3
Advanced Blueprint, Animation, and Sound

Well done on reading through till this Chapter. We will be learning how to animate our character using the very powerful *animation and sound tools* included in UE4. By the time the chapter is through, the character that we have created for Barrel Hopper will be moving around in well-blended articulated motion. UE4 provides an in-depth and easy-to-use animation pipeline that takes your characters from stoic, silent, statues to dynamic living features of the game world. The tools of this pipeline are very similar to Blueprint in nature yet very different in practice. You will learn how to master these tools so that you may bring true justice to the sounds and animations provided to you by artists, thus bringing your game characters to life.

During the course of this chapter, we will also explore the 3D sound functionality provided by Unreal as well as learn how we can utilize the sound system in conjuncture with the animation system to create animation-driven sound effects such as footsteps. These animations and sounds will require information from the game world—this chapter will also teach you how to communicate between the game world and these other components of the engine.

The chapter will cover the following points:

- UE4 animation pipeline and how to export animation assets
- Working with UE4 animation tool
- Creating events and loops with animation montages
- Creating animation graphs and how to apply these graphs to in-game characters
- Receiving queues from the game world/character state to play the correct animation

- Loading and modifying sounds
- Playing sounds in 3D space
- Working with the animation montage tool to create animation-driven sounds
- Advanced blueprint techniques
- Creating and utilizing basic HUD and font objects

Cleaning up shop

We left off with our Barrel Hopper project, having just implemented some *respawning logic*. In its current state our project is a basic **prototype** at best. During the course of this chapter we will be refining our previous implementations as well as adding new layers of polish via the inclusion of animation and sound effects. However, before we do this, we should clean up the functionality we already have in place.

As it is now, when our player dies we leave nothing in charge of the main camera. You may have noticed that, upon player collision with a barrel, the camera will disjoint and show a default view of the world's origin. We need to append to the functionality we created as an **event delegate** so that, when our player is destroyed, the game mode creates a new camera and sets it as our main view target. We can then interpolate from this new camera to our Player camera upon respawn to create a nice, clean respawn effect. Before we do this, I think now is a great time to introduce you to some new concepts that let you organize your Blueprint graphs.

Logic flow

Unreal Engine's Blueprint system features a few nodes whose sole purpose is to control the execution flow of the blueprint. They will divert or navigate the critical execution path to flow in a desired direction or sequence. You have already experienced this at a small level using a branch node that will dictate the execution flow path based on the state of a `bool` value, and the reroute node that lets you clean up your Blueprint graph lines. The other flow control nodes provided by Unreal are as follows:

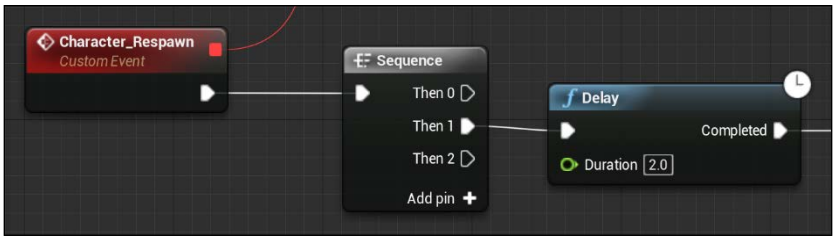
- **Branch Node**: This node will execute one of two paths based on the state of an input `bool`.
- **Do N**: Will only allow an execution path to enter a specified number of times. Execution count can be reset by an execution input.
- **Do Once/Do Once Multi input**: This node only allows a specified execution path to happen once.

- **FlipFlop**: Will execute one of two paths in an alternating fashion every time execution passes through the node.
- **ForLoop**: This node will act as a traditional `for` loop, a very common iteration structure used across most programming languages. You can specify a first and last index and an execution path for the loop, and one once the loop has completed.
- **ForLoopWithBreak**: As previous, but you may break out of the iteration logic early by navigating the execution flow to a break node.
- **While Loop**: Will remain in loop until a `bool` input is set to `false`.
- **Gate**: This node will continue the execution flow until it has been flagged to close, at which point the flow will stop at the Gate node.
- **MultiGate**: Similar to the previous bullet, however allows you to specify multiple execution flow outputs that can be chosen via an integer index.
- **Sequence**: Allows you to create a sequence of execution paths that will be fired in order.

Using a sequence

Before we begin to create the respawn camera functionality, we will implement a *sequence* to maintain a tidy Blueprint Event graph. As stated previously, the sequence node allows us to create a series of execution paths that will be executed in order. This allows us to associate similar functionality with a given sequence path. In our case we will be creating two groupings of functionality. One to spawn a new camera that can be used while our character is respawning, and another to respawn our character after a certain amount of time has passed. We have already created a basic implementation of the latter in the previous chapter.

Create a sequence node now and ensure that it has two output execution paths. They should be titled **Then 0** and **Then 1**. Place this node directly after the `Character_Respawn` event and connect the execution path to the input pin, then append the player respawn functionality we created in the last chapter from `Delay` onwards to the **Then 1** output execution path. You should see something similar to this:



We will now create our camera spawn functionality and append it to the **Then 0** output pin. This means that our camera spawn functionality will execute, *then* our player respawn functionality.

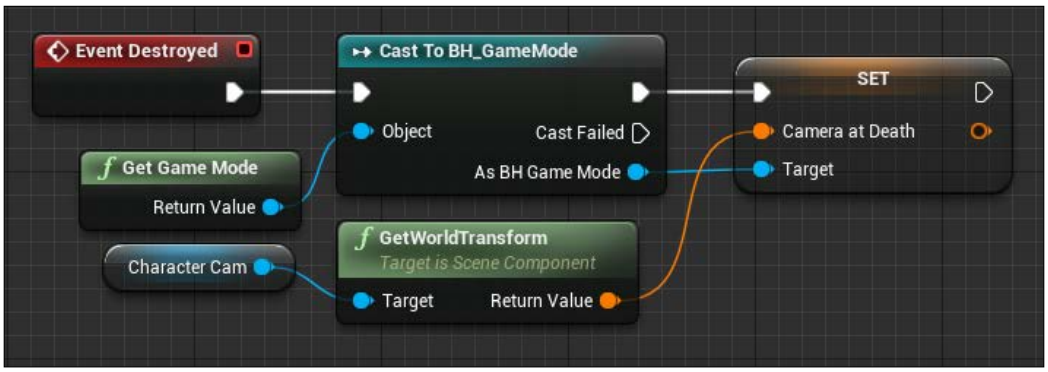
Creating custom cameras

We are going to be creating something called a camera actor then setting this to be our Target View. Camera actors are not cameras in themselves but are an in-world object that owns a camera component. Setting a camera actor as the Target View will use this camera component as the camera through which the player views the game world. You can use any actor that owns a camera component as a target for the view.

Where do we spawn the camera?

We are going to use the `SpawnActorFromClass` node to create our camera actor. However, just like in previous instances of using this node, we need to provide a *transform* for the actor to use at spawn. We need this transform to be that of the character's camera upon his untimely barrel-related demise. To do this we can create a transform variable in `BH_GameMode` called `CameraAtDeath` and ensure we set this from our `BH_Character` when the `OnDestroyed` event is hit.

Do this now. Summon the `Destroyed` event in `BH_Character`, use the `GetGameMode` function node to return a reference to the current game mode, cast this reference to `BH_GameMode`, then set the `CameraAtDeath` transform to that of the *player's camera* (reference name `CharacterCam`) via the `GetWorldTransform` function. The functionality should appear as shown in the `BH_Character` Blueprint:



Spawning the camera object

With the transform of the character's camera at death attained, navigate back to the `BH_GameMode` Blueprint, and we can now create our camera spawn functionality. Do this now by finding a `SpawnActorFromClass` node and setting the class to `CameraActor`, then transform to the `CameraAtDeath` variable and leave the rest as defaults. Be sure to connect this new node to the **Then0** execution output from the sequence node.

To save out our camera so we may have a reference to use in future, create a `CameraActor` reference variable in the **MyBlueprint** panel called `GameModeCamera`, and set the value of this reference to the output of the `SpawnActorFrom` Class node we just created. Before we do this however, we should be sure that we do not leave any floating camera objects around the world and clutter our memory. We need to first check whether there is already an existing game mode camera populating that reference. If there is, we need to destroy it, *then* assign the newly created camera.

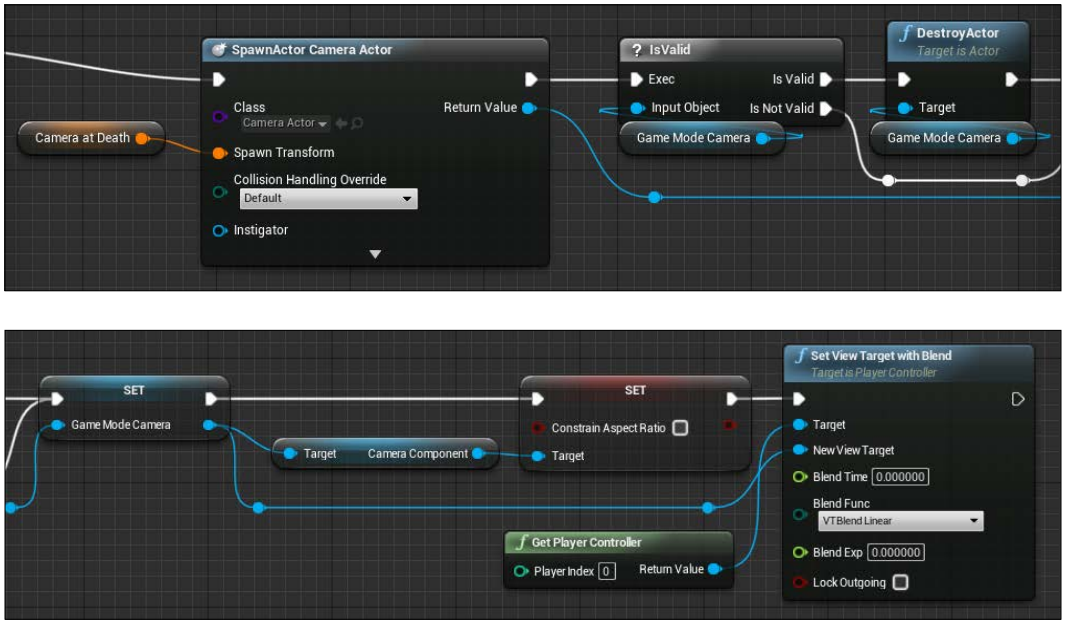
We can do this by using an `IsValid` node. This node will take in an object reference and output two execution paths, one if the reference is valid, and another if it is not. Spawn one of these nodes now and parse the `GameModeCamera` to the input labeled `InputObject`. From the **IsValid** path, spawn a `DestroyActor` node and parse the `GameModeCamera` reference to it. Then connect both the execution output of the `DestroyActor` node and the **IsNotValid** path to the **Set Node** for the `GameModeCamera` reference. This means that if there is a valid camera actor populating the `GameModeCamera` reference, then destroy that actor (thus freeing-up memory) and set the new camera actor, otherwise we avoid that functionality and assign the new camera actor to the reference.

Setting the Target View

To ensure that our new camera maintains consistent view properties with our previous camera, we are going to need to change one of the new camera's **default properties**. To do this, drag a line from our `GameModeCamera` reference and search for `GetCameraComponent`. Now we need to drag a line off the resultant camera component reference and set the `bool ConstrainAspectRatio` property to false. This will mean the new camera preserves the aspect ratio of the current viewport.

We need to set this camera actor as the **Target View**. Do this via the `SetViewTargetWithBlend` node. This node takes in a reference to the actor we would like to use as a target, a reference to the player controller who's view we will be changing, a blend time that will dictate how long the blend will take between views, a Blend function type, a Blend Exponential value, and a `bool` flag that will lock the last frame of the camera when blending.

Plug our new camera actor into the **NewViewTarget** input and use `GetPlayerController` with a player index of zero to get a `PlayerController` reference for the **target** input. We can leave the rest as default for now as we wish to have the view target swap happen instantaneously. You should see something that looks similar to this:



Respawning the player with a Blueprint function

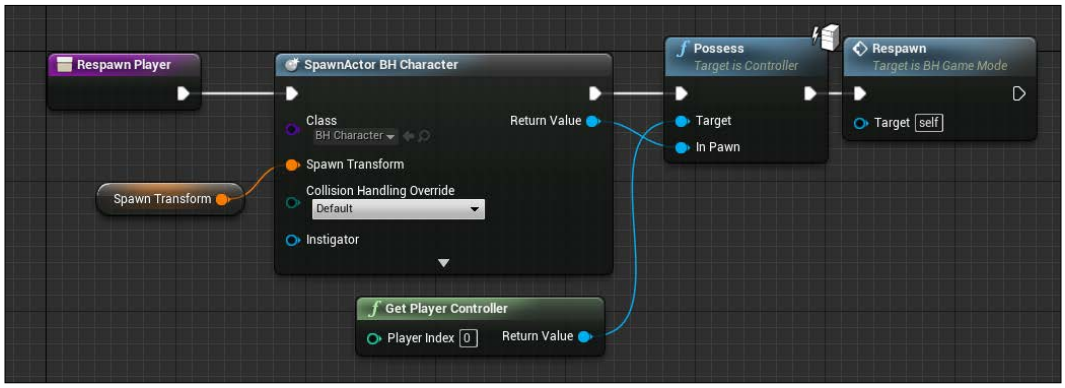
Now that we have our new view, we want the player to witness the aftermath of their death for a short time before respawning the character and setting the view back. The reason for this will become clear later in the chapter when we cover animation. The delay node we have created will do this for us. Feel free to adjust the respawn time, meaning the delay node length, to your liking. Immediately following the delay node, we have our previously created respawn functionality. We will be modifying this to include some new camera manipulation functionality so we can smoothly translate from the respawn camera back to the player camera.

Creating the Blueprint function

To keep the functionality following our sequence node tidy, we should create a **blueprint function** within which we can group the player's respawn functionality. Create the function now by pressing the small white plus next to the **Functions** category in the **MyBlueprint Panel** and title it `RespawnPlayer`. If you wish to rename the function later, simply right-click the function you wish to rename and select **rename** from the dropdown. With the new function selected, you will see some new fields in the **Details** panel on the right-hand side of the window.

It is in this panel where you can specify a description for the function, a category for the function to be grouped under, keywords to be used when searching for the function, a compact node title (similar to the AND and XOR nodes), an access specifier that will set the encapsulation level for the function, and a `bool` representing whether the function is pure (meaning it will not require an execution path, for example `GetPlayerController`). It is also where you will specify any inputs or outputs for the function node. We can leave these all as defaults for now. Open the new function by double-clicking `RespawnPalyer` in the **Details** panel.

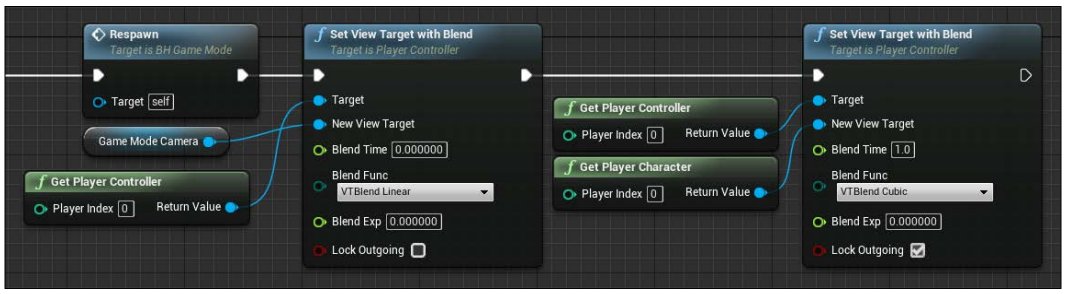
You will be presented with a new graph space that represents the *work area* for this function. Unlike the event graph, you will see only one entry point for the graph. The purple node titled **Respawn Player** will fire when the function is called. You will also notice you *cannot* summon event nodes within a function graph, you may call events thus triggering them, but you cannot *define* the functionality of an event node like you can from within an event graph. This is because function graphs must only contain functionality that specifically pertains to the function itself. What we are going to do is cut our player respawn functionality starting from the `SpawnActor BH_Character` node and ending at the `Respawn` event call node that we have in our event graph, and paste it into the `RespawnPlayer` function graph and reconnect the critical execution path. You should see something similar to this.



You may have noticed the strange icon located on the top right-hand corner of the Possess node. This simply means this function will only be executed on the **server authority**, something that we will cover later on in the book when we address networking.

The only issue with this current node set-up is that the moment we possess the newly spawned character, the target view will be overridden and the new character's camera will be used. This removes any possibility for us to do a nice blend between view targets. In our case we would like to have the target view *cubically interpolate* between the death camera and spawn camera views. To do this we will have to re-set our target view back to the `GameModeCamera` we made earlier, with a blend time of zero specified, then immediately set the target view back to our character camera again, however this time we specify a 1.0 second blend using `VTBlendCubic` specified for the `blend` function. This will produce a smooth transition effect from the camera we created when the player died, to the camera location when the player spawns.

To do this, summon a `SetViewTargetWithBlend` node and set the current `PlayerController` as the **Target**, the `GameModeCamera` as the **New View Target**, and leave the rest as defaults. Then summon another `SetViewTargetWithBlend` node, this time instead of using the `GameModeCamera` as our **New View Target** we are going to use the `Player Character` we just spawned! We can get a reference for this character by using the `GetPlayerCharacter` node and specifying an input index of zero. Because `UCharacter` inherits from `UActor` we are able to specify our new character as a valid input for the `SetViewTargetWithBlend` node. This time we will not be leaving the rest of the values as default. Specify 1.0 for the **Blend Time**, `VTBlendCubic` for the **Blend Func**, leave **Blend Exp** as zero, and check the **Lock Outgoing** `bool`. You should end up with a node arrangement that looks like this:



Awesome! Now the respawn functionality is complete. We will be making small edits to the `BH_GameMode` blueprint regarding respawning later on in the chapter. For now, we are going to be looking into Unreal Engine's animation systems.

Animation with UE4

Animation is a very effective way to establish the traits of a character. From the sleek movements of an assassin to the lumbering gait of a giant, the way the character moves and gestures is a core feature of that character's identity. UE4 offers a robust and easy-to-use workflow for animating characters. The following sections will illustrate how to set up a rudimentary animation implementation for the `BH_Character`. As it stands, our character is frozen in one pose. By the end of this section the `BH_Character` will be running, jumping, and falling in articulated motion.

Animation Conventions

Before we begin to break down the workflow of animation within UE4, it is important to establish a few animation conventions. These conventions are relevant among all animation workflows. In video games, animations are essentially a series of frames that are interpolated at real time to create the illusion of motion. Game developers have implemented multiple techniques to mask this interpolation so that there is no discernible skip between poses, leaving us with what appears to be smooth and natural motion.

If you think of a stop motion movie, animators will move figures within the scene a very small amount every frame to make it seem as though the characters are moving. A similar thing is happening during game animations. An animator will have created a series of animation **key frames** using a third-party tool. These frames will then be interpolated between at real time and a resultant pose will be calculated.

The way an animator articulates this motion is through the use of a skeleton. A skeleton is a collection of bones the animator can move to create the motion of a character. Each bone in a skeleton is bound to a section of geometry of the character mesh. When a certain bone moves, the bound geometry will also move. This means the animator can bring an otherwise static object to life. If the categories were to be broken down, it would look like this:

- **Animation**: A collection of frames that contain a character at varying stages of a given motion that will be played in sequence.
- **Skeleton**: The rig/frame the animator used to create the animation. This skeleton must be present in-engine for the animations to work as intended.
- **Frame**: A key frame or single frame of time within the animation.
- **Pose**: The resultant stance of the character given the amount of real time that has passed since the animation has begun play. Poses may be the result of two frames of animation that have been interpolated between to compensate for real-time precision, for example a pose between Frame 30 and Frame 31.

- **Blend**: A Blend is the transition between animations. You can swap from one animation to another via a Blend. The Blend will dictate how this transition takes place over time. You may want the transition to be instantaneous or you may want the character to blend between two animations smoothly over time.

Animation Blueprints

Much like our standard Blueprints, UE4 offers another workflow tool called Animation Blueprints. It is with these Animation Blueprints that we establish the state machine for the character's animation, set up our animation graph, and create an animation event graph. These core features are responsible for the following:

- **State Machine**: Animation state machines are a visual representation of the logic behind a character's animation state. It is from these state machines that you will determine what animation to play given the character's current physical state, for example is the character moving? Play the running animation. Is the character falling? Play the falling animation.
- **Animation Graph**: This animation graph utilizes the same node-based visual scripting system that is implemented in Blueprint. These graphs are responsible for the blending and merging of poses. A great example is blending the state machine pose for a character, such as the running or idle pose, with a hit reaction animation pose. The result will be the original running or idle pose influenced by the hit reaction pose, this will have that hit reaction animation look different for each character state, saving our animator a lot of time.
- **Event Graph**: Very similar to a Blueprint Event graph, this will be used to create blueprint functionality for animation-related events. This graph will have access to the owning character Blueprint. This means we can call functionality in our character blueprint based off of events that take place in our animations, and attain variable information from the character to dictate animation logics.

Importing and exporting animation assets

When creating animation flows for a character, you first need to obtain some animation assets. Loading and exporting animations into UE projects is very easy. Simply navigate to the animation asset that your artist has provided (`.fbx` file format is ideal) and either import this asset or click and drag from your file explorer into the content browser panel. During this process the animation asset creation wizard will open. This will ask you to choose a skeletal mesh to associate these animations with, plus some other import options. To export animations, simply right-click on the animation you wish to export within the UE content browser and hit export.

Thankfully the mannequin assets we borrowed included a set of animations that we can use to articulate our character. Navigate back to the content folder you migrated these assets to (**Content** | **Mannequin**) and open the animation folder. Inside you will find an animation blueprint, a **Blendspace**, and seven animation assets denoted by a green border. Select only the animation assets and move them to the `BH_Character` folder, we will be recreating our own versions of the animation Blueprint and Blendspace.

Creating your first animation Blueprint

When creating any animation asset within Unreal, the skeleton is the asset that creates a common association. This is important as our animation blueprint needs to know of this association so all of the assets we reference within the Blueprint are relevant to the target skeleton. Let's create the animation Blueprint for our character now by right-clicking in the content browser. Choose the parent class to be `AnimInstance` (this is the base class from which all animation blueprints inherit). For the target skeleton, select **UE4_Manniquin_Skeleton**. This will ensure that any assets that we are able to reference from within the animation blueprint also use the same skeleton. Name this new animation blueprint `BH_Character_AnimBP`.

You can right-click the skeleton asset itself and select an option from the create dropdown to automatically establish the skeleton association with the new asset.

Navigating animation Blueprints

Open this new animation blueprint; you will be presented with the **Animation Blueprint** window. There are a few new panels here that you will not have seen before, a new toolbar layout, and a new type of graph. The elements of this new window are detailed as follows:

1. **Toolbar**: Located at the top of the panel, this toolbar functions in a similar way to the toolbar featured in blueprints. There are some new buttons and features to the toolbar that will be detailed in the following.
2. **Viewport**: Located on the left-hand side of the window the viewport is a key feature of the window, it is through this Viewport that you can see the resultant output pose/animation with your given animation functionality. This allows you to see real-time previews of your animation logics.

3. **Anim Preview Editor**: Located below the viewport, this panel is where you will find any variables that have been used to influence the output of the anim graph. This means you can preview the resultant animation given a certain set of variable states.
4. **MyBlueprint**: Located in the bottom-center of the window, this panel acts in exactly the same way as the **MyBlueprint** panel featured in standard blueprints. It is here that you can find any technical features of the Animation Blueprint.
5. **Asset Browser**: The asset browser occupies the same space as the **MyBlueprint** panel and can be accessed via a tab by default. In the asset browser you will find any animation assets that are associated with the target skeleton of this Animation Blueprint.
6. **Details**: This time, the details are located in the bottom-right of the window. It is here that you will be able to change any exposed settings of the selected element.
7. **Anim Graph**: This is a new type of graph that is unique to Animation Blueprints and can be found in the center of the window. This graph is responsible for all of the animation flow logic. It is here that you will be performing animation blends and summoning various nodes that dictate how the character will animate.
8. **Event Graph**: Event graphs featured in Animation Blueprints allows you to create and receive *animation-specific events*. It is also here that you can update variables contained within the Animation Blueprint that may dictate the animation of the character.

The toolbar

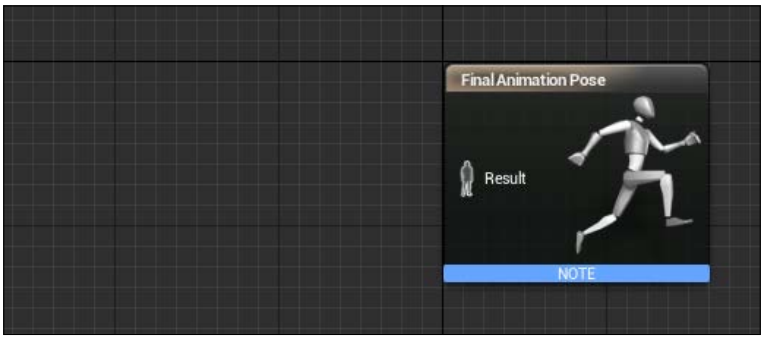
As stated previously, Animation Blueprints boast a few new buttons in the toolbar. These new buttons are specifically related to animation. The first are the options **Preview** and **RefPose**, which allow you to choose what you would like to occupy the Viewport. **Preview** will show you the output pose of your given animation functionality, **RefPose** will show you the base pose the animation will be blending from. The third option is titled **Record**. This allows you to record an animation sequence based on what is playing in your preview viewport. There is also a new navigation tool on the right-hand side of the toolbar panel that looks like this:

This section allows you to easily navigate between all of the assets that are relevant to this Animation Blueprint. They have been listed in a *hierarchical* order. The Skeleton is the base from which many animation assets will stem from. The Mesh is the **skeletal mesh** that you are using to preview the output from this Animation Blueprint. The Animation is the currently active animation asset associated with the given skeletal mesh, and the Graph is the Animation Blueprint itself.

The order of this hierarchy represents the polymorphic order of the animation pipeline. You can have *multiple* skeletal meshes that are based off of one skeleton. You can have multiple Animations based off of one skeletal mesh. This means that you can have one Animation Blueprint that can be used for multiple skeletal meshes and Animations, given that they use the same target skeleton.

Populating the Animation graph

Currently in our new `BH_Character_AnimBP` there will be no functionality present in the **anim graph** apart from the `FinalAnimationPose` node. This node takes in a new pin type that we haven't seen before. The pin is represented by the outline of a standing person. This pin is of type pose. Animation nodes will take in one or more poses and output a pose to then be fed into subsequent animation nodes. `FinalAnimationPose` itself is the end of the animation chain. The pose that is parsed into this node will be the pose that is used for this frame. Currently your **anim graph** will look like this:



What we need to do is create some functionality that outputs a pose to this node. We are going to be creating a **state machine**. As stated previously, a state machine will determine an output animation based on various state variables that are related to the character in some way. We need to create a state machine that will handle outputting the correct animation based on the character's current movement state.

To create a new state machine, right-click in the **anim graph** and search for **add new state machine**. Selecting this option will create a state machine node. With the node highlighted, you will notice that the Details panel in the bottom right-hand side of the window has updated to provide the appropriate fields. Rename this node to `Character_Locomotion` now, as this node will handle the animation based on the character's current locomotion. Drag from the output pose pin and connect it to the input pin on the `FinalAnimationPose`.

If you compile the blueprint, the compiler will output some warnings. They are all associated with the state machine currently being empty; we need to fill this state machine with relevant functionality. Double-click the state machine node now.

Working with state machines

With the `Character_Locomotion` state machine open you will now see another nearly blank graph with only a small node titled **Entry** in the center. State machine graphs are different to Blueprint graphs as the nodes that are created and the connections between them are represented differently. This is because, as opposed to the visual plotting of functionality, this is the visual plotting of **logic flow**.

State machine graphs are made up of **State Nodes** and **Transition Lines**. A State Node is a node the represents a logic state that can be assumed, they are usually responsible for playing an animation associated with the state. A Transition Line is an association between states that represents the ability for one state to transition to another based on a rule.

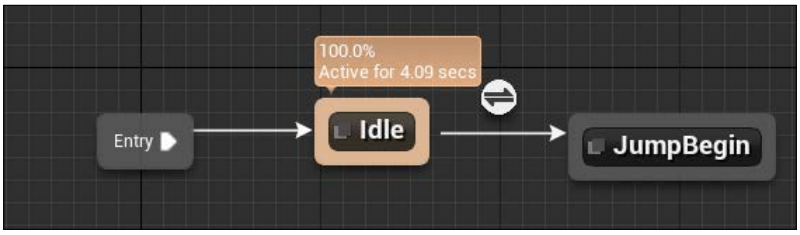
The `Entry` node present does exactly as is titled. This node will be the entry point when this state machine is queried for the output pose. What we need to do is create a new State Node and connect it to this entry node. The first state that we should create when working with a locomotion state machine is an Idle State. That is because, by default, the first state node that is connected to the entry node will be responsible for the default pose of that state machine. In other words, if none of the other transition rules resolve to be true, this state will be the one that is responsible for playing the animation. In the case of locomotion, if a character is doing nothing, we wish them to play an idle animation.

State nodes

To create new state nodes, simply right-click on the graph and select **Add State**. This will create a new state node titled `State` by default. If you select this state it will be bordered by an orange outline and the Details panel will update. If you look at the Details panel now you will see that you can rename the state node, change this now to `Idle`. You will also notice that, in the Details panel under the **Animation State** section, you can also specify event names that you wish to trigger when that state is entered. This is a very powerful tool as you can have other functionalities execute upon an animation state change. If you specify any non-null names in these fields, an `AnimNotify` event will be created with the entered name that can be summoned in the Animation Blueprints event graph.

Transition Rules

The next thing we need to do is create a transition between the entry node and the new `Idle` state node. Do this now by clicking and dragging from the small arrow present in the `Entry` node to the `Idle` node we just created. This will create a white arrow. As this is the transition from the entry node to this first state node, we will not be creating any transition rules, meaning this transition will take place every time. Usually transitions also include a rule that must evaluate to `true` before the transition takes place. Create another state node now and title it `JumpBegin`. Click and drag from the outside edge of the `Idle` node onto the `JumpBegin` node, another arrow will be created but there will also be a small white circle with a bi-directional arrow within. Your graph should look similar to this:



The small white circle represents a transition rule. Select this transition rule now so that it is highlighted, then address the Details panel. You will notice that this transition rule has many properties that can be changed to affect not only how the transition takes place, but also how the blend between the two state animations takes place. Double-click on the circle now. You will be presented with a new graph that has a single result node that takes in a `bool` titled `CanEnterTransition`. Simply, if the `bool` value that is parsed to this node resolves to `true`, the transition will take place. We want this value to be `true` when the character is jumping, and `false` when it is not.

We need to receive information from our character. We can do this from the Animation Blueprint Event Graph. Save this information in a relevant variable then check the state of this variable from within this transition. How do we do that? Member variables. Within the **MyBlueprint** panel, add a new `bool` variable and name it `IsInAir?`. Then drag a visual reference to this variable into the transition rule graph and plug it into the `CanEnterTransition` input pin. Alternatively, a much easier way to create variables on the fly is to drag a line from the `CanEnterTransition` input pin and then select **Promote To Variable** from the suggested list. This will create a new member variable automatically, saving the hassle of having to do it through the **My Blueprint Panel**. With the new variable created we can change the state of this `bool` value in the **event graph** for this AnimBP and it will affect this transition rule.

An important thing to note is the navigation bar at the top of the graph; it looks like this:

The purpose of this bar is to show you how nested you are within the animation graph. Currently we are looking at the `JumpBegin` state, which is in our `Character_Locomotion` state machine, which is in our `AnimGraph`! This is very useful as it allows us to easily transition back to any of the other graphs by clicking on the option in the navigational bar. There are also backwards and forwards arrows on the left-hand side of the bar.

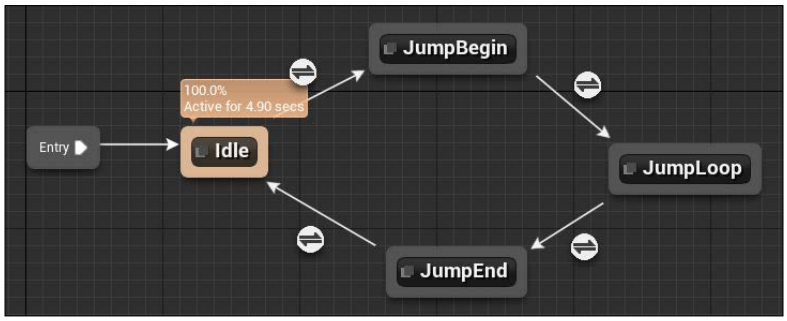
Playing animations from within states

Playing animations from within state nodes is very easy, simply double-click the state node and you will be presented with another `FinalAnimationPose` node. This node will output the final pose for this state to the state machine that will then be outputting that pose to the **anim graph**. Double-click the **Idle State** node now. The animation we want to play here is the `ThirdPersonIdle` animation. You can summon this animation into the graph by one of two ways. The first is by right-clicking in the graph and searching for **play ThirdPersonIdle**. The second is by looking in the asset browser for the correct animation then clicking and dragging the correct item into the graph. With the play `ThirdPersonIdle` node in the graph, click the output pose pin and drag it to the `FinalAnimationPose` input pin. Compile the blueprint now. With this connection made, you will see that the preview pose within the viewport has updated to the idle animation! You will also notice in the **AnimPreviewEditor** panel we can now see the `IsInAir?` variable we created earlier. We can now check and uncheck this variable to preview how our state machine will function.

Finishing our state machine

To finish our state machine, we simply need to add two more states and three more transitions. The main purpose of these states is to complete our jump loop. We have our jump loop in three different animations: we have a jump begin animation, a jump in air loop animation, and a jump land animation. The reason for this is we do not know how long the character will be in the air for, therefore the number of times in the air loop plays depends on the amount of time the character is in the air while jumping. Once the character has landed again we can then finish our jump loop and play the jump end animation.

Add two new states to our machine now, call them `JumpLoop` and `JumpEnd`. Also create transitions between `JumpBegin` and `JumpLoop`, `JumpLoop` and `JumpEnd`, and then `JumpEnd` and `Idle`. This has now created our closed state machine loop. If the character is not jumping it will be in an idle pose. Your state machine should look similar to this:



Now that we have the shell of our state machine, let's begin filling our states with the appropriate animations and our transitions with the appropriate conditional checks. Start by adding and connecting the play `ThirdPersonJump_Start` node to the `JumpBegin` state graph. The rule we will create between `JumpBegin` and `Jumploop` will be using information that utilizes the current animation that is playing. What we will be doing is checking whether the `ThirdPersonJump_Start` animation is 80% of the way through playing. If it is, we wish to transition to the `JumpLoop` state, initiating a blend between the jump begin and the jump loop animations.

Double-click the transition node between `JumpBegin` and `JumpLoop`. Within this transition graph we can utilize the context sensitive search to find the function we need. Right-click inside the graph and search for the function **Time Remaining (ratio)** (**ThirdPersonJump_Start**). You can simply search for the keyword ratio to find this node. As we are in the transition between a state that plays the `ThirdPersonJump_Start` and another, the ratio function found will automatically observe the appropriate animation.

This node will return the time remaining in the animation as a fraction. This value will be of type float. Take the result of this node and then search for a float less than float node (`<`). Plug this value into the top input then in the bottom input type `0.2`. This less than node will return `true` when the animation has less than 20% of the animation play time remaining. Your node arrangement should appear as follows:

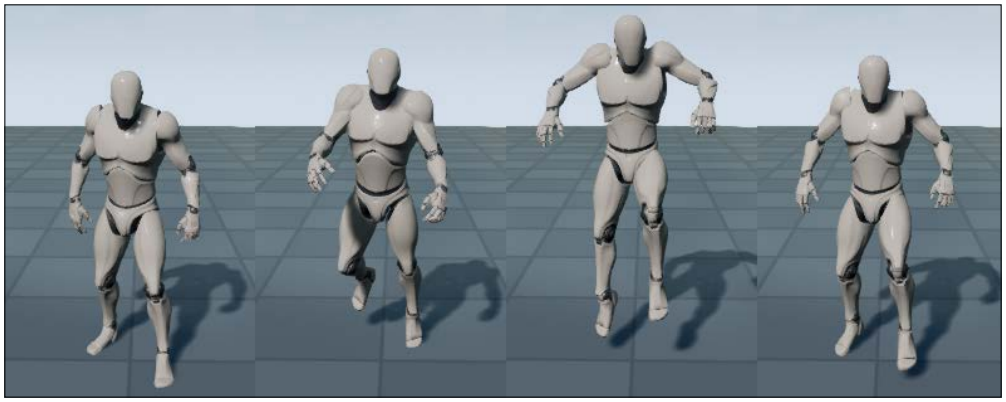
The next step is to play the jump loop animation from within the `JumpLoop` state. Do this now by adding and connecting the `Play ThridPersonJump_Loop` node to the state graph. Now go back out into the state machine graph and under the **AnimPreviewEditor**, tick the **IsInAir?** checkbox. You should see the character transition from an idle pose to the jump begin animation, then to the jump loop!

There is a small problem though, the transition between the jump begin and jump loop animations looks a little janky. The issue here is the jump begin animation is **looping** and starting the second loop before the transition blend begins with the jump loop animation. Even though we specified that the animation should transition before the animation ends, when you specify an animation to loop (which is set to `true` by default), the engine will blend the last few frames of the animation with the first few frames of the animation to create a smooth loop.

To fix this, open the `JumpBegin` state and select the `Play ThirdPersonJump_Start` node. Selecting this node will populate the Details panel with properties that are associated with this animation node. Under the settings category there is a **Loop Animation** checkbox. Uncheck this now and re-preview our animation sequence by checking the **IsInAir?** variable, you will see the transition is now much smoother! However, when you uncheck the variable, nothing happens, the character remains in the jump loop. That is because we have yet to specify our `JumpLoop` to `JumpEnd` transition rule.

Do this now by opening the transition between the two states. This time, instead of checking the `IsInAir?` variable directly, parse the value of this variable to a `NOT` node. This node will return `true` when the provided input is false. Plug the output from this `NOT` node into the `CanEnterTransition` input pin. You also need to provide the `JumpEnd` state the `play ThirdPersonJump_End` node, do this now. Also ensure that the `play ThirdPersonJump_End` node has the loop animation property (found in the Details panel) unchecked.

The last thing you need to do is specify the transition between the `JumpEnd` state and the Idle state. Use the same ratio node transition rule we used when specifying the transition rule between `JumpBegin` and `JumpLoop`. Now compile the Animation Blueprint and test our state machine using the **IsInAir?** checkbox in the **AnimPreviewEditor** panel. You should see a transition similar to this:



The Animation Blueprint Event graph

Despite our preview viewport showing a correct sequence, if we are to run the game, our character will still not animate at all. There are two reasons for this. This first is that we have not assigned the `IsInAir?` variable to anything relevant yet, the other is that we have not informed our `BH_Character` to use the `BH_Character_AnimBP`. We can do this now by opening the `BH_Character` blueprint, selecting the `Mesh` component from within the **Components** panel, and changing the **Anim Blueprint Generate Class** field under the **Animation** section of the Details panel to `BH_Character_AnimBP`.

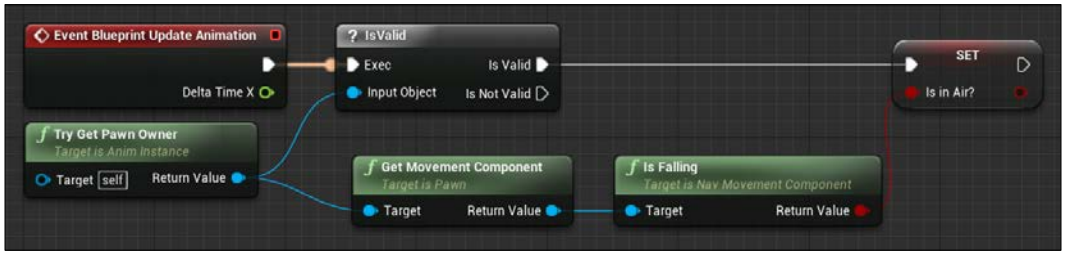
Now that we have associated our character with our custom Animation Blueprint, we can retrieve data from the owning character in the Animation Blueprint event graph and set our `IsInAir?` variable. Navigate back to the `BH_Character_AnimBP` and open the **EventGraph**. By default, you should see two translucent nodes, one is Event Blueprint Update animation. This event will be called every frame (similar to Event Tick in standard blueprints). This node will also provide the current delta time (time since last frame).

The other node is a pure node called `TryGetPawnOwner`. This will try and get the pawn that owns this animation instance. As our `BH_Character` inherits from `UCharacter`, which in turn inherits from `UPawn`, and our `BH_Character_AnimBP` inherits from `anim instance`, this node will work perfectly, we will just have to do some casting if we want access to `BH_Character`-specific functionality.

The first thing we are going to do is ensure that our `TryGetPawnOwner` node returns a valid reference. This is similar to checking the validity of pointers in C++. We can do this easily by finding the node `IsValid`. This node takes in a reference to a `UObject`, if the reference is valid one execution path will be fired, if it is not, the other will. This is a very powerful tool as it allows us to ensure that we only carry out object-dependent functionality if that object is valid.

Click and drag from the `TryGetPawnOwner` output pin and search for `IsValid?`. This will summon the node and automatically assign the output reference to the input of the new node. From the `IsValid` path summon a set node for our `IsInAir?` variable. We only want to do this if our `Pawn` reference is valid, otherwise we will try and access data that has yet to be created and we will either see an in-game crash or runtime error.

As the `UPawn` class is responsible for the movement objects of any given pawn or pawn child class, we don't require a cast of this reference. From the `TryGetPawnOwner` reference, click and drag a line and search for the function `GetMovementComponent`. This will return the movement component that is associated with the owning pawn. As you have seen with our `BH_Character`, the movement component is responsible for all movement variables and parameters. We can query whether the character is currently falling by dragging a reference from the `GetMovementComponent` output pin and searching for the function `IsFalling`. This function will return a `bool` representing the falling state of the character. Set our `IsInAir?` value to the output `bool`. Your graph should look similar to this:



Now we can run our project and our character will idle and jump accordingly!

Getting our character running

Despite being able to stand still and jump properly, our character is currently having a hard time running. When we move across our map, he simply slides. What we want to do is have our character run instead. Not only do we want the character to run but we also want the character to transition from standing to walking to running, smoothly. One of the best ways to do this is by using a **BlendSpace**.

A Blendspace allows you to set up a graph of animations you wish to transition between. The axis of this graph will represent a value that you provide to the Blendspace. When plotting the animations along the axis, you are saying that, when the provided value is X, I would like the animation at position X to be of full weighting, meaning that you want that animation to entirely dictate the motion of the character's mesh. This also means that you can have an animation A at value X on the graph and an animation B at value Y on the graph. Let's say that X is 100 and Y is 200. If the value you provide to the Blendspace is at 150, the output pose will be influenced 50% by animation A and 50% by animation B.

This is perfect for our running example. We use the character's speed as the value we will be parsing to the Blendspace. We can have an idle animation at value 0 (we won't be moving in this case), a walk animation at value 90, and a run animation at value 375. This means that, as our character increases in speed, the weighting of the output pose will transition smoothly between these three animations, resulting in a gradual change of the stance and running intensity of the character.

Working with Blendspaces

Blendspaces are another animation asset, however unlike Animation Blueprints, they themselves are classified as animations, meaning that you will be introduced to a new workspace, The animation workspace. As we are only concerned with one axis (speed), we need to create a one-dimensional Blendspace. Do this now by right-clicking in the content browser and creating a **Blendspace 1D** object, it can be found under the animation field. Name this `BH_Idle_Run_BS1D`. Double-click the Blendspace now to open it.

You will be presented with the animation window. There are a few new panels to work with here, they are detailed as follows:

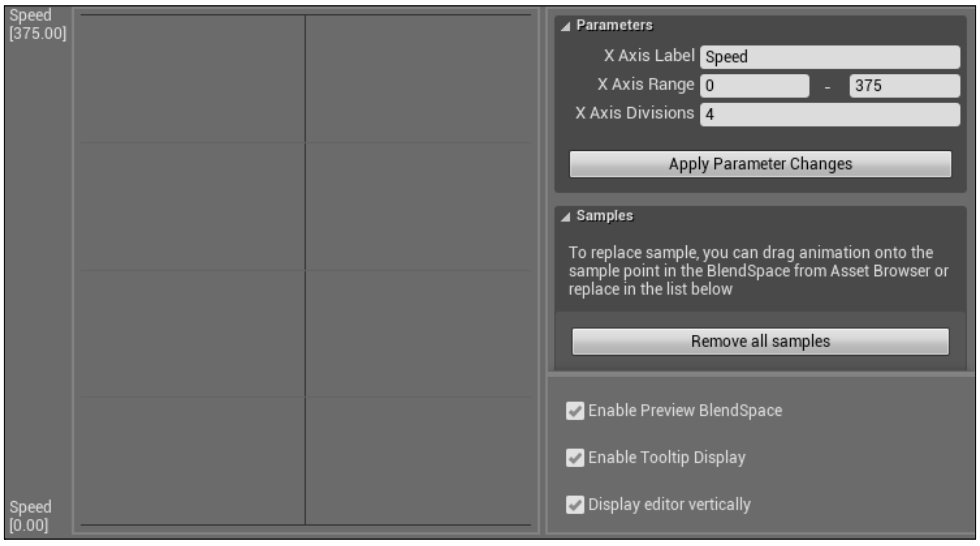
1. **Skeleton Tree**: Located in the top left-hand corner of the window, this panel shows you the current skeleton hierarchy of the animation asset you are working with.
2. **Anim Asset Details**: Located underneath the Skeleton Tree panel. This panel is similar to the details panel in Blueprints and allows you to modify specific parameters of the current animation asset you are working with.
3. **BH_Idle_Run_BS (Blendspace name)**: Located in the bottom-center of the window this area shows the workspace for the Blendspace itself. It is here where you will be establishing your axis values and plotting your animation points. Note that the name for this section will change depending on the animation asset.

4. **Asset Browser**: Located in the bottom-right corner of the window this panel shows you the animation assets associated with the same skeleton that you can reference in this asset.
5. **Details Panel**: Though unpopulated at the moment, this panel is used predominantly when setting up `anim` notifies in standard animation assets. This will be covered later in the chapter.
6. **Viewport**: This panel shows the current output of the animation asset and some other statistical information about the skeletal mesh.

The toolbar at the top of the window has also changed slightly. You can now import and export animations directly from the toolbar. You can also create another animation assets from here.

Creating the running Blendspace

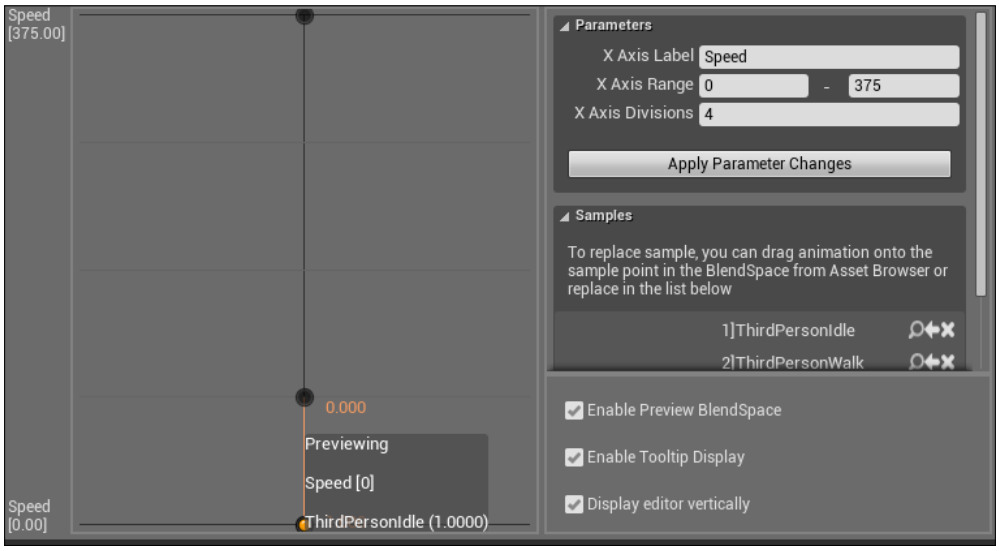
The first thing we need to do is set up our workspace. Under the **BH_Idle_Run_BS** panel there is a checkbox titled **display editor vertically**. Check this now if you wish. I prefer to have the display appear this way so I can maximize visible information in the panel. The next thing we need to do is set up our axis. Right now our single axis has no label and the range is from 0 to 100. We need to rescale our axis so that it goes from 0 to 370 and label it **Speed**. We can keep the number of divisions the same as we wish to have our walk animation play when our character is moving at roughly ¼ of its maximum speed. Do this now by filling in the corresponding fields within the panel then press the apply parameter changes button. Your parameters field should look like this:



Now that our axis has been set up, the next thing we need to do is provide the graph with animations. You can only plot animations at division points. This is so that the Blendspace can correctly interpolate between animations based on the current value and the difference between divisions. Click and drag the `ThirdPersonIdle` animation from the **Asset Browser** onto the bottom of the graph. You should be presented with a small node on the axis at the very bottom of the graph. This means that when our speed value is zero, the idle animation will play.

Do the same for the `ThirdPersonWalk` animation but drop this animation node at the first division. You can preview the output animation of a Blendspace by clicking in the Blendspace graph and hovering your mouse over the axis between these two nodes. This will use the value your mouse is currently at on the axis for the Blendspace weighting.

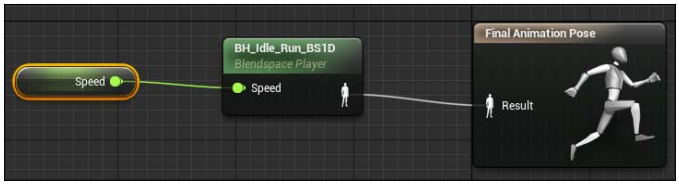
The next thing we need to do is drag and drop the `ThirdPersonRun` animation onto the top of the Blendspace graph. This means that at speed 0, the character will idle, at speed ~90, the character will walk, and at speed 375, the character will run. The Blendspace itself will handle the interpolation between these animations when the speed value deviates from these axis points. Your final Blendspace graph should look like this:



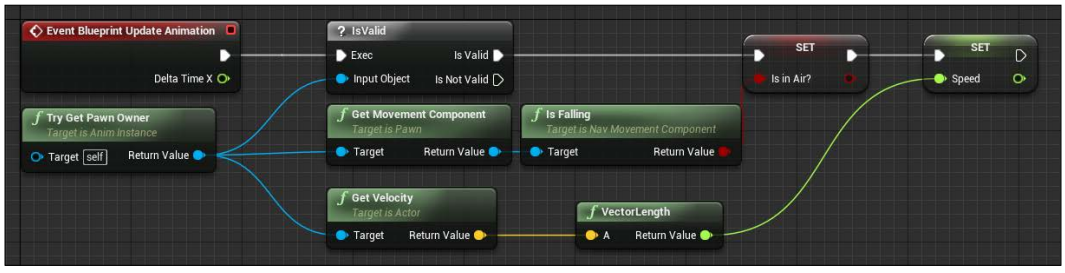
Utilizing the running Blendspace

Now, where do we call this Blendspace so we can get its output pose to influence our character animation? We are going to replace the play Idle animation in our idle state of the locomotion state machine with this Blendspace! This is a great example of combining animation assets within the UE4 animation pipeline to achieve polished results. We can use the state machine to dictate which animation asset we will be utilizing, then use the specific functionality of the Blendspace1D to add another level of polish to our character movement.

Open the `BH_Character_AnimBP` now and navigate to the `Idle` state node. Delete the play animation node that is there currently and search for the `BH_Idle_Run_BS1D` node, (keyword Blendspace). This will summon a node that takes in a float value representing the speed value we have specified in our Blendspace axis, and outputs a pose. Plug the pose output into the `FinalAnimationPose` node of this state graph. We do not currently have a value we can plug into the speed input pin. However do the same thing we did with our `IsInAir?` variable and create a member variable of type float called `Speed`, we will later assign a value to this variable in the Anim Blueprint Event Graph. Create this member variable now and assign it to the input pin of the Blendspace node. Your functionality should appear as follows:



Navigate back to the Animation Blueprint Event Graph. Drag another line off the `TryGetPawnOwner` node and search for a function called **GetVelocity**. This will return the Velocity Vector of the player. From the output of this node, search for a function called `VectorLength`. This will output the length of the vector as a float. As the character's velocity vector is un-normalized, it means we can use the length of this vector to get the speed of the character in cm/s. Summon a set node and plug this float value into the input pin. Your event graph should appear as follows:



Now, when you run the project, our character will animate when we jump and run! Great work!

Creating your first sound scape

Sound in games is one of the most important tools that you can leverage to enhance the immersion and enjoyment a player has with your title. UE offers a very easy-to-use sound toolset that we are going to be learning how to utilize. We will be learning how to create 3D sound queues, how to trigger sound effects based off of animation `ques`, and how to add global sound volumes to your game scenes.

Importing sounds and sound cues

The first thing we need to do is import the sounds that we will be using to create our ques. All raw sound assets must be added to your content folder then used to create a `Sound Cue` object. Sound cues allow you to play sounds from within Blueprints, codebase, and animations at runtime. These objects also allow you to apply any modifications to how our raw sound file is played back during runtime. This means we can perform sound modifications such as modulation, oscillation, attenuation, Doppler effects, and much more from within the engine.

We are going to utilize these sound cues and other UE4 sound functionalities to add explosions to our barrels, footsteps to our character, background noise to our scene, and a groan for when our character kicks the bucket. Import the sounds `deathsound.wav` and `footstep.wav` provided with this textbook (ensure to create the appropriate organizational folders to contain the new sounds). Then create two sound `ques` objects titled `BH_Footstep` and `BH_Death`. We can create sound `ques` by right-clicking in the content browser and selecting **Sounds** | **Sound Cue**.

Working with sound cues

Like many of the UE objects, sound cues feature a graph-like workspace that you can drag and drop nodes into, which will be linked via some path. In a similar way that animation graphs have an output pose node, sound cues have an output sound node that looks like this:

Unlike most graphs you encounter when working with Unreal, there will only be one type of pin. This means that all nodes that you work with will take in the same type for input and output. In this case the pins are of type *sound*. While working with sound cues you will pass this `sound` reference through nodes that perform the sound modification mentioned previously.

Open the `BH_Footstep` sound cue now. You will be presented with an empty graph featuring the previous node. You will notice that the layout for this workspace is much simpler than what we have encountered previously. You have a **Viewport** panel that shows you the sound `ques` workspace. A **Details** panel on the left-hand side of the window shows you any properties of the currently selected node. A **Palette** panel on the right-hand side of the window acts as a search directory for sound cue modification nodes.

Again you are presented with a **Toolbar** at the top of the window that provides you with Sound Cue-specific functionality. This time you are presented with three new buttons, **Play Cue**, **Play Node**, and **Stop**. **Play Cue** will play the cue in its entirety, taking into account the entire sound modification chain up to and including the output node. The **Play Node** button will allow you to play the sound modification chain up to a specific node. This is a very powerful debugging tool as you will be able to test how a sound plays before being parsed into the next modification node.

Sound modification

We need to summon a node that allows us to play the `footstep.wav` sound file we imported earlier. This is very similar to the way we played animations from our state machine earlier in the chapter. Right-click in the graph somewhere and search for a node titled `Wave Player`. This is the node that will play the raw sound file specified in the **Details** panel. Select this new `Wave Player` node and address the **Details** panel, change the **Sound Wave** element to **footstep** by selecting/searching for the appropriate asset via the dropdown menu provided. Plug the output of this node into the output node and press the **Play Cue** button. You will hear a sound similar to that of someone walking on the ground. Now if you press that play cue button multiple times you will notice that the sound always plays the same way. As we are creating a footstep sound cue, we need this sound to vary slightly each time it is played otherwise the player may find the sound to be monotonous and repetitive.

We can now utilize one of the aforementioned sound modification nodes. From the **Pallet** on the right-hand side of the window, find the **Modulator** node. This node will take a sound input then randomly vary the pitch and volume of this sound so the output of this node sounds slightly different every time it is played. How much modulation takes place is dependent on the ranges specified in the **Details** panel of the modulator node. I used 1.0 as a minimum and 1.5 as a maximum for both pitch and volume.

Ensure that all of the connections between nodes have been made, and repeatedly press the **Play Cue** button. You will notice that the sound varies slightly each time it is played. If this is not immediately apparent, select the wave player node, then use the **Play Node** button instead to hear the sound again without modification. The difference will be immediately noticeable.

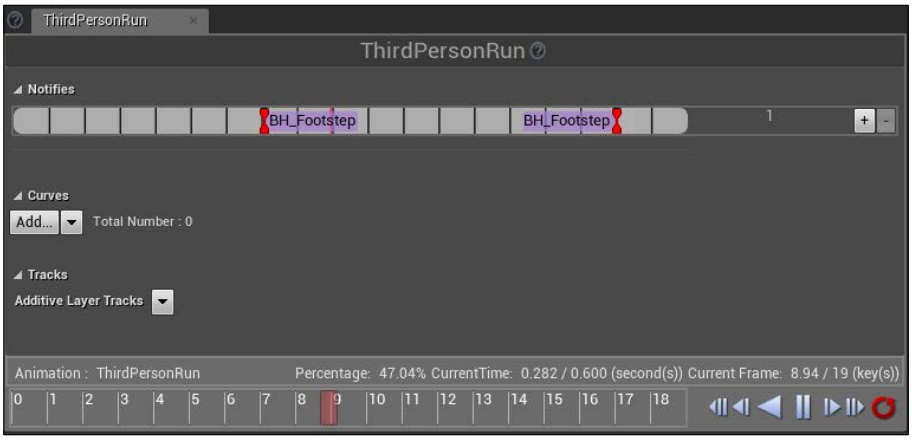
Our death sound cue can remain very simple. No modification is required as the sound will be played only at the point of the player's death, which will happen far less frequently than the player's footsteps. Simply summon the wave player node, select the **death.wav** asset via the **Details** panel, and parse the output to the Output node.

Playing sounds via animation notifications

Now that we have created our sound cues we need to play them at the correct time. For our footstep sound we are going to use an *animation notify* to determine when the sound is played. This means we can perfectly sync the sound cue to the foot striking the floor in the animation. To do this we are going to need to add a *play sound notify* to the animation asset that we are using for our run. We are going to have to add a similar notify to our walk animation as well, as both feature in the **Blendspace** we are using to drive the locomotion of the character.

Working with animation assets

Open the **ThirdPersonRun** animation, you will be presented with a window arrangement very similar to that of the Blendspace we made earlier. The main difference is the information that is presented in the **ThirdPersonRun** tab in the middle-bottom of the screen. You will notice that there is a new set of information being presented here and it looks like this:



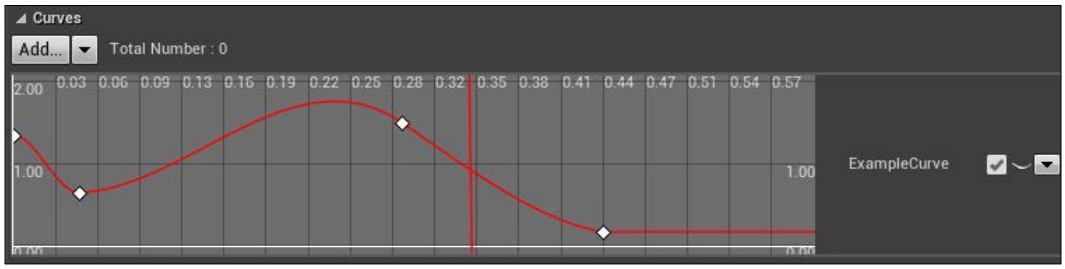
Notifies

The section at the top of the panel titled **Notifies** is where you can place notify markers along a time bar. These markers will enact some form of functionality when the animation reaches the specified point along the presented bar during runtime. The vertical lines on the bar represent the key frames of the animation. While the animation is playing, a small red line indicates the current play position of whatever animation is being previewed. This means you can play or scan the animation until a desired visual pose is seen (in our case, the moment the character's foot hits the floor), then address the **notifies** bar for the position of the red line, this is where you should place the notification.

Animation Notifies can be of three different types; play sound notify, play particle effect notify, or custom notify, which will be associated with a similarly named event that can be summoned in the event graph of an Animation Blueprint. The two notifies seen in the previous example are play sound notifies titled by the sound cue that is used at that notify point.

Curves

The next section titled **Curves** allows you to create something called a data curve. By default, the curves value progresses as the time of the animation progresses. For example, address the following image:



The previous curve shows that, as the animation plays, the value of `ExampleCurve` will vary between roughly 1.6 and 0.3 with a maximum value of about 1.9. The vertical red line represents the position at which time point the preview animation has reached. If we were to retrieve the value of this curve at this point, it would return around .99. These curves can be useful to generate meaningful data based on the progression of an animation. A great example would be if there were a *sweet spot* within a combat animation. You could use these curves to affect the damage amount that is received by an enemy. As the animation approaches the *sweet spot* in time, the curve's value would increase, then decrease as the animation progresses past this *sweet spot*. You could then use this value as a scalar when calculating attack damage.

Additive layers

The next section **Additive tracks** allow you to modify the current animation from inside UE as part of an **Additive Layer**. This means that you can take the existing animation and use an additive layer to produce an adjusted version of the animation. You can also create a separate, new animation with its own motion that is based on an original, utilizing the record animation button in the **Toolbar**.

Implementation of these additive layers is not within the scope of this book; however you can address the online documentation at `https://docs.unrealengine.com/latest/INT/Engine/Animation/AnimHowTo/LayerEditing/index.html` for more information.

Statistics bar

At the very bottom of the animation panel is a statics bar that shows you in-depth metrics for the given animation. This panel can be very useful to ascertain animation data at a given point so you may record the metrics to be used in logics. This bar also boasts the buttons that you can use to navigate the animation that you are working with. You can play in forward time and in reverse. You can also traverse the animation frame by frame via the buttons immediately next to the play/reverse buttons. There is also a red loop button that lets you turn looping on and off within the preview. These are preview controls and will have no effect on the animation during runtime.

Placing the animation notifies

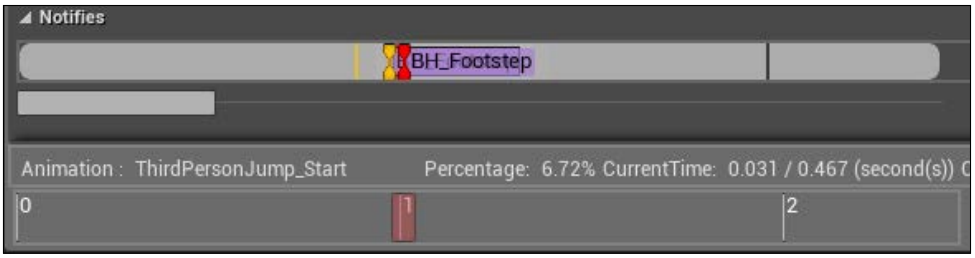
We need to add two notifies to your current **Notify** bar for the **ThirdPersonRun** animation. To do this we are going to navigate the animation using the frame-by-frame buttons at the bottom of the panel. We need to find the frames that each foot falls on the floor, it is at these key frames that we will be placing the play sound notifies. Navigate to Frame 6 of the animation by either scrubbing the red bar along the animation timeline or by pressing the **To Next** button in the controls.

You should see this in the preview window:

Now address the Notifies bar, you will see the vertical red line is overlapping one of the black dividing lines. Right-click on the red line now and select **PlaySound** from the **AddNotify** dropdown. This will add a new notify to the track titled `PlaySound`. With this notify selected (the notify marker will go from red to yellow), address the **Details** panel in the top right-hand side of the window. Here you can choose whether you want to modify the pitch or volume of the sound, whether you wish the sound to follow the parent, namely have the sound follow the character as it moves, and whether you would like to attach the sound emitter to a bone.

Select the Sound dropdown and search for the **BH_Footstep** cue we created earlier, then set the **volume multiplier** to 0.1 as we do not wish for the footsteps to dominate the sound scape. Then set the **pitch multiplier** to 1.8 as we want the footfalls to sound light. Create the exact same notify at key frame 16 for the other foot. When you play the animation within the preview panel you will hear sounds playing as the player's feet hit the ground! As we also use the **ThirdPersonWalk** animation in our running Blendspace, we need to add the same notification to the footfalls of the walk animation. They key frames are 8 and 24.

You may also add the footstep sounds to the **ThirdPersonJump_Start** animation at frame 1. To make the effect more prominent and punchier, you can set the Volume multiplier to 0.5 and have two notifies staggered directly after each other as seen in the following:

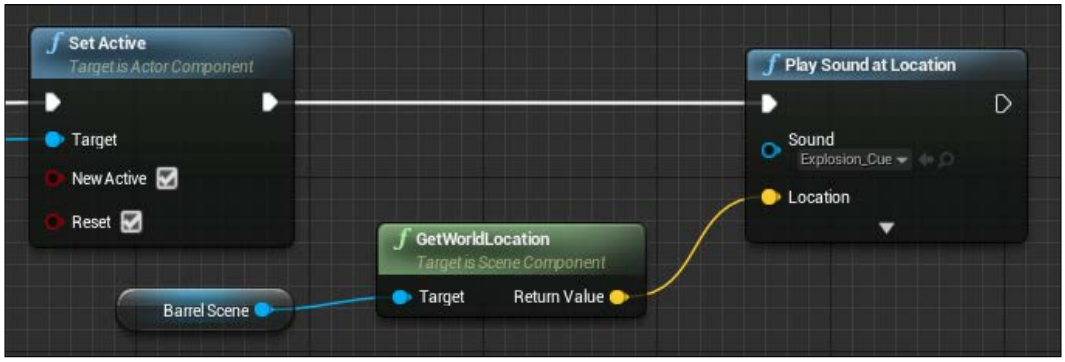


Finishing our soundscape

We need to do a few more things before the soundscape of Barrel Hopper is finished. We need to have the barrels play an explosion noise when they detonate, we need to have the player groan when they die, and we need to have some music looping in the background. These are all fairly easy to establish and will take simple world objects or calls to the sound queues.

Exploding barrels

Thanks to the starter content being included in our blank project, we have access to a couple of the Unreal provided assets, one them being an explosion sound cue. We simply have to play this sound cue when the barrel object receives its `OnDestroyed` event. Navigate to the `BH_Barrel` Blueprint. We are going to append to the functionality following the `Event Destroyed` node. From the `Set Active` node, drag a new execution path and search for **Play Sound at Location**. This node takes in a vector location and a reference to the cue we wish to use. For the **Sound** parameter, search in the dropdown menu for **Explosion Cue**, and for the **Location** parameter use `GetWorldLocation` with the `BarrelScene` component as the target. Your new functionality should appear as follows:



This functionality will play the `Explosion_Cue` at an emitter positioned at the same place as the `BarrelScene` component.

Players death rattle

Now we need to create similar functionality for our `BH_Death` cue from within our `BH_Character` Blueprint. Navigate to the blueprint and find the `OnDestroyed` event. Append a `PlaySoundAtLocation` node to the existing functionality. For the **sound** input, search for `BH_Death` from the dropdown menu, and for **Location** input use the `GetActorLocation` function with self as the target. This will mean that, when the play is destroyed, the death sound cue we created earlier will play.

Looping level sound

Creating persistent looping level sound is much easier to implement than it is for your sound designer to create non-pervasive background sound. All we need to do is create an **Ambient Sound** object, place it somewhere in the scene, and inform the object which que to play. We can do this by searching for the **Ambient Sound** object in the **Modes Panel** of the **Editor** window. Click and drag this object into the scene and, with it selected, address the **Details Panel**. Here you can change **Sound** parameter under the **Sound** section to `Start_Music_Cue`. This will use the ambient sound track that Unreal uses for the starter content level.

Now play the game, interact with the world, and behold the glorious soundscape we just created.

Adding the finishing touches to Barrel Hopper

Now that we have explored the basics of animation and sound with UE4, we can finish polishing our Barrel Hopper project. We are going to be adding a session timer to the HUD, a gameover screen, and ragdoll physics to our character so that when the player runs into a barrel, the character will go limp and hilarity will ensue.

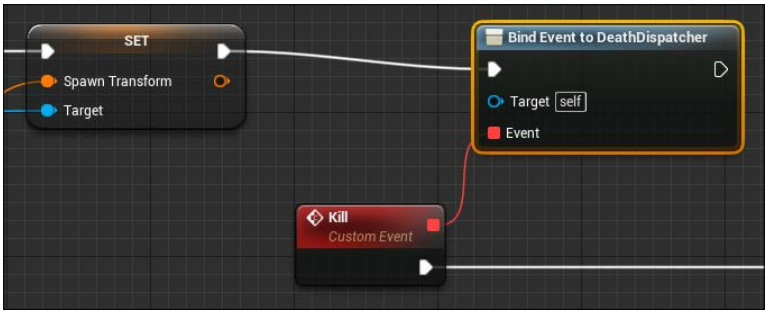
Ragdolls and Event dispatchers

Let's start by adding ragdoll physics to the character upon death. The reason we are starting here is to implement the ragdoll. We are also going to have to implement an event dispatcher. An event dispatcher allows us to bind a collection of events that will trigger when the single event dispatcher is executed. This means we can execute all bound functionality across multiple blueprints from one call to the event dispatcher. This is similar to the **Event Delegate** we currently use in the `BH_GameMode` but allows us to bind events to a dispatcher that has no explicit functionality of its own.

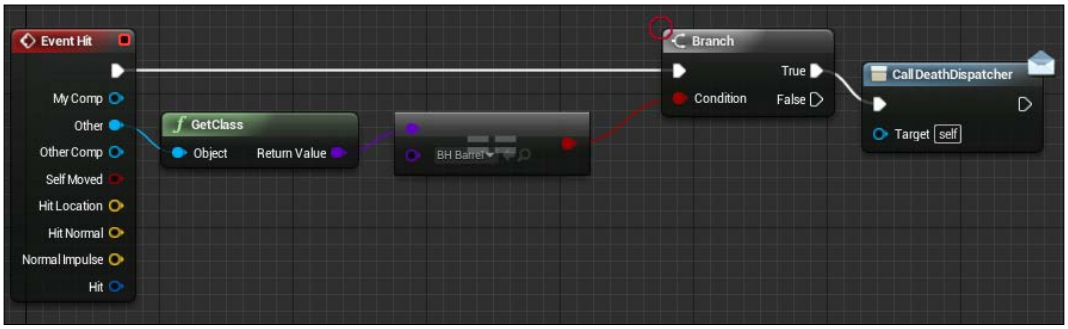
We are also going to have to create a custom event in the `BH_Character` to bind to this **Event Dispatcher**, which will replace the functionality following `Event Destroyed`. We cannot use `OnDestroy` as the moment we call the `Destroy` function, the mesh will disappear and we will not be able to see the character ragdoll. Instead we will use this new custom event and a delay node to destroy the character.

Create an event dispatcher now by adding one to the **MyBlueprint Panel** of the `BH_Character` blueprint. Call this dispatcher **DeathDispatcher**. Then create a new custom event in the `BH_Character` event graph titled **Kill**. We are going to be transferring all of the `Event Destroyed` functionality to the new `Kill` event. We then have to bind this event to our new `DeathDispatcher`. We will do this in exactly the same way as we set up our delegate association in *Chapter 2*, *Blueprints and Barrels – Your First Game*. Be sure to delete the old Event Destroyed node and the subsequent functionality we just copied.

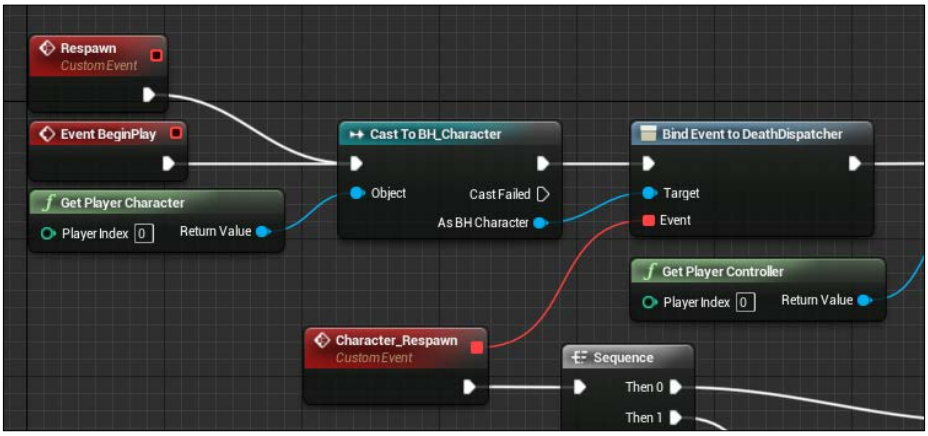
Navigate to the `BeginPlay` event node within the `BH_Character` Blueprint and you will see our old functionality that we created to set the spawn transform of the player. Append this functionality with a `BindEventToDeathDispatcher` node. This node can be found using the **pallet** or the context sensitive search. Now bind this to the `Kill` event by joining the red square pin in the top right-hand corner of the `Kill` event node to the red square pin in the bottom left-hand side of the `Bind Event to DeathDispatcher` node. Your finished arrangement should appear as follows:



Now we need to also make sure that we bind the correct event in the `BH_GameMode` and that we call the correct dispatcher when our player hits a barrel. Do this now by searching for the `Call DeathDispatcher` node and replacing the previous `Destroy` call that we had at the end of the `Event Hit` functionality in our `BH_Character` event graph. Then in our `BH_GameMode`, replace `BindEventToOnDestroy` with `BindEventToDeathDispatcher`. You will see these changes in the `BH_Character` Blueprint:

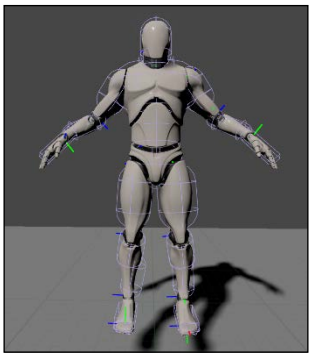


And this in the `BH_GameMode` Blueprint:



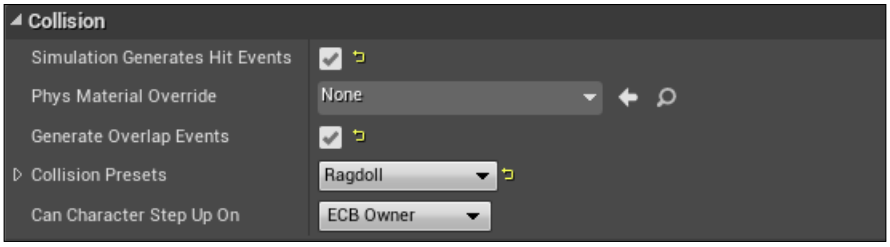
Now run the game and ensure that nothing has changed and our character dies and respawns as before. With this change in place we are now free to add our ragdoll functionality to the `Kill` event without worrying about our character deleting itself before the player's view has moved to the respawned character.

Enabling ragdoll on the character mesh is actually quite simple as the skeleton's physics asset has already been created for us. If you are unfamiliar with this asset, it's called **SK_Mannequin_PhysicsAsset** and it will have been migrated with the skeleton mesh. Open the asset now and you will be presented with this view:



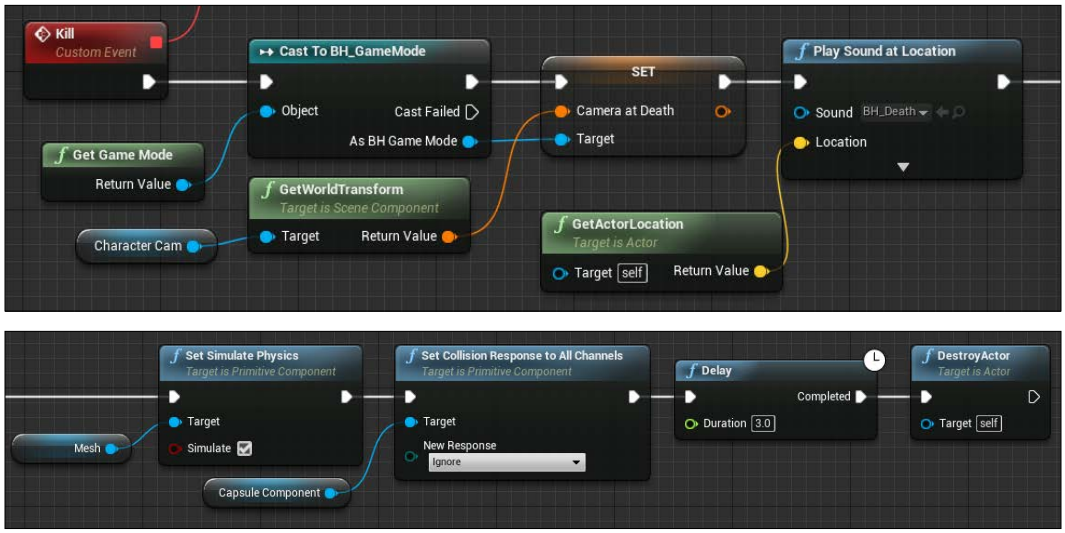
Each of those capsules surrounding the player acts as physics bodies that are constrained by joints. They will act as the driving manipulators for the bone positions of the mesh. The result, a limp ragdoll that allows for the motion of the skeletal mesh! Usually these assets have to be created by hand, one capsule at a time, when the mesh is imported into the engine. These assets must be created if you would like to utilize physics simulations for the character mesh.

To enable ragdoll on the character mesh, simply drag a reference node into the **Event Graph** for the `Mesh` object. Drag a line off of the reference pin and search for the `SetSimulatePhysics` node, then check the `bool` **simulate** parameter. To ensure that the mesh ragdolls properly and collides with surrounding physics volumes, be sure to set the collision preset on the `Mesh component` to Ragdoll, the setting can be found in the **Details** panel with the `Mesh component` selected. The arrangement can be seen as follows:



The next thing you must do is disable the collision on the `Capsule collider` as we no longer want objects in the world to collide with this component. You can do this by dragging a visual reference to the `capsule component` into the **Event Graph**, then searching for the `SetCollisionResponsesToAllChannels` node. Set the target to be the `capsule component` reference then set the **New Response** parameter to `Ignore` via the provided dropdown. Append this new ragdoll functionality to the `Kill` event functionality.

The last thing we have to do is ensure that we destroy the actor after a period of time. Summon a delay node, append this to the ragdoll functionality, then set the duration of the delay node to 3.0 seconds. Directly after this delay node, call `Destroy` actor. This means that, when the player dies, the camera will stay long enough over the dead character to see it ragdoll for a time, then after the camera moves back to the spawn position for the next character, the old character will delete itself. Your finished kill event functionality will look similar to this:



Now play the scene, deliberately kill the character by running into a barrel, and see the character mesh flail around, similar to what can be seen in the following:

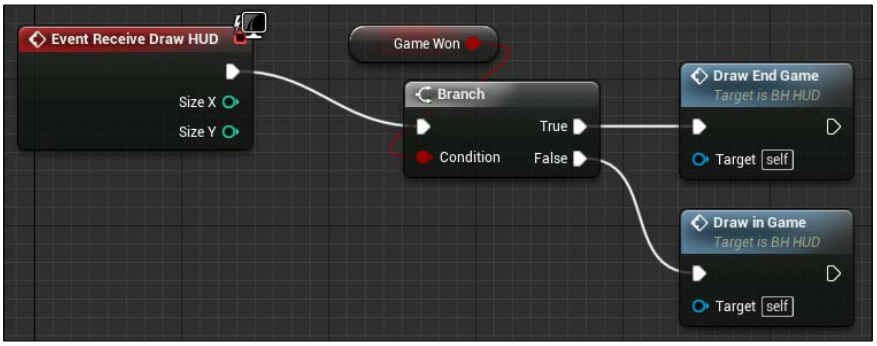
Creating a basic HUD

Now that we have all of our visual polish features in place, we can create a **HUD** that will show how long the player's current run attempt is. This HUD will also complete our game loop and allow the player to see their final time upon reaching the top of the level. We will also offer the player the choice to exit the game upon completion or to reset to the start position. To do this we are going to have to create two new objects, a `BH_HUD` object and a `BH_Winner` object. `BH_HUD` will inherit from the Unreal Engine `HUD` object and the `BH_Winner` will inherit from the `trigger box` object in a similar way to our `BH_BarrelKiller` objects.

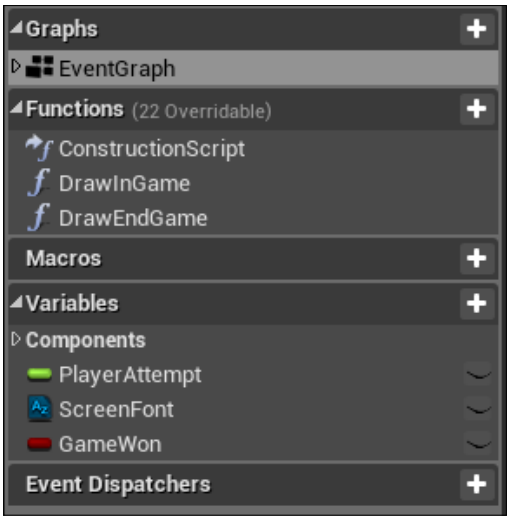
Making the HUD object

Our `BH_HUD` object is going to be responsible for drawing the in-game HUD as well as the end game screen. This means we are going to be creating two functions, one called `DrawInGame` and one called `DrawEndGame`. We will need to call either function based off of a bool that represents whether or not the player has reached the end of the level. Create a `BH_HUD` object now by opening the blueprint wizard, be sure that the parent class of the blueprint is of type `HUD`.

In the **MyBlueprint** panel of our `BH_HUD` blueprint we are going to be adding two functions, `DrawInGame` and `DrawEndGame`. Then we are going to add a `float` variable titled `PlayerAttempt`, a `Font reference` titled `ScreenFont`, and a `bool` value titled `GameWon`. Do this now. In the event graph of the `BH_HUD` there will be an event titled `EventRecieve DrawHUD`, this event will be called by the engine when the renderer deems it necessary to draw the HUD. From this event create a `branch` node based off of `GameWon`. From the `True` Branch call `DrawEndGame`, and from the false Branch call `DrawInGame`. Great! The shell of our HUD object has been created. Now we need to add meaningful functionality to the two draw functions. Before we move on, use these images to ensure you have set up the `BH_HUD` blueprint properly, your graph should appear as follows:



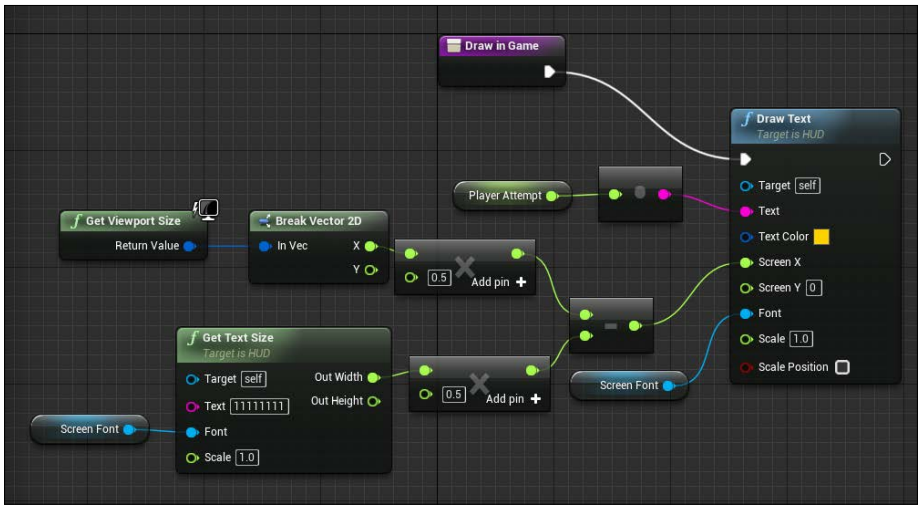
And your **MyBlueprint** panel will look like this:



Drawing the in game HUD

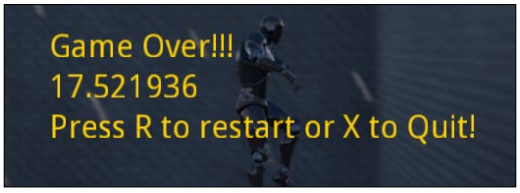
Open the `DrawInGame` function graph by double-clicking on the entry in the **Functions** sections in the **MyBlueprint** panel. We need to draw the player's current run attempt to the screen. We can do this via a function called `DrawText`. We need to figure out how we can position this text so it appears at the top-center of the screen. We can do this by getting our viewport dimensions, dividing the width value by two, then subtracting the width of our text that will be drawn to the screen from this value. This calculation will give us an X position to draw the text that will compensate for the width of the text string so that the text always appears in the center of the screen.

This is easily done with the following node arrangement:

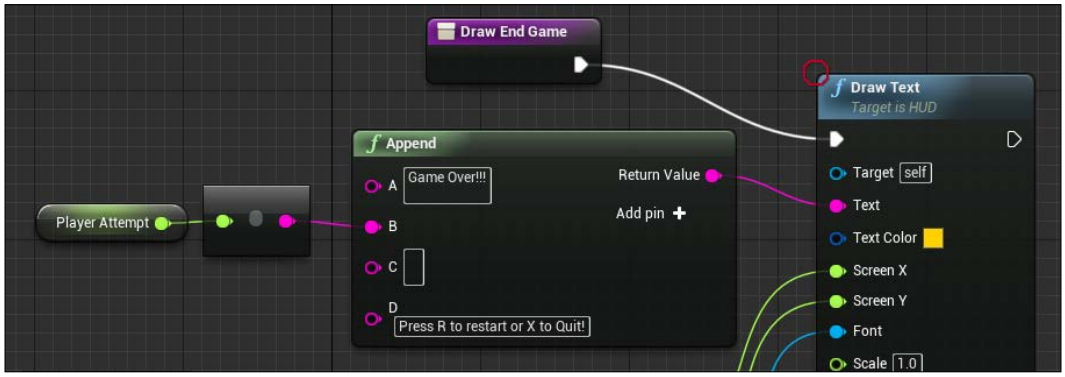


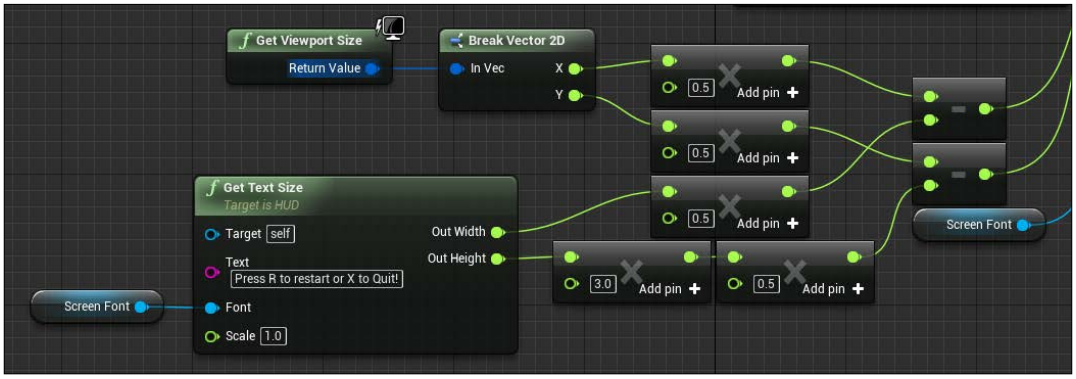
As you can see, we are using the `PlayerAttempt` variable we added earlier to determine what is to be drawn. We are then getting our viewport dimensions via a pure function, dividing the x component by two (or multiplying it by 0.5), then subtracting the resultant X value by the string text length multiplied by the width of a character using our provided font. The font type as of yet has not been specified, but we will get that later. You will notice that the Y value has been set to zero for the screen text as we wish the text to be flush with the top of the screen.

Now to add functionality to `DrawEndGame`. This will be slightly different as we need to draw a few additional lines, and we need to have the following text appear:



This is going to require more of the same calculations we did for the in-game text but we are going to position the text in the center of the screen. The arrangement of nodes appears as follows:





As you can see, the functionality is very similar to that of `DrawInGame`. This time however, we are also taking into account the **vertical scale** of the text as we wish for it to appear in the center of the screen. Also, instead of drawing three lines of text to the screen, we can instead create one big string that includes new line characters and draw that to the screen instead. To do this, simply hold shift when you press return to add a new line character to a text field. When taking the height into account, it's important to multiply the height by 3 as there are 3 lines of text that we are using for the end game scene. There will be a small margin of error as this does not account for the vertical spacing of the lines.

Making the font

`Font` objects are very easy to create and are done so in the same way as traditional Unreal Engine objects. Right-click in a content browser folder and create a font object found under the **User Interface** dropdown. Create a new font now and call it `BH_Font`. Open this font now. You will be presented with a new window, the font editing window. Here you can specify which installed font you wish to use. You can use any installed font file for this. You can also specify how the font is cached in memory as well as a default font size and name.

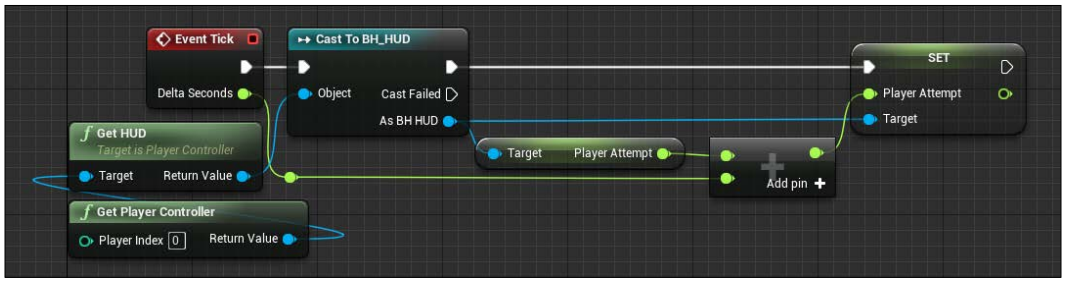
In our case, we need to press the **Add font** button that can be found in the **Composite Font** panel at the top of the window. This will then prompt you to name the font, select an installed font you wish to use, and which hinting algorithm to use with the font. Hinting of fonts is an algorithm used when rendering a font so that the outline of a font lines up with a rasterize grid (pixel grid). This makes the font appear smoother and removes visual artifacts when rendering a font to the screen. We can leave all of these values as default for now as our font will be nothing special.

We need to ensure our font is of a correct base size. The reason we will adjust the size of the font and not scale it from the `DrawText` node is because this way the font will maintain resolution. If we were to scale the font up using the `DrawText` node, it will appear pixelated. To adjust the size of the font, change the **Legacy Font Size** under the **Runtime Font** section in the details panel to `24`. Now navigate back to our `BH_HUD` object and be sure to set the default value of our font reference screen to `BH_Font`.

Parsing information to and setting the HUD

To make sure we populate the `PlayerAttempt` variable with the correct value, we need to be able to increment that value while the player is alive and has yet to reach the end of the level. To do this, we simply need to increment this value from inside the `BH_Character` blueprint via the `Tick` event. We can do this by navigating to the `Event Tick` node in the `BH_Character` event graph. To get the HUD object we have created, simply use the `GetPlayerController` node to get a reference to the `player controller`, then from the output reference search for the function `GetHUD`. Cast this reference to `BH_HUD`. Now all we need to do is get the `PlayerAttempt` value from this `BH_HUD` reference, increment by the `Delta Seconds` variable that has been parsed via the `Event Tick` node, then set the `PlayerAttempt` variable with the result.

This can be done with the following arrangement:



The next thing we need to do is ensure we use our new `BH_HUD` object as the HUD for the game. This can be done back in the **Details** Panel of the `BH_GameMode`. We can set the `HUD` class parameter under the **Classes** section of the **Details** Panel of `BH_GameMode` to ensure that our new HUD object is used.

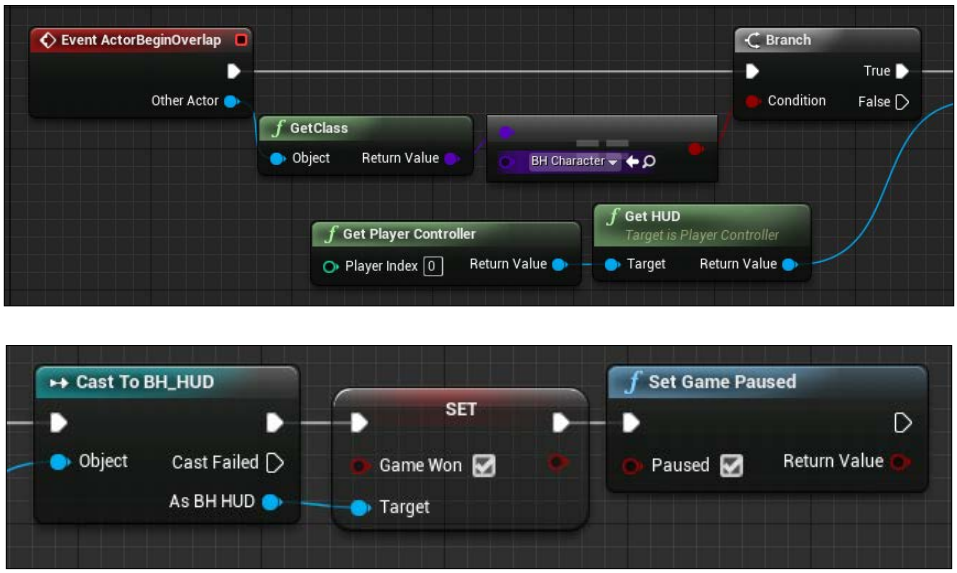
An end goal for the player, the chapter, and the project

Now all we need to do is create and flesh out the `BH_Winner` object. This object is simply going to dictate when the player reaches the end of the level, pause the game at this time, inform the HUD to end the game, and bring up the Game End text.

Start off by creating a blueprint that inherits from `Trigger Volume`. Call this blueprint `BH_Winner`. Address the blueprint **Viewport** and select the `Collision Component`. Make sure that the **Box Extent** settings in the **Shape** section of the **Details** panel is set to 96 across all axis. This will ensure that this box occupies $96cm^3$ of space in the scene.

We simply need to append functionality to the `ActorBeginOverlap`. Drag a line from the input `OtherActor` from the `Event ActorBeginOverlap` node. Call `GetClass` on this input reference via searching for it using the context sensitive search. Check that this class is equal to the `BH_Character` class by creating a class comparison node. Do this by searching for **==** after dragging a line from the purple class output pin. Create a `branch` node based off of this comparison operation. Now we need to get the `HUD` again and cast it to the `BH_HUD`. Do this in the same way we did when setting the `PlayerAttempt` variable. Instead, this time we will be setting `GameWon` to `true` then pausing the game by using the `SetGamePaused` function node.

The node arrangement appears as follows:

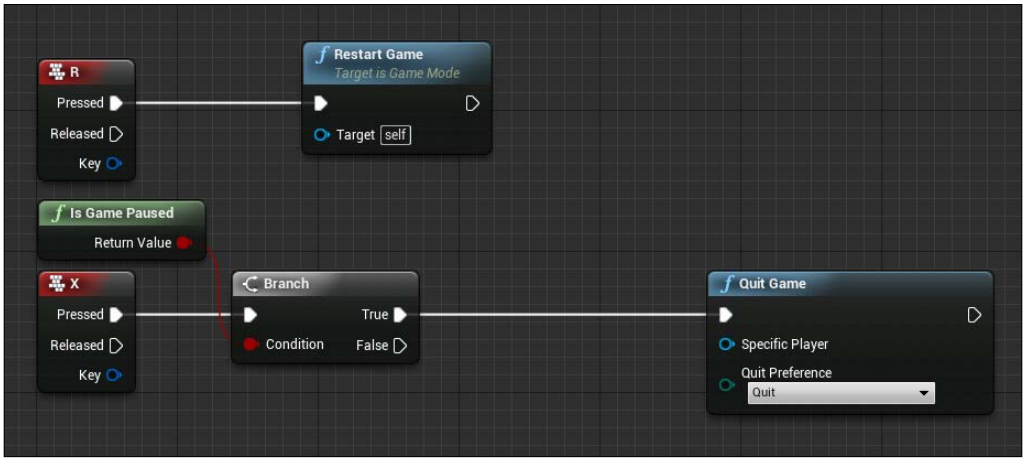


The only thing we have left to do is ensure that our game mode can receive input then when the player presses *R*, the game will restart or if the player has reached the end, press *X* to close the game.

This is very simply done by navigating to the event `Begin Play` in the `BH_GameMode`. From the functionality following this node, append one more node. Search for the **Enable Input** node. This node will take in a reference to the player controller you wish the object to receive input from. In our case we want our game mode to receive input from the default player controller (the keyboard) so we will use the `GetPlayerController` node and specify index 0, then plug the resultant player controller reference into the **Player Controller** parameter of the Enable input node.

Next, summon two input event nodes `X` and `R`. You can do this by simply searching for **X** in the **pallet** or context sensitive search. This is the other way to receive input events from within blueprints. It saves you having to create an input mapping, however you will not be able to change this input to something else without modifying the graph, like you can with an input mapping.

From the `R` event, summon a node called `RestartGame`. This node will reset the game to default values and positions. From the X event, check whether the game is paused by searching for the pure function `IsGamePaused`, then use a branch node to check whether this `bool` value resolves to true. If so, summon the node `QuiteGame`. This will take in a reference to a specific player and a quit preference. Leave both of these as default. The node arrangement should appear as follows:



Summary

We are done! Congratulations! You have finished your first Unreal Engine By Example project. You have gone from creating a small empty scene with a simple flaming sphere to a fully-fledged mini-game! Over the course of this chapter, you learnt about event dispatchers and how to tidy up your code graph using logical flow nodes. You have been introduced to the animation and sound toolsets provided by UE4, and you have added a HUD to the Barrel Hopper project. You are progressing well.

In the next chapter we are going to begin our journey by covering the fundamentals of using C++ and UE4. I look forward to guiding you through your first C++ Unreal Engine project!