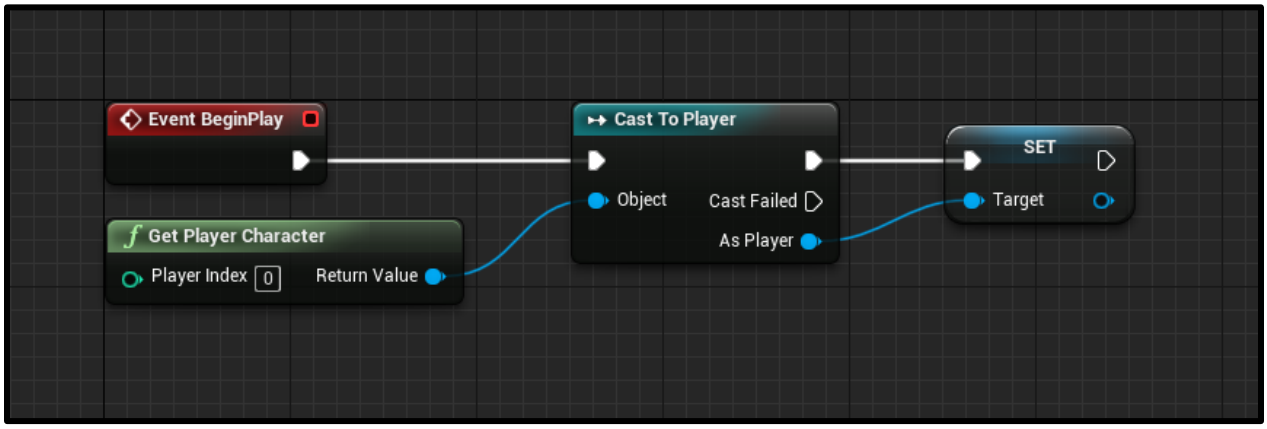


Now for the variables.

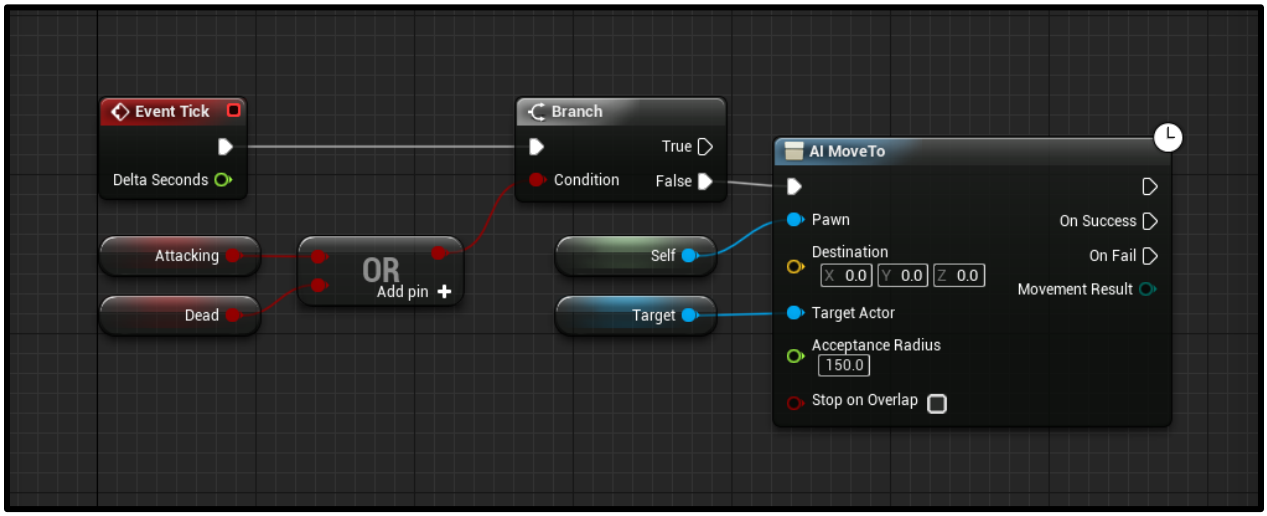
- Health (Integer)
    - Default value = 5
- Damage (Integer)
    - Default value = 1
- Attacking (Boolean)
- Dead (Boolean)
- Target (Player)



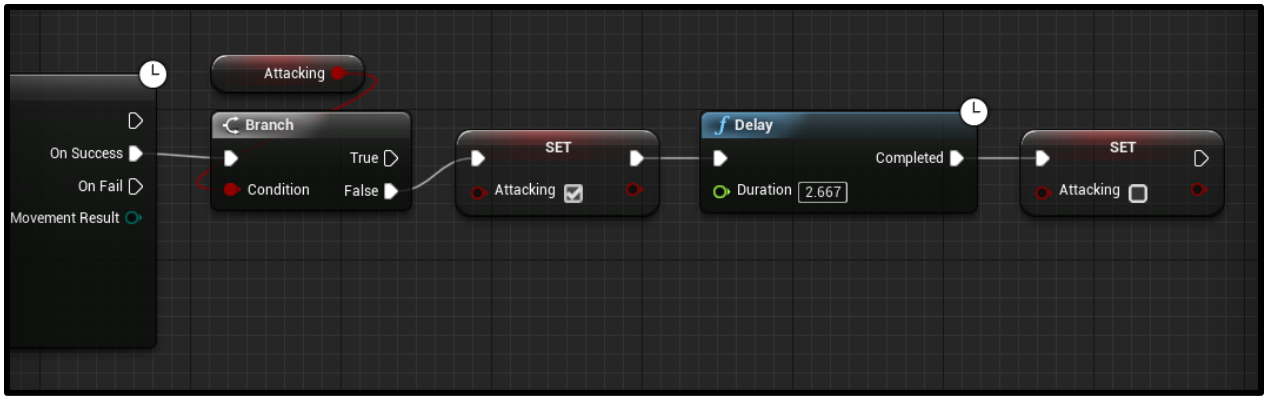
In the **Event Graph**, we're first going to get the player.



Then every frame we'll use the **AI Move To** node. This will use the nav mesh to move the enemy towards the player. Make sure to set the **Acceptance Radius** to *150* so the enemy won't go inside the player.

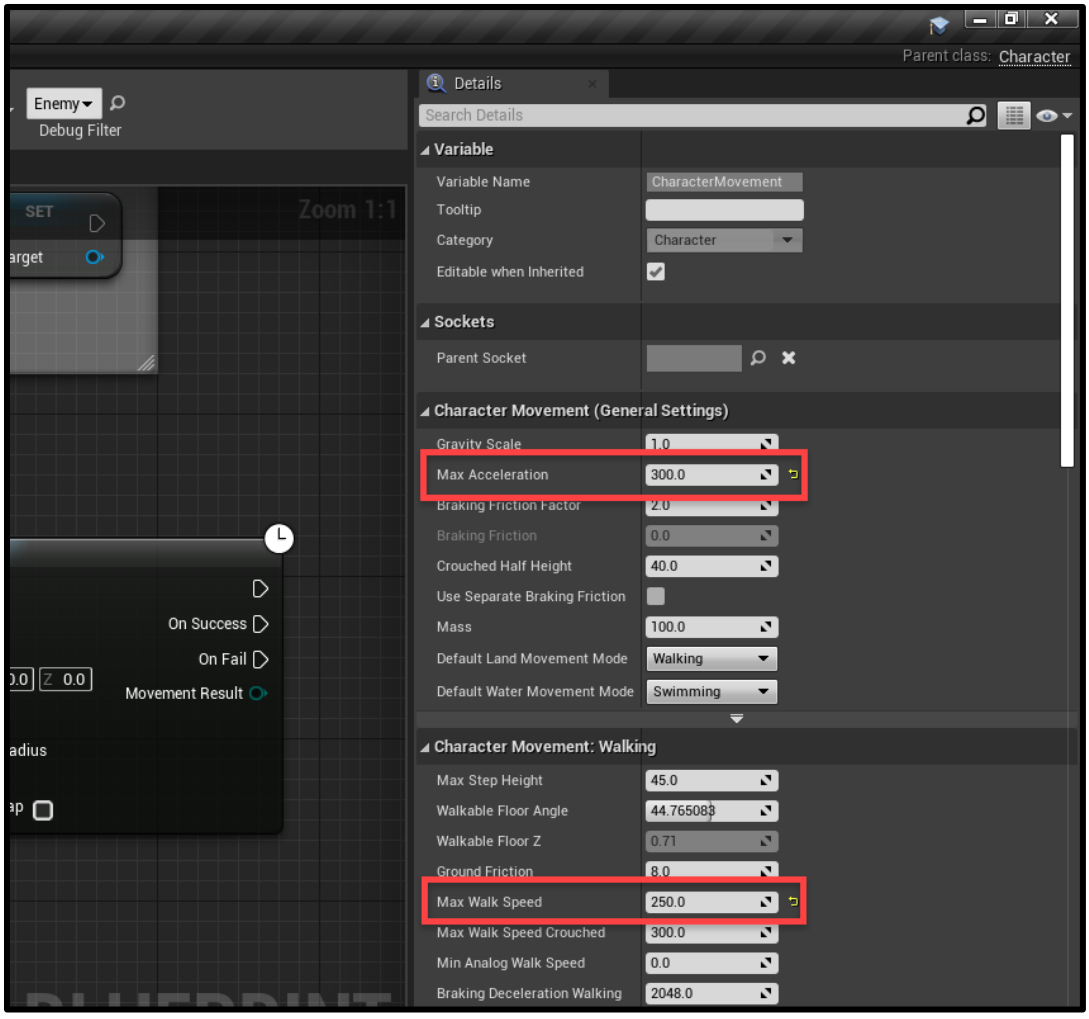


The **On Success** output gets triggered once the enemy reaches the player. When this happens we're going to enable the attacking variable if we can, wait 2.667 seconds (attack animation duration), then re-enable the attacking variable.



Select the **CharacterMovement** component.

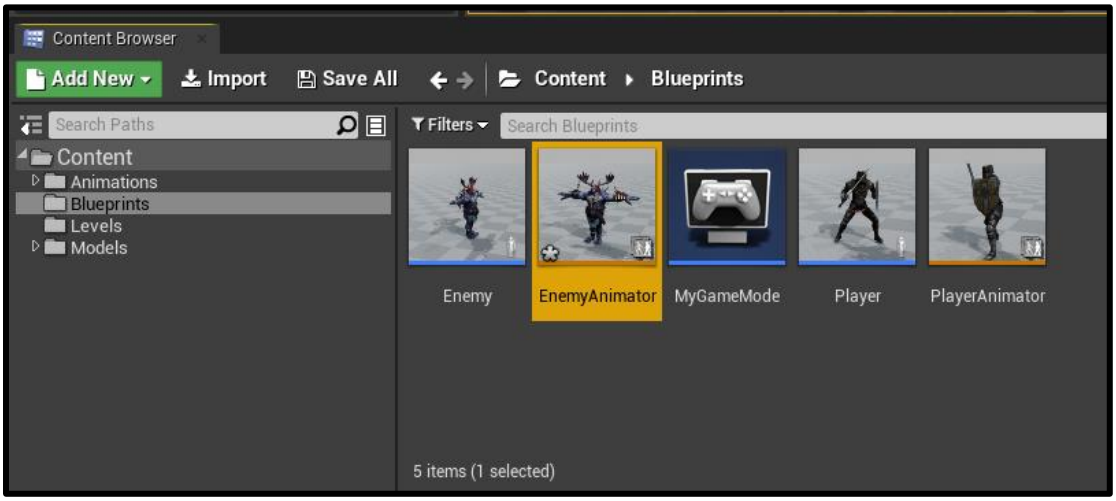
- Set the **Max Acceleration** to *300*
- Set the **Max Walk Speed** to *250*



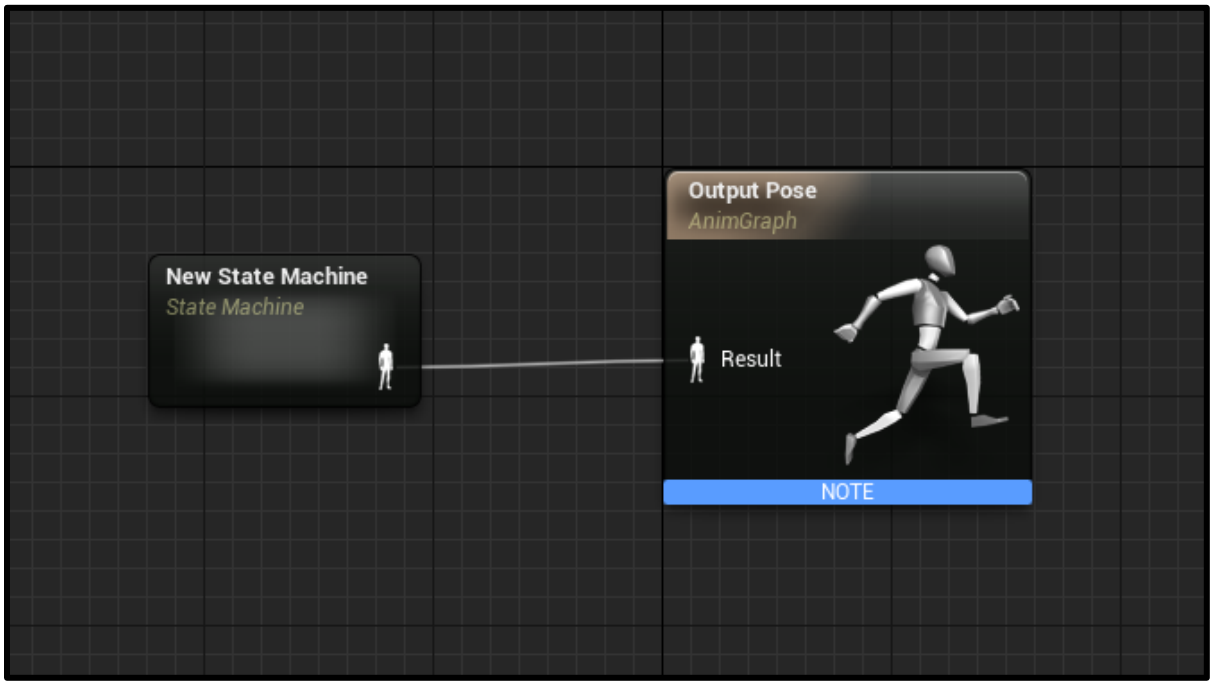
Back in the level editor, we can drag in an enemy, press play and test it out!

Enemy Animations

Like with the player, create a new animation blueprint called **EnemyAnimator**. Make sure you're also linking the enemy skeleton.



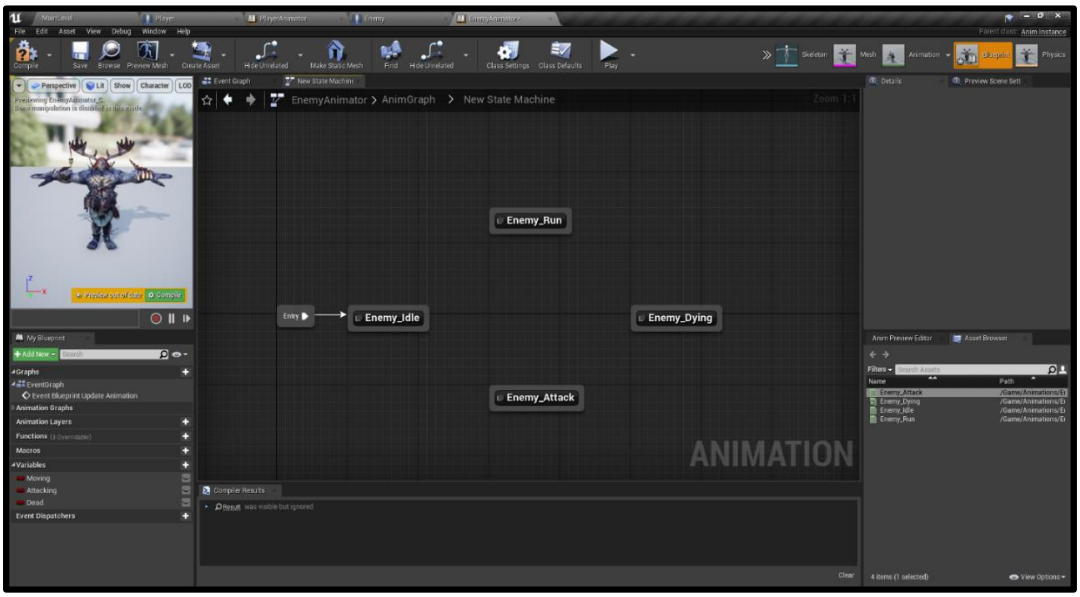
Inside the enemy animator, create a new state machine, then double-click it to enter.



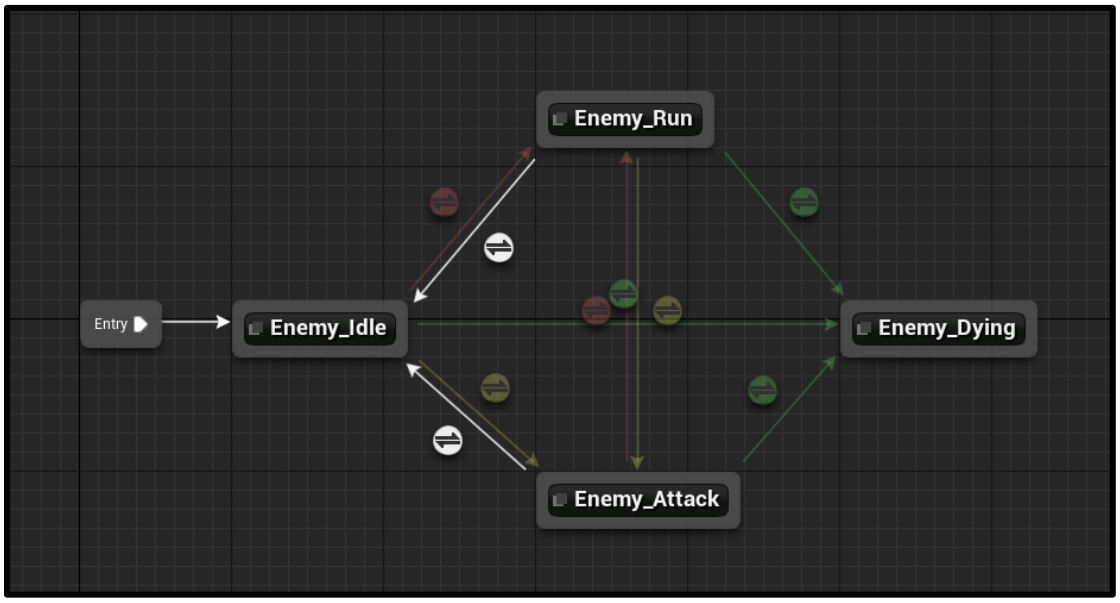
Start with the variables.

- Moving (Boolean)
- Attacking (Boolean)
- Dead (Boolean)

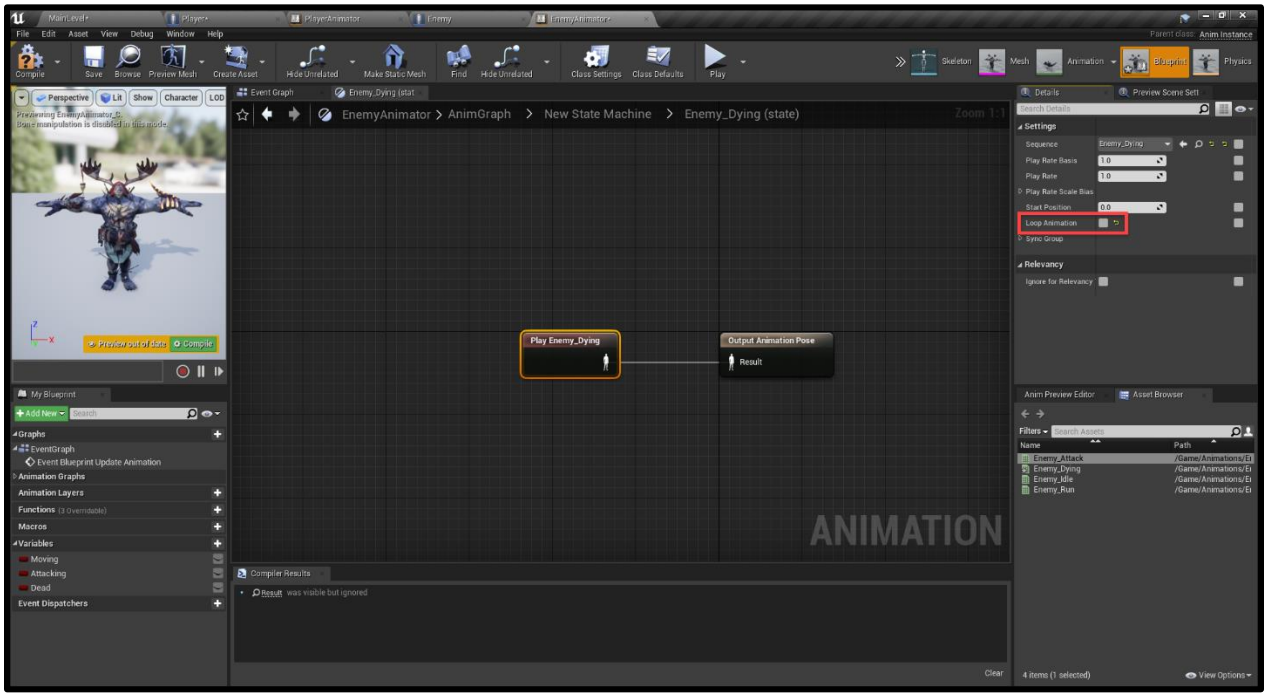
Then we can drag in the 4 animations. Connect the entry to the idle animation.



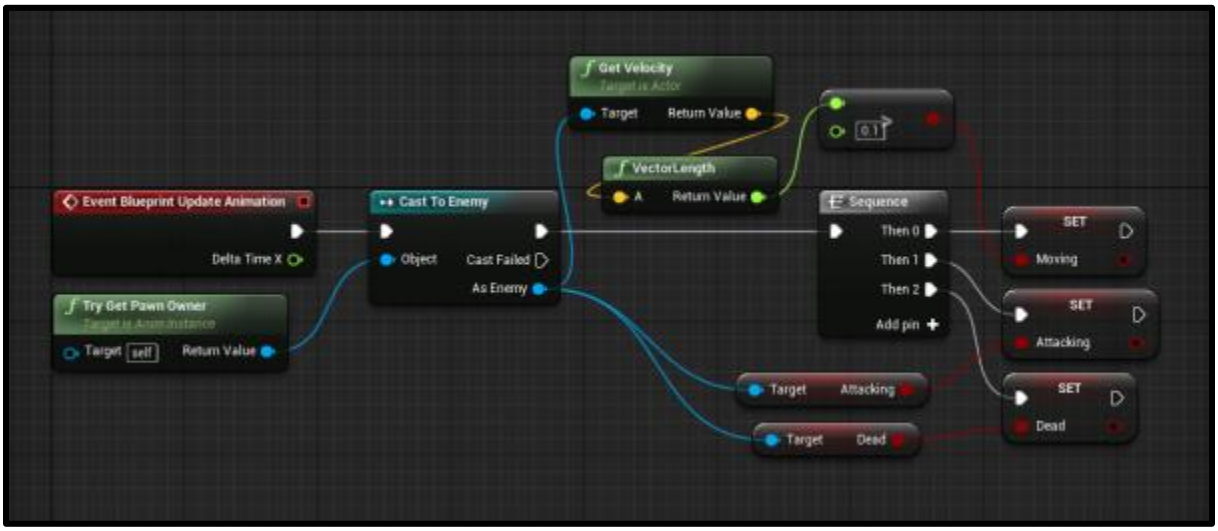
Hook the transitions up like below. Make sure that the dying animation has no exit transitions.



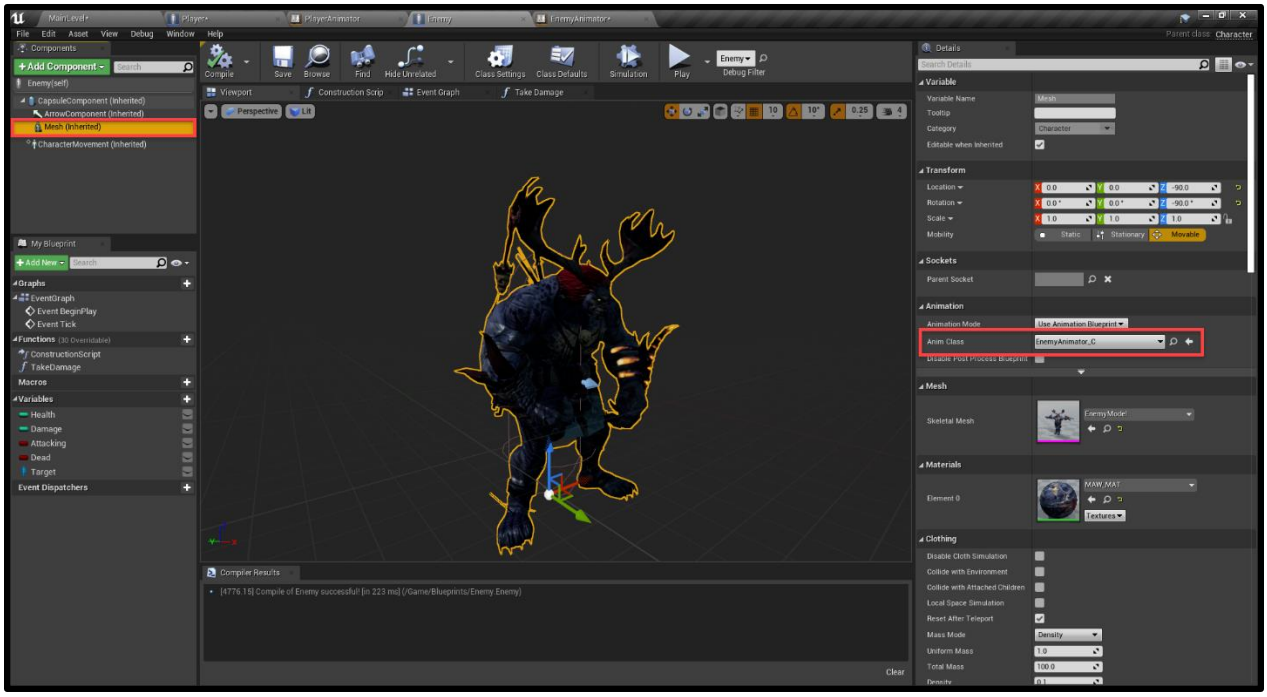
To prevent the dying animation from looping - double click it, select it and disable **Loop Animation**.



In the **Event Graph**, we'll want to hook it up like this.

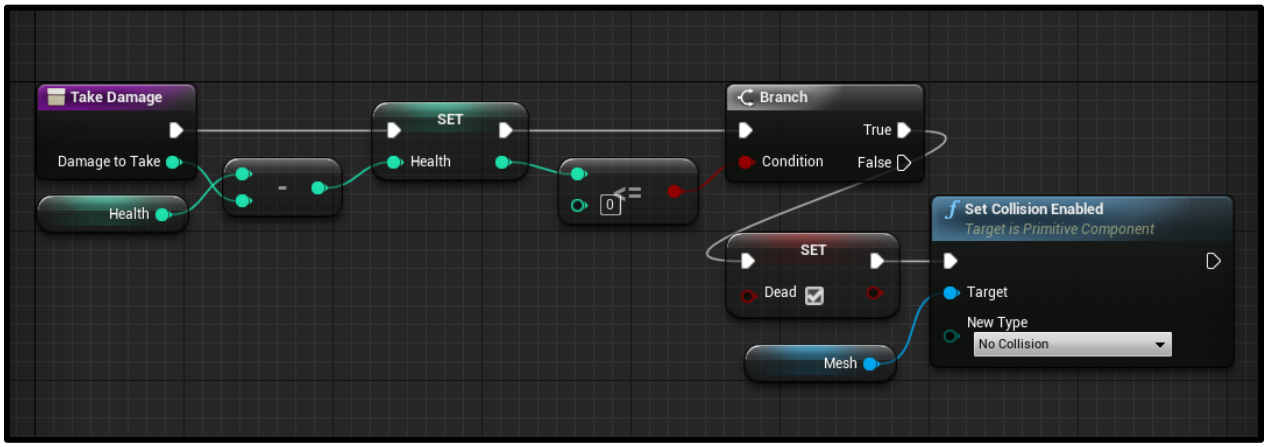


Finally, back in the **Enemy** blueprint, select the mesh and set the **Anim Class** to *EnemyAnimator*.

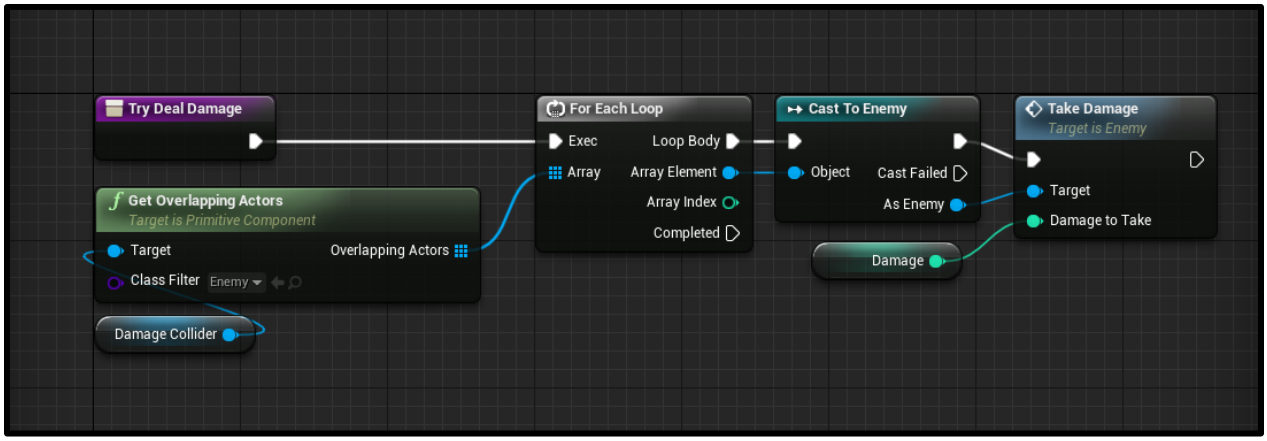


Damaging the Enemy

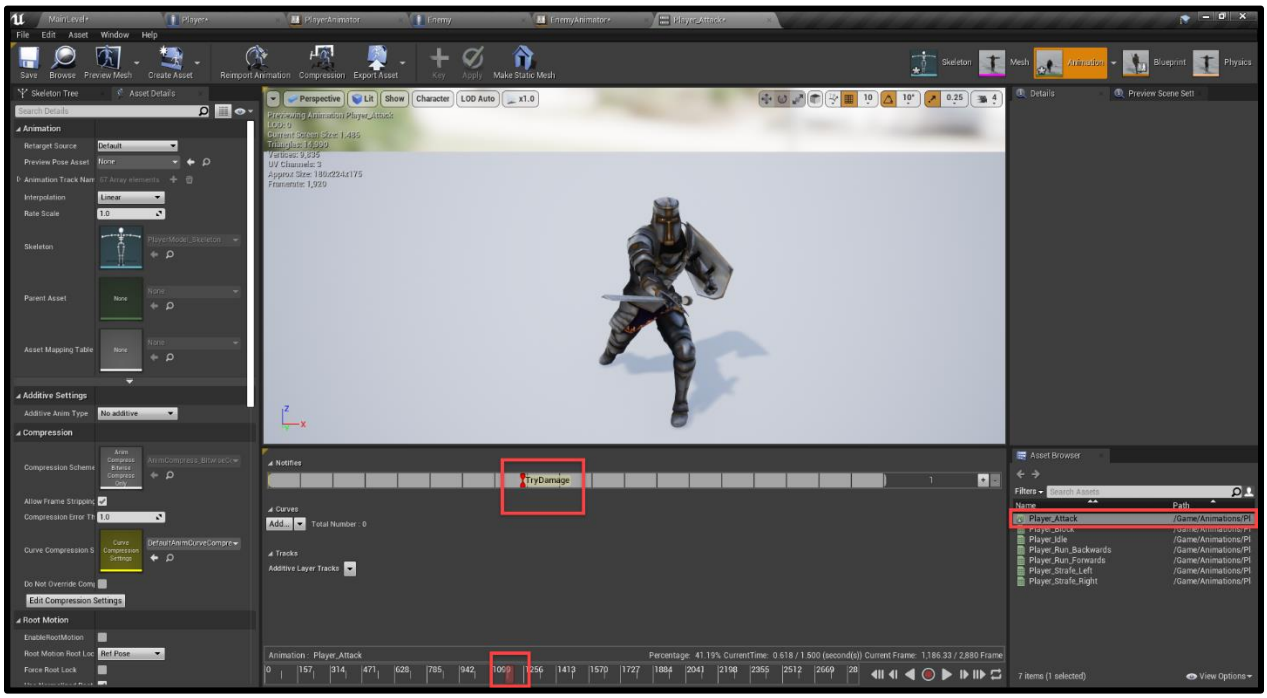
Let's now implement the ability for the player to damage the enemy. In the **Enemy** blueprint, let's create a new function called **TakeDamage.** Create an input of type integer called *DamageToTake*.



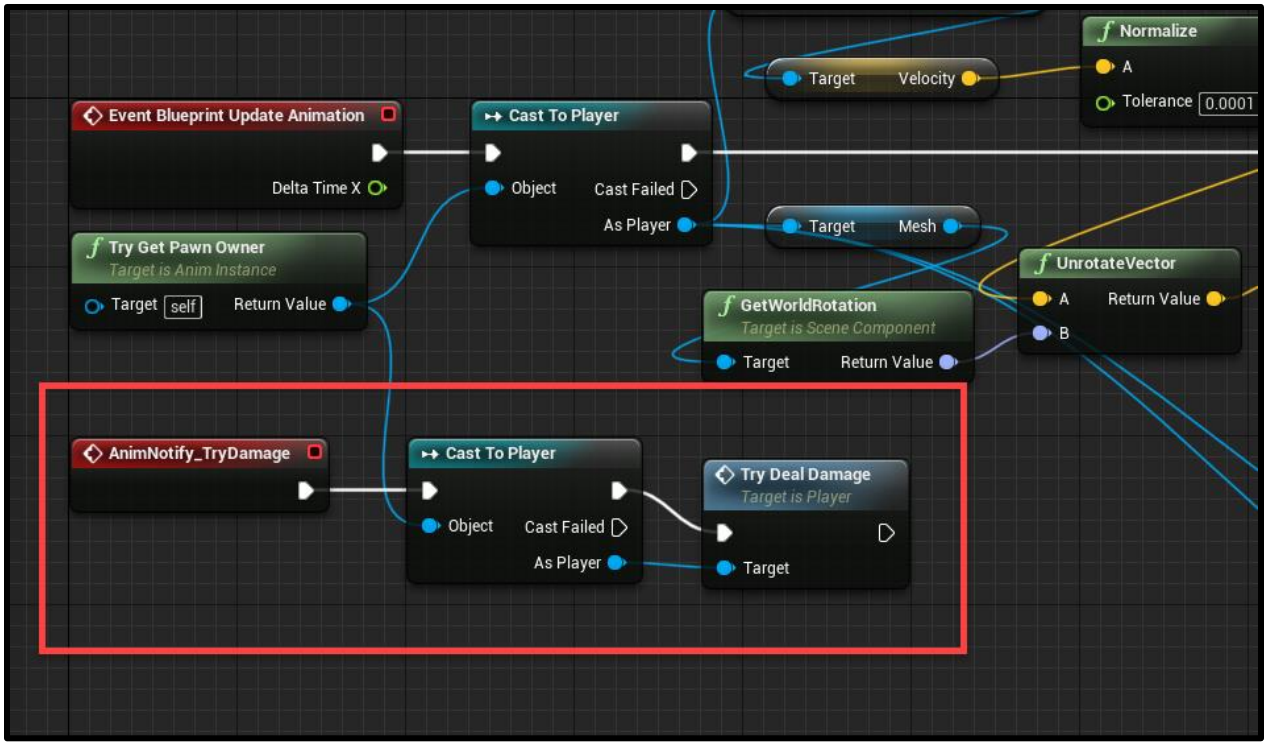
This will be called over in the **Player** blueprint. In there, create a new function called **TryDealDamage**. This will get an array of enemies overlapping the damage collider. We're just going to call their take damage function.



The **TryDealDamage** function will be called when our animation "hits" the target. So let's go to our **PlayerAnimator** and double click on the *Player_Attack* animation to open it up. Move the play head to where we want to attack. Right click the notfies timeline and select *Add Notify* > *New Notify*. Call this **TryDamage**. A notify is basically a custom event which triggers at a certain point in the animation.



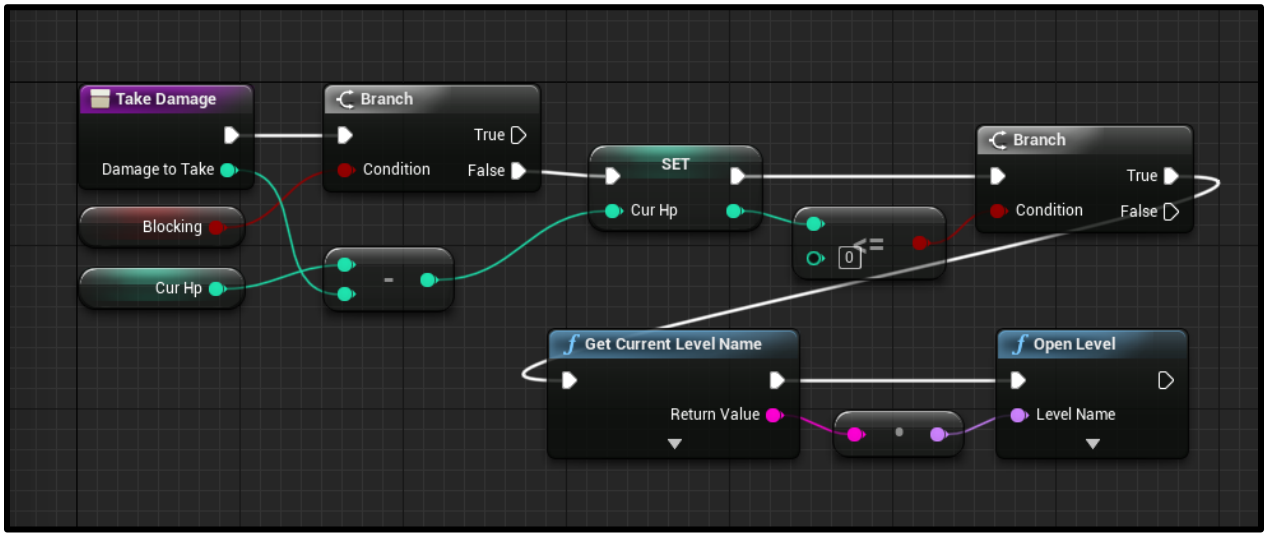
Over in the **PlayerAnimator** event graph, we can create the **AnimNotify_TryDamage** node. We just want to cast the player again and call the TryDealDamage function.



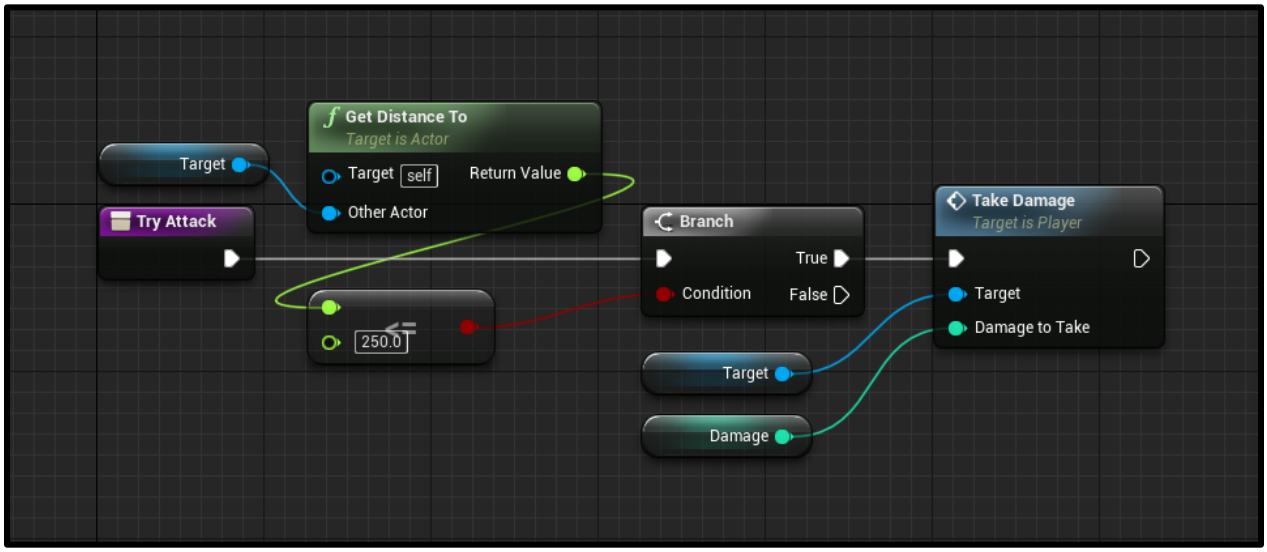
We should now be able to press play and defeat the enemy.

Attacking the Player

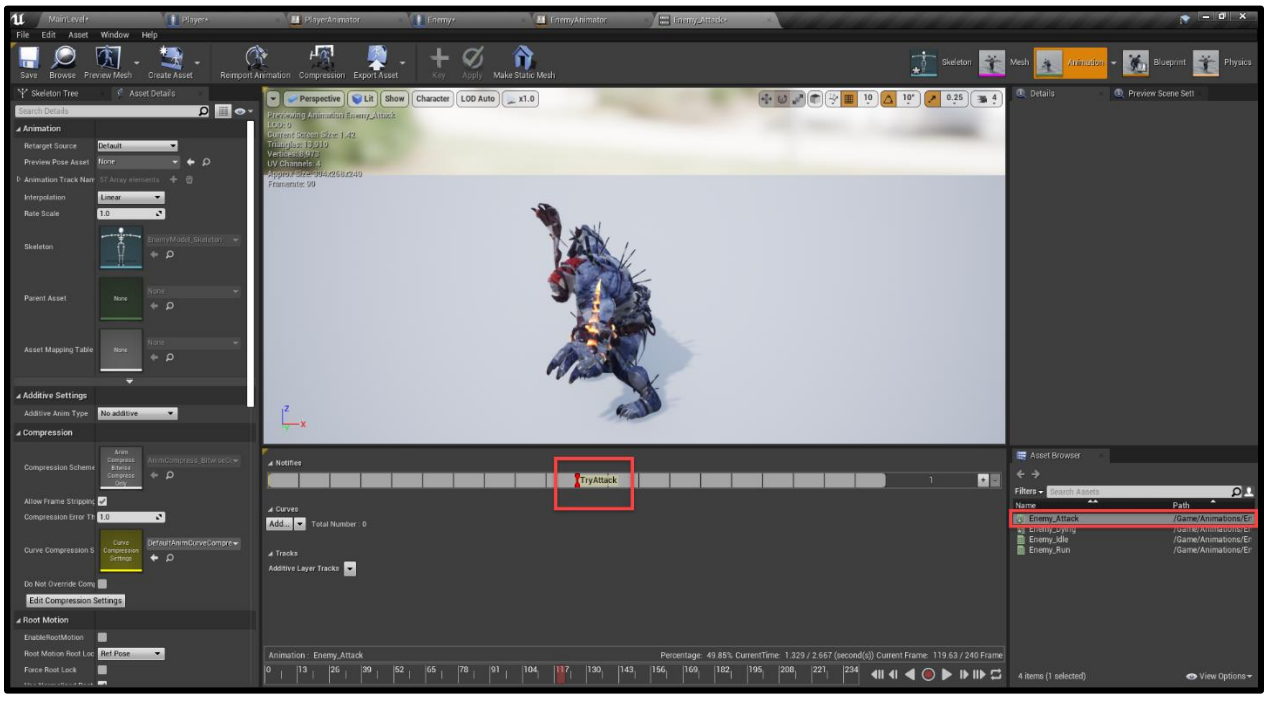
Now we need to implement the ability for the enemy to attack too. In the **Player** blueprint, let's begin by creating the **TakeDamage** function. This will have an input of type integer called *DamageToTake*. When the player's health reaches 0, the level will restart.



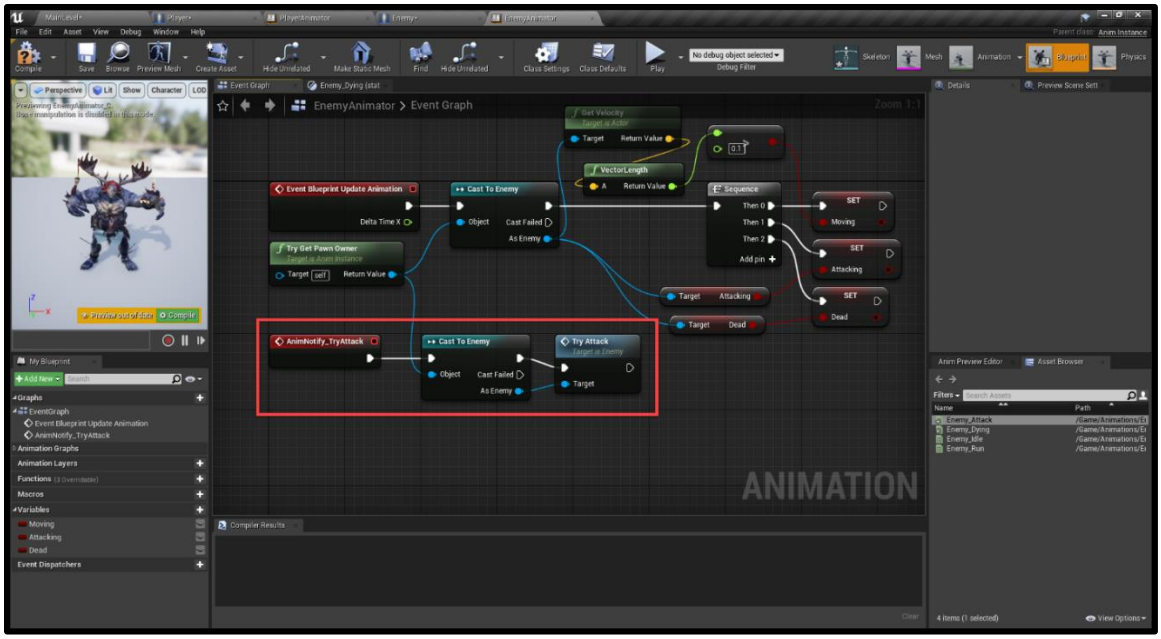
In the **Enemy** blueprint, create a new function called **TryAttack**. Here, we're just checking our distance from the player and if it's within a range, we'll deal damage.



Finally, we need to go to the **EnemyAnimator** blueprint, double-click on the **Enemy_Attack** animation and create a notify at the point of damage.



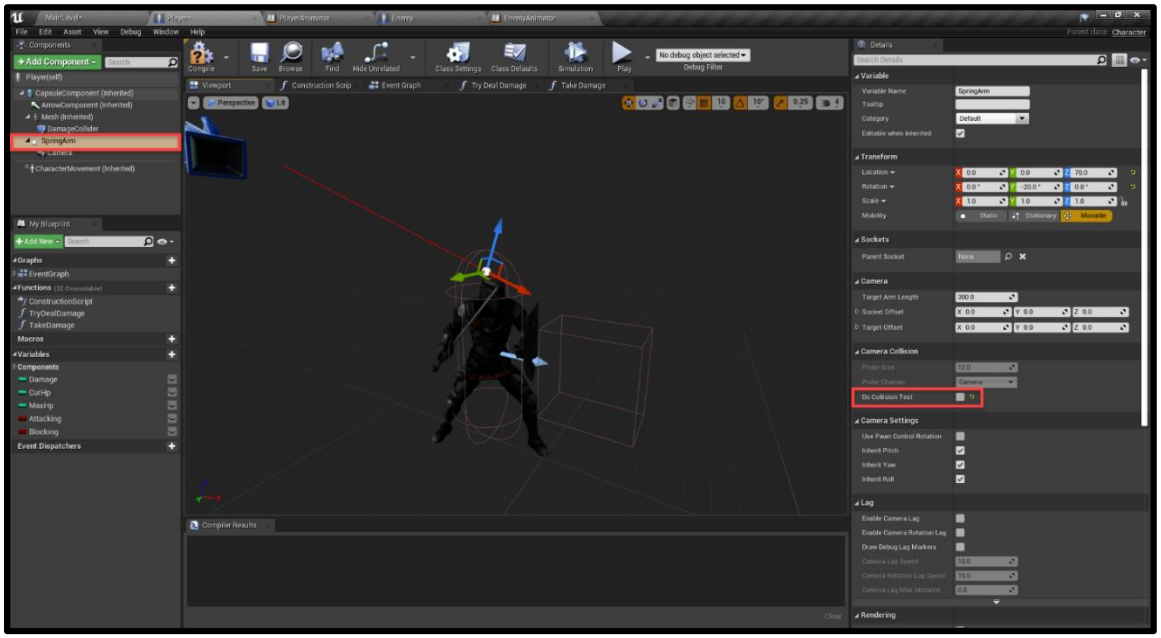
In the enemy animator event graph, we can create the notify event node and try attack like so.



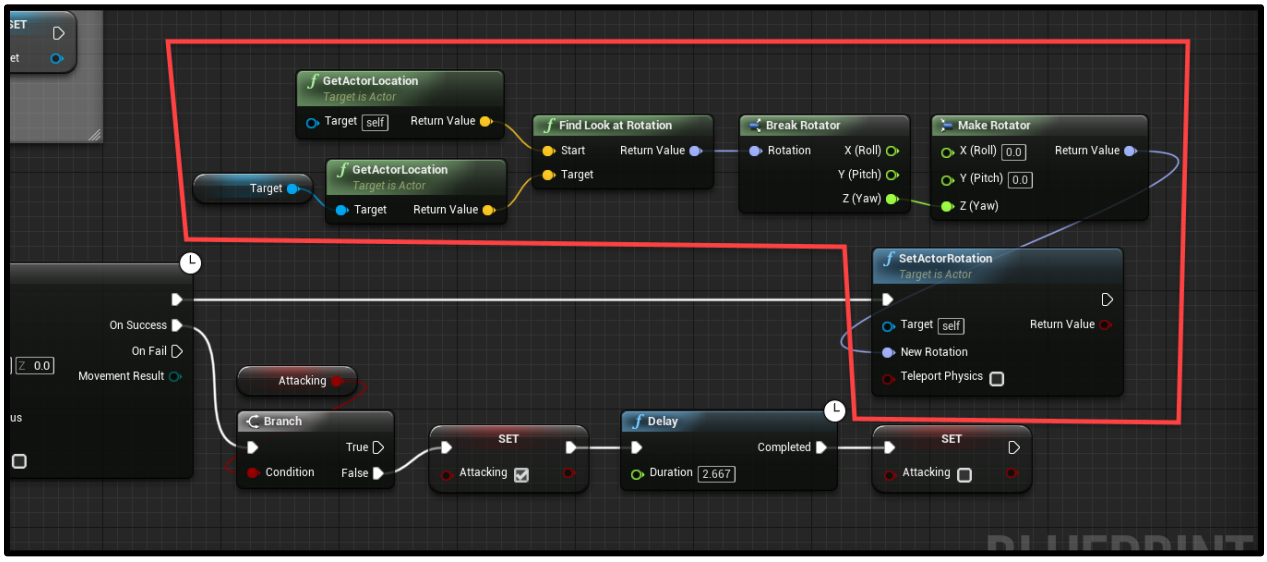
Now you can press play and test it out!

Fixing a Few Things

You may notice that when the enemy attacks, your camera glitches out a bit. This is because the camera automatically moves depending on if there's anything between it and the player. Go to the **Player** blueprint, select the spring arm and disable **Do Collision Test**.



You may also want the enemy to always be facing the player. To fix this, we can go to the **Enemy** blueprint and add these nodes to the event tick path.



Conclusion

And there you go! We now have a working action RPG!

Through this tutorial, we set up a third-person player controller, a rotatable camera, enemy AI, animation state machines, and more - all using Unreal Engine and blueprints. With these foundations in place, the project can easily be expanded should you want to add more levels, enemies, or a variety of other features. You can, of course, use the fundamentals covered here to create other various types of games.

Either way, we hope you enjoyed the tutorial, and good luck with your game projects!

Create a Puzzle Game in the Unreal Engine

Introduction

Ready to challenge your players to unique, brain-busting puzzles?

Puzzle games are a fairly unique genre in game development, requiring even more work than many other genres in its level design phase. However, they can also be a rewarding experience in terms of development, as they offer unique challenges and systems that can be used across many different game projects.

In this tutorial, we're going to be creating a puzzle game inside of Unreal Engine. This game will feature a player who can move around on a grid and whose goal is to collect a number of followers, evade traps, and reach the end goal. Over the course of this tutorial, not only will you learn to use blueprints in Unreal for this purpose, but also gain a fundamental understanding of how to design puzzles and challenges for your games.

Before we continue, do know that this is not an introductory tutorial. If this is your first time using Unreal Engine, then we recommend you check out our Intro to Unreal Engine course here.

Project Files

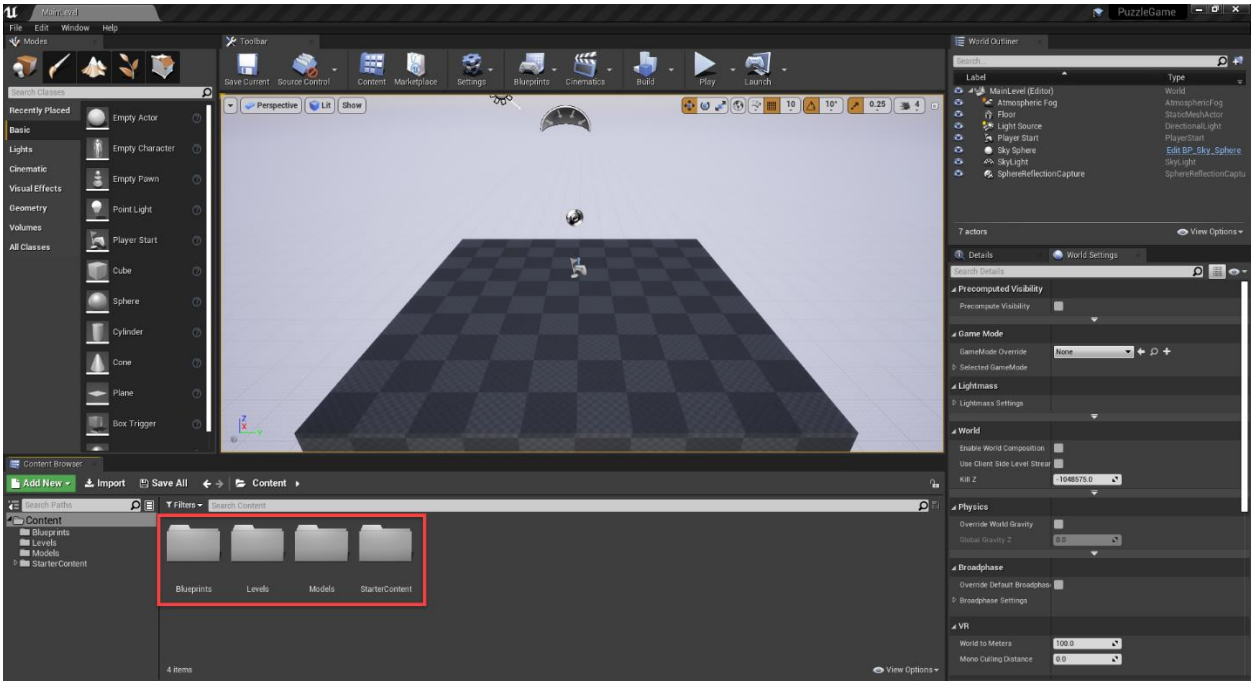
For this project, there are a few 3D models we'll be needing. You can use your own, although the tutorial will be made using these. The complete project can also be downloaded via the same link.

Creating the Project

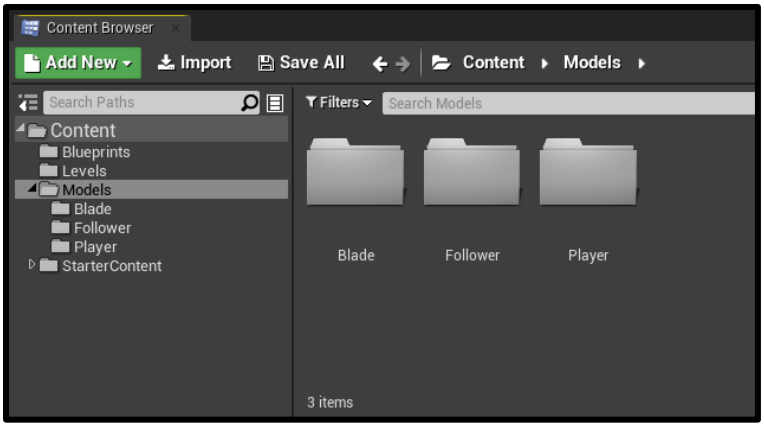
To begin, let's create a new Unreal Engine project. When creating the project, make sure to include the starter content as we'll be using a few of the materials. To begin, let's create three new folders.

- Blueprints
- Levels
- Models

Then, create a new level and save it to the *Levels* folder as **MainLevel**.

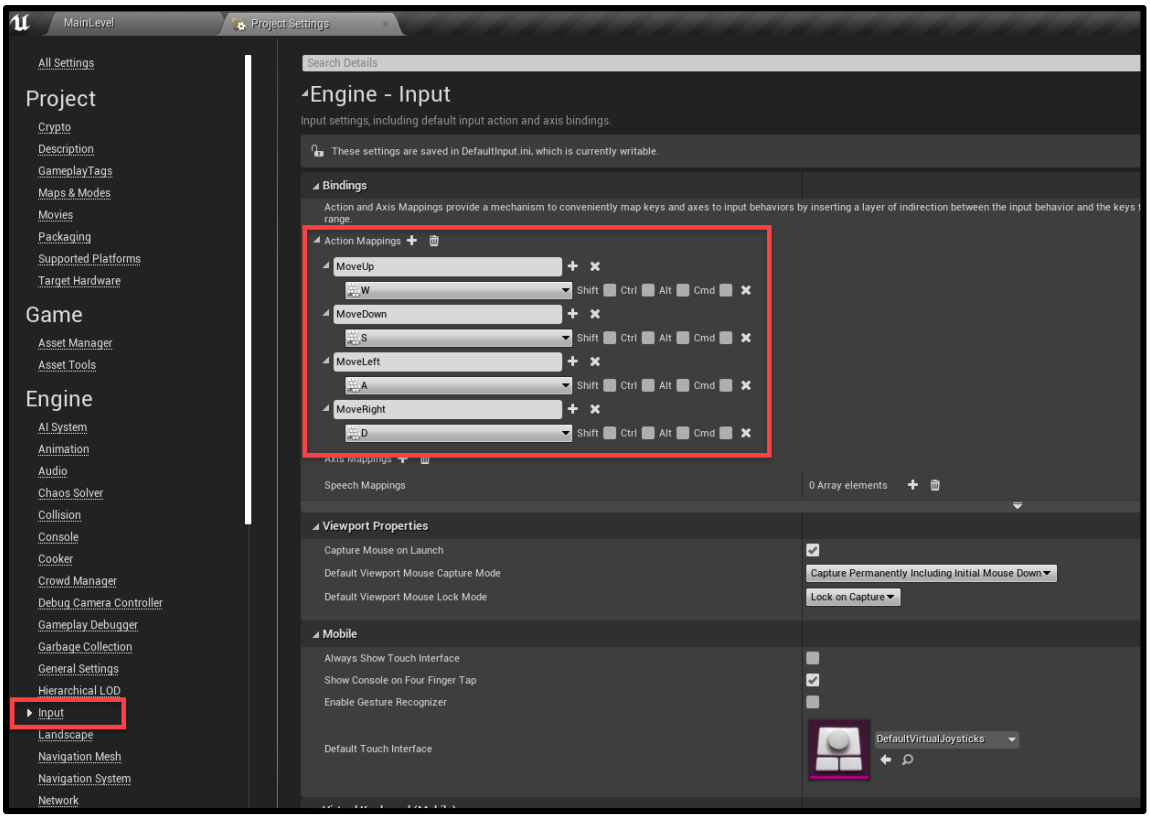


Let's then download the required assets (linked at the top of the tutorial). There is a folder called *Models Folder Content*. Inside of that, drag the contents into our **Content Browser**'s *Models* folder.



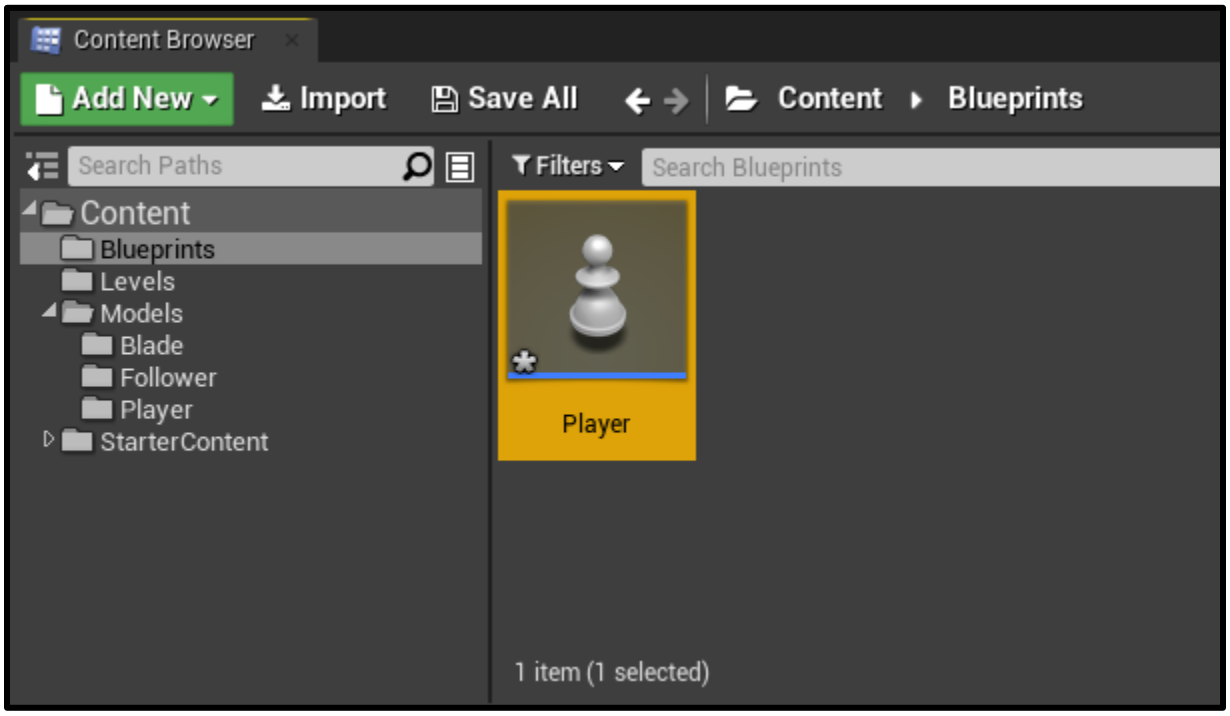
Next, let's setup some key bindings. We need to know which buttons we're going to use for player movement. Open the **Project Settings** window (*Edit > Project Settings...*). Click on the **Input** tab and create 4 new action mappings.

- MoveUp = W
- MoveDown = S
- MoveLeft = A
- MoveRight = D



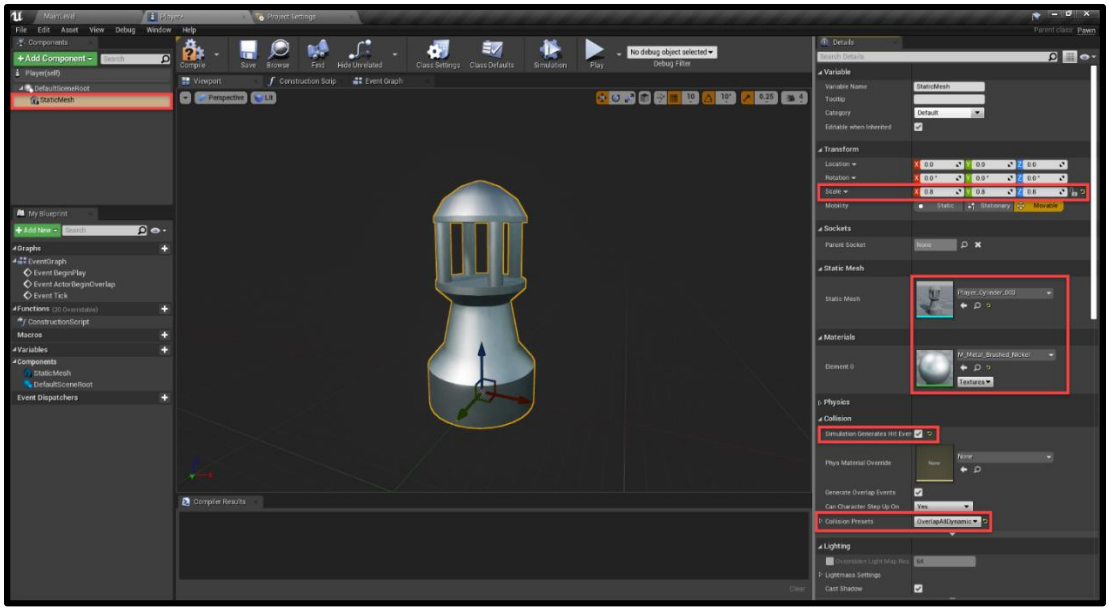
Creating the Player

Now that we've got our controls sorted out, let's create the player. In the *Blueprints* folder, create a new blueprint with a parent class of *Pawn*. Call it **Player**.



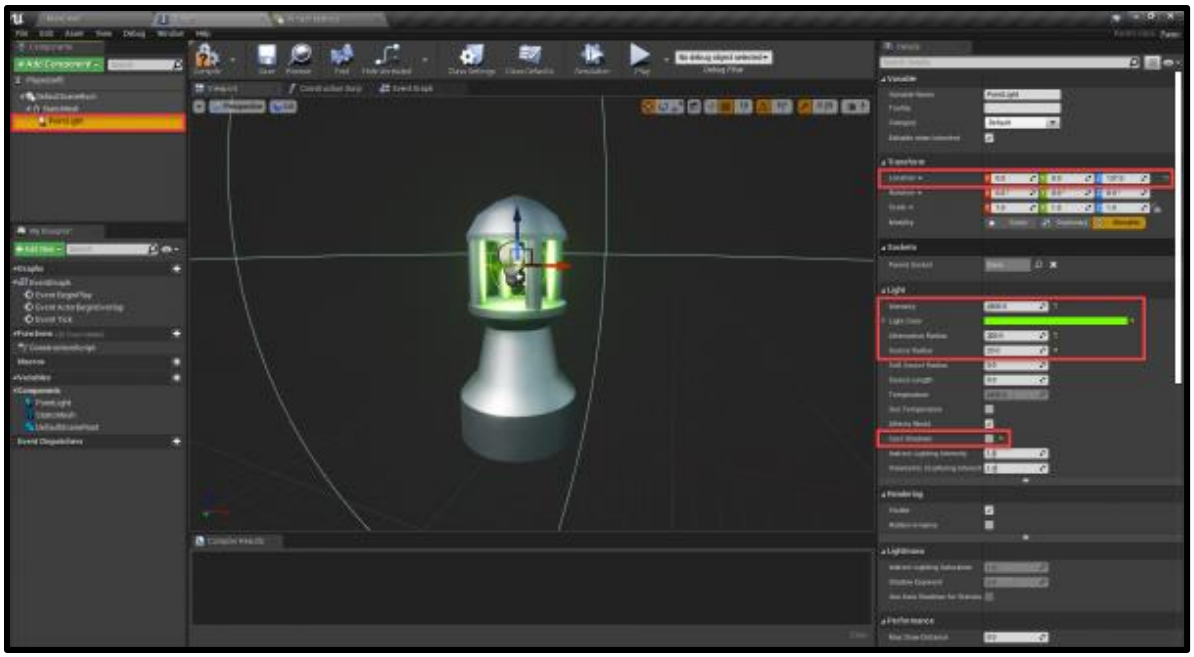
Open it up and we can begin to create the player. First, create a static mesh component.

- Set the **Static Mesh** to *Player_Cylinder_003*
- Set the **Material** to *M_Metal_Brushed_Nickel*
- Set the **Scale** to *0.8, 0.8, 0.8*
- Enable **Simulation Generates Hit Events**
- Set the **Collision Presets** to *OverlapAllDynamic*



Then as a child of the static mesh, create a **Point Light** component.

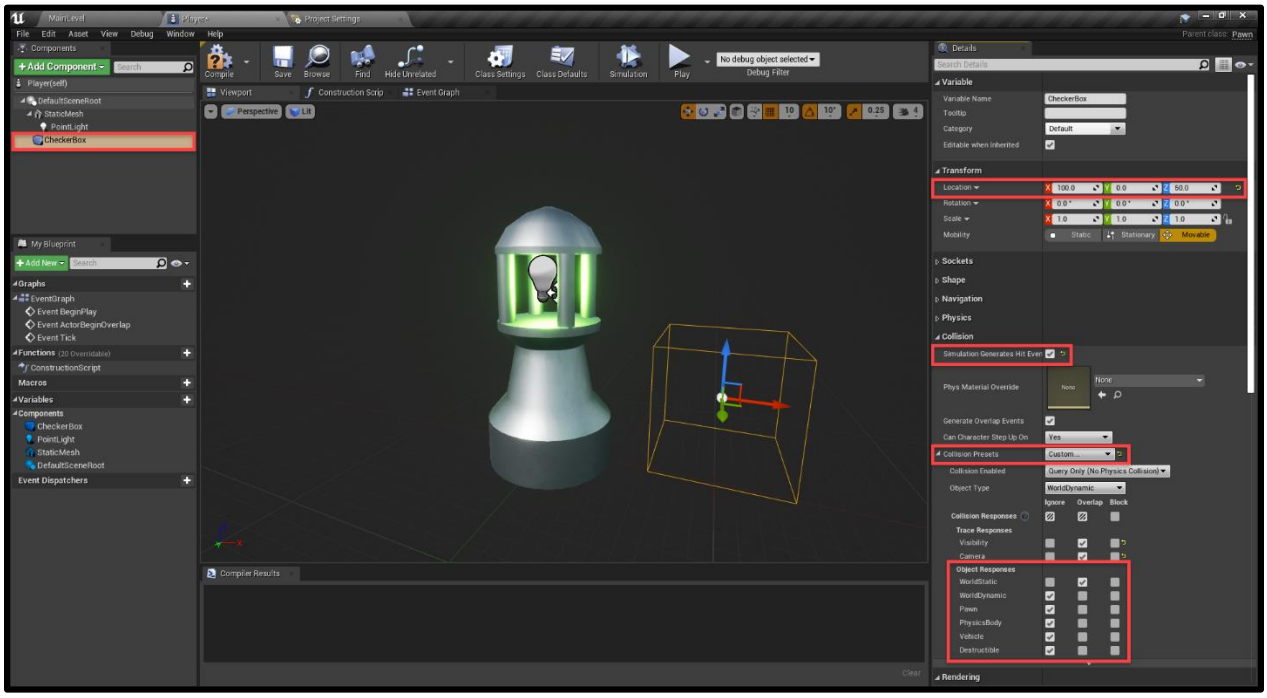
- Set the **Location** to *0, 0, 137*
- Set the **Intensity** to *4500*
- Set the **Light Color** to *Green*
- Set the **Attenuation Radius** to *300*
- Set the **Source Radius** to *20*
- Disable **Cast Shadows**



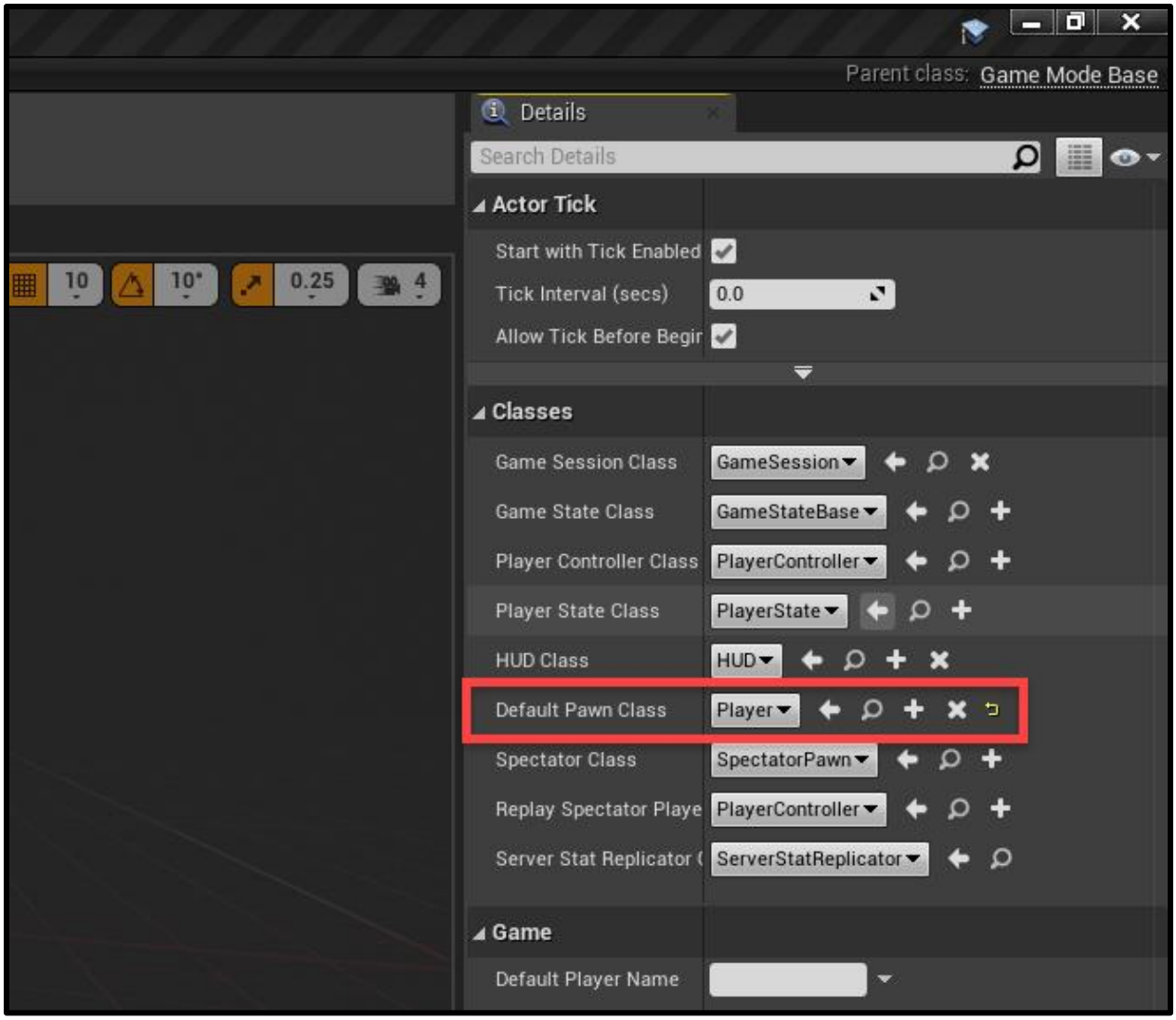
The final component will be a **Box Collision**, and this is used to know if we're going to run into a wall.

- Set the **Location** to *100, 0, 50*
- Enable **Simulation Generates Hit Events**
- Set the **Collision Presets** to *Custom...*

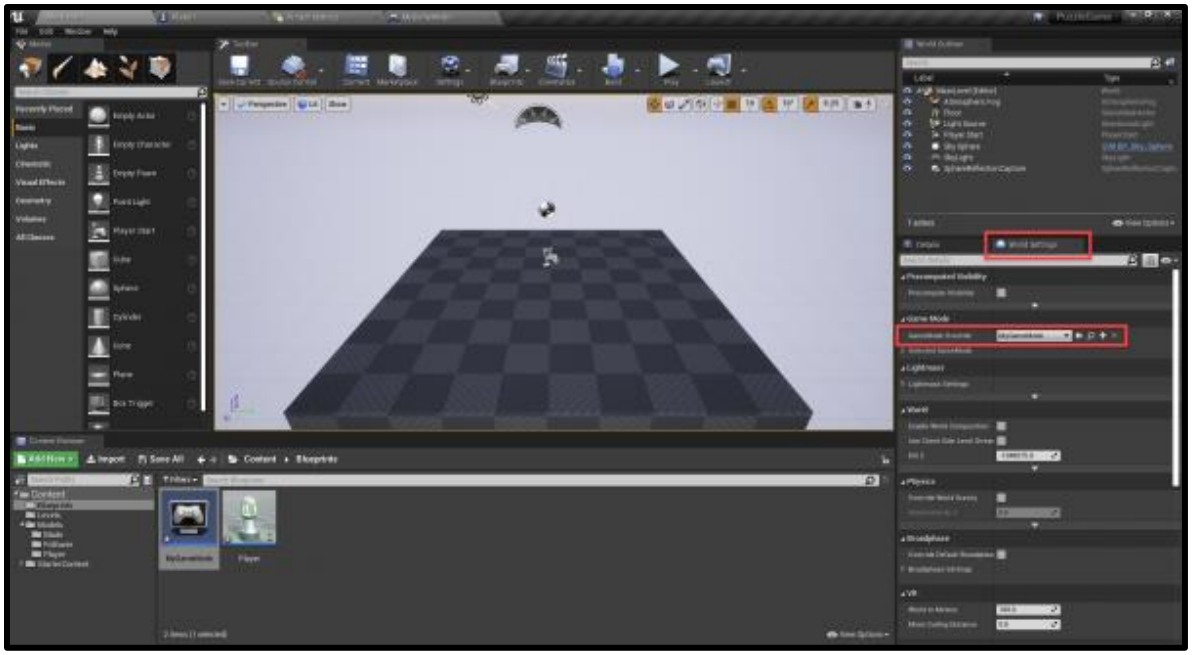
Custom collision presets means that we can choose what we want to detect with this collider. Since we're looking for walls, they fall under *WorldStatic*. So under **Object Responses**, set everything to *Ignore* except *WorldStatic*.



Now that we've got our player setup component-wise, let's place them into the level. Back in the level editor, create a new blueprint of type *GameMode Base* and call it **MyGameMode**. Open it up and all we want to do here is set the **Default Pawn Class** to our *Player*.

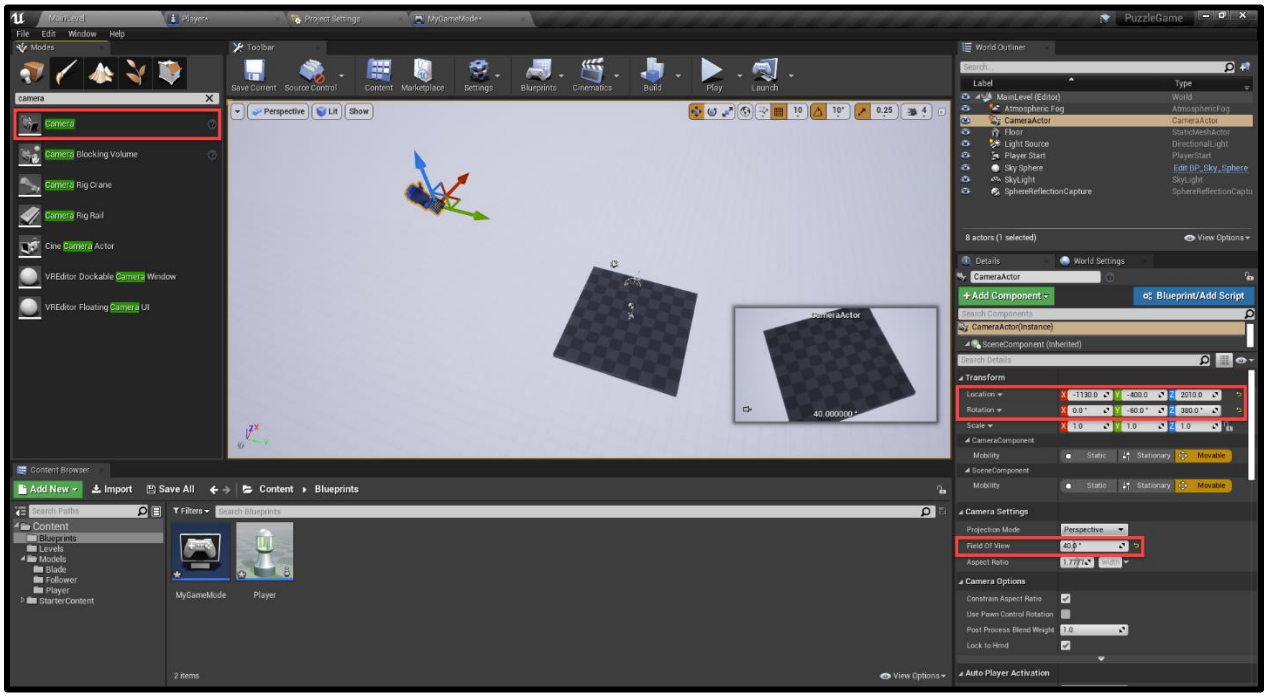


Save, compile then go back to the level editor. In the **World Settings** panel, set the **GameMode Override** to *MyGameMode*. Now when we press play, our player should spawn.



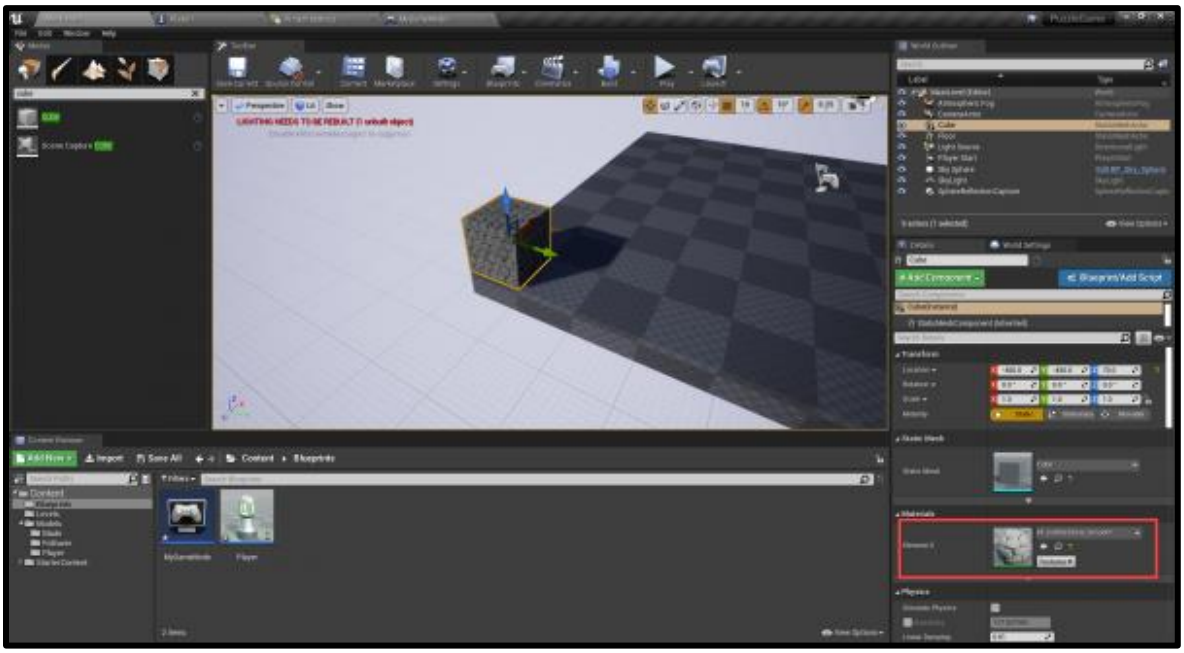
The default camera position is pretty bad, so let's search for **Camera** in the **Modes** panel and drag that in.

- Set the **Location** to *-1130, -400, 210*
- Set the **Rotation** to *0, -60, 380*
- Set the **Field of View** to *40*



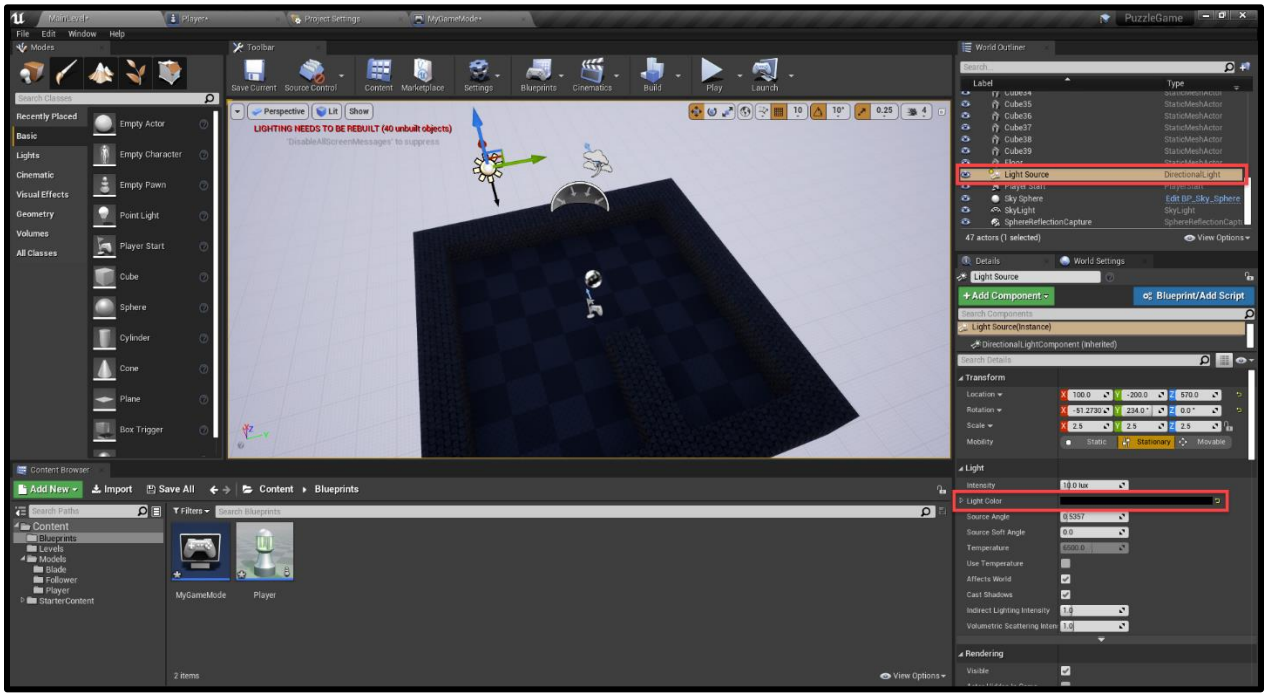
Now let's setup the level bounds. Create a new cube and position it to the corner of the platform.

- Set the **Material** to *M_CobbleStone_Smooth*



Copy and past that cube so we have something that looks like this. Also, select all the walls and in the details panel, enable **Simulation Generates Hit Events** and **Generate Overlap Events**.

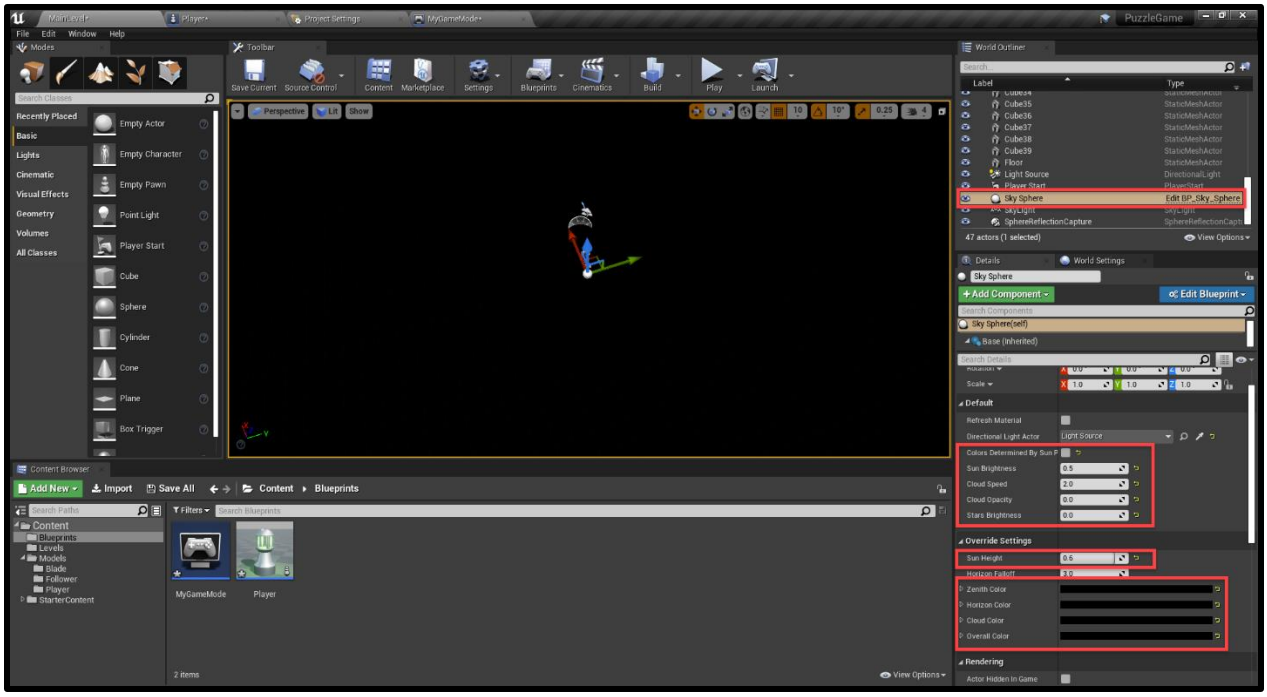
Since this puzzle game is in the dark with lights, let's select the **Light Source** and set the **Light Color** to *Black*.



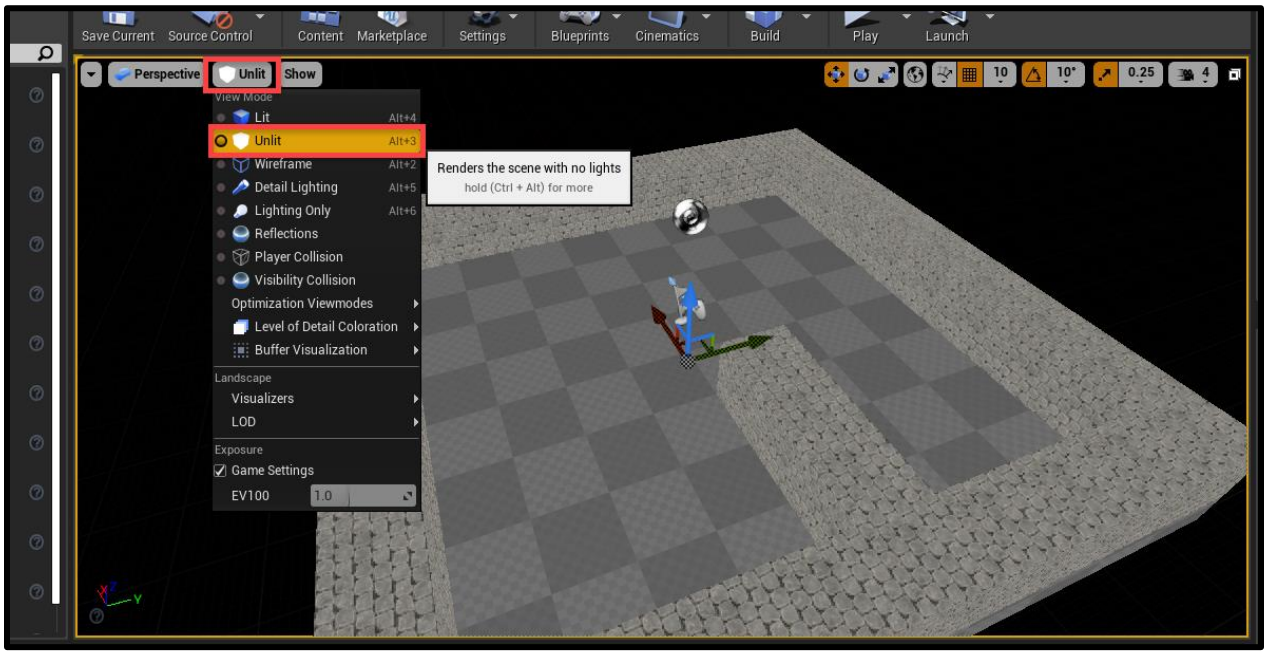
To make the sky night, we can select the **Sky Sphere**.

- Disable **Colors Determined by Sun Position**
- Set the **Sun Brightness** to *0.5*
- Set the **Cloud Opacity** and **Stars Brightness** to *0*
- Set the 4 colors to *Black*

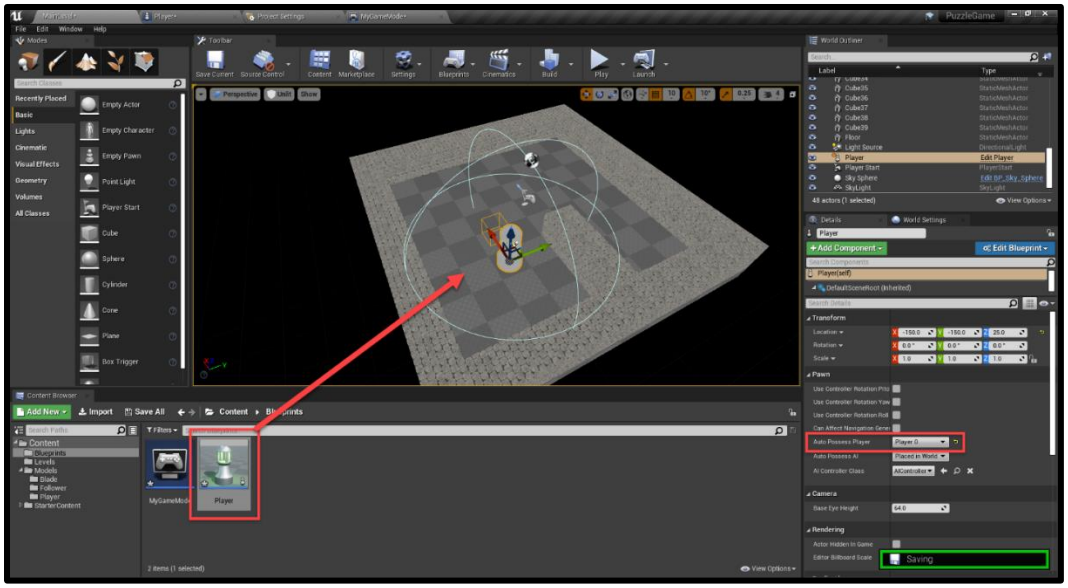
After this, we need to click on the down arrow next to the **Build** button (in the toolbar) and select *Build Lighting Only*.



It's pretty dark, so a way we can see is by disabling the lighting when working on the level. In the viewport, click on **Lit** and select *Unlit*.



Drag in the **Player** blueprint. Set the **Auto Possess Player** to *Player 0* so when we press play, we have control of the player.

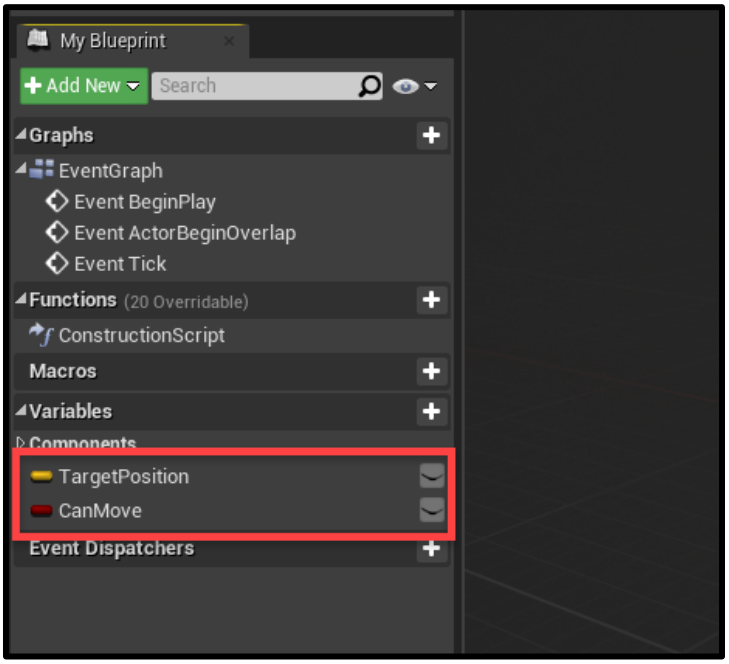


Moving the Player

We'll begin by adding in the ability for the player to move. First, we'll create the variables.

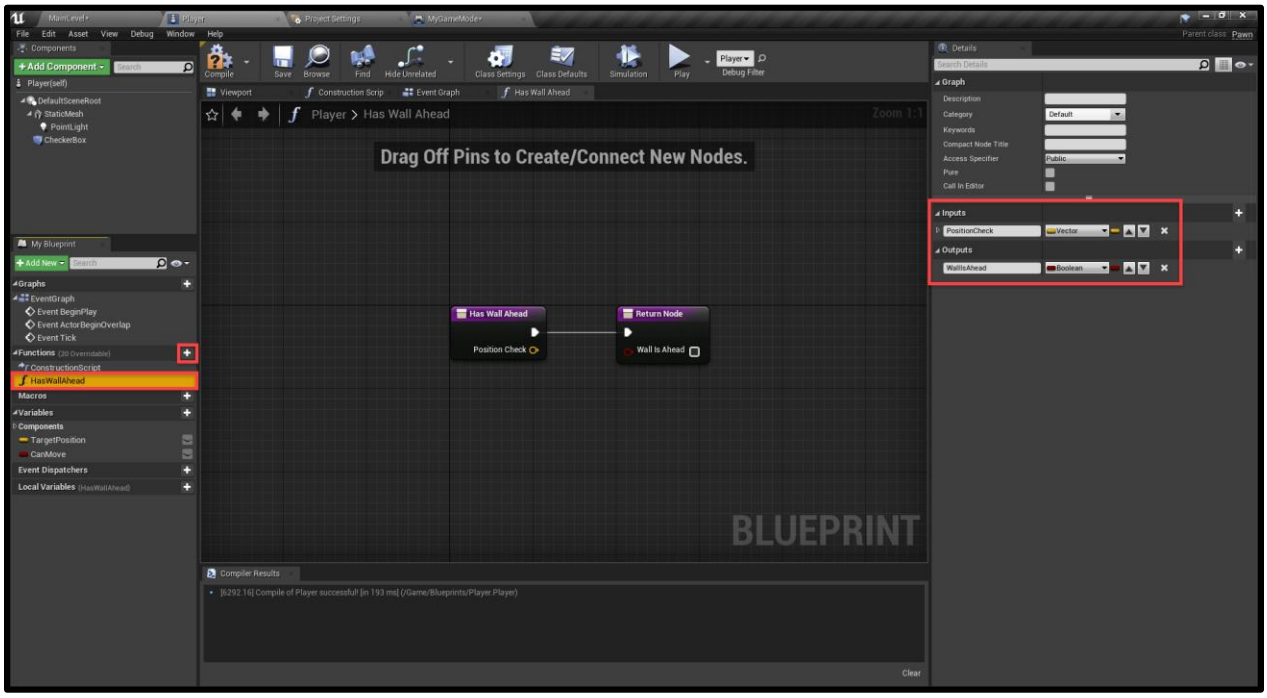
- TargetPositon (Vector)
- CanMove (Boolean)

Compile, then set the **CanMove** default value to *true*.

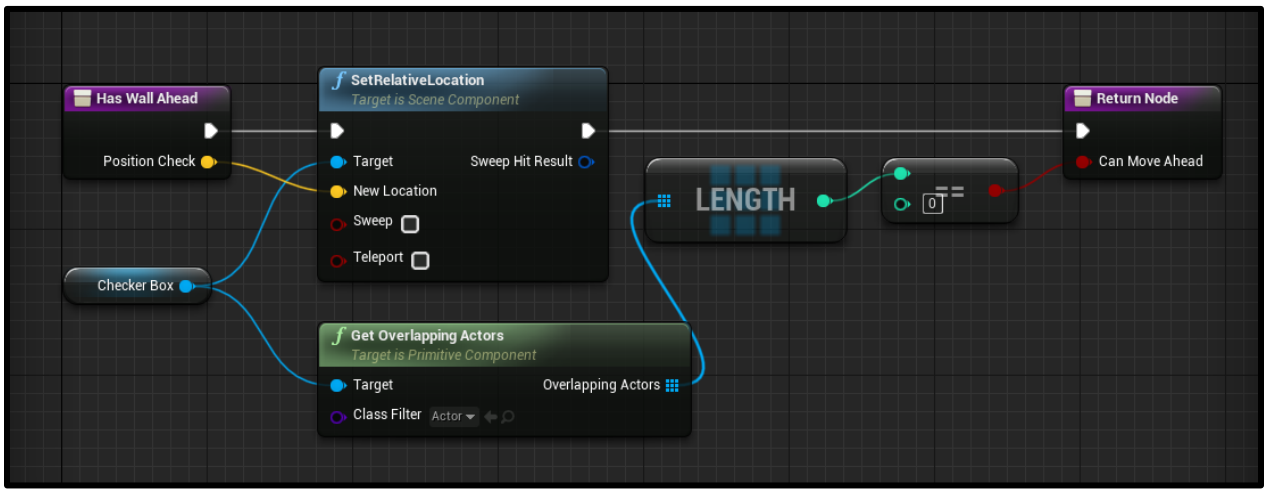


Whenever we want to move the player, we need to see if we're going to move into a wall. If we are, then don't move. This will be calculated in a new function called **HasWallAhead**.

- Create a vector input called **PositionCheck**
- Create a boolean output called **CanMoveAhead** (different from image)

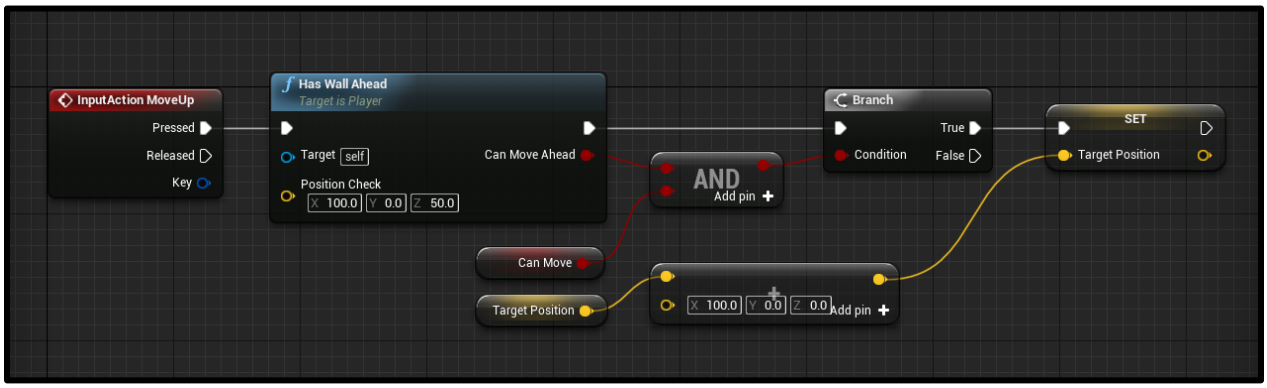


This function will change the position of the checker box, then see if it's colliding with anything, outputting whether or not we can move ahead.

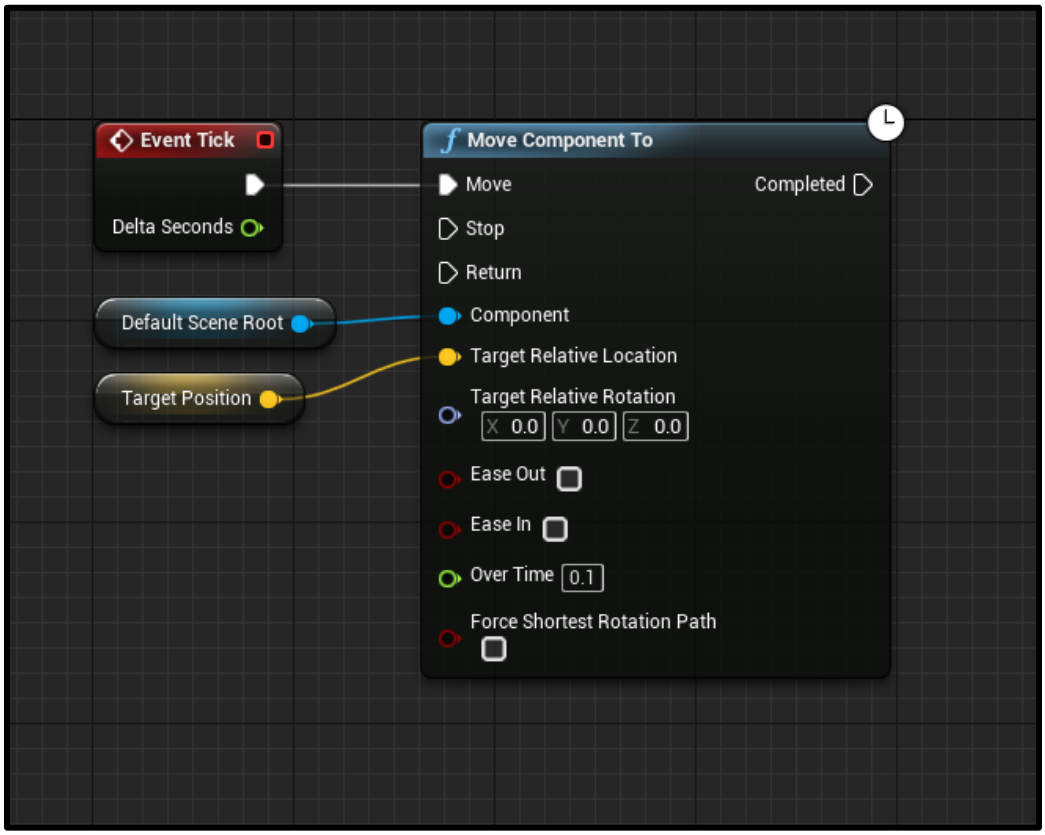


Back in the main **Event Graph**, we can begin to setup the movement inputs. Here's how the movement will work:

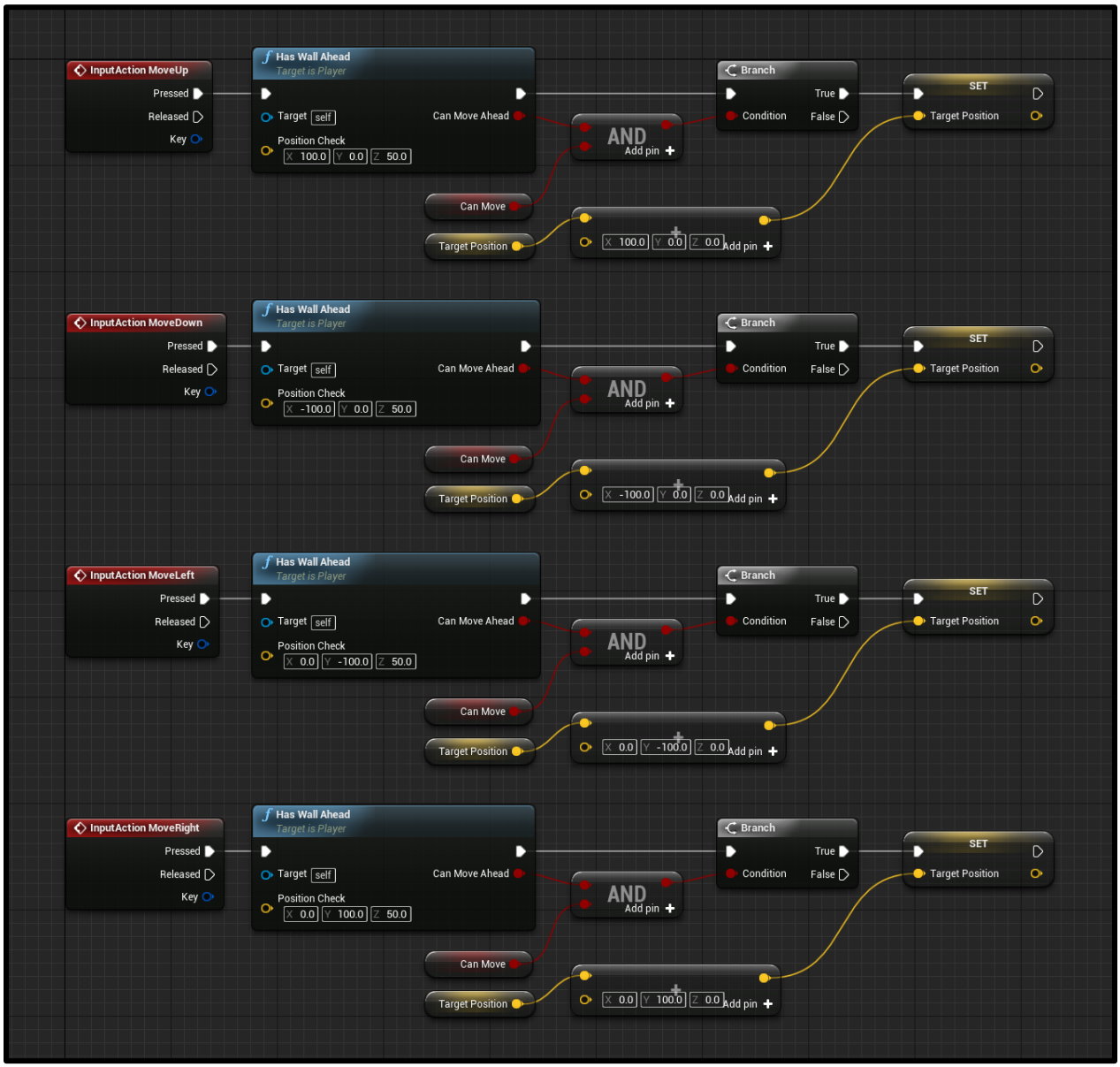
1. Detect movement key input (WASD)
2. Check if we can move to that location
3. Set the target position to that new position
4. Overtime move towards the target position



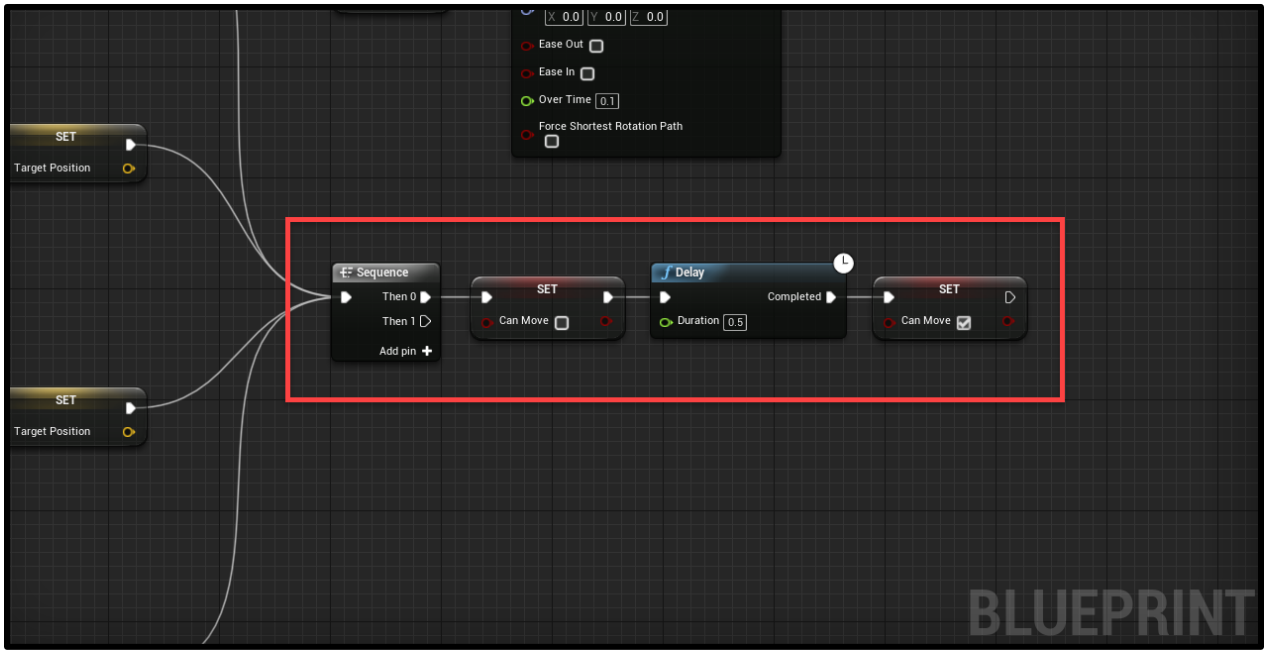
Using the tick node, we can move towards the target position every frame by plugging that into a **Move Component To** node.



If you press play, you'll be able to move forward with the W key. Now let's go ahead and setup the other three movement inputs.



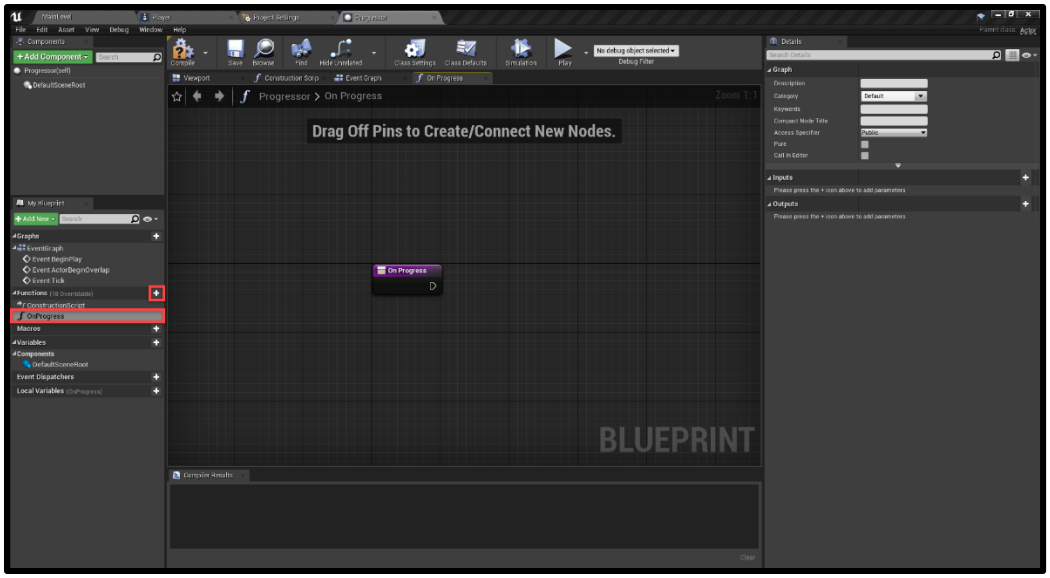
You may notice that we can move again even though we're not at the target position yet. To fix this, create a **Sequence** node (we'll be adding more to this is the future). Have the input connect to each of the set target position node outputs. What we want to do here is disable can move, wait half a second, then re-enable can move.



Creating the Progressor

This is going to be a game where everything moves only when the player moves. The followers, obstacles, etc. So to make it a nice system, we'll create a base blueprint which all others will inherit from.

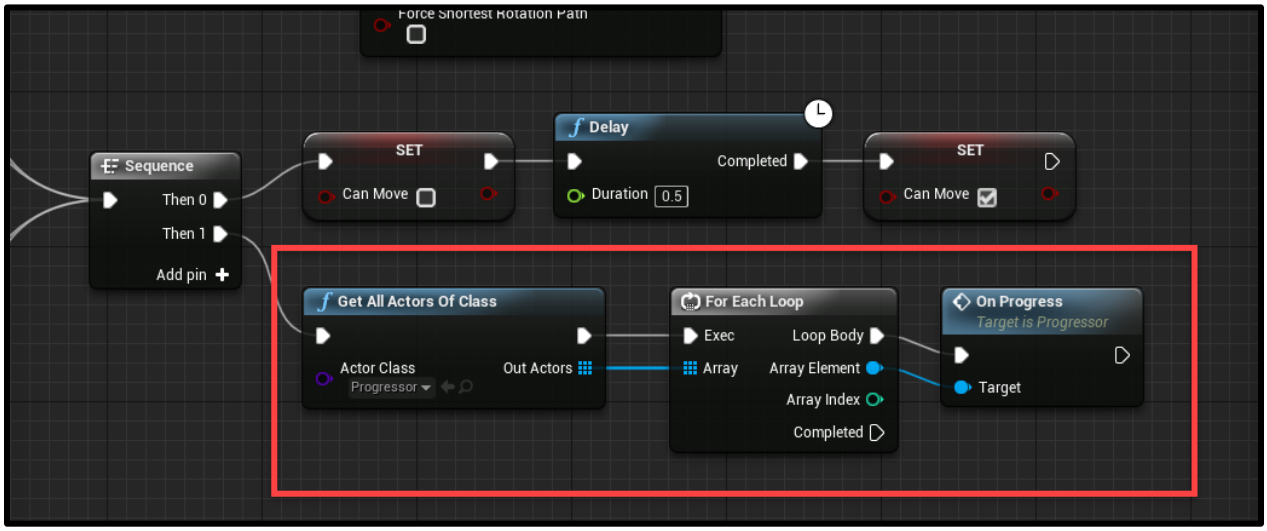
In the *Blueprints* folder, create a new blueprint of type *Actor* called **Progressor**. The only thing we want to do inside of this blueprint, is create a new function called *OnProgress*.



Now that we have the base of all progressors (followers, blades, etc), we can call the OnProgress function when the player moves.

In the **Player** blueprint, create the *GetAllActorsOfClass* node and connect that to the sequence node.

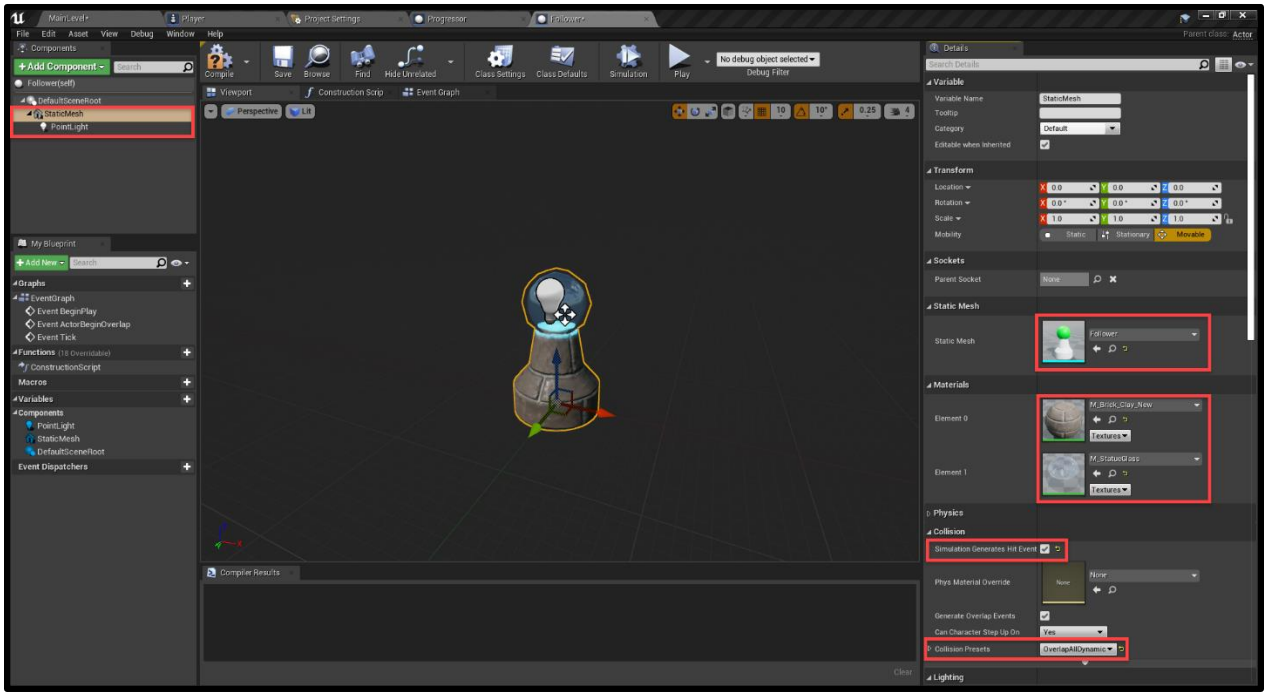
- Set the **Actor Class** to *Progressor*
- Connect that to a for each loop
    - This will loop through every progressor in the level
- We then want to call each progressor's *OnProgress* function



Creating the Followers

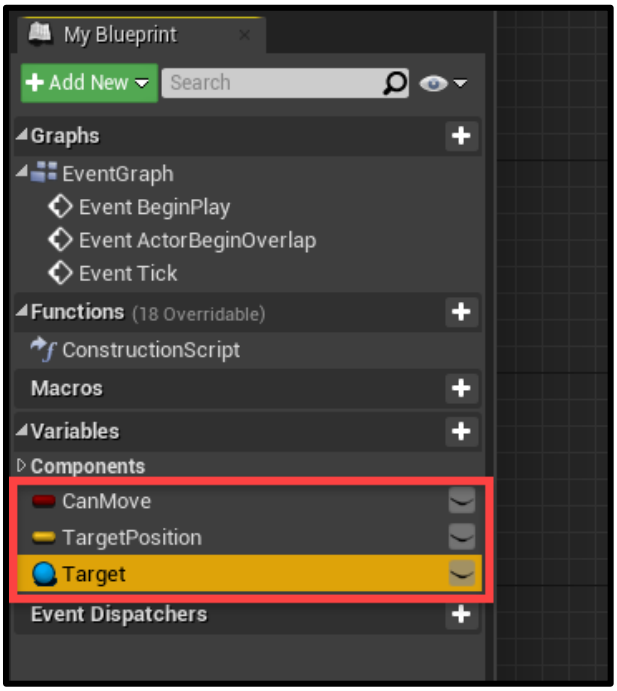
Now that we've got the progressor system in place, let's create our follower blueprint. This is going to be what the player can collect and then have follow them. Create a new blueprint with the parent class of *Progressor*. Call it **Follower**.

- Create a static mesh component
    - Set the **Static Mesh** to *Follower*
    - Set the **Materials** to *M_Brick_Clay_New* and *M_StatueGlass*
    - Enable **Simulation Generates Hit Events**
    - Set the **Collision Presets** to *OverlapAllDynamic*
- Create a point light with blue light

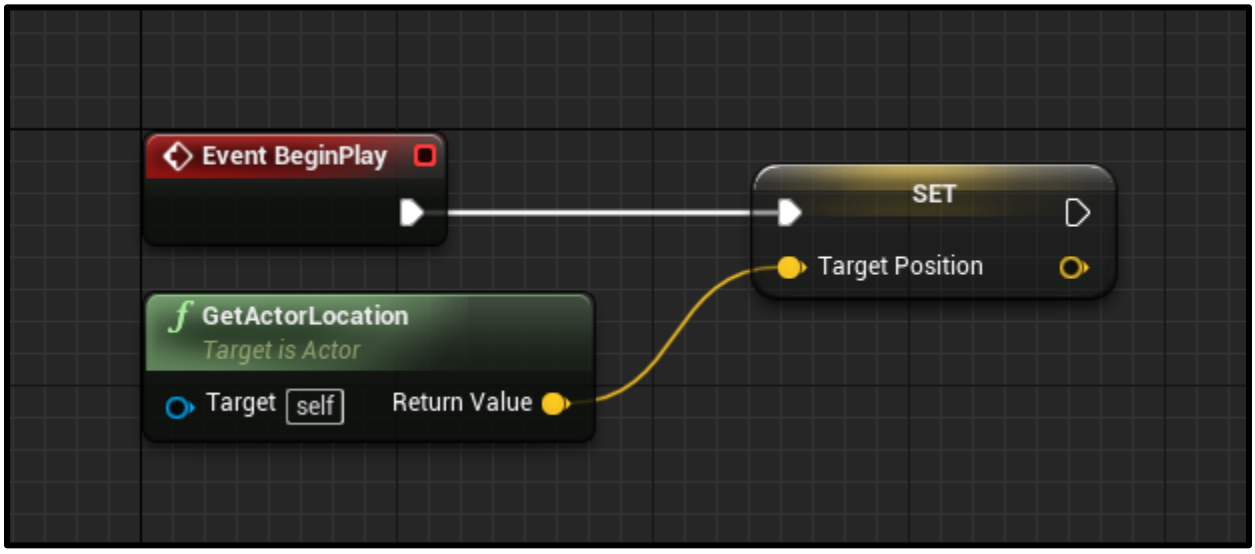


We then want to create three variables.

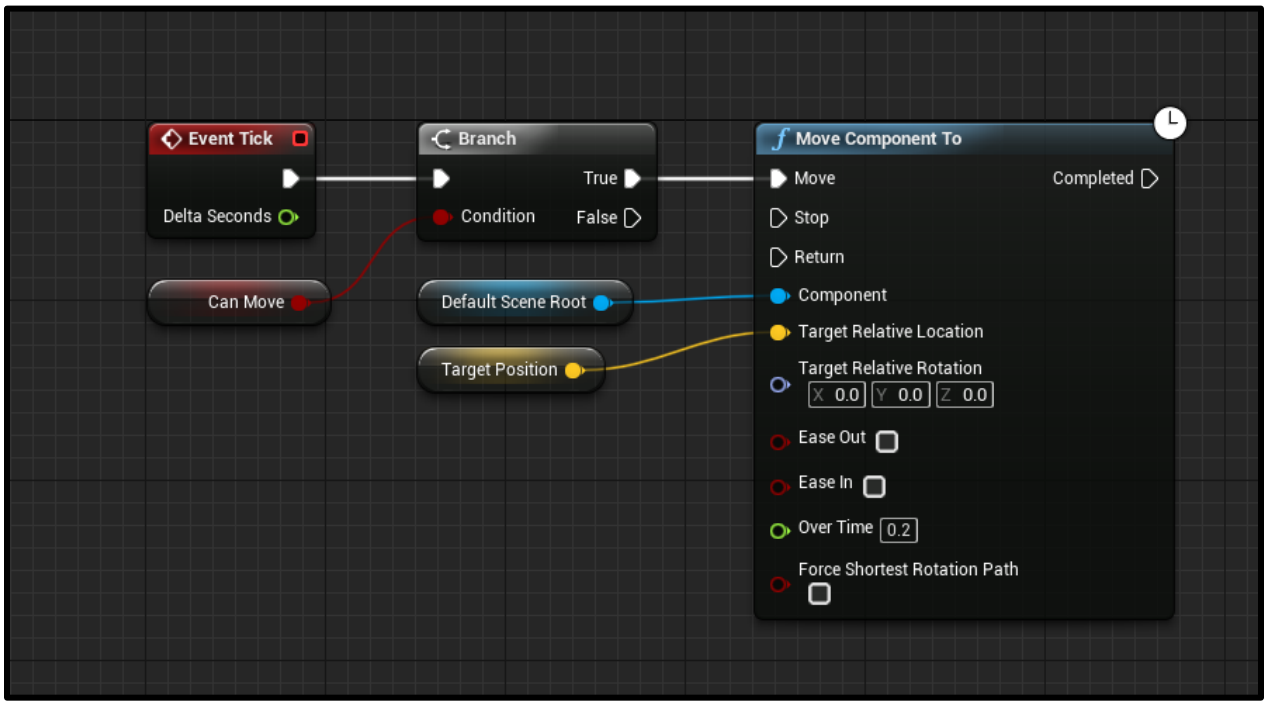
- CanMove (Boolean)
- TargetPosition (Vector)
- Target (Actor)



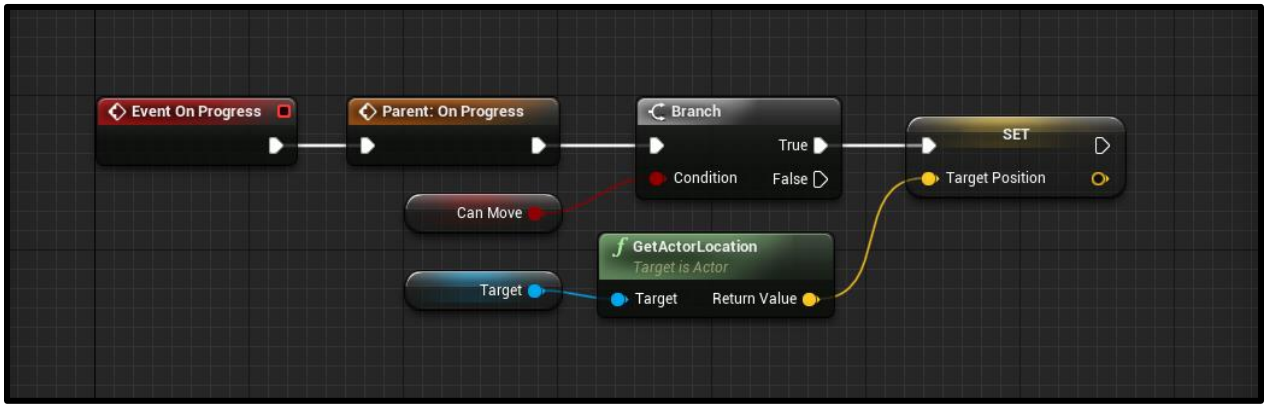
In the event graph, we first want to set the initial target position when the game starts.



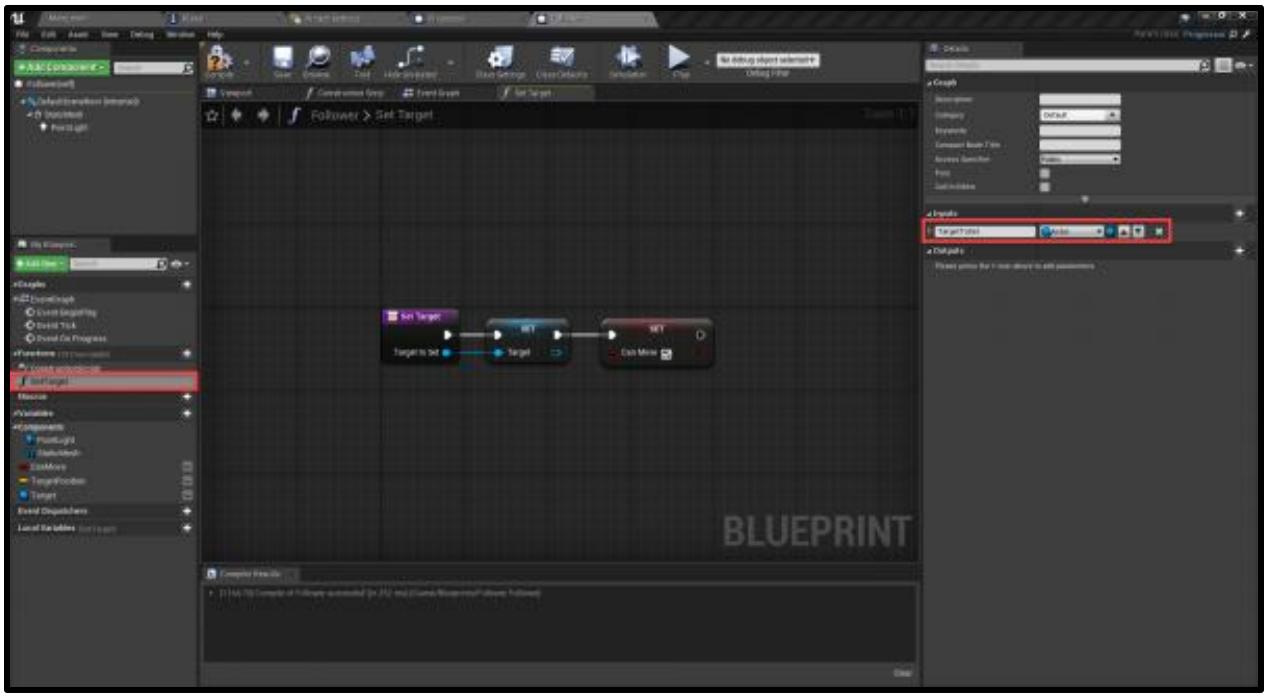
Using the *Event Tick* node, we want to make the follower move towards the target position every frame (only if they can move though).



When the player moves, we're calling all progressor's *OnProgress* function. Well in the follower, we can detect when that's being called and do something. Create the *Event OnProgress* node. We then need to get that parent event call, so right click on the red node and select **Add call to parent function**.

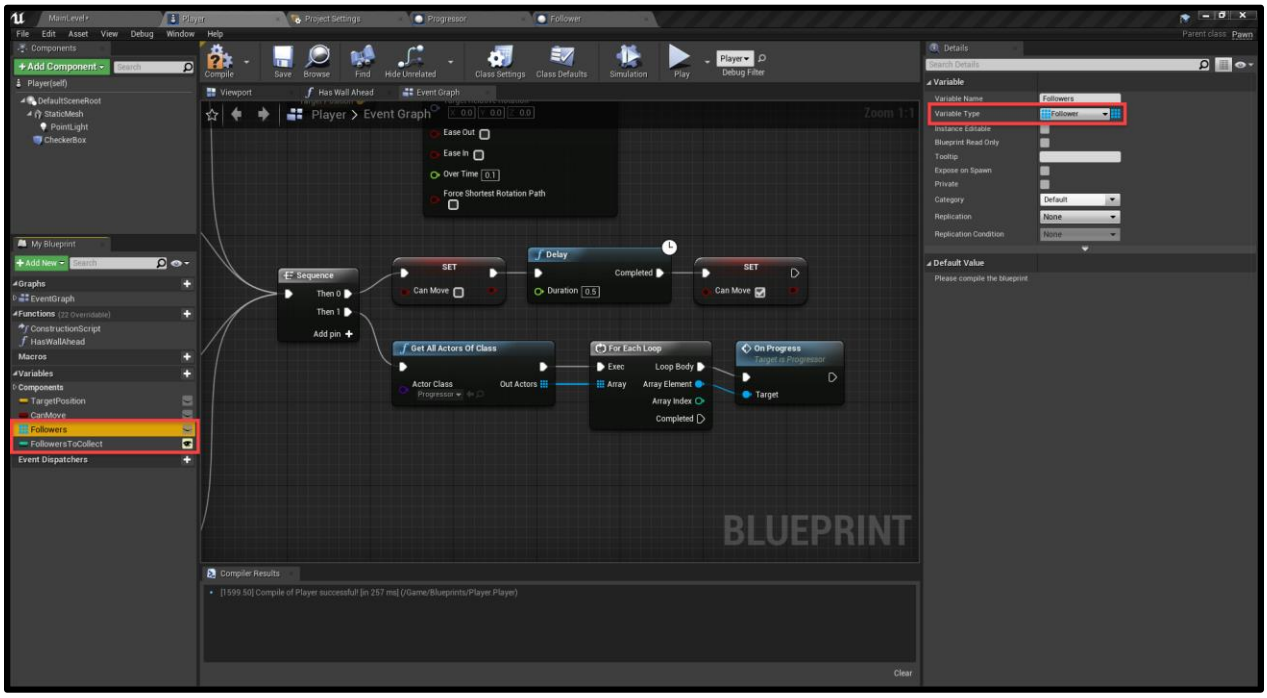


Finally for the follower, we need to create the **SetTarget** function. This will be called when the player collects them, allowing them to follow. Make sure to add an actor input for the target to be set to.

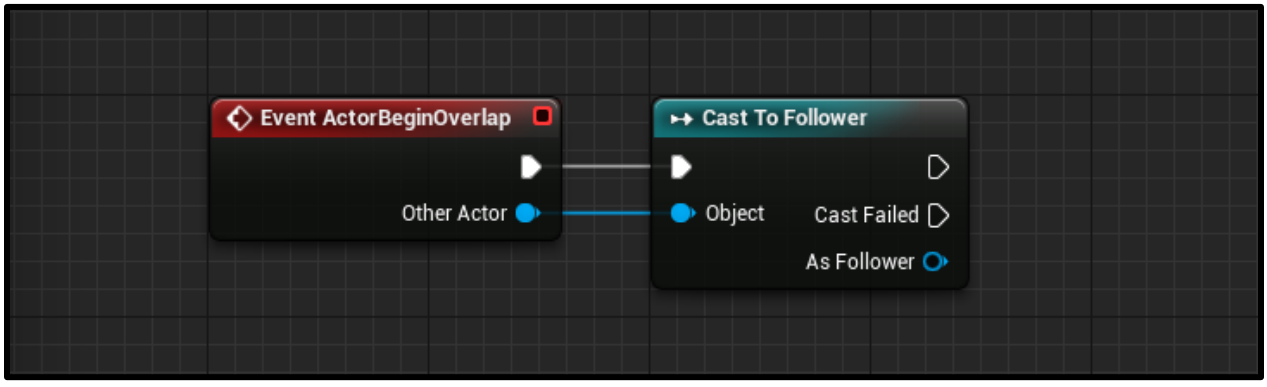


In the **Player** blueprint, we can setup the ability to collect these followers. First, we need some variables.

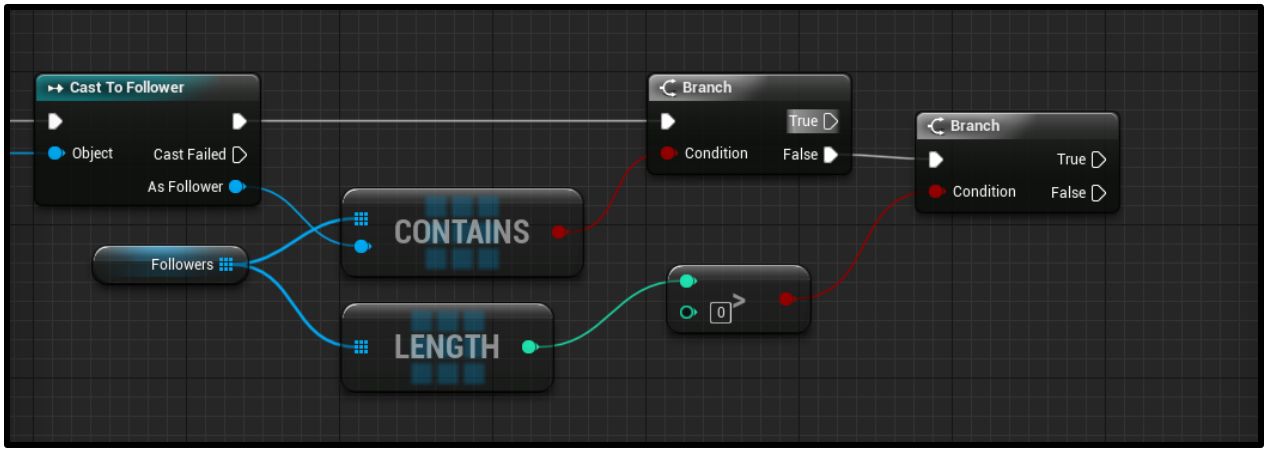
- Create a variable of type *Follower* called **Followers**
    - We want this to be an array, so in the Details panel, click on the icon left of the variable type and choose *Array*
- Create a variable of type *Integer* called **FollowersToCollect**
    - Set this to **Instance Editable**



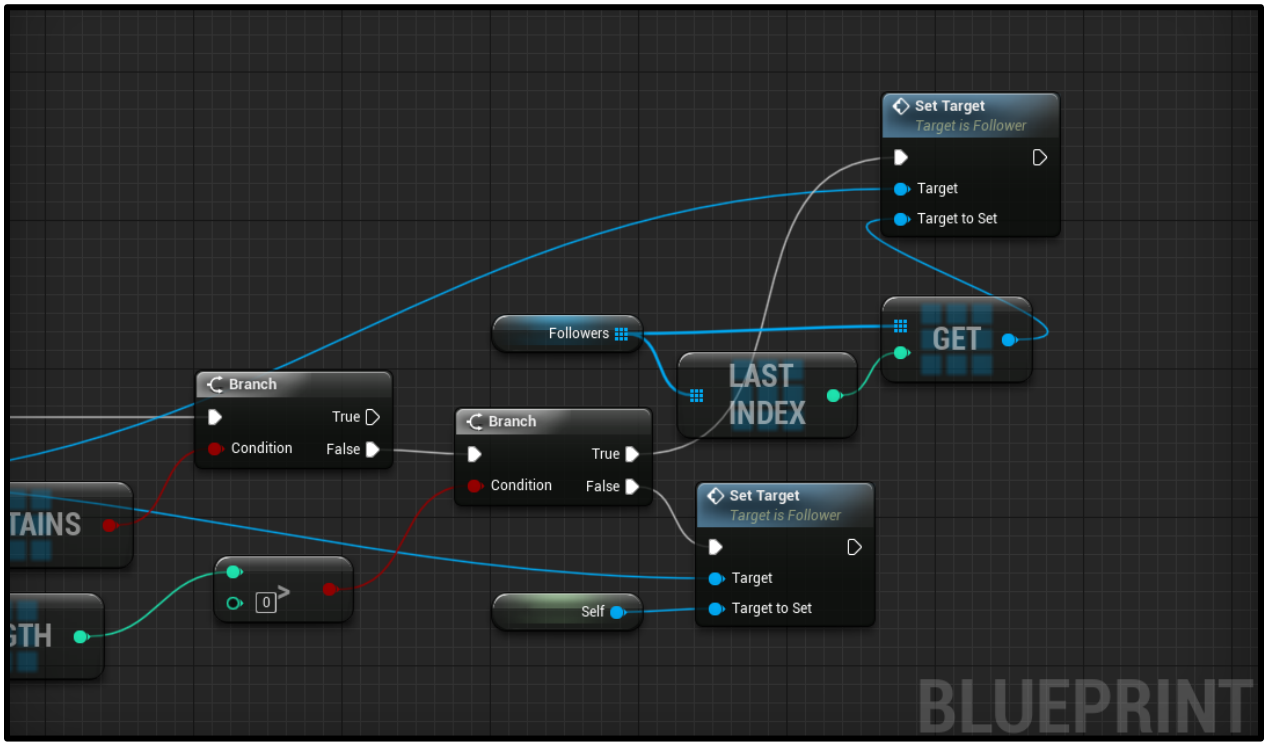
Let's begin by creating an *Event ActorBeginOverlap* node to detect when we've overlapped an object. Then we can cast that to a follower.



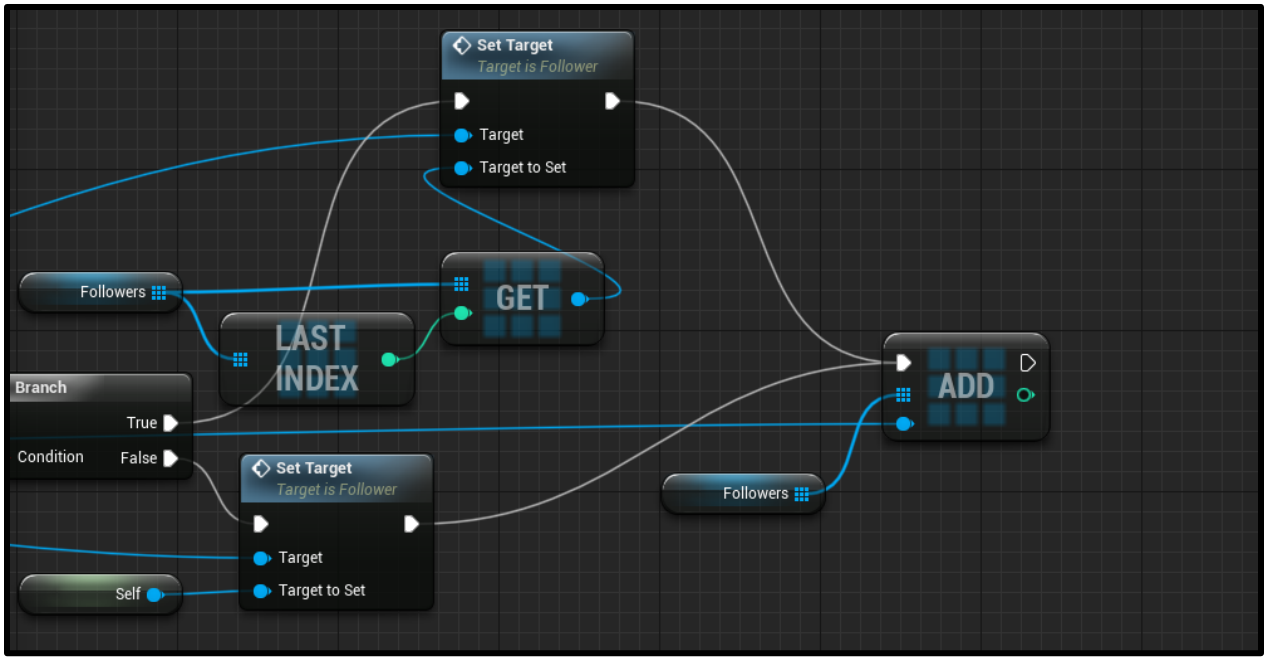
We then need to check if we've already collected the follower. If not, then we need to check if this is the first follower we're collecting.



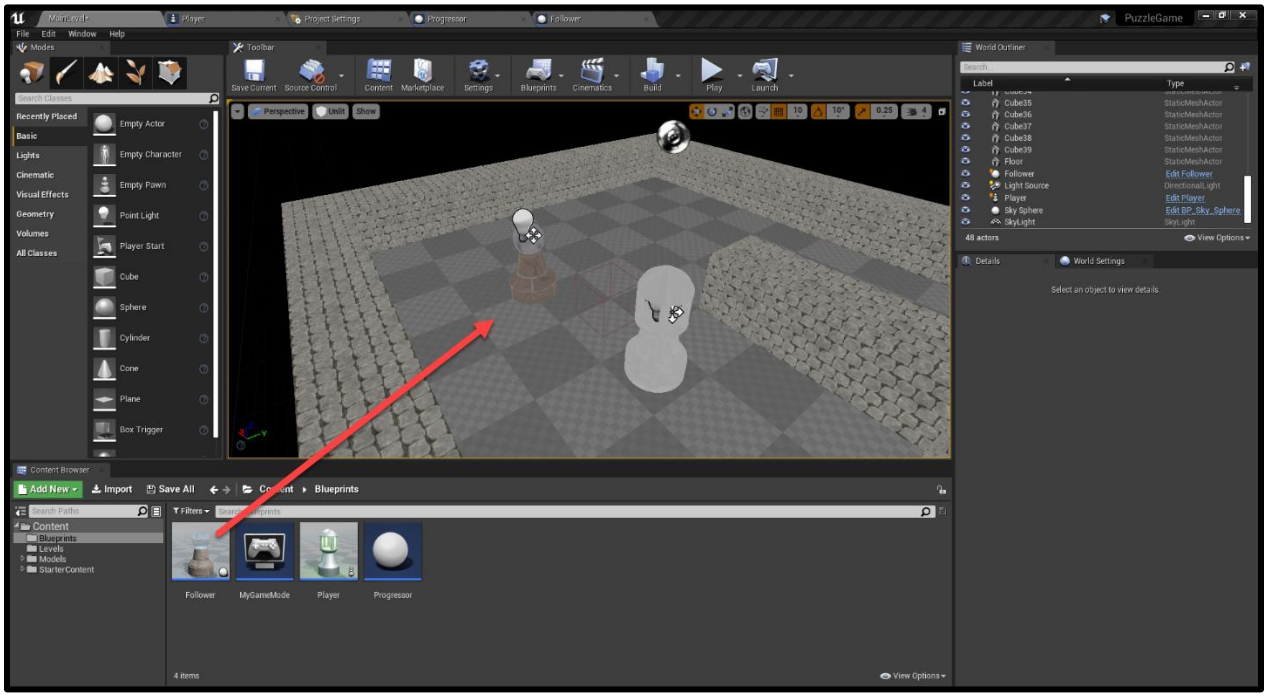
If this is the first follower, then set its target to us. Otherwise, set its target to the last follower in the line.



Finally, we want to add the follower, to the followers array.



Back in the level editor, drag in a follower and press play!

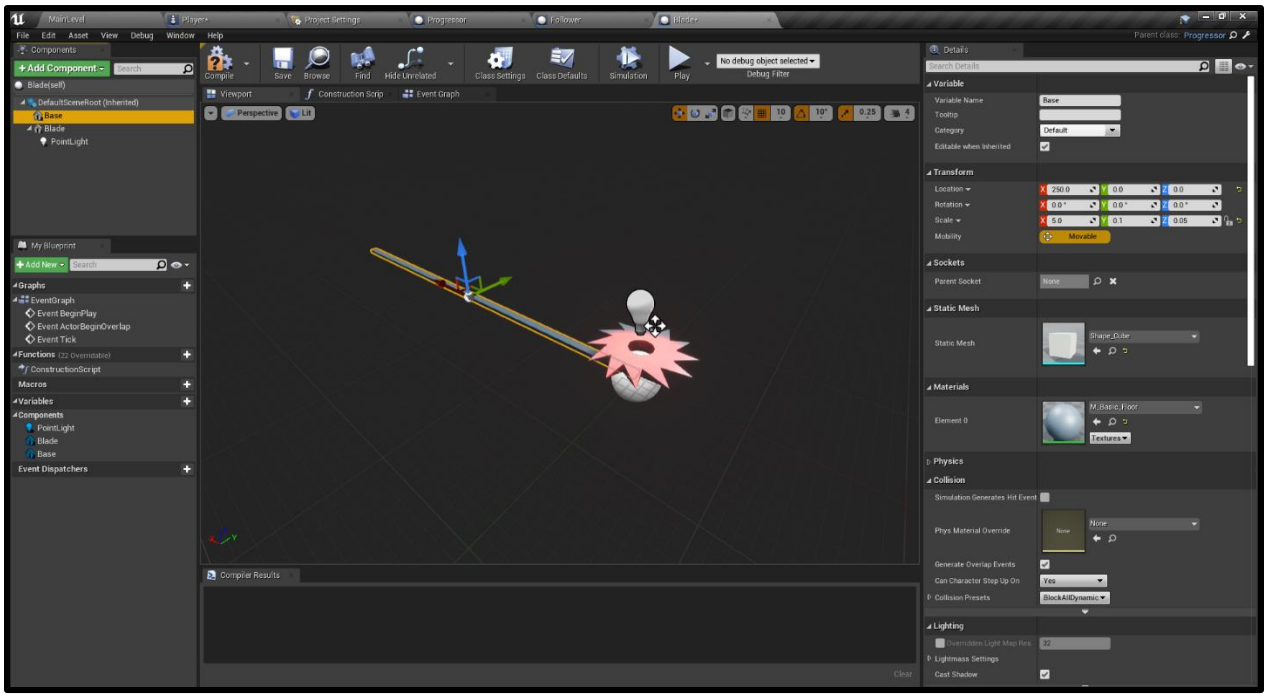


You can now also add in multiple followers and begin a chain!

Creating the Blade

As an obstacle, we can create a blade which will reset the player if they touch it. Create a new blueprint with the parent class of *Progressor*. Call it **Blade**.

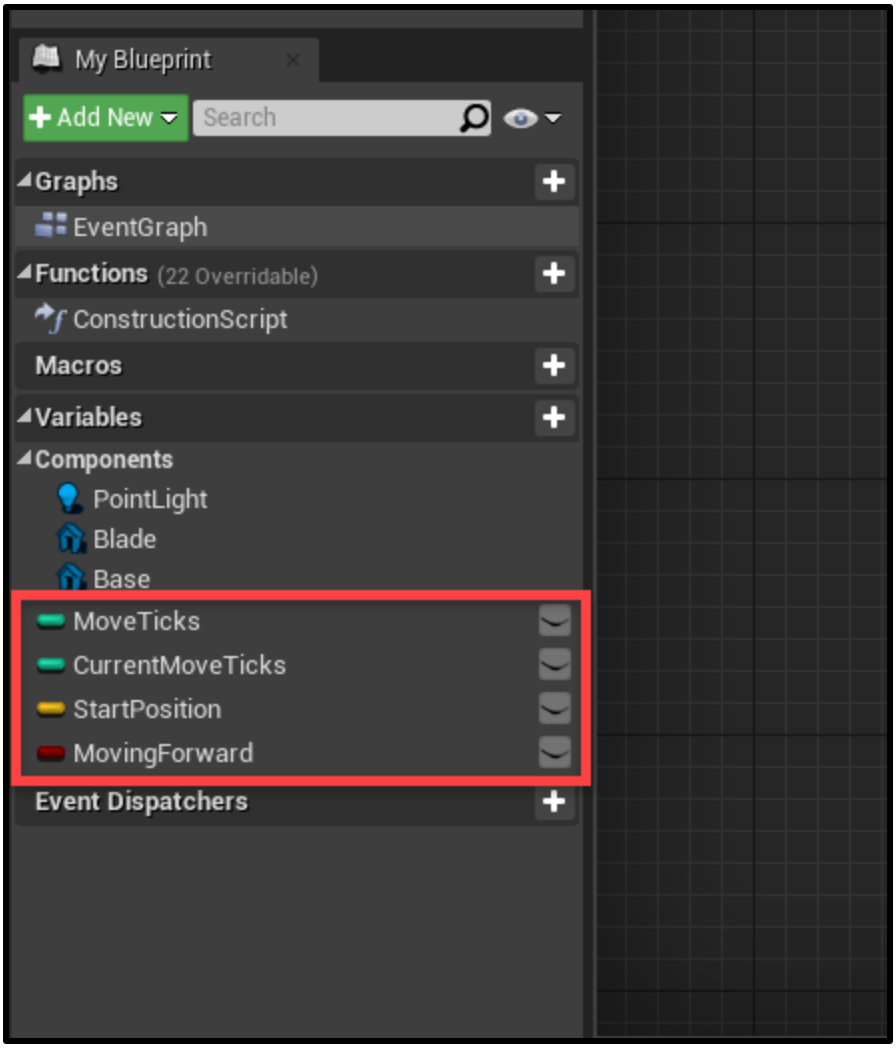
- Create a static mesh component called **Base**
    - Set the **Location** to *250, 0, 0*
    - Set the **Scale** to *5.0, 0.1, 0.05*
    - Set the mesh to *Cube*
- Create a static mesh component called **Blade**
    - Set the mesh to *Blade*
- As a child of the blade, create a point light component



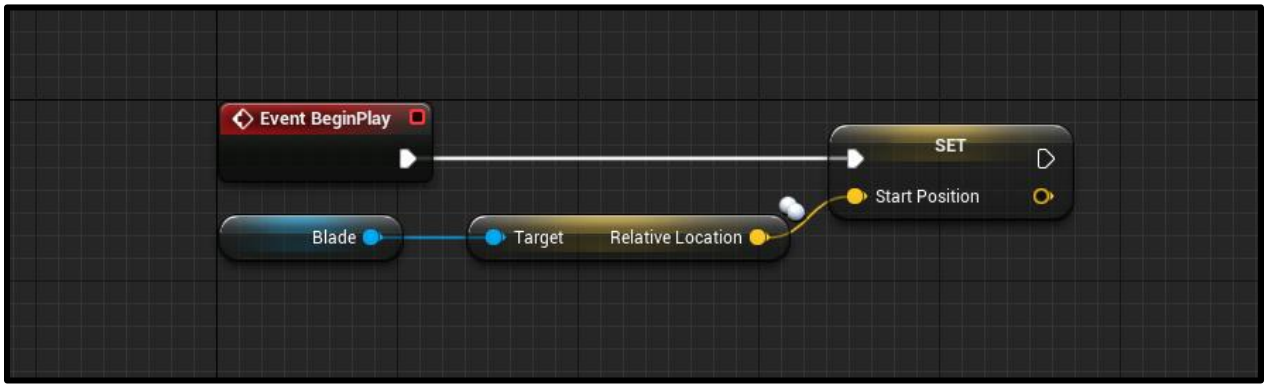
In the event graph, let's start with some variables.

- MoveTicks (Integer)
- CurrentMoveTicks (Integer)
- StartPosition (Vector)
- MovingForward (Boolean)

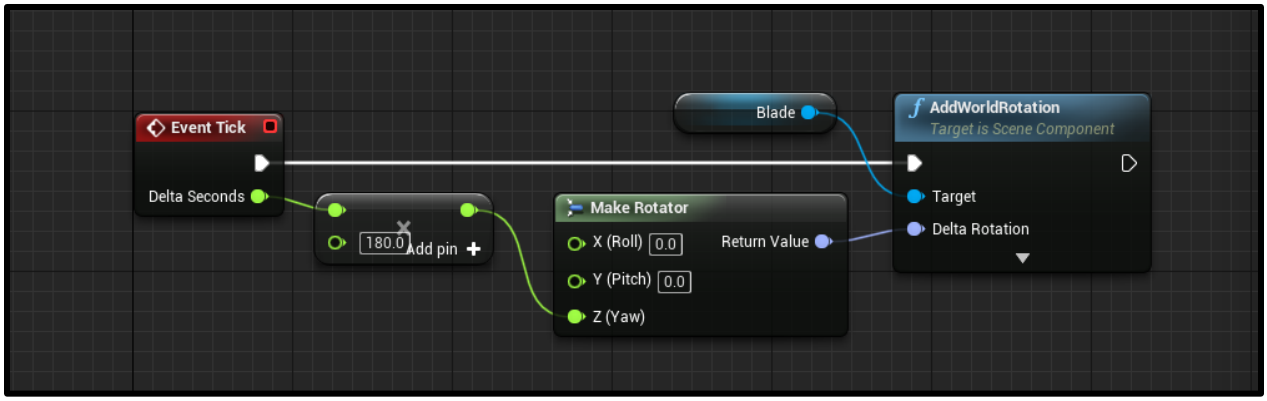
Compile, and set the **MoveTicks** default value to *5*.



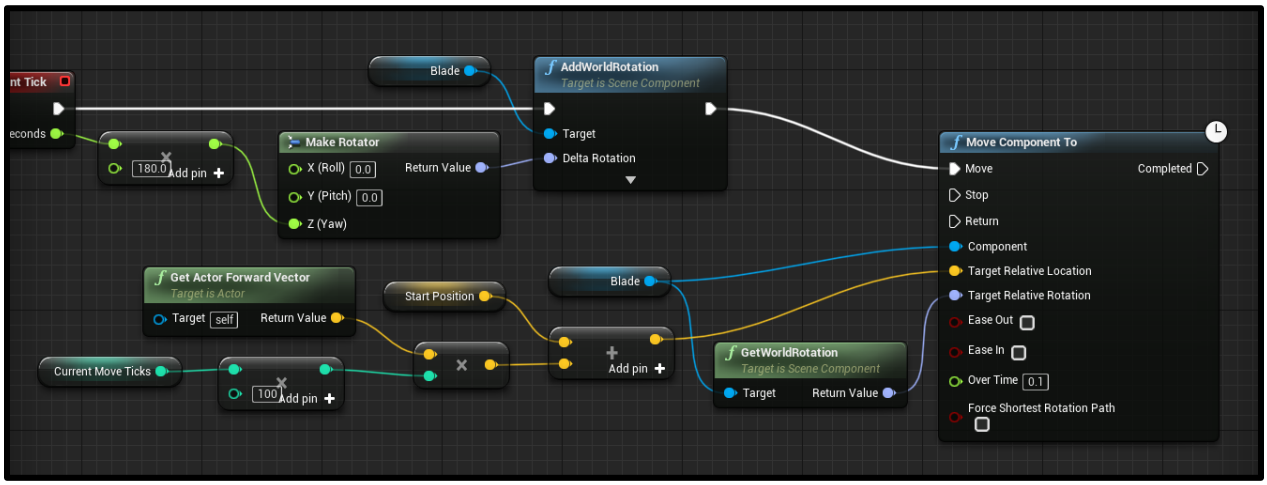
First, we'll have to set the start position.



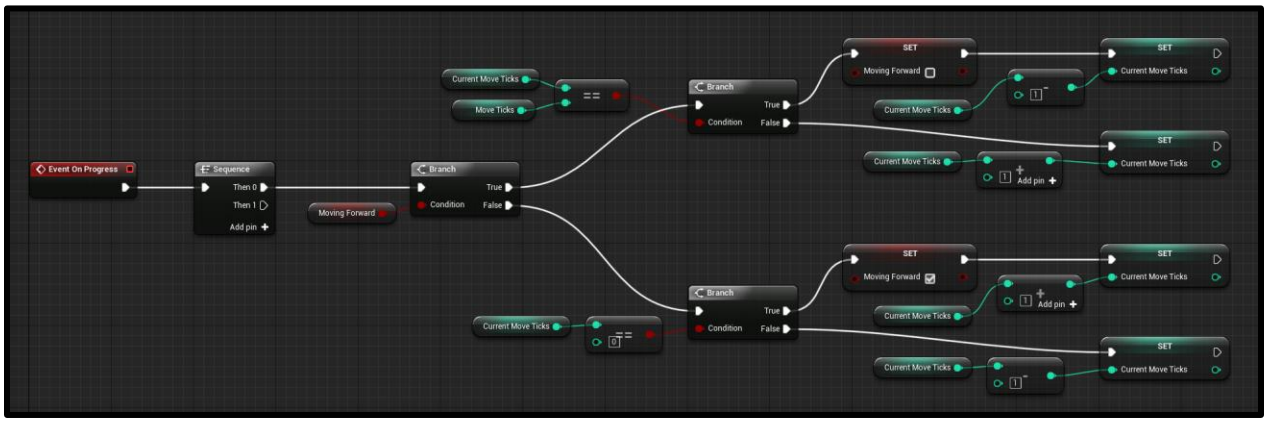
Overtime we want the blade to rotate.



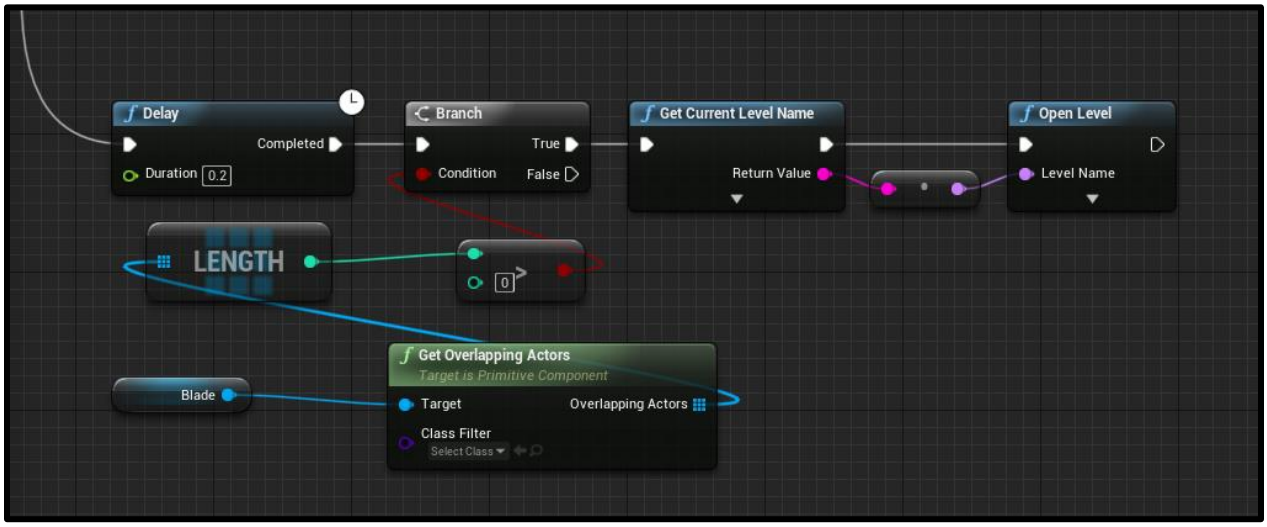
Continuing from that node, we'll make it so the blade moves based on the current move ticks variable.



The next set of nodes will increase or decrease the current move ticks whenever the On Progress event is triggered. This will make the blade move back and forth.



Coming from the second sequence execution output, let's check if the blade has collided with anything. If so, restart the level.

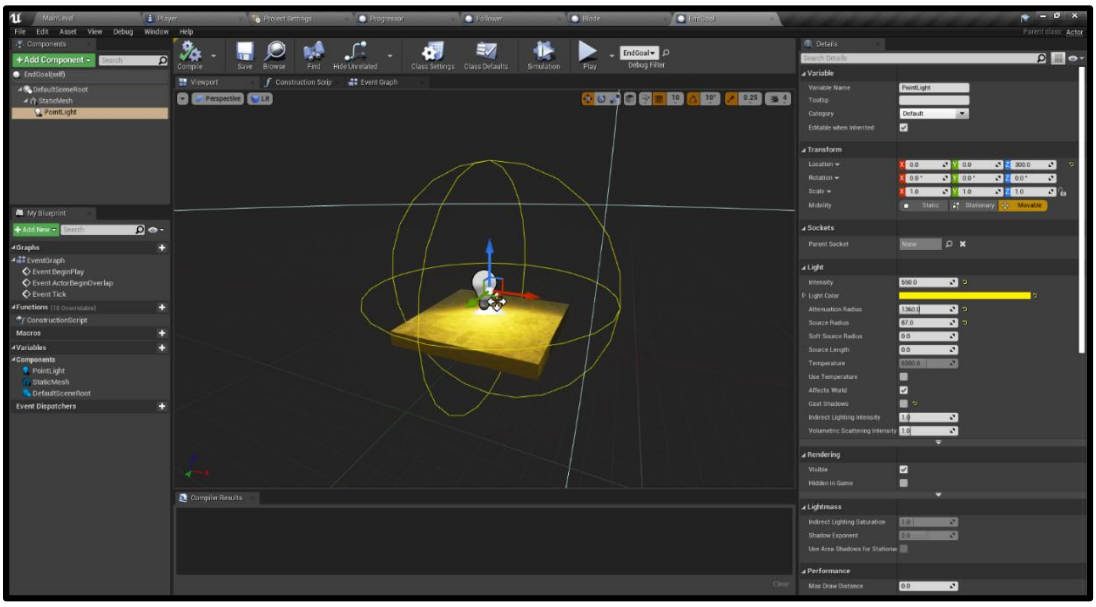


Now we should be able to play the game and reset the scene when colliding with the blade.

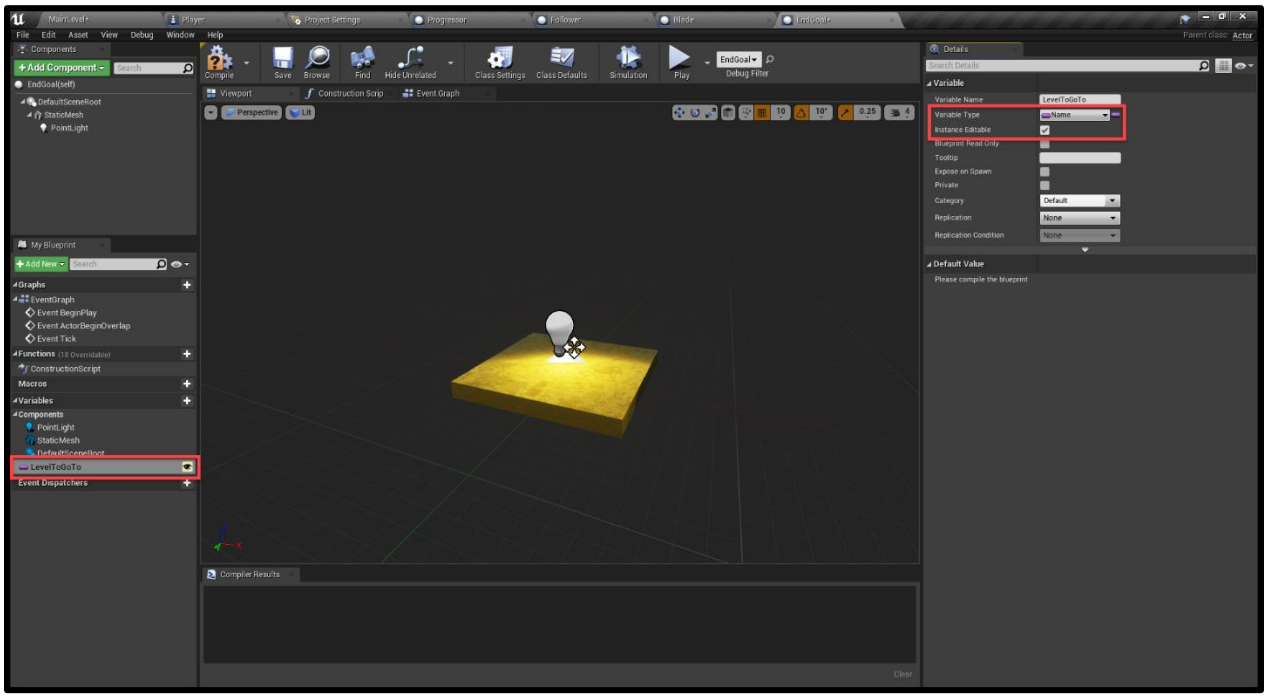
End Goal

Finally, we have the end goal. This will take us to other levels.

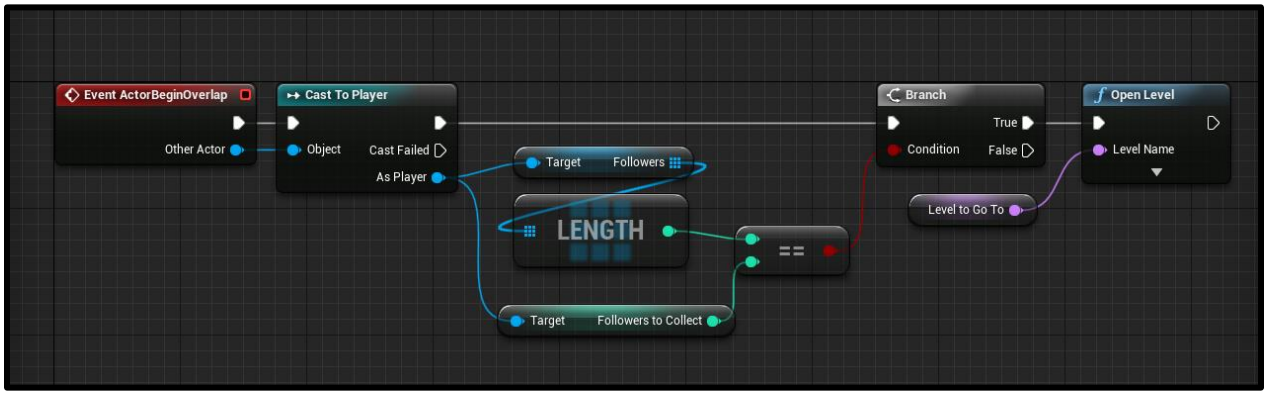
Create a new blueprint of type *Actor* called **EndGoal**. Create a cube static mesh with a light like so:



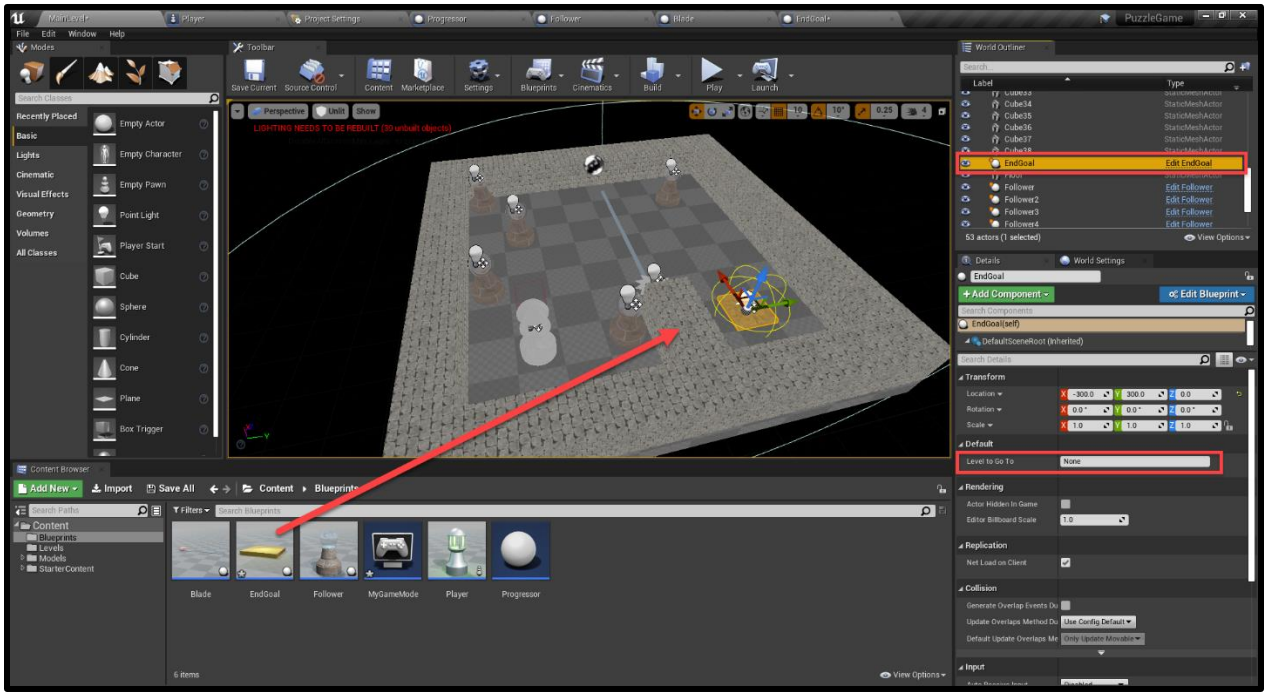
Then, create a new variable called **LevelToGoTo**. Make it of type *Name* and enable **Instance Editable**.



All we're going to do here, is check for collisions. If it's the player, we'll check to see if they have all the followers in the level and if so, open the next requested level.



Back in the level editor, place down the end goal and if you have a level to go to, enter the name in the detail panel.



Conclusion

Congratulations on completing the tutorial!

You now have the basis for a semi-turn based puzzle game. As stated at the beginning, our game features a player who needs to collect a number of followers. Once collected, the player needs to reach the end goal - all while avoiding a blade haunting their every step. Our game features all of this and more, most of which we accomplished with Unreal Engine's built-in features and blueprinting system.

From here, you can expand upon the game, adding in more levels, mechanics, etc. Or you could create an entirely new project with your newfound knowledge.

Whatever you decide, good luck with your future games and we hope you enjoyed developing a puzzle game with Unreal Engine!