

Game Development Projects with Unreal Engine

Learn to build your first game and bring your ideas to life using UE4 and C++.



Game Development Projects with Unreal Engine

Learn to build your first games and bring your ideas to life using UE4 and C++

Hammad Fozi | Gonçalo Marques | David Pereira | Devin Sherry



Contents

1. [Game Development Projects with Unreal Engine](#)

2. [Preface](#)

1. [About the Book](#)

1. [About the Authors](#)

2. [Audience](#)

3. [About the Chapters](#)

4. [Conventions](#)

5. [Before You Begin](#)

6. [Installing Visual Studio](#)

7. [Epic Games Launcher](#)

8. [Code Bundle](#)

9. [Get in touch](#)

10. [Please leave a review](#)

3. [1. Unreal Engine Introduction](#)

1. [Introduction](#)

1. [Exercise 1.01: Creating an Unreal Engine 4 Project](#)

2. [Getting to Know Unreal](#)

3. [Editor Windows](#)

4. [Viewport Navigation](#)

5. [Manipulating Actors](#)

1. [Exercise 1.02: Adding and Removing Actors](#)

6. [Blueprint Actors](#)

1. [Exercise 1.03: Creating Blueprint Actors](#)

7. [The Blueprint Editor](#)

8. [Event Graph](#)

1. [Exercise 1.04: Creating Blueprint Variables](#)

2. [Exercise 1.05: Creating Blueprint Functions](#)

9. The Float Multiplication Node
 10. BeginPlay and Tick
 1. Exercise 1.06: Offsetting the TestActor Class on the Z Axis
 11. ThirdPersonCharacter Blueprint Class
 12. Meshes and Materials
 1. Meshes
 2. Materials
 13. Manipulating Materials in UE4
 1. Activity 1.01: Propelling TestActor on the Z Axis Indefinitely
 14. Summary
4. 2. Working with Unreal Engine
 1. Introduction
 2. Creating and Setting Up a Blank C++ Project
 1. Exercise 2.01: Creating an Empty C++ Project
 3. Content Folder Structure in Unreal Engine
 4. Working with the Visual Studio Solution
 1. Solution Analysis
 1. The Engine Project
 2. Game Project
 2. Debugging Code in Visual Studio
 3. Exercise 2.02: Debugging the Third Person Template Code
 5. Importing the Required Assets
 1. Exercise 2.03: Importing a Character FBX File

6. The Unreal Game Mode Class

1. Game Mode Default Classes
2. Gameplay Events
3. Networking
4. GameModeBase versus GameMode

7. Levels

1. The Unreal Pawn Class
 1. The Default Pawn
 2. The Spectator Pawn
2. The Unreal Player Controller Class
3. Exercise 2.04: Setting Up the Game Mode, Player Controller, and Pawn

8. Animations

1. Animation Blueprints
2. Event Graph
3. The Anim Graph
4. State Machines
5. Transition Rules
6. Blend Spaces
7. Exercise 2.05: Creating a Mannequin Animation
8. Activity 2.01: Linking Animations to a Character

9. Summary

5. 3. Character Class Components and Blueprint Setup

1. Introduction
2. The Unreal Character Class
 1. Extending the Character Class
 2. Exercise 3.01: Creating and Setting Up a Third-Person Character C++ Class
3. Extending the C++ Class with Blueprints

1. Exercise 3.02: Extending C++ with Blueprints
2. Activity 3.01: Extending the C++ Character Class with Blueprint in the Animation Project

4. Summary

6. 4. Player Input
 1. Introduction
 2. Input Actions and Axes
 1. Exercise 4.01: Creating the Jump Action and Movement Axes

 3. Processing Player Input
 4. DefaultInput.ini
 1. Exercise 4.02: Listening to Movement Actions and Axes

 5. Turning the camera around the character
 1. Mobile platforms
 2. Exercise 4.03: Previewing on Mobile
 3. Exercise 4.04: Adding Touchscreen Input
 4. Activity 4.01: Adding Walking Logic to Our Character

 6. Summary

7. 5. Line Traces
 1. Introduction
 2. Collision
 3. Project Setup
 1. Exercise 5.01: Converting DodgeballCharacter to a Top-Down Perspective

 4. Line Traces
 5. Creating the EnemyCharacter C++ class

1. [Exercise 5.02: Creating the CanSeeActor Function, Which Executes Line Traces](#)
6. [Visualizing the Line Trace](#)
 1. [Exercise 5.03: Creating the LookAtActor Function](#)
7. [Creating the EnemyCharacter Blueprint Class](#)
8. [Sweep Traces](#)
 1. [Exercise 5.04: Executing a Sweep Trace](#)
 2. [Changing the Visibility Trace Response](#)
 3. [Multi Line Traces](#)
 4. [The Camera Trace Channel](#)
 5. [Exercise 5.05: Creating a Custom EnemySight Trace Channel](#)
 6. [Activity 5.01: Creating the SightSource Property](#)
9. [Summary](#)
8. [6. Collision Objects](#)
 1. [Introduction](#)
 2. [Object Collision in UE4](#)
 3. [Collision Components](#)
 4. [Collision Events](#)
 5. [Collision Channels](#)
 6. [Collision Properties](#)
 1. [Exercise 6.01: Creating the Dodgeball Class](#)
7. [Physical Materials](#)
 1. [Exercise 6.02: Adding a Projectile Movement Component to DodgeballProjectile](#)
8. [Timers](#)
9. [Spawning Actors](#)
 1. [Exercise 6.03: Adding Projectile-Throwing Logic to the EnemyCharacter](#)

10. Walls

1. Exercise 6.04: Creating Wall Classes

11. Victory Box

1. Exercise 6.05: Creating the VictoryBox class
2. Exercise 6.06: Adding the ProjectileMovementComponent Getter Function in DodgeballProjectile
3. Activity 6.01: Replacing the SpawnActor Function with SpawnActorDeferred in EnemyCharacter

12. Summary

9. 7. UE4 Utilities

1. Introduction
2. Blueprint Function Libraries
 1. Exercise 7.01: Moving the CanSeeActor Function to the Blueprint Function Library
3. Actor Components
 1. Exercise 7.02: Creating the HealthComponent Actor Component
 2. Exercise 7.03: Integrating the HealthComponent Actor Component
4. Interfaces
5. Blueprint Native Events
 1. Exercise 7.04: Creating the HealthInterface Class
 2. Activity 7.01: Moving the LookAtActor Logic to an Actor Component

6. Summary

10. 8. User Interfaces

1. [Introduction](#)
 2. [Game UI](#)
 3. [UMG Basics](#)
 1. [Exercise 8.01: Creating a Widget Blueprint](#)
 4. [Anchors](#)
 1. [Exercise 8.02: Editing UMG Anchors](#)
 2. [Exercise 8.03: Creating the RestartWidget C++ Class](#)
 3. [Exercise 8.04: Creating the Logic for Adding the RestartWidget to the Screen](#)
 4. [Exercise 8.05: Setting up the DodgeballPlayerController Blueprint Class](#)
 5. [Progress Bars](#)
 1. [Exercise 8.06: Creating the Health Bar C++ Logic](#)
 2. [Activity 8.01: Improving the RestartWidget](#)
 6. [Summary](#)
11. [9. Audio-Visual Elements](#)
1. [Introduction](#)
 2. [Audio in UE4](#)
 1. [Exercise 9.01: Importing an Audio File](#)
 2. [Exercise 9.02: Playing a Sound When the Dodgeball Bounces off a Surface](#)
 3. [Sound Attenuation](#)
 1. [Exercise 9.03: Turning the Bounce Sound into a 3D Sound](#)
 2. [Exercise 9.04: Adding Background Music to Our Game](#)
 4. [Particle Systems](#)
 1. [Exercise 9.05: Spawning a Particle System When the Dodgeball Hits the Player](#)

2. Activity 9.01: Playing a Sound When the Dodgeball Hits the Player
5. Level Design
 1. Exercise 9.06: Building a Level Blockout
 6. Extra Features
 7. Summary
12. 10. Creating a SuperSideScroller Game
 1. Introduction
 2. Project Breakdown
 3. The Player Character
 1. Exercise 10.01: Creating the Side-Scroller Project and Using the Character Movement Component
 2. Side Scroller versus 2D Side Scroller
 3. Activity 10.01: Making Our Character Jump Higher
 4. Features of Our Side-Scroller Game
 1. Enemy Character
 2. Power-Up
 3. Collectible
 4. HUD (Heads-Up Display)
 5. Steps in Animation
 6. Character Animation Pipeline
 1. The Concept Stage
 2. The 3D Modeling Stage
 3. The Rigging Stage
 4. Animation
 5. Asset Export and Import
 6. Exercise 10.02: Exploring the Persona Editor and Manipulating the Default Mannequin Skeleton Weights
 7. Activity 10.02: Skeletal Bone Manipulation and Animations

7. Animations in Unreal Engine 4
 1. Skeletons
 2. Skeletal Meshes
 3. Animation Sequences
 4. Exercise 10.03: Importing and Setting Up the Character and Animation
 5. Activity 10.03: Importing More Custom Animations to Preview the Character Running

8. Summary

13. 11. Blend Spaces 1D, Key Bindings, and State Machines

1. Introduction
2. Blend Spaces
 1. 1D Blend Space vs Normal Blend Space
 2. Exercise 11.01: Creating the CharacterMovement 1D Blend Space
 3. Activity 11.01: Adding the Walking and Running Animations to the Blend Space
3. Main Character Animation Blueprint
 1. Animation Blueprints
 2. Exercise 11.02: Adding the Blend Space to the Character Animation Blueprint
4. Velocity Vectors
 1. Exercise 11.03: Adding the Blend Space to the Character Animation Blueprint
 2. Activity 11.02: Previewing the Running Animation In-Game
5. Input Bindings
 1. Exercise 11.04: Adding Input for Sprinting and Throwing
 2. Exercise 11.05: Reparenting the Character Blueprint
 3. Exercise 11.06: Coding the Character Sprint

Functionality

4. Activity 11.03: Implementing the Throwing Input

6. Animation State Machines

1. Exercise 11.07: Player Character Movement and Jump State Machine
2. Transition Rules
3. Exercise 11.08: Adding States and Transition Rules to the State Machine
4. Exercise 11.09: Time Remaining Ratio Function
5. Activity 11.04: Finishing the Movement and Jumping State Machine

7. Summary

14. 12. Animation Blending and Montages

1. Introduction
2. Animation Blending, Anim Slots, and Animation Montages
 1. Exercise 12.01: Setting Up the Animation Montage
3. Animation Montages
 1. Exercise 12.02: Adding the Throw Animation to the Montage
4. Anim Slot Manager
 1. Exercise 12.03: Adding a New Anim Slot
5. Save Cached Pose
 1. Exercise 12.04: Save Cached Pose of the Movement State Machine
6. Layered blend per bone

1. [Exercise 12.05: Blending Animation with the Upper Body Anim Slot](#)
7. [The Throw Animation](#)
 1. [Exercise 12.06: Previewing the Throw Animation](#)
8. [The Super Side Scroller Game Enemy](#)
 1. [Exercise 12.07: Creating the Enemy Base C++ Class](#)
 2. [Exercise 12.08: Creating and Applying the Enemy Animation Blueprint](#)
9. [Materials and Material Instances](#)
 1. [Exercise 12.09: Creating and Applying the Enemy Material Instance](#)
 2. [Activity 12.01: Updating Blend Weights](#)
10. [Summary](#)
15. [13. Enemy Artificial Intelligence](#)
 1. [Introduction](#)
 2. [Enemy AI](#)
 3. [AI Controller](#)
 1. [Auto Possess AI](#)
 2. [Exercise 13.01: Implementing AI Controllers](#)
 4. [Navigation Mesh](#)
 1. [Exercise 13.02: Implementing a Nav Mesh Volume for the AI Enemy](#)
 5. [Recasting the Nav Mesh](#)
 1. [Exercise 13.03: Recasting Nav Mesh Volume Parameters](#)
 2. [Activity 13.01: Creating a New Level](#)

6. Behavior Trees and Blackboards

1. Composites
2. Tasks
3. Decorators
4. Services
5. Exercise 13.04: Creating the AI Behavior Tree and Blackboard
6. Exercise 13.05: Creating a New Behavior Tree Task
7. Exercise 13.06: Creating the Behavior Tree Logic
8. Activity 13.02: AI Moving to the Player Location
9. Exercise 13.07: Creating the Enemy Patrol Locations

7. Vector Transformation

1. Exercise 13.08: Selecting a Random Point in an Array
2. Exercise 13.09: Referencing the Patrol Point Actor
3. Exercise 13.10: Updating BTTask_FindLocation

8. Player Projectile

1. Exercise 13.11: Creating the Player Projectile
2. Exercise 13.12: Initializing Player Projectile Settings
3. Activity 13.03: Creating the Player Projectile Blueprint

9. Summary

16. 14. Spawning the Player Projectile

1. Introduction
2. Anim Notifies and Anim Notify States
 1. Exercise 14.01: Creating a UAnim Notify Class
 2. Exercise 14.02: Adding the Notify to the

Throw Montage

3. Playing Animation Montages

1. Playing Animation Montages in Blueprints
2. Playing Animation Montages in C++
3. Exercise 14.03: Playing the Throw Animation in C++

4. Game World and Spawning Objects

1. Exercise 14.04: Creating the Projectile Spawn Socket
2. Exercise 14.05: Preparing the SpawnProjectile() Function
3. Exercise 14.06: Updating the Anim_ProjectileNotify Class

5. Destroying Actors

1. Exercise 14.07: Creating the DestroyEnemy() Function
2. Exercise 14.08: Destroying Projectiles
3. Activity 14.01: Projectile Destroying Enemies

6. Visual and Audio Effects

1. Exercise 14.09: Adding Effects When the Enemy Is Destroyed
2. Exercise 14.10: Adding Effects to the Player Projectile
3. Exercise 14.11: Adding VFX and SFX Notifies
4. Activity 14.02: Adding Effects for When the Projectile Is Destroyed

7. Summary

17. 15. Collectibles, Power-Ups, and Pickups

1. Introduction
2. URotatingMovementComponent

1. Exercise 15.01: Creating the PickableActor_Base Class and Adding

URotatingMovementComponent

2. Activity 15.01: Player Overlap Detection and Spawning Effects in PickableActor_Base
3. Exercise 15.02: Creating the PickableActor_Collectable Class
4. Activity 15.02: Finalizing the PickableActor_Collectable Actor

3. Logging Variables Using UE_LOG

1. Exercise 15.03: Tracking the Number of Coins for the Player

4. UMG

5. Text Widget

1. Anchors
2. Text Formatting
3. Exercise 15.04: Creating the Coin Counter UI HUD Element

6. Adding and Creating UMG User Widgets

1. Exercise 15.05: Adding the Coin Counter UI to the Player Screen

7. Timers

1. Exercise 15.06: Adding the Potion Power-Up Behavior to the Player
2. Activity 15.03: Creating the Potion Power-Up Actor
3. Exercise 15.07: Creating the Brick Class
4. Exercise 15.08: Adding the Brick Class C++ Logic

8. Summary

18. 16. Multiplayer Basics

1. Introduction
2. Multiplayer Basics
3. Servers

1. [Dedicated Server](#)
 2. [The Listen Server](#)
4. [Clients](#)
 1. [Exercise 16.01: Testing the Third Person Template in Multiplayer](#)
5. [The Packaged Version](#)
6. [Connections and Ownership](#)
7. [Roles](#)
 1. [Exercise 16.02: Implementing Ownership and Roles](#)
 2. [The Server Window](#)
 3. [Server Character](#)
 4. [Client 1 Character](#)
 5. [The OwnershipTest Actor](#)
 6. [The Client 1 Window](#)
8. [Variable Replication](#)
 1. [Replicated Variables](#)
 2. [Exercise 16.03: Replicating Variables Using Replicated, RepNotify, DOREPLIFETIME, and DOREPLIFETIME_CONDITION](#)
 3. [The Server Window](#)
 4. [The Client 1 Window](#)
9. [2D Blend Spaces](#)
 1. [Exercise 16.04: Creating a Movement 2D Blend Space](#)
10. [Transform \(Modify\) Bone](#)
 1. [Exercise 16.05: Creating a Character That Looks up and down](#)
 2. [Activity 16.01: Creating a Character for the Multiplayer FPS Project](#)
11. [Summary](#)

19. 17. Remote Procedure Calls

1. Introduction
2. Remote Procedure Calls
 1. Server RPC
 1. Declaration
 2. Execution
 3. Valid Connection
 2. Multicast RPC
 1. Declaration
 2. Execution
 3. Client RPC
 1. Declaration
 2. Execution
 4. Important Considerations When Using RPCs
 5. Exercise 17.01: Using Remote Procedure Calls
3. Enumerations
 1. TEnumAsByte
 2. UMETA
 1. DisplayName
 2. Hidden
 3. BlueprintType
 1. MAX
 4. Exercise 17.02: Using C++ Enumerations in the Unreal Engine 4 Editor
4. Bi-Directional Circular Array Indexing
 1. Exercise 17.03: Using Bi-Directional Circular Array Indexing to Cycle between an

Enumeration

2. Activity 17.01: Adding Weapons and Ammo to the Multiplayer FPS Game

5. Summary

20. 18. Gameplay Framework Classes in Multiplayer

1. Introduction

2. Gameplay Framework Classes in Multiplayer

1. Exercise 18.01: Displaying the Gameplay Framework Instance Values
2. The Server Window
3. Server Character
4. Client 1 Character
5. The Client 1 Window
6. Client 1 Character
7. Server Character

3. Game Mode, Player State, and Game State

1. Game Mode

1. Constructor

2. Player State
3. Game State
4. Useful Built-in Functionality
5. Exercise 18.02: Making a Simple Multiplayer Pickup Game
6. Activity 18.01: Adding Death, Respawn, Scoreboard, Kill Limit, and Pickups to the Multiplayer FPS Game

4. Summary

Landmarks

1. Cover
2. Table of Contents

Game Development Projects with Unreal Engine

Copyright © 2020 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Authors: Hammad Fozi, Gonçalo Marques, David Pereira, and Devin Sherry

Reviewers: Lennard Fonteijn, Pranav Paharia, Sargey Rose, and Rahul Shekhawat

Managing Editor: Ashish James

Acquisitions Editors: Kunal Sawant, Archie Vankar, and Karan Wadekar

Production Editor: Shantanu Zagade

Editorial Board: Megan Carlisle, Samuel Christa, Mahesh Dhyani, Heather Gopsill, Manasa Kumar, Alex Mazonowicz, Monesh Mirpuri, Bridget Neale, Dominic Pereira, Shiny Poojary, Abhishek Rane, Brendan Rodrigues, Erol Staveley, Ankita Thakur, Nitesh Thakur, and Jonathan Wray

First published: November 2020

Production reference: 1251120

ISBN: 978-1-80020-922-0

Published by Packt Publishing Ltd.

Livery Place, 35 Livery Street

Birmingham B3 2PB, UK

Table of Contents

Preface

1. Unreal Engine Introduction

INTRODUCTION

EXERCISE 1.01: CREATING AN UNREAL ENGINE 4 PROJECT

GETTING TO KNOW UNREAL

EDITOR WINDOWS

VIEWPORT NAVIGATION

MANIPULATING ACTORS

EXERCISE 1.02: ADDING AND REMOVING ACTORS

BLUEPRINT ACTORS

EXERCISE 1.03: CREATING

BLUEPRINT ACTORS

THE BLUEPRINT EDITOR

EVENT GRAPH

EXERCISE 1.04: CREATING BLUEPRINT VARIABLES

EXERCISE 1.05: CREATING BLUEPRINT FUNCTIONS

THE FLOAT MULTIPLICATION NODE

BEGINPLAY AND TICK

EXERCISE 1.06: OFFSETTING THE TESTACTOR CLASS ON THE

ZAXIS

THIRDPERSONCHARACTER
BLUEPRINT CLASS

MESHES AND MATERIALS

MESHES

MATERIALS

MANIPULATING MATERIALS
IN UE4

ACTIVITY 1.01: PROPELLING
TESTACTOR ON THE Z AXIS
INDEFINITELY

SUMMARY

2. Working with Unreal Engine

INTRODUCTION

CREATING AND SETTING UP A BLANK C++ PROJECT

EXERCISE 2.01: CREATING AN EMPTY C++ PROJECT

CONTENT FOLDER STRUCTURE IN UNREAL ENGINE

WORKING WITH THE VISUAL STUDIO SOLUTION

SOLUTION ANALYSIS

THE ENGINE PROJECT

GAME PROJECT

DEBUGGING CODE IN VISUAL STUDIO

EXERCISE 2.02: DEBUGGING THE THIRD PERSON TEMPLATE CODE

IMPORTING THE REQUIRED ASSETS

EXERCISE 2.03: IMPORTING A CHARACTER FBX FILE

THE UNREAL GAME MODE CLASS

GAME MODE DEFAULT CLASSES

GAMEPLAY EVENTS

NETWORKING

GAMEMODEBASE VERSUS GAMEMODE

LEVELS

THE UNREAL PAWN CLASS

THE DEFAULT PAWN

THE SPECTATOR PAWN

THE UNREAL PLAYER CONTROLLER CLASS

EXERCISE 2.04: SETTING UP THE GAME MODE, PLAYER CONTROLLER, AND PAWN

ANIMATIONS

ANIMATION BLUEPRINTS

EVENT GRAPH

THE ANIM GRAPH

STATE MACHINES

TRANSITION RULES

BLEND SPACES

EXERCISE 2.05: CREATING A MANNEQUIN ANIMATION

ACTIVITY 2.01: LINKING ANIMATIONS TO A CHARACTER

SUMMARY

3. Character Class Components and Blueprint Setup

INTRODUCTION

THE UNREAL CHARACTER CLASS

EXTENDING THE CHARACTER CLASS

EXERCISE 3.01: CREATING

AND SETTING UP A THIRD- PERSON CHARACTER C++ CLASS

EXTENDING THE C++ CLASS WITH BLUEPRINTS

EXERCISE 3.02: EXTENDING C++ WITH BLUEPRINTS

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

SUMMARY

4. Player Input

INTRODUCTION

INPUT ACTIONS AND AXES

**EXERCISE 4.01: CREATING
THE JUMP ACTION AND
MOVEMENT AXES**

**PROCESSING PLAYER
INPUT**

DEFAULTINPUT.INI

**EXERCISE 4.02: LISTENING
TO MOVEMENT ACTIONS
AND AXES**

**TURNING THE CAMERA
AROUND THE CHARACTER**

MOBILE PLATFORMS

EXERCISE 4.03: PREVIEWING ON MOBILE

EXERCISE 4.04: ADDING TOUCHSCREEN INPUT

ACTIVITY 4.01: ADDING WALKING LOGIC TO OUR CHARACTER

SUMMARY

5. Line Traces

INTRODUCTION

COLLISION

PROJECT SETUP

EXERCISE 5.01:
CONVERTING
DODGEBALLCHARACTER
TO A TOP-DOWN
PERSPECTIVE

LINE TRACES

CREATING THE
ENEMYCHARACTER C++
CLASS

EXERCISE 5.02: CREATING
THE CANSEEACTOR
FUNCTION, WHICH
EXECUTES LINE TRACES

VISUALIZING THE LINE
TRACE

EXERCISE 5.03: CREATING THE LOOKATACTOR FUNCTION

CREATING THE ENEMYCHARACTER BLUEPRINT CLASS

SWEET TRACES

EXERCISE 5.04: EXECUTING A SWEEP TRACE

CHANGING THE VISIBILITY TRACE RESPONSE

MULTI LINE TRACES

THE CAMERA TRACE CHANNEL

EXERCISE 5.05: CREATING A CUSTOM ENEMYSIGHT TRACE CHANNEL

ACTIVITY 5.01: CREATING THE SIGHTSOURCE PROPERTY

SUMMARY

6. Collision Objects

INTRODUCTION

OBJECT COLLISION IN UE4

COLLISION COMPONENTS

COLLISION EVENTS

COLLISION CHANNELS

COLLISION PROPERTIES

EXERCISE 6.01: CREATING THE DODGEBALL CLASS

PHYSICAL MATERIALS

EXERCISE 6.02: ADDING A PROJECTILE MOVEMENT COMPONENT TO DODGEBALLPROJECTILE

TIMERS

SPAWNING ACTORS

EXERCISE 6.03: ADDING PROJECTILE-THROWING

LOGIC TO THE ENEMYCHARACTER

WALLS

EXERCISE 6.04: CREATING WALL CLASSES

VICTORY BOX

EXERCISE 6.05: CREATING THE VICTORYBOX CLASS

EXERCISE 6.06: ADDING THE PROJECTILEMOVEMENTCO MPONENT GETTER FUNCTION IN DODGEBALLPROJECTILE

ACTIVITY 6.01: REPLACING

THE SPAWNACTOR FUNCTION WITH SPAWNACTORDEFERRED IN ENEMYCHARACTER

SUMMARY

7. UE4 Utilities

INTRODUCTION

BLUEPRINT FUNCTION LIBRARIES

EXERCISE 7.01: MOVING THE CANSEEACTOR FUNCTION TO THE BLUEPRINT FUNCTION LIBRARY

ACTOR COMPONENTS

**EXERCISE 7.02: CREATING
THE HEALTHCOMPONENT
ACTOR COMPONENT**

**EXERCISE 7.03:
INTEGRATING THE
HEALTHCOMPONENT
ACTOR COMPONENT**

INTERFACES

BLUEPRINT NATIVE EVENTS

**EXERCISE 7.04: CREATING
THE HEALTHINTERFACE
CLASS**

ACTIVITY 7.01: MOVING THE

LOOKAT ACTOR LOGIC TO AN ACTOR COMPONENT

SUMMARY

8. User Interfaces

INTRODUCTION

GAME UI

UMG BASICS

EXERCISE 8.01: CREATING A WIDGET BLUEPRINT

ANCHORS

EXERCISE 8.02: EDITING

UMG ANCHORS

**EXERCISE 8.03: CREATING
THE RESTARTWIDGET C++
CLASS**

**EXERCISE 8.04: CREATING
THE LOGIC FOR ADDING
THE RESTARTWIDGET TO
THE SCREEN**

**EXERCISE 8.05: SETTING UP
THE
DODGEBALLPLAYERCONTR
OLLER BLUEPRINT CLASS**

PROGRESS BARS

**EXERCISE 8.06: CREATING
THE HEALTH BAR C++**

LOGIC

ACTIVITY 8.01: IMPROVING THE RESTARTWIDGET

SUMMARY

9. Audio-Visual Elements

INTRODUCTION

AUDIO IN UE4

EXERCISE 9.01: IMPORTING AN AUDIO FILE

EXERCISE 9.02: PLAYING A SOUND WHEN THE DODGEBALL BOUNCES OFF

A SURFACE

SOUND ATTENUATION

EXERCISE 9.03: TURNING THE BOUNCE SOUND INTO A 3D SOUND

EXERCISE 9.04: ADDING BACKGROUND MUSIC TO OUR GAME

PARTICLE SYSTEMS

EXERCISE 9.05: SPAWNING A PARTICLE SYSTEM WHEN THE DODGEBALL HITS THE PLAYER

ACTIVITY 9.01: PLAYING A

SOUND WHEN THE
DODGEBALL HITS THE
PLAYER

LEVEL DESIGN

EXERCISE 9.06: BUILDING A
LEVEL BLOCKOUT

EXTRA FEATURES

SUMMARY

10. Creating a
SuperSideScroller Game

INTRODUCTION

PROJECT BREAKDOWN

THE PLAYER CHARACTER

**EXERCISE 10.01: CREATING
THE SIDE-SCROLLER
PROJECT AND USING THE
CHARACTER MOVEMENT
COMPONENT**

**SIDE SCROLLER VERSUS
2D SIDE SCROLLER**

**ACTIVITY 10.01: MAKING
OUR CHARACTER JUMP
HIGHER**

**FEATURES OF OUR SIDE-
SCROLLER GAME**

ENEMY CHARACTER

POWER-UP

COLLECTIBLE

HUD (HEADS-UP DISPLAY)

STEPS IN ANIMATION

**CHARACTER ANIMATION
PIPELINE**

THE CONCEPT STAGE

THE 3D MODELING STAGE

THE RIGGING STAGE

ANIMATION

ASSET EXPORT AND IMPORT

EXERCISE 10.02: EXPLORING THE PERSONA EDITOR AND MANIPULATING THE DEFAULT MANNEQUIN SKELETON WEIGHTS

ACTIVITY 10.02: SKELETAL BONE MANIPULATION AND ANIMATIONS

ANIMATIONS IN UNREAL ENGINE 4

SKELETONS

SKELETAL MESHES

ANIMATION SEQUENCES

EXERCISE 10.03: IMPORTING AND SETTING UP THE CHARACTER AND ANIMATION

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

SUMMARY

11. Blend Spaces 1D, Key Bindings, and State Machines

INTRODUCTION

BLEND SPACES

1D BLEND SPACE VS NORMAL BLEND SPACE

EXERCISE 11.01: CREATING THE CHARACTERMOVEMENT 1D BLEND SPACE

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

MAIN CHARACTER ANIMATION BLUEPRINT

ANIMATION BLUEPRINTS

EXERCISE 11.02: ADDING THE BLEND SPACE TO THE CHARACTER ANIMATION BLUEPRINT

VELOCITY VECTORS

EXERCISE 11.03: ADDING THE BLEND SPACE TO THE CHARACTER ANIMATION BLUEPRINT

ACTIVITY 11.02: PREVIEWING THE RUNNING ANIMATION IN-GAME

INPUT BINDINGS

EXERCISE 11.04: ADDING INPUT FOR SPRINTING AND

THROWING

**EXERCISE 11.05:
REPARENTING THE
CHARACTER BLUEPRINT**

**EXERCISE 11.06: CODING
THE CHARACTER SPRINT
FUNCTIONALITY**

**ACTIVITY 11.03:
IMPLEMENTING THE
THROWING INPUT**

**ANIMATION STATE
MACHINES**

**EXERCISE 11.07: PLAYER
CHARACTER MOVEMENT
AND JUMP STATE MACHINE**

TRANSITION RULES

**EXERCISE 11.08: ADDING
STATES AND TRANSITION
RULES TO THE STATE
MACHINE**

**EXERCISE 11.09: TIME
REMAINING RATIO
FUNCTION**

**ACTIVITY 11.04: FINISHING
THE MOVEMENT AND
JUMPING STATE MACHINE**

SUMMARY

**12. Animation Blending
and Montages**

INTRODUCTION

ANIMATION BLENDING, ANIM SLOTS, AND ANIMATION MONTAGES

EXERCISE 12.01: SETTING UP THE ANIMATION MONTAGE

ANIMATION MONTAGES

EXERCISE 12.02: ADDING THE THROW ANIMATION TO THE MONTAGE

ANIM SLOT MANAGER

EXERCISE 12.03: ADDING A NEW ANIM SLOT

SAVE CACHED POSE

**EXERCISE 12.04: SAVE
CACHED POSE OF THE
MOVEMENT STATE
MACHINE**

LAYERED BLEND PER BONE

**EXERCISE 12.05: BLENDING
ANIMATION WITH THE
UPPER BODY ANIM SLOT**

THE THROW ANIMATION

**EXERCISE 12.06:
PREVIEWING THE THROW
ANIMATION**

THE SUPER SIDE

SCROLLER GAME ENEMY

**EXERCISE 12.07: CREATING
THE ENEMY BASE C++
CLASS**

**EXERCISE 12.08: CREATING
AND APPLYING THE ENEMY
ANIMATION BLUEPRINT**

**MATERIALS AND MATERIAL
INSTANCES**

**EXERCISE 12.09: CREATING
AND APPLYING THE ENEMY
MATERIAL INSTANCE**

**ACTIVITY 12.01: UPDATING
BLEND WEIGHTS**

SUMMARY

13. Enemy Artificial Intelligence

INTRODUCTION

ENEMY AI

AI CONTROLLER

AUTO POSSESS AI

EXERCISE 13.01: IMPLEMENTING AI CONTROLLERS

NAVIGATION MESH

EXERCISE 13.02: IMPLEMENTING A NAV MESH VOLUME FOR THE AI ENEMY

RECASTING THE NAV MESH

EXERCISE 13.03: RECASTING NAV MESH VOLUME PARAMETERS

ACTIVITY 13.01: CREATING A NEW LEVEL

BEHAVIOR TREES AND BLACKBOARDS

COMPOSITES

TASKS

DECORATORS

SERVICES

**EXERCISE 13.04: CREATING
THE AI BEHAVIOR TREE AND
BLACKBOARD**

**EXERCISE 13.05: CREATING
A NEW BEHAVIOR TREE
TASK**

**EXERCISE 13.06: CREATING
THE BEHAVIOR TREE LOGIC**

**ACTIVITY 13.02: AI MOVING
TO THE PLAYER LOCATION**

**EXERCISE 13.07: CREATING
THE ENEMY PATROL**

LOCATIONS

VECTOR TRANSFORMATION

EXERCISE 13.08:
SELECTING A RANDOM
POINT IN AN ARRAY

EXERCISE 13.09:
REFERENCING THE PATROL
POINT ACTOR

EXERCISE 13.10: UPDATING
BTTASK_FINDLOCATION

PLAYER PROJECTILE

EXERCISE 13.11: CREATING
THE PLAYER PROJECTILE

EXERCISE 13.12: INITIALIZING PLAYER PROJECTILE SETTINGS

ACTIVITY 13.03: CREATING THE PLAYER PROJECTILE BLUEPRINT

SUMMARY

14. Spawning the Player Projectile

INTRODUCTION

ANIM NOTIFIES AND ANIM NOTIFY STATES

EXERCISE 14.01: CREATING

AUANIM NOTIFY CLASS

**EXERCISE 14.02: ADDING
THE NOTIFY TO THE THROW
MONTAGE**

**PLAYING ANIMATION
MONTAGES**

**PLAYING ANIMATION
MONTAGES IN BLUEPRINTS**

**PLAYING ANIMATION
MONTAGES IN C++**

**EXERCISE 14.03: PLAYING
THE THROW ANIMATION IN
C++**

GAME WORLD AND

SPAWNING OBJECTS

**EXERCISE 14.04: CREATING
THE PROJECTILE SPAWN
SOCKET**

**EXERCISE 14.05:
PREPARING THE
SPAWNPROJECTILE()
FUNCTION**

**EXERCISE 14.06: UPDATING
THE
ANIM_PROJECTILENOTIFY
CLASS**

DESTROYING ACTORS

**EXERCISE 14.07: CREATING
THE DESTROYENEMY()**

FUNCTION

EXERCISE 14.08: DESTROYING PROJECTILES

ACTIVITY 14.01: PROJECTILE DESTROYING ENEMIES

VISUAL AND AUDIO EFFECTS

EXERCISE 14.09: ADDING EFFECTS WHEN THE ENEMY IS DESTROYED

EXERCISE 14.10: ADDING EFFECTS TO THE PLAYER PROJECTILE

EXERCISE 14.11: ADDING

VFX AND SFX NOTIFIES

ACTIVITY 14.02: ADDING EFFECTS FOR WHEN THE PROJECTILE IS DESTROYED

SUMMARY

15. Collectibles, Power-Ups, and Pickups

INTRODUCTION

UROTTATINGMOVEMENTCO MPONENT

EXERCISE 15.01: CREATING THE PICKABLEACTOR_BASE

CLASS AND ADDING UROTTATINGMOVEMENTCO MPONENT

ACTIVITY 15.01: PLAYER OVERLAP DETECTION AND SPAWNING EFFECTS IN PICKABLEACTOR_BASE

EXERCISE 15.02: CREATING THE PICKABLEACTOR_COLLECT ABLE CLASS

ACTIVITY 15.02: FINALIZING THE PICKABLEACTOR_COLLECT ABLE ACTOR

LOGGING VARIABLES USING UE_LOG

EXERCISE 15.03: TRACKING THE NUMBER OF COINS FOR THE PLAYER

UMG

TEXT WIDGET

ANCHORS

TEXT FORMATTING

EXERCISE 15.04: CREATING THE COIN COUNTER UI HUD ELEMENT

**ADDING AND CREATING
UMG USER WIDGETS**

EXERCISE 15.05: ADDING

THE COIN COUNTER UI TO THE PLAYER SCREEN

TIMERS

EXERCISE 15.06: ADDING THE POTION POWER-UP BEHAVIOR TO THE PLAYER

ACTIVITY 15.03: CREATING THE POTION POWER-UP ACTOR

EXERCISE 15.07: CREATING THE BRICK CLASS

EXERCISE 15.08: ADDING THE BRICK CLASS C++ LOGIC

SUMMARY

16. Multiplayer Basics

INTRODUCTION

MULTIPLAYER BASICS

SERVERS

DEDICATED SERVER

THE LISTEN SERVER

CLIENTS

EXERCISE 16.01: TESTING THE THIRD PERSON TEMPLATE IN MULTIPLAYER

THE PACKAGED VERSION

CONNECTIONS AND OWNERSHIP

ROLES

EXERCISE 16.02: IMPLEMENTING OWNERSHIP AND ROLES

THE SERVER WINDOW

SERVER CHARACTER

CLIENT 1 CHARACTER

THE OWNERSHIP TEST ACTOR

THE CLIENT 1 WINDOW

VARIABLE REPLICATION

REPLICATED VARIABLES

EXERCISE 16.03:
REPLICATING VARIABLES
USING REPLICATED,
REPNOTIFY,
DOREPLIFETIME, AND
DOREPLIFETIME_CONDITION

THE SERVER WINDOW

THE CLIENT 1 WINDOW

2D BLEND SPACES

EXERCISE 16.04: CREATING A MOVEMENT 2D BLEND SPACE

TRANSFORM (MODIFY) BONE

EXERCISE 16.05: CREATING A CHARACTER THAT LOOKS UP AND DOWN

ACTIVITY 16.01: CREATING A CHARACTER FOR THE MULTIPLAYER FPS PROJECT

SUMMARY

17. Remote Procedure Calls

INTRODUCTION

REMOTE PROCEDURE CALLS

SERVER RPC

DECLARATION

EXECUTION

VALID CONNECTION

MULTICAST RPC

DECLARATION

EXECUTION

CLIENT RPC

DECLARATION

EXECUTION

IMPORTANT CONSIDERATIONS WHEN USING RPCS

EXERCISE 17.01: USING REMOTE PROCEDURE CALLS

ENUMERATIONS

TENUMASBYTE

UMETA

DISPLAYNAME

HIDDEN

BLUEPRINTTYPE

MAX

**EXERCISE 17.02: USING C++
ENUMERATIONS IN THE
UNREAL ENGINE 4 EDITOR**

**BI-DIRECTIONAL CIRCULAR
ARRAY INDEXING**

**EXERCISE 17.03: USING BI-
DIRECTIONAL CIRCULAR
ARRAY INDEXING TO CYCLE
BETWEEN AN
ENUMERATION**

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

SUMMARY

18. Gameplay Framework Classes in Multiplayer

INTRODUCTION

GAMEPLAY FRAMEWORK CLASSES IN MULTIPLAYER

EXERCISE 18.01: DISPLAYING THE GAMEPLAY FRAMEWORK INSTANCE

VALUES

THE SERVER WINDOW

SERVER CHARACTER

CLIENT 1 CHARACTER

THE CLIENT 1 WINDOW

CLIENT 1 CHARACTER

SERVER CHARACTER

**GAME MODE, PLAYER
STATE, AND GAME STATE**

GAME MODE

CONSTRUCTOR

PLAYER STATE

GAME STATE

USEFUL BUILT-IN FUNCTIONALITY

EXERCISE 18.02: MAKING A SIMPLE MULTIPLAYER PICKUP GAME

ACTIVITY 18.01: ADDING DEATH, REPAWN, SCOREBOARD, KILL LIMIT, AND PICKUPS TO THE MULTIPLAYER FPS GAME

SUMMARY

Preface

About the Book

Game development can be both a creatively fulfilling hobby and a full-time career path. It's also an exciting way to improve your C++ skills and apply them in engaging and challenging projects.

Game Development Projects with Unreal Engine starts with the basic skills you'll need to get started as a game developer. The fundamentals of game design will be explained clearly and demonstrated practically with realistic exercises. You'll then apply what you've learned with challenging activities.

The book starts with an introduction to the Unreal Editor and key concepts such as actors, blueprints, animations, inheritance, and player input. You'll then move on to the first of three projects: building a dodgeball game. In this project, you'll explore line traces, collisions, projectiles, user interface, and sound effects, combining these concepts to showcase your new skills.

You'll then move on to the second project; a side-scroller game, where you'll implement concepts including animation blending, enemy AI, spawning objects, and collectibles. The final project is an FPS game, where you will cover the key concepts behind creating a multiplayer environment.

By the end of this Unreal Engine 4 game development book, you'll have the confidence and knowledge to get started on your own creative UE4 projects and bring your ideas to life.

About the Authors

Hammad Fozi is a lead game developer (Unreal Engine) at BIG IMMERSIVE.

Gonçalo Marques has been an active gamer since the age of 6. He worked as a freelancer for a Portuguese startup, Sensei Tech, where he developed an internal system using UE4 to produce datasets that assist with machine learning.

David Pereira started this game development career in 1998, where he learned to use Clickteam's The Games Factory and started making his own small games. David would like to acknowledge the following people: *I would like to thank my girlfriend, my family and my friends for supporting me on this journey. This book is dedicated to my grandmother Teresa ("E vai daí ós'pois...!").*

Devin Sherry is a technical designer at the game studio called People Can Fly and works on their newest IP that is built with Unreal Engine 4. Devin studied game development and game design at the University of Advancing Technology, where he obtained a bachelor's degree in game design in 2012.

Audience

This book is suitable for anyone who wants to get started using UE4 for game development. It will also be useful for anyone who has used Unreal Engine before and wants to consolidate, improve and apply their skills. To grasp the concepts explained in this book better, you must have prior knowledge of the basics of C++ and understand variables, functions, classes, polymorphism, and pointers. For full compatibility with the IDE used in this book, a Windows system is recommended.

About the Chapters

Chapter 1, Unreal Engine Introduction, explores the Unreal Engine editor. You will be introduced to the editor's interface, see how to manipulate actors in a level, understand the basics of the blueprint visual scripting language, and discover how to create material assets that can then be used by meshes.

Chapter 2, Working with Unreal Engine, introduces Unreal Engine game fundamentals, along with how to create a C++ project and set up the Content Folder of projects. You'll also be introduced to the topic of animations.

Chapter 3, Character Class Components and Blueprint Setup, introduces you to the Unreal Character class, along with the concept of object inheritance and how to work with input mappings.

Chapter 4, Player Input, introduces the topic of player input. You will learn how to associate a keypress or a touch input with an in-game action, such as jumping or moving, through the use of action mappings and axis mappings.

Chapter 5, Line Traces, starts a new project called Dodgeball. In this chapter, you will learn about the concept of line traces and the various ways in which they can be used in games.

Chapter 6, Collision Objects, explores the topic of object collision. You will learn about collision components, collision events, and physics simulation. You will also study the topic of timers, the projectile movement component, and physical materials.

Chapter 7, UE4 Utilities, teaches you how to implement some useful utilities available in Unreal Engine, including actor components, interfaces, and blueprint function libraries, which will help keep your projects well-structured and approachable for other people that join your team.

Chapter 8, User Interfaces, explores the topic of game UI. You will learn how to make menus and HUDs using Unreal Engine's UI system, UMG, as well as how to display the player character's health using a progress bar.

Chapter 9, Audio-Visual Elements, introduces the topic of sounds and particle effects in Unreal Engine. You will learn how to import sound files to the project and use them as both 2D and 3D sounds, as well as how to add existing particle systems to the game. Lastly, a new level will be made that uses all the game mechanics built in the last few chapters to conclude the Dodgeball project.

Chapter 10, Creating a SuperSideScroller Game, breaks down the SuperSideScroller game project's game mechanics. You will create the C++ SideScroller project template through the Epic Games Launcher, and learn the basic concepts of animation by manipulating the default mannequin skeleton and importing a custom skeletal mesh.

Chapter 11, Blend Spaces 1D, Key Bindings, and State Machines, introduces you to the tools available for developing smooth animation blending with Blend Spaces 1D and animation state machines. You will also jump into C++ code to develop the sprinting mechanics of the player character with the help of key bindings and the character movement component.

Chapter 12, Animation Blending and Montages, introduces you to animation montages and animation blending functionality within animation blueprints to develop the player's character throwing animation. You will learn about animation slots and use layered blends per bone to properly blend between the character's movement animations and the throw animation.

Chapter 13, Enemy Artificial Intelligence, covers AI and how to develop AI using behavior trees and blackboards. You will implement an AI that patrols along a custom path using a blueprint actor that you will develop.

Chapter 14, Spawning the Player Projectile, introduces you to Anim Notifies and how to spawn objects in the game world. You will implement a custom Anim Notify that spawns the player projectile at a specific frame of the throw animation. You will also develop functionality for the player projectile that allows this projectile to destroy enemy AI.

Chapter 15, Collectibles, Power-Ups, and Pickups, demonstrates how to create a custom potion power-up that manipulates the player's movement, as well as a collectible coin for the player character. You will also learn more about UMG by developing a simple UI to count the number of collectibles found by the player.

Chapter 16, Multiplayer Basics, introduces you to important multiplayer concepts such as the server-client architecture, connections, actor ownership, roles, and variable replication. You'll also learn how to make a 2D blendspace and how to use the Transform Modify Bone node. You'll start working on a multiplayer FPS project by creating a character that walks, jumps, looks up/down, and has two replicated stats: health and armor.

Chapter 17, Remote Procedure Calls, introduces you to remote procedure calls, how to use enumerations in Unreal Engine 4, and how to use bi-directional circular array indexing. You'll also expand the multiplayer FPS project by adding the concept of weapons and ammo.

Chapter 18, Gameplay Framework Classes in Multiplayer, is the final chapter of this book and explains where the gameplay framework classes exist in multiplayer, how to use the game state and player state classes, as well as how to implement some useful built-in functionality. You'll also see how to use match states and other concepts in game mode. Finally, you'll finish the multiplayer FPS project by adding the concepts of death, respawn, scoreboard, kill limit, and pickups.

Conventions

Code words in the text, folder names, filenames, file extensions, pathnames, dummy URLs, and user input are shown as follows:

"Open **Project Settings** and go to the **Collision** subsection within the **Engine** section."

Words that you see on the screen, for example, in menus or dialog boxes, also appear in the text like this:

"Click the **New Object Channel** button, name it **Dodgeball**, and set its **Default Response** to **Block**."

A block of code is set as follows:

```
if (bCanSeePlayer)
{
    //Start throwing dodgeballs

    GetWorldTimerManager().SetTimer(ThrowTime
rHandle, this,
&AEnemyCharacter::ThrowDodgeball, ThrowingIn
terval, true, ThrowingDelay);

}
```

New terms, abbreviations, and important words are shown like this: "In this chapter, we're going to cover **Remote Procedure Calls (RPCs)**, which is another important multiplayer concept that allows the server to execute functions on the clients and vice versa."

Before You Begin

This section will guide you through the installation and

configuration steps to get you set up with the necessary working environment.

Installing Visual Studio

Because we'll be using C++ while working with Unreal Engine 4, we'll need an **IDE (Integrated Development Environment)** that easily works alongside the engine. Visual Studio Community is the best IDE you have available for this purpose on Windows. If you're using macOS or Linux, you'll have to use another IDE, such as Visual Studio Code, QT Creator, or Xcode (available exclusively on macOS).

The instructions given in this book are specific to Visual Studio Community on Windows, so if you are using a different OS and/or IDE, then you will need to do your research on how to set these up for use in your working environment. In this section, you'll be taken through the installation of Visual Studio, so that you can easily edit UE4's C++ files:

1. Go to the Visual Studio download web page at <https://visualstudio.microsoft.com/downloads>. The recommended Visual Studio Community version for the Unreal Engine 4 version we'll be using in this book (4.24.3) is Visual Studio Community 2019. Be sure to download that version.
2. When you do, open the executable file that you just downloaded. It should

eventually take you to the following window where you'll be able to pick the modules of your Visual Studio installation. There, you'll have to tick the **Game Development with C++** module and then click the **Install** button in the bottom-right corner of the window. After you click that button, Visual Studio will start downloading and installing. When the installation is complete, it may ask you to reboot your PC. After rebooting your PC, Visual Studio should be installed and ready for use.

3. When you run Visual Studio for the first time, you may see a few windows, the first one of which is the login window. If you have a Microsoft Outlook/Hotmail account, you should use that account to log in, otherwise, you can skip login by clicking **Not now, maybe later.**

Note

If you don't input an email address, you will only have 30 days to use Visual Studio before it locks out and

you have to input an email address to continue using it.

4. After that, you will be asked to choose a color scheme. The **Dark** theme is the most popular and the one we will be using in this section.

Finally, you can select the **Start Visual Studio** option. When you do so, however, you can close it again. We will be taking a deeper look at how to use Visual Studio in one of the first chapters in this book.

Epic Games Launcher

To access Unreal Engine 4, you'll need to download the Epic Games Launcher, available at this link:

<https://www.unrealengine.com/get-now>. This link will allow you to download the Epic Games Launcher for Windows and macOS. If you use Linux, you'll have to download the Unreal Engine source code and compile it from the source – <https://docs.unrealengine.com/en-US/GettingStarted/DownloadingUnrealEngine>:

1. There, you'll have to choose the **Publishing License** option and click the **SELECT** button below it. This license will allow you to use UE4 to create projects that you can publish directly to your users (in a digital game

store, for instance). The **Creators License**, however, will not allow you to publish your work directly to your end users.

2. After that, you'll be asked to accept the terms and conditions, and once you accept those, a **.msi** file will be downloaded to your computer. Open this **.msi** file when it finishes downloading, which will prompt you to install the Epic Games Launcher. Follow the installation instructions and then launch the Epic Games Launcher. When you do so, you should be greeted with a login screen.
3. If you already have an account, you can simply log in with your existing credentials. If you don't, you'll have to sign up for an Epic Games account by clicking the **Sign Up** text at the bottom.

Once you log in with your account, you should be greeted by the **Home** tab. From there, you'll want to go to the **Unreal Engine** tab by clicking the text that says **Unreal Engine**.

- When you've done that, you'll be greeted with the **Unreal Engine** tab. The Unreal Engine tab acts as a hub for Unreal Engine resources. From this page, you'll be able to access the following:

- The **News** page, on which you'll be able to take a look at all the latest Unreal Engine news.
- The **Youtube** channel, on which you'll be able to watch dozens of tutorials and live streams that go into detail about several different Unreal Engine topics.
- The **AnswerHub** page, on which you'll be able to see, ask, and answer questions posed and answered by the Unreal Engine community.
- The **Forums** page, on which you'll be able to access the Unreal Engine forums.
- The **Roadmap** page, on which you'll be able to access the

Unreal Engine roadmap,
including features delivered
in past versions of the engine,
as well as features that are
currently in development for
future versions.

5. At the top of the Epic Games Launcher, while in the Unreal Engine tab, you'll be able to see several other tabs, such as the **Unreal Engine** tab (the sub-tab you're currently seeing), the **Learn** tab, and the **Marketplace** tab. Let's take a look at these Unreal Engine sub-tabs.
6. The **Learn** tab will allow you to access several resources related to learning how to use Unreal Engine 4. From here, you'll be able to access the **Get Started with Unreal Engine 4** page, which will take you to a page that allows you to choose how you want to begin learning about Unreal Engine 4.
7. You'll also be able to access the **Documentation** page, which contains a reference to the classes used in the engine's source code, and the **Unreal**

Online Learning page, which contains several courses on specific topics of Unreal Engine 4.

8. To the right of the **Learn** tab is the **Marketplace** tab. This tab shows you several assets and code plugins made by members of the Unreal Engine community. Here, you'll be able to find 3D assets, music, levels, and code plugins that will help you advance and accelerate the development of your game.
9. Finally, to the right of the **Marketplace** tab, we have the **Library** tab. Here, you'll be able to browse and manage all your Unreal Engine version installations, your Unreal Engine projects, and your Marketplace asset vault. Because we have none of these things yet, these sections are all empty. Let's change that.
10. Click the yellow plus sign to the right of the **ENGINE VERSIONS** text. This should make a new icon show up, where you'll be able to choose your

desired Unreal Engine version.

11. Throughout this book, we'll be using version **4.24.3** of Unreal Engine. After you've selected that version, click the **Install** button:

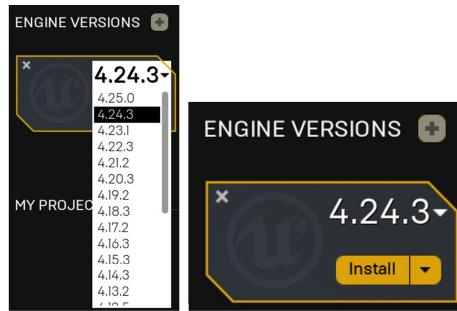


Figure 0.1: The icon that allows you to install Unreal Engine 4.24.3

12. After you've done this, you'll be able to choose the installation directory for this Unreal Engine version, which will be of your choosing, and you should then click the **Install** button again.

Note

If you are having issues installing the 4.24 version, make sure to install it on your D drive, with the shortest path possible (that is, don't try to install it too many folders deep and make sure

(those folders have short names).

13. This will result in the installation of Unreal Engine 4.24.3 starting. When the installation is done, you can launch the editor by clicking the **Launch** button of the version icon:



Figure 0.2: The version icon once installation has finished

Code Bundle

You can find the code files for this book on GitHub at <https://packt.live/38urh8v>. Here, you will find the exercise code, activity solutions, images, and any other assets such as datasets that are required to complete the practical elements in this book.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have any questions about this book, please mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you could report this to us. Please visit www.packtpub.com/support/errata and complete the form.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you could provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Please leave a review

Let us know what you think by leaving a detailed, impartial review on Amazon. We appreciate all feedback – it helps us continue to make great products and help aspiring developers build their skills. Please spare a few minutes to give your thoughts – it makes a big difference to us.

1. Unreal Engine Introduction

Overview

This chapter will be an introduction to the Unreal Engine editor. You will get to know about the editor's interface; how to add, remove, and manipulate objects in a level; how to use Unreal Engine's Blueprint visual scripting language; and how to use materials in combination with meshes.

By the end of this chapter, you will be able to navigate the Unreal Engine editor, create your own Actors, manipulate them inside the level, and create materials.

Introduction

Welcome to *Game Development Projects with Unreal Engine*. If this is the first time you're using **Unreal Engine 4 (UE4)**, this book will support you in getting started with one of the most in-demand game engines on the market. You will discover how to build up your game development skills and how to express yourself through the creation of your own video games. If you've already tried using UE4, this book will help you further develop your knowledge and skills, so you can build games more easily and effectively.

A game engine is a software application that allows you to produce video games from the ground up. Their feature sets vary significantly but usually allow you to import multimedia

files, such as 3D models, images, audio, and video, and manipulate those files through the use of programming, where you can use programming languages such as C++, Python, and Lua, among others.

Unreal Engine 4 uses two main programming languages, C++ and Blueprint, the latter being a visual scripting language that allows you to do most of what C++ also allows. Although we will be teaching a bit of Blueprint in this book, we will be focusing mostly on C++, and hence expect you to have a basic understanding of the language, including topics such as *Variables*, *Functions*, *Classes*, *Inheritance*, and *Polymorphism*. We will remind you about these topics throughout the book where appropriate.

Examples of popular video games made with Unreal Engine 4 include *Fortnite*, *Final Fantasy VII Remake*, *Borderlands 3*, *Star Wars: Jedi Fallen Order*, *Gears 5*, and *Sea of Thieves*, among many others. All of these have a very high level of visual fidelity, are well-known, and have or had millions of players.

At the following link, you'll be able to see some of the great games made with Unreal Engine 4:

<https://www.youtube.com/watch?v=lrPc2LorfN4>. This showcase will show you the variety of games that Unreal Engine 4 allows you to make, both in visual and gameplay style.

If you'd like to one day make games such as the ones shown in the video, or contribute to them in any way, you've taken the first step in that direction.

We'll now begin with this first step, where we will start learning about the Unreal Engine editor. We will learn about its interface, how to manipulate objects inside a level, how to create our own objects, how to use the Blueprint scripting language, and what the main game events do, as well as how to create materials for meshes.

Let's start this chapter by learning how to create a new Unreal Engine 4 project in this first exercise.

Note

Before you continue this chapter, make sure you have installed all the necessary software mentioned in the Preface.

Exercise 1.01: Creating an Unreal Engine 4 Project

In this first exercise, we will learn how to create a new Unreal Engine 4 project. UE4 has predefined project templates which allow you to implement a basic setup for your project. We'll be using the **Third Person** template project in this exercise.

The following steps will help you complete this exercise:

1. After installing Unreal Engine version 4.24, launch the editor by clicking the **Launch** button of the version icon.
2. After you've done so, you'll be greeted with the engine's projects window, which will show you the existing projects that you can open and work on and also give you the option to create a new project. Because we have no projects yet, the **Recent Projects**

section will be empty. To create a new project, you'll first have to choose

Project Category, which in our case will be **Games**.

3. After you've selected that option, click the **Next** button. After that, you'll see the project templates window. This window will show all the available project templates in the Unreal Engine. When creating a new project, instead of having that project start off empty, you have the option to add some assets and code out of the box, which you can then modify to your liking. There are several project templates available for different types of games, but we'll want to go with the **Third Person** project template in this case.
4. Select that template and click the **Next** button, which should take you to the **Project Settings** window.

In this window, you'll be able to choose a few options related to your project:

1. **Blueprint or C++:**

Choose whether you want to be able to add C++ classes.

The default option may be **Blueprint**, but in our case, we'll want to select the **C++** option.

2. **Quality:** Choose whether you want your project to have high-quality graphics or high performance. You can set this option to **Maximum Quality**.
3. **Raytracing:** Choose whether you want Raytracing enabled or disabled.
Raytracing is a novel graphics rendering technique which allows you to render objects by simulating the path of light (using light rays) over a digital environment. Although this technique is rather costly in terms of performance, it also provides much more realistic graphics, especially when it comes to lighting. You can set it to **disabled**.
4. **Target Platforms:**

Choose the main platforms you'll want this project to run on. Set this option to **Desktop/Console**.

5. **Starter Content:** Choose whether you want this project to come with an additional set of basic assets. Set this option to **With Starter Content**.
6. **Location and Name:** At the bottom of the window, you'll be able to choose the location where your project will be stored on your computer and its name.
5. After you've made sure that all the options are set to their intended values, click the **Create Project** button. This will cause your project to be created according to the parameters you set and may take a few minutes until it's ready.

Let's now start learning about Unreal Engine 4 by performing the steps in the next section, where we'll learn some of the basics of using the editor.

Getting to Know Unreal

You will now be introduced to the Unreal Engine editor, which is a fundamental topic to get familiar with Unreal Engine 4.

When your project has finished generating, you should see the Unreal Engine editor open automatically. This screen is likely the one that you will see the most when working with Unreal Engine, so it is important that you get accustomed to it.

Let's break down what we see in the editor window:

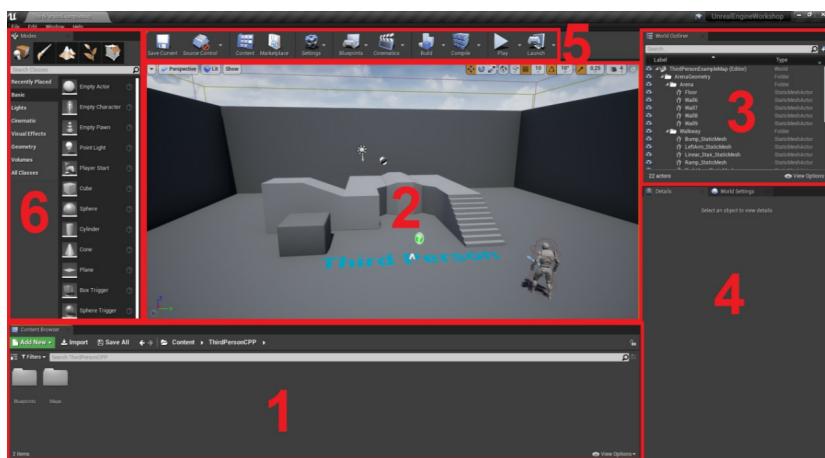


Figure 1.1: The Unreal Engine editor divided in its six main windows

- 1. Content Browser:** The window that occupies the majority of the bottom of the screen is the **Content Browser**. This window will let you browse and manipulate all the files and assets located inside your project's folder. As was mentioned at the start of the chapter, Unreal Engine will allow you to import several types of multimedia

files, and **Content Browser** is the window that will allow you to browse and edit them in their respective sub-editors. Whenever you create an Unreal Engine project, it will always generate a **Content** folder. This folder will be the **root directory** of the **Content Browser**, meaning you can only browse files inside that folder. You can see the directory you're currently browsing inside **Content Browser** by looking at the top of it, which, in our case, is **Content -> ThirdPersonCPP**.

If you click the icon to the left of the **Filters** button, at the very left of **Content Browser**, you will be able to see the directory hierarchy of the **Content** folder. This directory view allows you to select, expand, and collapse individual directories in the **Content** folder of your project:

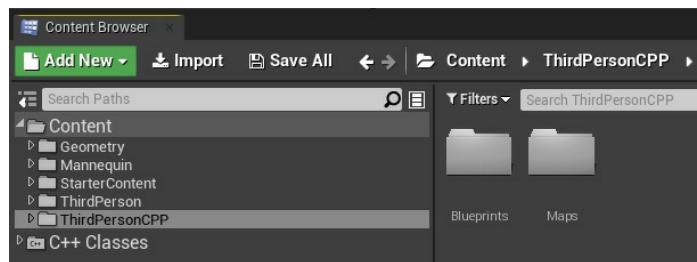


Figure 1.2: Content Browser's directory view

2. **Viewport:** At the very center of the screen, you'll be able to see the **Viewport** window. This will show you the content of the current level and will allow you to navigate through your level as well as adding, moving, removing, and editing objects inside it. It also contains several different parameters regarding visual filters, object filters (which objects you can see), and the lighting in your level.
3. **World Outliner:** At the top-right corner of the screen, you'll see **World Outliner**. This will allow you to quickly list and manipulate the objects that are at your level. **Viewport** and **World Outliner** work hand in hand in allowing you to manage your level, where the former will show you what it looks like and the latter will help you manage and organize it. Similar to **Content Browser**, **World Outliner** allows you to organize the objects in your level in directories, with the difference being that **Content**

Browser shows the *assets* in your project and **World Outliner** shows you the *objects* in your level.

4. The **Details** panel and **World Settings**: At the far right of the screen, below **World Outliner**, you'll be able to see two windows – the **Details** panel and the **World Settings** window. The **Details** window allows you to edit the properties of an object that you selected in your level. As there are no objects selected in the screenshot, it is empty. However, if you select any object in your level by *left-clicking* on it, its properties should appear in this window, as shown in the following screenshot:

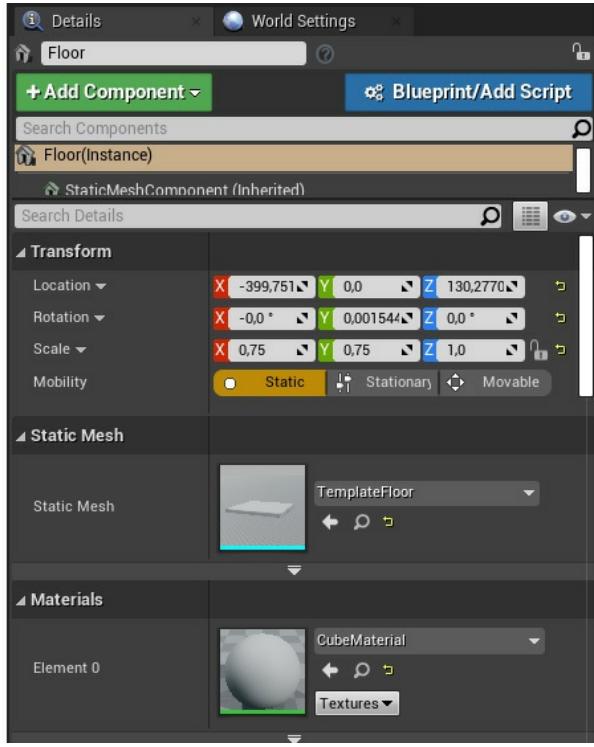


Figure 1.3: The Details tab

The **World Settings** window allows you to set the overall settings of your level, instead of those of individual objects. Here, you'll be able to change things such as the Kill Z (the height at which you want your objects to be destroyed) and the desired lighting settings, among others:

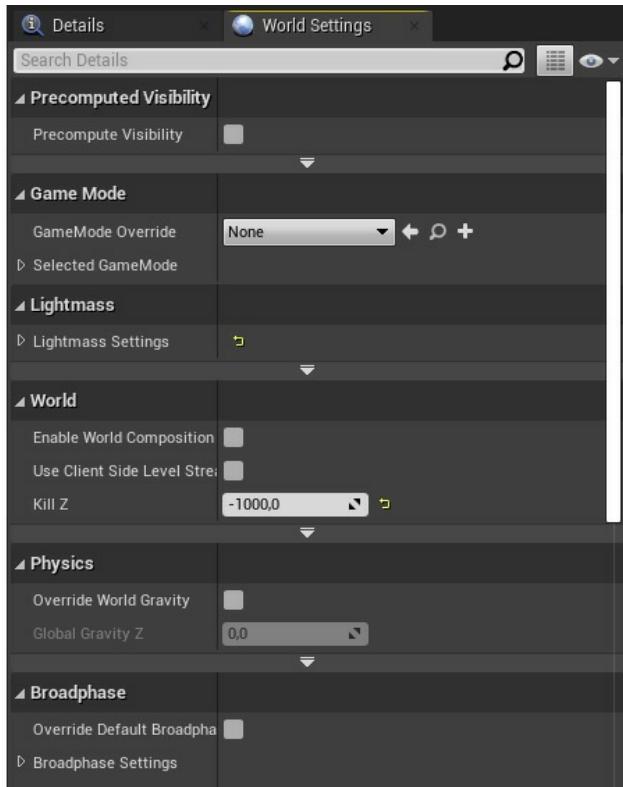


Figure 1.4: The World Settings window

5. **Toolbar:** At the top of the screen you'll see the editor **Toolbar**, where you'll be able to save your current level, access the project and editor settings, and play your level, among other things.

Note

*We will only be using some of the buttons from these toolbars, namely, the **Save Current Settings**,*

Blueprints, Build, and Play buttons.

6. **Modes:** At the very left of the screen, you'll see the **Modes** window. It will allow you to drag objects to your levels, such as cubes and spheres, light sources, and other types of objects designed for a wide variety of purposes.

Now that we have learned about the main windows of the Unreal Engine editor, let's take a look at how to manage those windows.

Editor Windows

As we've seen, the Unreal Engine editor is comprised of many windows, all of which are resizable, movable, and have a corresponding tab on top of them. You can *click and hold* a window's tab and drag it in order to move it somewhere else. You can hide tab labels by *right-clicking* them and selecting the **Hide** option:



Figure 1.5: How to hide a tab

If the tab labels have been hidden, you can get them to reappear by clicking the *yellow triangle* at the top-left corner of that window, as shown in the following figure:

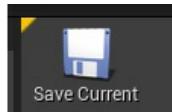


Figure 1.6: The yellow triangle that allows you to show a window's tab

Keep in mind that you can browse and open all the windows available in the editor, including the ones that were just mentioned, by clicking the **Window** button in the top-left corner of the editor.

Another very important thing you should know is how to play your level from inside the editor (also known as **PIE**). At the right edge of the editor **Toolbar**, you'll see the **Play** button. If you click it, you'll start playing the currently open level inside the editor.

Once you hit **Play**, you'll be able to control the player character in the level by using the *W*, *A*, *S*, and *D* keys to move the player character, the spacebar to jump, and move the **Mouse** to rotate the camera:

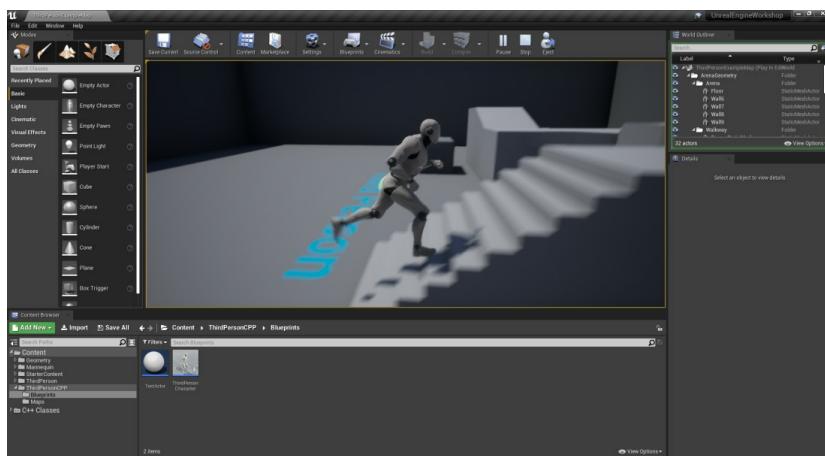


Figure 1.7: The level being played inside the editor

You can then press the *Esc* key (Escape) in order to stop playing the level.

Now that we've gotten accustomed to some of the editor's windows, let's take a deeper look at the **Viewport** window.

Viewport Navigation

We mentioned in the previous section that the **Viewport** window will allow you to visualize your level, as well as manipulating the objects inside it. Because this is a very important window for you to use and has a lot of functionality, we're going to learn more about it in this section.

Before we start learning about the **Viewport** window, let's quickly get to know about **Levels**. In UE4, levels represent a **collection of objects**, as well as their locations and properties. The **Viewport** window will always show you the contents of the currently selected level, which in this case was already made and was generated alongside the **Third Person** template project. In this level, you'll be able to see four wall objects, one ground object, a set of stairs, and some other elevated objects, as well as the player character represented by the UE4 mannequin. You can create multiple levels and switch between them by opening them from **Content Browser**.

In order to manipulate and navigate through the currently selected level, you'll have to use the **Viewport** window. If you press and hold the *left mouse button* inside the window, you'll be able to rotate the camera horizontally by moving the mouse *left* and *right*, and move the camera forward and backward by moving the mouse *forward* and *backward*. You can achieve similar results by holding the *right mouse button*, except the camera will rotate vertically when moving the mouse *forward* and *backward*, which allows you to rotate the camera both

horizontally and vertically.

Additionally, you can also move around the level by clicking and holding the **Viewport** window with the *right mouse button* (the *left mouse button* works too, but using it for movement is not as useful due to there not being as much freedom when rotating the camera) and using the *W* and *S* keys to move forward and backward, the *A* and *D* keys to move sideways, and the *E* and *Q* keys to move up and down.

If you look at the top-right corner of the **Viewport** window, you will see a small camera icon with a number next to it, which will allow you to change the speed at which the camera moves in the **Viewport** window.

Another thing you can do in the **Viewport** window is to change its visualization settings. You can change the type of visualization in the **Viewport** window by clicking the button that currently says **Lit**, which will show you all the options available for different lighting and other types of visualization filters.

If you click the **Perspective** button, you'll have the option to switch between seeing your level from a perspective view, as well as from an orthographic view, the latter of which may help you build your levels faster.

Let's now move on to the topic of manipulating objects, also known as **Actors**, in your level.

Manipulating Actors

In Unreal Engine, all the objects that can be placed in a level are referred to as **Actors**. In a movie, an actor would be a human playing a character, but in UE4, every single object you see in your level, including walls, floors, weapons, and characters, is an Actor.

Every Actor must have what's called a **Transform** property, which is a collection of three things:

- **Location:** A **Vector** property signifying the position of that Actor in the level in the X , Y , and Z axis. A vector is simply a tuple with three floating point numbers, one for the location of the point in each axis.
- **Rotation:** A **Rotator** property signifying the rotation of that Actor along the X , Y , and Z axis. A rotator is also a tuple with three floating point numbers, one for the angle of rotation in each axis.
- **Scale:** A **Vector** property signifying the scale (meaning size) of that Actor in the level in the X , Y , and Z axis. This is also a collection of three floating point numbers, one for the scale value in each axis.

Actors can be moved, rotated, and scaled in a level, which will modify their **Transform** property accordingly. In order to do this, select any object in your level by *left-clicking* on it. You should see the **Move** tool appear:

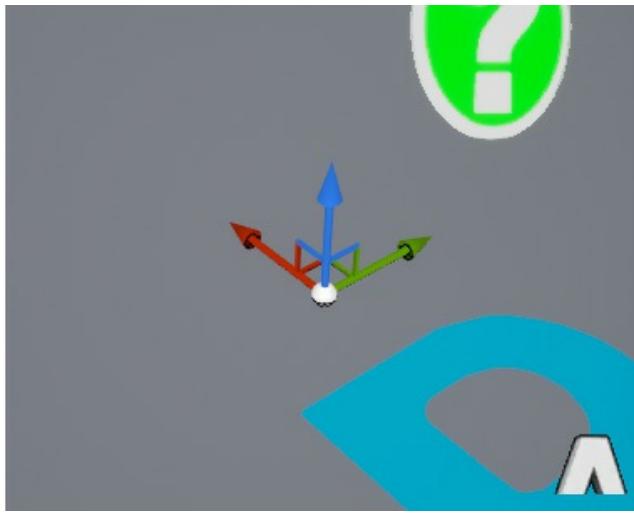


Figure 1.8: The Move tool, which allows you to move an Actor in the level

The Move tool is a three-axis gizmo that allows you to move an object in any of the axes simultaneously. The red arrow of the Move tool (pointing to the left in the preceding image) represents the X axis, the green arrow (pointing to the right in the preceding image) represents the Y axis, and the blue arrow (pointing up in the preceding image) represents the Z axis. If you *click and hold* either of these arrows and then drag them around the level, you will move your Actor along that axis in the level. If you click the handles that connect two arrows together, you will move the Actor along both those axes simultaneously, and if you click the white sphere at the intersection of all the arrows, you will move the Actor freely along all three axes:

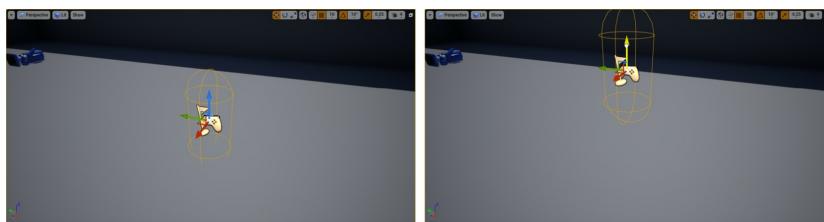


Figure 1.9: An actor being moved on the Z axis using the Move tool

The Move tool will allow you to move an Actor around the

level, but if you want to rotate or scale an Actor, you'll need to use the Rotate and Scale tools, respectively. You can switch between the Move, Rotate, and Scale tools by pressing the *W*, *E*, and *R* keys, respectively. Press *E* in order to switch to the Rotate tool:

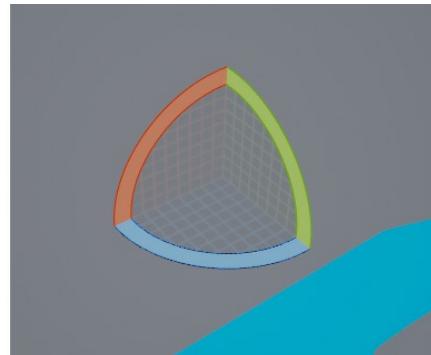


Figure 1.10: The Rotate tool, which allows you to rotate an Actor

The Rotate tool will, as expected, allow you to rotate an Actor in your level. You can *click and hold* any of the arcs in order to rotate the Actor around its associated axis. The red arc (upper left in the previous image) will rotate the Actor around the *X* axis, the green arc (upper right in the previous image) will rotate the Actor around the *Y* axis, and the blue arc (lower center in the previous image) will rotate the Actor around the *Z* axis:

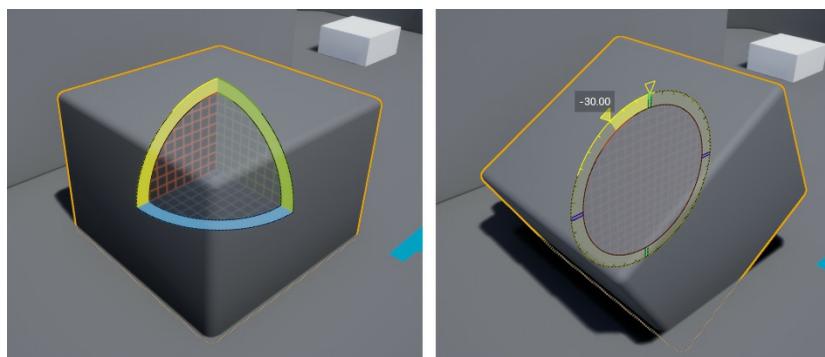


Figure 1.11: A cube before and after being rotated 30 degrees around the X axis

Keep in mind that an object's rotation around the *X* axis is usually designated as **Roll**, its rotation around the *Y* axis is usually designated as **Pitch**, and its rotation around the *Z* axis is usually designated as **Yaw**.

Lastly, we have the Scale tool. Press *R* in order to switch to it:

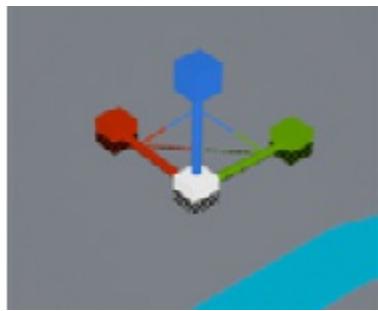


Figure 1.12: The Scale tool

The Scale tool will allow you to increase and decrease the scale (size) of an Actor in the *X*, *Y*, and *Z* axes, where the red handle (left in the previous image) will scale the Actor on the *X* axis, the green handle (right in the previous image) will scale the Actor on the *Y* axis, and the blue handle (upper in the previous image) will scale the Actor on the *Z* axis:

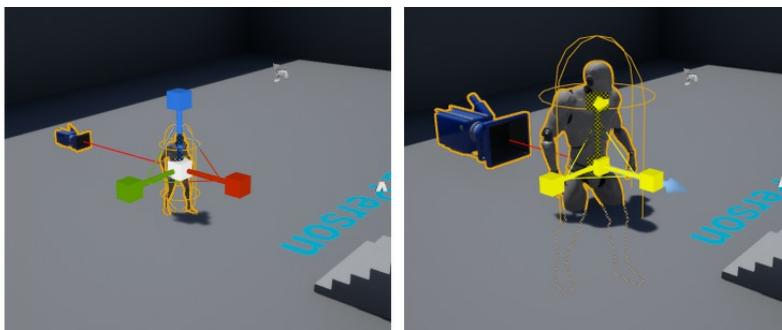


Figure 1.13: A character Actor before and after being scaled on all three axes

You can also toggle between the Move, Rotate, and Scale tools by clicking the following icons at the top of the **Viewport** window:

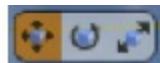


Figure 1.14: The Move, Rotate, and Scale tool icons

Additionally, you can change the increments with which you move, rotate, and scale your objects through the grid snapping options to the right of the Move, Rotate, and Scale tool icons. By pressing the buttons currently in orange, you'll be able to disable snapping altogether, and by pressing the buttons showing the current snapping increments, you'll be able to change those increments:

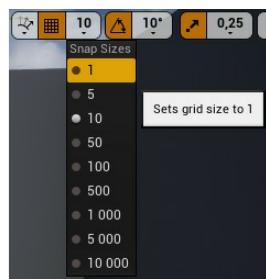


Figure 1.15: The grid snapping icons for moving, rotating, and scaling

Now that you know how to manipulate Actors already present in your level, let's learn how to add and remove Actors to and from our level in the next exercise.

Exercise 1.02: Adding and Removing Actors

In this exercise, we will be adding and removing Actors from our level.

When it comes to adding Actors to your level, there are two main ways in which you can do so: by dragging assets from **Content Browser**, or by dragging the default assets from

the **Modes** window's Place Mode.

The following steps will help you complete this exercise:

1. If you go to the **ThirdPersonCPP -> Blueprints** directory inside **Content Browser**, you will see the **ThirdPersonCharacter** Actor. If you drag that asset to your level using the *left mouse button*, you will be able to add an instance of that Actor to it, and it will be placed wherever you let go of the *left mouse button*:

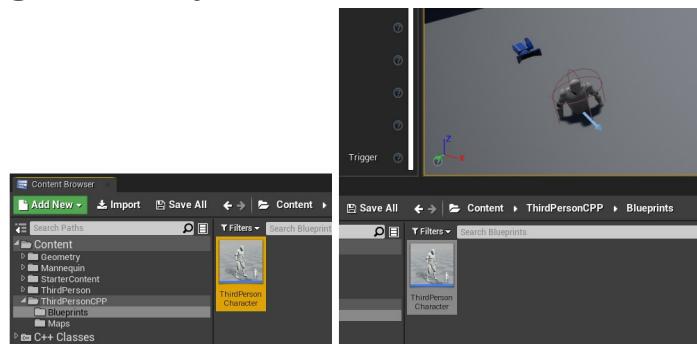


Figure 1.16: Dragging an instance of the ThirdPersonCharacter Actor to our level

2. You can similarly drag an Actor from the **Modes** window to your level as well:

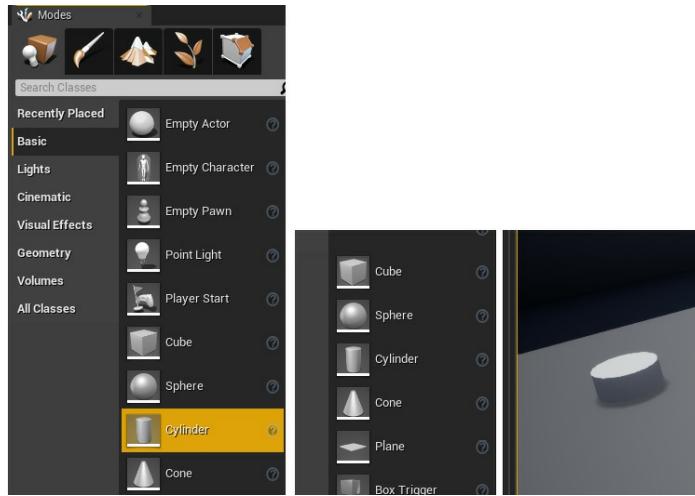


Figure 1.17: Dragging a Cylinder Actor to our level

3. In order to delete an Actor, you can simply select the Actor and press the *Delete* key. You can also *right-click* on an Actor to take a look at the many other options available to you regarding that Actor.

Note

Although we won't be covering this topic in this book, one of the ways in which developers can populate their levels with simple boxes and geometry, for prototyping purposes, is BSP Brushes. These can be quickly molded into your desired shape as you build your levels. To find more

information on BSP Brushes, go to this page:

<https://docs.unrealengine.com/en-US/Engine/Actors/Brushes>.

And with this, we conclude this exercise and have learned how to add and remove Actors to and from our level.

Now that we know how to navigate the **Viewport** window, let's learn about Blueprint Actors.

Blueprint Actors

In UE4, the word Blueprint can be used to refer to two different things: UE4's visual scripting language or a specific type of asset, also referred to as a Blueprint class or Blueprint asset.

As we've mentioned before, an Actor is an object that can be placed in a level. This object can either be an instance of a C++ class or an instance of a Blueprint class, both of which must inherit from the Actor class (either directly or indirectly). So, what is the difference between a C++ class and a Blueprint class, you may ask? There are a few:

- If you add programming logic to your C++ class, you'll have access to more advanced engine functionality than you would if you were to create a Blueprint class.
- In a Blueprint class, you can easily

view and edit visual components of that class, such as a 3D mesh or a Trigger Box Collision, as well as modifying properties defined in the C++ class that is exposed to the editor, which makes managing those properties much easier.

- In a Blueprint class, you can easily reference other assets in your project, whereas in C++, you can also do so but less simply and less flexibly.
- Programming logic that runs on Blueprint visual scripting is slower in terms of performance than that of a C++ class.
- It's simple to have more than one person work on a C++ class simultaneously without conflicts in a source version platform, whereas with a Blueprint class, which is interpreted as a binary file instead of a text file, this will cause conflicts in your source version platform if two different people edit the same Blueprint class.

Note

In case you don't know what a source version platform is, this is how several developers can work on the same project and have it updated with the work done by other developers. In these platforms, different people can usually edit the same file simultaneously, as long as they edit different parts of that file, and still receive updates that other programmers did without them affecting your work on that same file. One of the most popular source version platforms is GitHub.

Keep in mind that Blueprint classes can inherit either from a C++ class or from another Blueprint class.

Lastly, before we move on to creating our first Blueprint Class, another important thing you should know is that you can write programming logic in a C++ class and then create a Blueprint class that inherits from that class, but can also access its properties and methods if you specify that in the C++ class. You can have a Blueprint class edit properties defined in the C++ class as well as calling and overriding functions, using the Blueprint scripting language. We will be doing some of these things in this book.

Now that you know a bit more about Blueprint classes, let's create our own in this next exercise.

Exercise 1.03: Creating Blueprint Actors

In this short exercise, we will learn how to create a new Blueprint Actor.

The following steps will help you complete this exercise:

1. Go to the **ThirdPersonCPP** -> **Blueprints** directory inside **Content Browser** and *right-click* inside it. The following window should pop up:

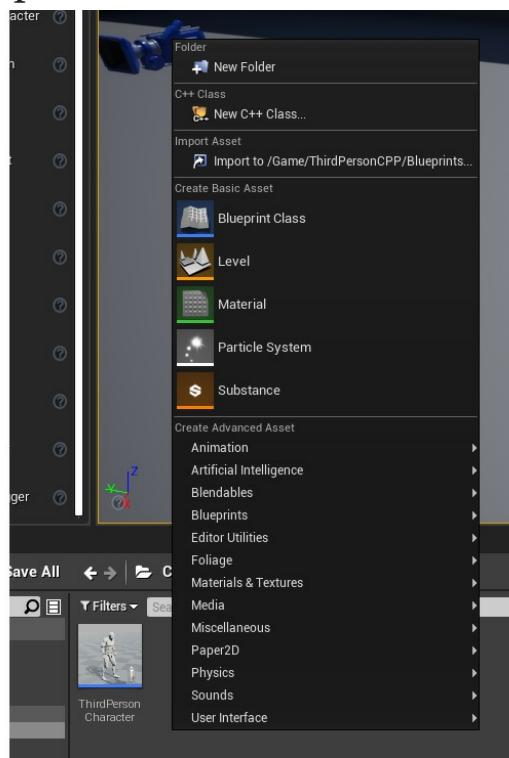


Figure 1.18: The options window that appears when you right-click inside content browser

This options menu contains the types of assets that you can create in UE4 (Blueprints are simply a type of asset, along with other types of assets, such as **Level**, **Material**, and **Sound**).

2. Click the **Blueprint Class** icon to create a new Blueprint class. When you do, you will be given the option to choose the C++ or Blueprint class that you want to inherit from:

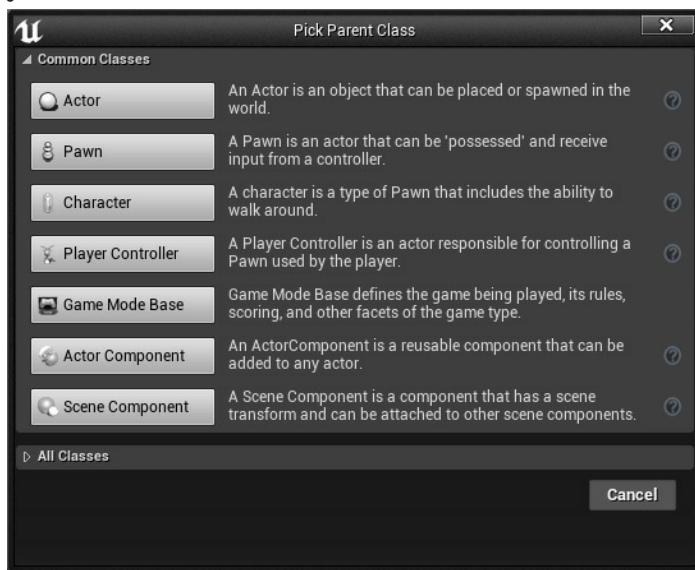


Figure 1.19: The Pick Parent Class window that pops up when you create a new Blueprint class

3. Select the first class from this window, the **Actor** class. After this, you will automatically select the text of the new Blueprint class to easily name it what

you want. Name this Blueprint class **TestActor** and press the **Enter** key to accept this name.

After following these steps, you will have created your Blueprint class and so have completed this exercise. After you've created this asset, double-click on it with the *left mouse button* to open the Blueprint editor.

The Blueprint Editor

The Blueprint editor is a sub-editor within the Unreal Engine editor specifically for Blueprint classes. Here, you'll be able to edit the properties and logic for your Blueprint classes, or those of their parent class, as well as their visual appearance.

When you open an Actor Blueprint class, you should see the Blueprint editor. This is the window that will allow you to edit your Blueprint classes in UE4. Let's learn about the windows that you're currently seeing:

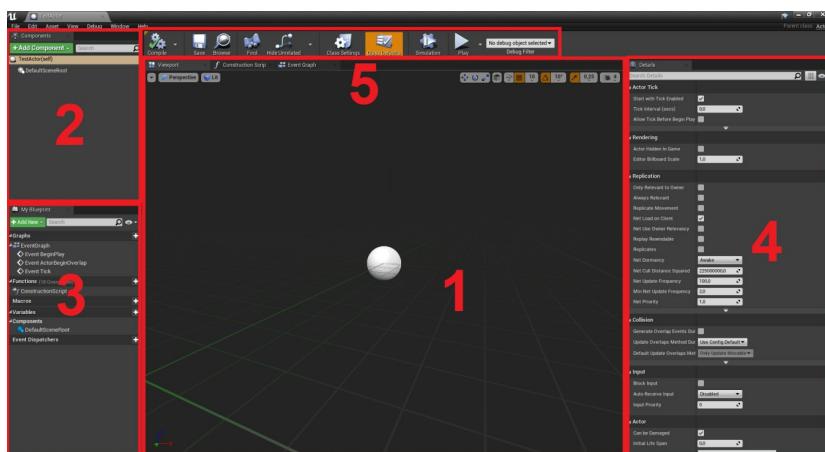


Figure 1.20: The Blueprint editor window is broken down into five parts

1. **Viewport**: Front and center in the editor you have the **Viewport** window. This window, similar to the **Level Viewport** window that we already learned about, will allow you to visualize your Actor and edit its components. Every actor can have several Actor Components, some of which have a visual representation, such as Mesh Components and Collision Components. We'll be talking about Actor Components in more depth in later chapters.

Technically, this center window contains three tabs, only one of which is the **Viewport** window, but we'll be talking about the other important tab, the **Event Graph** after we tackle this editor's interface. The third tab is the **Construction Script** window, which we will not be covering in this book.

2. **Components**: At the top left of the editor, you have the **Components** window. As mentioned in the previous description, Actors can have several Actor Components, and this window is

the one that will allow you to add and remove those Actor Components in your Blueprint class, as well as to access Actor Components defined in the C++ classes it inherits from.

3. **My Blueprint:** At the bottom left of the editor, you have the **My Blueprint** window. This will allow you to browse, add, and remove variables and functions defined in both this Blueprint class and the C++ class it inherits from. Keep in mind that Blueprints have a special kind of function, called an **event**, which is used to represent an event that happened in the game. You should see three of them in this window: **BeginPlay**, **ActorBeginOverlap**, and **Tick**. We'll be talking about these in a few paragraphs.

4. **Details:** At the right of the editor, you have the **Details** window. Similar to the editor's **Details** window, this window will show you the properties of the currently selected Actor Component, function, variable, event, or any other individual element

of this Blueprint class. If you currently have no elements selected, this window will be empty.

5. **Toolbar:** At the top center of the editor you have the **Toolbar** window. This window will allow you to compile the code you wrote in this Blueprint class, save it, locate it in **Content Browser**, and access this class's settings, among other things.

You can see the parent class of a Blueprint class by looking at the top-right corner of the Blueprint editor. If you click the name of the parent class, you'll be taken to either the corresponding Blueprint class, through the Unreal Engine editor, or the C++ class, through Visual Studio.

Additionally, you can change a Blueprint class's parent class by clicking on the **File** tab at the top left of the Blueprint editor and selecting the **Reparent Blueprint** option, which will allow you to specify the new parent class of this Blueprint class.

Now that we've learned about the basics of the Blueprint editor, let's take a look at its Event Graph.

Event Graph

The **Event Graph** window is where you'll be writing all of your Blueprint visual scripting code, creating your variables and functions, and accessing other variables and functions

declared in this class's parent class.

If you select the **Event Graph** tab, which you should be able to see to the right of the **Viewport** tab, you will be shown the **Event Graph** window instead of the **Viewport** window. On clicking the **Event Graph** tab, you will have the following window:

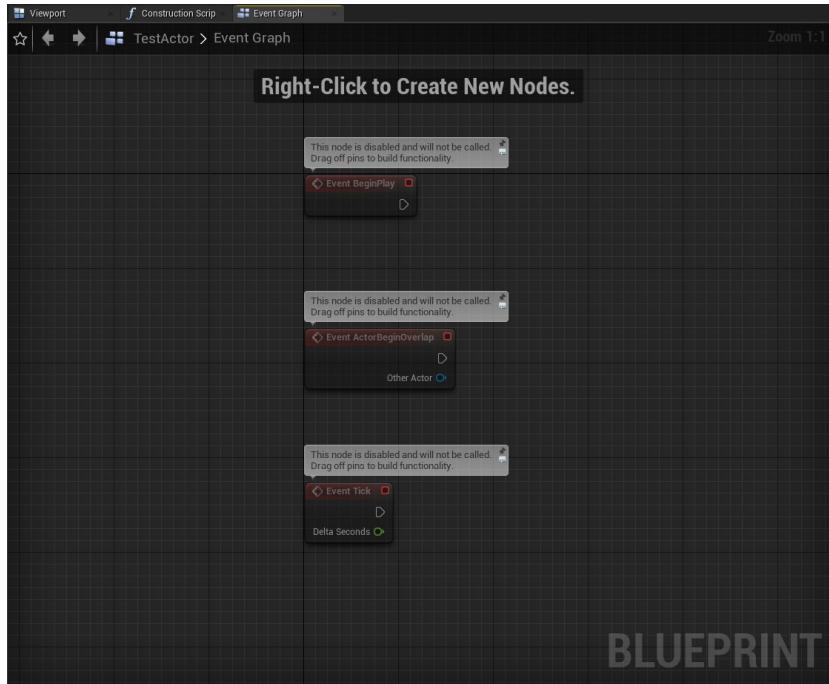


Figure 1.21: The Event Graph window, showing three disabled events

You can navigate the **Event Graph** by holding the *right mouse button* and dragging inside the graph, you can zoom in and out by scrolling the *mouse wheel*, and you can select nodes from the graph by either clicking with the *left mouse button* or by pressing and holding to select an area of nodes.

You can also *right-click* inside the **Event Graph** window to access the Blueprint's Actions menu, which allows you to access the actions you can do in the **Event Graph**, including getting and setting variables, calling functions or events, and many others.

The way scripting works in Blueprint is by connecting nodes using pins. There are several types of nodes, such as variables, functions, and events. You can connect these nodes through pins, of which there are two types:

1. **Execution pins:** These will dictate the order in which the nodes will be executed. If you want node 1 to be executed and then node 2 to be executed, you link the output execution pin of node 1 to the input execution pin of node 2, as shown in the following screenshot:

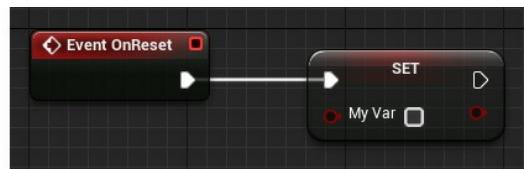


Figure 1.22: The output execution pin of the Event OnReset node being connected to the input execution pin of the setter node for MyVar

2. **Variable pins:** These work as parameters (also known as input pins), at the left of the node, and return values (also known as output pins), at the right side of the node, representing a value of a certain type (integer, float, Boolean, and others):



Figure 1.23: The Get Scalar Parameter Value function call node, which has two input variable pins and one output variable pin

Let's understand this better through the next exercise.

Exercise 1.04: Creating Blueprint Variables

In this exercise, we will see how to create Blueprint variables by creating a new variable of the **Boolean** type.

In Blueprint, variables work similarly to the ones you would use in C++. You can create them, get their value, and set them.

The following steps will help you complete this exercise:

1. To create a new Blueprint variable, head to the **My Blueprint** window and click the **+ Variable** button:

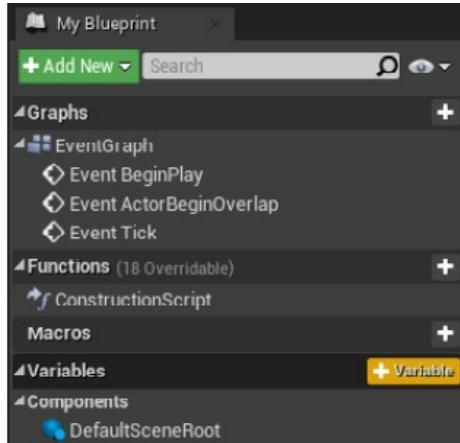


Figure 1.24: The + Variable button being highlighted in the My Blueprint window, which allows you to create a new Blueprint variable

2. After that, you'll automatically be allowed to name your new variable.

Name this new variable **MyVar**:



Figure 1.25: Naming the new variable MyVar

3. Compile your Blueprint by clicking the **Compile** button on the left side of the **Toolbar** window. If you now take a look at the **Details** window, you should see the following:

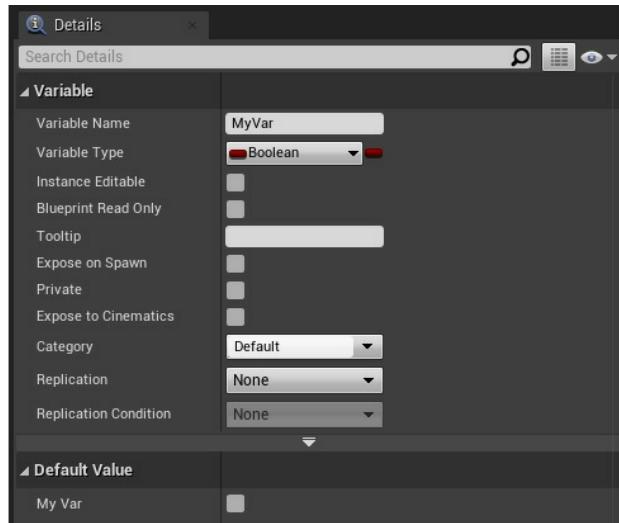


Figure 1.26: The MyVar variable settings in the Details window

4. Here, you'll be able to edit all the settings related to this variable, the most important ones being **Variable Name**, **Variable Type**, and its **Default Value** at the end of the settings. Boolean variables can have their value changed by clicking the gray box to their right:

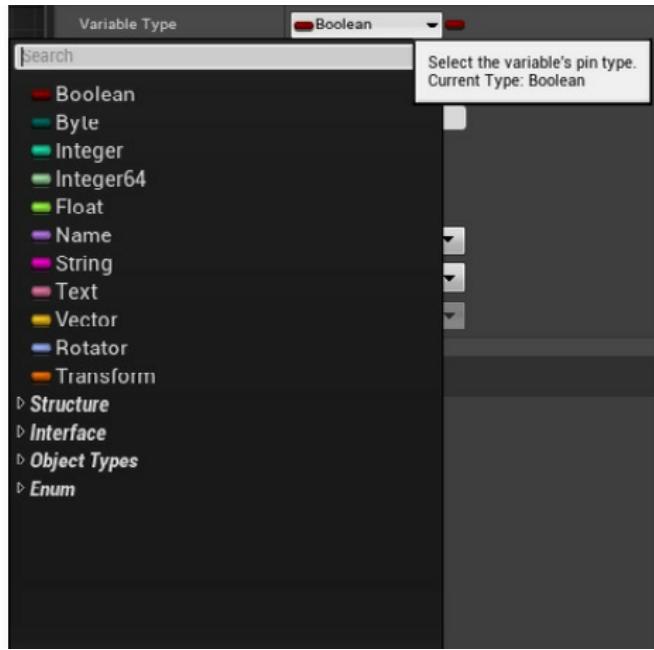


Figure 1.27: The variable types available from the Variable Type drop-down menu

5. You can also drag a getter or setter for a variable inside the **My Blueprint** tab into the **Event Graph** window:



Figure 1.28: Dragging the MyVar into the Event Graph window and choosing whether to add a getter or setter

Getters are nodes that contain the current value of a variable while setters

are nodes that allow you to change the value of a variable.

6. To allow a variable to be editable in each of the instances of this Blueprint class, you can click the eye icon to the right of that variable inside the **My Blueprint** window:

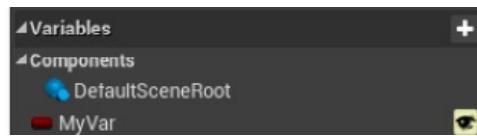


Figure 1.29: Clicking the eye icon to expose a variable and allow it to be instance-editable

7. You can then drag an instance of this class to your level, select that instance, and see the option to change that variable's value in the **Details** window of the editor:

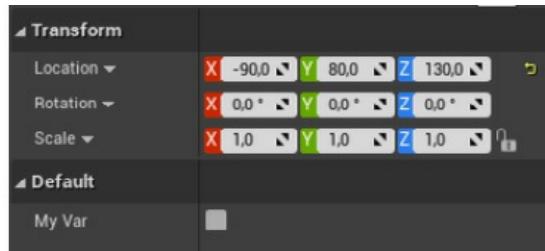


Figure 1.30: The exposed MyVar variable that can be edited through the Details panel of that object

And with that, we conclude this exercise and now know how to create our own Blueprint variables. Let's now take a look at

how to create Blueprint Functions in the next exercise.

Exercise 1.05: Creating Blueprint Functions

In this exercise, we will create our first Blueprint Function. In Blueprint, functions and events are relatively similar, the only difference being that an event will only have an output pin, usually because it gets called from outside of the Blueprint class:

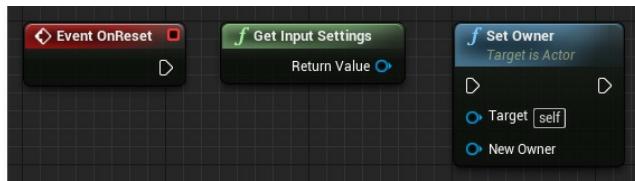


Figure 1.31: An event (left), a pure function call that doesn't need execution pins (middle), and a normal function call (right)

The following steps will help you complete this exercise:

1. Click the **+ Function** button inside the **My Blueprint** window:

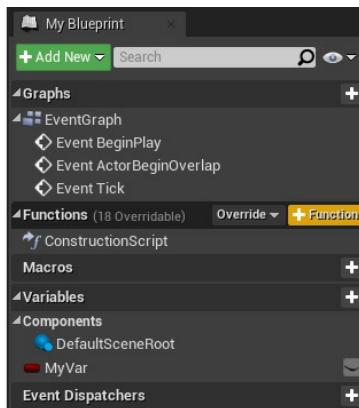


Figure 1.32: The + Function button being hovered over, which will create a new function

2. Name the new function **MyFunc**.
3. Compile your Blueprint by clicking the **Compile** button in the **Toolbar** window:

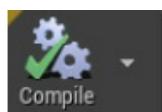


Figure 1.33: The Compile button

4. If you now take a look at the **Details** window, you should see the following:

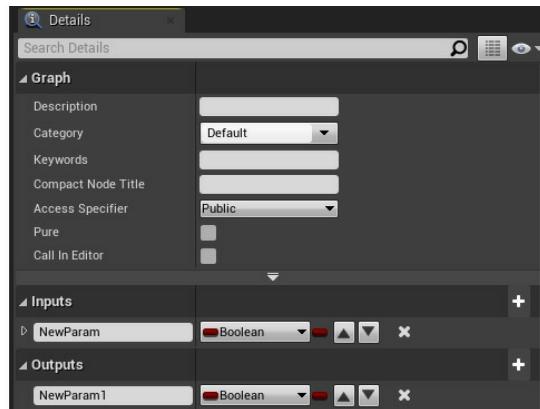


Figure 1.34: The Details panel after selecting the MyFunc function and adding an input and output pin

Here, you'll be able to edit all the

settings related to this function, the most important ones being **Inputs** and **Outputs** at the end of the settings. These will allow you to specify the variables that this function must receive and will return.

Lastly, you can edit what this function does by *clicking* it from the **My Blueprint** window. This will open a new tab in the center window that will allow you to specify what this function will do. In this case, this function will simply return **false** every time it is called:



Figure 1.35: The contents of the MyFunc function, receiving a Boolean parameter, and returning a Boolean type

5. To save the modifications we made to this Blueprint class, click the **Save** button next to the **Compile** button on the toolbar. Alternatively, you can have it so that the Blueprint automatically saves every time you compile it

successfully by selecting that option.

After following these steps, you now know how to create your own Blueprint Functions. Let's now take a look at a Blueprint node we'll be making use of later in this chapter.

The Float Multiplication Node

Blueprint contains many more nodes that are not related to variables or functions. One such example is arithmetic nodes (that is adding, subtracting, multiplying, and so on.). If you search for **float * float** on the Blueprint Actions menu, you'll find the *Float Multiplication* node:

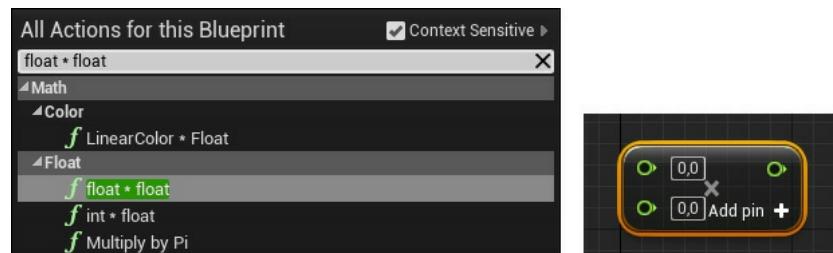


Figure 1.36: The Float Multiplication node

This node allows you to input two or more float parameters (you can add more by clicking the **+ icon** to the right of the **Add pin** text) and output the result of the multiplication of all of them. We will be using this node later, in this chapter's activity.

BeginPlay and Tick

Let's now take a look at two of the most important events in UE4: **BeginPlay** and **Tick**.

As mentioned previously, events will usually be called from outside the Blueprint class. In the case of the **BeginPlay** event, this event gets called either when an instance of this Blueprint class is placed in the level and the level starts being played, or when an instance of this Blueprint class is spawned dynamically while the game is being played. You can think of the **BeginPlay** event as the first event that will be called on an instance of this Blueprint, which you can use for initialization.

The other important event to know about in UE4 is the **Tick** event. As you may know, games run at a certain frame rate, the most frequent being either 30 FPS (frames per second) or 60 FPS: this means that the game will render an updated image of the game 30 or 60 times every second. The **Tick** event will get called every time the game does this, which means that if the game is running at 30 FPS, the **Tick** event will get called 30 times every second.

Go to your Blueprint class's **Event Graph** window and delete the three grayed-out events by selecting all of them and clicking the **Delete** key, which should cause the **Event Graph** window to become empty. After that, *right-click* inside the **Event Graph** window, type in **BeginPlay**, and select the **Event BeginPlay** node by either clicking the **Enter** key or by clicking on that option in the Blueprint Actions menu. This should cause that event to be added to the **Event Graph** window:

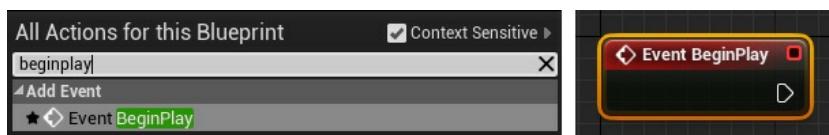


Figure 1.37: The BeginPlay event being added to the Event Graph window through the Blueprint Actions menu

Right-click inside the **Event Graph** window, type in **Tick**, and select the **Event Tick** node. This should cause that event to be added to the **Event Graph** window:

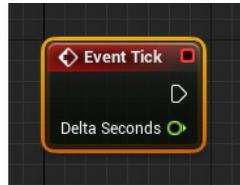


Figure 1.38: The Tick event

Unlike the **BeginPlay** event, the **Tick** event will be called with a parameter, **DeltaTime**. This parameter is a float that indicates the amount of time that passed since the last frame was rendered. If your game is running at 30 FPS, this means that the interval between each of the frames being rendered (the delta time) is going to be, on average, $1/30$ seconds, which is around 0.033 seconds (33.33 milliseconds). If frame 1 is rendered and then frame 2 is rendered 0.2 seconds after that, then frame 2's delta time will be 0.2 seconds. If frame 3 gets rendered 0.1 seconds after frame 2, frame 3's delta time will be 0.1 seconds, and so forth.

But why is the **DeltaTime** parameter so important? Let's take a look at the following scenario: you have a Blueprint class that increases its position on the Z axis by 1 unit every time a frame is rendered using the **Tick** event. However, you are faced with a problem: there's the possibility that players will run your game at different frame rates, such as 30 FPS and 60 FPS. The player who's running the game at 60 FPS will cause the **Tick** event to be called twice as much as the player who's running the game at 30 FPS, and the Blueprint class will end up moving twice as fast because of that. This is where the delta time comes into play: because the game that's running at 60 FPS will have the **Tick** event called with a lower delta time value (the interval between the frames being rendered is much smaller), you can use that value to change

the position on the *Z* axis. Although the **Tick** event is being called twice as much on the game running at 60 FPS, its delta time is half the value, so it all balances out. This will cause two players playing the game with different frame rates to have the same result.

Note

*If you want a Blueprint that is using the delta time to move, you can make it move faster or slower by multiplying the delta time by the number of units you want it to move per second (for example, if you want a Blueprint to move 3 units per second on the *Z* axis, you can tell it to move **3 * DeltaTime** units every frame).*

Let's now try another exercise, which will consist of working with Blueprint nodes and pins.

Exercise 1.06: Offsetting the TestActor Class on the Z Axis

In this exercise, you'll be using the **BeginPlay** event to offset (move) the **TestActor** on the *Z* axis when the game starts being played.

The following steps will help you complete this exercise:

1. Open the **TestActor** Blueprint class.
2. Using the **Blueprint Actions** menu, add the **Event BeginPlay**

node to the graph, if it's not already there.

3. Add the **AddActorWorldOffset** function and connect the **BeginPlay** event's output execution pin to this function's input execution pin. This function is responsible for moving an Actor in the intended axes (*X*, *Y*, and *Z*) and it receives the following parameters:
 1. **Target**: The Actor that this function should be called on, which will be the Actor calling this function. The default behavior is to call this function on the Actor calling this function, which is exactly what we want and is shown using the **self** property.
 2. **DeltaLocation**: The amount that we want to offset this Actor by in each of the three axes: *X*, *Y*, and *Z*.
 3. We won't be getting into the other two parameters, **Sweep** and **Teleport**, so you can

leave them as is. They are both Boolean types and should be left as **false**:

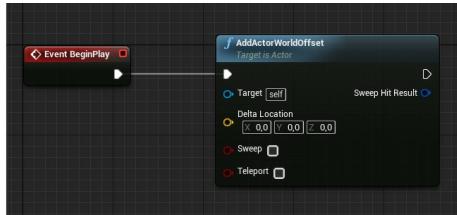


Figure 1.39: The BeginPlay event calling the AddActorWorldOffset function

4. Split the **Delta Location** input pin, which will cause this **Vector** property to be split into three float properties. You can do this to any variable type that is comprised of one or more subtypes (you wouldn't be able to do this to the float type because it's not comprised of any variable subtypes) by *right-clicking* on them and selecting **Split Struct Pin**:

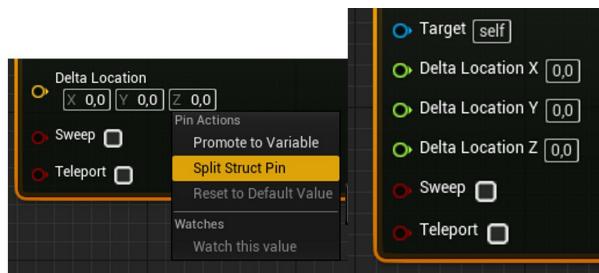


Figure 1.40: The Delta Location parameter being split from a vector into three floats

5. Set the **Z** property of **Delta Location** to **100** units by clicking with the *left mouse button*, typing that number, and then pressing the *Enter* key. This will cause our **TestActor** to move up on the *Z* axis by **100** units when the game starts.
6. Add a cube shape to your **TestActor**, using the **Components** window, so that we can see our Actor. You can do this by clicking the **+ Add Component** button, typing **Cube**, and then selecting the first option under the **Basic Shapes** section:

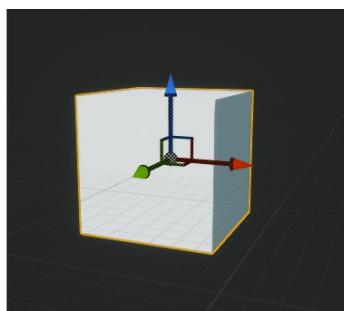


Figure 1.41: Adding a cube shape

7. Compile and save your Blueprint class by clicking the **Compile** button.
8. Go back to the level's **Viewport** window and place an instance of your **TestActor** Blueprint class inside the

level, if you haven't done so already:



Figure 1.42: Adding an instance of TestActor to the level

9. When you play the level, you should notice that the **TestActor** we added to the level is in a more elevated position:

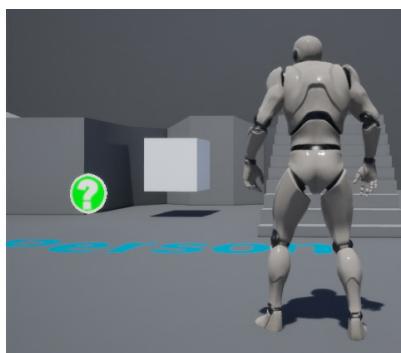


Figure 1.43: The TestActor increasing its position on the Z axis when the game starts

10. After making these modifications, save the changes made to our level by either pressing *Ctrl + S* or by clicking the **Save Current** button on the

editor **Toolbar**.

In this exercise, you've learned how to create your first Actor Blueprint class with your own Blueprint scripting logic.

Note

*Both the **TestActor** blueprint asset and the **Map** asset with the final result of this exercise can be found here:
<https://packt.live/3lfYOa9>.*

Now that we've done this, let's learn a bit more about the **ThirdPersonCharacter** Blueprint class.

ThirdPersonCharacter Blueprint Class

Let's take a look at the **ThirdPersonCharacter** Blueprint class, which is the Blueprint representing the character that the player controls, and take a look at the Actor Components that it contains.

Go to the **ThirdPersonCPP -> Blueprints** directory inside **Content Browser** and open the **ThirdPersonCharacter** asset:

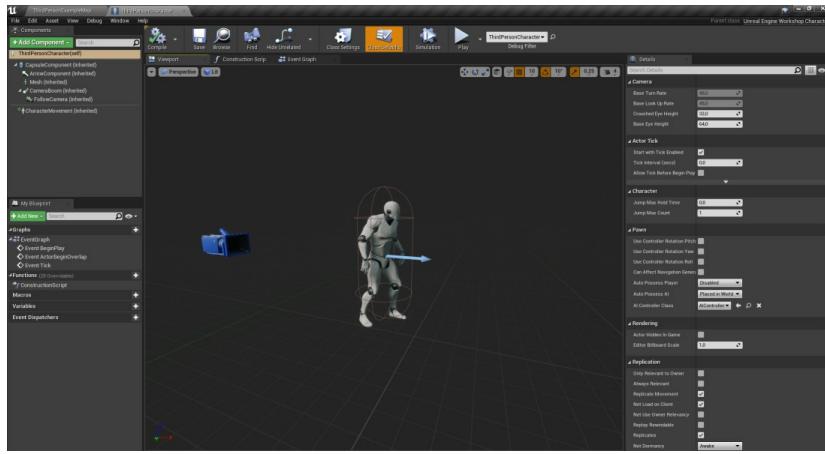


Figure 1.44: The ThirdPersonCharacter Blueprint class

In a previous section, where we introduced the **Components** window inside the Blueprint editor, we mentioned **Actor Components**.

Actor Components are entities that must live inside an Actor and allow you to spread the logic of your Actor into several different Actor Components. In this Blueprint, we can see that there are four visually represented Actor Components:

- A Skeletal Mesh Component, which shows the UE4 mannequin
- A Camera Component, which shows where the player will be able to see the game from
- An Arrow Component, which allows us to see where the character is facing (this is mainly used for development purposes, not while the game is being played)

- A Capsule Component, which specifies the collision range of this character

If you look at the **Components** window, you'll see a few more Actor Components than the ones we see in the **Viewport** window. This is because some Actor Components don't have a visual representation and are purely made up of C++ or Blueprint code. We'll be going into more depth on Actor Components in the next chapter and *Chapter 9, Audio-Visual Elements*.

If you take a look at this Blueprint class's **Event Graph** window, you'll see that it's essentially empty, like the one we saw with our **TestActor** Blueprint class, despite it having a bit of logic associated with it. That is because that logic is defined in the C++ class and not in this Blueprint class. We'll be taking a look at how to do this in the next chapter.

In order to explain this Blueprint class's Skeletal Mesh Component, we should first talk about meshes and materials.

Meshes and Materials

For a computer to visually represent a 3D object, it needs two things: a 3D mesh and a material.

Meshes

3D meshes allow you to specify the size and shape of an object, like this mesh representing a monkey's head:

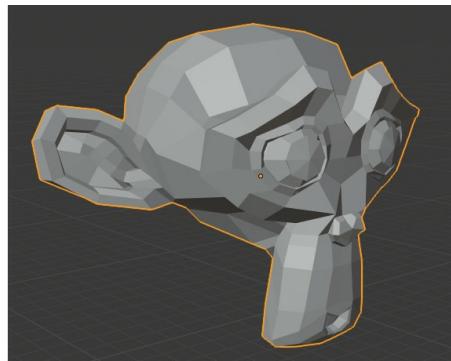


Figure 1.45: A 3D mesh of a monkey's head

Meshes are comprised of several vertices, edges, and faces. Vertices are simply a 3D coordinate with an X , Y , and Z position; an edge is a connection (that is, a line) between two vertices; and a face is a connection of three or more edges. You can see in the previous figure the individual vertices, edges, and faces of the mesh, where each face is colored between white and black, depending on how much light is reflecting off the face. Nowadays, video games can render meshes with thousands of vertices in such a way that you can't tell the individual vertices apart because there are so many of them so close together.

Materials

Materials, on the other hand, allow you to specify how a mesh is going to be represented. They allow you to specify a mesh's color, draw a texture on its surface, or even manipulate its individual vertices.

Creating meshes is something that, as of the time of writing this book, is not properly supported inside UE4 and should be done in another piece of software, such as Blender or Autodesk Maya, so we won't be going into this in great detail here. We will, however, learn how to create materials for existing meshes.

In UE4, you can add meshes through Mesh Components, which inherit from the Actor Component class. There are several types of Mesh Components, but the two most important ones are Static Mesh Components, for meshes that don't have animations (for example, cubes, static level geometry), and Skeletal Mesh Components, for meshes that have animations (for example, character meshes that play movement animations). As we saw earlier, the **ThirdPersonCharacter** Blueprint class contains a Skeletal Mesh Component because it's used to represent a character mesh that plays movement animations. In the next chapter, we'll be learning how to import assets such as meshes into our UE4 project.

Let's now take a look at materials in UE4 in the next exercise.

Manipulating Materials in UE4

In this section, we'll be taking a look at how materials work in UE4.

Go back to your **Level Viewport** window and select this **Cube** object:

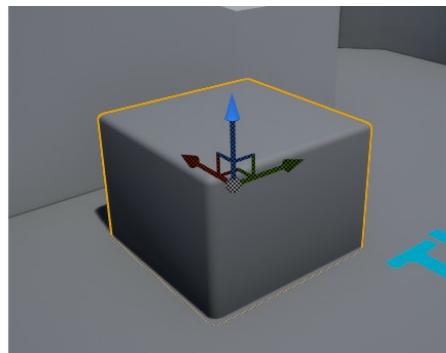


Figure 1.46: The Cube object, next to the text saying Third Person on the floor

Take a look at the **Details** window, where you'll be able to see both the mesh and material associated with this object's **Static Mesh** component:

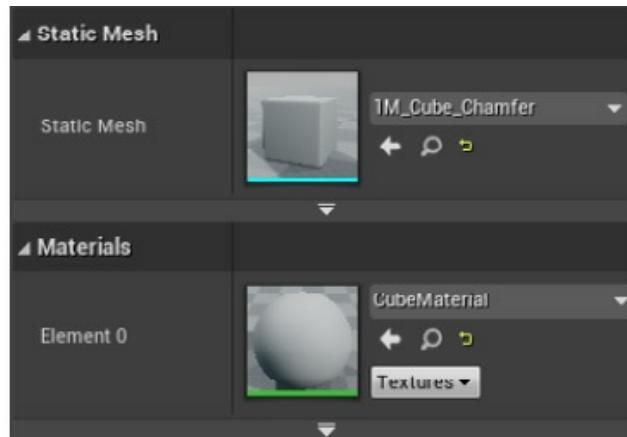


Figure 1.47: The Static Mesh and Materials (Element 0) properties of the Cube object's Static Mesh component

Note

Keep in mind that meshes can have more than one material, but must have at least one.

Click the *looking glass* icon next to the **Material** property to be taken to that material's location in **Content Browser**.

This icon works with any reference to any asset inside the editor, so you can do the same thing with the asset referenced as the cube object's **Static Mesh**:



Figure 1.48: The looking glass icon (left), which takes you to that asset's location in Content Browser (right)

*Double-click that asset with the left mouse button to open that asset in the **Material** editor. Let's break down the windows present in **Material editor**:*

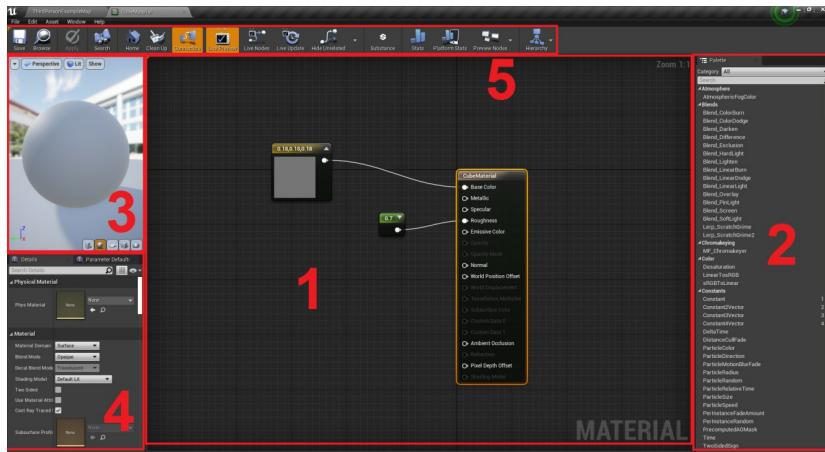


Figure 1.49: The Material editor window broken down into five parts

- 1. Graph:** Front and center in the editor, you have the **Graph** window. Similar to the Blueprint editor's **Event Graph** window, the **Material** editor's graph is also node-based, where you'll also find nodes connected by pins, although here you won't find execution pins, only input and output pins.
- 2. Palette:** At the right edge of the screen, you'll see the **Palette** window, where you'll be able to search all the nodes that you can add to the **Graph** window. You can also do this the same way as in the Blueprint

editor's **Event Graph** window by *right-clicking* inside the **Graph** window and typing the node you wish to add.

3. **Viewport:** At the top-left corner of the screen, you'll see the **Viewport** window. Here, you'll be able to preview the result of your material and how it will appear on some basic shapes such as spheres, cubes, and planes.
4. **Details:** At the bottom-left corner of the screen, you'll see the **Details** window where, similar to the Blueprint editor, you'll be able to see the details of either this **Material** asset or those of the currently selected node in the **Graph** window.
5. **Toolbar:** At the top edge of the screen, you'll see the **Toolbar** window, where you'll be able to apply and save the changes made to your material, as well as to perform several actions related to the **Graph** window.

In every single Material editor inside UE4, you'll find a node with the name of that **Material** asset, where you'll be able to

specify several parameters related to it by plugging that node's pins to other nodes.

In this case, you can see that there's a node called **0.7** being plugged into the **Roughness** pin. This node is a **Constant** node, which allows you to specify a number associated with it – in this case, **0.7**. You can create constant nodes of a single number, a 2 vector (for example, **(1, 0.5)**), a 3 vector (for example, **(1, 0.5, 4)**), and a 4 vector (for example, **(1, 0.5, 4, 0)**). To create these nodes, you can press the **Graph** window with the *left mouse button* while holding the **1, 2, 3**, or **4** number keys, respectively.

Materials have several input parameters, so let's go through some of the most important ones:

- **BaseColor:** This parameter is simply the color of the material. Generally, constants or texture samples are used to connect to this pin, to either have an object be a certain color or to map to a certain texture.
- **Metallic:** This parameter will dictate how much your object will look like a metal surface. You can do this by connecting a constant single number node that ranges from **0** (not metallic) to **1** (very metallic).
- **Specular:** This parameter will dictate how much your object will reflect light. You can do this by connecting a

constant single number node that ranges from 0 (doesn't reflect any light) to 1 (reflects all the light). If your object is already very metallic, you will see little to no difference.

- **Roughness:** This parameter will dictate how much the light that your object reflects will be scattered (the more the light scatters, the less clear this object will reflect what's around it). You can do this by connecting a constant single number node that ranges from 0 (the object essentially becomes a mirror) to 1 (the reflection on this object is blurry and unclear).

Note

*To learn more about **material** inputs like the ones above, go to <https://docs.unrealengine.com/en-US/Engine/Rendering/Materials/MaterialInputs>.*

UE4 also allows you to import images (.jpeg, .png) as **Texture** assets, which can then be referenced in a material using **Texture Sample** nodes:

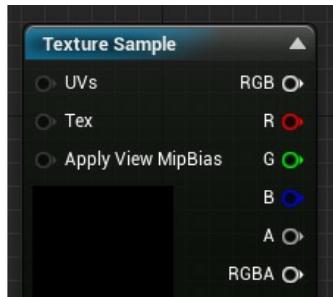


Figure 1.50: The Texture Sample node, which allows you to specify a texture and use it or its individual color channels as pins

Note

We will be taking a look at how to import files into UE4 in the next chapter.

In order to create a new **Material** asset, you can do so by right-clicking on the directory inside **Content Browser** where you want to create the new asset, which will allow you to choose which asset to create, and then select **Material**.

Now you know how to create and manipulate materials in UE4.

Let's now jump into this chapter's activity, which will be the first activity of this book.

Activity 1.01: Propelling TestActor on the Z Axis Indefinitely

In this activity, you will use the **Tick** event of **TestActor** to move it on the Z axis indefinitely, instead of doing this only

once when the game starts.

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.
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*.
4. Add a *Float Multiplication* node to **Event Graph** window.
5. Connect the **Tick** event's **Delta Seconds** output pin to the first input pin of the *Float Multiplication* node.
6. Create a new variable of the **float** type, call it **VerticalSpeed**, and set its default value to **25**.
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.

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:

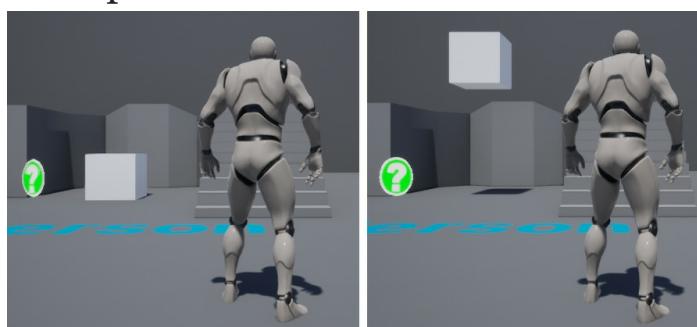


Figure 1.51: The TestActor propelling itself vertically

And with those steps completed, we conclude this activity – the first of many in this book. We've now consolidated adding and removing nodes to and from the Blueprint editor's **Event Graph** window, as well as using the **Tick** event and its **DeltaSeconds** property to create game logic that maintains consistency across different frame rates.

Note

The solution to this activity can be found at:

<https://packt.live/338jEBx>.

The **TestActor** blueprint asset can be found here:

<https://packt.live/2U8pAVZ>.

Summary

By completing this chapter, you have taken the first step in your game development journey by learning about Unreal Engine 4. You now know how to navigate the Unreal Engine editor, manipulate the Actors inside a level, create your own Actors, use the Blueprint scripting language, and how 3D objects are represented in Unreal Engine 4.

Hopefully, you realize that there's a whole world of possibilities ahead of you and that the sky is the limit in terms of the things you can create using this game development tool.

In the next chapter, you will recreate the project template that was automatically generated in this chapter from scratch. You will learn how to create your own C++ classes and then create Blueprint classes that can manipulate properties declared in their parent class and how to import character meshes and animations into Unreal Engine 4, as well as becoming familiar with other animation-related assets such as *Animation Blueprints*.

2. Working with Unreal Engine

Overview

This chapter will focus on many of the basic concepts and features within Unreal Engine. You will be shown how to create a C++ project, how to perform some basic debugging, and how to work with character-specific animations.

By the end of this chapter, you'll be able to create C++ template projects, be able to debug code within Visual Studio, understand the folder structure and the best practices involved, and finally, be able to set up character animations based on their states.

Introduction

In the previous chapter, we went through the basics of the Epic Games Launcher, along with Unreal Editor fundamentals. We saw how to work with Objects and what Blueprints are on a basic level, in addition to exploring the First Person Template. In this chapter, we'll be building upon those fundamentals by exploring the Third Person Template and working with Input and Animations.

Game development can be done in a wide variety of languages, such as C, C++, Java, C#, and even Python. While each language has pros and cons, we will be using C++ throughout this book as it is the primary programming

language used within the Unreal Engine.

In this chapter, we will get you up to speed on how to create a C++ project and basic level debugging in UE4. It is very important to be able to debug code as it helps the developer while dealing with bugs. The tools provided come in very handy and are essential for any Unreal Engine developer.

Following this, we will get up close and personal with the core classes involved in creating games and experiences in Unreal Engine. You will explore Game Mode and the relevant class concepts, followed by an exercise to gain a hands-on understanding of this.

The final section in this chapter is all about animations. Almost every single game features animations, some to a very basic extent, but some to a very high level that includes captivating details which are key to the gameplaying experience. Unreal Engine offers several tools you can use to create and deal with animations, including the Animation Blueprint, which has complex graphs and a State Machine.

Creating and Setting Up a Blank C++ Project

At the start of every project, you might want to start with any of the templates provided by Epic (which contain ready-to-execute basic code) and build on top of that. Most/some of the time, you might need to set up a blank or an empty project that you can mold and sculpt to your requirements. We'll learn how to do that in the following exercise.

Exercise 2.01: Creating

an Empty C++ Project

In this exercise, you will learn how to create an empty C++ project from the template provided by Epic. This will serve as the foundation for many of your future C++ projects.

The following steps will help you complete this exercise:

1. Launch Unreal Engine 4.24 from the Epic Games Launcher.
2. Click on the **Games** section and click **Next**.
3. Make sure the **Blank** project template is selected and click **Next**.
4. Click the **Blueprint** section dropdown and select **C++**.

Note

Make sure the project folder and project name are specified with an appropriate directory and name, respectively.

When everything is set up, click on the **Create Project** button. In this case, our project directory is inside a folder called **UnrealProjects**, which is

inside the **E** drive. The project name is set to **MyBlankProj** (it is recommended that you follow these names and project directories, but you can use your own if you wish to do so).

Note

The project name cannot have any spaces in it. It is preferable to have an Unreal directory as close to the root of a drive as possible (to avoid running into issues such as the 256-character path limit when creating or importing assets into your project's working directory; for small projects, it may be fine, but for more large-scale projects, where the folder hierarchy may become too complex, this step is important).

You will notice that after it's done generating code and creating the project files, the project will be opened, along with its Visual Studio solution (**.sln**) file.

Note

Make sure the Visual Studio solution configuration is set to Development Editor and that the solution platform is set to Win64 for Desktop development:

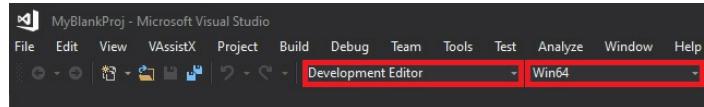


Figure 2.1: Visual Studio deployment settings

By completing this exercise, we now know how to create an empty C++ project on UE4, along with its considerations.

In the next section, we'll be talking a bit about the folder structure, along with the most basic and most used folder structure format that's used by Unreal developers.

Content Folder Structure in Unreal Engine

In your project directory (`E:/UnrealProjects/MyBlankProj` in our case), you will observe a **Content** folder. This is the primary folder that your project uses for different types of assets and project-relevant data (including Blueprints). The C++ code goes into the **Source** folder in your project. Please note that the best practice is to create new C++ code files directly through the Unreal Editor as this simplifies the process and results in fewer errors.

There are many different strategies you can use to organize the data inside your **Content** folder. The most basic and easy-to-understand is using folder names to depict the type of

content inside. Therefore, a **Content** folder directory structure may resemble the example at <https://packt.live/3lCVFkR>. In this example, you can see that each file is categorically placed under the name of the folder representing its type on the first level, with the following levels further grouping it into meaningful folders.

Note

*All blueprints should prefix **BP** in their name (to differentiate them from the default blueprints used by Unreal Engine). The rest of the prefixes are optional (however, it is best practice to format them with the prefixes shown earlier).*

In the next section, we will be looking at the Visual Studio solution.

Working with the Visual Studio Solution

Every C++ project in Unreal Engine has a Visual Studio solution. This, in turn, drives all the code and provides developers with the ability to set up execution logic and debug code in its running state.

SOLUTION ANALYSIS

The Visual Studio solution (**.sln**) file that's produced inside the project directory contains the entire project and any associated code that's been added to it.

Let's have a look at the files present in Visual Studio. *Double-click* the **.sln** file to open it within Visual Studio.

In **Solution Explorer**, you will see two projects called **Engine** and **Games**.

THE ENGINE PROJECT

At the base level, Unreal Engine itself is a Visual Studio project and has its own solution file. This contains all the code and third-party integrations that work together in Unreal Engine. All the code within this project is called the "source" code.

The Engine project consists of the external dependencies, configurations, plugins, shaders, and source code of Unreal Engine that are currently being used for this project. You can, at any time, browse the **UE4 -> Source** folder to view any of the engine code.

Note

As Unreal Engine is open source, Epic allows developers to both view and edit source code to suit their needs and requirements. However, you cannot edit the source code in the version of Unreal Engine that's installed via the Epic Games Launcher. To be able to make and build changes in source code, you need to download the source version of Unreal Engine, which can be found via GitHub. You can use the following guide to download the Source Version of the Unreal Engine: <https://docs.unrealengine.com/en-US/GettingStarted/DownloadingUnrealEngine/index.html>

*After downloading, you can also refer to the following guide for compiling/building the newly downloaded engine:
<https://docs.unrealengine.com/en-US/Programming/Development/BuildingUnrealEngine/index.html>*

GAME PROJECT

Under the **Games** directory is the solution folder with the name of your project. Upon expansion, you'll find a set of folders. You will be concerned with the following:

- **Config Folder:** Carries all the configurations that have been set up for the project and the build (these can optionally have platform-specific (such as Windows, Android, iOS, Xbox, or PS) settings as well).
- **Plugins Folder:** This is an optional folder that's created when you add any third-party plugin (downloaded from the Epic Marketplace or obtained through the internet). This folder will contain all of the source code of the plugins associated with this project.
- **Source Folder:** This is the primary folder we're going to be working with. It will contain the Build Target files, as well as all the source code for the project. The following is a description of the default files in the source folder:
- **Target and Build Files:** These (as shown in the following screenshot) contain code that specifies the Unreal

Build Tool (*the program that builds your game*) that you will use to build your game. It contains any extra modules that need to be added to the game, as well as other build-related settings. By default, there are two target files (one for Unreal Editor and another for the build as depicted by their names), which end with the **.Target.cs** extension, and one build file that ends with **Build.cs**.

- **ProjectName code files (.cpp & .h)**: By default, these files are created for each project and contain the code that's used to run the default game module code.
- **ProjectNameGameModeBase code files (.cpp & .h)**: By default, an empty Project Game Mode Base is created. It's not usually used in most cases.
- **ProjectName.uproject file**: Contains the descriptors used to provide basic information about the project and the list of plugins associated with it.

Debugging Code in Visual Studio

Visual Studio provides powerful debugging features with the help of breakpoints in code. It enables users to pause the game at a particular line of code so that the developer can see the current values of variables and step through the code and game in a controlled fashion (can proceed line by line, function by function, or so on).

This is useful when you have a lot of variables and code files in your game project, and you want to see the values of the variables being updated and used in a step-by-step fashion to debug the code, find out what issues there are, and solve them. Debugging is a fundamental process of any developer's work, and only after many continuous debugging, profiling, and optimization cycles does a project get polished enough for deployment.

Now that you've got the basic idea of the Visual Studio solution, we'll move on and cover a practical exercise on it.

Exercise 2.02: Debugging the Third Person Template Code

In this exercise, you'll be creating a project using the Third Person Template of Unreal Engine and will debug the code from within Visual Studio. We'll be investigating the value of a variable called **BaseTurnRate** in the **Character** class of this template project. We'll see how the value updates as we move through the code, line by line.

The following steps will help you complete this exercise:

1. Launch Unreal Engine from the Epic Games Launcher.
2. Click on the **Games** section and click **Next**.
3. Select **Third Person** and click **Next**.
4. Select C++, set the project name to **ThirdPersonDebug**, and click the **Create Project** button.
5. Now, close Unreal Editor, go to the Visual Studio solution, and open the **ThirdPersonDebugCharacter.cpp** file:

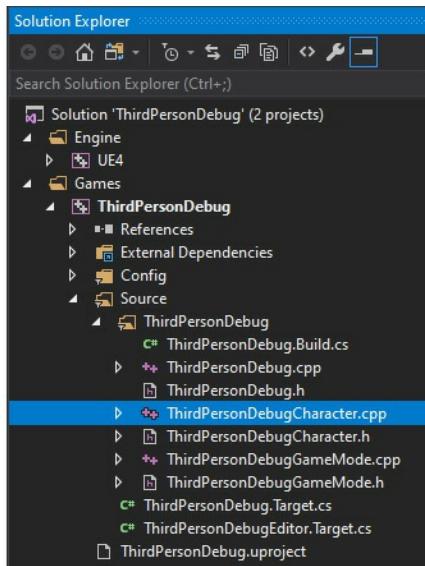
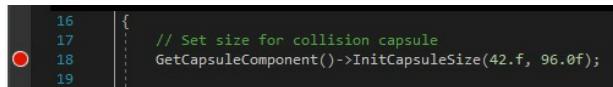


Figure 2.2:
ThirdPersonDebugCharacter.cpp

file location

6. *Left-click* on the bar on the left-hand side of line **18**. A red dot icon should appear on it (*you can toggle it off by clicking on it again*):



```
16
17
18     // Set size for collision capsule
19     GetCapsuleComponent()->InitCapsuleSize(42.f, 96.0f);
```

Figure 2.3: Collision capsule init code

Here, we are getting the **capsule** component (explained further in *Chapter 3, Character Class Components and Blueprint Setup*) of the character, which, by default, is the root component. Then, we are calling its **InitCapsuleSize** method, which takes in two parameters: the **InRadius** float and **InHalfHeight** float, respectively.

7. Make sure the solution configuration setting in VS is set to **Development Editor** and click on the **Local Windows Debugger** button:



Figure 2.4: Visual Studio build

settings

8. Wait until you're able to see the following window in the bottom-left corner:

Note

*If the window doesn't pop-up, you can open the window manually by opening **Autos** under **Debug > Windows > Autos**. Additionally, you may also use **locals**.*

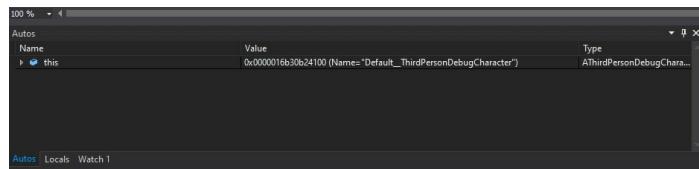


Figure 2.5: Visual Studio variable watch window

this shows the object itself. The object contains variables and methods that it stores, and by expanding it, we're able to see the state of the entire object and its variables at the current line of code execution.

9. Expand **this**, then **ACharacter**, and then **CapsuleComponent**. Here, you can see the values for the

CapsuleHalfHeight = 88.0 and
CapsuleRadius = 34.0 variables.

Next to line **18**, where the red dot initially was, you will see an arrow. This means that the code is at the end of line **17** and has not executed line **18** yet.

10. Click the **Step Into** button to go to the next line of code (*Shortcut: F11*). **Step Into** will move into code inside the function (if present) on the line. On the other hand, **Step Over** will just execute the current code and move to the next line. Since there is no function on the current line, **Step Into** will mimic the **Step Over** functionality.



Figure 2.6: Debug step into

11. Notice that the arrow has moved to line **21** and that the variables have been updated. **CapsuleHalfHeight = 96.0** and **CapsuleRadius = 42.0** are highlighted in red. Also, notice that the **BaseTurnRate** variable is initialized to **0.0**:

The screenshot shows the Unreal Engine 4 debugger's code editor and the 'Autos' panel. The code editor has a red stop sign icon at the top left. The current line of code is line 21: `BaseTurnRate = 45.f;`. The 'Autos' panel lists variables and their values:

Name	Type	Value
<code>BaseTurnRate</code>	float	45.000000
<code>this</code>	ACharacter	0x000016b30b24100 (Name: "Default_ThirdPersonDebugCharacter")
<code>APawn</code>	APawn	(Name: "Default_ThirdPersonDebugCharacter")
<code>Mesh</code>	USkeletalMeshComponent	0x000016b30e3040 (Name: "CharacterMesh0")
<code>CharacterMovement</code>	UCharacterMovement	0x000016b30b34b80 (Name: "CharMoveComp")
<code>CapsuleComponent</code>	UCapsuleComponent	0x000016b30acf200 (Name: "CollisionCylinder")
<code>UShapeComponent</code>	UShapeComponent	(Name: "CollisionCylinder")
<code>CapsuleHalfHeight</code>	float	95.000000
<code>CapsuleRadius</code>	float	42.000000
<code>CapsuleHeight_DEPRECATED</code>	float	0.00000000

Figure 2.7: BaseTurnRate initial value

12. Step in (*F11*) once again to go to line 22. Now, the **BaseTurnRate** variable has a value of **45 . 0** and **BaseLookUpRate** is initialized to **0 . 0**, as shown in the following screenshot:

The screenshot shows the Unreal Engine 4 debugger's code editor and the 'Autos' panel. The current line of code is line 21: `BaseTurnRate = 45.f;`. The 'Autos' panel lists variables and their values:

Name	Type	Value
<code>BaseLookUpRate</code>	float	0.00000000
<code>BaseTurnRate</code>	float	45.000000
<code>this</code>	ACharacter	0x000016b30b24100 (Name: "Default_ThirdPersonDebugCharacter")
<code>APawn</code>	APawn	(Name: "Default_ThirdPersonDebugCharacter")
<code>Mesh</code>	USkeletalMeshComponent	0x000016b30e3040 (Name: "CharacterMesh0")
<code>CharacterMovement</code>	UCharacterMovement	0x000016b30b34b80 (Name: "CharMoveComp")
<code>CapsuleComponent</code>	UCapsuleComponent	0x000016b30acf200 (Name: "CollisionCylinder")
<code>UShapeComponent</code>	UShapeComponent	(Name: "CollisionCylinder")
<code>CapsuleHalfHeight</code>	float	95.000000
<code>CapsuleRadius</code>	float	42.000000
<code>CapsuleHeight_DEPRECATED</code>	float	0.00000000

Figure 2.8: BaseTurnRate updated value

13. Step in (*F11*) once again to go to line
27. Now, the **BaseLookUpRate**
variable has a value of **45 . 0**.

Similarly, you are encouraged to step in and debug other sections of the code to not only familiarize yourself with the debugger but also to understand how the code works behind the scenes.

By completing this exercise, you've learned how to set up debug points in Visual Studio, as well as stop debugging at a point, and then continue line by line while watching an object and its variable's values. This is an important aspect for any developer, and many often use this tool to get rid of pesky bugs within code, especially when there's a lot of code flows and the number of variables is quite large.

Note

At any point, you can stop debugging, restart debugging, or continue with the rest of the code by using the following buttons on the top menu bar:

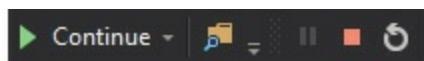


Figure 2.9: Debugging tools in Visual Studio

Now, we'll look at importing assets into an Unreal project.

Importing the Required Assets

Unreal Engine gives users the ability to import a wide range of

file types for users to customize their projects. There are several import options that developers can tweak and play around with to match their required settings.

Some common file types that game developers often import are FBX for scenes, meshes, animations (exported from Maya and other similar software), movie files, images (mostly for the user interface), textures, sounds, data in CSV files, and fonts. These files may be obtained from the Epic Marketplace or any other means (such as the internet) and used within the project.

Assets can be imported by dragging and dropping them into the **Content** folder, or by clicking the **Import** button in the **Content Browser**.

Now let's tackle an exercise where we'll learn how to import FBX files and see how this is done.

Exercise 2.03: Importing a Character FBX File

This exercise will focus on importing a 3D model from an FBX file. FBX files are widely used to export and import 3D models, along with their materials, animations, and textures.

The following steps will help you complete this exercise:

1. Download the **SK_Mannequin.FBX**, **ThirdPersonIdle.FBX**, **ThirdPersonRun.FBX** and **ThirdPersonWalk.FBX** files from the **Chapter02 -> Exercise2.03 ->**

ExerciseFiles directory, which can be found on GitHub.

Note

The **ExerciseFiles** directory can be found on GitHub at the following link:
<https://packt.live/2IiqTzq>.

2. Open the blank project we created in *Exercise 2.01, Creating an Empty C++ Project*.
3. In the **Content Browser** interface of the project, click **Import**:

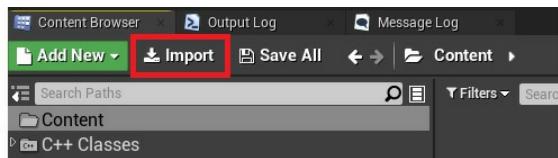


Figure 2.10: Content Browser Import button

4. Browse to the directory of the files we downloaded in *Step 1*, select **SK_Mannequin.FBX**, and click on the **Open** button.
5. Make sure the **Import Animations** button is **unchecked** and click the **Import All** button. You may get a

warning here stating that **There are no smoothing groups**. You can ignore this for now. With that, you have successfully imported a skeletal mesh from an FBX file. Now, we need to import its animations.

6. Click the **Import** button again, browse to the folder we created in *Step 1*, and select **ThirdPersonIdle.fbx**, **ThirdPersonRun.fbx**, and **ThirdPersonWalk.fbx**. Then click on the **Open** button.
7. Make sure the skeleton is set to the one you imported in *Step 5* and click **Import All**:

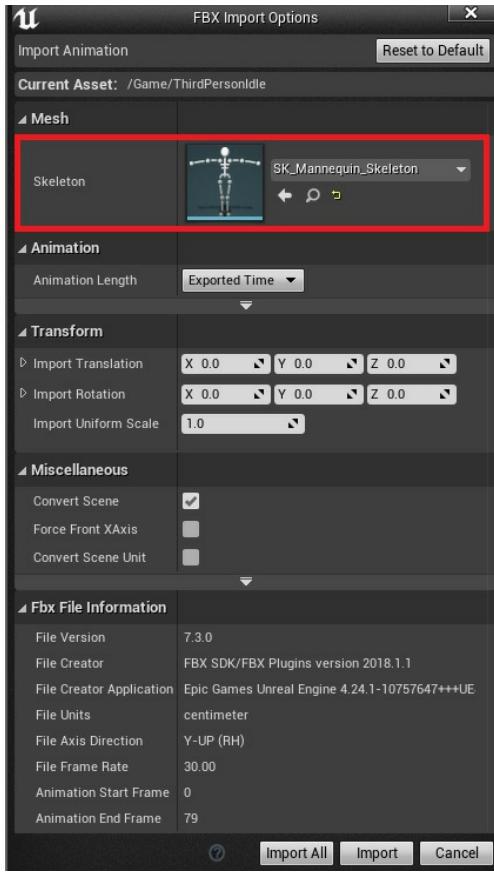


Figure 2.11: Animation FBX Import Options

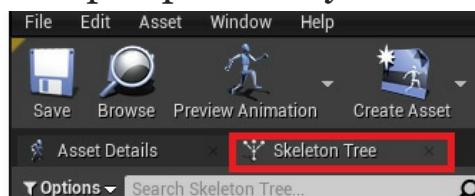
8. Now, you can see the three animations (**ThirdPersonIdle**, **ThirdPersonRun**, and **ThirdPersonWalk**) inside the **Content Browser**.
9. If you *double-click* on **ThirdPersonIdle**, you'll notice that the left arm is hanging down. This means that there's a retargeting issue. When the animations are imported

separately from the skeleton, the Unreal Engine internally maps all the bones from the animation to the skeleton but sometimes that results in a glitch. We're now going to resolve this glitch.



**Figure 2.12: ThirdPersonIdle UE4
mannequin animation glitch**

10. Open the **SK_Mannequin** Skeletal Mesh and open the **Skeleton Tree** tab if not open previously.



**Figure 2.13: SK_Mannequin
Skeleton Tree tab select**

11. Under **Options** enable the **Show Retargeting Options** checkbox.



Figure 2.14: Enabling retargeting options

12. Now inside the skeleton tree, reduce the **spine_01**, **thigh_l** and **thigh_r** bones to enable better visibility.
13. Now select the **spine_01**, **thigh_l** and **thigh_r** bones. *Right click* on them, and in the menu, click the **Recursively Set Translation Retargeting Skeleton** button. This will fix the bone translation issues we encountered before.
14. Re-open the **ThirdPersonIdle Animation** to verify the hanging arm has been fixed.



Figure 2.15: Fixed ThirdPersonIdle Animation

Note

You can locate the complete exercise code files on GitHub in the **Chapter02 -> Exercise2.03 -> Ex2.03-Completed.rar** directory by going to the following link:
<https://packt.live/2U8AScR>

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.

By completing this exercise, you've understood how to import assets and, more specifically, imported an FBX skeletal mesh and animation data into your project. This is crucial for the workflows of many game developers as assets are the building blocks of the entire game.

In the next section, we'll be looking at the Unreal core classes for creating a game, how important they are for creating a game or experience, and how to use them inside a project.

The Unreal Game Mode Class

Consider a situation where you want to be able to pause your game. All the logic and implementation required to be able to pause the game will be placed inside a single class. This class will be responsible for handling the game flow when a player enters the game. The game flow can be any action or a set of actions occurring in the game. For example, game pause, play, and restart are considered simple game flow actions.

Similarly, in the case of a multiplayer game, we require all the network-related gameplay logic to be placed together. This is exactly what the Game Mode class is there for.

Game Mode is a class that drives the game logic and imposes game-related rules on players. It essentially contains information about the current game being played, including gameplay variables and events, which are mentioned later on in this chapter. Game Mode can hold all the managers of the gameplay objects, it's a singleton class, and is directly accessible by any object or abstract class present in the game.

As with all the other classes, the Game Mode class can be extended in Blueprints or C++. This can be done to include extra functionality and logic that may be required to keep players updated about what's happening inside the game.

Let's go over some example game logic that goes inside the Game Mode class:

- Limiting the number of players that are allowed to enter the game
- Controlling the Spawn location and Player Controller logic of newly connected players

- Keeping track of the Game Score
- Keeping track of the Game Win/Lose condition
- Implementing the Game Over/Restart Game scenario

In the next section, we will look at the default classes provided by Game Mode.

GAME MODE DEFAULT CLASSES

In addition to itself, Game Mode uses several classes to implement game logic. It allows you to specify classes for its following defaults:

- **Game Session Class:** Handles admin-level game flow such as login approval.
- **Game State Class:** Handles the state of the game so that clients can see what's going on inside the game.
- **Player Controller Class:** The main class used to possess and control a pawn. Can be thought of as a brain that decides what to do.
- **Player State Class:** Holds the

current state of a player inside the game.

- **HUD Class:** Handles the user interface shown to the player.
- **Default Pawn Class:** The main actor that the player controls. This is essentially the player character.
- **Spectator Class:** Being a subclass of the **DefaultPawn** class, the Spectator Pawn Class specifies the pawn responsible for spectating the game.
- **Replay Spectator Player Controller:** The Player Controller responsible for manipulating replay during playback, within the game.
- **Server Stat Replicator Class:** Responsible for replicating server stat net data.

You can either use the default classes as is, or you can specify your own for custom implementation and behavior. These classes will work in conjunction with Game Mode and will automatically run without being placed inside the world.

Gameplay Events

In terms of a multiplayer game, when many players enter the game, it becomes essential to handle logic to allow their entry into the game, maintain their state, as well as to allow them to view other players' states and handle their interactions.

Game Mode provides you with several events that can be overridden to handle such multiplayer gameplay logic. The following events are especially useful for networking features and abilities (which they are mostly used for):

- **On Post Log In:** This event is called after the player is logged into the game successfully. From this point onward, it is safe to call replicated logic (used for networking in multiplayer games) on the Player Controller class.
- **Handle Starting New Player:** This event is called after the **On Post Log In** event and can be used to define what happens to the newly entered player. By default, it creates a pawn for the newly connected player.
- **SpawnDefaultPawnAtTransform:** This event triggers the actual pawn spawning within the game. Newly connected players can be spawned at particular transforms or at preset player start positions placed within the level (which can be added by Dragging and Dropping the Player Start from the

Models Window into the World).

- **On Logout:** This event is called when a player leaves the game or is destroyed.
- **On Restart Player:** This event is called to respawn the player. Similar to **SpawnDefaultPawnAtTransform**, the player can be respawned at specific transforms or pre-specified locations (using the player start position).

Networking

The Game Mode class is not replicated to any clients or joined players. Its scope is only limited to the server where it is spawned. Essentially, the client-server model dictates that the clients only act as inputs within the game that is being played on the server. Therefore, the gameplay logic should not exist for the clients, but only for the server.

GameModeBase versus GameMode

From version 4.14 onward, Epic introduced the **AGameModeBase** class, which acts as the parent class for all Game Mode classes. It is essentially a simplified version of the **AGameMode** class.

However, the Game Mode class contains some additional functionality that is better suited for Multiplayer Shooter type games as it implements the Match State concept. By default, the Game Mode Base is included in new template-based projects.

Game Mode also contains a State Machine that handles and keeps track of the player's state.

Levels

Levels, in gaming, are a section or a part of a game. Since many games are quite large, they are broken down into different levels. A level of interest is loaded into the game for the player to play, and then when they are done with that, another level may be loaded in (while the current one will be loaded out) so that the player can proceed. To complete a game, a player usually needs to complete a set of specific tasks to move on to the next level, eventually completing the game.

A Game Mode can be directly applied to the level. The level, upon loading, will use the assigned Game Mode class to handle all logic and gameplay for that particular level and override the game mode of the project for this level. This can be applied using the **World Settings** tab after opening a level.

A Level Blueprint is a blueprint that runs with the level, but cannot be accessed outside the scope of the level. Game Mode can be accessed in any blueprint (including the level blueprint) by the **Get Game Mode** node. This can later be cast to your Game Mode class, to obtain a reference to it.

Note

A level can only have one Game Mode class assigned to it.

However, a single Game Mode class can be assigned to multiple levels to imitate similar functionality and logic.

The Unreal Pawn Class

The **Pawn** class, in Unreal, is the most basic class of actors that can be possessed (either by a player or AI). It also graphically represents the player/bot in the game. Code inside this class should have everything to do with the game entities, including interaction, movement, and ability logic. The player can still only possess a single pawn at any time in the game. Also, the player can *unpossess* one pawn and *possess* another pawn during gameplay.

THE DEFAULT PAWN

Unreal Engine gives developers a **DefaultPawn** class (which inherits from the base **Pawn** class). On top of the **Pawn** class, this class contains additional code that allows it to move within the world, as you would in the editor version of the game.

THE SPECTATOR PAWN

Some games offer features to spectate games. Let's say you're waiting for a friend to finish their game before joining you, so you go ahead and spectate their game. This gives you the ability to observe the game the player is playing, through a camera that you can move around to get a view of the players or the game. Some games also offer spectate modes that can travel back in time, to show a particular action of the game that happened in the past or at any point in the game.

As the name suggests, this is a special type of pawn that

provides sample functionality to spectate a game. It contains all the basic tools (such as the Spectator Pawn Movement component) required to do so.

The Unreal Player Controller Class

The Player Controller class can be thought of as the player. It is essentially the *soul* of a pawn. A Player Controller takes input from the user and feeds it to the pawn and other classes for the player to interact with the game. However, you must take note of the following points while dealing with this class:

- Unlike the pawn, there can only be one Player Controller that the player represents in a level. (Just like when you travel in an elevator. While inside one, you can only control that elevator, but you can then exit it and enter another elevator in order to control that one.)
- The Player Controller persists throughout the game, but the pawn may not (for example, in a battle game, the player character may die and respawn, but the Player Controller would remain the same).
- Due to the temporary nature of the

pawn and the permanent nature of the Player Controller, developers need to keep in mind which code should be added to which class.

Let's understand this better through the next exercise.

Exercise 2.04: Setting Up the Game Mode, Player Controller, and Pawn

This exercise will use the blank project we created in *Exercise 2.01, Creating an Empty C++ Project*. We'll be adding our Game Mode, Player Controller, and **Pawn** class to the game and will be testing to see if our code works in Blueprints.

The following steps will help you complete this exercise:

1. Open the project we created in *Exercise 2.01, Creating an Empty C++ Project*.
2. Right-click inside the **Content Browser** and select **Blueprint Class**.
3. Under the **All Classes** section, find and select the **Game Mode** class:

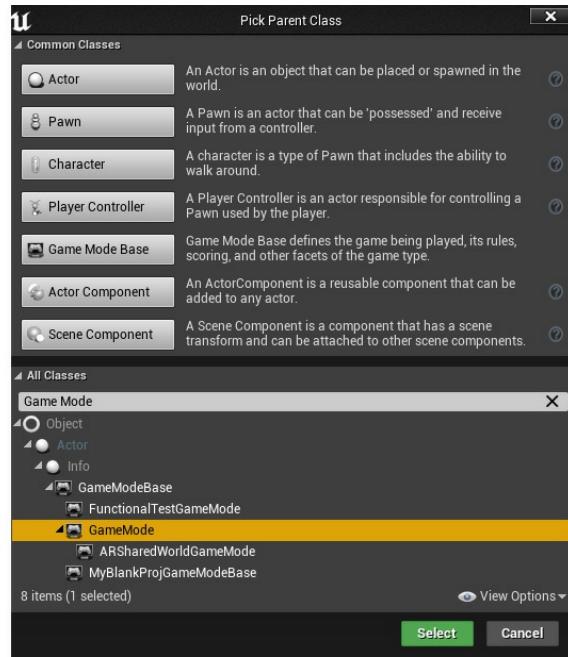


Figure 2.16: Selecting the Game Mode class

4. Set its name to **BP_MyGameMode**.
5. Repeat *Steps 2-4* and select the **Pawn** class from under the **Common Classes** section, as shown in the preceding screenshot. Set the name of this class to **BP_MyPawn**.
6. Repeat *Steps 2-4* and select the **Player Controller** class under the **Common Classes** section, as shown in the preceding screenshot. Set the name of this class to **BP_MyPC**:

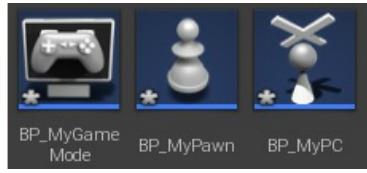


Figure 2.17: Game Mode, Pawn, and Player Controller names

7. Open **BP_MyGameMode** and open the **Event Graph** tab:



Figure 2.18: Event Graph tab in Blueprint

8. *Left-click and drag from the white pin in the Event BeginPlay node and then release the left mouse button to gain an Options menu. Type print and select the print node highlighted in the list:*

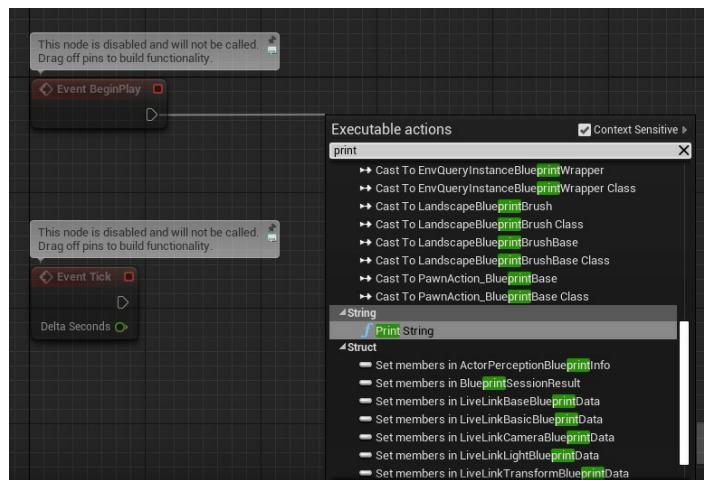


Figure 2.19: Print String node

(Blueprint)

9. In the resultant **Print String** node that gets placed under the **In String** parameter, type **My Game Mode has started!**.
10. Now, press the **Compile** and **Save** buttons on the top menu bar.
11. Repeat *Steps 7-10* for both the **BP_MyPawn** and **BP_MyPC** classes, setting the **In String** parameter to **My Pawn has started!** and **My PC has started!**, respectively.
12. Finally, open the **World Settings** tab, and under the **Game Mode** section, use the dropdown to set the **GameMode Override, Default Pawn Class**, and **Player Controller Class** options to our respective classes:

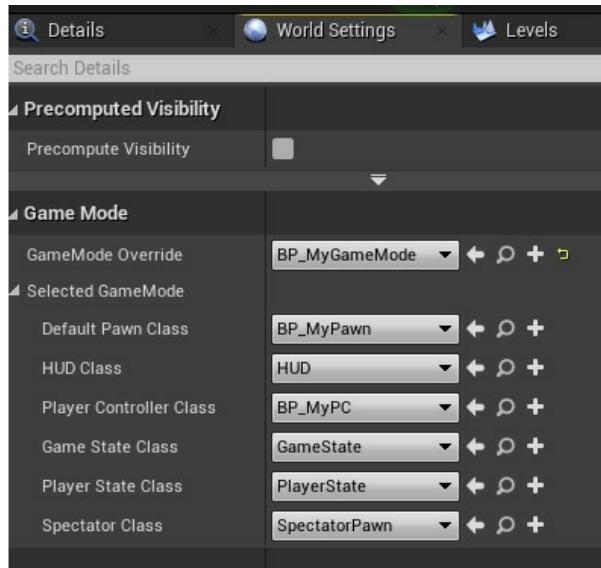


Figure 2.20: World Settings and Game Mode setup

13. Click **Play** to play your game and see the three print statements on the top. This means that the current **GameMode Override**, **Default Pawn Class**, and **Player Controller Class** options have been set to your specified classes and are running their code:

```
My Pawn has started!  
My PC has started!  
My Game Mode has started!
```

Figure 2.21: Output prints

Note

You can locate the completed exercise code files on GitHub, in the **Chapter02 -> Exercise2.04 -> Ex2.04-**

Completed.rar directory, at the following link:

<https://packt.live/3k7nS1K>

*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.*

Now that you know the basic classes and how they work in Unreal, in the next section, we will be looking at animations, what processes are involved, and how they complete them. We'll follow this with an exercise.

Animations

Animation is essential for adding life and richness to a game. Superb animations are one of the major factors that differentiate average games from the good and the great from the best. Visual fidelity is what keeps gamers excited and immersed in games, and hence animations are a core part of all games and experiences created in Unreal Engine.

Note

This chapter seeks to cover animation basics. A more in-depth approach to animation will be taken in Chapter 13, Blend Spaces 1D, Key Bindings, and State Machines.

Animation Blueprints

An Animation Blueprint is a specific kind of blueprint that allows you to control the animation of a Skeletal Mesh. It

provides users with a graph specifically for animation-related tasks. Here, you can define the logic for computing the poses of a skeleton.

Note

A Skeletal Mesh is a skeleton-based mesh that has bones, all of which come together to give form to the mesh, whereas a Static Mesh (as the name suggests) is an un-animatable mesh. Skeletal Meshes are normally used for characters and life-like objects (for example, a player hero), whereas Static Meshes are used for basic or lifeless objects (for example, a wall).

Animation Blueprints provide two kinds of graphs:
EventGraph and **AnimGraph**.

Event Graph

The Event Graph within an Animation Blueprint provides setup events related to animations, as we learned in *Chapter 1, Unreal Engine Introduction*, that can be used for variable manipulation and logic. Event graphs are mostly used within Animation Blueprints to update Blend Space values, which, in turn, drive the animations within **AnimGraph**. The most common events that are used here are as follows:

- **Blueprint Initialize Animation:**
Used to initialize the animation.
- **Blueprint Update Animation:** This event is executed every frame, giving developers the ability to perform calculations and update its values as

required:

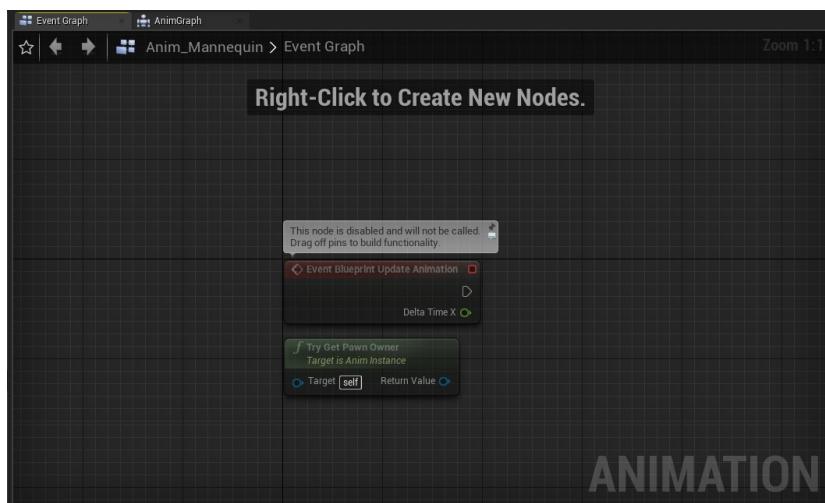


Figure 2.22: Animation Event Graph

In the preceding screenshot, you can see the default Event Graph. There are **Event Blueprint Update Animation** and **Try Get Pawn Owner** nodes here. You created new nodes and appended them to a graph to complete some meaningful tasks in *Exercise 2.04, Setting Up the Game Mode, Player Controller, and Pawn*.

The Anim Graph

The Anim Graph is dedicated to and responsible for playing animations and outputting the final pose of the skeleton, on a per-frame basis. It provides developers with special nodes to execute different logic. For example, the Blend node takes in multiple inputs and is used to decide which input is currently being used in the execution. This decision is usually dependent on some external input (such as an alpha value).

The Anim Graph works by evaluating nodes by following the flow of execution between the exec pins on the nodes being

used.

In the following screenshot, you can see a single **Output Pose** node on the graph. This is the final pose output of the animation that will be visible on the relevant Skeletal Mesh within the game. We will be using this in *Exercise 2.05, Creating a Mannequin Animation*:

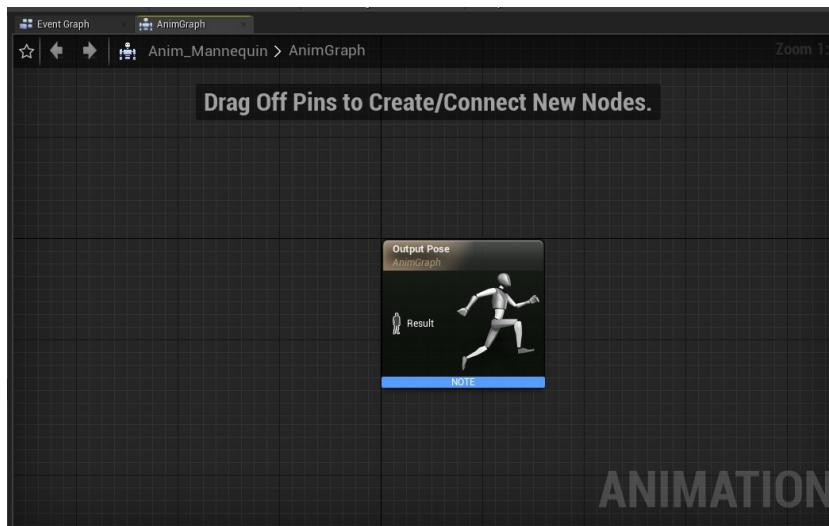


Figure 2.23: Animation AnimGraph

State Machines

You have already learned how animation nodes and logic can be set up, but one essential component is missing. Who decides when a particular animation or piece of logic should play or execute? This is where State Machines come into the picture. For example, a player may need to shift from crouching to a standing pose, so the animation needs to be updated. The code will call the Animation Blueprint, access the State Machine, and let it know that the state of the animation needs to be changed, resulting in a smooth animation transition.

A State Machine consists of states and rules that can be

thought of as depicting the state of an animation. A State Machine can always be in one state at a particular time. A transition from one state to another is carried out when certain conditions (which are defined by rules) are met.

Transition Rules

Each Transition Rule contains a Boolean node by the name of **Result**. If the Boolean is true, the transition can occur and vice versa:

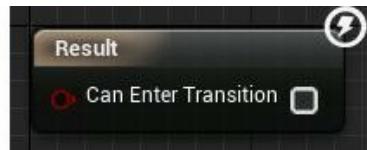


Figure 2.24: Transition Rules

Blend Spaces

When you're provided with a bunch of animations, you can create a State Machine and run those animations. However, a problem is presented when you need to transition from one animation to another. If you simply switch the animation, it will glitch since the new animation's starting pose might be different from the old animation's ending pose.

Blend Spaces are special assets used to interpolate between different animations based on their alpha values. This, in turn, removes the glitch issue and interpolates between the two animations, causing a swift and smooth change in animation.

Blend Spaces are created either in one dimension, known as a Blend Space 1D, or two dimensions, known as a Blend Space.

These blend any number of animations based on one or two input(s), respectively.

Exercise 2.05: Creating a Mannequin Animation

Now that you've gone through most of the concepts related to animations, we'll be diving in hands-on by adding some animation logic to the default mannequin. We'll be creating a Blend Space 1D, a State Machine, and Animation logic.

Our goal here is to create a running animation of our characters and thus gain insight into how animations work, as well as the way they are bound to the actual character in a 3D world.

The following steps will help you complete this exercise:

1. Download and extract all the contents of the **Chapter02 -> Exercise2.05 -> ExerciseFiles** directory, which can be found on GitHub. You can extract this to any directory you're comfortable with using on your machine.

Note

*The **ExerciseFiles** directory can be found on GitHub at the following link:
<https://packt.live/32tIFGJ>.*

2. *Double-click* the **CharAnim.uproject** file to start the project.
3. Press **Play**. Use the keyboard's *W, A, S, D* keys to move and the *Spacebar* to jump. Notice that, currently, there are no animations on the mannequin.
4. In the **Content** folder, browse to **Content -> Mannequin -> Animations**.
5. *Right-click* the **Content** folder, and from the **Animation** section, select **Blend Space 1D**.
6. Select **UE4_Mannequin_Skeleton**.
7. Rename the newly created file to **BS_IdleRun**.
8. *Double-click* **BS_IdleRun** to open it.
9. 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 **375.0**:

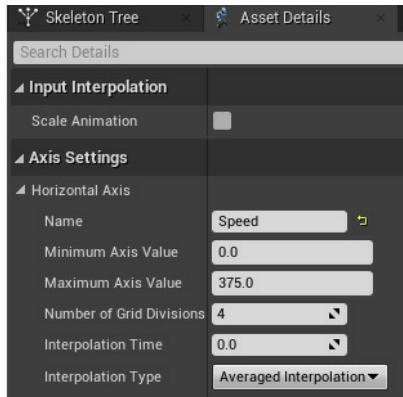


Figure 2.25: Blend Space 1D Axis Settings

10. Head down to the **Sample Interpolation** section and set **Target Weight Interpolation Speed Per Sec** to **5.0.**
11. Drag and drop the **ThirdPersonIdle**, **ThirdPersonWalk**, and **ThirdPersonRun** animations into the graph separately:

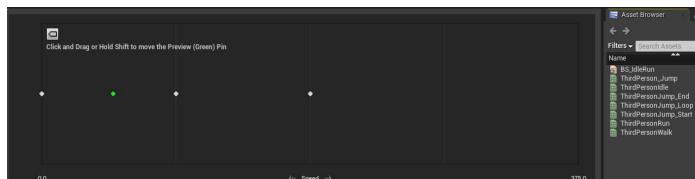


Figure 2.26: Blend Space previewer

12. Under the **Asset Details** tab, in **Blend Samples**, set the following

variable values:

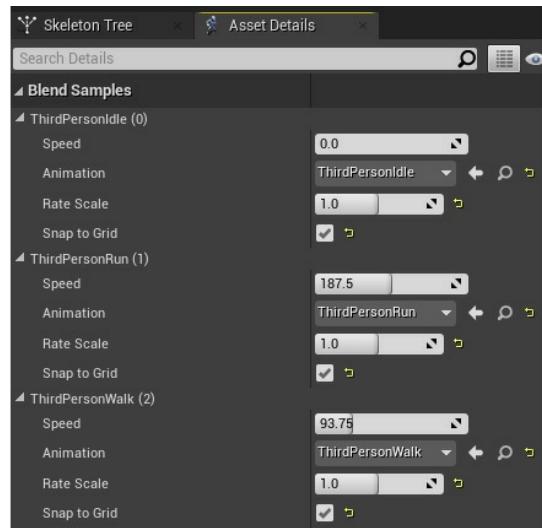


Figure 2.27: Blend Samples

13. Click **Save** and close this **Asset**.
14. *Right-click* inside the **Content** folder, and from the **Animation** section, select **Animation Blueprint**.
15. In the **Target Skeleton** section, select **UE4_Mannequin_Skeleton** and then click the **OK** button:

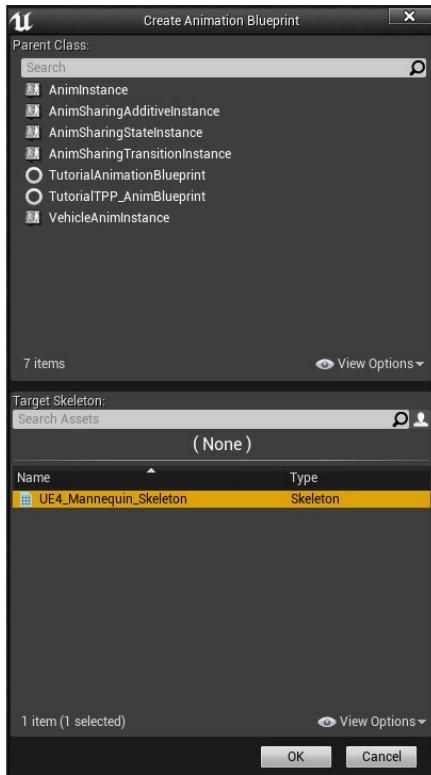


Figure 2.28: Creating the Animation Blueprint asset

16. Name the file **Anim_Mannequin** and press *Enter*.
17. *Double-click* the newly created **Anim_Mannequin** file.
18. Next, go to the **Event Graph** tab.
19. Create a **boolean** variable called **IsInAir?** by clicking the **+** icon in the variable section on the bottom left side. Be sure to assign the proper type:

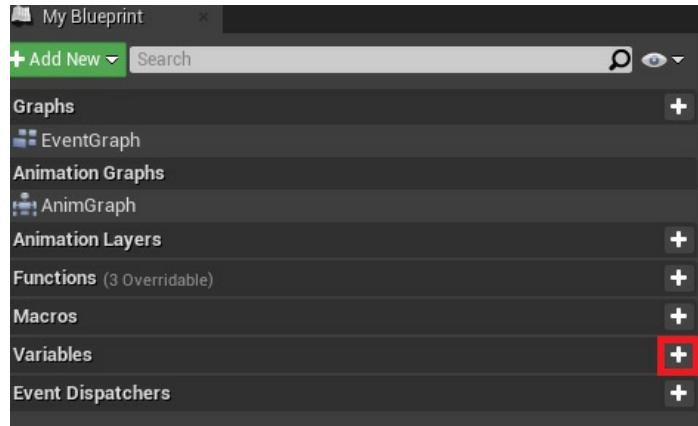


Figure 2.29: Adding variables

20. Create a float variable called **Speed**.

21. Drag off the **Try Get Pawn Owner** return value node and type in **Is Valid**. Select the bottom one:

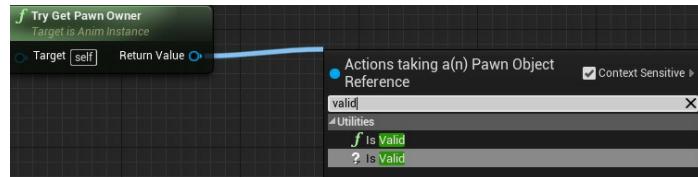


Figure 2.30: Event Graph Is Valid node

22. Connect the **Exec** pin from the **Event Blueprint Update Animation** node to the **Is Valid** node:

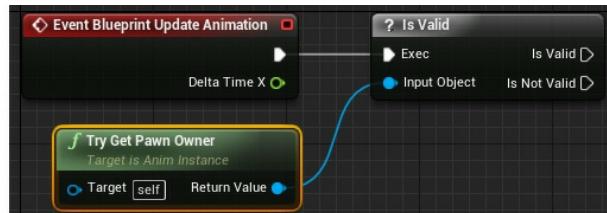


Figure 2.31: Connecting nodes

23. From the **Try Get Pawn Owner** node, use the **Get Movement Component** node.
24. From the node obtained in *Step 22*, 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 with the **Is Valid** exec pin:



Figure 2.32: Is in Air Boolean setup

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



Figure 2.33: Speed Boolean setup

26. Next, head to the **Anim Graph** tab.

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

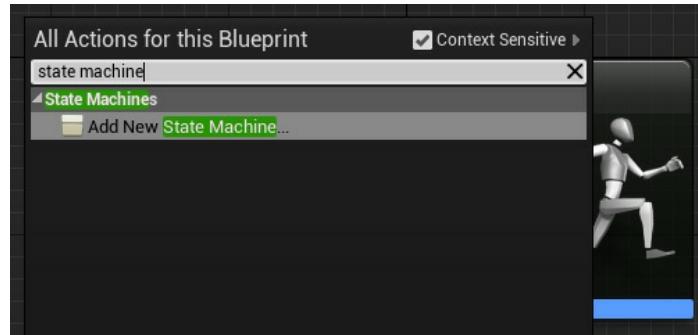


Figure 2.34: The Add New State Machine option

28. Make sure the node is selected and then press *F2* to rename it **MannequinStateMachine**.

29. Connect the output pin of **MannequinStateMachine** to the input pin for the **Output Pose** node and click the compile button on the top bar:

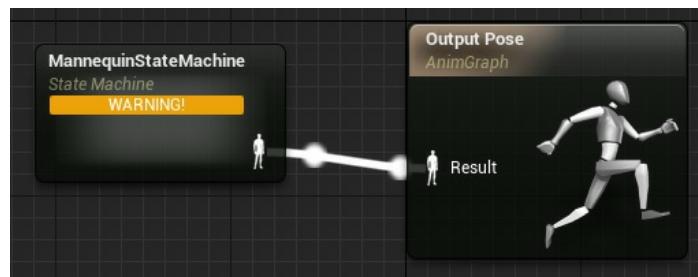


Figure 2.35: Configuring the State Machine result in the Output Pose node

30. *Double-click* the **MannequinstateMachine** node to enter the State Machine. You will see an **Entry** node. The state that will be connected to it will become the default state of the mannequin. In this exercise, this will be our **Idle Animation**.
31. *Right-click* on an empty area inside the State Machine, and from the menu, select **Add State**. Press *F2* to rename it **Idle/Run**.
32. Drag from the icon next to the **Entry** text, point it inside the **Idle/Run** node, and then release it to connect it:

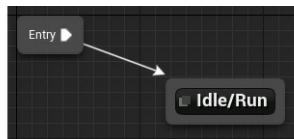


Figure 2.36: Connecting Added State to Entry

33. *Double-click* on the **Idle/Run** state to open it.

34. 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, as shown here:

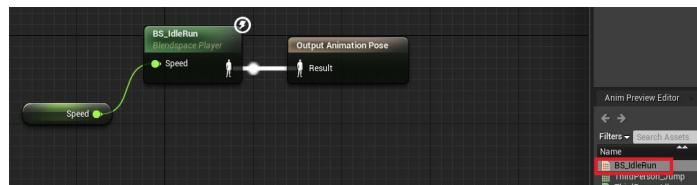


Figure 2.37: Idle/Run state setup

35. Head back to **MannequinStateMachine** by clicking on its breadcrumb in the top banner:



Figure 2.38: State Machine navigation breadcrumb

36. From the **Asset Browser** menu, drag and drop the **ThirdPersonJump_Start** Animation into the graph. Rename it **Jump_Start**.
37. Repeat *Step 35* for **ThirdPersonJump_Loop** and

ThirdPerson_Jump and rename them **Jump_Loop** and **Jump_End**, respectively:



Figure 2.39: State setup

38. Open the **Jump_Start** state. Click on the **Play ThirdPersonJump_Start** node. *Uncheck Loop Animation* in the **Settings** section.
39. Open the **Jump_Loop** state and click on the **Play ThirdPersonJump_Loop** node. Set **Play Rate** to **0.75**.
40. Open the **Jump_End** state and click on the **Play ThirdPerson_Jump** node. *Uncheck the Loop Animation Boolean*.
41. Since we can shift from **Idle/Run** to **Jump_Start**, drag from the **Idle/Run** state and drop it to the **Jump_Start** state. Similarly,

Jump_Start leads to **Jump_Loop**, then to **Jump_End**, and finally back to **Idle/Run**.

Drag and drop the arrows to set up the State Machine, as follows:

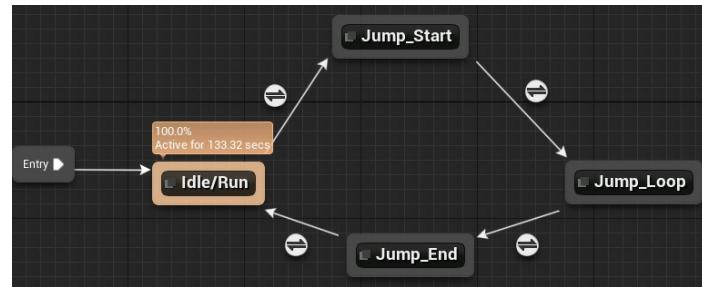


Figure 2.40: State connections

42. Double-click the **Idle/Run** to **Jump_Start** transition rule icon and connect the output of the **Is in Air?** variable to the result:



Figure 2.41: Idle/Run to Jump_Start transition rule setup

43. Open the **Jump_Start** to **Jump_Loop** transition rule. Get the **Time Remaining (ratio)** node for **ThirdPersonJump_Start** and check whether it is less than **0 . 1**. Connect

the resulting bool to the result:

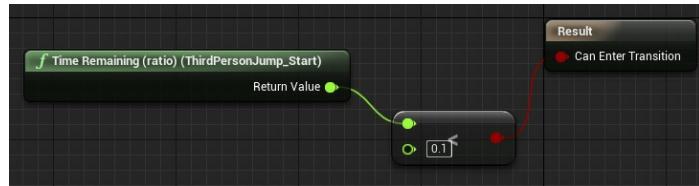


Figure 2.42: Jump_Start to Jump_End transition rule setup

44. Open the **Jump_Loop** to **Jump_End** transition rule. Connect the output of the inverse of the **Is in Air?** variable to the result:



Figure 2.43: Jump_Loop to Jump_End transition rule setup

45. Open the **Jump_End** to **Idle/Run** transition rule. Get the **Time Remaining (ratio)** node for **ThirdPerson_Jump** and check whether it is less than **0.1**. Connect the resulting bool to the result:



Figure 2.44: Jump_End to Idle/Run transition rule setup

46. Close the Animation Blueprint.
47. In the **Content** folder, browse to **Content -> ThirdPersonBP -> Blueprints folder** and open the **ThirdPersonCharacter Blueprint**.
48. Select **Mesh** in the **Components** tab:

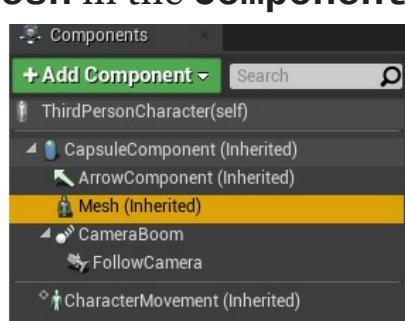


Figure 2.45: Mesh component

49. In the **Details** tab, set **Anim Class** to the **Animation Blueprint** class that you created:

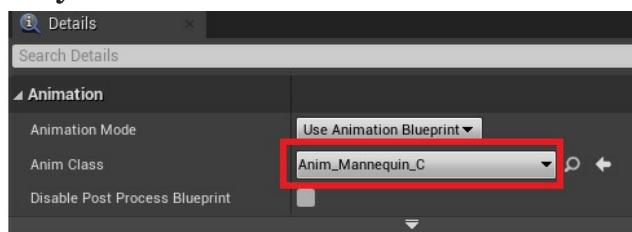


Figure 2.46: Specifying the Animation Blueprint in the Skeletal Mesh component

50. Close the Blueprint.

51. Play the game again and notice the animations.

The following should be the output you achieve. As you can see, our character is running, and the running animation is being shown:

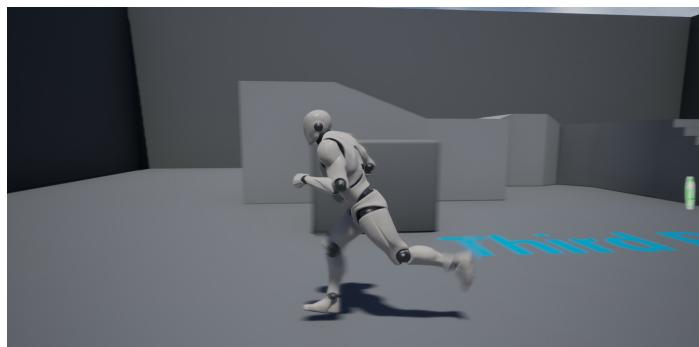


Figure 2.47: Character running animation

Note

You can find the complete exercise code files on GitHub, in the **Chapter02 -> Exercise2.05 -> Ex2.05-Completed.rar** directory, at the following link:
<https://packt.live/3kdIlSL>

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.

By completing this exercise, you've understood how to create State Machines, a Blend Space 1D, the Animation Blueprint, and how to tie it all together with the Skeletal Mesh of a character. You've also worked on play rates, transitional speed

and the transitional states, helping you understand how the world of animation intricately ties in together.

We kicked off this section by understanding how State Machines are used to represent and transition in-between Animation States. Next, we got to know how a Blend Space 1D gives us blending in-between those transitions. All this is used by the Animation Blueprint to decide what the current animation of the character is. Now, let's combine all these concepts together in an activity.

Activity 2.01: Linking Animations to a Character

Let's say, as an Unreal games developer, you've been provided with a character skeletal mesh and its animations, and you've been tasked with integrating them inside a project. In order to do that, in this activity, you'll be creating an Animation Blueprint, State Machines, and a Blend Space 1D of a new character. By completing this activity, you should be able to work with animations in Unreal Engine and link them to skeletal meshes.

The activity project folder contains a Third Person Template project, along with a new character, **Ganfault**.

Note

*This character and its animations were downloaded from mixamo.com. These have been placed in the **Content -> Ganfault** folder on our GitHub repository:
<https://packt.live/35eCGrk>*

Mixamo.com is a website that sells 3D characters with animations and is sort of an asset marketplace only for 3D

models. It also contains a library of free models, alongside the paid ones.

The following steps will help you complete this activity:

1. Create a Blend Space 1D for the Walking/Running animation and to set up the Animation Blueprint.
2. Next, go to **Content** -> **ThirdPersonBP** -> **Blueprints** and open the **ThirdPersonCharacter** Blueprint.
3. Click the Skeletal Mesh component on the left, and inside the **Details** tab on the right, replace the **SkeletalMesh** reference with **Ganfault**.
4. Similarly, update the **Animations Blueprint** section of the skeletal mesh component with the Animation Blueprint you created for **Ganfault**.

Note

For the State Machine, implement only Idle/Run and Jump State.

Once you've completed this activity, the Walk/Run and Jump

animations should be working properly, as shown in the following output:

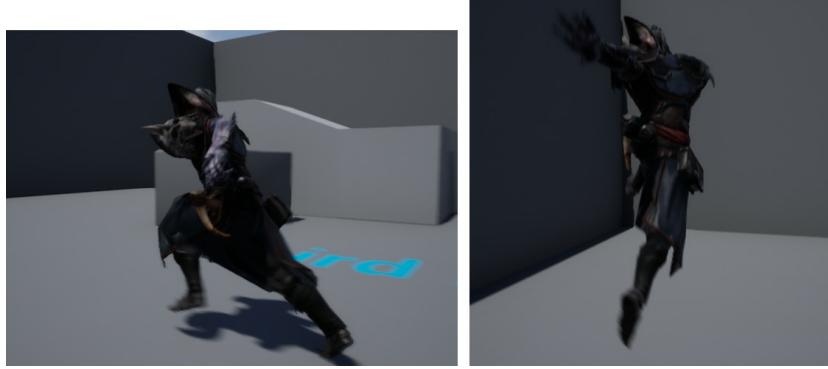


Figure 2.48: Activity 2.01 expected output (Left: Run; Right: Jump)

Note

*The solution to this activity can be found at:
<https://packt.live/338jEBx>.*

By completing this activity, you now know how to navigate your way around Unreal Engine with regard to the project, debugging code, and working with Animations. You also understand State Machines, which represent transitions between the Animation States and the Blend Spaces 1D used in that transition. You are now able to add animation to 3D models based on gameplay events and inputs.

Summary

To summarize this chapter, we first learned how to create an empty project. Then, we learned about the folder structure and how to organize files in the project directory. After that, we looked at template-based projects. We also learned how to set breakpoints in code so that we can watch variable values and debug entire objects while the game is running, which

would help us find and eradicate bugs in our code.

Thereafter, we saw how Game Mode, Player Pawn, and Player Controller are relevant classes used in Unreal Engine for setting up game flows (the execution order of code), as well as how they are set up inside a project.

Finally, we transitioned toward animation basics and worked with State Machines, Blend Spaces 1D, and Animation Blueprints to make our character animate (walk/run and jump) within the game according to the keyboard input.

Throughout this chapter, we became more familiar with the powerful tools in Unreal Engine that are essential to game development. Unreal's Game Mode and its default classes are required for making any kind of game or experience in Unreal Engine. Additionally, animations bring life to your character and help add layers of immersiveness inside your games. All game studios have animations, characters, and game logic since these are the core components that drive any game. These skills will help you numerous times throughout your game development journey.

In the next chapter, we will talk about the **Character** class in Unreal Engine, its components, and how to extend the class for additional setup. You'll be working on various exercises, followed by an activity.

3. Character Class Components and Blueprint Setup

Overview

*This chapter will focus on the **Character** class in C++. You will be shown how to extend the **Character** class in C++ and then extend this newly created **Character** class further in Blueprints via inheritance. You will also work with player input and some movement logic.*

By the end of this chapter, you will be able to understand how class inheritance works in UE4 and how to utilize it to your advantage. You will also be able to work with Axis and Action Input Mappings, which are key in driving player-related input logic.

Introduction

In the previous chapter, we learned how to create empty projects and import files, which folder structure to use, and how to work with animations. In this chapter, we'll explore some other key tools and functionality that you will work with when using Unreal Engine.

Game developers often need to use certain tools that save them time and energy when building game functionality. Unreal Engine's powerful object inheritance capabilities give

developers the edge they need to be more efficient. Developers can also work with both C++ and Blueprints interchangeably and use them to their benefit when developing games.

Another value-added benefit developers gain is the ability to extend code for use later in a project. Let's say your client has new requirements that build upon the old ones (as is the case in most game studios). Now, to extend functionality, developers can just inherit a class and add more functionality to it to get results quickly. This is very powerful, and it comes in handy in many situations.

In this chapter, we will discuss the Unreal **Character** class, create C++ code, and then extend it in Blueprints, before finally using it to create an in-game character.

The Unreal Character Class

Before we talk about the Unreal **Character** class, let's briefly touch on the concept of inheritance. If you're used to working with C++ or another similar language, you should already be familiar with this concept. Inheritance is the process whereby a class derives characteristics and behavior from another class. A C++ class can be extended to create a new class – the derived class – that retains properties of the base class and allows for these properties to be modified, or for new characteristics to be added. An example of this is the **Character** class.

A **Character** class is a special type of pawn and is a descendant of the Unreal **Pawn** class. Extending upon the **Pawn** class, the **Character** class has some movement capabilities by default, along with some inputs that add movement to the character. As standard, the **Character** class gives users the ability to get a character to walk, run,

jump, fly, and swim within the created world.

Since a **Character** class is an extension of the **Pawn** class, it contains all the code/logic of the pawn, and developers can extend this class to add more functionality to it. When extending the **Character** class, its existing components get carried over to the extended class as inherited components. (In this case, the Capsule component, Arrow component, and Mesh).

Note

*Inherited components cannot be removed. Their settings may be changed, but a component that's added to a base class will always be present in the extended class. In this case, the base class is the **Pawn** class, while the extended (or child) class is the **Character** class.*

The **Character** class provides the following inherited components:

- **Capsule component:** This is the root component that serves as the "origin" that other components get attached to within the hierarchy. This component can also be used for collisions and takes the form of a capsule that logically outlines many character forms (especially humanoid ones).
- **Arrow component:** This provides a simple arrow pointing toward the front of the hierarchy. By default, this is set

to **hide** when the game starts, but it can be tweaked to be visible. This component can be useful for debugging and adjusting game logic if required.

- **Skeletal Mesh component:** This is the primary component that developers are mostly concerned with within the **Character** class. The Skeletal Mesh, which is the form the character will take, can be set up here along with all the relevant variables, including animations, collisions, and so on.

Most developers usually prefer to code the game and character logic in C++ and extend that class to blueprints so that they can perform other simple tasks, such as connecting assets to the class. So, for example, a developer may create a C++ class that inherits from the **Character** class, write all the movement and jumping logic within that class, and then extend this class with a Blueprint, in which the developer updates the components with the required assets (such as the Skeletal Mesh and animation blueprint), and optionally code additional functionality into blueprints.

Extending the Character Class

The **Character** class is extended when it is inherited by

either C++ or Blueprints. This extended **Character** class will be a child of the **Character** class (*which will be called its parent*). Class extension is a powerful part of object-oriented programming, and classes can be extended to great depths and hierarchies.

Exercise 3.01: Creating and Setting Up a Third-Person Character C++ Class

In this exercise, you will create a C++ class based on a **Character** class. You will also initialize the variables that will be set in the default values for the class that will extend this **Character** class.

The following steps will help you complete this exercise:

1. Launch Unreal Engine, select the **Games** category, and click the **Next** button.
2. Select **Blank** and click the **Next** button.
3. Choose **C++** as the project type, set up the project name as **MyThirdPerson**, choose a suitable project directory, and click the **Create Project** button.

4. Right-click in the **Content Browser** interface and click the **New C++ Class** button:
5. In the dialog box that opens, select **Character** as the class type and click the **Next** button.
6. Name it **MyThirdPersonChar** and click the **Create Class** button.
7. Upon doing so, Visual Studio will open the **MyThirdPersonChar.cpp** and **MyThirdPersonChar.h** tabs.

Note

On some systems, it might be required to run the Unreal Engine editor with administrator privileges to automatically open the Visual Studio solution with the newly created C++ files.

8. Open the **MyThirdPersonChar.h** tab and add the following code under the **GENERATED_BODY()** text:

```
// Spring arm component  
which will act as a
```

```
placeholder for the player
camera

UPROPERTY(VisibleAnywhere,
BlueprintReadOnly, Category
= MyTPS_Cam, meta =
(AllowPrivateAccess =
"true"))

class USpringArmComponent*
CameraBoom;

// Follow camera

UPROPERTY(VisibleAnywhere,
BlueprintReadOnly, Category
= MyTPS_Cam, meta =
(AllowPrivateAccess =
"true"))

class UCameraComponent*
FollowCamera;
```

In the preceding code, we're declaring two components: the **Camera** itself and **Camera boom**, which acts as the placeholder for the camera at a certain distance from the player. These components will be initialized in the constructor in *Step 11*.

9. Add the following in the includes

section, under **#include**
"CoreMinimal.h", in the
MyThirdPersonChar.h file:

```
#include  
"GameFramework/SpringArmCom  
ponent.h"  
  
#include  
"Camera/CameraComponent.h"
```

10. Now, head over to the
MyThirdPersonChar.cpp tab and
add the following includes after the
#include MyThirdPersonChar.h
code:

```
#include  
"Components/CapsuleComponen  
t.h"  
  
#include  
"GameFramework/CharacterMov  
ementComponent.h"
```

In the preceding code snippet, the code adds the relevant classes to the class, which means we now have access to its methods and definitions.

11. In the
AMyThirdPersonChar::AMyThird

PersonChar() 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 camera that
will follow the character

FollowCamera =
CreateDefaultSubobject<UCameraComponent>
(TEXT("FollowCamera"));

FollowCamera-
>SetupAttachment(CameraBoom
,
USpringArmComponent::Socket
Name);

FollowCamera-
```

```
>bUsePawnControlRotation =  
false;
```

The last line of the preceding code snipped will set up the camera to bind its rotation with the pawn's. This means that the camera should, in turn, rotate with rotation of the player controller associated with this pawn.

12. Head back to the Unreal Engine project and click the **Compile** button in the top bar:

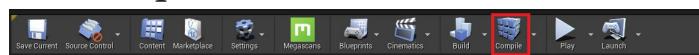


Figure 3.1: Compile button on the Unreal Editor top bar

A **Compile Complete!** message should appear on the bottom-right.

Note

You can locate the completed exercise code files on GitHub, in the **Chapter03 -> Exercise3.01** directory, at the following link: <https://packt.live/3khFrMt>.

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.

By completing this exercise, you've learned how to extend the **Character** class. You also learned how to initialize the default components of the **Character** class and how to compile the updated code from within Unreal Editor. Next up, you will learn how to extend the C++ class you created in Blueprints and why that is feasible in many situations.

Extending the C++ Class with Blueprints

As mentioned earlier, most developers extend the C++ code logic to blueprints in order to link this with the assets they will use. This is done to achieve easy asset assignment compared to finding and setting up the asset in code. Furthermore, it gives developers the ability to use powerful blueprint features such as Timelines, Events, and ready-to-use macros, in combination with their C++ code, to achieve the maximum benefit of developing with both C++ and Blueprints.

So far, we have made a C++ **Character** class. In it, we have set up components and movement capabilities. Now, we want to specify the assets that are going to be used in our class, as well as add input and movement ability. For this, it is easier to extend with Blueprint and set up the options there. This is what we will be doing in the next exercise.

Exercise 3.02: Extending C++ with Blueprints

In this exercise, you will learn how to extend the C++ class you created with Blueprints to add Blueprint code on top of the pre-existing C++ code. You will also be adding input key

bindings, which will be responsible for moving the character.

The following steps will help you complete this exercise:

1. Download and extract all the contents of the **Chapter03 | Exercise3 . 02 | ExerciseFiles** directory, which can be found on GitHub.

Note

*The **ExerciseFiles** directory can be found on GitHub at the following link:
<https://packt.live/2GOodG8>.*

2. Browse to the **Content** folder inside the **MyThirdPerson** project we created in *Exercise 3.01, Creating and Setting Up a Third-Person Character C++ Class*.
3. Copy the **MixamoAnimPack** folder we created in *Step 1* and paste it into the **Content** folder directory we opened in *Step 2*, as shown in the following screenshot:

Note

*The **MixamoAnimPack** assets were*

obtained from the Epic marketplace via the following link:

<https://www.unrealengine.com/marketplace/en-US/product/mixamo-animation-pack>.

Name	Date modified	Type	Size
Collections	5/4/2020 9:36 AM	File folder	
Developers	5/4/2020 9:36 AM	File folder	
MixamoAnimPack	5/5/2020 12:43 AM	File folder	

Figure 3.2: MixamoAnimPack placed in the project directory

4. Open the project. *Right-click* inside the **Content Browser** interface and click **Blueprint Class**.
5. In the **Search** dialogue, type **GameMode**, *right-click* the class matching the name, and click the **Select** button. Have a look at the following screenshot:

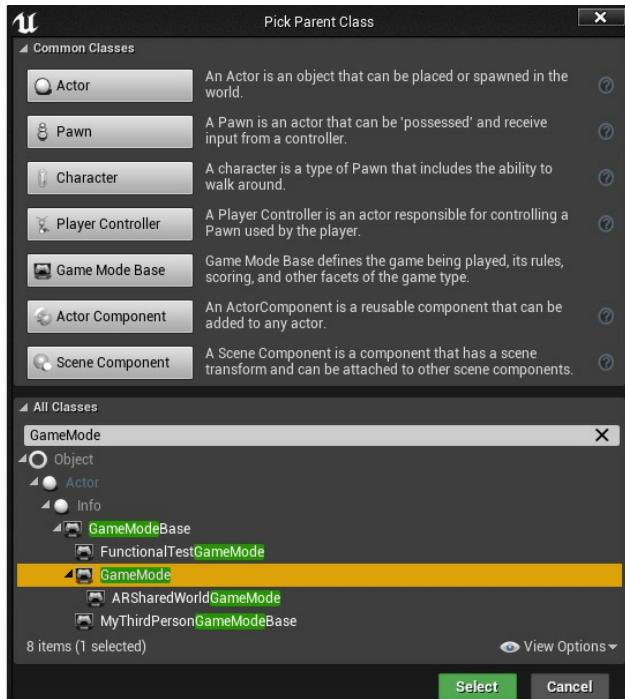


Figure 3.3: Creating the GameMode class

6. Name the blueprint we created in *Step 6 BP_GameMode*.
7. Now, repeat *Step 5*.
8. In the **Search** box, type **MyThirdPersonChar**, select the class, and then *right-click* on the **Select** button.
9. Name the blueprint we created in *Step 9 BP_MyTPC*.
10. In the **World Settings** tab, click the

None option next to **GameMode**
Override and select **BP_GameMode**:

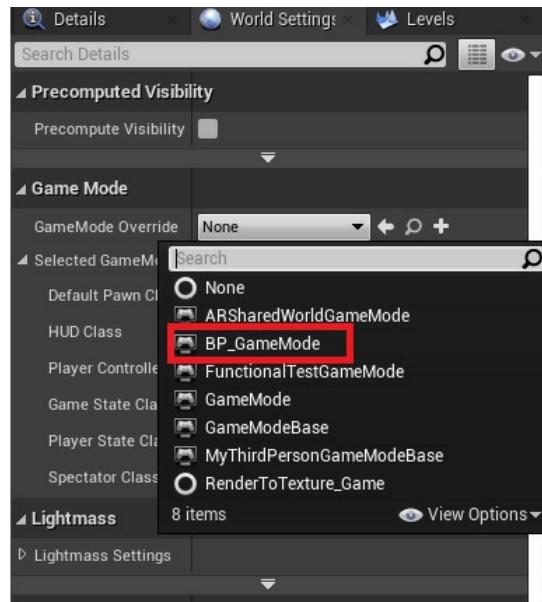


Figure 3.4: Specifying Game Mode in World Settings

11. Set **Default Pawn Class** to **BP_MyTPC**:

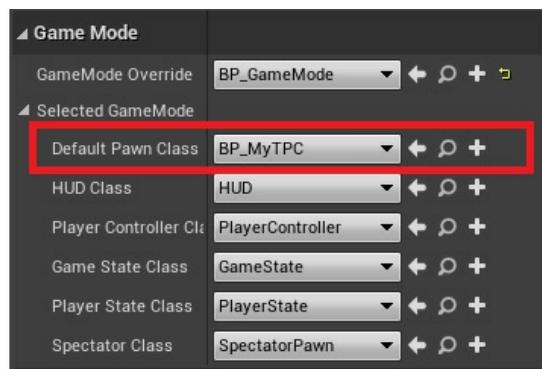


Figure 3.5: Specifying Default Pawn Class in Game Mode

12. Open **BP_MyTPC** and click on the **Mesh**

(**Inherited**) component in the hierarchy of the **Components** tab on the left-hand side.

13. In the **Details** tab, find the **Mesh** section and set **Skeletal Mesh** to **Maximo_Adam**.

Note

Meshes and Animations will be covered in depth in Chapter 13, Blend Spaces 1D, Key Bindings, and State Machines.

14. In the **Details** tab, find the **Animation** section and set **Anim Class** to **MixamoAnimBP_Adam_C**. You'll note that this class name gets suffixed with **_C** when selected. This is basically the instance of the blueprint created by UE4. Blueprints, in a working project/build, usually get suffixed this way to differentiate between a Blueprint Class and an instance of that class.

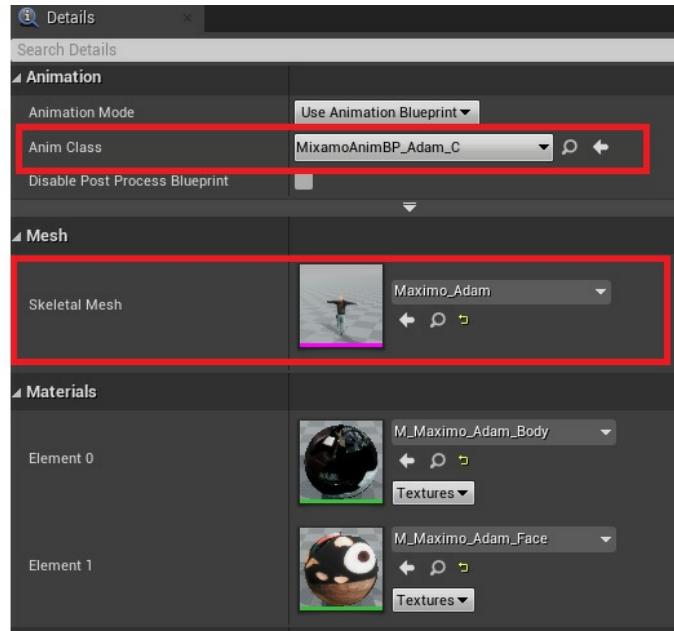


Figure 3.6: Setting up Anim Class and Skeletal Mesh

15. From the top-most menu, go to the **Edit** drop-down and click **Project Settings**.
16. Click on the **Input** section, which can be found in the **Engine** section:

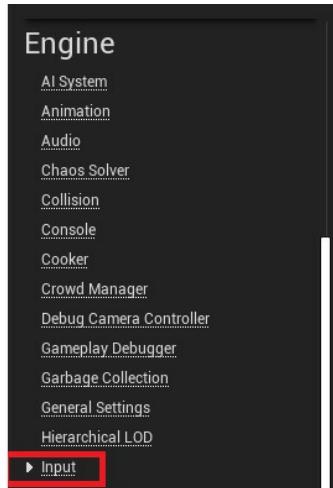


Figure 3.7: Input section of Project Settings

17. In the **Bindings** section, click the **+** icon next to **Axis Mappings** and expand the section.

Note

Action Mappings are single keypress actions that are performed such as jump, dash, or run, while Axis Mappings are float values that are assigned that will return a floating-point value based on the keypress of the user. This is more relevant in the case of gamepad controllers or VR controllers, where the analog thumb stick comes into play. In that case, it would return the floating value of the state of the thumb stick, which is very important for managing player movement or related functionalities.

18. Rename **NewAxisMapping_0** to **MoveForward**.
19. In the **MoveForward** section, click the drop-down menu and select **W**.

20. Click the **+** icon next to the **MoveForward** icon to add another field.
21. Set the new field as **S**. Set its scale to **-1.0** (since we want to move backward with the **S** key).
22. Create another axis mapping by repeating *Step 18*, name it **MoveRight**, and add two fields – **A** with **-1.0** for the scale and **D** with **1.0** for the scale:



Figure 3.8: Movement Axis Mappings

23. Open **BP_MyTPC** and click the **Event Graph** tab:



Figure 3.9: Event Graph tab

24. *Right-click* anywhere inside the graph, type **MoveForward**, and select the first node option:

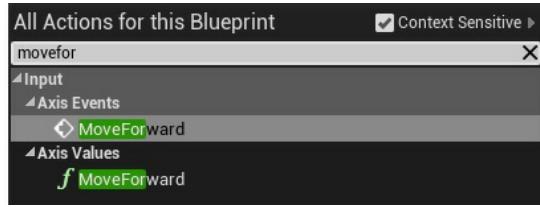


Figure 3.10: MoveForward Axis Event

25. *Right-click* inside the graph, search for **Get Control Rotation**, and select the first node option.

Note

*Since the camera associated with a player can choose not to show the pawn's yaw, roll, or pitch, the **Get Control Rotation** gives the pawn full aim rotation. This is useful in many calculations.*

26. *Left-click and drag from **Return Value** of the **Get Control Rotation** node, search for **Break Rotator**, and select it.*
27. *Right-click inside the graph, search for **Make Rotator**, and select the first node option.*

28. Connect the **Z (Yaw)** node from **Break Rotator** to the **Z (Yaw)** node of the **Make Rotator** node.

Note

Make Rotator create a rotator with the Pitch, Roll, and Yaw values, while the break rotator splits a rotator into its components (Roll, Pitch, and Yaw).

29. *Left-click and drag from Return Value* of the **Make Rotator** node, search for **Get Forward Vector**, and select it.
30. *Left-click and drag from Return Value* of the **Get Forward Vector** node, search for **Add Movement Input**, and select it.
31. Connect the **Axis Value** node from the **InputAxis MoveForward** node to the **Scale Value** node in the **Add Movement Input** node.
32. Finally, connect the white **Execution** pin from the **InputAxis MoveForward** node to the **Add**

Movement Input node.

33. *Right-click* inside the graph, search for **InputAxis MoveRight**, and select the first node option.
34. *Left-click* and drag from **Return Value** of the **Make Rotator** node, search for **Get Right Vector**, and select it.
35. *Left-click* and drag from **Return Value** of the **Get Right Vector** node, search for **Add Movement Input**, and select it.
36. Connect the **Axis Value** pin from the **InputAxis MoveRight** node to the **Scale Value** pin in the **Add Movement Input** node we created in the previous step.
37. Finally, connect the **white Execution** pin from the **InputAxis MoveRight** node to the **Add Movement Input** node we added in *Step 36*:

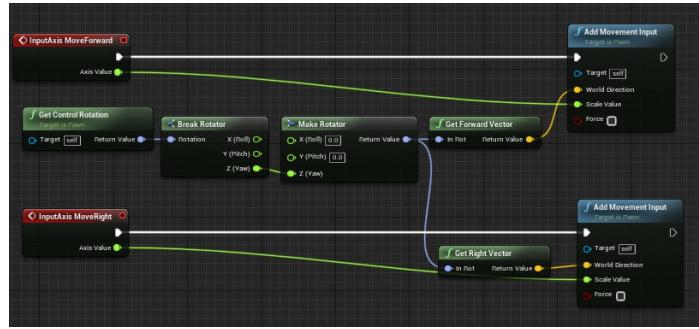


Figure 3.11: Movement logic

38. Now, head to the **Viewport** tab. Here, you will see that the character's front is not pointing in the direction of the arrow and that the character is displaced above the capsule component. Click on the **Mesh** component and select the object translation node located at the top of the viewport. Then, drag the arrows on the Mesh to 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 3.12: Translation Rotation and Scale Selector section

Once the character is aligned in the capsule, it will appear as the following screenshot:



Figure 3.13: Mesh adjusted within the capsule component

39. In the **Toolbar** menu, press the **Compile** button and then **Save**.
40. Go back to the map tab and press the **Play** button to view your character in-game. Use the *W, A, S, and D* keys to move around.

Note

*You can locate the completed exercise code files on GitHub, in the **Chapter03 -> Exercise3.02** directory, at the following link:
<https://packt.live/3keGxIU>.*

*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.

By completing this exercise, you are now able to understand how to extend C++ code with Blueprints, and why that is favorable in many situations for the developer. You also learned how to add input mappings and how they are used to drive player-related input logic.

In the activity for this chapter, you will be combining the skills you have gained from the previous exercises of this chapter and extending the project you completed in *Activity 2.01, Linking Animations to a Character* activity of *Chapter 2, Working with Unreal Engine*. This will allow you to build on your own created Blueprint and see how that maps to real-world scenarios.

Activity 3.01: Extending the C++ Character Class with Blueprint in the Animation Project

Now that you've created a C++ class and extended it with Blueprints, it is time to bring both concepts together in a real-world scenario. In this activity, your aim is to make our character from *Activity 2.01, Mixamo Character Animation*, which can be found in *Chapter 2, Working with Unreal Engine*, to jump using the *spacebar* key on your keyboard.

However, you need to create the **Character** class from scratch in C++ and then later extend it with Blueprint to reach the final goal.

The following steps will help you complete this activity:

1. Open the project from *Activity 2.01, Mixamo Character Animation*.
2. Create a **Character** class in C++ that will initialize the character variables, including the camera associated with the player.
3. Map the Jump input to the *spacebar* key in the project settings.
4. Extend the created C++ class with a blueprint to add the associated assets and jump functionality.

Expected Output:

The character should be able to jump when you press the *spacebar* key. The level should use the Blueprint that extends the C++ **Character** class:

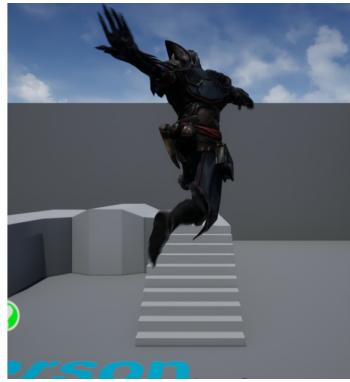


Figure 3.14: Ganfault jump activity expected output

Note

The solution to this activity can be found at:

<https://packt.live/338jEBx>.

By completing this activity, you've understood scenarios where C++ code is extended in Blueprints to implement functionalities and logic. This combination of C++ and Blueprints is the most powerful tool game developers possess to create masterful and unique games within Unreal Engine.

Summary

In this chapter, you learned how to create a C++ **Character** class, add initializer code to it, and then use Blueprints to extend it to set up assets and add additional code.

The result obeys the C++ code, as well as the Blueprint code, and can be used in any purposeful scenario.

You also learned how to set up Axis Mappings mapped to the *W*, *A*, *S*, and *D* keys to move players (which is the default movement mapping in many games). You also learned how to make the character jump within the game.

In the next chapter, you will explore Input Mapping in-depth and how to use the Mobile Previewer within Unreal Editor. This will help you create games with solid inputs mapped to game and player logic. It will also allow you to quickly test what your game will look and feel like on a mobile, all within Unreal Editor.

4. Player Input

Overview

This chapter will tackle the subject of player input. We will learn how to associate a keypress or touch input from a touch-enabled device with an in-game action such as jumping or moving.

*By the end of this chapter, you will know about **Action Mappings** and **Axis Mappings**, how to create and modify them, how to listen to each of those mappings, how to execute in-game actions when they're pressed and released, and how to preview your game as if you were playing on a mobile device.*

Introduction

In the previous chapter, we created our C++ class that inherits from the **Character** class and added all the necessary **Actor** components to be able to see the game from that character's perspective, as well as being able to see the character itself. We then created a **Blueprint** class that inherits from that C++ class, in order to visually set up all its necessary components. We also learned briefly about Action and Axis Mappings.

In this chapter, we will be going more in-depth on these topics, as well as covering their C++ usage. We will learn about how player input works in UE4, how the engine handles input events (*key presses and releases*), and how we can use

them to control logic in our game.

Let's start this chapter by getting to know how UE4 abstracts the keys pressed by the player to make it easier for you to be notified of those events.

Note

*In this chapter, we will be using an alternative version of the **Character** blueprint we created, called **BP_MyTPC**, in the previous chapter. This chapter's version will have the default UE4 Mannequin mesh and not one from Mixamo.*

Input Actions and Axes

Player input is the thing that distinguishes video games from other entertainment media: the fact that they're interactive. For a video game to be interactive, it must take into account the player's input. Many games do this by allowing the player to control a virtual character that acts upon the virtual world it's in, depending on the keys and buttons that the player presses, which is exactly what we'll be doing in this chapter.

Most game development tools nowadays allow you to abstract keypresses into Actions and Axes, which allow you to associate a name (for example, *Jump*) with several different player inputs (pressing a button, flicking a thumbstick, and so on). The difference between Actions and Axes is that Actions are used for binary inputs (inputs that can either be pressed or released, like the keys on the keyboard), while Axes are used for inputs that are scalar or continuous (that is, that can have a range of values, like thumbsticks, which can go from **-1** to **1** on both the *x* and *y* axes).

For instance, if you're making a racing game where the further you pull down the gamepad's right trigger button, the more

the car accelerates, that would be an **Axis**, because its value can range from **0** to **1**. However, if you wanted to allow the player to pause the game, that would be an Action, because it only requires knowing whether or not the player has pressed a certain key.

Usually, it's not a very good idea to have the player character jump when the player presses the *Spacebar* key explicitly, but instead to have the player jump when the *Jump* action is pressed. This *Jump* action can then have its associated keys edited elsewhere so that both developers and players can easily change which key causes the player character to jump. This is how UE4 allows you to specify player input events (although you can also listen to explicit keypresses as well, this is usually not the best way to go).

Open your UE4 project and go to the **Project Settings** window. You can do this by either clicking **Edit** in the top-left corner of the editor and then selecting **Project Settings...**, or by clicking **Settings** in the editor **Toolbar** and then selecting **Project Settings...**

This window will allow you to modify several settings related to your project, in a wide variety of categories. If you scroll down the left edge of **Project Settings**, you should find the **Input** option under the **Engine** category, which will take you to your project's input settings. Click this option.

When you do, you should see the input settings at the right edge of the window, where you'll be able to access your project's **Action Mappings** and **Axis Mappings**, among other things:

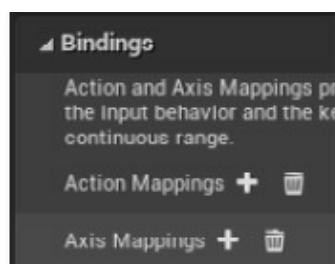


Figure 4.1: The Action and Axis Mappings available in the Input settings window

The **Action Mappings** property allows you to specify a list of actions in your project (for example, the *Jump* action) and their corresponding keys (for example, the *Spacebar* key).

Axis Mappings allows you to do the same thing, but for keys that do not have a binary value (either pressed or released) but instead have a continuous value, like the thumbsticks on a controller whose values can go from **-1** to **1** on the *x* and *y* axes, or the trigger buttons on a controller whose values can go from **0** to **1**.

For example, consider an Xbox One controller, which can be broken down into the following:

- **Left analog stick**, usually used for controlling movement in games
- **Dpad**, which can be used for controlling movement, as well as having a variety of other uses
- **Right analog stick**, usually used for controlling the camera and view perspective
- **Face buttons (X, Y, A, and B)**, which can have various uses depending on the game, but usually allow the player to perform actions in the game world
- **Bumpers and Triggers (LB, RB)**, which can be used for controlling movement, as well as having a variety of other uses

RB, LT, and RT), which can be used for actions such as aiming and shooting or accelerating and braking

You can also set up binary keys as axes if you want to; for instance, set up the player character's movement for both a gamepad thumbstick (which is a continuous key whose value goes from **-1** to **1**) and two binary keys on the keyboard (**W** and **S**).

We'll be taking a look at how to do this in this chapter.

When we generated the **Third Person** template project back in *Chapter 1, Unreal Engine Introduction*, it came with some inputs already configured, which were **W**, **A**, **S**, and **D** keys, as well as the **left thumbstick** for movement and the **Space Bar** key and **gamepad bottom face** button for jumping.

Let's now add new **Action** and **Axis Mappings** in the next exercise.

Exercise 4.01: Creating the Jump Action and Movement Axes

In this exercise, we'll be adding a new **Action Mapping** for the *Jump* action and a couple of new **Axis Mappings** for the *Movement* action.

To achieve this, follow these steps:

1. Open the **Input Settings** menu.
2. Press the **+** icon to the right of the **Action Mappings** property to create a new **Action Mapping**:

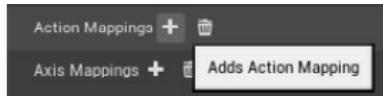


Figure 4.2: Adding a new Action Mapping

3. When you do so, you should see a new **Action Mapping** called **NewActionMapping_0** mapped to the **None** key (*meaning it's not mapped to any key*):



Figure 4.3: The default settings of a new Action Mapping

4. Change the name of this mapping to **Jump** and the key associated with it to the **Spacebar** key.

To change the key mapped to this action, you can click the drop-down property currently set to the **None** key, type **Space Bar**, and select the first

option:

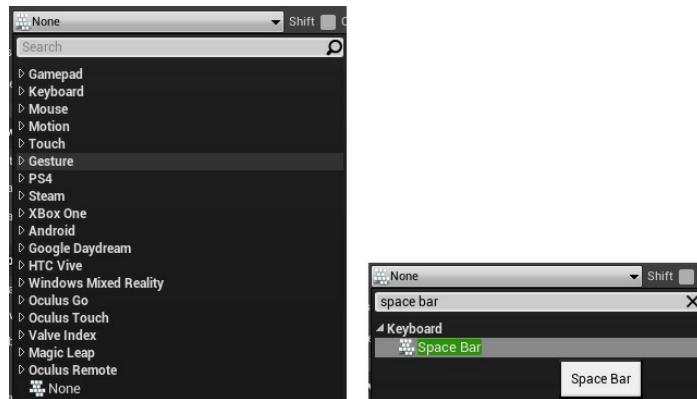


Figure 4.4: The key drop-down menu (top) where the Space Bar key is being selected (bottom)

5. You can specify whether or not you want this action to be executed when the player presses the specified key while holding one of the modifier keys – **Shift**, **Ctrl**, **Alt**, or **Cmd**, by checking each of their appropriate checkboxes. You can also remove this key from this **Action Mapping** by clicking the **X** icon:



Figure 4.5: The key drop-down menu and the options to specify modifier keys and removing this key from this Action Mapping

6. To add a new key to an **Action**

Mapping, you can simply click the **+** icon next to the name of that **Action Mapping**, and to remove an **Action Mapping** altogether, you can click the **x** icon next to it:



Figure 4.6: The name of the Action Mapping, with the + and x icons next to it

Let's now use a controller button to map to this **Action Mapping**.

Because most gamepads have the same keys in very similar places, UE4 abstracts most of their keys to generic terms using the **Gamepad** prefix.

7. Add a new key to this **Action Mapping** and set that new key to be the **Gamepad Face Button Bottom** key. If you're using an Xbox controller, this will be the **A** button, and if you're using a PlayStation controller, this will be the **X** button:



Figure 4.7: The Gamepad Face Button Bottom key added to the Jump Action Mapping

Now that we've set up our **Jump Action Mapping**, let's set up our **Movement Axis Mapping**.

8. Click the **+** icon next to the **Axis Mappings** property to add a new **Axis Mapping**. This new **Axis Mapping** will be used to move the character left and right. Name it **MoveRight** and assign to it the **Gamepad Left Thumbstick X-Axis** key, so that the player can use the *x* axis of the left thumbstick to move the character left and right:

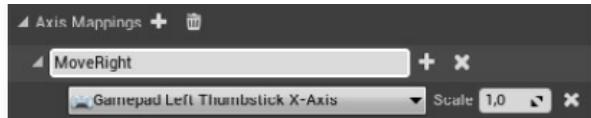


Figure 4.8: The MoveRight Axis Mapping with the Gamepad Left Thumbstick X-Axis key associated with it

If you look to the right of the key we assigned, instead of the modifier keys, you should see the **Scale** property of

that key. This property will allow you to invert an axis so that the player moves left when the player tilts the thumbstick to the right and vice versa, as well as increasing or decreasing the sensitivity of an axis.

To allow the player to move right and left using keyboard keys (which are either pressed or released and don't have a continuous value, unlike thumbsticks), we'll have to add two keys with inverted values on their scale.

Add two more keys to this **Axis Mapping**, the first being the **D** key, with a **Scale** of **1**, and the second one being the **A** key, with a **Scale** of **-1**. This will cause the character to move right when the player presses the **D** key and to move left when the player presses the **A** key:

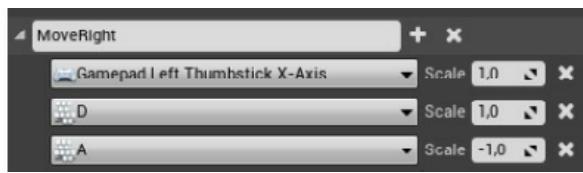


Figure 4.9: The MoveRight Axis Mapping with both the Gamepad and keyboard keys

9. After doing this, add another **Axis Mapping** with the name of **MoveForward** with the **Gamepad Left Thumbstick Y-Axis**, **W**, and **S** keys, the latter having a **Scale** of **-1**. This axis will be used to move the character forward and backward:



Figure 4.10: The MoveForward Axis Mapping

And with those steps completed, we've completed this chapter's first exercise, where you've learned how you can specify **Action** and **Axis Mappings** in UE4, allowing you to abstract which keys are responsible for which in-game actions.

Let's now take a look at how UE4 handles player input and processes it within the game.

Processing Player Input

Let's think about a situation where the player presses the **Jump** action, which is associated with the **Spacebar** key, to get the player character to jump. Between the moment the player presses the **Spacebar** key and the moment the game makes the player character jump, quite a few things have to connect those two events.

Let's take a look at all the steps necessary that lead from one event to the other:

1. **Hardware Input:** The player presses the **Spacebar** key. UE4 will be listening to this keypress event.
2. The **PlayerInput** class: After the key is pressed or released, this class will translate that key into an action or axis. If there is an action or axis associated with that key, it will notify all classes that are listening to the action that it was just pressed, released, or updated. In this case, it will know that the **Spacebar** key is associated with the *Jump* action.
3. The **Player Controller** class: This is the first class to receive these events, given that it's used to represent a player in the game.
4. The **Pawn** class: This class (and consequently the **Character** class, which inherits from it) can also listen to those events, as long as they are possessed by a Player Controller. If so, it will receive these events after that class. In this chapter, we will be using

our **Character** C++ class to listen to action and axis events.

Now that we know how UE4 handles player inputs, let's take a look at the **DefaultInput.ini** file and how it works.

DefaultInput.ini

If you go to your project's directory, using File Explorer, and then open its **Config** folder, you'll find some **.ini** files in it, one of which should be the **DefaultInput.ini** file. As the name suggests, this file holds the main settings and configuration for input-related properties.

In the first exercise of this chapter, where we edited the project's **Input** settings, what was happening, in reality, was that the editor was writing to and reading from the **DefaultInput.ini** file.

Open this file in a text editor of your choice. It contains many properties, but the ones we want to take a look at now are the list of **Action Mappings** and **Axis Mappings**. Near the end of the file, you should see, for instance, the *Jump* action being specified in this file:

```
+ActionMappings=
(ActionName="Jump", bShift=False, bCtrl=False
, bAlt=False, bCmd=False, Key=SpaceBar)

+ActionMappings=
(ActionName="Jump", bShift=False, bCtrl=False
,
bAlt=False, bCmd=False, Key=Gamepad_FaceButton_Bottom)
```

You can also see some axes being specified, such as the

MoveRight axis:

```
+AxisMappings=
(AxisName="MoveRight", Scale=1.000000,
Key=Gamepad_LeftX)

+AxisMappings=
(AxisName="MoveRight", Scale=1.000000, Key=D)

+AxisMappings=
(AxisName="MoveRight", Scale=-1.000000, Key=A
)
```

Instead of editing the project's **Input Settings**, you can directly edit this file to add, modify, and remove **Action Mappings** and **Axis Mappings**, although this isn't a very user-friendly way to do so. Keep in mind that this file will also be available when you package your project to an executable file, which means that the player will be able to edit this file to their liking.

Let's now see how we can listen to **Action Mappings** and **Axis Mappings** in C++ in the next exercise.

Exercise 4.02: Listening to Movement Actions and Axes

In this exercise, we will register the actions and axes we created in the previous section with our character class by binding those actions and axes to specific functions in our character class using C++.

For a **Player Controller** or **Character** to listen to Actions and Axes, the main way to do that is by registering the **Action** and **Axis** delegates using the

SetupPlayerInputComponent function. The **MyThirdPersonChar** class should already have a declaration and an implementation for this function. Let's have our character class listen to those events by following these steps:

1. Open the **MyThirdPersonChar** class header file in Visual Studio and make sure there's a declaration for a **protected** function called **SetupPlayerInputComponent** that returns nothing and receives a **class UInputComponent*** **PlayerInputComponent** property as a parameter. This function should be marked as both **virtual** and **override**:

```
virtual void  
SetupPlayerInputComponent(c  
lass UInputComponent*  
PlayerInputComponent)  
override;
```

2. Open this class's source file and make sure this function has an implementation:

```
void  
AMyThirdPersonChar::SetupPl  
ayerInputComponent(class
```

```
UInputComponent*  
PlayerInputComponent)  
{  
}
```

3. Inside its implementation, start by calling the **PlayerInputComponent** property's **BindAction** function. This function will allow this class to listen to a specific action, in this case, the **Jump** action. It receives the following parameters:

1. **FName ActionName** – The name of the action we want to listen to; in our case, the **Jump** action.
 2. **EInputEvent InputEvent** – The specific key event we want to listen to, which can be pressed, released, double-clicked, and so on. In our case, we want to listen to the pressed event, which we can specify by using the **IE_Pressed** value.
 3. **UserClass* Object** – The

object that the callback function will be called on; in our case, the **this** pointer.

4. **FInputActionHandlerSignature**::TUObjectMethodDelegate< UserClass >::FMethodPtr Func –

This property is a bit wordy, but is essentially a pointer to the function that will be called when this event happens, which we can specify by typing & followed by the class's name, followed by ::, followed by the function's name. In our case, we want this to be the existing **Jump** function belonging to the **Character** class, so we'll specify it with **&ACharacter::Jump**:

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

Note

*All functions used to listen to actions must receive no parameters unless you use **Delegates**, which are outside the scope of this book.*

4. In order to tell our character to stop jumping, you'll have to duplicate this line and then change the new line's input event to **IE_Released** and the function that's called to be the **Character** class's **StopJumping** function instead:

```
PlayerInputComponent -  
>BindAction("Jump",  
IE_Released, this,  
&ACharacter::StopJumping);
```

5. Because we'll be using the **InputComponent** class, we'll need to include it:

```
#include  
"Components/InputComponent.  
h"
```

6. Now that we're listening to the **Jump** action and having the character jump

when that action is executed, let's move on to its movement. Inside the class's header file, add a declaration for a **protected** function called **MoveRight**, which returns nothing and receives a **float Value** parameter. This is the function that will be called when the value of the **MoveRight** axis is updated:

```
void MoveRight(float  
Value);
```

7. In the class's source file, add this function's implementation, where we'll start by checking whether the **Controller** property is valid (not a **nullptr**) and whether the **Value** property is different than **0**:

```
void  
AMyThirdPersonChar::MoveRig  
ht(float Value)  
{  
    if (Controller != nullptr  
&& Value != 0.0f)  
    {  
    }  
}
```

```
}
```

8. If both these conditions are true, we'll want to move our character using the **AddMovementInput** function. One of the parameters of this function is the direction in which you want the character to move. To calculate this direction, we'll need to do two things:

1. Get the camera's rotation on the z axis (yaw), so that we move the character relative to where the camera is looking. To achieve this, we can create a new **FRotator** property with a value of **0** for pitch (rotation along the y axis) and roll (rotation along the x axis) and the value of the camera's current yaw for the property's yaw. To get the camera's yaw value, we can call the Player Controller's **GetControlRotation** function and then access its **Yaw** property:

```
const FRotator  
YawRotation(0,
```

```
Controller->  
GetControlRotation()  
.Yaw, 0);
```

Note

*The **FRotator** property's constructor receives the **Pitch** value, then the **Yaw** value, and then the **Roll** value.*

2. Get the resulting rotation's right vector and store it in an **FVector Direction** property. You can get a rotator's Right Vector by calling the **KismetMathLibrary** object's **GetRightVector** function. A rotator or vector's right vector is simply its perpendicular vector that points to its right. The result of this will be a vector that points to the right of where the camera is currently facing:

```
    const FVector  
    Direction =  
        UKismetMathLibrary::  
        GetRightVector(YawRo  
        tation);
```

We can now call the **AddMovementInput** function, passing as parameters the **Direction** and **Value** properties:

```
AddMovementInput(Dir  
ection, Value);
```

9. Because we'll be using both the **KismetMathLibrary** and **Controller** objects, we'll need to include them at the top of this source file:

```
#include  
"Kismet/KismetMathLibrary.h"  
"  
  
#include  
"GameFramework/Controller.h"  
"
```

10. After listening to the **Jump** action, inside this class's

SetupPlayerInputComponent function, listen to the **MoveRight** axis by calling the **PlayerInputComponent** property's **BindAxis** function. This function is used to listen to an Axis instead of an Action, and the only difference between its parameters and the **BindAction** function's parameters is that it doesn't need to receive an **EInputState** parameter. Pass as parameters to this function "**MoveRight**", the **this** pointer, and this class's **MoveRight** function:

```
PlayerInputComponent->BindAxis("MoveRight",  
    this,  
    &AMyThirdPersonChar::MoveRight);
```

Note

*All functions used to listen to an axis must receive a **float** property as a parameter unless you use **Delegates**, which is outside the scope of this book.*

Let's now listen to the **MoveForward**

axis in this class:

11. In the class's header file, add a similar declaration to that of the **MoveRight** function, but name it **MoveForward** instead:

```
void MoveForward(float  
Value);
```

12. In the class's source file, add an implementation to this new **MoveForward** function. Copy the implementation of the **MoveRight** function into this new implementation, but replace the call to the **KismetMathLibrary** object's **GetRightVector** function with the call to its **GetForwardVector** function. This will use the vector representing the direction the camera is facing instead of its Right Vector, which faces its right:

```
void  
AMyThirdPersonChar::MoveFor  
ward(float Value)  
  
{  
  
    if (Controller != nullptr
```

```

    && Value != 0.0f)

{

    const FRotator
    YawRotation(0, Controller->
    GetControlRotation().Yaw,
    0);

    const FVector Direction
    =
    UKismetMathLibrary::GetForwardVector(YawRotation);

    AddMovementInput(Direction, Value);

}
}

```

13. In the **SetupPlayerInputComponent** function's implementation, duplicate the line of code that listens to the **MoveRight** axis and replace the first parameter with "**MoveForward**" and the last parameter with a pointer to the **MoveForward** function:

```

PlayerInputComponent-
>BindAxis("MoveForward",
this,

```

```
&AMyThirdPersonChar ::MoveFo  
rward);
```

14. Now compile your code, open the editor, and open your **BP_MyTPS** Blueprint asset. Delete the **InputAction Jump** event, as well as the nodes connected to it. Do the same for the **InputAxis MoveForward** and **InputAxis MoveRight** events. We will be replicating this logic in C++ and need to remove its Blueprint functionality, so that there are no conflicts when handling input.
15. Now, play the level. You should be able to move the character using the keyboard's **W**, **A**, **S**, and **D** keys or the controller's left thumbstick, as well as jumping with the **Spacebar** key or **gamepad face button bottom**:



Figure 4.11: The player character moving

After following all these steps, you have concluded this exercise. You now know how to listen to **Action** and **Axis** events using C++ in UE4.

Note

*Instead of listening to a specific **Action** or **Axis**, you can listen to a specific key by using the **PlayerInputComponent** property's **BindKey** function. This function receives the same parameters as the **BindAction** function, except for the first parameter, which should be a key instead of an **FName**. You can specify keys by using the **EKeys** enum followed by ::.*

Now that we've set up all the logic necessary to have our character move and jump, let's add the logic responsible for rotating the camera around our character.

Turning the camera around the character

Cameras are an extremely important part of games, as they dictate what and how the player will see your game throughout the play session. When it comes to third-person games, which is the case for this project, the camera allows you not only to see the world around them but also the character you're controlling. Whether the character is taking damage, falling, or something else, it's important for the player to always know the state of the character they are controlling and to be able to have the camera face the direction they choose.

Much like with every modern, third-person game, we will

always have the camera rotate around our player character. To have our camera rotate around our character, after setting up the **Camera** and **Spring Arm** components in *Chapter 2*, *Working with Unreal Engine*, let's continue by adding two new **Axis Mappings**, the first one called **Turn**, which is associated with the **Gamepad Right Thumbstick X-Axis** and **MouseX** keys, and the second one called **LookUp**, which is associated with the **Gamepad Right Thumbstick Y-Axis** and **MouseY** keys, this latter key having a scale of **-1**.

These **Axis Mappings** will be used to have the player look right and left as well as up and down, respectively:



Figure 4.12: The Turn and LookUp Axis Mappings

Let's now add the C++ logic responsible for turning the camera with the player's input.

Go to the **MyThirdPersonChar** class's **SetupPlayerInputComponent** function implementation and duplicate either the line responsible for listening to the **MoveRight** axis or the **MoveForward** axis twice. In the first duplicated line, change the first parameter to "**Turn**" and the last parameter to the **Pawn** class's **AddControllerYawInput** function, while the second duplicated line should have the first parameter be "**LookUp**" and the last parameter be the **Pawn** class's **AddControllerPitchInput** function.

These two functions are responsible for adding rotation input around the z (turning left and right) and y (looking up and down) axes, respectively:

```
PlayerInputComponent->BindAxis("Turn",
this, &APawn::AddControllerYawInput);

PlayerInputComponent->BindAxis("LookUp",
this, &APawn::AddControllerPitchInput);
```

If you compile the changes made in this section, open the editor, and play the level, you should now be able to move the camera by rotating the mouse or by tilting the controller's right thumbstick:



Figure 4.13: The camera is rotated around the player

And that concludes the logic to rotate the camera around the player character with the player's input. In the next exercise, we'll take a broad look at the topic of mobile platforms such as Android and iOS.

Mobile platforms

Thanks to recent advancements in technology, the majority of the population now has access to affordable mobile devices such as smartphones and tablets. These devices, although small, still have quite a bit of processing power and can now do many things that bigger devices such as laptops and desktop computers do. One of those things is playing video games.

Because mobile devices are much more affordable and

versatile than other devices you can play video games on, you have a lot of people playing games on them. For this reason, it's worth considering developing video games for mobile platforms such as Android and iOS, the two biggest mobile app stores.

Let's now take a look at how to preview our game on a virtual mobile device in the next exercise.

Exercise 4.03: Previewing on Mobile

In this exercise, we'll be playing our game using **Mobile Preview**, to see what it's like to play our game on a mobile device. Before we can do this, we have to go to the **Android Platform** settings.

Have a look at the following steps:

1. Open the **Project Settings** window and scroll down its left edge until you find the **Android** option beneath the **Platforms** category.
Click that option. You should see the following to the right of the categories:

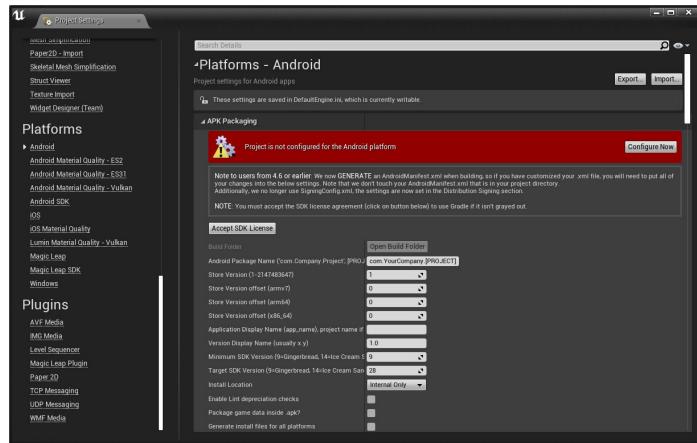


Figure 4.14: The Android Platform window warning that the project currently isn't configured for that platform

2. This warning is letting you know that the project has not yet been configured for Android. To change that, click the **Configure Now** button inside the *red warning*. When you do, it should be turned into a green warning, letting you know that the platform is configured:

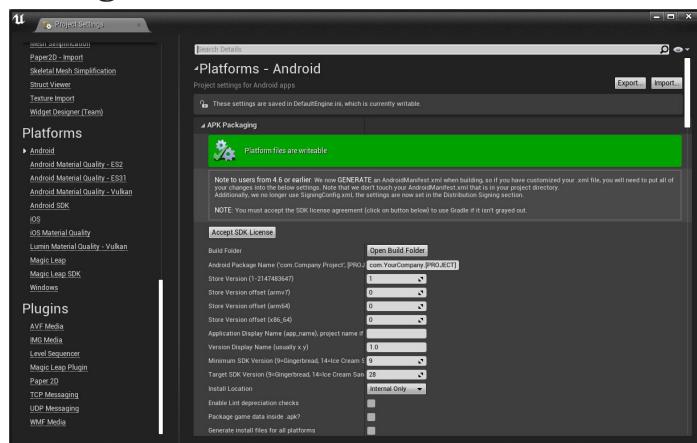


Figure 4.15: The Android platform window notifying you that the project is configured for this platform

3. After you've done this, you can close **Project Settings**, click the arrow next to the **Play** button in the editor's toolbar, and select the **Mobile Preview** option you see available:

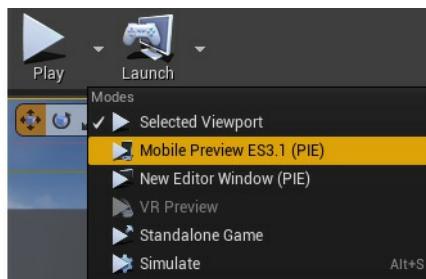


Figure 4.16: The Mobile Preview option under the Play button

This will cause the engine to start loading this preview, as well as compiling all the necessary shaders, which should take a few minutes.

When it's done, you should see the following:



Figure 4.17: The Mobile Preview window playing the game as if on an Android device

This preview should look similar to the normal preview inside the editor with a couple of notable differences:

- The visual fidelity has been lowered. Because mobile platforms don't have the same type of computing power as PCs and consoles, the visual quality is lowered to take that into account. On top of that, some rendering features available in high-end platforms are simply not supported in mobile platforms.
- Two added virtual joysticks at the *lower-left* and *lower-right* corner of the screen, which work similarly to those of a controller, where the left joystick controls the character's movement and the right joystick

controls the camera's rotation.

This window acts as a mobile screen where your mouse is your finger, so if you press and hold the left joystick using the left mouse button, and then drag it, that will cause the joystick to move on the screen and consequently make the character move, as shown in the following screenshot:



Figure 4.18: The character is moved using the left virtual joystick

And with that, we conclude this chapter by learning how to preview our game on the Android mobile platform and verify that its inputs are working.

Let's now jump into the next exercise, where we'll be adding touch input that causes the player character to jump.

Exercise 4.04: Adding Touchscreen Input

In this exercise, we'll be continuing from our previous exercise by making it so that the player character starts jumping when the player tabs the screen if they're playing on a touchscreen device.

To add touchscreen input to our game, follow these steps:

1. Go to the **MyThirdPersonChar** class's header file and add two declarations for protected functions that return nothing and receive the **ETouchIndex::Type** **FingerIndex** and **FVector Location** parameters, the first one of which indicates the index of the finger that touched the screen (whether it was the first, second, or third finger to touch the screen) and the second one, which indicates the position that was touched on the screen. Name one of these functions **TouchBegin** and the other one **TouchEnd**:

```
void
TouchBegin(ETouchIndex::Type
    FingerIndex, FVector
Location);

void
TouchEnd(ETouchIndex::Type
    FingerIndex, FVector
```

```
Location);
```

2. In the **MyThirdPersonChar** class's source file, add the implementation of both these functions, where the **TouchBegin** function will call the **Jump** function and the **TouchEnd** function will call the **StopJumping** function. This will cause our character to start jumping when the player touches the screen and to stop jumping when they stop touching the screen:

```
void  
AMyThirdPersonChar::TouchBe  
gin(ETouchIndex::Type  
FingerIndex, FVector  
Location)  
{  
    Jump();  
}  
  
void  
AMyThirdPersonChar::TouchEn  
d(ETouchIndex::Type  
FingerIndex, FVector  
Location)  
{
```

```
    StopJumping();  
}  
  
3. Go to the
```

SetupPlayerInputComponent function's implementation and add two calls to the **BindTouch** function of **PlayerInputComponent**, which will bind the event of the screen being touched to a function. This function receives the same parameters as the **BindAction** function except for the first one, **ActionName**. In the first function call, pass as parameters the input event **IE_Pressed**, the **this** pointer, and this class's **TouchBegin** function, and in the second call, pass the input event **IE_Released**, the **this** pointer, and this class's **TouchEnd** function:

```
PlayerInputComponent -  
    >BindTouch(IE_Pressed,  
    this,  
    &AMyThirdPersonChar::TouchB  
    egin);  
  
PlayerInputComponent -  
    >BindTouch(IE_Released,
```

```
this,  
&AMyThirdPersonChar::TouchE  
nd);
```

4. Preview the game using **Mobile Preview**, just like we did in the previous exercise. If you use the left mouse button to click the middle of the screen, the player character should jump:



Figure 4.19: The character jumping after clicking the middle of the screen

And with that, we conclude the logic that will make our character jump as long as the player is touching the screen if they're playing on a touchscreen device. Now that we've learned how to add inputs to our game and associate those inputs with in-game actions such as jumping and moving the player character, let's consolidate what we've learned in this chapter by adding a new **Walk** action to our game from start to finish in the next activity.

Activity 4.01: Adding Walking Logic to Our Character

In the current game, our character runs by default when we use the movement keys, but we need to reduce the character's speed and make it walk.

So, in this activity, we'll be adding logic that will make our character walk when we move it while holding the **Shift** key on the keyboard or the **Gamepad Face Button Right** key (**B** for the Xbox controller and **0** for PlayStation controller). Further, we will preview it on a mobile platform as well.

To do this, follow these 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.
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**.

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.
5. 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%.
6. Bind the **walk** action to the **BeginWalking** function when that action is pressed, and to the **StopWalking** function when it is released.

After following these steps, you should be able to have your character walk, which decreases its speed and slightly changes its animation, by holding

either the keyboard's *Left Shift* key or the controller's *Face Button Right* button.



Figure 4.20: The character running (left) and walking (right)

7. 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 4.21: The character walking in the mobile preview

And that concludes our activity. Our character should now be able to walk slowly as long as the player is holding the **Walk** action.

Note

The solution to this activity can be found at:

<https://packt.live/338jEBx>.

Summary

In this chapter, you've learned how to add, remove, and modify **Action Mappings** and **Axis Mappings**, which give you some flexibility when determining which keys trigger a specific action or axis, how to listen to them, and how to execute in-game logic when they're pressed and released.

Now that you know how to handle the player's input, you can allow the player to interact with your game and offer the agency that video games are so well known for.

In the next chapter, we'll start making our own game from scratch. It'll be called **Dodgeball** and will consist of the player controlling a character trying to run away from enemies that are throwing dodgeballs at it. In that chapter, we will have the opportunity to start learning about many important topics, with a heavy focus on collisions.

5. Line Traces

Overview

This chapter will be the start of a new game project called Dodgeball, where we will be creating a game from scratch that features mechanics based on collision concepts. In this chapter, you will modify the Third Person Template project to give it a top-down perspective. Then, you will be introduced to line traces, a key concept in game development, and learn about their potential and use cases.

By the end of this chapter, you will be able to use UE4's built-in Line Trace feature (also known as Raycasts or Raytraces in other game development tools) by executing different types of line traces; creating your own trace channels; and modifying an object's response to each trace channel.

Introduction

In the previous chapters, we learned how we can reproduce the Third Person Template project offered to us by the Unreal Engine team in order to understand some of the basic concepts of UE4's workflow and framework.

In this chapter, you will start creating another game from scratch. In this game, the player will control a character from a top-down point of view (*similar to games such as Metal Gear Solid 1, 2, and 3*). A top-down perspective implies that the player controls a character that is seen as if it was being looked down upon, usually with the camera rotation being

fixed (the camera doesn't rotate). In our game, the player character must go from point A to point B without being hit by dodgeballs that are being thrown at the player by the enemies that are spread throughout the level. The levels in this game will be maze-like in nature, and the player will have multiple paths to choose from, all of which will have enemies trying to throw dodgeballs at the player.

The specific topics we'll be approaching in this chapter will be Line Traces (Single and Multi), Sweep Traces, Trace Channels, and Trace Responses. In the first section, we begin by getting to know what *collision* is in the world of video games.

Collision

A collision is basically a point at which two objects come into contact with each other (for example, two objects colliding, an object hitting a character, a character walking into a wall, and so on). Most game development tools have their own set of features that allow for collision and physics to exist inside the game. This set of features is called a **Physics Engine**, which is responsible for everything related to collisions. It is responsible for executing Line Traces, checking whether two objects are overlapping each other, blocking each other's movement, bouncing off of a wall, and much more. When we ask the game to execute or notify us of these collision events, the game is essentially asking the Physics Engine to execute it and then show us the results of these collision events.

In the **Dodgeball** game you will be building, examples of where collision needs to be taken into account include checking whether enemies are able to see the player (which will be achieved using a Line Trace, covered in this chapter), simulating physics on an object that will behave just like a dodgeball, checking whether anything is blocking the player character's movement, and much more.

Collision is one of the most important aspects of most games, so understanding it is crucial in order to get started with game development.

Before we start building our collision-based features, we will first need to set up our new **Dodgeball** project in order to support the game mechanics we will be implementing. This process starts with the steps described in the next section: *Project Setup*.

Project Setup

Let's begin this chapter by creating our Unreal Engine project:

1. **Launch** UE4. Select the **Games** project category, then press **Next**.
2. Select the **Third Person template**, then press **Next**.
3. Make sure the first option is set to **C++** and not **Blueprint**.
4. Select the location of the project according to your preference and name your project **Dodgeball**, then press **Create Project**.

When the project is done being generated, you should see the following on your screen:

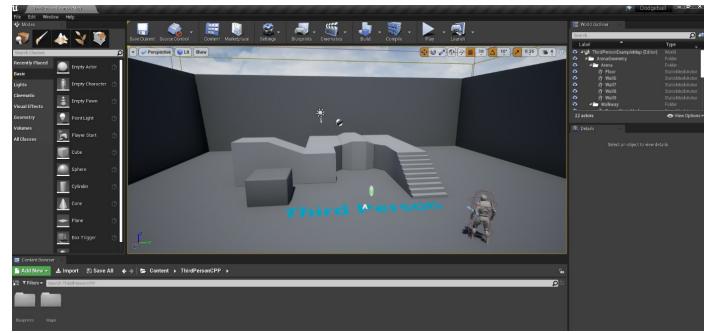


Figure 5.1: Dodgeball project loaded up

5. After the code has been generated and the project opens up, close the UE4 editor and open the files of the generated third-person Character class, **DodgeballCharacter**, in Visual Studio, as shown in the following figure:

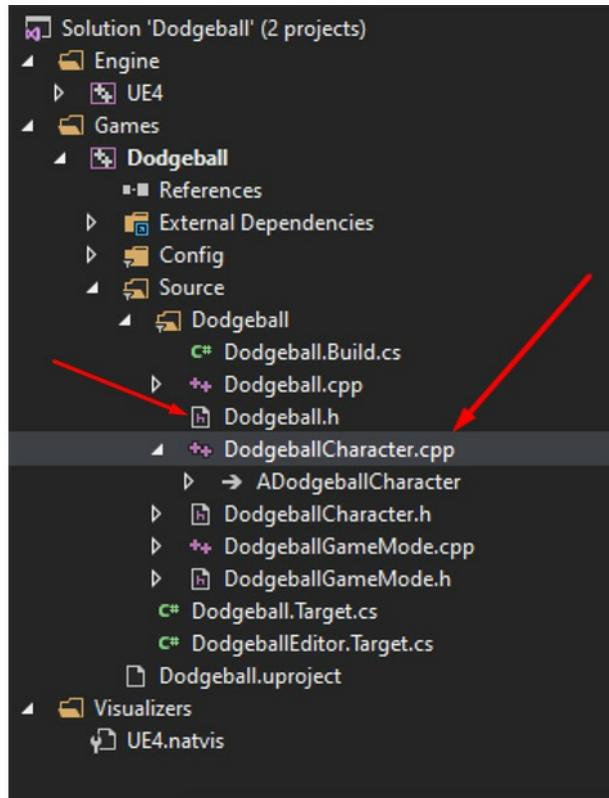


Figure 5.2: Files generated in Visual studio

As mentioned before, your project is going to have a top-down perspective. Given that we're starting this project from the Third Person Template, we'll have to change a few things before we turn this into a top-down game. This will mainly involve changing some lines of code in the existing Character class.

Exercise 5.01: Converting DodgeballCharacter to a Top-Down Perspective

In this exercise, you'll be performing the necessary changes to your generated **DodgeballCharacter** class. Remember, it currently features a third-person perspective, where the

rotation of the character is dictated by the player's input (*namely the mouse or right analog stick*).

In this exercise, you will change this to a top-down perspective, which remains the same regardless of the player's input, where the camera always follows the character from above.

The following steps will help you complete this exercise:

1. Head to the **DodgeballCharacter** class's constructor and update the **CameraBoom** properties, as mentioned in the following steps.
2. Change **TargetArmLength**, which is a property of **CameraBoom**, to **900.0f**, in order to add some distance between the camera and the player:

```
// The camera follows at
// this distance behind the
// character

CameraBoom->TargetArmLength
= 900.0f;
```
3. Next, add a line that sets the relative pitch to **-70°**, using the **SetRelativeRotation** function, so that the camera looks down at the player. The **FRotator** constructor's

parameters are the *pitch*, *yaw*, and *roll*, respectively:

```
//The camera looks down at  
the player  
  
CameraBoom-  
>SetRelativeRotation(FRotat  
or(-70.f, 0.f, 0.f));
```

4. Change **bUsePawnControlRotation** to **false**, so that the camera's rotation isn't changed by the player's movement input:

```
// Don't rotate the arm  
based on the controller
```

```
CameraBoom-  
>bUsePawnControlRotation =  
false;
```

5. Add a line that sets **bInheritPitch**, **bInheritYaw**, and **bInheritRoll** to **false**, so that the camera's rotation isn't changed by the character's orientation:

```
// Ignore pawn's pitch, yaw  
and roll
```

```
CameraBoom->bInheritPitch =
```

```
false;  
  
CameraBoom->bInheritYaw =  
false;  
  
CameraBoom->bInheritRoll =  
false;
```

After we've made these modifications, we're going to remove the character's ability to jump (we don't want the player to escape from the dodgeballs that easily) and to rotate the camera from the player's rotation input.

6. Go to the **SetupPlayerInputComponent** function in the **DodgeballCharacter**'s source file and remove the following lines of code in order to remove the ability to jump:

```
// REMOVE THESE LINES  
  
PlayerInputComponent->BindAction("Jump",  
IE_Pressed, this,  
&ACharacter::Jump);  
  
PlayerInputComponent->BindAction("Jump",  
IE_Released, this,
```

```
ACharacter::StopJumping);
```

7. Next, add the following lines in order to remove the player's rotation input:

```
// REMOVE THESE LINES

PlayerInputComponent-
>BindAxis("Turn", this,
&APawn::AddControllerYawInp
ut);

PlayerInputComponent-
>BindAxis("TurnRate", this,
&ADodgeballCharacter::TurnA
tRate);

PlayerInputComponent-
>BindAxis("LookUp", this,
&APawn::AddControllerPitchI
nput);

PlayerInputComponent-
>BindAxis("LookUpRate",
this,
&ADodgeballCharacter::LookU
pAtRate);
```

This step is optional, but in order to keep your code clean, you should remove the declarations and implementations of the **TurnAtRate**

and **LookUpAtRate** functions.

8. Finally, after you've made these changes, run your project from Visual Studio.
9. When the editor has loaded, play the level. The camera's perspective should look like this and should not rotate based on the player's input or the character's rotation:



Figure 5.3: Locked camera rotation to a top-down perspective

And that concludes the first exercise of this chapter, and the first step to your new project, **Dodgeball**.

Next, you will be creating the **EnemyCharacter** class. This character will be the enemy that throws dodgeballs at the player while the player is in view. But the question that arises here is this: how will the enemy know whether it can see the player character or not?

That will be achieved with the power of **Line Traces** (also known as **Raycasts** or **Raytraces**), which you will be looking at in the next section.

Line Traces

One of the most important features of any game development tool is its ability to execute Line Traces. These are available through the Physics Engine that the tool is using.

Line Traces are a way of asking the game to tell you whether anything stands between two points in the game world. The game will *shoot a ray* between those two points, specified by you, and return the objects that were hit (if any), where they were hit, at what angle, and much more.

In the following figure, you can see a representation of a Line Trace where we assume object **1** is ignored and object **2** is detected, due to their Trace Channel properties (further explained in the following paragraphs):

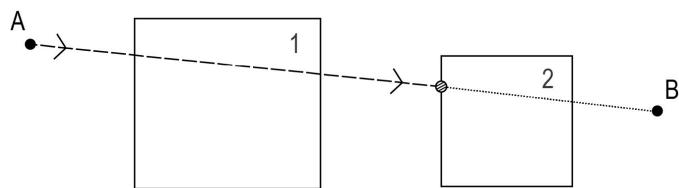


Figure 5.4: A Line Trace being executed from point A to point B

In *Figure 5.4*:

- The dashed line represents the Line Trace before it hits an object.
- The arrows represent the direction of the Line Trace.
- The dotted line represents the Line Trace after it hits an object.
- The striped circle represents the Line

Trace's impact point.

- The big squares represent two objects that are in the path of the Line Trace (objects **1** and **2**).

We notice that only object **2** was hit by the Line Trace and not object **1**, although it is also in the path of the Line Trace. This is due to assumptions made about object **1**'s Trace Channel properties, which are talked about later in this chapter.

Line Traces are used for many game features, such as:

- Checking whether a weapon hits an object when it fires
- Highlighting an item that the player can interact with when the character looks at it
- Rotating the camera around the player character automatically as it goes around corners

A common and important feature of Line Traces is **Trace Channels**. When you execute a Line Trace, you may want to check only specific types of objects, which is what Trace Channels are for. They allow you to specify filters to be used when executing a Line Trace so that it doesn't get blocked by unwanted objects. For example:

- You may want to execute a Line Trace only to check for objects that are

visible. These objects would block the **Visibility** Trace Channel. For instance, invisible walls, which are invisible pieces of geometry used in games to block the player's movement, would not be visible and therefore would not block the **Visibility** Trace Channel.

- You may want to execute a Line Trace only to check for objects that can be interacted with. These objects would block the **Interaction** Trace Channel.
- You may want to execute a Line Trace only to check for pawns that can move around the game world. These objects would block the **Pawn** Trace Channel.

You can specify how different objects react to different Trace Channels so that only some objects block specific Trace Channels and others ignore them. In our case, we want to know whether anything stands between the enemy and the player character, so that we know whether the enemy can see the player. We will be using Line Traces for this purpose, by checking for anything that blocks the enemy's line of sight to the player character, using a **Tick** event.

In the next section, we will be creating the **EnemyCharacter** class using C++.

Creating the EnemyCharacter C++ class

In our **Dodgeball** game, the **EnemyCharacter** class will constantly be looking at the player character, if they're within view. This is the same class that will later throw dodgeballs at the player; however, we'll leave that to the next chapter. In this chapter, we will be focusing on the logic that allows our enemy character to look at the player.

So, let's get started:

1. Right-click the **Content Browser** inside the editor and select **New C++ Class**.
2. Choose the **Character** class as the parent class.
3. Name the new class **EnemyCharacter**.

After you've created the class and opened its files in Visual Studio, let's add the **LookAtActor** function declaration in its **header** file. This function should be **public**, not return anything and only receive the **AActor* TargetActor** parameter, which will be the actor it should be facing. Have a look at the following code snippet, which shows this function:

```
// Change the rotation of the character to face the given actor
```

```
void LookAtActor(AActor* TargetActor);
```

Note

*Even though we only want the enemy to look at the player's character, in order to execute good software development practices, we're going to abstract this function a bit more and allow **EnemyCharacter** to look at any Actor, because the logic that allows an Actor to look at another Actor or at the player character will be exactly the same.*

Remember, you should not create unnecessary restrictions when writing code. If you can write similar code while at the same time allowing more possibilities, you should do so, if that doesn't overcomplicate the logic of your program.

Moving on ahead, if **EnemyCharacter** can't see the **Target Actor**, it shouldn't be looking at it. In order to check whether the enemy can see the Actor, it should be looking at the **LookAtActor** function which will call another function, the **CanSeeActor** function. This is what you'll be doing in the next exercise.

Exercise 5.02: Creating the CanSeeActor Function, Which Executes Line Traces

In this exercise, we will create the **CanSeeActor** function, which will return whether the enemy character can see the given Actor.

The following steps will help you complete this exercise:

1. Create the declaration for the **CanSeeActor** function in the header file of the **EnemyCharacter** class, which will return a **bool** and receive a **const Actor* TargetActor** parameter, which is the Actor we want to look at. This function will be a **const** function, because it doesn't change any of the class's attributes, and the parameter will also be **const** because we won't need to modify any of its properties; we'll only need to access them:

```
// Can we see the given
actor

bool CanSeeActor(const
AActor* TargetActor) const;
```

Now, let's get to the fun part, which is executing the Line Trace.

In order to call functions related to line tracing, we'll have to fetch the enemy's current world with the **GetWorld** function. However, we haven't included the **World** class in this file, so let's do so in the following step:

Note

*The **GetWorld** function is accessible to any Actor and will return the **World** object that the Actor belongs to. Remember, the world is necessary in order to execute the Line Trace.*

2. Open the **EnemyCharacter** source file and find the following code line:

```
#include "EnemyCharacter.h"
```

Add the following line right after the preceding line of code:

```
#include "Engine/World.h"
```

3. Next, create the implementation of the **CanSeeActor** function in the **EnemyCharacter** source file, where you'll start by checking whether our **TargetActor** is a **nullptr**. If it is, we return **false**, given that we have no valid Actor to check our sight to:

```
bool  
AEnemyCharacter::CanSeeActor  
(const AActor *  
TargetActor) const
```

```
{  
    if (TargetActor ==  
        nullptr)  
  
    {  
        return false;  
  
    }  
  
}
```

Next, before we add our Line Trace function call, we need to set up some necessary parameters; we will be implementing these in the following steps.

4. After the previous **if** statement, create a variable to store all the necessary data relative to the results of the Line Trace. Unreal already has a built-in type for this called the **FHitResult** type:

```
// Store the results of the  
Line Trace  
  
FHitResult Hit;
```

This is the variable we will send to our Line Trace function, which will populate it with the relevant info of the

executed Line Trace.

5. Create two **FVector** variables, for the **Start** and **End** locations of our Line Trace, and set them to our enemy's current location and our target's current location, respectively:

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

6. Next, set the Trace Channel we wish to compare against. In our case, we want to have a **Visibility** Trace Channel specifically designated to indicate whether an object blocks another object's view. Luckily for us, such a Trace Channel already exists in UE4, as shown in the following code snippet:

```
// The trace channel we  
want to compare against  
  
ECollisionChannel Channel =  
ECollisionChannel::ECC_Visibility;
```

The **ECollisionChannel** enum represents all the possible Trace Channels available to compare against. We will be using the **ECC_Visibility** value, which represents the **Visibility** Trace Channel.

7. Now that we've set up all our necessary parameters, we can finally call the **LineTrace** function, **LineTraceSingleByChannel**:

```
// Execute the Line Trace  
GetWorld()-  
>LineTraceSingleByChannel(Hit, Start, End, Channel);
```

This function will consider the parameters we send it, execute the Line Trace, and return its results by modifying our **Hit** variable.

Before we continue, there are still a couple more things we need to consider.

If the Line Trace starts from within our enemy character, which is what will happen in our case, that means it's very

likely that the Line Trace will simply hit our enemy character immediately and just stop there, because our character might block the **Visibility** Trace Channel. In order to fix that, we need to tell the Line Trace to ignore it.

8. Use the built-in **FCollisionQueryParams** type, which allows us to give even more options to our Line Trace:

```
FCollisionQueryParams  
queryParams;
```

9. Now, update the **Line Trace** to ignore our enemy, by adding itself to the list of Actors to ignore:

```
// Ignore the actor that's  
executing this Line Trace  
  
queryParams.AddIgnoredActor  
(this);
```

We should also add our target to our list of Actors to ignore because we don't want to know whether it blocks the **EnemySight** channel; we just simply want to know whether

something between the enemy and the player character blocks that channel.

10. Add the Target Actor to the list of Actors to be ignored as shown in the following code snippet:

```
// Ignore the target we're
// checking for
QueryParams.AddIgnoredActor
(TargetActor);
```

11. Next, send our **FCollisionQueryParams** to the Line Trace by adding it as the last parameter of the **LineTraceSingleByChannel** function:

```
// Execute the Line Trace
GetWorld()-
>LineTraceSingleByChannel(H
it, Start, End, Channel,
QueryParams);
```

12. Finalize our **CanSeeActor** function, by returning whether the Line Trace hits anything or not. We can do that by accessing our **Hit** variable and checking whether there was a blocking

hit, using the **bBlockingHit** property. If there was, that means we can't see our **TargetActor**. This can be achieved with the following code snippet:

```
return !Hit.bBlockingHit;
```

Note

*Although we won't need any more information from the **Hit** result, other than whether there was a blocking hit, the **Hit** variable can give us much more information on the Line Trace, such as:*

*Information on the Actor that was hit by the Line Trace (**nullptr** if no Actor was hit), by accessing the **Hit.GetActor()** function*

*Information on the Actor component that was hit by the Line Trace (**nullptr** if no Actor component was hit), by accessing the **Hit.GetComponent()** function*

Information on the location of the hit by accessing the

Hit.Location variable

*The distance of the hit can be found by accessing the **Hit.Distance** variable*

*The angle at which the Line Trace hit the object, which can be found by accessing the **Hit.ImpactNormal** variable*

Finally, our **CanSeeActor** function is complete. We now know how to execute a Line Trace and we can use it for our enemy's logic.

By completing this exercise, we have finished the **CanSeeActor** function; we can now get back to the **LookAtActor** function. However, there is something we should look at first: visualizing our Line Trace.

Visualizing the Line Trace

When creating new logic that makes use of Line Traces, it is very useful to actually visualize the Line Trace while it's being executed, which is something that the Line Trace function doesn't allow you to do. In order to do that, we must use a set of helper debug functions that can draw objects dynamically at runtime, such as lines, cubes, spheres, and so on.

Let's then add a visualization of our Line Trace. The first thing we must do in order to use the debug functions is to add the following **include** below our last **include** line:

```
#include "DrawDebugHelpers.h"
```

We will want to call the **DrawDebugLine** function in order to visualize the Line Trace, which needs the following inputs, very similar to the ones received by the Line Trace function:

1. The current **World**, which we will supply with the **GetWorld** function
2. The **Start** and **End** points of the line, which will be the same as the **LineTraceSingleByChannel** function
3. The desired color of the line in the game, which can be set to **Red**

Then, we can add the **DrawDebugLine** function call below our Line Trace function call as shown in the following code snippet:

```
// Execute the Line Trace  
  
GetWorld()->LineTraceSingleByChannel(Hit,  
Start, End, Channel, QueryParams);  
  
// Show the Line Trace inside the game  
  
DrawDebugLine(GetWorld(), Start, End,  
FColor::Red);
```

This will allow you to visualize the Line Trace as it is being executed, which is very useful.

Note

If you feel the need for it, you can also specify more of the visual Line Trace's properties, such as its lifetime and

thickness.

*There are many **DrawDebug** functions available that will draw cubes, spheres, cones, donuts, and even custom meshes.*

Now that we can both execute and visualize our Line Trace, let's use the **CanSeeActor** function, which we created in the last exercise, inside the **LookAtActor** function.

Exercise 5.03: Creating the LookAtActor Function

In this exercise, we will be creating the definition of our **LookAtActor** function, which will change the enemy's rotation so that it faces the given Actor.

The following steps will help you complete the exercise:

1. Create the **LookAtActor** function definition in the **EnemyCharacter** source file.
2. Start by checking whether our **TargetActor** is a **nullptr** and returns nothing immediately if it is (because it's not valid), as shown in the following code snippet:

```
void  
AEnemyCharacter::LookAtActor  
(AActor * TargetActor)
```

```
{  
    if (TargetActor ==  
        nullptr)  
  
    {  
        return;  
    }  
}
```

3. Next, we want to check whether we can see our Target Actor, using our **CanSeeActor** function:

```
if  
(CanSeeActor(TargetActor))  
{  
}
```

If this **if** statement is true, that means we can see the Actor, and we will set our rotation in such a way that we are facing that Actor. Luckily for us, there's already a function within UE4 that allows us to do that: the **FindLookAtRotation** function. This function will receive as input two points in the level, point A (the **Start** point) and point B (the **End** point), and

return the rotation that the object at the start point must have in order to face the object at the end point.

4. In order to use this function, include **KismetMathLibrary** as shown in the following code snippet:

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

5. The **FindLookAtRotation** function must receive a **Start** and **End** point, which will be our enemy's location and our Target Actor's location, respectively:

```
FVector Start =  
GetActorLocation();  
  
FVector End = TargetActor-  
>GetActorLocation();  
  
// Calculate the necessary  
rotation for the Start  
point to face the End point  
  
FRotator LookAtRotation =  
UKismetMathLibrary::FindLoo  
kAtRotation(Start, End);
```

- Finally, set your enemy character's rotation to the same value as our **LookAtRotation**:

```
//Set the enemy's rotation  
to that rotation  
  
SetActorRotation(LookAtRota  
tion);
```

And that's it for the **LookAtActor** function.

Now the last step is to call the **LookAtActor** function inside the Tick event and to send the player character as the **TargetActor**, the Actor that we want to look at.

- For us to fetch the character that is currently being controlled by the player, we can do so using the **GameplayStatics** object. As with other UE4 objects, we must first include them:

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

- Next, head to your Tick function's body and call the **GetPlayerCharacter** function from **GameplayStatics**:

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

This function receives as input:

1. A World context object, which is, essentially, an object that belongs to our current World, used to let the function know which World object to access. This World context object can simply be the **this** pointer.
 2. A player index, which, given that our game is supposed to be a single-player game, we can safely assume to be **0** (the first player).
9. Next, call the **LookAtActor** function, sending the player character that we just fetched:

```
// Look at the player
```

```
character every frame  
LookAtActor(PlayerCharacter  
);
```

10. The last step of this exercise is to compile your changes in Visual Studio.

Now that you've completed this exercise, your **EnemyCharacter** class has all the necessary logic to face the player character, if it's within view, and we can start creating the **EnemyCharacter** Blueprint class.

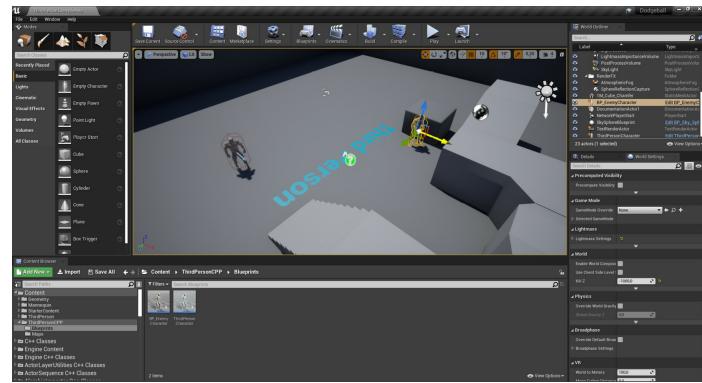
Creating the **EnemyCharacter** Blueprint Class

Now that we have finished the logic for our **EnemyCharacter** C++ class, we must create our blueprint class that derives from it:

1. Open our project in the Editor.
2. Go to the **Blueprints** folder inside the **ThirdPersonCPP** folder, in the **Content Browser**.
3. *Right-click* and select the option to create a new blueprint class.

4. Expand the **All Classes** tab near the bottom of the **Pick Parent Class** window, search for our **EnemyCharacter** C++ class, and select it as the parent class.
5. Name the Blueprint class **BP_EnemyCharacter**.
6. Open the Blueprint class, select the **SkeletalMeshComponent** (called **Mesh**) from the **Components** tab, and set its **Skeletal Mesh** property to **SK_Mannequin** and its **Anim Class** property to **ThirdPerson_AnimBP**.
7. Change the *Yaw* of **SkeletalMeshComponent** to **-90°** (on the *z-axis*) and its position on the *z-axis* to **-83** units.
8. After you've set up the Blueprint class, its mesh setup should look very similar to that of our **DodgeballCharacter** Blueprint class.
9. Drag an instance of the **BP_EnemyCharacter** class to your level, in a location near an object that

can block its line of sight, such as this location (the selected character is **EnemyCharacter**):



**Figure 5.5: Dragging the
BP_EnemyCharacter class into
the level**

10. Now we can finally play the game and verify that our enemy does look at our player character whenever it's within view:



**Figure 5.6: Enemy character with
a clear view of the player using a
Line Trace**

11. We can also see that the enemy stops seeing the player whenever it's not

within view, as shown in *Figure 5.7*:



Figure 5.7: Enemy losing sight of the player

And that concludes our **EnemyCharacter**'s logic. In the next section, we will be looking at Sweep Traces.

Sweep Traces

Before we continue with our project, it is important to know about a variant of the Line Trace, which is the **Sweep Trace**. Although we won't be using these in our project, it is important to know about them and how to use them.

While the Line Trace basically *shoots a ray* between two points, the Sweep Trace will simulate *throwing an object* between two points in a straight line. The object that is being *thrown* is simulated (doesn't actually exist in the game) and can have various shapes. In the Sweep Trace, the **Hit** location will be the first point at which the virtual object (which we will call **Shape**) hits another object, if it were thrown from the start point to the end point. The shapes of the Sweep Trace can be either a box, a sphere, or a capsule.

Here is a representation of a Sweep Trace from point **A** to point **B**, where we assume that object **1** is ignored due to its Trace Channel properties, using a box shape:

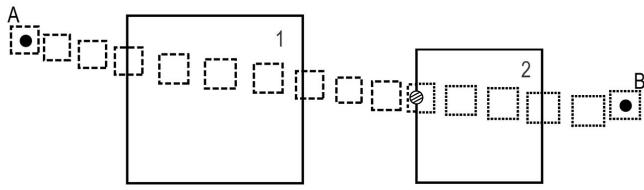


Figure 5.8: Representation of a Sweep Trace

From *Figure 5.8*, we notice the following:

- A Sweep Trace, using a box shape, being executed from point A to point B.
- The dashed boxes represent the Sweep Trace before it hits an object.
- The dotted boxes represent the Sweep Trace after it hits an object.
- The striped circle represents the Sweep Trace's impact point with object **2**, which is the point at which the Sweep Trace Box shape's surface and object **2**'s surface collide with each other.
- The big squares represent two objects that are in the path of the Line Sweep Trace (objects **1** and **2**).
- Object **1** is ignored in the Sweep Trace due to assumptions based on its Trace Channel properties.

Sweep Traces are more useful than regular Line Traces in a

few situations. Let's take the example of our enemy character, which can throw dodgeballs. If we wanted to add a way for the player to constantly visualize where the next dodgeball that the enemy throws will land, that could be better achieved with a Sweep Trace: we would do a Sweep Trace with the shape of our dodgeball (a sphere) toward our player, check the impact point, and show a sphere on that impact point, which would be visible to the player. If the Sweep Trace hits a wall or a corner somewhere, the player would know that, if the enemy were to throw a dodgeball at that moment, that's where it would hit first. You could use a simple Line Trace for the same purpose, but the setup would have to be rather complex in order to achieve the same quality of results, which is why Sweep Traces are a better solution in this case.

Let's now take a quick look at how we can do a Sweep Trace in code.

Exercise 5.04: Executing a Sweep Trace

In this exercise, we will implement a Sweep Trace in code. Although we won't be using it for our project, by performing this exercise, you will become familiar with such an operation.

Go to the end of the **CanSeeActor** function created in the previous sections and follow these steps:

1. The function responsible for the Sweep Trace is **SweepSingleByChannel**, which is available within UE4 and requires the following parameters as inputs:

An **FHitResult** type, to store the results of the sweep (we already have one of these, so there's no need to create another variable of this type):

```
// Store the results of the Line Trace  
  
FHitResult Hit;
```

Start and **End** points of the sweep (we already have both of these, so there's no need to create another variable of this type):

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

2. Use the intended rotation of the shape, which is in the form of an **FQuat** type (representing a quaternion). In this case, it's set to a rotation of **0** on all axes, by accessing the **FQuat**'s **Identity** property:

```
// Rotation of the shape used in the Sweep Trace
```

```
FQuat Rotation =  
FQuat::Identity;
```

3. Now, use the intended Trace Channel to compare it against (we already have one of these, so there's no need to create another variable of this type):

```
// The trace channel we  
want to compare against  
  
ECollisionChannel Channel =  
ECollisionChannel::ECC_Visibility;
```

4. Finally, use the shape of a box for the Sweep Trace by calling the **FCollisionShape MakeBox** function and supplying it with the radius (on all three axes) of the box shape we want. This is shown in the following code snippet:

```
// Shape of the object used  
in the Sweep Trace  
  
FCollisionShape Shape =  
FCollisionShape::MakeBox(FVector(20.f, 20.f, 20.f));
```

5. Next, call the **SweepSingleByChannel** function:

```
GetWorld()-  
>SweepSingleByChannel(Hit,  
  
Start,  
  
End,  
  
Rotation,  
  
Channel,  
  
Shape);
```

With these steps completed, we finish our exercise on Sweep Traces. Given that we won't be using Sweep Traces in our project, you should comment out the **SweepSingleByChannel** function, so that our **Hit** variable doesn't get modified and lose the results from our Line Trace.

Now that we've concluded the segment on Sweep Traces, let's get back to our **Dodgeball** project and learn how to change an object's response to a Trace Channel.

Changing the Visibility Trace Response

In our current setup, every object that is visible blocks the **Visibility** Trace Channel; however, what if we wanted to

change whether an object blocks that channel completely? In order to do this, we must change a component's response to that channel. Have a look at the following example:

1. We select the cube that we've been using to block the enemy's sight in our level as shown in *Figure 5.9*:



Figure 5.9: Default spawn of the character

2. Then, you go to the **Collision** section of this object's **Details Panel** (its default place in the **Editor's interface**):

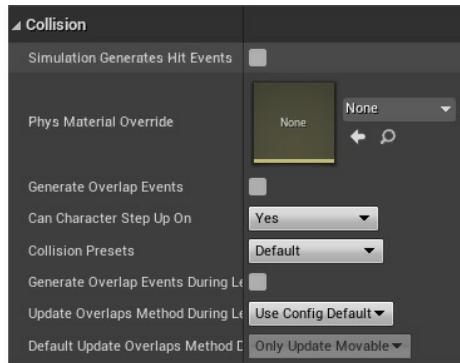


Figure 5.10: Collision tab in the Details Panel in Unreal

3. Here, you'll find several collision-related options. The one we want to pay attention to right now is the **CollisionPresets** option. Its current value is **Default**; however, we want to change it according to our own preferences, so we will click on the drop-down box and change its value to **Custom**.

4. Once you do this, you'll notice a whole group of new options pops up:

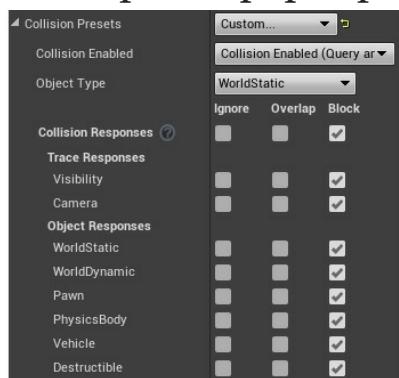


Figure 5.11: Collision Preset set to Custom

This group of options allows you to specify how this object responds to Line Traces and object collision, and the type of collision object it is.

The option you should be paying attention to is **Visibility**. You'll

notice it's set to **Block**, but that you can also set it to **Overlap** and **Ignore**.

Right now, the cube is blocking the **Visibility** trace channel, which is why our enemy can't see the character when it's behind this cube. However, if we change the object's response to the **Visibility** Trace Channel to either **Overlap** or **Ignore**, the object will no longer block Line Traces that check for visibility (which is the case for the Line Trace you've just written in C++).

5. Let's change the cube's response to the **Visibility** channel to **Ignore**, and then play the game. You'll notice that the enemy is still looking toward the player character, even when it's behind the cube:

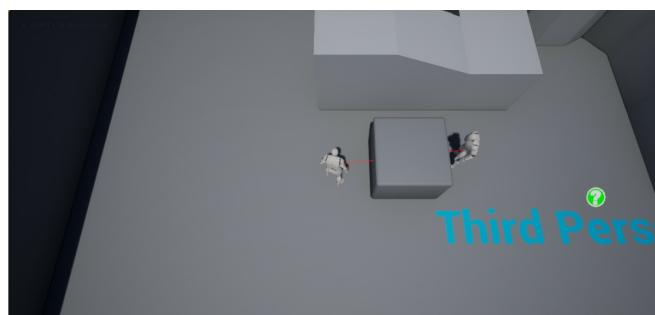


Figure 5.12: Enemy character looking through an object at the

player

This is because the cube no longer blocks the **Visibility** Trace Channel, and so the Line Trace the enemy is executing no longer hits anything when trying to reach the player character.

Now that we've seen how we can change an object's response to a specific Trace Channel, let's change the cube's response to the **Visibility** channel back to **Block**.

However, there's one thing that's worth mentioning: if we were to set the cube's response to the **Visibility** channel to **Overlap**, instead of **Ignore**, the result would be the same. But why is that, and what is the purpose of having these two responses? In order to explain that, we'll look at Multi Line Traces.

Multi Line Traces

While using the **CanSeeActor** function in *Exercise 5.02, Creating the*

CanSeeActor Function, Which Executes Line Traces, you might have wondered to yourself about the name of the Line Trace function we used, **LineTraceSingleByChannel**, specifically about why it used the word *Single*. The reason for that is because you can also execute **LineTraceMultiByChannel**.

But how do these two Line Traces differ?

While the Single Line Trace will stop checking for objects that block it after it hits an object, and tell us that was the object that it hit, the Multi Line Trace can check for any objects that are hit by the same Line Trace.

The Single Line Trace will:

1. Ignore the objects that have their response set to either **Ignore** or **Overlap** on the Trace Channel being used by the Line Trace
2. Stop when it finds an object that has its response set to **Block**

However, instead of ignoring objects that have their response set to **Overlap**, the Multi Line Trace will add them as objects that were found during the Line Trace, and only stop when it finds an object that blocks the desired Trace Channel (or *when it reaches the end point*). In the next figure, you'll find an illustration of a Multi Line Trace being executed:

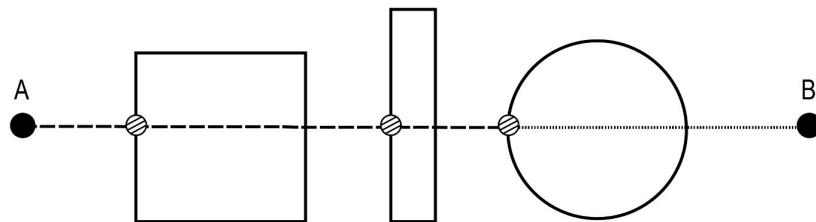


Figure 5.13: A Multi Line Trace being executed from point A to point B

In *Figure 5.13*, we notice the following:

- The dashed line represents the Line Trace before it hits an object that blocks it.
- The dotted line represents the Line Trace after it hits an object that blocks it.
- The striped circles represent the Line Trace's impact points, only the last one of which is a blocking hit in this case.

The only difference between the **LineTraceSingleByChannel** and the **LineTraceMultiByChannel** functions, when it comes to their inputs, is that the latter must receive a **TArray<FHitResult>** input, instead of a single

FHitResult. All other inputs are the same.

Multi Line Traces are very useful when simulating the behavior of bullets with strong penetration that can go through several objects before stopping completely. Keep in mind that you can also do Multi Sweep Traces by calling the **SweepMultiByChannel** function.

Note

*Another thing about the **LineTraceSingleByChannel** function that you might be wondering about is the **ByChannel** portion. This distinction has to do with using a Trace Channel, as opposed to the alternative, which is an Object Type. You can do a Line Trace that uses Object Types instead of Trace Channels by calling the **LineTraceSingleByObjectType** function, also available from the World object. Object Types are related to topics we will be covering in the next chapter, so we won't be going into detail on this function just yet.*

The Camera Trace Channel

When changing our cube's response to the **Visibility** Trace Channel, you may have noticed the other out-of-the-box Trace Channel: **Camera**.

This channel is used to specify whether an object blocks the line of sight between the camera's spring arm and the character it's associated with. In order to see this in action, we can drag an object to our level and place it in such a way that it will stay between the camera and our player character.

Have a look at the following example; we begin by duplicating the **floor** object.

Note

You can easily duplicate an object in the level by holding the Alt key and dragging one of the Move Tool's arrows in any direction.

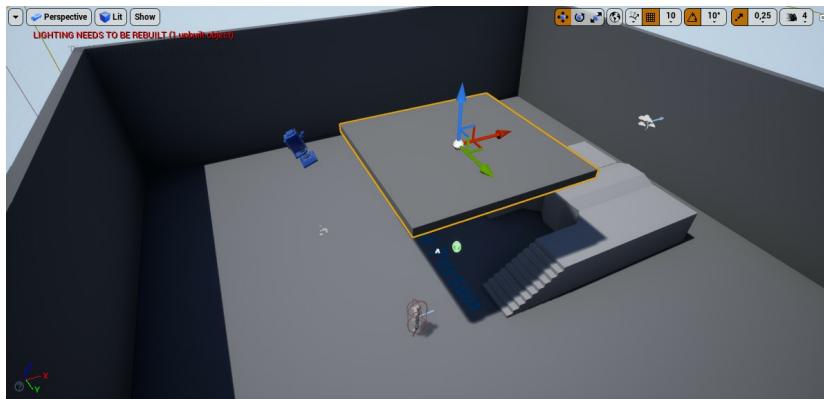


Figure 5.14: Floor object being selected

6. Next, we change its **Transform** values as shown in the following figure:



Figure 5.15: Updating the Transform values

7. Now when you play your game, you'll notice that when the character goes under our duplicated floor object, you

won't lose sight of the player's character, but rather the spring arm will cause the camera to move down until you can see the character:

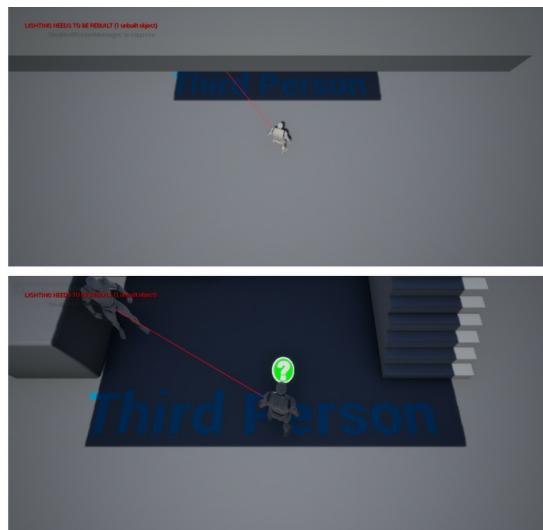


Figure 5.16: Changes in the camera angle

8. In order to see how the spring arm's behavior differs when an object isn't blocking the **Camera** Trace Channel, change our duplicated floor's response to the **Camera** channel to **Ignore** and play the level again. What will happen is that, when our character goes under the duplicated floor, we will lose sight of the character.

After you've done these steps, you can see that the **Camera** channel is used to specify whether an object will cause the spring arm to move the camera closer to the player when it

intersects that object.

Now that we know how to use the existing Trace Channels, what if we wanted to create our own Trace Channels?

Exercise 5.05: Creating a Custom EnemySight Trace Channel

As we've discussed before, UE4 comes with two out-of-the-box Trace Channels: **Visibility** and **Camera**. The first one is a general-use channel that we can use to specify which objects block the line of sight of an object, while the second one allows us to specify whether an object blocks the line of sight between the camera's spring arm and the character it's associated with.

But how can we create our own Trace Channels? That's what we'll be looking into in this exercise. We will create a new **EnemySight** Trace Channel and use it to check whether the enemy can see the player character, instead of the built-in **Visibility** channel:

1. Open **Project Settings** by pressing the **Edit** button at the top-left corner of the editor, and go to the **Collision** section. There you'll find the **Trace Channels** section. It's currently empty because we haven't yet created any of our own Trace Channels.

2. Select the **New Trace Channel** option. A window should pop up, giving you the option to name your new channel and set its default response by the objects in your project. Name our new Trace Channel **EnemySight** and set its default response to **Block**, because we want most objects to do exactly that.

3. After you've created the new Trace Channel, we must go back to our **EnemyCharacter** C++ class and change the trace we're comparing against in our Line Trace:

```
// The trace channel we
want to compare against

ECollisionChannel Channel =
ECollisionChannel::ECC_Visibility;
```

Given that we are no longer using the **Visibility** channel, we must reference our new channel, but how do we do that?

In your project's directory, you'll find the **Config** folder. This folder contains several **ini** files related to

your project, such as
DefaultGame.ini,
DefaultEditor.ini,
DefaultEngine.ini, and so on.
Each of these contains several properties that will be initialized when the project is loaded. The properties are set by name-value pairs (**property=value**), and you can change their values as desired.

4. When we created our **EnemySight** channel, the project's **DefaultEngine.ini** file was updated with our new Trace Channel. Somewhere in that file, you'll find this line:

```
+DefaultChannelResponses=
(Channel=ECC_GameTraceChann
el1,
DefaultResponse=ECR_Block, b
TraceType=True, bStaticObjec
t=False, Name="EnemySight")
```

Note

The preceding code line can be found highlighted here:

<https://packt.live/3eFpz5r>.

This line says that there is a custom Trace Channel called **EnemySight** that has a default response of Block and, most importantly, is available in C++ using the **ECC_GameTraceChannel1** value of the collision **enum** we mentioned before, **ECollisionChannel**. This is the channel we'll be referencing in our code:

```
// The trace channel we
want to compare against

ECollisionChannel Channel =
ECollisionChannel::ECC_Game
TraceChannel1;
```

5. Verify that our enemy's behavior remains the same after all the changes we've made. This means that the enemy must still face the player's character, as long as it's within view of said enemy.

By completing this exercise, we now know how to make our own Trace Channels for any desired purpose.

Going back to our enemy character, there are still ways that

we can improve its logic. Right now, when we fetch our enemy's location as the start point of the Line Trace, that point is somewhere around the enemy's hip, because that's where the origin of the Actor is. However, that's not usually where people's eyes are, and it wouldn't make much sense to have a humanoid character looking from its hip instead of its head.

So, let's change that, and have our enemy character check whether it sees the player character starting from its eyes, instead of its hip.

Activity 5.01: Creating the SightSource Property

In this activity, we will be improving our enemy's logic to determine whether it should look at the player. Currently, the Line Trace that's being done to determine that is being *shot* from around our character's hips, **(0, 0, 0)** in our **BP_EnemyCharacter** blueprint, and we want this to make a bit more sense, so we'll make it so that the Line Trace starts somewhere close to our enemy's eyes. Let's get started then.

The following steps will help you complete the activity:

1. Declare a new **SceneComponent** in our **EnemyCharacter** C++ class called **SightSource**. Make sure to declare this as a **UPROPERTY** with the **VisibleAnywhere**, **BlueprintReadOnly**, **Category = LookAt** and **meta =**

`(AllowPrivateAccess = "true")` tags.

2. Create this component in the **EnemyCharacter** constructor by using the **CreateDefaultSubobject** function, and attach it to the **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.
4. 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 was done in the *Creating the EnemyCharacter Blueprint Class* section to the **SkeletalMeshComponent** property of **BP_EnemyCharacter**.

Hint: This can be achieved from the **Transform** tab in the **Editor Panel** as shown in *Figure 5.17*.



Figure 5.17: Updating the SightSource component's values

Expected output:

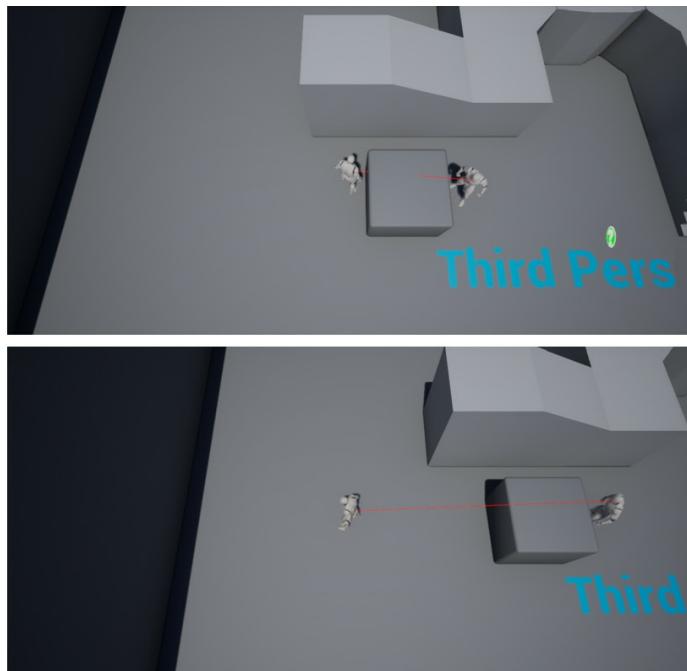


Figure 5.18: Expected output showing the updated Line Trace from the hip to the eye

Note

The solution to this activity can be found at:
<https://packt.live/338jEBx>.

By completing this activity, we have updated our **SightSource** property for our **EnemyCharacter**.

Summary

By completing this chapter, you have added a new tool to your belt: Line Traces. You now know how to execute Line Traces and Sweep Traces, both Single and Multi; how to change an object's response to a specific Trace Channel; and how to create your own Trace Channels.

You will quickly realize in the chapters ahead that these are essential skills when it comes to game development, and you will make good use of them on your future projects.

Now that we know how to use Line Traces, we're ready for the next step, which is Object Collision. In the next chapter, you will learn how to set up collisions between objects and how to use collision events to create your own game logic. You will create the Dodgeball Actor, which will be affected by real-time physics simulation; the Wall Actors, which will block both the characters' movements and the dodgeball; and the Actor responsible for ending the game when the player comes into contact with it.

6. Collision Objects

Overview

In this chapter, we will continue working on the collision-based game we introduced in the previous chapter by adding further mechanics and objects to our game. Initially, we will follow on from the previous chapter by introducing object collision. You will learn how to use collision boxes, collision triggers, overlap events, hit events, and physics simulation. You will also learn how to use timers, the Projectile Movement Component, and Physical Materials.

Introduction

In the previous chapter, we came across some of the basic concepts of collision, namely Line Traces and Sweep Traces. We learned how to execute different types of Line Traces, how to create our own custom Trace Channels, and how to change how an object responds to a specific channel. Many of the things you learned in the previous chapter will be used in this chapter, where we'll learn about object collision.

Throughout this chapter, we will continue to build upon our top-down **Dodgeball** game by adding game mechanics that revolve around object collision. We will create the **Dodgeball actor**, which will act as a dodgeball that bounces off of the floor and walls; a **Wall actor**, which will block all objects; a **Ghost Wall actor**, which will only block the player, not the enemies' lines of sight or the dodgeball; and a **Victory Box actor**, which will end the game when the player enters the

Victory Box, representing the end of the level.

Before we start creating our **Dodgeball** class, we will go over the basic concepts of object collision in the next section.

Object Collision in UE4

Every game development tool must have a physics engine that simulates collision between multiple objects, as explained in the previous chapter. Collision is the backbone of most games released nowadays, whether 2D or 3D. In many games, it's the main way in which the player acts upon the environment, be it running, jumping, or shooting, and the environment acts accordingly by making the player land, get hit, and so on. It is no understatement to say that, without simulated collision, it wouldn't be possible to make many games at all.

So, let's understand how object collision works in UE4 and the ways in which we can use it, starting with collision components.

Collision Components

In UE4, there are two types of components that can affect and be affected by collision; they are as follows:

- Meshes
- Shape objects

Meshes can be as simple as a cube, or as complex as a high-resolution character with tens of thousands of vertices. A mesh's collision can be specified with a custom file imported

alongside the mesh into UE4 (which is outside the scope of this book), or it can be calculated automatically by UE4 and customized by you.

It is generally a good practice to keep the collision mesh as simple (few triangles) as possible so that the physics engine can efficiently calculate collision at runtime. The types of meshes that can have collision are as follows:

- Static Meshes
- Skeletal Meshes
- Procedural Meshes
- And so on

Shape objects, which are simple meshes represented in wireframe mode that are used to behave as collision objects by causing and receiving collision events.

Note

Wireframe mode is a commonly used visualization mode in game development, usually for debugging purposes, which allows you to see meshes without any faces or textures – they can only be seen through their edges, which are connected by their vertices. You will see what wireframe mode is when we add a Shape component to an actor.

Please note that Shape objects are essentially invisible meshes and that their three types are as follows:

- Box Collision (Box Component in C++)
- Sphere Collision (Sphere Component)

in C++)

- Capsule Collider (Capsule Component
in C++)

Note

*There's a class that all the components that provide geometry and collision inherit from, which is the **Primitive Component**. This component is the basis for all components that contain any sort of geometry, which is the case for mesh components and shape components.*

So, how can these components collide, and what happens when they do so? We shall have a look at this in the next section, collision events.

Collision Events

Let's say that there are two objects colliding into one another. Two things can happen:

- They overlap each other, as if the other object weren't there, in which case the **Overlap** event is called.
- They collide and prevent each other

from continuing their course, in which case the **Block** event is called.

In the previous chapter, we learned how to change an object's response to a specific **Trace** channel. During this process, we learned that an object's response can be either **Block**, **Overlap**, or **Ignore**.

Now, let's see what happens in each of these responses during a collision.

Block: Two objects will only block each other if both of them have their response to the other object set to **Block**:

- Both objects will have their **OnHit** events called. This event is called whenever two objects block each other's path at the moment they collide. If one of the objects is simulating physics, that object must have its **SimulationGeneratesHitEvents** property set to **true**.
- Both objects will physically stop each other from continuing with their course.

Have a look at the following figure, which shows an example of when two objects are thrown and bounce off each other:

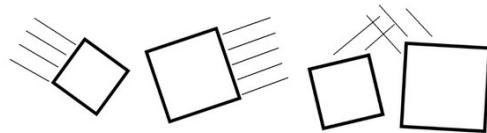


Figure 6.1: Object A and Object B blocking each other

Overlap: Two objects will overlap each other if they don't block each other and neither of them is ignoring the other:

- If both objects have the **GenerateOverlapEvents** property set to **true**, they will have their **OnBeginOverlap** and **OnEndOverlap** events called. These overlap events are called when an object starts and stops overlapping another object, respectively. If at least one of them doesn't have this property set to **true**, neither of them will call these events.
- The objects act as if the other object doesn't exist and will overlap each other.

As an example, suppose the player's character walks into a trigger box that marks the end of the level, which only reacts to the player's character.

Have a look at the following figure, which shows an example of two objects overlapping each other:

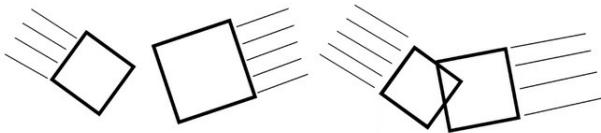


Figure 6.2: Object A and Object B overlapping each other

Ignore: Two objects will ignore each other if at least one of them is ignoring the other:

- There will be no events called on either object.
- Similar to the **Overlap** response, the objects will act as if the other object doesn't exist and will overlap each other.

An example of two objects ignoring each other would be when an object other than the player's character goes into a trigger box that marks the end of the level, which only reacts to the player's character.

Note

*You can look at the previous figure, where two objects overlap each other, to understand **Ignore**.*

Here is a table to help you understand the necessary responses that two objects must have in order to trigger the previously described situations:

		Object B	Block	Overlap	Ignore
		Object A	Block	Overlap	Ignore
Block	Block	Block	Overlap	Ignore	
	Overlap	Overlap	Overlap	Ignore	
Ignore	Ignore	Ignore	Ignore	Ignore	

Figure 6.3: Resulting responses on objects based on Block, Overlap, and Ignore

Following this table, consider that you have two objects – Object A and Object B:

- If Object A has set its response to Object B to **Block** and Object B has set its response to Object A to **Block**, they will **Block** each other.
- If Object A has set its response to Object B to **Block** and Object B has set its response to Object A to **Overlap**, they will **Overlap** each other.
- If Object A has set its response to Object B to **Ignore** and Object B has set its response to Object A to **Overlap**, they will **Ignore** each other.

Note

You can find a full reference to UE4's collision interactions here:

<https://docs.unrealengine.com/en-US/Engine/Physics/Collision/Overview>

.

A collision between objects has two aspects to it:

Physics: All collisions related to physics simulation, such as a ball being affected by gravity and bouncing off the floors and walls.

The physically simulated response of the collision within the game, which can be either:

- Both objects continuing their trajectories as if the other object wasn't there (no physical collision).
- Both objects colliding and changing their trajectories, usually with at least one of them continuing its movement, that is, blocking each other's paths.

Query: Query can be divided into two aspects of collision, as follows:

- The events related to the collision of the objects that are called by the game and that you can use to create additional logic. These events are the same ones we mentioned previously:
- The **OnHit** event

- The **OnBeginOverlap** event
- The **OnEndOverlap** event
- The physical response of the collision within the game, which can be either:
 - Both objects continuing their movement as if the other object wasn't there (no physical collision)
 - Both objects colliding and blocking each other's path

The physical response from the Physics aspect might sound similar to the physical response from the Query aspect; however, although those are both physical responses, they will cause objects to behave differently.

The physical response from the Physics aspect (physics simulation) only applies when an object is simulating physics (for example, being affected by gravity, bouncing off the walls and ground, and so on). Such an object, when hitting a wall, for instance, will bounce back and continue moving in another direction.

On the other hand, the physical response from the Query aspect applies to all objects that don't simulate physics. An object can move without simulating physics when being controlled by code (for example, by using the **SetActorLocation** function or by using the Character Movement Component). In this case, depending on which method you use to move the object and its properties, when an object hits a wall, it will simply stop moving instead of bouncing back. This is because you're simply telling the object to move in a certain direction and something is blocking its path, so the physics engine doesn't allow that object to

continue moving.

In the next section, we will be looking at Collision Channels.

Collision Channels

In the previous chapter, we took a look at the existing Trace Channels (*Visibility* and *Camera*) and learned how to make our own custom channel. Now that you know about Trace Channels, it's time to talk about Object Channels, also known as Object Types.

While Trace Channels are only used for Line Traces, Object Channels are used for object collision. You can specify a "purpose" for each **Object** Channel, much like with Trace Channels, such as Pawn, Static Object, Physics Object, Projectile, and so on. You can then specify how you want each Object Type to respond to all the other Object Types by blocking, overlapping, or ignoring objects of that type.

Collision Properties

Now that we've taken a look at how collision works, let's go back to the collision settings of the cube we selected in the previous chapter, where we changed its response to the Visibility Channel.

The cube can be seen in the following screenshot:

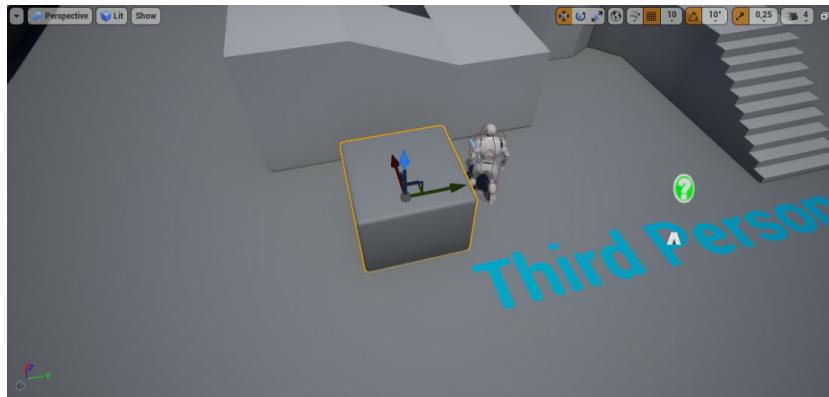


Figure 6.4: Cube blocking the SightSource of the enemy

With the level open in the editor, select the cube and go to the **Collision** section of its Details Panel:

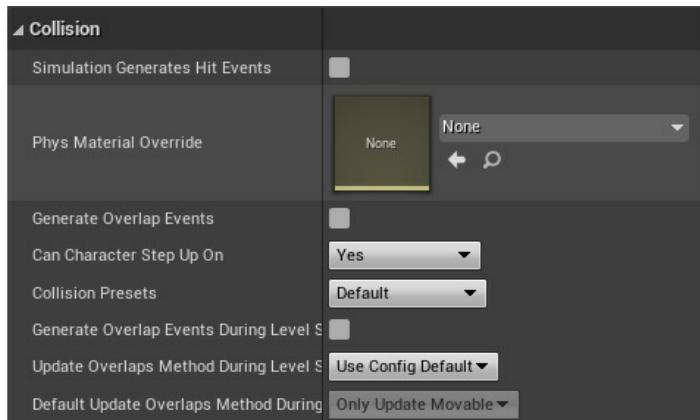


Figure 6.5: The changes in the level editor

Here, we can see some options that are important to us:

- **SimulationGeneratesHitEvents**, which allows the **OnHit** events to be called when an object is simulating physics (we'll talk about this later in this chapter).

- **GenerateOverlapEvents**, which allows the **OnBeginOverlap** and **OnEndOverlap** events to be called.
- **CanCharacterStepUpon**, which allows a character to easily step up onto this object.
- **CollisionPresets**, which allows us to specify how this object responds to each Collision Channel.

Let's change the **CollisionPresets** value from **Default** to **Custom** and take a look at the new options that show up:

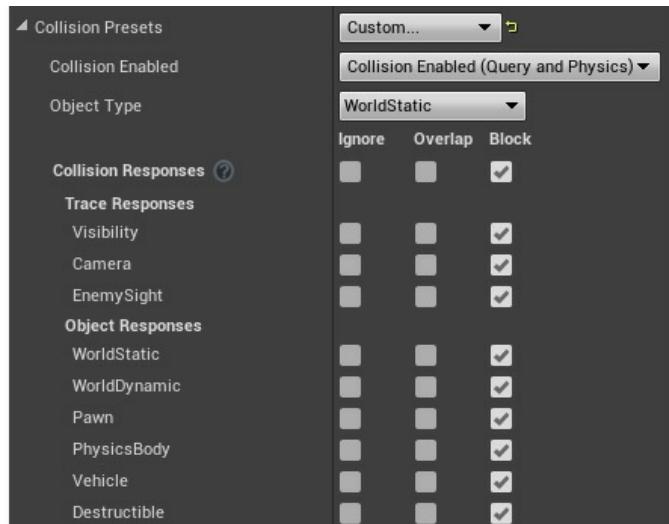


Figure 6.6: Changes in Collision Presets

The first of these options is the **CollisionEnabled** property. It allows you to specify which aspects of collision you want this object to be considered for: Query, Physics, Both, or None. Again, Physics Collision is related to physics simulation (whether this object will be considered by other

objects that simulate physics), while Query Collision is related to collision events and whether objects will block each other's movement:

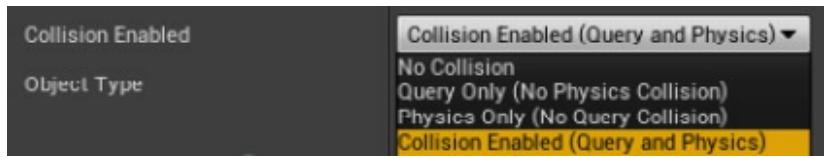


Figure 6.7: Collision Enabled for Query and Physics

The second option is the **ObjectType** property. This is very similar to the Trace Channel concept but is specifically for object collision and, most importantly, dictates what type of collision object this is. The Object Type values that come with UE4 are as follows:

- **WorldStatic**: An object that doesn't move (structures, buildings, and so on)
- **WorldDynamic**: An object that may move (objects whose movement is triggered by code, objects the player can pick up and move, and so on)
- **Pawn**: Used for Pawns that can be controlled and moved around the level
- **PhysicsBody**: Used for objects that simulate physics
- **Vehicle**: Used for Vehicle objects
- **Destructible**: Used for destructible meshes

As mentioned previously, you can create your own custom object types (which will be mentioned later in this chapter) as well, similar to how you can create your own Trace Channels (*which was covered in the previous chapter*).

The last option we have is related to **Collision**

Responses. Given that this **Cube** object has the default collision options, all the responses are set to **Block**, which means that this object will block all the Line Traces and all objects that block **WorldStatic** objects, given that that is this object's type.

Because there are so many different combinations of collision properties, UE4 allows you to group collision property values in the form of Collision Presets.

Let's go back to the **CollisionPresets** property, which is currently set to **Custom**, and click it so that we can see all the possible options. Some of the existing **Collision Presets** are as follows:

No Collision: Used for objects that aren't affected by collision whatsoever:

- **Collision Enabled: NoCollision**
- **Object Type: WorldStatic**
- Responses: Irrelevant
- Example: Objects that are purely visual and distant, such as an object that the player will never reach

Block All: Used for objects that are static and block all other objects:

- **Collision Enabled: Query and Physics**
- **Object Type: WorldStatic**
- Responses: **Block** all channels
- Example: Objects that are close to the player character and block their movement, such as the floor and walls, which will always be stationary

Overlap All: Used for objects that are static and overlap all other objects:

- **Collision Enabled: Query only**
- **Object Type: WorldStatic**
- Responses: **Overlap** all channels
- Example: Trigger boxes placed in the level, which will always be stationary

Block All Dynamic: Similar to the **Block All** preset, but for dynamic objects that may change their transform during gameplay (**Object Type: WorldDynamic**)

Overlap All Dynamic: Similar to the **Overlap All** preset, but for dynamic objects that may change their transform during gameplay (**Object Type: WorldDynamic**)

Pawn: Used for pawns and characters:

- **Collision Enabled: Query and Physics**
- **Object Type: Pawn**
- Responses: **Block** all channels,
Ignore Visibility Channel
- Example: Player character and non-playable characters

Physics Actor: Used for objects that simulate physics:

- **Collision Enabled: Query and Physics**
- **Object Type: PhysicsBody**
- Responses: **Block** all channels
- Example: Objects that are affected by physics, such as a ball that bounces off the floor and walls

Just like the other collision properties, you can also create your own Collision Presets.

Note

You can find a full reference to UE4's collision responses here: <https://docs.unrealengine.com/en-US/Engine/Physics/Collision/Reference>.

Now that we know about the basic concepts of collision, let's go ahead and start creating the **Dodgeball** class. The next exercise will guide you toward doing just that.

Exercise 6.01: Creating the Dodgeball Class

In this exercise, we'll be creating our **Dodgeball** class, which will be thrown by our enemies and bounce off the floor and walls, just like an actual dodgeball.

Before we actually start creating the **Dodgeball** C++ class and its logic, we should set up all the necessary collision settings for it.

The following steps will help you complete this exercise:

1. Open our **Project Settings** and go to the **Collision** subsection within the **Engine** section. Currently, there are no Object Channels, so you need to create a new one.
2. Press the **New Object Channel** button, name it **Dodgeball**, and set its **Default Response** to **Block**.
3. After you've done this, expand the **Preset** section. Here, you'll find all the default presets available in UE4. If you select one of them and press the

Edit option, you can change that **Preset** collision's settings.

4. Create your own **Preset** by pressing the **New** option. We want our **Dodgeball Preset** settings to be as follows:
 1. **Name: Dodgeball**
 2. **CollisionEnabled:**
Collision Enabled
(Query and Physics)
(we want this to be considered for physics simulation as well as collision events)
 3. **Object Type: Dodgeball**
 4. **Collision Responses:**
Select *Block* for most of the options, but *Ignore* the Camera and **EnemySight**
(we don't want the dodgeball to block the camera or the enemy's line of sight)
5. Once you've selected the correct options, press **Accept**.

Now that the **Dodgeball** class's collision settings have been set up, let's create the **Dodgeball** C++ class.

6. Inside the **Content Browser**, right-click and select **New C++ Class**.
7. Choose **Actor** as the parent class.
8. Choose **DodgeballProjectile** as the name of the class (our project is already named **Dodgeball**, so we can't name this new class that too).
9. Open the **DodgeballProjectile** class files in Visual Studio. The first thing we'll want to do is add the collision component of the Dodgeball, so we'll add a **SphereComponent** to our class header (*actor component properties are usually private*):

```
UPROPERTY(VisibleAnywhere,  
BlueprintReadOnly, Category  
= Dodgeball, meta =  
(AllowPrivateAccess =  
"true"))  
  
class USphereComponent*  
SphereComponent;
```

10. Next, include the **SphereComponent** class at the top of our source file:

```
#include  
"Components/SphereComponent  
.h"
```

Note

Keep in mind that all header file includes must be before the .generated.h include.

Now, head to the **DodgeballProjectile** class's constructor, within its source file, and perform the following steps.

11. Create the **SphereComponent** object:

```
SphereComponent =  
CreateDefaultSubobject<USph  
ereComponent>(TEXT("Sphere  
Collision"));
```

12. Set its **radius** to **35** units:

```
SphereComponent -  
>SetSphereRadius(35.f);
```

13. Set its **Collision Preset** to the

Dodgeball preset we created:

```
SphereComponent -  
>SetCollisionProfileName(FN  
ame("Dodgeball"));
```

14. We want the **Dodgeball** to simulate physics, so notify the component of this, as shown in the following code snippet:

```
SphereComponent -  
>SetSimulatePhysics(true);
```

15. We want the **Dodgeball** to call the **OnHit** event while simulating physics, so call the **SetNotifyRigidBodyCollision** function in order to set that to **true** (this is the same as the **SimulationGeneratesHitEvents** property that we saw in the **Collision** section of an object's properties):

```
//Simulation generates Hit  
events
```

```
SphereComponent -  
>SetNotifyRigidBodyCollisio  
n(true);
```

We will also want to listen to the **OnHit** event of **SphereComponent**.

16. Create a declaration for the function that will be called when the **OnHit** event is triggered, in the **DodgeballProjectile** class's header file. This function should be called **OnHit**. It should be **public**, return nothing (**void**), have the **FUNCTION** macro, and receive some parameters, in this order:

1. **UPrimitiveComponent***
HitComp: The component that was hit and belongs to this actor. A Primitive Component is an actor component that has a **Transform** property and some sort of geometry (for example, a **Mesh** or **Shape** Component).
2. **AActor* OtherActor**: The other actor involved in the collision.
3. **UPrimitiveComponent***
OtherComp: The component

that was hit and belongs to the other actor.

4. **FVector NormalImpulse:**

The direction in which the object will be moving after it has been hit, and with how much force (by checking the size of the vector). This parameter will only be non-zero for objects that are simulating physics.

5. **FHitResult& Hit:** The data of the **Hit** resulting from the collision between this object and the other object. As we saw in the previous chapter, it contains properties such as the location of the **Hit**, its normal, which component and actor it hit, and so on. Most of the relevant information is already available to us through the other parameters, but if you need more detailed information, you can access

this parameter:

```
UFUNCTION( )  
void  
OnHit(UPrimitiveComp  
onent* HitComp,  
AActor* OtherActor,  
UPrimitiveComponent*  
OtherComp, FVector  
NormalImpulse, const  
FHitResult& Hit);
```

Add the **OnHit** function's implementation to the class's source file and within that function, at least for now, destroy the dodgeball when it hits the player.

17. Cast the **OtherActor** parameter to our **DodgeballCharacter** class and check if the value is not a **nullptr**. If it's not, which means that the other actor we hit is a **DodgeballCharacter**, we'll destroy this **DodgeballProjectile** actor:

```
void  
ADodgeballProjectile::OnHit
```

```

(UPrimitiveComponent *
HitComp, AActor *
OtherActor,
UPrimitiveComponent *
OtherComp, FVector
NormalImpulse, const
FHitResult & Hit)

{
    if
(Cast<ADodgeballCharacter>
(OtherActor) != nullptr)

{
    Destroy();

}
}

```

Given that we're referencing the **DodgeballCharacter** class, we'll need to include it at the top of this class's source file:

```
#include
"DodgeballCharacter.h"
```

Note

In the next chapter, we'll change this

function so that we have the dodgeball damage the player before destroying itself. We'll do this when we talk about Actor Components.

18. Head back to the **DodgeballProjectile** class's constructor and add the following line at the end in order to listen to the **OnHit** event of **SphereComponent**:

```
// Listen to the  
OnComponentHit event by  
binding it to our function  
  
SphereComponent-  
>OnComponentHit.AddDynamic(  
this,  
&ADodgeballProjectile::OnHi  
t);
```

This will bind the **OnHit** function we created to this **SphereComponent** **OnHit** event (because this is an actor component, this event is called **OnComponentHit**), which means our function will be called alongside that event.

19. Lastly, make **SphereComponent** this

actor's **RootComponent**, as shown in the following code snippet:

```
// Set this Sphere  
Component as the root  
component,  
  
// otherwise collision  
won't behave properly  
  
RootComponent =  
SphereComponent;
```

Note

*In order for a moving actor to behave correctly on collision, whether it's simulating physics or not, it is usually necessary for the main collision component of the actor to be its **RootComponent**.*

*For example, the **RootComponent** of the **Character** class is a **Capsule Collider** component, because that actor will be moving around and that component is the main way the character collides with the environment.*

Now that we've added the

DodgeballProjectile C++ class's logic, let's go ahead and create our Blueprint class.

20. Compile your changes and open the editor.
21. Go to the **Content** > **ThirdPersonCPP** > **Blueprints** directory in the Content Browser, right-click, and create a new Blueprint class.
22. Expand the **All Classes** section and search for the **DodgeballProjectile** class, then set it as the parent class.
23. Name the new Blueprint class **BP_DodgeballProjectile**.
24. Open this new Blueprint class.
25. Notice the wireframe representation of the **SphereCollision** component in the actor's Viewport window (this is hidden by default during the game, but you can change that property in this component's **Rendering** section by changing its **HiddenInGame**

property):



Figure 6.8: Visual wireframe representation of the SphereCollision component

26. Now, add a new **Sphere** mesh as a child of the existing **Sphere Collision** component:

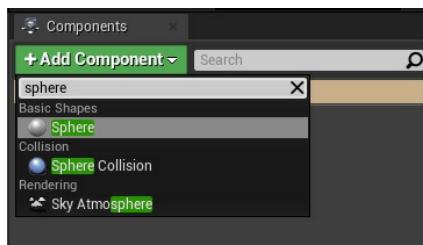


Figure 6.9: Adding a Sphere mesh

27. Change its scale to **0 . 65**, as shown in the following screenshot:



Figure 6.10: Updating the scale

28. Set its **Collision Presets** to

NoCollision:

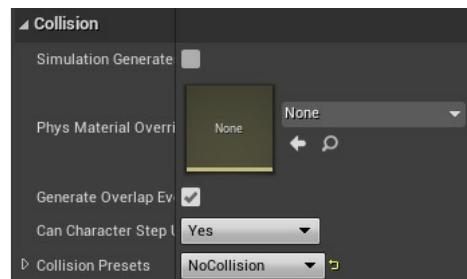


Figure 6.11: Updating Collision Presets to NoCollision

29. Finally, open our level and place an instance of the **BP_DodgeballProjectile** class near the player (this one was placed at a height of 600 units):

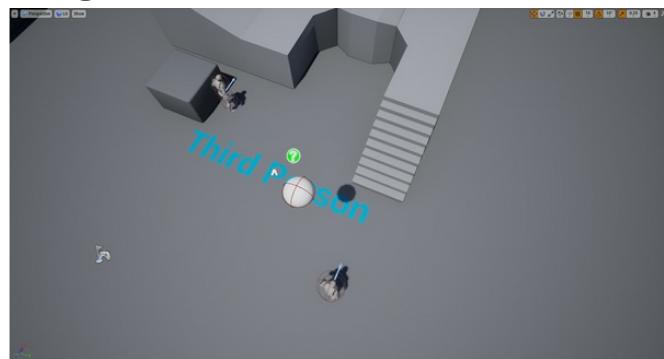


Figure 6.12: Dodgeball bouncing on the ground

After you've done this, play the level. You'll notice that the Dodgeball will be affected by gravity and bounce off the ground a couple of times before coming to a standstill.

By completing this exercise, you've created an object that behaves like a physics object.

You now know how to create your own collision object types,

use the **OnHit** event, and change an object's collision properties.

Note

*In the previous chapter, we briefly mentioned **LineTraceSingleByObjectType**. Now that we know how object collision works, we can briefly mention its use: when executing a Line Trace that checks for a Trace Channel, you should use the **LineTraceSingleByChannel** function; when executing a Line Trace that checks for an **Object Channel** (Object Type), you should use the **LineTraceSingleByObjectType** function. It should be made clear that this function, unlike the **LineTraceSingleByChannel** function, will not check for objects that block a specific Object Type, but those that are of a specific Object Type. Both those functions have the exact same parameters and both the Trace Channels and Object Channels are available through the **ECollisionChannel** enum.*

But what if you wanted the ball to bounce off the floor more times? What if you wanted to make it bouncier? Well, that's where Physical Materials come in.

Physical Materials

In UE4, the way you can customize how an object behaves while simulating physics is through Physical Materials. In order to get into this new type of asset, let's create our own:

1. Create a new folder inside the **Content** folder called **Physics**.

2. Right-click on the **Content Browser** while inside that folder and, under the **Create Advanced Asset** section, go to the **Physics** subsection and select **Physical Material**.
3. Name this new Physical Material **PM_Dodgeball**.
4. Open the asset and take a look at the available options.

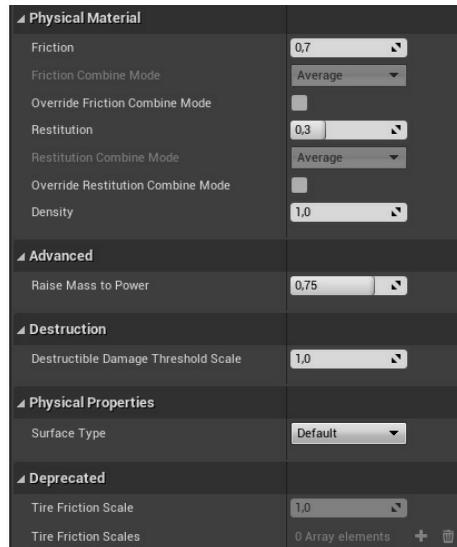


Figure 6.13: Asset options

The main options we should note are as follows:

- **Friction:** This property goes from **0** to **1** and specifies how much friction will affect this object (**0** means this object will slide as if it was on ice,

while **1** means this object will stick like a piece of gum).

- **Restitution** (also known as *Bounciness*): This property goes from **0** to **1** and specifies how much velocity will be kept after colliding with another object (**0** means this object will never bounce off of the ground, while **1** means this object will bounce for a long time).
- **Density**: This property specifies how dense this object is (that is, how heavy it is relative to its mesh). Two objects can be of the same size, but if one is twice as dense as the other, that means it will be twice as heavy.

To have our **DodgeballProjectile** object behave closer to an actual Dodgeball, it'll have to suffer quite a bit of friction (the default value is **0 . 7**, which is high enough) and be quite bouncy. Let's increase the **Restitution** property of this Physical Material to **0 . 95**.

After you've done this, open the **BP_DodgeballProjectile** Blueprint class and change the Sphere Collision component's Physical Material, inside its **Collision** section, to the one we just created, **PM_Dodgeball**:



Figure 6.14: Updating the BP_DodgeballProjectile Blueprint class

Note

Make sure the instance of the Dodgeball actor you added to your level also has this physical material.

If you play the level that we created in *Exercise 6.01, Creating the Dodgeball Class* again, you'll notice that our **BP_DodgeballProjectile** will now bounce off the ground several times before coming to a standstill, behaving much more like an actual dodgeball.

With all that done, we're just missing one thing to make our **Dodgeball** actor behave like an actual dodgeball. Right now, there is no way for us to be able to throw it. So, let's address that by creating a Projectile Movement Component, which is what we'll be doing in the next exercise.

In the previous chapters, when we replicated the Third Person template project, we learned that the **Character** class that comes with UE4 has a **CharacterMovementComponent**. This actor component is what allows an actor to move around in the level in various ways, and has many properties that allow you to customize that to your preference. However, there is another movement component that is also frequently used: **ProjectileMovementComponent**.

The **ProjectileMovementComponent** actor component is

used to attribute the behavior of a projectile to an actor. It allows you to set an initial speed, gravity force, and even some physics simulation parameters such as **Bounciness** and **Friction**. However, given that our **Dodgeball Projectile** is already simulating physics, the only property that we'll be using is **InitialSpeed**.

Exercise 6.02: Adding a Projectile Movement Component to DodgeballProjectile

In this exercise, we will be adding a **ProjectileMovementComponent** to our **DodgeballProjectile** so that it has an initial horizontal speed. We're doing this so that it can be thrown by our enemies and doesn't just fall vertically.

The following steps will help you complete this exercise:

1. Add a **ProjectileMovementComponent** property to the **DodgeballProjectile** class's header file:

```
UPROPERTY(VisibleAnywhere,  
BlueprintReadOnly, Category  
= Dodgeball, meta =  
(AllowPrivateAccess =
```

```
"true"))  
  
class  
UProjectileMovementComponen  
t* ProjectileMovement;
```

2. Include the
ProjectileMovementComponent
class at the top of the class's source file:

```
#include  
"GameFramework/ProjectileMo  
vementComponent.h"
```

3. At the end of the class's constructor,
create the
ProjectileMovementComponent
object:

```
ProjectileMovement =  
CreateDefaultSubobject<UPro  
jectileMovementComponent>  
(TEXT("Projec tile  
Movement"));
```

4. Then, set its **InitialSpeed** to **1500**
units:

```
ProjectileMovement -  
>InitialSpeed = 1500.f;
```

After you've done this, compile your project and open the

editor. To demonstrate the Dodgeball's initial speed, lower its position on the Z axis and place it behind the player (*this one was placed at a height of 200 units*):



Figure 6.15: Dodgeball moving along the X axis

When you play the level, you'll notice that the Dodgeball starts moving towards its *X* axis (*red arrow*):

And with that, we can conclude our exercise. Our **DodgeballProjectile** now behaves like an actual dodgeball. It falls, it bounces, and gets thrown.

The next step in our project is going to be adding logic to our **EnemyCharacter** so that it throws these dodgeballs at the player, but before we address that, we must address the concept of timers.

Timers

Given the nature of video games and the fact that they're strongly event-based, every game development tool must have a way for you to cause a delay, or a wait time, before something happens. For instance, when you're playing an online death match game, where your character can die and then respawn, usually, the respawn event doesn't happen the instant your character dies but a few seconds later. There is a

multitude of scenarios where you want something to happen, but only after a certain amount of time. This will be the case for our **EnemyCharacter**, which will be throwing dodge balls every few seconds. This delay, or wait time, can be achieved through timers.

A **timer** allows you to call a function after a certain amount of time. You can choose to loop that function call with an interval and also set a delay before the loop starts. If you want the Timer to stop, you can also do that.

We will be using timers so that our enemy throws a dodge ball every **X** amount of time, indefinitely, as long as it can see the player character, and then stop that timer when the enemy can no longer see its target.

Before we start adding logic to our **EnemyCharacter** class that will make it throw dodge balls at the player, we should take a look at another topic, which is how to spawn actors.

Spawning Actors

In *Chapter 1, Unreal Engine Introduction*, you learned how to place an actor that you created in the level through the editor, but what if you wanted to place that actor in the level as the game is being played? That's what we're going to be taking a look at now.

UE4, much like most other game development tools, allows you to place an actor in the game while the game itself is running. This process is called **spawning**. In order to spawn an actor in UE4, we need to call the **SpawnActor** function, available from the **World** object (which we can access using the **GetWorld** function, as mentioned previously). However, the **SpawnActor** function has a few parameters that need to be passed, as follows:

- A **UClass*** property, which lets the function know the class of the object that will be spawned. This property can be a C++ class, available through the **NameOfC++Class::StaticClass()** function, or a Blueprint class, available through the **TSubclassOf** property. It is generally a good practice not to spawn actors from a C++ class directly, but to create a Blueprint class and spawn an instance of that instead.
- The **TSubclassOf** property is a way for you to reference a Blueprint class in C++. It's used for referencing a class in C++ code, which might be a Blueprint class. You declare a **TSubclassOf** property with a template parameter, which is the C++ class that class must inherit from. We will be taking a look at how to use this property in practice in the next exercise.
- Either an **FTransform** property or the **FVector** and **FRotator** properties, which will indicate the location, rotation, and scale of the object we want to spawn.
- An optional

FActorSpawnParameters property, which allows you to specify more properties specific to the spawning process, such as who caused the actor to spawn (that is, the **Instigator**), how to handle the object spawning if the location that it spawns at is being occupied by other objects, which may cause an overlap or a block event, and so on.

The **SpawnActor** function will return an instance to the actor that was spawned from this function. Given that it is also a template function, you can call it in such a way that you receive a reference to the type of actor you spawned directly using a template parameter:

```
GetWorld()->SpawnActor<NameOfC++Class>
(ClassReference, SpawnLocation,
SpawnRotation);
```

In this case, the **SpawnActor** function is being called, where we're spawning an instance of the **NameOfC++Class** class. Here, we provide a reference to the class with the **ClassReference** property and the location and rotation of the actor to be spawned using the **SpawnLocation** and **SpawnRotation** properties, respectively.

You will learn how to apply these properties in *Exercise 6.03, Adding Projectile-Throwing Logic to the EnemyCharacter*.

Before we continue to the exercise, though, I'd like to briefly mention a variation of the **SpawnActor** function that may also come in handy: the **SpawnActorDeferred** function. While the **SpawnActor** function will create an instance of the

object you specify and then place it in the world, this new **SpawnActorDeferred** function will create an instance of the object you want, and only place it in the world when you call the actor's **FinishSpawning** function.

For instance, let's say we want to change the **InitialSpeed** of our Dodgeball at the moment we spawn it. If we used the **SpawnActor** function, there's a chance that the Dodgeball will start moving before we set its **InitialSpeed** property. However, by using the **SpawnActorDeferred** function, we can create an instance of the dodge ball, then set its **InitialSpeed** to whatever we want, and only then place it in the world by calling the newly created dodgeball's **FinishSpawning** function, whose instance is returned to us by the **SpawnActorDeferred** function.

Now that we know how to spawn an actor in the world, and also about the concept of timers, we can add the logic that's responsible for throwing dodge balls to our **EnemyCharacter** class, which is what we'll be doing in the next exercise.

Exercise 6.03: Adding Projectile-Throwing Logic to the **EnemyCharacter**

In this exercise, we will be adding the logic that's responsible for throwing the Dodgeball actor that we just created to our **EnemyCharacter** class.

Open the class's files in Visual Studio in order to get started. We will begin by modifying our **LookAtActor** function so that we can save the value that tells us whether we can see the player and use it to manage our timer.

Follow these steps to complete this exercise:

1. In the **EnemyCharacter** class's header file, change the **LookAtActor** function's return type from **void** to **bool**:

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

2. Do the same in the function's implementation, inside the class's source file, while also returning **true** at the end of the **if** statement where we call the **CanSeeActor** function. Also, return **false** in the first **if** statement where we check if **TargetActor** is a **nullptr** and also at the end of the function:

```
bool  
AEnemyCharacter::LookAtActo  
r(AActor * TargetActor)  
{
```

```

        if (TargetActor ==
nullptr) return false;

        if
(CanSeeActor(TargetActor))

{

    FVector Start =
GetActorLocation();

    FVector End =
TargetActor-
>GetActorLocation();

    // Calculate the
necessary rotation for the
Start point to face the End
point

    FRotator LookAtRotation
=
UKismetMathLibrary::FindLoo
kAtRotation(Start, End);

    //Set the enemy's
rotation to that rotation

    SetActorRotation(LookAt
Rotation);

    return true;
}

```

```
    return false;  
}
```

3. Next, add two **bool** properties, **bCanSeePlayer** and **bPreviousCanSeePlayer**, set to **protected** in your class's header file, which will represent whether the player can be seen in this frame from the enemy character's perspective and whether the player could be seen in the last frame, respectively:

```
//Whether the enemy can see  
the player this frame  
  
bool bCanSeePlayer = false;  
  
//Whether the enemy could  
see the player last frame  
  
bool bPreviousCanSeePlayer  
= false;
```

4. Then, go to your class's **Tick** function implementation and set the value of **bCanSeePlayer** to the return value of the **LookAtActor** function. This will replace the previous call to the **LookAtActor** function:

```
// Look at the player  
character every frame  
  
bCanSeePlayer =  
LookAtActor(PlayerCharacter  
);
```

5. After that, set the value of **bPreviousCanSeePlayer** to the value of **bCanSeePlayer**:

```
bPreviousCanSeePlayer =  
bCanSeePlayer;
```

6. In-between the previous two lines, add an **if** statement that checks whether the values of **bCanSeePlayer** and **bPreviousCanSeePlayer** are different. This will mean that either we couldn't see the player last frame and now we can, or that we could see the player last frame and now we can't:

```
bCanSeePlayer =  
LookAtActor(PlayerCharacter  
);  
  
if (bCanSeePlayer !=  
bPreviousCanSeePlayer)  
{
```

```
}

bPreviousCanSeePlayer =
bCanSeePlayer;
```

7. Inside this **if** statement, we want to start a timer if we can see the player and stop that timer if we can no longer see the player:

```
if (bCanSeePlayer !=  
bPreviousCanSeePlayer)  
{  
    if (bCanSeePlayer)  
    {  
        //Start throwing  
        dodgeballs  
    }  
    else  
    {  
        //Stop throwing  
        dodgeballs  
    }  
}
```

8. In order to start a timer, we'll need to

add the following properties to our class's header file, which can all be **protected**:

1. An **FTimerHandle** property, which is responsible for identifying which timer we want to start. It basically works as the identifier of a specific timer:

```
FTimerHandle  
ThrowTimerHandle;
```

2. A **float** property, which represents the amount of time to wait between throwing dodgeballs (the interval) so that we can loop the timer. We give this a default value of **2** seconds:

```
float  
ThrowingInterval =  
2.f;
```

3. Another **float** property, which represents the initial delay before the timer starts looping. Let's give it a default value of **0.5** seconds:

```
float ThrowingDelay  
= 0.5f;
```

4. A function to be called every time the timer ends, which we will create and call **ThrowDodgeball**. This function doesn't return anything and doesn't receive any parameters:

```
void  
ThrowDodgeball();
```

Before we can call the appropriate function to start the timer, we will need to add an **#include** to the object responsible for that, **FTimerManager**, in our source file.

Each **World** has one Timer Manager, which can start and stop timers and access relevant functions related to them, such as whether they're still active, how long they're running for, and so on:

```
#include  
"TimerManager.h"
```

9. Now, access the current World's Timer Manager by using the **GetWorldTimerManager** function:

```
GetWorldTimerManager()
```

10. Next, call the **SetTimer** function of the Timer Manager, if we can see the player character, in order to start the timer responsible for throwing dodge balls. The **SetTimer** function receives the following parameters:

1. The **FTimerHandle** representing the desired timer: **ThrowTimerHandle**.
2. The object that the function to be called belongs to: **this**.
3. The function to be called, which must be specified by prefixing its name with **&ClassName::**, resulting in **&AEnemyCharacter::ThrowDodgeball**.
4. The timer's rate, or interval:

ThrowingInterval.

5. Whether this timer will loop:
true.
6. The delay before this timer starts looping:
ThrowingDelay.

The following code snippet comprises these parameters:

```
if (bCanSeePlayer)

{
    //Start throwing
    dodgeballs

    GetWorldTimerManager()
        .SetTimer(ThrowT
        imerHandle, this,
        &AEnemyCharacter::Th
        rowDodgeball, Throwin
        gInterval, true,
        ThrowingDelay);

}
```

11. If we can no longer see the player and we want to stop the timer, we can do so using the **ClearTimer** function. This function only needs to receive an

FTimerHandle property as a parameter:

```
else

{
    //Stop throwing
    dodgeballs

    GetWorldTimerManager().ClearTimer(ThrowTimerHandle);

}
```

The only thing left is to implement the **ThrowDodgeball** function. This function will be responsible for spawning a new **DodgeballProjectile** actor. In order to do this, we'll need a reference to the class we want to spawn, which must inherit from **DodgeballProjectile**, so the next thing we need to do is create the appropriate property using the **TSubclassOf** object.

12. Create the **TSubclassOf** property in the **EnemyCharacter** header file, which can be **public**:

```
//The class used to spawn a
dodgeball object

UPROPERTY(EditDefaultsOnly,
BlueprintReadOnly, Category
= Dodgeball)

TSubclassOf<class
ADodgeballProjectile>
DodgeballClass;
```

13. Because we'll be using the **DodgeballProjectile** class, we also need to include it in the **EnemyCharacter** source file:

```
#include
"DodgeballProjectile.h"
```

14. Then, within the **ThrowDodgeball** function's implementation in the source file, start by checking if this property is a **nullptr**. If it is, we **return** immediately:

```
void
AEnemyCharacter::ThrowDodge
ball()

{
    if (DodgeballClass ==
```

```
nullptr)  
{  
    return;  
}  
}
```

15. Next, we will be spawning a new actor from that class. Its location will be **40** units in front of the enemy and its rotation will be the same as the enemy. In order to spawn the Dodgeball in front of the enemy character, we'll need to access the enemy's **ForwardVector** property, which is a unitary **FVector** (*meaning that its length is 1*) that indicates the direction an actor is facing, and multiply it by the distance at which we want to spawn our dodgeball, which is **40** units:

```
FVector ForwardVector =  
GetActorForwardVector();  
  
float SpawnDistance = 40.f;  
  
FVector SpawnLocation =  
GetActorLocation() +  
(ForwardVector *
```

```
SpawnDistance);  
//Spawn new dodgeball  
GetWorld()->SpawnActor<ADodgeballProjectile>(DodgeballClass,  
SpawnLocation,  
GetActorRotation());
```

This concludes the modifications we need to make to the **EnemyCharacter** class. Before we finish setting up the Blueprint of this logic, let's make a quick modification to our **DodgeballProjectile** class.

16. Open the **DodgeballProjectile** class's source file in Visual Studio.
17. Within its **BeginPlay** event, set its **LifeSpan** to **5** seconds. This property, which belongs to all actors, dictates how much longer they will remain in the game before being destroyed. By setting our dodgeball's **LifeSpan** to **5** seconds on its **BeginPlay** event, we are telling UE4 to destroy that object 5 seconds after it's spawned (*or, if it's already been placed in the level, 5*

seconds after the game starts). We will do this so that the floor isn't filled with dodge balls after a certain amount of time, which would make the game unintentionally difficult for the player:

```
void  
ADodgeballProjectile::Begin  
Play()  
{  
    Super::BeginPlay();  
  
    SetLifeSpan(5.f);  
}
```

Now that we've finished our C++ logic related to the **EnemyCharacter** class's Dodgeball throwing logic, let's compile our changes, open the editor, and then open our **BP_EnemyCharacter** Blueprint. There, head to the **Class Defaults** panel and change the **DodgeballClass** property's value to **BP_DodgeballProjectile**:

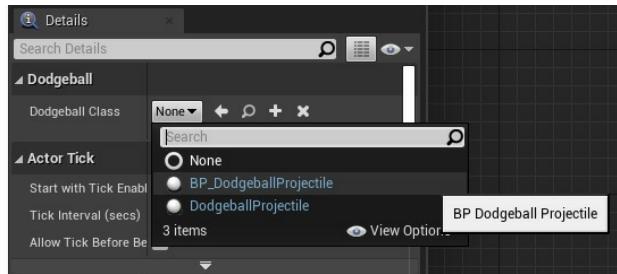


Figure 6.16: Updating the Dodgeball Class

After you've done this, you can remove the existing instance of the **BP_DodgeballProjectile** class we had placed in our level, if it's still there.

Now, we can play our level. You'll notice that the enemy will almost immediately start throwing dodge balls at the player and will continue to do so as long as the player character is in view:

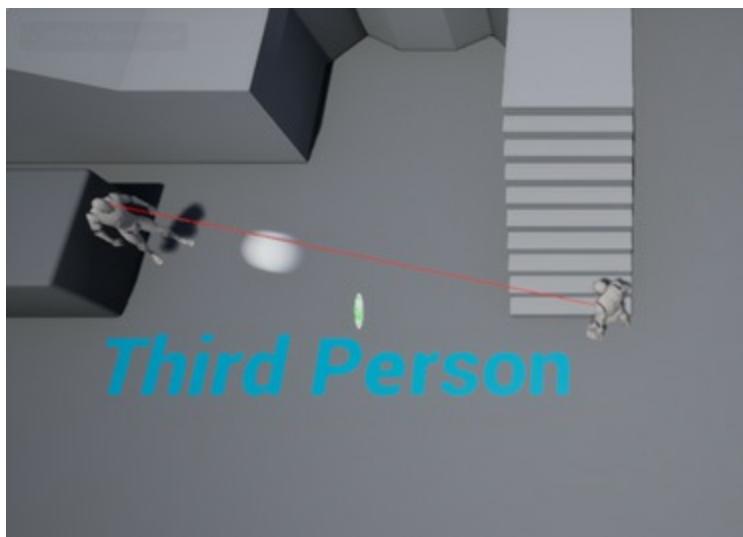


Figure 6.17: Enemy character throwing dodgeballs if the player is in sight

With that, we have concluded our dodge ball-throwing logic for the **EnemyCharacter**. You now know how to use timers, an essential tool for any game programmer.

Walls

The next step in our project is going to be creating the **Wall** classes. We will have two types of walls:

- A normal wall, which will block the enemy's line of sight, the player character, and the dodge ball.
- A ghost wall, which will only block the player character, and ignore the enemy's line of sight and the dodge ball. You might find this type of collision setup in specific types of puzzle games.

We'll create both these Wall classes in the next exercise.

Exercise 6.04: Creating Wall Classes

In this exercise, we will be creating the **Wall** classes that represent both a normal **Wall** and a **GhostWall**, which will only block the player character's movement, but not the enemies' lines of sight or the dodge balls they throw.

Let's start with the normal **Wall** class. This C++ class will basically be empty because the only thing that it'll need is a mesh in order to reflect the projectiles and block the enemies' lines of sight, which will be added through its Blueprint class.

The following steps will help you complete this exercise:

1. Open the editor.
2. In the top-left corner of the Content Browser, press the green **Add New** button.
3. Select the first option at the top; **Add Feature or Content Pack**.
4. A new window will show up. Select the **Content Packs** tab, then select the **Starter Content** pack and then press the **Add To Project** button.
This will add some basic assets to the project, which we'll use in this chapter and some of the following chapters.
5. Create a new C++ class, called **Wall**, with the **Actor** class as its parent.
6. Next, open the class's files in Visual Studio and add a **SceneComponent** as our Wall's **RootComponent**:
 1. The **Header** file will be as follows:

```
private:  
UPROPERTY(VisibleAny
```

```
where,  
BlueprintReadOnly,  
Category = Wall,  
meta =  
(AllowPrivateAccess  
= "true"))  
  
class  
USceneComponent*  
RootScene;
```

2. The **Source** file will be as follows:

```
AWall::AWall()  
{  
    // Set this actor  
    to call Tick() every  
    frame. You can turn  
    this off to improve  
    performance if you  
    don't need it.  
  
    PrimaryActorTick.b  
    CanEverTick = true;  
  
    RootScene =  
CreateDefaultSubobje  
ct<USceneComponent>  
(TEXT("Root"));
```

```
    RootComponent =  
    RootScene;  
  
}
```

7. Compile your code and open the editor.
8. Next, go to the **Content > ThirdPersonCPP >:Blueprints** directory inside the Content Browser, create a new Blueprint class that inherits from the **Wall** class, name it **BP_Wall**, and open that asset.
9. Add a Static Mesh Component and set its **StaticMesh** property to **Wall_400x300**.
10. Set its **Material** property to **M_Metal_Steel**.
11. Set the Static Mesh Component's location on the **X** axis to **-200** units (*so that the mesh is centered relative to our actor's origin*):

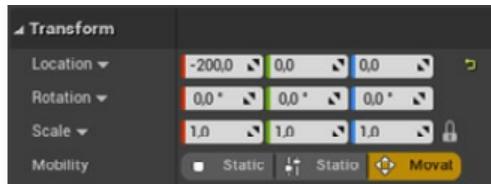


Figure 6.18: Updating the Static Mesh Component's location

This is what your Blueprint class's Viewport should look like:

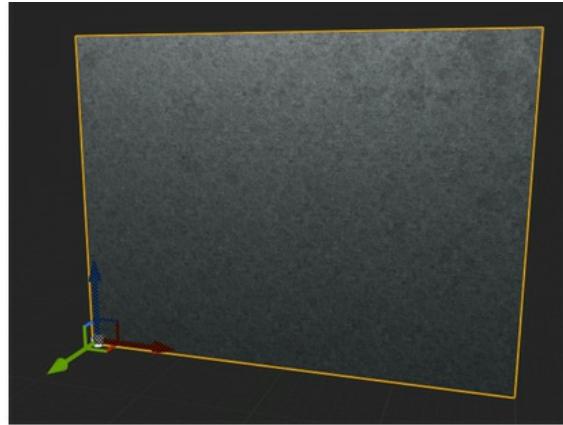


Figure 6.19: Blueprint class's Viewport Wall

Note

*It is generally good practice to add a **SceneComponent** as an object's **RootComponent**, when a collision component isn't necessary, in order to allow for more flexibility with its child components.*

*An actor's **RootComponent** cannot have its location or rotation modified, which is why, in our case, if we had created a *Static Mesh Component* in the **Wall C++ class** and set that as its Root Component, instead of using a *Scene Component*, we'd have a hard time offsetting it.*

Now that we've set up the regular **Wall** class, let's create our **GhostWall** class. Because these classes don't have any logic set up, we're just going to create the **GhostWall** class as a child of the **BP_Wall** Blueprint class and not our C++ class.

1. Right-click the **BP_Wall** asset and

select **Create Child Blueprint Class**.

2. Name the new Blueprint **BP_GhostWall**.
3. Open it.
4. Change the Static Mesh Component's Collision properties:
 1. Set its **CollisionPreset** to **Custom**.
 2. Change its response to both the **EnemySight** and **Dodgeball** channels to **Overlap**.
5. Change the Static Mesh Component's **Material** property to **M_Metal_Copper**.

Your **BP_GhostWall**'s Viewport should now look like this:

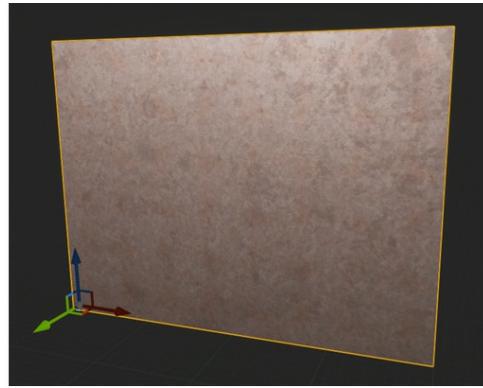


Figure 6.20: Creating the Ghost Wall

Now that you've created both these Wall actors, place each in the level to test them. Set their transforms to the following transform values:

- Wall: **Location: (-710, 120, 130)**
- Ghost Wall: **Location: (-910, -100, 130); Rotation: (0, 0, 90):**

Transform			Transform		
Location ▾	X -710.0	Y 120.0	Z 130.0	Location ▾	X -910.0
Rotation ▾	X 0.0°	Y 0.0°	Z 0.0°	Rotation ▾	X 0.0°
Scale ▾	X 1.0	Y 1.0	Z 1.0	Scale ▾	X 1.0

Figure 6.21: Updating the Ghost Wall's locations and rotation

The final outcome should look like this:

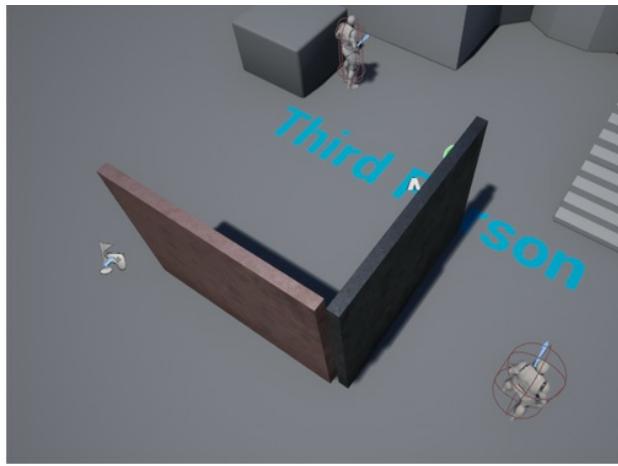


Figure 6.22: Final outcome with the Ghost Wall and the Wall

You'll notice that when you hide your character behind the normal **Wall** (the one on the right), the enemy won't throw dodgeballs at the player; however, when you try to hide your character behind the **GhostWall** (the one on the left), even though the enemy can't go through it, the enemy will throw dodgeballs at the character and they will pass through the Wall as if it wasn't there!

And that concludes our exercise. We have made our **Wall** actors, which will either behave normally or ignore the enemies' lines of sight and dodgeballs!

Victory Box

The next step in our project is going to be creating the **VictoryBox** actor. This actor will be responsible for ending the game when the player character enters it, given that the player has beaten the level. In order to do this, we'll be using the **Overlap** event. The following exercise will help us understand Victory Box.

Exercise 6.05: Creating the VictoryBox class

In this exercise, we will be creating the **VictoryBox** class, which, when entered by the player character, will end the game.

The following steps will help you complete this exercise:

1. Create a new C++ class that inherits from the actor and call it **VictoryBox**.
2. Open that class's files in Visual Studio.
3. Create a new **SceneComponent** property, which will be used as a **RootComponent**, just like we did with our **Wall** C++ class:

1. **Header** file:

```
private:  
    UPROPERTY(VisibleAny  
where,  
BlueprintReadOnly,  
Category =  
VictoryBox, meta =  
(AllowPrivateAccess  
= "true"))  
  
class
```

```
USceneComponent*  
RootScene;
```

2. **Source** file:

```
AVictoryBox::AVictor  
yBox()  
{  
    // Set this actor  
    to call Tick() every  
    frame. You can turn  
    this off to improve  
    performance if you  
    don't need it.  
  
    PrimaryActorTick.b  
    CanEverTick = true;  
  
    RootScene =  
    CreateDefaultSubobje  
    ct<USceneComponent>  
    (TEXT("Root"));  
  
    RootComponent =  
    RootScene;  
}
```

4. Declare a **BoxComponent** in the header file that will check for overlap events with the player character, which

should also be **private**:

```
UPROPERTY(VisibleAnywhere,  
BlueprintReadOnly, Category  
= VictoryBox, meta =  
(AllowPrivateAccess =  
"true"))  
  
class UBoxComponent*  
CollisionBox;
```

5. Include the **BoxComponent** file in the class's source file:

```
#include  
"Components/BoxComponent.h"
```

6. After creating the **RootScene** Component, create the **BoxComponent**, which should also be **private**:

```
RootScene =  
CreateDefaultSubobject<USce  
neComponent>(TEXT("Root"));  
  
RootComponent = RootScene;  
  
CollisionBox =  
CreateDefaultSubobject<UBox  
Component>(TEXT("Collision  
Box"));
```

7. Attach it to the **RootComponent** using the **SetupAttachment** function:

```
CollisionBox-
>SetupAttachment(RootCompon
ent);
```

8. Set its **BoxExtent** property to **60** units on all axes. This will cause the **BoxComponent** to be double that size (**120 x 120 x 120**):

```
CollisionBox-
>SetBoxExtent(FVector(60.0f
, 60.0f, 60.0f));
```

9. Offset its relative position on the *Z* axis by **120** units using the **SetRelativeLocation** function:

```
CollisionBox-
>SetRelativeLocation(FVecto
r(0.0f, 0.0f, 120.0f));
```

10. Now, you will require a function that will listen to the **BoxComponent**'s **OnBeginOverlap** event. This event will be called whenever an object enters the **BoxComponent**. This function must be preceded by the **UFUNCTION** macro, be **public**, return

nothing, and have the following parameters:

```
UFUNCTION()  
void  
OnBeginOverlap(UPrimitiveCo  
mponent* OverlappedComp,  
AActor* OtherActor,  
UPrimitiveComponent*  
OtherComp, int32  
OtherBodyIndex, bool  
bFromSweep, const  
FHitResult& SweepResult);
```

The parameters are as follows:

1. **UPrimitiveComponent***
OverlappedComp: The component that was overlapped and belongs to this actor.
2. **AActor* OtherActor**: The other actor involved in the overlap.
3. **UPrimitiveComponent***
OtherComp: The component that was overlapped and belongs to the other actor.

4. **int32 OtherBodyIndex:**

The index of the item in the primitive that was hit
(usually useful for Instanced Static Mesh components).

5. **bool bFromSweep:**

Whether the overlap originated from a Sweep Trace.

6. **FHitResult&**

SweepResult: The data of the Sweep Trace resulting from the collision between this object and the other object.

Note

*Although we won't be using the **OnEndOverlap** event in this project, you will most likely need to use it sooner or later, so here's the required function signature for that event, which looks very similar to the one we just learned about:*

```
UFUNCTION( )  
void  
OnEndOverlap(UPrimitiveComponent*  
OverlappedComp,  
AActor* OtherActor,  
UPrimitiveComponent*  
OtherComp, int32  
OtherBodyIndex);
```

11. Next, we need to bind this function to the **BoxComponent**'s **OnComponentBeginOverlap** event:

```
CollisionBox->OnComponentBeginOverlap.AddDynamic(this,  
&AVictoryBox::OnBeginOverlap);
```

12. Within our **OnBeginOverlap** function implementation, we're going to check whether the actor we overlapped is a **DodgeballCharacter**. Because we'll be referencing this class, we also need to include it:

```
#include
```

```

"DodgeballCharacter.h"

void
AVictoryBox::OnBeginOverlap
(UPrimitiveComponent *
OverlappedComp, AActor *
OtherActor,
UPrimitiveComponent *
OtherComp, int32
OtherBodyIndex, bool
bFromSweep, const
FHitResult & SweepResult)

{
    if
(Cast<ADodgeballCharacter>
(OtherActor))

{
}

}

```

If the actor we overlapped is a **DodgeballCharacter**, we want to quit the game.

13. We will use **KismetSystemLibrary** for this purpose. The **KismetSystemLibrary** class contains useful functions for general

use in your project:

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

14. In order to quit the game, we will call **KismetSystemLibrary**'s **QuitGame** function. This function receives the following:

```
UKismetSystemLibrary::QuitG  
ame(GetWorld(),  
nullptr,  
EQuitPreference::Quit,  
true);
```

The important parameters from the preceding code snippet are explained as follows:

1. A **World** object, which we can access with the **GetWorld** function.
2. A **PlayerController** object, which we will set to **nullptr**. We're doing this because this function will automatically find one this

way.

3. An **EQuitPreference**

object, which means the way in which we want to end the game, by either quitting or just putting it as a background process. We will want to actually quit the game, and not just put it as a background process.

4. A **bool**, which indicates whether we want to ignore the platform's restrictions when it comes to quitting the game, which we will set to **true**.

Next, we're going to create our Blueprint class.

15. Compile your changes, open the editor, go to the **Content** → **ThirdPersonCPP** → **Blueprint** directory inside the **Content Browser**, create a new Blueprint class that inherits from **VictoryBox**, and name it **BP_VictoryBox**. Open that asset and make the following

modifications:

1. Add a new Static Mesh Component.
2. Set its **StaticMesh** property to **Floor_400x400**.
3. Set its **Material** property to **M_Metal_Gold**.
4. Set its scale to **0.75** units on all three axes.
5. Set its location to **(-150, -150, 20)**, on the *X*, *Y*, and *Z* axes, respectively.

After you've made those changes, your Blueprint's Viewport tab should look something like this:

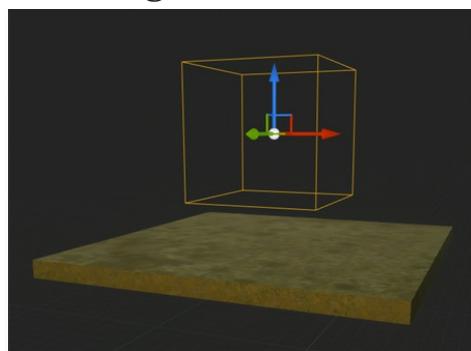


Figure 6.23: Victory box placed in the Blueprint's Viewport tab

Place that Blueprint in your level to test its functionality:

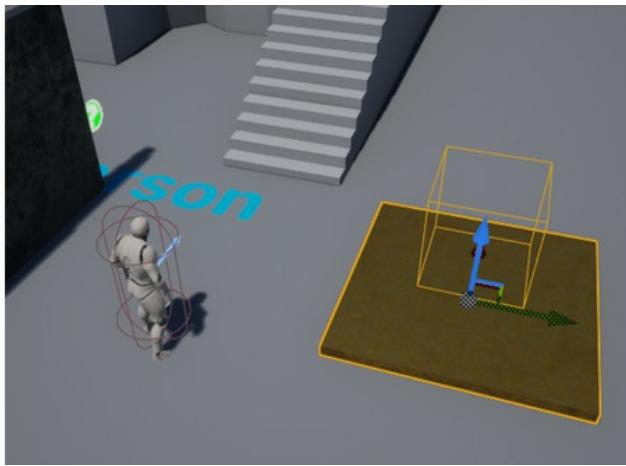


Figure 6.24: Victory Box blueprint in the level for testing

If you play the level and step onto the golden plate (and overlap the collision box), you'll notice that the game abruptly ends, as intended.

And with that, we conclude our **VictoryBox** class! You now know how to use the overlap events in your own projects. There's a multitude of game mechanics that you can create using these events, so congratulations on completing this exercise.

We are now very close to reaching the end of this chapter, where we'll be completing a new activity, but first, we'll need to make some modifications to our **DodgeballProjectile** class, namely adding a getter function to its **ProjectileMovementComponent**, which we'll be doing in the next exercise.

A getter function is a function that only returns a specific property and does nothing else. These functions are usually marked as inline, which means that, when the code compiles, a call to that function will simply be replaced with its content. They are also usually marked as **const**, given that they don't modify any of the class's properties.

Exercise 6.06: Adding the ProjectileMovementComponent Getter Function in DodgeballProjectile

In this exercise, we will be adding a getter function to the **DodgeballProjectile** class's **ProjectileMovement** property so that other classes can access it and modify its properties. We will be doing the same in this chapter's activity.

In order to do this, you'll need to follow these steps:

1. Open the **DodgeballProjectile** class's header file in Visual Studio.
2. Add a new **public** function called **GetProjectileMovementComponent**. This function will be an inline function, which in UE4's version of C++ is replaced with the **FORCEINLINE** macro. The function should also return a **UProjectileMovementComponent** * and be a **const** function:

```
FORCEINLINE class UProjectileMovementComponent* GetProjectileMovementCompon
```

```
ent() const  
{  
    return  
    ProjectileMovement;  
}
```

Note

*When using the **FORCEINLINE** macro for a specific function, you can't add the declaration of that function to the header file and its implementation to the source file. Both must be done simultaneously in the header file, as shown previously.*

With that, we conclude this quick exercise. Here, we have added a simple **getter** function to our **DodgeballProjectile** class, which we will be using in this chapter's activity, where we'll replace the **SpawnActor** function within the **EnemyCharacter** class with the **SpawnActorDeferred** function. This will allow us to safely edit our **DodgeballProjectile** class's properties before we spawn an instance of it.

Activity 6.01: Replacing the SpawnActor Function

with `SpawnActorDeferred` in `EnemyCharacter`

In this activity, you 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.

4. Then, update 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** properties we just created, as the second parameter.

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

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.

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

Expected output:

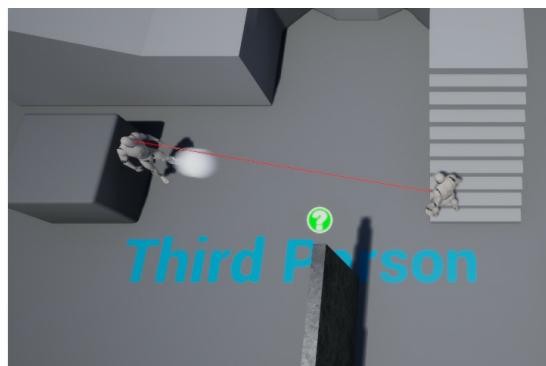


Figure 6.25: Dodgeball thrown at the player

Note

*The solution to this activity can be found at:
<https://packt.live/338jEBx>.*

By completing this activity, you've consolidated the use of the **SpawnActorDeferred** function and know how to use it in your future projects.

Summary

In this chapter, you've learned how to affect an object with physics simulations, create your own Object Types and Collision Presets, use the **OnHit**, **OnBeginOverlap**, and **OnEndOverlap** events, update an object's Physical Material, and use timers.

Now that you've learned these fundamental concepts of collision topics, you'll be able to come up with new and creative ways to use them when creating your own projects.

In the next chapter, we'll be taking a look at Actor Components, Interfaces, and Blueprint Function Libraries, which are very useful for keeping your project's complexity manageable and highly modular, thereby allowing you to easily take parts of one project and add them to another.

8. User Interfaces

Overview

In this chapter, we will continue our work on the Dodgeball-based game that we have been working on in the last few chapters. We will continue this project by learning about game UIs (short for user interfaces) and some of their forms, namely menus and HUDs. By the end of this chapter, you will be able to use UMG, UE4's game UI system, to make a menu with interactable buttons, as well as a HUD that displays the player character's current health points through a progress bar.

Introduction

In the previous chapter, we learned about general-purpose utilities that allow you to properly structure and organize the code and assets in your project by using Blueprint Function Libraries, Actor Components, and Interfaces.

In this chapter, we will dive into the topic of game UIs, which is something that's present in almost every video game. The game UI is one of the main ways to show information to the player, such as how many lives they have left, how many bullets are in their weapon, which weapon they are carrying, and so on, and to allow the player to interact with the game by choosing whether to continue the game, create a new game, choose which level they want to play in, and so on. This is shown to the player mostly in the form of images and text.

User Interfaces or **UIs** are usually added on top of the rendering of the game, which means that they are in front of everything else you see in the game and behave as layers (you can add them on top of one another just like in Photoshop). However, there is an exception to this: *diegetic UI*. This type of UI isn't layered onto the game's screen, but rather exists inside of the game itself. A great example of this can be found in the game *Dead Space*, where you control a character in a third-person view and can see their health points by looking at the contraption attached to their back, inside the game world.

Game UI

There are usually two different types of game UI: **menus** and **HUDs**.

Menus are UI panels that allow the player to interact with them, either by pressing a button or a key on their input device.

This can be done in the form of many different menus, including the following:

- Main menus, where the player can choose whether to continue the game, create a new game, exit the game, and so on
- Level select menus, where the player can choose which level to play
- And many other options

HUDs are UI panels that are present during gameplay that

give the player information that they should always know, such as how many lives they have left, which special abilities they can use, and so on.

We will be covering game UI and making both a menu and a HUD for our game in this chapter.

Note

We won't be covering diegetic UI here, as it is beyond the scope of this book.

So how do we go about creating a game UI in UE4? The main way to do that is by using **Unreal Motion Graphics (UMG)**, which is the tool that allows you to make a game UI (also called Widgets in UE4 terms) featuring menus and HUDs, and add them to the screen.

Let's jump into this topic in the following section.

UMG Basics

In UE4, the main way to create a game UI is by using the UMG tool. This tool will allow you to make a game UI in the form of **Widgets**, which can be created using UMG. It will allow you to easily edit your game UI in a visual manner, through UMG's **Designer** tab, while also allowing you to add functionality to your game UI through UMG's **Graph** tab.

Widgets are the way UE4 allows you to represent a game UI. Widgets can be basic UI elements such as **Buttons**, **Text** elements, and **Images**, but they can also be combined to create more complex and complete Widgets, such as menus and HUDs, which is exactly what we will be doing in this chapter.

Let's create our first Widget in UE4 using the UMG tool in the next exercise.

Exercise 8.01: Creating a Widget Blueprint

In this exercise, we will be creating our first Widget Blueprint, as well as learning the basic elements of UMG and how we can use them to create a game UI.

The following steps will help you complete this exercise:

1. In order to create our first Widget, open the editor, go to the **ThirdPersonCPP -> Blueprints** folder inside the **Content Browser**, and *right-click*.
2. Go to the very last section, **User Interface**, and select **Widget Blueprint**.

Selecting this option will create a new **Widget Blueprint**, which is the name for a Widget asset in UE4.

3. Name this Widget **TestWidget** and open it. You will see the interface for editing a Widget Blueprint, where you'll be creating your own Widgets

and UI. Here's a breakdown of all the tabs present in this window:

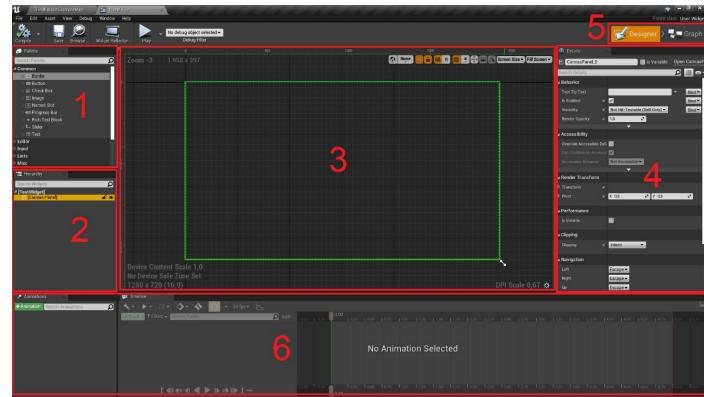


Figure 8.1: The Widget Blueprint editor broken down into six windows

The details about the tabs in the preceding figure are as follows:

1. **Palette** – This tab shows you all the individual UI elements that you can add to your Widget. This includes **Buttons**, **Text Boxes**, **Images**, **Sliders**, **Check Boxes**, and many more.
2. **Hierarchy** – This tab shows you all the UI elements currently present in your Widget. As you can see, currently we only have a

Canvas Panel element in our hierarchy.

3. **Designer** – This tab shows you how your Widget looks visually, according to the elements present in the hierarchy and how they're laid out. Because the only element we currently have in our Widget doesn't have a visual representation, this tab is currently empty.
4. **Details** – This tab shows you the properties of the UI element you have currently selected. If you select the existing **Canvas Panel** element, all the options in the preceding screenshot should appear.
5. Because this asset is a **Widget Blueprint**, these two buttons allow you to switch between the **Designer view**, which is the one presented in the screenshot, and the **Graph**

view, which looks exactly like the window of a normal Blueprint class.

6. **Animation** – Both these tabs are related to Widget animations. Widget Blueprints allow you to animate the properties of UI elements, including their **position, scale, color**, and so on, over time. The tab on the left allows you to create and select animations to edit in the right tab, where you'll be able to edit what properties they affect over time.

4. Let's now look at some of the available UI elements in our **Widget**, starting with the existing **Canvas Panel**.

Canvas Panels are usually added to the root of Widget Blueprints because they allow you to drag a UI element to any position you want in the **Designer** tab. This way, you can lay out these elements as you wish: at the

center of the screen, at the top-left corner, at the bottom center of the screen, and so on. Let's now drag another very important UI element into our Widget: a **Button**.

5. In the **Palette** tab, find the **Button** element and drag it into our **Designer** tab (hold the left mouse button while you drag):

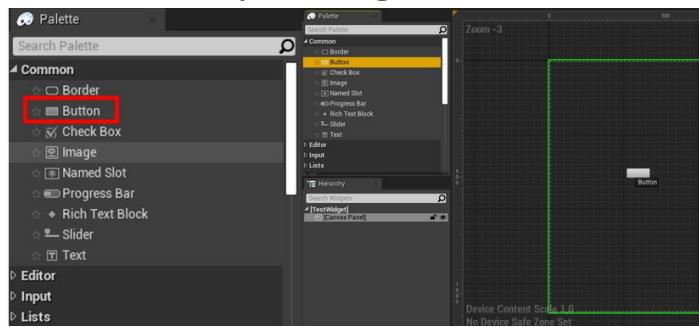


Figure 8.2: A Button element being dragged from the Palette window into the Designer window

Once you do this, you'll be able to resize the button to the size you want by dragging the little white dots around it (keep in mind that you'll only be able to do this to an element that is inside a Canvas Panel):

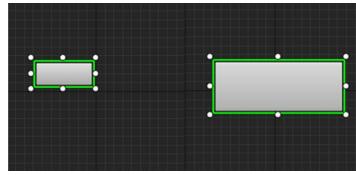


Figure 8.3: The result of resizing a UI element using the white dots around it

Another way for you to drag elements inside each other in a **Widget** is to drag them inside the **Hierarchy** tab, instead of the **Designer** tab.

6. Now drag a **Text** element inside our **Button**, but this time, use the **Hierarchy** tab:

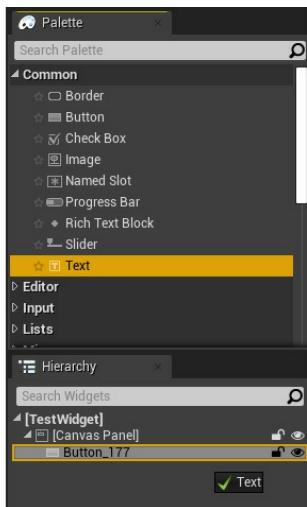


Figure 8.4: Dragging a Text element from the Palette window into the Hierarchy window

Text elements can contain text specified by you with a certain size and font that you can modify in the **Details** panel. After you've dragged the **Text** element inside the **Button** using the **Hierarchy** tab, this is what the **Designer** tab should look like:



Figure 8.5: The Button element in the Designer tab, after we add a Text element as its child

Let's change a few properties of this **Text** block.

7. Select it either in the **Hierarchy** tab or the **Designer** tab and take a look at the **Details** panel:

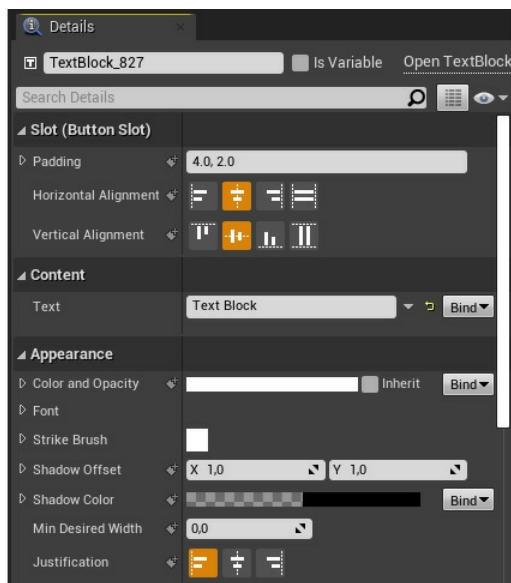


Figure 8.6: The Details panel, showing the properties of the Text element we added

Here you'll find several properties that you can edit to your liking. For now, we just want to focus on two of them: the **Content** of the text and its **Color and Opacity**.

8. Update the **Content** of the **Text** element from **Text Block** to **Button** 1:

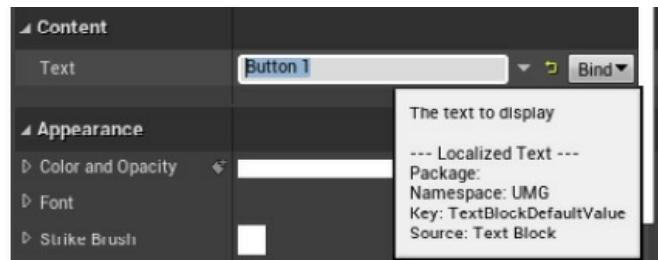


Figure 8.7: Changing the Text property of the Text element to Button 1

Next, let's change its **Color and Opacity** from **White** to **Black**.

9. Click the **Color and Opacity** property and take a look at the window that pops up, the **Color Picker**. This window pops up whenever you edit a

Color property in UE4. It allows you to input colors in many different ways, including a color wheel, a **Saturation** and **Value** bar, **RGB** and **HSV** value sliders, and a couple more options.

10. For now, change the color from white to black by dragging the **Value** bar (the one that goes from white to black from top to bottom) all the way to the bottom and then pressing **OK**:

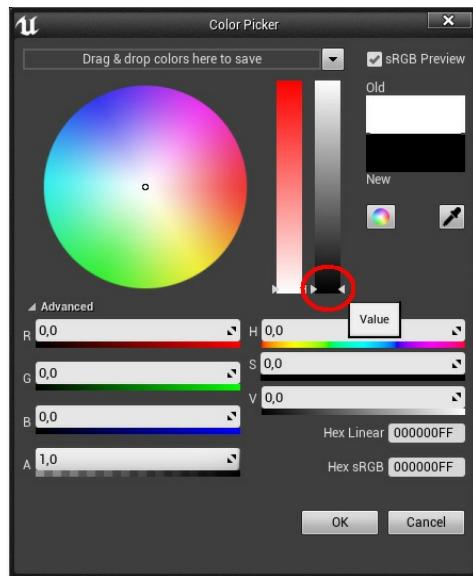


Figure 8.8: Selecting the color black in the Color Picker window

11. After these changes, this is what the button should look like:



Figure 8.9: The Button element after we change the Text element's Text property and its color

And with that, we conclude our first exercise for this chapter. You now know some of the essential basics of UMG, such as how to add **Button** and **Text** elements to your Widgets.

Before we jump into our next exercise, let's first learn about Anchors.

Anchors

As you might be aware, video games are played on many different screen sizes with many different resolutions. Because of that, it is important to make sure that the menus you create can adapt to all these different resolutions effectively. This is the main purpose of **Anchors**.

Anchors allow you to specify how you want a UI element's size to adapt as the screen resolution changes by specifying the proportion of the screen you want it to occupy. Using Anchors, you can have a UI element always at the top left of the screen, or always occupying half of the screen, no matter the size and resolution of that screen.

As the size of the screen or resolution changes, your Widget will scale and move relative to its Anchor. Only elements that are direct children of a **Canvas Panel** can have an Anchor, which you can visualize through the **Anchor Medallion**, a white flower-like shape in the **Designer** tab, when you select said element:

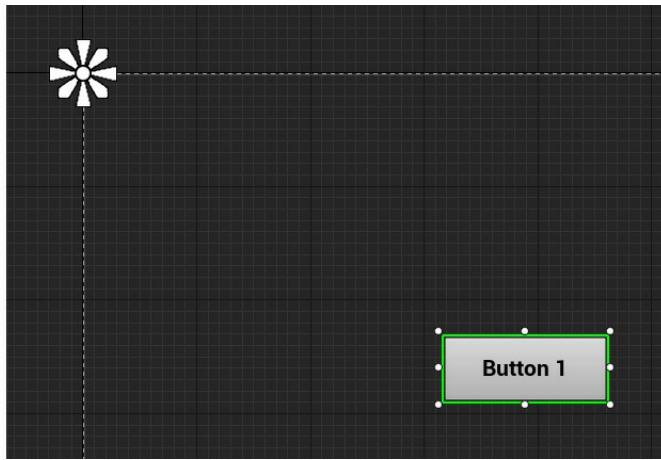


Figure 8.10: The Anchor Medallion at the top left of the outline shown in the Designer window

By default, the Anchor is collapsed into the top-left corner, which means that you won't have much control over how the button is scaled as the resolution changes, so let's change that in the next exercise.

Exercise 8.02: Editing UMG Anchors

In this exercise, we will be changing the Anchors in our Widget in order to have our Button's size and shape adapt to a wide range of screen resolutions and sizes.

The following steps will help you complete this exercise:

1. Select the Button we created in the previous exercise, then head to the **Details** panel and press the very first property you see, the **Anchors** property. Here you'll be able to see the

Anchor presets, which will align the UI element according to the pivots shown.

We'll want to have our button centered on the screen.

2. Click on the pivot that's at the center of the screen:

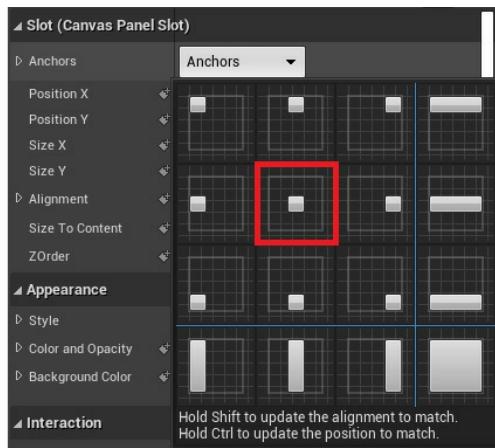


Figure 8.11: The Button's Anchors property, with the center Anchor outlined in a box

You'll see that our **Anchor Medallion** has now changed places:

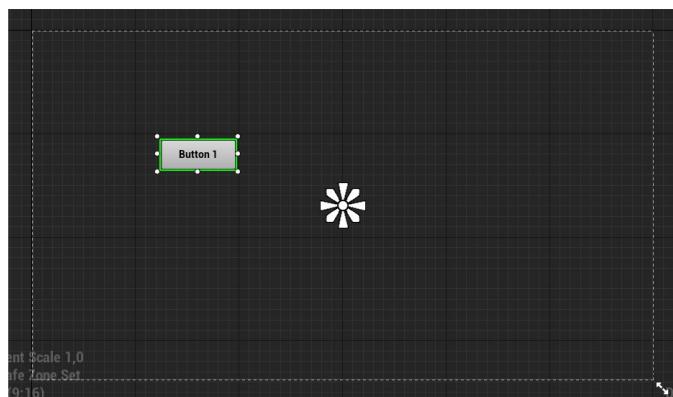


Figure 8.12: The Anchor Medallion after we change the Button's Anchor to the center

Now that the **Anchor Medallion** is at the center of the screen, we still won't have much control over how the Button will scale across different resolutions, but at least we know that it'll scale relative to the center of the screen.

In order to have our Button centered on the screen, we'll have to change the Button's position to be at the center of the screen as well.

3. Repeat the previous step of picking the center Anchor, but this time, before you select it, hold the *Ctrl* key in order to snap the Button's position to this Anchor. After you click it, release the *Ctrl* key. This should be the result:

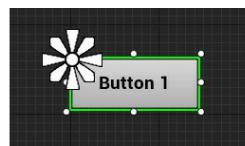


Figure 8.13: The Button element being moved near its selected Anchor in the center

As you can see from the preceding screenshot, our Button has changed position, but it isn't properly centered on the screen yet. This is because of its **Alignment**.

The **Alignment** property is of type **Vector2D** (a tuple with two **float** properties: **X** and **Y**) and dictates the center of the UI element relative to its total size. By default it's set to **(0, 0)**, meaning the center of the element is its top-left corner, which explains the result in the preceding screenshot. It can go all the way to **(1, 1)**, the bottom-right corner. In this case, given that we want the alignment to center the button, we want it to be **(0.5, 0.5)**.

4. In order to update a UI element's alignment when picking an **Anchor** point, you have to hold the *Shift* key and repeat the previous step.
Alternately, to update both the position and the alignment of the button, picking the center **Anchor** point while holding both the *Ctrl* and *Shift* keys will do the job. This should then be the

result:

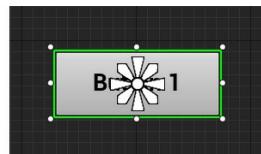


Figure 8.14: The Button element being centered relative to its selected Anchor in the center

At this point, when changing the resolution of the screen, we know that this button will always remain at the center of the screen. However, in order to maintain the Button's size relative to the resolution, we'll need to make a few more modifications.

5. Drag the bottom-right *petal* of the **Anchor Medallion** all the way to the bottom-right corner of the button:

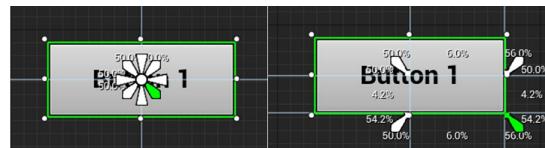


Figure 8.15: Dragging the lower-right petal of the Anchor Medallion to update the Button element's Anchor

6. Drag the top-left *petal* of the **Anchor Medallion** all the way to the top-left

corner of the button:

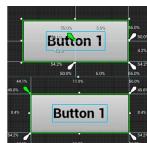


Figure 8.16: Dragging the upper-left petal of the Anchor Medallion to update the Button element's Anchor

Note

*The percentages you see around the button when changing the **Anchor** are the space the element is occupying on the screen as a percentage. For instance, looking at the last screenshot, we can see that the button is occupying **11.9%** of the Widget's space on the X coordinate and **8.4%** of the Widget's space on the Y coordinate.*

*You can set the size of a UI element to the size of its Anchor by holding the **Ctrl** key while moving the Anchor Medallion petals.*

Now our button will finally adapt to varying screen sizes and resolutions

due to these changes to its Anchor.

You can also use the **Details** panel to manually edit all of the properties we just edited by using the **Anchor Medallion** and moving the button:

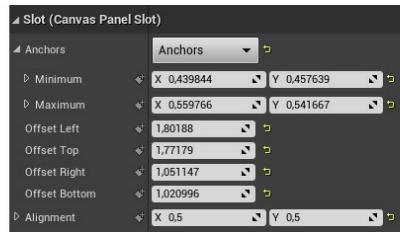


Figure 8.17: The properties we changed using the Anchor Medallion, shown in the Details window

Lastly, we need to know how we can visualize our Widget with different resolutions in the **Designer** tab.

7. Drag the double arrow at the bottom right of the outlined box inside the **Designer** tab:

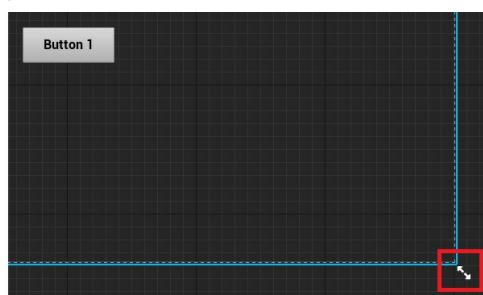


Figure 8.18: The double arrow at the bottom right of

the outlined box inside the Designer tab

By dragging the double arrow you can resize the **Canvas** to any screen resolution you want. In the following screenshot, you'll see the most used resolutions for a variety of devices, and you can preview your Widget in each of them:

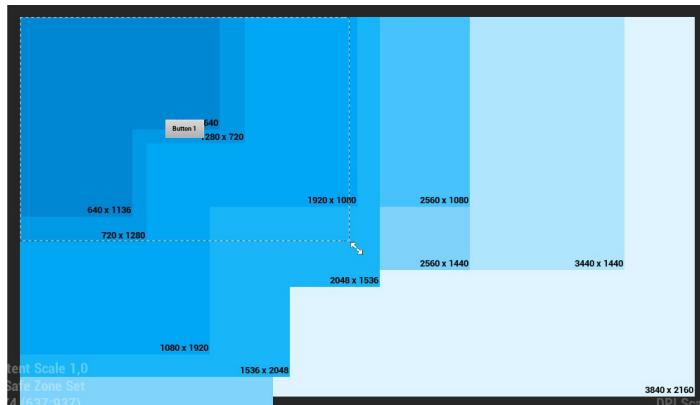


Figure 8.19: The resolutions we can choose to preview in the Designer window

Note

You can find a full reference to UMG's Anchors at <https://docs.unrealengine.com/en-US/Engine/UMG/UserGuide/Anchors>.

And that concludes our exercise. You've learned about Anchors and adapting your Widgets to varying screen sizes and resolutions.

Now that we've learned about some of the basics of UMG, let's see how we can create a Widget C++ class for this Widget Blueprint, which is what we're going to do in the next exercise.

Exercise 8.03: Creating

the RestartWidget C++ Class

In this exercise, we will learn how to create a Widget C++ class, from which the Widget Blueprint that we created will inherit from. It will get added to the screen when the player dies in our **Dodgeball** game so that the player can have the option to restart the level. This Widget will have a button that will restart the level when the player clicks it.

The first step of this exercise will be adding the UMG-related modules to our project. Unreal Engine comprises several different modules, and in each project, you have to specify which ones you're going to use. Our project came with a few general modules when the source code files were generated, but we'll need to add a few more.

The following steps will help you complete this exercise:

1. Open the **Dodgeball.build.cs** file, which is a C# file and not a C++ file, located inside your project's **Source** folder.
2. Open the file, and you'll find the **AddRange** function from the **PublicDependencyModuleNames** property being called. This is the function that tells the engine which modules this project intends to use. As a parameter, an array of strings is sent, with the names of all the intended

modules for the project. Given that we intend on using UMG, we'll need to add the UMG-related modules: **UMG**, **Slate**, and **SlateCore**:

```
PublicDependencyModuleNames  
.AddRange(new string[] {  
    "Core", "CoreUObject",  
    "Engine", "InputCore",  
    "HeadMountedDisplay",  
    "UMG", "Slate", "SlateCore"  
});
```

Now that we notified the engine that we'll be using the UMG modules, let's create our Widget C++ class:

3. Open the Unreal editor.
4. Right-click on the Content Browser and select **New C++ Class**.
5. Set the **Show All Classes** checkbox to **true**.
6. Search for the **UserWidget** class and choose that as the new class's parent class.
7. Name the new C++ class **RestartWidget**.

After the files have been opened in Visual Studio, start making modifications to our Widget C++ class as mentioned in the following steps:

8. The first thing we'll add to this class is a **public class UButton*** property called **RestartButton**, which represents the Button the player will press in order to restart the level. You will want it to be bound to a Button in the Blueprint class that inherits from this class, by using the **UPROPERTY** macro with the **BindWidget** meta tag. This will force that Widget Blueprint to have a **Button** called **RestartButton** that we can access in C++ through this property and then freely edit its properties, such as the size and position, in the Blueprint:

```
UPROPERTY(meta =
(BindWidget))

class UButton*
RestartButton;
```

Note

*Using the **BindWidget** meta tag will*

*cause a compilation error if the Widget Blueprint that inherits from this C++ class doesn't have an element with the same type and name. If you don't want this to happen, you will have to mark **UPROPERTY** as an optional **BindWidget** like so:*

```
so:UPROPERTY(meta = (BindWidget, OptionalWidget = true))
```

This will make it so that binding this property is optional and doesn't cause a compilation error when compiling the Widget Blueprint.

Next, we're going to add the function that will be called when the player clicks the **RestartButton**, which will restart the level. We will be doing this using the **GameplayStatics** object's **OpenLevel** function and then sending the name of the current level.

9. In the Widget class's header file, add a declaration for a **protected** function called **OnRestartClicked** that returns nothing and receives no parameters. This function must be marked as **UFUNCTION**:

```
protected:  
UFUNCTION()  
void OnRestartClicked();
```

10. In the class's source file, add an **include** for the **GameplayStatics** object:

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

11. Then, add an implementation for our **OnRestartClicked** function:

```
void  
URestartWidget::OnRestartClicked()  
{  
}
```

12. Inside its implementation, call the **GameplayStatics** object's **OpenLevel** function. This function receives as parameters a world context object, which will be the **this** pointer, and the name of the level, which we'll have to fetch using the **GameplayStatics** object's **GetCurrentLevelName** function.

This last function must also receive a world context object, which will also be the **this** pointer:

```
UGameplayStatics::OpenLevel  
(this,  
FName(*UGameplayStatics::Ge  
tCurrentLevelName(this)));
```

Note

*The call to the **GameplayStatics** object's **GetCurrentLevelName** function must be preceded with * because it returns an **FString**, UE4's string type, and must be dereferenced in order to be passed to the **FName** constructor.*

The next step will be binding this function in such a way that it is called when the player presses the **RestartButton**:

13. In order to do this, we'll have to override a function that belongs to the **UserWidget** class, called **NativeOnInitialized**. This function is called only once, similarly

to the Actor's **BeginPlay** function, which makes it appropriate to do our setup. Add a declaration for the **public NativeOnInitialized** function with both the **virtual** and **override** keyword in our Widget class's header file:

```
virtual void  
NativeOnInitialized()  
override;
```

14. Next, in the class's source file, add the implementation of this function. Inside it, call its **Super** function and add an **if** statement that checks whether our **RestartButton** is different than **nullptr**:

```
void  
URestartWidget::NativeOnIni  
tialized()  
{  
    Super::NativeOnInitialize  
d();  
  
    if (RestartButton !=  
nullptr)  
{
```

```
    }  
}
```

15. If the **if** statement is true, we'll want to bind our **OnRestartClicked** function to the button's **OnClicked** event. We can do this by accessing the button's **OnClicked** property and calling its **AddDynamic** function, sending as parameters the object we want to call that function on, the **this** pointer, and a pointer to the function to be called, the **OnRestartClicked** function:

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

16. Because we're accessing functions related to the **Button** class, we'll also have to include it:

```
#include  
"Components/Button.h"
```

Note

*A Button's **OnClicked** event will be called when the player presses and releases that button with the mouse. There are other events related to the button, including the **OnPressed** event (when the player presses the button), the **OnReleased** event (when the player releases the button), and the **OnHover** and **OnUnhover** events (when the player respectively starts and stops hovering the mouse over that button).*

*The **AddDynamic** function must receive as a parameter a pointer to a function marked with the **UFUNCTION** macro. If it doesn't, you will get an error when calling that function. This is why we marked the **OnRestartClicked** function with the **UFUNCTION** macro.*

After you've done these steps, compile your changes and open the editor.

17. Open the **TestWidget** Widget Blueprint that you created earlier. We'll want to associate this Widget Blueprint with the **RestartWidget** class we just created, so we need to reparent it.

18. From the Widget Blueprint's **File** tab, select the **Reparent Blueprint** option and choose the **RestartWidget** C++ class as its new parent class:

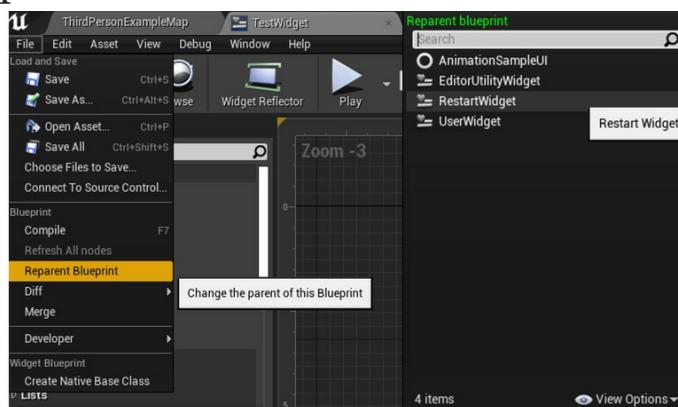


Figure 8.20: Reparenting the TestWidget's class to RestartWidget

You'll notice that the Widget Blueprint now has a compilation error related to the **BindWidget** meta tag that we created in the C++ class:

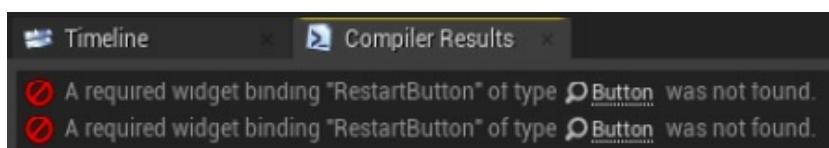


Figure 8.21: Compiler errors after setting the parent

class to the **RestartWidget** class

This is caused by the fact that the C++ class couldn't find any **Button** property called **RestartButton**.

In order to fix this, we'll need to rename our **Button** element inside the Widget Blueprint to **RestartButton**:

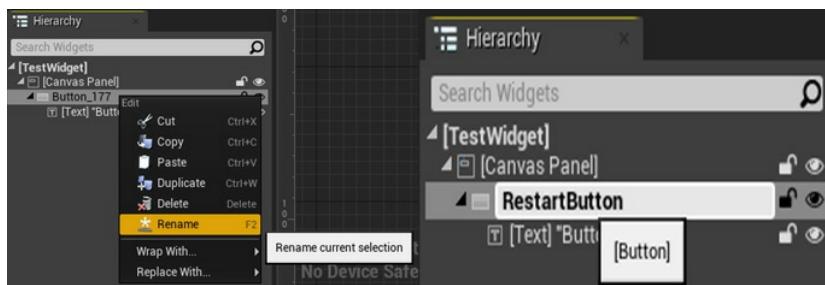


Figure 8.22: Renaming the Button element to **RestartButton**

After you've done this, close the Widget Blueprint and change its name from **TestWidget** to **BP_RestartWidget**, the same way you just did in the previous step.

That concludes the creation of our Widget class. You now know how to connect a Widget C++ class to a Widget Blueprint, a very important step toward handling Game UI in UE4.

The next thing we need to do is create our **Player Controller** C++ class, which will be responsible for instantiating our **RestartWidget** and adding it to the screen. We will be doing this in the following exercise.

Exercise 8.04: Creating the Logic for Adding the

RestartWidget to the Screen

In this exercise, we will create the logic responsible for adding our newly created **RestartWidget** to the screen. It will appear on screen when the player dies so that they have the option to restart the level.

In order to do this, we'll have to create a new **Player Controller** C++ class, which you can do by following these steps:

1. Open the Unreal editor.
2. *Right-click* on the **Content Browser** and select **New C++ Class**.
3. Search for the **Player Controller** class and choose that as the new class's parent class.
4. Name the new C++ class **DodgeballPlayerController**.
5. Open the class's files in Visual Studio.

When our player runs out of health points, the **DodgeballCharacter** class will access this **Player Controller** class and call a function that will add the **RestartWidget** to

the screen. Follow these steps ahead in order to make this happen.

In order to know the class of the Widget to add to the screen (which will be a Widget Blueprint and not a Widget C++ class), we'll need to use the **TSubclassOf** type.

6. In the class's header file, add a **public TSubclassOf<class URestartWidget>** property called **BP_RestartWidget**. Be sure to make it a **UPROPERTY** with the **EditDefaultsOnly** tag so that we can edit it in the Blueprint class:

```
public:  
    UPROPERTY(EditDefaultsOnly)  
    TSubclassOf<class  
        URestartWidget>  
    BP_RestartWidget;
```

In order to instantiate this Widget and add it to the screen, we'll need to save a reference to it.

7. Add a new **private** variable of type **class URestartWidget*** and call it

RestartWidget. Be sure to make it a **UPROPERTY** function with no tags:

```
private:  
  UPROPERTY()  
  class URestartWidget*  
  RestartWidget;
```

Note

*Although this property isn't supposed to be editable in a Blueprint class, we have to make this reference a **UPROPERTY**, otherwise the garbage collector will destroy the contents of this variable.*

The next thing we need is a function responsible for adding our Widget to the screen.

8. Add a declaration for a **public** function that returns nothing and receives no parameters called **ShowRestartWidget**:

```
void ShowRestartWidget();
```

9. Now, head to our class's source file. First, add an include to the

RestartWidget class:

```
#include "RestartWidget.h"
```

10. Then, add the implementation of our **ShowRestartWidget** function, where we'll start by checking whether our **BP_RestartWidget** variable is not a **nullptr**:

```
void  
ADodgeballPlayerController::  
ShowRestartWidget()  
{  
    if (BP_RestartWidget !=  
nullptr)  
    {  
    }  
}
```

11. If that variable is valid (different than **nullptr**), we want to pause the game using the **SetPause** function of **Player Controller**. This will make sure that the game stops until the player decides to do something (which in our case will be pressing the button that restarts the level):

```
SetPause(true);
```

The next thing we'll do is change the input mode. In UE4, there are three input modes: **Game Only**, **Game and UI**, and **UI Only**. If your **Input** Mode includes **Game**, that means that the player character and player controller will receive inputs through the **Input Actions**. If your **Input** Mode includes **UI**, that means that the Widgets that are on the screen will receive inputs from the player. When we show this Widget on the screen, we won't want the player character to receive any input.

12. Hence, update to the **UI Only Input** Mode. You can do this by calling the **Player Controller** **SetInputMode** function and passing the **FInputModeUIOnly** type as a parameter:

```
SetInputMode(FInputModeUIOnly());
```

After this, we want to show the mouse cursor, so that the player can see which button they are hovering the mouse on.

13. We will do this by setting the **Player Controller**'s **bShowMouseCursor** property to **true**:

```
bShowMouseCursor = true;
```

14. Now, we can actually instantiate our Widget using the **Player Controller**'s **CreateWidget** function, passing as a template parameter the C++ Widget class, which in our case is **RestartWidget**, and then as normal parameters the **Owning Player**, which is the **Player Controller** that owns this Widget and that we'll send using the **this** pointer, and the Widget class, which will be our **BP_RestartWidget** property:

```
RestartWidget =
CreateWidget<URestartWidget
>(this, BP_RestartWidget);
```

15. After we instantiate the Widget, we'll want to add it to the screen, using the Widget's **AddToViewport** function:

```
RestartWidget-
>AddToViewport();
```

16. That concludes our **ShowRestartWidget** function. However, we also need to create the function that will remove the **RestartWidget** from the screen. In the class's header file, add a declaration for a function just like the **ShowRestartWidget** function, but this time called **HideRestartWidget**:

```
void HideRestartWidget();
```

17. In the class's source file, add the implementation for the **HideRestartWidget** function:

```
void  
ADodgeballPlayerController:  
:HideRestartWidget()  
  
{  
  
}
```

18. The first thing we should do in this function is remove the Widget from the screen by calling its **RemoveFromParent** function, and destroy it using the **Destruct** function:

```
RestartWidget -  
>RemoveFromParent();  
  
RestartWidget->Destruct();
```

19. Then, we want to unpause the game using the **SetPause** function we used in the previous function:

```
SetPause(false);
```

20. And finally, set the **Input** Mode to **Game Only** and hide the mouse cursor the same way we did in the previous function (this time we pass the **FInputModeGameOnly** type instead):

```
SetInputModule(FInputModeGame  
Only());
```

```
bShowMouseCursor = false;
```

And that concludes the logic for our **Player Controller** C++ class. The next thing we should do is call the function that will add our Widget to the screen.

21. Go to the **DodgeballCharacter** class's source file and add the **include** keyword to our newly created

DodgeballPlayerController:

```
#include  
"DodgeballPlayerController.  
h"
```

22. Within the **DodgeballCharacter** class's implementation of the **OnDeath_Implementation** function, replace the call to the **QuitGame** function with the following:

1. Get the character's player controller using the **GetController** function.

You'll want to save the result in a variable of type **DodgeballPlayerController*** called

PlayerController.

Because the function will return a variable of type

Controller, you'll also need to cast it to our

PlayerController class:

```
ADodgeballPlayerController*  
PlayerController =  
Cast<ADodgeballPlaye
```

```
rController>  
(GetController());
```

2. Check whether the **PlayerController** variable is valid. If it is, call its **ShowRestartWidget** function:

```
if (PlayerController  
!= nullptr)  
{  
    PlayerController-  
>ShowRestartWidget()  
;  
}
```

After these modifications, the last thing left for us to do is to call the function that will hide our Widget from the screen. Open the **RestartWidget** class's source file and implement the following modifications.

23. Add an **include** to the **DodgeballPlayerController**, which contains the function that we

will be calling:

```
#include  
"DodgeballPlayerController.  
h"
```

24. Inside the **OnRestartClicked** function implementation, before the call to the **OpenLevel** function, we must fetch the Widget's **OwningPlayer**, which is of type **PlayerController**, using the **GetOwningPlayer** function, and cast it to the **DodgeballPlayerController** class:

```
ADodgeballPlayerController*  
PlayerController =  
Cast<ADodgeballPlayerController>(GetOwningPlayer());
```

25. Then, if the **PlayerController** variable is valid, we call its **HideRestartWidget** function:

```
if (PlayerController !=  
nullptr)  
{  
    PlayerController -
```

```
>HideRestartWidget();  
}  
  
After you've followed all these steps, close the editor, compile  
your changes and open the editor.
```

You have now concluded this exercise. We have added all the necessary logic to add our **RestartWidget** to the screen, and the only thing left for us to do is creating the Blueprint class of our newly created **DodgeballPlayerController**, which we'll be doing in the next exercise.

Exercise 8.05: Setting up the DodgeballPlayerController Blueprint Class

In this exercise, we will be creating the Blueprint class of our **DodgeballPlayerController** in order to specify which Widget we want to add to the screen, and tell UE4 to use this Blueprint class when the game starts.

In order to do that, follow these steps:

1. Go to the **ThirdPersonCPP -> Blueprints** directory in the Content Browser, right-click on it, and create a new Blueprint class.

2. Search for the **DodgeballPlayerController** class and select it as the parent class.
3. Rename this Blueprint class to **BP_DodgeballPlayerController**. After that, open this Blueprint asset.
4. Go to its **Class Defaults** tab and set the class's **BP_RestartWidget** property to the **BP_RestartWidget** Widget Blueprint we created.

Now, the only thing left for us to do is to make sure that this **Player Controller** Blueprint class is being used in the game.

In order to do this, we'll have to follow a few more steps.

5. Go to the **ThirdPersonCPP -> Blueprints** directory in the **Content Browser**, *right-click* on it and create a new Blueprint class. Search for the **DodgeballGameMode** class and select it as the parent class, then rename this **Blueprint** class to **BP_DodgeballGameMode**.

This class is responsible for telling the game which classes to use for each element of the game, such as which **Player Controller** class to use, among other things.

6. Open the asset, go to its **Class Defaults** tab, and set the class's **PlayerControllerClass** property to the **BP_DodgeballPlayerController** class we created:

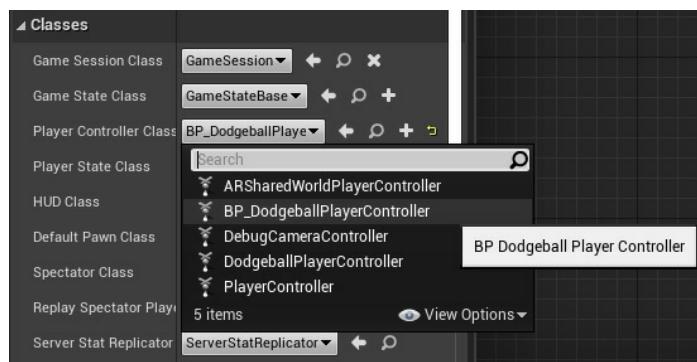


Figure 8.23: Setting the PlayerControllerClass property to BP_DodgeballPlayerController

7. Close the asset and select the **Blueprints** drop-down option inside the editor toolbar that is at the top of the **Level Viewport** window. From there, select **Game Mode** (which should currently be set to

DodgeballGameMode) -> Select **GameModeBase Class** -> **BP_DodgeballGameMode**. This will tell the editor to use this new **Game Mode** in all levels.

Now, play the game and let your character get hit by a Dodgeball **3** times. After the third time, you should see the game get paused and show **BP_RestartWidget**:

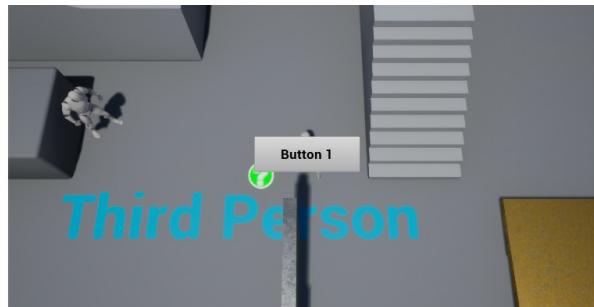


Figure 8.24: Our **BP_RestartWidget** being added to the screen after the player runs out of health points

And when you click **Button 1** using your mouse, you should see the level reset to its initial state:

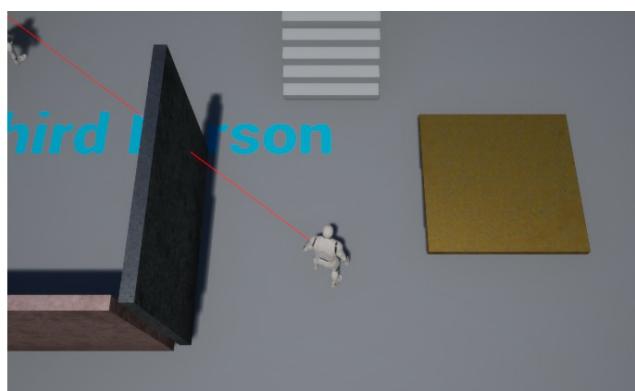


Figure 8.25: The level restarts after the player presses the button shown in the previous screenshot

And that concludes our exercise. You now know how to create Widgets and show them in your game. This is another crucial step in your journey toward becoming a skilled game developer.

Before we move on to the next exercise, let's take a look at Progress Bars in the next section.

Progress Bars

One of the ways that video games can represent character stats such as health, stamina, and so on is through **Progress Bars**, which are what we'll use to communicate to the player how much health their character has. Progress Bars are essentially a shape, usually rectangular, that can be filled and emptied in order to show the player how a specific stat is progressing. If you want to show the player that their character's health is only half its maximum value, you could do this by showing the Progress Bar as half full. This is exactly what we'll be doing in this section. This Progress Bar will be the only element in our Dodgeball game's HUD.

In order to create this **Health Bar**, we'll first need to create our HUD Widget. Open the editor, go to the **ThirdPersonCPP -> Blueprints** directory inside the Content Browser, and right-click and create a new **Widget Blueprint** class of the **User Interface** category. Name this new Widget Blueprint **BP_HUDWidget**. After that, open this new Widget Blueprint.

Progress Bars in UE4 are just another UI element, like **Buttons** and **Text** elements, which means we can drag it from the **Palette** tab into our **Designer** tab. Have a look at the following example:

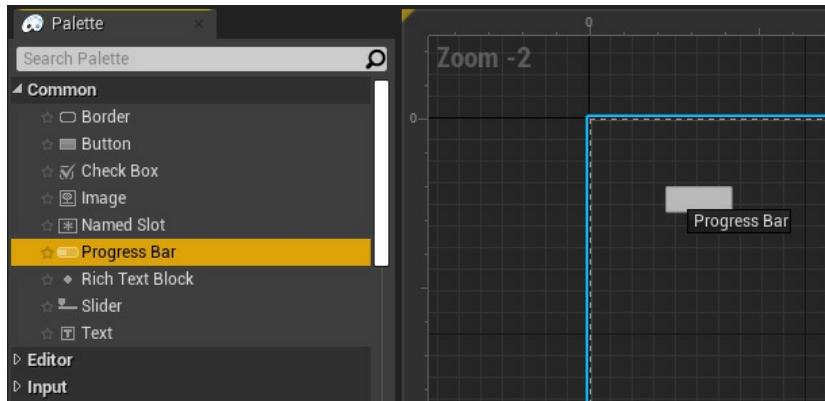


Figure 8.26: Dragging a Progress Bar element into the Designer window

At first, this Progress Bar might look similar to a button; however, it contains two specific properties that are important for a Progress Bar:

- **Percent** – allows you to specify this Progress Bar's progress, from **0** to **1**
- **Bar Fill Type** – allows you to specify how you want this Progress Bar to fill (from left to right, top to bottom, and so on):

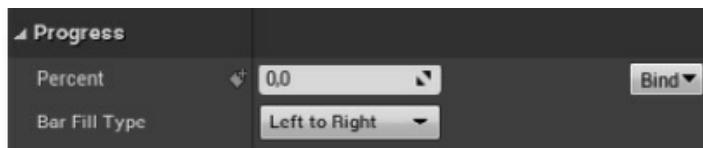


Figure 8.27: The Progress Bar's Percent and Bar Fill Type properties

If you set the **Percent** property to **0.5**, you should see the Progress Bar be updated accordingly to fill half of its length:

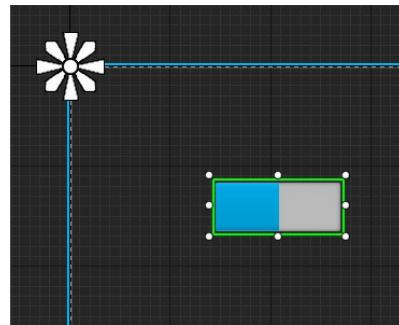


Figure 8.28: The Progress Bar filled halfway to the right

Before you continue, set the **Percent** property to **1**.

Let's now change the Progress Bar's color from blue (its default color) to red. In order to do this, go to the **Details** tab and, inside the **Appearance** category, set the **Fill Color and Opacity** property to red (**RGB(1, 0, 0)**):



Figure 8.29: The Progress Bar's Color being changed to red

After you've done this, your Progress Bar should now use red as its fill color.

To conclude our Progress Bar's setup, let's update its position, size, and Anchors. Follow these steps to achieve this:

1. In the Slot (**Canvas Panel Slot**) category, expand the **Anchors** property and set its properties to these values:
 1. **Minimum: 0 . 052** on the X

axis and **0.083** on the **Y** axis

2. **Maximum:** **0.208** on the **X** axis and **0.116** on the **Y** axis

2. Set the **Offset Left**, **Offset Top**, **Offset Right**, and **Offset Bottom** properties to **0**.

Your Progress Bar should now look like this:

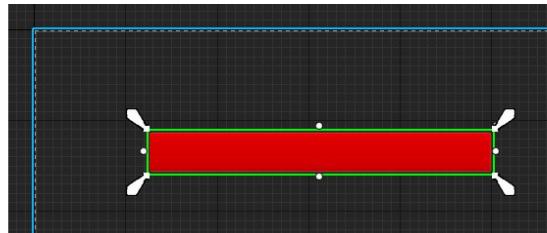


Figure 8.30: The Progress Bar after all the modifications in this section have been completed

And with that, we can conclude the topic of Progress Bars. Our next step is going to be adding all the logic necessary to use this Progress Bar as a health bar, by updating its **Percent** property alongside the player character's health. We'll do exactly this in the next exercise.

Exercise 8.06: Creating the Health Bar C++ Logic

In this exercise, we will be adding all the necessary C++ logic to update the Progress Bar inside our HUD as the player character's health changes.

In order to do this, follow these steps:

1. Open the editor and create a new C++ class that inherits from **UserWidget**, similar to how we did in *Exercise 8.03, Creating the RestartWidget C++ Class*, but this time call it **HUDWidget**. This will be the C++ class that will be used for our HUD Widget.
2. In the **HUDWidget** class's header file, add a new **public** property of type **class UProgressBar*** called **HealthBar**. This type is used to represent a Progress Bar, like the one we created in the previous section, in C++. Be sure to declare this property as a **UPROPERTY** function with the **BindWidget** tag:

```
UPROPERTY(meta =
(BindWidget))

class UProgressBar*
HealthBar;
```

3. Add a declaration for a **public** function called **UpdateHealthPercent**, which returns nothing and receives a **float**

HealthPercent property as a parameter. This function will be called in order to update the **Percent** property of our Progress Bar:

```
void  
UpdateHealthPercent(float  
HealthPercent);
```

4. In the **HUDWidget** class's source file, add the implementation for the **UpdateHealthPercent** function, which will call the **HealthBar** property's **SetPercent** function, passing the **HealthPercent** property as a parameter:

```
void  
UHUDWidget::UpdateHealthPer  
cent(float HealthPercent)  
{  
    HealthBar -  
    >SetPercent(HealthPercent);  
}
```

5. Because we'll be using the **ProgressBar** C++ class, we'll need to add an **include** to it at the top of the class's source file:

```
#include  
"Components/ProgressBar.h"
```

The next step will be adding all the necessary logic to our **Player Controller** responsible for adding the **HUDWidget** to the screen.

Implement the following steps in order to achieve this:

6. Inside the **DodgeballPlayerController** class's header file, add a **public** property of type **TSubclassOf<class UHUDWidget>** called **BP_HUDWidget**. Make sure to mark it as a **UPROPERTY** function with the **EditDefaultsOnly** tag.

This property will allow us to specify, in the **DodgeballPlayerController** Blueprint class, which Widget we want to use as our HUD:

```
UPROPERTY(EditDefaultsOnly)  
TSubclassOf<class UHUDWidget> BP_HUDWidget;
```

7. Add another property, this time **private**, of type **class UHUDWidget*** called **HUDWidget**.
Mark it as a **UPROPERTY**, but without any tags:

```
UPROPERTY()  
class UHUDWidget*  
HUDWidget;
```

8. Add a **protected** declaration for the **BeginPlay** function, and mark it as both **virtual** and **override**:

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

9. Add a declaration for a new **public** function, called **UpdateHealthPercent**, which returns nothing and receives a **float HealthPercent** as a parameter.

This function will be called by our player character class in order to update the Health Bar in our HUD:

```
void  
UpdateHealthPercent(float  
HealthPercent);
```

10. Now head over to the **DodgeballPlayerController** class's source file. Start by adding an **include** to our **HUDWidget** class:

```
#include "HUDWidget.h"
```

11. Then, add the implementation for the **BeginPlay** function, where we'll start by calling the **Super** object's **BeginPlay** function:

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

12. After that function call, check whether the **BP_HUDWidget** property is valid. If it is, call the **CreateWidget** function with the **UHUDWidget** template parameter and passing the **Owning Player**, **this**, and the Widget class, **BP_HUDWidget**, as parameters. Be sure to set the **HUDWidget** property to the return value of this function call:

```
if (BP_HUDWidget !=  
nullptr)  
{  
    HUDWidget =  
CreateWidget<UHUDWidget>  
(this, BP_HUDWidget);  
}
```

13. After setting the **HUDWidget** property, call its **AddToViewport** function:

```
HUDWidget->AddToViewport();
```

14. Lastly, add the implementation for the **UpdateHealthPercent** function, where we'll check if the **HUDWidget** property is valid and, if it is, call its **UpdateHealthPercent** function and pass the **HealthPercent** property as a parameter:

```
void  
ADodgeballPlayerController:  
:UpdateHealthPercent(float  
HealthPercent)  
{  
    if (HUDWidget != nullptr)  
    {
```

```
    HUDWidget -  
    >UpdateHealthPercent(Health  
    Percent);  
  
}  
  
}
```

Now that we've added the logic responsible for adding the HUD to the screen and allowing it to be updated, we'll need to make some modifications to other classes. Follow these steps in order to do so.

Currently, our **Health** interface, which we created in the previous chapter, only has the **OnDeath** event, which is called whenever an object runs out of health points. In order to update our Health Bar every time the player takes damage, we'll need to allow our **HealthInterface** class to notify an object whenever that happens.

15. Open the **HealthInterface** class's header file and add a declaration similar to the one we did for the **OnDeath** event in *Exercise 7.04, Creating the HealthInterface Class,*

but this time for the **OnTakeDamage** event. This event will be called whenever an object takes damage:

```
UFUNCTION(BlueprintNativeEvent, Category = Health)

void OnTakeDamage();

virtual void
OnTakeDamage_Implementation
() = 0;
```

16. Now that we have added this event to our **Interface** class, let's add the logic that calls that event: open the **HealthComponent** class's source file and, inside its implementation of the **LoseHealth** function, after subtracting the **Amount** property from the **Health** property, check whether the **Owner** implements the **Health** interface and, if it does, call its **OnTakeDamage** event. Do this the same way we already did later in that same function for our **OnDeath** event, but this time simply change the name of the event to **OnTakeDamage**:

```
if (GetOwner() ->Implements<UHealthInterface>())
```

```
e>()
{
    IHealthInterface::Execute
    _OnTakeDamage(GetOwner());
}
```

Because our Health Bar will require the player character's health points as a percentage, we need to do the following:

17. Add a **public** function to our **HealthComponent** that returns just that: in the **HealthComponent** class's header file, add a declaration for a **FORCEINLINE** function that returns a **float**. This function should be called **GetHealthPercent** and be a **const** function. Its implementation will simply consist of returning the **Health** property divided by **100**, which we will assume is the maximum amount of health points an object can have in our game:

```
FORCEINLINE float
GetHealthPercent() const {
    return Health / 100.f; }
```

18. Now go to the **DodgeballCharacter** class's header file and add a declaration for a **public virtual** function called **OnTakeDamage_Implementation**, which returns nothing and receives no parameters. Mark it as **virtual** and **override**:

```
virtual void  
OnTakeDamage_Implementation  
( ) override;
```

19. In the **DodgeballCharacter** class's source file, add an implementation for the **OnTakeDamage_Implementation** function we just declared. Copy the content of the **OnDeath_Implementation** function to this new function's implementation, but do this change: instead of calling the **ShowRestartWidget** function of **PlayerController**, call its **UpdateHealthPercent** function, and pass the return value of the **HealthComponent** property's **GetHealthPercent** function as a parameter:

```

void
ADodgeballCharacter::OnTake
Damage_Implementation()
{
    ADodgeballPlayerController*
    PlayerController =
    Cast<ADodgeballPlayerController>(GetController());
    if (PlayerController != nullptr)
    {
        PlayerController-
        >UpdateHealthPercent(Health
        Component-
        >GetHealthPercent());
    }
}

```

This concludes this exercise's code setup. After you've done these changes, compile your code, open the editor and do the following:

20. Open the **BP_HUDWidget** Blueprint and reparent it to the **HUDWidget** class, the same way you

did in *Exercise 8.03, Creating the RestartWidget C++ Class*.

21. This should cause a compilation error, which you'll be able to fix by renaming our Progress Bar element to **HealthBar**.
22. Close this Widget Blueprint, open the **BP_DodgeballPlayerController** Blueprint class and set its **BP_HUDWidget** property to the **BP_HUDWidget** Widget Blueprint:



Figure 8.31: Setting the BP_HUDWidget property to BP_HUDWidget

After you've done these changes, play the level. You should notice the **Health Bar** at the top left of the screen:

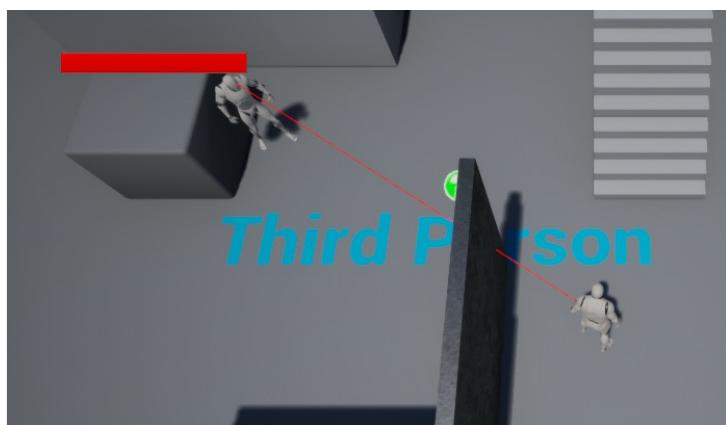


Figure 8.32: The Progress Bar shown at the top left of the screen

When the player character gets hit by a Dodgeball, you should notice the **Health Bar** being emptied:

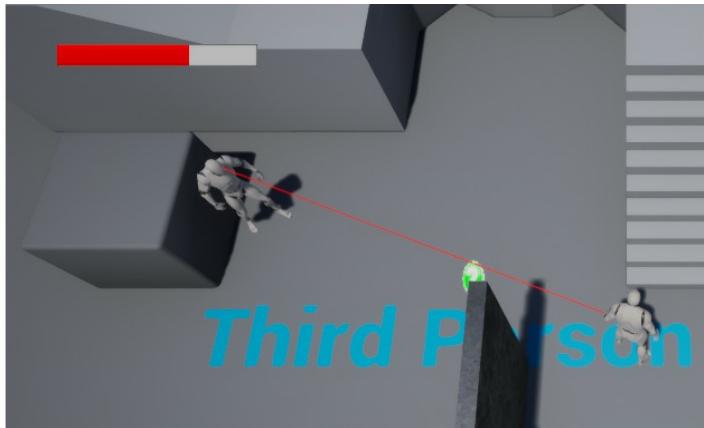


Figure 8.33: The Progress Bar being emptied as the Player Character loses health points

And with that, we conclude this exercise, in which you've learned all the necessary steps in order to add a HUD to the screen and to update it during the game.

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 to **Restart** so that the players know what should happen when they click that button.

The following steps will help you complete this activity:

1. Open the **BP_RestartWidget** Blueprint.

2. Drag a new **Text** element into the existing **Canvas Panel** element.
3. Modify the **Text** element's properties:
 1. 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.
 2. Set the **Offset Left**, **Offset Top**, **Offset Right**, and **Offset Bottom** properties to **0**.
 3. Set the **Text** property to **GAME OVER**.
 4. Set the **Color** and **Opacity** property to red: **RGBA(1.0, 0.082, 0.082, 1.0)**.
 5. Expand the **Font** property and set its **Size** to **100**.
 6. Set the **Justification** property to **Align Text Center**.

4. Select the other **Text** element inside the **RestartButton** property, and change its **Text** property to **Restart**.
 5. Duplicate the **RestartButton** property and change the copy's name to **ExitButton**.
 6. Change the **Text** property of the **Text** element inside the **ExitButton** property to **Exit**.
 7. Expand the **Anchor** property of the **ExitButton** property 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.
 8. Set the **ExitButton** properties of **Offset Left**, **Offset Top**, **Offset Right**, and **Offset Bottom** to **0**.
- After you've done these changes, we'll need to add the logic responsible for handling the **ExitButton** property click, which will exit the game:
9. 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**.

10. Duplicate the existing **RestartButton** property, but call it **ExitButton** instead.
11. Inside the **RestartWidget** class's source file, add an implementation for the **OnExitClicked** function. Copy the contents of the **OnBeginOverlap** function from inside the **VictoryBox** class's source file into the **OnExitClicked** function, but remove the cast being done to the **DodgeballCharacter** class.
12. In the **NativeOnInitialized** function implementation, bind the **OnExitClicked** function we created to the **OnClicked** event of the **ExitButton** property, the same way we did for the **RestartButton**

property in *Exercise 8.03, Creating the RestartWidget C++ Class*.

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 8.34: 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.

And that concludes our activity. You've consolidated the basics of using a **Widget Blueprint** and changing its element's properties and are now ready to start making your own menus.

Note

The solution to this activity can be found at:

<https://packt.live/338jEBx>.

Summary

With this chapter concluded, you have now learned how to make a game UI in UE4, understanding things such as menus and HUDs. You've seen how to manipulate a Widget Blueprint's UI elements, including **Buttons**, **Text** elements, and **Progress Bars**; work with Anchors effectively, which is instrumental in allowing your game UI to adapt elegantly to multiple screens; listen to mouse events in C++, such as the **OnClick** event, and use that to create your own game logic; as well as how to add the Widgets you create to the screen, either at specific events or have them present at all times.

In the next chapter, we'll be taking a look at polishing our Dodgeball game by adding audiovisual elements such as sound and particle effects, as well as making a new level.

9. Audio-Visual Elements

Overview

In this chapter, we will finish the dodgeball-based game that we have been working on in the past four chapters. We will conclude this chapter by adding sound effects, particle effects, and by creating another level, this time with an actual path the player must follow to finish it. By the end of this chapter, you will be able to add 2D and 3D sound effects, as well as particle effects, to your UE4 projects.

Introduction

In the previous chapter, we learned about game UI and how to create and add a user interface (also known as a widget) to the screen.

In this chapter, we will learn how to add audio and particle effects to our game. Both of these aspects will increase the quality of our game and produce a much more immersive experience for the player.

Sound in video games can come in the form of either sound effects (also known as SFX) or music. Sound effects make the world around you more believable and alive, while the music helps set the tone for your game. Both these aspects are very important to your game.

In competitive games such as *Counter-Strike (CS: GO)*, sound is also extremely important because players need to hear the

sounds around them, such as gunshots and footsteps, and which direction they came from, to gather as much information about their surroundings as possible.

Particle effects are important for the same reason that sound effects are important: they make your game world more believable and immersive.

Let's start this chapter by learning how audio works in UE4.

Audio in UE4

One of the essential components of any game is sound. Sounds make your game more believable and immersive, which will provide a much better experience for your player. Video games usually have two types of sounds:

- 2D sounds
- 3D sounds

2D sounds don't have any consideration for the listener's distance and direction, while 3D sounds can be higher or lower in volume and pan to the right or left, depending on the player's location. 2D sounds are usually used for music, while 3D sounds are usually used for sound effects. The main sound file types are **.wav** and **.mp3**.

Here are some of the assets and classes related to audio in UE4:

- **Sound Base:** Represents an asset that contains audio. This class is mainly used in C++ and Blueprint to reference

an audio file that can be played.

- **Sound Wave:** Represents an audio file that has been imported into UE4.
Inherits from **Sound Base**.
- **Sound Cue:** An audio asset that can contain logic related to things such as attenuation (how the volume changes as the listener's distance varies), looping, sound mixing, and other audio-related functionality. It inherits from **Sound Base**.
- **Sound Class:** An asset that allows you to separate your audio files into groups and manage some of their settings, such as volume and pitch. An example of this would be grouping all your sounds related to sound effects in the **SFX Sound Class**, all your character dialogue in the **Dialogue Sound Class**, and so on.
- **Sound Attenuation:** An asset that allows you to specify how a 3D sound will behave; for example, at which distance it will start to lower the volume, at which distance it will become inaudible (can't be heard), if

its volume will change linearly or exponentially as the distance increases, and so on.

- **Audio Component:** An actor component that allows you to manage the playback of audio files and their properties. Useful for setting up continuous playback of sounds, such as background music.

In UE4, we can import existing sounds the same way we would any other asset: either by dragging a file from the Windows File Explorer into the **Content Browser** or by clicking the **Import** button in the **Content Browser**. We'll do this in the next exercise.

Exercise 9.01: Importing an Audio File

In this exercise, you will import an existing sound file from your computer into UE4. This audio file will be played when the dodgeball bounces off a surface.

Note

*If you don't have an audio file (either an **.mp3** or **.wav** file) available to complete this exercise, you can download the **.mp3** or **.wav** file available at this link:
<https://www.freesoundeffects.com/free-track/bounce-1-468901/>.*

*Save this file as **BOUNCE.wav**.*

Once you have an audio file, follow these steps:

1. Open the editor.
2. Go to the **Content** folder inside the **Content Browser** interface and create a new folder called **Audio**:

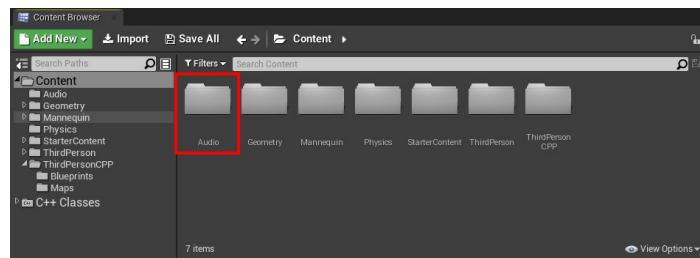


Figure 9.1: The Audio folder in the Content Browser

3. Go to the **Audio** folder you just created.
4. Import your audio file into this folder. You can do this by *dragging* the audio file from **Windows File Explorer** into **Content Browser**.
5. After you've done this, a new asset should appear with the name of your audio file, which you can listen to when clicking on it:

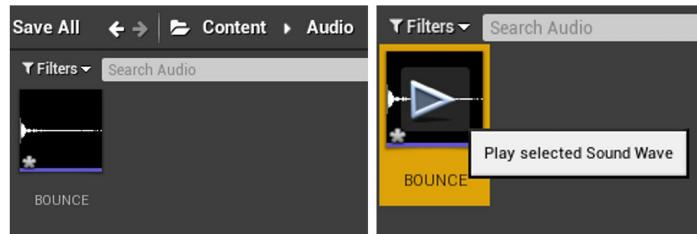


Figure 9.2: The imported audio file

6. Open this asset. You should see many properties available for editing. However, we'll be focusing solely on some of the properties inside the **Sound** category:

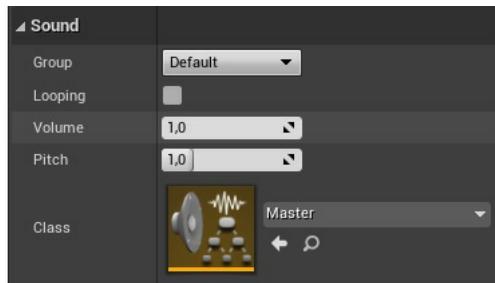


Figure 9.3: The Sound asset's settings

The following properties are available in the **Sound** category:

1. **Looping:** Whether this sound will loop while being played.
2. **Volume:** The volume of this sound.

3. **Pitch**: The pitch of this sound. The higher the pitch, the higher the frequency, and the higher in tone this sound will be.

4. **Class**: The **Sound Class** of this sound.

The only property we'll be changing is the **Class** property. We could use one of the existing **Sound** classes that comes with UE4, but let's create our own **Sound Class** for the dodgeball in order to create a new group of sounds for our game.

7. Go to the **Audio** folder inside the **Content Browser** interface.
8. *Right-click*, go to the **Sounds** category (the penultimate category), then the **Classes** category, and select **Sound Class**. This will create a new **Sound Class** asset. Rename this asset **Dodgeball**.

9. Open your imported sound asset and set its **Class** property to **Dodgeball**:

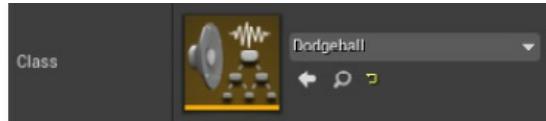


Figure 9.4: Changing the Class property to the Dodgeball Sound Class

Now that this imported sound asset belongs to a specific class, you can group other sound effects related to the dodgeball in the same **Sound Class** and edit their properties through that **Sound Class**, which includes **Volume**, **Pitch**, and many others.

And with that, we can conclude our exercise. You have learned how to import sounds into your project and how to change their basic properties. Now, let's move on to the next exercise, where we'll be playing a sound whenever a dodgeball bounces off a surface in our game.

Exercise 9.02: Playing a Sound When the Dodgeball Bounces off a Surface

In this exercise, we will add the necessary functionality to our **DodgeballProjectile** class so that a sound will play when the dodgeball bounces off a surface.

To do this, follow these steps:

1. Close the editor and open Visual Studio.
2. In the header file for the **DodgeballProjectile** class, add a protected **class USoundBase*** property called **BounceSound**. This property should be a **UPROPERTY** and have the **EditDefaultsOnly** tag so that it can be edited in the Blueprint:

```
// The sound the dodgeball  
will make when it bounces  
off of a surface  
  
UPROPERTY(EditAnywhere,  
Category = Sound)  
  
class USoundBase*  
BounceSound;
```

3. After you've done this, go to the **DodgeballProjectile** class's source file and add an include for the **GameplayStatics** object:

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

4. Then, at the beginning of the class's implementation of the **OnHit** function,

before the cast to the **DodgeballCharacter** class, check whether our **BounceSound** is a valid property (different than **nullptr**) and whether the magnitude of the **NormalImpulse** property is greater than **600** units (we can access the magnitude by calling its **Size** function).

As we saw in *Chapter 8, User Interfaces*, the **NormalImpulse** property indicates the direction and magnitude of the force that will change the dodgeball's trajectory after it has been hit. The reason why we want to check if its magnitude is greater than a certain amount is that when the dodgeball starts losing momentum and bounces off of the floor several times per second, we don't want to play **BounceSound** several times per second; otherwise, it will generate a lot of noise. So, we will check whether the impulse that the dodgeball is suffering is greater than that amount to make sure this doesn't happen. If both these things are true, we'll call the

- GameplayStatics** object's **PlaySoundAtLocation**. This function is responsible for playing 3D sounds. It receives five parameters:
1. A world context object, which we'll pass as the **this** pointer.
 2. A **SoundBase** property, which will be our **HitSound** property.
 3. The origin of the sound, which we'll pass using the **GetActorLocation** function.
 4. **VolumeMultiplier**, which we'll pass with a value of **1**. This value indicates how much higher or lower the volume of this sound will be when it's played. For instance, a value of **2** means it will have the volume twice as high.
 5. **PitchMultiplier**, which indicates how much higher or lower the pitch of this sound

will be when it's played. We'll be passing this value by using the **FMath** object's **RandRange** function, which receives two numbers as parameters and returns a random number between those two. To randomly generate a number between **0.7** and **1.3**, we'll be calling this function with these values as parameters.

Have a look at the following code snippet:

```
if (BounceSound !=  
nullptr &&  
NormalImpulse.Size()  
> 600.0f)  
{  
    UGameplayStatics::  
    PlaySoundAtLocation(  
        this, BounceSound,  
        GetActorLocation(),  
        1.0f,  
        FMath::RandRange(0.7  
        f, 1.3f));
```

}

Note

*The function responsible for playing 2D sounds is also available from the **GameplayStatics** object, and it's called **PlaySound2D**. This function will receive the same parameters as the **PlaySoundAtLocation** function, except for the third parameter, which is the origin of the sound.*

5. Compile these changes and then open Unreal Editor.
6. Open the **BP_DodgeballProjectile** Blueprint, go to its **Class Defaults** tab, and set the **BounceSound** property to the Sound asset you imported:

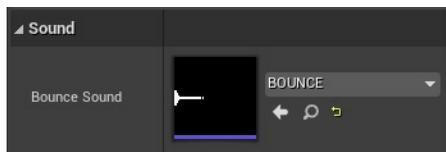


Figure 9.5: Setting the BounceSound property to our imported sound

7. Play the level again and enter the enemy character's line of sight. You should notice a sound playing with different pitch values every time the dodgeball thrown by the enemy character hits a wall or the floor (not the player character):

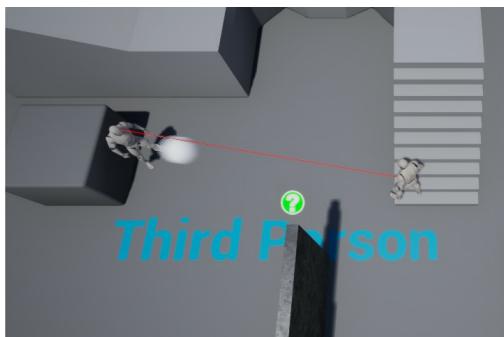


Figure 9.6: The player character causing the enemy character to throw dodgeballs

If this happens, congratulations – you've successfully played a sound using UE4! If you can't hear the sound playing, make sure that it is audible (it has a level of volume that you can hear).

However, another thing you'll probably notice is that the sound is always played at the same volume, regardless of the distance that the character is from the dodgeball that is bouncing: the sound isn't playing in 3D but rather is being played in 2D. To play a sound in 3D using UE4, we'll have to learn about Sound Attenuation assets.

Sound Attenuation

For a sound to be played in 3D inside UE4, you'll have to create a Sound Attenuation asset, as we mentioned in the first section of this chapter. A Sound Attenuation asset will let you specify how you want a specific sound to change volume as its distance from the listener increases. Have a look at the following example.

Open Unreal Editor, go to the **Audio** folder inside the **Content Browser** interface, *right-click*, go to the **Sounds** category, and select **Sound Attenuation**. Name this new asset **BounceAttenuation**:

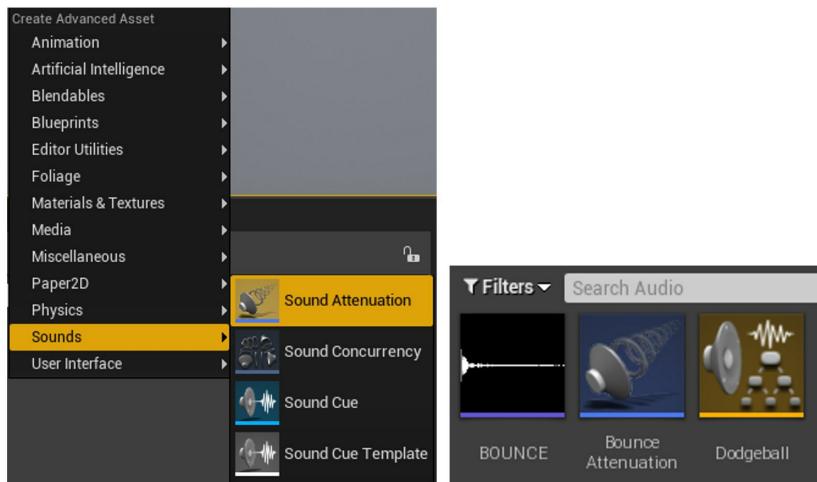


Figure 9.7: Creating the Sound Attenuation asset

Open this **BounceAttenuation** asset.

Sound Attenuation assets have many settings; however, we'll want to focus mainly on a couple of settings from the **Attenuation Distance** section:

- **Inner Radius:** This **float** property allows us to specify at what distance the sound will start lowering in

volume. If the sound is played at a distance less than this value, the volume won't be affected. Set this property to **200** units.

- **Falloff Distance:** This float property allows us to specify at what distance we want the sound to be inaudible. If the sound is played at a distance greater than this value, we won't hear it. The volume of the sound will vary according to its distance to the listener and whether it's closer to **Inner Radius** or **Falloff Distance**. Set this property to **1500** units:

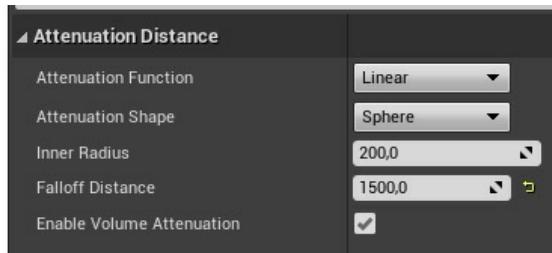


Figure 9.8: The Sound Attenuation asset settings

Think of this as two circles around the player, with the smaller circle being the Inner Circle (with a radius value of **Inner Radius**) and the bigger circle being the Falloff Circle (with a radius value of **Falloff Distance**). If a sound originates from inside the Inner Circle, it is played at full volume, while a sound that originates from outside the Falloff Circle is not played at all.

Note

You can find more information on Sound Attenuation assets here:

<https://docs.unrealengine.com/en-US/Engine/Audio/DistanceModelAttenuation>.

Now that you know about Sound Attenuation assets, let's move on to the next exercise, where we'll turn the sound that plays when the dodgeball bounces off the ground into a 3D sound.

Exercise 9.03: Turning the Bounce Sound into a 3D Sound

In this exercise, we'll be turning the sound that plays when a dodgeball bounces off the ground, which we added in the previous exercise, into a 3D sound. This means that when the dodgeball bounces off a surface, the sound it plays will vary in volume, depending on its distance to the player. We're doing this so that when the dodgeball is far away, the sound volume will be low, and when it's close, its volume will be high.

To use the **BounceAttenuation** asset we created in the previous section, follow these steps:

1. Go to the header file for **DodgeballProjectile** and add a **protected class** **USoundAttenuation*** property

called **BounceSoundAttenuation**.

This property should be a **UPROPERTY**, and have the **EditDefaultsOnly** tag so that it can be edited in the Blueprint:

```
// The sound attenuation of  
the previous sound  
  
UPROPERTY(EditAnywhere,  
Category = Sound)  
  
class USoundAttenuation*  
BounceSoundAttenuation;
```

2. Go to the **DodgeballProjectile** class' implementation of the **OnHit** function in its source file, and add the following parameters to the call to the **PlaySoundAtLocation** function:

1. **StartTime**, which we'll pass with a value of **0**. This value indicates the time that the sound will start playing. If the sound lasts 2 seconds, we can have this sound start at its 1-second mark by passing a value of **1**. We pass a value of **0** to have the sound play from the start.

2. **SoundAttenuation**, to which we'll pass our **BounceSoundAttenuation** property:

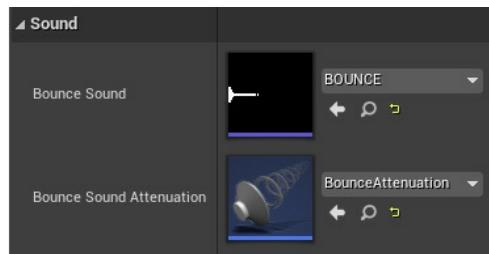
```
UGameplayStatics::PlaySoundAtLocation(this, BounceSound, GetActorLocation(), 1.0f, 1.0f, 0.0f, BounceSoundAttenuation);
```

Note

*Although we only want to pass the additional **SoundAttenuation** parameter, we have to pass all the other parameters that come before it as well.*

3. Compile these changes and then open the editor.
4. Open the **BP_DodgeballProjectile** Blueprint, go to its **Class Defaults** tab, and set the

BounceSoundAttenuation property to our **BounceAttenuation** asset:



**Figure 9.9: Setting the
BoundSoundAttenuation
property to the
BounceAttenuation asset**

5. Play the level again and enter the enemy character's line of sight. You should now notice that the sound that plays every time the dodgeball thrown by the enemy character hits a wall or the floor will be played at different volumes, depending on its distance, and that you won't hear it if the dodgeball is far away:

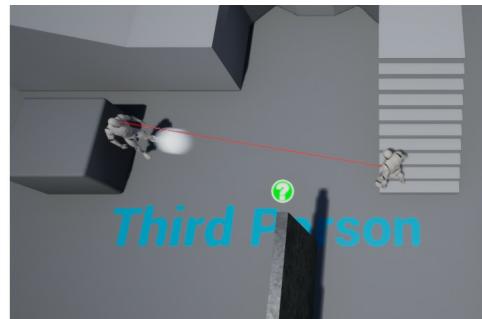


Figure 9.10: The player character causing the enemy character to throw dodgeballs

With that, we can conclude this exercise. You now know how to play 3D sounds using UE4. We'll add background music to our game in the next exercise.

Exercise 9.04: Adding Background Music to Our Game

In this exercise, we will add background music to our game. We will do this by creating a new Actor with an Audio component, which, as we mentioned earlier, is appropriate for playing background music. To achieve this, follow these steps:

1. Download the audio file located at <https://packt.live/3pg21sQ> and import it into the **Audio** folder of the **Content Browser** interface, just like we did in *Exercise 9.01, Importing an Audio File*.
2. Right-click inside the **Content Browser** interface and create a new C++ class with the **Actor** class as its parent class. Name this new class **MusicManager**.
3. When the files for this class are generated and Visual Studio has opened automatically, close the editor.

4. In the **MusicManager** class's header file, add a new **protected** property of the **class UAudioComponent*** type called **AudioComponent**. Make this a **UPROPERTY** and add the **VisibleAnywhere** and **BlueprintReadOnly** tags:

```
UPROPERTY(VisibleAnywhere,  
BlueprintReadOnly)  
  
class UAudioComponent*  
AudioComponent;
```

5. In the **MusicManager** class's source file, add an **include** for the **AudioComponent** class:

```
#include  
"Components/AudioComponent.  
h"
```

6. In the constructor for this class, change the **bCanEverTick** property to **false**:

```
PrimaryActorTick.bCanEverTi  
ck = false;
```

7. After this line, add a new one that creates the **AudioComponent** class by

calling the **CreateDefaultSubobject** function and passing the **UAudioComponent** class as a template parameter and "**Music Component**" as a normal parameter:

```
AudioComponent =  
CreateDefaultSubobject<UAud  
ioComponent>(TEXT("Music  
Component"));
```

8. After making these changes, compile your code and open the editor.
9. Go to the **ThirdPersonCPP -> Blueprints** folder in the **Content Browser** interface and create a new Blueprint class that inherits from the **MusicManager** class. Name it **BP_MusicManager**.
10. Open this asset, select its **Audio** component, and set that component's **Sound** property to your imported sound:

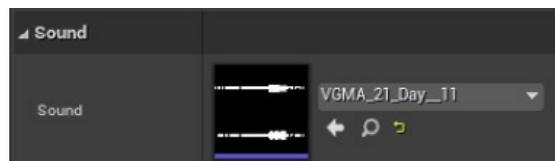


Figure 9.11: The Sound property being updated

11. Drag an instance of the **BP_MusicManager** class into the level.
12. Play the level. You should notice the music start playing when the game starts and it should also loop automatically when it reaches the end (this is done thanks to the Audio component).

Note

*Audio components will automatically loop whatever sound they're playing, so there's no need to change that Sound asset's **Looping** property.*

After completing all these steps, we've completed this exercise. You now know how to add simple background music to your game.

Now, let's jump into the next topic, which is Particle Systems.

Particle Systems

Let's talk about another very important element of many

video games: Particle Systems.

In video game terms, a particle is essentially a position in a 3D space that can be represented with an image. A Particle System is a collection of many particles, potentially with different images, shapes, colors, and sizes. In the following image, you will find an example of two Particle Systems made in UE4:



Figure 9.12: Two different Particle Systems in UE4

The Particle System on the left is supposed to be electrical sparks that could come from a cable that has been sliced and is now in short-circuit, while the one on the right is supposed to be a fire. Although the particle system on the left is relatively simple, you can tell that the one on the right has more than one type of particle inside it, which can be combined in the same system.

Note

*UE4 has two different tools for creating Particle Systems: **Cascade** and **Niagara**. Cascade is the tool that has been present since the beginning of UE4, while Niagara is a system that is more recent and sophisticated and has only been production-ready since May 2020, as of Unreal Engine version 4.25.*

Creating Particle Systems in UE4 is outside the scope of this book, but it is recommended that you use Niagara over Cascade, given that it is a more recent addition to the engine.

In this chapter, we will only be using Particle Systems that are already included in UE4, but if you want to create your own, these links will give you more information about both Cascade and Niagara:

Cascade: <https://docs.unrealengine.com/en-US/Engine/Rendering/ParticleSystems/Cascade>

https://www.youtube.com/playlist?list=PLZlv_No_O1gYDLyB3LVfjYIcbBe8NqR8t

Niagara: <https://docs.unrealengine.com/en-US/Engine/Niagara/EmitterEditorReference/index.html>

<https://docs.unrealengine.com/en-US/Engine/Niagara/QuickStart>

We'll learn how to add Particle Systems to our game in the next exercise. In this chapter, we will be simply using existing Particle Systems that were already made by the UE4 team.

Exercise 9.05: Spawning a Particle System When the Dodgeball Hits the Player

In this exercise, we will get to know how to spawn a Particle System in UE4. In this case, we will be spawning an **explosion** Particle System when a dodgeball thrown by the enemy hits the player.

To achieve this, follow these steps:

1. Close the editor and open Visual Studio.
2. In the **DodgeballProjectile** class's header file, add a protected **class UParticleSystem*** property called **HitParticles**.

The **UParticleSystem** type is the designation for a Particle System in UE4. Be sure to make this a **UPROPERTY** and give it the **EditDefaultsOnly** tag so that it can be edited in the Blueprint class:

```
// The particle system the
// dodgeball will spawn when
// it hits the player
UPROPERTY(EditAnywhere,
Category = Particles)
class UParticleSystem*
HitParticles;
```

3. In the **DodgeballProjectile** class's source file, inside its implementation of the **OnHit** function. Before the call to the **Destroy** function, check

whether our **HitParticles** property is valid. If it is, call the **GameplayStatics** object's **SpawnEmitterAtLocation** function.

This function will spawn an actor that will play the Particle System we pass as a parameter. It receives the following parameters:

1. A **World** object, which we'll pass using the **GetWorld** function.
2. A **UParticleSystem*** property, which will be our **HitParticles** property.
3. The **FTransform** of the actor that will play the Particle System, which we'll pass using the **GetActorTransform** function:

```
if (HitParticles !=  
nullptr)  
{
```

```
UGameplayStatics::
```

```
SpawnEmitterAtLocati  
on(GetWorld(),  
HitParticles,  
GetActorTransform())  
;  
}
```

Note

*Although we won't be using it in this project, there is another function related to spawning Particle Systems available from the **GameplayStatics** object, which is the **SpawnEmitterAttached** function. This function will spawn a Particle System and attach it to an actor, which might be useful if you want to, for instance, make a moving object light on fire so that the Particle System will always remain attached to that object.*

4. Compile these changes and then open

the editor.

5. Open the **BP_DodgeballProjectile** Blueprint, go to its **Class Defaults** tab, and set the **HitParticles** property to the **P_Exlosion** Particle System asset:



Figure 9.13: Setting the HitParticles property to P_Exlosion

6. Now, play the level and let your player character get hit by a dodgeball. You should now see the explosion Particle System being played:



Figure 9.14: The explosion particle system being played when the dodgeball hits the player

And that concludes this exercise. You now know how to play Particle Systems in UE4. Particle Systems will add visual flair to your game and make it more visually appealing.

In the next activity, we'll be consolidating our knowledge of playing audio in UE4 by playing a sound when the dodgeball hits the player.

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. In a video game, it's very important to transmit to the player's crucial information in many ways, so in addition to changing the player character's Health Bar, we'll also be playing a sound when the player gets hit so that the player knows that the character is taking damage.

To do this, follow these steps:

1. Import a sound file that will be played when the player character gets hit into the **Audio** folder inside the **Content Browser** interface.

Note

*If you don't have a sound file, you can use the one available at
<https://www.freesoundeffects.com/free-track/punch-426855/>.*

2. 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**.
3. 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.
4. Compile your changes and open the editor.
5. Open the **BP_DodgeballProjectile** Blueprint and set its **DamageSound** property to the sound file you imported

at the start of this activity.

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:



Figure 9.15: A sound should play when the player character gets hit

And with those steps complete, you have finished this activity and consolidated the use of playing both 2D and 3D sounds in UE4.

Note

The solution to this activity can be found at:

<https://packt.live/338jEBx>.

Now, let's wrap up this chapter by learning a bit about the concept of Level Design.

Level Design

Since *Chapter 5, Line Traces*, related to our dodgeball game, we've added quite a few game mechanics and gameplay opportunities, as well as some audio-visual elements, all of

which were handled in this chapter. Now that we have all these game elements, we must bring them together into a level that can be played from start to finish by the player. To do that, let's learn a bit about Level Design and level blockouts.

Level Design is a specific Game Design discipline that focuses on building levels in a game. The goal of a Level Designer is to make a level that is fun to play, introduces new gameplay concepts to the player by using the game mechanics built for that game, contains good pacing (a good balance of action-packed and relaxed gameplay sequences), and much more.

To test the structure of a level, Level Designers will first build what is called a **level blockout**. This is a very simple and boiled down version of the level that uses most of the elements that the final level will contain, but it is made using only simple shapes and geometry. The reason for this is for it to be easier and less time-consuming to modify the level in case parts of it need to be changed:

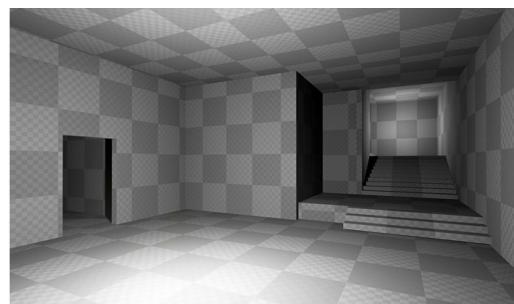


Figure 9.16: An example of a level blockout made in UE4 using BSP Brushes

Note

It should be noted that Level Design is its own specific game development skill and is worthy of its own book, of which there are quite a few, but diving into this topic is outside the scope of this book.

In the next exercise, we will be building a simple level

blockout using the mechanics we built in the last few chapters.

Exercise 9.06: Building a Level Blockout

In this exercise, we will be creating a new level blockout that will contain some structure, where the player will start in a certain place in the level and have to go through a series of obstacles to reach the end of the level. We will be using all the mechanics and objects that we built in the last few chapters to make a level that the player will be able to complete.

Although in this exercise we will be providing you with a solution, you are encouraged to let your creativity loose and come up with your solution, given that there is no right or wrong answer in this case.

To start this exercise, follow these steps:

1. Open the editor.
2. Go to the **ThirdPersonCPP -> Maps** folder in your **Content Browser**, duplicate the **ThirdPersonExampleMap** asset, and name it **Level1**. You can do this by either selecting the asset and pressing *Ctrl + W* or by right-clicking on the asset and selecting **Duplicate** (the third option).

3. Open the newly created **Level1** map.
4. Delete all the objects that have a mesh inside the map, except for the following:
 1. The player character
 2. The enemy character (note that both characters will look the same)
 3. The floor object
 4. Both the Wall objects that we created
 5. The Victory Box object

Keep in mind that assets related to lighting and sound should remain untouched.
5. Build the lighting for **Level1** by pressing the **Build** button. This button is to the left of the **Play** button, in the **Toolbar** at the top of the editor window.
6. Once you've followed these steps, you should have an empty floor with just the objects you'll be needing for this

level (the ones mentioned in *Step 4*).

Here's the **Level1** map before and after you followed *Steps 4 and 5*, respectively:



Figure 9.17: Before deleting the required objects

Once you have deleted the objects, your floor should look as follows:

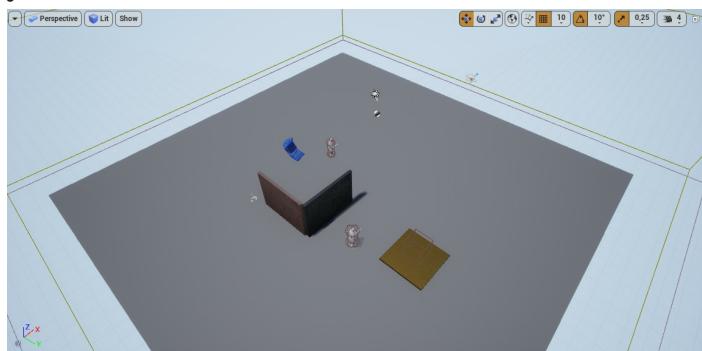


Figure 9.18: After deleting the required objects

Because building a level, even a simple one, is something that takes a lot of steps and instructions, you will simply be shown a few screenshots of a

possible level and, again, be encouraged to come up with your own.

7. In this case, we have simply used the existing **EnemyCharacter**, **Wall**, and **GhostWall** objects and duplicated them several times to create a simple layout that the player can traverse from start to finish. We also moved the **VictoryBox** object so that it matches the new level's end location:



Figure 9.19: The created level – isometric view

The level can be seen in a top-down view as follows:

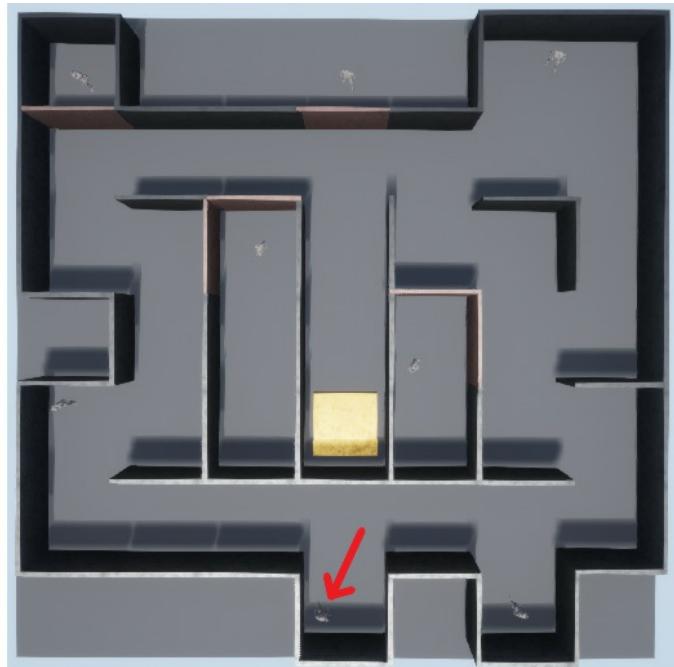


Figure 9.20: The created level – top-down view with the player character marked with an arrow

Once you're happy with the result, this means you have finished your Dodgeball game, and can now ask for your friends and family to play it and see what they think. Great job – you are one step closer to mastering the art of game development!

Extra Features

Before we conclude this chapter, here are some suggestions on what you can do next in this dodgeball project:

- Make it so that the normal **wall** class we created in a previous chapter doesn't block the enemy's line of sight. This way, the enemy will always throw

dodgeballs at the player, which should still be blocked from going through this wall.

- Add a new feature that will allow the player to visualize where the dodgeball thrown by the enemy character will impact first, using the concept of Sweep Traces.
- Add a new type of wall that blocks the player character, the enemy character, and the dodgeballs, but that also takes damage from dodgeballs and gets destroyed when it runs out of health points.

There is a whole world of possibilities for expanding the scope of this project. You are encouraged to use the skills you've learned, and to do further research, to build new features and add more complexity to your game.

Summary

You have now completed the dodgeball game project. In this chapter, you learned how to add polish to your game by playing audio and using Particle Systems. You now know how to add 2D and 3D sounds to your game, as well as some of the tools at your disposal in regard to that. Now, you can try to add even more sounds effects to your game, such as a special sound effect for when an enemy character sees you for the first time (such as in Metal Gear Solid), a footstep sound

effect, or a victory sound effect.

You also built a level using all the tools that you made throughout the last few chapters, thus culminating all the logic we built in this project.

In the next chapter, we'll be starting a new project: the **SuperSideScroller** game. In that project, you'll be introduced to such topics as power-ups, collectibles, enemy **Artificial Intelligence (AI)**, character animation, and much more. You will be creating a side-scrolling platformer game where you control a character that must complete a level, collect gems, and use power-ups to avoid the enemies. The two most important topics you will learn about are UE4's behavior trees and Blackboards, which fuel the AI system, and Animation Blueprints, which allow you to manage your character's animations.

10. Creating a SuperSideScroller Game

Overview

*In this chapter, we will set up the project for a new **SuperSideScroller** game. You will be introduced to the different aspects of a side-scroller game, including power-ups, collectibles, and enemy AI, all of which we will be using in our project. You will also learn about the Character animation pipeline in game development and see how to manipulate the movement of our game's Character.*

By the end of this chapter, you will be able to create a side-scroller project, manipulate the default mannequin skeleton for our Character, import Characters and animations, and create Character and Animation Blueprints.

Introduction

So far, we have learned a lot about the Unreal Engine, C++ programming, and general game development techniques and strategies. In previous chapters, we covered topics such as collisions, tracing, how to use C++ with Unreal Engine 4, and even the Blueprint Visual Scripting system. On top of that, we gained crucial knowledge of Skeletons, animations, and Animation Blueprints that we will utilize in the upcoming project.

For our newest project, **SuperSideScroller**, we will use

many of the same concepts and tools that we have used in previous chapters to develop our game features and systems. Concepts such as collision, input, and the HUD will be at the forefront of our project; however, we will also be diving into new concepts involving animation to recreate the mechanics of popular side-scrolling games. The final project will be a culmination of everything we have learned thus far in this book.

There are countless examples of side-scroller games out there that can be used as references for this project. Most recently, some popular side-scrolling games have included titles such as *Celeste*, *Hollow Knight*, and *Shovel Knight*, but there is also a deep, rich history behind the side-scroller/platformer genre, which we will discuss in this chapter.

Project Breakdown

Let's consider the example of the famous *Super Mario Bros*, released on the **Nintendo Entertainment System (NES)** console in 1985. This game was created by Nintendo and designed by Shigeru Miyamoto. For those who are unfamiliar with the franchise, the general idea is this: the player takes control of Mario, who must traverse the many hazardous obstacles and creatures of the Mushroom Kingdom in the hope of rescuing Princess Peach from the sinister King Koopa, Bowser.

Note

To have an even better understanding of how the game works, feel free to play it online for free at <https://supermariobros.io/>. A more in-depth wiki of the entire Super Mario Brothers franchise can be found here: https://www.mariowiki.com/Super_Mario_Bros.

The following are the core features and mechanics of games in this genre:

1. **Two-Dimensional Movement:** The player can only move in the x and y directions, using a 2D coordinate system. Refer to *Figure 10.1* to see a comparison of 2D and 3D coordinate systems if you are unfamiliar with them. Although our **SuperSideScroller** game will be in 3D and not pure 2D, the movement of our Character will work identically to that of Mario, only supporting vertical and horizontal movement:

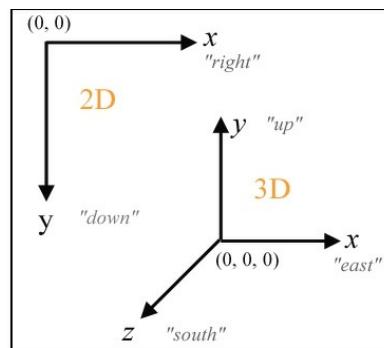


Figure 10.1: A comparison of 2D and 3D coordinate vectors

2. **Jumping:** Jumping is one of the most crucial aspects of any platformer game, and our **SuperSideScroller** game will be no different. There are many different games, such as *Celeste*,

Hollow Knight, and *Super Meat Boy*, as mentioned before, that use the jumping feature – all of which are in 2D.

3. **Character Power-Ups:** Without Character power-ups, many side-scrolling games lose their sense of chaos and replayability. For instance, in the game *Ori and the Blind Forest*, the developers introduce different Character abilities that change how the game is played. Abilities such as the triple-jump or the air dash open a variety of possibilities to navigate the level and allow level designers to create interesting layouts based on the movement abilities of the player.
4. **Enemy AI:** Enemies with various abilities and behaviors are introduced to add a layer of challenge for the player, on top of the challenge of navigating the level solely through the use of the available movement mechanics.

Note

*What are some ways that AI in games can interact with the player? For example, in *The Elder Scrolls V: Skyrim*, there are AI characters in various towns and villages that can have conversations with the player to exposit world-building elements such as history, sell items to the player, and even give quests to the player.*

5. **Collectibles:** Many games support collectible items in one form or another; *Sonic the Hedgehog* has rings, and *Ratchet & Clank* has bolts to collect. Our **SuperSideScroller** game will allow players to collect coins.

Now that we have evaluated the game mechanics that we want to support, we can break down the functionality of each mechanic as it relates to our **SuperSideScroller** and what we need to do to implement these features.

The Player Character

Almost all of the functionality that we want for our Character is given to us by default when using the **Side Scroller** game project template in Unreal Engine 4.

Note

At the time of writing, we are using Unreal Engine version 4.24.2; using another version of the engine could result in some differences in the editor, the tools, and how your logic will work later on, so please keep this in mind.

For now, let's begin creating our project in the following exercise.

Exercise 10.01: Creating the Side-Scroller Project and Using the Character Movement Component

In this exercise, you will be setting up Unreal Engine 4 with the **Side Scroller** template. This exercise will help you get started with our game.

The following steps will help you complete the exercise:

1. First, open the Epic Games Launcher, navigate to the **Unreal Engine** tab at the bottom of the options on the left-hand side, and select the **Library** option at the top.
2. Next, you will be prompted with a window asking you to either open an existing project or create a new project of a certain category. Among these options is the **Games** category; select

this option for our project. With your project category selected, you are now prompted to select the template for your project.

3. Next, click on the **Side Scroller** option because we want our game to use 3D Skeletal Meshes and animations, and not just 2D textures, flipbooks, and other features of the Paper2D toolset.

Note

*Be sure to select the correct **Side Scroller** option, because Unreal Engine 4 has two types of Side Scroller projects: **Side Scroller** and **2D Side Scroller**.*

We will discuss the main differences between these two project templates shortly after this exercise.

Lastly, we need to set up our project settings.

4. Choose to base the project on **C++**, not **Blueprints**, to include **Starter Content**, and to use

Desktop/Console as our platform. The remaining project settings can be left as their defaults. Select the location and name the project **SuperSideScroller** and save the project in an appropriate directory of your choice.

5. After these settings are applied, select **Create Project**. When it's done compiling the engine, both the Unreal Editor and Visual Studio will open, and we can get started.

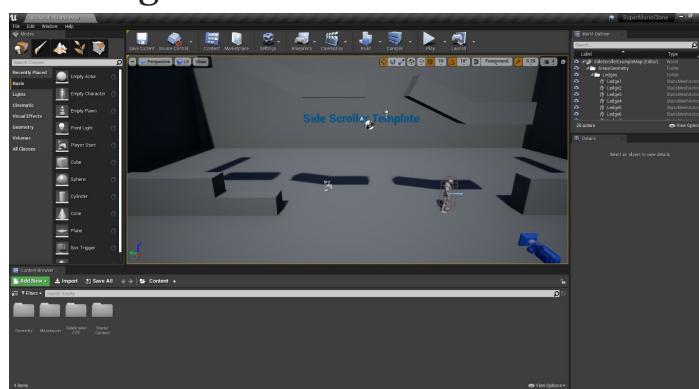


Figure 10.2: The Unreal Engine editor should now be open

Next, we continue to manipulate the Character movement component that exists inside the default **SideScroller** Character and see how this affects the Character. The **Character Movement** component

can only be implemented in **Character** classes and allows bipedal avatars to move by means of *walking*, *jumping*, *flying*, and *swimming*. This component also has a built-in network replication functionality that is necessary for multiplayer games.

6. In **Content Browser**, navigate to the **/SideScrollerCPP/Blueprints/** directory and find the **SideScrollerCharacter Blueprint**:

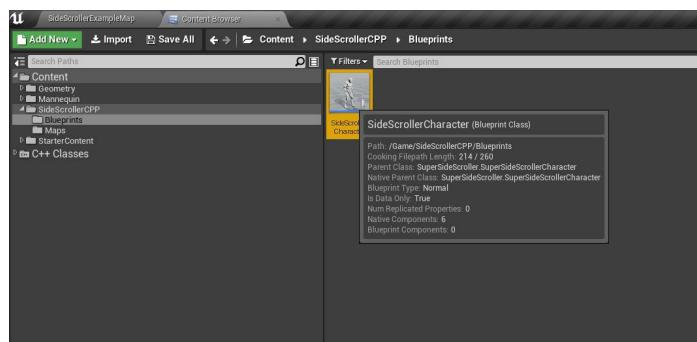


Figure 10.3: The default SideScrollerCharacter Blueprint selected inside Content Browser

7. Double *left-click* the **Blueprint** asset to open the **Blueprint**. Sometimes, if the **Blueprint** does not have any graph logic, you will see what is shown in *Figure 10.4*. If you see this, just *left-click* on **Open Full Blueprint**

Editor:

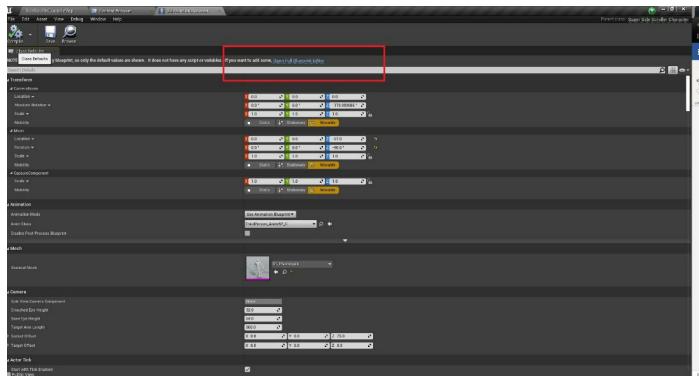


Figure 10.4: When a Blueprint has no graph logic

8. With the Character **Blueprint** opened, we can *left-click* the **CharacterMovement (Inherited)** component in the **Components** tab to view the parameters for this component.
9. Now, under the **Details** panel, we have access to dozens of parameters that affect Character movement. In the **Character Movement: Walking** category, we have the **Max Walk Speed** parameter. Change this value from **600.0f** to **2000.0f**.
10. Lastly, compile and save our Character **Blueprint**. Now, if we play in the editor, we can observe how fast our

player Character is moving:

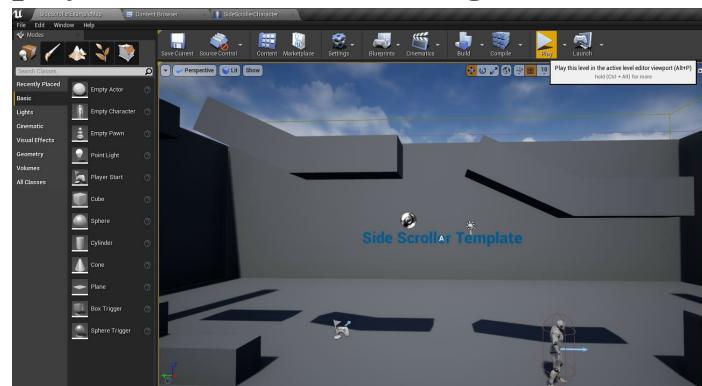


Figure 10.5: If we play in the editor, we can see that our Character moves much faster

Now that you have completed the exercise, you have experienced first-hand the control you have over how the player Character moves! Try changing the value of **Max Walk Speed** and observe in-game how such changes affect the Character.

Side Scroller versus 2D Side Scroller

Let's take a brief moment here to learn about the main differences between the **2D Side Scroller** project template and the **Side Scroller** template. The **2D Side Scroller** template uses the Paper2D system built with Unreal Engine 4, which takes advantage of texture-based animation via textures, sprites, and paper flipbooks.

Note

For more details about Paper2D, please refer to the following documentation: <https://docs.unrealengine.com/en-US/>

<US/Engine/Paper2D/index.html>

There is enough material about Paper2D to warrant its own textbook, so we will not cover much more of that topic. The **Side Scroller** template, however, is almost identical to the 2D version, except we are using 3D animated skeletons instead of 2D animation.

Now, let's move on and look at performing our first activity to manipulate the player Character's jump movement.

Activity 10.01: Making Our Character Jump Higher

In this activity, we will be manipulating a new parameter (**jump**) that exists within the **CharacterMovement** component of the default **Side Scroller** Character Blueprint to observe how these properties affect how our Character moves.

We will be implementing what we learned from *Exercise 10.01, Creating the Side-Scroller Project and Using the Character Movement Component*, and applying that to how to create our Character power-ups and the general movement feel of our Character.

The following steps will help you complete the activity:

1. Head to **SideScrollerCharacter** Blueprint and find the **Jump Z Velocity** parameter in the **CharacterMovement** component.

2. Change this parameter from the default **1000 . 0f** to a value of **2000 . 0f**.
3. Compile and save the **SideScrollerCharacter** Blueprint and play in the editor. Observe how high our Character can jump using the space bar on your keyboard.
4. Stop playing in the editor, return to the **SideScrollerCharacter** Blueprint, and update **Jump Z Velocity** from a value of **2000 . 0f** to **200 . 0f**.
5. Compile and save the Blueprint again, play in the editor, and watch the Character jump.

Expected output:



Figure 10.6: The expected output with the jumping Character

Note

The solution to this activity can be found at:

<https://packt.live/338jEBx>.

Now that we have completed this activity, we have a better understanding of how a few changes to the **CharacterMovement** component parameters can affect our player Character. We can use this later on when we need to give our Character basic movement behaviors such as **Walking Speed** and **Jump Z Velocity** to achieve the Character feel we want. Before moving on, return the Jump Z Velocity parameter back to its default value 1000.of.

We will also keep these parameters in mind when we develop our player Character power-ups later on in our project.

Features of Our Side-Scroller Game

Let's now take some time to lay out the specifics of the game we'll be designing. Many of these features will be implemented in later chapters, but now is a good time to lay out the vision for the project.

Enemy Character

One thing you should have noticed while playing the **SuperSideScroller** project is that there is no enemy AI provided to you by default. So, let's discuss the type of enemies we will want to support and how they will work. Our **SuperSideScroller** project will support one enemy type.

The enemy will have a basic back-and-forth movement pattern and will not support any attacks; only by colliding with the player Character will they be able to inflict any damage. However, we need to set the two locations to move between for the enemy AI, and next, we will need to decide whether the AI should change locations. Should they constantly move between locations, or should there be a pause before selecting a new location to move to?

Finally, we decide whether our AI should be aware of where the player is at all times. If the player comes within a certain range of our enemy, should the enemy know about this and aggressively move toward the player's last known location?

In *Chapter 13, Enemy Artificial Intelligence*, we will use the tools available in Unreal Engine 4 to develop this AI logic.

Power-Up

The **SuperSideScroller** game project will support one type of power-up, in the form of a potion that the player can pick up from the environment. This potion power-up will increase the movement speed of the player and the maximum height to which the player can jump. These effects will only last a short duration before they are removed.

Keeping in mind what you implemented in *Exercise 10.01, Creating the Side-Scroller Project and Using the Character Movement Component*, and *Activity 10.01, Making Our Character Jump Higher*, about the **CharacterMovement** component, you could develop a power-up that changes the effect of gravity on the Character, which would allow interesting new ways to navigate the level and combat enemies.

Collectible

Collectibles in video games serve different purposes. In some cases, collectibles are used as a form of currency to purchase upgrades, items, and other goods. In others, collectibles serve to improve your score or reward you when enough collectibles have been collected. For the **SuperSideScroller** game project, the coins will serve a single purpose: to give the player the goal of collecting as many coins as they can without being destroyed by the enemy.

Let's break down the main aspects of our collectible:

- The collectible needs to interact with our player; this means that we need to use collision detection for the player to collect it and for us to add information to our UI.
- The collectible needs a visual static mesh representation so that the player can identify it in the level.

The final element of our **SuperSideScroller** project is the brick block. The brick block will serve the following purposes for the **SuperSideScroller** game:

- Bricks are used as an element of the level's design. Bricks can be used to access otherwise unreachable areas; enemies can be placed on different elevated sections of bricks to provide

variation in gameplay.

- Bricks can contain collectible coins.
This gives the player an incentive to try
and see which blocks contain
collectibles and which do not.

HUD (Heads-Up Display)

The HUD UI can be used to display to the player important and relevant information, based on the type of game and the mechanics that you support. For the **SuperSideScroller** project, there will be one HUD element, which will display to the player how many coins they have collected. This UI will be updated each time the player collects a coin, and it will reset back to **0** when the player is destroyed.

Now that we have laid out some of the specifics that we will be working toward as part of this project, we will move on to the animation pipeline.

Steps in Animation

To be clear, this book is not going to cover animation. We will not discuss and learn how to make animations using 3D software tools such as 3D Studio Max, Maya, or Blender. However, we will learn how to import these assets into Unreal Engine, use animation assets inside the engine, and use the animation toolsets available to bring our Characters to life.

Character Animation Pipeline

For the purposes of this book, we will only be concerned with 3D animation and how animations work inside Unreal Engine 4; however, it's important to briefly discuss the pipeline used in many industries to create a Character and its animations.

The Concept Stage

The first stage is developing a concept of the Character that we want to create and later animate. This is almost always done in 2D, either by hand or through the use of a computer using programs such as Photoshop. It makes the job easier for the 3D modeler to have several references for how a Character looks, and the relative size of the Character, before starting the process of modeling. Below, we see a basic example of a stick figure Character in different poses. Notice how the Character is posed in different ways:

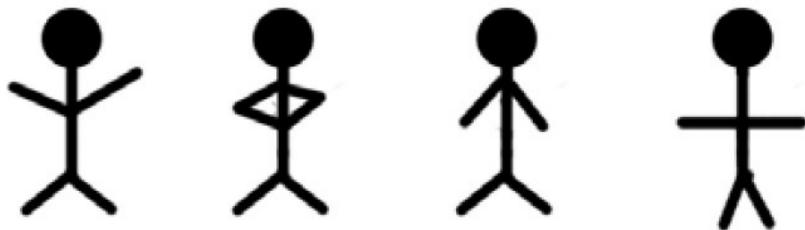


Figure 10.7: A very simple example of a 2D Character concept

The 3D Modeling Stage

Once the Character concepts are complete, the pipeline can

then move to the next stage: making a 3D model of the Character. Models are typically made in programs such as 3D Studio Max or Maya, but this software is relatively expensive, unless you have a student license, and is more often used in professional environments.

Without going into significant detail regarding the complexity of 3D modeling, all we need to know is that 3D artists use computer software to manipulate points in 3D space known as vertices to create objects. These objects are then sculpted into the shapes of our Characters or environment pieces.

The Rigging Stage

Once the final Character model is complete, it is ready to begin the rigging process. The software that was used to model the Character will usually be what is used to rig the Character. Rigging means building a series of bones that form the frame of a Character's skeleton.

In the case of humanoid Characters, we would typically see bones for the head, along the spine, the hips, the legs, and so on; but a skeleton can vary depending on the type of Character you are making. An elephant would have a completely different skeletal rig than a human. It is also possible for the same rig to be applied to different Characters.

Animation

Once we have our Character rigged and a hierarchy of bones, it is time for the animator to take this mesh and bring it to life with animation.

3D animation, in its most basic form, is the manipulation of skeletal bones over time. The process of recording the changes

to bone position, rotation, and scale over time is what results in an animation. With the animation complete, we can export the asset from the 3D software and import it into the engine.

Asset Export and Import

When we have our 3D Character mesh, its skeletal rig, and its animation, it's time to export these assets from the 3D software and import them into Unreal Engine 4. It is important to note that the artists working on the Character, the rig, and the animations will constantly be exporting work-in-progress assets into the engine to get a better idea of the final result as seen in-game. We will be implementing this later on in this chapter in *Activity 10.03, Importing More Custom Animations to Preview the Character Running*, and its accompanying exercise.

Exercise 10.02: Exploring the Persona Editor and Manipulating the Default Mannequin Skeleton Weights

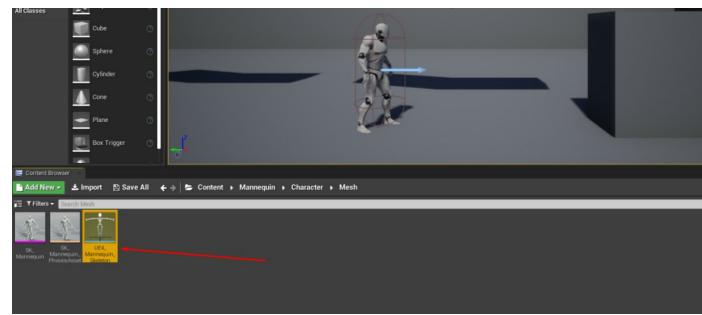
Now that we have a better understanding of the animation pipeline, let's go ahead and take a deeper look into the default mannequin skeletal mesh that is given to us in the **Side Scroller** template project.

Our goal here is to learn more about the default skeletal mesh and the tools that are given to us in the Persona Editor so that

we have a better understanding of how bones, bone weighting, and skeletons work inside Unreal Engine 4.

The following steps will help you complete the exercise:

1. Open the Unreal Engine Editor and navigate to **Content Browser**.
2. Navigate to the **/Mannequin/Character/Mesh/** folder and open the **UE4_Mannequin_Skeleton** asset:



**Figure 10.8: The
UE4_Mannequin_Skeleton asset
is highlighted and visible here**

Upon opening the Skeleton asset, we are shown the **Persona Editor**:

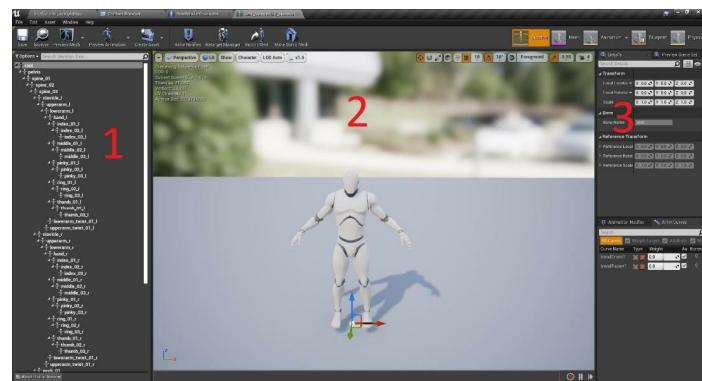


Figure 10.9: The Persona Editor

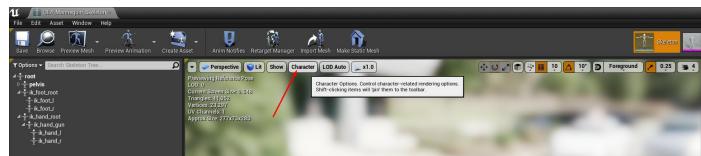
Let's briefly break down the Skeleton Editor of Persona:

1. On the left-hand side (*marked with a 1*), we see the hierarchy of bones that exist in the Skeleton. This is the Skeleton that was made during the rigging process of this Character. The **root** bone, as the name suggests, is the root of the skeletal hierarchy. This means that transformative changes to this bone will affect all of the bones in the hierarchy. From here, we can select a bone or a section of bones and see where they are on the Character mesh.
2. Next, we see the Skeletal Mesh preview window (*marked with a 2*). It shows us our Character mesh, and there are several additional options that we can toggle on that will give us a preview of

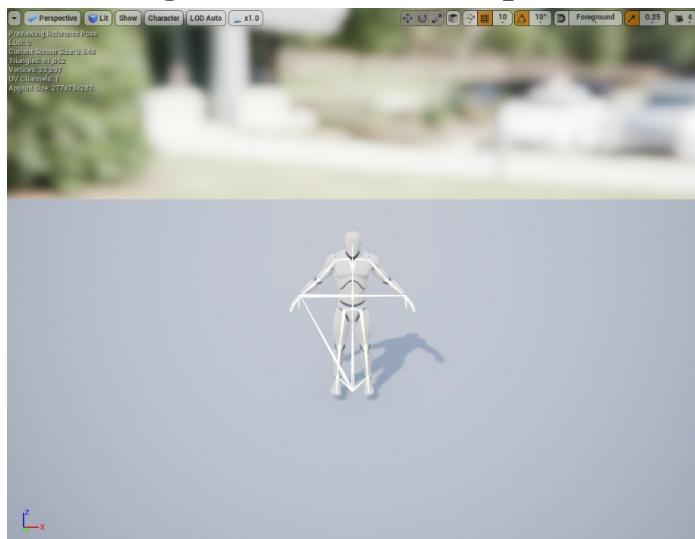
our Skeleton and weight
painting.

3. On the right-hand side (*marked with a 3*), we have basic transformation options where we can modify individual bones or groups of bones. There are also additional settings available that we will take advantage of in the next exercise. Now that we know more about what it is and what we are looking at, let's see what the actual Skeleton looks like on our mannequin.

3. Navigate to **Character**, as shown in *Figure 10.10*:



4. From the drop-down menu, select the **Bones** option. Then, make sure the option for **All Hierarchy** is selected. With this option selected, you will see the outlining Skeleton rendering above the mannequin mesh:



**Figure 10.11: The Skeleton
overlaid on top of the
mannequin Skeletal Mesh**

5. Now, hide the mesh and simply preview the skeletal hierarchy, for which we can disable the **Mesh** property:
 1. Navigate to **Character** and, from the drop-down menu, select the **Mesh** option.
 2. Deselect the option for **Mesh**

and the result should be what we see below:

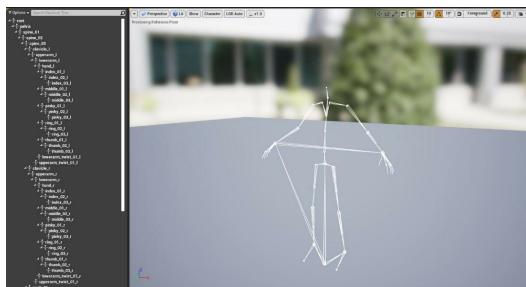


Figure 10.12: The skeletal hierarchy of the default Character

For the purposes of this exercise, let's toggle **Mesh** visibility back on so that we see both the mesh and the Skeleton hierarchy.

Finally, take a look together at the weight scaling for our default Character.

6. To preview this, navigate to **Character** and, from the drop-down menu, select the **Mesh** option. Then, select the option for **Selected Bone Weight** toward the bottom in the section labeled **Mesh Overlay Drawing:**

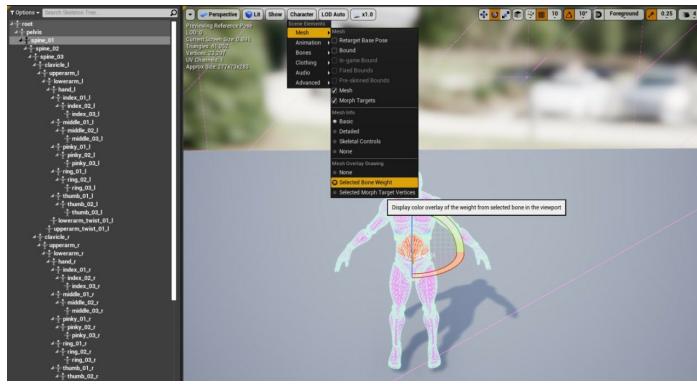


Figure 10.13: Drop-down option to show the selected bone weight of a bone for the mannequin

7. Now, if we select a bone or a group of bones from our hierarchy, we can see how each bone affects a certain area of our mesh:

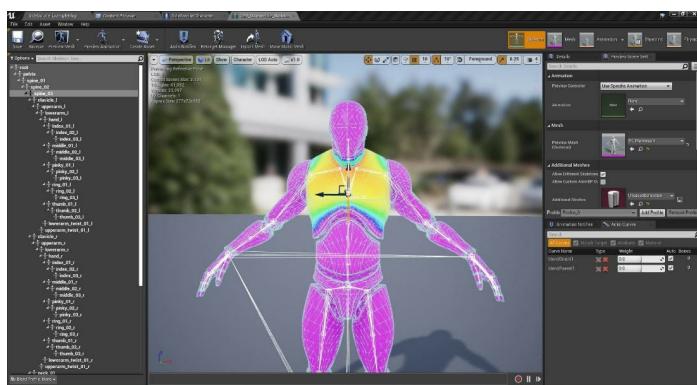


Figure 10.14: This is the weight scaling for the spine_o3 bone

You will notice that when we are previewing the weight scaling for a particular bone, there is a spectrum of colors across different sections of the

Skeletal Mesh. This is the weight scaling shown visually instead of numerically. Colors such as **red**, **orange**, and **yellow** indicate larger weighting for a bone, meaning that the highlighted area of the mesh in these colors will be more affected. In areas that are **blue**, **green**, and **cyan**, they will still be affected, but not as significantly. Lastly, areas that have no overlay highlight will not be affected at all by the manipulation of the selected bone. Keep in mind the hierarchy of the Skeleton because even though the left arm does not have an overlay color, it will still be affected when you are rotating, scaling, and moving the **spine_03** bone, since the arms are children of the **spine_03** bone. Please refer to the image below to see how the arms are connected to the spine:

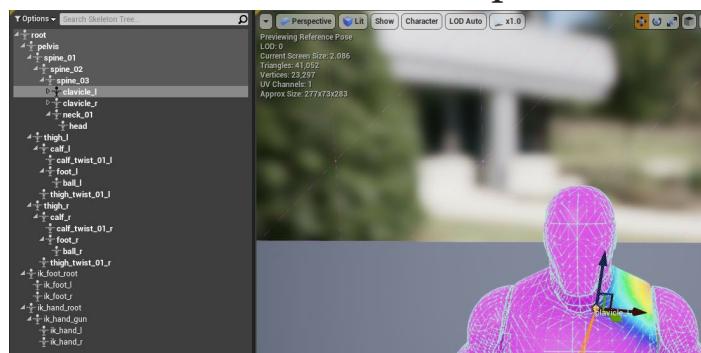
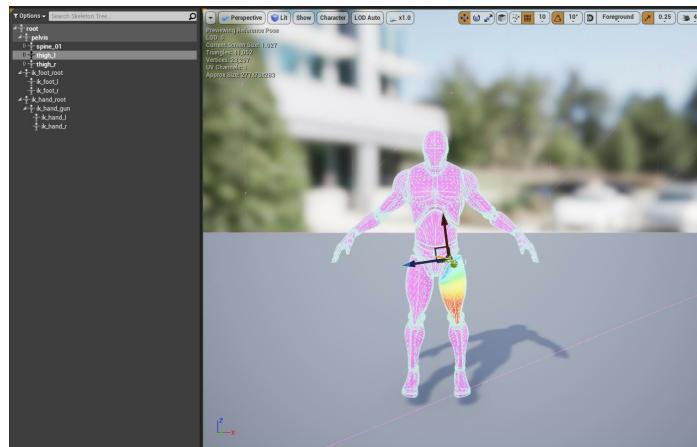


Figure 10.15: The clavicle_1 and

clavicle_r bones are children of the spine_03 bone

Let's continue by manipulating one of the bones on the mannequin Skeletal Mesh and see how these changes affect its animation.

8. In the Persona Editor, *left-click* the **thigh_1** bone in the skeletal hierarchy:



**Figure 10.16: Here, the thigh_1
bone is selected**

With the **thigh_1** bone selected, we have a clear indication of how the weight scaling will affect other parts of the mesh. Also, because of how the Skeleton is structured, any modifications to this bone will not impact the upper body of the mesh:

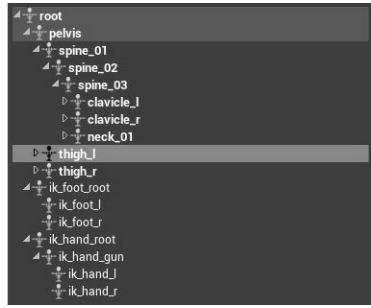


Figure 10.17: You can see that on the skeletal bone hierarchy, the **thigh_l** bone is a child of the **pelvis** bone

- Using the knowledge from earlier chapters, change the Local Location, Local Rotation, and Scale values to offset the transform of the **thigh_l** bone. The image below shows an example of values to use.

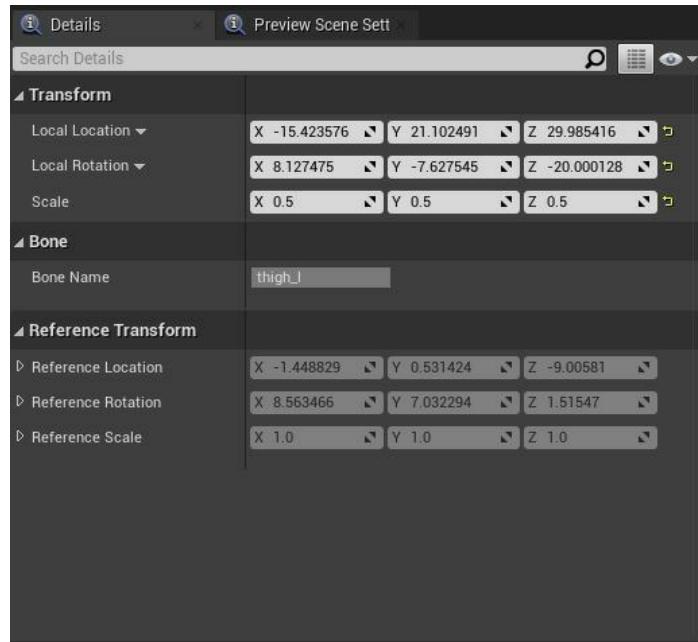


Figure 10.18: The thigh_l values updated

After making the changes to the bone transform, you will see that the mannequin's left leg is completely changed and looks ridiculous:

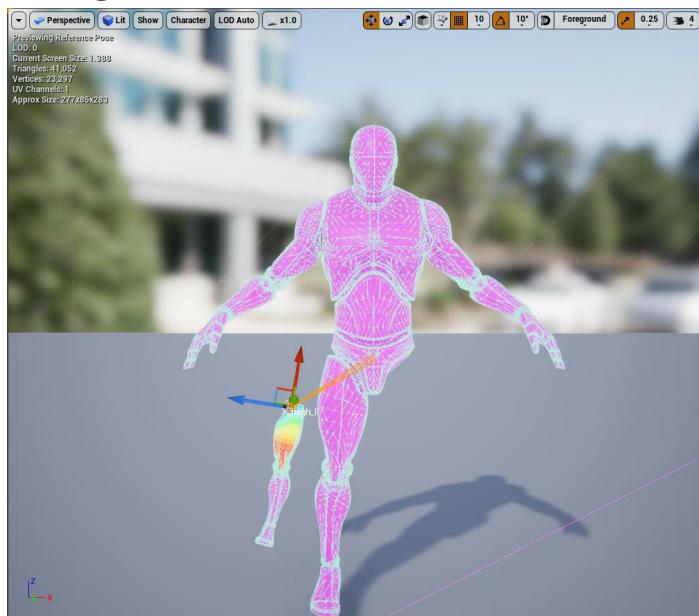


Figure 10.19: The left leg of the Mannequin Character is completely changed

10. Next, in the **Details** panel, head to the tab labeled **Preview Scene Settings**. *Left-click* this tab and you will see new options, displaying some default parameters and an **Animation** section.

11. Use the **Animation** section to preview animations and how they are affected by the changes made to the Skeleton.

For the **Preview Controller** parameter, change that to the **Use Specific Animation** option. By doing this, a new option labeled **Animation** will appear. The **Animation** parameter allows us to choose an animation associated with the Character Skeleton to preview.

12. Next, *left-click* on the drop-down menu and select the **ThirdPersonWalk** animation.
13. Finally, now you can see the mannequin Character playing the walking animation, but their left leg is completely misplaced and mis-scaled:

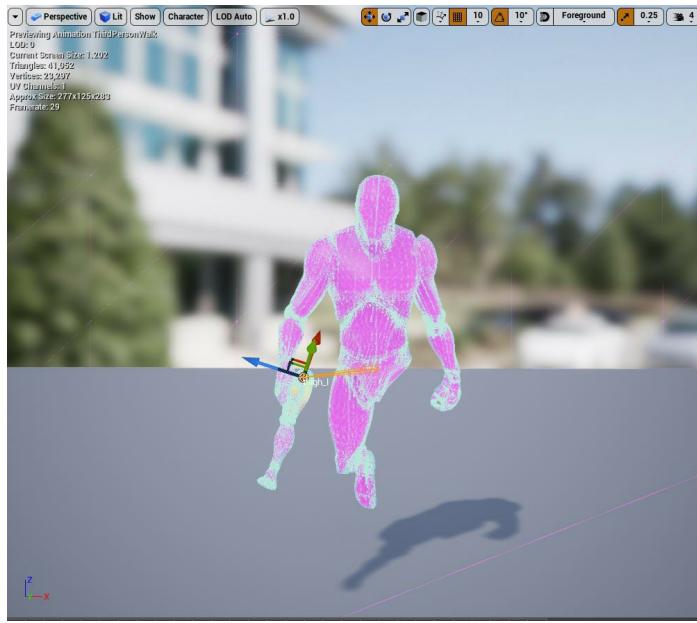


Figure 10.20: Preview of the updated animation for the mannequin Character

Before moving on, make sure to return the **thigh_1** bone back to its original Local Location, Local Rotation, and Scale; otherwise, animations moving forward will not look correct.

Now that you have completed this final part of our second exercise, you have experienced first-hand how skeletal bones affect Characters and animations.

Now, let's move on and perform our second activity to manipulate a different bone on the mannequin Character and observe the results of applying different animations.

Activity 10.02: Skeletal Bone Manipulation and Animations

For this activity, we will put into practice the knowledge we have gained about manipulating bones on the default mannequin to affect how the animations are played out on the Skeleton.

The following steps will help you complete this activity:

1. Select the bone that will affect the entire Skeleton.
2. Change the scale of this bone so that the Character is half its original size. Use these values to change **Scale** to **(X=0.500000, Y=0.500000, Z=0.500000)**.
3. Apply the running animation to this Skeletal Mesh from the **Preview Scene Settings** tab and observe the animation for the half-size Character:

Here is the expected output:

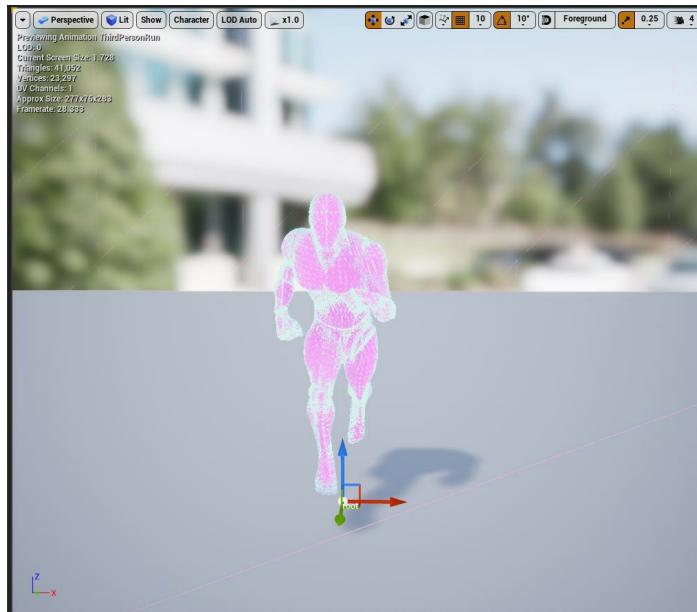


Figure 10.21: Character that is halved in size performing the running animation

Note

The solution to this activity can be found at:

<https://packt.live/338jEBx>.

With this activity complete, you now have practical knowledge of how bone manipulation of Skeletons and Skeletal Meshes affects how animations are applied. You have also seen first-hand the effects of weight scaling for the bones of a Skeleton.

Animations in Unreal Engine 4

Let's break down the main aspects of animations as they function inside Unreal Engine. More in-depth information about the topics in this section can be found in the

documentation that is available directly from Epic Games:
<https://docs.unrealengine.com/en-US/Engine/Animation>.

Skeletons

Skeletons are Unreal Engine's representation of the Character rig that was made in external 3D software; we saw this in *Activity 10.02, Skeletal Bone Manipulation and Animations*. There isn't much more to skeletons that we haven't discussed already, but the main takeaway is that once the Skeleton is in the engine, we can view the skeleton hierarchy, manipulate each individual bone, and add objects known as sockets. What sockets allow us to do is attach objects to the bones of our Character, and we can use these sockets to attach objects such as meshes and manipulate the transformation of the sockets without disrupting the bones' transformation. In first-person shooters, typically a weapon socket is made and attached to the appropriate hand.

Skeletal Meshes

A Skeletal Mesh is a specific kind of mesh that combines the 3D Character model and the hierarchy of bones that make up its skeleton. The main difference between a Static Mesh and a Skeletal Mesh is that Skeletal Meshes are required for objects that use animations, while Static Meshes cannot use animations due to their lack of skeleton. We will look more into our main Character Skeletal Mesh in the next chapter, but we will be importing our main Character Skeletal Mesh in *Activity 10.03, Importing More Custom Animations to Preview the Character Running*, later in this chapter.

Animation Sequences

Finally, an animation sequence is an individual animation that can be played on a specific Skeletal Mesh; the mesh it applies to is determined by the Skeleton selected while importing the animation into the engine. We will look at importing our own Character Skeletal Mesh and a single animation asset together in *Activity 10.03, Importing More Custom Animations to Preview the Character Running.*

Included in our animation sequence is a timeline that allows us to preview the animation frame by frame, with additional controls to pause, loop, rewind, and so on:

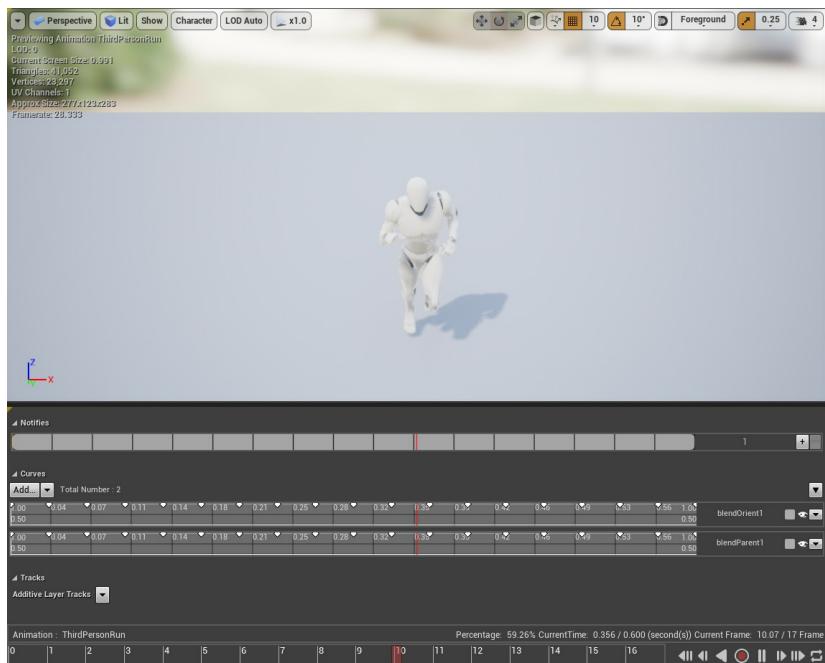


Figure 10.22: The animation sequence timeline and preview window

In the following exercise, you will import a custom Character and an animation. The custom Character will include a Skeletal Mesh and a Skeleton, and the animation will be imported as an animation sequence.

Exercise 10.03: Importing and Setting Up the Character and Animation

For our final exercise, we will import our custom Character and a single animation that we will use for the **SuperSideScroller** game's main Character, as well as creating the necessary Character Blueprint and Animation Blueprint.

Note

*Included with this chapter is a set of files in a folder labeled **Assets**, and it is these files that we will import into the engine. These assets come from Mixamo:*

<https://www.mixamo.com/>; feel free to create an account and view the free 3D Character and animation content available there.

*The **Assets** content is available on our GitHub:*

<https://packt.live/2IcXIOo>.

The following steps will help you complete the exercise:

1. Head to the Unreal Editor.
2. In **Content Browser**, create a new folder named **MainCharacter**.
Within this folder, create two new folders called **Animation** and **Mesh**.
Our **Content Browser** tab should now look like the image below:

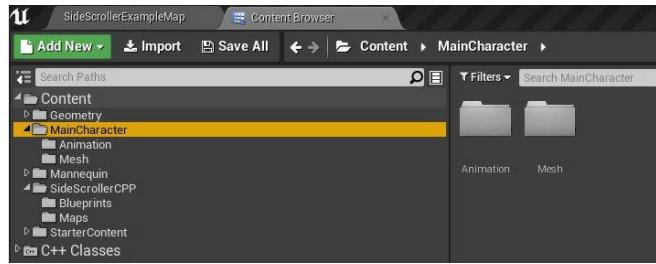


Figure 10.23: Folders added in the MainCharacter directory in Content Browser

3. Next, import our Character mesh. Inside the **Mesh** folder we created, *right-click* and select the **Import** option, which will open the File Explorer menu. Navigate to the directory where you saved the **Assets** folder that accompanies this chapter and find the **MainCharacter.fbx** asset inside the **Character Mesh** folder – for example, **\Assets\Character Mesh\MainCharacter.fbx** – and open that file.

4. When selecting this asset, the FBX import options window will appear. Make sure that the options for **Skeletal Mesh** and **Import Mesh** are set to **check** in their respective checkboxes and leave every other

option set to its default setting.

5. Lastly, we can select the **Import** option so that our FBX asset will be imported into the engine. This will include the necessary materials created within the FBX; a Physics Asset, which will automatically be created for us and assigned to the **Skeletal Mesh**; and the **Skeleton Asset**.

Note

*Ignore any warnings that may appear when importing the **FBX** file; they are unimportant and will not affect our project moving forward.*

Now that we have our Character, let's import an animation.

6. Inside our **Animation** folder in the **MainCharacter** folder directory, again *right-click* and select the option to **Import**.
7. Navigate to the directory where you saved the **Assets** folder that accompanies this chapter and locate

the **Idle.fbx** asset inside the **Animations/Idle** folder – for example,
\Assets\Animations\Idle\Idle.fbx – and open that file.

When selecting this asset, an almost identical window will appear as when we imported our Character Skeletal Mesh. Since this asset is only an animation and not a Skeletal Mesh/Skeleton, we don't have the same options as before, but there is one crucial parameter that we need to set correctly: **Skeleton**.

The **Skeleton** parameter under the **Mesh** category of our **FBX** import options tells the animation to which Skeleton the animation applies. Without this parameter set, we cannot import our animation, and applying the animation to the wrong Skeleton can have disastrous results or cause the animation to not import altogether. Luckily for us, our project is simple and we have already imported our Character Skeletal Mesh and Skeleton.

8. Select **MainCharacter_Skeleton** and choose the option at the bottom, **Import**; leave all other parameters set to their defaults.

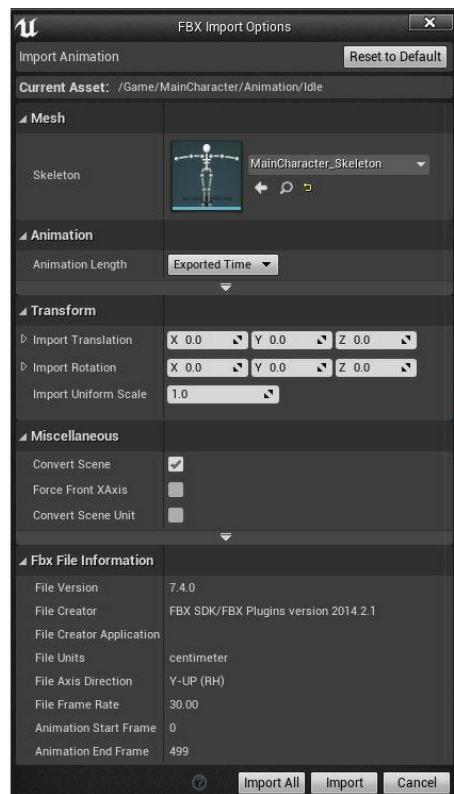


Figure 10.24: The settings when importing the Idle.fbx animation

Now we know to import both a custom Character mesh and an animation. Understanding the importing process for both types of assets is crucial, and in the next activity, you will be challenged to import the remaining animations. Let's continue this exercise by creating both the Character

Blueprint and the Animation Blueprint for the **SuperSideScroller** game's main Character.

Now, although the Side Scroller template project does include a Blueprint for our Character and other assets such as an Animation Blueprint, we will want to create our own versions of these assets for the sake of organization and good practice as game developers.

9. Create a new folder under our **MainCharacter** directory in **Content Browser** and name this folder **Blueprints**. In this directory, create a new Blueprint based on the **SideScrollerCharacter** class under **All Classes**. Name this new Blueprint **BP_SuperSideScroller_MainCharacter**:

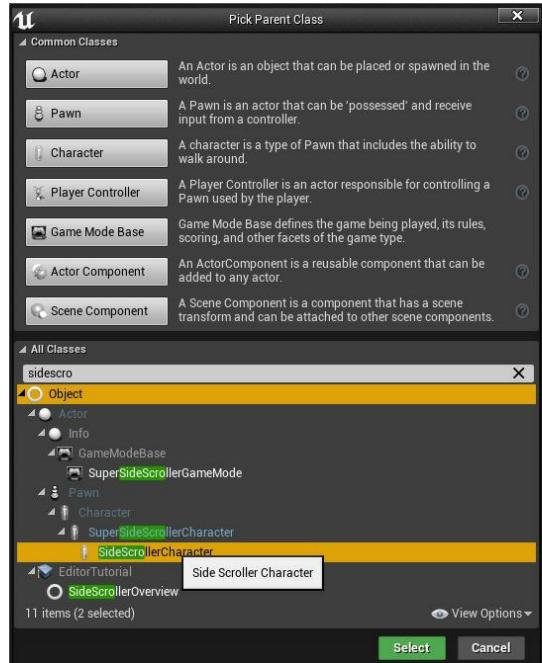


Figure 10.25: The SideScrollerCharacter class to be used as the parent class for our Character Blueprint

10. In our **Blueprints** directory, *right-click* in an empty area of **Content Browser**, hover over the **Animation** option, and select **Animation Blueprint**:

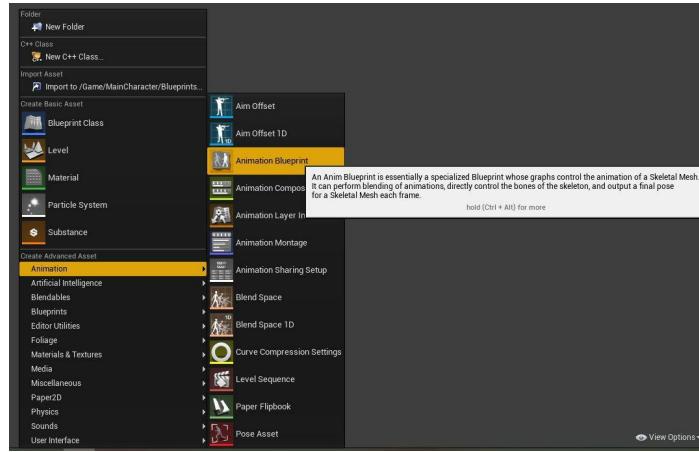


Figure 10.26: The Animation Blueprint option under the Animation category

11. After we select this option, a new window will appear. This new window requires us to apply a parent class and a Skeleton to our Animation Blueprint. In our case, use **MainCharacter_Skeleton**, select OK and name the Animation Blueprint asset **AnimBP_SuperSideScroller_MainCharacter**:

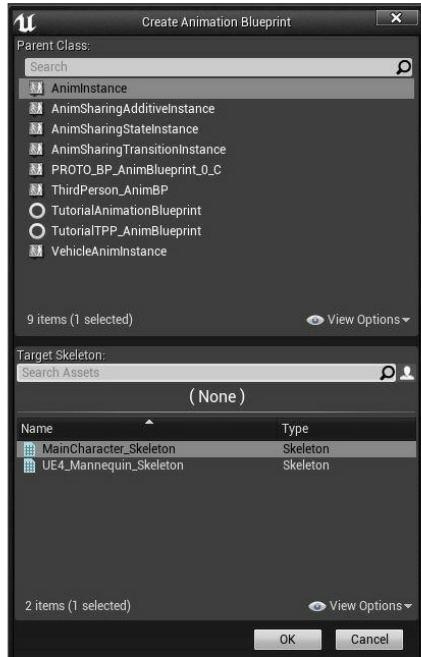


Figure 10.27: The settings we need when creating our Animation Blueprint

12. When we open our Character Blueprint, **BP_SuperSideScroller_MainCharacter**, and select the **Mesh** component, we will find a handful of parameters that we can change:

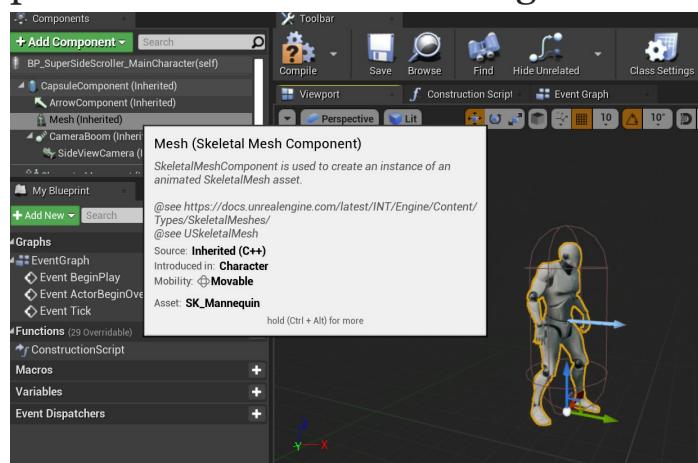


Figure 10.28: The SuperSideScroller Character Blueprint using the mannequin Skeletal Mesh

13. Under the **Mesh** category, we have the option to update the **Skeletal Mesh** used. Find our **MainCharacter** Skeletal Mesh and assign it to this parameter:

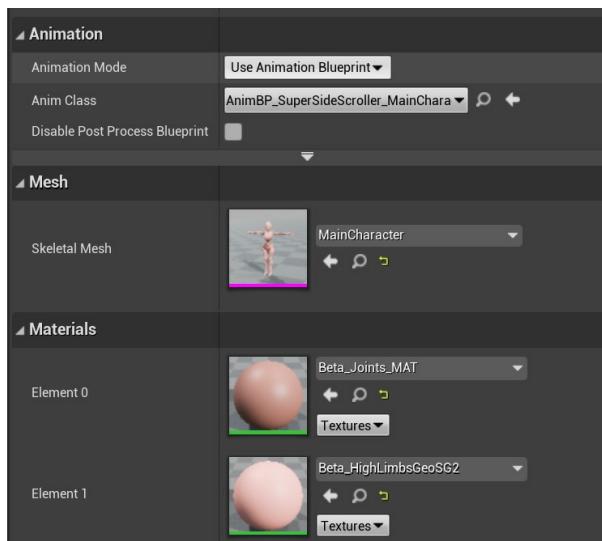


Figure 10.29: The settings we need for our Mesh component to properly use our new Skeletal Mesh and our Animation Blueprint

While still in our Character Blueprint and with the **Mesh** component selected, we can find the **Animation**

category just above the **Mesh** category. Luckily, by default, the **Animation Mode** parameter is already set to **Use Animation Blueprint**, which is the setting we need.

14. Now assign the **Anim** class parameter to our new Animation Blueprint, **AnimBP_SuperSideScroller_MainCharacter**. Finally, head back to our default **SideScrollerExampleMap** level and replace the default Character with our new Character Blueprint.

15. Next, make sure that we have **BP_SuperSideScroller_MainCharacter** selected in our **Content Browser** and then *right-click* on the default Character in our map and choose to replace it with our new Character:

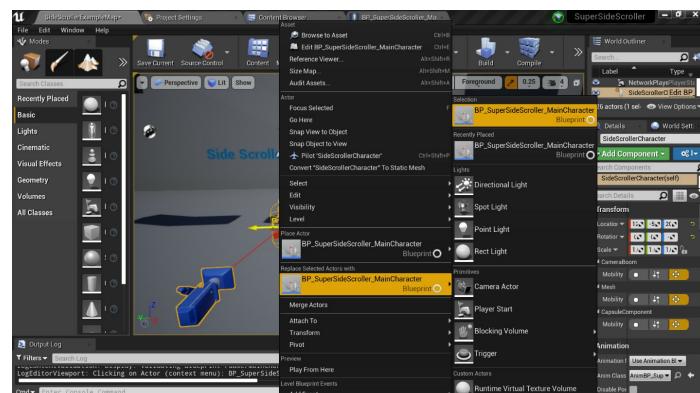


Figure 10.30: With the Character Blueprint selected in Content Browser, we can simply right-click on the default Character in the level and replace it with our new Character

16. With our new Character in the level, we can now play in the editor and move around the level. The result should look something like the image below; our Character in the default T-pose is moving around the level environment:



Figure 10.31: You now have the custom Character running around the level

With our final exercise complete, you now have a full understanding of how to import custom Skeletal Meshes and animations. Additionally, you learned how to create a Character Blueprint and an Animation Blueprint from scratch and how to use those assets to create the base of the **SuperSideScroller** Character.

Let's move on to the final activity of this chapter, where you will be challenged to import the remaining animations for the

Character and to preview the running animation inside Persona Editor.

Activity 10.03: Importing More Custom Animations to Preview the Character Running

This activity aims to import the remaining animations, such as running for the player Character, and to preview the running animation on the Character Skeleton to ensure that it looks correct.

By the end of the activity, all of the player Character animations will be imported into the project and you will be ready to use these animations to bring the player Character to life in the next chapter.

The following steps will help you complete the activity:

1. As a reminder, all of the animation assets we need to import exist in the **\Assets\Animations** directory, wherever you may have saved the original **zip** folder. Import all of the remaining animations in the **MainCharacter/Animation** folder. Importing the remaining animation assets will work the same way as in *Exercise 10.03, Importing and Setting*

Up the Character and Animation,
when you imported the **Idle**
animation.

2. Navigate to the **MainCharacter** skeleton and apply the **Running** animation you imported in the previous step.
3. Finally, with the **Running** animation applied, preview the Character animation in the Persona Editor.

Here is the expected output:

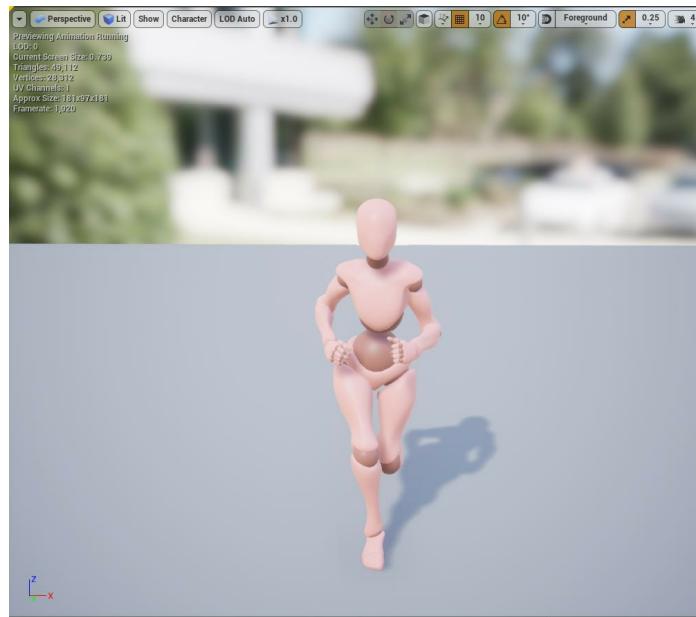


Figure 10.32: The expected output of the Character with additional custom imported assets

Note

The solution to this activity can be found at:

<https://packt.live/338jEBx>.

With this final activity completed, you have now experienced first-hand the process of importing custom skeletal and animation assets into Unreal Engine 4. The import process, regardless of the type of asset you are importing, is commonplace in the games industry and it's vital that you are comfortable with it.

Summary

With the player Character Skeleton, Skeletal Mesh, and animations imported into the engine, we can move on to the next chapter, where you will prepare the Character movement and UpdateAnimation Blueprint so that the Character can animate while moving around the level.

From the exercises and activities of this chapter, you learned about how the Skeleton and bones are used to animate and manipulate the Character. With first-hand experience of importing and applying animations into Unreal Engine 4, you now have a strong understanding of the animation pipeline, from the Character concept to the final assets being imported for your project.

Additionally, you have learned about topics that we will use in the next chapter, such as blend spaces for Character movement animation blending. With the **SuperSideScroller** project template created and the player Character ready, in the next chapter, let's move on to animating the Character with an Animation Blueprint.

11. Blend Spaces 1D, Key Bindings, and State Machines

Overview

This chapter begins by creating the Blend Space asset needed to allow movement animation blending from idle to walking, and finally to running, based on the speed of the player character. We will then implement new key mappings and use those mappings in C++ to code gameplay functionality for the player character, like sprinting. Lastly, we will create a new animation state machine within our character animation blueprint so that the player animations can smoothly transition between movement and jumping.

*By the end of this chapter, you will have the **SuperSideScroller** player character animating correctly when moving around the environment and moving in a way that feels best for the game. This means that the player will support an idle, walking, and sprinting animation, while also supporting the animations needed for jumping.*

Introduction

In the previous chapter, we had a high-level look at animation and the development of the game design for your **SuperSideScroller** project. You were provided with just the beginning steps in the development of the project itself.

You also prepared the player' characters' animation blueprint, character blueprint, and imported all of the required skeletal and animation assets.

At this point, the character can move around the level, but is stuck in the T-Pose and does not animate at all. This can be fixed by creating a new Blend Space for the player character, which will be done in the very first exercise of this chapter. Once the Blend Space is complete, you will implement this in the character animation blueprint in order for the character to animate while moving.

In this chapter, you will be working with many new functions, asset types, and variables in order to achieve the desired movement of the player character. Some of these include the **Try Get Pawn Owner** function within the **Animation Blueprint**, the **1D Blend space asset** type, and **Input Bindings** in the project configuration files.

Let's start this chapter by first learning about Blend Spaces and then creating your Blend Space asset that you will need in order to get the player character animating while moving.

Blend Spaces

Blend Spaces, as the name suggests, allow you to blend between multiple animations based on one or more conditions. Blend Spaces are used in different types of video games, but, more often than not, in games where the player can view the entire character. Blend spaces are not usually used when the player can only see the character arms, such as in the First-Person template project provided in Unreal Engine 4, as shown below:



Figure 11.1: The first-person perspective of the default character in the First-Person project template in Unreal Engine 4.

It is more common in third-person games where there is a need to use Blend Spaces to smoothly blend movement-based animations of the character. A good example is the Third-Person template project provided in Unreal Engine 4, as shown below:



Figure 11.2: The third-person perspective of the default character, in the First-Person project template in Unreal Engine 4

Blend Spaces allow the player character to blend between animations based on a variable, or a set of variables. For example, in the case of *Joel*, from *The Last of Us*, his movement animation is based on the speed at which he is moving, and this speed is provided by the player through the controller sticks (or joystick). With increased speeds, his animation updates from walking, to running, and then to sprinting. This is what we are trying to achieve with our character in this chapter.

Let's look at the Blend Space asset provided by Unreal Engine when creating the **Side Scroller** project template by opening

/Mannequin/Animations/ThirdPerson_IdleRun_2D.

This is a 1D Blend Space asset created for the **Side Scroller** mannequin skeletal mesh so that the player character can smoothly blend between idle, walking, and running animations based on the speed of the character.

If you check **Persona**, in the **Asset Details** panel on the left-hand side, you will see the **Axis Settings** category with the **Horizontal Axis** parameter where we have settings for this axis, which essentially acts as a variable that we can reference in our animation blueprint. Please refer to the image below to see the **Axis Settings** within **Persona**.



Figure 11.3: Shown here are the axis settings for the 1D Blend Space

Below the preview window, we will also see a small graph with points along the line from left to right; one of these points will be highlighted **green**, while the others are **white**. We can *left-click* and drag this **green** point along the horizontal axis to preview the blended animation based on its value. At speed **0**, our character is in **Idle** and, as we move our preview along the axis, the animation will begin to blend Walking, followed by **Running**. Please refer to the following image to view the single-axis graph.

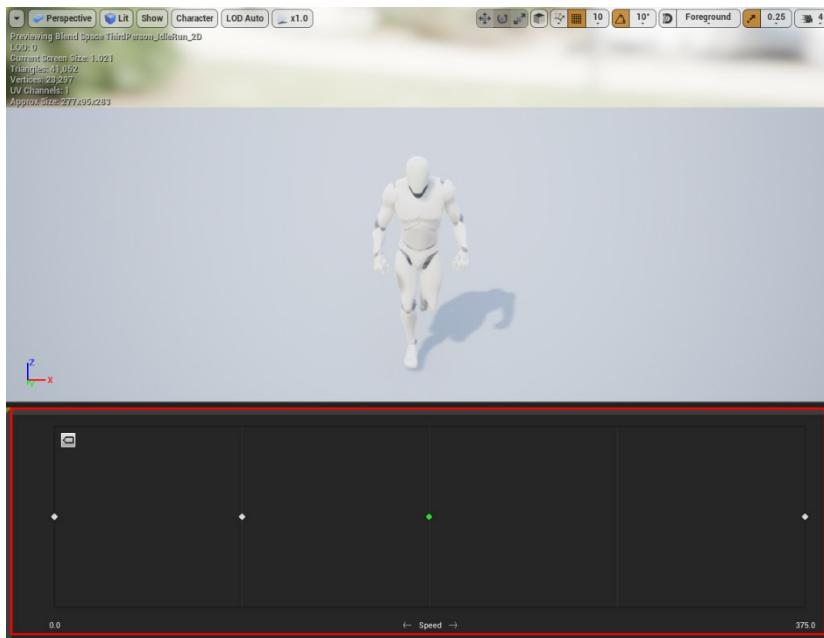


Figure 11.4: Highlighted here is the key frame timeline of the 1D Blend Space

In the next section, we will be looking at 1D Blend Spaces vis-à-vis a normal Blend Space.

1D Blend Space vs

Normal Blend Space

Before moving forward with the 1D Blend Space, let's take a moment to distinguish the main differences between a Blend Space and a 1D Blend Space in Unreal Engine 4.

- The Blend Space asset in Unreal is controlled by two variables, represented by the X and Y axes of the Blend Space graph.
- On the other hand, the 1D Blend Space only supports one axis.

Try to imagine this as a 2D graph. As you know that each axis has its own direction, you can better visualize why and when you would need to use this Blend Space rather than a 1D Blend Space, which only supports a single axis.

Say, for example, you wanted to make the player character strafe left and right while also supporting forward and backward movement. If you were to map this movement out on a graph, it would look like the following figure:

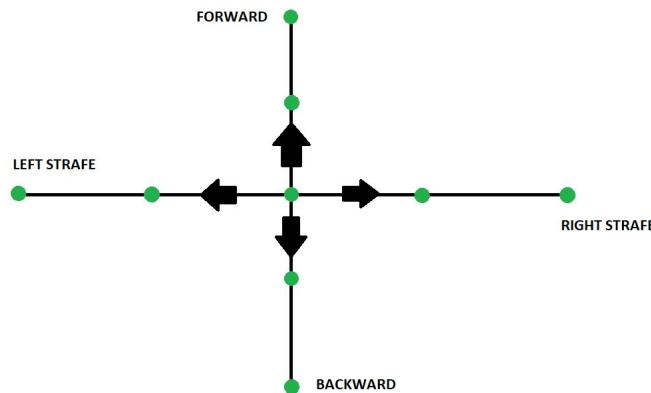


Figure 11.5: This is what a Blend Space movement

would look like on a simple graph

Now, visualize the movement for the player character, keeping in mind the fact that the game is a **Side Scroller**. The character won't be supporting left or right strafing or forward and backward movement. The player character will only need to animate in one direction because the **Side Scroller** character rotates toward the direction of movement by default. Having to only support one direction is why you are using a 1D Blend Space instead of a normal Blend Space.

We will need to set up this type of Blend Space asset for our main character and use the Blend Space for the same purpose, for movement-based animation blending. In the next exercise, let's start by creating the Blend Space asset together using our custom animation assets.

Exercise 11.01: Creating the CharacterMovement 1D Blend Space

In order to get the player character to animate while he moves, you need to first create a Blend Space as discussed previously.

In this exercise, you will create the Blend Space asset, add the idle animation, and update the **CharacterMovement** component so that you assign an appropriate walking speed value that corresponds with the Blend Space.

The following steps will help you to complete the exercise:

1. Navigate to the **/MainCharacter/Animation** folder

in **Content Browser**, where all the new animations you imported in the last chapter are located.

2. Now, *right-click* in the main area of **Content Browser** and, from the drop-down menu, hover over the option for **Animation** and, from its additional drop-down menu, select **Blend Space 1D** by *left-clicking*.
3. Make sure to select **MainCharacter_Skeleton**, and not **UE4_Mannequin_Skeleton**, as the skeleton for the Blend Space.

Note

If you apply the incorrect skeleton, the Blend Space will not be functional for the player character and its custom skeletal mesh when selecting the skeleton for assets such as Blend Spaces or animation blueprints that require one. Here, you are telling this asset with which skeleton it is compatible. By doing so, in the case of a Blend Space, you are able to use animations that are made for this

skeleton and thereby ensure that everything is compatible with everything else.

4. Name this Blend Space asset **SideScroller_IdleRun_1D**.
5. Next, open the **SideScroller_IdleRun_1D** Blend Space asset. You can see the single-axis graph below the preview window:

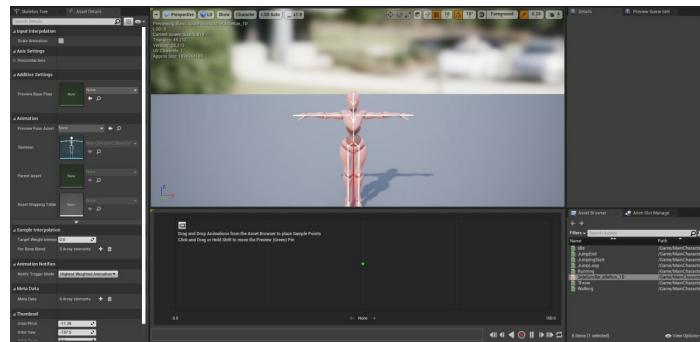


Figure 11.6: The editing tool used to create Blend Spaces in Unreal Engine 4

On the left-hand side of the editor, you have the **Asset Details** panel that contains the **Axis Settings** category. It is here that you will label the axis and provide both a minimum and maximum float value that will later be of use to you in the **Animation Blueprint** for the

player character. Please refer to the figure below to see the default values set for **Horizontal Axis**.

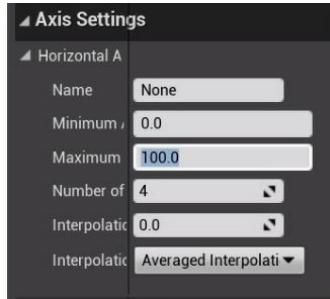


Figure 11.7: The axis settings that affect the axis of the Blend Space

6. Now, rename the **Horizontal Axis** as **Speed**:



Figure 11.8: The horizontal axis is now named Speed

7. The next step is to establish **Minimum Axis Value** and **Maximum Axis Value**. You will want the minimum value to be **0.0f**, which is set by default, because the player character will be in **Idle** when he is not moving

at all.

But what about the **Maximum Axis Value**? This one is a little trickier because you need to bear the following points in mind:

1. You will be supporting a sprinting behavior for the character that allows the player to move faster when holding down the *Left Shift* keyboard button. When released, the player will return to the default walking speed.
2. The walking speed to match the characters' **Max Walk Speed** parameter of the **CharacterMovementComponent**.

Before you set the **Maximum Axis Value**, you need to set the character's **Max Walk Speed** to a value that suits the **SuperSideScroller** game.

8. For this, navigate to **/Game/MainCharacter/Blueprints/** and open the **BP_SuperSideScroller_MainCharacter** blueprint:

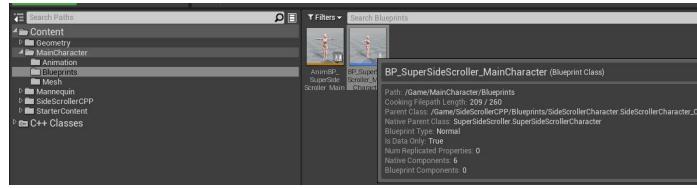


Figure 11.9: The directory of the SuperSideScroller main character blueprint

9. Select the **Character Movement** component and, in the **Details** panel, under the **Character Movement: Walking** category, find the **Max Walk Speed** parameter and set this value to **300.0f**.

With the **Max Walk Speed** parameter set, return to the **SideScroller_IdleRun_1D** Blend Space and set the **Maximum Axis Value** parameter. If the walking speed was **300.0f**, what should the maximum value be? Keeping in mind that you will support sprinting for the player character, this maximum value

needs to be more than the walking speed.

10. Update the **Maximum Axis Value** parameter to a value of **500 . 0f**.
11. Lastly, set the **Number of Grid Divisions** parameter to a value of **5**.
The reason for this is that when working with divisions, a **100** unit spacing between each grid point makes it easier to work with since **Maximum Axis Value** is **500 . 0f**. This is useful in the case of grid point snapping when you apply the movement animations along the grid.
12. Leave the remaining properties set as their defaults:

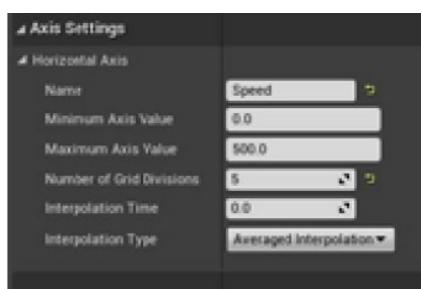


Figure 11.10: The final axis settings for the Blend Space

What you have done here with these settings is that you are telling the

Blend Space to use an incoming float value between **0 . 0f** and **500 . 0f** to blend between the animations that you will place in the next step and the activity. By dividing the grid into **5** divisions, you are able to easily add the animations needed at the correct float value along the axis graph.

Let's continue creating the Blend Space by adding the first animation to the axis graph, the **Idle** animation.

13. To the right of the grid, there is the **Asset Browser** tab. Notice that the list of assets includes all of the animations of the player character that you imported in *Chapter 12, Animation Blending and Montages*. This is because you selected the **MainCharacter_Skeleton** asset when creating the Blend Space.
 14. Next, *left-click* and drag the **Idle** animation to our grid at position **0 . 0**:



Figure 11.11: Dragging the Idle animation to the grid

position 0.0

Notice that when dragging this animation to the grid, it will snap to the grid point. Once the animation is added to the Blend Space, the player character changes from its default T-Pose and starts to play the **Idle** animation:

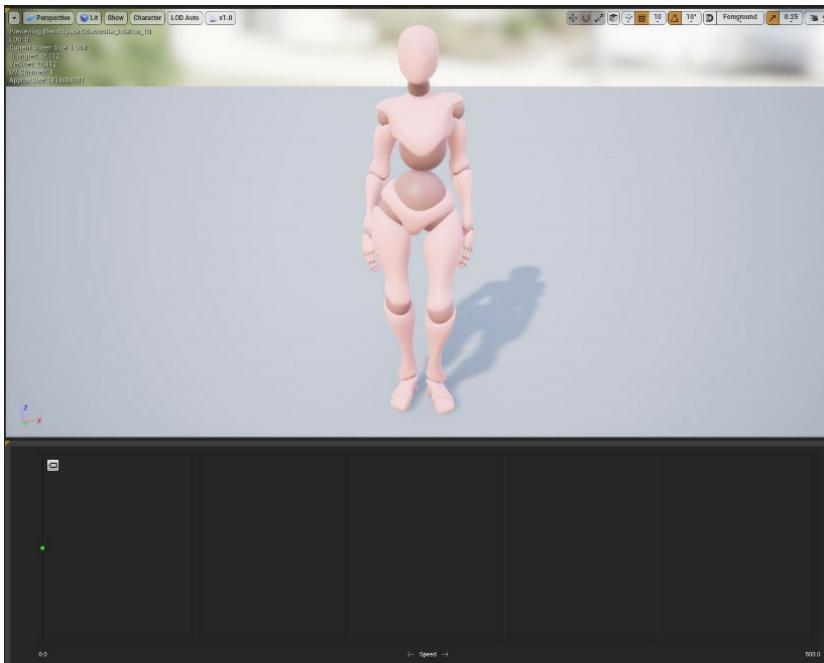


Figure 11.12: With the Idle animation added to the 1D Blend Space, the player character begins to animate

With this exercise complete, you now have an understanding of how to create a 1D Blend Space and, more importantly, you know the differences between a 1D Blend Space and a normal Blend Space. Additionally, you know the importance of aligning the values between the player character movement component and the Blend Space and why you need to ensure that the walking speed correlates appropriately with the values in the Blend Space.

Let's now move on to the first activity of this chapter, where you will be applying the remaining **Walking** and **Running** animations to the Blend Space just as you added the **Idle** animation.

Activity 11.01: Adding the Walking and Running Animations to the Blend Space

The 1D movement Blend Space is coming together nicely so far, but you are missing the walking and running animations. In this activity, you will finish the Blend Space by adding these animations to the Blend Space at the appropriate horizontal axis values that make sense for the main character.

Using the knowledge acquired from *Exercise 11.01, Creating the CharacterMovement 1D Blend Space*, perform the following steps to finish up the character movement Blend Space:

1. Continuing on from *Exercise 11.01, Creating the CharacterMovement 1D Blend Space*, head back to **Asset Browser**.
2. Now, add the **Walking** animation to the horizontal grid position **300.0f**.
3. Finally, add the **Running** animation to the horizontal grid position **500.0f**.

Note

Remember that you can left-click and

drag the green preview grid point along the grid axis to see how the animation blends together based on the axis value, so pay attention to the character animation preview window to make sure that it looks correct.

The expected output is as follows:

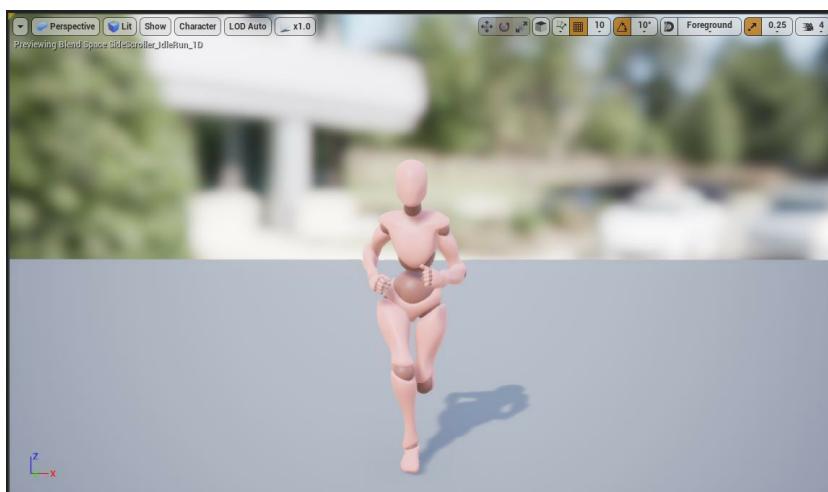


Figure 11.13: The Running animation in the Blend Space

When this activity is complete, you will have a functional Blend Space that blends the character movement animations from **Idle** to **Walking** to **Running** based on the value of the horizontal axis that represents the player character's speed.

Note

*The solution to this activity can be found at:
<https://packt.live/338jEBx>.*

Main Character Animation Blueprint

With the animations added to the Blend Space, you should be able to walk around and see those animations at work, right? Well, no. If you select Play-In-Editor, you will notice that the main character is still moving in the T-Pose. The reason is because you aren't yet telling the animation blueprint to use our Blend Space asset, which you will do later in this chapter.

Animation Blueprints

Before jumping into using the animation blueprint you created in the last chapter, let's briefly discuss what this type of blueprint is, and what its main function is. An animation blueprint is a type of blueprint that allows you to control the animation of a skeleton and skeletal mesh, in this instance, the player character skeleton and mesh you imported in the last chapter.

An animation blueprint is broken into two main graphs:

- Event Graph
- Anim Graph

The Event Graph works as in a normal blueprint where you can use events, functions, and variables to script gameplay logic. The Anim Graph, on the other hand, is unique to an animation blueprint, and this is where you use logic to determine the final pose of the skeleton and skeletal mesh at any given frame. It is here where you can use elements such as state machines, anim slots, Blend Spaces, and other animation-related nodes to then output to the final animation

for the character.

Have a look at the following example (you can follow along).

Open the **AnimBP_SuperSideScroller_MainCharacter** animation blueprint in the **MainCharacter/Blueprints** directory.

By default, **AnimGraph** should open where you can see the character preview, our **Asset Browser** tab, and the main graph. It is inside this **AnimGraph** that you will implement the Blend Space you just created in order to have the player character animate correctly when moving around the level.

Let's get started with the next exercise, where we will do this and learn more about animation blueprints.

Exercise 11.02: Adding the Blend Space to the Character Animation Blueprint

For this exercise, you will add the Blend Space to the animation blueprint and prepare the necessary variable to help control this Blend Space based on the movement speed of the player character. Let's begin by adding the Blend Space to **AnimGraph**.

The following steps will help you to complete this exercise:

1. Add the Blend Space to **AnimGraph** by finding the **Asset Browser** on the

right-hand side, and *left-click* and drag the **SideScroller_IdleRun_1D** Blend Space asset into **AnimGraph**.

Notice that the variable input for this Blend Space node is labeled **Speed**, just like the horizontal axis inside the Blend Space. Please refer to *Figure 11.14* to see the Blend Space in **Asset Browser**.

Note

*If you were to name the **Horizontal Axis** differently, the new name would be shown as the input parameter of the Blend Space.*

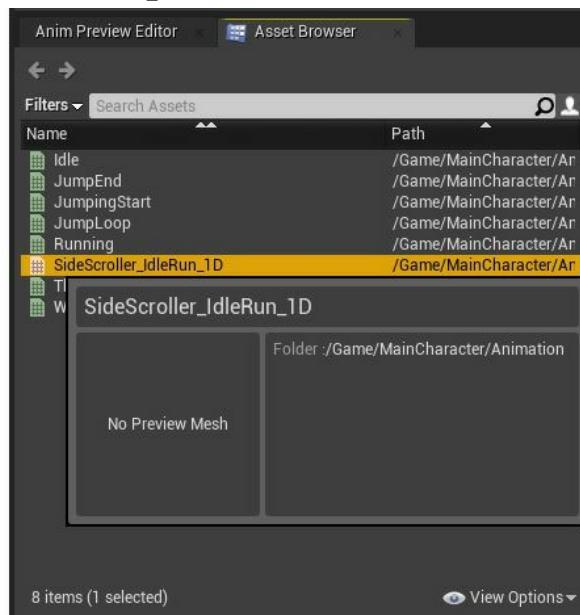


Figure 11.14: Asset Browser gives

**you access to all animation assets
related to the
MainCharacter_Skeleton**

2. Next, connect the **Output Pose** asset of the Blend Space node to the **Result** pin of the **Output Pose** node. Now, the animation pose in the preview shows the character in the **Idle** animation pose:

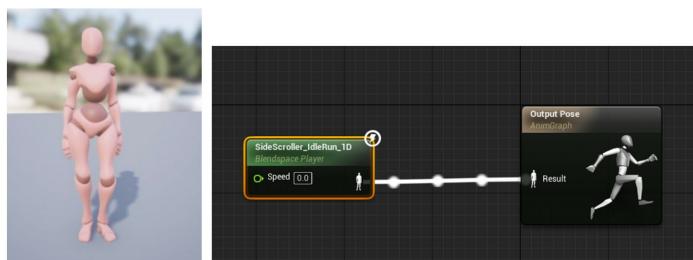


Figure 11.15: You now have limited control of the Blend Space and can manually enter values into the Speed parameter to update the character movement animations

3. If you use **PIE, (Play-In-Editor)**, the player character will be moving around, but will play the **Idle** animation instead of remaining in the T-Pose:

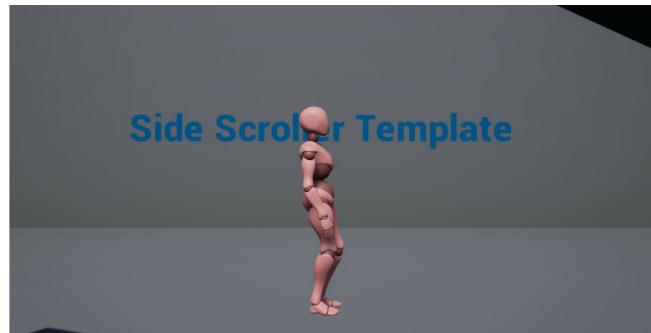


Figure 11.16: The player character now plays the Idle animation in-game

Now, you have the ability to control our Blend Space with our **Speed** input variable. With the ability to use the Blend Space in place, you need a way to store the character's movement speed, and pass that value to the **Speed** input parameter of the Blend Space. This is what you will need to do:

4. Navigate to the **Event Graph** of our animation blueprint. By default, there will be the **Event Blueprint Update Animation** event and a pure **Try Get Pawn Owner** function. Please refer to *Figure 11.17* to see the default setup of **Event Graph**. The event is updated each frame that the animation is updated, and returns the

Delta Time between each frame update and the owning pawn of this animation blueprint. You need to make sure that the owning pawn is of the **SuperSideScroller** player character blueprint class before attempting to get any more information.



Figure 11.17: Animation blueprints include this event and function pair by default to use in your Event Graph

Note

*The main difference between a **Pure** and **Impure** function in Unreal Engine 4 is that a **Pure** function implies that the logic it contains will not modify a variable or member of the class in which it is being used. In the case of **Try Get Pawn Owner**, it is simply returning a reference to the **Pawn** owner of the animation*

*blueprint. **Impure** functions do not have this implication and are free to modify any variable or member it wants.*

5. Get the **Return Value** from the **Try Get Pawn Owner** function and, from the **Context Sensitive** menu that appears, search for the cast to **SuperSideScrollerCharacter**:

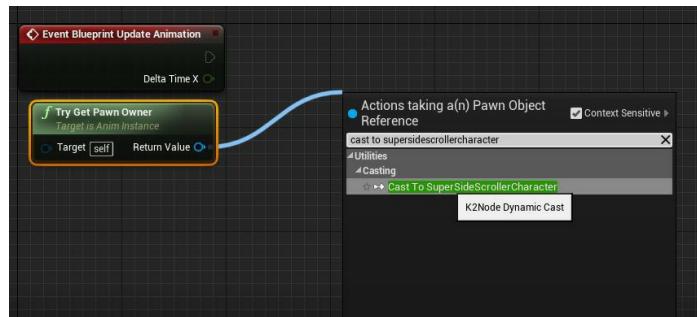


Figure 11.18: The context-sensitive menu finds the related function or variable on which basis actions can be taken on the object you are checking from

6. Connect the execution output pin from **Event Blueprint Update Animation** to the execution input pin of the cast:

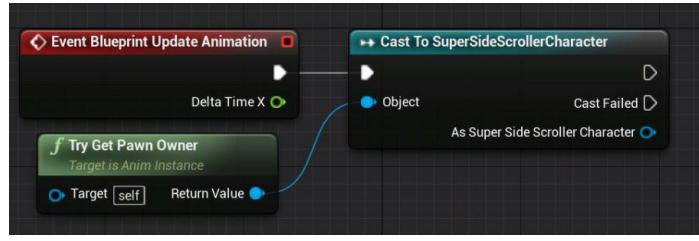


Figure 11.19: Inside the Event Graph, use the Try Get Pawn Owner function to cast the returned Pawn object to the SuperSideScrollerCharacter class

The character blueprint you created inherits from the **SuperSideScrollerCharacter** class. Since the owning pawn of this animation blueprint is your **BP_SuperSideScroller_MainCharacter** character blueprint and this blueprint inherits from the **SuperSideScrollerCharacter** class, the cast function will execute successfully.

7. Next, store the returned value from the cast to its own variable; that way, we have a reference to it in case we need to use it again in our animation blueprint. Refer to *Figure 11.20* and make sure to name this new variable

MainCharacter.

Note

*There is the option in the context-sensitive dropdown for **Promote to Variable**, which allows you to store any valid value type to its own variable.*

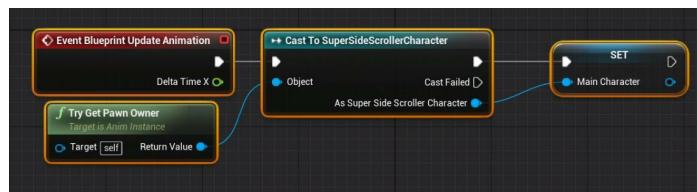


Figure 11.20: As long as the cast is successful, you will want to keep track of the owning character

8. Now, to track the character's speed, use the **Get Velocity** function from the **MainCharacter** variable. Every object from the **Actor** class has access to this function that returns the magnitude and direction vector that the object is moving in:

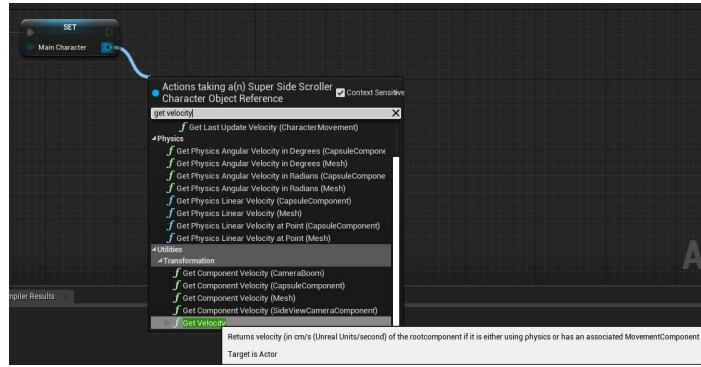


Figure 11.21: The **GetVelocity** function can be found under **Utilities/Transformation**

9. From **Get Velocity**, you can use the **VectorLength** function to get the actual speed:

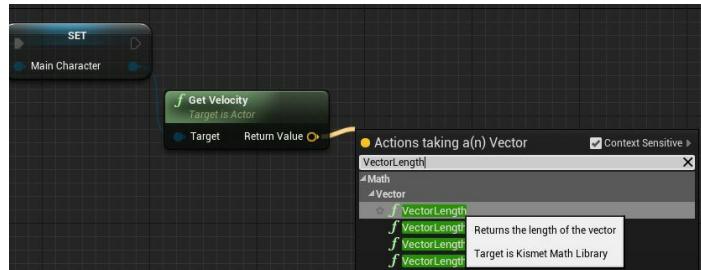


Figure 11.22: The **VectorLength** function returns the magnitude of the vector, but not the direction

10. **Return Value** from the **VectorLength** function can then be promoted to its own variable named **Speed**:



Figure 11.23: Every actor has the Get Velocity function that returns the magnitude and direction of the actor's movement

In this exercise, you were able to obtain the player character speed by using the **GetVelocity** function. The vector returned from the **GetVelocity** function gives the length of the vector to ascertain the actual speed. By storing this value in the **Speed** variable, you can now reference this value in the **AnimGraph** of the animation blueprint to update your Blend Space, which you will do in the next exercise.

Velocity Vectors

Before moving on to the next step, let's explain what you are doing when you get the velocity of the character and promote the vector length of that vector to the **Speed** variable.

What is velocity? Velocity is a vector that has a given **magnitude** and a **direction**. To think about it another way, a vector can be drawn like an *arrow*. The *length of the arrow* represents the **magnitude**, or *strength*, and the *direction of the arrowhead* represents the **direction**. So, if you want to know how fast the player character is moving, you will want to get the length of that vector. That is exactly what you are doing when we use the **GetVelocity** function and the **VectorLength** function on the returned velocity vector; you are getting the value of the **Speed** variable of our character.

That is why you store that value in a variable and use it to control the Blend Space, as shown in the following figure, which is an example of vectors. Where one has a positive (right) direction with a magnitude of **100**, the other has a

negative (left) direction with a magnitude of **35**.

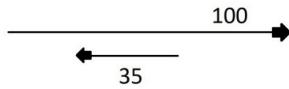


Figure 11.24: Figure showing two different vectors

Exercise 11.03: Adding the Blend Space to the Character Animation Blueprint

Now that you have a better understanding of **vectors** and how to store the **Speed** variable of the player character from the previous exercise, you can follow these next steps to apply the speed to the 1D Blend Space you created earlier in this chapter.

The following steps will help you to complete the exercise:

1. Navigate to the **AnimGraph** within your **AnimBP_SuperSideScroller_MainCharacter** animation blueprint.
2. Use the **Speed** variable to update the Blend Space in real time in the **AnimGraph** by *left-clicking* and dragging the **Speed** variable onto the

graph, and connecting the variable to the input of the **Blendspace Player** function:

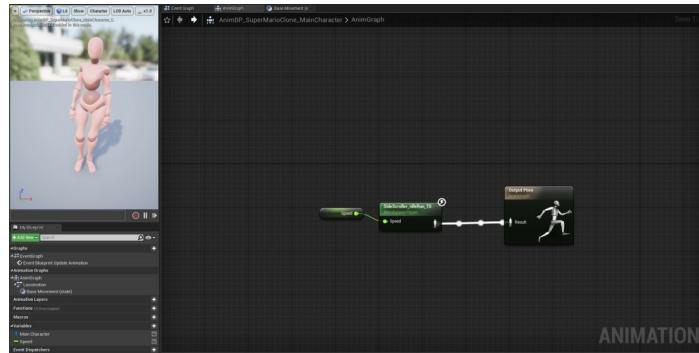


Figure 11.25: You can now use the Speed variable to update the Blend Space in every frame when the animation is updated

3. Next, compile the animation blueprint.

You now have the ability to update the Blend Space based on the speed of the player character. When you use **PIE**, you can see the character in **Idle** and in the **Walking** animation when you move:



Figure 11.26: The player character is finally able to walk around in the level

Finally, the main character is using the movement animations based on movement speed. In the next activity, you will update the character movement component so that you can preview the character running animation from the Blend Space.

Activity 11.02: Previewing the Running Animation In-Game

With the animation blueprint updating and getting the speed of the player character, you are now able to preview the **Idle** and **Walking** animations in-game.

In this activity, you will update the **CharacterMovement** component of the player character blueprint so that you can preview the **Running** animation in-game as well.

Perform the following steps to achieve this:

1. Navigate to, and open, the **BP_SuperSideScroller_MainCharacter** player character blueprint.
2. Access the **CharacterMovement** component.
3. Modify the **Max Walk Speed** parameter to a value of **500 . 0** so that

your character can move fast enough to blend its animation from the **Idle** to **Walking** and finally, to **Running**.

At the end of this activity, you will have allowed the player character to reach a speed that allows you to preview the **Running** animation in-game.

The expected output is as follows:

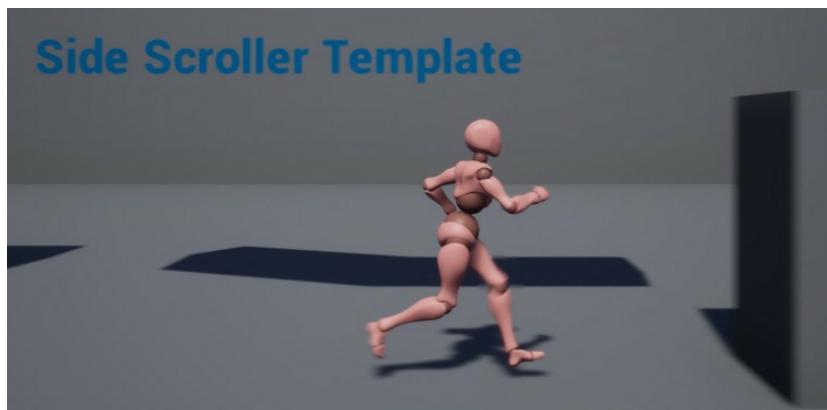


Figure 11.27: The player character running

Note

The solution to this activity can be found at:

<https://packt.live/338jEBx>.

Now that you have handled the player character movement blending from **Idle** to **Walk** and finally to **Run**, let's move on to the next step to add the functionality to allow the player character to move even quicker by sprinting.

Input Bindings

Every game requires input from the player, whether it is the keys on a keyboard such as *W*, *A*, *S*, and *D* for moving the player character, or the thumb sticks on a controller; this is what makes video games an interactive experience. Unreal Engine 4 allows us to map keyboard, mouse, gamepad, and other types of controls to labeled actions or axes that you can then reference in Blueprint or C++ to allow character or gameplay functionality to occur. It is important to point out that each unique action or axis mapping can have one or more key bindings, and that the same key binding can be used for multiple mappings. Input bindings are saved into an initialization file called **DefaultInput.ini** and can be found in the **Config** folder of your project directory.

Note

*Input bindings can be edited directly from the **DefaultInput.ini** file or through **Project Settings** in the editor itself; the latter being more easily accessible and less error-prone when editing.*

Let's add a new input binding for the player character's **Sprint** functionality.

Exercise 11.04: Adding Input for Sprinting and Throwing

With the player character moving around the level, you will now implement a unique character class for the player character that derives from the base **SuperSideScrollerCharacter** C++ class. The reason to do this is so that you can easily differentiate between classes

of the player character and the enemy later on, instead of relying solely on unique blueprint classes.

While creating the unique C++ character class, you will implement the *sprinting* behavior to allow the player character to *walk* and *sprint* as desired.

Let's begin by implementing the **Sprinting** mechanic by first adding the input binding for **Sprint**:

1. Navigate to the **Edit** option on the toolbar at the top of the editor and, from the drop-down list, select **Project Settings**.
2. Within **Project Settings**, navigate to the **Input** option under the **Engine** category on the left-hand side. By default, the **Side Scroller** template project provided by Unreal Engine comes with Action Mappings for **Jump** with the keys *W*, *Up Arrow Key*, *Space Bar*, and *Gamepad Face Button Bottom* bound to it.
3. Add new **Action Mapping** by *left-clicking* on the **+** button next to **Action Mappings**. Label this mapping **Sprint** and add two keys for its controls; **Left Shift** and **Gamepad Right Shoulder**. Please

refer to the figure below for the updated bindings.

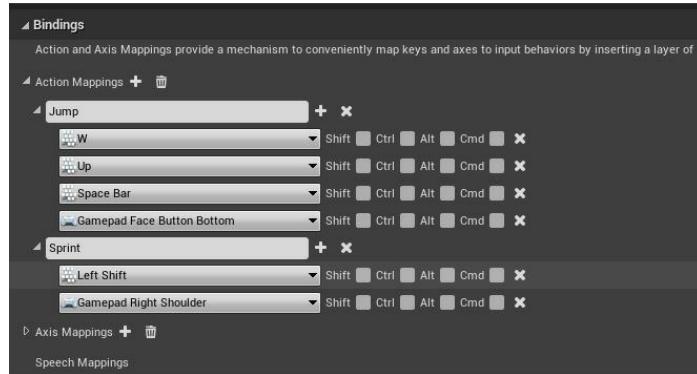


Figure 11.28: The Jump and Sprint Action Mappings applied to key bindings

With the **Sprint** input binding in place, you need to create a new C++ class for the player character based on the **SuperSideScroller** character class.

4. Head back inside the editor, navigate to **File** and, from the drop-down list, select the **New C++ Class** option.
5. The new player character class will inherit from the **SuperSideScrollerCharacter** parent class because this base class has a majority of the functionality needed for the player character. After selecting

the parent class, *left-click* on **Next**.

Please refer to the following image to see how to find the **SuperSideScrollerCharacter** class.

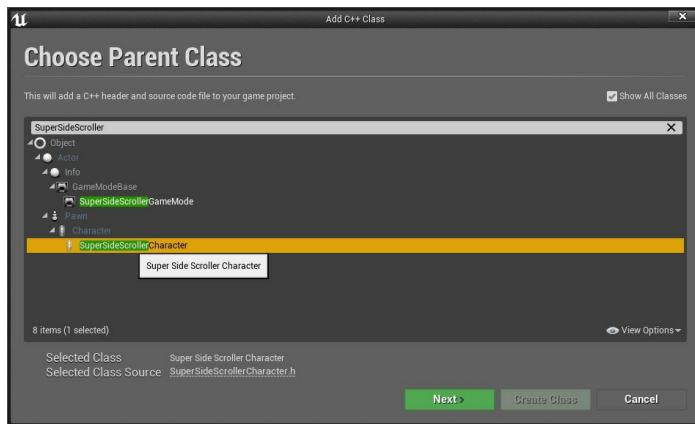


Figure 11.29: Selecting the SuperSideScrollerCharacter parent class

6. Name this new class

SuperSideScroller_Player.

Leave the path as the default that Unreal Engine provides for you, unless you have a need to adjust the file directory of this new class. After naming the new class and selecting the directory to save the class in, *left-click* **Create Class**.

After selecting **Create Class**, Unreal Engine will generate the source and

header files for you and Visual Studio will automatically open these files. You will notice that both the header file and the source file are almost empty. This is OK because you are inheriting from the **SuperSideScrollerCharacter** class and much of the logic you want is done in that class.

7. In **SuperSideScroller_Player**, you will add only the functionality you need on top of what you inherit. You can view the line where the inheritance is taking place inside
- SuperSideScroller_Player.h:**

```
class SUPERSIDESCRROLLER_API
ASuperSideScroller_Player :
public
ASuperSideScrollerCharacter
```

This class declaration is saying that the new **ASuperSideScroller_Player** class inherits from the **ASuperSideScrollerCharacter** class.

By completing this exercise, you were able to add the necessary **Input Binding** for the **Sprint** mechanic that can then be referenced in C++ and used to allow the player to

sprint. Now that you have also created the C++ class for the player character, you can update the code with the **Sprint** functionality, but first you will need to update the **Blueprint** character and the animation blueprint to reference this new class. Let's do this in the next exercise.

What happens when you reparent a blueprint to a new class? Each blueprint inherits from a parent class. In most cases, this is **Actor**, but in the case of your character blueprint, its parent class is **SuperSideScrollerCharacter**. Inheriting from a parent class allows a blueprint to inherit the functionality and variables of that class so that logic can be reused on the blueprint level.

For example, when inheriting from the **SuperSideScrollerCharacter** class, the blueprint inherits components such as the **CharacterMovement** component and the **Mesh** skeletal mesh component, that can then be modified in Blueprint.

Exercise 11.05: Reparenting the Character Blueprint

Now that you have created a new character class for the player character, you need to update the **BP_SuperSideScroller_MainCharacter** blueprint to use the **SuperSideScroller_Player** class as its parent class. If you don't, then any logic you add to the new class will not affect the character made in Blueprint.

Follow these steps to reparent the blueprint to the new character class:

1. Navigate to
/Game/MainCharacter/Blueprints/ and open the
BP_SuperSideScroller_MainCharacter blueprint.
2. Select the **File** option on the toolbar
and, from the drop-down menu, select
the **Reparent Blueprint** option.
3. When selecting the **Reparent Blueprint** option, Unreal will ask for
the new class to reparent the blueprint
to. Search for
SuperSideScroller_Player and
select the option from the dropdown by
left-clicking.

Once you select the new parent class
for the blueprint, Unreal will reload the
blueprint and recompile it, both of
which will happen automatically for
you.

Note

*Be careful when reparenting
blueprints to new parent classes
because this can lead to compile errors*

or settings to be erased or reverted back to class defaults. Unreal Engine will display any warnings or errors that may occur after compiling the blueprint after reparenting to a new class. These warnings and errors usually occur if there is blueprint logic that references variables or other class members that no longer exist in the new parent class. Even if there are no compile errors, it is best to confirm that any logic or settings you have made to your blueprint are still present after the reparenting before moving on with your work.

Now that your character blueprint is correctly reparented to the new **SuperSideScroller_Player** class, you need to also update the **AnimBP_SuperSideScroller_MainCharacter** animation blueprint to ensure that you are casting to the correct class when using the **Try Get Pawn Owner** function.

4. Next, navigate to the **/MainCharacter/Blueprints/** directory and open the

AnimBP_SuperSideScroller_MainCharacter animation blueprint.

5. Open **Event Graph**. From the **Return Value** of the **Try Get Pawn Owner** function, search for **Cast to**

SuperSideScroller_Player:

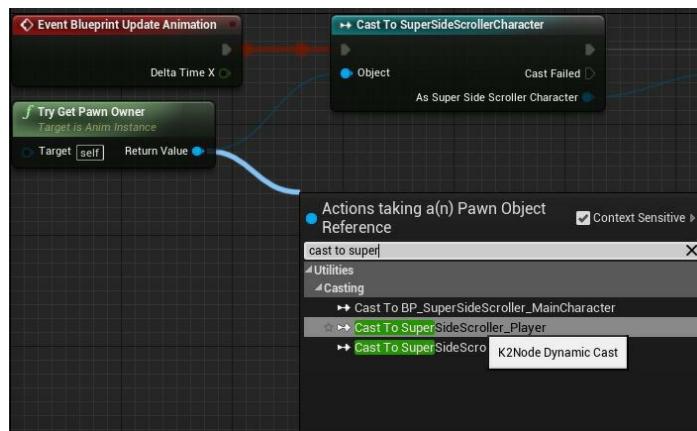


Figure 11.30: Instead of casting to the base SuperSideScrollerCharacter class, you can cast to the new SuperSideScroller_Player class

6. You can then connect the output as a **SuperSideScroller_Player** cast to the **MainCharacter** variable. This works because the **MainCharacter** variable is of the **SuperSideScrollerCharacter** type and the new

SuperSideScroller_Player class
inherits from that class:

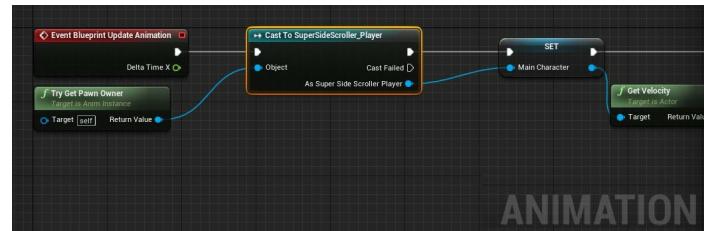


Figure 11.31: You can still use the MainCharacter variable because SuperSideScroller_Player is based on the SuperSideScrollerCharacter due to inheritance

Now that both the **BP_SuperSideScroller_MainCharacter** character blueprint and the **AnimBP_SuperSideScroller_MainCharacter** animation blueprint are referencing your new **SuperSideScroller_Player** class, it is now safe to move forward into C++ and code the character sprint functionality.

Exercise 11.06: Coding the Character Sprint Functionality

With the new **SuperSideScroller_Player** class reference correctly implemented in blueprints following the previous exercise, it is time to start coding the functionality that will allow the player character to sprint.

Perform the following steps to add the **Sprinting** mechanic to the character:

1. The first thing to take care of is the constructor of the **SuperSideScroller_Player** class. Navigate back to Visual Studio and open the **SuperSideScroller_Player.h** header file.
2. You will use the **constructor** function later in this exercise to set initialized values for variables. For now, it will be an empty constructor. Make sure that the declaration is made under the **public** access modifier heading, as seen in the following code:

```
//Constructor  
ASuperSideScroller_Player()  
;
```

3. With the constructor declared, create the constructor function definition in the **SuperSideScroller_Player.cpp** source file:

```
ASuperSideScroller_Player::  
ASuperSideScroller_Player()  
{
```

```
}
```

With the constructor in place, it's time to create the **SetupPlayerInputComponent** function so that you can use the key bindings created earlier to call functions within the **SuperSideScroller_Player** class.

The **SetupPlayerInputComponent** function is a function that the character class has built-in by default, so you need to declare it as a **virtual** function with the **override** specifier. This tells Unreal that you are using this function and intend to redefine its functionality in this new class. Make sure that the declaration is made under the **Protected** access modifier heading.

4. The **SetupPlayerInputComponent** function requires an object of the **UInputComponent** class to be passed into the function, like so:

```
protected:  
    //Override base character  
    class function to setup our
```

```
player input component

virtual void
SetupPlayerInputComponent(c
lass UInputComponent*
PlayerInputComponent)
override;
```

The **UInputComponent*** **PlayerInputComponent** variable is inherited from the **UCharacter** base class that our **ASuperSideScroller_Player()** class derives from, and therefore, must be used as the input parameter of the **SetupPlayerInputComponent()** function. Using any other name will result in a compilation error.

5. Now, in the source file, create the definition of the **SetupPlayerInputComponent** function. In the body of the function, we will use the **Super** keyword to call it:

```
//Not always necessary, but
good practice to call the
function in the base class
with Super.
```

```
Super::SetupPlayerInputComponent(PlayerInputComponent)
;
```

The **Super** keyword enables us to call the **SetupPlayerInputComponent** parent method. With the **SetupPlayerInputComponent** function ready, you need to include the following header files in order to continue with this exercise without any compile errors:

1. **#include**
"Components/InputComponent.h"
2. **#include**
"GameFramework/CharacterMovementComponent.h"

You will need to include the header for the input component in order to bind the key mappings to the sprint functions you will be creating next. The header for the **Character Movement** component will be necessary

for the sprint functions because you will be updating the **Max Walk Speed** parameter based on whether the player is sprinting. The following code is all of the headers that need to be included for the player character:

```
#include  
"SuperSideScroller_Pl  
ayer.h"  
  
#include  
"Components/InputCom  
ponent"  
  
#include  
"GameFramework/Chara  
cterMovementComponen  
t.h"
```

With the necessary headers included in the source file of the **SuperSideScroller_Player** class, you can now create the sprint functions used to make the player character move faster. Let's begin by

declaring the required variable and functions.

6. Under the **Private** access modifier in the header file of the **SuperSideScroller_Player** class, declare a new Boolean variable called **bIsSprinting**. This variable will be used as a failsafe in order to know for certain whether the player character is sprinting before making any changes to the movement speed:

```
private:  
    //Bool to control if we are  
    //sprinting. Failsafe.  
    bool bIsSprinting;
```

7. Next, declare two new functions, **Sprint()**; and **StopSprinting()**. These two functions will not take any arguments and will not return anything. Declare the functions under the **Protected** access modifier:

```
//Sprinting  
void Sprint();
```

```
//StopSprinting  
void StopSprinting();
```

The **Sprint()**; function will be called when the player *presses/holds* the **Sprint** keys mapping to the binding, and **StopSprinting()** will be called when the player *releases* the keys mapped to the binding.

8. Start with the definition of the **Sprint()**; function. In the source file of the **SuperSideScroller_Player** class, create the definition for this function, as shown here:

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

9. Within the function, you will first want to check the value of the **bIsSprinting** variable. If the player is **NOT** sprinting, meaning that **bIsSprinting** is **False**, then move forward with the rest of the function.

10. Within the **If** statement, set the **bIsSprinting** variable to **True**. Then, you can access the **GetCharacterMovement()** function and modify the **MaxWalkSpeed** parameter. Set **MaxWalkSpeed** to **500.0f**. Remember that the **Maximum Axis Value** parameter of the movement Blend Space is **500.0f**. This means that the player character will reach the speed necessary to use the **Running** animation:

```
void
ASuperSideScroller_Player::
Sprint()
{
    if (!bIsSprinting)
    {
        bIsSprinting =
        true;

        GetCharacterMovement() ->MaxWalkSpeed = 500.0f;
    }
}
```

The **StopSprinting()** function will look almost identical to the **Sprint()** function you just wrote, but it works in the opposite manner. You first want to check whether the player *is* sprinting, meaning that **bIsSprinting** is **True**. If so, move forward with the rest of the function.

11. Inside the **If** statement, set **bIsSprinting** to **False**. Then, access the **GetCharacterMovement()** function to modify **MaxWalkSpeed**. Set **MaxWalkSpeed** back to **300.0f**, the default speed for the player character walking. This means that the player character will reach only the speed necessary for the **Walking** animation:

```
void
ASuperSideScroller_Player::
StopSprinting()

{
    if (bIsSprinting)

    {
        bIsSprinting = false;
```

```
        GetCharacterMovement(  
    )->MaxWalkSpeed = 300.0f;  
  
    }  
  
}
```

Now that you have the functions needed for sprinting, it is time to bind these functions to the action mappings you created earlier. In order to do this, perform the following steps within the **SetupPlayerInputComponent** function.

12. Let's begin by binding the **Sprint()** function. Inside the **SetupPlayerInputComponent** function, use the **PlayerInputComponent** variable that is passed to the function so as to call the **BindAction** function.

The parameters we need for **BindAction** are the following:

1. The name of the action mapping as written in **Project Settings** that you set up earlier in this exercise, in this case,

Sprint.

2. The enumerator value of the **EInputEvent** type that you want to use for this binding; in this case you will use **IE_Pressed** because this binding will be for when the **Sprint** key(s) are pressed.

Note

*For more information regarding the **EInputEvent** enumerator type, please refer to the following documentation from Epic Games:*

<https://docs.unrealengine.com/en-US/API/Runtime/Engine/Engine/EInputEvent/index.html>

3. A reference to the class object. In this case, it's the **ASuperSideScroller_Player** class, but you can use the **this** keyword, which

represents the same thing.

4. A delegate to the function that you want to call when this action happens.

The resulting function call will look as shown:

```
//Bind pressed  
action Sprint to  
your Sprint function  
  
PlayerInputComponent  
-  
>BindAction"Sprint",  
IE_Pressed, this,  
&ASuperSideScroller_  
Player::Sprint);
```

13. You will do the same thing for the **StopSprinting()** function, but this time you need to use the **IE_Released** enumerator value, and reference the **StopSprinting** function:

```
//Bind released action  
Sprint to your  
StopSprinting function  
  
PlayerInputComponent-
```

```
>BindAction("Sprint",
IE_Released, this,
&ASuperSideScroller_Player:
:StopSprinting);
```

With **Action Mappings** bound to the sprint functions, the last thing you need to do is set default initialized values of the **bIsSprinting** variable and the **MaxWalkSpeed** parameter from the **Character Movement** component.

14. Inside the **constructor** function in the source file of your **SuperSideScroller_Player** class, add the **bIsSprinting = false** line. This variable is constructed as false because the player character should not be sprinting by default.
15. Finally, set the **MaxWalkSpeed** parameter of the character movement component to **300.0f** by adding the line **GetCharacterMovement() ->MaxWalkSpeed = 300.0f**. Please review the following code:

```
ASuperSideScroller_Player::  
ASuperSideScroller_Player()
```

```
{  
    //Set sprinting to false  
    by default.  
  
    bIsSprinting = false;  
  
    //Set our max Walk Speed  
    to 300.0f  
  
    GetCharacterMovement() -  
    >MaxWalkSpeed = 300.0f;  
}
```

With the initialization of the variables added to the constructor, the **SuperSideScroller_Player** class is done, for now. Return to Unreal Engine and *left-click* on the **Compile** button on the toolbar. This will recompile the code and perform a hot-reload of the editor.

After recompiling and hot-reloading the editor, you can Play-In-Editor and see the fruits of your labor. The base movement behavior is the same as before, but now if you hold *Left Shift* or *Gamepad Right Shoulder* on a controller, the player character will sprint and begin to play the **Running**

animation.

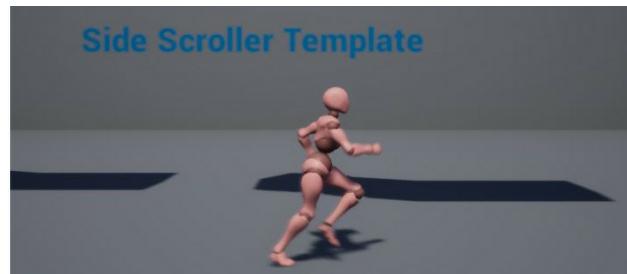


Figure 11.32: The player character can now sprint

With the player character able to sprint, let's move on to the next activity, where you will implement the base **Throw** functionality in a very similar way.

Activity 11.03: Implementing the Throwing Input

One of the features included with this game is the ability for the player to throw projectiles at the enemy. You won't be creating the projectile or implementing the animation in this chapter, but you will set up the key bindings and the C++ implementation for use in the next chapter.

In this activity, you need to set up the key bindings for the **Throw** projectile functionality and implement a debug log in C++ for when the player presses the key(s) mapped to **Throw** by doing the following.

1. Add a new **Throw** input to **Project Settings** in the input binding. Name this binding **ThrowProjectile** and

bind it to *Left-Mouse Button* and *Gamepad Right Trigger*.

2. Within Visual Studio, add a new function to the header file of **SuperSideScroller_Player**. Name this function **ThrowProjectile()**. This will be a void function without parameters.
3. Create the definition in the source file of the **SuperSideScroller_Player** class. In the definition of this function, use **UE_LOG** to print a message letting you know that the function is being called successfully.

Note

*You can learn more about **UE_LOG** here:*

https://www.ue4community.wiki/Logging/Logs,_Printing_Messages_To_Yourself_During_Runtime

The expected result by the end of this activity is that when you use the *left mouse button* or the *gamepad right trigger*, a log will appear in the **Output Log**, letting you know that the **ThrowProjectile** function is being called successfully. You will use this function later on to spawn your projectile.

The expected output is as follows:

```
LogInit: FAudioDevice initialized.  
LogLoad: Game class is 'SuperSideScrollerGa  
LogWorld: Bringing World /Game/SideScroller  
LogWorld: Bringing up level for play took:  
LogOnline: OSS: Creating online subsystem i  
PIE: Play in editor start time for /Game/Si  
LogBlueprintUserMessages: Late PlayInEditor  
p.SideScrollerExampleMap_C' with ClassGener  
LogTemp: Warning: THROW PROJECTILE!  
LogTemp: Warning: THROW PROJECTILE!  
LogTemp: Warning: THROW PROJECTILE!
```

Figure 11.33: The expected output log

Note

The solution to this activity can be found at:

<https://packt.live/338jEBx>.

With this activity complete, you now have functionality in-place for when you create the player projectile in *Chapter 13, Enemy Artificial Intelligence*. You also now have the knowledge and experience of adding new key mappings to your game, and implementing functionality in C++ that utilizes these mappings to enable gameplay functionality. Now, you will continue with updating player character movement to allow the jumping animation to play correctly when the player jumps.

Animation State Machines

Now, let's get to know what state machines are in the context of Unreal Engine 4 and in animation. State machines are a means of categorizing an animation, or sets of animations, into their own state. A state can be thought of as a condition that the player character is in at a specific time. Is the player currently walking? Is the player jumping? In many third-

person games such as *The Last of Us*, this is the separation of movement, jumping, crouching, and climbing animations into their own state. Each state is then accessible when certain conditions are met while the game is played. Conditions can include whether the player is jumping, the speed of the player character, and whether or not the player is in the crouched state. The job of the state machine is to transition between each state using logical decisions called **Transition Rules**. When you create multiple states with multiple transition rules that intertwine with one another, the state machine begins to look like a web.

Please refer to the following image to see how the state machine looks for the **ThirdPerson_AnimBP** animation blueprint.

Note

A general overview of state machines can be found here:
<https://docs.unrealengine.com/en-US/Engine/Animation/StateMachineOverview/index.html>

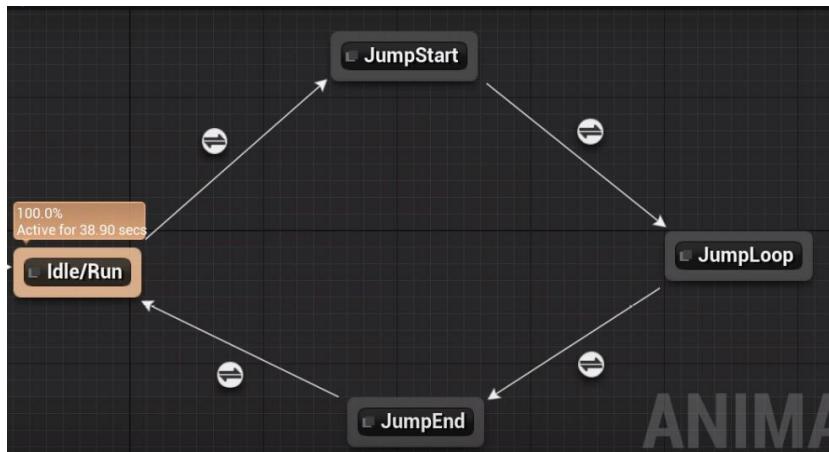


Figure 11.34: The state machine of **ThirdPerson_AnimBP** that is included with the **SideScroller** project template

In the case of the state machine for the player character, this

state machine will handle the states of default player movement and jumping. Currently, you have the player character animating simply by using a Blend Space that is controlled by the speed of the character. What you will do in the next exercise is create a new state machine and move the movement Blend Space logic into its own state within that state machine. Let's begin by creating the new state machine.

Exercise 11.07: Player Character Movement and Jump State Machine

In this exercise, you will implement a new animation state machine and integrate the existing movement Blend Space into the state machine. Additionally, you will set up the states for when the player jump starts, and for when the player is in the air during that jump.

Let's start by adding this new state machine:

1. Navigate to the **/MainCharacter/Blueprints/** directory and open the **AnimBP_SuperSideScroller_MainCharacter** animation blueprint.
2. In **AnimGraph**, *right-click* in the empty space of the graph and search for **state machine** inside the context-sensitive search to find the

Add New State Machine option.

Name this new state machine

Movement.

3. Now, instead of plugging the output pose of the **SideScroller_IdleRun** Blend Space, we can connect the output post of the new state machine, **Movement**, to the output pose of the animation:

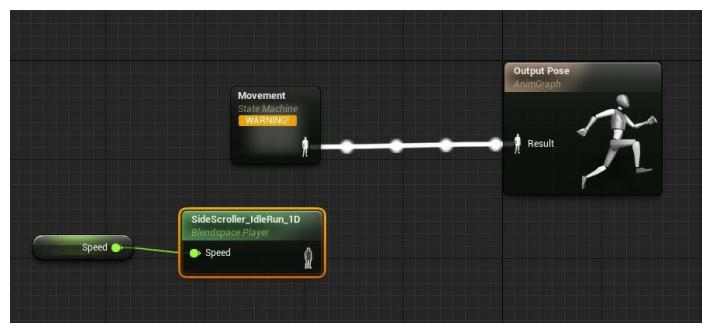


Figure 11.35: The new Movement state machine replaces the old Blend Space

Connecting an empty state machine into the **Output Pose** of the animation blueprint will result in the warnings displayed below. All this means is that there is nothing happening within that state machine and that the result will be invalid to **Output Pose**. Don't worry; you will fix this next.

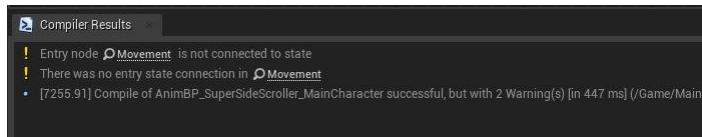


Figure 11.36: The empty state machine results in compile warnings

4. Double left-click on the **Movement** state machine to open the state machine itself. The image below shows what this looks like.



Figure 11.37: This is the empty state machine that is created

You will start by adding a new state that will handle what the character was doing before; **Idle**, **Walking**, and **Running**.

5. From the **Entry** point, *left-click* and drag out to open the context-sensitive search. You will notice that there are only two options – **Add Conduit** and **Add State**. For right now, you will

add a new state and name this state **Movement**. Refer to the following images to see the **Movement** state created.



Figure 11.38: Inside the state machine, you need to add a new state that will handle the movement Blend Space you created earlier

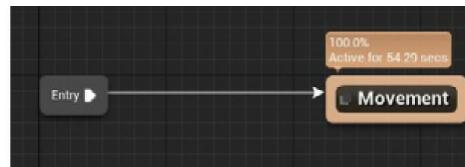


Figure 11.39: The new Movement state

6. Copy and paste the logic you had where you connected the **Speed** variable to the **SideScroller_IdleRun** Blend Space into the new **Movement** state created in the last step. Connect it to the **Result** pin of the **Output Animation Pose** node of this state:

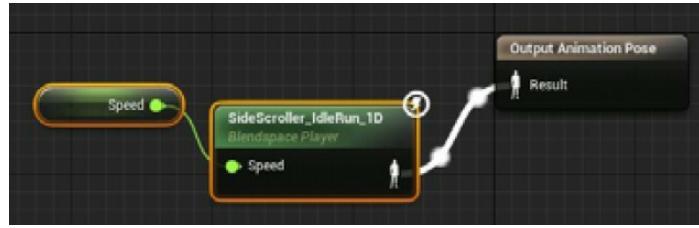


Figure 11.40: Connecting the output pose of the Blend Space to the output pose of this state

Now, if you recompile the animation blueprint, you will notice first that the warnings you saw earlier are now gone. This is because you added a new state that outputs an animation to **Output Animation Pose** instead of having an empty state machine.

By completing this exercise, you have constructed your very first state machine. Although it is a very simple one, you are now telling the character to enter and use the **Movement** state by default. If you now PIE, you will see that the player character is now moving around like he was earlier prior to making the state machine. This means that your state machine is functioning, and you can continue to the next step, which will be adding the initial states required for jumping.

Let's start by creating the **JumpStart** state.

Transition Rules

Conduits are a way of telling each state the conditions under which it can transition from one state to another. In this case, a transition rule is created as a connection between the **Movement** and **JumpStart** states. This is indicated by the directional arrow of the connection between the states again. The tool-tip mentions the term *transition rule*, and this means that you need to define how the transition between these states will happen, using a Boolean value to do so.

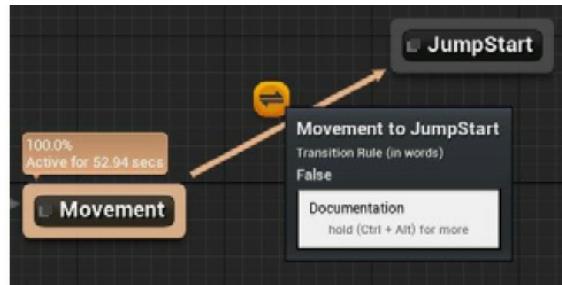


Figure 11.41: There needs to be a transition rule to go from movement to the start of the character's jump

Exercise 11.08: Adding States and Transition Rules to the State Machine

In the case of transitioning from the player character's default movement Blend Space to the beginning of the jump animation, you will need to know when the player decides to jump. This can be done using a useful function called **IsFalling** from the **Character Movement** component of the player character. You will want to track whether the player is currently falling in order to transition in and out of jumping. The best way to do this is to store the result of the **IsFalling** function in its own variable, just like how you did when tracking the player's speed.

The following steps will help you to complete this exercise:

1. Back in the overview of the state machine itself, *left-click* and drag from the edge of the **Movement** state to

open the context-sensitive menu again.

2. Select the option to **Add State** and name this state **JumpStart**. When you do this, Unreal will automatically connect these states and implement an empty **Transition Rule** for you:

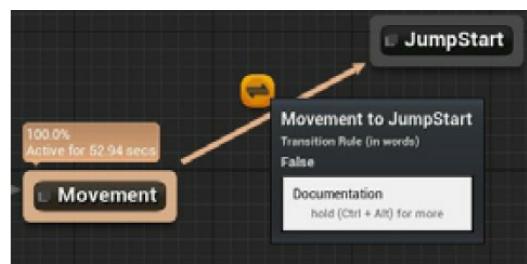


Figure 11.42: The Transition Rule that Unreal automatically creates for you when connecting two states

3. Navigate back to **Event Graph** inside the animation blueprint, where you had used the Event Blueprint update animation event to store the **Speed** of the player character.

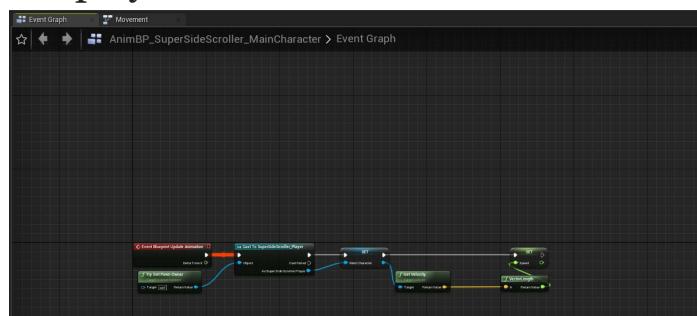


Figure 11.43: EventGraph of the

SuperSideScroller player animation blueprint

4. Create a getter variable for the **MainCharacter** and access the **Character Movement** component.

From the **Character Movement** component, *left-click* and drag to access the context-sensitive menu.

Search for **IsFalling**:

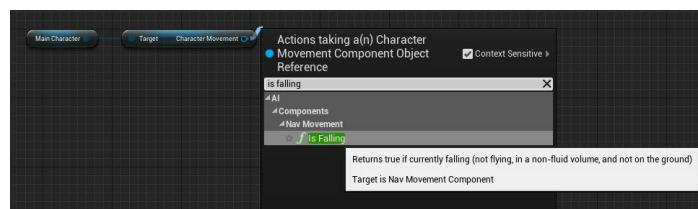


Figure 11.44: How to find the **IsFalling** function

5. The character movement component can tell you whether the player character is currently in the air with the help of the **IsFalling** function:

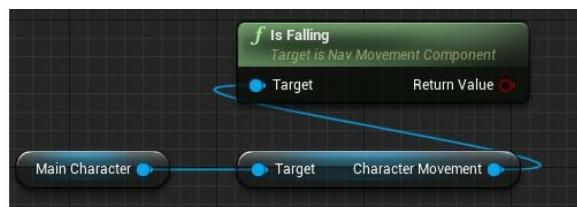


Figure 11.45: The character movement component showing the state of player character

6. From the **Return Value** Boolean of the **IsFalling** function, *left-click* and drag to search for the **Promote to Variable** option from the context-sensitive menu. Name this variable **bIsInAir**. When promoting to a variable, the Return Value output pin should automatically connect to the input pin of the newly promoted variable. If it does not, remember to connect them.

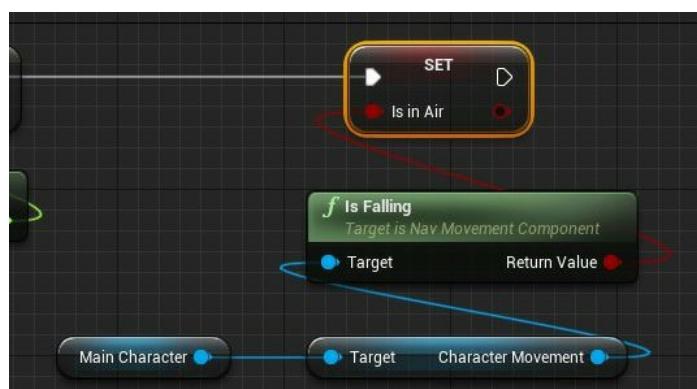


Figure 11.46: A new variable, **bIsInAir, that contains the value of the **IsFalling** function**

Now that you are storing the state of the player and whether or not they are falling, this is the perfect candidate for the transition rule between the **Movement** and **JumpStart** states.

7. In the **Movement State** machine,

double left-click on the **Transition Rule** to enter its graph. You will find only one output node, **Result**, with the parameter **Can Enter Transition**. All you need to do here is use the **bIsInAir** variable and connect it to that output. Now, the **Transition Rule** is saying that if the player is in the air, the transition between the **Movement** state and the **JumpStart** states can happen.

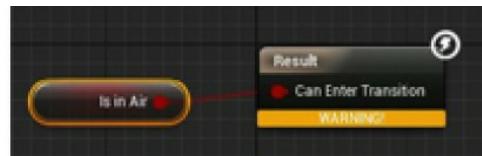


Figure 11.47: When in the air, the player will transition to the start of the jumping animation

With **Transition Rule** in place between the **Movement** and **JumpStart** states ready, all that is left to do is to tell the **JumpStart** state which animation to use.

8. From the state machine graph, *double left-click* on the **JumpStart** state to enter its graph. From **Asset Browser**, *left-click* and drag the

JumpingStart animation to the graph:

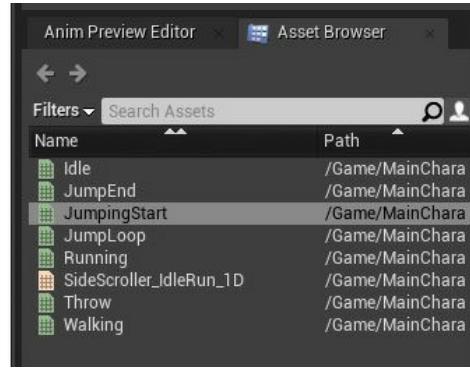


Figure 11.48: Make sure to have the **JumpingStart animation selected in Asset Browser before left-clicking and dragging it into the state**

9. Connect the output of the **Play JumpingStart** node to the **Result** pin of the **Output Animation Pose** node:



Figure 11.49: Connecting the **JumpingStart animation to the **Output Animation Pose** of the **JumpStart** state**

Before you can move forward with the next state, there are settings that need

to be changed on the **JumpingStart** animation node.

10. *Left-click on the **Play** **JumpingStart** animation node and update the **Details** panel to have the following settings:*

1. **Loop Animation =**

False

2. **Play Rate = 2.0**

Please refer to the following figure to see the final settings for the **Play JumpingStart** animation node.

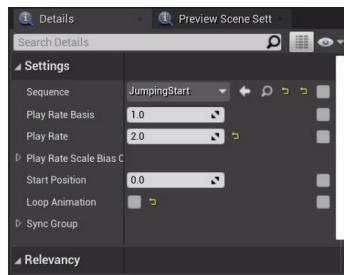


Figure 11.50: Due to the slowness of the JumpStart animation, increasing the play rate will result in a smoother jumping animation overall

You are setting the **Loop Animation** parameter to **False** because there is no reason that this animation should loop; it should only play once in any case. The only way that this animation would loop is if the player character is somehow stuck in this state, but this will never happen because of the

next state you will create. The reason for setting **Play Rate** to **3.0** is because the animation itself, **JumpingStart**, is too long for the purposes of the game you are making. The animation has the character bend their knees drastically, and jump upward over the course of more than a second. For the **JumpStart** state, you want the character to play this animation quicker so that it is more fluid and offers a smoother transition to the next state; the **JumpLoop**.

Once the player character has begun the **JumpStart** animation, there is a point in time during that animation where the player is in the air and should transition to a new state. This new state will loop until the player is no longer in the air and can transition into the final state of ending the jump. Next, let's create this new state that will transition from the **JumpStart** state.

11. From the state machine graph, *left-click* and drag from the **JumpStart** state and select the **Add State** option. Name this new state **JumpLoop**. As before, Unreal will automatically provide you with a **Transition Rule** between these states that you will add to in the next exercise. Finally, recompile the Animation Blueprint and ignore any warnings that may appear under Compiler Results.

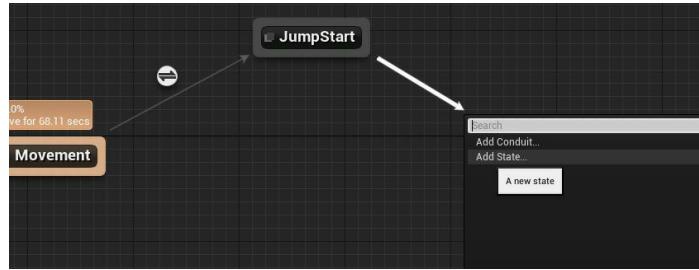


Figure 11.51: Creating another state that will handle the animation of the character while in the air after the initial jump start

By completing this exercise, you have added and connected your own states for **JumpStart** and **JumpLoop**. Each of these states is connected via a **Transition Rule** and you now have a better understanding of how states within a state machine transition from one to another via the rules established in each transition rule.

In the next exercise, you will look more into how to transition from the **JumpStart** state to the **JumpLoop** state via a function, **Time Remaining Ratio**.

Exercise 11.09: Time Remaining Ratio Function

In order for the **JumpStart** state to smoothly transition to the **JumpLoop** state, you need to take a moment to think exactly how you want this transition to work. Based on how the **JumpStart** and **JumpLoop** animations work, it is best to transition in the **JumpLoop** animation after a specified set of time has elapsed on the **JumpStart** animation. That way, the **JumpLoop** state plays smoothly after X seconds of the **JumpStart** animation playing.

Perform the following steps to achieve this:

1. *Double left-click* on the **Transition Rule** between **JumpStart** and **JumpLoop** to open its graph. The **Transition Rule** you will apply here is to check how much time is remaining from the **JumpingStart** animation. This is done because a certain percentage of time remains in the **JumpingStart** animation, and you can safely assume that the player is in the air and is ready to transition to the **JumpingLoop** animation state.
2. To do this, first make sure that the **JumpingStart** animation is selected in **Asset Browser**, and then *right-click* in **Event Graph** of the **Transition Rule** and find the **Time Remaining Ratio** function.

Let's take a moment to talk about the **Time Remaining Ratio** function and what it is doing. This function returns a float between **0.0f** and **1.0f** that tells you how much time is remaining in the specified animation. The values **0.0f** and **1.0f** can directly

be translated to a percentage value so that they are easier to consider. In the case of the **JumpingStart** animation, you would want to know whether less than 60% of the animation is remaining in order to transition successfully to the **JumpingLoop** state. This is what you will do now.

3. From the **Return Value** float output parameter of the **Time Remaining Ratio** function, search for the **Less Than comparative operative** node from the context-sensitive search menu. Since you are working with a returned value between **0.0f** and **1.0f**, in order to know whether less than 60% of the animation remains, you need to compare this returned value with a value of **0.6f**. The final result is as follows:

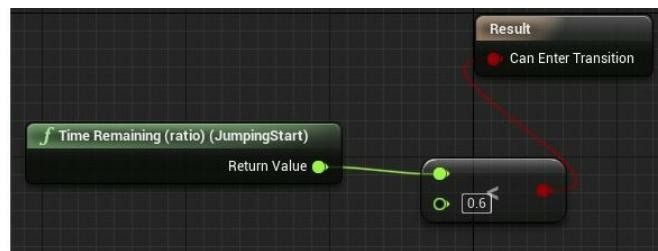


Figure 11.52: You will need to know how much time is left in the

JumpingStart animation before transitioning to the JumpLoop animation

With this **Transition Rule** in place, all that is left to do is to add the **JumpLoop** animation to the **JumpLoop** state.

4. In the **Movement** state machine, *double left-click* on the **JumpLoop** state to enter its graph. With the **JumpLoop** animation asset selected in **Asset Browser**, *left-click* and drag it onto the graph. Connect its output to the **Result** input of **Output Animation Pose**, as shown below. The default settings of the **Play JumpLoop** node will remain unchanged.



Figure 11.53: The JumpLoop animation connected to Output Animation Pose of the new state

With the **JumpLoop** animation in place in the **JumpLoop** state, you can now compile the animation blueprint and PIE. You will notice that the movement and sprinting animations

are still present, but what happens when you try to jump? The player character begins the **JumpStart** state and plays the **JumpLoop** animation while in the air. This is great, the state machine is working, but what happens when the player character reaches the ground and is no longer in the air? The player character does not transition back to the **Movement** state, which makes sense because you have not yet added the state for **JumpEnd** or the transitions between **JumpLoop** and **JumpEnd**, and from **JumpEnd** back to the **Movement** state. You will do this in the next activity. See below for an example of a player character stuck in the **JumpLoop** state:

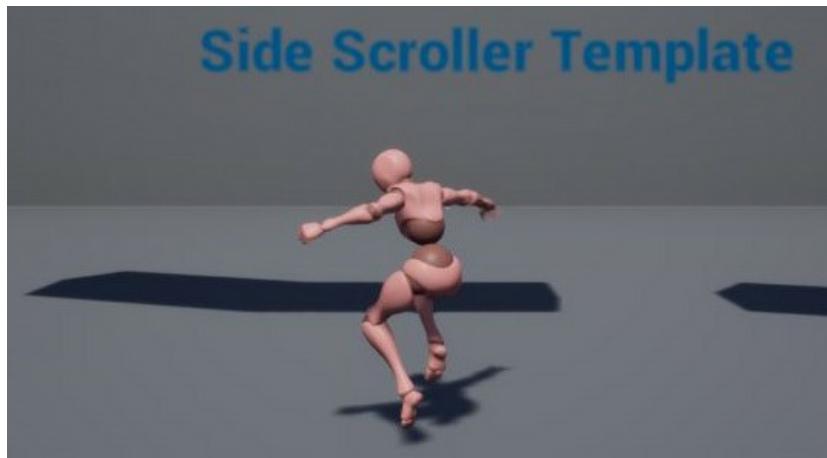


Figure 11.54: The player character can now play the **JumpingStart** animation and the **JumpLoop** animation, but cannot transition back to the default movement

By completing this exercise, you were able to successfully transition from the **JumpStart** state to the **JumpLoop** state by using the **Time Remaining Ratio** function. This function allows you to know how far along an animation has played, and with this information, you had the state machine transition into the **JumpLoop** state. The player can now successfully transition from the default **Movement** state to the **JumpStart** state and then to the **JumpLoop** state, resulting in an interesting issue. The player is now stuck in the **JumpLoop** state because the state machine does not contain

the transition back to the **Movement** state. Let's fix this in the next activity.

Activity 11.04: Finishing the Movement and Jumping State Machine

With half of the state machine completed, it's time to add the state for when the jump ends, as well as the Transition Rules that allow you to transition from the **JumpLoop** state to this new state, and the transition from this new state back to the **Movement** state.

Do the following to complete the **Movement** state machine:

1. Add a new state for **Jump End** that transitions from **JumpLoop**. Name this state **JumpEnd**.
2. Add the **JumpEnd** animation to the new **JumpEnd** state.
3. Based on the **JumpEnd** animation and how quickly we want to transition between the **JumpLoop**, **JumpEnd**, and **Movement** states, consider modifying the parameters of the animation like you did for the **JumpStart** animation. The **loop animation** parameter

needs to be **False** and the **Play Rate** parameter needs to be set to **3.0**.

4. Add a **Transition Rule** from the **JumpLoop** state to the **JumpEnd** state based on the **bIsInAir** variable.
5. Add a **Transition Rule** from the **JumpEnd** state to the **Movement** state based on the **Time Remaining Ratio** function of the **JumpEnd** animation. (Look at the **JumpStart** to **JumpLoop** Transition Rule).

By the end of this activity, you will have a fully functioning movement state machine that allows the player character to idle, walk, and sprint, as well as being able to jump and animate correctly at the start of the jump, while in the air, and when landing.

The expected output is as follows:



Figure 11.55: Player character with idle, walk, sprint, and jump animation

Note

The solution to this activity can be found at:

<https://packt.live/338jEBx>.

By completing this activity, you have now finished the movement state machine for the player character. By adding the remaining **JumpEnd** state and the **Transition Rules** to transition to the state from the **JumpLoop** state, and to transition from the **JumpEnd** state back to the **Movement** state, you successfully created your first animation state machine. Now, you can run around the map and jump onto elevated platforms, all while animating correctly and transitioning between movement and jump states.

Summary

With the player movement Blend Space created and the player

character animation blueprint using a state machine to transition from movement to jumping, you are ready to move on to the next chapter, where you will prepare the required animation slot, animation montage, and update the animation blueprint for the throw animation that will use only the upper body of the character.

From the exercises and activities in this chapter, you learned how to create a 1D Blend Space that allows the smooth blending of movement-based animations such as idling, walking, and running using the speed of the player character to control the blending of animations.

Additionally, you learned how to integrate new key bindings into the project settings and bind those keys in C++ to enable character gameplay mechanics such as sprinting and throwing.

Lastly, you learned how to implement your very own animation state machine within the character animation blueprint in order for the player to have the ability to transition between movement animations, to the various states of jumping, and back to movement again. With all of this logic in place, in the next chapter let's move on and create the assets and logic for allowing the player character to play the throwing animation, and set up the base class for the enemy.

12. Animation Blending and Montages

Overview

*By the end of this chapter, you will be able to use the **Animation Montage** tool to create a unique throwing animation using the **Throw** animation sequence you imported in Chapter 10, Creating a SuperSideScroller Game. With this montage, you will create and use Anim Slots that will allow you to blend animations in the Animation Blueprint for the player character. You will also get to know how to use blending nodes to effectively blend the movement and throwing animations of the character.*

*After finalizing the player character animation, you will create the required class and assets for the enemy AI and learn more about Materials and **Material Instances**, which will give this enemy a unique visual color so that it can be differentiated in-game. Finally, the enemy will be ready for Chapter 13, Enemy Artificial Intelligence, where you will begin to create the AI behavior logic.*

Introduction

In the last chapter, you were able to bring the player character to life by implementing movement animations in a **Blend Space** and using that **Blend Space** in an Animation Blueprint to drive the animations based on the player's speed. You were then able to implement functionality in C++ based

on player input to allow the character to sprint. Lastly, you took advantage of the animation state machine built-in Animation Blueprints to drive the character movement state and jumping states to allow fluid transitions between walking and jumping.

With the player character Animation Blueprint and state machine working, it's time to introduce Animation Montages and Anim Slots by implementing the character's **Throw** animation. In this chapter, you will learn more about animation blending, see how Unreal Engine handles the blending of multiple animations by creating an **Animation Montage**, and work with a new **Anim Slot** for the player's throwing animation. From there, you will use the Anim Slot in the player's Animation Blueprint by implementing new functions such as **Save Cached Pose** and **Layered blend per bone** so that the player can correctly blend the movement animations you handled in the previous chapter with the new throwing animation you will implement in this chapter.

Let's start by learning about what Animation Montages and Anim Slots are and how they can be used for character animation.

Animation Blending, Anim Slots, and Animation Montages

Animation blending is the process of transitioning between multiple animations on a skeletal mesh as seamlessly as possible. You are already familiar with the techniques of animation blending because you created a **Blend Spaces** asset for the player character in *Chapter 11, Blend Spaces 1D*,

Key Bindings, and State Machines. In this **Blend Space**, the character smoothly blends between the **Idle**, **Walking**, and **Running** animations. You will now extend this knowledge by exploring and implementing new additive techniques to combine the movement animations of the character with a throwing animation. Through the use of an **Anim Slot**, you will send the throwing animation to a set of upper body bones, and its children's bones, to allow movement and throwing animations to apply at the same time without negatively impacting the other. But first, let's talk more about Animation Montages.

Animation Montages are a very powerful asset that allows you to combine multiple animations and split these combined animations into what is called **Sections**. Sections can then be played back individually, in a specific sequence, or even looped.

Animation Montages are also useful because you can control animations through montages from Blueprints or C++; this means you can call logic, update variables, replicate data, and so on based on the animation section being played, or if any **Notifies** are called within the montage. In C++, there is the **UAnimInstance** object, which you can use to call functions such as **UAnimInstance::Montage_Play**, which allows you to access and play montages from C++.

Note

This method will be used in Chapter 14, Spawning the Player Projectile, when you begin to add polish to the game. More information about how animations and Notifies are handled by Unreal Engine 4 in C++ can be found at <https://docs.unrealengine.com/en-US/API/Runtime/Engine/Animation/AnimNotifies/UAnimNotifyState/index.html>.

You will learn more about Notifies in the first exercise of

this chapter, and you will code your own **notify** state in Chapter 14, Spawning the Player Projectile.

The image below shows the **Persona** editor for Animation Montages. However, this will be broken down even further in *Exercise 12.01, Setting Up the Animation Montage*:



Figure 12.1: The Persona editor, which opens when editing an Animation Montage

Just like in Animation Sequences, Animation Montages allow **Notifies** to be triggered along the timeline of a section of an animation, which can then trigger sounds, particle effects, and events. **Event Notifies** will allow us to call logic from Blueprint or C++. Epic Games provides an example in their documentation of a weapon reload **Animation Montage** that is split between animations for **reload start**, **reload loop**, and **reload complete**. By splitting these animations and applying **Notifies** for **sounds** and **events**, developers have complete control over how long the **reload loop** will play based on internal variables, and control over any additional sounds or effects to play during the course of the animation.

Lastly, Animation Montages support what are called **Anim Slots**. Anim Slots allow you to categorize an animation, or a set of animations, that can later be referenced in Animation Blueprints to allow unique blending behavior based on the slot. This means that you can define an Anim Slot that can later be used in Animation Blueprints to allow animations

using this slot to blend on top of the base movement animations in any way you want; in our case, only affecting the upper body of the player character and not the lower body.

Let's begin by creating the **Animation Montage** for the player character's **Throw** animation in the first exercise.

Exercise 12.01: Setting Up the Animation Montage

One of the last things you need to do for the player character is to set up the Anim Slot that will separately categorize this animation as an upper body animation. You will use this Anim Slot in conjunction with blending functions in the Animation Blueprint to allow the player character to throw a projectile, while still correctly animating the lower body while moving and jumping.

By the end of this exercise, the player character will be able to play the **Throw** animation only with their upper body, while their lower body will still use the **movement animation** that you defined in the previous chapter.

Let's begin by creating the **Animation Montage** for the character, throwing and setting up the Anim Slot there:

1. First, navigate to the **/MainCharacter/Animation** directory, which is where all of the animation assets are located.
2. Now, *right-click* in the content browser

and hover over the **Animation** option from the available drop-down menu.

3. Then, *left-click* to select the **Animation Montage** option from the additional drop-down menu that appears.
4. Just as with creating other animation-based assets, such as **Blend Spaces** or **Animation Blueprints**, Unreal Engine will ask you to assign a **Skeleton** object for this **Animation Montage**. In this case, select **MainCharacter_Skeleton**.
5. Name the new **Animation Montage** **AM_Throw**. Now, *double-left-click* to open the montage:

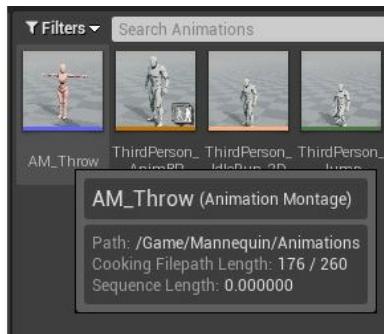


Figure 12.2: You have successfully created an Animation Montage asset

When you open the **Animation Montage** asset, you are

presented with a similar editor layout, as you would when opening an **Animation Sequence**. There is a **Preview** window that shows the main character skeleton in the default T pose, but once you add animations to this montage, the skeleton will update to reflect those changes.

With this exercise complete, you have successfully created an **Animation Montage** asset for the **Super SideScroller** project. Now it is time to learn more about Animation Montages and how you can add the **Throw** animation and Anim Slot you need in order to blend the **Throw** animation with the existing character movement animations.

Animation Montages

Have a look at the following figure:

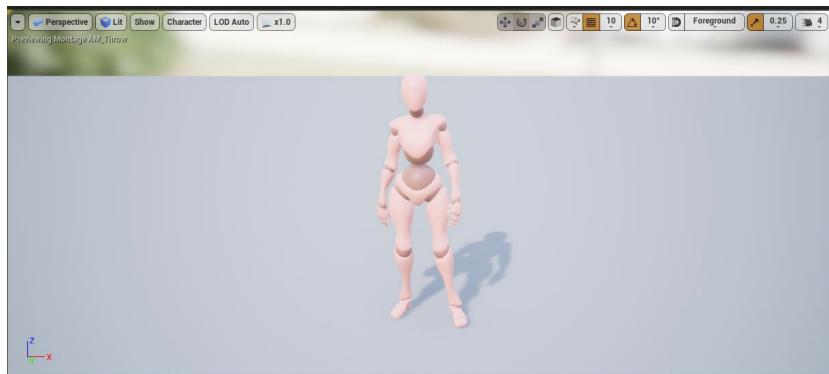


Figure 12.3: The animation Preview window in the Animation Montage Persona editor

Underneath the **Preview** window, you have the main montage timeline, in addition to other sections; let's evaluate these sections from top to bottom:

- **Montage:** The **Montage** section is a collection of animations that can add

one or more animations to. You can also *right-click* on any point in the timeline to create a **section**. Sections allow you to compartmentalize the different parts of the montage into their own self-contained section that can be referenced and manipulated in the **Sections** area.

- **Sections:** Sections, as mentioned before, allow you to set the order of how the individual animation sequences are played and whether a section should loop.

For the purposes of the throw montage, you do not need to use this feature since you will only be using one animation in this montage:

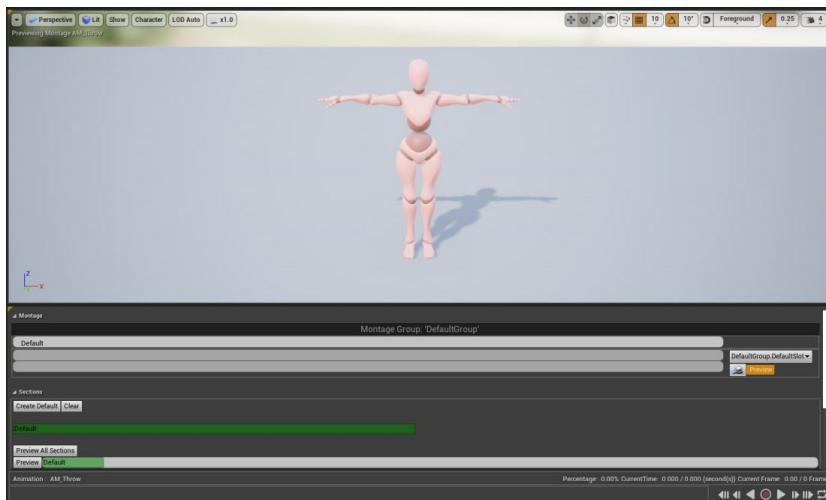


Figure 12.4: The Preview window and the Montage and Sections areas

- **Element Timing:** The **Elemental Timing** section gives you a preview of the montage and the sequential order of the varied aspects of the montage. The playback order of **Notifies**, the **Montage** section, and other elements will be visually displayed here to give you a quick preview of how the montage will work.
- **Notifies:** **Notifies** gives you the ability to add points to an animation time frame that can then notify other systems to perform an action or to call logic from both Blueprints and C++. Notify options, such as **Play Sound** or **Play Particle Effect**, allow you to play a sound or particle at a specific time in the animation. One example is during a reload animation of a weapon; you can add a notify to the timeline of the animation to play a reload sound at the precise moment of reloading. You will use these **Notifies** later on in this project when you implement the throwing projectile:

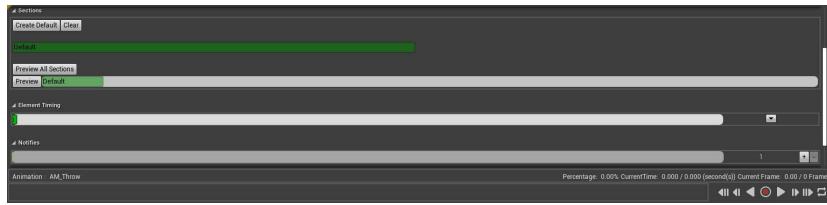


Figure 12.5: The Element Timing and Notifies areas

Now that you are familiar with the interface for Animation Montages, you can add the **Throw** animation to the montage by following the next exercise.

Exercise 12.02: Adding the Throw Animation to the Montage

Now that you have a better understanding of what Animation Montages are and how these assets work, it is time to add the **Throw** animation to the montage you created in *Exercise 12.01, Setting Up the Animation Montage*. Although you will only be adding one animation to this montage, it is important to emphasize that you can add multiple unique animations to a montage that you can then play back. Now, let's start by adding the **Throw** animation you imported into the project in *Chapter 10, Creating a SuperSideScroller Game*:

In **Asset Browser**, find the **Throw** animation asset. Then, *left-click* and drag it onto the timeline under the **Montage** section:

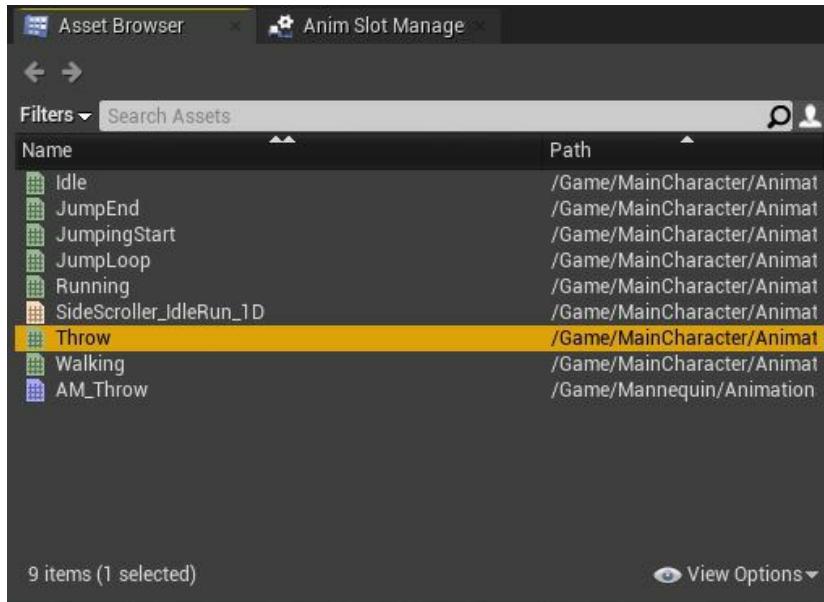


Figure 12.6: The Asset Browser window with animation-based assets

Once an animation is added to the Animation Montage, the character skeleton in the **Preview** window will update to reflect this change and begin playing the animation:

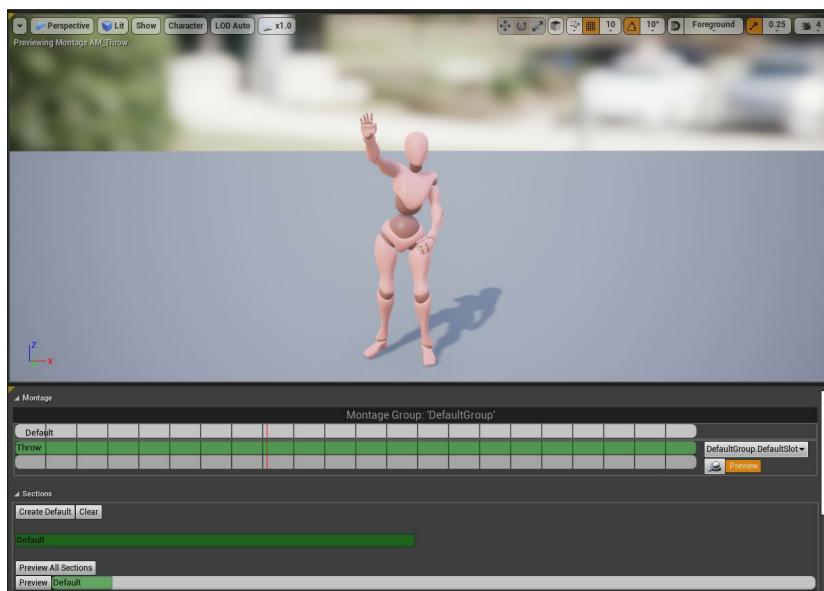


Figure 12.7: The player character begins to animate

Now that the **Throw** animation has been added to the

Animation Montage, you can move on to create the **Anim Slot**.

The **Anim Slot Manager** tab should be docked next to the **Asset Browser** tab on the right-hand side. If you don't see the **Anim Slot Manager** tab, you can access it by navigating to the **Window** tab in the toolbar at the top of the **Animation Montage** editor window. There, *left-click* to select the option for **Anim Slot Manager** and the window will appear.

With this exercise complete, you have added the **Throw** animation to your new Animation Montage and you were able to play back the animation to preview how it looks in the editor through **Persona**.

Now, you can move on to learn more about Anim Slots and **Anim Slot Manager** before adding your own unique Anim Slot to use for animation blending later in this chapter.

Anim Slot Manager

Anim Slot Manager is where you, as the name suggests, manage your **Anim Slots**. From this tab, you can create new **Groups**, which allow greater organization of your slots. For example, you can create a **Group** by *left-clicking* on the **Add Group** option and labeling it **Face** to articulate to others that the slots within this group affect the face of the character. By default, Unreal Engine provides you with a **Group** called **DefaultGroup** and an **Anim Slot** called **DefaultSlot** that is in that group.

Let's create a new Anim Slot.

Exercise 12.03: Adding a New Anim Slot

Now that you have a better understanding of Anim Slots and **Anim Slot Manager**, you can follow these steps to create a new Anim Slot, which you will call **Upper Body**. Once you have this new slot created, it can then be used and referenced in your Animation Blueprint to handle animation blending, which you will do in a later exercise.

Let's create the Anim Slot by doing the following:

1. In **Anim Slot Manager**, *left-click* on the **Add Slot** option.
2. When adding a new slot, Unreal will ask you to give this **Anim Slot** a name. Name this slot **Upper Body**.
Anim Slot naming is important, much like naming any other assets and parameters, because you will be referencing this slot in the Animation Blueprint later.

With the Anim Slot created, you can now update the slot used for the **Throw** montage.
3. In the **Montage** section, there is a drop-down menu that displays the applied **Anim Slot**; by default, it's set

to **DefaultGroup.DefaultSlot**.
Left-click, and from the drop-down menu, select **DefaultGroup.Upper Body**:

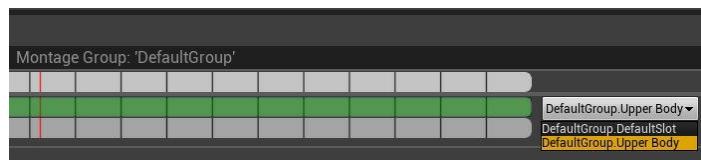


Figure 12.8: The new Anim Slot will appear in the drop-down list

Note

*After changing **Anim Slot**, you may notice that the player character stops animating and returns to the T pose. Don't worry – if this happens, just close the **Animation Montage** and reopen it. Once reopened, the character will play the **Throw** animation again.*

With your **Anim Slot** created and in place in the **Throw** montage, it is now time for you to update the Animation Blueprint so that the player character is aware of this slot and animates correctly based on it.

4. Navigate to the

AnimBP_SuperSideScroller_MainCharacter asset in the **/MainCharacter/Blueprints/** directory.

5. Open this asset by *double-left-clicking* and opening **Anim Graph**.

With this exercise complete, you have created your first Anim Slot using **Anim Slot Manager**, available in the Animation Montage. With this slot in place, it can now be used and referenced in the player character Animation Blueprint to handle the animation blending required to blend the **Throw** animation and the movement animations you implemented in the previous chapter. Before you do this, you need to learn more about the **Save Cached Pose** node in Animation Blueprints.

Save Cached Pose

There are cases when working with complex animations and characters requires you to reference a pose that is outputted by a state machine in more than one place. If you hadn't noticed already, the output pose from your **Movement** state machine cannot be connected to more than one other node. This is where the **Save Cached Pose** node comes in handy; it allows you to cache, or store, a pose that can then be referenced in multiple places at once. You will need to use this to set up the new Anim Slot for the upper body animation.

Let's get started.

Exercise 12.04: Save Cached Pose of the Movement State Machine

To effectively blend the **Throw** animation, which uses the **Upper Body Anim Slot** you created in the previous exercise with the movement animations already in place for the player character, you need to be able to reference the **Movement** state machine in the Animation Blueprint. To do this, do the following to implement the **Save Cached Pose** node in the Animation Blueprint:

1. In **Anim Graph**, right-click and search for **New Save Cached Pose**. Name this **Movement Cache**:

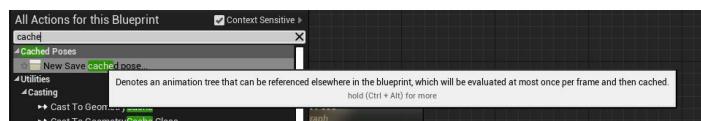


Figure 12.9: The Pose will be evaluated once per frame and then cached

2. Now, instead of connecting your **Movement** state machine directly to the output pose, connect it to the cache node:



Figure 12.10: The Movement state machine is being cached

3. With the **Movement** state machine pose being cached, all you have to do now is reference it. This can be done by searching for the **Use Cached Pose** node.

Note

All cached poses will show in the context-sensitive menu. Just make sure you select the cached pose with the name you gave it in Step 1.

4. With the cached pose node available, connect it to **Output Pose** of the **AnimGraph**:

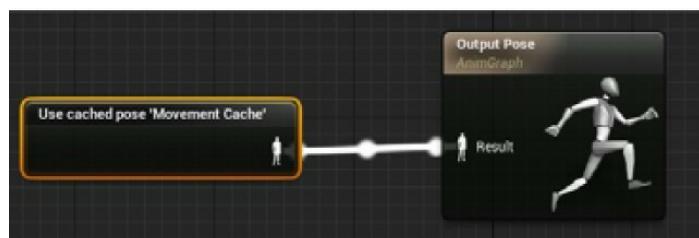


Figure 12.11: This is identical to having the Movement state machine directly connected to Output Pose

You will notice now, after *Step 4*, that the main character will animate correctly and move as you expect after the last chapter. This proves that the caching of the **Movement** state machine is working. The image below shows the player

character back in his **Idle** animation in the preview window of the Animation Blueprint.

Now that you have the caching of the **Movement** state machine working, you will use this cache to blend animations through the skeleton based on the **Anim Slot** you created:



Figure 12.12: The main character is animating as expected

With this exercise complete, you now have the ability to reference the cached **Movement** state machine pose anywhere you would like within the Animation Blueprint. With this accessibility in place, you can now use the cached pose to begin the blending between the cached movement pose and the **Upper Body** Anim Slot using a function called **Layered blend per bone**.

Layered blend per bone

The node that you will use to blend animations here is **Layered blend per bone**. This node masks out a set of bones on the character skeleton for an animation to ignore those bones.

In the case of our player character and the **Throw** animation, you will mask out the lower body so that only the upper body animates. The goal is to be able to perform the throw and movement animations at the same time and have these animations blend together; otherwise, when you perform the throw, the movement animations would completely break.

Exercise 12.05: Blending Animation with the Upper Body Anim Slot

The **Layered blend per bone** function allows us to blend the **Throw** animation with the movement animations you implemented in the previous chapter, and give you control over how much influence the **Throw** animation will have on the player character skeleton.

In this exercise, you will use the **Layered blend per bone** function to completely mask out the lower body of the character when playing the **Throw** animation so that it does not influence the character movement animation of the lower body.

Let's begin by adding the **Layered blend per bone** node and discuss its input parameters and its settings:

1. Inside the Animation Blueprint, *right-click* and search for **Layered blend per bone** in the **Context Sensitive** search.

Figure 12.13 shows the **Layered**

blend per bone node and its parameters.

1. The first parameter, **Base Pose**, is for the base pose of the character; in this case, the cached pose of the **Movement** state machine will be the base pose.
2. The second parameter is the **Blend Pose 0** node that you want to layer on top of **Base Pose**; keep in mind that selecting **Add Pin** will create additional **Blend Pose** and **Blend Weights** parameters. For now, you will only be working with one **Blend Pose** node.
3. The last parameter is **Blend Weights**, which is how much **Blend Pose** will affect **Base Pose** on a scale from **0 . 0** to **1 . 0** as an alpha:



Figure 12.13: The Layered blend per bone node

Before you connect anything to this node, you will need to add a layer to its properties.

2. *Left-click* to select the node and navigate to **Details**. You will need to *left-click* on the arrow next to **Layer Setup** to find the first index, **0**, of this setup. *Left-click* on **+** next to **Branch Filters** to create a new filter.

There are again two parameters here, namely the following:

1. **Bone Name:** The bone to specify where the blending will take place and determine the child hierarchy of bones masked out. In the case of the main character skeleton for this project, set **Bone Name** to **Spine**. *Figure 12.14* shows how the **Spine** bone and its children are unassociated with the lower body of the main character. This can be seen in the **Skeleton** asset, **MainCharacter_Skeleton**:
:

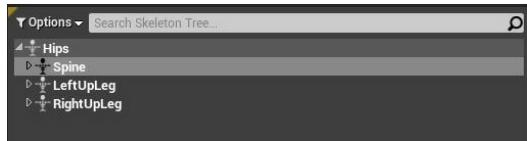


Figure 12.14: The Spine bone and its children are associated with the upper body of the main character

- **Blend Depth:** The depth in which bones and their children will be affected by the animation. A value of **0** will not affect the rooted children of the selected bone.
- **Mesh Space Rotation Blend:** Determines whether or not to blend bone rotations in **mesh space** or in **local space**. **Mesh Space** rotation refers to the skeletal mesh's bounding box as its base rotation, while **Local Space** rotation refers to the local rotation of the bone name in question. In this case, we want the rotation blend to occur in mesh space, so we will set this parameter to true.

Blending is propagated to all the children of a bone to stop blending on particular bones, add them to the array, and make their blend depth value **0**. The final result is as follows:

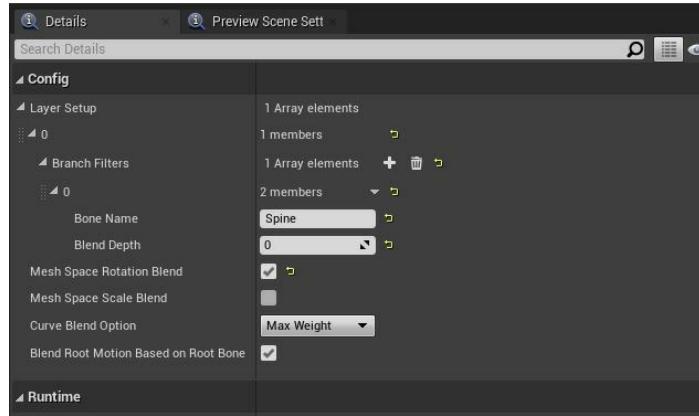


Figure 12.15: You can set up multiple layers with one blend node

3. With the settings in place on the **Layered blend per bone** node, you can connect the **Movement Cache** cached pose into the **Base Pose** node of the layered blend. Make sure you connect the output of the **Layered blend per bone** node to **Output Pose** of the Animation Blueprint:

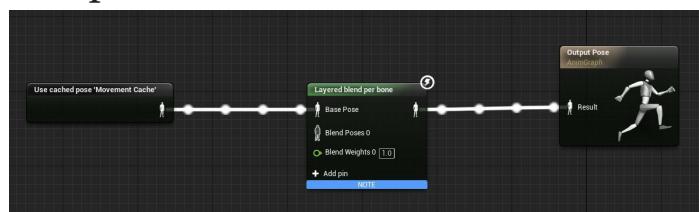


Figure 12.16: Add the cached pose for the Movement state machine to the Layered blend per bone node

Now it's time to use the Anim Slot you created earlier to filter only the animations using this slot through the **Layered blend per bone** node.

4. *Right-click* in the **AnimGraph** and search for **DefaultSlot**. *Left-click* to select the **Slot** node and navigate to **Details**. There, you will find the **Slot Name** property. *Left-click* on this drop-down to find and select the **DefaultGroup.Upper Body** slot.

When changing the **Slot Name** property, the **Slot** node will update to represent this new name. The **Slot** node requires a source pose, which will again be a reference to the **Movement** state machine. This means that you need to create another **Use Cached Pose** node for the **Movement Cache** pose.

5. Connect the cached pose into the source of the **Slot** node:

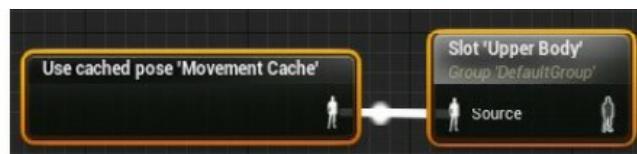


Figure 12.17: Filtering the cached Movement pose through the Anim Slot

6. All that is left to do now is connect the **Upper Body** slot node to the **Blend Pose 0** input. Then, connect the final pose of **Layered blend per bone** to the result of the **Output Pose** Animation Blueprint:

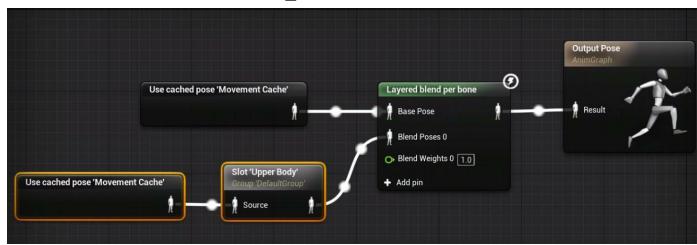


Figure 12.18: The final setup of the main character's Animation Blueprint

With the Anim Slot and the **Layered blend per bone** node in place within the main character's Animation Blueprint, you are finally done with the animation side of the main character.

Next, let's have a brief discussion about the importance of animation blending for the **Throw** animation and what the **Throw** animation will be used for, before you move on to *Exercise 12.06, Previewing the Throw Animation*, where you will preview the **Throw** animation in the game.

The Throw Animation

So far, you have put a lot of work into ensuring that the **Throw** animation blends correctly with the **Movement** animations that you set up in the Animation Blueprint in the previous chapter. The main reason behind this effort is to ensure the visual fidelity of the character when performing multiple animations at once. You will learn first-hand the visual consequences of incorrectly setting up animation blending in the exercises and activity ahead.

Getting back to the **Throw** animation, every modern video game implements animation blending in one form or another, so long as the art direction and the game mechanics require such a feature. An example of a modern game franchise that extraordinarily uses animations is the *Uncharted* series developed by *Naughty Dog*.

If you are unfamiliar with the franchise, you can watch the full gameplay of the latest installment here:
https://www.youtube.com/watch?v=5evF_funE8A.

What the *Uncharted* series does very well is use thousands of animations and blending techniques to give the player character an incredible sense of *realism*, *weight*, and *movement* that feels really good while you play the game. Although the **Super SideScroller** project will not be anywhere as polished as this, you are learning the basics of what is needed to make incredible animations for video games:

Exercise 12.06: Previewing the Throw Animation

In the previous exercise, you did a lot of work to allow animation blending between the player character's **Movement**

animations and the **Throw** animation by using the **Save Cached Pose** and **Layered blend per bone** nodes. Perform the following steps to preview the **Throw** animation in-game and see the fruits of your labor:

1. Navigate to the **/MainCharacter/Blueprints/** directory and open the character's **BP_SuperSideScroller_MainCharacter Blueprint**.
2. If you recall, in the last chapter you created **Input Action** for throwing with the **ThrowProjectile** name.
3. Inside **Event Graph** of the character's Blueprint, *right-click* and search for **ThrowProjectile** in the **Context Sensitive** drop-down search. Select it with a *left-click* to create the event node in the graph.

With this event in place, you need a function that allows you to play an **Animation Montage** when the player uses the *left mouse button* to throw.
4. *Right-click* in **Event Graph** and search for **Play Montage**. Make sure

not to confuse this with a similar function **Play Anim Montage**.

The **Play Montage** function requires two important inputs:

1. **Montage to Play**
2. **In Skeletal Mesh Component**

Let's first handle **Skeletal Mesh Component**.

5. The player character has a **Skeletal Mesh Component** that can be found in the Components tab labeled **Mesh**. Left-click and drag out a **Get** reference to this variable and connect it to the **In Skeletal Mesh Component** input of this function:

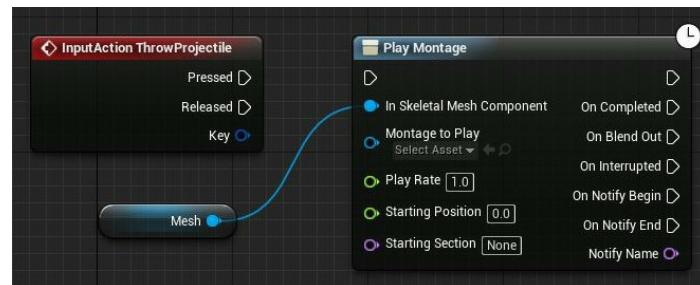


Figure 12.19: The mesh of the player character connected to the In Skeletal Mesh Component

input

The last thing to do now is to tell this function which montage to play. Luckily for you, there is only one montage that exists in this project: **AM_Throw**.

6. *Left-click* on the drop-down menu under the **Montage to Play** input and *left-click* to select **AM_Throw**.
7. Finally, connect the **Pressed** execution output of the **ThrowProjectile** event to the execution input pin of the **Play Montage** function:

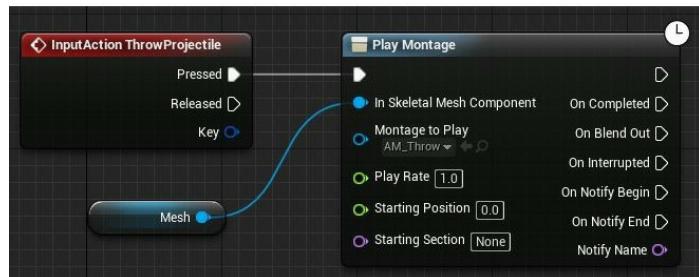


Figure 12.20: When the player presses the ThrowProjectile input actions, the AM_Throw montage will be played

8. Now, when you click your *left mouse button*, the player character will play

the throwing **Animation Montage**.

Notice now how you can walk and run at the same time as throwing, and each animation blends together so as to not interfere with one another:

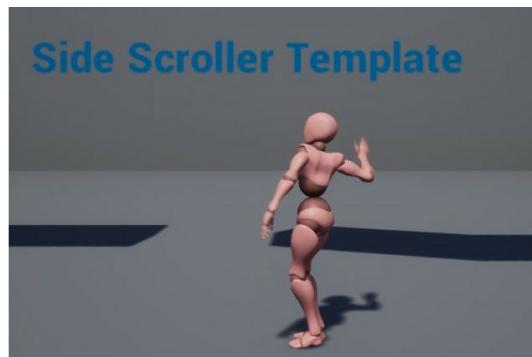


Figure 12.21: The player character can now move and throw

Don't worry about any bugs you might see when using the *left mouse button* action repeatedly to play the **Throw** montage; these issues will be addressed when you implement the projectile that will be thrown in a later chapter for this project. For now, you just want to know that the work done on the **Anim Slot** and the **Animation Blueprint** give the desired result for animation blending.

Let's continue with the **SuperSideScroller** project by now creating the C++ class, the Blueprints, and the Materials necessary to set up the enemy for use in the next chapter.

The Super Side Scroller Game Enemy

With the player character animating correctly when moving

and performing the **Throw** animation, it is time to talk about the enemy type that the **SuperSideScroller** game will feature. We will have a simple type of enemy.

This enemy will have a basic back-and-forth movement pattern and will not support any attacks; only by colliding with the player character will it be able to inflict damage.

In the next exercise, you will set up the base enemy class in C++ for the first enemy type and configure the enemy's Blueprint and Animation Blueprint in preparation for *Chapter 13, Enemy Artificial Intelligence*, where you will implement the AI of this enemy. For the sake of efficiency and time, you will use the assets already provided by Unreal Engine 4 in the **SideScroller** template for the enemy. This means you will be using the skeleton, skeletal mesh, animations, and the Animation Blueprint of the default mannequin asset. Let's begin by creating the first enemy class.

Exercise 12.07: Creating the Enemy Base C++ Class

The goal of this exercise is to create a new enemy class from scratch and to have the enemy ready to use in *Chapter 13, Enemy Artificial Intelligence*, when you develop the AI. To start, create a new enemy class in C++ by following these steps:

1. In the editor, navigate to **File** and select **New C++ Class** to get started with creating your new enemy class.

2. Next, make sure you check the **Show All Classes** box toward the top of the **Choose Parent Class** window prompt before attempting to search for a class. Then, search for **SuperSideScrollerCharacter** and *left-click* it to select it as the parent class.
3. Lastly, you need to give this class a name and select a directory. Name this class **EnemyBase** and do not change the directory path. When ready, *left-click* on the **Create Class** button to have Unreal Engine create the new class for you.

When you create a new class, Unreal Engine will automatically open Visual Studio for you with the **.cpp** and **.h** files ready to go. For now, you will not make any changes to the code, so close Visual Studio.

Let's create the folder structure in the content browser for the enemy assets next.

4. Head back to the Unreal Engine 4 editor, navigate to the content browser,

and create a new folder called **Enemy**:

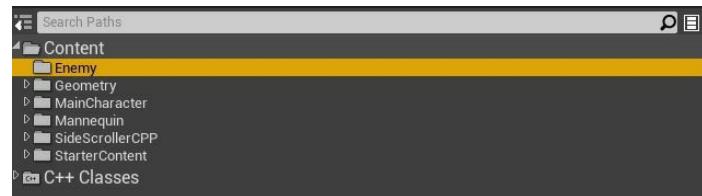


Figure 12.22: New folders are created by right-clicking on existing folders and selecting New Folder

5. In the **Enemy** folder, create another folder called **Blueprints**, where you will create and save the Blueprint assets for the enemy.
6. In the **/Enemy/Blueprints** directory, *right-click* and select **Blueprint Class**. From **Pick Parent Class**, search for the new C++ class you just made, **EnemyBase**, as shown:

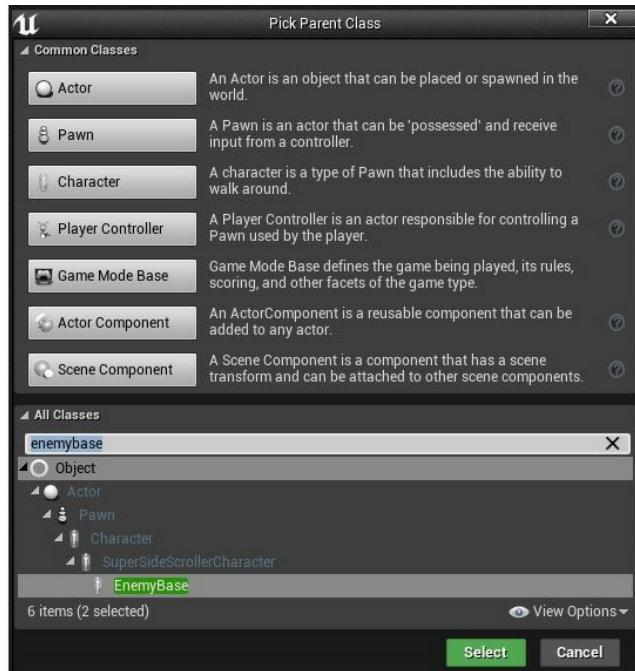


Figure 12.23: Now, the new **EnemyBase** class is available for you to create a Blueprint from

7. Name this **BP_Enemy**.

Now that you have the **Blueprint** for the first enemy using the **EnemyBase** class as the parent class, it is time to handle the **Animation Blueprint**. You will use the default **Animation Blueprint** that is provided to you by Unreal Engine in the **SideScroller** template project. Follow the steps in the next exercise to create a duplicate of the existing **Animation Blueprint** and move it to the **/Enemy/Blueprints** directory.

Exercise 12.08: Creating

and Applying the Enemy Animation Blueprint

In the previous exercise, you created a **Blueprint** for the first enemy using the **EnemyBase** class as the parent class. In this exercise, you will be working with the Animation blueprint.

The following steps will help you complete this exercise:

1. Navigate to the **/Mannequin/Animations** directory and find the **ThirdPerson_AnimBP** asset.
2. Now, duplicate the **ThirdPerson_AnimBP** asset. There are two ways to duplicate an asset:
 1. Select the desired asset in the content browser and press **CTRL + W**.
 2. *Right-click* on the desired asset in the content browser and select **Duplicate** from the drop-down menu.
3. Now, *left-click* and drag this duplicate asset into the **/Enemy/Blueprints** directory and select the option to move

when you release the *left-click* mouse button.

4. Name this duplicate asset **AnimBP_Enemy**. It is best to create a duplicate of an asset that you can later modify if you so desire without risking the functionality of the original:

With the enemy **Blueprint** and **Animation Blueprint** created, it's time to update the enemy Blueprint to use the default **Skeletal Mesh** mannequin and the new **Animation Blueprint** duplicate.
5. Navigate to **/Enemy/Blueprints** and open **BP_Enemy**.
6. Next, navigate to the **Mesh** component and select it to access its **Details** panel. First, assign **SK_Mannequin** to the **Skeletal Mesh** parameter, as shown:

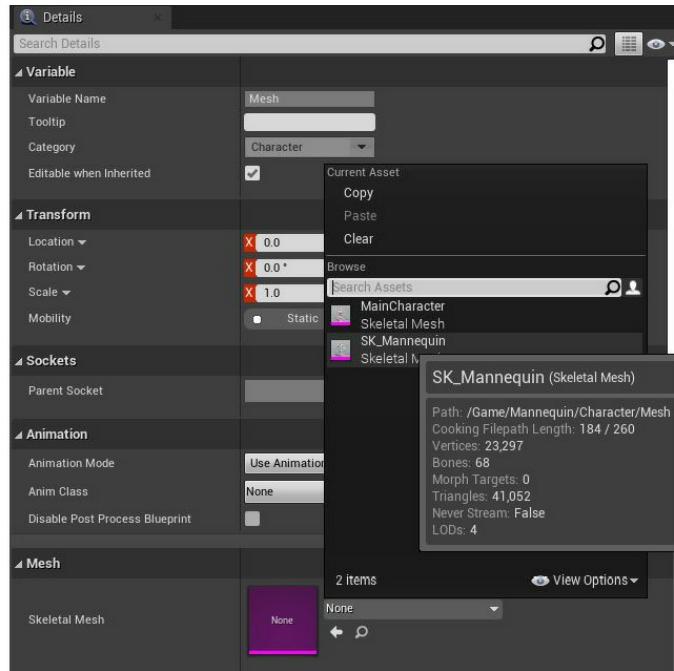


Figure 12.24: You will use the default **SK_Mannequin** skeletal mesh for the new enemy

7. Now you need to apply the **AnimBP_Enemy Animation Blueprint** to the **Mesh** component. Navigate to the **Animation** category of the **Mesh** component's **Details** panel, and under **Anim Class**, assign **AnimBP_Enemy**:

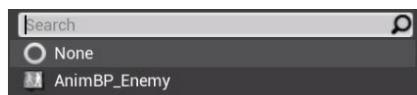


Figure 12.25: Assign the new **AnimBP_Enemy Animation Blueprint** as the **Anim Class** for

the enemy character

8. Lastly, you will notice that the character mesh is positioned and rotated incorrectly when previewing the character in the **Preview** window. Fix this by setting the **Transform** property of the **Mesh** component to the following:

1. **Location:** (**X = 0.000000**,
Y = 0.000000, **Z = -90.000000**)
2. **Rotation:** (**Roll= 0.000000**, Pitch= **0**, Yaw= **-90.000000**)
3. **Scale:** (**X = 1.000000, one**
= 1.000000, Z = 1.000000)

The **Transform** settings will appear as follows:



Figure 12.26: These are the transform settings so that your character is positioned and rotated correctly

The following figure shows the settings of the **Mesh** component so far. Please make sure your settings match what is displayed here:

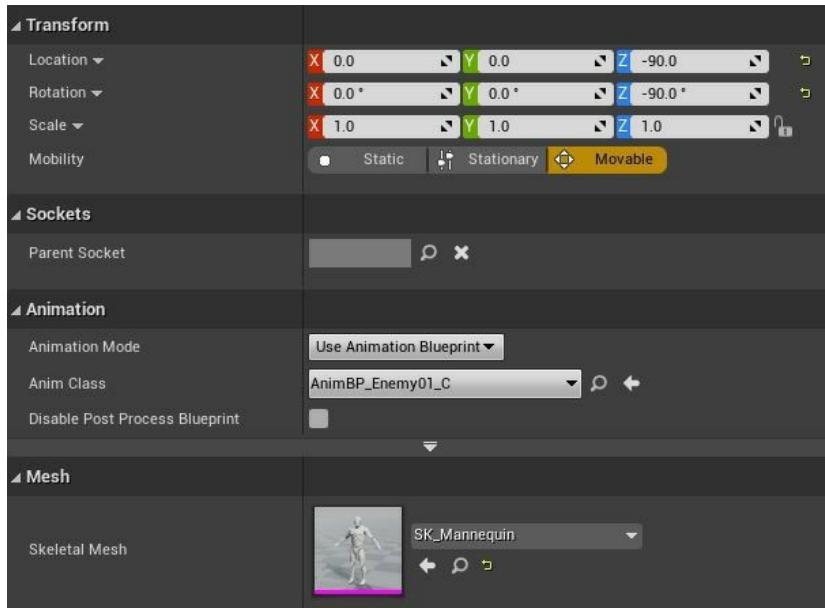


Figure 12.27: The settings for the Mesh component of your enemy character

The last thing to do here is to create a **Material Instance** of the mannequin's primary material so that this enemy can have a unique color that helps differentiate it from the other enemy type.

Let's begin by first learning more about Materials and **Material Instances**.

Materials and Material Instances

Before moving on to the next exercise, we need to first briefly discuss what Materials and **Material Instances** are before you can work with these assets and apply them to the new enemy character. Although this book is more focused on the technical aspects of game development using Unreal Engine 4, it is still important that you know, on a surface level, what Materials and **Material Instances** are and how they

are used in video games.

Note

*For more information about Materials, please refer to the following Epic Games documentation:
<https://docs.unrealengine.com/en-US/Engine/Rendering/Materials/index.html>.*

A Material is a type of asset that can be applied to a mesh that will then control how the mesh looks in-game. The **Material** editor gives you control over many parts of how the end visual result will look, including control over parameters such as **Textures**, **Emissive**, and **Specular**, among others. The following image shows the default mannequin skeletal mesh with the **M_Ue4Man_Body**-applied **Material** asset:



Figure 12.28: The default mannequin skeletal mesh with a basic Material applied

A **Material Instance** is an extension of a **Material** where you do not have access or control over the base **Material** from which the **Material Instance** derives, but you do have control over the parameters that the creator of the **Material** exposes to you. Many parameters can be exposed to you to work with from inside **Material**

Instances.

Unreal Engine provides us with an example of a **Material Instance** in the **Side Scroller** template project called **M_UE4Man_ChestLogo**, found in the **/Mannequin/Character/Materials/** directory. The following image shows the set of exposed parameters given to the **Material Instance** based on the parent material, **M_UE4Man_Body**. The most important parameter to focus on is the **Vector** parameter, called **BodyColor**. You will use this parameter in the **Material Instance** you create in the next exercise to give the enemy a unique color:

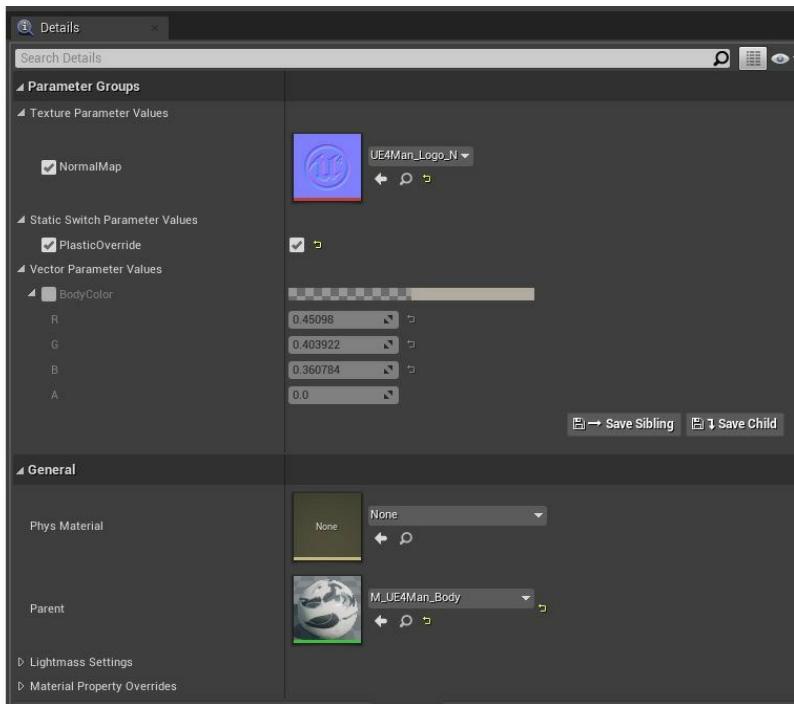


Figure 12.29: The list of parameters for the M_UE4Man_ChestLogo Material Instance asset

Exercise 12.09: Creating and Applying the Enemy

Material Instance

Now that you have a basic understanding of what materials and material instances are, it is time to create your own **Material Instance** from the **M_UE4ManBody** asset. With this **Material Instance**, you will adjust the **BodyColor** parameter to give the enemy character a unique visual representation. Let's start by creating the new **Material Instance**.

The following steps will help you complete this exercise:

1. Navigate to the **/Mannequin/Character/Materials** directory to find the **Material** used by the default mannequin character, **M_UE4ManBody**.
2. A **Material Instance** can be created by *right-clicking* on the **Material** asset, **M_UE4Man_Body**, and *left-clicking* on the **Create Material Instance** option. Name this asset **MI_Enemy01**.

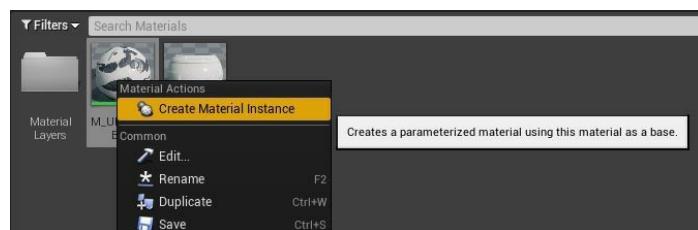


Figure 12.30: Any material can be used to create a Material Instance

Create a new folder called **Materials** in the **Enemy** folder. *Left-click* and drag the **Material Instance** into the **/Enemy/Materials** directory to move the asset to this new folder:

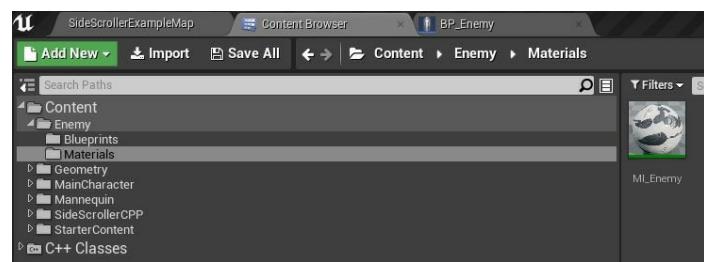


Figure 12.31: Rename the Material Instance MI_Enemy

3. *Double-left-click* the **Material Instance** and find the **Details** panel on the left-hand side. There, you will find a **Vector Parameter** property called **BodyColor**. Make sure the checkbox is checked to enable this parameter, and then change its value to a red color. Now, **Material Instance** should be colored red, as shown:



Figure 12.32: Now, the enemy material is red

4. Save the **Material Instance** asset and navigate back to the **BP_Enemy01** Blueprint. Select the **Mesh** component and update the **Element 0** material parameter to **MI_Enemy**:

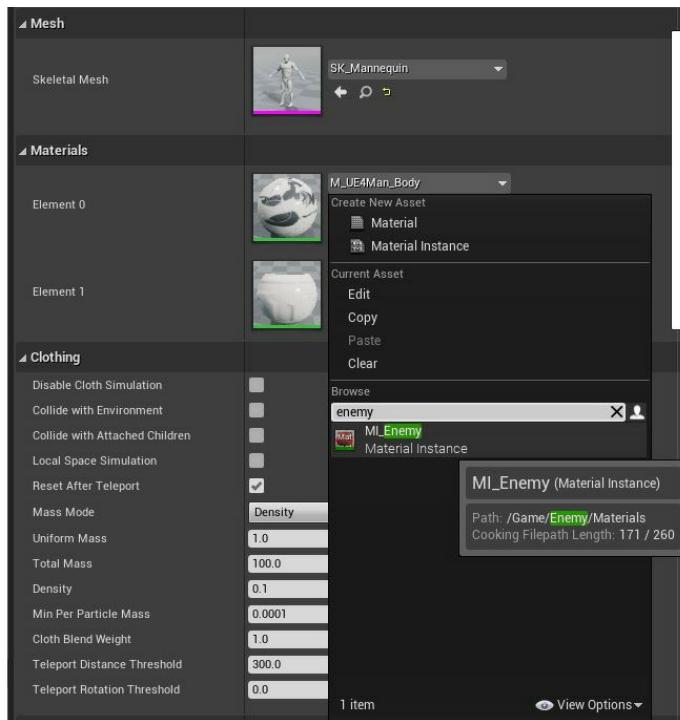


Figure 12.33: Assigning the new

Material Instance asset, MI_Enemy, to Element 0 of the materials for the Mesh component

5. Now, the first enemy type is visually ready and has the appropriate **Blueprint** and Animation Blueprint assets prepared for the next chapter, where you will develop its AI:



Figure 12.34: The final enemy character set up

With this exercise complete, you have now created a **Material Instance** and applied it to the enemy character so that it has a unique visual representation.

Let's conclude this chapter by moving on to a short activity that will help you better understand the blending of animations using the **Layered blend per bone** node that was used in the earlier exercises.

Activity 12.01: Updating Blend Weights

At the end of *Exercise 12.06, Previewing the Throw Animation*, you were able to blend the movement animations and the **Throw** animation so that they both could be played in tandem without negatively influencing each other. The result is the player character animating correctly when walking or running, while also performing the **Throw** animation on the upper body.

In this activity, you will experiment with the blend bias values and parameters of the **Layered blend per bone** node to have a better understanding of how animation blending works.

The following steps will help you complete the activity:

1. Update the **Blend weights** input parameter of the **Layered blend per bone** node so that there is absolutely no blending of the **Throw** animation additive pose with the base movement pose. Try using values here such as **0.0f** and **0.5f** to compare the differences in the animation.

Note

*Make sure to return this value to **1.0f** after you are done so as not to affect the blending you set up in the previous*

exercise.

2. Update the settings of the **Layered blend per bone** node to change which bone is affected by the blend so that the whole character's body is affected by the blend. It's a good idea to start with the root bone in the skeleton hierarchy of the **MainCharacter_Skeleton** asset.
3. Keeping the settings from the previous step in place, add a new array element to the branch filters and, in this new array element, add the bone name and a blend depth value of **-1.0f**, which allows only the character's left leg to continue to animate the movement correctly when blending the **Throw** animation.

Note

*After this activity, make sure to return the settings of the **Layered blend per bone** node to the values you set at the end of the first exercise to ensure no progress is lost in the character's animation.*

The expected outputs are as follows:

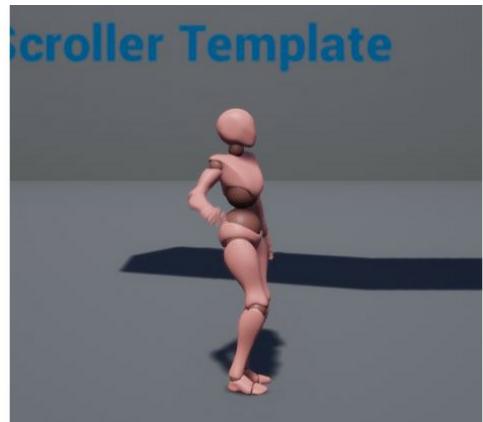


Figure 12.35: Output showing the entire character body affected

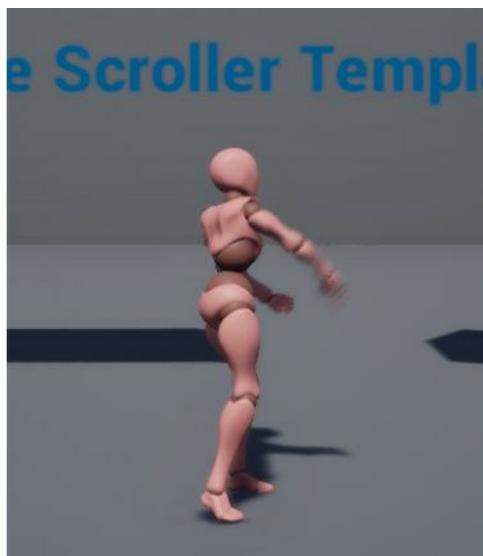


Figure 12.36: The left leg continues to animate the movement correctly when blending the Throw animation

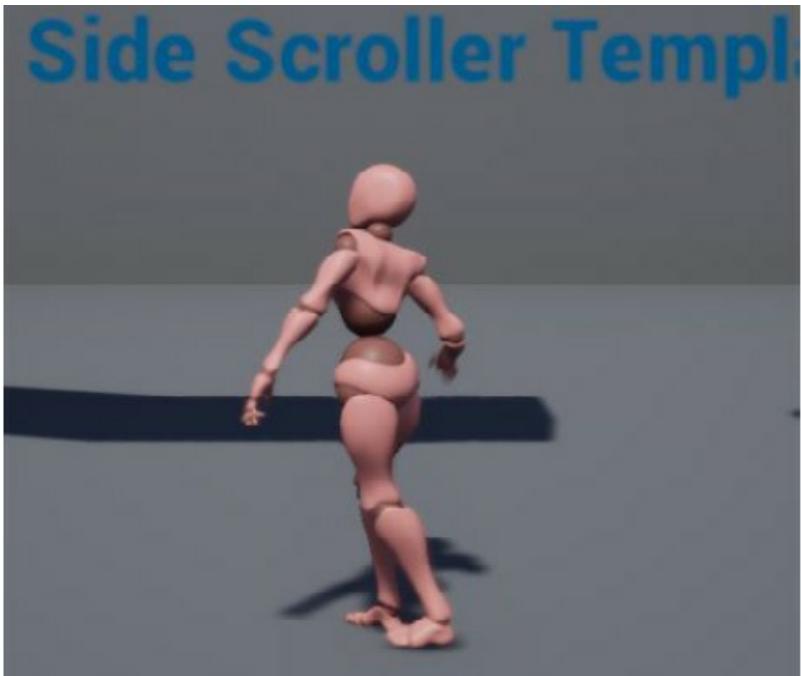


Figure 12.37: The character's right leg animating while moving with the end of the Throw animation

Note

The solution to this activity can be found at:

<https://packt.live/338jEBx>.

*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>.

With this activity complete, you now have a stronger understanding of how animation blending works and how blending weighting can affect the influence of additive poses on base poses using the **Layered blend per bone** node.

Note

There are a lot of techniques for animation blending that you haven't used in this project, and it's strongly recommended that you research these techniques, starting with the documentation at <https://docs.unrealengine.com/en-US/Engine/Animation/AnimationBlending/index.html>.

Summary

With the enemy set up with the C++ class, Blueprint, and Material, you are ready to move on to the next chapter, where you will create the AI for this enemy by taking advantage of systems such as Behavior Trees in Unreal Engine 4.

From the exercises and activities of this chapter, you learned how to create an **Animation Montage** that allows the playing of animations. You also learned how to set up an Anim Slot within this montage to categorize it for the player character's upper body.

Next, you learned how to cache the output pose of a state machine by using the **Use Cached Pose** node so that this pose can be referenced in multiple instances for more complex Animation Blueprints. Then, by learning about the **Layered blend per bone** function, you were able to blend the base movement pose with the additive layer of the **Throw** animation by using the Anim Slot.

Lastly, you put together the base of the enemy by creating the C++ class, Blueprint, and other assets so that they will be ready for the next chapter. With the enemy ready, let's move on to creating the AI of the enemy so that it can interact with the player.

13. Enemy Artificial Intelligence

Overview

*This chapter begins with a brief recap of how the enemy AI will behave for the **SuperSideScroller** game. From there, you will learn about Controllers in the context of Unreal Engine 4 and learn how to create an AI Controller. Then, you will learn more about AI navigation in Unreal Engine 4 by adding a Nav Mesh to the main level of the game.*

By the end of this chapter, you will be able to create a navigable space where the enemy can move. You will also be able to create an enemy AI pawn and navigate it across locations using Blackboard and behavior trees. Lastly, you will know how to create and implement a player projectile class and add visual elements to it.

Introduction

In the previous chapter, you added layered animations for the player character using Animation Blending with a combination of Anim Slots, Animation Blueprints, and blending functions such as Layered Blend per Bone.

In this chapter, you will learn how to use a Nav Mesh to create a navigable space inside of the game world that the enemy can move in. Defining the navigable space of a level is crucial for allowing **artificial intelligence (AI)** to access and move to

specific areas of your level.

Next, you will create an enemy AI pawn that can navigate between patrol point locations inside the game world using a combination of the AI tools present inside Unreal Engine 4, including *Blackboards* and *behavior trees*.

You will also learn how to use a Nav Mesh to create a navigable space inside the game world where the enemy can move. Defining the navigable space of a level is crucial for allowing the AI to access and move to specific areas of your level.

Lastly, you'll learn how to create a player projectile class in C++ and how to implement the `OnHit()` collision event function to recognize and log when the projectile hits an object in the game world. In addition to creating the class, you will then create a Blueprint of this player projectile class and add visual elements to the player projectile, such as a Static Mesh.

The **SuperSideScroller** game is finally coming together and you will be in a good position by the end of this chapter to move on to *Chapter 14, Spawning the Player Projectile*, where you will handle adding elements of polish to the game, such as SFX and VFX.

The primary focus of this chapter is to take the C++ enemy class you created in *Chapter 12, Animation Blending and Montages*, and bring this enemy to life using AI. Unreal Engine 4 uses many different tools to achieve AI such as AI Controllers, Blackboards, and behavior trees, all of which you will learn about and use in this chapter. Before you jump into these systems, let's take a moment to learn about how AI is used in games in recent history. AI has certainly evolved since the days of *Super Mario Bros.*

Enemy AI

What is AI? This term can mean many things, depending on the field and context that it is used in, so let's define it in a way that makes sense regarding the subject of video games.

AI is an entity that is aware of its environment and performs choices that will help optimally achieve its intended purpose. AI uses what are called **finite state machines** to switch between more than one state based on the input it receives from the user or its environment. For example, a video game AI can switch between an offensive state to a defensive state based on its current health.

In games such as *Hello Neighbor*, which was developed in Unreal Engine 4, and *Alien: Isolation*, the goal of the AI is to find the player as efficiently as possible, but also to follow some predetermined patterns defined by the developers to ensure that the player can outsmart it. *Hello Neighbor* adds a very creative element to its AI by having it learn from the players' past actions and tries to outsmart the player based on the knowledge it learns.

You can find an informative breakdown of how the AI works in this video by the publishers of the game, *TinyBuild Games*, here: <https://www.youtube.com/watch?v=Hu7Z52RaBGk>.

Interesting and fun AI is crucial to any game, and depending on the game you are making, this can mean a very complex or very simplistic AI. The AI that you will be creating for the **SuperSideScroller** game will not be as sophisticated as those mentioned previously, but it will fill the needs of the game we are seeking to create.

Let's break down how the enemy will behave:

- The enemy will be a very simple enemy that has a basic back and forth

movement pattern and will not support any attacks; only by colliding with the player character will they be able to inflict any damage.

- However, we need to set the locations to move between for the enemy AI.
- Next, we decide whether the AI should change locations, should constantly move between locations, or should there be a pause in between selecting a new location to move to?

Fortunately for us, Unreal Engine 4 provides us with a wide array of tools that we can use to develop such complex AI. In the case of our project, however, we will use these tools to create a simplistic enemy type. Let's start by discussing what an AI Controller is in Unreal Engine 4.

AI Controller

Let's discuss what the main difference is between a **Player Controller** and an **AI Controller**. Both of these actors derive from the base **Controller class**, and a Controller is used to take control of a **Pawn or Character** in order to control the actions of said pawn or character.

While a Player Controller relies on the input of an actual player, an AI Controller applies AI to the characters they possess and responds to the environment based on the rules set forth by the AI. By doing so, the AI can make intelligent decisions in response to the player and other external factors,

without the actual player explicitly telling it to do so. Multiple instances of the same AI pawn can share the same AI Controller, and the same AI Controller can be used across different AI pawn classes. AI, like all actors inside Unreal Engine 4, are spawned through the **UWorld** class.

Note

*You will be learning more about the **UWorld** class in Chapter 14, Spawning the Player Projectile, but as a reference, please read more here: <https://docs.unrealengine.com/en-US/API/Runtime/Engine/Engine/UWorld/index.html>.*

The most important aspect of both the Player Controller and the AI Controller is the pawns they will control. Let's learn more about how AI Controllers handle this.

Auto Possess AI

Like all Controllers, the AI Controller must possess a *pawn*. In C++, you can use the following function to possess a pawn:

```
void AController::Possess(APawn* InPawn)
```

You can also use the following function to unpossess a pawn:

```
void AController::UnPossess()
```

There's also the **void AController::OnPossess(APawn* InPawn)** and **void AController::OnUnPossess()** functions, which are called whenever the **Possess()** and **UnPossess()** functions are called, respectively.

When it comes to AI, especially in the context of Unreal Engine 4, there are two methods in which AI Pawns or Characters can be possessed by an AI Controller. Let's take a

look at these options:

- **Placed in World:** This first method is how you will be handling AI in this project; you will manually place these enemy actors into your game world, and the AI will take care of the rest once the game begins.
- **Spawned:** This second method is only a little more complicated because it requires an explicit function call, either in C++ or Blueprint, to **Spawn** an instance of a specified class. The **Spawn Actor** method requires a handful of parameters, including the **World** object and **Transform** parameters such as **Location** and **Rotation**, to ensure that the instance that is spawned is spawned correctly.
- **Placed in World or Spawned:** If you are unsure of which method you want to use, a safe option would be **Placed in World or Spawned**; that way, both methods are supported.

For the purposes of the **SuperSideScroller** game, you will be using the **Placed In World** option because the AI you will create will be manually placed in the game level.

Exercise 13.01: Implementing AI Controllers

Before the enemy pawn can do anything, it needs to be possessed by an AI Controller. This also needs to happen before any logic can be performed by the AI. This exercise will be performed within the Unreal Engine 4 editor. By the end of this exercise, you will have created an AI Controller and applied it to the enemy that you created in the previous chapter. Let's begin by creating the AI Controller actor.

The following steps will help you complete this exercise:

1. Head to the **Content Browser** interface and navigate to the **Content/Enemy** directory.
2. *Right-click* on the **Enemy** folder and select the **New Folder** option. Name this new folder **AI**. In the new **AI** folder directory, *right-click* and select the **Blueprint Class** option.
3. From the **Pick Parent Class** dialogue box, expand **All Classes** and manually search for the **AIController** class.
4. *Left-click* this class option and then *left-click* on the green **Select** option

at the bottom to create a new **Blueprint** from this class. Please refer to the following screenshot to know where to find the **AIController** class. Also, take note of the tooltip that appears when hovering over the class option; it contains useful information about this class from the developers:

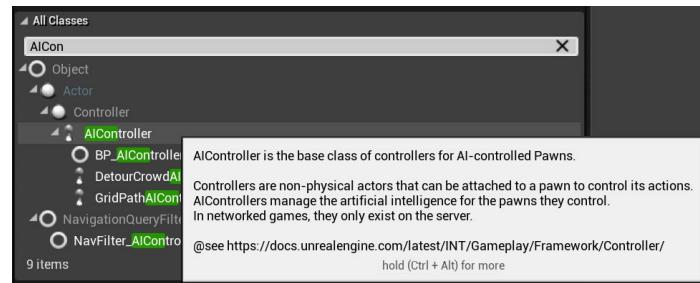


Figure 13.1: The AIController asset class, as found in the Pick Parent Class dialogue box

5. With this new **AIController Blueprint** created, name this asset **BP_AIControllerEnemy**.

With the AI Controller created and named, it's time to assign this asset to the first enemy Blueprint that you made in the previous chapter.

6. Navigate to the **/Enemy/Blueprints** directly to find **BP_Enemy**. Double-

click to open this Blueprint.

7. In the **Details** panel of the first enemy **Blueprint**, there is a section labeled **Pawn**. This is where you can set different parameters regarding the AI functionality of **Pawn** or **Character**.
8. The **AI Controller Class** parameter determines, as its name suggests, which AI Controller to use for this enemy. *Left-click* on the dropdown to find and select the AI Controller you made earlier; that is, **BP_AIController_Enemy**.

With this exercise complete, the enemy AI now knows which AI Controller to use. This is crucial because it is in the AI Controller where the AI will use and execute the behavior tree you will create later in this chapter.

The AI Controller is now assigned to the enemy, which means you are almost ready to start developing the actual intelligence for this AI. There is still one important topic to discuss before doing so, and that is the Navigation Mesh.

Navigation Mesh

One of the most crucial aspects of any AI, especially in video games, is the ability to navigate the environment in a sophisticated manner. In Unreal Engine 4, there is a way for

the engine to tell the AI which parts of an environment navigable and which parts are not. This is done through a **Navigation Mesh**, or **Nav Mesh** for short.

The term Mesh is misleading here because it's implemented through a volume in the editor. We will need a Navigation Mesh in our level so that our AI can effectively navigate the playable bounds of the game world. We'll add one together in the following exercise.

Unreal Engine 4 also supports a **Dynamic Navigation Mesh**, which allows the Nav Mesh to update in real-time as dynamic objects move around the environment. This results in the AI recognizing these changes in the environment and updating their pathing/navigation appropriately. This book will not cover this, but you can access the configuration options via **Project Settings -> Navigation Mesh -> Runtime Generation**.

Exercise 13.02: Implementing a Nav Mesh Volume for the AI Enemy

In this exercise, you will add a Navigation Mesh to **SideScrollerExampleMap** and explore how Navigation Meshes work in Unreal Engine 4. You'll also learn how to parameterize this volume for the needs of your game. This exercise will be performed within the Unreal Engine 4 editor.

By the end of this exercise, you will have a stronger understanding of the Nav Mesh. You will also be able to implement this volume in your own level in the activity that follows this exercise. Let's begin by adding the Nav Mesh

volume to the level.

The following steps will help you complete this exercise:

1. If you do not already have the map open, please open **SideScrollerExampleMap** by navigating to **File** and *left-clicking* on the **Open Level** option. From the **Open Level** dialogue box, navigate to **/SideScrollerCPP/Maps** to find **SideScrollerExampleMap**. Select this map with a *left-click* and then *left-click Open* at the bottom to open the map.
2. With the map opened, navigate to the right-hand side to find the **Modes** panel. The **Modes** panel is a set of easily accessible actor types such as **Volumes**, **Lights**, **Geometry**, and others. Under the **Volumes** category, you will find the **Nav Mesh Bounds Volume** option.
3. *Left-click* and drag this volume into the map/scene. By default, you will see the outline of the volume in the editor. Press the **P** key to visualize the

Navigation area that the volume encompasses, but make sure that the volume is intersecting with the ground geometry in order to see the green visualization, as shown in the following screenshot:



Figure 13.2: Areas outlined in green are perceived as navigable by the engine and the AI

With the **Nav Mesh** volume in place, let's adjust its shape so that the volume extends to the entire area of the level. After this, you'll learn how to adjust the parameters of the **Nav Mesh** volume for the purposes of the game.

4. *Left-click* to select **NavMeshBoundsVolume** and navigate to its **Details** panel. There is a section labeled **Brush Settings** that allows you to adjust the shape and size

of the volume. Find the values that fit best for you. Some suggested settings are **Brush Type: Additive**, **Brush Shape: Box, X: 3000.0, Y: 3000.0**, and **Z: 3000.0**.

Notice that when the shape and dimensions of

NavMeshBoundsVolume change, **Nav Mesh** will adjust and recalculate the navigable area. This can be seen in the following screenshot. You will also notice that the upper platforms are not navigable; you will fix this later:



Figure 13.3: Now, NavMeshBoundsVolume extends to the entire playable area of the example map

By completing this exercise, you have placed your first **NavMeshBoundsVolume** actor into the game world and, using the debug key, '**P**', visualized the navigable area in the default map. Next, you will learn more about the **RecastNavMesh** actor, which is also created when placing **NavMeshBoundsVolume** into the level.

Recasting the Nav Mesh

When you added **NavMeshBoundsVolume**, you may have noticed that another actor was created automatically: a **RecastNavMesh** actor called **RecastNavMesh-Default**. This **RecastNavMesh** acts as the "brain" of the Nav Mesh because it contains the parameters needed to adjust the Nav Mesh that directly influences how the AI navigates the given area.

The following screenshot shows this asset, as seen from the **World Outliner** tab:

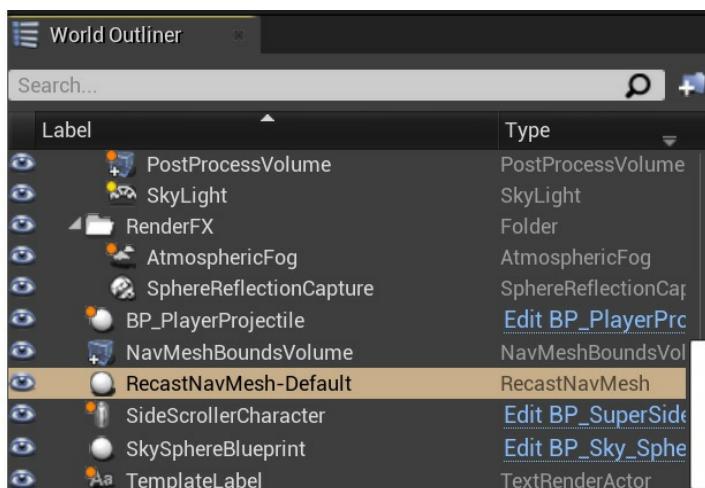


Figure 13.4: The RecastNavMesh actor, as seen from the World Outliner tab

Note

*There are a lot of parameters that exist in **RecastNavMesh**, and we will only be covering the important parameters in this book. For more information, check out <https://docs.unrealengine.com/en-US/API/Runtime/NavigationSystem/NavMesh/ARecastNavMesh/index.html>.*

There are only two primary sections that are important to you right now:

1. **Display:** The **Display** section, as the name suggests, only contains parameters that affect the visual debug display of the generated navigable area of **NavMeshBoundsVolume**. It is recommended that you try toggling each of the parameters under this category to see how it affects the display of the generated Nav Mesh.

2. **Generation:** The **Generation** category contains a set of values that act as a rule set for how the Nav Mesh will generate and determine which areas of geometry are navigable, and which are not. There are many options here, which can make the concept very daunting, but let's discuss just a handful of the parameters under this category:

1. **Cell Size** refers to the accuracy in which the Nav Mesh can generate navigable space within an area. You will be updating this value in the next step of this exercise, so

you'll see how this affects the navigable area in real time.

2. **Agent Radius** refers to the radius of the actor that will be navigating this area. In the case of your game, the radius to set here is the radius of the collision component of the character with the largest radius.
3. **Agent Height** refers to the height of the actor that will be navigating this area. In the case of your game, the height to set here is the Half Height of the collision component of the character with the largest Half Height. You can multiply it by **2.0f** to get the full height.
4. **Agent Max Slope** refers to the slope angle for inclines that can exist in your game world. By default, the value is **44** degrees, and this is a parameter you will leave alone unless your game

requires that it changes.

5. Agent Max Step Height

refers to the height of steps,
regarding staircase steps,
that can be navigated by the
AI. Much like **Agent Max Slope**, this is a parameter
that you will more than likely
leave alone unless your game
specifically requires this
value to change.

Now that you have learned about the Recast Nav Mesh parameters, let's put this knowledge into practice with the next exercise, where will walk you through changing a few of these parameters.

Exercise 13.03: Recasting Nav Mesh Volume Parameters

Now that you have the **Nav Mesh** volume in the level, it is time to change the parameters of the **Recast Nav Mesh** actor so that the Nav Mesh allows the enemy AI to navigate across platforms that are thinner than others. This exercise will be performed within the Unreal Engine 4 editor.

The following step will help you complete this exercise:

1. You will be updating **Cell Size** and **Agent Height** so that they fit the needs of your character and the accuracy needed for the Nav Mesh:

Cell Size: 5.0f

Agent Height: 192.0f

The following screenshot shows that the upper platforms are now navigable because of the changes we made to

Cell Size:



Figure 13.5: Changing Cell Size from 19.0f to 5.0f allows for the narrow upper platforms to be navigable

With **SuperSideScrollerExampleMap** set up with its own **Nav Mesh**, you can now move on and create the AI logic for the enemy. Before doing so, complete the following activity to create your own level, with its own unique layout and **NavMeshBoundsVolume** actor that you can use for the remainder of this project.

Activity 13.01: Creating a New Level

Now that you have added **NavMeshBoundsVolume** to the example map, it is time to create your own map for the purposes of the rest of the **Super SideScroller** game. By creating your own map, you will have a better understanding of how **NavMeshBoundsVolume** and the properties of **RecastNavMesh** affect the environment they are placed in.

Note

*Before moving on to the solution for this activity, if you need an example level that will work for the remaining chapters of the **SuperSideScroller** game, then don't worry – this chapter comes with the **SuperSideScroller.umap** asset, as well as a map called **SuperSideScroller_NoNavMesh**, which does not contain the **NavMeshBoundsVolume**. You can use **SuperSideScroller.umap** as a reference for how to create your own level, or to get ideas on how to improve your own level. You can download the map here: <https://packt.live/3lo7v2f>.*

Perform the following steps to create a simplistic map:

1. Create a **New Level**.
2. Name this level
SuperSideScroller.
3. Using the static mesh assets provided by default in the **Content Browser** interface of this project, create an

interesting space with different elevations to navigate. Add your player character **Blueprint** to the level, and make sure it is possessed by **Player Controller 0**.

4. Add the **NavMeshBoundsVolume** actor to your level and adjust its dimensions so that it fits the space you created. In the example map provided for this activity, the dimensions set should be **1000.0, 5000.0**, and **2000.0** in the *X*, *Y*, and *Z* axes, respectively.
5. Make sure to enable debug visualization for **NavMeshBoundsVolume** by pressing the **P** key.
6. Adjust the parameters of the **RecastNavMesh** actor so that **NavMeshBoundsVolume** works well for your level. In the case of the provided example map, the **Cell Size** parameter 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.

Expected Output:

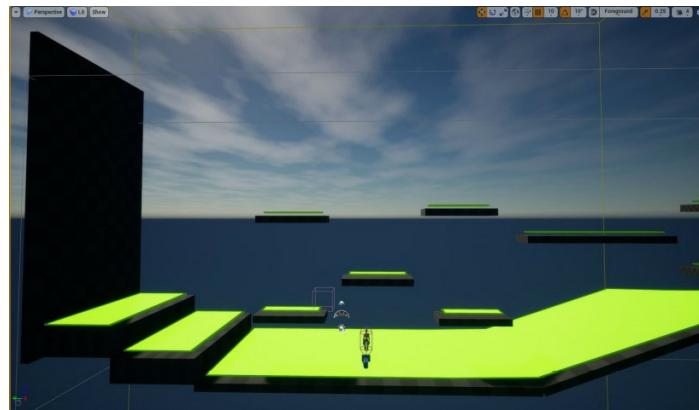


Figure 13.6: SuperSideScroller map

By the end of this activity, you will have a level that contains the required **NavMeshBoundsVolume** and settings for the **RecastNavMesh** actor. This will allow the AI we'll develop in the upcoming exercises to function correctly. Again, if you are unsure of how the level should look, please refer to the provided example map, **SuperSideScroller.umap**. Now, it is time to jump into developing the AI for the **SuperSideScroller** game.

Note

*The solution to this activity can be found at:
<https://packt.live/338jEBx>.*

Behavior Trees and Blackboards

Behavior trees and Blackboards work together to allow our AI to follow different logical paths and make decisions based on a variety of conditions and variables.

A **behavior tree (BT)** is a visual scripting tool that allows you to tell a pawn what to do based on certain factors and parameters. For example, a BT can tell an AI to move to a certain location based on whether the AI can see the player.

To give an example of how BTs and Blackboards are used in games, let's look at the game *Gears of War 5*, which was developed with Unreal Engine 4. The AI in Gears of War 5, and throughout the Gears of War series, always try to flank the player, or force the player out of cover. In order to do this, a key component of the AI logic is to know who the player is, and where the player is. A reference variable to the player, and a location vector to store the location of the player, exist in the Blackboard. The logic that determines how these variables are used and how the AI will use this information is performed inside the behavior tree.

The Blackboard is where you define the set of variables that are required in order to have the behavior tree perform actions and use those values for decision-making.

The behavior tree is where you create the Tasks that you want the AI to perform, such as moving to a location, or performing a custom Task that you create. Like many of the in-editor tools in Unreal Engine 4, behavior trees are, for the most part, a very visual scripting experience.

Blackboards are where you define the variables, also known as **Keys**, that will then be referenced by the **behavior tree**. The Keys you create here can be used in **Tasks**, **Services**, and **Decorators** to serve different purposes based on how you want the AI to function. The following screenshot shows an example set of variable Keys that can be referenced by its associated behavior tree.

Without a Blackboard, behavior trees would have no way of passing and storing information across different Tasks, Services, or Decorators, rendering it useless:

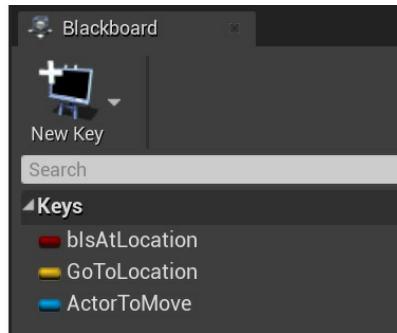


Figure 13.7: An example set of variables inside a Blackboard that can be accessed in the behavior tree

Behavior trees are composed of a set of **objects** – that is, **Composites**, **Tasks**, **Decorators**, and **Services** – that work together to define how the AI will behave and respond based on the conditions and logic flow that you set. All behavior trees begin with what is called the Root where the logic flow begins; this cannot be modified and has only one execution branch. Let's take a look at these objects in more detail:

Composites

Composite nodes function as a means to tell the behavior tree how to go about performing Tasks and other actions. The following screenshot shows the full list of Composite nodes that Unreal Engine gives you by default: Selector, Sequence, and Simple Parallel.

Composite nodes can also have Decorators and Services attached to them in order to have optional conditions applied before a behavior tree branch is executed:

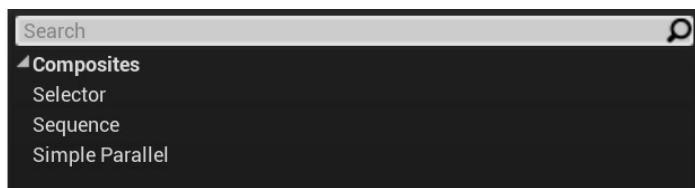


Figure 13.8: The full list of Composite nodes – Selector, Sequence, and Simple Parallel

- **Selector:** The Selector composite node executes its children from left to right and will stop executing when one of the children Tasks succeeds. Using the example shown in the following screenshot, if the **FinishWithResult** task is successful, the parent Selector succeeds, which will cause the Root to execute again and **FinishWithResult** to execute once more. This pattern will continue until **FinishWithResult** fails. The Selector will then execute **MakeNoise**. If **MakeNoise** fails, the **Selector** fails, and the Root will execute again. If the **MakeNoise** task succeeds, then the Selector will succeed, and the Root will execute again. Depending on the flow of the behavior tree, if the Selector fails or succeeds, the next composite branch will begin to execute. In the following screenshot, there are no other composite nodes, so if the Selector fails or succeeds, the Root node will be executed again. However, if there were a Sequence composite

node with multiple Selector nodes underneath, each Selector would attempt to successfully execute its children. Regardless of success or failure, each Selector will attempt execution sequentially:

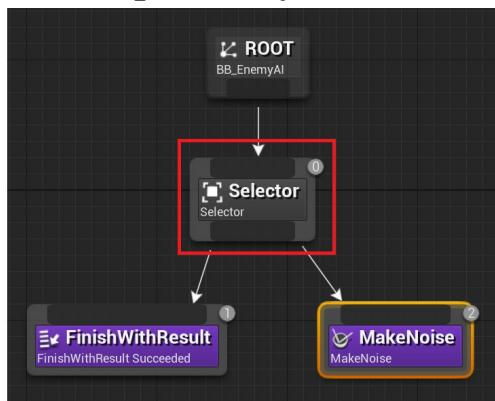


Figure 13.9: An example of how a Selector Composite node can be used in a behavior tree

Note that when adding Tasks and **Composite** nodes, you will notice numeric values on the top-right corners of each node. These numbers indicate the order in which these nodes will be executed. The pattern follows the *top to bottom, left to right*, paradigm, and these values help you keep track of the ordering. Any disconnected Task or **Composite** node will be given a value of **-1** to indicate that it is unused.

- **Sequence:** The **Sequence** composite node executes its children from left to right and will stop executing when one of the children Tasks fails. Using the example shown in the following screenshot, if the **Move To** task is

successful, then the parent Sequence node will execute the **Wait** task. If the **Wait** task is successful, then the Sequence is successful, and **Root** will execute again. If the **Move To** task fails, however, the Sequence will fail and **Root** will execute again, causing the **Wait** task to never execute:

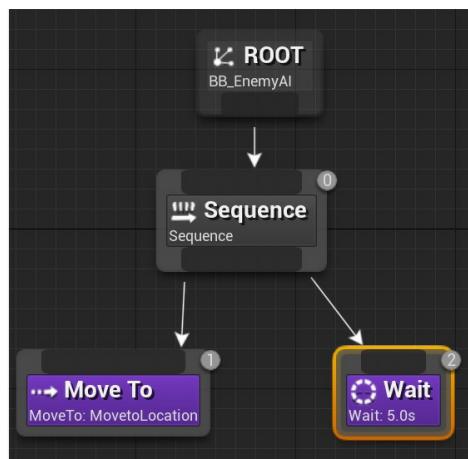


Figure 13.10: An example of how a Sequence Composite node can be used in a behavior tree

- **Simple Parallel:** The **Simple Parallel** composite node allows you to execute a **Task** and a new standalone branch of logic simultaneously. The following screenshot shows a very basic example of what this will look like. In this example, a task used to **Wait** for 5 seconds is being executed at the same

time as a new **Sequence** of Tasks is being executed:

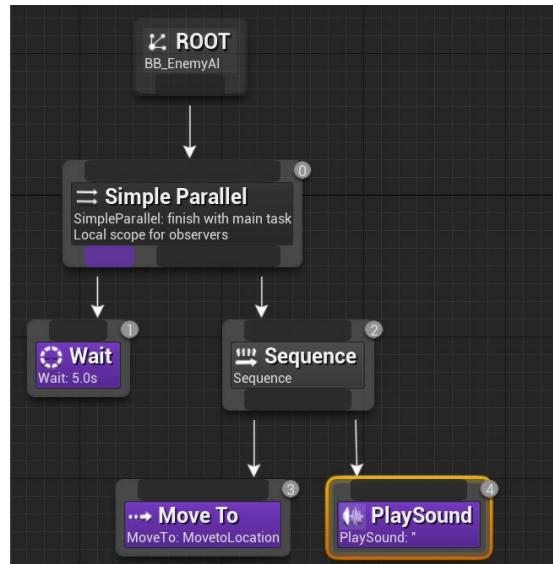


Figure 13.11: An example of how a Selector Composite node can be used in a behavior tree

The **Simple Parallel** composite node is also the only **Composite** node that has a parameter in its **Details** panel, which is **Finish Mode**. There are two options:

- **Immediate:** When set to **Immediate**, the Simple Parallel will finish successfully once the main Task finishes. In this case, after the **Wait** task finishes, the background tree Sequence will abort and the entire **Simple Parallel** will execute again.
- **Delayed:** When set to **Delayed**, the Simple Parallel will finish successfully once the background tree finishes its

execution and the Task finishes. In this case, the **Wait** task will finish after **5** seconds, but the entire **Simple Parallel** will wait for the **Move To** and **PlaySound** Tasks to execute before restarting.

Tasks

These are Tasks that our AI can perform. Unreal provides us with built-in Tasks for us to use by default, but we can also create our own in both Blueprints and in C++. This includes Tasks such as telling our AI to **Move To** a specific location, **Rotate To a direction**, and even telling the AI to fire its weapon. It's also important to know that you can create your own custom Tasks using Blueprints. Let's briefly discuss two of the Tasks you will be using to develop the AI for the enemy character:

- **Move To Task:** This is one of the more commonly used Tasks in behavior trees, and you will be using this task in the upcoming exercises in this chapter. **Move To task** uses the navigation system to tell the AI how and where to move based on the location it is given. You will use this task to tell the AI enemy where to go.
- **Wait Task:** This is another

commonly used task in behavior trees because it allows a delay in between task execution if the logic requires it. This can be used to allow the AI to wait a few seconds before moving to a new location.

Decorators

Decorators are conditions that can be added to Tasks or **Composite** nodes, such as a **Sequence** or **Selector**, that allows branching logic to occur. As an example, we can have a **Decorator** that checks whether or not the enemy knows the location of the player. If so, we can tell that enemy to move toward that last known location. If not, we can tell our AI to generate a new location and move there instead. It is also important to know that you can create your own custom Decorators using Blueprints.

Let's also briefly discuss the Decorator you will be using to develop the AI for the enemy character – the **Is At Location** decorator. This determines whether the controlled pawn is at the location specified in the Decorator itself. This will be useful to you to ensure that the behavior tree is not executing until you know the AI has reached its given location.

Services

Services work a lot like Decorators because they can be linked with **Tasks** and **Composite** nodes. The main difference is that a **Service** allows us to execute a branch of nodes based on the interval defined in the Service. It also

important to know that you can create your own custom Services using Blueprints.

Exercise 13.04: Creating the AI Behavior Tree and Blackboard

Now that you have had an overview of behavior trees and Blackboards, this exercise will guide you through creating these assets, telling the AI controller to use the behavior tree you created, and assigning the Blackboard to the behavior tree. The Blackboard and behavior tree assets you will create here will be used for the **SuperSideScroller** game. This exercise will be performed within the Unreal Engine 4 editor.

The following steps will help you complete this exercise:

1. Within the **Content Browser** interface, navigate to the **/Enemy/AI** directory. This is the same directory where you created the AI Controller.
2. In this directory, *right-click* within the blank area of the **Content Browser** interface, navigate to the **Artificial Intelligence** option, and select **Behavior Tree** to create the **Behavior Tree** asset. Name this asset **BT_EnemyAI**.

3. In the same directory as the previous step, *right-click* again within the blank area of the **Content Browser** interface, navigate to the **Artificial Intelligence** option, and select **Blackboard** to create the **Blackboard** asset. Name this asset **BB_EnemyAI**.

Before we move on to telling the AI controller to run this new behavior tree, let's first assign the Blackboard to this behavior tree so that they are properly connected.

4. Open **BT_EnemyAI** by *double-clicking* the asset in the **Content Browser** interface. Once opened, navigate to the **Details** panel on the right-hand side and find the **Blackboard Asset** parameter.
5. *Left-click* the dropdown menu on this parameter and find the **BB_EnemyAI Blackboard** asset you created earlier. Compile and save the behavior tree before closing it.
6. Next, open the AI Controller

BP_AIController_Enemy asset by *double-clicking* it inside the **Content Browser** interface. Inside the controller, *right-click* and search for the **Run Behavior Tree** function.

The **Run Behavior Tree** function is very straightforward: you assign a **Behavior Tree** to the controller and the function returns whether the behavior tree successfully began its execution.

7. Lastly, connect the **Event BeginPlay** event node to the execution pin of the **Run Behavior Tree** function and assign **Behavior Tree asset BT_EnemyAI**, which you created earlier in this exercise:

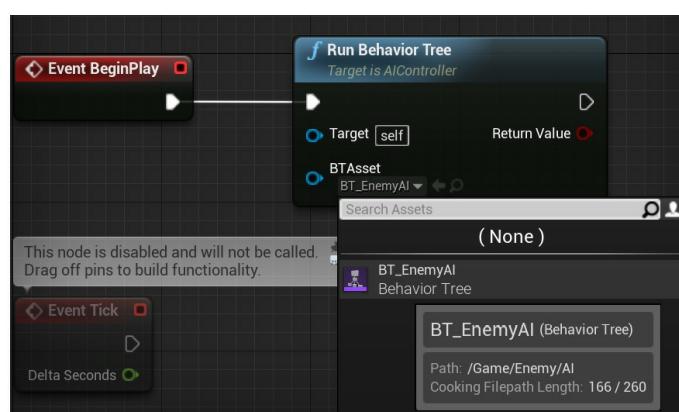


Figure 13.12: Assigning the BT_EnemyAI behavior tree

With this exercise complete, the enemy AI Controller now knows to run the **BT_EnemyAI** behavior tree, and this behavior tree knows to use the Blackboard asset called **BB_EnemyAI**. With this in place, you can begin to use the behavior tree logic to develop the AI so that the enemy character can move around the level.

Exercise 13.05: Creating a New Behavior Tree Task

The goal of this exercise is to develop an AI task for the enemy AI that will allow the character to find a random point to move to within the bounds of the **Nav Mesh** volume in your level.

Although the **SuperSideScroller** game will only allow two-dimensional movement, let's get the AI to move anywhere within the 3D space of the level that you created in *Activity 13.01, Creating a New Level*, and then work to constrain the enemy to two dimensions.

Follow these steps to create this new Task for the enemy:

1. First, open the Blackboard asset you created in the previous exercise, **BB_EnemyAI**.
2. *Left-click* on the **New Key** option at

the top-left of **Blackboard** and select the **Vector** option. Name this vector **MoveToLocation**. You will use this **vector** variable to track the next move for the AI as it decides where to move to.

For the purposes of this enemy AI, you will need to create a new **Task** because the currently available Tasks inside Unreal do not fit the needs of the enemy behavior.

3. Navigate to and open the **Behavior Tree** asset you created in the previous exercise, **BT_EnemyAI**.
4. *Left-click* on the **New Task** option on the top toolbar. When creating a new **Task**, it will automatically open the task asset for you. However, if you have already created a Task, a dropdown list of options will appear when selecting the **New Task** option. Before working on the logic of this **Task**, you will rename the asset.
5. Close the **Task** asset window and navigate to **/Enemy/AI/**, which is

where the **Task** was saved to. By default, the name provided is

BTask_BlueprintBase_New.

Rename this asset

BTask_FindLocation.

6. With the new **Task** asset named, *double-click* to open **Task Editor**. New Tasks will have their Blueprint graphs completely empty and will not provide you with any default events to use in the graph.
7. *Right-click* within the graph and from the context-sensitive search, find the **Event Receive Execute AI** option.
8. *Left-click* the **Event Receive Execute AI** option to create the event node in the **Task** graph, as shown in the following screenshot:



Figure 13.13: Event Receive Execute AI returns both the Owner Controller and the Controlled Pawn

Note

The Event Receive Execute AI event will give you access to both the **Owner Controller** and the **Controlled Pawn**. You will use the **Controlled Pawn** for this Task in the upcoming steps.

9. Each **Task** requires a call to the **Finish Execute** function so that the **Behavior Tree** asset knows when it can move onto the next **Task** or branches off the tree. *Right-click* in the graph and search for **Finish Execute** via the context-sensitive search.
10. *Left-click* the **Finish Execute** option from the context-sensitive search to create the node inside the Blueprint graph of your **Task**, as shown in the following screenshot:



Figure 13.14: The Finish Execute

function, which has a Boolean parameter that determines whether the Task is successful

The next function that you need is called

GetRandomLocationInNavigableRadius. This function, as the name suggests, returns a random vector location within a defined radius of the navigable area. This will allow the enemy character to find random locations and move to those locations.

11. *Right-click* in the graph and search for **GetRandomLocationInNavigableRadius** inside the context-sensitive search. *Left-click* the **GetRandomLocationInNavigableRadius** option to place this function inside the graph.

With these two functions in place, and with **Event Receive Execute AI** ready, it is time to obtain the random location for the enemy AI.

12. From the **Controlled Pawn** output of **Event Receive Execute AI**, find the **GetActorLocation** function

via the context-sensitive search:

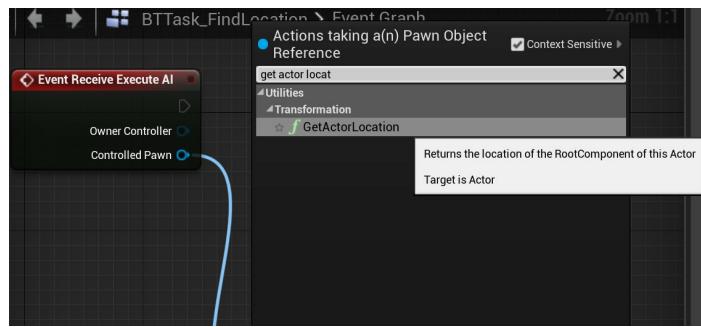


Figure 13.15: The enemy pawn's location will serve as the origin of the random point selection

13. Connect the vector return value from **GetActorLocation** to the **Origin** vector input parameter of the **GetRandomLocationInNavigableRadius** function, as shown in the following screenshot. Now, this function will use the enemy AI pawn's location as the origin for determining the next random point:

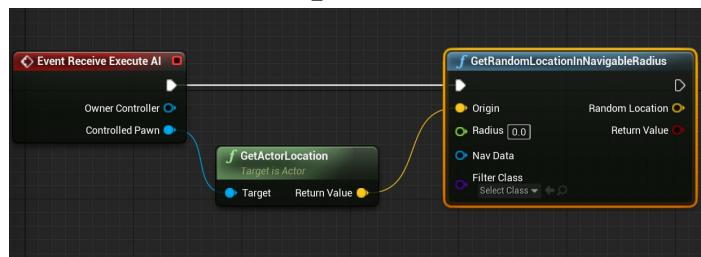


Figure 13.16: Now, the enemy pawn location will be used as the origin of the random point vector search

14. Next, you need to tell the **GetRandomLocationInNavigableRadius** function the **Radius** in which to check for the random point in the navigable area of the level. Set this value to **1000.0f**.

The remaining parameters, **Nav Data** and **Filter Class**, can remain as they are. Now that you are getting a random location from **GetRandomLocationInNavigableRadius**, you will need to be able to store this value in the **Blackboard** vector that you created earlier in this exercise.

15. To get a reference to the **Blackboard** vector variable, you need to create a new variable inside of this **Task** that's of the **Blackboard Key Selector** type. Create this new variable and name it **NewLocation**.
16. You now need to make this variable a **Public** variable so that it can be exposed inside the behavior tree. *Left-click* on the 'eye' icon so that the eye is visible.

17. With the **Blackboard Key Selector** variable ready, *left-click* and drag out a **Getter** of this variable. Then, pull from this variable and search for **Set Blackboard Value as Vector**, as shown in the following screenshot:

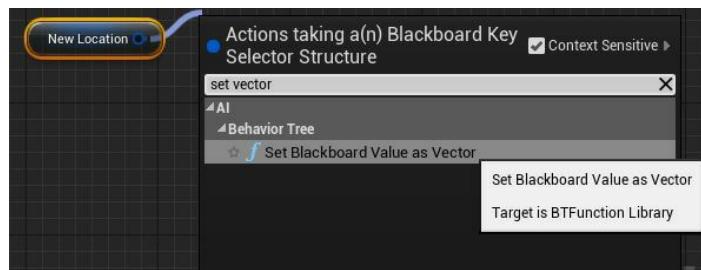


Figure 13.17: Set Blackboard Value has a variety of different types to support the different variables that can exist inside the Blackboard

18. Connect the **RandomLocation** output vector from **GetRandomLocationInNavigableRadius** to the **Value** vector input parameter of **Set Blackboard Value as Vector**. Then, connect the execution pins of these two function nodes. The result will look as follows:

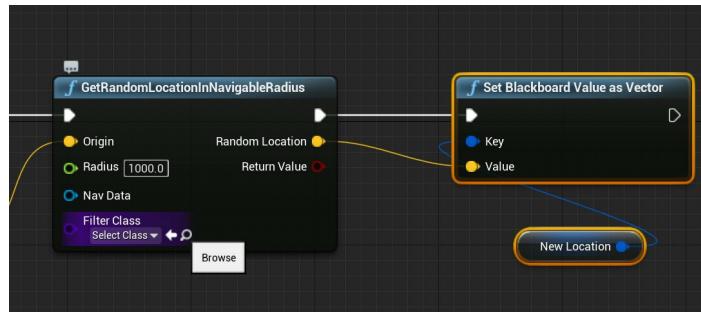


Figure 13.18: Now, the Blackboard vector value is assigned this new random location

Lastly, you will use the **Return Value** Boolean output parameter of the **GetRandomLocationInNavigableRadius** function as the means to determine whether the **Task** executes successfully.

19. Connect the Boolean output parameter to the **Success** input parameter of the **Finish Execute** function and connect the execution pins of the **Set Blackboard Value as Vector** and **Finish Execute** function nodes. The following screenshot shows the final result of the **Task** logic:

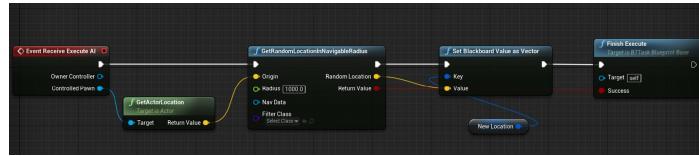


Figure 13.19: The final setup for the Task

Note

You can find the preceding screenshot in full resolution for better viewing at the following link:

<https://packt.live/3lmLyk5>.

By completing this exercise, you have created your first custom **Task** using Blueprints in Unreal Engine 4. You now have a task that finds a random location within the navigable bounds of the **Nav Mesh Volume** of your level using the enemy pawn as the origin of this search. In the next exercise, you will implement this new **Task** in the behavior tree and see the enemy AI move around your level.

Exercise 13.06: Creating the Behavior Tree Logic

The goal of this exercise is to implement the new **Task** you created in the previous exercise inside the behavior tree in order to have the enemy AI find a random location within the navigable space of your level and then move to this location. You will use a combination of the **Composite**, **Task**, and **Services** nodes to accomplish this behavior. This exercise will be performed within the Unreal Engine 4 editor.

The following steps will help you complete this exercise:

1. To start, open the behavior tree you created in *Exercise 13.04, Creating the AI Behavior Tree and Blackboard*, which is **BT_EnemyAI**.
2. Inside this **Behavior Tree**, *left-click* and drag from the bottom of the **Root** node and select the **Sequence** node from the context-sensitive search. The result will be the **Root** that's connected to the **Sequence** composite node.
3. Next, from the **Sequence** node, *left-click* and drag to bring up the context-sensitive menu. In this menu, search for the **Task** you created in the last previous, **BTTask_FindLocation**.
4. By default, the **BTTask_FindLocation** task should automatically assign the **New Location** key selector variable to the **MovetoLocation** vector variable from **Blackboard**. If this doesn't happen, you can assign this selector manually in the **Details** panel of the Task.

Now, **BTTask_FindLocation** will

assign the **NewLocation** selector to the **MovetoLocation** vector variable from **Blackboard**. This means that the random location that's returned from the task will be assigned to the **Blackboard** variable and that you can reference this variable in other Tasks.

Now that you are finding a valid random location and assigning this location to the **Blackboard** variable, that is, **MovetoLocation**, you can use the **Move To** task to tell the AI to move to this location.

5. *Left-click and pull from the **Sequence** composite node. Then, from the context-sensitive search, find the **Move To** task. Your **Behavior Tree** will now look as follows:*

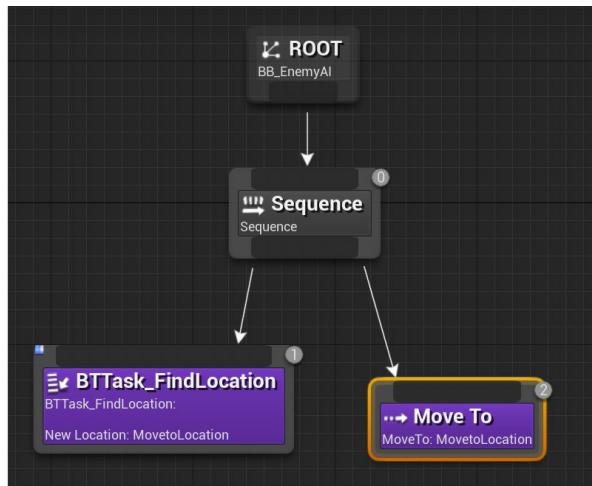


Figure 13.20: After selecting the random location, the Move To task will let the AI move to this new location

6. By default, the **Move To** task should assign **MoveToLocation** as its **Blackboard Key** value. If it doesn't, select the task. In its **Details** panel, you will find the **Blackboard Key** parameter, which is where you can assign the variable. While in the **Details** panel, also set **Acceptable Radius** to **50.0f**.

Now, the behavior tree finds the random location using the **BTask_FindLocation** custom task and tells the AI to move to that location using the **MoveTo** task. These two Tasks communicate the location to each other by referencing the **Blackboard** vector variable called **MovetoLocation**.

The last thing to do here is to add a **Decorator** to the **Sequence** composite node so that it ensures that the enemy character is not at a random

location before executing the tree again to find and move to a new location.

7. *Right-click* on the top area of the **Sequence** and select **Add Decorator**. From the dropdown, *left-click* and select **Is at Location**.
8. Since you already have a vector parameter inside **Blackboard**, this **Decorator** should automatically assign **MovetoLocation** as **Blackboard Key**. Verify this by selecting the **Decorator** and making sure **Blackboard Key** is assigned to **MovetoLocation**.
9. With the Decorator in place, you have completed the behavior tree. The final result will look as follows:

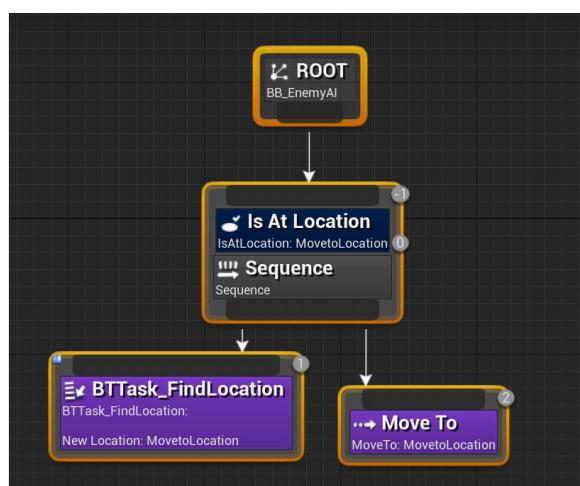


Figure 13.21: The final setup for the behavior tree for the AI enemy

This behavior tree is telling the AI to find a random location using **BTTTask_FindLocation** and assign this location to the Blackboard value called **MovetoLocation**. When this task is successful, the behavior tree will execute the **MoveTo** task, which will tell the AI to move to this new random location. The Sequence is wrapped in a **Decorator** that ensures that the enemy AI is at **MovetoLocation** before executing again, just as a safety net for the AI.

10. Before you can test the new AI behavior, make sure to place a **BP_Enemy AI** into your level if one is not already there from previous exercises and activities.
11. Now, if you use **PIE**, or **Simulate**, you will see the enemy AI run around the map and move to random locations within **Nav Mesh Volume**:

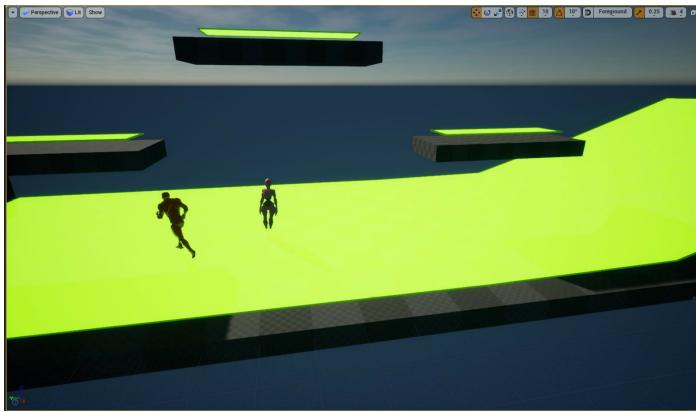


Figure 13.22: The enemy AI will now move from location to location

Note

*There can be some cases where the enemy AI will not move. This can be caused by the **GetRandomLocationInNavigableRadius** function not returning **True**. This is a known issue, and if it occurs, please restart the editor and try again.*

By completing this exercise, you have created a fully functional behavior tree that allows the enemy AI to find and move to a random location within the navigable bounds of your level using **Nav Mesh Volume**. The Task you created in the previous exercise allows you to find this random point, while the **Move To** task allows the AI character to move toward this new location.

Due to how the **Sequence** composite node works, each task must complete successfully before it can move on to the next task, so first, the enemy successfully finds a random location and then moves toward this location. Only when the **Move To** task completes will the entire behavior tree start over and choose a new random location.

Now, you can move on to the next activity, where you will add

to this behavior tree in order to have the AI wait between selecting a new random point so that the enemy isn't constantly moving.

Activity 13.02: AI Moving to the Player Location

In the previous exercise, you were able to have the AI enemy character move to random locations within the bounds of **Nav Mesh Volume** by using a custom **Task** and the **MoveTo** task together.

In this activity, you will continue from the previous exercise and update the behavior tree. You will take advantage of the **Wait** task by using a **Decorator**, and also create your own new custom task to have the AI follow the player character and update its position every few seconds.

The following steps will help you complete this activity:

1. Inside the **BT_EnemyAI** behavior tree that you created in the previous exercise, you will continue from where you left off and create a new Task. Do this by selecting **New Task** from the toolbar and choosing **BTask_BlueprintBase**. Name this new Task **BTask_FindPlayer**.
2. In the **BTask_FindPlayer** Task, create a new Event called **Event**

Receive Execute AI.

3. Find the **Get Player Character** function in order to get a reference to the player; make sure to use **Player Index 0**.
4. From the player character, call the **Get Actor Location** function in order to find the players' current location.
5. Create a new Blackboard Key **Selector** variable inside this Task. Name this variable **NewLocation**.
6. *Left-click* and drag the **NewLocation** variable into the graph. From this variable, search for the **Set Blackboard Value** function as **Vector**.
7. Connect **Set Blackboard Value** as a **Vector** function to the execution pin of the Event's **Receive Execute AI** node.
8. Add the **Finish Execute** function, ensuring that the Boolean **Success** parameter is **True**.

9. Lastly, connect **Set Blackboard Value** as a **Vector** function to the **Finish Execute** function.
10. Save and compile the Task **Blueprint** and return to the **BT_EnemyAI** behavior tree.
11. Replace the **BTask_FindLocation** Task with the new **BTask_FindPlayer** Task so that this new Task is now the first task underneath the **Sequence** composite node.
12. Add a new **PlaySound** task as the third task underneath the **Sequence** composite node by following the custom **BTask_FindLocation** and **Move To** Tasks.
13. In the **Sound to Play** parameter, add the **Explosion_Cue SoundCue** asset.
14. Add an **Is At Location** Decorator to the **PlaySound** Task and ensure that the **MovetoLocation** Key is assigned to this **Decorator**.

15. Add a new **Wait** Task as the fourth Task underneath the **Sequence** composite node following the **PlaySound** Tasks.
16. Set the **Wait** task to wait **2.0f** seconds before completing successfully.

The expected output is as follows:

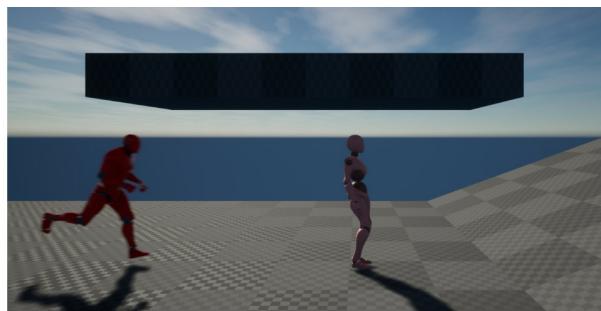


Figure 13.23: Enemy AI following the player and updating to the player location every 2 seconds

The enemy AI character will move to the players' last known location in the navigable space of the level and pause for **2.0f** seconds between each player position.

Note

*The solution to this activity can be found at:
<https://packt.live/338jEBx>.*

With this activity complete, you have learned to create a new Task that allows the AI to find the player location and move to the player's last known position. Before moving onto the next set of exercises, remove the **PlaySound** task and replace the **BTask_FindPlayer** task with the **BTask_FindLocation** task you created in *Exercise 13.05*,

Creating a New Behavior Tree Task. Please refer to *Exercise 13.05, Creating a New Behavior Tree Task* and *Exercise 13.06, Creating the Behavior Tree Logic*, to ensure the Behavior Tree is returned correctly. You will be using the **BTask_FindLocation** task in the upcoming exercises.

In the next exercise, you will address this issue by developing a new **Blueprint** actor that will allow you to set up specific positions that the AI can move toward.

Exercise 13.07: Creating the Enemy Patrol Locations

The current issue with the AI enemy character is that they can move freely around the 3D navigable space because the behavior tree allows them to find a random location within that space. Instead, the AI needs to be given patrol points that you can specify and change in the editor. It will then choose one of these patrol points at random to move to. This is what you will do for the **SuperSideScroller** game: create patrol points that the enemy AI can move to. This exercise will show you how to create these patrol points using a simple *Blueprint* actor. This exercise will be performed within the Unreal Engine 4 editor.

The following steps will help you complete this exercise:

1. First, navigate to the **/Enemy/Blueprints/** directory.
This is where you will create the new **Blueprint** actor that will be used for

the AI patrol points.

2. In this directory, *right-click* and choose the **Blueprint Class** option by *left-clicking* this option from the menu.
3. From the **Pick Parent Class** menu prompt, *left-click* the **Actor** option to create a new **Blueprint** based on the **Actor** class:

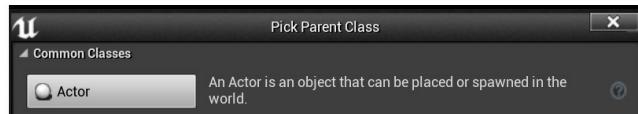


Figure 13.24: The Actor class is the base class for all objects that can be placed or spawned in the game world

4. Name this new asset **BP_AIPoints** and open this **Blueprint** by *double-clicking* the asset in the **Content Browser** interface.

Note

The interface for Blueprints shares many of the same features and layouts as other systems, such as Animation

Blueprints and Tasks, so this should all look familiar to you.

5. Navigate to the **Variables** tab on the left-hand side of the Blueprint UI and *left-click* on the **+Variable** button. Name this variable **Points**.
6. From the **Variable Type** dropdown, *left-click* and select the **Vector** option.
7. Next, you will need to make this vector variable an **Array** so that you can store multiple patrol locations. *Left-click* the yellow icon next to **Vector** and *left-click* to select the **Array** option.
8. The last step for setting up the **Points** vector variable is to enable **Instance Editable** and **Show 3D Widget**:
 1. The **Instance Editable** parameter allows this vector variable to be publicly visible on the actor when placed in a level, allowing each instance of this actor to have this variable available to edit.

2. **Show 3D Widget** allows you to position the vector value by using a visible 3D transform widget in the editor viewport. You will see what this means in the next steps of this exercise. It is also important to note that the **Show 3D Widget** option is only available for variables that involve an actor transform, such as **Vectors** and **Transforms**.

With the simple actor set up, it is time to place the actor into the level and begin setting up the *Patrol Point* locations.

9. Add the **BP_AIPoints** actor Blueprint into your level, as shown in the following screenshot:

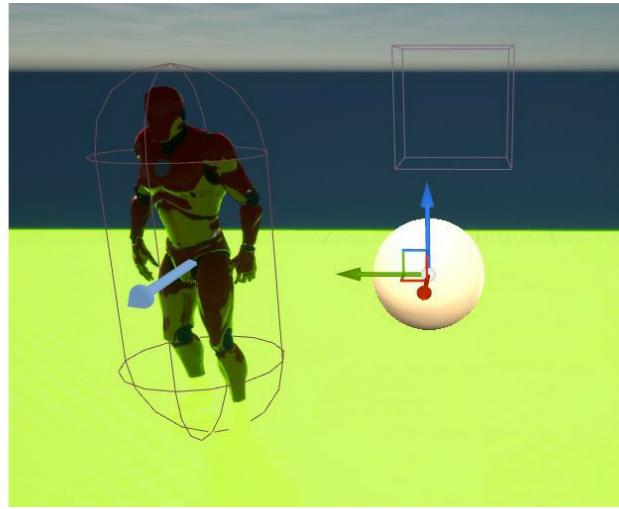


Figure 13.25: The BP_AIPoints actor now in the level

10. With the **BP_AIPoints** actor selected, navigate to its **Details** panel and find the **Points** variable.
11. Next, you can add a new element to the vector array by *left-clicking* on the **+** symbol, as shown here:

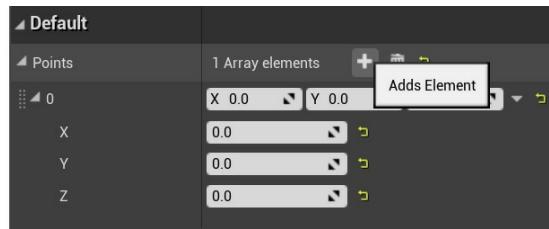


Figure 13.26: You can have many elements inside of an array, but the larger the array, the more memory is allocated

12. When you add a new element to the

vector array, you will see a 3D widget appear that you can then *left-click* to select and move around the level, as shown here:

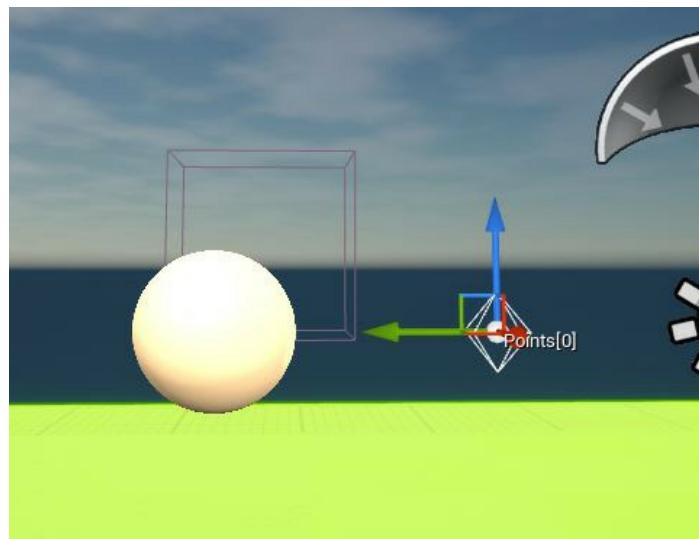


Figure 13.27: The first Patrol Point vector location

Note

*As you update the position of the 3D widget that represents the element of the vector array, the 3D coordinates will update in the **Details** panel for the **Points** variable.*

13. Finally, add as many elements into the vector array as you would like for the context of your level. Keep in mind that the positions of these patrol points

should line up so that they make a straight line along the horizontal axis, parallel to the direction in which the character will move. The following screenshot shows the setup in the example **SideScroller.umap** level included in this exercise:



Figure 13.28: The example Patrol Point path, as seen in the SideScroller.umap example level

14. Continue to repeat the final step to create multiple patrol points and position the 3D widgets as you see fit. You can use the provided **SideScroller.umap** example level as a reference on how to set up these **Patrol Points**.

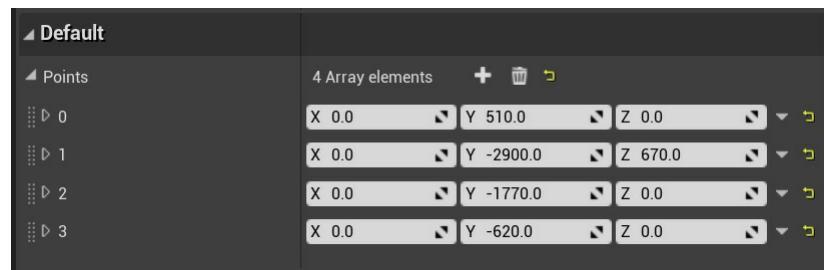
By completing this exercise, you have created a new **Actor Blueprint** that contains a **Vector** array of positions that you can now manually set using a 3D widget in the editor. With the ability to manually set the *Patrol Point* positions, you have full control over where the AI can move to, but there is one problem. There is no functionality in place to choose a point

from this array and to pass it to the behavior tree so that the AI can move between these *Patrol Points*. Before you set up this functionality, let's learn a bit more about Vectors and Vector Transformation, as this knowledge will prove useful in the next exercise.

Vector Transformation

Before you jump into the next exercise, it is important that you get to know about Vector Transformation and, more importantly, what the **Transform Location** function does. When it comes to an actor's location, there are two ways of thinking of its position: in terms of world space and local space. An actor's position in world space is its location relative to the world itself; in more simple terms, this is the location where you place the actual actor into the level. An actor's local position is its location relative to either itself or a parent actor.

Let's consider the **BP_AIPoints** actor as an example of what world space and local space are. Each of the locations of the **Points** array are local-space Vectors because they are positions relative to the world-space position of the **BP_AIPoints** actor itself. The following screenshot shows the list of Vectors in the **Points** array, as shown in the previous exercise. These values are positions relative to the location of the **BP_AIPoints** actor in your level:



The screenshot shows the Unreal Engine Editor's Details panel. At the top, there is a section labeled "Default". Below it, there is a table-like structure for the "Points" array, which contains four elements. Each element has three columns: X, Y, and Z. The values for the first element are X: 0.0, Y: 510.0, Z: 0.0. The values for the second element are X: 0.0, Y: -2900.0, Z: 670.0. The values for the third element are X: 0.0, Y: -1770.0, Z: 0.0. The values for the fourth element are X: 0.0, Y: -620.0, Z: 0.0.

Default		
Points		4 Array elements
⋮ 0	X 0.0	Y 510.0 Z 0.0
⋮ 1	X 0.0	Y -2900.0 Z 670.0
⋮ 2	X 0.0	Y -1770.0 Z 0.0
⋮ 3	X 0.0	Y -620.0 Z 0.0

Figure 13.29: The local-space position Vectors of the Points array, relative to the world-space position of

the BP_AIPoints actor

In order to have the enemy AI move to the correct world space location of these **Points**, you need to use a function called **Transform Location**. This function takes in two parameters:

- **T**: This is the supplied **Transform** that you use to convert the vector location parameter from a local-space into a world-space value.
- **Location**: This is the **location** that is to be converted from local space to world space.

The result of this vector transformation is then returned as the Return Value of the function. You will use this function in the next exercise to return a randomly selected vector point from the **Points** array and convert that value from a local-space vector into a world-space vector. This new world-space vector will then be used to tell the enemy AI where to move relative to the world. Let's implement this now.

Exercise 13.08: Selecting a Random Point in an Array

Now that you have more information about Vectors and Vector Transformation, you can move onto this exercise, where you will create a simple **Blueprint** function to select

one of the *Patrol Point* vector locations and transform its vector from a local space value into a world space value using a built-in function called **Transform Location**. By returning the world space value of the vector position, you can then pass this value to the *behavior tree* so that the AI will move to the correct position. This exercise will be performed within the Unreal Engine 4 editor.

The following steps will help you complete this exercise. Let's start by creating the new function:

1. Navigate back to the **BP_AIPoints** Blueprint and create a new function by *left-clicking* the **+** button next to the **Functions** category on the left-hand side of the Blueprint editor. Name this function **GetNextPoint**.
2. Before you add logic to this function, select this function by *left-clicking* it under the **Functions** category to access its **Details** panel.
3. In the **Details** panel, enable the **Pure** parameter so that this function is labeled as a **Pure Function**. You learned about **Pure Functions** in *Chapter 11, Blend Spaces 1D, Key Bindings, and State Machines*, when working in the Animation Blueprint for the player character; the same thing is

happening here.

4. Next, the **GetNextPoint** function needs to return a vector that the behavior tree can use to tell the enemy AI where to move to. Add this new output by *left-clicking* on the + symbol under the **Outputs** category of the **Details** function. Make the variable of type **Vector** and give it the name **NextPoint**, as shown in the following screenshot:

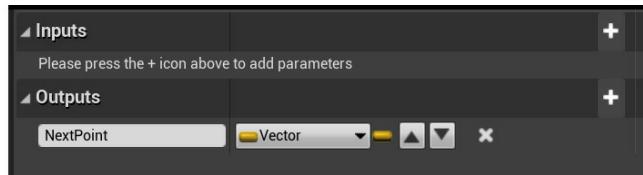


Figure 13.30: Functions can return multiple variables of different types, depending on the needs of your logic

5. When adding an **Output** variable, the function will automatically generate a **Return** node and place it into the function graph, as shown in the following screenshot. You will use this output to return the new vector patrol point for the enemy AI to move to:

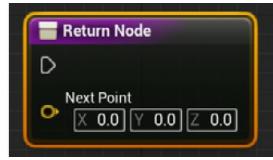


Figure 13.31: The automatically generated Return Node for the function, including the NewPoint vector output variable

Now that the function groundwork is completed, let's start adding the logic.

6. In order to pick a random position, first, you need to find the length of the **Points** array. Create a **Getter** of the **Points** vector and from this vector variable, *left-click* and drag to search for the **Length** function, as shown in the following screenshot:

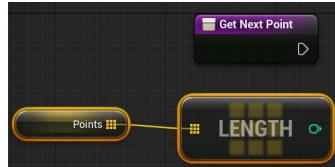


Figure 13.32: The Length function is a pure function that returns the length of the array

7. With the integer output of the **Length** function, *left-click* and drag out to use the context-sensitive search to find the

Random Integer function, as shown in the following screenshot. The **Random Integer** function returns a random integer between **0** and **Max value**; in this case, this is the **Length** of the **Points** vector array:

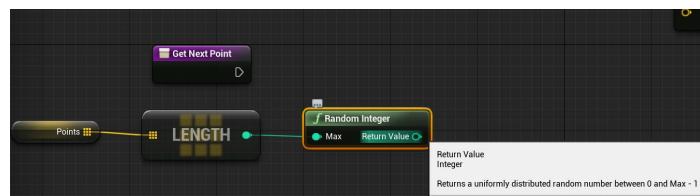


Figure 13.33: Using Random Integer will allow the function to return a random vector from the Points vector array

So far, you are generating a random integer between **0** and the Length of the **Points** vector array. Next, you need to find the element of the **Points** vector array at the index position of the returned **Random Integer**.

8. Do this by creating a new **Getter of the Points** vector array. Then, *left-click* and drag to search for the **Get (a copy)** function.
9. Next, connect the Return Value of the **Random Integer** function to the

input of the **Get (a copy)** function. This will tell the function to choose a random integer and use that integer as the index to return from the **Points** vector array.

Now that you are getting a random vector from the **Points** vector array, you need to use the **Transform Location** function in order to convert the location from a local space into a world space vector.

As you have learned already, the Vectors in the **Points** array are local space positions relative to the position of the **BP_AIPoints** actor in the level.

As a result, you need to use the **Transform Location** function to convert the randomly selected local space vector into a world-space vector so that the AI enemy moves to the correct position.

10. *Left-click and drag from the vector output of the **Get (a copy)** function and via the context-sensitive search, find the **Transform Location** function.*

11. Connect the vector output of the **Get (a copy)** function to the **Location** input of the **Transform Location** function.
12. The final step is to use the transform of the Blueprint actor itself as the **T** parameter of the **Transform Location** function. Do this by *right-clicking* inside the graph and via the context-sensitive search, find the **GetActorTransform** function and connect it to the **Transform Location** parameter, **T**.
13. Finally, connect the **Return Value** vector from the **Transform Location** function and connect it to the **NewPoint** vector output of the function:

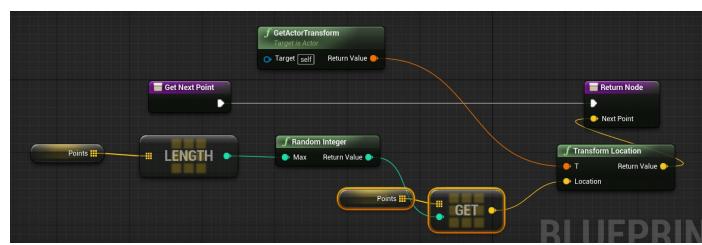


Figure 13.34: The final logic set up for the GetNextPoint function

Note

You can find the preceding screenshot in full resolution for better viewing at the following link:

<https://packt.live/35jlilb>.

By completing this exercise, you have created a new Blueprint function inside the **BP_AIPoints** actor that takes a random index from the **Points** array variable, transforms it into a world space vector value using the **Transform Location** function, and returns this new vector value. You will use this function inside the **BTask_FindLocation** task, inside the AI behavior tree, so that the enemy will move to one of the points you have set up. Before you can do this, the enemy AI needs a reference to the **BP_AIPoints** actor so that it knows which points it can select from and move to. We'll do this in the following exercise.

Exercise 13.09: Referencing the Patrol Point Actor

Now that the **BP_AIPoints** actor has a function that returns a random transformed location from its array of vector patrol points, you need to have the enemy AI reference this actor in the level so that it knows which patrol points to reference. To do this, you will add a new **Object Reference** variable to the enemy character Blueprint and assign the **BP_AIPoints** actor that you placed in your level earlier. This exercise will be performed within the Unreal Engine 4 editor. Let's get started by adding the *Object Reference*.

Note

An **Object Reference Variable** stores a reference to a specific class object or actor. With this reference variable, you can get access to the publicly exposed variables, events, and functions that this class has available.

The following steps will help you complete this exercise:

1. Navigate to the **/Enemy/Blueprints/** directory and open the enemy character Blueprint **BP_Enemy** by *double-clicking* the asset from the **Content Browser** interface.
2. Create a new variable of the **BP_AIPoints** type and make sure the variable type is of **Object Reference**.
3. In order to reference the existing **BP_AIPoints** actor in your level, you need to make the variable from the previous step a **Public Variable** by enabling the **Instance Editable** parameter. Name this variable **Patrol Points**.
4. Now that you have the object reference

set, navigate to your level and select your enemy AI. The following screenshot shows the enemy AI placed in the provided example level; that is, **SuperSideScroller.umap**. If you don't have an enemy placed in your level, please do so now:

Note

Placing an enemy into a level works the same as it does for any other actor in Unreal Engine 4. Left-click and drag the enemy AI Blueprint from the Content Browser interface into the level.

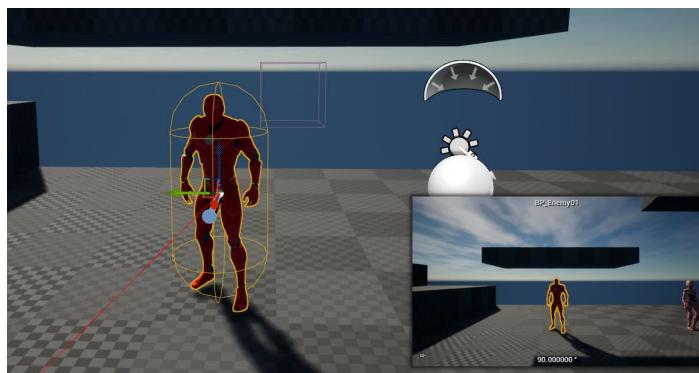


Figure 13.35: The enemy AI placed in the example level **SuperSideScroller.umap**

5. From its **Details** panel, find the **Patrol Points** variable under the

Default category. The last thing to do here is assign the **BP_AIPoints** actor we already placed in the level in *Exercise 13.07, Creating the Enemy Patrol Locations*. Do this by *left-clicking* the dropdown menu for the **Patrol Points** variable and finding the actor from the list.

With this exercise complete, the enemy AI in your level now has a reference to the **BP_AIPoints** actor in your level. With a valid reference in place, the enemy AI can use this actor to determine which set of points to move between inside the **BTTTask_FindLocation** task. All that is left to do now is update the **BTTTask_FindLocation** task so that it uses these points instead of finding a random location.

Exercise 13.10: Updating BTTTask_FindLocation

The final step in completing the enemy AI patrolling behavior is to replace the logic inside **BTTTask_FindLocation** so that it uses the **GetNextPoint** function from the **BP_AIPoints** actor instead of finding a random location within the navigable space of your level. This exercise will be performed within the Unreal Engine 4 editor.

As a reminder, check back and see how the **BTTTask_FindLocation** task looked at the end of *Exercise 13.05, Creating a New Behavior Tree Task*, before you start.

The following steps will help you complete this exercise:

1. The first thing to do is take the returned **Controlled Pawn** reference from **Event Receive Execute AI** and cast it to **BP_Enemy**, as shown in the following screenshot. This way, you can access the **Patrol Points** object reference variable from the previous exercise:

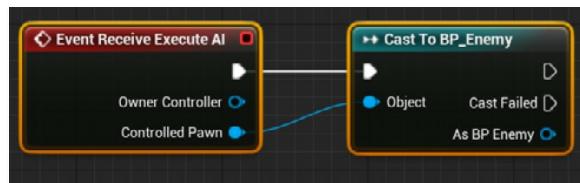


Figure 13.36: Casting also ensures that the returned Controlled Pawn is of the BP_Enemy class type

2. Next, you can access the **Patrol Points** object reference variable by *left-clicking* and dragging from the **As BP Enemy** pin under the cast to **BP_Enemy**, and via the context-sensitive search, finding **Patrol Points**.
3. From the **Patrol Points** reference, you can *left-click* and drag to search for

the **GetNextPoint** function that you created in *Exercise 13.08, Selecting a Random Point in an Array*.

4. Now, you can connect the **NextPoint** vector output parameter of the **GetNextPoint** function to the **Set Blackboard Value as Vector** function and connect the execution pins from the cast to the **Set Blackboard Value as Vector** function. Now, each time the **BTask_FindLocation** task is executed, a new random patrol point will be set.
5. Lastly, connect the **Set Blackboard Value as Vector** function to the **Finish Execute** function and manually set the **Success** parameter to **True** so that this task will always succeed if the cast is successful.
6. As a failsafe, create a duplicate of **Finish Execute** and connect to the **Cast Failed** execution pin of the **Cast** function. Then, set the **Success** parameter to **False**. This will act as a

failsafe so that if, for any reason, **Controlled Pawn** is not of the **BP_Enemy** class, the task will fail. This is a good debugging practice to ensure the functionality of the task for its intended AI class:

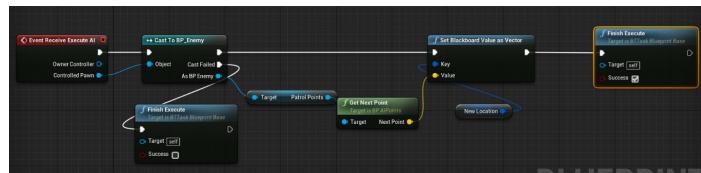


Figure 13.37: It is always good practice to account for any casting failures in your logic

Note

You can find the preceding screenshot in full resolution for better viewing at the following link:
<https://packt.live/3n58THA>.

With the **BTask_FindLocation** task updated to use the random patrol point from the **BP_AIPoints** actor reference in the enemy, the enemy AI will now move between the patrol points at random.

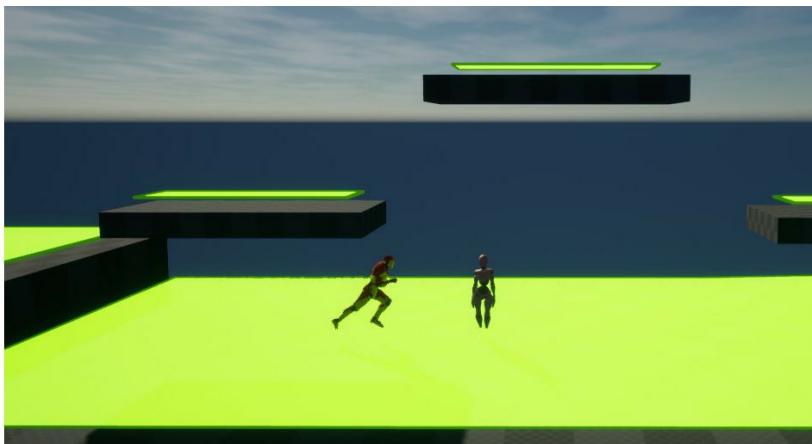


Figure 13.38: The enemy AI now moving between the patrol point locations in the level

With this exercise complete, the enemy AI now uses the reference to the **BP_AIPoints** actor in the level to find and move to the patrol points in the level. Each instance of the enemy character in the level can have its own reference to another unique instance of the **BP_AIPoints** actor or can share the same instance reference. It is up to you how you would like each enemy AI to move throughout the level.

Player Projectile

For the last section of this chapter, you will focus on creating the base of the player projectile, which can be used to destroy enemies. The goal is to create the appropriate actor class, introduce the required collision and projectile movement components to the class, and set up the necessary parameters for the projectile's motion behavior.

For the sake of simplicity, the player projectile will not use gravity, will destroy enemies with one hit, and the projectile itself will be destroyed on hitting any surface; it will not bounce off walls, for example. The primary goal of the player projectile is to have a projectile that the player can spawn and use to destroy enemies throughout the level. In this chapter, you will set up the basic framework functionality, while in *Chapter 14, Spawning the Player Projectile*, you will add sound and visual effects. Let's get started by creating the player projectile class.

Exercise 13.11: Creating the Player Projectile

Up until this point, we have been working on in the Unreal Engine 4 editor to create our enemy AI. For the player projectile, we will be using C++ and Visual Studio to create this new class. The player projectile will allow the player to destroy enemies that are placed in the level. This projectile will have a short lifespan, travel at a high speed, and will collide with both enemies and the environment.

The goal of this exercise is to set up the base actor class for the player projectile and begin outlining the functions and components needed in the header file for the projectile.

The following steps will help you complete this exercise:

1. First, you will need to create a new C++ class using the **Actor** class as the parent class for the player projectile.
Next, name this new actor class **PlayerProjectile** and *left-click* on the **Create Class** option at the bottom-right of the menu prompt.

After creating the new class, Visual Studio will generate the required source and header files for the class and open these files for you. The actor base class comes included with a handful of default functions that you will not need for the player projectile.

2. Find the following lines of code inside the **PlayerProjectile.h** file and remove them:

```
protected:  
    // Called when the game  
    starts or when spawned  
  
    virtual void BeginPlay()  
override;  
  
public:  
    // Called every frame  
  
    virtual void Tick(float  
DeltaTime) override;
```

These lines of code represent the declarations of the **Tick()** and **BeginPlay()** functions that are included in every Actor-based class by default. The **Tick()** function is called on every frame and allows you to perform logic on every frame, which can get expensive, depending on what you are trying to do. The **BeginPlay()** function is called when this actor is initialized and play has started. This can be used to perform logic on the actor as soon as it enters the world. These functions are being removed because they are not required for **Player Projectile** and will just clutter the code.

3. After removing these lines from the **PlayerProjectile.h** header file, you can now remove the following lines from the **PlayerProjectile.cpp** source files as well:

```
// Called when the game
starts or when spawned

void
APlayerProjectile::BeginPla
y()

{
    Super::BeginPlay();

}

// Called every frame

void
APlayerProjectile::Tick(flo
at DeltaTime)

{
    Super::Tick(DeltaTime);

}
```

These lines of code represent the function implementations of the two functions you removed in the previous step; that is, **Tick()** and

BeginPlay(). Again, these are being removed because they serve no purposes for **Player Projectile** and just add clutter to the code. Additionally, without the declarations inside the **PlayerProjectile.h** header file, you would receive a compilation error if you were to try to compile this code as is. The only remaining function will be the constructor for the projectile class, which you will use to initialize the components of the projectile in the next exercise. Now that you have removed the unnecessary code from the **PlayerProjectile** class, let's add the functions and components required for the projectile.

4. Inside the **PlayerProjectile.h** header file, add the following components. Let's discuss these components in detail:

```
public:  
    //Sphere collision  
    component  
  
    UPROPERTY(VisibleDefaults
```

```
Only, Category =
Projectile)

    class USphereComponent*
CollisionComp;

private:

    //Projectile movement
component

    UPROPERTY(VisibleAnywhere
, BlueprintReadOnly,
Category = Movement, meta =
(AllowPrivateAccess =
"true"))

    class
UProjectileMovementComponen
t* ProjectileMovement;

    //Static mesh component

    UPROPERTY(VisibleDefaults
Only, Category =
Projectile)

    class
UStaticMeshComponent*
MeshComp;
```

There are three different components you are adding here. The first is the collision component, which you will

use for the projectile to recognize collisions with enemies and environment assets. The next component is the projectile movement component, which you should be familiar with from the last project. This will allow the projectile to behave like a projectile. The final component is the static mesh component. You will use this to give this projectile a visual representation so that it can be seen in-game.

5. Next, add the following function signature code to the **PlayerProjectile.h** header file, under the **public** access modifier:

```
UFUNCTION()  
void  
OnHit(UPrimitiveComponent*  
HitComp, AActor*  
OtherActor,  
UPrimitiveComponent*  
OtherComp, FVector  
NormalImpulse, const  
FHitResult& Hit);
```

This final event declaration will allow

the player projectile to respond to **OnHit** events from the **CollisionComp** component you created in the previous step.

6. Now, in order to have this code compile, you will need to implement the function from the previous step in the **PlayerProjectile.cpp** source file. Add the following code:

```
void  
APlayerProjectile::OnHit(UP  
rimitiveComponent* HitComp,  
AActor* OtherActor,  
UPrimitiveComponent*  
OtherComp, FVector  
NormalImpulse, const  
FHitResult& Hit)  
  
{  
}  
}
```

The **OnHit** event provides you with a lot of information about the collision that takes place. The most important parameter that you will be working with in the next exercise is the **OtherActor** parameter. The **OtherActor** parameter will tell you

the actor in which this **OnHit** event is responding to. This will allow you to know if this other actor was an enemy. You will use this information to destroy the enemies when the projectile hits them.

7. Lastly, navigate back into the Unreal Engine editor and *left-click* the **Compile** option to compile the new code.

With this exercise complete, you now have the framework ready for the **Player Projectile** class. The class has the required components for **Projectile Movement**, **Collision**, and **Static Mesh**, as well as the event signature ready for the **OnHit** collision so that the projectile can recognize collisions with other actors.

In the next exercise, you will continue to customize and enable parameters for **Player Projectile** so that it behaves the way you need it to for the **SuperSideScroller** project.

Exercise 13.12: Initializing Player Projectile Settings

Now that the framework of the **PlayerProjectile** class is in place, it's time to update the constructor of this class with the default settings needed for the projectile so that it moves and behaves as you want it to. In order to do this, you will

need to initialize the **Projectile Movement, Collision**, and **Static Mesh** components.

The following steps will help you complete this exercise:

1. Open Visual Studio and navigate to the **PlayerProjectile.cpp** source file.
2. Before adding any code to the constructor, include the following files inside the **PlayerProjectile.cpp** source file:

```
#include  
"GameFramework/ProjectileMovementComponent.h"  
  
#include  
"Components/SphereComponent.h"  
  
#include  
"Components/StaticMeshComponent.h"
```

These header files will allow you to initialize and update the parameters of the projectile movement component, the sphere collision component, and the static mesh component respectively. Without these files included, the **PlayerProjectile**

class wouldn't know how to handle these components and how to access their functions and parameters.

3. By default, the **APlayerProjectile::APlayerProjectile()** constructor function includes the following line:

```
PrimaryActorTick.bCanEverTick = true;
```

This line of code can be removed entirely because it is not required in the player projectile.

4. In the **PlayerProjectile.cpp** source file, add the following lines to the **APlayerProjectile::APlayerProjectile()** constructor:

```
CollisionComp =
CreateDefaultSubobject
<USphereComponent>
(TEXT("SphereComp"));

CollisionComp-
>InitSphereRadius(15.0f);

CollisionComp-
>BodyInstance.SetCollisionP
```

```
rofileName("BlockAll");  
  
CollisionComp-  
>OnComponentHit.AddDynamic(  
this,  
&APlayerProjectile::OnHit);
```

The first line initializes the sphere collision component and assigns it to the **CollisionComp** variable you created in the previous exercise.

Sphere Collision Component has a parameter called **InitSphereRadius**. This will determine the size, or radius, of the collision actor by default; in this case, a value of **15.0f** works well. Next, set **Collision Profile Name** for the collision component to **BlockAll** so that the collision profile is set to **BlockAll**, which means this collision component will respond to **OnHit** when it collides with other objects. Lastly, the last line you added allows the **OnComponentHit** event to respond with the function you created in the previous exercise:

```
void  
APlayerProjectile::OnHit(UP
```

```
    primitiveComponent* HitComp,  
    AActor* OtherActor,  
    UPrimitiveComponent*  
    OtherComp, FVector  
    NormalImpulse, const  
    FHitResult& Hit)  
  
{  
  
}
```

This means that when the collision component receives the **OnComponentHit** event from a collision event, it will respond with that function; however, this function is empty at the moment. You will add code to this function later in this chapter.

5. The last thing to do with **Collision Component** is to set this component as the **root** component of the player projectile actor. Add the following line of code in the constructor, after the lines from *Step 4*:

```
// Set as root component  
  
RootComponent =  
CollisionComp;
```

6. With the collision component set up and ready, let's move on to the **Projectile Movement** component.

Add the following lines to the constructor:

```
// Use a  
ProjectileMovementComponent  
to govern this projectile's  
movement  
  
ProjectileMovement =  
CreateDefaultSubobject<UProj  
ectileMovementComponent>  
(TEXT("ProjectileComp")) ;  
  
ProjectileMovement -  
>UpdatedComponent =  
CollisionComp;  
  
ProjectileMovement -  
>ProjectileGravityScale =  
0.0f;  
  
ProjectileMovement -  
>InitialSpeed = 800.0f;  
  
ProjectileMovement -  
>MaxSpeed = 800.0f;
```

This first line initializes **Projectile Movement Component** and assigns it

to the **ProjectileMovement** variable you created in the previous exercise.

Next, we set **CollisionComp** as the updated component of the projectile movement component. The reason we're doing this is because the **Projectile Movement** component will use the **root** component of the actor as the component to move. Then, you are setting the gravity scale of the projectile to **0.0f** because the player projectile should not be affected by gravity; the behavior should allow the projectile to travel at the same speed, at the same height, and not be influenced by gravity. Lastly, you are setting both the **InitialSpeed** and **MaxSpeed** parameters to **500.0f**.

This will allow the projectile to instantly start moving at this speed and remain at this speed for the duration of its lifetime. The player projectile will not support any kind of acceleration motion.

7. With the projectile movement component initialized and set up, it is time to do the same for **Static Mesh**

Component. Add the following code after the lines from the previous step:

```
MeshComp =  
CreateDefaultSubobject<UStaticMeshComponent>  
(TEXT("MeshComp"));  
  
MeshComp-  
>AttachToComponent(RootComponent,  
FAttachmentTransformRules::  
KeepWorldTransform);
```

This first line initializes **Static Mesh Component** and assigns it to the **MeshComp** variable you created in the previous exercise. Then, you attach this static mesh component to **RootComponent** using a struct called **FAttachmentTransformRules** to ensure that the **Static Mesh Component** keeps its world transform during the attachment which is **CollisionComp** from *Step 5* of this exercise.

Note

You can find more information about

the FAttachmentTransformRules

struct here:

<https://docs.unrealengine.com/en-US/API/Runtime/Engine/Engine/FAttachmentTransformRules/index.html>.

8. Lastly, let's give **Player**

Projectile an initial life span of **3** seconds so that the projectile will automatically be destroyed if it doesn't collide with anything after this time. Add the following code to the end of the constructor:

```
InitialLifeSpan = 3.0f;
```

9. Lastly, navigate back into the Unreal Engine editor and *left-click* the **Compile** option to compile the new code.

By completing this exercise, you have set up the groundwork for **Player Projectile** so that it can be created as a *Blueprint* actor inside the editor. All three required components are initialized and contain the default parameters that you want for this projectile. All we need to do now is create the *Blueprint* from this class to see it in the level.

Activity 13.03: Creating

the Player Projectile Blueprint

To conclude this chapter, you will create the **Blueprint** actor from the new **PlayerProjectile** class and customize this actor so that it uses a placeholder shape for **Static Mesh Component** for debugging purposes. This allows you to view the projectile in the game world. Then, you will add a **UE_LOG()** function to the **APlayerProjectile::OnHit** function inside the **PlayerProjectile.cpp** source file so that you can ensure that this function is called when the projectile comes into contact with an object in the level. You will need to perform the following steps:

1. Inside the **Content Browser** interface, create a new folder called **Projectile** in the **/MainCharacter** directory.
2. In this directory, create a new Blueprint from the **PlayerProjectile** class, which you created in *Exercise 13.11, Creating the Player Projectile*. Name this Blueprint **BP_PlayerProjectile**.
3. Open **BP_PlayerProjectile** and navigate to its components. Select the **MeshComp** component to access its settings.

4. Add the **Shape_Sphere** mesh to the Static Mesh parameter of the **MeshComp** component.
5. Update the Transform of **MeshComp** so that it fits the **Scale and Location of the CollisionComp** component. Use the following values:

Location:
(X=0.000000, Y=0.000000, Z=-1
0.000000)

Scale:
(X=0.200000, Y=0.200000, Z=0.
200000)
6. Compile and save the **BP_PlayerProjectile** Blueprint.
7. Navigate to the **PlayerProjectile.cpp** source file in Visual Studio and find the **APlayerProjectile::OnHit** function.
8. Inside the function, implement the **UE_LOG** call so that the logged line is of **LogTemp, Warning log level**, and displays the text **HIT. UE_LOG was**

covered back in *Chapter 11, Blend Spaces 1D, Key Bindings, and State Machines*.

9. Compile your code changes and navigate to the level where you placed the **BP_PlayerProjectile** actor in the previous exercise. If you don't have this actor added to the level, do so now.
10. Before testing, make sure to open the Output Log in the **Window** option. From the **Window** dropdown, hover over the **Developers Tools** option and *left-click* to select **Output Log**.
11. Use **PIE** and watch out for the log warning inside **Output Log** when the projectile collides with something.

The following is the expected output:

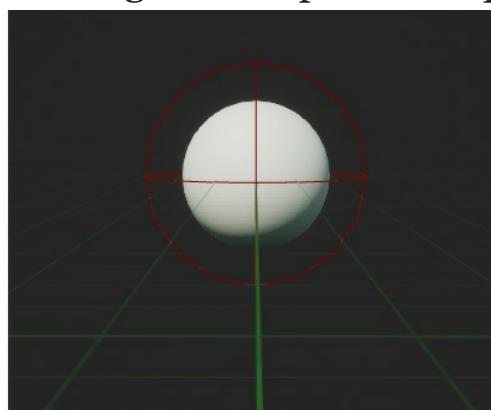


Figure 13.39: Scale of the MeshComp better fits the

size of the Collision Comp

The log warning should be as follows:

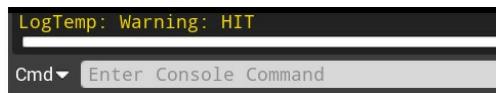


Figure 13.40: When the projectile hits an object, the text HIT is shown in the Output Log

With this final activity complete, **Player Projectile** is ready for the next chapter, where you will spawn this projectile when the player uses the **Throw** action. You will update the **APlayerProjectile::OnHit** function so that it destroys the enemy that it collides with and becomes an effective offensive tool for the player to use against the enemies.

Note

The solution to this activity can be found at:

<https://packt.live/338jEBx>.

Summary

In this chapter, you learned how to use the different aspects of the AI tools offered by Unreal Engine 4, including Blackboards, behavior trees, and AI Controllers. With a combination of both custom created Tasks and default Tasks provided by Unreal Engine 4, and with a Decorator, you were able to have the enemy AI navigate within the bounds of the Nav Mesh you added to your own level.

On top of this, you created a new Blueprint actor that allows you to add patrol points with the use of a **Vector** array variable. You then added a new function to this actor that

selects one of these points at random, converts its location from local space into world space, and then returns this new value for use by the enemy character.

With the ability to randomly select a Patrol Point, you updated the custom **BTask_FindLocation** task to find and move to the selected Patrol Point, allowing the enemy to move from each Patrol Point at random. This brought the enemy AI character to a whole new level of interaction with the player and the environment.

Lastly, you created the Player Projectile that the player will be able to use in order to destroy enemies within the environment. You took advantage of both **Projectile Movement Component** and **Sphere Component** to allow for both projectile movement and to recognize and respond to collisions within the environment.

With the Player Projectile in a functional state, it is time to move on to the next chapter, where you will use **Anim Notifies** to spawn the projectile when the player uses the **Throw** action.

14. Spawning the Player Projectile

Overview

*In this chapter, you will learn about **Anim Notifies** and **Anim States**, which can be found inside Animation Montages. You will code your own **Anim Notify** using C++ and implement this notify in the **Throw** Animation Montage. Lastly, you will learn about Visual and Audio Effects, and how these effects are used in games.*

*By the end of this chapter, you will be able to play Animation Montages in both Blueprints and C++ and know how to spawn objects into the game world using C++ and the **UWorld** class. These elements of the game will be given audio and visual components as an added layer of polish, and your **SuperSideScroller** player character will be able to throw projectiles that destroy enemies.*

Introduction

In the previous chapter, you made great progress with the enemy character's AI by creating a behavior tree that would allow the enemy to randomly select points from the **BP_AIPoints** actor you created. This gives the **SuperSideScroller** game more life as you can now have multiple enemies moving around your game world. Additionally, you learned the different tools available in

Unreal Engine 4 that are used together to make artificial intelligence of various degrees of complexity. These tools included the **Navigation Mesh**, behavior trees, and Blackboards.

Now that you have enemies running around your level, you need to allow the player to defeat these enemies with the player projectile you started to create at the end of the previous chapter.

In this chapter, you will learn how to use the **UAnimNotify** class to spawn the player projectile at a specific frame of the **Throw** Animation Montage. You will also learn how to add this new notify to the Montage itself, and how to add a new **Socket** to the main character skeleton from which the projectile will spawn. Lastly, you will learn how to use **Particle Systems** and **SoundCues** to add a layer of visual and audio polish to the game.

Let's begin this chapter by learning about **Anim Notifies** and **Anim Notify States**. After that, you'll get your hands dirty by creating your own **UAnimNotify** class so that you can spawn the player projectile during the **Throw** Animation Montage.

Anim Notifies and Anim Notify States

When it comes to creating polished and complex animations, there needs to be a way for animators and programmers to add custom events within the animation that will allow for additional effects, layers, and functionality to occur. The solution in Unreal Engine 4 is to use **Anim Notifies** and **Anim Notify States**.

The main difference between **Anim Notify** and **Anim Notify State** is that **Anim Notify State** possesses three distinct events that **Anim Notify** does not. These events are **Notify Begin**, **Notify End**, and **Notify Tick**, all of which can be used in Blueprints or C++. When it comes to these events, Unreal Engine 4 secures the following behaviors:

- **Notify State** will always start with **Notify Begin Event**.
- **Notify State** will always finish with **Notify End Event**.
- **Notify Tick Event** will always take place between the **Notify Begin** and **Notify End** events.

Anim Notify, however, is a much more simplified version that uses just a single function, **Notify()**, to allow programmers to add functionality to the notify itself. It works with the mindset of *fire and forget*, meaning you don't need to worry about what happens at the start, end, or anywhere in-between the **Notify()** event. It is due to this simplicity of **Anim Notify**, and due to the fact that we do not need the events included with **Anim Notify State** that we will use **Anim Notify** to spawn the player projectile for the Super Side-Scroller game.

Before moving on to the following exercise, where you will create your own custom **Anim Notify** in C++, let's briefly discuss some examples of existing **Anim Notifies** that Unreal Engine 4 provides by default. A full list of default **Anim Notifies** states can be seen in the following screenshot:

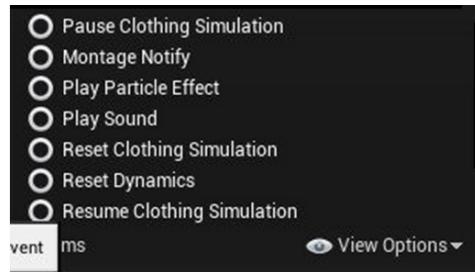


Figure 14.1: The full list of default Anim Notifies provided in Unreal Engine 4

There are two **Anim Notifies** that you will be using later on in this chapter: **Play Particle Effect** and **Play Sound**. Let's discuss these two in more detail so that you are familiar with them by the time you use them:

- **Play Particle Effect:** The **Play Particle Effect** notify, as the name suggests, allows you to spawn and play a particle system at a certain frame of your animation. As shown in the following screenshot, you have options to change the VFX being used, such as updating the **location**, **rotation**, and **scale** settings of the particle. You can even attach the particle to a specified **Socket Name** if you so choose:

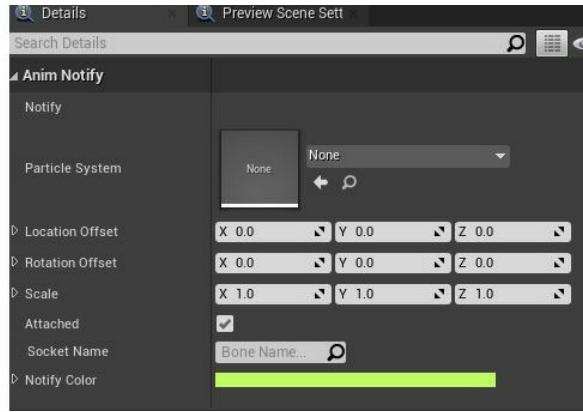


Figure 14.2: The Details panel of the Play Particle Effect notify, which allows you to customize the particle

Note

Visual Effects, or VFX for short, are crucial elements for any game. Visual Effects, in Unreal Engine 4, are created using a tool called Cascade, inside the editor. Since Unreal Engine version 4.20, a new tool called Niagara has been introduced as a free plugin to improve the quality and pipeline for how VFX are made. You can learn more about Niagara here:

<https://docs.unrealengine.com/en-US/Engine/Niagara/Overview/index.html>.

A very common example used in games is to use this type of notify to spawn dirt or other effects underneath the player's feet while they walk or run. Having the ability to specify at which frame of the animation these effects spawn is very powerful and allows you to create convincing effects for your character.

- **Play Sound:** The **Play Sound** notify allows you to play a **Soundcue** or **Soundwave** at a certain frame of your

animation. As shown in the following screenshot, you have options to change the sound being used, update its **volume** and **pitch** values, and even have the sound follow the owner of the sound via attaching it to a specified

Socket Name:

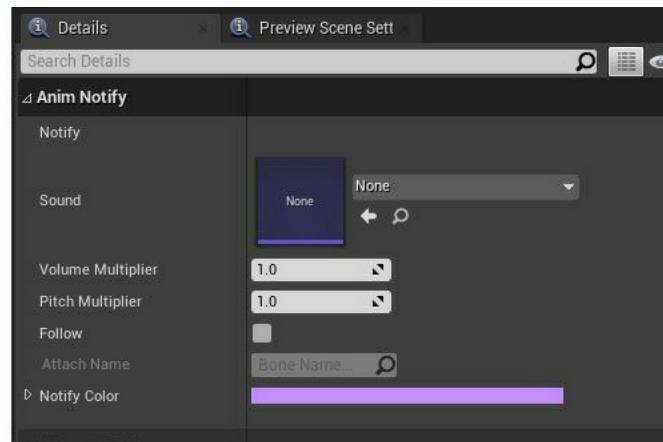


Figure 14.3: The Details panel of the Play Sound notify, which allows you to customize the sound

Much like the example given for the **Play Particle Effect** notify, the **Play Sound** notify can also be commonly used to play the sounds of footsteps while the character is moving. By having control of exactly where on the animation timeline you can play a sound, it is possible to create believable sound effects.

Although you will not be using **Anim Notify States**, it is still important to at least know the options that are available to you by default, as shown in the following screenshot:

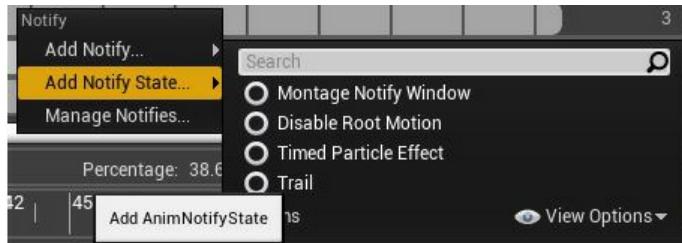


Figure 14.4: The full list of default Anim Notify States provided to you in Unreal Engine 4

Note

The two **Notify** states that are not available in animation sequences are the *Montage Notify Window* and *Disable Root Motion* states, as shown in the preceding screenshot. For more information regarding notifies, refer to the following documentation: docs.unrealengine.com/en-US/Engine/Animation/Sequences/Notifies/index.html.

Now that you are more familiar with **Anim Notify** and **Anim Notify State**, let's move on to the next exercise, where you will create your own custom **Anim Notify** in C++ that you will use to spawn the player projectile.

Exercise 14.01: Creating a UAnim Notify Class

The main offensive ability that the player character will have in the **SuperSideScroller** game is the projectile that the player can throw at enemies. In the previous chapter, you set up the framework and base functionality of the projectile, but right now, there is no way for the player to use it. In order to make spawning, or throwing, the projectile convincing to the eye, you need to create a custom **Anim Notify** that you will then add to the **Throw** Animation Montage. This **Anim**

Notify will let the player know it's time to spawn the projectile.

Do the following to create the new **UAnimNotify** class:

1. Inside Unreal Engine 4, navigate to the **File** option and *left-click* to select the option for **New C++ Class**.
2. From the **Choose Parent Class** dialogue window, search for **AnimNotify** and *left-click* the **AnimNotify** option. Then, *left-click* the **Next** option to name the new class.
3. Name this new class **Anim_ProjectileNotify**. Once named, *left-click* to select the **Create Class** option so that Unreal Engine 4 recompiles and hot-reloads the new class in Visual Studio. Once Visual Studio opens, you will have both the header file, **Anim_ProjectileNotify.h**, and the source file, **Anim_ProjectileNotify.cpp**, available to you.
4. The **UAnimNotify** base class has one function that needs to be implemented

inside your class:

```
virtual void  
Notify(USkeletalMeshComponent* MeshComp,  
UAnimSequenceBase* Animation);
```

This function is called automatically when the notify is hit on the timeline it is being used in. By overriding this function, you will be able to add your own logic to the notify. This function also gives you access to both the **Skeletal Mesh** component of the owning notify and the Animation Sequence currently being played.

5. Next, let's add the override declaration of this function to the header file. In the header file

Anim_ProjectileNotify.h, add the following code underneath the **GENERATED_BODY()**:

```
public: virtual void  
Notify(USkeletalMeshComponent*  
MeshComp, UAnimSequenceBase*  
Animation) override;
```

Now that you added the function to the header file, it is time to define the function inside the **Anim_ProjectileNotify** source file.

6. Inside the **Anim_ProjectileNotify.cpp** source file, define the function and add a **UE_LOG()** call that prints the text "**Throw Notify**", as shown in the following code:

```
void  
UAnim_ProjectileNotify::Notify(USkeletalMeshComponent*  
MeshComp,  
UAnimSequenceBase*  
Animation)  
{  
    UE_LOG(LogTemp, Warning,  
    TEXT("Throw Notify"));  
}
```

For now, you will just use this **UE_LOG()** debugging tool to know that this function is correctly being called when you add this notify to the **Throw**

Animation Montage in the next exercise.

In this exercise, you created the groundwork necessary to implement your own **AnimNotify** class by adding the following function:

```
Notify(USkeletalMeshComponent* MeshComp,  
UAnimSequenceBase* Animation)
```

Inside this function, you are using **UE_LOG()** to print the custom text "**Throw Notify**" in the output log so that you know that this notify is working correctly.

Later in this chapter, you will update this function so that it calls logic that will spawn the player projectile, but first, let's add the new notify to the **Throw** Animation montage.

Exercise 14.02: Adding the Notify to the Throw Montage

Now that you have your **Anim_ProjectileNotify** notify, it is time to add this notify to the **Throw** Animation Montage so that it can actually be of use to you.

In this exercise, you will add **Anim_ProjectileNotify** to the timeline of the **Throw** Montage at the exact frame of the animation that you'd expect the projectile to spawn.

Complete the following steps to achieve this:

1. Back inside Unreal Engine, navigate to

the **Content Browser** interface and go to the **/MainCharacter/Animation/** directory. Inside this directory, *double-click* the **AM_Throw** asset to open the **Animation Montage** editor.

At the very bottom of the **Animation Montage** editor, you will find the timeline for the animation. By default, you will observe that the *red colored bar* will be moving along the timeline as the animation plays.

2. *Left-click* this **red** bar and manually move it to the 22nd **frame**, as close as you can, as shown in the following screenshot:



Figure 14.5: The red colored bar allows you to manually position notifications anywhere on the timeline

The 22nd frame of the **Throw** animation is the exact moment in the throw that you would expect a projectile to spawn and be thrown by the player. The following screenshot

shows the frame of the throw animation, as seen inside the editor within **Persona**:



Figure 14.6: The exact moment the player projectile should spawn

3. Now that you know the position on the timeline that the notify should be played, you can now *right-click* on the thin **red** line within the **Notifies** timeline.

This will show you a popup where you can add a **Notify** or a **Notify State**. In some cases, the **Notifies** timeline may be collapsed and hard to find; simply left-click on the word **Notifies** to toggle between collapsed and expanded.

4. Select **Add Notify** and, from the

options provided, find and select **Anim Projectile Notify**.

5. After selecting to add **Anim Projectile Notify** to the Notifies timeline, you will see the following:



Figure 14.7:
Anim_ProjectileNotify
successfully added to the Throw
Animation Montage

6. With the **Anim_ProjectileNotify** notify in place on the **Throw** Animation Montage timeline, save the montage.
7. If the **Output Log** window is not visible, please re-enable the window by navigating to the **Window** option and hover over it for **Developer Tools**. Find the option for **Output Log** and *left-click* to enable it.
8. Now, use **PIE**, and, once in-game, use the *left mouse button* to start playing the **Throw** montage.

At the point in the animation where you added the notify, you will now see the debugging log text **Throw Notify** appear in the output log.

As you may recall from *Chapter 12, Animation Blending and Montages*, you added the **Play Montage** function to the player character blueprint, **BP_SuperSideScroller_MainCharacter**. For the sake of learning C++ in the context of Unreal Engine 4, you will be moving this logic from Blueprint to C++ in the upcoming exercises. This is so that we don't rely too heavily on Blueprint scripts for the base behavior of the player character.

With this exercise complete, you have successfully added your custom **Anim Notify** class, **Anim_ProjectileNotify**, to the **Throw** Animation Montage. This notify was added at the precise frame in which you expect a projectile to be thrown from the player's hand. Since you added the Blueprint logic to the player character in *Chapter 12, Animation Blending and Montages*, you are able to play this **Throw** Animation Montage when the **InputAction** event, **ThrowProjectile**, is called when using the *left mouse button*. Before making the transition from playing the Throw Animation Montage in Blueprints to playing the Montage from C++, let's discuss playing Animation Montages some more.

Playing Animation Montages

As you learned in *Chapter 12, Animation Blending and Montages*, these items are useful for allowing animators to combine individual animation sequences into one complete montage. By splitting the Montage into its own unique sections and adding notifies for particles and sound,

animators and animation programmers can make complex sets of montages that handle all the different aspects of the animation.

But once the Animation Montage is ready, how do we play this Montage on a character? You are already familiar with the first method, which is via Blueprints.

Playing Animation Montages in Blueprints

In Blueprints, the **Play Montage** function is available for you to use, as shown in the following screenshot:



Figure 14.8: The Play Montage function in Blueprints

You have already used the function to play the **AM_Throw** Animation Montage. This function requires the **Skeletal Mesh** component that the Montage must be played on, and it requires the Animation Montage to play.

The remaining parameters are optional, depending on how your Montage will work. Let's have a quick look at these parameters:

- **Play Rate:** The **Play Rate** parameter allows you to increase or decrease the playback speed of the Animation Montage. For faster playback, you would increase this value; otherwise, you would decrease the value for slower playback speed.
- **Starting Position:** The **Starting Position** parameter allows you to set the starting position, in seconds, along the Montage timeline from which the Montage will start playing. For example, in an Animation Montage that has a 3-second timeline, you could choose to have the Montage start at the **1.0f** position instead of at **0.0f**.
- **Starting Section:** The **Starting Section** parameter allows you to tell the Animation Montage to start at a specific section. Depending on how your Montage is set up, you could have multiple sections created for different parts of the montage. For example, a shotgun weapon reloading Animation Montage would include a section for the initial movement for reload, a

looped section for the actual bullet reload, and a final section for re-equipping the weapon so that it is ready to fire again.

When it comes to the outputs of the **Play Montage** function, you have a few different options:

- **On Completed:** The **On Completed** output is called when the Animation Montage has finished playing and has been fully blended out.
- **On Blend Out:** The **On Blend Out** output is called when the Animation Montage begins to blend out. This can occur during **Blend Out Trigger Time**, or if the Montage ends prematurely.
- **On Interrupted:** The **On Interrupted** output is called when the Montage begins to blend out due to this Montage being interrupted by another Montage that is trying to play on the same skeleton.
- **On Notify Begin & On Notify End:** Both the **On Notify Begin** and **On Notify End** outputs are called if

you are using the **Montage Notify** option under the **Notifies** category in the Animation Montage. The name given to the **Montage Notify** is returned via the **Notify Name** parameter.

Playing Animation Montages in C++

On the C++ side, there is only one thing you need to know about, and that is the **UAnimInstance::Montage_Play()** function. This function requires the Animation Montage to play, the play rate in which to play back the montage, a value of the **EMontagePlayReturnType** type, a **float** value for determining the start position to play the montage, and a **Boolean** value for determining whether playing this Montage should stop or interrupt all montages.

Although you will not be changing the default parameter of **EMontagePlayReturnType**, which is **EMontagePlayReturnType::MontageLength**, it is still important to know the two values that exist for this enumerator:

- **Montage Length:** The **Montage Length** value returns the length of the Montage itself, in seconds.
- **Duration:** The **Duration** value returns the play duration of the

montage, which is equal to the length of the montage, divided by the play rate.

Note

*For more details regarding the **UAnimMontage** class, please refer to the following documentation:
<https://docs.unrealengine.com/en-US/API/Runtime/Engine/Animation/UAnimMontage/index.html>.*

You will learn more about the C++ implementation of playing an Animation Montage in the next exercise.

Exercise 14.03: Playing the Throw Animation in C++

Now that you have a better understanding of play Animation Montages in Unreal Engine 4, both via Blueprints and C++, it is time to migrate the logic for playing the **Throw** Animation Montage from Blueprints to C++. The reason behind this change is because the Blueprint logic was put into place as a placeholder method so that you could preview the **Throw** montage. This book is a more heavily focused C++ guide to game development, and as such, it is important to learn how to implement this logic in code.

Let's begin by removing the logic from Blueprints, and then move on to recreating the logic in C++ inside the player character class.

The following steps will help you complete this exercise:

1. Navigate to the player character Blueprint, **BP_SuperSideScroller_MainCharacter**, which can be found in the following directory:
/MainCharacter/Blueprints/.
Double-click this asset to open it.
2. Inside this Blueprint, you will find the **InputAction ThrowProjectile** event and the **Play Montage** function that you created to preview the **Throw** Animation Montage, as shown in the following screenshot. Delete this logic and then recompile and save the player character Blueprint:

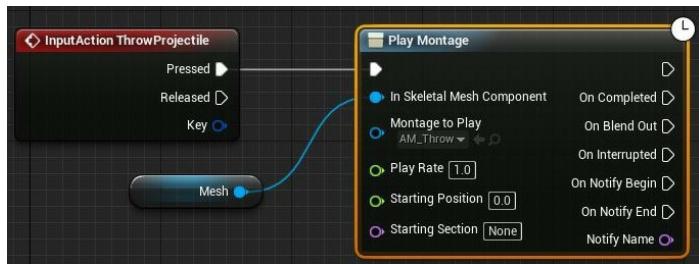


Figure 14.9: You no longer need this placeholder logic inside the player character Blueprint

3. Now, use **PIE** and attempt to throw with the player character by using the *left mouse button*. You will observe that the player character no longer plays the **Throw** Animation Montage. Let's fix this by adding the required logic in C++.
4. Open up the header file for the player character in Visual Studio, **SuperSideScroller_Player.h**.
5. The first thing you need to do is create a new variable for the player character that will be used for the **Throw** animation. Add the following code under the **Private** access modifier:

```
UPROPERTY(EditAnywhere)  
class UAnimMontage*  
ThrowMontage;
```

Now that you have a variable that will represent the **Throw** Animation Montage, it is time to add the logic for playing the Montage inside the **SuperSideScroller_Player.cpp** file.
6. Before you can make the call to

UAnimInstance::Montage_Play()

, you need to add the following **include** directory to the existing list at the top of the source file in order to have access to this function:

```
#include  
"Animation/AnimInstance.h"
```

As we know from *Chapter 9, Audio-Visual Elements*, the player character already has a function called **ThrowProjectile** that is called whenever the *left mouse button* is pressed. As a reminder, this is where the binding occurs in C++:

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

7. Update **ThrowProjectile** so that it plays **ThrowMontage**, which you set up earlier in this exercise. Add the following code to the

ThrowProjectile() function. Then, we can discuss what is happening here:

```
void
ASuperSideScroller_Player::
ThrowProjectile( )

{
    if (ThrowMontage)

    {
        bool bIsMontagePlaying
= GetMesh()->GetAnimInstance()->
Montage_IsPlaying(ThrowMont
age);

        if (!bIsMontagePlaying)

        {
            GetMesh()->GetAnimInstance()->
Montage_Play(ThrowMontage,
2.0f);

        }
    }
}
```

The first line is checking if the **ThrowMontage** is valid; if we don't have a valid Animation Montage

assigned, there is no point in continuing the logic, and also it can be dangerous to use a NULL object in further function calls as it could result in a crash. Next, we are declaring a new Boolean variable, called **bIsMontagePlaying**, that determines whether **ThrowMontage** is already playing on the player character's skeletal mesh. This check is made because the **Throw** Animation Montage should not be played while it is already playing; this will cause the animation to break if the player repeatedly presses the *left mouse button*.

Next, there is an **If** statement that checks that **ThrowMontage** is valid and that the Montage is not playing. As long as these conditions are met, it is safe to move on and play the Animation Montage.

8. Inside the **If** statement, you are telling the player's skeletal mesh to play the **ThrowMontage** Animation Montage with a play rate of **1.0f**. The **1.0f** value is used so that the Animation

Montage plays back at the speed it is intended to. Values larger than **1.0f** will make the Montage play back faster, while values lower than **1.0f** will make the Montage play back slower. The other parameters that you learned about, such as the start position or the **EMontagePlayReturnType** parameter, can be left at their **defaults.Head**. Back inside the Unreal Engine 4 editor, perform a recompile of the code, as you have done in the past.

9. After the code recompiles successfully, navigate back to the player character blueprint, **BP_SuperSideScroller_MainCharacter**, which can be found in the following directory:
/MainCharacter/Blueprints/.
Double-click this asset to open it.
10. In the **Details** panel of the player character, you will now see the **Throw Montage** parameter that you added.
11. *Left-click* on the drop-down menu for

the **Throw Montage** parameter to find the **AM_Throw** montage. *Left-click* again on the **AM_Throw** option to select it for this parameter. Please refer to the following screenshot to see how the variable should be set up:

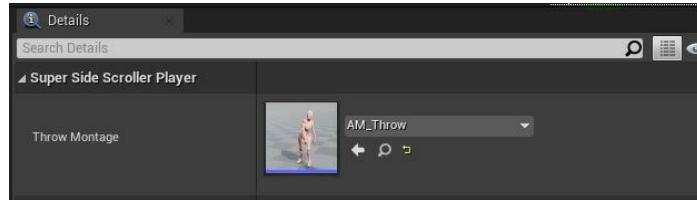


Figure 14.10: Now, the Throw Montage is assigned the AM_Throw montage

12. Recompile and save the player character blueprint. Then, use **PIE** to spawn the player character and use the *left mouse button* to play **Throw Montage**. The following screenshot shows this in action:

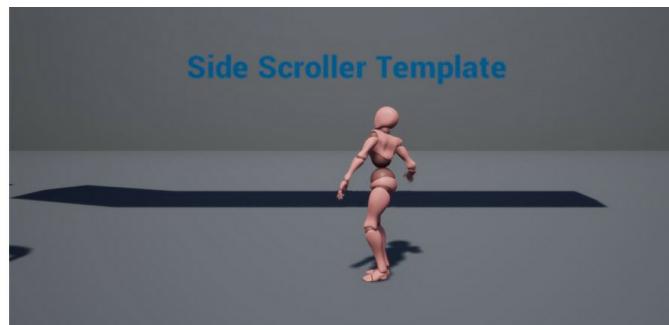


Figure 14.11: The player character is now able to perform the Throw animation again

By completing this exercise, you have learned how to add an **Animation Montage** parameter to the player character, as well as how to play the Montage in C++. In addition to playing **Throw** Animation Montage in C++, you also added the ability to control how often the **Throw** animation can be played by adding the check for whether the Montage is already playing. By doing this, you prevent the player from spamming the **Throw** input and causing the animation to break or not play entirely.

Note

*Try setting the play rate of **Animation Montage** from **1.0f** to **2.0f** and recompile the code. Observe how increasing the play rate of the animation affects how the animation looks and feels for the player.*

Game World and Spawning Objects

When it comes to spawning objects into the game world, it is actually the **World** object that represents your level that handles the creation of said objects. You can think of the **UWorld** class object as the single, top-level object that represents your level.

The **UWorld** class can do many things, such as spawning and removing objects from the world, detect when levels are being changed or streamed in/out, and even perform line traces to assist with inter-object detection. For the sake of this chapter, we'll focus on spawning objects.

The **UWorld** class has multiple variations of the **SpawnActor()** function, depending on how you want to

spawn the object, or by which parameters you have access to in the context in which you are spawning this object. The three consistent parameters to take into consideration are the following:

- **UClass**: The **UClass** parameter is simply the class of the object that you want to spawn in.
- **FActorSpawnParameters**: This is a struct of variables that give the spawned object more context and references to what has spawned it. For a list of all of the variables included within this struct, please refer to this article from the Unreal Engine 4 Community Wiki:
<https://www.ue4community.wiki/Actor#Spawn>

Let's briefly discuss one of the more crucial variables included in **FActorSpawnParameters**: the **Owner** actor. **Owner** is the actor that has spawned this object, and in the case of the player character and the projectile, it will be important for you to explicitly reference the player as the owner of the projectile. The reason behind this, especially in the context of

this game, is that you don't want the projectile to collide with its **Owner**; you want this projectile to ignore the owner entirely so that it can only collide with enemies or the level environment.

- **Transform:** When spawning an object into the world, the world needs to know the **location**, **rotation**, and **scale** properties of this actor before it can spawn it. In some templates of the **SpawnActor()** function, it requires a full **Transform** to be passed, while in other templates, **Location** and **Rotation** need to be passed in individually.

Before moving on to spawning the player projectile, let's set up the **Socket** location in the player character's **Skeleton** so that the projectile can spawn from the *players' hand* during the **Throw** animation.

Exercise 14.04: Creating the Projectile Spawn Socket

In order to spawn the player projectile, you need to determine the **Transform** in which the projectile will spawn while

primarily focusing on **Location** and **Rotation**, rather than **Scale**.

In this exercise, you will create a new **Socket** on the player character's **Skeleton** that you can then reference in code in order to obtain the location from which to spawn the projectile.

Let's get started:

1. Inside Unreal Engine 4, navigate to the **Content Browser** interface and find the **/MainCharacter/Mesh/** directory.
2. In this directory, find the **Skeleton** asset; that is, **MainCharacter_Skeleton.uasset**. *Double-click* to open this **Skeleton**.

To determine the best position for where the projectile should spawn, we need to add the **Throw Animation Montage** as the preview animation for the skeleton.

3. In the **Details** panel, under the **Animation** category, find the **Preview Controller** parameter and select the **Use Specific Animation** option.

4. Next, *left-click* on the drop-down menu to find and select the **AM_Throw** Animation Montage from the list of available animations.

Now, the player character's **Skeleton** will start previewing the **Throw** Animation Montage, as shown in the following screenshot:



Figure 14.12: The player character previewing the Throw Animation Montage

If you recall from *Exercise 14.02, Adding the Notify to the Throw Montage*, you added **Anim_ProjectileNotify** at the 22nd frame of the **Throw** animation.

5. Using the timeline at the bottom of the

Skeleton editor, move the **red** bar to as close to the **22nd** frame as you can.

Please refer to the following screenshot:



Figure 14.13: The same 22nd frame in which you added Anim_ProjectileNotify in an earlier exercise

At the **22nd** frame of the **Throw** animation, the player character should look as follows:



Figure 14.14: At the 22nd frame of the Throw Animation Montage, the character's hand is in position to release a projectile

As you can see, the player character will be throwing the projectile from their right hand, so the new **Socket** should be attached to the *right hand*. Let's take a look at the skeletal hierarchy of the player character, as shown in the following screenshot:

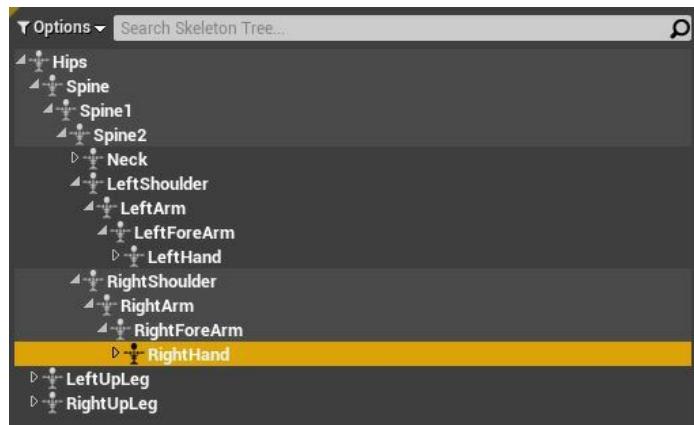


Figure 14.15: The RightHand bone found within the hierarchy of the player character's skeleton

6. From the skeletal hierarchy, find the **RightHand** bone. This can be found underneath the **RightShoulder** bone hierarchy structure.
7. *Right-click* on the **RightHand** bone and *left-click* the **Add Socket** option from the list of options that appear. Name this socket **ProjectileSocket**.

Also, when adding a new **Socket**, the hierarchy of the entire **RightHand** will expand and the new socket will appear at the bottom.

8. With **ProjectileSocket** selected, use the **Transform** widget gizmo to position this **Socket** at the following location:

Location =
(X=12.961717, Y=25.448450, Z=
-7.120584)

The final result should look as follows:

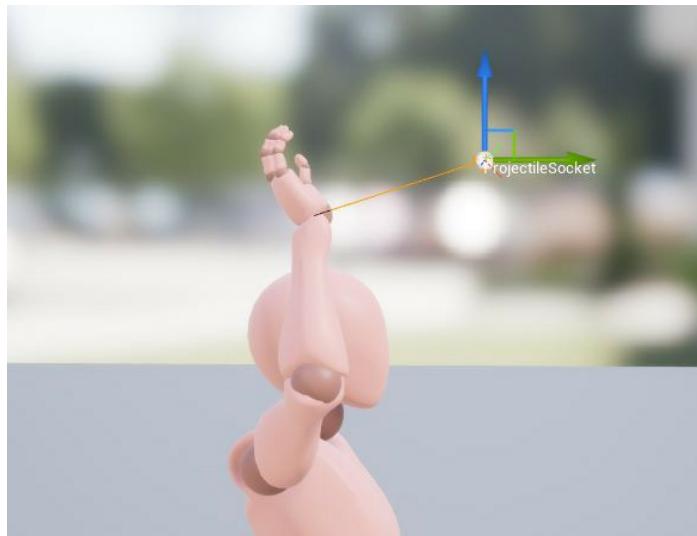


Figure 14.16: The final position of ProjectileSocket at the 22nd frame of the Throw Animation in world space.

If your gizmo looks a bit different, that is because the above image shows the socket location in world space, not local space.

9. Now that **ProjectileSocket** is positioned where you want it, save the **MainCharacter_Skeleton** asset.

With this exercise complete, you now know the location that the player projectile will spawn from. Since you used the **Throw Animation Montage** in the preview, and used the same 22nd frame of the animation, you know this position will be correct based on when **Anim_ProjectileNotify** will fire.

Now, let's move on to actually spawning the player projectile in C++.

Exercise 14.05: Preparing the SpawnProjectile() Function

Now that you have **ProjectileSocket** in place and there is now a location from which to spawn the player projectile, let's add the code necessary to spawn the player projectile.

By the end of this exercise, you will have the function ready to spawn the projectile and it will be ready to call from the **Anim_ProjectileNotify** class.

Perform the following steps:

1. From Visual Studio, navigate to the **SuperSideScroller_Player.h** header file.
2. You need a class reference variable to the **PlayerProjectile** class. You can do this using the variable template class type known as **TSubclassOf**. Add the following code to the header file, under the **Private** access modifier:

```
UPROPERTY(EditAnywhere)  
TSubclassOf<class  
APlayerProjectile>  
PlayerProjectile;
```

Now that you have the variable ready, it is time to declare the function you will use to spawn the projectile.

3. Add the following function declaration under the declaration of the void **ThrowProjectile()** function and the **Public** access modifier:

```
void SpawnProjectile();
```

4. Before preparing the definition of the **SpawnProjectile()** function, add

the following **include** directories to
the list of includes in the
SuperSideScroller_Player.cpp
source file:

```
#include  
"PlayerProjectile.h"  
  
#include "Engine/World.h"  
  
#include  
"Components/SphereComponent  
.h"
```

You need to include
PlayerProjectile.h because it is
required in order to reference the
collision component of the projectile
class. Next, use of the
Engine/World.h include is
necessary in order to use the
SpawnActor() function and access
the **FActorSpawnParameters** struct.
Lastly, you need to use the
Components/SphereComponent.h
include in order to update the collision
component of the player projectile so
that it will ignore the player.

5. Next, create the definition of the
SpawnProjectile() function at the

bottom of the
SuperSideScroller_Player.cpp
source file, as shown here:

```
void
ASuperSideScroller_Player::
SpawnProjectile( )

{
```

The first thing this function needs to do is check whether the **PlayerProjectile** class variable is valid. If this object is not valid, there is no point in continuing to try and spawn it.

6. Update the **SpawnProjectile()** function so that it looks as follows:

```
void
ASuperSideScroller_Player::
SpawnProjectile( )

{
    if(PlayerProjectile)

    {
        }

}
```

Now, if the **PlayerProjectile** object is valid, you'll want to obtain the **UWorld** object that the player currently exists in and ensure that this world is valid before continuing.

7. Update the **SpawnProjectile()** function to the following:

```
void
ASuperSideScroller_Player::
SpawnProjectile()

{
    if(PlayerProjectile)

    {
        UWorld* World =
GetWorld();

        if (World)

        {
            }

    }
}
```

At this point, you have made safety checks to ensure that both **PlayerProjectile** and **UWorld** are

valid, so now it is safe to attempt to spawn the projectile. The first thing to do is declare a new variable of the **FActorSpawnParameters** type and assign the player as the owner.

8. Add the following code within the most recent **if** statement so that the **SpawnProjectile()** function looks like this:

```
void
ASuperSideScroller_Player::
SpawnProjectile()

{
    if(PlayerProjectile)

    {
        UWorld* World =
GetWorld();

        if (World)

        {
            FActorSpawnParame
ters SpawnParams;

            SpawnParams.Owner
= this;

        }
    }
}
```

```
 }  
 }
```

As you have previously learned, the **SpawnActor()** function call from the **UWorld** object will require the **FActorSpawnParameters** struct as part of the spawned objects' initialization. In the case of the player projectile, you can use the **this** keyword as a reference to the player character class for the owner of the projectile. This will come in handy later on in this function when you update the collision of the projectile after it spawns.

9. Next, you need to handle the **Location** and **Rotation** parameters of the **SpawnActor()** function. Add the following lines under the latest line, **SpawnParams.Owner = this**:

```
FVector SpawnLocation =  
this->GetMesh()->GetSocketLocation(FName("ProjectileSocket"));  
  
FRotator Rotation =  
GetActorForwardVector().Rot
```

```
ation();
```

In the first line, you declare a new **FVector** variable called **SpawnLocation**. This vector uses the **Socket** location of the **ProjectileSocket** socket that you created in the previous exercise. The **Skeletal Mesh** component returned from the **GetMesh()** function contains a function called **GetSocketLocation()** that will return the location of the socket with the **FName** that is passed in; in this case, the name **ProjectileSocket**.

In the second line, you are declaring a new **FRotator** variable called **Rotation**. This value is set to the player's forward vector, converted into a **Rotator** container. This will ensure that the rotation, or in other words, the direction in which the player projectile will spawn, will be in front of the player, and it will move away from the player.

Now, all of the parameters required to spawn the projectile are ready.

10. Add the following line underneath the code from the previous step:

```
APlayerProjectile*  
Projectile = World-  
>SpawnActor<APlayerProjecti  
le>(PlayerProjectile,  
SpawnLocation, Rotation,  
SpawnParams);
```

The **World->SpawnActor()** function will return an object of the class you are attempting to spawn in; in this case, **APlayerProjectile**. This is why you are adding **APlayerProjectile*** **Projectile** before the actual spawning occurs. Then, you are passing in the **SpawnLocation**, **Rotation**, and **SpawnParams** parameters to ensure that the projectile is spawning where and how you want.

11. Finally, you can add the player character to the array of actors to ignore on the player projectile by adding the following lines of code:

```
if (Projectile)
```

```
{  
    Projectile-  
    >CollisionComp->  
    MoveIgnoreActors.Add(SpawnP  
    arams.Owner);  
}
```

Now that you have a reference to the projectile, this line is updating the **CollisionComp** component so that the player, or **SpawnParams.Owner**, is added to the **MoveIgnoreActors** array. This array of actors will be ignored by the projectile's collision as it moves, which is perfect because this projectile should not collide with the player that has thrown it.

12. Return to the editor to recompile the newly added code. After the code compiles successfully, this exercise is complete.

With this exercise complete, you now have a function that will spawn the player projectile class that is assigned inside the player character. By adding safety checks for the validity of both the projectile and the world, you ensure that if an object is spawned, it is a valid object inside a valid world.

Next, you set up the appropriate **location**, **rotation**, and

FActorSpawnParameters parameters for the **UWorld** **SpawnActor()** function to ensure that the player projectile spawns at the right location, based on the socket location from the previous exercise, with the appropriate direction so that it moves away from the player, and with the player character as its **Owner**.

Now, it is time to update the **Anim_ProjectileNotify** source file so that it spawns the projectile.

Exercise 14.06: Updating the **Anim_ProjectileNotify** Class

You have the function ready to allow the player projectile to spawn, but you aren't calling this function anywhere yet. Back in *Exercise 14.01, Creating a UAnim Notify Class*, you created the **Anim_ProjectileNotify** class, while in *Exercise 14.02, Adding the Notify to the Throw Montage*, you added this notify to the **Throw** Animation Montage.

Now it is time to update the **Uanim Notify** class so that it calls the **SpawnProjectile()** function.

Do the following to achieve this:

1. In Visual Studio, open the **Anim_ProjectileNotify.cpp** source file.

In the source file, you have the following code:

```

#include
"Anim_ProjectileNotify.h"

void
UAnim_ProjectileNotify::Not
ify(USkeletalMeshComponent*
MeshComp,
UAnimSequenceBase*
Animation)

{
    UE_LOG(LogTemp, Warning,
TEXT("Throw Notify"));

}

```

2. Remove the **UE_LOG()** line from the **Notify()** function.
3. Next, add the following **include** lines underneath **Anim_ProjectileNotify.h**:

```

#include
"Components/SkeletalMeshCom
ponent.h"

#include
"SuperSideScroller/SuperSid
eScroller_Player.h"

```

You need to include the

SuperSideScroller_Player.h

header file because it is required in order to call the

SpawnProjectile() function you created in the previous exercise. We also included

SkeletalMeshComponent.h

because we will reference this component inside the **Notify()** function, so it's best to include it here too.

The **Notify()** function passes in a reference to the owning **Skeletal Mesh**, labeled **MeshComp**. You can use the skeletal mesh to get a reference to the player character by using the **GetOwner()** function and casting the returned actor to your **SuperSideScroller_Player** class. We'll do this next.

4. Inside the **Notify()** function, add the following line of code:

```
ASuperSideScroller_Player*
Player =
Cast<ASuperSideScroller_Player>(MeshComp->GetOwner());
```

5. Now that you have a reference to the player, you need to add a validity check of the **Player** variable before making a call to the **SpawnProjectile()** function. Add the following lines of code after the line from the previous step:

```
if (Player)
{
    Player-
    >SpawnProjectile();
}
```

6. Now that the **SpawnProjectile()** function is being called from the **Notify()** function, return to the editor to recompile and hot-reload the code changes you have made.

Before you are able to use **PIE** to run around and throw the player projectile, you need to assign the **Player Projectile** variable from the previous exercise.

7. Inside the **Content Browser** interface, navigate to the **/MainCharacter/Blueprints**

directory to find the **BP_SuperSideScroller_MainCharacter** Blueprint. *Double-click* to open the Blueprint.

8. In the **Details** panel, underneath the **Throw Montage** parameter, you will find the **Player Projectile** parameter. *Left-click* the drop-down option for this parameter and find **BP_PlayerProjectile**. *Left-click* on this option to assign it to the **Player Projectile** variable.
9. Recompile and save the **BP_SuperSideScroller_MainCharacter** Blueprint.
10. Now, use **PIE** and use the *left mouse button*. The player character will play the **Throw** animation and the player projectile will spawn.

Notice that the projectile is spawned from the **ProjectileSocket** function you created and that it moves away from the player. The following screenshot shows this in action:



Figure 14.17: The player can now throw the player projectile

With this exercise complete, the player can now throw the player projectile. The player projectile, in its current state, is ineffective against enemies and just flies through the air. It took a lot of moving parts between the **Throw** Animation Montage, the **Anim_ProjectileNotify** class, and the player character to get the player to throw the projectile.

In the upcoming exercises, you will update the player projectile so that it destroys enemies and play additional effects such as particles and sound.

Destroying Actors

So far in this chapter, we have put a lot of focus on spawning, or creating, actors inside the game world; the player character uses the **UWorld** class in order to spawn the projectile. Unreal Engine 4 and its base **Actor** class come with a default function that you can use to destroy, or remove, an actor from the game world:

```
bool AActor::Destroy( bool bNetForce, bool  
bShouldModifyLevel )
```

You can find the full implementation of this function in Visual Studio by finding the **Actor.cpp** source file in the **/Source/Runtime/Engine/Actor.cpp** directory. This function exists in all the classes that extend from the **Actor** class, and in the case of Unreal Engine 4, it exists in all classes that can be spawned, or placed, inside the game world. To be more explicit, both the **EnemyBase** and **PlayerProjectile** classes are *children* of the **Actor** class, and therefore, can be destroyed.

Looking further into the **AActor::Destroy()** function, you will find the following line:

```
World->DestroyActor( this, bNetForce,  
bShouldModifyLevel );
```

We will not be going into further detail about what exactly the **UWorld** class does in order to destroy an actor, but it is important to emphasize the fact that the **UWorld** class is responsible for both the creation and destruction of actors inside the world. Feel free to dig deeper into the source engine code to find more information about how the **UWorld** class handles the destruction and spawning of actors.

Now that you have more context regarding how Unreal Engine 4 handles the destruction and removal of actors from the game world, we'll implement this ourselves for the enemy character.

Exercise 14.07: Creating the DestroyEnemy() Function

The main part of the gameplay for the **Super SideScroller** game is for the player to move around the

level and use the projectile to destroy enemies. At this point in the project, you have handled the player movement and spawning the player projectile. However, the projectile does not destroy enemies yet.

In order to get this functionality in place, we'll start by adding some logic to the **EnemyBase** class so that it knows how to handle its destruction and remove it from the game once it collides with the player projectile.

Complete the following steps to achieve this:

1. First, navigate to Visual Studio and open the **EnemyBase.h** header file.
2. In the header file, create the declaration of a new function called **DestroyEnemy()** under the **Public** access modifier, as shown here:

```
public:  
    void DestroyEnemy();
```

Make sure this function definition is written underneath **GENERATED_BODY()**, within the class definition.

3. Save these changes to the header file and open the **EnemyBase.cpp** source file in order to add the implementation of this function.

4. Below the `#include` lines, add the following function definition:

```
void  
AEnemyBase::DestroyEnemy()  
{  
}
```

For now, this function will be very simple. All you need to do is call the inherited **Destroy()** function from the base **Actor** class.

5. Update the **DestroyEnemy()** function so that it looks like this:

```
void  
AEnemyBase::DestroyEnemy()  
{  
    Destroy();  
}
```

6. With this function complete, save the source file and return to the editor so that you can recompile and hot-reload the code.

With this exercise complete, the enemy character now has a function that can easily handle the destruction of the actor

whenever you choose. The **DestroyEnemy()** function is publicly accessible so that it can be called by other classes, which will come in handy later when you handle the destruction of the player projectile.

The reason you're creating your own unique function to destroy the enemy actor is because you will use this function later in this chapter to add VFX and SFX to the enemy when they are destroyed by the player projectile.

Before moving on to the polishing elements of the enemy's destruction, let's implement a similar function inside the player projectile class so that it can also be destroyed.

Exercise 14.08: Destroying Projectiles

Now that the enemy characters can handle being destroyed through the new **DestroyEnemy()** function you implemented in the previous exercise, it is time to do the same for the player projectile.

By the end of this exercise, the player projectile will have its own unique function to handle its own destruction and removal from the game world.

Let's get started:

1. In Visual Studio, open the header file for the player projectile; that is, **PlayerProjectile.h**.
2. Under the **Public** access modifier, add the following function declaration:

```
void ExplodeProjectile();
```

3. Next, open the source file for the player projectile; that is,

PlayerProjectile.cpp.

4. Underneath the void **APlayerProjectile::OnHit** function, add the definition of the **ExplodeProjectile()** function:

```
void  
APlayerProjectile::ExplodeP  
rojectile()  
{  
}
```

For now, this function will work identically to the **DestroyEnemy()** function from the previous exercise.

5. Add the inherited **Destroy()** function to the new **ExplodeProjectile()** function, like so:

```
void  
APlayerProjectile::ExplodeP  
rojectile()  
{
```

```
    Destroy();  
}
```

6. With this function complete, save the source file and return to the editor so that you can recompile and hot-reload the code.

With this exercise complete, the player projectile now has a function that can easily handle the destruction of the actor whenever you choose. The reason you need to create your own unique function to handle destroying the player projectile actor is the same reason you created the **DestroyEnemy()** function – you will use this function later in this chapter to add VFX and SFX to the player projectile when it collides with another actor.

Now that you have experience with implementing the **Destroy()** function inside both the player projectile and the enemy character, it is time to put these two elements together.

In the next activity, you will enable the player projectile in order to destroy the enemy character when they collide.

Activity 14.01: Projectile Destroying Enemies

Now that both the player projectile and the enemy character can handle being destroyed, it is time to go the extra step and allow the player projectile to destroy the enemy character when they collide.

Perform the following steps to achieve this:

1. Add the **#include** statement for the **EnemyBase.h** header file toward the top of the **PlayerProjectile.cpp** source file.
2. Within the void **APlayerProjectile::OnHit()** function, create a new variable of the **AEnemyBase*** type and call this variable **Enemy**.
3. Cast the **OtherActor** parameter of the **APlayerProjectile::OnHit()** function to the **AEnemyBase*** class and set the **Enemy** variable to the result of this cast.
4. Use an **if()** statement to check the validity of the **Enemy** variable.
5. If the **Enemy** is valid, call the **DestroyEnemy()** function from this **Enemy**.
6. After the **if()** block, make a call to the **ExplodeProjectile()** function.
7. Save the changes to the source file and return to the Unreal Engine 4 editor.

8. Use **PIE** and then use the player projectile against an enemy to observe the results.

The expected output is as follows:

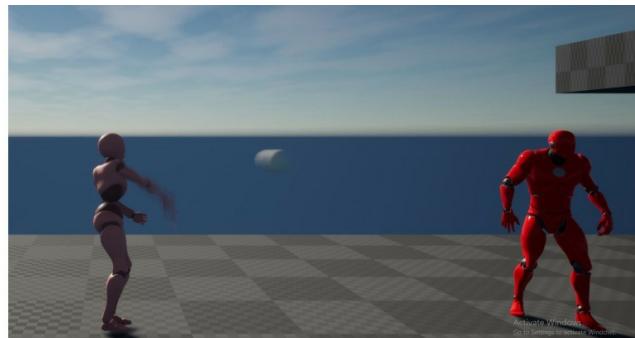


Figure 14.18: The player throwing the projectile

As the projectile hits the enemy, the enemy character is destroyed, as shown here:

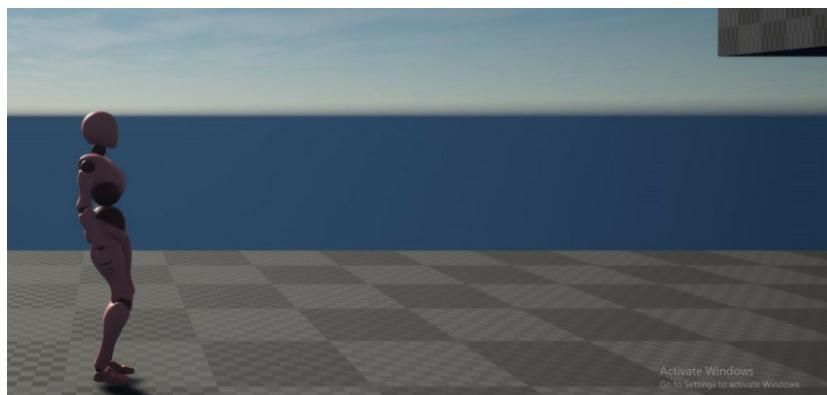


Figure 14.19: The projectile and enemy destroyed

With this activity complete, the player projectile and the enemy character can be destroyed when they collide with each other. Additionally, the player projectile will be destroyed whenever another actor triggers its **APlayerProjectile::OnHit()** function.

With that, a major element of the **Super SideScroller**

game has been completed: the player projectile spawning and the enemies being destroyed when they collide with the projectile. You can observe that destroying these actors is very simple and not very interesting to the player.

This is why, in the upcoming exercises in this chapter, you will learn more about Visual and Audio Effects, or VFX and SFX, respectively. You will also implement these elements with regard to the enemy character and player projectile.

Now that both the enemy character and the player projectile can be destroyed, let's briefly discuss what VFX and SFX are, and how they will impact the project.

Note

*The solution to this activity can be found at:
<https://packt.live/338jEBx>.*

Visual and Audio Effects

Visual Effects such as particle systems and sound effects such as Sound Cues play an important role in video games. They add a level of polish on top of systems, game mechanics, and even basic actions that make these elements more interesting or more pleasing to perform.

Let's start by understanding Visual Effects, followed by Audio Effects.

Visual Effects (VFX)

Visual Effects, in the context of Unreal Engine 4, are made up of what's called **Particle Systems**. Particle systems are made up of emitters, and emitters are comprised of modules. In these modules, you can control the appearance and behaviors

of the emitter using materials, meshes, and mathematical modules. The end result can be anything from a fire torch, or snow falling, to rain, dust, and so on.

Note

You can learn more here: <https://docs.unrealengine.com/en-US/Resources>Showcases/Effects/index.html>.

Audio Effects (SFX)

Audio Effects, in the context of Unreal Engine 4, are made up of a combination of Sound Waves and Sound Cues:

- Sound Waves are **.wav** audio format files that can be imported into Unreal Engine 4.
- Sound Cues combine Sound Wave audio files with other nodes such as Oscillator, Modulator, and Concatenator to create unique and complex sounds for your game.

Note

You can learn more here:
<https://docs.unrealengine.com/en-US/Engine/Audio/SoundCues/NodeReference/index.html>.

Let's use the game *Portal 2*, developed by Valve, as an

example.

In *Portal 2*, the player uses a portal gun to fire two portals: one *orange* and one *blue*. These portals allow the player to travel across gaps, move objects from one location to another, and utilize other simple mechanics that layer on top of each other to create complex puzzles. The use of these portals, the sound effects for firing the portals, and the visual VFX of these portals make the game more enjoyable to play. If you are unfamiliar with the game, please watch the full walkthrough here: <https://www.youtube.com/watch?v=ZFqk8aj4-PA>.

Note

For further reading regarding the importance of sound and sound design, please refer to the following Gamasutra article:

https://www.gamasutra.com/view/news/318157/7_games_worth_studying_for_their_excellent_sound_design.php.

In the context of Unreal Engine 4, VFX were originally created using a tool called **Cascade**, where artists could combine the use of **materials**, **static meshes**, and **math** to create interesting and convincing effects for the game world. This book will not dive into how this tool works, but you can find information about Cascade here:

https://www.ue4community.wiki/Legacy/Introduction_to_Plugins_in_UE4_-_2_-_Cascade_at_a_Glance.

In more recent versions of the engine, starting in the 4.20 update, there is a plugin called **Niagara** that can be enabled to create Visual Effects. **Niagara**, unlike Cascade, uses a system similar to Blueprints where you can visually script the behaviors of the effect rather than use a preset of modules with pre-defined behavior. You can find more information about Niagara here: <https://docs.unrealengine.com/en-US/Engine/Niagara/Overview/index.html>.

In *Chapter 9, Audio-Visual Elements*, you learned more about audio and how audio is handled inside Unreal Engine 4. All that needs to be known right now is that Unreal Engine 4 uses the **.wav** file format to import audio into the engine. From there, you can use the **.wav** file directly, referred to as Sound Waves in the editor, or you can convert these assets into Sound Cues, which allow you to add Audio Effects on top of the sound wave.

Lastly, there is one important class to know about that you will be referencing in the upcoming exercises, and this class is called **UGameplayStatics**. This is a static class featured in Unreal Engine that can be used from both C++ and Blueprints, and it offers a variety of useful gameplay-related functions. The two functions you will be working with in the upcoming exercise are as follows:

```
UGameplayStatics::SpawnEmitterAtLocation
```

```
UGameplayStatics::SpawnSoundAtLocation
```

These two functions work in very similar ways; they both require a **World** context object in which to spawn the effect, the particle system or audio to spawn, and the location in which to spawn the effect. You will be using these functions to spawn the destroy effects for the enemy in the next exercise.

Exercise 14.09: Adding Effects When the Enemy Is Destroyed

In this exercise, you will add new content to the project that comes included with this chapter and exercise. This includes the particle VFX and sound SFX, and all of their required assets. Then, you will update the **EnemyBase** class so that it can use audio and particle system parameters to add the layer

of polish needed when the enemy is destroyed by the player projectile.

By the end of this exercise, you will have an enemy that is visually and audibly destroyed when it collides with the player projectile.

Let's get started:

1. To begin, we need to migrate specific assets from the **Action RPG** project, which can be found in the **Learn** tab of **Unreal Engine Launcher**.
2. From **Epic Games Launcher**, navigate to the **Learn** tab and, under the **Games** category, you will find **Action RPG**:

Note

You will be taking additional assets from the Action RPG project in later exercises of this chapter, so you should keep this project open so as to avoid redundant opening of the project.

3. Left-click the **Action RPG** game project and then left-click the **Create Project** option.

4. From here, select engine version 4.24 and choose which directory to download the project to. Then, *left-click* the **Create** button to start installing the project.
5. Once the **Action RPG** project has finished downloading, navigate to the **Library** tab of **Epic Games Launcher** to find **ActionRPG** under the **My Projects** section.
6. *Double-click* the **ActionRPG** project to open it in the Unreal Engine editor.
7. In the editor, find the **A_Guardian_Death_Cue** audio asset in the **Content Browser** interface. *Right-click* this asset and select **Asset Actions** and then **Migrate**.
8. After selecting **Migrate**, you will be presented with all the assets that are referenced in **A_Guardian_Death_Cue**. This includes all audio classes and sound wave files. Choose **OK** from the **Asset Report** dialogue window.

9. Next, you will need to navigate to the **Content** folder for your **Super SideScroller** project and *left-click Select Folder*.
10. Once the migration process is complete, you will be given a notification in the editor saying that the migration was completed successfully.
11. Do the same migration steps for the **P_Goblin_Death** VFX asset. The two primary assets you are adding to the project are as follows:

A_Guardian_Death_Cue
P_Goblin_Death

The **P_Goblin_Death** particle system asset references additional assets such as materials and textures that are included in the **Effects** directory, while **A_Guardian_Death_Cue** references additional sound wave assets included in the **Assets** directory.
12. After migrating these folders into your **Content** directory, open the Unreal

Engine 4 editor of your **SuperSideScroller** project to find the new folders included in your project's **Content Browser**.

The particle you will be using for the enemy character's destruction is called **P_Goblin_Death** and can be found in the **/Effects/FX_Particle/** directory. The sound you will be using for the enemy character's destruction is called **A_Guardian_Death_Cue** and can be found in the **/Assets/Sounds/Creatures/Guardian/** directory. Now that the assets you need have been imported into the editor, let's move on to the code.

13. Open Visual Studio and navigate to the header file for the enemy base class; that is, **EnemyBase.h**.
14. Add the following **UPROPERTY()** variable. This will represent the particle system for when the enemy is destroyed. Make sure this is declared under the **Public** access modifier:

```
UPROPERTY(EditAnywhere,  
BlueprintReadOnly)
```

```
class UParticleSystem*
DeathEffect;
```

15. Add the following **UPROPERTY()** variable. This will represent the sound for when the enemy is destroyed. Make sure this is declared under the **Public** access modifier:

```
UPROPERTY(EditAnywhere,
BlueprintReadOnly)
```

```
class USoundBase*
DeathSound;
```

With these two properties defined, let's move on and add the logic required to spawn and use these effects for when the enemy is destroyed.

16. Inside the source file for the enemy base class, **EnemyBase.cpp**, add the following includes for the **UGameplayStatics** and **UWorld** classes:

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

You will be using the

UGameplayStatics and **UWorld** classes to spawn the sound and particle system into the world when the enemy is destroyed.

17. Within the **AEnemyBase::DestroyEnemy()** function, you have one line of code:

```
Destroy();
```

18. Add the following line of code above the **Destroy()** function call:

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

It is necessary to define the **UWorld** object before attempting to spawn a particle system or sound because a **World** context object is required.

19. Next, use an **if()** statement to check the validity of the **World** object you just defined:

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

20. Within the **if()** block, add the following code to check the validity of

the **DeathEffect** property, and then spawn this effect using the **SpawnEmitterAtLocation** function from **UGameplayStatics**:

```
if(DeathEffect)
{
    UGameplayStatics::Spawn
    EmitterAtLocation(World,
    DeathEffect,
    GetActorTransform());
}
```

It cannot be emphasized enough that you should ensure an object is valid before attempting to spawn or manipulate the object. By doing so, you can avoid engine crashes.

21. After the **if(DeathEffect)** block, perform the same validity check of the **DeathSound** property and then spawn the sound using the **UGameplayStatics::SpawnSoundAtLocation** function:

```
if(DeathSound)
{
```

```
    UGameplayStatics::Spawn  
    SoundAtLocation(World,  
    DeathSound,  
    GetActorLocation());  
}
```

Before calling the **Destroy()** function, you need to make checks regarding whether both the **DeathEffect** and **DeathSound** properties are valid, and if so, spawn those effects using the proper **UGameplayStatics** function. This ensures that regardless of whether either property is valid, the enemy character will still be destroyed.

22. Now that the **AEnemyBase::DestroyEnemy()** function has been updated to spawn these effects, return to the Unreal Engine 4 editor to compile and hot-reload these code changes.
23. Within the **Content Browser** interface, navigate to the **/Enemy/Blueprints/** directory. *Double-click* the **BP_Enemy** asset to open it.

24. In the **Details** panel of the enemy Blueprint, you will find the **Death Effect** and **Death Sound** properties. *Left-click* on the drop-down list for the **Death Effect** property and find the **P_Goblin_Death** particle system.
25. Next, underneath the **Death Effect** parameter, *left-click* on the drop-down list for the **Death Sound** property and find the **A_Guardian_Death_Cue** Sound Cue.
26. Now that these parameters have been updated and assigned the correct effect, compile and save the enemy Blueprint.
27. Using **PIE**, spawn the player character and throw a player projectile at an enemy. If an enemy is not present in your level, please add one. When the player projectile collides with the enemy, the VFX and SFX you added will play, as shown in the following screenshot:



Figure 14.20: Now, the enemy explodes and gets destroyed in a blaze of glory

With this exercise complete, the enemy character now plays a particle system and a Sound Cue when it is destroyed by the player projectile. This adds a nice layer of polish to the game, and it makes it more satisfying to destroy the enemies.

In the next exercise, you will add a new particle system and audio components to the player projectile so that it looks and sounds more interesting while it flies through the air.

Exercise 14.10: Adding Effects to the Player Projectile

In its current state, the player projectile functions the way it is intended to; it flies through the air, collides with objects in the game world, and is destroyed. However, visually, the player projectile is just a ball with a plain white texture.

In this exercise, you will add a layer of polish to the player projectile by adding both a particle system and an audio component so that the projectile is more enjoyable to use.

Complete the following steps to achieve this:

1. Much like the previous exercises, we will need to migrate assets from the **Action RPG** project to our **Super SideScroller** project. Please refer to *Exercise 14.09, Adding Effects When the Enemy Is Destroyed*, on how to install and migrate assets from the **Action RPG** project.

The two primary assets you are adding to the project are as follows:

`P_Env_Fire_Grate_01`

`A_Ambient_Fire01_Cue`

The `P_Env_Fire_Grate_01` particle system asset references additional assets, such as materials and textures, that are included in the **Effects** directory, while

`A_Ambient_Fire01_Cue` references additional sound wave and sound attenuation assets included in the **Assets** directory.

The particle you will be using for the player projectile is called `P_Env_Fire_Grate_01` and can be found in the `/Effects/FX_Particle/` directory.

This is the same directory that's used by the **P_Goblin_Death** VFX from the previous exercise. The sound you will be using for the player projectile is called **A_Ambient_Fire01_Cue** and can be found in the **/Assets/Sounds/Ambient/** directory.

2. *Right-click* on each of these assets in the **Content Browser** interface of the **Action RPG** project and select **Asset Actions** and then **Migrate**.
3. Make sure to choose the directory of the **Content** folder for your **Super SideScroller** project before confirming the migration.

Now that the required assets have been migrated to our project, let's continue creating the player projectile class.

4. Open Visual Studio and navigate to the header file for the player projectile class; that is, **PlayerProjectile.h**.
5. Under the **Private** access modifier, underneath the declaration of the **UStaticMeshComponent***

MeshComp class component, add the following code to declare a new audio component for the player projectile:

```
UPROPERTY(VisibleDefaultsOn  
ly, Category = Sound)  
class UAudioComponent*  
ProjectileMovementSound;
```

6. Next, add the following code underneath the declaration of the audio component in order to declare a new particle system component:

```
UPROPERTY(VisibleDefaultsOn  
ly, Category = Projectile)  
class  
UParticleSystemComponent*  
ProjectileEffect;
```

Instead of using properties that can be defined within the Blueprint, such as in the enemy character class, these effects will be components of the player projectile. This is because these effects should be attached to the collision component of the projectile so that they move with the projectile as it travels across the level when thrown.

- With these two components declared in the header file, open the source file for the player projectile and add the following includes to the list of **include** lines at the top of the file:

```
#include  
"Components/AudioComponent.  
h"  
  
#include  
"Engine/Classes/Particles/P  
articleSystemComponent.h"
```

You need a reference to both the audio component and the particle system classes in order to create these sub-objects using the **CreateDefaultSubobject** function, and to attach these components to **RootComponent**.

- Add the following lines in order to create the default sub-object of the **ProjectileMovementSound** component, and to attach this component to **RootComponent**:

```
ProjectileMovementSound =  
CreateDefaultSubobject<UAud  
ioComponent>
```

```
(TEXT("ProjectileMovementSo  
und"));  
  
    ProjectileMovementSound-  
    >AttachToComponent(RootComp  
onent,  
FAttachmentTransformRules::  
KeepWorldTransform);
```

9. Next, add the following lines in order to create the default sub-object for the **ProjectileEffect** component, and to attach this component to **RootComponent**:

```
ProjectileEffect =  
CreateDefaultSubobject<UPar  
ticleSystemComponent>  
(TEXT("Projectile  
Effect"));  
  
ProjectileEffect-  
>AttachToComponent(RootComp  
onent,  
FAttachmentTransformRules::  
KeepWorldTransform);
```

10. Now that you have created, initialized, and attached these two components to **RootComponent**, return to the Unreal Engine 4 editor to recompile and hot-

reload these code changes.

11. From the **Content Browser** interface, navigate to the **/MainCharacter/Projectile/** directory. Find the **BP_PlayerProjectile** asset and *double-click* it to open the Blueprint.

In the **Components** tab, you will find the two new components you added using the preceding code. Observe that these components are attached to the **CollisionComp** component, also known as **RootComponent**.

12. *Left-click* to select the **ProjectileEffect** component and, within the **Details** panel, assign the **P_Env_Fire_Grate_01** VFX asset to this parameter, as shown in the following screenshot:

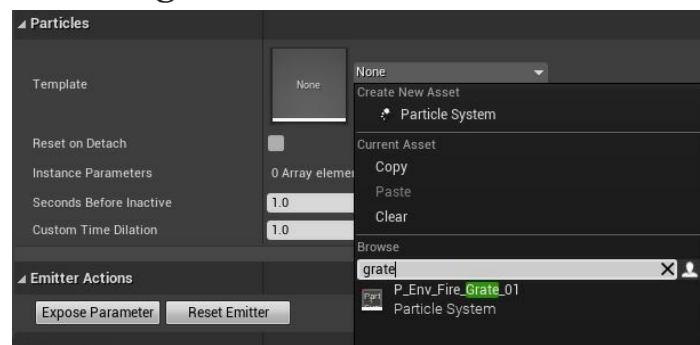


Figure 14.21: Now, you can apply

the P_Env_fire_Grate_o1 VFX asset to the Particle System component you added earlier

13. Before assigning the audio component, let's adjust the **Transform** of the **ProjectileEffect** VFX asset.

Update the **Rotation** and **Scale** parameters of the **Transform** for the VFX so that they match what is shown in the following screenshot:

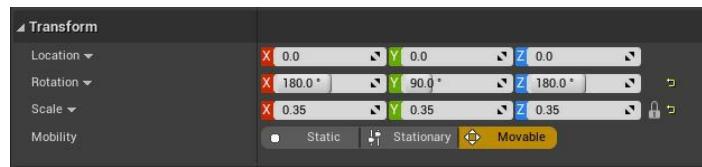


Figure 14.22: The updated Transform of the particle system component so that it fits better with the projectile

14. Navigate to the **Viewport** tab within the Blueprint to view these changes to the **Transform**. **ProjectileEffect** should look as follows:

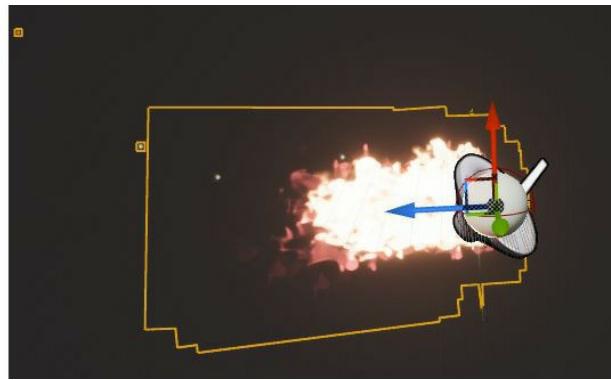


Figure 14.23: Now, the fire VFX has been scaled and rotated appropriately

15. Now that the VFX has been set up, *left-click* the **ProjectileMovementSound** component and assign **A_Ambient_Fire01_Cue** to this component.
16. Save and recompile the **BP_PlayerProjectile** Blueprint. Use **PIE** and observe that when you throw the projectile, it now shows the VFX asset and plays the assigned sound:



Figure 14.24: The player projectile now has a VFX and an SFX as it flies through the air

With this exercise complete, the player projectile now has a VFX and an SFX that play together while it flies through the air. These elements bring the projectile to life and make the projectile much more interesting to use.

Since the VFX and SFX are created as components of the projectile, they are also destroyed when the projectile is destroyed.

In the next exercise, you will add a particle notify and a sound notify to the **Throw** Animation Montage in order to provide more of an impact when the player throws the player projectile.

Exercise 14.11: Adding VFX and SFX Notifies

So far, you have been implementing polish elements to the game via C++, which is a valid means of implementation. In order to give variety, and to expand your knowledge of the Unreal Engine 4 toolset, this exercise will walk you through how to use notifies in Animation Montages to add particle systems and audio within the animation. Let's get started!

Much like the previous exercises, we will need to migrate assets from the **Action RPG** project to our **Super SideScroller** project. Please refer to *Exercise 14.09, Adding Effects When the Enemy Is Destroyed*, to learn how to install and migrate assets from the **Action RPG** project. Perform the following steps:

1. Open the **ActionRPG** project and

navigate to the **Content Browser** interface.

The two primary assets you are adding to the project are as follows:

P_Skill_001

A_Ability_FireballCast_Cue

The **P_Skill_001** particle system asset references additional assets such as *materials* and *textures* that are included in the **Effects** directory, while

A_Ability_FireballCast_Cue references additional *sound wave* assets included in the **Assets** directory.

The particle you will be using for the player when the projectile is thrown is called **P_Skill_001** and can be found in the **/Effects/FX_Particle/** directory. This is the same directory used by the **P_Goblin_Death** and **P_Env_Fire_Grate_01** VFX assets from the previous exercises. The sound you will be using for the enemy character destruction is called **A_Ambient_Fire01_Cue** and can be

found in the
/Assets/Sounds/Ambient/
directory.

2. *Right-click* on each of these assets in the **Content Browser** interface of the **Action RPG** project and select **Asset Actions** and then **Migrate**.
3. Make sure to choose the directory of the **Content** folder for your **Super SideScroller** project before confirming the migration.

Now that the assets you need have been migrated into your project, let's move on to adding the required notifies to the **AM_Throw** asset. Make sure to return to your **Super SideScroller** project before continuing with this exercise.

4. From the **Content Browser** interface, navigate to the **/MainCharacter/Animation/** directory. Find the **AM_Throw** asset and *double-click* it to open it.
5. Underneath the preview window in the center of the **Animation Montage**

editor, find the **Notifies** section. This is the same section where you added **Anim_ProjectileNotify** earlier in this chapter.

6. To the right of the **Notifies** track, you will find a + sign that allows you to use additional notify tracks. *Left-click* to add a new track, as shown in the following screenshot:

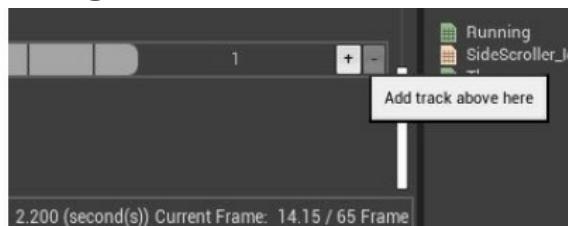


Figure 14.25: It is useful to add multiple tracks to the timeline in order to keep things organized when adding multiple notifies

7. In the same frame as **Anim_ProjectileNotify**, *right-click* within the new track you created in the previous step. From the **Add Notify** list, *left-click* to select **Play Particle Effect**.
8. Once created, *left-click* to select the new notify and access its **Details**

panel. In **Details**, add the **P_Skill_001** VFX asset to the **Particle System** parameter.

After you've added this new VFX, you will notice that the VFX is placed almost toward the bottom, where the player characters' feet are, but not exactly where you want it. This VFX should be placed directly on the floor, or at the base of the character. The following screenshot demonstrates this location:



Figure 14.26: The location of the particle notify is not on the ground

In order to fix this, you need to add a new **Socket** to the player character skeleton.

9. Navigate to the **/MainCharacter/Mesh/** directory.
Double-click the **MainCharacter_Skeleton** asset to open it.
10. From the **Skeleton** bone hierarchy on the left-hand side, *right-click* on the **Hips** bone and *left-click* to select the **Add Socket** option. Name this new socket **EffectSocket**.
11. *Left-click* this socket from the hierarchy of bones in order to view its current location. By default, its location is set to the same position as the **Hips** bone. The following screenshot shows this location:

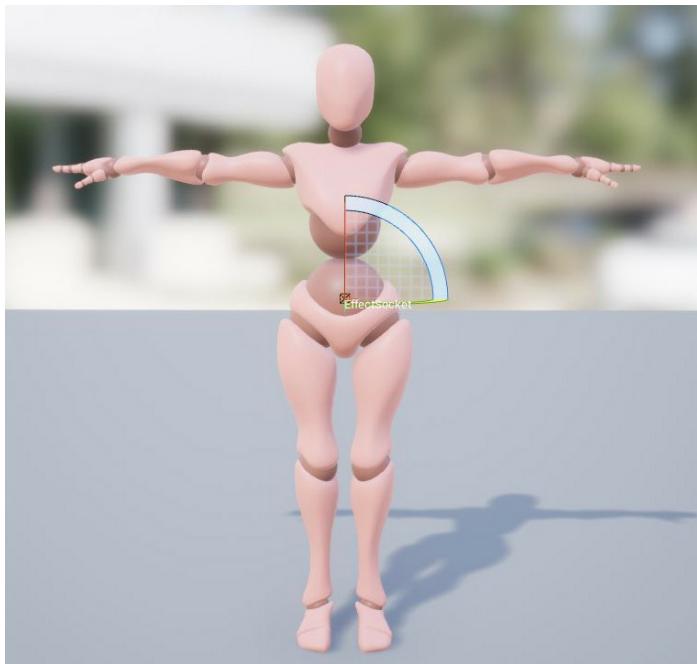


Figure 14.27: The default location of this socket is in the center of the player skeleton

Using the **Transform** gizmo widget, move the position of **EffectSocket** so that its position is set to the following:

(**X=0 . 000000 , Y=100 . 000000 , Z=0 . 000000**)

This position will be closer to the ground and the player characters' feet. The final location can be seen in the following screenshot:

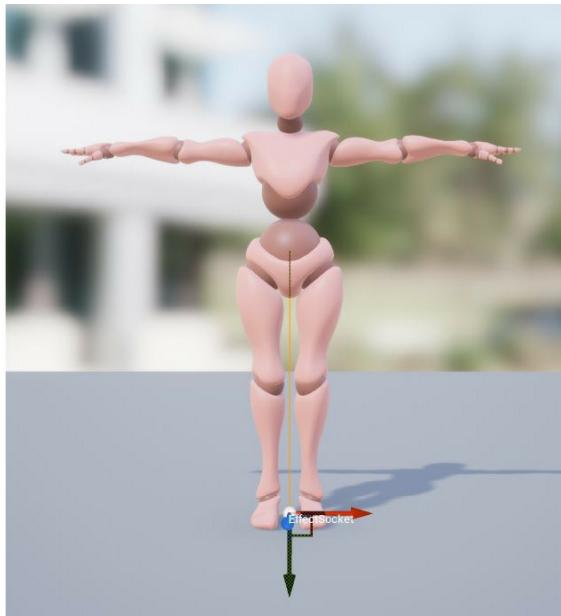


Figure 14.28: Moving the socket location to the base of the player skeleton

12. Now that you have a location for the particle notify, return to the **AM_Throw** Animation Montage.
13. Within the **Details** panel of the **Play Particle Effect** notify, there is the **Socket Name** parameter. Use **EffectSocket** as the name.

Note

*If **EffectSocket** does not appear via the autocomplete, close and reopen the Animation Montage. Once reopened, the **EffectSocket** option should appear for you.*

14. Lastly, the scale of the particle effect is a little too big, so adjust the scale of the projectile so that its value is as follows:

(X=0 . 500000 , Y=0 . 500000 , Z=0 . 500000)

Now, when the particle effect is played via this notify, its position and scale will be correct, as shown here:



Figure 14.29: The particle now plays at the base of the player character skeleton

15. To add the **Play Sound** notify, add a new track to the **Notifies** timeline section; you should now have three in total.
16. On this new track, and at the same frame position as both the **Play Particle Effect** and **Anim_ProjectileNotify** notifies, *right-click* and select the **Play Sound** notify from the **Add Notify** selection. The following screenshot shows where to find this notify:

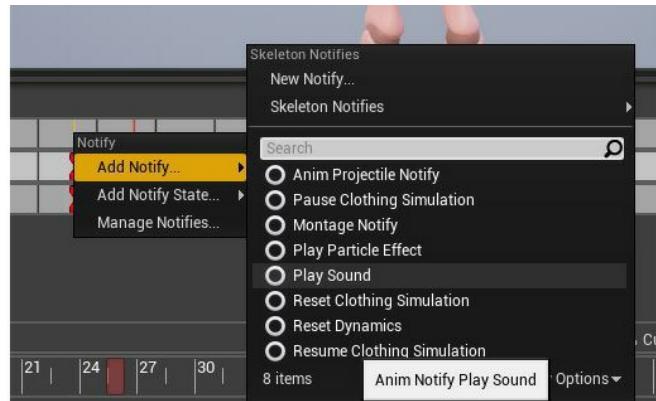


Figure 14.30: The Play Sound notify that you learned about earlier in this chapter

17. Next, *left-click* to select the **Play Sound** notify and access its **Details** panel.
18. From the **Details** panel, find the **Sound** parameter and assign **A_Ability_FireballCast_Cue**.

With the sound assigned, when the **Throw** animation is played back, you will see the VFX play and you will hear the sound. The **Notifies** tracks should look as follows:



Figure 14.31: The final notify set up on the Throw Animation Montage timeline

19. Save the **AM_Throw** asset and use **PIE** to throw the player projectile.
20. Now, when you throw the projectile, you will see the particle notify play the **P_Skill_001** VFX and you will hear the **AAbility_FireballCast_Cue** SFX. The result will look as follows:

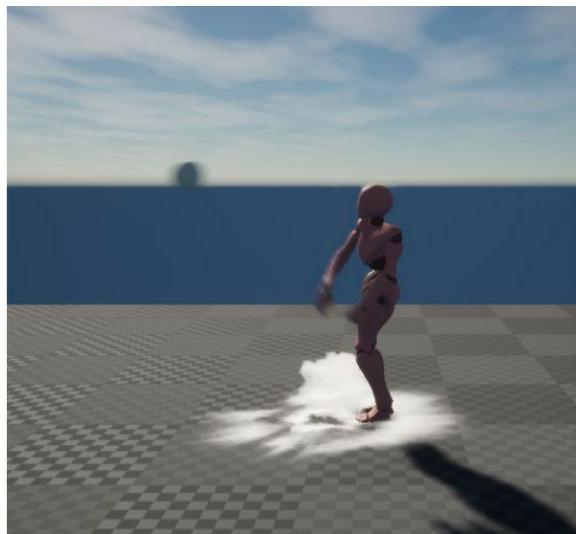


Figure 14.32: Now, when the player throws the projectile, powerful VFX and SFX are played

With this final exercise complete, the player now plays powerful VFX and SFX when the player projectile is thrown. This gives the throw animation more power and it feels like the player character is using a lot of energy to throw the projectile.

In the following final activity, you will use the knowledge you've gained from the last few exercises to add VFX and SFX to the player projectile when it is destroyed.

Activity 14.02: Adding Effects for When the Projectile Is Destroyed

In this final activity, you will use the knowledge that you've gained from adding VFX and SFX elements to the player projectile and the enemy character to create an explosion effect for when the projectile collides with an object instead. The reason we're adding this additional explosion effect is to add a level of polish on top of destroying the projectile when it collides with environment objects. It would look awkward and out of place if the player projectile were to hit an object and disappear without any audio or visual feedback from the player.

You will add both a particle system and Sound Cue parameters to the player projectile and spawn these elements when the projectile collides with an object.

Perform the following steps to achieve the expected output:

1. Inside the **PlayerProjectile.h** header file, add a new particle system variable and a new sound base variable.
2. Name the particle system variable **DestroyEffect** and name the sound base variable **DestroySound**.
3. In the **PlayerProjectile.cpp** source file, add the include for

UGameplayStatics to the list of includes.

4. Update the **APlayerProjectile::ExplodeProjectile()** function so that it now spawns both the **DestroyEffect** and **DestroySound** objects. Return to the Unreal Engine 4 editor and recompile the new C++ code. Inside the **BP_PlayerProjectile** Blueprint, assign the **P_Explosion** VFX, which is already included in your project by default, to the **Destroy Effect** parameter of the projectile.
5. Assign the **Explosion_Cue** SFX, which is already included in your project by default, to the **Destroy Sound** parameter of the projectile.
6. Save and compile the player projectile Blueprint.
7. Use **PIE** to observe the new player projectile's destruction VFX and SFX.

The expected output is as follows:



Figure 14.33: Projectile VFX and SFX

With this activity complete, you now have experience with adding polish elements to the game. Not only have you added these elements through C++ code, but you've added elements through other tools from Unreal Engine 4. At this point, you have enough experience to add particle systems and audio to your game without having to worry about how to implement these features.

Note

*The solution to this activity can be found at:
<https://packt.live/338jEBx>.*

Summary

In this chapter, you learned a lot about the importance of Visual and Audio Effects in the world of game development. Using a combination of C++ code and notifies, you were able to bring gameplay functionality to the player projectile and the enemy character colliding, as well as a layer of polish to

this functionality by adding VFX and SFX. On top of this, you learned about how objects are spawned and destroyed in Unreal Engine 4.

Moreover, you learned about how Animation Montages are played, both from Blueprints and through C++. By migrating the logic of playing the **Throw** Animation Montage from Blueprint to C++, you learned how both methods work and how to use both implementations for your game.

By adding a new Animation Notify using C++, you were able to add this notify to the **Throw** Animation Montage, which allows the player to spawn the player projectile you created in the previous chapter. Through the use of the **UWorld->SpawnActor()** function, and adding a new socket to the player skeleton, you were able to spawn the player projectile at the exact frame of the **Throw** animation, and at the exact position that you wanted to.

Lastly, you learned how to use the **Play Particle Effect** and **Play Sound** notifies within the Throw Animation Montage to add VFX and SFX to the throw of the player projectile. This chapter has given you the opportunity to learn about the different methods that exist inside Unreal Engine 4 when it comes to using VFX and SFX for your game.

Now that the player projectile can be thrown and destroy enemy characters, it is time to implement the final set of mechanics for the game. In the next chapter, you will create the collectibles that the player can collect, and you will also create a powerup for the player that will improve the players' movement mechanics for a short period of time.

15. Collectibles, Power-Ups, and Pickups

Overview

In this chapter, we will create collectible coins and potion power-ups for the player. Furthermore, we will design the UI for the collectible coins using the Unreal Motion Graphics UI Designer, or UMG as it is more commonly known. Lastly, we will create bricks that will have these collectibles hidden inside of them. By the end of this chapter, you will be able to implement collectibles and power-ups for a player character to find within a level environment.

Introduction

In the previous chapter, you created the player projectile and used **Anim Notifies** to spawn the player projectile during the **Throw** animation. The player projectile will serve as the player's main offensive gameplay mechanic to use against the enemies throughout the level. Due to the combination of default **Anim Notifies** provided by Unreal Engine 4 and your own custom **Anim_ProjectileNotify** class, the player projectile mechanic looks and feels great.

The last set of mechanics that we need to develop are the coin collectible and potion power-up. Let's briefly break down how collectibles and power-ups have influenced other games, and what they will accomplish for our **SuperSideScroller** game.

Coin Collectible

Collectibles give the player an incentive to explore the level thoroughly. In many games, such as *Hollow Knight*, collectibles also serve as a form of currency that can be used to purchase upgrades for your character and items. In other, more classic platformers, such as Super Mario or Sonic, collectibles serve to improve the player's score as they traverse the level.

In today's gaming landscape, it is expected that games include achievements. Collectibles are a great way to integrate achievements into your game; for example, an achievement for collecting all the coins in a level, or the entire game. For the **SuperSideScroller** game, the coin collectible will serve as a satisfactory means for the player to explore the game's levels to find as many coins as possible.

Potion Power-Up

Power-ups give the player either permanent or temporary advantages against enemies or the environments that the player must navigate through. There are many game examples that utilize power-ups, and one of the most famous is the Metroid series. Metroid uses power-ups to allow the player to explore new areas and battle against stronger enemies.

Power-ups are also another way to integrate achievements into your game. For example, you can have an achievement to destroy a certain number of enemies with a specific power-up. For the **SuperSideScroller** game, the potion power-up will serve as a means to improve the player's ability to navigate the level environment by increasing their movement speed and jump height.

In this chapter, you will learn how to create the coin collectible and potion power-up using C++ to add more layers of gameplay to the **SuperSideScroller** game. These gameplay elements will derive from the same base **actor** class that you will create. You will also be adding visual and

audio elements to both the collectible and the power-up so that they are more polished.

To make the coin collectible and potion power-up more visually interesting to the player, we will add a rotating component to these actors in order to draw the player's attention to them. This is where

URotatingMovementComponent can be very useful; it allows us to add rotation to actors in a very optimized and straightforward way, as opposed to coding our own logic to handle the constant rotation of the actor. Let's begin by learning more about this component.

URotatingMovementComponent

URotatingMovementComponent is one of a few movement components that exist within Unreal Engine 4. You are already familiar with **CharacterMovementComponent** and **ProjectileMovementComponent** from the **SuperSideScroller** game project alone, and **RotatingMovementComponent** is just that – another movement component. As a refresher, movement components allow different types of movements to occur on actors, or characters, that they belong to.

Note

CharacterMovementComponent, which allows you to control the movement parameters of your character such as their movement speed and jump height, was covered in Chapter 10, Creating a SuperSideScroller Game, when you created the **SuperSideScroller** player character.

ProjectileMovementComponent, which allows you to

add projectile-based movement functionality to actors such as speed and gravity, was covered in Chapter 14, Spawning the Player Projectile, when you developed the player projectile.

RotatingMovementComponent is a very simple movement component compared to **CharacterMovementComponent** and that's because it only involves rotating the actor that **RotatingMovementComponent** is a part of; nothing more. **RotatingMovementComponent** performs the continuous rotation of a component based on the defined **Rotation Rate**, pivot translation, and the option to use rotation in local space or world space.

Additionally, **RotatingMovementComponent** is much more efficient compared to other methods of rotating an actor, such as through the **Event Tick** or **Timelines** within Blueprints.

Note

More information about movement components can be found here: <https://docs.unrealengine.com/en-US/Engine/Components/Movement/index.html#rotatingmovementcomponent>.

We will be using **RotatingMovementComponent** to allow the coin collectible and potion power-up to rotate in-place along the Yaw axis. This rotation will draw the player's attention to the collectible and give them a visual cue that the collectible is important.

Now that you have a better understanding of **RotatingMovementComponent**, let's move on and create the **PickableActor_Base** class, which is what the coin collectible and the potion power-up will derive from.

Exercise 15.01: Creating the PickableActor_Base Class and Adding URotatingMovementComponent

In this exercise, you will be creating the **PickableActor_Base** actor class, which will be used as the base class that both the collectible coin and potion power-up will derive from. You will also create a Blueprint class from this C++ base class to preview how **URotatingMovementComponent** works. Follow these steps to complete this exercise:

Note

*You have performed many of the following steps numerous times throughout the **SuperSideScroller** game project, so there will be limited images to help guide you. Only when introducing a new concept will there be an accompanying image.*

1. Inside the Unreal Engine 4 editor, *left-click* the **File** option at the top-left of the editor and *left-click* the option for **New C++ Class**.
2. From the **Choose Parent Class** window, select the **Actor** option, and then *left-click* on the **Next** button at

the bottom of this window.

3. Name this class

PickableActor_Base and leave the default **Path** directory as it is. Then, select the **Create Class** button at the bottom of this window.

4. After selecting the **Create Class** button, Unreal Engine 4 will recompile the project code and automatically open Visual Studio with both the header and source files for the **PickableActor_Base** class.

5. By default, **Actor** classes provide you with the **virtual void Tick(float DeltaTime)** **override;** function declaration inside the header file. For the purposes of the **PickableActor_Base** class, we will not require the **Tick** function, so remove this function declaration from the **PickableActor_Base.h** header file.

6. Next, you will also need to remove the function from the **PickableActor_Base.cpp** file;

otherwise, you will receive a compile error. In this source file, find and remove the following code:

```
void  
PickableActor_Base::Tick(f1  
oat DeltaTime)  
{  
    Super::Tick(DeltaTime);  
}
```

Note

*In many cases, the use of the **Tick()** function for movement updates can lead to performance issues as the **Tick()** function is called every single frame. Instead, try using **Gameplay Timer** functions to perform certain updates at specified intervals, rather than on each frame. You can learn more about **Gameplay Timers** here: <https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Timers/index.html>.*

7. Now, it is time to add the components that the **PickableActor_Base** class

requires. Let's start with **USphereComponent**, which you will use to detect overlap collision with the player. Add the following code underneath the **Protected** access modifier inside the **PickableActor_Base.h** header file:

```
UPROPERTY(VisibleDefaultsOn  
ly, Category =  
PickableItem)  
  
class USphereComponent*  
CollisionComp;
```

The declaration of **USphereComponent** should be very familiar to you by now; we've done this in previous chapters, such as *Chapter 16, Multiplayer Basics*, when we created the **PlayerProjectile** class.

8. Next, add the following code underneath the declaration of **USphereComponent** to create a new **UStaticMeshComponent**. This will be used to visually represent either the coin collectible or the potion power-up:

```
UPROPERTY(VisibleDefaultsOn  
ly, Category =
```

```
PickableItem)

class UStaticMeshComponent*
MeshComp;
```

9. Finally, add the following code underneath the declaration of **UStaticMeshComponent** to create a new **URotatingMovementComponent**.

This will be used to give the collectible coin and potion power-up simple rotational movement:

```
UPROPERTY(VisibleDefaultsOn
ly, Category =
PickableItem)

class
URotatingMovementComponent*
RotationComp;
```

10. Now that you have the components declared inside the **PickableActor_Base.h** header file, navigate to the **PickableActor_Base.cpp** source file so that you can add the required **#includes** for these added components. Add the following lines after the first **#include**

"PickableActor_Base.h", at the top of the source file:

```
#include "Components/SphereComponent.h"  
  
#include "Components/StaticMeshComponent.h"  
  
#include "GameFramework/RotatingMovementComponent.h"
```

11. Now that you have the necessary **include** files for the components, you can add the necessary code to initialize these components within the **APickableActor_Base::APickableActor_Base()** constructor function:

```
APickableActor_Base::APickableActor_Base()  
{  
}
```

12. First, initialize the **USphereComponent** component variable, **CollisionComp**, by adding

the following code to
APickableActor_Base::APickableActor_Base():

```
CollisionComp =  
CreateDefaultSubobject  
<USphereComponent>  
(TEXT("SphereComp"));
```

13. Next, initialize **USphereComponent** with a default sphere radius of **30.0f** by adding the following code underneath the code provided in the previous step:

```
CollisionComp-  
>InitSphereRadius(30.0f);
```

14. Since the player character needs to overlap with this component, you will need to add the following code so that, by default, **USphereComponent** has the collision settings for **Overlap All Dynamic**:

```
CollisionComp-  
>BodyInstance.SetCollisionProfileName("OverlapAllDynamic");
```

15. Lastly, **CollisionComp**

USphereComponent should be the root component of this actor. Add the following code to assign this:

```
RootComponent =  
CollisionComp;
```

16. Now that **CollisionComp** **USphereComponent** has been initialized, let's do the same for **MeshComp** **UStaticMeshComponent**. Add the following code. After, we'll discuss what the code is doing for us:

```
MeshComp =  
CreateDefaultSubobject<USta  
ticMeshComponent>  
(TEXT("MeshComp"));  
  
MeshComp-  
>AttachToComponent(RootComp  
onent,  
FAttachmentTransformRules::  
KeepWorldTransform);  
  
MeshComp-  
>SetCollisionEnabled(ECollisi  
onEnabled::NoCollision);
```

The first line initializes **MeshComp**

UStaticMeshComponent using the **CreateDefaultSubobject()** template function. Next, you are attaching **MeshComp** to the root component, which you made for **CollisionComp**, using the **AttachTo()** function. Lastly, **MeshComp**

UStaticMeshComponent should not have any collision by default, so you are using the **SetCollisionEnabled()** function and passing in the **ECollisionEnable::NoCollision** enumerator value.

17. Lastly, we can initialize **URotatingMovementComponent** **RotationComp** by adding the following code:

```
RotationComp =  
CreateDefaultSubobject<URot  
atingMovementComponent>  
(TEXT("RotationComp"));
```

18. With all the components initialized, compile the C++ code and return to the Unreal Engine 4 editor. After

compilation succeeds, you will move on to creating a Blueprint class for **PickableActor_Base**.

19. In the **Content Browser** window, create a new folder called **PickableItems** by *right-clicking* on the **Content** folder and selecting the **New Folder** option.
20. In the **PickableItems** folder, *right-click* and select **Blueprint Class**. From the **Pick Parent Class** window, search for the **PickableActor_Base** class and *left-click Select* to create a new Blueprint.
21. Name this Blueprint **BP_PickableActor_Base** and *double-left-click* the Blueprint to open it.
22. In the **Components** tab, select **MeshComp Static Mesh Component** and assign the **Shape_Cone** static mesh to the **Static Mesh** parameter in the **Details** panel. Please refer to the

following screenshot:

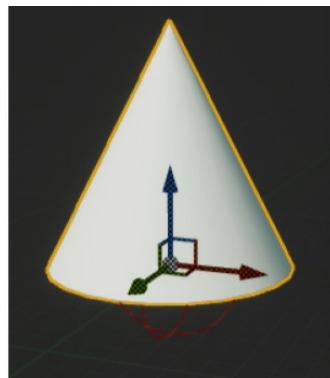


Figure 15.1: The Shape_Cone mesh assigned to MeshComp UStaticMeshComponent of the BP_Pickable_Base actor class

23. Next, select **RotationComp** **URotatingMovementComponent** and find the **Rotation Rate** parameter under the **Rotating Component** category of the **Details** panel.
24. Set **Rotation Rate** to the following values:

$$(X=100.000000, Y=100.000000, Z=100.000000)$$

These values determine how fast the actor will rotate along each axis per second. This means that the cone-shaped actor will rotate along each axis

at 100 degrees per second on each axis.

25. Compile the **PickableActor_Base**

Blueprint and add this actor to your level.

26. Now, if you use PIE and look at the

PickableActor_Base actor in the level, you will see that it is now rotating. Please refer to the following screenshot:

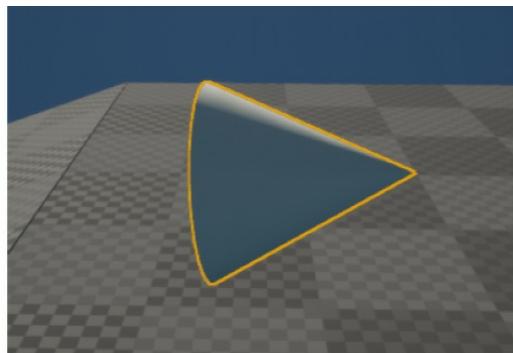


Figure 15.2: Now, the Cone mesh rotates along all the axes, as per the values we added to the Rotation Rate window of URotatingMovementComponent

Note

You can find the assets and code for this exercise here:

<https://packt.live/3njhwyt>.

With this exercise complete, you've created the base components required for the **PickableActor_Base** class and learned how to implement and use **URotatingMovementComponent**. With the **PickableActor_Base** class ready, and with

URotatingMovementComponent implemented on the Blueprint actor, we can complete the class by adding overlap detection functionality, destroying the collectible actor, and spawning audio effects when the actor is picked up by the player. In the following activity, you will add the remaining functionality required for the **PickableActor_Base** class.

Activity 15.01: Player Overlap Detection and Spawning Effects in PickableActor_Base

Now that the **PickableActor_Base** class has all the required components and has its constructor initializing the components, it is time to add the remaining aspects of its functionality. These will be inherited by the coin collectible and potion power-up later in this chapter. This additional functionality includes player overlap detection, destroying the collectible actor, and spawning an audio effect to give feedback to the player that it has been successfully picked up by. Perform the following steps to add functionality that allows a **USoundBase** class object to be played when the collectible overlaps with the player:

1. Create a new function in the **PickableActor_Base** class that takes in a reference to the player as an input parameter. Call this function **PlayerPickedUp**.
2. Create a new **UFUNCTION** called

BeginOverlap(). Make sure to include all the required input parameters for this function before moving on. Refer to *Chapter 6, Collision Objects*, where you used this function inside the **VictoryBox** class.

3. Add a new **UPROPERTY()** for the **USoundBase** class and name it **PickupSound**.
4. In the **PickableActor_Base.cpp** source file, create the definitions for both the **BeginOverlap()** and **PlayerPickedUp()** functions.
5. Now, add the required **#include** files for the **SuperSideScroller_Player** class and the **GameplayStatics** class at the top of the source file.
6. In the **BeginOverlap()** function, create a reference to the player using the **OtherActor** input parameter of the function.
7. In the **PlayerPickedUp()** function, create a variable for the **UWorld***

object that's returned by the **GetWorld()** function.

8. Use the **UGameplayStatics** library to spawn **PickUpSound** at the location of the **PickableActor_Base** actor.
9. Then, call the **Destroy()** function so that the actor gets destroyed and removed from the world.
10. Finally, in the **APickableActor_Base::APickableActor_Base()** constructor, bind the **OnComponentBeginOverlap** event of **CollisionComp** to the **BeginOverlap()** function.
11. Download and install the **Unreal Match 3** project from the **Learn** tab of **Epic Games Launcher**. Migrate the **Match_Combos** soundwave asset from this project into your **SuperSideScroller** project using the knowledge you gained in *Chapter 14, Spawning the Player Projectile*.
12. Apply this sound to the **PickupSound** parameter of the

BP_PickableActor_Base

Blueprint.

13. Compile the Blueprint, and if one does not exist in your level, add the **BP_PickableActor_Base** actor to your level now.
14. In **PIE**, have your character overlap with the **BP_PickableActor_Base** actor.

Expected output:



Figure 15.3: The BP_PickableActor_Base object can be overlapped and picked up by the player

Note

The solution to this activity can be found at:

<https://packt.live/338jEBx>.

With this activity complete, you have proven your knowledge regarding how to add the **OnBeginOverlap()** functionality to your actor classes and how to use this function to perform

logic for your actor. In the case of **PickableActor_Base**, we added logic that will spawn a custom sound and destroy the actor.

Now that the **PickableActor_Base** class is set and ready, it is time to develop the collectible coin and power-up potion classes that will derive from it. The coin collectible class will inherit from the **PickableActor_Base** class you have just created. It will override key functionality, such as the **PlayerPickedUp()** function, so that we can implement unique logic for the collectible when it's picked up by the player. In addition to overriding functionality from the inherited parent **PickableActor_Base** class, the coin collectible class will have its own unique set of properties, such as its current coin value and unique pickup sound. We'll create the coin collectible class together in the next exercise.

Exercise 15.02: Creating the **PickableActor_Collectable** Class

In this exercise, you will be creating the **PickableActor_Collectable** class, which will be derived from the **PickableActor_Base** class you created in *Exercise 15.01, Creating the PickableActor_Base Class and Adding URotatingMovement Component* and finished in *Activity 15.01, Player Overlap Detection and Spawning Effects in PickableActor_Base*. This class will be used as the main collectible coin that the player can collect within the level. Follow these steps to complete this exercise:

1. Inside the Unreal Engine 4 editor, left-

*click the **File** option at the top-left of the editor and *left-click* the option for **New C++ Class**.*

2. From the **Choose Parent Class** window, select the **PickableActor_Base** option, and then *left-click* on the **Next** button at the bottom of this window.
3. Name this class **PickableActor_Collectable** and leave the default **Path** directory as it is. Then, select the **Create Class** button at the bottom of this window.
4. After selecting the **Create Class** button, Unreal Engine 4 will recompile the project code and will automatically open Visual Studio with both the header and source files for the **PickableActor_Collectable** class.
5. By default, the **PickableActor_Collectable.h** header file has no declared functions or variables within its class declaration. You will need to add the override for

the **BeginPlay()** function underneath a new **Protected Access Modifier**. Add the following code:

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

The reason we are overriding the **BeginPlay()** function is that **URotatingMovementComponent** requires the actor to initialize and use **BeginPlay()** to correctly rotate the actor. Therefore, we need to create the override declaration of this function and create a basic definition inside the source file. First, however, we need to override another important function from the **PickableActor_Base** parent class.

6. Override the **PlayerPickedUp()** function from the **PickableActor_Base** parent class by adding the following code under **Protected Access Modifier**:

```
virtual void  
PlayerPickedUp(class
```

```
ASuperSideScroller_Player*  
Player) override;
```

With this, we are saying that we are going to use, and override, the functionality of the **PlayerPickedUp()** function.

7. Lastly, create a new integer called **UPROPERTY()** that will hold the value that the coin collectible will have; in this case, it will have a value of **1**. Add the following code to do this:

```
public:  
  
    UPROPERTY(EditAnywhere,  
Category = Collectable)  
  
        int32 CollectableValue =  
    1;
```

Here, we are creating the integer variable that will be accessible in Blueprints and has a default value of **1**. If you so choose, with the **EditAnywhere UPROPERTY()** keyword, you can change how much a coin collectible is worth.

8. Now, we can move on to the **PickableActor_Collectable.cp**

p source file to create the definition of the overridden **PlayerPickedUp()** function. Add the following code to the source file:

```
void  
APickableActor_Collectable::  
PlayerPickedUp(class  
ASuperSideScroller_Player*  
Player)  
{  
}
```

9. For now, we need to make a call to the **PlayerPickedUp()** parent function by using the **Super** keyword. Add the following code to the **PlayerPicked()** function:

```
Super::PlayerPickedUp(Playe  
r);
```

The call to the parent function using **Super::PlayerPickedUp(Player)** will ensure that the functionality you created in the **PickableActor_Base** class is called. As you may recall, the **PlayerPickedUp()** function in the parent class makes a call to spawn the

PickupSound sound object and destroys the actor.

10. Next, create the definition of the **BeginPlay()** function inside the source file by adding the following code:

```
void  
APickableActor_Collectable::  
:BeginPlay()  
{  
}
```

11. The last thing to do here in C++ is to once again make the call to the **BeginPlay()** parent function using the **Super** keyword. Add the following code to the **BeginPlay()** function inside the **PickableActor_Collectable** class:

```
Super::BeginPlay();
```

12. Compile the C++ code and return to the editor.

Note

*You can find the assets and code for this exercise at the following link:
<https://packt.live/35fRN3E>.*

Now that you've successfully compiled the **PickableActor_Collectable** class, you have created the framework needed for the coin collectible. In the following activity, you will create a Blueprint from this class and finalize the coin collectible actor.

Activity 15.02: Finalizing the PickableActor_Collectable Actor

Now that the **PickableActor_Collectable** class has all of the necessary inherited functionality and unique properties it needs, it is time to create the Blueprint from this class and add a **Static Mesh**, update its **URotatingMovementComponent**, and apply a sound to the **PickUpSound** property. Perform the following steps to finalize the **PickableActor_Collectable** actor:

1. From **Epic Games Launcher**, find the **Content Examples** project from the **Learn** tab, underneath the **Engine Feature Samples** category.

2. Create and install a new project from the **Content Examples** project.
3. Migrate the **SM_Pickup_Coin** asset and all its referenced assets from the **Content Examples** project to your **SuperSideScroller** project.
4. Create a new folder within the **Content/PickableItems** directory in the **Content Browser** window and name it **Collectable**.
5. In this new **Collectable** folder, create a new Blueprint from the **PickableActor_Collectable** class that you created in *Exercise 15.02, Creating the PickableActor_Collectable Class*. Name this new Blueprint **BP_Collectable**.
6. In this Blueprint, set the **Static Mesh** parameter of the **MeshComp** component to the **SM_Pickup_Coin** mesh you imported earlier in this activity.
7. Next, add the **Match_Combo** sound

asset to the **PickupSound** parameter of the collectible.

8. Lastly, update the **RotationComp** component so that the actor rotates along the Z-axis at 90 degrees per second.
9. Compile the Blueprint, place **BP_Collectable** in your level, and use PIE.
10. Overlap the player character with the **BP_Collectable** actor and observe the results.

Expected output:



Figure 15.4: The coin collectible rotates and can be overlapped by the player

Note

The solution to this activity can be found at:

<https://packt.live/338jEBx>.

With this activity complete, you have proven that you know how to migrate assets into your Unreal project and how to use and update **URotatingMovementComponent** to fit the needs of the coin collectible. Now that the coin collectible actor is complete, it is time to add functionality to the player so that the player can keep track of how many coins they have collected.

First, we will create the logic that will count the coins using **UE_LOG** and later implement the coin counter using UMG on the game's UI.

Logging Variables Using **UE_LOG**

In *Chapter 11, Blend Spaces 1D, Key Bindings, and State Machines*, we used and learned about the **UE_LOG** function in order to log when the player should throw the projectile. We then used the **UE_LOG** function in *Chapter 13, Enemy Artificial Intelligence*, to log when the player projectile hit an object. **UE_LOG** is a robust logging tool we can use to output important information from our C++ functions into the **Output Log** window inside the editor when playing our game. Up until this point, we have only logged **FStrings** to display general text in the **Output Log** window to know that our functions were being called. Now, it is time to learn how to log variables in order to debug how many coins the player has collected.

Note

*There is another useful debug function available in C++ with Unreal Engine 4, known as **AddOnScreenDebugMessage**.*

You can learn more about this function here:

<https://docs.unrealengine.com/en-US/API/Runtime/Engine/Engine/UEngine/AddOnScreenDebugMessage/1/index.html>.

When creating the **FString** syntax used by the **TEXT()** macro, we can add format specifiers to log different types of variables. We will only be discussing how to add the format specifier for integer variables.

Note

You can find more information on how to specify other variable types by reading the following documentation:

https://www.ue4community.wiki/Logging#Logging_an_FString.

This is what **UE_LOG()** looks like when passing in **FString "Example Text"**:

```
UE_LOG(LogTemp, Warning, TEXT("Example  
Text"));
```

Here, you have **Log Category**, **Log Verbose Level**, and the actual **FString**, **"Example Text"**, to display in the log.

To log an integer variable, you need to add **%d** to your **FString** within the **TEXT()** macro, followed by the integer variable name outside the **TEXT()** macro, separated by a comma. Here is an example:

```
UE_LOG(LogTemp, Warning, TEXT("My integer  
variable %d), MyInteger);
```

The format specifier is identified by the **%** symbol, and each

variable type has a designated letter that corresponds with it. In the case of integers, the letter **d** is used. You will be using this method of logging integer variables to log the number of coin collectibles the player has in the next exercise.

Exercise 15.03: Tracking the Number of Coins for the Player

In this exercise, you will be creating the necessary properties and functions that will allow you to track how many coins the player collects throughout the level. You will use this tracking to show the player using UMG later in this chapter. Follow these steps to complete this exercise:

1. In Visual Studio, find and open the **SuperSideScroller_Player.h** header file.
2. Under **Private Access Modifier**, create a new **int** variable called **NumberofCollectables**, as shown here:

```
int32 NumberofCollectables;
```

This will be a private property that will keep track of the current number of coins the player has collected. You will be creating a public function that will

return this integer value. We do this for safety reasons to ensure that no other classes can modify this value.

3. Next, under the existing **public** access modifier, create a new **UFUNCTION()** using the **BlueprintPure** keyword called **GetCurrentNumberofCollectables()**. This function will return an **int**. The following code adds this as an inline function:

```
UFUNCTION(BlueprintPure)
int32
GetCurrentNumberofCollectab
les() { return
NumberofCollectables; };
```

We are using **UFUNCTION()** and the **BlueprintPure** keyword to expose this function to Blueprints so that we can use it later in UMG.

4. Declare a new **void** function, under the **public** access modifier, called **IncrementNumberofCollectable
s()** that takes in a single integer parameter called **Value**:

```
void  
IncrementNumberofCollectab  
les(int32 Value);
```

This is the main function you will use to keep track of how many coins the player has collected. We will also add some safety measures to ensure this value is never negative.

5. With the

**IncrementNumberofCollectable
s()** function declared, let's create the definition of this function inside the **SuperSideScroller_Player.cpp** source file.

6. Write the following code to create the definition of the **IncrementNumberofCollectable
s** function:

```
void  
ASuperSideScroller_Player::  
IncrementNumberofCollectabl  
es(int32 Value)  
{  
}
```

7. The main case to handle here is if the

integer value that's passed into this function is less than or equal to **0**. In this case, we do not want to bother incrementing the **NumberofCollectables** variable.

Add the following code to the **IncrementNumberofCollectable s()** function:

```
if(Value== 0)  
{  
    return;  
}
```

This **if()** statement says that if the **value** input parameter is less than or equal to **0**, the function will end. With the **IncrementNumberofCollectable s()** function returning **void**, it is perfectly okay to use the **return** keyword in this way.

We're adding this check of ensuring the **value** parameter that's passed into the **IncrementNumberofCollectable s()** function is neither 0 nor negative

because it is important to establish good coding practices; this guarantees that all possible outcomes are handled. In an actual development environment, there could be designers or other programmers who attempt to use the **IncrementNumberofCollectable** **s()** function and try to pass in a negative value, or a value that equals 0. If the function does not take these possibilities into account, there is potential for bugs later on in development.

8. Now that we've handled the edge case where **value** is less than or equal to **0**, let's continue with the function using an **else()** statement to increase **NumberofCollectables**. Add the following code under the **if()** statement from the previous step:

```
else
{
    NumberofCollectables +=
    Value;
}
```

9. Next, let's log **NumberofCollectables** using **UE_LOG** and the knowledge we learned about logging variables. Add the following code after the **else()** statement to properly log **NumberofCollectables**:

```
UE_LOG(LogTemp, Warning,
TEXT("Number of Coins:
%d"),
NumberofCollectables);
```

With this **UE_LOG()**, we are making a more robust log to track the number of coins. This lays out the groundwork of how the UI will work. This is because we are essentially logging the same information to the player through the use of UMG later in this chapter.

With **UE_LOG()** added, all we need to do is call the **IncrementNumberofCollectable** **s()** function inside the **PickableActor_Collectable** class.

10. In the **PickableActor_Collectable.cp**

p source file, add the following header:

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

11. Next, inside the **PlayerPickedUp()** function, add the following function call before the **Super::PlayerPickedUp(Player)** line:

```
Player->IncrementNumberofCollectables(CollectableValue);
```

12. Now that our **PickableActor_Collectable** class is calling our player's **IncrementNumberofCollectable**s function, recompile the C++ code and return to the Unreal Engine 4 editor.
13. Within the UE4 editor, open the **Output Log** window by *left-clicking Window*, and then hovering over the **Developer Tools** option. From this additional dropdown, select **Output Log**.

14. Now, add multiple **BP_Collectable** actors to your level and then use PIE.
15. When you overlap over each coin collectible, observe the **Output Log** window to find that each time you collect a coin, the **Output Log** window will show you how many coins you've collected.

Note

You can find the assets and code for this exercise here:

<https://packt.live/36t6xM5>.

With this exercise completed, you have now completed half of the work needed to develop the UI element of tracking the number of coins collected by the player. The next half will be using the functionality developed in this activity inside UMG to show this information to the player on-screen. To do this, we need to learn more about UMG inside of Unreal Engine 4.

UMG

UMG, or Unreal Motion Graphics UI Designer, is Unreal Engine 4's main tool for creating UI for things such as menus, in-game HUD elements such as health bars, and other user interfaces you may want to present to the player.

In the **SuperSideScroller** game, we will only be using the

Text widget to construct our **Coin Collection UI** in *Exercise 15.04, Creating the Coin Counter UI HUD Element*. We'll learn more about the **Text** widget in the next section.

Text Widget

The **Text** widget is one of the simpler widgets that exists. This is because it only allows you to display text information to the user and customize the visuals of this text. Almost every single game uses text in one way or another to display information to its players. Overwatch, for example, uses a text-based UI to display crucial match data to its players. Without the use of text, it would be very difficult – maybe even impossible – to convey key pieces of statistical data to the player, such as total damage dealt, total time playing the game, and many others.

The **Text** widget appears in the **Palette** tab within UMG. When you add a **Text** widget to the **Canvas** panel, it will display the text **Text Block** by default. You can customize this text by adding your text to the **Text** parameter of the widget. Alternatively, you can use **Function Binding** to display more robust text that can reference internal or external variables. **Function Binding** should be used whenever you need to display information that can change; this could be text that represents a player's score, how much money the player has, or in our case, the number of coins the player has collected:

You will be using the **Function Binding** functionality of the **Text** widget to display the number of coins collected by the player using the **GetCurrentNumberofCollectables()** function you created in *Exercise 15.03, Tracking the Number of Coins for the Player*.

Now that we have the **Text** widget in the **Canvas** panel, it is

time to position this widget where we need it to be. For this, we will take advantage of Anchors.

Anchors

Anchors are used to define where a widget's desired location should be on the **Canvas** panel. Once defined, the **Anchor** will ensure that the widget will maintain this position with varying screen sizes through different platform devices such as phones, tablets, and computers. Without an anchor, a widget's position can become inconsistent between different screen resolutions, which is never desired.

Note

For more information about Anchors, please refer to the following documentation:

<https://docs.unrealengine.com/en-US/Engine/UMG/UserGuide/Anchors/index.html>.

For the purposes of our **Coin Collection UI** and the **Text** widget you will use, the **Anchor** point will be at the top-left corner of the screen. You will also add a position offset from this **Anchor** point so that the text is more visible and readable to the player. Before moving onto creating our **Coin Collection UI**, let's learn about **Text Formatting**, which you will use to display the current number of collected coins to the player.

Text Formatting

Much like the **UE_LOG()** macro available to us in C++, Blueprints offers a similar solution to display text and format

the text to allow custom variables to be added to it. The **Format Text** function takes in a single text input labeled **Format** and returns the **Result** text out. This can then be used to display information:

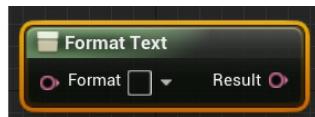


Figure 15.5: The **Format Text** function allows us to customize the text using formatted arguments that can be passed in

Instead of using the % symbol like **UE_LOG()** does, the **Format Text** function uses the {} symbols to denote arguments that can be passed into the string. In-between the {} symbols, you need to add an argument name; this can be anything you want, but it should be representative of what the argument is. Refer to the example shown in the following screenshot:

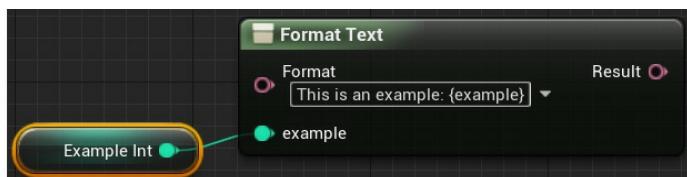


Figure 15.6: Here, we are passing an example integer into the formatted text

The **Format Text** function only supports **Byte**, **Integer**, **Float**, **Text**, or **EText Gender** variable types, so if you are attempting to pass any other type of variable into the function as an argument, you must convert it into one of the supported types.

Note

*The **Format Text** function is also used for **Text***

Localization, where you can support multiple languages for your game. More information about how this can be done in both C++ and Blueprints can be found here:
<https://docs.unrealengine.com/en-US/Gameplay/Localization/Formatting/index.html>.

You will be using the **Format Text** function in conjunction with the **Text** widget in UMG in the next exercise, where we will be creating the **Coin Counter UI** widget to display the number of coins that have been collected by the player. You will also be using **Anchors** to position the **Text** widget at the top-left corner of the screen.

Exercise 15.04: Creating the Coin Counter UI HUD Element

In this exercise, you will be creating the UMG UI asset, which will display and update the number of coins collected by the player. You will use the **GetCurrentNumberofCollectables()** inline function you created in *Exercise 15.02, Creating the PickableActor_Collectable Class*, to display this value on the screen using a simple **Text** widget. Follow these steps to accomplish this:

1. Let's start by creating a new folder inside the **Content Browser** window called **UI**. Do this by *right-clicking* on the **Content** folder at the top of the browser directory in the editor and

selecting **New Folder**.

2. Inside the new **/Content/UI** directory, *right-click* and instead of selecting **Blueprint Class**, hover over the **User Interface** option at the bottom of this list and *left-click* the **Widget Blueprint** option.
3. Name this new **Widget Blueprint BP_UI_CoinCollection**, and then *double-left-click* the asset to open the UMG editor.
4. By default, the **Widget** panel is empty, and you will find an empty hierarchy on the left-hand side, as shown in the following screenshot:

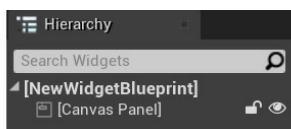


Figure 15.7: The Widget panel Hierarchy outlines how the different elements of the UI are layered with one another

5. Above the **Hierarchy** tab is the **Palette** tab, which lists all the available widgets you can use inside

your UI. We will only focus on the **Text** widget, which is listed under the **Common** category. Do not mistake this option with the Rich Text Block widget.

Note

*For a more detailed reference to all the available **Widgets** inside UMG, please read the following documentation from Epic Games:
<https://docs.unrealengine.com/en-US/Engine/UMG/UserGuide/WidgetTypeReference/index.html>.*

6. Add the **Text** widget to the **UI** panel by either *left-clicking* and dragging the **Text** widget from the **Palette** tab to the **Hierarchy** tab underneath the **Canvas** panel root, or by *left-clicking* and dragging the **Text** widget directly into the **Canvas** panel itself in the middle of the UMG editor.

Before changing the text of this widget, we need to update its anchor, position, and font size in order to fit the needs we have for displaying the information to the player.

7. With the **Text** widget selected, you will see many options under its **Details** panel to customize this text. The first thing to do here is anchor the **Text** widget to the top-left corner of the **Canvas** panel. *Left-click* on the **Anchors** dropdown and select the top-left anchoring option, as shown in the following screenshot:

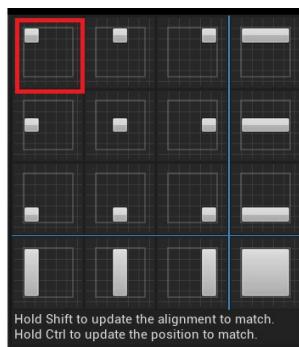


Figure 15.8: By default, there are options to anchor a widget at different locations of the screen

Anchoring allows the widget to maintain its desired location within the **Canvas** panel, regardless of varying screen sizes.

Now that the **Text** widget is anchored to the top-left corner, we need to set its relative position to this anchor so that there is an offset for better positioning

and readability of the text.

8. In the **Details** panel, underneath the **Anchors** option, are parameters for **Position X** and **Position Y**. Set both these parameters to **100.0f**.
9. Next, enable the **Size To Content** parameter so that the size of the **Text** widget will automatically resize itself, depending on the size of the text it is displaying, as shown in the following screenshot:

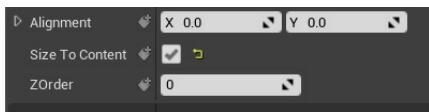


Figure 15.9: The Size To Content parameter will ensure that the Text widget will display its full content and not be cut off

10. The last thing we need to do here is to update the size of the font used for the **Text** widget. Underneath the **Appearance** tab of the **Details** panel for the **Text** widget, you will find the **Size** parameter. Set this value to **48**.

11. The final **Text** widget will look like this:

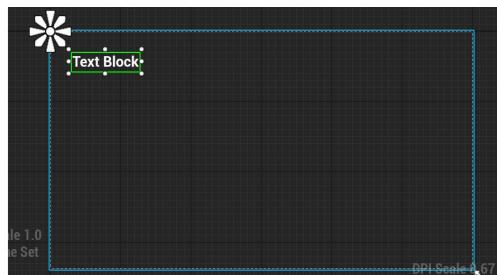


Figure 15.10: The Text widget is now anchored to the top-left of the Canvas panel, with a small relative offset and a larger font for better readability for the player

Now that we have the **Text** widget positioned and sized the way we need it to be, let's add a new binding to the text so that it will automatically update and match the value of the number of collectibles the player has.

12. With the **Text** widget selected, find the **Text** parameter in its **Details** panel, under the **Content** category. There, you will find the **Bind** option.
13. *Left-click* the **Bind** option and select **Create Binding**. When doing this, the new **Function Binding** will be

created automatically and be given the name **GetText_0**. Please refer to the following screenshot:



Figure 15.11: It is important to always rename the bind functions because their default names are too generic

14. Rename this function **Get Number of Collectables**.
15. Before continuing with this function, create a new object reference variable called **Player** that's of the **SuperSideScroller_Player** type. Make this variable **Public** and exposable on spawn by enabling both the **Instance Editable** and **Expose on Spawn** parameters of the variable, as shown in the following screenshot:

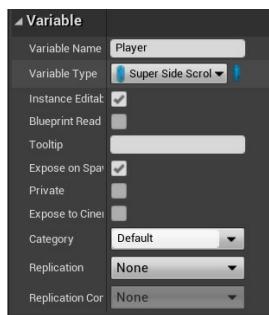


Figure 15.12: The Player variable should now have the Instance Editable and Expose on Spawn parameters enabled

By making the **Player** variable **Public** and exposed on spawn, you will be able to assign this variable when creating the widget and adding it to the screen. We will do this in *Exercise 15.05, Adding the Coin Counter UI to the Player Screen.*

Now that we have a reference variable to **SuperSideScroller_Player**, let's continue with the **Get Number of Collectables** bind function.

16. Add a **Getter** of the **Player** variable to the **Get Number of Collectables** function.
17. From this variable, *left-click* and drag and from the context-sensitive drop-down menu, and find and select the **Get Current Number of Collectables** function. Please refer to the following screenshot:

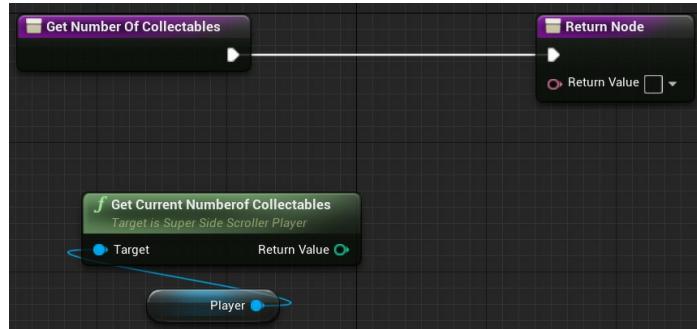


Figure 15.13: The Get Current Numberof Collectables C++ function you created in Exercise

15.03

18. Next, *left-click* and drag out the **Return Value** text parameter of the **Get Number of Collectables** to **Return Node**. From the context-sensitive drop-down menu, search for and select the **Format Text** option, as shown in the following screenshot:

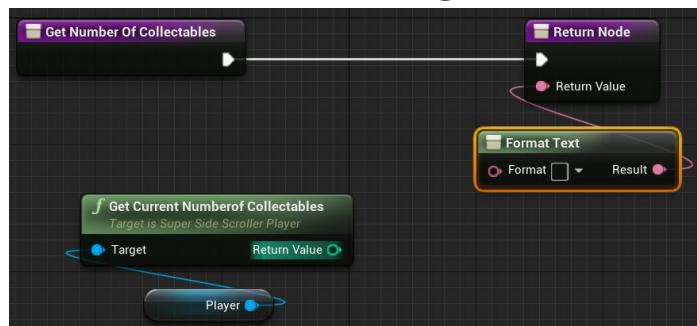


Figure 15.14: Now, we can create customized and formatted text to suit the needs of the text

19. Within the **Format Text** function,

add the following text:

Coins: {coins}

Please refer to the following screenshot:

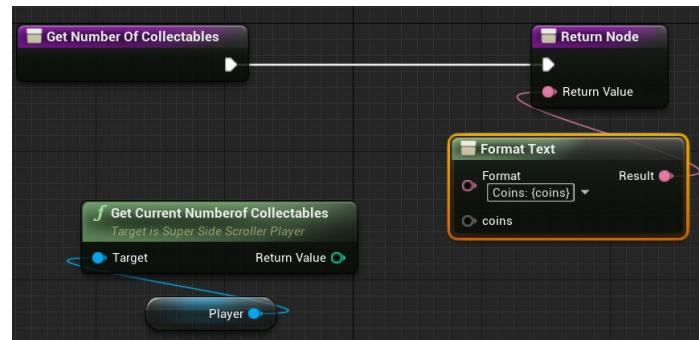


Figure 15.15: Now, there is a new input argument to the formatted text that we can use to display custom information

Remember that using the {} symbols denotes a text argument that allows you to pass variables into the text.

20. Finally, connect the int **Return Value** of the **GetCurrentNumberofCollectables()** function to the wildcard **coins** input pin of the **Format Text** function, as shown here:

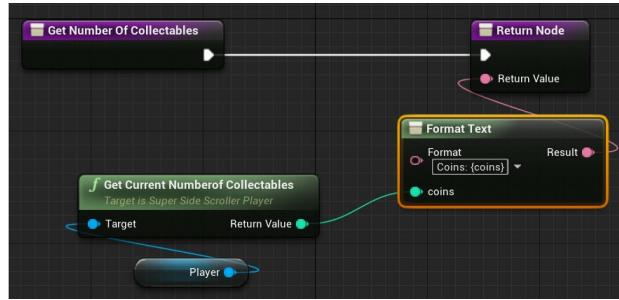


Figure 15.16: Now, the Text widget will update automatically based on the updated value returned from the Get Current Numberof Collectables function

21. Compile and save the **BP_UI_CoinCollection** widget Blueprint.

Note

*You can find the assets and code for this exercise here:
<https://packt.live/3eQJjTU>.*

With this exercise completed, you have created the **UI UMG** widget needed to display the current number of coins collected by the player. By using the **GetCurrentNumberofCollectables()** C++ function and the binding functionality of the **Text** widget, the UI will always update its value based on the number of coins collected. In the next exercise, we will add this UI to the player's screen, but first, we'll briefly learn about how to add and remove UMG from the player screen.

Adding and Creating UMG User Widgets

Now that we have created the Coin Collection UI in UMG, it is time to learn how to add and remove the UI to and from the player screen. By adding the Coin Collection UI to the player screen, the UI becomes visible to the player and can be updated as the player collects coins.

In Blueprints, there is a function called **Create Widget**, as shown in the following screenshot. Without a class assigned, it will be labeled **Construct None**, but do not let this confuse you:

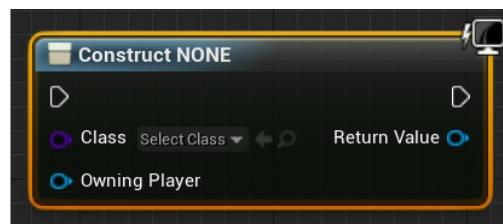


Figure 15.17: The Create widget as it is by default, without a class applied

This function requires the class of the **User** widget to be created and requires a **Player Controller** that will be referenced as the owning player of this UI. This function then returns the spawned user widget as its **Return Value**, where you can then add to the player's viewport using the **Add to Viewport** function. The **Create Widget** function only instantiates the widget object; it does not add this widget to the player's screen. It is the **Add to Viewport** function that makes this widget visible on the player's screen:

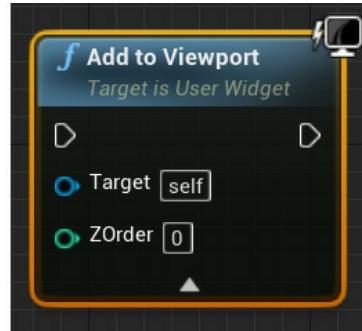


Figure 15.18: Add to Viewport function with ZOrder

The viewport is the game screen that overlays your view of the game world, and it uses what is called **ZOrder** to determine the overlay depth in cases where multiple UI elements need to overlap above or below one another. By default, the **Add to Viewport** function will add the **User** widget to the screen and make it fill the entire screen; that is, unless the **Set Desired Size In Viewport** function is called to set the size that it should fill manually:

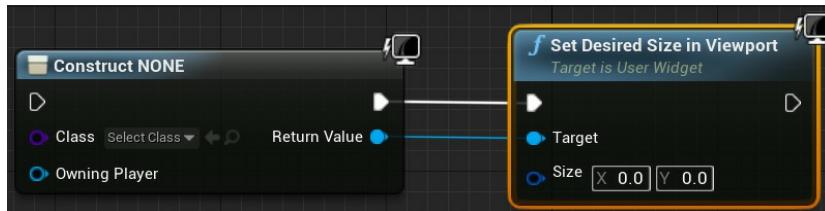


Figure 15.19: The Size parameter determines the desired size of the passed in User widget

In C++, you also have a function called **CreateWidget()**:

```
template<typename WidgetT, typename OwnerT>
WidgetT * CreateWidget
(
    OwnerT * OwningObject,
    TSubclassOf < UUserWidget >
    UserWidgetClass,
```

```
    FName WidgetName  
)  
}
```

The **CreateWidget()** function is available through the **UserWidget** class, which can be found in **/Engine/Source/Runtime/UMG/Public/Blueprint/UserWidget.h**.

An example of this can be found in *Chapter 8, User Interfaces*, where you used the **CreateWidget()** function to create **BP_HUDWidget**:

```
HUDWidget = CreateWidget<UHUDWidget>(this,  
BP_HUDWidget);
```

Refer back to *Chapter 8, User Interfaces*, and *Exercise 8.06, Creating the Health Bar C++ Logic*, for more information regarding the **CreateWidget()** function in C++.

This function works almost identically to its Blueprint counterpart because it takes in the **Owning Object** parameter, much like the **Owning Player** parameter of the Blueprint function, and it requires the **User Widget** class to be created. The C++ **CreateWidget()** function also takes in an **FName** parameter to represent the widget's name.

Now that we have learned about the methods to use to add UI to the player screen, let's put this knowledge to the test. In the following exercise, you will be implementing the **Create Widget** and **Add to Viewport** Blueprint functions so that we can add the coin collection UI that we created in *Exercise 15.04, Creating the Coin Counter UI HUD Element*, to the player screen.

Exercise 15.05: Adding

the Coin Counter UI to the Player Screen

In this exercise, you will be creating a new **Player Controller** class so that you can use the player controller to add the **BP_UI_CoinCollection** widget Blueprint to the player's screen. From there, you will also create a new **Game Mode** class and apply this game mode to the **SuperSideScroller** project. Perform the following steps to complete this exercise:

1. In the Unreal Engine 4 editor, navigate to **File** and then **New C++ Class**.
2. From the **Choose Parent Class** dialog window, find and select the **Player Controller** option.
3. Name the new **Player Controller** class **SuperSideScroller_Controller** and then *left-click* the **Create Class** button. Visual Studio will automatically generate and open the source and header files for the **SuperSideScroller_Controller** class, but for now, we will stay inside the Unreal Engine 4 editor.
4. In the **Content Browser** window,

under the **MainCharacter** folder directory, create a new folder called **PlayerController**.

5. In the **PlayerController** folder, *right-click* and create a new **Blueprint Class** using the new **SuperSideScroller_Controller** class. Please refer to the following screenshot:

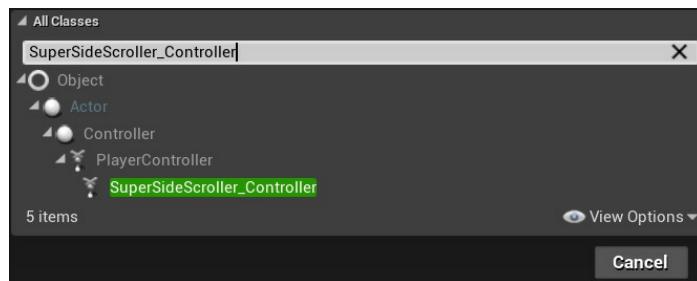


Figure 15.20: Finding the new SuperSideScroller_Controller class to create a new Blueprint from

6. Name this new Blueprint **BP_SuperSideScroller_PC** and then *double-left-click* the asset to open it.

To add the **BP_UI_CoinCollection** widget to the screen, we need to use the **Add to Viewport** function and the **Create Widget** function. We

want the UI to be added to the player's screen after the player character has been **Possessed** by the player controller.

7. *Right-click* inside the Blueprint graph and from the context-sensitive menu, find the **Event On Possess** option and *left-click* to add it to the graph. Please refer to the following screenshot:

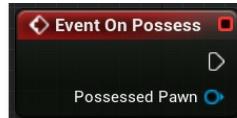


Figure 15.21: The Event On Possess option will be called each time this controller class possesses a new pawn

The **Event On Possess** event node returns **Possessed Pawn**. We will use this pawn to pass into our **BP_UI_CoinCollection UI Widget**, but first, we need to **Cast To** the **SuperSideScroller_Player** class.

8. *Left-click* and drag from the output the **Possessed Pawn** parameter of the

Event On Possess node. Then, search for and find the **Cast to SuperSideScroller_Player** node.

Please refer to the following screenshot:



Figure 15.22: We need to Cast To SuperSideScroller_Player to ensure we are casting to the right player character class

9. Now, *right-click* and search for the **Create Widget** function to add it to the Blueprint graph.
10. From the drop-down class parameter, find and assign the **BP_UI_CoinCollection** asset you created in *Exercise 15.04, Creating the Coin Counter UI HUD Element*. Please refer to the following screenshot:



Figure 15.23: The Create Widget

function will create a new UI object using the UMG class passed into it

After updating the **Class** parameter to the **BP_UI_CoinCollection** class, you will notice that the **Create Widget** function will update to show the **Player** variable you created, set to **Exposed on Spawn**.

11. *Right-click* in the Blueprint graph to search for and find the **Self** reference variable from the context-sensitive drop-down menu. Connect the **Self** object variable to the **Owning Player** parameter of the **Create Widget** function, as shown in the following screenshot:



Figure 15.24: The Owning Player input parameter is of the Player Controller type

The **Owning Player** parameter refers

to the **Player Controller** type that will show and own this UI object. Since we are adding this UI to the **SuperSideScroller_Controller** Blueprint, we can just use the **Self** reference variable to pass into the function.

12. Next, pass in the returned **SuperSideScroller_Player** variable from the **Cast** node to the **Player** input node of the **Create Widget** function. Then, connect the execution pins of the **Cast** node and the **Create Widget** function, as shown in the following screenshot:



Figure 15.25: If the Cast To SuperSideScroller_Player is valid, we can create the BP_UI_CoinCollection widget and pass in the player that has been possessed

Note

You can find the preceding screenshot in full resolution for better viewing at the following link:
<https://packt.live/3f89m99>.

13. Right-click inside the Blueprint graph again to search and find the **Add to Viewport** function so that you can place it in the graph.
14. Connect the output **Return Value** parameter of the **Create Widget** function to the **Target** input parameter of the **Add to Viewport** function; do not change the **ZOrder** parameter.
15. Lastly, connect the execution pins of the **Create Widget** and **Add to Viewport** functions, as shown here:



Figure 15.26: After creating the BP_UI_CoinCollection widget, we can add it to the player viewport

Note

*You can find the preceding screenshot in full resolution for better viewing at the following link:
<https://packt.live/2UwufBd>.*

Now that the player controller adds the **BP_UI_CoinCollection** widget to the player's viewport, we need to create a **GameMode** Blueprint and apply both the **BP_SuperSideScroller_MainCharacter** and **BP_SuperSideScroller_PC** classes to this game mode.

16. In the **Content Browser** window, create a new folder by *right-clicking* the **Content** folder and selecting **New Folder**. Name this folder **GameMode**.
17. Next, *right-click* and select **Blueprint Class** to begin creating the game mode Blueprint. From the **Pick Parent Class** dialog window, search for and find **SuperSideScrollerGameMode** under **All Classes**.
18. Name this new **GameMode** Blueprint

BP_SuperSideScroller_GameMode

. *Double-left-click* this asset to open it.

The **GameMode** Blueprint contains a list of classes that you can customize with your unique classes. For now, we will only worry about **Player Controller Class** and **Default Pawn Class**.

19. *Left-click* the **Player Controller Class** dropdown to find and select the **BP_SuperSideScroller_PC** Blueprint you created earlier in this exercise.
20. Then, *left-click* the **Default Pawn Class** dropdown to find and select the **BP_SuperSideScroller_MainCharacter** Blueprint.

Now that we have a custom **GameMode** that utilizes our custom **Player Controller** and **Player Character** classes, let's add this game mode to the **Project Settings** window so that the game mode is used by default when using PIE and when cooking builds of the project.

21. From the Unreal Engine 4 editor, navigate to the **Edit** option at the top of the screen. *Left-click* this option and from the drop-down menu, find and select the **Project Settings** option.
22. On the left-hand side of the **Project Settings** window, you are provided with a list of categories divided into sections. Under the **Project** section, *left-click* the **Maps & Modes** category.
23. In the **Maps & Modes** section, you have a handful of parameters related to your project's default maps and game mode. At the top of this section, you have the **Default GameMode** option. *Left-click* this dropdown to find and select the **SuperSideScroller_GameMode** Blueprint you created earlier in this exercise.

Note

*Changes made to the **Maps & Modes** section are automatically saved and written to the **DefaultEngine.ini** file, which can be found in your*

*project's **Config** folder. Default **GameMode** can be overwritten per level by updating the **GameMode Override** parameter, which can be found in the **World Settings** window of your level.*

24. Close the **Project Settings**

window and return to your level. Use PIE and start collecting coins. Observe that the **BP_UI_CoinCollection** widget is shown and updated each time you collect a coin, as shown in the following screenshot:

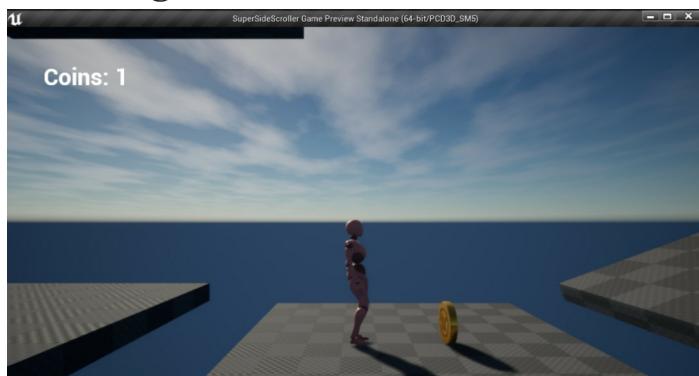


Figure 15.27: Now, every coin you collect will appear on the player UI

Note

*You can find the assets and code for this exercise here:
<https://packt.live/2JRfSFz>.*

With this exercise complete, you have created the **UI UMG**

widget needed to display the current number of coins collected by the player. By using the **GetCurrentNumberofCollectables()** C++ function and the binding functionality of the **Text** widget, the UI will always update its value based on the number of coins collected.

So far, we have focused on the collectible coin and allowing players to collect these coins and add the total coins collected to the player's UI. Now, we will focus on the potion power-up and granting movement speed and jump height increases to the player for a short period of time. To implement this functionality, we first need to study timers.

Timers

Timers in Unreal Engine 4 allow you to perform actions after a delay or every X number of seconds. In the case of the **SuperSideScroller** potion power-up, a timer will be used to restore the player's movement and jump to their defaults after 8 seconds.

Note

In Blueprints, you can use a Delay node in addition to Timer Handles to achieve the same results. However, in C++, Timers are the best means to achieve delays and reoccurring logic.

Timers are managed by **Timer Manager**, or **FTimerManager**, which exists in the **UWorld** object. There are two main functions that you will be using from the **FTimerManager** class, called **SetTimer()** and **ClearTimer()**:

```

void SetTimer
(
    FTimerHandle & InOutHandle,
    TFunction < void > && Callback,
    float InRate,
    bool InbLoop,
    float InFirstDelay
)
void ClearTimer(FTimerHandle& InHandle)

```

You may have noticed that, in both functions, there is a required **FTimerHandle**. This handle is used to control the timer you have set. Using this handle, you can pause, resume, clear, and even extend the timer.

The **SetTimer()** function also has other parameters to help you customize this **Timer** when initially setting it. The callback function will be called after the **Timer** has been completed, and if the **InbLoop** parameter is **True**, it will continue to call the callback function indefinitely, until the timer has been stopped. The **InRate** parameter is the duration of the timer itself, while **InFirstDelay** is an initial delay that's applied to the timer before it begins its timer for **InRate**.

The header file for the **FTimerManager** class can be found here:
[/Engine/Source/Runtime/Engine/Public/TimerManager.h](#).

Note

*You can learn more about timers and **FTimerHandle** by*

reading the documentation here:

<https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Timers/index.html>.

In the following exercise, you will create your own **FTimerHandle** in the **SuperSideScroller_Player** class and use it to handle to control how long the effects of the potion power-up last on the player.

Exercise 15.06: Adding the Potion Power-Up Behavior to the Player

In this exercise, you will be creating the logic behind the potion power-up and how it will affect the player character. You will take advantage of timers and timer handles to ensure that the power-up effects only last for a short duration. Follow these steps to accomplish this:

1. In Visual Studio, navigate to and open the **SuperSideScroller_Player.h** header file.
2. Under **our Private Access Modifier**, add a new variable of the **FTimerHandle** type and name it **PowerupHandle**:

```
FTimerHandle PowerupHandle;
```

This timer handle will be responsible

for keeping track of how much time has elapsed since it was initiated. This will allow us to control how long the potion power-up's effects will last.

3. Next, add a Boolean variable under our **Private Access Modifier** called **bHasPowerupActive**:

```
bool bHasPowerupActive;
```

We will use this Boolean variable when updating the **Sprint()** and **StopSprinting()** functions to ensure we update the player's sprint movement speed appropriately based on whether the power-up is active.

4. Next, declare a new void function called **IncreaseMovementPowerup()** under our **Public Access Modifier**:

```
void  
IncreaseMovementPowerup();
```

This is the function that will be called from the potion power-up class to enable the effects of the power-up for the player.

- Finally, you need to create a function that handles when the power-up effects end. Create a function called **EndPowerup()** under **Protected Access Modifier**:

```
void EndPowerup();
```

With all the necessary variables and functions declared, it's time to start defining these new functions and handling the power-up effects on the player.

- Navigate to the **SuperSideScroller_Player.cpp** source file.
- First, add the header file **#include "TimerManager.h"** to the top of the source file; we will need this class in order to use **Timers**.
- Define the **IncreaseMovementPowerup()** function by adding the following code to the source file:

```
void
ASuperSideScroller_Player::
IncreaseMovementPowerup()
```

```
{  
}
```

9. When this function is called, the first thing we need to do is set the **bHasPowerupActive** variable to **true**. Add the following code to the **IncreaseMovementPowerup()** function:

```
bHasPowerupActive = true;
```

10. Next, add the following code to increase both the **MaxWalkSpeed** and **JumpZVelocity** components of the player character movement component:

```
GetCharacterMovement() ->MaxWalkSpeed = 500.0f;
```

```
GetCharacterMovement() ->JumpZVelocity = 1500.0f;
```

Here, we are changing **MaxWalkSpeed** from the default **300.0f** value to **500.0f**. As you may recall, the default sprinting speed is also **500.0f**. We will address this later in this activity to increase the sprinting speed when the power-up is active.

11. To take advantage of timers, we need to get a reference to the **UWorld** object. Add the following code:

```
UWorld* World = GetWorld();  
  
if (World)  
{  
}  
}
```

As we've done many times before in this project, we're using the **GetWorld()** function to get a reference to the **UWorld** object and saving this reference in its variable.

12. Now that we have the reference to the **World** object and have performed a validity check, it is safe to use the **TimerManager** to set the power-up timer. Add the following code within the **if()** statement shown in the previous step:

```
World-  
>GetTimerManager().SetTimer  
(PowerupHandle, this,  
&ASuperSideScroller_Player:  
:EndPowerup, 8.0f, false);
```

Here, you are using the **TimerManager** class to set a timer. The **SetTimer()** function takes in the **FTimerHandle** component to use; in this case, the **PowerupHandle** variable you created. Next, we need to pass in a reference to the player class by using the **this** keyword. Then, we need to provide the callback function to call after the timer has ended, which in this case is the **&ASuperSideScroller_Player::EndPowerup** function. **8.0f** represents the duration of the timer; feel free to adjust this as you see fit, but for now, 8 seconds is fine. Lastly, there is the parameter that determines whether this timer should loop; in this case, it should not.

13. Create the function definition for the **EndPowerup()** function:

```
void
ASuperSideScroller_Player::
EndPowerup()

{



}
```

14. The first thing to do when the **EndPowerup()** function is called is set the **bHasPowerupActive** variable to **false**. Add the following code within the **EndPowerup()** function:

```
bHasPowerupActive = false;
```

15. Next, change the **MaxWalkSpeed** and **JumpZVelocity** parameters of the character movement component back to their default values. Add the following code:

```
GetCharacterMovement() ->MaxWalkSpeed = 300.0f;  
GetCharacterMovement() ->JumpZVelocity = 1000.0f;
```

Here, we are changing both the **MaxWalkSpeed** and **JumpZVelocity** parameters of the character movement component to their default values.

16. Again, to take advantage of the timers and to clear the timer to handle **PowerupHandle**, we need to get a reference to the **UWorld** object. Add this code:

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

```
if (world)
{
}
```

17. Finally, we can add the code to clear the timer handle's **PowerupHandle**:

```
World-
>GetTimerManager().ClearTim
er(PowerupHandle);
```

By using the **ClearTimer()** function and passing in **PowerupHandle**, we are ensuring that this timer is no longer valid and will no longer affect the player.

Now that we have created the functions that handle the power-up effects and the timer associated with the effects, we need to update both the **Sprint()** and **StopSprinting()** functions so that they also take into account the speed of the player when the power-up is active.

18. Update the **Sprint()** function to the following:

```
void
```

```

ASuperSideScroller_Player::
Sprint()

{
    if (!bIsSprinting)
    {
        bIsSprinting = true;
        if (bHasPowerupActive)
        {
            GetCharacterMovement(
            )->MaxWalkSpeed = 900.0f;
        }
        else
        {
            GetCharacterMovement(
            )->MaxWalkSpeed = 500.0f;
        }
    }
}

```

Here, we are updating the **Sprint()** function to take into account whether **bHasPowerupActive** is true. If this variable is true, then we increase

MaxWalkSpeed while sprinting from **500.0f** to **900.0f**, as shown here:

```
if (bHasPowerupActive)
{
    GetCharacterMovement()-
    >MaxWalkSpeed = 900.0f;
}
```

If **bHasPowerupActive** is false, then we increase **MaxWalkSpeed** to **500.0f**, as we did by default.

19. Update the **StopSprinting()** function to the following:

```
void
ASuperSideScroller_Player::
StopSprinting()
{
    if (bIsSprinting)
    {
        bIsSprinting = false;
        if (bHasPowerupActive)
        {
            GetCharacterMovement(
```

```

) ->MaxWalkSpeed = 500.0f;

}

else

{

    GetCharacterMovement(
) ->MaxWalkSpeed = 300.0f;

}

}

```

Here, we are updating the **StopSprinting()** function to take into account whether **bHasPowerupActive** is true. If this variable is true, then we set the **MaxWalkSpeed** value to **500.0f** instead of **300.0f**, as shown here:

```

if (bHasPowerupActive)

{

    GetCharacterMovement()-
->MaxWalkSpeed = 500.0f;

}

```

If **bHasPowerupActive** is false, then we set **MaxWalkSpeed** to **300.0f**, as

we did by default.

20. Finally, all we need to do is recompile the C++ code.

Note

*You can find the assets and code for this exercise here:
<https://packt.live/3eP39yL>.*

With this exercise complete, you have created the potion power-up effects within the player character. The power-up increases both the default movement speed of the player and increases their jump height. Moreover, the effects of the power-up increase the sprinting speed. By using timer handles, you were able to control how long the power-up effect would last.

Now, it is time to create the potion power-up actor so that we can have a representation of this power-up in the game.

Activity 15.03: Creating the Potion Power-Up Actor

Now that the **SuperSideScroller_Player** class handles the effects of the potion power-up, it's time to create the potion power-up class and Blueprint. The aim of this activity is to create the potion power-up class, inherit from the **PickableActor_Base** class, implement the overlap

functionality to grant the movement effects that you implemented in *Exercise 15.06, Adding the Potion Power-Up Behavior to the Player*, and to create the Blueprint actor for the potion power-up. Perform these steps to create the potion power-up class and to create the potion Blueprint actor:

1. Create a new C++ class that inherits from the **PickableActor_Base** class and name this new class **PickableActor_Powerup**.
2. Add the override function declarations for both the **BeginPlay()** and **PlayerPickedUp()** functions.
3. Create the function definition for the **BeginPlay()** function. Within the **BeginPlay()** function, add the call to the parent class function.
4. Create the function definition for the **PlayerPickedUp()** function. Within the **PlayerPickedUp()** function, add the call to the **PlayerPickedUp()** parent class function.
5. Next, add the necessary **#include** file for the **SuperSideScroller_Player** class so that we can reference the player

class and its functions.

6. In the **PlayerPickedUp()** function, use the **Player** input parameter of the function itself to make the function call to **IncreaseMovementPowerup()**.
7. From **Epic Games Launcher**, find the **Action RPG** project from the **Learn** tab, under the **Games** category. Use this to create and install a new project.
8. Migrate the **A_Character_Heal_Mana_Cue** and **SM_PotionBottle** assets, as well as all of their referenced assets, from the **Action RPG** project to your **SuperSideScroller** project.
9. Create a new folder in the **Content Browser** window within the **PickableItems** directory called **Powerup**. Create a new Blueprint within this directory based on the **PickableActor_Powerup** class and name this asset **BP_Powerup**.
10. In **BP_Powerup**, update the

MeshComp component in order to use the **SM_PotionBottle** static mesh.

11. Next, add **A_Character_Heal_Mana_Cue**, which you imported as the **Pickup Sound** parameter.
12. Finally, update the **RotationComp** component so that the actor will rotate 60 degrees per second around the **Pitch** axis and rotate 180 degrees per second around the **Yaw** axis.
13. Add **BP_Powerup** to your level and use PIE to observe the results when overlapping with the power-up.

Expected output:



Figure 15.28: The potion power-up now has a nice visual representation and can be overlapped by the player to enable its power-up effects

Note

The solution to this activity can be found at:

<https://packt.live/338jEBx>.

With this activity complete, you were able to put your knowledge to the test in terms of creating a new C++ class that inherits from the **PickableActor_Base** class and overrides the **PlayerPickedUp()** function to add custom logic. By adding the call to the **IncreaseMovementPowerup()** function from the player class, you were able to add the movement power-up effects to the player when overlapping with the actor. Then, by using a custom mesh, material, and audio assets, you were able to bring the Blueprint actor to life from the **PickableActor_Powerup** class.

Now that we have created the coin collectible and the potion power-up, we need to implement a new gameplay feature into the project: the **Brick** class. In games such as Super Mario, bricks contain hidden coins and power-ups for the players to find. These bricks also serve as a means of reaching elevated platforms and areas within the level. In our **SuperSideScroller** project, the **Brick** class will serve the purpose of containing hidden coin collectibles for the player, and as a means of allowing the player to reach areas of the level by using the bricks as paths to access hard-to-reach locations. So, in the next section, we will create the **Brick** class that needs to be broken to find the hidden coins.

Exercise 15.07: Creating the Brick Class

Now that we have created the coin collectible and the potion power-up, it is time to create the **Brick** class, which will contain hidden coins for the player. The brick is the final gameplay element of the **SuperSideScroller** project. In

this exercise, you will be creating the **Brick** class, which will be used as part of the platforming mechanic of the **SuperSideScroller** game project, but also as a means to hold collectibles for players to find. Follow these steps to create this **Brick** class and its Blueprint:

1. In the Unreal Engine 4 editor, navigate to **File** and then **New C++ Class**.
2. From the **Choose Parent Class** dialog window, find and select the **Actor** class.
3. Name this class **SuperSideScroller_Brick** and *left-click Create Class*. Visual Studio and Unreal Engine will recompile the code and open this class for you.

By default, the **SuperSideScroller_Brick** class comes with the **Tick()** function, but we will not need this function for the **Brick** class. Remove the function declaration for **Tick()** from the **SuperSideScroller_Brick.h** header file and remove the function definition from the **SuperSideScroller_Brick.cpp**

source file before continuing.

4. Under the **Private Access Modifier** for the **SuperSideScroller_Brick.h** file, add the following code to declare a new **UStaticMeshComponent*** **UPROPERTY()** function to represent the brick in our game world:

```
UPROPERTY(VisibleDefaultsOn  
ly, Category = Brick)  
  
class UStaticMeshComponent*  
BrickMesh;
```

5. Next, we need to create a **UBoxComponent** **UPROPERTY()** that will handle the collision with the player character. Add the following code to add this component under our **Private Access Modifier**:

```
UPROPERTY(VisibleDefaultsOn  
ly, Category = Brick)  
  
class UBoxComponent*  
BrickCollision;
```

6. Create the **UFUNCTION()** declaration for the **OnHit()** function under our

Private Access Modifier. This will be used to determine when **UBoxComponent** is hit by the player:

```
UFUNCTION()  
void  
OnHit(UPrimitiveComponent*  
HitComp, AActor*  
OtherActor,  
UPrimitiveComponent*  
OtherComp, FVector  
NormalImpulse, const  
FHitResult& Hit);
```

Note

*Recall that you used the **OnHit()** function when developing the **PlayerProjectile** class in Chapter 13, Enemy Artificial Intelligence, for this project. Please review that chapter for more information about the **OnHit()** function.*

7. Next, create a new Boolean **UPROPERTY()** under our **Private Access Modifier** using the **EditAnywhere** keyword called **bHasCollectable**:

```
UPROPERTY(EditAnywhere)  
bool bHasCollectable;
```

This Boolean will determine whether the brick contains a coin collectible for the player.

8. Now, we need a variable that holds how many coin collectibles are available within this brick for the player. We will do this by creating an integer variable called **Collectable Value**. Make this a **UPROPERTY()**, under the **private access modifier**, with the **EditAnywhere** keyword, and give it a default value of **1**, as shown here:

```
UPROPERTY(EditAnywhere)  
int32 CollectableValue = 1;
```

The brick will need to contain a unique sound and particle system so that it has a nice layer of polish for when the brick is destroyed by the player. We'll add these properties next.

9. Create a new **Public Access Modifier** in the

SuperSideScroller_Brick.h

header file.

10. Next, create a new **UPROPERTY()** using the **EditAnywhere** and **BlueprintReadOnly** keywords for a variable of the **USoundBase** class. Name this variable **HitSound**, as shown here:

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

11. Then, create a new **UPROPERTY()** using the **EditAnywhere** and **BlueprintReadOnly** keywords for a variable of the **UParticleSystem** class. Make sure to put this under the **public access modifier**, and name this variable **Explosion**, as shown here:

```
UPROPERTY(EditAnywhere,  
BlueprintReadOnly, Category  
= Brick)  
  
class UParticleSystem*  
Explosion;
```

Now that we have all the necessary properties for the **Brick** class, let's move onto the **SuperSideScroller_Brick.cpp** source file, where we will initialize the components.

12. Let's start by adding the following **#include** directories for **StaticMeshComponent** and **BoxComponent**. Add the following code to the **#include** list of the source file:

```
#include  
"Components/StaticMeshCompo  
nent.h"  
  
#include  
"Components/BoxComponent.h"
```

13. First, initialize the **BrickMesh** component by adding the following code to the **ASuperSideScroller_Brick::ASuperSideScroller_Brick()** constructor function:

```
BrickMesh =  
CreateDefaultSubobject<USta  
ticMeshComponent>
```

```
(TEXT("BrickMesh"));
```

14. Next, the **BrickMesh** component should have collision so that the player can walk on top of it for platforming gameplay purposes. To ensure this occurs by default, add the following code to set the collision to **"BlockAll"**:

```
BrickMesh-
>SetCollisionProfileName("B
lockAll");
```

15. Lastly, the **BrickMesh** component will serve as the root component of the **Brick** actor. Add the following code to do this:

```
RootComponent = BrickMesh;
```

16. Now, add the following code to the constructor function to initialize our **BrickCollision UBoxComponent**:

```
BrickCollision =
CreateDefaultSubobject<UBox
Component>
(TEXT("BrickCollision"));
```

17. Just like the **BrickMesh** component,

the **BrickCollision** component will also need to have its collision set to "**BlockAll**" in order to receive the **OnHit()** callback events we will be adding later in this exercise. Add the following code:

```
BrickCollision->SetCollisionProfileName("BlockAll");
```

18. Next, the **BrickCollision** component needs to be attached to the **BrickMesh** component. We can do this by adding the following code:

```
BrickCollision->AttachToComponent(RootComponent,  
FAttachmentTransformRules::  
KeepWorldTransform);
```

19. Before we can finish the initialization of the **BrickCollision** component, we need to add the function definition for the **OnHit()** function. Add the following definition to the source file:

```
void  
ASuperSideScroller_Brick::OnHit(UPrimitiveComponent*
```

```
    HitComp, AActor*
    OtherActor,
    UPrimitiveComponent*
    OtherComp, FVector
    NormalImpulse, const
    FHitResult& Hit)

{



}
```

20. Now that we have the **OnHit()** function defined, we can assign the **OnComponentHit** callback to the **BrickCollision** component. Add the following code to the constructor function:

```
BrickCollision-
>OnComponentHit.AddDynamic(
this,
&ASuperSideScroller_Brick::
OnHit);
```

21. Compile the C++ code for the **SuperSideScroller_Brick** class and return to the Unreal Engine 4 editor.
22. In the **Content Browser** window, *right-click* on the **Content** folder and

select the **New Folder** option. Name this folder **Brick**.

23. *Right-click* inside the **Brick** folder and select **Blueprint Class**. From the **All Classes** search bar in the **Pick Parent Class** dialog window, search for and select the **SuperSideScroller_Brick** class.
24. Name this new Blueprint **BP_Brick**, and then *double-left-click* the asset to open it.
25. Select the **BrickMesh** component from the **Components** tab and set its **Static Mesh** parameter to the **Shape_Cube** mesh.
26. With the **BrickMesh** component still selected, set the **Element 0** material parameter to **M_Brick_Clay_Beveled**.
M_Brick_Clay_Beveled is a material provided by Epic Games by default when creating a new project. It can be found within the **StarterContent** directory, in the **Content Browser** window.

The last thing we need to do with the **BrickMesh** component is to adjust its scale so that it fits the needs of the player character, as well as the platforming mechanics of the **SuperSideScroller** game project.

27. With the **BrickMesh** component selected, make the following change to its **Scale** parameter:

(X=0 . 750000 , Y=0 . 750000 , Z=0 . 750000)

Now that the **BrickMesh** component is **75%** of its normal size, the **Brick** actor will become more manageable for us as designers when we place the actor into the game world, as well as when we're developing interesting platforming sections within the level.

The final step here is to update the location of the **BrickCollision** component so that it only has some of its collision sticking out from the bottom of the **BrickMesh** component.

28. Select the **BrickCollision** component from the **Components** tab

and update its **Location** parameter to the following values:

(X=0.000000, Y=0.000000, Z=30
.000000)

The **BrickCollision** component should now be positioned as follows:

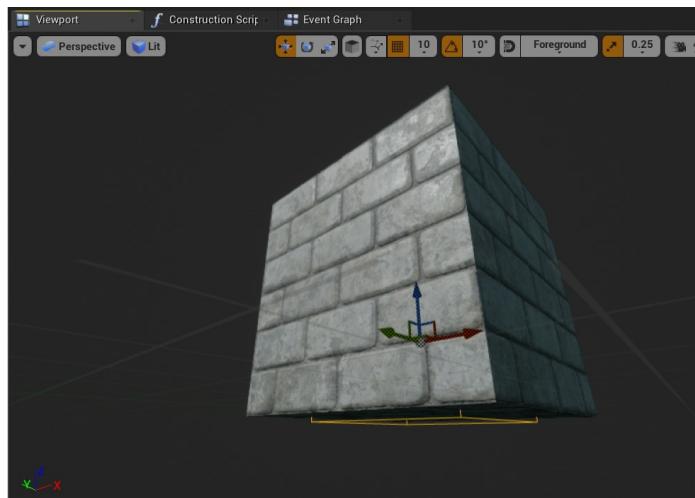


Figure 15.29: Now, the **BrickCollision** component is just barely outside the **BrickMesh** component

We are making this adjustment to the position of the **BrickCollision** component so that the player can only hit **UBoxComponent** when jumping underneath the brick. By making it slightly outside of the **BrickMesh** component, we can control this better and ensure that this component cannot be hit by the player in any other way.

Note

You can find the assets and code for this exercise here:
<https://packt.live/3kr7rh6>.

With this exercise complete, you were able to create the base framework for the **SuperSideScroller_Brick** class and put together the Blueprint actor to represent the brick in the game world. By adding a cube mesh and brick material, you added a nice visual polish to the brick. In the following exercise, you will add the remaining C++ logic to the brick. This will allow the player to destroy the brick and obtain a collectible.

Exercise 15.08: Adding the Brick Class C++ Logic

In the previous exercise, you created the base framework for the **SuperSideScroller_Brick** class by adding the necessary components and creating the **BP_Brick** Blueprint actor. In this exercise, you will add on top of the C++ code of *Exercise 15.07, Creating the Brick Class*, to grant logic to the **Brick** class. This will allow the brick to give players coin collectibles. Perform the following steps to accomplish this:

1. To begin, we need to create a function that will add the collectible to the player. Add the following function declaration to the **SuperSideScroller_Brick.h** header file, under our **Private Access Modifier**:

```
void AddCollectable(class ASuperSideScroller_Player* Player);
```

We want to pass in a reference to the **SuperSideScroller_Player** class so that we can call the **IncrementNumberofCollectable** **s()** function from that class.

2. Next, create a void function declaration called **PlayHitSound()** under our **Private Access Modifier**:

```
void PlayHitSound();
```

The **PlayHitSound()** function will be responsible for spawning the **HitSound** property you created in *Exercise 15.07, Creating the Brick Class*.

3. Finally, create another void function declaration called **PlayHitExplosion()** under our **Private Access Modifier**:

```
void PlayHitExplosion();
```

The **PlayHitExplosion()** function will be responsible for spawning the **Explosion** property you created in *Exercise 15.07, Creating the Brick Class*.

With the remaining functions needed for the **SuperSideScroller_Brick** class declared in the header file, let's move on and define these functions inside the source file.

4. At the top of the **SuperSideScroller_Brick.cpp** source file, add the following **#includes** to the list of **include** directories that already exist for this class:

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

The includes for the **World** and **GameplayStatics** classes are necessary to spawn both the **HitSound** and the **Explosion** effects for the brick. Including the **SuperSideScroller_Player** class is required to make the call to the **IncrementNumberofCollectable** **s()** class function.

5. Let's start with the function definition for the **AddCollectable()** function.

Add the following code:

```
void
ASuperSideScroller_Brick::AddCollectable(class
ASuperSideScroller_Player*
Player)

{
```

```
}
```

6. Now, make the call to the **IncrementNumberofCollectable** **s()** function by using the **Player** function input parameter:

```
Player-
>IncrementNumberofCollectab
les(CollectableValue);
```

7. For the **PlayHitSound()** function, you will need to get a reference to the **UWorld*** object and verify whether the **HitSound** property is valid before making the function call to **SpawnSoundAtLocation** from the **UGameplayStatics** class. This is a process you have done many times, so

this is the entire function code:

```
void
ASuperSideScroller_Brick::PlayHitSound()
{
    UWorld* World =
GetWorld();

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

8. Just like the **PlayHitSound()** function, the **PlayHitExplosion()** function will work in an almost similar way, and it's a process you have done many times in this project. Add the

following code to create the function definition:

```
void
ASuperSideScroller_Brick::P
layHitExplosion( )

{
    UWorld* World =
GetWorld();

    if (World)

    {
        if (Explosion)

        {
            UGameplayStatics::Spa
wnEmitterAtLocation(World,
Explosion,
GetActorTransform());
        }
    }
}
```

With these functions defined, let's update the **OnHit()** function so that if the player does hit the **BrickCollision** component, we can

spawn **HitSound** and **Explosion**,
and also add a coin collectible to the
player's collection.

9. First, in the **OnHit()** function, create a new variable called **Player** of the **ASuperSideScroller_Player** type that equals the **Cast** of the **OtherActor** input parameter of the function, as shown here:

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

10. Next, we only want to continue with this function if **Player** is valid and **bHasCollectable** is **True**. Add the following **if()** statement:

```
if (Player &&  
bHasCollectable)  
{  
}
```

11. If the conditions in the **if()** statement are met, that is when we need to make the calls to the **AddCollectable()**,

PlayHitSound(), and
PlayHitExplosion() functions.

Make sure to also pass in the **Player** variable inside the **AddCollectable()** function:

```
AddCollectable(Player);
```

```
PlayHitSound();
```

```
PlayHitExplosion();
```

12. Finally, add the function call to destroy the brick inside of the **if()** statement:

```
Destroy();
```

13. With the **OnHit()** function defined as we need, recompile the C++ code but do not return to the Unreal Engine 4 editor just yet.
14. For the VFX and SFX of the brick's explosion, we will need to migrate assets from two separate projects available to us from **Epic Games Launcher**: the **Blueprints** project and the **Content Examples** project.
15. Using your knowledge from previous exercises, download and install these projects using engine version 4.24.

Both projects can be found in the **Learn** tab, under the **Engine Feature Samples** category.

16. Once installed, open the **Content Examples** project and find the **P_Pixel_Explosion** asset in the **Content Browser** window.
17. *Right-click* this asset, then select **Asset Actions** and then **Migrate**. Migrate this asset and all its referenced assets into your **SuperSideScroller** project.
18. Once this asset has been successfully migrated, close the **Content Examples** project and open the **Blueprints** project.
19. From the **Content Browser** window of the **Blueprints** project, find the **Blueprints_TextPop01** asset.
20. *Right-click* this asset, then select **Asset Actions**, and then **Migrate**. Migrate this asset and all its referenced assets into your **SuperSideScroller** project.

With these assets migrated to your project, return to the Unreal Engine 4 editor of your **SuperSideScroller** project.

21. Navigate to the **Brick** folder in the **Content Browser** window and *double-left-click* the **BP_Brick** asset to open it.
22. In the **Details** panel of the actor, find the **Super Side Scroller Brick** section and set the **HitSound** parameter to the **Blueprints_TextPop01** soundwave you imported.
23. Next, add the **P_Pixel_Explosion** particle you imported into the **Explosion** parameter.
24. Recompile the **BP_Brick** Blueprint and add two of these actors to your level.
25. Set one of the bricks so that the **bHasCollectable** parameter is **True**; set the other to **False**. Please refer to the following screenshot:



Figure 15.30: This Brick actor is set to have a collectible spawn

26. Using PIE, observe the differences in behavior between the two brick actors when you attempt to hit the bottom of the brick with the character's head when jumping, as shown in the following screenshot:



Figure 15.31: Now, the player can hit the brick and it will be destroyed

When **bHasCollectable** is **True**, **SuperSideScroller_Brick** will play our **HitSound**, spawn the **Explosion** particle system, add a coin collectible to the player, and be destroyed.

Note

You can find the assets and code for this exercise here:
<https://packt.live/3pjhoAv>.

With this exercise complete, you have now finished developing the gameplay mechanics for the

SuperSideScroller game project. Now, the **SuperSideScroller_Brick** class can be used for both the platforming gameplay and the coin collecting mechanic that we want for the game.

Now that the brick can be destroyed and hidden coins can be collected, all the gameplay elements that we set out to create for the **SuperSideScroller** game project are complete.

Summary

In this chapter, you put your knowledge to the test to create the remaining gameplay mechanics for the **SuperSideScroller** game project. Using a combination of C++ and Blueprints, you developed the potion power-up and coins for the player to collect in the level. Also, by using your knowledge from *Chapter 14, Spawning the Player Projectile*, you added unique audio and visual assets to these collectible items to add a nice layer of polish to the game.

You learned and took advantage of the **UMG UI** system within Unreal Engine 4 to create a simple, yet effective, UI feedback system to display the number of coins that the player has collected. By using the binding feature of the **Text** widget, you were able to keep the UI updated with the number of coins the player has currently collected. Lastly, you created a **Brick** class using the knowledge you learned from the **SuperSideScroller** project to hide coins for the player so that they can collect and find them.

The **SuperSideScroller** project has been an extensive project that expanded over many of the tools and practices available within Unreal Engine 4. In *Chapter 10, Creating a SuperSideScroller Game*, we imported custom skeleton and animation assets to use in developing the Animation Blueprint of the player character. In *Chapter 11, Blend Spaces 1D, Key*

Bindings, and State Machines, we used **Blend spaces** to allow the player character to blend between idle, walking, and sprinting animations, while also using an **Animation State Machine** to handle the jumping and movement states of the player character. We then learned how to control the player's movement and jump height using the character movement component.

In *Chapter 12, Animation Blending and Montages*, we learned more about animation blending inside **Animation Blueprints** by using the **Layered Blend per Bone** function and **Saved Cached Poses**. By adding a new **AnimSlot** for the upper body animation of the player character's throw animation, we were able to have both the player movement animations and the throw animation blend together smoothly. In *Chapter 13, Enemy Artificial Intelligence*, we used the robust systems of Behavior Trees and Blackboards to develop AI behavior for the enemy. We created our own **Task** that will allow the enemy AI to move in-between points from a custom Blueprint that we also developed to determine patrol points for the AI.

In *Chapter 14, Spawning the Player Projectile*, we learned how to create an **Anim Notify** and how to implement this notify in our **Animation Montage** for the player character's throw to spawn the player projectile. Then, we learned about how to create projectiles and how to use **Projectile Movement Component** to have the player projectile move in the game world.

Finally, in this chapter, we learned how to create UI using the **UMG** toolset for the coin collectible, as well as how to manipulate our **Character Movement Component** to create the potion power-up for the player. Lastly, you created a **Brick** class that can be used to hide coins for the player to find and collect.

This summarization only really scratches the surface of what

we learned and accomplished in the **SuperSideScroller** project. Before you move on, here are some challenges for you to test your knowledge and expand upon the project:

1. Add a new power-up that lowers the gravity that's applied to the player character. Import a custom mesh and audio assets to give this power-up a unique look compared to the potion power-up you made.
2. When the player character collects 10 coins, grant the player a power-up.
3. Implement the functionality that allows the player to be destroyed when it's overlapping with the AI. Include being able to respawn the player when this happens.
4. Add another power-up that gives immunity to the player so that they cannot be destroyed when they're overlapping with an enemy. (In fact, when overlapping an enemy with this power-up, it could destroy the enemy.)
5. Using all the gameplay elements you've developed for the **SuperSideScroller** project, create a new level that takes advantage of

these elements to make an interesting platforming arena to play in.

6. Add multiple enemies with interesting patrol points to challenge the player when they're navigating the area.
7. Place power-ups in hard-to-reach areas so that players need to improve their platforming skills to obtain them.
8. Create dangerous pitfalls for the player to navigate across and add functionality that will destroy the player if they fall off the map.

In the next chapter, you will learn about the basics of multiplayer, server-client architectures, and the gameplay framework classes used for multiplayer inside Unreal Engine 4. You will use this knowledge to expand upon the multiplayer FPS project in Unreal Engine 4.

16. Multiplayer Basics

Overview

In this chapter, you will be introduced to some important multiplayer concepts in order to add multiplayer support to your game using Unreal Engine 4's network framework.

*By the end of this chapter, you'll know basic multiplayer concepts such as the server-client architecture, connections, and actor ownership, along with roles and variable replication. You'll be able to implement these concepts to create a multiplayer game of your own. You'll also be able to make a 2D Blend Space, which allows you to blend between animations laid out in a 2D grid. Finally, you'll learn how to use **Transform (Modify) Bone** nodes to control Skeletal Mesh bones at runtime.*

Introduction

In the previous chapter, we completed the **SuperSideScroller** game and used 1D Blend Spaces, animation blueprints, and animation montages. In this chapter, we're going to build on that knowledge and learn how to add multiplayer functionality to a game using Unreal Engine.

Multiplayer games have grown quite a lot in the last decade. Games such as Fortnite, PUBG, League of Legends, Rocket League, Overwatch, and CS: GO have gained a lot of popularity in the gaming community and have had great

success. Nowadays, almost all games need to have some kind of multiplayer experience in order to be more relevant and successful.

The reason for that is it adds a new layer of possibilities on top of the existing gameplay, such as being able to play with friends in cooperative mode (*also known as co-op mode*) or against people from all around the world, which greatly increases the longevity and value of a game.

In the next topic, we will be discussing the basics of multiplayer.

Multiplayer Basics

You may have heard the term multiplayer a lot while gaming, but what does it mean for game developers? Multiplayer, in reality, is just a set of instructions sent through the network (*internet or local area network*) between the server and its connected clients in order to give players the illusion of a shared world.

For this to work, the server needs to be able to talk to clients, but also the other way around (client to server). This is because clients are typically the ones that affect the game world, so they need a way to be able to inform the server of their intentions while playing the game.

An example of this back and forth communication between the server and a client is when a player tries to fire a weapon during a game. Have a look at the following figure, which shows a client-server interaction:

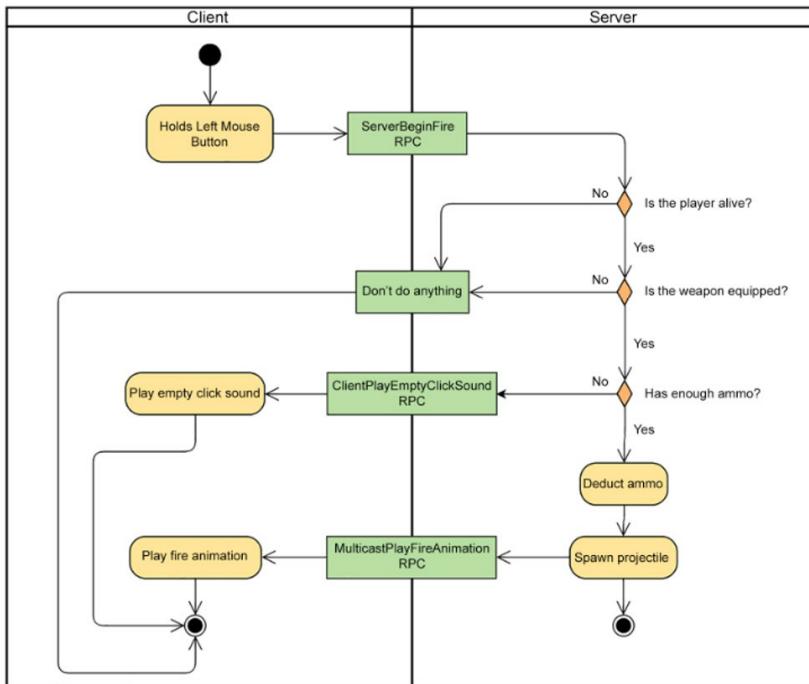


Figure 16.1: Client-server interaction when a player wants to fire a weapon in a multiplayer game

Let's examine what was shown in *Figure 16.1*:

1. The player holds the *left mouse button* down and the client of that player tells the server that it wants to fire a weapon.
 2. The server validates whether the player can fire the weapon by checking the following:
 1. If the player is alive
 2. If the player has a weapon equipped
 3. If the player has enough

ammo

3. If all of the validations are valid, then the server will do the following:
 1. Run the logic to deduct ammo
 2. Spawn the projectile actor on the server, which is automatically sent to all of the clients
 3. Play the fire animation on the character instance in all of the clients to ensure certain synchronicity between all of them, which helps to sell the idea that it's the same world, even though it's not
4. If any of the validations fail, then the server tells the specific client what to do:
 1. Player is dead – don't do anything
 2. Player doesn't have a weapon equipped – don't do anything
 3. Player doesn't have enough

ammo – play an empty click sound

Remember, if you want your game to support multiplayer, then it's highly recommended that you do that as soon as possible in your development cycle. If you try to run a single-player project with multiplayer enabled, you'll notice that some functionalities might *just work*, but probably most of them won't be working properly or as expected.

The reason for that is when you execute the game in single-player, the code runs locally and instantly, but when you add multiplayer into the equation, you are adding external factors such as an authoritative server that talks to clients on a network with latency, as you saw in *Figure 16.1*.

In order to get everything working properly, you need to break apart the existing code into the following:

- Code that only runs on the server
- Code that only runs on the client
- Code that runs on both, which can take a lot of time depending on the complexity of your single-player game

In order to add multiplayer support to games, Unreal Engine 4 comes with a very powerful and bandwidth-efficient network framework already built in, using an authoritative server-client architecture.

Here is a diagram of how it works:

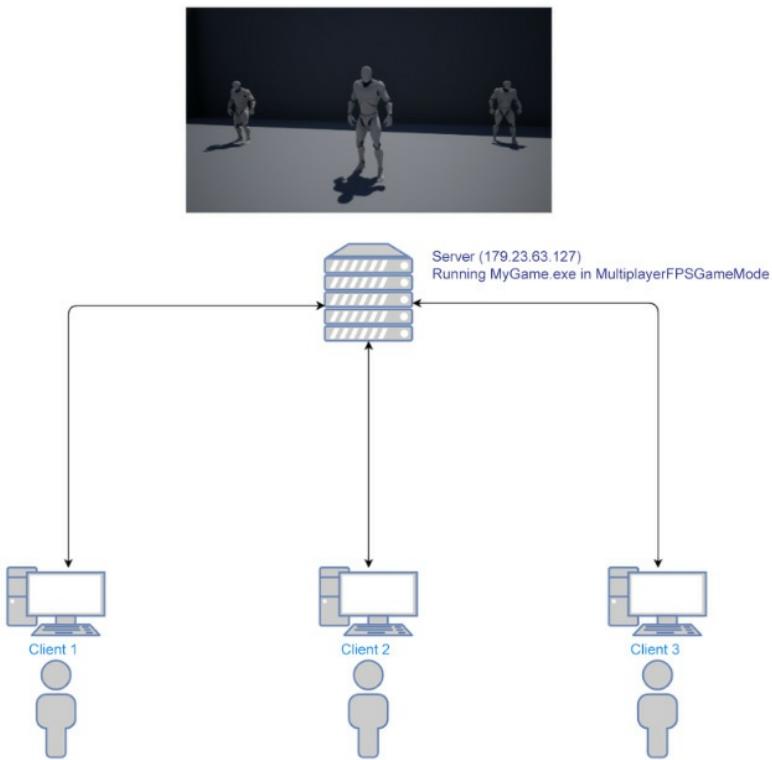


Figure 16.2: Server-client architecture in Unreal Engine 4

In *Figure 16.2*, you can see how the server-client architecture works in Unreal Engine 4. Each player controls a client that communicates with the server using a **two-way connection**. The server runs a specific level with a game mode (*which only exists in the server*) and controls the flow of information so that the clients can see and interact with each other in the game world.

Note

Multiplayer can be a very advanced topic, so these next few chapters will serve as an introduction to help you understand the essentials, but it will not be an in-depth look. For that reason, some concepts might be omitted for simplicity's sake.

In the next section, we will be looking at servers.

Servers

The server is the most critical part of the architecture since it's responsible for handling most of the work and making important decisions.

Here is an overview of the main responsibilities of a server:

1. Creating and managing the shared world instance:

The server runs its own instance of the game in a specific level and game mode (*this will be covered in the chapters ahead*) and that will serve as the shared world between all of the connected clients.

The level being used can be changed at any point in time and, if applicable, the server can bring along all of the connected clients with it automatically.

2. Handling client join and leave requests:

If a client wants to connect to a server, it needs to ask for permission. To do this, the client sends a join request to the server, through a direct IP connection (*explained in the next section*) or an online subsystem such as Steam. Once the join request reaches the server, it will perform some validations to determine whether the

request is accepted or rejected.

However, you should know that there are a few reasons why the server would want to reject a request to join a game. The most common ones are if the server is already at full capacity and can't take any more clients or if the client is using an out-of-date version of the game. If the server accepts the request, then a player controller with a connection is assigned to the client and the **PostLogin** function in the game mode is called. From that point on, the client will enter the game and is part of the shared world where the player will be able to see and interact with other clients. If a client disconnects at any point in time, then all of the other clients will be notified and the **Logout** function in the game mode will be called.

3. **Spawning the actors that all of the clients need to know about:** If you want to spawn an actor that exists in all of the clients, then you need to do that on the server. The reason for this is the server has the authority and is

the only one that can tell each client to create its own instance of that actor.

This is the most common way of spawning actors in multiplayer, since most actors need to exist in all of the clients. An example of this would be with a power-up, which is something that all clients can see and interact with.

4. **Running critical gameplay logic:**

In order to make sure that the game is fair to all of the clients, the critical gameplay logic needs to be executed only on the server. If clients were responsible for handling the deduction of health, it would be very exploitable, because a player could use a tool to change in memory the current value of health to 100% all the time, so the player would never die in the game.

5. **Handling variable replication:** If you have a replicated variable (*covered in this chapter*), then its value should only be changed on the server. This will ensure that all of the clients will have the value updated automatically. You can still change the value on the

client, but it will always be replaced with the latest value from the server, in order to prevent cheating and to make sure all of the clients are in sync.

6. Handling RPCs from the client:

The server needs to process the remote procedure calls (*Chapter 17, Remote Procedure Calls*) sent from the clients.

Now that you know what a server does, we can talk about the two different ways of creating a server in Unreal Engine 4.

Dedicated Server

The dedicated server only runs the server logic, so you won't see the typical window with the game running where you control a character as a local player. Additionally, if you run the dedicated server with the **-log** command prompt, you'll have a console window that logs relevant information about what is happening on the server, such as if a client has connected or disconnected, and so on. You, as a developer, can also log your own information by using the **UE_LOG** macro.

Using dedicated servers is a very common way of creating servers for multiplayer games, and since it's more lightweight than a listen server (*covered in the next section*), you could just host it on a server stack and leave it running.

To start a dedicated server in Unreal Engine 4, you can use the following command arguments:

- Run the following command to start a dedicated server inside an editor through a shortcut or Command Prompt:

```
<UE4 Install  
Folder>\Engine\Binaries\Win  
64\UE4Editor.exe <UProject  
Location> <Map Name> -  
server -game -log
```

Here's an example:

```
C:\Program Files\Epic  
Games\UE_4.24\Engine\Binari  
es\Win64\UE4Editor.exe  
D:\TestProject\TestProject.  
uproject TestMap -server -  
game -log
```

- A packaged project requires a special build of the project built specifically to serve as a dedicated server.

Note

*You can find out more about setting up a packaged dedicated server by visiting
<https://allarsblog.com/2015/11/06/support-dedicated-servers/> and*

[https://www.ue4community.wiki/Dedicated_Server_Guide_\(Windows\)](https://www.ue4community.wiki/Dedicated_Server_Guide_(Windows)).

The Listen Server

The listen server acts as a server and client at the same time, so you'll also have a window where you can play the game as a client with this server type. It also has the advantage of being the quickest way of getting a server running, but it's not as lightweight as a dedicated server, so the number of clients that can be connected at the same time will be limited.

To start a listen server, you can use the following command arguments:

- Run the following command to start a dedicated server inside an editor through a shortcut or Command Prompt:

```
<UE4 Install  
Folder>\Engine\Binaries\Win  
64\UE4Editor.exe <UProject  
Location> <Map Name>?Listen  
-game
```

Here's an example:

```
C:\Program Files\Epic  
Games\UE_4.24\Engine\Binari  
es\Win64\UE4Editor.exe  
D:\TestProject\TestProject.
```

```
uproject TestMap?Listen -  
game
```

- A packaged project (development builds only) requires a special build of the project built specifically to serve as a dedicated server:

```
<Project Name>.exe <Map  
Name>?Listen -game
```

Here's an example:

```
D:\Packaged\TestProject\Tes  
tProject.exe TestMap?Listen  
-game
```

In the next section, we will discuss clients.

Clients

The client is the simplest part of the architecture because most of the actors will have the authority on the server, so in those cases, the work will be done on the server and the client will just obey its orders.

Here is an overview of the main responsibilities of a client:

1. **Enforcing variable replication from the server:** The server typically has authority over all of the actors that the client knows, so when the value of a

replicated variable is changed on the server, the client needs to enforce that value as well.

2. Handling RPCs from the server:

The client needs to process the remote procedure calls (covered in *Chapter 17, Remote Procedure Calls*) sent from the server.

3. Predicting movement when simulating:

When a client is simulating an actor (*covered later in this chapter*) it needs to locally predict where it's going to be based on the actor's velocity.

4. Spawning the actors that only a client needs to know about: If you want to spawn an actor that only exists on a client, then you need to do that on that specific client.

This is the least common way of spawning actors since there are few instances when you want an actor to only exist on a client. An example of this is the placement preview actor you see in multiplayer survival games, where the player controls a semi-

transparent version of a wall that other players can't see until it's actually placed.

A client can join a server in different ways. Here is a list of the most common methods:

- Using the Unreal Engine 4 console (by default is the ` key) to open it and type:

Open <Server IP Address>

For instance:

Open 194.56.23.4

- Using the **Execute Console Command** Blueprint node. An example is as follows:



Figure 16.3: Joining a server with an example IP with the Execute Console Command node

- Using the **ConsoleCommand** function in **APlayerController** as follows:

```
PlayerController->ConsoleCommand( "Open
```

```
<Server IP Address>");
```

Here's an example:

```
PlayerController-
>ConsoleCommand("Open
194.56.23.4");
```

- Using the editor executable through a shortcut or Command Prompt:

```
<UE4 Install
Folder>\Engine\Binaries\Win
64\UE4Editor.exe <UProject
Location> <Server IP
Address> -game
```

Here's an example:

```
C:\Program Files\Epic
Games\UE_4.24\Engine\Binaries
\Win64\UE4Editor.exe
D:\TestProject\TestProject.u
project 194.56.23.4 -game
```

- Using a packaged development build through a shortcut or Command Prompt:

```
<Project Name>.exe <Server
IP Address>
```

Here's an example:

**D:\Packaged\TestProject\Test
Project.exe 194.56.23.4**

In the following exercise, we will test the Third Person template that comes with Unreal Engine 4 in multiplayer.

Exercise 16.01: Testing the Third Person Template in Multiplayer

In this exercise, we're going to create a Third Person template project and play it in multiplayer.

The following steps will help you complete the exercise.

1. Create a new **Third Person** template project using **Blueprints** called **TestMultiplayer** and save it to a location of your choosing.

Once the project has been created, it should open the editor. We'll now test the project in multiplayer to see how it behaves:

2. In the editor, to the right of the **Play** button, you have an option with an arrow pointing down. Click on it and you should see a list of options. Under

the **Multiplayer Options** section, you can configure how many clients you want to use and whether or not you want a dedicated server.

3. Leave **Run Dedicated Server** unchecked, change **Number of Players** to **3**, and click on **New Editor Window (PIE)**.
4. You should see three windows on top of each other representing the three clients:

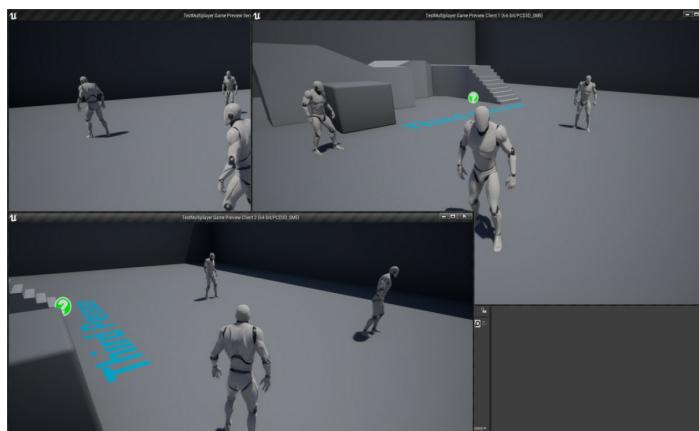


Figure 16.4: Launching three client windows with a listen server

As you can see, this is a bit cluttered, so let's change the size of the windows. Press *Esc* on your keyboard to stop playing.

5. Click once again on the downward-pointing arrow next to the **Play** button and pick the last option, **Advanced Settings**.
6. Search for the **Game Viewport Settings** section. Change **New Viewport Resolution** to **640x480** and close the **Editor Preferences** tab.
7. Play the game again and you should see the following:

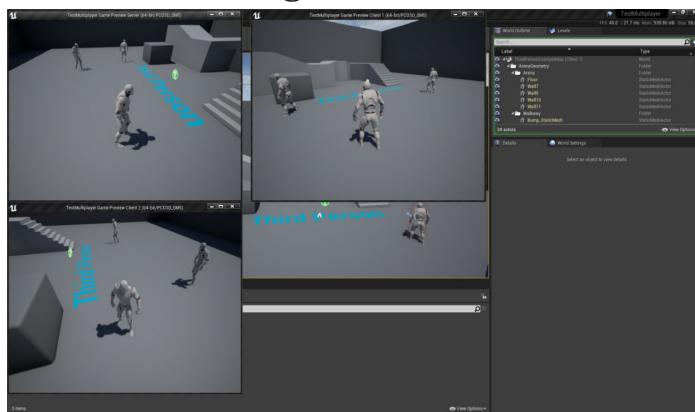


Figure 16.5: Launching three client windows using a 640x480 resolution with a listen server

Once you start playing, you'll notice that the title bars of the windows say **Server**, **Client 1**, and **Client 2**. Since you can control a character in the **Server** window, that means we're running a **listen server**, where you have the server and a client running in the same window. When that happens, you should interpret the window title as **Server + Client 0** instead of just **Server** to avoid confusion.

By completing this exercise, you now have a setup where you'll have one server and three clients running (**Client 0**, **Client 1**, and **Client 2**).

Note

When you have multiple windows running at the same time, you'll notice that you can only have input focus on one window at a time. To shift the focus to another window, just press Shift + F1 to lose the current input focus and then just click on the new window you want to focus on.

If you play the game in one of the windows, you'll notice that you can move around and jump and the other clients will also be able to view that.

The reason why everything works is that the character movement component, which comes with the character class, replicates the location, rotation, and falling state (used to show whether you are jumping or not) for you automatically. If you want to add a custom behavior such as an attack animation, you can't just tell the client to play an animation locally when a key is pressed, because that will not work on the other clients. That's why you need the server, to serve as an intermediary and tell all the clients to play the animation when one client presses the key.

The Packaged Version

Once you've finished the project, it's good practice to package it (*as covered in the previous chapters*) so that we have a pure standalone version that doesn't use Unreal Engine editor, which will run faster and is more lightweight.

The following steps will help you create the packaged version of *Exercise 16.01, Testing the Third Person Template in*

Multiplayer file:

1. Go to **File -> Package Project -> Windows -> Windows (64-bit)**.
2. Pick a folder to place the packaged build and wait for it to finish.
3. Go to the selected folder and open the **WindowsNoEditor** folder inside it.
4. *Right-click* on **TestMultiplayer.exe** and pick **Create Shortcut**.
5. Rename the new shortcut **Run Server**.
6. *Right-click* on it and pick **Properties**.
7. On the Target, append **ThirdPersonExampleMap?Listen -server**, which creates a listen server using **ThirdPersonExampleMap**.
You should end up with this:

```
"<Path>\WindowsNoEditor\Test  
Multiplayer.exe"  
ThirdPersonExampleMap?
```

Listen -server

8. Click **OK** and run the shortcut.
9. You should get a Windows Firewall prompt, so allow it.
10. Leave the server running and go back to the folder and create another shortcut from **TestMultiplayer.exe**.
11. Rename it **Run Client**.
12. *Right-click* on it and pick **Properties**.
13. On the Target, append **127.0.0.1**, which is the IP of your local server. You should end up with "
<Path>\WindowsNoEditor\TestMultiplayer.exe" 127.0.0.1".
14. Click **OK** and run the shortcut.
15. You are now connected to the listen server, so you can see each other's characters.
16. Every time you click on the **Run Client** shortcut, you'll add a new

client to the server, so you can have a few clients running on the same machine.

In the next section, we will be looking at connections and ownership.

Connections and Ownership

When using multiplayer in Unreal Engine, an important concept to understand is that of a connection. When a client joins a server, it will get a new **Player Controller** with a connection associated with it.

If an actor doesn't have a valid connection with the server, then the actor won't be able to do replication operations such as variable replication (*covered later in this chapter*) or call RPCs (covered in *Chapter 17, Remote Procedure Calls*).

If the Player Controller is the only actor that holds a connection, then does that mean that it's the only place you can do replication operations? No, and that's where the **GetNetConnection** function, defined in **AActor**, comes into play.

When doing replication operations (such as variable replication or call RPCs) on an actor, the Unreal framework will get the actor's connection by calling the **GetNetConnection()** function on it. If the connection is valid, then the replication operation will be processed, if it's not, nothing will happen. The most common implementations of **GetNetConnection()** are from **APawn** and **AActor**.

Let's take a look at how the **APawn** class implements the **GetNetConnection()** function, which is typically used for characters:

```
class UNetConnection*
APawn::GetNetConnection() const

{
    // if have a controller, it has the net
    connection

    if ( Controller )
    {
        return Controller->GetNetConnection();
    }

    return Super::GetNetConnection();
}
```

The preceding implementation, which is part of the Unreal Engine 4 source code, will first check whether the pawn has a valid controller. If the controller is valid, then it will use its connection. If the controller is not valid, then it will use the parent implementation of the **GetNetConnection()** function, which is on **AActor**:

```
UNetConnection* AActor::GetNetConnection()
const

{
    return Owner ? Owner->GetNetConnection()
    : nullptr;
}
```

The preceding implementation, which is also part of the Unreal Engine 4 source code, will check whether the actor has a valid owner. If it does, it will use the owner's connection; if it doesn't, it will return an invalid connection. So what is this

Owner variable? Every actor has a variable called **Owner** (where you can set its value by calling the **SetOwner** function) that shows which actor *owns* it, so you can think of it as the parent actor.

Using the owner's connection in this implementation of **GetNetConnection()** will work like a hierarchy. If, while going up the hierarchy of owners, it finds an owner that is a Player Controller or is being controlled by one, then it will have a valid connection and will be able to process replication operations. Have a look at the following example.

Note

In a listen server, the connection for the character controlled by its client will always be invalid, because that client is already a part of the server and therefore doesn't need a connection.

Imagine a weapon actor was placed in the world and it's just sitting there. In that situation, the weapon won't have an owner, so if the weapon tries to do any replication operations, such as variable replication or calling RPCs, nothing will happen.

However, if a client picks up the weapon and calls **SetOwner** on the server with the value of the character, then the weapon will now have a valid connection. The reason for this is because the weapon is an actor, so in order to get its connection, it will use the **AActor** implementation of **GetNetConnection()**, which returns the connection of its owner. Since the owner is the client's character, it will use the implementation of **GetNetConnection()** of **APawn**. The character has a valid Player Controller, so that is the connection returned by the function.

Here is a diagram to help you understand this logic:

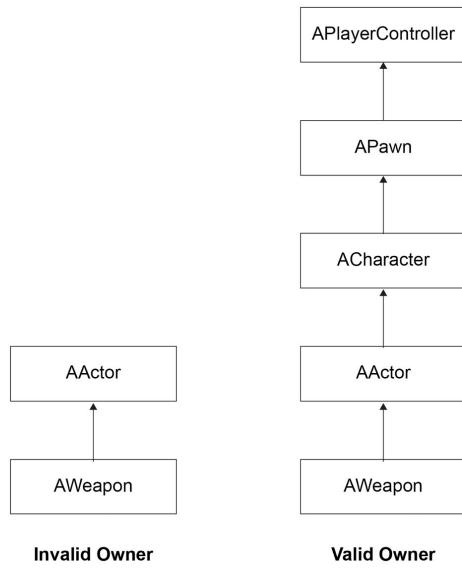


Figure 16.6: Connections and ownership example of a weapon actor

Let's understand the elements of an invalid owner:

- **AWeapon** doesn't override the **GetNetConnection** function, so to get the weapon's connection, it will call the first implementation found, which is **AActor::GetNetConnection**.
- The implementation of **AActor::GetNetConnection** calls **GetNetConnection** on its owner. Since there is no owner, the connection is invalid.

A valid owner will consist of the following elements:

- **AWeapon** doesn't override the

GetNetConnection function, so to get its connection, it will call the first implementation found, which is **AActor::GetNetConnection**.

- The implementation of **AActor::GetNetConnection** calls **GetNetConnection** on its owner. Since the owner is the character that picked up the weapon, it will call **GetNetConnection** on it.
- **ACharacter** doesn't override the **GetNetConnection** function, so to get its connection, it will call the first implementation found, which is **APawn::GetNetConnection**.
- The implementation of **APawn::GetNetConnection** uses the connection from the owning player controller. Since the owning player controller is valid, then it will use that connection for the weapon.

Note

*In order for **SetOwner** to work as intended, it needs to be executed on the authority which, in most cases, means*

*the server. If you only execute **SetOwner** on the client, it still won't be able to execute replication operations.*

Roles

When you spawn an actor on the server, there will be a version of the actor created on the server and one on each client. Since there are different versions of the same actor on different instances of the game (**Server**, **Client 1**, **Client 2**, and so on), it is important to know which version of the actor is which. This will allow us to know what logic can be executed in each of these instances.

To help with this situation, every actor has the following two variables:

- **Local Role:** The role that the actor has on the current game instance. For example, if the actor was spawned on the server and the current game instance is also the server, then that version of the actor has authority, so you can run more critical gameplay logic on it. It's accessed by calling the **GetLocalRole()** function.
- **Remote Role:** The role that the actor has on the remote game instance. For example, if the current game instance

is the server, then it returns the role the actor has on clients and vice versa. It's accessed by calling the **GetRemoteRole()** function.

The return type of the **GetLocalRole()** and **GetRemoteRole()** functions is **ENetRole**, which is an enumeration that can have the following possible values:

- **ROLE_None**: The actor doesn't have a role because it's not being replicated.
- **ROLE_SimulatedProxy**: The current game instance doesn't have authority over the actor and it's not controlling it through a Player Controller. That means that its movement will be simulated/predicted by using the last value of the actor's velocity.
- **ROLE_AutonomousProxy**: The current game instance doesn't have authority over the actor, but it's being controlled by a Player Controller. That means that we can send to the server more accurate movement information, based on the player's inputs, instead of just using the last value of the actor's velocity.

- **ROLE_Authority:** The current game instance has complete authority over the actor. That means that if the actor is on the server, the changes made to the replicated variables of the actor will be treated as the value that every client needs to have enforced through variable replication.

Let's have a look at the following example code snippet:

```
ENetRole MyLocalRole = GetLocalRole();
ENetRole MyRemoteRole = GetRemoteRole();
FString String;
if(MyLocalRole == ROLE_Authority)
{
    if(MyRemoteRole == ROLE_AutonomousProxy)
    {
        String = «This version of the actor is
the authority and
its being controlled by a player on
its client»;
    }
    else if(MyRemoteRole ==
ROLE_SimulatedProxy)
    {
        String = «This version of the actor is
the authority but
```

```

    it's not being controlled by a player
on its client»;

}

}

else String = "This version of the actor
isn't the authority";

GEngine->AddOnScreenDebugMessage(-1, 0.0f,
FColor::Red, String);

```

The preceding code snippet will store the values of the local role and remote role to **MyLocalRole** and **MyRemoteRole** respectively. After that, it will print different messages on the screen depending on whether that version of the actor is the authority or whether it's being controlled by a player on its client.

Note

*It is important to understand that if an actor has a local role of **ROLE_Authority**, it doesn't mean that it's on the server; it means that it's on the game instance that originally spawned the actor and therefore has authority over it.*

*If a client spawns an actor, even though the server and the other clients won't know about it, its local role will still be **ROLE_Authority**. Most of the actors in a multiplayer game will be spawned by the server; that's why it's easy to misunderstand that the authority is always referring to the server.*

Here is a table to help you understand the roles that an actor will have in different scenarios:

Server		Client	
Local Role	Remote Role	Local Role	Remote Role
Actor spawned on the server	ROLE_Authority	ROLE_SimulatedProxy	ROLE_SimulatedProxy
Actor spawned on the client	Won't exist	Won't exist	ROLE_Authority
Player-owned pawn spawned on the server	ROLE_Authority	ROLE_AutonomousProxy	ROLE_AutonomousProxy
Player-owned pawn spawned on the client	Won't exist	Won't exist	ROLE_Authority

Figure 16.7: Roles that an actor can have in different scenarios

In the preceding table, you can see the roles that an actor will have in different scenarios.

Let's analyze each scenario and explain why the actor has that role:

Actor Spawned on the Server

The actor spawns on the server, so the server's version of that actor will have the local role of **ROLE_Authority** and the remote role of **ROLE_SimulatedProxy**, which is the local role of the client's version of the actor. For the client's version of that actor, its local role will be **ROLE_SimulatedProxy** and the remote role will be **ROLE_Authority**, which is the local role of the server's actor version.

Actor Spawned on the Client

The actor was spawned on the client, so the client's version of that actor will have the local role of **ROLE_Authority** and the remote role of **ROLE_SimulatedProxy**. Since the actor wasn't spawned on the server, then it will only exist on the client that spawned it, so there won't be versions of this actor on the server and the other clients.

Player-Owned Pawn Spawned on the Server

The pawn was spawned on the server, so the server's version of that pawn will have the local role of **ROLE_Authority** and

the remote role of **ROLE_AutonomousProxy**, which is the local role of the client's version of the pawn. For the client's version of that pawn, its local role will be **ROLE_AutonomousProxy**, because it's being controlled by **PlayerController**, and the remote role **ROLE_Authority**, which is the local role of the server's pawn version.

Player-Owned Pawn Spawned on the Client

The pawn was spawned on the client, so the client's version of that pawn will have the local role of **ROLE_Authority** and the remote role of **ROLE_SimulatedProxy**. Since the pawn wasn't spawned on the server, then it will only exist on the client that spawned it, so there won't be versions of this pawn on the server and the other clients.

Exercise 16.02: Implementing Ownership and Roles

In this exercise, we're going to create a C++ project that uses the Third Person template as a base.

Create a new actor called **OwnershipTestActor** that has a static mesh component as the root component, and on every tick, it'll do the following:

- On the authority, it will check which character is closest to it within a certain radius (configured by the **EditAnywhere** variable called

OwnershipRadius) and will set that character as its owner. When no character is within the radius, then the owner will be **nullptr**.

- Display its local role, remote role, owner, and connection.
- Edit **OwnershipRolesCharacter** and override the **Tick** function so that it displays its local role, remote role, owner, and connection.
- Create a new header file called **OwnershipRoles.h** that contains the **ROLE_TO_String** macro, which converts **ENetRole** into an **FString** variable.

The following steps will help you complete the exercise:

1. Create a new **Third Person** template project using **C++** called **OwnershipRoles** and save it to a location of your choosing.
2. Once the project has been created, it should open the editor as well as the Visual Studio solution.

3. Using the editor, create a new C++ class called **OwnershipTestActor** that derives from **Actor**.
4. Once it finishes compiling, Visual Studio should pop up with the newly created **.h** and **.cpp** files.
5. Close the editor and go back to Visual Studio.
6. In Visual Studio, open the **OwnershipRoles.h** file and add the following macro:

```
#define  
ROLE_TO_STRING(Value)  
FindObject<UEnum>  
(ANY_PACKAGE,  
TEXT("ENetRole"), true)-  
>GetNameStringByIndex((int3  
2)Value)
```

This macro will convert the **ENetRole** enumeration that we get from the **GetLocalRole()** function and **GetRemoteRole()** into an **FString**. The way it works is by finding the **ENetRole** enumeration type through Unreal Engine's reflection system and

from that, it converts the **Value** parameter into an **FString** variable so it can be printed on the screen.

7. Now, open the **OwnershipTestActor.h** file.
8. Declare the protected variables for the static mesh component and the ownership radius as shown in the following code snippet:

```
UPROPERTY(VisibleAnywhere,  
BlueprintReadOnly, Category  
= "Ownership Test Actor")  
  
UStaticMeshComponent* Mesh;  
  
UPROPERTY(EditAnywhere,  
BlueprintReadOnly, Category  
= "Ownership Test Actor")  
  
float OwnershipRadius =  
400.0f;
```

In the preceding code snippet, we declare the static mesh component and the **OwnershipRadius** variable, which allows you to configure the radius of the ownership.

9. Next, delete the declaration of

BeginPlay and move the constructor and the **Tick** function declarations to the protected area.

10. Now, open the **OwnershipTestActor.cpp** file and add the required header files as mentioned in the following code snippet:

```
#include  
"DrawDebugHelpers.h"  
  
#include "OwnershipRoles.h"  
  
#include  
"OwnershipRolesCharacter.h"  
  
#include  
"Components/StaticMeshCompo  
nent.h"  
  
#include  
"Kismet/GameplayStatics.h"
```

In the preceding code snippet, we include **DrawDebugHelpers.h** because we'll call the **DrawDebugSphere** and **DrawDebugString** functions. We include **OwnershipRoles.h**, **OwnershipRolesCharacter.h** and

StaticMeshComponent.h so that the .cpp file knows about those classes. We finally include **GameplayStatics.h** because we'll call the **GetAllActorsOfClass** function.

11. In the constructor definition, create the static mesh component and set it as the root component:

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

12. Still in the constructor, set **bReplicates** to **true** to tell Unreal Engine that this actor replicates and should also exist in all of the clients:

```
bReplicates = true;
```

13. Delete the **BeginPlay** function definition.
14. In the **Tick** function, draw a debug sphere to help visualize the ownership radius, as shown in the following code snippet:

```
DrawDebugSphere(GetWorld(),
GetActorLocation(),
OwnershipRadius, 32,
FColor::Yellow);
```

15. Still in the **Tick** function, create the authority specific logic that will get the closest **AOwnershipRolesCharacter** within the ownership radius, and if it's different from the current one, then set it as the owner:

```
if (HasAuthority())
{
    AActor* NextOwner =
    nullptr;
    float MinDistance =
    OwnershipRadius;
    TArray<AActor*> Actors;
    UGameplayStatics::GetAllA
ctorsOfClass(this,
AOwnershipRolesCharacter::S
taticClass(), Actors);

    for (AActor* Actor :
Actors)
    {
```

```

        const float Distance =
GetDistanceTo(Actor);

        if (Distance <=
MinDistance)

    {

        MinDistance =
Distance;

        NextOwner = Actor;

    }

}

if (GetOwner() !=

NextOwner)

{

    SetOwner(NextOwner);

}

}

```

16. Still in the **Tick** function, convert the values for the local/remote roles (using the **ROLE_TO_STRING** macro we created earlier), the current owner, and the connection to the strings:

```

const FString
LocalRoleString =

```

```

ROLE_TO_STRING(GetLocalRole
());
const FString
RemoteRoleString =
ROLE_TO_STRING(GetRemoteRole());
const FString OwnerString =
GetOwner() != nullptr ?
GetOwner()->GetName() :
TEXT("No Owner");
const FString
ConnectionString =
GetNetConnection() != nullptr ? TEXT("Valid
Connection") :
TEXT("Invalid Connection");

```

17. To finalize the **Tick** function, use **DrawDebugString** to display onscreen the strings we converted in the previous step:

```

const FString Values =
FString::Printf(TEXT("Local
Role = %s\nRemoteRole =
%s\nOwner = %s\nConnection
= %s"), *LocalRoleString,
*RemoteRoleString,

```

```
*OwnerString,  
*ConnectionString);  
  
DrawDebugString(GetWorld(),  
GetActorLocation(), Values,  
nullptr, FColor::White,  
0.0f, true);
```

Note

*Instead of constantly using
GetLocalRole() ==
ROLE_Authority to check whether
the actor has authority, you can use
the **HasAuthority()** helper function,
defined in **AActor**.*

18. Next, open **OwnershipRolesCharacter.h** and declare the **Tick** function as protected:

```
virtual void Tick(float  
DeltaTime) override;
```

19. Now, open **OwnershipRolesCharacter.cpp** and include the header files as shown in the following code snippet:

```
#include  
"DrawDebugHelpers.h"
```

```
#include "OwnershipRoles.h"
```

20. Implement the **Tick** function:

```
void  
AOwnershipRolesCharacter::T  
ick(float DeltaTime)  
{  
    Super::Tick(DeltaTime);  
}
```

21. Convert the values for the local/remote roles (using the **ROLE_TO_STRING** macro we created earlier), the current owner, and the connection to strings:

```
const FString  
LocalRoleString =  
ROLE_TO_STRING(GetLocalRole  
());  
  
const FString  
RemoteRoleString =  
ROLE_TO_STRING(GetRemoteRol  
e());  
  
const FString OwnerString =  
GetOwner() != nullptr ?  
GetOwner()->GetName() :  
TEXT("No Owner");
```

```
const FString  
ConnectionString =  
GetNetConnection() !=  
nullptr ? TEXT("Valid  
Connection") :  
TEXT("Invalid Connection");
```

22. Use **DrawDebugString** to display onscreen the strings we converted in the previous step:

```
const FString Values =  
FString::Printf(TEXT("Local  
Role = %s\nRemoteRole =  
%s\nOwner = %s\nConnection  
= %s"), *LocalRoleString,  
*RemoteRoleString,  
*OwnerString,  
*ConnectionString);  
  
DrawDebugString(GetWorld(),  
GetActorLocation(), Values,  
nullptr, FColor::White,  
0.0f, true);
```

Finally, we can test the project.

23. Run the code and wait for the editor to fully load.
24. Create a new Blueprint called

OwnershipTestActor_BP in the **Content** folder that derives from **OwnershipTestActor**. Set **Mesh** to use a cube mesh, and drop an instance of it in the world.

25. Go to **Multiplayer Options** and set the number of clients to **2**.
26. Set the window size to **800x600**.
27. Play using **New Editor Window (PIE)**.

You should get the following output:

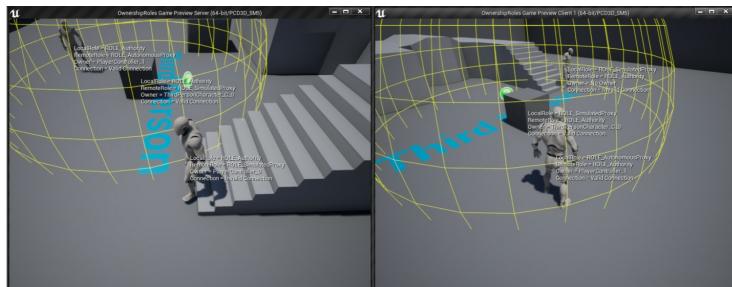


Figure 16.8: Expected result on the server and Client 1 window

By completing this exercise, you'll have a better understanding of how connections and ownership works. These are important concepts to know as everything related to replication is dependent on them.

Next time you see that an actor is not doing replication operations, you'll know that you need to check first whether it has a **valid connection** and an **owner**.

Now, let's analyze the displayed values in the server and client windows.

The Server Window

Have a look at the following output screenshot of the **Server** window from the previous exercise:

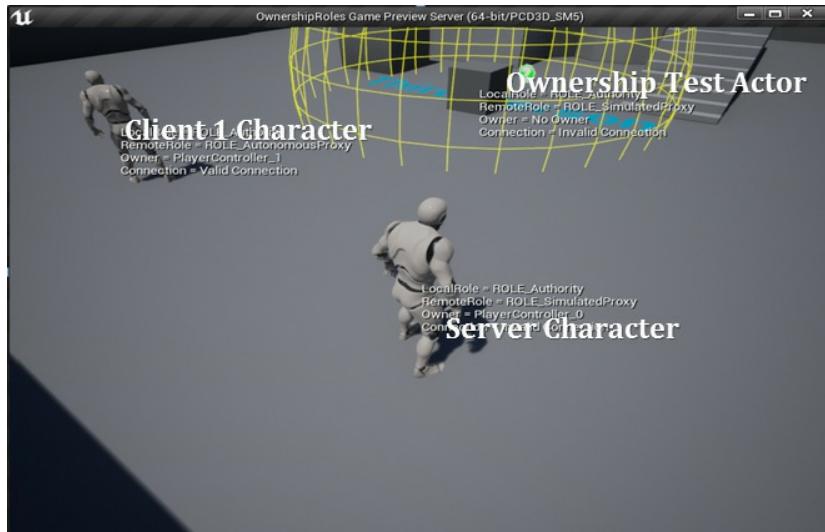


Figure 16.9: The Server window

Note

*The text that says **Server Character**, **Client 1 Character**, and **Ownership Test Actor** is not part of the original screenshot and was added to help you understand which character and actor is which.*

In the preceding screenshot, you can see **Server Character**, **Client 1 Character**, and the **Ownership Test** cube actor.

Let's first analyze the values for **Server Character**.

Server Character

This is the character that the listen server is controlling. The values associated with this character are as follows:

- **LocalRole = ROLE_Authority:**
because this character was spawned on the server, which is the current game instance.
- **RemoteRole = ROLE_SimulatedProxy:** because this character was spawned on the server, so the other clients should only simulate it.
- **Owner = PlayerController_0:**
because this character is being controlled by the client of the listen server, which uses the first **PlayerController** instance called **PlayerController_0**.
- **Connection = Invalid**
Connection: because we're the client of the listen server, so there is no need for a connection.

Next, we are going to be looking at **Client 1 Character** in the same window.

Client 1 Character

This is the character that **Client 1** is controlling. The values associated with this character are as follows:

- **LocalRole = ROLE_Authority:**
because this character was spawned on the server, which is the current game instance.
- **RemoteRole = ROLE_AutonomousProxy:** because this character was spawned on the server, but it's being controlled by another client.
- **Owner = PlayerController_1:**
because this character is being controlled by another client, which uses the second **PlayerController** instance called **PlayerController_1**.
- **Connection = Valid**
Connection: because this character is being controlled by another client, so a connection to the server is required.

Next, we are going to be looking at the **OwnershipTest** actor in the same window.

The OwnershipTest Actor

This is the cube actor that will set its owner to the closest character within a certain ownership radius. The values associated with this actor are as follows:

- **LocalRole = ROLE_Authority:** because this actor was placed in the level and spawned on the server, which is the current game instance.
- **RemoteRole = ROLE_SimulatedProxy:** because this actor was spawned in the server, but it's not being controlled by any client.
- **Owner** and **Connection** will have their values based on the closest character. If there isn't a character inside the ownership radius, then they will have the values of **No Owner** and **Invalid Connection**.

Now, let's have a look at the **Client 1** window:

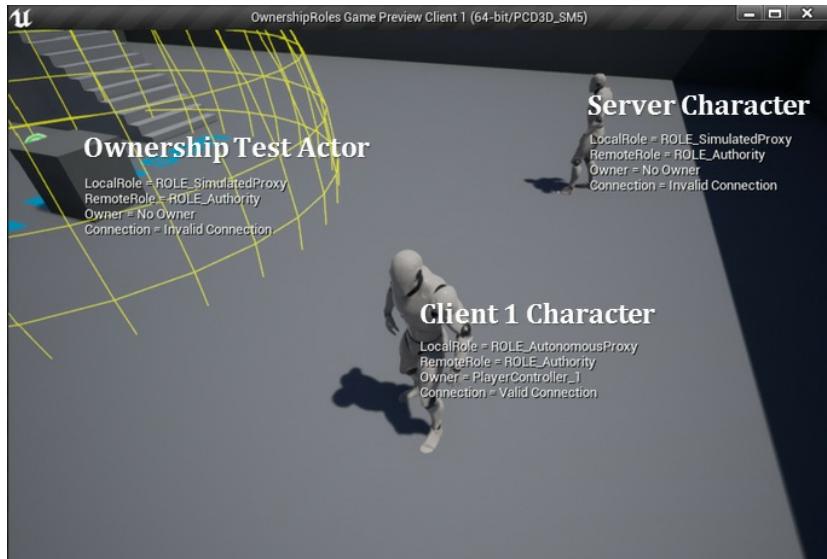


Figure 16.10: The Client 1 window

The Client 1 Window

The values for the **Client 1** window will be exactly the same as the **Server** window, except the values of **LocalRole** and **RemoteRole** will be reversed because they are always relative to the game instance that you are in.

Another exception is that the server character has no owner and the other connected clients won't have a valid connection. The reason for that is clients don't store player controllers and connections of other clients, only the server does, but this will be covered in more depth in *Chapter 18, Gameplay Framework Classes in Multiplayer*.

In the next section, we will be looking at variable replication.

Variable Replication

One of the ways the server can keep the clients synchronized

is by using variable replication. The way it works is that every specific number of times per second (defined per actor in the **AActor : NetUpdateFrequency** variable, which is also exposed to blueprints) the variable replication system in the server will check whether there are any replicated variables (*explained in the next section*) in the client that needs to be updated with the latest value.

If the variable meets all of the replication conditions, then the server will send an update to the client and enforce the new value.

For example, if you have a replicated **Health** variable and the client on its end uses a hacking tool to set the value of the variable from **10** to **100**, then the replication system will enforce the real value from the server and change it back to **10**, which nullifies the hack.

Variables are only sent to the client to be updated if:

- The variable is set to replicate.
- The value was changed on the server.
- The value on the client is different than on the server.
- The actor has replication enabled.
- The actor is relevant and meets all of the replication conditions.

One important thing to take into consideration is that the logic that determines whether a variable should be replicated or not is only executed **AActor : NetUpdateFrequency** times a second. In other words, the server doesn't send an update request to a client immediately after you change the

value of a variable on the server. It will only send that request when the variable replication system executes, which is **AActor::NetUpdateFrequency** times a second, and it has determined that the values from the client are different than the values from the server.

An example would be if you had an integer replicate a variable called **Test** that has a default value of **5**. If you call a function on the server that sets **Test** to **3** and in the next line changes it to **8**, then only the latter change would send an update request to the clients. The reason for this is these two changes were made in between the **NetUpdateFrequency** interval, so when the variable replication system executes, the current value is **8**, and as that is different to the value of the clients (which is still **5**), it will update them. If instead of setting it to **8**, you set it back to **5**, then no changes would be sent to the clients.

Replicated Variables

In Unreal Engine, any variable that can use the **UPROPERTY** macro can be set to replicate, and you can use two specifiers to do that.

Replicated

If you just want to say that a variable is replicated, then you use the **Replicated** specifier.

Have a look at the following example:

```
UPROPERTY(Replicated)  
float Health = 100.0f;
```

In the preceding code snippet, we declare a float variable called **Health**, as we normally do. The difference is that we've

added **UPROPERTY(Replicated)** to tell Unreal Engine that the **Health** variable will be replicated.

RepNotify

If you want to say that a variable is replicated and calls a function every time it's updated, then you use the **ReplicatedUsing=<Function Name>** specifier. Have a look at the following example:

```
UPROPERTY(ReplicatedUsing=OnRep_Health)  
float Health = 100.0f;  
  
UFUNCTION()  
void OnRep_Health()  
{  
    UpdateHUD();  
}
```

In the preceding code snippet, we declare a float variable called **Health**. The difference is that we've added **UPROPERTY(ReplicatedUsing=OnRep_Health)** to tell Unreal Engine that this variable will be replicated and every time it's updated it will call the **OnRep_Health** function, which, in this specific case, will call a function to update **HUD**.

Typically, the naming scheme for the callback function is **OnRepNotify_<Variable Name>** or **OnRep_<Variable Name>**.

Note

*The function used in the **ReplicatingUsing** specifier needs to be marked as **UFUNCTION()**.*

GetLifetimeReplicatedProps

Besides marking the variable as replicated, you'll also need to implement the **GetLifetimeReplicatedProps** function in the actor's **cpp** file. One thing to take into consideration is that this function is declared internally once you have at least one replicated variable, so you shouldn't declare it in the actor's header file. The purpose of this function is for you to tell how each replicated variable should replicate. You do this by using the **DOREPLIFETIME** macro and its variants on every variable that you want to replicate.

DOREPLIFETIME

This macro tells the replication system that the replicated variable (entered as an argument) will replicate to all clients without a replication condition.

Here's its syntax:

```
DOREPLIFETIME(<Class Name>, <Replicated  
Variable Name>);
```

Have a look at the following example:

```
void  
AVariableReplicationActor::GetLifetimeRepli  
catedProps(TArray< FLifetimeProperty >&  
OutLifetimeProps) const  
{  
    Super::GetLifetimeReplicatedProps(OutLife  
timeProps);  
  
    DOREPLIFETIME(AVariableReplicationActor,  
    Health);  
}
```

In the preceding code snippet, we use the **DOREPLIFETIME** macro to tell the replication system that the **Health** variable in the **AVariableReplicationActor** class will replicate without an extra condition.

DOREPLIFETIME_CONDITION

This macro tells the replication system that the replicated variable (entered as an argument) will replicate only to the clients that meet the condition (entered as an argument).

Here's the syntax:

```
DOREPLIFETIME_CONDITION(<Class Name>,
<Replicated Variable Name>, <Condition>);
```

The condition parameter can be one of the following values:

- **COND_InitialOnly:** The variable will only replicate once, with the initial replication.
- **COND_OwnerOnly:** The variable will only replicate to the owner of the actor.
- **COND_SkipOwner:** The variable won't replicate to the owner of the actor.
- **COND_SimulatedOnly:** The variable will only replicate to actors that are simulating.
- **COND_AutonomousOnly:** The variable will only replicate to actors that are autonomous.
- **COND_SimulatedOrPhysics:** The variable will only replicate to actors that are simulating or to actors with **bRepPhysics** set to true.

- **COND_InitialOrOwner**: The variable will only replicate once, with the initial replication or to the owner of the actor.
- **COND_Custom**: The variable will only replicate if its **SetCustomIsActiveOverride** Boolean condition (used in the **AActor::PreReplication** function) is true.

Have a look at the following example:

```
void
AVariableReplicationActor::GetLifetimeReplicatedProps(TArray< FLifetimeProperty >&
OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLife
timeProps);

    DOREPLIFETIME_CONDITION(AVariableReplicat
ionActor, Health, COND_OwnerOnly);
}
```

In the preceding code snippet, we use the **DOREPLIFETIME_CONDITION** macro to tell the replication system that the **Health** variable in the **AVariableReplicationActor** class will replicate only for the owner of this actor.

Note

*There are more **DOREPLIFETIME** macros available, but they won't be covered in this book. To see all of the variants, please check the **UnrealNetwork.h** file from the Unreal Engine 4 source code. See the instructions at: <https://docs.unrealengine.com/en-US/GettingStarted/DownloadingUnrealEngine/index.html>.*

Exercise 16.03: Replicating Variables Using Replicated, RepNotify, **DOREPLIFETIME**, and **DOREPLIFETIME_CONDITION**

In this exercise, we're going to create a C++ project that uses the Third Person template as a base and add two variables to the character that replicate in the following way:

- Variable **A** is a float that will use the **Replicated UPROPERTY** specifier and the **DOREPLIFETIME** macro.
- Variable **B** is an integer that will use the **ReplicatedUsing UPROPERTY** specifier and the **DOREPLIFETIME_CONDITION** macro.

The following steps will help you complete the exercise:

1. Create a new **Third Person** template project using **C++** called **VariableReplication** and save it to a location of your choosing.
2. Once the project has been created, it should open the editor as well as the Visual Studio solution.
3. Close the editor and go back to Visual Studio.
4. Open the **VariableReplicationCharacter.h** file.
5. Next, include the **UnrealNetwork.h** header file before the **VariableReplicationCharacter.generated.h**, which has the definition of the **DOREPLIFETIME** macros that we're going to use:

```
#include  
"Net/UnrealNetwork.h"
```
6. Declare the protected variables **A** and **B** as **UPROPERTY** using their respective

replication specifiers:

```
UPROPERTY(Replicated)  
float A = 100.0f;  
  
UPROPERTY(ReplicatedUsing =  
OnRepNotify_B)  
  
int32 B;
```

7. Declare the **Tick** function as protected:

```
virtual void Tick(float  
DeltaTime) override;
```

8. Since we've declared variable **B** as **ReplicatedUsing = OnRepNotify_B**, then we also need to declare the protected **OnRepNotify_B** callback function as **UFUNCTION**:

```
UFUNCTION()  
void OnRepNotify_B();
```

9. Now, open the **VariableReplicationCharacter.cpp** file and include the headers **Engine.h**, so we can use the **AddOnScreenDebugMessage** function, and **DrawDebugHelpers.h**,

so we can use the
DrawDebugString function:

```
#include "Engine/Engine.h"  
  
#include  
"DrawDebugHelpers.h"
```

10. Implement the
GetLifetimeReplicatedProps
function:

```
void  
AVariableReplicationCharact  
er::GetLifetimeReplicatedPr  
ops(TArray<  
FLifetimeProperty >&  
OutLifetimeProps) const  
  
{  
  
    Super::GetLifetimeReplica  
tedProps(OutLifetimeProps);  
  
}
```

11. Set that as the **A** variable, which will
replicate without any extra conditions:

```
DOREPLIFETIME(AVariableRepl  
icationCharacter, A);
```

12. Set that as the **B** variable, which will
only replicate to the owner of this

actor:

```
DOREPLIFETIME_CONDITION(AVariableReplicationCharacter,  
B, COND_OwnerOnly);
```

13. Implement the **Tick** function:

```
void  
AVariableReplicationCharacter::Tick(float DeltaTime)  
{  
    Super::Tick(DeltaTime);  
}
```

14. Next, run the authority-specific logic that adds **1** to **A** and **B**:

```
if (HasAuthority())  
{  
    A++;  
    B++;  
}
```

Since this character will be spawned on the server, then only the server will execute this logic.

15. Display the values of **A** and **B** on the

location of the character:

```
const FString Values =  
FString::Printf(TEXT("A =  
%.2f B = %d"), A, B);  
  
DrawDebugString(GetWorld(),  
GetActorLocation(), Values,  
nullptr, FColor::White,  
0.0f, true);
```

16. Implement the **RepNotify** function for variable **B**, which displays on the screen a message saying that the **B** variable was changed to a new value:

```
void  
AVariableReplicationCharacter::OnRepNotify_B()  
{  
  
    const FString String =  
FString::Printf(TEXT("B was  
changed by the server and  
is now %d!"), B);  
  
    GEngine-  
>AddOnScreenDebugMessage(-1  
, 0.0f,  
FColor::Red, String);  
}
```

Finally, you can test the project:

17. Run the code and wait for the editor to fully load.
18. Go to **Multiplayer Options** and set the number of clients to **2**.
19. Set the window size to **800x600**.
20. Play using **New Editor Window (PIE)**.

Once you complete this exercise, you will be able to play on each client and you'll notice that the characters are displaying their respective values for **A** and **B**.

Now, let's analyze the values displayed in the **Server** and **Client 1** windows.

The Server Window

In the **Server** window, you have the values for **Server Character**, which is the character controlled by the server, and in the background, you have values for **Client 1 Character**:



Figure 16.11: The Server window

The outputs that can be observed are as follows:

- **Server Character – A = 674.00
B = 574**
- **Client 1 Character – A =
670.00 B = 570**

At this specific point in time, **Server Character** has a value of **674** for **A** and **574** for **B**. The reason why **A** and **B** have different values is because **A** starts at **100** and **B** starts at **0**, which is the correct value after **574** ticks of **A++** and **B++**.

As for why the **Client 1 Character** doesn't have the same values as the Server Character, that is because **Client 1** was created slightly after the server, so in this case, the count will be off by **4** ticks of **A++** and **B++**.

Next, we will be looking at the **Client 1** window.

The Client 1 Window

In the **Client 1** window, you have the values for **Client 1 Character**, which is the character controlled by **Client 1**, and in the background, you have values for **Server Character**:

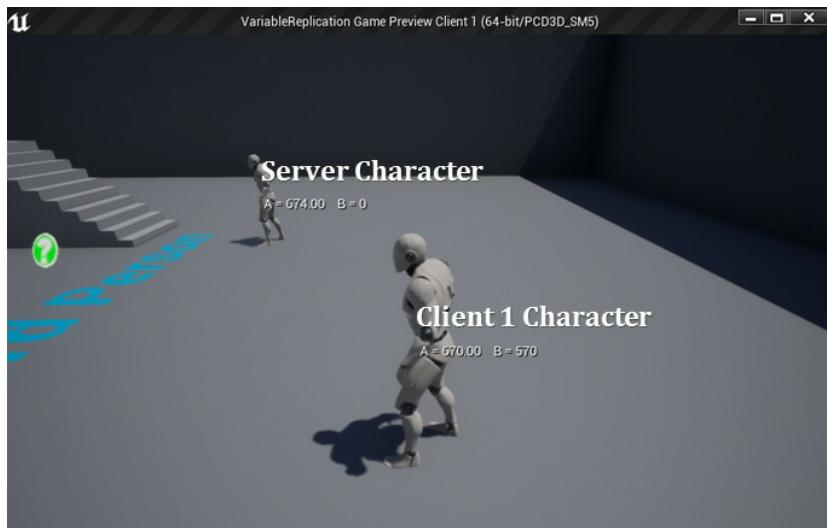


Figure 16.12: The Client 1 window

The outputs that can be observed are as follows:

- **Server Character – A = 674.00
B = 0**
- **Client 1 Character – A =
670.00 B = 570**

Client 1 Character has the correct values from the server, so the variable replication is working as intended. If you look at **Server Character**, **A** is **674**, which is correct, but **B** is **0**. The reason for that is **A** is using **DOREPLIFETIME**, which doesn't add any additional replication conditions, so it

will replicate the variable and keep the client up to date every time the variable is changed on the server.

The variable **B**, on the other hand, uses **DOREPLIFETIME_CONDITION** with **COND_OwnerOnly**, and since **Client 1** is not the client that owns **Server Character** (*the client of the listen server is*), then the value is not replicated and remains unchanged from the default value of **0**.

If you go back to the code and change the replication condition of **B** to use **COND_SimulatedOnly** instead of **COND_OwnerOnly**, you'll notice that the results will be reversed in **Client 1 window**. The value of **B** will be replicated for **Server Character**, but it won't replicate for its own character.

Note

*The reason why the **RepNotify** message is showing in the **Server** window instead of the client window is that, when playing in the editor, both windows share the same process, and therefore printing text on the screen won't be accurate. To get the correct behavior, you'll need to run the packaged version of the game.*

2D Blend Spaces

In *Chapter 2, Working with Unreal Engine*, we created a 1D Blend Space to blend between the movement states (*idle, walk, and run*) of a character based on the value of the Speed axis. For that specific example, it worked pretty well because you only needed one axis, but if we wanted the character to also be able to strafe, then we couldn't really do that.

To explore that case, Unreal Engine allows you to create 2D Blend Spaces. The concepts are almost exactly the same; the only difference is that you have an extra axis for animations, so you can blend between them not only horizontally, but also vertically.

Exercise 16.04: Creating a Movement 2D Blend Space

In this exercise, we're going to create a Blend Space that uses two axes instead of one. The vertical axis will be **Speed**, which will be between **0** and **800**. The horizontal axis will be **Direction**, which represents the relative angle (**-180** to **180**) between the velocity and the rotation/forward vector of the pawn.

The following figure will help you calculate the direction in this exercise:

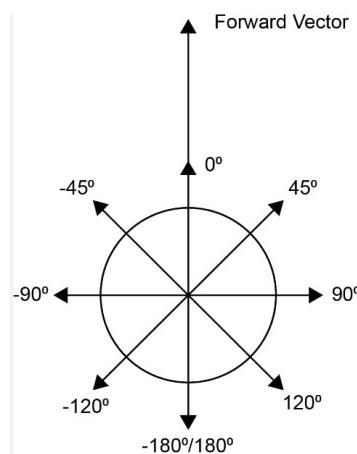


Figure 16.13: Direction values based on the angle between the forward vector and the velocity

In the preceding figure, you can see how the direction will be

calculated. The forward vector represents the direction that the character is currently facing, and the numbers represent the angle that the forward vector would make with the velocity vector if it was pointing in that direction. If the character was looking in a certain direction and you pressed a key to move the character to the right, then the velocity vector would be perpendicular to the forward vector. That would mean that the angle would be 90° , so that would be our direction.

If we set up our 2D Blend Space with that logic in mind, we can use the correct animation based on the character's movement angle.

The following steps will help you complete the exercise:

1. Create a new **Third Person** template project using **Blueprints** called **Blendspace2D** and save it to a location of your choosing.
2. Once the project has been created, it should open the editor.
3. Next, you will be importing the movement animations. In the editor, go to the **Content\Mannequin\Animations** folder.
4. Click on the **Import** button.
5. Go to the **Chapter16\Exercise16.04\Asse**

ts folder, select all of the **fbx** files, and hit the **Open** button.

6. In the import dialog, make sure you pick the character's skeleton and hit the **Import All** button.
7. Save all of the new files in the **Assets** folder.
8. Click on the **Add New** button and pick **Animation -> Blend Space**.
9. Next, select the character's skeleton.
10. Rename the Blend Space **BS_Movement** and open it.
11. Create the horizontal **Direction** axis (**-180 to 180**) and the vertical **Speed** axis (**0 to 800**) as shown in the following figure:

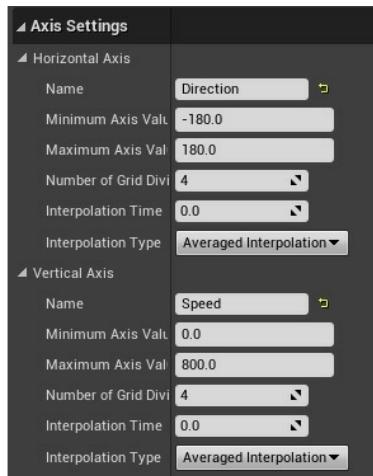


Figure 16.14: 2D Blend Space Axis Settings

12. Drag the **Idle_Rifle_Ironsights** animation onto the **5** grid entries where **Speed** is **0**.
13. Drag the **Walk_Fwd_Rifle_Ironsights** animation where **Speed** is **800** and **Direction** is **0**.
14. Drag the **Walk_Lt_Rifle_Ironsights** animation where **Speed** is **800** and **Direction** is **-90**.
15. Drag the **Walk_Rt_Rifle_Ironsights** animation where **Speed** is **800** and **Direction** is **90**.

You should end up with a Blend Space that can be previewed by holding *Shift* and moving the mouse.

16. Now, on the **Asset Details** panel, set the **Target Weight** **Interpolation Speed Per Sec** variable to **5** to make the

interpolation smoother.

17. Save and close the Blend Space.
18. Now, update the animation Blueprint to use the new Blend Space.
19. Go to
Content\Mannequin\Animations
and open the file that comes along with the Third Person template –
ThirdPerson_AnimBP.
20. Next, go to the event graph and create a new float variable called **Direction**.
21. Set the value of **Direction** with the result of the **Calculate Direction** function, which calculates the angle (-180° to 180°) between the pawn's **velocity** and **rotation**:

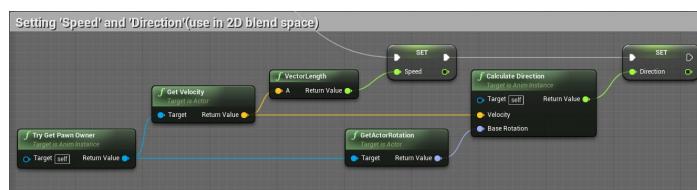


Figure 16.15: Calculating the Speed and Direction to use on the 2D Blend Space

Note

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

22. In **AnimGraph**, go to the **Idle/Run** state where the old 1D Blend Space is being used, as shown in the following screenshot:

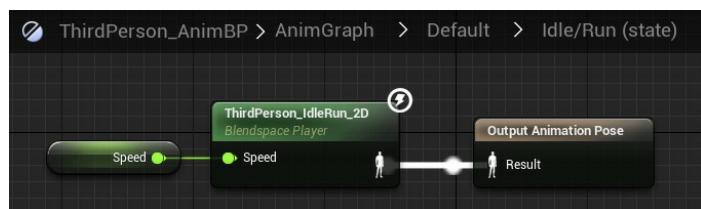


Figure 16.16: Idle/run state in the AnimGraph

23. Replace that Blend Space with **BS_Movement** and use the **Direction** variable like so:

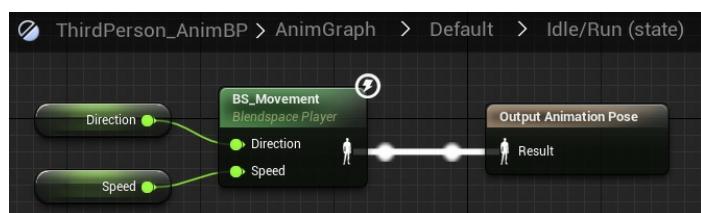


Figure 16.17: 1D Blend Space has been replaced by the new 2D Blend Space

24. Save and close the animation Blueprint. Now you need to update the

character.

25. Go to the
Content\ThirdPersonBP\Blueprints
ints folder and open
ThirdPersonCharacter.
26. On the **Details** panel for the character, set **Use Controller Rotation Yaw** to **true**, which will make the character's **Yaw** rotation always face the control rotation's Yaw.
27. Go to the character movement component and set **Max Walk Speed** to **800**.
28. Set **Orient Rotation to Movement** to **false**, which will prevent the character from rotating toward the direction of the movement.
29. Save and close the character Blueprint.

If you play the game now with two clients and move the character, it will walk forward and backward, but it will also strafe, as shown in the following screenshot:



Figure 16.18: Expected output on the server and Client 1 windows

By completing this exercise, you will have improved your understanding of how to create 2D Blend Spaces, how they work, and the advantages they provide compared to just using the regular 1D Blend Space.

In the next section, we will be looking at transforming a character's bone so that we can rotate the torso of the player up and down based on the camera's Pitch.

Transform (Modify) Bone

Before we move on, there is a very useful node that you can use in the AnimGraph called the **Transform (Modify) Bone** node, which allows you to translate, rotate, and scale a bone of a skeleton at *runtime*.

You can add it in the **AnimGraph** by *right-clicking* on an empty space, typing **transform modify**, and picking the node from the list. If you click on the **Transform (Modify) Bone** node, you'll have quite a few options on the **Details** panel.

Here's an explanation of what each option does.

- The **Bone to Modify** option will tell the node what bone is going to be transformed.

After that option, you have three sections representing each transform operation (**Translation**, **Rotation**, and **Scale**). In each section, you can do the following:

- **Translation, Rotation, Scale:** This option will tell the node how much of that specific transform operation you want to apply. The final result will depend on the mode (*covered in the next section*) you have selected.

There are two ways you can set this value:

- Setting a constant value such as **(X=0.0, Y=0.0, Z=0.0)**
- Using a variable, so it can be changed at runtime. To enable this, you need to take the following steps (this example is for **Rotation**, but the same concepts apply for **Translation** and **Scale**):

1. Click the checkbox next to the constant value and make sure it is checked.

Once you do that, the text boxes for the constant value will disappear.



Figure 16.19: Check the checkbox

Transform (Modify) Bone will add an input so you can plug in your variable:

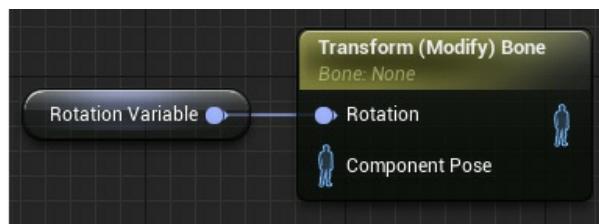


Figure 16.20: Variable used as an input on the Transform (Modify) Bone node

Setting the mode

This will tell the node what to do with the value. You can pick from one of these three options:

- **Ignore:** Don't do anything with the supplied value.
- **Add to Existing:** Grab the current value of the bone and add the supplied value to it.
- **Replace Existing:** Replace the current value of the bone with the

supplied value.

Setting the space

This will define the space the node should apply the transform to. You can pick from one of these four options:

- **World Space:** The transform will happen in the world space.
- **Component Space:** The transform will happen in the skeletal mesh component space.
- **Parent Bone Space:** The transform will happen in the parent bone's space of the selected bone.
- **Bone Space:** The transform will happen in the space of the selected bone.

Last but not least, you have the **Alpha**, which is a value that allows you to control the amount of transform that you want to apply. As an example, if you have the **Alpha** value as a float, then you'll have the following behavior with different values:

- If **Alpha** is 0.0, then no transform will be applied.
- If **Alpha** is 0.5, then it will only apply

half of the transform.

- If **Alpha** is 1.0, then it will apply the entire transform.

In the next exercise, we will use the **Transform (Modify) Bone** node to enable the character from *Exercise 16.04, Creating a Movement 2D Blend Space*, to look up and down based on the camera's rotation.

Exercise 16.05: Creating a Character That Looks up and down

In this exercise, we're going to duplicate the project from *Exercise 16.04, Creating a Movement 2D Blend Space*, and enable the character to look up and down based on the camera's rotation. To achieve this, we're going to use the **Transform (Modify) Bone** node to rotate the **spine_03** bone in the component space based on the Pitch of the camera.

The following steps will help you complete the exercise:

1. First, you need to duplicate and rename the project from *Exercise 16.04, Creating a Movement 2D Blend Space*.
2. Copy the **Blendspace2D** project

folder from *Exercise 16.04, Creating a Movement 2D Blend Space*, paste it in a new folder, and rename it **TransformModifyBone**.

3. Open the new project folder, rename the **Blendspace2D.uproject** file **TransformModifyBone.uproject**, and open it.

Next, you will be updating the animation Blueprint.

4. Go to **Content\Mannequin\Animations** and open **ThirdPerson_AnimBP**.
5. Go to the **Event Graph**, create a float variable called **Pitch**, and set it with the Pitch of the subtraction (or delta) between the pawn's rotation and the base aim rotation, as shown in the following figure:

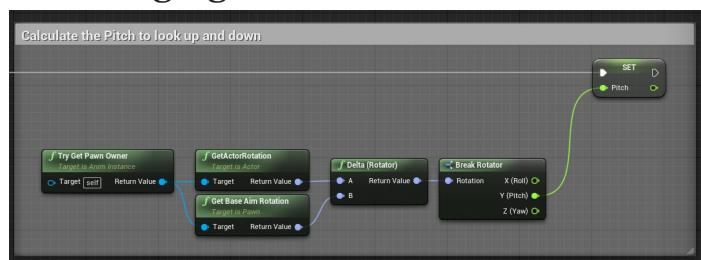


Figure 16.21: Calculating the Pitch

As an alternative to using the **Break Rotator** node, you can *right-click* on **Return Value** and pick **Split Struct Pin**.

Note

*The **Break Rotator** node allows you to separate a **Rotator** variable into three float variables that represent the **Pitch**, **Yaw**, and **Roll**. This is useful when you want to access the value of each individual component or if you only want to work with one or two components, and not with the whole rotation.*

Take into consideration that the **Split Struct Pin** option will only appear if **Return Value** is not connected to anything. Once you do the split, it will create three separate wires for **Roll**, **Pitch**, and **Yaw**, just like a break but without the extra node.

You should end up with the following:

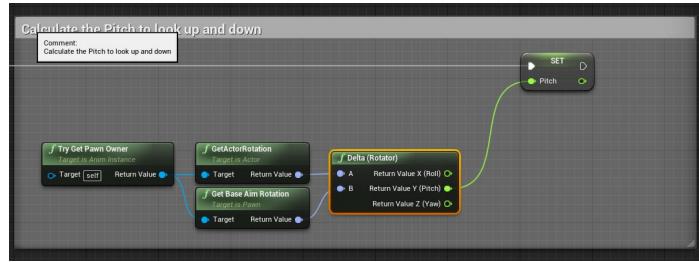


Figure 16.22: Calculating the Pitch to look up using the Split Struct Pin option

This logic uses the rotation of the pawn and subtracts it from the camera's rotation to get the difference in **Pitch**, as shown in the following figure:

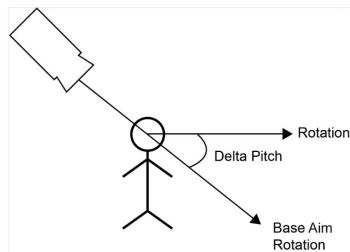


Figure 16.23: How to calculate the Delta Pitch

6. Next, go to **AnimGraph** and add a **Transform (Modify) Bone** node with the following settings:

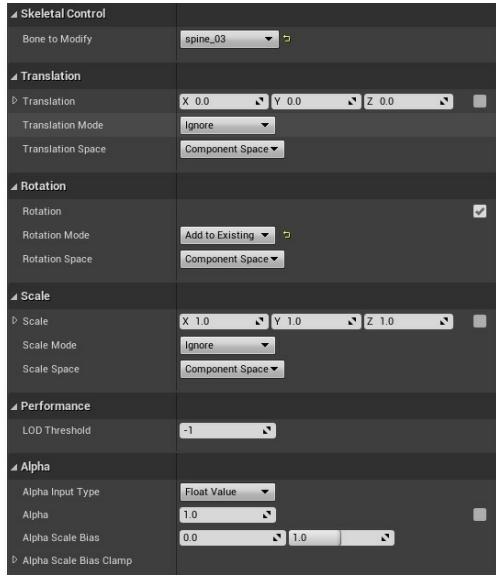


Figure 16.24: Settings for the Transform (Modify) Bone node

In the preceding screenshot, we've set **Bone to Modify** to **spine_03** because that is the bone that we want to rotate. We've also set **Rotation Mode** to **Add to Existing** because we want to keep the original rotation from the animation and add an offset to it. The rest of the options need to have their default value.

7. Connect the **Transform (Modify) Bone** node to the **State Machine** and the **Output Pose**, as shown in the following screenshot:

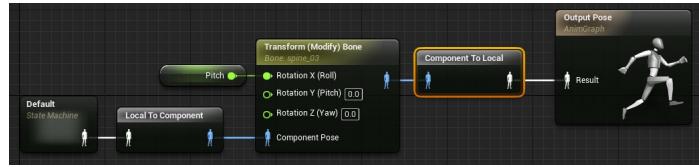


Figure 16.25: Transform (Modify) Bone connected to the Output Pose

In the preceding figure, you see the full **AnimGraph**, which will allow the character to look up and down by rotating the **spine_03** bone based on the camera Pitch. The **State Machine** will be the starting point, and from there, it will need to be converted into component space in order to be able to use the **Transform (Modify) Bone** node, which will connect to the **Output Pose** node after being converted back to local space.

Note

*The reason why we connect **Pitch** variable to the **Roll** is that the bone in the skeleton is internally rotated that way. You can use the **Split Struct Pin** on input parameters as well, so you don't have to add a **Make Rotator** node.*

If you test the project with two clients and move the mouse *up* and *down* on one of the characters, you'll notice that it will Pitch up and down, as shown in the following screenshot:

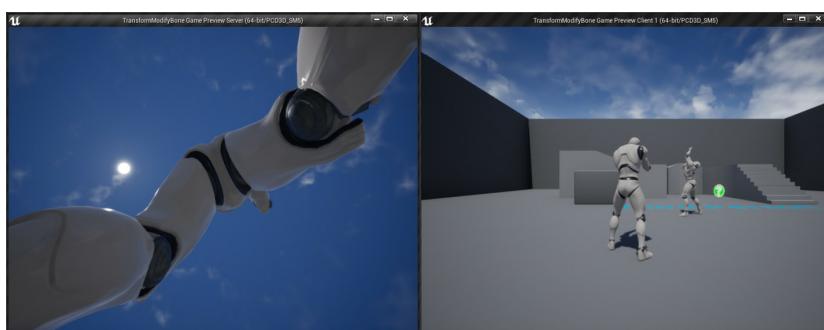


Figure 16.26: Character mesh pitching up and down,

based on the camera rotation

By completing this final exercise, you'll have an understanding of how to modify bones at runtime by using the **Transform (Modify) Bone** node in an animation blueprint. This node can be used in various scenarios, so it might prove really useful for you.

In the next activity, you're going to put everything you've learned to the test by creating the character we're going to use for our multiplayer FPS project.

Activity 16.01: Creating a Character for the Multiplayer FPS Project

In this activity, you'll create the character for the multiplayer FPS project that we're going to build in the next few chapters. The character will have a few different mechanics, but for this activity, you just need to create a character that walks, jumps, looks up/down, and has two replicated stats: health and armor.

The following steps will help you complete the activity:

1. Create a **Blank C++** project called **MultiplayerFPS** without the starter content.
2. Import the skeletal mesh and the animations from the **Activity16.01\Assets folder**

and place them in the
Content\Player\Mesh and
Content\Player\Animations
folders respectively.

3. Import the following sounds from the
Activity16.01\Assets folder to
Content\Player\Sounds:
 1. **Jump.wav**: Play this sound
on the
Jump_From_Stand_Irons
ights animation with a
Play Sound anim notify.
 2. **Footstep.wav**: Play this
sound every time a foot is on
the floor in every walk
animation by using the **Play**
Sound anim notify.
 3. **Spawn.wav**: Use this on the
SpawnSound variable in the
character.
4. Set up the skeletal mesh by retargeting
its bones and creating a socket called
Camera that is a child of the head bone
and has a Relative Location of
(X=7.88, Y=4.73, Z=-10.00).

5. Create a 2D Blend Space in
Content\Player\Animations
called **BS_Movement** that uses the
imported movement animations and a
Target Weight Interpolation
Speed Per Sec of 5.
6. Create the input mappings in the
Project Settings, using the
knowledge acquired in *Chapter 4,*
Player Input:
 1. Jump (action mapping) –
Spacebar
 2. Move Forward (axis
mapping) – *W* (scale **1.0**)
and *S* (scale **-1.0**)
 3. Move Right (axis mapping) –
A (scale **-1.0**) and *D* (scale
1.0)
 4. Turn (axis mapping) –
Mouse X (scale **1.0**)
 5. Look Up (axis mapping) –
Mouse Y (scale **-1.0**)
7. Create a C++ class called
FPSCharacter that does the

following:

1. Derives from the **Character** class.
2. Has a camera component attached to the skeletal mesh on the **Camera** socket and has **pawn control rotation** set to **true**.
3. Has variables for **health** and **armor** that only replicate to the owner.
4. Has variables for the maximum **health** and **armor**, as well as the percentage of how much damage the armor absorbs.
5. Has a constructor that initializes the camera, disables ticking, and sets **Max Walk Speed** to **800** and **Jump Z Velocity** to **600**.
6. On **BeginPlay**, plays the spawning sound and initializes the **health** with **max health** if it has

authority.

7. Creates and binds the functions to handle the input actions and axis.
8. Has functions to add/remove/set health. It also ensures the situation where the character is dead.
9. Has functions to add/set/absorb armor. The armor absorption reduces the armor based on the **ArmorAbsorption** variable and changes the damage value based on the formula:

$$\text{Damage} = (\text{Damage} * (1 - \text{ArmorAbsorption})) - \text{FMath}::\text{Min}(\text{RemainingArmor}, o);$$

8. Create an animation Blueprint in **Content\Player\Animations** called **ABP_Player** that has a **StateMachine** with the following states:

1. **Idle/Run:** Uses **BS_Movement** with the

Speed and Direction variables

2. **Jump:** Plays the jump animation and transitions from the **Idle/Run** states when the **Is Jumping** variable is **true**

It also uses **Transform (Modify) Bone** to make the character Pitch up and down based on the camera's Pitch.

9. Create a **UMG** widget in **Content\UI** called **UI_HUD** that displays the **Health** and **Armor** of the character in the format **Health: 100** and **Armor: 100**, using the knowledge acquired in *Chapter 15, Collectibles, Power-ups, and Pickups*.
10. Create a Blueprint in **Content\Player** called **BP_Player** that derives from **FPSCharacter** and set up the mesh component to have the following values:
 1. Use the **SK_Mannequin**

skeletal mesh

2. Use the **ABP_Player** animation Blueprint
3. Set **Location** to be equal to ($X=0.0, Y=0.0, Z=-88.0$)
4. Set **Rotation** to be equal to ($X=0.0, Y=0.0, Z=-90.0$)

Also, on the **Begin Play** event, it needs to create a widget instance of **UI_HUD** and add it to the viewport.

11. Create a Blueprint in **Content\Blueprints** called **BP_GameMode** that derives from **MultiplayerFPSGameModeBase**, which will use **BP_Player** as the **DefaultPawn** class.
12. Create a test map in **Content\Maps** called **DM-Test** and set it as the default map in **Project Settings**.

Expected output:

The result should be a project where each client will have a first-person character that can move, jump, and look around.

These actions will also be replicated, so each client will be able to see what the other client's character is doing.

Each client will also have a HUD that displays the health and the armor value.

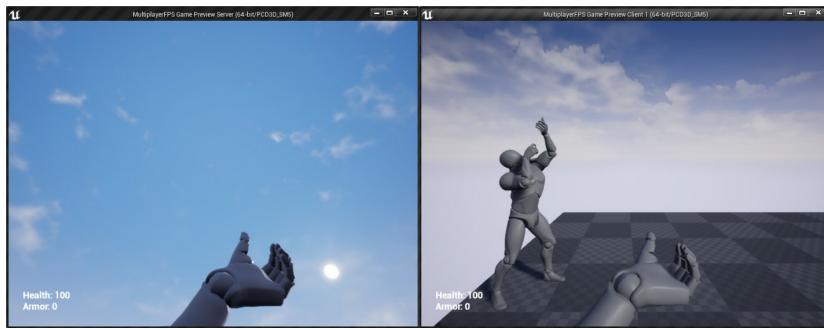


Figure 16.27: Expected output

Note

The solution to this activity can be found at:

<https://packt.live/338jEBx>.

The end result should be two characters that can see each other moving, jumping, and looking around. Each client also displays its character's health and armor value.

By completing this activity, you should have a good idea of how the server-client architecture, variable replication, roles, 2D Blend Spaces, and the **Transform (Modify) Bone** node work.

Summary

In this chapter, we have learned about some critical multiplayer concepts, such as how the server-client architecture works, the responsibilities of the server and the client, how the listen server is quicker to set up than the

dedicated server but not as lightweight, ownership and connections, roles, and variable replication.

We've also learned some useful techniques for animation, such as how to use 2D Blend Spaces, which allow you to have a two-axis grid to blend between animations, and the Transform (Modify) Bone node, which has the ability to modify the bones of a skeletal mesh at runtime. To finish off the chapter, we created a first-person multiplayer project where you have characters that can walk, look, and jump around, which will be the foundation of the multiplayer first-person shooter project that we will be working on for the next few chapters.

In the next chapter, we'll learn how to use RPCs, which allow clients and servers to execute functions on each other. We'll also cover how to use enumerations in the editor and how to use bi-directional circular array indexing, which allows you to cycle forward and backward through an array and loop back when it's outside the limits.

17. Remote Procedure Calls

Overview

In this chapter, you will be introduced to Remote Procedure Calls, which is another important multiplayer concept of Unreal Engine 4's Network Framework. You'll also learn how to use enumerations in Unreal Engine 4 and how to use bi-directional circular array indexing, which is a way to help you iterate arrays in both directions and be able to loop around when you go beyond its index limits.

By the end of this chapter, you'll understand how Remote Procedure Calls work to make the server and the clients execute logic on one another. You'll also be able to expose enumerations to the Unreal Engine 4 editor and use bi-directional circular array indexing to cycle through arrays.

Introduction

In the previous chapter, we covered some critical multiplayer concepts, including the server-client architecture, connections and ownership, roles, and variable replication. We also saw how the listen server is quicker to set up compared to the dedicated server but is not as lightweight. We used that knowledge to create a basic first-person shooter character that walks, jumps, and looks around.

In this chapter, we're going to cover **Remote Procedure**

Calls (RPCs), which is another important multiplayer concept that allows the server to execute functions on the clients and vice versa. So far, we've learned variable replication as a form of communication between the server and the clients, but that won't be enough, because the server might need to execute specific logic on the clients that doesn't involve updating the value of a variable. The client also needs a way to tell its intentions to the server, so that the server can validate the action and let the other clients know about it. This will ensure that the multiplayer world is synchronized, and we'll explore this in more detail within this chapter. We'll also cover how to use enumerations in Unreal Engine 4, as well as bi-directional circular array indexing, which helps you iterate an array in both directions and loops around when you go beyond its index limits.

In the first topic, we will be looking at RPCs.

Remote Procedure Calls

We've covered variable replication in *Chapter 16, Multiplayer Basics*, and, while a very useful feature, it is a bit limited in terms of allowing the execution of custom code in remote machines (client-to-server or server-to-client) for two main reasons:

- The first one is that variable replication is strictly a form of server-to-client communication, so there isn't a way for a client to use variable replication to tell the server to execute some custom logic by changing the value of a variable.

- The second reason is that variable replication, as the name suggests, is driven by the values of variables, so even if variable replication allowed the client-to-server communication, it would require you to change the value of a variable on the client to trigger a **RepNotify** functionality on the server to run the custom logic, which is not very practical.

To solve this problem, Unreal Engine 4 supports RPCs. An RPC works just like a normal function that can be defined and called, but instead of executing it locally, it will execute it on a remote machine. The main goal of using RPCs is to have the possibility to execute specific logic, which is not directly tied to a variable, on a remote machine. To be able to use RPCs, make sure you are defining them in an actor that has replication turned on.

There are three types of RPCs, and each one serves a different purpose:

- Server RPC
- Multicast RPC
- Client RPC

Let's go into detail about these three types and explain when you should use them:

Server RPC

You use a Server RPC every time you want the server to run a function on the actor that has defined the RPC. There are two main reasons why you would want to do this:

- The first one is for security reasons because when making multiplayer games, especially competitive ones, you always have to assume that the client will try to cheat. The way to make sure there is no cheating is by forcing the client to execute the functions that are critical to gameplay on the server.
- The second reason is for synchronicity because since the critical gameplay logic is only executed on the server, which means that the important variables are only going to be changed there, which will trigger the variable replication logic to update the clients whenever they are changed.

An example of this would be when a client's character tries to fire a weapon. Since there's always the possibility that the client might try to cheat, you can't just execute the fire weapon logic locally. The correct way of doing this is by having the client call a Server RPC that tells the server to validate the **Fire** action by making sure the character has enough ammo

and has the weapon equipped, and so on. If everything checks out, then it will deduct the ammo variable, and finally, it will execute a Multicast RPC (*covered in the next RPC type*) that will tell all of the clients to play the fire animation on that character.

DECLARATION

To declare a Server RPC, you use the **Server** specifier on the **UFUNCTION** macro. Have a look at the following example:

```
UFUNCTION(Server, Reliable, WithValidation)

void ServerRPCFunction(int32
IntegerParameter, float FloatParameter,
AActor* ActorParameter);
```

In the preceding code, the **Server** specifier is used on the **UFUNCTION** macro to state that the function is a Server RPC. You can have parameters on a Server RPC just like a normal function, but with some caveats that will be explained later in this topic, as well as the purpose of the **Reliable** and **WithValidation** specifiers.

EXECUTION

To execute a Server RPC, you call it from a client on the actor instance that defined it. Take a look at the following examples:

```
void ARPCTest::CallMyOwnServerRPC(int32
IntegerParameter)

{
    ServerMyOwnRPC(IntegerParameter);

}

void
```

```

ARPCTest::CallServerRPCofAnotherActor(AActor*
                                         OtherActor* OtherActor)

{
    if(OtherActor != nullptr)
    {
        OtherActor->ServerAnotherActorRPC();
    }
}

```

The first code snippet implements the **CallMyOwnServerRPC** function, which calls the **ServerMyOwnRPC** RPC function, defined in its own **ARPCTest** class, with an integer parameter. This will execute the implementation of the **ServerMyOwnRPC** function on the server version of that actor's instance.

The second code snippet implements the **CallServerRPCofAnotherActor** function, which calls the **ServerAnotherActorRPC** RPC function, defined in **AAnotherActor**, on the **OtherActor** instance as long as it's valid. This will execute the implementation of the **ServerAnotherActorRPC** function on the server version of the **OtherActor** instance.

VALID CONNECTION

One important thing to take into consideration when calling a Server RPC from a client is that the actor that defines it needs to have a valid connection. If you try to call a Server RPC on an actor that doesn't have a valid connection, then nothing will happen. You have to make sure that the actor is either a player controller, is being possessed by one (*if applicable*), or that its owning actor has a valid connection.

Multicast RPC

You use a Multicast RPC when you want the server to tell all clients to run a function on the actor that has defined the RPC.

An example of this would be when a client's character tries to fire a weapon. After the client calls the Server RPC to ask permission to fire the weapon and the server has handled the request (all validations checked out, the ammo has been deducted, and the line trace/projectile was processed), then we need to do a Multicast RPC so that all of the instances of that specific character play the fire animation. This will ensure that the character will always be playing the fire animation independent of which client is looking at the character.

DECLARATION

To declare a Multicast RPC, you need to use the **NetMulticast** specifier on the **UFUNCTION** macro. Have a look at the following example:

```
UFUNCTION(NetMulticast)  
void MulticastRPCFunction(int32  
IntegerParameter, float FloatParameter,  
AActor* ActorParameter);
```

In the preceding code, the **NetMulticast** specifier is used on the **UFUNCTION** macro to say that the following function is a Multicast RPC. You can have parameters on a Multicast RPC just like a normal function, but with the same caveats as the Server RPC.

EXECUTION

To execute a Multicast RPC, you call it from the server on the actor instance that defined it. Take a look at the following

examples:

```
void ARPCTest::CallMyOwnMulticastRPC(int32
IntegerParameter)

{
    MulticastMyOwnRPC(IntegerParameter);

}

void
ARPCTest::CallMulticastRPCofAnotherActor(AA
notherActor* OtherActor)

{
    if(OtherActor != nullptr)

    {
        OtherActor->MulticastAnotherActorRPC();
    }
}
```

The first code snippet implements the **CallMyOwnMulticastRPC** function, which calls the **MulticastMyOwnRPC** RPC function, defined in its own **ARPCTest** class, with an integer parameter. This will execute the implementation of the **MulticastMyOwnRPC** function on all of the clients' versions of that actor's instance.

The second code snippet implements the **CallMulticastRPCofAnotherActor** function, which calls the **MulticastAnotherActorRPC** RPC function, defined in **AAnotherActor**, on the **OtherActor** instance as long as it's valid. This will execute the implementation of the **MulticastAnotherActorRPC** function on all of the clients' versions of the **OtherActor** instance.

Client RPC

You use a Client RPC when you want to run a function only on the owning client of the actor that has defined the RPC. To set the owning client, you need to call Set Owner on the server and set it with the client's player controller.

An example of this would be when a character is hit by a projectile and plays a pain sound that only that client will hear. By calling a Client RPC from the server, the sound will only be played on the owning client, so the other clients won't hear it.

DECLARATION

To declare a Client RPC, you need to use the **Client** specifier on the **UFUNCTION** macro. Have a look at the following example:

```
UFUNCTION(Client)  
void ClientRPCFunction(int32  
IntegerParameter, float FloatParameter,  
AActor* ActorParameter);
```

In the preceding code, the **Client** specifier is used on the **UFUNCTION** macro to say that the following function is a Client RPC. You can have parameters on a Client RPCs just like a normal function, but with the same caveats as the Server RPC and Multicast RPC.

EXECUTION

To execute a Client RPC, you call it from the server on the actor instance that defined it. Take a look at the following examples:

```

void ARPCTest::CallMyOwnClientRPC(int32
IntegerParameter)

{
    ClientMyOwnRPC(IntegerParameter);

}

void
ARPCTest::CallClientRPCOfAnotherActor(AAno
therActor* OtherActor)

{
    if(OtherActor != nullptr)
    {
        OtherActor->ClientAnotherActorRPC();
    }
}

```

The first code snippet implements the **CallMyOwnClientRPC** function, which calls the **ClientMyOwnRPC** RPC function, defined in its own **ARPCTest** class, with an integer parameter. This will execute the implementation of the **ClientMyOwnRPC** function on the owning client's version of that actor's instance.

The second code snippet implements the **CallClientRPCOfAnotherActor** function, which calls the **ClientAnotherActorRPC** RPC function, defined in **AAnotherActor**, on the **OtherActor** instance as long as it's valid. This will execute the implementation of the **ClientAnotherActorRPC** function on the owning client's version of the **OtherActor** instance.

Important Considerations

When Using RPCs

RPCs are very useful, but there are a couple of things that you need to take into consideration when using them, such as:

Implementation

The implementation of an RPC differs slightly to that of a typical function. Instead of implementing the function as you normally do, you should only implement the **_Implementation** version of it, even though you didn't declare it in the header file. Have a look at the following examples:

Server RPC:

```
void
ARPCTest::ServerRPCTest_Implementation(int3
2 IntegerParameter, float FloatParameter,
AActor* ActorParameter)

{



}
```

In the preceding code snippet, we implement the **_Implementation** version of the **ServerRPCTest** function, which uses three parameters.

Multicast RPC:

```
void
ARPCTest::MulticastRPCTest_Implementation(i
nt32 IntegerParameter, float
FloatParameter, AActor* ActorParameter)

{



}
```

In the preceding code snippet, we implement the **_Implementation** version of the **MulticastRPCTest**

function, which uses three parameters.

Client RPC:

```
void ARPCTest::ClientRPCTest_Implementation(int32 IntegerParameter, float FloatParameter, AActor* ActorParameter)
{
}
```

In the preceding code snippet, we implement the **_Implementation** version of the **ClientRPCTest** function, which uses three parameters.

As you can see from the previous examples, independent of the type of the RPC you are implementing, you should only implement the **_Implementation** version of the function and not the normal one, as demonstrated in the following code snippet:

```
void ARPCTest::ServerRPCFunction(int32 IntegerParameter, float FloatParameter, AActor* ActorParameter)
{
}
```

In the preceding code, we're defining the normal implementation of **ServerRPCFunction**. If you implement the RPC like this, you'll get an error saying that it was already implemented. The reason for this is that when you declare the RPC function in the header file, Unreal Engine 4 will automatically create the normal implementation internally, which will later call the **_Implementation** version. If you create your version of the normal implementation, the build will fail because it will find two implementations of the same function. To fix this, just make sure to only implement the **_Implementation** version of the RPC.

Next, we move to name prefixes.

Name Prefix

In Unreal Engine 4, it's good practice to prefix RPCs with their corresponding types. Have a look at the following examples:

- A **Server RPC** called RPCFunction
should be named
ServerRPCFunction.
- A **Multicast RPC** called RPCFunction
should be named
MulticastRPCFunction.
- A **Client RPC** called RPCFunction
should be named
ClientRPCFunction.

Return Value

Since the call and execution of RPCs are typically done on different machines, you can't have a return value, so it always needs to be void.

Overriding

You can override the implementation of an RPC to expand or bypass the parent's functionality by declaring and implementing the **_IMPLEMENTATION** function in the child class without the **UFUNCTION** macro. Here is an example:

The declaration on the parent class:

```
UFUNCTION(Server)
```

```
void ServerRPCTest(int32 IntegerParameter);
```

In the preceding code snippet, we have the declaration of the parent class of the **ServerRPCTest** function, which uses one integer parameter.

The overridden declaration on the child class:

```
virtual void  
ServerRPCTest_Implementation(int32  
IntegerParameter) override;
```

In the preceding code snippet, we override the declaration of the **ServerRPCTest_Implementation** function in the child class header file. The implementation of the function is just like any other override, with the possibility of calling **Super::ServerRPCTest_Implementation** if you still want to execute the parent functionality.

Supported Parameter Types

When using RPCs, you can add parameters just like any other function. Currently, most common types are supported, including **bool**, **int32**, **float**, **FString**, **FName**, **TArray**, **TSet**, and **TMap**. The types that you have to pay more attention to are pointers to any **UObject** class or sub-class, especially actors.

If you create an RPC with an actor parameter, then that actor also needs to exist on the remote machine, or else it will be **nullptr**. Another important thing to take into account is that the instance names of each version of the actor can be different. That means that if you call an RPC with an actor parameter, then the instance name of the actor when calling the RPC might be different than the one when executing the RPC on the remote machine. Here is an example to help you understand this:



Figure 17.1: The listen server and two clients running

In the preceding example, you can see three clients running (one of them is a listen server) and each window is displaying the name of all of the character instances. If you look at the Client 1 window, its controlled character instance is called **ThirdPersonCharacter_C_0**, but on the Server window, that equivalent character is called **ThirdPersonCharacter_C_1**. That means that if Client 1 calls a Server RPC and passes its **ThirdPersonCharacter_C_0** as an argument, then when the RPC is executed on the server, the parameter will be **ThirdPersonCharacter_C_1**, which is the instance name of the equivalent character in that machine.

Executing RPCs on the Target Machine

You can call RPCs directly on its target machine and it will still execute. In other words, you can call a Server RPC on the server and it will execute, as well as a Multicast/Client RPC on the client, but in this case, it will only execute the logic on the client that called the RPC. Either way, in these cases, you should always call the **_Implementation** version directly instead so as to execute the logic faster.

The reason for this is that the **_Implementation** version just holds the logic to execute and doesn't have the overhead of creating and sending the RPC request through the network that the regular call has.

Have a look at the following example of an actor that has authority on the server:

```

void ARPCTest::CallServerRPC(int32
IntegerParameter)

{
    if(HasAuthority())
    {
        ServerRPCFunction_Implementation(IntegerParameter);
    }
    else ServerRPCFunction(IntegerParameter);
}

```

In the preceding example, you have the **CallServerRPC** function that calls **ServerRPCFunction** in two different ways. If the actor is already on the server, then it calls **ServerRPCFunction_Implementation**, which will skip the overhead as mentioned previously.

If the actor is not on the server, then it executes the regular call by using **ServerRPCFunction**, which adds the required overhead of creating and sending the RPC request through the network.

Validation

When you define an RPC, you have the option of using an additional function to check whether there are any invalid inputs before the RPC is called. This is used to avoid processing the RPC if the inputs are invalid, due to cheating or for some other reason.

To use validation, you need to add the **WithValidation** specifier in the **UFUNCTION** macro. When you use that specifier, you will be forced to implement the **_Validate** version of the function, which will return a Boolean stating whether the RPC can be executed.

Have a look at the following example:

```
UFUNCTION(Server, WithValidation)
void ServerSetHealth(float NewHealth);
```

In the preceding code, we've declared a validated Server RPC called **ServerSetHealth**, which takes a float parameter for the new value of **Health**. As for the implementation, this is as follows:

```
bool
ARPCTest::ServerSetHealth_Validate(float
NewHealth)

{
    return NewHealth <= MaxHealth;
}

void
ARPCTest::ServerSetHealth_Implementation(f1
oat NewHealth)

{
    Health = NewHealth;
}
```

In the preceding code, we implement the **_Validate** function, which will check whether the new health is less than or equal to the maximum value of the health. If a client tries to hack and call **ServerSetHealth** with **200** and **MaxHealth** is **100**, then the RPC won't be called, which prevents the client from changing the health with values outside a certain range. If the **_Validate** function returns **true**, the **_Implementation** function is called as usual, which sets **Health** with the value of **NewHealth**.

Reliability

When you declare an RPC, you are required to either use the

Reliable or **Unreliable** specifiers in the **UFUNCTION** macro. Here's a quick overview of what they do:

- **Reliable:** Used when you want to make sure the RPC is executed, by repeating the request until the remote machine confirms its reception. This should only be used for RPCs that are very important, such as executing critical gameplay logic. Here is an example of how to use it:

```
UFUNCTION(Server, Reliable)  
void  
ServerReliableRPCFunction(i  
nt32 IntegerParameter);
```

- **Unreliable:** Used when you don't care whether the RPC is executed due to bad network conditions, such as playing a sound or spawning a particle effect. This should only be used for RPCs that aren't very important or are called very frequently to update values, since it wouldn't matter if one call missed because it's updating very often. Here is an example of how to use it:

```
UFUNCTION(Server,
```

```
Unreliable)

void
ServerUnreliableRPCFunction
(int32 IntegerParameter);
```

Note

For more information on RPCs, please visit

<https://docs.unrealengine.com/en-US/Gameplay/Networking/Actors/RPCs/index.html>.

In the next exercise, you will see how to implement different types of RPCs.

Exercise 17.01: Using Remote Procedure Calls

In this exercise, we're going to create a C++ project that uses the **Third Person** template and we're going to expand it in the following way:

- Add a fire timer variable that will prevent the client from spamming the fire button during the fire animation.
- Add a new Ammo integer variable that

defaults to **5** and replicates to all of the clients.

- Add a **Fire Anim montage** that is played when the server tells the client that the shot was valid.
- Add a **No Ammo Sound** that will play when the server tells the client that they didn't have sufficient ammo.
- Every time the player presses the *Left Mouse Button*, the client will perform a reliable and validated Server RPC that will check whether the character has sufficient ammo. If it does, it will subtract **1** from the **Ammo** variable and call an unreliable Multicast RPC that plays the fire animation in every client. If it doesn't have ammo, then it will execute an unreliable Client RPC that will play the **No Ammo Sound** that will only be heard by the owning client.

The following steps will help you complete the exercise:

1. Create a new **Third Person** template project using **C++** called **RPC** and save it to a location of your choosing.

2. Once the project has been created, it should open the editor as well as the Visual Studio solution.
3. Close the editor and go back to Visual Studio.
4. Open the **RPCCharacter.h** file and include the **UnrealNetwork.h** header file, which has the definition of the **DOREPLIFETIME_CONDITION** macro that we're going to use:

```
#include  
"Net/UnrealNetwork.h"
```

5. Declare the protected timer variable to prevent the client from spamming the **Fire** action:

```
FTimerHandle FireTimer;
```

6. Declare the protected replicated ammo variable, which starts with **5** shots:

```
UPROPERTY(Replicated)  
int32 Ammo = 5;
```

7. Next, declare the protected animation montage variable that will be played when the character fires:

```
UPROPERTY(EditDefaultsOnly,  
Category = "RPC Character")  
  
UAnimMontage*  
FireAnimMontage;
```

8. Declare the protected sound variable that will be played when the character has no ammo:

```
UPROPERTY(EditDefaultsOnly,  
Category = "RPC Character")  
  
USoundBase* NoAmmoSound;
```

9. Override the **Tick** function:

```
virtual void Tick(float  
DeltaSeconds) override;
```

10. Declare the input function that will process the pressing of the *Left Mouse Button*:

```
void OnPressedFire();
```

11. Declare the reliable and validated Server RPC for firing:

```
UFUNCTION(Server, Reliable,  
WithValidation, Category =  
"RPC Character")  
  
void ServerFire();
```

12. Declare the unreliable Multicast RPC that will play the fire animation on all of the clients:

```
UFUNCTION(NetMulticast,  
Unreliable, Category = "RPC  
Character")  
  
void MulticastFire();
```

13. Declare the unreliable Client RPC that will play a sound only in the owning client:

```
UFUNCTION(Client,  
Unreliable, Category = "RPC  
Character")  
  
void  
ClientPlaySound2D(USoundBas  
e* Sound);
```

14. Now, open the **RPCCharacter.cpp** file and include

DrawDebugHelpers.h,
GameplayStatics.h,
TimerManager.h and **World.h**:

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

```
#include "TimerManager.h"  
#include "Engine/World.h"
```

15. At the end of the constructor, enable the **Tick** function:

```
PrimaryActorTick.bCanEverTi  
ck = true;
```

16. Implement the **GetLifetimeReplicatedProps** function so that the **Ammo** variable will replicate to all of the clients:

```
void  
ARPCCharacter::GetLifetimeR  
eplicatedProps(TArray<  
FLifetimeProperty >&  
OutLifetimeProps) const  
{  
    Super::GetLifetimeReplica  
tedProps(OutLifetimeProps);  
  
    DOREPLIFETIME(ARPCCharact  
er, Ammo);  
}
```

17. Next, implement the **Tick** function, which displays the value of the

Ammo variable:

```
void
ARPCCharacter::Tick(float
DeltaSeconds)

{
    Super::Tick(DeltaSeconds)
;

    const FString AmmoString
=
FString::Printf(TEXT("Ammo
= %d"), Ammo);

    DrawDebugString(GetWorld(
), GetActorLocation(),
AmmoString, nullptr,
FColor::White, 0.0f, true);

}
```

18. At the end of the **SetupPlayerInputController** function, bind the **Fire** action to the **OnPressedFire** function:

```
PlayerInputComponent-
>BindAction("Fire",
IE_Pressed, this,
&ARPCCharacter::OnPressedFi
re);
```

19. Implement the function that will process the press of the *Left Mouse Button*, which will call the fire Server RPC:

```
void  
ARPCCharacter::OnPressedFire()  
{  
    ServerFire();  
}
```

20. Implement the fire Server RPC validation function:

```
bool  
ARPCCharacter::ServerFire_V  
alidate()  
{  
    return true;  
}
```

21. Implement the fire Server RPC implementation function:

```
void  
ARPCCharacter::ServerFire_I  
mplementation()
```

```
{
```

```
}
```

22. Now, add the logic to abort the function if the fire timer is still active since we fired the last shot:

```
if  
  (GetWorldTimerManager().IsT  
imerActive(FireTimer))  
  
{  
  return;  
  
}
```

23. Check whether the character has ammo. If it doesn't, then play **NoAmmoSound** only in the client that controls the character and abort the function:

```
if (Ammo == 0)  
  
{  
  ClientPlaySound2D(NoAmmoS  
ound);  
  
  return;  
  
}
```

24. Deduct the ammo and schedule the **FireTimer** variable to prevent this function from being spammed while playing the fire animation:

```
Ammo--;  
  
GetWorldTimerManager().SetT  
imer(FireTimer, 1.5f,  
false);
```

25. Call the fire Multicast RPC to make all the clients play the fire animation:

```
MulticastFire();
```

26. Implement the fire Multicast RPC, which will play the fire animation montage:

```
void  
ARPCCharacter::MulticastFir  
e_Implementation()  
{  
    if (FireAnimMontage !=  
        nullptr)  
    {  
        PlayAnimMontage(FireAni  
mMontage);  
    }  
}
```

```
}
```

27. Implement the Client RPC that plays a 2D sound:

```
void  
ARPCCharacter::ClientPlaySo  
und2D_Implementation(USound  
Base* Sound)  
  
{  
  
    UGameplayStatics::PlaySou  
nd2D(GetWorld(), Sound);  
  
}
```

Finally, you can launch the project in the editor.

28. Compile the code and wait for the editor to fully load.
29. Go to **Project Settings**, go to **Engine**, then **Input**, and add the **Fire** action binding:

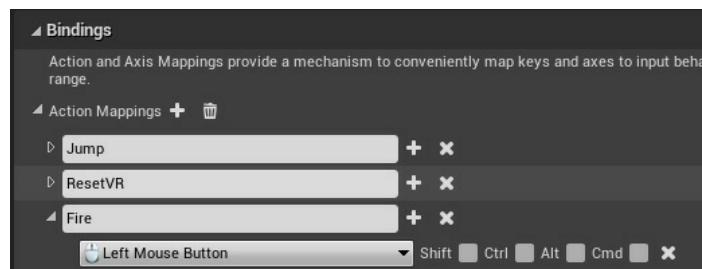


Figure 17.2: Adding the new Fire

action binding

30. Close **Project Settings**.
31. In **Content Browser**, go to the **Content\Mannequin\Animations** folder.
32. Click the **Import** button, go to the **Exercise17.01\Assets** folder and import the **ThirdPersonFire.fbx** file, and then make sure it's using the **UE4_Mannequin_Skeleton** skeleton.

Note

*The **Assets** folder mentioned earlier is available on our GitHub repository at <https://packt.live/36pEvAT>.*

33. Open the new animation and, on the details, the panel finds the **Enable Root Motion** option and sets it to true. This will prevent the character from moving when playing the animation.
34. Save and close **ThirdPersonFire**.

35. *Right-click* **ThirdPersonFire** on **Content Browser** and pick **Create -> AnimMontage**.

36. Rename **AnimMontage** to **ThirdPersonFire_Montage**.

37. The **Animations** folder should look like this:

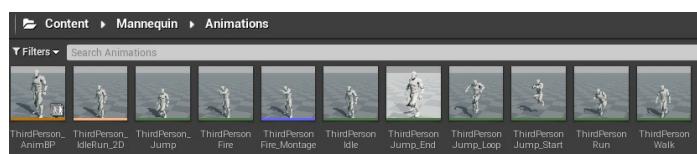


Figure 17.3: The animations folder for the Mannequin

38. Open **ThirdPerson_AnimBP** and then open **AnimGraph**.

39. *Right-click* on an empty part of the graph, add a **DefaultSlot** node (to be able to play the animation montage), and connect it between **State Machine** and **Output Pose**. You should get the following output:

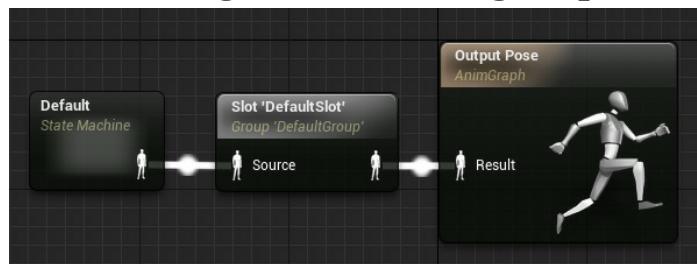


Figure 17.4: The AnimGraph of the character

40. Save and close **ThirdPerson_AnimBP**.
41. In **Content Browser**, go to the **Content** folder, create a new folder called **Audio**, and open it.
42. Click the **Import** button and go to the **Exercise17.01\Assets** folder, import **noammo.wav**, and save it.
43. Go to **Content\ThirdPersonCPP\Blueprints** and open the **ThirdPersonCharacter** blueprint.
44. In the class defaults, set **No Ammo Sound** to use **noammo**, and set **Fire Anim Montage** to use **ThirdPersonFire_Montage**.
45. Save and close **ThirdPersonCharacter**.
46. Go to the Multiplayer options and set the number of clients to **2**.

47. Set the window size to 800x600 and play using PIE.

You should get the following output:



Figure 17.5: The end result of the exercise

By completing this exercise, you will be able to play on each client, and every time you press the *Left Mouse Button*, the character of the client will play the **Fire Anim** montage, which all clients will be able to see, and its ammo will reduce by **1**. If you try to fire when the ammo is **0**, that client will hear **No Ammo Sound** and won't do the fire animation, because the server didn't call the Multicast RPC. If you try to spam the fire button, you'll notice that it will only trigger a new fire once the animation is finished.

In the next section, we will be looking at enumerations, which are used in game development for many different things, such as managing the state of a character (whether it's idle, walking, attacking, or dead, and so on) or to assign a human-friendly name for each index in the gear slot array (head, primary weapon, secondary weapon, torso, hands, belt, pants, and so on).

Enumerations

An enumeration is a user-defined data type that holds a list of integer constants, where each item has a human-friendly

name assigned by you, which makes the code easier to read. As an example, we could use an integer variable to represent the different states that a character can be in – **0** means it's idle, **1** means it's walking, and so on. The problem with this approach is that when you start writing code such as **if(State == 0)**, it will become hard to remember what **0** means, especially if you have a lot of states, without using some documentation or comments to help you remember. To fix this problem, you should use enumerations, where you can write code such as **if(State == EState::Idle)**, which is much more explicit and easier to understand.

In C++, you have two types of enums, the older raw enums and the new enum classes, introduced in C++11. If you want to use C++ enumerations in the editor, your first instinct might be to do it in the typical way, which is by declaring a variable or a function that uses the enumeration as a parameter, with **UPROPERTY** or **UFUNCTION**, respectively.

The problem is, if you try to do that, you'll get a compilation error. Take a look at the following example:

```
enum class ETestEnum : uint8
{
    EnumValue1,
    EnumValue2,
    EnumValue3
};
```

In the preceding code snippet, we declare an enum class called **ETestEnum** that has three possible values – **EnumValue1**, **EnumValue2**, and **EnumValue3**.

After that, try either one of the following examples:

```
UPROPERTY()
```

```
ETestEnum TestEnum;  
  
UFUNCTION()  
  
void SetTestEnum(ETestEnum NewTestEnum) {  
    TestEnum = NewTestEnum; }
```

In the preceding code snippet, we declare a **UPROPERTY** variable and **UFUNCTION** function that uses the **ETestEnum** enumeration in a class. If you try to compile, you'll get the following compilation error:

```
error : Unrecognized type 'ETestEnum' -  
type must be a UCLASS, USTRUCT or UENUM
```

Note

*In Unreal Engine 4, it's good practice to prefix the name of an enumeration with the letter E. Examples include **EWeaponType** and **EAMmoType**.*

This error happens because when you try to expose a class, struct, or enumeration to the editor with the **UPROPERTY** or **UFUNCTION** macro, you need to add it to the Unreal Engine 4 Reflection System by using the **UCLASS**, **USTRUCT**, and **UENUM** macros, respectively.

Note

*You can learn more about the Unreal Engine 4 Reflection System by visiting the following link:
<https://www.unrealengine.com/en-US/blog/unreal-property-system-reflection>.*

With that knowledge in mind, it is simple to fix the previous error, so just do the following:

```
UENUM()
```

```
enum class ETestEnum : uint8
{
    EnumValue1,
    EnumValue2,
    EnumValue3
};
```

In the next section, we will look at the **TEnumAsByte** type.

TEnumAsByte

If you want to expose a variable to the engine that uses a raw enum, then you need to use the **TEnumAsByte** type. If you declare a **UPROPERTY** variable using a raw enum (not enum classes), you'll get a compilation error.

Have a look at the following example:

```
UENUM()
enum ETestRawEnum
{
    EnumValue1,
    EnumValue2,
    EnumValue3
};
```

If you declare a **UPROPERTY** variable using **ETestRawEnum**, such as the following:

```
UPROPERTY()
ETestRawEnum TestRawEnum;
```

You'll get this compilation error:

```
error : You cannot use the raw enum name as
a type for member variables, instead use
TEnumAsByte or a C++11 enum class with an
explicit underlying type.
```

To fix this error, you need to surround the enum type of the variable, which in this case is **ETestRawEnum**, with **TEnumAsByte<>**, like so:

```
UPROPERTY()
TEnumAsByte<ETestRawEnum> TestRawEnum;
```

UMETA

When you use the **UENUM** macro to add an enumeration to the Unreal Engine Reflection System, this will allow you to use the **UMETA** macro on each value of the enum. The **UMETA** macro, just like with other macros such as **UPROPERTY** or **UFUNCTION**, can use specifiers that will inform Unreal Engine 4 on how to handle that value. Here is a list of the most commonly used **UMETA** specifiers:

DISPLAYNAME

This specifier allows you to define a new name that is easier to read for the enum value when it's displayed in the editor.

Take a look at the following example:

```
UENUM()
enum class ETestEnum : uint8
{
```

```

    EnumValue1 UMETA(DisplayName = "My First
Option",
    EnumValue2 UMETA(DisplayName = "My Second
Option",
    EnumValue3 UMETA(DisplayName = "My Third
Option"
);

```

Let's declare the following variable:

```

UPROPERTY(EditDefaultsOnly,
BlueprintReadOnly, Category = "Test")
ETestEnum TestEnum;

```

Then, when you open the editor and look at the **TestEnum** variable, you will see a dropdown where **EnumValue1**, **EnumValue2**, and **EnumValue3** have been replaced by **My First Option**, **My Second Option**, and **My Third Option**, respectively.

HIDDEN

This specifier allows you to hide a specific enum value from the dropdown. This is typically used when there is an enum value that you only want to be able to use in C++ and not in the editor.

Take a look at the following example:

```

UENUM()
enum class ETestEnum : uint8
{
    EnumValue1 UMETA(DisplayName = "My First
Option"),

```

```

    EnumValue2 UMETA(Hidden),
    EnumValue3 UMETA(DisplayName = "My Third
Option")
};


```

Let's declare the following variable:

```

UPROPERTY(EditDefaultsOnly,
BlueprintReadOnly, Category = "Test")
ETestEnum TestEnum;

```

Then, when you open the editor and look at the **TestEnum** variable, you will see a dropdown. You should notice that **My Second Option** doesn't appear in the dropdown and therefore can't be selected.

Note

For more information on all of the UMETA specifiers, visit <https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Reference/Metadata/#enummetadataspecifiers>.

In the next section, we will look at the **BlueprintType** specifier for the **UENUM** macro.

BlueprintType

This **UENUM** specifier will expose the enumeration to blueprints. This means that there will be an entry for that enumeration on the dropdown that is used when making new variables or inputs/outputs for a function, as in the following example:

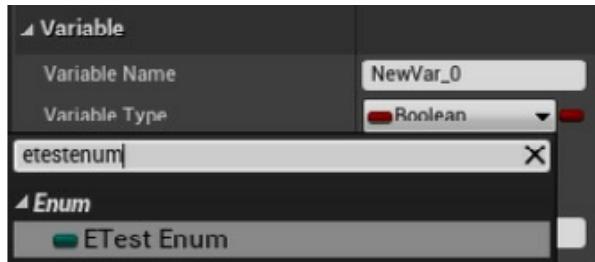


Figure 17.6: Setting a variable to use the ETestEnum variable type.

It will also show additional functions that you can call on the enumeration in the editor, as in this example:

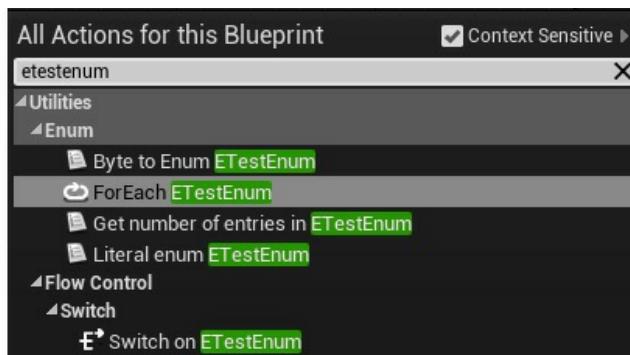


Figure 17.7: List of additional functions available when using BlueprintType

MAX

When using enumerations, it's common to want to know how many values it has. In Unreal Engine 4, the standard way of doing this is by adding **MAX** as the last value, which will be automatically hidden in the editor.

Take a look at the following example:

```
UENUM( )

enum class ETestEnum : uint8
{
```

```
    EnumValue1,  
    EnumValue2,  
    EnumValue3,  
    MAX  
};
```

If you want to know how many values **ETestEnum** has in C++, you just need to do the following:

```
const int32 MaxCount =  
(int32)ETestEnum::MAX;
```

This works because enumerations in C++ are internally stored as numbers, where the first value is **0**, the second is **1**, and so on. This means that as long as **MAX** is the last value, it will always have the total number of values in the enumeration. An important thing to take into consideration is that in order for **MAX** to give you the correct value, you cannot change the internal numbering order of the enumeration, like so:

```
UENUM()  
  
enum class ETestEnum : uint8  
{  
    EnumValue1 = 4,  
    EnumValue2 = 78,  
    EnumValue3 = 100,  
    MAX  
};
```

In this case, **MAX** will be **101** because it will use the number immediately next to the previous value, which is **EnumValue3 = 100**.

Using **MAX** is only meant to be used in C++ and not in the

editor, because the **MAX** value is hidden in blueprints, as previously mentioned. To get the number of entries of an enumeration in blueprints, you should use the **BlueprintType** specifier in the **UENUM** macro in order to expose some useful functions on the context menu. After that, you just need to type the name of your enumeration in the context menu. If you select the **Get number of entries in ETestEnum** option, you will have a function that returns the number of entries of an enumeration.

In the next exercise, you will be using C++ enumerations in the Unreal Engine 4 editor.

Exercise 17.02: Using C++ Enumerations in the Unreal Engine 4 Editor

In this exercise, we're going to create a new C++ project that uses the **Third Person** template, and we're going to add the following:

- An enumeration called **EWeaponType** containing **3** weapons – a pistol, shotgun, and rocket launcher.
- An enumeration called **EAmmoType** containing **3** ammo types – bullets, shells, and rockets.
- A variable called **Weapon** that uses **EWeaponType** to tell the type of the

current weapon.

- An integer array variable called **Ammo** that holds the amount of ammo for each type, which is initialized with the value **10**.
- When the player presses the *1*, *2*, and *3* keys, it will set **Weapon** to **Pistol**, **Shotgun**, and **Rocket Launcher**, respectively.
- When the player presses the *Left Mouse Button*, this will consume ammo from the current weapon.
- With every **Tick** function call, the character will display the current weapon type and the equivalent ammo type and amount.

The following steps will help you complete the exercise:

1. Create a new **Third Person** template project using **C++** called **Enumerations** and save it to a location of your choosing.

Once the project has been created, it should open the editor as well as the Visual Studio solution.

2. Close the editor and go back to Visual Studio.
3. Open the **Enumerations.h** file.
4. Create a macro called **ENUM_TO_INT32** that will convert an enumeration to an **int32** datatype:

```
#define  
ENUM_TO_INT32(Value)  
(int32)Value
```

5. Create a macro called **ENUM_TO_FSTRING** that will get the display name for a value of an **enum** datatype and convert it to an **FString** datatype:

```
#define  
ENUM_TO_FSTRING(Enum,  
Value) FindObject<UEnum>  
(ANY_PACKAGE, TEXT(Enum),  
true)->GetDisplayNameTextByIndex(  
(int32)Value).ToString()
```

6. Declare the enumerations **EWeaponType** and **EAmmoType**:

```
UENUM(BlueprintType)
```

```
enum class EWeaponType :  
    uint8  
{  
    Pistol UMETA(Display Name  
= «Glock 19»),  
    Shotgun UMETA(Display  
Name = «Winchester M1897»),  
    RocketLauncher  
UMETA(Display Name =  
«RPG»),  
    MAX  
};  
UENUM(BlueprintType)  
enum class EAmmoType :  
    uint8  
{  
    Bullets UMETA(DisplayName  
= «9mm Bullets»),  
    Shells UMETA(Display Name  
= «12 Gauge Shotgun  
Shells»),  
    Rockets UMETA(Display  
Name = «RPG Rockets»),
```

```
MAX
```

```
};
```

7. Open the **EnumerationsCharacter.h** file, include the **Enumerations.h** header:

```
#include "Enumerations.h"
```

8. Declare the protected **Weapon** variable that holds the weapon type of the selected weapon:

```
UPROPERTY(BlueprintReadOnly  
, Category = "Enumerations  
Character")
```

```
EWeaponType Weapon;
```

9. Declare the protected **Ammo** array that holds the amount of ammo for each type:

```
UPROPERTY(EditDefaultsOnly,  
BlueprintReadOnly, Category  
= "Enumerations Character")
```

```
TArray<int32> Ammo;
```

10. Declare the protected overrides for the **Begin Play** and **Tick** functions:

```
virtual void BeginPlay()
```

```
override;  
  
virtual void Tick(float  
DeltaSeconds) override;
```

11. Declare the protected input functions:

```
void OnPressedPistol();  
  
void OnPressedShotgun();  
  
void  
OnPressedRocketLauncher();  
  
void OnPressedFire();
```

12. Open the **EnumerationsCharacter.cpp** file, include the **DrawDebugHelpers.h** header:

```
#include  
"DrawDebugHelpers.h"
```

13. Bind the new action bindings at the end of the **SetupPlayerInputController** function, as shown in the following code snippet:

```
PlayerInputComponent-  
>BindAction("Pistol",  
IE_Pressed, this,  
&AEnumerationsCharacter::On
```

```

    PressedPistol);

    PlayerInputComponent-
        >BindAction("Shotgun",
    IE_Pressed, this,
    &AEnumerationsCharacter::On
    PressedShotgun);

    PlayerInputComponent-
        >BindAction("Rocket
    Launcher", IE_Pressed,
    this,
    &AEnumerationsCharacter::On
    PressedRocketLauncher);

    PlayerInputComponent-
        >BindAction("Fire",
    IE_Pressed, this,
    &AEnumerationsCharacter::On
    PressedFire);

```

14. Next, implement the override for **BeginPlay** that executes the parent logic, but also initializes the size of the **Ammo** array with the number of entries in the **EAmmoType** enumeration. Each position in the array will also be initialized with a value of **10**:

```

void
AEnumerationsCharacter::Beg

```

```
inPlay()
{
    Super::BeginPlay();

    const int32 AmmoCount =
        ENUM_TO_INT32(EAmmoType::MA
X);

    Ammo.Init(10, AmmoCount);

}
```

15. Implement the override for **Tick**:

```
void
AEnumerationsCharacter::Tic
k(float DeltaSeconds)

{
    Super::Tick(DeltaSeconds)
;
}
```

16. Convert the **Weapon** variable to **int32** and the **Weapon** variable to an **FString**:

```
const int32 WeaponIndex =
    ENUM_TO_INT32(Weapon);

const FString WeaponString
=
```

```
ENUM_TO_FSTRING( "EWeaponType", Weapon);
```

17. Convert the ammo type to an **FString** and get the ammo count for the current weapon:

```
const FString  
AmmoTypeString =  
ENUM_TO_FSTRING("EAmmoType"  
, Weapon);  
  
const int32 AmmoCount =  
Ammo[WeaponIndex];
```

We use **Weapon** to get the ammo type string because the entries in **EAmmoType** match the type of ammo of the equivalent **EWeaponType**. In other words, **Pistol = 0** uses **Bullets = 0**, **Shotgun = 1** uses **Shells = 1**, and **RocketLauncher = 2** uses **Rockets = 2**, so it's a 1-to-1 mapping that we can use in our favor.

18. Display in the character's location the name of the current weapon and its corresponding ammo type and ammo count, as shown in the following code snippet:

```

const FString String =
FString::Printf(TEXT("Weapo
n = %s\nAmmo Type =
%s\nAmmo Count = %d"),
*WeaponString,
*AmmoTypeString,
AmmoCount);

DrawDebugString(GetWorld(),
GetActorLocation(), String,
nullptr, FColor::White,
0.0f, true);

```

19. Implement the equip input functions that sets the **Weapon** variable with the corresponding value:

```

void
AEnumerationsCharacter::OnP
ressedPistol()

{
    Weapon =
EWeaponType::Pistol;

}

void
AEnumerationsCharacter::OnP
ressedShotgun( )

{

```

```

    Weapon =
EWeaponType::Shotgun;

}

void
AEnumerationsCharacter::OnP
ressedRocketLauncher()

{
    Weapon =
EWeaponType::RocketLauncher
;

}

```

20. Implement the fire input function that will use the weapon index to get the corresponding ammo type count and subtract **1**, as long as the resulting value is greater or equal to **0**:

```

void
AEnumerationsCharacter::OnP
ressedFire()

{
    const int32 WeaponIndex =
ENUM_TO_INT32(Weapon);

    const int32
NewRawAmmoCount =

```

```

Ammo[WeaponIndex] - 1;

    const int32 NewAmmoCount
    =
    FMath::Max(NewRawAmmoCount,
    0);

    Ammo[WeaponIndex] =
    NewAmmoCount;

}

```

21. Compile the code and run the editor.
22. Go to **Project Settings** and then **Engine**, then **Input**, and add the new action **bindings**:

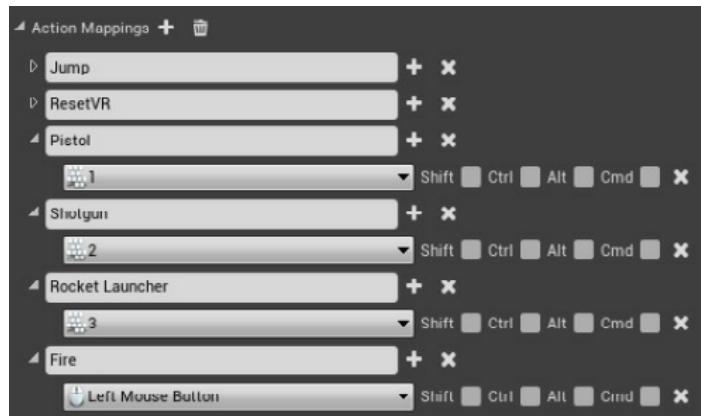


Figure 17.8: Adding the Pistol, Shotgun, Rocket Launcher, and Fire bindings

23. Close the **Project Settings**.
24. Play in **New Editor Window (PIE)**

in single-player mode (one client and dedicated server disabled):



Figure 17.9: The end result of the exercise

By completing this exercise, you will be able to use the `1`, `2`, and `3` keys to select the current weapon. You'll notice that every tick will display the type of the current weapon and its corresponding ammo type and ammo count. If you press the fire key, this will deduct the ammo count for the current weapon, but it will never go below `0`.

In the next section, you will be looking at bi-directional circular array indexing.

Bi-Directional Circular Array Indexing

Sometimes, when you use arrays to store information, you might want to iterate it in a bi-directional circular fashion. An example of this is the previous/next weapon logic in shooter games, where you have an array with weapons and you want to be able to cycle through them in a particular direction, and when you reach the first or the last index, you want to loop back around to the last and first index, respectively. The typical way of doing this example would be the following:

```

AWeapon * APlayer::GetPreviousWeapon()
{
    if(WeaponIndex - 1 < 0)
    {
        WeaponIndex = Weapons.Num() - 1;
    }
    else WeaponIndex--;
    return Weapons[WeaponIndex];
}

AWeapon * APlayer::GetNextWeapon()
{
    if(WeaponIndex + 1 > Weapons.Num() - 1)
    {
        WeaponIndex = 0;
    }
    else WeaponIndex++;
    return Weapons[WeaponIndex];
}

```

In the preceding code, we adjust the weapon index to loop back if the new weapon index is outside the limits of the weapons array, which can happen in two cases. The first case is when the player has the last weapon of the inventory equipped and asks for the next weapon. In this case, it should go back to the first weapon.

The second case is when the player has the first weapon of the inventory equipped and asks for the previous weapon. In this case, it should go to the last weapon.

While the example code works, it's still quite a lot of code to

solve such a trivial problem. To improve this code, there is a mathematical formula that will help you contemplate these two cases automatically in just one function. It's called the modulo (represented in C++ by the % operator), which gives you the remainder of a division between two numbers.

So how do we use the modulo to do a bi-directional circular array indexing? Let's rewrite the previous example using the modulo:

```
AWeapon * APlayer::GetNewWeapon(int32  
Direction)  
{  
    const int32 WeaponCount = Weapons.Num();  
  
    const int32 NewIndex = WeaponIndex +  
Direction;  
  
    const int32 ClampedNewIndex = NewIndex %  
WeaponCount;  
  
    WeaponIndex = (ClampedNewIndex +  
WeaponCount) % WeaponCount;  
  
    return Weapons[WeaponIndex];  
}
```

This is the new version, and you can tell right away that it's a bit harder to understand, but it's more functional and compact. If you don't use the variables to store the intermediate values of each operation, you can probably make the entire function in one or two lines of code.

Let's break down the preceding code snippet:

const int WeaponCount = Weapons.Num(): We need to know the size of the array to determine the index where it should circle back to **0**. In other words, if **WeaponCount = 4**, then the array has the indexes **0, 1, 2**, and **3**, so that tells us that index **4** is the cutoff index where it should go back to **0**.

```
const int32 NewIndex = WeaponIndex +
```

Direction: This is the new raw index without clamping it to the limits of the array. The **Direction** variable is used to indicate the offset we want to navigate the array, and this is either **-1** if we want the previous index or **1** if we want the next index.

```
const int32 ClampedNewIndex = NewIndex %
```

WeaponCount: This will make sure that **NewIndex** is within the **0** to **WeaponCount - 1** interval due to the modulo properties.

If **Direction** was always **1**, then **ClampedNewIndex** would be enough for what we need. The problem is, the modulo operation doesn't work very well with negative values, which can happen when **WeaponIndex** is **0** and **Direction** is **-1**, which would cause **NewIndex** to be **-1**. To fix this limitation, we need to do some additional calculations.

```
WeaponIndex = (ClampedNewIndex + WeaponCount)
```

% WeaponCount: This will add **WeaponCount** to **ClampedNewIndex** to make it positive and apply the modulo again to get the correct clamped index, which fixes the problem.

return Weapons[WeaponIndex]: This returns the weapon in the calculated **WeaponIndex** index position.

Let's take a look at a practical example to help you visualize how all this works:

Weapons =

- [0] Knife
- [1] Pistol
- [2] Shotgun

- [3] Rocket Launcher

WeaponCount = **Weapons**.**Num()** = 4.

Let's assume that **WeaponIndex** = 3 and **Direction** = 1.

Then:

NewIndex = *WeaponIndex* + *Direction* = 3 + 1 = 4

ClampedIndex = *NewIndex* % *WeaponCount* = 4 % 4 = 0

WeaponIndex = (*ClampedIndex* + *WeaponCount*) %

WeaponCount = (0 + 4) % 4 = 0

In this example, the starting value for the weapon index is 3 (which is the Rocket Launcher), and we want the next weapon (since **Direction** is 1). Performing the calculations,

WeaponIndex will now be 0 (which is the Knife). This is the desired behavior because we have 4 weapons, so we circled back to In this case, since **Direction** is 1, we could've just used **ClampedIndex** without doing the extra calculations.

Let's debug it again using different values.

Let's assume that **WeaponIndex** = 0 and **Direction** = -1:

NewIndex = *WeaponIndex* + *Direction* = 0 + -1 = -1

ClampedIndex = *NewIndex* % *WeaponCount* = -1 % 4 = -1

WeaponIndex = (*ClampedIndex* + *WeaponCount*) %

WeaponCount = (-1 + 4) % 4 = 3

In this example, the starting value for the weapon index is 0 (which is the Knife), and we want the previous weapon (since **Direction** is -1). Doing the calculations, **WeaponIndex** will

now be 3 (which is the Rocket Launcher). This is the desired behavior because we have 4 weapons, so we circled back to 3. In this specific case, **NewIndex** is negative, so we can't just use **ClampedIndex**; we need to do the extra calculation to get the correct value.

Exercise 17.03: Using Bi-Directional Circular Array Indexing to Cycle between an Enumeration

In this exercise, we're going to use the project from *Exercise 17.02, Using C++ Enumerations in the Unreal Engine 4 Editor*, and add two new action mappings for cycling the weapons. The mouse wheel up will go to the previous weapon type, and the mouse wheel down will go to the next weapon type.

The following steps will help you complete the exercise:

1. First, open the Visual Studio project from *Exercise 17.02, Using C++ Enumerations in the Unreal Engine 4 Editor*.

Next, you will be updating **Enumerations.h** and add a macro that will handle the bi-directional array cycling in a very convenient way, as shown in the following steps.

2. Open **Enumerations.h** and add the **GET_CIRCULAR_ARRAY_INDEX** macro that will apply the modulo formula that we've already covered previously:

```
#define  
GET_CIRCULAR_ARRAY_INDEX(Index,  
                           Count) (Index % Count  
                           + Count) % Count
```

3. Open **EnumerationsCharacter.h** and declare the new input functions for the weapon cycling:

```
void  
OnPressedPreviousWeapon();  
  
void OnPressedNextWeapon();
```

4. Declare the **CycleWeapons** function, as shown in the following code snippet:

```
void CycleWeapons(int32  
                  Direction);
```

5. Open **EnumerationsCharacter.cpp** and bind the new action bindings in the **SetupPlayerInputController** function:

```
PlayerInputComponent->BindAction("Previous Weapon", IE_Pressed, this, &AEnumerationsCharacter::OnPressedPreviousWeapon);

PlayerInputComponent->BindAction("Next Weapon", IE_Pressed, this, &AEnumerationsCharacter::OnPressedNextWeapon);
```

6. Now, implement the new input functions, as shown in the following code snippet:

```
void AEnumerationsCharacter::OnPressedPreviousWeapon()

{
    CycleWeapons(-1);
}

void AEnumerationsCharacter::OnPressedNextWeapon()

{
    CycleWeapons(1);
}
```

```
}
```

In the preceding code snippet, we define the functions that handle the action mappings for **Previous Weapon** and **Next Weapon**. Each function uses the **CycleWeapons** function, with a direction of **-1** for the previous weapon and **1** for the next weapon.

7. Implement the **CycleWeapons** functions, which does the bi-directional cycling using the **Direction** parameter based on the current weapon index:

```
void  
AEnumerationsCharacter::Cyc  
leWeapons(int32 Direction)  
{  
    const int32 WeaponIndex =  
        ENUM_TO_INT32(Weapon);  
  
    const int32 AmmoCount =  
        Ammo.Num();  
  
    const int32  
    NextRawWeaponIndex =  
        WeaponIndex + Direction;
```

```

        const int32
        NextWeaponIndex =
        GET_CIRCULAR_ARRAY_INDEX(Ne
        xtRawWeaponIndex ,
        AmmoCount);

        Weapon =
        (WeaponType)NextWeaponInde
        x;

    }

```

In the preceding code snippet, we implement the **CycleWeapons** function that uses the modulo operator to calculate the next valid weapon index based on the direction supplied.

8. Compile the code and run the editor.
9. Go to **Project Settings** and then to **Engine**, then **Input**, and add the new action **bindings**:

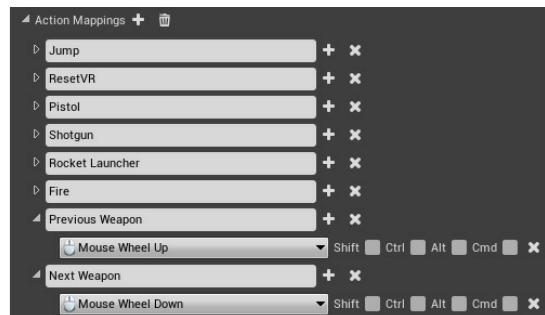


Figure 17.10: Adding the Previous Weapon and Next Weapon

bindings

10. Close the **Project Settings**.

11. Now, play in **New Editor Window (PIE)** in single-player mode (one client and dedicated server disabled):



Figure 17.11: The end result of the exercise

By completing this exercise, you will be able to use the mouse wheel to cycle between the weapons. If you select the rocket launcher and use the mouse wheel down to go to the next weapon, it will go back to the pistol. If you use the mouse wheel down to go to the previous weapon with the pistol selected, it will go back to the rocket launcher.

In the next activity, you will be adding the concept of weapons and ammo to the multiplayer FPS project we started in *Chapter 16, Multiplayer Basics*.

Activity 17.01: Adding Weapons and Ammo to

the Multiplayer FPS Game

In this activity, you'll add the concept of weapons and ammo to the Multiplayer FPS project we embarked on in the previous chapter activity. You will need to use the different types of RPCs covered in this chapter to complete this activity.

The following steps will help you complete this activity:

1. Open the **MultiplayerFPS** project from *Activity 16.01, Creating a Character for the Multiplayer FPS Project.*
2. Create a new **AnimMontage** slot called **Upper Body**.
3. Import the animations (**Pistol_Fire.fbx**, **MachineGun_Fire.fbx**, and **Railgun_Fire.fbx**) from the **Activity17.01\Assets** folder to **Content\Player\Animations**.

Note

*The Assets folder, **Activity17.01\Assets**, can be found on our GitHub repository at*

<https://packt.live/2It4Plb>.

4. Create an anim montage for **Pistol_Fire**, **MachineGun_Fire**, and **Railgun_Fire** and make sure they have the following configurations:

Pistol_Fire_Montage: The **Blend In** time of **0 . 01**, the **Blend Out** time of **0 . 1**, and make sure it uses the **Upper Body** slot.

MachineGun_Fire_Montage: The **Blend In** time of **0 . 01**, the **Blend Out** time of **0 . 1**, and make sure it uses the Upper Body slot.

Railgun_Fire_Montage: Make sure it uses the **Upper Body** slot.

5. Import **SK_Weapon.fbx**, **NoAmmo.wav**, **WeaponChange.wav**, and **Hit.wav** from the **Activity17.01\Assets** folder to **Content\Weapons**.

6. Import **Pistol_Fire_Sound.wav** from **Activity17.01\Assets** to **Content\Weapons\Pistol** and use it on an **AnimNotify Play Sound** in

the **Pistol_Fire** animation.

7. Create a simple green-colored material called **M_Pistol** and place it on **Content\Weapons\Pistol**.
8. Import **MachineGun_Fire_Sound.wav** from **Activity17.01\Assets** to **Content\Weapons\MachineGun** and use it on an **AnimNotify** Play Sound in the **MachineGun_Fire** animation.
9. Create a simple red-colored material called **M_MachineGun** and place it on **Content\Weapons\MachineGun**.
10. Import **Railgun_Fire_Sound.wav** from **Activity17.01\Assets** to **Content\Weapons\Railgun** and use it on an **AnimNotify** Play Sound in the **Railgun_Fire** animation.
11. Create a simple white-colored material called **M_Railgun** and place it on **Content\Weapons\Railgun**.
12. Edit the **SK_Mannequin** skeletal mesh

and create a socket called **GripPoint** from **hand_r** with Relative Location ($X=-10.403845, Y=6.0, Z=-3.124871$) and Relative Rotation ($X=0.0, Y=0.0, Z=90.0$).

13. Add the following input mappings in **Project Settings**, using the knowledge acquired in *Chapter 4, Player Input*:
 1. Fire (Action mapping): Left Mouse Button
 2. Previous Weapon (Action mapping): Mouse Wheel Up
 3. Next Weapon (Action mapping): Mouse Wheel Down
 4. Pistol (Action mapping): 1
 5. Machine Gun (Action mapping): 2
 6. Railgun (Action mapping): 3
14. In **MultiplayerFPS.h**, create the **ENUM_TO_INT32(Enum)** macro, which casts an enumeration to **int32** and

**GET_CIRCULAR_ARRAY_INDEX(Ind
ex, Count)** that uses bi-directional
circular array indexing to convert the
index into an index that is within the
interval of **0** and a count of **-1**.

15. Create a header file called **EnumTypes.h**, which holds the following enumerations:

EWeaponType: Pistol, MachineGun,
Railgun, MAX

EWeaponFireMode: Single,
Automatic

EammoType: Bullets, Slugs, MAX

16. Create a C++ class **Weapon** that derives from the **Actor** class that has a skeletal mesh component called **Mesh** as the root component. In terms of variables, it stores the name, the weapon type, the ammo type, the fire mode, how far the hitscan goes, how much damage the hitscan does when it hits, the fire rate, the animation montage to use when firing, and the sound to play when it has no ammo. In terms of functionality, it needs to be

able to start the fire (and also stop the fire, because of the automatic fire mode), which checks whether the player can fire. If it can, then it plays the fire animation in all of the clients and shoots a line trace in the camera position and direction with the supplied length to damage the actor it hits. If it doesn't have ammo, it will play a sound only on the owning client.

17. Edit **FPSCharacter** to support the new mappings for **Fire**, **Previous/Next Weapon**, **Pistol**, **Machine Gun**, and **Railgun**. In terms of variables, it needs to store the amount of ammo for each type, the currently equipped weapon, all of the weapons classes and spawned instances, the sound to play when it hits another player, and the sound when it changes weapons. In terms of functions, it needs to be able to equip/cycle/add weapons, manage ammo (add, remove, and get), handle when the character is damaged, play an anim montage on all of the clients, and play a sound on the owning client.

18. Create **BP_Pistol** from **AWeapon**,
place it on
Content\Weapons\Pistol, and
configure it with the following values:

1. Skeletal Mesh:

**Content\Weapons\SK_We
apon**

2. Material:

**Content\Weapons\Pisto
l\M_Pistol**

3. Name: **Pistol Mk I**

4. Weapon Type: **Pistol**,

Ammo Type: **Bullets**, Fire
Mode: **Automatic**

5. Hit Scan Range: **9999.9**, Hit

Scan Damage: **5.0**, Fire

Rate: **0.5**

6. Fire Anim Montage:

**Content\Player\Animat
ions\Pistol_Fire_Mont
age**

7. NoAmmoSound:

**Content\Weapons\NoAmm
o**

19. Create **BP_MachineGun** from **AWeapon** and place it on **Content\Weapons\MachineGun** and configure it with the following values:

1. Skeletal Mesh:

Content\Weapons\SK_Weapon

2. Material:

Content\Weapons\MachineGun\M_MachineGun

3. Name: **Machine Gun Mk I**

4. Weapon Type: **Machine Gun**, Ammo Type: **Bullets**, Fire Mode: **Automatic**

5. Hit Scan Range: **9999.9**, Hit Scan Damage: **5.0**, Fire Rate: **0.1**

6. Fire Anim Montage:

Content\Player\Animations\MachineGun_Fire_Montage

7. NoAmmoSound:

Content\Weapons\NoAmm

o

20. Create **BP_Railgun** from **AWeapon** and place it on **Content\Weapons\Railgun** and configure it with the following values:

1. Skeletal Mesh:

Content\Weapons\SK_Weapon

2. Material:

Content\Weapons\Railgun\M_Railgun

3. Name: Railgun Mk I,

Weapon Type: **Railgun**,

AmmoType: **Slugs**, Fire

Mode: **Single**

4. Hit Scan Range: **9999.9**, Hit

Scan Damage: **100.0**, Fire

Rate: **1.5**

5. Fire Anim Montage:

Content\Player\Animations\Railgun_Fire_Montage

6. No Ammo Sound:

Content\Weapons\NoAmm

0

21. Configure **BP_Player** with the following values:

1. Weapon Classes (Index 0:
BP_Pistol, Index 1:
BP_MachineGun, Index 2:
BP_Railgun).
2. Hit Sound:
Content\Weapons\Hit.
3. Weapon Change Sound:
Content\Weapons\WeaponChange.
4. Make the mesh component block the visibility channel so it can be hit by the hitscans of the weapons.
5. Edit **ABP_Player** to use a **Layered Blend Per Bone** node, with **Mesh Space Rotation Blend** enabled, on the **spine_01** bone so that the upper body animations use the Upper Body slot.

6. Edit **UI_HUD** so that it displays a white dot crosshair in the middle of the screen and the current weapon and ammo count under the Health and Armor indicators:

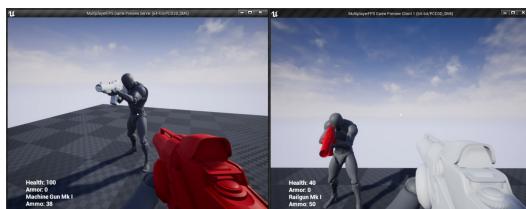


Figure 17.12: The expected result of the activity

The result should be a project where each client will have weapons with ammo and will be able to use them to fire at and damage other players. You will also be able to select weapons by using the **1**, **2**, and **3** keys and by using the mouse wheel up and down to select the previous and next weapon.

Note

The solution to this activity can be found at:

<https://packt.live/338jEBx>.

Summary

In this chapter, you learned how to use RPCs to allow the server and the clients to execute logic on one another. We also learned how enumerations work in Unreal Engine 4 by using the **UENUM** macro and how to use bi-directional circular array indexing, which helps you iterate an array in both directions and loops around when you go beyond its index limits.

With the activity of this chapter complete, you'll have a basic playable game where players can shoot each other and switch weapons, but there is still more we can add to make it even more interesting.

In the next chapter, we'll learn where the instances of the most common gameplay framework classes exist in multiplayer, as well as learn about the Player State and Game State classes, which we haven't covered yet. We'll also cover some new concepts in the game mode that are used in multiplayer matches, as well as some useful general-purpose, built-in functionality.

18. Gameplay Framework Classes in Multiplayer

Overview

In this chapter, you will learn where the instances of the gameplay framework classes exist in multiplayer. You'll also learn how to use the game state and player state classes, as well as some new concepts in the game mode, including match states. We'll also cover some useful built-in functionality that can be used in different types of games.

By the end of this chapter, you'll be able to use the game state and player state classes to store information about the game and about specific players that can be accessed by any client. You'll also know how to make the most of the game mode class and other related functionality.

Introduction

In the previous chapter, we covered *Remote Procedure Calls*, which allow the server and the clients to execute remote functions on each other. We also covered enumerations and *Bi-Directional Circular Array Indexing*.

In this chapter, we're going to take a look at the most common gameplay framework classes and see where their instances exist in a multiplayer environment. This is important to

understand, so you know which instances can be accessed in a specific game instance. An example of this would be that only the server should be able to access the game mode instance, so if you were playing Fortnite, a player shouldn't be able to access it and modify the rules of the game.

We'll also be covering the game state and player state classes in this chapter. As the name implies, these store information about the state of the game and of each player that is playing the game. Finally, toward the end of this book, we'll cover some new concepts in the game mode, as well as some useful built-in functionality.

We will begin with how gameplay framework classes work in multiplayer.

Gameplay Framework Classes in Multiplayer

Unreal Engine 4 comes with a gameplay framework, which is a set of classes that allow you to create games more easily. The gameplay framework does this by providing built-in common functionality that is present in most games, such as a way to define the game rules (game mode), and a way to control a character (the player controller and pawn/character class).

When an instance of a gameplay framework class is created in a multiplayer environment, it can exist on the server, on the clients, and on the owning client, which is the client that has its player controller as the owner of that instance. This means that the instances of the gameplay framework classes will always fall into one of the following categories:

- **Server Only:** The instances of the class will only exist on the server.

- **Server and Clients:** The instances of the class will exist on the server and the clients.
- **Server and Owning Client:** The instances of the class will exist on the server and on the owning client.
- **Owning Client Only:** The instances of the class will only exist on the owning client.

Take a look at the following diagram, which shows each category and the purpose of the most common classes on the gameplay framework:

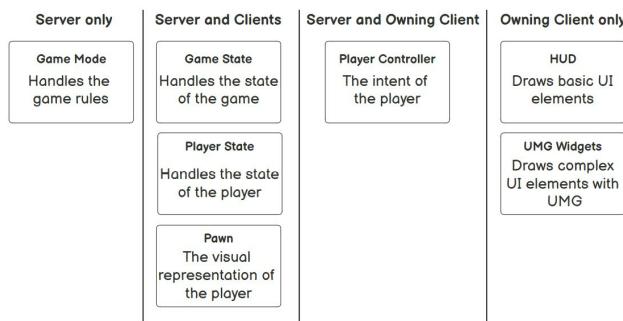


Figure 18.1: The most common gameplay framework classes divided into categories

Let's go into a bit more detail on each class in the preceding diagram:

- **Game Mode (Server Only):** The game mode class defines the rules of the game and its instance can only be accessed by the server. If a client tries

to access it, the instance will always be invalid, in order to prevent clients from changing the game rules.

- **Game State (Server and Clients):**

The game state class stores the state of the game and its instance can be accessed both by the server and the clients. The game state will be covered in greater depth in a future topic.

- **Player State (Server and Clients):**

The player state class stores the state of a player and its instance can be accessed both by the server and the clients. The player state will be covered in greater depth in a future topic.

- **Pawn (Server and Clients):** The pawn class is the visual representation of a player and its instance can be accessed by the server and the clients.

- **PlayerController (Server and Owning Client):** The player controller class represents the intent of a player, which is relayed to the currently possessed pawn, and its instance can only be accessed on the server and the owning client. For security reasons, clients can't access

other client's player controllers, so they should use the server to communicate.

If a client calls the

UGameplayStatics::GetPlayerController function with an index other than **0** (which would return its player controller), the returned instance will always be invalid. This means that the server is the only place that has access to all of the player controllers. You can find out whether a player controller instance is in its owning client by calling the **AController::IsLocalController** function.

- **HUD (Owning Client Only):** The HUD class is used as an immediate mode to draw basic shapes and text on the screen. Since it's used for UI, its instance is only available on the owning client, because the server and the other clients don't need to know about it.
- **UMG Widgets (Owning Client Only):** The UMG widget classes are used to display complex UI on the screen. Since it's used for UI, its

instance is only available on the owning client, because the server and the other clients don't need to know about it.

To help you understand these concepts, we can use Dota 2 as an example. The game mode defines that there are different phases of the game (*pre-game for hero picking, the actual game, and the post-game with the winner*) and that the end goal is to destroy the other team's ancient. Since it's a class that is critical to gameplay, clients can't be allowed to access it:

- The game state stores the elapsed time, whether it's day or night, the score of each team, and so on, so the server and the clients need to be able to access it.
- The player state stores the name, the hero selected, and the kill/death/assist ratio of a player, so the server and the clients need to be able to access it.
- The pawn would be the hero, the courier, the illusions, and so on, controlled by the player, so the server and the clients need to be able to access it.
- The player controller is what relays the input information to the controlled pawn, so only the server and the

owning client need to be able to access it.

- The UI classes (**HUD** and **User** widget) would display all of the information on the owning client, so it only needs to be accessed there.

In the next exercise, you will be displaying the instance values of the most common gameplay framework classes.

Exercise 18.01: Displaying the Gameplay Framework Instance Values

In this exercise, we're going to create a new C++ project that uses the Third Person template, and we're going to add the following:

- On the owning client, the player controller creates and adds to the viewport a simple UMG widget that displays the name of the menu instance.
- On the tick function, the character displays the value of its own instance

(as a pawn) as well as whether it has a valid instance for the game mode, game state, player state, player controller, and HUD.

Note

If needed, you can refer back to Chapter 1, Unreal Engine Introduction, for a recap of the tick function.

The following steps will help you complete the exercise:

1. Create a new **Third Person** template project using **C++** called **GFIInstances** (*as in Gameplay Framework Instances*) and save it on a location of your choosing. Once the project has been created, it should open the editor as well as the Visual Studio solution.
2. In the editor, create a new **C++** class called **GFIInstancePlayerController** that is derived from **PlayerController**. Wait for the

compilation to end, close the editor, and then go back to Visual Studio.

3. Open the **GFIInstancesCharacter.h** file and declare the protected override for the **Tick** function:

```
virtual void Tick(float  
DeltaSeconds) override;
```

4. Open the **GFIInstancesCharacter.cpp** file and include **DrawDebugHelpers.h** and **PlayerController.h**:

```
#include  
"DrawDebugHelpers.h"  
  
#include  
"GameFramework/PlayerContro  
ller.h"
```

5. Implement the **Tick** function:

```
void  
AGFIInstancesCharacter::Tick  
(float DeltaSeconds)  
{  
    Super::Tick(DeltaSeconds)  
};
```

```
}
```

6. Get the instances for the game mode, game state, player controller, and HUD:

```
AGameModeBase* GameMode =
GetWorld()->GetAuthGameMode();

AGameStateBase* GameState =
GetWorld()->GetGameState();

APlayerController*
PlayerController =
Cast<APlayerController>
(GetController());

AHUD* HUD =
PlayerController != nullptr
? PlayerController->GetHUD() : nullptr;
```

In the preceding code snippet, we store the instances for the game mode, game state, player controller, and HUD in separate variables, so that we can check whether they are valid.

7. Create a string for each gameplay framework class:

```
const FString
```

```

GameModeString = GameMode
!= nullptr ? TEXT("Valid")
: TEXT("Invalid");

const FString
GameStateString = GameState
!= nullptr ? TEXT("Valid")
: TEXT("Invalid");

const FString
PlayerStateString =
GetPlayerState() != nullptr
? TEXT("Valid") :
TEXT("Invalid");

const FString PawnString =
GetName();

const FString
PlayerControllerString =
PlayerController != nullptr
? TEXT("Valid") :
TEXT("Invalid");

const FString HUDString =
HUD != nullptr ?
TEXT("Valid") :
TEXT("Invalid");

```

Here, we create strings to store the name of the pawn and whether the other gameplay framework instances

are valid.

8. Display each string on the screen:

```
const FString String =
FString::Printf(TEXT("Game
Mode = %s\nGame State =
%s\nPlayerState = %s\nPawn
= %s\nPlayer Controller =
%s\nHUD = %s"),
*GameModeString,
*GameStateString,
*PlayerStateString,
*PawnString,
*PlayerControllerString,
*HUDString);

DrawDebugString(GetWorld(),
GetActorLocation(), String,
nullptr, FColor::White,
0.0f, true);
```

In this code snippet, we print the strings created in the preceding code, which indicate the name of the pawn and whether the other gameplay framework instances are valid.

9. Before we can move on to the **AGFInstancesPlayerController** class, we need to tell Unreal Engine

that we want to use UMG functionality in order to be able to use the **UUserWidget** class. To do this, we need to open **GFIInstances.Build.cs** and add **UMG** to the **PublicDependencyModuleNames** string array, like so:

```
PublicDependencyModuleNames  
.AddRange(new string[] {  
    "Core", "CoreUObject",  
    "Engine", "InputCore",  
    "HeadMountedDisplay", "UMG"  
});
```

If you try to compile and get errors from adding the new module, then clean and recompile your project. If that doesn't work, try restarting your IDE.

10. Open **GFIInstancesPlayerController.h** and add the protected variables to create the UMG widget:

```
UPROPERTY(EditDefaultsOnly,  
BlueprintReadOnly, Category  
= "GF Instance Player"
```

```
Controller")  
TSubclassOf<UUserWidget>  
MenuClass;  
UPROPERTY()  
UUserWidget* Menu;
```

11. Declare the protected override for the **BeginPlay** function:

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

12. Open **GFIInstancesPlayerController.cpp** and include **UserWidget.h**:

```
#include  
"Blueprint/UserWidget.h"
```

13. Implement the **BeginPlay** function:

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

14. Abort the function if it's not the owning

client or if the menu class is invalid:

```
if (!IsLocalController() ||  
    MenuClass == nullptr)  
{  
    return;  
}  
}
```

15. Create the widget and add it to the viewport:

```
Menu =  
CreateWidget<UUserWidget>  
(this, MenuClass);  
  
if (Menu != nullptr)  
{  
    Menu->AddToViewport(0);  
}
```

16. Compile and run the code.

17. In **Content Browser**, go to the **Content** folder, create a new folder called **UI**, and open it.

18. Create a new widget blueprint called **UI_Menu** and open it.

19. Add a **Text Block** called **tbText** to the root canvas panel and set it to be a variable by clicking the checkbox **Is Variable** next to its name on the top of the Details panel.
20. Set **tbText** to have **Size To Content** to **true**.
21. Go to the **Graph** section and, in **Event Graph**, implement the **Event Construct** in the following manner:

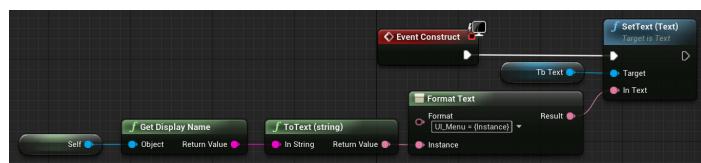


Figure 18.2: The Event Construct that displays the name of the UI_Menu instance

Note

You can find the preceding screenshot in full resolution for better viewing at the following link:

<https://packt.live/38wvSr5>.

22. Save and close **UI_Menu**.
23. Go to the **Content** folder and create a

blueprint called **BP_PlayerController** that is derived from **GFIInstancesPlayerController**.

24. Open **BP_PlayerController** and set the **Menu Class** to use **UI_Menu**.
25. Save and close **BP_PlayerController**.
26. Go to the **Content** folder and create a blueprint called **BP_GameMode** that is derived from **GFIInstancesGameMode**.
27. Open **BP_GameMode** and set the **Player Controller Class** to use **BP_PlayerController**.
28. Save and close **BP_GameMode**.
29. Go to **Project Settings** and pick **Maps & Modes** from the left panel, which is in the **Project** category.
30. Set the **Default GameMode** to use **BP_GameMode**.
31. Close **Project Settings**.

Finally, you can test the project.

32. Run the code and wait for the editor to fully load.
33. Go to **Multiplayer Options** and set the number of clients to **2**.
34. Set the window size to **800x600**.
35. Play in **New Editor Window (PIE)**.

Once you complete this exercise, you will be able to play on each client. You'll notice that the characters are displaying whether the instances for the game mode, game state, player state, player controller, and HUD are valid. It also displays the name of the instance of the pawn.

Now, let's analyze the values displayed in the **Server** and **Client 1** windows. Let's begin with the **Server** window first.

The Server Window

In the **Server** window, you have the values for **Server Character**, and in the background, you have the values for **Client 1 Character**. You should be able to see **Server Character**, **Client 1 Character**, and the **UI_Menu** UMG widget in the top-left corner. The UMG widget instance is only created for the player controller of **Server Character**, since it's the only player controller in this window that actually controls a character.

Let's first analyze the values for **Server Character**.

Server Character

This is the character that the listen server, which is a server that also has a client integrated that can play the game as well, is controlling. The values displayed on this character are as follows:

- **Game Mode = Valid** because the game mode instance only exists in the server, which is the current game instance.
- **Game State = Valid** because the game state instance exists on the clients and the server, which is the current game instance.
- **Player State = Valid** because the player state instance exists on the clients and the server, which is the current game instance.
- **Pawn = ThirdPersonCharacter_2** because pawn instances exist on the clients and the server, which is the current game instance.
- **Player Controller = Valid** because player controller instances exist on the

owning client and the server, which is the current game instance.

- **HUD = Valid** because HUD instances only exist on the owning client, which is the case.

Next, we are going to be looking at **Client 1 Character** in the same window.

Client 1 Character

This is the character that **Client 1** is controlling. The values displayed on this character are as follows:

- **Game Mode = Valid** because the game mode instance only exists in the server, which is the current game instance.
- **Game State = Valid** because the game state instance exists on the clients and the server, which is the current game instance.
- **Player State = Valid** because the player state instance exists on the clients and the server, which is the current game instance.

- **Pawn = ThirdPersonCharacter_o**
because pawn instances exist on the clients and the server, which is the current game instance.
- **Player Controller = Valid** because player controller instances exist on the owning client and the server, which is the current game instance.
- **HUD = Invalid** because HUD instances only exist on the owning client, which is not the case.

The Client 1 Window

In the **Client 1** Window, you have the values for **Client 1 Character** and, in the background, you have values for **Server Character**. You should see **Client 1 Character**, **Server Character**, and the **UI_Menu** UMG widget in the top-left corner. The UMG widget instance is only created for the player controller of **Client 1 Character**, since it's the only player controller in this window that actually controls a character.

Let's first analyze the values for **Client 1 Character**.

Client 1 Character

This is the character that **Client 1** is controlling. The values displayed on this character are as follows:

- **Game Mode = Invalid** because the game mode instance only exists in the server, which is not the current game instance.
- **Game State = Valid** because the game state instance exists on the server and the clients, which is the current game instance.
- **Player State = Valid** because the player state instance exists on the server and the clients, which is the current game instance.
- **Pawn = ThirdPersonCharacter_o** because pawn instances exist on the server and the clients, which is the current game instance.
- **Player Controller = Valid** because player controller instances exist on the server and the owning client, which is the current game instance.
- **HUD = Valid** because HUD instances only exist on the owning client, which is the case.

Next, we are going to be looking at **Server Character** in the same window.

Server Character

This is the character that the listen server is controlling. The values displayed on this character are as follows:

- **Game Mode = Invalid** because the game mode instance only exists in the server, which is not the current game instance.
- **Game State = Valid** because the game state instance exists on the server and the clients, which is the current game instance.
- **Player State = Valid** because the player state instance exists on the server and the clients, which is the current game instance.
- **Pawn = ThirdPersonCharacter_2** because pawn instances exist on the server and the clients, which is the current game instance.
- **Player Controller = Invalid** because player controller instances exist on the server and the owning client, which is not the current game instance.
- **HUD = Invalid** because HUD

instances only exist on the owning client, which is not the case.

By completing this exercise, you should have a better understanding of where each instance of the gameplay framework classes exists and where it doesn't. Next, we're going to cover the player state and game state classes, as well as some additional concepts on the game mode and useful built-in functionalities.

Game Mode, Player State, and Game State

So far, we've covered most of the important classes in the gameplay framework, including the game mode, player controller, and the pawn. In this chapter, we're going to cover the player state, game state, and some additional concepts of the game mode, as well as some useful built-in functionalities.

Game Mode

We've already talked about the game mode and how it works, but there are a few concepts that haven't yet been covered.

CONSTRUCTOR

To set the default class values, you can use the constructor like so:

```
ATestGameMode::ATestGameMode()
```

```

{
    DefaultPawnClass =
AMyCharacter::StaticClass();

    PlayerControllerClass =
AMyPlayerController::StaticClass();

    PlayerStateClass =
AMyPlayerState::StaticClass();

    GameStateClass =
AMyGameState::StaticClass();

}

```

The preceding code lets you specify which classes to use when spawning pawns, player controllers, player states, and game states when we are using this game mode.

Getting the Game Mode Instance

If you want to access the game mode instance, you need to get it from the **GetWorld** function by using the following code:

```

AGameModeBase* GameMode = GetWorld()-
>GetAuthGameMode();

```

The preceding code allows you to access the current game mode instance so you can run functions and consult the values of certain variables. You have to make sure to call it only on the server, since this will be invalid on the clients, for security reasons.

Match States

So far, we've only been using the **AGameModeBase** class, which is the most basic game mode class in the framework, and although it's more than enough for certain types of games, there are cases where you require a bit more functionality. An example of this would be if we wanted to do a lobby system, where the match only starts if all players have marked that they were ready. This example wouldn't be possible to do with

the **AGameModeBase** class. For these cases, it's better to use the **AGameMode** class instead, which is a child class of **AGameModeBase** that adds support to multiplayer matches through the use of match states. The way match states work is by using a state machine that can only be in one of the following states at a given time:

- **EnteringMap**: This is the starting state when the world is still loading and the actors aren't ticking yet. It will transition to the **WaitingToStart** state once the world finishes loading.
- **WaitingToStart**: This state is set when the world has finished loading and the actors are ticking, although the pawns for the players won't be spawned because the game hasn't started yet. When the state machine enters this state, it will call the **HandleMatchIsWaitingToStart** function. The state machine will transition to the **InProgress** state if the **ReadyToStartMatch** function returns **true** or if the **StartMatch** function is called somewhere in the code.
- **InProgress**: This state is where the actual game takes place. When the state machine enters this state, it will

spawn the pawns for the players, call **BeginPlay** on all of the actors in the world, and call the **HandleMatchHasStarted** function. The state machine will transition to the **WaitingPostMatch** state if the **ReadyToEndMatch** function returns **true** or if the **EndMatch** function is called somewhere in the code.

- **WaitingPostMatch:** This state is set when the match ends. When the state machine enters this state, it will call the **HandleMatchHasEnded** function. In this state, actors still tick, but new players cannot join. It will transition to the **LeavingMap** state when it starts unloading the world.
- **LeavingMap:** This state is set while it's unloading the world. When the state machine enters this state, it will call the **HandleLeavingMap** function. The state machine will transition to the **EnteringMap** state when it starts loading the new level.
- **Aborted:** This is a failure state that can only be set by calling the

AbortMatch function, which is used to flag that something went wrong that prevented the match from happening.

To help you understand these concepts better, we can use Dota 2 again as an example:

- **EnteringMap**: The state machine will be in this state when the map is loading.
- **WaitingToStart**: The state machine will be in this state once the map is loaded and the players are picking their heroes. The **ReadyToStartMatch** function will check whether all players have selected their heroes; if they have, then the match can start.
- **InProgress**: The state machine will be in this state when the game is actually underway. The players control their heroes to farm and fight against other players. The **ReadyToEndMatch** function will constantly check the health of each ancient to see whether one of them was destroyed; if it was, then the match ends.

- **WaitingPostMatch:** The state machine will be in this state when the game has ended and you are seeing the destroyed ancient and showing the final scores for each player.
- **LeavingMap:** The state machine will be in this state when it's unloading the map.
- **Aborted:** The state machine will be in this state if one of the players failed to connect in the initial stage, therefore aborting the whole match.

Respawning the Player

When the player dies and you want to respawn it, you typically have two options. The first option is to reuse the same pawn instance, manually reset its state back to the defaults, and teleport it to the respawn location. The second option is to destroy the pawn and spawn a new one, which will already have its state reset. If you prefer the latter option, then the **AGameModeBase::RestartPlayer** function handles the logic of spawning a new pawn instance for a certain player controller for you and places it on a player start.

One important thing to take into consideration is that the function spawns a new pawn instance only if the player controller doesn't already possess a pawn, so make sure to destroy the controlled pawn before calling **RestartPlayer**.

Take a look at the following example:

```
void
```

```

ATestGameMode::OnDeath(APlayerController*
VictimController)

{
    if(VictimController == nullptr)
    {
        return;
    }

    APawn* Pawn = VictimController-
>GetPawn();

    if(Pawn != nullptr)
    {
        Pawn->Destroy();
    }

    RestartPlayer(VicitimController);
}

```

In the preceding code, we have the **OnDeath** function that takes the player controller of the player that died, destroys its controlled pawn, and calls the **RestartPlayer** function to spawn a new instance on a player start. By default, the player start actor used will always be the same as the player spawned the first time. If you want the function to spawn on a random player start, then you need to override the **AGameModeBase::ShouldSpawnAtStartSpot** function and force it to **return false**, like so:

```

bool
ATestGameMode::ShouldSpawnAtStartSpot(ACont
roller* Player)

```

```
{  
    return false;  
}
```

The preceding code will make the game mode use a random player start instead of always using the same.

Note

For more information about the game mode, please visit <https://docs.unrealengine.com/en-US/Gameplay/Framework/GameMode/#gamemodes> and <https://docs.unrealengine.com/en-US/API/Runtime/Engine/GameFramework/AGameMode/index.html>.

Player State

The player state class stores the state of a player, such as the current score, kills/deaths, and coins picked up. It's mostly used in multiplayer mode to store the information that other clients need to know about the player, since they can't access its player controller. The most widely used built-in variables are **PlayerName**, **Score**, and **Ping**, which give you the name, score, and ping of the player, respectively.

A scoreboard entry on a multiplayer shooter is a good example of how to use the player state, because every client needs to know the names, kills/deaths, and pings of all the players. The player state instance can be accessed in the following ways:

AController::PlayerState

This variable has the player state associated with the controller and it can only be accessed by the server and the

owning client. The following example will demonstrate how to use the variable:

```
APlayerState* PlayerState = Controller->PlayerState;
```

AController::GetPlayerState()

This function returns the player state associated with the controller and it can only be accessed by the server and the owning client. This function also has a templated version, so you can cast it to your own custom player state class. The following examples will demonstrate how to use the default and template versions of this function:

```
// Default version

APlayerState* PlayerState = Controller->GetPlayerState();

// Template version

ATestPlayerState* MyPlayerState =
Controller->GetPlayerState<ATestPlayerState>();
```

APawn::GetPlayerState()

This function returns the player state associated with the controller that is possessing the pawn and it can be accessed by the server and the clients. This function also has a templated version, so you can cast it to your own custom player state class. The following examples will demonstrate how to use the default and template versions of this function:

```
// Default version

APlayerState* PlayerState = Pawn->GetPlayerState();

// Template version

ATestPlayerState* MyPlayerState = Pawn->GetPlayerState<ATestPlayerState>();
```

The preceding code demonstrates the two ways you can use the **GetPlayerState** function. You can use the default **APlayerState** version or the template version that casts automatically for you.

AGameState::PlayerArray

This variable stores the player state instances for each player and it can be accessed on the server and the clients. The following example will demonstrate how to use this variable:

```
TArray<APlayerState*> PlayerStates =  
GameState->PlayerArray;
```

To help you understand these concepts better, we can use Dota 2 again as an example. The player state would have at least the following variables:

Name: The name of the player

Hero: The selected hero

Health: The health of the hero

Mana: The mana of the hero

Stats: The hero stats

Level: The level the hero is currently in

Kill / Death / Assist: The kill/death/assist ratio for the player

Note

For more information about the player state, please visit <https://docs.unrealengine.com/en-US/API/Runtime/Engine/GameFramework/APlayerState/index.html>.

Game State

The game state class stores the state of the game, including the match's elapsed time and the score required to win the game. It's mostly used in multiplayer mode to store the information that other clients need to know about the game, since they can't access the game mode. The most widely used variable is **PlayerArray**, which is an array with the player state of every connected client. A scoreboard on a multiplayer shooter is a good example of how to use the game state, because every client needs to know how many kills are required to win and also the name and ping of each player.

The game state instance can be accessed in the following ways:

UWorld::GetGameState()

This function returns the game state associated with the world and can be accessed on the server and the clients. This function also has a templated version, so you can cast it to your own custom game state class. The following examples will demonstrate how to use the default and template versions of this function:

```
// Default version  
  
AGameStateBase* GameState = GetWorld()->GetGameState();  
  
// Template version  
  
AMyGameState* MyGameState = GetWorld()->GetGameState<AMyGameState>();
```

AGameModeBase::GameState

This variable has the game state associated with the game mode and it can only be accessed on the server. The following example will demonstrate how to use the variable:

```
AGameStateBase* GameState = GameMode->GameState;
```

AGameModeBase::GetGameState()

This function returns the game state associated with the game mode and it can only be accessed on the server. This function also has a templated version, so you can cast it to your own custom game state class. The following examples will demonstrate how to use the default and template versions of this function:

```
// Default version  
  
AGameStateBase* GameState = GameMode->GetGameState<AGameStateBase>();  
  
// Template version  
  
AMyGameState* MyGameState = GameMode->GetGameState<AMyGameState>();
```

To help you understand these concepts better, we can use Dota 2 again as an example. The game state would have the following variables:

Elapsed Time: How long the match has been going on for

Radiant Kills: How many Dire heroes the Radiant team has killed

Dire Kills: How many Radiant heroes the Dire team has killed

Day/Night Timer: Used to determine whether it is day or night

Note

For more information about the game state, please visit <https://docs.unrealengine.com/en-US/>

<US/Gameplay/Framework/GameMode/#gamestate> and
<https://docs.unrealengine.com/en-US/API/Runtime/Engine/GameFramework/AGameState/index.html>.

Useful Built-in Functionality

Unreal Engine 4 comes with a lot of useful functionality built in. Here are some examples of some functions and a component that will be useful when developing your game:

void AActor::EndPlay(const EEndPlayReason::Type EndPlayReason)

This function is called when the actor has stopped playing, which is the opposite of the **BeginPlay** function. You have the **EndPlayReason** parameter, which tells you why the actor stopped playing (if it was destroyed, if you stopped PIE, and so on). Take a look at the following example, which prints to the screen the fact that the actor has stopped playing:

```
void ATestActor::EndPlay(const
EEndPlayReason::Type EndPlayReason)

{
    Super::EndPlay(EndPlayReason);

    const FString String =
FString::Printf(TEXT("The actor %s has just
stopped playing"), *GetName());

    GEngine->AddOnScreenDebugMessage(-1,
2.0f, FColor::Red, String);
}
```

void ACharacter::Landed(const FHitResult& Hit)

This function is called when a player lands on a surface after being in the air. Take a look at the following example, which plays a sound when a player lands on a surface:

```
void ATestCharacter::Landed(const  
FHitResult& Hit)  
{  
    Super::Landed(Hit);  
  
    UGameplayStatics::PlaySound2D(GetWorld(),  
    LandSound);  
}  
  
bool UWorld::ServerTravel(const FString& FURL,  
bool bAbsolute, bool bShouldSkipGameNotify)
```

This function will make the server load a new map and bring all of the connected clients along with it. This is different from using other methods that load maps, such as the **UGameplayStatics::OpenLevel** function, because it won't bring the clients along; it will just load the map on the server and disconnect the clients.

One important thing to take into consideration is that server travel only works properly in the packaged version, so it won't bring the clients along when playing in the editor. Take a look at the following example, which gets the current map name and uses server travel to reload it and bring along the connected clients:

```
void ATestGameModeBase::RestartMap()  
{  
    const FString URL = GetWorld()->GetName();  
  
    GetWorld()->ServerTravel(URL, false,  
    false);  
}
```

void TArray::Sort(const PREDICATE_CLASS& Predicate)

The **TArray** data structure comes with the **Sort** function, which allows you to sort the values of an array by using a **lambda** function that returns whether value **A** should be ordered first, followed by value **B**. Take a look at the following example, which sorts an integer array from the smallest value to the highest:

```
void ATestActor::SortValues()
{
    TArray<int32> SortTest;
    SortTest.Add(43);
    SortTest.Add(1);
    SortTest.Add(23);
    SortTest.Add(8);
    SortTest.Sort([](const int32& A, const int32& B) { return A < B; });
}
```

The preceding code will sort the **SortTest** array with the values [43, 1, 23, 8] from smallest to highest [1, 8, 23, 43].

void AActor::FellOutOfWorld(const UDamageType& DmgType)

In Unreal Engine 4, there is a concept called **Kill Z**, which is a plane on a certain value in **Z** (set in the **World Settings** panel), and if an actor goes below that **Z** value, it will call the **FellOutOfWorld** function, which, by default, destroys the actor. Take a look at the following example, which prints to the screen the fact that the actor fell out of the world:

```
void AFPSCharacter::FellOutOfWorld(const
```

```

UDamageType& DmgType)

{

    Super::FallOutOfWorld(DmgType);

    const FString String =
FString::Printf(TEXT("The actor %s has fell
out of the world"), *GetName());

    GEngine->AddOnScreenDebugMessage(-1,
2.0f, FColor::Red, String);

}

```

URotatingMovementComponent

This component rotates the owning actor along time with a certain rate on each axis, defined in the **RotationRate** variable. To use it, you need to include the following header:

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

Declare the component variable:

```
UPROPERTY(VisibleAnywhere,
BlueprintReadOnly, Category = "Test Actor")

URotatingMovementComponent*
RotatingMovement;
```

And finally, initialize it in the actor constructor, like so:

```
RotatingMovement = CreateDefaultSubobject
<URotatingMovementComponent>("Rotating
Movement");

RotatingMovement->RotationRate =
FRotator(0.0, 90.0f, 0);
```

In the preceding code, **RotationRate** is set to rotate **90** degrees per second on the **Yaw** axis.

Exercise 18.02: Making a Simple Multiplayer Pickup Game

In this exercise, we're going to create a new C++ project that uses the Third Person template and we're going to add the following:

- On the owning client, the player controller creates and adds to the viewport a UMG widget that, for each player, displays the score, sorted from highest to lowest, and how many pickups it has collected.
- Create a simple pickup actor class that gives 10 points to the player that picked it up. The pickup will also rotate 90 degrees per second on the **Yaw** axis.
- Set **Kill Z** to **-500** and make the player respawn and lose 10 points every time he falls from the world.
- The game will end when there are no more pickups available. Once the game ends, all characters will be destroyed and, after 5 seconds, the server will do a server travel call to reload the same map and bring along the connected

clients.

The following steps will help you complete the exercise:

1. Create a new **Third Person** template project using **C++** called **Pickups** and save it to a location of your choosing.
2. Once the project has been created, it should open the editor as well as the Visual Studio solution.

Now, let's create the new C++ classes we're going to use:

3. Create the **Pickup** class that is derived from **Actor**.
4. Create the **PickupsGameState** class that is derived from **GameState**.
5. Create the **PickupsPlayerState** class that is derived from **PlayerState**.
6. Create the **PickupsPlayerController** class that is derived from **PlayerController**.

7. Close the editor and open Visual Studio.

Next, let's work on the **Pickup** class.

8. Open **Pickup.h** and clear all existing functions.

9. Declare the protected **Static Mesh** component called **Mesh**:

```
UPROPERTY(VisibleAnywhere,  
BlueprintReadOnly, Category  
= "Pickup")
```

```
UStaticMeshComponent* Mesh;
```

10. Declare the protected rotating movement component called **RotatingMovement**:

```
UPROPERTY(VisibleAnywhere,  
BlueprintReadOnly, Category  
= "Pickup")
```

```
class  
URotatingMovementComponent*  
RotatingMovement;
```

11. Declare the protected **PickupSound** variable:

```
UPROPERTY(EditDefaultsOnly,
```

```
BlueprintReadOnly, Category  
= "Pickup")  
  
USoundBase* PickupSound;
```

12. Declare the protected constructor and **BeginPlay** override:

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

13. Declare the protected **OnBeginOverlap** function:

```
UFUNCTION()  
  
void  
OnBeginOverlap(UPrimitiveCo  
mponent* OverlappedComp,  
AActor* OtherActor,  
UPrimitiveComponent*  
OtherComp, int32  
OtherBodyIndex, bool  
bFromSweep, const  
FHitResult& Hit);
```

14. Open **Pickup.cpp** and include **PickupsCharacter.h**, **PickupsGameState.h**, **StaticMeshComponent.h**, and

RotatingMovementComponent.h:

```
#include  
"PickupsCharacter.h"  
  
#include  
"PickupsGameState.h"  
  
#include  
"Components/StaticMeshCompo  
nent.h"  
  
#include  
"GameFramework/RotatingMove  
mentComponent.h"
```

15. In the constructor, initialize the **Static Mesh** component to overlap with everything and call the **OnBeginOverlap** function when it's overlapped:

```
Mesh =  
CreateDefaultSubobject<USta  
ticMeshComponent>("Mesh");  
  
Mesh-  
>SetCollisionProfileName("O  
verlapAll");  
  
RootComponent = Mesh;
```

16. Still in the constructor, initialize the

rotating movement component to
rotate **90** degrees per second on the
Yaw axis:

```
RotatingMovement =
CreateDefaultSubobject
<URotatingMovementComponent
>("Rotating Movement");

RotatingMovement-
>RotationRate =
FRotator(0.0, 90.0f, 0);
```

17. To finalize the constructor, enable replication and disable the **Tick** function:

```
bReplicates = true;

PrimaryActorTick.bCanEverTi
ck = false;
```

18. Implement the **BeginPlay** function, which binds the begin overlap event to the **OnBeginOverlap** function:

```
void APickup::BeginPlay()
{
    Super::BeginPlay();

    Mesh-
>OnComponentBeginOverlap.Ad
```

```
    dDynamic(this,  
    &APickup::OnBeginOverlap);  
  
}
```

19. Implement the **OnBeginOverlap** function, which checks whether the character is valid and has authority, removes the pickup on the game state, plays the pickup sound on the owning client, adds **10** points and the pickup to the character. Once all of that is done, the pickup destroys itself.

```
void  
APickup::OnBeginOverlap(UPr  
imitiveComponent*  
OverlappedComp, AActor*  
OtherActor,  
UPrimitiveComponent*  
OtherComp, int32  
OtherBodyIndex, bool  
bFromSweep, const  
FHitResult& Hit)  
  
{  
  
    APickupsCharacter*  
    Character =  
    Cast<APickupsCharacter>  
(OtherActor);
```

```

    if (Character == nullptr
    || !HasAuthority())
    {
        return;
    }

    APickupsGameState*
GameState =
Cast<APickupsGameState>
(GetWorld()-
>GetGameState());

    if (GameState != nullptr)
    {
        GameState-
>RemovePickup();
    }

    Character-
>ClientPlaySound2D(PickupSo
und);

    Character->AddScore(10);

    Character->AddPickup();

    Destroy();
}

```

Next, we're going to work on the

PickupsGameState class.

20. Open **PickupsGameState.h** and declare the protected replicated integer variable **PickupsRemaining**, which tells all clients how many pickups remain in the level:

```
UPROPERTY(Replicated,  
BlueprintReadOnly)  
  
int32 PickupsRemaining;
```

21. Declare the protected override for the **BeginPlay** function:

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

22. Declare the protected **GetPlayerStatesOrderedByScore** function:

```
UFUNCTION(BlueprintCallable  
)  
  
TArray<APlayerState*>  
GetPlayerStatesOrderedByScore() const;
```

23. Implement the public **RemovePickup** function, which removes one pickup

from the **PickupsRemaining** variable:

```
void RemovePickup() {  
    PickupsRemaining--;  
}
```

24. Implement the public **HasPickups** function, which returns whether there are still pickups remaining:

```
bool HasPickups() const {  
    return PickupsRemaining >  
        0; }
```

25. Open **PickupsGameState.cpp** and include **Pickup.h**, **GameplayStatics.h**, **UnrealNetwork.h**, and **PlayerState.h**:

```
#include "Pickup.h"  
  
#include  
"Kismet/GameplayStatics.h"  
  
#include  
"Net/UnrealNetwork.h"  
  
#include  
"GameFramework/PlayerState.  
h"
```

26. Implement the

GetLifetimeReplicatedProps
function and make the
PickupRemaining variable replicate
to all clients:

```
void
APickupsGameState::GetLifetimeReplicatedProps(TArray<
FLifetimeProperty >&
OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);
    DOREPLIFETIME(APickupsGameState, PickupsRemaining);
}
```

27. Implement the **BeginPlay** override
function and set the value of
PickupsRemaining by getting all of
the pickups in the world:

```
void
APickupsGameState::BeginPlay()
{
    Super::BeginPlay();
```

```

TArray<AActor*> Pickups;

UGameplayStatics::GetAllActorsOfClass(this,
APickup::StaticClass(),
Pickups);

PickupsRemaining =
Pickups.Num();

}

```

28. Implement the **GetPlayerStatesOrderedByScore** function, which duplicates the **PlayerArray** variable and sorts it so that the players with the highest scores show up first:

```

TArray<APlayerState*>
APickupsGameState::GetPlayerStatesOrderedByScore()
const

{
    TArray<APlayerState*>
    PlayerStates(PlayerArray);

    PlayerStates.Sort([
        (const APlayerState& A,
        const APlayerState& B) {
            return A.Score > B.Score;
    }
}

```

```
    } );  
  
    return PlayerStates;  
  
}
```

Next, let's work on the **PickupsPlayerState** class.

29. Open **PickupsPlayerState.h**, and declare the protected replicated integer variable **Pickups**, which indicates how many pickups a player has collected:

```
UPROPERTY(Replicated,  
BlueprintReadOnly)  
  
int32 Pickups;
```

30. Implement the public **AddPickup** function, which adds one pickup to the **Pickups** variable:

```
void AddPickup() {  
    Pickups++; }
```

31. Open **PickupsPlayerState.cpp** and include **UnrealNetwork.h**:

```
#include  
"Net/UnrealNetwork.h"
```

32. Implement the

GetLifetimeReplicatedProps

function and make the **Pickups**

variable replicate to all clients:

```
void  
APickupsPlayerState::GetLifetimeReplicatedProps(TArray  
< FLifetimeProperty >&  
OutLifetimeProps) const  
{  
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);  
  
    DOREPLIFETIME(APickupsPlayerState, Pickups);  
}
```

Next, let's work on the

PickupsPlayerController class.

33. Open

PickupsPlayerController.h and

declare the protected

ScoreboardMenuClass variable,

which enables the UMG widget we

want to use for our scoreboard to be

selected:

```
UPROPERTY(EditDefaultsOnly,  
BlueprintReadOnly, Category
```

```
= "Pickup Player  
Controller")  
  
TSubclassOf<class  
UUserWidget>  
ScoreboardMenuClass;
```

34. Declare the protected **ScoreboardMenu** variable, which stores the scoreboard UMG widget instance we create on the **BeginPlay** function variable:

```
UPROPERTY()  
  
class UUserWidget*  
ScoreboardMenu;
```

35. Declare the protected override for the **BeginPlay** function:

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

36. Open **PickupsPlayerController.cpp** and include **UserWidget.h**:

```
#include  
"Blueprint/UserWidget.h"
```

37. Implement the **BeginPlay** override function, which, for the owning client,

creates and adds the scoreboard UMG
widget to the viewport:

```
void
APickupsPlayerController::BeginPlay()
{
    Super::BeginPlay();

    if (!IsLocalController())
        || ScoreboardMenuClass ==
        nullptr)

    {
        return;
    }

    ScoreboardMenu =
    CreateWidget<UUserWidget>
    (this,
    ScoreboardMenuClass);

    if (ScoreboardMenu !=
    nullptr)
    {
        ScoreboardMenu-
        >AddToViewport(0);
    }
}
```

```
}
```

Now, let's edit the **PickupsGameMode** class:

38. Open **PickupsGameMode.h** and replace the **include** for **GameModeBase.h** with **GameMode.h**:

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

39. Make the class derive from **AGameMode** instead of **AGameModeBase**:

```
class APickupsGameMode :  
public AGameMode
```

40. Declare the protected game state variable **MyGameState**, which holds the instance to the **APickupsGameState** class:

```
UPROPERTY()  
class APickupsGameState*  
MyGameState;
```

41. Move the constructor to the protected area.

42. Declare the protected override for the

BeginPlay function:

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

43. Declare the protected override for the
ShouldSpawnAtStartSpot
function:

```
virtual bool
ShouldSpawnAtStartSpot(ACon
troller* Player) override;
```

44. Declare the protected overrides for the
match state functions of the game
mode:

```
virtual void
HandleMatchHasStarted()
override;
```

```
virtual void
HandleMatchHasEnded()
override;
```

```
virtual bool
ReadyToStartMatch_Implement
ation() override;
```

```
virtual bool
ReadyToEndMatch_Implementat
ion() override;
```

45. Declare the protected **RestartMap** function:

```
void RestartMap();
```

46. Open **PickupsGameMode.cpp** and include **GameplayStatics.h**, **PickupGameState.h**, **Engine/World.h**, **TimerManager.h**, and **Engine.h**:

```
#include "Kismet/GameplayStatics.h"  
#include "PickupsGameState.h"  
#include "Engine/World.h"  
#include "Engine/Public/TimerManager.h"  
#include "Engine/Engine.h"
```

47. Implement the **BeginPlay** override function, which stores the **APickupGameState** instance:

```
void APickupsGameMode::BeginPlay()
```

```
{  
    Super::BeginPlay();  
  
    MyGameState =  
        GetGameState<APickupsGameSt  
ate>();  
  
}
```

48. Implement the **ShouldSpawnAtStartSpot** override function, which indicates that we want the players to respawn on a random player start and not always on the same one:

```
bool  
APickupsGameMode::ShouldSpa  
wnAtStartSpot (AController*  
Player)  
  
{  
    return false;  
}
```

49. Implement the **HandleMatchHasStarted** override function, which prints to the screen, informing players that the game has started:

```
void
APickupsGameMode::HandleMatchHasStarted()
{
    Super::HandleMatchHasStarted();

    GEngine-
>AddOnScreenDebugMessage(-1
, 2.0f, FColor::Green, "The
game has started!");

}
```

50. Implement the **HandleMatchHasEnded** override function, which prints to the screen, informing players that the game has ended, destroys all characters, and schedules a timer to restart the map:

```
void
APickupsGameMode::HandleMatchHasEnded()
{
    Super::HandleMatchHasEnded();

    GEngine-
>AddOnScreenDebugMessage(-1
```

```

        , 2.0f, FColor::Red, "The
game has ended!");

    TArray<AActor*>
Characters;

    UGameplayStatics::GetAl
lActorsOfClass(this,
APickupsCharacter::StaticCl
ass(), Characters);

    for (AActor* Character :
Characters)

{
    Character->Destroy();

}
FTimerHandle TimerHandle;

GetWorldTimerManager().Se
tTimer(TimerHandle, this,
&APickupsGameMode::RestartM
ap, 5.0f);

}

```

51. Implement the **ReadyToStartMatch_Implementa**
tion override function, which
indicates that the match can start
straight away:

```
bool  
APickupsGameMode::ReadyToStartMatch_Implementation()  
{  
    return true;  
}
```

52. Implement the **ReadyToEndMatch_Implementation** override function, which indicates that the match ends when the game state has no more pickups remaining:

```
bool  
APickupsGameMode::ReadyToEndMatch_Implementation()  
{  
    return MyGameState !=  
nullptr && !MyGameState-  
>HasPickups();  
}
```

53. Implement the **RestartMap** function, which indicates that the server travels to the same level and brings all clients along (*only in the packaged version*):

```
void
```

```

APickupsGameMode::RestartMa
p()

{
    GetWorld()->ServerTravel(GetWorld()->GetName(), false, false);
}

```

Now, let's edit the **PickupsCharacter** class.

54. Open **PickupsCharacter.h** and declare the protected sound variables for falling and landing:

```

UPROPERTY(EditDefaultsOnly,
BlueprintReadOnly, Category
= "Pickups Character")
USoundBase* FallSound;

UPROPERTY(EditDefaultsOnly,
BlueprintReadOnly, Category
= "Pickups Character")
USoundBase* LandSound;

```

55. Declare the protected **override** functions:

```
virtual void EndPlay(const
```

```
EEndPlayReason::Type  
EndPlayReason) override;  
  
virtual void Landed(const  
FHitResult& Hit) override;  
  
virtual void  
FellOutOfWorld(const  
UDamageType& DmgType)  
override;
```

56. Declare the public functions that add scores and pickups to the player state:

```
void AddScore(const float  
Score);  
  
void AddPickup();
```

57. Declare the public client RPC that plays a sound on the owning client:

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

58. Open **PickupsCharacter.cpp** and include **PickupsPlayerState.h**, **GameMode.h**, and **GameplayStatics.h**:

```
#include  
"PickupsPlayerState.h"  
  
#include  
"GameFramework/GameMode.h"  
  
#include  
"Kismet/GameplayStatics.h"
```

59. Implement the **EndPlay** override function, which plays the fall sound if the character was destroyed:

```
void  
APickupsCharacter::EndPlay(  
const EEndPlayReason::Type  
EndPlayReason)  
  
{  
    Super::EndPlay(EndPlayRea  
son);  
  
    if (EndPlayReason ==  
EEndPlayReason::Destroyed)  
  
    {  
        UGameplayStatics::PlayS  
ound2D(GetWorld(),  
FallSound);  
  
    }  
}
```

60. Implement the **Landed** override function, which plays the landed sound:

```
void  
APickupsCharacter::Landed(c  
onst FHitResult& Hit)  
{  
    Super::Landed(Hit);  
  
    UGameplayStatics::PlaySou  
nd2D(GetWorld(),  
    LandSound);  
}
```

61. Implement the **FellOutOfWorld** override function, which stores the controller, removes **10** points from the score, destroys the character (which makes the controller invalid), and tells the game mode to restart the player using the previous controller:

```
void  
APickupsCharacter::FellOutO  
fWorld(const UDamageType&  
DmgType)  
{
```

```

AController*
PreviousController =
Controller;

AddScore(-10);

Destroy();

AGameMode* GameMode =
GetWorld()->GetAuthGameMode<AGameMode>();
if (GameMode != nullptr)
{
    GameMode->RestartPlayer(PreviousController);
}
}

```

62. Implement the **AddScore** function, which adds a score to the **Score** variable in the player state:

```

void
APickupsCharacter::AddScore
(const float Score)

{

```

```

APlayerState*
MyPlayerState =
GetPlayerState();

    if (MyPlayerState !=
nullptr)

    {

        MyPlayerState->Score +=

Score;

    }

}

```

63. Implement the **AddPickup** function, which adds a pickup to the **Pickup** variable in our custom player state:

```

void
APickupsCharacter::AddPicku
p()

{
    APickupsPlayerState*
MyPlayerState =
GetPlayerState<APickupsPlay
erState>();

    if (MyPlayerState !=
nullptr)

    {

```

```
    MyPlayerState->AddPickup();  
}  
}
```

64. Implement the **ClientPlaySound2D_Implementation** function, which plays a sound on the owning client:

```
void APickupsCharacter::ClientPlaySound2D_Implementation(USoundBase* Sound)  
{  
    UGameplayStatics::PlaySound2D(GetWorld(), Sound);  
}
```

65. Open **Pickups.Build.cs** and add the **UMG** module to **PublicDependencyModuleNames**, like so:

```
PublicDependencyModuleNames  
.AddRange(new string[] {  
    "Core", "CoreUObject",  
    "Engine", "InputCore",
```

```
"HeadMountedDisplay", "UMG"  
});
```

If you try to compile and get errors from adding the new module, then clean and recompile your project. If that doesn't work, try restarting your IDE.

66. Compile and run the code until the editor loads.

First, let's import the sound files.

67. In **Content Browser**, create and go to the **Content\Sounds** folder.
68. Import **Pickup.wav**,
Footstep.wav, **Jump.wav**,
Land.wav, and **Fall.wav** from the **Exercise18.02\Assets** folder.
69. Save the new files.

Next, let's add the **Play Sound** anim notifies to some of the character's animations.

70. Open **ThirdPersonJump_Start animation**, located in **Content\Mannequin\Animations**,

and add a **Play Sound** anim notify at frame **0** using the **Jump** sound.

71. Save and close **ThirdPersonJump_Start**.
72. Open the **ThirdPersonRun** animation, located in **Content\Mannequin\Animations**, and add two **Play Sound** anim notifies using the **Footstep** sound at time **0.24** sec and **0.56** sec.
73. Save and close **ThirdPersonRun**.
74. Open the **ThirdPersonWalk** animation, located in **Content\Mannequin\Animations**, and add two **Play Sound** anim notifies using the **Footstep** sound at time **0.24** sec and **0.79** sec.
75. Save and close **ThirdPersonWalk**.

Now, let's set the sounds for the character blueprint.
76. Open the **ThirdPersonCharacter** blueprint, located in **Content\ThirdPersonCPP\Blueprints**

rints, and set the **Fall Sound** and **Land Sound** to use the sounds **Fall** and **Land**, respectively.

77. Save and close **ThirdPersonCharacter**.

Now, let's create the blueprint for the pickup.

78. Create and open the **Content\Blueprints** folder.
79. Create a new blueprint called **BP_Pickup** that is derived from the **Pickup** class and open it.
80. Configure the **Static Mesh** component in the following way:

Scale = 0.5, 0.5, 0.5

Static Mesh =
Engine\BasicShapes\Cube

Material Element 0 =
CubeMaterial

Note

To display the Engine content, you need to go to View Options on the

bottom right of the dropdown for the static mesh and make sure the Show Engine Content flag is set to true.

81. Set the **Pickup Sound** variable to use the **Pickup** sound.
82. Save and close **BP_Pickup**.

Next, let's create the scoreboard UMG widgets.

83. Create and go to the **Content\UI** folder.
84. Create a new widget blueprint called **UI_Scoreboard_Header**:
 1. Add a text block called **tbName** to the root canvas panel with **Is Variable** set to **true**, **Size To Content** set to **true**, **Text** set to **Player Name**, and **Color and Opacity** set to use the color **green**.
 2. Add a text block called **tbScore** to the root canvas panel with **Is Variable** set to **true**, **Position X =**

500, Alignment = 1.0,
0.0, Size To Content set
to true, Text set to Score,
and Color and Opacity
set to use the color green.

3. Add a text block called
tbPickups to the root
canvas panel with **Is**
Variable set to **true**,
Position X = 650,
Alignment = 1.0, 0.0,
Size To Content set to
true, Text set to **Pickups**,
and **Color and Opacity**
set to use the color **green**.
85. From the **Hierarchy** panel, select the
three new text blocks and copy them.
86. Save and close
UI_Scoreboard_Header.
87. Go back to **Content\UI**, create a new
UMG widget called
UI_Scoreboard_Entry, and open it.
88. Paste the copied text blocks on the root
canvas panel and change them to be

white instead of **green** and make them all variables.

89. Go to the **Graph** section and create the **Player State** variable with the following configuration:

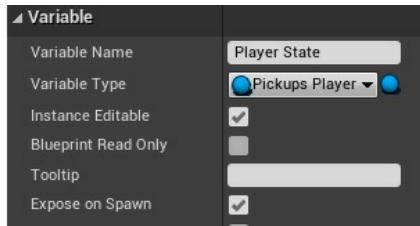


Figure 18.3: Creating the Player State variable

90. Go back to the Designer section and create a bind for **tbName** that does the following:

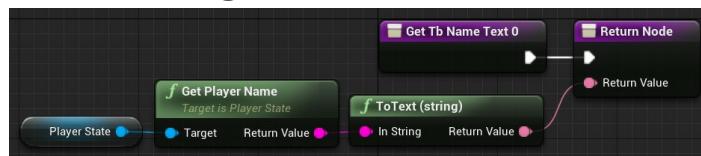


Figure 18.4: Displaying the player name

Note

You can find the preceding screenshot in full resolution for better viewing at the following link:

<https://packt.live/3pCk9Nt>.

91. Create a bind for **tbScore** that does the following:

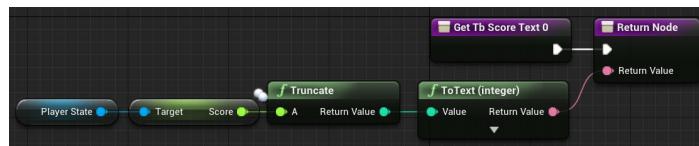


Figure 18.5: Displaying the player score

Note

You can find the preceding screenshot in full resolution for better viewing at the following link:

<https://packt.live/3nuckYv>.

92. Create a bind for **tbPickups** that does the following:

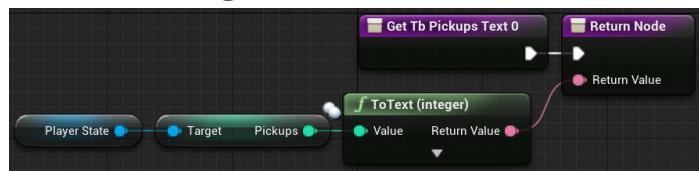


Figure 18.6: Displaying the pickups count

Note

You can find the preceding screenshot in full resolution for better viewing at the following link:

<https://packt.live/36pEGMz>.

93. Create a pure function called **Get Typeface** that does the following:

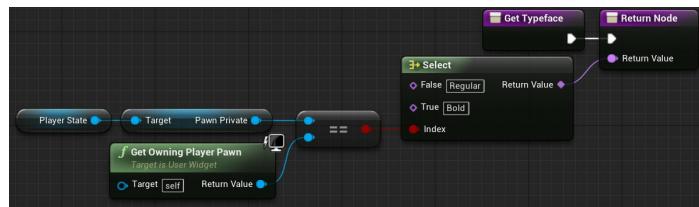


Figure 18.7: Determining whether the entry should be displayed in bold or regular

Note

You can find the preceding screenshot in full resolution for better viewing at the following link:

<https://packt.live/2JW9Zam>.

In the preceding code, we use a select node, which can be created by dragging a wire from the return value and releasing it on an empty space, and type "select" on the filter. From there, we pick the select node from the list. In this specific function we use the select node to pick the name of the typeface we're going to use, so it should return **Regular** if the player state's pawn is

not the same as the pawn that owns the widget and **Bold** if it is. We do this to highlight in bold the player state entry so that the player knows what their entry is.

94. Implement **Event Construct** in the following way:

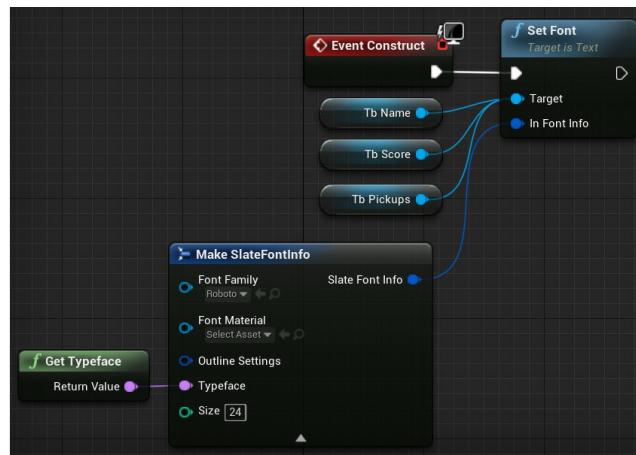


Figure 18.8: The Event Graph that sets the text for the name, score, and pickups count

Note

You can find the preceding screenshot in full resolution for better viewing at the following link:

<https://packt.live/2JOdP58>.

In the preceding code, we set the font for **tbName**, **tbScore**, and

tbPickups to use the **Bold** typeface to highlight which scoreboard entry is relative to the player of the current client. For the remainder of the players, use the **Regular** typeface.

95. Save and close **UI_Scoreboard_Entry**.
96. Go back to **Content\UI** and then create a new UMG widget called **UI_Scoreboard** and open it.
97. Add a vertical box called **vbScoreboard** to the root canvas panel with **Size To Content** enabled.
98. Add a text block to **vbScoreboard** called **tbGameInfo** that has the **Text** value defaulted to **Game Info**.
99. Go to the **Graph** section and create a new variable called **Game State** of the **Pickups Game State** type.
100. Implement **Event Construct** in the following way:

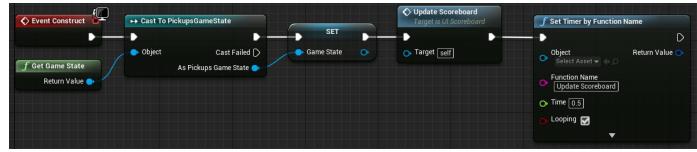


Figure 18.9: The Event Construct that sets a timer to update the scoreboard every 0.5 seconds

Note

You can find the preceding screenshot in full resolution for better viewing at the following link:

<https://packt.live/3kemyuo>.

In the preceding code, we get the game state instance, update the scoreboard, and schedule a timer to automatically update the scoreboard every 0.5 seconds.

101. Go back to the designer section and make the following bind for **vbScoreboard**:

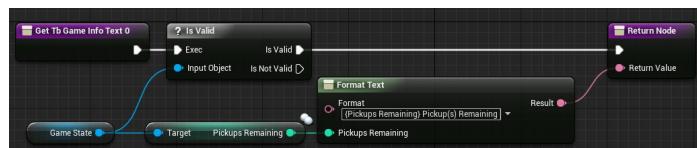


Figure 18.10: Displaying the number of pickups remaining in the world

Note

You can find the preceding screenshot in full resolution for better viewing at the following link:

<https://packt.live/38xUDTE>.

102. Add a vertical box to **vbScoreboard** called **vbPlayerStates** with **Is Variable** set to **true** and a top padding of **50**, so you should have the following:

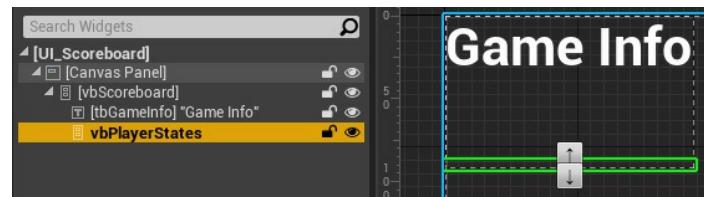


Figure 18.11: The UI_Scoreboard widget hierarchy

103. Go back to the Graph section and implement the **Update Scoreboard** event in the following way:



Figure 18.12: The update scoreboard function, which clears and recreates the entry widgets

Note

You can find the preceding screenshot in full resolution for better viewing at the following link:

<https://packt.live/3pf8EeN>.

In the preceding code, we do the following:

1. Clear all of the previous entries in **vbPlayerStates**.
 2. Create a scoreboard header entry and add it to **vbPlayerStates**.
 3. Loop through all of the player states ordered by score and create an entry for each one, as well as adding it to **vbPlayerStates**.
104. Save and close **UI_Scoreboard**.
- Now, let's create the blueprint for the player controller.
105. Go to **Content\Blueprints** and create a new blueprint called **BP_PlayerController** that is

derived from the
PickupPlayerController class.

106. Open the new blueprint and set the **Scoreboard Menu Class** to use **UI_Scoreboard**.

107. Save and close **BP_PlayerController**.

Next, let's create the blueprint for the game mode.

108. Go to **Content\Blueprints** and create a new blueprint called **BP_GameMode** that is derived from the **PickupGameMode** class, open it, and change the following variables:

Game State Class =
PickupsGameState

Player Controller Class =
BP_PlayerController

Player State Class =
PickupsPlayerState

Next, let's configure **Project Settings** to use the new game mode.

109. Go to **Project Settings** and pick

Maps & Modes from the left panel, which is in the **Project** category.

110. Set **Default GameMode** to use **BP_GameMode**.

111. Close **Project Settings**.

Now, let's modify the main level.

112. Make sure you have **ThirdPersonExampleMap** opened, located in **Content\ThirdPersonCPP\Maps**.
113. Add some cube actors to act as platforms and make sure they have gaps between them to force the player to jump on them and possibly fall from the level.
114. Add a couple of player start actors to different parts of the map.
115. Add at least 50 instances of **BP_Pickup** and spread them across the entire map.
116. Here is an example of a possible way of configuring the map:

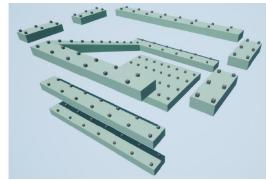


Figure 18.13: An example of the map configuration

117. Run the code and wait for the editor to fully load.
118. Go to **Multiplayer Options** and set the number of clients to **2**.
119. Set the window size to **800x600**.
120. Play in **New Editor Window (PIE)**:

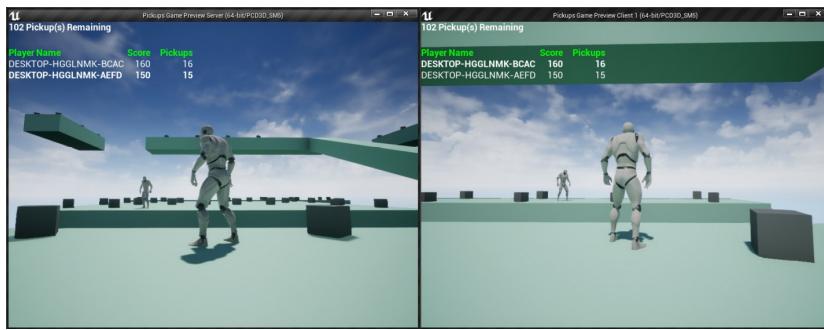


Figure 18.14: The listen Server and Client 1 picking up cubes in the world

Once you complete this exercise, you will be able to play on each client and you'll notice that the characters can collect pickups and gain **10** points just by overlapping with them. If a character falls from the level, it will respawn on a random player start and lose **10** points.

Once all pickups have been collected, the game will end, and after **5** seconds, it will perform a server travel to reload the same level and bring all the clients with it (*only in the packaged version*). You can also see that the UI displays how many pickups are remaining in the level, as well as the scoreboard with the information about the name, score, and pickups for each player.

Activity 18.01: Adding Death, Respawn, Scoreboard, Kill Limit, and Pickups to the Multiplayer FPS Game

In this activity, you'll add the concept of death, respawning, and the ability to use pickups to the character. We'll also add a way to check the scoreboard and a kill limit to the game so that it has an end goal.

The following steps will help you complete this activity:

1. Open the **MultiplayerFPS** project from *Activity 17.01, Adding Weapons and Ammo to the Multiplayer FPS Game*. Compile the code and run the editor.
2. Next, you're going to create the C++ classes that we're going to need. Create

a C++ class called **FPSGameState**, which is derived from the **GameState** class, and has a kill limit variable and a function that returns the player states ordered by kills.

3. Create a C++ class called **FPSPlayerState**, which is derived from the **PlayerState** class, and stores the number of kills and deaths of a player.
4. Create a C++ class called **PlayerMenu**, which is derived from the **UserWidget** class, and has some **BlueprintImplementableEvent** functions to toggle the scoreboard visibility, set the scoreboard visibility, and notify when a player was killed.
5. Create a C++ class called **FPSPlayerController**, which derives from **APlayerController**, that creates the **PlayerMenu** UMG widget instance on the owning client.
6. Create a C++ class called **Pickup**, which is derived from the **Actor** class, and has a static mesh that rotates 90

degrees per second on the **Yaw** axis, and can be picked up by the player on the overlap. Once picked up, it plays a pickup sound, and disables collision and visibility. After a certain amount of time, it will make it visible and able to collide again.

7. Create a C++ class called **AmmoPickup**, which is derived from the **Pickup** class, and adds a certain amount of an ammo type to the player.
8. Create a C++ class called **ArmorPickup**, which is derived from the **Pickup** class, and adds a certain amount of armor to the player.
9. Create a C++ class called **HealthPickup**, which is derived from the **Pickup** class, and adds a certain amount of health to the player.
10. Create a C++ class called **WeaponPickup**, which is derived from the **Pickup** class, and adds a certain weapon type to the player. If the player already has the weapon, it will add a certain amount of ammo.

11. Edit the **FPSCharacter** class so that it does the following:

1. After the character is damaged, checks to see whether it's dead. If it's dead, it registers the kill for the killer character and the death of the character, as well as respawning the player. If the character is not dead, then it plays the pain sound on the owning client.
2. When the character dies and executes the **EndPlay** function, it should destroy all of its weapon instances.
3. If the character falls from the world, it will register the death of the player and respawn it.
4. If the player presses the *Tab* key, it will toggle the visibility of the scoreboard menu.

12. Edit the **MultiplayerFPSGameModeBase** class so that it does the following:

1. Stores the number of kills necessary to win the game.
 2. Uses the new player controller, player state, and game state classes.
 3. Makes it implement the match state functions so that the match starts immediately and ends if there is a player that has the required number of kills.
 4. When the match ends, it will perform a server travel to the same level after 5 seconds.
 5. Handles when a player dies by adding the kill (when killed by another player) and the death to the respective player state, as well as respawning the player on a random player start.
13. Import **AmmoPickup.wav** from **Activity18.01\Assets** to **Content\Pickups\Ammo**.
14. Create **BP_PistolBullets_Pickup**

from **AAmmoPickup**, place it in **Content\Pickups\Ammo**, and configure it with the following values:

1. Scale: (**X=0.5, Y=0.5, Z=0.5**)
2. Static Mesh:
Engine\BasicShapes\Cube
3. Material:
Content\Weapon\Pistol\M_Pistol
4. Ammo Type: **Pistol Bullets**, Ammo Amount:
25
5. Pickup Sound:
Content\Pickup\Ammo\AmmoPickup

15. Create **BP_MachineGunBullets_Pickup** from **AAmmoPickup**, place it in **Content\Pickups\Ammo**, and configure it with the following values:

1. Scale: (**X=0.5, Y=0.5, Z=0.5**)

2. Static Mesh:
Engine\BasicShapes\Cube
 3. Material:
Content\Weapon\MachineGun\M_MachineGun
 4. Ammo Type: **Machine Gun Bullets**, Ammo Amount:
50
 5. Pickup Sound:
Content\Pickup\Ammo\AAmmoPickup
16. Create **BP_Slugs_Pickup** from **AAmmoPickup**, place it in **Content\Pickups\Ammo**, and configure it with the following values:
1. Scale: (**X=0.5, Y=0.5, Z=0.5**)
 2. Static Mesh:
Engine\BasicShapes\Cube
 3. Material:
Content\Weapon\Railgun\M_Railgun

4. Ammo Type: **Slugs**, Ammo Amount: **5**
 5. Pickup Sound:
Content\Pickup\Ammo\AmmoPickup
17. Import **ArmorPickup.wav** from **Activity18.01\Assets** to **Content\Pickups\Armor**.
18. Create the material **M_Armor** in **Content\Pickups\Armor**, which has **Base Color** set to **blue** and **Metallic** set to **1**.
19. Create **BP_Armor_Pickup** from **AArmorPickup**, place it in **Content\Pickups\Armor**, and configure it with the following values:
1. Scale: (**X=1.0, Y=1.5, Z=1.0**)
 2. Static Mesh:
Engine\BasicShapes\Cube
 3. Material:
Content\Pickup\Armor

M_Armor

4. Armor Amount: **50**
 5. Pickup Sound:
**Content\Pickup\Armor\
ArmorPickup**
20. Import **HealthPickup.wav** from
Activity18.01\Assets to
Content\Pickups\Health.
21. Create the material **M_Health** in
Content\Pickups\Health, which
has **Base Color** set to **blue** and
Metallic/Roughness set to **0.5**.
22. Create **BP_Health_Pickup** from
AHealthPickup, place it in
Content\Pickups\Health, and
configure it with the following values:
1. Static Mesh:
**Engine\BasicShapes\Sp
here**
 2. Material:
**Content\Pickup\Health
\M_Health**
 3. Health Amount: **50**

4. Pickup Sound:
**Content\Pickup\Health
\HealthPickup**
23. Import **WeaponPickup.wav** from
Activity18.01\Assets to
Content\Pickups\Weapon.
24. Create **BP_Pistol_Pickup** from
AWeaponPickup, place it in
Content\Pickups\Weapon, and
configure it with the following values:
 1. Static Mesh:
**Content\Pickup\Weapon
\SM_Weapon**
 2. Material:
**Content\Weapon\Pistol
\M_Pistol**
 3. Weapon Type: **Pistol**,
Ammo Amount: **25**
 4. Pickup Sound:
**Content\Pickup\Weapon
\WeaponPickup**
25. Create **BP_MachineGun_Pickup**
from **AWeaponPickup**, place it in

Content\Pickups\Weapon, and
configure it with the following values:

1. Static Mesh:

Content\Pickup\Weapon
\SM_Weapon

2. Material:

Content\Weapon\Machin
eGun\M_MachineGun

3. Weapon Type: **Machine**

Gun, Ammo Amount: **50**

4. Pickup Sound:

Content\Pickup\Weapon
\WeaponPickup

26. Create **BP_Pistol_Pickup** from
AWeaponPickup, place it in
Content\Pickups\Weapon, and
configure it with the following values:

1. Static Mesh:

Content\Pickup\Weapon
\SM_Weapon

2. Material:

Content\Weapon\Railgu
n\M_Railgun

3. Weapon Type: **Railgun**,

Ammo Amount: **5**

4. Pickup Sound:
Content\Pickup\Weapon
\WeaponPickup

27. Import **Land.wav** and **Pain.wav** from **Activity18.01\Assets** to **Content\Player\Sounds**.
28. Edit **BP_Player** so that it uses the **Pain** and **Land** sounds, as well as deleting all of the nodes that create and add the **UI_HUD** instance to the viewport in the **Begin Play** event.
29. Create a UMG widget called **UI_Scoreboard_Entry** in **Content\UI** that displays the name, kills, deaths, and ping of **AFPSPlayerState**.
30. Create a UMG widget called **UI_Scoreboard_Header** that displays the headers for the name, kills, deaths, and ping.
31. Create a UMG widget called **UI_Scoreboard** that displays the kill

limit from the game state, a vertical box that has **UI_Scoreboard_Header** as the first entry, and then add a **UI_Scoreboard_Entry** for each **AFPSPlayerState** in the game state instance. The vertical box will update every 0.5 seconds, through a timer, by clearing its children and adding them again.

32. Edit **UI_HUD** so that it adds a new text block called **tbKilled** that starts with **Visibility** set to **Hidden**. When the player kills someone, it will make the text block visible, display the name of the killed player, and hide after 1 second.
33. Create a new blueprint called **UI_PlayerMenu** from **UPlayerMenu** and place it in **Content\UI**. Use a widget switcher with an instance of **UI_HUD** in index **0** and an instance of **UI_Scoreboard** in index **1**. In the event graph, make sure to override the **Toggle Scoreboard, Set Scoreboard Visibility**, and

Notify Kill events that were set as **BlueprintImplementableEvent** in C++. The **Toggle Scoreboard** event toggles the widget switcher's active index between **0** and **1**, the **Set Scoreboard Visibility** event sets the widget switcher's active index to **0** or **1**, and the **Notify Kill** event tells the **UI_HUD** instance to set the text and fade out the animation.

34. Create **BP_PlayerController** from **AFPSPlayerController**, place it in the **Content** folder, and set the **PlayerMenuClass** variable to use **UI_PlayerMenu**.
35. Edit **BP_GameMode** and set **Player Controller Class** to use **BP_PlayerController**.
36. In the **Input** section of **Project Settings**, create an action mapping called **Scoreboard** that uses the **TAB** key.
37. Edit the **DM-Test** level so that you have at least three new player starts

placed in different locations, **Kill z** to **-500** in **World Settings**, and an instance placed of every different pickup.

Expected output:



Figure 18.15: The expected output of the activity

The result should be a project where each client's character can pick up, use, and switch between three different weapons. If a character kills another, it should register the kill and the death, as well as respawning the character that died at a random player start. You should have a scoreboard that displays the name, kill count, death count, and ping, for each player. A character can fall from the level, which should only count as a death, and respawn at a random player start. The character should also be able to pick up the different pickups in the level to get ammo, armor, health, and weapons. The game should end when the kill limit has been reached by showing the scoreboard and server travel to the same level after 5 seconds.

Note

*The solution to this activity can be found at:
<https://packt.live/338jEBx>.*

Summary

In this chapter, you learned that the gameplay framework class's instances exist in some game instances, but not in others. Having that knowledge will help you understand which instances can be accessed in a particular game instance. You also learned the purpose of the game state and player state classes, as well as learning new concepts for the game mode and some useful built-in functionalities.

At the end of this chapter, you have made a basic but functional multiplayer shooter that can be used as a foundation to build upon. You can add new weapons, ammo types, fire modes, pickups, and so on, to make it more feature-complete and fun.

Having completed this book, you should now have a better understanding of how to use Unreal Engine 4 to make your own games come to life. We've covered a lot of topics in this book, ranging from the simple to more advanced. You started out by learning how to create projects using the different templates and how to use blueprints to create actors and components. You then saw how to create a fully functioning Third Person template from scratch by importing and setting up the required assets, setting up the animation blueprint and Blend Space, and creating your own game mode and character, as well as defining and handling the inputs.

You then moved on to your first project; a simple stealth game that uses game physics and collisions, projectile movement components, actor components, interfaces, blueprint function libraries, UMG, sounds, and particle effects. Following this, you learned how to create a simple side-scrolling game by using AI, Anim Montages, and Destructible Meshes. Finally, you discovered how to create a first-person multiplayer shooter by using the Server-Client architecture, variable replication, and RPCs that come with the Network Framework, and learned how the Player State, Game State, and Game Mode classes work.

By working on various projects that use different parts of the engine, you now have a strong understanding of how Unreal Engine 4 works, and although this is the end of this book, it is just the start of your journey into the world of game development using Unreal Engine 4.

7. UE4 Utilities

Overview

This chapter will resume work on the dodgeball-based game that we started making in the previous chapters. We will continue with the dodgeball game by learning about a helpful set of utilities that you can implement in UE4 in order to improve the quality of your project's structure and reuse logic from it in other projects. We will specifically be talking about Blueprint Function Libraries, Actor Components, and Interfaces. By the end of this chapter, you will be able to use these utilities and other tools in your projects.

Introduction

In the previous chapter, we learned about the remaining collision-related concepts in UE4, such as collision events, object types, physics simulation, and collision components. We learned how to have objects collide against one another, changing their responses to different collision channels, as well as how to create our own collision presets and spawn actors and use timers.

In this chapter, we will go into several UE4 utilities that will allow you to easily move logic from one project to another and to keep your project well structured and organized, which will make life much easier for you in the long run and also make it easier for other people in your team to understand your work and modify it in the future. Game development is a tremendously hard task and is rarely done individually, but

rather in teams, so it's important to take these things into account when building your projects.

We will mainly be talking about Blueprint Function Libraries, Actor Components, and Interfaces.

Blueprint Function Libraries will be used to move some generic functions in our project from a specific actor to a Blueprint Function Library so that it can be used in other parts of our project's logic.

Actor Components will be used to move part of some actor classes' source code into an Actor Component, so that we can easily use that logic in other projects. This will keep our project loosely coupled. **Loose coupling** is a software engineering concept that refers to having your project structured in such a way that you can easily remove and add things as you need. The reason you should strive for loose coupling is if you want to reuse parts of one of your projects for another project as a game developer, loose coupling will allow you to do that much more easily.

A practical example of how you could apply loose coupling is if you had a player character class that was able to fly and also had an inventory with several usable items. Instead of implementing the code responsible for both those things in that player character class, you would implement the logic for each of those in separate Actor Components, that you then add to the class. This will not only make it easier to add and remove things that this class will do, by simply adding and removing the Actor Components responsible for those things, but also allow you to reuse those Actor Components in other projects where you have a character that has an inventory or is able to fly. This is one of the main purposes of Actor Components.

Interfaces, much like Actor Components, make our project better structured and organized.

Let's get started with these concepts by talking about

Blueprint Function Libraries.

Blueprint Function Libraries

In UE4, there's a class called **BlueprintFunctionLibrary**, which is meant to contain a collection of static functions that don't really belong to any specific actor and can be used in multiple parts of your project.

For instance, some of the objects that we used previously, such as the **GameplayStatics** object and **Kismet** libraries such as **KismetMathLibrary** and **KismetSystemLibrary**, are **Blueprint Function** Libraries. These contain functions that can be used in any part of your project.

There is at least one function in our project created by us that can be moved to a Blueprint Function library: the **CanSeeActor** function defined in the **EnemyCharacter** class.

Let's then, in the first exercise of this chapter, create our own **Blueprint Function** library, so that we can then move the **CanSeeActor** function from the **EnemyCharacter** class to the **Blueprint Function** library class.

Exercise 7.01: Moving the CanSeeActor Function to the Blueprint Function Library

In this exercise, we will be moving the **CanSeeActor** function that we created for the **EnemyCharacter** class to a **Blueprint Function Library**.

The following steps will help you complete this exercise:

1. Open the Unreal editor.
2. *Right-click* inside the **Content Browser** and select **New C++ Class**.
3. Choose **BlueprintFunctionLibrary** as the parent class of this C++ class (you'll find it by scrolling to the end of the panel).
4. Name the new C++ class **DodgeballFunctionLibrary**.
5. After the class's files have been generated in Visual Studio, open them and close the editor.
6. In the header file of **DodgeballFunctionLibrary**, add a declaration for a **public** function called **CanSeeActor**. This function will be similar to the one we created in the **EnemyCharacter** class, however, there will be some differences.

The new **CanSeeActor** function will be **static**, will return a **bool**, and will receive the following parameters:

1. A **const UWorld* World** property, which we will use to access the **Line Trace** functions.
2. An **FVector Location** property, which we will use as the location of the actor that is checking whether it can see the target actor.
3. A **const AActor* TargetActor** property, which will be the actor we're checking visibility for.
4. A **TArray<const AActor*> IgnoreActors** property, which will be the actors that should be ignored during the **Line Trace** functions. This property can have an empty array as a default argument:

```
public:  
    // Can we see the
```

```
given actor

    static bool
    CanSeeActor(
        const UWorld* World,
        FVector Location,
        const AActor*
        TargetActor,
        TArray<const
        AActor*>
        IgnoreActors =
        TArray<const
        AActor*>());
```

7. Create the implementation of this function in the class's source file and copy the implementation of the **EnemyCharacter** class's version into this new class. After you've done that, make the following modifications to the implementation:

1. Change the value of the **Start** location of **Line Trace** to the **Location** parameter:

```
// Where the Line
Trace starts and
```

```
ends  
  
FVector Start =  
Location;
```

2. Instead of ignoring this actor (using the **this** pointer) and **TargetActor**, ignore the entire **IgnoreActors** array using the **AddIgnoredActors** function of **FCollisionQueryParams**, and sending that array as a parameter:

```
FCollisionQueryParam  
s QueryParams;  
  
// Ignore the actors  
specified  
  
QueryParams.AddIgnor  
edActors(IgnoreActor  
s);
```

3. Replace both calls to the **GetWorld** function with the received **World** parameter:

```
// Execute the Line  
Trace
```

```

World-
>LineTraceSingleByCh
annel(Hit, Start,
End, Channel,
QueryParams);

// Show the Line
Trace inside the
game

DrawDebugLine(World,
Start, End,
FColor::Red);

```

4. Add the necessary includes to the top of the **DodgeballFunctionLibrary** class as shown in the following code snippet:

```

#include
"Engine/World.h"

#include
"DrawDebugHelpers.h"

#include
"CollisionQueryParams.h"

```

8. After you've created the new version of the **CanSeeActor** function inside

DodgeballFunctionLibrary, head to our **EnemyCharacter** class and make the following changes:

1. Remove the declaration and implementation of the **CanSeeActor** function, inside its header and source file respectively.
2. Remove the **DrawDebugHelpers** include, given that we will no longer need that file:

```
// Remove this line  
  
#include  
"DrawDebugHelpers.h"
```

3. Add an include for **DodgeballFunctionLibrary**:

```
#include  
"DodgeballFunctionLibrary.h"
```

4. Inside the class's **LookAtActor** function, just before the **if** statement that calls the **CanSeeActor**

function, declare a **const**
TArray<const AActor*>
IgnoreActors variable and
set it to both the **this**
pointer and the
TargetActor parameter:

```
const TArray<const  
AActor*>  
IgnoreActors =  
{this, TargetActor};
```

Note

*Introducing this last code
snippet might give you an
IntelliSense error in Visual
Studio. You can safely ignore
it, as your code should
compile with no issues
regardless.*

9. Replace the existing call to the **CanSeeActor** function with the one we just created, by sending the following as parameters:
 1. The current world, through the **GetWorld** function

2. The **SightSource** component's location, using its **GetComponentLocation** function
3. The **TargetActor** parameter
4. The **IgnoreActors** array we just created:

```
if
(UDodgeballFunctionL
ibrary::CanSeeActor(
    GetWorld(),
    SightSource-
    >GetComponentLocatio
    n(),
    TargetActor,
    IgnoreActors))
```

Now that you've made all those changes, compile your code, open your project, and verify that the **EnemyCharacter** class still looks at the player as it walks around, as long as it's in the

enemy character's sight, as shown in the following screenshot:

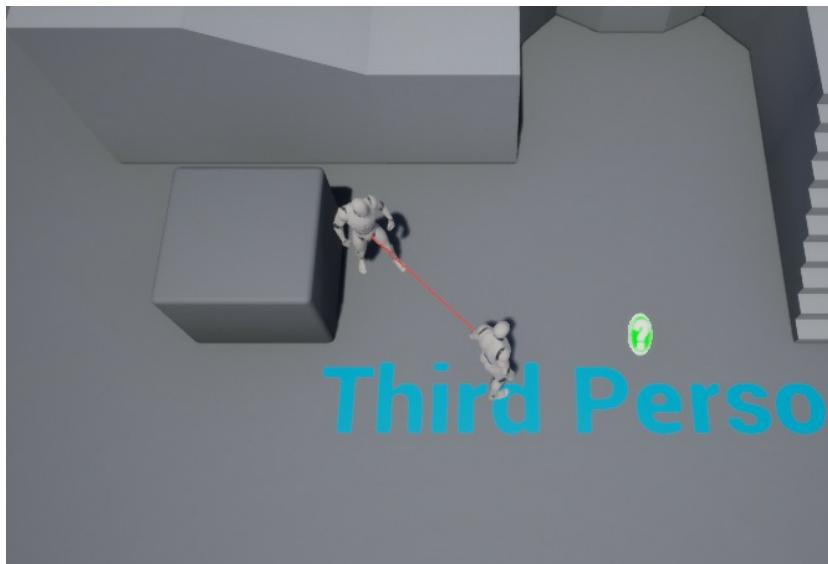


Figure 7.1: The enemy character still looking at the player character

And that concludes our exercise. We've put our **CanSeeActor** function inside of a **Blueprint Function Library** and can now reuse it for other actors that require the same type of functionality.

The next step in our project is going to be learning more about Actor Components and how we can use them to our advantage. Let's take a look at those.

Actor Components

As we've seen in the first chapters of this book, Actors are the main way to create logic in UE4. However, we've also seen that Actors can contain several Actor Components.

Actor Components are objects that can be added to an Actor and can have multiple types of functionality, such as being responsible for a character's inventory or making a character fly. Actor Components must always belong to and live inside an Actor, which is referred to as their **Owner**.

There are several different types of existing Actor Components. Some of these are listed here:

- Code-only Actor Components, which act as their own class inside of an actor. They have their own properties and functions and can both interact with the Actor they belong to and be interacted with by it.
- Mesh Components, which are used to draw several types of Mesh objects (Static Meshes, Skeletal Meshes, and so on).
- Collision Components, used to receive and generate collision events.
- Camera Components.

This leaves us with two main ways to add logic to our Actors: directly in the **Actor** class or through **Actor Components**.

In order to follow good software development practices, namely loose coupling (mentioned previously), you should strive to use Actor Components instead of placing logic directly inside an Actor whenever possible. Let's take a look at a practical example in order to understand the usefulness of Actor Components.

Let's say you're making a game where you have the player character and enemy characters, both of which have health, and where the player character must fight enemies, who can also fight back. If you had to implement the health logic, which includes gaining health, losing health, and tracking the character's health, you'd have two options:

- You implement the health logic in a base character class, from which both the player character class and the enemy character class would inherit.
- You implement the health logic in an Actor Component and add that component to both the player character and enemy character classes separately.

There are a few reasons why the first option is not a good option, but the main one is this: If you wanted to add another piece of logic to both character classes (for example, stamina, which would limit the strength and frequency of the characters' attacks), doing so using the same approach of a base class wouldn't be a viable option. Given that in UE4, C++ classes can only inherit from one class and there's no such thing as multiple inheritance, that would be very hard to manage. It would also only get more complicated and unmanageable the more logic you decided to add to your project.

With that said, when adding logic to your project can be encapsulated in a separate component, allowing you to achieve Loose Coupling, you should always do so.

Let's now create a new Actor Component, which will be

responsible for keeping track of an actor's health, as well as gaining and losing that health. We will do this in the next exercise.

Exercise 7.02: Creating the HealthComponent Actor Component

In this exercise, we will be creating a new actor component responsible for gaining, losing, and keeping track of an actor's health (its **Owner**).

In order for the player to lose, we'll have to make the player character lose health and then end the game when it runs out of health. We'll want to put this logic inside an actor component so that we can easily add all this health-related logic to other actors if we need to.

The following steps will help you complete the exercise:

1. Open the editor and create a new C++ class, whose parent class will be the **ActorComponent** class. Its name will be **HealthComponent**.
2. After this class has been created and its files have been opened in Visual Studio, go to its header file and add a **protected float** property called **Health**, which will keep track of the Owner's current health points. Its

default value can be set to the number of health points its **Owner** will start the game with. In this case, we'll initialize it with a value of **100** health points:

```
// The Owner's initial and  
current amount health  
points  
  
UPROPERTY(EditDefaultsOnly,  
Category = Health)  
  
float Health = 100.f;
```

3. Create a declaration for the function responsible for taking health away from its **Owner**. This function should be **public**; return nothing; receive a **float Amount** property as input, which indicates how many health points its **Owner** should lose; and be called **LoseHealth**:

```
// Take health points from  
its Owner  
  
void LoseHealth(float  
Amount);
```

Now, in the class's source file, let's start by notifying it that it should never use the **Tick** event so that its

performance can be slightly improved.

4. Change the **bCanEverTick** property's value to **false** inside the class's constructor:

```
PrimaryComponentTick.bCanEverTick = false;
```

5. Create the implementation for our **LoseHealth** function, where we'll start by removing the **Amount** parameter's value from our **Health** property:

```
void  
UHealthComponent::LoseHealth(float Amount)  
{  
    Health -= Amount;  
}
```

6. Now, in that same function, we'll check whether the current amount of health is less than or equal to **0**, which means that it has run out of health points (*has died or been destroyed*):

```
if (Health <= 0.f)
```

```
{  
}
```

7. If the **if** statement is true, we'll do the following things:

1. Set the **Health** property to **0**, in order to make sure that our **Owner** doesn't have negative health points:

```
    Health = 0.f;
```

2. Quit the game, the same way we did in *Chapter 6, Collision Objects*, when creating the **VictoryBox** class:

```
UKismetSystemLibrary  
::QuitGame(this,
```

```
    nullptr,
```

```
    EQuitPrefe  
rence::Quit,
```

```
    true);
```

3. Don't forget to include the **KismetSystemLibrary**

object:

```
#include  
"Kismet/KismetSystem  
Library.h"
```

With this logic done, whenever any actor that has **HealthComponent** runs out of health, the game will end. This isn't exactly the behavior we want in our **Dodgeball** game. However, we'll change it when we talk about Interfaces later in this chapter.

In the next exercise, we'll be making the necessary modifications to some classes in our project in order to accommodate our newly created **HealthComponent**.

Exercise 7.03: Integrating the HealthComponent Actor Component

In this exercise, we will be modifying our **DodgeballProjectile** class in order for it to damage the player's character when it comes into contact with it, and the **DodgeballCharacter** class, in order for it to have a Health Component.

Open the **DodgeballProjectile** class's files in Visual Studio and make the following modifications:

1. In the class's header file, add a **protected float** property called

Damage and set its default value to **34**, so that our player character will lose all of its health points after being hit 3 times. This property should be a **UPROPERTY** and have the **EditAnywhere** tag so that you can easily change its value in its Blueprint class:

```
// The damage the dodgeball  
will deal to the player's  
character  
  
UPROPERTY(EditAnywhere,  
Category = Damage)  
  
float Damage = 34.f;
```

In the class's source file, we'll have to make some modifications to the **OnHit** function.

2. Since we'll be using the **HealthComponent** class, we'll have to add the **include** statement for it:

```
#include  
"HealthComponent.h"
```

3. The existing cast that is being done for **DodgeballCharacter** from the **OtherActor** property, which we did

in *Step 17 of Exercise 6.01, Creating the Dodgeball Class*, and is inside the **if** statement, should be done before that **if** statement and be saved inside a variable. You should then check whether that variable is **nullptr**. We are doing this in order to access the player character's Health Component inside the **if** statement:

```
ADodgeballCharacter* Player  
= Cast<ADodgeballCharacter>  
(OtherActor);  
  
if (Player != nullptr)  
{  
}  
}
```

4. If the **if** statement is true (that is, if the actor we hit is the player's character), we want to access that character's **HealthComponent** and reduce the character's health. In order to access **HealthComponent**, we have to call the character's **FindComponentByClass** function and send the **UHealthComponent** class as a template parameter (in order to indicate the class of the component

we want to access):

```
UHealthComponent*  
HealthComponent = Player->  
FindComponentByClass<UHealthComponent>();
```

Note

*The **FindComponent(s) ByClass** function, included in the **Actor** class, will return a reference(s) to the actor component(s) of a specific class that the actor contains. If the function returns **nullptr**, that means the actor doesn't have an Actor Component of that class.*

*You may also find the **GetComponents** function inside the **Actor** class useful, which will return a list of all the Actor Components inside that actor.*

5. After that, check whether **HealthComponent** is **nullptr** and, if it isn't, we'll call its **LoseHealth** function and send the **Damage** property as a parameter:

```
if (HealthComponent !=  
nullptr)  
{  
    HealthComponent -  
    >LoseHealth(Damage);  
  
}  
  
Destroy();
```

6. Make sure the existing **Destroy** function is called after doing the null check for **HealthComponent**, as shown in the previous code snippet.

Before we finish this exercise, we'll need to make some modifications to our **DodgeballCharacter** class.

Open the class's files in Visual Studio and follow these steps.

7. In the class's header file, add a **private** property of type **class UHealthComponent*** called **HealthComponent**:

```
class UHealthComponent*  
HealthComponent;
```

8. In the class's source file, add an

include statement to the **HealthComponent** class:

```
#include  
"HealthComponent.h"
```

9. At the end of the class's constructor, create **HealthComponent**, using the **CreateDefaultSubobject** function, and name it **HealthComponent**:

```
HealthComponent =  
CreateDefaultSubobject<UHea  
lthComponent>(TEXT("Health  
Component"));
```

After you've made all these changes, compile your code, and open the editor. When you play the game, if you let your player character get hit by a dodgeball **3** times, you'll notice that the game abruptly stops, as intended:

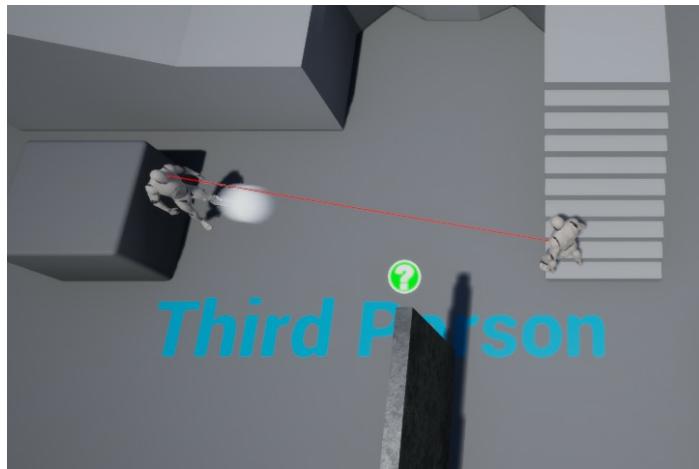


Figure 7.2: The enemy character throwing dodgeballs at the player character

Once the game is stopped, it will look like the following screenshot:

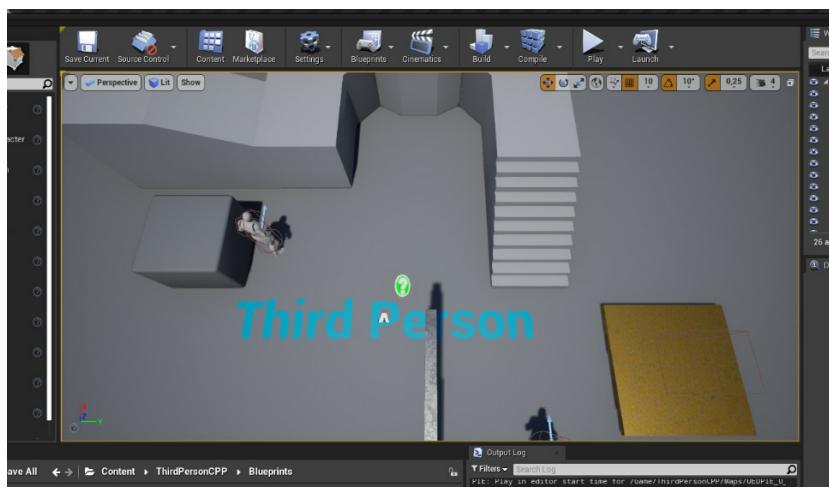


Figure 7.3: The editor after the player character runs out of health points and the game stops

And that completes this exercise. You now know how to create your own Actor Components and how to access an actor's Actor Components. This is a very important step toward making your game projects more understandable and better structured, so good job.

Now that we've learned about Actor Components, we will be learning about another way to make your projects better structured and organized – Interfaces – in the next section.

Interfaces

There's a chance that you might already know about Interfaces, given that other programming languages, such as Java, for instance, already have them. If you do, they work pretty similarly in UE4, but if you don't, let's see how they work, taking the example of the Health Component we created.

As you've seen in the previous exercise, when the **Health** property of the Health Component reaches **0**, that component will simply end the game. However, we don't want that to happen every time an actor's health points run out: some actors might simply be destroyed, some might notify another actor that they have run out of health points, and so on. We want each actor to be able to determine what happens to them when they run out of health points. But how can we handle this?

Ideally, we would simply call a specific function that belongs to the Owner of the Health Component, which would then choose how to handle the fact that the Owner has run out of health points, but in which class should you implement that function, given that our Owner can be of any class, as long as it inherits from the Actor class? As we discussed at the beginning of the chapter, having a class responsible just for this would quickly become unmanageable. Luckily for us, Interfaces solve this problem.

Interfaces are classes that contain a collection of functions that an object must have if it implements that Interface. It essentially works as a contract that the object signs, saying that it will implement all the functions present on that

Interface. You can then simply check whether an object implements a specific Interface and call the object's implementation of the function defined in the Interface.

In our specific case, we'll want to have an Interface that has a function that will be called when an object runs out of health points so that our Health Component can simply check whether its Owner implements that Interface and then call that function from the Interface. This will make it easy for us to specify how each actor behaves when running out of health points: some actors might simply be destroyed, others might trigger an in-game event, and others might simply end the game (which is the case with our player character).

However, before we create our first Interface, we should first talk a bit about Blueprint Native Events.

Blueprint Native Events

When using the **UFUNCTION** macro in C++, you can turn a function into a Blueprint Native Event by simply adding the **BlueprintNativeEvent** tag to that macro.

So what is a Blueprint Native Event? It's an event that is declared in C++ that can have a default behavior, which is also defined in C++, but that can be overridden in Blueprint. You declare a Blueprint Native Event called **MyEvent** by declaring a **MyEvent** function using the **UFUNCTION** macro with the **BlueprintNativeEvent** tag, followed by the **virtual MyEvent_Implementation** function:

```
UFUNCTION(BlueprintNativeEvent)
void MyEvent();
virtual void MyEvent_Implementation();
```

The reason why you have to declare these two functions is that

the first one is the Blueprint signature, which allows you to override the event in Blueprint, while the second one is the C++ signature, which allows you to override the event in C++.

The C++ signature is simply the name of the event followed by **_Implementation**, and it should always be a **virtual** function. Given that you declared this event in C++, in order to implement its default behavior, you have to implement the **MyEvent_Implementation** function, and not the **MyEvent** function (that one should remain untouched). In order to call a Blueprint Native Event, you can simply call the normal function, without the **_Implementation** suffix; in this case, **MyEvent()**.

We will take a look at how to use Blueprint Native Events in practice in the next exercise, where we'll create a new Interface.

Exercise 7.04: Creating the HealthInterface Class

In this exercise, we will be creating the Interface responsible for handling how an object behaves when it runs out of health points.

In order to do this, follow these steps:

1. Open the editor and create a new C++ class that inherits from **Interface** (called **Unreal Interface** in the scrollable menu) and call it **HealthInterface**.

2. After the class's files have been generated and opened in Visual Studio, go to the newly created class's header file. You'll notice that the generated file has two classes, **UHealthInterface** and **IHealthInterface**.
3. These will be used in combination when checking whether an object implements the interface and calling its functions. However, you should only add function declarations in the class prefixed with **I**, in this case, **IHealthInterface**. Add a **public** Blueprint Native Event called **OnDeath** that returns nothing and receives no parameters. This is the function that is going to be called when an object runs out of health points:

```
UFUNCTION(BlueprintNativeEvent, Category = Health)  
void OnDeath();  
  
virtual void  
OnDeath_Implementation() =  
0;
```

Note that the **OnDeath_Implementation** function

declaration needs its own implementation. However, there is no need for the Interface to implement that function because it would simply be empty. In order to notify the compiler that this function has no implementation in this class, we add = 0 to the end of its declaration.

4. Go to the **DodgeballCharacter** class's header file. We'll want this class to implement our newly created **HealthInterface**, but how do we do that? The first thing we have to do is include the **HealthInterface** class. Make sure you include it before the **.generated.h include** statement:

```
// Add this include  
  
#include  
"HealthInterface.h"  
  
#include  
"DodgeballCharacter.generated.h"
```

5. Then, replace the line in the header file that makes the **DodgeballCharacter** class inherit

from the **Character** class with the following line, which will make this class have to implement **HealthInterface**:

```
class ADodgeballCharacter :  
public ACharacter, public  
IHealthInterface
```

6. The next thing we have to do is implement the **OnDeath** function in the **DodgeballCharacter** class. In order to do this, add a declaration for the **OnDeath_Implementation** function that overrides the Interface's C++ signature. This function should be **public**. In order to override a **virtual** function, you have to add the **override** keyword to the end of its declaration:

```
virtual void  
OnDeath_Implementation()  
override;
```

7. In this function's implementation, within the class's source file, simply quit the game, the same way that is being done in the **HealthComponent** class:

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

8. Because we're now using **KismetSystemLibrary**, we'll have to include it:

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

9. Now, we have to go to our **HealthComponent** class's source file. Because we'll no longer be using **KistemSystemLibrary** and will be using the **HealthInterface** instead,

replace the **include** statement to the first class with an **include** statement to the second one:

```
// Replace this line  
  
#include  
"Kismet/KismetSystemLibrary  
.h"  
  
// With this line  
  
#include  
"HealthInterface.h"
```

10. Then, change the logic that is responsible for quitting the game when the **Owner** runs out of health points. Instead of doing this, we'll want to check whether the **Owner** implements **HealthInterface** and, if it does, call its implementation of the **OnDeath** function. Remove the existing call to the **QuitGame** function:

```
// Remove this  
  
UKismetSystemLibrary::QuitG  
ame(this,  
  
nullptr,
```

```
EQuitPreference::Quit,
```

```
true);
```

11. In order to check whether an object implements a specific interface, we can call that object's **Implements** function, using the Interface's class as a template parameter. The class of the Interface that you should use in this function is the one that is prefixed with **U**:

```
if (GetOwner() ->Implements<UHealthInterface>())  
{  
}
```

12. Because we'll be using methods belonging to the **Actor** class, we'll also need to include it:

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

If this **if** statement is true, that means that our **Owner** implements

HealthInterface. In this case, we'll want to call its implementation of the **OnDeath** function.

13. In order to do this, call it through the Interface's class (this time the one that is prefixed with **I**). The function inside the Interface that you'll want to call is **Execute_OnDeath** (note that the function you should call inside the Interface will always be its normal name prefixed with **Execute_**). This function must receive at least one parameter, which is the object that the function will be called on and that implements that Interface; in this case, **Owner**:

```
if (GetOwner()->Implements<UHealthInterface>())
{
    IHealthInterface::Execute
    _OnDeath(GetOwner());
}
```

Note

*If your interface's function receives parameters, you can send them in the function call after the first parameter mentioned in the last step. For instance, if our **OnDeath** function received an **int** property as a parameter, you would call it like this: **IHealthInterface::Execute_OnDeath(GetOwner(), 5).***

*The first time you try to compile your code after adding a new function to an Interface and then calling **Execute_version**, you may get an **Intellisense** error. You can safely ignore this error.*

After you've made all these changes, compile your code, and open the editor. When you play the game, try letting the character get hit by 3 dodgeballs:



Figure 7.4: The enemy character throwing dodgeballs

at the player character

If the game ends after that, then that means that all our changes worked and the game's logic remains the same:

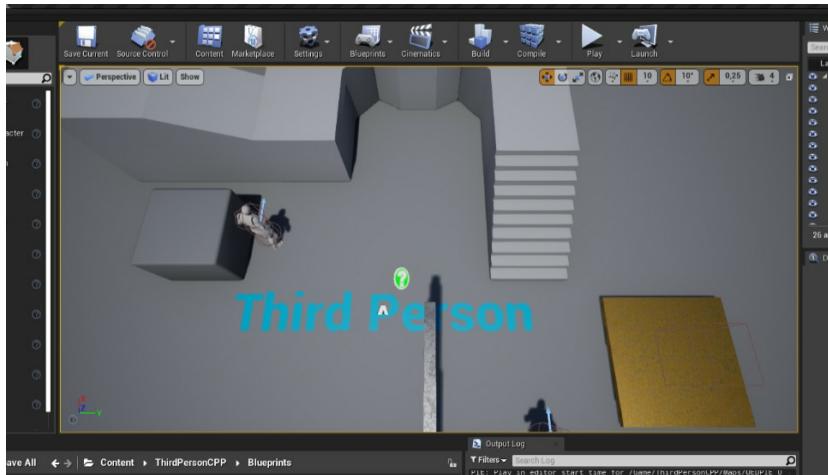


Figure 7.5: The editor after the player character runs out of health points and the game stops

And with that, we conclude this exercise. You now know how to use Interfaces. The benefit to the change that we just made is that we can now have other actors that lose health and can specify what happens when they run out of health points, using this **Health** Interface.

The next and last step of this chapter will be an activity where we'll move all of the logic related to the **LookAtActor** function to its own **Actor Component** and use it to replace the **SightSource** component we created.

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 (similarly to how we moved the **CanSeeActor** function to a Blueprint Function Library). This way, if we want an actor (that isn't an **EnemyCharacter**) to look at another actor, we will simply be able to add this component to it.

The following steps will help you complete this activity:

1. Open the editor and create a new C++ class that inherits from **SceneComponent**, called **LookAtActorComponent**.

Head to the class's files, which are opened in Visual Studio.

2. Go to its header file and add a declaration for the **LookAtActor** function, which should be **protected**, return a **bool**, and receive no parameters.

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 AAActor*** property called **TargetActor**. This property will represent the actor we want to look at.
4. Add a **protected bool** property called **bCanSeeTarget**, with a default value of **false**, which will indicate whether **TargetActor** can be seen.
5. Add a declaration for a **public FORCEINLINE** function, covered in *Chapter 6, Collision Objects* called **SetTarget**, which will return nothing and receive **AAActor* NewTarget** as a parameter. The implementation of this function will simply set the **TargetActor** property to the value of the **NewTarget** property.
6. Add a declaration for a **public FORCEINLINE** function called **CanSeeTarget**, which will be **const**, return a **bool**, and receive no parameters. The implementation of this function will simply return the

value of the **bCanSeeTarget** property.

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

7. In the class's **TickComponent** function, set the value of the **bCanSeeTarget** property to the return value of the **LookAtActor** function call.
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**.
9. Make the following modifications to the **LookAtActorComponent** class's implementation of the **LookAtActor** function:
 1. Change the first element of the **IgnoreActors** array to be the **Owner** of the Actor Component.
 2. Change the second parameter

of the **CanSeeActor** function call to be this component's location.

3. Change the value of the **Start** property to be the location of **Owner**.
4. Finally, replace the call to the **SetActorRotation** function with a call to the **SetActorRotation** function of **Owner**.
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.

We'll also need to add an include to the **Actor** class since we'll be accessing

several functions belonging to that class.

Now make some further modifications to our **EnemyCharacter** class:

11. In its header file, remove the declaration of the **LookAtActor** function.
12. Replace the **SightSource** property with a property of type **ULookAtActorComponent*** called **LookAtActorComponent**.
13. In the class's source file, add an include to the **LookAtActorComponent** class.
14. 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 and its parameter should be "**Look At Actor Component**".

15. Remove the class's implementation of the **LookAtActor** function.
16. In the class's **Tick** function, remove the line of code where you create the **PlayerCharacter** property, and add that exact line of code to the end of the class's **BeginPlay** function.
17. After this line, call the **SetTarget** function of **LookAtActorComponent** and send the **PlayerCharacter** property as a parameter.
18. Inside the class's **Tick** function, set the **bCanSeePlayer** property's value to the return value of the **CanSeeTarget** function call of **LookAtActorComponent**, instead of the return value of the **LookAtActor** function call.

Now, there's only one last step we have to do before this activity is completed.

19. Close the editor (if you have it opened), compile your changes in Visual Studio, open the editor, and open the **BP_EnemyCharacter** Blueprint. Find

LookAtActorComponent and change its location to **(10, 0, 80)**.

Expected output:

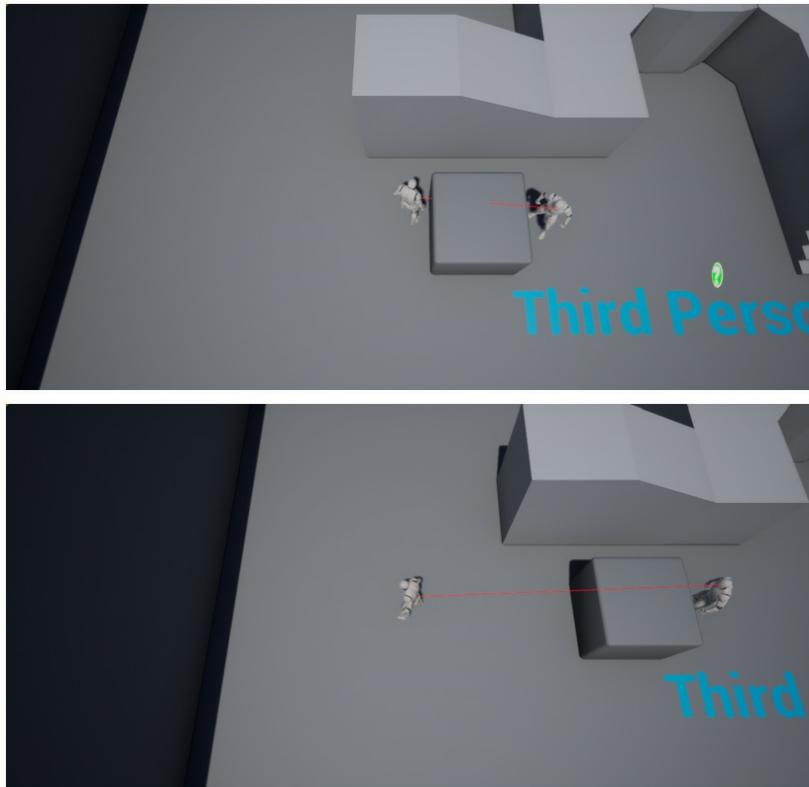


Figure 7.6: The enemy character's looking at the player character remains functional

And with that, we conclude our activity. You have now applied your knowledge of refactoring part of an actor's logic into an Actor Component so that you can reuse it in other parts of your project, or even in other projects of your own.

Note

*The solution to this activity can be found at:
<https://packt.live/338jEBx>.*

Summary

You now know about several utilities that will help you to keep your projects more organized and allow better reuse of the things that you make.

You have learned how to: create a Blueprint Function Library; create your own Actor Components and use them to refactor the existing logic in your project; and create Interfaces and call functions from an object that implements a specific Interface. Altogether, these new topics will allow you to refactor and reuse all the code that you write in a project in that same project, or in another project.

In the next chapter, we'll be taking a look at UMG, UE4's system for creating user Interfaces, and learning how to create our own user Interfaces.