Courtesy of AltSpace

UNREAL ENGINE

UNREAL FAST TRACK

WORKSHOP FIVE

Workshop Five: Finishing the Twin-Stick Shooter and Your Own Project

Learning Points

Test Drive (2 hours)

- Adding damage, health, and score system to the "Twin-Stick Shooter with Blueprints" project
- Spawning AI enemies
- Blending animations and creating animation State Machines
- Packaging a game

Grand Prix (1 hour)

- Building an enemy spawner

Off-Roading and Discussion (2 hours)

- Crafting a spellcasting system on your own

Test Drive

For the last workshop in the Unreal Fast Track, we will finish the "Twin-Stick Shooter with Blueprints" Unreal Online Learning course. Before going back to the videos, open up the course and reacquaint yourself with the project.

For some of the videos, we've included occasional mentions of tips and changes since the course came out here in the companion document. If needed, the tips and changes are time-stamped so you know when they are relevant. If there is no timestamp, you can wait until the end to read them.

13: Weapon Firing Behavior

7:00: This node has been renamed to **"Clear Timer by Function Name".**

10:12: This node has been renamed to **"AttachToComponent".**

10:37: The settings on this node have been reframed. To match the settings on the new node, set its properties to the following:

Location Type: "Snap to Target"
Rotation Type: "Snap to Target"
Scale Type: "Keep World"

17: Spawning the Enemies

5:36: This node has been renamed to **"Get (a copy)".**

8:40: This property has been renamed to **"Instance Editable".**

20: Hero Animation Blend Space

2:43: Before Unreal Engine version 4.14, the combined animation editor was known as Persona. For 4.14 it was broken out into four separate editors. While their functionality remains basically the same, some of the interface has changed, and now you can have multiple animation editor windows open in different states. For example, you can now have a tab open for each animation sequence in your game, instead of having a single tab that switches between them (this behavior still remains if you select an animation sequence in the **Asset Browser**). The four separate editors are listed below.

- The **Skeleton Editor** is for examining and modifying the Skeleton of a Skeletal Mesh. This is where you'd add sockets to joints or test how your Skeletal Mesh's joints are rotating.
- The **Skeletal Mesh Editor** is where you can import LODs or assign default Materials to your Skeletal Mesh. It's also where you'll find ways to attach APEX clothing, modify the Skeletal Mesh's bounds, and even assign a specialized shadow Physics asset.
- The **Animation Editor** is where you can work with animation sequences and other animation assets, such as blend spaces and animation montages. Here you will create anim notifies, work with Pose assets, adjust compression and timing of animation sequences, and build your animation montages.
- The **Animation Blueprint Editor** is where you will create the rules for when and how your animations are played. Here you'll be able to use complex State Machines and different blends to bring your characters to life.

6:25: The **Parameter** settings have moved. They are now in the **Asset Details** panel on the left side of the screen, under the **Axis Settings** section. "**X-axis**" is now listed as "**Horizontal Axis**", and "**Y-axis**" is now listed as "**Vertical Axis**".

7:28: You no longer need to click "**Apply Parameter Changes**". Just click "**Save**".

8:52: Instead of moving the mouse, you now click and drag the green diamond to where you want to see the animation blend.

22: Character Death Animations

6:14: The default UI has since changed so that both the **Asset Browser** and the **My Blueprint** panel are both accessible at the same time.

11:06: The property is now just called "**Blend Time**". After unchecking the box, right-click on the node and select "**Refresh Node**" to remove the **Blend Time** pins.

23: Attaching the Gun

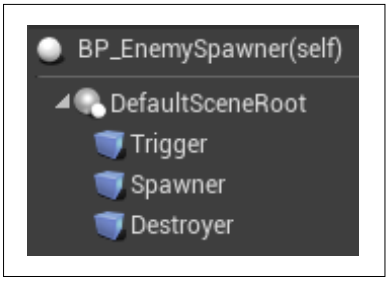
0:57: The **Skeleton** tab you want has moved. It is now a tab labeled "**Skeleton Tree**" and is located next to the Asset Details tab on the left side of the screen.

If your gun isn't attaching correctly to the Gun Socket, make sure you check the spelling of the name of the socket in the mesh and in the Blueprint node, including spaces.

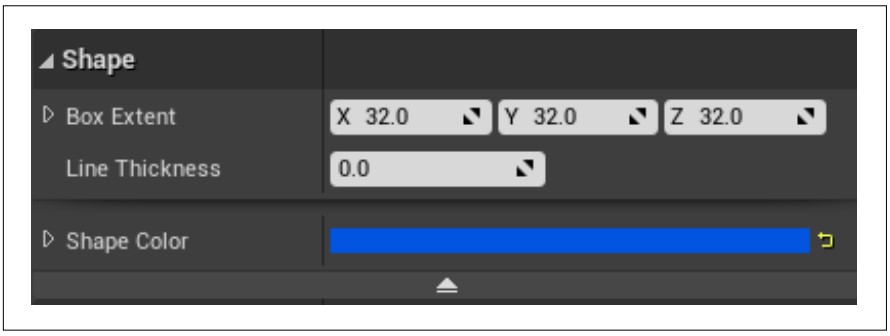
Grand Prix

In the final Grand Prix, you will add enemy spawners to your game. You built one during the "Twin-Stick Shooter with Blueprints" course, but since your players are most likely traversing Levels, we need spawners that can spawn and delete enemies as the player moves so that the game is less resource-intensive.
Let's begin by building the enemy spawner Blueprint itself.

1. In the **Content Browser**, right-click and create a new Blueprint Actor. Name it "**BP_EnemySpawner**" and open it up.
2. Add three **Box Collision** components. Name them "**Trigger**", "**Spawner**", and "**Destroyer**". Make sure none of them is a child of either of the other two.



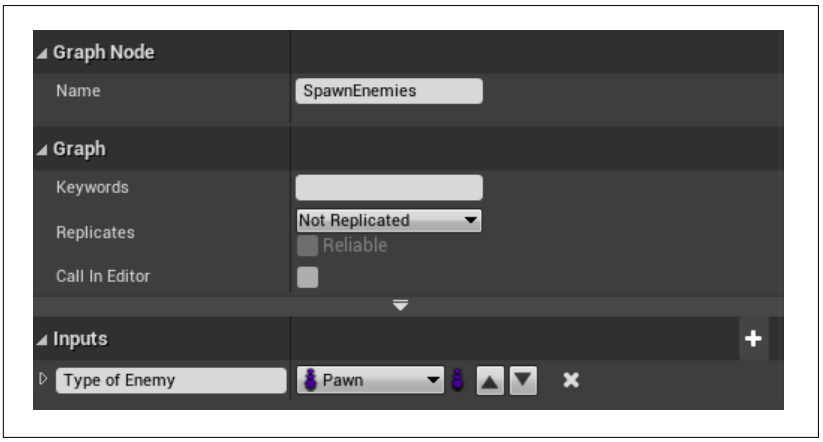
3. Set the **Scale** property of the three components to the following: **Trigger** to "**–180.0, 0.0, 0.0**", **Spawner** to "**0.0, 0.0, 0.0**", and **Destroyer** to "**180.0, 0.0, 0.0**".
4. In the **Details** panel for each of the three **Box Collision** components, find the **Shape** section and click on the triangle to the left to see the drop-down. Set the **Shape Color** property of the components to the following: **Trigger** to "**red**", **Spawner** to "**green**", and **Destroyer** to "**blue**".



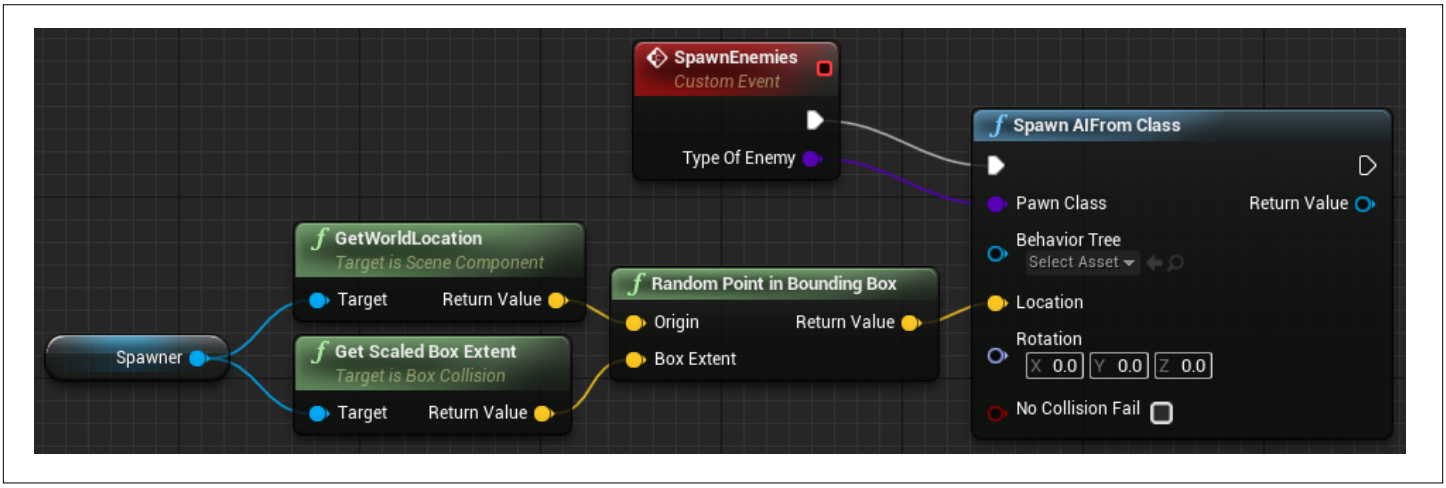
As you can see, the spawner has three parts. The first part is the Trigger, which tells the game to spawn enemies. The second part is the Spawner, which is the space in which the enemies can spawn. The last part is the Destroyer, which destroys all the Actors you spawned with this spawner when you move to a different area.

Move over to the Event Graph now. It is time to create the spawning behavior.

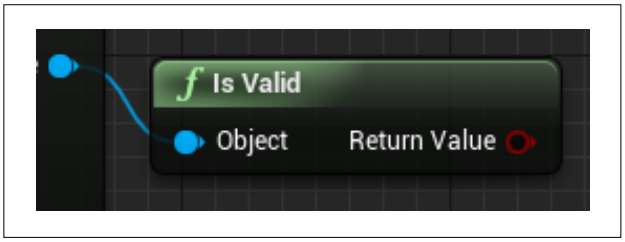
1. Create a custom event called "**SpawnEnemies**". Give it a new input parameter called "**TypeOfEnemy**". Set **Variable Type** to "**Pawn → Class Reference**". Compile the Blueprint.



2. From the event node, add a **Spawn AIFrom Class** node. Connect the event node's **TypeOfEnemy** pin to the **Spawn AIFrom Class** node's **Pawn Class** pin.
3. For the **Spawn AIFrom Class** node's **Location** pin, you are going to do exactly what you did in the "Twin-Stick Shooter with Blueprints" course.
    a. Drag and drop a reference to the **Spawner** component onto the Event Graph.
    b. From the **Spawner** reference, add a **GetWorldLocation** node and a **Get Scaled Box Extent** node.
    c. From the **GetWorldLocation** node's **Return Value** pin, add a **Random Point in Bounding Box** node. Connect the **Get Scaled Box Extent** node's **Return Value** pin to the **Random Point in Bounding Box** node's **Box Extent** pin.
    d. Connect the **Random Point in Bounding Box** node's **Return Value** pin to the **Spawn AIFrom Class** node's **Location** pin.

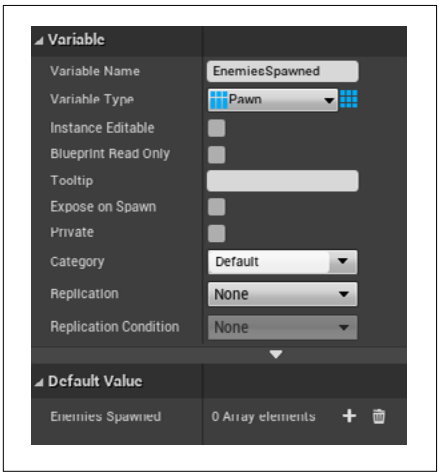


4. From the **Spawn AIFrom Clas**s node's **Return Value** pin, add an Is Valid node. Be careful to pick the correct node, shown here.

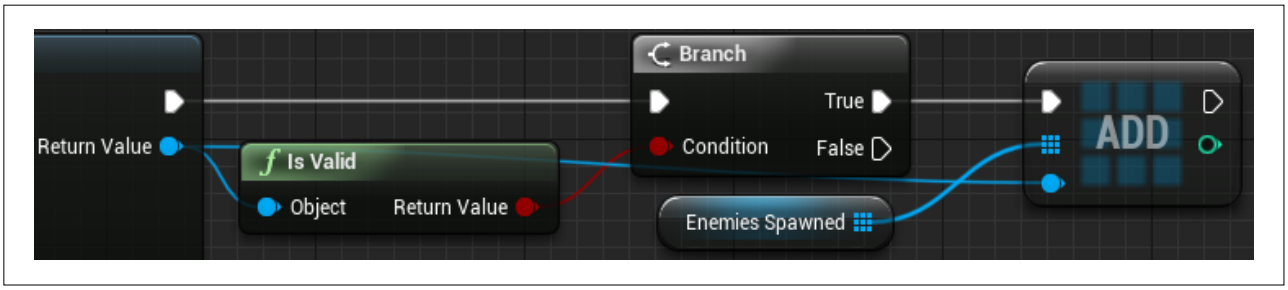


Is Valid is a helpful node that allows the engine to make sure everything is happening as it should. So if something goes wrong, and the **Spawn AIFrom Class** node produces something not right, we can triage before it produces a game-breaking bug.

5. From the **Spawn AIFrom Class** node's output execution pin, add a **Branch** node. Connect the **Is Valid** node's **Return Value** pin to the **Branch** node's **Condition** pin.
6. Create a new variable called "**EnemiesSpawned**". Set **Variable Type** to "**Pawn → Object Reference**". Then, to the right of the **Variable Type** drop-down, click the blue line icon and change the container type from "**Single Variable**" to "**Array**", the icon with the nine little squares. Compile the Blueprint.

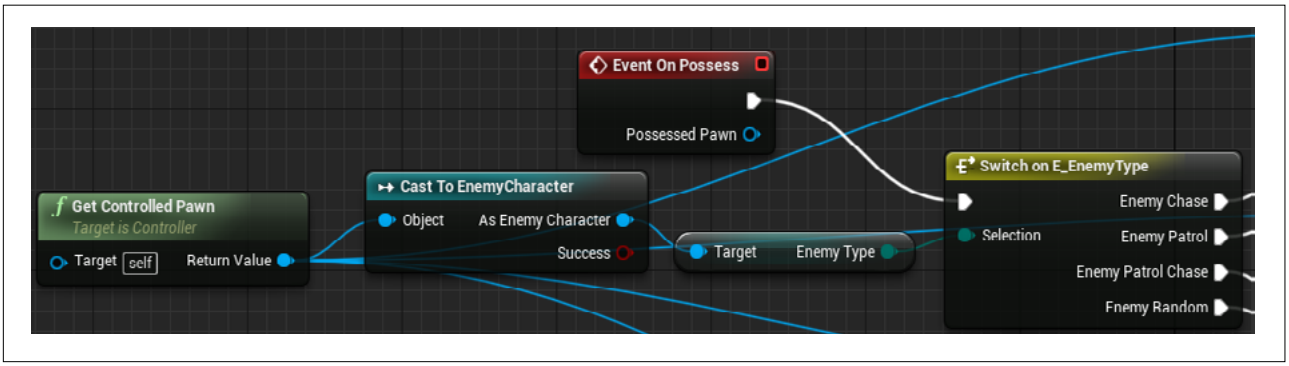


7. Drag and drop a reference to **EnemiesSpawned** onto the Event Graph.
8. From the **EnemiesSpawned** reference, add an **Add** node. Connect the **Add** node's input execution pin to the **Branch** node's **True** pin.
9. Connect the **Spawn AIFrom Class** node's **Return Value** pin to the **Add** node's bottom input pin.
10. Compile the Blueprint.



If you were to set up the enemy spawner in-game and call the **SpawnEnemies** event while playing, you would get an error with the **EnemyAI** Blueprint we built in Workshop Four. The error would occur because **Cast To EnemyCharacter** would fail and the game wouldn't be able to run AI on the spawned enemies.

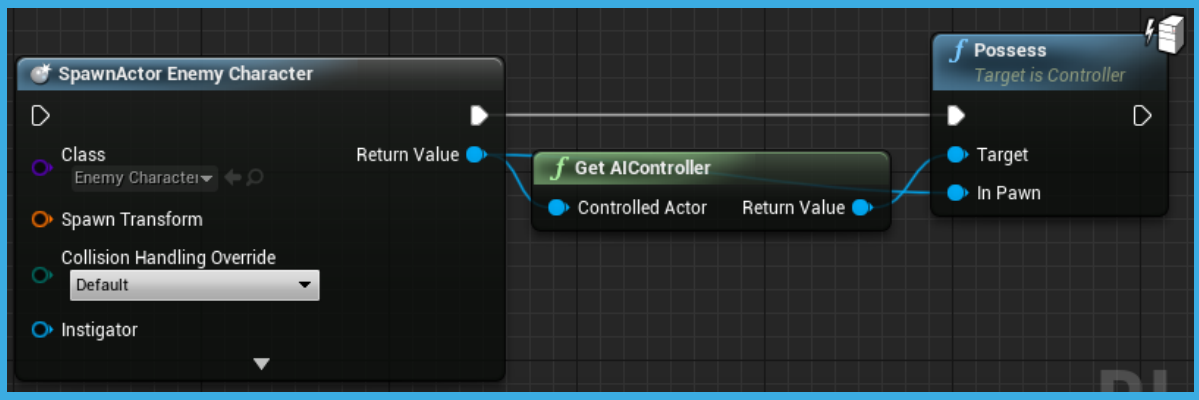
1. Go into the **EnemyAI** Blueprint.
2. Find the **Event BeginPlay** node.
3. Delete the **Event BeginPlay** node and replace it with an **Event OnPossess** node. Don't forget to reattach the execution wires.



Why does the cast fail?
Two important Unreal Engine lessons.

1. **Blueprint nodes sometimes contain more than one function.**

The **Spawn AIFrom Class** node is a great example of this. If you go into the source code of Unreal Engine, you will see that the code for the **Spawn AIFrom Class** node is actually three nodes in one. **Spawn AIFrom Class** is made up of a **SpawnActorFromClass** node, a **Get AIController** node, and a **Possess** node. So when you run **Spawn AIFrom Class**, you are running the following:



If you think of **Spawn AIFrom Class** as three nodes, it makes sense why the cast fails with an **Event BeginPlay** node. When we run **Spawn AIFrom Class**, it spawns the Actor first, which immediately triggers **Event BeginPlay** on the **EnemyAI** Controller. The first thing the **EnemyAI Controller** runs on **Event BeginPlay** is the **Get Controlled Pawn** node, but since the game has not run the **Possess** node yet, the game gets a null reference for **Get Controlled Pawn**, and thus the **Cast To EnemyCharacter** node fails.

This example shows the other important Unreal Engine lesson that is very helpful to know.

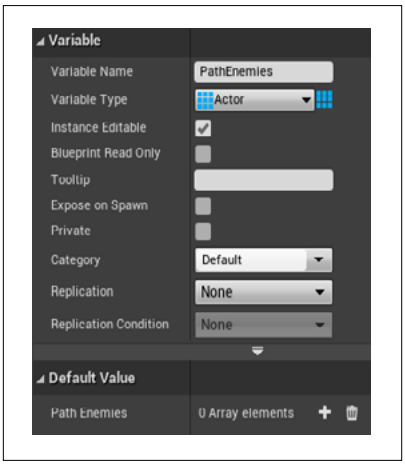
2. **The Controller and the Pawn are separate.**

This is a fundamental thing to understand when using Unreal Engine. While it is possible to handle all input in the Pawn, especially for less complex cases, if you have more complex needs, like multiple players on one game client, or the ability to change characters dynamically at runtime, it might be better to handle input in the Player Controller. In this case, the Player Controller decides what to do and then issues commands to the Pawn (e.g., "start crouching," "jump").

Also, in some cases, putting input handling or other functionality into the Player Controller is necessary. The Player Controller persists throughout the game, while the Pawn can be transient. For example, in deathmatch-style gameplay, you may die and respawn, so you would get a new Pawn but your Player Controller would be the same. In this example, if you kept your score on your Pawn, the score would reset, but if you kept your score on your Player Controller, it would not.

The basic spawning functionality that we have set up in the **EnemySpawner** Blueprint works perfectly for our **EnemyChase** and **EnemyRandom** enemies, but because of how we set up creating paths in the **EnemyCharacter** Blueprint (by editing splines already added to the scene), we cannot spawn **EnemyPatrol** or **EnemyPatrolChase** enemies in the same way. Instead, we will just make them invisible, and when the other enemies spawn, these enemies will become visible.

1. Start by creating a new variable called "**PathEnemies**". Set **Variable Type** to "**Actor → Object Reference**". Set the container type to "**Array**". Compile the Blueprint. Make sure **PathEnemies** is a public variable.



2. Add an **Event BeginPlay** node to the Event Graph.
3. Drag and drop a reference to **PathEnemies** onto the graph. Connect it to a **ForEachLoop** node. Connect the **ForEachLoop** node's Exec pin to the **Event BeginPlay** node.
4. Off the **ForEachLoop** node's **Array Element** pin, add a **Set Actor Hidden In Game** node.
5. Connect the **ForEachLoop** node's **Loop Body** pin to the **Set Actor Hidden In Game** node's input execution pin and check the **New Hidden** box.
6. Compile the Blueprint.

You have now set up the logic for the **EnemyPatrol** and **EnemyPatrolChase** enemies to be invisible at the start of the game.

Next let's focus on finishing the spawning. We have to tell the spawner how many enemies to spawn.

1. Create a custom event called "**BeginSpawn**". Off that node, add a **Sequence** node. This will make it easier to read what is going on.
2. Off the **Sequence** node's **Then 0** pin, add a **ForLoop** node (note: not a **ForEachLoop** node).
3. Create two **Integer** variables. Name one "**EnemyChasetoSpawn**" and the other "**EnemyRandomtoSpawn**". Make sure both are public variables.
4. Drag and drop a reference to **EnemyChasetoSpawn** onto the graph. Off that node, connect an **integer – integer** node. Set the value of the integer – integer node's bottom pin to "**1**" and then connect the node's output pin to the **ForLoop** node's **Last Index** pin.
5. Off the **ForLoop** node's **Loop Body** pin, add a **SpawnEnemies** node. Set the **TypeOfEnemy** property to "**Enemy Chase**".

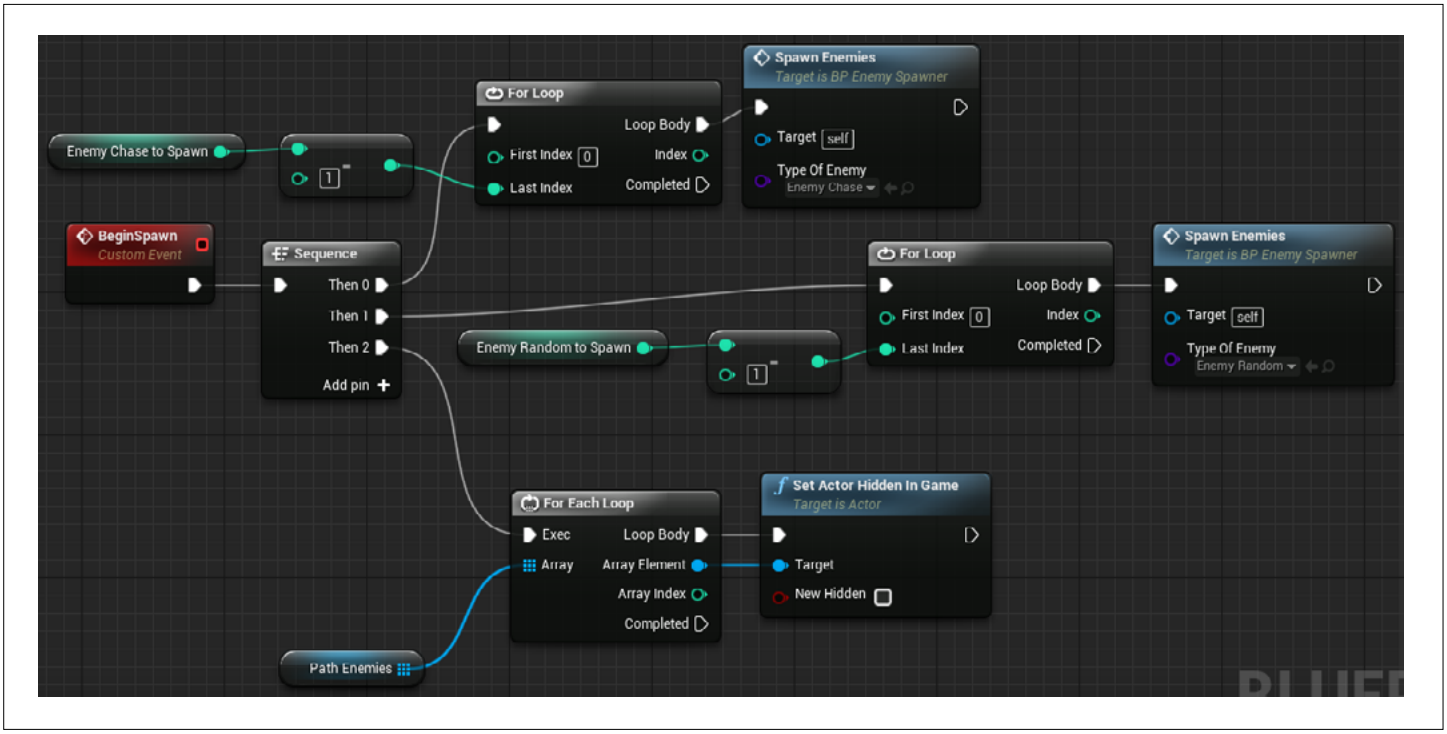
We will use these nodes to tell the game how many enemies we want to spawn. We have to subtract by 1 because we are using an array and arrays start at 0. This next part will look very similar.

1. Off the **Sequence** node's **Then 1** pin, add a **ForLoop** node (note: not a **ForEachLoop** node).
2. Drag and drop a reference to **EnemyRandomtoSpawn** onto the graph. Off that node, connect an **integer – integer** node. Set the value of the integer – integer node's bottom pin to "**1**" and then connect the node's output pin to the **ForLoop** node's **Last Index** pin.

3. Off the **ForLoop** node's **Loop Body** pin, add a **SpawnEnemies** node. Set the **TypeOfEnemy** property to "**Enemy Random**".

Finally, we will do something slightly different.

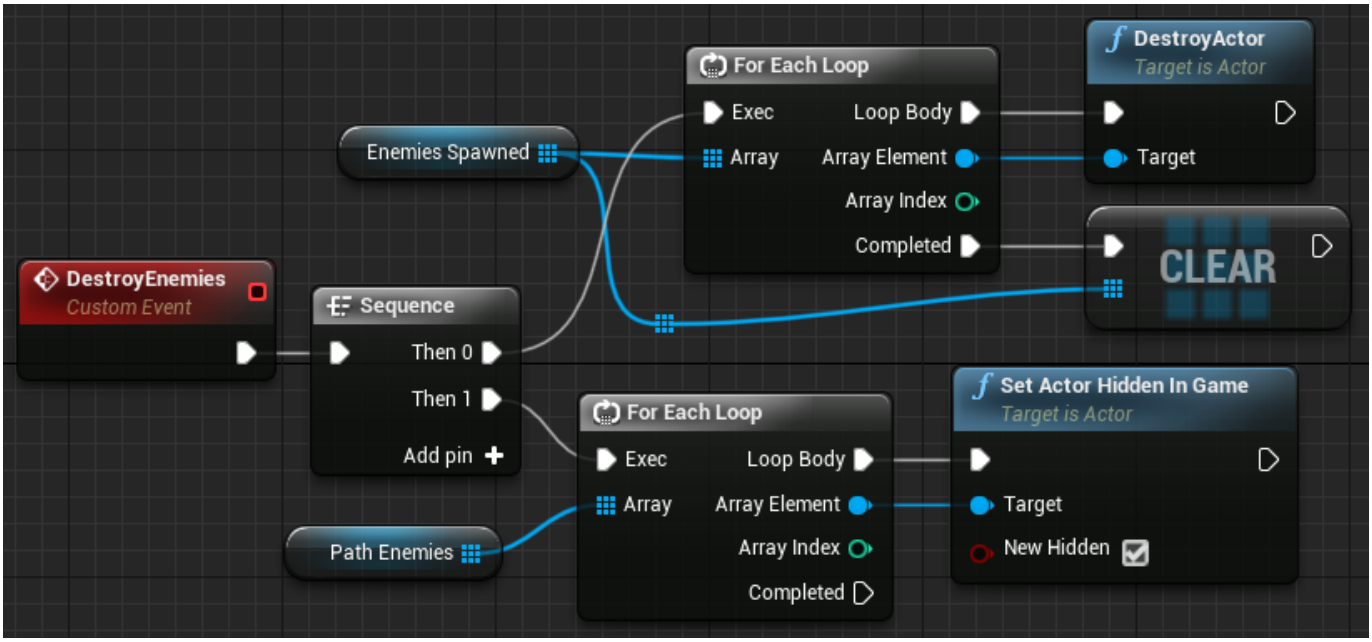
1. Drag and drop a reference to **PathEnemies** onto the graph. Off the reference, add a **ForEachLoop** node.
2. Connect the **Sequence** node's **Then 2** pin to the **ForEachLoop** node's **Exec** pin.
3. Off the **ForEachLoop** node's **Loop Body** pin, add a **Set Actor Hidden In Game** node. Do not check the **New Hidden** box.
4. Connect the **ForEachLoop** node's **Array Element** pin to the **Set Actor Hidden In Game** node's **Target** pin.
5. Compile the Blueprint.



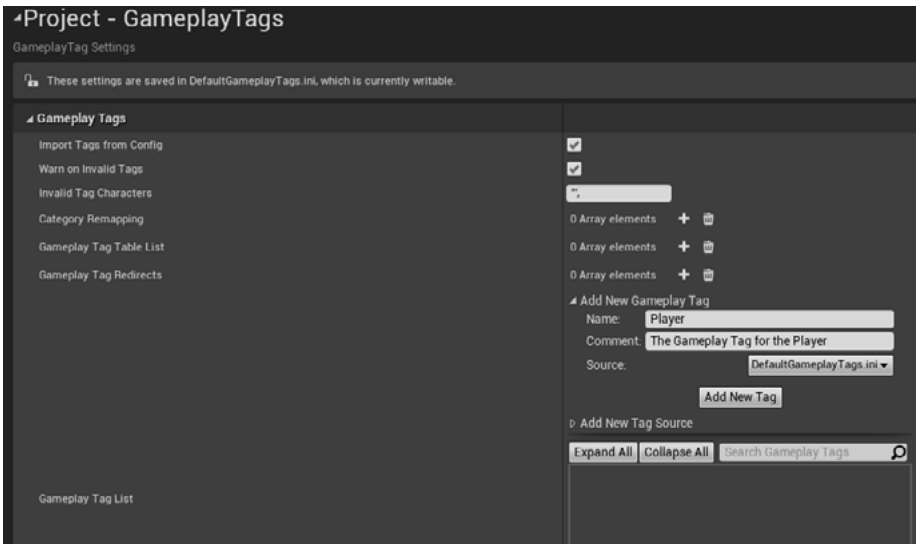
With **EnemyChasetoSpawn** and **EnemyRandomtoSpawn** being public variables, when you add this Blueprint to your Level, you will be able to adjust how many enemies should spawn for each **EnemySpawner** Blueprint.

We should handle destroying enemies when you leave the area so that you can free up resources.

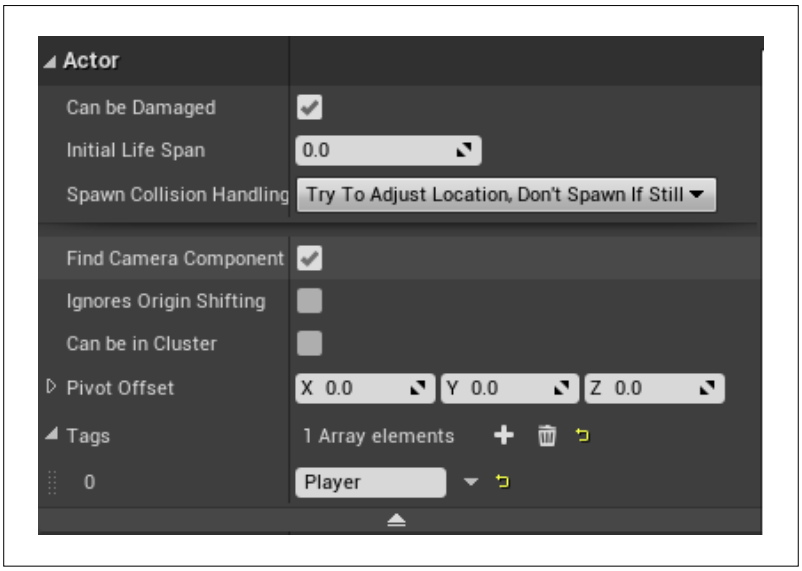
1. Add a custom event called "**DestroyEnemies**". Off that node, add a **Sequence** node for easier reading again.
2. Drag and drop a reference to **EnemiesSpawned** onto the graph. From the reference node, add a **ForEach Loop** node.
3. Connect the **Sequence** node's **Then 0** pin to the **ForEachLoop** node's **Exec** pin.
4. Off the **ForEachLoop** node's **Loop Body** pin, add a **Destroy Actor** node.
5. Connect the **ForEachLoop** node's **Array Element** pin to the **Destroy Actor** node's **Target** pin.
6. From the **EnemiesSpawned** reference you created, add a **Clear** node.
7. Connect the **Clear** node's input execution pin to the **ForEachLoop** node's **Completed** pin.
8. Now drag and drop a reference to **PathEnemies** onto the graph. From the reference, add a **ForEachLoop** node.
9. Connect the **Sequence** node's **Then 1** pin to the **ForEachLoop** node's **Exec** pin.
10. Off the **ForEachLoop** node's **Loop Body** pin, add a **Set Actor Hidden In Game** node and check the **New Hidden** box.
11. Connect the **ForEachLoop** node's **Array Element** pin to the **Set Actor Hidden In Game** node's **Target** pin.
12. Compile the Blueprint.



We have all of the logic ready to spawn our enemies. The last thing we need to do is to add the Overlap events to actually call all of this logic! But before we do that, we need to add a Gameplay Tag.

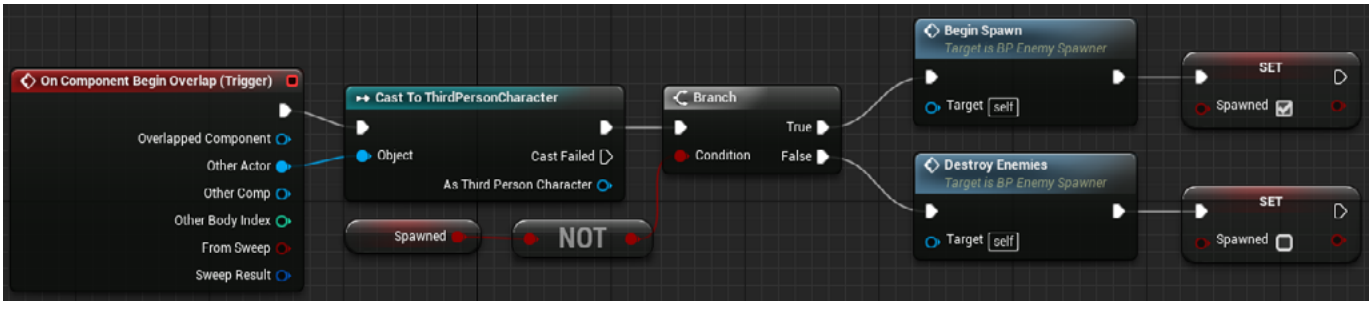


1. Go to the **Project Settings** menu and select "**Gameplay Tags**".
2. Find "**Add New Gameplay Tag**", click on the triangle to the left, and add a Gameplay Tag named "**Player**".
3. Click the **Add New Tag** button.



With that out of the way, back to the Event Graph.

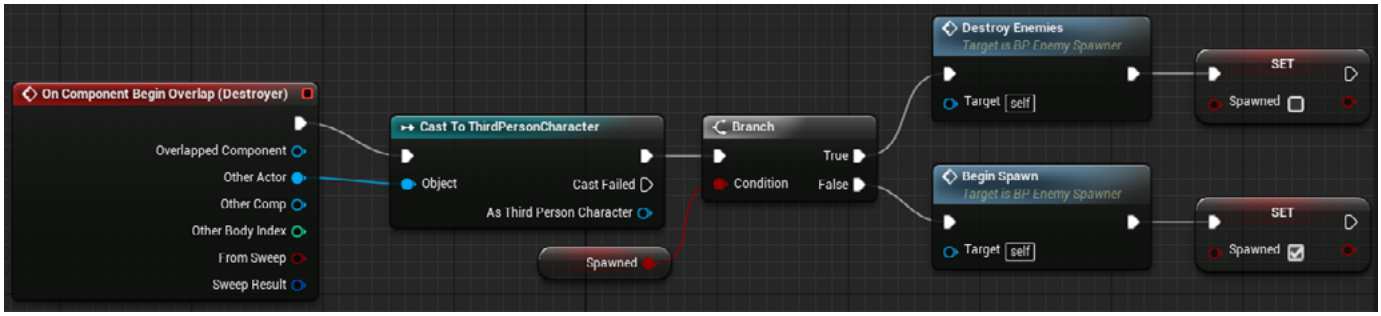
1. Create a **Boolean** variable called "**Spawned**". Compile the Blueprint.
2. Select the **Trigger** component and add an **OnComponentBeginOverlap (Trigger)** event to the graph.
3. From the **Overlap** event's **Other Actor** pin, add an **Actor Has Tag** node and type "**Player**" in the node's **Tag** box.
4. Off the **Overlap** event's output execution pin, add a **Branch** node and connect its **Condition** pin to the **Actor Has Tag** node's **Return Value** pin.
5. From the **Branch** node's **True** pin, add another **Branch** node.
6. Drag and drop a reference to **Spawned** onto the graph. Connect it to a **Not** node, and connect the **Not** node to the second **Branch** node's **Condition** pin.
7. From the **Branch** node's **True** pin, add a **BeginSpawn** node.
8. Off the **BeginSpawn** node, add a **Set Spawned** node and set it to "**true**".
9. From the **Branch** node's **False** pin, add a **DestroyEnemies** node.
10. Off the **DestroyEnemies** node, add a **Set Spawned** node and set it to "**false**".
11. Compile the Blueprint.



Now for the **Destroyer** collision. Be sure not to miss the subtle but important differences between this event and the **Trigger** event.

1. Select the **Destroyer** component and create an **OnComponentBeginOverlap (Destroyer)** event.
2. From the **Overlap** event's **Other Actor** pin, add an **Actor Has Tag** node and type "**Player**" in the node's **Tag** box.
3. Off the **Overlap** event's output execution pin, add a **Branch** node and connect its **Condition** pin to the **Actor Has Tag** node's **Return Value** pin.
4. From the **Branch** node's **True** pin, add another **Branch** node.
5. Drag and drop a reference to **Spawned** onto the graph and connect it to the second **Branch** node's **Condition** pin.

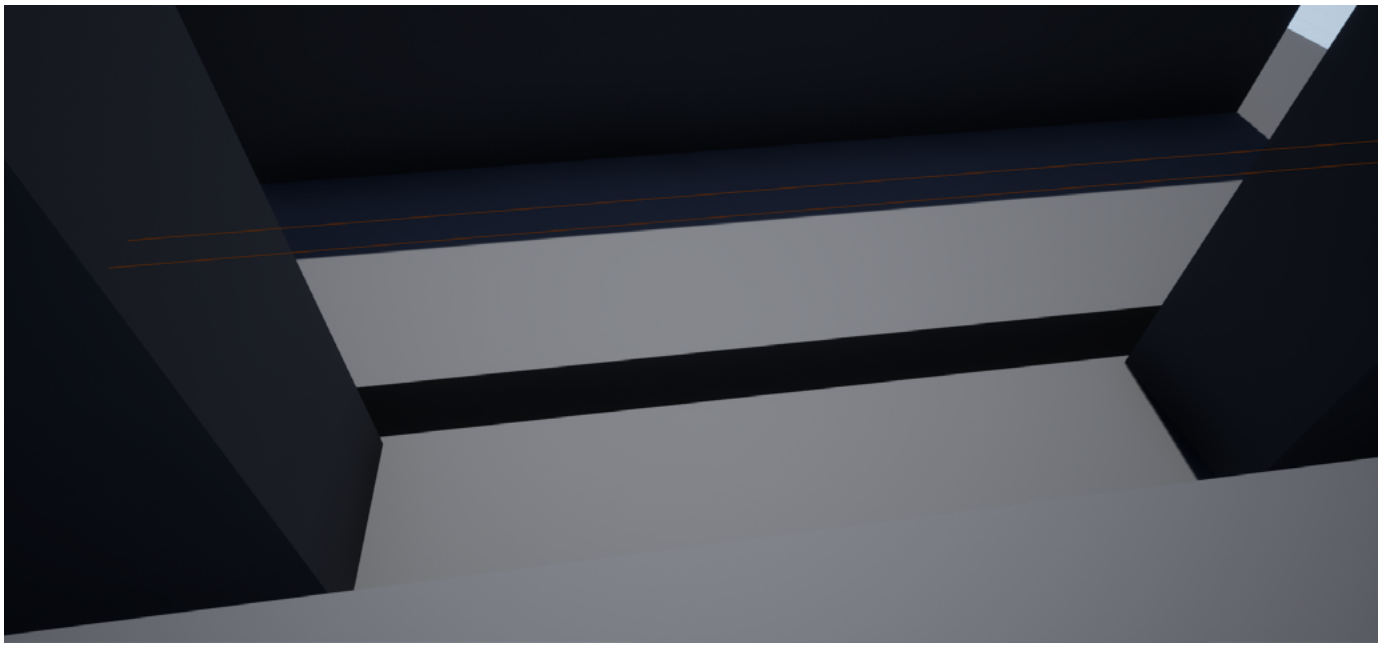
6. From the **Branch** node's **True** pin, add a **DestroyEnemies** node.
7. Off the **DestroyEnemies** node, add a **Set Spawned** node and set it to "**false**".
8. From the Branch node's False pin, add a BeginSpawn node.
9. Off the **BeginSpawn** node, add a **Set Spawned** node and set it to "**true**".
10. Compile the Blueprint.



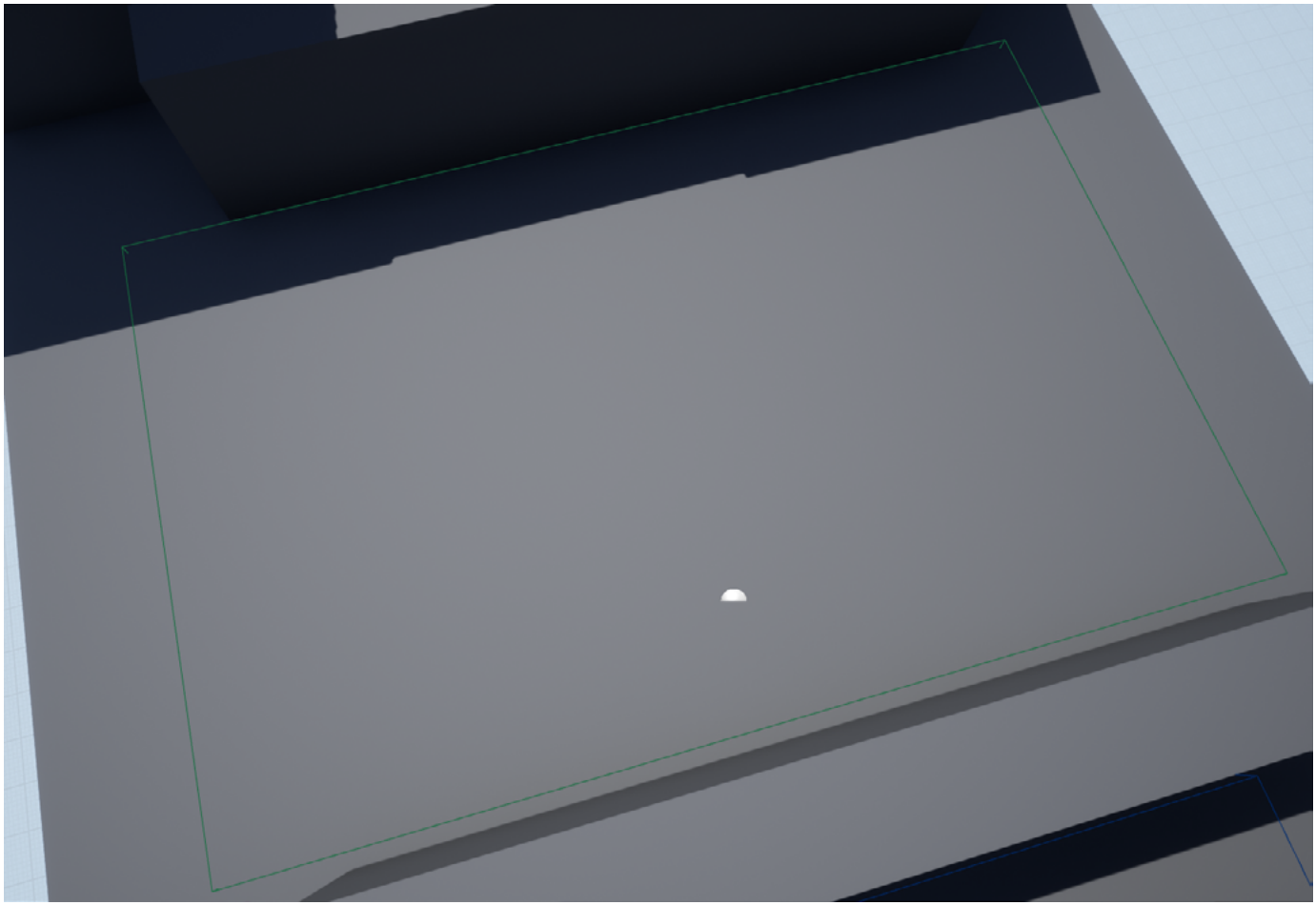
The Overlap events are set up to spawn the enemies into your game and then destroy them when you leave an area, and to do the opposite if you move back to that area.

Now you can add enemy spawners to your game. As many as you need! For each **EnemySpawner** Blueprint you add to your Level, you will have to separately move the Trigger, Spawner, and Destroyer. Luckily, they are different colors so you can differentiate them.

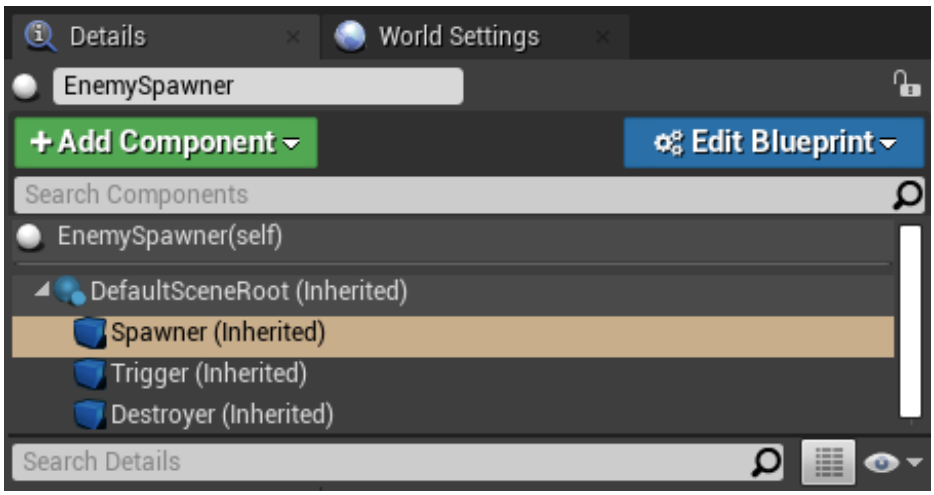
You want the Triggers and Destroyers to be elevated and thin and unavoidable. You don't want players able to sit inside the Box Collision, and you don't want them to accidentally not overlap the Trigger or Destroyer. It also helps if you set up your Level so the player does not see the spawning actually happen, making it seem completely seamless. Here is an example Trigger:



The Spawner, much like the one in the "Twin-Stick Shooter with Blueprints" course, should cover the entire area you want the enemies to spawn in. Here's an example Spawner:



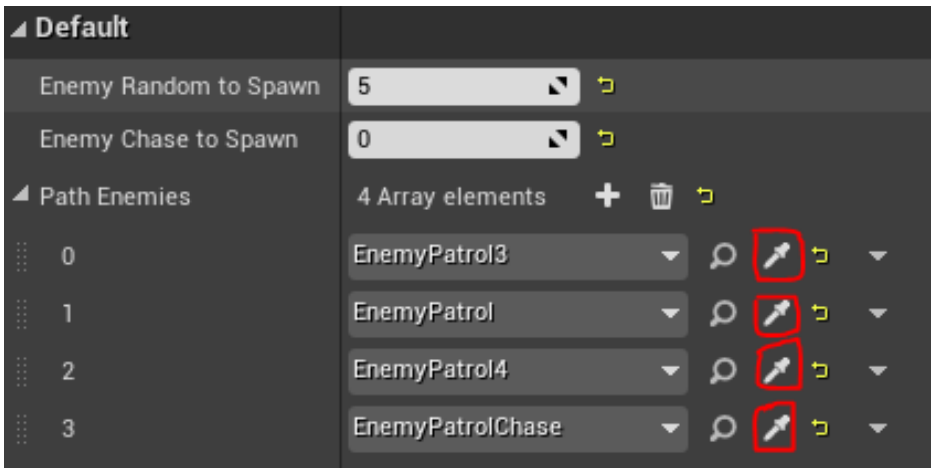
To move and scale the **Trigger**, **Spawner**, or **Destroyer**, click on the individual components in the **Details** panel.



To change the number of **PatrolChase** and **PatrolRandom** enemies that are spawned with each spawner, select the **EnemySpawner** Blueprint you have placed in the world, and under "**Default**" in the Details panel you should see **EnemyChasetoSpawn** and EnemyRandomtoSpawn as changeable variables!

The final step is adding **Patrol** and **PatrolChase** enemies to your enemy spawner so that they disappear and reappear.

1. Select the **EnemySpawner** Blueprint you have placed in the world.
2. In the **Details** panel under the **Default** section, you should see "**Path Enemies**". Use the plus sign to add the number of **Patrol** and **PatrolChase** enemies you want to be a part of this spawner.
3. Once the number of array elements matches the number of enemies you want to add, click the eyedropper icon and then select one of the enemies from the scene. Repeat this until you have selected every enemy you want to be added and they will work with the spawners!



The Potential Issue with Casting
Throughout the Fast Track, we have used a lot of Cast nodes. For small projects, casting is fine, but for larger projects, casting can become an issue with memory and performance. Every time you use a Cast node, you are creating a hard reference to that class and loading the entire class. This can quickly become a problem if you are casting many times in a large project.

An alternative to checking for collision is to use Gameplay Tags, which we did this time in the Grand Prix. An alternative to using casting to access other Blueprints is to use Blueprint Interfaces. Blueprint Interfaces were touched on briefly during the "Twin-Stick Shooter with Blueprints" course and are an excellent topic to learn more about so that you can make sure your projects have great performance.

Off-Roading and Discussion

For the final Off-Roading and Discussion, you will be adding spellcasting to your game. However, instead of following a guide step-by-step, you will be told what spells to add to your game and do it yourself! Using what you have learned and working with your teammates for help, try and figure out how to add the spellcasting and various spells to your game.

For each challenge, after you've completed it on your own, look back at this guide and see how we suggest doing it. If what you made works, great! There is no one right way, and turning an idea into something playable is how you practice your knowledge with Unreal Engine.

To help with the spellcasting, download the Infinity Blade: Effects pack from the Marketplace. This will give you a lot of particle effects to work with. When you add the content to your game, it will include an Overview Level that shows many of the effects you can use.

Once you have completed the challenges and looked through our solutions, go to the end for the final workshop discussion.

Challenge One: Give the player the ability to shoot out a small tornado they jump on for a bounce.
Bonus challenge: Make the tornado pull enemies in.

Challenge Two: Give the player a ground stomp ability that pushes enemies away.
Bonus challenge: Make it so the higher the player jumps, the farther they push enemies away.

Challenge Three: Create a dash that leaves behind a wall that enemies can't walk through.

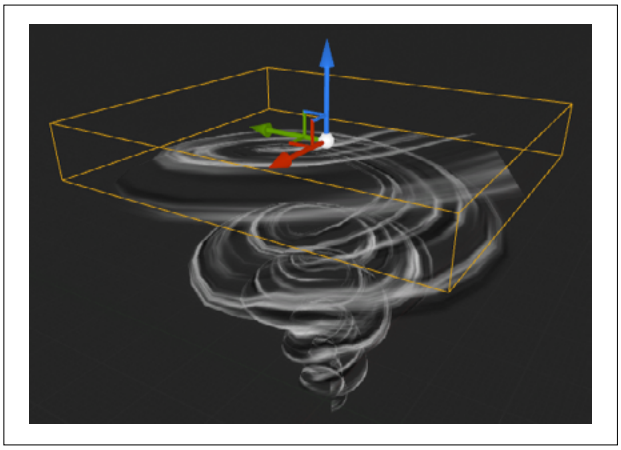
Challenge Four: Create a cooldown system for the spells and make a UI to show what spells are available.

Deathgarden: BLOODHARVEST | Behaviour Interactive

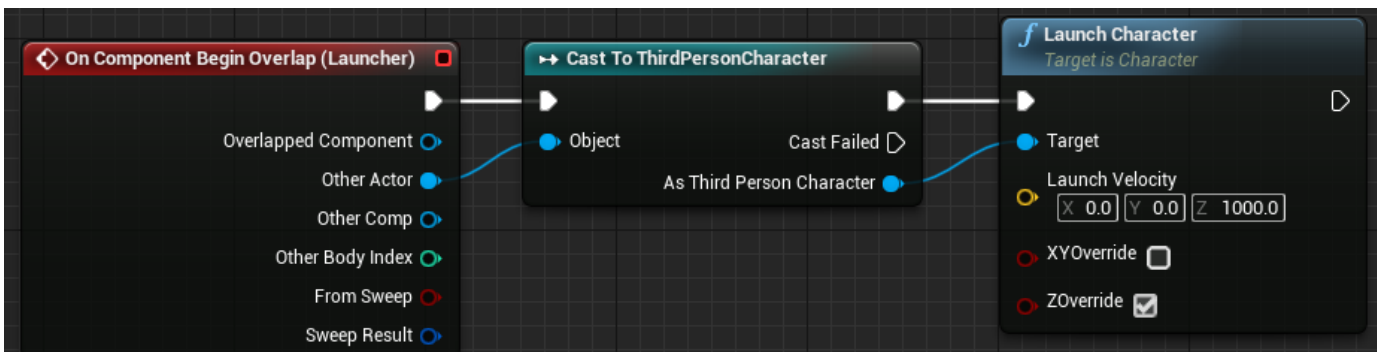
Challenge One:
Give the player the ability to shoot out a small tornado they jump on for a bounce.

This is the easiest spell to create out of all of the challenges. Let's start by creating the tornado asset itself; we want the tornado to be a Blueprint Actor.

1. Right-click in the **Content Browser** and create a new Blueprint Actor. Name it "**BP_Tornado**" and open it up.
2. In the **Components** panel, select "**BP_Tornado (self)**", and in the **Details** panel change the **Initial Life Span** property to "**5.0**".
3. Add a **Static Mesh** component (note: not a **Static Mesh Simulation** component).
4. In the **Details** panel, select the Static Mesh "**SM_TornadoSpiral**". (Note: If the Texture isn't showing up correctly, double-click on the image of the tornado in the Details panel, and under "**Material Slots**" change the **Element 0** setting from "**WorldGridMaterial**" to "**M_Tornado**" and save.)
5. Set the **Static Mesh** component's **Scale** property to "**7.0, 7.0, 7.0**".
6. Drag and drop the **Static Mesh** component onto the root component to make it the root instead.
7. Add a **Box Collision** component. Name it "**Launcher**". Position it at the top of the Static Mesh, covering the whole thing, and make it thin.

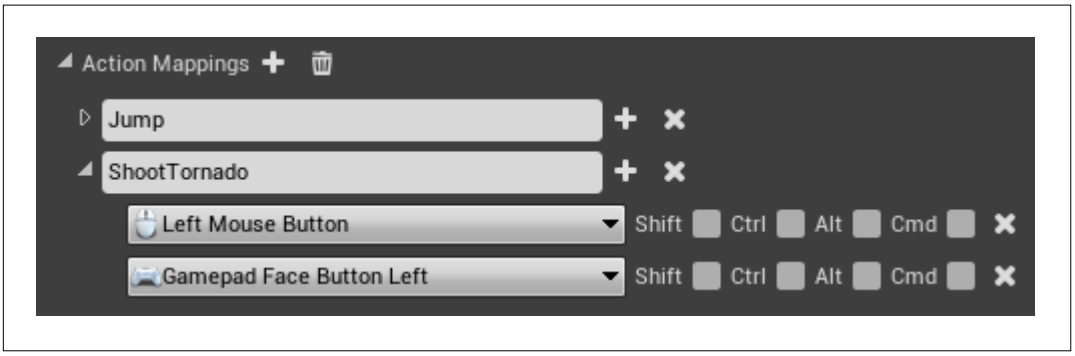


8. Make sure the **Box Collision** component is a child of the **Static Mesh** component.
9. Add a **ProjectileMovement** component. Set **Initial Speed** to "**300.0**", **Projectile Gravity Scale** to "**0.0**", and **Friction** to "**0.0**".
10. Move to the Event Graph. Select the **Launcher** component in the **Components** panel and create an **OnComponentBeginOverlap (Launcher)** event.
11. Off the event node's **Other Actor** pin, add a **Cast To ThirdPersonCharacter** node.
12. From the **Cast** node's **As Third Person Character** pin, add a **Launch Character** node. Set the **Launch Character** node's **Launch Velocity** property to "**0.0**, **0.0**, **1000.0**" and check the **ZOverride box**.
13. Compile the Blueprint.

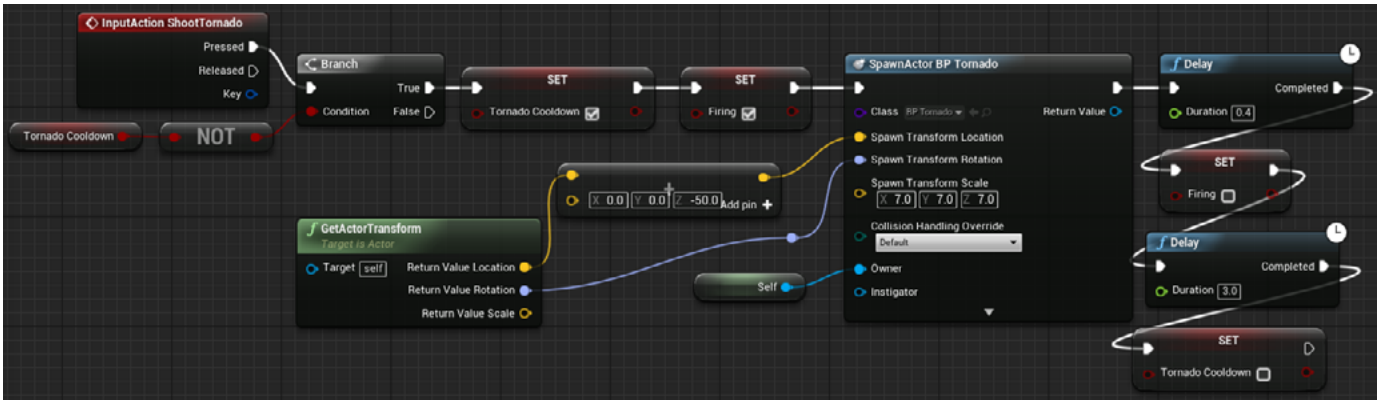


Now that we've built the tornado asset, we can add the ability for our character to shoot out the tornado.

1. Go to the **Project Settings** menu and select "**Input**". Add a new Action Mapping. Name the Action "**ShootTornado**" and select "**Left Mouse Button**" and "**Gamepad Face Button Left**" as the key values.



2. Open the **ThirdPersonCharacter** Blueprint.
3. Add a **ShootTornado** event node to the Event Graph.
4. Create two **Boolean** variables, one called "**Firing**" and the other called "**TornadoCooldown**". The **Firing** variable is for the animation State Machine. Compile the Blueprint.
5. Off the **ShootTornado** event's **Pressed** pin, add a **Branch** node.
6. To set up the condition of the **Branch** node, drag and drop a **Get TornadoCooldown** node onto the graph and connect it to a **Not** node. Connect the **Not** node to the **Condition** pin.
7. From the **Branch** node's **True** pin, add a **Set TornadoCooldown** node and set it to "**true**".
8. Off the **SetTornadoCooldown** node, add a **Set Firing** node and set it to "**true**" too.
9. From the **Set Firing** node, add a **SpawnActorFromClass** node.
10. Set the **SpawnActor** node's **Class** property to "**BP_Tornado**".
11. Right-click on the **SpawnActor** node's **Spawn Transform** pin and select "**Split Struct Pin**". Set the **Spawn Transform Scale** property to "**7.0**, **7.0**, **7.0**".
12. Add a **GetActorTransform** node to the graph. Right-click on its **Return Value** pin and select "**Split Struct Pin**".
13. From the **Return Value Location** pin, add a **vector + vector** node.
14. Set the value of the bottom input pin to "**0.0**, **0.0**, **–50.0**". Connect the **vector + vector** node to the **SpawnActor** node's Spawn Transform Location pin. This is done so that the tornado does not spawn too high in the air.
15. Connect the **GetActorTransform** node's **Return Value Rotation** pin to the **SpawnActor** node's **Spawn Transform Rotation** pin.
16. From the **SpawnActor** node's output execution pin, add a **Delay** node and set its **Duration** property to "**0.4**".
17. From the **Delay** node's **Completed** pin, add a **Set Firing** node and set it to "**false**".
18. Then, off the **Set Firing** node, add another **Delay** node, and this time set the **Duration** property to "**3.0**".
19. From that **Delay** node's **Completed** pin, add a **Set TornadoCooldown** node and set it to "**false**".
20. Compile the Blueprint.

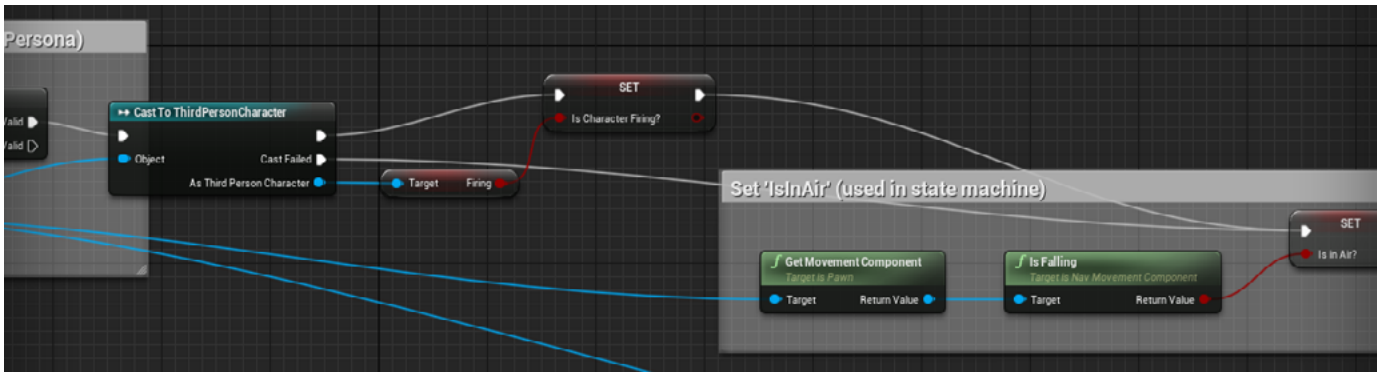


The Potential Issue with Delays
Throughout the Fast Track, we have used a lot of Delay nodes. For small projects, delays are fine, but for larger projects, using the Set Timer node would be much better for performance. Timers are often the best option for projects and will lead to the least amount of potential failure or bugs.

With the **ThirdPersonCharacter** and **BP_Tornado** Blueprints set up, the player should be able to shoot out tornados and jump on them for a bounce. However, without animations, it does not look as cool as it could!

One of the biggest and most common challenges in game development is creating a great experience while working within the constraints you have. For example, the player character has limited animations; but we can utilize what we do have to make these spells still look great.

1. Open the **ThirdPerson_AnimBP** and go to the Event Graph. We are going to edit it slightly.
2. First, add a **Boolean** variable called "**IsCharacterFiring**?"
3. Now, from the **Try Get Pawn Owner** node, add a **Cast To ThirdPersonCharacter** node.
4. Connect the **Cast** node's input execution pin to the **IsValid** node's **Is Valid** pin.
5. From the **Cast** node's **As Third Person Character** pin, add a **Get Firing** node.
6. From the **Get Firing** node's output pin, add a **Set IsCharacterFiring?** node.
7. Connect the **Set IsCharacterFiring?** node's input execution pin to the **Cast** node's output execution pin.
8. Connect the **Set IsCharacterFiring?** node's output execution pin to the **Set IsInAir?** node's input execution pin.
9. Connect the **Cast** node's **Cast Failed** pin to the **Set IsInAir?** node's input execution pin too. (We are doing this because the enemies use the same animation Blueprint, so this cast will fail, and the Blueprint needs to continue.)



10. Open up the State Machine. Off the **Idle/Run** state, create a new state called "**Tornado**".
11. Double-click on the **Transition Rule**.
12. Drag and drop a **Get IsCharacterFiring?** node onto the graph and connect it to the **Result** node's **Can Enter Transition** pin.
13. Return to the State Machine and create a new state leading from the **Tornado** state to the **JumpEnd** state.
14. Double-click on the new **Transition Rule**.
15. Drag and drop a **Get IsCharacterFiring?** node onto the graph and connect it to a **Not** node. Connect the **Not** node to the **Result** node's **Can Enter Transition** pin.
16. Double-click on the **Tornado** state.
17. Drag and drop a **ThirdPersonJump_Start** animation onto the Event Graph and connect it to the **Output Animation Pose** node.
18. In the **Details** panel, uncheck the **Loop Animation** property.
19. Compile the Blueprint.

Now the player character will do a small hand and leg raise when summoning the tornado. It's a small touch, but just the small move adds a lot of character to the character.

Bonus challenge: Make the tornado pull enemies in.

1. Open "**BP_Tornado**".
2. Add a **Radial Force** component. Set the **Radius** property to "**150.0**". Using the left and top orthographic views, position the bottom of the sphere at the bottom of the tornado.
3. Set the **Force Strength** property to "**–500000.0**".
4. Compile the Blueprint.

The only issue now is that our player also gets pulled in by the tornado! We can fix that by making the player a new Object type.

1. Go to the **Project Settings** menu and select "**Collision**".
2. In the **Object Channels** section, click "**New Object Channel**".
3. Name the channel "**Player**" and set **Default Response** to "**Block**".
4. Click "**Accept**".
5. Next, go down to the **Preset** section and click "**New**".

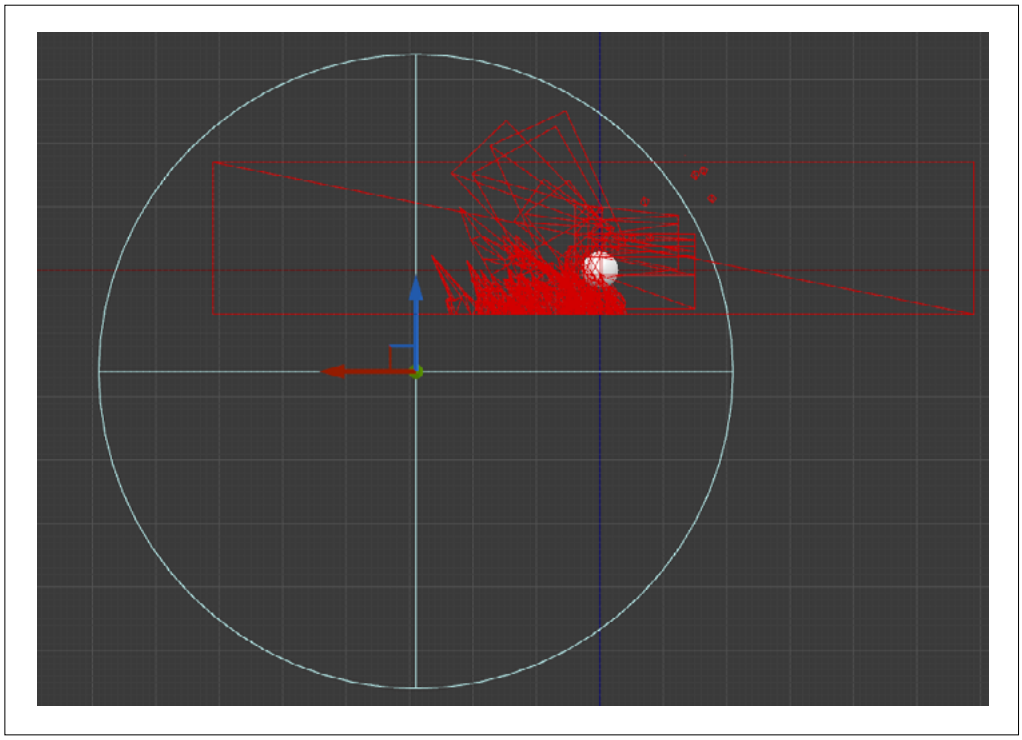
6. Name the profile "**PlayerPreset**", change **CollisionEnabled** to "**Collision Enabled (Query and Physics)**", and set **Object Type** to "**Player**".
7. The rest of the settings should match those shown here.
8. Click "**Accept**".

9. **Important: Double-click on each of the other presets in the list and ensure that for each one the same box is checked for both "Player" and "Pawn"**. If you run into unexpected collision outcomes with Objects (i.e., he powerups), check their collision settings for the **Player** channel.
10. Open up your **ThirdPersonCharacter** Blueprint and select the **Capsule** component.
11. In the **Details** panel, under "**Collision**", change **Collision Presets** to "**PlayerPreset**".
12. Compile your **ThirdPersonCharacter** Blueprint and our player should no longer be affected by the tornado!

/// Challenge Two: Give the player a ground stomp ability that pushes enemies away.

This spell will be different from the tornado spell and should also feel more powerful. Because it will feel more powerful, we should make it more difficult to hit with.

1. Right-click in the **Content Browser** and create a new Blueprint Actor. Name it "**BP_GroundStomp**" and open it up.
2. Add a **Particle System** component.
3. In the **Details** panel, under "**Particles**", select the **P_HeldCharge_Fire_02** template.
4. Add a **Radial Force** component. Set the **Radius** property to "**500.0**" and the **Impulse Strength** property to "**1000.0**". Check **Impulse Vel Change**.
5. Move the **Radial Force** component so that it roughly matches the screenshot shown here.



6. In the Event Graph, add a **Fire Impulse** node off the **Event BeginPlay** node.
7. Next, from the **Fire Impulse** node, add a **Delay** node and set its **Duration** property to "**1.0**".
8. Then, from the **Delay** node's **Completed** pin, add a **Destroy Actor** node.
9. Compile the Blueprint.

Before we build the functionality, we should add a camera shake. This will make the move feel powerful to the player and hide the lack of animations.

1. In the **Content Browser**, right-click and create a new Blueprint with the **Camera Shake** class. You will need to search for the class in the **Search** bar under "**All Classes**". Name the Blueprint "**BP_GroundStompShake**".
2. Open the Blueprint and match the settings to those shown here. After changing the settings, compile and save the Blueprint.

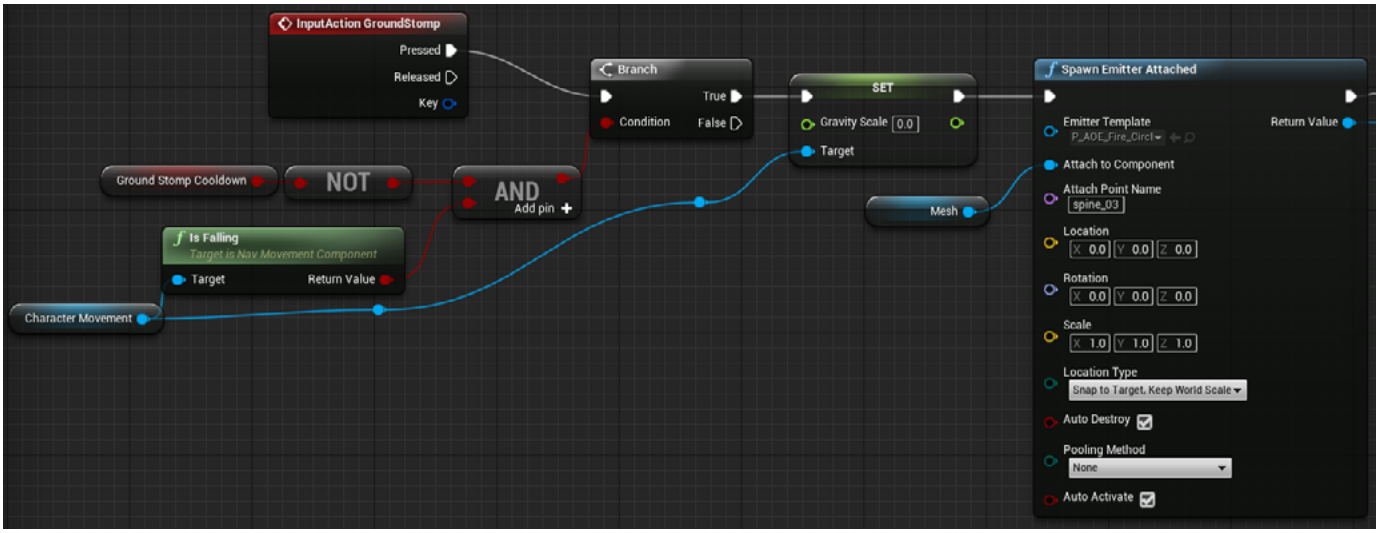
With the ground stomp asset and the camera shake built, we can now build the functionality in our player character. The beginning steps will seem familiar if you just went through Challenge One.

1. Go to the **Project Settings** menu and select "Input". Add a new Action Mapping. Name the Action "**GroundStomp**" and select "**Right Mouse Button**" and "**Gamepad Face Button Right**" as the key values.
2. Open the **ThirdPersonCharacter** Blueprint.
3. Add a **GroundStomp** event node to the Event Graph.
4. Create two **Boolean** variables, one called "**Stomping**" and the other called "**GroundStompCooldown**". Compile the Blueprint.

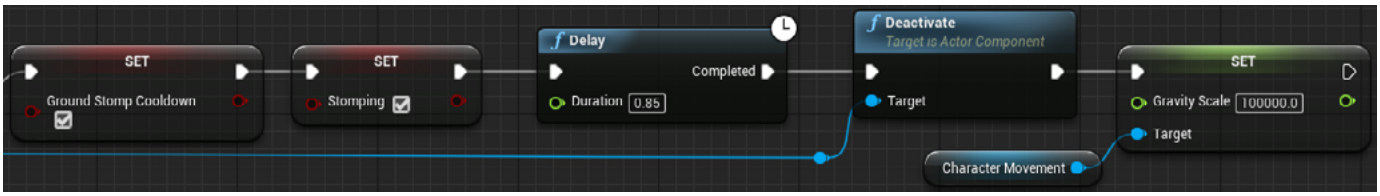
5. Off the **GroundStomp** event's **Pressed** pin, add a **Branch** node.
6. To set up the condition of the **Branch** node:
    a. Drag and drop a **Get GroundStompCooldown** node onto the graph and connect it to a **Not** node.
    b. Drag and drop a reference to the **CharacterMovement** component.
    c. Off the reference node, add an **Is Falling** node.
    d. Place an **And** node in the graph and connect its top input pin to the **Not** node and its bottom input pin to the **Is Falling** node.
    e. Finally, connect the **And** node to the **Condition** pin.

To reiterate, being able to create a great experience with given constraints is a skill every great designer needs. There are many right ways to do the ground stomp code without proper animations. For this method, changing the gravity scale, using particles, and adding a camera shake is how the ground stomp effect is achieved. Once again, if you did something else, that is great! This is just one way to solve the problem.

7. Drag a wire from the **CharacterMovement** reference node and add a **Set Gravity Scale** node. Set the **Gravity Scale** property to "**0.0**" and connect the node's input execution pin to the **Branch** node's **True** pin.
8. Connect a **Spawn Emitter Attached** node to the **Set Gravity Scale** node.
    a. Set the **Spawn Emitter** node's **Emitter Template** property to "**P_AOE_Fire_CircleAttack**".
    b. Drag and drop a reference to the **Mesh** component onto the graph and connect it to the **Spawn Emitter** node's **Attach to Component** pin.
    c. For the **Attach Point Name** property, click "**None**" and type in "**spine_03**".
    d. Set the **Scale** property on the **Spawn Emitter** node to "**1.0**, **1.0**, **1.0**".
    e. Set **Location Type** to "**Snap to Target**, **Keep World Scale**".
    f. Keep **Pooling Method** set to "**None**".
    g. Make sure **Auto Destroy** and **Auto Activate** are checked.



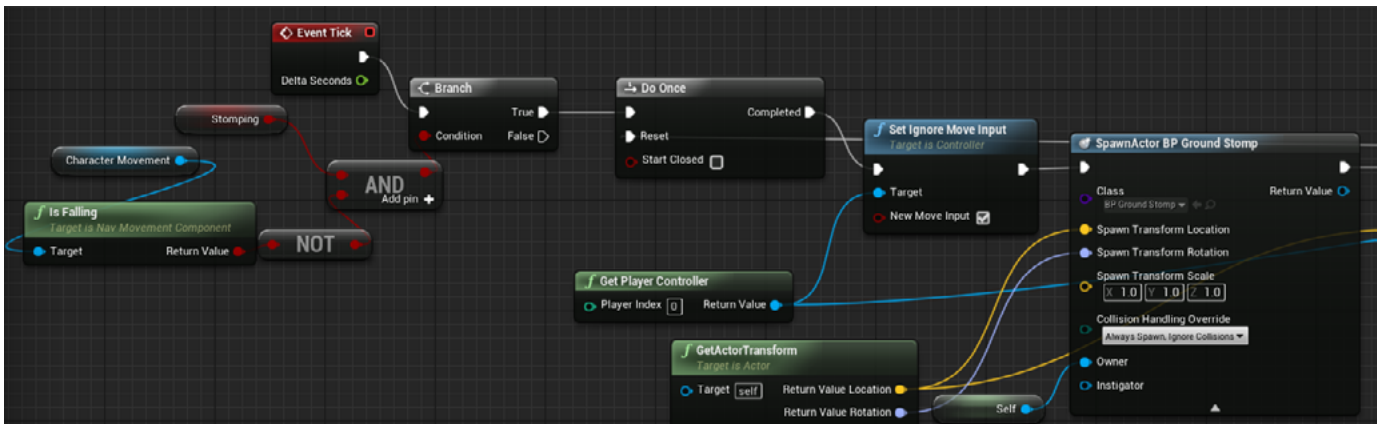
9. From the **Spawn Emitter** node's output execution pin, add a **Set GroundStompCooldown** node and set it to "**true**".
10. Off the **Set GroundStompCooldown** node, add a **Set Stomping** node and set it to "**true**" too.
11. From the **Set Stomping** node, add a **Delay** node and set its **Duration** property to "**0.85**".
12. Next, drag a wire from the **Spawn Emitter** node's **Return Value** pin and add a **Deactivate** node, placing it after the **Delay** node.
13. Off the **Deactivate** node, add another **Set Gravity Scale** node and connect its **Target** pin to the **CharacterMovement** reference. This time, set the **Gravity Scale** property to "**100000.0**". This will cause the player to drop very quickly.
14. Compile the Blueprint.



We are not going to continue with this sequence of nodes for the next problem that needs to be solved. We want to produce the **GroundStomp** Blueprint when the player hits the ground. To do this, we should be checking every frame while the player is in the middle of ground stomping for when the player hits the ground or, in other words, is not falling.

To do this, we are going to use a powerful but dangerous tool. **Event Tick** is an event that runs every single frame. Because it runs every single frame, it can be extremely resource-intensive if used too often. This is why, for example, we do not check if the player is dead every frame, only when the player gets hurt; if we checked every frame, we would be wasting resources.

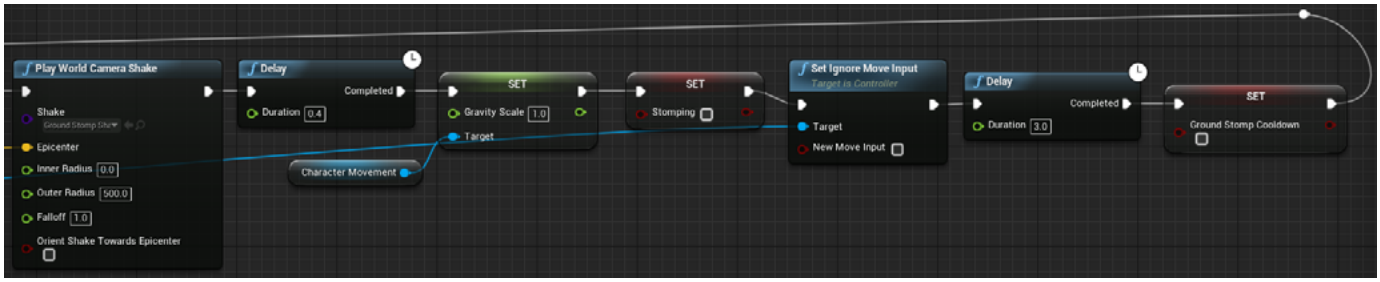
1. Add an Event Tick to the Event Graph. From the Event Tick node, add a Branch node.
2. To set up the condition of the Branch node:
    a. Drag and drop a **Get Stomping** node onto the graph.
    b. Drag and drop a reference to the **CharacterMovement** component.
    c. From the reference node, add an **Is Falling** node and connect it to a **Not** node.
    d. Place an **And** node in the graph and connect its bottom input pin to the **Not** node and its top input pin to the **Get Stomping** node.
    e. Finally, connect the **And** node to the **Condition** pin. (For the initial **GroundStomp** attack, we are checking to make sure the player is in the air and isn't already stomping. Here we are checking if the player is no longer in the air and is currently stomping.)
3. Connect the **Branch** node's **True** pin to a **DoOnce** node. (We only want the ground stomp effect to go off once.)
4. Add a **Get Player Controller** node to the graph. Off its **Return Value** pin, add a **Set Ignore Move Input** node. Be sure to check the **New Move Input** box.
5. Connect the **Set** node's input execution pin to the **DoOnce** node's **Completed** pin.
6. Next, add a **SpawnActorFromClass** node to the graph.
7. Set the **SpawnActor** node's Class property to "**BP_GroundStomp**". Right-click on the **Spawn Transform** pin and select "**Split Struct Pin**".
8. Add a **GetActorTransform** node to the graph. Right-click on the **GetActorTransform** node's **Return Value** pin and select "**Split Struct Pin**". Connect the **Return Value Location** pin to the **SpawnActor** node's **Spawn Transform Location** pin, and the **Return Value Rotation** pin to the **SpawnActor** node's **Spawn Transform Rotation** pin.
9. Set the **SpawnActor** node's **Collision Handling Override** property to "**Always Spawn, Ignore Collisions**" and its **Spawn Transform Scale** property to "**1.0**, **1.0**, **1.0**".
10. Add a **Get Reference To Self** node and connect it to the **SpawnActor** node's **Owner** pin.



11. Off the **SpawnActor** node, add a **Play World Camera Shake** node.
12. Set the **Play World Camera Shake** node's **Shake** property to "**GroundStompShake**".
13. Connect the **Play World Camera Shake** node's **Epicenter** pin to the **GetActorTransform** node's **Return Value Location** pin.
14. Set the **Play World Camera Shake** node's **Outer Radius** property to "**500.0**". The **Falloff** property should be set to "**1.0**".

The next few nodes are going to be easy for you at this point.

15. Off the **Play World Camera Shake** node, add a **Delay** node and set its **Duration** property to "**0.4**".
16. Next, drag and drop a reference to the **CharacterMovement** component onto the graph.
17. Off the reference node, add another **Set Gravity Scale node**. Set the **Gravity Scale** property to "**1.0**" and connect the node's input execution pin to the **Delay** node's **Completed** pin.
18. From the **Set Gravity Scale** node, add a **Set Stomping** node and set it to "**false**".
19. Drag a wire from the **Get Player Controller** node's **Return Value** pin and add a **Set Ignore Move Input** node, placing it after the **Set Stomping** node. Do not check the **New Move Input** box. Connect the **Set Ignore Move Input** node's input execution pin to the **Set Stomping** node's output execution pin.
20. Off the **Set Ignore Move Input** node, add one more **Delay** node and set the **Duration** property to "**3.0**".
21. Off the **Delay** node, add a **Set GroundStompCooldown** node and set it to "**false**".
22. Finally, connect the **Set GroundStompCooldown** node's output execution pin to the **DoOnce** node's **Reset** pin, located all the way back on the left. By doing this, we've reset the node line so it can be called again.
23. Compile the Blueprint.



The final thing to add is a tiny animation to match the ground stomp behavior. We don't have any new animations to add, but we can mess with an existing one.For the condition of the Branch:

1. Open up the **ThirdPerson_AnimBP** and go to the Event Graph. Assuming you followed along with the first challenge, we should already have the **Cast** node set up. If not, look at the Challenge One solution for how we added the **Cast** node.
2. Add a **Boolean** variable called "**IsCharacterStomping**?"
3. From the **Cast** node's **As Third Person Character** pin, add a **Get Stomping** node.
4. Off the **Get Stomping** node's output pin, add a **Set IsCharacterStomping?** node.
5. Place the **Set IsCharacterStomping?** node between the **Set IsCharacterFiring?** and **Set IsInAir?** nodes. Be sure not to accidentally remove the wire connecting the **Cast** node's **Cast Failed** pin and the **Set IsInAir?** node's input execution pin.
6. Open up the State Machine. Off the **JumpLoop** state, create a new state called "**GroundStomp**".
7. Double-click on the **Transition Rule**.
8. Drag and drop a **Get IsCharacterStomping**? node onto the graph and connect it to the **Result** node's **Can Enter Transition** pin.
9. Return to the State Machine and create a new state leading from the **GroundStomp** state to the **Idle/Run** state.
10. Double-click on the new **Transition Rule**.
11. Drag and drop a **Get IsCharacterStomping?** node onto the graph and connect it to a **Not** node. Connect the **Not** node to the **Result** node's **Can Enter Transition** pin.
12. Double-click on the **GroundStomp** state.

13. Drag and drop a **ThirdPersonJump_Start** animation onto the Event Graph and connect it to the **Output Animation Pose** node.
14. In the **Details** panel, set the **Play Rate** (not **Play Rate Basis**) property to "–**0.5**". (This will play the animation at half speed, and because the value is negative, it will play in reverse, which looks very good for a ground stomp effect.)
15. Uncheck the **Loop Animation** property. (With the added 0.4-second delay on the Blueprint, the character will strike a very brief pose upon landing, thus increasing the feeling of strength and power from it.)
16. Compile the Blueprint.

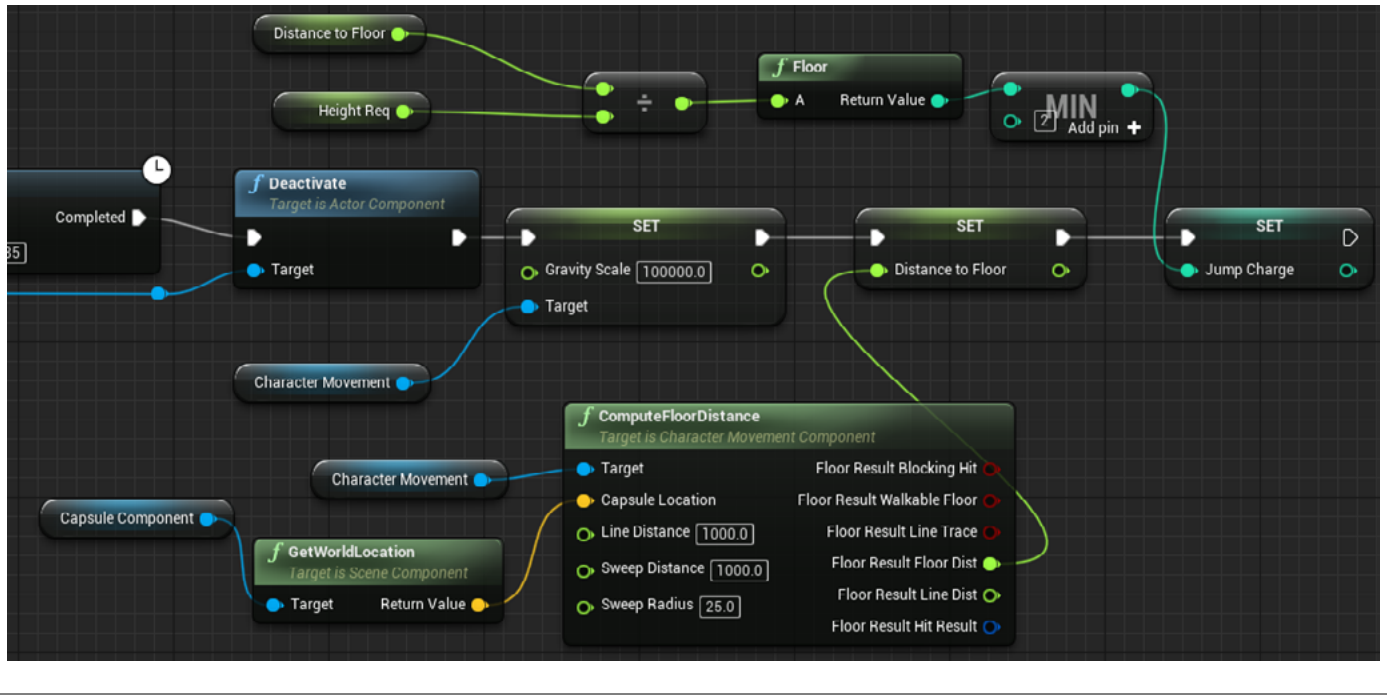
Try the move! Knock enemies around!

/// Bonus challenge: Make it so the higher the player jumps, the farther they push enemies away.

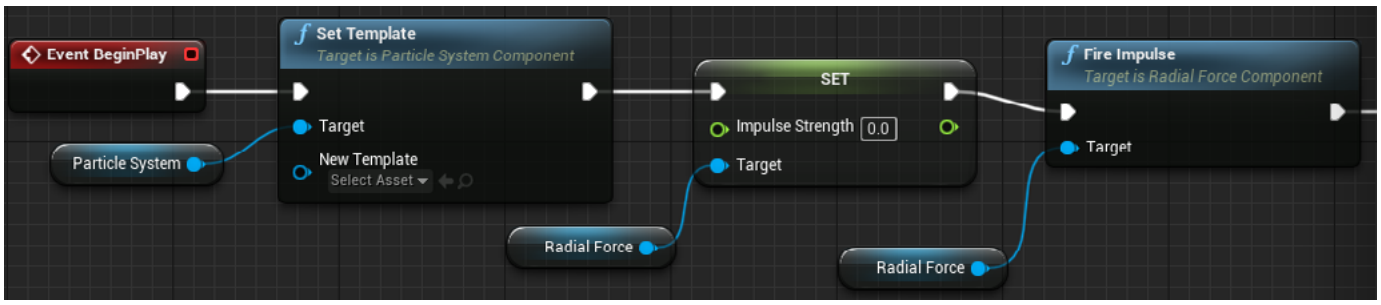
To solve the bonus challenge, we are going to have three levels of stomping. The level will be determined by recording the character's distance to the floor when they start the stomp and dividing it by the height requirement we will set for each level of stomping. The quotient will tell us what level of stomping the player is doing.

15. The first step is to create the following three variables. After you create each one, compile the Blueprint.
    a. A **Float** variable called "**DistanceToFloor**".
    b. Another **Float** variable called "**HeightReq**". Set its default value to "**500**". (This may need to change depending on your character's set jump height.)
    c. An **Integer** variable called "**JumpCharge**".
16. Find the **Set Gravity Scale** node where we set the **Gravity Scale** property to "**100000.0**". Off that node's output execution pin, add a **Set DistanceToFloor** node.
17. Next, add a **ComputeFloorDistance** node to the graph. It should automatically add a **CharacterMovement** component reference in its **Target** pin.
18. Drag and drop a **Capsule Component** reference onto the Event Graph and connect it to a **GetWorldLocation** node.
19. Connect the **GetWorldLocation** node's **Return Value** pin to the **ComputeFloorDistance** node's **Capsule Location** pin.
20. Set the **ComputeFloorDistance** node's **Line Distance** and **Sweep Distance** properties to "**1000.0**" and its Sweep Radius property to "**25.0**".
21. Right-click on the **ComputeFloorDistance** node's **Floor Result** pin and select "**Split Struct Pin**". Connect the **Floor Result Floor Dist** pin to the **Set DistanceToFloor** node's green input pin.
22. Off the **Set DistanceToFloor** node's output execution pin, add a **Set JumpCharge** node.
23. Drag and drop a **Get DistanceToFloor** node and a **Get HeightReq** node onto the graph.
24. Add a **float ÷ float** node to the graph and connect its top input pin to the **DistanceToFloor** node and its bottom input pin to the **HeightReq** node.
25. From the **float ÷ float** node's output pin, add a **Floor** node. This node immediately rounds down any number put into it, which guarantees that we create an integer.
26. Next, connect the **Floor** node's **Return Value** pin to a **Min** (integer) node. Set the value of the **Min** node's bottom input pin to "**2**". Finally, connect the **Min** node's output pin to the **Set JumpCharge** node. The **Min** node ensures that if the number divided is greater than 2, we still always get 2.
27. Compile the Blueprint.

Even though we have three levels of charge, we are using a maximum of two because we will be utilizing an array and arrays begin at 0, not at 1. Now that we can get the height distance, we need to tell **BP_GroundStomp** what those numbers are and what to do with them.

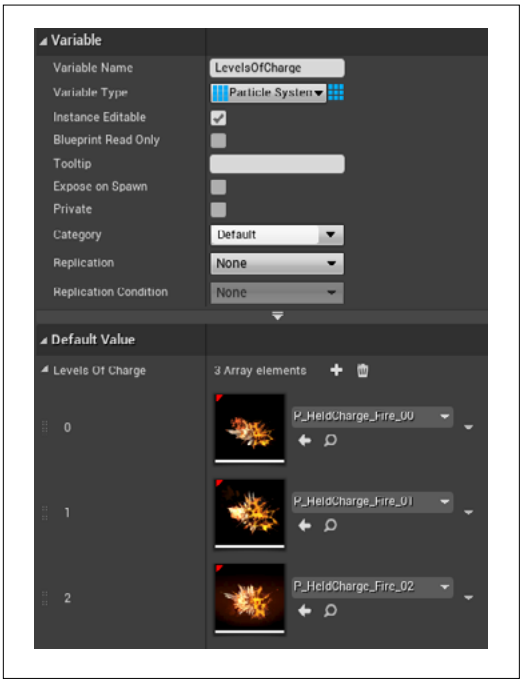


1. Open "**BP_GroundStomp**". We are going to be putting a few more nodes between the **Event BeginPlay** and **Fire Impulse** nodes.
2. First, drag and drop references to the **Particle System** component and the **Radial Force** component.
3. From the **Particle System** reference, add a **Set Template** node. Connect the **Set Template** node's input execution pin to the **Event BeginPlay** node's output execution pin.
4. Off the **Set Template** node, add a **Set Impulse Strength** node. Connect the **Set Impulse Strength** node's output execution pin to the **Fire Impulse** node's input execution pin.



5. Back at the **Set Template** node, right-click on the **New Templat**e pin and select "**Promote to Variable**". Call the new variable "**LevelsOfCharge**".
6. Set the container type to "**Array**" and select "**Change Variable Type**" in the pop-up that appears afterward.
7. When you do this, the **Set Template** node on the Event Graph does not like what just happened. Break the link between the **LevelsOfCharge** node and the **Set Template** node.
8. Compile the Blueprint.

9. Select the **LevelsOfCharge** node and in the **Details** panel under the **Default Value** section, add elements using the plus sign until there are three array elements. Set them in the following order:
    a. **P_HeldCharge_Fire_00**
    b. **P_HeldCharge_Fire_01**
    c. **P_HeldCharge_Fire_02**



10. From the **LevelsOfCharge** reference, add a **Get (a copy)** node and connect its output pin to the **Set Template** node's **New Template** pin.
11. Separately, add a **Get Owner** node to the graph. (Note: During Challenge Two's main challenge, we set the **ThirdPersonCharacter** as the owner of this Blueprint. This needs to be done for this to work.)
12. From the **Get Owner** node, add a **Cast To ThirdPersonCharacter** node and set it to a pure cast by right-clicking on the node and selecting "**Convert to Pure Cast**".
13. From the **Cast** node's **As Third Person Character** pin, add a **Get JumpCharge** node.
14. Connect the **Get JumpCharge** node's output pin to the **Get (a copy)** node's **Integer** pin.
15. Compile the Blueprint.

The ground stomp now emits a particle effect in line with the level of jump: tiny jump, tiny particle effect; BIG JUMP, BIG PARTICLE EFFECT. Let's get the power of the ground stomp in line with the particle effects.

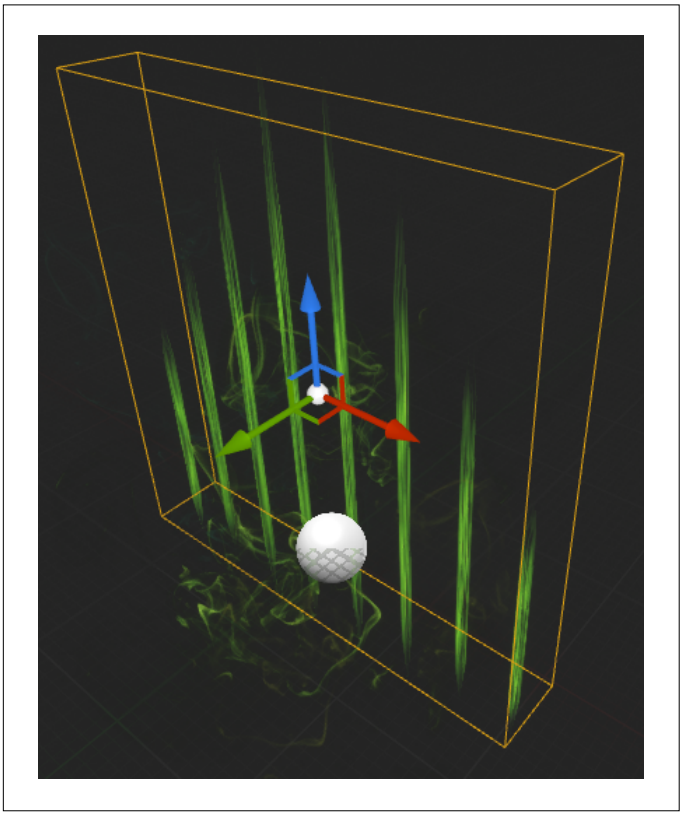
1. From the **JumpCharge** reference node, add an **integer + integer** node and set the value of its bottom pin to "**1**". (Because the array goes from 0 to 2, and we want **Impulse Strength** to go from 1,000 to 3,000, we need to add 1 to the **JumpCharge** amount.)
2. Connect the **integer + integer** node's output pin to an **integer * integer** node. Set the value of the **integer ***

integer node's bottom pin to "**1000**". (Now this equation should return an integer on a scale of 1,000 to 3,000.)
3. Finally, connect the **integer * integer** node's output pin to the **Set Impulse Strength** node's green input pin. The Event Graph should automatically convert the integer into a float.
4. Compile and save the Blueprint.

Now, depending on how high the player character jumps, they produce three different particle effects with three different levels of strength on the ground stomp!

Challenge Three: Create a dash that leaves behind a wall that enemies can't walk through.

1. Right-click in the **Content Browser** and create a new Blueprint Actor. Name it "**BP_Dashwall**" and open it up.
2. Add a **Particle System** component.
3. In the **Details** panel, under "**Particles**", select the **P_Gate_Green_Locked_01** template. Set the **Location** property to "0.0, 0.0, –120.0".
4. Add a **Box Collision** component. Scale and move the box so that it encompasses the particle effect.

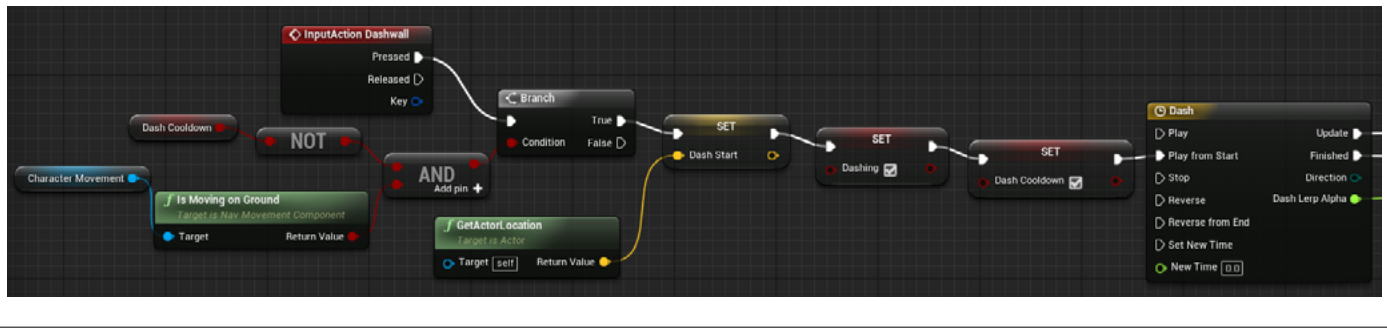


5. In the **Box Collision** component's Details panel, check the collision settings. Set **Collision Presets** to "**Custom**", and change the **Camera ObjectType** collision response to "**Ignore**". This will ensure our camera doesn't get stuck.
6. In the same settings, change the **Player ObjectType** collision response to "**Overlap**". This will allow the player to run through the wall, but not the enemies. (If you do not have the **Player ObjectType**, please see the solution guide for Challenge One's bonus challenge, where we added it.)
7. In the Event Graph, add a **Delay** node off the **Event BeginPlay** node and set its **Duration** property to "**5.0**".
8. From the **Delay** node's **Completed** pin, add a **Destroy Actor** node.
9. Compile the Blueprint.

With the dashwall made, now we can begin adding functionality in our player character.

1. Go to the **Project Settings** menu and select "**Input**". Add a new Action Mapping. Name the Action "**Dashwall**" and select "**Left Shift**" and "**Gamepad Face Button Top**" as the key values.

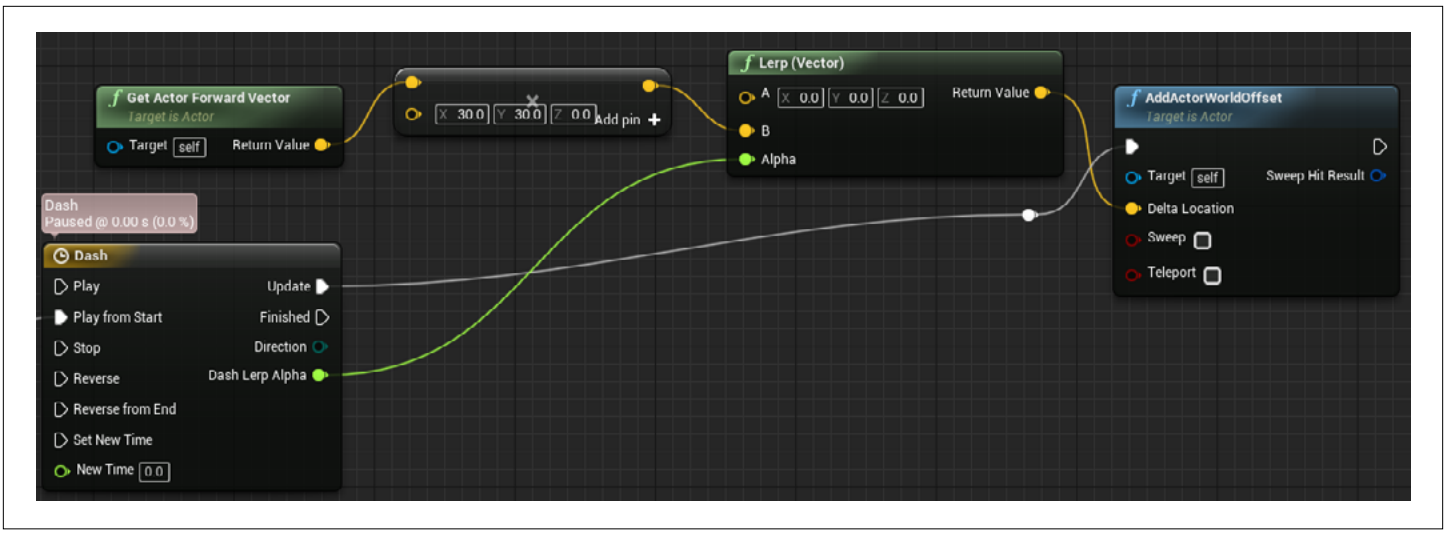
2. Open the **ThirdPersonCharacter** Blueprint.
3. Add a **Dashwall** event node to the Event Graph.
4. Create two **Boolean** variables, one called "**Dashing**" and the other called "**DashCooldown**". Compile the Blueprint.
5. Create a **Vector** variable called "**DashStart**". Compile the Blueprint.
6. Off the **Dashwall** event node's **Pressed** pin, add a **Branch** node.
7. To set up the condition of the **Branch** node:
    a. Drag and drop a **Get DashCooldown** node onto the graph and connect it to a **Not** node.
    b. Drag and drop a reference to the **CharacterMovement** component.
    c. From the reference node, add an **Is Moving on Ground** node.
    d. Place an **And** node in the graph and connect its top input node to the **Not** node and its bottom input pin to the **Is Moving on Ground** node.
    e. Finally, connect the **And** node to the **Condition** pin.
8. Add a **Set DashStart** node to the graph and connect its input execution pin to the **Branch** node's **True** pin.
9. Add a **GetActorLocation** node to the graph and connect its **Return Value** pin to the **Set DashStart** node's yellow input pin.
10. From the **Set DashStart** node, add a **Set Dashing** node and set it to "**true**".
11. Off the **Set Dashing** node, add a **Set DashCooldown** node and set it to "**true**" too.
12. Add a **Timeline** node by right-clicking in the graph, typing "**Add Timeline**", and pressing **Enter.** Call it "**Dash**". Connect the **Set DashCooldown** node's output execution pin to the **Dash** Timeline node's **Play from Start** pin.
13. Compile the Blueprint.



14. Double-click on the **Dash** Timeline node to open the Timeline Editor.
15. In the upper-left corner of the Timeline Editor, click the **f+** button to add a Float Track. Name it "**DashLerpAlpha**".
16. Set the **Length** property to "**0.25**".

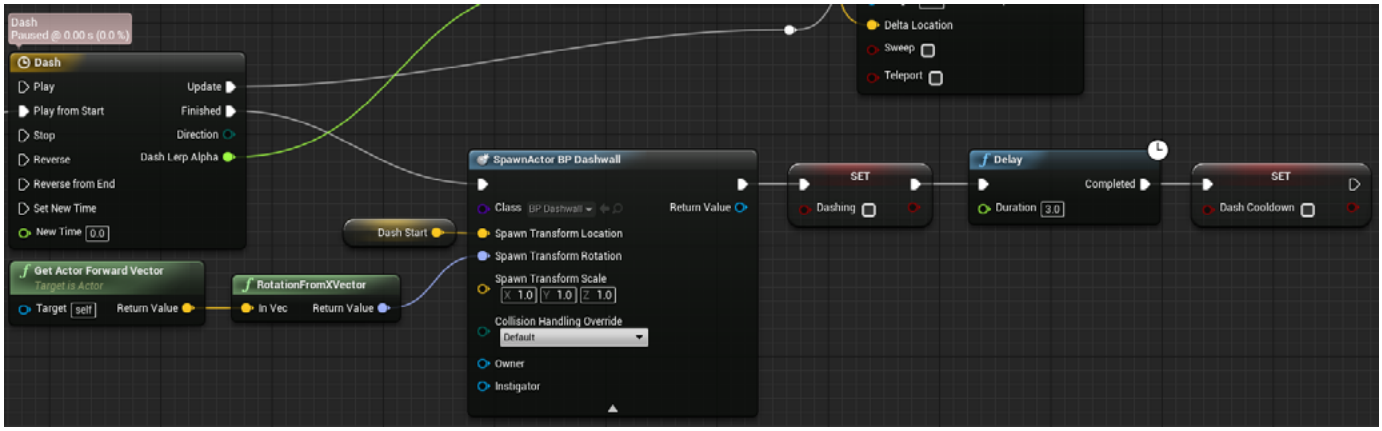
17. Hold down the **Shift** key and left-click at the "**0.0**, **0.0**" coordinates. Adjust the numbers if you misclick.
18. Hold down the **Shift** key and left-click at the "**0.25**, **1.0**" coordinates. Once again, adjust if you misclick.
19. Compile the Blueprint and head back to the Event Graph.

20. From the **Dash** Timeline node's **Update** pin, add an **AddActorWorldOffset** node.
21. Now add a **Get Actor Forward Vector** node to the graph. Off its **Return Value** pin, add a **vector * vector** node and set the value of its bottom pin to "**30.0**, **30.0**, **0.0**".
22. Add a **Lerp (Vector)** node to the graph and connect the **vector * vector** node's output pin to the **Lerp** node's **B** pin. Keep the value of the **A** pin set to "**0.0**, **0.0**, **0.0**".
23. Connect the **Lerp** node's **Return Value** pin to the **AddActorWorldOffset** node's **Delta Location** pin.
24. Back at the Dash Timeline node, connect the **Dash Lerp Alpha** pin to the Lerp node's **Alpha** pin.



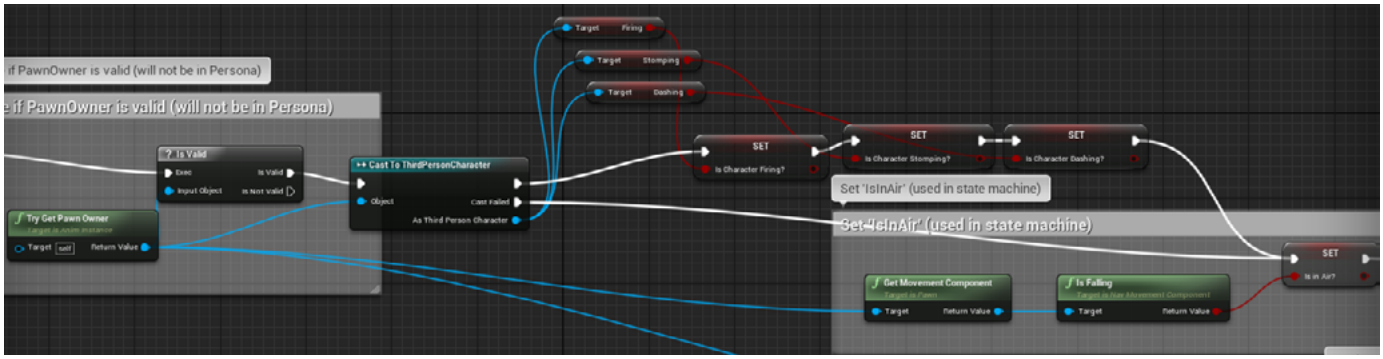
25. Off the **Dash** Timeline node's **Finished** pin, add a **SpawnActorFromClass** node.
26. Set the **SpawnActor** node's Class property to "**BP_Dashwall**".
27. Right-click on the **SpawnActor** node's **Spawn Transform** pin and select "**Split Struct Pin**".
28. Drag and drop a reference to **DashStart** onto the graph and connect it to the **SpawnActor** node's **Spawn Transform Location** pin.
29. Find the **Get Actor Forward Vector** node or add another one. Off its **Return Value** pin, add a **RotationFromXVector** node.
30. Connect the **RotationFromXVector** node's **Return Value** pin to the **SpawnActor** node's **Spawn Transform Rotation** pin.

31. From the **SpawnActor** node's output execution pin, add a **Set Dashing** node and set it to "**false**".
32. From the **Set Dashing** node's output execution pin, add a **Delay** node and set its **Duration** property to "**3.0**".
33. From the Delay node's **Completed** pin, add a **Set DashCooldown** node and set it to "**false**".

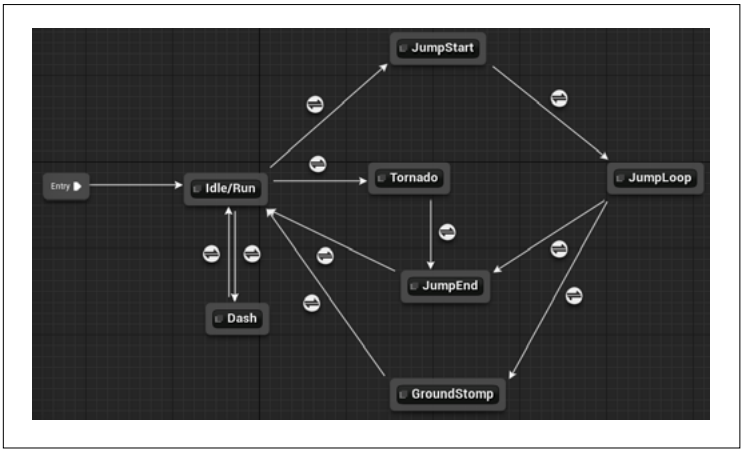


That should do it! The **ThirdPersonCharacter** and **BP_Dashwall** Blueprints will move the player and leave a wall behind them that will block enemies. Last thing we need to do is add an animation!

1. Open up the **ThirdPerson_AnimBP** and go to the Event Graph. Assuming you followed along with the first or second challenge, we should already have the **Cast** node set up. If not, look at the Challenge One solution for how we added the **Cast** node.
2. Add a **Boolean** variable called "**IsCharacterDashing**?"
3. From the **Cast** node's **As Third Person Character** pin, add a **Get Dashing** node.
4. From the **Get Dashing** node's output pin, add a **Set IsCharacterDashing?** node.
5. Place the **Set IsCharacterDashing**? node between the **Set IsCharacterStomping?** and **Set IsInAir?** nodes. (These nodes were added in Challenges One and Two.) Be sure not to accidentally remove the wire connecting the **Cast** node's **Cast Failed** pin and the **Set IsInAir?** node's input execution pin.



6. Open up the State Machine. Off the **Idle/Run** state, add a new state called "**Dash**".
7. Double-click on the **Transition Rule**.
8. Drag and drop a **Get IsCharacterDashing?** node onto the graph and connect it to the **Result** node's **Can Enter Transition** node.
9. Return to the State Machine and create a new state leading from the **Dash** state back to the **Idle/Run** state.
10. Double-click on the new **Transition Rule**.
11. Drag and drop a **Get IsCharacterDashing?** node onto the graph and connect it to a **Not** node. Connect the **Not** node to the **Result** node's **Can Enter Transition** pin.



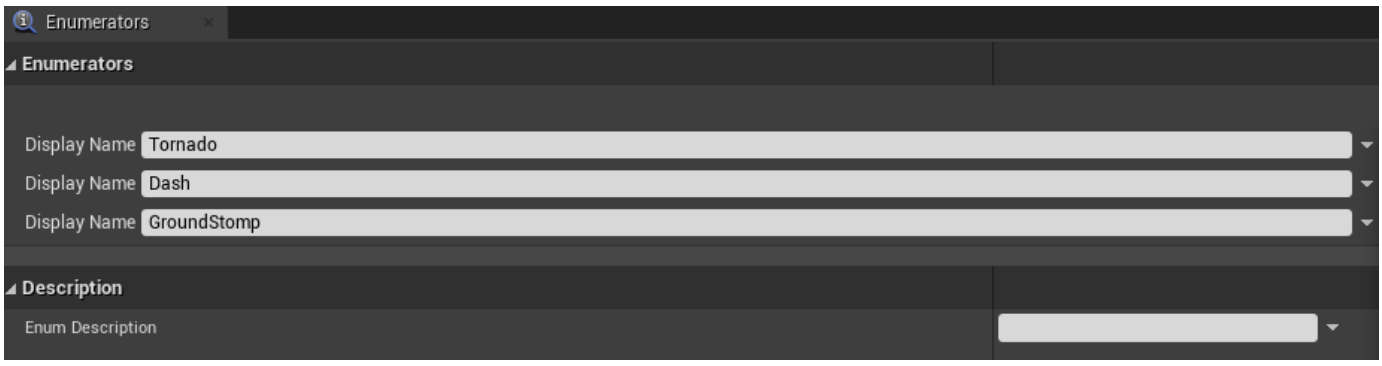
12. Double-click on the **Dash** state.
13. Drag and drop a **ThirdPersonRun** animation onto the Event Graph and connect it to the **Output Animation Pose** node.
14. In the **Details** panel, set the **Play Rate** (not **Play Rate Basis**) property to "**10.0**". (This will make the character look like they are running very quickly when dashing.)
15. Compile and save the Blueprint.

Try dashing with this new animation! Block off enemies and push them away using all of the spells at your disposal!

/// Challenge Four: Create a cooldown system for the spells and make a UI to show what spells are available.

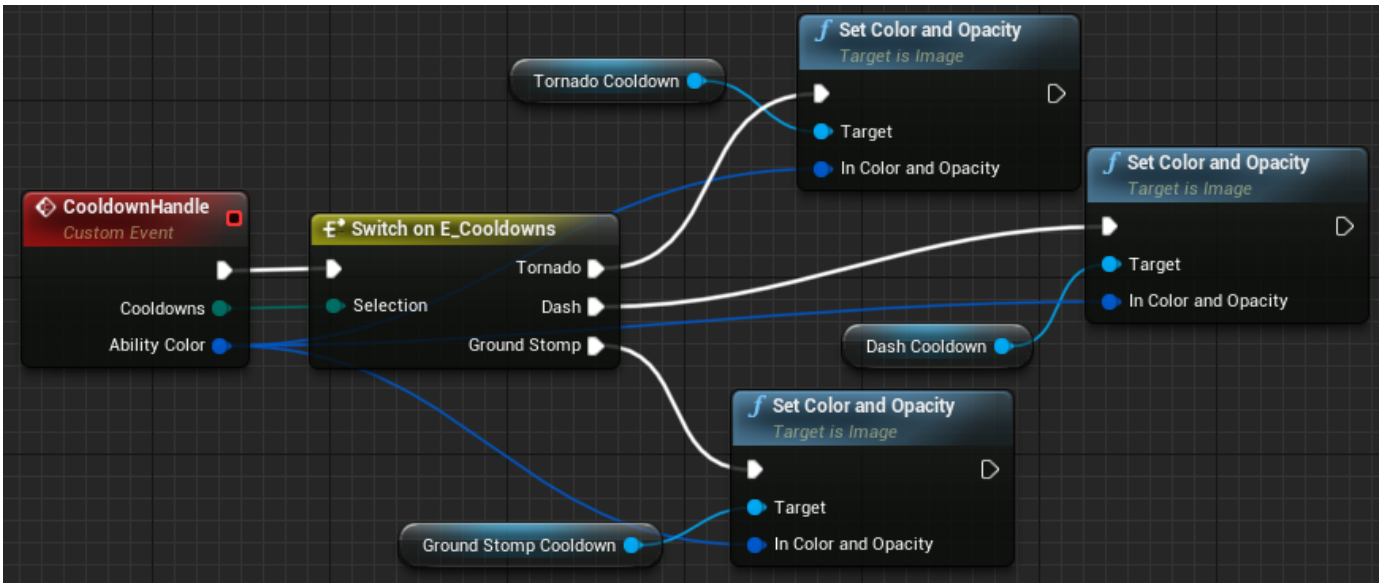
If you have been following the other challenges above, you'll see that we've added **Boolean** variables for **TornadoCooldown**, **DashCooldown**, and **GroundStompCooldown**. If you have not been following along, you can look back now to see where we added the variables. With the variables set and the **Delay** nodes in place, we already have a cooldown system set up, and we know exactly where the cooldowns start and end. Now we just need the Character to speak to the HUD about when the spells are available.

1. First, go to the **Content Browser** and create an enumeration named "**E_Cooldowns**". Add three enumerators: "**Tornado**", "**Dash**", and "**GroundStomp**". Save the enumeration.

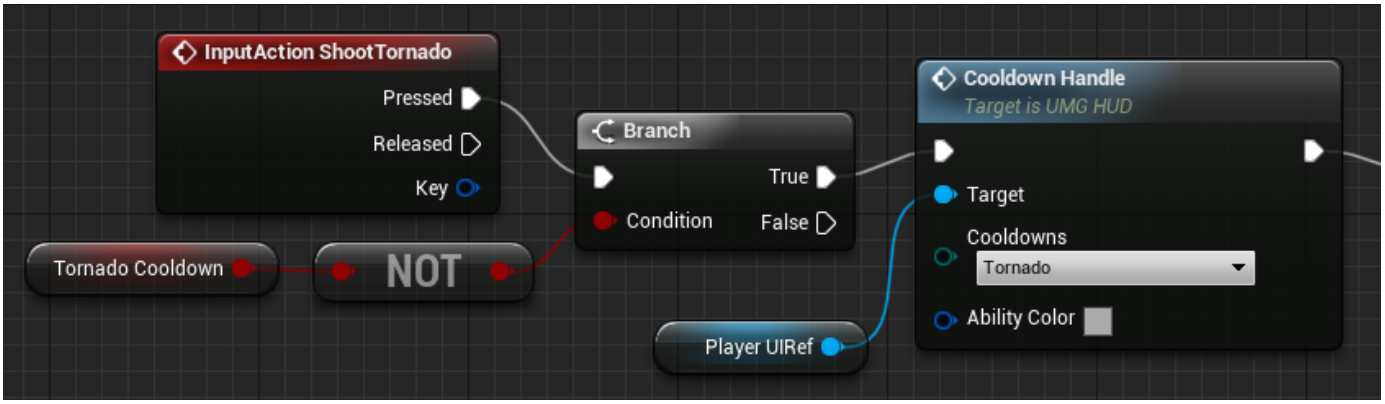


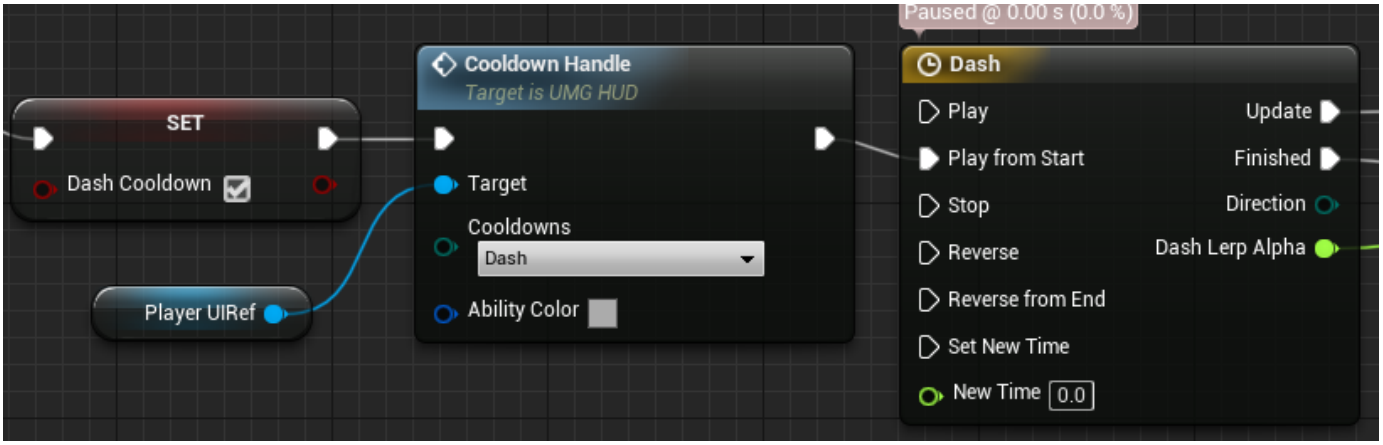
2. Open up **UMG_HUD**.
3. Add a **Horizontal Box** widget to the **Canvas** panel and anchor it to the bottom right.

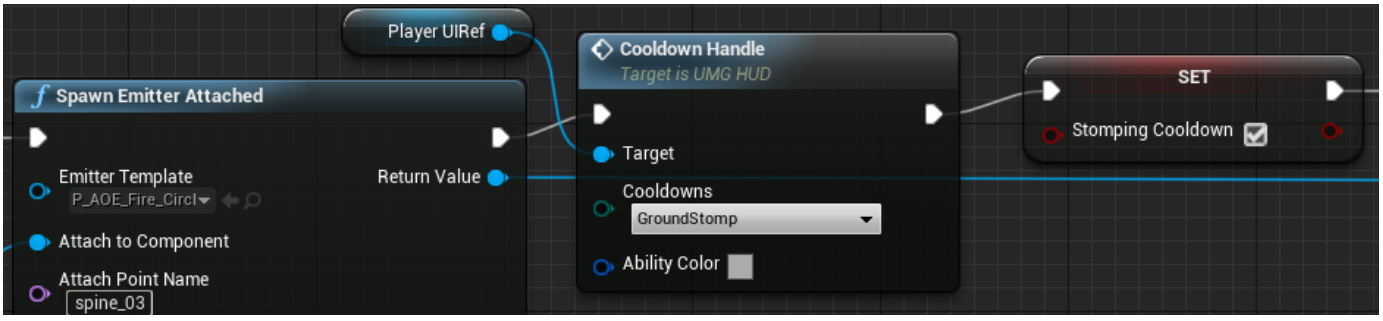
4. Inside the **Horizontal Box** widget, add three **Image** widgets. In the **Details** panel for each of the **Image** widgets, set the **Size** property to "**Fill**"; the **Horizontal Alignment** property to "**Horizontally Align Fill**", and the **Vertical Alignment** property to "**Vertically Align Fill**".
    a. Name the first image "**TornadoCooldown**" and set its **color** to "**blue**".
    b. Name the next image "**DashCooldown**" and set its **color** to "**green**".
    c. Name the final image "**GroundStompCooldown**" and set its **color** to "**red**".
5. Make sure **Is Variable** is checked for all of the **Image** widgets.
6. Move over to the UMG Event Graph.
7. Create a custom event called "**CooldownHandle**". Give it two input parameters. Name the first parameter "**Cooldowns**" and set its type to "**E_Cooldowns**". Name the second one "**Ability Color**"; set its type to "**Linear Color**" (not "**Color**") and its default value to "**gray**".
8. Compile the Blueprint.
9. From the event node's **Cooldowns** pin, add a **Switch on E_Cooldowns** node.
10. Drag and drop three references onto the Event Graph: **TornadoCooldown**, **DashCooldown**, and **GroundStompCooldown**.
11. From each one of those references, connect a separate **Set Color and Opacity** node. Then connect each node's input execution pin to the relevant execution output pin on the **Switch** node.
12. Connect the event node's **Ability Color** pin to each of the three **In Color and Opacity** pins on the **Set** nodes.
13. Compile the Blueprint.



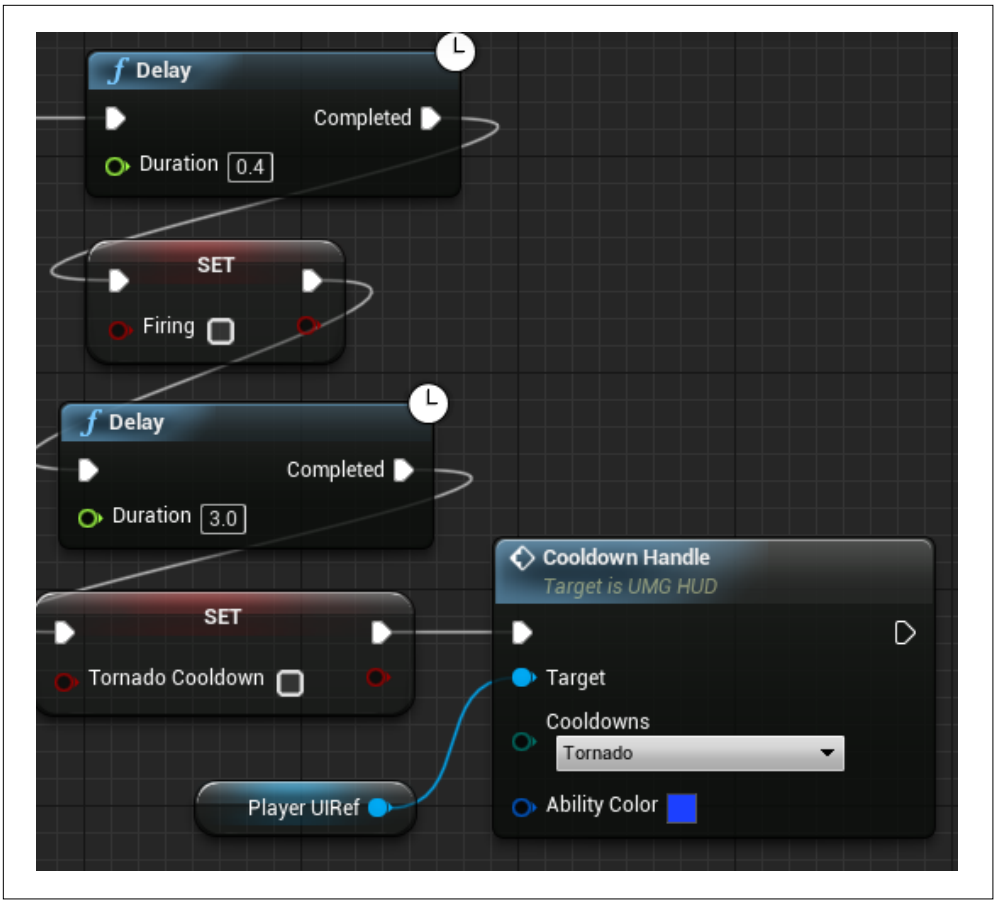
14. Now open the **ThirdPersonCharacter** Blueprint.
15. In the Event Graph, find the spots where we placed the **Set TornadoCooldown**, **Set DashCooldown**, and **Set GroundStompCooldown** nodes that are set to "**true**".
    a. Drag and drop three references to **PlayerUIRef** (you created this reference in Workshop Three) onto the graph and place one near each node.
    b. Then, from each reference node, add a **CooldownHandle** function.
    c. Connect the **CooldownHandle** node to the node line.
    d. Set the **Cooldowns** enumeration on each **CooldownHandle** node to the relevant spell.
    e. Keep **Ability Color** on each **CooldownHandle** node set to the default "**gray**".

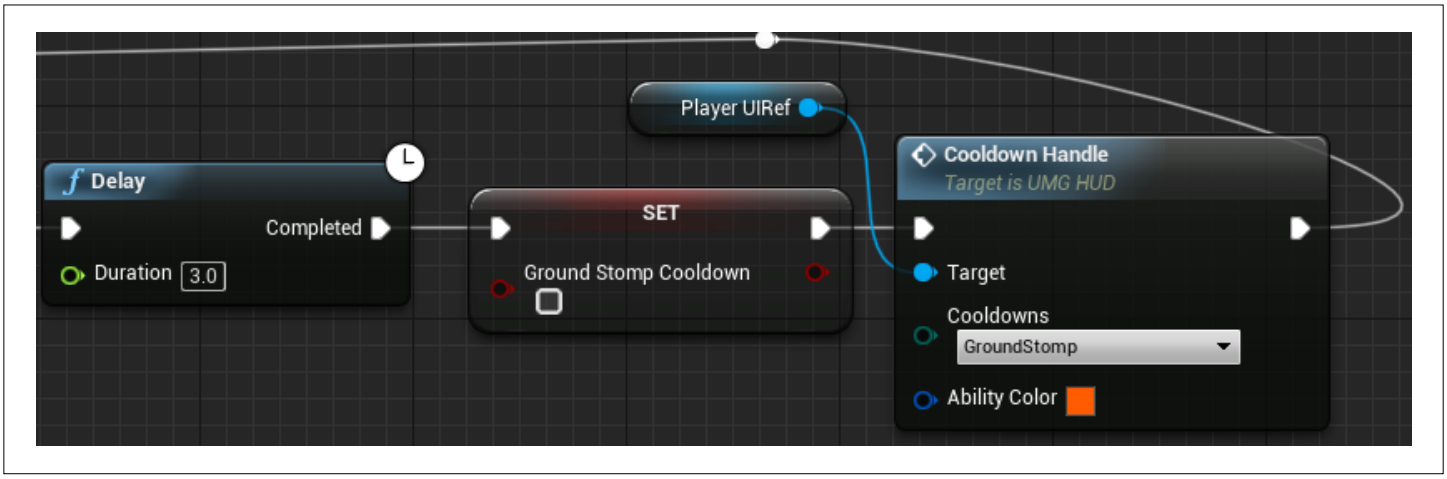


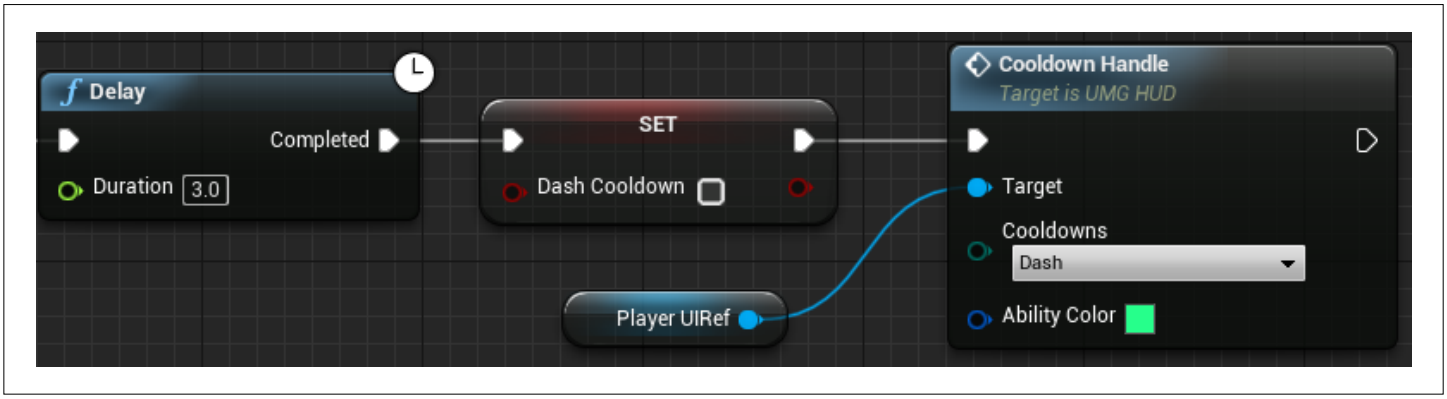




16. Finally, at the end of each node line for each spell, find the spots where we placed the **Set TornadoCooldown**, **Set DashCooldown**, and **Set GroundStompCooldown** nodes that are set to "**false**".
    a. Drag and drop three references to **PlayerUIRef** onto the graph and place one near each node.
    b. Then, from each reference node, add a **CooldownHandle** function.
    c. Connect the **CooldownHandle** node to the node line. (Note: For the **GroundStomp** event, do not forget to attach the **Cooldown Handle** node to the **DoOnce** node's **Reset** pin.)
    d. Set the **Cooldowns** enumeration to the relevant spell.
    e. Set **Ability Color** on each **CooldownHandle** node to match the colors you used for each one in the **UMG_HUD** Widget Blueprint.


Delay
Completed
Duration 0.4
SET
Firing
Delay
Completed
Duration 3.0
SET
Tornado Cooldown
Cooldown Handle
Target is UMG HUD
Target
Cooldowns
Tornado
Ability Color
Player UIRef


Player UIRef
Delay
Completed
Duration 3.0
SET
Ground Stomp Cooldown
Cooldown Handle
Target is UMG HUD
Target
Cooldowns
GroundStomp
Ability Color


Delay
Completed
Duration 3.0
SET
Dash Cooldown
Cooldown Handle
Target is UMG HUD
Target
Cooldowns
Dash
Ability Color
Player UIRef

Final Discussion

Now that you have completed the Unreal Fast Track, you should feel comfortable creating your own project. As you look back at the projects you made, the courses you took, and the game you created, talk with your team once more about the 3 **D**'s.

- **Difficulties**: What was the most difficult or confusing part of the whole Unreal Fast Track?
- **Discoveries**: What were some interesting, fascinating, or exciting things you discovered while completing the Fast Track? What features or tools have you found that you are most excited to try out?
- **Dreams**: What kind of projects do you want to make with Unreal Engine? Did you think of an idea during the Fast Track that you want to try bringing to life now? Now is the time to go for it!

CONGRATULATIONS!

You've finished Workshop Five of the Fast Track!

The Cycle | YAGER