APPENDIX

CHAPTER 1: UNREAL ENGINE INTRODUCTION

ACTIVITY 1.01: MOVING TESTACTOR ON THE Z AXIS INDEFINITELY

In this activity, we will be using the **`TestActor Tick`** event to move it on the *Z* axis indefinitely.

The following steps will help you complete this activity:

1. Open the **`TestActor`** Blueprint class.
2. Add the **`Event Tick`** node to the Blueprint's **`Event Graph`** window:

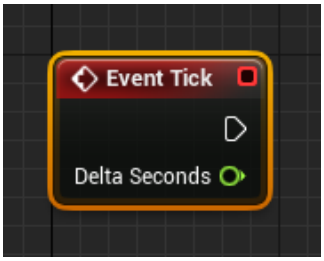

Figure 1.52: The Tick event node

3. Add the **`AddActorWorldOffset`** function, split its **`DeltaLocation`** pin, and connect the **`Tick`** event's output execution pin to this function's input execution pin, similar to what we did in *Exercise 1.01*, *Creating an Unreal Engine 4 Project*:

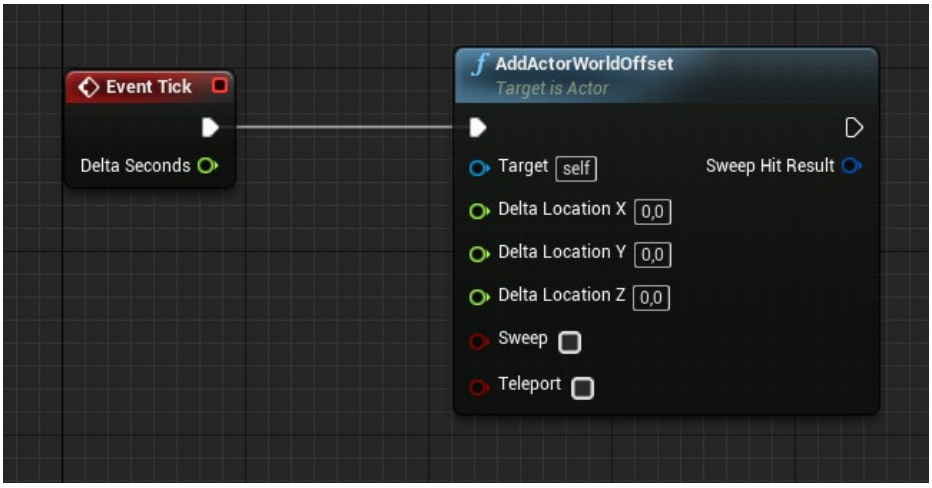

Figure 1.53: The AddActorWorldOffset node connected to the Tick node

4. Add a *Float Multiplication* node to **`Event Graph`**, as shown in the *Float Multiplication Node* section:

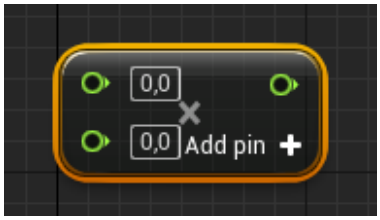

Figure 1.54: The Float Multiplication node

5. Connect the **`Tick`** event's **`Delta Seconds`** output pin to the first input pin of the *Float Multiplication* node:

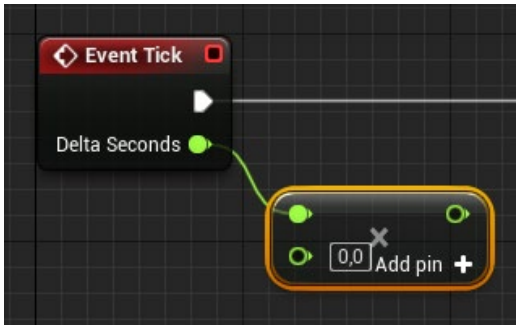

Figure 1.55: The Float Multiplication node connected to the Tick node's Delta Seconds output pin

6. Create a new variable of the **`float`** type, call it **`VerticalSpeed`**, and set its default value to **`25`**:

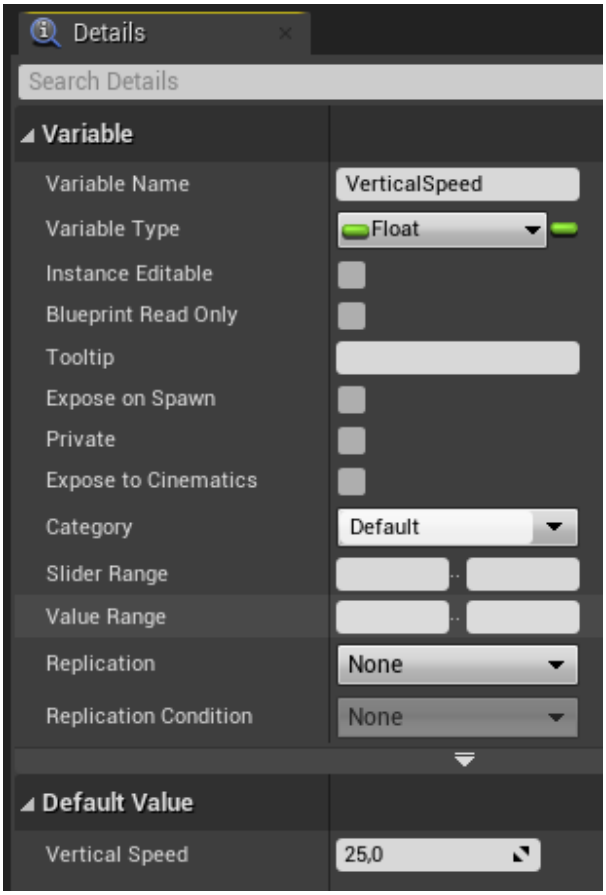

Figure 1.56: Details of the new VerticalSpeed variable

7. Add a getter to the **`VerticalSpeed`** variable to the **`Event Graph`** window and connect its pin to the second input pin of the *Float* Multiplication node. After that, connect the *Float Multiplication* node's output pin to the **`Delta Location Z`** pin of the **`AddActorWorldOffset`** function:

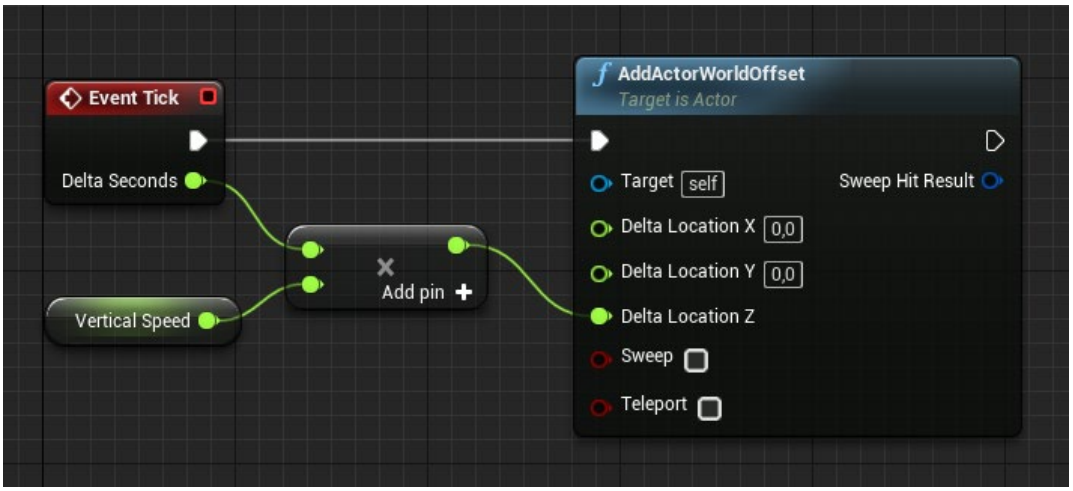

Figure 1.57: All the previous nodes being connected together

8. Delete the **`BeginPlay`** event and the **`AddActorWorldOffset`** function connected to it, both of which we created in *Exercise 1.01*, *Creating an Unreal Engine 4 Project*.

9. Play the level and notice our **`TestActor`** rising from the ground and up into the air over time:

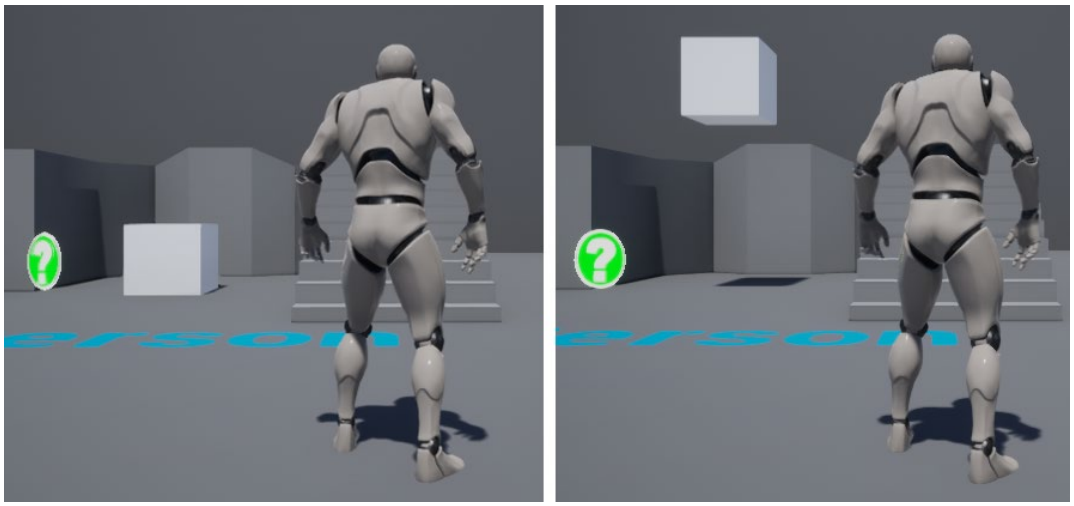

Figure 1.58: The TestActor propelling itself vertically

> **NOTE**
>
> The **`TestActor`** blueprint asset can be found here: https://packt.live/2U8pAVZ.

CHAPTER 2: WORKING WITH UNREAL ENGINE

ACTIVITY 2.01: LINKING ANIMATIONS TO A CHARACTER

The following steps will help you complete this activity:

1. Download and extract all the contents of the **`Chapter02`** -> **`Activity2.01`** -> **`ActivityFiles`** directory, which can be found on GitHub.

> **NOTE**
>
> The **`ActivityFiles`** directory can be found on GitHub at the following link: https://packt.live/3kje98A.

2. In the **`Content`** folder, browse to **`Content`** -> **`Ganfault`**-> **`Anims`**.
3. *Right-click* inside the **`Content`** folder, and from the **`Animation`** section, select **`Blend Space 1D`**.
4. Select **`Ganfault_Skeleton`**.
5. Rename the newly created file **`BS_IdleRun`**.
6. *Double-click* **`BS_IdleRun`** to open it.
7. Under the **`Asset Details`** tab, inside the **`Axis Settings`** section, expand the **`Horizontal Axis`** section and set **`Name`** to **`Speed`** and **`Maximum Axis Value`** to **`400.0`**
8. Drag and drop the **`Ganfault_Idle`**, **`Ganfault_Walk`**, and **`Ganfault_Run`** animations into the graph.

9. Under the **Asset Details** tab, inside **Blend Samples**, set the following variable values:

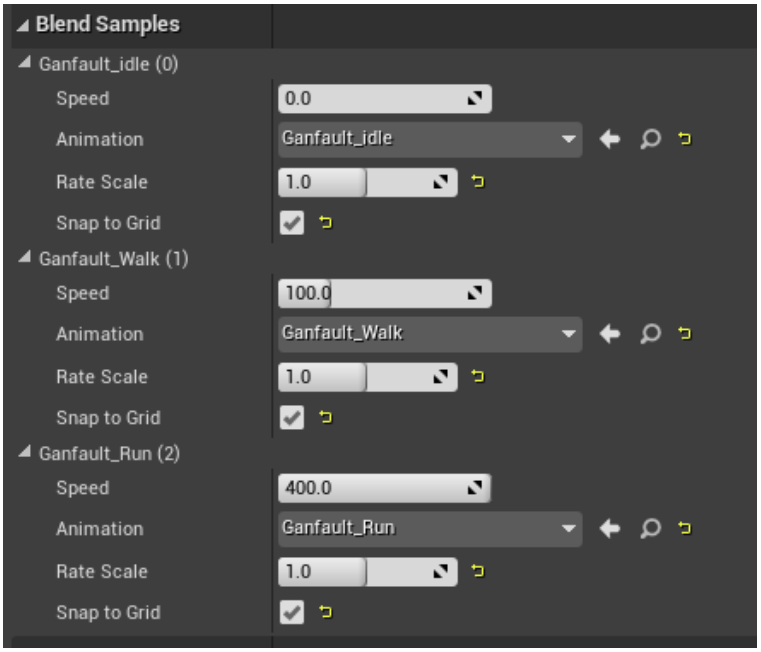

Figure 2.49: Blend Spaces

10. Click **Save** and close this **Asset**.
11. *Right-click* inside the **Content** folder, and from the **Animation** section, select **Animation Blueprint**.
12. In the **Target Skeleton** section, select **Ganfault_Skeleton** and then click the **OK** button.
13. Name the file **Anim_Ganfault** and press *Enter*.
14. *Double-click* the newly created **Anim_Ganfault** file.
15. Go to the **Event Graph** tab.
16. Create a Boolean variable called **IsInAir?**.
17. Create a float variable called **Speed**.
18. Drag off the **Try Get Pawn Owner** return value node and type in **Is Valid**.
19. Connect the **Exec** pin from the **Event Blueprint Update Animation** node to the **Is Valid** node:

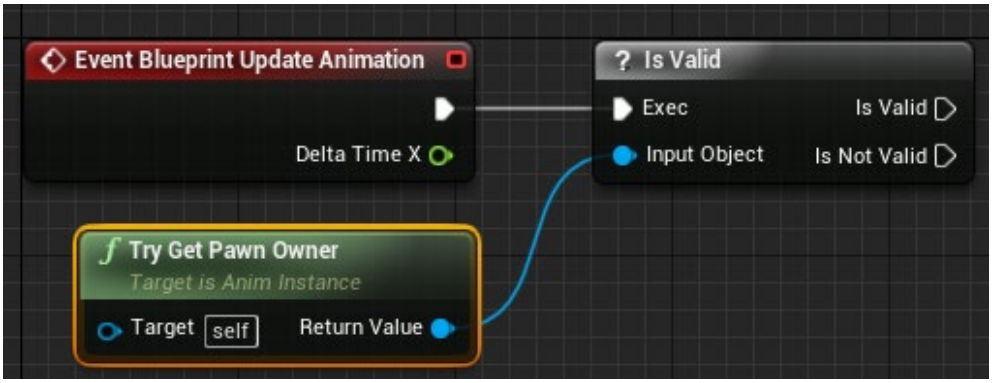

Figure 2.50: Event Graph Is Valid node

20. From the **`Try Get Pawn Owner`** node, use the **`Get Movement Component`** node. From that, get the **`Is Falling`** node and connect the Boolean return value to a set node for the **`Is in Air?`** Boolean. Connect the **`Set node`** exec pin to the **`Is Valid`** exec pin:

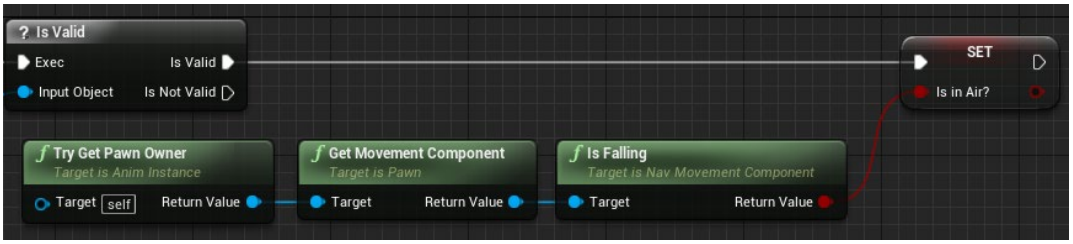

Figure 2.51: Is in Air Variable Setup

21. From the **`Try Get Pawn Owner`** node, use the **`Get Velocity`** node, get its **`VectorLength`**, and connect the output to a **`Variable Set`** node of **`Speed`**:

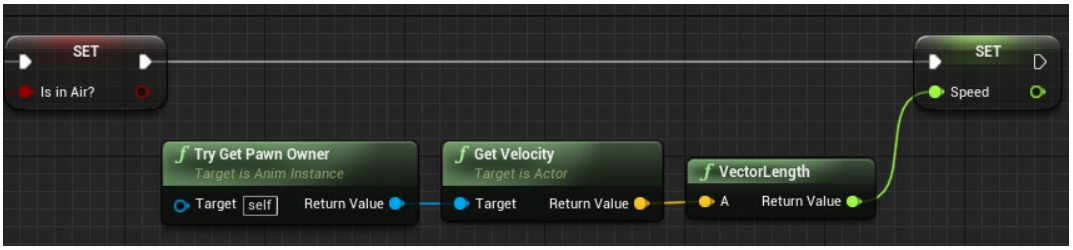

Figure 2.52: Speed variable setup

22. Head to the **`Anim Graph`** tab.

23. *Right-click* anywhere inside **`AnimGraph`**, type **`state machine`**, and click on **`Add New State Machine`**.

24. Make sure the node is selected and then press *F2* to rename it **`GanfaultStateMachine`**.
25. Connect the output pin of **`GanfaultStateMachine`** to the input pin for the **`Output Pose`** node:

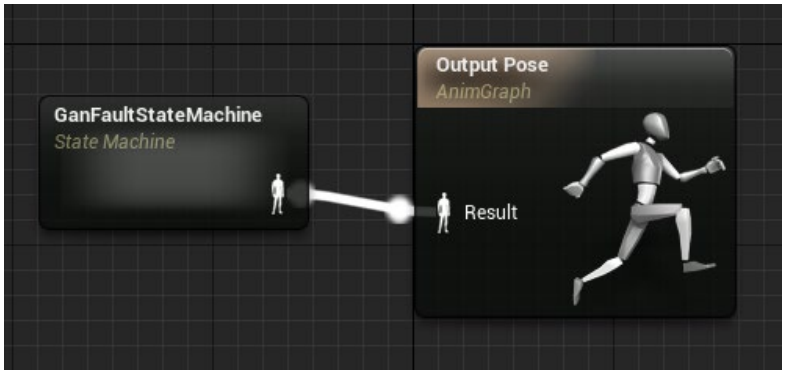

Figure 2.53: Connecting the State Machine result in Output Pose

26. *Double-click* the **`GanfaultStateMachine`** node to enter the State Machine.
27. *Right-click* on an empty area inside the State Machine, and from the menu, select **`Add State`**. Press *F2* to rename it **`Idle/Run`**.
28. Drag from the icon next to the **`Entry`** text, point it inside the **`Idle/Run`** node, and then release it to connect them.
29. *Double-click* on the **`Idle/Run`** state to open it.
30. From the **`Asset`** browser menu in the bottom-right corner, select and drag the **`BS_IdleRun`** Animation onto the graph. Get the **`Speed`** variable from the **`Variable`** section on the left and connect it.
31. Head back to **`GanfaultStateMachine`**, and from the **`Asset`** browser, drag and drop the **`Ganfault_Jump`** Animation into the graph and rename it **`Jump`**.
32. Open the **`Jump`** state. Click on the **`Play Ganfault_Jump`** node. Set **`Play Rate`** to **`0.5`**

33. Since we can shift from **`Idle/Run`** to **`Jump`** and back again, drag from the **`Idle/Run`** state and drop it at the **`Jump`** state and vice versa:

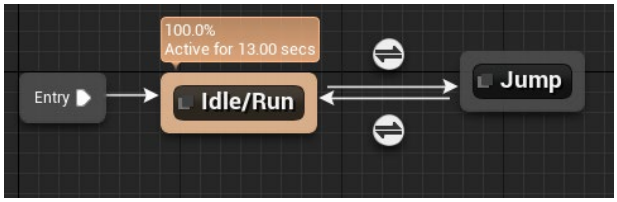

Figure 2.54: State Machine setup

34. *Double-click* the **`Idle/Run`** to **`Jump`** transition rule icon and connect the output of the **`IsinAir?`** variable to the result.

35. Open the **`Jump`** to **`Idle/Run`** transition rule. Connect the output of the inverse of the **`IsinAir?`** variable to the result.

36. Browse to the **`ThirdPersonCharacter`** blueprint.

37. Update the Mesh and Anim Class and play the game.

    You should get the following output:

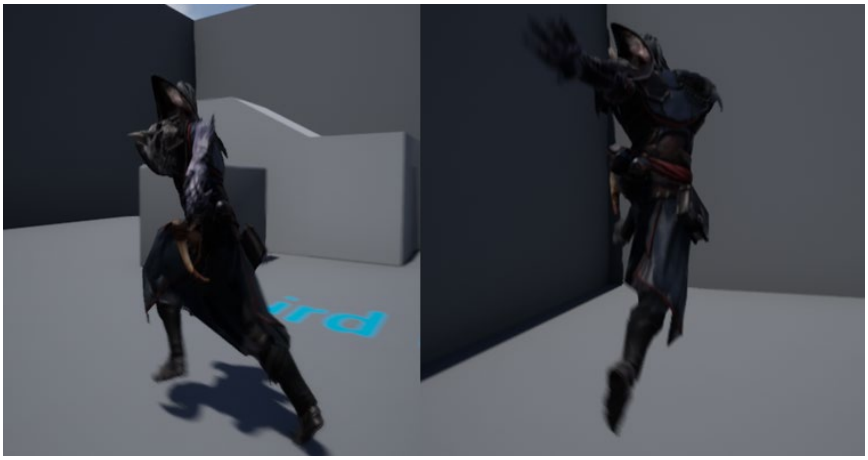

Figure 2.55: Expected output

> **NOTE**
>
> You can locate the complete activity code files on GitHub in the **`Chapter02`** -> **`Activity2.01`** -> **`Act2.01-Completed.rar`** directory at the following link:
>
> https://packt.live/3phfikF.
>
> After extracting the **`.rar`** file, *double-click* the **`.uproject`** file. You will see a prompt asking **`Would you like to rebuild now?`**. Click **`Yes`** on that prompt so that it can build the necessary intermediate files, after which it should open the project in Unreal Editor automatically.

CHAPTER 3: CHARACTER CLASS COMPONENTS AND BLUEPRINT SETUP

ACTIVITY 3.01: EXTENDING THE C++ CHARACTER CLASS WITH BLUEPRINT IN THE ANIMATION PROJECT

The following steps will help you complete this activity:

1. Open the Unreal Engine project.
2. *Right-click* inside the **`Content Browser`** interface and click the **`New C++ Class`** button.
3. In the dialog box that opens, select **`Character`** as the class type and click the **`Next`** button.
4. Name it **`MyThirdPersonChar`** and click the **`Create Class`** button. Upon doing so, Visual Studio will open the **`MyThirdPersonChar.cpp`** and **`MyThirdPersonChar.h`** tabs.
5. Open the **`MyThirdPersonChar.h`** tab and add the following code under the **`GENERATED_BODY()`** text:

```
// Camera boom positioning the camera behind the character
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
  MyTPS_Cam, meta = (AllowPrivateAccess = "true"))
USpringArmComponent* CameraBoom;


// Follow camera
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
  MyTPS_Cam, meta = (AllowPrivateAccess = "true"))
UCameraComponent* FollowCamera;
```

   In the preceding code snippet, we are adding code to add components to this class. This will add additional components to our class that we'll need to use, in addition to the ones inherited from the **`Character`** class.

6. Add the following **`include`** statements:

```
#include "GameFramework/SpringArmComponent.h"
#include "Camera/CameraComponent.h"
```

7. Now, head over to the **`MyThirdPersonChar.cpp`** tab and add the following **`include`** statements:

```
#include "Components/CapsuleComponent.h"
#include "GameFramework/CharacterMovementComponent.h"
```

8. Inside the **AMyThirdPersonChar::AMyThirdPersonChar()** function, add the following lines:

```
// Set size for collision capsule
GetCapsuleComponent()->InitCapsuleSize(42.f, 96.0f);

// Don't rotate when the controller rotates. Let that just
  affect the camera.
bUseControllerRotationPitch = false;
bUseControllerRotationYaw = false;
bUseControllerRotationRoll = false;

// Configure character movement
GetCharacterMovement()->bOrientRotationToMovement = true;

// Create a camera boom (pulls in towards the player if there
   is a collision)
CameraBoom =
  CreateDefaultSubobject<USpringArmComponent>(TEXT("CameraBoom"));
CameraBoom->SetupAttachment(RootComponent);
CameraBoom->TargetArmLength = 300.0f;
CameraBoom->bUsePawnControlRotation = true;

// Create a follow camera
FollowCamera =
  CreateDefaultSubobject<UCameraComponent>(TEXT("FollowCamera"));
FollowCamera->SetupAttachment(CameraBoom,
USpringArmComponent::SocketName);
FollowCamera->bUsePawnControlRotation = false;
```

In the preceding code snippet, we are initializing the capsule component with a specific radius and height. Next, we are making sure that when the player controller wants to rotate, the camera rotates and not the player itself. Then, we are setting up the **CameraBoom** component so that it's placed 300 units behind the mesh, and then setting up the follow camera so that it's attached to the **CameraBoom**. This will make the camera fixed to being 300 units behind the player and will move with the **CameraBoom** component.

9. Head back to the Unreal Engine project and click the **Compile** button in the top bar.
10. *Right-click* inside the **Content Browser** interface and click **Blueprint Class**.
11. In the **Search** box, type **MyThirdPersonChar**, select the class, and then click on the **Select** button.
12. Name the blueprint **BP_MyTPC**.
13. In the **World Settings** tab, click the **None** option next to **GameMode Override** and make sure **ThirdPersonGameMode** is selected.
14. Open **ThirdPersonGameMode**, which is located in **Content->ThirdPersonBP->Blueprints**.
15. Change **Default Pawn Class** to **BP_MyTPC**:

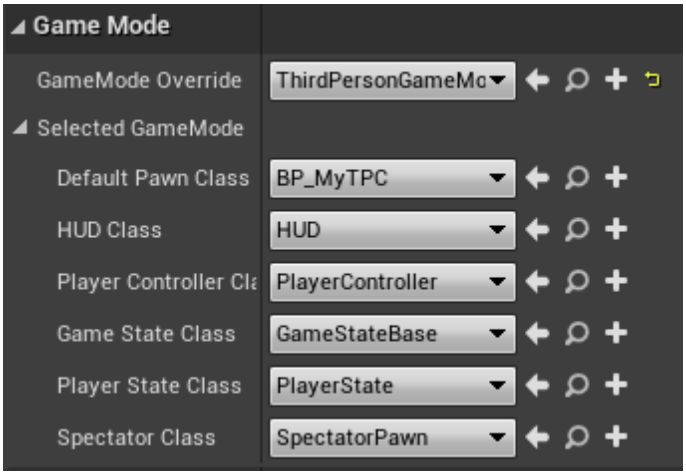

Figure 3.15: Setting the Game Mode

16. In the world, select **`ThirdPersonCharacter`**, which has been placed in the level, and press the *Delete* button to remove it:

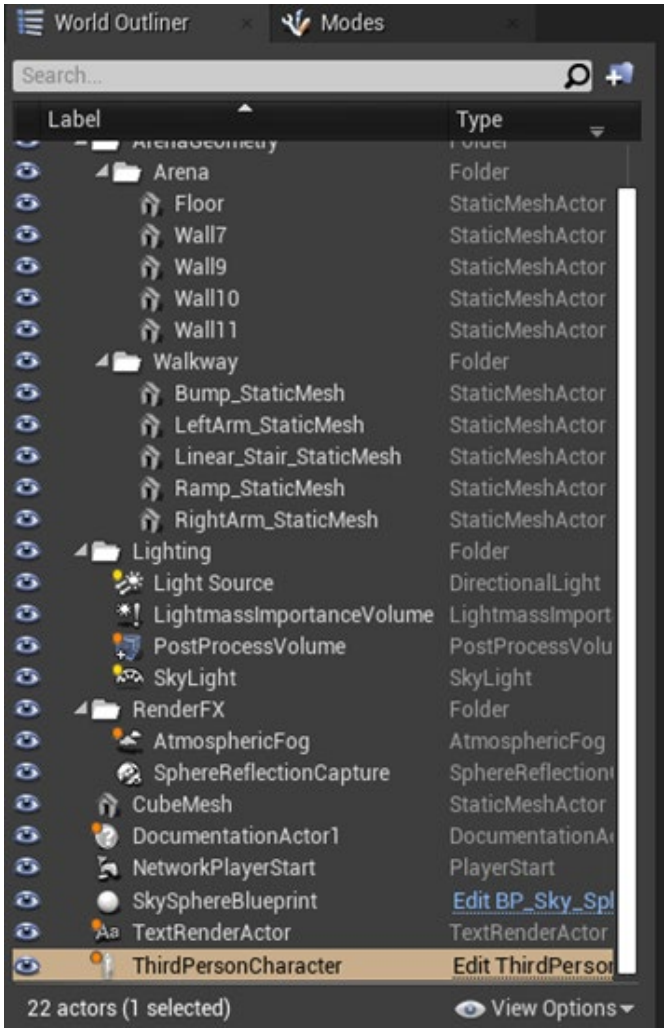

Figure 3.16: ThirdPersonCharacter visible in the World Outliner

17. Open **`BP_MyTPC`** and click on the **`Mesh (Inherited)`** component in the hierarchy of the **`Components`** tab on the left.

18. In the **`Details`** tab, find the **`Mesh`** section and set **`Skeletal Mesh`** to **`Ganfault_Aure`**.

19. In the **`Details`** tab, find the **`Animation`** section and set **`Anim Class`** to **`Anim_Ganfault_C`**:

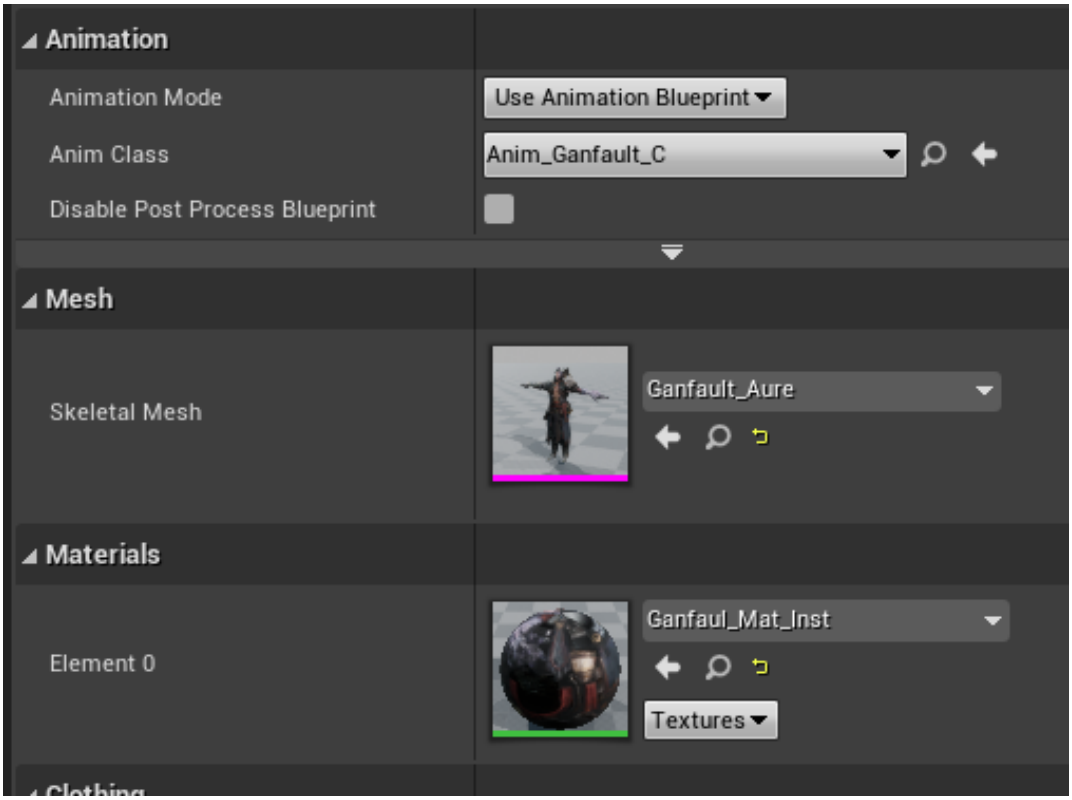

Figure 3.17: Ganfault Mesh and Anim Class setup

20. Open **`BP_MyTPC`** and click the **`Event Graph`** tab:

Figure 3.18: Event Graph tab

21. *Right-click* anywhere in the graph, type **`Input Jump`**, and select the first node option.

22. *Right-click* in the graph, search for **`Character Jump`**, and select the blue **`f`** icon node option:

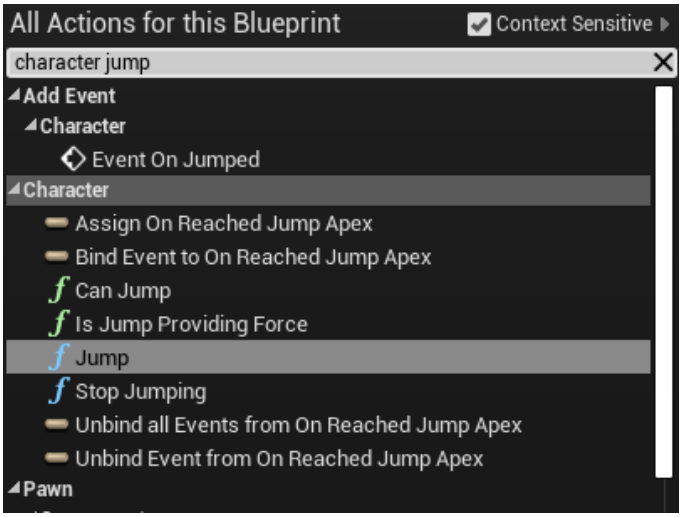

Figure 3.19: Jump event

23. *Right-click* in the graph, search for **`stop jumping`**, and select the first node option.

24. On the **`InputAction Jump`** node, connect the **`Pressed`** execution pin to the **`Jump`** node and the **`Released`** execution pin to the **`Stop Jumping`** node:

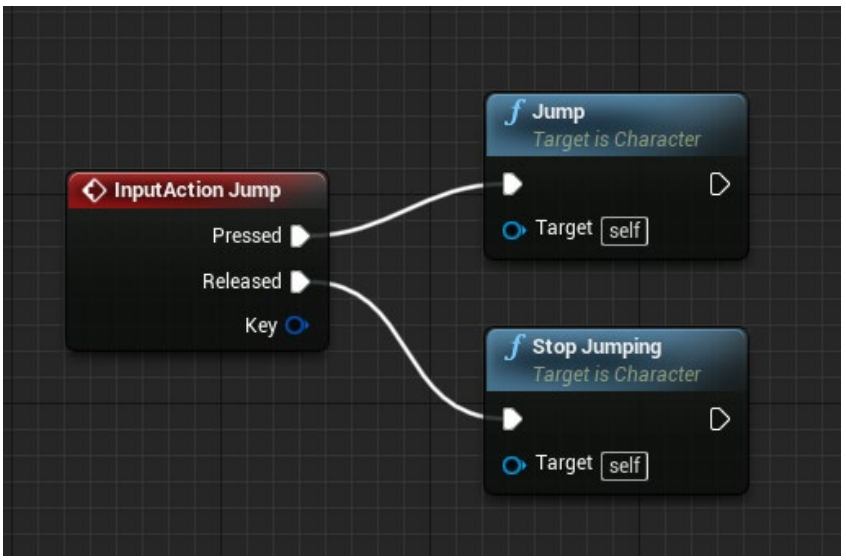

Figure 3.20: Jump logic Blueprint setup

25. Go to the **`Viewport`** tab. Here, you can see that the character's front is not pointing in the direction of the arrow and that the character is displaced in that it's preceding the capsule component. Adjust it so that the feet align with the bottom of the capsule component and the Mesh is rotated to point toward the arrow:

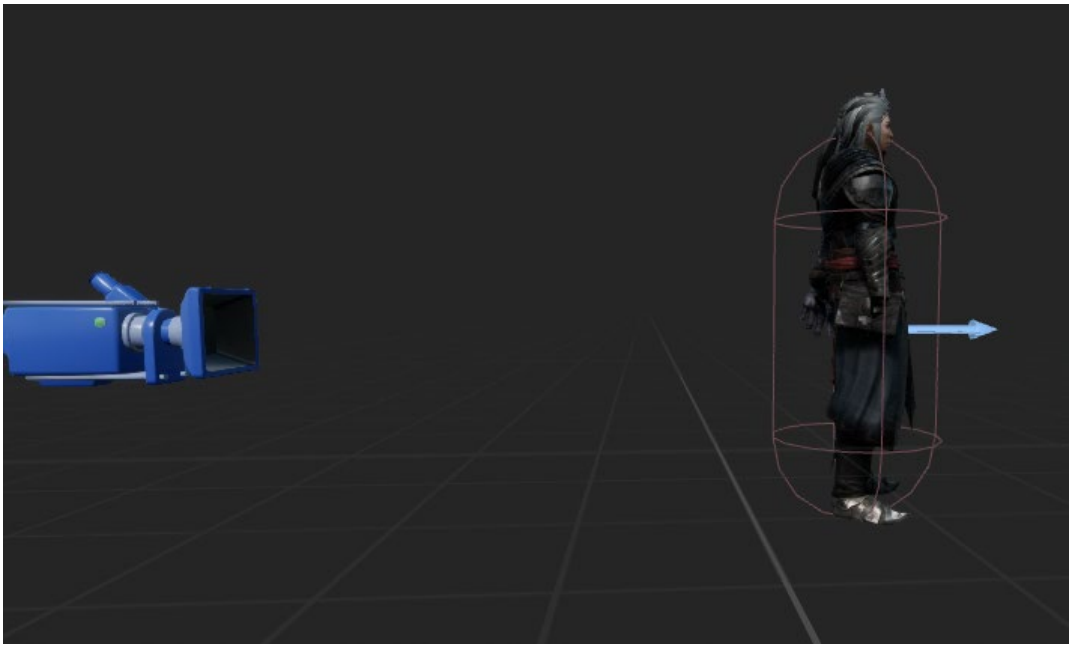

Figure 3.21: Adjusting the Mesh with the capsule component

26. In the **`Toolbar`** menu, press the **`Compile`** button and then **`Save`**.
27. Go back to the map and press the **`Play`** button to view your character in-game. Use the *spacebar* key to jump.

> **NOTE**
>
> You can locate the completed exercise code files on GitHub, in the **`Chapter03`** -> **`Activity3.01`** directory, at the following link:
>
> https://packt.live/3n84ymM.
>
> After extracting the **`.rar`** file, *double-click* the **`.uproject`** file. You will see a prompt asking **`Would you like to rebuild now?`**. Click **`Yes`** on that prompt so that it can build the necessary intermediate files, after which it should open the project in Unreal Editor automatically.

CHAPTER 4: PLAYER INPUT

ACTIVITY 4.01: ADDING WALKING LOGIC TO OUR CHARACTER

Perform the following steps:

1. Open **`Input Settings`** through the **`Project Settings`** window.
2. Add a new **`Action Mapping`** called **`Walk`** and associate it with the **`Left Shift`** and **`Gamepad Face Button Right`** keys:

Figure 4.22: The Walk Action mapped to both the Left Shift key and the Gamepad Face Button Right key

3. Open the **`MyThirdPersonChar`** class's header file and add declarations for two **`protected`** functions that return nothing and receive no parameters, called **`BeginWalking`** and **`StopWalking`**:

```
void BeginWalking();
void StopWalking();
```

4. Add the implementations for both these functions in the class's source file. In the implementation of the **`BeginWalking`** function, change the character's speed to 40% of its value by modifying the **`CharacterMovementComponent`** property's **`MaxWalkSpeed`** property accordingly. To access the **`CharacterMovementComponent`** property, use the **`GetCharacterMovement`** function:

```
void AMyThirdPersonChar::BeginWalking()
{
  GetCharacterMovement()->MaxWalkSpeed *= 0.4f;
}
```

The implementation for the **`StopWalking`** function will be the inverse of that of the **`BeginWalking`** function, which will increase the character's walk speed by 250%:

```
void AMyThirdPersonChar::StopWalking()
{
```

```
    GetCharacterMovement()->MaxWalkSpeed /= 0.4f;
}
```

5. Listen to the **`Walk`** action by going to the **`SetupPlayerInputComponent`** function's implementation and adding two calls to the **`BindAction`** function, the first one of which passes as parameters the **`"Walk"`** name, the **`IE_Pressed`** input event, the **`this`** pointer, and this class's **`BeginWalking`** function, while the second passes the **`"Walk"`** name, the **`IE_Released`** input event, the **`this`** pointer, and this class's **`StopWalking`** function:

```
PlayerInputComponent->BindAction("Walk", IE_Pressed, this,
  &AMyThirdPersonChar::BeginWalking);
PlayerInputComponent->BindAction("Walk", IE_Released, this,
  &AMyThirdPersonChar::StopWalking);
```

After following these steps, you should be able to have your character walk, which decreases its speed and slightly changes its animation, by pressing either the keyboard's **`Left Shift`** key or the controller's **`Face Button Right`**:

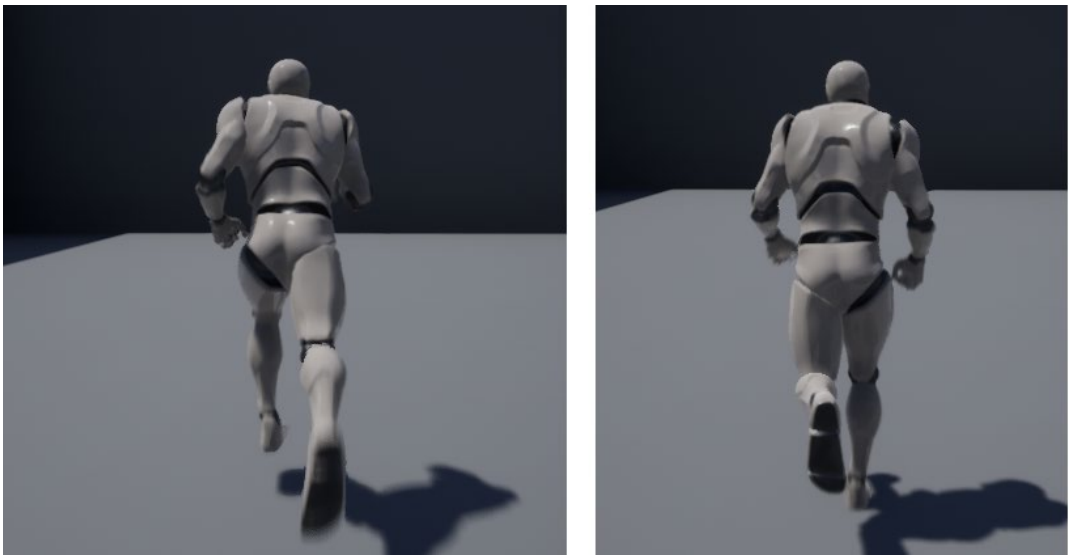

Figure 4.23: The character running (left) and walking (right)

6. Let's now preview our game on a mobile platform, as we did in *Exercise 4.03*, *Previewing on Mobile*, and drag the left analog stick just slightly to get our character to walk slowly. The result should look similar to the following screenshot:

Figure 4.24: The character walking in the mobile preview

CHAPTER 5: LINE TRACES

ACTIVITY 5.01: CREATING THE SIGHTSOURCE PROPERTY

The following steps will help you complete this activity:

1. Declare a new **`SceneComponent`** in our **`EnemyCharacter`**'s C++ class called **`SightSource`**:

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
  LookAt, meta = (AllowPrivateAccess = "true"))
class USceneComponent* SightSource;
```

2. Create this component in the **`EnemyCharacter`**'s constructor and attach it to the **`RootComponent`**:

```
SightSource =
  CreateDefaultSubobject<USceneComponent>(TEXT("Sight
  Source"));
SightSource->SetupAttachment(RootComponent);
```

3. Change the **`Start`** location of the Line Trace in the **`CanSeeActor`** function to the **`SightSource`** component's location, instead of the Actor's location:

```
// Where the Line Trace starts and ends
FVector Start = SightSource->GetComponentLocation();
FVector End = TargetActor->GetActorLocation();
```

4. Compile your changes in Visual Studio and then open the editor.

5. Open the **`BP_EnemyCharacter`** Blueprint class and change the **`SightSource`** component's location to the location of the enemy's head, **`(10, 0, 80)`**, as done in the *Creating the EnemyCharacter Blueprint class* section to the **`BP_EnemyCharacter`**'s **`SkeletalMeshComponent`** property:

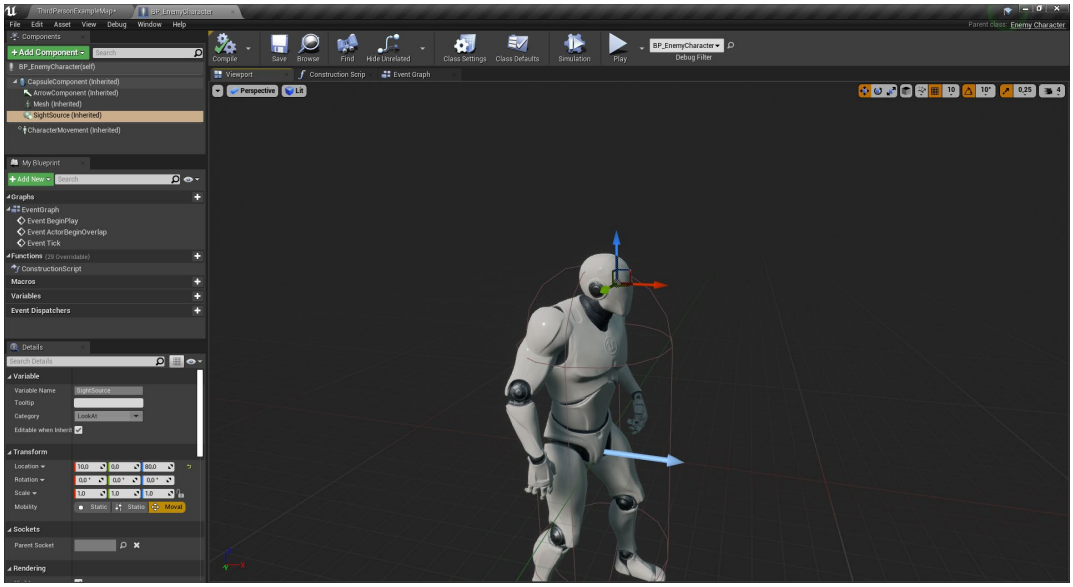

Figure 5.19: SightSource updated from the hip to the head

6. The result is that if the player character is brushing against the cube next to the enemy character, the enemy can't see it. However, if the character goes further away from the cube, eventually, the enemy will be able to see it:

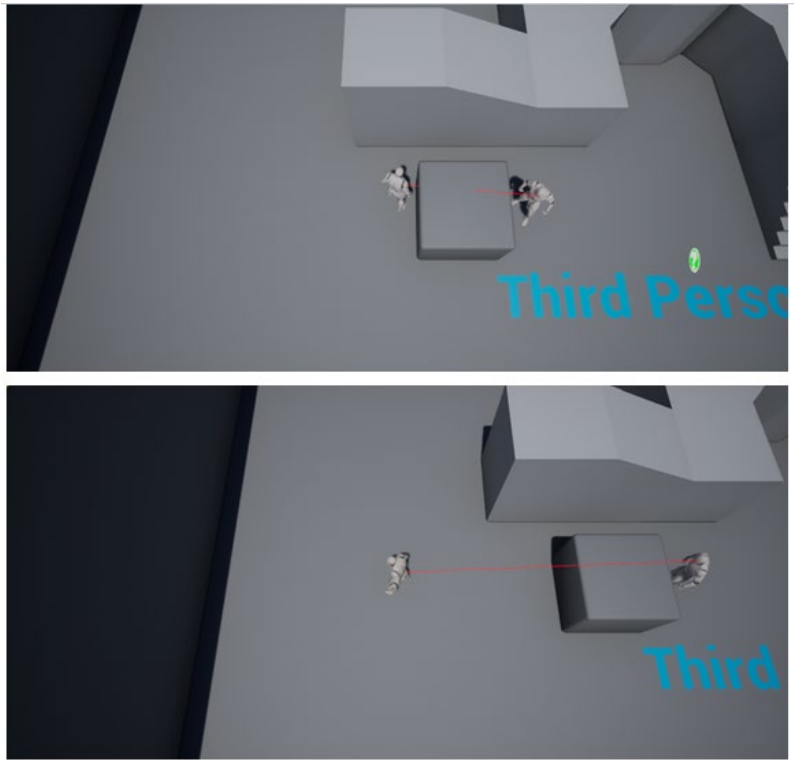

Figure 5.20: Expected output with updated SightSource

CHAPTER 6: COLLISION OBJECTS

ACTIVITY 6.01: REPLACING THE SPAWNACTOR FUNCTION WITH SPAWNACTORDEFERRED IN ENEMYCHARACTER

In this activity, we will be changing the **`EnemyCharacter`**'s **`ThrowDodgeball`** function in order to use the **`SpawnActorDeferred`** function instead of the **`SpawnActor`** function so that we can change the **`DodgeballProjectile`**'s **`InitialSpeed`** before spawning it.

The following steps will help you complete this activity.

1. Open the **`EnemyCharacter`** class's source file in Visual Studio.
2. Go to the **`ThrowDodgeball`** function's implementation.
3. Because the **`SpawnActorDeferred`** function can't just receive a spawn location and rotation properties and must instead receive an **`FTransform`** property, we'll need to create one of those before we call that function. Let's call it **`SpawnTransform`** and send the spawn rotation and location, in that order, as inputs for its constructor, which will be this enemy's rotation and the **`SpawnLocation`** property, respectively:

```
FTransform SpawnTransform(GetActorRotation(), SpawnLocation);
```

4. Then, change the **`SpawnActor`** function call into the **`SpawnActorDeferred`** function call. Instead of sending the spawn location and spawn rotation as its second and third parameters, replace those with the **`SpawnTransform`** property we just created, as the second parameter:

```
//Spawn new dodgeball
GetWorld()->SpawnActorDeferred<ADodgeballProjectile>(DodgeballClass,
  SpawnTransform);
```

5. Make sure you save the return value of this function call inside a **`ADodgeballProjectile*`** property called **`Projectile`**:

```
//Spawn new dodgeball
ADodgeballProjectile* Projectile = GetWorld()
  ->SpawnActorDeferred<ADodgeballProjectile>(DodgeballClass,
  SpawnTransform);
```

After you've done this, you will have successfully created a new **`DodgeballProjectile`** object. However, we still need to change its **`InitialSpeed`** property and actually spawn it.

6. After you've called the **SpawnActorDeferred** function, call the **Projectile** property's **GetProjectileMovementComponent** function, which returns its Projectile Movement Component, and change its **InitialSpeed** property to **2200** units:

```
Projectile->GetProjectileMovementComponent()->InitialSpeed =
  2200.f;
```

7. Because we'll be accessing properties belonging to the Projectile Movement Component inside the **EnemyCharacter** class, we'll need to include that component, just like we did in *Exercise 6.02*, *Adding a Projectile Movement Component to DodgeballProjectile*.

```
#include "GameFramework/ProjectileMovementComponent.h"
```

8. After you've changed the value of the **InitialSpeed** property, the only thing left to do is call the **Projectile** property's **FinishSpawning** function, which will receive the **SpawnTransform** property we created as a parameter:

```
Projectile->FinishSpawning(SpawnTransform);
```

9. After you've done this, compile your changes and open the editor.

   Now, when you play the level, you'll notice that the dodgeball being thrown by the enemy will be thrown slightly faster. If we want to, we can also change that speed by changing the value of the **InitialSpeed** property inside the **ThrowDodgeball** function:

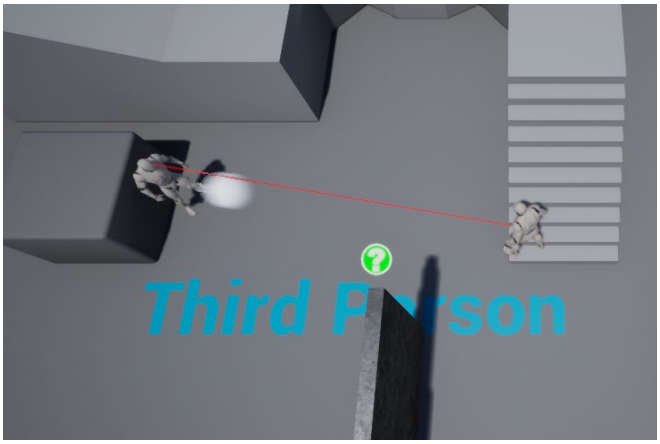

Figure 6.26: Dodgeball thrown at the player

CHAPTER 7: UE4 UTILITIES

ACTIVITY 7.01: MOVING THE LOOKATACTOR LOGIC TO AN ACTOR COMPONENT

In this activity, we'll be moving all of the logic related to the **`LookAtActor`** function, inside the **`EnemyCharacter`** class, to its own **`Actor`** Component. This way, if we want an actor (that isn't **`EnemyCharacter`**) to look at another actor, we will simply be able to add this component to it:

1. Open the editor and create a new C++ class that inherits from **`SceneComponent`**, called **`LookAtActorComponent`**.
2. When this class's files are opened in Visual Studio, go to its header file and add a declaration for the **`LookAtActor`** function, which should be **`protected`**, return a **`bool`**, and receive no parameters:

```
// Change the rotation of the character to face the given
  actor
// Returns whether the given actor can be seen
bool LookAtActor();
```

> **NOTE**
>
> While the **`LookAtActor`** function of **`EnemyCharacter`** received the **`AActor* TargetActor`** parameter, this Actor Component will have its **`TargetActor`** as a class property, which is why we won't need to receive it as a parameter.

3. Add a **`protected AActor*`** property called **`TargetActor`**. This property will represent the actor we want to look at:

```
AActor* TargetActor;
```

4. Add a **`protected bool`** property called **`bCanSeeTarget`**, with a default value of **`false`**, which will indicate whether the **`TargetActor`** can be seen:

```
//Whether the enemy can currently see the target
bool bCanSeeTarget = false;
```

5. Add a declaration for a **public FORCEINLINE** function called **SetTarget**, which will return nothing and receive **AActor* NewTarget** as a parameter. This function will be called by this actor component's **Owner** to set the desired target. The implementation of this function will simply set the **TargetActor** property to the value of the **NewTarget** property:

```
FORCEINLINE void SetTarget(AActor* NewTarget) { TargetActor =
  NewTarget; }
```

6. Add a declaration for a **public FORCEINLINE** function called **CanSeeTarget**, which will be **const**, return a **bool**, and receive no parameters. This function will be called by this actor component's **Owner** to check whether the target can be seen. The implementation of this function will simply return the value of the **bCanSeeTarget** property:

```
FORCEINLINE bool CanSeeTarget() const {return bCanSeeTarget;}
```

Now, go to the class's source file and take the following steps.

7. In the class's **TickComponent** function, after you've called the **TickComponent** function of **Super**, set the value of the **bCanSeeTarget** property to the return value of the **LookAtActor** function call:

```
Super::TickComponent(DeltaTime, TickType, ThisTickFunction);
 bCanSeeTarget = LookAtActor();
```

8. Add an empty implementation of the **LookAtActor** function and copy the **EnemyCharacter** class's implementation of the **LookAtActor** function into the implementation of **LookAtActorComponent**:

```
bool ULookAtActorComponent::LookAtActor()
{
  if (TargetActor == nullptr) return false;

  TArray<const AActor*> IgnoreActors = { this, TargetActor };
  if (UDodgeballFunctionLibrary::CanSeeActor(GetWorld(),
                                             SightSource-
  >GetComponentLocation(),
                                             TargetActor,
                                             IgnoreActors))
  {
    FVector Start = GetActorLocation();
    FVector End = TargetActor->GetActorLocation();
    // Calculate the necessary rotation for the Start point to
  face the End point
```

```
    FRotator LookAtRotation =
UKismetMathLibrary::FindLookAtRotation(Start, End);

    //Set the enemy's rotation to that rotation
    SetActorRotation(LookAtRotation);
    return true;
  }

  return false;
}
```

9. After you've copied the **`EnemyCharacter`** class's implementation of the **`LookAtActor`** function, make the following modifications to the **`LookAtActorComponent`** class's implementation of the **`LookAtActor`** function:

- Change the first element of the **`IgnoreActors`** array to be the **`Owner`** of the Actor Component, using the **`GetOwner`** function instead of the **`this`** pointer:

```
TArray<const AActor*> IgnoreActors = { GetOwner(), TargetActor
  };
```

- Change the second parameter of the **`CanSeeActor`** function call to be this component's location, instead of the **`SightSource`** component's location:

```
if (UDodgeballFunctionLibrary::CanSeeActor(
                               GetWorld(),
                               GetComponentLocation(),
                               TargetActor,
                               IgnoreActors))
```

- Change the value of the **`Start`** property to be the location of **`Owner`**, using the **`GetActorLocation`** function:

```
FVector Start = GetOwner()->GetActorLocation();
```

- Finally, replace the call to the **`SetActorRotation`** function with a call to the **`SetActorRotation`** function of **`Owner`**:

```
GetOwner()->SetActorRotation(LookAtRotation);
```

10. Because of the modifications we've made to the implementation of the **LookAtActor** function, we'll need to add some includes to our **LookAtActorComponent** class and remove some includes from our **EnemyCharacter** class. Remove the includes to **KismetMathLibrary** and **DodgeballFunctionLibrary** from the **EnemyCharacter** class and add them to the **LookAtActorComponent** class:

```
#include "Kismet/KismetMathLibrary.h"
#include "DodgeballFunctionLibrary.h"
```

11. We'll also need to add an **include** statement to the **Actor** class, since we'll be accessing several functions belonging to that class:

```
#include "GameFramework/Actor.h"
```

We'll now have to make some further modifications to our **EnemyCharacter** class.

12. In its header file, remove the declaration of the **LookAtActor** function:

```
// Remove this line
bool LookAtActor(const AActor* TargetActor);
```

13. Replace the **SightSource** property with a property of type **ULookAtActorComponent*** called **LookAtActorComponent**. Be sure to leave the **UPROPERTY** macro and **class** keyword as is:

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
  LookAt, meta = (AllowPrivateAccess = "true"))
class ULookAtActorComponent* LookAtActorComponent;
```

14. In the class's source file, add an include to **LookAtActorComponent**:

```
#include "LookAtActorComponent.h"
```

15. Inside the class's constructor, replace the references to the **SightSource** property with a reference to the **LookAtActorComponent** property. Additionally, the **CreateDefaultSubobject** function's template parameter should be the **ULookAtActorComponent** class instead of the **USceneComponent** class and its parameter should be **"Look At Actor Component"** instead of **"Sight Source"**:

```
LookAtActorComponent =
  CreateDefaultSubobject<ULookAtActorComponent>(TEXT("Look At
  Actor Component"));
LookAtActorComponent->SetupAttachment(RootComponent);
```

16. Remove the class's implementation of the **`LookAtActor`** function.
17. In the class's **`Tick`** function, remove the line of code where you create the **`PlayerCharacter`** property and set it to the value of the **`GameplayStatics`**' **`GetPlayerCharacter`** function, and add that exact line of code to the end of the class's **`BeginPlay`** function:

```
Super::BeginPlay();


// Fetch the character currently being controlled by the
  player
ACharacter* PlayerCharacter =
  UGameplayStatics::GetPlayerCharacter(this, 0);
```

18. After this line, inside the **`BeginPlay`** function, call the **`SetTarget`** function of **`LookAtActorComponent`** and send the **`PlayerCharacter`** property as a parameter:

```
LookAtActorComponent->SetTarget(PlayerCharacter);
```

19. Inside the class's **`Tick`** function, change the line where you set the **`bCanSeePlayer`** property's value to the return value of the **`LookAtActor`** function call to the return value of the **`CanSeeTarget`** function call of **`LookAtActorComponent`** instead:

```
// Look at the player character every frame
bCanSeePlayer = LookAtActorComponent->CanSeeTarget();
```

Now, there's only one last step we have to do before this activity is completed. Compile your changes in Visual Studio, open the editor and open the **`BP_EnemyCharacter`** Blueprint. In that Blueprint's **`Components`** tab, find **`LookAtActorComponent`** and change its location to **`(10, 0, 80)`**.

Expected output:

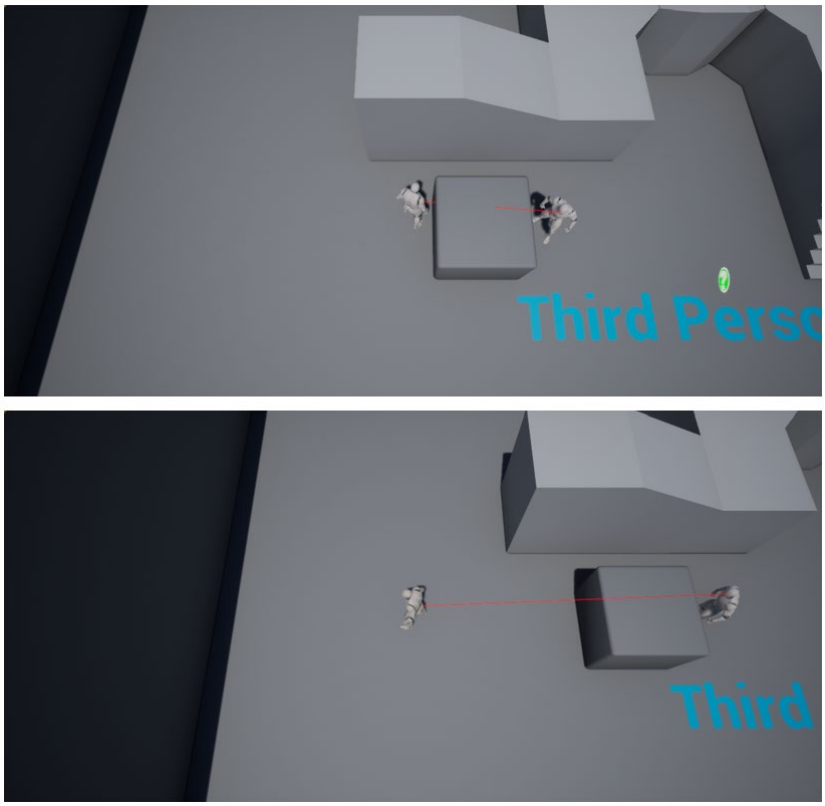

Figure 7.7: The enemy character's looking at the player character remains functional

CHAPTER 8: USER INTERFACES

ACTIVITY 8.01: IMPROVING THE RESTARTWIDGET

In this activity, we will be adding a **`Text`** element to our **`RestartWidget`** reading **`Game Over`** in order for the player to know that they just lost the game; adding an **`Exit`** button, which will allow the player to exit the game; and also updating the text of our existing button, which simply says **`Button 1`**, to **`Restart`**, so that the players know what should happen when they click that button. Let's complete the following steps:

1. Open the **`BP_RestartWidget`** Widget Blueprint.
2. Drag a new Text element from the **`Palette`** tab into the existing **`Canvas Panel`** element, inside the **`Hierarchy`** tab:

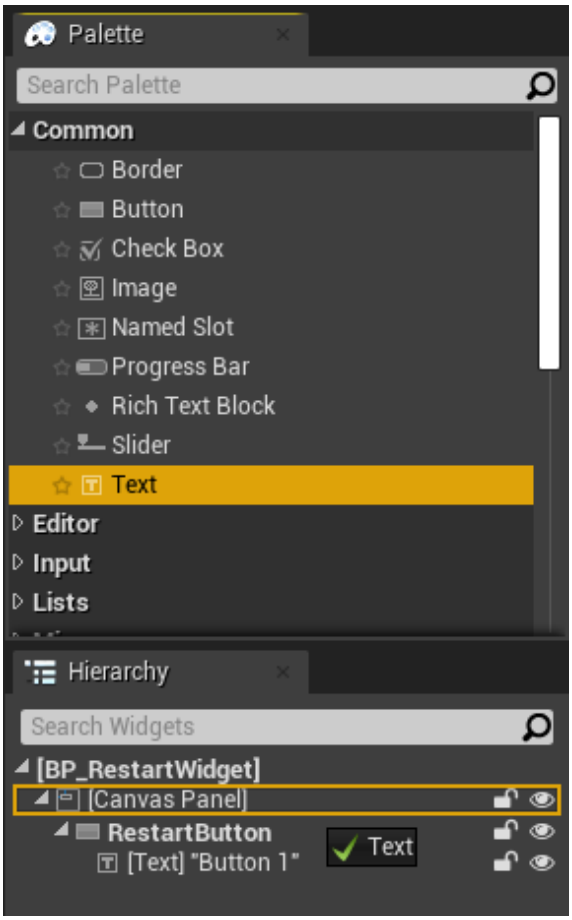

Figure 8.35: Dragging a text element into the Hierarchy window

3. Select this new **Text** element and go to its **Details** panel. Change the following properties inside the **Slot (Canvas Panel Slot)** category (that is, the first category):

- Expand the **Anchors** property and set its **Minimum** to **0.291** on the **X** axis and **0.115** on the **Y** axis, and its **Maximum** to **0.708** on the **X** axis and **0.255** on the **Y** axis.
- Set the **Offset Left**, **Offset Top**, **Offset Right**, and **Offset Bottom** properties to **0**.

4. Change the following properties inside the **Content** category:

- Set the **Text** property to **GAME OVER**.

5. Change the following properties inside the **Appearance** category:

- Set the **Color and Opacity** property to red: **RGBA(1.0, 0.082, 0.082, 1.0)**.
- Expand the **Font** property and set its **Size** to **100**.

- Set the **`Justification`** property to **`Align Text Center`** (the option in the middle):

Figure 8.36: The desired properties of the Game Over Text element

6. Select the other **`Text`** element inside the **`RestartButton`**, and change its **`Text`** property inside the **`Content`** category from **`Button 1`** to **`Restart`**.
7. Duplicate the **`RestartButton`** and change the copy's name to **`ExitButton`**.
8. Change the **`Text`** property of the **`Text`** element inside the **`ExitButton`** property to **`Exit`**.

9. Expand the **Anchors** property of **ExitButton** and set its **Minimum** to **0.44** on the **X** axis and **0.615** on the **Y** axis, and its **Maximum** to **0.558** on the **X** axis and **0.692** on the **Y** axis.

10. Set the **ExitButton** properties of **Offset Left**, **Offset Top**, **Offset Right**, and **Offset Bottom** properties **0**.

    This should be the result from your Widget Blueprint's **Designer** tab after all these modifications:

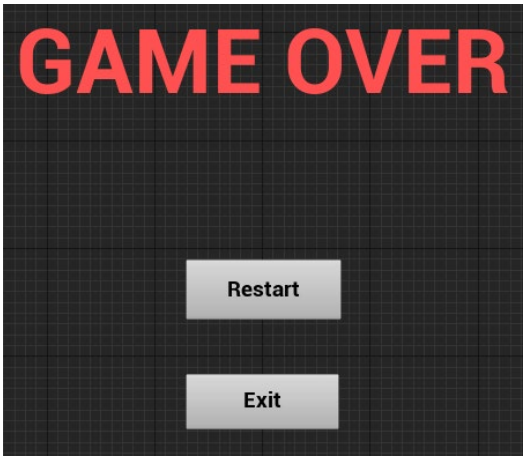

Figure 8.37: BP_RestartWidget after all the modifications specified in the preceding list

    After you've done these changes, we'll need to add the logic responsible for handling the **ExitButton** click, which will exit the game.

11. Save the changes made to the **BP_RestartWidget** Widget Blueprint and open the **RestartWidget** class's header file in Visual Studio. In this file, add a declaration for a **protected** function called **OnExitClicked** that returns nothing and receives no parameters. Be sure to mark it as a **UFUNCTION**:

```
UFUNCTION()
void OnExitClicked();
```

12. Duplicate the existing **RestartButton** property, but call it **ExitButton** instead:

```
UPROPERTY(meta = (BindWidget))
class UButton* ExitButton;
```

13. Inside the **RestartWidget** class's source file, add an implementation for the **OnExitClicked** function. Copy the contents of the **OnBeginOverlap** function, inside the **VictoryBox** class's source file, into the contents of the **OnExitClicked** function, but remove the cast being done to the **DodgeballCharacter**:

```
void URestartWidget::OnExitClicked()
{
  UKismetSystemLibrary::QuitGame(this,
nullptr,
EQuitPreference::Quit,
true);
}
```

14. In the **NativeOnInitialized** function implementation, bind the **OnExitClicked** function we created to the **OnClicked** event of **ExitButton** property, the same way we did for the **RestartButton** in *Exercise 8.03*, *Creating the RestartWidget C++ Class*.

```
if (ExitButton != nullptr)
{
  ExitButton->OnClicked.AddDynamic(this,
  &URestartWidget::OnExitClicked);
}
```

And that concludes our code setup for this activity. Compile your changes, open the editor, then open the **BP_RestartWidget** and compile it just to make sure there are no compilation errors due to the **BindWidget** tags.

Once you've done so, play the level again, let the player character be hit by three Dodgeballs, and notice the **`Restart`** Widget appear with our new modifications:

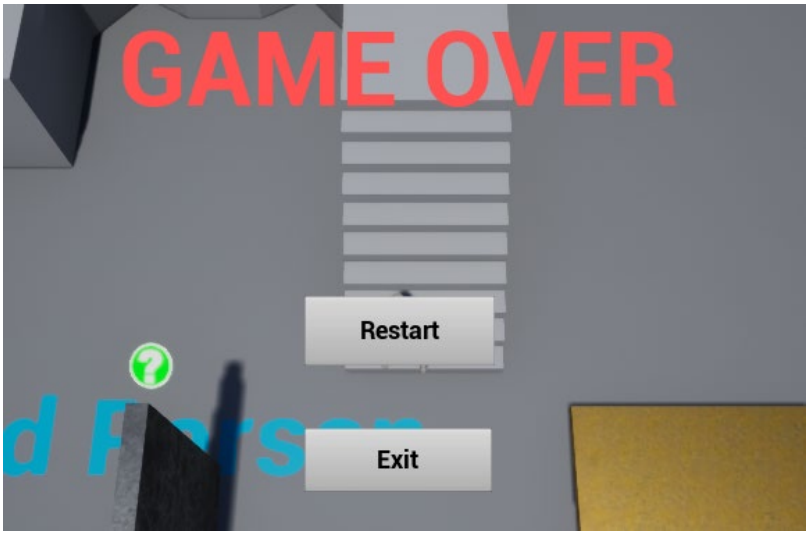

Figure 8.38: The updated BP_RestartWidget being shown after the player runs out of health points

If you press the **`Restart`** button, you should be able to replay the level, and if you press the **`Exit`** button, the game should end.

CHAPTER 9: AUDIO-VISUAL ELEMENTS

ACTIVITY 9.01: PLAYING A SOUND WHEN THE DODGEBALL HITS THE PLAYER

In this activity, we will be creating the logic responsible for playing a sound every time the player character gets hit by a dodgeball. To do this, follow these steps:

1. Open the Unreal Engine editor.
2. Import the sound file that will be played when the player character gets hit into the **`Audio`** folder inside the Content Browser:

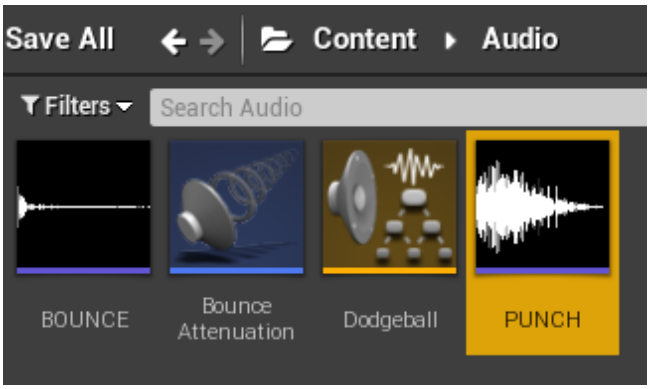

Figure 9.21: The imported audio file

3. Open the **`DodgeballProjectile`** class's header file. Add a **`SoundBase*`** property, just like we did in *Exercise 9.02*, *Playing a Sound When the Dodgeball Bounces off of a Surface*, but this time call it **`DamageSound`**:

```
// The sound the dodgeball will make when it hits the player
UPROPERTY(EditAnywhere, Category = Sound)
class USoundBase* DamageSound;
```

4. Open the **`DodgeballProjectile`** class's source file. In the **`OnHit`** function's implementation, after you've damaged the player character and before you call the **`Destroy`** function, check whether the **`DamageSound`** property is valid. If it is, call the **`GameplayStatics`** object's **`PlaySound2D`** function (mentioned in *Exercise 9.02*, *Playing a Sound When the Dodgeball Bounces off of a Surface*), passing **`this`** and **`DamageSound`** as the parameters to that function call:

```
if (DamageSound != nullptr)
{
  UGameplayStatics::PlaySound2D(this, DamageSound);
}
```

5. Compile your changes and open the editor.
6. Open the **`BP_DodgeballProjectile`** Blueprint and set its **`DamageSound`** property to the sound file you imported at the start of this activity:

Figure 9.22: Setting the DamageSound property to our imported sound

When you play the level, you should notice that every time the player gets hit by a dodgeball, you will hear the sound you imported being played.

> **NOTE**
>
> The complete source code and header files can be found at the following link to this book's GitHub repository: https://packt.live/2It5J13.

CHAPTER 10: CREATING A SUPERSIDESCROLLER GAME

ACTIVITY 10.01: MAKING OUR CHARACTER JUMP HIGHER

1. Open your Unreal Engine Editor.
2. Next, head to the **`CharacterMovement`** component of **`SideScrollerCharacter`**:

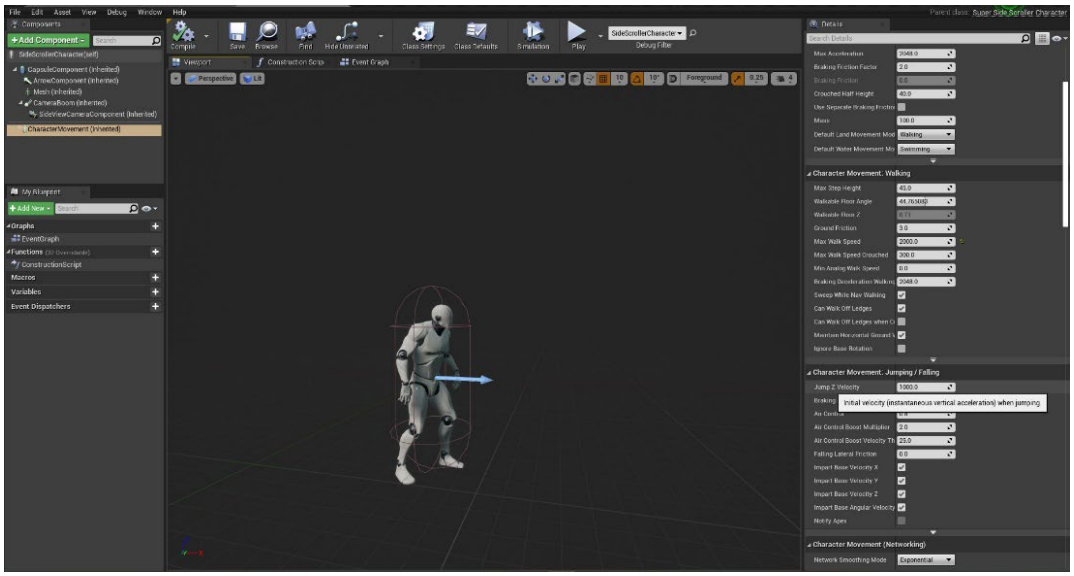

Figure 10.33: The Jump Z Velocity parameter is shown under the Character Movement: Jumping/Falling category

3. Update the **`Jump Z Velocity`** parameter, which you can see in *Figure 10.45*. It is found under the **`Character Movement: Jumping/Falling`** category:

> **NOTE**
>
> By default, the value is set to **`1000.0`**f. Change the value to **`2000.0`**f, and then compile and save the Character Blueprint.

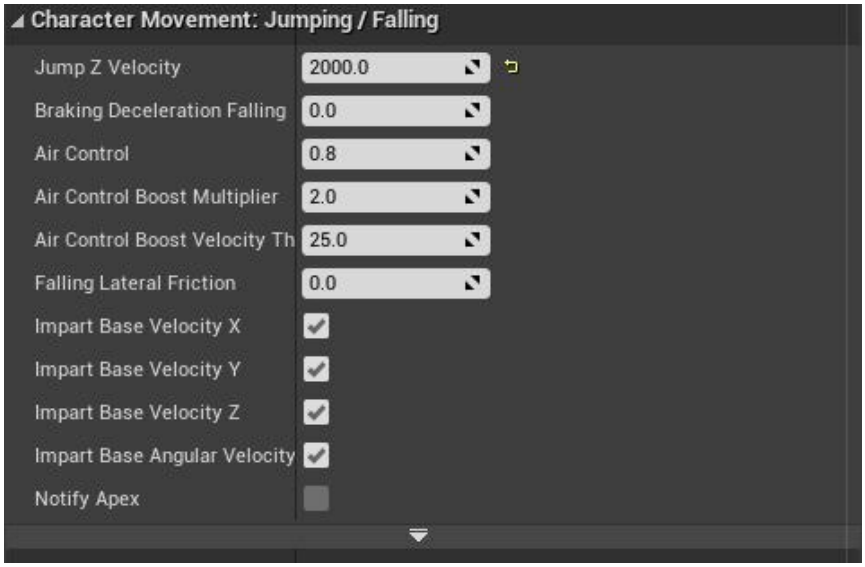

Figure 10.34: Jump Z Velocity has been updated to a value of 2000.0f

4. Now, play in the editor and jump using the *Spacebar*, and you will see that the player Character can now jump much higher, as shown in *Figure 10.46*:

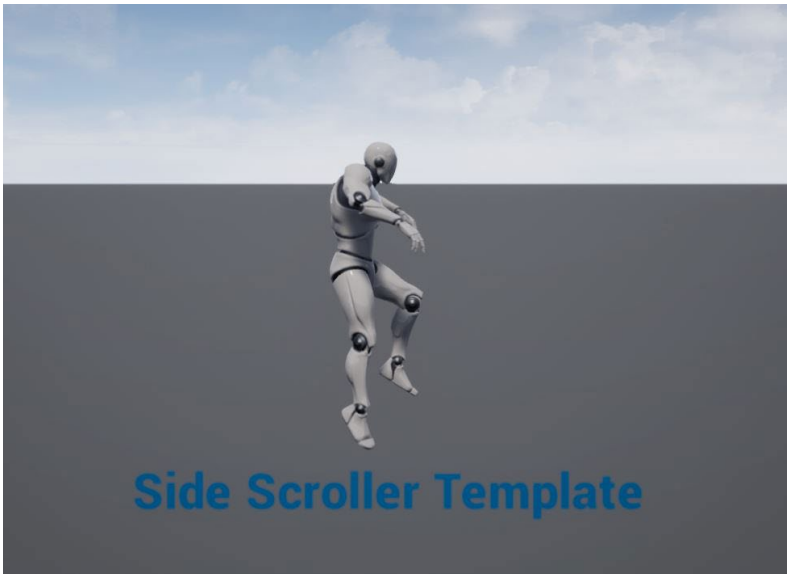

Figure 10.35: Our Character can now jump twice as high as before

5. Return to the **`SideScrollerCharacter`** Blueprint. Select the **`CharacterMovement`** component and change **`Jump Z Velocity`** from **`2000.0`**f to **`200.0`**f:

Figure 10.36: Jump Z Velocity has been updated to a value of 200.0f

6. Compile and save the Blueprint again and play in the editor. Now you will see that our Character cannot jump very high at all, as shown in *Figure 10.48*:

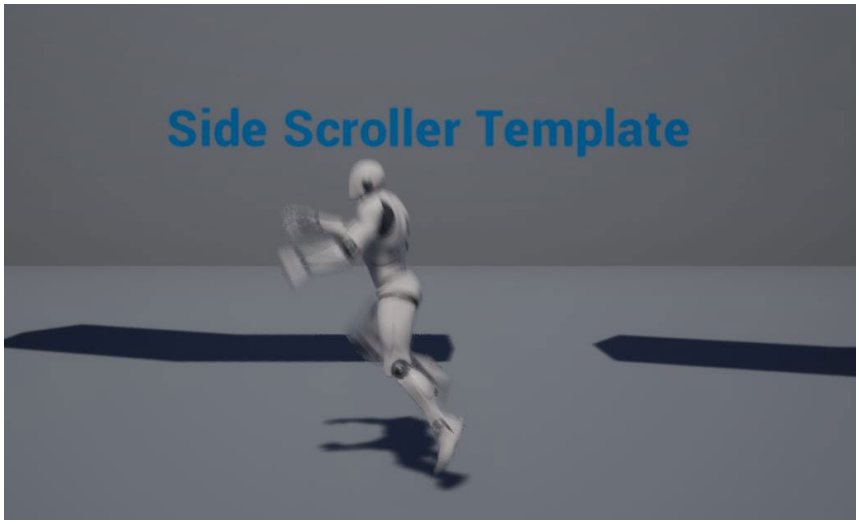

Figure 10.37: Our Character can now barely make it off the ground

ACTIVITY 10.02: SKELETAL BONE MANIPULATION AND ANIMATIONS

1. Inside **`Content Browser`**, navigate to the **`Mannequin skeleton Content\Mannequin\Character\Mesh`** directory.
2. Select **`UE4_Mannequin_Skeleton`** and open it by *double left-clicking* the asset.
3. With the Skeleton open inside the Persona Editor, you can look at the skeletal bone hierarchy on the left-hand side. The bone you need to select for the first step of this activity is the **`root`** bone:

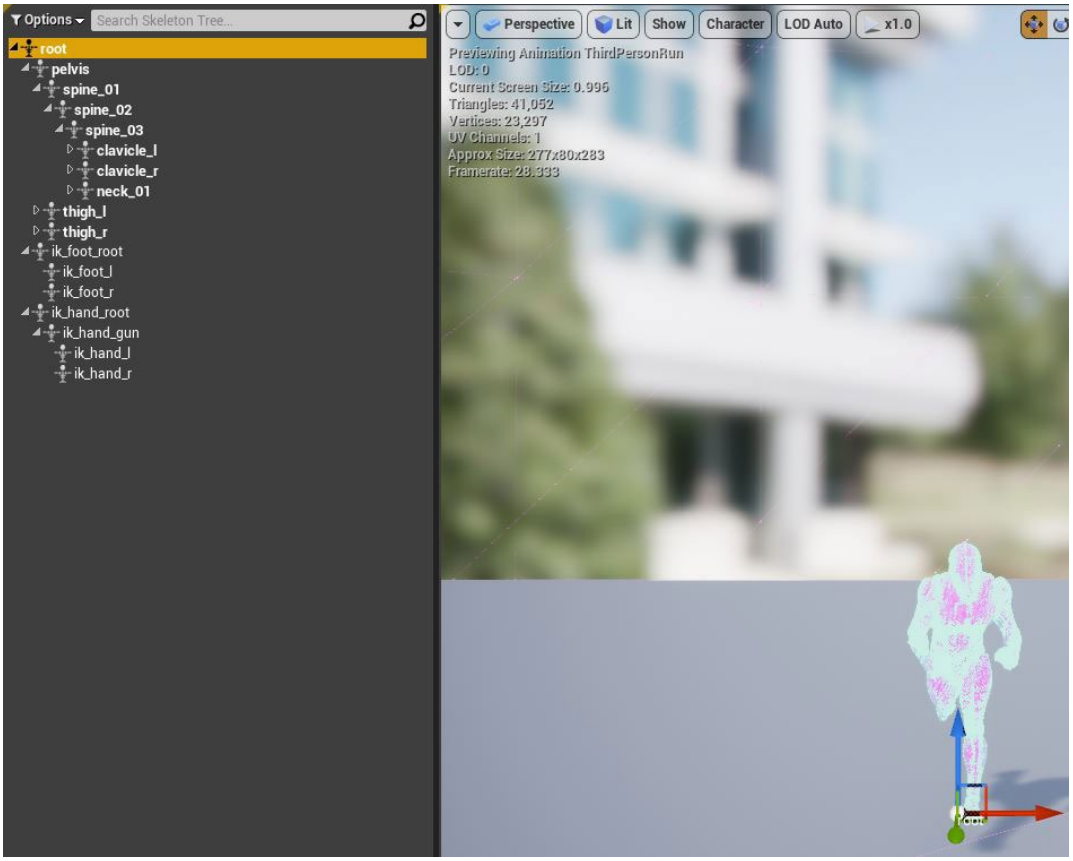

Figure 10.38: The root bone of the skeleton has been selected

4. With the **`root`** bone selected, access its **`Transform`** properties in the **`Details`** panel. This activity requires that we make the Character half its current size, so change **`Scale`** to (**`X=0.500000, Y=0.500000, Z=0.500000`**):

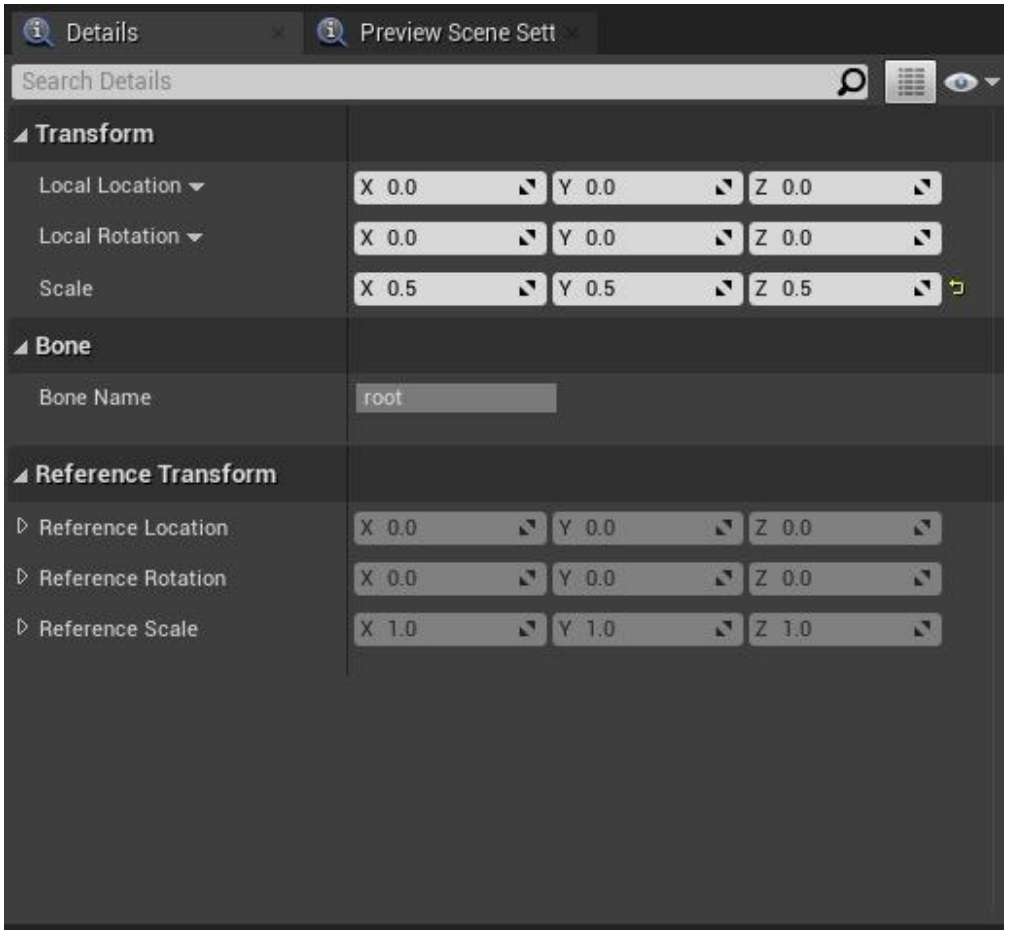

Figure 10.39: Updating the scale with the root bone selected

You will notice how our Character has now been halved in size:

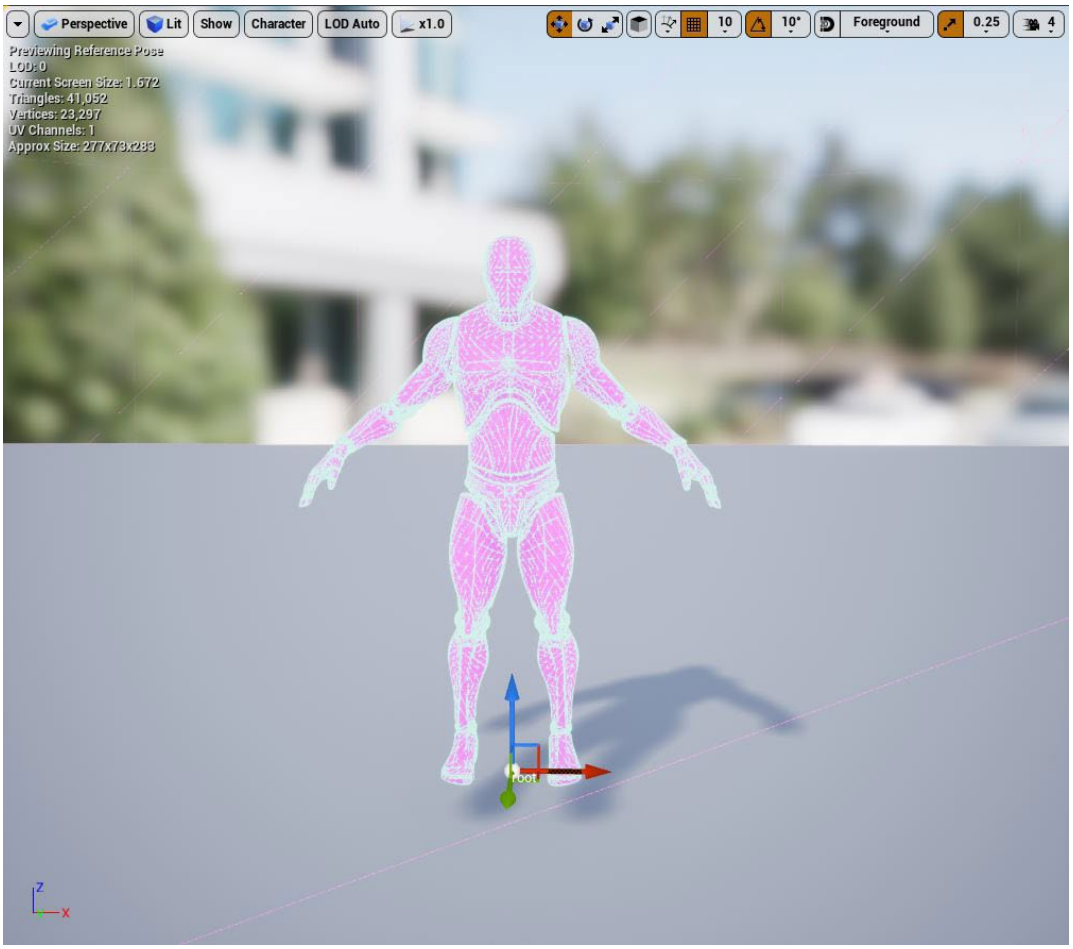

Figure 10.40: Our character is halved in size

5. Lastly, apply the running animation to the mannequin Character. To do this, access the **`Preview Scene Settings`** tab. Then, for the **`Preview Controller`** option, set this to **`Use Specific Animation`**. With this parameter set, the new **`Animation`** option will appear. From the dropdown of the **`Animation`** parameter, search for **`ThirdPersonWalk`**. Please refer to *Figure 10.52*:

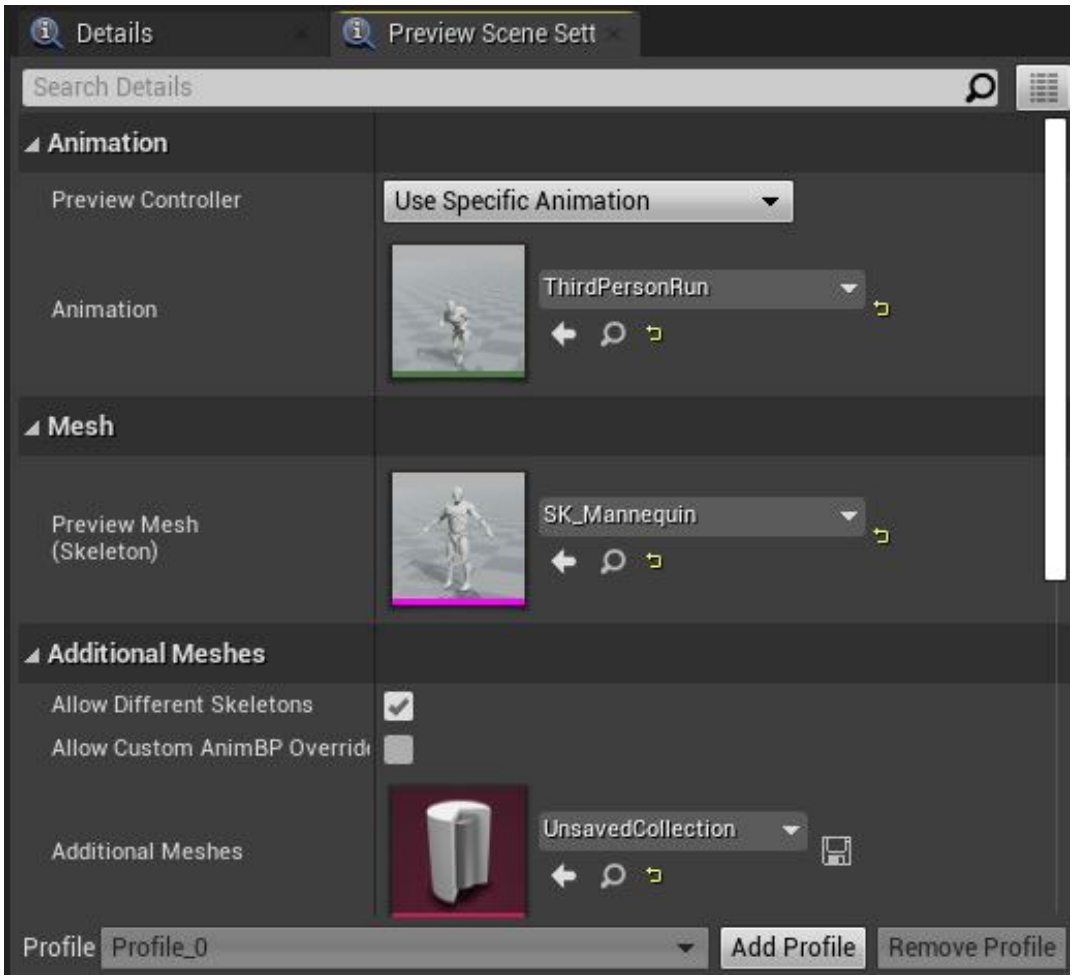

Figure 10.41: The ThirdPersonRun animation applied to our Character in the Preview Scene Settings tab

Now, the mannequin Character is half the previous size and playing the running animation in the preview, as shown in *Figure 10.53*:

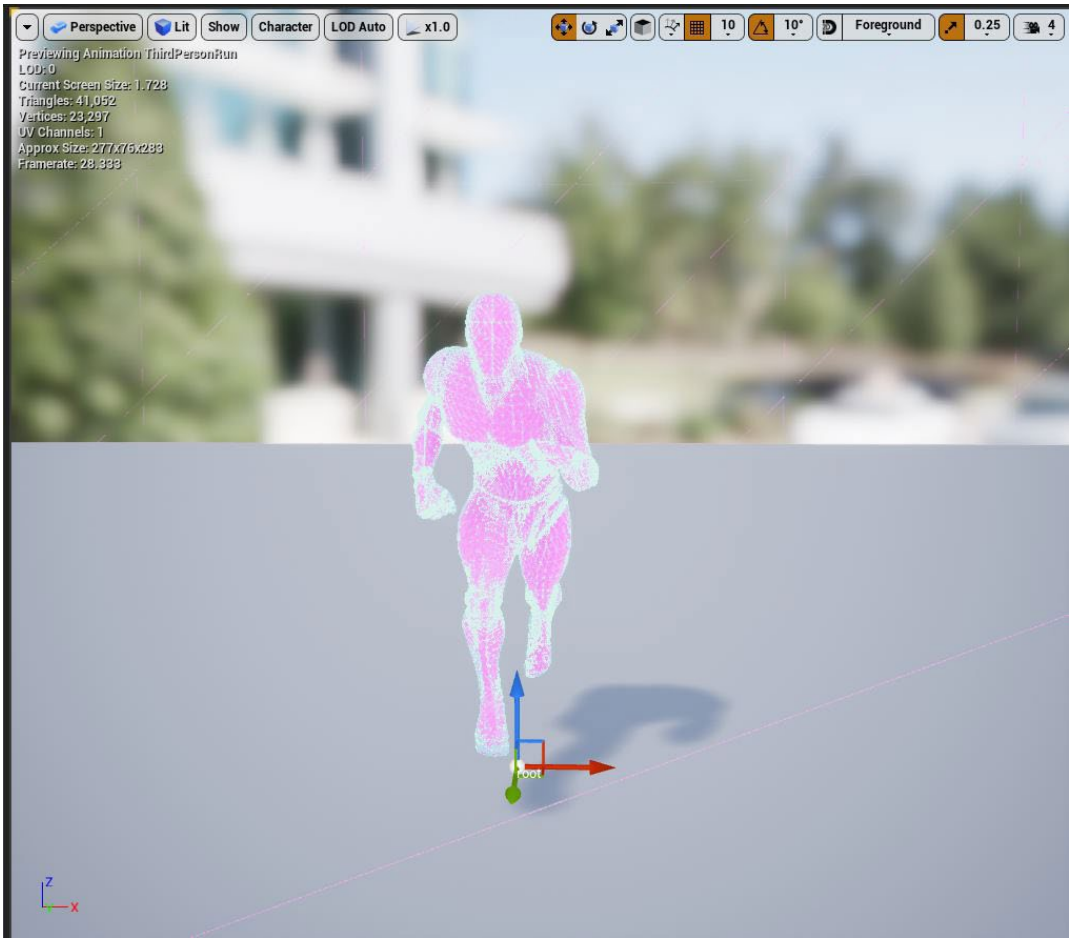

Figure 10.42: The mannequin Character is playing the running animation and is half the previous size

ACTIVITY 10.03: IMPORTING MORE CUSTOM ANIMATIONS TO PREVIEW THE CHARACTER RUNNING

1. As mentioned in this activity, importing the remaining animations will work the same way as it did when importing the **`Idle`** animation in *Exercise 10.03*, *Importing and Setting Up the Character and Animation*. When importing these assets one at a time, you can follow the same steps as we did in the previous exercise, but if you want to import multiple animations at once, be sure to select **`Import All`** as shown in *Figure 10.54*. Otherwise, you will be prompted with the import settings for each individual animation you wanted to import:

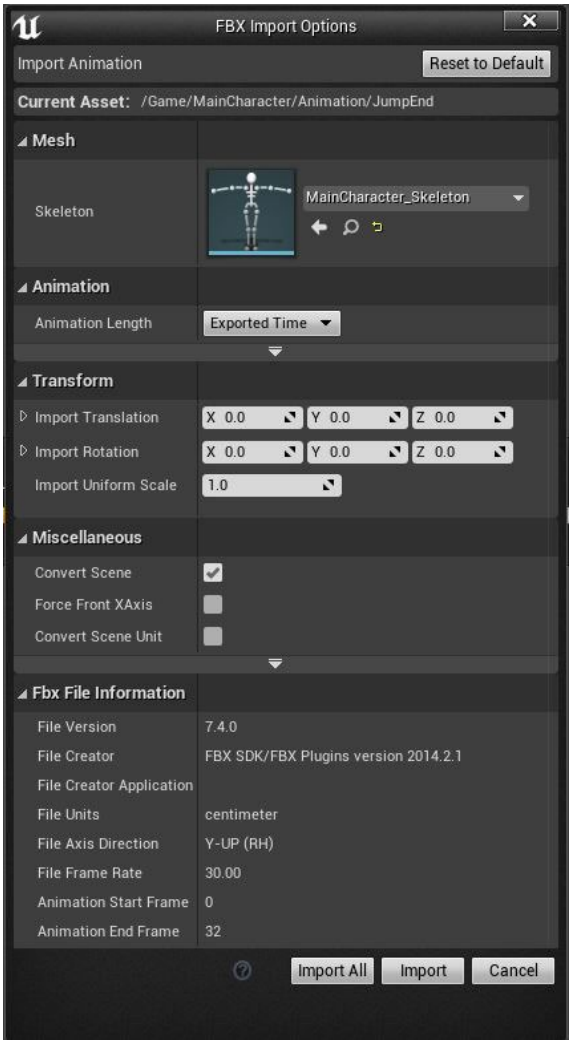

Figure 10.43: Using the Import All option to import all assets with these settings

2. Next, navigate to the **`/MainCharacter/Mesh/`** directory where you will find the **`MainCharacter_Skeleton`** asset. Double *left-click* this asset to open **`Persona`**:

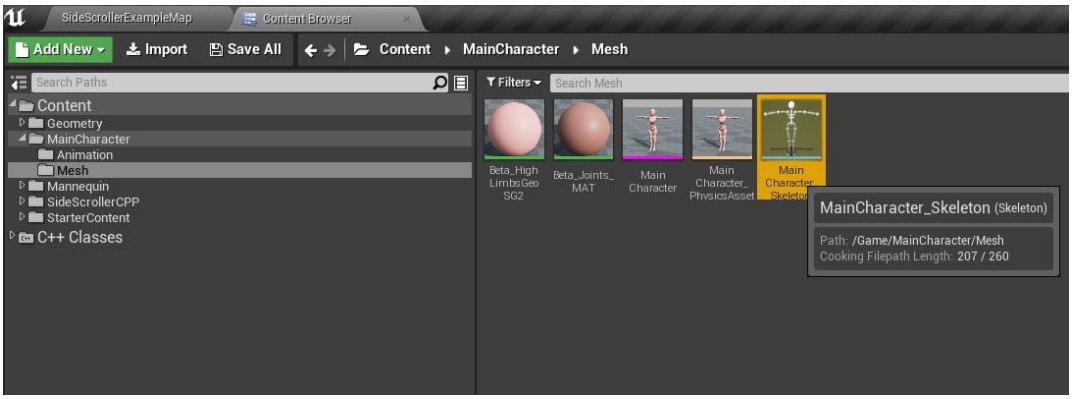

Figure 10.44: The MainCharacter_Skeleton asset you need to access

3. Next to the **`Details`** panel, *left-click* the **`Preview Scene Settings`** tab:

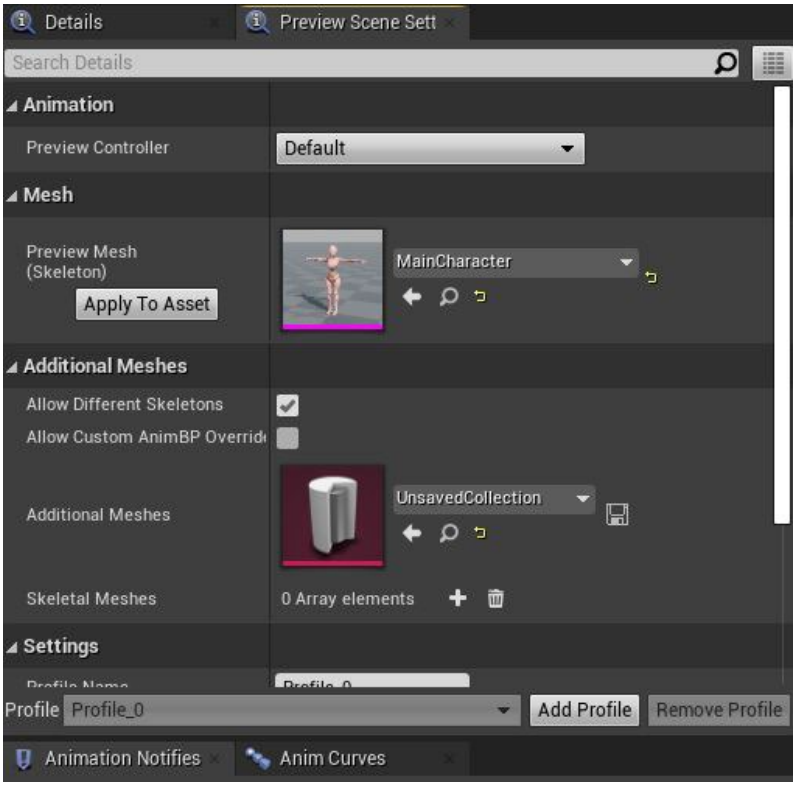

Figure 10.45: The Preview Scene Settings tab

4. Just like in *Activity 10.02*, *Skeletal Bone Manipulation and Animations*, we can set the **`Preview Controller`** parameter to **`Use Specific Animation`** and select the **`Running`** animation:

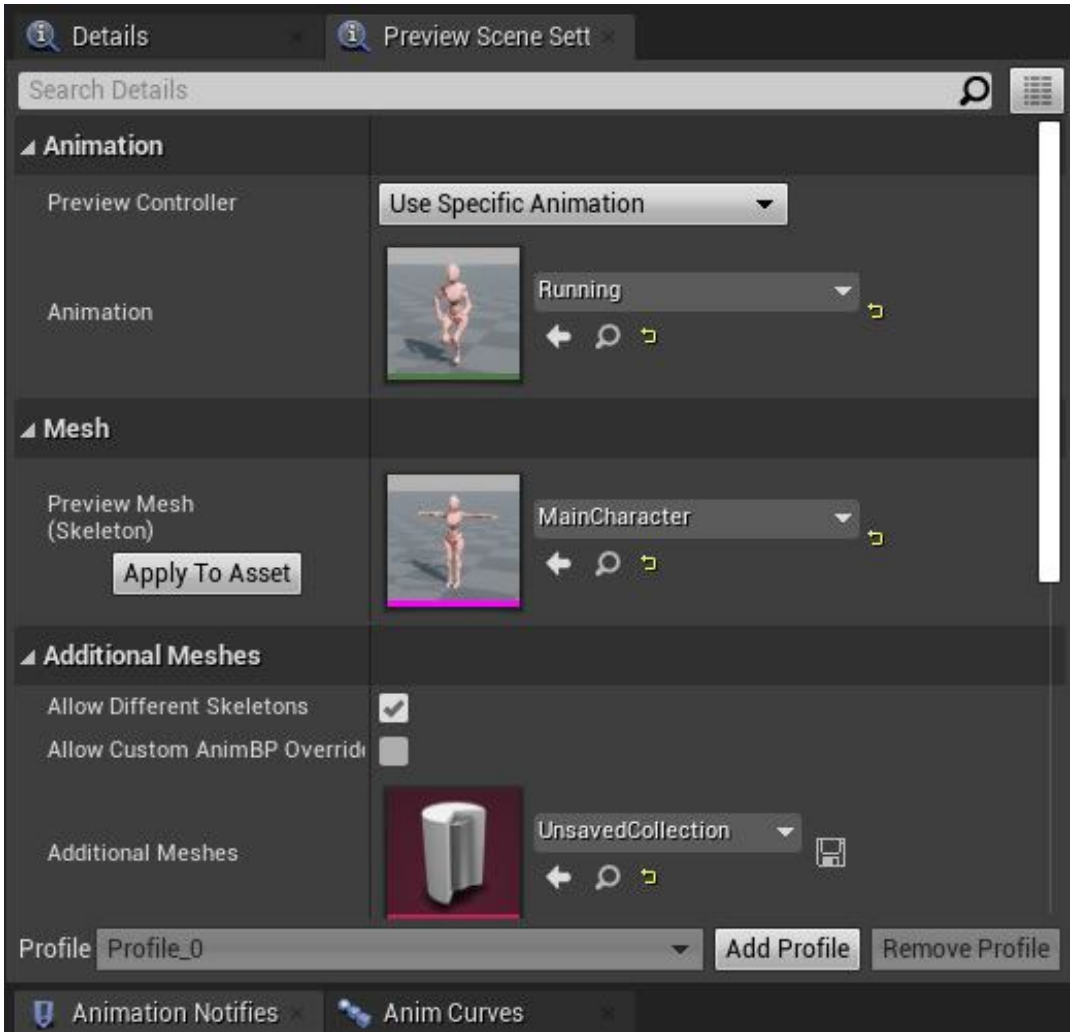

Figure 10.46: The Running animation set as the preview animation for the SuperSideScroller Character

5. Now we can see the custom **`SuperSideScroller`** main Character animating in the preview window:

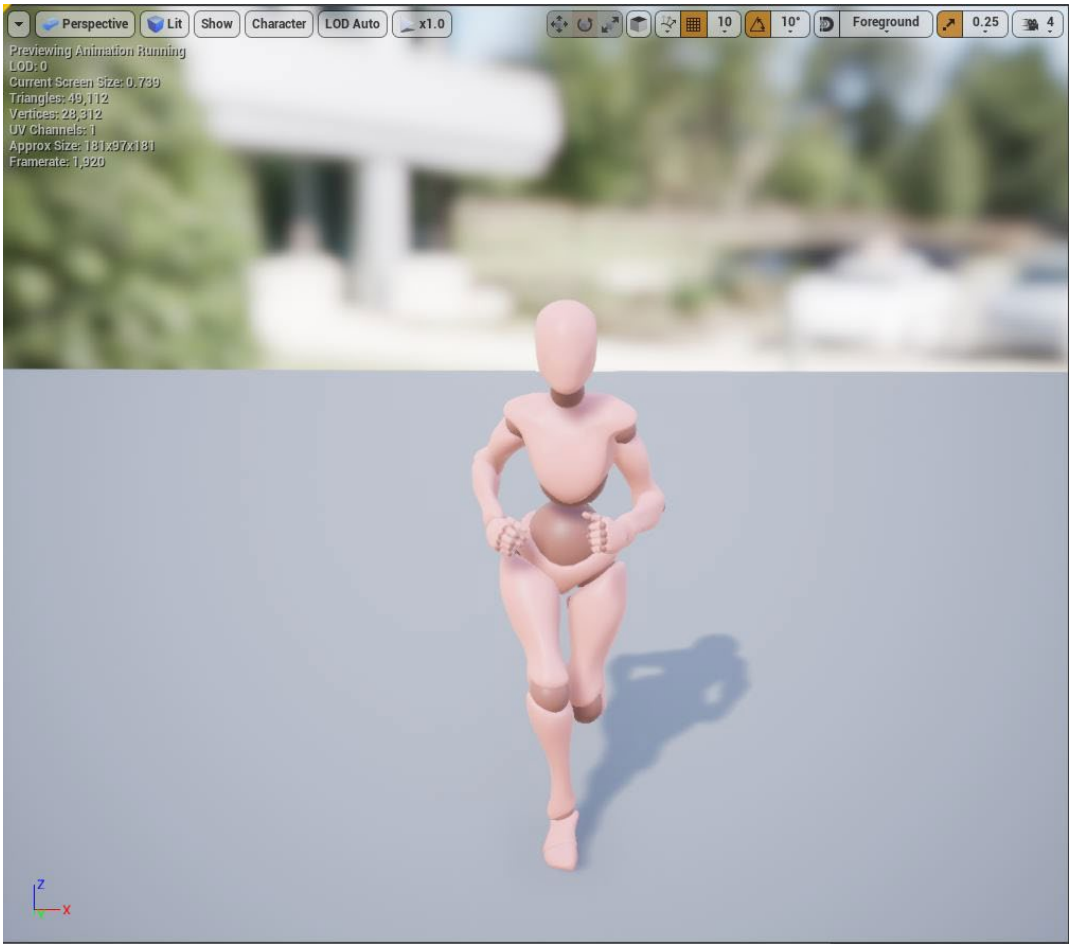

Figure 10.47: The custom animation we imported

CHAPTER 11: BLEND SPACES 1D, KEY BINDINGS, AND STATE MACHINES

ACTIVITY 11.01: ADDING THE WALKING AND RUNNING ANIMATIONS TO THE BLEND SPACE

1. Add the **`Walking`** animation to the Blend Space horizontal axis. Just like in *Exercise 11.01*, *Creating the CharacterMovement 1D Blend Space*, where you added the **`Idle`** animation, all you need to do here is *left-click* and drag the **`Walking`** animation from the **`Asset Browser`** to our timeline. Please refer to *Figure 11.74* to view the **`Walking`** animation added to the axis graph.

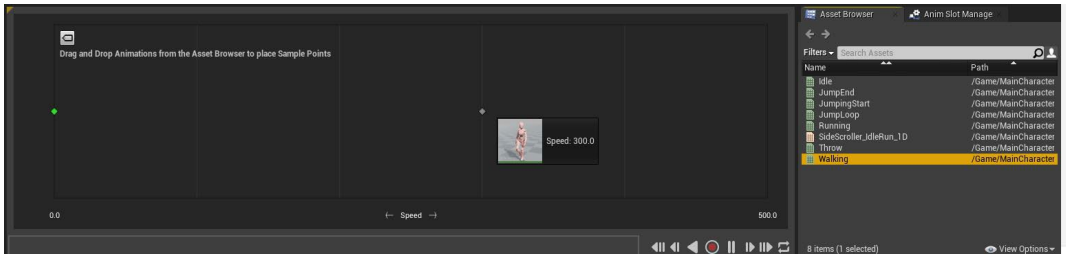

Figure 11.56: Dragging the Walking animation to the grid position 300.0

2. By setting the value for **`Number of Grid Divisions`** to **`5.0f`** and **`Maximum Axis Value`** to **`500.0f`** in the previous exercise, the grid spacing is now split into five 100-unit sections. It is now much easier to adjust the position of this animation to match **`300.0f`** by *left-clicking and* dragging the animation key frame and snapping it to the correct position.

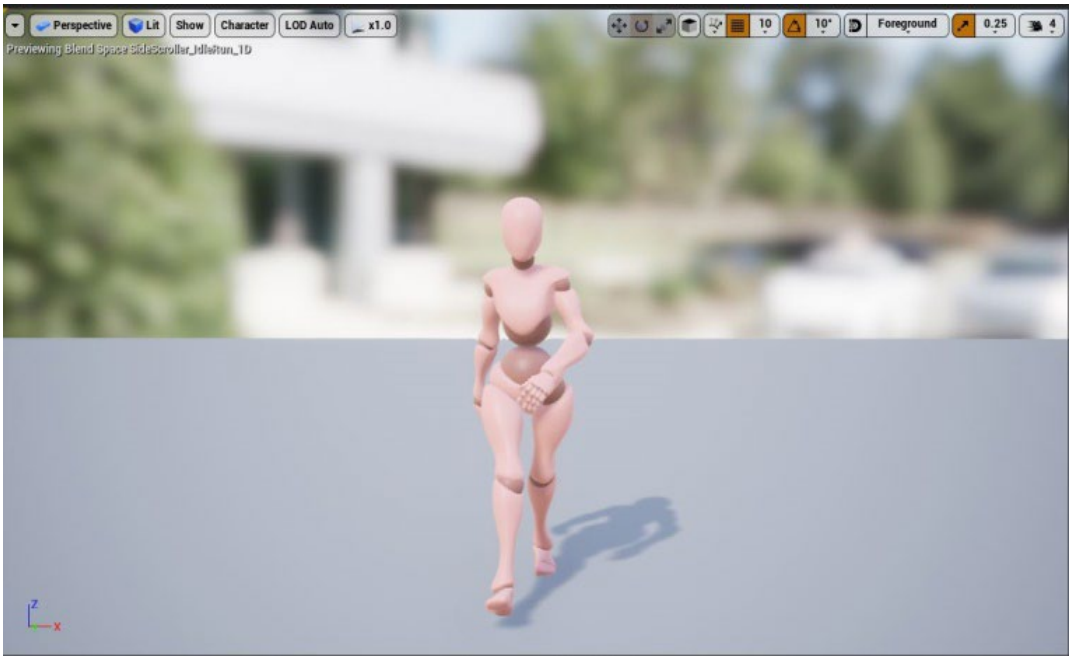

Figure 11.57: When previewing the character animation at a position between the Idle and Walking animations, you can see the Walking animation on the character

3. Next, *left-click* and drag the **`Running`** animation from **`Asset Browser`** to the graph and adjust the key frame to match the value **`500.0f`**, the far right-hand side of the timeline. Please refer to *Figure 11.76* to view the **`Running`** animation added to the axis graph.

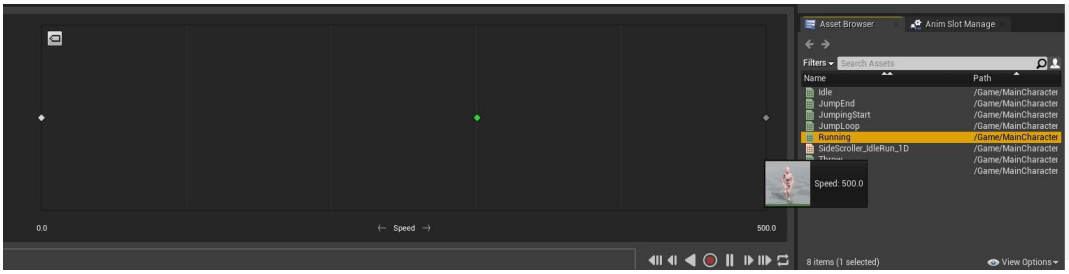

Figure 11.58: Dragging the Running animation to the grid position 300.0

Now, in the preview window, you can see the character performing the **`Running`** animation when the preview speed reaches the full **`500.0f`** value.

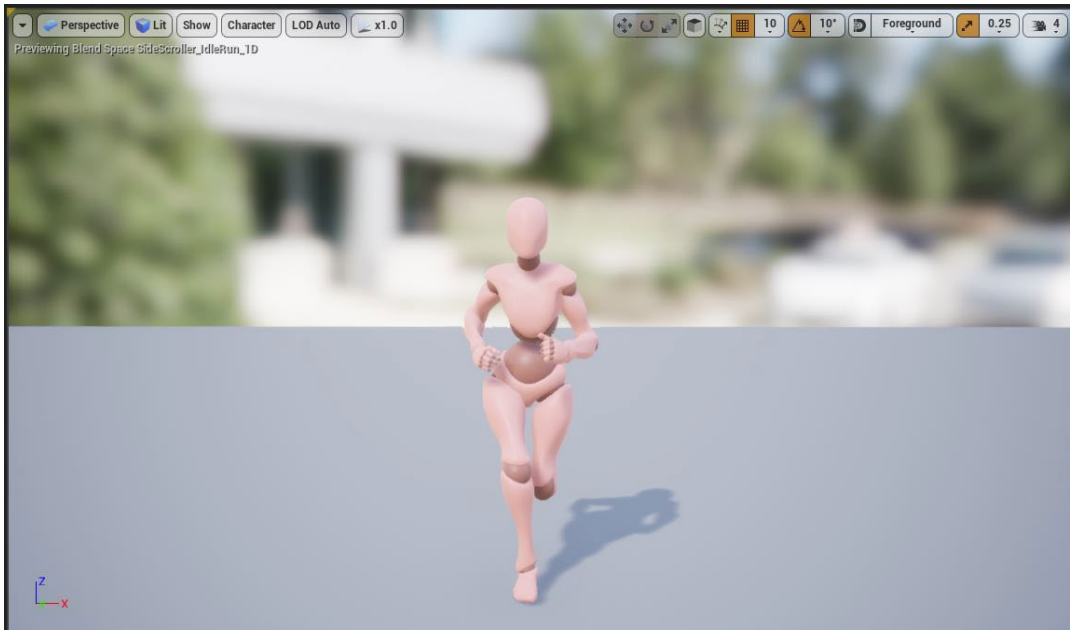

Figure 11.59: Previewing the Running animation in the Blend Space

With the remaining movement animations implemented in the Blend Space, you now have the knowledge and experience of creating a 1D Blend Space asset that is ready for use in the animation blueprint of the player character.

ACTIVITY 11.02: PREVIEWING THE RUNNING ANIMATION IN-GAME

1. First, navigate to the **`/Game/MainCharacter/Blueprints/`** directory and find the **`BP_SuperSideScroller_MainCharacter`** character blueprint.

2. *Double left-click* to open the **`Blueprint`** and select **`CharacterMovementComponent`** from the component list of the player character by *left-clicking* the component. Please refer to *Figure 11.78* to see where the character movement component is located.

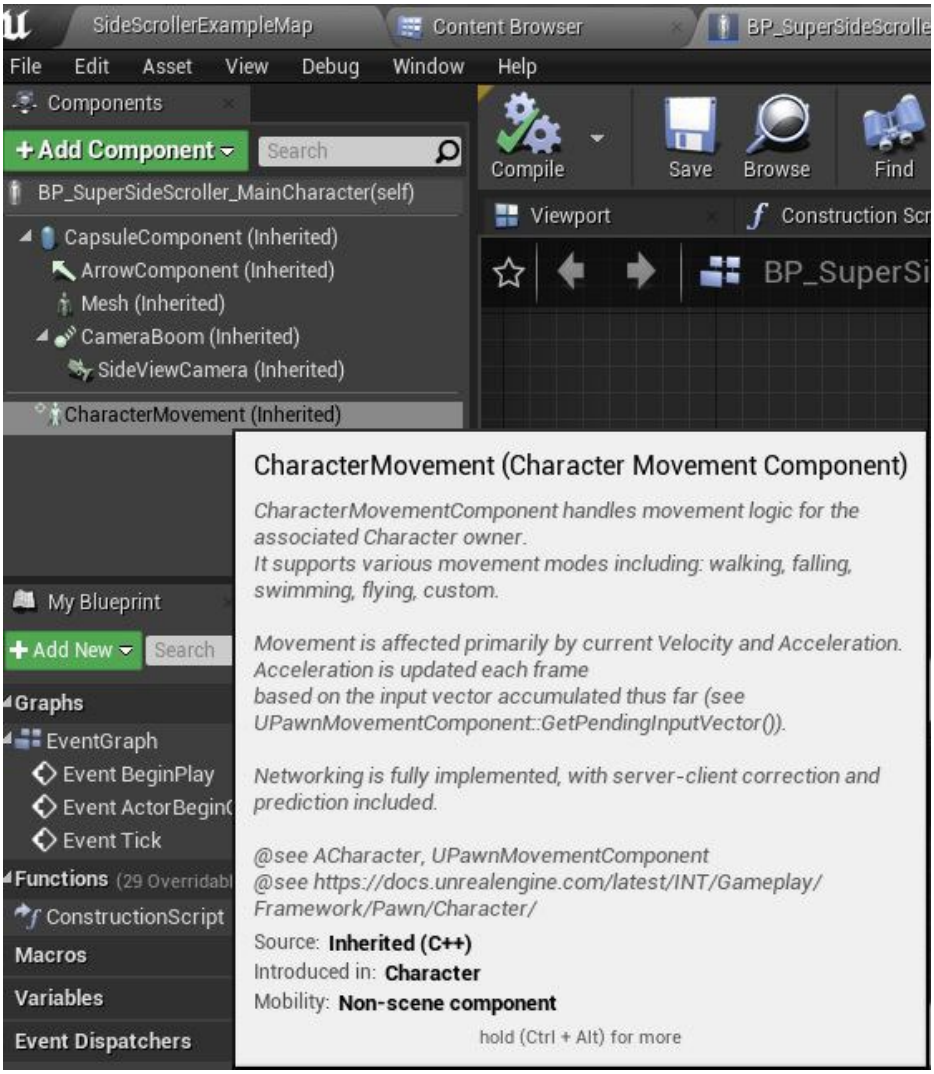

Figure 11.60: The Character Movement Component is found in the Components tab inside the character blueprint

3. In **`Character Movement Component`**, find the **`Character Movement: Walking`** section and then find the **`Max Walk Speed`** parameter.

4. Set this value to **`500.0f`**. Remember that in the Blend Space, the character animation blends from **`Walking`** to **`Running`** when the speed value is between **`300.0f`** and **`500.0f`**. Please refer to *Figure 11.79* to view the updated value for **`Max Walk Speed`**.

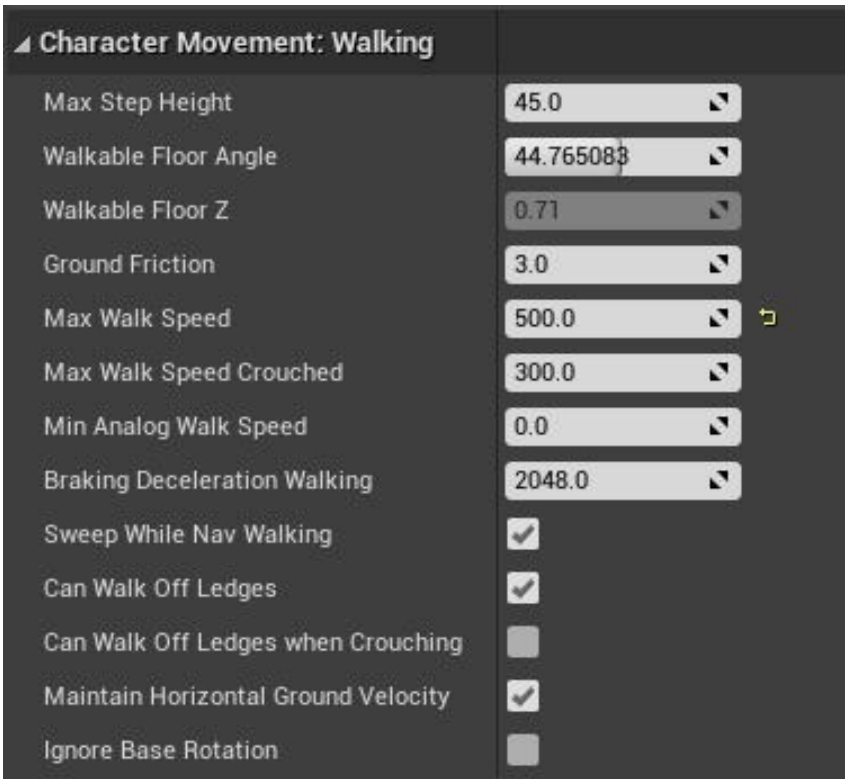

Figure 11.61: Changing the value of Max Walk Speed to 500.0f

5. Compile and save the character blueprint and Play-In-Editor. You will see that the player character now moves faster and blends into the **`Running`** animation, as shown in *Figure 11.80*.

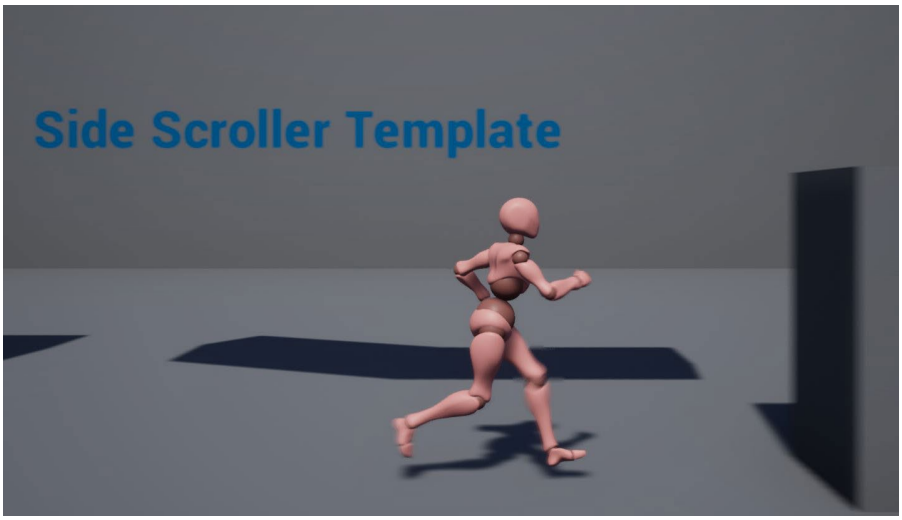

Figure 11.62: The player character can now run

Now that you can see the character movement animation blending correctly to the **`Running`** animation, set **`Max Walk Speed`** back to **`300.0f`** in the character movement component of the player character blueprint.

ACTIVITY 11.03: IMPLEMENTING THE THROWING INPUT

1. First, head to **`Project Settings`** by selecting the **`Edit`** option from the toolbar, and then navigate to the **`Input`** section on the left-hand side. Refer to *Figure 11.81*.

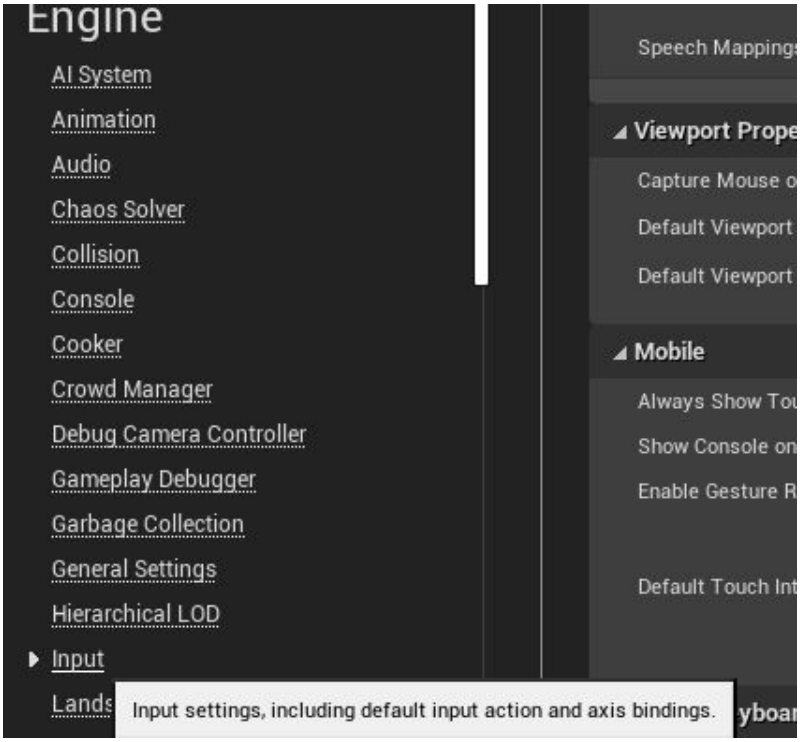

Figure 11.63: Under the Engine category is the Input section where we can add new key bindings, among other things

2. Add a new **`Action Mapping`** parameter called **`ThrowProjectile`** by left-clicking the **+** symbol next to **`Action Mappings`**. This mapping should have the keys **`Left Mouse Button`** and **`Gamepad Right Trigger`**. Remember that you can add more than one key under each mapping by left-clicking the **+** symbol next to the newly added mapping, as shown in *Figure 11.82*.

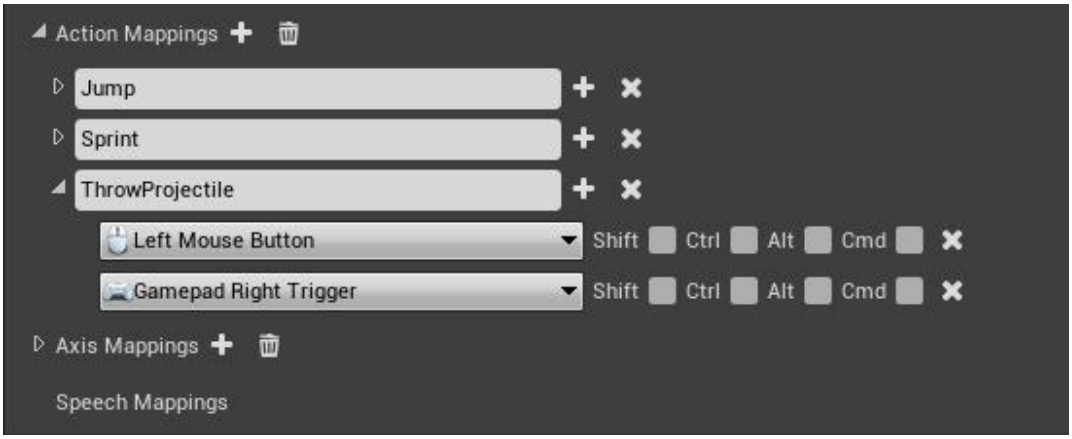

Figure 11.64: Mapping the ThrowProjectile input to Left Mouse Button and Gamepad Right Trigger

With the action mapping added, return to Visual Studio and go into the **`SuperSideScroller_Player`** class header file to add the declaration of the new function you will use to throw the projectile.

3. Inside the **`SuperSideScroller_Player.h`** header file, declare the **`ThrowProjectile()`** function under the **`Protected`** access modifier. This will be a void function because it will not return anything and it will also not take any arguments as shown here:

```
//ThrowProjectile
void ThrowProjectile();
```

4. With the function declared, define the function inside the **`SuperSideScroller_Player.cpp`** source file as follows:

```
void ASuperSideScroller_Player::ThrowProjectile()
{
}
```

For now, you will not code any logic relating to throwing the projectile or animation because you have not yet created the projectile itself; this will happen in *Chapter 13*, *Enemy Artificial Intelligence*. Instead, here you will add a **`UE_LOG`** function call to the **`ThrowProjectile`** function to log a message in the **`Output Message`** log to let you know that the function is being called.

5. **`UE_LOG`** requires a few parameters. The first is the category, and in this case, you will use **`LogTemp`** to let Unreal know that this message should be put into a temporary log file. Next, is the **`Log Level`** that this message will be labeled as; in this case **`Warning`** will log the message in a yellow text labeled as **`Warning`**. Finally, the last parameter is a string that holds the message that will be displayed in the log. For now, add the string **`TEXT("THROW PROJECTILE")`**:

```
void ASuperSideScroller_Player::ThrowProjectile()
{
  UE_LOG(LogTemp, Warning, TEXT("THROW PROJECTILE!"));
}
```

With the function done, now you can set up the binding of the **`ThrowProjectile`** action to the **`ThrowProjectile`** function you just created in the **`SetupPlayerInputComponent()`** function.

6. Add the binding for **`ThrowProjectile`**, as shown in the following code snippet:

```
//Bind pressed action ThrowProjectile to your ThrowProjectile
  function
PlayerInputComponent->BindAction("ThrowProjectile", IE_Pressed,
  this, &ASuperSideScroller_Player::ThrowProjectile);
```

Again, you will use **`IE_Pressed`**, but instead, you will bind to the new function, **`&SuperSideScroller_Player::ThrowProjectile`**, and the mapping is labeled **`ThrowProjectile`**.

7. Return to the editor to recompile and hot-reload. Now, if you select **`PIE`** and use **`Left-Mouse Button`** or **`Gamepad Right Trigger`**, you will see the **`Output Log`** message in yellow appear, as shown in *Figure 11.83*.

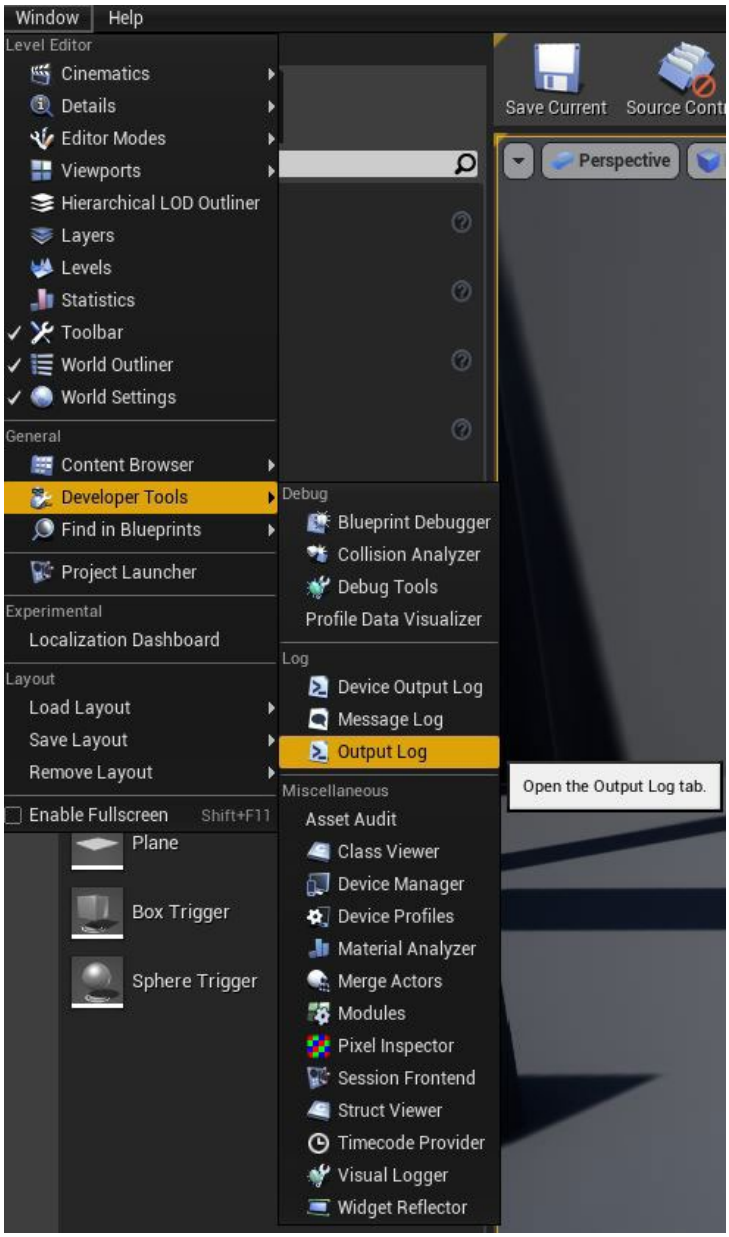

Figure 11.65: The output log can be accessed from the Window toolbar option, and can be found under Developer Tools

ACTIVITY 11.04: FINISHING THE MOVEMENT AND JUMPING STATE MACHINE

1. First, within the **`Movement`** state machine, *left-click* and drag from the **`JumpLoop`** state and select the **`Add State`** option from the context-sensitive drop-down menu. Please refer to *Figure 11.84*.

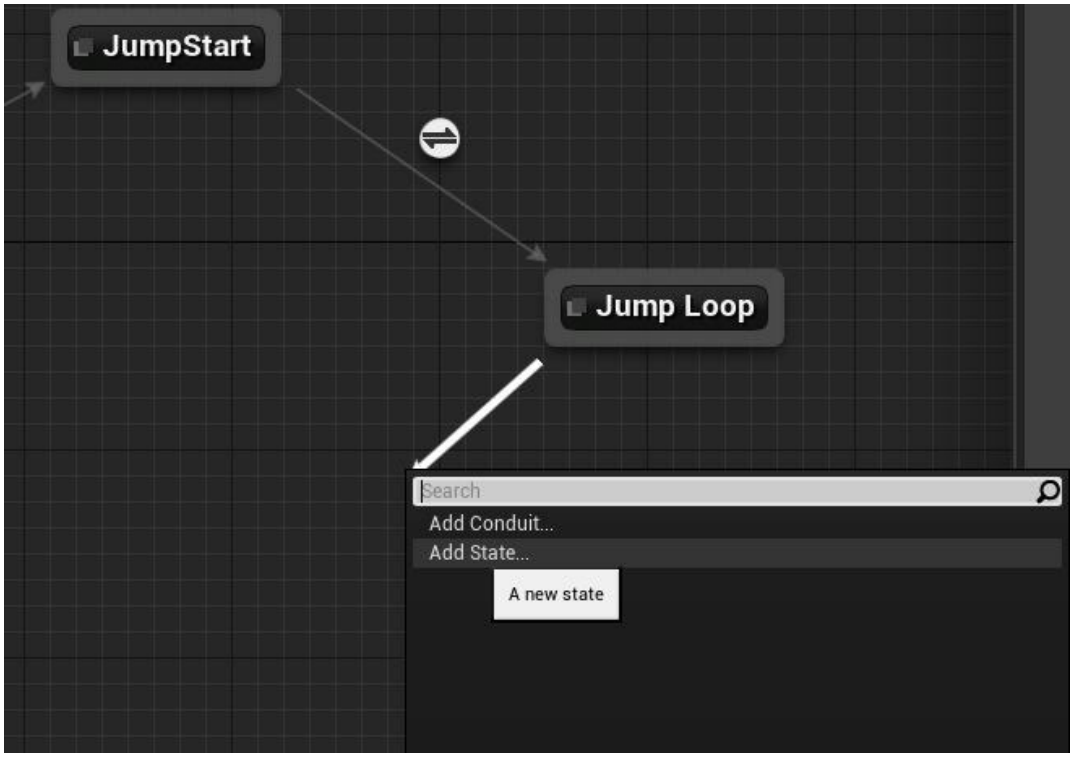

Figure 11.66: Adding a new state called JumpEnd

2. Name this state **`JumpEnd`**. When creating a new state, Unreal Engine automatically creates a **`Transition Rule`** between the two states that you can use to create the **`Transition Rule`** to apply. *Figure 11.85* shows the **`JumpEnd`** state and the automatically generated **`Transition Rule`**.

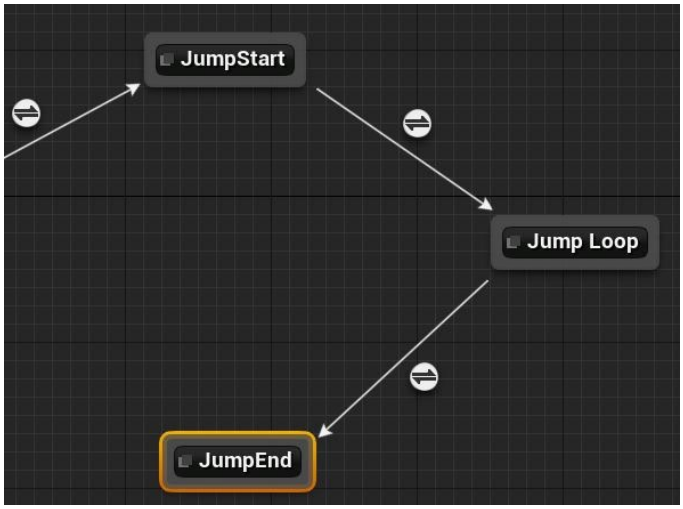

Figure 11.67: The final state of the JumpEnd state machine

3. Now, *double left-click* on the **`JumpEnd`** state to enter its graph. *Left-click* and drag the **`JumpEnd`** animation from **`Asset Browser`** and connect it to **`Output Animation Pose`** of the state. Please refer to *Figure 11.86* and *Figure 11.87* to view how to perform this step.

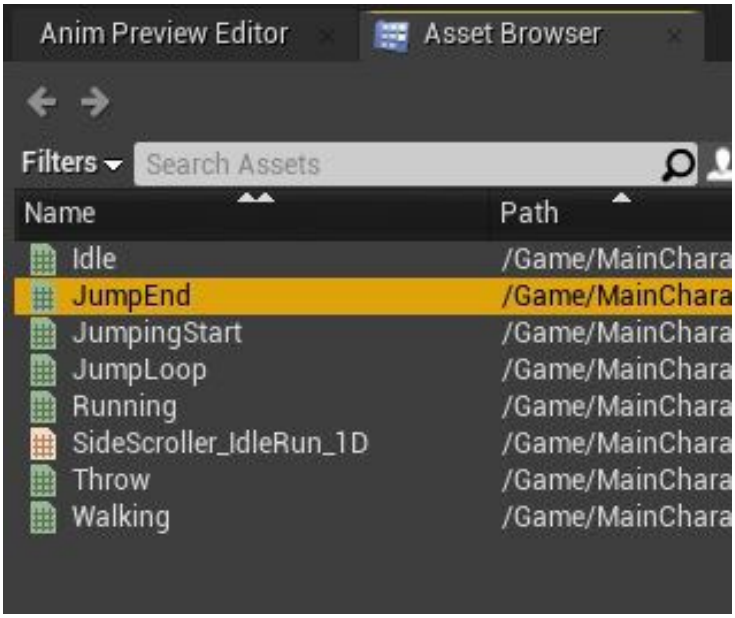

Figure 11.68: Make sure that the JumpEnd animation is selected in Asset Browser before dragging it into the new state

4. Connect this **JumpEnd** animation with the **Result** pin of **Output Animation Pose**:

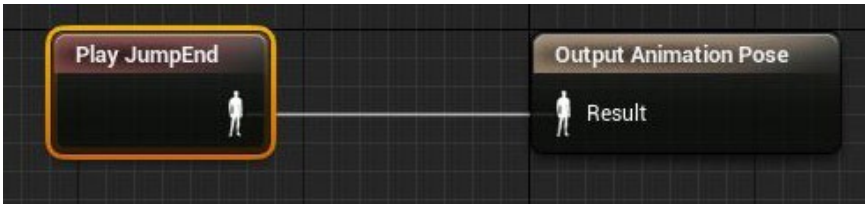

Figure 11.69: The JumpEnd animation connected to the Output Animation Pose of the new state

5. Before moving on to the **Transition Rule** between the **JumpLoop** and **JumpEnd** states, you need to update the parameters of the **JumpEnd** animation. The settings are shown in *Figure 11.88*, but you will remember that these settings are identical to the **JumpStart** animation. The **Loop Animation** parameter needs to be **False** and the **Play Rate** parameter needs to be set to **3.0** in order for the transition between the states to be as smooth as possible.

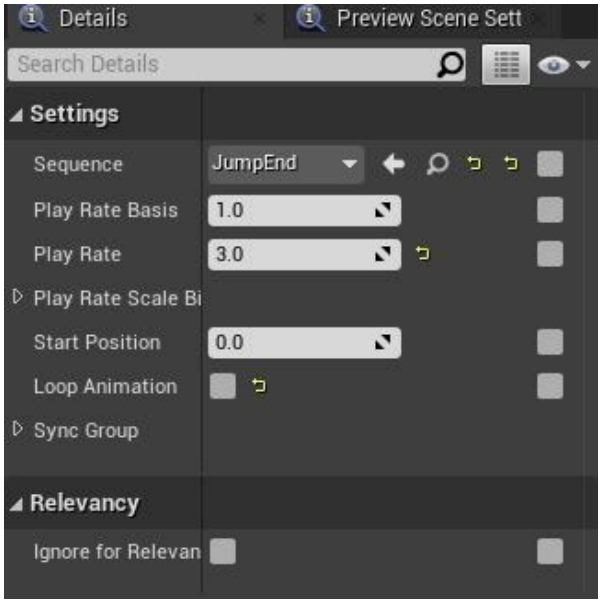

Figure 11.70: Due to the slowness of the JumpEnd animation, increasing the play rate will result in a smoother jumping animation overall

Now, you can add the **Transition Rule** between the **JumpLoop** and **JumpEnd** states.

6. Now, *double left-click* to enter the graph. In this case, you want this transition to occur when the player character is no longer in the air. You can use the **`bIsInAir`** variable that you created in the last exercise to do this. You just need to use the **`NOT`** node to ensure that when the player is **`NOT`** in the air, the **`Transition Rule`** returns **`True`**, and the state transition can take place. Please refer to *Figure 11.89* to see the final result of this **`Transition Rule`**.

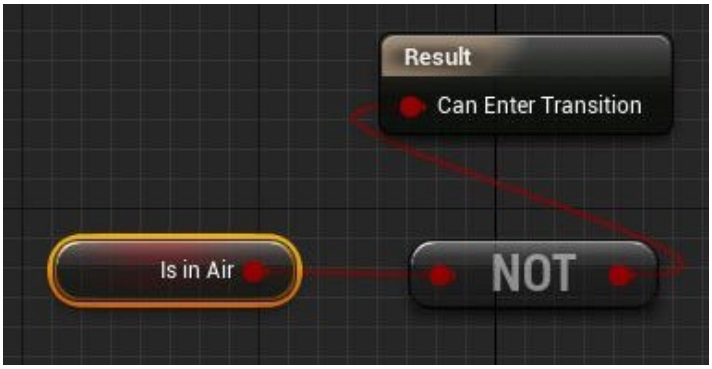

Figure 11.71: Transition to the JumpEnd animation when the character is no longer in the air

7. With the first **`Transition Rule`** complete, return to the overview graph of the **`Movement`** state machine. *Left-click* and drag from the **`JumpEnd`** state to the **`Movement`** state in order to create the final **`Transition Rule`** of your state machine. Refer to *Figure 11.90* to see this.

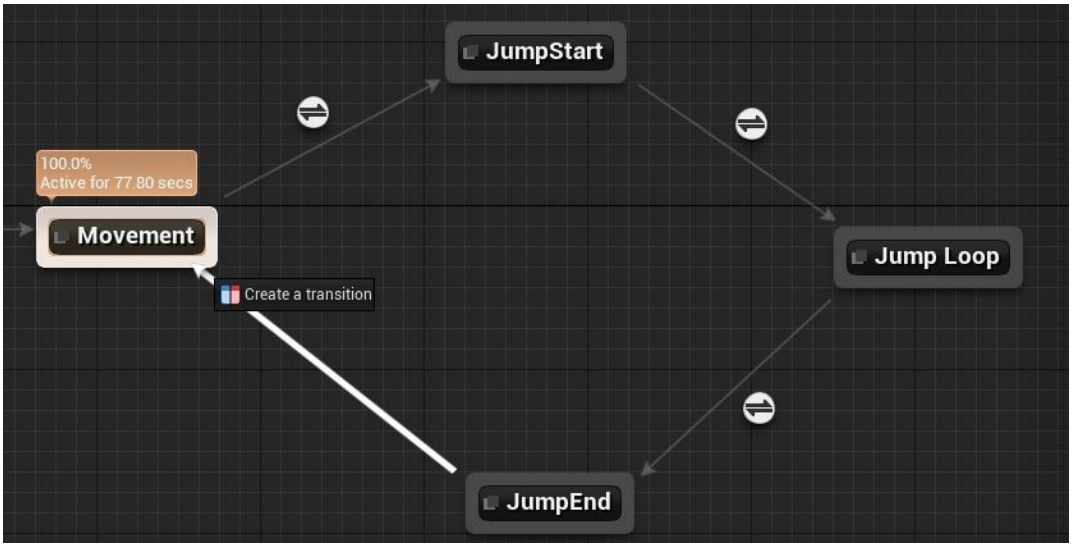

Figure 11.72: Adding a new transition between the JumpEnd and Movement states

8. *Double left-click* on this new **`Transition Rule`** to open its graph. For this final **`Transition Rule`**, you will use similar logic that you used to transition from the **`JumpStart`** to **`JumpLoop`** states, but this time using the **`Time Remaining Ratio`** of the **`JumpEnd`** animation.

   Make sure that the **`JumpEnd`** animation is selected in **`Asset Browser`** and then *right-click* to search for **`Time Remaining Ratio`** in the context-sensitive drop-down menu, as shown in *Figure 11.91*.

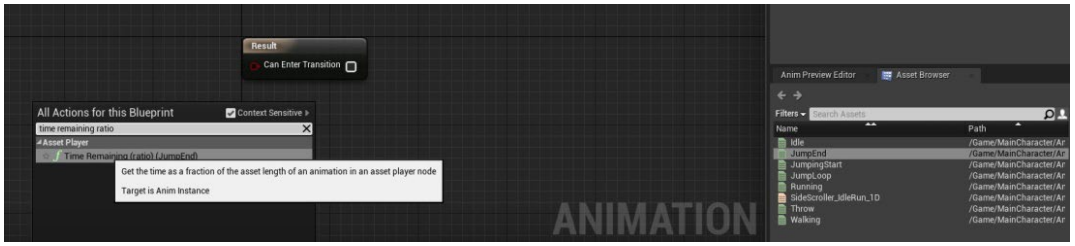

Figure 11.73: How to find the Time Remaining Ratio function

9. Now, use the **`Less Than`** comparative operator node to compare the value returned from the **`Time Remaining Ratio`** function with the value **`0.3f`**, or 30%. Please refer to *Figure 11.92* to view this comparison logic. This means that once the remaining time ratio is less than 30%, the state machine will transition back to the **`Movement`** state and the player will move around as normal.

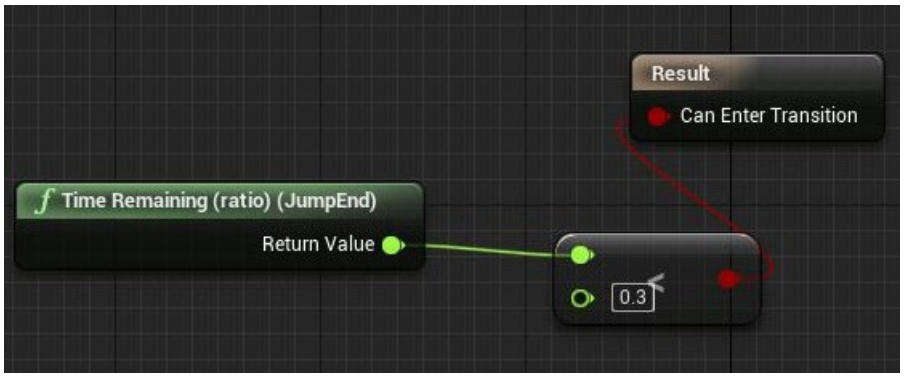

Figure 11.74: You will need to know how much time is left in the JumpEnd animation before transitioning back to the default movement

10. With the final Transition Rule in place, your **`Movement`** state machine is complete. Please refer to *Figure 11.93* for a final view of the state machine.

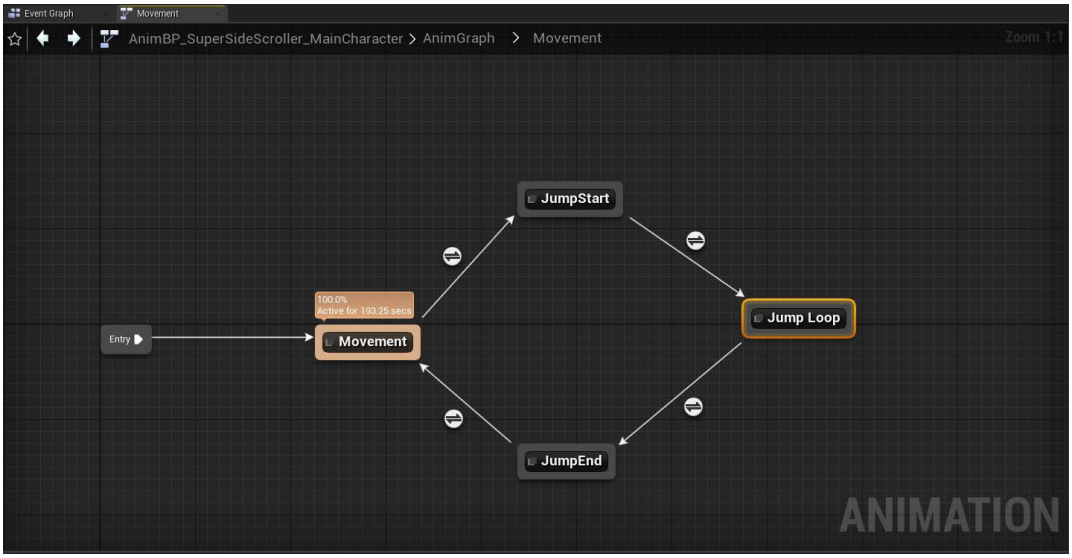

Figure 11.75: The final setup of your state machine

Now, the player character can idle, walk, sprint, and jump around the level, with all of the required animations being supported and playing correctly. Refer to *Figure 11.94* for an example of what this looks like.

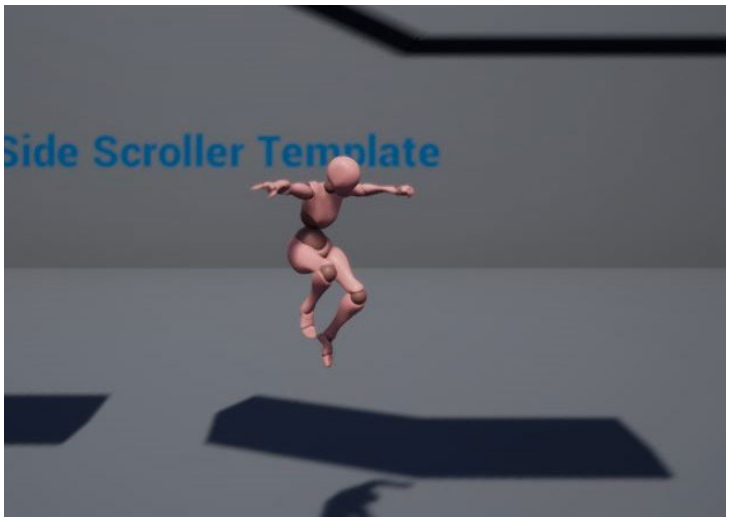

Figure 11.76: Now, the player character can idle, walk, sprint, and jump

CHAPTER 12: ANIMATION BLENDING AND MONTAGES

ACTIVITY 12.01: UPDATING BLEND WEIGHTS

Let's begin by updating the **`Blend Weights`** parameters of the **`Layered blend per bone`** node in the Animation Blueprint:

1. Navigate to the **`AnimBP_SuperSideScroller_MainCharacter`** Animation Blueprint in the **`MainCharacter/Blueprints`** directory and open it.
2. In **`AnimGraph`**, find the **`Layered blend per bone`** node you implemented in the previous exercise. The value currently implemented here is **`1.0f`**, which means there is a full blending of the additive **`Throw`** animation on top of the base movement pose.
3. If you want no blending at all, the value to use is **`0.0f`**. Apply this value to the **`Blend Weights 0`** input parameter, as shown in *Figure 12.60*:

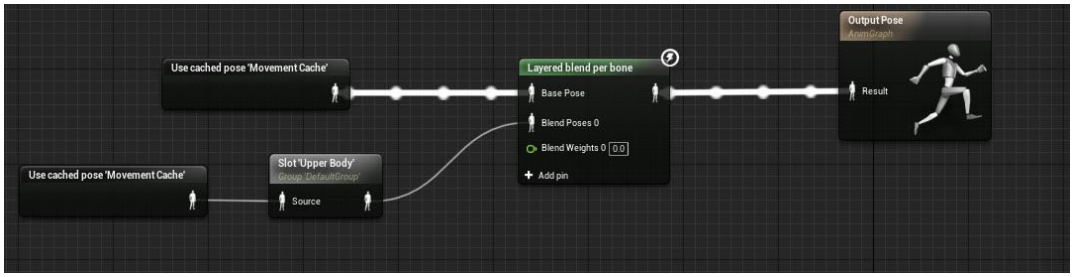

Figure 12.38: With a weight value of 0.0f, the additive pose will have absolutely no influence on the character's skeletal mesh

Now, if you use **`PIE`**, the character will not animate at all when you use the *left mouse button* to throw a projectile. To explore this further, follow the next steps to set up a **`50%`** blend between the movement base pose and the throw additive pose.

4. Change the value of the **`Blend Weights 0`** parameter to **`0.5f`**, as shown in *Figure 12.61*:

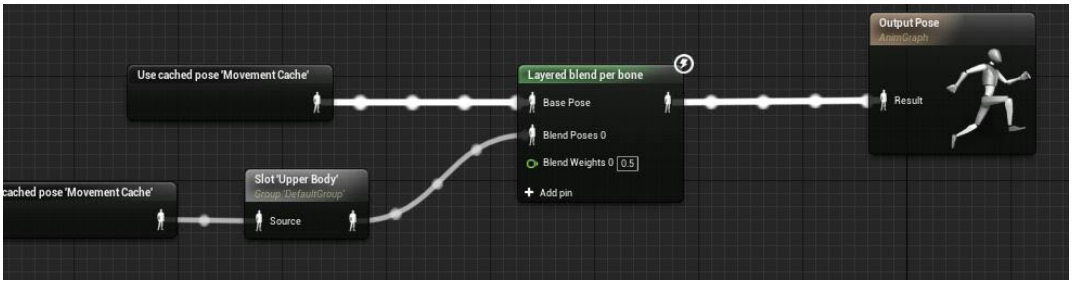

Figure 12.39: The value will now blend these poses at an influence value of 50%

5. Now, if you use **`PIE`**, the **`Throw`** animation will play, but it will only blend at a value of 50%. Please refer to *Figure 12.62* to see an example of what this looks like:

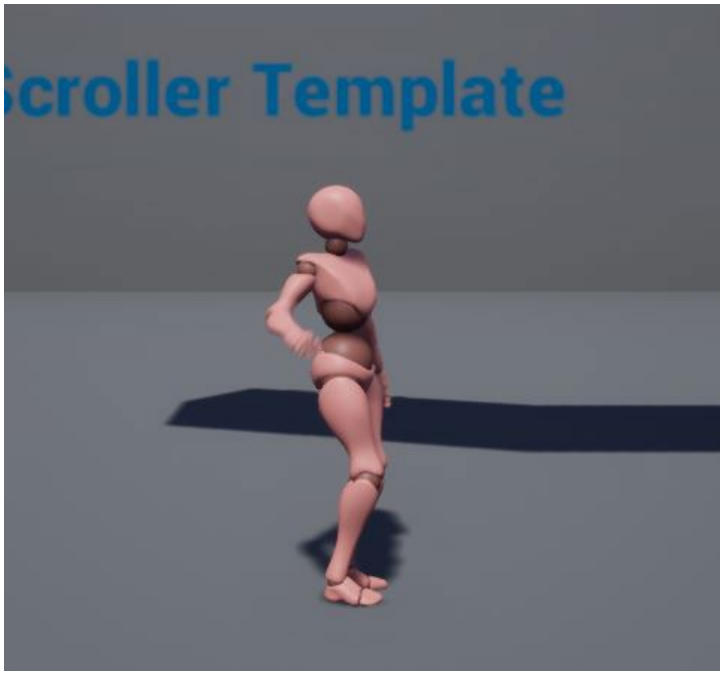

Figure 12.40: Now, the Throw animation additive pose will only be blended at an influence of 50%

Before continuing with this activity, please return the **`Blend Weights 0`** parameter to a value of **`1.0f`**.

Next, you need to update the settings of the **`Layered blend per bone`** node so that the character's entire skeletal mesh is influenced by the throw additive pose, instead of just the upper body.

6. First, navigate to the **`MainCharacter_Skeleton`** asset in the **`MainCharacter/Mesh/`** directory. Now, *double-left-click* to open this asset.

7. Looking at the hierarchy of the bones for this skeleton, you can see that the **`Hips`** bone is the **`root`** bone and the children of this bone include the entire character body, including the legs. Please refer to *Figure 12.63* to see how the hierarchy is structured for the player character:

Figure 12.41: The bone named Hips will influence the entire skeleton of the character

8. Back in the Animation Blueprint, select the **`Layered blend per bone`** node and update its settings to use the **`Hips`** bone for **`Bone Name`**. *Figure 12.64* shows the updated settings:

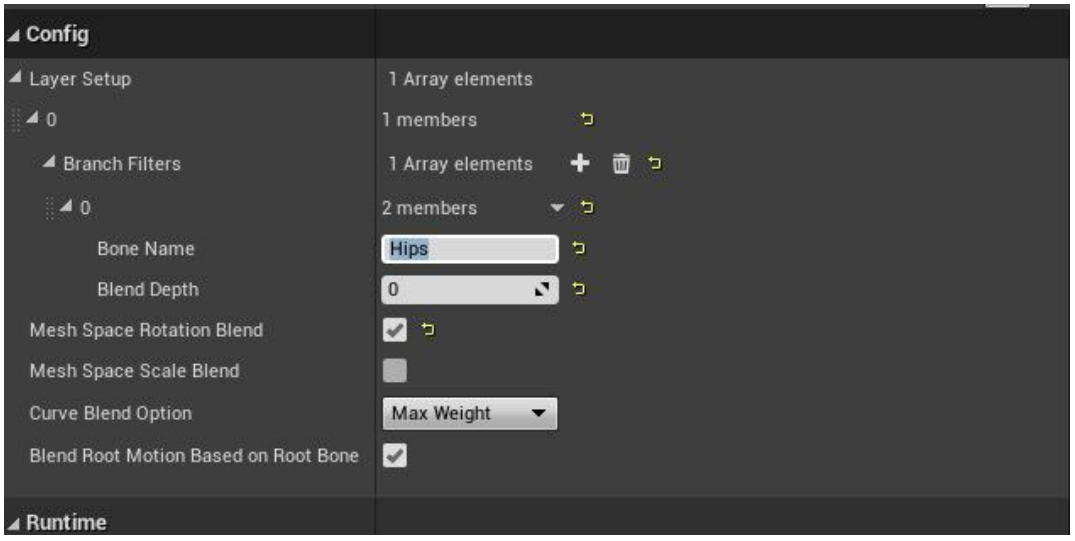

Figure 12.42: Using the Hips bone name here tells Layered blend per bone to allow the Throw animation to influence the entire skeleton

9. Now, if you use **`PIE`** when you use the *left mouse button* to throw, the animation for the throw completely overrides the **`Movement`** animations. This results in the player incorrectly animating the legs of the character while moving and throwing simultaneously. *Figure 12.65* demonstrates this result:

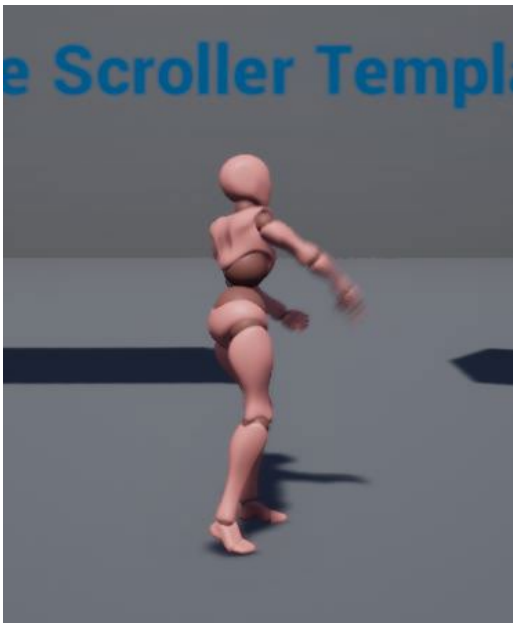

Figure 12.43: The Throw animation completely overrides the character movement animation

To finish this activity, you need to update the layer setup of the **`Layered blend per bone`** node to include a new branch filter that would allow the character's left leg to animate correctly when moving and throwing at the same time.

10. First, let's look at the skeleton bone hierarchy again to know which bone to reference. Navigate to the **`MainCharacter_Skeleton`** asset in the **`MainCharacter/Mesh/`** directory. Now, *double-left-click* to open this asset.

11. In the skeleton bone hierarchy, there is the **`LeftUpLeg`** bone. This bone includes its own hierarchy that influences the entire left leg and is the perfect candidate to use in the **`Layered blend per bone`** node. Please refer to *Figure 12.66* to preview the hierarchy of this bone:

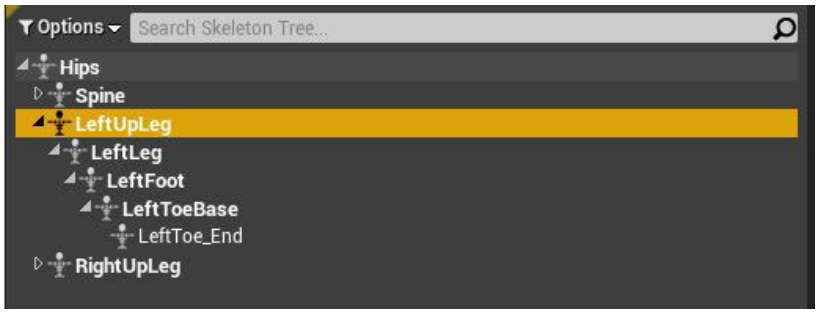

Figure 12.44: The LeftUpLeg bone includes the entire hierarchy of the character's left leg

12. Back in **`AnimBP_SuperSideScroller_MainCharacter`**, select the **`Layered blend per bone`** node to access its settings. In the settings of the **`Layered blend per bone`** node, add a new branch filter by *left-clicking* on the **+** button to create a new element.

13. For this new **`Branch Filter`** element, set **`Bone Name`** to **`LeftUpLeg`** and **`Blend Depth`** to **`-1`** so that the additive throw pose completely ignores this bone name and its children. Please refer to *Figure 12.67* to make sure the settings are correct:

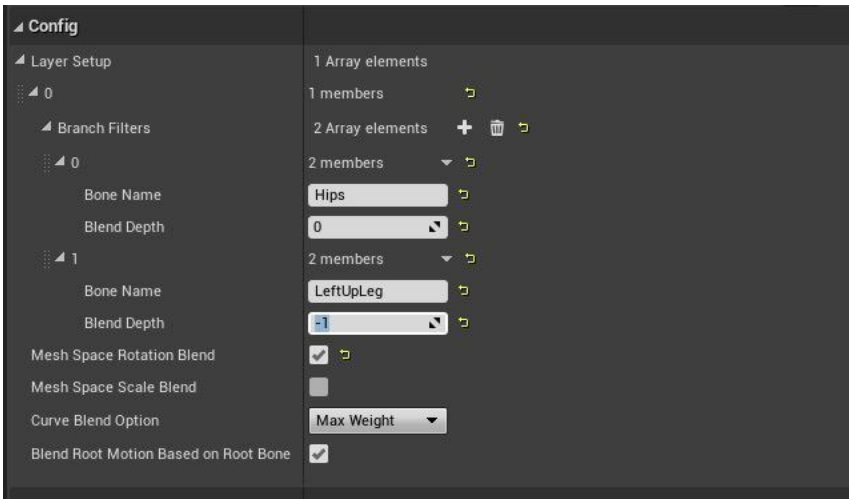

Figure 12.45: Now, the Layered blend per bone node will influence the entire body, but ignore the LeftUpLeg bone and its children

14. Now, using **`PIE`**, when you use the *left mouse button* to throw, the animation for the **`Throw`** only overrides the movement animations of the right leg. This results in the player correctly animating the left leg of the character while moving and throwing simultaneously, but the right leg is still completely influenced by the **`Throw`** animation. *Figure 12.68* demonstrates this result:

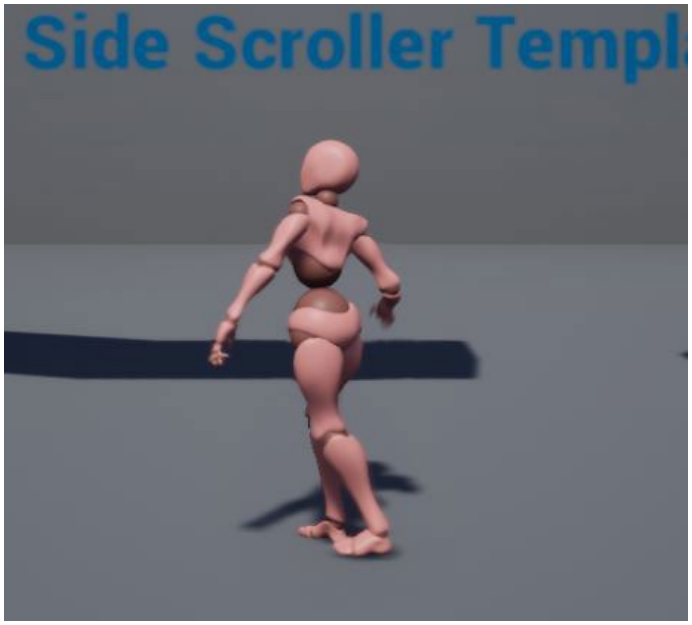

Figure 12.46: Now, the player character will animate the movement pose correctly, but only the left leg

> **NOTE**
>
> Before concluding this activity, please return the **`Layered blend per bone`** settings to the values you set at the end of *Exercise 12.05*, *Blending Animation with the Upper Body Anim Slot*. If you do not return these values back to their original settings, the animation results in upcoming exercises and activities in the next chapters will not be the same. You can either set back the original values manually or refer to the file with these settings at the following link: https://packt.live/2GKGMxM.

CHAPTER 13: ENEMY ARTIFICIAL INTELLIGENCE

ACTIVITY 13.01: CREATING A NEW LEVEL

1. First, create a new level by navigating to the **`File`** option at the top-left corner of the Unreal Engine 4 editor and *left-clicking* it to find the option for **`New Level`**, as shown in the following screenshot. *Left-click* this option to start creating a new map:

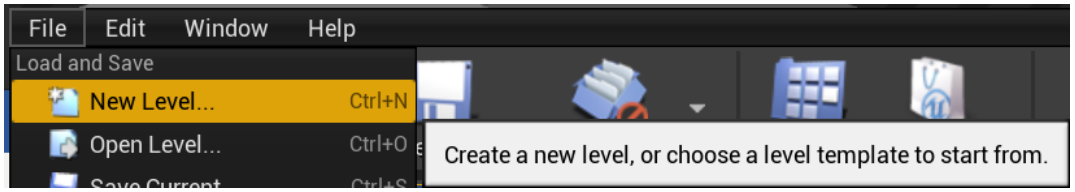

Figure 13.41: The remainder of this project can be done in either SideScrollerExampleMap or in the new map that you create

2. When creating a new level or map, you will be prompted with the **`New Level`** dialogue window, where you will be presented with different templates to start with. These templates allow you to have a minimal map that includes important actors such as **`lights`**, **`skyboxes`**, **`floors`**, and so on. If you want, the option for an **`Empty Level`** will give you an absolutely empty canvas to begin creating your level. The **`SuperSideScroller.umap`** level provided was created using the **`TimeOfDay`** template shown in the following screenshot:

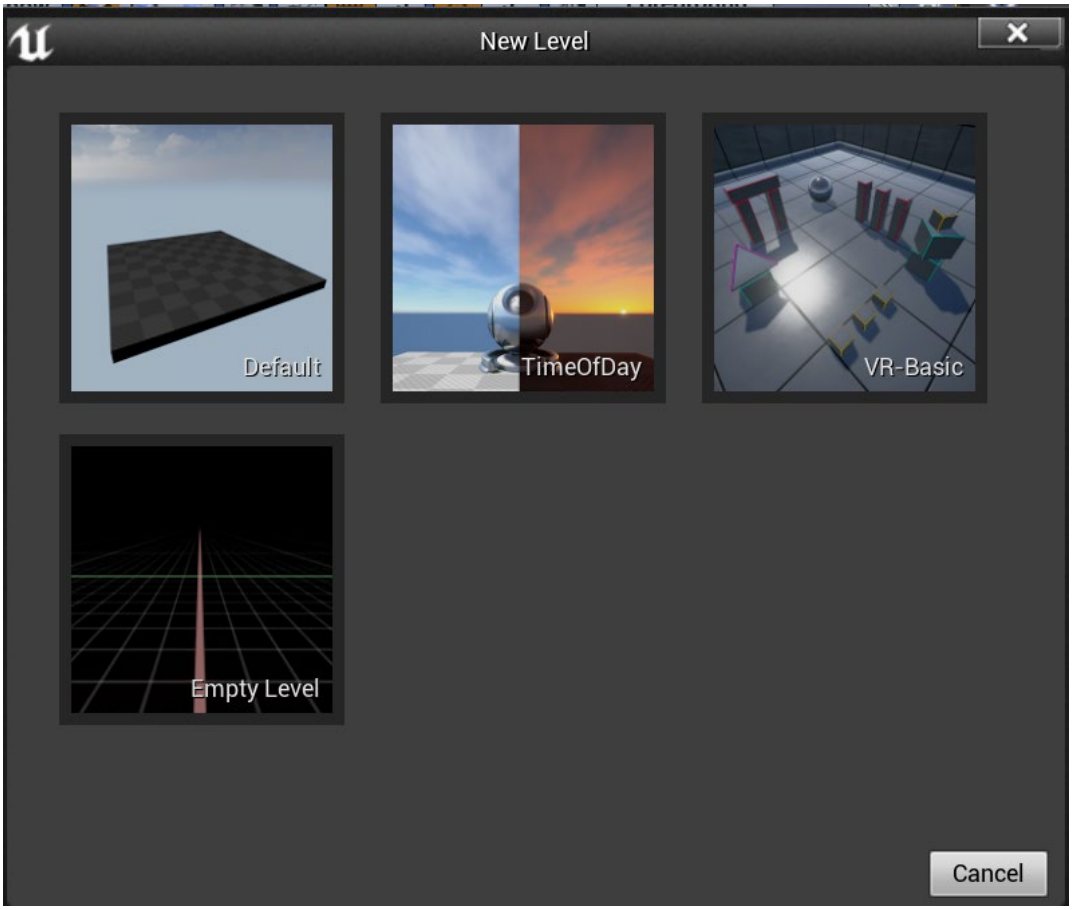

Figure 13.42: You are presented with a few templates when creating a new level

> **NOTE**
>
> The remaining steps are done in-editor using the **`SuperSideScroller.umap`** level provided for this activity. The images included in these steps will show this map. You can download the map here: https://packt.live/3lo7v2f.

3. Navigate to the **`/MainCharacter/Blueprints`** directory in the **`Content Browser`** interface to find the player character Blueprint, **`BP_SuperSideScroller_MainCharacter`**.

4. Next, add the main character Blueprint actor to the scene by *left-clicking* and dragging it from the **`Content Browser`** interface into the scene. The following screenshot shows the main character actor inside the **`SuperSideScroller.umap`** example map. Position the main character actor wherever you see fit for your level:

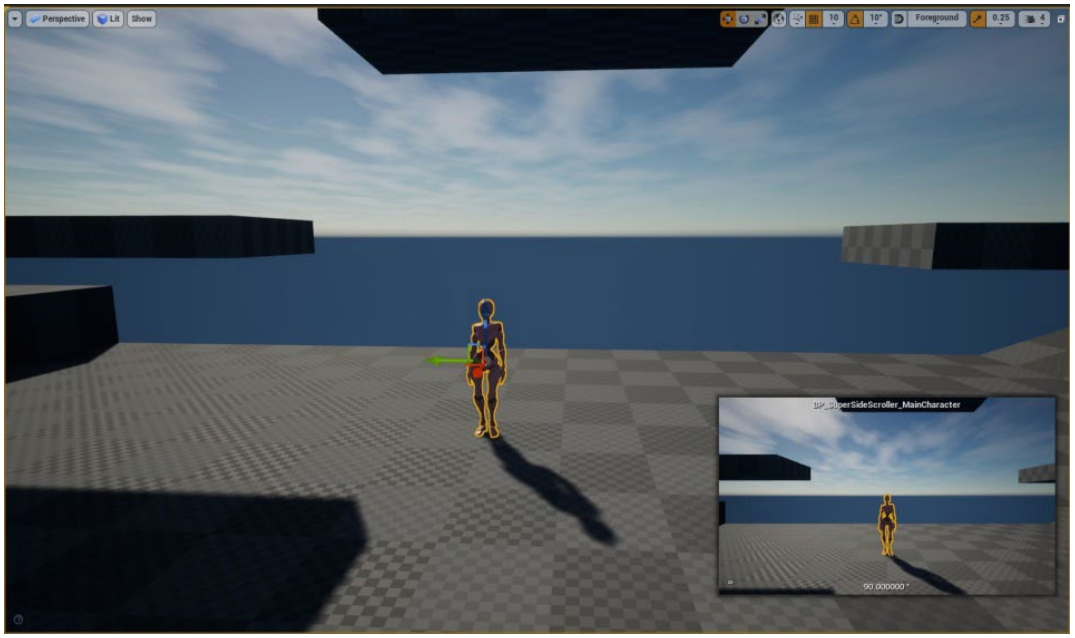

Figure 13.43: The main character placed in the example level

5. Instead of relying on **`GameMode`** to properly spawn the player character and having the player controller possess the character, let's implement a quick workaround to have the player auto possess your main character. Select the character Blueprint and navigate to the **`Details`** panel of the actor. Under the **`Pawn`** category, there is the **`Auto Possess Player`** option, as shown in the following screenshot. Set this option to **`Player 0`**:

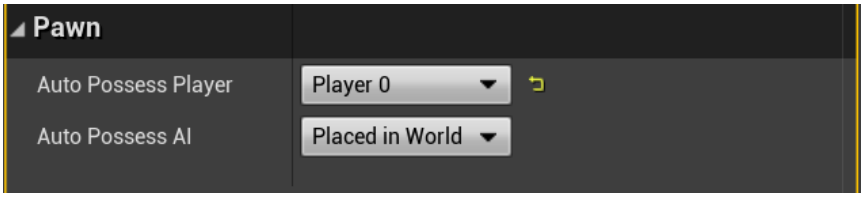

Figure 13.44: The Main Character will automatically become possessed by Player Controller 0, which is the controller given to the player by default

6. Next, add **`NavMeshBoundsVolume`** by navigating to the **`Modes`** panel, and under the **`Volumes`** category. *Left-click* and drag it into your level.

7. Adjust the **`Brush Settings`** options appropriately based on the size of the navigable space in your level. These values will be different than the ones shown here:

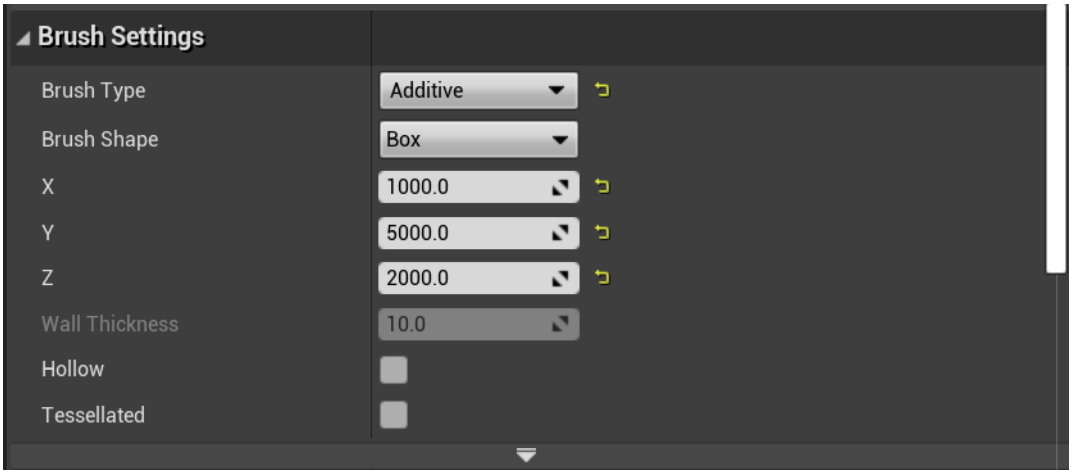

Figure 13.45: These values work for the playable space in the provided example level

8. Next, set the **`Generation`** parameters of the **`RecastNavMesh`** actor, as described in the final step of this activity: **`Cell Size`** is set to **`5.0f`**, **`Agent Radius`** is set to **`42.0f`**, and **`Agent Height`** is set to **`192.0f`**. Use these values as a reference, but these parameters may need to change based on how you set up your level.

9. Lastly, use the debug visualization feature of **`NavMeshBoundsVolume`** by pressing the **`P`** key, as shown in the following screenshot. Alternatively, you can navigate to the **`Show`** option and toggle the **`Navigation`** parameter to visualize the nav volume:

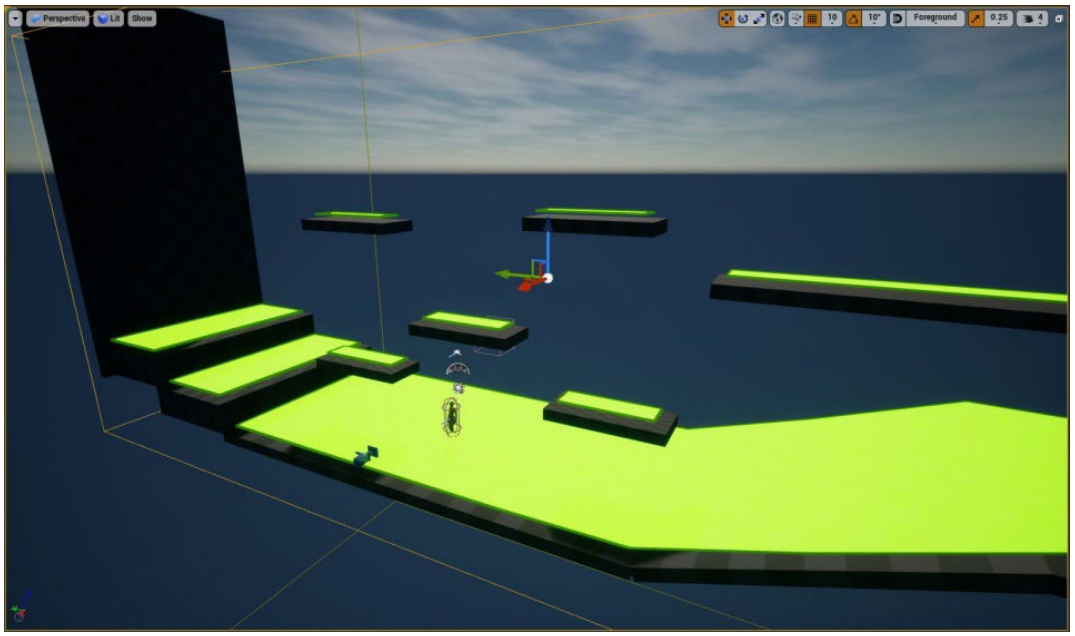

Figure 13.46: NavMeshBoundsVolume visualized

ACTIVITY 13.02: AI MOVING TO THE PLAYER LOCATION

1. Navigate to and open the **`BT_EnemyAI Behavior Tree`** asset you created earlier in this chapter, which is located in the **`/Enemy/AI/`** directory.
2. On the top toolbar, *left-click* the **`New Task`** option and select the **`BTTask_BlueprintBase`** to create a new task.
3. Name this new task **`BTTask_FindPlayer`** and *double-click* to open the asset.
4. *Right-click* and search for **`Event Receive Execute AI`**. *Left-click* this option from the context-sensitive menu to add the event node to the graph. This can be seen in the following screenshot:

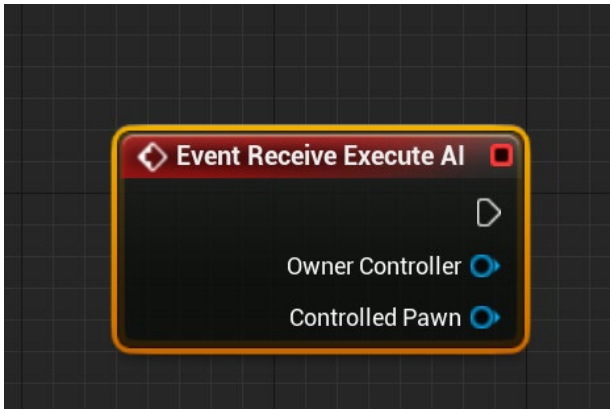

Figure 13.47: The Event Receive Execute AI event node

5. Right-click and search for **`Get Player Character`**. *Left-click* this option from the context-sensitive menu to add the function to the graph. Please refer to the following screenshot:

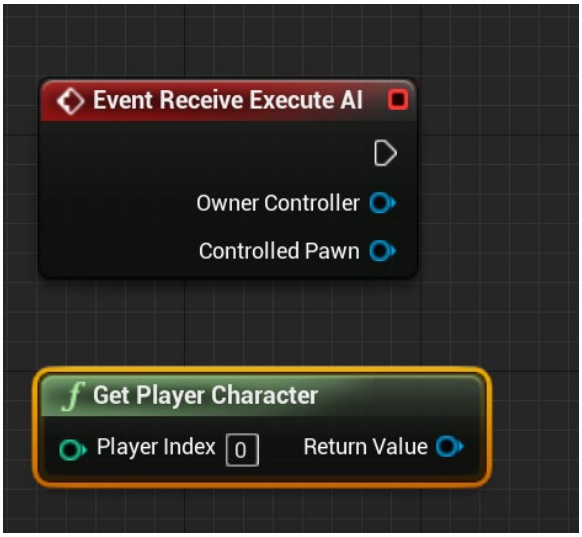

Figure 13.48: The Get Player Character returns the player character at index 0

The **`Get Player Character`** function returns the current player character possessed by the specified Player Index.

6. From the **`Return Value of the Player Character`** function, *left-click* and drag to show the context-sensitive search menu. Within this menu, find the **`GetActorLocation`** function and *left-click* the option to add it to the graph. Please refer to the following screenshot:

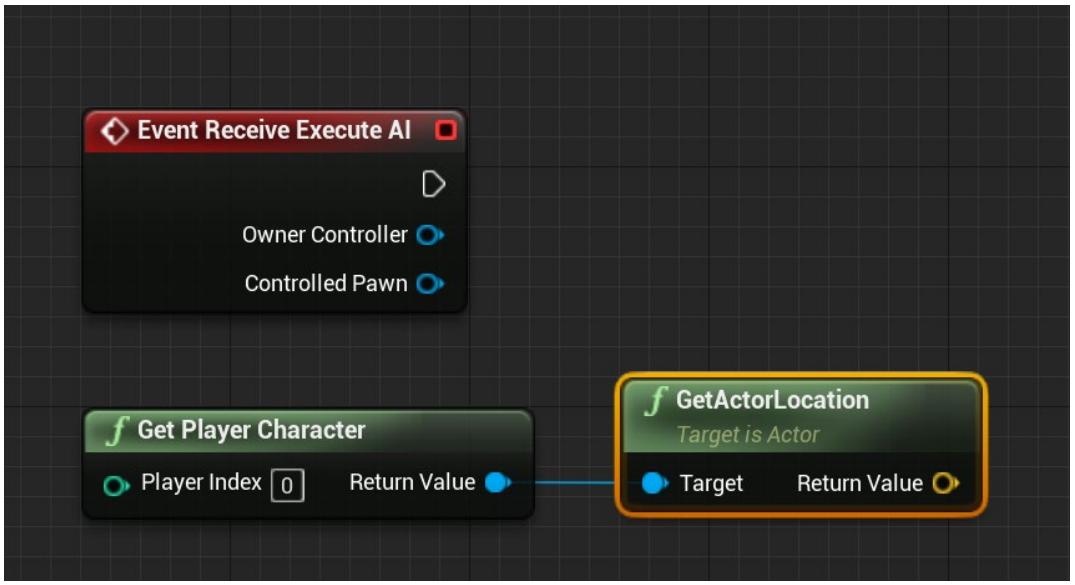

Figure 13.49: Now, you can access the actor location of the player character

7. Create a new public variable named **`NewLocation`** of the **`Blackboard Key Selector`** variable type.

8. *Left-click* and drag the **`NewLocation`** variable into the graph. From this variable, *left-click* and drag to show the context-sensitive search menu. Within this menu, find the **`Set Blackboard Value as Vector`** function and *left-click* the option to add it to the graph:

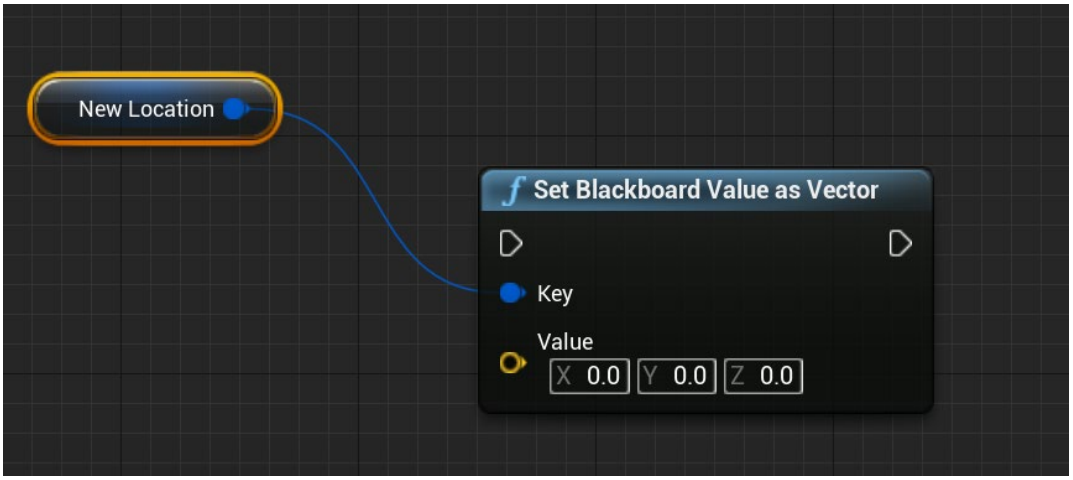

Figure 13.50: Now, you'll need to set the Blackboard value as a vector

- Connect the **`Vector`** return value of the **`GetActorLocation`** function to the **`Value`** input parameter of the **`Set Blackboard Value as Vector`** function. Then, connect the execution pins of the **`Event Receive Execute AI`** node and the **`Set Blackboard Value as Vector`** function:

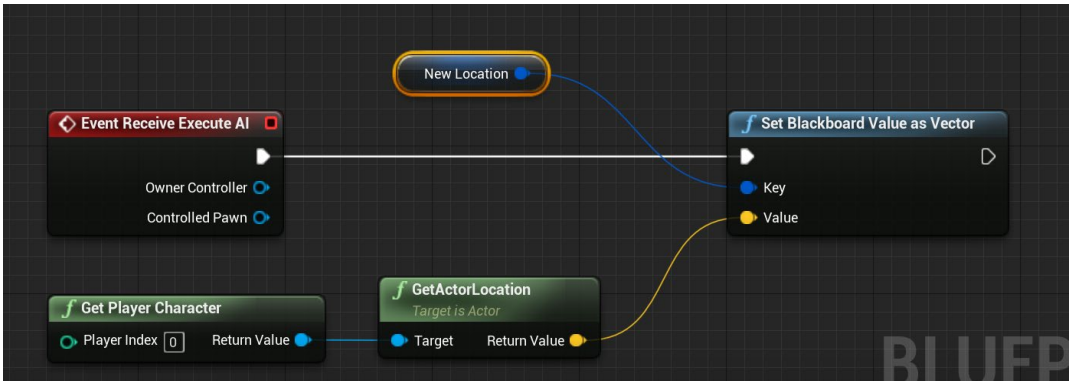

Figure 13.51: You are now setting the NewLocation key value to the location of the player character

> **NOTE**
>
> You can find the preceding screenshot in full resolution for better viewing at the following link: https://packt.live/3fiWHQZ.

9. *Right-click* and search for the **`Finish Execute`** function. *Left-click* this option from the context-sensitive menu to add the function to the graph.

10. Ensure that the **`Success`** Boolean parameter is set to **`True`** and connect its execution pin to the **`Set Blackboard Value as Vector`** function:

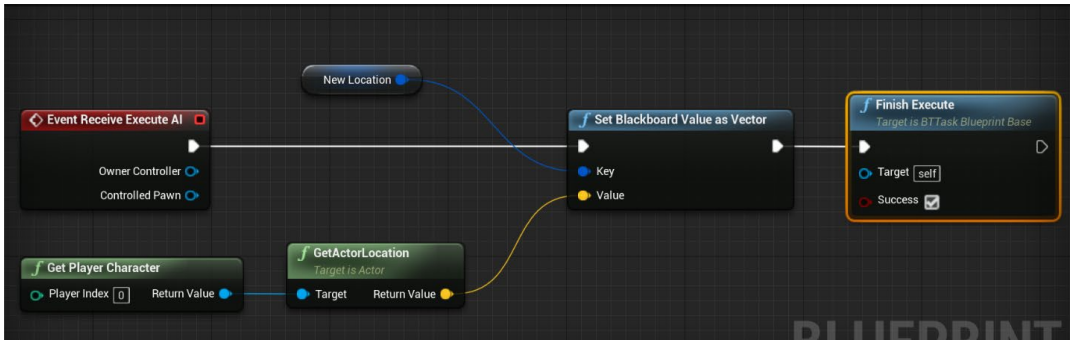

Figure 13.52: You always need to end the task Blueprint flow with the Finish Execute function

> **NOTE**
>
> You can find the preceding screenshot in full resolution for better viewing at the following link: https://packt.live/3eO9drp.

11. Save and compile the **`BTTask_FindPlayer`** task and return to the **`BT_EnemyAI`** behavior tree.

12. Delete the **`BTTask_FindLocation`** task for now; you will need to return this task to the behavior tree at the end of this activity!

13. From the **`Sequence`** node, *left-click* and drag out the pin to use the context-sensitive search to find the **`BTTask_FindLocation`** task.

14. Position the **`BTTask_FindLocation`** task so that it is in the first position under the **`Sequence`** node:

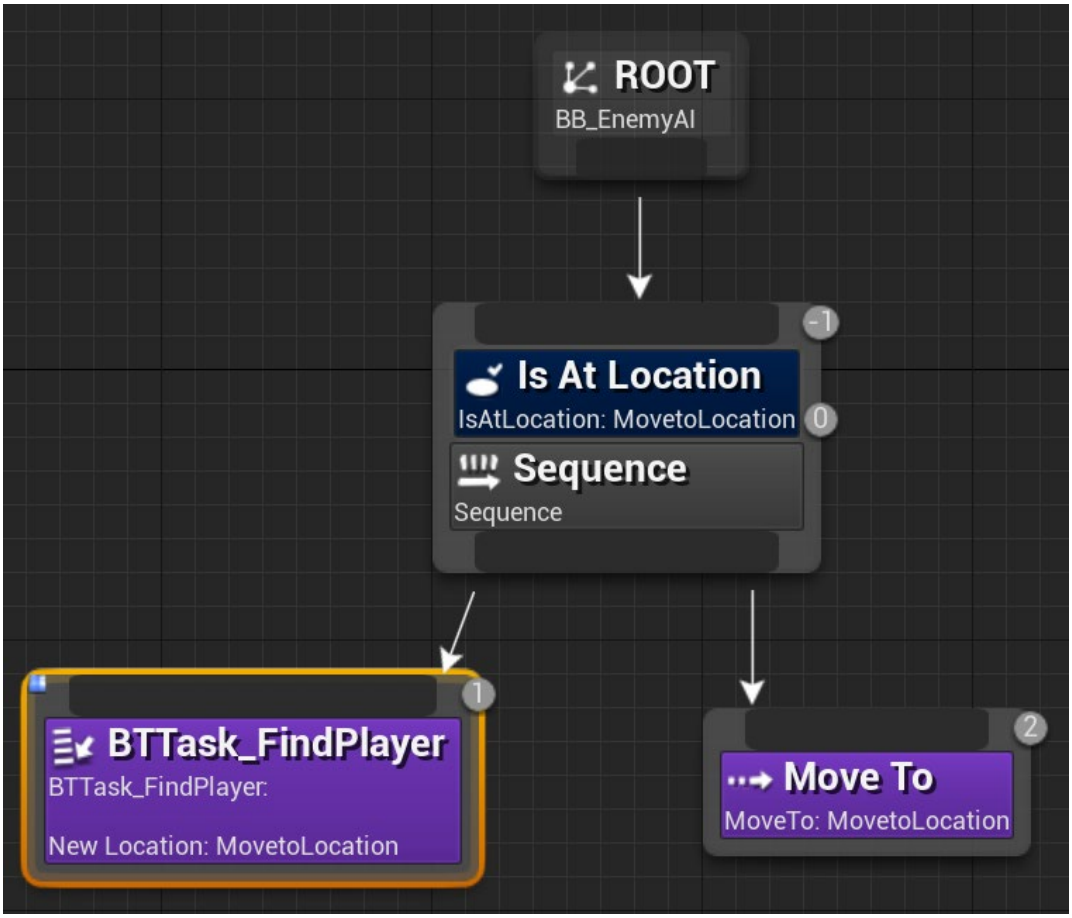

Figure 13.53: Now, the new BTTask_FindPlayer task is used in the behavior tree

15. From the **`Sequence`** node, *left-click* and drag out the pin to use the context-sensitive search to find the **`PlaySound`** task:

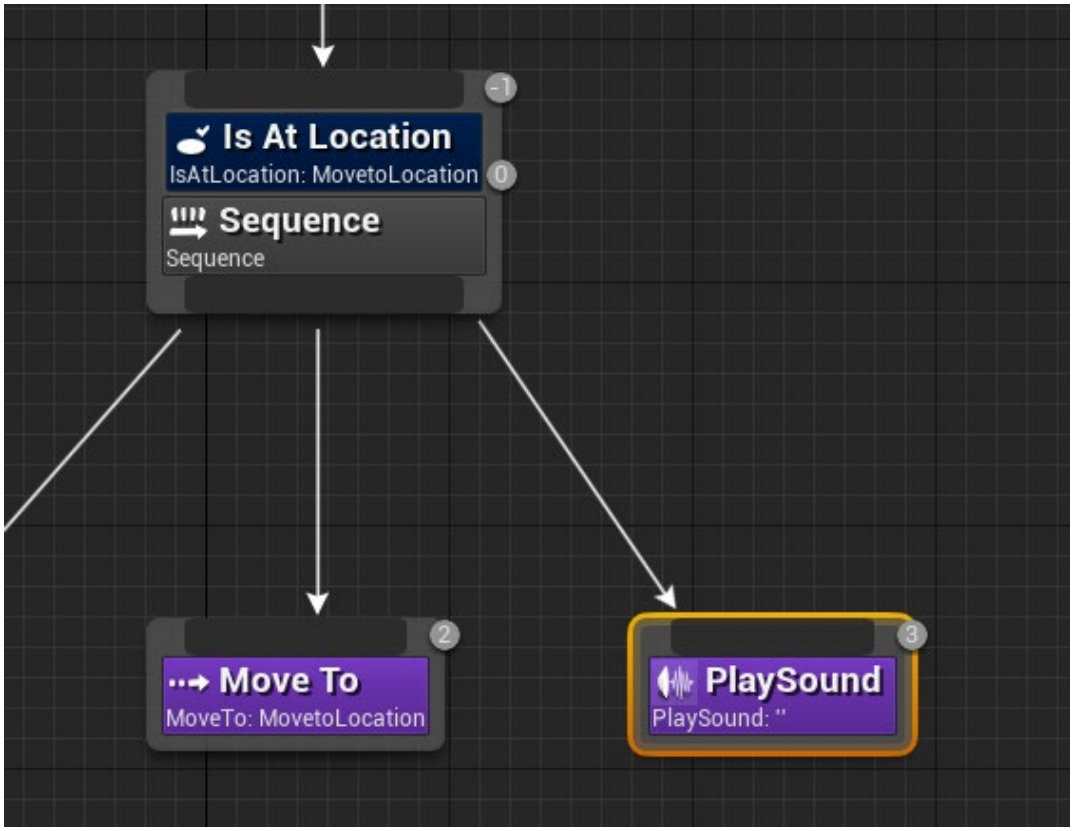

Figure 13.54: The PlaySound task is a straightforward task that simply plays a sound

16. Select the **`PlaySound`** task, and in its properties, find the **`Sound to Play`** parameter:

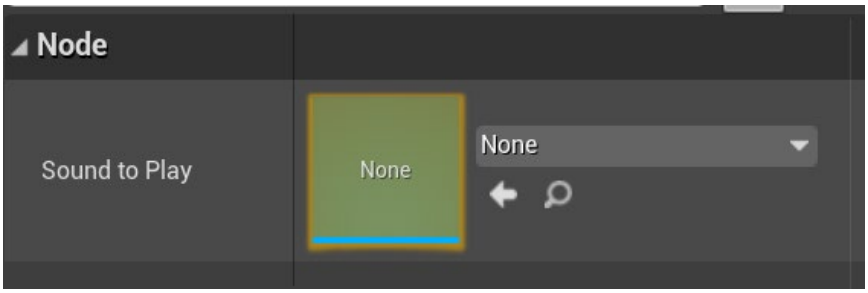

Figure 13.55: Here is where you assign the SoundCue to play

17. Search for and assign the **`Explosion_Cue SoundCue`** asset.

18. Next, *right-click* on the **`PlaySound`** task node and select **`Add Decorator`**. From the available list of options, select **`Is At Location`**:

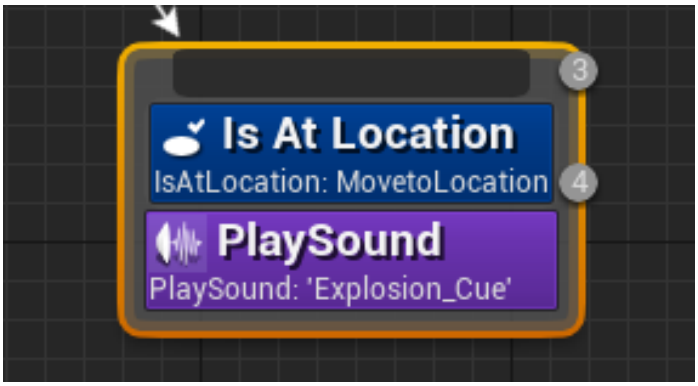

Figure 13.56: Now, the Explosion_Cue sound will only play when the AI is at MovetoLocation

19. From the **`Sequence`** node, *left-click* and drag out the pin to use the context-sensitive search to find the **`Wait`** task. The following screenshot shows where to find this node:

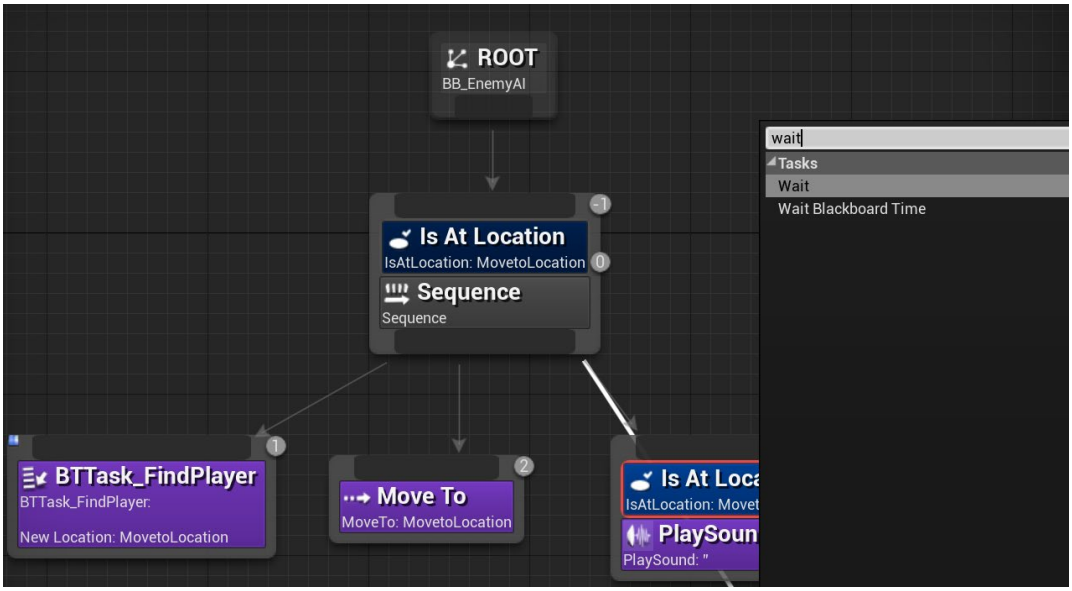

Figure 13.57: The Wait task is one of the handful of Tasks available to you by default in Unreal Engine 4

20. With the **`Wait`** task selected, access its **`Details`** panel to set the **`Wait Time`** parameter to **`2.0f`** seconds, as shown in the following screenshot:

Figure 13.58: The Wait Time parameter determines how long this Task will take to execute successfully

21. Make sure to position the **`Wait`** task to the right of both the **`BTTask_FindLocation`** and **`Move To`** Tasks so that it is the third task to be executed underneath the **`Sequence`** node, as shown in the following screenshot:

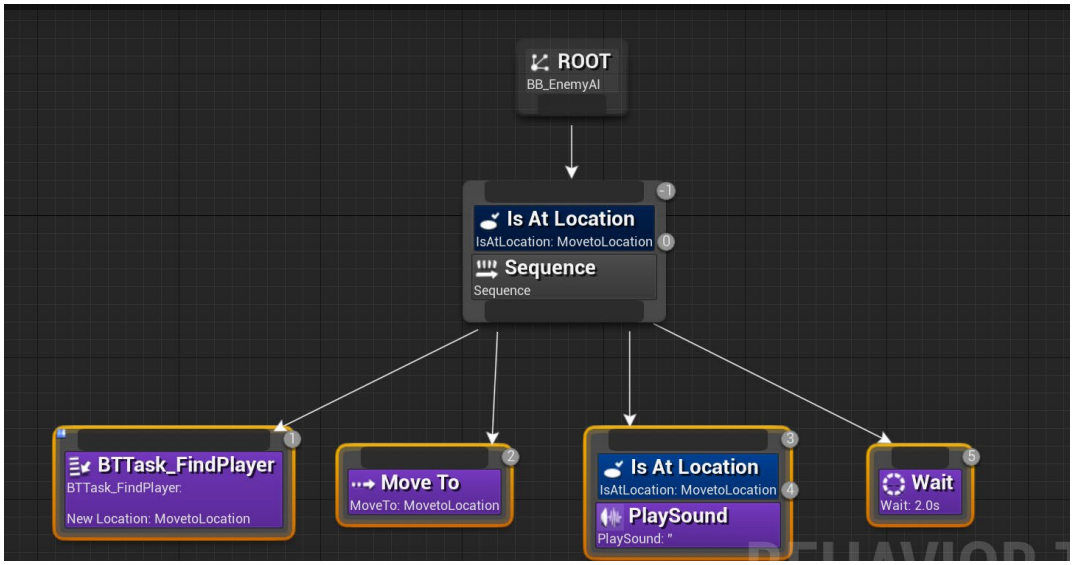

Figure 13.59: The final setup for the behavior tree

22. Lastly, compile the changes you've made to the behavior tree and save the asset.

ACTIVITY 13.03: CREATING THE PLAYER PROJECTILE BLUEPRINT

1. Back inside the editor, navigate to the **`/MainCharacter`** directory in the **`Content Browser`** interface. *Right-click* on the **`MainCharacter`** folder and select the **`New Folder`** option. Name this new folder **`Projectile`**, as shown here:

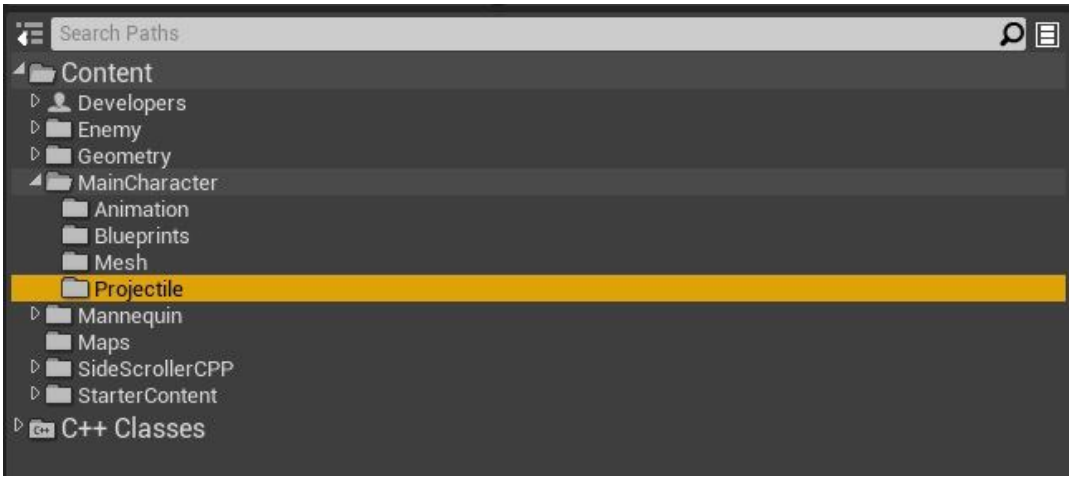

Figure 13.60: The new directory for the Player Projectile Blueprint

2. Inside the new **`Projectile`** folder, *right-click* and select the **`New Blueprint Class`** option. From the **`Pick Parent Class`** window, search for **`PlayerProjectile`** from the **`All Classes`** dropdown menu, as shown in the following screenshot. *Left-click* on the **`Select`** button to create the Blueprint:

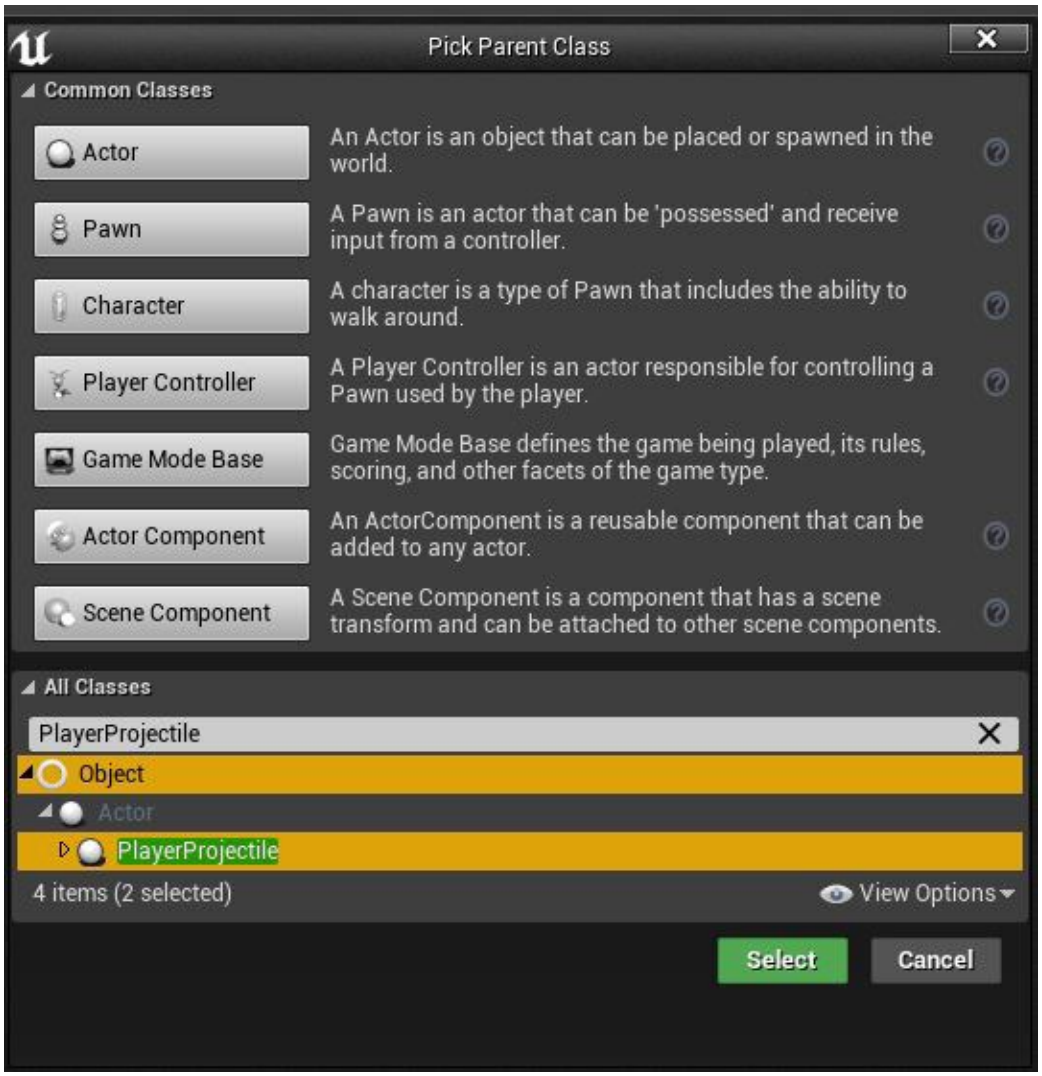

Figure 13.61: Now, you can create a new Blueprint from the PlayerProjectile class you created in the last few exercises

3. Name the new asset **`BP_PlayerProjectile`** and *double-click* it to open the Blueprint.

   From the **`Components`** tab, you will see the three components you added in the previous exercises: **`CollisionComp`**, **`MeshComp`**, and **`ProjectileMovement`**.

4. Select the **`MeshComp`** component and inside of its **`Details`** panel, find the **`Static Mesh`** parameter. *Left-click* the dropdown for this parameter and find the **`Shape_Sphere`** static mesh. Select this option to assign the simple sphere as the mesh representation of the projectile, as shown in the following screenshot:

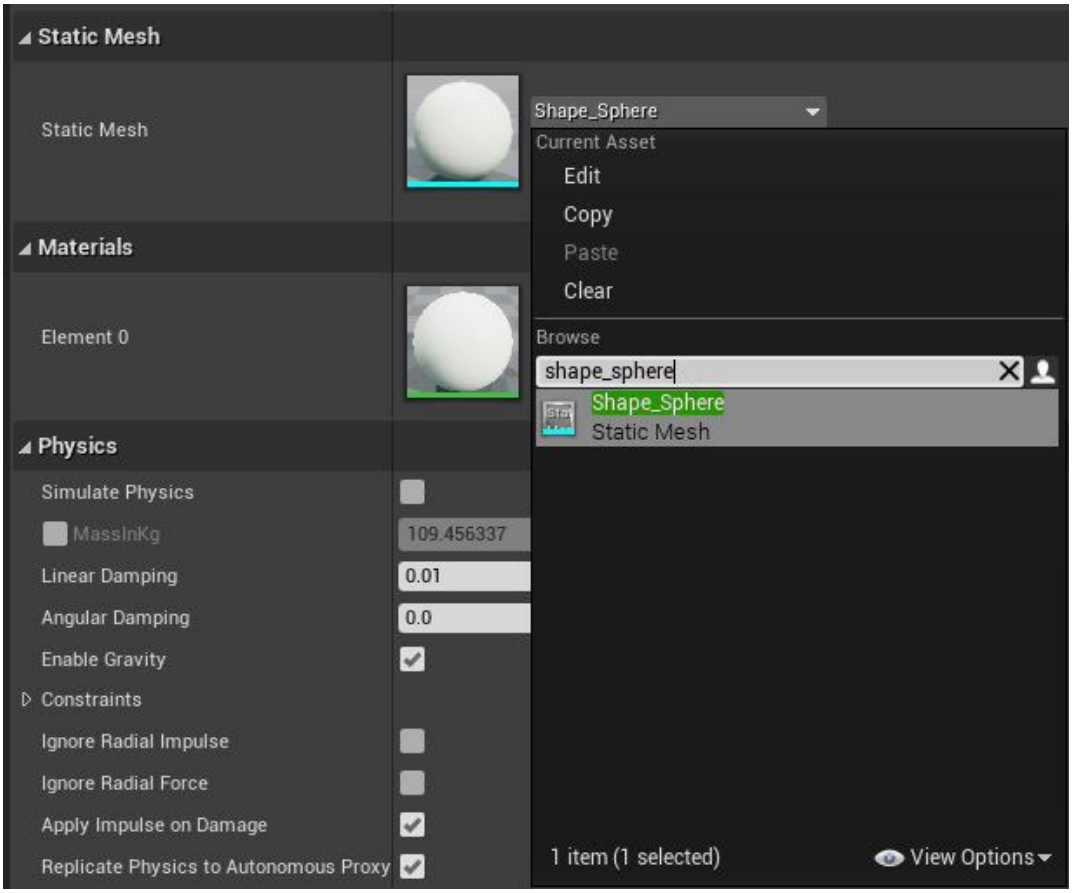

Figure 13.62: You will use a simple sphere shape as a visual representation of the projectile

5. Next, you need to adjust the **`Scale`** and **`Location`** parameters of **`MeshComp`** so that it better fits the location and size of **`CollisionComp`**, which you set in C++. Set the **`Transform`** settings like so:

```
Location:(X=0.000000,Y=0.000000,Z=-10.000000)
Scale: (X=0.200000,Y=0.200000,Z=0.200000)
```

The settings screen will be as follows:

Figure 13.63: The Transform settings for MeshComp so that it better fits the size and position of CollisionComp

Now if you navigate to the Viewport of the **`BP_PlayerProjectile`** actor, you will see what the projectile looks like, as shown here:

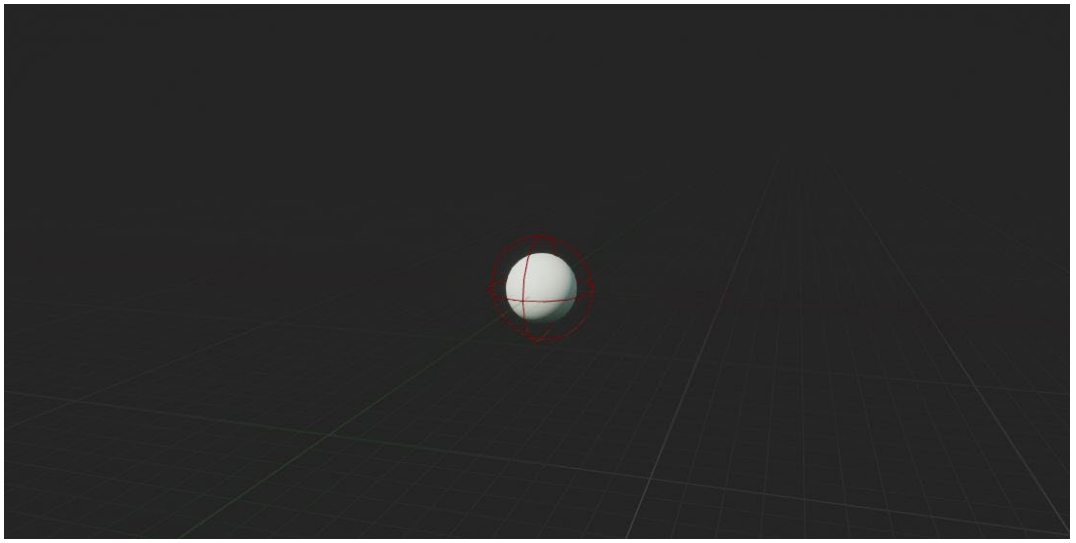

Figure 13.64: The current visual state of the PlayerProjectile actor

6. Now, for testing purposes, place the **BP_PlayerProjectile** actor into your level and use **PIE** to simulate the game. You will see the projectile fly through the air. The following screenshot shows an example inside the provided **SuperSideScroller.umap** example map:

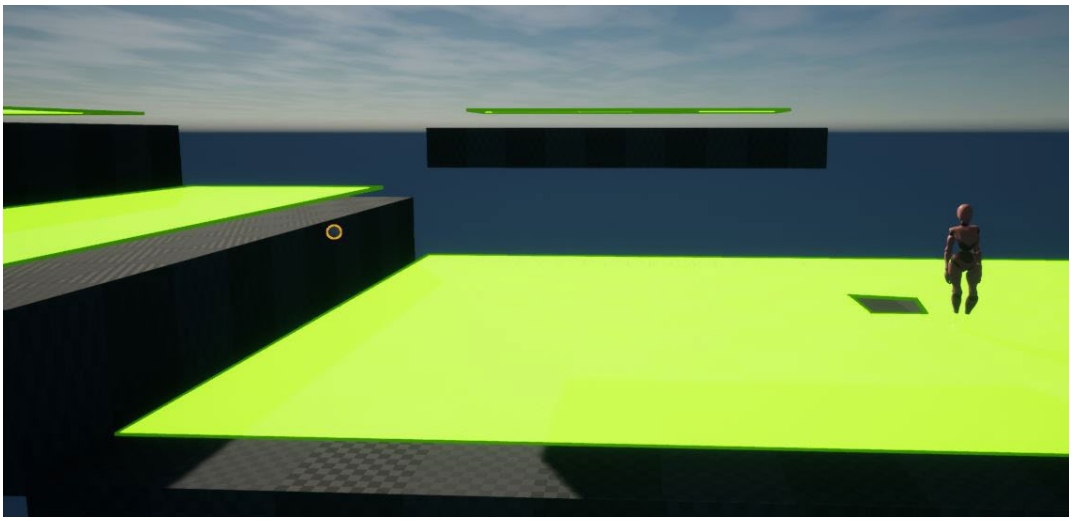

Figure 13.65: PlayerProjectile now moves through the level as you would expect a projectile would

7. Navigate to the **PlayerProjectile.cpp** source file inside Visual Studio. Here, you will find the empty **APlayerProjectile::OnHit** function you implemented in *Exercise 13.11*, *Creating the Player Projectile*.

8. Add the following line of code inside this function to correctly implement the **UE_LOG** line:

```
UE_LOG(LogTemp, Warning, TEXT("HIT"));
```

Now, the function should look like this:

```
void APlayerProjectile::OnHit(UPrimitiveComponent* HitComp, AActor*
  OtherActor, UPrimitiveComponent* OtherComp, FVector NormalImpulse,
const
  FHitResult& Hit)
{
  UE_LOG(LogTemp, Warning, TEXT("HIT"));
}
```

Now, the **`APlayerProjectile::OnHit()`** function will log the text **`HIT`** inside the output log, telling you that it was called successfully. In the next chapter, you will add more to this function to determine when the projectile hits an enemy.

9. Return to the Unreal Engine 4 editor and compile the code.

10. After the code compiles successfully, navigate to the **`Window`** option. From the **`Window`** dropdown, select the **`Developers Tools`** option and find **`Output Log`**. *Left-click* to select **`Output Log`**, as shown here:

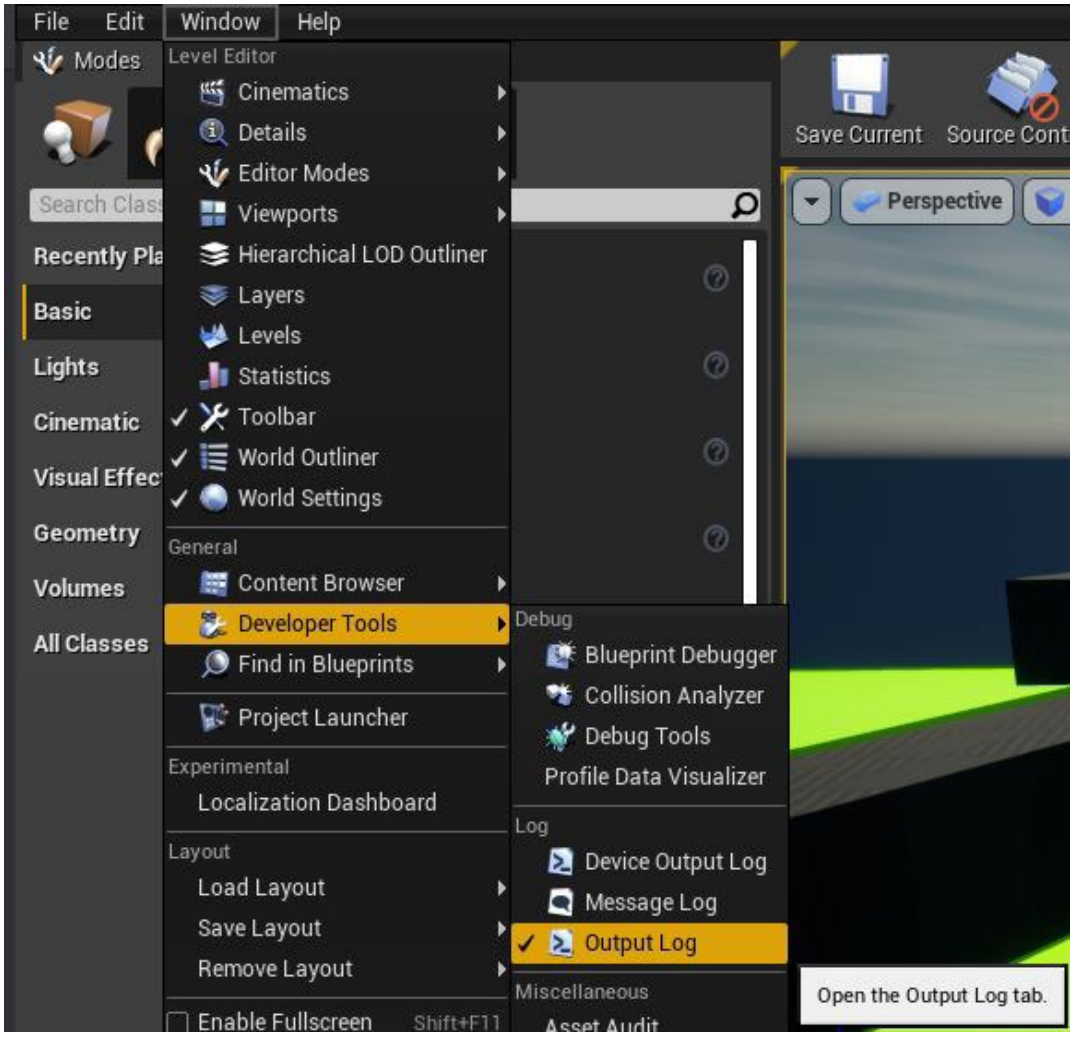

Figure 13.66: The Output Log shows you all the logged messages that occur during gameplay

11. Now, when you use **`PIE`** and the player projectile hits an object, you will see the log text **`HIT`**, as shown in the following screenshot:

Figure 13.67: When the Player Projectile successfully hits an object, the warning will be displayed

CHAPTER 14: SPAWNING THE PLAYER PROJECTILE

ACTIVITY 14.01: PROJECTILE DESTROYING ENEMIES

1. Navigate to Visual Studio and open the **`PlayerProjectile.cpp`** source file.
2. Add the following **`include`** to the list of includes at the top of the source file:

```
#include "EnemyBase.h"
```

3. Next, create the **`Enemy`** variable, which is of the **`AEnemyBase*`** type, and use the result of the cast from the **`OtherActor`** parameter of the **`APlayerProjectile::OnHit()`** function as the value. This is demonstrated here:

```
AEnemyBase* Enemy = Cast<AEnemyBase>(OtherActor);
```

4. After declaring this variable and setting its value, you need to ensure it's valid by using an **`if()`** statement, as shown here:

```
if (Enemy)
{
}
```

5. If **`Enemy`** is valid, then you can make the function call to **`DestroyEnemy()`** from within this **`if()`** block.

   The **`if()`** statement should look like this now:

```
if (Enemy)
{
  Enemy->DestroyEnemy();
}
```

6. After the **`if()`** block, add the following line of code:

```
ExplodeProjectile();
```

   Adding the call to **`ExplodeProjectile`** after the **`if()`** block will ensure that the player projectile will still be destroyed due to **`APlayerProjectile::OnHit()`**, regardless of whether **`Enemy`** is valid.

The **`APlayerProjectile::OnHit()`** function should now look like this:

```
void APlayerProjectile::OnHit(UPrimitiveComponent* HitComp,
  AActor* OtherActor, UPrimitiveComponent* OtherComp, FVector
  NormalImpulse, const FHitResult& Hit)
{
  AEnemyBase* Enemy = Cast<AEnemyBase>(OtherActor);
  if (Enemy)
  {
    Enemy->DestroyEnemy();
  }
  ExplodeProjectile();
}
```

7. With the **`APlayerProjectile::OnHit()`** function complete, return to the Unreal Engine 4 editor to recompile and hot-reload the code changes.

8. Lastly, use **`PIE`** to use the player projectile against an enemy. This should have been placed in your level in the previous chapter:

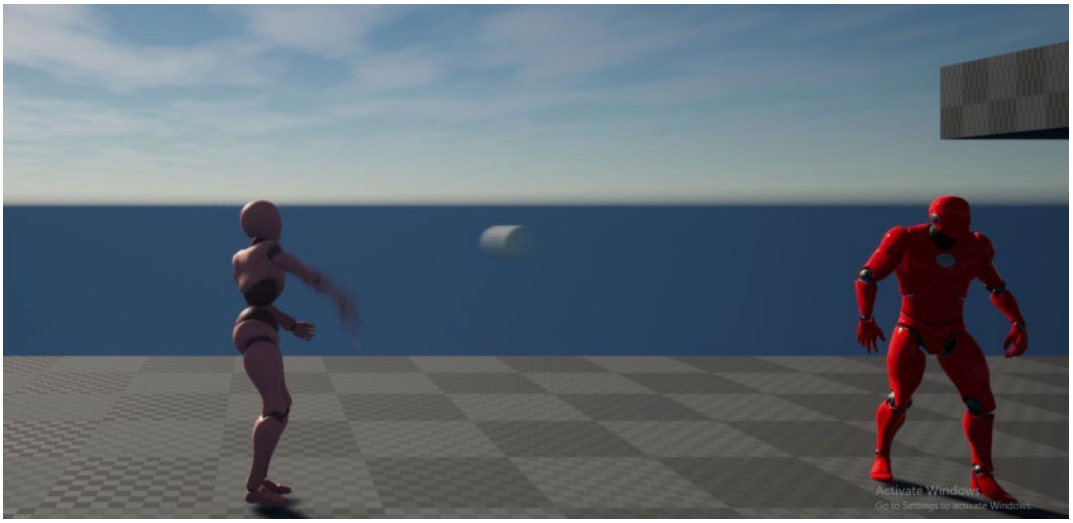

Figure 14.34: The player throwing the projectile

As the projectile hits the enemy, the enemy character is destroyed, as shown in the following screenshot:

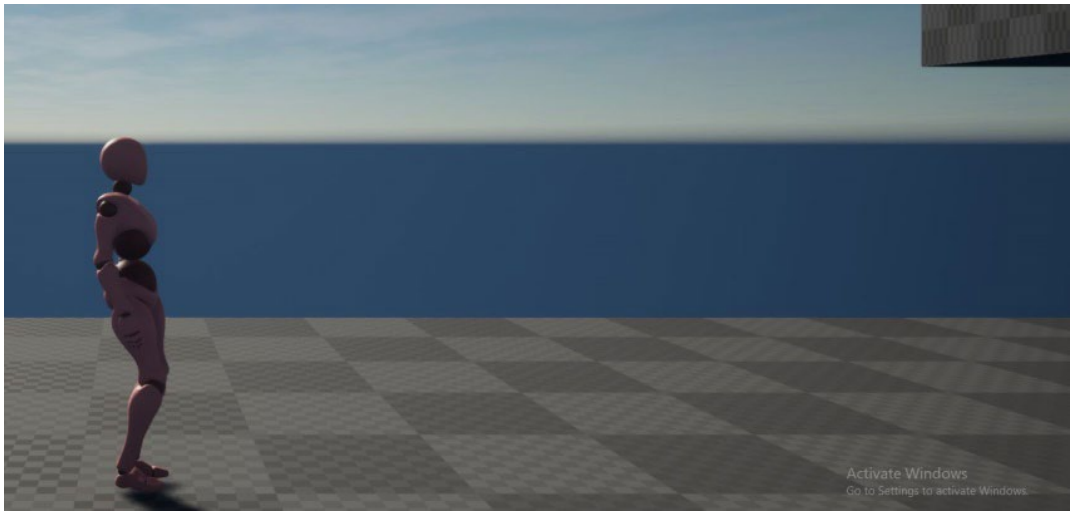

Figure 14.35: The projectile and enemy destroyed

Observe that when the projectile hits the enemy, both the enemy and the projectile are destroyed.

ACTIVITY 14.02: ADDING EFFECTS FOR WHEN THE PROJECTILE IS DESTROYED

1. In Visual Studio, navigate to and open the **`PlayerProjectile.h`** header file.
2. Add the following code under the **`Public`** access modifier to create the particle system parameter for the projectile, and name it **`DestroyEffect`**:

```
UPROPERTY(EditAnywhere, BlueprintReadOnly)
class UParticleSystem* DestroyEffect;
```

3. Add the following code under the **`Public`** access modifier to create the sound parameter for the projectile, and name it **`DestroySound`**:

```
UPROPERTY(EditAnywhere, BlueprintReadOnly)
class USoundBase* DestroySound;
```

4. Next, navigate to the source file of the player projectile class, **`PlayerProjectile.cpp`**, and add the following include for the **`GameplayStatics`** class to the top of the file:

```
#include "Kismet/GameplayStatics.h"
```

5. Update the **`APlayerProjectile::ExplodeProjectile()`** function to get a reference to the **`UWorld`** context object and store this in a reference variable, as shown here:

```
UWorld* World = GetWorld();
```

6. Next, use an **`if()`** statement to check the validity of the **`UWorld`** object reference:

```
if (World)
{
}
```

7. Within the **`if()`** block from the previous step, check the validity of the **`DestroyEffect`** parameter with a new **`if()`** statement. Then, make a call to the **`UGameplayStatics::SpawnEmitterAtLocation()`** function, as shown here:

```
if (DestroyEffect)
{
  UGameplayStatics::SpawnEmitterAtLocation(World,
  DestroyEffect, GetActorTransform());
}
```

8. After the **`If(DestroyEffect)`** block, do the same thing, but this time using the **`DestroySound`** parameter and the **`UGameplayStatics::SpawnSoundAtLocation()`** function:

```
if (DestroySound)
{
  UGameplayStatics::SpawnSoundAtLocation(World,
    DestroySound,      GetActorLocation());
}
```

9. With the **`APlayerProjectile::ExplodeProjectile()`** function updated, return to the Unreal Engine 4 editor to recompile and hot-reload these code changes.

10. After the code successfully recompiles, navigate to the **`/MainCharacter/Projectile/`** directory. *Double-click* the **`BP_PlayerProjectile`** asset to open it.

11. In the **`Details`** panel of the player projectile, you will now find both the **`Destroy Effect`** and **`Destroy Sound`** parameters.

12. Apply the **`P_Explosion`** VFX to the **`Destroy Effect`** parameter, as shown here:

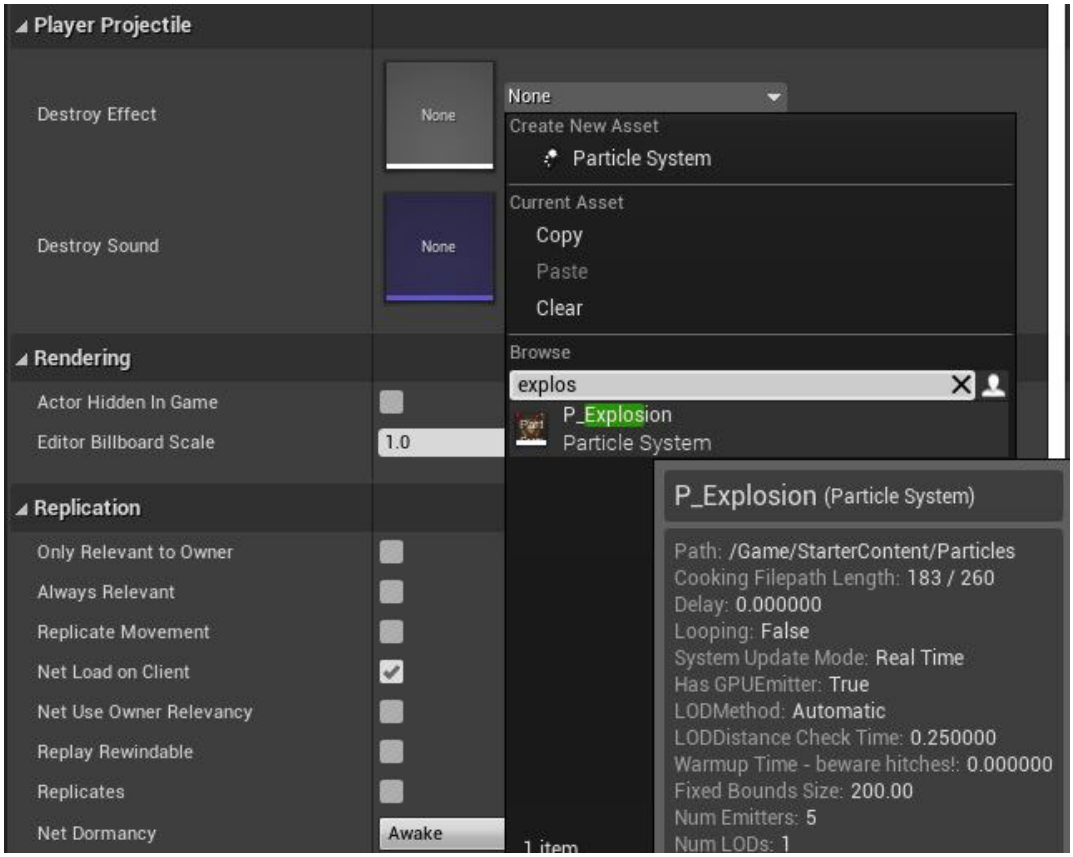

Figure 14.36: Now, you can add the P_Explosion VFX asset to the Destroy Effect parameter

13. Next, apply the **`Explosion_Cue`** VFX to the **`Destroy Sound`** parameter, as shown here:

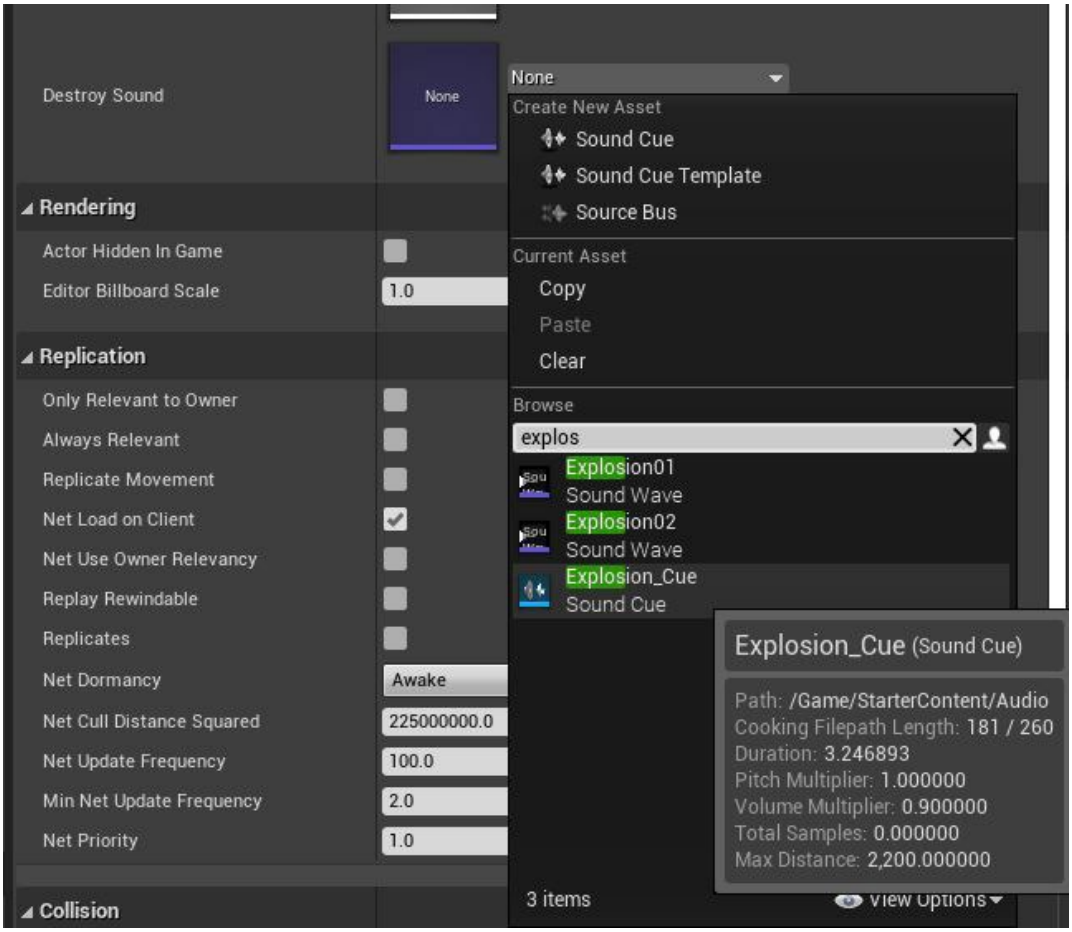

Figure 14.37: Now, you can add the Explosion_Cue SFX asset to the Destroy Sound parameter

14. With these two parameters assigned the VFX and SFX assets, compile and save the player projectile Blueprint.

15. Lastly, use **`PIE`** to spawn the player character and throw the player projectile. When the projectile collides with an object, the explosion VFX and SFX will play:

Figure 14.38: Projectile VFX and SFX

CHAPTER 15: COLLECTIBLES, POWER-UPS, AND PICKUPS

ACTIVITY 15.01: PLAYER OVERLAP DETECTION AND SPAWNING EFFECTS IN PICKABLEACTOR_BASE

> **NOTE**
>
> You can find the assets and code files for this activity at the following link: https://packt.live/35hLyME.

1. Inside the **PickableActor_Base.h** header file, underneath **Protected Access Modifier**, add the declaration for the **PlayerPickUp()** function, as shown here:

```
virtual void PlayerPickedUp(class ASuperSideScroller_Player* Player);
```

The **virtual** keyword implies that we will redefine this function in derived child classes, which we will apply later in this chapter.

2. Next, underneath the **PlayerPickedUp()** function, add the following code:

```
private:
  UFUNCTION()
  void BeginOverlap(UPrimitiveComponent* OverlappedComponent, AActor*
    OtherActor, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex,
bool
    bFromSweep, const FHitResult& SweepResult);
```

The reason why this **BeginOverlap()** function is declared underneath the **private** access modifier is so that no other classes outside the **PickableActor_Base** class can access or call it. There are a lot of input parameters for the **BeginOverlap()** function, so please refer back to *Chapter 6*, *Collision Objects*, where you learned and used this function inside the **VictoryBox** class.

3. Now, add the following code to create the new **UPROPERTY() USoundBase*** variable called **PickupSound** underneath **Public Access Modifier**:

```
UPROPERTY(EditAnywhere, BlueprintReadOnly)
class USoundBase* PickupSound;
```

UPROPERTY() contains the **EditAnywhere** and **BlueprintReadOnly** keywords so that **PickUpSound** can be assigned an asset in Blueprints. You will do this later in this activity.

4. Next, navigate to the **PickableActor_Base.cpp** source file and add the definition of the **BeginOverlap()** function:

```
void APickableActor_Base::BeginOverlap(UPrimitiveComponent*
  OverlappedComponent, AActor* OtherActor, UPrimitiveComponent*
OtherComp,
  int32 OtherBodyIndex, bool bFromSweep, const FHitResult&
SweepResult)
{
}
```

5. Then, add the definition of the **PlayerPickedUp()** function to the **PickableActor_Base.cpp** source file:

```
void APickableActor_Base::PlayerPickedUp(class ASuperSideScroller_
Player*
  Player)
{
}
```

6. Before we add logic to these functions, we need to add the **#include** header files for the classes we will be using:

```
#include "Kismet/GameplayStatics.h"
#include "SuperSideScroller_Player.h"
```

The **UGameplayStatics** class will be used to spawn **PickupSound**. You will be using the **SuperSideScroller_Player** class to **Cast** the overlapped actor from the **BeginOverlap()** function.

7. Now, we can add the following code to the **BeginOverlap()** function to determine whether **APickableActor_Base** has overlapped with **SuperSideScroller_Player**:

```
ASuperSideScroller_Player* Player =
  Cast<ASuperSideScroller_Player>(OtherActor);
```

8. Next, we need to check whether the **Player** variable is valid before moving on and calling the **PlayerPickedUp()** function:

```
if (Player)
{
}
```

9. To finish creating the **BeginOverlap()** function, add the following code within the **if()** statement we created in the previous step:

```
PlayerPickedUp(Player);
```

Now, the **BeginOverlap()** function will determine whether **SuperSideScroller_Player** has overlapped with **PickableActor_Base**. If so, we will call the **PlayerPickedUp()** function. Let's continue and create the definition of the **PlayerPickedUp()** function.

10. It is within the **PlayerPickedUp()** function that we need to spawn **PickupSound**, and to spawn the sound, we need a reference to **UWorld**. Add the following code to the **PlayerPickedUp()** function:

```
const UWorld* World = GetWorld();
```

11. Next, we need to make a valid check for the **World** variable before we can attempt to use it. This is a safety measure to ensure that no objects we use are **NULL**. Add the following code:

```
if (World)
{
}
```

12. With the same mindset that we used in the previous step, we also need to add a valid check for the **PickupSound** variable before we try to spawn it. Add the following code within the **if()** statement we created in the previous step:

```
if (PickupSound)
{
}
```

13. Now, if both **if()** statements are true, it is only then that we will attempt to spawn the sound using the **SpawnSoundAtLocation()** function from the **UGameplayStatics** class. Add the following code within the **if()** statement from the previous step:

```
UGameplayStatics::SpawnSoundAtLocation(World, PickupSound,
  GetActorLocation());
```

The **SpawnSoundAtLocation()** function was used in *Chapter 14*, *Spawning the Player Projectile*. Refer to that chapter for more information regarding this function.

The **PlayerPickedUp()** function should now look like this:

```
void APickableActor_Base::PlayerPickedUp(class ASuperSideScroller_
Player*
  Player)
{
  UWorld* World = GetWorld();
```

```
    if (World)
    {
      if (PickupSound)
      {
        UGameplayStatics::SpawnSoundAtLocation(World, PickupSound,
          GetActorLocation());
      }
    }
  }
```

14. The last thing to do in the **`PlayerPickedUp()`** function is to make a call to the **`Destroy()`** function of the **`PickableActor_Base`** class. This ensures that the object is removed from the game world. Add the following line of code outside both **`if()`** statements from the previous steps, but before the end of the **`PlayerPickedUp()`** function:

```
  Destroy();
```

    This means that regardless of whether the **`World`** or **`PickupSound`** variables are valid, the **`PickableActor_Base`** object will be destroyed and removed from the game world.

15. The only thing left to do in the **`PickableActor_Base.cpp`** source file is to bind the **`OnComponentBeginOverlap`** event of the **`CollisionComp Sphere Component`** to the **`OnBeginOverlap()`** function. Add the following code to the **`APickableActor_Base::APickableActor_Base()`** constructor, beneath the **`CollisionComp->BodyInstance.SetCollisionProfileName("OverlapAllDynamic");`** line:

```
  CollisionComp->OnComponentBeginOverlap.AddDynamic(this,
    &APickableActor_Base::BeginOverlap);
```

    This will bind the **`OnComponentBeginOverlap`** event of our **`Sphere Component`** to the **`BeginOverlap()`** function of the **`PickableActor_Base`** class so that whenever our **`Sphere Component`** receives an **`Overlap`** event, it will automatically call our **`BeginOverlap()`** function.

16. Compile the C++ code, but do not return to the Unreal Engine 4 editor; in fact, keep the editor closed.

17. Inside **`Epic Games Launcher`**, navigate to the **`Learn`** tab and the **`Games`** section to find the **`Unreal Match 3`** project.

18. Create a new project using **`Unreal Match 3`** and make sure to use engine version 4.24. Select a directory on your computer that you'd prefer to install this project in.

19. Once the **`Unreal Match 3`** project has been installed, open the project from the **`Library`** tab in **`Epic Games Launcher`**, under the **`My Projects`** category.

20. From the **`Content Browser`** window of your Unreal Match 3 project, find the **`Match_Combo`** audio asset and migrate this asset to your **`SuperSideScroller`** project.

21. With the new **`Match_Combo`** audio asset migrated into our project, return to the **`SuperSideScroller`** project in the Unreal Editor.

22. Next, open the **`BP_PickableActor_Base`** Blueprint you created in *Exercise 15.01: Creating the PickableActor_Base Class and Adding URotatingMovementComponent*.

23. In the **`Details`** panel of this Blueprint, you will find a section labeled **`Pickable Actor Base`**. In this section, you will find the **`PickupSound`** parameter you created earlier in this activity, as shown in the following screenshot:

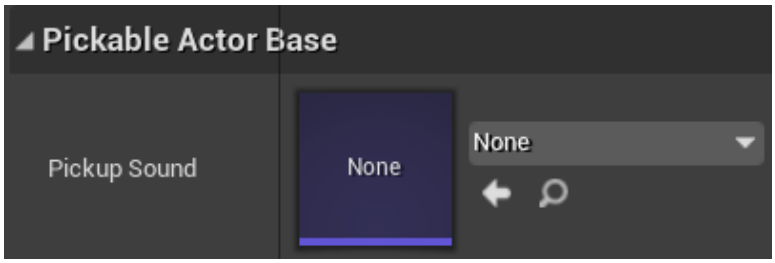

Figure 15.32: The PickupSound parameter you created in C++ can now be edited in the Blueprint

24. *Left-click* on the drop-down menu of this parameter and find the **`Match_Combo`** soundwave asset that you migrated to your project.

25. With this sound added to the **`PickupSound`** parameter, recompile the Blueprint.

26. If the **`BP_PickableActor_Base`** actor does not exist in your level, add the **`BP_PickableActor_Base`** actor to your level now.

27. Now, if you use PIE and overlap the player character with the **BP_PickableActor_Base** actor in your level, you will hear the **Match_Combo SoundWave** play, and the actor will be destroyed. Please refer to the following screenshot:

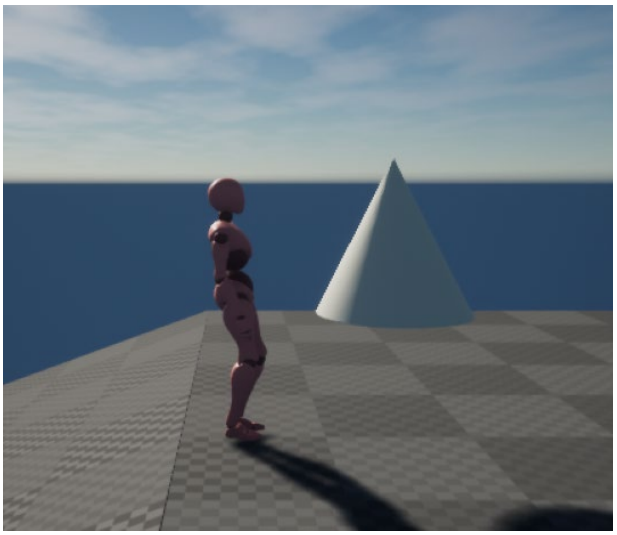

Figure 15.33: If the player overlaps with the Sphere Component, the actor will be destroyed and the Match_Combo sound will play

ACTIVITY 15.02: FINALIZING THE PICKABLEACTOR_COLLECTABLE ACTOR

> **NOTE**
>
> You can find the assets and code for this activity at the following link: https://packt.live/32xlk64.

1. Download and install the **Content Examples** project. This can be found in **Epic Games Launcher** from the **Learn** tab, underneath the **Engine Feature Samples** category.
2. Install this project using engine version 4.24 and install it in a directory of your choice.
3. Once installed, find the **SM_Pickup_Coin** asset in the **Content Browser** window of the **Content Examples** project.

4. *Right-click* and select the **`Asset Actions`** option, and then **`Migrate`**. Choose the **`Content`** folder of your **`SuperSideScroller`** project to migrate to.

5. Once you've migrated these assets into the **`Content`** directory, reopen the Unreal Engine 4 editor and navigate to the **`/Content/PickableItems`** directory, where you created the **`BP_PickableActor_Base`** actor.

6. In this new directory, create a new Blueprint by *right-clicking* in the empty directory in the **`Content Browser`** window, selecting **`Blueprint Class`**, and finding the **`PickableActor_Collectable`** class using the **`All Classes`** search. Please refer to the following screenshot:

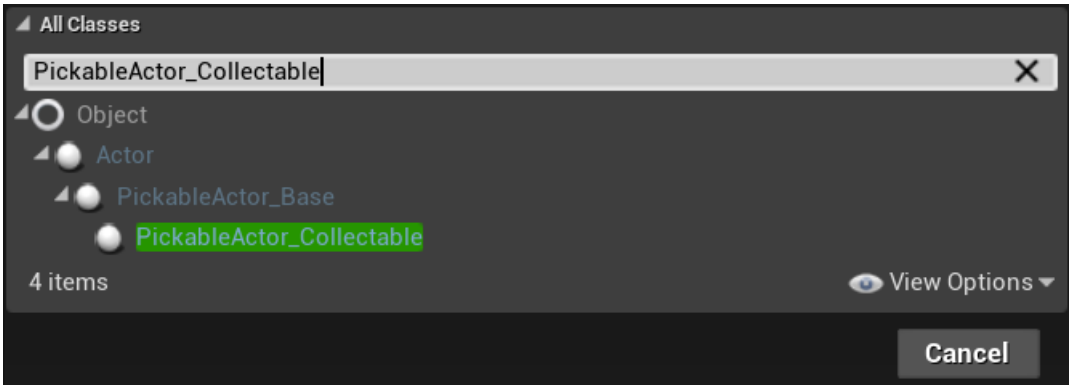

Figure 15.34: You can now find the PickableActor_Collectable class

7. Name this new Blueprint **`BP_Collectable`** and *double-left-click* to open it.

8. Select the **`MeshComp`** component from the **`Components`** tab, and in the **`Details`** panel, find the **`Static Mesh`** parameter.

9. Set this parameter for the **`SM_Pickup_Coin`** mesh that you added to the project in *Step 2* of this activity, as shown in the following screenshot:

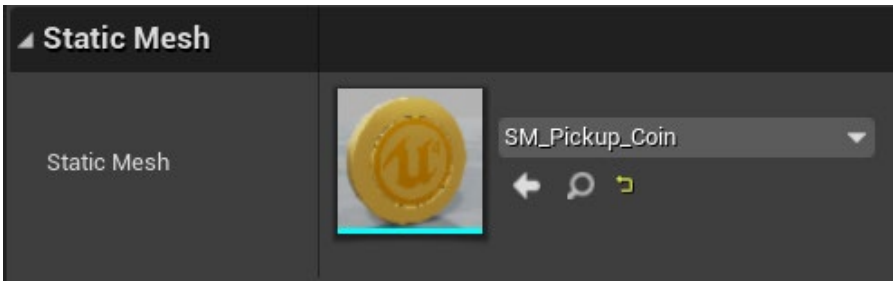

Figure 15.35: Now, the collectible is using the coin mesh you added to the project

10. Next, access the **`Details`** panel of the **`BP_Collectable`** actor and find the **`PickupSound`** parameter.
11. Set this parameter to the **`Match_Combo`** sound that you added to the project in *Activity 15.01*, *Player Overlap Detection and Spawning Effects in PickableActor_Base.* Please refer to the following screenshot:

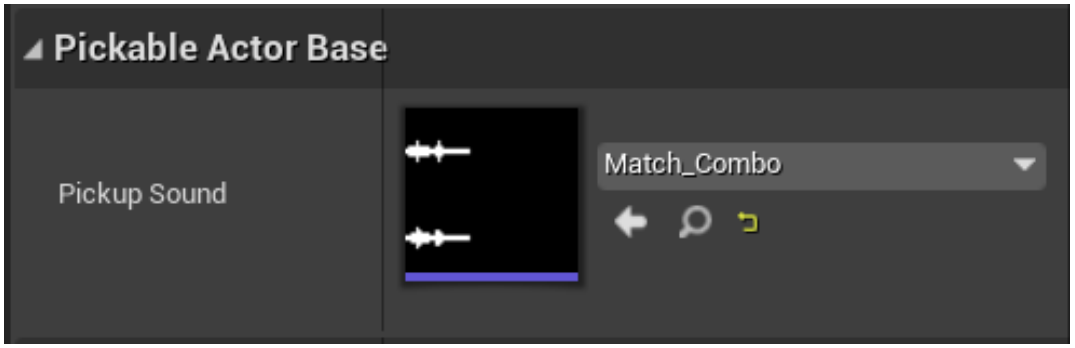

Figure 15.36: The coin collectible will use the same PickupSound as the BP_PickableActor_Base test actor

12. Finally, select the **`RotationComp`** component and find the **`Rotation Rate`** parameter under the **`Rotating Component`** section of its **`Details`** panel.
13. By default, **`Rotation Rate`** is set to 180-degree rotation in the Yaw axis; change this value from 180 to 90 in the Z parameter, which presents the Yaw axis, as shown in the following screenshot:

Figure 15.37: Rotation Rate set to rotate 90 degrees per second in the Yaw axis

14. Recompile the **`BP_Collectable`** Blueprint and save it.
15. Place the **`BP_Collectable`** Blueprint in your level and use PIE.

16. With your character, observe that the coin rotates in place and that when you overlap with the collectible coin, the **`Match_Combo`** sound plays and the actor disappears. Please refer to the following screenshot:

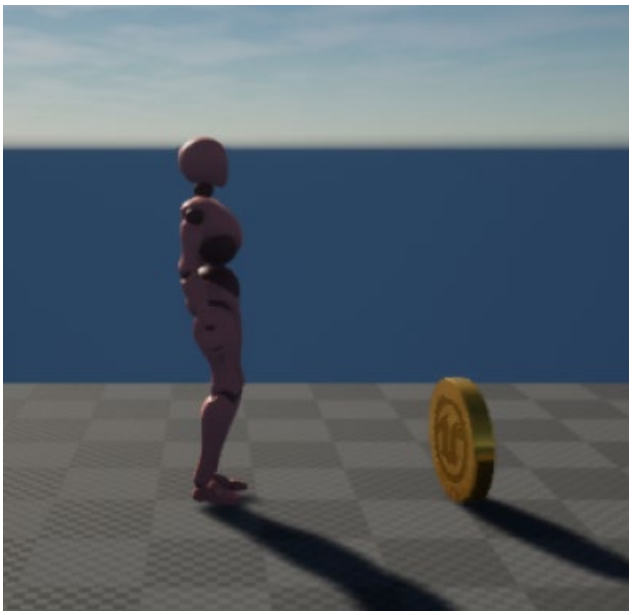

Figure 15.38: The coin collectible will use the same PickupSound as the BP_PickableActor_Base test actor

ACTIVITY 15.03: CREATING THE POTION POWER-UP ACTOR

> **NOTE**
>
> You can find the assets and code for this activity here: https://packt.live/3n5e1eO.

1. In the Unreal Engine 4 editor, navigate to the **`File`** option and select **`New C++ Class`**.
2. From the **`Choose Parent Class`** dialog window, select **`Show All Classes`** and select the **`PickableActor_Base`** class as the parent class.
3. Name this new class **`PickableActor_Powerup`** and click the **`Create Class`** button at the bottom of the window.

4. After creating the class, Unreal Engine 4 and Visual Studio will recompile and open the **PickableActor_Powerup** header and source files for you.

5. In the **PickableActor_Powerup.h** header file, create a new **Protected Access Modifier**:

```
protected:
```

6. Underneath this **Protected Access Modifier**, create the virtual override declaration of the **PlayerPickedUp()** class by adding the following code:

```
virtual void PlayerPickedUp(class ASuperSideScroller_Player* Player)
  override;
```

7. Next, add the virtual override declaration of the **BeginPlay()** function underneath **Protected Access Modifier** as well:

```
virtual void BeginPlay() override;
```

8. Now that we have the two override functions declared in the header file, let's move on and define these functions in the source file. Navigate to the **PickableActor_Powerup.cpp** source file.

9. Create the function definition of the **BeginPlay()** function and add the call to the parent class function using the **Super** keyword, as shown here:

```
void APickableActor_Powerup::BeginPlay()
{
  Super::BeginPlay();
}
```

10. Before creating the function definition of the **PlayerPickedUp()** function, we need to **#include** the **SuperSideScroller_Player** class. Add the following **#include** underneath the existing **#include**; that is, **"PickableActor_Powerup.h"**:

```
#include "SuperSideScroller_Player.h"
```

11. Now that we are including the **SuperSideScroller_Player** class, create the **PlayerPickedUp()** function definition by adding the following code:

```
void APickableActor_Powerup::PlayerPickedUp(class ASuperSideScroller_
Player*
  Player)
{
}
```

12. Next, add the call to the parent class function using the **Super** keyword, as shown here:

```
Super::PlayerPickedUp(Player);
```

13. Finally, use the **Player** input parameter variable to make the call to the **IncreaseMovementPowerup()** function you created in the previous exercise. Add the following code:

```
Player->IncreaseMovementPowerup();
```

14. Recompile the C++ code, but do not return to the Unreal Engine 4 editor.

15. Download and install the **Action RPG** project. This can be found by going to **Epic Games Launcher**, then the **Learn** tab, and then to the **Games** category.

16. Install this project using engine version 4.24 and install it into a directory of your choice.

    Once installed, find the **A_Character_Heal_Mana_Cue** asset in the **Content Browser** window of the **Action RPG** project.

17. *Right-click* and select the **Asset Actions** option, and then **Migrate**. Choose the **Content** folder of the **SuperSideScroller** project to migrate to.

18. Next, find the **SM_PotionBottle** asset and migrate this asset, as well as its referenced assets, to your **SuperSideScroller** project.

    These are the audio, material, and static mesh assets required for the potion power-up actor.

19. In the Unreal Engine 4 editor, create a new folder within the **/Content/PickableItems** directory called **Powerup**.

20. *Right-click* inside this directory and select the **Blueprint Class** option. From the **All Classes** search bar that appears in the **Pick Parent Class** dialog window, search for and select the **PickableActor_Powerup** class.

21. Name this new Blueprint **BP_Powerup** and *double-left-click* the asset to open it.

22. Select the **MeshComp** component from the **Components** tab and assign the **SM_PotionBottle** mesh to the **Static Mesh** parameter of the **Details** panels. Please refer to the following screenshot:

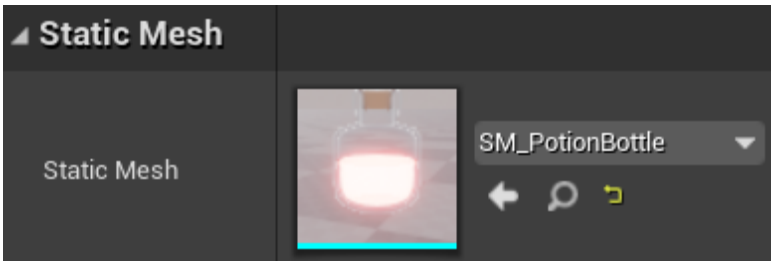

Figure 15.39: The SM_PotionBottle asset applied to the Static Mesh parameter of the MeshComp component

> **NOTE**
>
> The **M_PotionGlass** material that's applied to the **SM_PotionBottle** static mesh uses interesting logic within its material to give the illusion that the bottle is moving up and down. Take some time to look into this material to learn more about how this was done.

23. In the **Details** panel of the **PickableActor_Powerup** class, assign **A_Character_Heal_Mana_Cue** to the **Pickup Sound** parameter, as shown in the following screenshot:

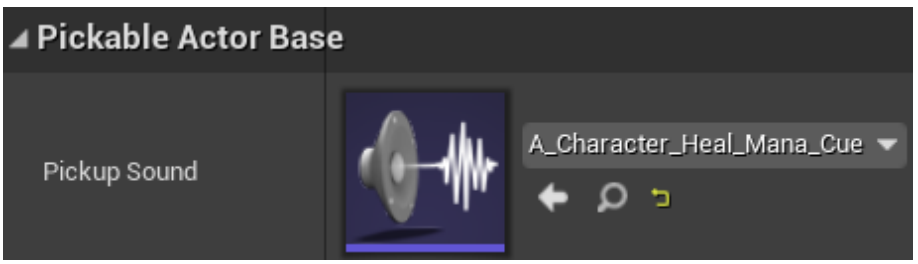

Figure 15.40: The A_Character_Heal_Mana_Cue sound cue applied to the Pickup Sound parameter

24. Finally, select the **RotationComp** component from the **Components** tab and update its **Rotation Rate** parameter, inside its **Details** panel, to the following values:

```
(X=60.000000,Y=180.000000,Z=0.000000)
```

25. Recompile the Blueprint and place one or more of the **BP_Powerup** actors into your level.

26. Use PIE to control your character and overlap the player character with the potion power-up. Observe that when you overlap with the potion, it will disappear, the **A_Character_Heal_Mana_Cue** sound will play, and your character will move faster and jump higher. Please refer to the following screenshot:

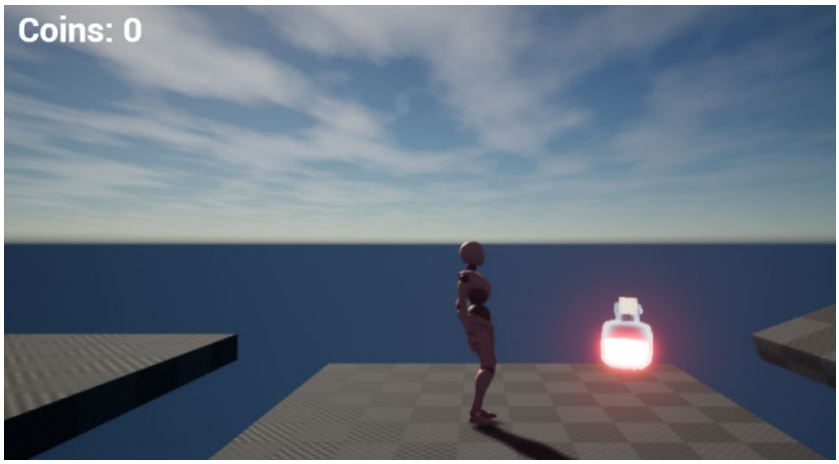

Figure 15.41: The potion power-up now rotates in-place and can be picked up by the player

CHAPTER 16: MULTIPLAYER BASICS

ACTIVITY 16.01: CREATING A CHARACTER FOR THE MULTIPLAYER FPS PROJECT

You begin by creating the project:

1. Create a new **`Blank`** project using C++ called **`MultiplayerFPS`**, make sure it has no **`Starter Content`** and save it on a location of your choosing.
2. Once the project has been created, it should open the editor as well as the Visual Studio solution. Next, you need to import the assets.
3. In **`Content Browser`**, create the folder structure for **`Content\Player\Mesh`** and open the folder.
4. Click on the **`Import`** button, go to the **`Activity16.01\Assets`** folder, and import **`SK_Mannequin.fbx`**. You should get the following:

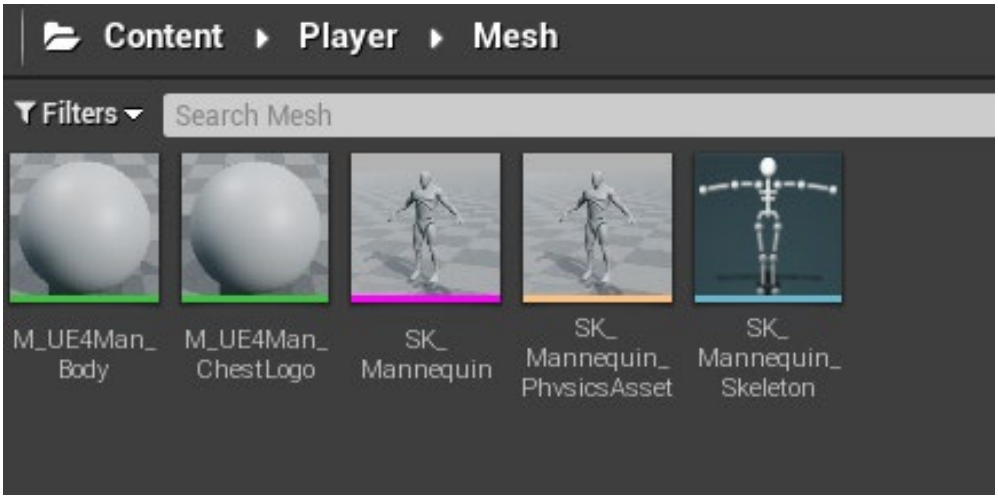

Figure 16.28: Importing the Mannequin

5. Jump back to **`Content\Player`**, create the **`Animations`** folder, and open it.
6. Click on the **`Import`** button and go to the **`Activity16.01\Assets`** folder, select the **`animation`** FBX files, and click **`Ok`**.

7. In the import dialog, make sure you select **`SK_Mannequin_Skeleton`** and click **`Import All`**. You should get the following:

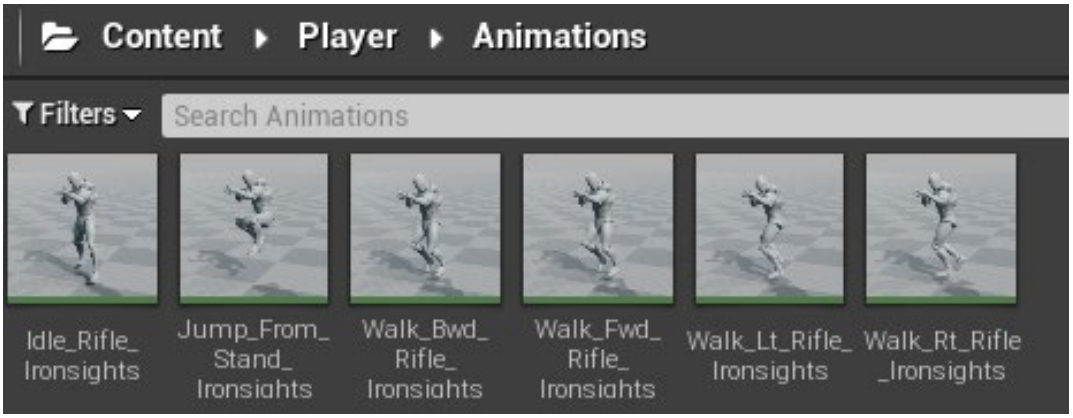

Figure 16.29: Importing the Mannequin Animations

8. Jump back to **`Content\Player`**, create the **`Sounds`** folder, and open it.

9. Click on the **`Import`** button, go to the **`Activity16.01\Assets`** folder, select **`Spawn.wav`**, **`Footstep.wav`**, and **`Jump.wav`**, and click **`Ok`**.

10. Open the **`Jump_From_Stand_Ironsights`** animation and add a play sound anim notify at time **`0.115`** sec, that uses the **`Jump`** sound, like so:

Figure 16.30: Play sound anim notify added to the jump animation

11. Open each walk animation individually and place a play sound anim notify using **`Footstep.wav`** at the following time signatures:

Animation	Time signatures to place each Anim Notify
Walk_Fwd_Rifle_Ironsights	0.26 sec, 0.7 sec, 1.1 sec and 1.6 sec
Walk_Bwd_Rifle_Ironsights	0.36 sec, 0.8 sec, 1.23 sec, 1.66 sec
Walk_Lt_Rifle_Ironsights	0.26 sec, 0.7 sec, 1.1 sec and 1.6 sec
Walk_Rt_Rifle_Ironsights	0.26 sec, 0.7 sec, 1.1 sec and 1.6 sec

Figure 16.31: Time signatures for animation

After adding the play sound anim notifies, it should look something like the following:

Figure 16.32: Play sound anim notify added to the walk animations

12. Finally, save everything.

    Next, set up the skeletal mesh with retargeted bones (covered in *Chapter 2*, *Working with Unreal Engine*) and create the **`Camera`** socket.

13. Open the **`SK_Mannequin_Skeleton`** asset in **`Content\Player\Mesh`**.

14. In the **`Skeleton Tree`** tab, click the **`Options`** button and make sure **`Show Retargeting Options`** is set to **`true`**:

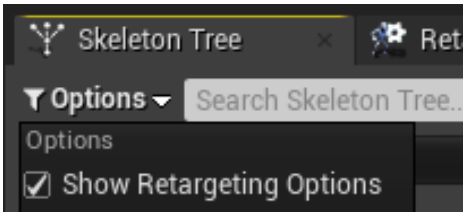

Figure 16.33: Show Retargeting Options in Skeleton Tree

15. *Right-click* on the **`root`** bone and select **`Recursively Set Translation Retargeting Skeleton`**.

16. For the **`root`** and **`pelvis`** bones, click their **`Translation Retarget`** dropdown and pick **`Animation`**.

17. You should have the **`root`** and **`pelvis`** bones set to **`Animation`** and the rest set to **`Skeleton`**:

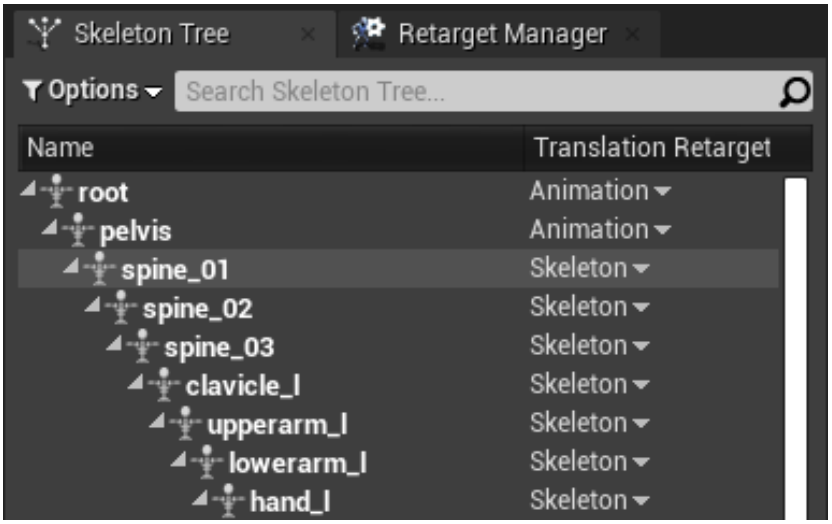

Figure 16.34: Setting the Translation Retarget for the root and pelvis bones

18. In the **`Skeleton Tree`** tab, go to the **`head`** bone, *right-click* on it, pick **`Add Socket`**, and rename it **`Camera`**.

19. Select the socket and, on the **`Details`** panel, set the **`Relative Location`** to (*X=7.88, Y=4.73, Z=-10.00*). You should get the following:

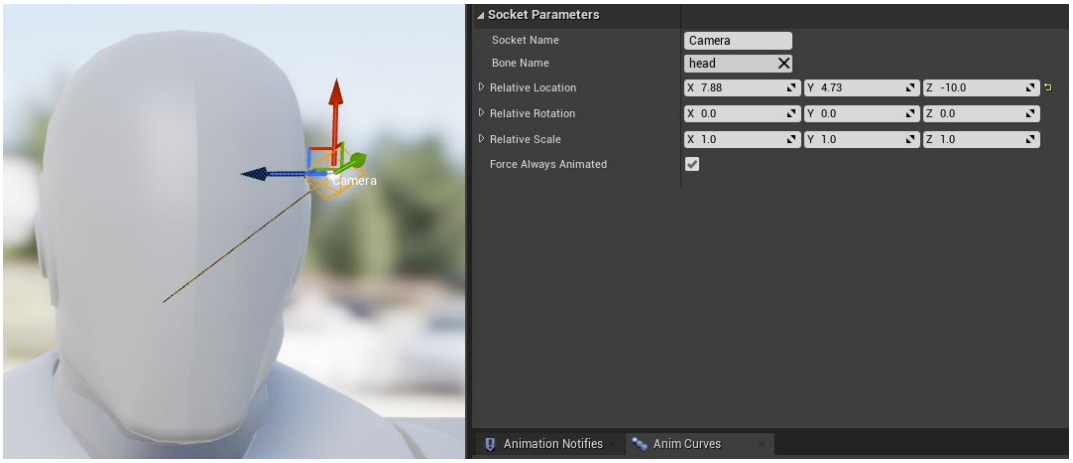

Figure 16.35: Setting the socket location

20. Save and close **`SK_Mannequin_Skeleton`**.

    Next, you have to create the movement 2D Blend Space:

21. Go to the **`Content\Player\Animations`** folder, click on the **`Add New`** button, and pick **`Animation -> Blendspace`**.

22. Select the character's skeleton.

23. Rename the Blend Space **`BS_Movement`** and open it.

24. Create the **`horizontal Direction`** axis (**`-180 to 180`**) and the **`vertical Speed`** axis (**`0 to 800`**).

25. Drag the **`Idle_Rifle_Ironsights`** animation on the five grid entries where **`Speed`** is **`0`**.

26. Drag the **`Walk_Fwd_Rifle_Ironsights`** animation where **`Speed`** is **`800`** and **`Direction`** is **`0`**.

27. Drag the **`Walk_Lt_Rifle_Ironsights`** animation where **`Speed`** is **`800`** and **`Direction`** is **`-90`**.

28. Drag the **`Walk_Rt_Rifle_Ironsights`** animation where **`Speed`** is **`800`** and **`Direction`** is **`90`**.

29. Drag the **`Walk_Bwd_Rifle_Ironsights`** animation where **`Speed`** is **`800`** and **`Direction`** is **`-180`** and **`180`**.

30. You should end up with the following Blend Space:

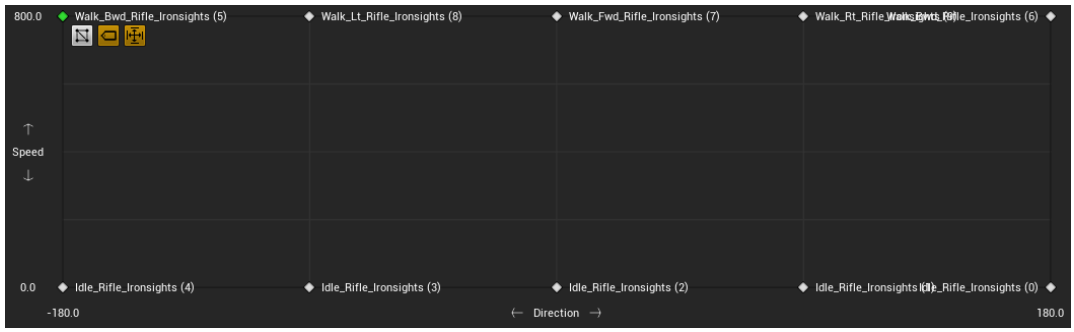

Figure 16.36: 2D Blend Space grid with all of the animations

31. On the **`Asset Details`** panel, set the **`Target Weight Interpolation Speed Per Sec`** variable to **`5`** to make the interpolation smoother.

32. Save and close the Blend Space.

    Next, you will need to create the input mappings:

33. Go to **`Project Settings`** and pick **`Input`** from the left panel, which is in the **`Engine`** category.

34. Create an action mapping called **`Jump`** that uses the spacebar key.

35. Create an axis mapping called **`Move Forward`** that uses the *W* key with scale **`1.0`** and *S* with scale **`-1.0`**.

36. Create an axis mapping called **`Move Right`** that uses the *A* key with scale **`-1.0`** and *D* with scale **`1.0`**.

37. Create an axis mapping called **`Turn`** that uses **`Mouse X`** with scale **`1.0`**.

38. Create an axis mapping called **`Look Up`** that uses **`Mouse Y`** with scale **`-1.0`**.

39. Close **`Project Settings`**.

    Now, create the C++ class for our character:

40. Create a new C++ class called **`FPSCharacter`** that derives from **`Character`** and, once it's compiled, delete all of the functions and variables declared and implemented, so we can start with a clean sheet.

41. Open **`FPSCharacter.h`** and declare the protected **`Camera`** component:

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "FPS
  Character")
class UCameraComponent* Camera;
```

42. Declare the protected **`Health`** variables:

```
UPROPERTY(Replicated, BlueprintReadOnly, Category = "FPS
  Character")
float Health = 0.0f;
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "FPS
  Character")
float MaxHealth = 100.0f;
```

43. Declare the protected **`Armor`** variables:

```
UPROPERTY(Replicated, BlueprintReadOnly, Category = "FPS
  Character")
float Armor = 0.0f;
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "FPS
  Character")
float MaxArmor = 100.0f;
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "FPS
  Character")
float ArmorAbsorption = 0.5;
```

44. Declare the protected **`SpawnSound`** variable:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "FPS
  Character")
USoundBase* SpawnSound;
```

45. Move the constructor, **`BeginPlay`** and **`SetupInputComponent`** to the protected area:

```
AFPSCharacter();
virtual void BeginPlay() override;
virtual void SetupPlayerInputComponent(class UInputComponent*
  PlayerInputComponent) override;
```

46. Declare the protected input handling functions as shown in the following code snippet:

```
void OnPressedJump();
void OnAxisMoveForward(float Value);
void OnAxisMoveRight(float Value);
void OnAxisLookUp(float Value);
void OnAxisTurn(float Value);
```

47. Declare and implement public the **`Health`** functions:

```
void AddHealth(float Amount) { SetHealth(Health + Amount); }
void RemoveHealth(float Amount) { SetHealth(Health - Amount); }
void SetHealth(float NewHealth) { Health =
  FMath::Clamp(NewHealth, 0.0f, MaxHealth); }
bool IsDead() const { return Health == 0.0f; }
```

48. Declare the **`Armor`** functions and implement the public **`AddArmor`** and **`SetArmor`** functions:

```
void AddArmor(float Amount) { SetArmor(Armor + Amount); }
void SetArmor(float Amount) { Armor = FMath::Clamp(Amount, 0.0f,
  MaxArmor); }
void ArmorAbsorbDamage(float & Damage);
```

49. Open **`FPSCharacter.cpp`** and include the **`UnrealNetwork.h`**, **`CameraComponent.h`**, **`CharacterMovementComponent.h`**, **`SkeletalMeshComponent.h`** and **`GameplayStatics.h`** header files:

```
#include "Net/UnrealNetwork.h"
#include "Camera/CameraComponent.h"
#include "GameFramework/CharacterMovementComponent.h"
#include "Components/SkeletalMeshComponent.h"
#include "Kismet/GameplayStatics.h"
```

50. In the constructor, initialize the **`Camera`** component and attach it to the **`Camera`** socket in the mesh:

```
Camera = CreateDefaultSubobject<UCameraComponent>("Camera");
Camera->bUsePawnControlRotation = true;
Camera->SetupAttachment(GetMesh(), "Camera");
```

51. Still in the constructor, set **`MaxWalkSpeed`** to **`800`** and **`JumpZVelocity`** to **`600`** in the character movement component:

```
GetCharacterMovement()->MaxWalkSpeed = 800.0f;
GetCharacterMovement()->JumpZVelocity = 600.0f;
```

52. To finalize the constructor, disable the **`Tick`** function, because we're not going to need it:

```
PrimaryActorTick.bCanEverTick = false;
```

53. In the **BeginPlay** function, play the spawn sound in every version of the character:

```
UGameplayStatics::PlaySound2D(GetWorld(), SpawnSound);
```

54. Still in **BeginPlay**, exit the function if the character doesn't have authority, because we're going to change an important variable:

```
if (!HasAuthority())
{
  return;
}
```

55. To finalize **BeginPlay**, initialize the **Health** variable with the value of **MaxHealth**:

```
SetHealth(MaxHealth);
```

56. In the **SetupPlayerInputComponent** function, bind the action mapping of **Jump** to the **OnPressedJump** function:

```
PlayerInputComponent->BindAction("Jump", IE_Pressed, this,
  &AFPSCharacter::OnPressedJump);
```

57. To finalize **SetupPlayerInputComponent**, bind the axis mappings for **Move Forward**, **Move Right**, **Look Up**, and **Turn**:

```
PlayerInputComponent->BindAxis("Move Forward", this,
  &AFPSCharacter::OnAxisMoveForward);
PlayerInputComponent->BindAxis("Move Right", this,
  &AFPSCharacter::OnAxisMoveRight);
PlayerInputComponent->BindAxis("Look Up", this,
  &AFPSCharacter::OnAxisLookUp);
PlayerInputComponent->BindAxis("Turn", this,
  &AFPSCharacter::OnAxisTurn);
```

58. Implement the **GetLifetimeReplicatedProps** function and make **Health** and **Armor** only replicate to the actor's owner:

```
void AFPSCharacter::GetLifetimeReplicatedProps(TArray<
  FLifetimeProperty >& OutLifetimeProps) const
{
  Super::GetLifetimeReplicatedProps(OutLifetimeProps);
  DOREPLIFETIME_CONDITION(AFPSCharacter, Health, COND_OwnerOnly);
  DOREPLIFETIME_CONDITION(AFPSCharacter, Armor, COND_OwnerOnly);
}
```

59. Implement the input handling functions:

```
void AFPSCharacter::OnPressedJump()
{
  Jump();
}
void AFPSCharacter::OnAxisMoveForward(float Value)
{
  AddMovementInput(GetActorForwardVector(), Value);
}
void AFPSCharacter::OnAxisMoveRight(float Value)
{
  AddMovementInput(GetActorRightVector(), Value);
}
void AFPSCharacter::OnAxisLookUp(float Value)
{
  AddControllerPitchInput(Value);
}
void AFPSCharacter::OnAxisTurn(float Value)
{
  AddControllerYawInput(Value);
}
```

60. Implement the **`ArmorAbsorbDamage`** function, which will calculate how much damage was absorbed by the armor and recalculate the damage using the supplied formula:

```
void AFPSCharacter::ArmorAbsorbDamage(float & Damage)
{
  const float AbsorbedDamage = Damage * ArmorAbsorption;
  const float RemainingArmor = Armor - AbsorbedDamage;

  SetArmor(RemainingArmor);

  Damage = (Damage * (1 - ArmorAbsorption)) -
  FMath::Min(RemainingArmor, 0.0f);
}
```

Now create the animation Blueprint for the player, following up on the concepts covered in *Chapter 2*, *Working with Unreal Engine*:

61. In the **`Content\Player\Animations`** folder, click on the **`Add New`** button and pick **`Animation`** | **`Animation Blueprint`**.
62. Set the parent class as **`AnimInstance`**, the skeleton as **`SK_Mannequin_Skeleton`**, and hit **`Ok`**.
63. Rename the animation Blueprint **`ABP_Player`** and open it.
64. In order to avoid always calling **`Try Get Pawn Owner`** on every tick in **`Event Blueprint Update Animation`**, override **`Event Blueprint Initialize Animation`**, and create a **`Character`** variable that stores the reference to **`FPSCharacter`**:

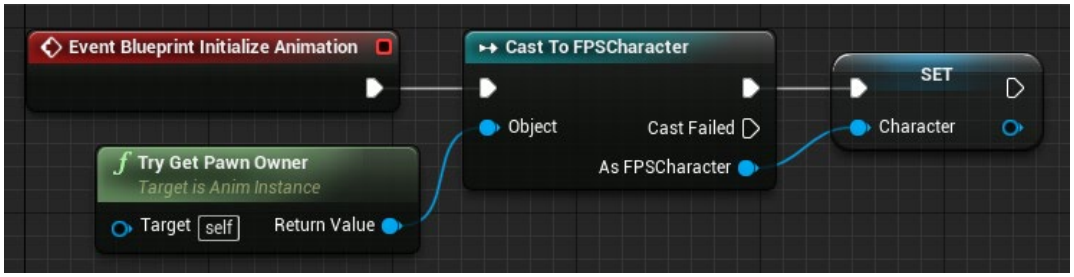

Figure 16.37: Overriding Event Blueprint Initialize Animation

65. Create the variables for **`Speed`** (float), **`Direction`** (float), **`Is Jumping`** (bool), and **`Pitch`** (float).
66. Assign the **`Speed`** value with **`VectorLengthXY`** from the velocity of the character:

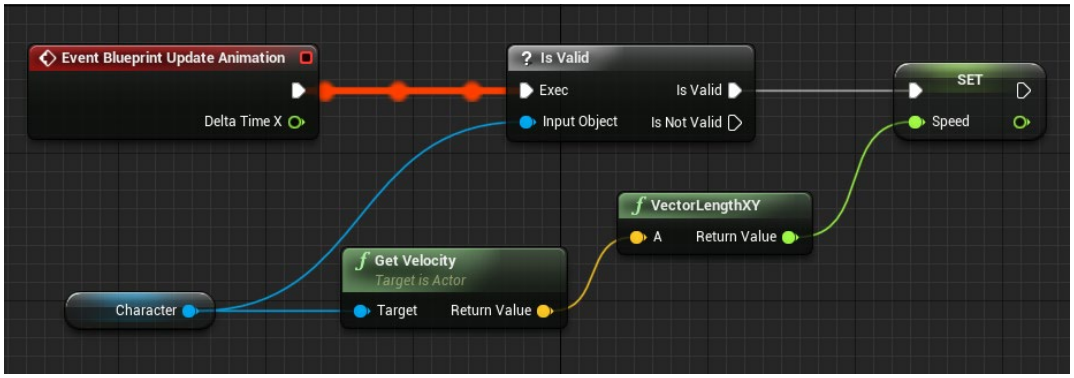

Figure 16.38: Calculating Speed from the character's velocity

67. Assign the **`Direction`** value with the **`Calculate Direction`** node between the velocity and the rotation of the character:

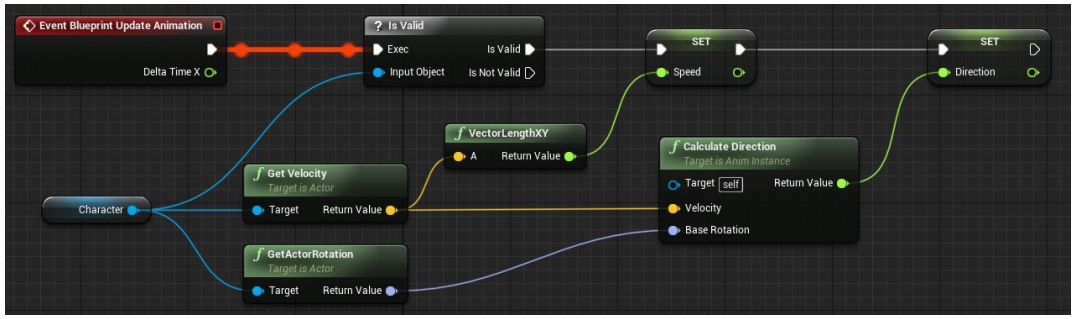

Figure 16.39: Calculating the direction from the velocity and the actor rotation

68. Assign the **`Is Jumping`** value with the result of the Boolean comparison if the **`Z`** component of the velocity is greater than **`0`**:

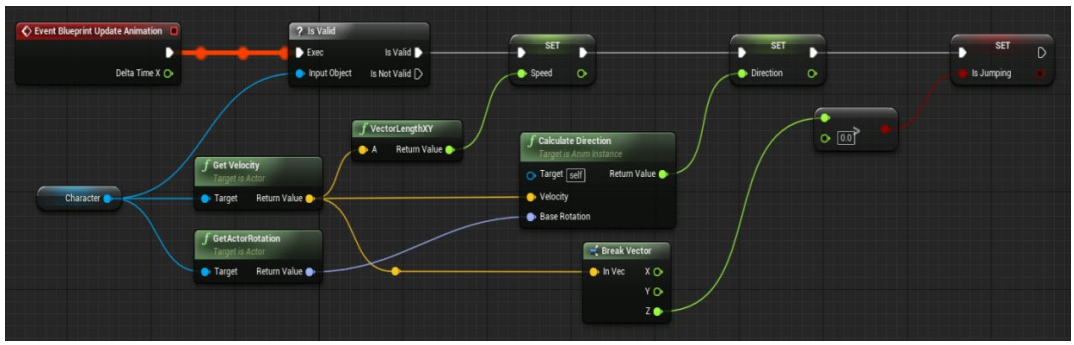

Figure 16.40: Calculating whether the character is jumping

69. Assign the **`Pitch`** value with the subtraction (or delta) between the character's rotation and the base aim rotation:

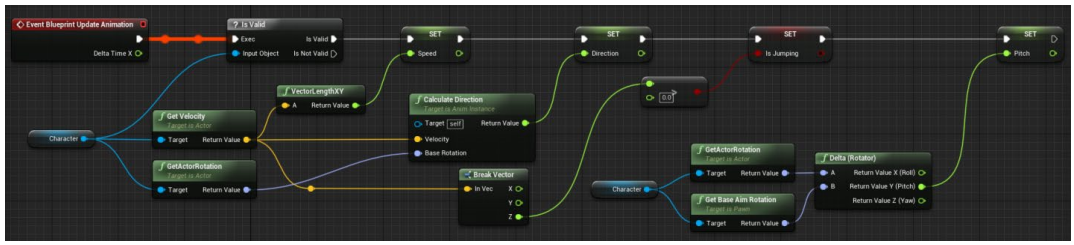

Figure 16.41: Calculating the Pitch

70. Go to the **`AnimGraph`** and create a **`State Machine`** called **`Movement`** that's connected to the *Output Pose* node.

71. *Double-click* the **`State Machine`** and create the **`Idle / Run`** and **`Jump`** states.
72. Connect **`Entry`** to **`Idle / Run`**.
73. Create a transition rule from **`Idle / Run`** to **`Jump`** and from **`Jump`** to **`Idle / Run`** like so:

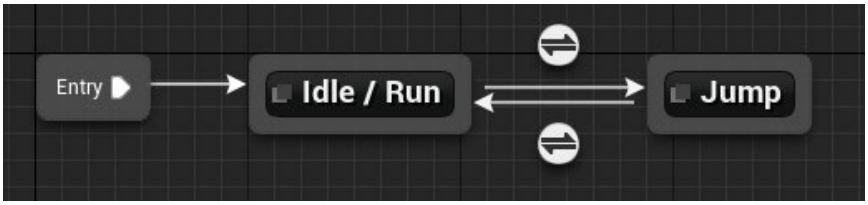

Figure 16.42: Transition rule from Idle / Run to Jump and vice versa

74. *Double-click* the **`Idle / Run`** to **`Jump`** transition rule and simply connect **`Is Jumping`** to **`Can Enter Transition`**:

Figure 16.43: Transition to the state if the character is jumping

75. Go back to the **`State Machine`** and *double-click* on the **`Jump`** to **`Idle / Run`** transition rule and set **`Can Enter Transition`** with the result of the Boolean comparison if **`Get Relevant Anim Time Remaining (Jump)`** is less than **`0.2`**:

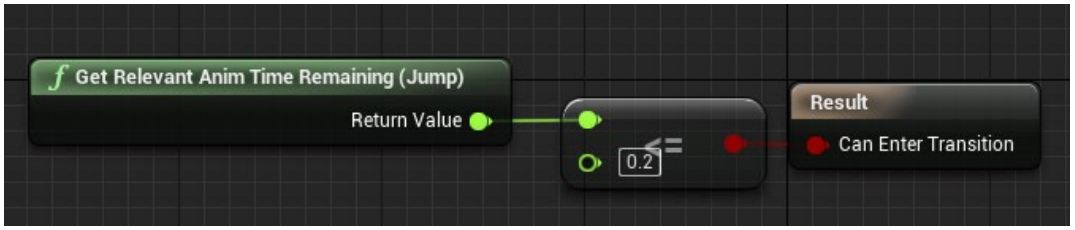

Figure 16.44: Transition to the state if the remaining time is less than or equal to 0.2 seconds

76. Go back to **`State Machine`** and open the **`Idle / Run`** state, add the 2D **`BS_Movement`** Blend Space, and make sure it's using the **`Speed`** and **`Direction`** variables:

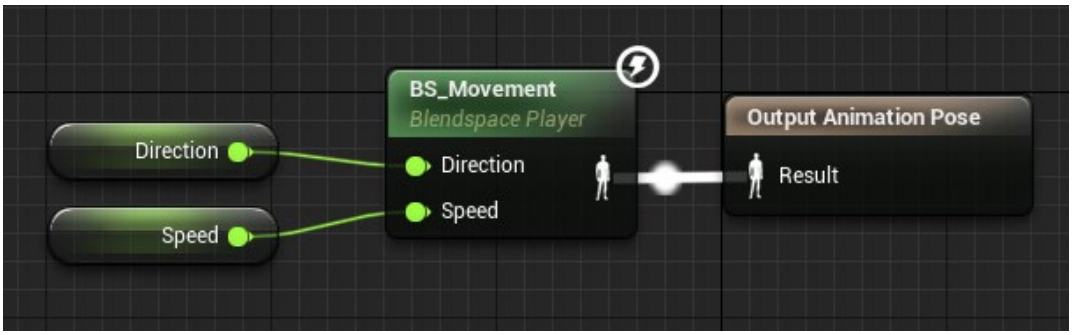

Figure 16.45: Blend Space using the Speed and Direction variables

77. Go back to the state machine, open the **`Jump`** state, and add the jump animation **`Jump_From_Stand_Ironsights`**:

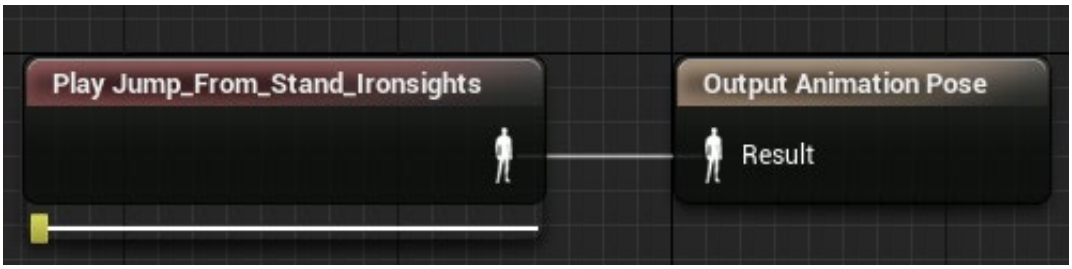

Figure 16.46: Making the Jump state use the Jump_From_Stand_Ironsights animation

78. Go to the **`AnimGraph`** and add a **`Transform (Modify) Bone`** node with the following settings:

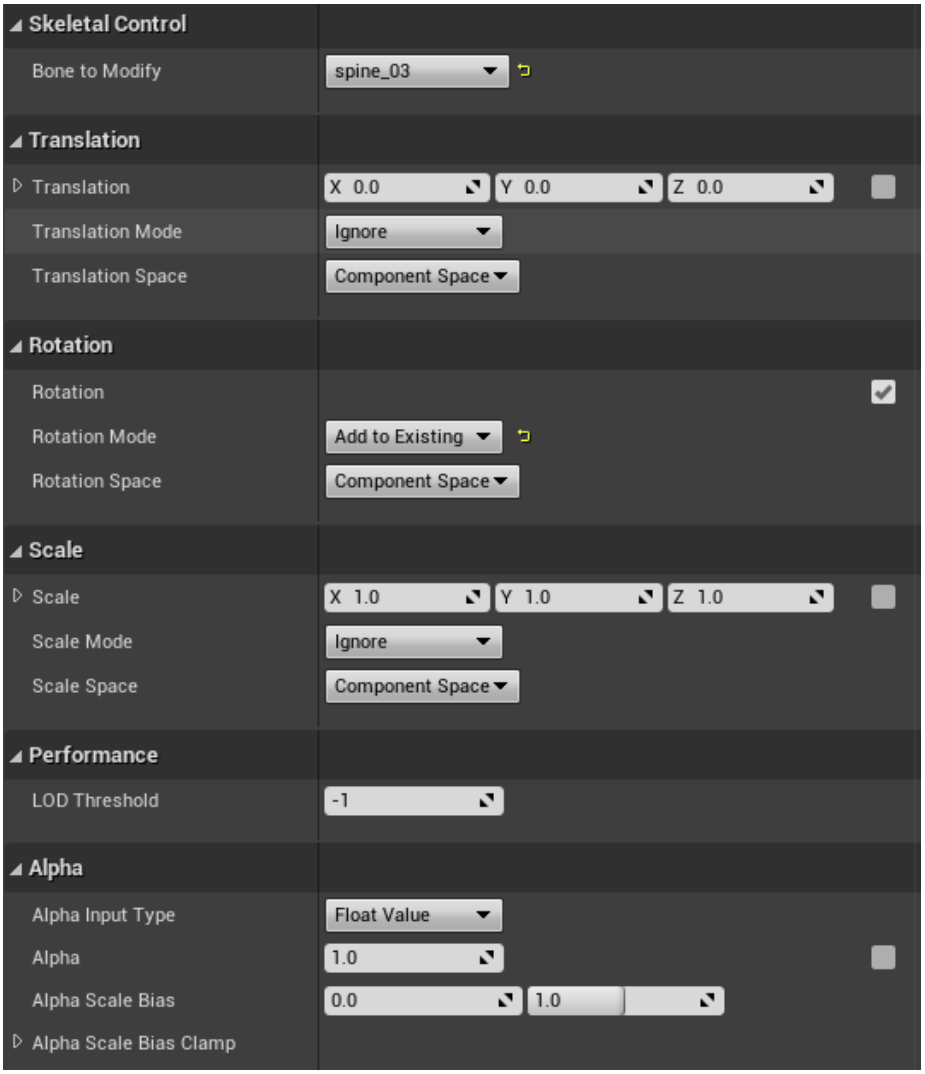

Figure 16.47: Transform (Modify) Bone settings

79. Connect the **`Transform (Modify) Bone`** node to **`State Machine`** and **`Output Pose`**:

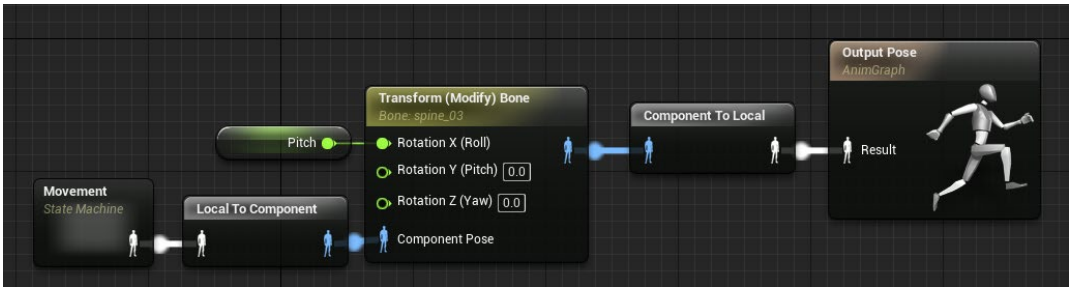

Figure 16.48: Transform (Modify) Bone node connected to State Machine and Output Pose

80. Save and close **`ABP_Player`**.

81. Now create the **`UMG`** widget that displays the health and the armor, following up on the concepts already explained in *Chapter 15*, *Collectibles, Power-ups, and Pickups*.

82. Create the **`Content\UI`** folder and open it.

83. Click on the **`Add New -> User Interface -> Widget`** Blueprint, rename it **`UI_HUD`**, and open it.

84. In the **`Designer`** section, create a vertical box anchored to the bottom-left corner with the following settings:

Figure 16.49: Vertical box canvas slot settings

85. Place two text blocks named **`tbHealth`** and **`tbArmor`** inside the vertical box.

86. Set their default text to be **`Health: 100`** and **`Armor: 100`**, so we can easily see what they need to display.

87. Create a text binding for **tbHealth** and make it return the health value of the owning player pawn:

Figure 16.50: Text binding for tbHealth

88. Create a text binding for **tbArmor** and make it return the armor value of the owning player pawn:

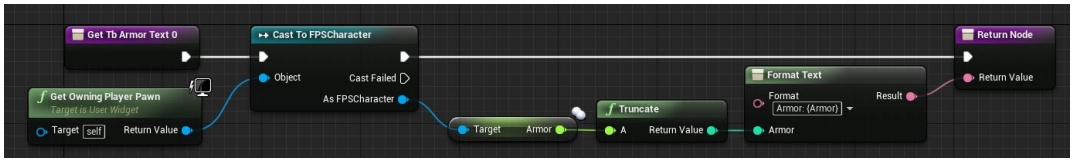

Figure 16.51: Text binding for tbArmor

89. Save and close **UI_HUD**.

    Now, create a new Blueprint for the player based on the **FPSCharacter** class, following up on the concepts already explained in *Chapter 3*, *Character Class Components and Blueprint Setup*:

90. Go to the **Content\Player** folder in the **Content Browser**, then create a new Blueprint called **BP_Player** that derives from **FPSCharacter**, and open it.

91. Select the mesh component and set its **Skeletal Mesh** to use **SK_Mannequin**.

92. Select the animation Blueprint to use **ABP_Player**.

93. Set the location to be (*X=0.0, Y=0.0, Z=-88.0*) and the rotation to be (*X=0.0, Y=0.0, Z=-90.0*):

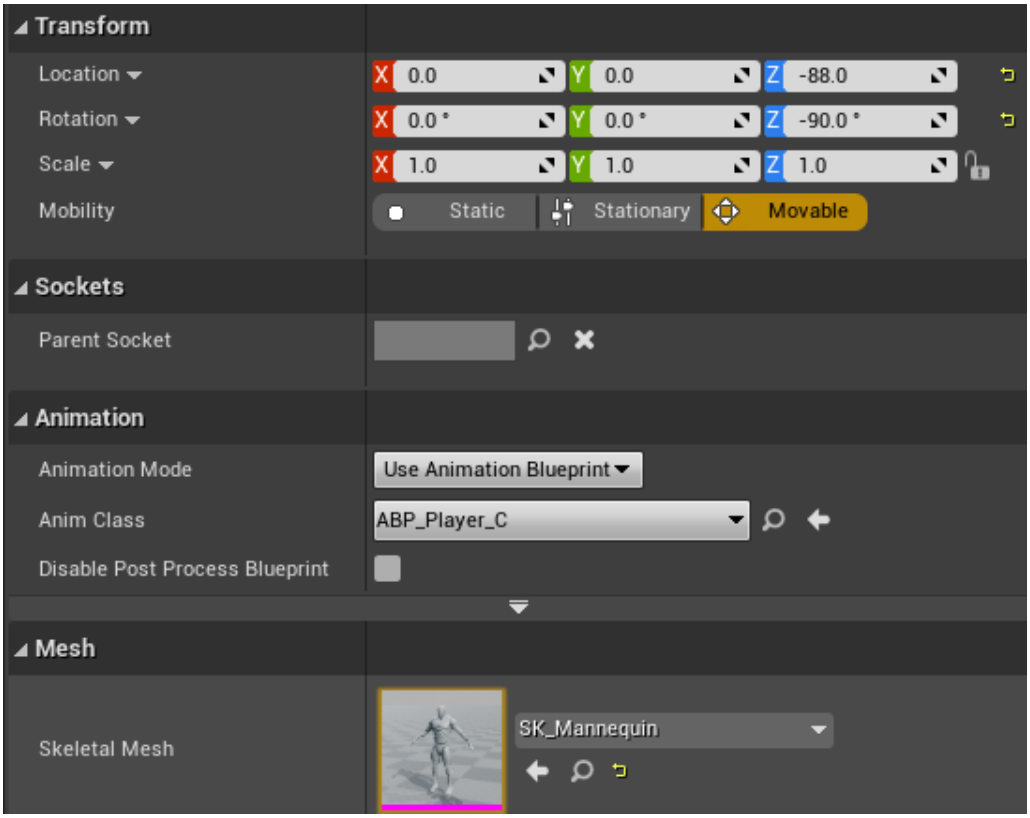

Figure 16.52: Skeletal Mesh component settings

94. Set the spawn sound to use **`Spawn`**:

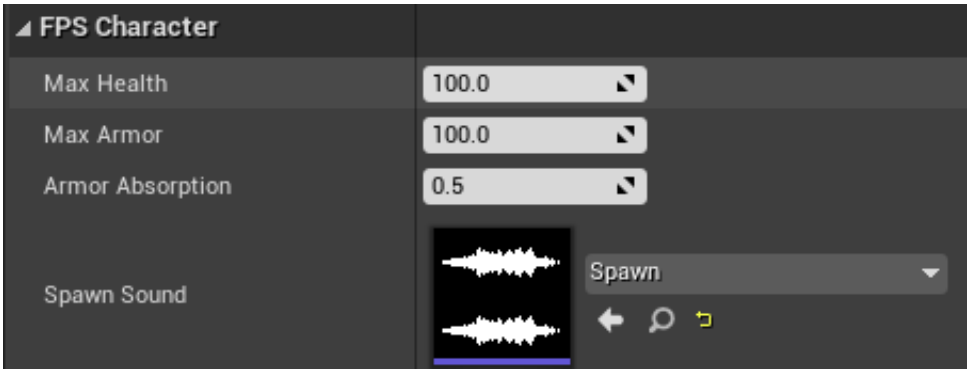

Figure 16.53: Setting Spawn Sound to use Spawn

95. Go to the event graph and on the **`BeginPlay`** event, use the **`Create Widget`** node to make an instance of **`UI_HUD`** and then add it to the viewport like so:

Figure 16.54: Creating the UI_HUD widget and adding it to the viewport

96. Save and close **`BP_Player`**.

    Now, save the test map we're going to use to play the game:

97. Create the **`Content\Maps`** folder and open it.

98. Make sure you are on the **`Untitled`** tab and go to **`File -> Save Current`**:

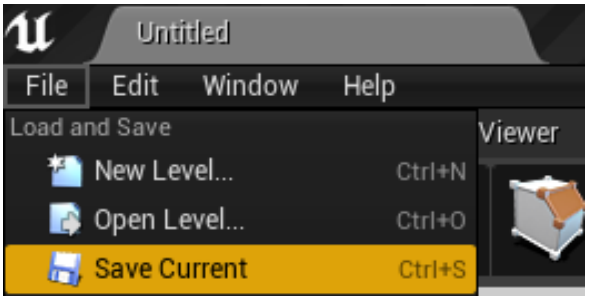

Figure 16.55: Saving the test map

99. Save it as **`DM-Test`** inside the **`Content\Maps`** folder.

    Next, we're going to create the **`Game Mode`** Blueprint, using what we have learned from *Chapter 3*, *Character Class Components and Blueprint Setup*:

100. Go to the **`Content`** folder and create a Blueprint that derives from **`MultiplayerFPSGameModeBase`**, rename it **`BP_GameMode`**, and open it.

101. Set the **`Default Pawn`** class to **`BP_Player`**:

Figure 16.56: Setting the Default Pawn class to use BP_Player

102. Save and close **`BP_GameMode`**.

To finish up, let's set the game mode we want to use, as well as the map to load by default when the editor and the packaged version run, following up on the concepts explained in *Chapter 3*, *Character Class Components and Blueprint Setup*:

103. Go to **`Project Settings`** and pick **`Maps & Modes`** from the left panel, which is in the **`Project`** category.

104. Set **`Default GameMode`** to use **`BP_GameMode`**.

105. In the **`Default Maps`** section, set **`Editor Startup Map`** and **`Game Default Map`** to use **`DM-Test`**:

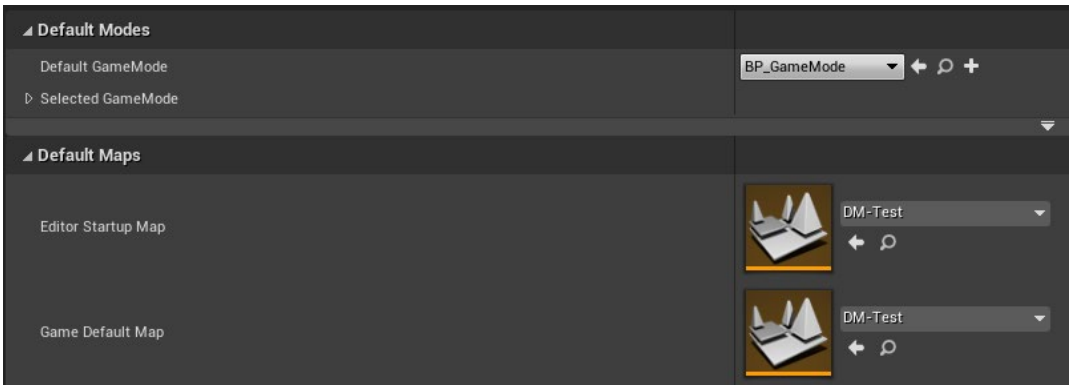

Figure 16.57: Setting the game mode to use BP_GameMode and the startup maps to use DM-Test

106. Close **`Project Settings`**.

107. We can finally test the project:

    Click on the down arrow next to the **`Play`** button and pick the last option, **`Advanced Settings`**.

108. In the **`Game Viewport Settings`** section, change the **`New Viewport Resolution`** to **`800x600`** and close the **`Editor Preferences`** tab.

109. Click again on the down arrow next to the **`Play`** button, set the number of clients to **`2`**, and click on **`New Editor Window (PIE)`**:

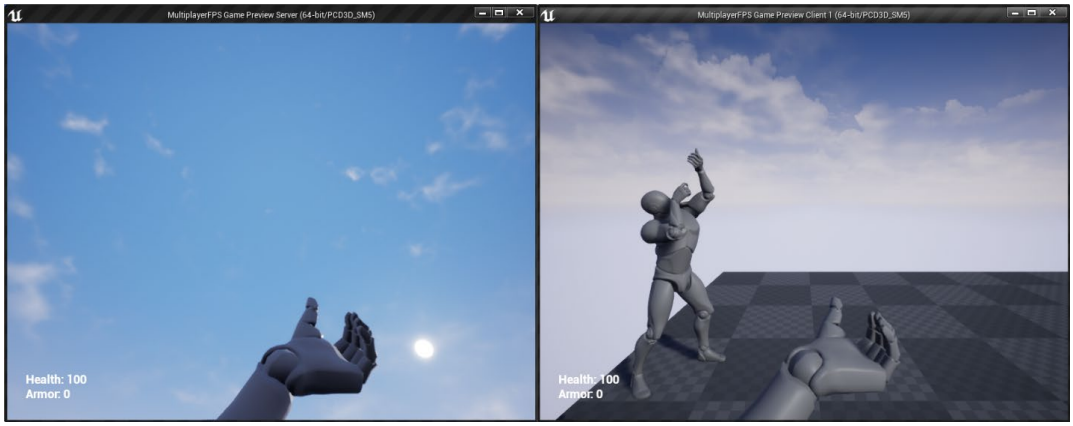

Figure 16.58: Expected output

CHAPTER 17: REMOTE PROCEDURE CALLS

ACTIVITY 17.01: ADDING WEAPONS AND AMMO TO THE MULTIPLAYER FPS GAME

1. Open the **`MultiplayerFPS`** project from *Activity 16.01*, *Creating a Character for the Multiplayer FPS Project*.
2. Compile the code and run the editor.

   Next, you need to import the new animations and create anim montages for them:

3. In **`Content Browser`**, go to the **`Content\Player\Animations`** folder.
4. Import the **`Pistol_Fire.fbx`**, **`MachineGun_Fire.fbx`**, and **`Railgun_Fire.fbx`** files and make sure they're using the **`SK_Mannequin_Skeleton`** skeleton.
5. Select the imported animations in the **`Content Browser`**, *right-click* on one of them and select **`Create -> Create AnimMontage`**.
6. Save everything, and you should get the following output:

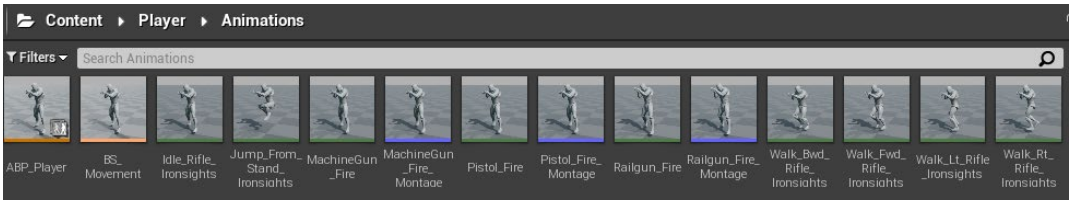

Figure 17.13: The animations folder of the player

Now, to add the anim montage Upper Body slot and configure the anim montages we've just created, perform the following steps:

7. Open **`Pistol_Fire_Montage`**.
8. Go to **`Window`** -> **`Anim Slot Manager`**.

9. Add a new slot called **`Upper Body`**:

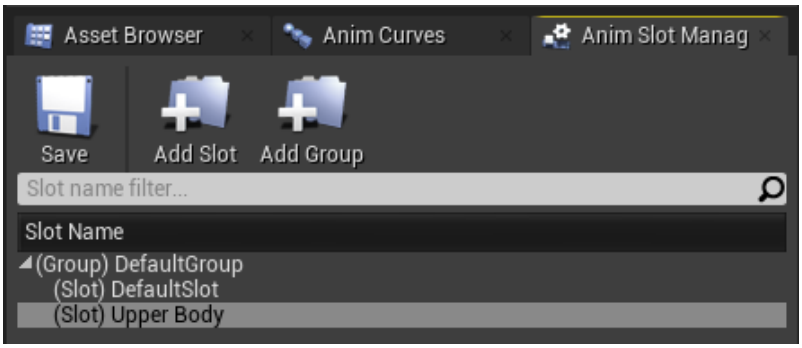

Figure 17.14: Creating the Upper Body slot

10. Set **`Pistol_Fire_Montage`** to use the **`Upper Body`** slot, like so:

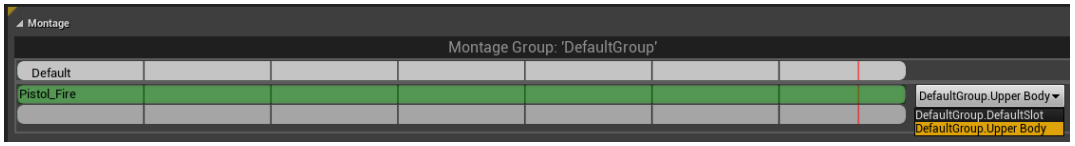

Figure 17.15: Setting the UpperBody slot

11. Set the **`Blend In`** time to **`0.01`** and the **`Blend Out`** time to **`0.1`**:

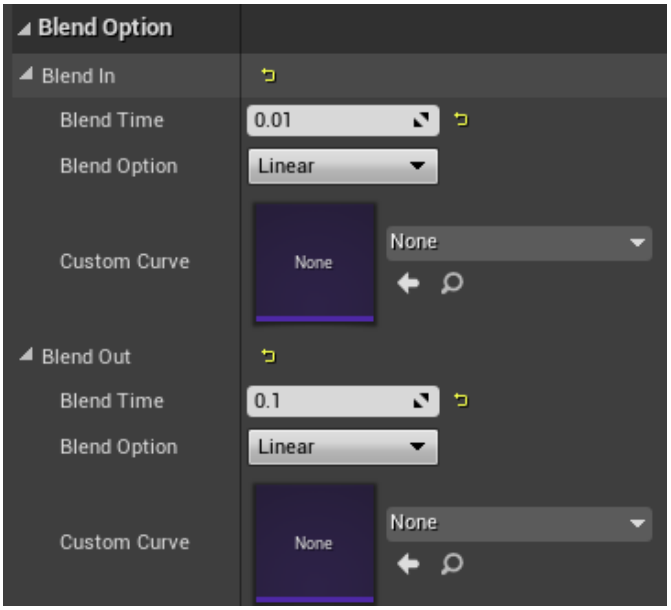

Figure 17.16: Setting the Blend In/Blend Out times for the pistol fire animation

12. Save and close **`Pistol_Fire_Montage`**.
13. Open **`MachineGun_Fire_Montage`** and set it to the **`Upper Body`** slot.
14. Set the **`Blend In`** time to **`0.01`** and the **`Blend Out`** time to **`0.1`**:

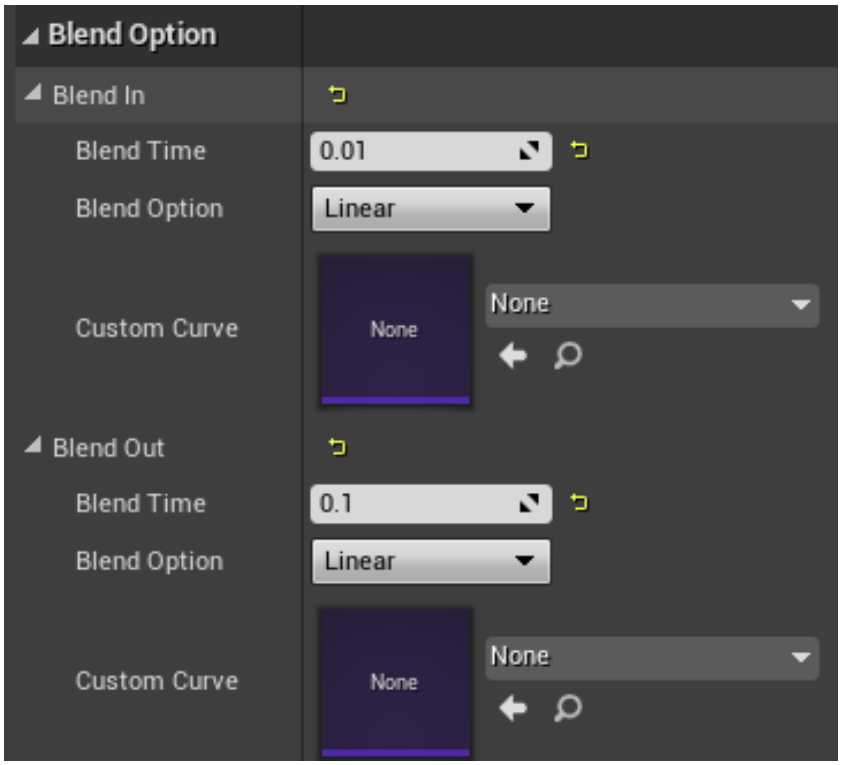

Figure 17.17: Setting the Blend In/Blend Out times for the machine gun fire animation

15. Save and close **`MachineGun_Fire_Montage`**.
16. Open **`Railgun_Fire_Montage`** and set it to the **`Upper Body`** slot.
17. Save and close **`Railgun_Fire_Montage`**.

    Now, let's create the remaining folders and import the remainder of the assets:

18. Create the **`Content\Weapons`** folder and, inside it, create three subfolders called **`Pistol`**, **`MachineGun`**, and **`Railgun`** as follows:

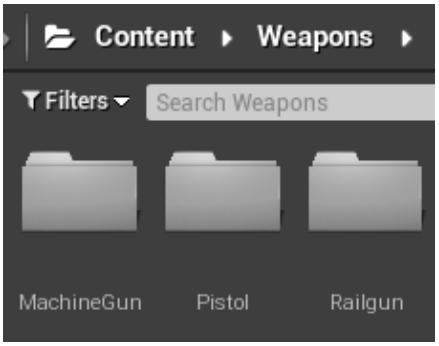

Figure 17.18: Creating the subfolders for each weapon type

19. Still in **`Content\Weapons`**, import **`SK_Weapon.fbx`** and the sounds **`NoAmmo.wav`**, **`WeaponChange.wav`**, and **`Hit.wav`**. Make sure your folder looks like this:

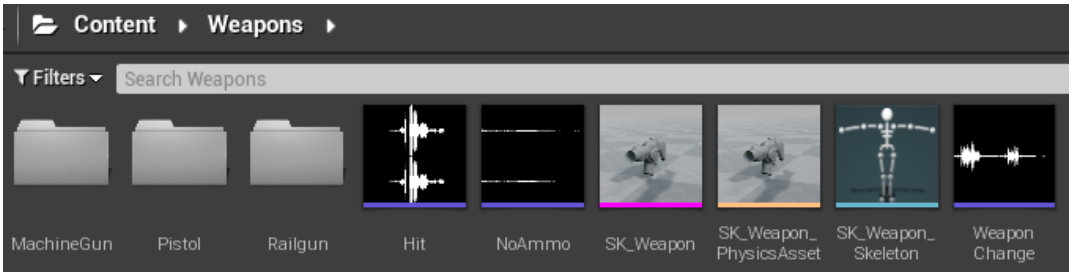

Figure 17.19: The Weapon folder

20. Go to **`Content\Weapons\Pistol`** and create a new material called **`M_Pistol`** and just set it to a green color in the base color.

21. Still in **`Content\Weapons\Pistol`**, import and save **`Pistol_Fire_Sound.wav`** from **`Activity17.01\Assets`**.

22. Go to **`Content\Player\Animations`** and open **`Pistol_Fire`**.

23. Add an Anim Notify Play Sound at time 0 sec using **`Pistol_Fire_Sound`** and place it on a frame that syncs well with the movement of the animation.

24. Save and close **`Pistol_Fire`**.

25. Go to **`Content\Weapons\MachineGun`**, create a new material called **`M_MachineGun`**, and just set it to a red color in the base color.

26. Still in **Content\Weapons\MachineGun**, import and save **MachineGun_Fire_Sound.wav** from **Activity17.01\Assets**.
27. Go to **Content\Player\Animations** and open **MachineGun_Fire**.
28. Add an Anim Notify Play Sound at time 0 sec using **MachineGun_Fire_Sound** and place it on a frame that syncs well with the movement of the animation.
29. Save and close **MachineGun_Fire**.
30. Go to **Content\Weapons\Railgun**, create a new material called **M_Railgun**, and just set it to a white color in the base color.
31. Still in **Content\Weapons\Railgun**, import and save **Railgun_Fire_Sound.wav** from **Activity17.01\Assets**.
32. Go to **Content\Player\Animations** and open **Railgun_Fire**.
33. Add an Anim Notify Play Sound at time 0 sec using **Railgun_Sound** and place it on a frame that syncs well with the movement of the animation.
34. Save and close **Railgun_Fire**.

    Next, let's create the **GripPoint** socket in the **SK_Mannequin** skeletal mesh.

35. Open **SK_Mannequin** and, in **Skeleton Tree**, *right-click* on **hand_r** and select **Add Socket**.
36. Rename the socket to **GripPoint**, select it, and use the following values:

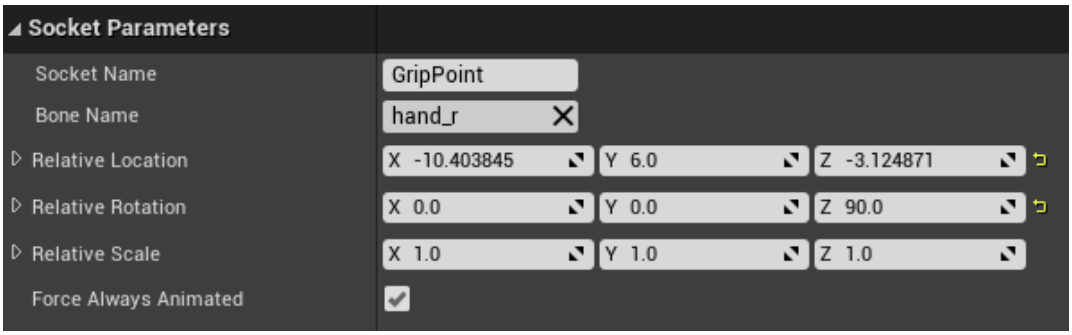

Figure 17.20: Setting the GripPoint socket values

Next, you'll need to set up the new input action bindings:

37. Go to **Project Settings** and pick **Input** from the left panel, which is in the Engine category.

38. Create an action mapping called **`Fire`**, using the *Left Mouse Button*.
39. Create an action mapping called the **`Previous Weapon`**, using Mouse Wheel Up.
40. Create an action mapping called **`Next Weapon`**, using Mouse Wheel Down.
41. Create an action mapping called **`Pistol`**, using the *1* key.
42. Create an action mapping called **`Machine Gun`**, using the *2* key.
43. Create an action mapping called **`Railgun`**, using the *3* key.
44. Close **`Project Settings`**.

    Next, you need to add some macros to **`MultiplayerFPS.h`** and create **`EnumTypes.h`** to store the enumerations:

45. Close the engine and go back to Visual Studio.
46. Open **`MultiplayerFPS.h`** and add the **`ENUM_TO_INT32`** and **`GET_CIRCULAR_ARRAY_INDEX`** macros:

```
#define ENUM_TO_INT32(Value) (int32)Value
#define GET_CIRCULAR_ARRAY_INDEX(Index, Count) (Index % Count +
  Count) % Count
```

47. In **`Solution Explorer`**, *right-click* on the **`Source\MultiplayerFPS`** folder and pick **`Add`** -> **`New Item`**:

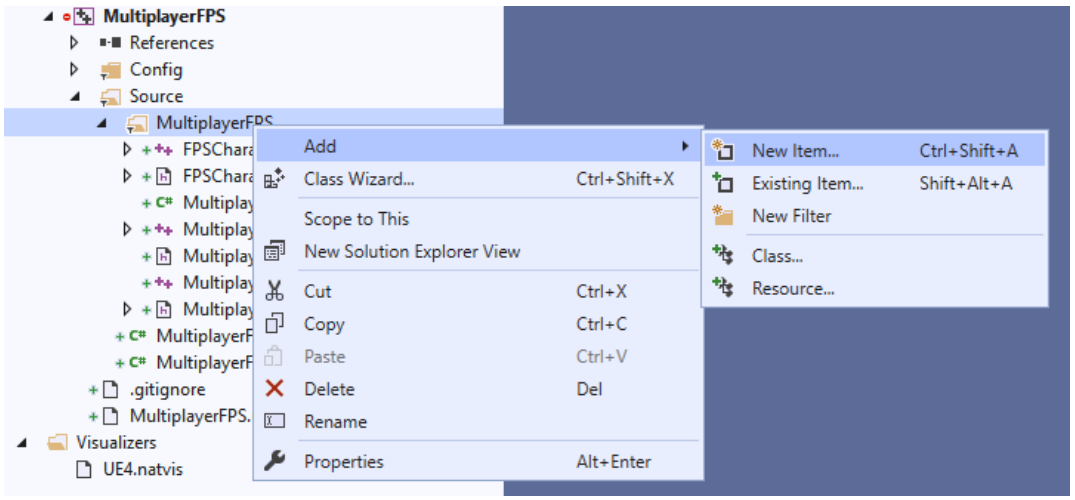

Figure 17.21: Creating the new EnumTypes.h header file

48. Pick **Header File (.h)** and name it **EnumTypes.h**, make sure it's saved in the **MultiplayerFPS\Source\MultiplayerFPS** folder, and then click **Add**.

49. In **EnumTypes.h**, add the **EWeaponType** enumeration:

```
UENUM()
enum class EWeaponType : uint8
{
  Pistol,
  MachineGun,
  Railgun,
  MAX
};
```

50. Add the **EWeaponFireMode** enumeration:

```
UENUM()
enum class EWeaponFireMode : uint8
{
  Single,
  Automatic
};
```

51. Add the **EAmmoType** enumeration:

```
UENUM()
enum class EAmmoType : uint8
{
  Bullets,
  Slugs,
  MAX
};
```

52. Save and close **EnumTypes.h**.

    Next, create the **Weapon C++** class using the editor:

53. Compile the code and run the editor.

54. Create a new C++ class called **Weapon** that derives from **Actor** and, once it's compiled, delete all of the functions and variables declared and implemented, so we can start with a clean sheet.

55. Open **`Weapon.h`** and include **`EnumTypes.h`** and **`AnimMontage.h`**:

```
#include "EnumTypes.h"
#include "Animation/AnimMontage.h"
```

56. Declare the protected skeletal mesh component called **`Mesh`**:

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
  "Weapon")
USkeletalMeshComponent* Mesh;
```

57. Declare the protected **`Name`** variable:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category =
  "Weapon")
FName Name;
```

58. Declare the protected variables for **`WeaponType`**, **`AmmoType`**, and **`FireMode`**:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category =
  "Weapon")
EWeaponType WeaponType;
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category =
  "Weapon")
EAmmoType AmmoType;
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category =
  "Weapon")
EWeaponFireMode FireMode;
```

59. Declare the protected variables for **`HitscanRange`**, **`HitscanDamage`**, and **`FireRate`**:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category =
  "Weapon")
float HitscanRange = 9999.9f;
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category =
  "Weapon")
float HitscanDamage = 20.0f;
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category =
  "Weapon")
float FireRate = 1.0f;
```

In the preceding code snippet, we declare the variables that indicate how far away we shoot the line trace, how much damage it does to the character that was hit, and how long, in seconds, each shot takes from the moment the player pressed the button to the moment it's able to fire again.

60. Declare the remaining protected variables:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category =
  "Weapon")
UAnimMontage* FireAnimMontage;
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category =
  "Weapon")
USoundBase* NoAmmoSound;

FTimerHandle FireTimer;

bool bWantsFire;

class AFPSCharacter* Character;
```

In the preceding code snippet, we declare the variables that tell what animation montage to use when firing the weapon, what sound to play when there is no more ammo, a timer to prevent firing while playing the fire animation, a state variable to state whether the trigger is being pressed with a view to controlling the automatic fire mode, and a reference to the character that owns the weapon.

61. Declare the protected constructor and the **`SetOwner`** override function:

```
AWeapon();
virtual void SetOwner(AActor* NewOwner) override;
```

62. Declare the protected functions, **`StartFire`** and **`FireHitscan`**:

```
void StartFire();
void FireHitscan(FVector FireLocation, FVector FireDirection);
```

63. Declare the protected functions, **`ServerStartFire`** and **`ServerStopFire`**:

```
UFUNCTION(Server, Reliable)
void ServerStartFire();
UFUNCTION(Server, Reliable)
void ServerStopFire();
```

64. Declare and implement the public function, **`GetAmmoType`**:

```
EAmmoType GetAmmoType() const { return AmmoType; }
```

65. Declare the public functions, **`OnPressedFire`** and **`OnReleasedFire`**:

```
virtual void OnPressedFire();
virtual void OnReleasedFire();
```

66. Go to **`Weapon.cpp`** and include **`FPSCharacter.h`**, **`TimerManager.h`** and **`SkeletalMeshComponent.h`**:

```
#include "FPSCharacter.h"
#include "TimerManager.h"
#include "Components/SkeletalMeshComponent.h"
```

67. In the constructor, initialize the skeletal mesh component and set it as the root component:

```
Mesh = CreateDefaultSubobject<USkeletalMeshComponent>("Mesh");
RootComponent = Mesh;
```

68. Still in the constructor, enable replication for this actor:

```
bReplicates = true;
```

69. To finalize the constructor, disable the tick function because we're not going to need it:

```
PrimaryActorTick.bCanEverTick = false;
```

70. Implement the **`SetOwner`** override function to store the reference to the character owner:

```
void AWeapon::SetOwner(AActor* NewOwner)
{
  Super::SetOwner(NewOwner);
  Character = Cast<AFPSCharacter>(NewOwner);
}
```

71. Implement the **`StartFire`** function:

```
void AWeapon::StartFire()
{
}
```

72. Abort if the trigger is no longer down or if there is still time remaining in **`FireTimer`**:

```
if (!bWantsFire ||
  GetWorldTimerManager().GetTimerRemaining(FireTimer) > 0.0f)
{
  return;
}
```

73. Next, we need to check if the character has ammo. We do this by calling **`GetAmmo`**, which we will implement in the next section. If the character doesn't have enough ammo, then call **`ClientPlaySound2D`** to play **`NoAmmoSound`** in the owning client and then abort:

```
if (Character->GetAmmo(AmmoType) == 0)
{
  if(NoAmmoSound != nullptr)
  {
    Character->ClientPlaySound2D(NoAmmoSound);
  }
  return;
}
```

74. Consume the ammo by calling **`ConsumeAmmo`** on the character, which we will implement in the next section:

```
Character->ConsumeAmmo(AmmoType, 1);
```

75. Get the character's camera location and direction to use on the **`FireHitscan`** function call:

```
const FVector FireLocation = Character->GetCameraLocation();
const FVector FireDirection = Character->GetCameraDirection();
FireHitscan(FireLocation, FireDirection);
```

76. Call the multicast RPC that will play the fire animation in all of the instances of the character:

```
if (FireAnimMontage != nullptr)
{
  Character->MulticastPlayAnimMontage(FireAnimMontage);
}
```

77. If the fire mode is automatic, then schedule the **`FireTimer`** variable to call **`StartFire`** again after **`FireRate`** seconds have passed to make it keep firing. If the fire mode is single, then just schedule the timer with **`FireRate`**, but without a callback, just to prevent the player from spamming the fire button:

```
if (FireMode == EWeaponFireMode::Automatic && bWantsFire)
{
  GetWorldTimerManager().SetTimer(FireTimer, this,
  &AWeapon::StartFire, FireRate);
}
else if (FireMode == EWeaponFireMode::Single)
```

```
{
  GetWorldTimerManager().SetTimer(FireTimer, FireRate, false);
}
```

78. Implement the **`FireHitscan`** function:

```
void AWeapon::FireHitscan(FVector FireLocation, FVector
  FireDirection)
{
}
```

79. Perform a line trace on the visibility channel that ignores the owning character and has a start and end location calculated using **`FireLocation`**, **`FireDirection`**, and **`HitscanRange`**:

```
FHitResult Hit(ForceInit);
FCollisionQueryParams TraceParams("Fire Trace", false,
  Character);
const FVector Start = FireLocation;
const FVector End = Start + FireDirection * HitscanRange;
GetWorld()->LineTraceSingleByChannel(Hit, Start, End,
  ECC_Visibility, TraceParams);
```

80. If **`Hit Actor`** is an **`FPSCharacter`**, then call the **`ApplyDamage`** function with **`HitscanDamage`** as the amount of damage and the owning character:

```
AFPSCharacter* HitCharacter =
  Cast<AFPSCharacter>(Hit.Actor.Get());
if (HitCharacter != nullptr)
{
  HitCharacter->ApplyDamage(HitscanDamage, Character);
}
```

81. Implement the **`ServerStartFire_Implementation`** function:

```
void AWeapon::ServerStartFire_Implementation()
{
  bWantsFire = true;

  StartFire();
}
```

82. Implement the **`ServerStopFire_Implementation`** function:

```
void AWeapon::ServerStopFire_Implementation()
{
  bWantsFire = false;
}
```

83. Implement the **`OnPressedFire`** function:

```
void AWeapon::OnPressedFire()
{
  ServerStartFire();
}
```

84. Implement the **`OnReleasedFire`** function:

```
void AWeapon::OnReleasedFire()
{
  ServerStopFire();
}
```

Now, to improve the **`FPSCharacter`** C++ class and add support for the weapons and ammo, perform the following steps:

85. Open **`FPSCharacter.h`** and include the header files **`MultiplayerFPS.h`**, **`EnumTypes.h`**, **`CameraComponent.h`**, and **`Weapon.h`**:

```
#include "MultiplayerFPS.h"
#include "EnumTypes.h"
#include "Camera/CameraComponent.h"
#include "Weapon.h"
```

86. Declare the protected ammo array variable:

```
UPROPERTY(Replicated, BlueprintReadOnly, Category = "FPS
  Character")
TArray<int32> Ammo;
```

87. Declare the protected weapon-related variables:

```
UPROPERTY(Replicated, BlueprintReadOnly, Category = "FPS
  Character")
TArray<AWeapon*> Weapons;
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "FPS
  Character")
TArray<TSubclassOf<AWeapon>> WeaponClasses;
UPROPERTY(Replicated, BlueprintReadOnly, Category = "FPS
  Character")
```

```
AWeapon* Weapon;
int32 WeaponIndex = INDEX_NONE;
```

In the preceding code snippet, we declare the variables that store the array of weapon instances that were spawned, the array of weapon classes to use when spawning the weapon instances, the reference to the currently equipped weapon, and the index of the currently equipped weapon.

88. Declare the protected sound-related variables:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "FPS
  Character")
USoundBase* HitSound;
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "FPS
  Character")
USoundBase* WeaponChangeSound;
```

89. Declare the protected functions that handle input:

```
void OnPressedFire();
void OnReleasedFire();
void OnPressedPistol();
void OnPressedMachineGun();
void OnPressedRailgun();
void OnPressedPreviousWeapon();
void OnPressedNextWeapon();
```

90. Declare the protected Server RPCs:

```
UFUNCTION(Server, Reliable)
void ServerCycleWeapons(int32 Direction);

UFUNCTION(Server, Reliable)
void ServerEquipWeapon(EWeaponType WeaponType);
```

91. Declare the protected **`EquipFunction`** function:

```
bool EquipWeapon(EWeaponType WeaponType, bool bPlaySound = true);
```

92. Declare the public **`ApplyDamage`** function:

```
void ApplyDamage(float Damage, AFPSCharacter* DamageCauser);
```

93. Declare the public RPC functions:

```
UFUNCTION(NetMulticast, Unreliable)
void MulticastPlayAnimMontage(UAnimMontage* AnimMontage);

UFUNCTION(Client, Unreliable)
void ClientPlaySound2D(USoundBase* Sound);
```

94. Declare the public **`AddWeapon`** function:

```
void AddWeapon(EWeaponType WeaponType);
```

95. Define the public ammo functions:

```
UFUNCTION(BlueprintCallable, Category = "FPS Character")
int32 GetWeaponAmmo() const { return Weapon != nullptr ?
  Ammo[ENUM_TO_INT32(Weapon->GetAmmoType())] : 0; }

void AddAmmo(EAmmoType AmmoType, int32 Amount) {
  SetAmmo(AmmoType, GetAmmo(AmmoType) + Amount); }

void ConsumeAmmo(EAmmoType AmmoType, int32 Amount) {
  SetAmmo(AmmoType, GetAmmo(AmmoType) - Amount); }

int32 GetAmmo(EAmmoType AmmoType) const { return
  Ammo[ENUM_TO_INT32(AmmoType)]; }

void SetAmmo(EAmmoType AmmoType, int32 Amount) {
  Ammo[ENUM_TO_INT32(AmmoType)] = FMath::Max(0, Amount); }
```

In the preceding code snippet, we define the functions that get the ammo count for the currently equipped weapon, add an amount of ammo of a certain type, consume an amount of ammo of a certain type, get the ammo count of a certain type, and set the amount of ammo of a certain type.

96. Define the public camera functions:

```
FVector GetCameraLocation() const { return Camera-
  >GetComponentLocation(); }
FVector GetCameraDirection() const { return
  GetControlRotation().Vector(); }
```

97. At the end of the **`BeginPlay`** function, initialize the **`Weapons`** array with the size of **`EWeaponType::MAX`** and the value of **`nullptr`** for each position:

```
const int32 WeaponCount = ENUM_TO_INT32(EWeaponType::MAX);
Weapons.Init(nullptr, WeaponCount);
```

98. Initialize the **`Ammo`** array with the size of **`EAmmoType::MAX`** and the value of **`50`** for each position:

```
const int32 AmmoCount = ENUM_TO_INT32(EAmmoType::MAX);
Ammo.Init(50, AmmoCount);
```

99. Add all of the weapons:

```
for (int32 i = 0; i < WeaponCount; i++)
{
  AddWeapon((EWeaponType)i);
}
```

100. Equip the machine gun to make sure there is always an equipped weapon:

```
EquipWeapon(EWeaponType::MachineGun, false);
```

101. Add the new action mappings to the end of the **`SetupPlayerInputComponent`** function:

```
PlayerInputComponent->BindAction("Fire", IE_Pressed, this,
  &AFPSCharacter::OnPressedFire);
PlayerInputComponent->BindAction("Fire", IE_Released, this,
  &AFPSCharacter::OnReleasedFire);

PlayerInputComponent->BindAction("Previous Weapon", IE_Pressed,
  this, &AFPSCharacter::OnPressedPreviousWeapon);
PlayerInputComponent->BindAction("Next Weapon", IE_Pressed, this,
  &AFPSCharacter::OnPressedNextWeapon);

PlayerInputComponent->BindAction("Pistol", IE_Pressed, this,
  &AFPSCharacter::OnPressedPistol);
PlayerInputComponent->BindAction("Machine Gun", IE_Pressed, this,
  &AFPSCharacter::OnPressedMachineGun);
PlayerInputComponent->BindAction("Railgun", IE_Pressed, this,
  &AFPSCharacter::OnPressedRailgun);
```

In the preceding code snippet, we bind the functions to handle the press/release of the **`Fire`** action mapping, the press of the **`Previous`**/**`Next`** weapon action mappings, and the press of the pistol, machine gun, and railgun action mappings.

102. Add the new replicated variables to the end of the **`GetLifetimeReplicatedProps`** function:

```
DOREPLIFETIME_CONDITION(AFPSCharacter, Weapon, COND_OwnerOnly);
DOREPLIFETIME_CONDITION(AFPSCharacter, Weapons, COND_OwnerOnly);
DOREPLIFETIME_CONDITION(AFPSCharacter, Ammo, COND_OwnerOnly);
```

103. Implement the **`OnPressedFire`** and **`OnReleaseFire`** functions:

```
void AFPSCharacter::OnPressedFire()
{
  if (Weapon != nullptr)
  {
      Weapon->OnPressedFire();
  }
}
void AFPSCharacter::OnReleasedFire()
{
  if (Weapon != nullptr)
  {
      Weapon->OnReleasedFire();
  }
}
```

In the preceding code snippet, we check in both functions whether the currently equipped weapon is valid. If it is, then execute the **`OnPressedFire`** and **`OnReleasedFire`** functions on the weapon instance.

104. Implement the **`OnPressedPistol`**, **`OnPressedMachineGun`**, and **`OnPressedRailgun`** functions:

```
void AFPSCharacter::OnPressedPistol()
{
  ServerEquipWeapon(EWeaponType::Pistol);
}
void AFPSCharacter::OnPressedMachineGun()
{
  ServerEquipWeapon(EWeaponType::MachineGun);
}
void AFPSCharacter::OnPressedRailgun()
{
  ServerEquipWeapon(EWeaponType::Railgun);
}
```

105. Implement the **`OnPressedPreviousWeapon`** and **`OnPressedNextWeapon`** functions:

```
void AFPSCharacter::OnPressedPreviousWeapon()
{
  ServerCycleWeapons(-1);
```

```
}
void AFPSCharacter::OnPressedNextWeapon()
{
  ServerCycleWeapons(1);
}
```

106. Implement the **`ServerCycleWeapons_Implementation`** function, which uses the **`GET_CIRCULAR_ARRAY_INDEX`** macro created earlier to cycle the weapons array in the direction that was supplied in the parameter. It will then perform a **`for`** loop to see which weapon can be equipped:

```
void AFPSCharacter::ServerCycleWeapons_Implementation(int32
  Direction)
{
  const int32 WeaponCount = Weapons.Num();
  const int32 StartWeaponIndex = GET_CIRCULAR_ARRAY_INDEX(WeaponIndex
+ Direction, WeaponCount);
  for (int32 i = StartWeaponIndex; i != WeaponIndex; i =
  GET_CIRCULAR_ARRAY_INDEX(i + Direction, WeaponCount))
  {
      if (EquipWeapon((EWeaponType)i))
      {
          break;
      }
  }
}
```

107. Implement the **`ServerEquipWeapon`** function:

```
void AFPSCharacter::ServerEquipWeapon_Implementation(EWeaponType
  WeaponType)
{
  EquipWeapon(WeaponType);
}
```

108. Implement the **`EquipWeapon`** function:

```
bool AFPSCharacter::EquipWeapon(EWeaponType WeaponType, bool
  bPlaySound)
{
}
```

109. Validate the equip functionality by checking for invalid indexes and by seeing whether we are trying to equip the same weapon:

```
const int32 NewWeaponIndex = ENUM_TO_INT32(WeaponType);
if (!Weapons.IsValidIndex(NewWeaponIndex))
{
  return false;
}
AWeapon* NewWeapon = Weapons[NewWeaponIndex];
if (NewWeapon == nullptr || Weapon == NewWeapon)
{
  return false;
}
```

110. Unequip the current weapon and equip the new weapon:

```
if (Weapon != nullptr)
{
  Weapon->SetActorHiddenInGame(true);
}
Weapon = NewWeapon;
WeaponIndex = NewWeaponIndex;
Weapon->SetActorHiddenInGame(false);
```

111. Play the weapon change sound only on the owning client:

```
if (WeaponChangeSound != nullptr && bPlaySound)
{
  ClientPlaySound2D(WeaponChangeSound);
}

return true;
```

112. Implement the **`AddWeapon`** function:

```
void AFPSCharacter::AddWeapon(EWeaponType WeaponType)
{
}
```

113. Validate the add by checking for invalid indexes and by seeing whether we already have that weapon:

```
const int32 NewWeaponIndex = ENUM_TO_INT32(WeaponType);
if (!WeaponClasses.IsValidIndex(NewWeaponIndex)
||     Weapons[NewWeaponIndex] != nullptr)
{
  return;
}
UClass* WeaponClass = WeaponClasses[NewWeaponIndex];
if (WeaponClass == nullptr)
{
  return;
}
```

114. Spawn the new weapon with this character as its owner:

```
FActorSpawnParameters SpawnParameters = FActorSpawnParameters();
SpawnParameters.Owner = this;
SpawnParameters.SpawnCollisionHandlingOverride =
  ESpawnActorCollisionHandlingMethod::AlwaysSpawn;
AWeapon* NewWeapon = GetWorld()->SpawnActor<AWeapon>(WeaponClass,
  SpawnParameters);
if (NewWeapon == nullptr)
{
  return;
}
```

115. Hide the weapon, assign it in the **`Weapons`** array, and attach it to the **`GripPoint`** socket in the mesh:

```
NewWeapon->SetActorHiddenInGame(true);
Weapons[NewWeaponIndex] = NewWeapon;
NewWeapon->AttachToComponent(GetMesh(),
FAttachmentTransformRules::SnapToTargetNotIncludingScale,
  "GripPoint");
```

116. Implement the **`MulticastPlayAnimMontage_Implementation`** and **`ClientPlaySound2D_Implementation`** functions:

```
void AFPSCharacter::MulticastPlayAnimMontage_Implementation(UanimMonta
  ge* AnimMontage)
{
  PlayAnimMontage(AnimMontage);
}
void AFPSCharacter::ClientPlaySound2D_Implementation(USoundBase*
```

```
    Sound)
{
  UGameplayStatics::PlaySound2D(GetWorld(), Sound);
}
```

117. Implement the **`ApplyDamage`** function, which will abort the function if the character is dead. If the character is not dead, then it will deduct the armor and the health and play the hit sound on the owning client of the **`DamageCauser`** actor:

```
void AFPSCharacter::ApplyDamage(float Damage, AFPSCharacter*
  DamageCauser)
{
  if (IsDead())
  {
    return;
  }
  ArmorAbsorbDamage(Damage);
  RemoveHealth(Damage);
  if (HitSound != nullptr && DamageCauser != nullptr)
  {
    DamageCauser->ClientPlaySound2D(HitSound);
  }
}
```

118. Compile the code and wait for the editor to load.

119. Go to **`Content\Weapons\Pistol`** and create **`BP_Pistol`**, which is a child of **`Weapon`** and configure it with the following values:

- Skeletal Mesh: **`Content\Weapons\SK_Weapon`**
- Material: **`Content\Weapons\Pistol\M_Pistol`**
- Name: **`Pistol Mk I`**
- Weapon Type: **`Pistol`**
- Ammo Type: **`Bullets`**
- Fire Mode: **`Automatic`**
- Hit Scan Range: **`9999.9`**
- Hit Scan Damage: **`5.0`**
- Fire Rate: **`0.5`**

- Fire Anim Montage: **Content\Player\Animations\Pistol_Fire_Montage**
- NoAmmoSound: **Content\Weapons\NoAmmo**

120. Save and close **BP_Pistol**.

121. Go to **Content\Weapons\MachineGun** and create **BP_MachineGun**, which is a child of **Weapon** and configure it in the following way:

- Skeletal Mesh: **Content\Weapons\SK_Weapon**
- Material: **Content\Weapons\MachineGun\M_MachineGun**
- Name: **Machine Gun Mk I**
- Weapon Type: **Machine Gun**
- Ammo Type: **Bullets**
- Fire Mode: **Automatic**
- Hit Scan Range: **9999.9**
- Hit Scan Damage: **5.0**
- Fire Rate: **0.1**
- Fire Anim Montage: **Content\Player\Animations\MachineGun_Fire_Montage**
- NoAmmoSound: **Content\Weapons\NoAmmo**

122. Save and close **BP_MachineGun**.

123. Go to **Content\Weapons\Railgun** and create **BP_Railgun**, which is a child of **Weapon** and configure it in the following way:

- Skeletal Mesh: **Content\Weapons\SK_Weapon**
- Material: **Content\Weapons\Railgun\M_Railgun**
- Name: **Railgun Mk I**
- Weapon Type: **Railgun**
- AmmoType: **Slugs**
- Fire Mode: **Single**
- Hit Scan Range: **9999.9**

- Hit Scan Damage: **`100.0`**
- Fire Rate: **`1.5`**
- Fire Anim Montage: **`Content\Player\Animations\Railgun_Fire_Montage`**
- No Ammo Sound: **`Content\Weapons\NoAmmo`**

124. Save and close **`BP_Railgun`**.

125. Go to **`Content\Player`** and open **`BP_Player`** and configure it with the following values:

- Weapon classes:

  Index 0: **`BP_Pistol`**

  Index 1: **`BP_MachineGun`**

  Index 2: **`BP_Railgun`**

- Hit Sound: **`Content\Weapons\Hit`**
- Weapon Change Sound: **`Content\Weapons\WeaponChange`**

126. Select the mesh component, set **`Collision Presets`** to **`Custom`**, and block the visibility channel so that it can be hit by the hitscans of the weapons:

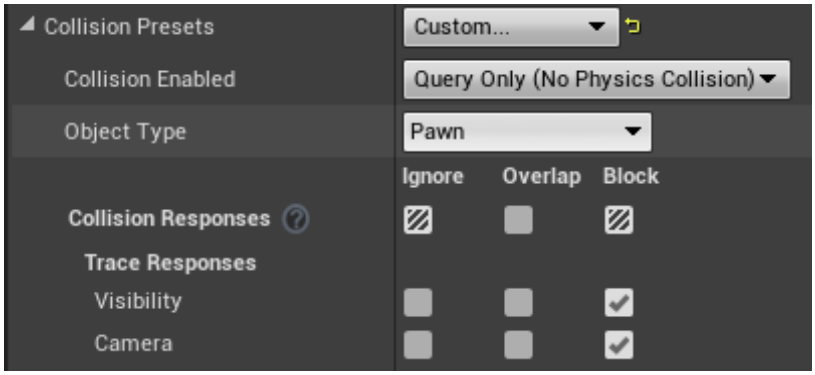

Figure 17.22: Using the Custom collision preset and setting the visibility channel to block

127. Save and close **`BP_Player`**.

128. Go to **`Content\Player\Animations`** and open **`ABP_Player`** with **`AnimGraph`**.

129. Cache the state machine and call it **`Movement Cache`**:

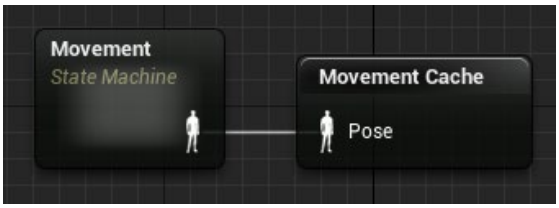

Figure 17.23: Caching the movement state machine

130. Add the **`Layered blend per bone`** node on **`spine_01`** and make sure **`Mesh Space Rotation Blend`** is enabled:

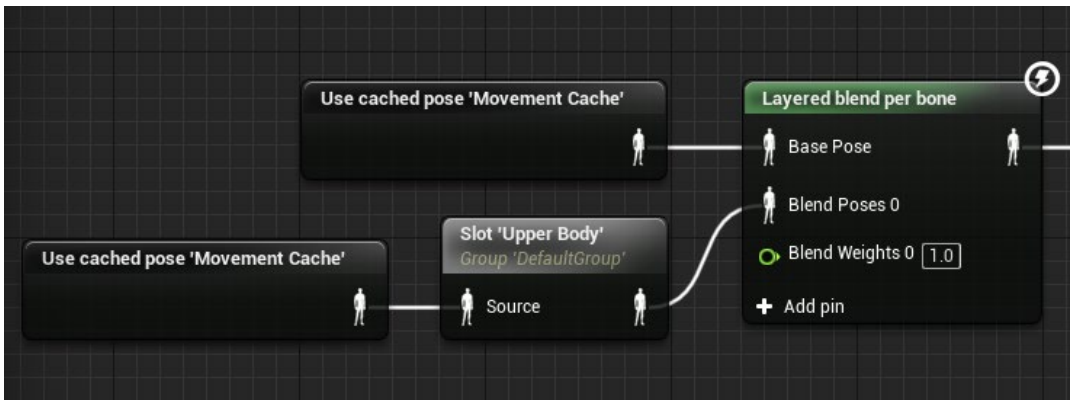

Figure 17.24: Adding a layered blend per bone node to play upper body animations

The layer setup is configured as follows:

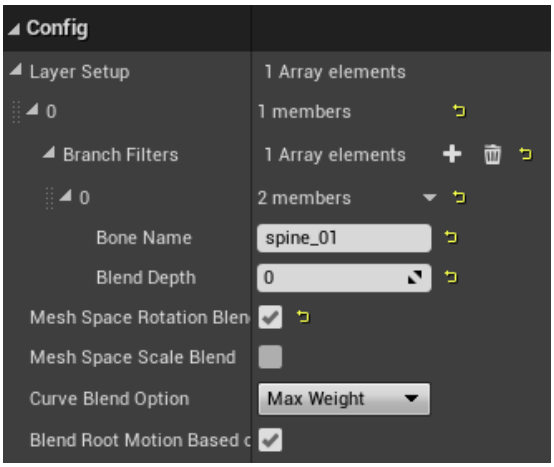

Figure 17.25: Configuring the layered blend per bone node

131. Save and close **`ABP_Player`**.

132. Go to **`Content\UI`** and open **`UI_HUD`**.

133. Add an image widget called **`imgCrosshair`** to the **`Canvas Panel`** root anchored in the center with the following settings:

Figure 17.26: Setting the size and alignment of imgCrosshair

134. Add a text block to the end of the vertical box called **`tbWeapon`** and set its default text to **`Machine Gun`**.

135. Create a text binding for **`tbWeapon`** and get the weapon's name with the following code:

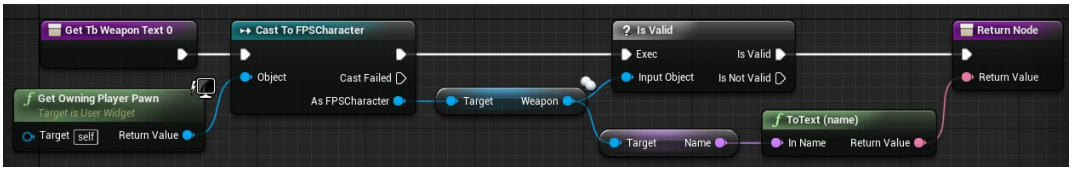

Figure 17.27: The function graph to get the name of the current weapon

136. Add a text block to the end of the vertical box called **`tbAmmo`** and set its default text to **`Ammo: 50`**.

137. Create a text binding for **`tbAmmo`** and get the weapon's ammo with the following code:

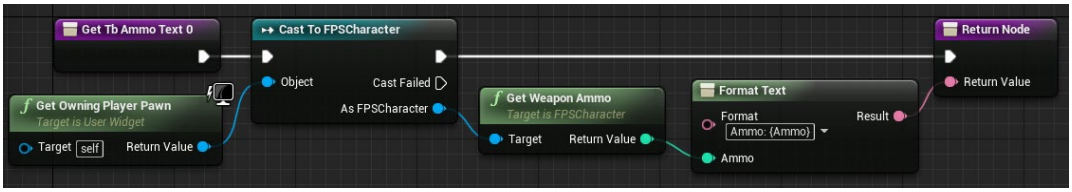

Figure 17.28: The function graph to get the ammo count of the current weapon

138. You should have the following hierarchy:

Figure 17.29: The hierarchy of the widgets in UI_HUD

139. Save and close **`UI_HUD`**.

    Now, let's test the project.

140. Click on the down arrow next to the **`Play`** button and pick the last option: **`Advanced Settings`**.

141. In the **`Game Viewport Settings`** section, change **`New Viewport Resolution`** to **`800x600`** and close the **`Editor Preferences`** tab.

142. Click again on the down arrow next to the **`Play`** button, set the number of clients to **`2`**, and click **`New Editor Window (PIE)`**:

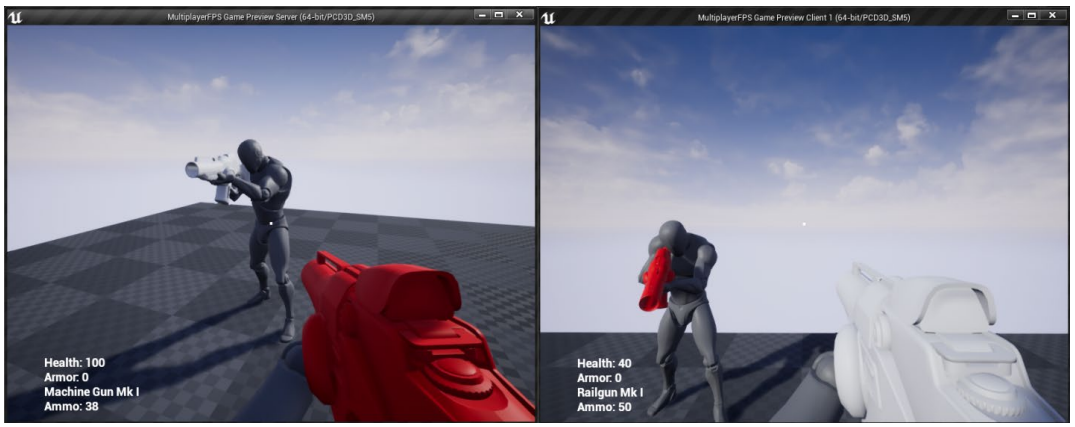

Figure 17.30: The end result of the activity

The end result is 2 characters that can carry 3 weapons, each with 50 ammo, which you can fire and damage the other players with. You can also select weapons by using the *1*, *2*, and *3* keys or by scrolling the mouse wheel up and down to get the previous and next weapons. The character will take damage when you hit its mesh, but right now, it won't die if the health reaches **0**. When a weapon is out of ammo, you'll hear the **`No Ammo Sound`** only in the owning client. You'll also see in the center of the HUD the crosshair, the name of the currently selected weapon, and how much ammo it has under the health and armor.

CHAPTER 18: GAMEPLAY FRAMEWORK CLASSES IN MULTIPLAYER

ACTIVITY 18.01: ADDING DEATH, RESPAWN, SCOREBOARD, KILL LIMIT, AND PICKUPS TO THE MULTIPLAYER FPS GAME

> **NOTE**
>
> The files for this activity can be found in our GitHub repository at: https://packt.live/3phl0TQ.

You begin by duplicating the project from *Activity 17.01*, *Adding Weapons and Ammo to the Multiplayer FPS Game*:

1. Copy the **`MultiplayerFPS`** project folder from *Activity 17.01*, *Adding Weapons and Ammo to the Multiplayer FPS Game*, and paste it in a new folder.
2. Open the new project folder and delete the **`Binaries`**, **`Intermediate`**, and **`Saved`** folders.
3. *Right-click* on **`MultiplayerFPS.uproject`** and pick **`Generate Visual Studio project files`**.
4. Open the Visual Studio solution once it's regenerated.
5. Compile and execute the code and wait for the editor to load.

   Now, let's create the new C++ classes we're going to use:

6. Create the **`FPSGameState`** class that is derived from **`GameState`**.
7. Create the **`FPSPlayerState`** class that is derived from **`PlayerState`**.
8. Create the **`FPSPlayerController`** class that is derived from **`PlayerController`**.
9. Create the **`PlayerMenu`** class that is derived from **`UserWidget`**.
10. Create the **`Pickup`** class that is derived from **`Actor`**.
11. Create the **`AmmoPickup`** class that is derived from **`Pickup`**.
12. Create the **`ArmorPickup`** class that is derived from **`Pickup`**.
13. Create the **`HealthPickup`** class that is derived from **`Pickup`**.

14. Create the **WeaponPickup** class that is derived from **Pickup**.

    Next, let's work on the **FPSGameState** class.

15. Open **FPSGameState.h** and declare the protected **KillLimit** variable:

```
UPROPERTY(Replicated, BlueprintReadOnly, Category = "FPS Game
  State")
int32 KillLimit;
```

16. Declare the public **GetPlayerStatesOrderedByKills** function, which returns an array of player states ordered by the number of kills:

```
UFUNCTION(BlueprintCallable)
TArray<class AFPSPlayerState*> GetPlayerStatesOrderedByKills() const;
```

17. Implement the public **SetKillLimit** function, which allows the game mode to set its kill limit on the game state:

```
void SetKillLimit(int32 NewKillLimit) { KillLimit = NewKillLimit;
  }
```

18. Open **FPSGameState.cpp** and include **UnrealNetwork.h** and **FPSPlayerState.h**:

```
#include "Net/UnrealNetwork.h"
#include "FPSPlayerState.h"
```

19. Implement the **GetLifetimeReplicatedProps** function and set **KillLimit** to replicate to all of the clients without an extra condition:

```
void AFPSGameState::GetLifetimeReplicatedProps(TArray<
  FLifetimeProperty >& OutLifetimeProps) const
{
  Super::GetLifetimeReplicatedProps(OutLifetimeProps);
  DOREPLIFETIME(AFPSGameState, KillLimit);
}
```

20. Implement the **GetPlayerStatesOrderedByKills** function, which goes through all of the player states and returns an array sorted by the number of kills:

```
TArray<AFPSPlayerState*>
  AFPSGameState::GetPlayerStatesOrderedByKills() const
{
  TArray<AFPSPlayerState*> PlayerStates;


  for (APlayerState* PlayerState : PlayerArray)
```

```
    {
      AFPSPlayerState* FPSPlayerState =
        Cast<AFPSPlayerState>(PlayerState);

      PlayerStates.Add(FPSPlayerState);
    }

    PlayerStates.Sort([](const AFPSPlayerState& A, const
      AFPSPlayerState& B) { return A.GetKills() > B.GetKills(); });

    return PlayerStates;
  }
```

Now, let's work on the **`FPSPlayerState`** class.

21. Open **`FPSPlayerState.h`** and declare the protected variables **`Kills`** and **`Deaths`**:

```
UPROPERTY(Replicated, BlueprintReadOnly, Category = "FPS Player
  State")
int32 Kills;

UPROPERTY(Replicated, BlueprintReadOnly, Category = "FPS Player
  State")
int32 Deaths;
```

22. Declare and implement the **`AddKill`**, **`AddDeath`**, and **`GetKills`** public functions:

```
void AddKill() { Kills++; }
void AddDeath() { Deaths++; }
int32 GetKills() const { return Kills; }
```

23. Open **`FPSPlayerState.cpp`** and include **`UnrealNetwork.h`**

```
#include "Net/UnrealNetwork.h"
```

24. implement the **`GetLifetimeReplicatedProps`** function, and set the **`Kills`** and **`Death`** variables to replicate to all of the clients without an extra condition:

```
void AFPSPlayerState::GetLifetimeReplicatedProps(TArray<
  FLifetimeProperty >& OutLifetimeProps) const
{
  Super::GetLifetimeReplicatedProps(OutLifetimeProps);
```

```
    DOREPLIFETIME(AFPSPlayerState, Kills);
    DOREPLIFETIME(AFPSPlayerState, Deaths);
}
```

Next, let's work on the **PlayerMenu** class.

25. Open **PlayerMenu.h** and declare the public **ToggleScoreboard** function, which will be implemented in blueprints:

```
UFUNCTION(BlueprintImplementableEvent)
void ToggleScoreboard();
```

26. Declare the public **SetScoreboardVisibility** function, which will be implemented in blueprints:

```
UFUNCTION(BlueprintImplementableEvent)
void SetScoreboardVisibility(bool bIsVisible);
```

27. Declare the public **NotifyKill** function, which will be implemented in blueprints:

```
UFUNCTION(BlueprintImplementableEvent)
void NotifyKill(const FString& Name);
```

Next, let's work on the **FPSPlayerController** class.

28. Open **FPSPlayerController.h** and declare the **PlayerMenuClass** and **PlayerMenu** public variables:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "FPS
  Player Controller")
TSubclassOf<class UPlayerMenu> PlayerMenuClass;


UPROPERTY()
class UPlayerMenu* PlayerMenu;
```

29. Override the protected **BeginPlay** function:

```
virtual void BeginPlay() override;
```

30. Declare the public **ToggleScoreboard** function:

```
void ToggleScoreboard();
```

31. Declare the public **ClientNotifyKill** RPC function:

```
UFUNCTION(Client, Reliable)
void ClientNotifyKill(const FString& Name);
```

32. Declare the public **`ClientShowScoreboard`** RPC function:

```
UFUNCTION(Client, Reliable)
void ClientShowScoreboard();
```

33. Open **`FPSPlayerController.cpp`** and include **`FPSCharacter.h`**, **`PlayerMenu.h`** and **`UserWidget.h`**.

```
#include "FPSCharacter.h"
#include "PlayerMenu.h"
#include "Blueprint/UserWidget.h"
```

34. Implement the override for **`BeginPlay`**:

```
void AFPSPlayerController::BeginPlay()
{
  Super::BeginPlay();

  if (!IsLocalController() || PlayerMenuClass == nullptr)
  {
    return;
  }

  PlayerMenu = CreateWidget<UPlayerMenu>(this, PlayerMenuClass);

  if (PlayerMenu != nullptr)
  {
    PlayerMenu->AddToViewport(0);
  }
}
```

In the preceding code snippet, we make sure that the controller is local and that the **`Player Menu`** class is valid. If everything checks out, then we can create the player menu widget by using the **`Player Menu`** class and add it to the viewport if the created instance is valid.

35. Implement the **`ToggleScoreboard`** function, which calls the **`ToggleScoreboard`** function in the **`PlayerMenu`** variable, if it's valid:

```
void AFPSPlayerController::ToggleScoreboard()
{
  if (PlayerMenu != nullptr)
  {
    PlayerMenu->ToggleScoreboard();
```

```
    }
}
```

36. Implement the **`ClientNotifyKill_Implementation`** function, which calls the **`NotifyKill`** function in the **`PlayerMenu`** variable, if it's valid:

```
void AFPSPlayerController::ClientNotifyKill_Implementation(const
  FString& Name)
{
  if (PlayerMenu != nullptr)
  {
    PlayerMenu->NotifyKill(Name);
  }
}
```

37. Implement the **`ClientShowScoreboard_Implementation`** function, which calls the **`SetScoreboardVisibility`** function with **`true`** on the **`PlayerMenu`** variable, if it's valid:

```
void AFPSPlayerController::ClientShowScoreboard_Implementation()
{
  if (PlayerMenu != nullptr)
  {
    PlayerMenu->SetScoreboardVisibility(true);
  }
}
```

Next, let's work on the **`Pickup`** class.

38. Open **`Pickup.h`** and declare the protected **`Static Mesh`** component variable called **`Mesh`**:

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
  "Pickup")
UStaticMeshComponent* Mesh;
```

39. Declare the protected rotating movement component variable called **`RotatingMovement`**:

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
  "Pickup")
Class URotatingMovementComponent* RotatingMovement;
```

40. Declare the protected variable **`RespawnTime`**, which indicates how long it takes to respawn the pickup when it's picked up by the player:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category =
  "Pickup")
float RespawnTime = 30.0f;
```

41. Declare the protected variable **`PickupSound`**, which tells the sound to play when it's picked up by the player:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category =
  "Pickup")
USoundBase* PickupSound;
```

42. Declare the protected variables for the respawn timer and whether the pickup is enabled:

```
FTimerHandle RespawnTimer;
bool bIsEnabled = true;
```

43. Move the constructor and **`BeginPlay`** to the protected area.

44. Declare the protected **`OnPickedUp`** function:

```
virtual void OnPickedUp(class AFPSCharacter* Character);
```

45. Declare the protected **`SetIsEnabled`** function:

```
void SetIsEnabled(bool NewbIsEnabled);
```

46. Declare the protected **`Respawn`** function:

```
void Respawn();
```

47. Declare the protected **`OnBeginOverlap`** function:

```
UFUNCTION()
void OnBeginOverlap(UPrimitiveComponent* OverlappedComp, AActor*
  OtherActor, UPrimitiveComponent* OtherComp, int32
  OtherBodyIndex, bool bFromSweep, const FHitResult& Hit);
```

48. Open **`Pickup.cpp`** and include **`FPSCharacter.h`**, **`StaticMeshComponent.h`**, **`RotatingMovementComponent.h`** and **`TimerManager.h`**:

```
#include "FPSCharacter.h"
#include "Components/StaticMeshComponent.h"
#include "GameFramework/RotatingMovementComponent.h"
#include "Engine/Public/TimerManager.h"
```

49. In the constructor, initialize the **`Mesh`** component (with a callback function for **`OnComponentBeginOverlap`**) and the **`RotatingMovement`** component with a permanent **`Yaw`** rotation of 90 degrees per second. It will also enable replication and disable the **`Tick`** function:

```
APickup::APickup()
{
  Mesh = CreateDefaultSubobject<UStaticMeshComponent>("Mesh");
  Mesh->SetCollisionProfileName("OverlapAll");
  RootComponent = Mesh;

  RotatingMovement = CreateDefaultSubobject
    <URotatingMovementComponent>("Rotating Movement");
  RotatingMovement->RotationRate = FRotator(0.0, 90.0f, 0);

  bReplicates = true;

  PrimaryActorTick.bCanEverTick = false;
}
```

50. Add to the **`BeginPlay`** function the bind for the begin overlap event:

```
void APickup::BeginPlay()
{
  Super::BeginPlay();

  Mesh->OnComponentBeginOverlap.AddDynamic(this,
    &APickup::OnBeginOverlap);
}
```

51. Implement the **`OnPickedUp`** function, which plays **`PickupSound`** only on the owning client of the character.

```
void APickup::OnPickedUp(AFPSCharacter* Character)
{
  if (PickupSound != nullptr)
  {
    Character->ClientPlaySound2D(PickupSound);
  }
}
```

52. Implement the **`SetIsEnabled`** function, which sets the value of **`bIsEnabled`** and whether the pickup is visible and can collide with the player:

```
void APickup::SetIsEnabled(bool NewbIsEnabled)
{
  bIsEnabled = NewbIsEnabled;

  SetActorHiddenInGame(!bIsEnabled);
  SetActorEnableCollision(bIsEnabled);
}
```

53. Implement the **`Respawn`** function, which calls **`SetIsEnabled`** with **`true`**:

```
void APickup::Respawn()
{
  SetIsEnabled(true);
}
```

54. Implement the **`OnBeginOverlap`** function, which checks if the character is valid and alive, as well as if it has authority. If everything checks out, then call **`OnPickedUp`** so it runs the appropriate implementation on its children, disable the pickup and schedule the respawn timer.

```
void APickup::OnBeginOverlap(UPrimitiveComponent* OverlappedComp,
  AActor* OtherActor, UPrimitiveComponent* OtherComp, int32
  OtherBodyIndex, bool bFromSweep, const FHitResult& Hit)
{
  AFPSCharacter* Character = Cast<AFPSCharacter>(OtherActor);

  if (Character == nullptr || Character->IsDead() ||
    !HasAuthority())
  {
    return;
  }

  OnPickedUp(Character);

  SetIsEnabled(false);

  GetWorldTimerManager().SetTimer(RespawnTimer, this,
    &APickup::Respawn, RespawnTime);
}
```

Next, let's work on the **`AmmoPickup`** class.

55. Open **AmmoPickup.h** and include EnumTypes.h:

```
#include "EnumTypes.h"
```

56. Declare the **AmmoType** and **AmmoAmount** protected variables:

```
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Ammo
  Pickup")
EAmmoType AmmoType;


UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Ammo
  Pickup")
int32 AmmoAmount;
```

57. Declare the override for the protected **OnPickedUp** function:

```
virtual void OnPickedUp(class AFPSCharacter* Character) override;
```

58. Open **AmmoPickup.cpp** and include **FPSCharacter.h**:

```
#include "FPSCharacter.h"
```

59. Implement the **OnPickedUp** function, which will execute the parent class implementation, but also add ammo to the character:

```
void AAmmoPickup::OnPickedUp(AFPSCharacter* Character)
{
  Super::OnPickedUp(Character);


  Character->AddAmmo(AmmoType, AmmoAmount);
}
```

Next, let's work on the **ArmorPickup** class.

60. Open **ArmorPickup.h** and declare the protected **ArmorAmount** variable, which defaults to **20**:

```
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Armor
  Pickup")
float ArmorAmount = 20.0f;
```

61. Declare the override for the **OnPickedUp** function:

```
virtual void OnPickedUp(class AFPSCharacter* Character) override;
```

62. Open **ArmorPickup.cpp** and include **FPSCharacter.h**:

```
#include "FPSCharacter.h"
```

63. Implement the **OnPickedUp** function, which will execute the parent class implementation, but also adds armor to the character:

```
void AArmorPickup::OnPickedUp(AFPSCharacter* Character)
{
  Super::OnPickedUp(Character);

  Character->AddArmor(ArmorAmount);
}
```

Next, let's work on the **HealthPickup** class.

64. Open **HealthPickup.h** and declare the protected **HealAmount** variable, which defaults to **20**:

```
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Health
  Pickup")
float HealAmount = 20.0f;
```

65. Declare the override for the **OnPickedUp** function:

```
virtual void OnPickedUp(class AFPSCharacter* Character) override;
```

66. Open **HealthPickup.cpp** and include **FPSCharacter.h**:

```
#include "FPSCharacter.h"
```

67. Implement the **OnPickedUp** function, which will execute the parent class implementation, but also adds health to the character:

```
void AHealthPickup::OnPickedUp(AFPSCharacter* Character)
{
  Super::OnPickedUp(Character);

  Character->AddHealth(HealAmount);
}
```

Next, let's work on the **WeaponPickup** class.

68. Open **WeaponPickup.h** and include **EnumTypes.h**:

```
#include "EnumTypes.h"
```

69. Declare the protected **`WeaponType`** and **`AmmoAmount`** variables:

```
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Weapon
  Pickup")
EWeaponType WeaponType;


UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Weapon
  Pickup")
int32 AmmoAmount;
```

70. Declare the override for the **`OnPickedUp`** function:

```
virtual void OnPickedUp(class AFPSCharacter* Character) override;
```

71. Open **`WeaponPickup.cpp`** and include **`FPSCharacter.h`**:

```
#include "FPSCharacter.h"
```

72. Implement the **`OnPickedUp`** function, which will execute the parent class implementation, but also adds the weapon and a little bit of ammo to the character:

```
void AWeaponPickup::OnPickedUp(AFPSCharacter* Character)
{
  Super::OnPickedUp(Character);


  Character->AddWeapon(WeaponType);
  Character->AddAmmo((EAmmoType)WeaponType, AmmoAmount);
}
```

Next, you need to add the new functionality to the **`FPSCharacter`** class.

73. Open **`FPSCharacter.h`** and add the new protected sound variables for pain and landing:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "FPS
  Character")
USoundBase* PainSound;


UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "FPS
  Character")
USoundBase* LandSound;
```

74. Add a variable that stores the reference to the game mode:

```
UPROPERTY()
class AMultiplayerFPSGameModeBase* GameMode;
```

75. Add the new protected override functions:

```
virtual void EndPlay(const EEndPlayReason::Type EndPlayReason)
  override;

virtual void FellOutOfWorld(const UDamageType& DmgType) override;

virtual void Landed(const FHitResult& Hit) override;
```

76. Add the input function that toggles the scoreboard:

```
void OnPressedScoreboard();
```

77. Open **`FPSCharacter.cpp`** and include **`CapsuleComponent.h`**, **`MultiplayerFPSGameModeBase.h`**, **`FPSPlayerController.h`**, and **`Weapon.h`**:

```
#include "Components/CapsuleComponent.h"
#include "MultiplayerFPSGameModeBase.h"
#include "FPSPlayerController.h"
#include "Weapon.h"
```

78. Store a reference to the game mode at the end of the **`BeginPlay`** function:

```
GameMode = Cast<AMultiplayerFPSGameModeBase>(GetWorld()-
  >GetAuthGameMode());
```

79. At the end of **`SetupPlayerInputComponent`**, add the bind to the callback function for the **`Scoreboard`** action:

```
PlayerInputComponent->BindAction("Scoreboard", IE_Pressed, this,
  &AFPSCharacter::OnPressedScoreboard);
```

80. Add the **`OnPressedScoreboard`** input function:

```
void AFPSCharacter::OnPressedScoreboard()
{
  AFPSPlayerController* PlayerController =
    Cast<AFPSPlayerController>(GetController());

  if (PlayerController != nullptr)
  {
    PlayerController->ToggleScoreboard();
  }
}
```

81. At the end of **`ApplyDamage`**, add a check to see whether the player is dead. If it is, then tell the game mode that the **`DamageCauser`** actor killed the character. If the player isn't dead, then play the pain sound only on the owning client:

```
if (IsDead())
{
  if (GameMode != nullptr)
  {
    GameMode->OnKill(DamageCauser->GetController(),
    GetController());
  }
  else
  {
    ClientPlaySound2D(PainSound);
  }
}
```

82. Implement the override for the **`FellOutOfWorld`** function, which calls the **`OnKill`** function in the game mode instead of calling the parent version of **`FellOutOfWorld`**, which just destroys the actor:

```
void AFPSCharacter::FellOutOfWorld(const UDamageType& DmgType)
{
  if (GameMode != nullptr)
  {
    GameMode->OnKill(nullptr, GetController());
  }
}
```

83. Implement the override for the **`EndPlay`** function that, besides calling the parent implementation, makes sure to destroy all of the character's weapons:

```
void AFPSCharacter::EndPlay(const EEndPlayReason::Type
  EndPlayReason)
{
  Super::EndPlay(EndPlayReason);

  // Destroy all of the weapons
```

```
    for (AWeapon* WeaponToDestroy : Weapons)
    {
      WeaponToDestroy->Destroy();
    }
  }
```

84. Implement the override for the **`Landed`** function, which plays the landed sound on every version of the character:

```
void AFPSCharacter::Landed(const FHitResult& Hit)
{
  Super::Landed(Hit);

  UGameplayStatics::PlaySound2D(GetWorld(), LandSound);
}
```

Next, we need to work on the **`MultiplayerFPSGameModeBase`** class.

85. Open **`MultiplayerFPSGameModeBase.h`** and remove the **`include`** to **`GameModeBase.h`** and replace it with **`GameMode.h`**:

```
#include "GameFramework/GameMode.h"
```

86. Change the parent class from **`AGameModeBase`** to **`AGameMode`** so that we can use matching states:

```
class MULTIPLAYERFPS_API AMultiplayerFPSGameModeBase : public
  AGameMode
```

87. Declare the protected variable for the kill limit, which, by default, is **`30`**:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category =
  "Multiplayer FPS Game Mode")
int32 KillLimit = 30;
```

88. Declare the protected constructor and overridden functions:

```
AMultiplayerFPSGameModeBase();


virtual bool ShouldSpawnAtStartSpot(AController* Player)
  override;
virtual void HandleMatchHasStarted() override;
virtual void HandleMatchHasEnded() override;
virtual bool ReadyToEndMatch_Implementation() override;
```

89. Declare the protected **`RestartMap`** function, which restarts the current map with server travel:

```
void RestartMap();
```

90. Declare the public **`HasWinner`** function, which returns if there is a player that reached the kill limit:

```
bool HasWinner() const;
```

91. Declare the public **`OnKill`** function, which handles matters when a player is killed:

```
void OnKill(AController* KillerController, AController*
  VictimController);
```

92. Open **`MultiplayerFPSGameModeBase.cpp`** and include **`FPSPlayerState.h`**, **`FPSGameState.h`**, **`FPSPlayerController.h`**, **`GameplayStatics.h`**, **`World.h`**, and **`TimerManager.h`**:

```
#include "FPSPlayerState.h"
#include "FPSGameState.h"
#include "FPSCharacter.h"
#include "FPSPlayerController.h"
#include "Kismet/GameplayStatics.h"
#include "Engine/World.h"
#include "Engine/Public/TimerManager.h"
```

93. Implement the constructor and set the default classes:

```
AMultiplayerFPSGameModeBase::AMultiplayerFPSGameModeBase()
{
  DefaultPawnClass = AFPSCharacter::StaticClass();
  PlayerControllerClass = AFPSPlayerController::StaticClass();
  PlayerStateClass = AFPSPlayerState::StaticClass();
  GameStateClass = AFPSGameState::StaticClass();
}
```

94. Implement the **`ShouldSpawnAtStartPoint`** function, which returns **`false`** to make sure every player spawns on a random player start instead of a fixed one:

```
bool AMultiplayerFPSGameModeBase::ShouldSpawnAtStartSpot(AController*
  Player)
{
  return false;
}
```

95. Implement the **`HandleMatchHasStarted`** function:

```
void AMultiplayerFPSGameModeBase::HandleMatchHasStarted()
{
  Super::HandleMatchHasStarted();

  // Tell the kill limit to the game state

  AFPSGameState* FPSGameState = Cast<AFPSGameState>(GameState);

  if (FPSGameState != nullptr)
  {
    FPSGameState->SetKillLimit(KillLimit);
  }
}
```

In the preceding code snippet, we define what should happen when the match starts. In this case, we still want to run the parent logic. After that, we set the kill limit on the game state.

96. Implement the **`HandleMatchHasEnded`** function, which goes through all of the player controllers to destroy its pawn and make it display the scoreboard. After that, it will schedule a timer to call the **`RestartMap`** function:

```
void AMultiplayerFPSGameModeBase::HandleMatchHasEnded()
{
  Super::HandleMatchHasEnded();

  TArray<AActor*> PlayerControllers;

  UGameplayStatics::GetAllActorsOfClass(this,
    AFPSPlayerController::StaticClass(), PlayerControllers);

  for (AActor* PlayerController : PlayerControllers)
```

```
    {
      AFPSPlayerController* FPSPlayerController =
      Cast<AFPSPlayerController>(PlayerController);

      if (FPSPlayerController == nullptr)
        continue;

      APawn* Pawn = FPSPlayerController->GetPawn();

      if (Pawn != nullptr)
      {
        Pawn->Destroy();
      }

      FPSPlayerController->ClientShowScoreboard();
    }

    FTimerHandle TimerHandle;
    GetWorldTimerManager().SetTimer(TimerHandle, this,
      &AMultiplayerFPSGameModeBase::RestartMap, 5.0f);
}
```

97. Implement the **`ReadyToEndMatch_Implementation`** function, which returns the condition to end the match, so in this case, whether we have a winner:

```
bool AMultiplayerFPSGameModeBase::ReadyToEndMatch_Implementation()
{
  return HasWinner();
}
```

98. Implement the **`HasWinner`** function:

```
bool AMultiplayerFPSGameModeBase::HasWinner() const
{
  for (APlayerState* PlayerState : GameState->PlayerArray)
  {
    AFPSPlayerState* FPSPlayerState =
      Cast<AFPSPlayerState>(PlayerState);

    if (FPSPlayerState != nullptr && FPSPlayerState->GetKills()
      == KillLimit)
    {
      return true;
```

```
      }
    }

    return false;
  }
```

In the preceding code snippet, we go through all of the player states and check whether any of the players have reached the specified kill limit.

99. Implement the **`OnKill`** function, which handles the death of a player:

```
void AMultiplayerFPSGameModeBase::OnKill(AController*
  KillerController, AController* VictimController)
{
}
```

100. Abort the function if the match is no longer in progress:

```
if (!IsMatchInProgress())
{
  return;
}
```

101. Add the kill to the killer and show it in the killer's character HUD:

```
if (KillerController != nullptr && KillerController !=
  VictimController)
{
  AFPSPlayerState* KillerPlayerState = Cast<AFPSPlayerState>(KillerCo
ntroller->PlayerState);

  if (KillerPlayerState != nullptr)
  {
    KillerPlayerState->AddKill();
  }

  AFPSPlayerController* KillerFPSController =
    Cast<AFPSPlayerController>(KillerController);

  if (KillerFPSController != nullptr && VictimController !=
    nullptr && VictimController->PlayerState != nullptr)
  {
    KillerFPSController->
```

```
      ClientNotifyKill(VictimController->PlayerState->
      GetPlayerName());
    }
  }
```

In the preceding code snippet, we check whether the killer controller is valid and whether it's different from the victim's controller. We do this to make sure there was a killer (if you fall from the level, the killer controller will be invalid, so we need to contemplate this case) and to make sure we don't add a kill to the kill counter if the player kills itself accidentally. If everything checks out, then we add the kill to the killer's player state and send an RPC just to the killer client, which displays a UI message displaying which player was killed.

102. Add the death to the victim, destroy the controlled pawn, and only respawn the player if it wasn't the winning kill:

```
if (VictimController != nullptr)
{
  AFPSPlayerState* VictimPlayerState =
    Cast<AFPSPlayerState>(VictimController->PlayerState);

  if (VictimPlayerState != nullptr)
  {
    VictimPlayerState->AddDeath();
  }

  APawn* Pawn = VictimController->GetPawn();

  if (Pawn != nullptr)
  {
    Pawn->Destroy();
  }

  if (!HasWinner())
  {
    RestartPlayer(VictimController);
  }
}
```

In the preceding code snippet, we check whether we have a valid victim controller. If we do, then we add a death to the victim's player state, destroy its controlled pawn, and respawn a new pawn for the player only if it wasn't the winning kill.

103. Implement **`RestartMap`**, which restarts the current map with server travel:

```
void AMultiplayerFPSGameModeBase::RestartMap()
{
  GetWorld()->ServerTravel(GetWorld()->GetName(), false, false);
}
```

104. Compile and execute the code and wait for the editor to load.

Next, we're going to create the different types of pickup.

105. In **`Content Browser`**, create the **`Content\Pickups`** folder and open it.

106. Create a folder for each type of pickup (**`Ammo`**, **`Armor`**, **`Health`**, and **`Weapon`**).

107. Open the **`Content\Pickups\Ammo`** folder and import the **`AmmoPickup.wav`** sound from the **`Activity18.01\Assets`** folder.

108. Save **`AmmoPickup`**.

109. Create and open a new blueprint called **`BP_PistolBullets_Pickup`** that is derived from the **`AmmoPickup`** class and configure it with the following values:

- Scale: **`(X=0.5, Y=0.5, Z=0.5)`**
- Static Mesh: **`Engine\BasicShapes\Cube`**
- Material: **`Content\Weapon\Pistol\M_Pistol`**
- Ammo Type: **`Pistol Bullets`**
- Ammo Amount: **`25`**
- Pickup Sound: **`Content\Pickup\Ammo\AmmoPickup`**

110. Save and close **`BP_PistolBullets_Pickup`**.

111. Create and open a new blueprint called **`BP_MachineGunBullets_Pickup`** that is derived from the **`AmmoPickup`** class and configure it with the following values:

- Scale: **`(X=0.5, Y=0.5, Z=0.5)`**
- Static Mesh: **`Engine\BasicShapes\Cube`**
- Material: **`Content\Weapon\MachineGun\M_MachineGun`**
- Ammo Type: **`Machine Gun Bullets`**
- Ammo Amount: **`50`**

- Pickup Sound: **Content\Pickup\Ammo\AmmoPickup**

112. Save and close **BP_MachineGunBullets_Pickup**.

113. Create and open a new blueprint called **BP_Slugs_Pickup** that is derived from the **AmmoPickup** class and configure it with the following values:

- Scale: (**X=0.5, Y=0.5, Z=0.5)**
- Static Mesh: **Engine\BasicShapes\Cube**
- Material: **Content\Weapon\Railgun\M_Railgun**
- Ammo Type: **Slugs**
- Ammo Amount: **5**
- Pickup Sound: **Content\Pickup\Ammo\AmmoPickup**

114. Save and close **BP_Slugs_Pickup**.

115. Open the **Content\Pickups\Armor** folder and import the **ArmorPickup.wav** sound from the **Activity18.01\Assets** folder.

116. Save **ArmorPickup**.

117. Create and open a new material called **M_Armor**, which has **Base Color** set to **blue** and **Metallic** set to **1**.

118. Save and close **M_Armor**.

119. Create and open a new blueprint called **BP_Armor_Pickup** that is derived from the **ArmorPickup** class and configure it with the following values:

- Scale: **(X=1.0, Y=1.5, Z=1.0)**
- Static Mesh: **Engine\BasicShapes\Cube**
- Material: **Content\Pickup\Armor\M_Armor**
- Armor Amount: **50**
- Pickup Sound: **Content\Pickup\Armor\ArmorPickup**

120. Save and close **BP_Armor_Pickup**.

121. Open the **Content\Pickups\Health** folder and import the **HealthPickup.wav** sound from the **Activity18.01\Assets** folder.

122. Save **HealthPickup**.

123. Create and open a new material called **M_Health**, which has **Base Color** set to **blue** and **Metallic/Roughness** set to **0.5**.

124. Save and close **M_Health**.

125. Create and open a new blueprint called **BP_Health_Pickup** that is derived from the **HealthPickup** class and configure it with the following values:

- Static Mesh: **Engine\BasicShapes\Sphere**
- Material: **Content\Pickup\Health\M_Health**
- Health Amount: **50**
- Pickup Sound: **Content\Pickup\Health\HealthPickup**

126. Save and close **BP_Health_Pickup**.

127. Open the **Content\Pickups\Weapon** folder and import the **WeaponPickup.wav** sound and the **SM_Weapon.fbx** static mesh from the **Activity18.01\Assets** folder.

128. Save **WeaponPickup** and **SM_Weapon**.

129. Create and open a new blueprint called **BP_Pistol_Pickup** that is derived from the **WeaponPickup** class and configure it with the following values:

- Static Mesh: **Content\Pickup\Weapon\SM_Weapon**
- Material: **Content\Weapon\Pistol\M_Pistol**
- Weapon Type: **Pistol**
- Ammo Amount: **25**
- Pickup Sound: **Content\Pickup\Weapon\WeaponPickup**

130. Save and close **BP_Pistol_Pickup**.

131. Create and open a new blueprint called **BP_MachineGun_Pickup** that is derived from the **WeaponPickup** class and configure it with the following values:

- Static Mesh: **Content\Pickup\Weapon\SM_Weapon**
- Material: **Content\Weapon\MachineGun\M_MachineGun**
- Weapon Type: **Machine Gun**
- Ammo Amount: **50**

- Pickup Sound: **Content\Pickup\Weapon\WeaponPickup**

132. Save and close **BP_MachineGun_Pickup**.

133. Create and open a new blueprint called **BP_Railgun_Pickup** that is derived from the **WeaponPickup** class and configure it with the following values:

- Static Mesh: **Content\Pickup\Weapon\SM_Weapon**
- Material: **Content\Weapon\Railgun\M_Railgun**
- Weapon Type: **Railgun**
- Ammo Amount: **5**
- Pickup Sound: **Content\Pickup\Weapon\WeaponPickup**

134. Save and close **BP_Railgun_Pickup**.

Now, we're going to import the remaining assets and use them in the player blueprint:

135. Go to **Content\Player\Sounds** and import **Land.wav** and **Pain.wav**.

136. Save **Land** and **Pain**.

137. Open **Content\Player\BP_Player** and set **Pain Sound** to use the **Pain** sound and **Land Sound** to use the **Land** sound.

138. Go to **Event Graph**.

139. Delete all of the nodes that create and add the **UI_HUD** instance to the viewport in the **Begin Play** event.

140. Save and close **BP_Player**.

Next, we're going to create the Scoreboard UMG widgets.

141. Go to **Content\UI** and create three new widget blueprints, called **UI_Scoreboard**, **UI_Scoreboard_Header**, and **UI_Scoreboard_Entry**.

142. Open **UI_Scoreboard_Entry**, which represents each player entry in the scoreboard.

143. Add a border called **B Background** to the root canvas panel as a variable that uses the **Stretch Both** anchor and has **Brush Color** set to **0.5** on all channels.

144. Go to the **Graph** section of the widget and create a new variable called **Player State** of the **FPSPlayerState** type that has **Instance Editable** and **Expose on Spawn** set to true.

145. In **Event Graph**, implement **Event Construct** in the following manner:

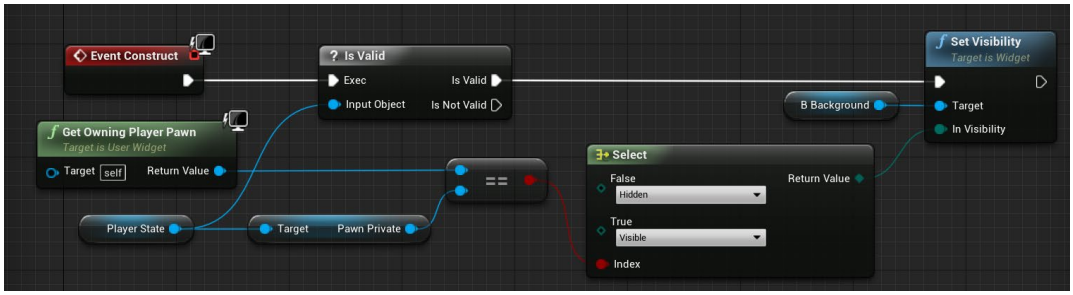

Figure 18.16: The Event Construct that sets the visibility of the background image

> **NOTE**
>
> You can find the preceding screenshot in full resolution for better viewing at the following link: https://packt.live/32REEvQ.

The preceding code first checks whether **Player State** is valid. If it is, then it sets the visibility of the **BBackground** border based on whether the pawn of **Player State** is the same as the owner of the UMG widget. In other words, the border will only be visible for the owning player's scoreboard entry, which makes it easier to find.

146. Go back to the designer and add a text block called **`tbPlayer`** to the root canvas panel with **`Size To Content`** set to **`true`**, **`Text`** set to **`Player Name`**, and bind it to use the following function:

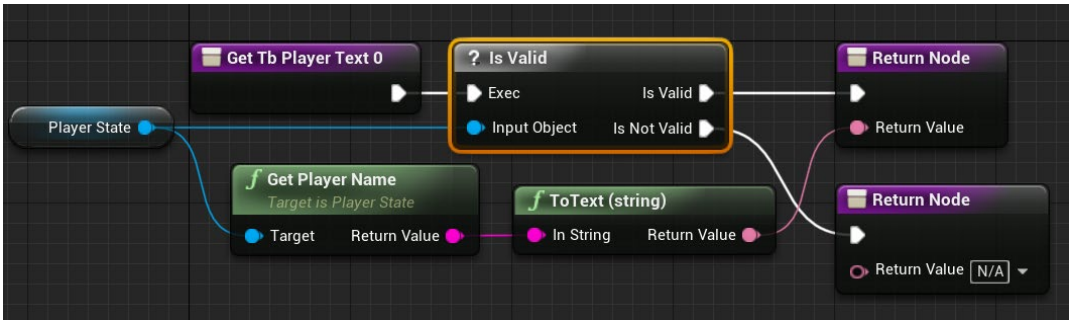

Figure 18.17: The bind function that displays the name of the player

> **NOTE**
>
> You can find the preceding screenshot in full resolution for better viewing at the following link: https://packt.live/3kBHlYv.

147. Add a text block called **`tbKills`** to the root canvas panel with **`Position X = 560`**, **`Alignment = 1.0 and 0.0`**, **`Size To Content`** set to **`true`**, **`Text`** set to **`999`**, and bind it to use the following function:

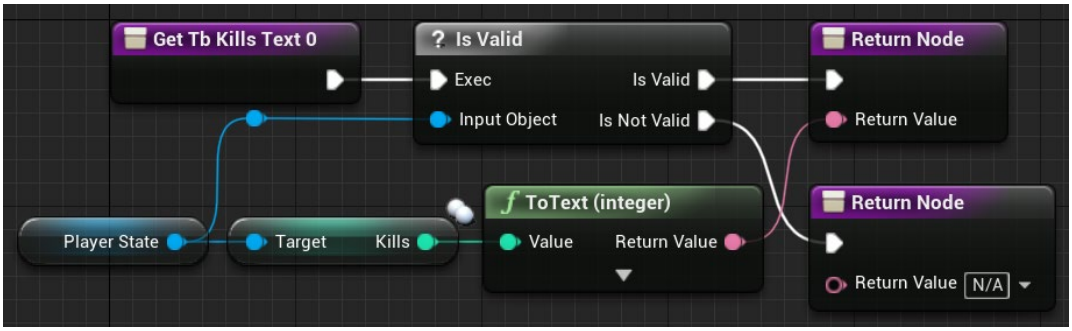

Figure 18.18: The bind function that displays the kills of the player

> **NOTE**
>
> You can find the preceding screenshot in full resolution for better viewing at the following link: https://packt.live/2Io7gWj.

148. Add a text block called **`tbDeaths`** to the root canvas panel with **`Position X = 715`**, **`Alignment = 1.0 and 0.0`**, **`Size To Content`** set to **`true`**, **`Text`** set to **`999`**, and bind it to use the following function:

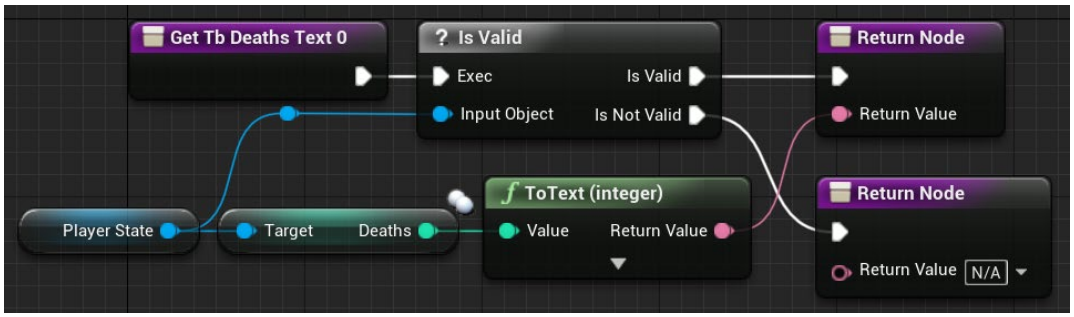

Figure 18.19: The bind function that displays the deaths of the player

> **NOTE**
>
> You can find the preceding screenshot in full resolution for better viewing at the following link: https://packt.live/3eLt6PE.

149. Add a text block called **`tbPing`** to the root canvas panel with **`Position X = 830`**, **`Alignment = 1.0 and 0.0`**, **`Size To Content`** set to **`true`**, **`Text`** set to **`999`**, and bind it to use the following function:

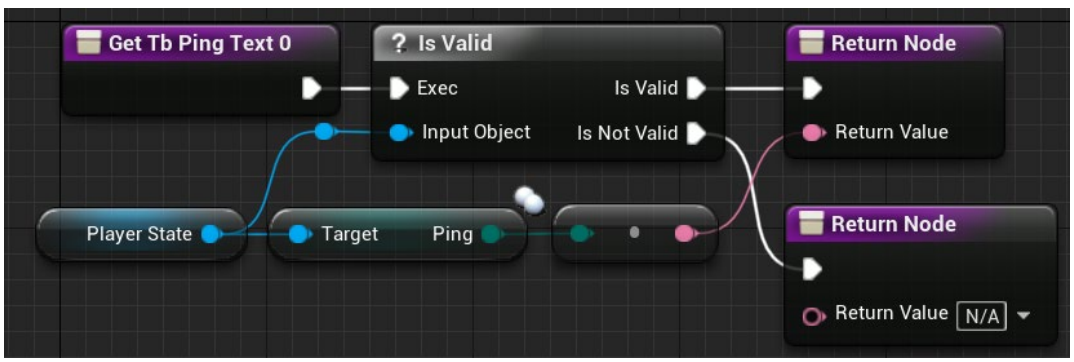

Figure 18.20: The bind function that displays the ping of the player

> **NOTE**
>
> You can find the preceding screenshot in full resolution for better viewing at the following link: https://packt.live/38wKHda.

150. Go back to the **Designer** section.

151. On the **Hierarchy** menu, select the four text blocks (by clicking on **tbPlayer**, and then hold the *Shift* key and click on **tbPing**) and press *Ctrl* + *C* to copy it to memory.

152. Save and close **UI_Scoreboard_Entry**.

153. Open **UI_Scoreboard_Header**, select the root canvas panel, and press *Ctrl* + *V* to paste the four text blocks, like so:

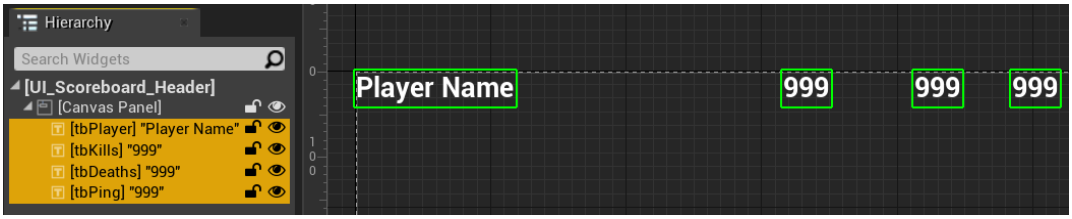

Figure 18.21: The UI_Scoreboard_Header widget hierarchy

154. While keeping the four text blocks still selected, set **Color and Opacity** to **black**.

155. Select **tbKills** and set its **Text** field to **Kills**.

156. Select **tbDeaths** and set its **Text** field to **Deaths**.

157. Select **tbPing** and set its **Text** field to **Ping**.

158. You should end up with the following:

Figure 18.22: The look of text blocks after changing their default text

159. Save and close **UI_Scoreboard_Header**.

160. Open **UI_Scoreboard** and go to the **Graph** section. Create a new variable called **Game State** with the **FPSGameState** type.

161. On **`Event Graph`**, implement **`Event Construct`** in the following manner:

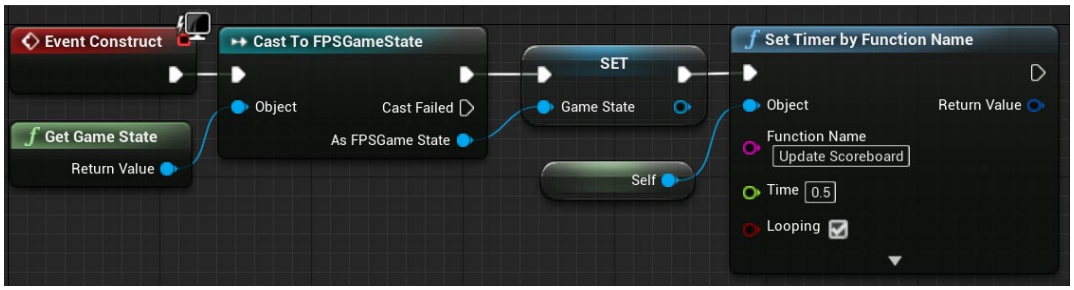

Figure 18.23: The Event Construct that sets a timer to update the scoreboard every 0.5 seconds

> **NOTE**
>
> You can find the preceding screenshot in full resolution for better viewing at the following link: https://packt.live/3kxwCOY.

The preceding code saves a reference for the game state and schedules a looping timer that calls the **`Update Scoreboard`** event, that will create in the next few lines, every 0.5 seconds.

162. Go back to the Designer section and add a new text block called **`tbKillLimit`** to the root canvas panel that uses the **`Top Center`** anchor, **`Position Y = 250`**, **`Alignment = 0.5 and 0.5`**, **`Size To Content`** set to **`true`**, **`Font Size = 36`**, **`Text`** set to **`Kill Limit: 999`**, and bind it to use the following function:

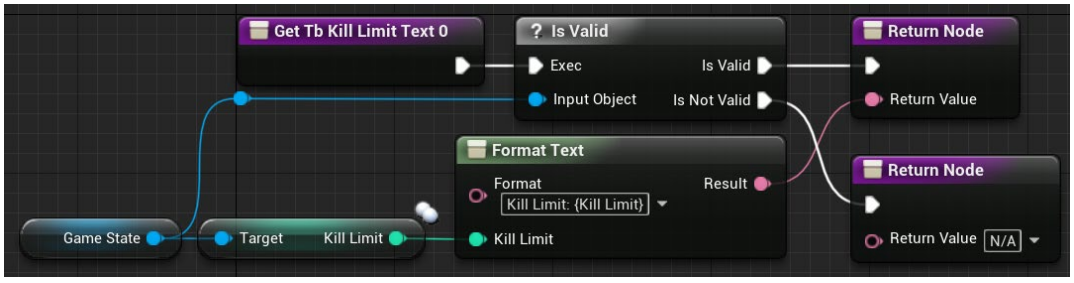

Figure 18.24: The bind function that displays the kill limit of the game

> **NOTE**
>
> You can find the preceding screenshot in full resolution for better viewing at the following link: https://packt.live/3pz9u6l.

163. Add a vertical box called **`vbScoreboard`** to the root canvas panel that uses **`Is Variable`** set to **`true`** the **`Top Center`** anchor, **`Position Y = 300`**, **`Alignment = 0.5 and 0.5`**, and **`Size To Content`** set to **`true`**.

164. Create a new function called **`Add Scoreboard Header`** that adds an instance of **`UI_Scoreboard_Header`** to **`vbScoreboard`**, as shown in the following screenshot:

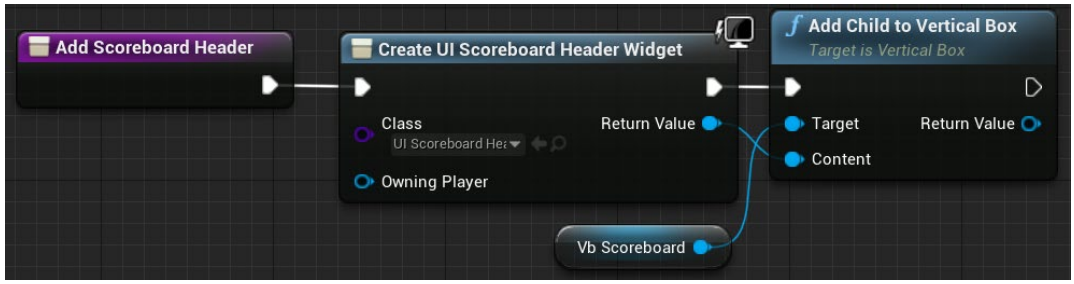

Figure 18.25: The Add Scoreboard Header function graph

> **NOTE**
>
> You can find the preceding screenshot in full resolution for better viewing at the following link: https://packt.live/3ngOLCv.

165. Create a new function called **`Add Scoreboard Entries`**. This goes through all of the player states ordered by kills and adds an instance of **`UI_Scoreboard_Entry`** to **`VbScoreboard`**, as demonstrated in the following screenshot:

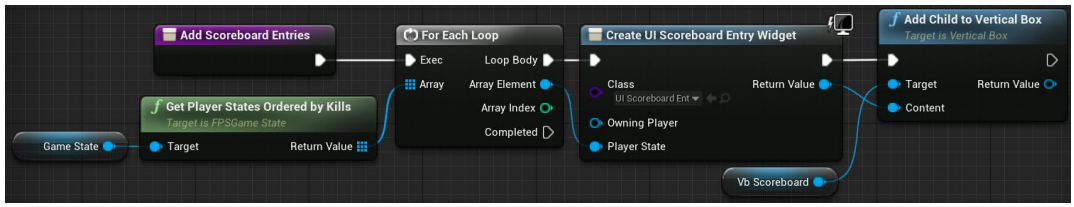

Figure 18.26: The Add Scoreboard Entries function graph

> **NOTE**
>
> You can find the preceding screenshot in full resolution for better viewing at the following link: https://packt.live/2Ugwb0U.

166. Implement the **`Update Scoreboard`** event, by right clicking on an empty space and typing on the filter **`Custom Event`**. This event clears the widgets in **`VbScoreboard`** and calls **`Add Scoreboard Header`** and **`Add Scoreboard Entries`**, as shown in the following screenshot:

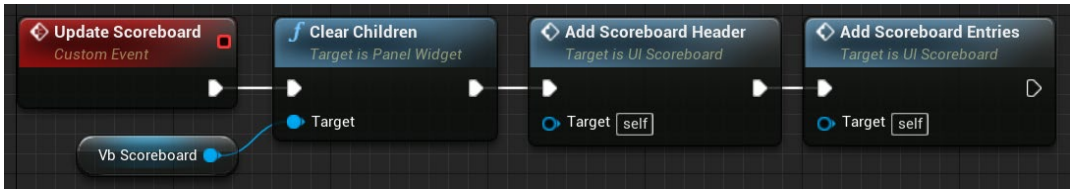

Figure 18.27: The Update Scoreboard event graph

> **NOTE**
>
> You can find the preceding screenshot in full resolution for better viewing at the following link: https://packt.live/3ly0M69.

167. Save and close **`UI_Scoreboard`**.

Next, we're going to edit **`UI_HUD`** to add the kill message.

1. Open **`Content\UI\UI_HUD`**.
2. Add a new text block called **`tbKilled`** to the **`root canvas`** panel that uses the **`Center Center`** anchor, **`Is Variable`** set to **`true`**, **`Text`** set to **`You killed Player`**, **`Position Y = -100`**, **`Alignment = 0.5 and 0.5`**, **`Size To Content`** set to **`true`** and **`Visibility`** set to **`Hidden`**.

3. Go to the **`Graph`** section and create a new event called **`NotifyKill`** that has a **`String`** parameter called **`Name`** and the following implementation:

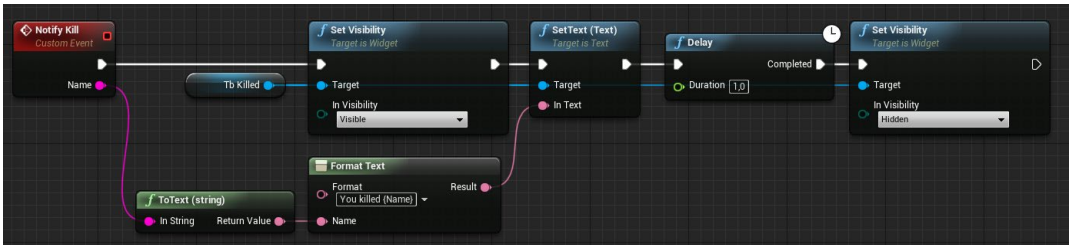

Figure 18.28: Shows the name of killer player and hides it after 1 second

> **NOTE**
>
> You can find the preceding screenshot in full resolution for better viewing at the following link: https://packt.live/35jNvrV.

The preceding code sets the **`tbKilled`** text block to display the name of the player that was killed and makes it fade out by playing the animation we created.

4. Save and close **`UI_HUD`**.

   Now, we're going to create the player menu UMG widget that combines **`UI_HUD`** and **`UI_Scoreboard`**.

5. Go to **`Content\UI`** and create a new blueprint called **`UI_PlayerMenu`**, which is derived from the **`PlayerMenu`** class.

6. Add a widget switcher called **`wsMenus`** as the root panel.

7. Using the palette, add a **`UI_HUD`** instance called **`HUD`** to **`wsMenus`** and make sure it's set to a variable.

8. Using the palette, add a **`UI_Scoreboard`** instance called **`Scoreboard`** to **`wsMenus`** and make sure it's set to a variable.

9. You should end up with the following hierarchy:

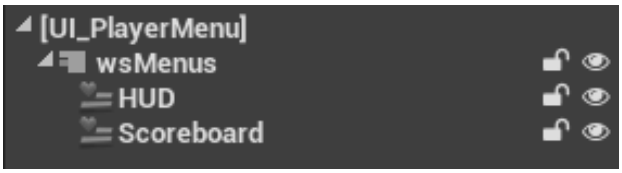

Figure 18.29: The UI_Scoreboard widget hierarchy

10. Go to the **`Graph`** section and override the **`Notify Kill`** event and implement it in the following way:

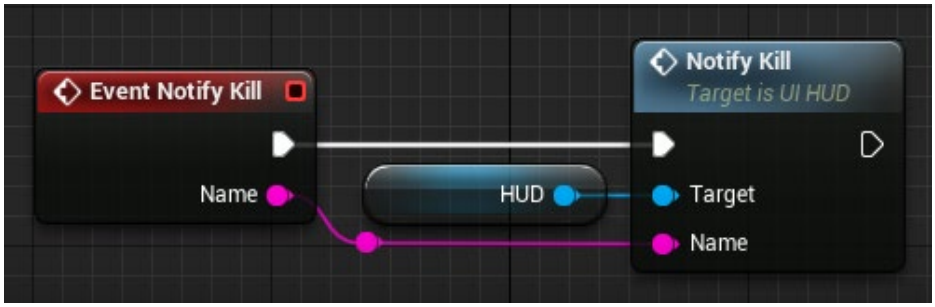

Figure 18.30: Calling the Notify Kill function in the HUD menu

> **NOTE**
>
> You can find the preceding screenshot in full resolution for better viewing at the following link: https://packt.live/2JTaIJf.

11. Override the **`Set Scoreboard Visibility`** event and implement it in the following way:

Figure 18.31: Setting scoreboard visibility by changing the switcher's active widget index

> **NOTE**
>
> You can find the preceding screenshot in full resolution for better viewing at the following link: https://packt.live/3pGKf1V.

12. Override the **`Toggle Scoreboard`** event and implement it in the following way:

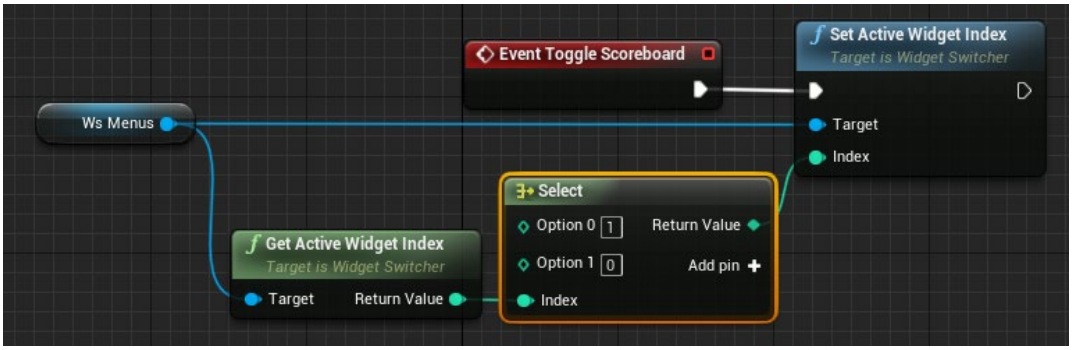

Figure 18.32: Toggling scoreboard visibility by changing the switcher's active widget index

> **NOTE**
>
> You can find the preceding screenshot in full resolution for better viewing at the following link: https://packt.live/2Ip59lv.

13. Save and close **`UI_PlayerMenu`**.

    Now, we're going to create a blueprint for **`FPSPlayerController`**.

14. Go to the **`Content`** folder and create a new blueprint called **`BP_PlayerController`**, which is derived from **`FPSPlayerController`**.

15. Open **`BP_PlayerController`** and set the **`Player Menu`** variable to **`UI_PlayerMenu`**, like so:

Figure 18.33: Setting the Player Menu Class

16. Save and close **BP_PlayerController**.

    Next, we need to edit the game mode to use the new player controller blueprint.

17. Go to the **Content** folder, open **BP_GameMode**, and set **Player Controller Class** to **BP_PlayerController**, like so:

Figure 18.34: Setting the player controller class

18. Save and close **BP_GameMode**.

    Next, we're going to add the action binding to toggle the scoreboard.

19. Go to **Project Settings** and select **Input** from the left panel, which is in the **Engine** category.

20. Create an action mapping called **Scoreboard**, using the **TAB** key.

21. Close **Project Settings**.

    Next, we need to change some things in the **DM-Test** map.

22. Go to **Window -> World Settings** and set **Kill Z** to **-500**, like so:

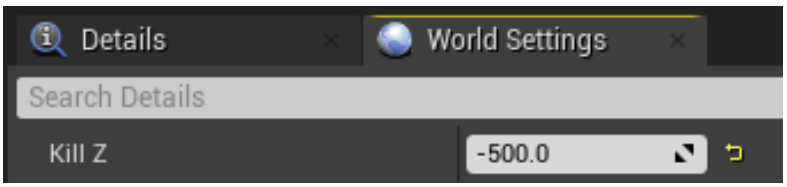

Figure 18.35: Setting the Kill Z value in World Settings

23. Place in the world at least two new player starts in different locations.

24. Place in the world an instance for every different pickup available.

25. You could end up with something like the following:

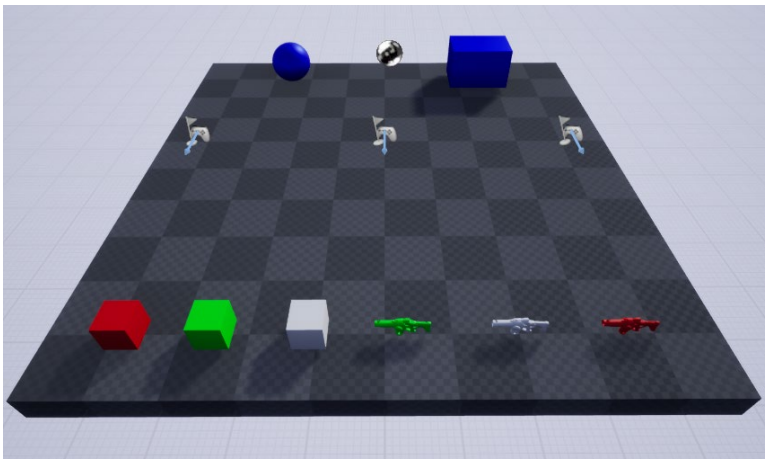

Figure 18.36: An example of a map configuration

26. Save **`DM-Test`**.

    Now, let's test the project.

27. Click on the down arrow next to the **`Play`** button and pick the last option, **`Advanced Settings`**.

28. In the **`Game Viewport Settings`** section, change **`New Viewport Resolution`** to **`800x600`** and close the **`Editor Preferences`** tab.

29. Click again on the down arrow next to the **`Play`** button, set the number of clients to **`2`**, and click on **`New Editor Window (PIE)`**:

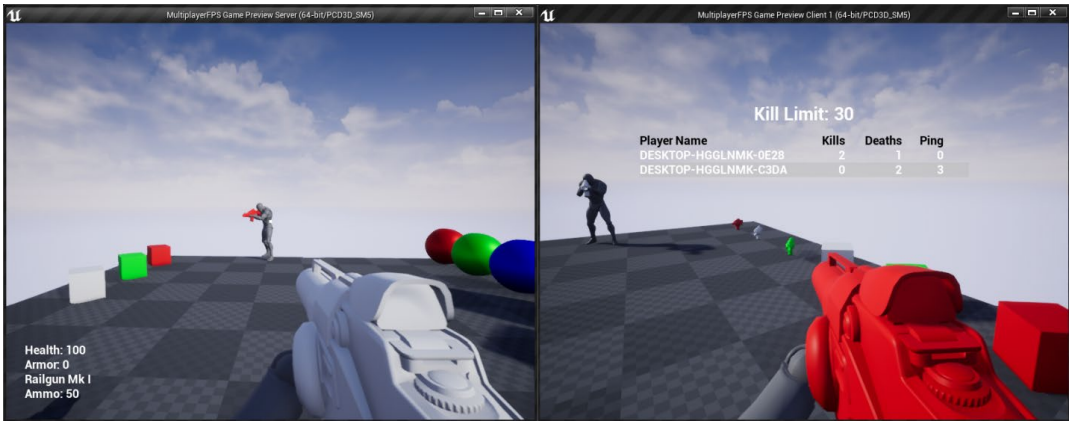

Figure 18.37: The listen Server and Client 1 playing against each other

The end result is 2 characters that can carry 3 weapons (each with 50 ammo), which you can fire and damage the other players with. You can also select weapons by using the *1*, *2*, and *3* keys or by moving the mouse wheel up and down to get the previous and next weapons. A character will take damage only if you hit its skeletal mesh. When a character dies, it will respawn in a random player start. You will also be able to toggle the scoreboard by pressing *TAB*, which will display the name, kill count, death count, and ping for each player. If a character falls from the level, it will count as a death but not as a kill from another player. The character can also overlap with the different pickups in the level to get ammo, armor, health, and weapons. Once the kill limit has been reached, the characters will be destroyed, the scoreboard will show the winner, and the level will be reloaded with server travel after 5 seconds.