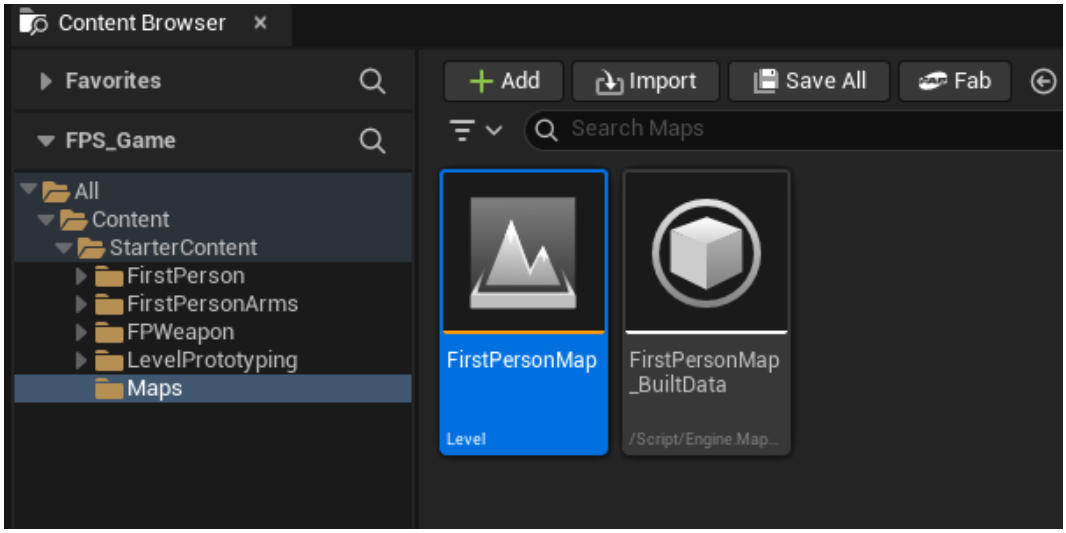
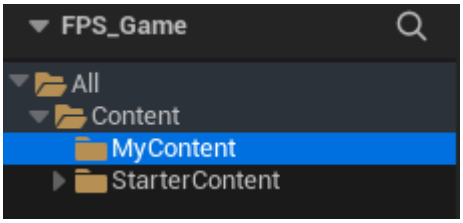
FPS Game Part 1

Setup

1. Download the **FPS_Game** starter project.
2. Unzip to a location on your hard drive (Do not unzip to a OneDrive or cloud synching folder).
3. Double click on the **FPS_Game** Unreal Project icon.
    a. It may ask you to upgrade if you are running a newer version of Unreal. Choose to do this if possible.
4. If your viewport is black, you will need to load the level map. Go to **StarterContent** > Maps then double click on **FirstPersonMap.**



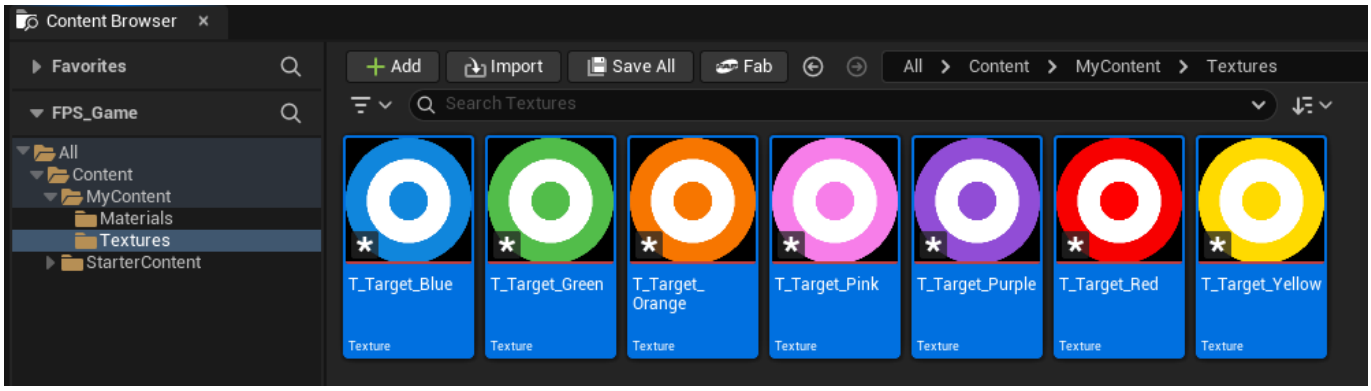
5. Click on Play and check that the game is running.
    a. You should be able to move, pick up the weapon and fire the weapon.
6. We will create a new folder under the Content folder for all our work to go into.
    a. Project management and digital hygiene are of paramount importance in game dev so we will use best practices from here on.
7. RMB on the Content folder and create a new folder called **MyContent**.



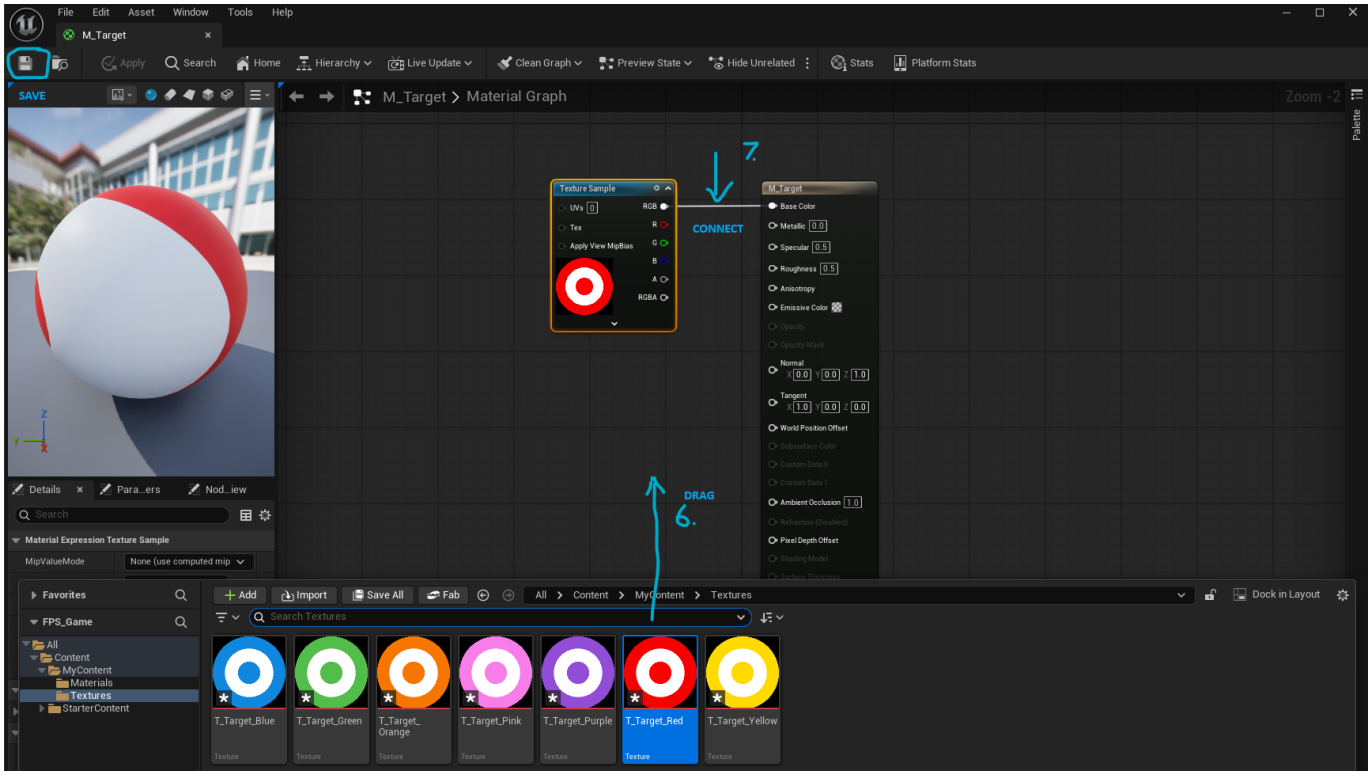
8. It would be good to now set the project up as a **GitHub** repository if you already haven't.
9. Follow the instructions in the accompanying GitHub Setup document.

Creating the Target Master Material

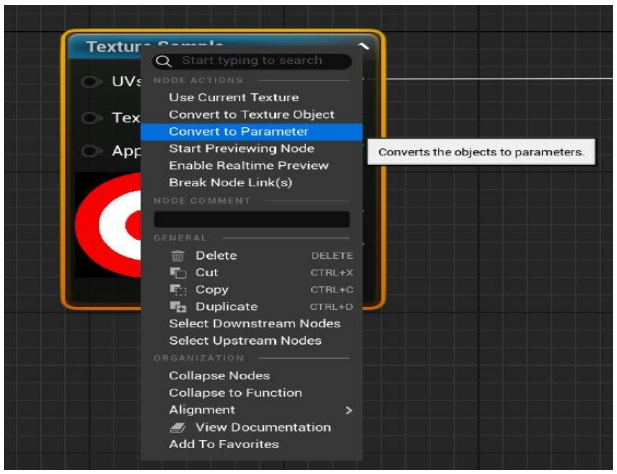
1. In the **MyContent** folder, Create 2 new folders:
    a. **Materials**.
    b. **Textures**.
2. Drag in the Target textures from the downloaded assets to the Textures folder.
    a. You will see that they all start with T_. Everything in Unreal has specific prefixes to make it easy to find and identify the type. T_ = Texture file.

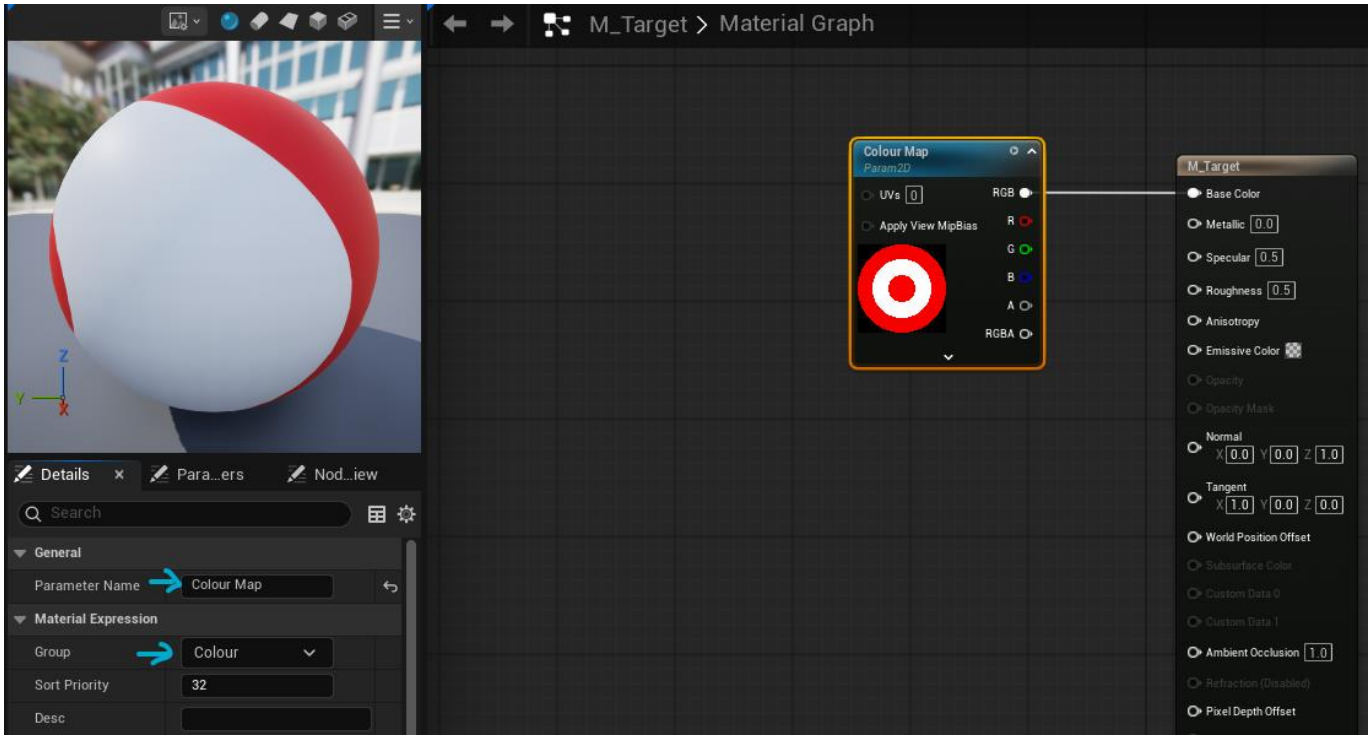


3. In the Materials folder, RMB and select **Material.**
4. Rename this to **M_Target**.
    a. Naming convention: M_ = Material file.
5. Double click on the **M_Target** asset to open the Material Editor.
6. Select the Content Drawer > Drag and drop the red target texture into the node graph.
7. Drag and drop the Texture Sample node's RGB output pin, onto the **M_Target** node's **Base Colour** input pin.

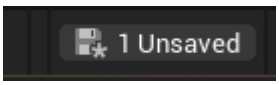


8. RMB click on the Texture Sample node > Select **Convert to Parameter.**





9. Name the node **Colour Map**.
10. Select the Colour Map node > Details panel > Under the Material Expression category > Select Group > Type in **Colour**.
11. **Save** the Material.
12. Close the Material Editor.

*Note if you see an asteric (*) on something in the Content Browser, it means that it hasn't been saved in the project. To fix, you can CTRL+SHIFT+S or click on the button in the bottom left of the editor. 

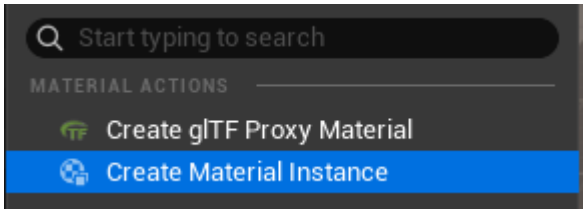
For further details on Materials, read the documentation here:
[Unreal Engine Materials | Unreal Engine 5.6 Documentation | Epic Developer Community]

Creating the Target Material Instances

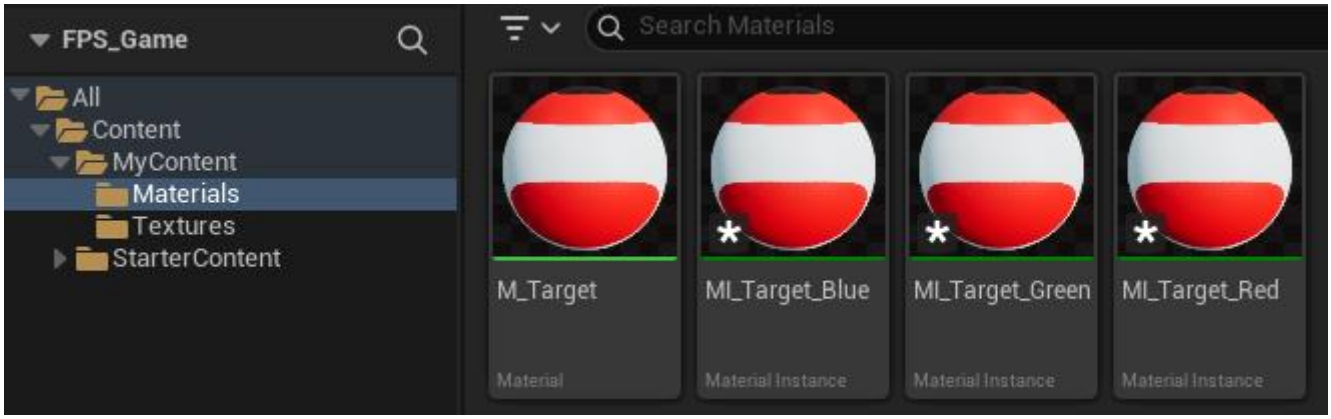
Now that we have created the master material, we want to have a different material for the red target, a different one for the blue, green, etc.

To do this, we need to create instances of the master material it to change its properties. We will not use the master material on our objects, but the instance versions. That way if we need to change anything that we want to apply to all materials (for example, to make them emissive) we just change the master material and it will apply to all the instances of the material. We can then make individual changes on the instance material and it will only apply to that material.

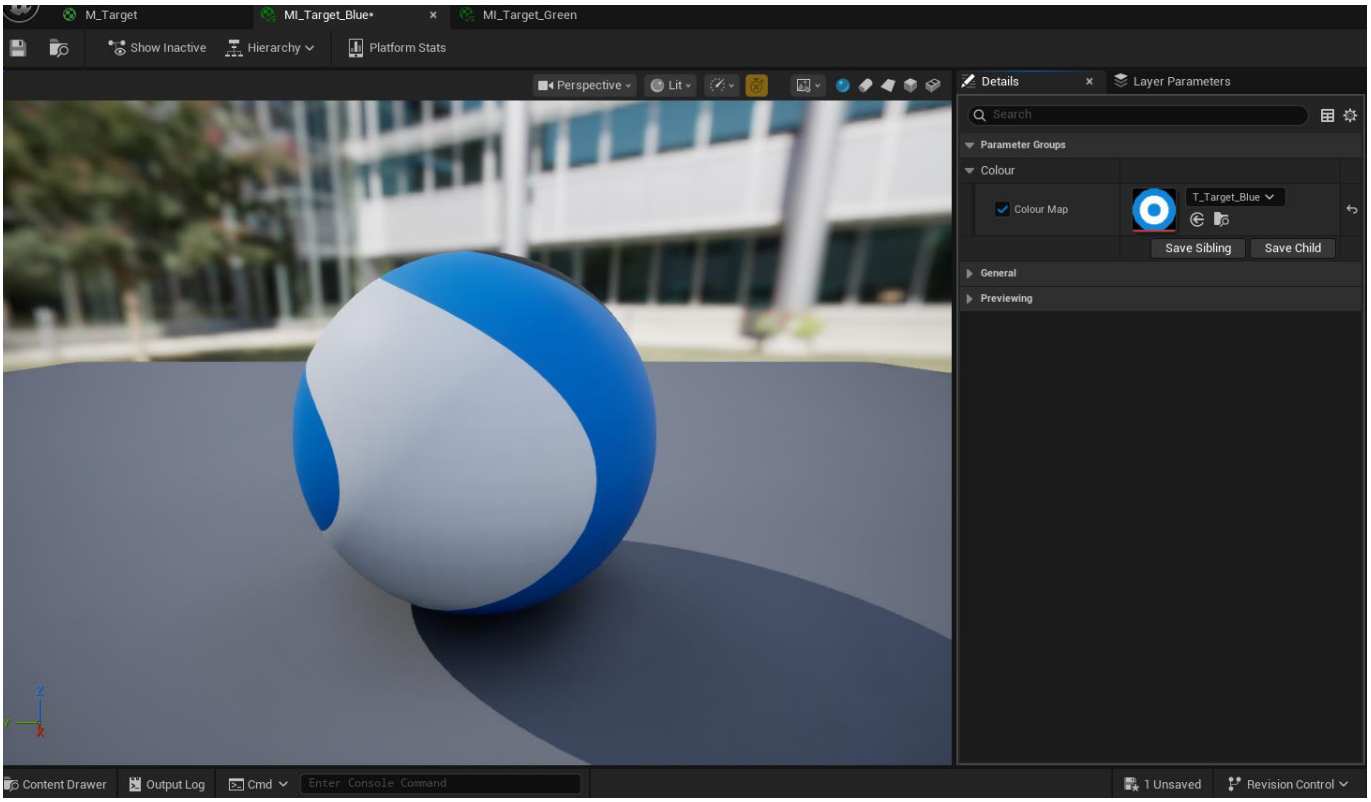
1. In the MyContent/Materials folder, RMB on the **M_Target** material and select **Create Material Instance.**



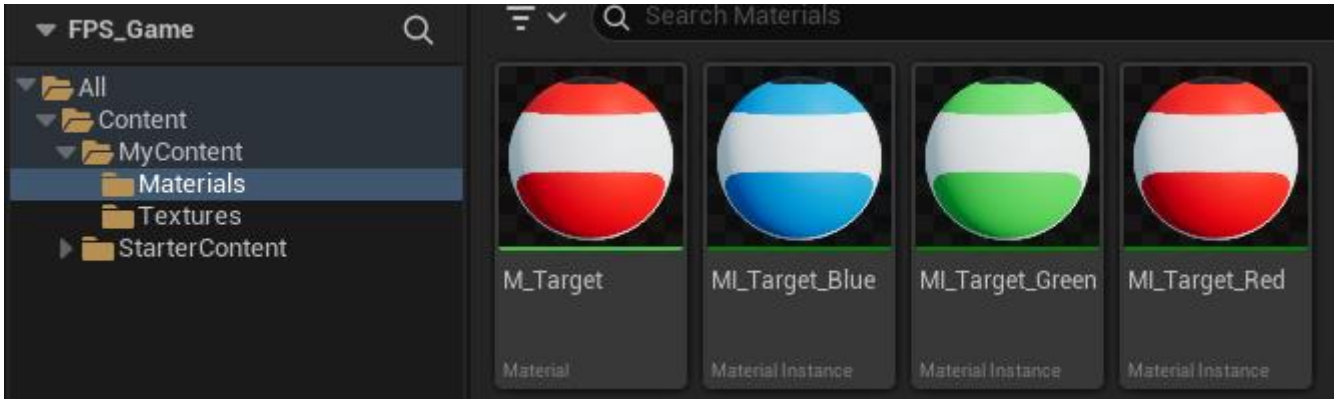
2. Rename this to **MI_Target_Red.**
    a. MI_ = Material Instance.
3. Do it 2 more times, this time naming them **MI_Target_Blue** and **MI_Target_Green**
    a. You can add more for the other colours later.



4. You will see that they are all currently red.
5. Double click on the **MI_Target_Blue** to open the material Instance.
6. Under Parameter Groups you will see an option for Colour. Check on the box and you can then change the texture to the **T_Target_Blue** one.
7. Save the material Instance.



8. Do this again for the **MI_Target_Green**, changing the Colour Map to the **T_Target_Green**
9. You should now have material Instances for each colour.

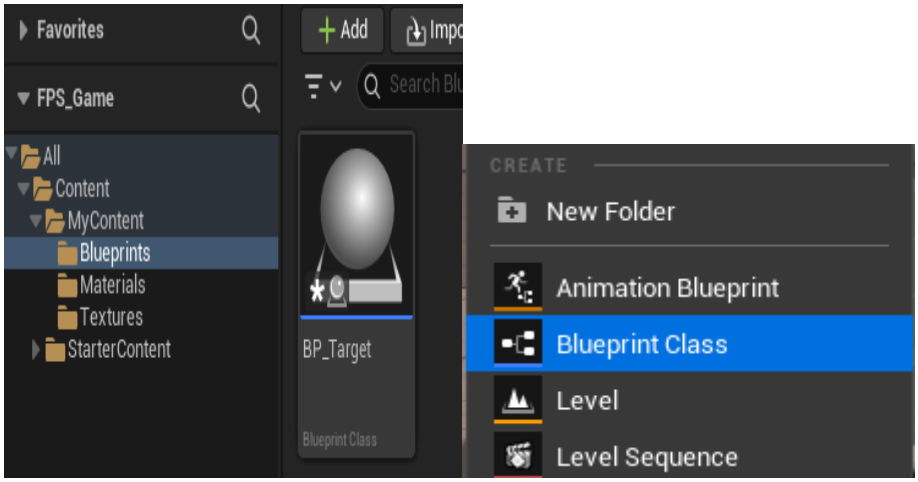


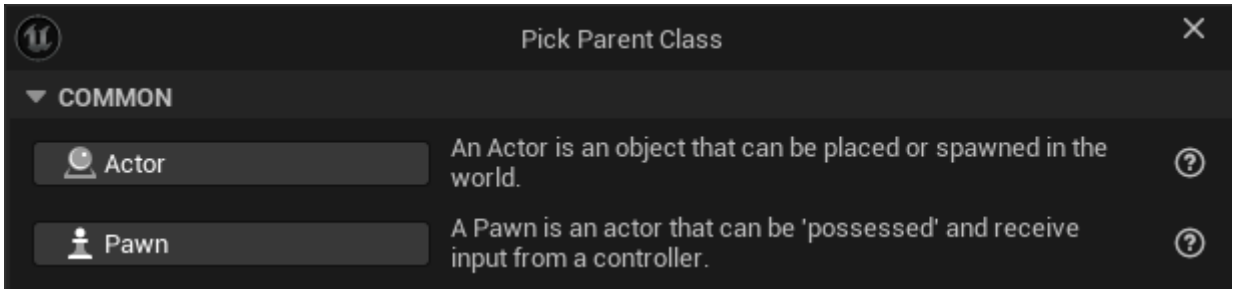
For further details on Material Instances, read the documentation here:
[Creating and Using Material Instances in Unreal Engine | Unreal Engine 5.6 Documentation | Epic Developer Community]

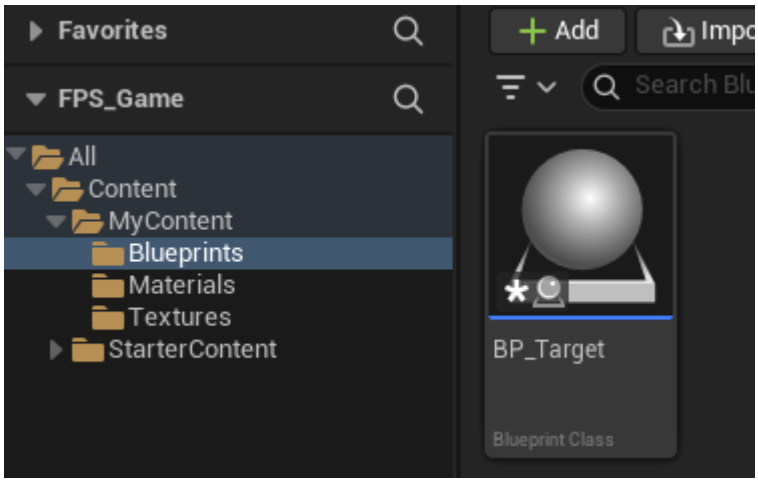
Creating Our Target Blueprint

It is time now to create our targets. For all our target functionality, we will wrap this into a blueprint. Recall from the notes, that a blueprint is what contains all the logic for a particular part of our game. Our target blueprint will contain the variables and functions needed for it to behave.

1. In your **MyContent** folder, RMB and create a new folder called **Blueprints.**
2. In the **Blueprints** folder RMB and select **Blueprint Class.**
3. In the 'Pick Parent Class' popup window > Select **Actor**.
4. Name the actor Blueprint > **BP_Target.**
5. Double click the **BP_Target** asset to open it in the Blueprint editor.



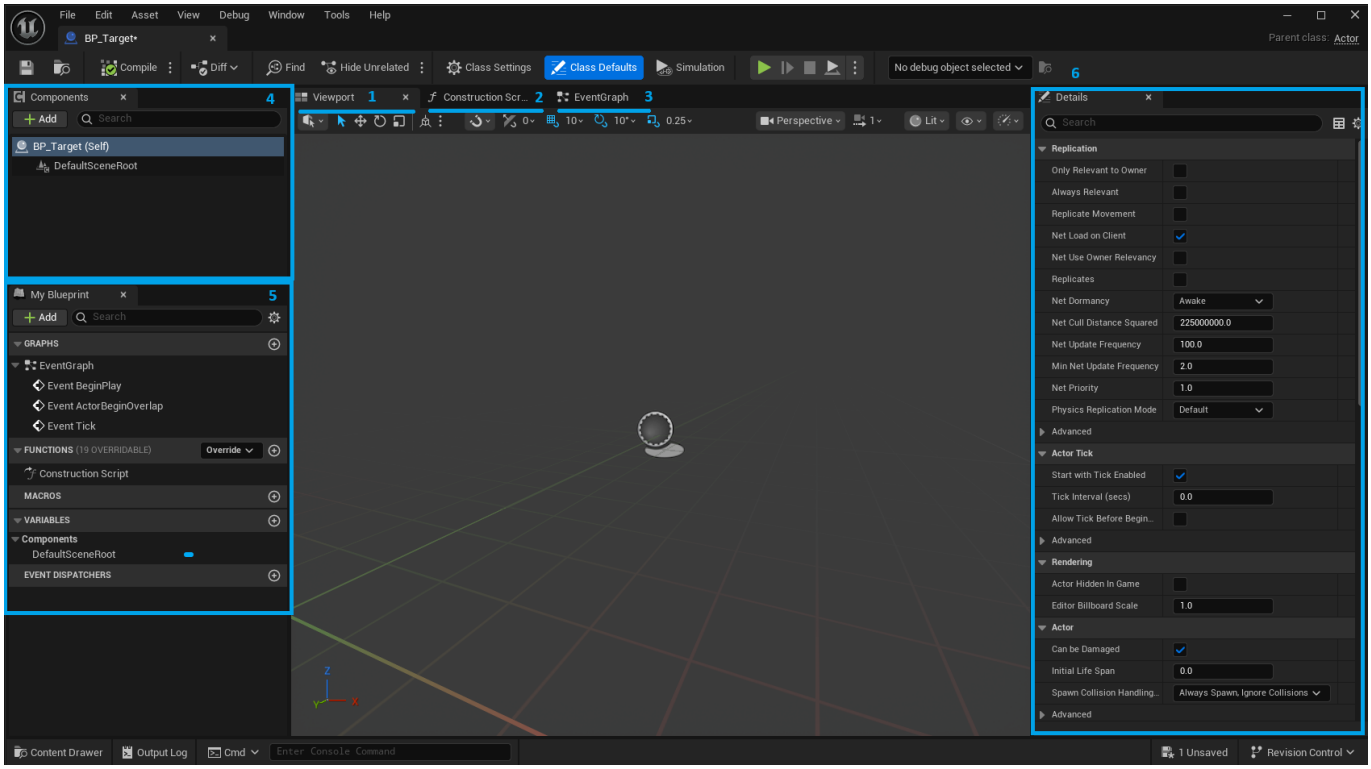




The Blueprint Editor

The Blueprint Editor is comprised of several different panels and tabs:

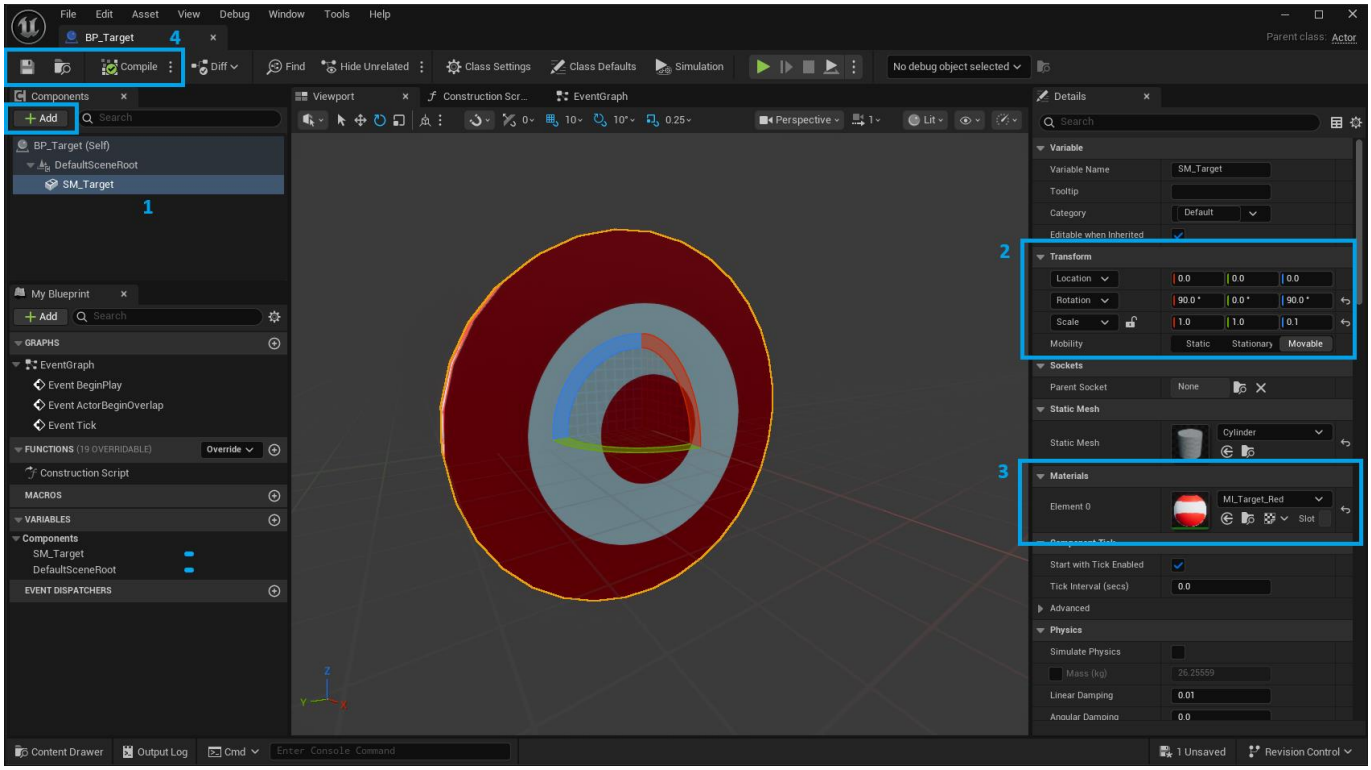
1. **The Viewport Tab:**
    a. Similar to your typical 3D viewport, this is where you'll build your game objects and add components to them.
2. **The Construction Scripts Tab:**
    a. This tab is used to build functionality, to perform initialisation actions when instances of a Blueprint are created.
3. **The Event Graph Tab:**
    a. In this tab, you'll build your actor logic using Blueprints.
4. **The Components Panel:**
    a. Add different components to your game object / actor.
5. **The My Blueprint Panel:**
    a. Displays all the elements within the game actors Blueprint.
    b. This includes event graphs, functions, macros, variables, and event dispatchers.
6. **The Details Panel:**
    a. Tweak settings on individual components.



Making the Target Object

The Blueprint Editor is comprised of several different panels and tabs:

7. Add a Mesh component for Our Target:
    a. In the Components panel > Select '**+ Add**' button to add a component.
    b. In the Basic Shapes section > Select 'Cylinder'. Alternatively, you can search for 'Cylinder' in the search bar.
    c. Rename the Cylinder to > **SM_Target**.
    d. Naming Convention: SM_ = Static Mesh.
8. With the **SM_Target** selected > Details panel:
    a. Adjust the Transform component:
        i. Rotation X: 90.
        ii. Rotation Z: 90.
        iii. Scale Z: 0.1.
9. Materials > Apply the **MI_Target_Red** material instance.
10. Compile and Save
    a. You must always compile your Blueprints. Compiling checks for errors and ensures that the code can run.



Target Variables

First we'll start by working out what variables we need and of what data type they are:

- **Direction** – Where will the target move to [Vector].
- **Speed** – How fast the target will move [Float].
- **MovementDuration** – How long it moves before changing direction [Float].
- **HitPoints** – How many hits can the target take [Integer].

Creating the variables

1. In the My Blueprint panel > Variables section > Select the '+' icon to create a new variable.
2. Create a variable called **Speed** and make it **Float** type.
    a. To change the data type, click on the dropdown next to the data type.
3. Create a variable called **MovementDuration** and make it **Float** type.
    a. We generally do not have spaces when we use multiple words in a variable or function name. This is known as Pascal case and is the default in Unreal.
4. Create a variable called **Direction** and make it **Vector** type.
5. Create a variable called **HitPoints** and make it an **Integer** type.

Now that we have the variables setup, we need to make them **Instance Editable** (they can be changed individually on different target blueprints to make them unique).

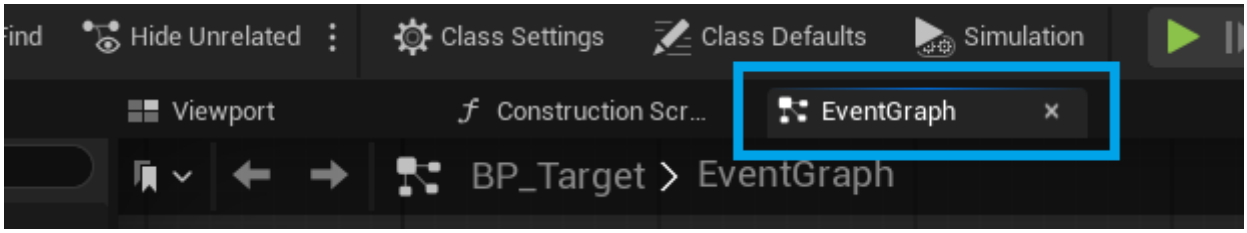
6. Next to each variable you will see a closed eye. Click on that to open the eye and make it **Instance Editable**.

We also want to set some default values for each variable. To do this we must first **Compile** the blueprint (top left corner of the window).

7. Select the **Speed** variable, and in the details panel you should be able to change the Default Value. Set it to 1.
8. Select the **MovementDuration** variable > Set the Default Value to 3.
9. Select the **Direction** variable > The default value for this variable is three float values. These will set the direction the target will travel, on the global X, Y and Z axis. For now, let's set the Y value to 20. This will move the target to the left.
10. Select the **HitPoints** and set its Default Value to 3.
11. **Compile** again and **Save** the Blueprint.

Target Movement Behaviour

We need to now change to the Graph Editor view. Click on the Event Graph tab near the top of the Blueprint.

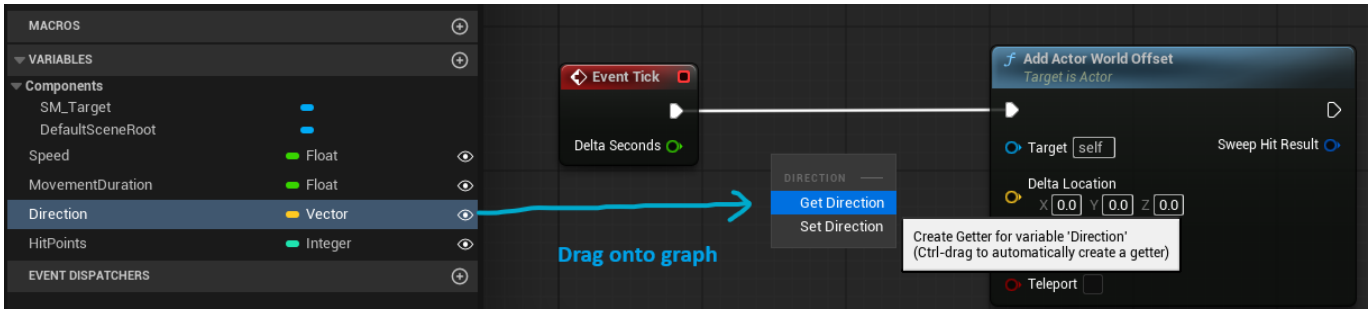


You will see a few nodes in the Event Graph. These come by default whenever you create a new Blueprint. They all start with the keyword Event followed by what they do.

- The Event **BeginPlay** node will execute whenever the game begins playing once only.
- The Event **ActorBeginOverlap** node will execute whenever this actor (object) overlaps (intersects) with another object. Generally used for trigger zones and events.
- The Event Tick node will execute every frame of the game. Generally, this will run 60 times per second.

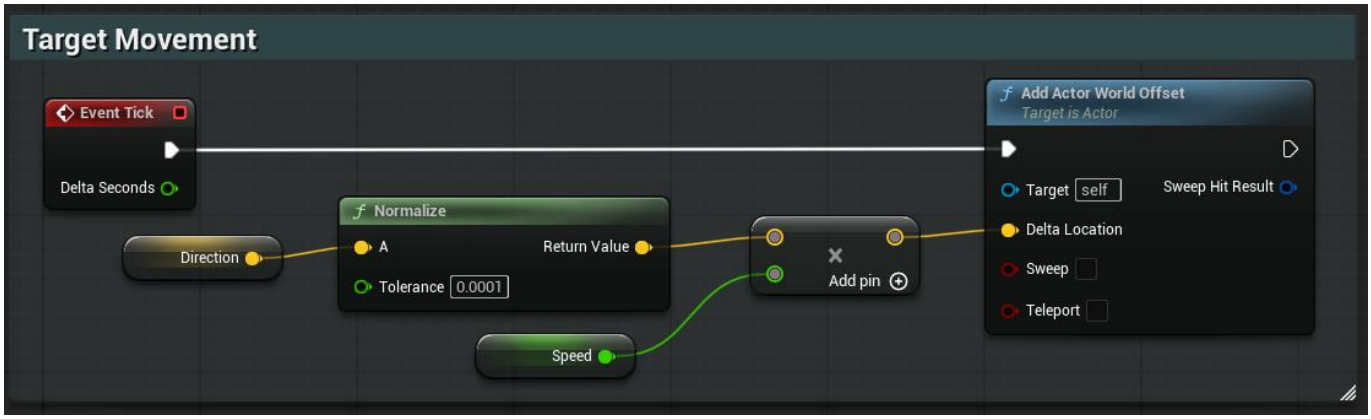
There are many other in-built Events that we can use. In this course we will only use a handful.

1. Start by deleting the **ActorBeginOverlap** node, as we won't be needing this.
2. From the **Event Tick** node > Drag a connection wire off the execution pin then let go in empty space > A search window of different nodes will appear.
3. Search and add the **Add Actor World Offset** node. This node will be responsible for moving our targets in a direction.
    a. The Target will be the target object itself. It should read as 'Self' on the node.
    b. The Delta Location will need a reference to the direction that the target will move in.
4. From the My Blueprint panel > Drag and drop the **Direction** variable into the graph.
    a. A small window will popup asking you to either get or set the variable.
    b. The **Get Direction** gives us access to default value stored in the variable.
    c. The **Set Direction** allows us to change the default value.
    d. Select **Get Direction**.



5. RMB click in the node graph to bring up the search window > Type **Normalize**.
    a. This will bring up a few different Normalize nodes
    b. The one we're interested in, is under the Vector tab, has a green function (f) symbol, and is simply called Normalize.

   c. The Normalize node takes a vector value and normalises it to a value of one. This makes it easier to set directions.
6. Plug the Direction variable's output pin, into the Normalize node's A input
   a. We'll leave the Tolerance value at default
7. RMB click in the node graph to bring up the search window > Type **'Multiply'**
   a. This will bring up a few different Multiply nodes
   b. The one we're interested in, is under the Operators tab, called 'Multiply'
8. Plug the **Normalize** node's Return Value output pin into the top input slot of the multiply node.
   a. This will automatically change the bottom input slot into a vector, which we don't want as our **Speed** variable is a Float. What we want to do, is multiply the direction by the speed, so we can control how fast the target moves.
   b. To fix this > RMB click on the **Multiply** node's bottom input slot > Under the Pin Conversions section > Select **To Float (SinglePrecision).**
9. From the My Blueprint panel > Drag and drop the **Speed** variable onto the **Multiply** node's bottom input slot.
   a. This will connect it the same way as if we dragged it onto the Graph Editor and selected the **Get Speed** option and connected it manually.
10. From here, plug the **Multiply** node's output pin into the **Delta Location** input on the **Add Actor World Offset** node.
11. We should keep things nice and tidy, so let's select everything and press the 'c' key.
   a. This will wrap all the nodes into a comment, which we can add notes to, move around easily, and just easily group nodes that do specific functionality.
   b. You can style comments through the Details Panel.
12. Compile and Save the graph > See graph image below for setup
13. Select the Level tab to return to the main window
14. From the Content Browser > Drag and drop the **BP_Target** into the scene
15. Press the Play button > Your target should move to the right on the Y axis.



Target Change Direction Behaviour

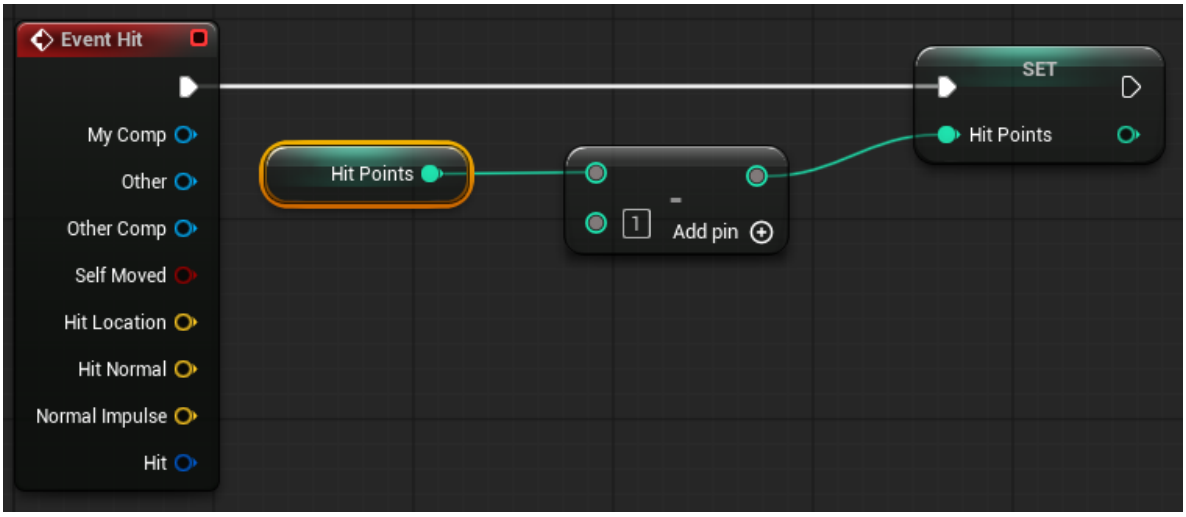
Our target is moving, which is great. But now we want it to reverse direction based on a timer.

1. Navigate back to the Event Graph for the BP_Target.
2. From the Event BeginPlay node > Drag a connection wire off the execution pin then let go in empty space > Search and add a **Set Timer by Event** node.
    a. This node will call a custom event for us after a specified amount of time.
3. Drag and drop the **MovementDuration** variable onto the Time input pin of the **Set Timer by Event** node.
4. Tick the Looping parameter, so that the event will be called repeatedly.
5. RMB on the Graph Editor and search for an '**AddCustom Event**' node.
    a. A CustomEvent node allows us to create an event that we can then add logic to and keep it self-contained.
    b. We use this to break down Blueprint logic into smaller self-contained sections.
    c. This will become important as we go on to better manage our Blueprints.
6. Change the name of the Custom Event node to **Event_FlipDirection**.
7. Connect the red square output at the top of the **Event_FlipDirection** node to the Event input of the **Set Timer by Event** node.
    a. Note you can double click on a connection line to add a manipulation point that you can use to straighten out your connection lines and keep things neat.
8. Drag and drop the **Direction** variable into the node graph > Select **GetDirection**
9. RMB click in the node graph > Search and add another **Multiply** node
10. Plug the **Direction** node's output pin into the top slot of the **Multiply** node
11. RMB click on the bottom slot of the **Multiply** node > Under the Pin Conversions > Select '**To Float (Single Precision)**'
12. Set the value of the float in the **Multiply** node to negative one (-1).
13. Drag another **Direction** variable into the node graph again. This time select **SetDirection**.
14. From the **Event_FlipDirection** node > Drag the output execution pin, and plug it into the Set node's execution pin.
15. Plug the **Multiply** node's output pin into the Set node's Direction input pin.
16. Wrap it all into a comment.
17. **Compile** and **Save**.
18. Return to the Level tab > Press Play. Your target should now move left and right after three seconds (the value for the **MovementDuration** variable).

Target Hit Point Behaviour

Our target is moving back and forth, but when we shoot it, nothing happens. We now need it to destroy after its **HitPoints** variable reaches 0.

1. Navigate back to the Event Graph for the BP_Target.
2. RMB click in the node graph > Search and add the **Event Hit** node.
    a. This node will fire off code when something hits the object.
3. Drag the **HitPoints** variable onto the Graph Editor and choose **Set HitPoints.**
    a. The logic will be that when the Target gets hit, we set its **HitPoints** to whatever **HitPoints** currently is minus 1**.**
4. Drag another **HitPoints** variable onto the Graph Editor, this time selecting **Get HitPoints**.
5. From the out pin of this node, connect to a **Subtract** node.
6. Replace the 0 in the **Subtract** node with a 1.
7. Connect the out pin of the **Subtract** node to the **HitPoint** in pin on the Set node.
8. This is a very common pattern in reducing a value that you will use over and over.

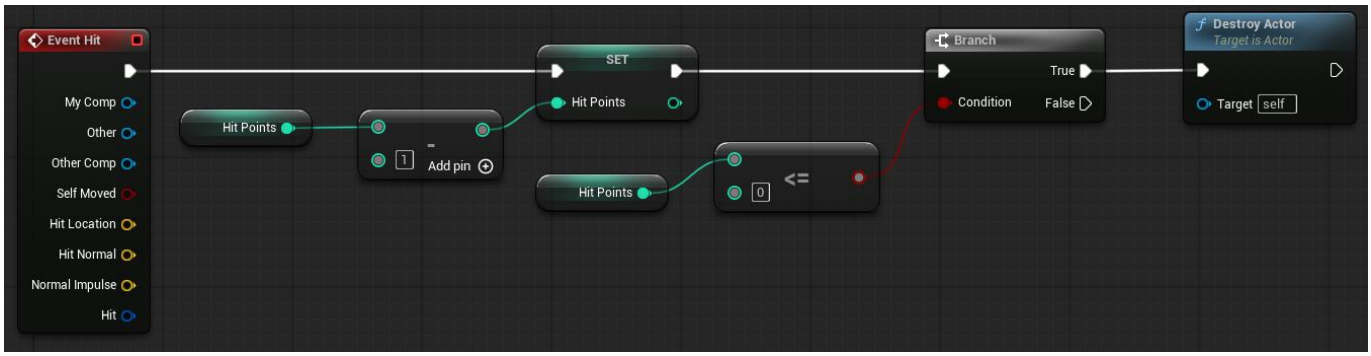


9. Search for a **Branch** node (found under Flow Control).
    a. A **Branch** node is one of the most used nodes that you will encounter.
    b. It is a basic true or false check that relies on a **Condition** that is checked.
    c. The corresponding actions will then execute depending on the result.

   d. In our case, we want to check the Targets **HitPoints** when it is hit with the projectile.
      i. if the Targets **HitPoints** are 0, then **Destroy** the Target.
      ii. If the Targets **HitPoints** are above 0, then don't **Destroy** the Target (but perhaps to something else).
10. Drag another **HitPoints** variable onto the Graph Editor and choose **Get HitPoints**.
11. From this out pin, connect a Less Equal (<=) node.
   a. The Less Equal (<=) node checks if a value is less than or equal to another value.
   b. We want to check if the HitPoints value is less than or equal to 0.
   c. There is another node called Equal (==) that checks if a value is Equal to another.
   d. Logically you would think that we would want to check if the HitPoints are Equal to 0 then Destory the Target.
   e. However, there may be times where our HitPoints go below 0. Using a Less Equal (<=) is more a failsafe way to do this check.
      i. In our specific case, we are always taking a value of 1 off the HitPoints so the Equal (==) would work, however later on we may add a weapon that could take 2 off the HitPoints. In this case, if we were at 1 HitPoint and took 2 off, we would skip 0 and go to –1, bypassing the Destroy Target logic we have.
12. Connect the out pin of the Less Equal (<=) node to the Condition of the Branch node.

Now we just need to decide what to do in each of these conditions. First we will just get the Target to get destroyed when the Condition is true (HitPoints <= 0).

13. From the True out of the Branch node, connect a Destroy Actor node.
   a. The Destroy Actor node does exactly what it sounds like – Destroy an Actor.
14. Compile, Save and Play.

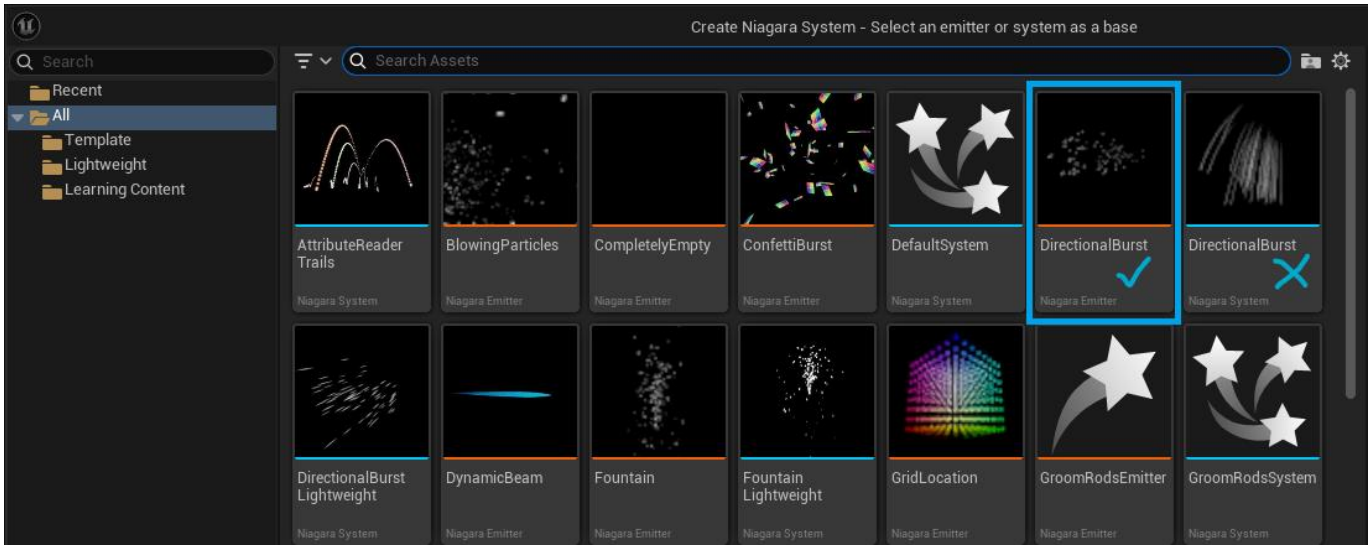


Improving the Target Hit Functionality

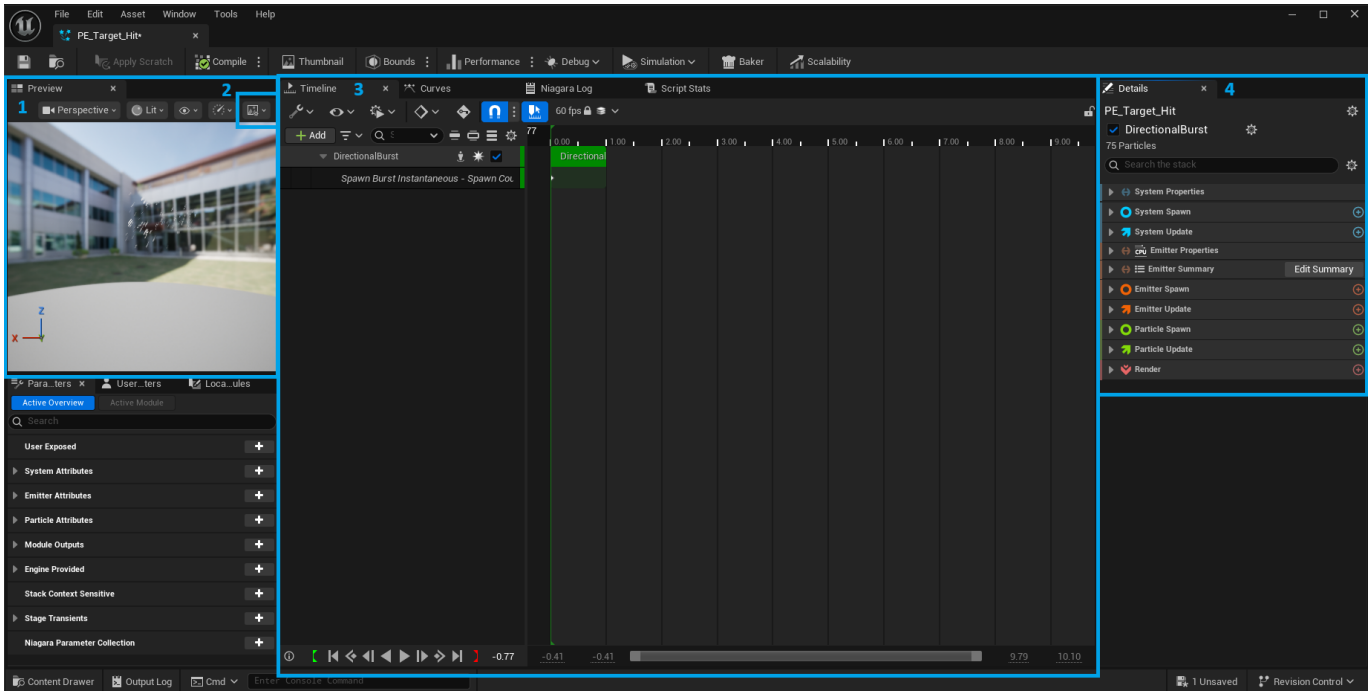
Our target should be destroyed now when its HitPoints reach 0. But it is a little lacking and doesn't give the player much feedback. Also when the Target is destroyed, it just immediately disappears which looks a little janky.

Often when we Destroy an Actor, we try to mask this. Playing a particle effect is a common way to do this.

1. Back in the Content Browser, create a new folder under the **MyContent** folder called **FX**.
2. RMB in the **FX** folder and choose **Niagara System**.
    a. Niagara is the name of Unreal's particle system.
3. From the popup, there are many starter templates categorised into Niagara System and Niagara Emitter.
    a. A Niagara Emitter is the thing that emits particles.
    b. A Niagara System is a collection or multiple Niagara Emitters.
4. Select the **DirectionalBurst** Niagara Emitter (not the DirectionalBurst Niagara System).



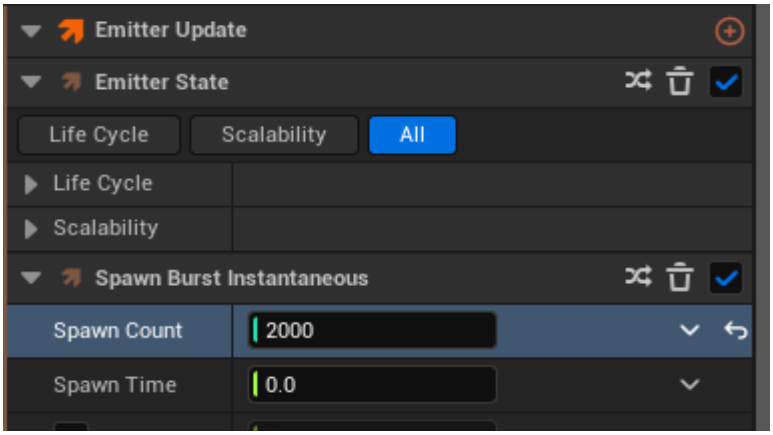
5. Name this PE_Target_Hit
    a. Naming Convention: PE_ = Particle Emitter
6. Double click on the PE_Target_Hit to open it.



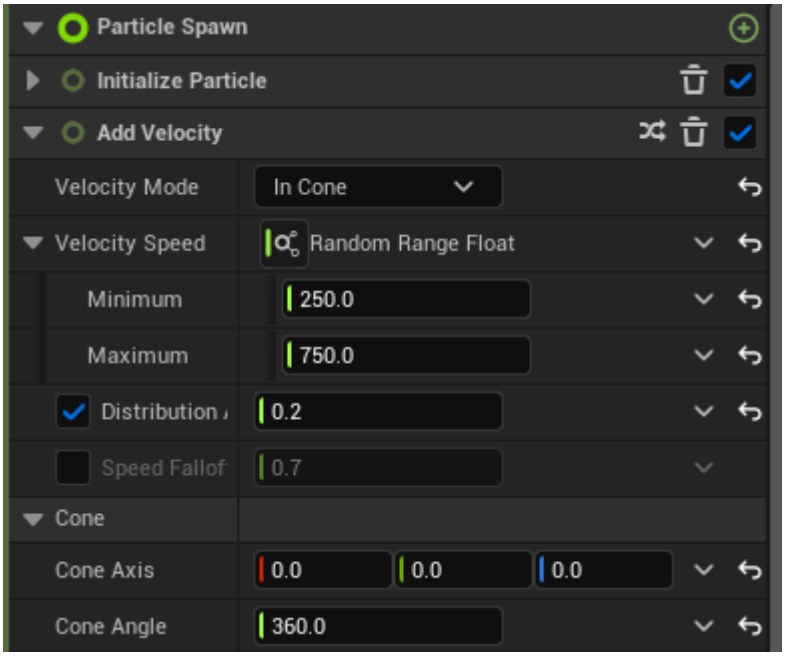
7. There are countless options to change in a particle emitter to achieve whatever result you are after. It can be quite overwhelming. We are going to keep things very basic for now.

1 = Preview window.
2 = Allows you to change the preview window background.
3 = Particle timeline where you can play and tweak the playback of the particles.
4 = The Details panel, where we will be doing most of our changes. In the image, all the sections are folded up. We will use just a select few.

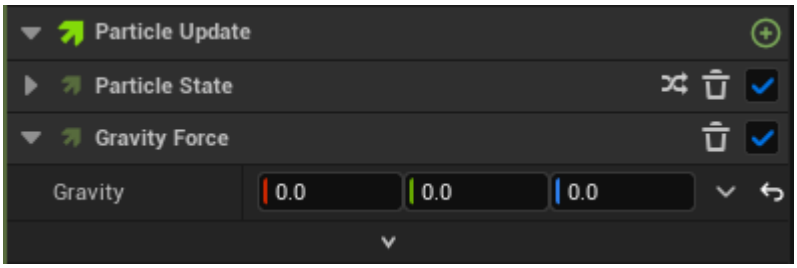
8. In the Details Panel, go to Emitter Update > Emitter State > Spawn Burst Instantaneous
    a. Change the Spawn Count to 2000.



9. go to > Particle Spawn > Add Velocity > Cone.
    a. Change the Cone Axis to 0, 0, 0.
    b. Change the Cone Angle to 360.



10. In the Details Panel, go to Particle Update > Gravity Force.
    a. Change the Gravity to 0, 0, 0.

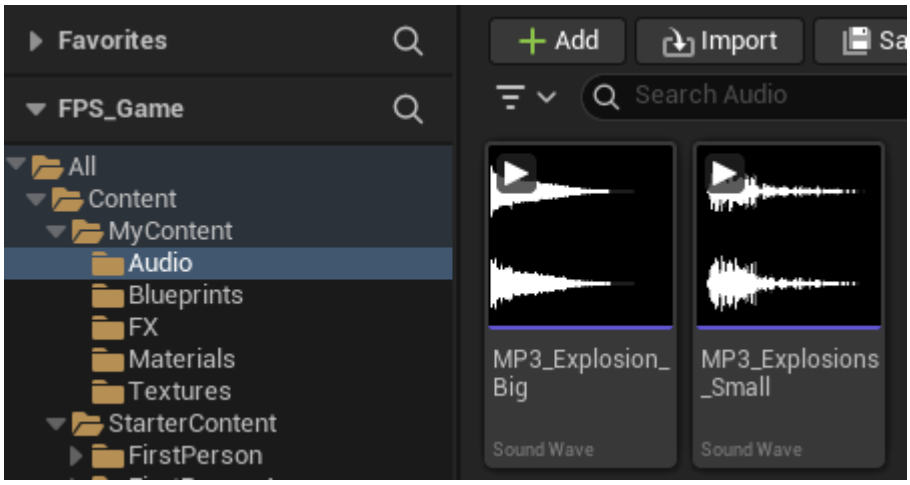


11. That should look like a fireworks explosion and give us a good enough start point for now.
    a. Feel free to play around with other settings in here once everything is complete.
12. Compile and Save

Connecting it all up

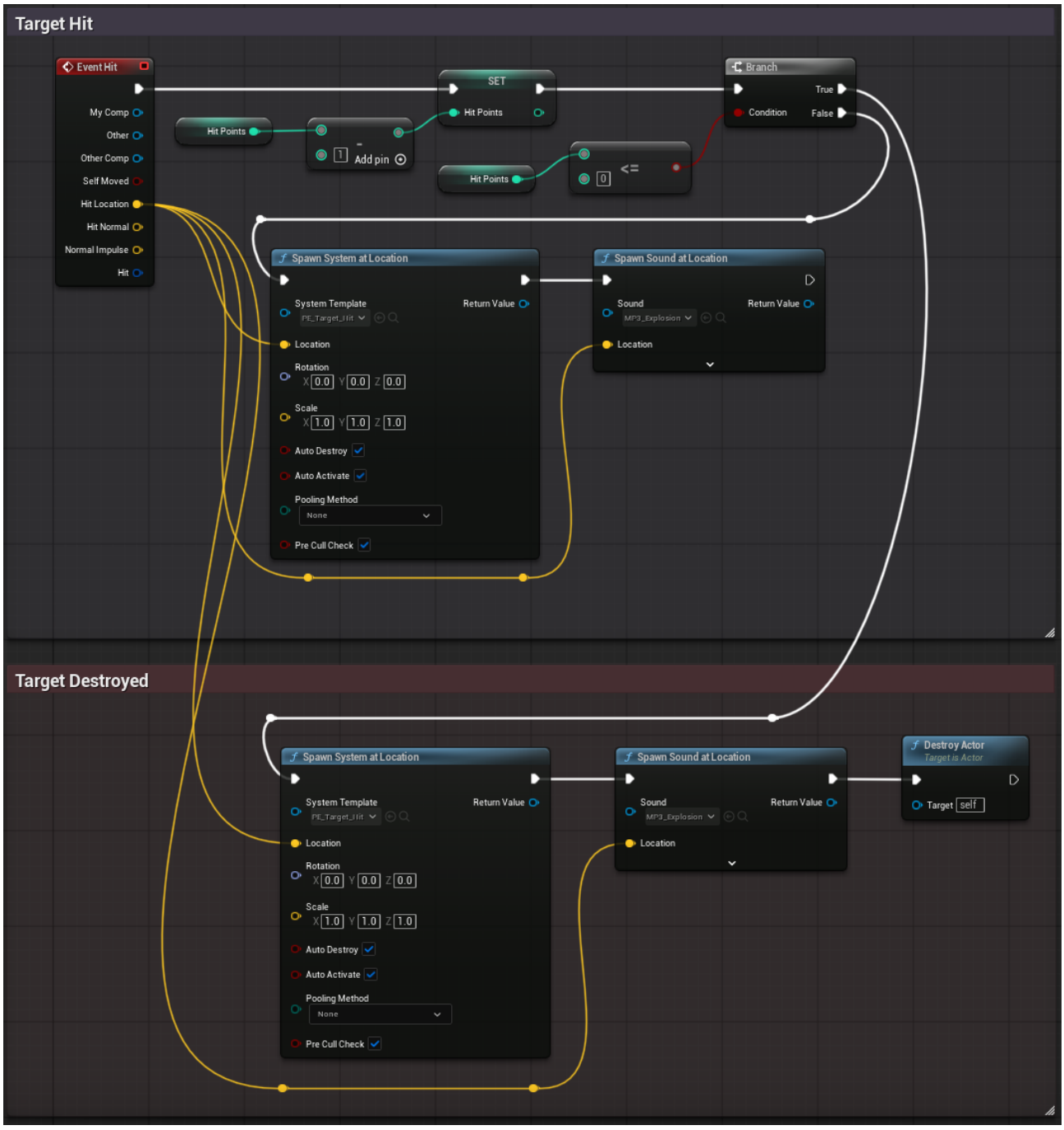
Now we have a particle system to play when the Actor is hit, the last thing we should add is a sound effect that plays. There should be some sound Effects to download on Sharepoint called MP3_Explosion_Big and MP3_Explosion_Small (or get your own).

1. Back in the Content Browser, create a new folder under the **MyContent** folder called **Audio**.
2. Drag the downloaded audio files into your **Audio** folder.



3. Open up our **BP_Target** Blueprint.
4. Search for a **Spawn System at Location** node
    a. This will allow us to spawn a particle system at a specific location.
5. Under the System Template dropdown, select the **PE_Target_Hit.**
6. Connect the False from the **Branch** node to the input of the **Spawn System at Location**.
7. From the out of the **Spawn System at Location** node, connect a **Spawn Sound at Location** node.
8. Under the Sound dropdown, select the **MP3_Explosions_Small** sound.
9. Go to the **Event Hit** node and connect the **Hit Location** to the **Location** of both the **Spawn System at Location** and **Spawn Sound at Location**.
10. Copy and paste the **Spawn System at Location** and **Spawn Sound at Location** nodes.
11. Break the connection between the **Branch True** and the **Destroy** nodes by holding ALT + LMB click on the connecting line.
12. Connect the **Branch True** to the pasted **Spawn System at Location**.
13. Change the **Sound** in the pasted **Spawn Sound at Location** to **MP3_Explosion_Big**.

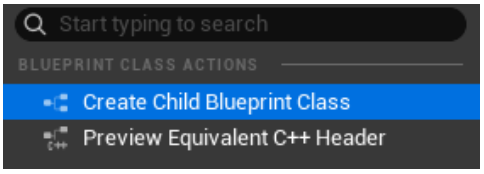
14. Connect the output on the pasted **Spawn Sound at Location** to the **Destroy** node.
15. Connect the Hit Location from the Event Hit node to the **Location** of both pasted **Spawn System at Location** and **Spawn Sound at Location**.
16. Organise your nodes into comments and avoid having connecting lines overlap each other.
17. Compile and Save.



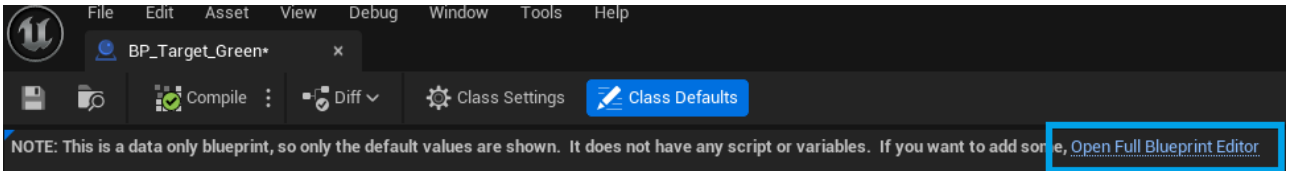
Adding the Targets to our Game

We will want to have a variety of different targets in our scene (different colours, movement patterns, hit points) so much like our Target Material we will create a unique copy of our Target Blueprint which is known as a Child Blueprint Class. This concept of Parent/Child is prevalent in all coding/development discipline. Changes made to the parent apply to all the children.

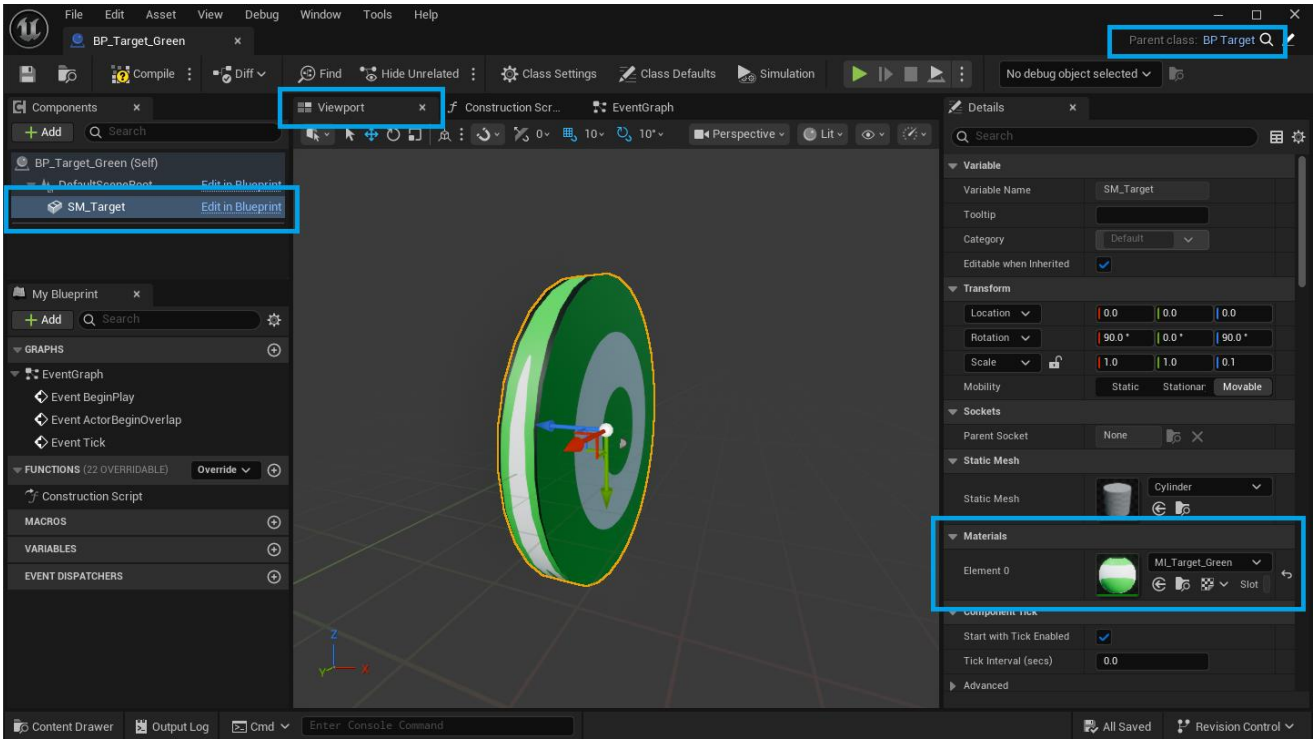
1. In the Blueprints folder of the **MyContent** folder, RMB click on **BP_Target** and choose **Create Child Blueprint Class**.



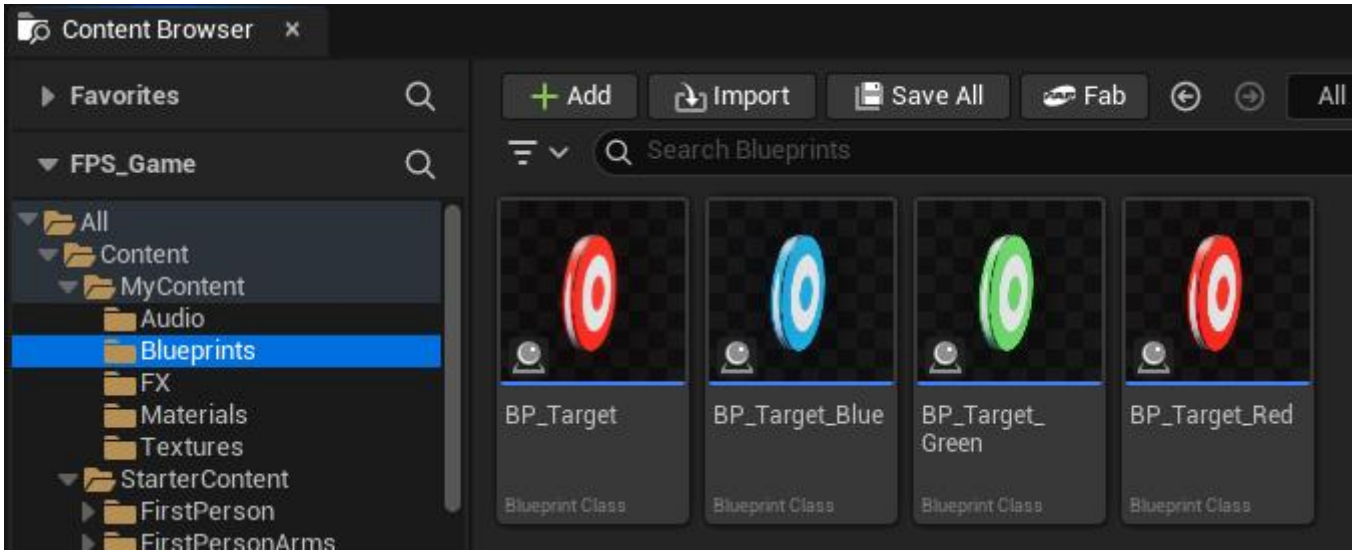
2. Let's rename this to **BP_Target_Green** (or whatever colour you want to work with).
3. When you open the child Blueprint, you may only see a list of values and this message at the top. If so, click on **Open Full Blueprint Editor**.



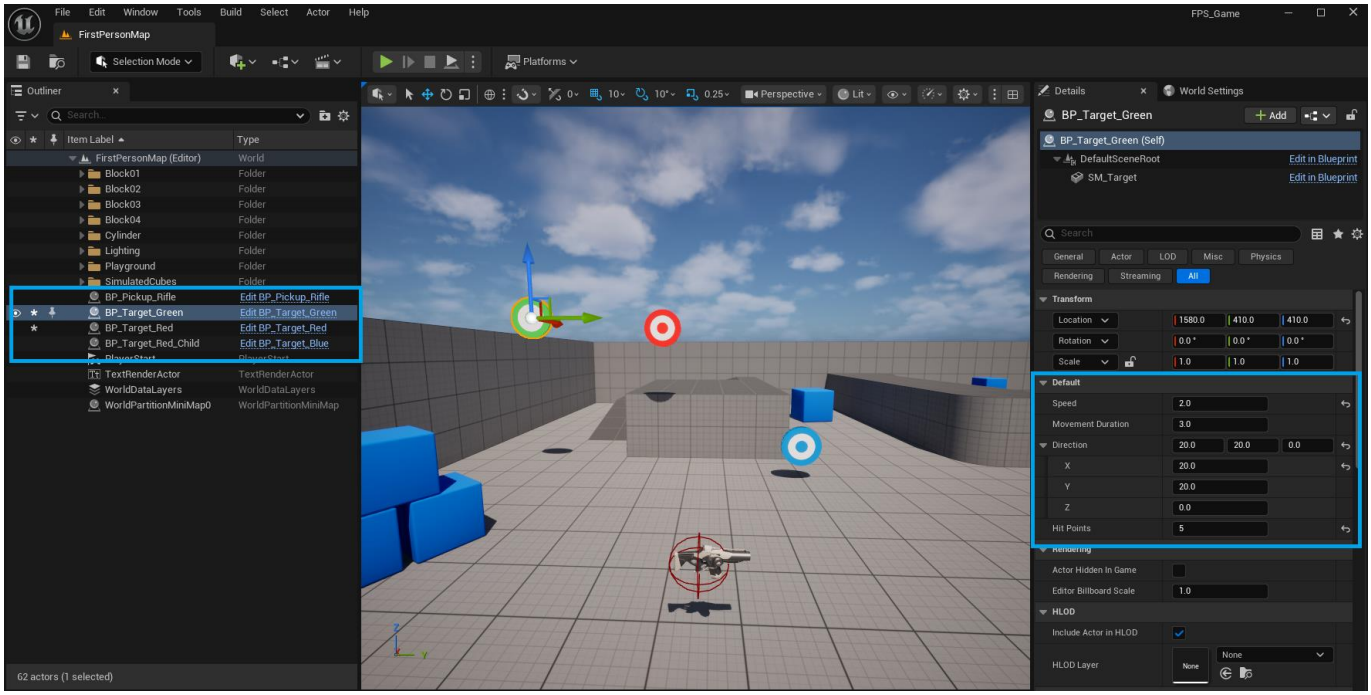
4. This will likely open the Event Graph view. If so, click on the Viewport tab near the top and you will see the target actor.
5. Click on the **SM_Target** Component, and you can change the Material to the appropriate Material here.
6. Note in the top right corner, it says that this Blueprints parent class is **BP_Target**.



7. Compile and Save.
8. Repeat for the other Target variations you want (Blue, Red, etc).



9. Now we can just drag the new child Targets into the scene from the Content Browser.
    a. We won't drag in the parent as we just use that as the reference for the others.
    b. You should see these in the Outliner.
10. If you click on a Target now, you can change its variables **Speed, HitPoints, Directions** and **MovementDuration**.
11. Play around with giving different Targets different behaviours.



Homework

1. **Complete:** the in-class tasks.
2. **Level Design**: Build out your level to accommodate this action shooter style of gameplay.
3. **Extra Mechanic**: Add some of your own functionality / Expand upon the functionality we covered. Some examples:
    a. Different colour particle explosions depending on the target hit.
    b. Targets can move to more than two locations.
    c. A timer where you must destroy all the targets before it runs out.
4. **Documentation**: Document your process and justify your choices. How do they add to the game? Remember to screenshot both new blueprints and level.