



# Unreal Engine: Game Development from A to Z

Develop fantastic games and solve common development problems with Unreal Engine 4



**Packt**

LEARNING PATH

# Unreal Engine: Game Development from A to Z

---

# Table of Contents

[Unreal Engine: Game Development from A to Z](#)

[Unreal Engine: Game Development from A to Z](#)

[Credits](#)

[Preface](#)

[What this learning path covers](#)

[What you need for this learning path](#)

[Who this learning path is for](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Errata](#)

[Piracy](#)

[Questions](#)

## 1. Module 1

[1. An Overview of Unreal Engine](#)

[What goes into a game?](#)

[What is a game engine?](#)

[The history of Unreal Engine](#)

[Game development](#)

[Artists](#)

[Cinematic creators](#)

[Sound designers](#)

[Game designers](#)

[Programmers](#)

[The components of Unreal Engine 4](#)

[The sound engine](#)

[The physics engine](#)

[The graphics engine](#)

[Input and the Gameplay framework](#)

[Light and shadow](#)

[Post-process effects](#)

[Artificial intelligence](#)

[Online and multiplatform capabilities](#)

[Unreal Engine and its powerful editors](#)

[Unreal Editor](#)

[Material Editor](#)

[The Cascade particle system](#)

[The Persona skeletal mesh animation](#)

[Landscape – building large outdoor worlds and foliage](#)

[Sound Cue Editor](#)

[Matinee Editor](#)

[The Blueprint visual scripting system](#)

[Unreal programming](#)

[Unreal objects](#)

[A beginner's guide to the Unreal Editor](#)

[The start menu](#)

[Project Browser](#)

[Content Browser](#)

[Toolbar](#)

[Viewport](#)

[Scene Outliner](#)

[Modes](#)

[Summary](#)

## 2. Creating Your First Level

[Exploring preconfigured levels](#)

[Creating a new project](#)

[Navigating the viewport](#)

[Views](#)

[Control keys](#)

[Creating a level from a new blank map](#)

[Creating the ground using the BSP Box brush](#)

[Useful tip – selecting an object easily](#)

[Useful tip – changing View Mode to aid visuals](#)

[Adding light to a level](#)

[Useful tip – positioning objects in a level](#)

[Adding the sky to a level](#)

[Adding Player Start](#)

[Useful tip – rotating objects in a level](#)

[Viewing a level that's been created](#)

[Saving a level](#)

[Configuring a map as a start level](#)

[Adding material to the ground](#)

[Adding a wall](#)

[Duplicating a wall](#)

[Creating an opening for a door](#)

[Adding materials to the walls](#)

[Sealing a room](#)

[Adding props or a static mesh to the room](#)

[Adding Lightmass Importance Volume](#)

[Applying finishing touches to a room](#)

[Useful tip – using the drag snap grid](#)

[Summary](#)

### [3. Game Objects – More and Move](#)

[BSP Brush](#)

[Background](#)

[Brush type](#)

[Brush solidity](#)

[Static Mesh](#)

[BSP Brush versus Static Mesh](#)

[Making Static Mesh movable](#)

[Materials](#)

[Creating a Material in Unreal](#)

[Materials versus Textures](#)

[Texture/UV mapping](#)

[How to create and use a Texture Map](#)

[Multitexturing](#)

[A special form of texture maps – Normal Maps](#)

[Level of detail](#)

[Collisions](#)

[Collision configuration properties](#)

[Simulation Generates Hit Events](#)

[Generate Overlap Events](#)

[Collision Presets](#)

[Collision Enabled](#)

[Object Type](#)

[Collision Responses](#)

[Trace Responses](#)

[Object Responses](#)

[Collision hulls](#)

[Interactions](#)

[Static Mesh creation pipeline](#)

[Introducing volumes](#)

[Blocking Volume](#)

[Camera Blocking Volume](#)

[Trigger Volume](#)

[Nav Mesh Bounds Volume](#)

[Physics Volume](#)

[Pain Causing Volume](#)

[Kill Z Volume](#)

[Level Streaming Volume](#)

[Cull Distance Volume](#)

[Audio Volume](#)

[PostProcess Volume](#)

[Lightmass Importance Volume](#)

[Introducing Blueprint](#)

[Level Blueprint](#)

[Using the Trigger Volume to turn on/off light](#)

[Using Trigger Volume to toggle light on/off \(optional\)](#)

[Summary](#)

## [4. Material and Light](#)

[Materials](#)

[The Material Editor](#)

[The rendering system](#)

[Physical Based Shading Model](#)

[High Level Shading Language](#)

[Getting started](#)

[Creating a simple custom material](#)

[Creating custom material using simple textures](#)

[Using custom materials to transform the level](#)

[Rendering pipeline](#)

[Shaders](#)

[APIs – DirectX and OpenGL](#)

[DirectX](#)

[DirectX12](#)

[Pipeline state representation](#)

[Work submission](#)

[Resource access](#)

[Lights](#)

[Configuring a Point Light with more settings](#)

[Attenuation Radius](#)

[Intensity](#)

[Use Inverse Squared Falloff](#)

[Color](#)

[Adding and configuring a Spot Light](#)

[Inner cone and outer cone angle](#)

[Using the IES Profile](#)

[Downloading IES Light Profiles](#)

[Importing IES Profiles into the Unreal Engine Editor](#)

[Using IES Profiles](#)

[Adding and configuring a Directional Light](#)

[Example – adding and configuring a Sky light](#)

[Static, stationary, or movable lights](#)

[Common light/shadow definitions](#)

[Static Light](#)

[Stationary Light](#)

[Movable Light](#)

[Exercise – extending your game level \(optional\)](#)

[Useful tips](#)

[Guidelines](#)

[Area expansion](#)

[Part 1 – lengthening the current walkway](#)

[Part 2 – creating a big room \(living and kitchen area\)](#)

[Part 3 – creating a small room along the walkway](#)

[Part 4 – Creating a den area in the big room](#)

[Creating windows and doors](#)

[Part 1 – creating large glass windows for the dining area](#)

[Part 2 – creating an open window for the window seat](#)

[Part 3 – creating windows for the room](#)

[Part 4 – creating the main door area](#)

[Creating basic furniture](#)

[Part 1 – creating a dining table and placing chairs](#)

[Part 2 – decorating the sitting area](#)

[Part 3 – creating the window seat area](#)

[Part 4 – creating the Japanese seating area](#)

[Part 5 – creating the kitchen cabinet area](#)

[Summary](#)

## [5. Animation and AI](#)

[What is animation?](#)

[Understanding how to animate a 3D model](#)

[Preparing before animation](#)

[How is animation created?](#)

[What Unreal Engine 4 offers for animation in games](#)

## [Importing animation from Maya/3ds Max](#)

[Tutorial – importing the animation pack from Marketplace](#)

## [What can you do with Persona?](#)

[Tutorial – assigning existing animation to a Pawn](#)

## [Why do we need to blend animations?](#)

[Tutorial – creating a Blend Animation](#)

[Tutorial – setting up the Animation Blueprint to use a Blend Animation](#)

[AnimGraph](#)

[EventGraph](#)

## [Artificial intelligence](#)

### [Understanding a Behavior Tree](#)

[Exercise – designing the logic of a Behavior Tree](#)

[Example – creating a simple Behavior Tree](#)

[How to implement a Behavior Tree in Unreal Engine 4](#)

### [Navigation Mesh](#)

[Tutorial – creating a Navigation Mesh](#)

### [Tutorial – setting up AI logic](#)

[Creating the Blueprint AIController](#)

[Creating the Blueprint character](#)

[Adding and configuring Mesh to a Character Blueprint](#)

[Linking AIController to the Character Blueprint](#)

[Adding basic animation](#)

[Configuring AIController](#)

[Nodes to add in EventGraph](#)

[Adjusting movement speed](#)

[Creating the BlackBoardData](#)

[Adding a variable into BlackBoardData](#)

[Creating a Behavior Tree](#)

[Creating a simple BT using a Wait task](#)

[Using the Behavior Tree](#)

[Creating a custom task for the Behavior Tree](#)

[Using the PickTargetLocation custom task in BT](#)

[Replacing the Wait task with Move To](#)

### [Implementing AI in games](#)

## [Summary](#)

## [6. A Particle System and Sound](#)

### [What is a particle system?](#)

### [Exploring an existing particle system](#)

### [The main components of a particle system](#)

#### [Modules](#)

### [The design principles of a particle system](#)

#### [Research](#)

#### [The iterative creative process](#)

### [Example – creating a fireplace particle system](#)

#### [Crafting P\\_Fireplace](#)

[Observing the solo emitters of the system](#)

[Deleting non-essential emitters](#)

[Focusing on editing the Flame emitter](#)

[Looking at the complete particle system](#)

## [Sound and music](#)

### [How do we produce sound and music for games?](#)

#### [Audio quality](#)

#### [How are sounds recorded?](#)

#### [The Unreal audio system](#)

### [Getting audio into Unreal](#)

#### [The audio format](#)

#### [The sampling rate](#)

#### [Bit depth](#)

#### [Supported sound channels](#)

### [Unreal sound formats and terminologies](#)

### [The Sound Cue Editor](#)

#### [How to open the Sound Cue Editor](#)

#### [Exercise – importing a sound into the Unreal Editor](#)

#### [Exercise – adding custom sounds to a level](#)

#### [Configuring the Sound Cue Editor](#)

## Summary

### 7. Terrain and Cinematics

#### Introducing terrain manipulation

Exercise – creating hills using the Landscape tool

Landscape creation options

Multiple landscapes

Using custom material

Importing height maps and layers

Scale

The number of components

Section Size

#### Introducing cinematics

#### Why do we need cut scenes?

#### Cinematic techniques

Adjusted camera functions

Zoom

Field of view

Depth of field

#### Camera movement

Tilt

Pan

Dolly/track/truck

Pedestal

#### Capturing a scene

Lighting

Framing

Some framing rules

Shot types

Shot plan

#### Getting familiar with the Unreal Matinee Editor

#### Exercise – creating a simple matinee sequence

#### Summary

## 2. Module 2

### 1. Getting Acquainted with the UE4 Interface

#### Introduction

#### Installing UE4 and folder structure

How to do it...

There's more...

#### UI overview

Getting ready

How to do it...

#### Navigating the viewport

How to do it...

#### The Content Browser overview

Getting ready

How to do it...

#### Importing your own content

Getting ready

How to do it...

### 2. Level Design – Building Out Levels or Greyboxing

#### Introduction

#### Building a room

Getting ready

How to do it...

#### Building out a level

Getting ready

Some keyboard tips

Seeing double – duplicating elements

How to do it...

#### Applying materials to geometry brushes

Getting ready

How to do it...

#### Converting brushes to static meshes or volumes

Getting ready

How to do it...

### [3. Creating Quality Interior Environments](#)

[Introduction](#)

[Placing static meshes](#)

[Getting ready](#)

[How to do it...](#)

[Placing a particle system](#)

[Getting ready](#)

[How to do it...](#)

[Using Groups](#)

[Getting ready](#)

[How to do it...](#)

[Meshing an example map](#)

[Getting ready](#)

[How to do it...](#)

[Adding life to static meshes](#)

[Getting ready](#)

[How to do it...](#)

### [4. Building the Great Outdoors – Exterior Environments](#)

[Introduction](#)

[Creating a landscape](#)

[Getting ready](#)

[How to do it...](#)

[Building an exterior level using the Sculpt mode](#)

[Getting ready](#)

[How to do it...](#)

[Creating rivers with the Flatten tool](#)

[Getting ready](#)

[How to do it...](#)

[Placing trees and rocks using the Foliage tool](#)

[Getting ready](#)

[How to do it...](#)

[Streaming levels](#)

[Getting ready](#)

[How to do it...](#)

### [5. Lights, Camera, Action – Cinematics](#)

[Introduction](#)

[An introduction to Matinee](#)

[Getting ready](#)

[How to do it...](#)

[Creating an opening cutscene](#)

[Getting ready](#)

[How to do it...](#)

[Playing a Matinee via Blueprints](#)

[Getting ready](#)

[How to do it...](#)

[Preventing a player from moving via cinematic mode](#)

[Getting ready](#)

[How to do it...](#)

[See also](#)

### [6. Lighting and Shadows](#)

[Introduction](#)

[Lighting overview – learning the types of lights](#)

[Getting ready](#)

[How to do it...](#)

[Adding moveable lights – flashlight, part 1](#)

[Getting ready](#)

[How to do it...](#)

[Creating a Day/Night cycle](#)

[Getting ready](#)

[How to do it...](#)

[See also](#)

### [7. Art Pipeline – Working with Materials](#)

[Introduction](#)

[Creating a custom material](#)

[Getting ready](#)

[How to do it...](#)

### [Creating a mirror material](#)

[Getting ready](#)

[How to do it...](#)

### [Using Textures and normal maps with Materials](#)

[Getting ready](#)

[How to do it...](#)

### [Creating glowing materials with static emissive lighting](#)

[Getting ready](#)

[How to do it...](#)

### [Seeing through walls](#)

[Getting ready](#)

[How to do it...](#)

[See also](#)

## [8. Blueprint Scripting – Level Effects](#)

### [Introduction](#)

#### [Building a flickering light](#)

[Getting ready](#)

[How to do it...](#)

#### [Converting from Level to Class Blueprints](#)

[Getting ready](#)

[How to do it...](#)

#### [Using Trigger Volumes – opening a door using Matinee](#)

[Getting ready](#)

[How to do it...](#)

#### [Adding to an existing Blueprint – flashlight, part 2](#)

[Getting ready](#)

[How to do it...](#)

#### [Creating a Health/Damage system, part 1 – taking damage](#)

[Getting ready](#)

[How to do it...](#)

[See also](#)

## [9. C++ Programming – Gameplay](#)

### [Introduction](#)

[When to use C++/Blueprints](#)

#### [Setting up your development environment](#)

[Getting ready](#)

[How to do it...](#)

#### [Displaying text during runtime](#)

[Getting ready](#)

[How to do it...](#)

#### [Networking 101 – creating collectables with networking](#)

[Getting ready](#)

[How to do it...](#)

#### [Saving or loading games and keyboard input with C++](#)

[Getting ready](#)

[How to do it...](#)

#### [Creating custom blueprint nodes](#)

[Getting ready](#)

[How to do it...](#)

[See also](#)

## [10. User Interface](#)

### [Introduction](#)

#### [Creating a Health/Damage system, part 2 – creating a healthbar](#)

[Getting ready](#)

[How to do it...](#)

#### [Dynamic enemy healthbars](#)

[Getting ready](#)

[How to do it...](#)

#### [Creating a main menu](#)

[How to do it...](#)

#### [Animating a menu](#)

[Getting ready](#)

[How to do it...](#)

[See also](#)

## [11. Publishing and Deployment](#)

[Introduction](#)

[Packaging your project](#)

[Getting ready](#)

[How to do it...](#)

[Creating an installer for Windows](#)

[Getting ready](#)

[How to do it...](#)

## [3. Module 3](#)

### [1. Getting Started with Unreal 4](#)

[What to expect](#)

[System requirements](#)

[Downloading and installing UE4](#)

[The Windows directory structure](#)

[Windows DirectXRedit](#)

[Launcher](#)

[4.X folders](#)

[The Engine Launcher](#)

[News](#)

[Learn](#)

[Marketplace](#)

[Library](#)

[UE4 Links](#)

[Summary](#)

### [2. Launching Unreal Engine 4](#)

[Meet the Editor](#)

[The Unreal Project Browser](#)

[The user interface](#)

[The tab bar and the menu bar](#)

[The toolbar](#)

[Viewport](#)

[Modes](#)

[World Outliner](#)

[Content Browser](#)

[Details](#)

[Hotkeys and controls](#)

[Summary](#)

### [3. Building the Game – First Steps](#)

[Projects](#)

[Creating a new project](#)

[Opening an existing project](#)

[Project directory structure](#)

[Bloques](#)

[Concept](#)

[Controls](#)

[Creating the project for the game](#)

[BSP brushes](#)

[Default BSP brush shapes](#)

[Editing BSP brushes](#)

[Blocking out the rooms with BSP brushes](#)

[The first room](#)

[The second room](#)

[The third room](#)

[The fourth room](#)

[Content Browser](#)

[Migrating and importing assets](#)

[Importing assets](#)

[Migrating assets](#)

[Placing actors](#)

[Materials](#)

[The Material Editor](#)

[The tab and menu bar](#)

[The toolbar](#)

[The Palette panel](#)

[The Stats panel](#)

[The Details panel](#)  
[The Viewport panel](#)  
[The Graph panel](#)  
[Applying materials](#)  
[Creating the materials](#)

- [Pedestals](#)
- [Doors](#)
- [Key Cubes](#)
- [Decorative assets](#)

[Lighting](#)

- [Mobility](#)
- [Lighting up the environment](#)

[Summary](#)

**4. Using Actors, Classes, and Volumes**

[Basic classes](#)

- [Adding basic class actors to the game](#)
- [Placing the Player Start actor](#)
- [Adding triggers](#)
- [Room 1](#)
- [Room 2](#)
- [Room 3](#)
- [Room 4](#)

[Visual Effects](#)

- [Adding Visual Effect actors to the game – Post Process Volume](#)

[Volumes](#)

- [Adding Volumes to the game](#)
- [Lightmass Importance Volume](#)
- [Nav Mesh Bounds Volume](#)
- [Room 3](#)
- [Room 4](#)

[All Classes](#)

- [Adding actors from All Classes](#)
- [Camera](#)
- [Matinee actors](#)
- [Target Point](#)
- [Room 3](#)
- [Room 4](#)

[Summary](#)

**5. Scripting with Blueprints**

[How Blueprint works](#)

[The Level Blueprint user interface](#)

- [The tab and menu bars](#)
- [The toolbar](#)
- [The Details panel](#)
- [The Compiler Results panel](#)
- [My Blueprint panel](#)
- [The Event Graph](#)

[Using Level Blueprint in the game](#)

- [Key cube pickup and placement](#)

[The Blueprint class](#)

- [Creating a Blueprint class](#)
- [Viewport](#)
- [The Construction Script](#)
- [The Event Graph](#)

[Scripting basic AI](#)

[Summary](#)

**6. Using Unreal Matinee**

[What is Unreal Matinee?](#)

- [Adding Matinee actors](#)

[The Unreal Matinee user interface](#)

- [The tab and menu bar](#)
- [The toolbar](#)
- [The Curve Editor](#)
- [The Tracks panel](#)
- [The Details panel](#)

[Animating the door](#)

[Room 1](#)

[Room 2](#)

[A bridge for the AI character](#)

[Summary](#)

[7. Finishing Packaging and Publishing the Game](#)

[Adding the main menu using Unreal Motion Graphics](#)

[UMG Editor](#)

[The tab and menu bar](#)

[The toolbar](#)

[The Graph Editor](#)

[The Details panel](#)

[The Palette panel](#)

[The Hierarchy panel](#)

[The Animations panel](#)

[Creating the main menu](#)

[Installing the Android SDK](#)

[Setting up the Android device](#)

[Packaging the project](#)

[The Maps & Modes settings](#)

[The Packaging settings](#)

[The Android app settings](#)

[Building a package](#)

[Developer Console](#)

[ALL APPLICATIONS](#)

[APK](#)

[Store Listing](#)

[Content Rating](#)

[Pricing & Distribution](#)

[In-app Products](#)

[Services & APIs](#)

[GAME SERVICES](#)

[Game details](#)

[Linked apps](#)

[Events](#)

[Achievements](#)

[Leaderboards](#)

[Testing](#)

[Publishing](#)

[REPORTS](#)

[SETTINGS](#)

[ALERTS](#)

[Publishing your game](#)

[Activating Google services](#)

[Preparing the project for shipping](#)

[Uploading the game on the Play Store](#)

[Monetization methods](#)

[Mobile performance and optimization](#)

[Summary](#)

[A. What Next?](#)

[Learn](#)

[AnswerHub](#)

[Forums](#)

[Summary](#)

[A. Bibliography](#)

[Index](#)

# Unreal Engine: Game Development from A to Z

---

# **Unreal Engine: Game Development from A to Z**

Develop fantastic games and solve common development problems with Unreal Engine 4

A course in three modules



BIRMINGHAM - MUMBAI

# Unreal Engine: Game Development from A to Z

Copyright © 2016 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.

Published on: August 2016

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78712-328-1

[www.packtpub.com](http://www.packtpub.com)

# Credits

## Authors

Joanna Lee

John P. Doran,

Nitish Misra

## Reviewers

Michele Bertolini

Kyle Langley

Daniel Jonathan Valik

Katax Emperore

Dennis Glowacki

Devin Sherry

David Pol

Steve Santello

Nicola Valcasara

## Content Development Editor

Aishwarya Pandere

## Production Coordinator

Nilesh Mohite

# Preface

Considering the craze for playing games across the globe, have you ever wondered to create a game that will wow the players? Unreal Engine technology powers hundreds of games, and thousands of individuals have built careers and companies around skills developed using this engine. Unreal Engine provides very polished game development tools and capabilities that allow vast amounts of customization for almost any game that you can dream of. The Game Development using Unreal Engine course gives you an insight on the rich functionalities to create 2D and 3D games across multiple platforms.

## What this learning path covers

[Module 1](#), *Learning Unreal Engine Game Development*, explains how to create your first room using the Box Brush, add materials to texture the walls/floor, and learn how to place static objects to enhance the look of the room. It covers the structure of a simple object type, and describes how objects in Unreal interact with one another. This module shows you how to add the final touches to your level using terrain manipulation and cinematics.

[Module 2](#), *Unreal Engine Game Development Cookbook*, acquaints you with the UE4 Interface, starts you off with learning Unreal Engine 4's interface and explores the most commonly used aspects in development. It teaches readers about how to work with terrain, add trees and rocks, build rivers, and stream multiple levels in the game at once. This module introduces different types of lights in Unreal Engine 4 and on learning how to make them mobile like a flashlight the player can carry. It also explains how to create dynamic lighting with a day/night cycle.

[Module 3](#), *Learning Unreal Engine Android Game Development*, deals with BSP Brushes, teaches you how to block out the level using them, how to create materials in the material editor and the content browser, and how to use lights in UE4. It looks at the various types of classes offered in the Modes panel which includes Basic Classes, Visual Classes, and Volumes, and how to implement them into our game. This module also gives us an insight into the Matinee to create cut scenes and other animations essential for the game.

# What you need for this learning path

You will need to create a free account with Epic Games to start using Unreal Engine 4.

For developing projects with Unreal Engine 4, it is recommended that your computer has the following specs:

ff Desktop PC or Mac

ff Windows 7 64-bit or Mac OS X 10.9.2 or later

ff Quad-core Intel or AMD processor, 2.5 GHz or faster

ff NVIDIA GeForce 470 GTX or AMD Radeon 6870 HD series card or higher

ff 8 GB RAM

UE4 will run on desktops and laptops below these recommendations, but performance may be limited. While most of the recipes can be completed on a Mac or Windows, the writer has created them using a Windows machine.

You will require:

Unreal Engine 4

The Android Development Kit

The main aim for the third module is to use as few software as possible, so that you can only focus on the Engine, and do not have to first download dozens of other software and programs to get ready.

## **Who this learning path is for**

If you have a fancy for computer games and hanker after developing them with Unreal Engine or wish to level up with Android and iOS game development, then this course is ideal for you. Prior understanding of c++ is recommended.

## Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <[feedback@packtpub.com](mailto:feedback@packtpub.com)>, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at [www.packtpub.com/authors](http://www.packtpub.com/authors).

# Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

## Downloading the example code

You can download the example code files for this course from your account at <http://www.packtpub.com>. If you purchased this course elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at <https://github.com/PacktPublishing/Game-Development-using-Unreal-Engine>. We also have other code bundles from our rich catalog of books, videos, and courses available at <https://github.com/PacktPublishing/>. Check them out!

## Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the course in the search field. The required information will appear under the **Errata** section.

## Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <[copyright@packtpub.com](mailto:copyright@packtpub.com)> with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

## Questions

If you have a problem with any aspect of this course, you can contact us at <[questions@packtpub.com](mailto:questions@packtpub.com)>, and we will do our best to address the problem.

# **Part 1. Module 1**

***Learning Unreal Engine Game Development***

*A step-by-step guide that paves the way for developing fantastic games with Unreal Engine 4*

# Chapter 1. An Overview of Unreal Engine

First of all, thank you for picking up this book. I am sure you are excited to learn how to make your own game. In this chapter, I will run you through the different fundamental components in a game and what Unreal Engine 4 offers to help you make your dream game.

The following topics will be covered in this chapter:

- What is in a game?
- The history of **Unreal Engine ( UE )**
- How is game development done?
- The components of UE and its editors

## What goes into a game?

When you play a game, you probably are able to identify what needs to go into a game. In a simple PC shooting game example, when you press the left mouse button, the gun triggers. You see bullets flying, hear the sound of the gun and look around to see if you have shot anything. If you did hit something, for example, a wall, the target receives some form of damage.

As a game creator, we need to learn breakdown what we see in a game to figure out what we need for a game. A simple breakdown without going into too much detail: link the mouse click to the firing of the bullets, play a sound file that sounds like a gun firing, display sparks (termed as **particle effect**) near the barrel of the gun and the target shows some visible damage.

Bearing this example in mind, try visualizing and breaking any game down into its fundamental components. This will greatly help you in designing and creating a game level.

There is a lot going on behind the scenes when you are playing a game. With the help of Unreal Engine, the interaction of the many components has been designed and you will need to customize it for your own game. This is a huge time saver when you use an engine to create a game.

# What is a game engine?

What a game engine does is that it provides you with tools and programs to help you customize and build a game; it gives you a head-start in making your own game. Unreal Engine is one of the more popular choices in the market currently and it is free for anyone to use for development (royalties need to be paid only if your game makes a profit; visit <https://www.unrealengine.com/custom-licensing> for more information). Its popularity is mainly due to its extensive customizability, multiplatform capabilities, and the ability to create high quality AAA games with it. If you intend to start a career in game development, this is definitely one of the engines you want to start playing with and using to build your portfolio.

# The history of Unreal Engine

Before explaining what this amazingly powerful game engine can do and how it works, let us take a short trip back into the past to see how UE came about and how it has evolved into what we have today.

For gamers, you are probably familiar with the Unreal game series. Do you know how the first Unreal game was made? The engineers at Epic Games built an engine to help them create the very first Unreal game. Over the years, with the development of each generation the Unreal game series, more and more functionalities were added to the engine to aid in the development of the game. This, in turn, increased UE's capabilities and improved the game engine very quickly over the years.

In 1998, the first version of UE made the modding of a first player shooting game possible. You could replace Unreal content using your own and tweak the behavior of the **non-player characters (NPCs)**, also known as **bots** (players that are controlled by the computer through artificial intelligence) using UnrealScript. Then multiplayer online features were added into UE through the development of *Unreal Tournament*, which is an online game. This game also added PlayStation 2 to the list of compatible platforms in addition to the PC and Mac.

By 2002, UE had improved by leaps and bounds, bringing it into the next generation with the development of a particle system (a system to generate effects such as fog and smoke), static mesh tools (tools to manipulate objects), a physics engine (allows interaction between objects such as collisions) and a Matinee (a tool to create cut scenes, which is a brief, non interactive movie). This improvement saw to the development of the *Unreal Championship* and *Unreal Tournament 2003*. The release of *Unreal Championship* also added the Xbox game console to the list, with multiplayer capabilities in Xbox Live.

The development of Epic's next game *Unreal II: The Awakening* edged UE forward with an animation system and overall improvement with their existing engine. The development of faster Internet speeds in the early 2000s also increased the demand of multiplayer online gaming. *Unreal Tournament 2004* allowed players to engage in online battles with one another. This saw the creation of vehicles and large battlefields, plus improvements in online network capabilities. In 2005, the release of *Unreal Champion 2* on the Xbox game console reinforced UE capabilities on the Xbox console. It also saw the creation of a very important feature of a new third-person camera. This opened up greater possibilities in the types of games that could be created using the engine.

*Gears of War*, one of the most well-known franchises in the video games industry, pushed Epic Games to create and release the third version of its game engine, *Unreal Engine 3*, in 2006.

The improvement of the graphics engine used DirectX 9/10 to allow more realistic characters and objects to be made. The introduction of **Kismet**, which is a visual scripting system, allowed game and level designers to create game play logic for more engaging combat play without having to delve into writing codes. Platform capabilities of UE3 include Xbox360 and PlayStation 3 was added. There was a revamp in the light control and materials. UE3 also had a new physics engine. *Gears of War 2* released in 2008 saw the progressive improvements to UE3. In 2013, the *Gears of War Judgment* was released.

PC online gaming was also under the radar of Epic Game's developers. In 2009, Atlas Technology was released to be used in conjunction with UE to allow **massively multiplayer online games (MMOG)** to be created.

The increasing demand of mobile gaming also led to UE3 being pushed in the direction of increasing its supportability for various mobile platforms. All these advancements and technological capabilities have made UE3 the most popular version of Unreal Engine and it is still very widely used today.

UE3 dominated the market for 8 years until UE4 came along. UE4 was launched in 2014 and introduced the biggest change by replacing Kismet with the new concept of **Blueprint**. We will discuss more about the features of UE4 later in the chapter.

# **Game development**

Each game studio has its own set of processes to ensure the successful launch of its game. Game production typically goes through several stages before a game is launched. In general, there is a preproduction/planning, production stage, and postproduction stage. Most of the time is normally spent in the production stage.

Game development is an iterative process. The birth of an idea is the start of this process. The idea of the game must first be tested to see if it is actually fun to the target audience. This is done through prototyping the level quickly. Iterations of this prototype into a fully-fledged game can go from weeks to months to years.

The development team takes care of this iteration process. Everyone's contribution of the game throughout the development cycle directly affects the game and its success.

Development teams loosely consist of several specialized groups: artists (2D/3D modeler, animator), cinematic creators, sound designers, game designers, and programmers.

## **Artists**

They create all visible objects in the game from menu buttons to the trees in the game level. Some artists specialize in 3D modeling, while others are focused on animation. Artists make the game look beautiful and realistic. Artists have to learn how to import their created images/models, which are normally created first using other software such as 3DMax, Maya, and MODO into UE4. They would most likely need to make use of Blueprint to create certain custom behaviors for the game.

## **Cinematic creators**

Many cinematic experts are also trained artists. They have a special eye and creative skills to create short movie scenes/cut scenes. The Matinee tool in UE4 will be what they would be using most of the time.

## **Sound designers**

Sound designers have an acute sense of hearing and they are mostly musically trained. They work in the sound labs to create custom sounds/music for the game. They are in charge of importing sound files into UE4 to be played at suitable instances in the game. When using UE4, they would be spending most of their time using the Sound Cue Editor.

## **Game designers**

Designers determine what happens in the game, what goes on in the game, and what the game will be about. In the planning stage, most of the time will be spent in discussion, presentations, and documentation. In the production stage, they will oversee the game prototyping process to ensure that the game level is created as designed. Very often designers spend their time in the Unreal Editor to customize and fine-tune the level.

## **Programmers**

They are the group that looks into the technology and software the team needs to create the game. In pre-production, they are responsible for deciding which software programs are required and are capable of creating the game. They also have to ensure that the different software used are compatible with one another. Programmers also write codes to make the objects created by the artist come alive according to the idea that the designers came up with. They program the rules and functionality of the game. Some programmers are also involved in creating tools and research for the games. They are not directly involved in creating the game but instead are supporting the production pipeline. Games with extreme graphics usually have a team of researchers optimizing the graphics and creating more realistic graphics for the game. They spend most of their time in codes, probably coding in Visual Studio using C++. They are also able to modify and extend the features of UE4 to support the needs of the game that they are developing.

# The components of Unreal Engine 4

Unreal Engine is a game engine that helps you make games. Unreal Engine is made up of several components that work together to drive the game. Its massive system of tools and editors allows you to organize your assets and manipulate them to create the gameplay for your game.

Unreal Engine components include a sound engine, physics engine, graphics engine, input and the Gameplay framework, and an online module.

## The sound engine

The sound engine is responsible for having music and sounds in the game. Its integration into Unreal allows you to play various sound files to set the mood and add realism to the game. There are many uses for sounds in the game. Ambient sounds are constantly in the background. Sound effects can be repeated when needed or one-off and are triggered by specific events in the game.

In a forest setting, you can have a combination of bird sounds, wind, trees, and leaves rustling as the ambient sound. These individual sounds can be combined as a forest ambient sound and be constantly playing softly in the background when the game character is in the forest. Recurring sounds such as footprint sound files can be connected to the animation of the walking movement. One-time sound effects, such as the explosion of a particular building in the city, can be linked to an event trigger in the game. In Unreal, the triggering of the sounds is implemented through cues known as **Sound Cue**.

## The physics engine

In the real world, objects are governed by the laws of physics. Objects collide and are set in motion according to Newton's laws of motion. Attraction between objects also obeys the law of gravity and Einstein's theory of general relativity. In the game world, for objects to react similarly to real life, it has to have the same system built through programming. Unreal physics engine makes use of the PhysX engine, developed by NVIDIA, to perform calculations for lifelike physical interactions, such as collisions and fluid dynamics. The presence of this advanced physics engine in place allows us to concentrate on making the game instead of spending time making objects interact with the game world correctly.

## The graphics engine

For an image to show up on screen, it has to be rendered onto your display monitor (such as your PC/TV or mobile devices). The graphics engine is responsible for the output on your display by taking in information about the entire scene such as color, texture, geometry, the shadow of an individual object and lighting, and the viewpoint of a scene, and consider the cross-interaction of the factors that affect the overall color, light, shadow, and occlusion of the objects.

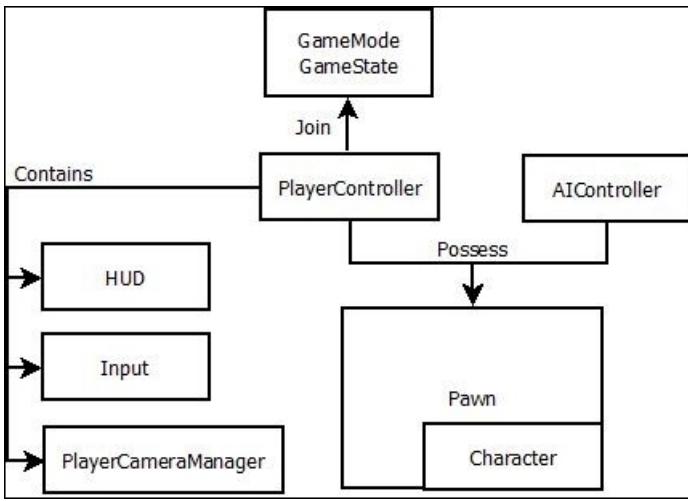
The graphics engine then undergoes massive calculations in the background using all these information before it is able to output the final pixel information to the screen. The power of a graphics engine affects how realistic your scene will look. Unreal graphics engine has the capabilities to output photorealistic qualities for your game. Its ability to optimize the scene and to process huge amount calculations for real-time lighting allows users to create realistic objects in the game.

This engine can be used to create games for all platforms (PC, Xbox, PlayStation, and mobile devices). It supports DirectX 11/12, OpenGL, and JavaScript/WebGL rendering.

## Input and the Gameplay framework

Unreal Engine consists of an input system that converts key and button presses by the player into actions performed by the in-game character. This input system can be configured through the Gameplay framework. The Gameplay framework contains the functionality to track game progress and control the rules of the game. **Heads-up displays ( HUDs ) / user interfaces ( UIs )** are part of the Gameplay framework to provide feedback to the player during the course of the game. Gameplay classes such as `GameMode`, `GameState`, and `PlayerState` set the rules and control the state of the game. The in-game characters are controlled either by players (using the `PlayerController` class) or AI (using `AIController` class). In-game characters, whether controlled by the player or AI, are part of a base class known as the **Pawn** class. The **Character** class is a subset of the **Pawn** class, which is specifically made for vertically-oriented player representation, for example, a human.

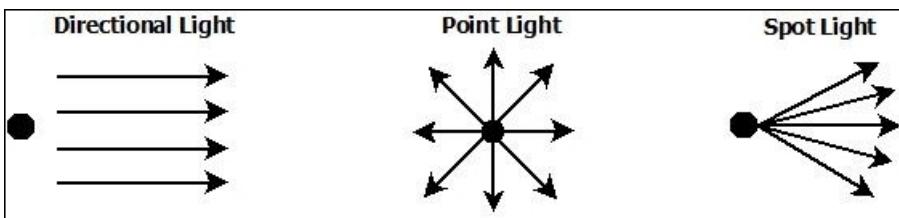
With the Unreal Gameplay framework and controllers in place, it allows for full customization of the player's behavior and flexibility, as shown in the following figure:



## Light and shadow

Light is a powerful tool in game creation. It can be used in many ways, such as to create the mood of a scene or focus a player's attention on objects in the game. Unreal Engine 4 provides a set of basic lights that could be easily placed in your game level. They are **Directional Light**, **Point Light**, **Spot Light**, and **Sky Light**.

Directional Light emits beams of parallel lights, Point Light emits light like a light bulb (from a single point radially outward in all directions), Spot Light emits light in a conical shape outwards, and Sky Light mimics light from the sky downwards on the objects in the level:



The effective design of light also creates realistic shadows for your game. By choosing the types of light in the level, you can affect both the mood and time it takes to render the scene, which in turn affects the frames per second of your game. In the game world, you can have two types of shadows: static and dynamic. Static shadows can be prebaked into the scene and, which makes them quick to render. Dynamic shadows are changed during runtime and are more expensive to render. We will learn more about lights and shadows in [Chapter 4, Material and Light](#).

## Post-process effects

Post-process effects are effects that are added at the end to improve the quality of the scene. Unreal Engine 4 provides a very good selection of post-process effects, which you can add to your level to accentuate the overall scene.

It offers full scene **high dynamic range rendering ( HDRR )**. This allows objects that are bright to be very bright and dark to be very dark, but we are still able to see details in them. (This is NVIDIA's motivation for HDR rendering.)

UE4 post-process effects include Anti-Aliasing using **Temporal Anti-Aliasing ( TXAA )**, Bloom, Color Grading, Depth of Field, Eye Adaptation, Lens Flare, Post Process Materials, Scene Fringe, Screen Space Reflection, and Vignette. Although a game is often designed with the post-process effects in mind, users are normally given the option to turn them off, if desired. This is because they often consume reasonable amount of additional resources in return for better visuals.

## Artificial intelligence

If you are totally new to the concept of **artificial intelligence ( AI )**, it can be thought of as intelligence created by humans to mimic real life. Humans created AI to give objects a brain, the ability to think, and make decisions on their own.

Fundamentally, AI is made up of complex rule sets that help objects make decisions and perform their designed function/behavior. In games, NPCs are given some form of AI so that players can interact with them. For example, give NPCs the ability to find a sweet spot to attack. If being attacked, they will run, hide, and find a better position to fight back.

Unreal Engine 4 provides a good basic AI and lays the foundation for you to customize and improve the AI of the NPCs in your game. More details on how AI is designed in Unreal Engine will be discussed in [Chapter 5, Animation and AI](#).

## Online and multiplatform capabilities

Unreal Engine 4 offers the ability to create game for many platforms. If you create a game using Unreal Engine 4, it is portable into different platforms, such as Web, iOS, Linux, Windows, and Android. Also, **Universal Windows Platform ( UWP )** will soon be added as well. It also has an online subsystem to provide games the ability to integrate functionalities that are available on Xbox Live, Facebook, Steam, and so on.

# Unreal Engine and its powerful editors

After learning about the different components of Unreal Engine, it is time to learn more about the various editors and how they are able to empower us with the actual functionalities to create a game.

## Unreal Editor

Unreal Engine has a number of editors that help in the creation of the game. By default, the Unreal Editor is the startup editor for Unreal Engine. It can be considered as the main editor that allows access to other subsystems, such as the Material and Blueprint subsystems.

The Unreal Editor provides a visual interface made up of viewports and windows to enable you to import, organize, edit, and add behaviors/interactions to your game assets. Other subeditors/subsystems have very specialized functions that allow you to control details of an asset (how it looks, how it behaves).

The Unreal Editor, together with all the subsystems, is a great tool especially for designers. It allows physical placement of assets and gives users the ability to control gameplay variables without having to make changes in the code.

## Material Editor

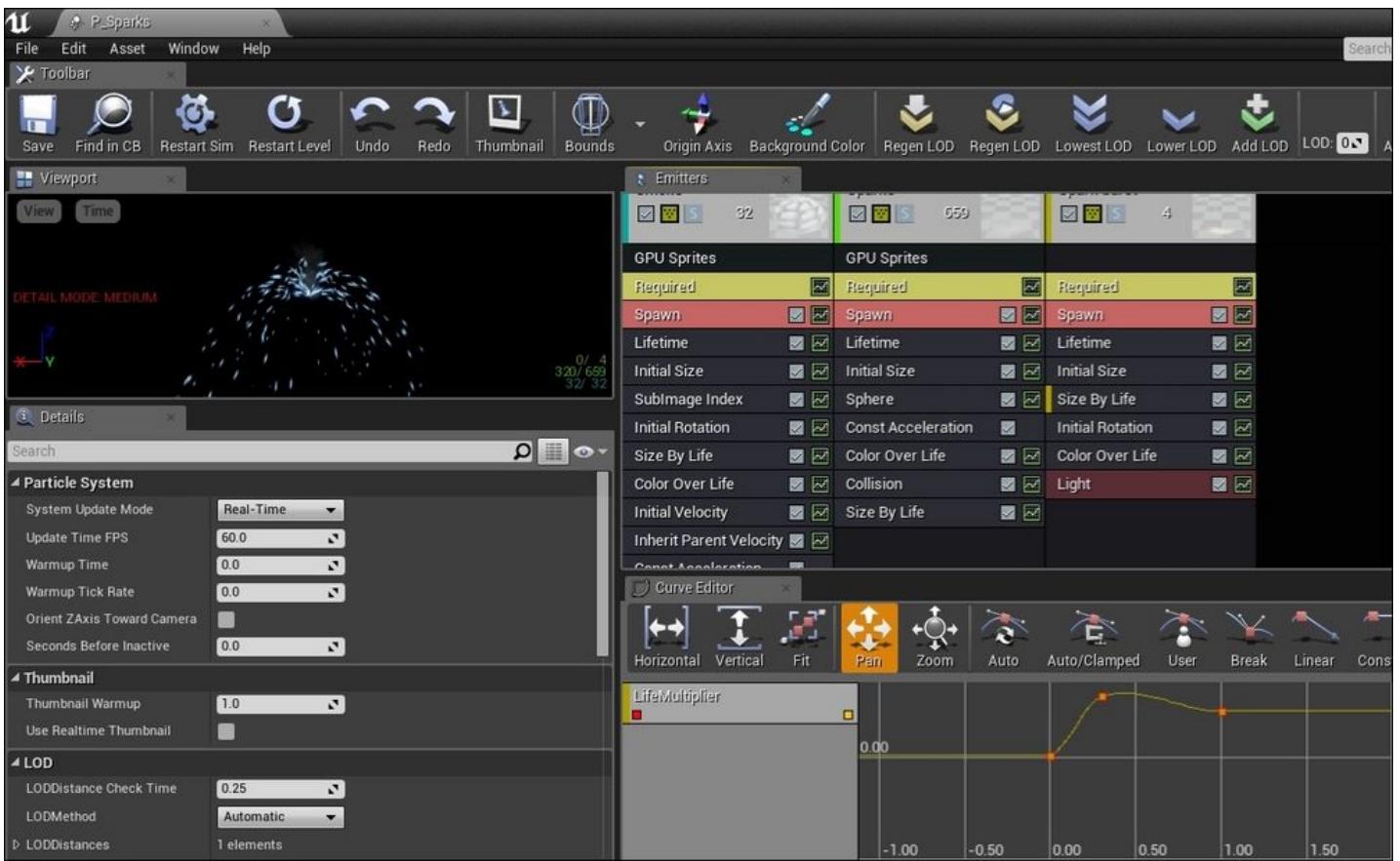
Shaders and Materials give objects its unique color and texture. Unreal Engine 4 makes use of physically-based shading. This new material pipeline gives artists greater control over the look and feel of an object. Physically-based shading has a more detailed relationship of light and its surface. This theory binds two physical attributes (micro surface detail and reflectivity) to achieve the final look of the object.

In the past, much of the final look is achieved by tweaking values in the shader/material algorithms. In Unreal Engine 4, we are now able to achieve high quality content by adjusting the values of the light and shading algorithms, which produces more consistent and predictable results. More details about Shaders and Materials will be provided in [Chapter 4 , Material and Light](#). The following screenshot shows the Material Editor in UE4:



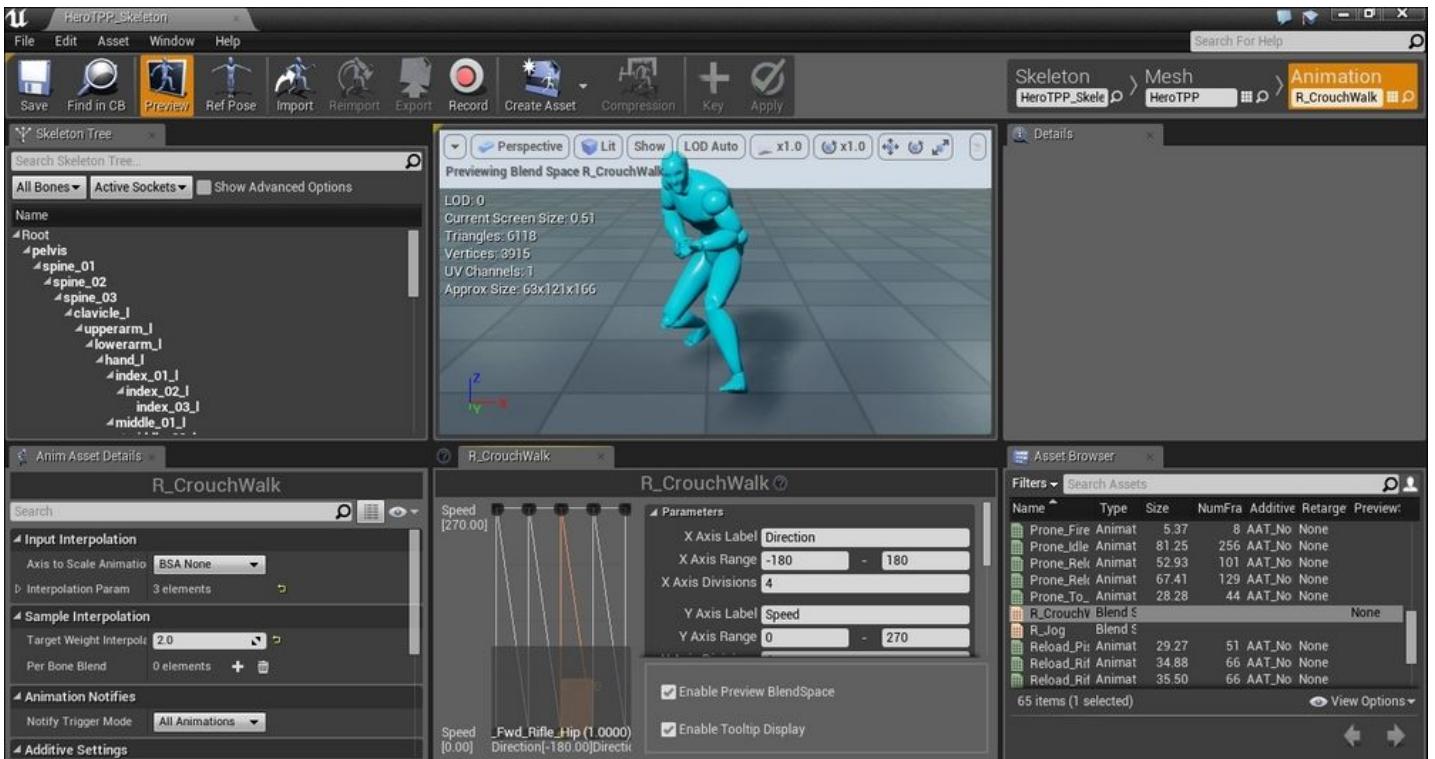
## The Cascade particle system

The Cascade particle system provides extensive capabilities to design and create particle effects. Effects from things such as smoke, sparks, and fire can be created by designing the size, color, and texture of each particle and how groups of these particles interact with each other to mimic real-life particle effect behavior. The following screenshot shows the Cascade particle system in UE4:



## The Persona skeletal mesh animation

The Persona animation system lets you design and control the animation of the skeleton, skeleton mesh, and sockets of a character. This tool can be used to preview a character's animation and set up blend animation between key frames. The physics and collision properties can also be adjusted through **Physics Asset Tool (PhAT)**. The following screenshot shows the Persona animation system in UE4:



## Landscape – building large outdoor worlds and foliage

To create large outdoor spaces using the editor, Unreal Engine provides sculpting and painting tools through the Landscape system to help us with it. An efficient level of detail (LOD) system and memory utilization allows large scaled terrain shaping. There is also a Foliage editor to apply grass, snow, and sand into the

outdoor environment.

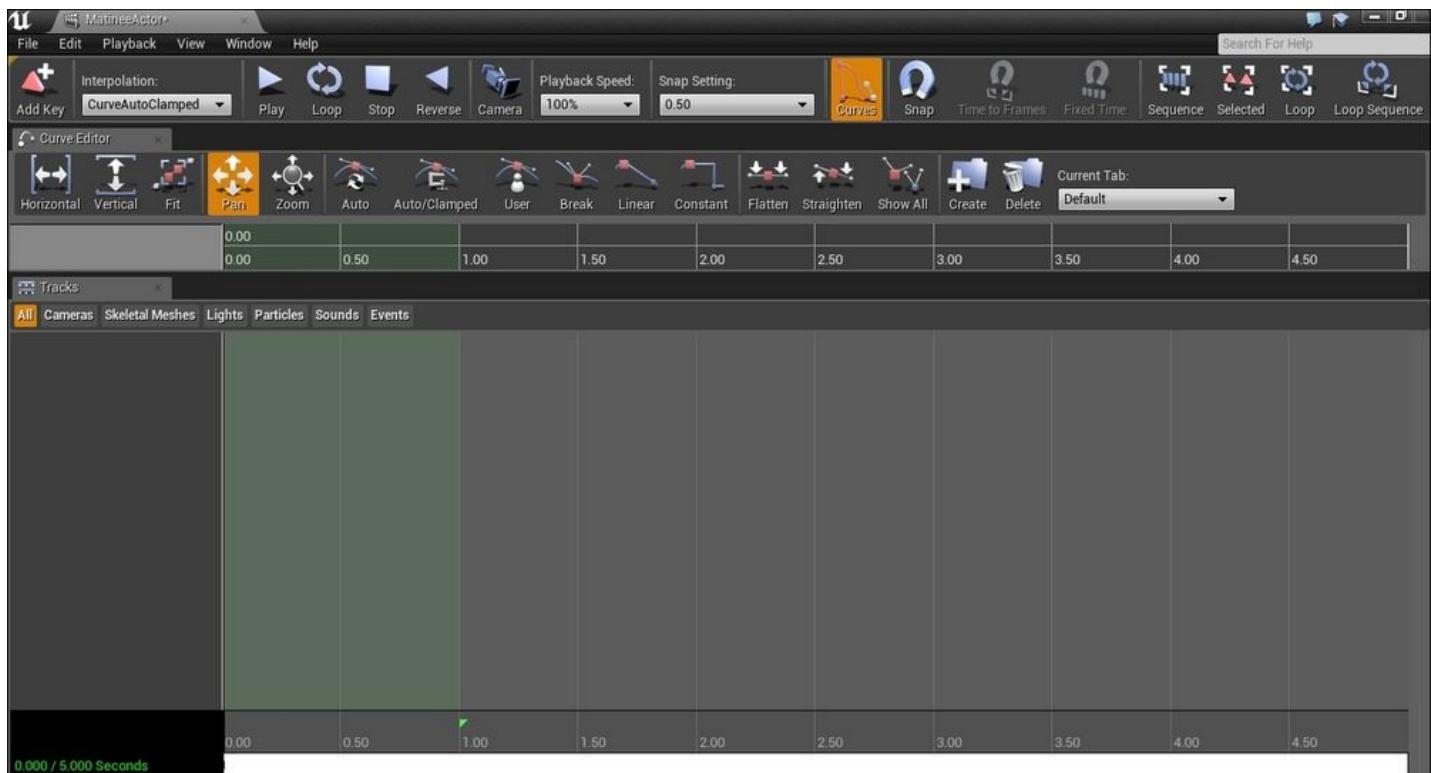
## Sound Cue Editor

The control of sound and music is done via the Sound Cue Editor. Sounds and music are triggered to play via cues known as **Sound Cues**. The ability to start/stop/repeat/fade in or out can be achieved using this editor. The following screenshot shows the Sound Cue Editor in UE4:



## Matinee Editor

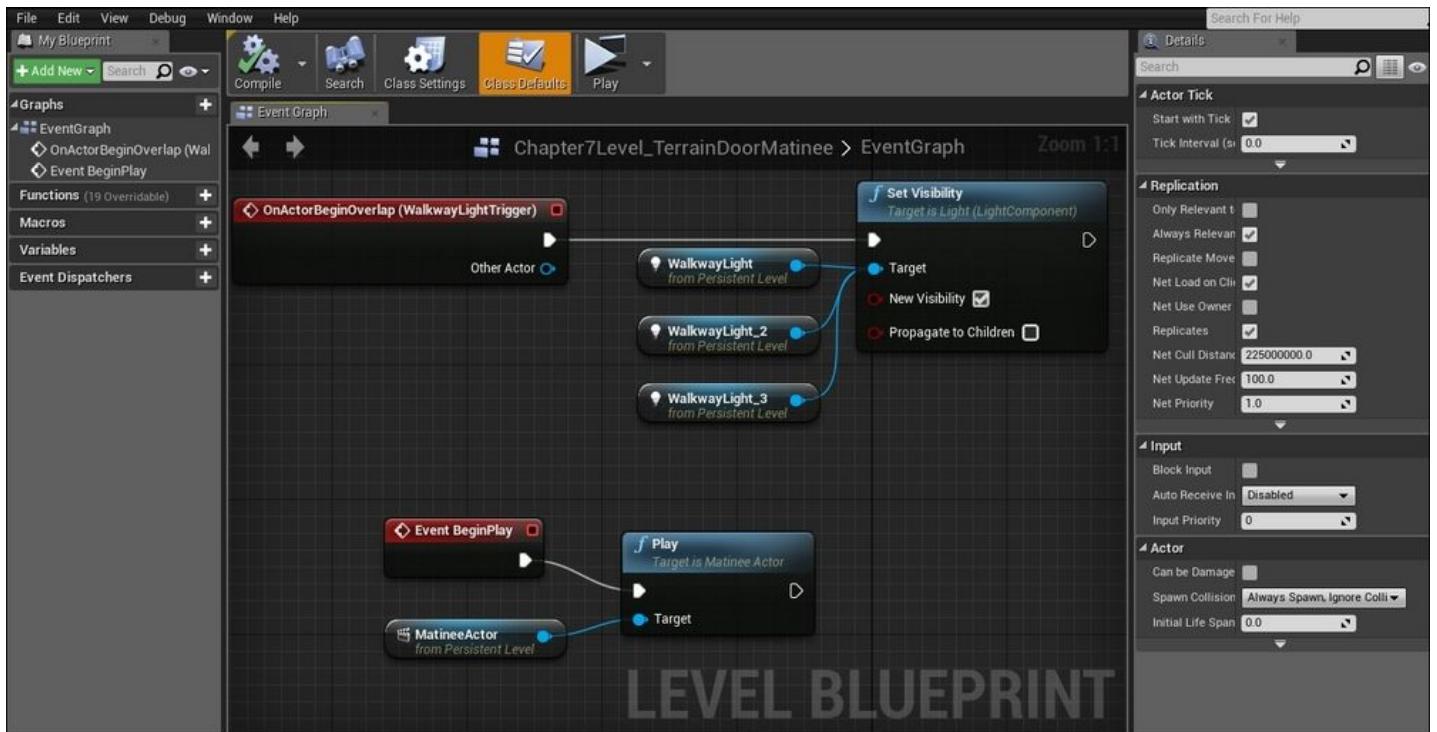
The Matinee Editor toolset enables the creation of game cut scenes and movies. These short clips created could be used to introduce the start of a game level, tell a story before the game begins or even as a promotional video for the game. The following screenshot shows the Matinee Editor in UE4:



## The Blueprint visual scripting system

The Blueprint system is a new feature in Unreal Engine. Unreal Engine 4 is the first engine to utilize this revolutionary system. For those who are familiar with Unreal Engine 3, it can be thought of as the enhanced and improved combined version of the Unreal scripting system, Kismet, and the Prefab functionality. The Blueprint visual scripting system enables you to extend code functionality using visual scripting language (box-like flow diagrams joined with lines). This capability means that you do not have to write or compile code in order to create, arrange, and customize behavior/interaction of in-game objects. This also provides nonprogrammers (artists/designers) with the ability to prototype or create a level quickly and manipulate gameplay without having to tackle the challenges of game programming. A cool feature of Blueprint is that you can create variables like in programming by clicking on the object and selecting **Create Variable**. This opens up what developers can do without messing around with complex coding.

To help developers debug Blueprint scripting logic, you can see the sequence of events and property values visually on the flow diagrams as it is being executed. Similar to troubleshooting in coding, break points can also be set to pause a Blueprint sequence. The following screenshot shows the Level Blueprint Editor in UE4:



# Unreal programming

The access to Unreal Engine's source code gives users the freedom to create almost about anything they can dream of. Functionalities of the base code can be extended and customized to create whatever the game needs to have. Learning how Unreal Engine works from the inside can unlock its full potential in game creation.

Unreal Engine has also incorporated very useful debugging features for the coding folks. One of them is the **Hot Reload** function. This tool enables changes in the C++ code to be reflected immediately in the game. To facilitate quick changes in code, Unreal Engine has also included **Code View**. By clicking on a function of an object in the **Code View** category, it shows you directly the relevant codes in Visual Studio where you could make code changes to the object.

Versioning and source control can be set up for game projects that include code changes.

## Unreal objects

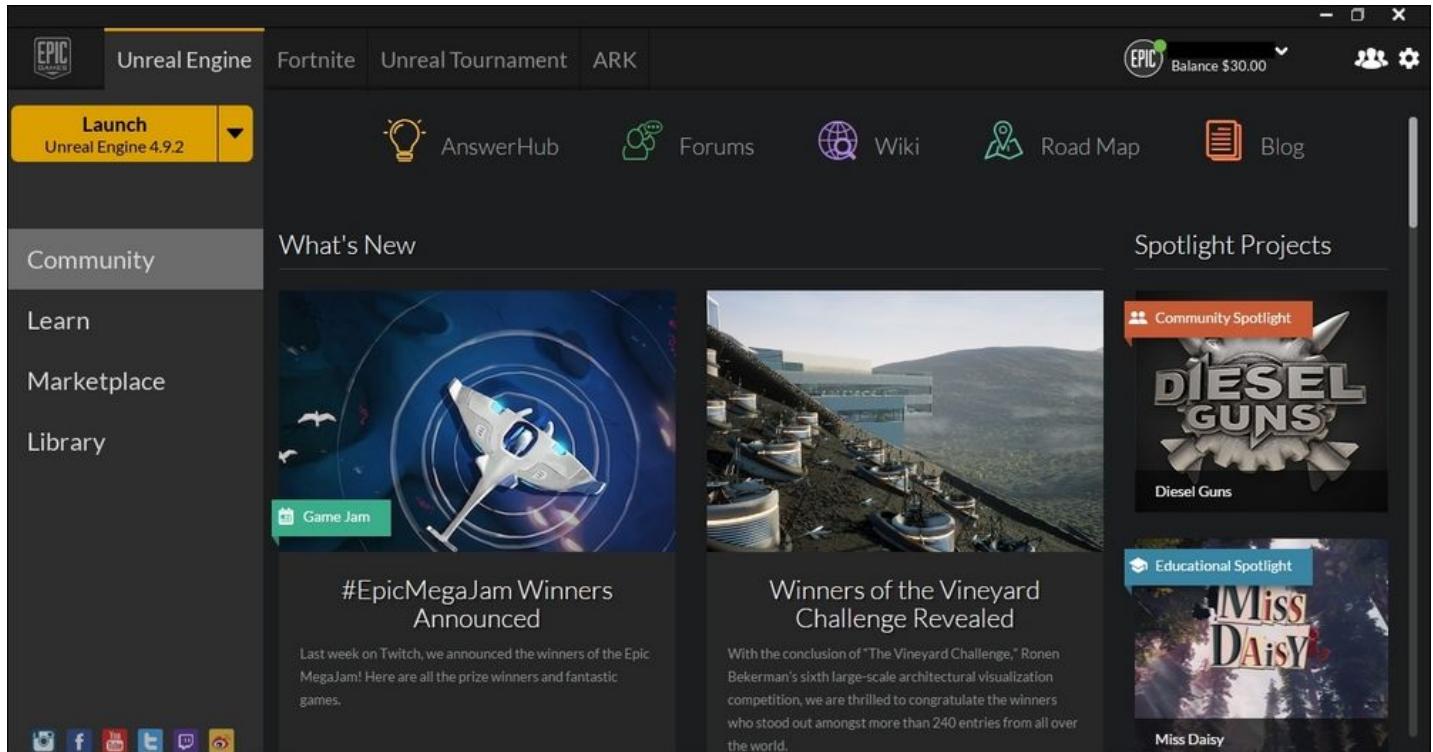
Actors are the base class of all gameplay objects in Unreal. For the Actors to have more properties and functionalities, the Actor class is extended to various more complex classes. In terms of programming, the Actor class acts as a container class to hold specialized objects called Components. The combination of the functionalities of the Components gives the Actor its unique properties.

# A beginner's guide to the Unreal Editor

This is a quick overview of what we can do with the Unreal Editor. We will briefly touch on how we can use the various windows in the editor to create a game.

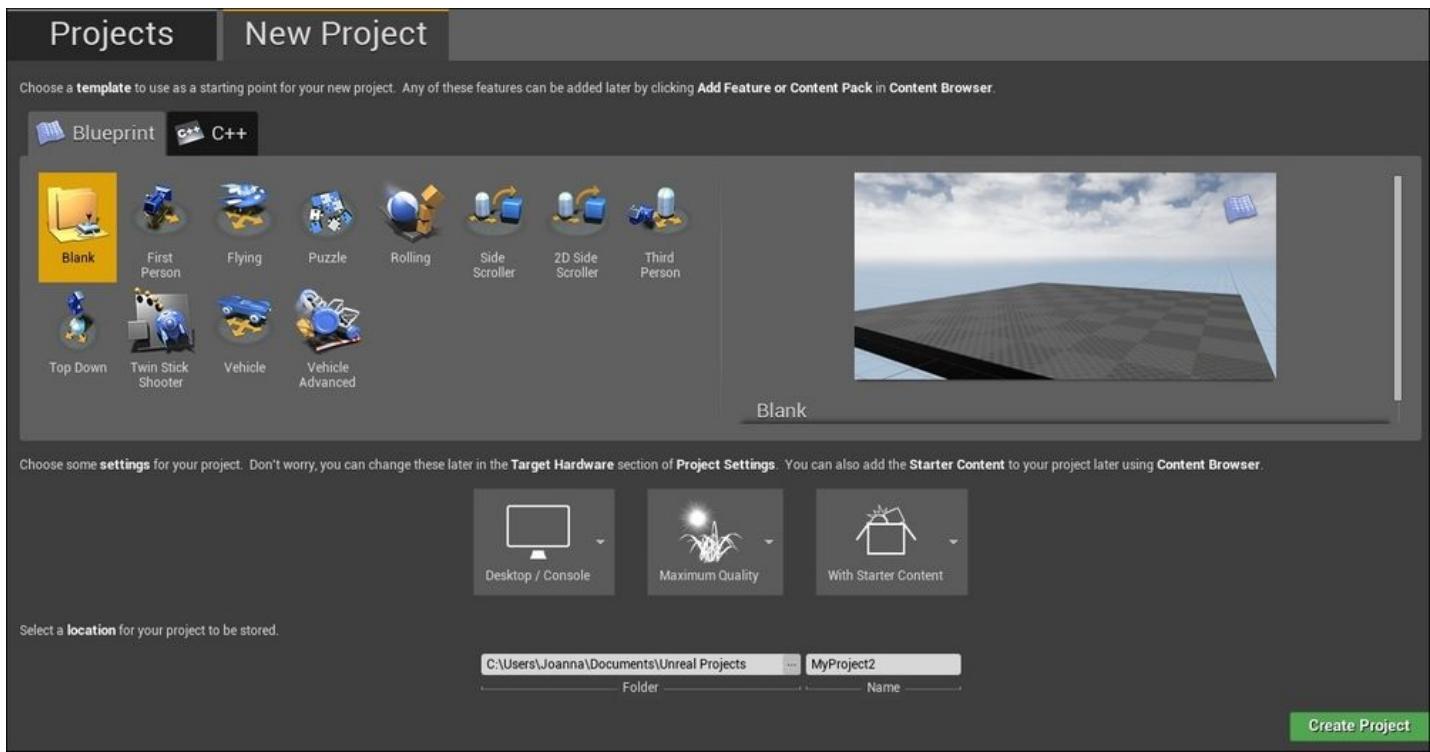
## The start menu

When starting up Unreal Engine, you will be first brought to a menu window by default. This new start menu is simple and easy to navigate. It features a large tab that allows you to select which version of game engine you want to launch and has a clear representation of the projects you have created. It also provides access to Marketplace, which is a library of game samples that are created by others, which you could download (some free, some paid). The menu also provides latest updates and news from Epic to ensure developers are kept abreast of the latest development and changes. The following screenshot shows the start menu:



## Project Browser

After launching the desired version of Unreal Engine, the Unreal Project Browser pops up. This browser provides you with the option to create game levels that have been pre-customized. This means that you have a list of generic levels, which you can start building your game levels with. For those who are new to game making, this feature lets you dive straight into building various types of games quickly. You can have a first-person shooting level and third-person game setup, or a 2D/3D side-scrolling platform level directly in either **Blueprint** or **C++** as the base template. What is so awesome about the **New Project** tab is that it also allows you to select your target device (PC/mobile), image quality target, with or without the Unreal content included in the startup project. The following screenshot shows the Project Browser:



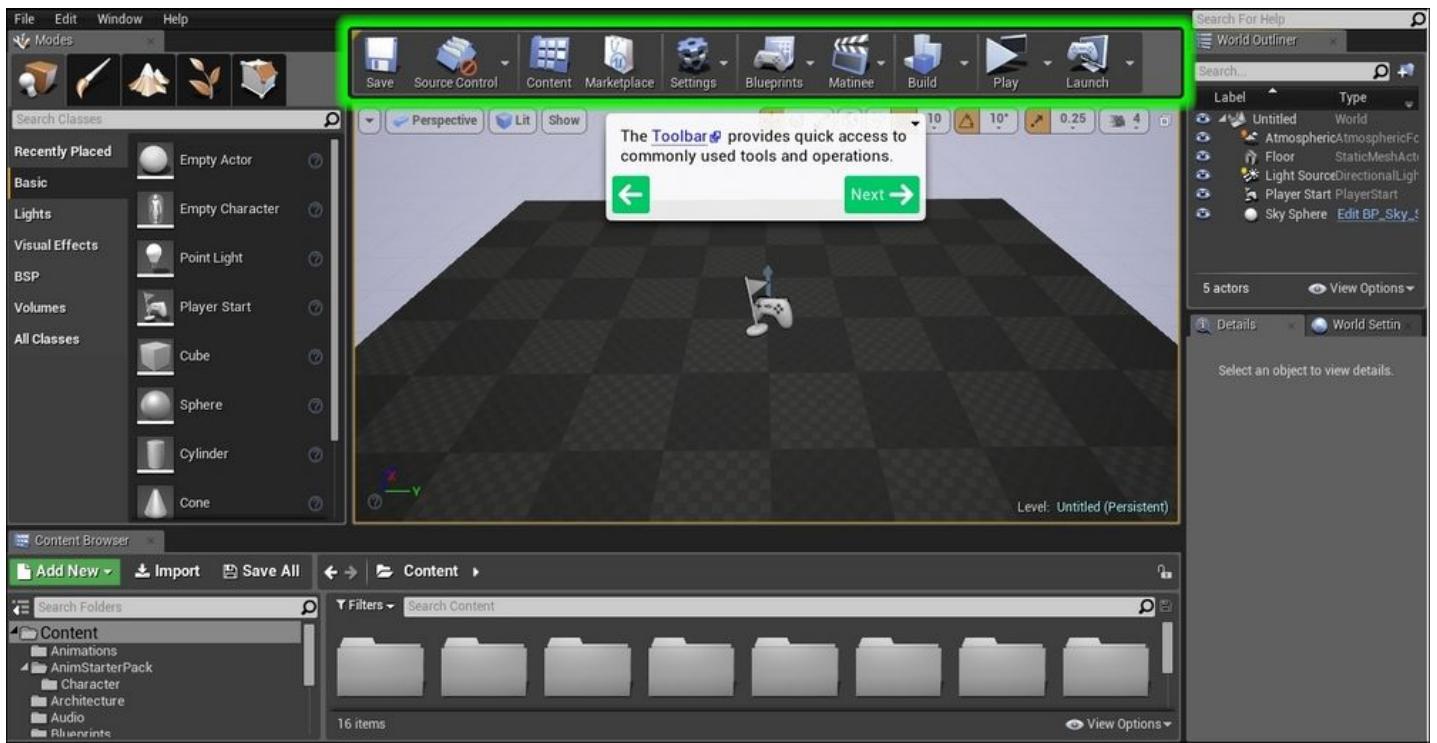
## Content Browser

When the Unreal Editor starts, there is a default layout of various windows and panels. One of them is the **Content Browser**. The **Content Browser** is a window where you can find all the content (game assets) that you have. It categorizes your assets into different folders such as **Audio**, **Materials**, **Animations**, **Particle Effects**, and so on. This window has also the **Import** button, which lets you bring in game assets that were created using other software into the game. The following screenshot shows the default location of the **Content Browser** (outlined in green):



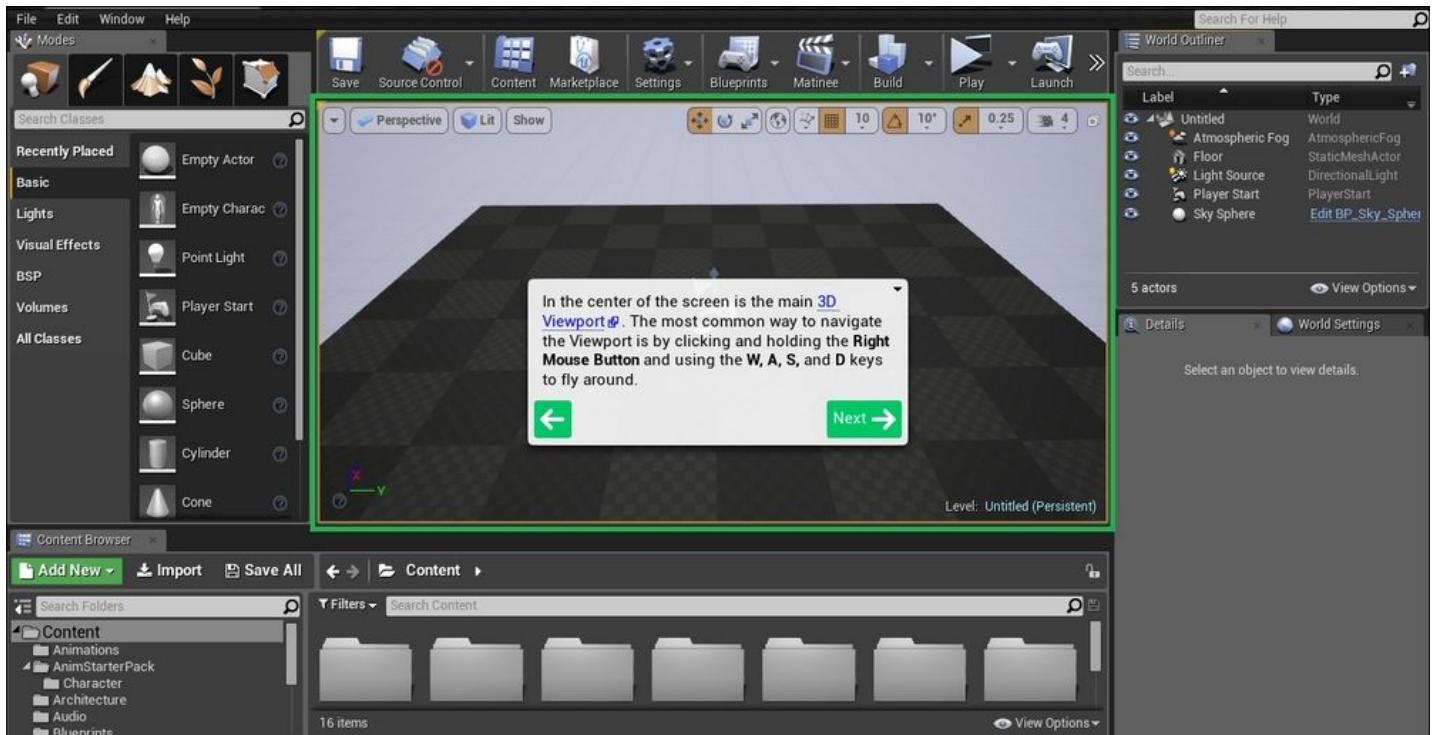
## Toolbar

The **Toolbar** is a customizable ribbon that provides quick access to tools and editors. The default layout includes quick access to the Blueprint and Matinee editors. Quick play and launch game function is also part of the standard ribbon layout. These buttons allow you to quickly view your creation in-game. The following screenshot shows the default **Toolbar**:



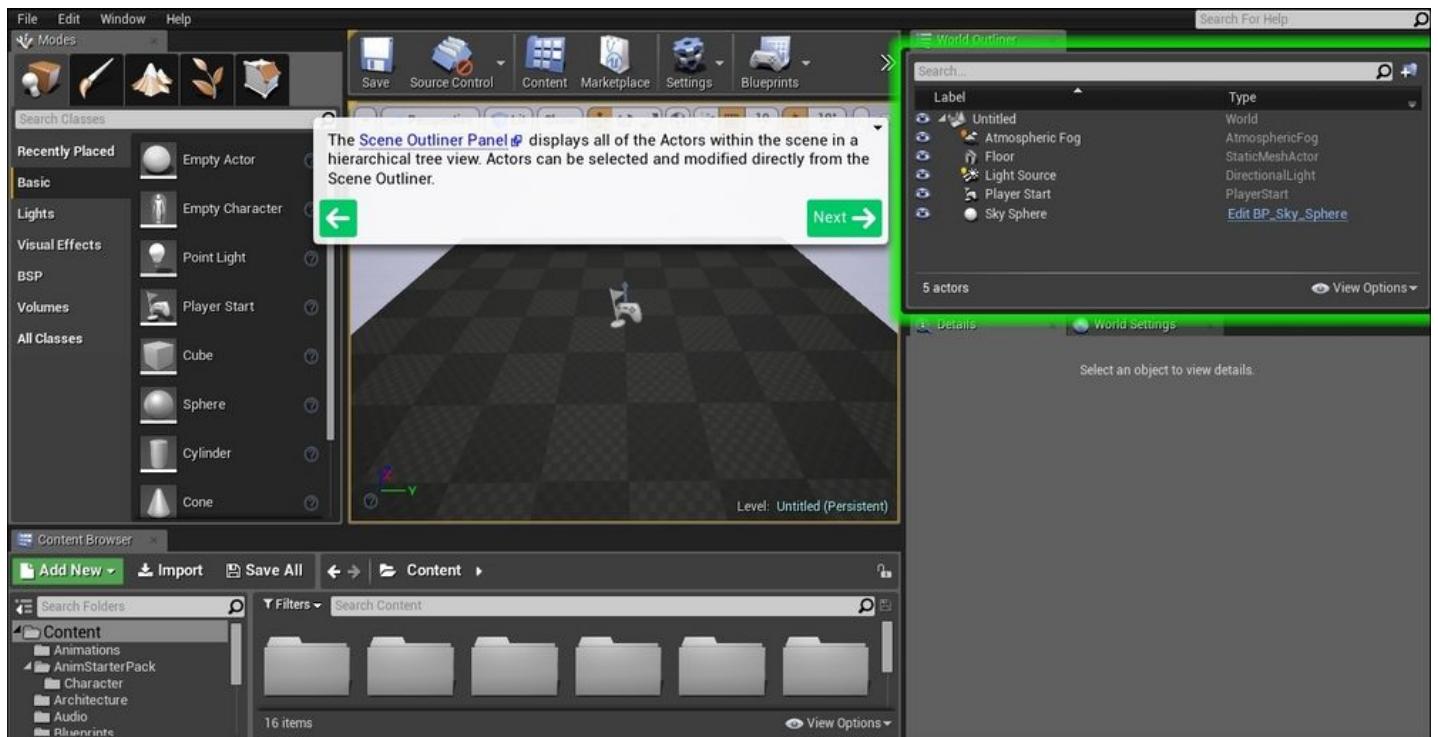
## Viewport

The **Viewport** is the window to the game world so what you see is what is in the game. If you have created a level using one of the options provided in the **New Project** menu, you would notice that the camera has been adjusted accordingly to the settings of that pre-customized level. This is the window that you will use to place objects into and move them around. When you click on the **Play** button in the toolbar, this **Viewport** window comes alive and allows you to interact with game level. The following screenshot shows the **Viewport** window being highlighted in the editor:



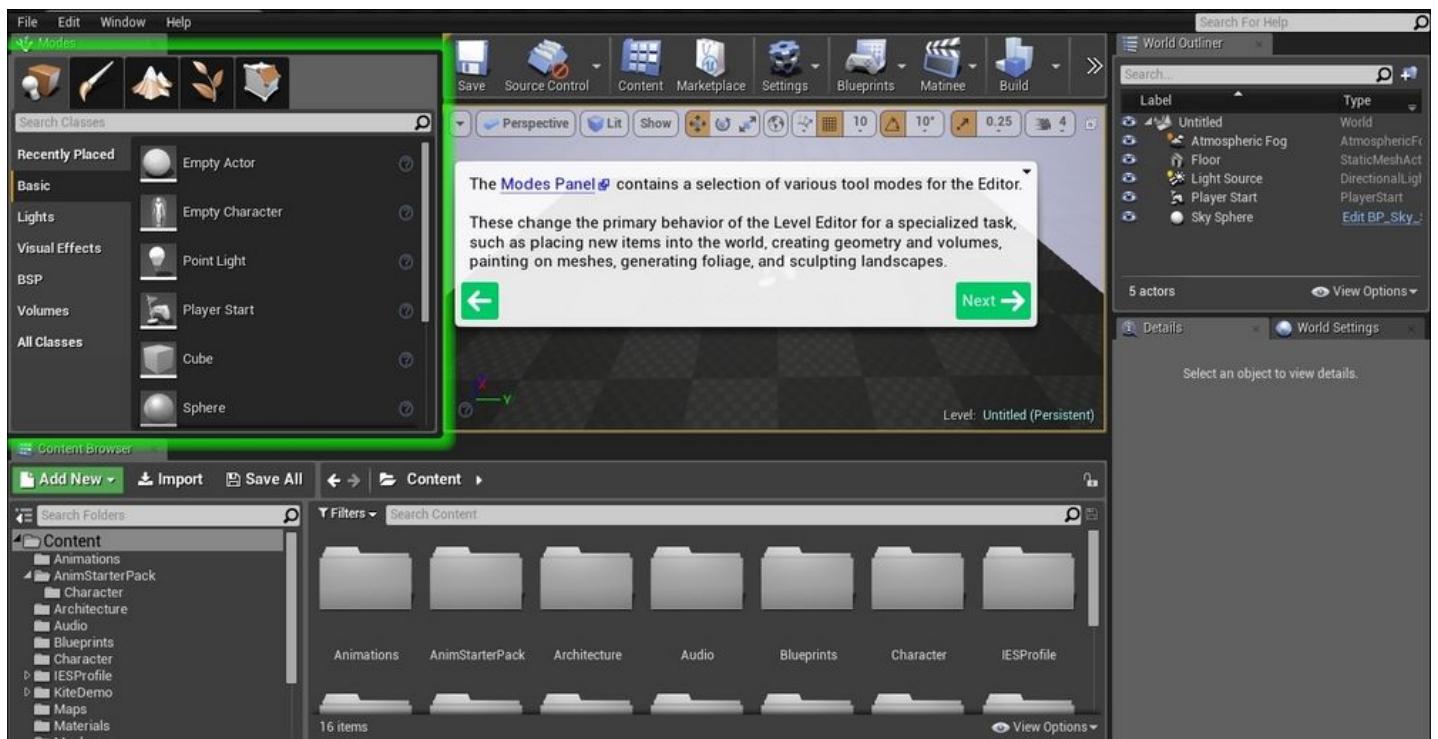
## Scene Outliner

The **Scene Outliner** contains the list of objects that are placed in the scene. It is only what is loaded currently in the scene. You can create folders and have customized names for the objects (to help you identify the objects easily). It is also a quick way to group items so that you can select them and make changes in bulk. The following screenshot shows the **Scene Outliner** highlighted in the editor:



## Modes

The **Modes** window gives you the power to create and place objects into the game world. You can select the type of activity you wish to execute. Select from Place, Paint, Landscape, Foliage and Geometry Editing. Place is to put objects into the game world. Paint allows you to paint vertices and textures of objects. Landscape and Foliage are useful tools for making large scale natural terrains in the game. Geometry Editing provides the tools to modify and edit the object. The highlighted area in the following screenshot shows the **Modes** window:



# Summary

In this chapter, we covered introductory content about what a game engine is, specifically Unreal Engine 4 and its history. We also talked a little about how games are developed and various roles that exist in a game company to help create different components of a game. Then, we covered the different components of Unreal Engine and how we can use these different features to help us make our game. Lastly, we covered the different editors that are available to us to help us customize each of the components of the game.

In the upcoming chapters, we'll be going into the details of the functionalities and features of Unreal Engine 4. In the next chapter, you will be exposed to some basic functions in the Unreal Editor and start making your own game level.

# Chapter 2. Creating Your First Level

In this chapter, you will create and run a simple level with the help of step-by-step instructions. Since the objective of this book is to equip you with the skills to confidently create your own game using Unreal Engine 4 and not to simply follow a list of steps to create a fixed example, I will provide as much additional information as possible that you could use to create your own game level as we go about learning the basic techniques.

In this chapter, we will cover the following topics:

- How to control views and viewports
- How to move, scale, and rotate objects in a level
- How to use the BSP Box brush to create the ground and a wall using the **Additive** mode
- How to carve a hole in a wall using the **Subtractive** mode of the BSP Box brush
- How to add a simple **Directional Light** to a level to mimic sunlight
- How to spawn a player who's facing the right direction on a map using **Player Start**
- How to create the sky in your map using atmospheric fog
- How to save the map you've created and set it as the default load up map for a project
- How to add a material to the geometries you've created so that it looks realistic
- How to duplicate BSP Brushes to help create things quickly
- How to add props (which are also known as **static meshes**) to a room
- How to concentrate light on important parts of a map using **Lightmass Importance Volume**

## Exploring preconfigured levels

Before we create a level, it is good to have an idea of what levels look like in Unreal Engine 4. Unreal Engine 4 offers the possibility to load up various types of game levels with a default playable level that's straight from the **Project Browser** menu option (this pops up immediately after launching the Unreal Editor). Personally, I really like this particular new feature of Unreal Engine 4 as it gives me a quick feel of the types of presets that are available, and I could easily select something as a base for the game level I want to create.

We will create a new map using one of the preset project types as the base for our first level.

### Tip

#### How to quickly explore different project types

I normally click on the **Play** button on the toolbar after a project loads with the default level. The play function allows you to be in a game and you can see what has been precreated for you in the level.

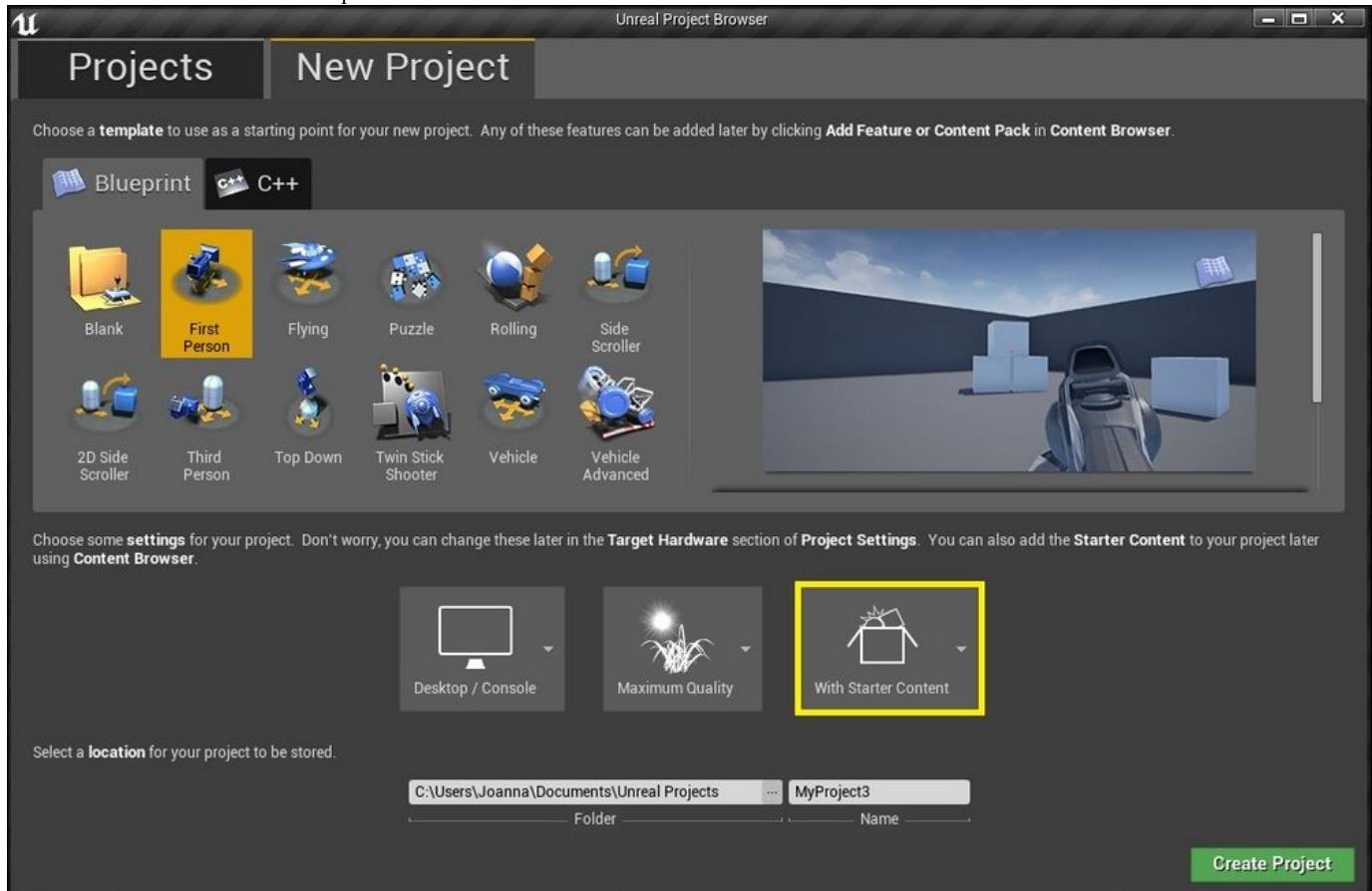
# Creating a new project

In this chapter, we will use the **Blueprint First Person** template to create our first game project.

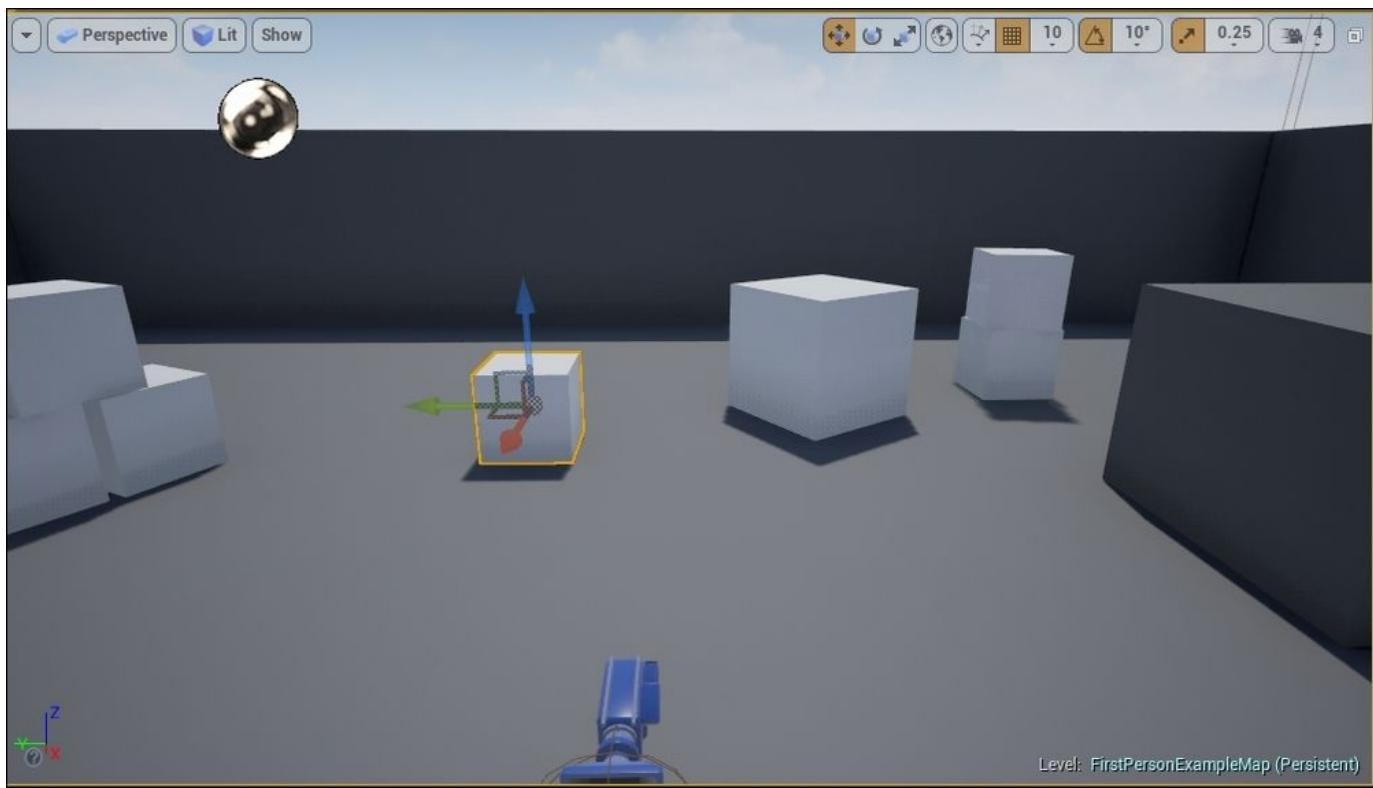
The steps to create a new **Blueprint First Person Project** are as follows:

1. Launch Unreal Engine 4.
2. Select the **New Project** tab.
3. Select **Blueprint** and then **First Person**.
4. Choose a name and path for the project (or leave it as the default `MyProject`).
5. Click on **Create Project**.

Ensure that the **With Starter Content** option is selected.



On creation of the project, the default example level for Blueprint First Person will load. The following screenshot shows how the default level looks:



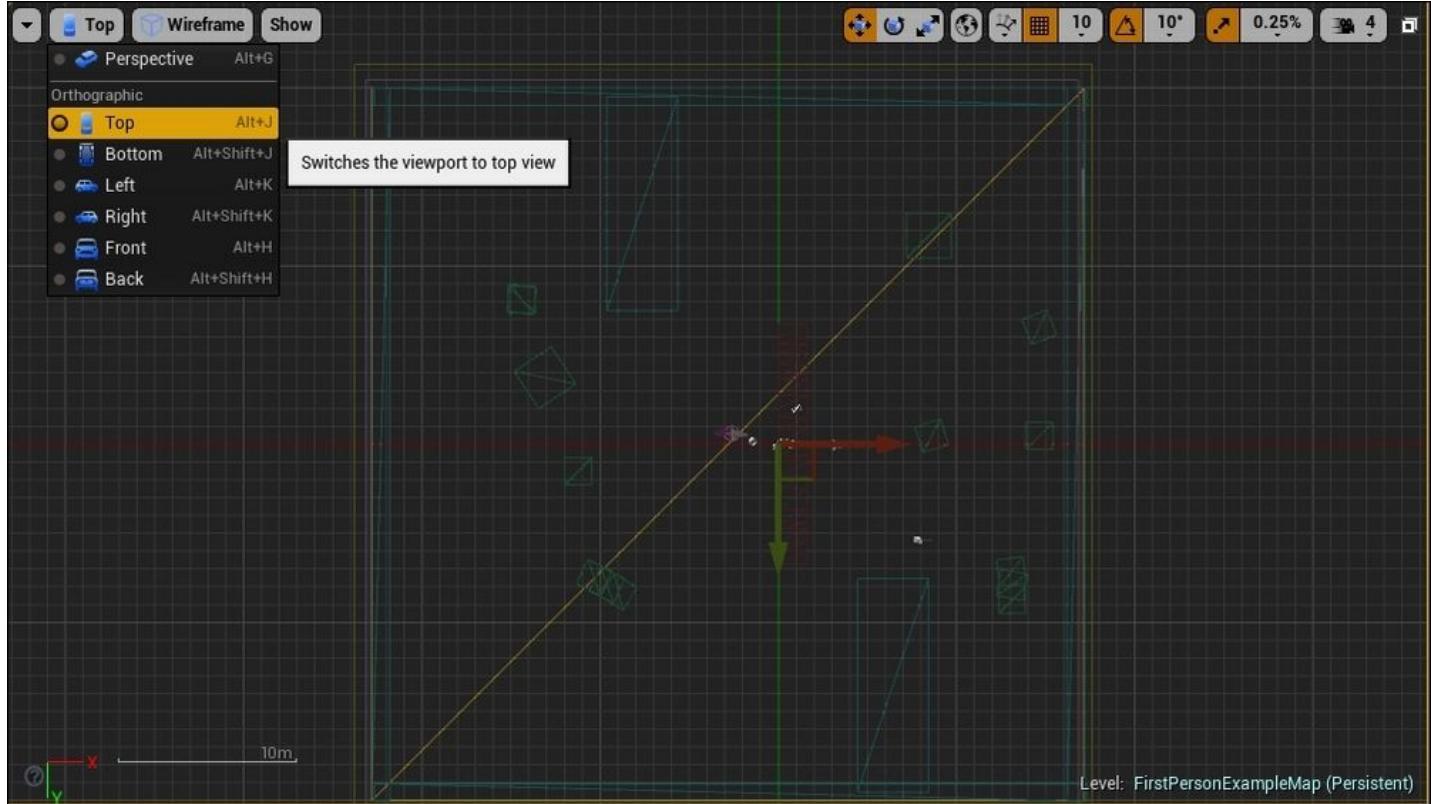
Using the preset project type with the example level, the first thing you'll probably want to do is run the level and see what the default game level contains.

# Navigating the viewport

Using the loaded example level, you should get yourself familiarized with the mouse and keyboard controls in order to navigate in the viewport. You might consider bookmarking this section until you can navigate the viewport to zoom in/out or view any object from all angles easily.

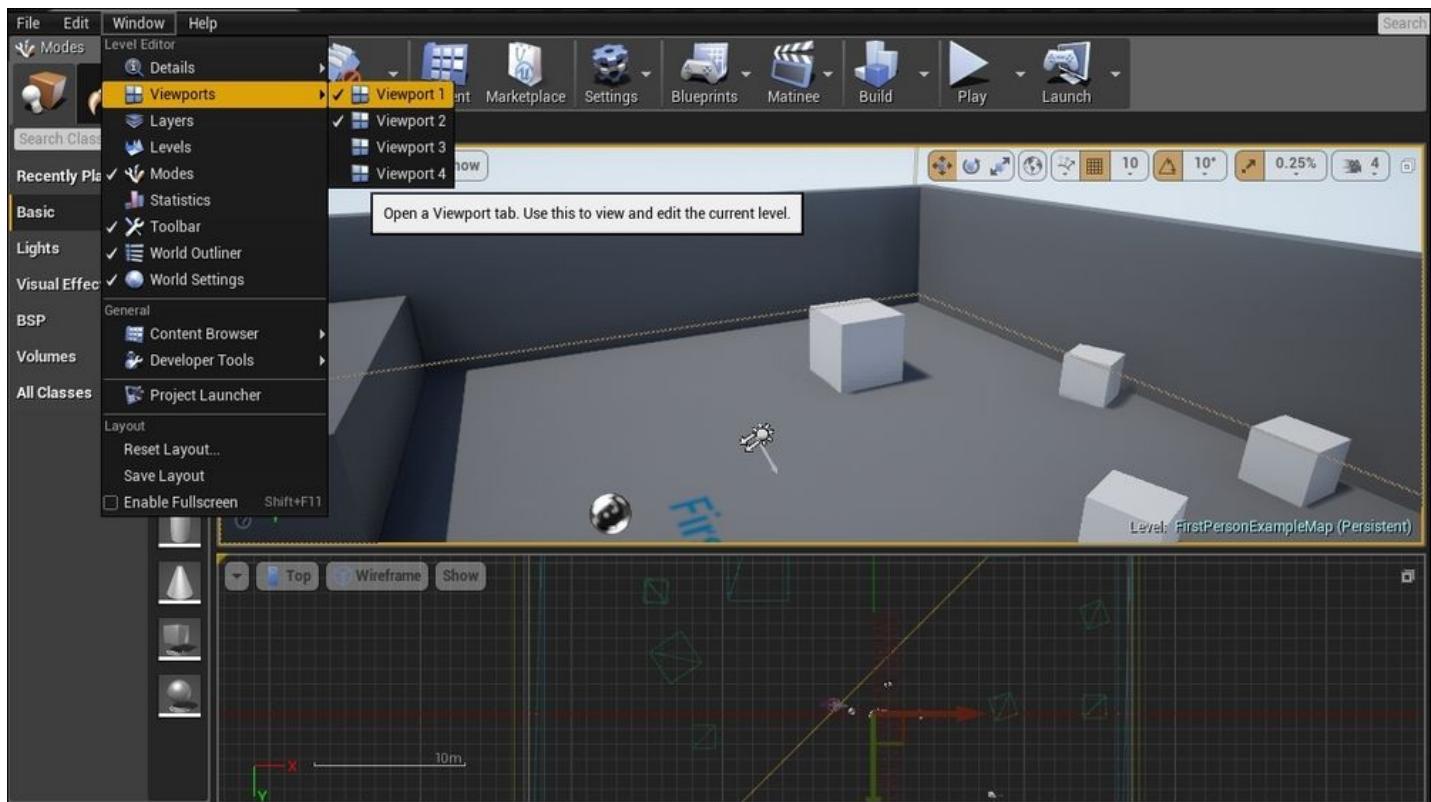
## Views

Here is some quick information on the different views in 3D modeling creation: the example map is loaded by default in the **Perspective** view. Other than having the map in the **Perspective** view, you can change what you see in the viewport in the top, side, or front views, respectively. The option to switch to any of these is in the left-hand corner of the viewport. The following screenshot shows the location of the button to press so that you can switch views:



If you wish to see more than one view concurrently, navigate to **Windows | Viewports** and then select any of the viewports (The default viewport uses **Viewport 1**).

The selected viewport number will pop up. You can drag and dock this **Viewport** window and add it to the default **Viewport 1**. The following screenshot shows **Viewport 1** and **Viewport 2** displayed at the same time (one in the **Perspective** view and the other in the **Top** view):



## Control keys

Here are some of the key presses to help you move around and view objects:

In the **Perspective** view:

Shortcut action	Description
Left-click + drag	This moves the camera forward and backward and rotates from left to right
Right-click + drag	This rotates the viewport camera
Left-click + right-click + drag	This moves objects up and down

In the **Orthographic** ( **Top** , **Front** , and **Side** ) view:

Shortcut	Description
Left-click + drag	This creates a marquee selection box
Right-click + drag	This pans the viewport camera
Left-click + right-click + drag	This zooms the viewport camera in and out

For those of you who are familiar with games, you can use WASD to navigate the camera in the editor too.

WASD control in the **Perspective** view:

Shortcut action	Description
Any mouse click + W	This moves the camera forward
Any mouse click + A	This moves the camera to the left

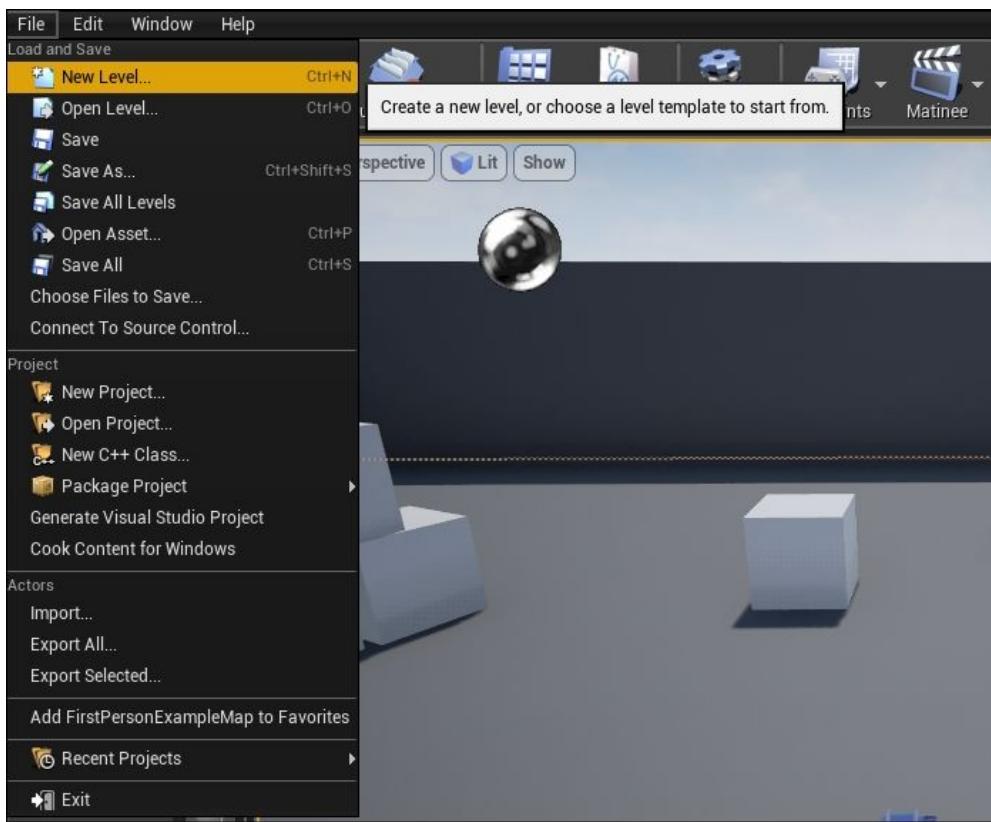
<b>Shortcut action</b>	<b>Description</b>
Any mouse click + <i>S</i>	This moves the camera backward
Any mouse click + <i>D</i>	This moves the camera to the right

On selection of an object:

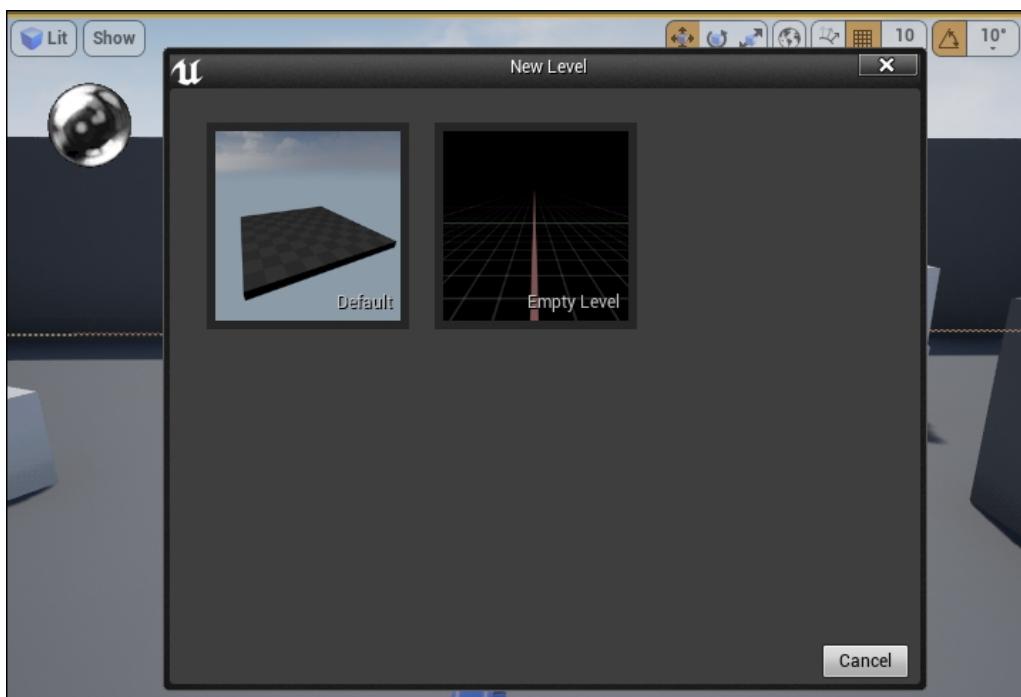
<b>Shortcut action</b>	<b>Description</b>
<i>W</i>	This displays the <b>Translation</b> tool
<i>E</i>	This displays the <b>Rotation</b> tool
<i>R</i>	This displays the <b>Scale</b> tool
<i>F</i>	This focuses the camera on a selected object
<i>Alt + Shift + Drag along the x / y / z axis</i>	This duplicates an object and moves it along the x / y / z axis

# Creating a level from a new blank map

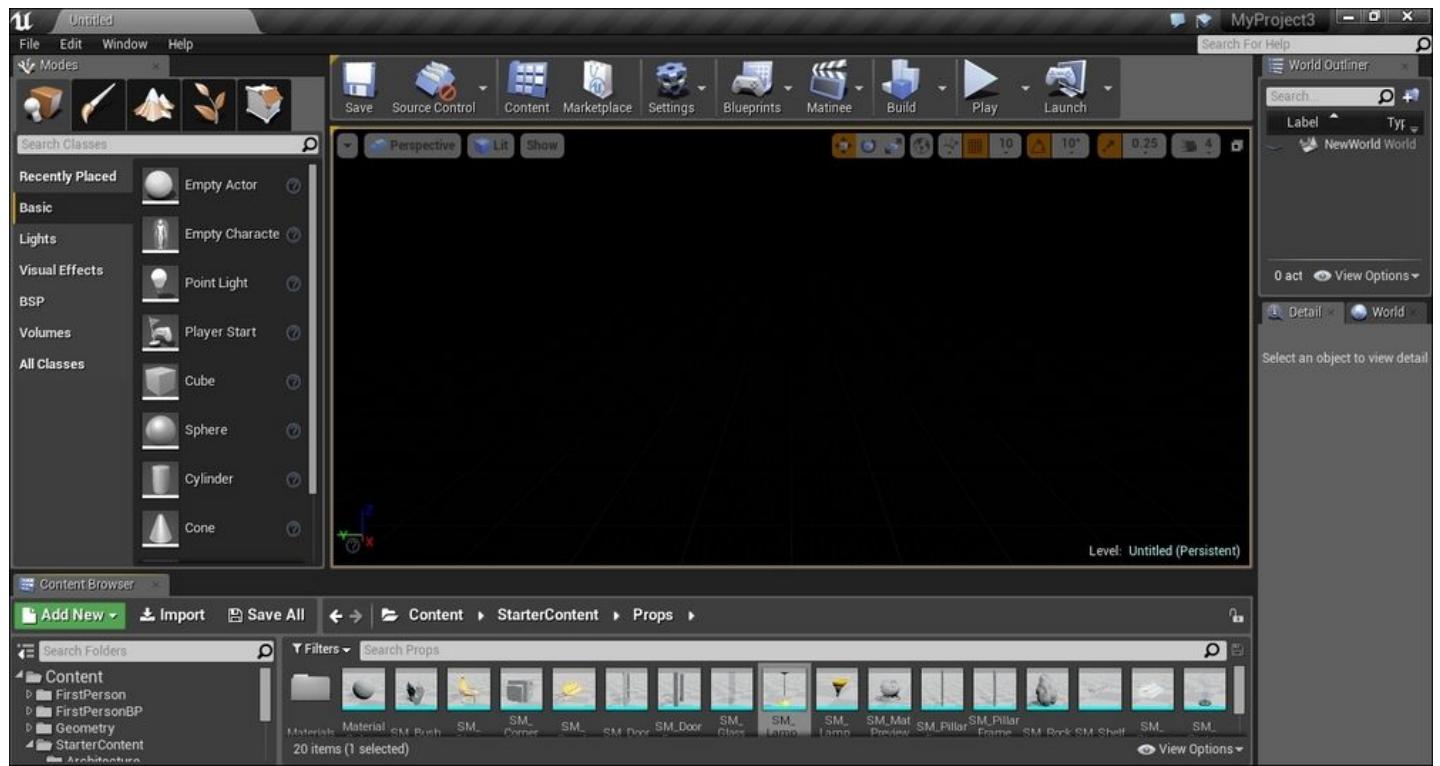
Now that you are familiar with the controls, you are ready to create a map on your own. In this chapter, we will go through how to build a basic room from scratch. To create a new map for your first person game, go to **File | New Level...**. The following screenshot shows how to create a new level:



There are two options when creating a new level: **Default** and **Empty Level**. Select **Empty Level** to create a completely blank map. The following screenshot shows you the options that are available when creating a new level:



Do not be surprised when the viewport is void. We will add objects to the level in the next few sections. The following screenshot shows what an empty level looks like in the **Perspective** view:

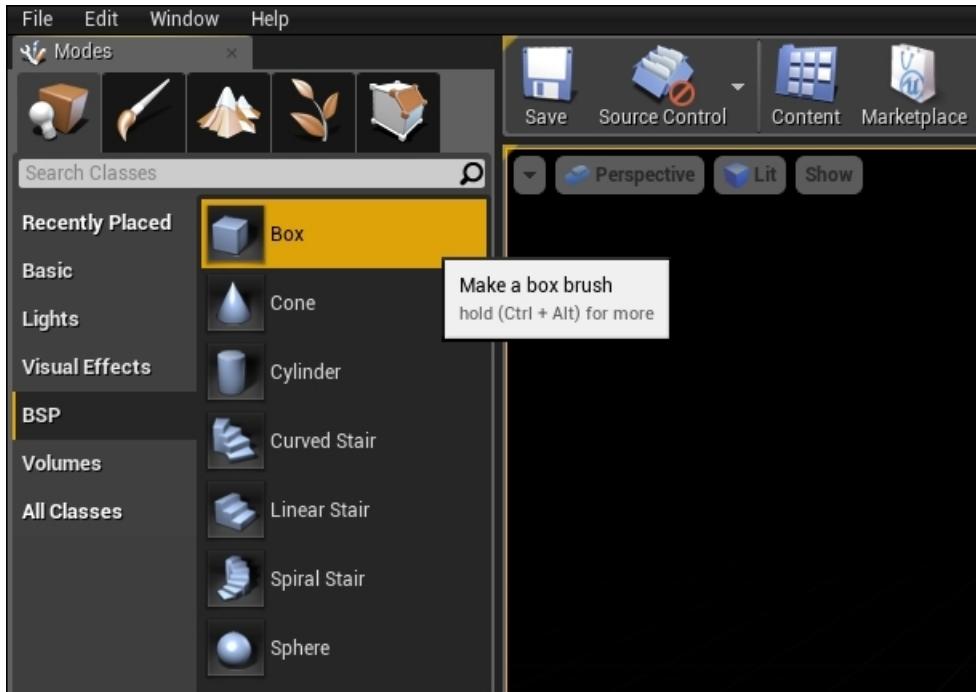


# Creating the ground using the BSP Box brush

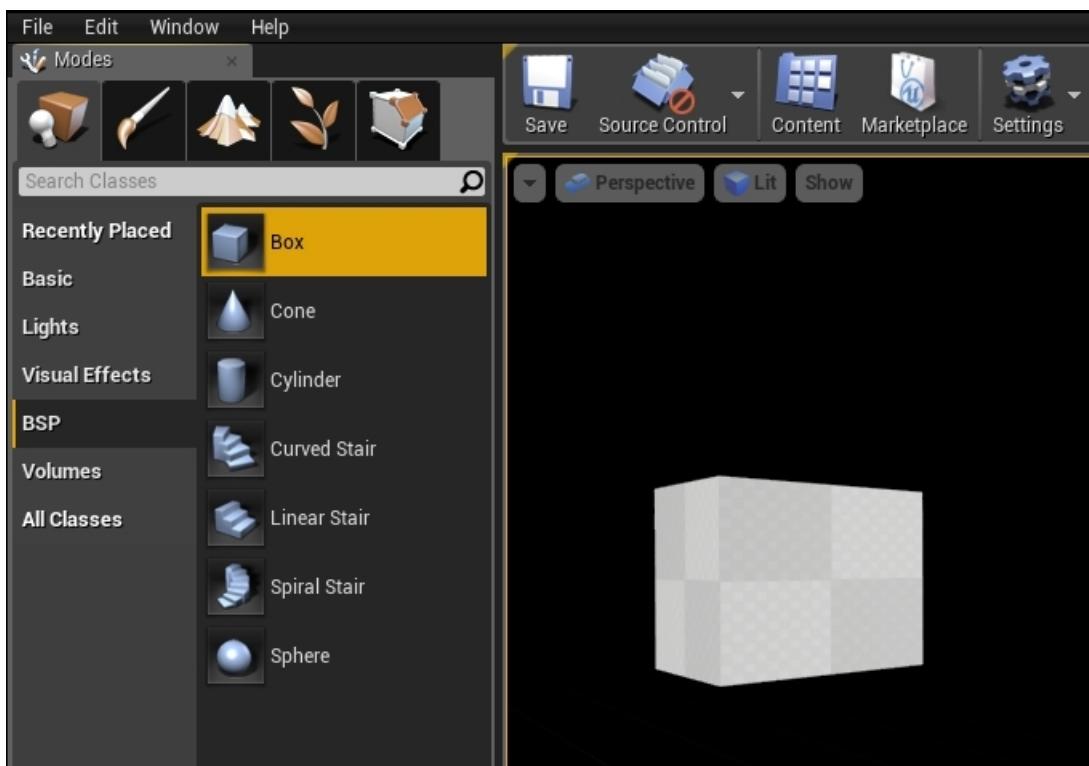
The BSP Box brush can be used to create rectangular objects in the map. The first thing to do when creating a level is to have a ground to stand on.

Before we begin with this, make sure the viewport is in the **Perspective** view. We will mainly use this view for most of the level creation unless specified explicitly.

Go to the **Modes** window, click on **BSP** and then click and drag **Box** into the viewport. This is where you can find the Box brush:

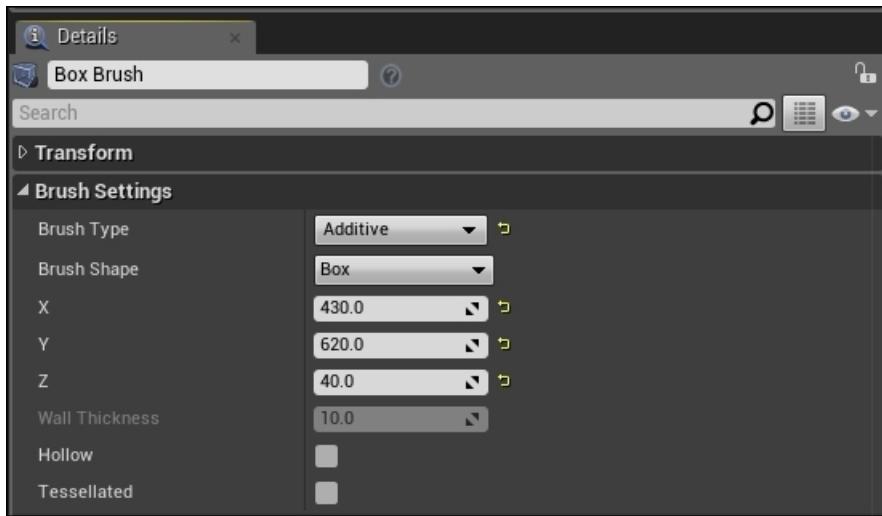


Here, a Box brush has been successfully added to the viewport:

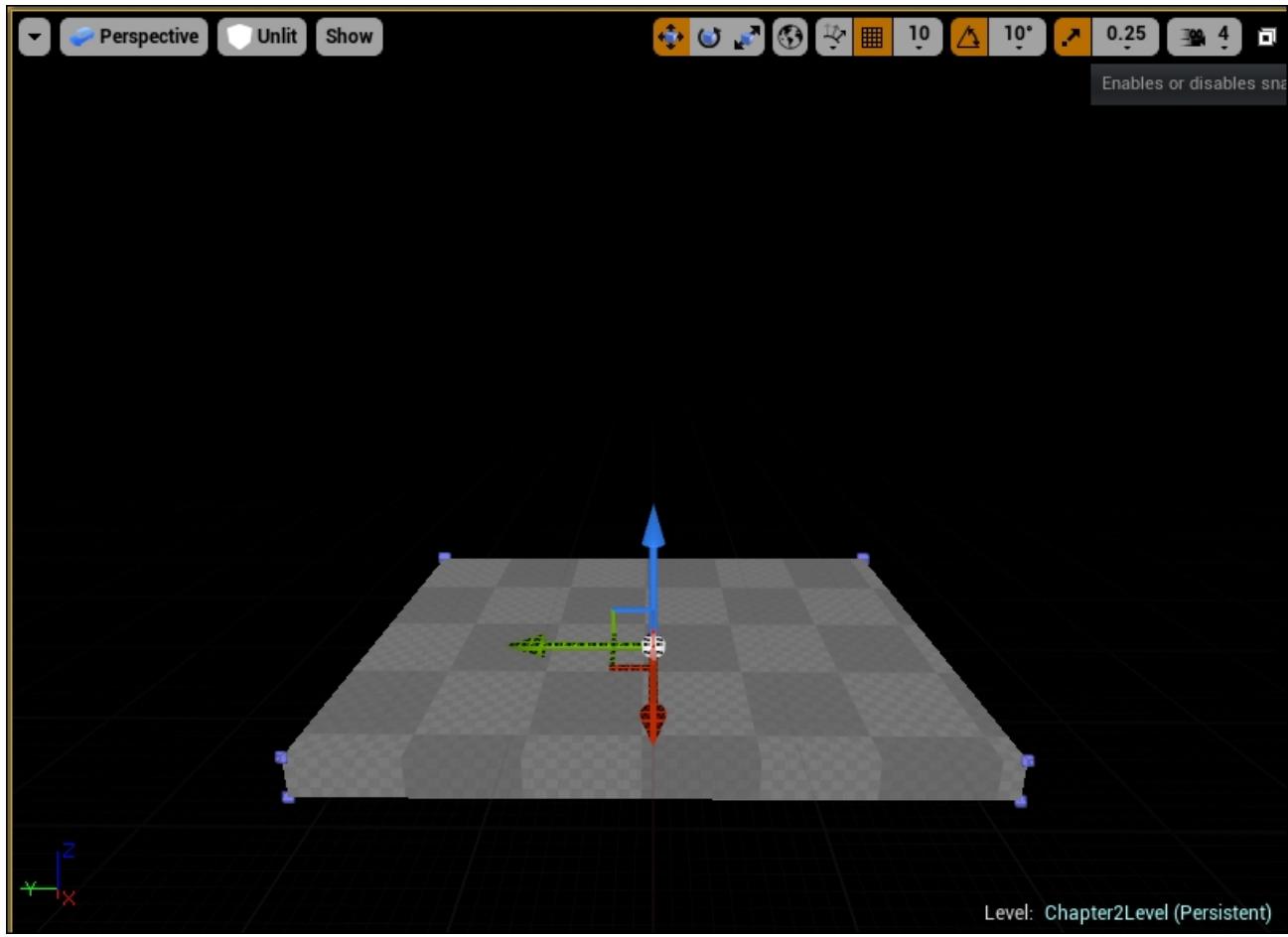


You have now successfully created your first object in the level. We will go on to change the size of this box to a suitable size so that it can act as the ground for the level.

Select the box that was just created, and go to **Details | Brush Settings**. Fill in the following values for **X**, **Y**, and **Z**. The following screenshot shows the values that need to be set:

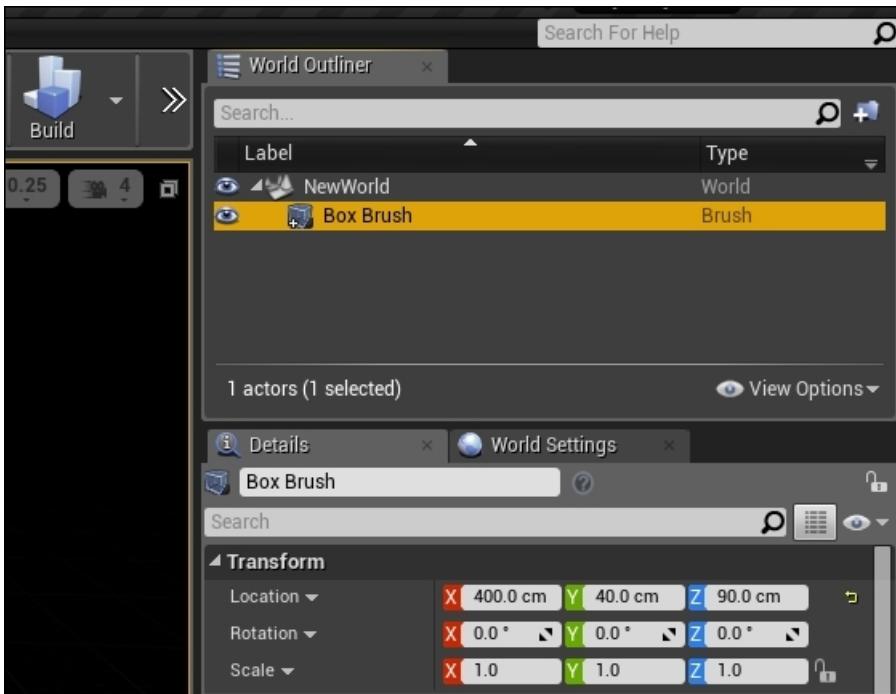


When you have set the values correctly, the box should look like this:



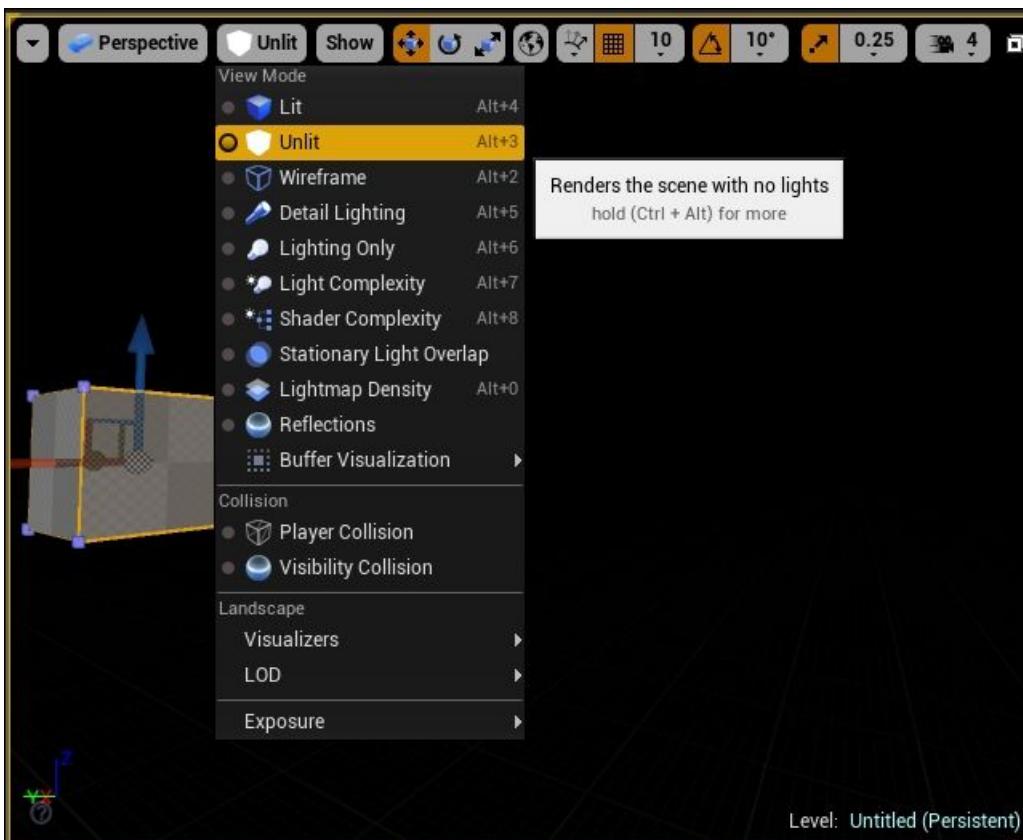
## Useful tip – selecting an object easily

To help you select objects in the level more easily, you can go to **World Outliner** (its default location is in the top right-hand corner of the editor), and you will see a full list of all the objects in the level. Click on the name of an object to select it and its details will also be displayed. This is a very useful way to help you select objects when you have many objects in the level. The following screenshot shows how **World Outliner** can be used to select the Box brush (which we've just created) in the level:



## Useful tip – changing View Mode to aid visuals

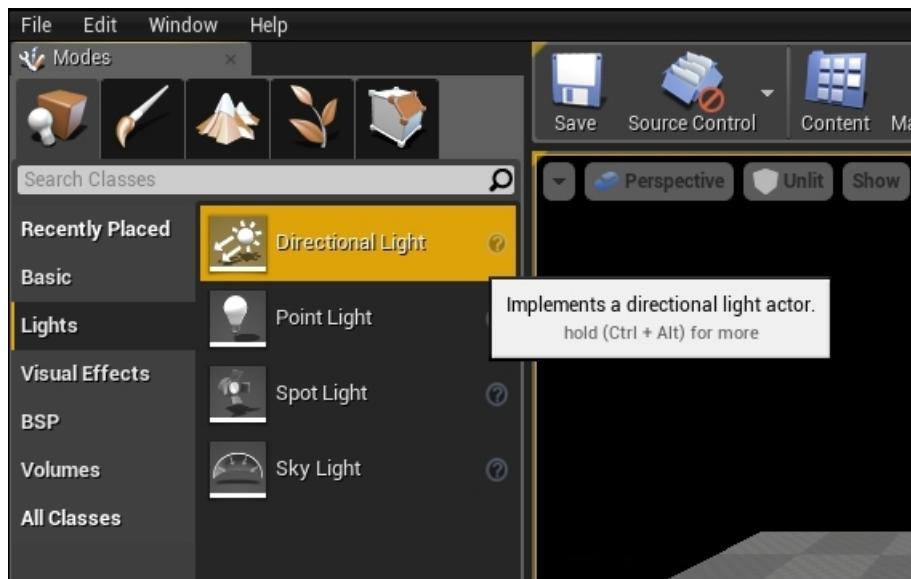
If you have difficulties seeing the box, you can change **View Mode** to **Unlit** (the button is in the viewport that's next to the **Perspective** button). The following screenshot shows you how to change **View Mode** to **Unlit**:



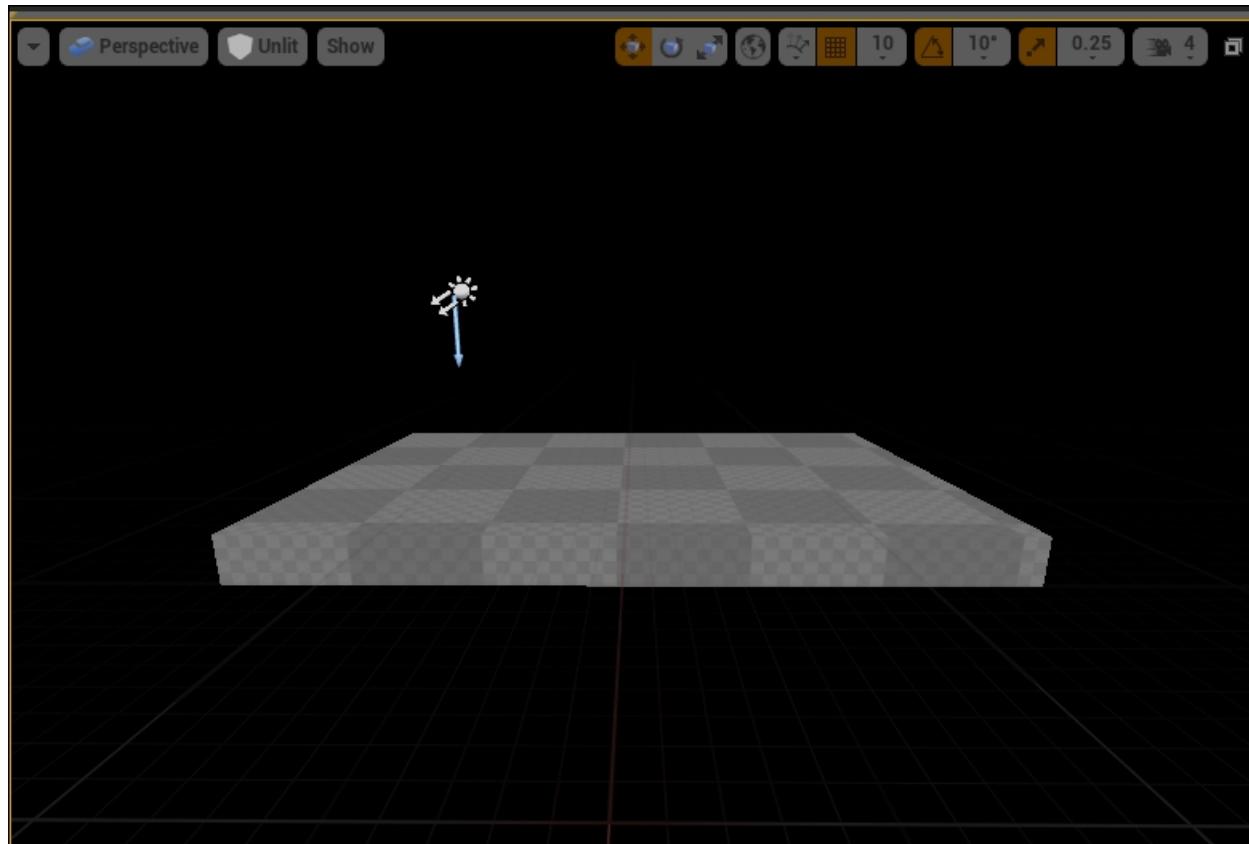
# Adding light to a level

To help us see the level better, it is time to learn how to illuminate the level. To mimic ambient light from the sun, we will use **Directional Light** for the level.

In the same way as adding a BSP Box brush, we will go to **Modes Window | Lights | Directional Light**. Click and drag **Directional Light** into the **Viewport** window. The following screenshot zooms in on the **Modes** window, showing that the **Directional Light** item can be created by dragging it into the viewport:



For now, let's place the light just slightly above the BSP Box brush as shown in the following screenshot:



## Useful tip – positioning objects in a level

To position an object in a level, we use the **Transform** tool to move objects in the x, y, and z directions. Select the object and press the **W** key to

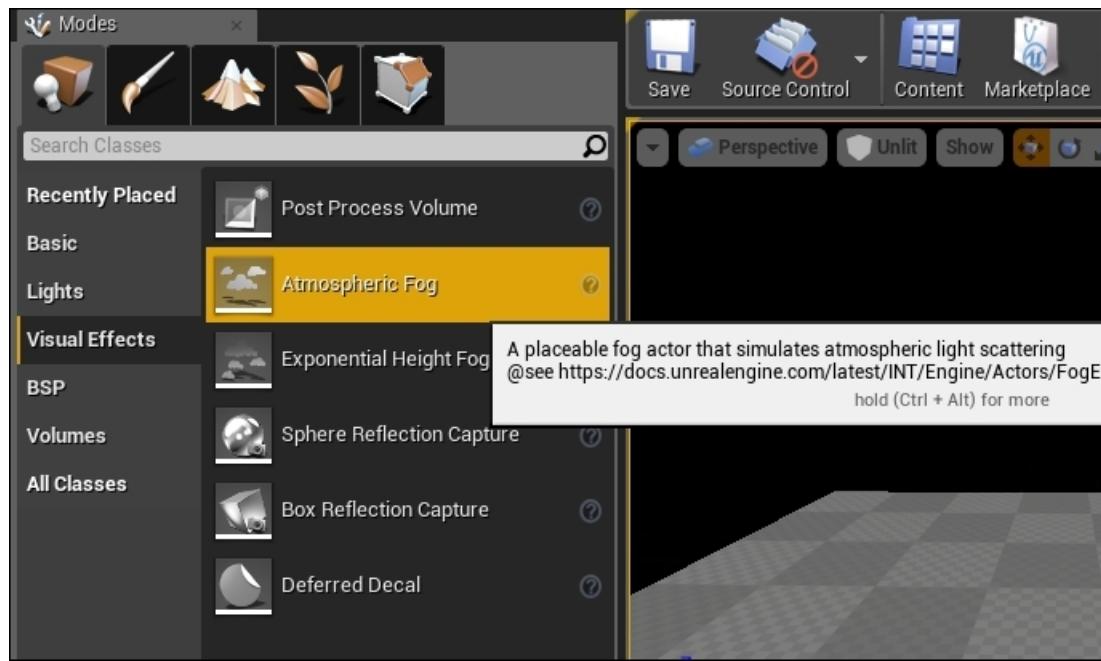
display the **Transform** tool. Three arrows will appear to extrude from the object. Click and hold the red arrow to move the object along the *x* axis, the green arrow to move it along the *y* axis, and the blue arrow to it move along the *z* axis.

To help you position the objects more accurately, you can also switch to the **Top** view when moving objects in the *x* and *y* directions, the **Side** view for adjustments in the *y* and *z* directions, and the **Front** view to adjust the *x* and *z* directions.

For those of you who want precise position control, you can use **Details**. Select the object to display details. Go to **Transform | Location**. You can select **Relative** or **World position** by clicking on the arrow next to **Location**. Change the **X**, **Y**, and **Z** values to move the object with more precision.

# Adding the sky to a level

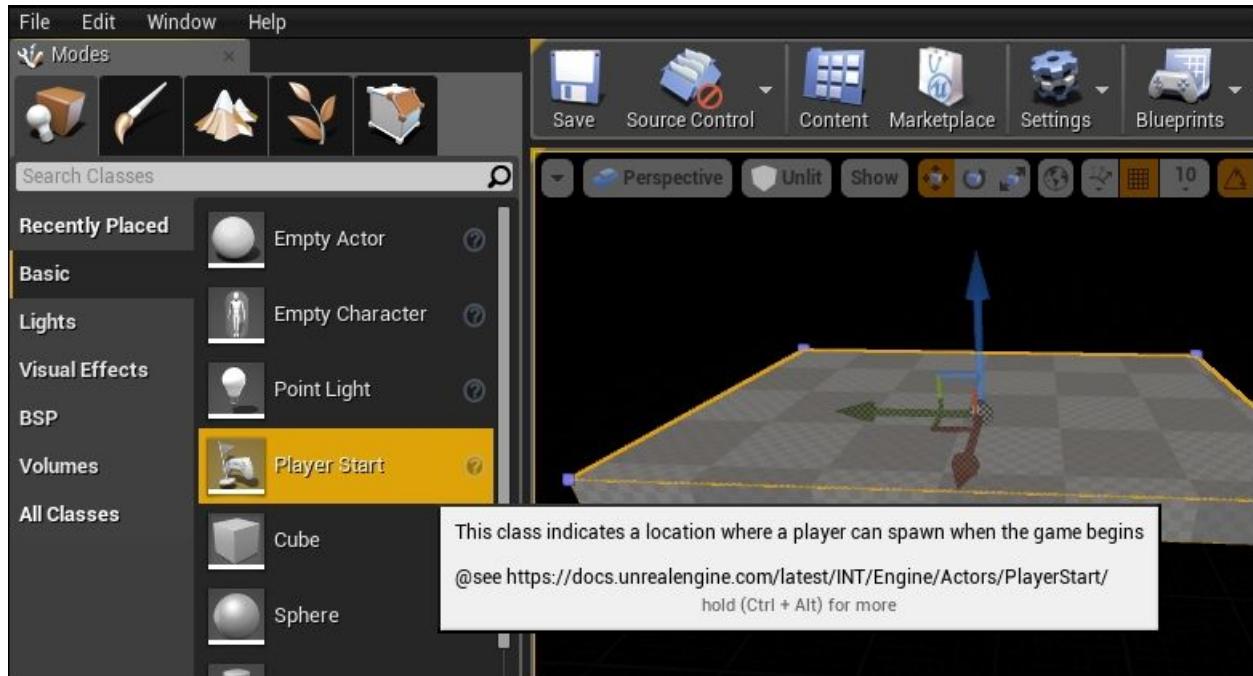
After the addition of light to the level, we will proceed to add the sky to the level. Click on **Modes** | **Visual** | **Atmospheric Fog**. In a similar way to adding light and adding a Box BSP, click, hold, and drag this into the viewport. We are almost ready to take a first look at what we have just created. Hang in there.



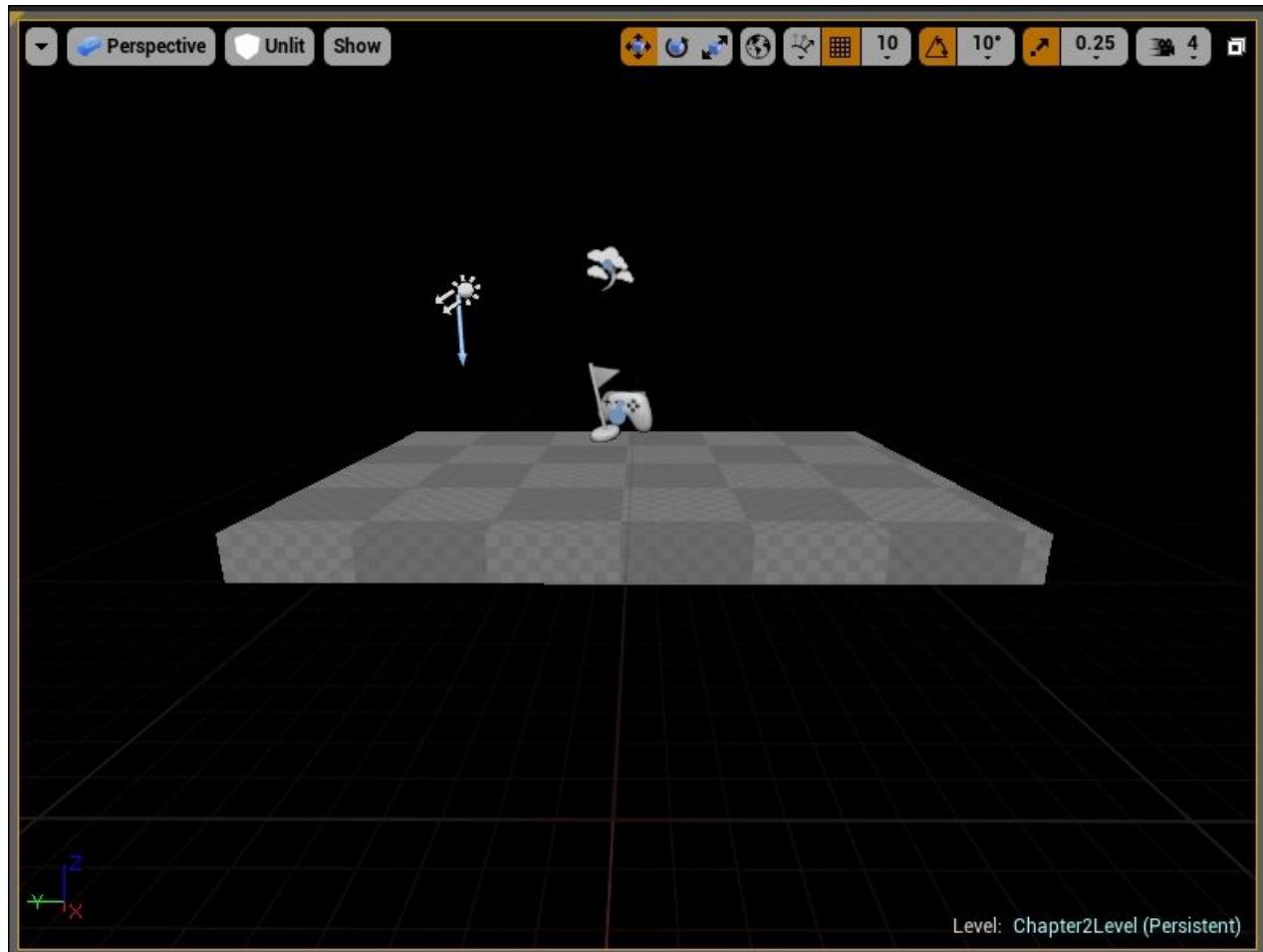
# Adding Player Start

For every game, you need to set where the player will spawn. Go to **Modes | Basic | Player Start**. Click, hold, and drag **Player Start** into the viewport.

This screenshot shows the **Modes** window with **Player Start**:



Place **Player Start** in the center of the ground or slightly above it as shown in the following screenshot:



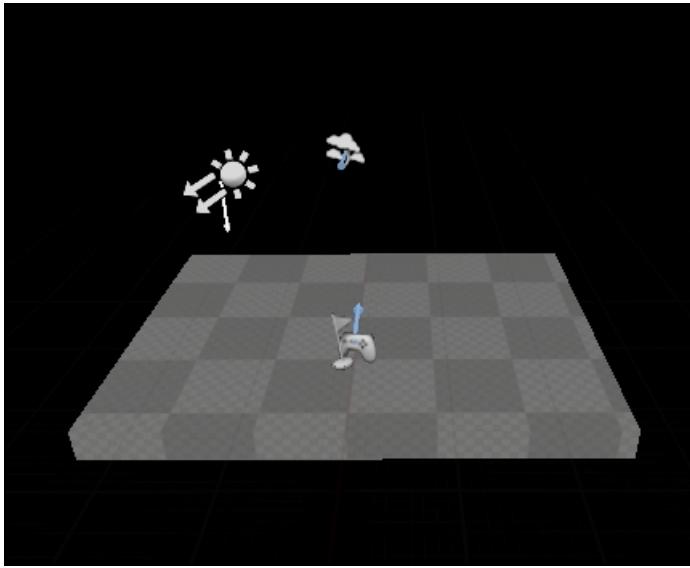
Deselect `Player Start` by pressing the `Esc` key. The light blue arrow from `Player Start` indicates the direction in which the player will spawn the game starts. To adjust the direction that the player faces upon spawning, rotate `Player Start` until the light blue arrow points in this direction. Take a look at the following tip on how to rotate an object.

## Useful tip – rotating objects in a level

To rotate an object in a level, we use the **Rotate** tool to rotate objects around the x (row), y (pitch), and z (yaw) directions. Select the object and press the `E` key to display the **Rotate** tool. Three lines with a box tip will appear to extrude from the object. Click and hold the red arrow to rotate the object around the *x* axis, the green arrow to rotate it around the *y* axis, and the blue arrow to rotate it around the *z* axis.

Another way to rotate an object more accurately is by controlling its rotation through the actual rotation values found under **Details**. (Select the object to be rotated to display its details). In the **Transform** tab, go to **Rotation**, and set the **X**, **Y**, and **Z** values to rotate the object. There is an arrow next to **Rotation** that you can click on to select if you want to adjust the rotation values for **Relative** or **World**. When you select to rotate an object using the **Relative** setting, the object will rotate relative to its current position. When the object is rotated using the **World** setting, it will be relative to the world's position.

If you want the player controller (as shown in the preceding screenshot) to have the light blue arrow facing inwards and away from you, you will need to rotate the player controller 180 degrees around the *y* axis. Enter **Y** as `180` under the **Relative** setting. The player controller will be rotated in the manner shown in this screenshot:



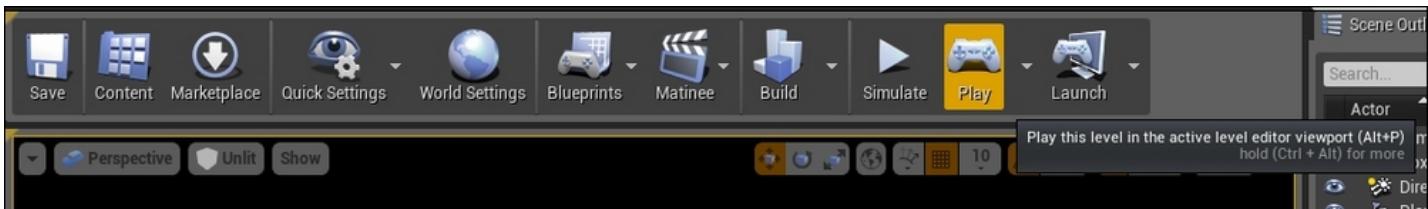
# Viewing a level that's been created

We are now ready to view the simple level that we have just created.

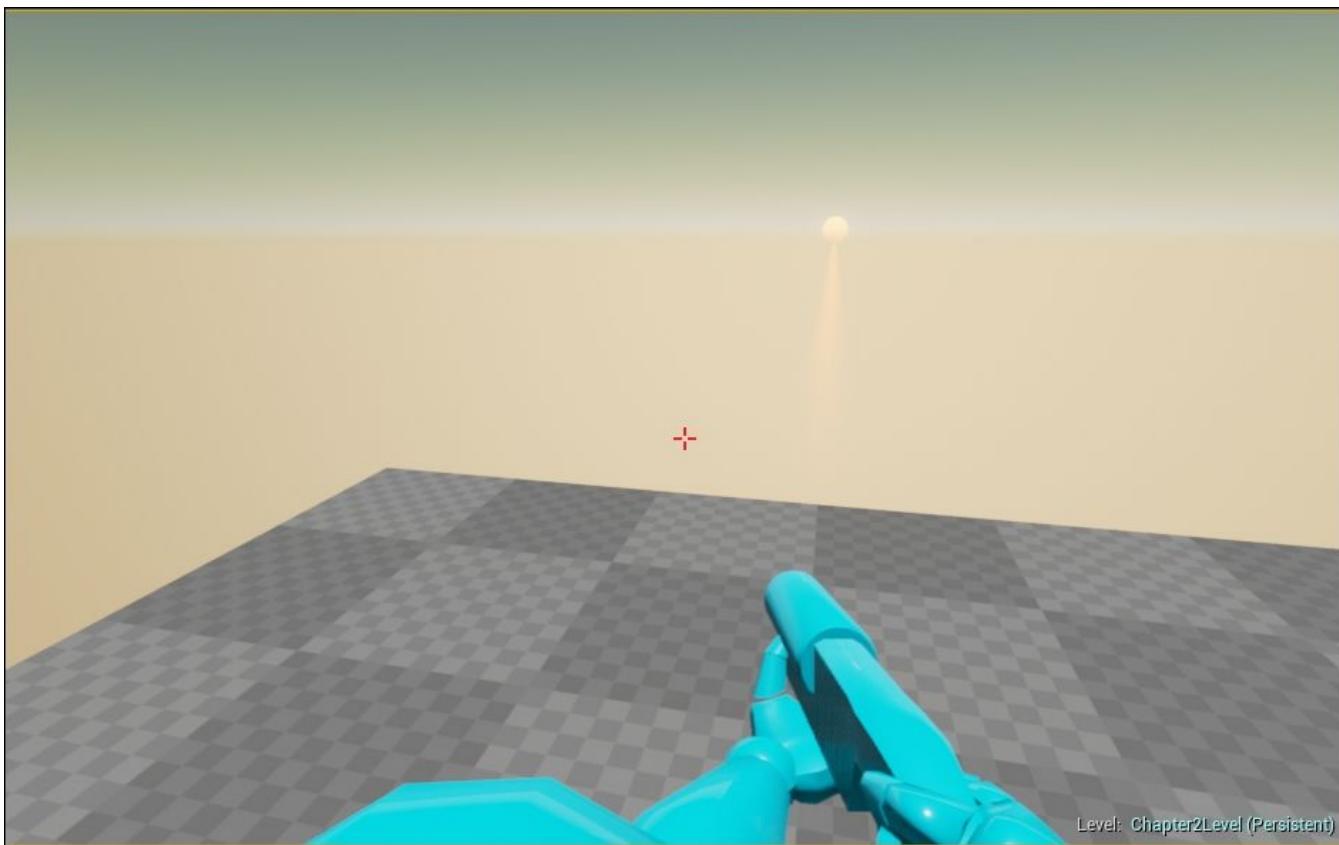
Before viewing the level, click on the **Build** button, as shown in the following screenshot, to build the light, materials, and so on, needed for this level. This step ensures that light is properly rendered in the level.



After building the level, click on the **Play** button, as shown in this screenshot, to view the level:



The following screenshot shows how the level looks. Move the mouse up, down, left, and right to see the level. Use *W*, *A*, *S*, and *D* to move the character around the level. To return to the editor, press *ESC*.



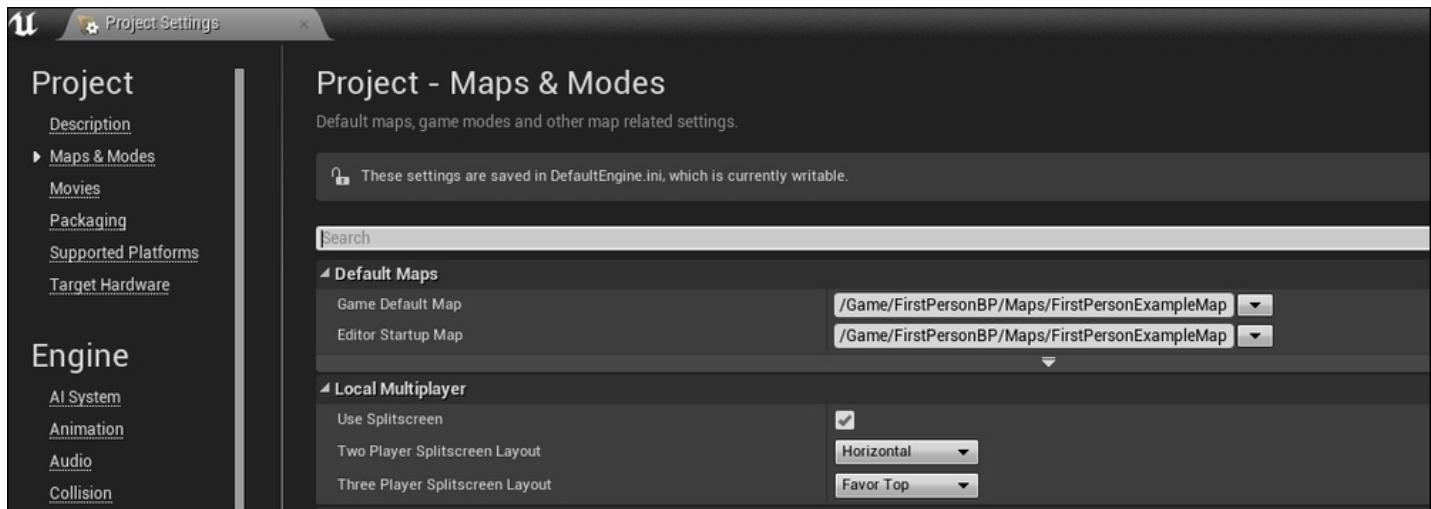
## Saving a level

Navigate to **File | Save As...** and give the map you have just created a name. In our example here, I have saved it as `Chapter2Level` in the `.../UnrealProjects/MyProject/Content/Maps` path, where `MyProject` is the name of the project.

# Configuring a map as a start level

After saving your new map, you may want to also set this project to load this map as the default map. You can have several maps linked to this project and load them at specific points in the game. For now, we want to replace the current `Example_Map` with the newly created map that we have. To do so, go to **Edit | Project Settings**. This opens up a page with configurable values for the project. Go to **Game | Maps & Modes**. Refer to the following screenshot to take a look at how **Maps & Modes** is selected.

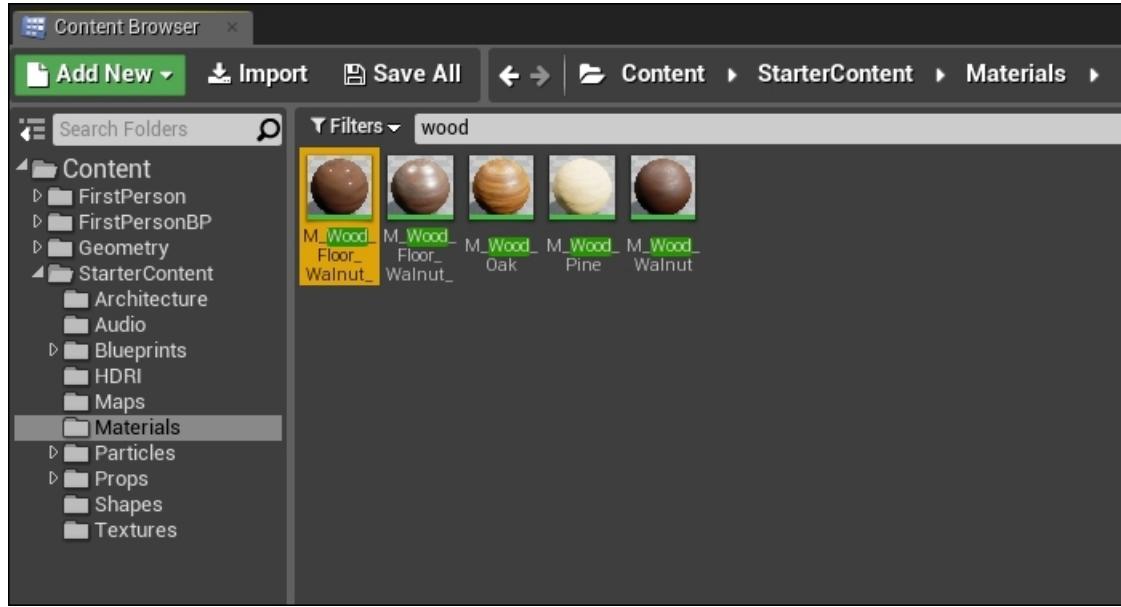
Look under **Default Maps** and change both **Game Default Map** and **Editor Default Map** in the map that you have just saved. In my case, it will be `Chapter2Level`. Then, close the project settings. When you start the editor and run the game the next time, your new map will be loaded by default.



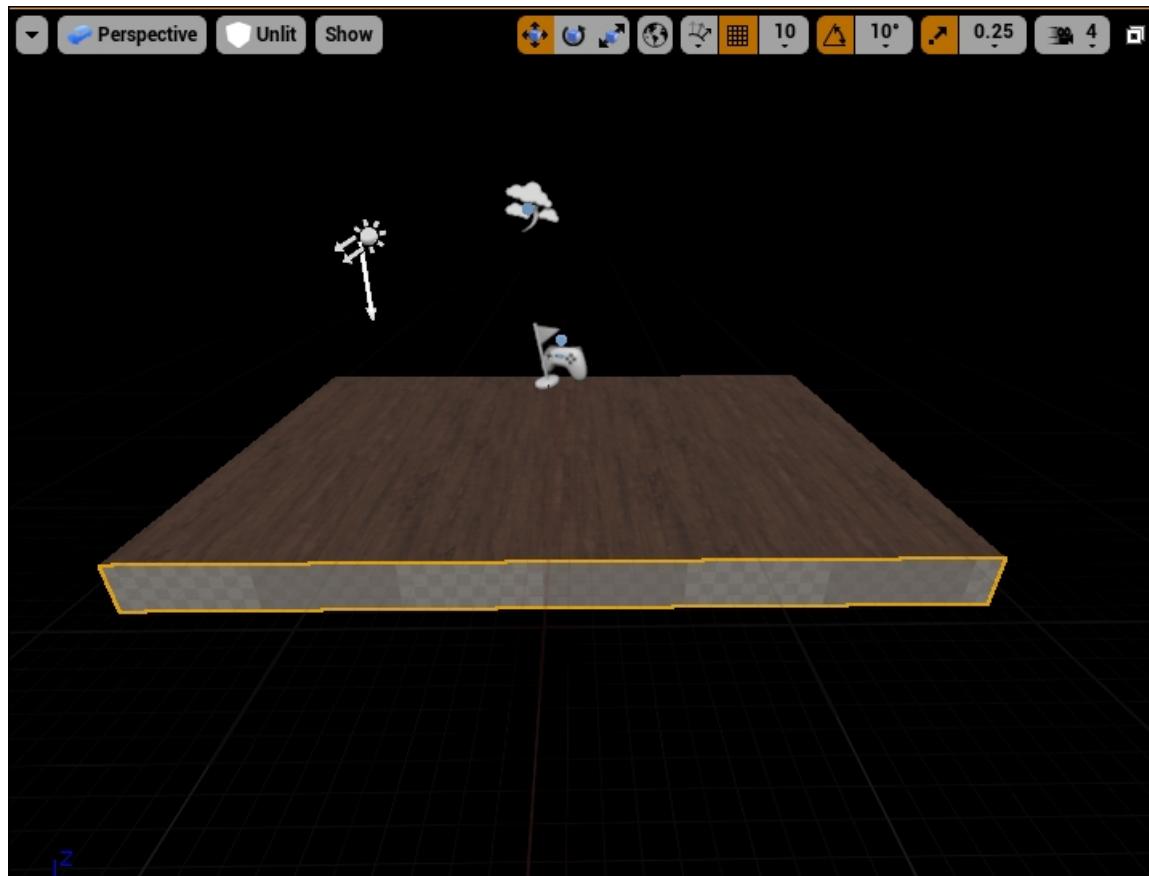
# Adding material to the ground

Now that we have created the ground, let us make the ground look more realistic by applying a material to it.

Go to **Content Browser** | **Content** | **StarterContent** | **Materials**. Type **wood** into the **Filters** box. The following screenshot shows the walnut polished material that we want to use for the ground's material:



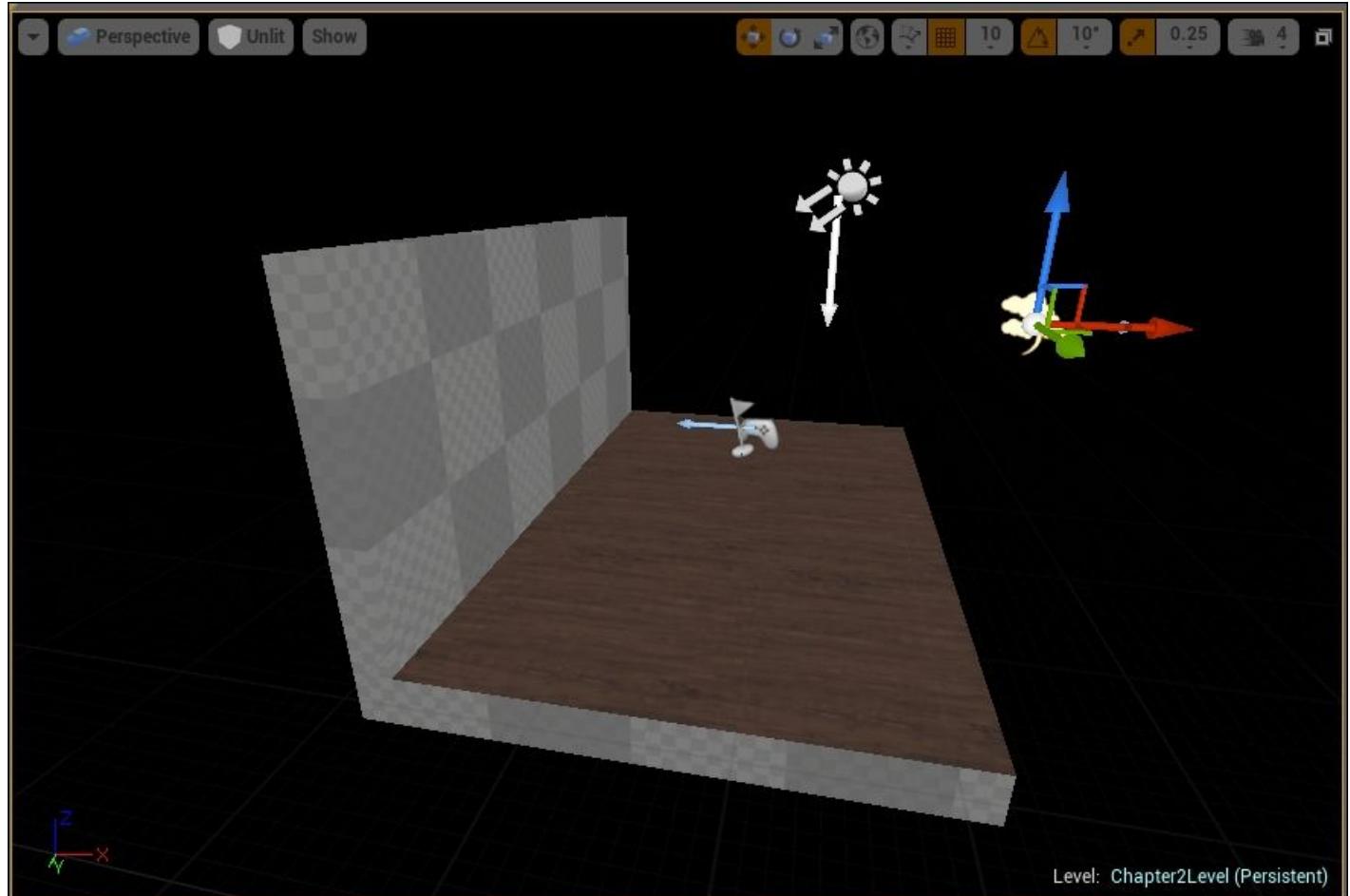
Click, hold, and drag **M\_Wood\_Floor\_Walnut\_Polished** into the viewport area and drop it on the top surface of the ground. The resulting effect should look like this:



## Adding a wall

Now we are ready to add walls to prevent the player from falling off the map. To create walls, we will use the same BSP Box brush to create a wall. As we have just added a material in the previous step, you will need to clear this material selection by clicking on anything in **Content Browser**. This will prevent new geometries from being created using the same material.

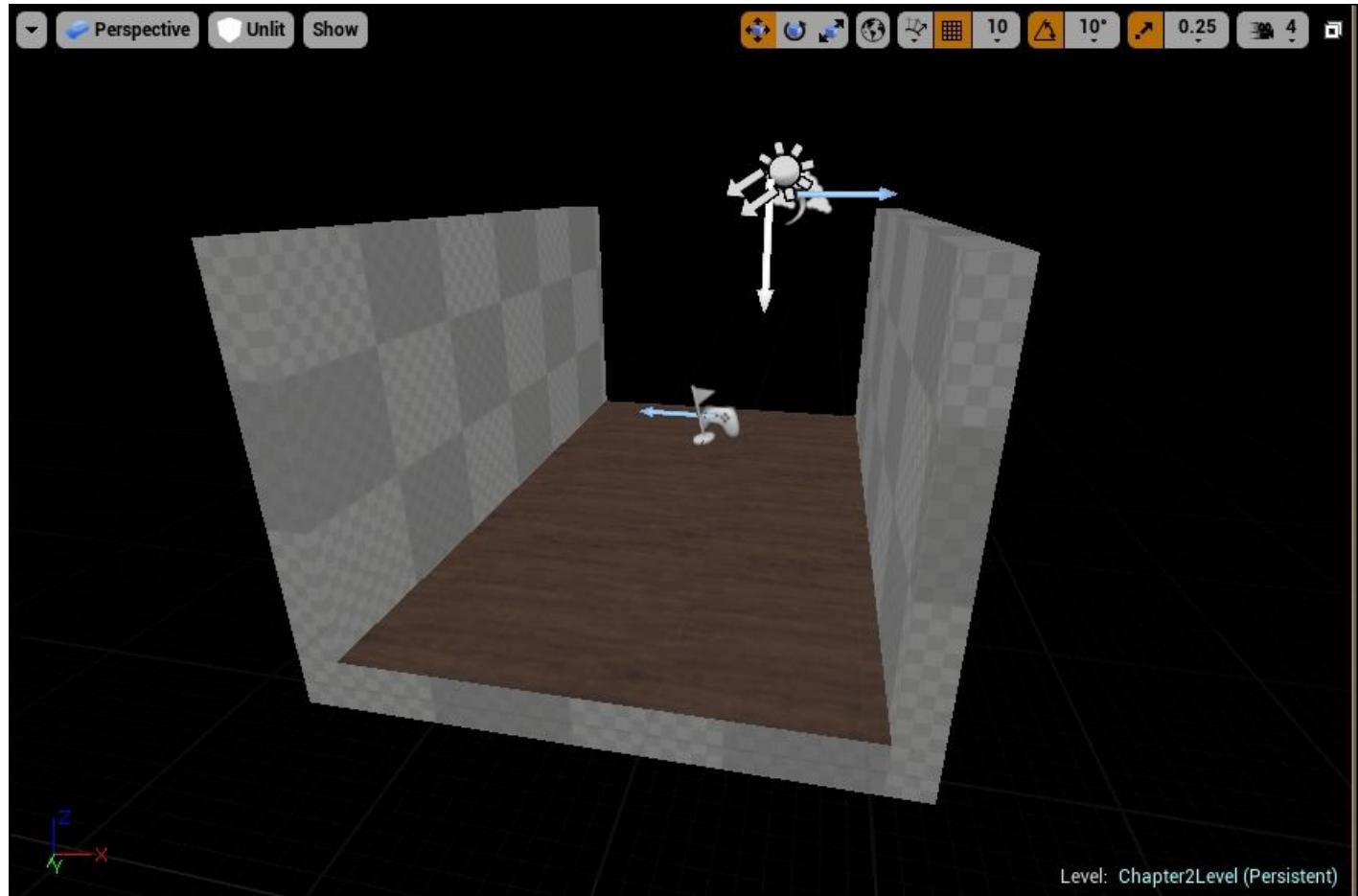
Similar to creating the ground, go to **Modes | BSP | Box**. Click, hold, and drag into the viewport. Set the dimensions of the BSP box as X = 30, Y = 620, and Z = 280. To help us view and position the wall, use the controls to rotate the viewport. You can also use the different views to help position the wall onto the ground. Here, you can see how the wall should be positioned (note that I have panned the camera to view the level from a different angle):



# Duplicating a wall

Now duplicate the wall by first selecting the wall created in the earlier step. Make sure the **Transform** tool is displayed (if not, press *W* once when object is selected).

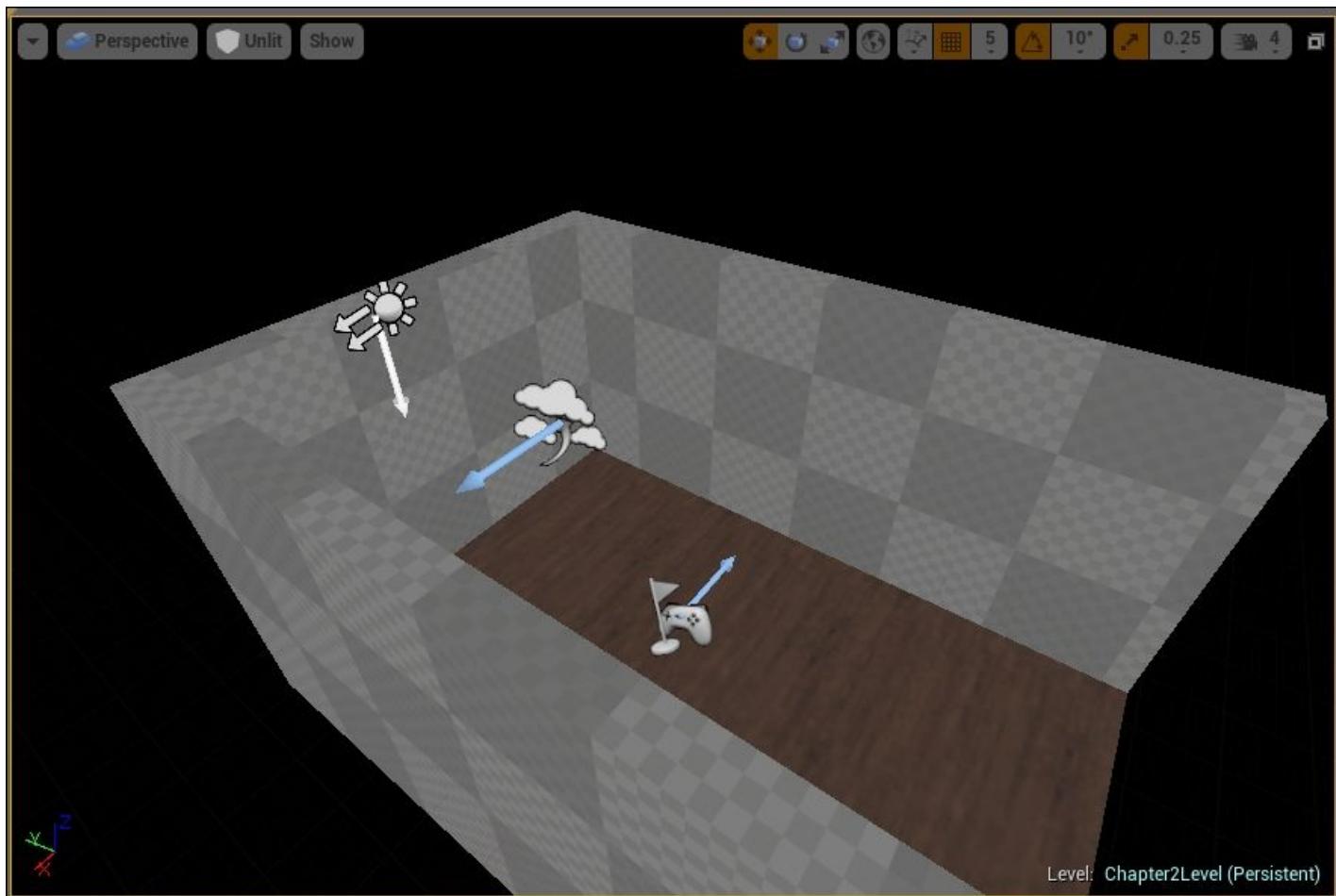
Click and hold one of the axes (the *x* axis, in the preceding example case) while holding down *Alt + Shift* as you drag the current wall in the *x* direction. You would notice that there is another copy of the wall moving in this direction. Release the keys when the wall is in the right position. Use normal translation controls to position the wall as shown here:



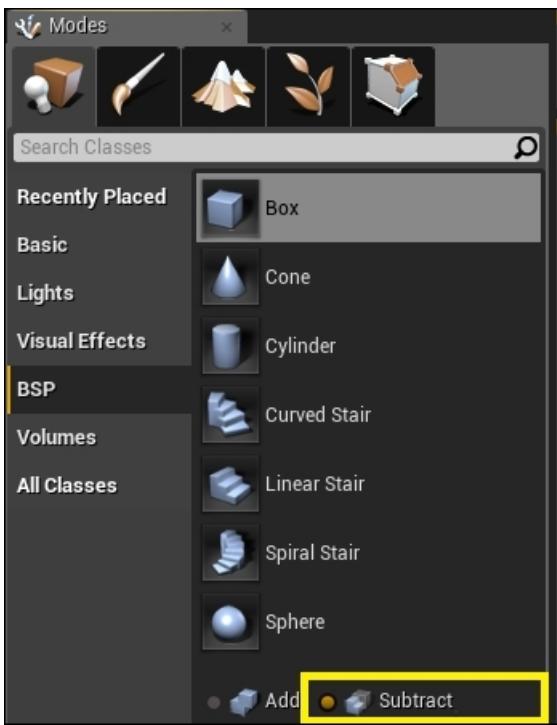
# Creating an opening for a door

The room is now almost complete. We will learn how to carve into a BSP Box brush to create an opening for a door.

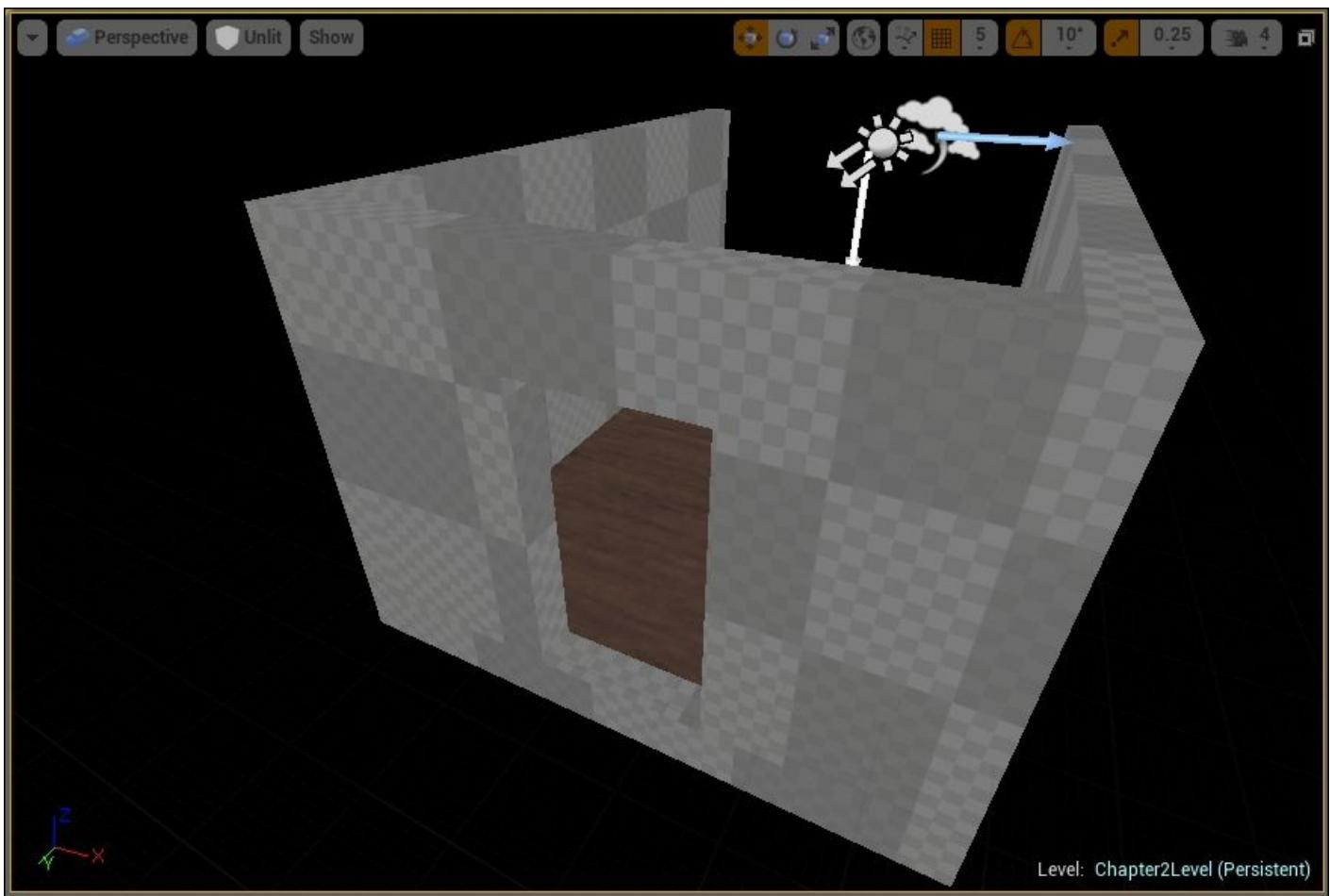
Drag a new BSP Box brush into the map: X = 370, Y = 30, and Z = 280. Position this wall to seal one side of the room as shown in the following screenshot:



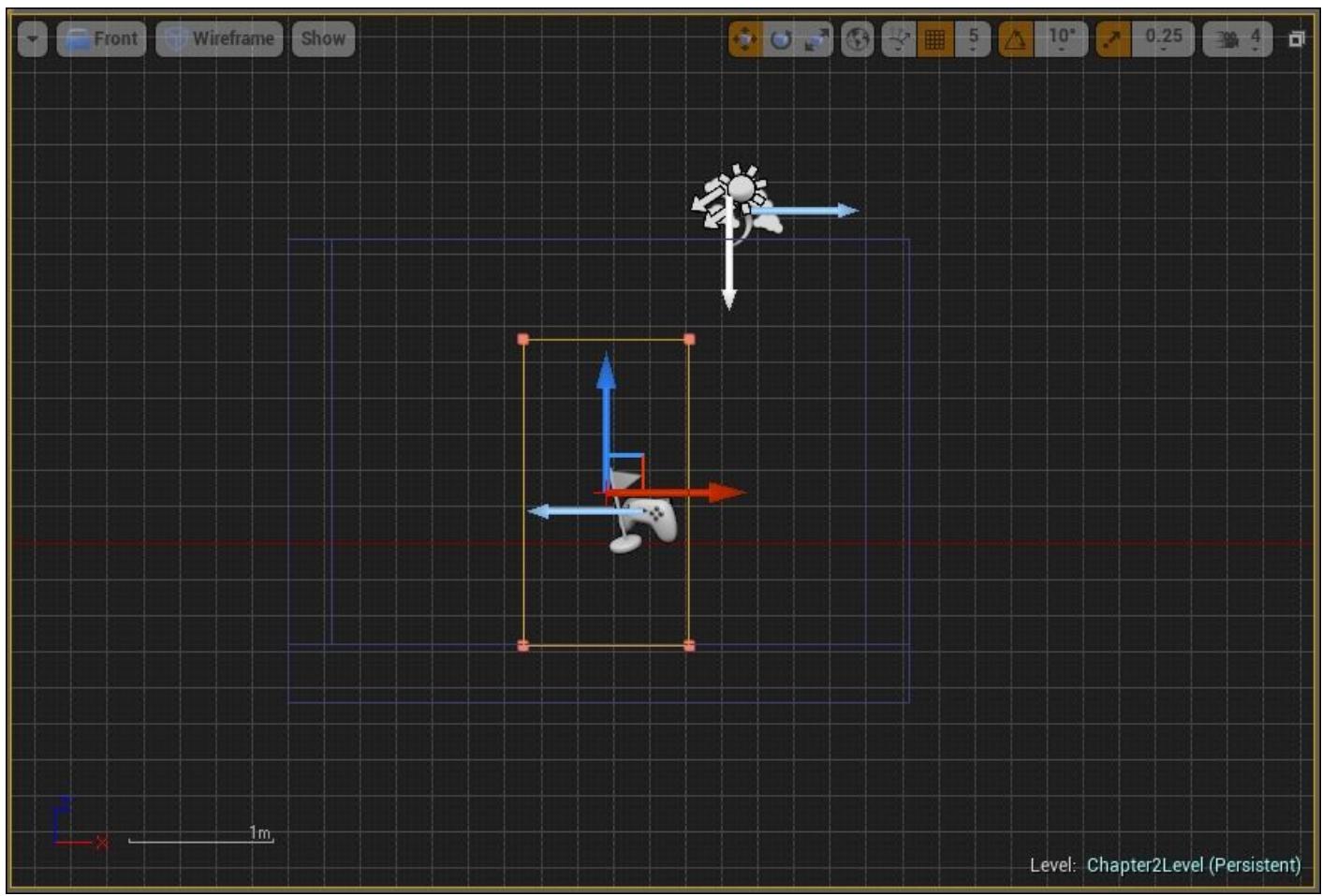
Till now, we have been using the **Additive** mode (add the radio button that is selected) to create a BSP Box brush. To create an opening in the wall, we will create another BSP Box brush using the **Subtractive** mode. Ensure that you have selected it as shown in the following screenshot. Drag and drop the BSP Box brush in the same manner as before into the viewport. As for the dimensions of this brush, we will approximate it to the size of the door, where X = 115, Y = 30, and Z = 212.



When the **Subtractive** BSP Box brush is positioned correctly, it will look something like this:



To help you position the **Subtractive** BSP Box brush, you can switch to the **Front** view to place the door more or less in the center. The following screenshot shows the **Front** view with the **Subtractive** BSP Box brush selected:



## Adding materials to the walls

To make the ground look more realistic, we will apply a material to it. Go to **Content Browser** | **Content** | **StarterContent** | **Materials**. Type `Wall` into the **Filters** box. Select **M\_Basic\_Wall** and drag it onto the surface of the wall with the door. Then, we will use a different material. Type `Brick` into the **Filters** box. Select **M\_Brick\_Clay\_New** to apply to the inner surface of the two other walls.

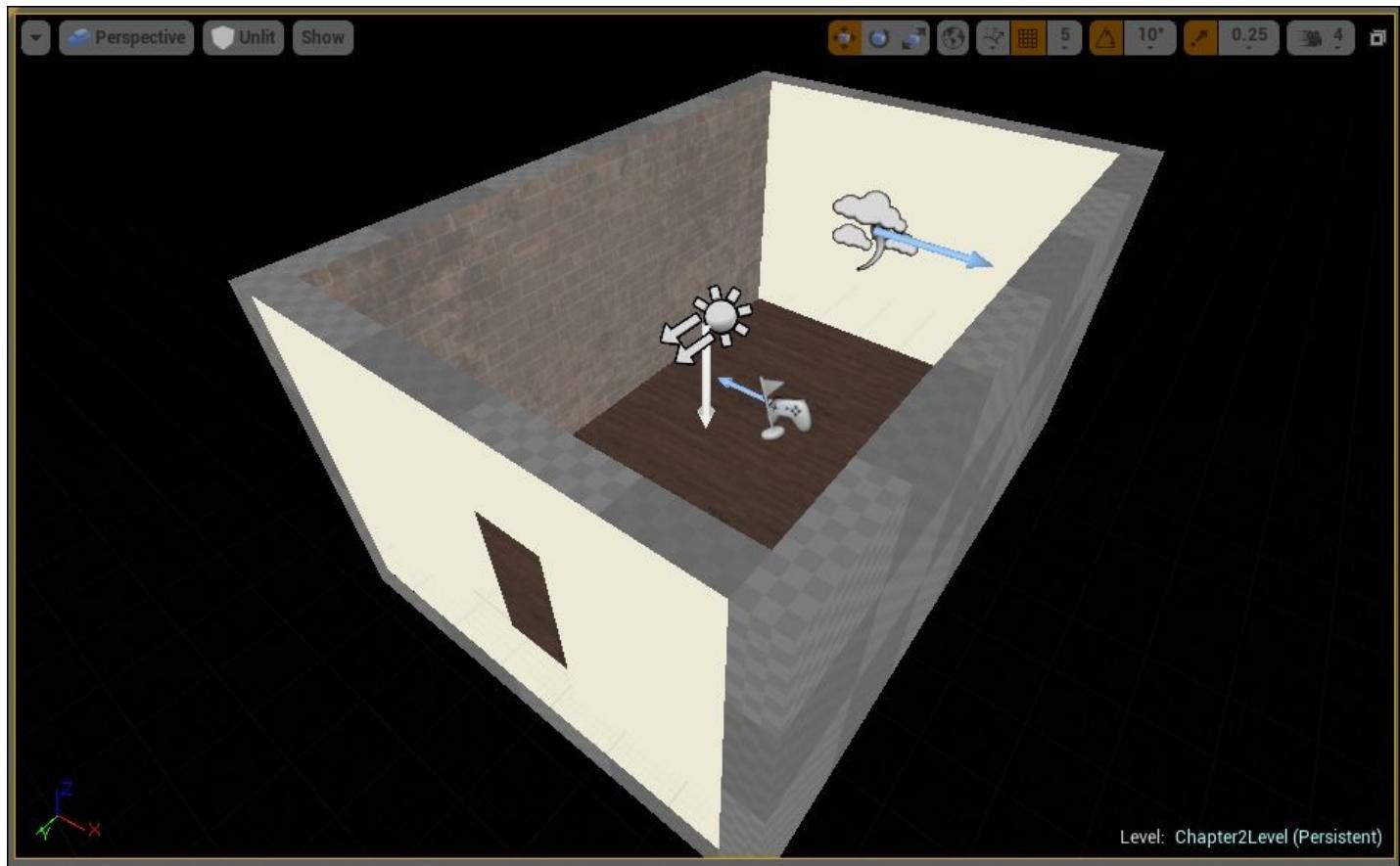
Here, you can take a look at how the level looks in the **Unlit** mode after applying the materials mentioned previously:



Build the light before running the level again to see how the level looks now.

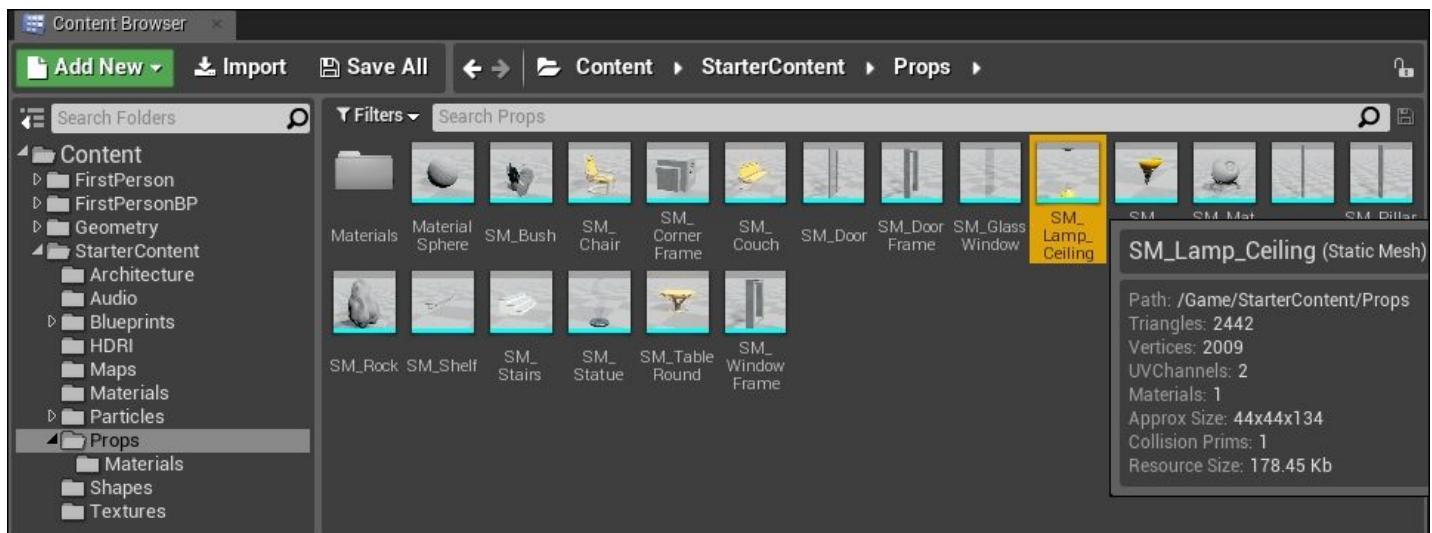
## Sealing a room

For now, let's duplicate the wall with the door to seal the room. Click on the wall, hold down *Alt + Shift*, and drag it across to the other side of the room. The following screenshot shows how it looks when the room is sealed:



# Adding props or a static mesh to the room

Let's now add some objects to the empty room. Go to **Content Browser** | **Content** | **StarterContent** | **Props**. Find **SM\_Lamp\_Ceiling** and drag it into the room.

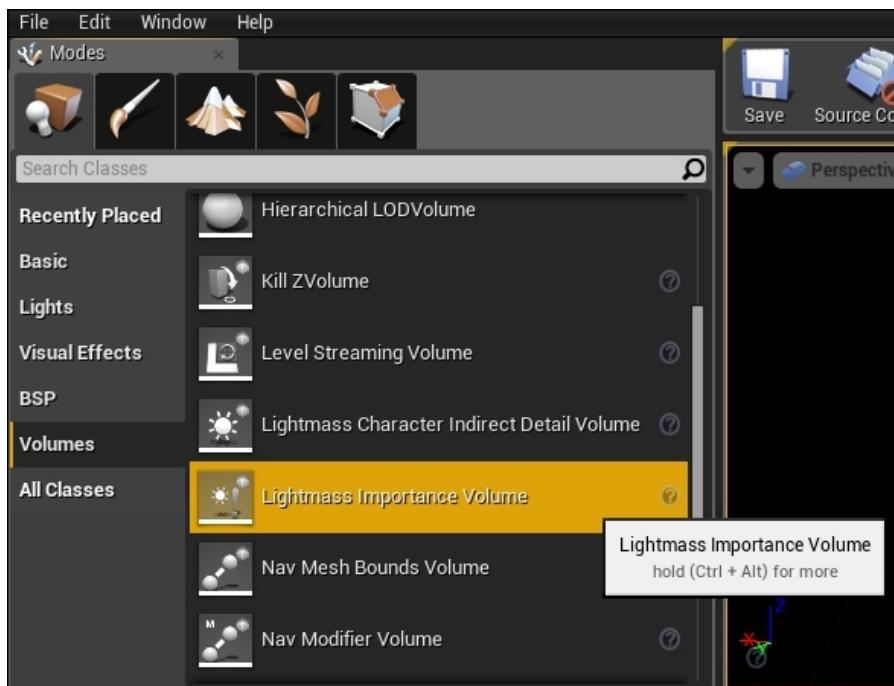


As we want to use a ceiling lamp prop as a floor lamp, you will need to rotate the lamp by rotating it about the *x* axis by 180 degrees. Set X = 180 degrees using the **Relative** mode. The following screenshot shows the rotated lamp positioned at one end of the room. Note that I have built the light and changed the view mode to the **Lit** mode. Feel free to position the lamp anywhere to see how it looks.

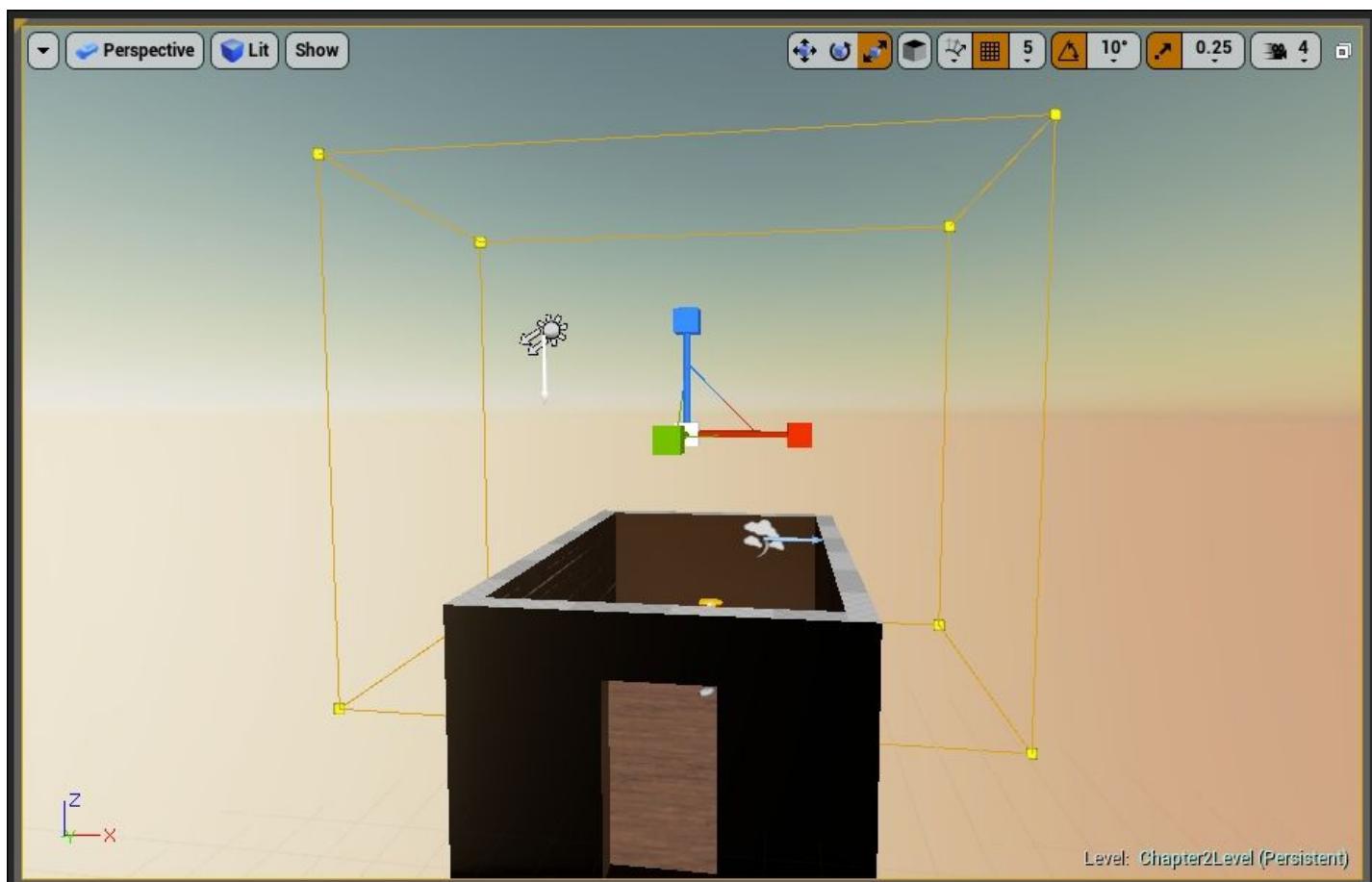


# Adding Lightmass Importance Volume

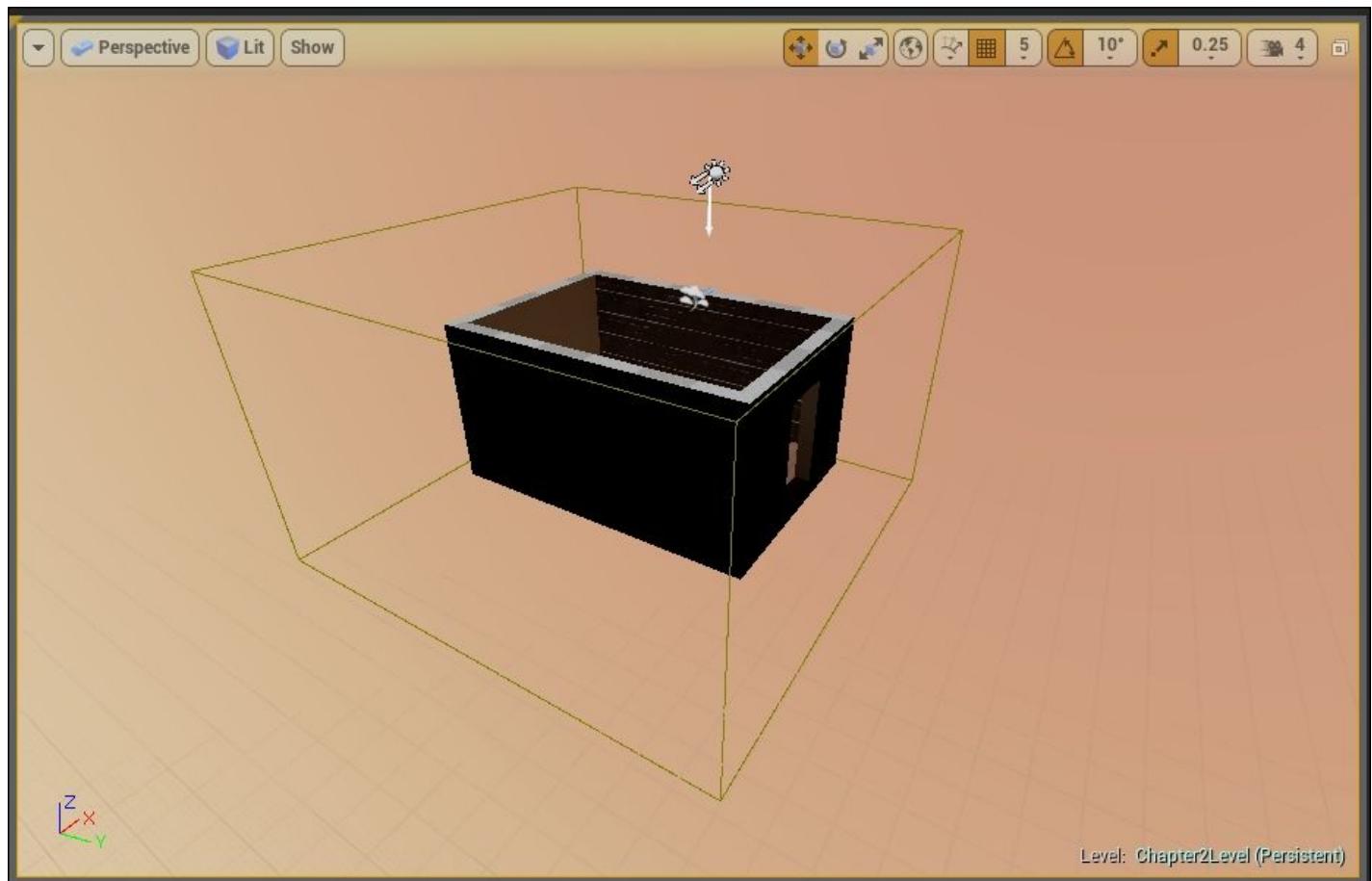
Since our room only takes up a small portion of the map, we can concentrate light on a small region by adding an item known as **Lightmass Importance Volume** to the map. The bounded box of the **Lightmass Importance Volume** tells the engine where light is needed in the map so it should encompass the entire area of the map that has objects. Drag and drop **Lightmass Importance Volume** into the map. Here, you can see where to find the **Lightmass Importance Volume**:



By default, the wireframe box that's been dropped (which is the **Lightmass Importance Volume**) is a cube. You will need to scale it to fit your room. With the **Lightmass Importance Volume** selected, press **R** to display the **Scale** tool. Use the **x**, **y**, and **z** axes to adjust the size of the box till it fits the level. The following screenshot shows the scaling of the box using the **Scale** tool:



After scaling and translating the box to fit the level, the **Lightmass Importance Volume** should look something like what is shown in the following screenshot, where the wireframe box is large enough to fit the room inside it. The size of the wireframe for the **Lightmass Importance Volume** can be larger than the map.



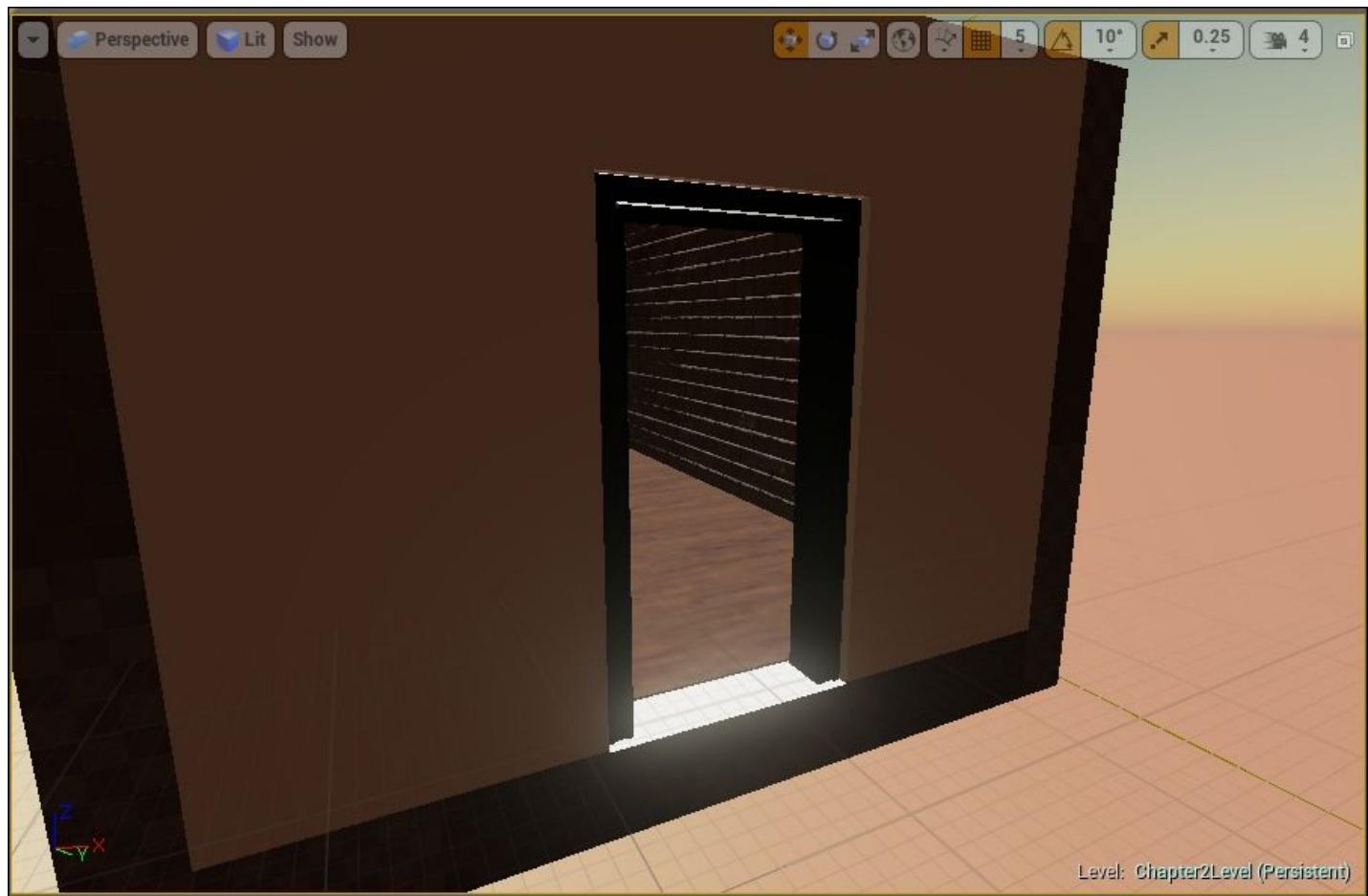
# Applying finishing touches to a room

Our room is almost complete. You would have noticed that the door now is just a hole in the wall. To make it look like a door, we need to add a door frame and a door as follows:

1. Go to **Content Browser** | **Content** | **StarterContent** | **Props**.
2. Click and drop **SM\_DoorFrame** into the viewport.
3. Adjust it to fit in the wall.

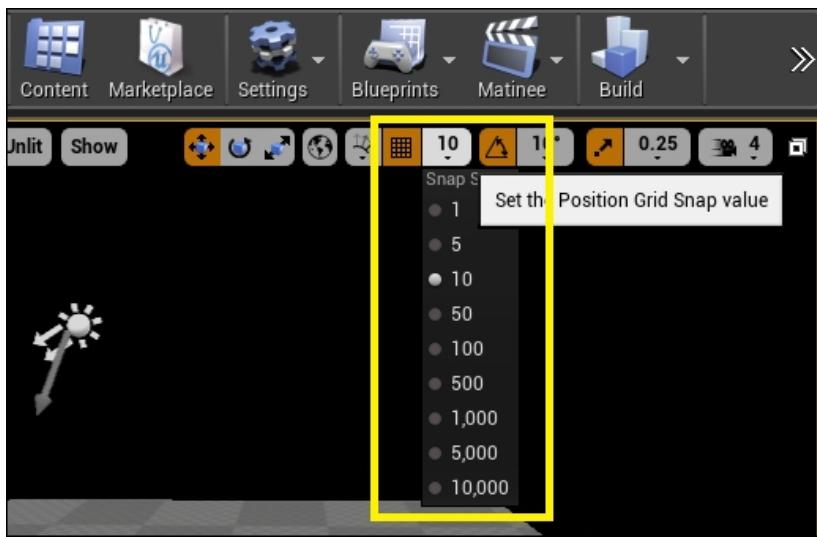
When done, it should look like what is shown in the following screenshot.

I've used different views, such as top, side, and front, to adjust the frame nicely to fit the door. You can adjust **Snap Sizes** for some fine-tuning.



## Useful tip – using the drag snap grid

To help you move objects into position more accurately, you can make use of the snap grid button at the top of the viewport as shown in the following screenshot. Turning the drag snap grid on allows you to translate objects according to the grid size you select. Click on the mesh-like symbol to toggle snap grid on/off. The numbers displayed on the right are the minimum grid sizes by which an object will move when translated.



I have also noticed that a portion of the floor is not textured yet. Use the same wood texture as you did previously to make sure that the ground is fully textured using **M\_Wood\_Floor\_Walnut\_Polished**.

Then, click and drag **SM\_Door** into the viewport. Rotate the door and fit it into the door frame in the same manner as shown previously. Here, you can see how the door is in place:



# Summary

We have learned how to navigate the viewport and set up/save a new map in a new project. We also created our first room with a door using the BSP Box brush, added materials to texture walls and floors, and learned to place static objects to enhance the look of the room. Although it is still kind of empty right now, we will continue to work on it in the next few chapters and expand on this map. In the next chapter, we will spice up the level by adding some objects that we can interact with.

# Chapter 3. Game Objects – More and Move

We created our first room in the Unreal Editor in [Chapter 2 , Creating Your First Level](#). In this chapter, we will cover some information about the structure of objects we have used to prototype the level in [Chapter 2 , Creating Your First Level](#). This is to ensure that you have a solid foundation in some important core concepts before moving forward. Then, we will progressively introduce various concepts to make the objects move upon a player's interaction.

In this chapter, we will cover the following topics:

- BSP Brush
- Static Mesh
- Texture and Materials
- Collision
- Volumes
- Blueprint

## BSP Brush

We used the BSP Box Brush in [Chapter 2 , Creating Your First Level](#), extensively to create the ground and the walls.

BSP Brushes are the primary building blocks for level creation in the game development. They are used for quick prototyping levels like how we have used them in [Chapter 2 , Creating Your First Level](#).

In Unreal, BSP Brushes come in the form of primitives (box, sphere, and so on) and also predefined/custom shapes.

### Background

BSP stands for **binary space partitioning**. The structure of a BSP tree allows spatial information to be accessed quickly for rendering, especially in 3D scenes made up of polygons. A scene is recursively divided into two, until each node of the BSP tree contains only polygons that can render in arbitrary order. A scene is rendered by traversing down the BSP tree from a given node (viewpoint).

Since a scene is divided using the BSP principle, placing objects in the level could be viewed as cutting into the BSP partitions in the scene. Geometry Brushes use **Constructive Solid Geometry ( CSG )** technique to create polygon surfaces. CSG combines simple primitives/custom shapes using Boolean operators such as union, subtraction, and intersection to create complex shapes in the level.

So, the CSG technique is used to create surfaces of the object in the level, and rendering the level is based on processing these surfaces using the BSP tree. This relationship has resulted in Geometry Brushes being known also as BSP Brushes, but more accurately, CSG surfaces.

### Brush type

BSP Brushes can either be additive or subtractive in nature. Additive brushes are like volumes that fill up the space. Additive brushes were used for the ground and the walls in our map in [Chapter 2 , Creating Your First Level](#).

Subtractive brushes can be used to form hollow spaces. These were used to create a hole in the wall in which to place a door and its frame in [Chapter 2 , Creating Your First Level](#).

### Brush solidity

For additive brushes, there are various states it can be in: solid, semi-solid, or non-solid.

Since subtractive brushes create empty spaces, players are allowed to move freely within them. Subtractive brushes can only be solid brushes.

Refer to the following table for comparison of their properties:

Brush solidity	Brush type	Degree of blocking	BSP cutting
Solid	Additive and subtractive	Blocks both players and projectiles	Creates BSP cuts to the surrounding world geometry
Semi-solid	Additive only	Blocks both players and projectiles	Does not cause BSP cuts to the surrounding world geometry
Non-solid	Additive only	Does not block players or projectiles	Does not cause BSP cuts to the surrounding world geometry

# Static Mesh

Static Mesh is a geometry made up of polygons. Looking more microscopically at what a mesh is made of, it is made up of lines connecting vertices.

Static Mesh has vertices that cannot be animated. This means is that you cannot animate a part of the mesh and make that part move relative to itself. But the entire mesh can be translated, rotated, and scaled. The lamp and the door that we have added in [Chapter 2](#), *Creating Your First Level*, are examples of Static Meshes.

A higher-resolution mesh has more polygons as compared to a lower-resolution mesh. This also implies that a higher resolution mesh has a larger number of vertices. A higher resolution mesh takes more time to render but is able to provide more details in the object.

Static Meshes are usually first created in external software programs, such as Maya or 3ds Max, and then imported into Unreal for placement in game maps.

The door, its frame, and the lamp that we added in [Chapter 2](#), *Creating Your First Level*, are Static Meshes. Notice that these objects are not simple geometry looking objects.

# BSP Brush versus Static Mesh

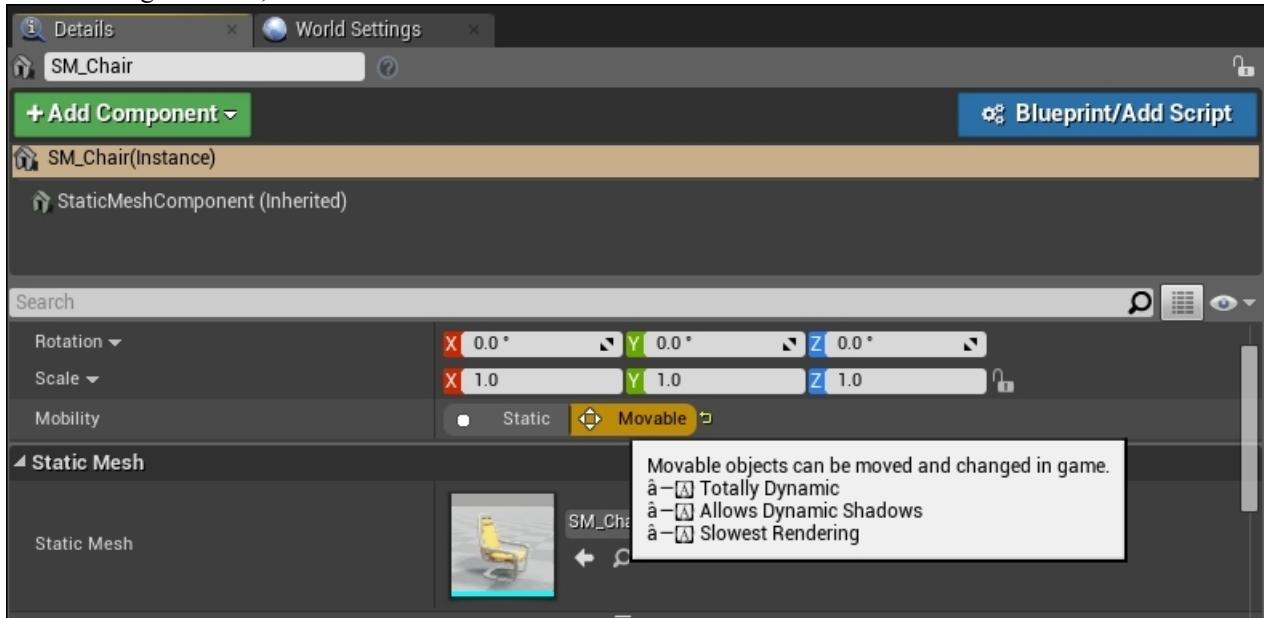
In game development, many objects in the game are Static Meshes. Why is that so? Static Mesh is considered more efficient, especially for a complex object with many vertices, as they can be cached to a video memory and are drawn by the computer's graphics card. So, Static Meshes are preferred when creating objects as they have better render performance, even for complex objects. However, this does not mean that BSP Brushes do not have a role in creating games.

When BSP Brush is simple, it can still be used without causing too much serious impact to the performance. BSP Brush can be easily created in the Unreal Editor, hence it is very useful for quick prototyping by the game/level designers. Simple BSP Brushes can be created and used as temporary placeholder objects while the actual Static Mesh is being modeled by the artists. The creation of a Static Mesh takes time, even more so for a highly detailed Static Mesh. We will cover a little information about the Static Mesh creation pipeline later in this chapter, so we have an idea of the amount of work that needs to be done to get a Static Mesh into the game. So, BSP Brush is great for an early game play testing without having to wait for all Static Meshes to be created.

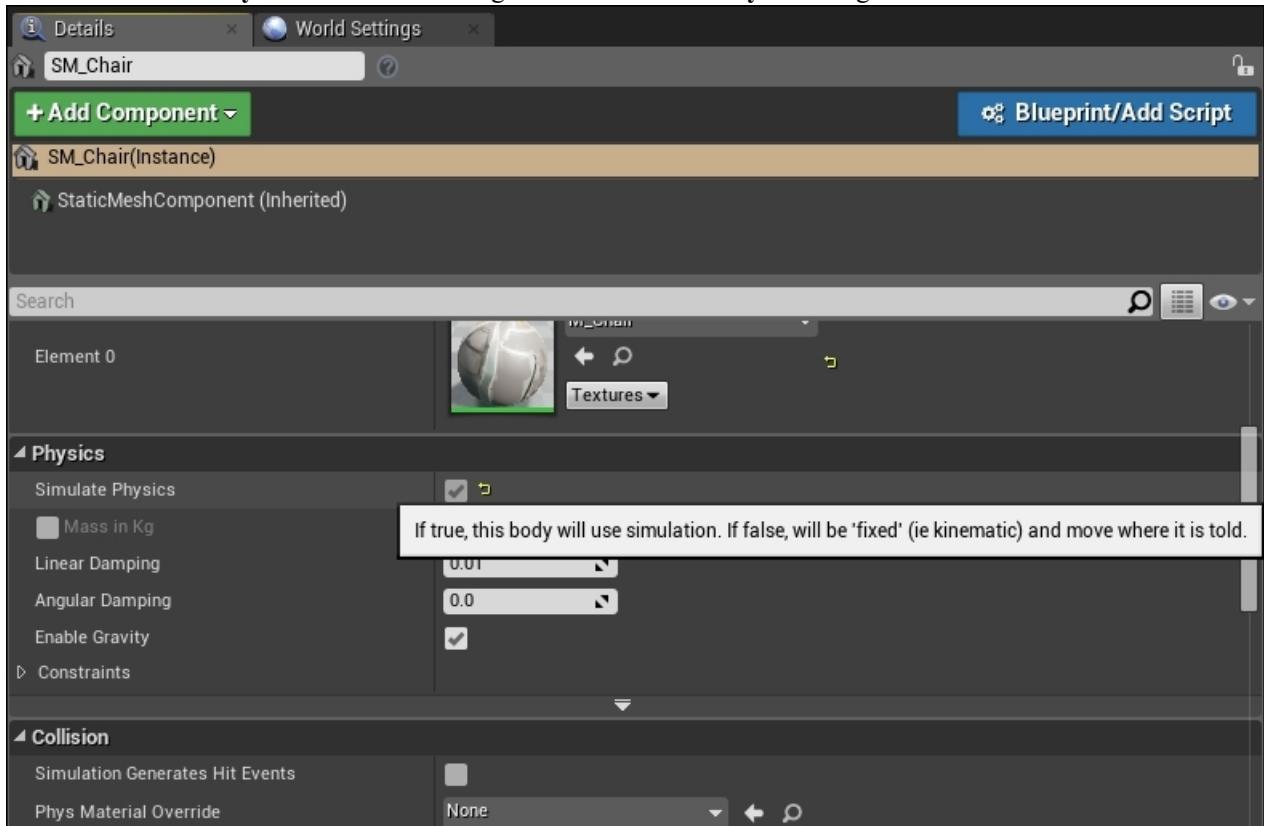
# Making Static Mesh movable

Let us open our saved map that we have created in [Chapter 2](#), *Creating Your First Level*, and let us first save the level as a new `Chapter3Level`.

1. Go to **Content Browser** | **Content** | **StarterContent** | **Props**, and search for **SM\_Chair**, which is a standard Static Mesh prop. Click and drag it into our map.
2. The chair we have in the level now is unmovable. You can quickly build and run the level to check it out. To make it movable, we need to change a couple of settings under the chair's details.
3. First, ensure **SM\_Chair** is selected, go to the **Details** tab. Go to **Transform** | **Mobility**, change it from **Static** to **Movable**. Take a look at the following screenshot, which describes how to make the chair movable:



4. Next, we want the chair to be able to respond to us. Scroll a little down the **Details** tab to change the **Physics** setting for the chair. Go to **Details** | **Physics**. Make sure the checkbox for **Simulate Physics** is checked. When this checkbox is checked, the auto-link setting sets the **Collision** to be a **PhysicsActor**. The following screenshot shows the **Physics** settings of the chair:



Let us now build and play the level. When you walk into the chair, you will be able to push it around. Just to note, the chair is still known as Static Mesh, but it is now movable.

# Materials

In [Chapter 2](#), *Creating Your First Level*, we selected a walnut polished material and applied it to the ground. This changed the simple dull ground into a brown polished wood floor. Using materials, we are able to change the look and feel of the objects.

The reason for a short introduction of materials here is because it is a concept that we need to have learned about before we can construct a Static Mesh. We already know that we need Static Meshes in the game and we cannot only rely on the limited selection that we have in the default map package. We will need to know how to create our own Static Meshes, and we rely heavily on Materials to give the Static Meshes their look and feel.

So, when do we apply Materials while creating our custom Static Mesh? Materials are applied to the Static Mesh during its creation process outside the editor, which we will cover in a later section of this chapter. For now, let us first learn how Materials are constructed in the editor.

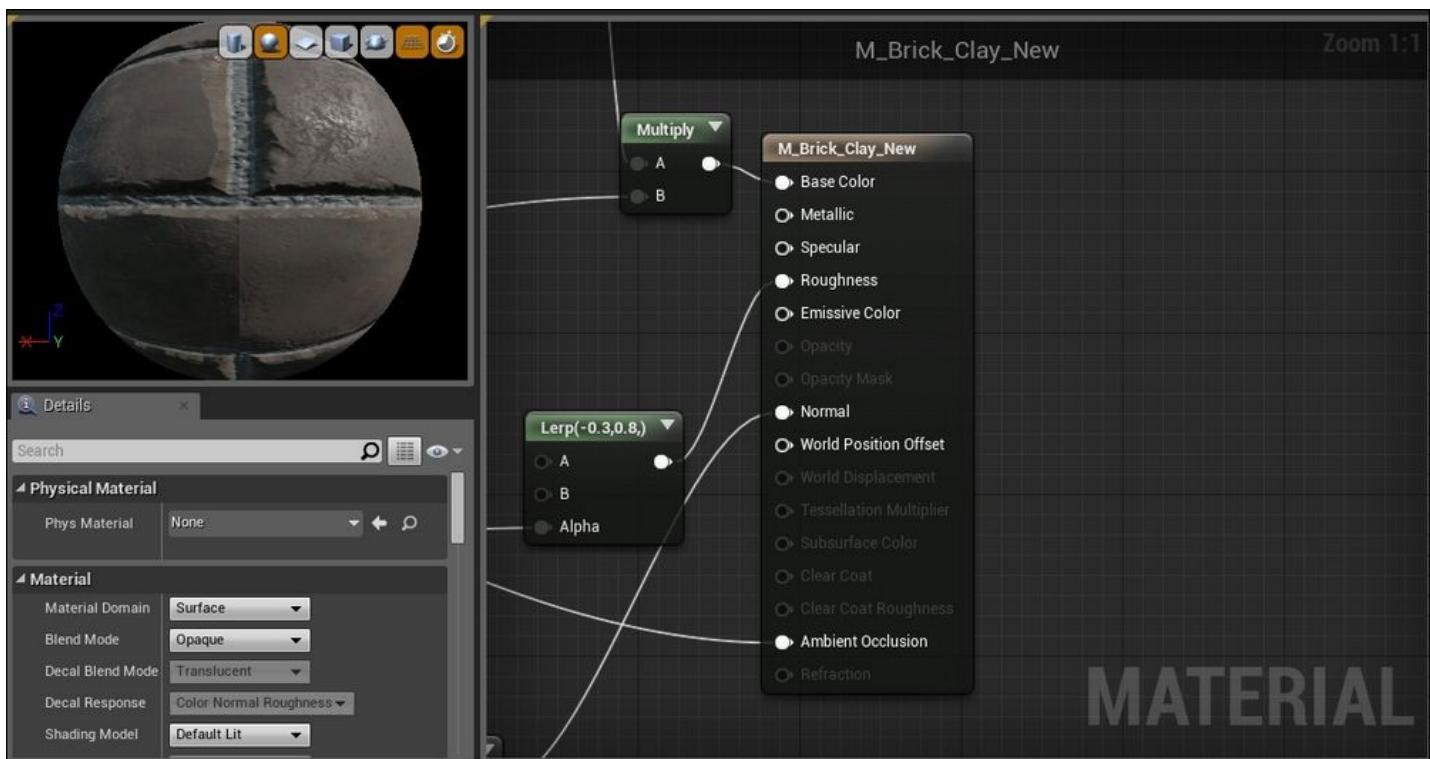
## Creating a Material in Unreal

To fully understand the concept of a Material, we need to break it down into its fundamental components. How a surface looks is determined by many factors, including color, presence of print/pattern/designs, reflectivity, transparency, and many more. These factors combine together to give the surface its unique look.

In Unreal Engine, we are able to create our very own material by using the Material Editor. Based on the explanation given earlier, a Material is determined by many factors and all these factors combine together to give the Material its own look and feel.

Unreal Engine offers a base Material node that has a list of customizable factors, which we can use to design our Material. By using different values to different factors, we can come up with our very own Material. Let us take a look at what is behind the scene in a material that we have used in [Chapter 2](#), *Creating Your First Level*.

Go to **Content Browser** | **Content** | **Starter Content** | **Materials** and double-click on **M\_Brick\_Clay\_New**. This opens up the Material Editor. The following screenshot shows the zoomed-in version of the base Material node for the brick clay material. You might notice that **Base Color**, **Roughness**, **Normal**, and **Ambient Occlusion** have inputs to the base **M\_Brick\_Clay\_New** material node. These inputs make the brick wall look like a brick wall.



The inputs to these nodes can take on values from various sources. Take **Base Color** for example, we can define the color using RGB values or we can take the color from the texture input. Textures are images in formats, such as `.bmp`, `.jpg`, `.png`, and so on, which we can create using tools, such as Photoshop or ZBrush.

We will talk more about the construction of the materials a little later in this book. For now, let us just keep in mind that materials are applied to the surfaces and textures are what we can use in combination, to give the materials its overall visual look.

## Materials versus Textures

Notice that I have used both Materials and Textures in the previous section. It has often caused quite a bit of confusion for a newbie in the game development. Material is what we apply to surfaces and they are made up of a combination of different textures. Materials take on the properties from the textures depending on what was specified, including color, transparency, and so on.

As explained earlier, Textures are simple images in formats such as `.tga`, `.bmp`, `.jpg`, `.png`, and so on.

## Texture/UV mapping

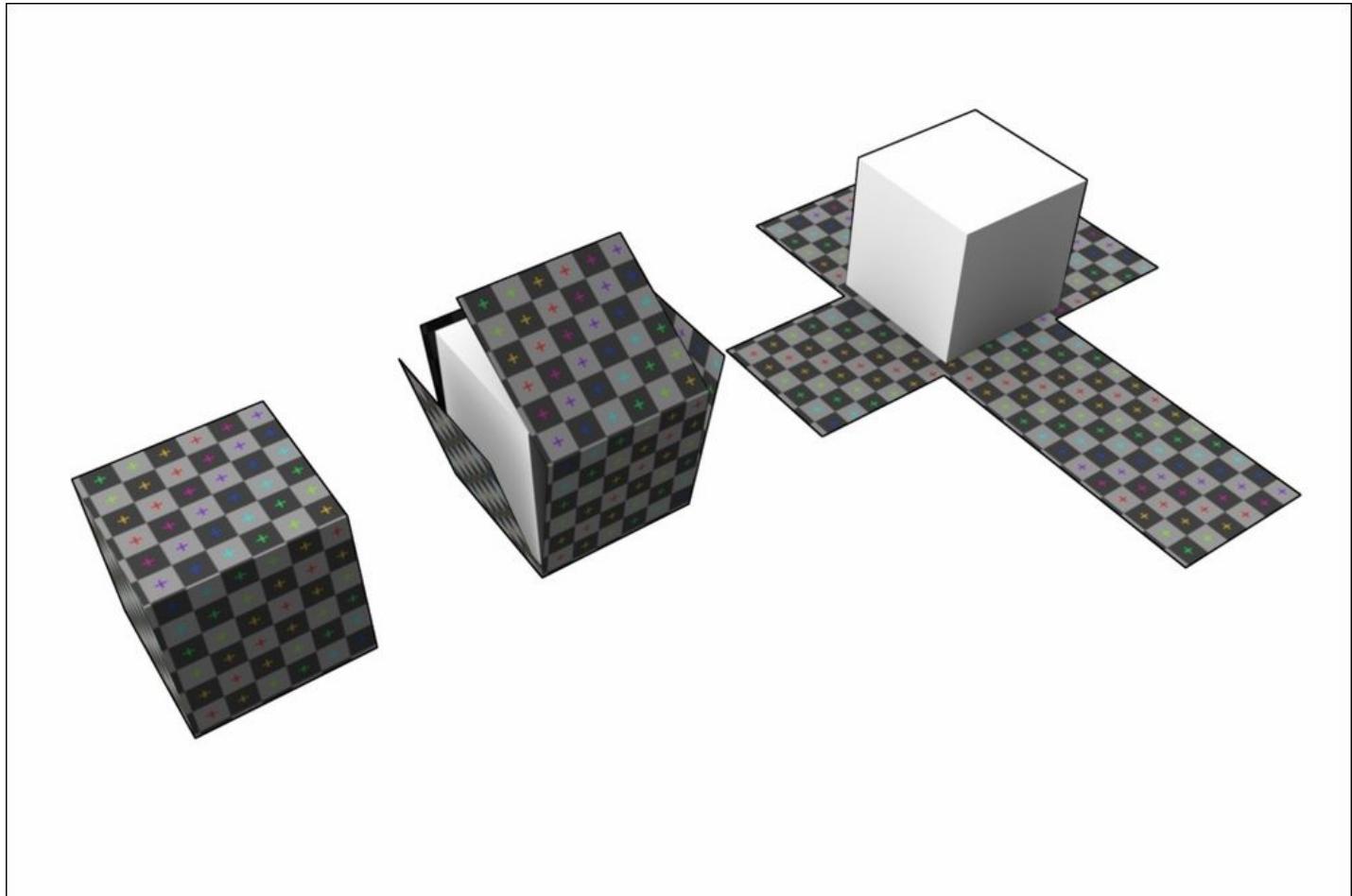
Now, we understand that a custom material is made up of a combination of textures and material is applied onto surfaces to give the polygon meshes its identity and realism. The next question is how do we apply these numerous textures that come with the material onto the surfaces? Do we simply slap them onto the 3D object?

There must be a predictable manner in which we paint these textures onto the surfaces. The method used is called **Texture Mapping**, which was pioneered by Edwin Catmull in 1974.

Texture mapping assigns pixels from a texture image to a point on the surface of the polygon. The texture image is called a **UV texture map**. The reason we are using UV as an alternative to the XY coordinates is because we are already using XY to describe the geometric space of the object. So the UV coordinates are the texture's XY coordinates, and it is solely used to determine how to paint a 3D surface.

### How to create and use a Texture Map

We will first need to unwrap a mesh at its seams and lay it out flat in 2D. This 2D surface is then painted upon to create the texture. This painted texture (also known as **Texture Map**) will then be wrapped back around the mesh by assigning the UV coordinates of the texture on each face of the mesh. To help you better visualize, take a look at the following illustration:



Source: Wikipedia ([https://en.wikipedia.org/wiki/UV\\_mapping](https://en.wikipedia.org/wiki/UV_mapping))

As a result of this, shared vertices can have more than one set of UV coordinates assigned.

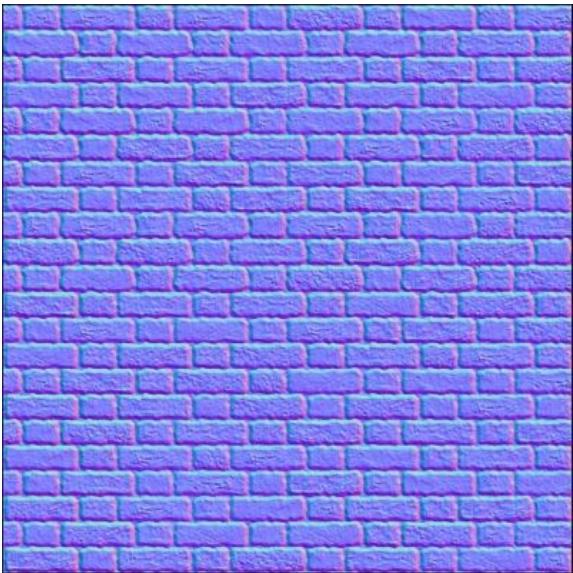
### Multitexturing

To create a better appearance in surfaces, we can use multiple textures to create the eventual end result desired. This layering technique allows for many different textures to be created using different combinations of textures. More importantly, it gives the artists better control of details and/or lighting on a surface.

### A special form of texture maps – Normal Maps

Normal Maps are a type of texture maps. They give the surfaces little bumps and dents. Normal Maps add the details to the surfaces without increasing the number of polygons. One very effective use of Normal Mapping is to generate Normal Maps from a high polygon 3D model and use it to texture the lower polygon model, which is also known as **baking**. We will discuss why we need the same 3D model with different number of polygons in the next section.

Normal maps are commonly stored as regular RGB images where the RGB components correspond to the X, Y, and Z coordinates, respectively, of the surface normal. The following image shows an example of a normal map taken from <http://www.bricksntiles.com/textures/>:

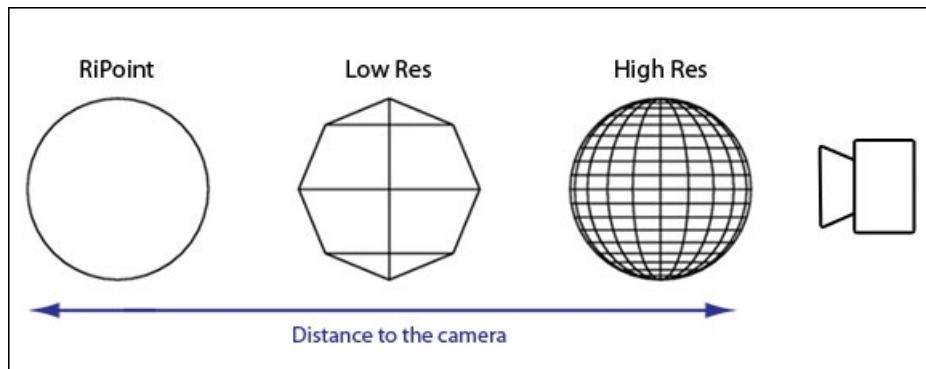


# Level of detail

We create objects with varying **level of details ( LODs )** to increase the efficiency of rendering. For objects that are closer to the player, high LODs objects are rendered. Objects with higher LODs have a higher number of polygons. For objects that are far away from the player, a simpler version of the object is rendered instead.

Artists can create different LOD versions of the 3D object using automated LOD algorithms, deployed through software or manually reducing the number of vertices, normals, edges in the 3D Models, to create a lower polygon count model. When creating models of different LODs, note that we always start by creating the most detailed model with the most number of polygons first and then reduce the number accordingly to create the other LOD versions. It is much harder to work the models the other way around. Do remember to keep the UV coherent when working with objects with different LODs. Currently, different LODs need to be light mapped separately.

The following image is taken from <http://renderman.pixar.com/view/level-of-detail> and very clearly shows the polygon count based on the distance away from the camera:

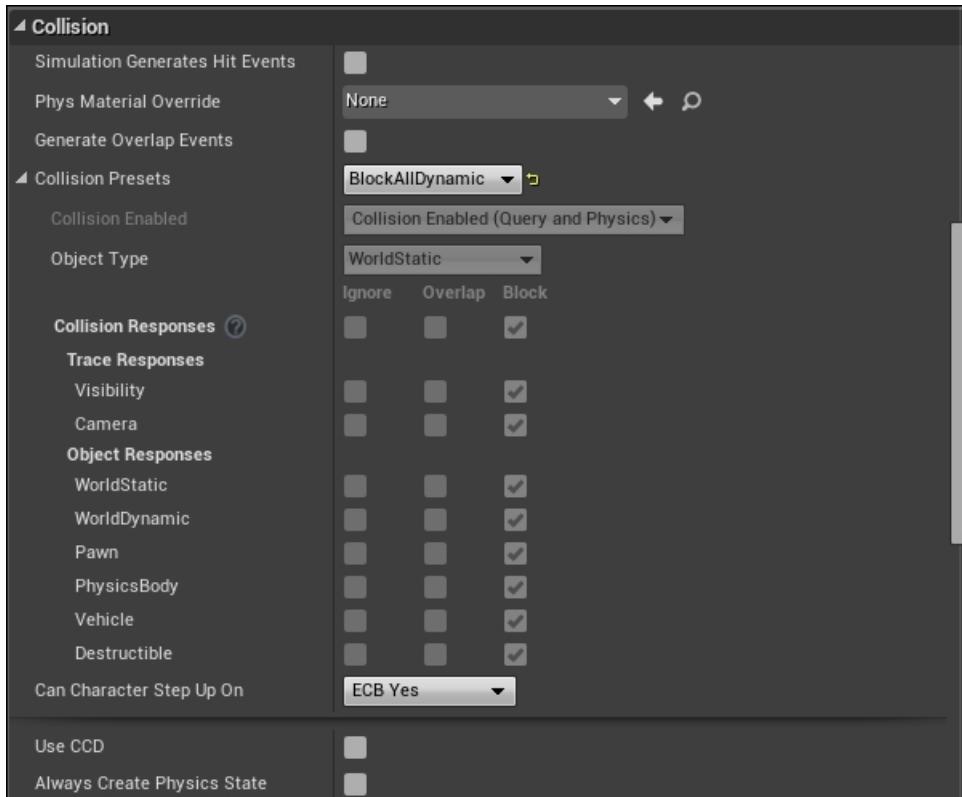


# Collisions

Objects in Unreal Engine have collision properties that can be modified to design the behavior of the object when it collides with another object.

In real life, collisions occur when two objects move and meet each other at a point of contact. Their individual object properties will determine what kind of collision we get, how they respond to the collision, and their path after the collision. This is what we try to achieve in the game world as well.

The following screenshot shows the collision properties available to an object in Unreal Engine 4:



If you are still confused about the concept of collision, imagine Static Mesh to give an object its shape (how large it is, how wide it is, and so on), while the collision of the object is able to determine the behavior of this object when placed on the table—whether the object is able to fall through the table in the level or lay stationary on the table.

## Collision configuration properties

Let us go through some of the possible configurations in Unreal's **Collision** properties that we should get acquainted with.

### Simulation Generates Hit Events

When an object has the **Simulation Generates Hit Events** flag checked, an alert is raised when the object has a collision. This alert notification can be used to trigger the onset of other game actions based on this collision.

### Generate Overlap Events

The **Generate Overlap Events** flag is similar to the **Simulation Generates Hit Events** flag, but when this flag is checked, in order to generate an event, all the object needs is to have another object to overlap with it.

### Collision Presets

The **Collision Presets** property contains a few frequently used settings that have been preconfigured for you. If you wish to create your own custom collision properties, set this to **Custom**.

### Collision Enabled

The **Collision Enabled** property allows three different settings: **No Collision**, **No Physics Collision**, and **Collision Enabled**. **No Physics Collision** is selected when this object is used only for non-physical types of collision such as raycasts, sweeps, and overlaps. **Collision Enabled** is selected when physics collision is needed. **No Collision** is selected when absolutely no collision is wanted.

### Object Type

Objects can be categorized into several groups: **WorldStatic** , **WorldDynamic** , **Pawn** , **PhysicsBody** , **Vehicle** , **Destructible** , and **Projectile** . The type selected determines the interactions it takes on as it moves.

## Collision Responses

The **Collision Responses** option sets the property values for all **Trace** and **Object Responses** that come with it. When **Block** is selected for **Collision Responses** , all the properties under **Trace** and **Object Responses** are also set to **Block** .

### Trace Responses

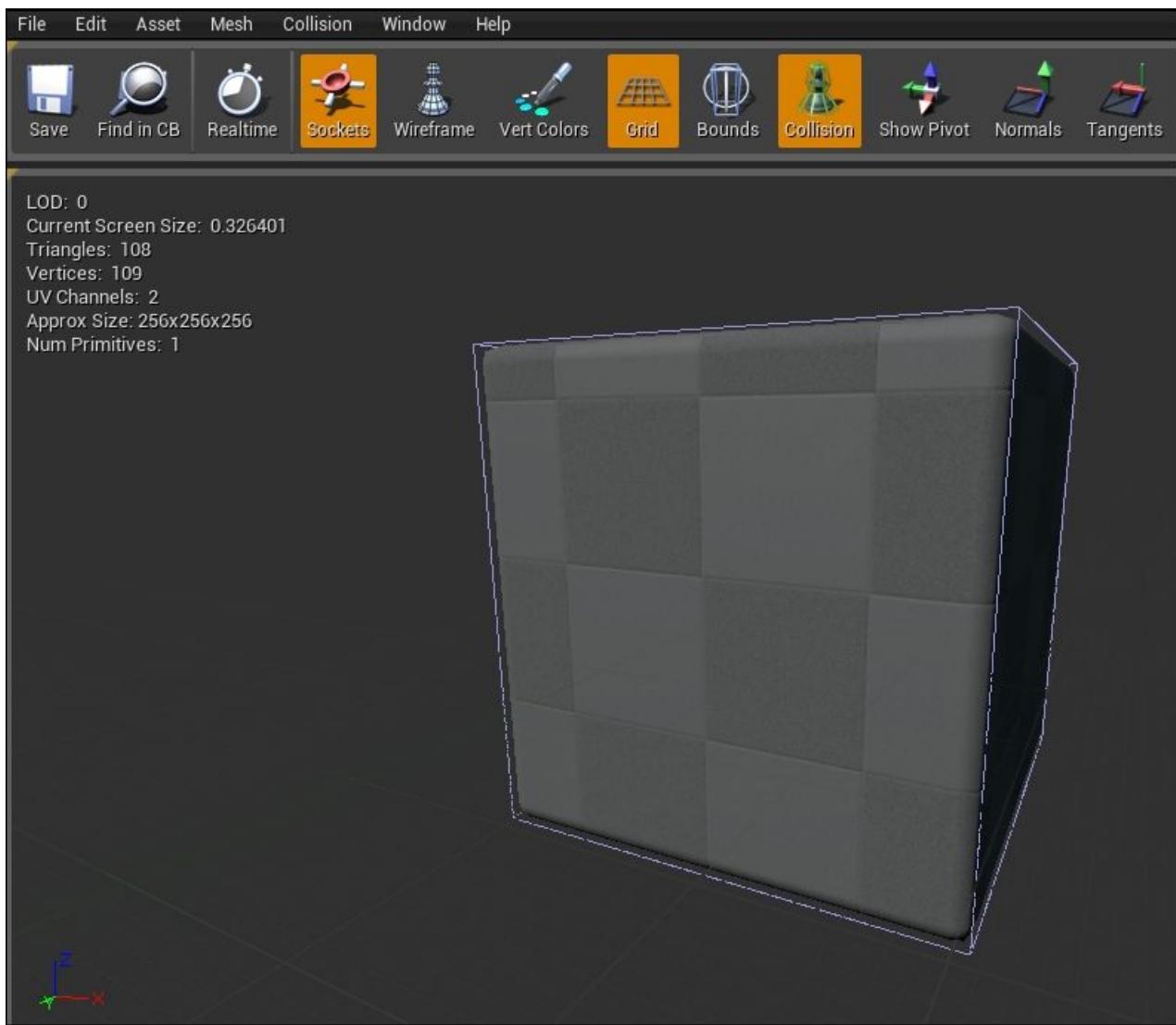
The **Trace Responses** option affects how the object interacts with traces. **Visibility** and **Camera** are the two types of traces that you can choose to block, overlap, or ignore.

### Object Responses

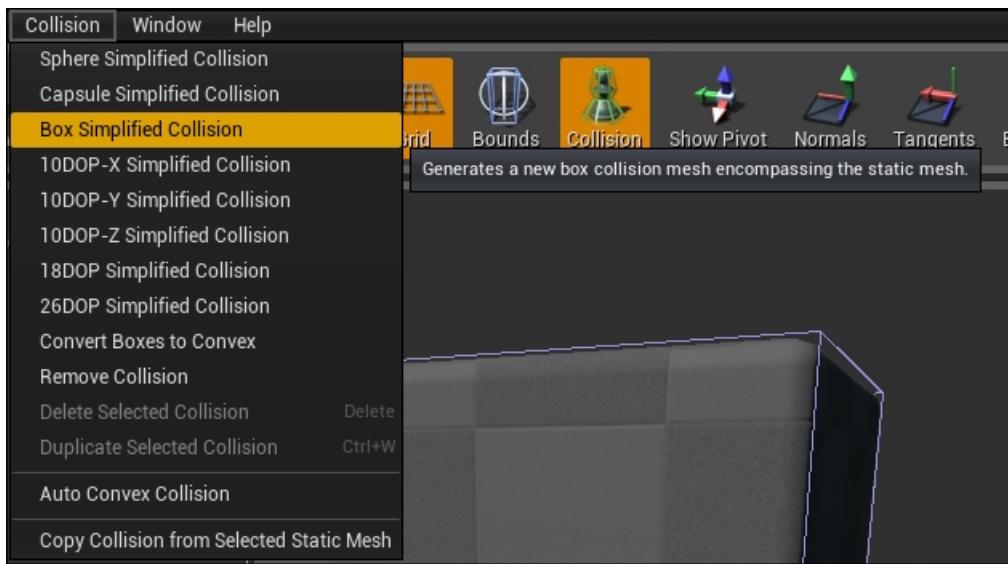
The **Object Responses** option affects how this object interacts with other object types. Remember the **Object Type** selection earlier? The **Object Type** property determines the type of object, and under this category, you can configure the collision response this object has with the different types of objects.

## Collision hulls

For a collision to occur in Unreal Engine, hulls are used. To view an example of the collision hull for a Static Mesh, take a look at the light blue lines surrounding the cube in the following screenshot; it's a box collision hull:



Hulls can be generated in Static Mesh Editor for static meshes. The following screenshot shows the menu options available for creating an auto-generated collision hull in Static Mesh Editor:



Simple geometry objects can be combined and overlapped to form a simple hull. A simple hull/bounding box reduces the amount of calculation it needs during a collision. So for complex objects, a generalized bounding box can be used to encompass the object. When creating static mesh that has a complex shape, not a simple geometry type of object, you will need to refer to the *Static Mesh creation pipeline* section later on in the chapter to learn how to create a suitable collision bounding box for it.

## Interactions

When designing collisions, you will also need to decide what kind of interactions the object has and what it will interact with.

To block means they will collide, and to overlap can mean that no collision will occur. When a block or an overlap happens, it is possible to flag the event so that other actions resulting from this interaction can be taken. This is to allow customized events, which you can have in game.

Note that for a block to actually occur, both objects must be set to **Block** and they must be set so that they block the right type of objects too. If one is set to block and the other to overlap, the overlap will occur but not the block. Block and overlap can happen when objects are moving at a high speed, but events can only be triggered on either overlap or block, not both. You can also set the blocking to ignore a particular type of object, for example, **Pawn**, which is the player.

# Static Mesh creation pipeline

Static Mesh creation pipeline is done outside of the editor using 3D modeling tools such as Autodesk's Maya and 3D's Max. Unreal Engine 4 is compatible to import the FBX 2013 version of the files.

This creation pipeline is used mainly by the artists to create game objects for the project.

The actual steps and naming convention when importing Static Mesh into the editor are well documented on the Unreal 4 documentation website. You may refer to <https://docs.unrealengine.com/latest/INT/Engine/Content/FBX/StaticMeshes/index.html> for more details.

# Introducing volumes

Volumes are invisible areas that are created to help the game developers perform a certain function. They are used in conjunction with the objects in the level to perform a specific purpose. Volumes are commonly used to set boundaries that are intended to prevent players from gaining access to trigger events in the game, or use the Lightmass Importance Volume to change how light is calculated within an area in the map as in [Chapter 2, Creating Your First Level](#).

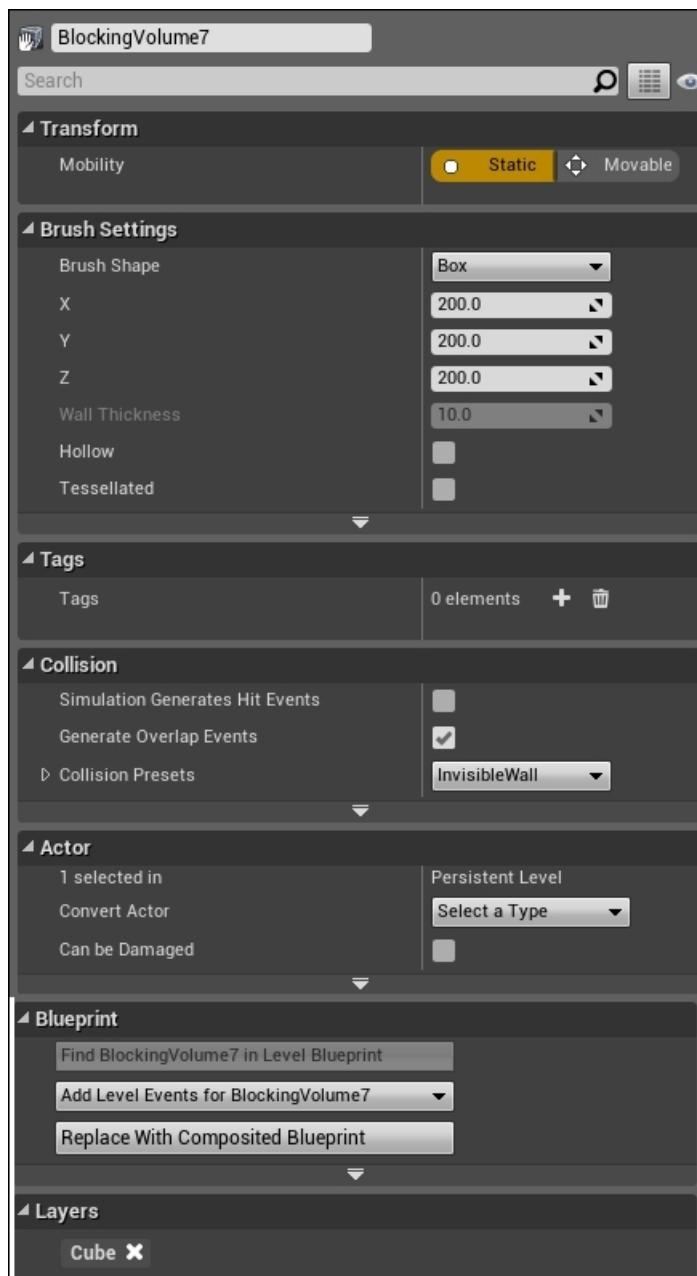
Here's a list of the different types of volumes that can be customized and used in Unreal Engine 4. But feel free to quickly browse through each of the volumes here for now, and revisit them later when we start learning how to use them later in the book. For this chapter, you may focus your attention first on the Trigger Volume, as we will be using that in the later examples of this chapter.

## Blocking Volume

The Blocking Volume can be used to prevent players/characters/game objects from entering a certain area of the map. It is quite similar to collision hull which we have described earlier and can be used in place of Static Mesh collision hull, as they are simpler in shapes (block shapes), hence easier to calculate the response of the collision. These volumes also have the ability to detect which objects overlap with themselves quickly.

An example of the usage of the Blocking Volume is to prevent the player from walking across a row of low bushes. In this case, since the bushes are rather irregularly shaped but are roughly forming a straight line, like a hedge, an invisible Blocking Volume would be a very good way of preventing the player from crossing the bushes.

The following screenshot shows the properties for the Blocking Volume. We can change the shape and size of the volume under **Brush Settings**. Collision events and triggers other events using Blueprint. This is pretty much the basic configuration we will get for all other volumes too.



## Camera Blocking Volume

The Camera Blocking Volume works in the same way as the Blocking Volume but it is used specifically to block cameras. It is useful when you want to limit the player from exploring with the camera beyond a certain range.

## Trigger Volume

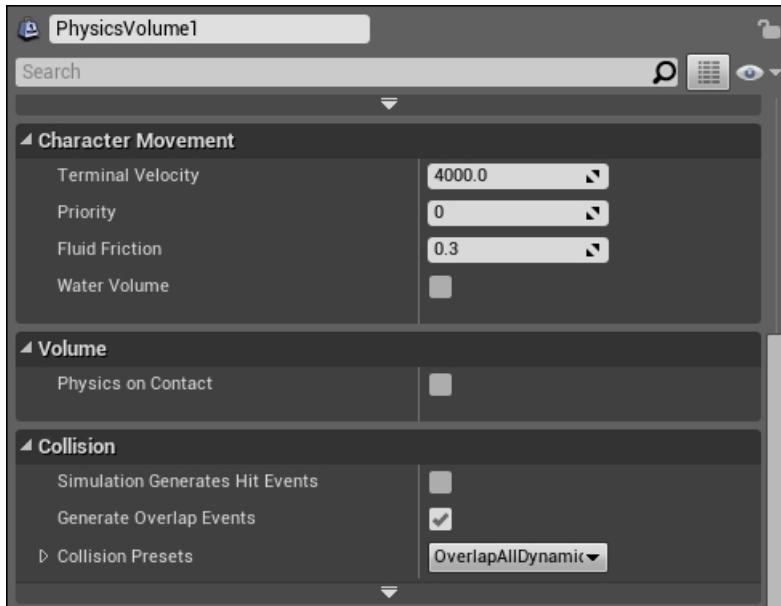
The Trigger Volume is probably one of the most used volumes. This is also the volume which we would be using to create events for the game level that we have been working on. As the name implies, upon entering this volume, we can trigger events, and via Blueprint, we can create a variety of events for our game, such as moving an elevator or spawning NPCs.

## Nav Mesh Bounds Volume

The Nav Mesh Bounds Volume is used to indicate the space in which NPCs are able to freely navigate around. NPCs could be enemies in the game who need some sort of path finding method to get around the level on their own. This Nav Mesh Bounds Volume will set up the area in the game that they are able to walk through. This is important as there could be obstacles such as bridges that they will need to use to in order get across to the other side (instead of walking straight into the river and possibly drowning).

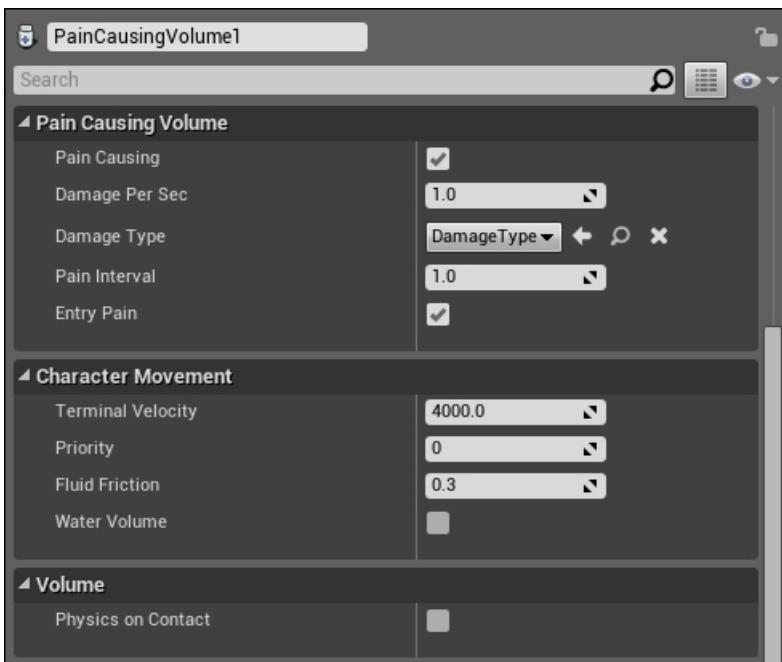
## Physics Volume

The Physics Volume is used to create areas in which the physics properties of the player/objects in the level change. An example of this would be altering the gravity within a space ship only when it reaches the orbit. When the gravity is changed in these areas, the player starts to move slower and float in the space ship. We can then turn this volume off when the ship comes back to earth. The following screenshot shows the additional settings we get from the Physics Volume:



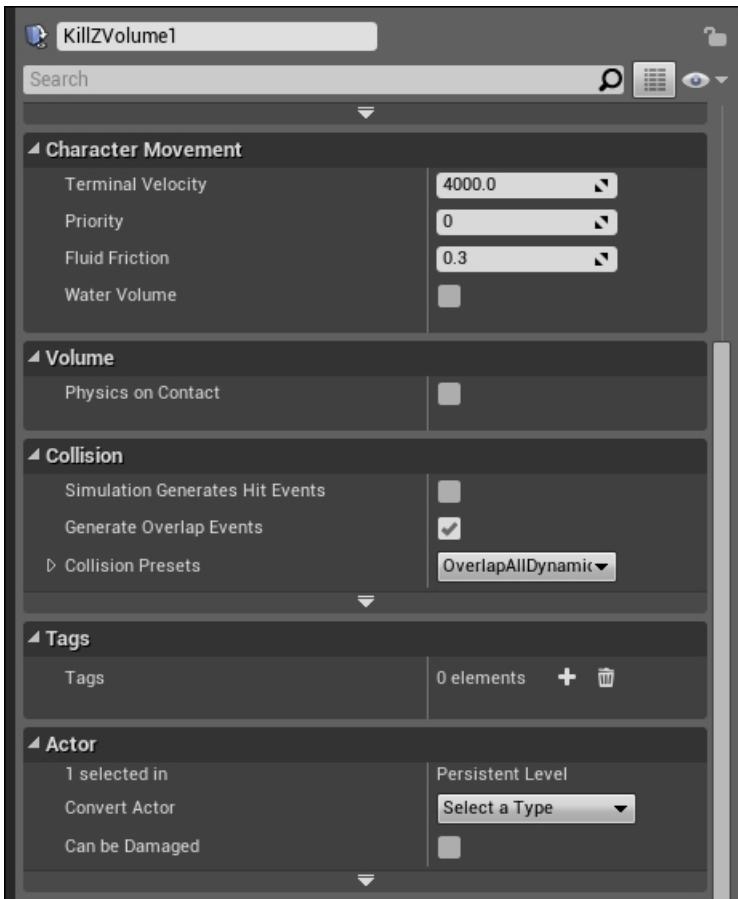
## Pain Causing Volume

The Pain Causing Volume is a very specialized volume used to create damage to the players upon entry. It is a "milder" version of the Kill Z Volume. Reduction of health and the amount of damage per second are customizable, according to your game needs. The following screenshot shows the properties you can adjust to control how much pain to inflict on the player:



## Kill Z Volume

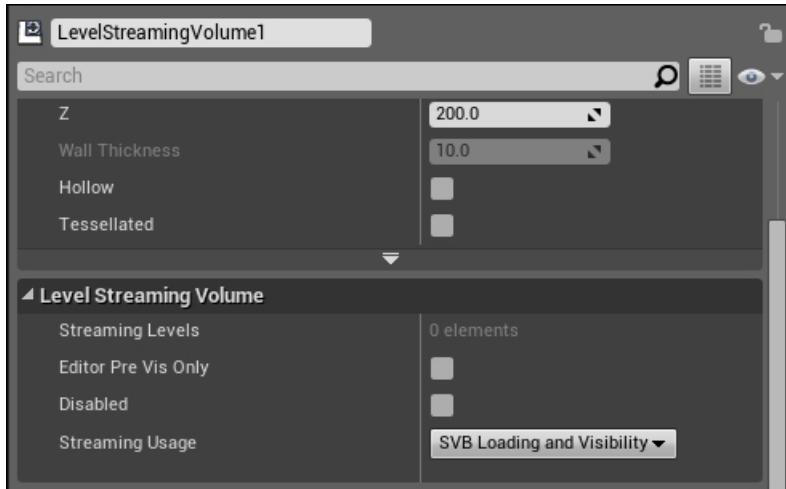
We kill the player when it enters the Kill Z Volume. This is a very drastic volume that kills the player immediately. An example of its usage is to kill the player immediately when the player falls off a high building. The following screenshot shows the properties of Kill Z Volume to determine the point at which the player is killed:



## Level Streaming Volume

The Level Streaming Volume is used to display the levels when you are within the volume. It generally fills the entire space where you want the level to be loaded. The reason we need to stream levels is to give players an illusion that we have a large open game level, when in fact the level is broken up into chunks for more efficient rendering. The following screenshot shows the properties that can be configured for the Level Streaming

## Volume:

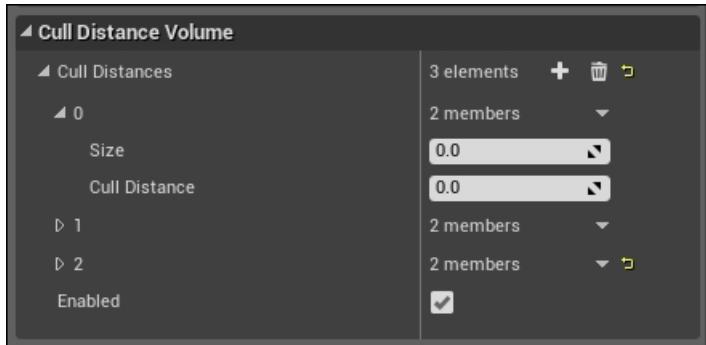


## Cull Distance Volume

The Cull Distance Volume allows objects to be culled in the volume. The definition of cull is to select from a group. The Cull Distance Volume is used to select objects in the volume that need to disappear (or not rendered) based on the distance away from the camera. Tiny objects that are far away from the camera cannot be seen visibly. These objects can be culled if the camera is too far away from those objects. Using the Cull Distance Volume, you would be able to decide upon the distance and size of objects, which you want to cull within a fixed space. This can greatly improve performance of your game when used effectively.

This might seem very similar to the idea of occlusion. Occlusion is implemented by selecting object by object, when it is not rendered on screen. These are normally used for larger objects in the scene. Cull Distance Volume can be used over a large area of space and using conditions to specify whether or not the objects are rendered.

The following screenshot shows the configuration settings that are available to the Cull Distance Volume:



## Audio Volume

The Audio Volume is used to mimic real ambient sound changes when one transits from one place to another, especially when transitioning to and from very different environments, such as walking into a clock shop from a busy street, or walking in and out of a restaurant with a live band playing in the background.

The volume is placed surrounding the boundaries of one of the areas creating an artificial border dividing the spaces into interior and exterior. With this artificially created boundary and settings that come with this Audio Volume, sound artists are able to configure how sounds are played during this transition.

## PostProcess Volume

The PostProcess Volume affects the overall scene using post-processing techniques. Post-processing effects include Bloom effects, Anti-Aliasing, and Depth of Field.

## Lightmass Importance Volume

We have used Lightmass Importance Volume in [Chapter 2](#), *Creating Your First Level*, to focus the light on the section of the map that has the objects in. The size of the volume should encompass your entire level.

# Introducing Blueprint

The Unreal Editor offers the ability to create custom events for game levels through a visual scripting system. Before Unreal Engine 4, it was known as the **Kismet system**. In Unreal Engine 4, this system was revamped with more features and capabilities. The improved system was launched with the new name of Blueprint.

There are several types of Blueprint: Class Blueprint, Data-Only Blueprint, and Level Blueprint. These are more or less equivalent to what we used to know as Kismet, which is now known as Level Blueprint.

Why do I need Blueprint? The simple answer is that through Blueprint, we are able to control gameplay without having to dive into the actual coding. This makes it convenient for non-programmers to design and modify the gameplay. So, it mainly benefits the game designers/artists who can configure the game through the Blueprint editor.

So, how can we use Blueprint and what can I use Blueprint for? Blueprint is just like coding with an interface. You can select, drag, and drop function nodes into the editor, and link them up logically to evoke the desired response to specified scenarios in your game. For programmers, they will be able to pick it up pretty quickly, since Blueprint is in fact coding but through a visual interface.

For the benefit of everyone who is new to Unreal Engine 4 and maybe programming as well, we will go through a basic example of how Level Blueprint works here and use that as an example to go through some basic programming concepts at the same time.

What will we be using Blueprint for? Blueprint has the capabilities to prototype, implement, or modify virtually any gameplay element. The gameplay elements affect how game objects are spawned, what gets spawned, where they are spawned, and under what conditions they are spawned. The game objects can include lights, camera, player's input, triggers, meshes, and character models. Blueprint can control properties of these game objects dynamically to create countless gameplay scenarios. The examples of usage include altering the color of the lights when you enter a room in the game, triggering the door to shut behind you after entering the room and playing the sound effect of the door closing shut, spawning weapons randomly among three possible locations in the map, and so on.

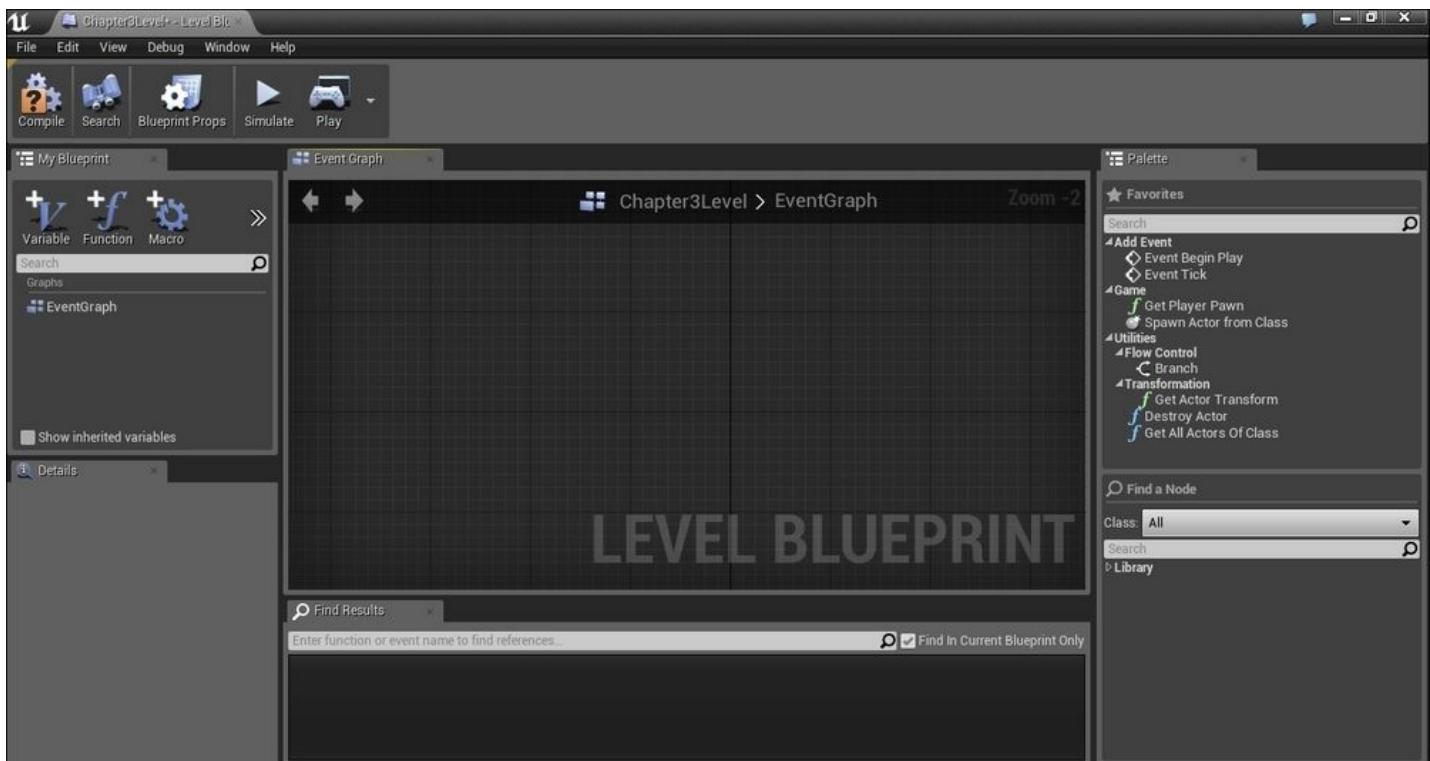
In this chapter, we will focus on Level Blueprint first, since it is the most commonly used form of Blueprint.

## Level Blueprint

Level Blueprint is a type of Blueprint that has influence over what happens in the level. Events that are created in this Blueprint affect what happens in the level, and are made specific to the situation by specifying the particular object it targets.

Feel free to jump to the next section first where we will go through a Blueprint example, so that we are able to understand Level Blueprint a little better.

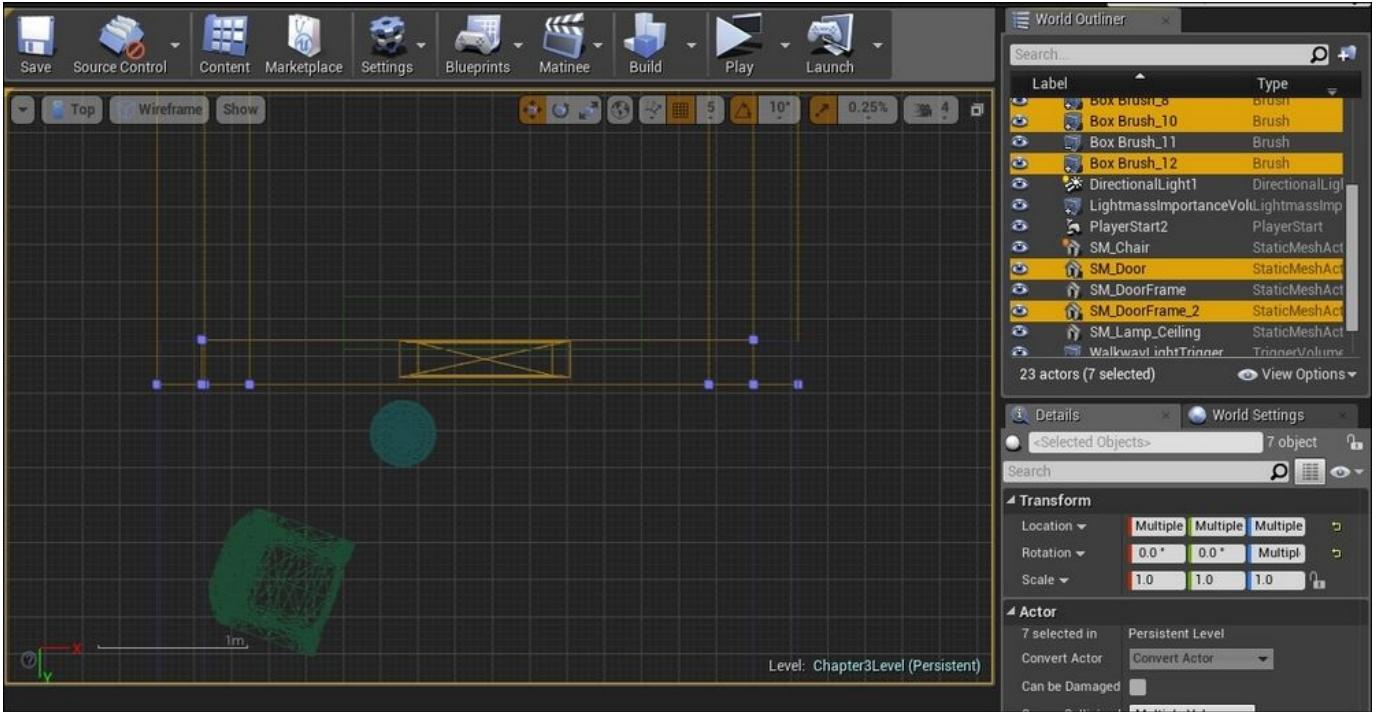
The following screenshot shows a blank Level Blueprint. The most used window is **Event Graph**, which is in the center. Using different node types in **Event Graph** and linking it up appropriately creates a responsive interaction within the game. The nodes come with variables, values, and other similar properties used in programming to control the game events graphically (without writing a single line of script or code).



# Using the Trigger Volume to turn on/off light

We are now ready to use what we have learned to construct the next room for our game. We will duplicate the first room we have created in order to create our second room.

1. Open the level that we created in [Chapter 2, Creating Your First Level](#), (`Chapter2_Level`) and save it as a new level called `Chapter3_Level`.
2. Select all the walls, the floor, the door, and the door frame.
3. Hold down `Alt + Shift` and drag to duplicate the room.
4. Place the duplicated room with the duplicated door aligned to the wall of the first room. Refer to the following screenshot to see how the walls are aligned from a **Top** view perspective:



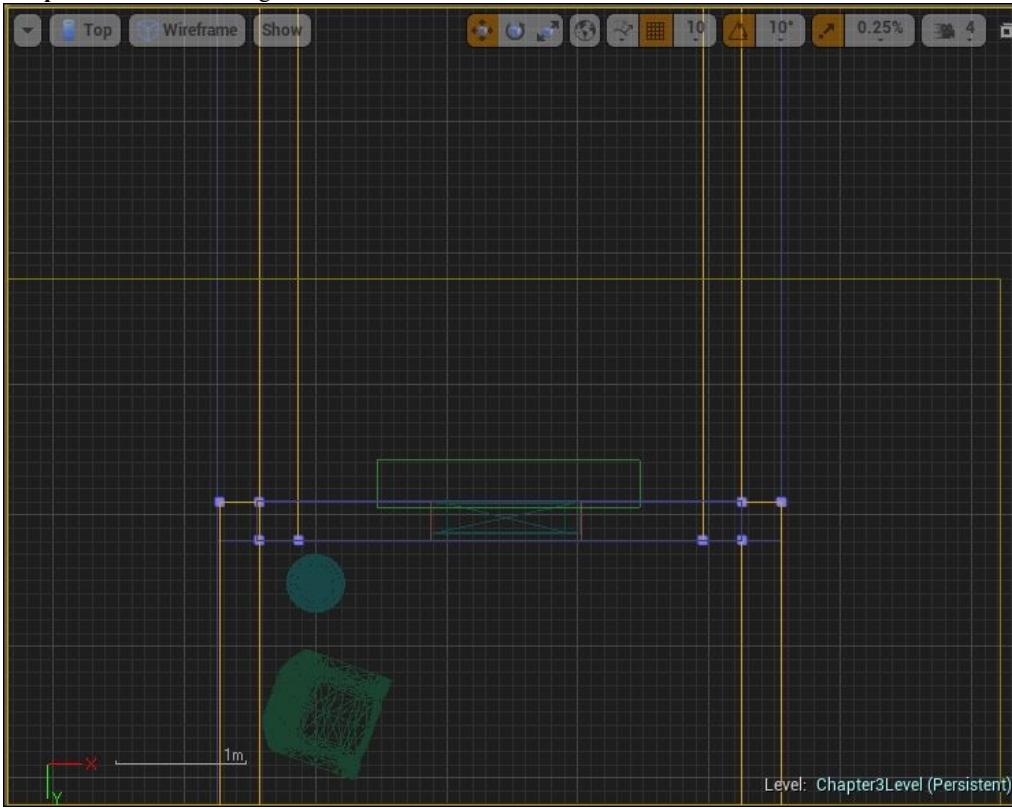
5. Delete the back wall of the first room to link both the rooms.
6. Delete all the doors to allow easy access to the second room.
7. Move the standing lamp and chair to the side. Take a look the following screenshot to understand how the rooms look at this point:

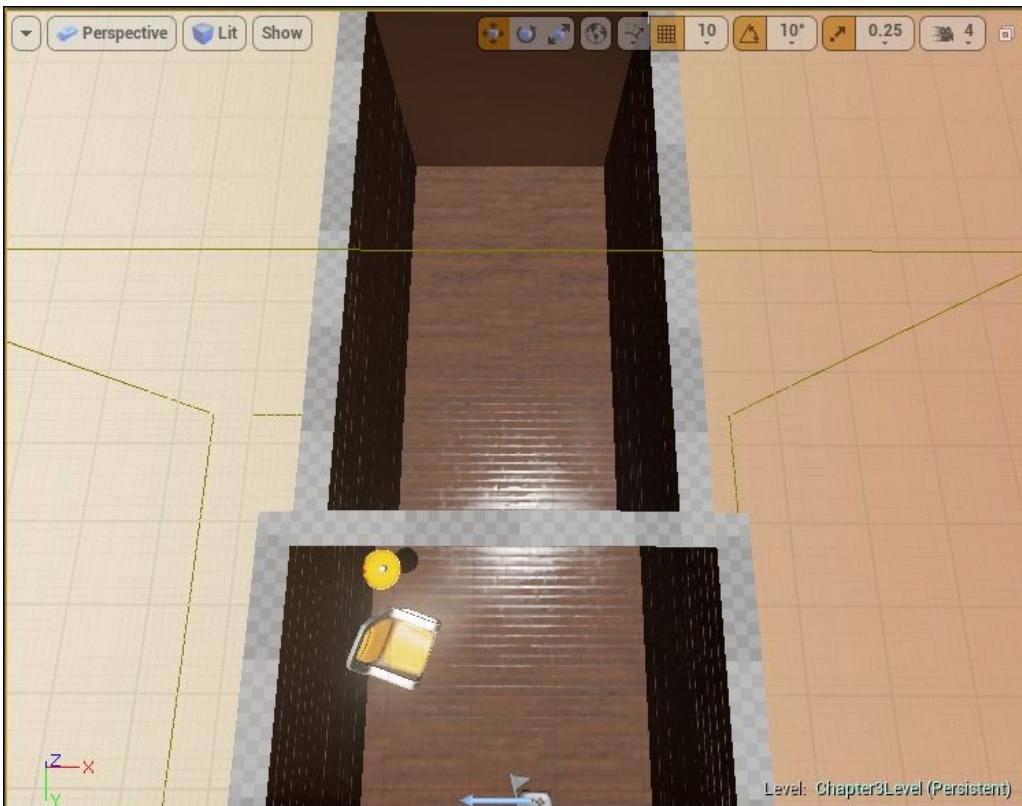


8. Rebuild the lights. The following screenshot shows the room correctly illuminated after building the lights:

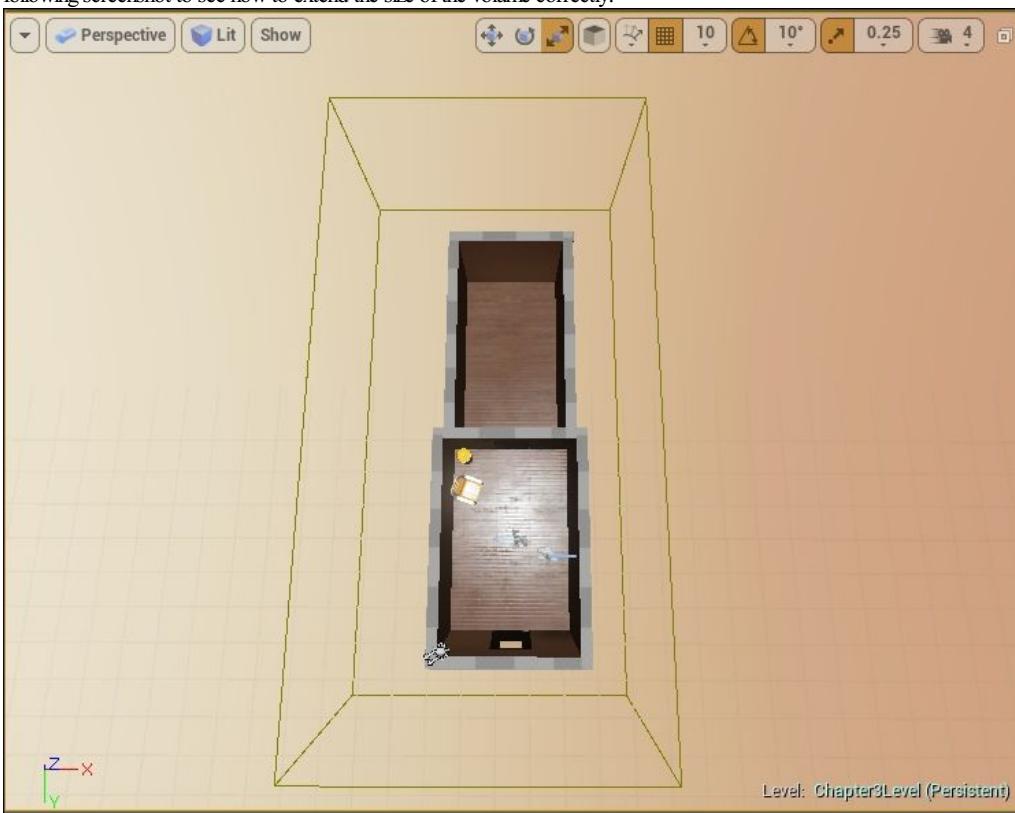


9. Now, let us focus on working on the second room. We will create a narrower walkway using the second room that we have just created.
10. Move the sidewalls closer to each other—about 30 cm from the previous sidewall towards the center. Refer to the next two screenshots for the **Top** and **Perspective** views after moving the sidewalls:

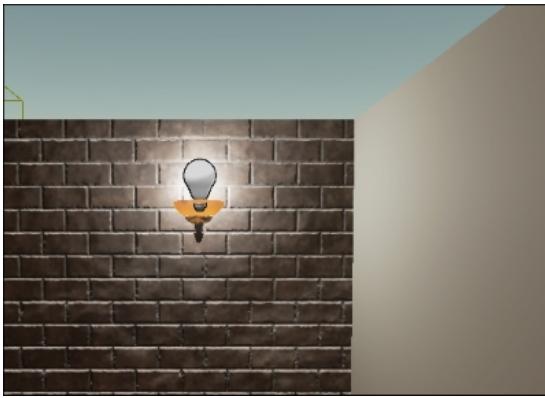




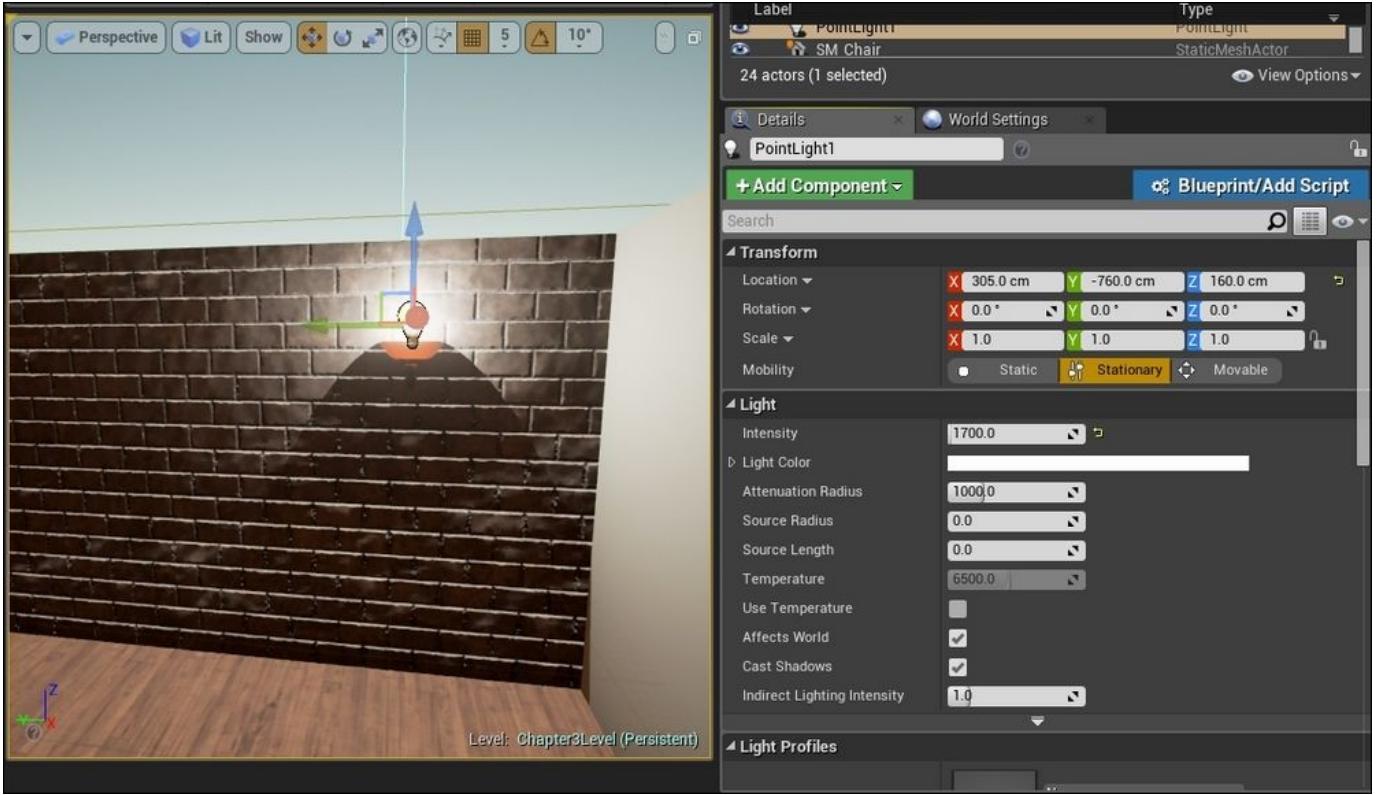
11. Note that LightMass Importance Volume is not encompassing the entire level now. Increase the size of the volume to cover the whole level. Take a look at the following screenshot to see how to extend the size of the volume correctly:



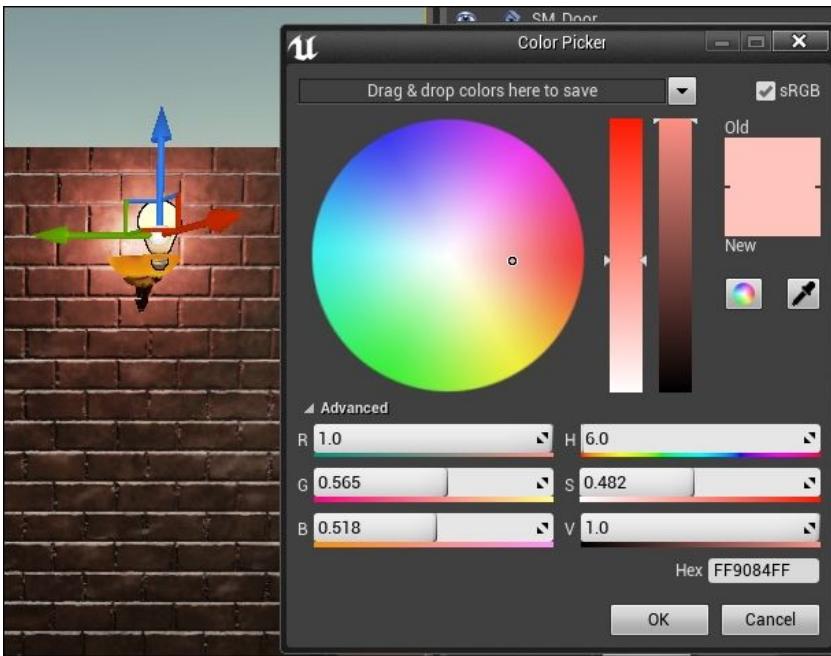
12. Go to **Content Browser** | **Props**. Click and drop **SM\_Lamp\_Wall** into the level. Rotate the lamp if necessary so that it lies nicely on the side wall.
13. Go to **Modes** | **Lights**. Click and drop a Point Light into the second room. Place it just above the light source on the wall light, which we added in the previous step. Take a look at the following screenshot to see the placement of the lamp and Point Light that we have just added:



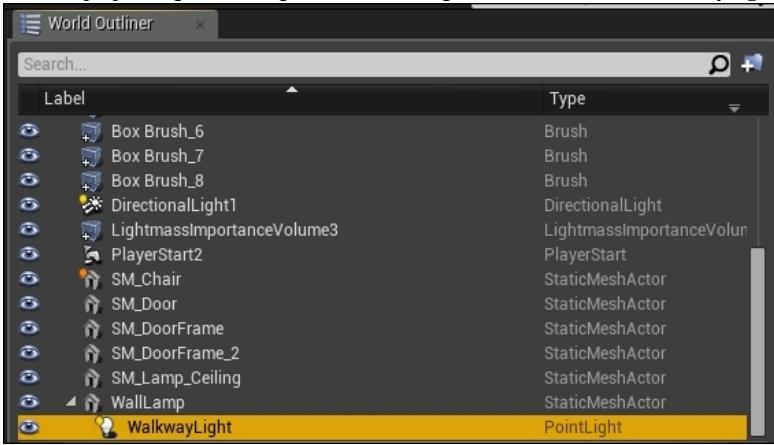
14. Adjust the Point Light settings: Intensity = 1700.0. This is approximately the light intensity coming off a light bulb. The following screenshot shows the settings for the Point Light:



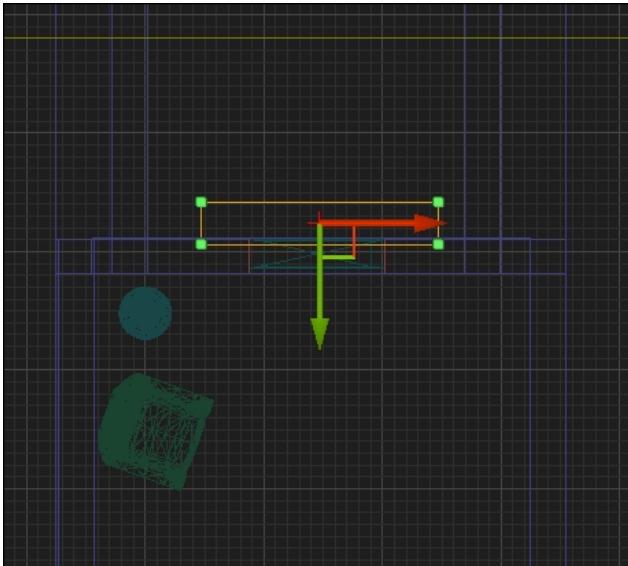
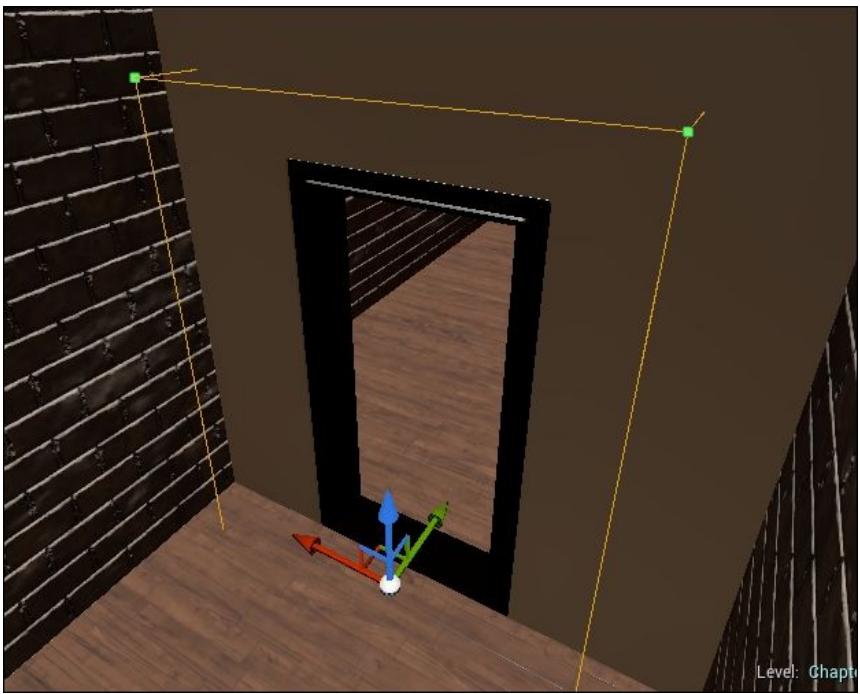
15. Next, go to **Light Color** and adjust the color of the light to #FF9084FF , to adjust the mood of the level.



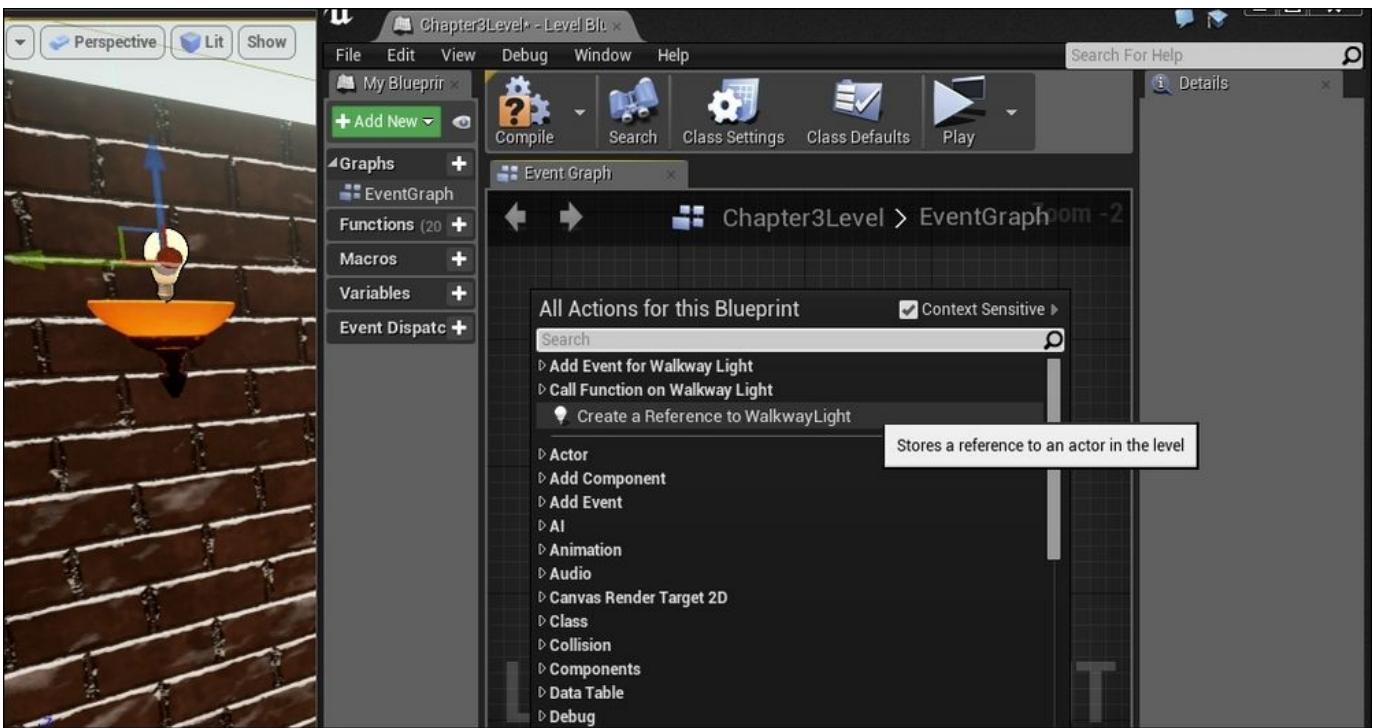
16. Now, let us rename the Point Light to **WalkwayLight** and the **Wall Lamp prop** to **WallLamp**.
17. Select the Point Light and right-click to display the contextual menu. Go to **Attach To** and select **WallLamp**. This attaches the light to the prop so that when we move the prop, the light moves together. The following screenshot shows that **WalkwayLight** is linked to **WallLamp**:



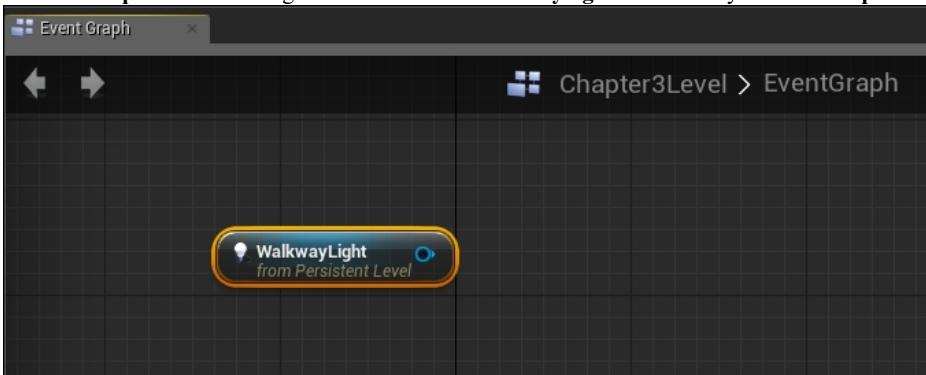
18. Now, let us create a Trigger Volume. Go to **Modes | Volumes**. Click and drag the Trigger Volume into the level.
19. Resize the volume to cover the entrance of the door dividing the two rooms. Refer to the next two screenshots on how to position the volume (**Perspective** view and **Top** view). Make sure that the volume covers the entire space of the door.



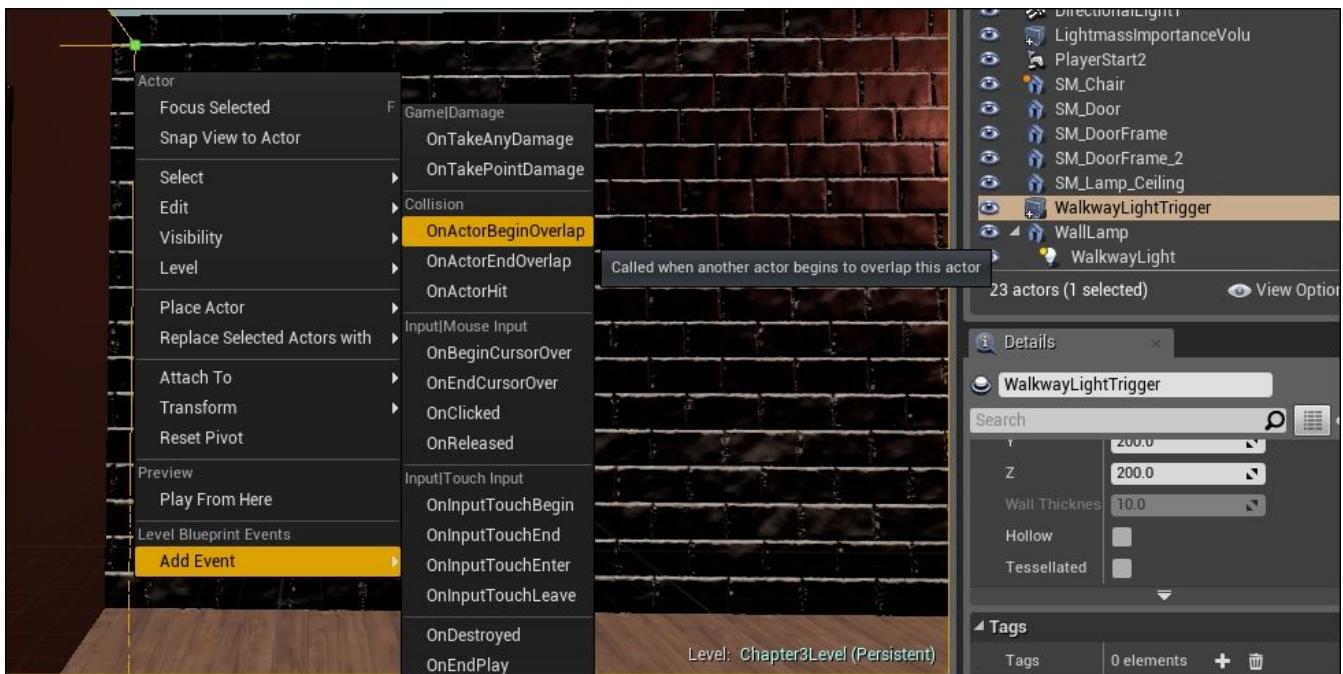
20. Rename **Trigger Volume** to `WalkwayLightTrigger`.
21. In order to use the Trigger Volume to turn the light on and off, we need to figure out which property from the Point Light controls this feature. Click on the Point Light (`WalkwayLight`) to display the properties of the light. Scroll down to **Rendering** and uncheck the property box for **Visible**. Notice that the light is now turned off. We want to keep the light turned off until we trigger it.
22. So, the next step is to link the sequence of events up. This is done via **Level Blueprint**. We will need to trigger this change in property using the Trigger Volume, which we have created and turn the light back on.
23. With the Point Light still selected, go to the top ribbon and select **Blueprints | Open Level Blueprint**. This opens up the **Level Blueprint** window. Make sure that the Point Light (`WalkwayLight`) is still selected as shown in the following screenshot:



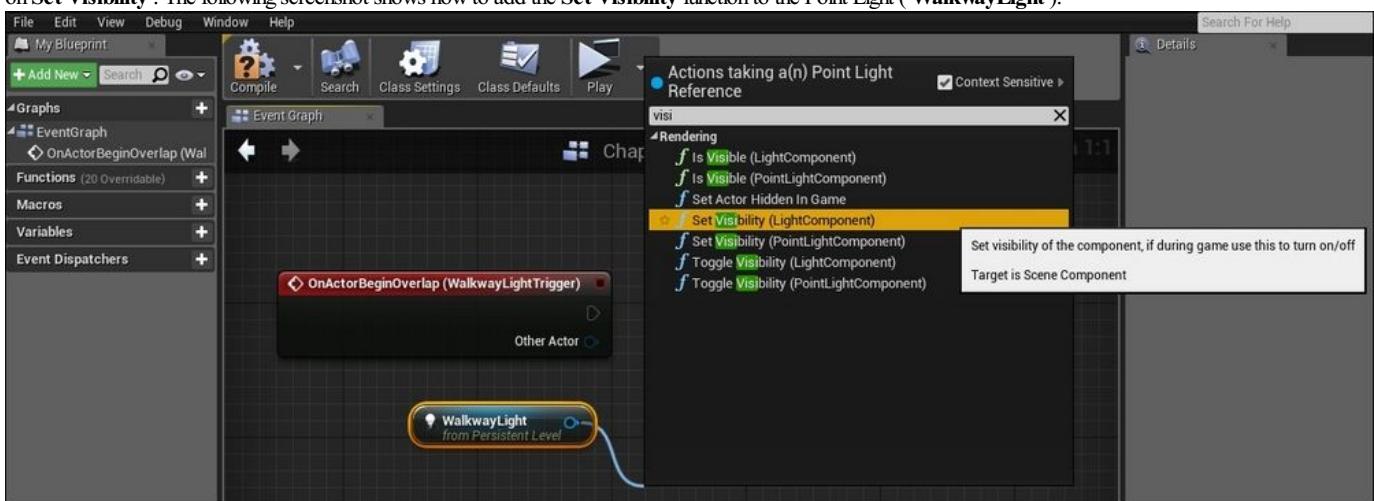
24. Right-click in the **Event Graph** of the **Level Blueprint** window to display what actions can be added to the **Level Blueprint**.
25. Due to Level Blueprint's ability to guide what actions are possible, we can simply select **Add Reference to WalkwayLight**. This creates the **WalkwayLight** actor in **Level Blueprint**. The following screenshot shows the **WalkwayLight** actor correctly added in **Blueprint**:



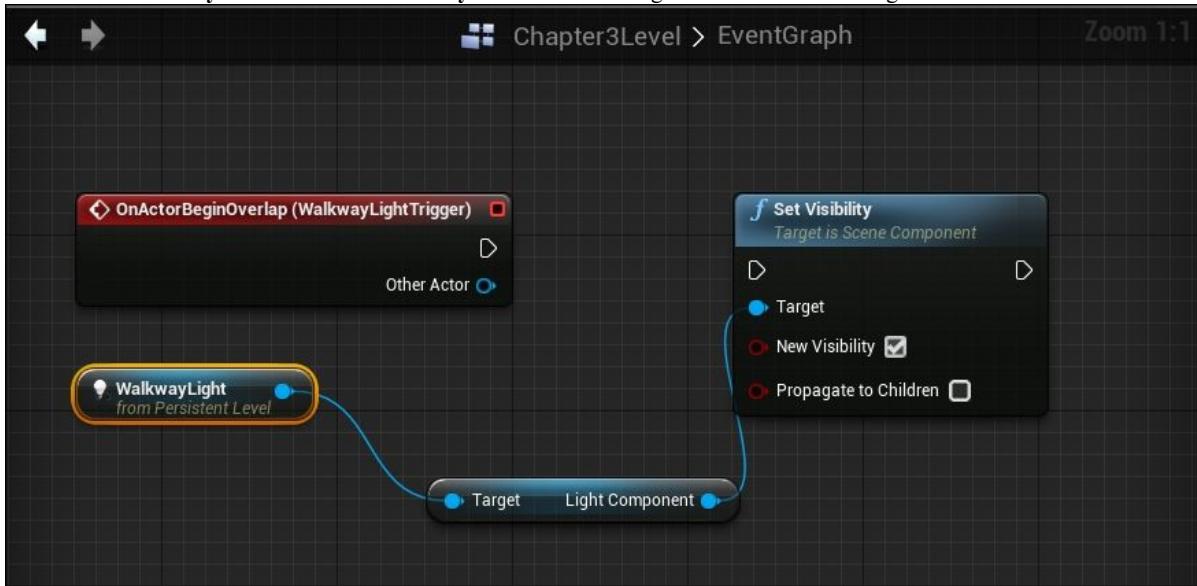
26. You can keep the **Level Blueprint** window open, and go to the Trigger Volume we have created the in the level.
27. Select the Trigger Volume (**WalkwayLightTrigger**), right-click and select **Add Event** and then **OnActorBeginOverlap**. The following screenshot shows how to add **OnActorBeginOverlap** in **Level Blueprint**:



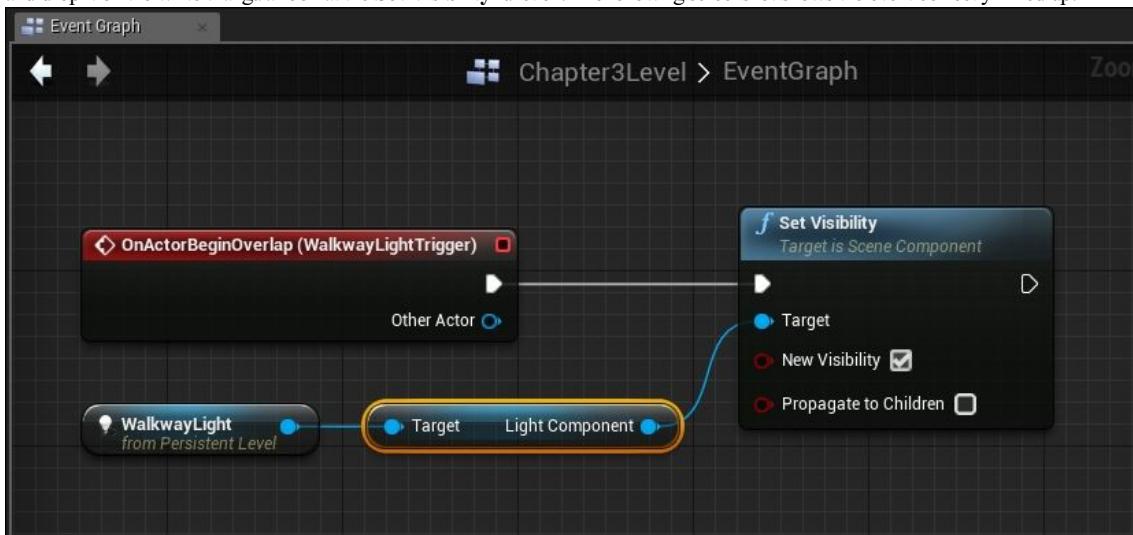
28. To control a variable in the Point Light, we will click and drag on the tiny blue circle on the **WalkwayLight** node added. This creates a blue line originating from the tiny blue circle. This also opens up a menu, where we can see what action can be done to the Point Light. Enter **visi** into the search bar to display the options. Click on **Set Visibility**. The following screenshot shows how to add the **Set Visibility** function to the Point Light (**WalkwayLight**):



29. Check the **New Visibility** checkbox in the **Set Visibility** function. The following screenshot shows the configuration we want:



30. Now, we are ready to link the **OnActorBeginOverlap** event to the **Set Visibility** function. Click and drag the white triangular box from **OnActorBeginOverlap** and drop it on the white triangular box at the **Set Visibility** function. The following screenshot shows the event correctly linked up:



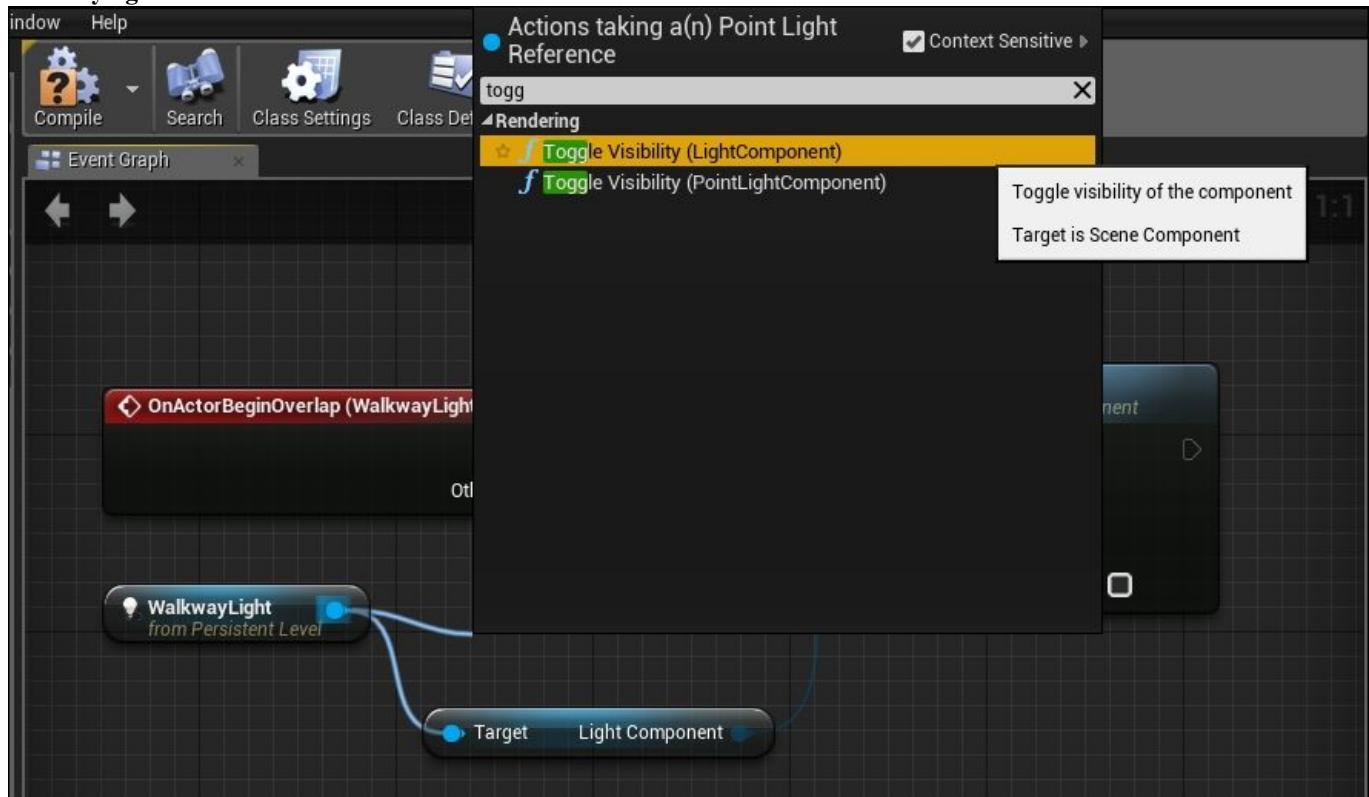
31. Now, build the level and play. Walk through the door from the first room to the second room. The light should be triggered on.

But what happens when you walk back into the first room? The light remained turned on and nothing happens when you walk back into the second room. In the next example, we will go through how you can toggle the light on and off as you walk in and out the room. It is an alternative way to implement the control of the light and I shall leave it as optional for you to try it out.

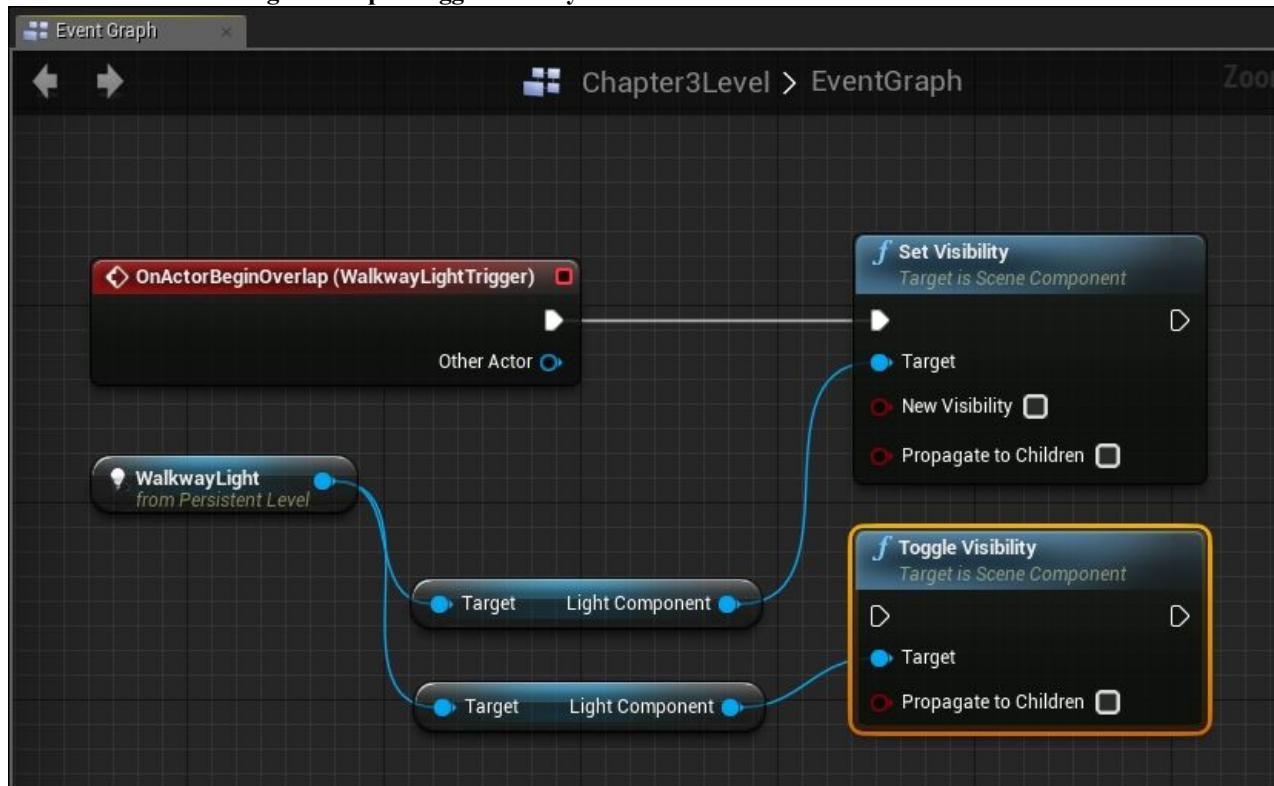
# Using Trigger Volume to toggle light on/off (optional)

The following steps can be used to trigger volume to toggle lights on or off

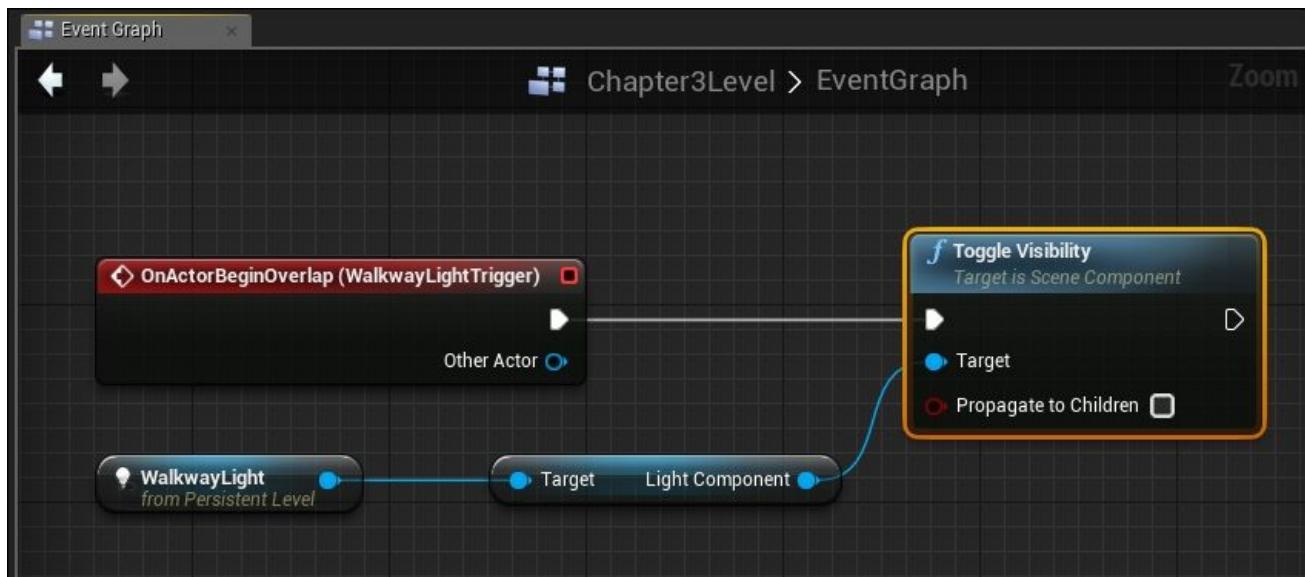
1. We need to replace the **Set Visibility** node in **Event Graph**. Click and drag the blue dot from Point Light (**WalkwayLight**) and drop it onto any blank space. This opens up the contextual menu. The following screenshot shows the contextual menu to place a new node from **WalkwayLight**:



2. Select **Toggle Visibility**. This creates an additional new node in **Event Graph**; we will need to rewire the links as per the following screenshot in order to link **OnActorBeginOverlap** to **Toggle Visibility**:



3. The last step is to delete the **Set Visibility** node and we are ready to toggle the light on and off as we move in and out of the room. The following screenshot shows the final **Event Graph** we want. Compile and play the level to see how you can toggle the light on and off.



# Summary

We have covered a number of very important concepts about the objects that we use to populate our game world in Unreal Engine 4. We have broken one of the most common types of game object, Static Mesh, into its most fundamental components in order to understand its construction. We have also compared two types of game objects (Static Meshes and BSP), how they are different, and why they have their spot in the game. This will help you decide what kind of objects need to be created and how they will be created for your game level.

The chapter also briefly introduced textures and materials, how they are created, and applied onto the meshes. We will go into more details about Materials in the next chapter. So you might want to read [Chapter 4 , Material and Light](#) , first before creating/applying materials to your newly created game objects. To help you optimize your game, this chapter also covered the mesh creation pipeline and the concept of LOD. For interactions to take place, we also needed to learn how objects interact and collide with one another in Unreal, what object properties are configurable to allow different physics interaction.

This chapter also covered our first introduction to Blueprint, the graphical scripting of Unreal Engine4. Through a simple Blueprint example, we learned how to turn on and off lights for our level using one of the many useful volumes that are in Unreal, Trigger Volume. In the next chapter, we will continue to build on the level we have created with more exciting materials and lights.

# Chapter 4. Material and Light

In this chapter, we will learn in detail about the materials and the lights in Unreal Engine 4. We have grouped both Material and Light together in this chapter because how an object looks is largely determined by both—material and lighting.

Material is what we apply to the surface of an object and it affects how the object looks in the game. Material/Shader programming is a hot ongoing research topic as we always strive to improve the texture performance—seeking higher graphic details/realism/quality with limited CPU/GPU rendering power. Researchers in this area need to find ways to make the models we have in a game look as real as possible, with as little calculations/data size as possible.

Lighting is also a very powerful tool in world creation. There are many uses of light. Lights can create a mood for the level. When effectively used, it can be used to focus attention on objects in the level and guide players through your level. Light also creates shadow. In a game level, shadow needs to be created artificially. Hence, we will also learn how we get shadows rendered appropriately for our game.

## Materials

In the previous chapter, we briefly touched on what a material is and what a texture is. A texture is like a simple image file in the format of `.png` / `.tga` . A material is a combination of different elements, including textures to create a surface property that we apply to our objects in the game. We have also briefly covered what UV coordinates are and how we use them to apply a 2D texture to the surface of a 3D object.

So far, we have only learned how to apply materials that are available in the default Unreal Engine. In this chapter, we will dive deeper into how we can actually create our own custom material in Unreal Engine 4. Fundamentally, the material creation for the objects falls into the scope of an artist. For special customized textures, they are sometimes hand painted by 2D artists using tools such as Photoshop or taken from photographs of textures from the exact objects we want, or similar objects. Textures can also be tweaked from existing texture collection to create the customized material that is needed for the 3D models. Due to the vast number of realistic textures needed, textures are sometimes also generated algorithmically by the programmers to allow more control over its final look. This is also an important research area for the advancing materials for computer graphics.

Material manipulation here falls under the scope of a specialized group of programmers known as **graphic programmers** . They are sometimes also researchers that look into ways to better compress texture, improve rendering performance, and create special dynamic material manipulation.

# The Material Editor

In Unreal Engine 4, material manipulation can be achieved using the Material Editor. What this editor offers is the ability to create material expressions. Material expressions work together to create an overall surface property for the material. You can think of them as mathematical formulas that add/multiply together to affect the properties of a material. The Material Editor makes it easy to edit/formulate material expressions to create customized material and provides the capability to quickly preview the changes in the game. Through Unreal's Blueprint capabilities and programming, we can also achieve dynamic manipulation of materials as needed by the game.

## The rendering system

The rendering system in Unreal Engine 4 uses the DirectX 11 pipeline, which includes deferred shading, global illumination, lit translucency, and post processing. Unreal Engine 4 has also started branching to work with the newest DirectX 12 pipeline for Windows 10, and DirectX 12 capabilities will be available to all.

## Physical Based Shading Model

Unreal Engine 4 uses the **Physical Based Shading Model ( PBSP )**. This is a concept used in many modern day game engines. It uses an approximation of what light does in order to give an object its properties. Using this concept, we give values (0 to 1) to these four properties: **Base Color**, **Roughness**, **Metallic**, and **Specular** to approximate the visual properties.

For example, the bark of a tree trunk is normally brown, rough, and not very reflective. Based on what we know about how the bark should look like, we would probably set the metallic value to low value, roughness to a high value, and the base color to display brown with a low specular value.

This improves the process of creating materials as it is more intuitive as visual properties are governed by how light reacts, instead of the old method where we approximate the visual properties based on how light should behave.

For those who are familiar with the old terms used to describe material properties, you can think of it as having **Diffuse Color** and **Specular Power** replaced by **Base Color**, **Metallic**, and **Roughness**.

The advantage of using PBSP is that we can better approximate material properties with more accuracy.

## High Level Shading Language

The Material Editor enables visual scripting of the **High Level Shading Language ( HLSL )**, using a network of nodes and connection. Those who are completely new to the concept of shaders or HLSL should go on to read the next section about shaders, DirectX and HLSL first, so that you have the basic foundation on how the computer renders material information on the screen. HLSL is a proprietary shading language developed by Microsoft. OpenGL has its own version, known as GLSL. HLSL is the programming language used to program the stages in the graphics pipeline. It uses variables that are similar to C programming and has many intrinsic functions that are already written and available for use by simply calling the function. HLSL shaders can be compiled at author-time or at runtime, and set at runtime into the appropriate pipeline stage.

## Getting started

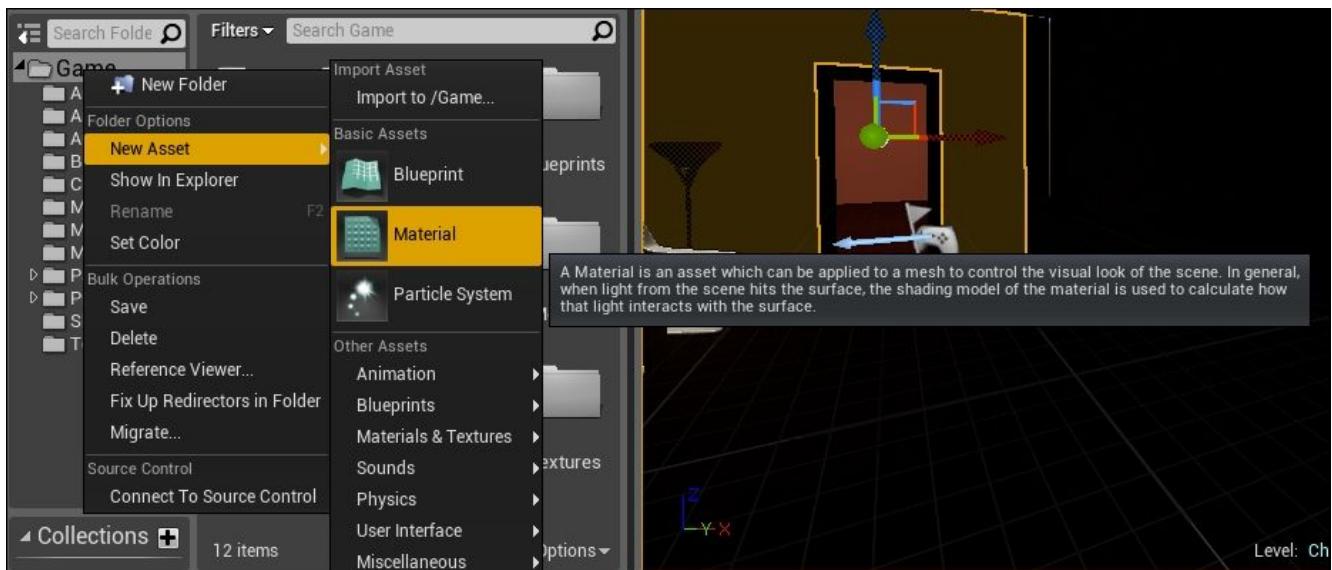
To open the Material Editor in Unreal Engine 4, go to **Content Browser | Material** and double-click on any material asset. Alternatively, you can select a material asset, right-click to open the context menu and select **Edit** to view that asset in the Material Editor.

If you want to learn how to create a new material, you can try out the example, which is covered in the upcoming section.

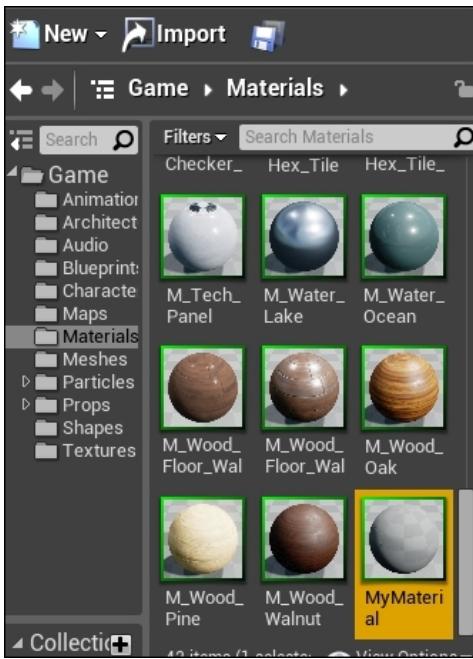
## Creating a simple custom material

We will continue to use the levels we have created. Open `Chapter3Level.umap` and rename it `Chapter4Level.umap` to prevent overwriting what we have completed at the end of the previous chapter.

To create a new Material asset in our game package, go to **Content Browser | Material**. With **Material** selected, right-click to open the contextual menu, navigate to **New Asset | Material**. This creates the new material in the **Material** folder (we want to place assets in logical folders so that we can find game assets easily). Alternatively, you can go to **Content Browser | New | Material**.

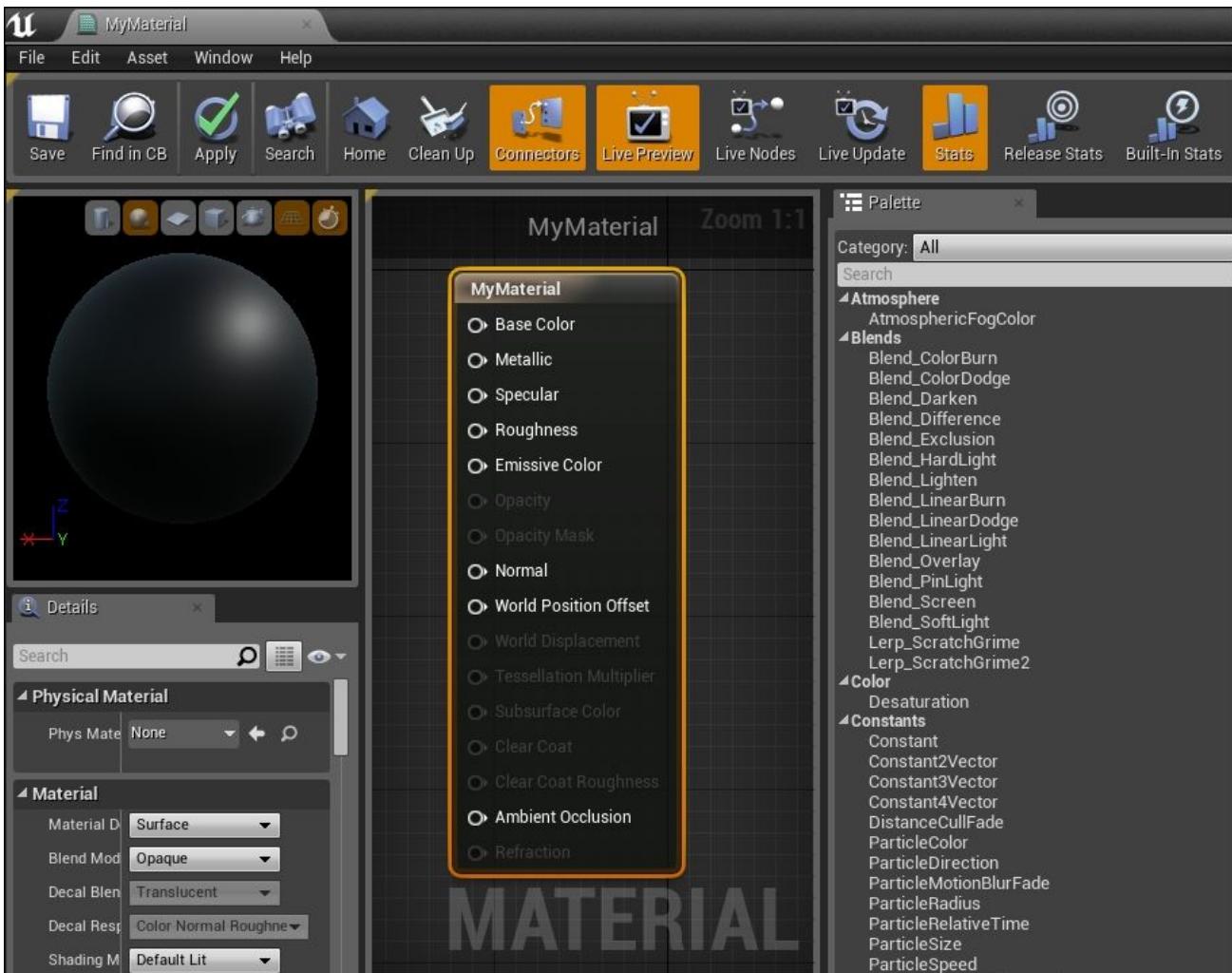


Rename the new material to **MyMaterial**. The following screenshot shows the new **MyMaterial** correctly created:



Note that the thumbnail display for the new **MyMaterial** shows a grayed-out checkered material. This is the default material when no material has been applied.

To open the Material Editor to start designing our material, double-click on **MyMaterial**. The following screenshot shows the Material Editor with a blank new material. The spherical preview of the material shows up as black since no properties have been defined yet.

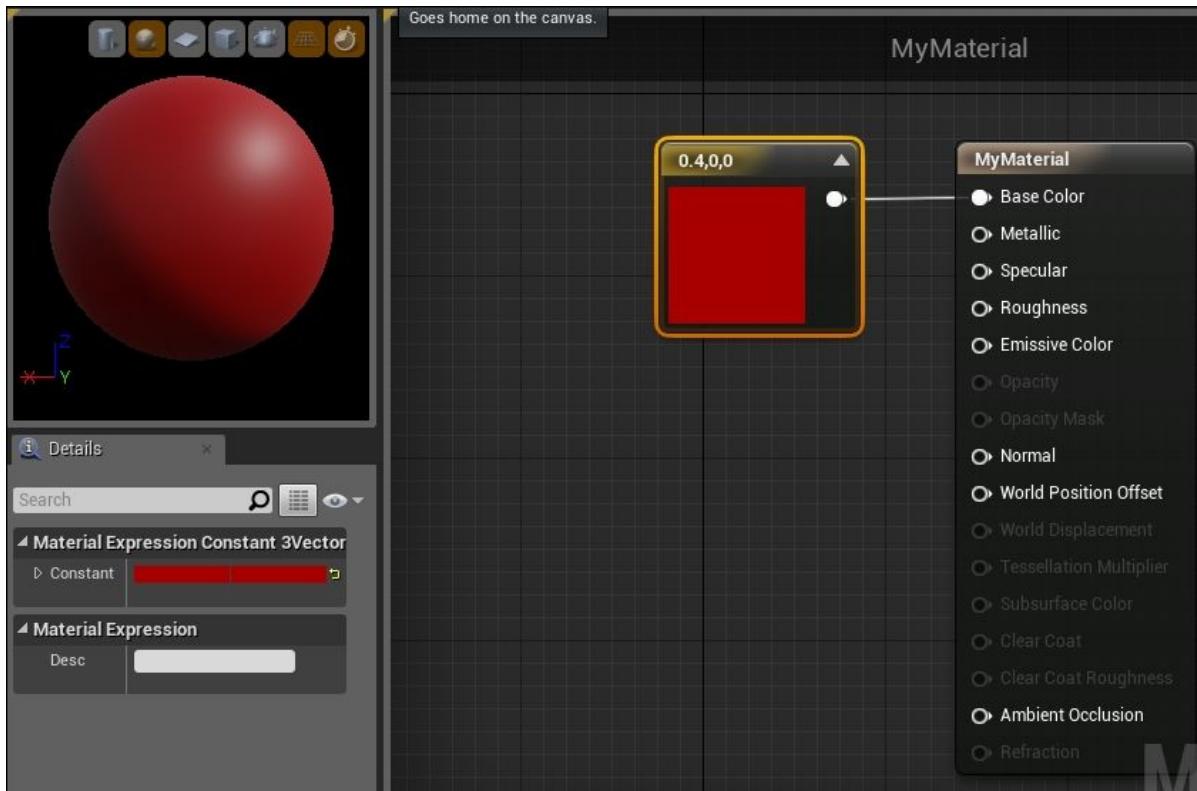


Let's start to define some properties for the **MyMaterial** node to create our very own unique material. **Base Color**, **Metallic**, and **Roughness** are the three values we will learn to configure first.

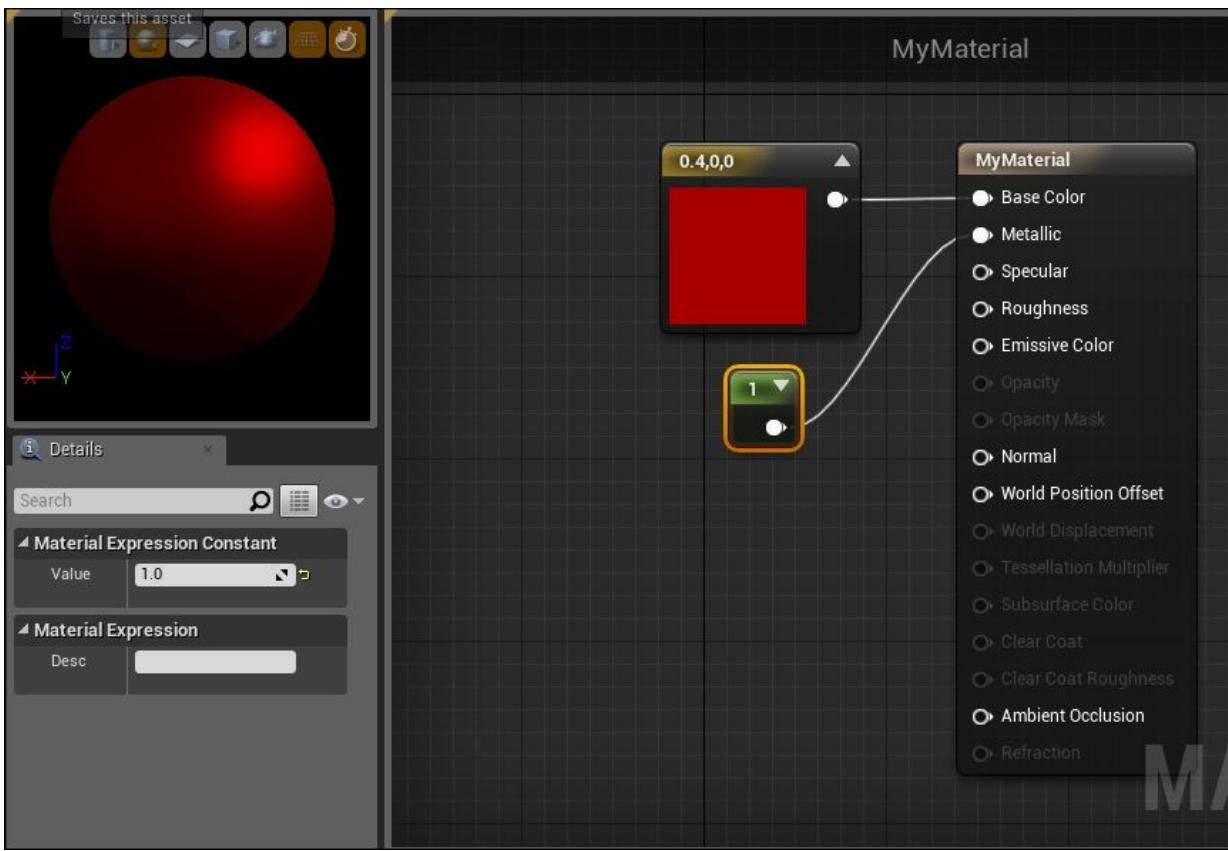
**Base Color** is defined by the red, green, and blue values in the form of a vector. To do so, we will drag and drop **Constant3Vector** from **MyPalette** on the right-hand side into the main window where the **MyMaterial** node is in. Alternatively, you can right-click to open the context menu and type **vector** into the search box to filter the list. Click and select **Constant3Vector** to create the node. Double-click on the **Constant3Vector** to display the **Color Picker** window. The following screenshot shows the setting of **Constant3Vector** we want to use to create a material for a red wall. (**R = 0.4**, **G = 0.0**, **B = 0.0**, **H = 0.0**, **S = 1.0**, **V = 0.4**):



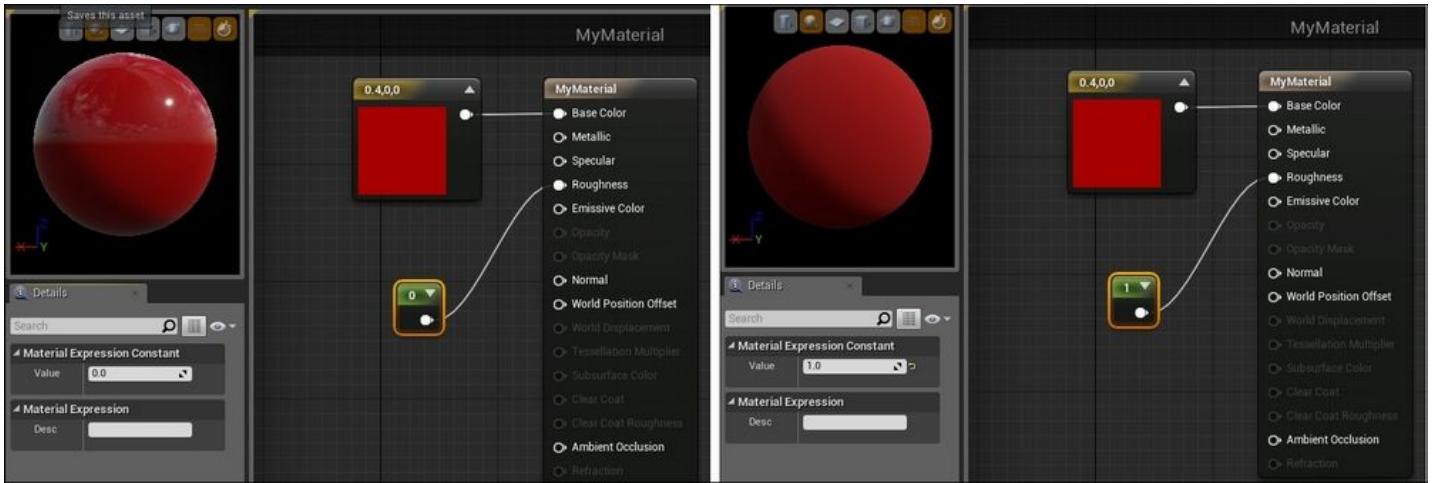
Connect the **Constant3Vector** to the **MyMaterial** node as shown in the following screenshot by clicking and dragging from the small circle from the **Constant3Vector** node to the small circle next to the **Base Color** property in the **MyMaterial** node. This **Constant3Vector** node now provides the base color to the material. Notice how the spherical preview on the left updates to show the new color. If the color is not updated automatically, make sure that the **Live Preview** setting on the top ribbon is selected.



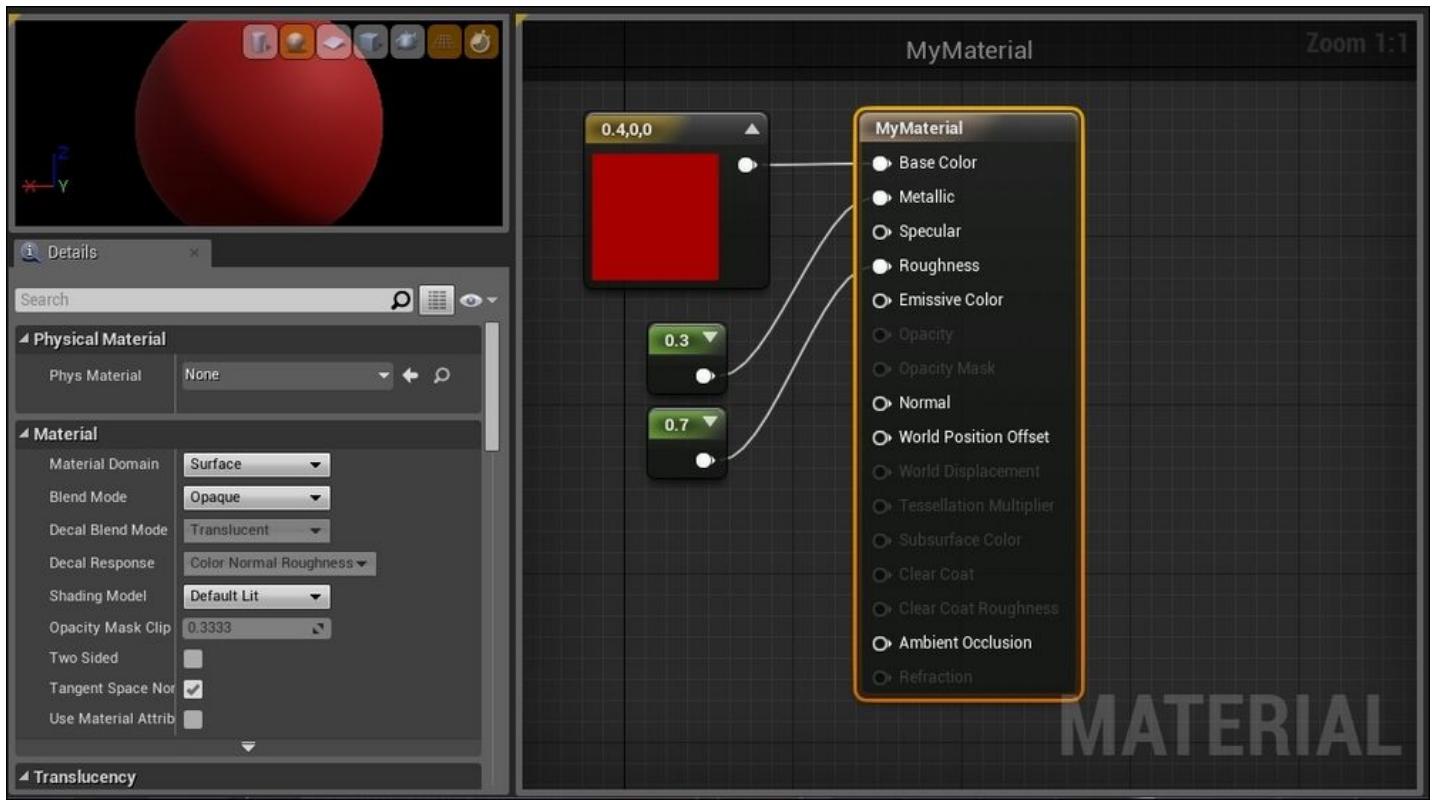
Now, let us set the **Metallic** value for the material. This property takes a numerical value from 0 to 1, where 1 is for a 100% metal. To create an input for a value, click and drag **Constant** from **MyPalette** or right-click in the Material Editor to open the menu; type in **Constant** into the search box to filter and select **Constant** from the filtered list. To edit the value in the constant, click on the **Constant** node to display the **Details** window and fill in the value. The following screenshot shows how the material would look if **Metallic** is set to 1:



After seeing how the **Metallic** value affects the material, let us see what **Roughness** does. **Roughness** also takes a **Constant** value from 0 to 1, where 0 is completely smooth and makes the surface very reflective. The left-hand screenshot shows how the material looks when **Roughness** is set to 0, whereas the right-hand screenshot shows how the material will look when **Roughness** is set to 1:



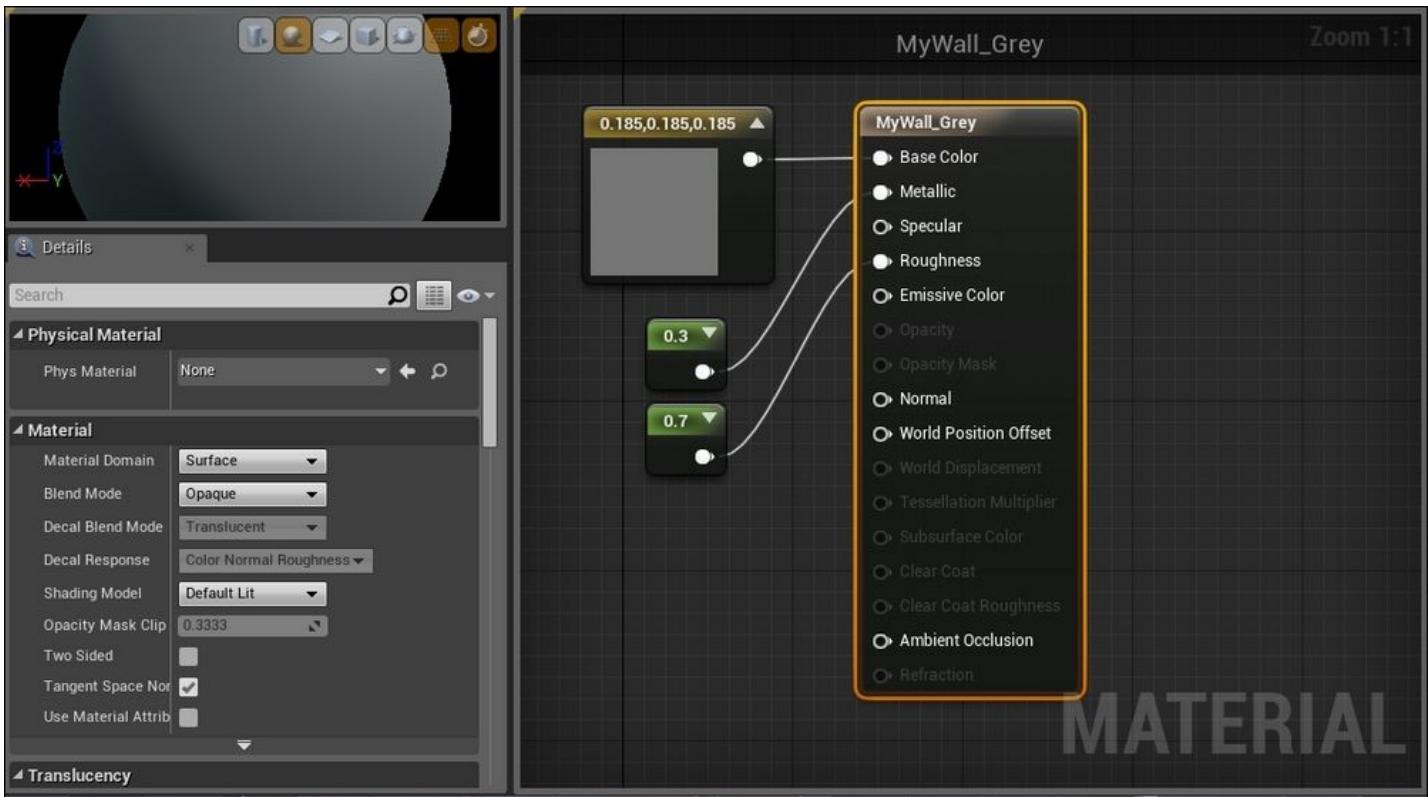
We want to use this new material to texture the walls. So, we have set **Metallic** as **0.3** and **Roughness** as **0.7**. The following screenshot shows the final settings we have for our first custom material:



Go to **MyMaterial** in **Content Browser** and duplicate **MyMaterial**. Rename it **MyWall\_Grey**. Change the base color to gray using the following values as shown in the picker node for the **Constant3Vector** value for **Base Color**. ( **R = 0.185** , **G = 0.185** , **B = 0.185** , **H = 0.0** , **S = 0.0** , **V = 0.185** ):



The following screenshot shows the links for the **MyWall\_Grey** node. ( **Metallic = 0.3** , **Roughness = 0.7** ):



## Creating custom material using simple textures

To create a material using textures, we must first select a texture that is suitable. Textures can be created by artists or taken from photos of materials. For learning purposes, you can find suitable free source images from the Web, such as [www.textures.com](http://www.textures.com), and use them. Remember to check for conditions of usage and other license-related clauses, if you plan to publish it in a game.

There are two types of textures we need for a custom material using a simple texture. First, the actual texture that we want to use. For now, let us keep this selection simple and straightforward. Select this texture based on the color and it should have the overall properties of what you want the material to look like. Next, we need a normal texture. If you still remember what a normal map is, it controls the bumps on a surface. The normal map gives the grooves in a material. Both of these textures will work together to give you a realistic-looking material that you can use in your game.

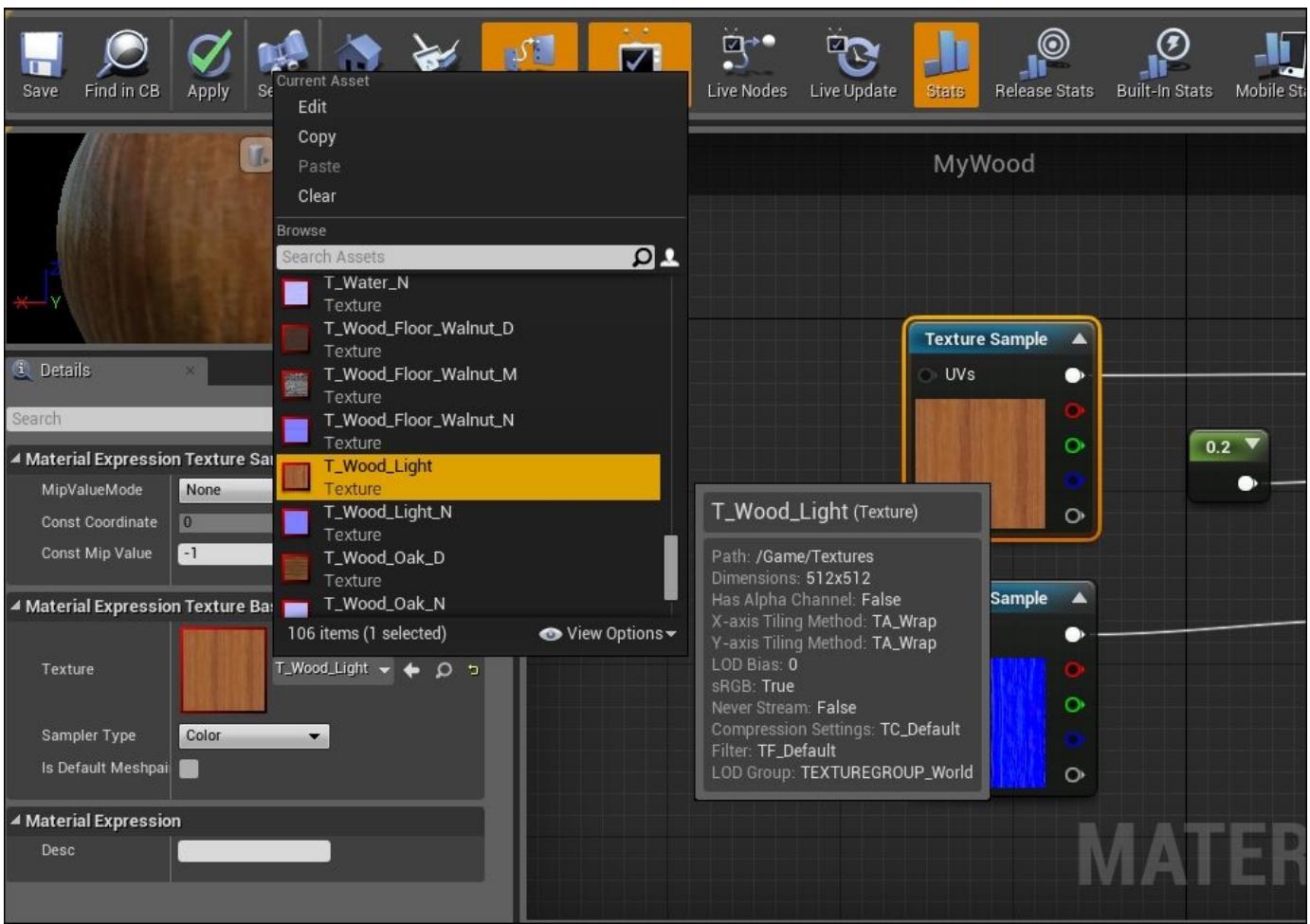
In this example, we will create another wood texture that we will use to replace the wood texture from the default package that we have already applied in the room.

Here, we will start first by importing the textures that we need in Unreal Engine. Go to **Content Browser | Textures**. Then click on the **Import** button at the top. This opens up a window to browse to the location of your texture. Navigate to the folder location where your texture is saved, select the texture and click on **Open**. Note that if you are importing textures that are not in the power of two (256 x 256, 1024 x 1024, and so on), you would have a warning message. Textures that are not in the power of two should be avoided due to poor memory usage. If you are importing the example images that I am using, they are already converted to the power of two so you would not get this warning message on screen.

Import both **T\_Wood\_Light** and **T\_Wood\_Light\_N**. **T\_Wood\_Light** will be used as the main texture, we want to have, and **T\_Wood\_Light\_N** is the normal map texture, which we will use for this wood.

Next, we follow the same steps to create a new material, as in the previous example. Go to **Content Browser | Material**. With the **Material** folder selected, to open the contextual menu, navigate to **New Asset | Material**. Rename the new material **MyWood**.

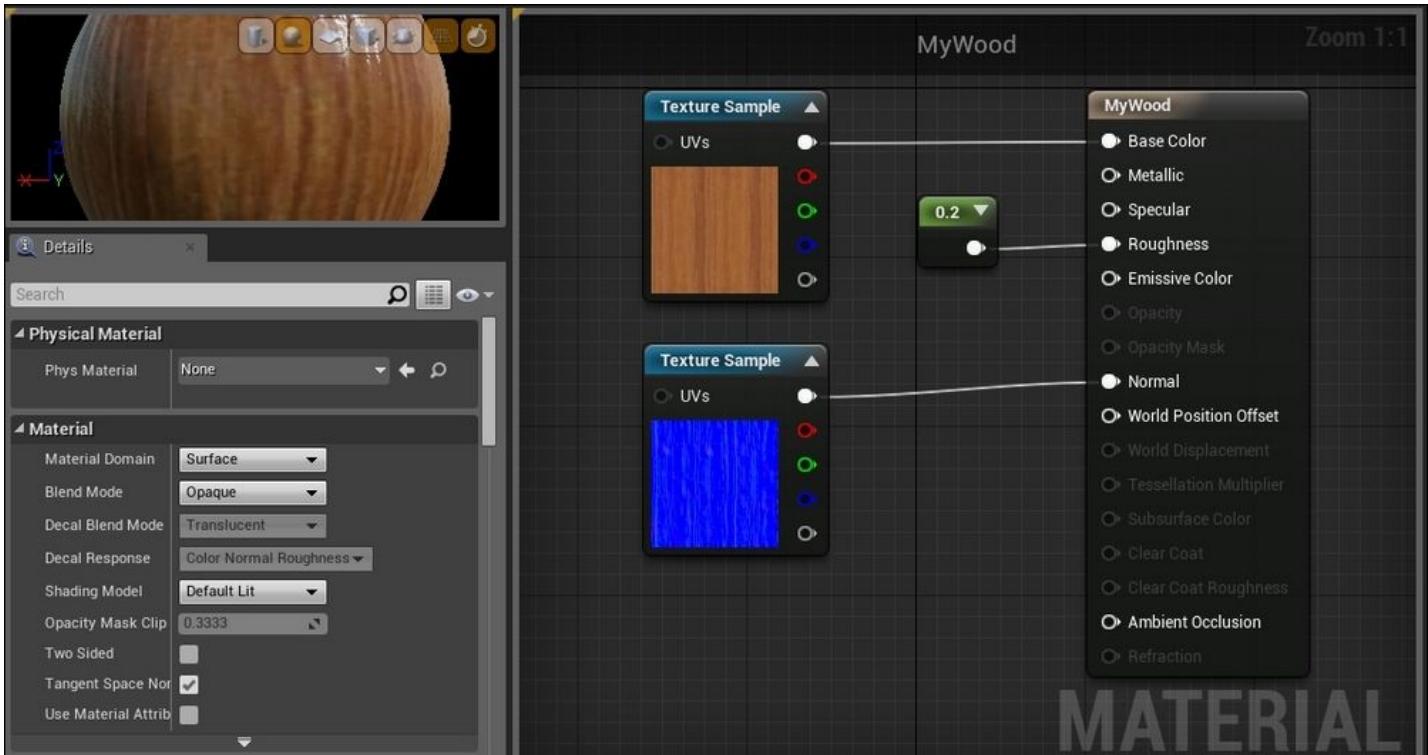
Now, instead of selecting **Constant3Vector** to provide values to the base color, we will use **TextureSample**. Go to **MyPalette** and type in **Texture** to filter the list. Select **TextureSample**, drag and drop it into the Material Editor. Click on the **TextureSample** node to display the **Details** panel, as shown in the following screenshot. On the **Details** panel, go to **Material Expression Texture Base** and click on the small arrow next to it. This opens up a popup with all the suitable assets that you can use. Scroll down to select **T\_Wood\_Light**.



Now, we have configured **TextureSample** with the wood texture that we have imported into the editor earlier. Connect **TextureSample** by clicking on the white hollow circle connector, dragging it and dropping it on the **Base Color** connector on the **MyWood** node.

Repeat the same steps to create a **TextureSample** node for the **T\_Wood\_Light\_N** normal map texture and connect it to the **Normal** input for **MyWood**.

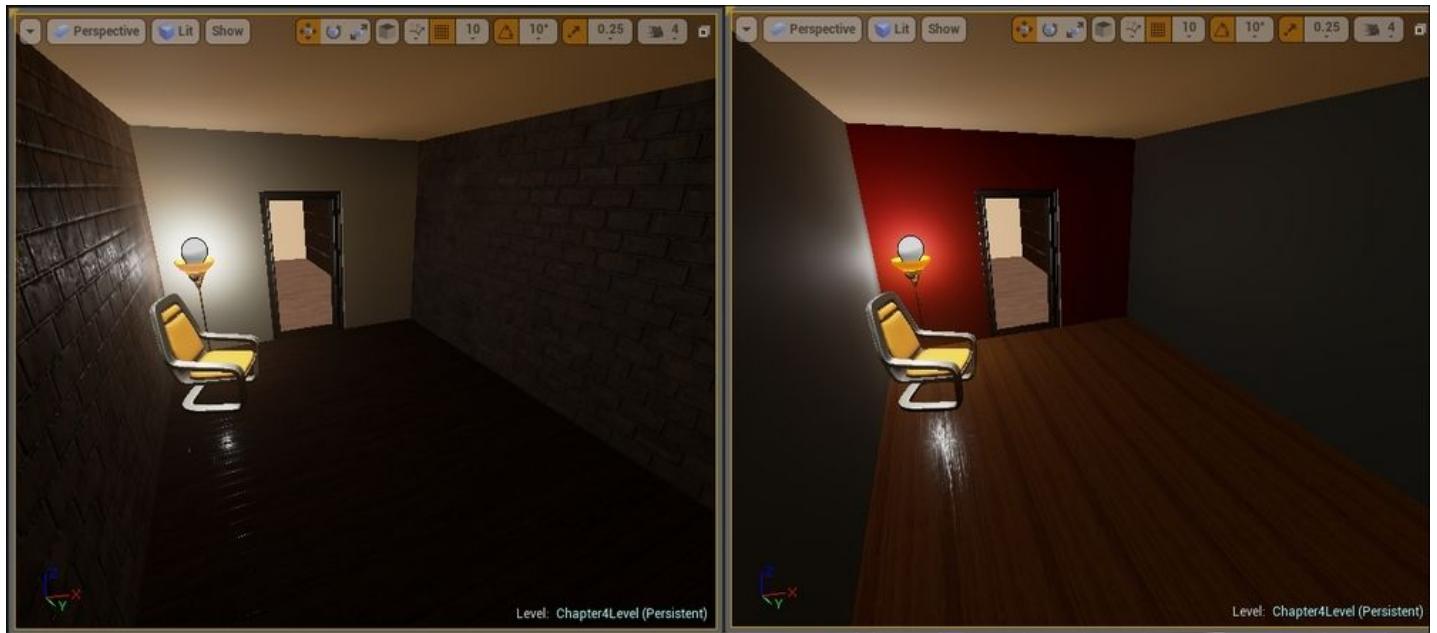
The following screenshot shows the settings that we want to have for **MyWood**. To have a little glossy feel for our wood texture, set **Roughness** to **0.2** by using a **Constant** node. (Recap: drag and drop a **Constant** node from **MyPalette** and set the value to **0.2**, connect it to the **Roughness** input of **MyWood**.)



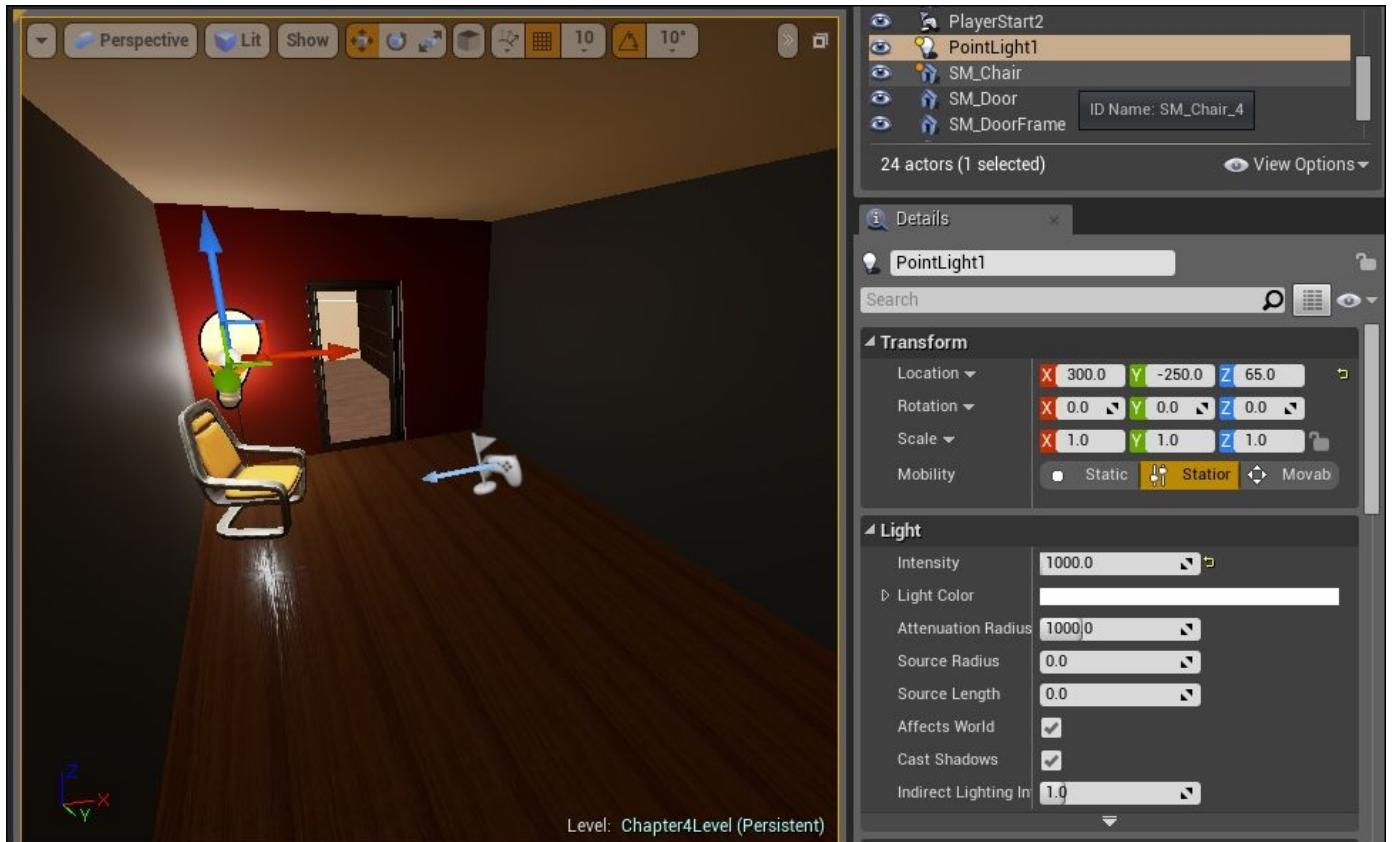
## Using custom materials to transform the level

Using the custom materials that we have created in the previous two examples, we will replace the current materials that we have used.

The following screenshot shows the before and after look of the first room. Notice how the new custom materials have transformed the room into a modern looking room.



From the preceding screenshot, we also have added a Point Light and placed it onto the lamp prop, making it seem to be emitting light. The following screenshot shows the Point Light setting we have used ( **Light Intensity = 1000.0** , **Attenuation Radius = 1000.0** ):



Next, we added a ceiling to cover up the room. The ceiling of the wall uses the same box geometry as the rest of the walls. We have applied the **M\_Basic\_Wall** material onto it.

Then, we use the red wall material ( **MyMaterial** ) to replace the material on wall with the door frame. The gray wall material ( **MyWall\_Grey** ) is used to replace the brick material for the walls at the side. The glossy wood material ( **MyWood** ) is used to replace the wooden floor material.

# Rendering pipeline

For an image to appear on the screen, the computer must draw the images on the screen to display it. The sequence of steps to create a 2D representation of a scene by using both 2D and 3D data information is known as the graphics or rendering pipeline. Computer hardware such as **central processing unit ( CPU )** and **graphics processing unit ( GPU )** are used to calculate and manipulate the input data needed for drawing the 3D scene.

As games are interactive and rely heavily on real-time rendering, the amount of data necessary for rendering moving scenes is huge. Coordinate position, color, and all display information needs to be calculated for each vertex of the triangle polygon and at the same time, taking into account the effect of overlapping polygons before they can be displayed on screen correctly. Hence, it is very crucial to optimize both the CPU and GPU capabilities to process this data and deliver them timely on the screen. Continuous improvement in this area has been made over the years to allow better quality images to be rendered at higher frame rates for a better visual effect. At this point, games should run at a minimum frame rate of 30fps in order for players to have a reasonable gaming experience.

The rendering pipeline today uses a series of programmable shaders to manipulate information about an image before displaying the image on the screen. We'll cover shaders and Direct3D 11 graphics pipeline in more detail in the upcoming section.

# Shaders

Shaders can be thought of as a sequence of programming codes that tells a computer how an image should be drawn. Different shaders govern different properties of an image. For example, Vertex Shaders give properties such as position, color, and UV coordinates for individual vertices. Another important purpose of vertex shaders is to transform vertices with 3D coordinates into the 2D screen space for display. Pixel shaders processes pixels to provide color, z-depth, and alpha value information. Geometry shader is responsible for processing data at the level of a primitive (triangle, line, and vertex).

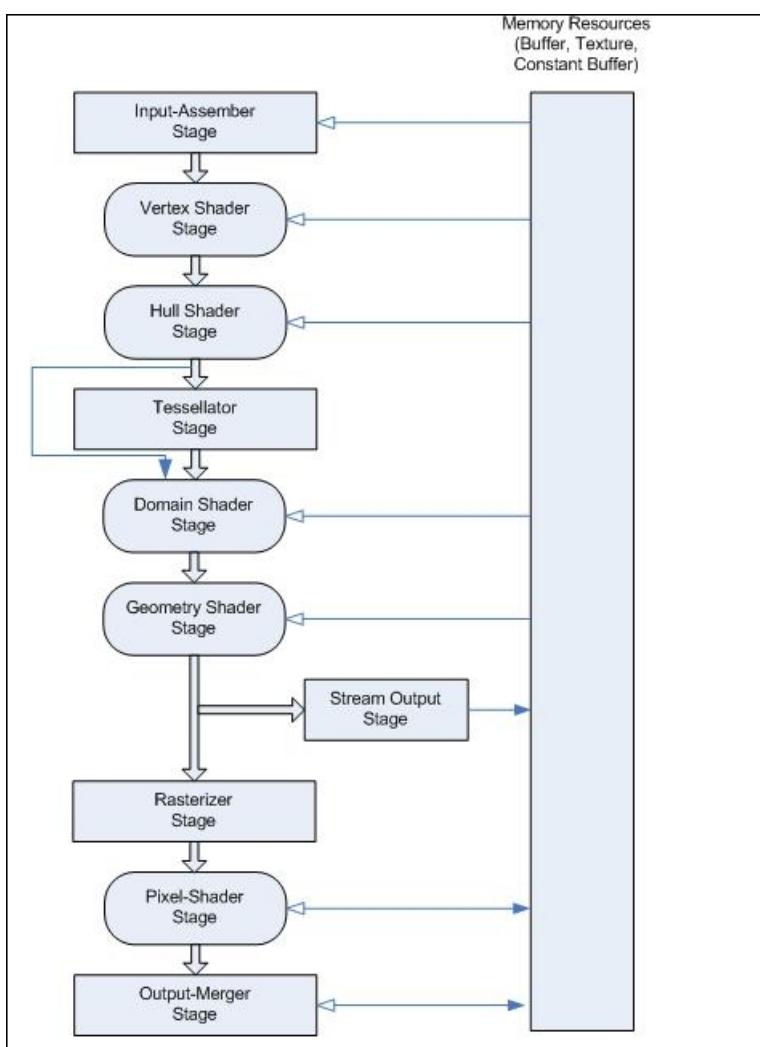
Data information from an image is passed from one shader to the next for processing before they are finally output through a frame buffer.

Shaders are also used to incorporate post-processing effects such as Volumetric Lighting, HDR, and Bloom effects to accentuate images in a game.

The language which shaders are programmed in depends on the target environment. For Direct3D, the official language is HLSL. For OpenGL, the official shading language is **OpenGL Shading Language ( GLSL )**.

Since most shaders are coded for a GPU, major GPU makers Nvidia and AMD have also tried developing their own languages that can output for both OpenGL and Direct3D shaders. Nvidia developed Cg (deprecated now after version 3.1 in 2012) and AMD developed Mantle (used in some games, such as *Battlefield 4*, that were released in 2014 and seems to be gaining popularity among developers). Apple has also recently released its own shading language known as **Metal Shading Language** for iOS 8 in September 2014 to increase the performance benefits for iOS. Kronos has also announced a next generation graphics API based on OpenGL known as **Vulkan** in early 2015, which appears to be strongly supported by member companies such as Valve Corporation.

The following image is taken from a Direct3D 11 graphics pipeline on MSDN ([http://msdn.microsoft.com/en-us/library/windows/desktop/ff476882\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ff476882(v=vs.85).aspx)). It shows the programmable stages, which data can flow through to generate real-time graphics for our game, known as the rendering pipeline state representation.

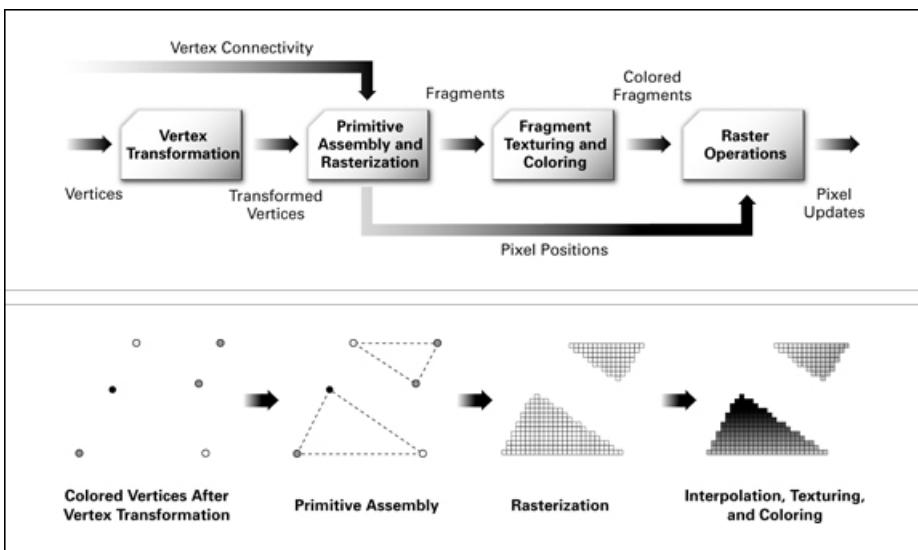


The information here is taken from Microsoft MSDN page. You can use the Direct3D 11 API to configure all of the stages. Stages such as vertex, hull, domain, geometry, and pixel-shader (those with the rounded rectangular blocks), are programmable using HLSL. The ability to configure this pipeline programmatically makes it flexible for the game graphics rendering.

What each stage does is explained as follows:

Stage	Function
Input-assembler	This stage supplies data (in the form of triangles, lines, and points) to the pipeline.
Vertex-shader	This stage processes vertices such as undergoing transformations, skinning, and lighting. The number of vertices does not change after undergoing this stage.
Geometry-shader	This stage processes entire geometry primitives such as triangles, lines, and a single vertex for a point.
Stream-output	This stage serves to stream primitive data from the pipeline to memory while on its way to the rasterizer.
Rasterizer	This clips primitives and prepare the primitives for the pixel-shader.
Pixel-shader	Pixel manipulation is done here. Each pixel in the primitive is processed here, for example, pixel color.
Output-merger	This stage combines the various output data (pixel-shader values, depth, and stencil information) with the contents of the render target and depth/stencil buffers to generate the final pipeline result.
Hull-shader, tessellator, and domain-shader	These tessellation stages convert higher-order surfaces to triangles to prepare for rendering.

To help you better visualize what happens in each of the stages, the following image shows a very good illustration of a simplified rendering pipeline for vertices only. The image is taken from an old Cg tutorial. Note that different APIs have different pipelines but rely on similar basic concepts in rendering (source: <http://goanna.cs.mit.edu/~gl/teaching/rtr&3dgp/notes/pipeline.html> ).



Example flow of how graphics is displayed:

- The CPU sends instructions (compiled shading language programs) and geometry data to the graphics processing unit, located on the graphics card.
- The data is passed through into the vertex shader where vertices are transformed.
- If the geometry shader is active in the GPU, the geometry changes are performed in the scene.
- If a tessellation shader is active in the GPU, the geometries in the scene can be subdivided. The calculated geometry is triangulated (subdivided into triangles).
- Triangles are broken down into fragments. Fragment quads are modified according to the fragment shader.
- To create the feel of depth, the z buffer value is set for the fragments and then sent to the frame buffer for displaying.

# APIs – DirectX and OpenGL

Both DirectX and OpenGL are collections of **application programming interfaces** ( APIs ) used for handling multimedia information in a computer. They are the two most common APIs used today for video cards.

DirectX is created by Microsoft to allow multimedia related hardware, such as GPU, to communicate with the Windows system. OpenGL is the open source version that can be used on many operating system including Mac OS.

The decision to use DirectX or OpenGL APIs to program is dependent on operating system of the target machine.

## DirectX

Unreal Engine 4 was first launched using DirectX11. Following the announcement that DirectX 12 ships with Windows 10, Unreal has created a DirectX 12 branch from the 4.4 version to allow developers to start creating games using this new DirectX 12.

An easy way to identify APIs that are a part of DirectX is that the names all begin with Direct. For computer games, the APIs that we are most concerned about are Direct3D, which is the graphical API for drawing high performance 3D graphics in games, and DirectSound3D, which is for the sound playback.

DirectX APIs are integral in creating high-performance 2D and 3D graphics for the Windows operating system. For example, DirectX11 is supported in Windows Vista, Windows 7 and Windows 8.1. The latest version of DirectX can be updated through service pack updates. DirectX 12 is known to be shipped with Windows 10.

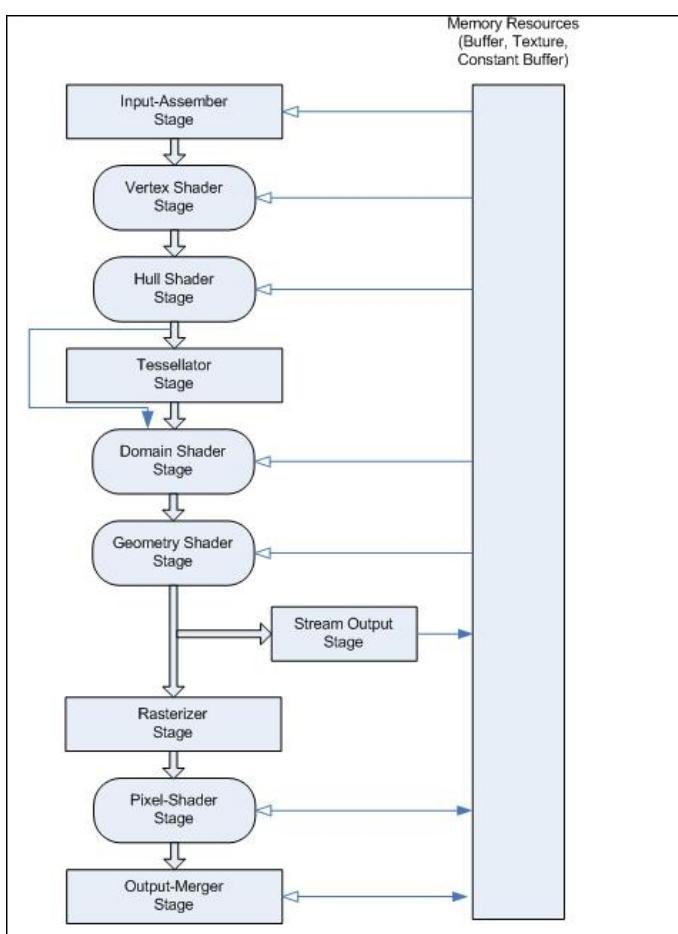
## DirectX12

Direct3D 12 was announced in 2014 and has been vastly revamped from Direct3D 11 to provide significant performance improvement. This is a very good link to a video posted on the MSDN blog that shows the tech demo for DirectX 12: <http://channel9.msdn.com/Blogs/DirectX-Developer-Blog/DirectX-Techdemo> .

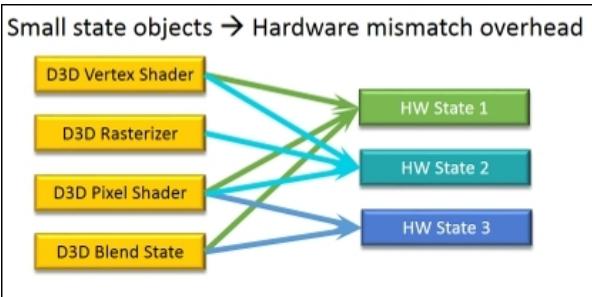
(If you are unfamiliar with Direct3D 11 and have not read the *Shaders* section earlier, read that section before proceeding with the rest of the DirectX section.)

### Pipeline state representation

If you can recall from the *Shaders* section, we have looked at the programmable pipeline for Direct3D 11. The following image is the same from the *Shaders* section (taken from MSDN) and it shows a series of programmable shaders:

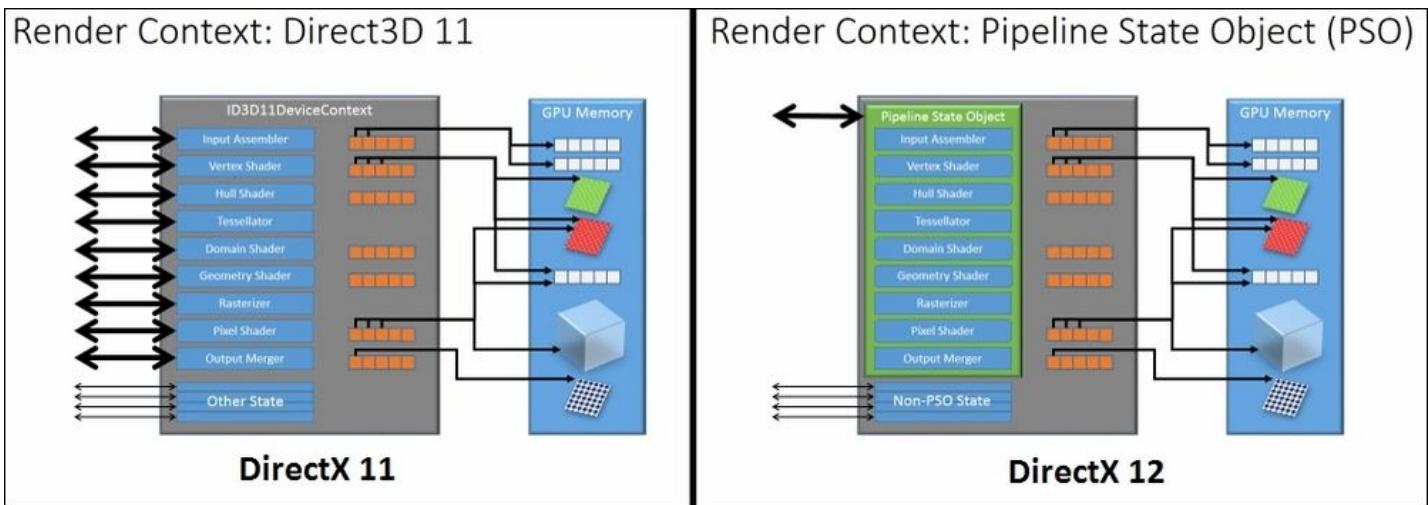


In Direct3D 11, each of the stages is configurable independently and each stage is setting states on the hardware independently. Since many stages have the capability to set the same hardware state due to interdependency, this results in hardware mismatch overhead. The following image is an excellent illustration of how hardware mismatch overhead happens:



The driver will normally record these states from the application (game) first and wait until the draw time, when it is ready to send it to the display monitor. At draw time, these states are then queried in a control loop before they are translated into a GPU code for the hardware in order to render the correct scene for the game. This creates an additional overhead to record and query for all the states at draw time.

In Direct3D 12, some programmable stages are grouped to form a single object known as **pipeline state object (PSO)** so that each hardware state is set only once by the entire group, preventing hardware mismatch overhead. These states can now be used directly, instead of having to spend resources computing the resulting hardware states before the draw call. This reduces the draw call overhead, allowing more draw calls per frame. The PSO that is in use can still be changed dynamically based on whatever hardware native instructions and states that are required.



## Work submission

In Direct3D 11, work submission to the GPU is immediate. What is new in Direct3D 12 is that it uses command lists and bundles that contain the entire information needed to execute a particular workload.

Immediate work submission in Direct3D 11 means that information is passed as a single stream of command to the GPU and due to the lack of the entire information, these commands are often deferred until the actual work can be done.

When work submission is grouped in the self-contained command list, the drivers can precompute all the necessary GPU commands and then send that list to the GPU, making Direct3D 12 work submission a more efficient process. Additionally, the use of bundles can be thought of as a small list of commands that are grouped to create a particular object. When this object needs to be duplicated on screen, this bundle of commands can be "played back" to create the duplicated object. This further reduces computational time needed in Direct3D 12.

## Resource access

In Direct3D 11, the game creates resource views that bind these views to slots at the shaders. These shaders then read the data from these explicit bound slots during a draw call. If the game wants to draw using different resources, it will be done in the next draw call with a different view.

In Direct3D 12, you can create various resource views by using descriptor heaps. Each descriptor heap can be customized to be linked to a specific shader using specific resources. This flexibility to design the descriptor heap allows you to have full control over the resource usage pattern, fully utilizing modern hardware capabilities. You are also able to describe more than one descriptor heap that is indexed to allow easy flexibility to swap heaps, to complete a single draw call.

# Lights

We have briefly gone through the types of light in [Chapter 1](#), *An Overview of Unreal Engine*. Let us do a quick recap first. Directional Light emits beams of parallel lights. Point Light emits light like a light bulb (from a single point radially outward in all directions). Spot Light emits light in a conical shape outwards and Sky Light mimics light from the sky downwards on the objects in the level.

In this chapter, we will learn how to use these basic lights to illuminate an interior area. We have already placed a Point Light in [Chapter 2](#), *Creating Your First Level*, and learned how to adjust its intensity to 1700. Here in this chapter, we will learn more about the parameters that we can adjust with each type of light to create the lighting that we want.

Let us first view a level that has been illuminated using these Unreal lights. Load `Chapter4Level_Prebuilt.umap`, build and play the level to look around. Click on the lights that are placed in the level and you will notice that most of lights used are Point or Spot Light. These two forms of lights are quite commonly found in interior lighting.

The next section will guide you to extend the level on your own. Alternatively, you can use the `Chapter4Level_Prebuilt` level to help you along in the creation of your own level since it does take a fair amount of time to create the entire level. If you wish to skip to the next section, feel free to simply use the prebuilt version of the map provided, and go through the other examples in this chapter using the prebuilt map as a reference. However, it will be a great opportunity to revise what you have learned in the previous chapters and extend the level on your own.

Before we embark on the optional exercise to extend the level, let us go through a few tutorial examples on how we can place and configure the different types of light.

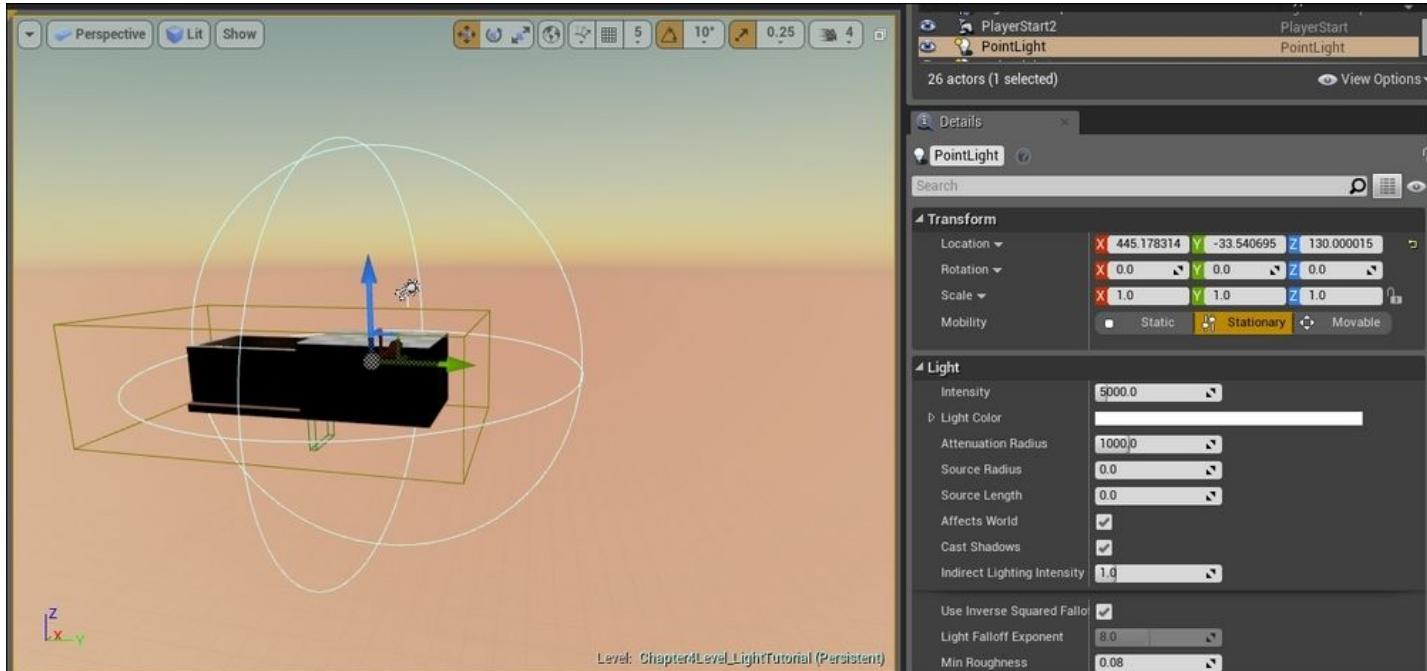
## Configuring a Point Light with more settings

Open `Chapter4Level.umap` and rename it `Chapter4Level_PointLight.umap`.

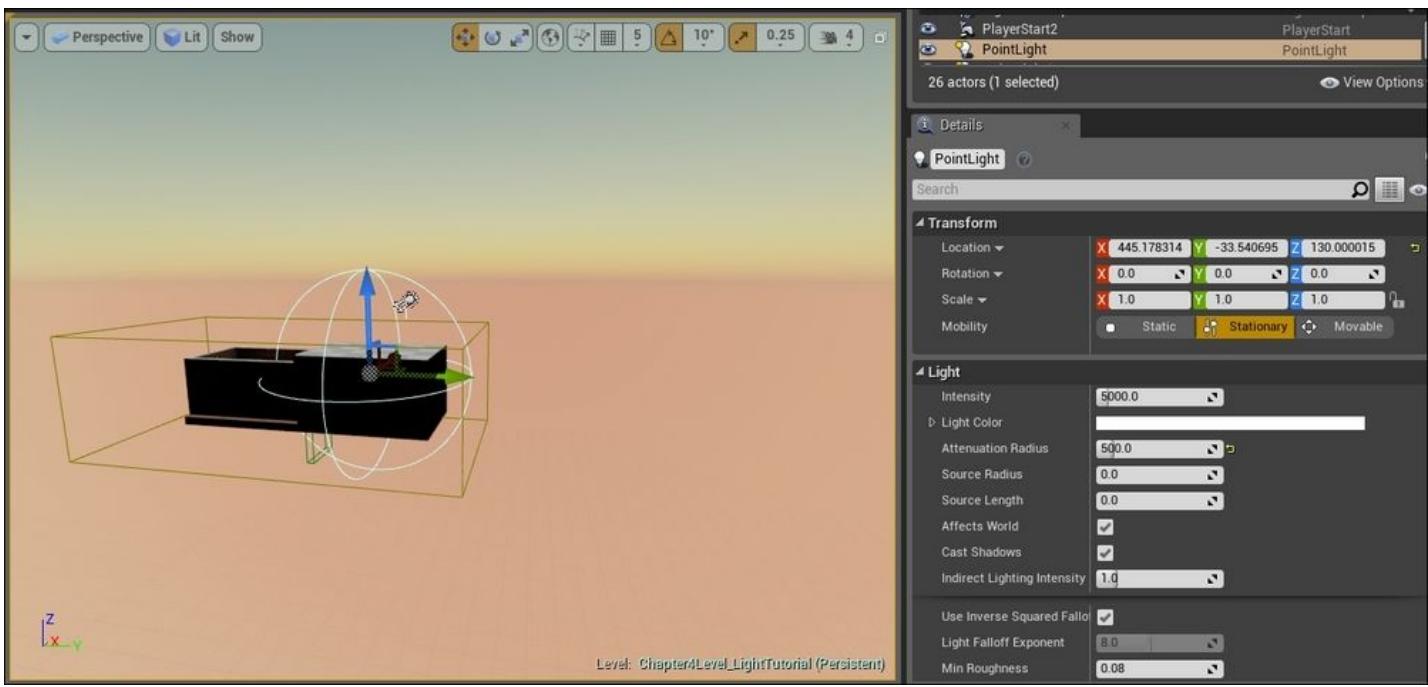
Go to **Modes | Lights**, drag and drop a Point Light into the level. As Point Light emits light equally in all directions from a single point, **Attenuation Radius**, **Intensity**, and **Color** are the three most common values that are configured for a Point Light.

### Attenuation Radius

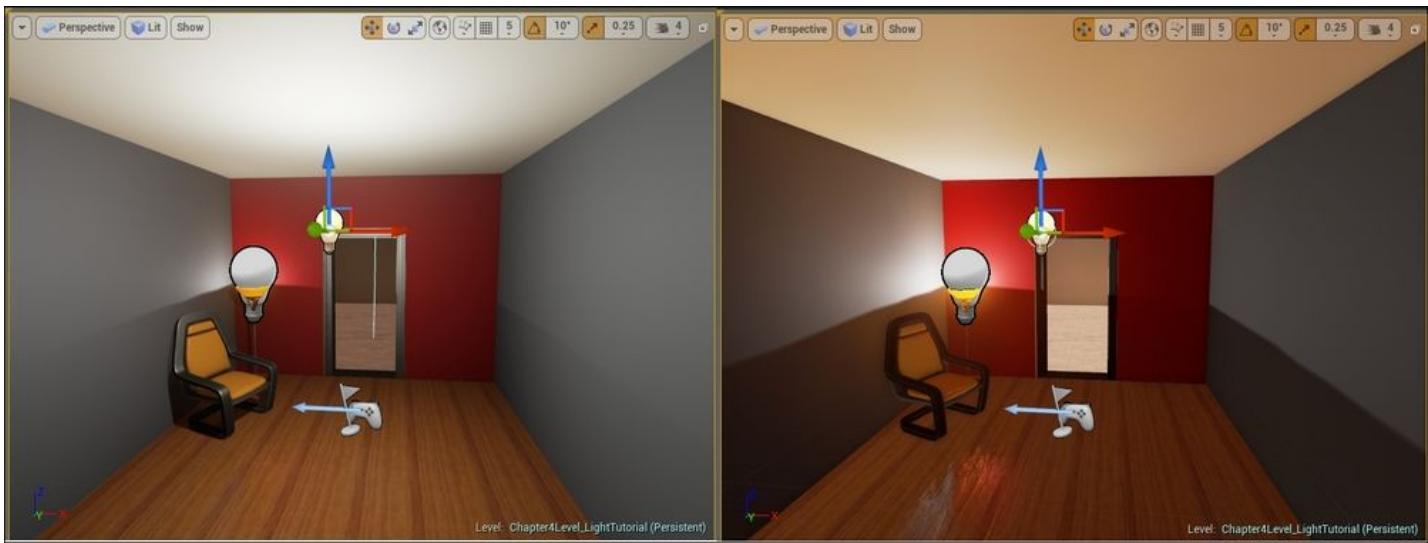
The following screenshot shows when the Point Light has its default **Attenuation Radius** of **1000**. The radius of the three blue circles is based on the attenuation radius of the Point Light and is used to show its area of effect on the environment.



The following screenshot shows when the attenuation radius is reduced to 500. In this situation, you probably cannot see any difference in the lighting since the radius is still larger than the room itself.



Now, let us take a look at what happens when we adjust the radius much smaller. The following screenshot shows the difference in light brightness when the radius changes. The image on the left is when the attenuation radius is set as 500 and the right when attenuation radius is set as 10.

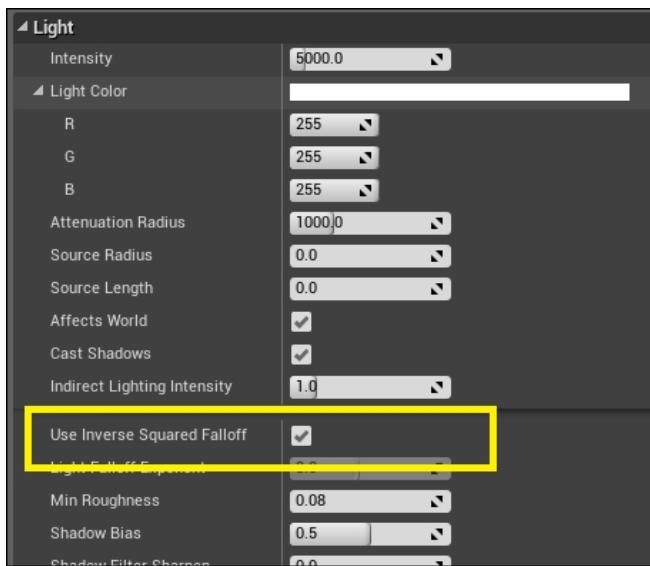


## Intensity

Another setting for Point Light is **Intensity**. Intensity affects the brightness of the light. You can play around the Intensity value to adjust the brightness of the light. Before we determine what value to use for this field and how bright we want our light to be, you should be aware of another setting, **Use Inverse Squared Falloff**.

### Use Inverse Squared Falloff

Point Lights and Spot Lights have physically based inverse squared falloff set on, as default. This setting is configurable as a checkbox found in the **Light** details under **Advanced**. The following screenshot shows where this property is found in the **Details** panel:

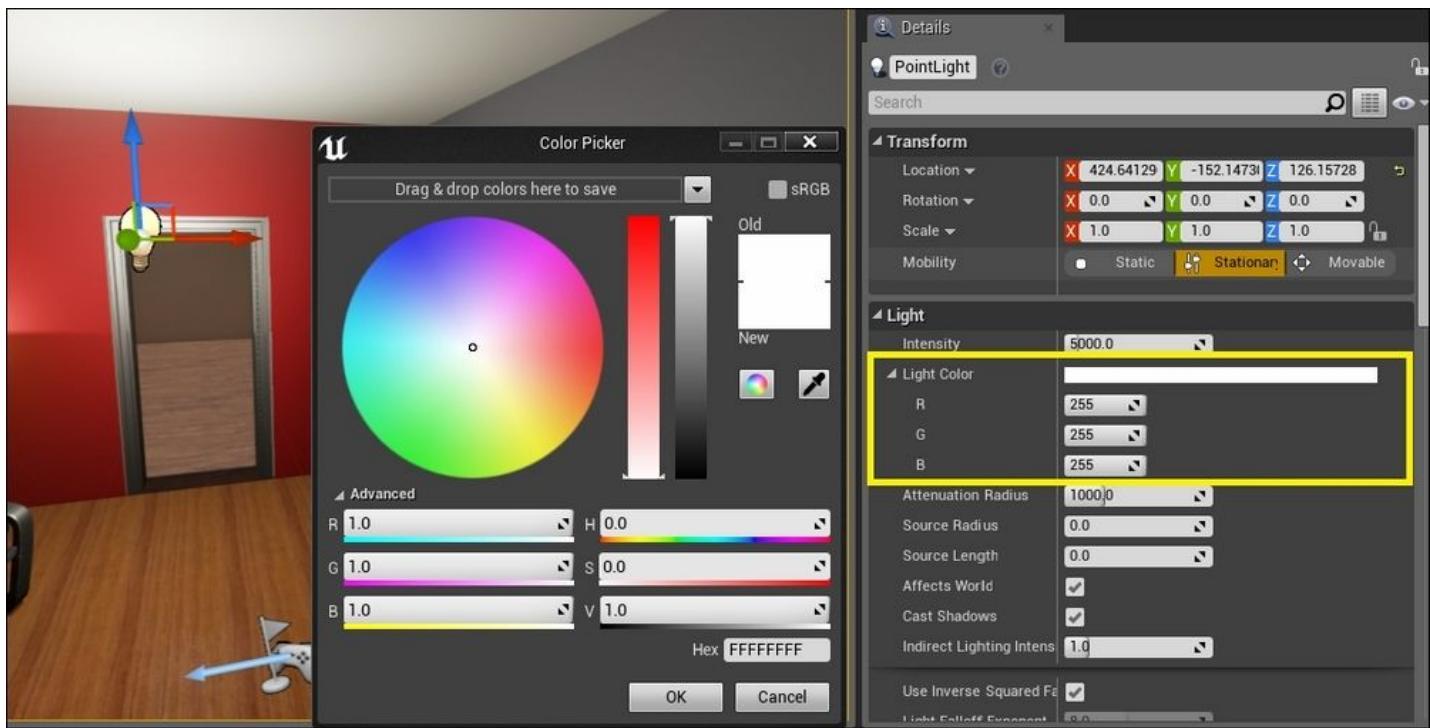


Inverse squared falloff is a physics law that describes how light intensity naturally fades over distance. When we have this setting, the units for intensity use the same units as the lights we have in the real world, in lumens. When inverse squared distance falloff is not used, intensity becomes just a value.

In the previous chapter where we have added our first Point Light, we have set intensity as 1700. This is equivalent to the brightness of a light bulb that has 1700 lumens because inverse squared distance falloff is used.

## Color

To adjust the color of Point Light, go to **Light | Color**. The following screenshot shows how the color of the light can be adjusted by specifying the RGB values or using the color picker to select the desired color:



## Adding and configuring a Spot Light

Open `Chapter4Level.umap` and rename it `Chapter4Level_SpotLight.umap`. Go to **Modes | Lights**, drag and drop a Spot Light into the level.

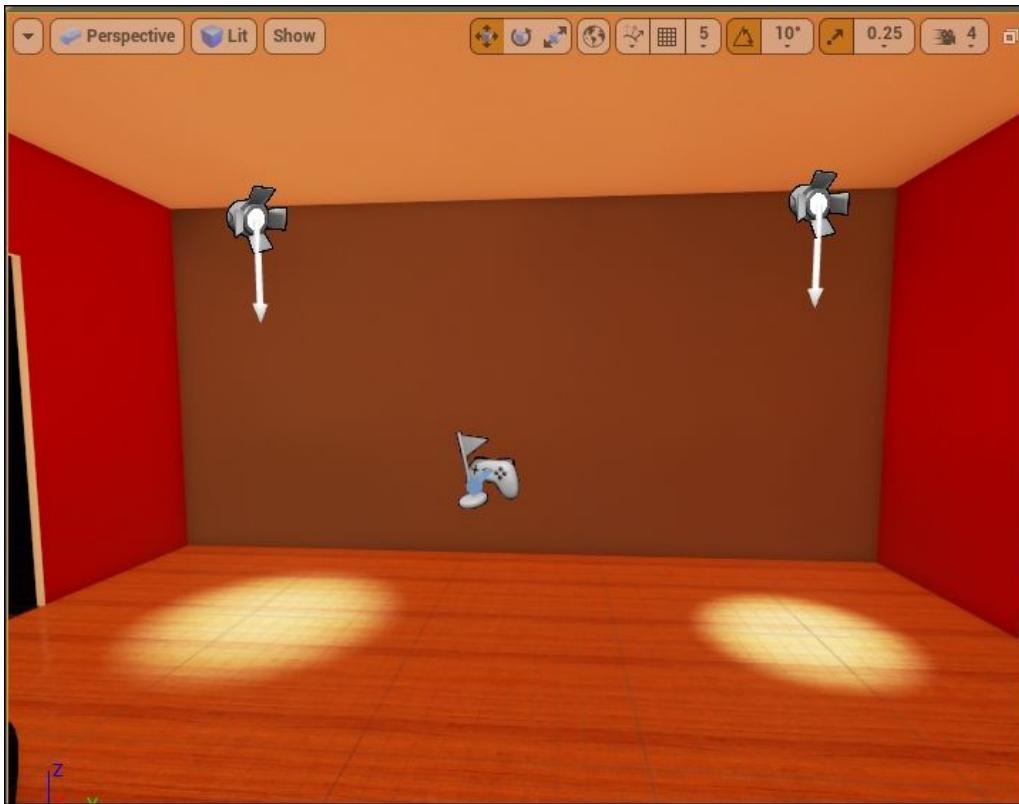
The brightness, visible influence radius, and color of a Spot Light can be configured in the same way as the Point Light through the value of **Intensity**, **Attenuation Radius**, and **Color**.

Since Point Light has light emitting in all directions and a Spot Light emits light from a single point outwards in a conical shape with a direction, the Spot Light has additional properties such as inner cone and outer cone angle, which are configurable.

### Inner cone and outer cone angle

The unit for the outer cone angle and inner cone angle is in degrees. The following screenshot shows the light radius that the spotlight has when the outer cone angle =

20 (on the left) and outer cone angle = 15 (on the right). The inner cone angle value did not produce much visible results in the screenshot, so very often the value is 0. However, the inner cone angle can be used to provide light in the center of the cone. This would be more visible for lights with a wider spread and certain IES Profiles.



## Using the IES Profile

Open `Chapter4Level_PointLight.umap` and rename it `Chapter4Level_IESPProfile.umap`.

IES Light Profile is a file that contains information that describes how a light will look. This is created by light manufacturers and can be downloaded from the manufacturers' websites. These profiles could be used in architectural models to render scenes with realistic lighting. In the same way, the IES Profile information can be used in Unreal Engine 4 to render more realistic lights. IES Light Profiles can be applied to a Point Light or a Spot Light.

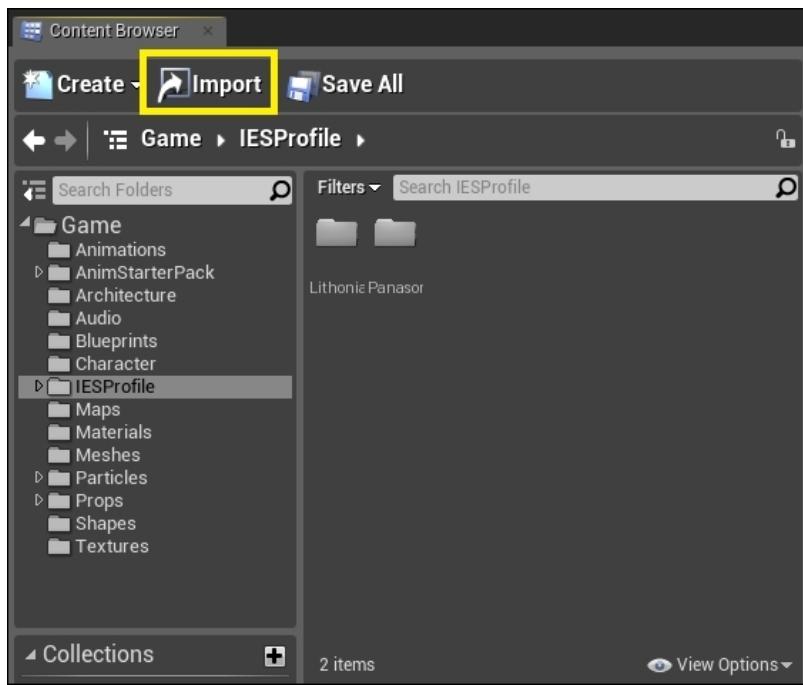
### Downloading IES Light Profiles

IES Light Profiles can be downloaded from light manufacturers' websites. Here's a few that you can use:

- Cooper Industries : [http://www.cooperindustries.com/content/public/en/lighting/resources/design\\_center\\_tools/photometric\\_tool\\_box.html](http://www.cooperindustries.com/content/public/en/lighting/resources/design_center_tools/photometric_tool_box.html)
- Philips : [http://www.usa.lighting.philips.com/connect/tools\\_literature/photometric\\_data\\_1.wpd](http://www.usa.lighting.philips.com/connect/tools_literature/photometric_data_1.wpd)
- Lithonia : <http://www.lithonia.com/photometrics.aspx>

### Importing IES Profiles into the Unreal Engine Editor

From **Content Browser**, click on **Import**, as shown in the following screenshot:



I prefer to have my files in a certain order, hence I have created a new folder called `IESProfile` and created subfolders with the names of the manufacturers to better categorize all the light profiles that were imported.

## Using IES Profiles

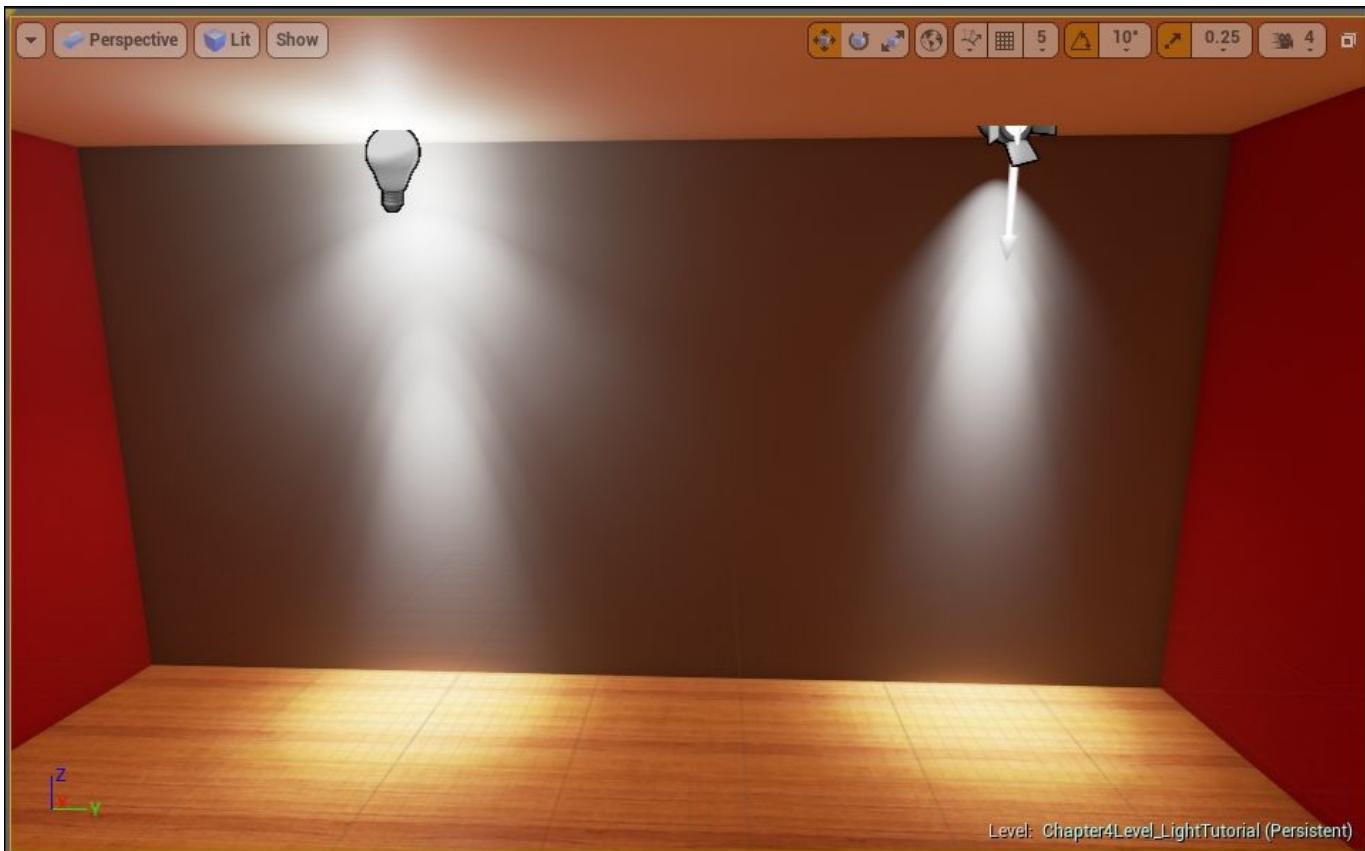
Continuing from the previous example, select the right Spot Light which we have in the scene and make sure it is selected. Go to the **Details** panel and scroll down to show the Light Profile of the light.

Then go to **Content Browser** and go to the `IESProfile` folder where we have imported the light profiles into. Click on one of the profiles that you want, drag and drop it on the IES Texture of the Spot Light. Alternatively, you can select the profile and go back to the **Details** panel of the **Light** and click on the arrow next to **IES Texture** to apply the profile on the Spot Light. In the following screenshot, I applied one of the profiles downloaded from the Panasonic website labeled **144907**.



I reconfigured the Spot Light with **Intensity = 1000**, **Attenuation Radius = 1000**, **Outer Cone Angle = 40**, and **Inner Cone Angle = 0**.

Next, I deleted the other Spot Light and replaced it with a Point Light where I set **Intensity = 1000** and **Attenuation Radius = 1000**. I also set the **Rotation-Y = -90** and then applied the same IES Profile to it. The following screenshot shows the difference when the same light profile is applied to a Spot Light and a Point Light. Note that the spread of the light in the Spot Light is reduced. This reinforces the concept that a Spot Light provides a conical shaped light with a direction spreading from the point source outwards. The outer cone angle determines this spread. The point light emits light in all directions and equally out, so it did not attenuate the light profile settings allowing the full design of this light profile to be displayed on the screen. This is one thing to keep in mind while using the IES Light Profile and which types of light to use them on.

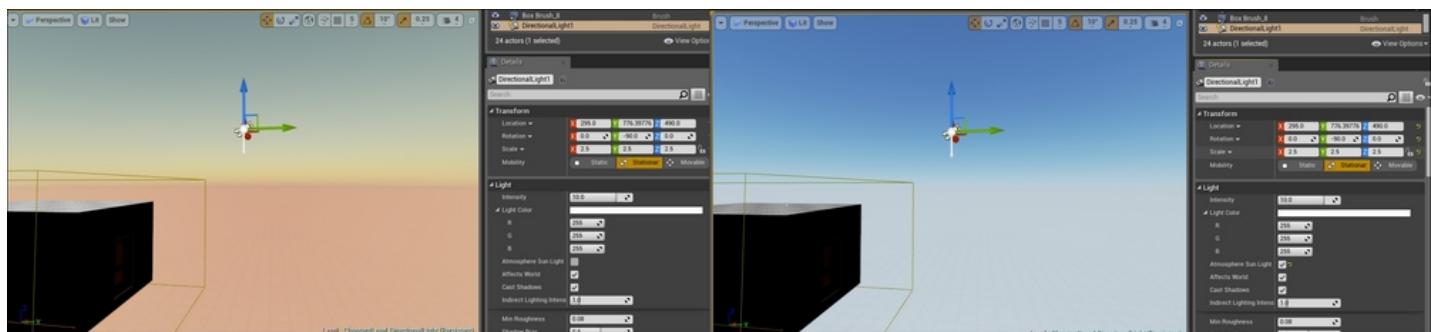


## Adding and configuring a Directional Light

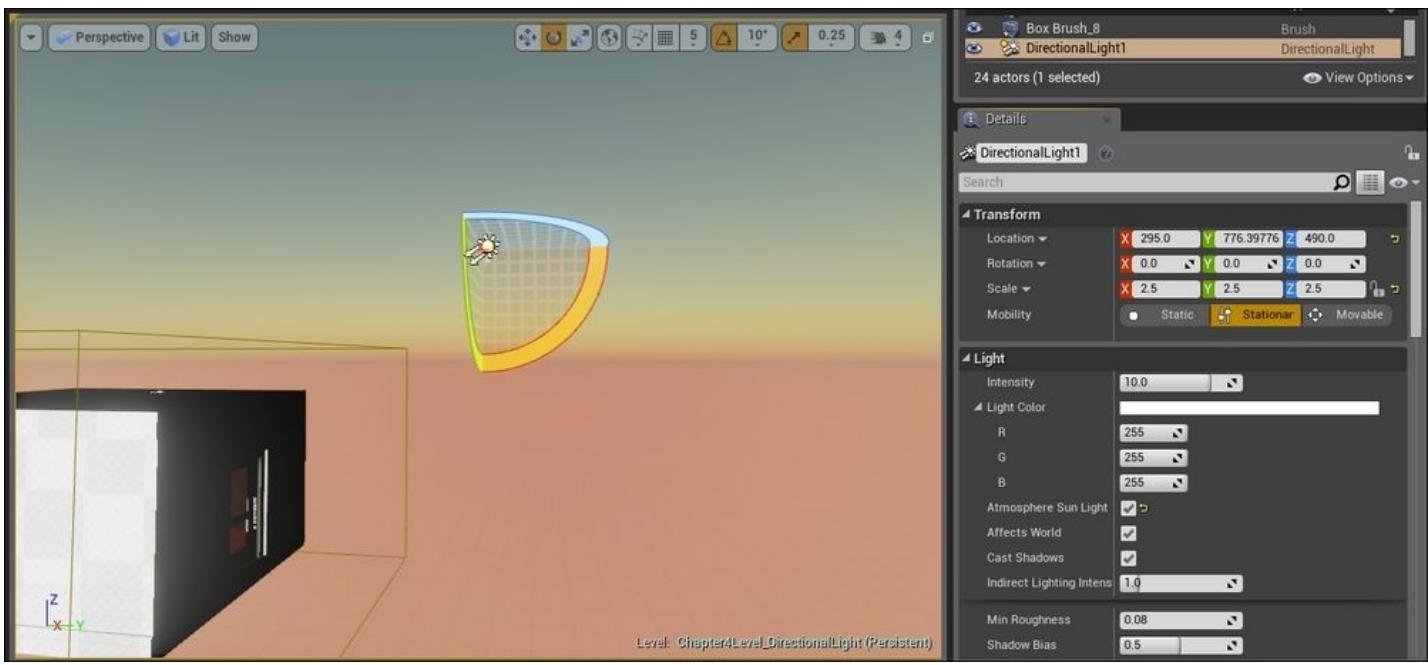
Open `Chapter4Level.umap` and rename it `Chapter4Level_DirectionalLight.umap`.

We have already added a Directional Light into our level in [Chapter 2, Creating Your First Level](#), and it provides parallel beams of light into the level.

Directional Light can also be used to light the level by controlling the direction of the sun. The screenshot on the left shows the Directional Light when the **Atmosphere Sun Light** checkbox is unchecked. The screenshot on the right shows the Directional Light when the **Atmosphere Sun Light** checkbox is checked. When the **Atmosphere Sun Light** checkbox is checked, you can control the direction of the sunlight by adjusting the rotation of Directional Light.



The following screenshot shows how this looks when **Rotation-Y=0**. This looks like an early sunset scene:



## Example – adding and configuring a Sky light

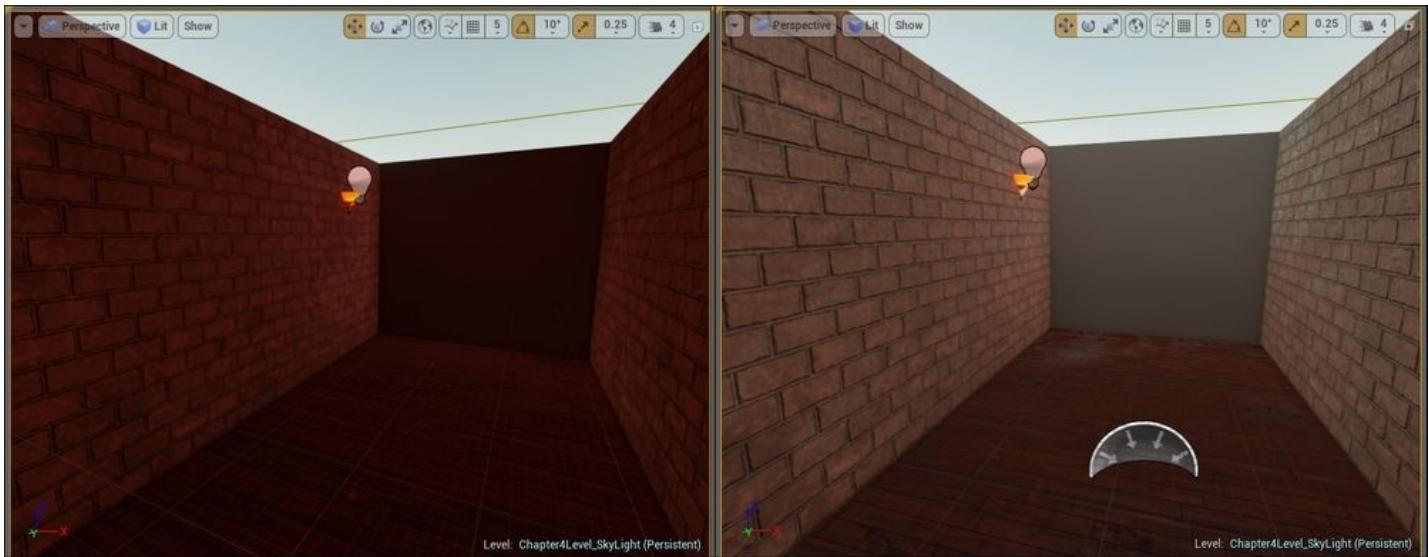
Open Chapter4Level\_DirectionalLight.umap and rename it Chapter4Level\_SkyLight.umap .

In the previous example, we have added sunlight control in the Directional Light. Build and compile to see how the level now looks.

Now, let us add a Sky Light into the level by going to **Modes | Lights** and then clicking and dragging Sky Light into the level. When adding a Sky Light to the level, always remember to build and compile first in order to see the effect of the Sky Light.

What does a Sky Light do? Sky Light models the color/light from the sky and is used to light up the external areas of the level. So the external areas of the level look more realistic as the color/light is reflecting off the surfaces (instead of using simple white/colored light).

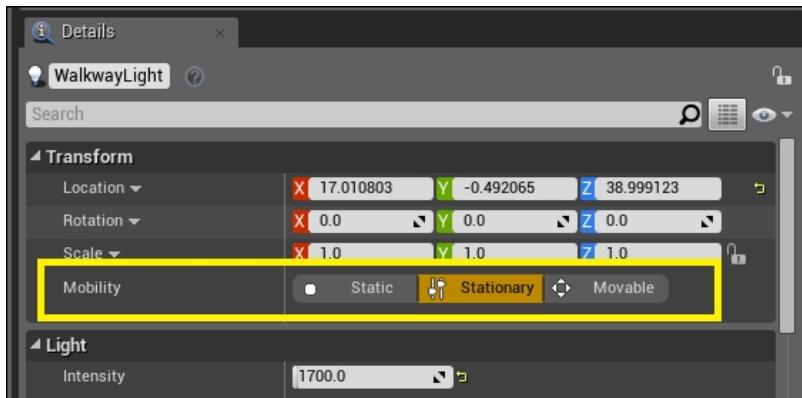
The following screenshot shows the effect of a Sky Light. The left image shows the Sky Light not in the level. The right one shows the Sky Light. Note that the walls now have a tinge of the color of the sky.



## Static, stationary, or movable lights

After learning how to place and configure the different lights, we need to consider what kind of lights we need in the level. If you are new to the concept of light, you might want to briefly go through the useful light terms section to help in your understanding.

The following screenshot shows the **Details** panel where you can change a light to be static, stationary, or movable.



**Static** and **Stationary** light sounds pretty much similar. What is the difference? When do you want to use a **Static** light and when do you want to use a **Stationary** light?

### Common light/shadow definitions

The common light/shadow definitions are as follows:

- **Direct Light** : This is the light that is present in the scene directly due to a light source.
- **Indirect Light** : This is the light in the scene that is not directly from a light source. It is reflected light bouncing around and it comes from all sides.
- **Light Map** : This is a data structure that stores the light/brightness information about an object. This makes the rendering of the object much quicker because we already know its color/brightness information in advance and it is not necessary to compute this during runtime.
- **Shadow Map** : This is a process created to make dynamic shadows. It is fundamentally made up of two passes to create shadows. More passes can be added to render nicer shadows.

### Static Light

In a game, we always want to have the best performance, and Static Light will be an excellent option because a Static Light needs only to be precomputed once into a Light Map. So for a Static Light, we have the lowest performance cost but in exchange, we are unable to change how the light looks, move the light, and integrate the effect of this light with moving objects (which means it is unable to create a shadow for the moving object as it moves within the influence of the light) into the environment during gameplay. However, a Static Light can cast shadow on the existing stationary objects that are in the level within its influence of radius. The radius of influence is based on the source radius of the light. In return for low performance cost, a Static Light has quite a bit of limitation. Hence, Static Lights are commonly used in the creation of scenes targeted for devices with low computational power.

### Stationary Light

Stationary Light can be used in situations when we do not need to move, rotate, or change the influence radius of the light during gameplay, but allow the light the capacity to change color and brightness. Indirect Light and shadows are prebaked in Light Map in the same way as Static Light. Direct Light shadows are stored within Shadow Maps.

Stationary Light is medium in performance cost as it is able to create static shadow on static objects through the use of distance field shadow maps. Completely dynamic light and shadows is often more than 20 times more intensive.

### Movable Light

Movable Light is used to cast dynamic light and shadows for the scene. This should be used sparingly in the level, unless absolutely necessary.

### Exercise – extending your game level (optional)

Here are the steps that I have taken to extend the current **Level14** to the prebuilt version of what we have right now. They are by no means the only way to do it. I have simply used a Geometry Brush to extend the level here for simplicity. The following screenshot shows one part of the extended level:



## Useful tips

Group items in the same area together when possible and rename the entity to help you identify parts of the level more quickly. These simple extra steps can save time when using the editor to create a mock-up of a game level.

## Guidelines

If you plan to extend the game level on your own, open and load `Level4.umap`. Then save map as `Level4_MyPreBuilt.umap`. You can also open a copy of the extended level to copy assets or use it as a quick reference.

### Area expansion

We will start by extending the floor area of the level.

#### Part 1 – lengthening the current walkway

The short walkway was extended to form an L-shaped walkway. The dimensions of the extended portion are X1200 x Y340 x Z40.

BSPs needed	X	Y	Z
Ceiling	1200	400	40
Floor	1200	400	40
Left wall	1570	30	280
Right wall	1260	30	280

#### Part 2 – creating a big room (living and kitchen area)

The walkway leads to a big room at the end, which is the main living and kitchen area.

BSPs needed	X	Y	Z
Ceiling	2000	1600	40
Floor	2000	1600	40
The left wall dividing the big room and walkway (the wall closest to you as you enter the big room from the walkway)	30	600	340

The light wall dividing the big room and walkway (the wall closest to you as you enter the big room from the walkway)	30	600	340
<b>BSPs needed</b>	<b>X</b>	<b>Y</b>	<b>Z</b>
The left wall of the big room (where the kitchen area is)	1200	30	340
The right wall of the big room (where the dining area is)	2000	30	340
The left wall to the door (the wall across the room as you enter from the walkway, where the window seats are)	30	350	340
The right wall to the door (the wall across the room as you enter from the walkway, where the long benches are)	30	590	340
Door area (consists of brick walls, door frames, and door)			
Wall filler left	30	130	340
Wall filler right	30	126	340
Door x 2	20	116	250
Side door frame x 2	25	4	250
Horizontal door frame	25	242	5
Side brick wall x 2	30	52	340
Horizontal brick wall	30	242	74

#### Part 3 – creating a small room along the walkway

To create the walkway to the small room, duplicate the same doorframe that we have created in the first room.

BSPs needed	X	Y	Z
Ceiling	800	600	40
Floor	800	600	40
Side wall x 2	30	570	340
Opposite wall (wall with the windows)	740	30	340

#### Part 4 – Creating a den area in the big room

BSPs needed	X	Y	Z
Sidewall x 2	30	620	340
Wall with shelves	740	30	340

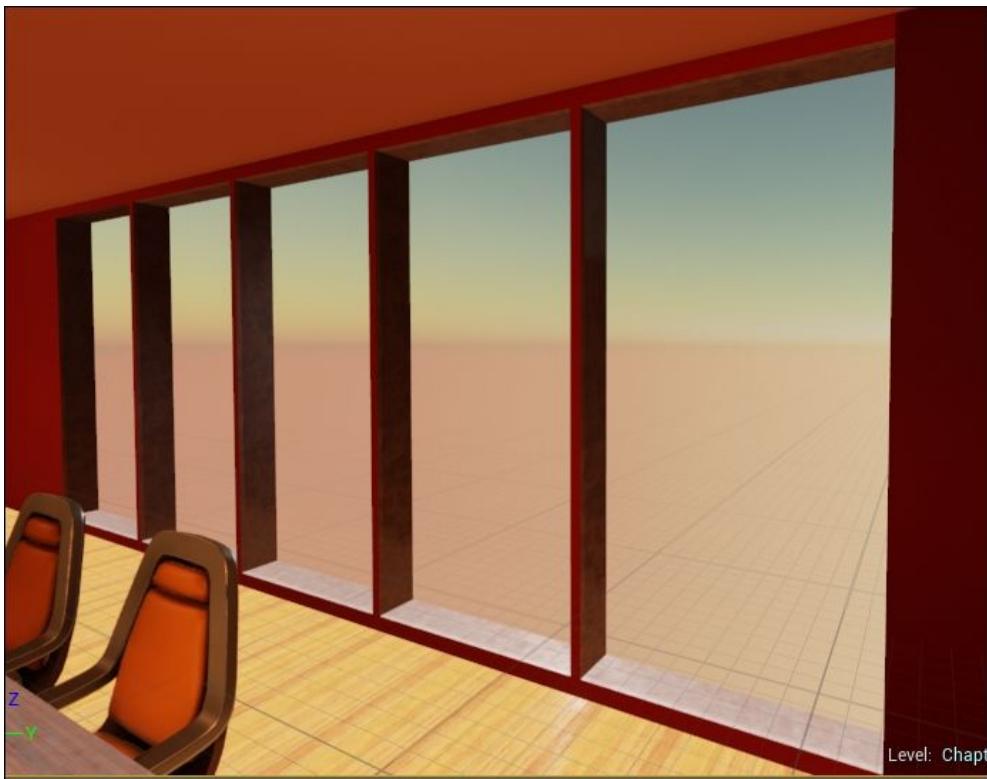
#### Creating windows and doors

Now that we are done with rooms, we can work on the doors and windows.

#### Part 1 – creating large glass windows for the dining area

To create the windows, we use a subtractive Geometry Brush to create holes in the wall. First, create one of size X144 x Y30 x Z300 and place it right in the middle between the ceiling and ground. Duplicate this and convert it to an additive brush; adjust the size to X142 x Y4 x Z298.

Apply **M\_Metal\_Copper** for the frame and **M\_Glass** to the addition brush, which was just created. Now, group them and duplicate both the brushes four times to create five windows. The screenshot of the dining area windows is shown as follows:



#### Part 2 – creating an open window for the window seat

To create the window for the window seat area, create a subtractive geometry brush of size X50 x Y280 x Z220. For this window, we have a protruding ledge of X50 x Y280 x Z5 at the bottom of the window. Then for the glass, we duplicate the subtractive brush of size X4 x Y278 x Z216, convert it to additive brush and adjust it to fit.

Apply **M\_Metal\_Brushed** for the frame and **M\_Glass** to the addition brush that was just created.



#### Part 3 – creating windows for the room

For the room windows, create a subtractive brush of size X144 x Y40 x Z94. This is to create a hollow in the wall for the prop frame: **SM\_WindowFrame**. Duplicate the subtractive brush and prop to create two windows for the room.

#### Part 4 – creating the main door area

For the main door area, we start by creating the doors and its frame, then the brick walls around the door and lastly, the remaining concrete plain wall.

We have two doors with frames then some brick wall to augment before going back to the usual smooth walls. Here are the dimensions for creating this door area:

BSPs needed	X	Y	Z
Actual door x 2	20	116	250

BSPs needed	X	Y	Z
Top frame	25	242	5

Here are the dimensions for creating the area around the door:

BSPs needed	X	Y	Z
Brick wall side x 2	30	52	340
Brick wall top	30	242	74
Smooth wall left	30	126	340
Smooth wall right	30	130	360

### Creating basic furniture

Let us begin it part by part as follows.

#### Part 1 – creating a dining table and placing chairs

For the dining table, we will be customizing a wooden table with a table top of size X480 x Y160 x Z12 and two legs each of size X20 x Y120 x Z70 placed 40 from the edge of the table. Material used to texture is **M\_Wood\_Walnut**.

Then arrange eight chairs around the table using **SM\_Chair** from the **Props** folder.

#### Part 2 – decorating the sitting area

There are two low tables in the middle and one low long table at the wall. Place three **SM\_Couch** from the **Props** folder around the low tables. Here are the dimensions for the larger table:

BSPs needed	X	Y	Z
Square top	140	140	8
Leg x 2	120	12	36

Here are the dimensions for the smaller table:

BSPs needed	X	Y	Z
Leg x 2	120	12	36

Here are the dimensions for a low long table at the wall:

BSPs needed	X	Y	Z
Block	100	550	100

#### Part 3 – creating the window seat area

Next to the open window, place a geometry box of size X120 x Y310 x Z100. This is to create a simplified seat by the window.

#### Part 4 – creating the Japanese seating area

The Japanese square table with surface size X200 x Y200 x Z8 and 4 short legs, each of size X20 x Y20 x Z36) is placed close to the corner of the table.

To create a leg space under the table, I used a subtractive brush (X140 x Y140 x Z40) and placed it on the ground under the table. I used the corner of this subtractive brush as a guide as to where to place the short legs for the table.

#### Part 5 – creating the kitchen cabinet area

This is a simplified block prototype for the kitchen cabinet area. The following are the dimensions for L-shaped area:

BSPs needed	Material	X	Y	Z

Shorter L: cabinet under tabletop	<b>M_Wood_Walnut</b>	140	450	100
<b>BSPs needed</b>	<b>Material</b>	<b>X</b>	<b>Y</b>	<b>Z</b>
Longer L: cabinet under tabletop	<b>M_Wood_Walnut</b>	890	140	100
Shorter L: tabletop	<b>M_Metal_Brushed_Nickel</b>	150	450	10
Longer L: tabletop	<b>M_Metal_Brushed_Nickel</b>	900	150	10
Shorter L: hanging cabinet	<b>M_Wood_Walnut</b>	100	500	100
Longer L: hanging cabinet	<b>M_Wood_Walnut</b>	900	100	100

The following are the dimensions for the island area (hood):

<b>BSPs needed</b>	<b>Material</b>	<b>X</b>	<b>Y</b>	<b>Z</b>
Hood (wooden area)	<b>M_Wood_Walnut</b>	400	75	60
Hood (metallic area)	<b>M_Metal_Chrome</b>	500	150	30

The following are the dimensions for the island area (table):

<b>BSPs needed</b>	<b>Material</b>	<b>X</b>	<b>Y</b>	<b>Z</b>
Cabinet under the table	<b>M_Wood_Walnut</b>	500	150	100
Tabletop	<b>M_Metal_Chrome</b>	550	180	10
Sink (use a subtractive brush)	<b>M_Metal_Chrome</b>	100	80	40
Stovetop	<b>M_Metal_Burnished_Steel</b>	140	100	5

# Summary

In this chapter, we covered in-depth information about materials and lights. We learned how the rendering system works and the underlying graphics pipeline/technology such as Directx 11, DirectX 12, and OpenGL/Vulkan. We also learned how to use the Unreal 4 Material Editor to create custom materials and apply it into your level.

We also explored the different types of lights and adjusting **Intensity**, **Attenuation Radius**, and other settings to customize lights for the level. We also learned how to import IES light profiles from light manufacturer's website to create realistic lights for the level. We learned about the differences between **Static**, **Stationary**, and **Movable** lights and how the different lights cast shadows for the level.

In the next chapter, we will learn about animation and artificial intelligence in games. Stay tuned for more!

# Chapter 5. Animation and AI

This chapter is about animation and **artificial intelligence ( AI )**.

Animation is what we need in order to see things move in a game. AI is what is required for characters (other than the player) to know how to behave and react while you are in the game.

We will cover the following topics in this chapter:

- Definition of animation
- 3D animation
- Tools required for animation in Unreal Engine 4
- Learning to add animation to your game
- Using an Animation Blueprint
- Learning about Blend Animation
- AI in games
- Designing a **Behavior Tree ( BT )**
- Using a Blueprint to implement AI in your game

## What is animation?

Animation is the simulation of movement through a series of images or frames.

Before computers came into the picture, animation was created using traditional techniques such as hand-drawn animation and stop-motion animation (or model animation). Hand-drawn animation, as the name suggests, involves hand-drawn scenes on paper. Each scene is repeated on the next sheet of paper with a slight change in the scene. All the papers are put together in sequence and the pages are turned very quickly, like a flipbook. The slight changes on the sheets of paper create 2D animation, and this can be filmed into a motion film. This technique is used very often in Disney cartoons and movies. As you can imagine, this is a very time-consuming way to produce animation, as you would need thousands of drawings to create seconds of the film.

Stop-motion animation involves creating models, moving them a little in each frame to mimic movement, and filming this sequence to construct an entire scene. The tedious process of capturing countless snippets has limited the use of this method in favor of more mainstream animation techniques today.

Computer animation is quite similar to stop-motion animation as computer graphics is moved a little in each frame; these frames are then rendered on screen. For computer games, we use computer animation by creating 3D models using tools, such as Maya and 3ds Max. Then, we animate these models to simulate life-like behavior and actions for the game. Animation is needed for all things in order to make them move. Characters need to be animated so that they can look real—they can be in an idle position, walk, run, or execute any other action that needs to be performed in the course of the game.

Motion capture is also another very popular way to animate characters these days. This technology basically uses recorded human actions to create the computer graphic character's behavior. If you have watched the movie *Avatar*, the blue avatar characters were, in fact, played by human actors and then enhanced to look the way they did using computer graphics. For the filming of the movie, they advanced the motion capture technology into what is now called **performance capture**. This advancement in technology has empowered film and game makers to capture the details in animation in such a way that can make a CG character stand out.

# Understanding how to animate a 3D model

Although the objective of this book is not to teach you how to animate a model, it is important to understand how animation is done so that you can understand better how to get game characters in a game to move and behave according to design.

As mentioned earlier, we can animate 3D models using tools, such as Maya or 3ds Max. We can then record their changes and then render these animations on screen when needed.

## Preparing before animation

In game development, the creation of animation falls under the responsibility of an animator. Before an animation can be first created, we need to first have a 3D model that's been created by a 3D modeler. The 3D modeler is responsible for giving the object its shape and texturing it. Depending on the type of object we're dealing with, the exact process to get an object properly rigged can be slightly different. Rigging needs to be done before handing over the object to the animator to create specific animations. Sometimes, animators also need to fine-tune the rigs for better control of the animation.

Rigging is a process where a skeleton is placed in the mesh and joints that are created for the skeleton. The collection of bones/joints is known as the **rig**. The rig provides control points, which the animator can use to move the object in order to create the desired animation. I will use a human character model in my explanation here so that you can understand this concept easily.

The 3D or character modeler first shows how the face and body of a model are shaped. It then determines how tall the model is, creates all the required features by adding primitives to the model, and then textures it to give color to its eyes, hair, and so on. The model is now ready but still jelly on the inside because we have not given it any internal structure. Rigging is the process where we add bones to the body to hold it up. The arm can be rotated because we have given it a shoulder bone (scapula), arm bone (humerus), and a joint that can mimic the ball and socket joint. The joint we have in place for rigging is made up of a group of constraints that limit movement in various planes and angles. Hierarchies are also applied to the bone structure to help the bones link each other. The fingers are linked to the hand, which is linked to the arm. Such a relationship can be put in place so that movement looks real when one of parts moves and the rest of the parts naturally move together as well.

Tools, such as Maya and 3ds Max, provide some simplification to the rigging process, as you can use standard rigs as the base, and tweak this base according to the needs of the model. Some models are taller and require longer bones. A 3D model must have a simple skeletal structure that adheres closely to the shape and size of a 3D model. Similar sized 3D models can share the same skeletal structure.

To better understand how we can add animation to our game levels, let's learn how computer animation is created and how we can make these models move.

## How is animation created?

Animation basically mimics how life moves in the real world. Many companies go to great lengths to make computer animation as accurate as possible through the use of motion capture. They film actual movements in real life and then recreate these movements using computer 3D models.

When creating animations, the animator makes use of the bones and joints created during the rigging process and adjusts them in place using as much detail as possible to mimic their natural movement. The joints and bones work together to affect the body posture. These movements are then recorded as short animation clips known as an animation sequence. Animation sequences form the most basic blocks of animation, and they can be played once or repeatedly to create an action. For example, a walking animation is only 1.8 seconds long but can be replayed over and over to simulate walking. When this sequence is repeated again, it is commonly known as an animation loop.

Animation sequences can also be linked to form a chain of actions. While transitioning from one sequence to another, some blending might be needed in order for the movement to look natural.

# What Unreal Engine 4 offers for animation in games

Animation in Unreal Engine 4 is mostly done in the Persona editor. This editor offers four different modes: **Skeleton**, **Mesh**, **Animation**, and **Graph**. These modes mainly exist so that you can jump straight into one of them to edit/create the animations more effectively. So, they are simply a loose group of functions that can be used to control the different aspects of animation. We will learn how to make use of the functions in Persona to add animation to our level.

To help improve team collaboration, Unreal Engine 4 also released a previously in-house-only toolset, which is a plugin for Maya (compatible for Maya 2013 and higher versions), known as **Animation and Rigging Toolset (ART)**. This toolset provides a user interface to allow the creation of a skeleton, placement of the skeleton, and rig creation within Maya itself. We will not go into the details of this toolset, but you can find more information on this in Unreal's online documentation at <https://docs.unrealengine.com/latest/INT/Engine/Content/Tools/MayaRiggingTool/index.html>.

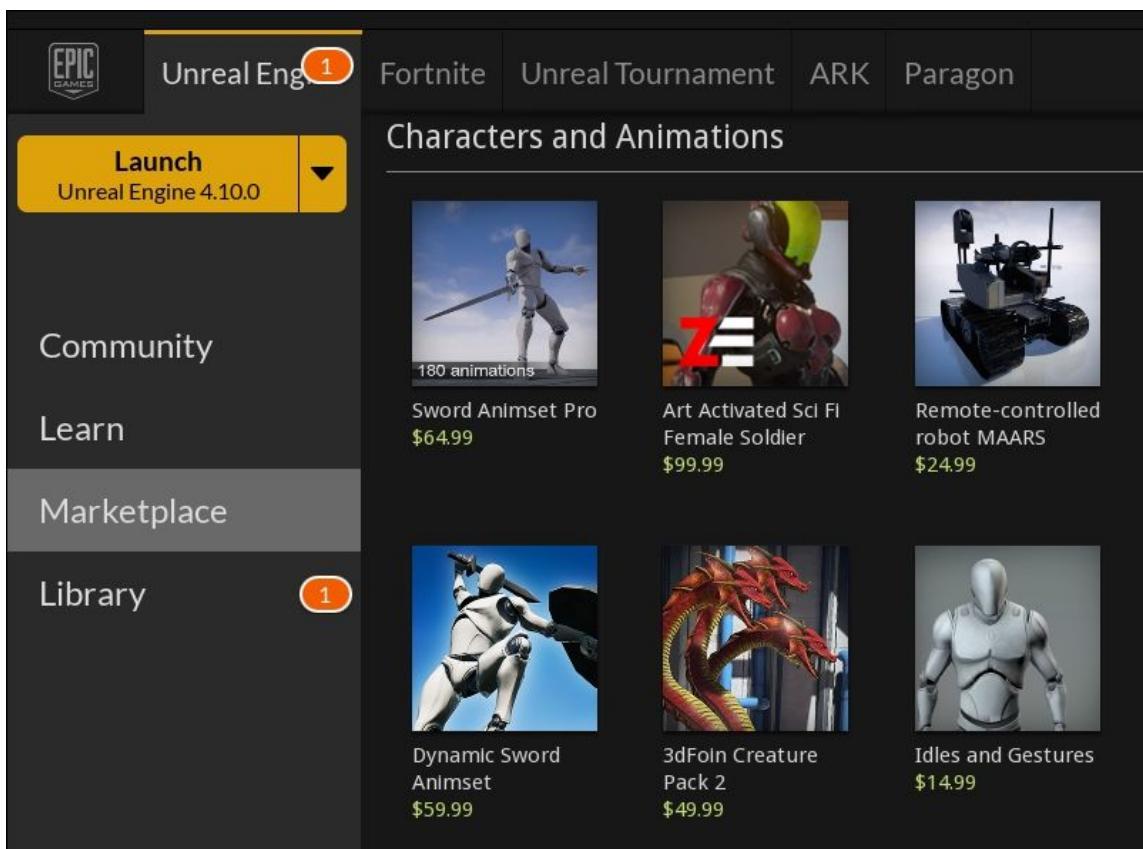
## Importing animation from Maya/3ds Max

As many artists use Maya and 3ds Max to create 3D models and animation, Unreal Engine 4 has a great FBX Import pipeline that allows you to successfully import skeletal models, animation sequences, and morph targets. This makes it easy to transfer assets to the Unreal Editor and put them into the game. Unreal also tries to stabilize the import of art assets from other software, such as Blender and MODO.

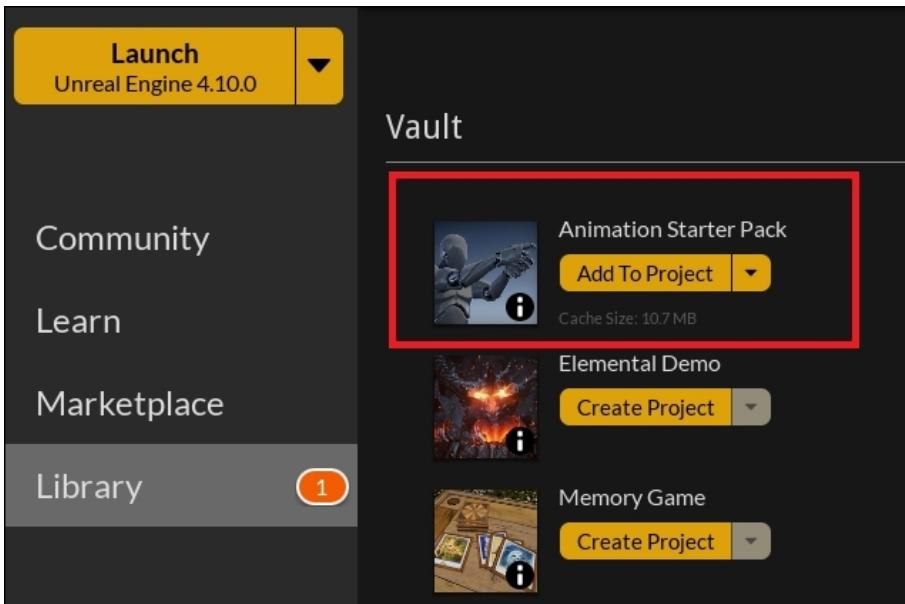
### Tutorial – importing the animation pack from Marketplace

Since 3D models and animation are first created outside Unreal Engine, for the purpose of learning about how animation works, we will import an animation pack that contains a 3D model with a number of animation sequences first, and we'll then learn how to make use of the different tools in the Unreal Editor for animation.

Unreal Engine offers a number of downloadable packs in Marketplace. Marketplace is in the start menu screen, which is under the **Launch** button. The following screenshot shows the startup screen that has the **Marketplace** tab selected for the downloadable packs. Search for **Animation Starter Pack** in Marketplace under **Characters and Animations**. This particular pack is free to download. Click on **Animation Started Pack** to download it.

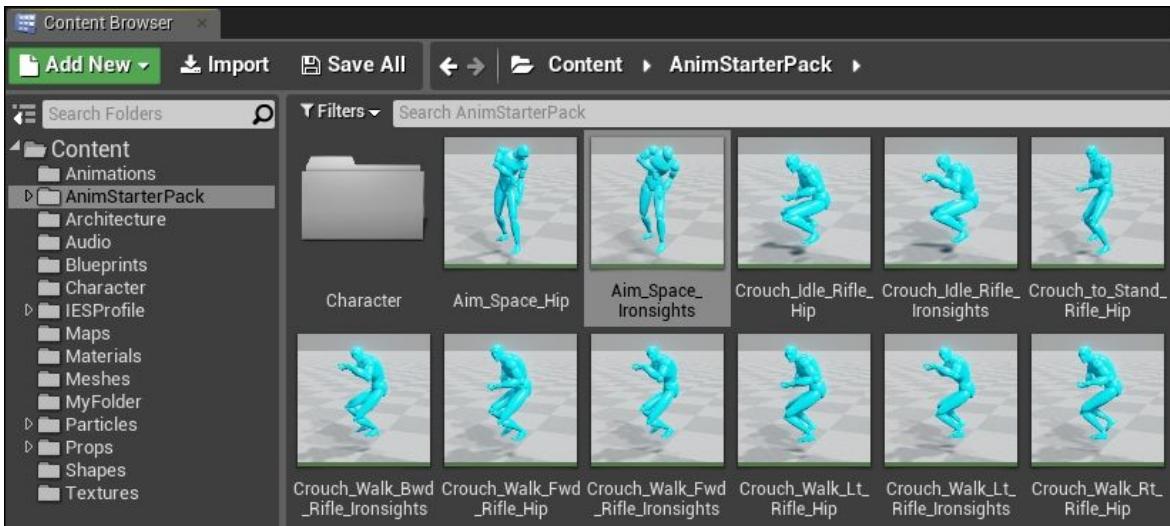


After the pack is downloaded, you will find the pack added to the **Library**. The following screenshot shows where **Animator Starter Pack** is found in **Library** under **Vault**:



Now that we have the **Animation Starter Pack** in our **Library**, we can add it to our current project and start playing with the animations.

Click on **Add To Project** and a pop-up screen with all the current projects that are present in Unreal Engine will appear. Select the name of the project that you have been creating for all the various levels and all the tutorial examples. If you have followed the same project and level naming convention as me, it will be `MyProject`. I have also opened `Chapter4Level` from the previous chapter and renamed it `Chapter5Level`. The following screenshot shows `AnimStarterPack` loaded in the project:



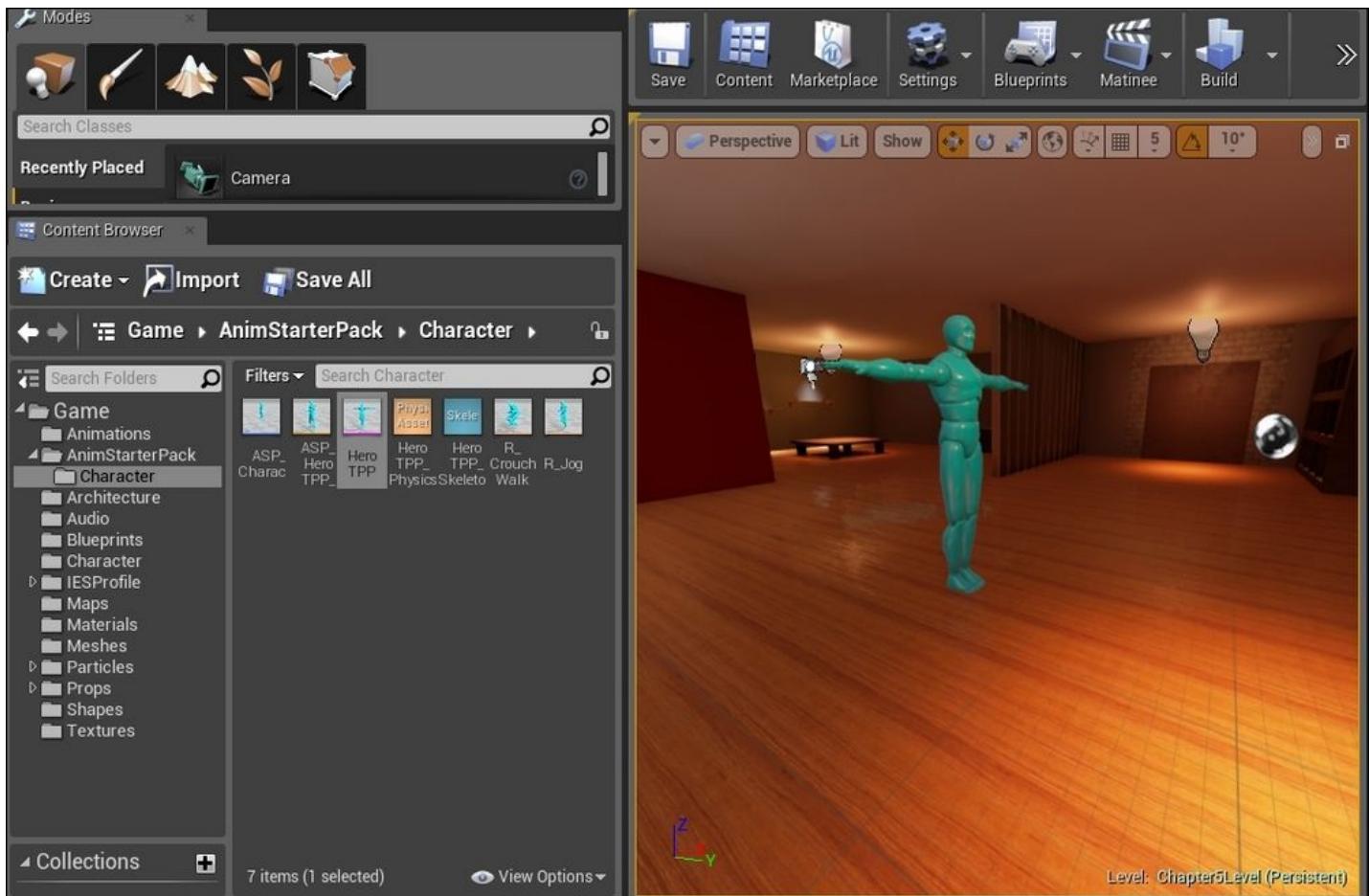
## What can you do with Persona?

Persona gives game developers the ability to playback and preview animation sequences, combine animation sequences into a single animation by blending, creating montages, editing skeletons/sockets, and controlling animation with Blueprints. I hope you still remember what you have learned about Blueprints in [Chapter 3](#), *Game Object – More and Move*.

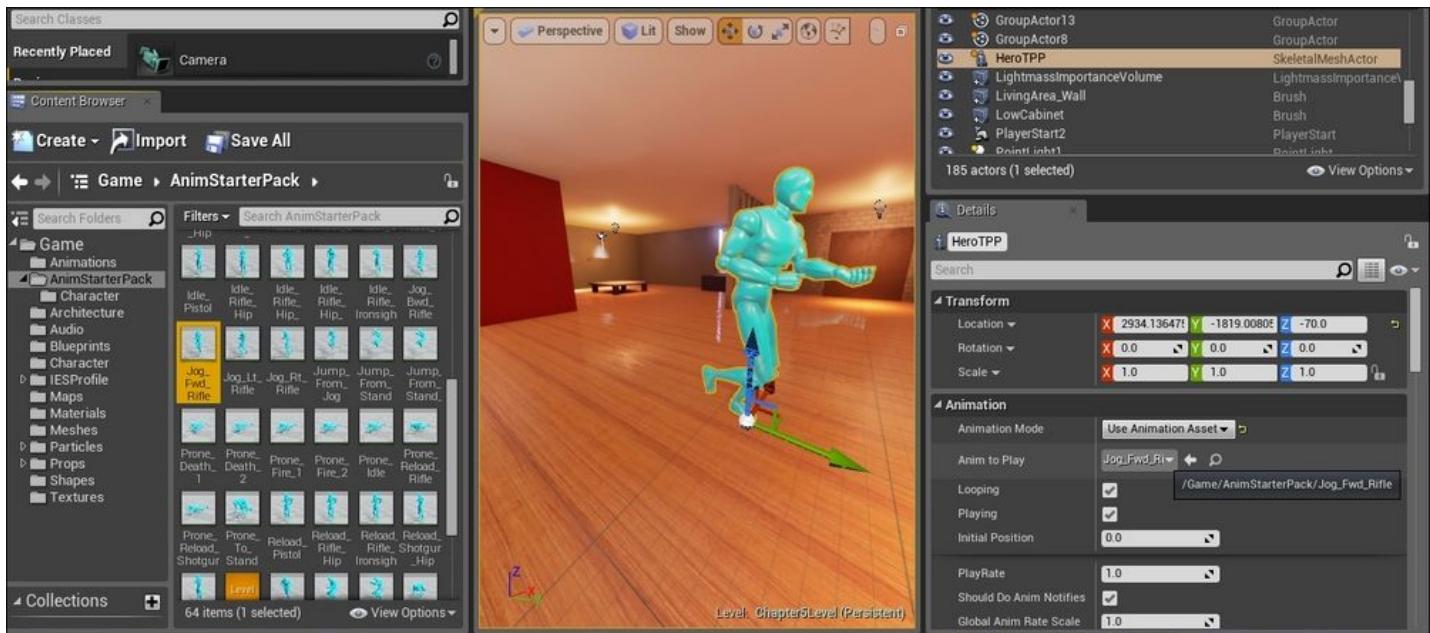
### Tutorial – assigning existing animation to a Pawn

After adding the free animation pack into your project in the previous exercise, it is time to add some animation to the level. First of all, open `Chapter4Level`, rename it `Chapter5Level`, and then navigate to the `AnimStarterPack` folder using **Content Browser**. Go to the `Character` subfolder and click and drag `HeroTPP` into the level.

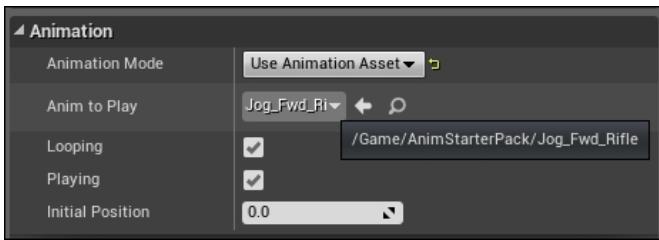
This screenshot shows how `HeroTPP` is added to the level:



The **HeroTPP** looks fake and frozen, right? Now, let's give him a better pose. Click on **HeroTPP** to display the details. Go to the **Animation** tab under **Details** and input the **Animation Mode** settings. Use **Animation Asset**, navigate and click on **Jog\_Fwd\_Rifle** in **AnimStarterPack** (in **Content Browser**), and then click on the arrow next to **Anim to Play**.



Here is a zoomed-in view of the **Animation** settings:



Now, build and play the level. You will see the character that you have just added to the level, is jogging.

This is the straightforward way to animate a character. However, the character continues to loop through this animation no matter what is happening around. We probably want the character to be able to react to the environment and conditions of the game. So, how can we do this?

## Why do we need to blend animations?

In the previous exercise, we learned how to make a skeletal mesh take on a single animation. But can we make the skeletal mesh start running in a straight line? The next few sections of animation exercises will explain how we can do this and, subsequently, add more to this basic animation.

First of all, you need to remember that animation sequences/poses are played when you tell them to. While animating character, you need to look into the details so that the character looks normal.

Now, let's quickly recap what we did in the previous exercise: the skeletal mesh character was a zombie with no animation attached. When we linked the run animation and set it to play, the character immediately seemed like it was running. So, if we want the character to stop running, we can remove the run animation. The character goes back to looking like a zombie that hasn't been animated. If we did this in a game, you would probably think that there is something very wrong with the animation. Zombie->Running->Zombie. Nothing realistic about it.

How can we improve this? We start with an idle pose for the character; an idle pose is one where the character stands at a fixed spot and breathes. Breathing is part of animation too. It makes the character look like it's alive. Next, we set it to play the run animation. To stop this animation, we allow the character to take the idle position again. Not a bad attempt for this iteration. The character doesn't look like a zombie now, but it looks and feels real.

What else can we do to make this even better? Let's use an analogy of someone driving a car normally (not a race car driver). When moving from the start position, you accelerate from a speed of 0 up to a comfortable cruising speed. When you want to stop, you reduce the cruising speed by stepping on the brakes and then gradually go back to 0 (to avoid a stopping suddenly and giving your passengers the unpleasant experience of being thrown forward). Similarly, we can use this to help us design our character's transition from a stationary position. We will use a tool called **Blend Animation** to create this transition so that we can make the movement of the character a little more realistic.

Blend Animation, as the name suggests, blends various types of animation using variables. It can be a simple one-dimensional relationship where we use speed as an axis to blend the animations or a two-dimensional relationship where we use both speed and direction to blend animations. Unreal Engine's Blend Animation tool is capable of setting up the blending of animations in different ways.

## Tutorial – creating a Blend Animation

In this example, we will use speed as the parameter to blend the animation. Let's quickly cover the thought process here first before listing the steps to follow in the Unreal Editor to achieve this. This would help in your understanding of how this process works instead of simply following the process to make something happen.

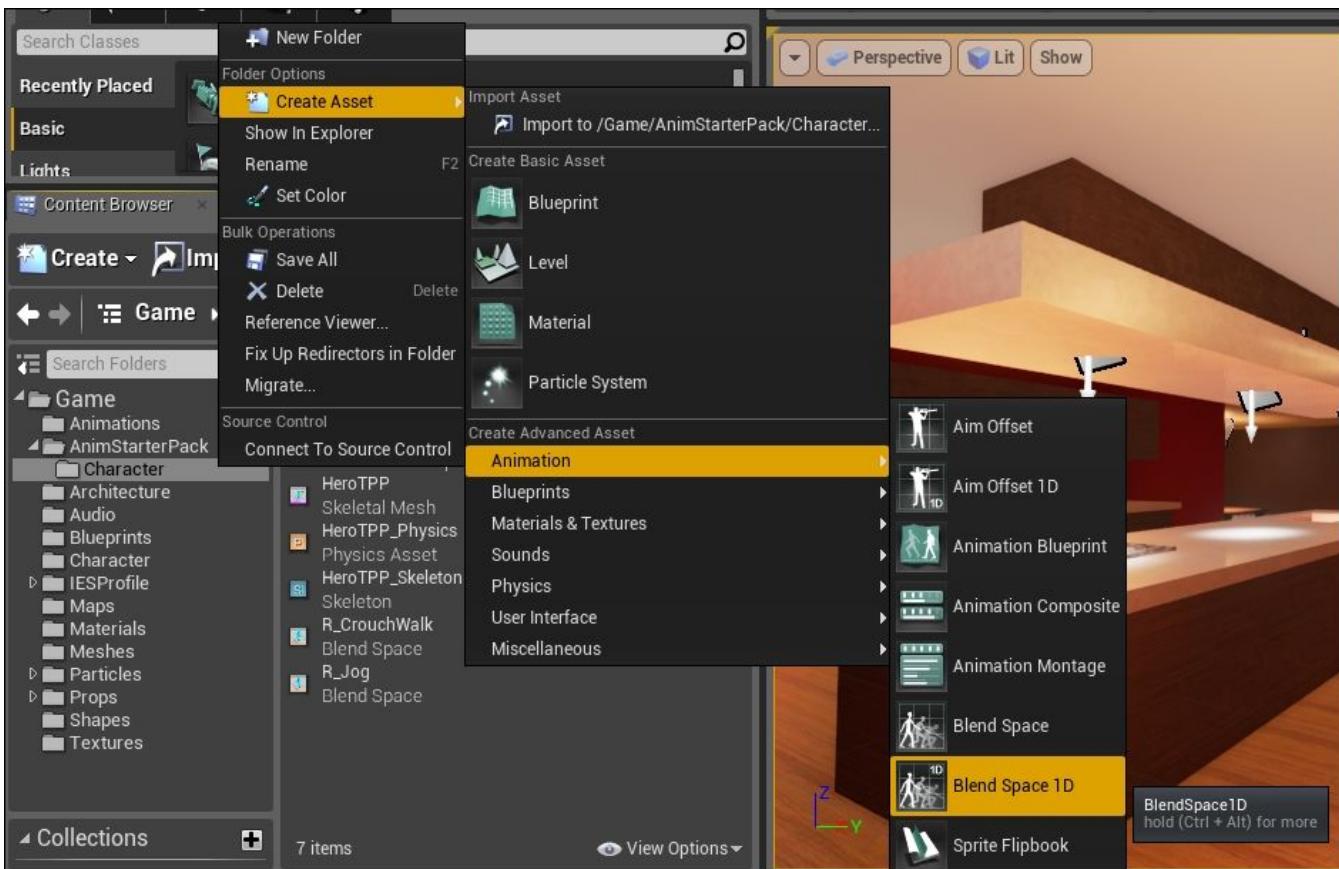
At speed = 0, we assign the idle pose. As the speed increases, we should switch the animation from an idle to a walking animation. As the speed increases even more, the animation switches from walking to jogging, and then running. Here's an illustration of how the blend would look:



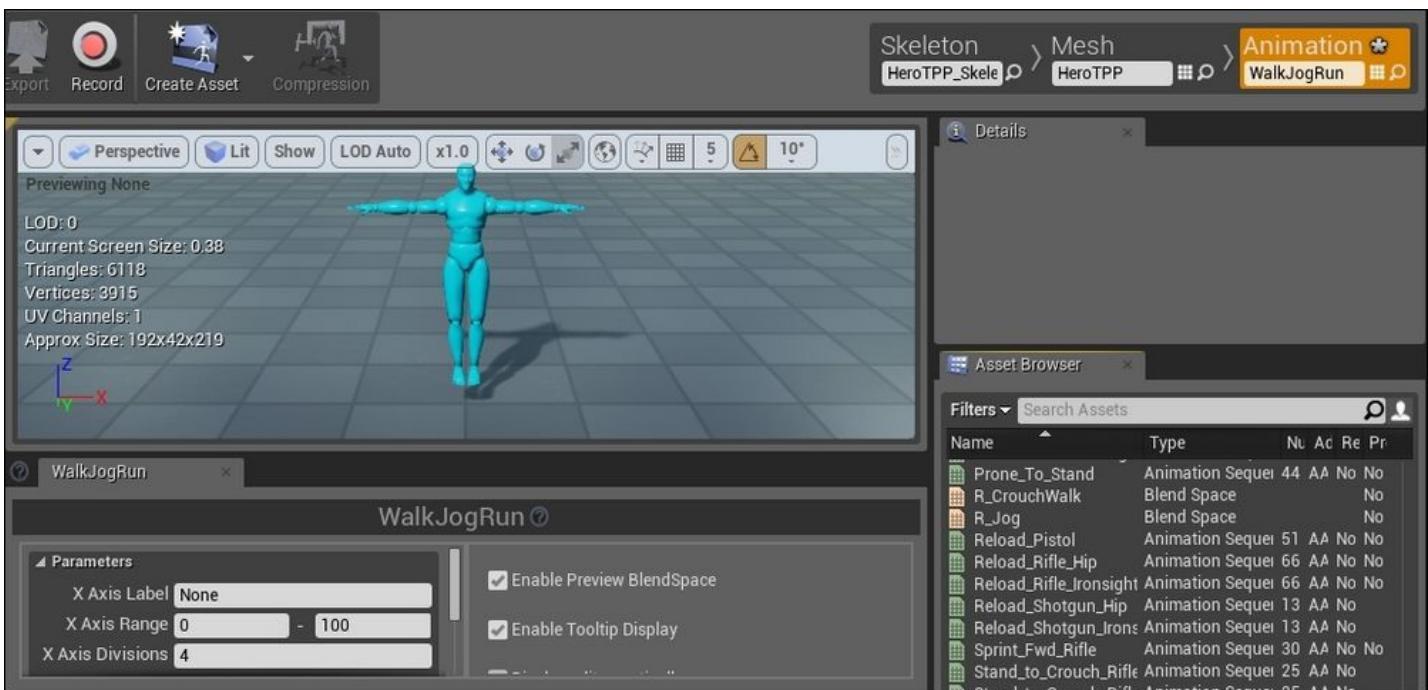
Next, let's identify which animation sequences we have in the animation pack and would be suitable for each of the stages:

- Idle\_Rifle\_Hip
- Walk\_Fwd\_Rifle\_Ironsights
- Jog\_Fwd\_Rifle
- Sprint\_Fwd\_Rifle

To create a simple 1D Blend Space, we can right-click on the **Character** folder, and go to **Create Asset | Animation | Blend Space 1D**. Alternatively, you can select the **Character** folder in **Content Browser**, click on the **Create** button at the top, go to **Animation**, and then **Blend Space 1D**.



Select **HeroTPP\_Skeleton**; clicking on this creates a new Blend Space 1D. Rename **newblendspace1d** to **WalkJogRun**. Double-click on the newly created **WalkJogRun** to open the editor. This will propel you straight to the **Animation** tab of the editor. Notice that this part is highlighted in the following screenshot. In the **SkeletonMesh** field, we have **HeroTPP\_Skeleton**, which was what we selected when creating the blend space earlier.



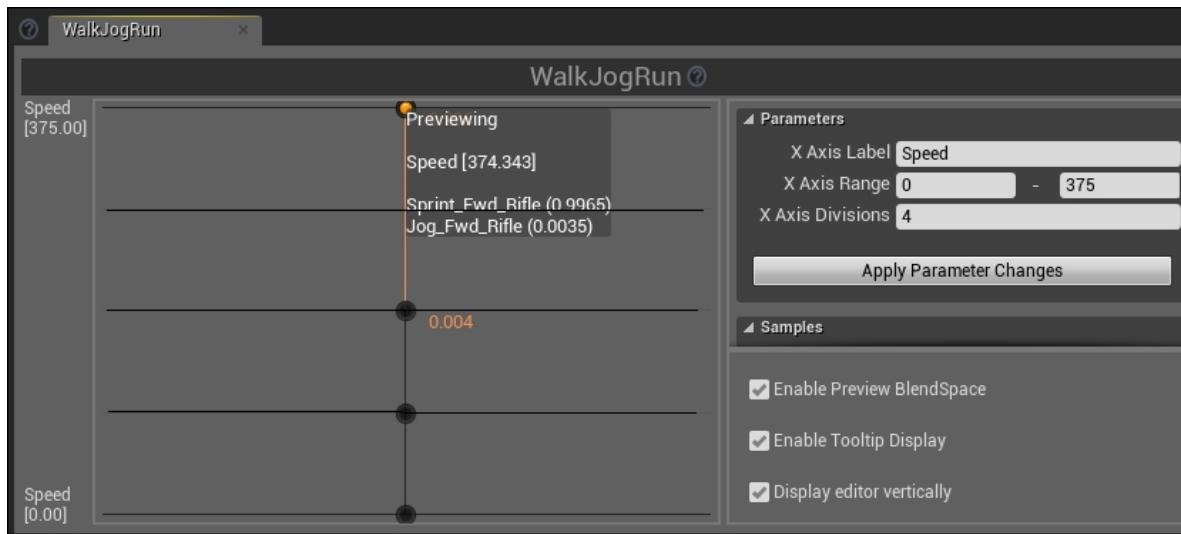
In the **Animation** editor, you have access to **Asset Browser** (which is, by default, in the bottom right-hand side of the screen). Clicking on the animation assets will allow you to preview how the animation looks.

Let's first set the **X Axis Label** to **Speed**. **X Axis Range** is from 0 to 375. Leave **X Axis Divisions** as 4.

The number of divisions creates segments in the speed graph that we have. Based on what we selected earlier for the Idle, Walk, Jog, and Run states, find the animation using **Asset Browser**, click and drop the animation into the **WalkJogRun** tab into the appropriate sections, as shown in the following screenshot:

**Idle\_Rifle\_Hip** is at speed = 0. Set **Walk\_Fwd\_Rifle\_Ironsight** in the first division line. When you drag an animation into the graph, it creates a node and snaps

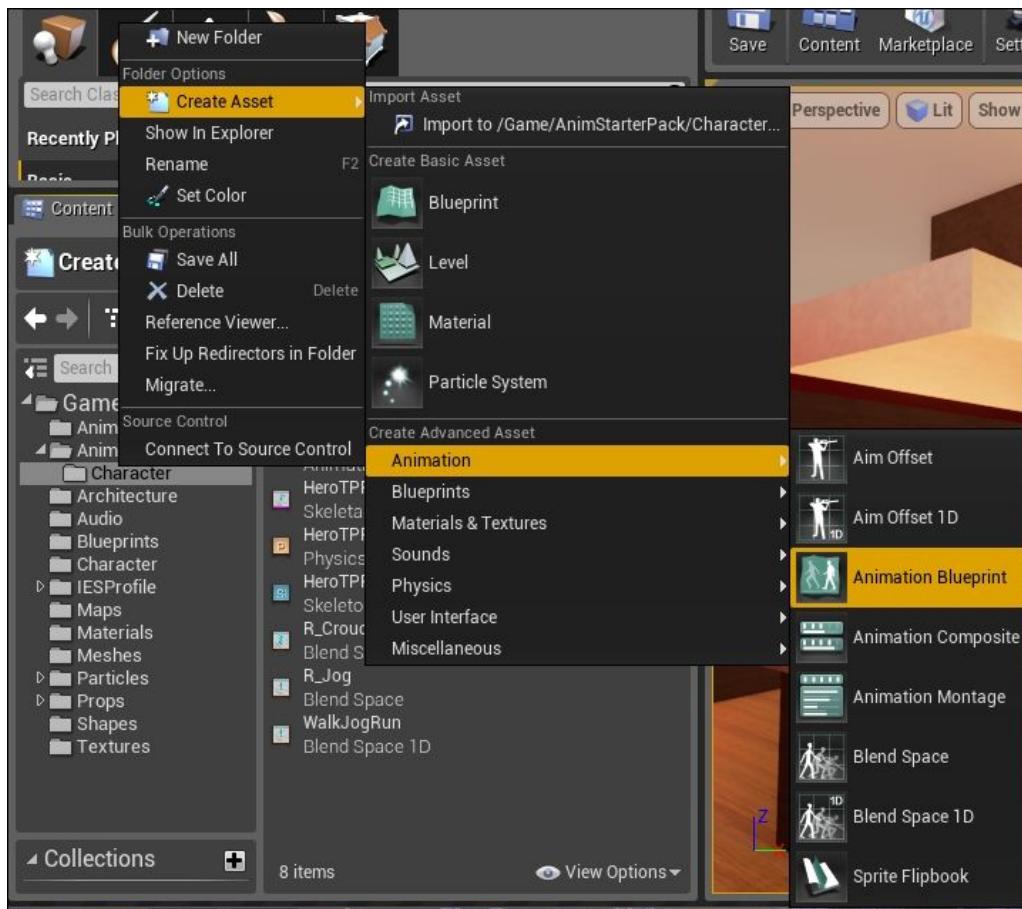
at one of the division lines. Set **Jog\_Fwd\_Rifle** in the second division line and set **Sprint\_Fwd\_Rifle** at speed = 375. To preview how the animation blends, move the mouse over the graph along the vertical axis.



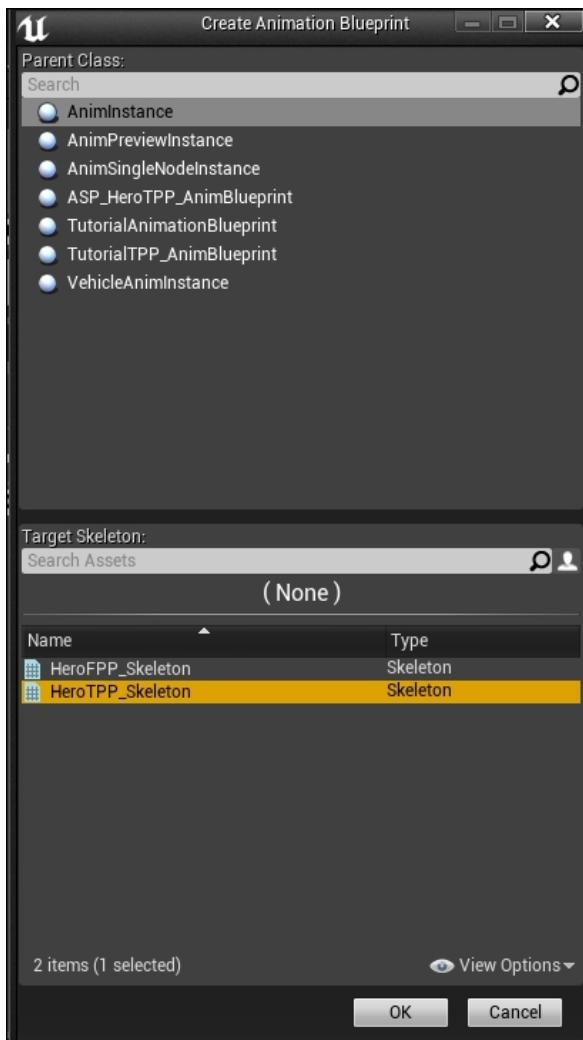
## Tutorial – setting up the Animation Blueprint to use a Blend Animation

Now we have created a Blend Animation that uses speed as a parameter. How do we make an NPC change speed and then link this animation to it so that as the speed changes and the animation that is played also changes?

For a simple implementation of getting the speed and animation to change, we will set up the Animation Blueprint. Go to **Content Browser**. Navigate to **Animation | Character**; then, navigate and click on **Create Asset | Animation | Animation Blueprint**:



Upon selecting **Animation Blueprint**, the editor will prompt you about the base class that you want the Animation Blueprint to be created in. This screenshot shows the options that are available for selection:

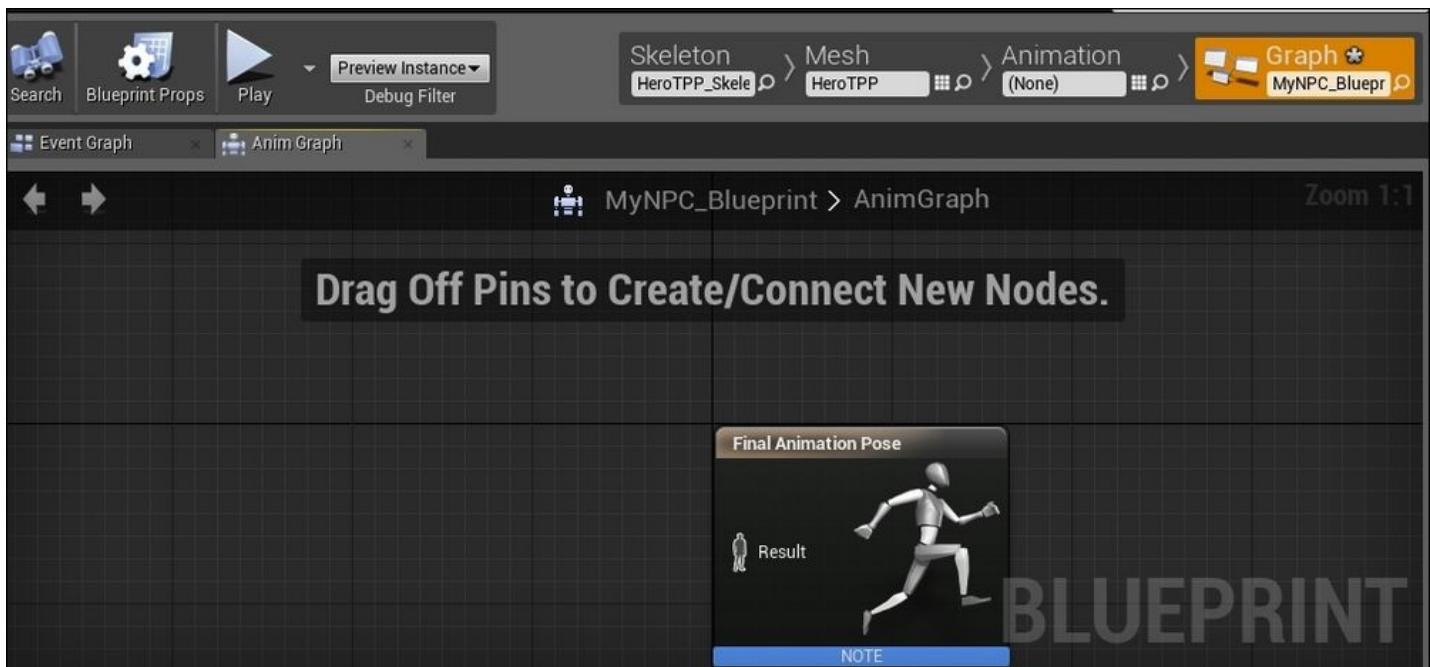


In this example, we will pick the most basic generic class, `AnimInstance`, to build our Animation Blueprint in. Select **HeroTPP\_Skeleton** as the target skeletal mesh for this blueprint. Name this Animation Blueprint `MyNPC_Blueprint`.

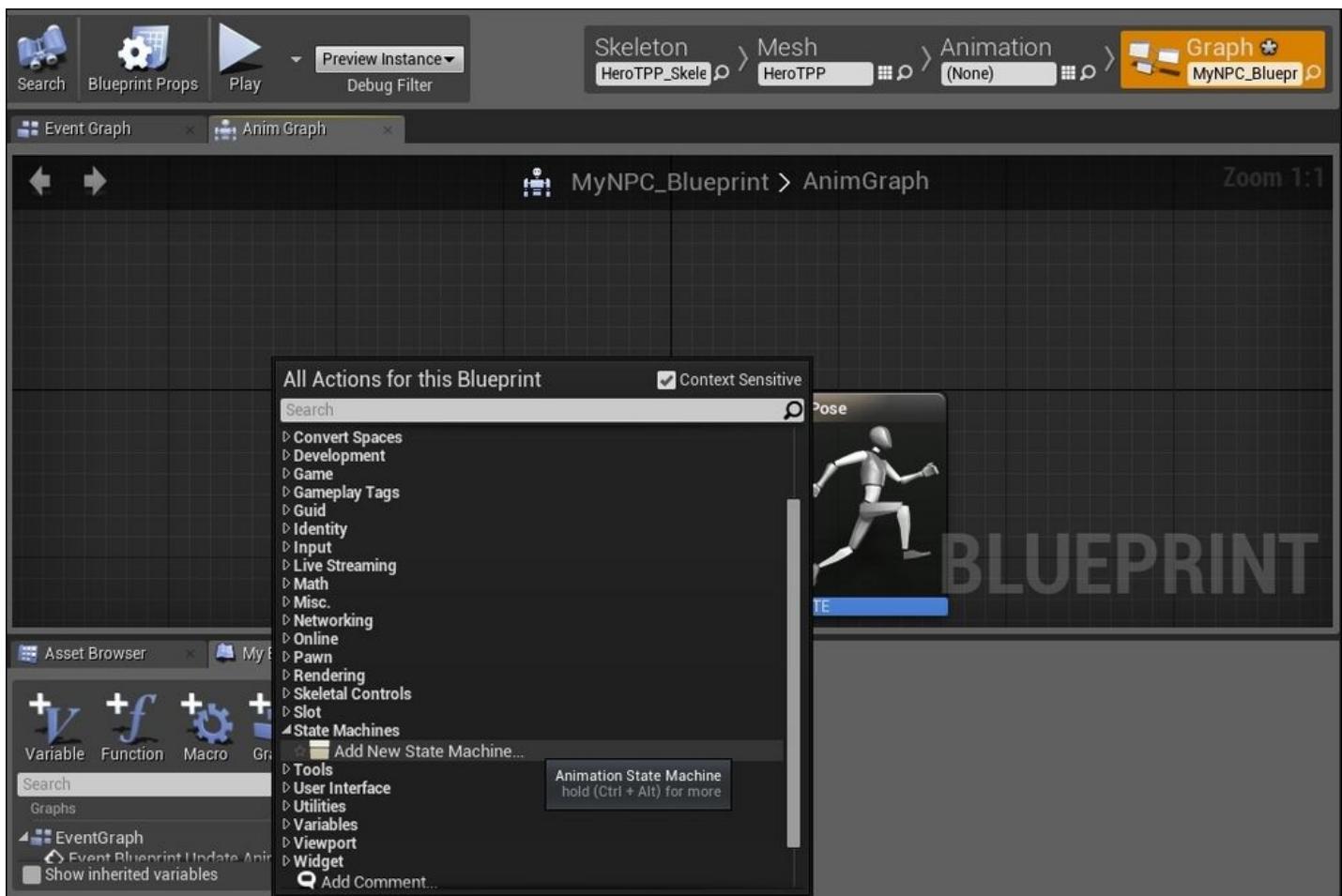
To check whether you have selected the correct target skeletal mesh, look in the **Skeleton** tab in the **Blueprint** window, as shown in the following screenshot. You should see **HeroTPP\_Skeleton** in the box. The screenshot also shows the **Graph** tab that's been selected with the empty default AnimGraph showing. We will proceed through this exercise with the **Graph** tab selected, unless specified otherwise.

#### AnimGraph

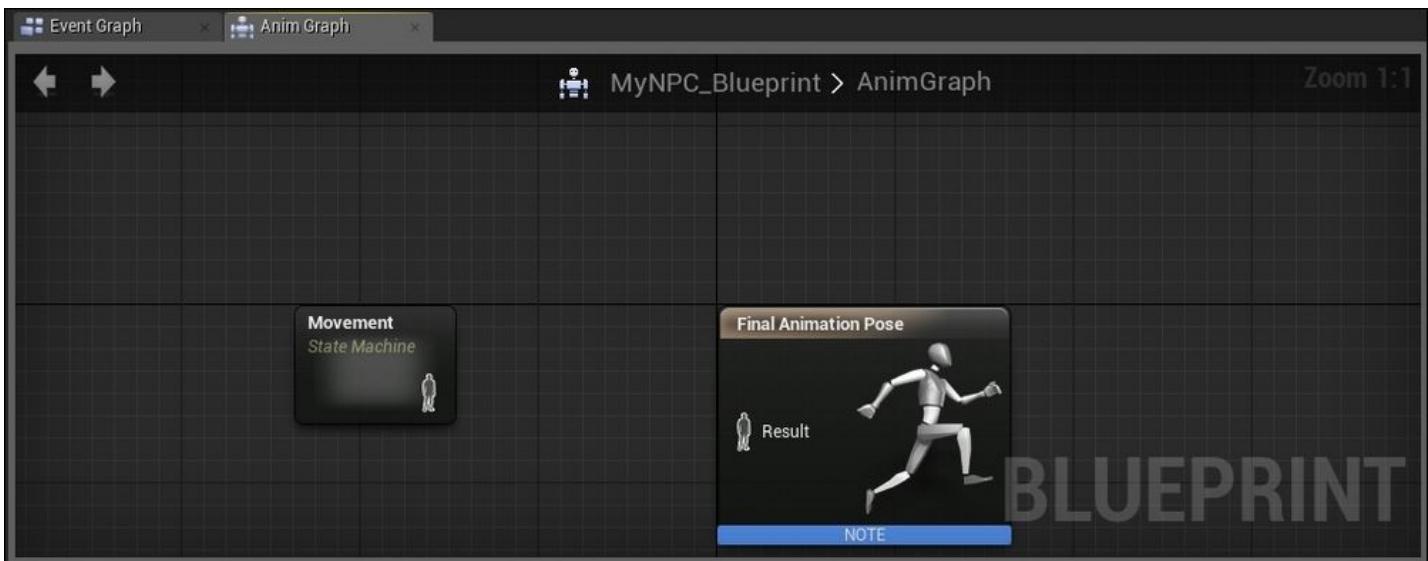
This screenshot shows the default blank AnimGraph. **Final Animation Pose** will receive the output of the skeletal mesh that's been specified:



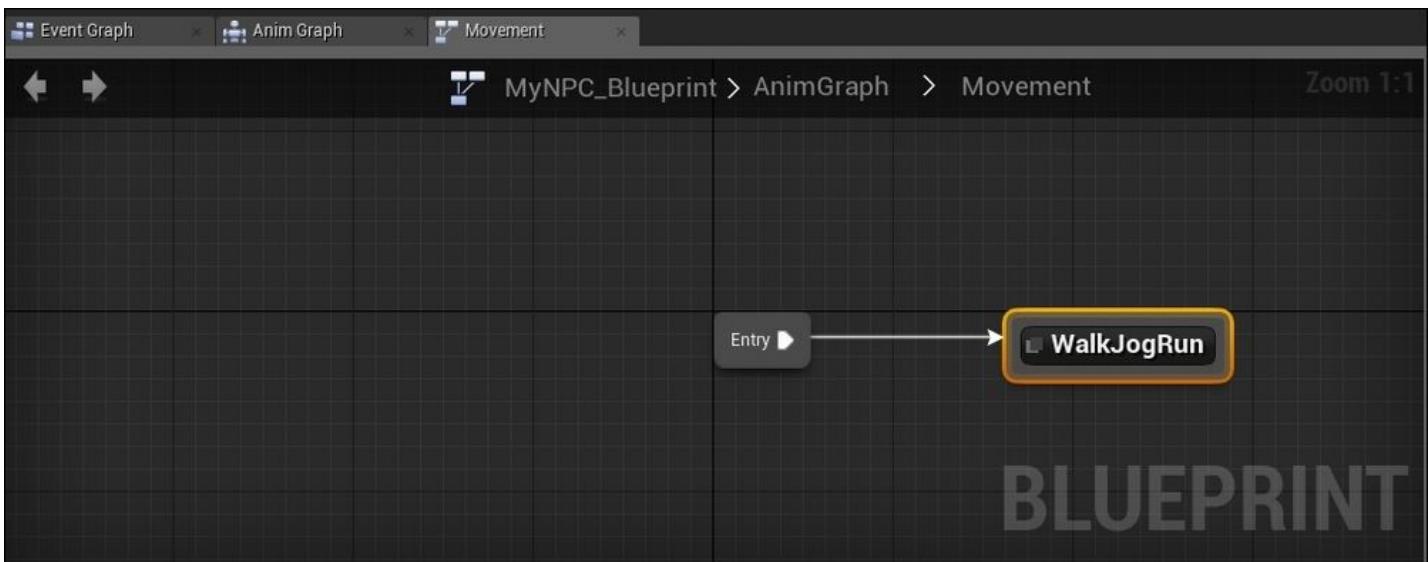
First, we want to add a state machine by right-clicking within the AnimGraph and navigating to **State Machines | Add New State Machine...**, as shown in the following screenshot:



Rename the newly created state machine **Movement**:

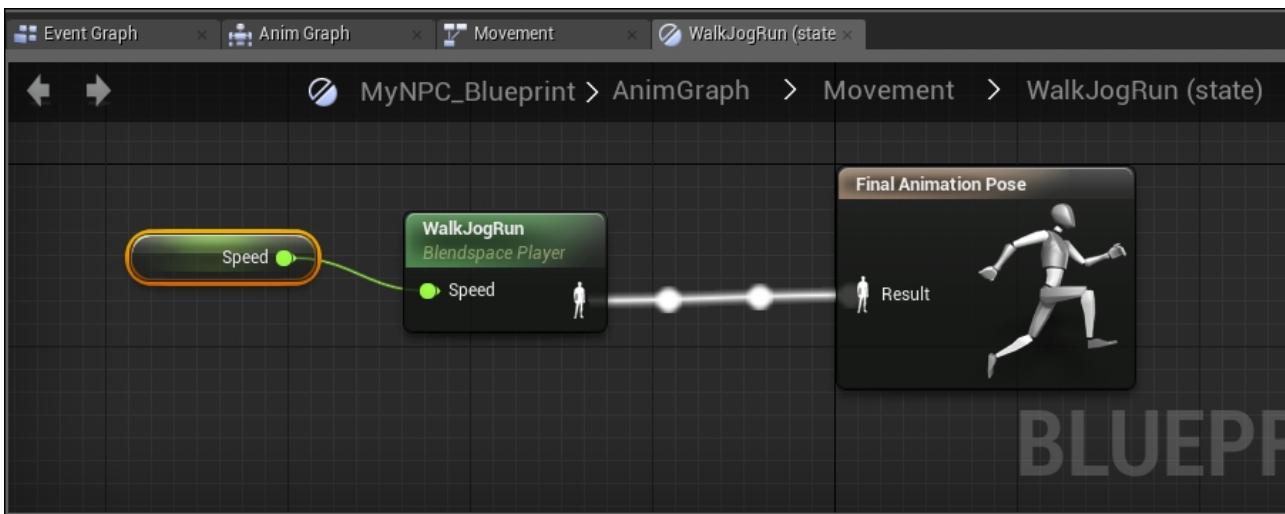


Double-click on **Movement**. Create a new state named **WalkJogRun**:

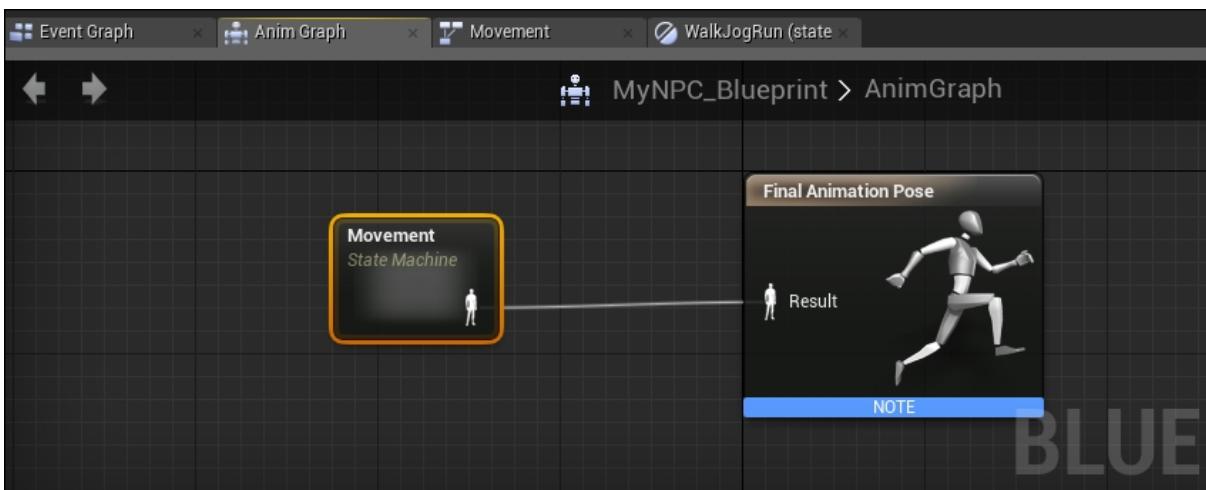


Double-click on the newly created **WalkJogRun** state to modify the state in a new tab. Go to the **Asset Browser** tab, look for **WalkJogRun** blendspace, which we created in the previous exercise, and click and drag it into the editor. Link **WalkJogRun** blendspace to the final animation, as shown in the following screenshot. Notice that speed = 0.00 is specified in the blendspace node; this was the variable that we defined to control the change of the animation when we created blendspace in the earlier exercise.

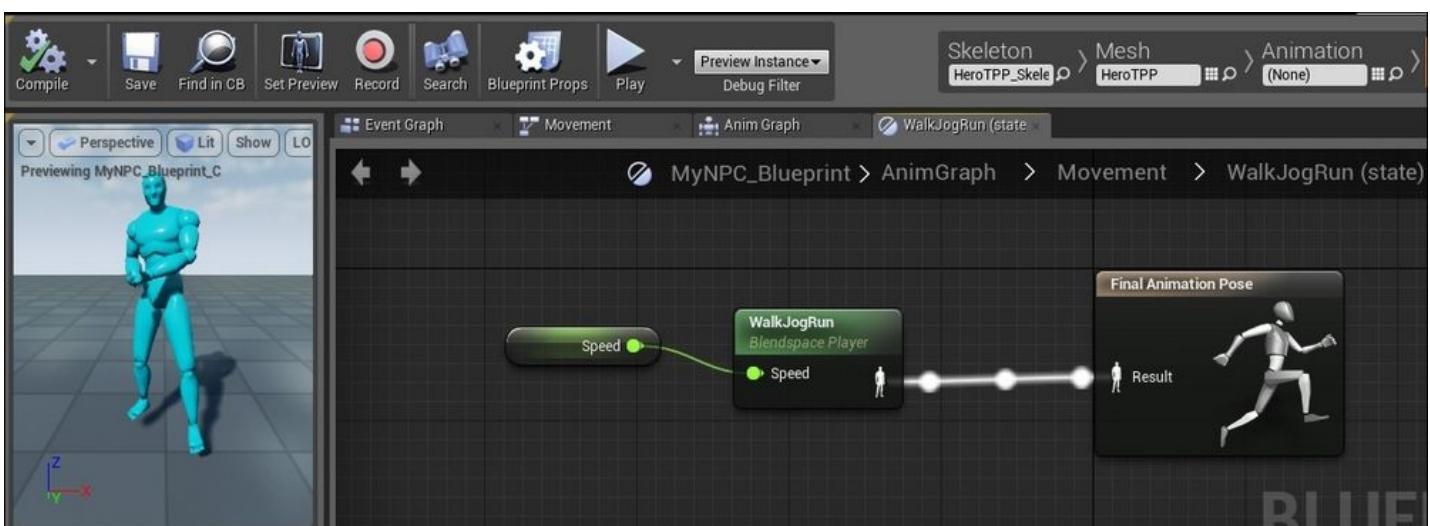
Next, we need to create a variable so that we can pass in a value to the **WalkJogRun** blendspace speed variable. To do so, we need to click and drag the green dot next to the **Speed** on the blendspace node to open up a contextual menu, look for **Promote to Variable**, and then click on it. This promotes speed in the blendspace node to a float variable, which we would set to control the speed and type of animation that will be played. Rename this new variable **Speed**. The following screenshot shows how we have created and connected a **Speed** variable to **WalkJogRun** blendspace, which is linked to **Final Animation Pose**:



Now, go back to link **Movement** to **Final Animation Pose** :



Now, the entire AnimGraph is linked up. Click on **Compile**, and you would see the preview of the character model updated, as shown in the following screenshot. The white moving dots show how data flows through the system. The speed is 0 here.



We can also use this tab to see live preview as we change the value to **Speed**. The following screenshot shows you when speed is 50. The character model assumes a walking pose.



Through AnimGraph, we were able to set up **Speed** as a variable and link this variable to **WalkJogRun** blendspace, which, in turn, controls what animation to play at which speed. We need to now think about how to provide some logic to determine how the speed of the NPC changes.

### EventGraph

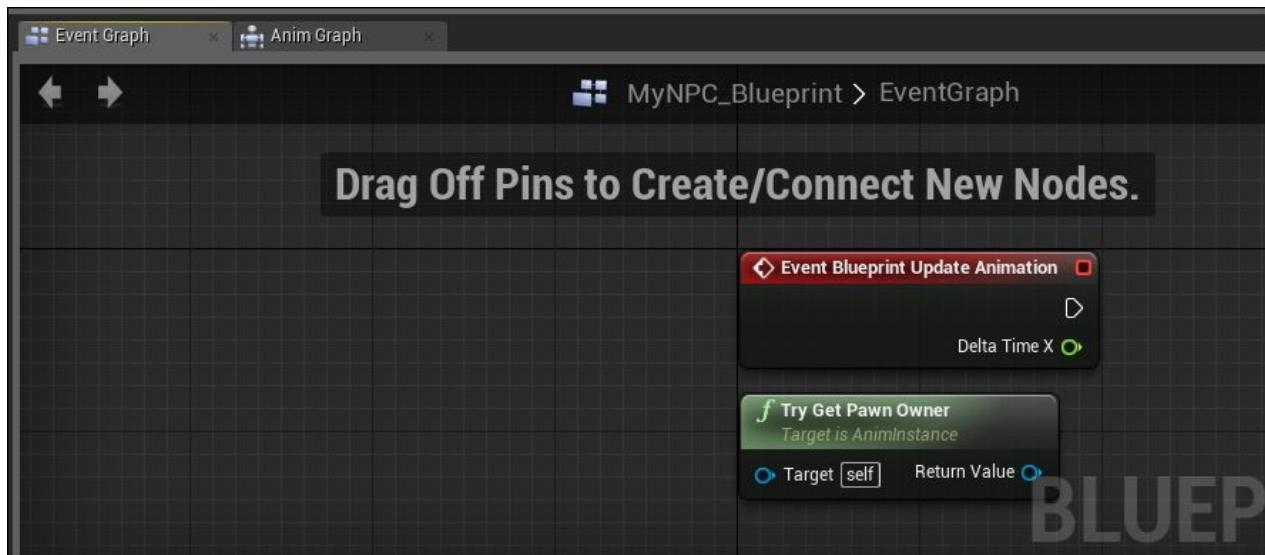
EventGraph is used to program logic into the Blueprint.

In this example, we will use EventGraph to create logic to change the speed values that will, in turn, affect the NPC's animation control.

To create a more complex intelligent decision-making process, which is termed as AI, we will need to use a set of AI-related nodes in EventGraph. We will learn more about creating AI in the next section.

The following screenshot shows the default new **EventGraph** tab in the Animation Blueprint.

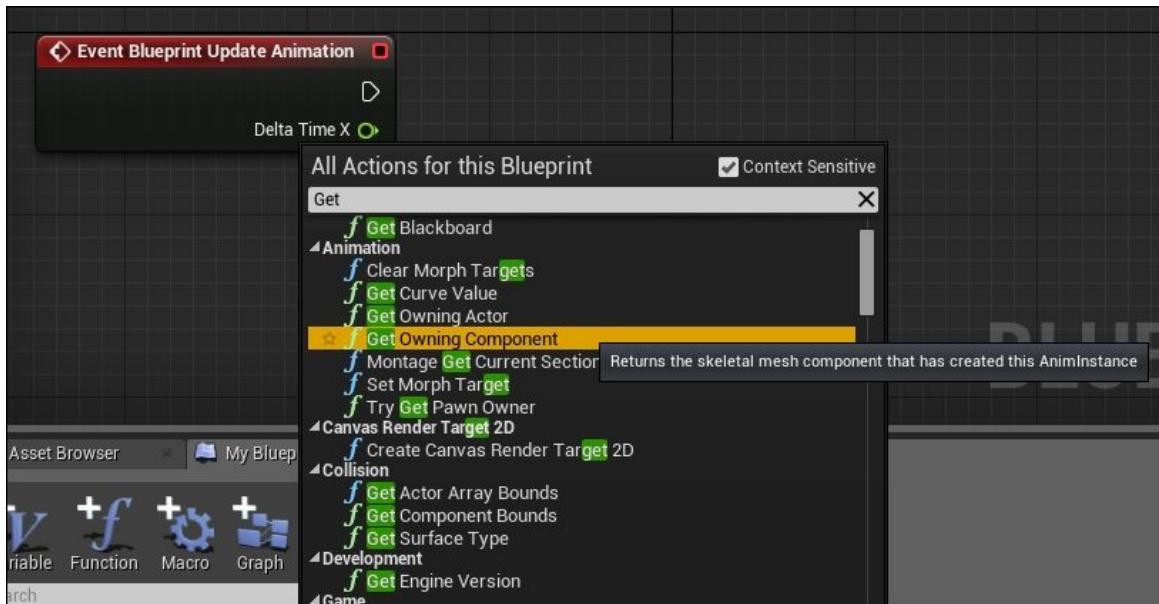
The **Event Blueprint Update Animation** node can be thought of as the source that sends a pulse through the EventGraph network. As this pulse travels through the network, it goes through a series of questions that you design to determine which animation is played.



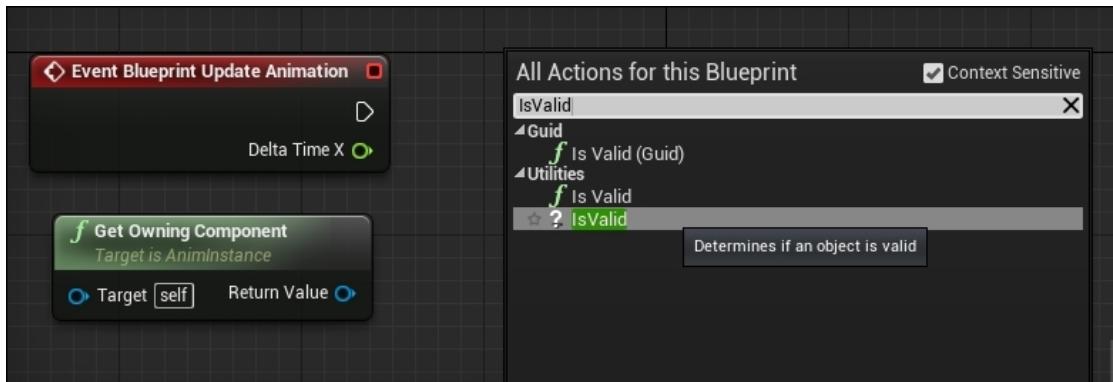
**Try Get Pawn Owner** is to get the owner that Animation Blueprint is assigned to. This is simply used in combination with another node, **IsValid**, to ensure that we have a valid owner before setting values to change the animation.

To make **MyNPC\_Blueprint** work for the **Hero TPP** mesh that we have in the level, we will need to first delete the **Try Get Pawn Owner** node and replace it with **Get Owning Component**. Right-click on the EventGraph and type **Get**. In the contextual menu that is opened, scroll down to find **Get Owning Component**.

This screenshot shows where the **Get Owning Component** node is:



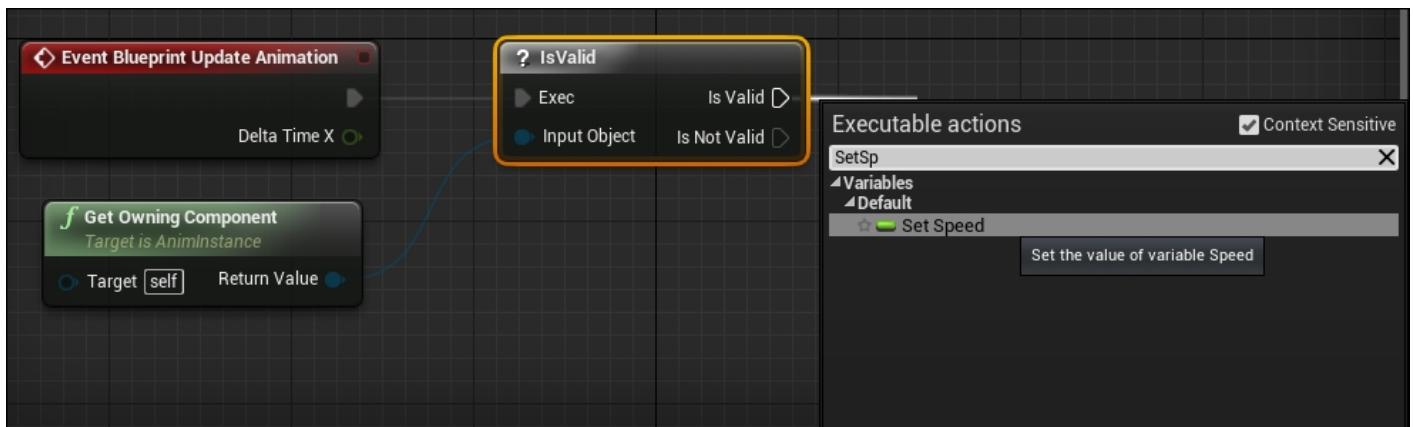
In the same way, right-click in the editor and type `IsValid` to look for the node. This screenshot shows where to get the **IsValid** node:



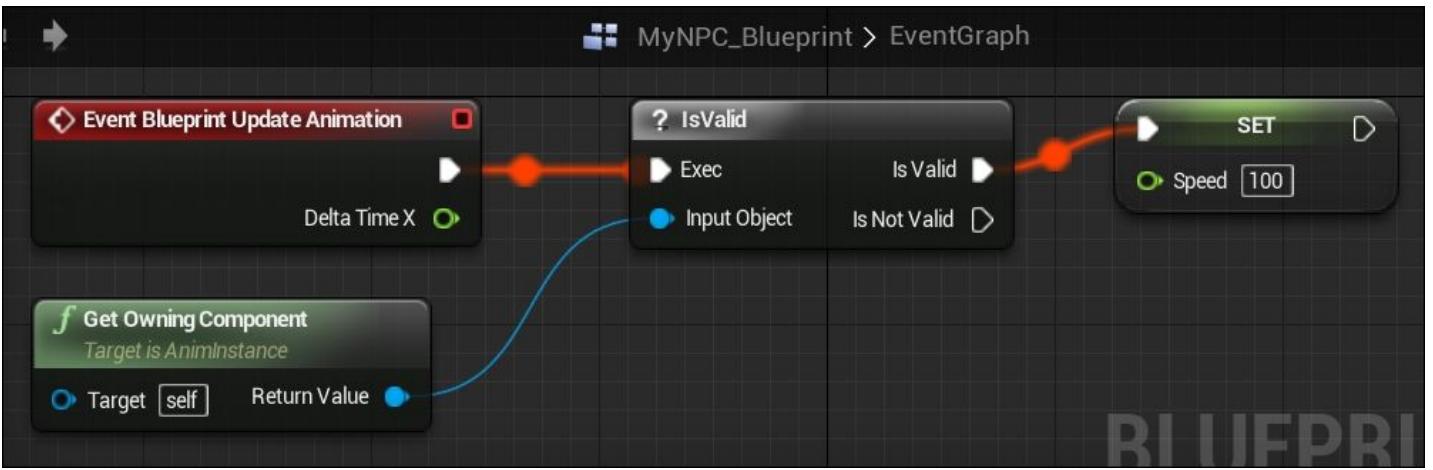
Now, link the triangular output from **Event Blueprint Update Animation** to the **Exec** input of the **IsValid** node (which is also a triangular input). Link **Return Value** (this has a blue circle next to it) output from **Get Owning Component** to **Input Object** (this has a blue circle next to it) of the **IsValid** node. The following screenshot shows the linkage of the three nodes.

The explanation for this is that at every tick, we need to check whether the target skeleton mesh is valid.

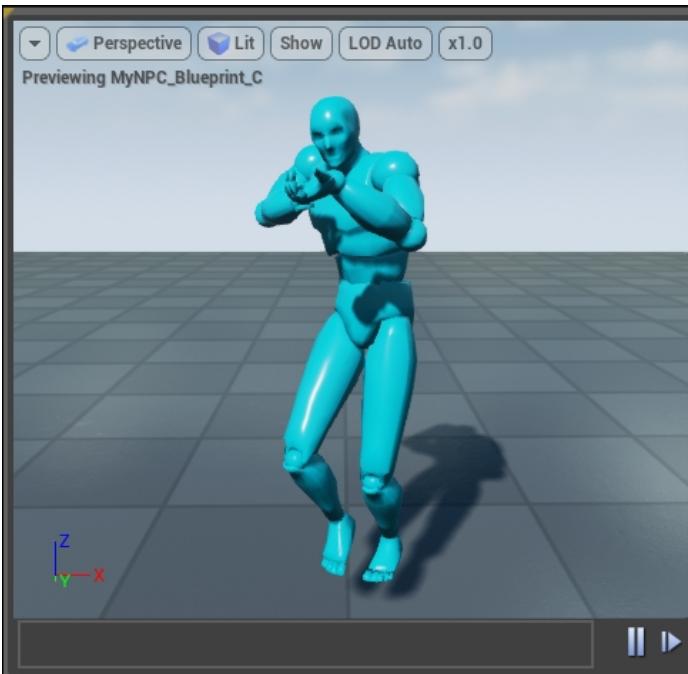
For now, let's simply set the speed of the NPC to 100 if the target skeleton mesh is valid. So, right-click on the EventGraph area, and type `SetSpeed` to filter the options. Click and select **Set Speed**, as shown in this screenshot:



Link the **Is Valid** output of the **IsValid** node to the input (this has a triangular symbol) of the **SET Speed** node. Then, click on the box next to **Speed** and type **100** to set the speed:

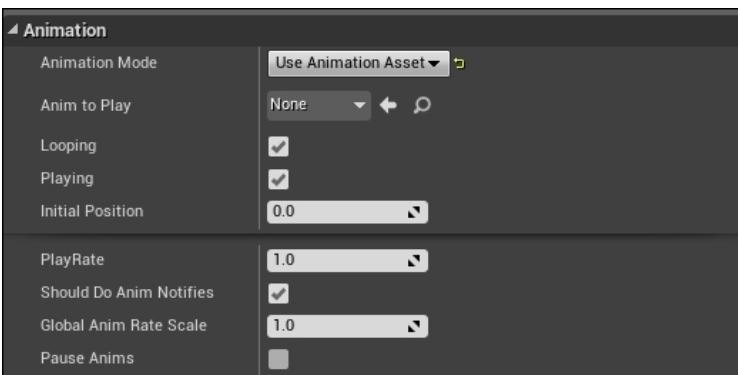


Save and recompile now to see how the preview model changes. The following screenshot shows the model playing the walk animation when speed is set to 100:

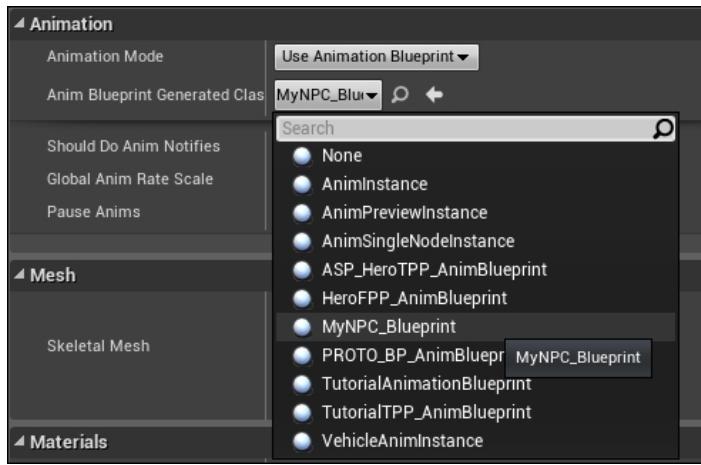


Now, Animation Blueprint is ready for use in the game level. We need to assign this Animation Blueprint to a character in the game. Save and close the Animation Blueprint editor to go back to the main editor.

To assign the Blueprint to the skeleton mesh, we will click on the existing **HeroTPP** to display the details panel. Focus on the animation part of the panel; the following screenshot shows the original setting that I have when there is no animation sequence linked to the skeleton mesh and it does not use an Animation Blueprint. Set **Animation Mode** to **Use Animation Asset** and **Anim to Play** to **None**:



To use **MyNPC\_Blueprint** for this skeleton mesh, set **Animation Mode** to **Use Animation Blueprint**. Select **MyNPC\_Blueprint** for **Anim Blueprint Generated Class**:



Now, compile and run the game; you would see the NPC walking on the same spot with the speed set as 100.

# Artificial intelligence

AI is a decision-making process that adds NPCs in a game. AI is a programmable decision-making process for NPCs to govern their responses and behaviors in a game. A game character that is not controlled by a human player has no form of intelligence, and when these characters need to have a higher form of decision-making process, we apply AI to them.

AI in games has progressed tremendously over the years and NPCs can be programmed to behave in a certain way, sometimes, with some form of randomness, making it almost unpredictable so that players do not have a simple, straightforward strategy to win the level.

The decision-making process, which is also the logic of the NPCs, is stored in a data structure known as a Behavior Tree. We will first learn how to design a simple Behavior Tree then learn how to implement this in Unreal Engine 4.

## Understanding a Behavior Tree

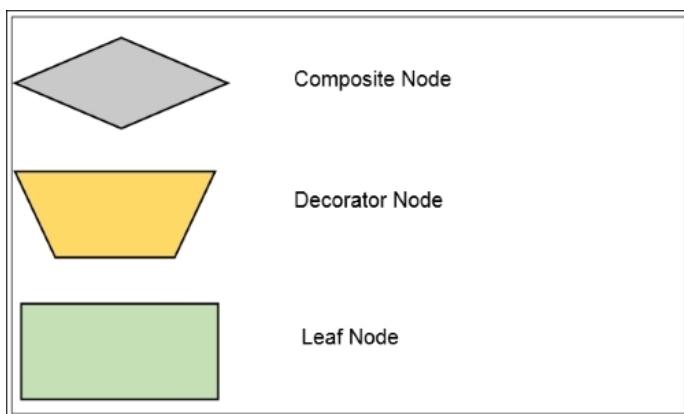
Learning how to design a good decision-making tree is very important. This is the foundation on which programmers or scripters rely to create the behavior of a character in a game. The Behavior Tree is the equivalent of a construction blueprint for architects who design your house.

A Behavior Tree has roots that branch out into layers of child nodes, which are ordered from left to right (this means that you always start from the left-most node when traversing the child nodes) that describe the decision-making process. The nodes that make up the Behavior Tree mainly fall into three categories: Composite, Decorator, or Leaf. Once you are familiar with a couple of the common types of nodes in each of the three categories, you would be ready to create your own complex behaviors:

	Composite	Decorator	Leaf
Children nodes	Having one or more children are possible.	This can only have a single child node.	This cannot have any children at all.
Function	Children nodes are processed, depending on the particular type of composite node.	This either transforms results from a child node's status, terminates the child, or repeats the processing of the child, depending on the particular type of Decorator.	This executes specific game actions/tasks or tests.
Node examples	The <b>Sequence</b> node processes the children nodes from the left-most child in sequence, collects results from each child, and passes the overall success or failure result over to the parent (note that even when only one child fails and the rest succeed, the overall result is failure). This can be thought of as an <b>AND</b> node.	The <b>Inverter</b> node converts a success to a failure and pass this inverted result back to the parent. It works vice versa as well.	The <b>Shoot Once</b> leaf node shows that the NPC would shoot once and return a success or failure, depending on the result.

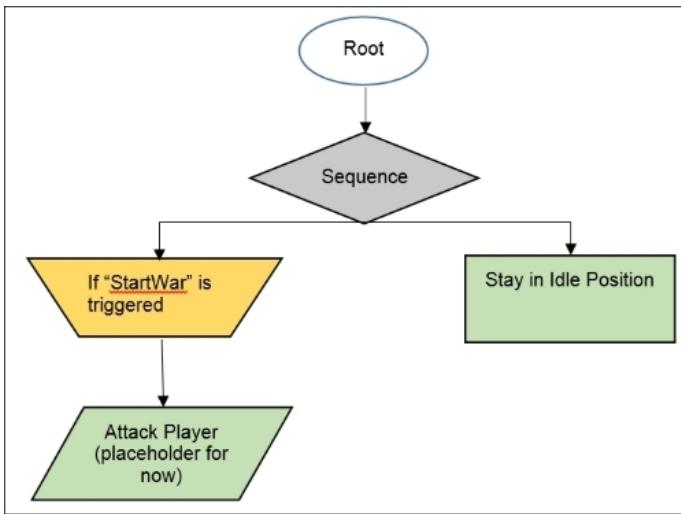
## Exercise – designing the logic of a Behavior Tree

This is a simple walkthrough of how a Behavior Tree can be constructed. The following legend will help you identify the different components of a Behavior Tree:

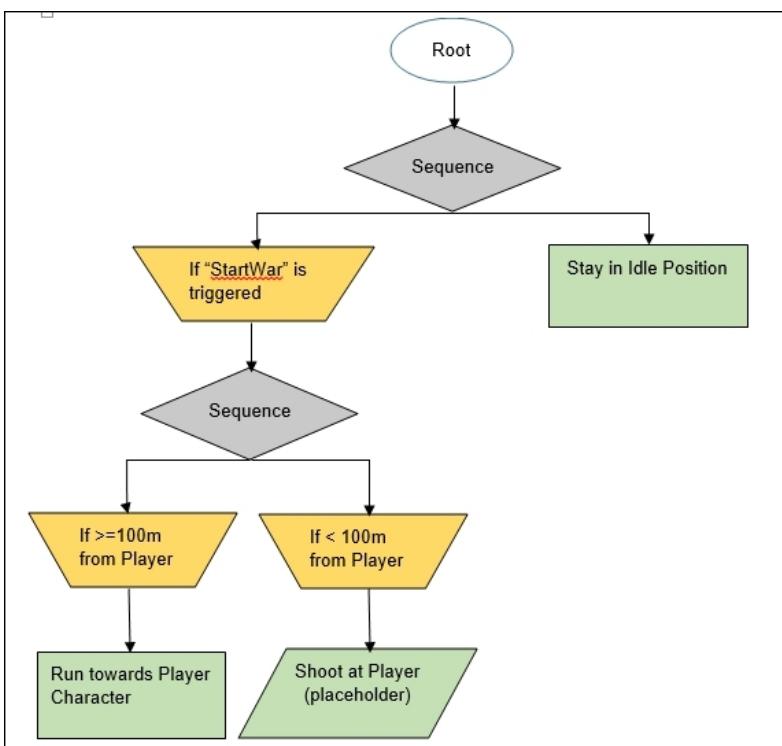


## Example – creating a simple Behavior Tree

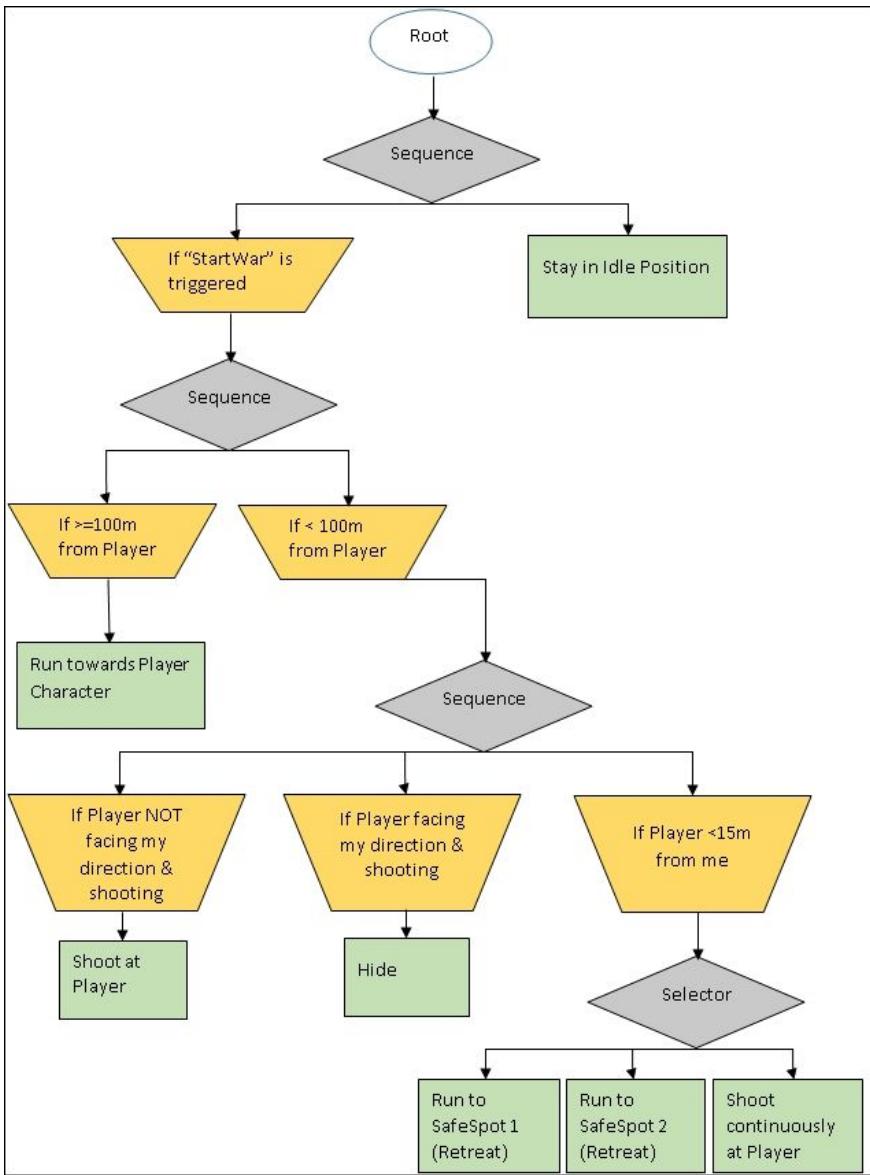
The following figure shows a simple response for an enemy NPC. The enemy will only start attacking when the war starts.



The following figure has been expanded on the earlier Behavior Tree. It gives a more detailed description of how the enemy NPC should approach the target. The NPC will run towards the target (the player character in this case), and if it is close enough, it starts shooting the player.



Next, we set more behaviors that show how the NPC will shoot the player. We give the enemy NPC a little intelligence: hide if someone is shooting at it and start shooting if no one is shooting at it; if the player starts moving in toward it, the NPC starts moving backward to a better spot or goes for a death match (it shoots the player at close range).



## How to implement a Behavior Tree in Unreal Engine 4

The Unreal Editor allows complex Behavior Trees to be designed using the visual scripting Blueprints together with several AI components.

There is also an option in Unreal Engine 4 where very complex AI behaviors can be programmed in the conventional way or in combination with Blueprint visual scripting.

The nodes for BT in UE4 are broadly divided into five categories. Just to recap, we have already learned a little about the first four in the previous section; **Service** is the only new category here:

- **Root** : The starting node for a Behavior Tree and every Behavior Tree has only one root.
- **Composite** : These are the nodes that define the root of a branch and the base rules for how this branch is executed.
- **Decorator** : This is also known as a **conditional** . These attach themselves to another node and make decisions on whether or not a branch in the tree, or even a single node, can be executed.
- **Task** : This is also known as a Leaf in a typical BT. These are the leaves of the tree, that is, the nodes that "do" things.
- **Service** : These are attachments to composite nodes. They are executed at a defined frequency, as long as their branch is being executed. These are often used to make checks and update the **Blackboard** . These take the place of traditional parallel nodes in other Behavior Tree systems.

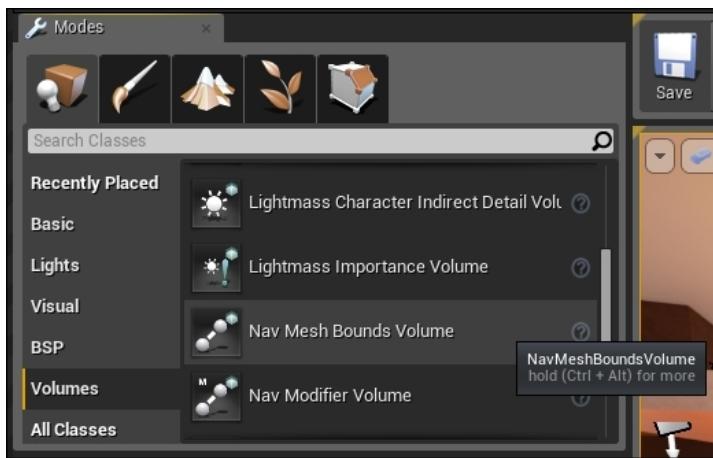
## Navigation Mesh

For AI characters to move around in the game level, we need to specifically tell the AI character which areas in the map are accessible.

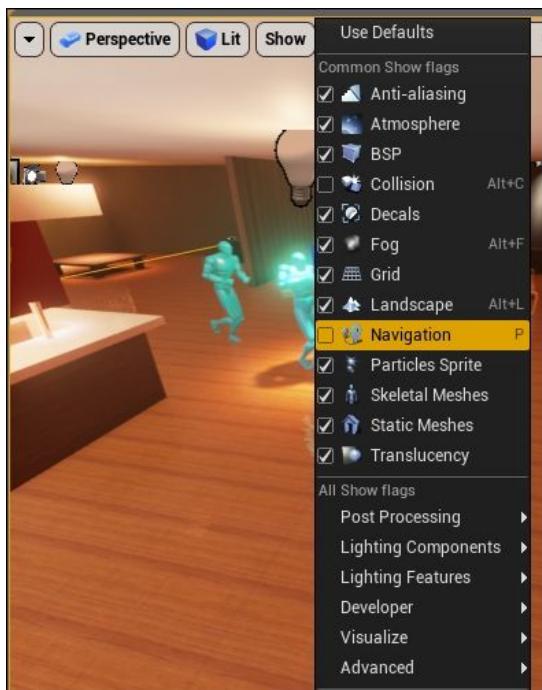
Unreal Engine has implemented a mesh-like component known as **Navigation Mesh** . The Navigation Mesh is pretty much like a block volume; you could scale the size of the mesh to cover a specific area in the game level that an AI character can move around in. This limits the area in which an AI can go and makes the movement of the character more predictable.

### Tutorial – creating a Navigation Mesh

Go to **Modes | Volumes** . Click and drop **Nav Mesh Bounds Volume** into your game level. The following screenshot shows where you can find **Nav Mesh Bounds Volume** in the editor:



If you are unable to see **Nav Mesh Bounds Volume** in your map, go to the **Show** settings within the editor, as shown in the following screenshot. Make sure the checkbox next to **Navigation** is checked:



Scale and move the Navigation Mesh to cover the area of the floor you want the AI character to be able to access. What I have done in the following screenshot is to scale the mesh to fit the floor area which I want my AI character to walk in. Translate the mesh upward and downward to allow it to be slightly above the actual ground mesh. The Navigation Mesh should sort of enclose the ground mesh. This screenshot shows how the mesh looks when it is visible:



## Tutorial – setting up AI logic

Here's an overview of the components that we will create for this tutorial:

- Blueprint AIController (**MyNPC\_AIController**)
- Blueprint Character (**MyNPC\_Character**)
- BlackBoard (**MyNPC\_Brain**)
- Behavior Tree (**MyNPC\_BT**)
- Blueprint Behavior Tree Task (**Task\_PickTargetLocation**)

The important takeaway from this tutorial is to learn how the components are linked up to work together to create logic; we make use of this logic to control the behavior of the NPC.

In terms of file structure in **Content Browser** for these different file types, you can group the different components into different folders. For this example, since we are only creating one NPC character with logic, I will put all these components into a single folder for simplicity. I created **MyFolder** under the main directory for this purpose.

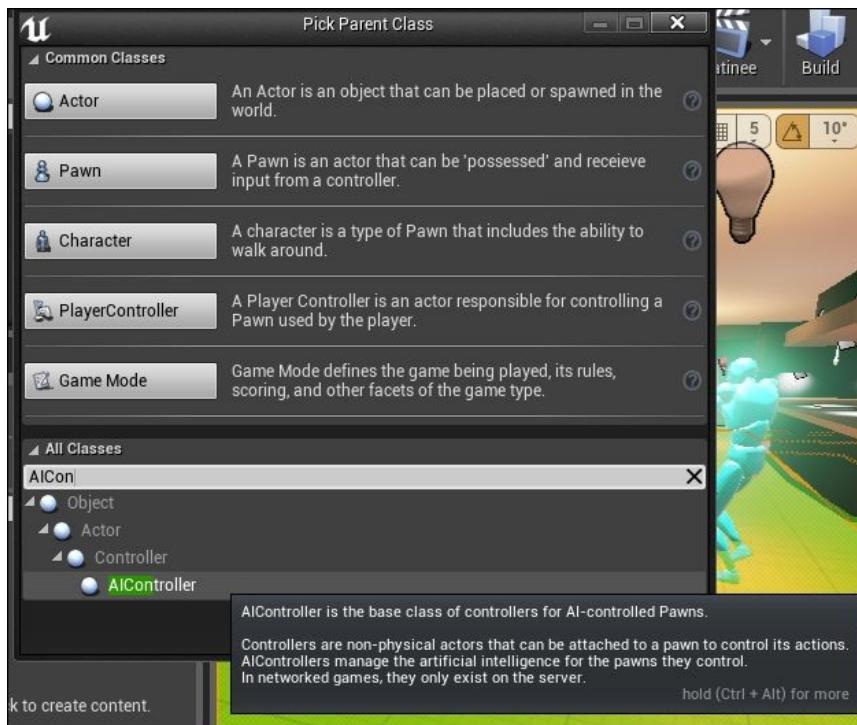
We start creating the AI logic of our NPC starting with AIController and Character. The Character Blueprint is the object that contains the link to the mesh, and we will drag and drop this Character Blueprint into the level map after we make some initial configurations. The AIController is the component that gives the NPC character its logic.

We will discuss the rest of the other three components as we go along.

### Creating the Blueprint AIController

Go to **Create | Blueprint**. Type in **AIController** into the textbox to filter by class, as shown in the following screenshot. Select **AIController** as the parent class.

Rename this AIController Blueprint as **MyNPC\_AIController**:



We will come back to configure this later.

## Creating the Blueprint character

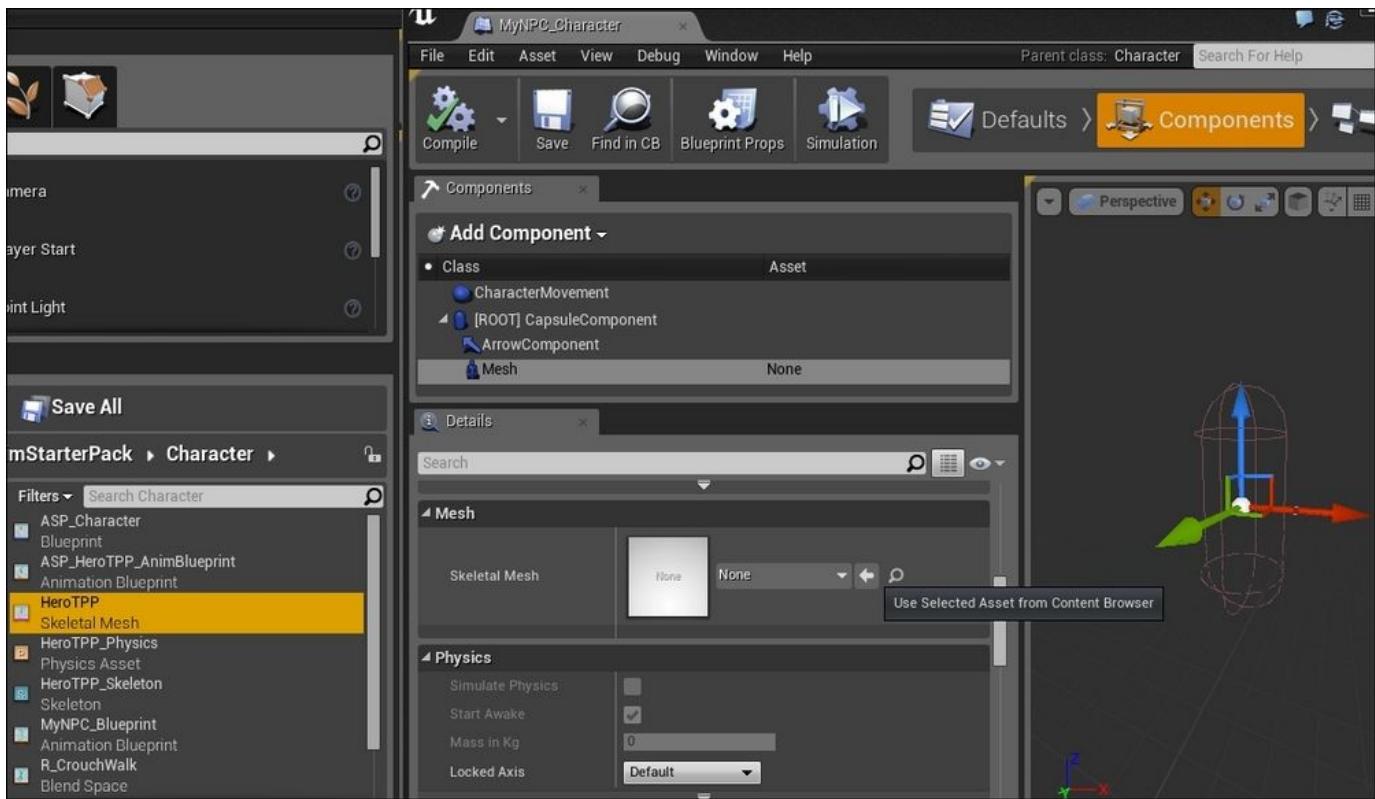
Go to **Create | Blueprint**, and type in `Character` in the textbox to filter by class. Select **Character** as the parent class for the Blueprint, as shown in the following screenshot. Rename this Blueprint as `MyNPC_Character`.



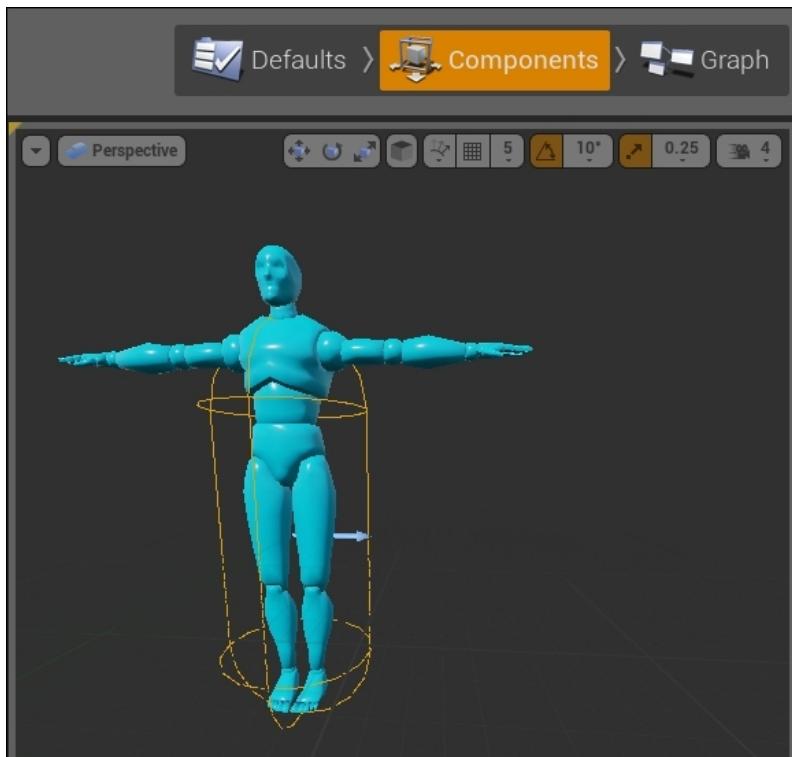
## Adding and configuring Mesh to a Character Blueprint

Double-click on `MyNPC_Character` in **Content Browser** to open the Character Blueprint editor. Go to the **Components** tab.

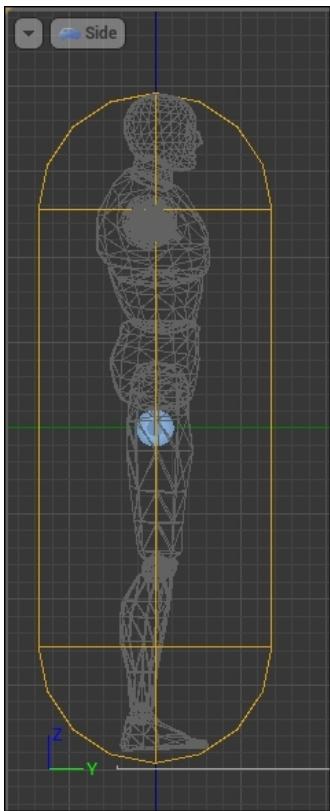
In the **Perspective** space view, you will see an empty wireframe-capsule-shaped object, as shown in the following screenshot. In the **Details** panel in the Blueprint editor, scroll to the **Mesh** section, and we will add a mesh to this Blueprint by selecting an existing mesh we have. You can go to **Content Browser**, select `HeroTPP`, and click on the arrow next to it. Alternatively, you can click on the search button next to the box and find `HeroTPP`:



After selecting **HeroTPP** as the skeletal mesh, you will see the mesh appearing in the wireframe capsule. Notice that the **HeroTPP** skeletal mesh is much larger than the capsule wireframe, as shown in the following screenshot. We want to be able to adjust the size of the wireframe to surround the height and width of the skeletal mesh as closely as possible. This will define the collision volume of the character.

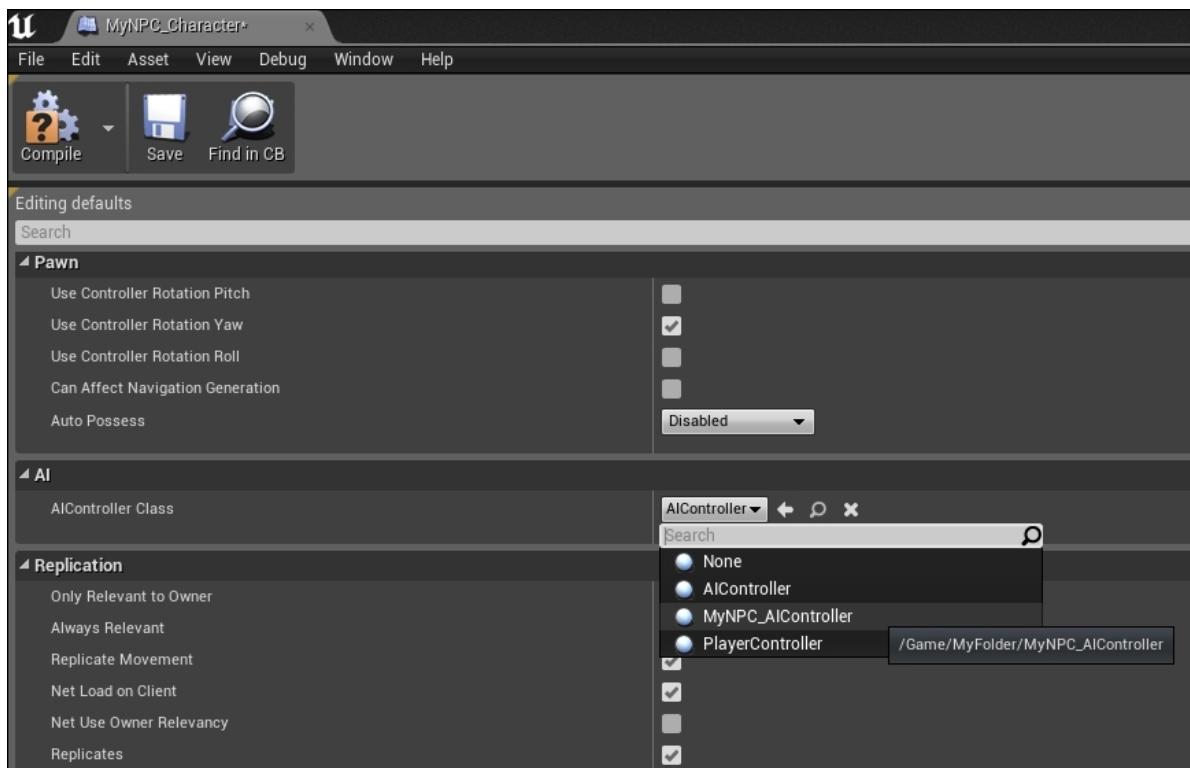


This figure shows when the wireframe for the skeletal mesh is the correct height:



## Linking AIController to the Character Blueprint

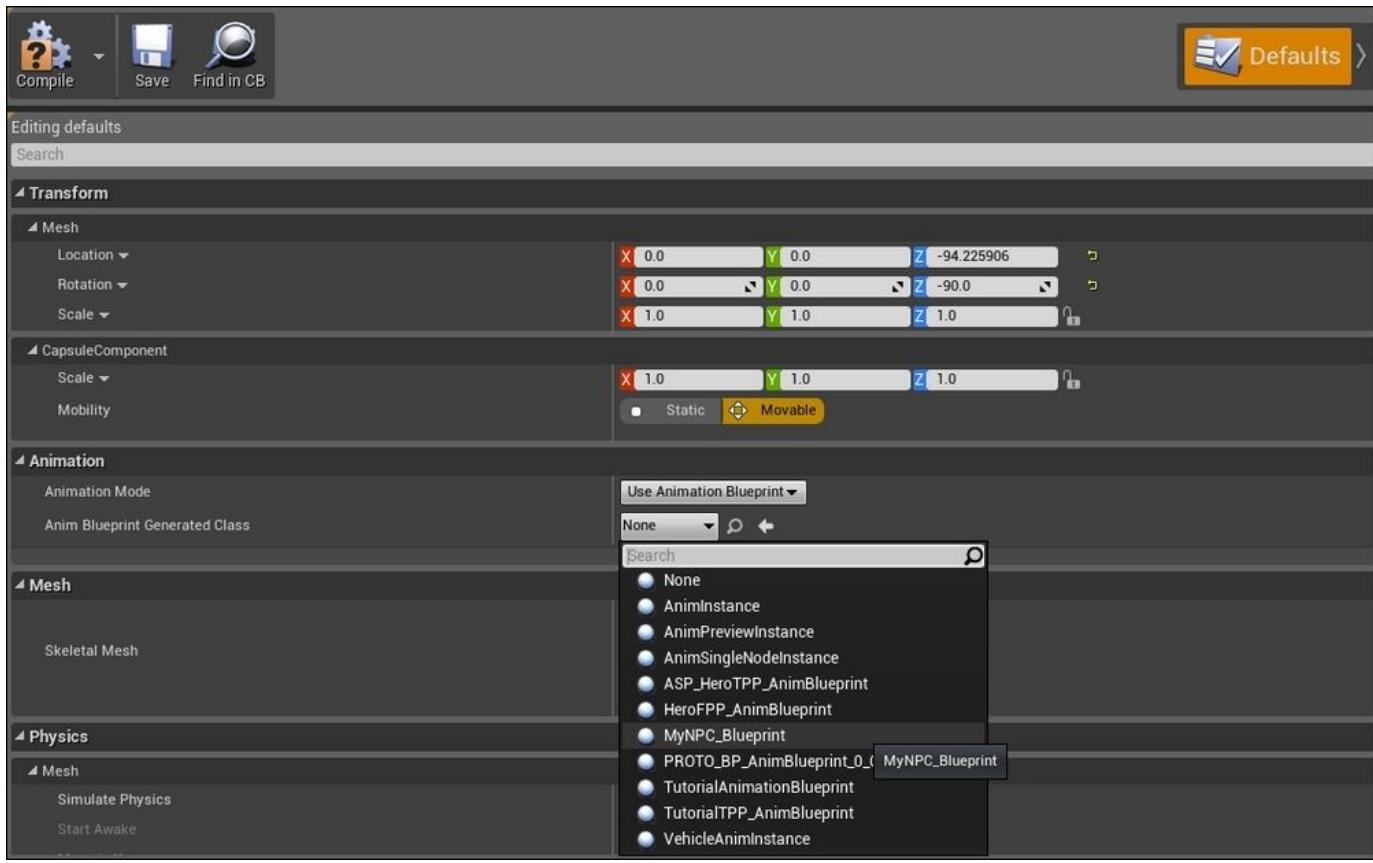
Go to the **Default** tab of **MyNPC\_Character**, scroll to the AI section, and click on the scroll box to display the options available for AIControllers. Select **MyNPC\_AIController** to assign the character to use this AIController, as shown in this screenshot. Compile, save, and close **MyNPC\_Character** for now.



Go to **Content Browser**, and click and drop **MyNPC\_Character** into the level map. Compile and play the level. You will see that the character appears in the level but it is static.

## Adding basic animation

Similar to the early implementation of assigning an animation to the mesh, we will add animation to **MyNPC\_Character**. Double-click on **MyNPC\_Character** to open the editor. Go the **Default** tab, scroll to the **Animation** section, and assign the Animation Blueprint (**MyNPC\_Blueprint**), which we created earlier for this Character Blueprint. The following screenshot shows how we can assign animation to the character. Compile and save **MyNPC\_Character**:



Now, play the level again, and you will see that the character is now walking on the spot (as we have set the speed to 100 in the Animation Blueprint, **MyNPC\_Blueprint**).

## Configuring AIController

Go to **Content Browser**. Then, go to **MyFolder** and double-click on **MyNPC\_AIController** to open the editor. We will now add nodes in EventGraph to design the logic.

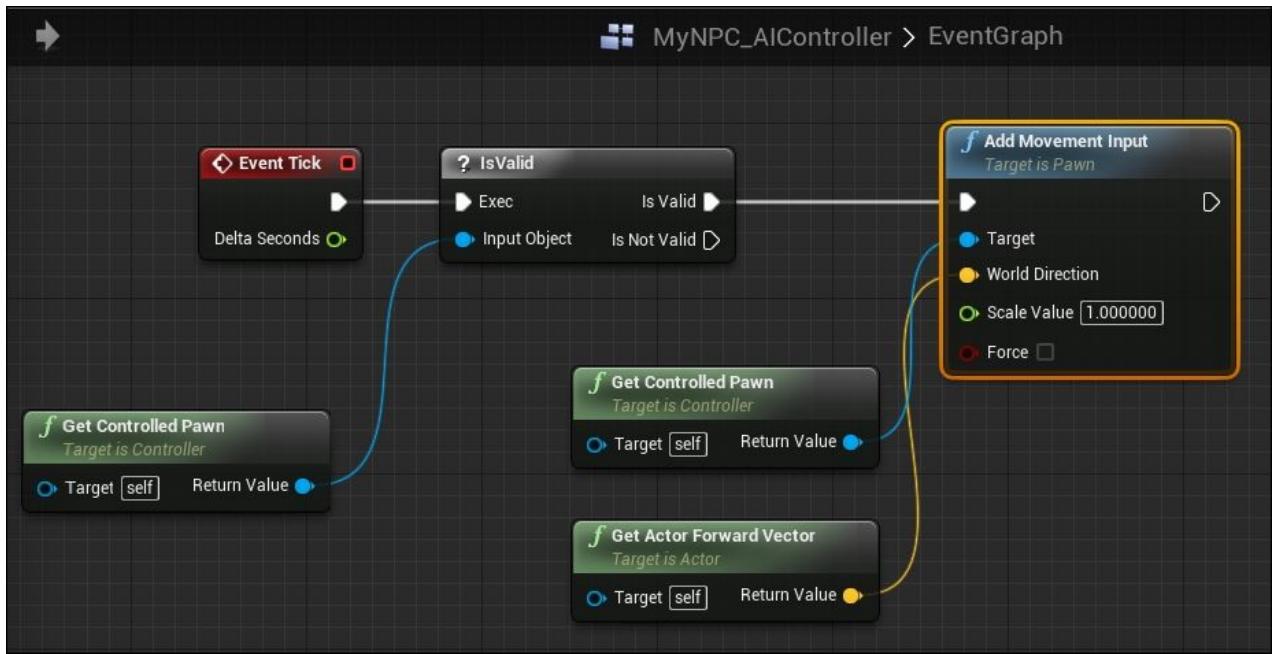
Our first mission is to get the character to move forward (instead of just walking on the same spot).

### Nodes to add in EventGraph

The following are the nodes to be added in EventGraph:

- **Event Tick** : This is used to trigger the loop to run at every tick
- **Get Controlled Pawn** : This returns the pawn of AIController (which will be the pawn of **HeroTPP**)
- **Get Actor Forward Vector** : This gets the forward vector
- **Add Movement Input** : This links the target to **Get Controlled Pawn** and **Link World Direction** to the output of **Get Actor Forward Vector**
- **IsValid** : This is to ensure that the pawn exists first before actually changing the pawn values

The following screenshot shows the final EventGraph that we want to create:



Now, play the level again, and you will see that the character is now walking forward. But it's doing this a little too quickly. We want to adjust the maximum speed at which the character moves.

### Adjusting movement speed

Double-click on **MyNPC\_Character** to open the editor. Go to the **Default** tab, scroll to the **Character Movement** section, and set **Max Walk Speed** to **100**, as shown in this screenshot:



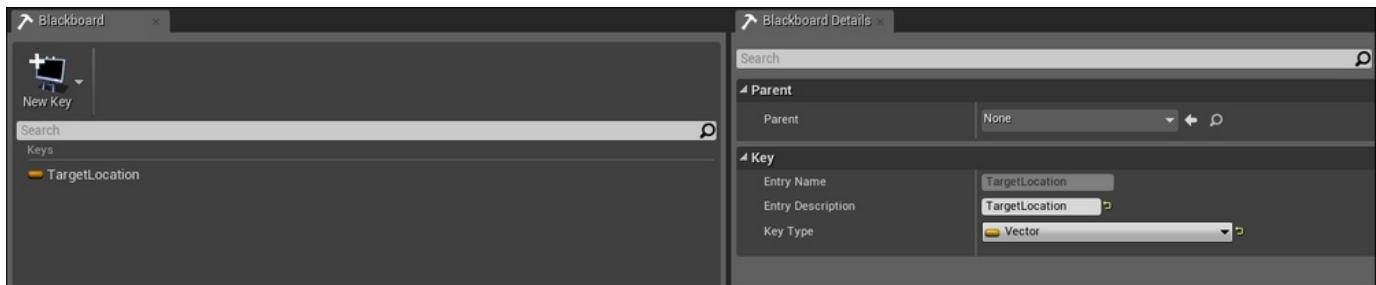
### Creating the BlackBoardData

BlackBoardData functions as the memory unit of the brain of the NPC. This is where you store and retrieve data that would be used to control the behavior of the NPC. Go to **Content Browser**, and navigate to **Create | Miscellaneous | Blackboard**. Rename it **MyNPC\_Brain**.



### Adding a variable into BlackBoardData

Double-click on **MyNPC\_Brain** to open the BlackBoardData editor. Click on **New Key**, select **Key Type** as **Vector**, and name it **TargetLocation**. This screenshot shows that **TargetLocation** is created correctly. Save and close the editor.



### Creating a Behavior Tree

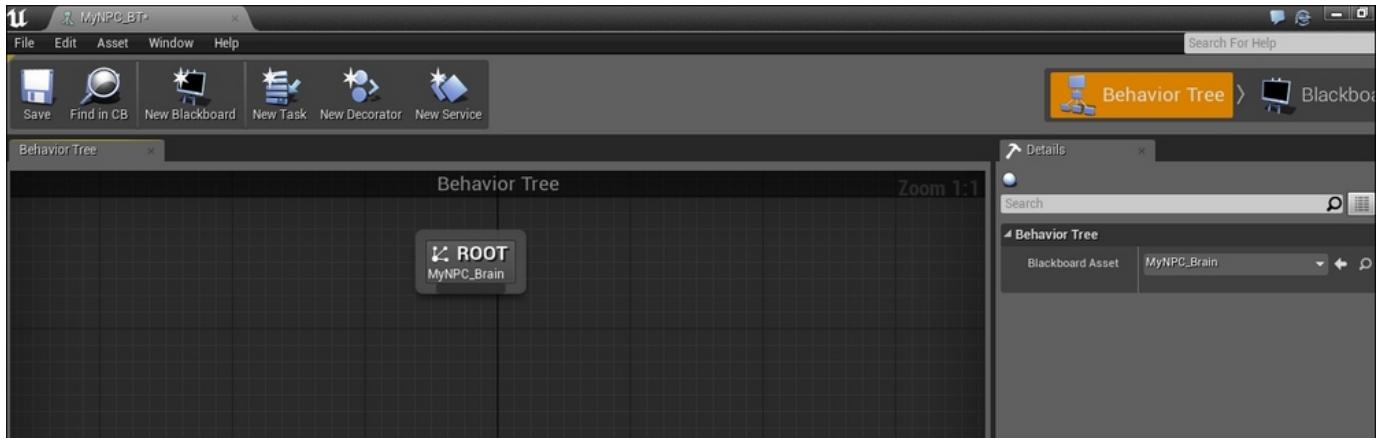
Behavior Tree is the logic path that NPC goes through to determine what course of action to take.

To create a Behavior Tree in Unreal Engine, go to **Content Browser | Create | Miscellaneous**, and then click on **Behavior Tree**. Rename it **MyNPC\_BT**.



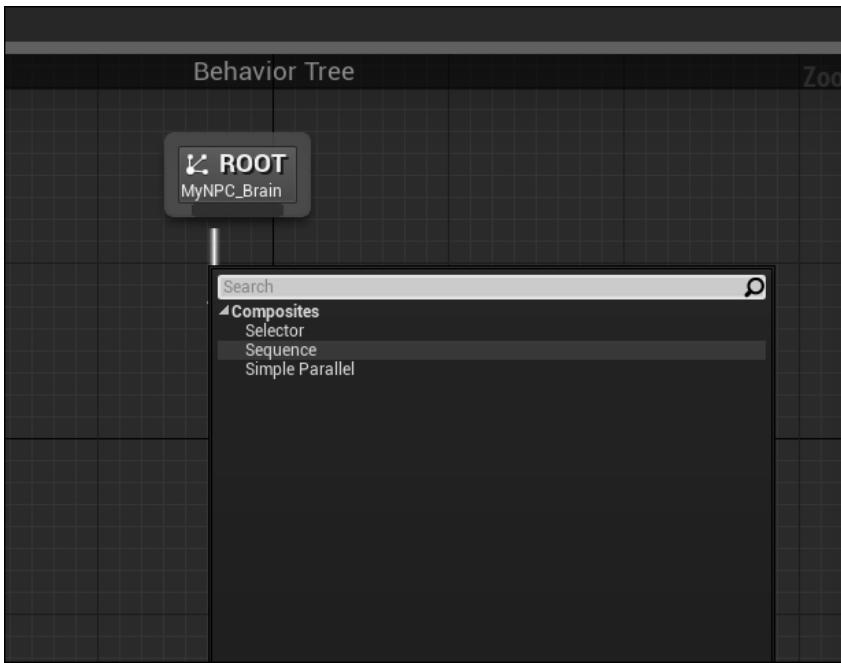
Double-click on **MyNPC\_BT** to open the Behavior Tree editor. The following screenshot shows the setting that we want for **MyNPC\_BT**. It should have **MyNPC\_Brain** set as the BlackBoard asset. If it doesn't, search for **MyNPC\_Brain** and assign it as the BlackBoard asset.

If you have already gone through the earlier exercise and are familiar with a Behavior Tree, you will notice that in this editor that there is a **Root** node, which you could use to start building out your NPC's behavior.

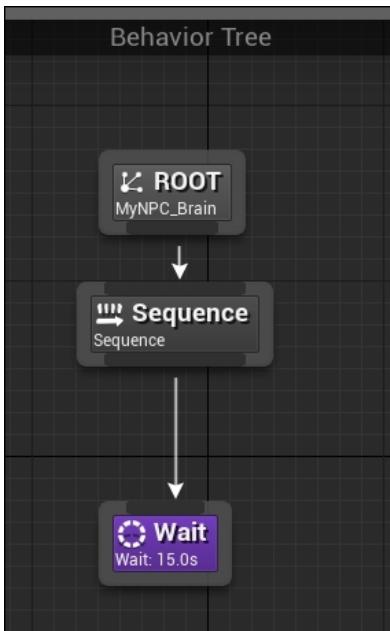


### Creating a simple BT using a Wait task

The next step here is to add on a composite node (either **Sequence**, **Selector**, or **Simple Parallel**). In this example, we will select and use a **Sequence** node to extend our Behavior Tree here. You can click and drag from the **Root** node to open up the contextual menu, as shown in the following screenshot. Alternatively, just right-click to open up the menu and select the node that you want to create.



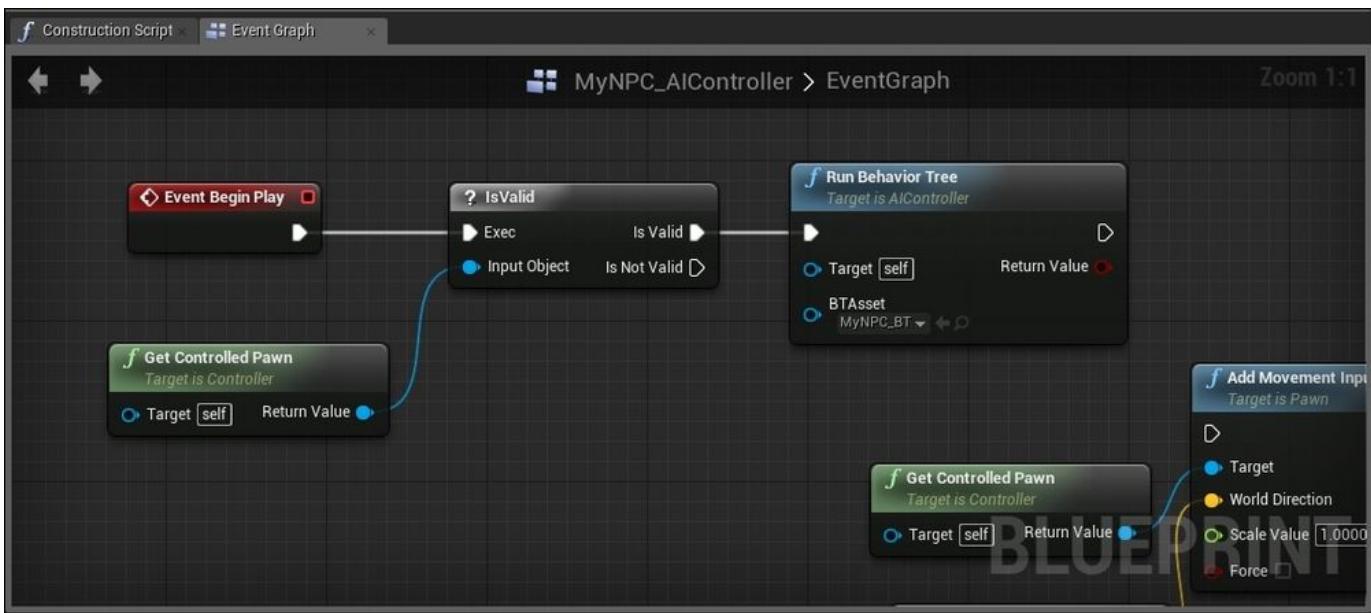
We will add a **Wait** task from the **Sequence** node. Click and drag to create a new connection from the **Sequence** node. From the contextual menu, select **Wait**. Set **Wait** to be **15.0s**, as shown in this screenshot. Save and compile **MyNPC\_BT**.



After compiling, click on **Play** in the Behavior Tree editor. You would see the light moving through the links and especially from the **Sequence** node to the **Wait** task for 15s.

## Using the Behavior Tree

Now that we have a simple implementation of the Behavior Tree, we want our NPC character to start using it. How do we do this? Go to **Content Browser | MyFolder**, and double-click on **MyNPC\_AIController** to open up the editor. Go to the **EventGraph** tab where we initially created a simple move forward implementation. Break the initial links between the **IsValid** node and **Add Movement Input**. Rewire them based on the following screenshot by linking the **IsValid** node to a new **Run** Behavior Tree node. In the **Run** Behavior Tree node, assign **BTAsset** to **MyNPC\_BT**. Next, replace **Event Tick** with **Event Begin Play** (since the BT will now replace the thinking function here). Save and compile.

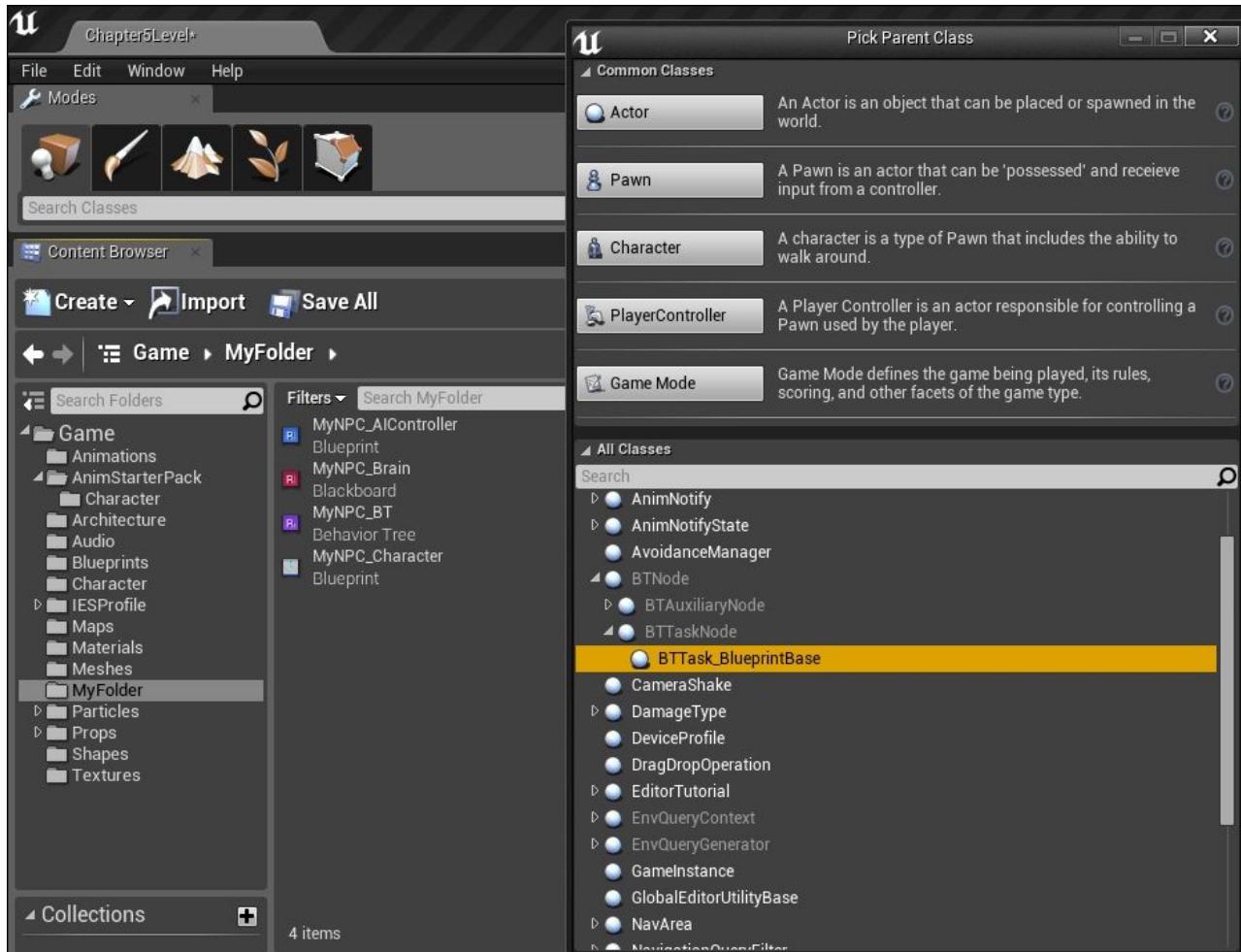


## Creating a custom task for the Behavior Tree

We want to now make the NPC select a location on the map and walk toward it.

This requires the creation of a custom task where the NPC has to select a target location. We have already created an entry in the BlackBoardData to store a vector value. However, we have not made a way to assign values to the data yet. This would be done by creating a custom Behavior Tree task.

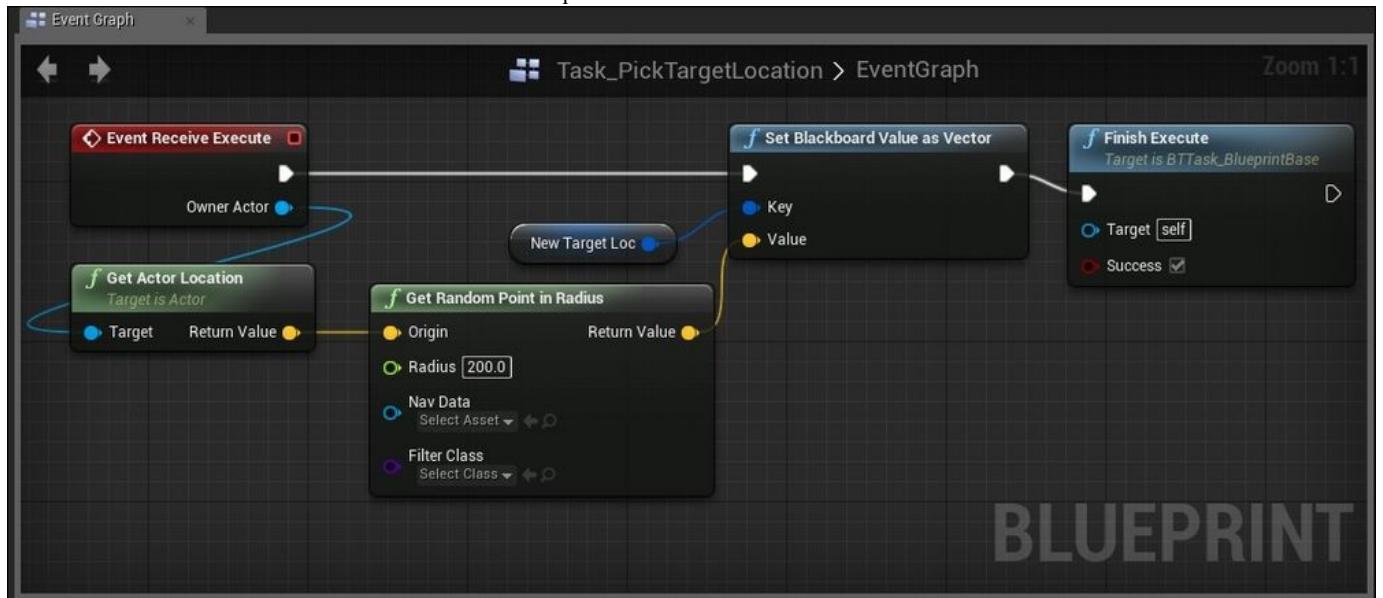
Go to **Content Browser** | **Create** | **Blueprint**. For the parent class, search for **BTNode** and select **BTTTask\_BlueprintBase**, as shown in the following screenshot. Rename this task as **Task\_PickTargetLocation**.



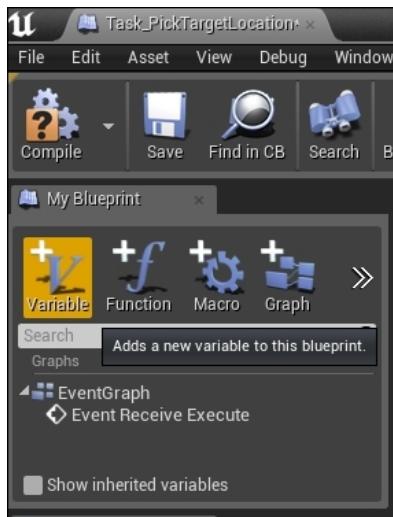
Double-click on the newly created **Task\_PickTargetLocation**. Go to **EventGraph**, create the following nodes, and link these nodes:

- **Event Receive Execute** : Link **Owner Actor** to the target of **Get Actor Location**. When **PickTargetLocation** is executed, **Event Receive Execute** starts.

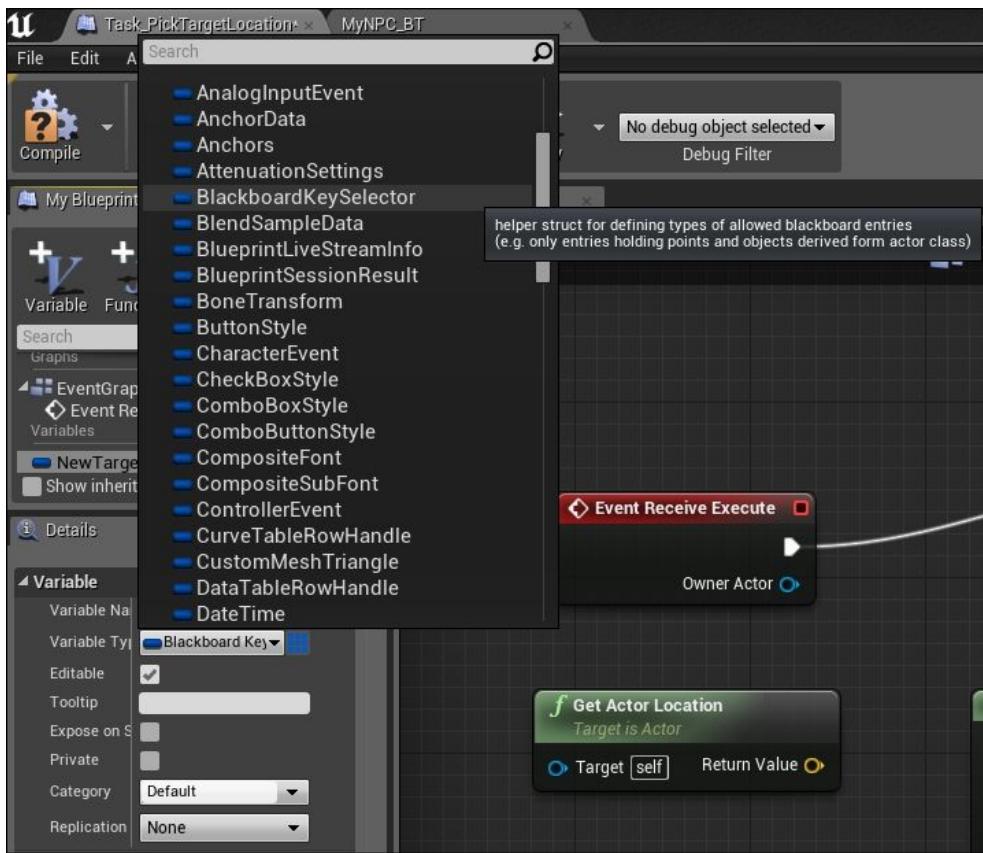
- **Get Actor Location** : Link **Return Value** to **Origin** of **Get Random Point in the Radius** node.
- **Set Blackboard Value as Vector** : Link **Event Receive Execute** to the execution arrow of **Set Blackboard Value as Vector**.
- **Get Random Point in Radius** : Link **Return Value** to the **Value** input for **Set Blackboard Value as Vector**.
- **Finish Execute** : Link **Set Blackboard Value as Vector** to the input execution of **Finish Execute**.



Notice that there is a **New Target Loc** variable linked to **Key** of **Set Blackboard Value as Vector**. We need to create a new variable for this. Click on **+Variable** , as shown in the following screenshot, to create a new variable. Name the new variable **New Target Loc** .



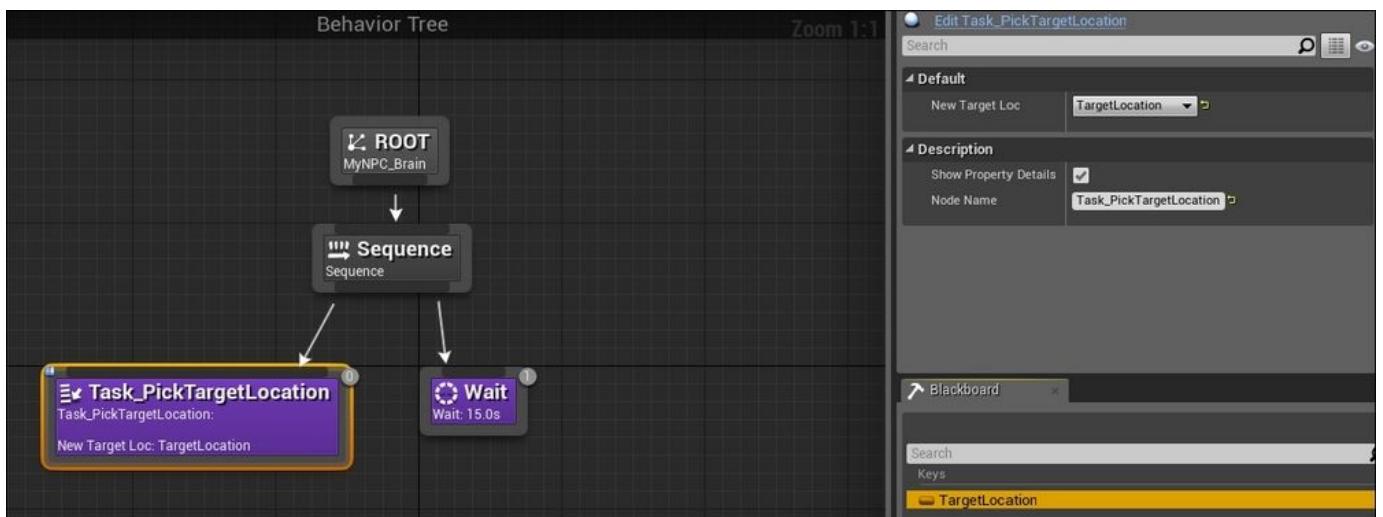
Click on the newly created **New Target Loc** to display the details of the variable. Select **BlackBoardKeySelector** as the variable type, as shown in this screenshot:



Save and compile the custom task.

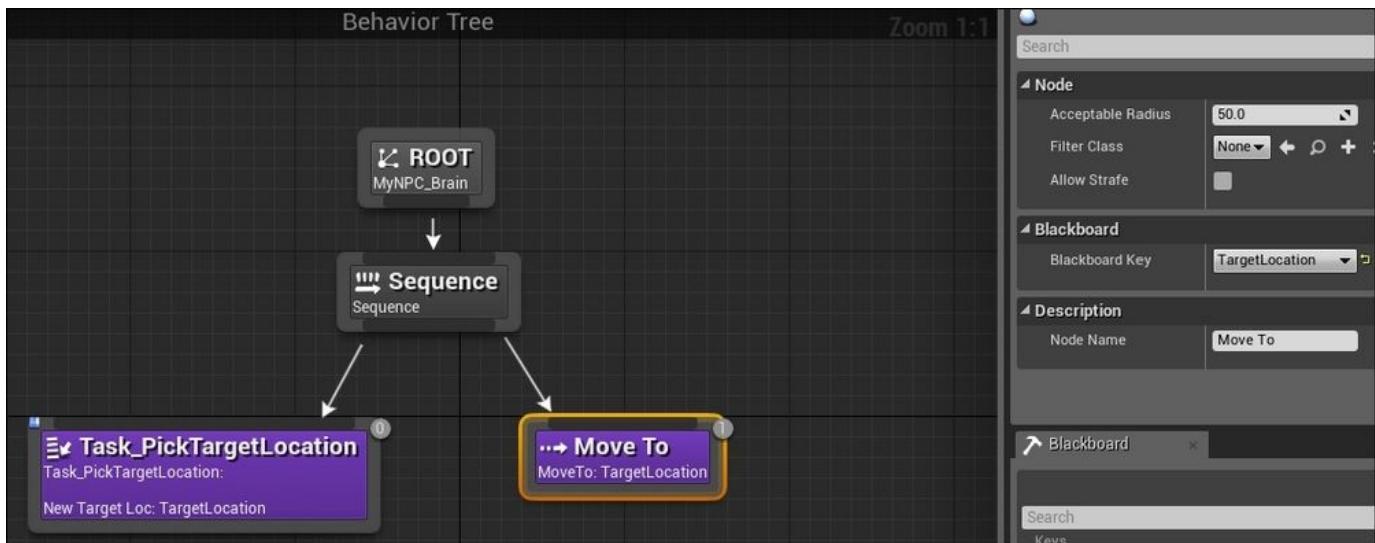
### Using the PickTargetLocation custom task in BT

Add a new link from the current **Sequence** composite node. Place the **Task\_PickTargetLocation** node to the left of the **Sequence** node so that it would be executed first, as shown in the following screenshot. Make sure that **New Target Loc** is set as **TargetLocation**:

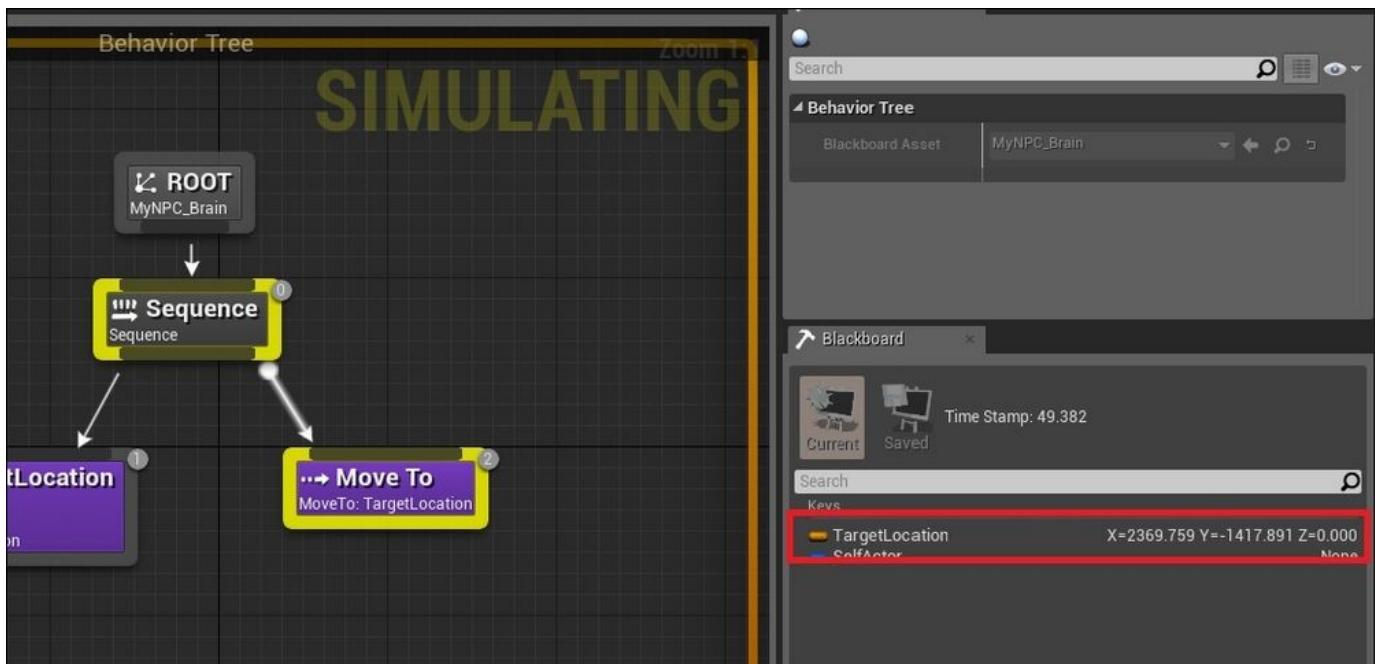


### Replacing the Wait task with Move To

Delete the **Wait** node, and add the **Move To** node in its place. Make sure that **Blackboard Key** for **Move To** is set as **TargetLocation**, as show in this screenshot:



After compiling, click on **Play** to run the game. Double-click on **MyNPC\_BT** to open the Behavior Tree editor. You would see the light moving through the links and the **TargetLocation** value changing in the Blackboard, as shown in this screenshot:



Remember to go back to the map level and see how the NPC is behaving now. The NPC now selects a target location and then move to the target location. Then, it selects a new target location and moves to another spot.

With this example, you have gained a detailed understanding of how to set up AI behavior and getting AI to work in your level. Challenge yourself to create more complex behaviors using the knowledge gained in this section.

## Implementing AI in games

I am sure you have noticed that we definitely need to create more complex behaviors to make a game interesting. In terms of implementation, it is often easier to implement more complex AI through a combination of programming and use the editor functions to take this a step further. So, it is important to know how AI can be triggered via the editor and how you can customize AI for your game.

# Summary

This chapter covers both animation and artificial intelligence. These are huge topics in game development and there is definitely more to learn about them. I hope that through this chapter, you now have a strong understanding of these two topics and will use your skills to further explore more functions in the Unreal Editor to create cooler stuff.

We learned a little about the history of animation, how animation is created today in 3D computer games through various 3D modeling software, and finally, how to import this animation into Unreal Engine to be used in games. An animation sequence is the format in which animation is stored/played in Unreal, and you've learned about a simple blend technique to combine different animation sequences.

Personally, I love how AI contributes to a game. In this chapter, you learned about the different components that make up AI logic. The main AI logic is executed through the Behavior Tree, and we learned how to construct a Behavior Tree in terms of logic as well as how to replicate this into the Unreal Editor itself through the use of BlackBoardData, Task, Composite, and other nodes.

Ending this chapter, we have covered a huge portion of what we need to create a game. In the next chapter, you will learn how to add sounds and particle effects into a game.

# Chapter 6. A Particle System and Sound

In this chapter, we will touch on the components of a game that are extremely important but often go unnoticed unless they are badly designed and out of place. Yes, we will cover particle system and sound in this chapter. In most games, they blend in so naturally that they are easily forgotten. They can also be used to create the most memorable moments in a game.

Just to recap, particle systems are used very often to create sparks, explosions, smoke, rain, snow, and other similar effects in a game that are dynamic, kind of fuzzy, and random in nature. Sound can be in the form of ambient sounds, such as the sound of rustling leaves and wind, one-off sounds, such as a pan dropping in the kitchen, or repetitive sounds, such as the running steps of a character or even music playing on the radio. Sound can be used to set the mood of a game, alert the player to something that needs attention, and provide realism to a level to make a place come alive. Let's get started.

## What is a particle system?

A particle system is a way to model fuzzy objects, such as rain, fire, clouds, smoke, and water, which do not have smooth, well-defined surfaces and are nonrigid. The system is an optimized method to achieve such fluid-looking and dynamic visual representations by controlling the movement, behavior, interaction, and look of many tiny geometry objects or sprites.

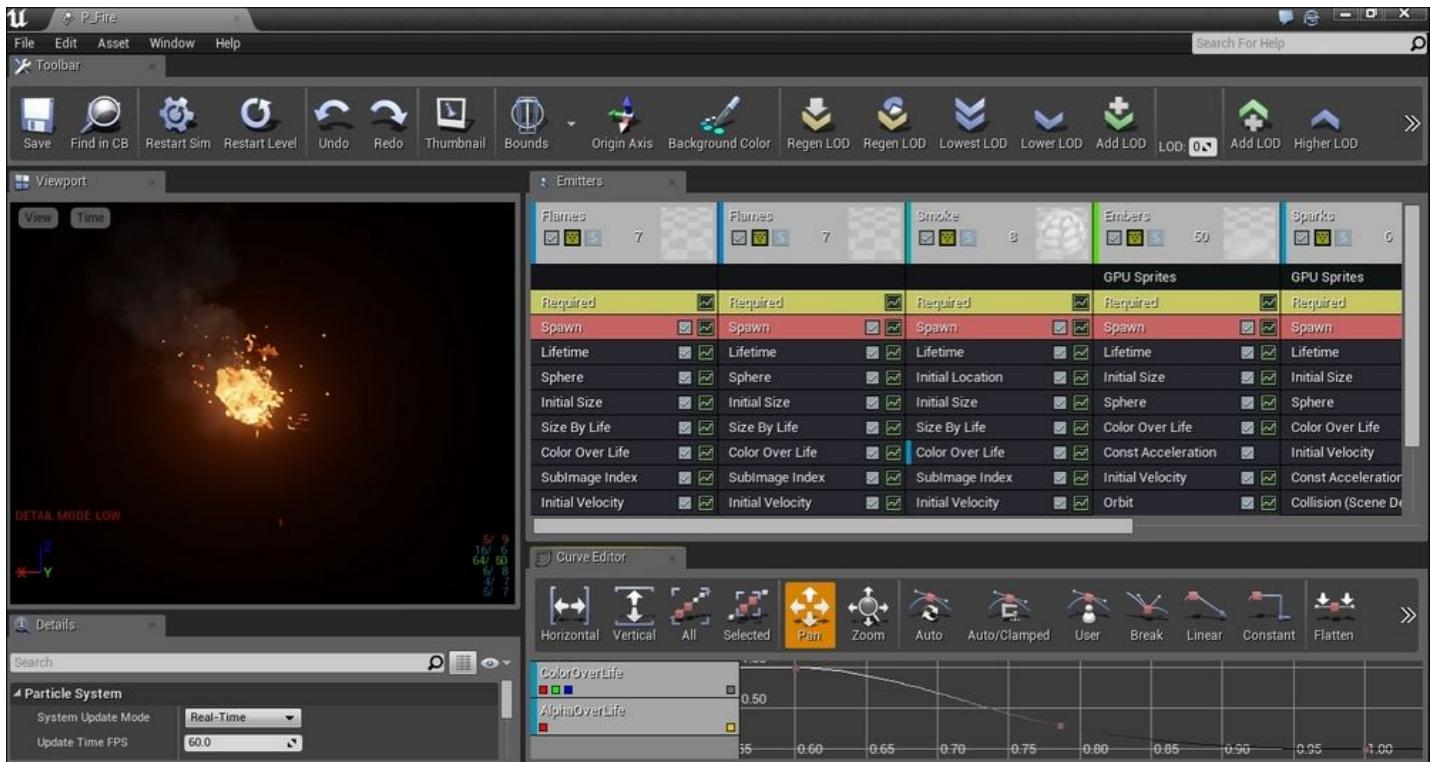
Using a combination of different particles made of different shapes, sizes, materials, and textures, with different movement speeds, rotation direction/speeds, spawn rates, concentration, visibility duration, and many more factors, we are able to create a huge variety of dynamic complex systems.

In this chapter, we will learn about the components of the particle system using Unreal's Particle System editor and Cascade editor and use these editors to create a few additions for your level.

# Exploring an existing particle system

We will start by first seeing what kind of particle systems we get in the default package of Unreal Engine 4. Go to **Content Browser** | **Game** | **Particles**. There are a couple of particle systems that we can already drag and place in the level and check out how they look.

To open a particle system, simply double-click on any of the systems. Let's take a look at **P\_Fire** together. Feel free to check out the rest of the systems as well. However, I will use this as an example to understand what we need in order to create a new particle system for our level. This screenshot shows **P\_Fire** in the editor:



On the left-hand side is **Viewport** where we can preview the particle system. On the right-hand side, in the **Emitters** tab, you can see several columns of boxes with **Flames** (twice), **Smoke**, **Embers**, and **Sparks** mentioned on top of each of the columns.

Emitters can be thought of as separate components that make up the particle system, and you can give each emitter different properties depending on what you want to create. When you put a bunch of emitters together, you will see them combining to give you a whole visual effect. In this **P\_Fire** particle system, you can see flames moving in an unpredictable manner with some sparks and embers floating around and smoke simulating a fire bursting into flames. In the next section, let's go through more concrete terminology that describes the particle system in Unreal Engine 4.

# The main components of a particle system

Very briefly, the following paragraph (taken from the official Unreal 4 documentation that's available online) very aptly describes the relationship between the different components that are used in particle systems:

*"Modules, which define particle behavior and are placed within...Emitters, which are used to emit a specific type of particle for an effect, and any number of which can be placed within a...Particle System, which is an asset available in the Content Browser, and which can then in turn be referenced by an...Emitter Actor, which is a placeable object that exists within your level, controlling where and how the particles are used in your scene.",*

Read this several times to make sure that you are clear on the relationship between the different components.

So, as described in the earlier section where we looked at **P\_Fire**, we know that the emitters are labelled as **Flames**, **Embers**, **Sparks**, **Smoke**, and so on. The different properties of each of the emitters are defined by adding modules, such as **Lifetime**, **Initial Velocity**, and so on, into them. Together, all the emitters make up a particle system. Lastly, when you place the emitters in your game level, you are, in fact, dragging the emitter actor, which references a particular particle system.

## Modules

The **Default Required** and **Spawn** modules are the modules that every emitter needs to have. There is also a long list of other optional modules that the Cascade Particle editor offers to customize your particle system. In the current version of the editor that I am using, I have the **Acceleration**, **Attractor**, **Beam**, **Camera**, **Collision**, **Color**, **Event**, **Kill**, **Lifetime**, **Location**, **Orbit**, **Orientation**, **Parameter**, **Rotation**, **Rotation Rate**, **Size**, **Spawn**, **SubUV**, **Vector Field**, and **Velocity** modules.

We will cover a few of the frequently used modules from this long list of modules through a simple exercise that's based on **P\_Fire**. I understand that it would be very boring and not very useful when grasping the basics here if I simply gave you all those definitions that you can find easily online. Instead, we will go through this section by customizing **P\_Fire** to create a fireplace for our level. At the same time, we will go through the key values within the different modules that you can adjust. Thus, you can take a look at how these values impact the look of the particle system.

For more detailed documentation on the definition of each module and parameter, you can refer to the Unreal 4 online documentation (<https://docs.unrealengine.com/latest/INT/Engine/Rendering/ParticleSystems/Reference/index.html>).

The commonly used modules are listed as follows:

Module	Key parameters it can control
<b>Required</b>	Material used for the particles
<b>Spawn</b>	Rate and distribution of the spawn
<b>Initial Size</b>	Size of the initial particle
<b>Lifetime</b>	Time duration for which the particle stays visible
<b>Color Over Life</b>	Color of the particles over their lifetimes

# The design principles of a particle system

The design principles of a particle system can be configured through a research and iterative creative process. Let's take a look at each one of them in the following section.

## Research

Details are probably key to designing a realistic particle system. Very often, creating a particle system lies in the realm of an artist as we need an artistic touch to create a visually appealing and somewhat realistic replica of the effect that we want to create.

For starters, it is good to research a little on what the actual effect looks like. Here are some steps to help you get started:

- Identify the different components that are needed (break the particle effects down into the different components).
- Determine the relationship among the different components (the size of the particles that are relative to one another, spawn rate, lifetimes, and so on).
- Next, look at other similar effects that are created in the **Computer Graphics ( CG )** space. The reason for doing this is that sometimes, actual effects can be a little too monotonous, and there are many amazing visual effect people out there who you can learn from to spice things up a little. So, it is a great idea to spend a little time checking out what others have done already, rather than spending a whole lot of time experimenting and not getting what you want to achieve.

## The iterative creative process

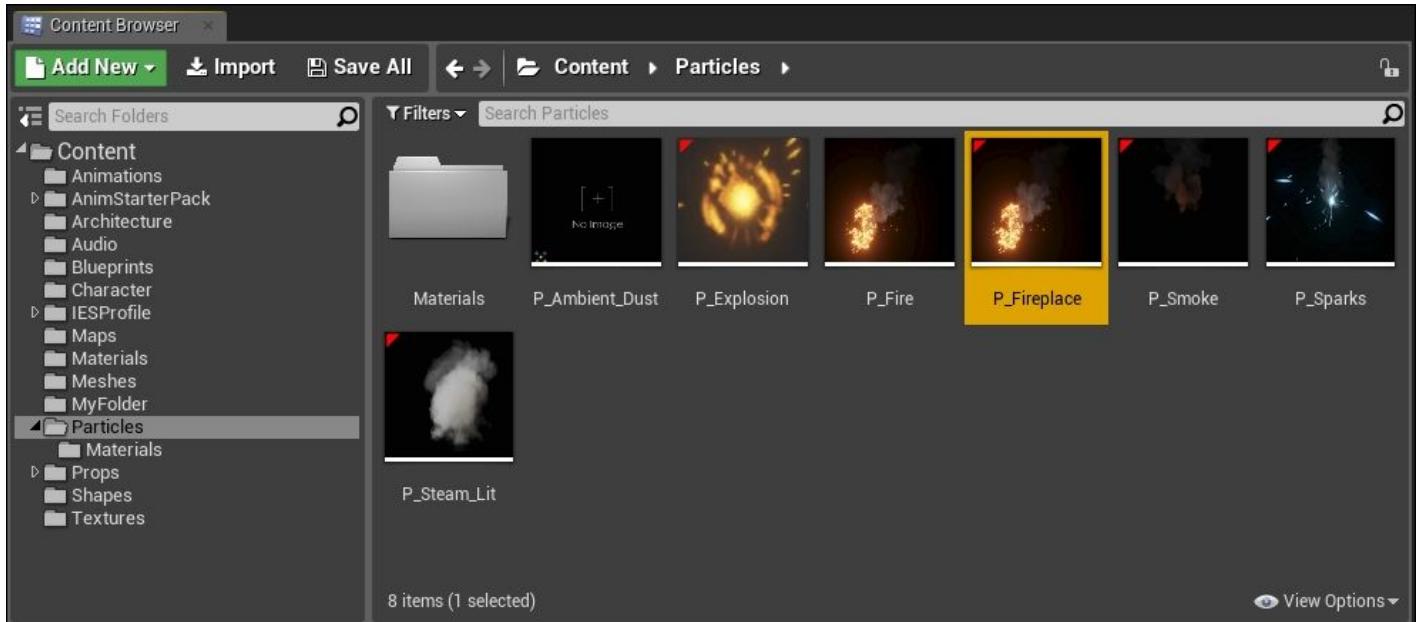
Creating the perfect looking particle system that you want usually involves quite a bit of tweaking and playing around with the parameters that you have. The key to doing this is knowing what parameters there are and what they affect. During the initial phase of design, you should also try adding or removing certain modules to see how they actually impact the overall look of the system. This does not mean that more is always better. Additionally, it is also wise to save backup copies of your iterations so that you can always go back to the previous versions easily.

Being extremely proficient in creating the particle system, I think, involves a combination of good design planning, having the patience to iterate, and making small adjustments to get the look that you eventually want.

# Example – creating a fireplace particle system

In this example, we will duplicate **P\_Fire** and edit it to create a fire for a fireplace in the level. We will also change a part of the current level in which we have to place this new fireplace particle system.

Go to **Content Browser | Particles**, select **P\_Fire**, and duplicate it. Rename it **P\_Fireplace**. This screenshot shows how **P\_Fireplace** is created in the **Particles** folder:



Let's open **Chapter5Level** and rename it **Chapter6Level** first. We will first add a fireplace structure to the level to set the context for this fireplace effect. This will help you follow the creation process better. This screenshot shows the original living room space:



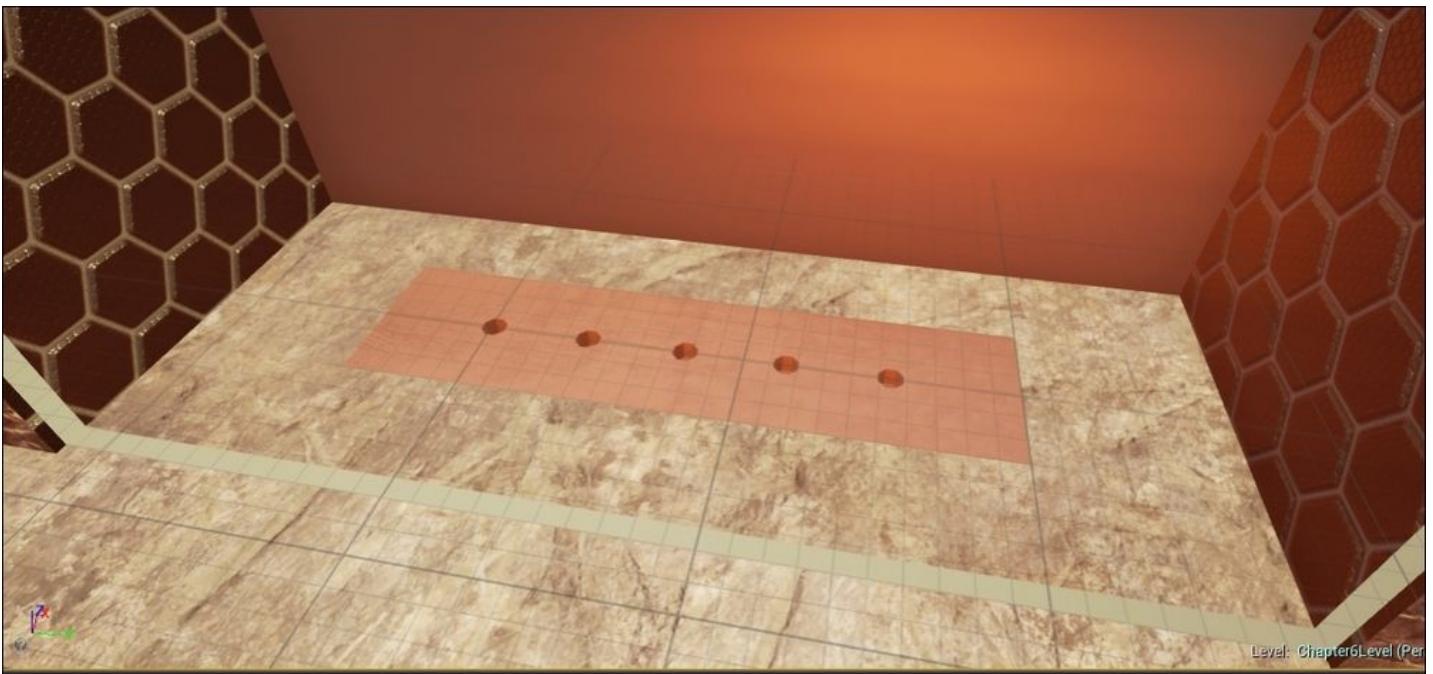
The following screenshot shows the modified living room space with a fireplace:



This screenshot shows a zoomed in version of the fireplace structure if you intend to construct it:



Zooming in on the metal vents will look like this:



What we did here was delete the lights and low cabinet structure and replaced it with this:

- **TopWoodPanel** (material: **M\_Wood\_Walnut**) : X = 120, Y = 550, Z = 60
- Concrete pillars around the glass (material: **M\_Brick\_Cut\_Stone**)
- **ConcretePillar\_L** and **ConcretePillar\_R** : X = 100, Y = 150, Z = 220
- **ConcretePillar\_Top** : X = 100, Y = 250, Z = 100
- Fireplace glass and inside (material: **M\_Glass**)
- **Fireglass** : X = 5, Y = 250, Z = 120
- **MetalPanel** and **MetalPanel\_Subtractive** : X = 40, Y = 160, Z = 10
- **FireVent1** to **FireVent5**

Use the BSP subtractive cylinder with the following setting, as shown in the following screenshot. Here, **Z** is **10**, **Outer Radius** is **3**, and **Sides** is **8**:



The lower extended structure (made up of two BSPs) consists of the following:

- **Thinner extension platform** : X = 140, Y = 550, Z = 10
- **Thicker base** : X = 120, Y = 550, Z = 30

## Crafting P\_Fireplace

Now, double-click on **P\_Fireplace** to open up the Cascade Particle System editor. Since we duplicated it from **P\_Fire**, it has the same emitters as **P\_Fire**: the two **Flame**, one **Smoke**, one **Sparks**, one **Embers**, and one **Distortion** module.

Observe the current viewport. What do you see? The original **P\_Fire** effect is more like a sequence of random bursts of flames that disappear pretty quickly after the initial burst. What kind of fire do we need for the fireplace that we have created? We need more or less continuous and slower moving flames that hover in a fixed position.

With this difference and objective in mind, we will next determine which of the components of **P\_Fire** we want to keep as our fire effect for the fireplace.

## Observing the solo emitters of the system

Using the solo button and checkbox in each of the modules, toggle S on or off, and alternatively mark/unmark the checkbox to observe the individual components. This screenshot shows you the location of the solo button and checkbox:



## Deleting non-essential emitters

The first step was to delete the second **Flame** emitter (the first being the left-most) and the **Smoke** emitter. The reason for this was, I think, so that I could work with a single flame to create a fire for the fireplace. The **Smoke** emitter was removed mainly because of it is a gas/electric fire; thus, I would expect less smoke. You could alternatively unmark the checkbox at the top of the window to hide the entire emitter first before deleting it permanently.

## Focusing on editing the Flame emitter

Keeping the only **Flame** emitter, the flame was still appearing at random spots within a certain radius and then disappearing quickly after that. We will address each of the issues here one by one:

- **Configure Lifetime** : So, since we need to have the fire burning continuously instead of in short bursts, I will first adjust the **Lifetime** property so that the fire burns for a longer period of time before disappearing. Change **Distribution Float Uniform**, with **Min** kept as **0.7** , **Max** as **1.0** , and **Distribution Constant** as **1.2** .
- **Remove Const Acceleration+** : Now, the flame lingers longer on screen before disappearing. However, the flames seem to be drifting away from the spawn location after they are spawned. For a fireplace, flames more or less remain in the same location. So, I turn off **Const Acceleration+** in the **Flames** module by unmarking the checkbox. The flames now seem to be moving away from the spawn location a lot less.
- **Remove Initial Velocity** : After removing the acceleration module, it still seems like the flames are moving away; my guess for this is that the particles had some initial velocity, and so I turned off this module to confirm my suspicion and it seemed to work.
- **Configure Spawn** : The flames looked quite sparse as they are small, and this creates some blank space within the spawn area during short intervals. I could adjust the size of the flame to make it bigger, but when I did this, the flame looked too distorted. So, I decided to increase the spawn rate instead so that more flames could occur per minute. Change the spawn rate for **Rate Scale Distribution** from **5.0** to **20.0** . Increase **Distribution Float Constant** from **1.0** to **3.0** .

## Looking at the complete particle system

Now, I've turned the other emitters back on again to look at the whole particle system effect and also see if it requires more editing. It looks pretty okay for a fireplace fire now so I've stopped here. Feel free to go ahead and adjust the other properties to improve the design. These are the very basics of modifying an existing particle system, and I hope you have familiarized yourself with the particle system editor through this exercise.

# Sound and music

Sound and music are an essential part of the game experience. Ever watched television with the volume switched off? Just watching subtitles and lip movements is not enough. You want to hear what the character on the screen is saying and how they are saying it. For games, it is pretty much similar, and on top of this, pretty often, you get cues through the sound and music. If you have played *Alien: Isolation*, you need to listen to the sounds in the game to know whether you have an alien coming in your direction. This can be a matter of life and death in the game. It pretty much determines whether you end up as a winner or simply a delicious meal for the alien. So, are we ready now to learn how sound and music are created for games, and how we use the Unreal Editor to incorporate them into our game level?

# How do we produce sound and music for games?

Many game productions have original music written for in-game scenarios; some also use actual songs sung by professional singers as theme songs. Music in games is a big thing and it's dearly remembered by fans of the game. Sometimes, the music itself is enough to trigger memories of the gaming experience. Thus, game studios need to spend time creating suitable music to complement their games.

If you are a huge fan of video game music, there are also concerts that you can go to where the orchestra plays music from popular games (check out Video Games Live at <http://www.videogameslive.com/index.php?s=home> ).

Creating music for a game is very similar to composing music for a piece; it should trigger appropriate emotions when it's played. The choice of music needs to match the pace and situations of the game. Using a JRPG game as an example, you should be able to differentiate between in-battle music versus the music that's played when you are in a menu, loading the game, or when you've just won a battle. Very often, music is created on the basis of the needs of the game, and the music composer has to probably come up with a few different versions and let the team and/or management review it before the best piece is selected.

If you do not intend to create original music or sound for your game, you can find many free downloadable sounds and music online these days. When using free online music and sounds, do ensure that you do not violate any digital rights or copyrights when incorporating them in your game.

# **Audio quality**

The reason why we are discussing audio quality is because sound quality, like image quality, is of huge importance these days. We already use the 4K resolution image quality today, and there will be more devices and games that would support this in the future. How about sounds? The listening experience needs to match the quality of the image and provide more than just mono or stereo sounds. Sound experience has also progressed to multichannel surround sound, starting at 5.1, 7.1, and beyond these days, to obtain a life-like immersive audio experience. This is definitely something to think about when creating, storing, and playing audio files.

# How are sounds recorded?

Sounds are generated in the form of analog waves, which are continuous waves, which you'll see shortly in the upcoming figure. We can record surround sound through a recording device. For multichannel sound recording, you need to have certain methods to record music that can use a simple recording setup known as **Deca Tree**. Here, microphones are placed in a particular fashion to capture sounds from the left, right, front, and back of the source. There are also many processing techniques that can filter and convert sounds that are recorded to mimic the various components needed for each of the channels.

We take samples of the analog sound waves that are produced by a piano at close intervals (the rate at which the samples are taken between intervals is known as sampling frequency). The process of taking samples from analog waves to store them digitally is known as **Pulse Code Modulation (PCM)**. These samples can be stored in uncompressed PCM-like formats or be compressed into a smaller and more manageable file size using audio compression techniques. Wav, MP3, Ogg Vorbis, Dolby TrueHD, and DTS-HD are some of the formats that audio is commonly saved as. Ideally, we want to save audio into a lossless compressed format so that we get a small manageable file that contains amazing sounds.

When the digital format of the sound is played back, the analog sound wave is reconstructed using the stored information. Close resemblance to the original analog sound waves is one way to ensure sounds of good quality. By increasing the number of channels to create a 3D sound effect using the basic 5.1 surround, which requires five speakers, one for front left, one front right, one center, one back left (as surround), one back right (as surround) and a subwoofer, also greatly improves the listening experience.

# The Unreal audio system

We now have a general understanding of why we need audio in games and how it's created and recorded. Let's learn about the Unreal audio system and the editor that can be used to import these audio files into the game, and we'll also learn about the tools that can be used to edit and control playbacks.

# Getting audio into Unreal

How do we get the audio files into Unreal? What do you need to take note of?

## The audio format

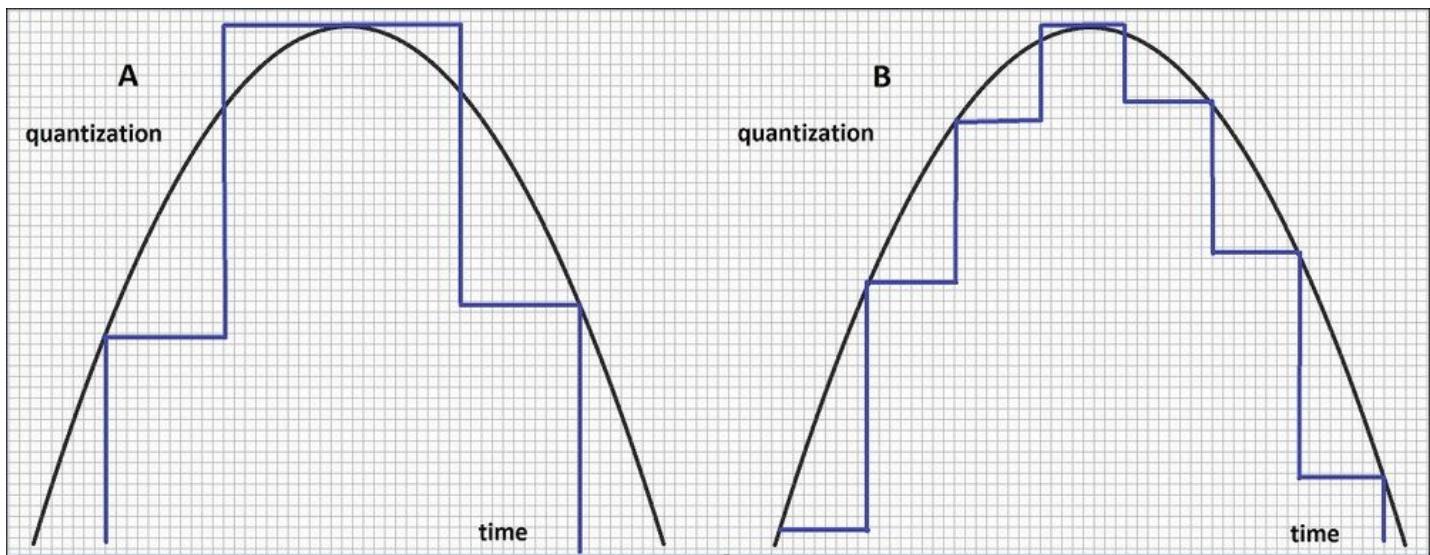
Unreal supports the importing of sounds only in the `.wav` format. The `.wav` format is a widely used format that can store raw uncompressed sound data.

## The sampling rate

The sampling rate is recommended at 44100 Hz or 22050 Hz. As mentioned earlier, the sampling rate determines how often the analog wave is recorded. The higher the frequency (measured in Hertz or Hz), the more data points of the analog wave that are collected, which aids in a better reconstruction of the wave.

## Bit depth

The bit depth is set as 16. It determines the granularity at which the amplitude of the audio wave can be recorded, which is also known as the resolution of the sound. For a bit depth of 16, you can get up to 65,536 integer values (2<sup>16</sup>). The reason why we are concerned with the bit depth is because during the sampling process of the analog waves, the actual value of the amplitude of the wave is approximated to one of the integer values that can be stored based on the bit depth. The following figure shows two different bit depths. The figure on the left-hand side illustrates when the bit depth is low, and the signal is more inaccurately sampled because it is sampled in larger increments. The figure on the right-hand side illustrates when the bit depth is higher, and it can be sampled at smaller increments, resulting in a more accurate representation of the wave:



The loss in accuracy of the representation of the wave can be termed as a quantization error. When the bit depth is too low, the quantization error is high.

The **Signal to Quantization Noise Ratio (SQNR)** is the measurement used to determine the quality of this conversion. It is calculated using the ratio between the maximum nominal signal strength and the quantization error. The better the ratio, the better the conversion.

## Supported sound channels

Unreal currently supports channels such as mono, stereo, 2.1, 4.1, 5.1, 6.1, and 7.1.

When importing files into Unreal, take note of the file naming convention that is in place so that the right sound is played from the right channel.

The following table shows the 7.1 surround sound configuration with all the file naming conventions that are necessary for the correct playback:

<b>Speakers</b>	Front-left	Front-center	Front-right
<b>Extension</b>	<code>_f1</code>	<code>_fc</code>	<code>_fr</code>
<b>Speakers</b>	Side-left	Low frequency (commonly known as subwoofer)	Side-right
<b>Extension</b>	<code>_sl</code>	<code>_lf</code>	<code>_sr</code>
<b>Speakers</b>	Back-left		Back-right
<b>Extension</b>	<code>_bl</code>		<code>_br</code>

This table shows you the files that are used for the 5.1 surround system:

<b>Speakers</b>	Front-left	Front-center	Front-right
<b>Extension</b>	_f1	_fc	_fr
<b>Speakers</b>	Side-left	Low frequency (commonly known as subwoofer)	Side-right
<b>Extension</b>	_sl	_lf	_sr

This table shows you the files that are used for the 4.0 system:

<b>Speakers</b>	Front-left	Front-right
<b>Extension</b>	_f1	_fr
<b>Speakers</b>	Side-left	Side-right
<b>Extension</b>	_sl	_sr

# Unreal sound formats and terminologies

There are a couple of terms in the Unreal Sound system that we need to get acquainted with:

- **Sound waves** : These are the actual audio files that are in the .wav format.
- **Sound cues** : This is the control system for a sound wave file. Sound cues are what we use to manipulate the volume, start, and end of the sound waves. So, in order to control how an audio file is played in the game, you can edit the properties on the Sound Cue, which, in turn, affects the wave file or files that it is associated with.
- **Ambient Sound Actor** : This is the class actor that you add to the game level. This actor is associated with the Sound Cue to play the audio files that you need for the game.

Now, we are ready to use the Sound Editor in Unreal.

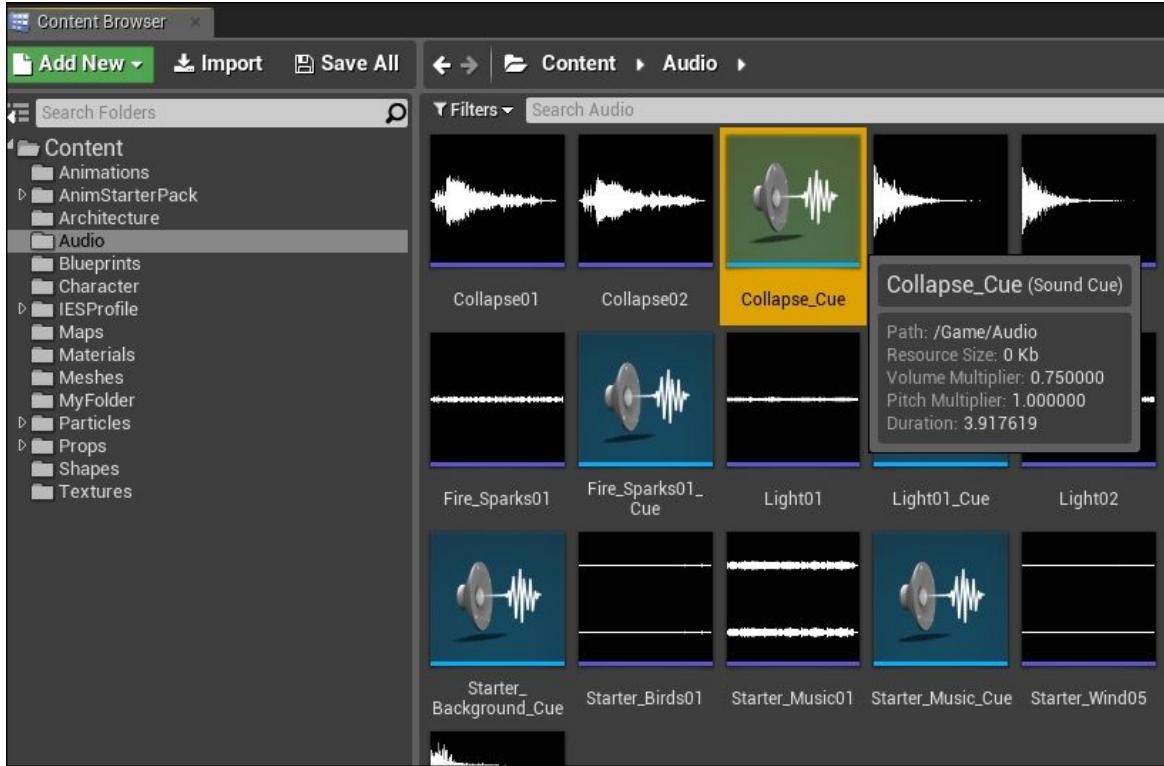
# The Sound Cue Editor

Since we are not editing the actual audio file per se, the sound editor in Unreal is known as the Sound Cue Editor. We are, in fact, editing the way the sound can be played through a control device known as a Sound Cue.

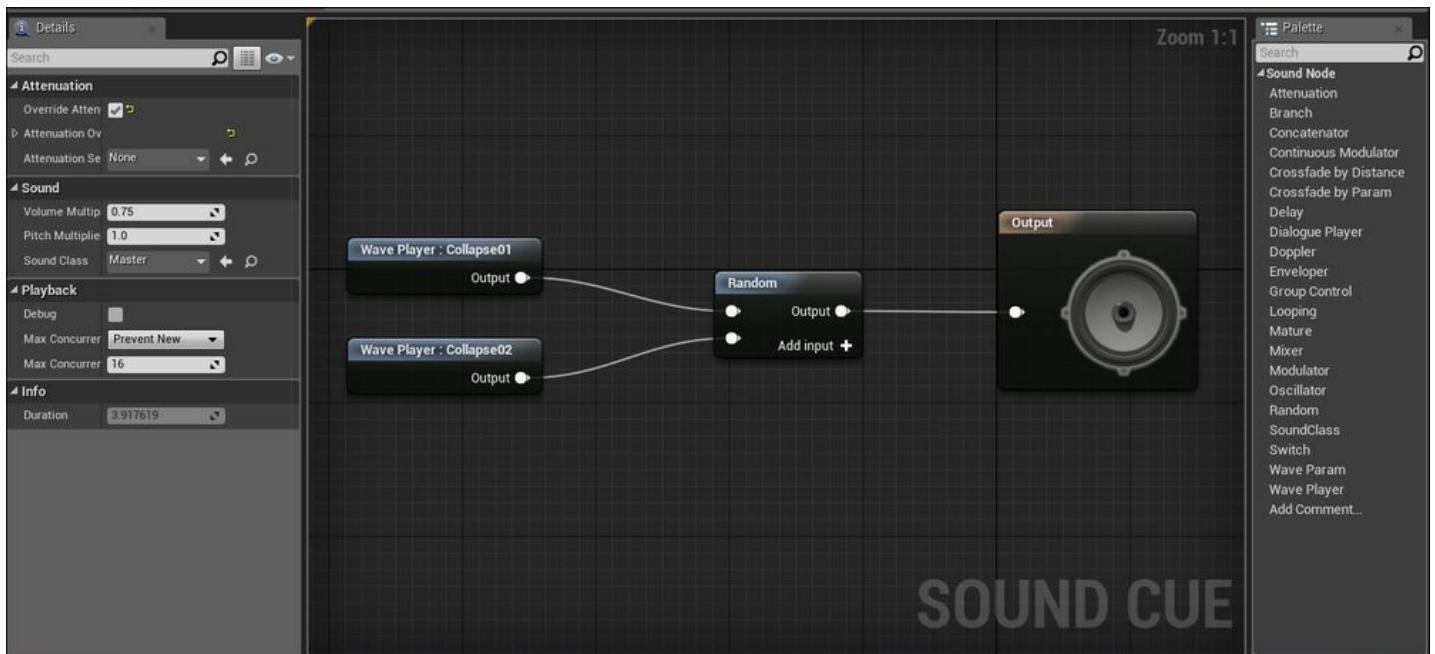
Let's learn more about the functionalities of the Sound Cue Editor.

## How to open the Sound Cue Editor

Go to **Content Browser | Audio**. Go to any Sound Cue file, and double-click to open the Sound Cue Editor. This screenshot shows where I could find a Sound Cue in **Content Browser**:



When you double-click on a Sound Cue, the Sound Cue Editor opens up, and it looks quite a lot like the Blueprint Editor with modules and lines. This screenshot shows you what the Sound Cue Editor for **Collapse\_Cue** looks like:



Notice that in the preceding screenshot **Collapse\_Cue** it has two inputs called **Wave Player: Collapse 01** and **Wave Player: Collapse 02**. These are joined to a **Random** node, and the output goes to the final node known as **Output**. What this does is that when this Sound Cue is played, one of the two collapse sounds gets

randomly selected and is played. This creates a variety when sounds are played in the same circumstance; they are both collapse sound effects but slightly different.

We will learn more about the components that we could use to design the Sound Cues later. We'll also go through an exercise later to create our own Sound Cue in the editor.

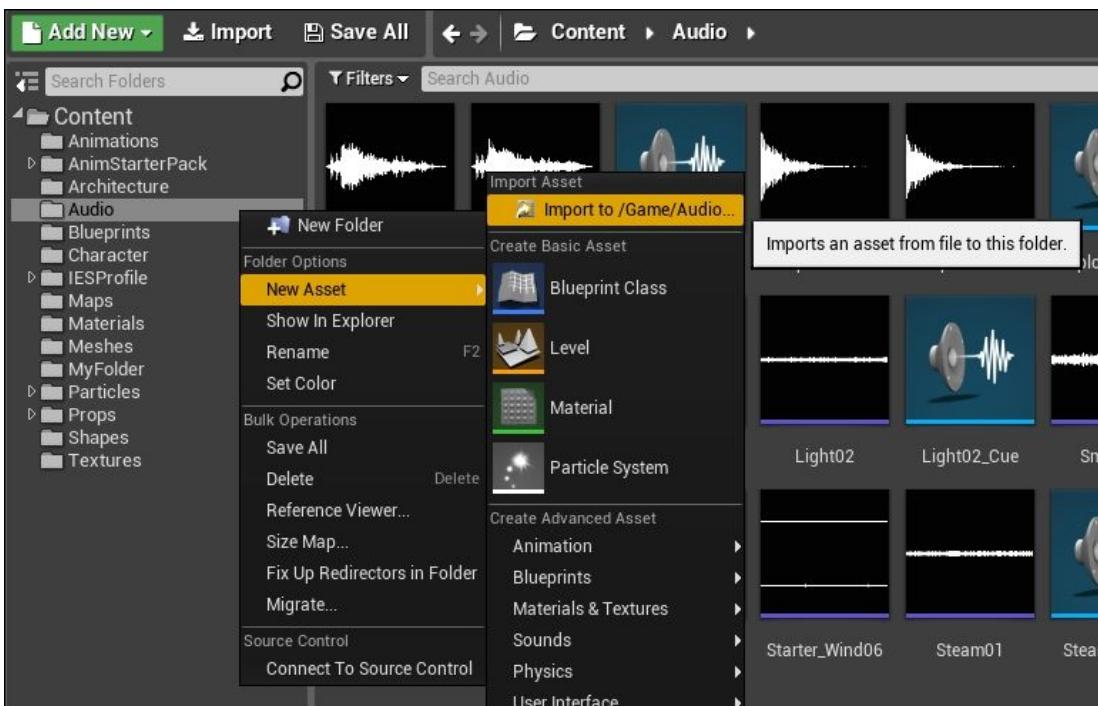
## Exercise – importing a sound into the Unreal Editor

You may come across a situation where you have created your own audio effect file and want to use it in the game. We will first start by importing this file.

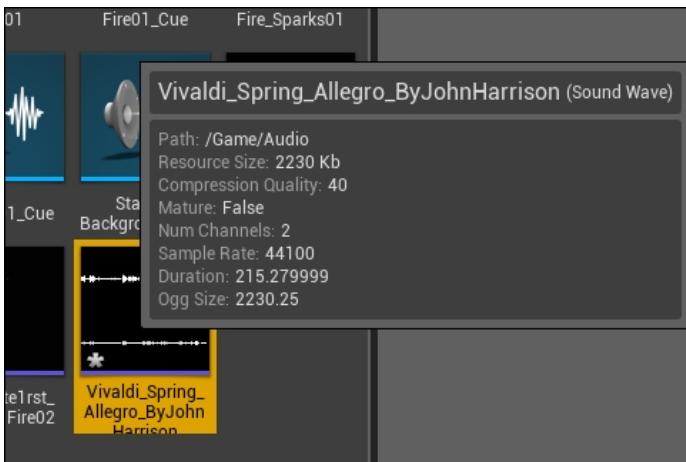
For this exercise, I have used an audio clip downloaded from a Wikipedia site ([https://en.wikipedia.org/wiki/The\\_Four\\_Seasons\\_\(Vivaldi\)](https://en.wikipedia.org/wiki/The_Four_Seasons_(Vivaldi))) with a Vivaldi piece from The Four Seasons. This is shared by John Harrison.

This file is in the Ogg format, and yes, Unreal only supports .wav files. First, I converted the file type from .ogg to .wav using software that's listed on the Vorbis website at <http://vorbis.com/software/>. Be careful about the WAV file settings that Unreal is expecting it to be in.

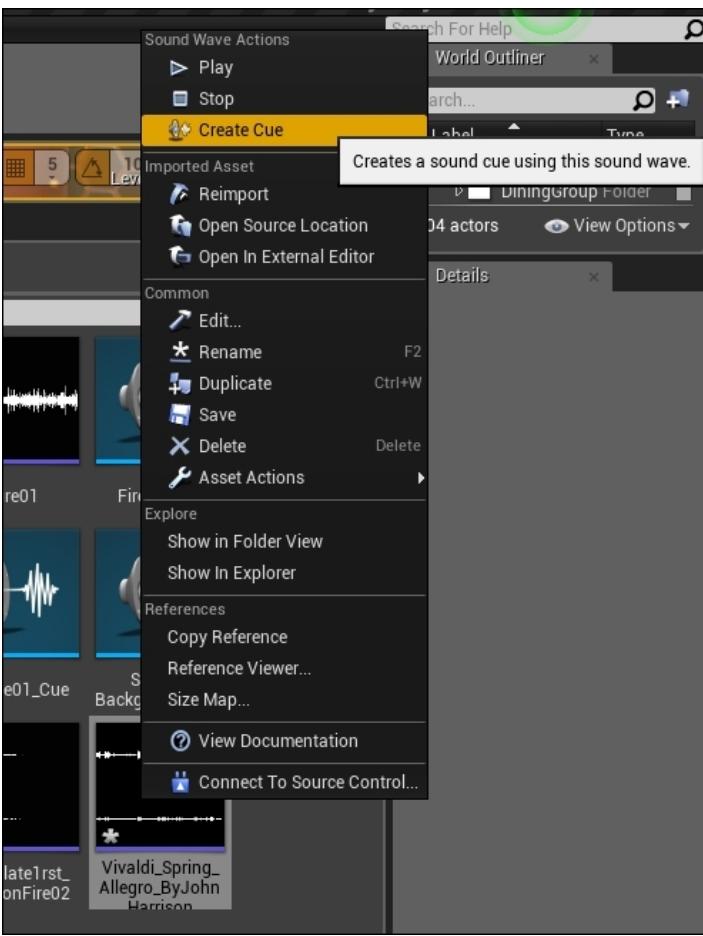
After getting the right wav file, we are ready to import it into the Sound Editor. Go to **Content Browser** | **Content** | **Audio**, right-click on it to display the contextual menu, navigate to **New Asset** | **Import to /Game/Audio**, and browse to the folder where you saved the .wav file and select it. This screenshot shows where you can find the function in the editor to import the .wav file:



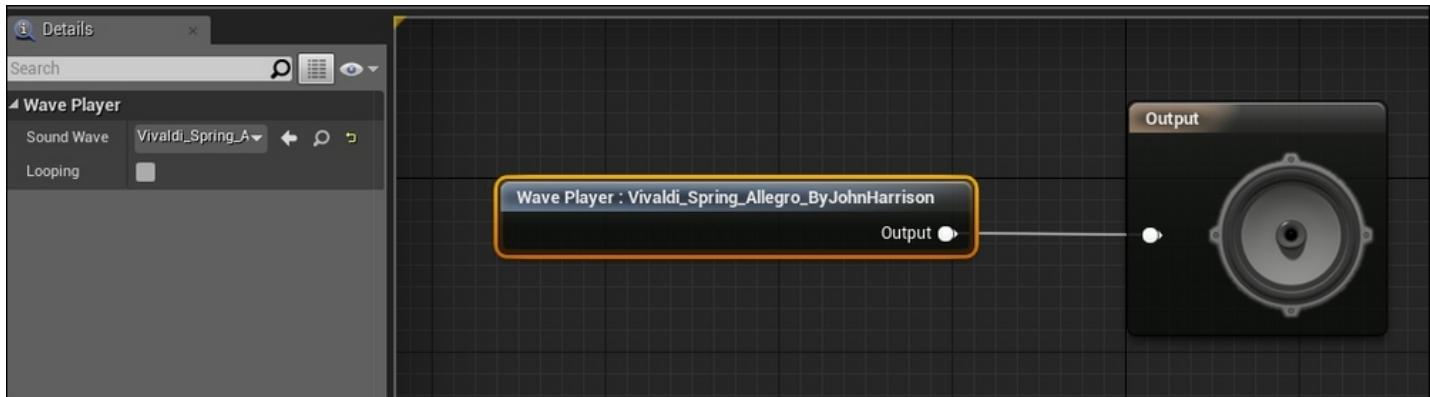
This screenshot shows you how the Vivaldi WAV file is successfully imported as a sound wave in the `Audio` folder with the WAV file settings:



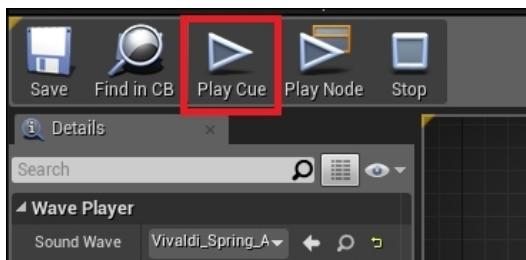
Next, create a Sound Cue for the Vivaldi sound wave that we have just imported. To recap, a Sound Cue is used to control the playback of the sound wave file. A sound wave file merely has the contents of the audio file. Right-click on the sound wave asset, as shown in this screenshot, and select **Create Cue** in the contextual menu:



Double-click on the newly created Sound Cue (which has a default name with the same name as the sound wave file with a `Cue` suffix). In the example here, it will be `Vivaldi_Spring_Allegro_ByJohnHarrison_Cue`. Double-click on this Cue to view the contents. The following screenshot shows the contents of `Vivaldi_Spring_Allegro_ByJohnHarrison_Cue`. The wave player output is connected directly to **Output**. This is the simplest connection for a Sound Cue where we input the wave to the **Output**.



Now, let's hear the sound we have imported. Within the Sound Cue Editor, look for the **Play Cue** button in the top-left corner of the editor. Take a look at the following screenshot for location of the button. After clicking the button, you would hear the music we have just imported. You have just successfully imported a custom wave file into Unreal. Now, let's transfer it to the game level.

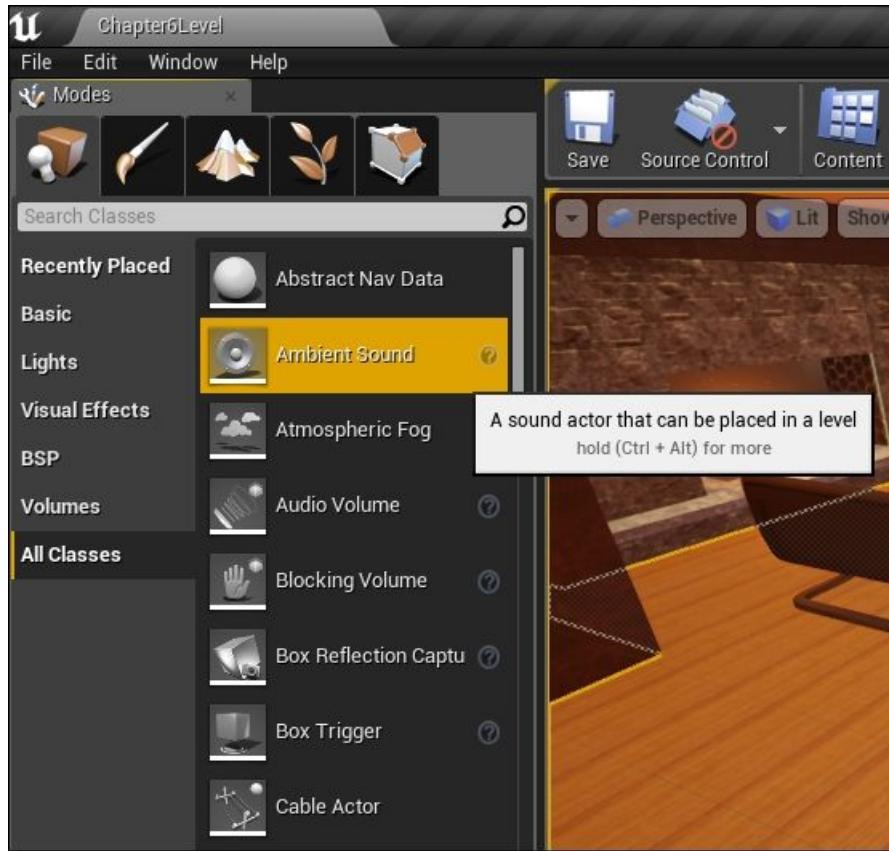




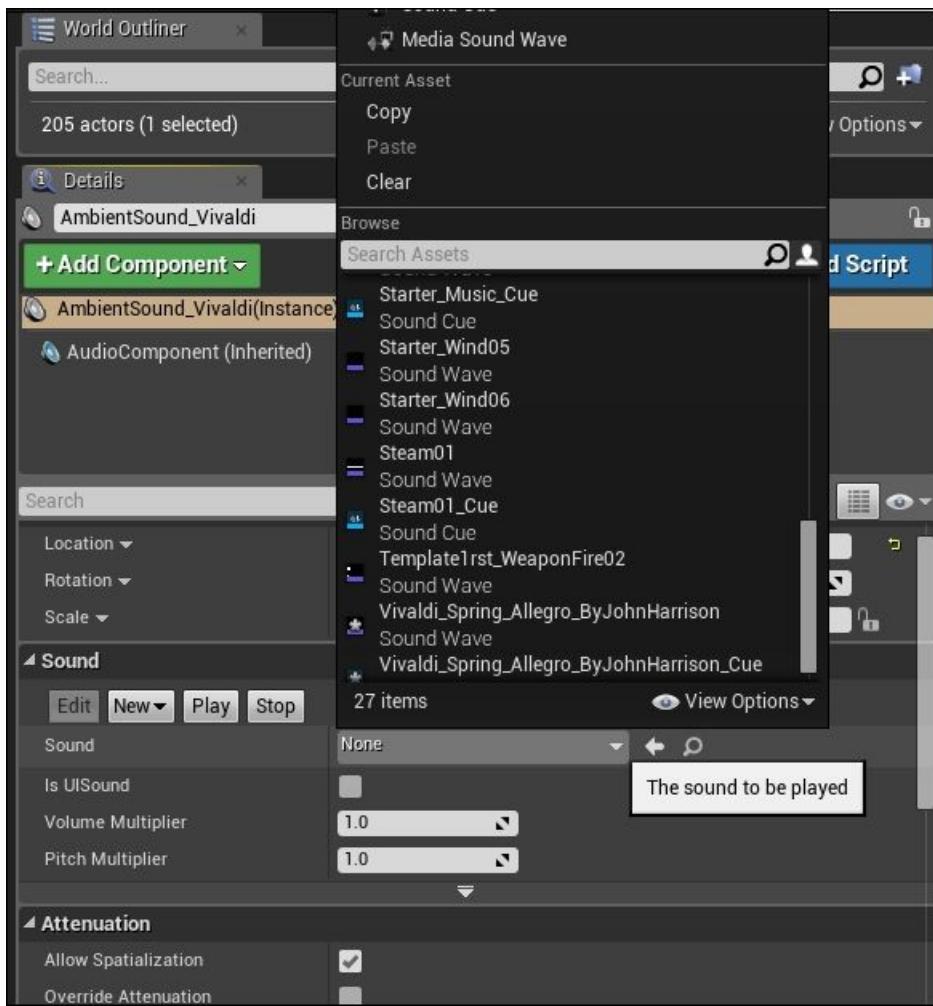
# Exercise – adding custom sounds to a level

In order to place sound in the level, you need to use the **Ambient Sound** node to associate it with a sound cue, which would, in turn, play the audio files.

To create an **Ambient Sound** node, go to **Modes | All Classes** , drag and drop **Ambient Sound** into the game level:



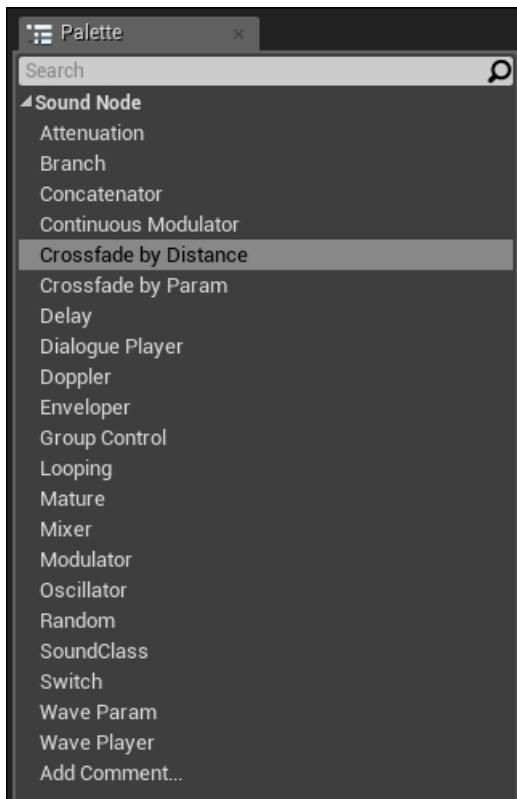
Click on the Ambient Sound Actor that you have just placed into the level, and rename it `AmbientSound_Vivaldi` . In the **Details** panel, scroll to the **Sound** section, click on the arrow next to **Sound** to display the sound assets that you have in the game level packages, as shown in the following screenshot. Select `Vivaldi_Spring_Allegro_ByJohnHarrison_Cue` .



Check whether you can still hear the music by clicking on the **Play** button in the **Details** panel of **AmbientSound\_Vivaldi**. Now, let's build the level and run it. Notice that the music plays when you start the level.

# Configuring the Sound Cue Editor

Double-click on **Vivaldi\_Spring\_Allegro\_ByJohnHarrison\_Cue** to open the Sound Cue Editor. Notice that on the right-hand side, there is **Palette** with a list of nodes, as shown in the following screenshot. These nodes can be used to control how the sounds are played or heard.



If you find that your sound design cannot be achieved using the nodes in the list, you can alternatively request for new nodes to be created via the UE4 source code.

# Summary

Both particles and sound are very interesting components of a game and require very specialized skills that are very apt for their design and creation. Particle system creators often have strong artistic and technical backgrounds; an artistic touch is needed to create suitable textures, and a technical ability helps to adjust distributions/values that create an appropriate overall effect. Audio engineers often have a strong music background. They are probably composers and musicians themselves with a passion for games.

In the first half of the chapter, we learned about what a particle system is. We learned how particle systems are used to create in-game effects, such as falling snow, rainfall, flames, fireworks, explosion effects, and much more. A particle system can efficiently render small moving fuzzy particles using textures through a combination of emitters. Each emitter has many configurable modules that can control properties, such as a spawn rate, lifetime, velocity, and the acceleration needed to create the required effect. In this chapter, we covered how to edit an existing fire explosion particle system, turn it into a fireplace effect, and place it in a living room. Through this example, we also went through some basic principles that could be applied to the particle system design process, and how to make minor adjustments to a few popular basic modules to create the effect we wanted.

The second half of the chapter covered how to include sounds in a level. We learned how sounds/music are conceptualized, created, recorded, and eventually, imported into the Unreal Editor. We also covered the audio format that the Unreal Editor currently supports, and a little explanation of each of the components is given to give you a better insight into sounds. Next, we went through a simple exercise to import an online audio file and get the music we have downloaded playing in the game level.

I hope you have gained a little more understanding about the creation process of the particle system and the audio effects that are needed for the games in this chapter. We will continue to improve our game level with a little terrain editing and also create cinematic effects in the next chapter.

# Chapter 7. Terrain and Cinematics

In this chapter, we will cover a few level-enhancing features. We will create some outdoor terrain for our level as well as add a short cinematic sequence at the start of the game level.

In this chapter, we will look at the following topics:

- Creating an outdoor terrain
- Adding a shortcut for a cinematic sequence at the beginning of the same level

## Introducing terrain manipulation

Terrain manipulation is needed when you want to create large natural landscape areas, such as mountainous or valley areas that are covered with foliage. This can be in the form of trees/grass, lakes, and rivers that are covered with rocks or snow, and so on. The Landscape tool in Unreal Engine 4 allows you to creatively design a variety of terrains for your game maps easily, while allowing the game to run at a reasonable frame rate.

When playing in a map that has large outdoor terrains, for example, maps with a large number of trees or many elevations, such as mountains, the effective frame rate is expected to be reduced due to an increase in the number of polygons that need to be rendered on the screen. Hence, being well-versed in landscaping so that polygon counts are kept under control is important to ensure that the map is actually playable. It is also good to bear in mind to make use of optimization techniques, such as LOD and fog to mask the distant places, which can give you a sense of unending open land. If you are planning to create an open world, you can also use the Procedural Foliage tool (available in Unreal 4.8 and higher versions) to spawn these features for you.

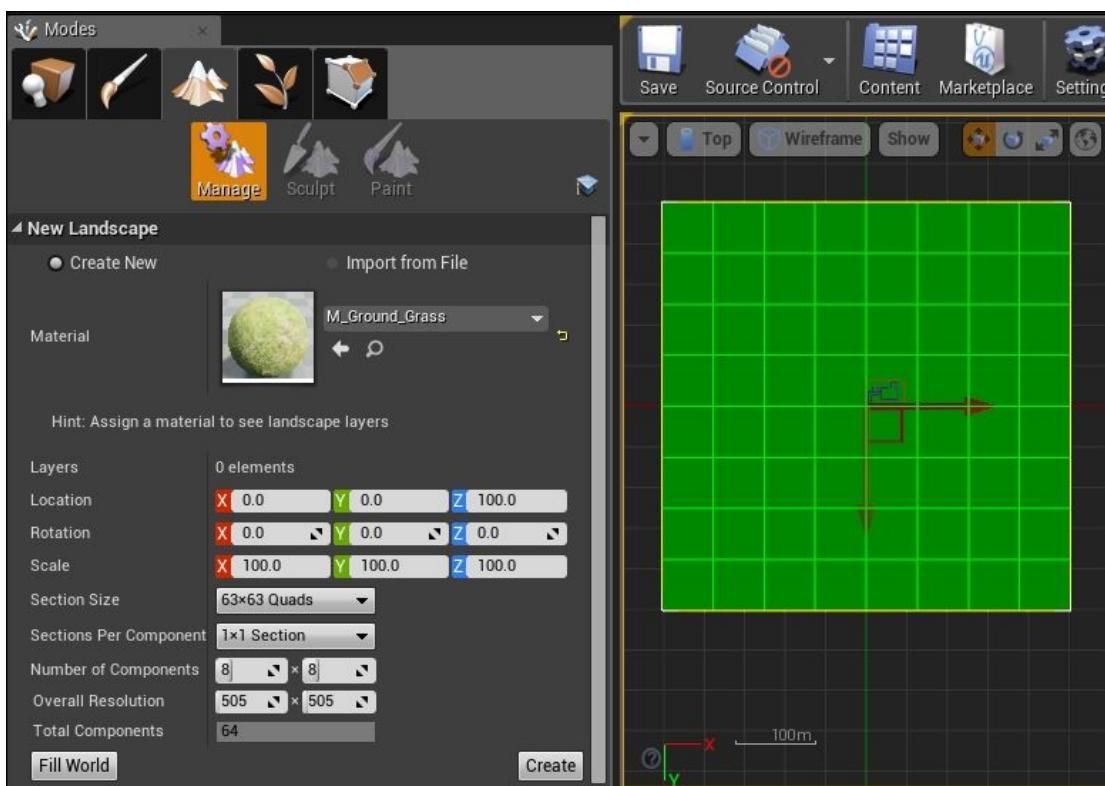
Let's get ourselves familiarized with the Unreal Landscaping tool and start creating some outdoor environments for our game level. We will learn how to perform simple contouring of the outdoor space with low hills, grass, and trees. Then, we will create a small pond in the area. For more accurate landscaping, we can import a height map to help us with the creation of the landscape.

### Exercise – creating hills using the Landscape tool

Let's perform the following steps to create hills using the Landscape tool:

1. Open Chapter6.umap and save it under Chapter7\_Terrain.umap .
2. Go to **Modes** , click on the Landscape tool (the icon looks like a mountain) and then click on **Manage** .
3. Select **Create New** (the other option here is to make use of a height map, which we will cover later in the chapter).
4. To select a Material, you can click on the search icon and type **M\_Ground\_Grass** , or go to **Content Browser | Content | Materials** , select **M\_Ground\_Grass** , and click on the arrow next to **Landscape Material** to assign the material.
5. For this example, we are going to leave all of the landscape settings at their default values that are listed, as follows. The next section will explain the options for the rest of the values in further detail:
  - **Scale** : X = 100 Y = 100 Z = 100
  - **Section Size** : 63 x 63 quads
  - **Section Per Component** : 1 x 1 section
  - **Number of Components** : 8 x 8
  - **Overall Resolution** : 505 x 505

The following screenshot shows the top view of the grass landscape that we have created. Notice the 64 green squares. You will need to switch to the **Top** view to view it.



Now, we'll switch over to the **Perspective** view. The grass landscape seems like it's covering half the house. Take a look at the following screenshot:



Note that if we had created the landscape on an empty map, we would not have this issue, as we would have built the house on the landscape grass instead. So, we have to perform an additional step here to move the landscape grass under the house so that we do not have a house that's submerged under the grass. You need to select **Landscape** and **LandscapeGizmoActiveActor** from **World Outliner**, as shown on the right-hand side of the following screenshot. Remember to switch **Mode** back to **Place**, instead of the **Landscape** we were in to create the grass. The **Place** mode allows the translation/rotation of the selected object. Move the grass to just below the house, as shown in the following screenshot:



## Note

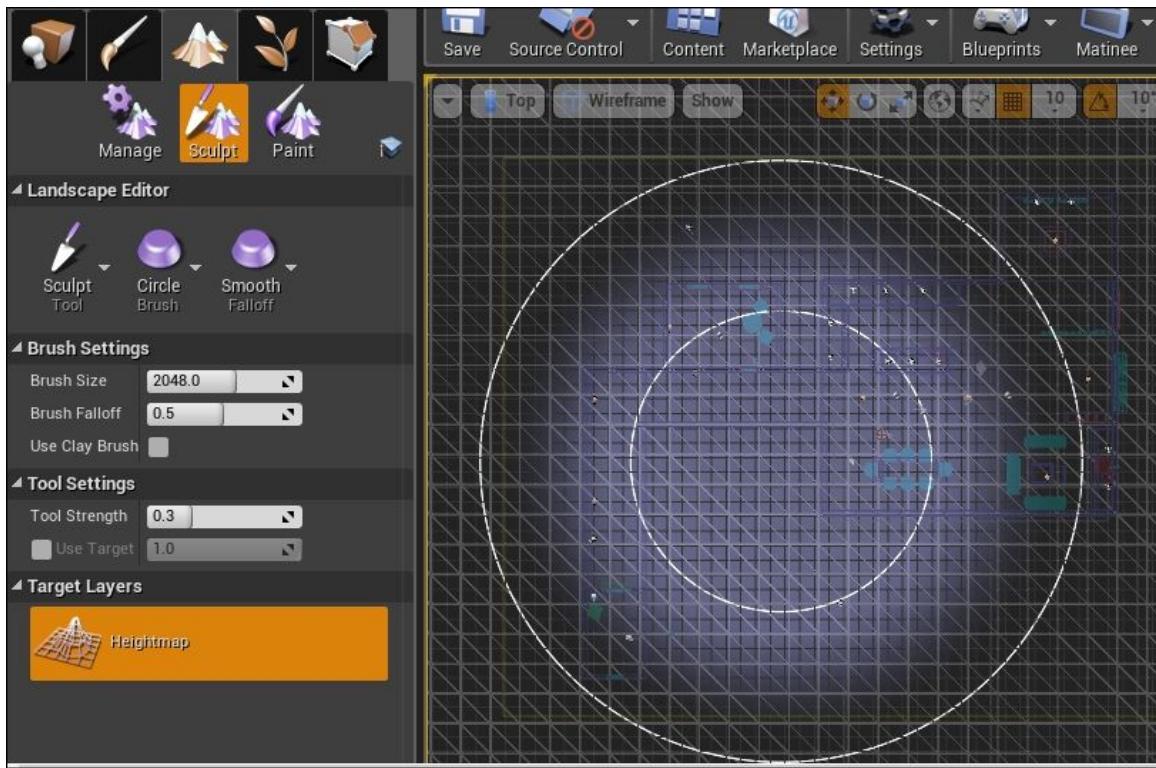
Note that this step is performed because we add the landscape grass after we've built the house.

Now, we are ready to sculpt this flat land into some terrain. Go to **Modes** | **Landscape** | **Sculpt** again. Use the Sculpt tool, **Circle Brush**, and the **Smooth Falloff** combination, as shown in the upcoming screenshot. The default settings should be as follows:

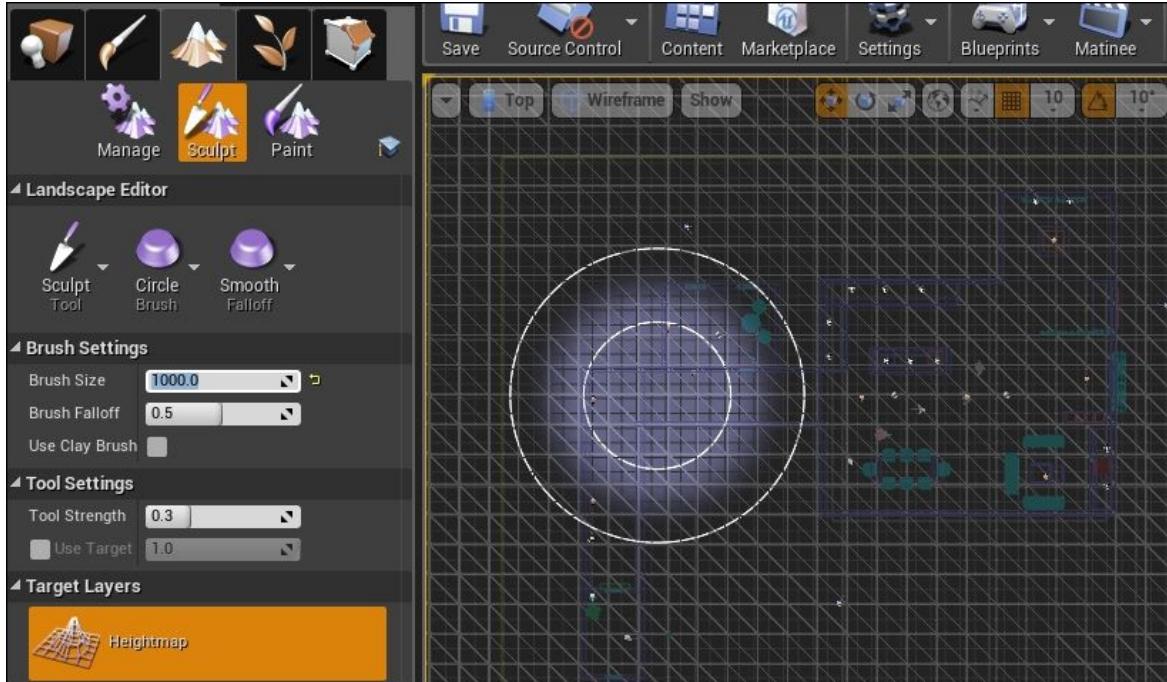
- **Brush Size : 2048**
- **Brush Falloff : 0.5**

- **Tool Strength : 0.3**

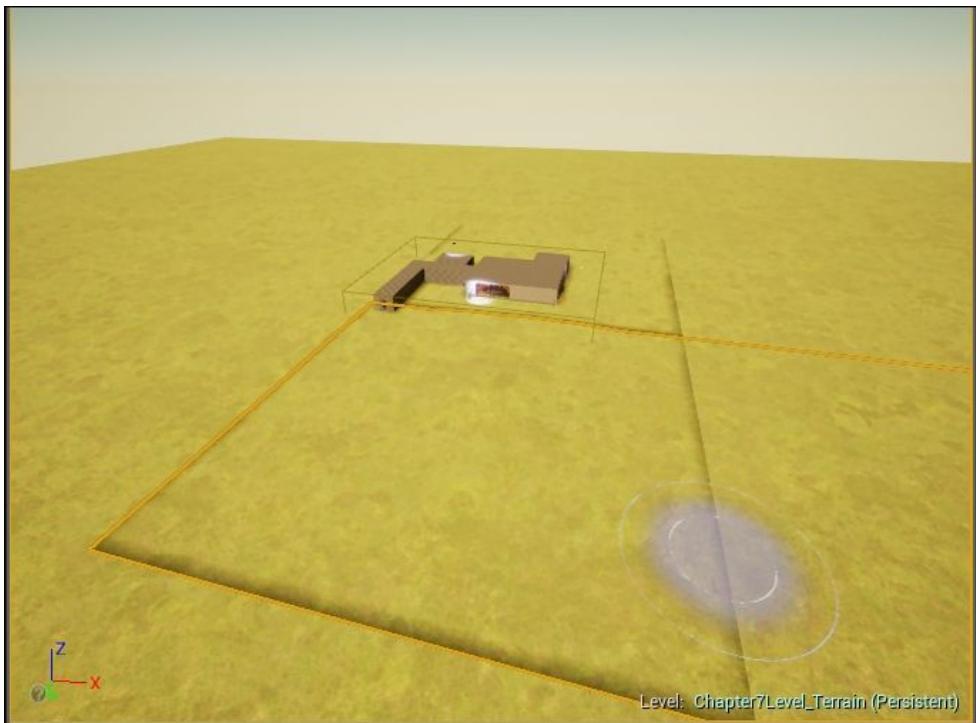
To illustrate the size of the 2048 brush, I have switched to the **Top** view:



When **Brush Size** is set to **1000**, the brush radius is reduced, as shown in the following screenshot:

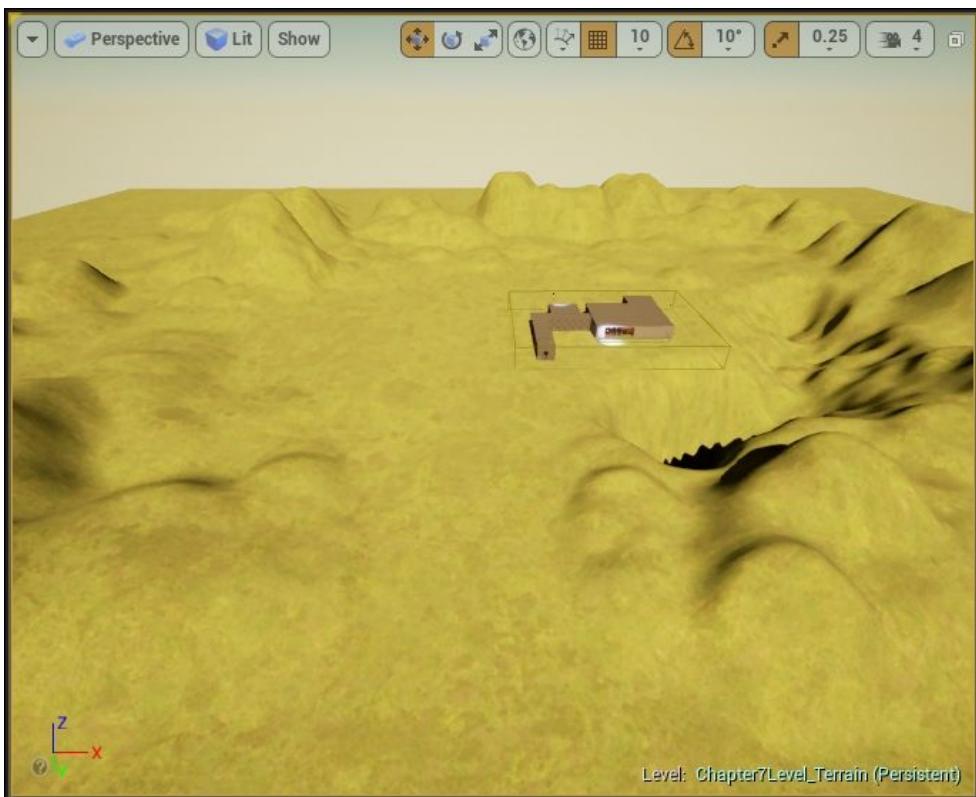


Now that we have an idea about the difference in radii, we will switch back to the **Perspective** view. Position your working screen to a slightly angled top perspective view, as shown in the following screenshot. Set **Brush size** to **1000** and **Tool Strength** to **0.4**:

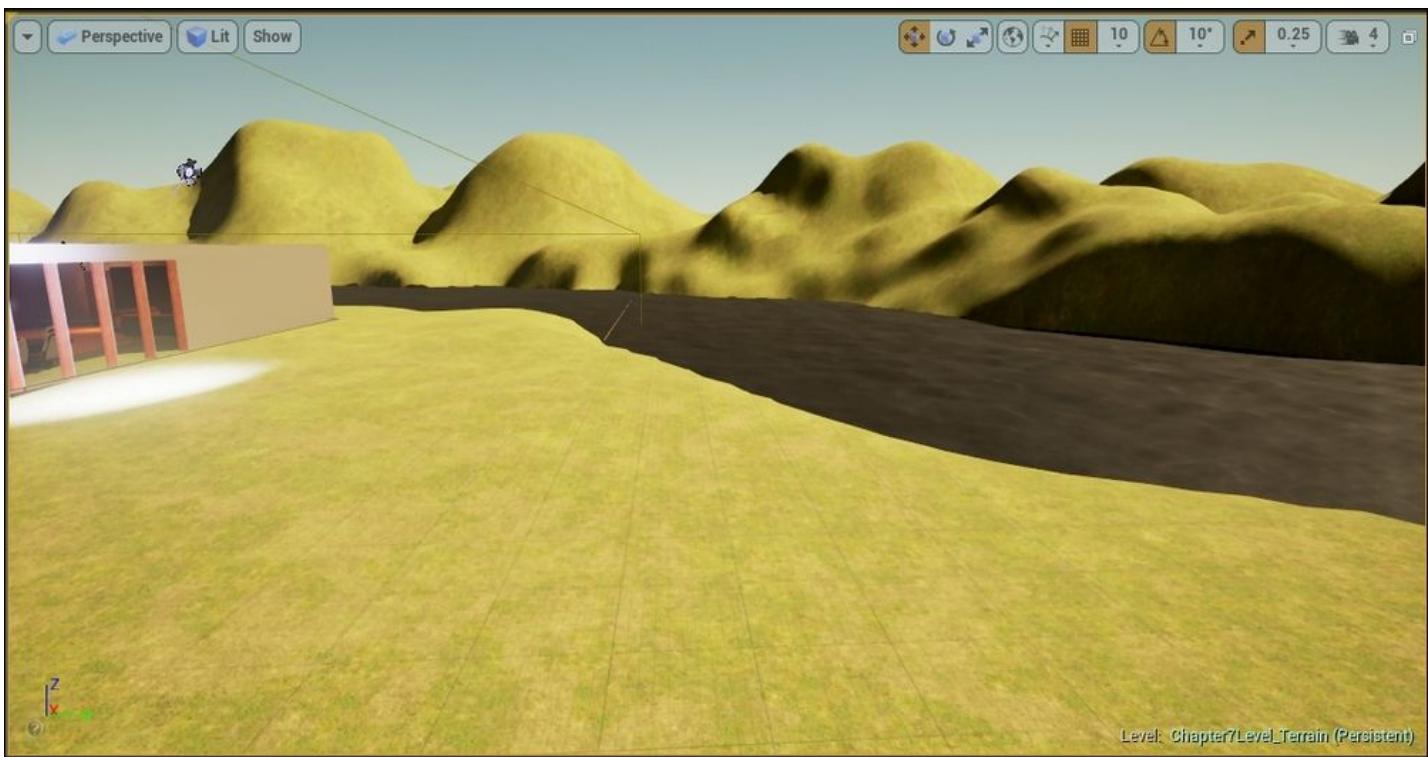


Start by creating low hills around the house by clicking on the area around the house. I used a mix between a brush size of 1000 and 2048.

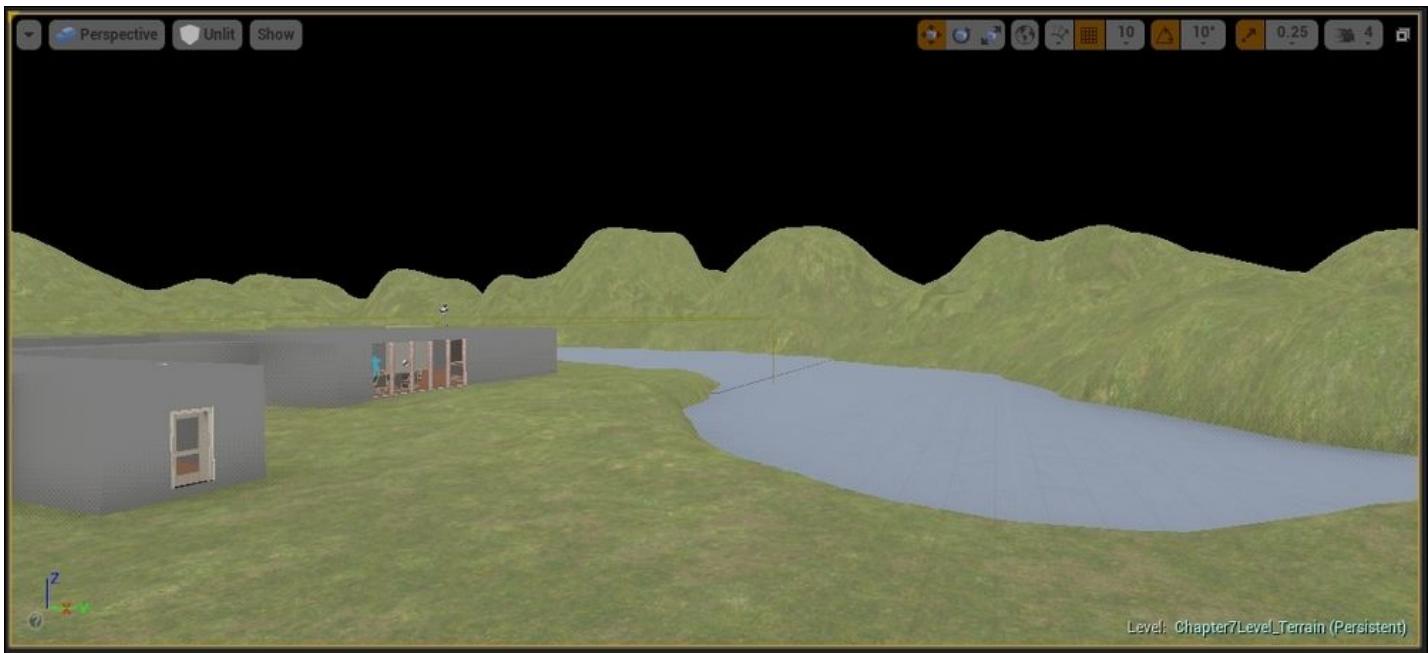
The following screenshot shows how the area looked after I worked on it for a bit. Note that the area in front of the wide windows where I created a depression. This is achieved by holding *Ctrl* and then clicking on the area. This depression will take the form of a lake in front of the dining area.



Create two box BSPs to fill up the depressed area. Apply the Lake Water material to the box BSPs. The following screenshot shows the same area with the box BSPs put in place. Use the Translation tool to keep both BSP areas on the same ground level at the location of the depression.



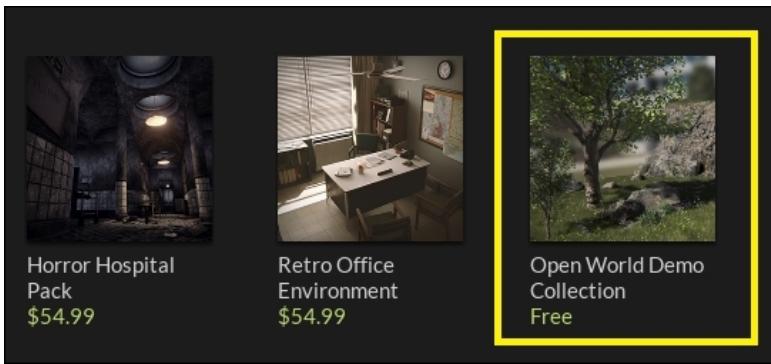
Next, I touched up the external area of the house. Use the **Unlit** mode to help you see the house better. This screenshot shows you how the house and area around it look after touching them up with the **MyGreyWall** material:



Go back to the **Lit** mode, build the level, and then take a look at it. Adjust any lighting in the map so that it's lit up appropriately. Rebuild until you are satisfied with what you get.

Add trees and plants to make the area a little more realistic. I have downloaded a package from Marketplace that has some foliage to help me with this.

Go to Marketplace on the Unreal Start Page. Under **Environments**, look for free downloadable content called **Open World Demo Collection**. The following screenshot shows free **Open World Demo Collection** in Marketplace. After downloading the package, add it to the project that you are working on.



We now have a basic outdoor terrain for our map.

## Landscape creation options

After going through the preceding exercise, you now have a good idea about how landscaping in Unreal Engine 4 fundamentally functions. In this section, we will add to the skills we have acquired so far and learn how to adjust or utilize features/functions of the Landscaping tool that is available to us.

### Multiple landscapes

It is possible to have multiple landscapes in the same map. This allows you to split the creation process into different layers. If you have more than one landscape in the map, you will need to select a layer before modifying it.

### Using custom material

You can import any material you want to use for the landscape; you can make your own grass, crops, sand texture, and so on. Since the custom material is mostly used for large areas of the map, it is good to bear in mind that you need to keep the material repeatable and optimized.

### Importing height maps and layers

Why do we use height maps in landscaping? These allow a quicker and more precise way to create elevations/troughs in the Unreal Editor. For example, we can use a height map to store elevation information for a mountain that is 3000m in height and of a certain diameter. When we import the height map, the terrain is automatically shaped according to it. It is definitely a time-saving method that helps us create more precise landscape features without having to click, click, click to sculpt.

Height maps and layers can first be created externally using common tools, such as Photoshop, World Machine, ZBrush, and Mudbox by artists. Detailed instructions need to be followed to ensure the successful importation of the height map. This can be found in the Unreal Engine 4 documentation at <https://docs.unrealengine.com/latest/INT/Engine/Landscape/Custom/index.html>.

### Scale

The **Scale** settings determine the scaling of the landscape. We have used X: 100 and Y: 100 to give us the area of the land that this landscape will cover. The Z value is kept as 100 to provide some height to create elevation.

### The number of components

A component is the basic unit for rendering and culling. There is a fixed cost that's associated with the overall number of components; hence, it is capped at 32 x 32. Going beyond this value would affect the performance of your game level.

### Section Size

**Section Size** determines how large each section is. It determines how the landscape is divided up. Large sections mean fewer overall components because the pie is divided into larger chunks. Fewer chunks to manage indicate a lower overall CPU cost.

However, a large section is not as effective when managing the LOD as compared to a smaller section. When there are smaller sections, we also get smaller component sizes (when the pie is of the same size, cutting it into smaller chunks indicates that you have less on your plate if you take one chunk). Since components are the basic unit used for culling and rendering, this means quicker responses to LOD changes due to the reduced area. LOD determines the number of vertices that need to be calculated. If LOD is more effective, we have fewer calculations to do, and the CPU cost is more optimized with smaller sections.

The catch here is balancing the size of the sections to avoid having too many components to go through and too few components might result in poor LOD management.

### Note

#### Sections Per Component

You have options ranging from 1 x 1 or 2 x 2 sections per component. What this means is that you have the option of having either one or four sections in each component. Since a component is the most basic unit in rendering and culling, for the 1 x 1 section, you can have one section rendered at the same time. For 2 x 2 sections per component, you can have four sections rendered at the same time. To limit the number of calculations needed to render a component, the size of each section should not be too large.

# Introducing cinematics

Cinematics were developed largely for motion pictures, films, and movies. Today, we apply cinematic techniques to non-interactive game sequences, known as **cut scenes**, to enhance the gaming experience. The overall gaming experience has to be designed with cut scenes in mind as they usually fulfil certain game design objectives. These objectives are often slotted in between gameplay to enrich the storytelling experience in games.

Very much like shooting a movie, we would need to decide what kind of shots need to be taken, which angles to shoot from, how much zooming is needed, how many cameras to use, and what path the camera needs to take in order to develop a motion picture sequence of our object/objects of focus. The techniques employed to create this clip are known as **cinematic techniques**.

So, in this chapter, we will first go through a few key objectives that explain why cinematics are needed in games, and you learn a couple of simple cinematic techniques that we could use. You will also learn about the tools that Unreal Engine 4 offers to apply the techniques we have learned in order to create appropriate cinematic sequences for your game.

Cinematic techniques are created by cinematic experts who focus on creating cut scenes for your games. Alternatively, you could also engage a cinematic creation contracting company to get this done for you professionally.

# Why do we need cut scenes?

When a game is designed, a fair amount of the game designing time is put into designing how players interact with the objects in the game and how this interaction can be made fun. The interactive portion of the game needs to be supplemented and cut scenes can help fill the gaps.

Cut scenes can be employed in games to help designers tell a story when you are playing the game. This technique can be employed before the game begins to draw the players into the mission itself and justifies why a mission has to be accomplished for the player. This helps the player to understand the storyline, create meaning for their actions, and draw the player into the game.

Another objective of cut scenes can be to highlight key areas in the game in order to give the players a glimpse of what to expect and provide subtle hints to successfully win the game. This information would be useful, especially in difficult to beat game levels or when the player is meeting the chief monster in the game.

Game designers also sometimes use cut scenes to reward players after a difficult battle. They amplify the effect of their success and play out the happy ending of their win in order to create positive emotions in the players. I am sure that there are endless creative ways to utilize cut scenes in games and how we could positively include them to enhance the gaming experience.

However, it is necessary to ensure that the use of cut scenes is justified well because cut scenes actually take the control of the game away from the player. Games are expected to be interactive, and we do not want to convert this into a passive multimedia experience when there are too many cut scenes.

Keeping these basic game design objectives in mind, let's now explore some technical cinematic fundamentals that will provide you the groundwork to design your own cinematics in games.

# Cinematic techniques

The camera is the main tool that's used to create effects for cinematics. You can achieve various cinematic effects by adjusting the camera functions and finding/moving the camera to a good spot to capture a significant key object(s) of interest. This section will provide some technical terms that you can use to describe to your coworker/contractor how a particular cinematic sequence should be recorded.

## Adjusted camera functions

Here are some commonly used functions that you can adjust on a camera to capture a scene.

### Zoom

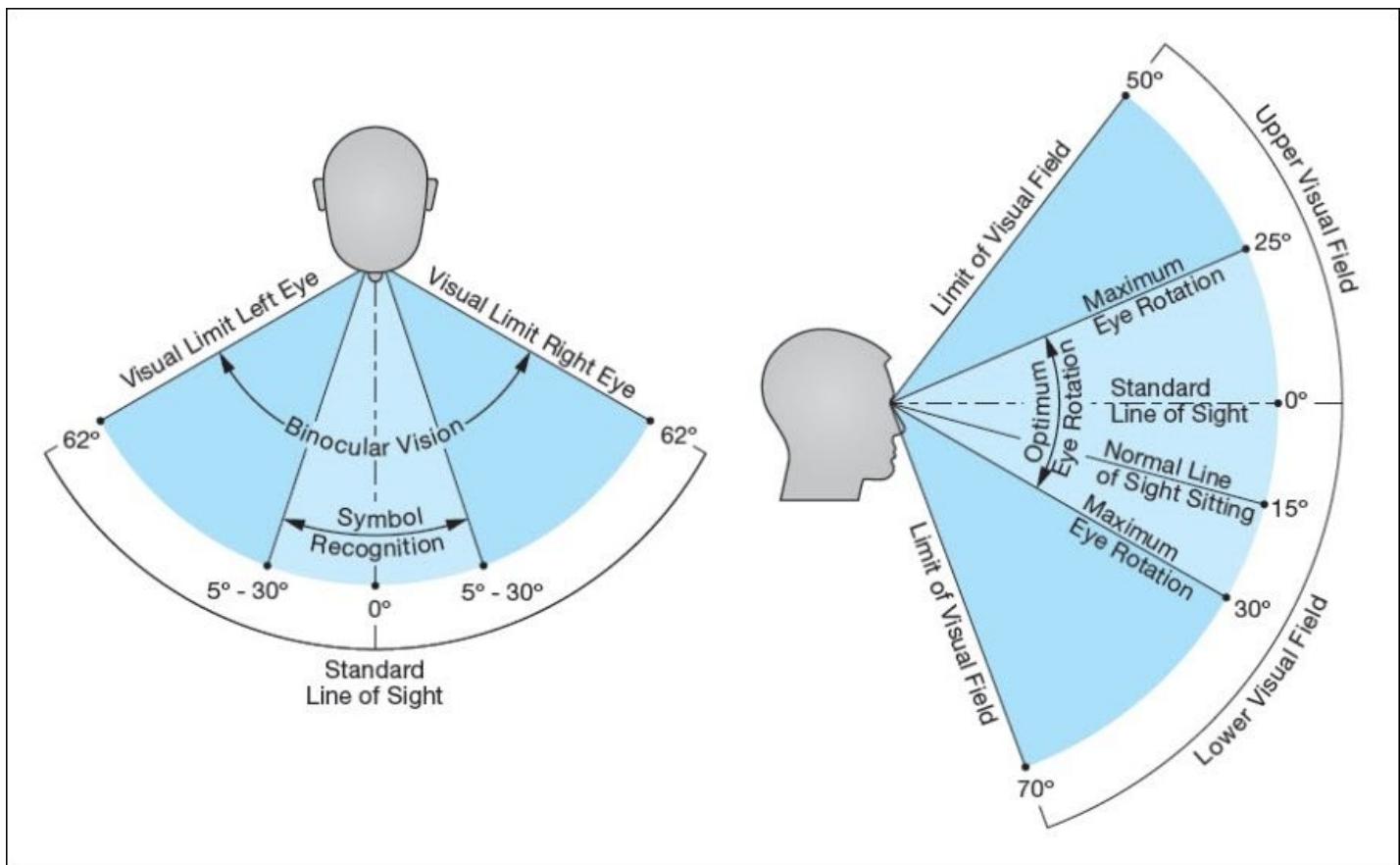
Zooming in on an object gives you a closer view on the object; providing more details about it. Zooming out takes your view further away from the object; it provides a perspective for the object with regard to its surroundings.

Zooming is achieved by adjusting the focal length of the camera lens; the camera itself stays in the same position.

### Field of view

**Field of view ( FOV )** is the area that is visible from a particular position and orientation in space. FOV for a camera is dependent on the lens and can be expressed as  $FOV = 2 \arctan(SensorSize/2f)$ , where  $f$  is the focal length.

For humans, FOV is the area that we can see without moving our head. The horizontal FOV kind of ends at the outer corner of the eye, as shown in the following image, which is about 62 degrees to the left-hand side and right-hand side (source: <http://buildmedia.com/what-are-survey-accurate-visual-simulations/> ):



What this means is that whatever is outside this FOV is not visible to the entity.

### Depth of field

**Depth of field ( DOF )** is best expressed as a photo, such as the following one, where only the object of interest is very sharp and anything behind it blurred. In the following image (source: <http://vegnews.com/articles/page.do?catId=2&pageId=2125> ), the gyoza/dumplings appear sharp and beyond these, the bowl/bottle is blurred. The small DOF in the photo allows the foreground (gyoza) to be emphasized and the background to be de-emphasized. This is a very good technique to bring visual attention to objects of interest in photography as well as in cinematics.



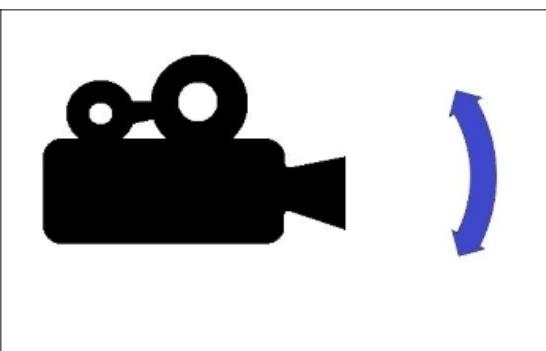
DOF is also known to provide an effective focus range. The method to determine this range is to measure the distance between the closest object and farthest object in a scene that appears to be sharp in an image. Although a lens is made to focus on one distance at a time, the gradual decrease in sharpness is difficult to perceive under normal viewing conditions.

## Camera movement

In filming, the camera is positioned at different angles and locations, and the camera moves with the actor/vehicle and so on. This camera movement can be described using some of the terms here.

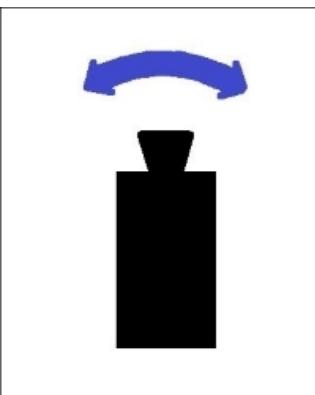
### Tilt

The camera is moved in a similar way to how you nod your head. The camera is pivoted at a fixed spot, and turning it up/down is known as **tilting**. The following figure shows the side view of the camera with arrows illustrating the tilting:



### Pan

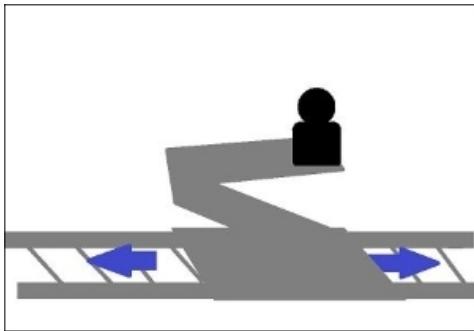
The camera is moved in a similar way to how you turn your head to look to the left-hand side and the right-hand side. The camera is pivoted at a fixed spot and turning it to the left-hand side/right-hand side is known as **panning**. This figure shows the top view of the camera with arrows demonstrating how panning works:



### Dolly/track/truck

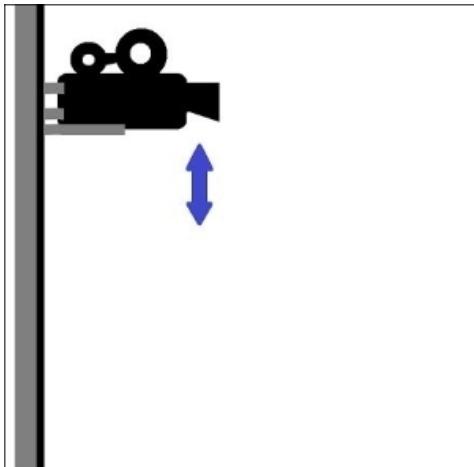
A dolly moves the entire camera toward or away from the object. It is quite similar to zooming in/out since you also going closer/further to the object, except that dollying moves the camera along a path toward/away from the object.

Trucking moves the camera sideways, that is, to the left-hand side or right-hand, along a track. Trucking is often confused with panning. The entire camera moves in trucking, but in panning, the camera stays in a fixed location and only the lens is swept to the left-hand side/right-hand side. Tracking is a specific form of trucking as it follows an object of interest in parallel. The following figure shows the back view of a camera dollying along a path:



## Pedestal

Pedestal is the moving of the camera up and down a vertical track. The following figure illustrates the camera moving up and down a vertical track:



## Capturing a scene

When capturing a scene, the overall scene is what matters most. You need to keep certain things in mind, such as what comprises the scene and its lighting; what you select determines how impactful the cut scene is. Here are a few factors that need to be addressed when composing a good cut scene.

### Lighting

Light affects how a scene shows up in photo/cut scene. We need to have the right lighting in place to capture the mood of the scene.

### Framing

Framing decides how the shot needs to be taken. Everything in the frame is important and you should pay attention to everything that is within the frame. How each shot transitions to the next also needs to be considered when creating a cut scene.

#### Some framing rules

The framing rules are as follows:

- Make sure the horizontals are level in the frame and the verticals are straight up along the frame.
- The Rule of Thirds. This rule divides the frame into nine sections. The points of interest should occur at one-third or two-thirds of the way up (or across) the frame rather than in the center. For example, the sky takes up approx. Two-thirds of this frame.
- Strategic empty spaces are provided in front, above, or behind the subject to allow space for the subject to move into/look into.
- Avoid having half an object captured in the frame.

### Shot types

Here are some terms used to describe shots that can be taken for the frame:

- **Extreme Wide Shot (EWS) / Extreme Long Shot (ELS)** : This shot puts the subject into the environment. The shot is taken from a distance so that the environment around the subject can be seen. This type of a shot is very often used to establish a scene.
- **Wide Shot (WS) / Long Shot (LS)** : In a wide or long shot, the subject takes up the full frame. The subject is in the frame entirely with little space around it.
- **Medium Shot (MS)** : The medium shot has more of the subject in the frame and less of the environment.
- **Close Up Shot (CU)** : The subject covers approximately half the frame. This increases the focus on the subject.
- **Extreme Close Up Shot (ECU)** : The camera focuses on an important part of the subject.

## Shot plan

This is a plan that describes how the scene will be captured. It also describes how many cameras to use, the sequence in which the cameras come on, and the kind of shots that need to be taken in order to play out the required effect for the scene.

## Getting familiar with the Unreal Matinee Editor

The Unreal Matinee Editor is similar to nonlinear video editors, so it is quite easy to pick up if you already have experience using software such as Adobe Flash. Creating keyframes for cameras and moving them along paths combined with modifying camera properties creates the matinee/cut scene for games. Additionally, you can also make or convert static objects to become dynamic and then animate them using this Matinee Editor.

## **Exercise – creating a simple matinee sequence**

Now, let's get hands-on and create a simple matinee sequence for your game. The plan is to showcase the area that we created at the beginning of the game. We will start with an extreme wide shot taken from the front of the house. We will use the dolly to take the camera toward the large windows in the dining area, into the kitchen area, and then the fireplace. Then, using the second camera, from the corner of the room, we will move toward a running guy and focus on his face.

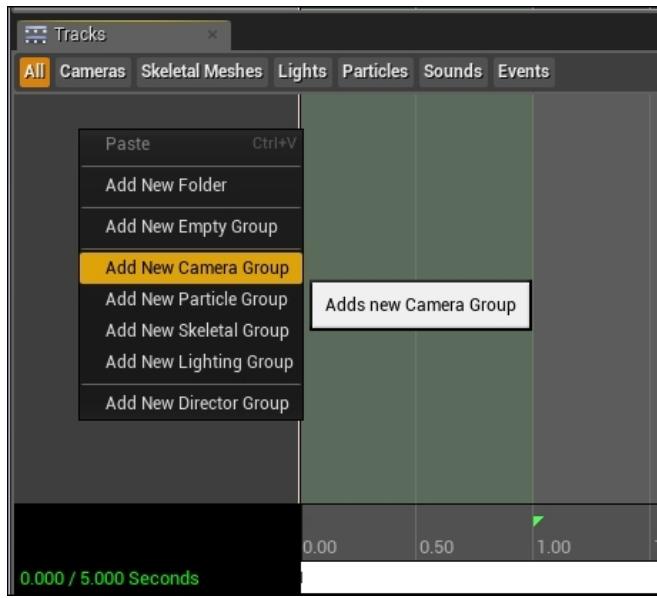
Create a new matinee sequence from the ribbon, as shown in the following screenshot. Click on **Matinee** and select **Add Matinee**:



This opens up the Matinee Editor, as shown in the following screenshot:



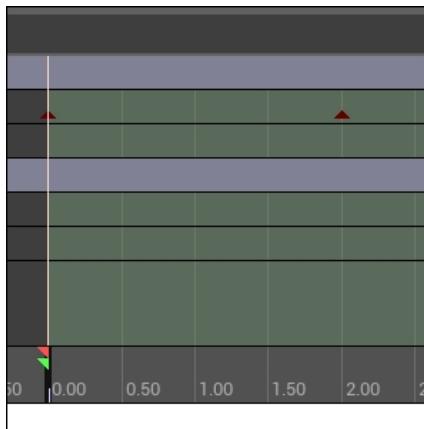
To create the first camera, we will right-click on the **Tracks** area and select **Add New Camera Group**:



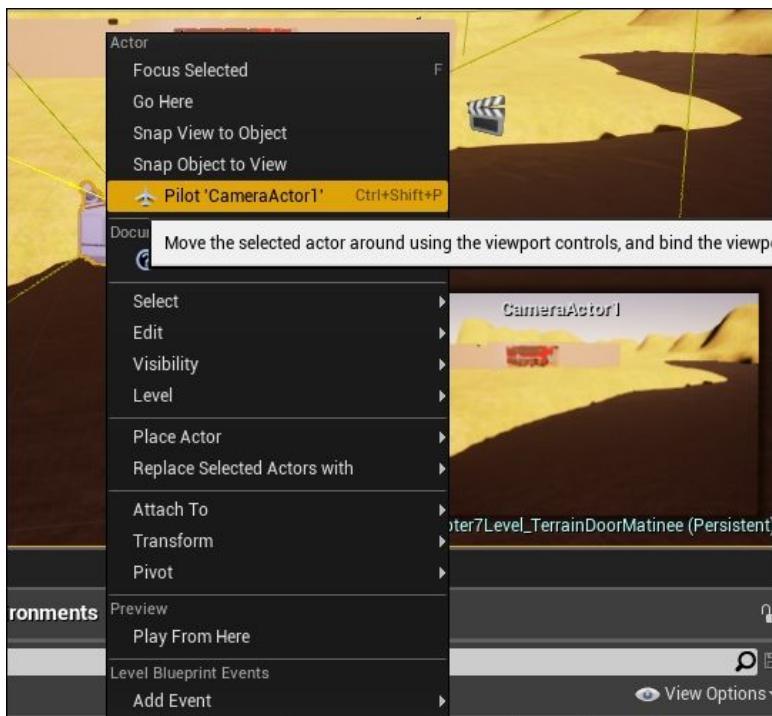
Going back to the map, you can see a small window in the corner of the map that shows what the camera is looking at. This screenshot shows where our first shot starts:



To create the next key where the next shot has to be taken, go back to the **Camera1** track, click on the small red arrow at 0.0 in the **Movement** row, and hit *Enter*. This duplicates the key. Press *Ctrl* and click and drag the red arrow to 2.00. This screenshot shows how to do it correctly:



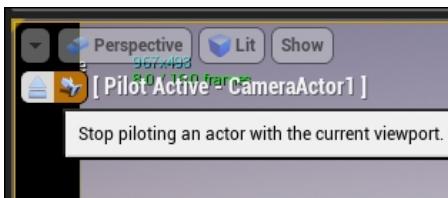
Now, click on the red arrow at 2.00 and go back to **Camera1** in the map. Right-click on it and select **Pilot 'Camera Actor1'**, as shown in this screenshot:



Move the viewport to the position you want to have the second keyframe in. This screenshot shows the position of the second keyframe camera:



When the viewport is positioned, as shown in the preceding screenshot, click on the icon in the top-left corner of the viewport to stop the pilot mode in order to fix the keyframe here. The location of the icon is shown here:



Following the shot plan we decided on, I have moved **Camera1** along the path up to the fireplace. To add the second camera, repeat the steps to create a new camera group and name the new camera as **Camera2**.

Now, move the first keyframe to the end of Camera1's final keyframe timeline. For me, this is set at **8.50s**; I moved the camera to the corner of the room, as shown in the following screenshot:



Repeat the steps to create keyframes for **Camera2**, move it along the path toward the running man, and then focus on the running man's face.

Now, we have two cameras that need to be told which one is playing at which part along the timeline. To do so, we need to create a new director group. The director group will dictate which camera is on air and what will be showing on screen. Go back to **Tracks** in the Matinee Editor. Right-click and select **Add New Director Group**, as shown in this screenshot:



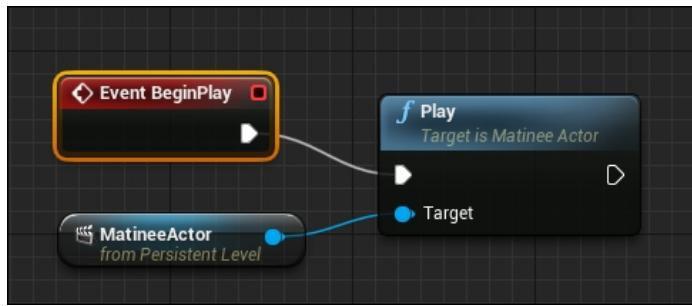
This creates a **Director** track above the camera tracks. Select the newly added **Director** track at 0.00, go to the ribbon at the top, and select **Add Key**, as shown in this screenshot:



The contextual menu will request that you select **Camera1** or **Camera2**. In this case, select **Camera1**. This fills up the entire duration of the cinematics. To create a key at 8.50s where **Camera1** and **Camera2** overlap, click on the **Director** track again and select **Add Key**. This time round, select **Camera2**. Move this key to **8.50**. This screenshot shows where the cameras are set up so that they can play correctly:



Finally, we are ready to play the cut scene. To tell the game to play the cut scene when starting the game, we need to use Blueprint. I hope you still remember how to use the Blueprint Editor. Click and open the Level Blueprint. Add the **Event BeginPlay** node and right-click and search for **Play**. Select the **Play Matinee Actor** option and link the nodes, as shown in the following screenshot. Now, save and play the level. You will see the entire matinee play before you control the player in the level.



# Summary

We covered terrain creation and matinee creation in this chapter. I hope you were able to enhance the game level with the new skills we explored.

Terrain manipulation covers large areas of a map; hence, we also went through the factors that affect the playability of the map. We also went through a simple exercise to create the outdoor terrain of our map with some hills and a lake.

Matinee creation involves a lot more technical planning before we start playing around with the editor itself. The use of the editor is pretty simple as it is similar to current video editors in the market. The techniques to create good cinematics were covered to help you understand their backgrounds a little better.

This is the last chapter of the book and the final summary. I sincerely hope that you enjoyed reading this book and had fun playing around with Unreal Engine 4. Lastly, I would like to wish you all the best in creating your own games. Do keep at it; there is always more to learn and other new tools out there to help you create what you want. I am sure that you love creating games; if not, you would not have survived this boring book right to the end. This book only serves to introduce you to the world of game development and shows you the basic tools to create a game using Unreal Engine. The rest of this journey is now left to you to create a game that is fun. Good luck! Don't forget to drop me a note to let me know about the games you create in the future. I am waiting to hear from you.

## **Part 2. Module 2**

***Unreal Engine Game Development Cookbook***

*Over 40 recipes to accelerate the process of learning game design and solving development problems using Unreal Engine*

# Chapter 1. Getting Acquainted with the UE4 Interface

In this chapter, we'll cover the following recipes:

- Installing UE4 and folder structure
- UI overview
- Navigating the viewport
- The Content Browser overview
- Importing your own content

## Introduction

UE4, created by Epic Games, is a robust game engine that contains several different game development tools, which can create any kind of game imaginable, with many areas for specialization. It would be good for newcomers to the Engine to first have a basic understanding of what the entire Unreal Engine consists of and then dive into the different areas that they are interested in.

These first recipes may seem a bit difficult for those who are unfamiliar with game development, but after a short period of time, it will become second nature to them. In this chapter, readers will gain some fundamental knowledge and have some awareness that will help and prepare them for the upcoming chapters.

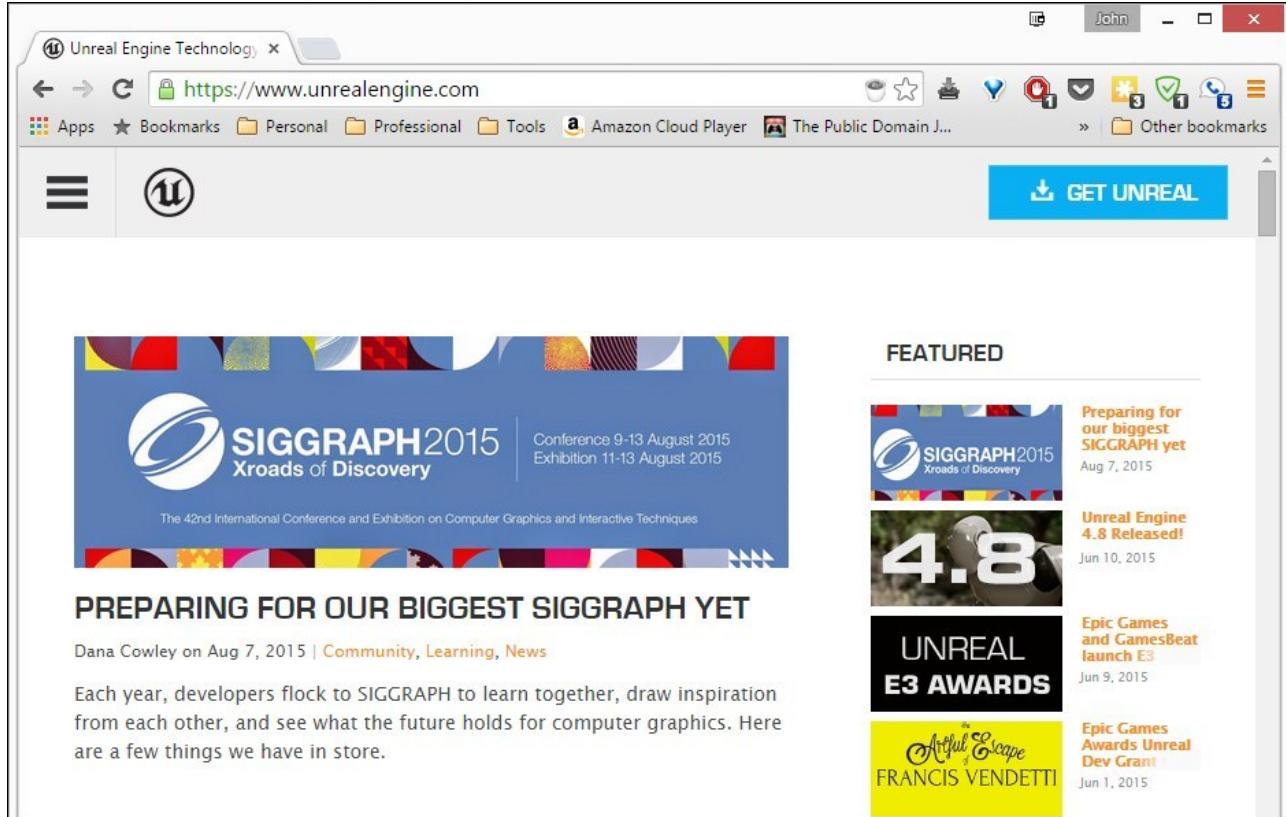
# Installing UE4 and folder structure

Now, in order to use UE4 at all, we have to have it installed on our computer. Even after we install it, we may also want an overview of what it is. In case you haven't installed it yet, here is how you can do this.

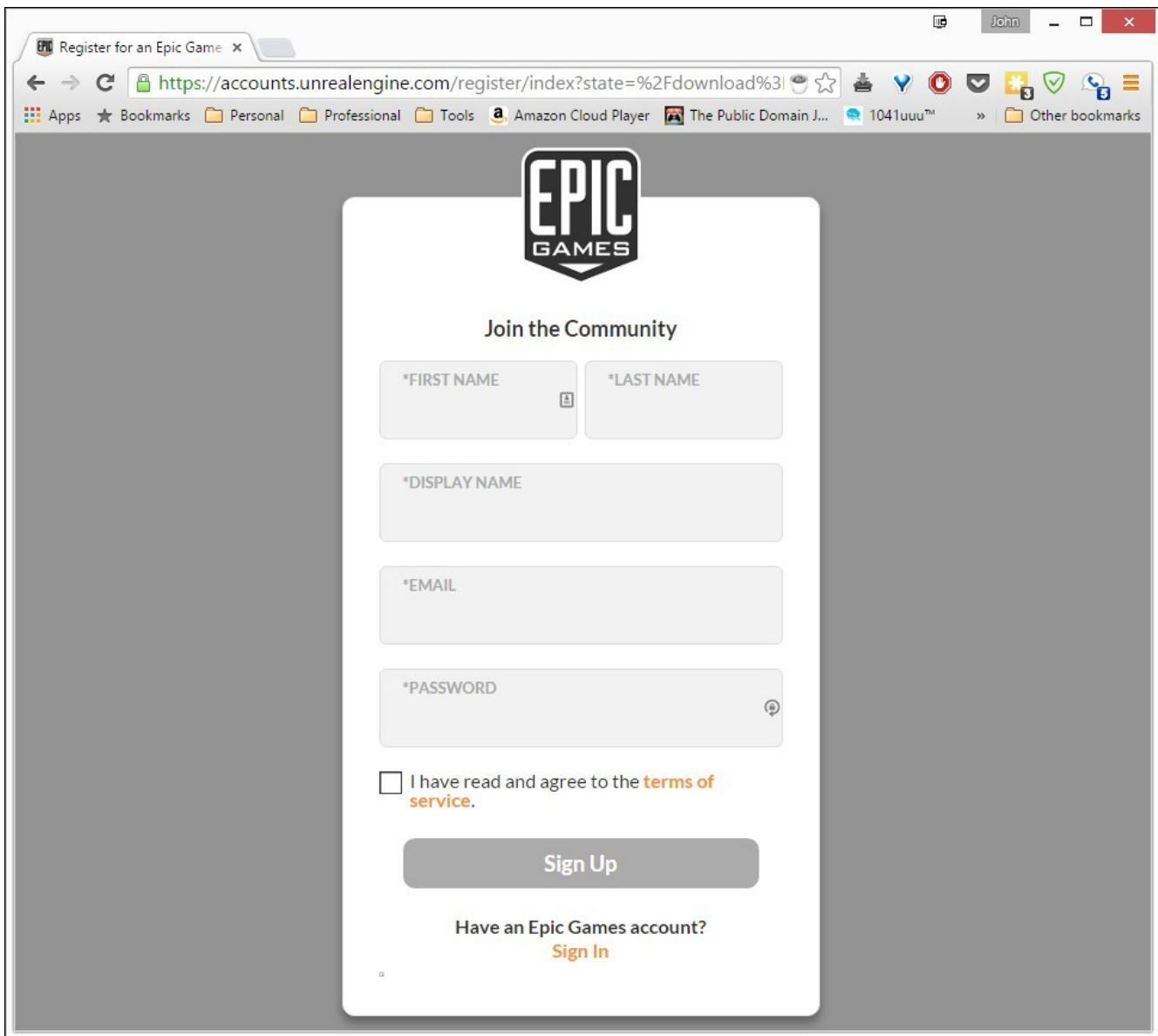
## How to do it...

Now that we have our project set up, let's get started with creating our player:

1. In the web browser of your choice, go to <http://unrealengine.com>.



2. Once on this page, click on the light blue **GET UNREAL** button.



- Once on the **Join the Community** page, fill out the information and create your account!

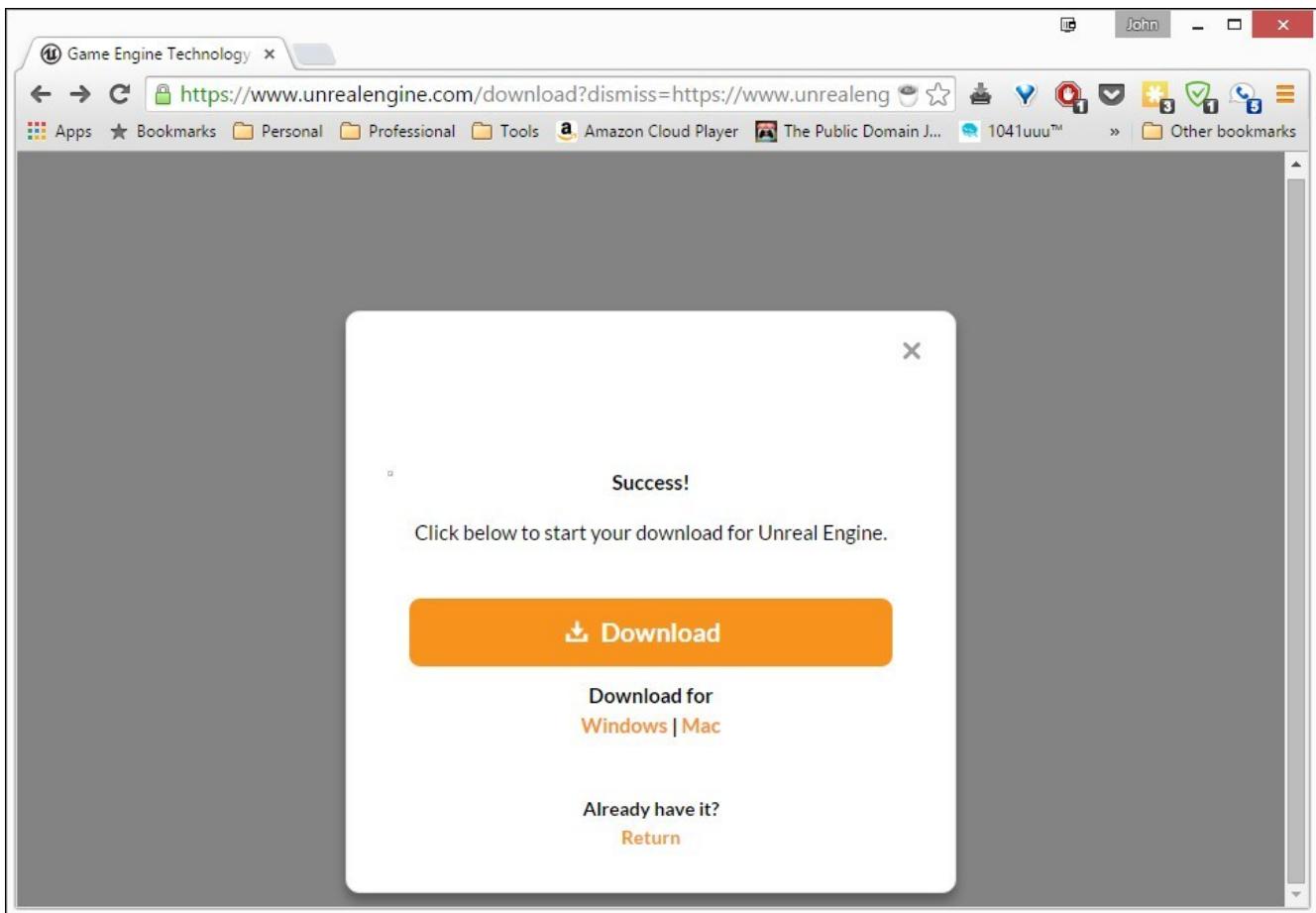
It's important at this point to note the subscription plan that Unreal has for its user. You need to pay a 5 percent royalty to Unreal for any games that you publish. Of course, if the title is released for free, you don't need to pay anything.

#### Note

If you happen to be a student who is 13 or above and are enrolled in a degree or diploma granting course of study, GitHub has a pack of resources for student developers that currently includes free subscription to Unreal Engine 4 for a year. If you have a school-issued e-mail address, valid student identification card, or other official proof of enrollment, check it out at <https://education.github.com/pack>.

If you happen to be a teacher or school administrator, it may also be possible to get access to UE4 for your school. Find out more at <https://www.unrealengine.com/education>.

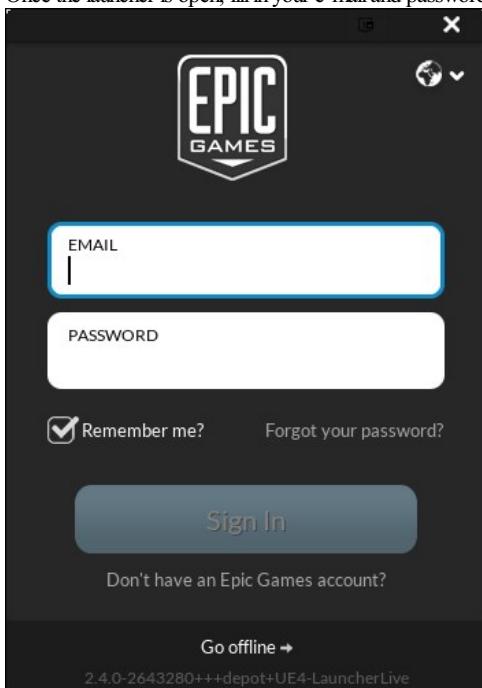
- Once you are logged in to your account, make sure that you are on the **Subscription** tab and then click on the **Download** button on the right-hand side of the screen for the operating system of your choice (I am using Windows).



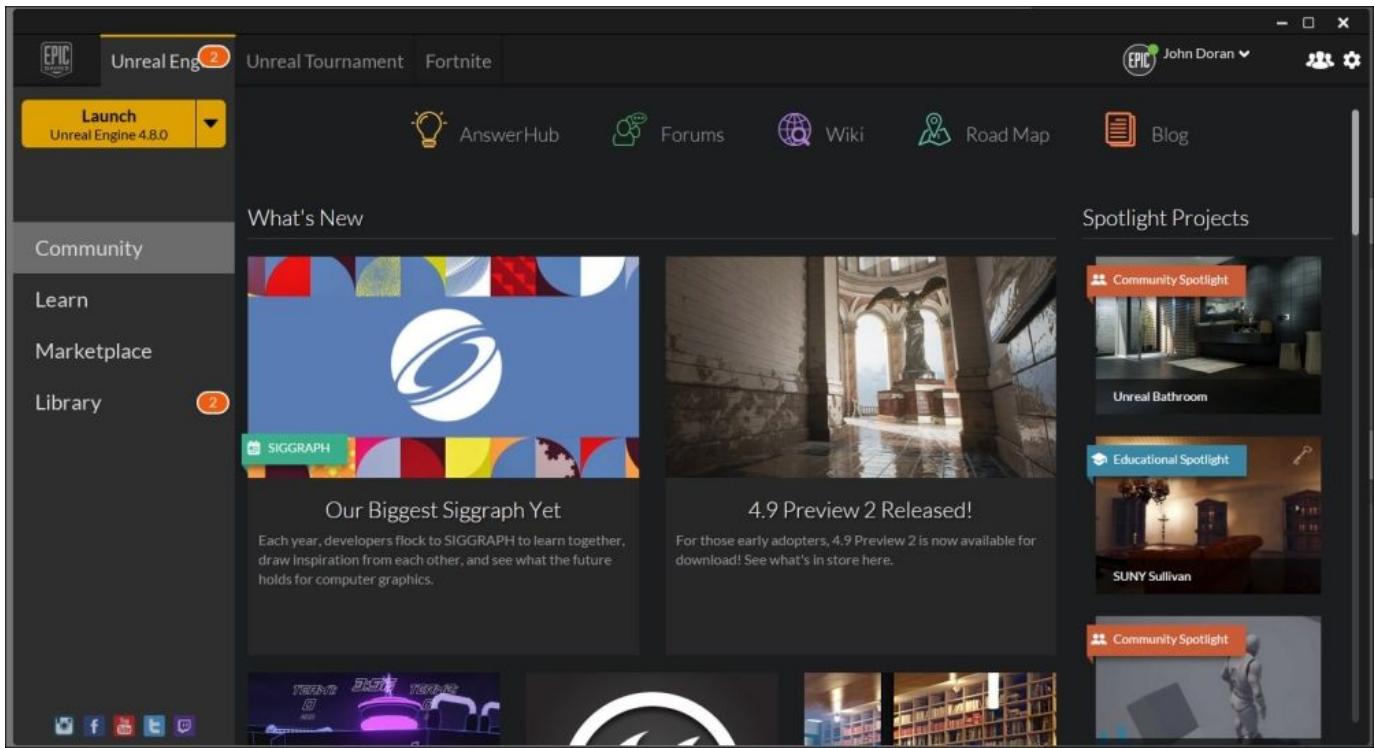
## Note

If by chance you don't see the preceding screen, you can always get the latest version of the software at <https://www.unrealengine.com/dashboard>.

5. Next, once the installer has finished downloading, open it up and start the installation. If you see a security warning, click on the **Run** button.
6. Go through the installation process, but make sure that the destination folder you're installing it in has a lot of disk space as UE4 will take up around 8 GB of space for each version that you have installed. Once the installation is complete, the Epic Games Launcher should open. If it doesn't, open it from your desktop.
7. Once the launcher is open, fill in your e-mail and password and then click on the **Sign In** button.



8. When you log in, the current version of UE4 will begin to download. Take a break as this will most likely take a while. Once the download is finished, you can see the **Launch** button lit up.



## There's more...

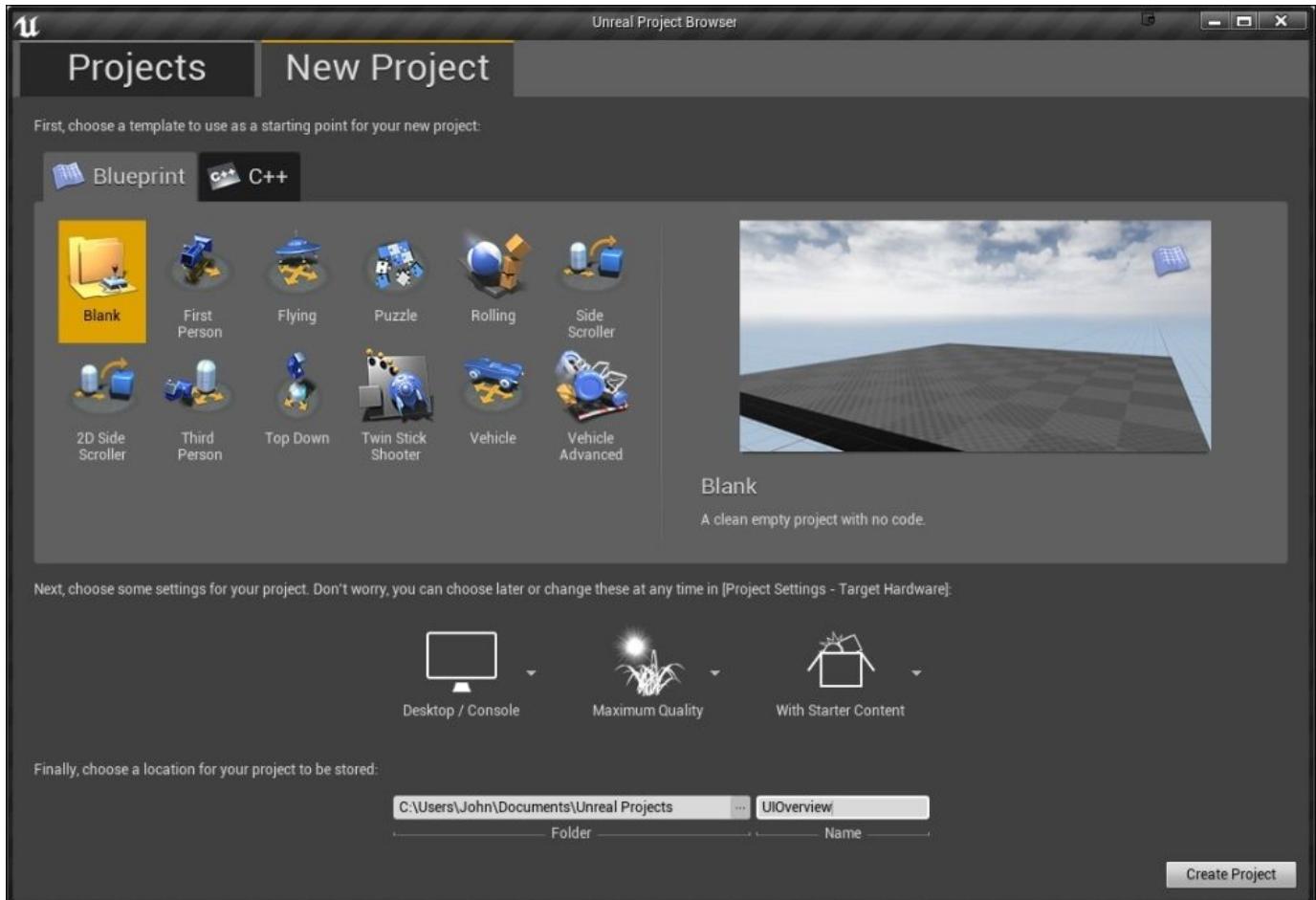
Now that you have your UE4 installation completed, it's a good idea to see what actually has been installed. For a definition of what all of these folders are used for, please refer to <https://docs.unrealengine.com/latest/INT/Engine/Basics/DirectoryStructure/index.html>.

# UI overview

One of the hardest things to understand when first starting out with a tool is knowing how to actually access all of the tools that are contained in the engine. Let's take a look at the interface of the Unreal Editor.

## Getting ready

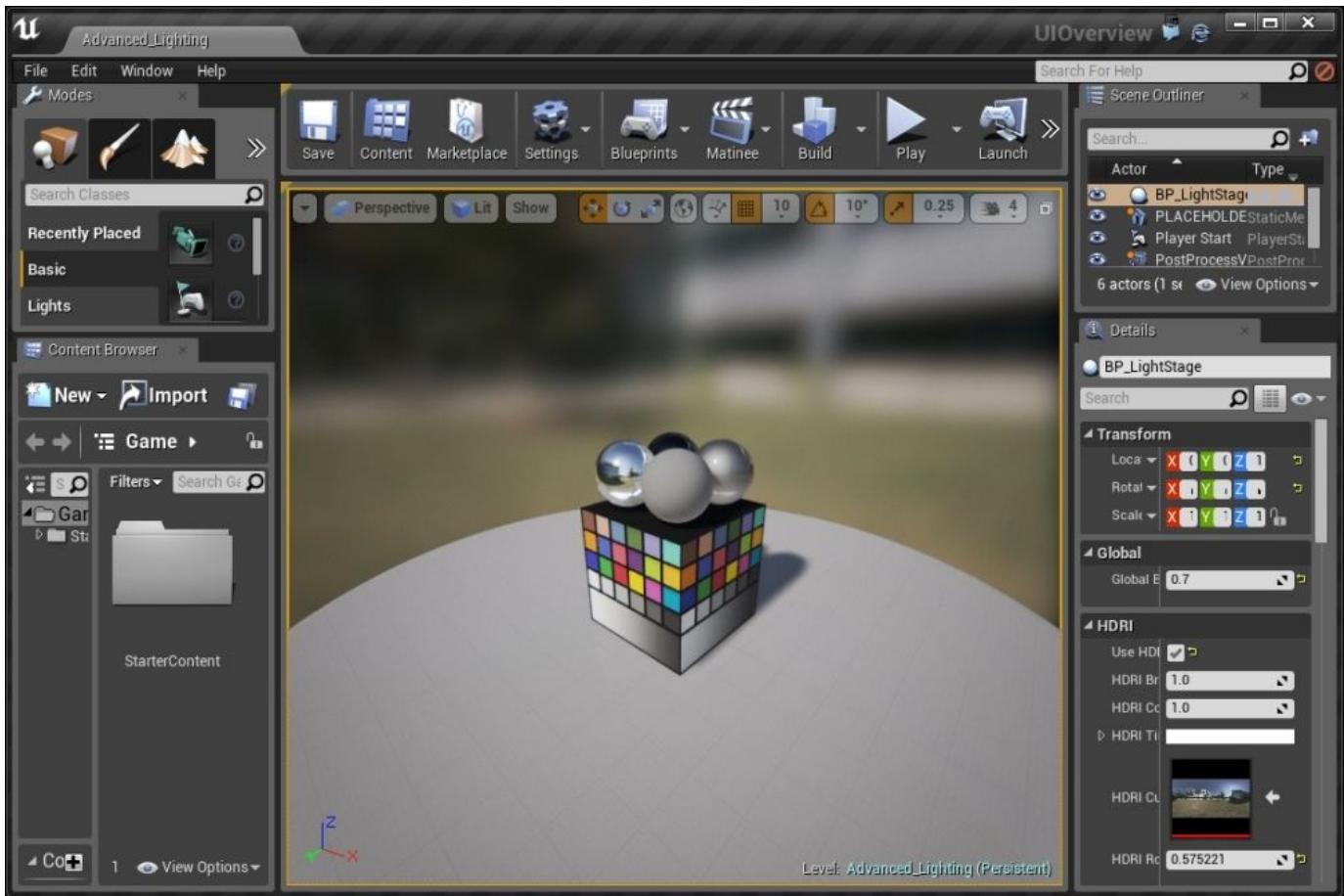
When you actually launch UE4 from the launcher, the first thing that you'll be brought to is **Unreal Project Browser**. Switch to the **New Project** tab, and you'll be given a choice of various templates to use for projects. For now, we'll stay with **Blueprint** visual scripting and will create a **Blank** project with starter content included so that we can see some stuff in the project. For the project's name, I have put **UIOverview**. Once finished, click on **Create Project**.



## How to do it...

Now that we have our project set up, let's get started and see what the editor looks like:

- Once the project is opened, close the tutorial popup that comes up (if it does). This new window that's opened is referred to as the level editor, which is the core of content creation in UE4. Here, you can see the default interface layout:



2. At the top-left of the editor, you can see the **Modes** tab, which contains various tool modes to allow you to put things into the world, such as BSP brushes, painting and foliage and terrain. Below this, you can see the **Content Browser** tab, which contains all of the models, textures, and data that make up our game worlds. We'll be exploring this much more in *The Content Browser overview* recipe later in this chapter.
3. In the center, the largest window that you see is the **viewport**, which is the actual level that we are building. We will talk more about viewports in the *Navigating the viewport* recipe.
4. To the right of the window, you will see the **Scene Outliner** tab, which will display all of the actors within our level. This is a useful tool for being able to find actors easily as well as adding a parent/child relationship to objects. Below the **Scene Outliner** tab, you'll see the **Details** tab, which contains information about whatever object is currently selected in the level or the Scene Outliner tab. For each component on the object, it will display the functionality for it, such as the transform and the materials the object uses.
5. At the top, you'll see the tab bar, which will show the name of your project as well as a tab for the level that you currently have running with its name.
6. Below this, you'll see the menu bar that will provide access to general tools and commands:
  - The **File** menu lets you save and open maps as well as projects. It also allows you to import/export actors.
  - The **Edit** menu allows you to copy and paste actors as well as configure properties in the editor. In this menu, users can configure **Editor** and **Project Settings** as well. It is in these settings that let you create the icons for the game launcher, set up input actions for your game type, and so on.
  - The **Window** menu allows you to toggle visibility of the various things that UE4 contains and save or reset your layouts.
  - The **Help** menu has a number of additional resources to help make working in UE4 as painless as possible.
7. On the right-hand side of the menu bar, you'll see a search bar that you can use to look for help. The far right of the bar shows whether you are currently connected to source control through Subversion (SVN) or Perforce, which would be useful when you're in teams.
8. Finally, below this in the center is the toolbar that contains a group of commonly used shortcuts to make it easier to find certain things.

### Note

For more information on the default interface, check out <https://docs.unrealengine.com/latest/INT/Engine/UI/LevelEditor/index.html>.

# Navigating the viewport

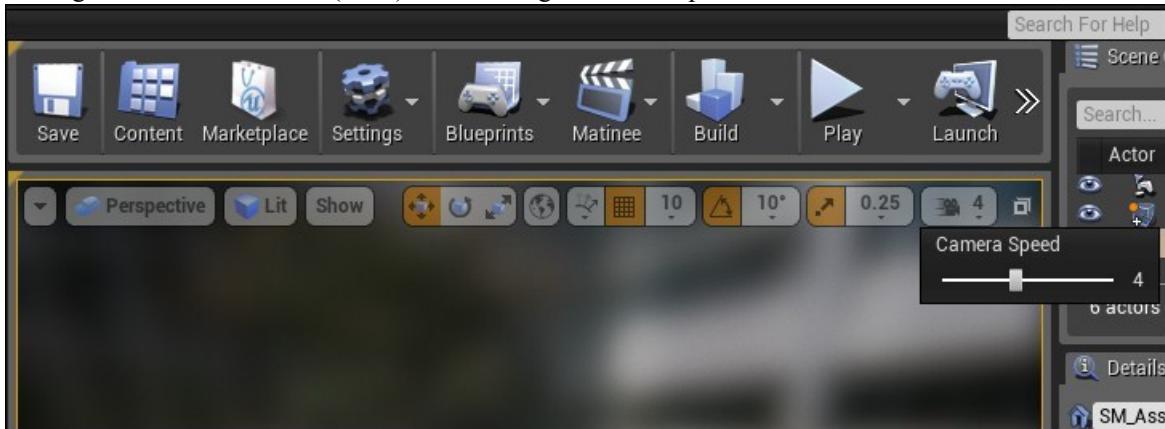
Now that we have an understanding of what the UI actually is all about, let's work with the viewport and learn how to move around and use it.

## How to do it...

To get started, let's first try to move around in the game world a bit by using just the mouse:

1. Inside the viewport, with the left mouse button clicked and held, move your mouse forward and you should notice the level moving as well. If you move your mouse backward, you should notice that the camera is moving in the same way, and when we move the mouse to the left and right, the camera turns, it doesn't move.
2. Holding the right mouse button and moving the mouse will rotate the viewport camera in a similar manner to a **First-Person Shooter ( FPS )** game. Moving the mouse up and down will make the camera loop upward and downward. And, when we move the mouse to the left and right, the camera behaves in the same manner.
3. Holding the middle mouse button (scroll wheel) and moving the mouse will pan the camera in the direction you move it as if it is on a track.

You can adjust the speed with which the camera moves by modifying the **Camera Speed** property in the top-right of the viewport to increase or decrease the amount of movement you need to travel via the camera. Alternatively, holding the left or right mouse button and scrolling the middle mouse button (wheel) will also change the camera speed.



4. In addition to rotating the camera like an FPS game, when holding down the right mouse button you can also use the *W*, *A*, *S*, and *D* keys to allow you to move just like you would in an FPS in **Spectator** mode.

### Note

If you're not a fan of the right mouse button, you can hold any other button on the mouse and move. If you aren't a fan of the *W*, *A*, *S*, and *D* keys, you can also use the arrow keys or *8*, *4*, *6*, and *2* keys on the numpad.

5. You can also use the *E* and *Q* keys in order to rise or fall in the air and the *C* and *Z* keys to zoom in and out, respectively, by changing the **field of view ( FOV )**. This change in FOV is only temporary though as when you release the right mouse button, it will reset back to normal.
6. The final way we can move through the viewport is very similar to how Maya users can move around their models. We activate this mode by holding the *Alt* key. If we click and drag, we will tumble around whatever is there in our current pivot in a similar manner as we orbit around the pivot. Clicking on the right mouse button and dragging will zoom the camera in and out of the pivot, while holding down the middle mouse button and dragging will move the camera in the direction of the mouse movement. We can change where our pivot is easily by selecting the object we want to move around and then pressing the *F* key to focus on it.

### Note

For more information on moving around the viewport, refer to <https://docs.unrealengine.com/latest/INT/Engine/QuickStart/2/index.html>.

# The Content Browser overview

The **Content Browser** is a central repository for creating, importing, and modifying all of the content that we use within UE4. This contains all of the assets that our project is made of, and it's important to have a good idea about how to use it.

## Getting ready

This recipe assumes that you have a project open with the sample assets included. If you do not have that yet, feel free to follow the instructions in the *Getting ready* section of the *UI overview* recipe.

## How to do it...

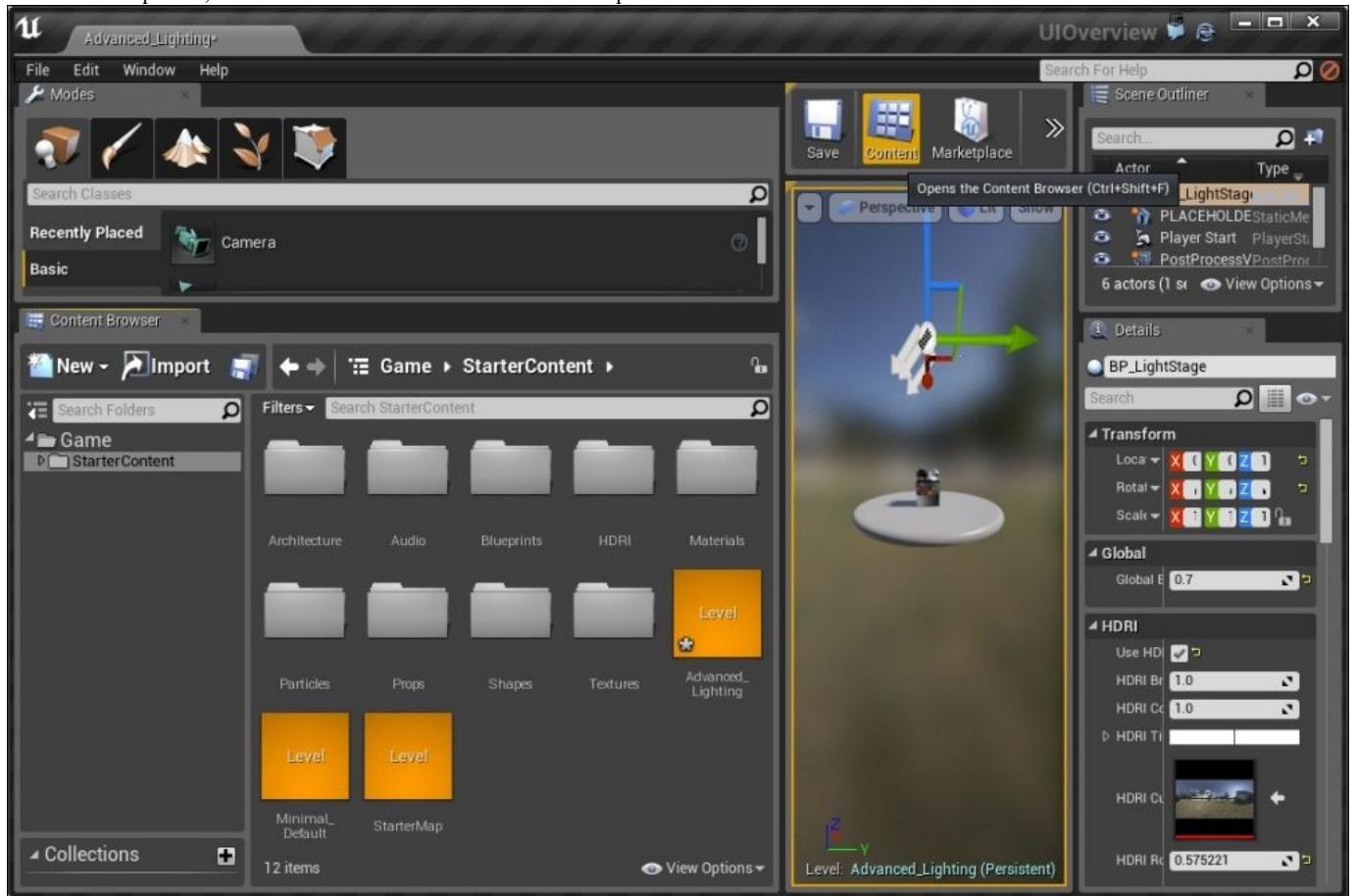
Now that we know how to move around the viewport, we will next want to get acquainted with the **Content Browser**, which is what we use in order to import or modify contents for our project:

1. By default, the **Content Browser** tab is docked in the lower-left corner of the main Level Editor interface, but it can be redocked anywhere within the Level Editor or floated as its own window. You can make it float as a separate window by clicking on the **Content Browser** tab and dragging it off. If you have a second monitor, having one for the **Content Browser** tab can often be a nice way to work as you'll often be grabbing things from there and bringing them into the world when building levels.
2. Close the **Content Browser** tab by clicking on the X button in the top-right corner of it. To bring it back, go to **Window | Content Browser | Content Browser 1**

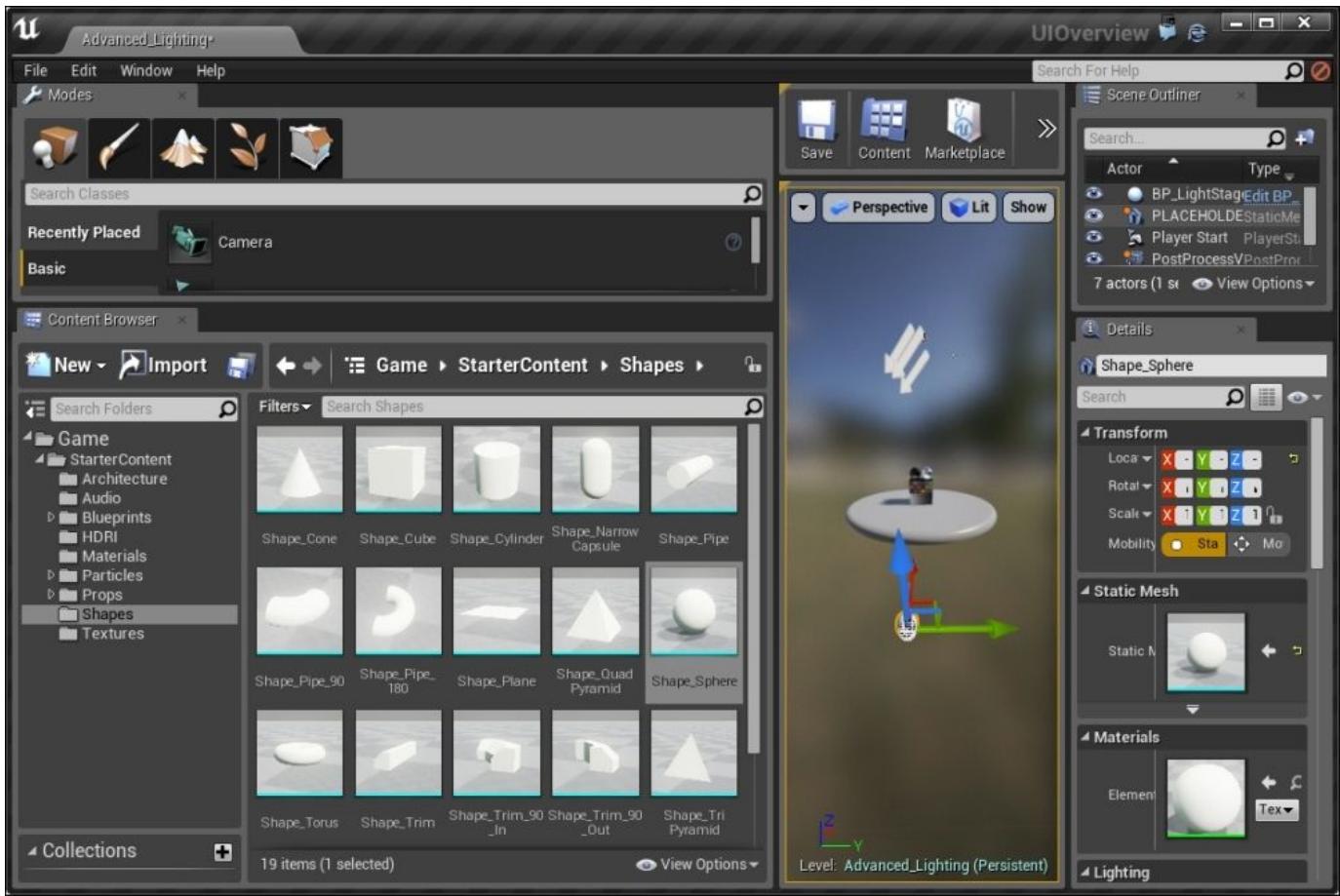
### Note

Alternatively, you can also click on the Content shortcut icon above the viewport or press *Ctrl + Shift + F* to bring the **Content Browser** tab back.

3. You can snap the **Content Browser** tab back to where it was by dragging the tag over it and releasing. You can also create multiple **Content Browser** tabs to allow you to see multiple things at a time or to move assets between folders easily.
4. The interface of the **Content Browser** tab needs space to look nice. Move your mouse over to the edge of the Content Browser tab and drag to extend it. Do the same from the top. Next, double-click on the `StarterContent` folder to open it.



5. The top section is called the navigation bar. It allows you to create, import, and save assets on the left-hand side as well as to move through the different folders in a similar way to a web browser.
6. Below this on the left-hand side is the sources view. This contains a list of all the folders and collections inside the project, formatted in **Folder Hierarchy**. Extend the `StarterContent` folder in the view to see all the folders.
7. Below this is the **Collections** view, which provides easy access to your created collections. **Collections** are a way for us to organize assets into personally-defined groups, such as all characters or environment meshes for a level. Unlike being in a folder, you can think of all of the objects in a collection as being a reference or shortcut to that content. This can be collapsed if you're not using it by clicking on the icon to the left of the **Collections** text.
8. On the right-hand side, below the navigation bar, is the asset management area. This is used mostly for filtering out files or searching for a particular asset that we will see below in the asset view.
9. Below that is the asset view, which is the largest section of the UI. This is a grid displaying all the items that meet the filter requirements in the navigation bar's folder. Right-clicking on an asset or folder will show contextual options based on the objects. All of the assets you see can be dragged and dropped into a scene easily by clicking on the `Shapes` folder and dragging one of the objects into your scene.



10. You can also create new objects within the folder you have selected by right-clicking on some open space and then selecting the desired asset from the menu.
11. In the bottom-right corner of the **Content Browser** tab, you'll see **View Options**. Select it and notice that you can view these assets in three different styles. Go through each of them and note the differences. Each of them have their own advantages and disadvantages; it's good to know that they all exist. You can also change the size of the thumbnails and this may be helpful as the number of objects that you have increases.

In **View Options**, users can also see the game engine's contents by selecting **Show Engine Content**. This will allow you to see all of the content included in the engine, by default, which can be quite useful for creating content for the game projects of your own.

# Importing your own content

Now that we have a good foundation on the Content Browser tab, let's start off by bringing in some of our own content into the game.

## Getting ready

This recipe assumes that you have a project open with the sample assets included. If you do not have that yet, feel free to follow the instructions in the *Getting ready* section of the *UI overview* recipe.

In addition, this recipe uses assets from the example code provided for the book. If you do not have it, download it from the Packt Publishing site at <http://www.packtpub.com>

### Tip

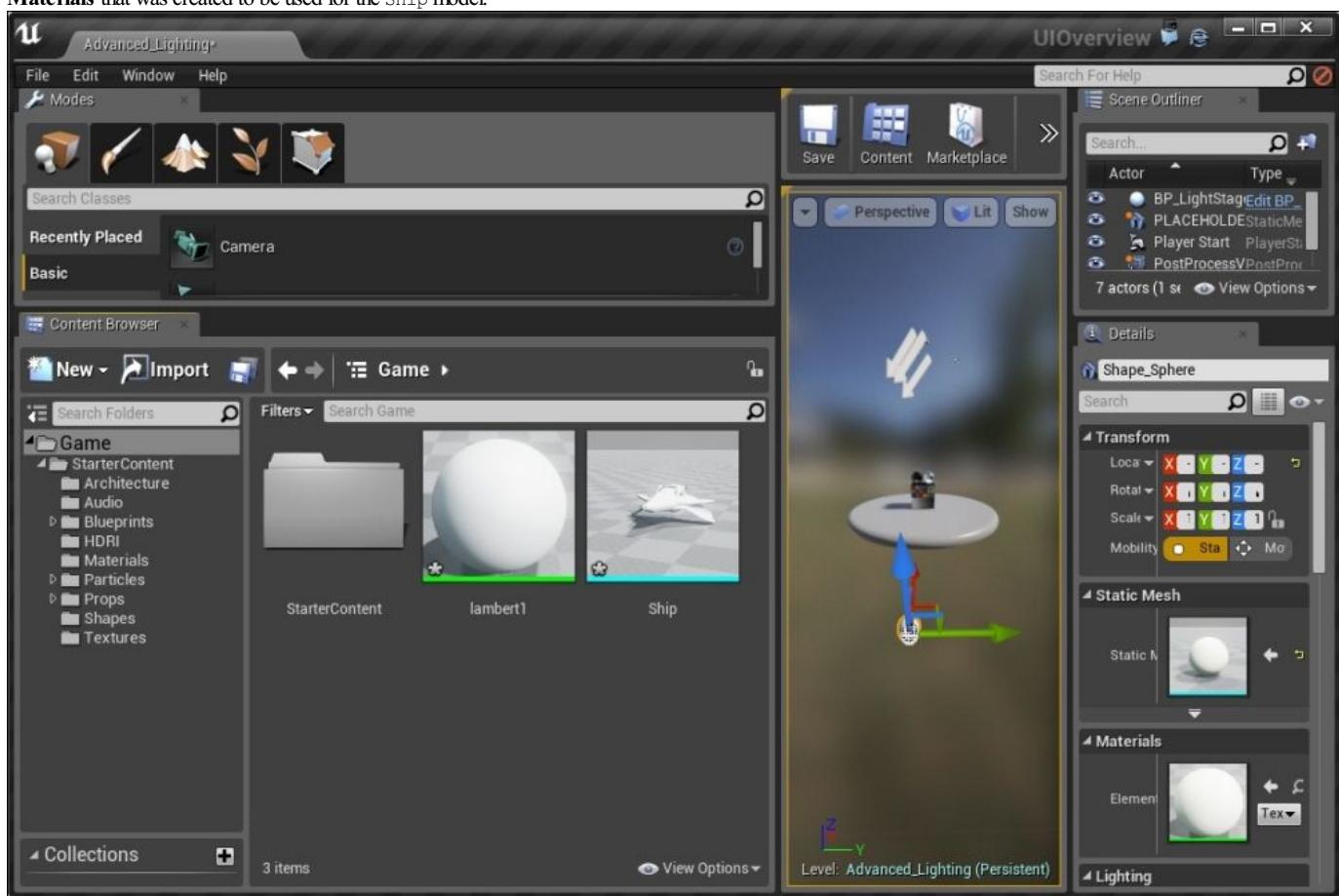
#### Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

## How to do it...

Let's start off by importing a simple model from the **Content Browser** tab.

1. Make sure that the **Content Browser** tab is at the **Game** folder and then click on the **Import** button
2. Once there, browse to the location where the example code of this chapter is placed and open the **Ship** folder. Select the **Ship.fbx** file and then click **Open**.
3. You'll be prompted with an FBX import dialog, click on **Import**, and you should see your new asset included in the **Content Browser** tab, accompanied by **Materials** that was created to be used for the **Ship** model.



4. The other way to import assets is simply by dragging and dropping them into the **Content Browser** tab. Let's do this by opening up our **Ship** folder in our operating system and then dragging it onto the asset view of the **Content Browser** tab. You'll notice that textures (images) do not have a dialogue like the model did.
5. Currently, all of the objects have a \* in the bottom-left corner of their images. This is because they are currently not saved to the project. Fix this by clicking on the **Save** icon (blue floppy disk) in the **Content Browser** tab.

Alternatively, you may also right click within the **Content Browser** tab and then navigate to **Import to /Game** to import assets into your scene from whatever folder you are currently in.

# Chapter 2. Level Design – Building Out Levels or Greyboxing

In this chapter, we'll cover the following recipes:

- Building a room
- Building out a level
- Applying materials to geometry brushes
- Converting brushes to static meshes or volumes

## Introduction

In the game industry, there are two main roles in level creation: **environment artist** and **level designer**.

An **environment artist** is a person who builds the assets that go into the environment. They use tools such as **3ds Max** or **Maya** to create the 3D model and then use other tools such as **Photoshop** or **Gimp** to create textures, including diffuse and normal maps.

A **level designer** is responsible for taking the assets that an environment artist creates and assembling them in an environment for players to enjoy. He/she designs the gameplay elements, creates the scripted events, and tests the gameplay. Typically, a level designer creates environments through a combination of scripting techniques and uses some tools that may still be in development. In our case, that tool is **Unreal Engine 4**.

### Note

One thing that is important to note is that most companies have their own definitions for different roles. In some companies, a level designer may need to create assets, and an environment artist may need to create a level layout. There are also some places that hire someone to just do lighting or place meshes (called a **meshing**) because he/she is so good at it.

In this chapter, we will be exploring the topics that are relevant to the role of a level designer.

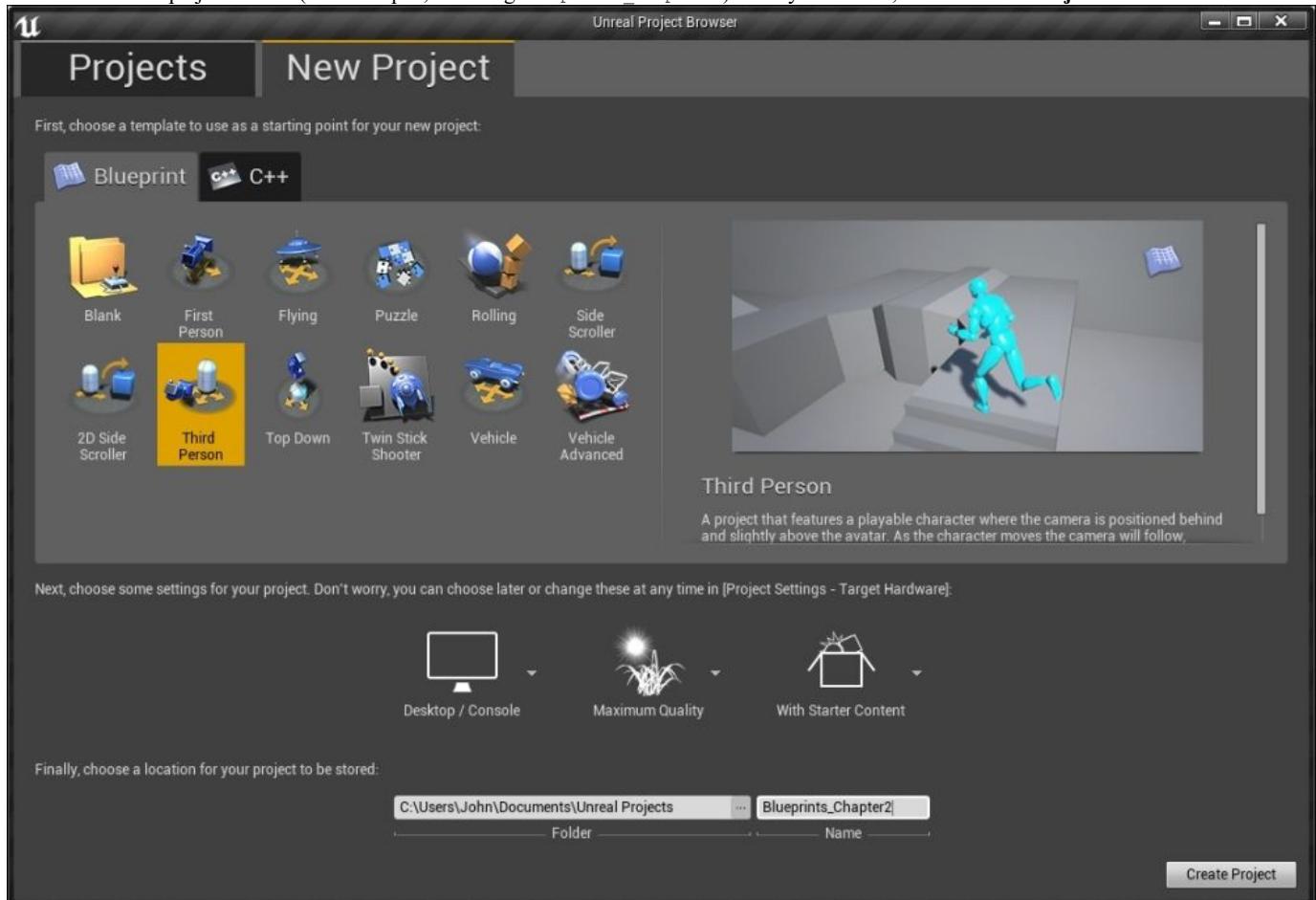
# Building a room

Now that we have an understanding of the interface of UE4, let's start with creating one of the most basic things, a room.

## Getting ready

Before we start working within the Unreal Editor, we will need to have a project to work with:

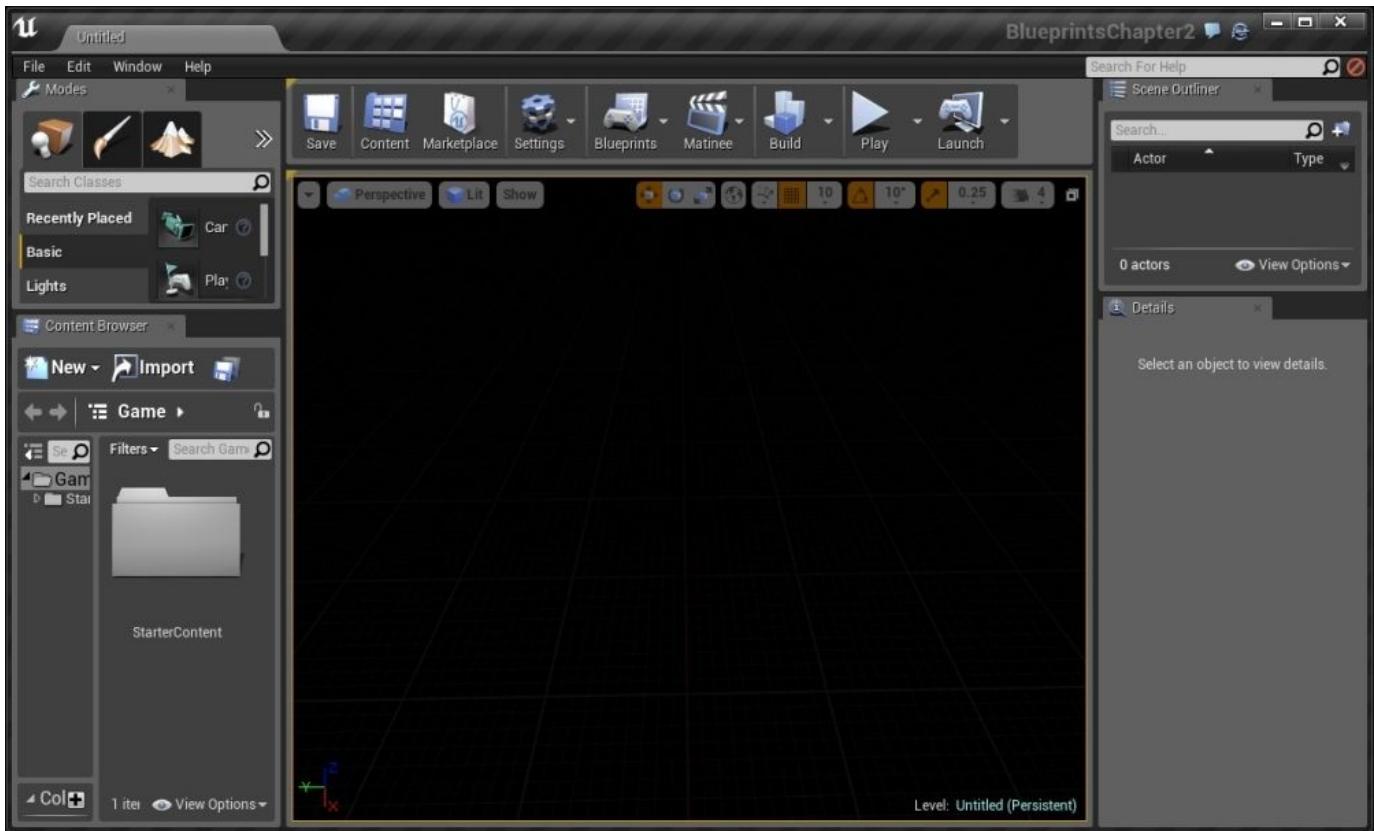
1. Open the Unreal Editor by clicking on the **Launch** button from the Unreal Engine Launcher.
2. Start a new project from the **Unreal Project Browser** window by selecting the **New Project** tab. Select **Third Person** and make sure that **With Starter Content** is selected. Give the project a Name (for this chapter, I am using `Blueprints_Chapter2`). Once you are done, click on **Create Project**.



## How to do it...

Now that we have our project set up, let's get started with building the room.

1. With the editor open, we will first create a new level for us to work in. To do that, go to the top-left corner of the screen and go to **File | New Level...** or press *Ctrl + N*.
2. From here, the **New Level** window pops up. Click on the **Empty Level** template, and you will find an empty level to work with.



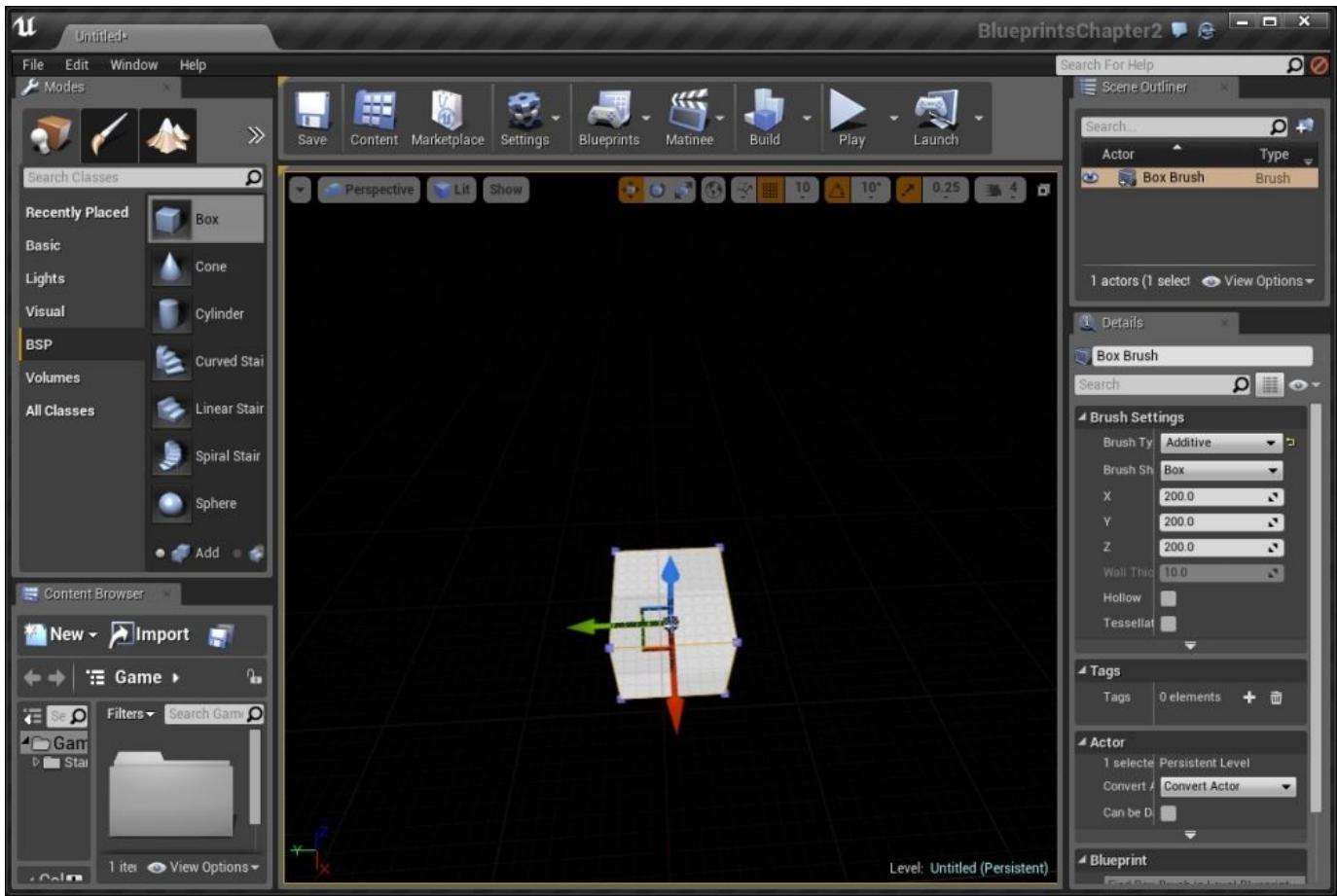
If you look closely and tilt downward, you should see a never-ending grid that fades into the distance. This is a guide to help show you distances and make it easier to work within a 3D environment.

3. Next, we will want to place some geometry into the world in which we can build out our room. To start out with, we will create a box. To do so, under the **Modes** tab, in the top-left corner, click on the place icon (the one on the far left that resembles a cube and lightbulb) to switch to the **Place** mode if not there already.

#### Note

For more info on the **Place** mode, refer to <https://docs.unrealengine.com/latest/INT/Engine/UI/LevelEditor/Modes/PlaceMode/index.html>.

4. Now select the **BSP** category on the left-hand side, below the mode icons. From there, click and drag the **Box** button into the world. You may see it show up as some red lines initially, but give it a minute, and you'll see the box appear on the screen, as shown in the following screenshot:



## Note

If you don't happen to see the box, that's because, by default, UE4 has the default lighting mode set to **Lit**, and as there is no light, everything will be just black once the lighting is built. You can switch to **Unlit** mode by going to the top-left area of the **Level** window and clicking on the **Lit** button and then selecting **Unlit** from the dropdown, or just using the hotkeys—*Alt + 3* for **Unlit** and *Alt + 4* for **Lit**.

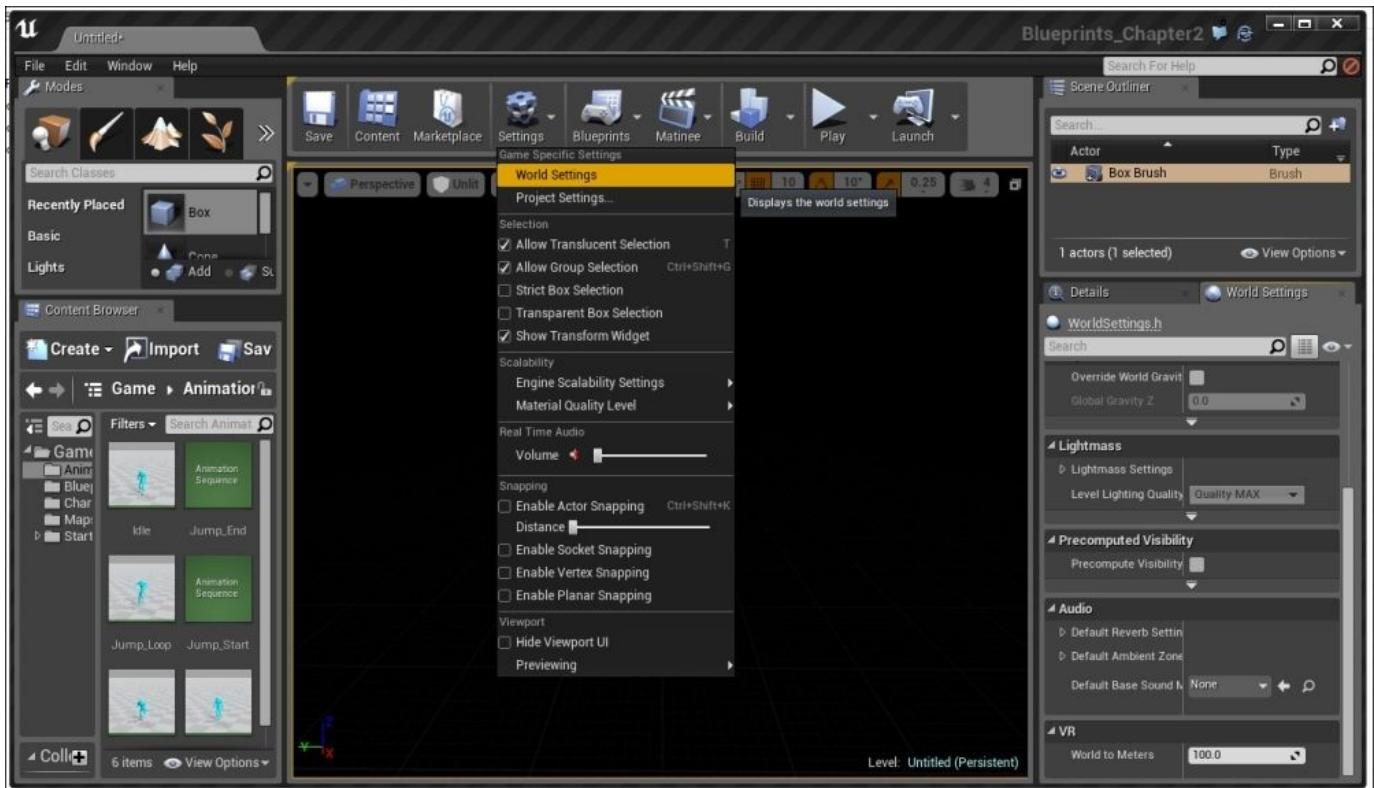
Once the box is created and selected, note the **Details** tab on the right-hand side. This tab contains all the information about this brush that can be modified. If you know what size your brush will be, this can be very useful.

- From **Brush Settings** under the **Details** tab, change the **X** and **Y** values of the brush to **1000** and the **Z** value to **25**.

This will create a room that is about  $10 \times 10$  m with a width of 25 cm. Be sure to note that it is **Z** that is the vertical axis inside the Unreal Engine. The **X** and **Y** values define how large we want our room to be, but I used a **Z** value of 25 because inside normal buildings, the joists or horizontal foundation supports between levels are about 10 inches (approximately, 25 cm) thick.

## Tip

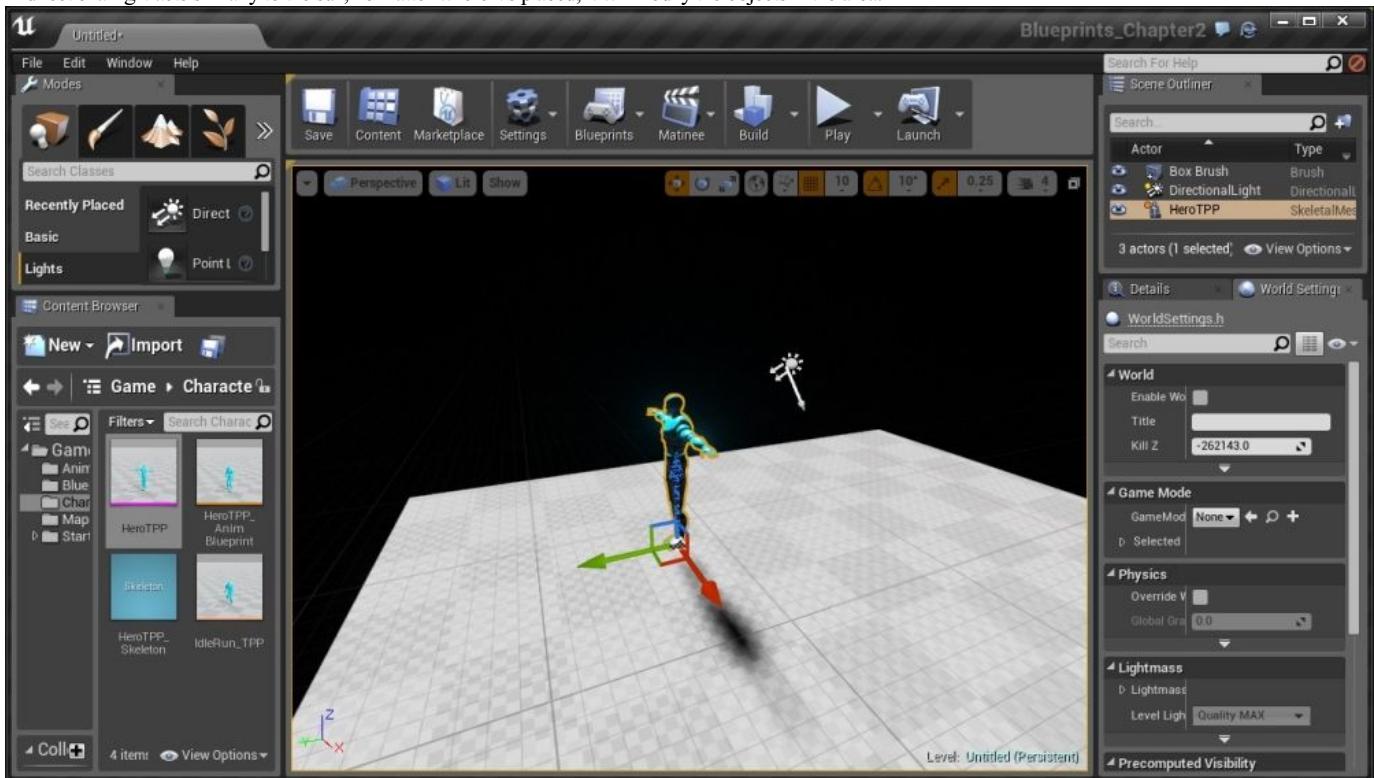
By default, 1 unit in Unreal Engine is equal to 1 cm. If you'd like to change this or customize this value, you can go to **Settings | World Settings** and modify the **VR | World to Meters** value to **100** (1 m is equal to 100 cm).



- Now that we have a floor created, let's add a reference for us to work with. Under the **Content Browser** tab on the bottom-left, go into the **Character** folder and drag and drop the **HeroTPP** skeletal mesh onto the scene.

- Having a character reference in the world makes it a lot nicer for us to see how the sizes of objects relate to our player.
- If we were to run the game now, we won't be able to see anything due to lack of light. Let's fix this. Under the **Modes** tab, change the section to **Lights** and drag and drop **Directional Light** into the level.

A directional light acts similarly to the sun; no matter where it's placed, it will modify the objects in the area.



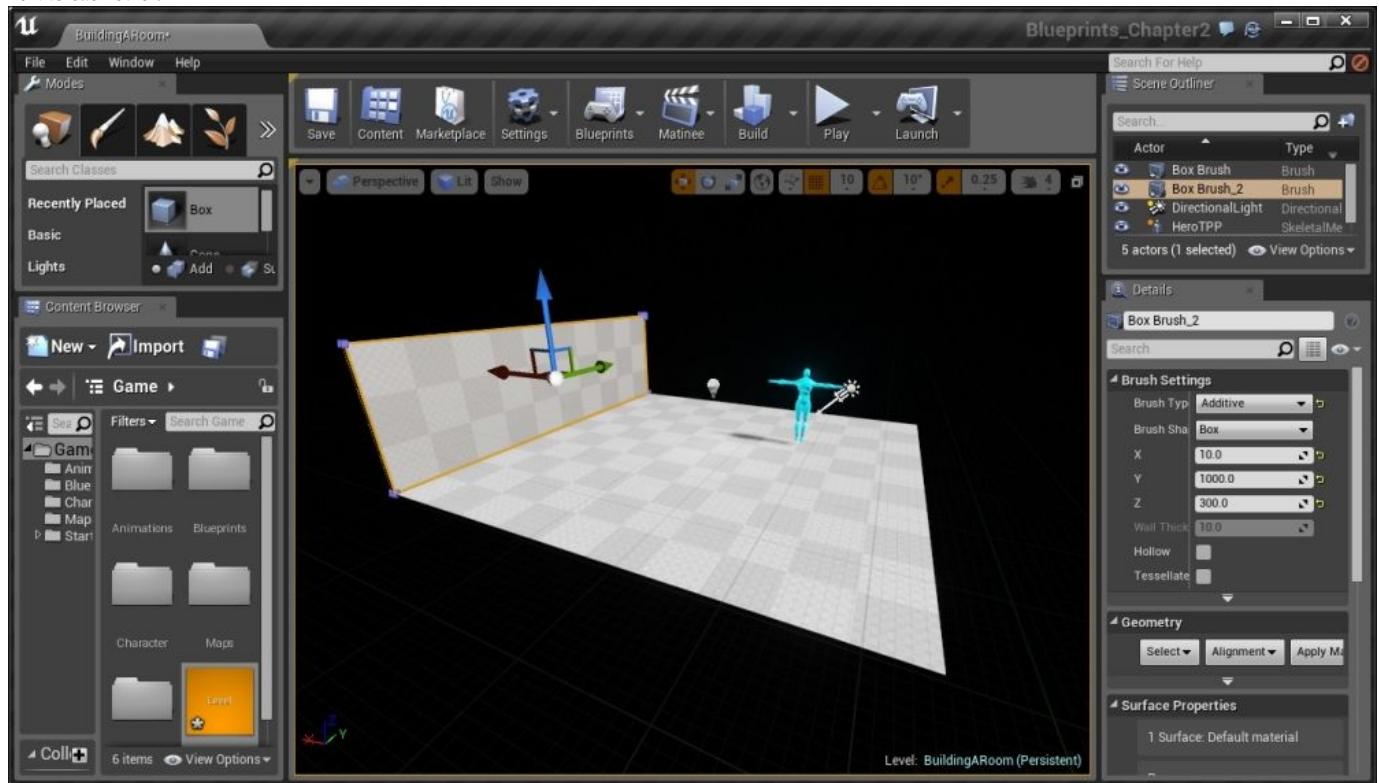
*Effect of a directional light*

- The next thing we are going to do is to create walls for this room. To do this, let's first drag another box into our world, this time placing it above our floor. Once created, inside the **Details** tab, under the **Brush Settings** section, set **X** to 10, **Y** to 1000, and **Z** to 300.

#### Note

Creating levels based on real world locations can be an excellent exercise for a budding level designer. Should you want to do that, the real world walls normally have a thickness of 12 for interior walls and 16 for exterior walls.

- Once finished, select the wall and use the transform gizmo (the three arrows) to move the object over the ground of our room (if you don't see the arrows, press the *W* key, and if you still do not see it, toggle Game Mode by pressing the *G* key) Once over it, press the *End* key, and you should see the brush fall down to hit the floor. Once this is finished, move the wall in the **X** and **Y** axis using the transform tool until it is flush with the edge of the level, which is to say they should be placed next to each other.

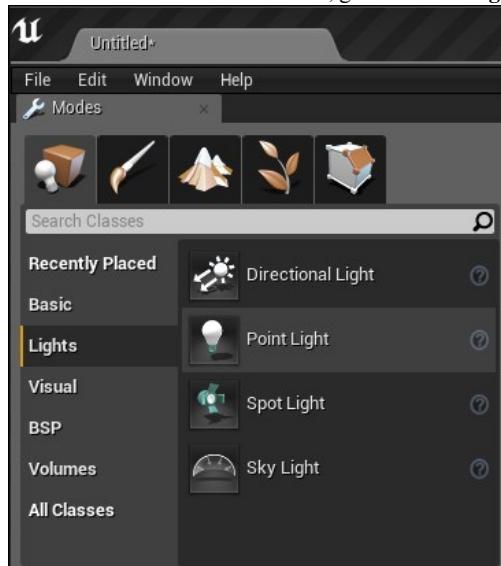


*Creating walls for a room*

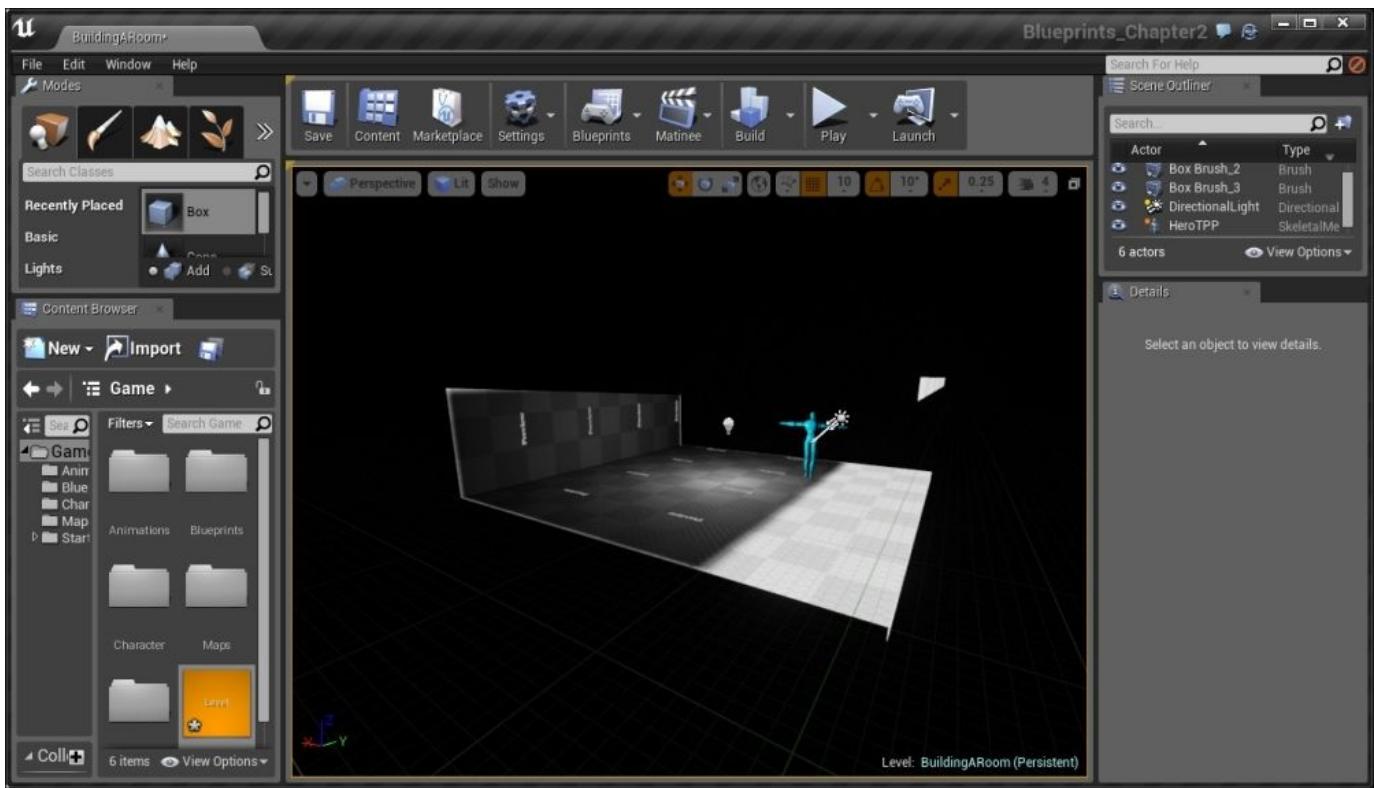
- We can add a ceiling to our room very easily. Click on the floor that we've created, hold down the *Alt* key and then drag the item by the blue arrow to be flush with the top of our wall that we created previously.

You'll notice that with the ceiling placed, there are now shadows covering a good portion of our level. If we want to be able to see in the room, we will need to add a new light, a **Point Light**.

- Back in the **Modes** tab in **Place** mode, go down to the **Lights** section and select it. Once selected, you should see the **Point Light** option on the right-hand side.



- Drag and drop **Point Light** into the center of the room. You'll see that we can see in the room now!



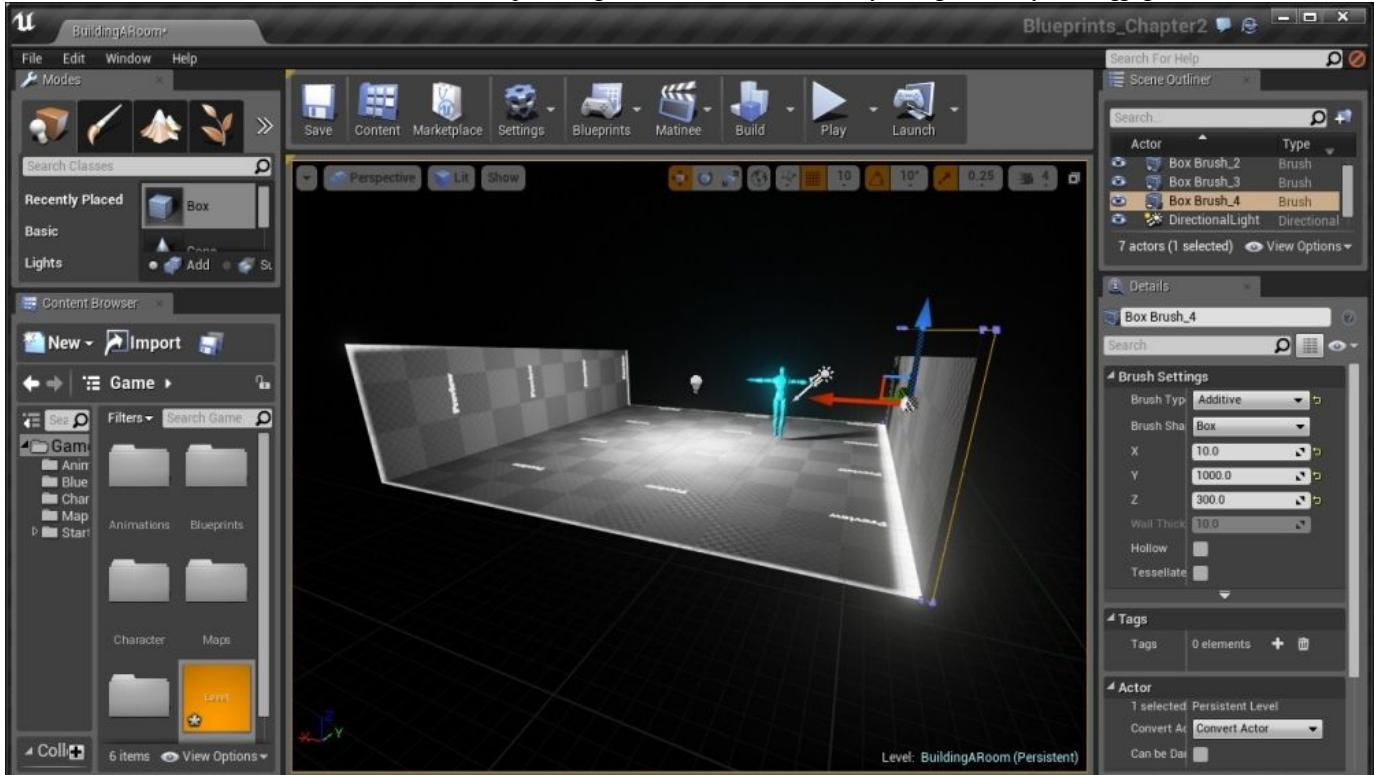
A **Point Light** works similar to how a lightbulb does in the real world, shining equally in all directions from a single point.

### Tip

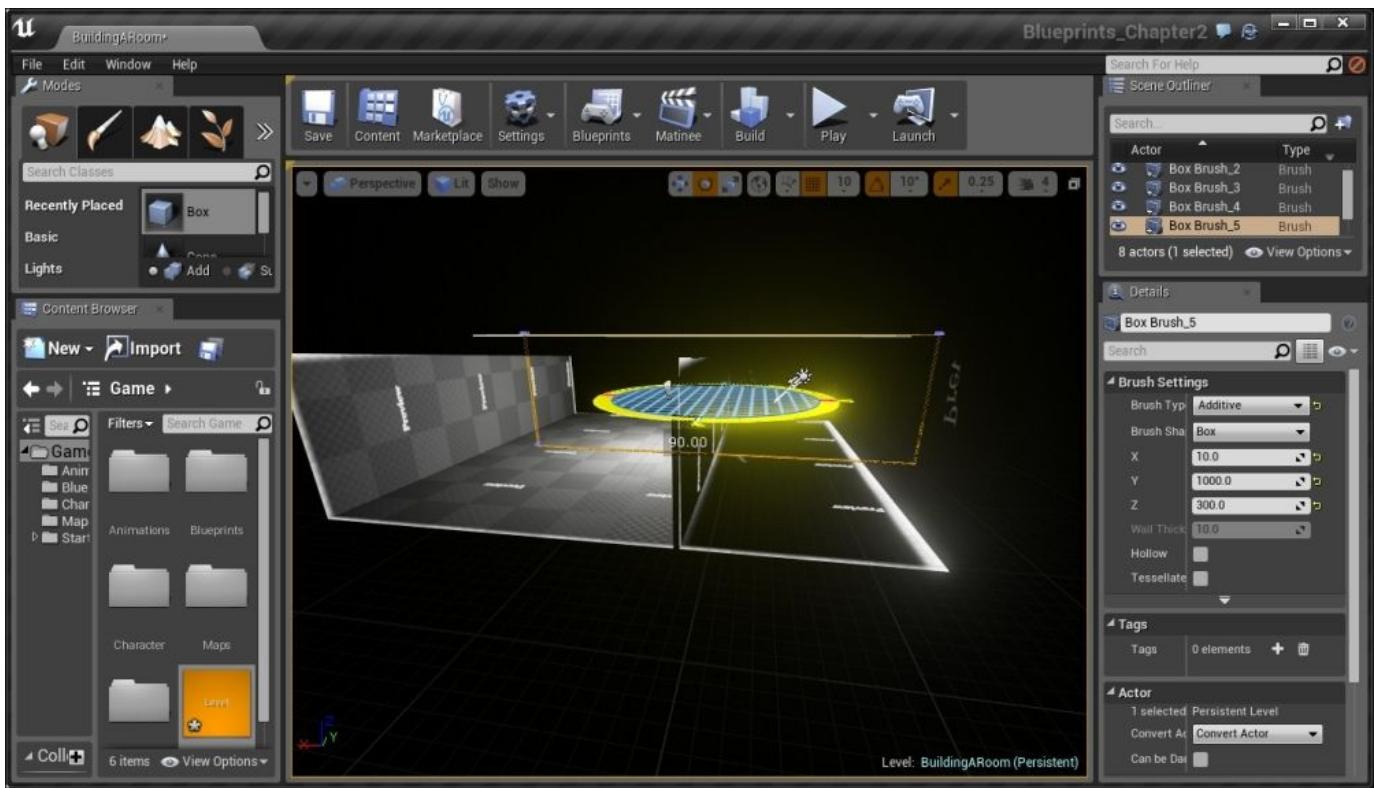
Alternatively, you can create a point light by holding the *L* key and clicking on the center of the room.

For more information on point lights, refer to <https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightTypes/Point/index.html>.

13. Next, let's close this room. With the wall selected, create a duplicate to go to the other side of the room by holding the *Alt* key and dragging it to the other side.

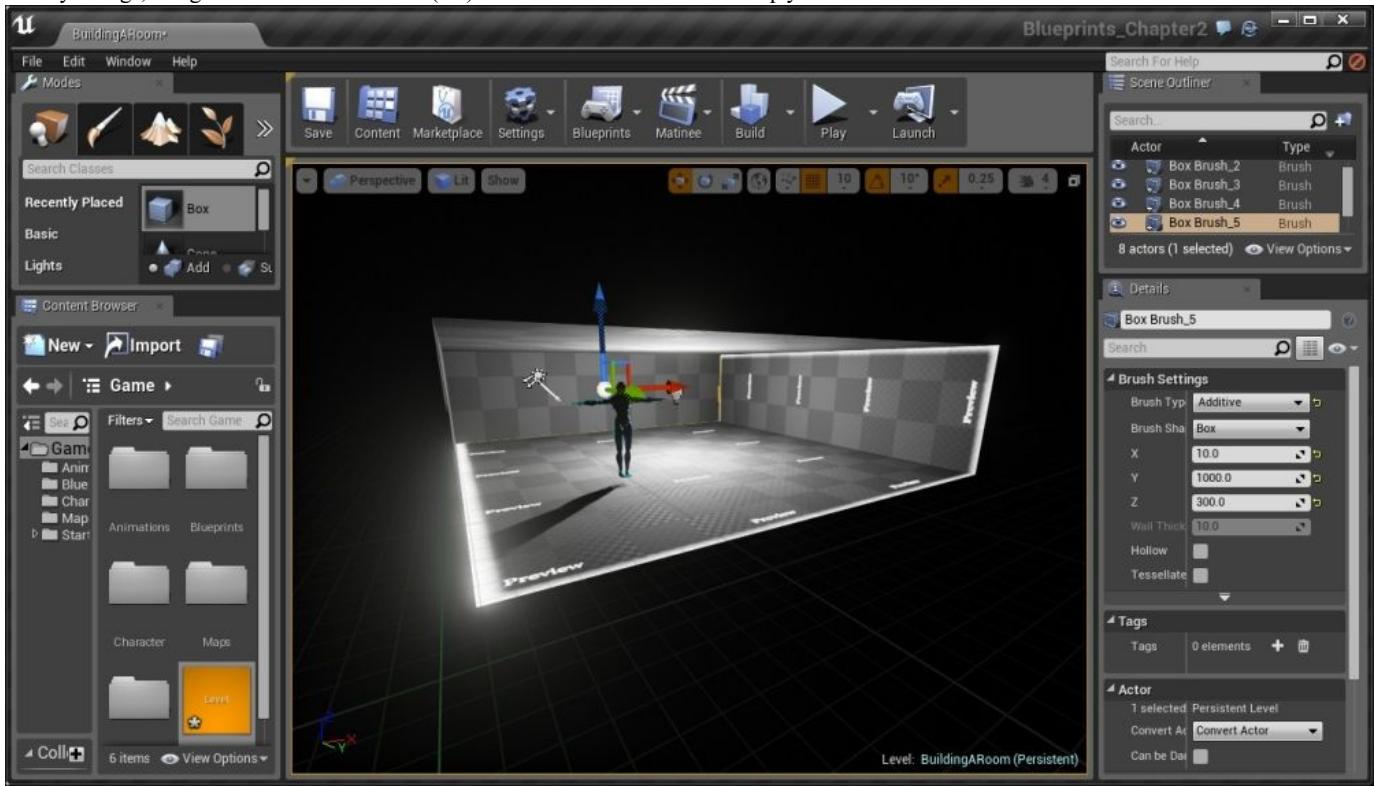


14. Next, we will place the other two walls. We could create another box like the one we created before and give it different values, but in this instance, we will use the rotation tool instead. Select one of the walls and hold the *Alt* key and drag the item in the X axis (red). After that switch to the rotation tool (*E*) and then rotate the wall 90 degrees along the Z axis (blue).



*The original wall has been duplicated and rotated*

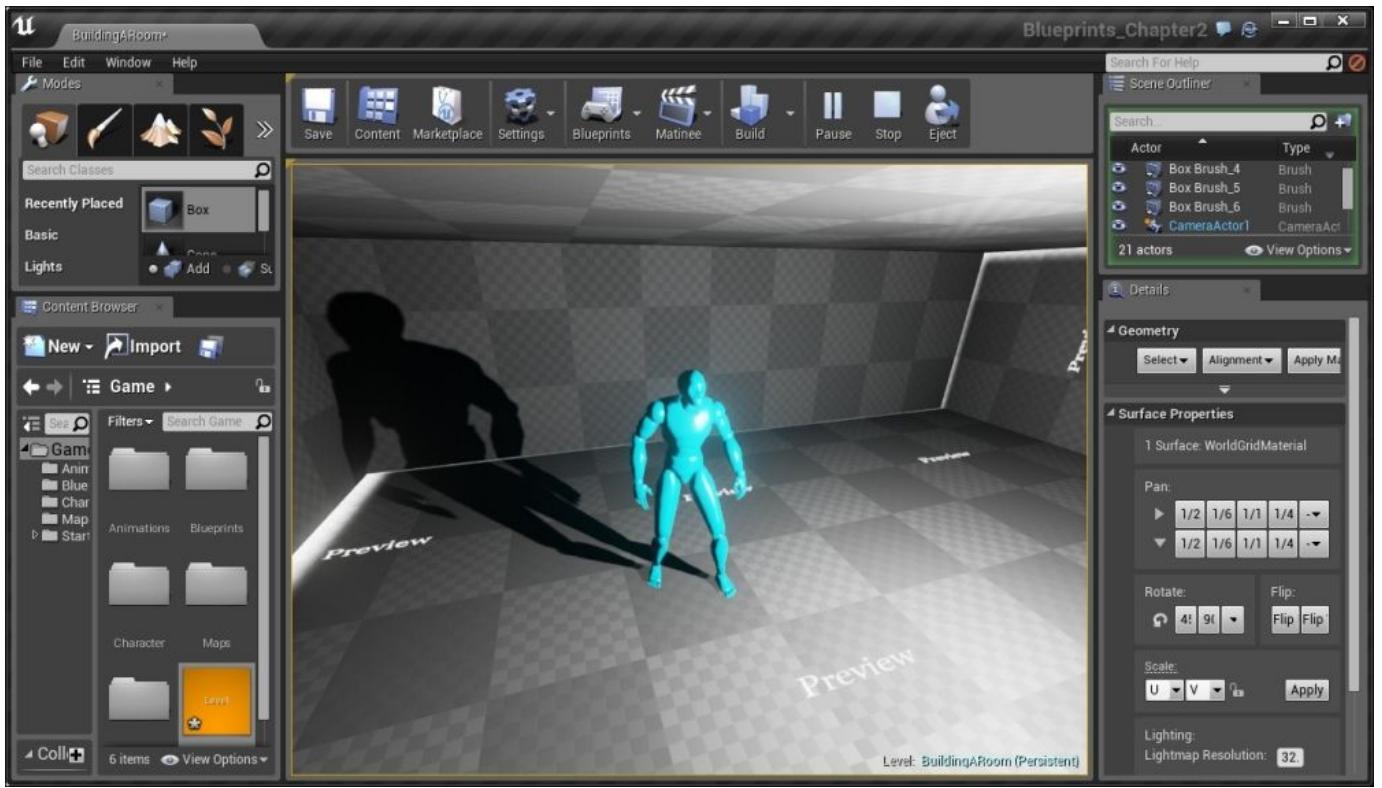
- When you let go, change back to the Translate tool (*W*) and then move the wall to fit the empty slot.



### Tip

You can also use spacebar to toggle through the transform, rotation, and scaling widgets. You should be able to tell which is which, based on the gizmo that shows up, and can just hit the spacebar until you find the one you want.

- Then, duplicate this newly created wall in order to close the gap. Now, to see what's going on in the level, you'll need to move into the map. Once inside, right-click on the floor and select **Play From Here**.



With that done, you now have a room that we can walk into!

### Tip

If, by any chance, you fall from the edges of the surface, press *Esc* to exit the game, switch your view from **Perspective** to **Top**, right-click on one of the objects (walls or surface), press *F* to focus on that object and then, finally, switch your view back to **Perspective**.

Alternatively, you can also double-click on any object in **Scene Outliner** to zoom the camera to it.

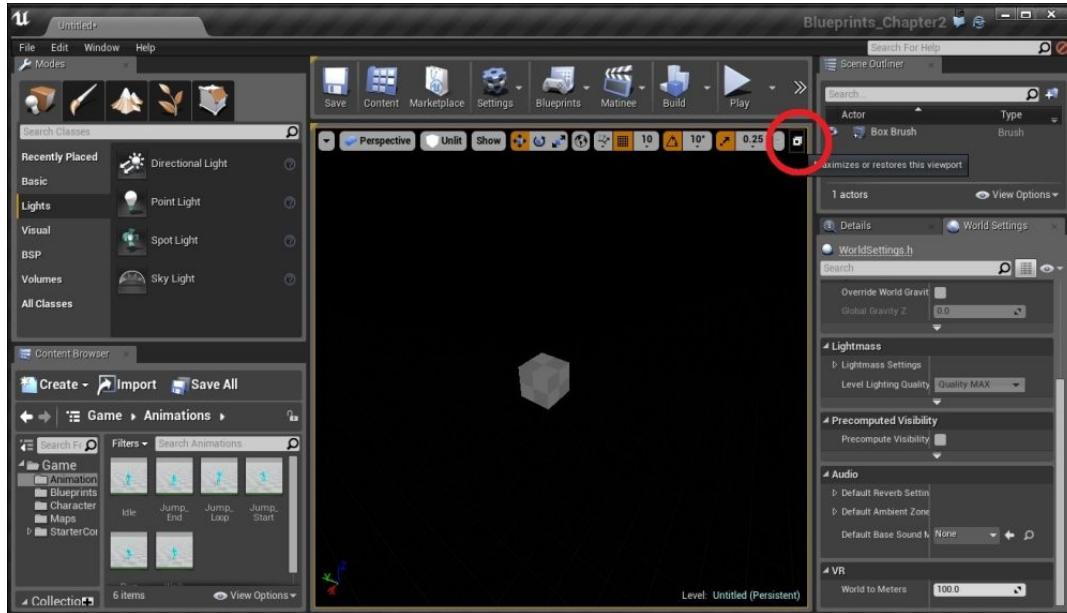
It is also possible to create a room by creating a box with a size of **1010**, **1010**, and **300** with **Hollow** enabled and a **Wall Thickness** of **10**, but you will lose a lot of control for larger areas.

# Building out a level

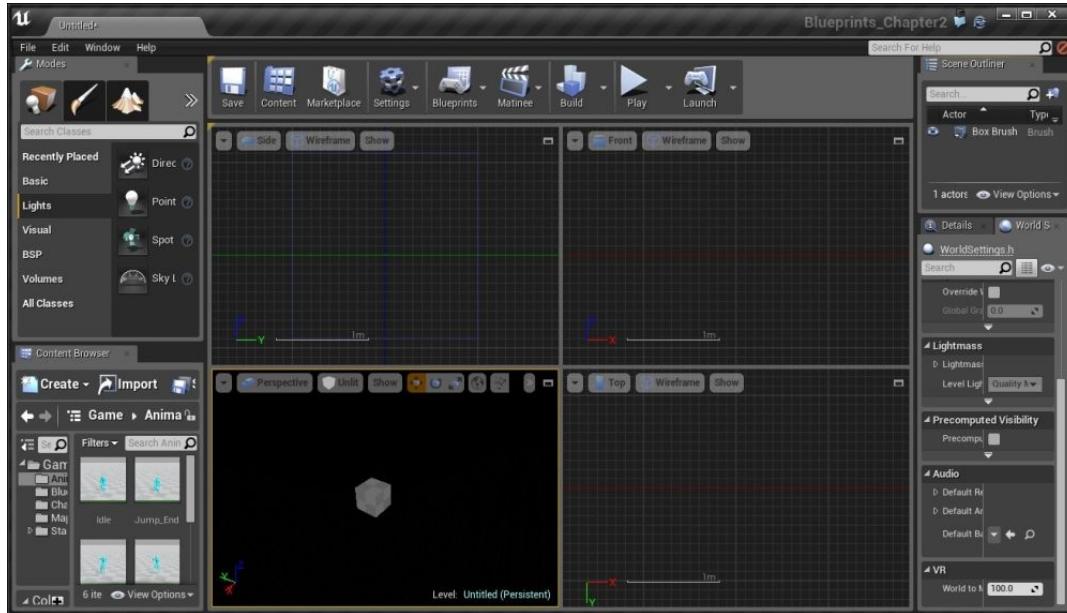
When I started using Unreal Engine, it took me a long time in order to create any type of new level. I would keep placing static meshes to hide areas of geometry that I missed or added in blocking volumes to make it actually possible to go through certain places. One of the key reasons I had problems with this, is because I didn't know how to create a good workflow and the key to mapping things very easily by using brushes in the **Geometry Editing mode**.

## Getting ready

Minimize the **Perspective** viewport by clicking on the button in the top-right corner of the viewport (circled in the following image):



You will then see four windows and notice a grid of sorts in all the other viewports.



## Tip

You can click on the same button in the top-right corner of any of the four windows to maximize it.

When in a different viewport, controls work a bit differently. Most importantly, right-clicking and dragging will pan the camera. For more information on the differences, refer to <https://docs.unrealengine.com/latest/INT/Engine/UI/LevelEditor/Viewports/ViewportControls/index.html>.

We will be using the grid as a guideline in the creation of our levels in the same way that we use paper to draw things out, which is something some level designers do to get the general feel of an area. Starting to build a general area needs to have planning ahead of time to have a general idea of how you want to place buildings and guide the traversal of the player.

## Some keyboard tips

Holding the left mouse button and dragging will allow you to select all the objects that are contained within it, which we refer to as a **marquee selection**. If you are in the Geometry mode, this will let you select individual/overlapping vertices, allowing you to increase or decrease the size of your brushes very easily which we will be using to create our environment.

Another useful tip is if you press **Ctrl** and hold and drag the left mouse button anywhere in a viewport, it will move the brush, actor, and/or vertices that you have selected from any position. This is a good way to move objects that may not be far away or not have to move the screen and don't want to use the widget that is usually by the object.

If you hold **Ctrl + Shift** while moving an object, your camera will move with you as well. This can be really useful for repeating things in a certain direction.

Also, you can select multiple objects by holding **Ctrl** and then clicking on multiple objects.

## Seeing double – duplicating elements

Duplicating things that we have already created, such as walls or buildings, are an effective way of blocking out an environment very quickly.

## Note

I did this in the setup of the previous recipe by duplicating the room that we created in the first section.

As what we care about most here is creating the best gameplay possible. We pay less attention to fine details here and basically, want to just block out an area so that we can iterate as quickly as possible. After all, you're a lot more willing to get rid of or change a huge box than a ridiculously detailed office building.

After placing a single brush in our level, you don't really need the builder brush again. Unless you are creating something other than a box, you can just duplicate brushes and mould them using the Geometry Editing mode to quickly shape out areas that usually makes it much quicker to build.

## How to do it...

With the knowledge of how to start a workflow, we can create a level:

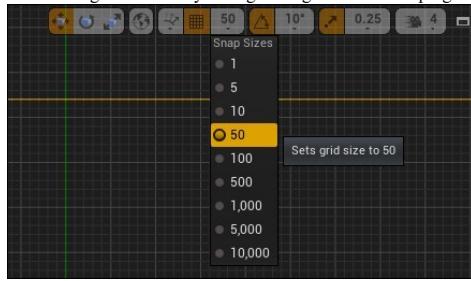
1. Create a new level by going to **File | New Level...** and from that window, select Default.
2. The floor that currently exists in the game is actually a static mesh. We don't need this, so let's get rid of it by selecting it and then pressing the **Delete** key.
3. Next, we are going to the **Restore Viewport** button on the top-right of the viewport to get the four viewport split screens.
4. Once there, let's add a box for our foundation. Go to the **Modes** tab and select the **Place** button and go to the **BSP** section. Once there, drag and drop the **Box** to the viewport to bring it into the world. Once created, go to the **Details** tab and change the **Brush Settings**—**X** to 5000, **Y** to 3000, and **Z** to 300.

## Note

I intentionally made this really tall because later, I will decrease the value of this brush to create a little area that my player can safely get around, and thus, I will not need to worry about them falling out of the world.

You may have noticed that when we move brushes around, they snap to certain positions. This effect is known as **grid snapping** and the amount of space moved is dependent on the **Grid Snapping** variable in the top-right of your viewport (by default, it's 10).

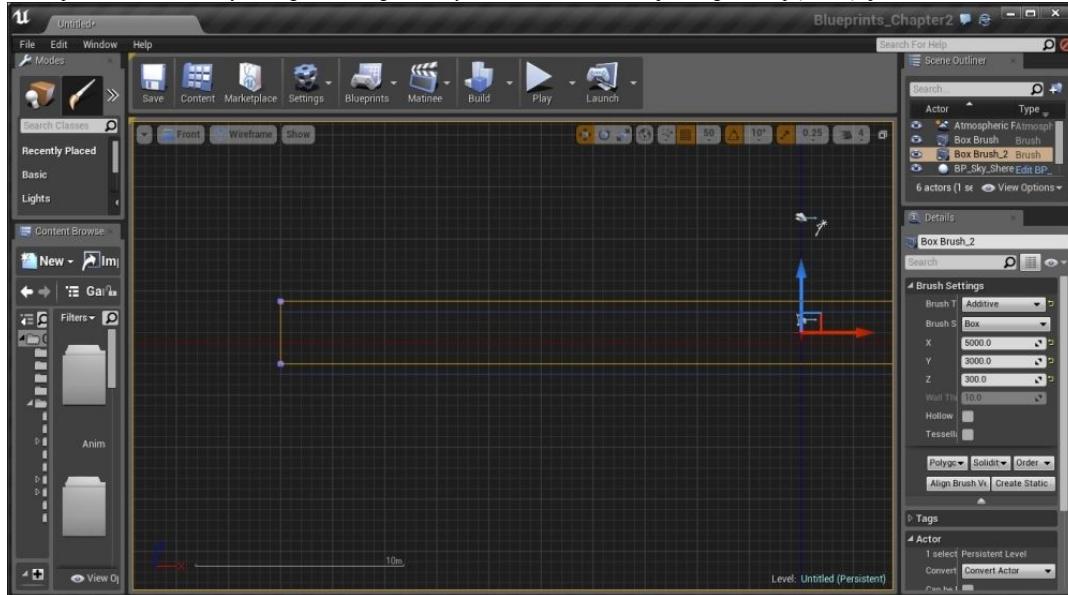
5. Increase the grid size to 50 by clicking on the grid icon on the top-right of any viewport and selecting 50 for the **Snap Sizes** value.



Another way to modify the grid space is by pressing the **/** and **\** keys in the editor that will decrease and increase the grid snap points, respectively, making the level more or less detailed in your brush placement. Some people will want to use a smaller area, but I argue that when blocking something out, we really only care about the big picture and getting the overall feel of the area.

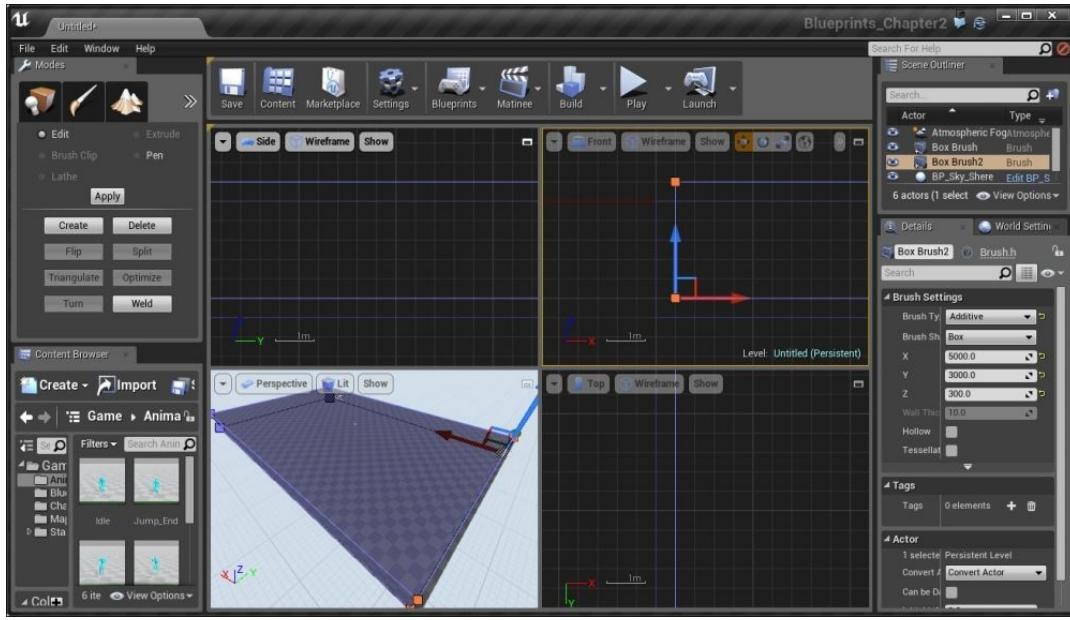
In case your brush is not aligned to the grid, you can right-click on the vertices, and it will automatically snap it onto the grid. Working with the grid is a fundamental way of making sure that you don't get any holes and/or overlaps of your brushes while creating a level.

6. Now duplicate our current brush by selecting it and holding the **Alt** key and then from the **Front** viewport, drag it one step (50 units) upward.



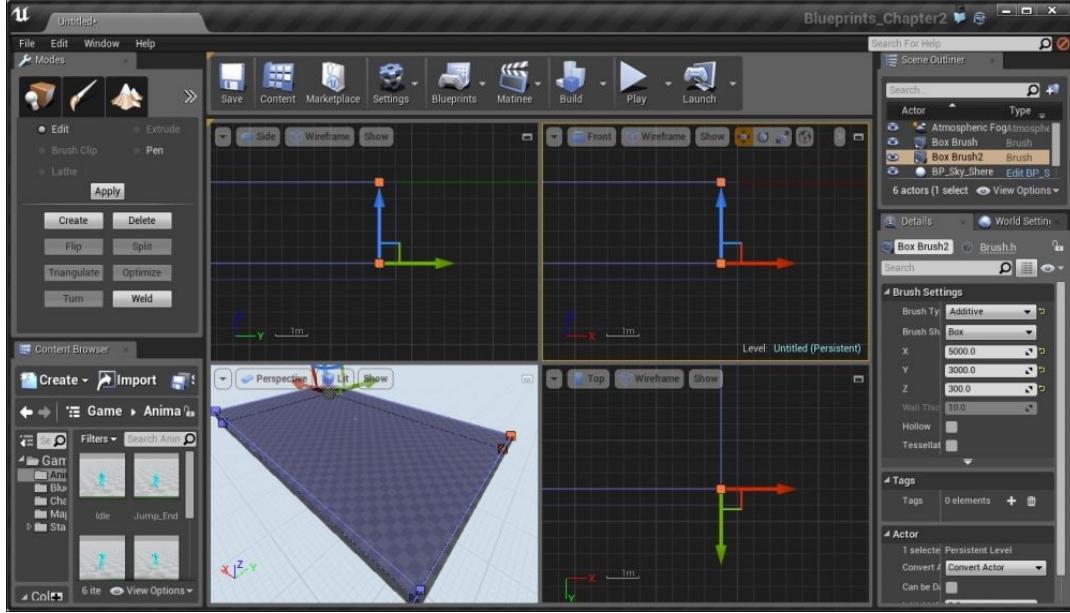
Duplicating the current brush upwards

7. Now, under the **Modes** tab, select the **Geometry Editing** mode, which is the furthest on the right. You'll notice that the edges or vertices of the selected brush will be larger than they were earlier. Select the two vertices on the left-hand side of the **Front** viewport and drag it one step (50 units) inward.



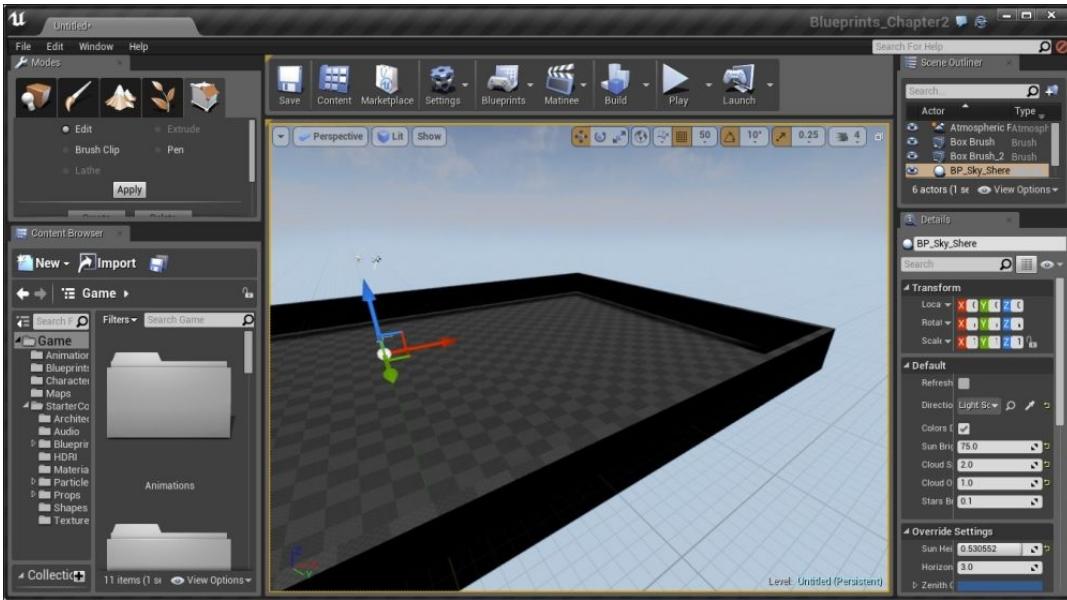
Selecting vertices via Geometry Editing mode

8. Do the same thing for the right side and the top of the brush. To do this for the top, which is much larger than the others, zoom out, select them both and then zoom in to do the movement.
9. Finally, we will need to move to the **Top** viewport. To do this, click on the **Front** selection and select **Top** and then drag the top and bottom vertices one step inward from here.



10. Finally, move back to the perspective view to see things clearly. Under the **Details** tab, change **Brush Type** to **Subtractive**.

A **subtractive brush** is used whenever you want to remove solid space, such as when you want to create a door or window. A nice change from Unreal Engine 3 is the fact that subtractive brushes only carves out space from the earlier created additive brushes so that we can place additional brushes on top of it.



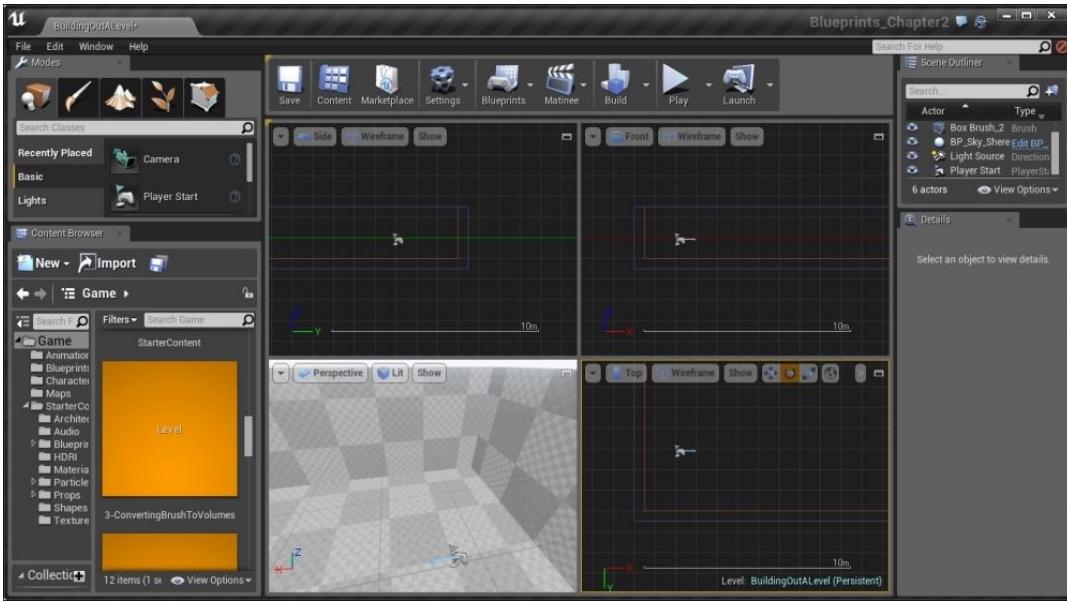
*The use of the subtractive brush*

With this, we now have an area that our player can inhabit. The player can jump on things that are approximately 200 pixels high, so we need to keep that in mind when developing our platforms as these walls are 250 pixels tall.

11. Go to the **Scene Outliner** tab on the top-right of the screen and then double-click on the **Player Start** object to center your camera on it. This is where the player will start the game from, so let's translate it to one of the edges of our map, pressing the *End* key to make it land on the ground.

#### Note

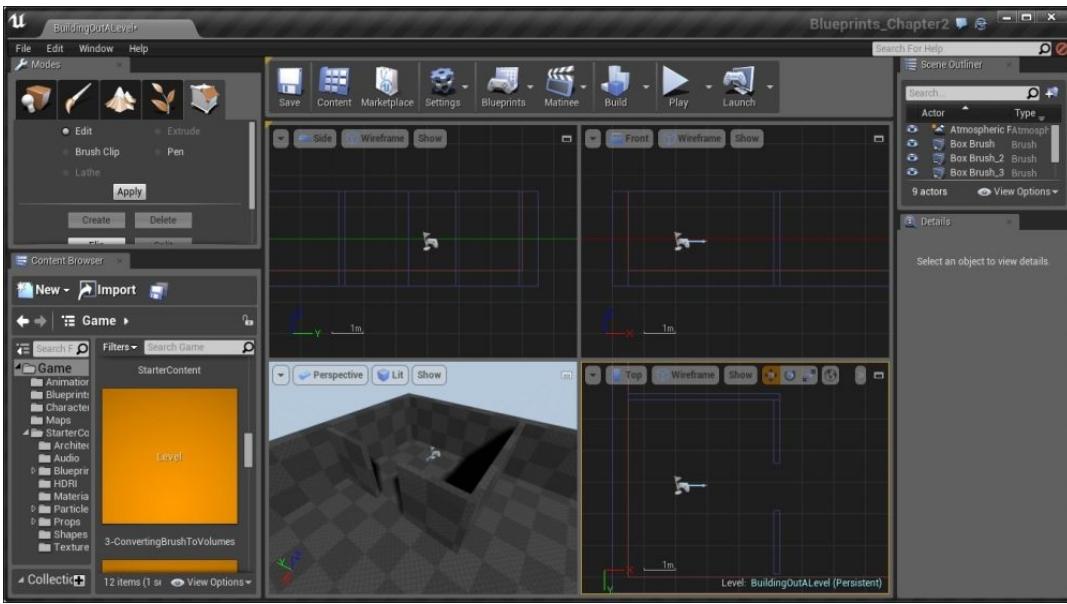
Note that on the **Player Start** object there is a light blue arrow pointing in a direction. This is the direction that the player will face when spawning. You can also use the rotation tool to rotate it to make the player face the direction you want.



12. Now, we want to give the player some guidance on where to go, so I'm going to close off some areas inside the level. Drag and drop a new box into the level. Give it a size of 20, 200, 300 and put it flush up against the wall.
13. Once this is done, create another duplicate and leave some space to create an opening for the player to walk through. Finally, duplicate the brush again, rotate it 90 degrees in the Zaxis and then use the **Geometry Editing** mode in order to have it fit the room by selecting the vertices using a marquee selection and then using the translate tool to move them into place.

#### Note

As a reminder, to do a marquee selection, click and drag some empty space on your screen and create a box that will hold the vertices you want to select. If you have a brush selected and you are in the Geometry mode, it will only select vertices from there.



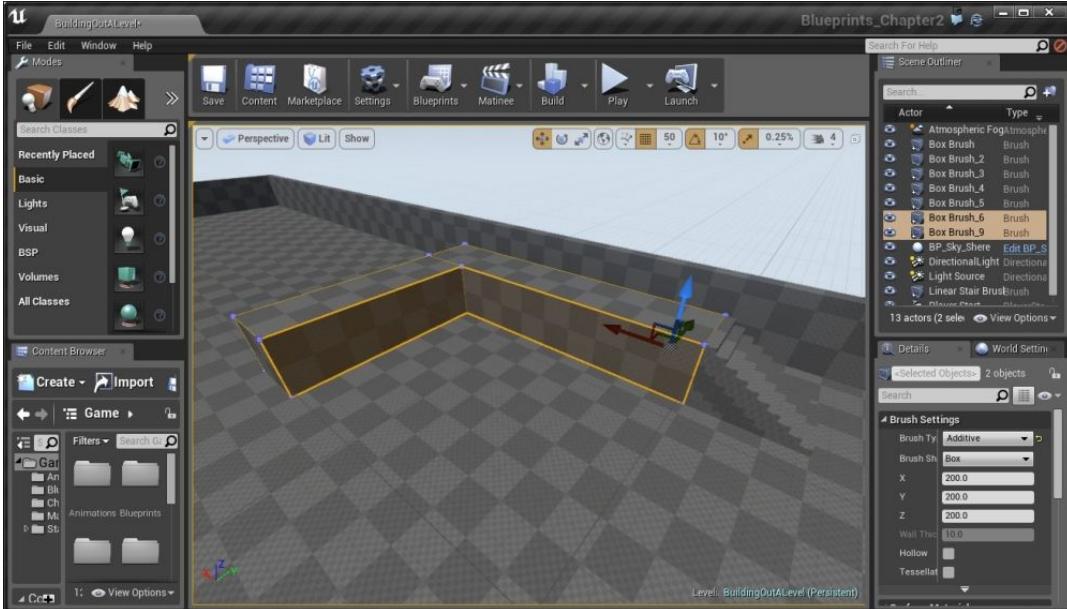
*Creating walls with openings for a player to walk through*

14. Next, let's build something a little more interesting, a staircase. We can do that by going into the **Modes** tab, selecting **Place**, going to the **BSP** section and then selecting **Linear Stair** and dragging it into the world and finally, placing it onto the ground.

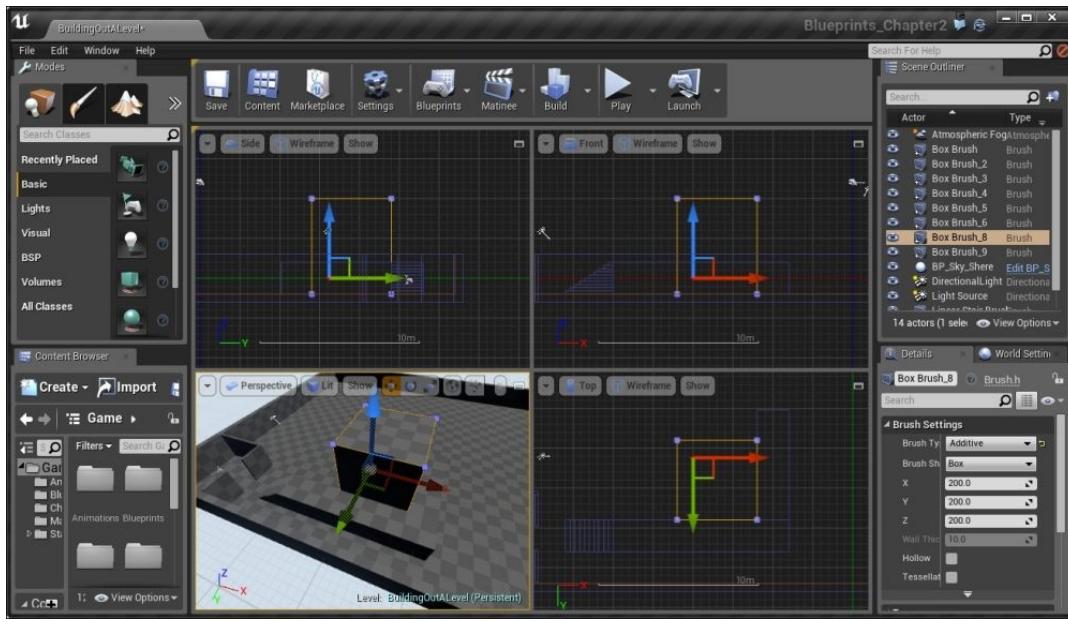
#### Note

By default, there are 10 steps and each step is 20 units tall, so by default, the stairs are 200 units tall.

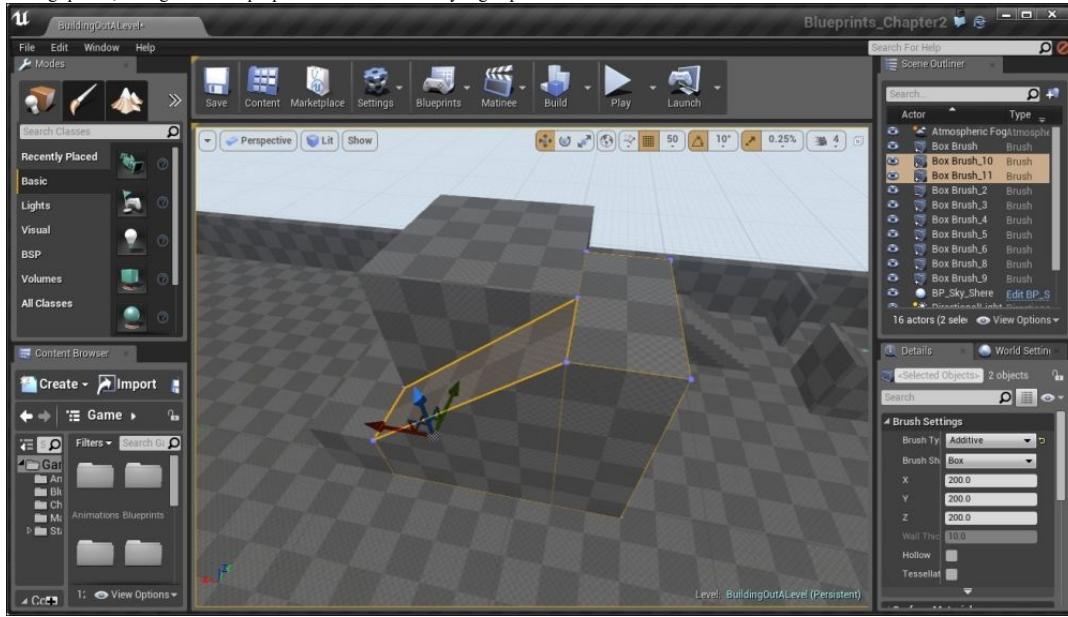
15. From this staircase, we are going to create a path to a tower. Create another box to create a walkway along the staircase. Make this walkway 11.00 units long. Duplicate the brush, rotate it 90 degrees along the **Z** axis, and have it go down only 700 units.



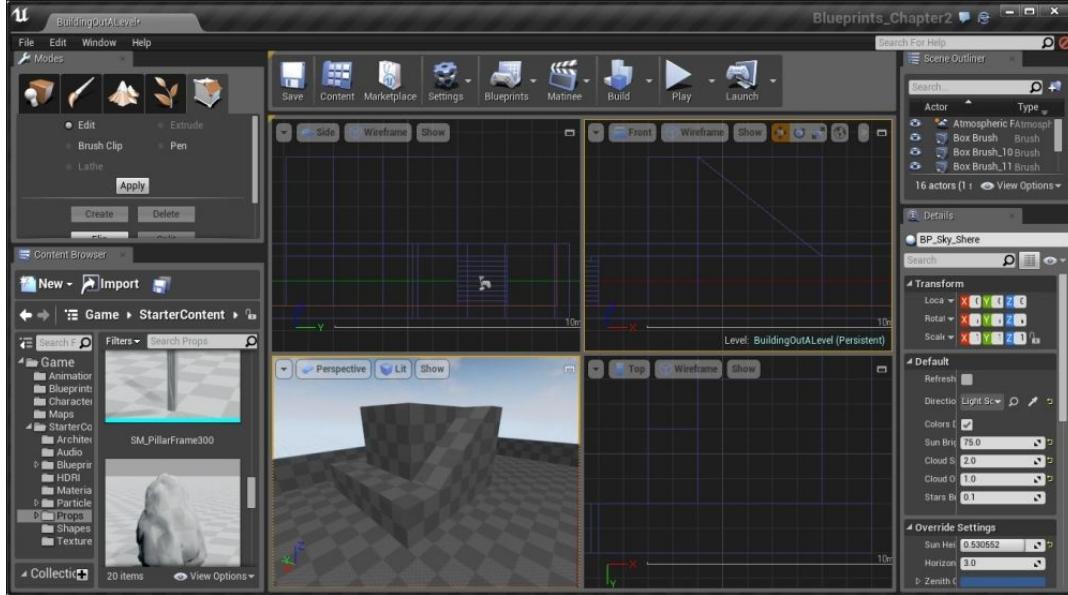
16. Duplicate the walkway again and have it only 500 units long. In the little 500 x 500 x 600 area, we've created a box to fill that hole.



17. To create a path for the player to get up there, on the last created walkway, click on the left-hand side vertices from the Front viewport and drag it up to match the new tower. It should look similar to a ramp heading up there, making it easier for people to add one final walkway to get up to there.

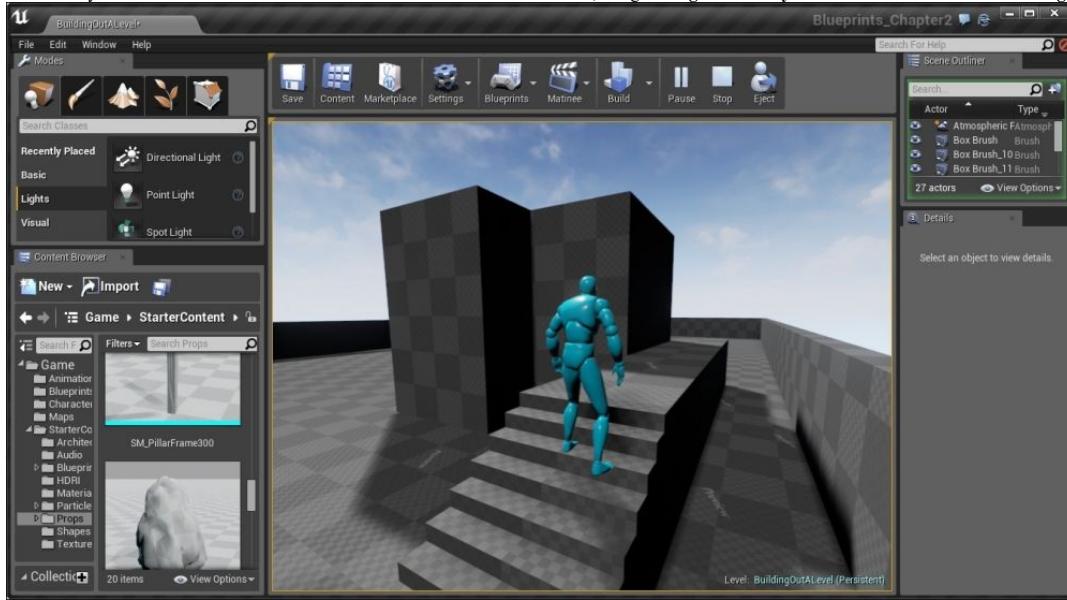


18. Once it's all put together, we should have something that looks similar to the following image:



19. In the **Perspective** viewport, it may be a bit hard to see what's going on. To help with this, place an additional **Directional Light** into the world with its rotation in the opposite direction of the **Light Source** object

that is already there so that we can see where the shadows are. With this in the **Details** tab, change the **Light's Intensity** to 1. This is what's referred to as a **fill light**.



*Placing an additional Directional Light as a fill light*

We now have a firm basis with which we can build even more complex and interesting levels!

## Applying materials to geometry brushes

Once you have your level blocked out, you'll probably want it to look better than just some greyboxes. We can very simply add some life to our world by adding **materials** to the surfaces of our brushes. Let's do this now.

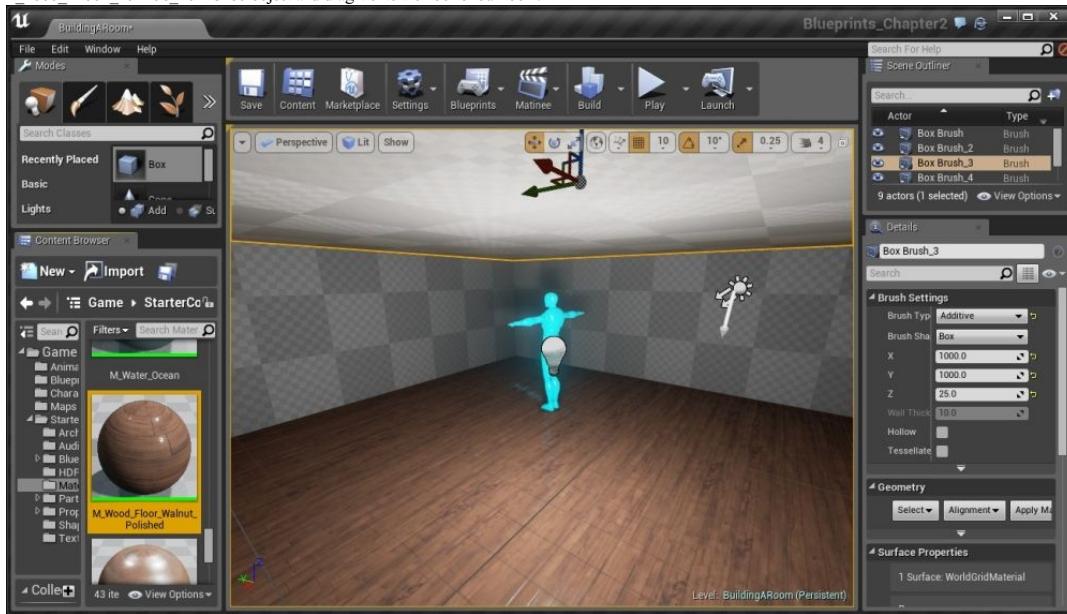
### Getting ready

This recipe assumes that you have a project open with the Sample Assets included as well as a room created with Geometry Brushes (BSP). If you do not have that yet, feel free to follow the instructions for the *Building a room* recipe.

### How to do it...

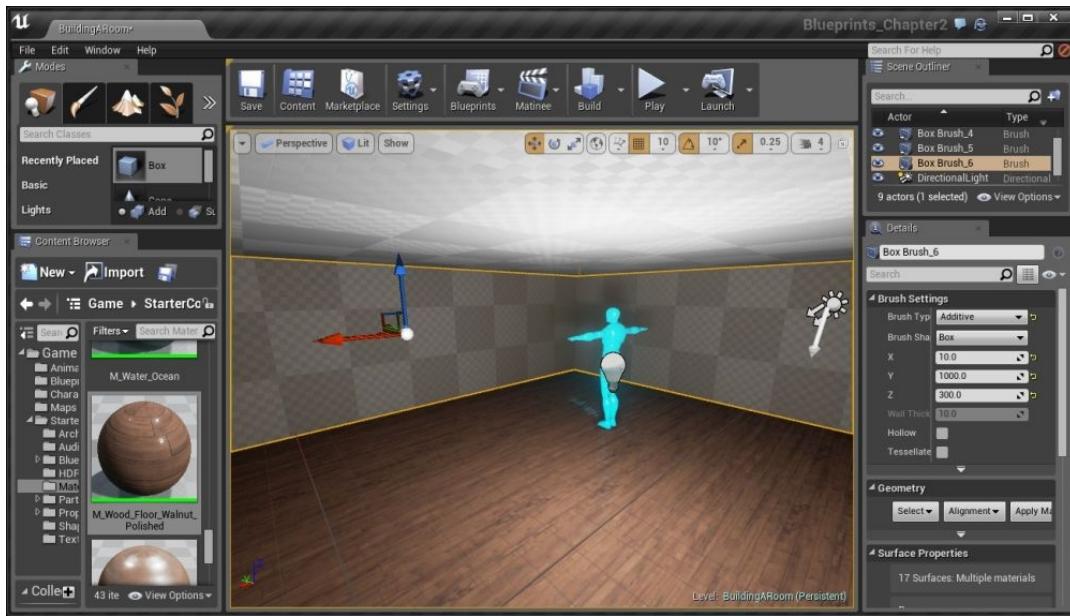
Now that we have our room created, let's add some materials to it:

- Inside the **Content Browser** tab, go to the **StarterContent/Materials** folder. Here, you'll see a number of materials that we can apply to the surfaces in our world. Select the **M\_Wood\_Floor\_Walnut\_Polished** object and drag it onto the floor of our room.



You can already see how it's making the room look a lot nicer. This works well for a one-time event, and technically, we can do the same thing for the walls, however, it can become quite tedious over time. Thankfully, there are some tools we can use to make our lives easier.

- Select one of the walls inside our room. Once there, go to the **Details** tab and under the **Geometry** section, click on the **Select** dropdown menu and then select **Select All Adjacent Wall Surfaces**.



3. You should see lines along all of the walls in place. After this, let's select a material to use for the walls (I am using `M_Basic_Wall`) and then back at the **Details** tab under **Geometry**, click on the **Apply Material** dropdown menu and then select **Apply Material: M\_Basic\_Wall**.
4. Finally, we will want to add a ceiling to our world. Select the `M_Concrete_Tiles` material and drag it onto the ceiling.

This is a great starting point, but there are some other features you may want to be aware of. For instance, this wooden floor looks fine, but maybe you want the planks of wood to be larger. You can very easily do this by going into the **Details** tab, and under **Surface Properties**, modifying the **Scale** value to `2.0` in the U and V directions and then clicking on **Apply**.

You may also want the planks to face horizontally instead of vertically. To do this, click on the **90** button in the **Rotate:** section. You can also move the textures a little at a time in order to get it perfect using the **Pan** option.



We can now work with materials on our brushes.

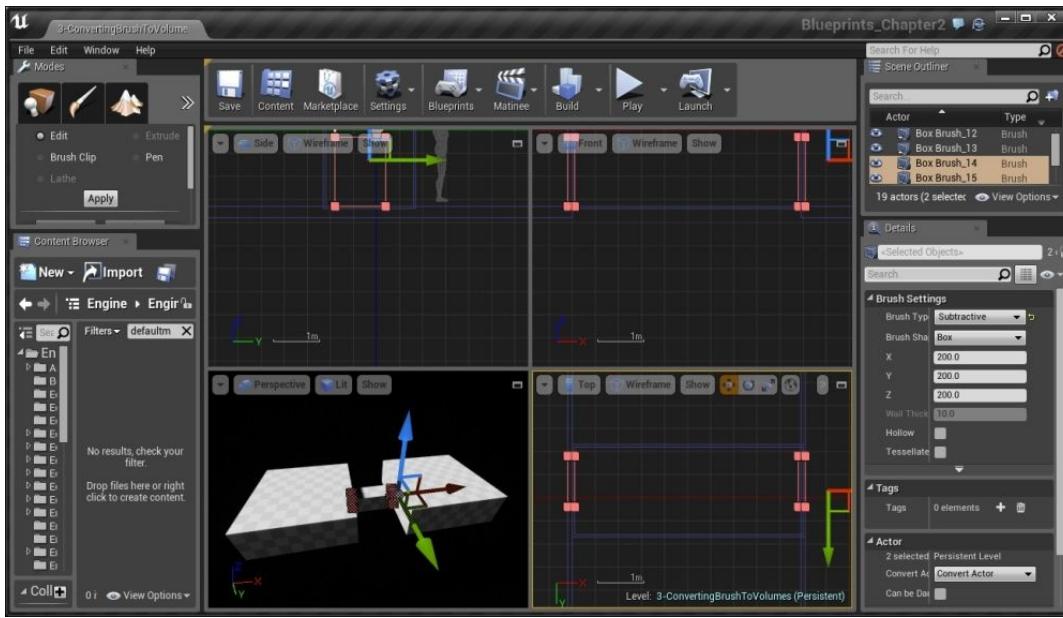
## Converting brushes to static meshes or volumes

As level designers, we will often greybox our levels to playtest and make sure that a level is fun and works correctly. Once we get to a point where the level is not going to change very much, we give our environment artists a model with the environment and then they can go in and make everything pretty with complex meshes or create pieces for us to build it out modularly. The first thing we will be doing in this recipe is creating the mesh that we can give to our artists. The second thing we will be learning is how to convert brushes into volumes. As you may be aware, the high-poly meshes that our environment artists would create could be very computationally expensive to check against collision with other objects at runtime. A trick that we will often use is to convert the brushes into a blocking volume that will still have the collision as we've been using and then filling the world with meshes to look the best it can.

In addition, you may have something like a doorway, and when you enter the hall, you want something to happen. We can use a Trigger volume for this. Rather than creating a trigger volume and then scaling it, it would be much easier to just use the doorway brush as a base.

### Getting ready

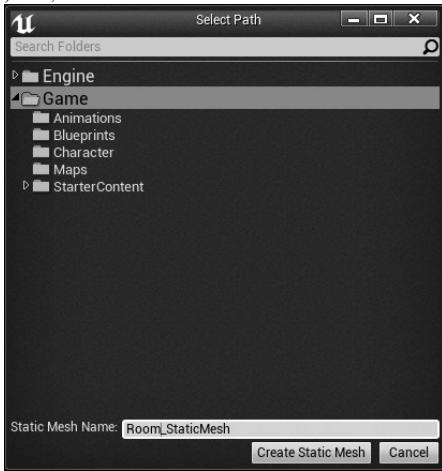
This recipe uses a level of two rooms connected via a hallway with a subtractive brush opening the way, which should be easily completed if you followed the *Building out a level* recipe. If not, you can open the **3-ConvertingBrushToVolume (Persistent)** setup level and use it for this task.



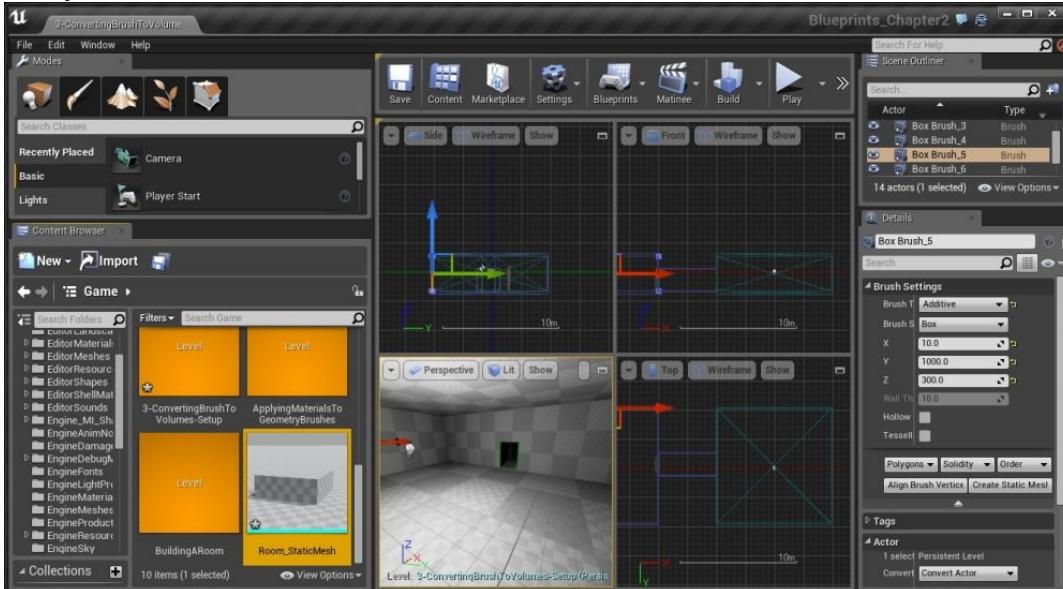
## How to do it...

Let's first export one of these rooms so that we can give them to our artist to work with:

1. If you have not already, minimize the Perspective viewport by clicking on the top-right minimize button in the viewport to go to the four-window default view. With this done, select all of the brush actors in the right room by doing a marquee selection. You may have to unclick on an object, such as a point light by holding the *Ctrl* key and then clicking on them. Once you have only brushes selected, in the **Brush Settings** section under the **Details** tab, click on the downward facing arrow to open up the advanced options and then click on **Create Static Mesh**. You'll be given an option to name it (I am putting *Room\_StaticMesh*). Then, click on **Create Static Mesh**.



2. Once this is done, if you go into the **Content Browser** tab and the **Game** folder, you should see your Static Mesh there! Finally, we need to export it so that our artist can open it. Right click on the mesh and then select **Export** and save it as an **FBX** file!



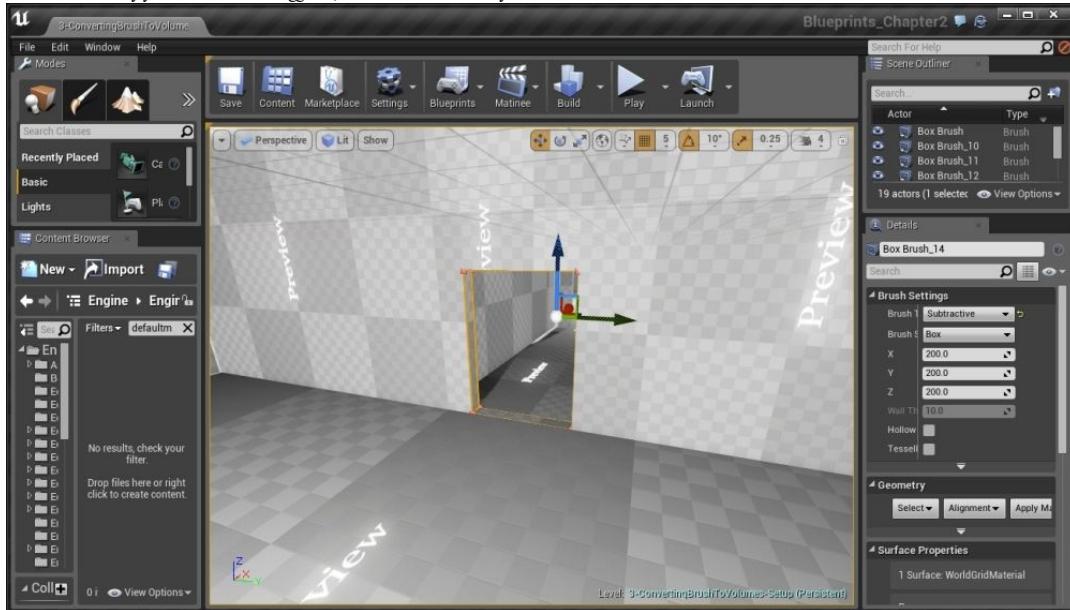
After this, you can give the model to your artist, and they will know the size of the area that you've created!

#### Note

It's important to note that after you turn something into a static mesh, you cannot convert it back to brushes again, so before doing this, I always suggest saving a copy of the level.

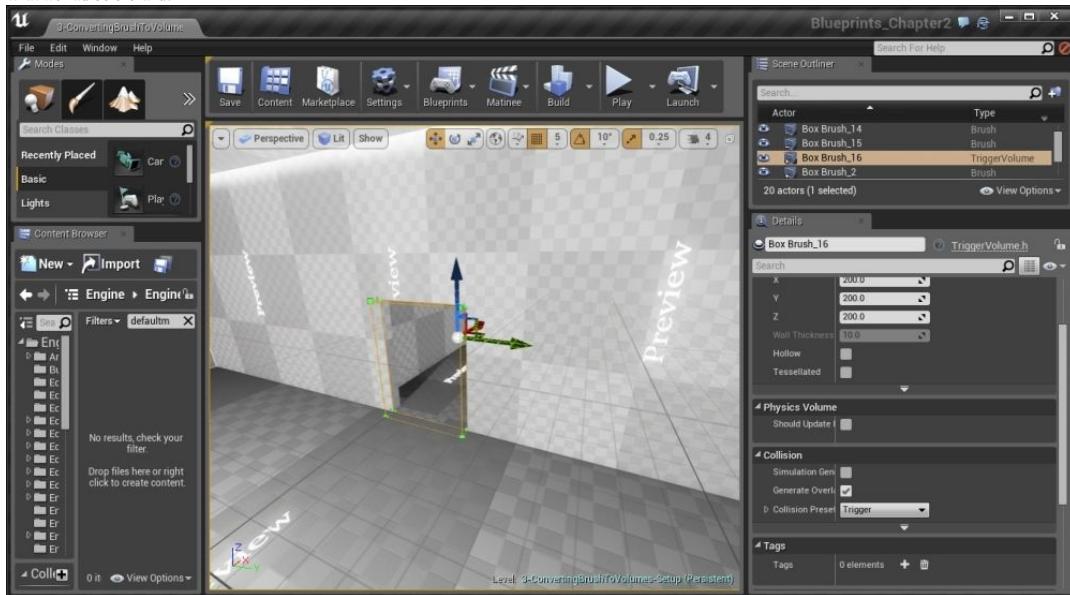
Another thing that we may want to do, is for something to happen when we enter the hallway, so we will want to have a trigger volume in the position. Rather than creating one from scratch, let's use our subtractive brush as a base.

- Inside of the hallway you want to add the trigger to, click on a wall created by the subtractive brush.



Selecting the subtractive brush

- Next, create a duplicate of this brush by holding the *Alt* key and then dragging it out somewhat on the X axis (red).
- In the Details tab under the Actor section, go to the Convert Actor action and select Trigger Volume. After a few seconds, you'll see that it's converted to have green edges and is of the same shape of brush as what we had beforehand!



With this, we now have a good idea on how to convert brushes into other brush-like things!

#### Note

If you are interested in learning what Trigger Volumes are and how they can be used, check out the *Using Trigger Volumes – opening a door using Matinee* recipe of [Chapter 8](#), *Blueprint Scripting – Level Effects*.

## Chapter 3. Creating Quality Interior Environments

In this chapter, we'll cover the following recipes:

- Placing static meshes
- Placing a particle system
- Using Groups
- Meshing an example map
- Adding life to static meshes

## Introduction

Continuing with the theme from the previous chapter, once we have created a level and have it playtested, only then we will want to make it look nicer.

In fact, level designers will often pass off the level geometry to an environment artist to recreate the level with new art assets.

Here, we will talk about the various ways in which we can polish our game to make the insides of buildings look great. Meshes will fill up a scene and give some depth to help with the suspension of disbelief, which is very

important in game development.

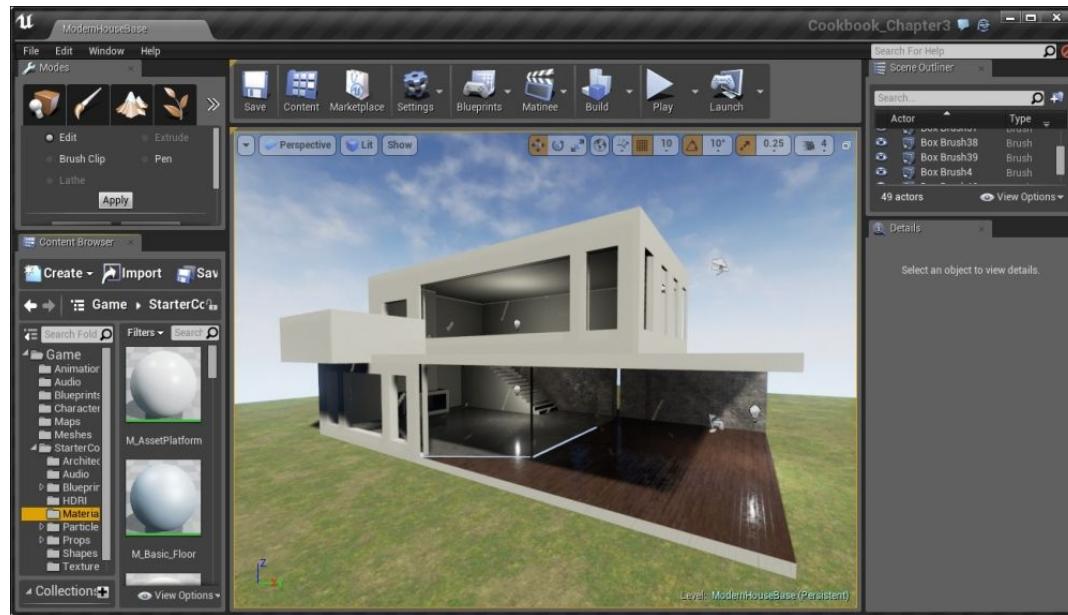
## Placing static meshes

Once we have our level at a point where it needs to be polished, one of the first things we can do is add meshes to decorate the level and make it feel more lived in. It's actually quite simple to do, so let's get started!

### Getting ready

This recipe assumes that you have a project open with the Sample Assets included as well as a room created with Geometry Brushes (BSP). I have provided a sample level that will be used for this demonstration (`ModernHouseBase`), which is included in the Example Code that you can access from Packt's website at <http://www.packtpub.com>. If you are not familiar with building levels using BSP, feel free to follow the instructions for the *Building a Room* recipe from [Chapter 2](#), *Level Design – Building Out Levels or Greyboxing*.

To open up the level files, create a new project with **Starter Content** included and then move the `Maps` folder (or just the `.umap` file) into your project's `Content` folder. You can easily access this folder from the Unreal Editor by going to the **Content Browser** tab and then right-clicking on the `Game` folder and selecting **Open with Explorer**.

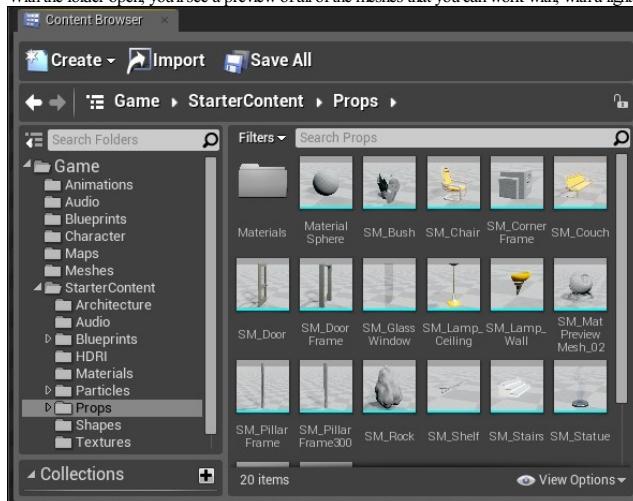


### How to do it...

Now that we have our level open, let's start off by placing a single static mesh into our level:

1. Open the **Content Browser** tab. We first need to find the mesh that we want to use. If you have **Starter Content** added, the meshes are located in the `StarterContent\Props` folder.

With the folder open, you'll see a preview of all of the meshes that you can work with, with a light blue border around them.

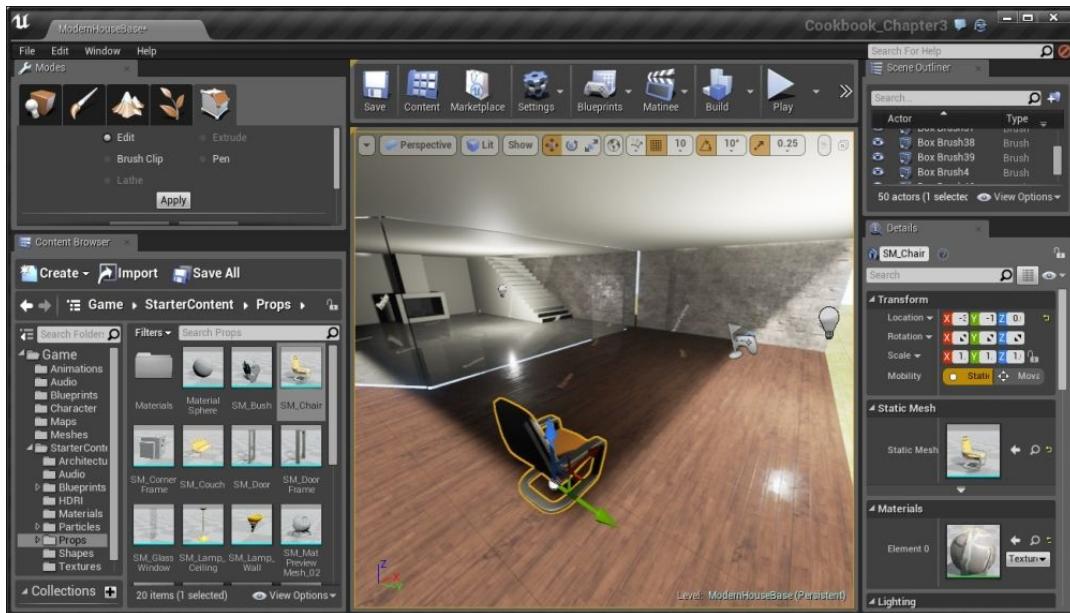


### Note

You can modify the size of the images by going to **View Options** and modifying the **Scale** slider. There are a number of other options that you can use as well to customize the Content Browser tab to your liking, such as changing **View Type** of the contents from the default **Tiles** to **List** or **Columns**.

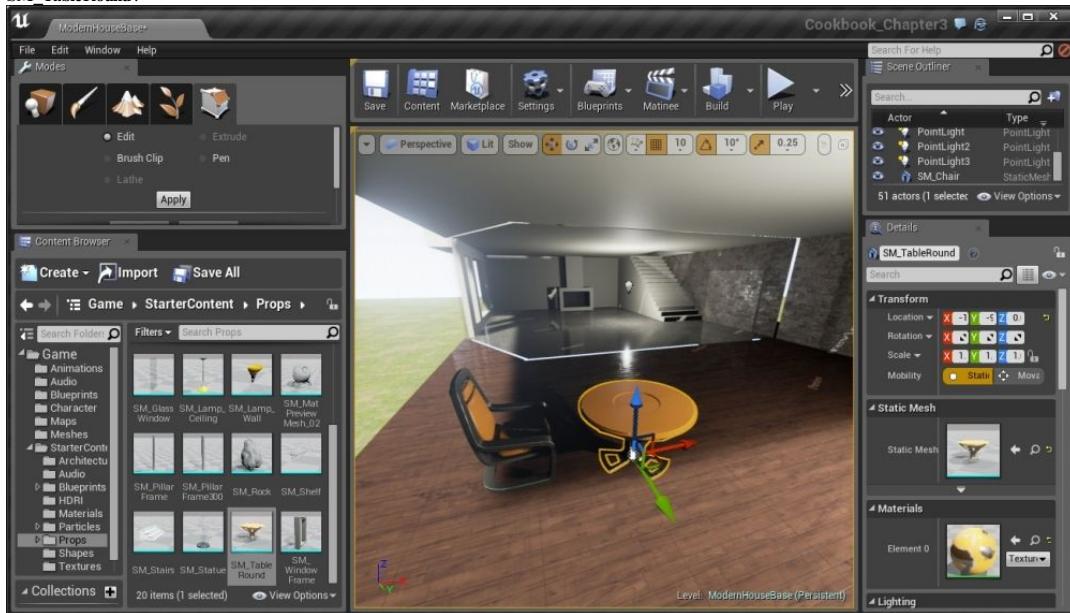
For more information on customizing the Content Browser UI, refer to <https://docs.unrealengine.com/latest/INT/Engine/Content/Browser/UI/index.html>.

2. Click on the `SM_Chair` mesh and then drag it into the level on the wooden floor of our level. Once here, release it to place the mesh into the world.



Placing a chair mesh on the level's floor

- Let's add one more mesh to our level in another way. Select the `SM_TableRound` object in the **Content Browser** tab and then right-click inside the level, near the table. From here, go to **Place Actor | SM\_TableRound**.



With this, we've learned two ways to add a static mesh to our level!

Note that just like working with brushes, whenever an object is selected, the **Details** panel will fill itself with information about the actor that was added. From here, we can modify the **Scale**, **Location**, and **Rotation** values of the added meshes and can also change the **Mobility** from **Static** to **Moveable**, which we will use later.

## Placing a particle system

In a designer's toolbox, one of the most effective tricks used to make things look stunning are particle systems. We can spawn a large number of particles which are small simple images or meshes, without much of a performance hit. Particle systems control these particles and their display and movement. They are very useful for elements such as liquid, smoke, clouds, magic, and in this instance, fire, as we bring life to our house's fireplace.

### Getting ready

This recipe assumes that you have a project open with the Sample Assets included as well as a room created with **Geometry Brushes (BSP)**. I have provided a sample level (`ModernHouseBase`), which will be used for this demonstration. It is included in the Example Code, which you can access from Packt's website. If you are not familiar with building levels using BSP, feel free to follow the instructions for the *Building a Room* recipe of [Chapter 2, Level Design – Building Out Levels or Greyboxing](#).

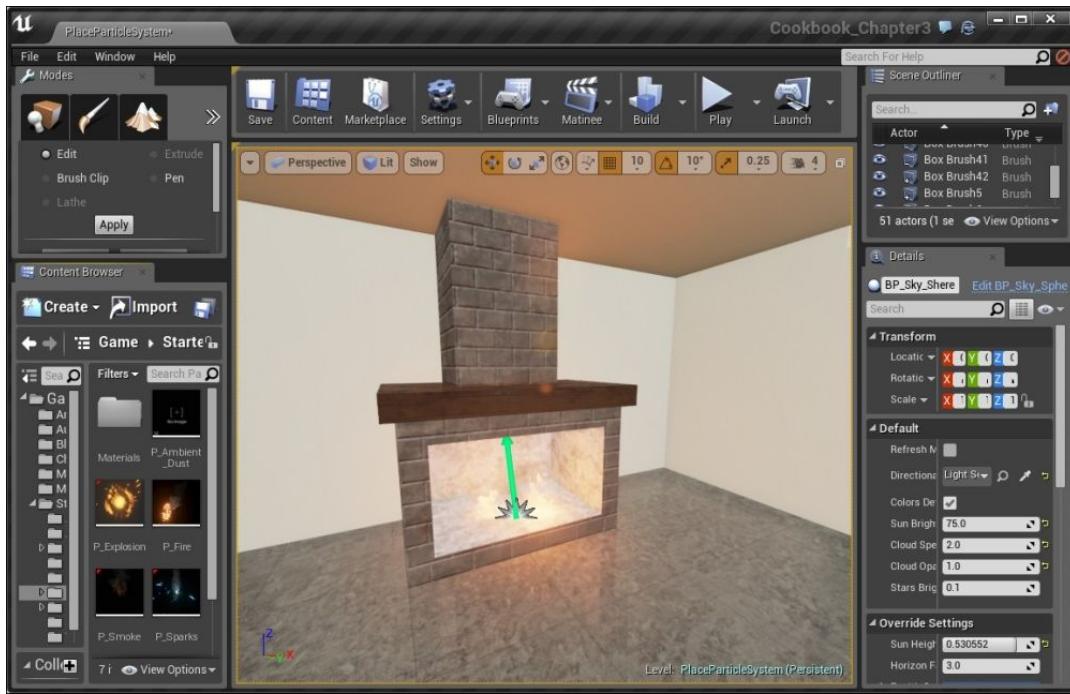
### How to do it...

Now that we have our level open, let's start off by placing a particle system into our level:

- With the level opened, move your camera to the fireplace on the bottom story.
- Adding particle systems is very similar to adding a static mesh. Open the **Content Browser** tab, and we will first need to find the particle system that we want to use. If you have the Starter Content added, the particle systems are located in the `StarterContent\Particles` folder.

With the folder open, you'll see a preview of all of the particle systems that you can work with, with a white border around all of them.

- Click on the `P_Fire` system and then drag it into the level into the fireplace. Once here, release it to place the particle system into the world.
- Now, starting off the particle system will be too large for our fireplace, but just like working with the Geometry Brushes, we can also modify the properties here under the **Details** tab. Change **Rotation** to  $-90, -90, -90$  and **Scale** to  $.5$  in all the axes.



## Tip

A quick way to scale all of the axes together at once is by going to the **Details** tab with the object selected and then, from the **Transform** component, clicking on the lock button to the right of the three parameters of the **Scale** property. Once we've locked the axes, we can then put in a .5 value in one of the parts and the other two will change as well. This will make the object always maintain its proper shape.

At this point, we now have an idea of just how easy it is to make our level look much nicer!

## Tip

For more information on creating custom particle systems of your own refer to <https://docs.unrealengine.com/latest/INT/Engine/Rendering/ParticleSystems/index.html>.

## Using Groups

When working on large projects, it's important to keep all of your content organized. With that in mind, it's a good idea to keep your levels organized as well. In this section, we will learn how we can use Groups to make your life easier. This will allow you to easily manage multiple Actors at once.

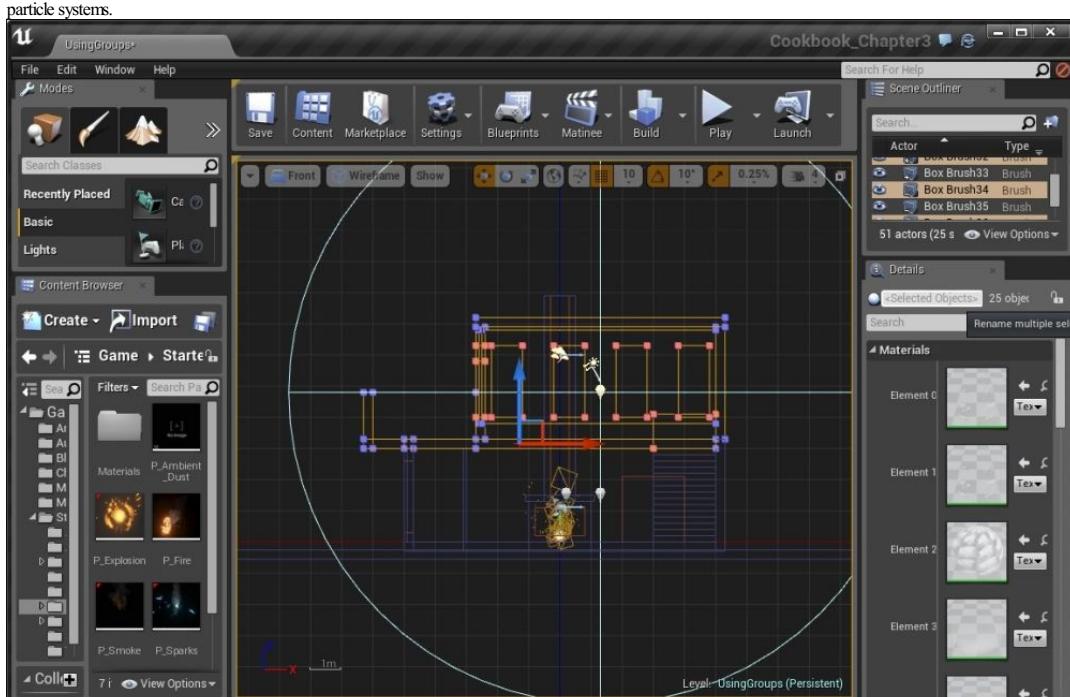
### Getting ready

This recipe assumes that you have a project open with a level created. I have provided a sample level that will be used for this demonstration (`ModernHouseBase`). It is included in the Example Code that you can access from Packt's website at <http://www.packtpub.com>. If you are not familiar with building levels, feel free to follow the instructions for the *Building a Room* recipe from [Chapter 2](#), *Level Design – Building Out Levels or Greyboxing*.

### How to do it...

Let's see just how easy it is to create a group:

1. The first group we are going to make is for the second floor of our house. With that in mind, go back to the four viewport view by clicking on the top-right minimize button.
2. From the Front (also seen as **World Outliner** in future version of UE4) viewport, click and drag from the top-left of the house to the floor of the second floor/ceiling of the first floor. Once finished, hold the **Shift** button and select any items you may have missed in either the viewport or in **Scene Outliner** (also seen as **World Outliner** in future version of UE4), and remove the two brushes used for the fireplace as well as particle systems.



## Selecting items on the second floor

### Tip

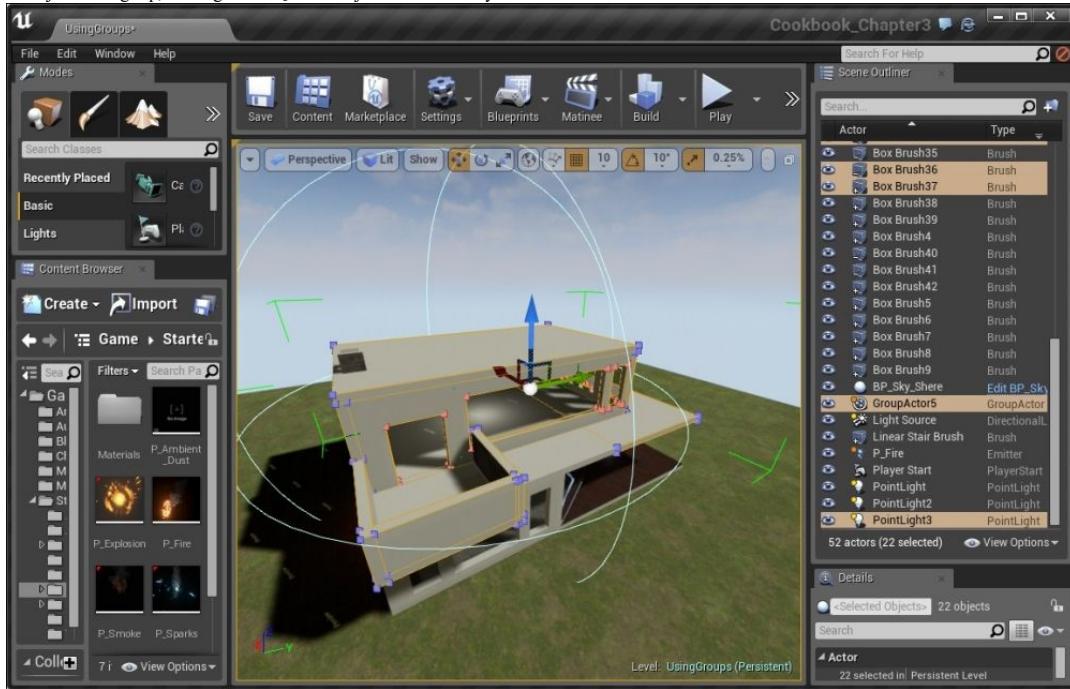
While selecting objects, you can also hold *Ctrl* and click to select or deselect the unwanted brushes or objects that were selected.

3. With the items you want to create selected, right-click and select **Group** or press *Ctrl + G*.
4. Now deselect the items and then click on one of them again. You'll notice now that all of the objects are selected if you select any of the items.

### Tip

If you do not see the **Group** option on the context menu, make sure that **Allow Group Selection** is enabled from the **Settings** menu located on the top toolbar.

When you create a **Group**, an object of the **GroupActor** type is created inside the **Scene Outliner** (in our case, it's named **GroupActor5**, but it may be a different number in your case). Just like selecting any of the objects in the group, selecting the **GroupActor** object will automatically select all of the others as well.

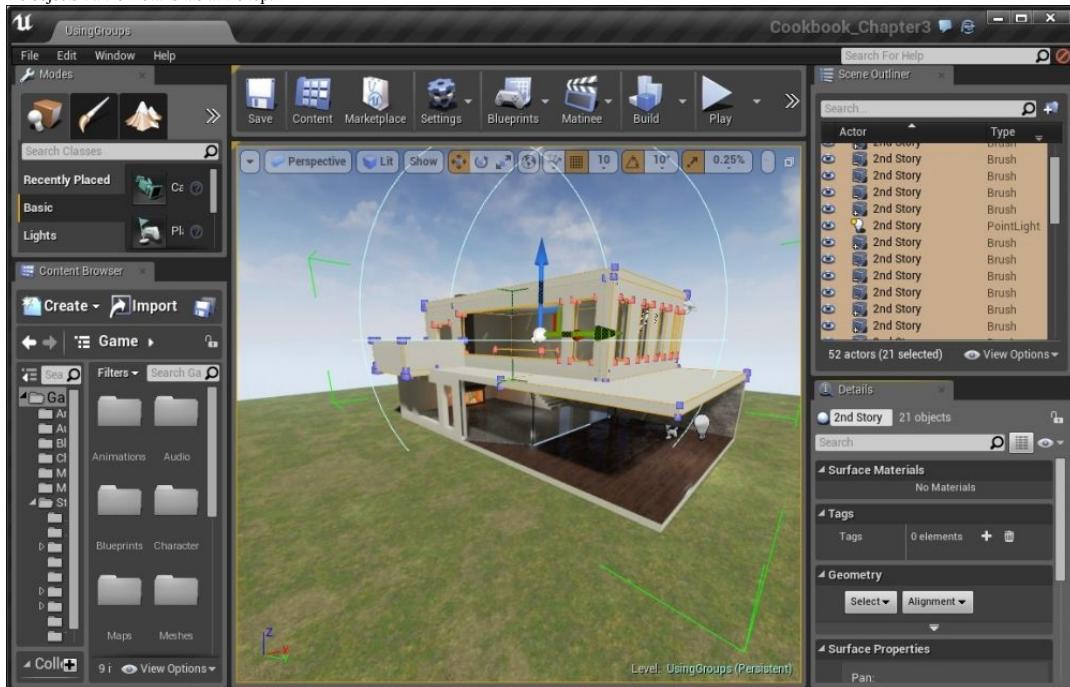


5. One of the useful properties of having a group is that you can hide the meshes by hitting the *H* key or pressing the eye icon to the right of the actor's name in the **Scene Outliner** or by right-clicking and navigating to **Visibility | Hide Selected**.

### Tip

You can click on the closed eye icon in the **Details** panel or press *Ctrl + H* to unhide all the objects that are currently hidden.

6. Another thing that you can do to make it easier to tell which group objects are attached is to rename the objects as the name of our group. To do this, go to the **Details** tab and select those brushes. Then, at the top, change the name of the objects to **2nd Story** by either double-clicking on the name in the **Scene Outliner** tab or selecting the object/objects in it. After that, press *F2* and edit the name. You can also rename the objects via the **Details** tab at the top:



## Renaming objects as name of a group

With this, we now have a good foundation of how we can use Groups to make our levels nicer!

## Note

For more information on Grouping, refer to <https://docs.unrealengine.com/latest/INT/Engine/Actors/Grouping/index.html>.

Alternatively, you may also create a folder in the **Scene Outliner** tab and put levels inside that. That way you can organize scene contents in whichever way you'd like.

## Meshing an example map

Watching a single mesh being placed is nice and all, but actually watching an example workflow is one of the best ways to see some of the tricks you can use to really polish a level up. In this section, we will be doing just that.

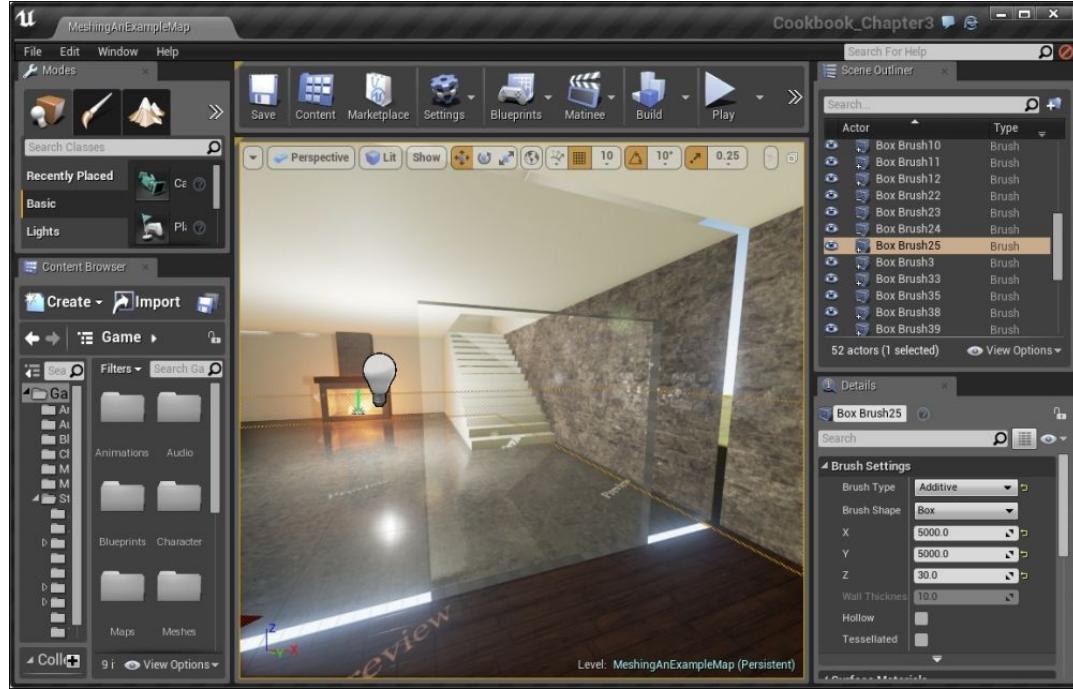
### Getting ready

This recipe assumes that you have the example project and the level I provided (`ModernHouseBase`) opened that is included in the example code which you can access from Packt's website.

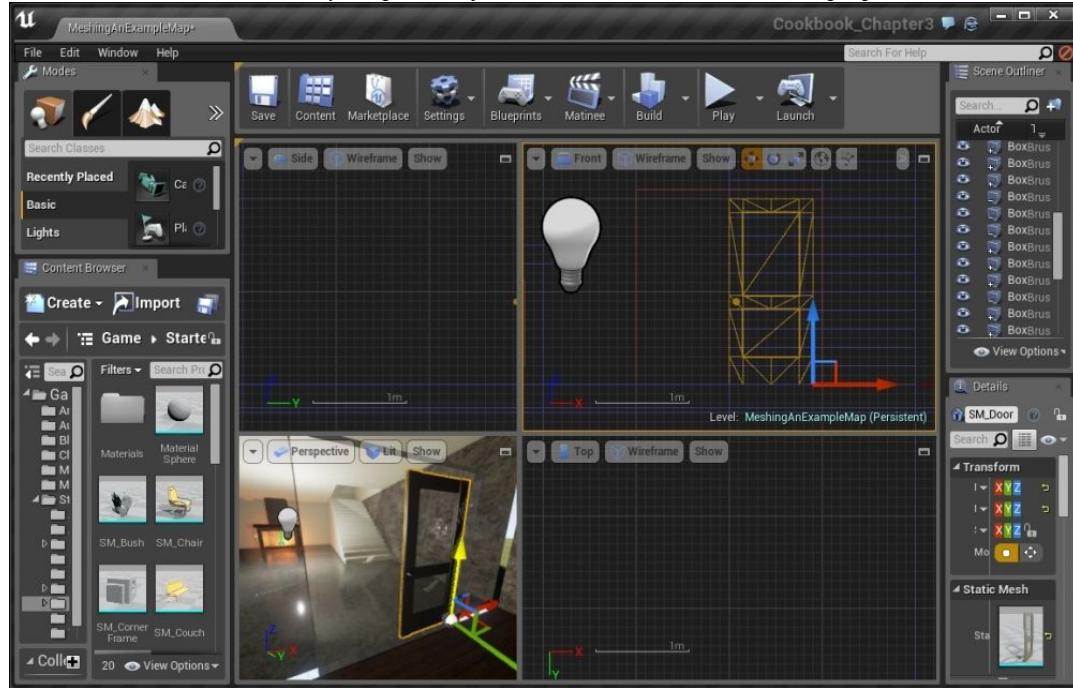
### How to do it...

Let's first export one of these rooms so that we can give them to our artist to work with:

1. Let's first add in a door. Move your viewport to the first floor of the house to the right of the deck where you can see the glass opened up.

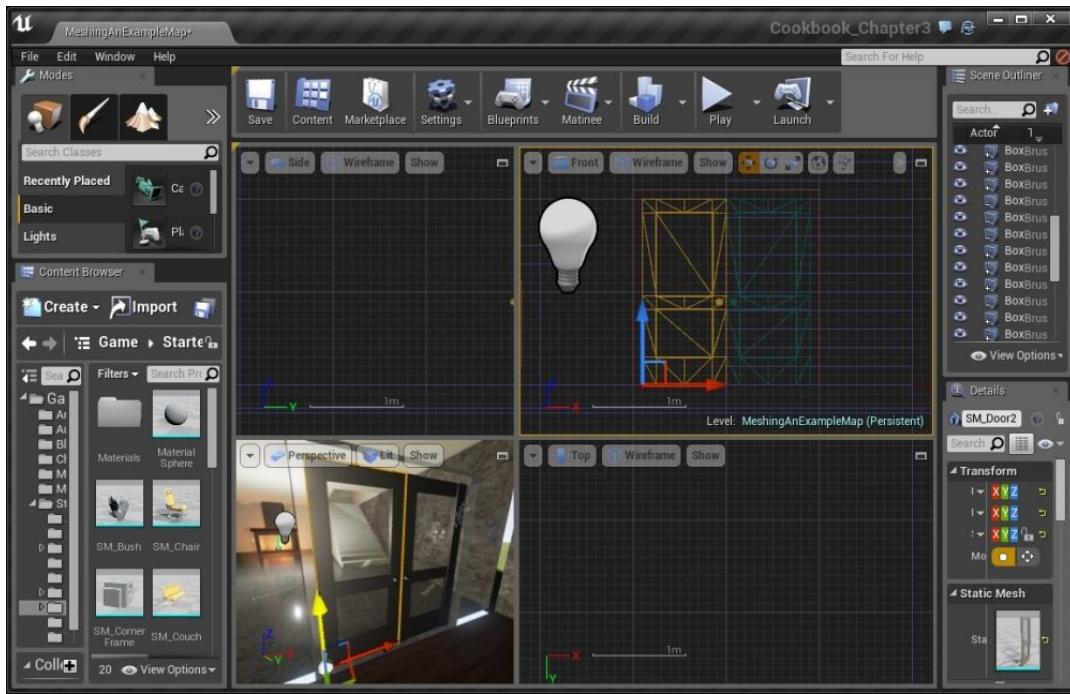


2. Once there, let's access the `StarterContent/Props` folder and select the `SM_Door` object and drag it into the world. Once placed, it will need to be rotated to face our doorway (~90 degrees). After this, use the Translate tool to move the door into the doorway, leaving 10 units of space in the Z and X directions, as shown in the following image:

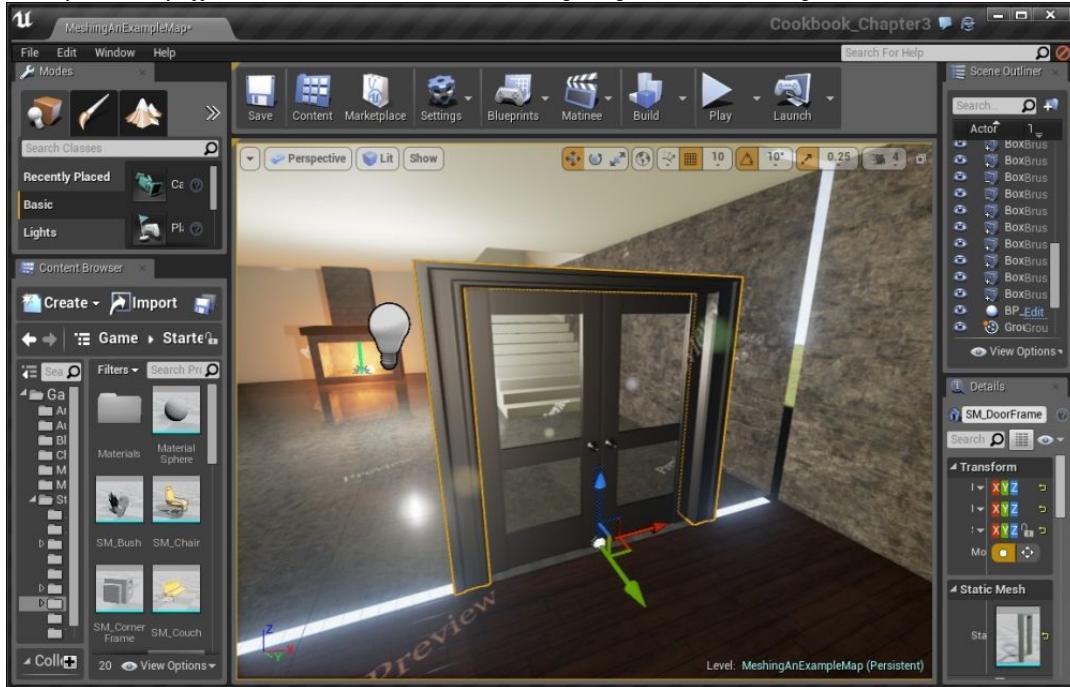


Placing a door

3. Now select the door and duplicate it from the **Front** viewport by holding down the `Alt` key and dragging it in the **X** axis. We don't want the doorknob to be on the same side, so we will want to flip the door by right-clicking and navigating to **Transform | Mirror X**. Then, move the door over to be flush with the other one so that they are touching each other.



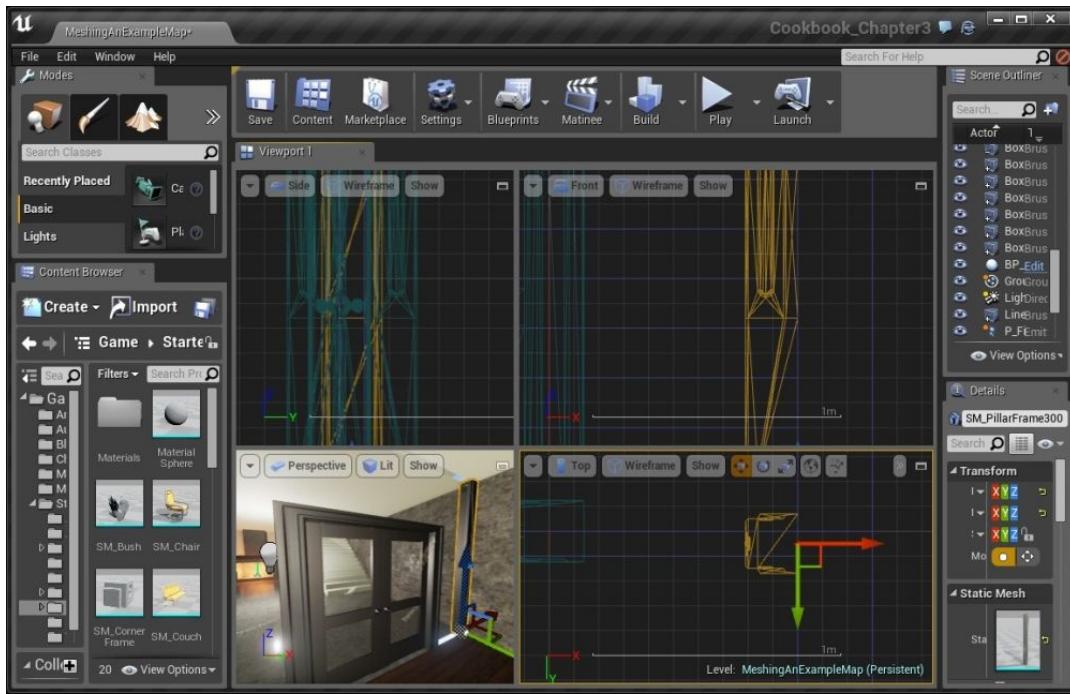
- Next, let's add in the doorway. From the **Content Browser** tab, select `SM_DoorFrame` and drag it into the world.
- It's initially created to only support one door at a time, but we can fix that with some slight changes. In the **Details** tab, change the **Scale** value in the Y axis to `2.0`. After this, rotate the door to fit our doorway.



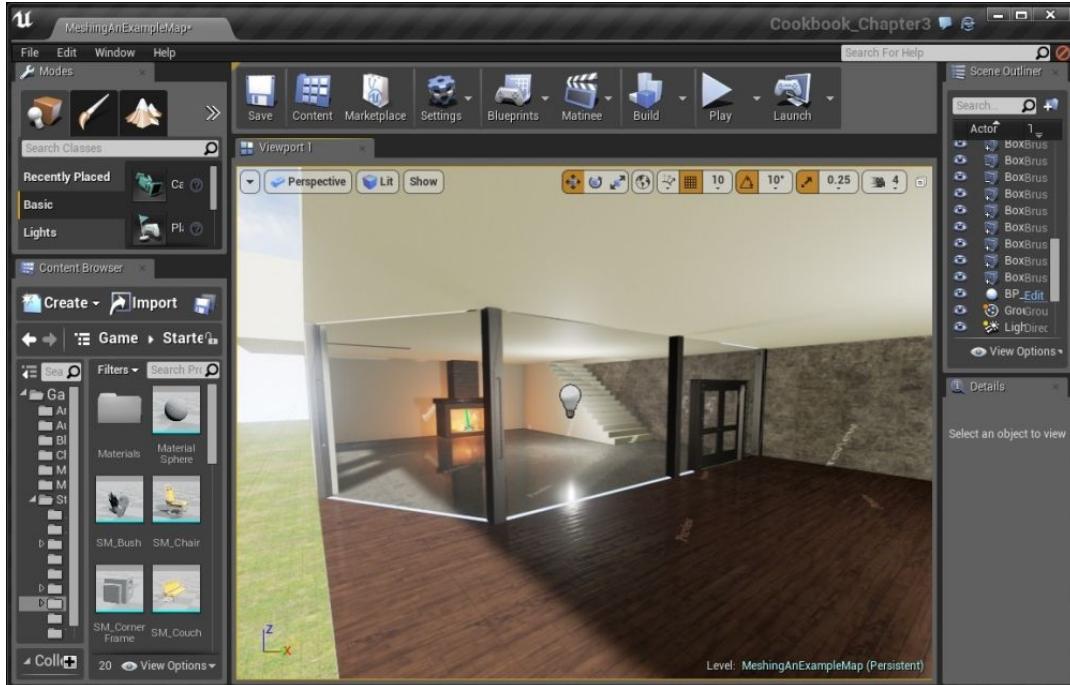
- The next thing you may notice is the edges on our glass look strange. Rather than creating some additional brushes to fill it in, we can use some additional meshes to make our level look much nicer and more detailed, and cover up these tiny issues that occur. In the **Content Browser** tab, select the `SM_FillerFrame300` object and then place it to cover our wall (rotated) so that it can face the glass. In order to fix the mesh in the middle of the glass, you may need to temporarily change the snapping to `5`.

#### Tip

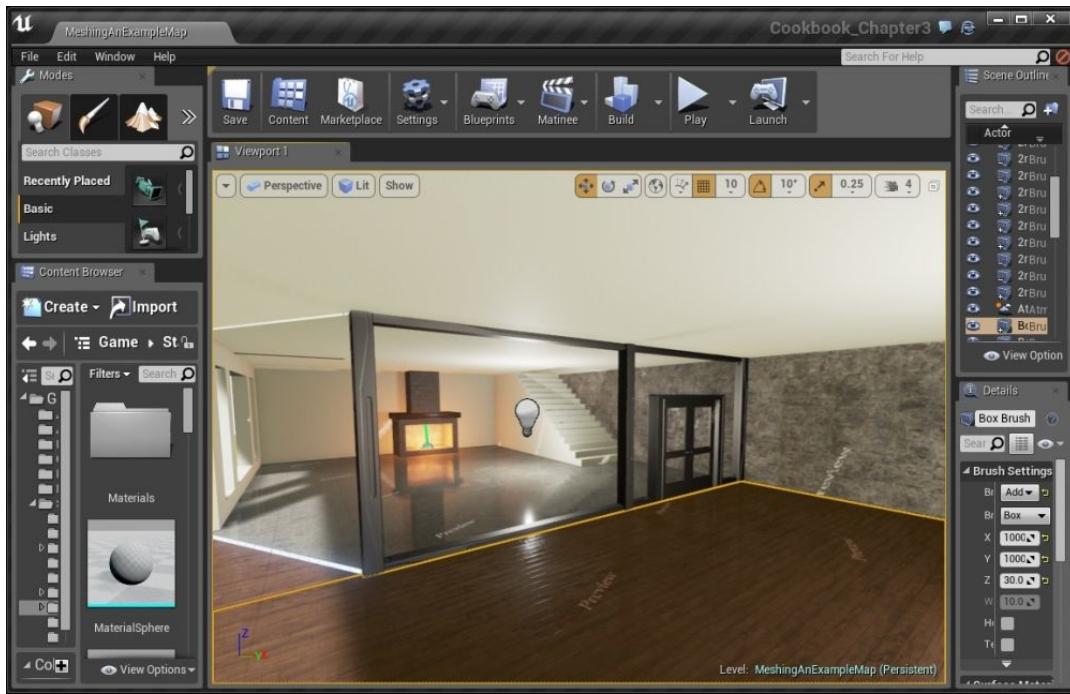
As a reminder, to change the snapping of the grid you can press the left/right bracket keys (`/` / `\`), which will decrease and increase the snapping respectively, otherwise, you can adjust the snapping by clicking on the dropdown menu next to the grid icon at the top of the main viewport window.



7. After this, we will duplicate the frame to fit it at an approximately equal distance from the door as the first frame for symmetry. Then, we will place the frame at the edges of the glass doors to help make the glass feel more realistic.

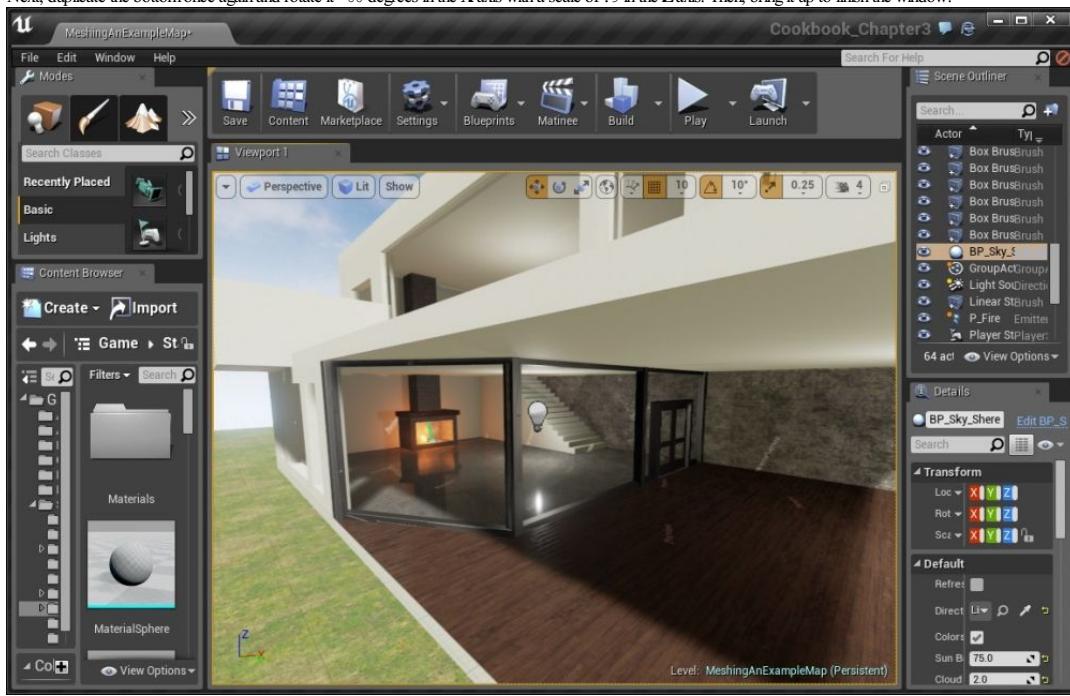


8. Next, we will need to take care of the tops and bottoms. From the **Content Browser** tab, select **SM\_PillarFrame** and rotate it down along the **Y** axis so that it faces the floor. Once you complete that, scale it to **.1.2** in the **Z** axis and place it between the door frame and the edge of the glass.  
 9. After this, duplicate it upwards with a scale of **.1.95** to fill up the top and then duplicate it once more with a scale of **.2** to fill the gap on the other side of the door. When you're done, it should look similar to this:

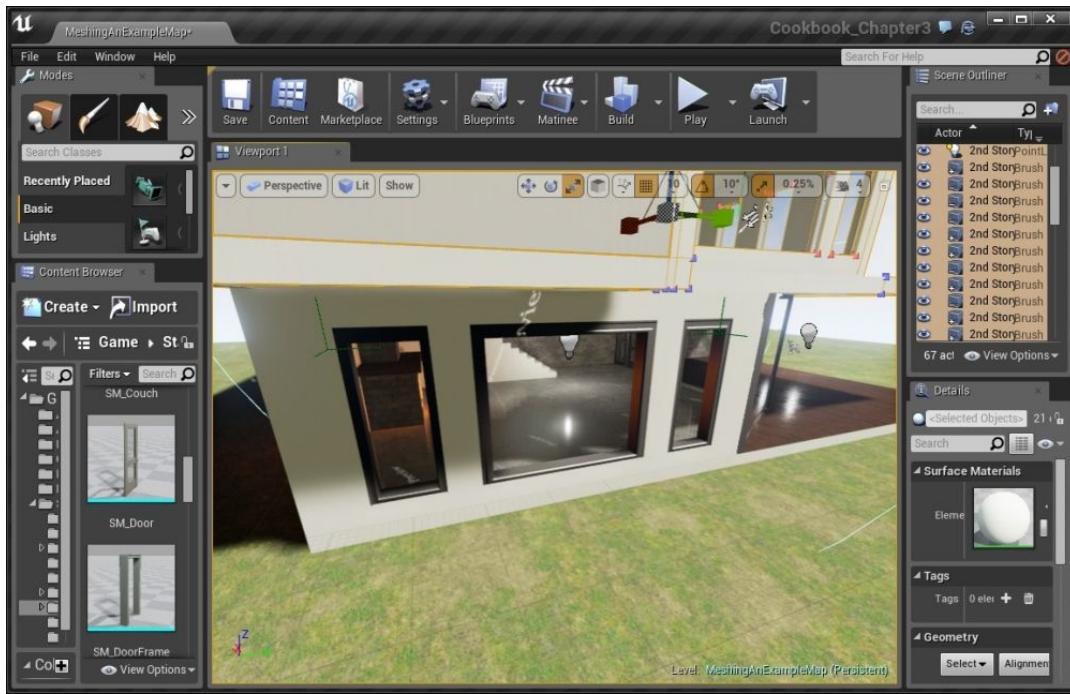


*Scaling the objects to fit the frame*

10. Next, duplicate the bottom once again and rotate it -60 degrees in the X axis with a scale of .9 in the Z axis. Then, bring it up to finish the window!



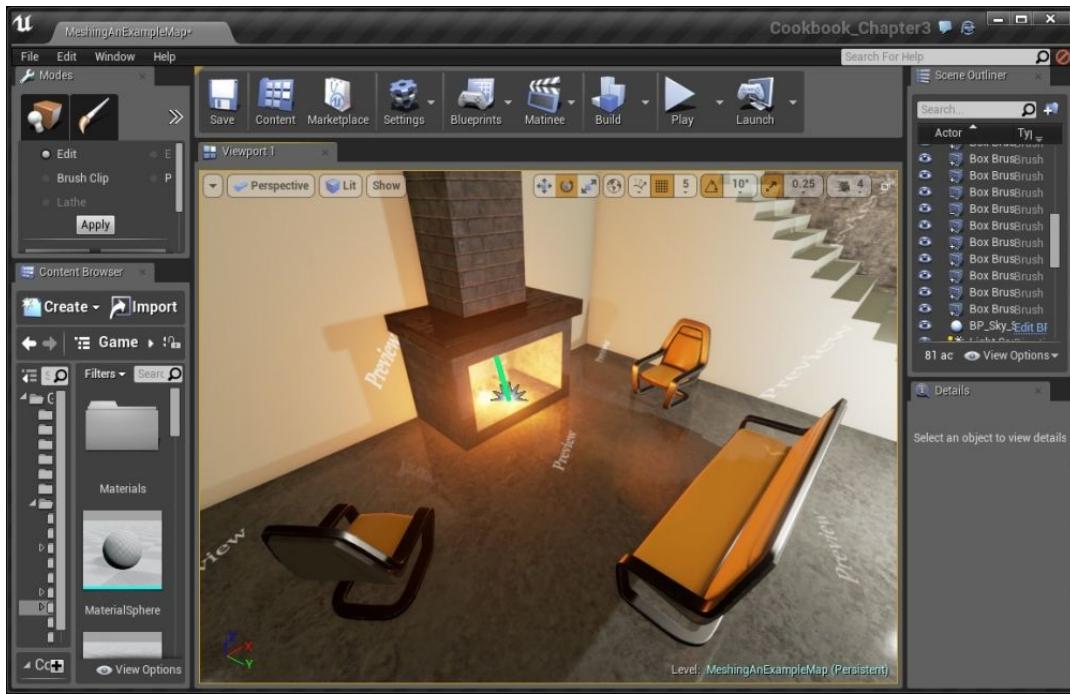
11. Now that we have this section down, we can move over to the right to tackle the windows. With this, we can select the SM\_WindowFrame and drag it into place with the smaller slot. Just like earlier, we can scale the object to fit it in the whole we've created. In this instance, we will set it to (1.2, 1.1 and 1.7). To fit the larger middle window, increase the Y Scale until it fits (I am using 3.5).



12. With this knowledge, we can also fill out all of the windows and the door on the second floor:



13. Next, let's create a little decor inside the actual house. Move back to the fireplace and place a `SM_Couch` object in front of it. After that, place two chairs on each side, making sure to rotate them a little bit toward the fire. Giving some difference in angle will help to make the world seem more realistic because objects in the real world that humans interact with are almost always not at a fixed angle (except architecture).



And with this, we have a good idea about how to add meshes to a level to add life and realism to it!

## Adding life to static meshes

Though it is nice having a level with a lot of static meshes (this is by default), they don't move at all (hence, they are called static). Static meshes are used for efficiency's sake, but there comes a time where you want the player to be able to interact with those objects, such as moving crates or shooting at things.

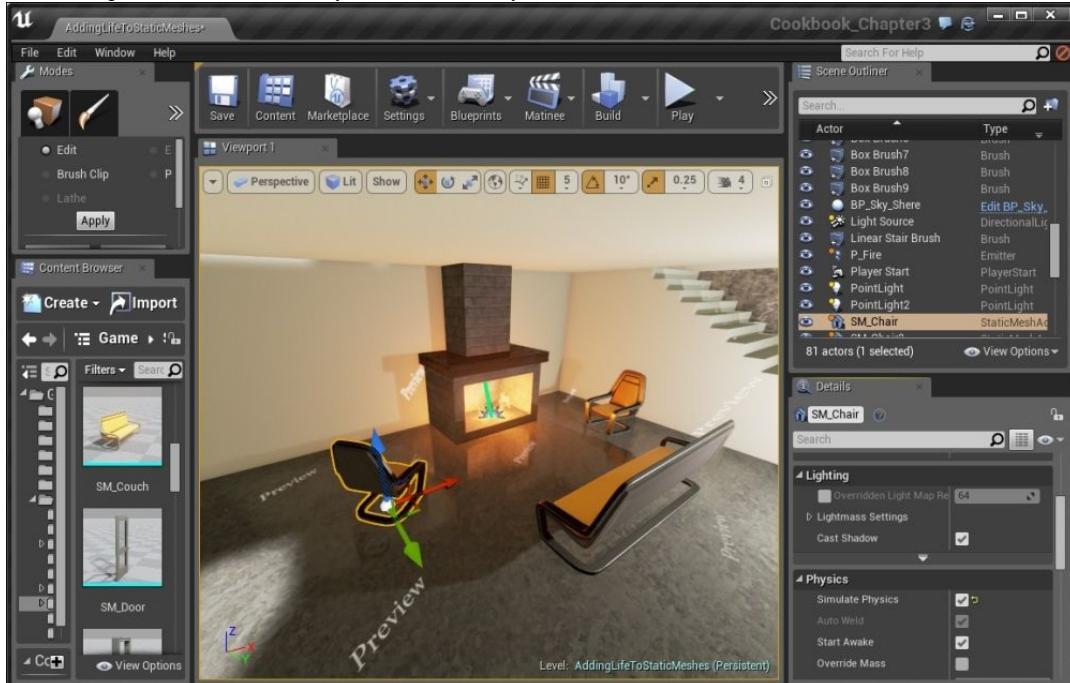
### Getting ready

This recipe assumes that you have the example project and the level I provided (`MeshingAnExampleMap`) opened.

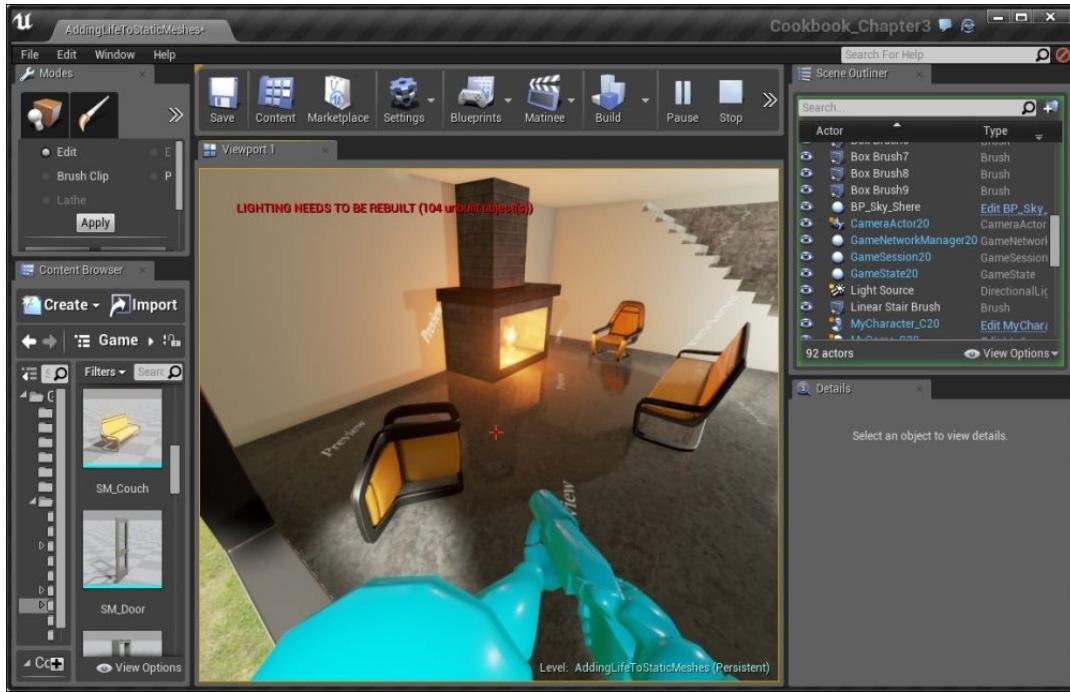
### How to do it...

With that in mind, let's learn how we can breathe some life into our static meshes:

1. Select a static mesh inside your level, such as the `SM_Chair` object we have placed in the fireplace.
2. Once selected, go to the **Details** tab and under **Physics**, check **Simulate Physics**.



3. Once finished, go in and play the game.



At this point, if you run into the chair or shoot at the object, they will react correctly, according to physics.

#### Note

You may also want to make a static mesh movable for something such as moving platforms and the like using **Blueprints**. To enable that functionality, in the **Details** tab, change the **Mobility** property to **Movable**.

In the preceding image, you'll notice that it says, **LIGHTING NEEDS TO BE REBUILT**. This is said anytime the world has changed in such a way that the current way the lighting data has will not show the most up-to-date version of the level. To get a full appreciation of the level and to have a better representation of how our light will be in the game, build your lighting by going to **Build | Build Lighting Only** and wait for it to finish (when the popup on the bottom-right says, "Lighting build completed").

#### Note

For more information on lighting, refer to <https://docs.unrealengine.com/latest/TNT/Engine/Rendering/LightingAndShadows/QuickStart/index.html>.

## Chapter 4. Building the Great Outdoors – Exterior Environments

In this chapter, we'll cover the following recipes:

- Creating a landscape
- Building an exterior level using the Sculpt mode
- Creating rivers with the Flatten tool
- Placing trees and rocks using the Foliage tool
- Streaming levels

## Introduction

We've seen how to build levels using BSP and meshes. This is really great for man-made structures, such as office buildings, houses, and floors, which are all made of things that look similar to each other. However, things outdoors are very chaotic, with changes in elevation and randomness everywhere. To help facilitate that, Unreal has tools such as Landscapes that allow us to create massive worlds that our players can interact in.

## Creating a landscape

One of the first things that we will need to do to work with build terrain is to create a landscape through Unreal Engine's **Landscape** mode, so let's get started!

### Getting ready

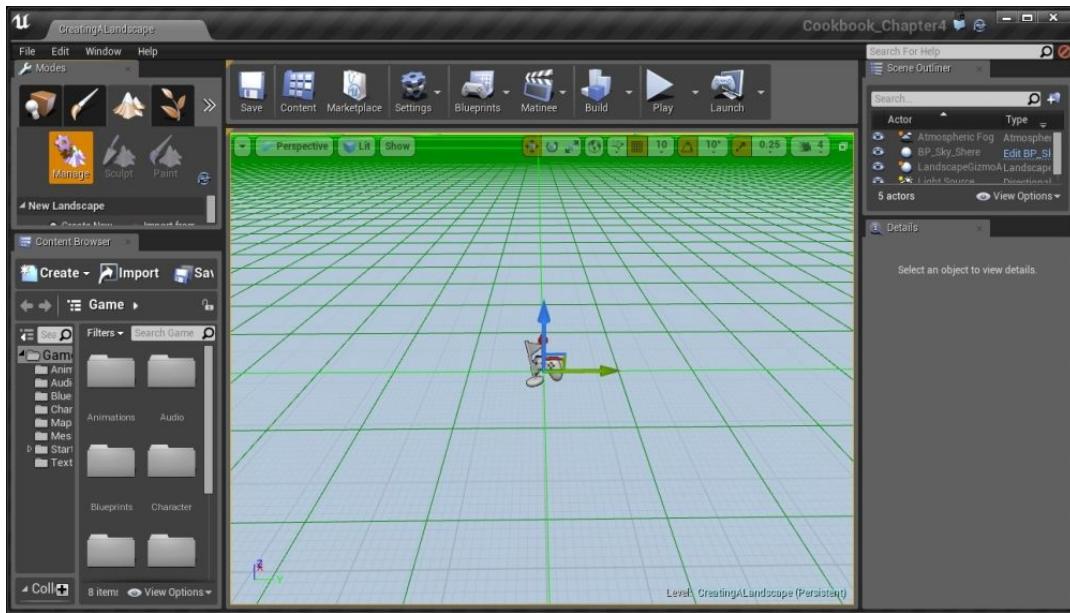
Before we start working within the Unreal Editor, we will need to have a project to work with:

1. First, open up the Unreal Editor by clicking on the **Launch** button from **Unreal Engine Launcher**.
2. Start a new project from the **Project Browser** tab by selecting the **New Project** tab. Select **First person** and make sure that **With Starter Content** is selected. Give the project a Name (**Cookbook\_Chapter4**). Once you are done, click on **Create Project**.

### How to do it...

Now that we have our project set up, let's get started with building a landscape:

1. With the editor up, we will first want to create a new level for us to work in. To do this, go to the top-left of the screen and select **File | New Level...** or press **Ctrl + N**.
2. From here, you'll have the **New Level** window pop up. Click on the **Default** template, and you will have an empty level to work with.
3. Since we are going to create everything from a terrain, we don't need to have the default floor. So, select the **SM\_Template\_Map\_Floor** mesh and delete it by going to **Edit | Delete** or pressing the **Delete** key.
4. After this, we will need to access the **Modes** tab on the top-left of the screen and click on the icon that looks like a mountain to change the game to the **Landscape** mode.



You'll know that we're in the **Landscape** mode when you can see a green grid showing up inside the level.

### Tip

You can also access the **Landscape** mode by pressing *Shift + 3* on your keyboard.

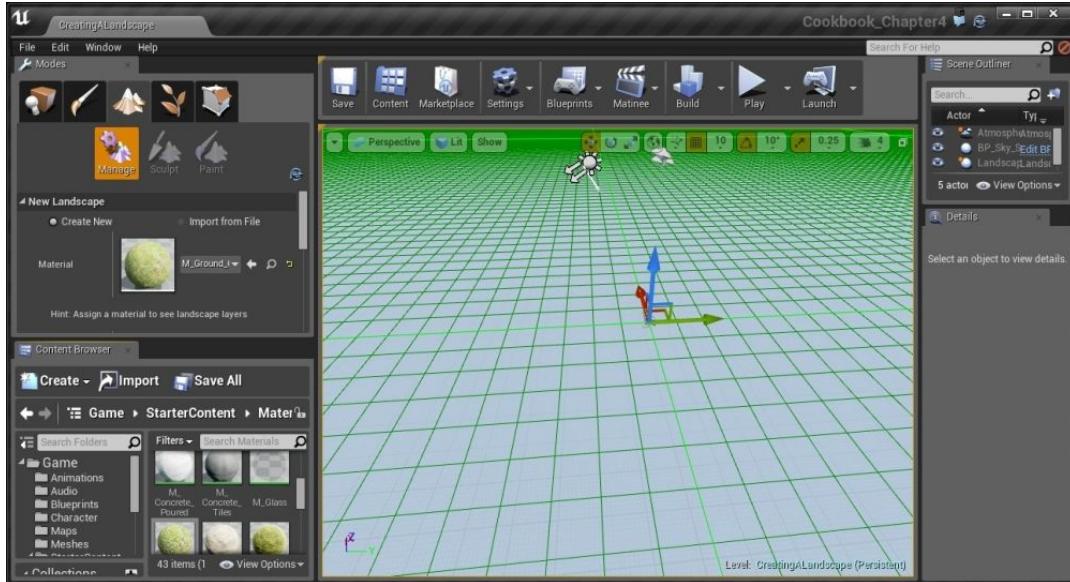
Under the mode image, you'll notice that the properties have changed to show three modes, with two grayed out and the first one, **Manage**, selected. When selected, the **Manage** mode allows us to create new landscapes and modify the properties that we have already created. We will learn about the other modes later in this chapter.

Right now, we'll only see the **New Landscape** section. Now, we don't actually have a landscape created yet, but let's first set up some properties to make this seem more realistic:

1. Go into the **Content Browser** tab and the `StarterContent\Material` folder. Once here, drag and drop the `M_Ground_Grass` material into the **Material** property under the **New Landscape** section. This will make our landscape use this material on all of the terrain, by default. Alternatively, you can also select the material in the **Content Browser** tab and then within **New Landscape**, click the arrow to the right of the **Material** variable and apply the material to the landscape.

### Note

It's possible to have multiple materials in your landscape to help create cool effects. For details on this, refer to <https://docs.unrealengine.com/latest/INT/Engine/Landscape/Editing/PaintMode/index.html>.

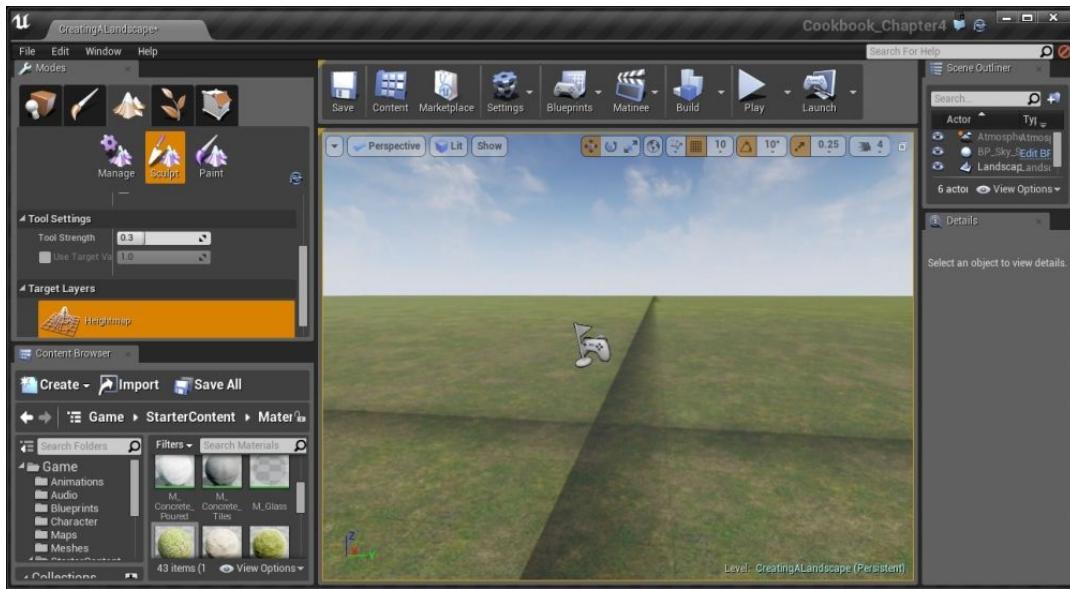


There are a number of properties (that we can change at this point), which will alter the landscape such as **Location**, **Rotation**, and **Scale**. There are also some new properties that will affect the size and quality of the landscape. The default values are balanced for quality and build time, so we're going to use them for the most part.

### Note

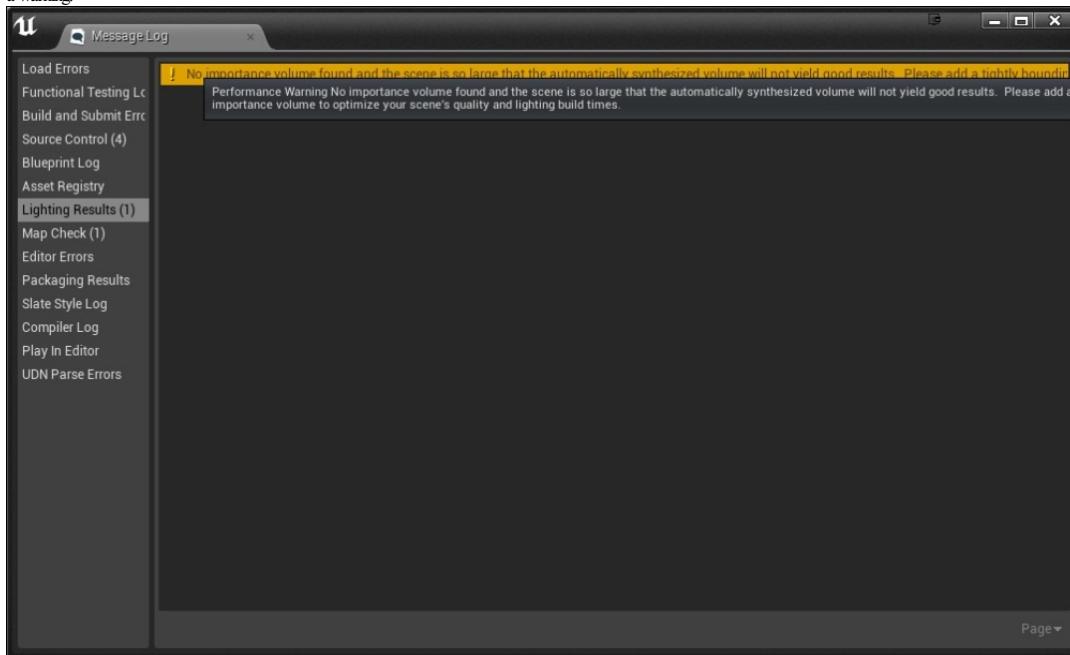
For more information on the New Landscape properties, refer to <https://docs.unrealengine.com/latest/INT/Engine/Landscape/Creation/index.html#creatinganewlandscapeusingthelandscapetool>.

2. Continuing in the **New Landscape** section of the **Modes** tab, change the **Z** property of **Location** to **-10** to place it below our character's starting point (the **Player Start** object that looks similar to a hole in one with a gamepad controller to its next).
3. Finally, at the bottom-right of the **New Landscape** section, click on the **Create** button to place the landscape into the world with our specifications.



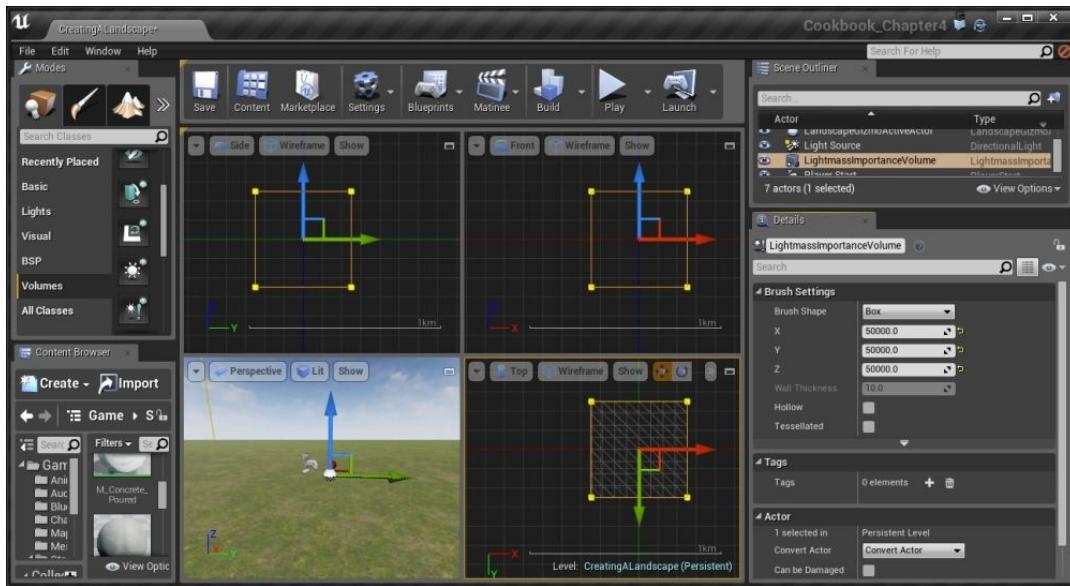
*Creating a landscape*

- When it's first created, you may notice these dark lines in grids everywhere. This is because we haven't built lighting yet, so let's go ahead and do this by selecting the **Build** section's drop-down menu from the top toolbar above the level and selecting **Build Lighting Only** or by pressing **Ctrl + Shift + ;** on your keyboard. Wait for a few seconds, and the lighting should complete, however, there is a popup, which will display a warning:



This is alerting us to the fact that we should add **Lightmass Importance Volume** around the important parts of the level. Being a warning, it is something that we could technically avoid. As game developers, it's important for us to remove warnings wherever possible. To fix this, follow these steps:

- Switch back to the **Place** tab, go under **Volumes**, and scroll down to the **Lightmass Importance Volume** option. Drag and drop it into the center of the world. Once the option is created under the **Details** tab, change the **X**, **Y**, and **Z** properties to **50000**. To check that everything is contained, restore the viewports and maneuver as needed to get the yellow lines to fit in your level.



In my example, I made the volume as big as needed to fit the terrain, but you may make it smaller to fit with just what your level contains.  
6. Build the lighting in the same manner as before, and everything should be set up correctly!

#### Note

For more information on creating landscapes and what all of the properties mean, check out the Unreal documentation at <https://docs.unrealengine.com/latest/INT/Engine/Landscape/Creation/index.html>.

## Building an exterior level using the Sculpt mode

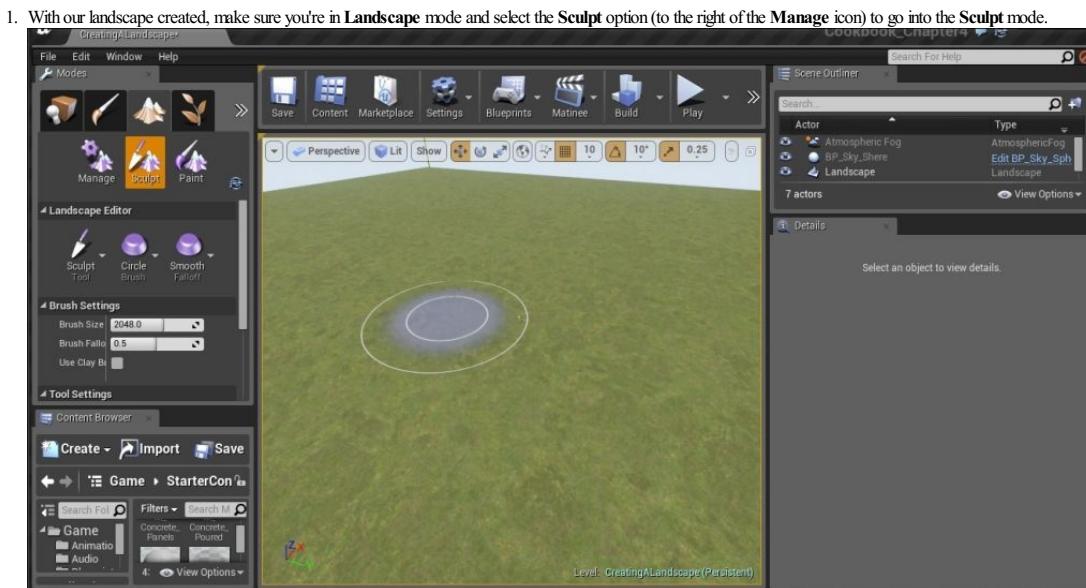
Of course, creating a landscape on its own looks just the same as a flat plane. In this section, we are going to explore the foundation of building a level by making use of the **Sculpt** mode of the **Landscape** tool.

### Getting ready

In order to follow this recipe, you will need to have a landscape created. If you need assistance with this, check out the *Creating a landscape* recipe earlier in this chapter.

### How to do it...

With the knowledge of how to start a workflow, we can now apply that by quickly creating a level:

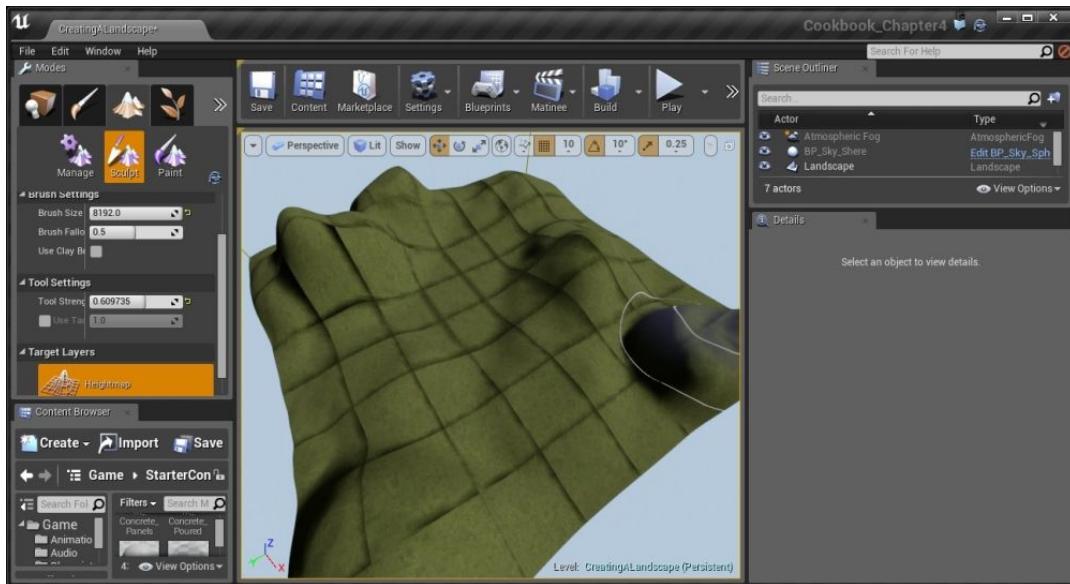


This will bring us to options for modifying our terrain in many different ways. Currently, the **Sculpt** tool is selected and if you move your mouse over the level, you'll notice a light gray circle moving around with the mouse pulsing on and off; this is the brush for our terrain creation and where we will either rise or fall.

- Clicking will raise the terrain into the air. Click around the edges of the world to create some hills that we can work with. Changing the **Brush Size** setting will increase or decrease the size of your brush, and the **Brush Falloff** setting will determine how smooth or rough the brush will be. The **Tool Strength** setting will make things go up quicker. Holding down **Shift** while clicking will lower the terrain instead.

#### Tip

Similar to setting the grid size, when working in the **Landscape** mode, you can use the **/** and **/** keys to increase or decrease the radius of your brush.



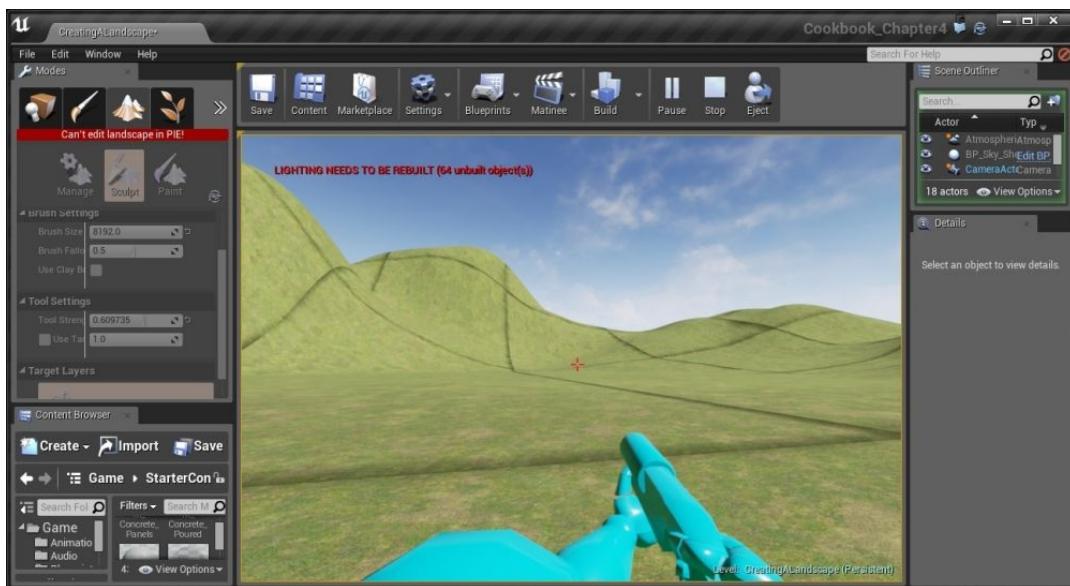
*Adding height variation within our terrain*

Make sure that your **Player Start** object is above your terrain or else you will fall through the world. If this does happen, drag the player to start up on top of the terrain and hit the *End* key for it to fall.

#### Note

When creating hills, it's a good idea to look at the multiple angles so that you can make sure that none are too high or too short. Generally, you want to have taller hills the further back you go or else you cannot see the smaller ones since they're blocked.

To move your camera around, you can hold down the right mouse button and drag it in the direction you want the camera to move around, pressing the *W*, *A*, *S*, and *D* keys to pan. The scroll wheel can be scrolled to zoom in and out from where the camera is.



*Playing the game after placing the Player Start object on the terrain*

With this, we have added some hills to the world!

## Creating rivers with the Flatten tool

Now that we have some hills going on, let's add some holes, making use of an additional tool, the **Flatten** tool.

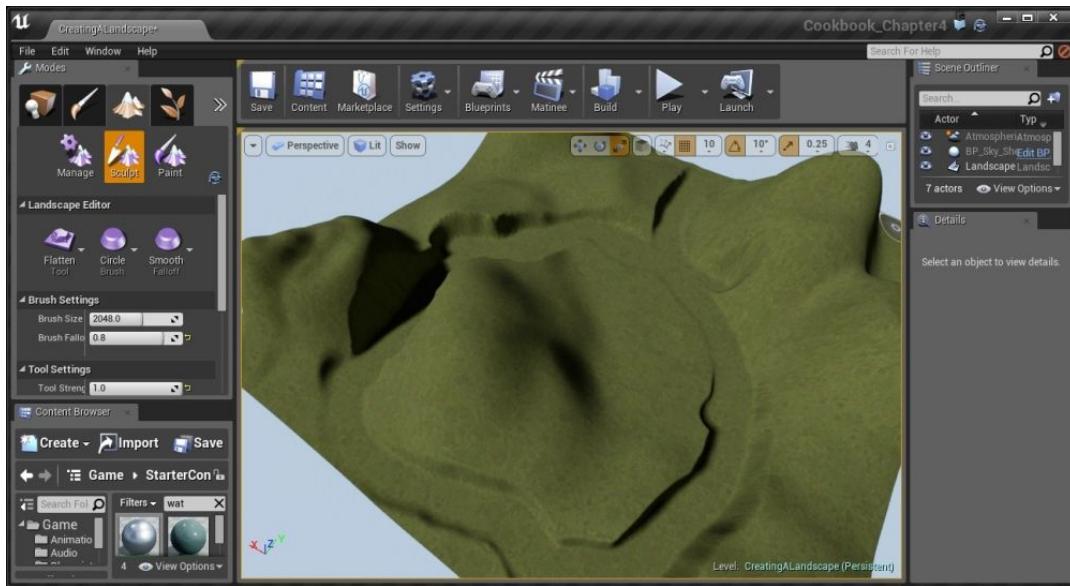
### Getting ready

You will need to have a landscape created for this recipe. If you need assistance with this, check out the *Creating a landscape* recipe earlier in this chapter.

### How to do it...

With the knowledge of how to start a workflow, we can now apply that by quickly creating a level!

1. In the **Landscape Mode** tab, click on the **Sculpt** tool tab and select the **Flatten** tool.
2. Check the **Flatten Target** selection and put in  $-100$ . By default, the **Flatten** tool will flatten to the middle of the map, but this allows us to pick a new position for it to go to.
3. Next, we want to increase **Tool Strength** to  $.8$  to make it easier to dig out the river by clicking and dragging around wherever we want water to be placed.



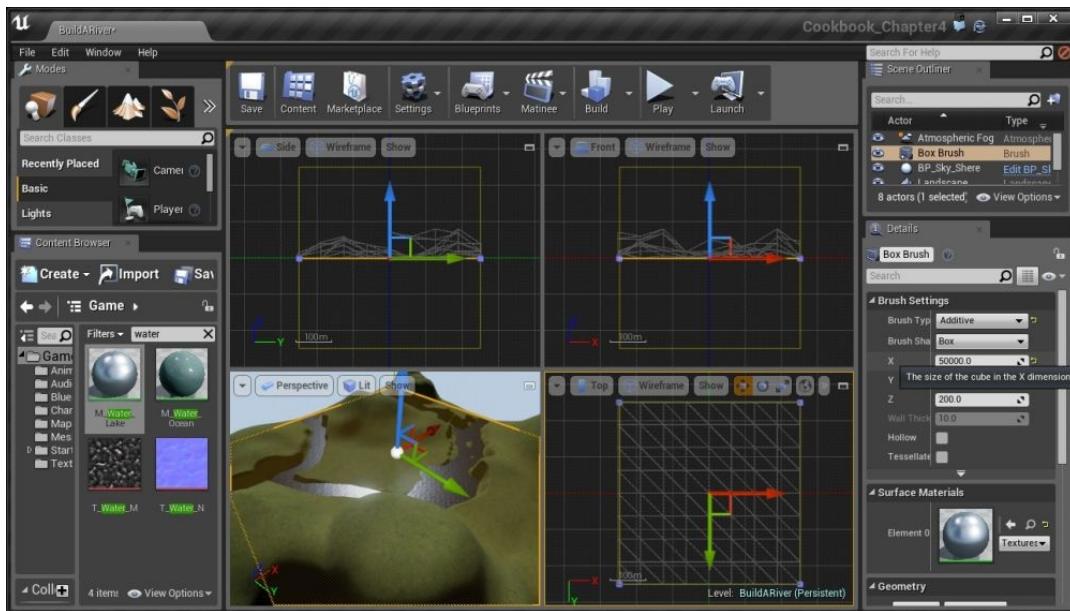
## Note

There are a lot of other sculpting tools that can be used for other own purposes, but we don't have enough room to cover them all. For more information on all of the **Sculpt** mode tools, refer to <https://docs.unrealengine.com/latest/INT/Engine/Landscape/Editing/SculptMode/index.html>.

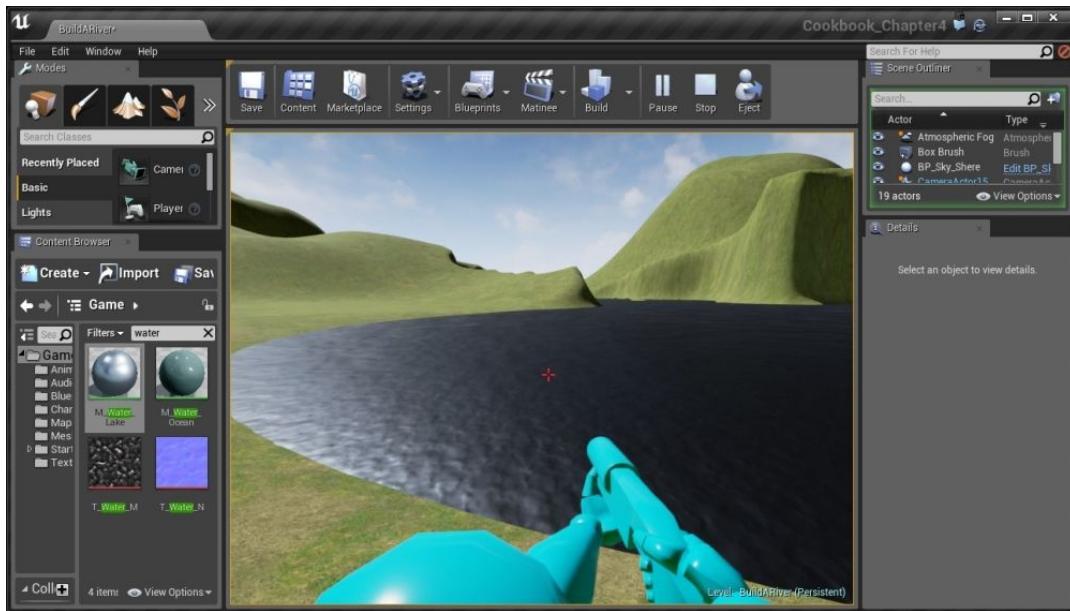
- Now that we have the ground for the river to be placed, we need to get our water placed in the world. In the **Modes** tab, select the **Place** mode and drag and drop a **BSP Box** object into the world. Under the **X** and **Y** axis, set the size to **50000**, just like our terrain, and shift it so that it is centered in the world, being above the water but below our normal terrain.
- Then, go to the **Materials** folder and drag and drop the **M\_Water\_Lake** material to the **Details** tab under the **Element 0** property of **Surface Materials**.

## Note

One thing to note is that if you have a material selected before you create a BSP box, the material will automatically be applied to it once it is brought into the world.

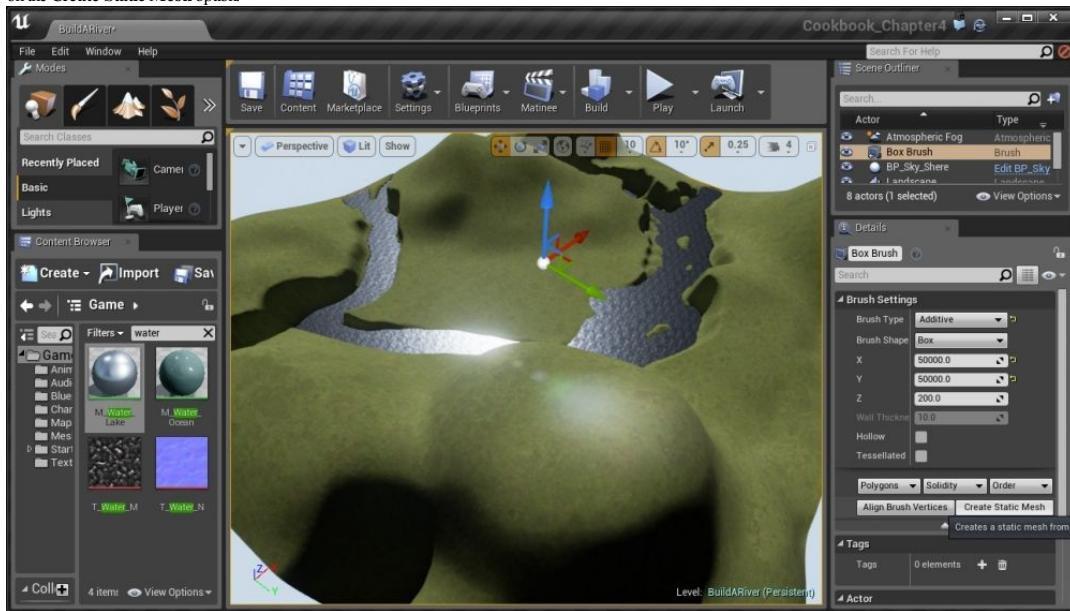


- Play the game and move over the water.



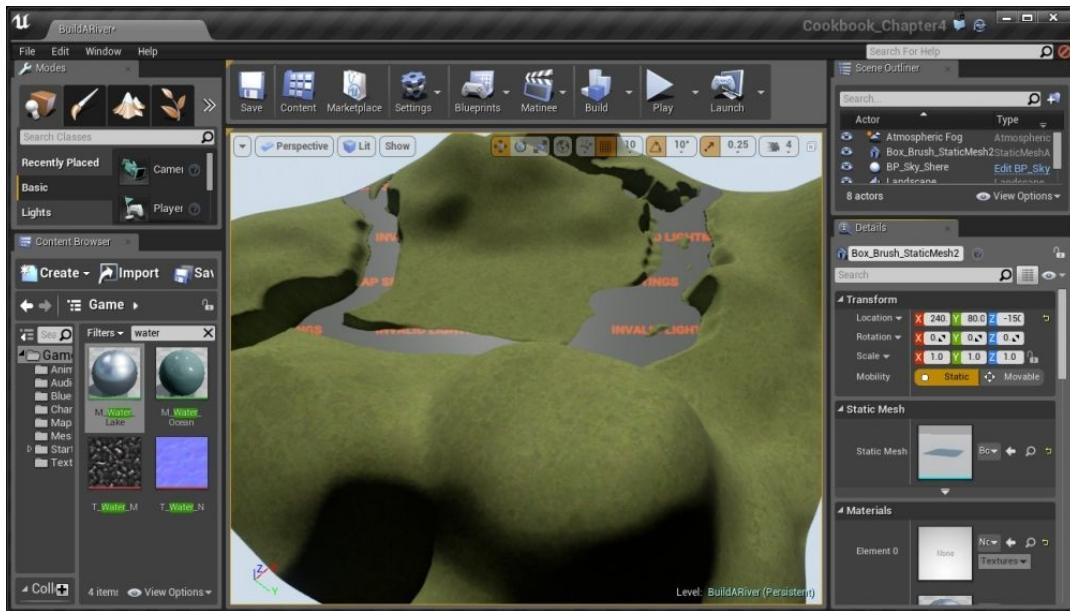
*Playing the game with water in the world*

7. This is looking really nice, except that we can currently walk on water. That's because brushes always have collision. But, we can fix this by converting the BSP into a static mesh.
8. Select the **Box Brush** object from the **Scene Outliner** tab and under the **Details** panel, go to **Brush Settings** and click on the little arrow at the bottom of the part to show the extended options. Once here, click on the **Create Static Mesh** option.



*Option to convert object into a static mesh*

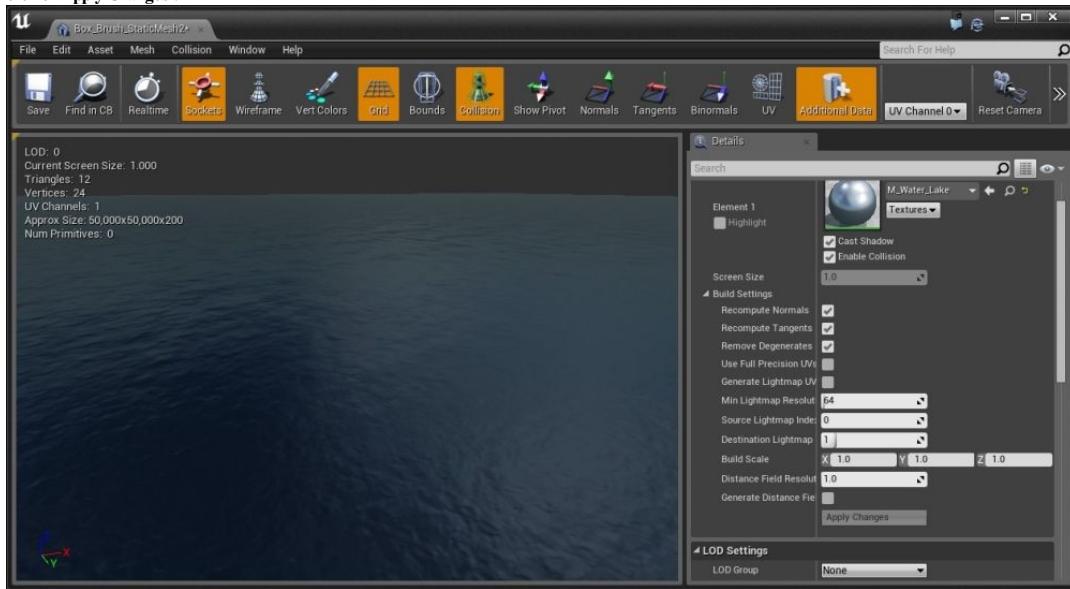
9. It will ask where you want to save it. Click on **Create Static Mesh** once again and it will finally convert the object into a static mesh.



The river object is now converted into a static mesh

But you will see that on the geometry, it says **Invalid Lighting Settings**, as seen in the preceding image. To fix this, go to the mesh's location in the **Content Browser** tab and double-click on it to open up the **Static Mesh Editor**.

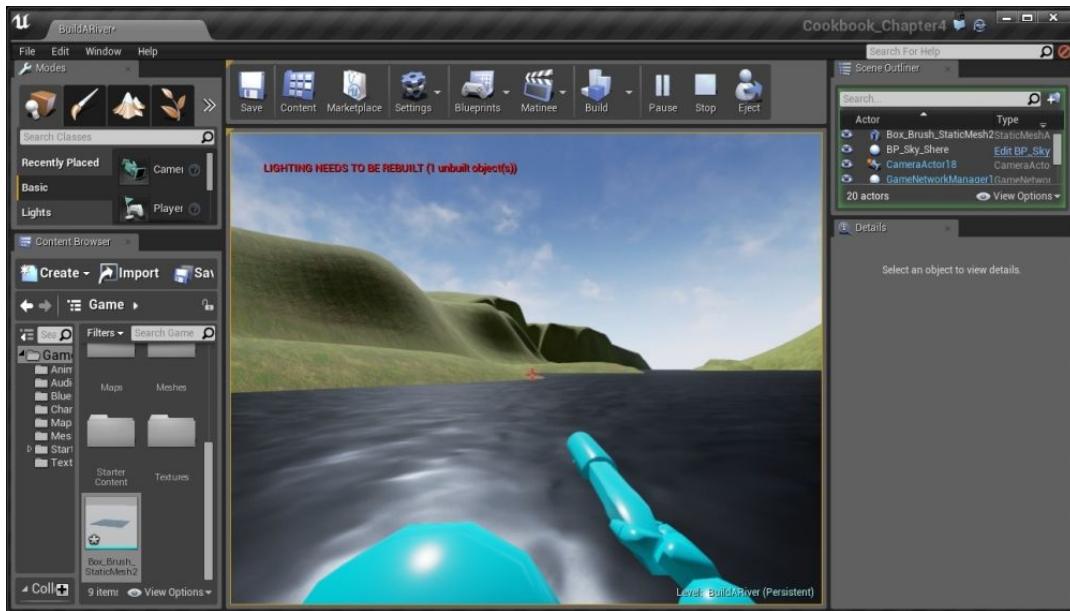
- Inside the **Static Mesh Editor**, in the **Details** tab, go to the **LOD Settings** section and expand **Build Settings** if it isn't expanded already. From here, uncheck the **Generate Lightmap UVs** option and then click on **Apply Changes**:



- Finally, exit the **Static Mesh Editor** and go back into the Unreal Editor and select the mesh once more. From the **Details** tab, change **Collision** to **NoCollision**.

#### Note

Alternatively, in the **Static Mesh Editor** for the mesh, you can also go to **Collision | Remove Collision**, but this will make it so that all the objects with that mesh will not have collision, whereas the **Details** tab is only for that specific object in your level.



*Playing the game with the removed collision allows the player go down into the water*

Now you can go down into the water with no issues! It's looking pretty awesome!

#### Note

Another cool way of creating rivers is making use of the Landscape Spline tool. You can learn more about it at <https://docs.unrealengine.com/latest/INT/Engine/Landscape/Editing/Splines/index.html>

## Placing trees and rocks using the Foliage tool

Now that we have a world basis for our level, we want it to be more than just land and a river.

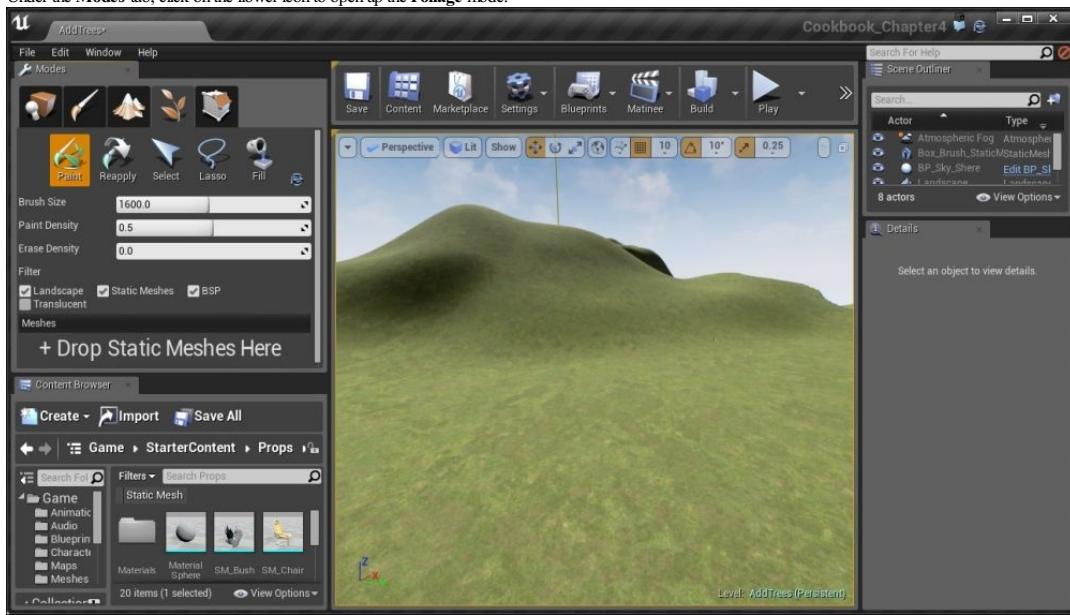
### Getting ready

This recipe assumes that you have a project open with the sample assets and landscape included. If you do not have that yet, feel free to follow the instructions in the *Creating a landscape* recipe.

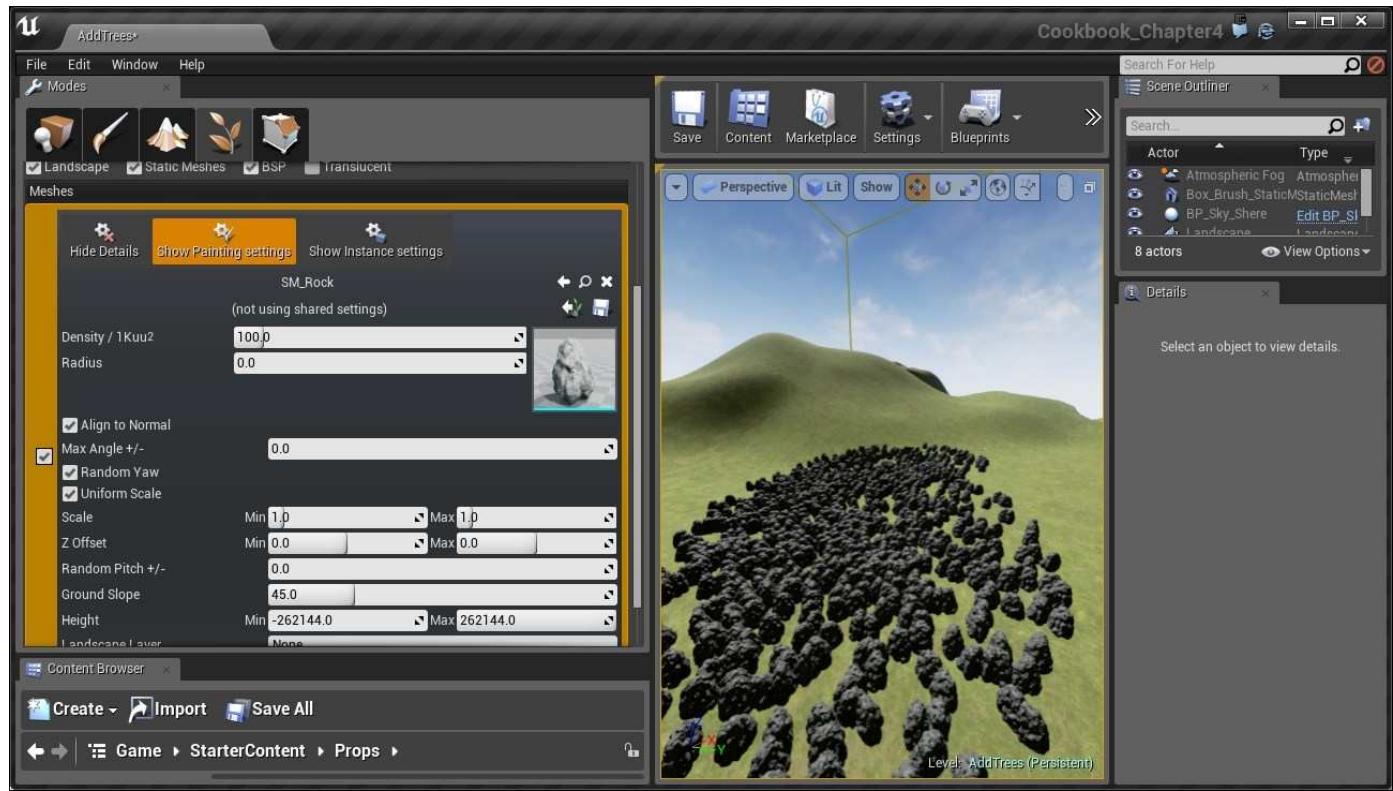
### How to do it...

Now that we have our room created, let's add some materials to it.

- Under the Modes tab, click on the flower icon to open up the Foliage mode:



- This should look fairly similar to the Landscape tool. But this time, there is a **Meshes** section at the bottom with the text **Drop Static Meshes Here**. With this in mind, go into the **StarterContent/Props** folder and drag a **SM\_Rock** object into there.
- At this point, there is a variety of options that can be used to modify how the rocks will be placed as well as how they will be drawn. First of all, if you draw on the level, you will see far more rocks than you need to have drawn.



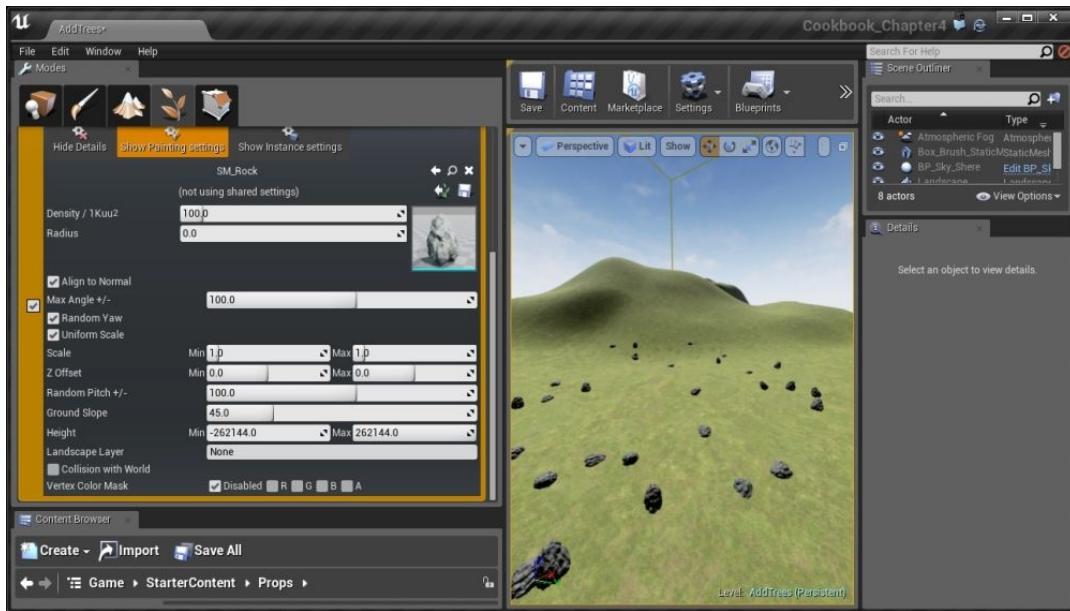
*Placing rocks all over the level*

This is nice for something like grass, but not at all for rocks.

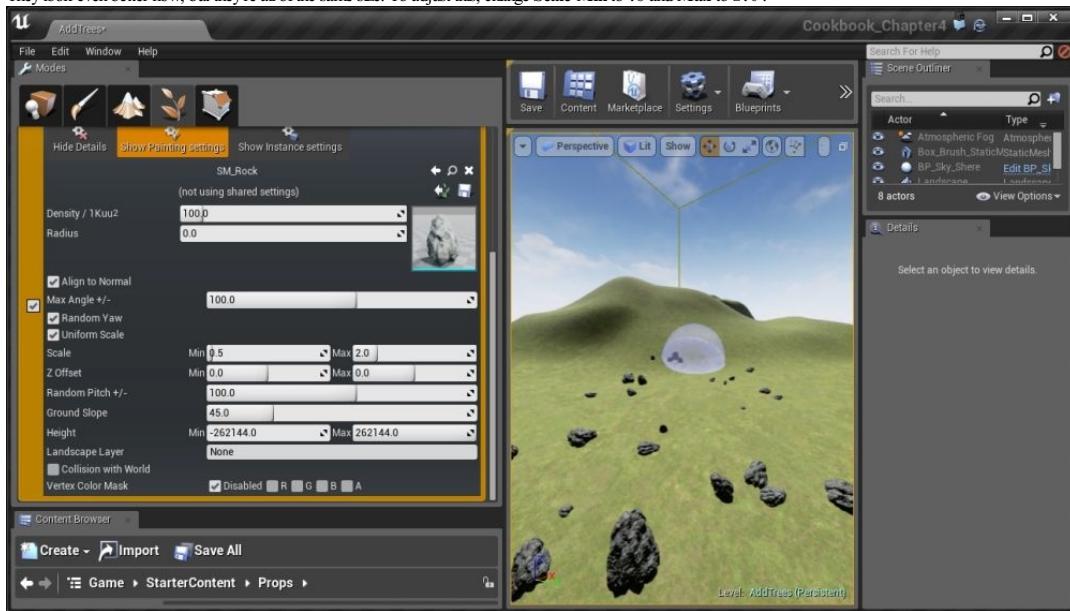
4. Press **Ctrl + Z** to undo what we did before. To lessen the amount, modify the **Paint Density** property up at the top of the **Foliage Mode Paint** section to a much lower number such as **.006**.



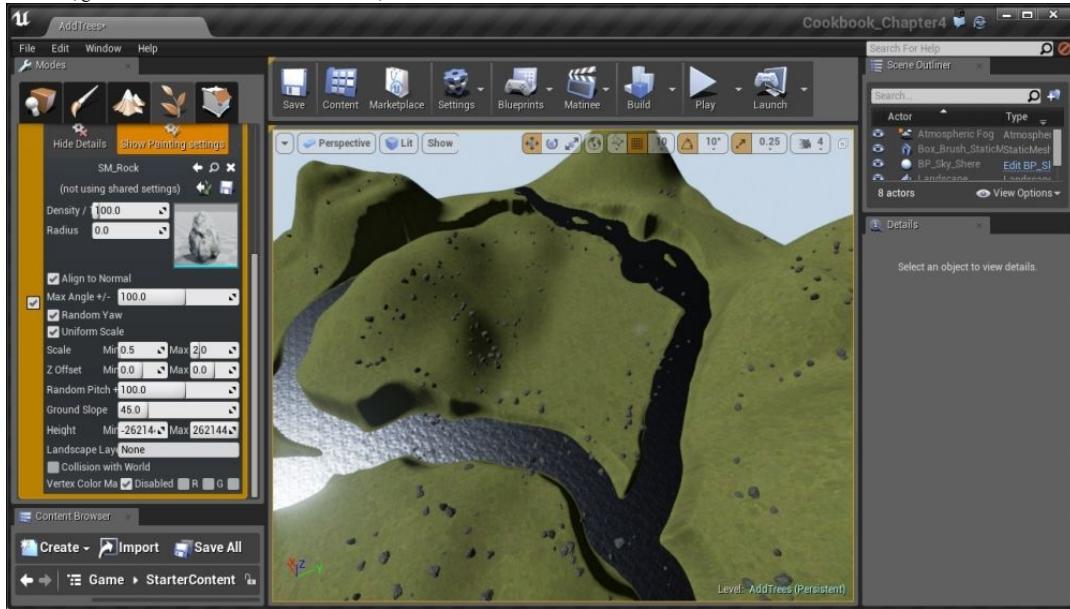
5. Now they're showing up at a nice rate, but they all look very similar. This is fine for something like trees, but rocks are much more random. To fix this, under the **Meshes** tab, make sure that **Show Painting settings** is up and change **Max Angle +/-** and **Random Pitch +/-** to **100**.



6. They look even better now, but they're all of the same size. To adjust this, change **Scale Min** to **.5** and **Max** to **2.0**.



7. Finally, by default, foliage has no collision, but rocks need to collide so that the players can collide when they touch the rocks. To fix this, go to the top of the **Meshes** section and select **Show Instance settings**. Afterward, go to **Collision** and under **Collision Presets**, select **BlockAll**.



It's nice! At this point, we now have painted rocks all over the level.

### Note

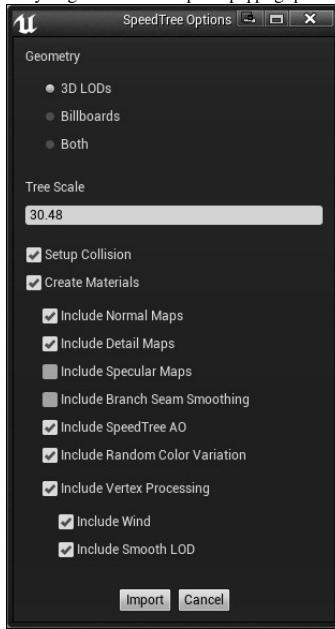
For some users, the **Show Instance settings** option may be hidden due to the **Modes** tab not being wide enough, so expand it if that is the case.

8. Of course, this is nice to work with now, but in the real world, we have other things, such as trees, which will be a lot more up above the ground. But, by default, Unreal does not give us tree meshes to work with. However, SpeedTree has some sample trees that are free to download at <http://store.speedtree.com/unreal-engine-4/>.

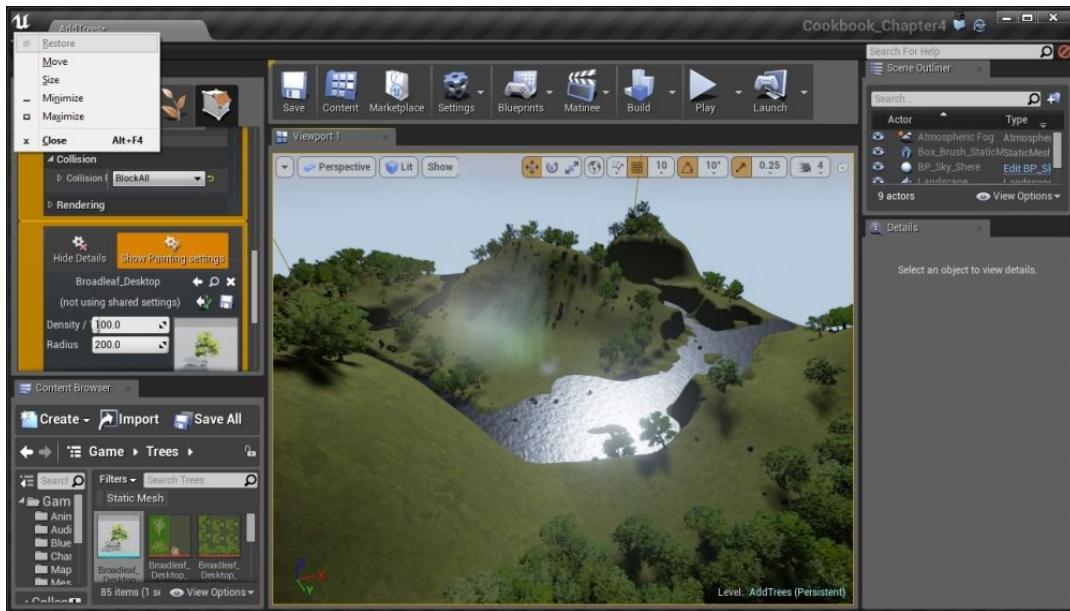
The screenshot shows a web browser window displaying the SpeedTree store. The URL in the address bar is [store.speedtree.com/unreal-engine-4/](http://store.speedtree.com/unreal-engine-4/). The page title is "SpeedTree for UE4 Subscription". A search bar and a "Search" button are visible on the left. On the right, there are links for "Home", "Cart", and "My Account". Below the title, a sub-headline reads: "New with Unreal Engine 4® Subscription, users get the AAA vegetation toolset that's been the game industry's premier solution since 2002!". A large image shows a scene from a game with several SpeedTree-generated trees in front of a building. Below this image, text says: "Our beautiful trees and plants, which come fully-enabled with lightmapping, dynamic LOD and wind effects, are available in three options:". There are three buttons: "Buy Trees From Our Store", "Subscribe to the SpeedTree for UE4® Modeler", and "Download Free Sample Trees". To the left of the main content, there is a sidebar titled "Product Categories" with a hierarchical menu:

- > Unreal Engine 4
  - > Applications
  - > Trees on Sale!
  - > Tree Packages
  - > Trees
- > Unity 5
  - > Applications
  - > Trees on Sale!
  - > Tree Packages
  - > Trees
- > Visual Effects
  - > Applications
  - > Full Versions
  - > Evaluation Versions
- > Trees v7
  - > Broadleaves
  - > Conifers
  - > Palms & Cacti
  - > Shrubs & Flowers
  - > Marine
  - > Miscellaneous & Fantasy
- > Trees v6
  - > Broadleaves
  - > Conifers
  - > Palms & Cacti
  - > Shrubs & Flowers
  - > Marine
  - > Miscellaneous & Fantasy
  - > Hand Drawn

9. At the bottom-left of the website, there is a section that says you can download it directly; do that. Once it finishes downloading, go to the ZIP file and unzip it. Next, rename the folder to **Trees**.
10. We will be moving a large amount of stuff into our game, so first save your project and then drag and drop the **Trees** folder into the **Game** folder from the **Content Browser** tab, and it will start to import the content into your game. You'll see options popping up from time to time; hit **Import** each time to complete the process.



11. Now that everything is imported, you can save it by clicking on the **Save All** button and selecting **Save Selected**.
12. Next, drag and drop the **Broadleaf\_Desktop** static mesh into the **Foliage** tool. Go into **Show Instance settings** and add collision to the object as we described earlier.
13. Afterward, we want to add some distance between each tree, so switch back to **Show Painting settings** and change the **Radius** property to 200. After this, start painting once again around the level where you want trees to be.



*Adding trees to the map*

- Finally, build all this again and start up the game. You may notice that the build time for lighting is probably going to take a lot longer than it has previously.



*Final view of the terrain with trees and rocks*

With this, we now have a much nicer looking level!

## Streaming levels

As levels get more and more complex, they can sometimes get quite unwieldy. One of the tools that we have as designers is the ability to break our levels apart and load them in at runtime. This is also incredibly useful for working in teams, where one person can work on one part of the game and bring it all together later.

We will look at the easiest way to load in a level instantly, which is to have the level always being loaded.

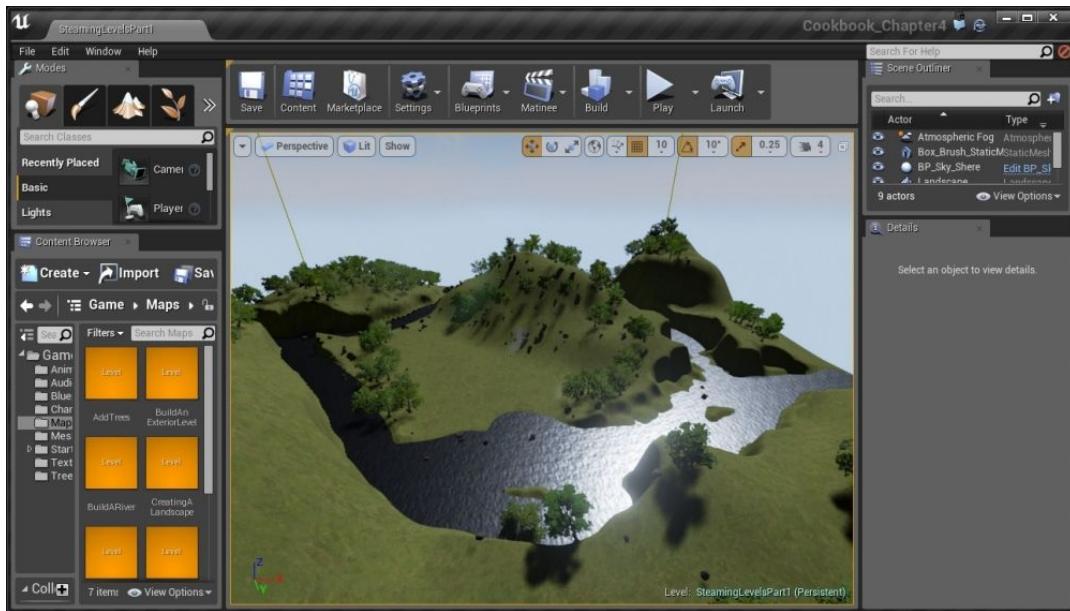
### Getting ready

This recipe assumes you have two levels you'd like to have loaded at the same time. I will be using the levels created in the *Placing trees and rocks using the Foliage tool* recipe in this chapter and the *Meshing an example map* recipe from [Chapter 3](#), *Creating Quality Interior Environments*.

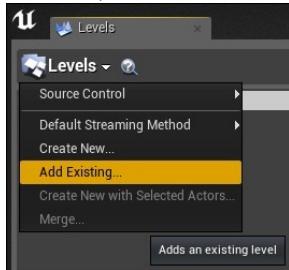
### How to do it...

Let's first export one of these rooms so that we can give them to our artist to work with:

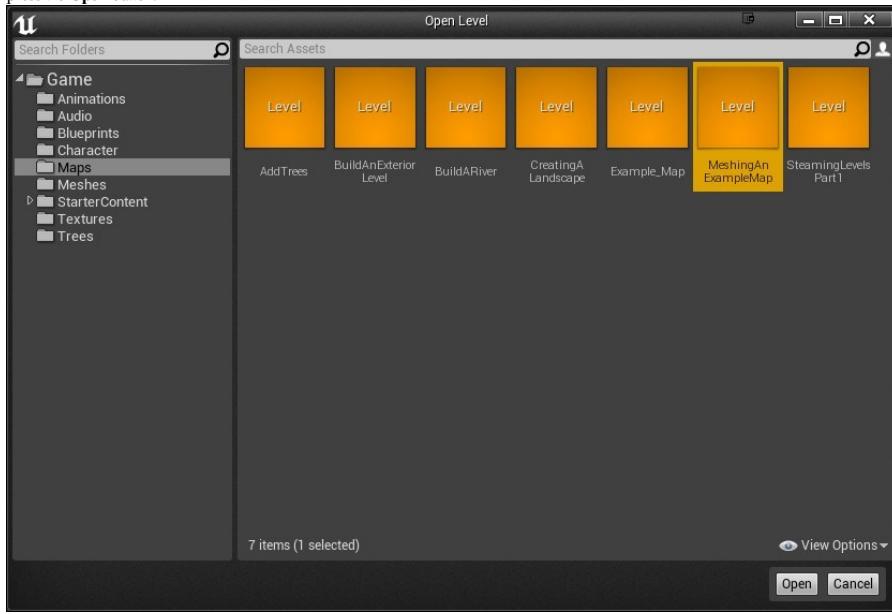
- Open up the level that will be considered your base level. In our case, I will be using the map that contains our completed landscape.



2. Under **Window**, select **Levels** to open up the **Levels** window.
3. From **Levels**, click and select **Add Existing...**



4. From here, you'll be brought to the **Open Level** dialog box. From there, select the mpx you want to be brought into the world. I selected the **MeshingAn ExampleMap** level. After you have the level selected, press the **Open** button.



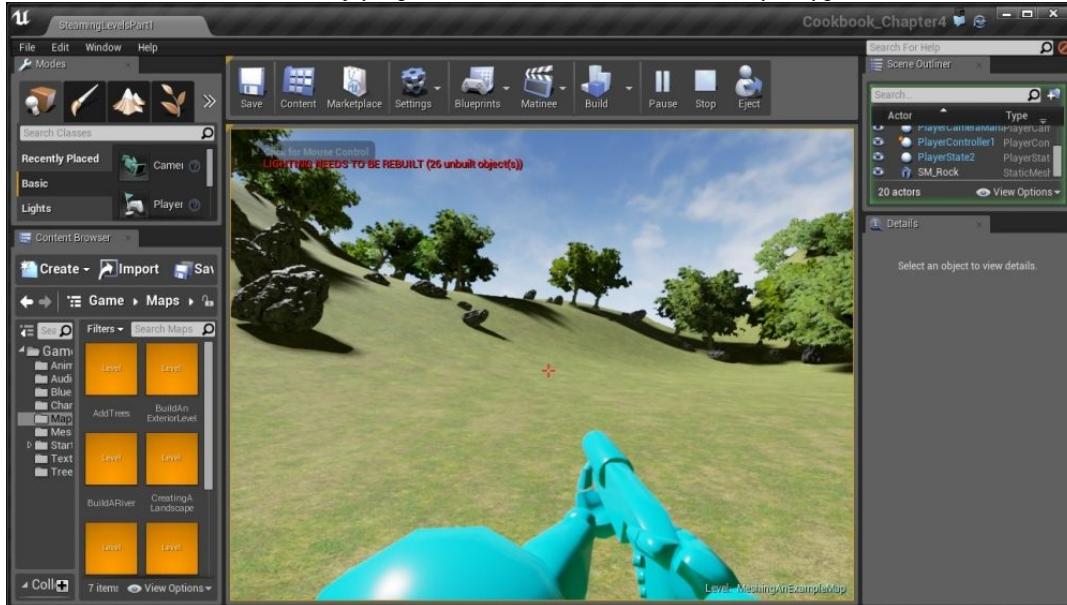
5. Now if you planned ahead, your house and terrain will fit together nicely.



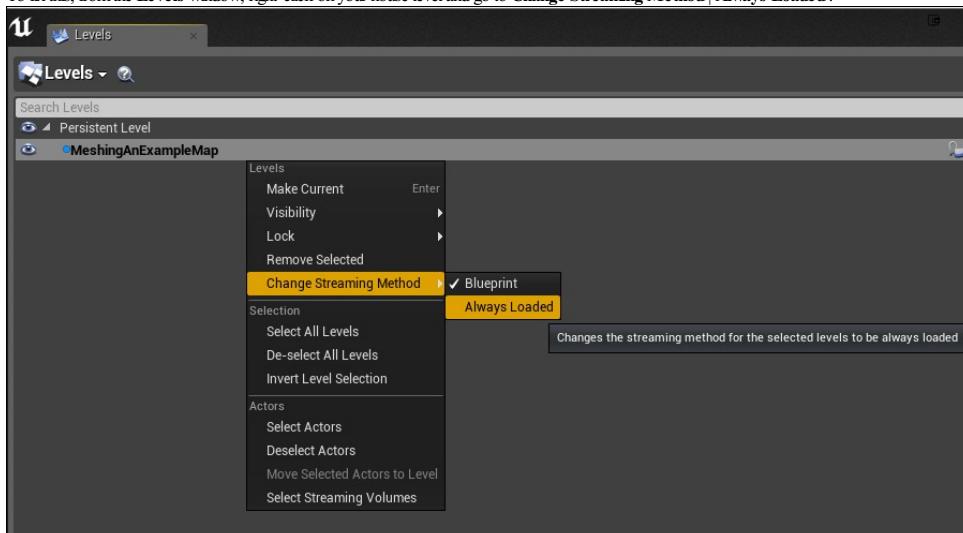
*Final view of the house and the terrain*

If this is not the case, go to the **Levels** window and right-click on one of the levels and select **Select Actors**. Then, move the level to fit where you want it to be placed. Of course, this assumes that you have some flat terrain somewhere. If not, use the **Flatten** tool on the **Foliage** tab to flatten the area to make it easier for people to see.

6. Of course, it seems like we are done now, but if we play the game, we won't be able to find the house. There's actually a really good reason for this; the level isn't actually loaded.



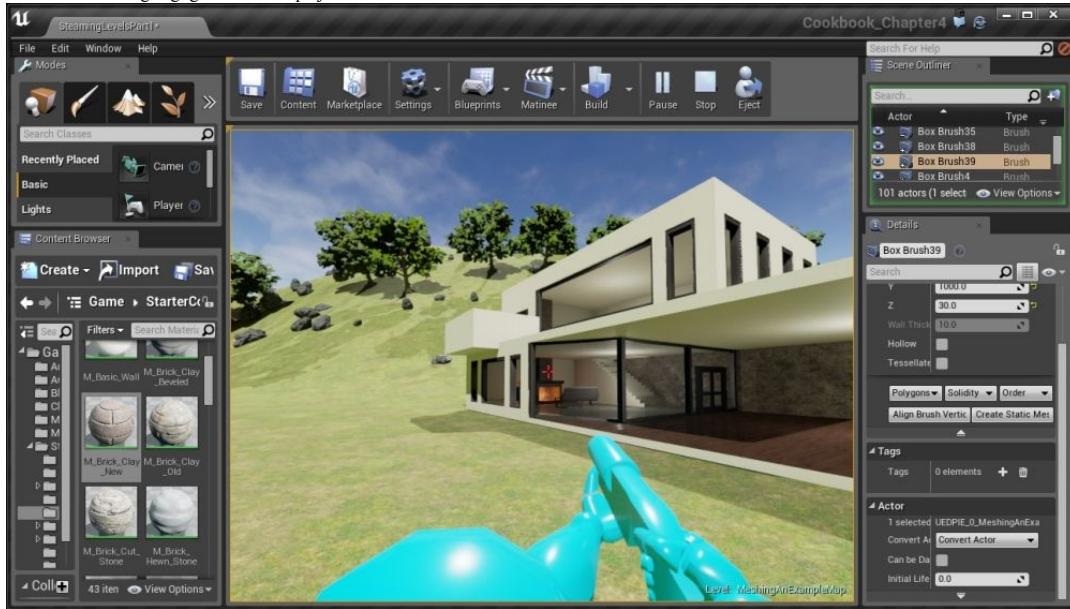
7. To fix this, from the **Levels** window, right-click on your house level and go to **Change Streaming Method | Always Loaded**.



### Note

If for some reason the method is grayed out, check to make sure your level isn't locked. If it is, click on the lock on the right-hand side to enable changes on the level. This is also a good tool to use if you want to make sure that you're only modifying one level at a time.

8. Now let's build our lighting again and run the project!



Now, we can see our levels loaded together!

### Note

For those interested in having levels load dynamically over time rather than all at once, you can use blueprints to have levels stream in and out to create what seems like a seamless world. To see how to do this, refer to [https://wiki.unrealengine.com/Blueprint\\_Manual\\_Level\\_Streaming](https://wiki.unrealengine.com/Blueprint_Manual_Level_Streaming).

## Chapter 5. Lights, Camera, Action – Cinematics

In this chapter, we'll cover the following recipes:

- An introduction to Matinee
- Creating an opening cutscene
- Playing a Matinee via Blueprints
- Preventing a player from moving via cinematic mode

### Introduction

Now that we have an understanding of creating environments, let's take some time to dive a little deeper into creating cinematics that we can use to give our titles even more depth.

## An introduction to Matinee

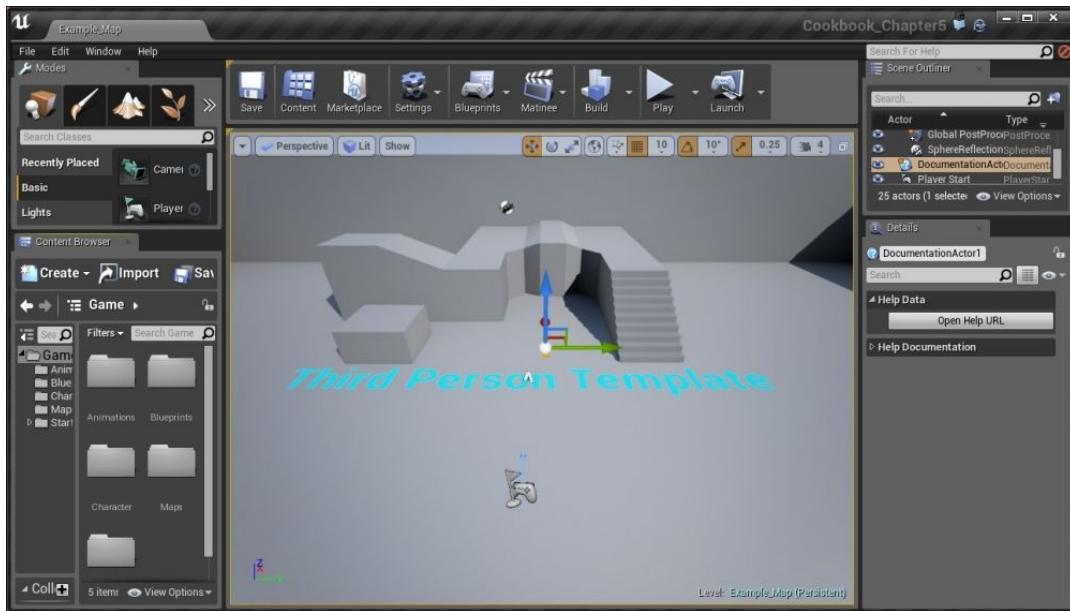
The Matinee tool is the driving force behind all the cinematic scenes within the Unreal Engine. It gives users the ability to be the director within your game, giving you control over the camera, the actor's movement, sound, as well as different cuts and animation effects. Think of any moment in an Unreal game where you didn't have direct control over the character, like in a cutscene. Chances are that was done in Matinee. However, Matinee can be used for many other things, which we will discuss at the end of the chapter.

In order to create a Matinee, we will use the (aptly named) **Matinee Editor** that can be accessed from the toolbar at the top of the screen.

### Getting ready

Before we start working within the Unreal Editor, we will need to have a project to work with:

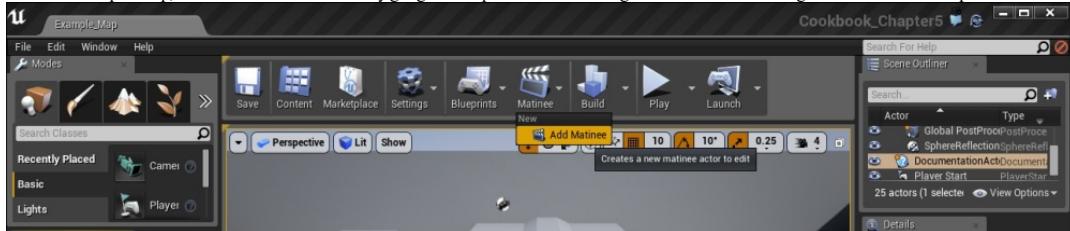
1. First, open up the Unreal Editor by clicking on the **Launch** button from the Unreal Engine Launcher.
2. Start a new project from the **Project Browser** tab by selecting the **New Project** tab. Select **Third person** and make sure that **With Starter Content** is selected and give the project a Name (**Cookbook\_Chapter5**). Once you are finished, click on **Create Project**.
3. You should see a level similar to this:



## How to do it...

Having defined what Matinee is used for, let's begin using it.

- With our level opened up, let's create our first **Matinee** by going to the top toolbar and selecting **Matinee** and then clicking on the **Add Matinee** option.



- From here, you'll be brought to the **Matinee Editor** window.

### Note

You can also create a Matinee by going to the **Modes** tab and then going to the **Place** mode (*Ctrl + I*) on the far left if you aren't there already and then clicking on **All Classes** and finding the **Matinee** selection from there. Afterward, drag and drop it into your level and once it's there, click on the **Open Matinee** button.



Now, there are a lot of things up here currently and it may be confusing to look at them first, so let's briefly go over these windows:

- Toolbar**: The item at the very top is the toolbar; it contains various tools that we can use to check on our animations.
- Curve Editor**: Below the toolbar, we have Curve Editor, which is used to have fine control over properties that change over time (properties that make use of Distributions). Mainly, this'll be the key frames that you create. In order to see something on Curve Editor, you'll need to click on the gray box toggle button from the **Tracks** tab.
- Details**: On the right-hand side, we have the Details tab that will tell you the details about whatever key we have selected, such as whether it is active or how the key should move.
- Tracks**: Here is where we will manipulate objects and where the bulk of your work will be done. For those who have worked with video editing or animation software, this will be very familiar as a timeline of the changes we want to take place over time.

## Creating an opening cutscene

Now that we have an understanding of what Matinee is, let's start building our first cutscene!

## Getting ready

Before we start working, you should have a level opened and entered the Matinee Editor. For assistance with this, check out the *An introduction to Matinee* recipe.

## How to do it...

In order to create a cutscene, you need to perform the following steps:

1. Matinee animates movable objects over time, but it needs to know what objects to animate and what properties of them to change. Under the **Tracks** tab on the left-hand side, right click and select **Add New Camera Group**. Once there, it will ask you for the name of your group; we will call it **Camera01**.

When doing this, a camera is automatically created in the scene at the current position of the user camera within the editor.

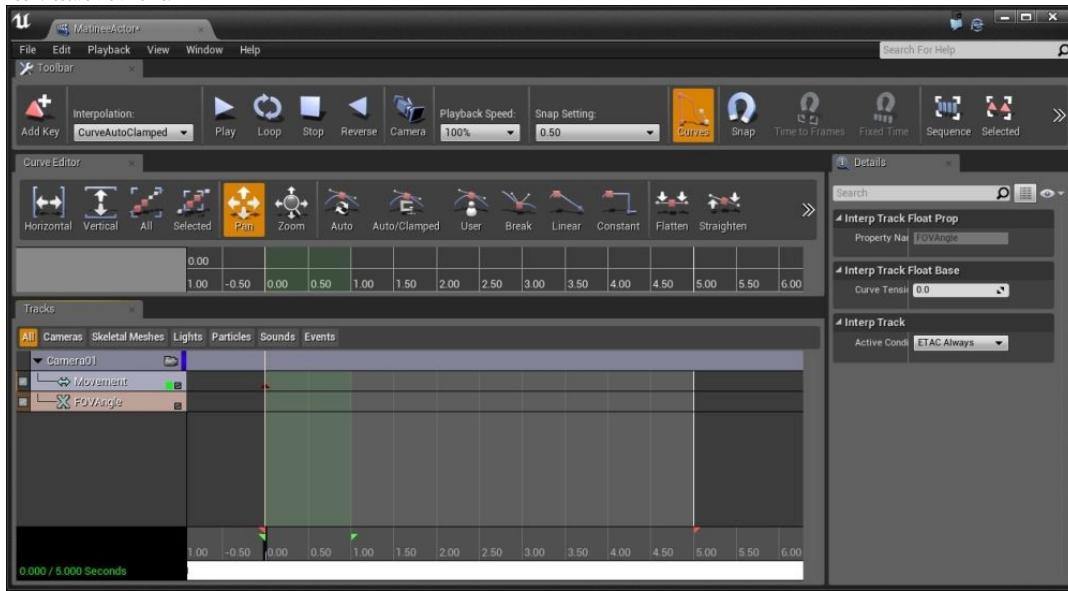
### Note

You can also create a camera actor in the scene by right-clicking and navigating to **Place Actor | Camera Actor**. Once placed in the scene, with the camera selected, you can create a Camera group using that camera within Matinee.



Camera group added to the Tracks tab

At the bottom of the track, you'll notice a series of numbers separated by lines. This represents the time that has passed from the beginning of the animation. Use the mouse wheel, scrolling in and out, to zoom/rescale the timeline.



Rescaling the timeline

The bright red triangles at the bottom symbolize the size of the entire animation and you can click and drag them to modify its length. The green ones are used to determine which part will you see when you play in the editor.



*Dark red triangle showing key frame*

Next, you may notice the dark red triangle at the 0.00 time marker under the **Movement** row in the tracks. This is known as a key frame, which is a point in time at which we want our Actor to be at a certain value. The first one is automatically created for us, and basically stands for the starting point.

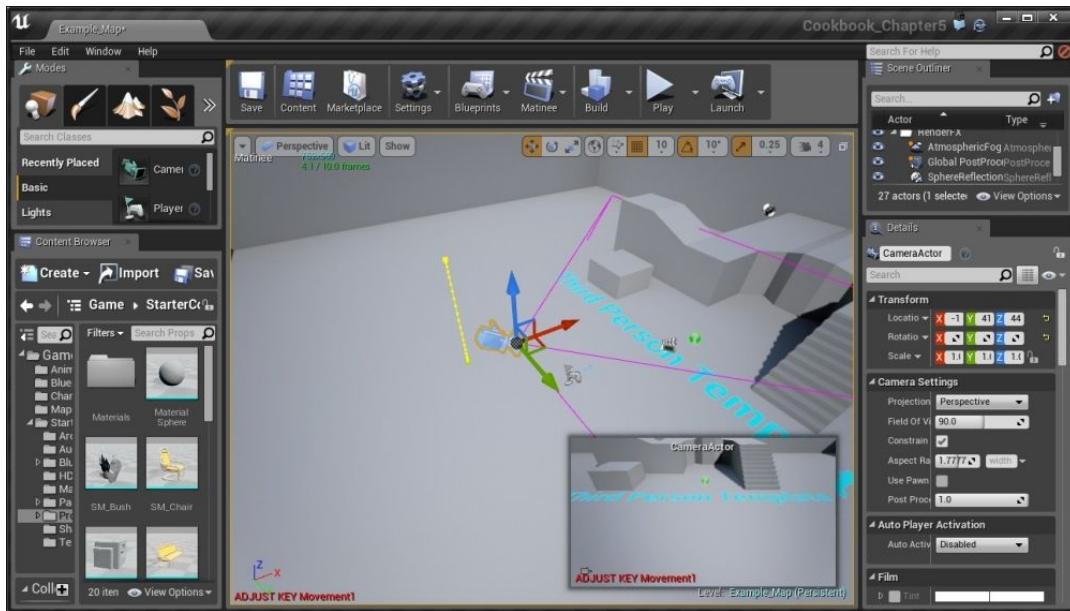
- Now drag the time bar to the 2.00 second mark so that we can add a key there. Then, click on the **Movement** part of our **Camera01** group and then press the **Add Key** button on the top-left of the **Toolbar** tab to add a key.



### Tip

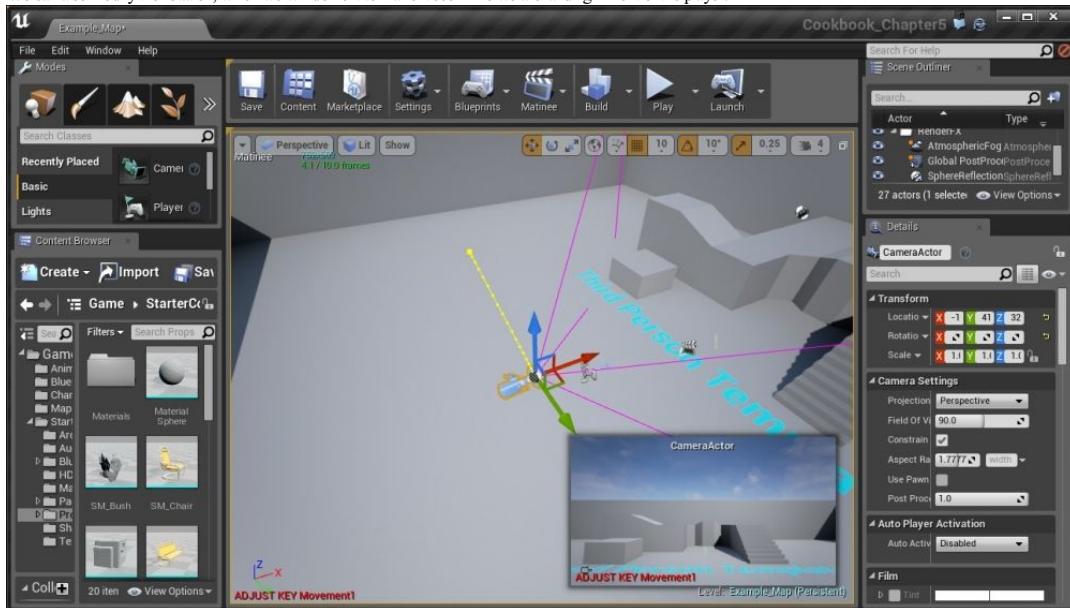
You can also press *Enter* to create a key.

- With the key selected (it is outlined), minimize the Matinee Editor and move back to the main editor.
- You will notice that you have **CameraActor** selected in the **Scene Outliner** tab. Either way, double-click on it inside the **Scene Outliner** tab to zoom to its position. At this point, we can move our camera to modify its position. I will move the actor down in the Z axis, and you should notice a line moving to show the transition between the two.



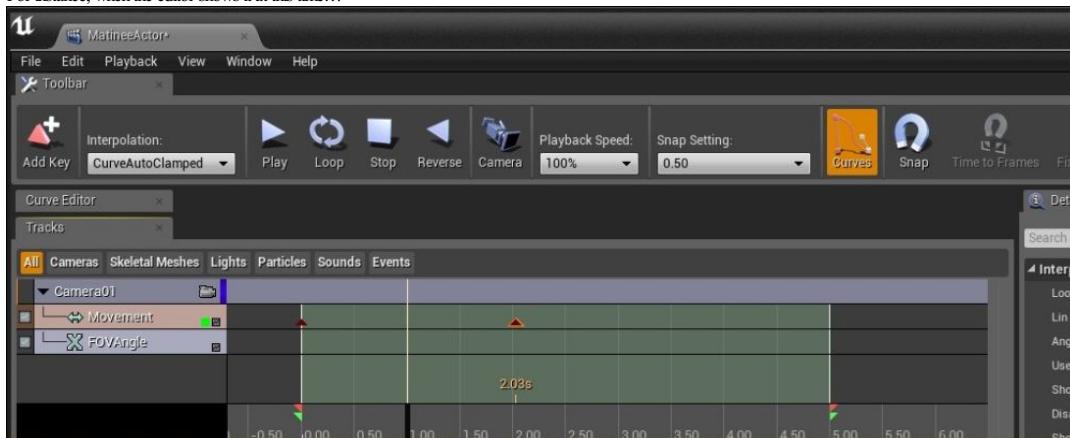
*Modifying position of the CameraActor object*

5. We can also modify the rotation, which we will do next to make it seem like we are landing in front of the player.

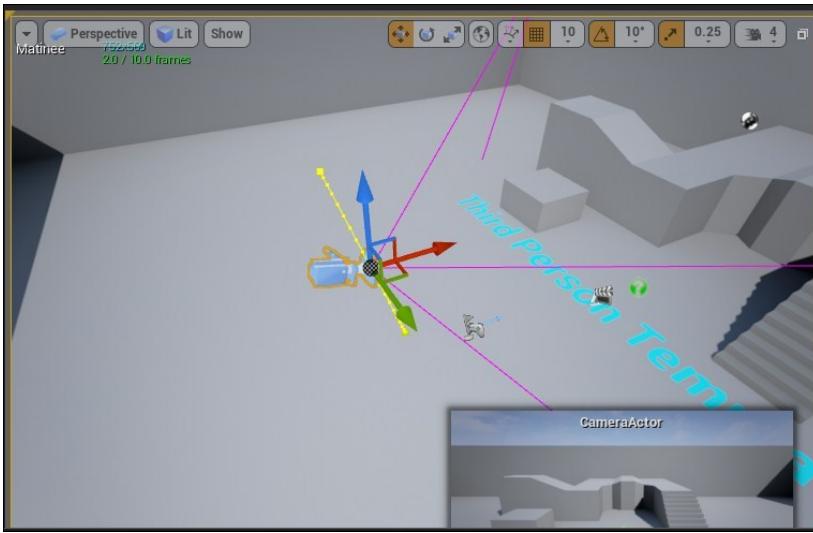


*Modifying the rotation of the CameraActor object*

If you have both the editors up at the same time, you'll notice that you can move the time bar and see the transition and exactly how it is done. You can also press the **Play** and **Loop** buttons to watch it in real time. For instance, when the editor shows it at this time...



...the editor window will look similar to this:



### Tip

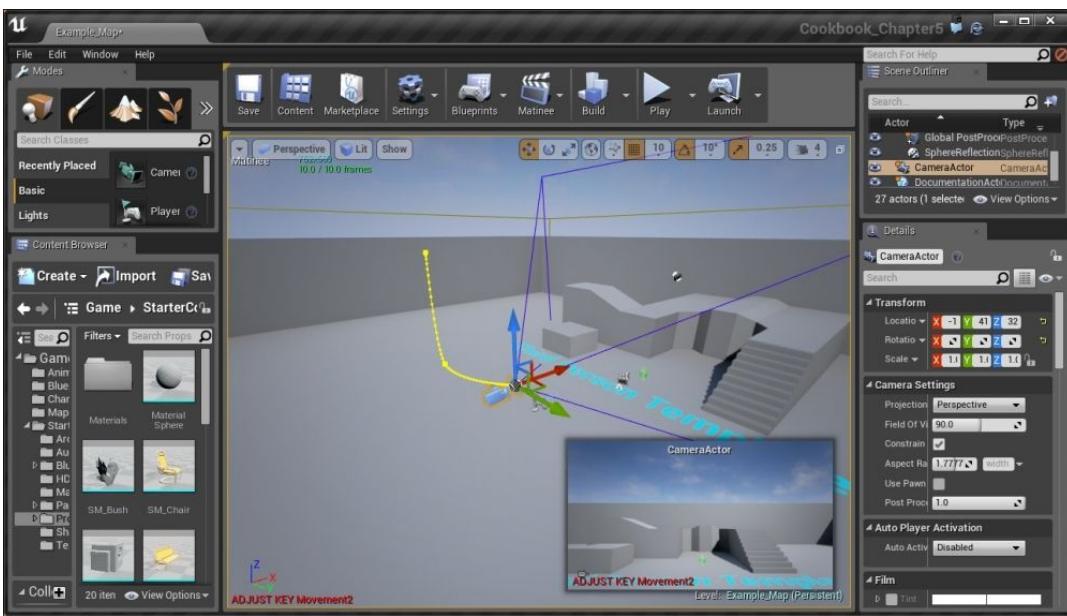
You can also click on the camera icon, next to `Camera_01` in the **Tracks** tab in the Matinee Editor so that the entire viewport of the level editor shows the view of the camera during the animation.

- After creating this first key frame, we want to add another one at the end of the animation to bring the camera forward, so we can create another key by pressing the *Enter* key. After this, right-click on the newly created key and select **Set Time** and under **New Time**, put in 5 .

This can be nice when you know how exactly you want things to change. Play around with the key frames, adding and modifying them, until you get whatever animation you want to have.

### Tip

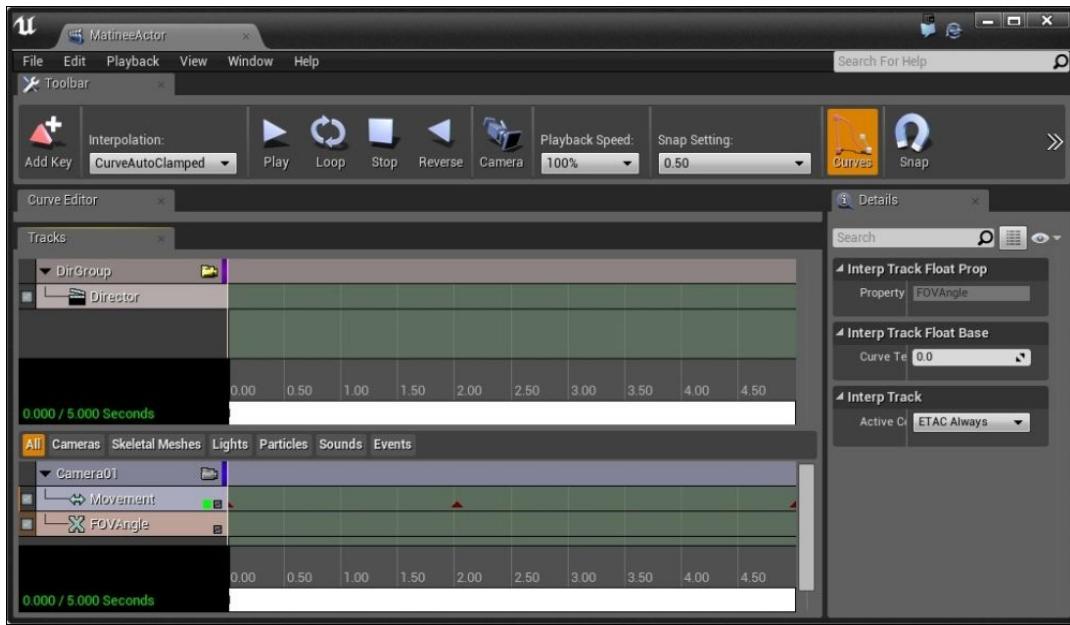
Another way to get the exact numbers for your key frames is by using the **Snap** option, which you can select from the toolbar or from **View | Snap**. This will make it so that when you add a key, it will be to the nearest .5 second or when you move, it will snap between the options.



### Tip

When in the Matinee Editor, you can manually move key frames after placing them initially into the timeline by holding *Ctrl* and clicking to drag that key frame in the timeline in both a positive and a negative direction.

- Lastly, by default, Unreal will use whatever camera is attached to the player. We need to use something called a **director group** in order to dictate which camera we should be using at what time. So with that in mind, right-click underneath the `Camera01` section under the **Tracks** tab and select **Add New Director Group**.



*Creating a director group*

- With this in mind, we now have a **DirGroup** control in place with no key frames. Move the time bar to 0, select the **Director** property for modification, and create a new key by clicking on **Add Key** in the top-left of the screen. Once there, it will ask which camera you wish to use. Select **Camera01** and press **Ok**.

You may notice that on the top-left of each group, there is a camera icon on the far right of the different groups with the **DirGroup** currently being selected. This shows which camera you are currently looking at in the **Game** viewport in the main editor. This can be nice for visualizing what you are doing.

- When the animation is over, we want to return the control of the camera back to the player, so create a new key at the end of the animation and change it to use **DirGroup**. This will instantly change back to the player's camera, but we can also add in a transition by right-clicking on the key and selecting **Set Transition Time** and putting in 1 for the amount of time in seconds we want the transition for.

## Playing a Matinee via Blueprints

After you have your animation ready, we need to play it in the actual game.

### Getting ready

Before we start working, you should have a level opened and have a completed Matinee. For assistance with this, check out the *Creating an opening cutscene* recipe.

### How to do it...

You need to execute these steps to play a Matinee via Blueprints:

- Exit out of the Matinee Editor if you are in it and back in the **Game** tab, select the **MatineeActor** object inside the scene (it looks like a clipboard) and then from the top toolbar, navigate to **Blueprints | Open Level Blueprint**.

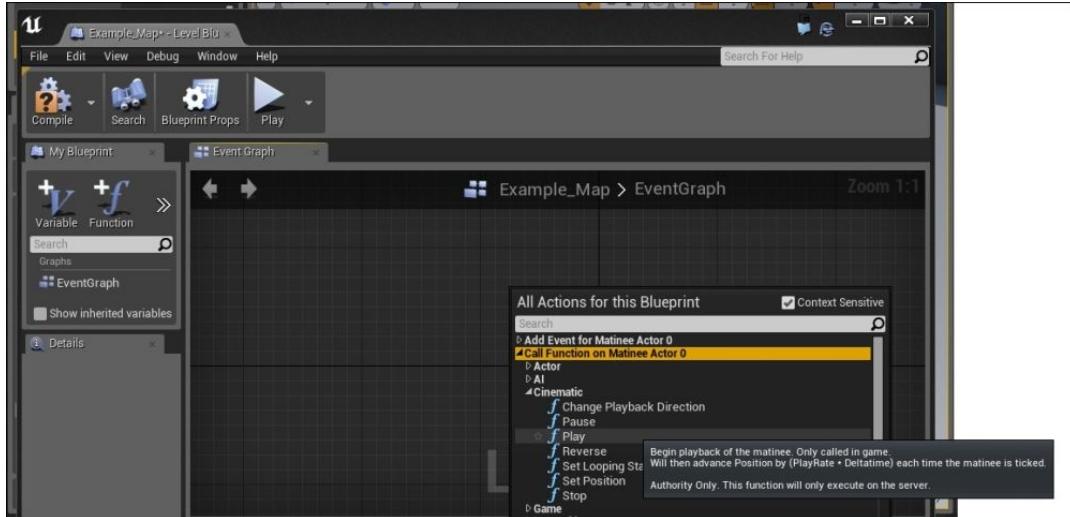
#### Note

Alternatively, you can go to the **Scene Outliner** tab and select the object from there, either by scrolling down until you see it or typing in **Matinee** in the **Search** bar above it.

- Once inside the **Blueprint Editor**, right-click inside the **Event Graph** tab and navigate to **Call Function On Matinee Actor 0 | Cinematic | Play**.

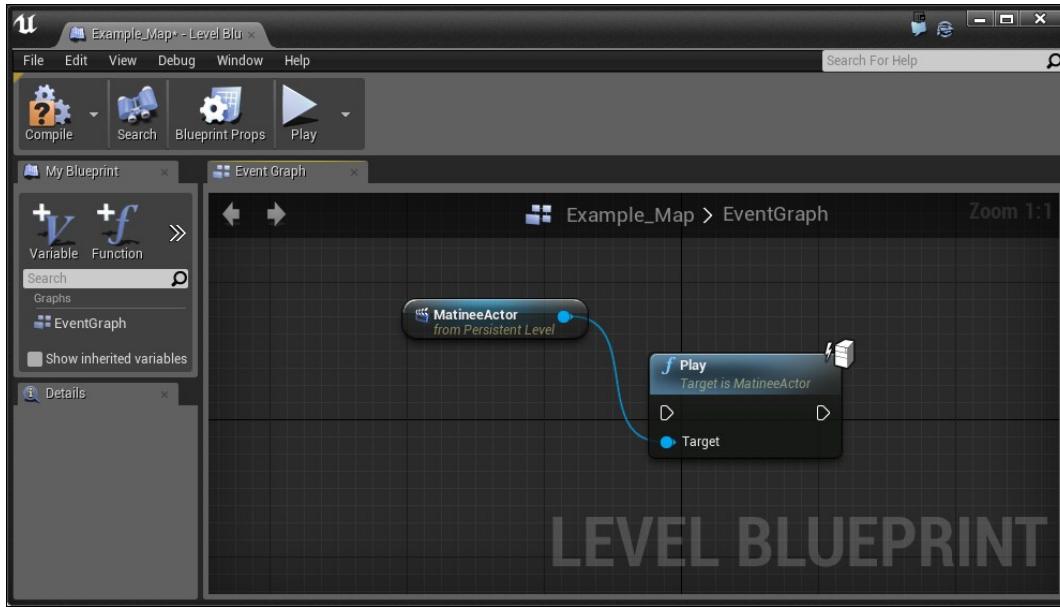
#### Tip

For the dropdowns, you need to click on the triangle on the left-hand side to expand it.



*Adding a Play action to call on the MatineeActor*

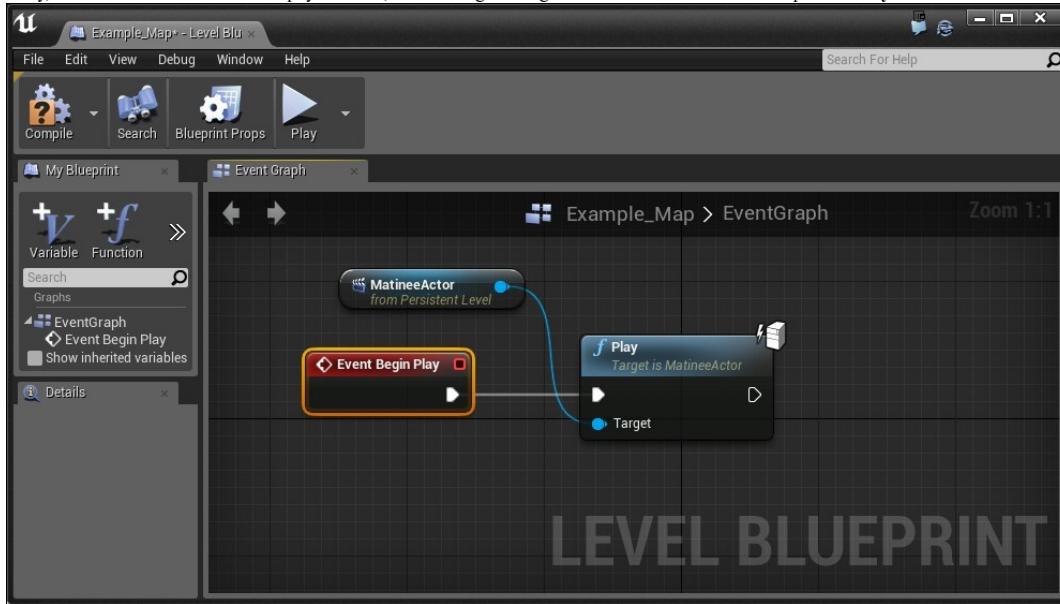
If all goes well, you should see something like this:



### Tip

If you do not see this, make sure that the `MatineeActor` object is selected from the **Scene Outliner** tab.

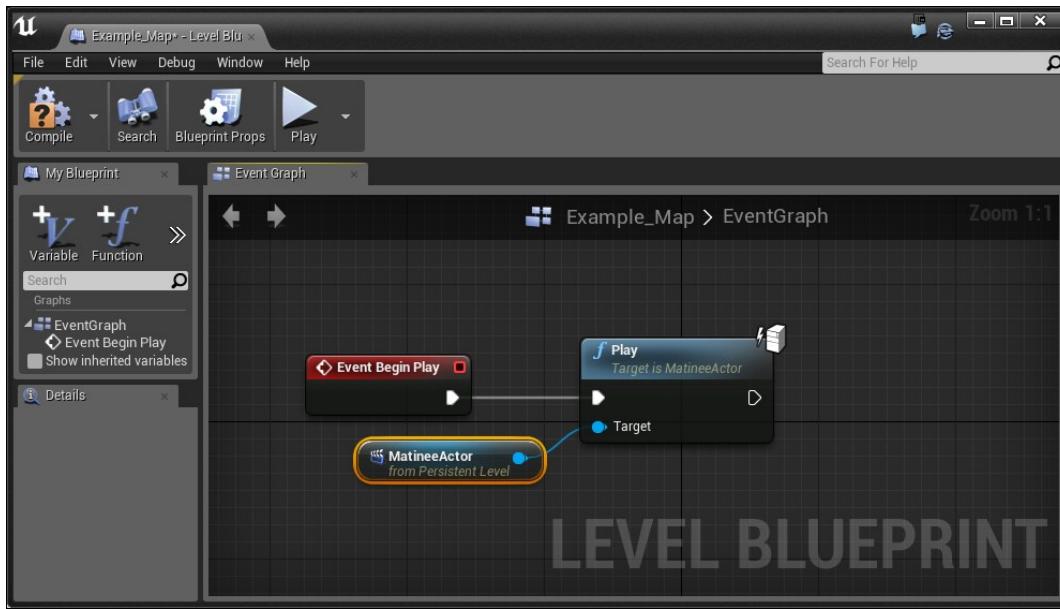
3. Now, this is calling a function called `Play`, which will play our Matinee cutscene, but there is no indication as to when this should be triggered (or called, in programmer lingo). So, we need to add an **Event**, saying to play this animation when the game starts. To do this, right-click and go to **Add Event | Event Begin Play**.
4. Lastly, we need to connect the event to the play action. So, click and drag on the right-side's arrow and connect it to the input of the `Play` action on the left-hand side.



### Tip

When working with Blueprints, you may hear actions referred to as actions, nodes, or whatever type they are (event, function, and so on).

5. Lastly, we'll clean up a bit by clicking and dragging the `MatineeActor` variable below, near the `Target` connection.



- Now, jump in and play the game! Just like we were planning, we fall down to the player and move back to the normal player's control.

#### Note

For more information on working with Matinee, refer to <https://docs.unrealengine.com/latest/TNT/Engine/Matinee/UserGuide/index.html>.

## Preventing a player from moving via cinematic mode

Now that we know how to create Matinees, we should be able to create amazing cinematic that make your levels really come alive! However, by default, when you play a cinematic, the player can still move around and turn. This may be the desired behavior depending on whether you're using something like a custom camera system, such as *Resident Evil*, or the Matinee for moving objects such as moving platforms. But, for a cinematic, we don't want it. Thankfully, there is a way to prevent this via Blueprints.

### Getting ready

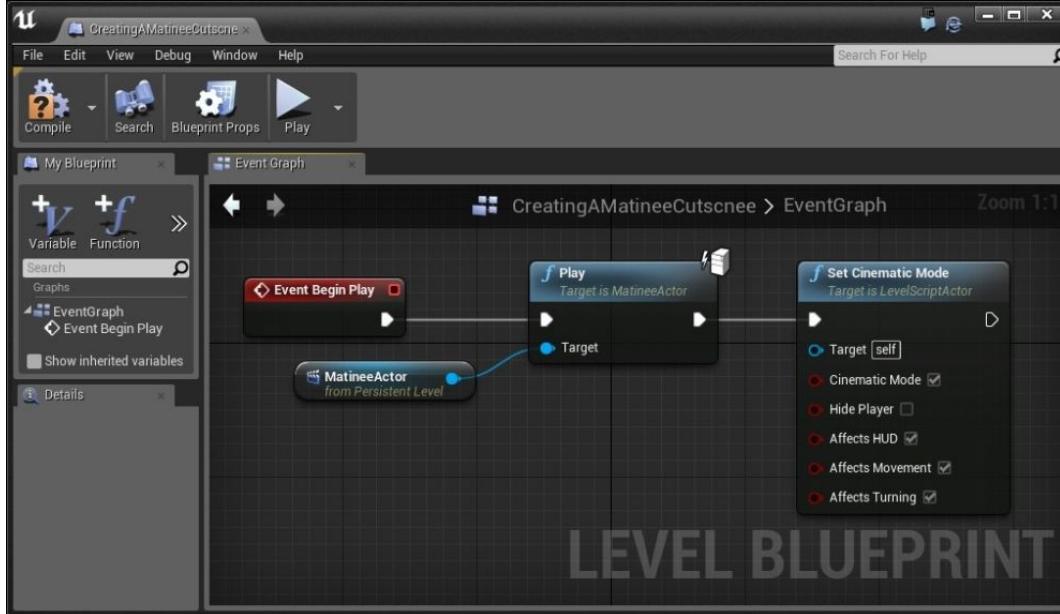
Before we start working, you should have a level opened and have a complete Matinee. You should also have it playing in the actual game. For assistance with this, check out the *Playing a Matinee via Blueprints* recipe.

### How to do it...

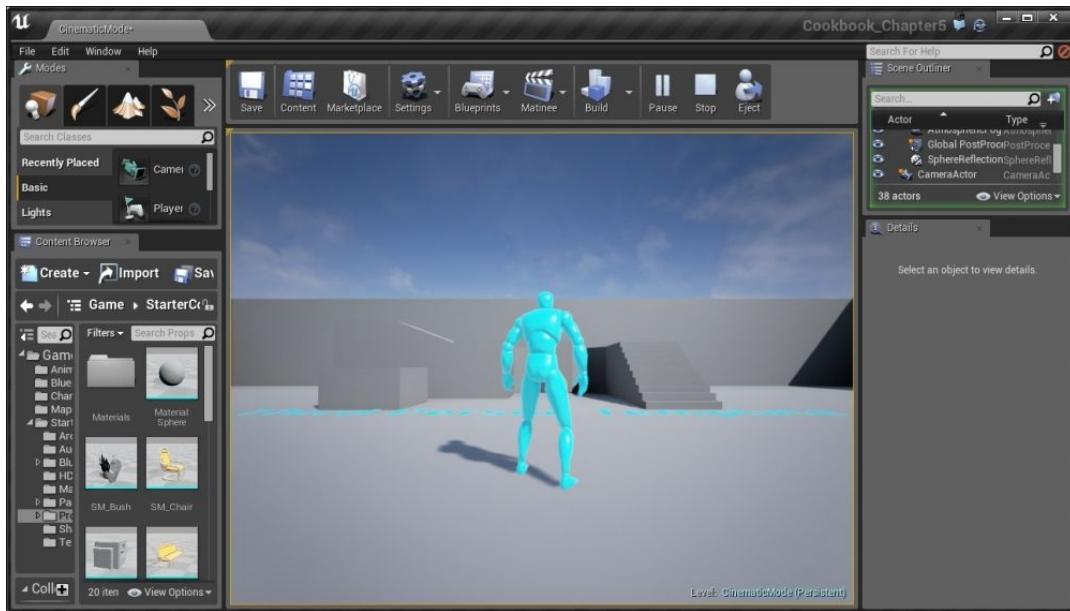
- Open up the level blueprint by going from the top tool bar and navigating to **Blueprints | Open Level Blueprint**.
- From here, we should have the appropriate nodes to play our Matinee.
- To the right of the **Play** node, right click and type in **Cinematic** in the search bar. At the bottom, you'll see the **Game | Cinematic | Set Cinematic Mode** option. Select it to create its node and drag it so that the arrows are alongside each other.

The **Set Cinematic Mode** option enables us to remove the game's HUD, hide the player, and prevent movement and turning in the game. Of course, depending on the situation, you may want to use some or none of these options so that they're able to be customized from the node.

- Connect the output of the **Play** node to the input of the **Set Cinematic Mode** option. Toggle the **Cinematic Mode** option to turn on the cinematic mode. Uncheck **Hide Player** so that we can still see the player. Then, check **Affects Movement** and **Affects Tuning**: this will prevent the player from moving or using the mouse to move around while the cutscene is playing.

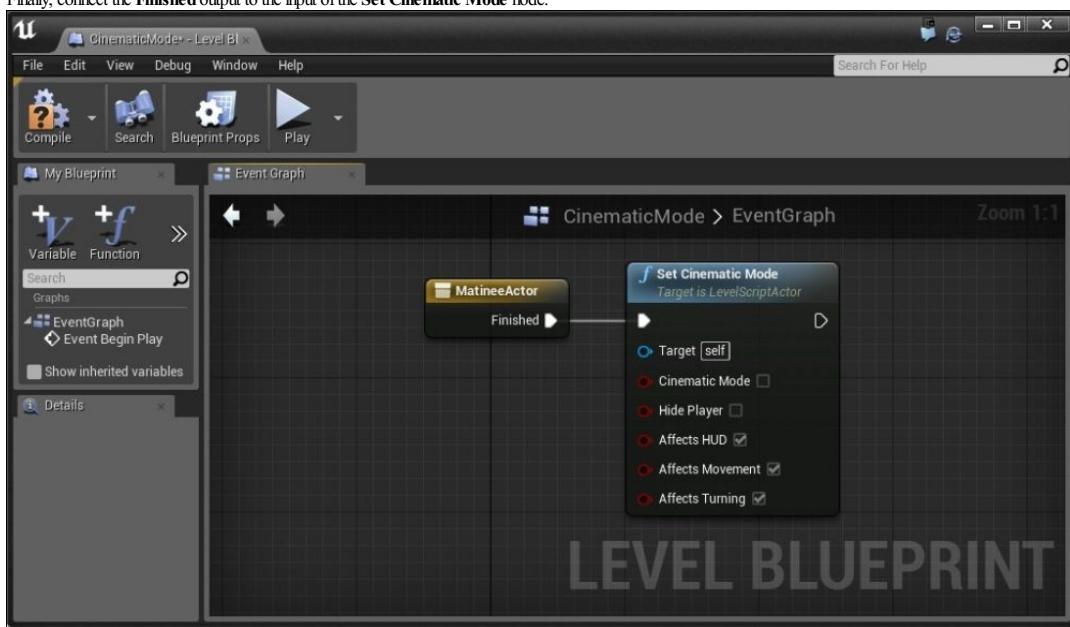


If you were to play the game now, you would notice that it works...maybe a little too well.

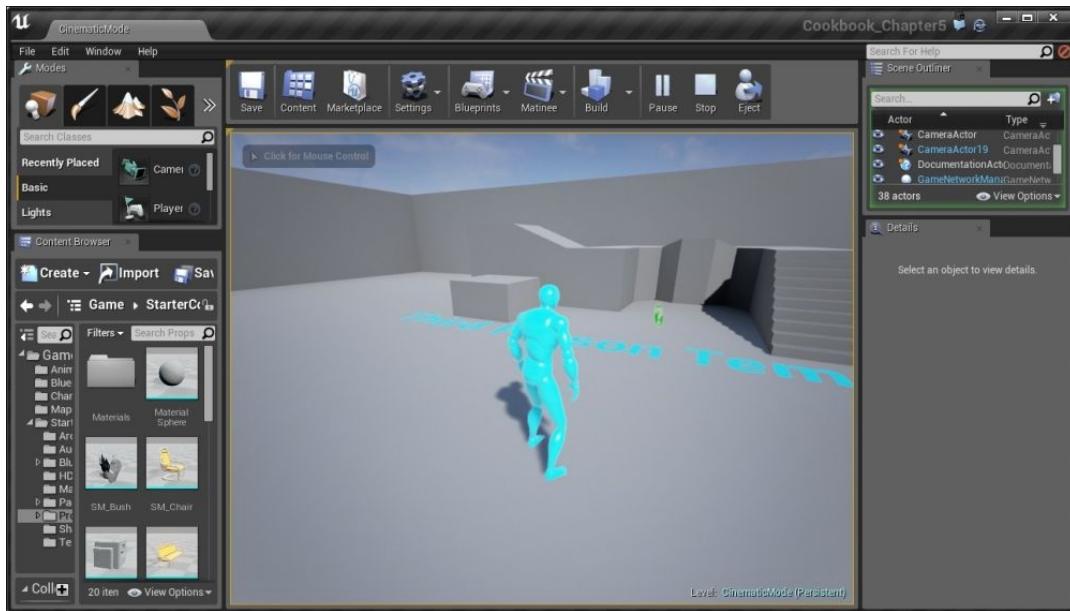


That's because after we enabled cinematic mode, we never disabled it. Let's do this next.

1. To fix this, we will need to have an event whenever the Matinee ends. Select your **MatineeActor** in the **Scene Outliner** tab and then go back into your level blueprint.
2. From here, right-click and select **Create a Matinee Controller for MatineeActor**. You'll notice that it gives us a new node that contains a new **Finished** output that will be activated whenever the matinee will end.
3. Next, create another **Set Cinematic Mode** action and uncheck **Cinematic Mode** and **Hide Player** while checking the other three properties below.
4. Finally, connect the **Finished** output to the input of the **Set Cinematic Mode** node.



5. At this point, if we go in and play the game, the player will not move during the cinematic, but as soon as it's over, it can!



The final view after adding cinematic

## Tip

To do this in another way, you can also use a **Delay** action and then set it to another cinematic mode or even use Boolean values, once you've got more experience in scripting to toggle their properties.

## See also

There is an amazing amount of things that you can do within the Matinee editor. Sadly, I don't have the space in the book to write about everything, but I do have some links that, if you would like to know more, about some of the many other things you can do:

- In addition to cutscenes, you can also use Matinee to create gameplay elements such as moving platforms. A nice text tutorial about doing that is available at [https://wiki.unrealengine.com/Blueprint\\_Lift\\_Tutorial](https://wiki.unrealengine.com/Blueprint_Lift_Tutorial).
- Vincent Stimpson has a series of two tutorial videos on Matinee—the first deals with cameras with a fade effect (<https://youtu.be/0tQFrJElkio>), and the second video is on some of the advanced techniques that you can use to work with Matinee with animations and particle effects (<https://www.youtube.com/watch?v=071KTzYiyo>).

## Chapter 6. Lighting and Shadows

In this chapter, we'll cover the following recipes:

- Lighting overview – learning the types of lights
- Adding moveable lights – flashlight, part 1
- Creating a Day/Night cycle

## Introduction

Lighting is one of the most important elements when creating environments. It can communicate the theme, create atmosphere, and give a certain tone to each place that you use it for. However, if done poorly, it can make levels look amateurish. In this chapter, we will look at how to bring life to our game worlds.

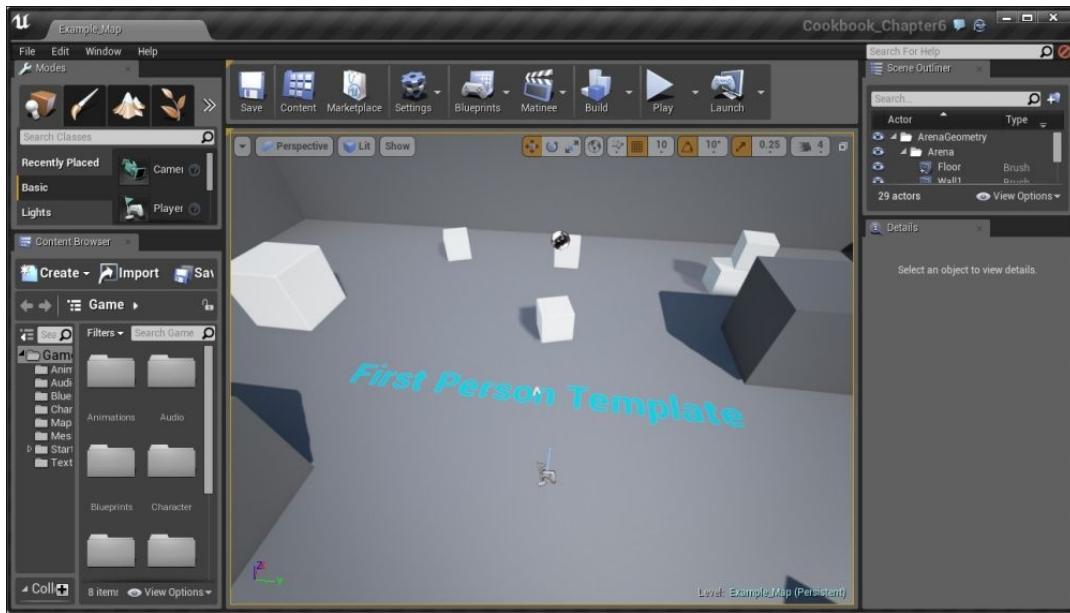
## Lighting overview – learning the types of lights

Having defined why lighting is so important, let's start creating each of the types and talking about what they do. In this recipe, we will see each of the different kinds of lights available in the engine. We will also discuss how they are different and when and why to use each one.

### Getting ready

Before we start working within the Unreal Editor, we will need to have a project to work with:

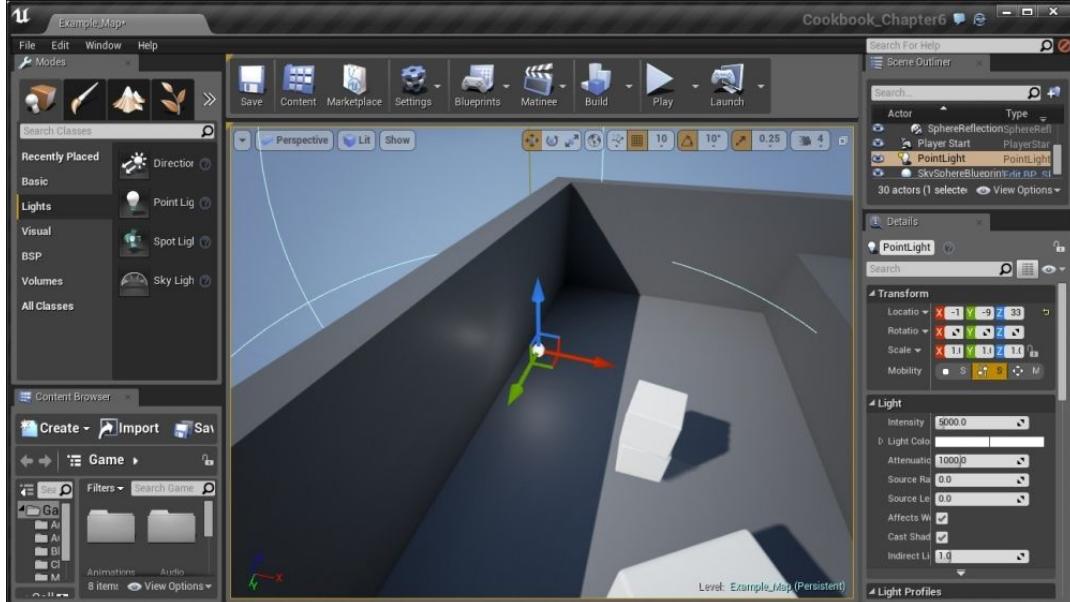
1. First, open up the Unreal Editor by clicking on the **Launch** button from the Unreal Engine Launcher.
2. Start a new project from the **Project Browser** tab by selecting the **New Project** tab. Select **First person** and make sure that **With Starter Content** is selected. Give the project a Name (`Cookbook_Chapter6`). Once you have finished, click on **Create Project**.
3. You should see a level similar to this:



## How to do it...

To get started, let's see how we can go about adding lights to our level:

1. Move the camera toward a dark area of our level that we want to light up. Once there, go to **Modes | Lights** and click on the **Point** light and drag and drop the light into your level.



*Adding a Point Light to your level*

### Tip

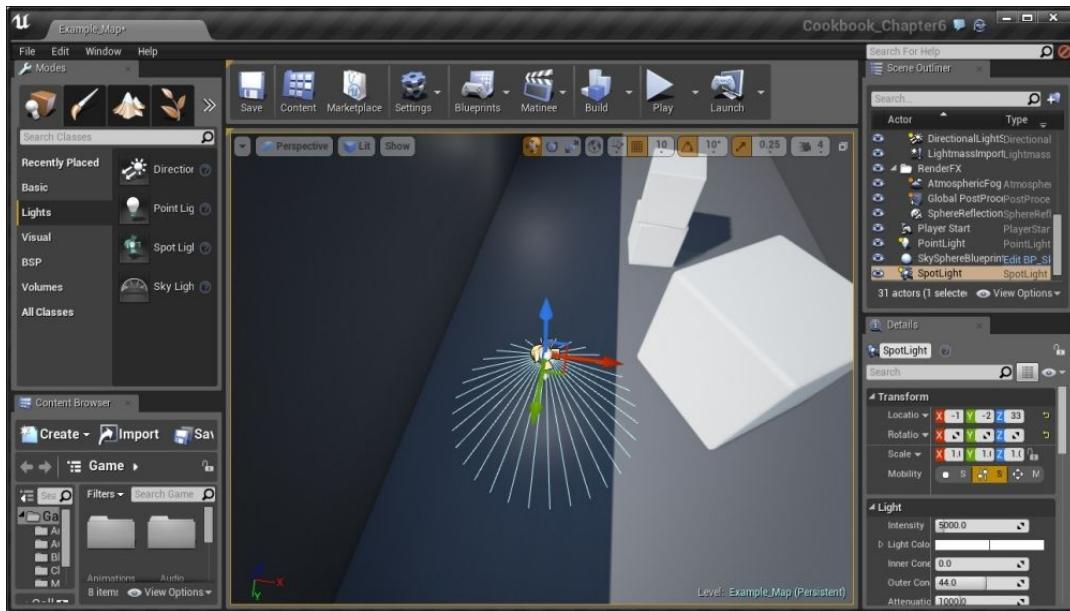
You can also press and hold the **L** key and then click to create a point light at the location that was clicked.

**Point Lights** act in a similar manner to a light bulb; they will emit light in all the directions from their center for a certain radius, losing power as the light gets further away.

### Note

For more information on Point Lights, refer to <https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightTypes/Point/index.html>.

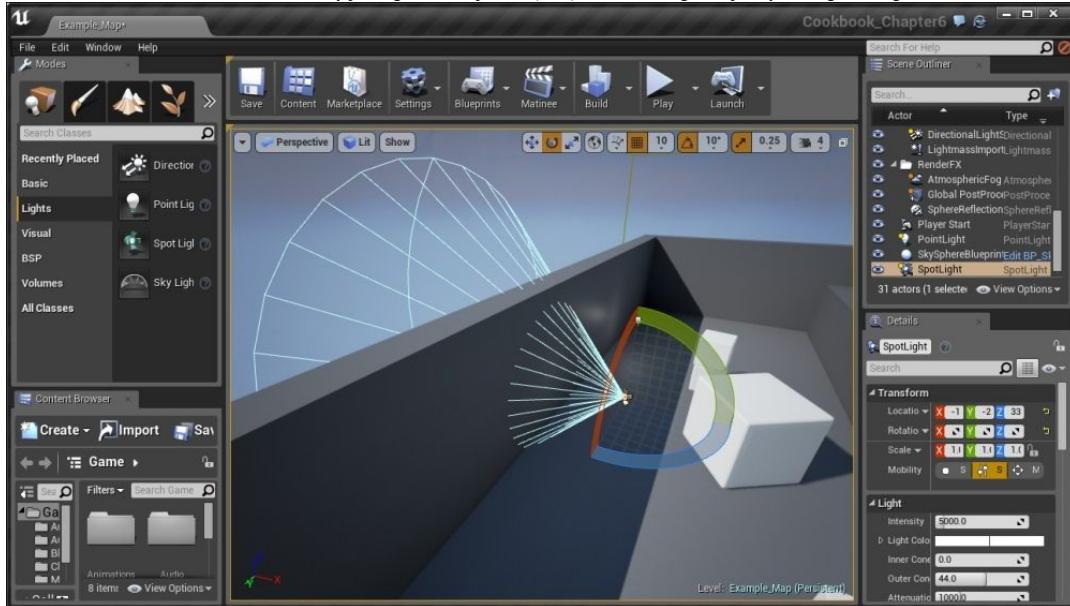
2. We can also create lights from our **Level** tab. Let's do this by right-clicking anywhere in the level and navigating to **Place Actor | Spot Light**.



*Creating a Spot Light in your level*

The object will appear facedown with lights going out in the direction that it is shining. This actually goes much further out, so let's check this out.

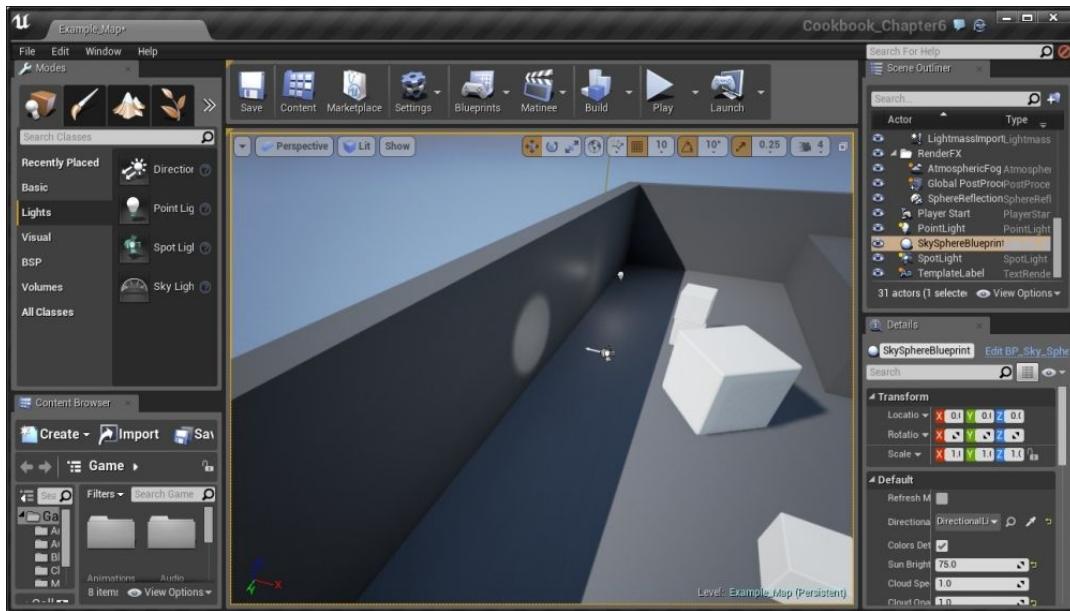
- From Translation mode, switch to Rotation mode by pressing down the spacebar (or *E*) and then rotating the object by  $-90$  degrees along the **Y** axis to face the dark wall.



*The cone shape effect of a Spot light*

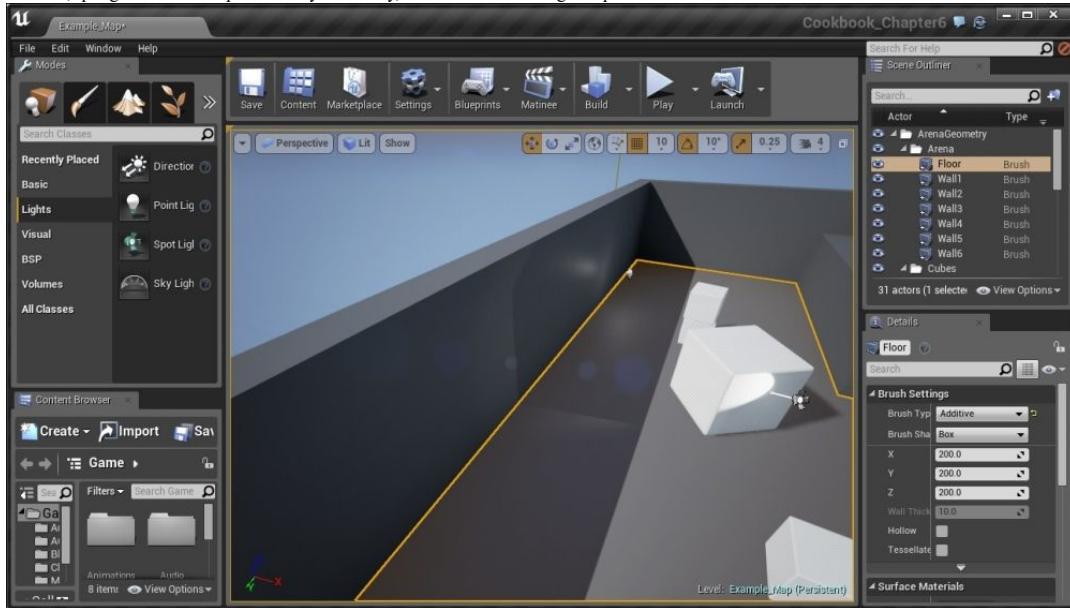
Spot lights act much like they do in the real world; specifically, they will emit light from a point out in a cone shape. However, just like real spotlights, there is a center section that is fully bright with the outer edge decreasing in power. We can do this in Unreal as well using the **Inner Cone Angle** and **Outer Cone Angle** properties.

- From the **Light** section under the **Details** panel, set the **Inner Cone Angle** value to  $25$  and the **Outer Cone Angle** value to  $32$ . After that, increase the **Intensity** to  $50000.0$ .



*Managing cone angles and intensity*

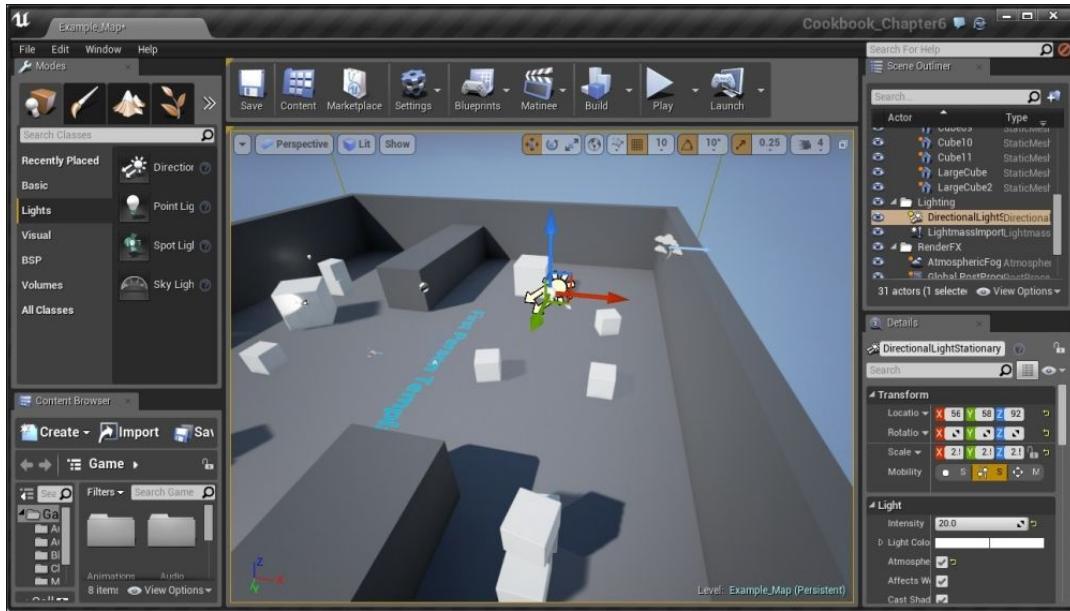
However, spotlights can also lose power as they move away, as shown in the following example:



### Note

For more information about Spot Lights, refer to <https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightTypes/Spot/index.html>.

5. The next light I want to point out is already in the example level by default. Go to the Scene Outliner tab and select the **DirectionalLight** object by double-clicking on it.



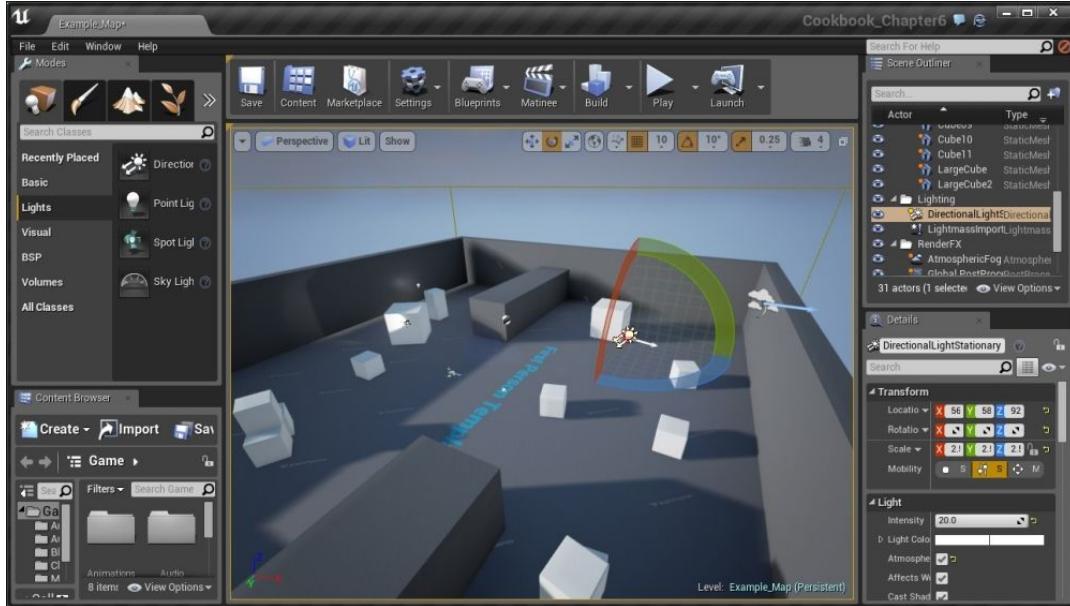
Selecting directional light

**Directional lights** act much like the sun. Depending on their properties, they affect the entire outer section of the level. You'll notice the arrows pointing in a certain direction which is where the light will be emitting from.

6. Rotate the light by 40 degrees in the Y direction.

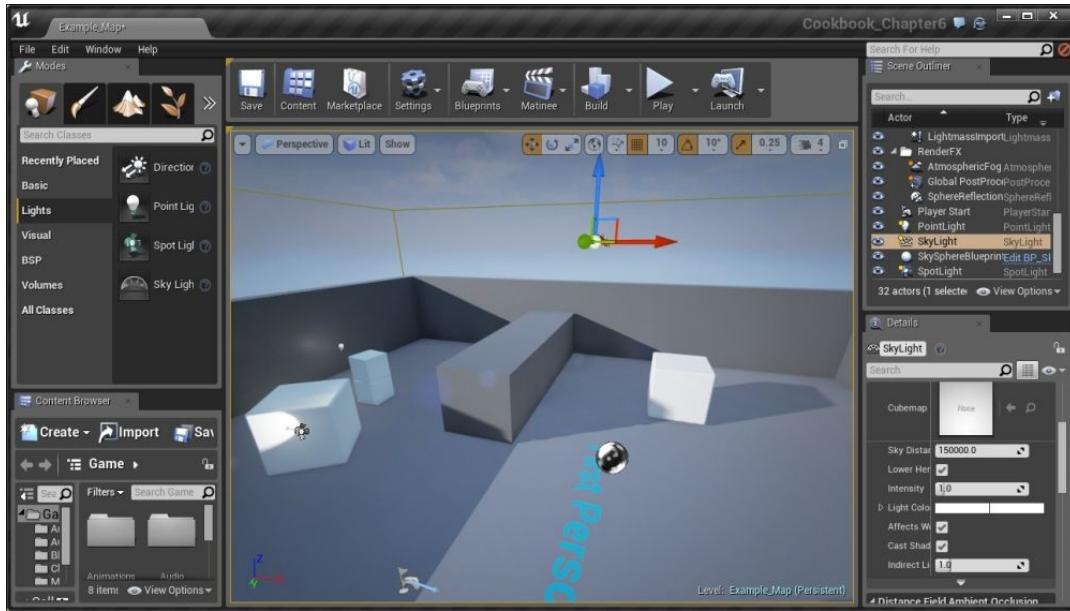
#### Note

For more information on Directional Lights, refer to <https://docs.unrealengine.com/latest/TNT/Engine/Rendering/LightingAndShadows/LightTypes/Directional/index.html>.



Rotating the directional light

7. Now the last light is the **Sky Light**, which is used to create ambient light from the distant areas in the world. It then applies that light to the world so that the level's lighting will match the rest of the level. To add this, use either method mentioned for the Sky Light and put it into the level.



8. The Sky Light will only capture the screen when we rebuild the lighting by selecting **Recapture Scene** or manually going to **Build | Update Reflection Captures**.

#### Note

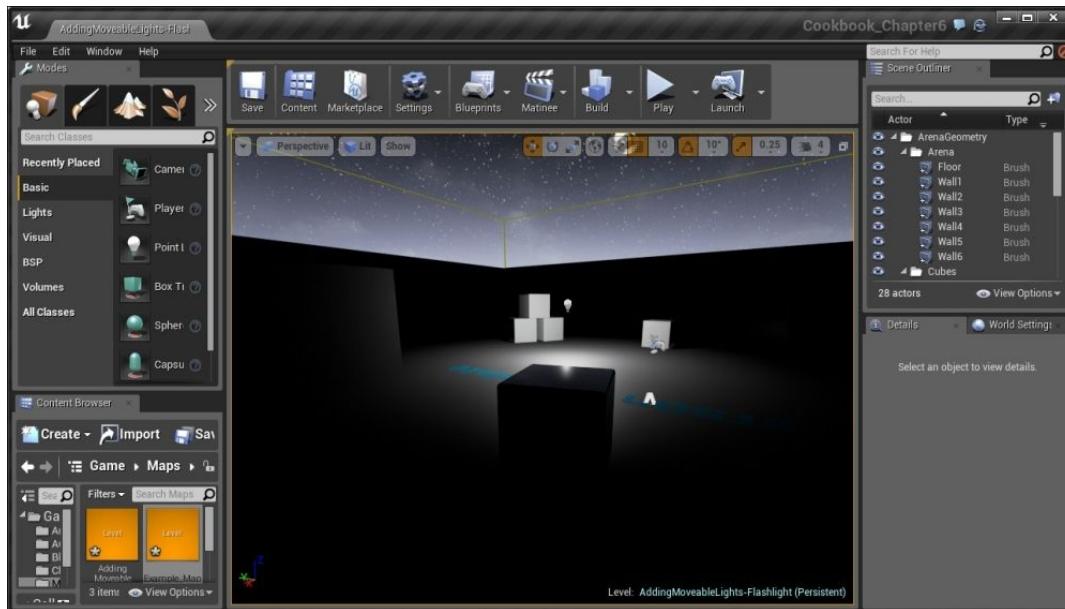
For more information on Sky Lights, refer to <https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightTypes/SkyLight/index.html>.

## Adding moveable lights – flashlight, part 1

Now we have an understanding of how these lights work, but so far, they've all been static, non-moving. However, in certain instances, we may want the lighting to change while the game is going on. Let's do that by creating a moving light, in a similar manner to a flashlight!

### Getting ready

Before we start working, we should have the `NightScene` level opened. This is provided in the `Example` Code folder for this chapter that you can get off of Packt's website.



*The NightScene level*

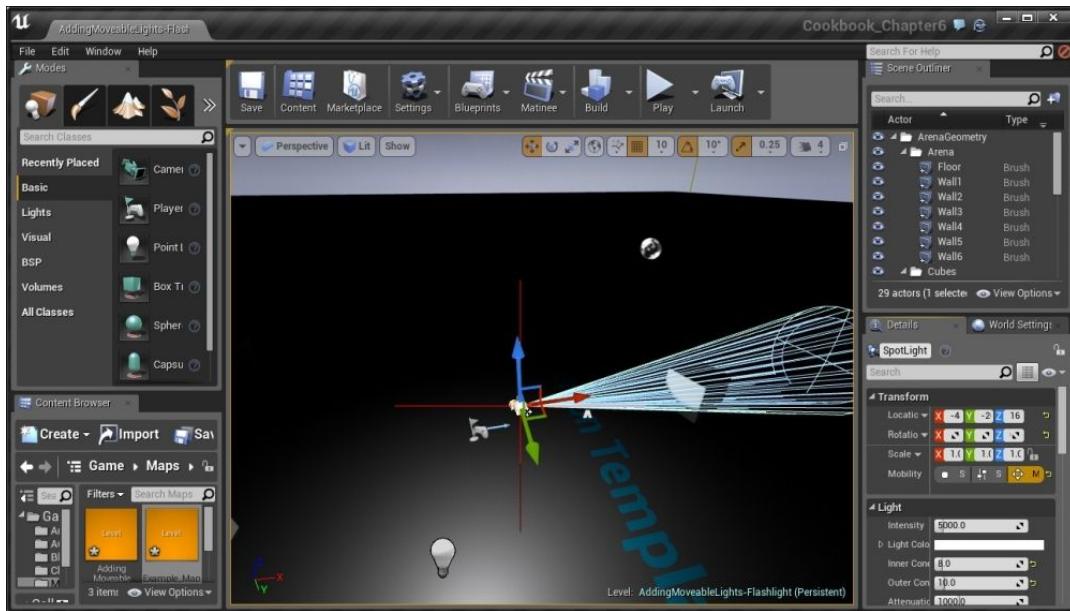
#### Note

This level is a quick example of a night environment with the lighting reduced significantly. For more information on creating a scene using lighting for night time, refer to <https://docs.unrealengine.com/latest/INT/Resources>Showcases/RealisticRendering/NightScene/index.html>.

### How to do it...

To create a flashlight, perform the following steps:

1. The first thing we will want to do is create a **SpotLight** to use as our flashlight. We can do that by right-clicking on the ground near the player and then navigating to **Place Actor | Spot Light**.
2. Rotate the spotlight 90 degrees so that it is facing away from the player's spawn point. Then, change the **Inner Cone Angle** value to 8 and the **Outer Cone Angle** value to 10. Lastly, we want this actor to be movable so that it can move, so in the **Details** tab under **Transform | Mobility**, select the option that is the furthest to the right (**Movable**).

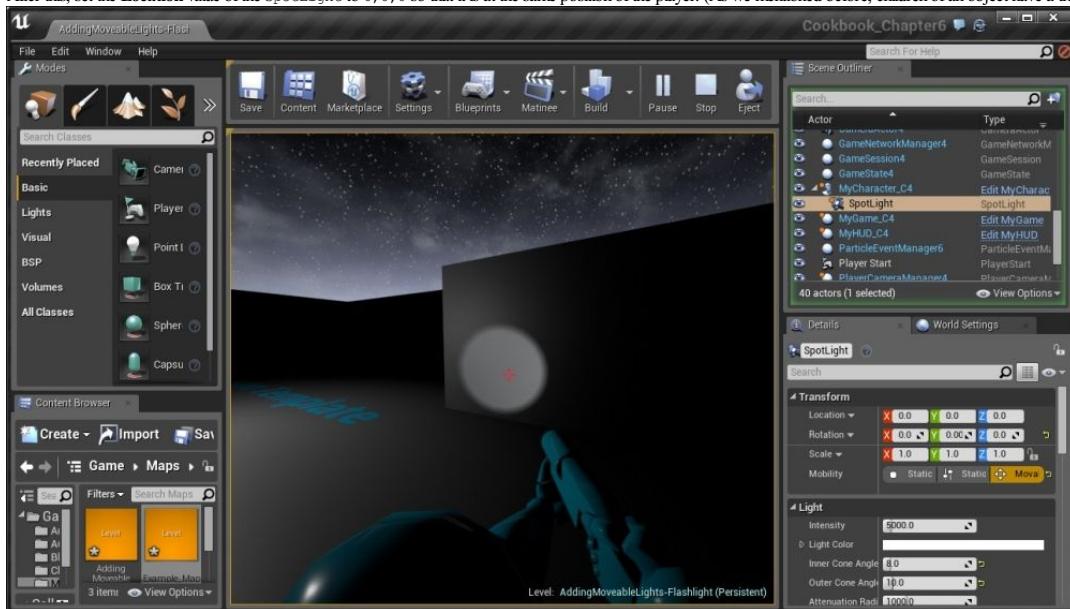


*Adjusting the cone angles of the spot light and making the player movable*

#### Note

For more information on moveable lights, refer to <https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightMobility/DynamicLights/index.html>.

3. Now if you play the game, you will see additional objects in the **Scene Outliner** tab, including an object called `MyCharacter_C4` (or something similar). Select it and scroll down to your `SpotLight`. Drag and drop the `SpotLight` object on top of the `MyCharacter` object (use the mouse wheel to move the object selections up while holding down the mouse).
4. After this, set the **Location** value of the `SpotLight` to `0, 0, 0` so that it is at the same position of the player. (As we mentioned before, children of an object have a transition relative to the position of their parent.)



*Setting the Location value of the spot light*

As you can see, we now have a working flashlight in Unreal. There are a few issues with this, for example, if we look up or down, the light will not go with us because the camera has a different transform than our `Character` class. There's also the issue that since we did this in the game (after the player has been spawned), when we will leave, it'll reset to what it was beforehand. We know that the functionality is possible now, so how can we make this work all the time, and have functionality? Adding it to our `Character`'s Blueprint, we'll dive into how to do this exactly, and the other things to be considered in our *Adding to an existing Blueprint – flashlight, part 2* recipe, which you can find in [Chapter 8, Blueprint Scripting – Level Effects](#).

## Creating a Day/Night cycle

If we are creating an open world title or any kind of game where the player is outdoors, one thing that would be nice is the ability to have our world change the time of day at the same rate.

### Getting ready

Before we start working, we should have a level created with a `Sky_Sphere` and a directional light (the default map has this). To be sure, you can use the `DayNightCycle-Before` map from the example code folder.

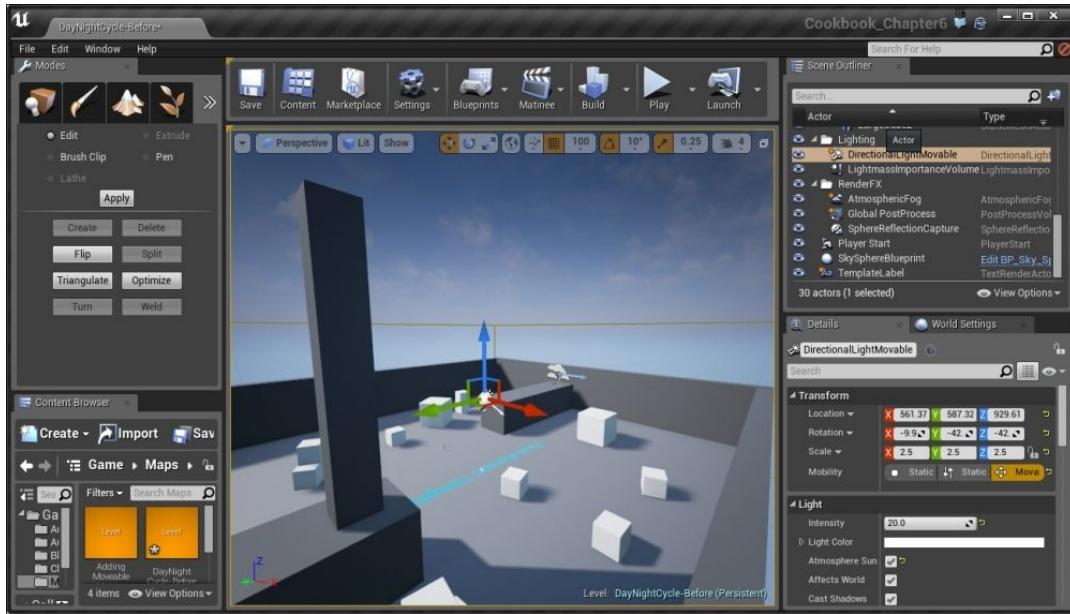
### How to do it...

Perform the following steps to create a day/night cycle:

1. As we discussed before, we can use Directional Lights to act like the sun. For our day/night cycle we will want our lighting to change over time, so let's select the `DirectionalLightStationary` object from the **Scene Outliner** tab and then from the **Details** tab, change the **Mobility** property to `Movable`. Finally, change the name of the object to `DirectionalLightMovable`.

#### Note

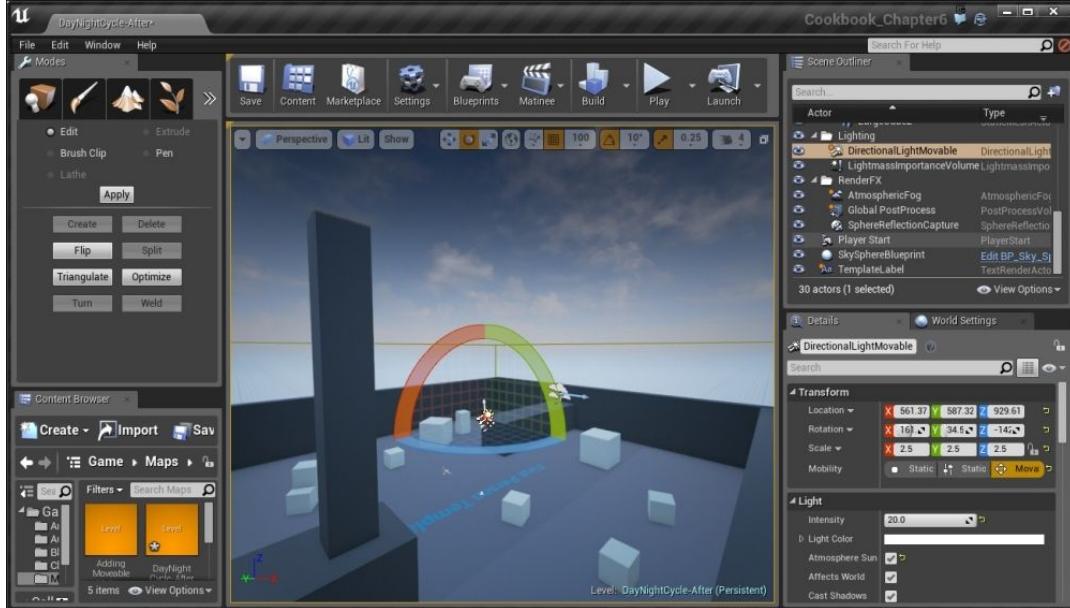
You can either rename the object from the **Details** tab or press `F2` when selected in the **Scene Outliner** tab.



*Making our DirectionalLight moveable*

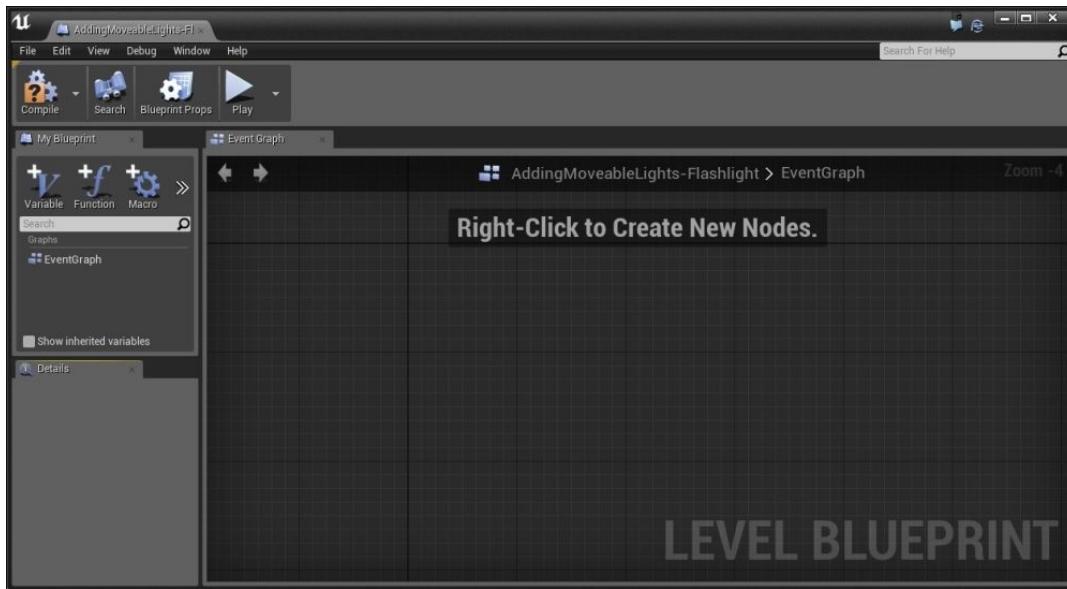
- To have this lighting change reflect correctly, the next thing we should do is build the lighting of our level by going to the toolbar above our level view and navigating to **Build | Build Lighting Only**.

Now, if we were to rotate the light right now, we would see the Sky's color will be modified:

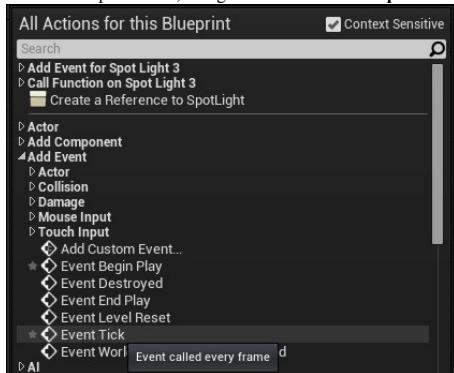


However, the sun that is part of our `SunSphere` will not be modified by the changes. To fix this, we will need to use Unreal's Blueprints system.

- Next, from the top-center toolbar, navigate to **Blueprints | Open Level Blueprint**, and you should be brought to the following screen:

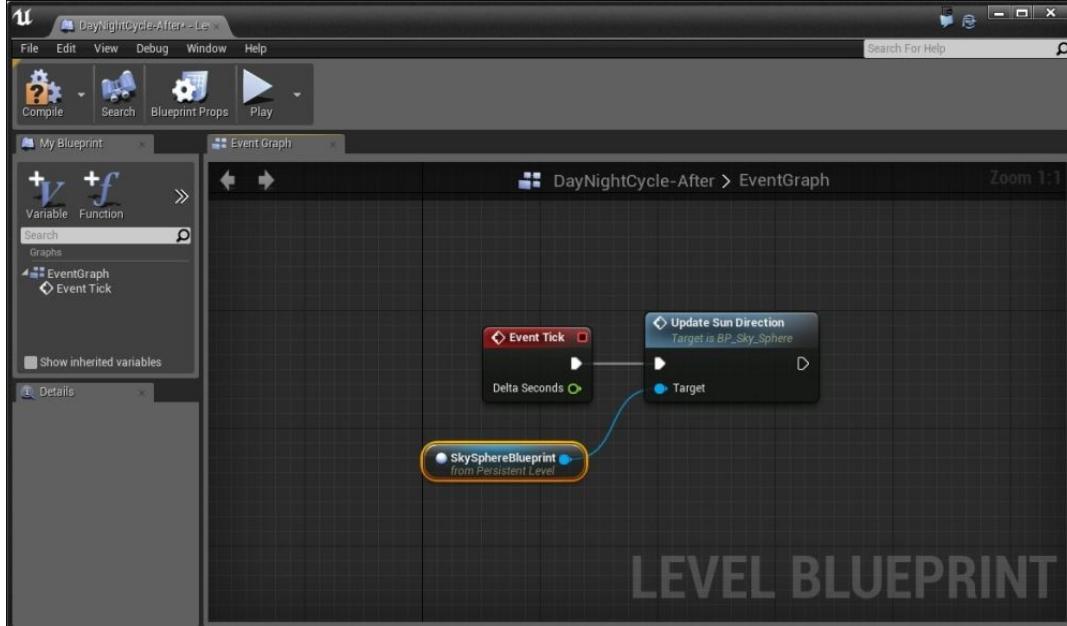


4. From the Blueprints editor, we right-click inside **EventGraph** and then select **Add Event | Event Tick**.



This creates a new event that will get called every frame that it is active and it has a parameter of **Delta Seconds**, which is how much time has elapsed for this particular frame.

5. We want to have the Sky Sphere's sun at the same position as our Directional Light, so let's do that first. Go to the **Scene Outliner** tab and select the `SkySphereBlueprint` object by clicking on it. Once this is done, go back into the Blueprints screen and to the right of the Event Tick event, type in `Sun` to show the properties that contain those letters. From there, select the **Update Sun Direction** option:

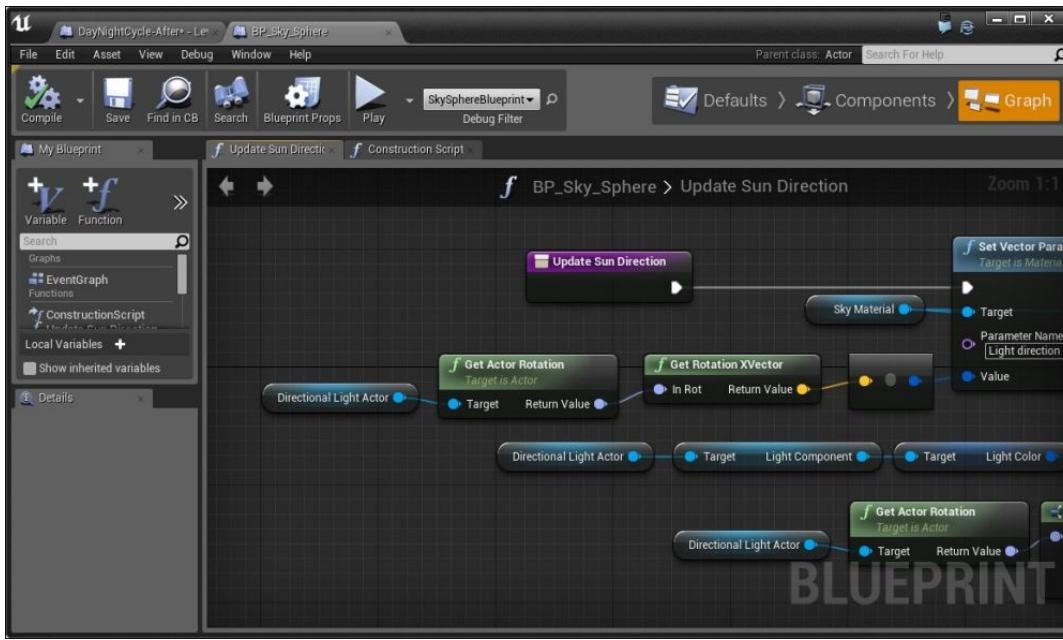


If the object was selected correctly, we should see the **Target** variable already filled in with the `SkySphere`.

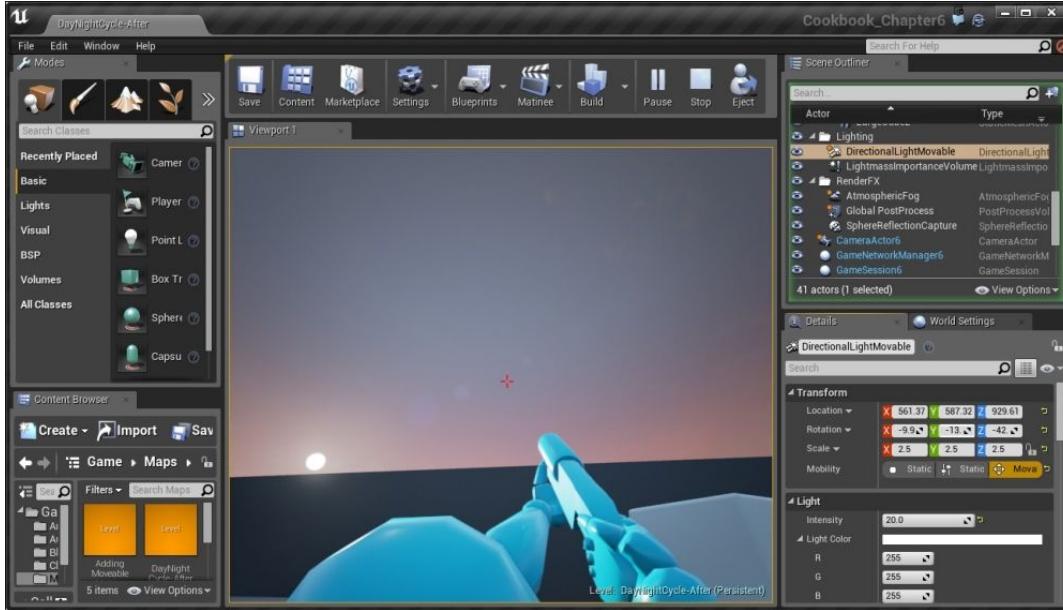
To zoom in, use the mouse wheel. This action will find the rotation of the directional light in our level and modify it so that the sun will face the same way.

#### Note

If you want to see what **Update Sun Direction** is actually doing, feel free to double-click on the action. We'll talk about what those actions mean later on in [Chapter 8, Blueprint Scripting - Level Effects](#).

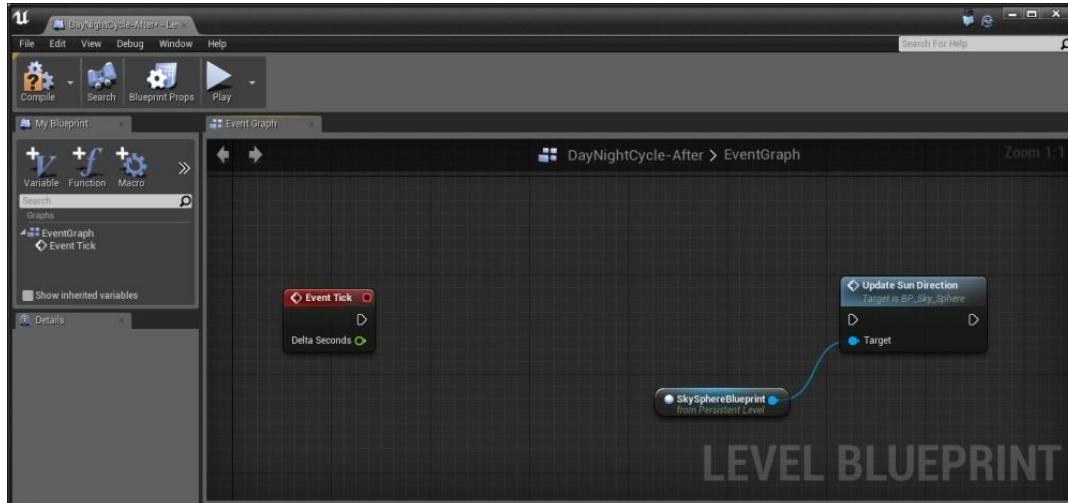


Blueprints are run while the game is played, so at this point, if we play the game, selecting the `DirectionalLightMovable` object, and rotating it from the `Details` tab, we'll notice that the sun in the sky rotates to face it!



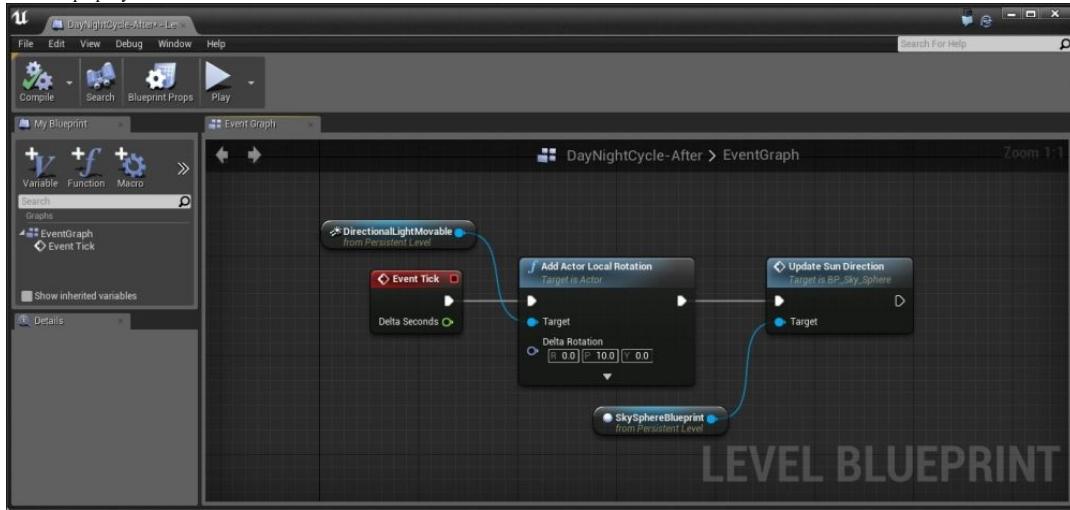
Of course, at this point, we could rotate the light using a matinee for cutscenes and we'd be done, but in this instance, we want it to move automatically, so let's modify the level blueprint to do that.

1. Next, we want to have the directional light continuously rotate along the Y axis. To do this, let's first move the `Update Sun Direction` action and the `SkySphere Blueprint` object off to the side so that we can put something in between by selecting both items and then dragging them. Finally, highlight the arrow from the left side of the `Update Sun Direction` action, hold down the `Alt` key and then click in order to break the connector.



The current state of the Event Graph (note the broken connection)

2. From there, select the **DirectionalLightMoveable** actor from the **Scene Outliner** tab and then right-click on **Event Graph** and select **Add Actor Local Rotation** (searching for **Rotation** from the top search bar). Connect the **Event Tick** event's output arrow to the **Add Actor Local Rotation** node and then connect its output to the **Update Sun Direction** input. Finally, change the **P** (Pitch) value in the **Delta Rotation** property of the **Add Actor Local Rotation** action to **10**.



Working with **DirectionalLightMoveable** actor in the blueprint

The **Add Actor Local Rotation** node takes whatever value the **Target** object (directional light) has as its current Rotation and adds to it by the Delta Rotation vector. We do this before we update the sun's direction because we want the sun and the light to be at the same position (the sun would be a tad off, otherwise). The **10** value acts as the speed at which we want to move in that direction and can be modified to change the speed of our movement.

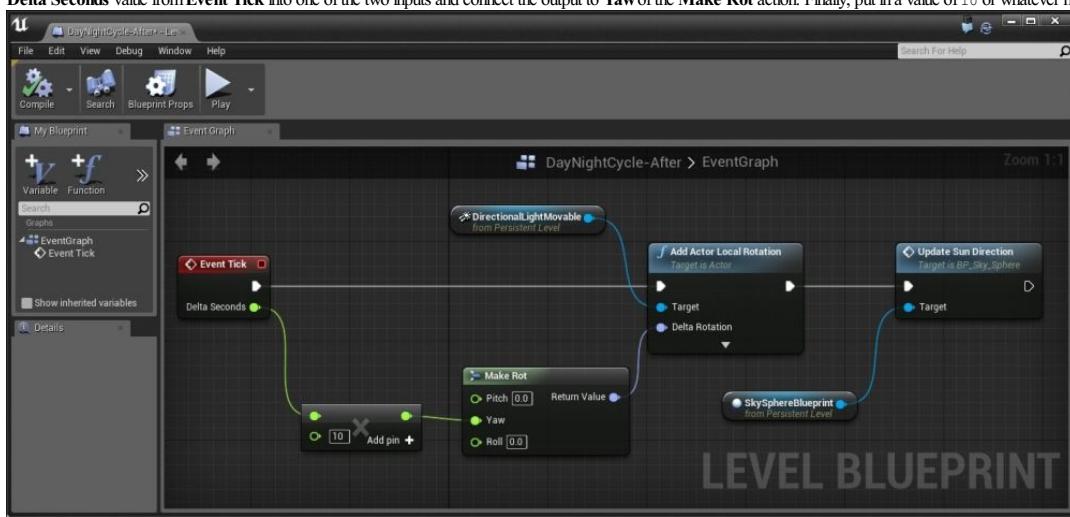
Now if we play the game, we'll notice that the sun is moving exactly as we wanted, but it is moving extremely fast! We could modify this value to fit whatever speed we'd like, but first, there's something that we need to note as we build even more complex actions—the **Delta Seconds** property that we talked about earlier.

The **Delta Seconds** value (commonly referred to as the **Delta Time** or **dt** in mathematics) is extremely important because in games, even though we often want things to run as quickly as possible (usually 60 times per second), the more complex or graphically intensive things get, the more time it takes the computer to do all of the calculations needed to update and render the completed frame. Using this value as a modifier makes sure that our events are dependent on time, not by frame, ensuring that the same things happen as time goes on. Since the value will also normally be **1/60** (or **~0.016**), being multiplied by **10** will give us a much smaller number, which is what we want.

This value is a **float**, or floating point number; this is a variable or container of information that we can store and use in various areas of code.

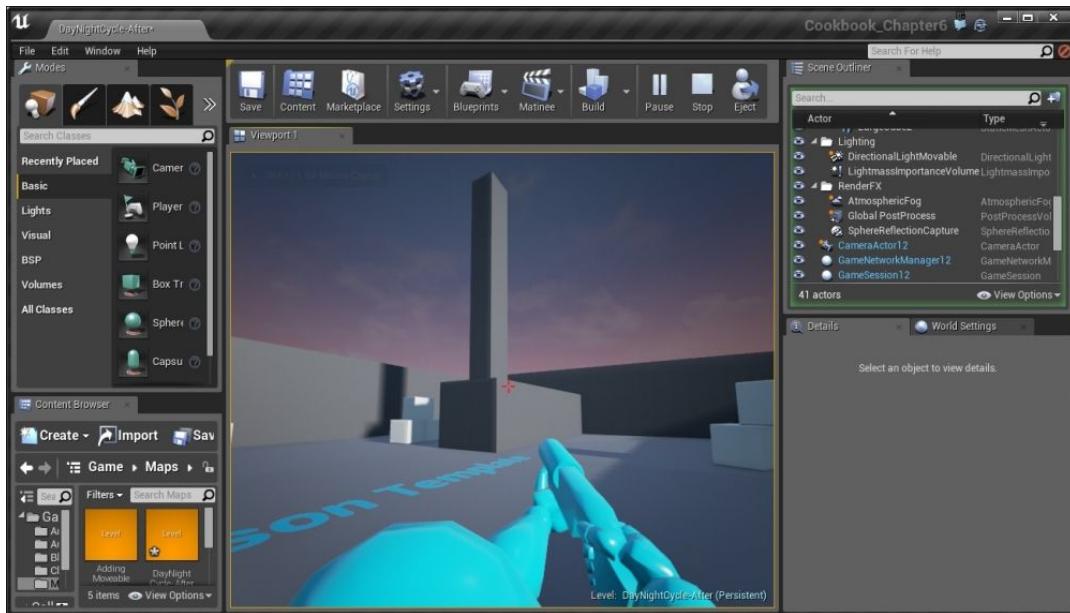
Taking that into account, let's use the **Delta Seconds** value in creating our **Delta Rotation**:

1. To the left of the **Delta Rotation** vector, right-click and create a **Make Rot** action (it makes a rotator from a **Pitch**, **Yaw**, and **Roll** value). Connect the **Return Value** from the **Make Rot** action to the **Delta Rotation** vector of the **Add Actor Local Rotation** action.
2. To the left of the **Make Rot** action, right-click and create a **Float \* Float** action, which we will use to multiply two floating point numbers together (in coding, the operator for multiplication is **\***). Connect the **Delta Seconds** value from **Event Tick** into one of the two inputs and connect the output to **Yaw** of the **Make Rot** action. Finally, put in a value of **10** or whatever modifier you want into the other input.



Using delta seconds within our rotation

3. At this point, click on the **Compile** button, exit the level blueprint, and run the game!



It works perfectly and we can modify this 10 value to make the value go faster (20 for 2x faster) or slower (1 for 10x slower), without having to worry about how complex the game is!

## See also

We just touched the tip of the iceberg when it comes to working in lighting! After all, we can't condense what someone's full time job is to 20 pages. However, I do have some external resources that may be useful should you decide to do more with it:

- While creating lighting for efficiency's sake, it's important to keep in mind what areas need to have the highest quality lighting and what can be reduced. To do that we make use of lightmass. For information on that, refer to <https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/Lightmass/Basics/index.html>.
- While it's not needed so much for games, there are a lot of people who want to have realistic lighting for their levels for things such as architectural demos. For more details on that, visit <https://docs.unrealengine.com/latest/INT/Resources>Showcases/RealisticRendering/index.html>.
  - For this, you can also download the maps shown from the Epic Games Launcher under the **Learn** tab if you select the **Realistic Rendering** option.
- In case you run into any problems with your lighting, a good guide to look into for troubleshooting can be found at <https://wiki.unrealengine.com/LightingTroubleshootingGuide>.
- You may also be interested in seeing the lighting of a map. 335Razor has a tutorial that recreates the lighting in one of the Unreal test maps at <https://www.youtube.com/watch?v=KmY7X9cBQZ0>.

## Chapter 7. Art Pipeline – Working with Materials

In this chapter, we'll cover the following recipes:

- Creating a custom material
- Creating a mirror material
- Using Textures and normal maps with Materials
- Creating glowing materials with static emissive lighting
- Seeing through walls

## Introduction

Artists are incredibly empowered by working in the Unreal Engine with many features created to make their work come out as aesthetically pleasing as possible. In this chapter, we are going to look at some of the ways that you can work within the art pipeline of Unreal Engine 4, with some additional resources at the end of the chapter with even more content.

## Creating a custom material

**Materials** are the building block of creating environments in Unreal and they are what we use to apply to all of the surfaces of our environment, similar to putting paint onto a wall in the real world.

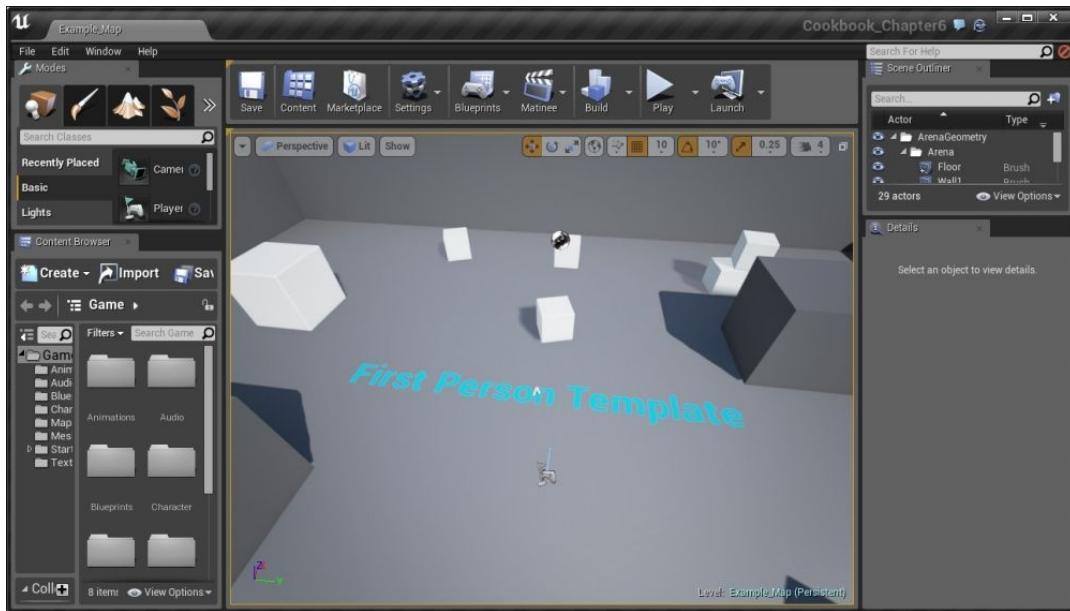
Specifically, it is a collection of textures and commands that will create a surface with properties such as shininess and color. We've used materials previously when we added them to walls when creating levels, but we've only been using the **Starter Content** folder that has been provided to us.

However, you may want to create a material of your very own. Let's look into that now.

### Getting ready

Before we start working within the Unreal Editor, we will need to have a project to work with:

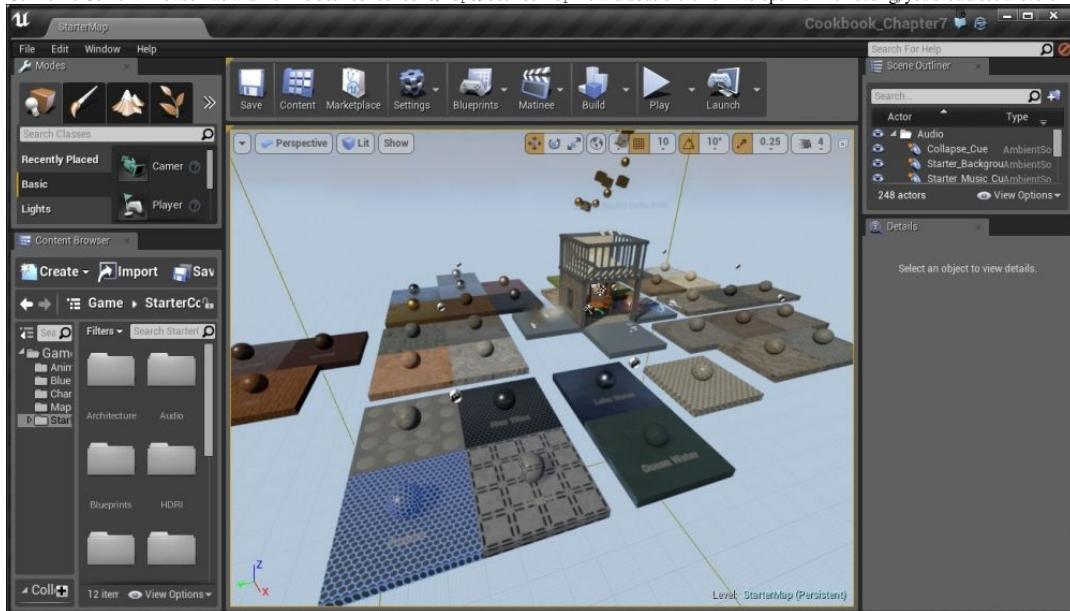
1. First, open up the Unreal Editor by clicking on the **Launch** button from the Unreal Engine Launcher.
2. Start a new project from the **Project Browser** tab by selecting the **New Project** tab. Select **First person** and make sure that **With Starter Content** is selected. Give the project a Name (**Cookbook\_Chapter7**). Once you are finished, click on **Create Project**.
3. You should see a level similar to this:



## How to do it...

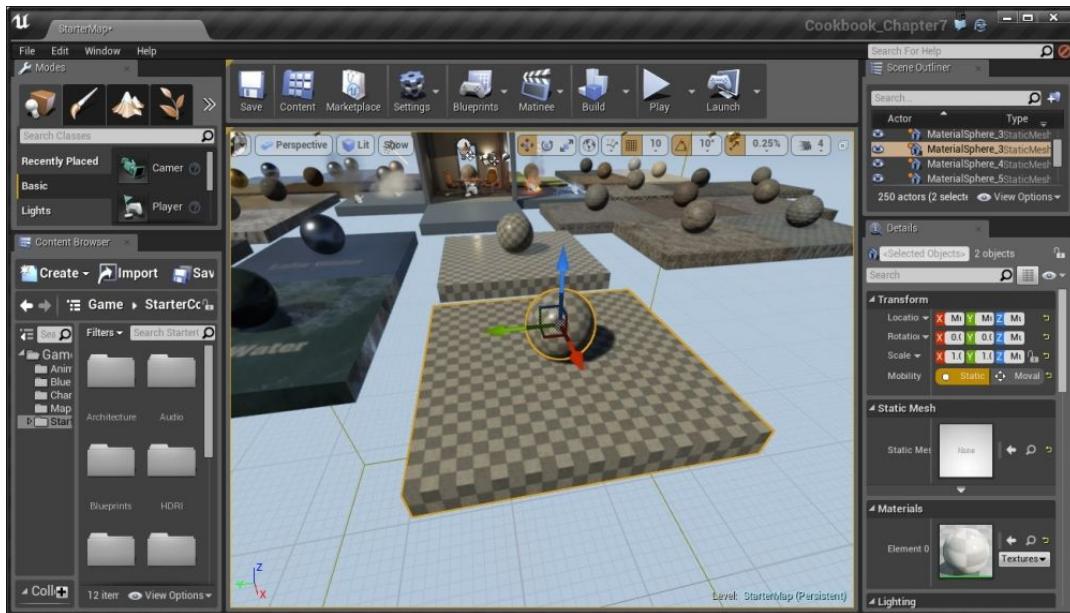
To start off with, we want to have an easy way to see what kind of material we are going to create. To do that, let's open a map that has a lot of examples for us to work with.

1. Go into the **Content Browser** tab and then the `StarterContent/Maps/StarterMap` file and double-click on it to open it. After loading, you should see a level similar to this:



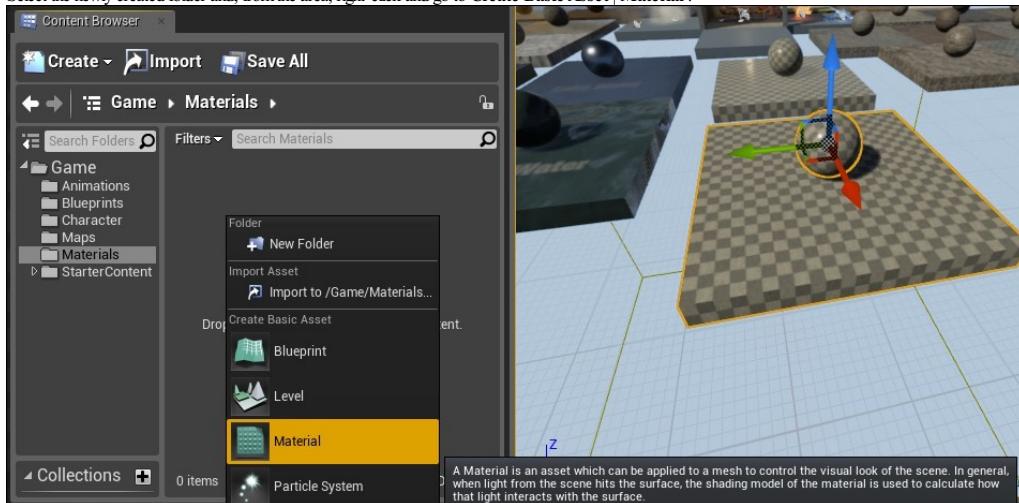
As you can see, all of the materials here are included currently in both a floor tile and circle object. This can be quite useful as you may want to create materials that work in all kinds of situations.

2. Let's make a copy of a base here for viewing our own material that we will create. Select the set of items with no material (the checkerboard pattern), hold down the *Alt* key, and drag it forward so that we have a new place to put our objects.

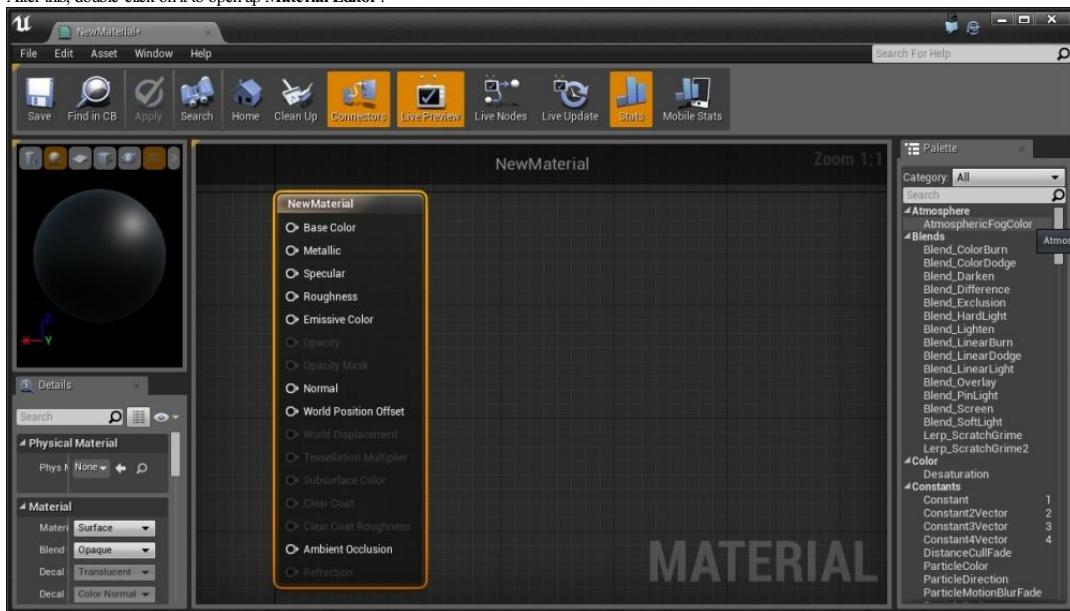


*Creating our duplicated base*

3. Now I want to differentiate my own content from the **Starter Content** folder, so from the **Game** folder in the **Content Browser** tab, right-click and select to create **New Folder** and name it **Materials**.
4. Select the newly created folder and, from the area, right-click and go to **Create Basic Asset | Material**.



5. After this, double-click on it to open up **Material Editor**.



Now just like normal in Unreal, all of the editors have a lot of content in them, so let's break it down.

First, in the center is the actual information that will be used for creating our material, and the entire area is for different operators and variables to modify these properties, much like working in Blueprints. The base material

node in the center is included with every material and has a series of inputs associated with a different aspect of the material.

The top-left is a preview window to allow you to see how the material you're creating looks similar to, and you can click on the buttons in the top-right to change the shape used for the preview to help you, depending on what the material will go on. You can use the mouse to move around, just like working with the editor, and you can also toggle the grid as well.

The bottom-left features the **Details** tab or properties for our element. We will use this in order to modify the various properties that each node will have.

To the right, you'll see **Palette** panel, which lists all of the possible nodes you can use to modify your material. However, you can also access this menu by right-clicking inside any empty gray area in the center and typing in the name of the item if you know what it is.

#### Note

For more information on the Material Editor UI check out <https://docs.unrealengine.com/latest/TNT/Engine/Rendering/Materials/Editor/Interface/index.html>

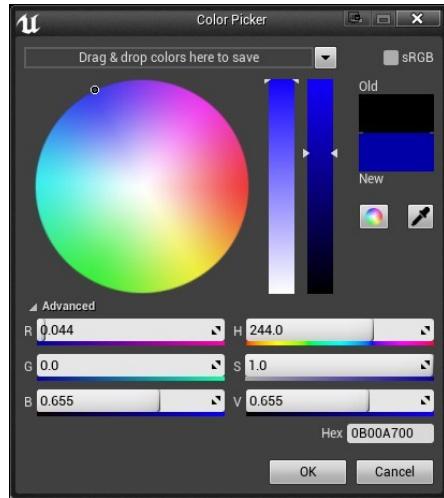
Now that we know what everything is, let's start to modify this material, starting with modifying the color:

1. First, you'll notice **Base Color**, which is similar to a diffuse map in 3D modeling programs. It basically is the color that will be displayed. We can either give it a texture to use, or we can use `Constant3Vector`, which is, to say, a number with three elements to assign the amount of red, green, and blue values that an item has. To do this, right-click in the empty space to the left of the **Base Color** to bring up the object list and type in `Constant3Vector` and then select this new object.

#### Tip

You can also hold down the 3 key and then click in the graph area to create a `Constant3Vector` node.

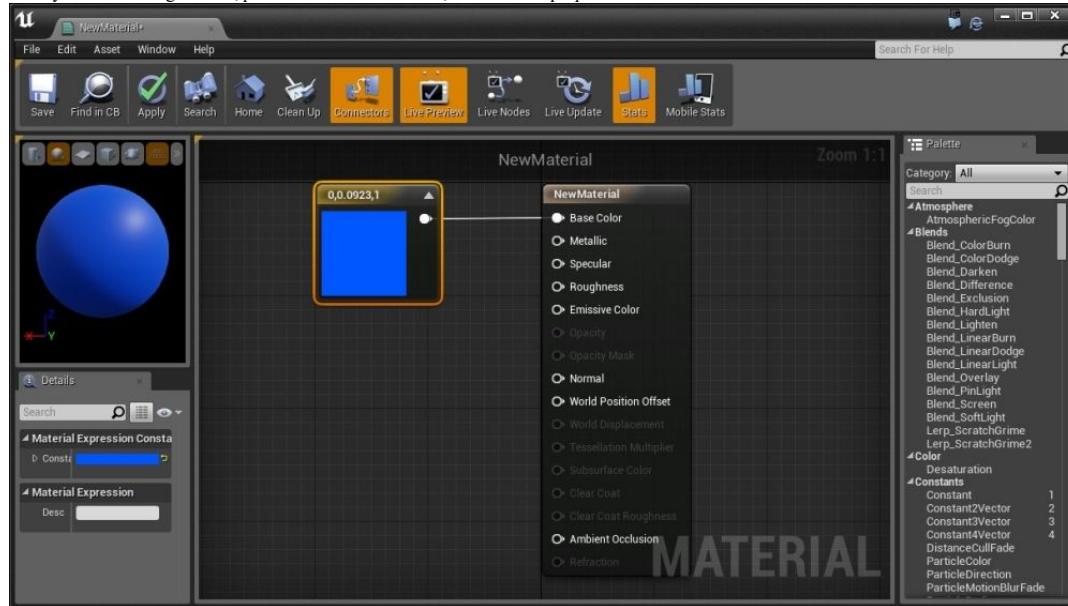
2. After this, from the **Details** tab, click on the box in front of the **Constant** section and change the color to something you'd like, making sure to bring up the black bar so that you can actually see something other than black.



#### Note

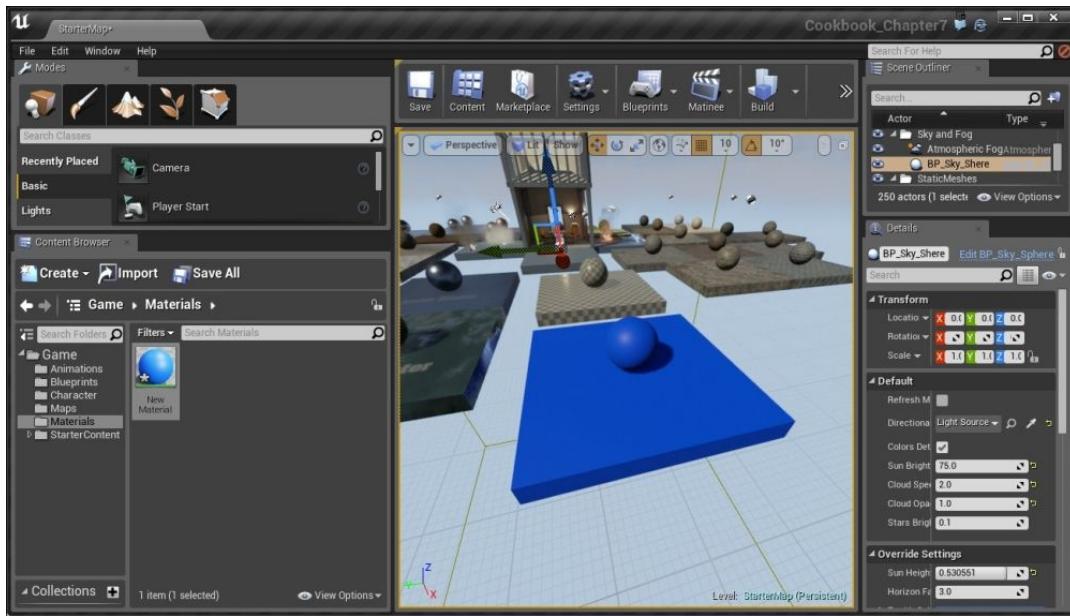
If you happen to know the **Hex** value of a color through something like Photoshop, you can also feel free to put that in. Alternatively, you can also use the **Color Picker** tool to pick a color as well.

3. When you're done setting the color, press the **OK** button. After this, connect the output pin from the `ConstantVector3` color into the **Base Color** value.



*Setting the color*

4. Now click on the **Apply** button on the top toolbar to commit those changes into the editor. Then click on **Save** and move back into the editor. From there, drag and drop the new material we have created onto both the objects.



*Applying the colored material to our base*

And with this, we've created a very simple colored material!

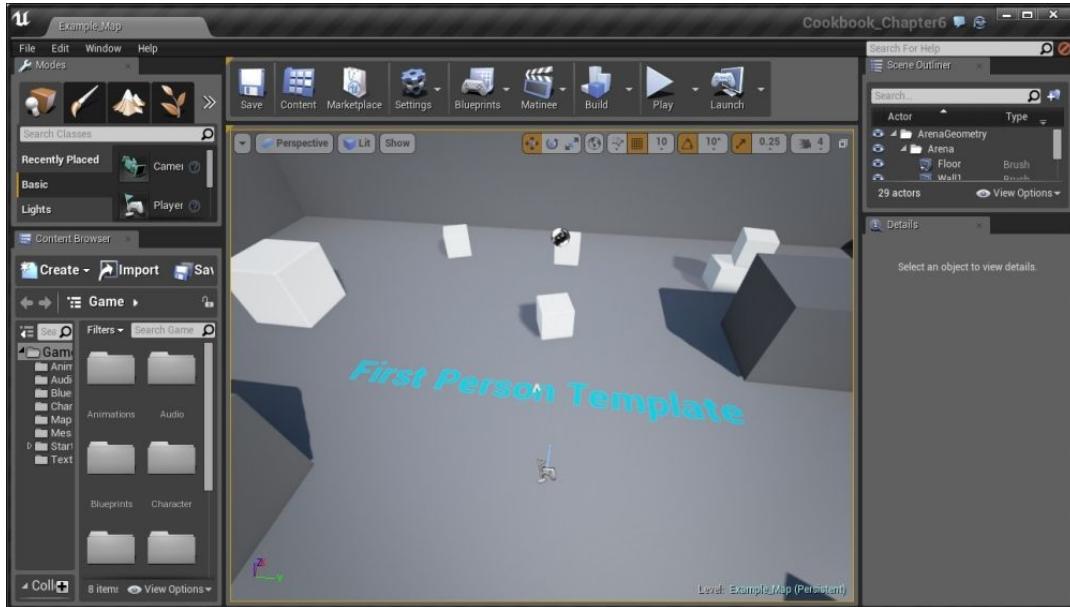
## Creating a mirror material

Now that we've created one of the simplest materials, let's get a little more complex by creating a mirror material while learning about some of the other properties the Material Editor has in the process!

### Getting ready

Before we start working within the Unreal Editor, we will need to have a project to work with:

1. First, open up the Unreal Editor by clicking on the **Launch** button from the Unreal Engine Launcher.
2. Start a new project from the **Project Browser** window by selecting the **New Project** tab. Select **First person** and make sure that **With Starter Content** is selected and give the project a Name (**Cookbook\_Chapter7**). Once you are done, click on **Create Project**.
3. You should see a level similar to this:



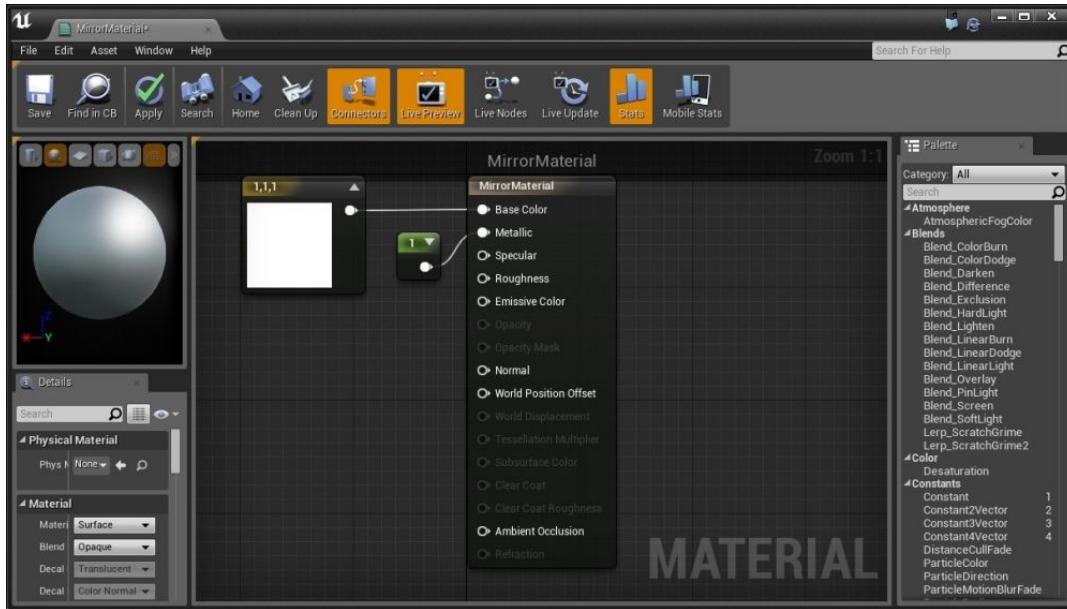
## How to do it...

To start creating a mirror material, we will perform similar steps to what we've done before:

1. Go in and create another material in the **Content Browser** tab by going into the **Materials** folder and right-clicking and selecting **Material**.
2. Rename the material to **Mirror Material** and double-click on it to enter into Material Editor.
3. Just like before, we want to give a color to our mirror. To do this, right-click to the left of the **Base Color** value and create **Constant3Vector** and give it a value of **1, 1, 1** (also known as white).
4. Next, we will want to modify the **Metallic** property, which will modify our material to be more metallic and shiny. Create a **constant** value by right-clicking and searching for **Constant** and set the value to **1**.

### Tip

Alternatively, you can also press and hold the **I** key and then click on the material graph to create a single scalar constant value.

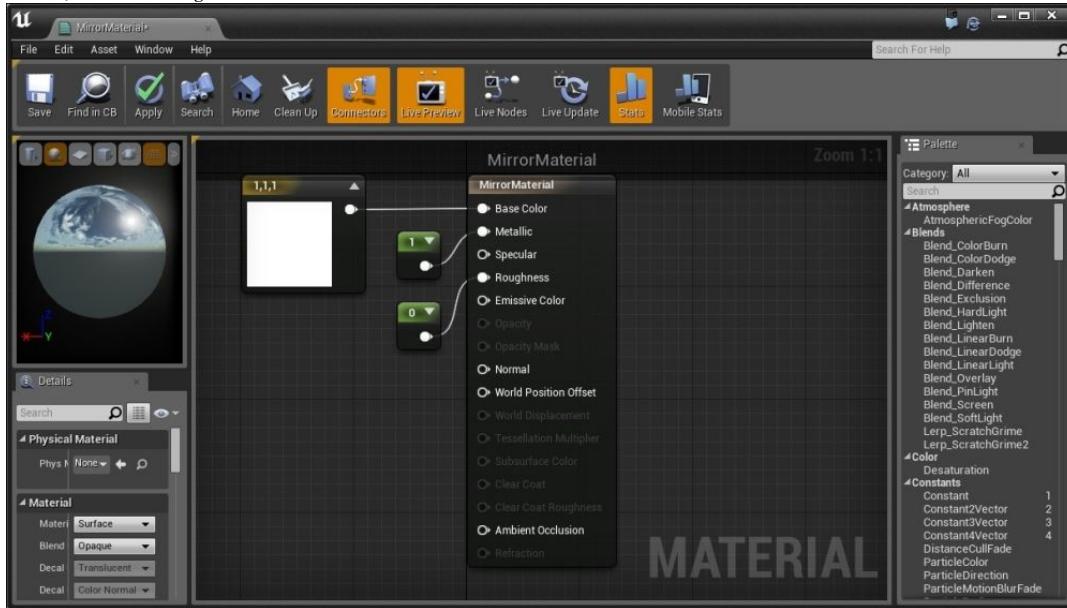


Notice how it makes the item much more shiny just as metals are in real life. When we set this value to 1, it means that it is a full metal object; we use 0 for non-metal objects and increase the number for more metallic things.

#### Note

For more information on the Metallic property, visit [https://docs.unrealengine.com/latest/INT/Resources/ContentExamples/MaterialNodes/1\\_2/index.html](https://docs.unrealengine.com/latest/INT/Resources/ContentExamples/MaterialNodes/1_2/index.html).

5. Next, we will modify **Roughness**; this will modify the material in such a way that it will look more rugged or shiny depending on what we set it to, and it just so happens that when things are shiny, they are reflective, so we will set **Roughness** to 0.

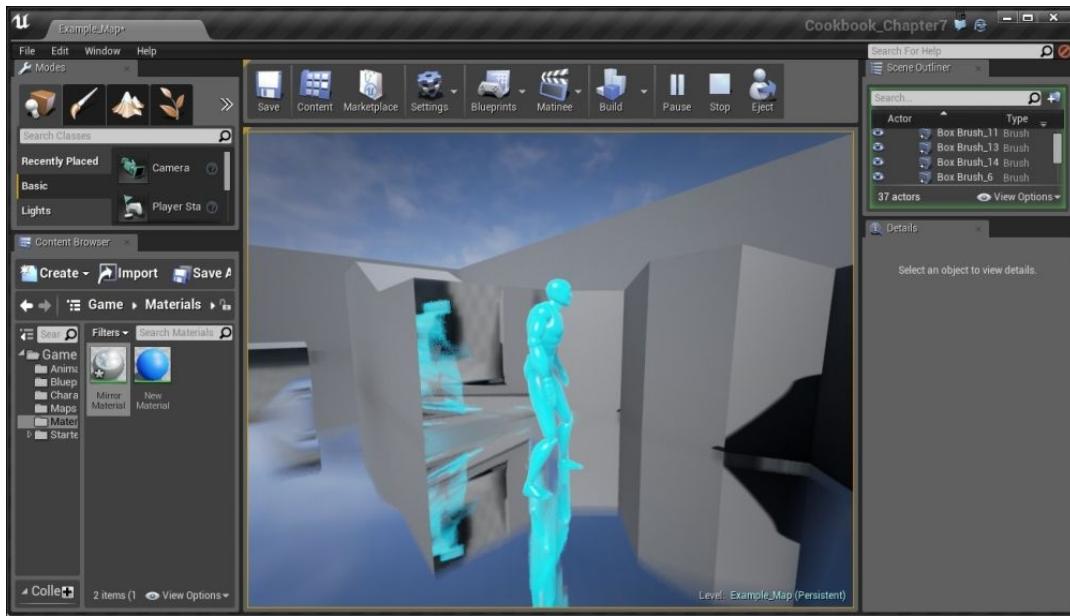


From here, you get to see a skybox being reflected in the background that is representative of what you'd see if you applied this material; everything around it will be reflective like a mirror. The higher the value is for roughness, the less reflective it will become.

#### Note

For more information on the Roughness property, visit [https://docs.unrealengine.com/latest/INT/Resources/ContentExamples/MaterialNodes/1\\_4/index.html](https://docs.unrealengine.com/latest/INT/Resources/ContentExamples/MaterialNodes/1_4/index.html).

6. With this, click on the **Apply** button and close the Editor.
7. Jump into the `Example_Map` level in the `Maps` folder and apply the newly created `Mirror Material` to some of the surfaces by dragging and dropping them onto the scene.



*Applying the mirror effect*

With this, we now have a new material that we can work with, with a really cool effect while also seeing how the **Roughness** and **Metallic** properties work!

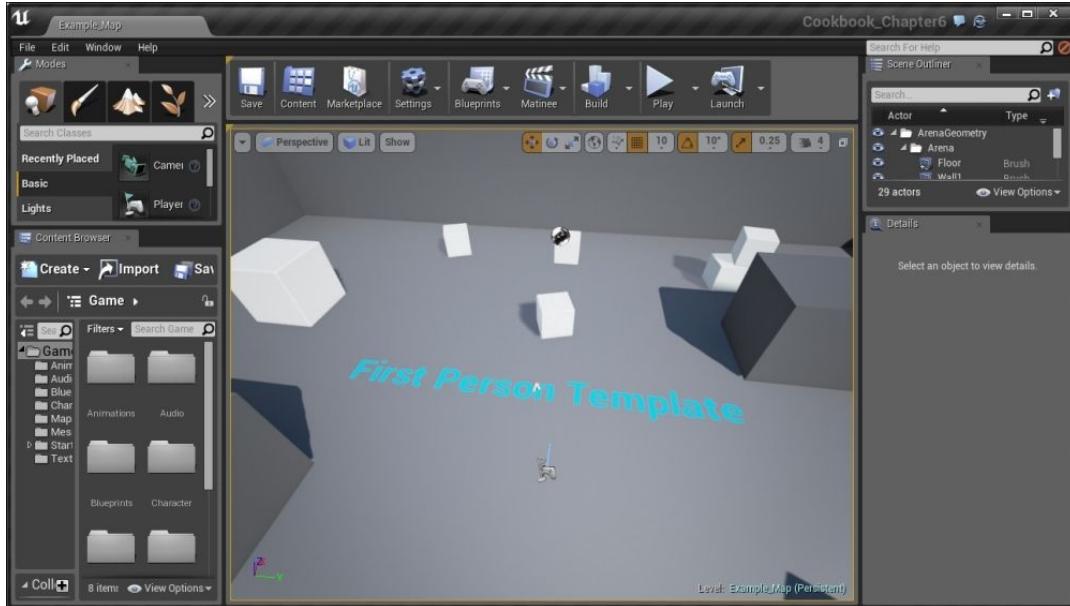
## Using Textures and normal maps with Materials

As time goes on, working with just colors isn't enough. You'll want to have more detailed materials that look more like things in the real world, such as wood, water, and walls. To do this, artists will often provide images that we can use for color data for the different parts of materials.

### Getting ready

Before we start working within the Unreal Editor, we will need to have a project to work with:

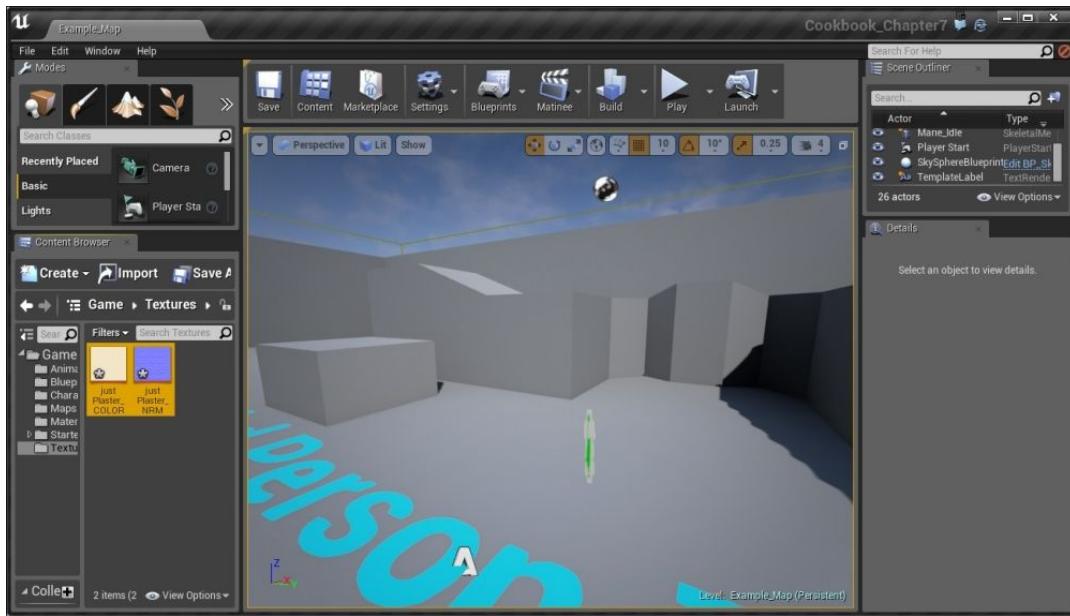
1. First, open up the Unreal Editor by clicking on the **Launch** button from the Unreal Engine Launcher.
2. Start a new project from the **Project Browser** tab by selecting the **New Project** tab. Select **First person** and make sure that **With Starter Content** is selected. Give the project a Name (**Cookbook\_Chapter7**). Once you are done, click on **Create Project**.
3. You should see a level similar to this:



### How to do it...

Let's see how we can do this now:

1. The first thing we are going to do is add textures for us to work with. So, from the **Content Browser** tab, right-click on the **Game** folder and select to create **New Folder**. From there, set the name as **Textures**.
2. Inside the **Example Code** folder for this chapter, open the **Images** folder and drag and drop the images into this folder.



You may get a notification saying that `JustPlaster_NRM` file was imported as a normal map. This is important as we'll be using it later, but first, let's get this information into a material.

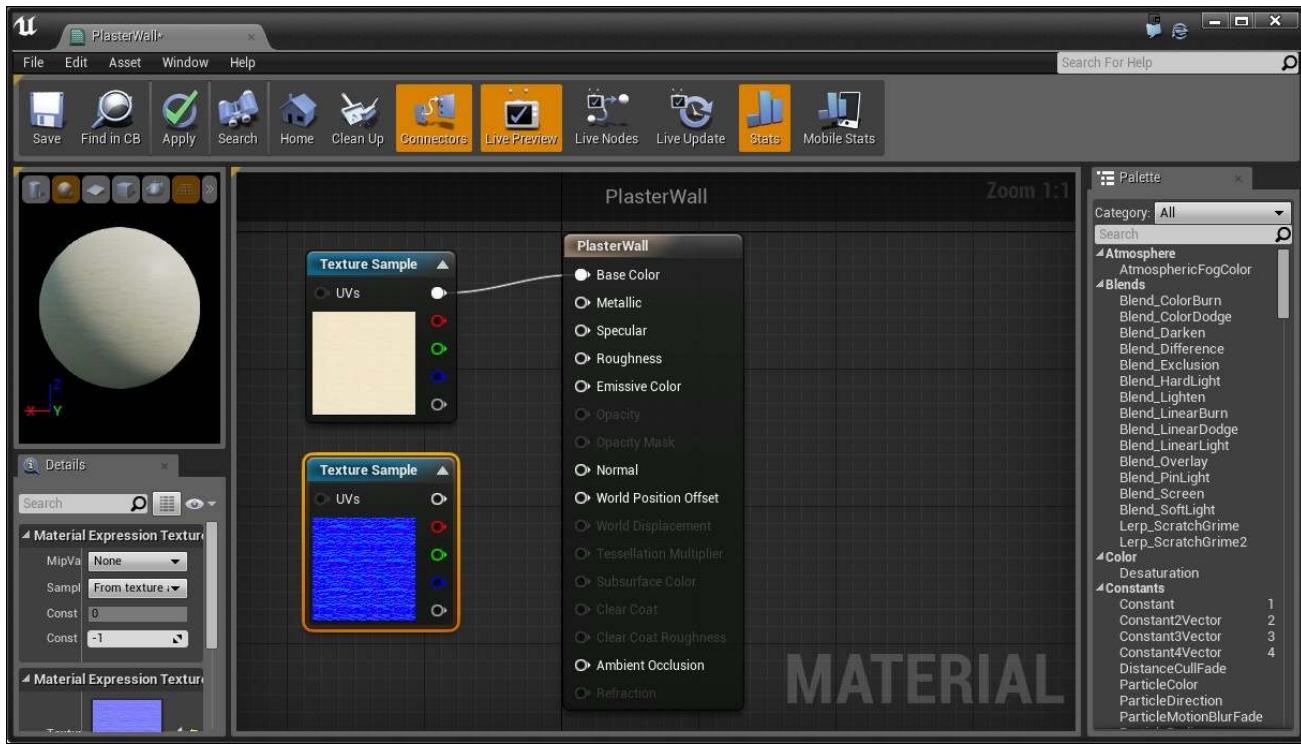
3. Go to our **Materials** folder and then right-click and create a new Material which we will call `PlasterWall`. Then double-click to open the Material Editor.
4. To add the texture's data as **Texture Sample**, we can simply drag and drop them into the editor and move them around so the blueish normal map is below the other texture.



### Tip

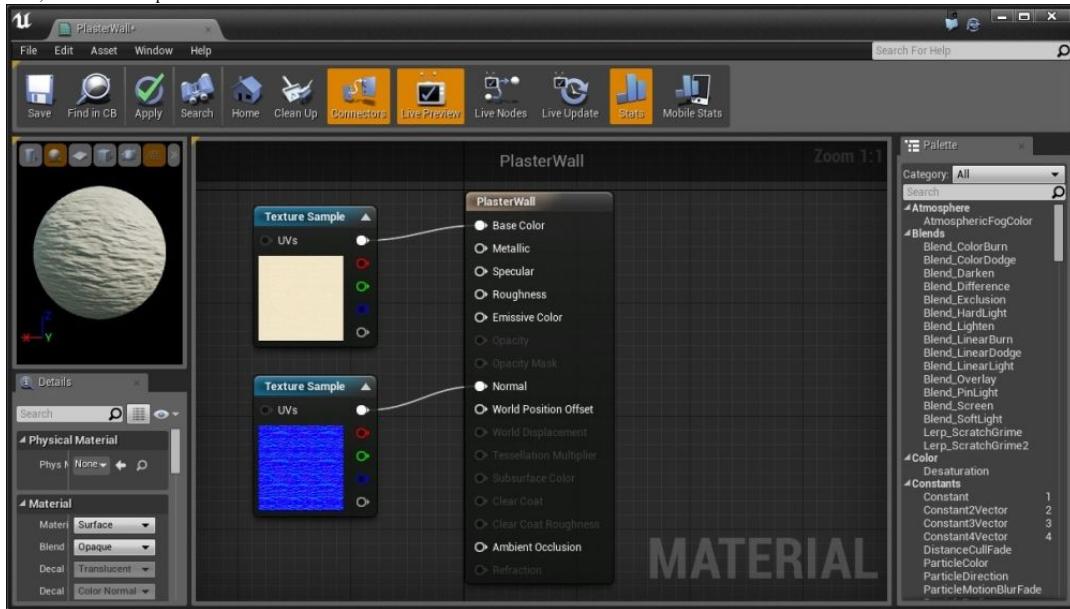
Another way to place a texture sample is by selecting the texture maps in the **Content Browser** tab and then in the material editor, pressing and holding the **T** key, and clicking on the **Material Editor** graph to create the texture sample node with a specific texture from the **Content Browser** tab.

5. Next, connect the color material's top white pin into **Base Color**.



You'll notice that it gives us more variation than just a simple color.

6. Next, we set the white pin of the bluish normal texture to **Normal** of the material.



You'll notice immediately that **Normal** has a big effect on how the material works. A normal map is used to give the illusion of lumps and bumps on a surface by simulating the pixels being at different areas.

This effect may be larger than what we are looking for and from the normal map's information, it seems like the green aspect of the map is making it look a lot more pronounced than it should be. A Normal is a vector which points in a particular direction to give each pixel an *angle* that the object is. That angle is defined by assigning the **X**, **Y**, and **Z** direction of the surface normal to the **R** (Red), **G** (Green), and **B** (Blue) channels of the texture, respectively. Thankfully, we can modify this using some additional operators.

#### Note

For more information on normal maps, visit [http://wiki.polycount.com/wiki/Normal\\_map](http://wiki.polycount.com/wiki/Normal_map).

1. Move the normal map's texture sample over to the left to make room for another node. Next, add a **Multiply** action by right clicking and searching for **Multiply**.
2. Then below the normal map, right-click and create a **Constant3Vector** and set its value to **0.1, 0.1, 1** in the **Details** tab.
3. Disconnect the normal from the **Normal** section of the material by holding down the **Alt** key and then clicking on it. Then connect it to the **A** part of the **Multiply** node.
4. Next, connect the white pin from the **Constant3Vector** we created to the **B** slot.
5. Finally, connect the white pin of **Multiply** to the **Normal** section of our material. It should look similar to the following image:

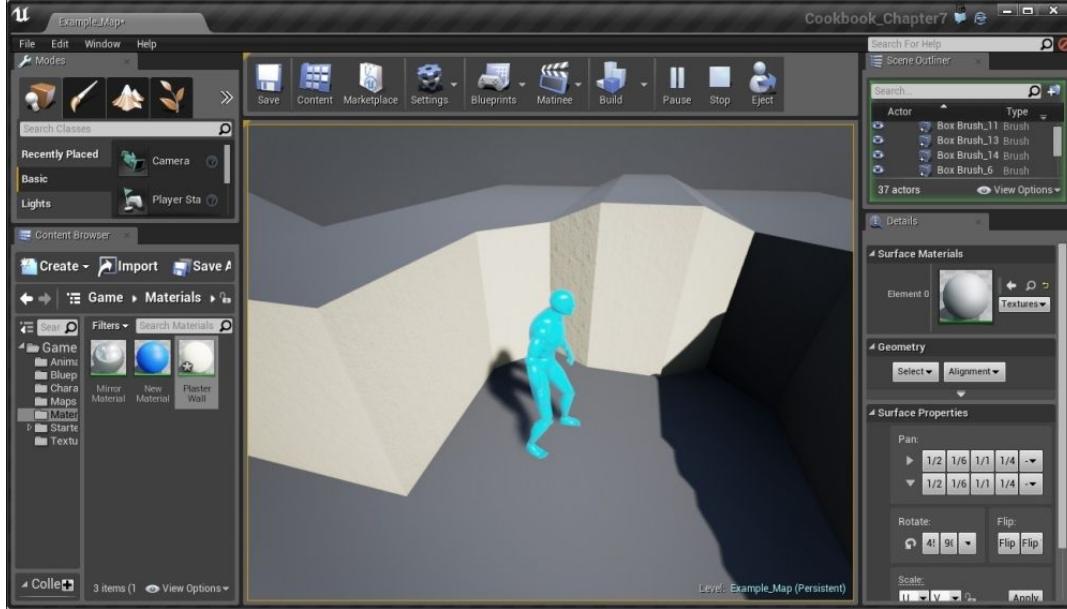


We multiplied each of the channels of the texture sample by a number we provided in our purple-looking item. The green channel is set to `.1`, which means that we are removing the harshness we were seeing previously by removing 90% of what was there previously.

#### Note

Note that this is not the same as the dot product or cross product, which are totally different things (but are the two ways that you can *multiply* vectors with vectors which have special mathematical properties that are often used in collision), but are very useful for game development if you're a programmer.

6. After this, click on **Apply** and then exit back into the Editor and apply our material to some walls.
7. Save our newly created content and then click on **Play** !



As you can see, there are now some differences between the areas of the walls, making it look much more realistic.

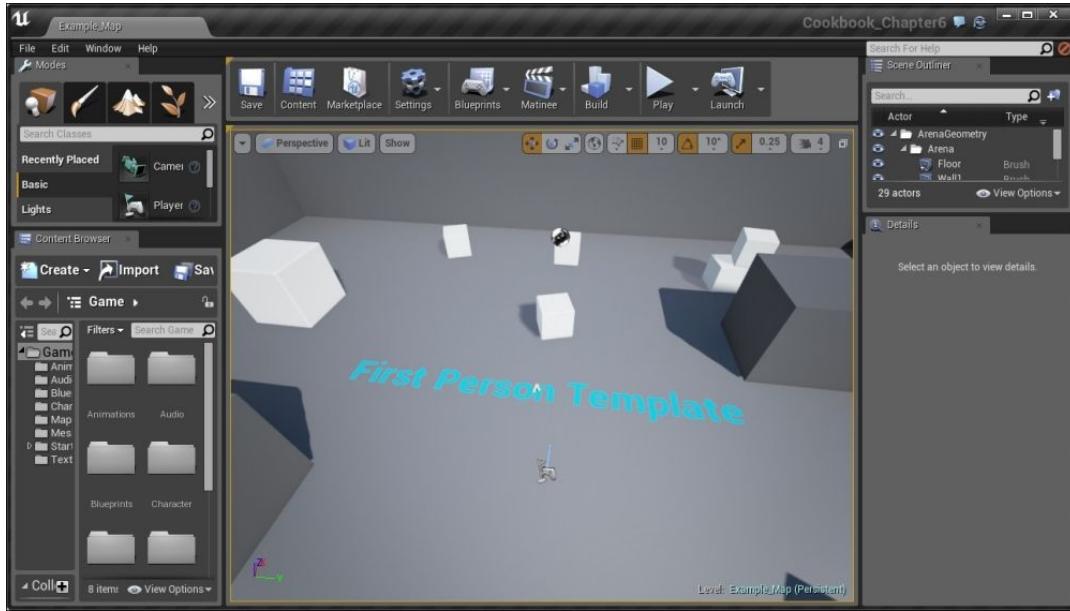
## Creating glowing materials with static emissive lighting

If you've ever seen a movie such as *Tron: Legacy* or any game area with neon lights and the like, it may seem very computationally expensive to have that in our levels because of how they have a lot of lighting going on. We can add this efficiently to our materials by using the emissive property.

#### Getting ready

Before we start working within the Unreal Editor, we will need to have a project to work with:

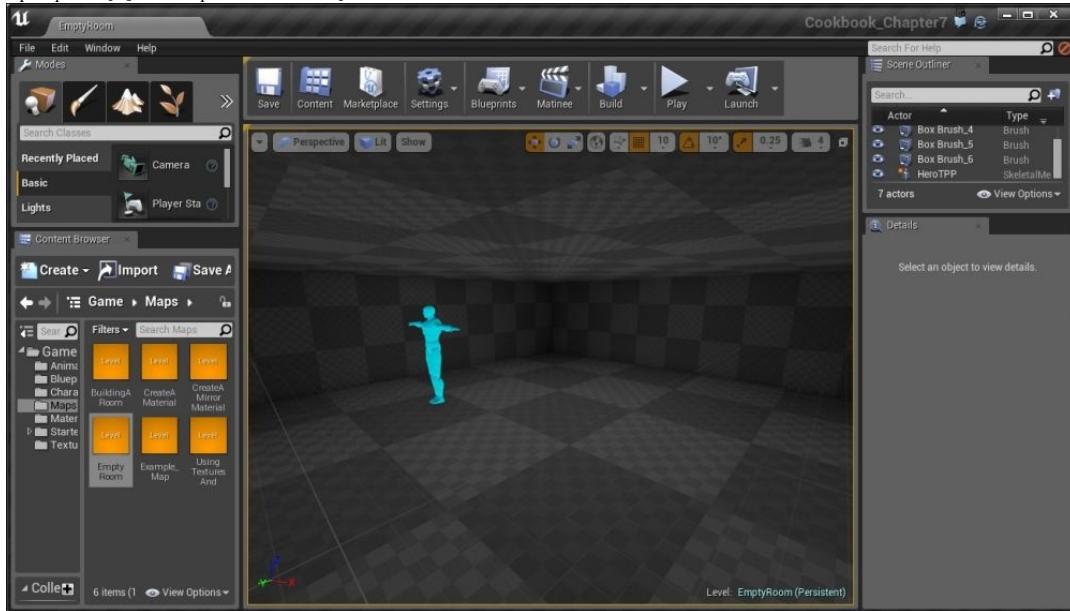
1. First open up the Unreal Editor by clicking on the **Launch** button from the Unreal Engine Launcher.
2. Start a new project from the **Project Browser** tab by selecting the **New Project** tab. Select **First person** and make sure that **With Starter Content** is selected. Give the project a Name (`Cookbook_Chapter7`). Once you are done, click on **Create Project**.
3. You should see a level similar to this:



## How to do it...

The first thing we are going to want to do is open up a level that is made for this kind of lighting.

1. Open up the `Empty Room` map created in the `Example Code` folder.



*The Empty Room map*

This map is the same as the map created in our previous *Building a room* recipe in [Chapter 2](#), *Level Design – Building Out Levels or Greyboxing* aside from removing the directional and point light created in there. Needless to say, if lighting was built and you played the game, it would be in pitch blackness.

2. Let's create a new material which will glow for us. To do that, right click on the `Materials` folder and select to create a new material. Name it `GlowMaterial` and then double-click on it to access the Material Editor.

3. First, let's set **Shading Model** to **Unlit**. This will make it so that it will only output the emissive value for color, which is perfect for this particular application.

### Note

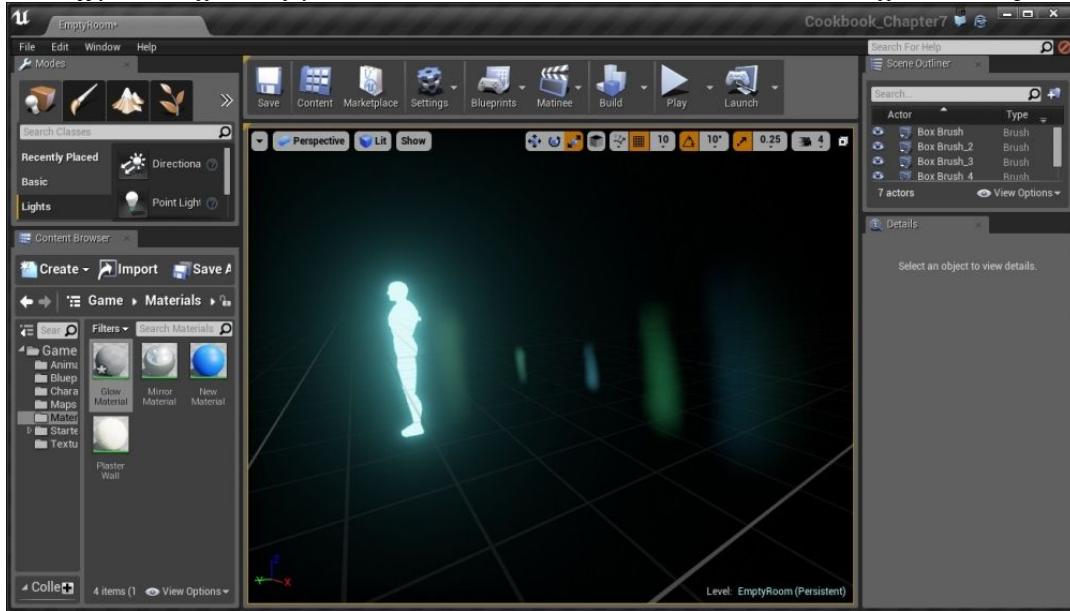
For more information on Shading Models, refer to <https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/MaterialProperties/LightingModels/index.html>.

4. Next, let's create a color to represent the color we want to have glowing. To do this, right-click on the left-hand side of the **Emissive Color** section and select `Constant3Vector` and change the color to something you like.
5. After this, in order for **Emissive** to give us a bloom color, we need to make the value larger than 1. We can also modify the color itself using actions. Let's add in a **Multiply** action to the right of the color.
6. Connect the white pin of the color to the **A** value of the **Multiply(10)** action. Then, connect the white pin at the end of the **Multiply(10)** action to the **Emissive Color** input.
7. Next, select the **Multiply** action and rather than creating a new node for the **B** value, just go to the **Details** tab and put in a value in the **Const B** value, such as **10**. If all goes well, you should see a nice glow appear, as shown here:

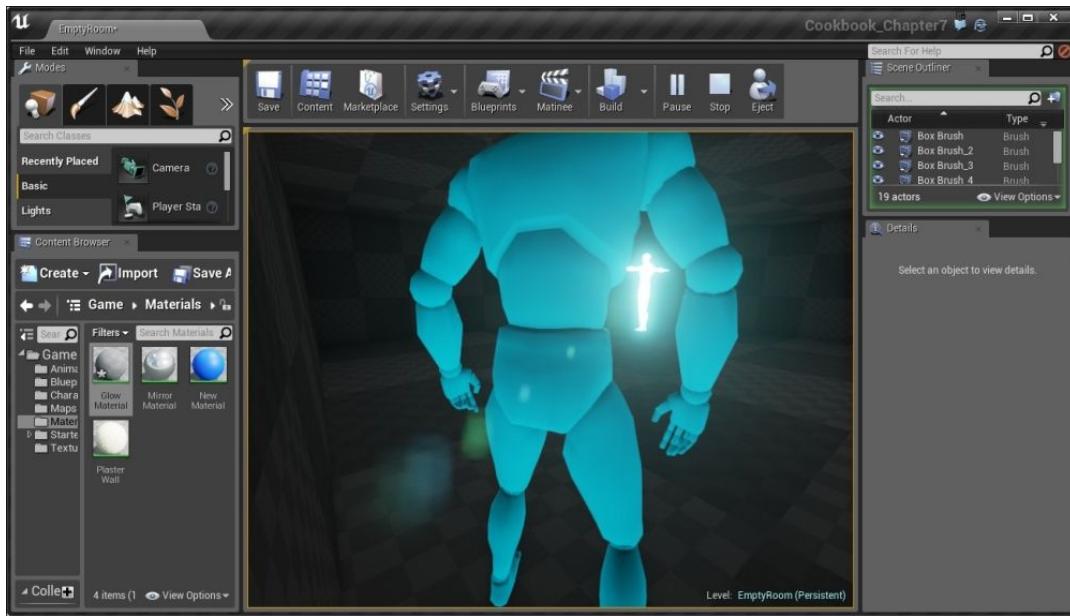


*Creating a glowing material*

- Click on **Apply** to have this applied to our players. Now, set the material to our character in the screen and after a few seconds, it will appear to have a nice glow effect.



- Finally, let's play the game and see it in action.



## Note

It's important to note that this does not actually cast a light into the scene, but merely provides a glowing effect.

There are some in-progress tools that use a light propagation volume to add lighting. You can check these at <https://forums.unrealengine.com/showthread.php?6914-Light-Propagation-Volume-now-works-with-Emissive-Material>.

For more information on the Emissive property, refer to [https://docs.unrealengine.com/latest/INT/Resources/ContentExamples/MaterialNodes/1\\_5/index.html](https://docs.unrealengine.com/latest/INT/Resources/ContentExamples/MaterialNodes/1_5/index.html).

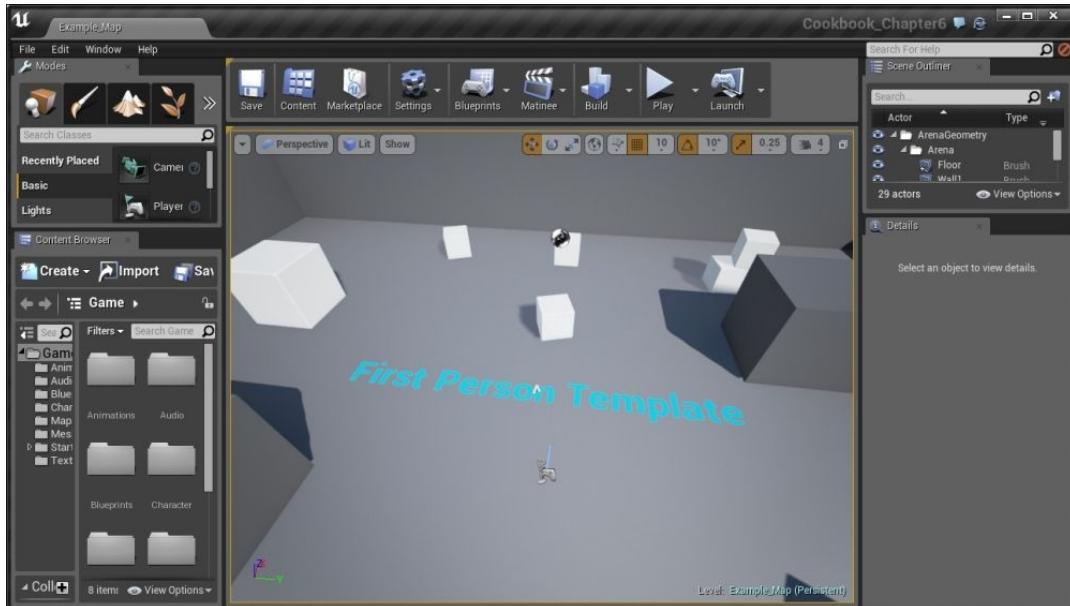
## Seeing through walls

A neat feature of many titles is being able to see enemies or important objects through walls, similar to the detective mode in *Batman: Arkham Knight*, the thermal vision in *Splinter Cell*, or the outlines in *Evolve* and other games. Let's implement similar functionality now.

### Getting ready

Before we start working within the Unreal Editor we will need to have a project to work with:

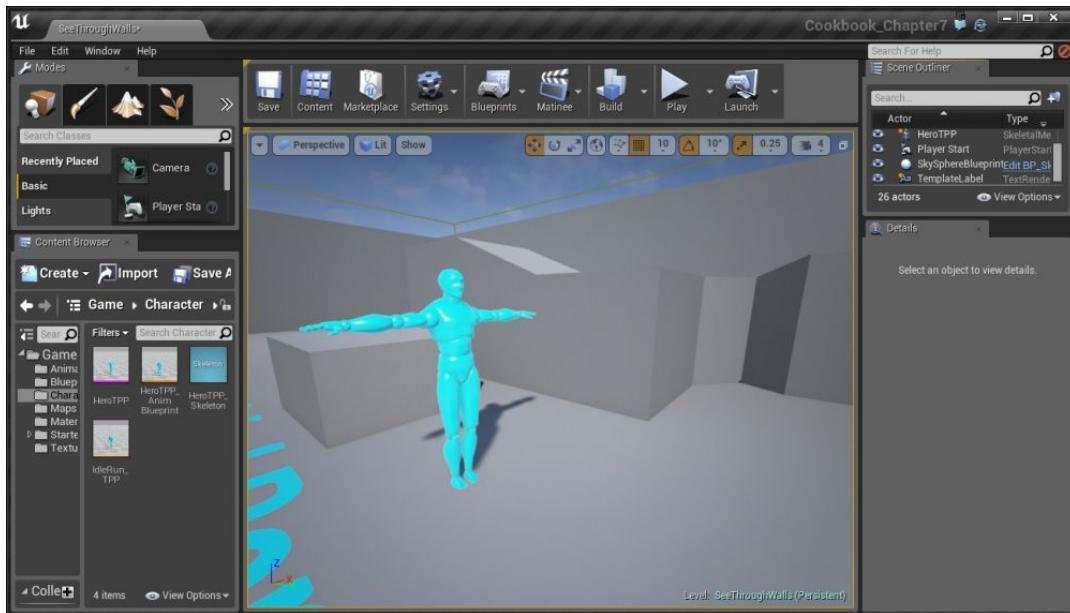
1. First, open up the Unreal Editor by clicking on the **Launch** button from the Unreal Engine Launcher.
2. Start a new project from the **Project Browser** tab by selecting the **New Project** tab. Select **First person** and make sure that **With Starter Content** is selected. Give the project a Name (Cookbook\_Chapter7). Once you are done, click on **Create Project**.
3. You should see a level similar to this:



## How to do it...

In order to see other objects, let's first add an object that we'd like to see into the level:

1. Open up the `Example_Map` file and from the **Content Browser** tab, go into the `Character` folder and drag a `HeroTPP` skeletal mesh into the level. This will be our enemy that we will want to see no matter where they will be.



*Adding the enemy*

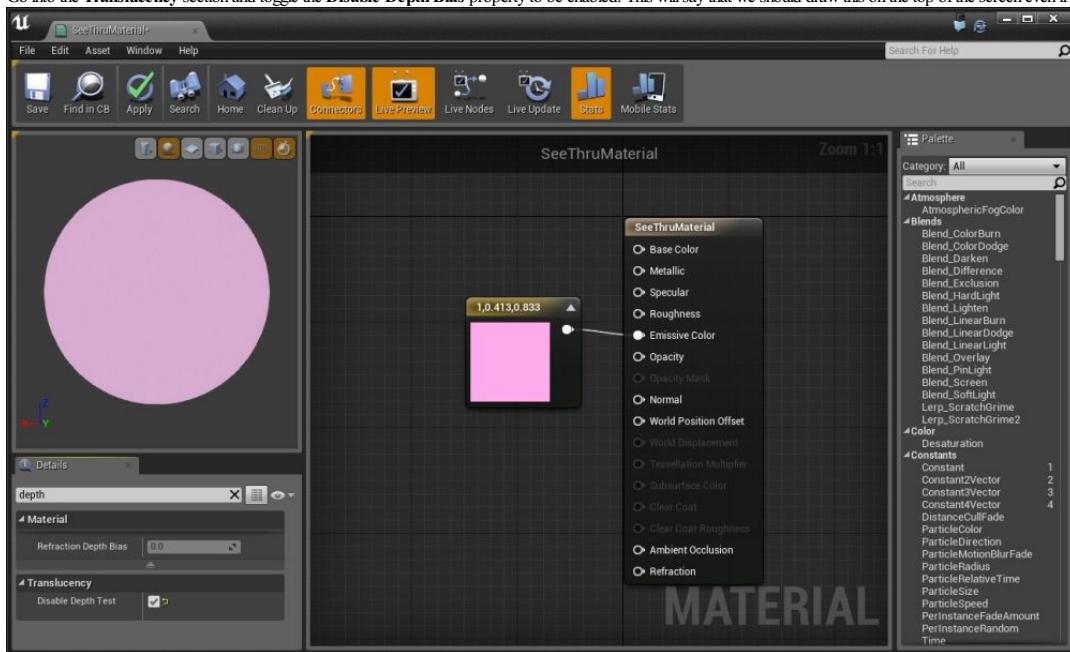
2. Let's now create a new material that we can work with and give it a name (`SeeThruMaterial`). Once it's created, open up the Material Editor.
3. First, create `Constant3Vector` with a color of your choice and assign it to the `Emissive Color` property.
4. After this, deselect it so that nothing is selected and go to the `Details` tab. From here, change the `Blend Mode` to `Translucent`.

This blending mode is used for objects that require some form of translucency and is applied onto areas in front of it, but this mode also has another feature that is perfect for what we want to do. We will look into it next.

#### Note

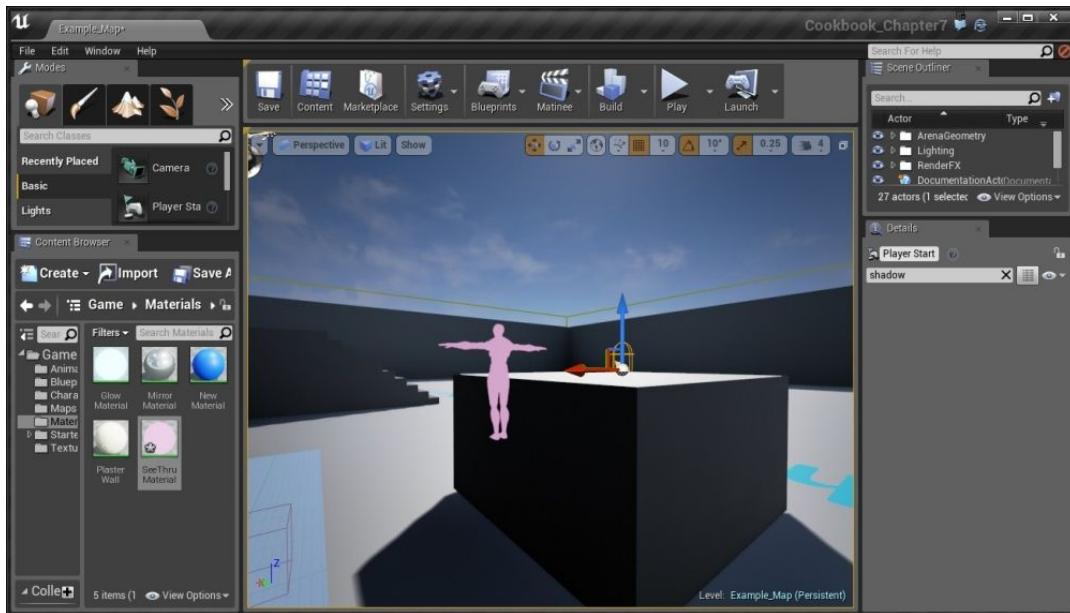
For more information on how to use Transparency effectively, refer to <https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/HowTo/Transparency/index.htm>.

5. Go into the `Translucency` section and toggle the `Disable Depth Bias` property to be enabled. This will say that we should draw this on the top of the screen even if we're behind something else.



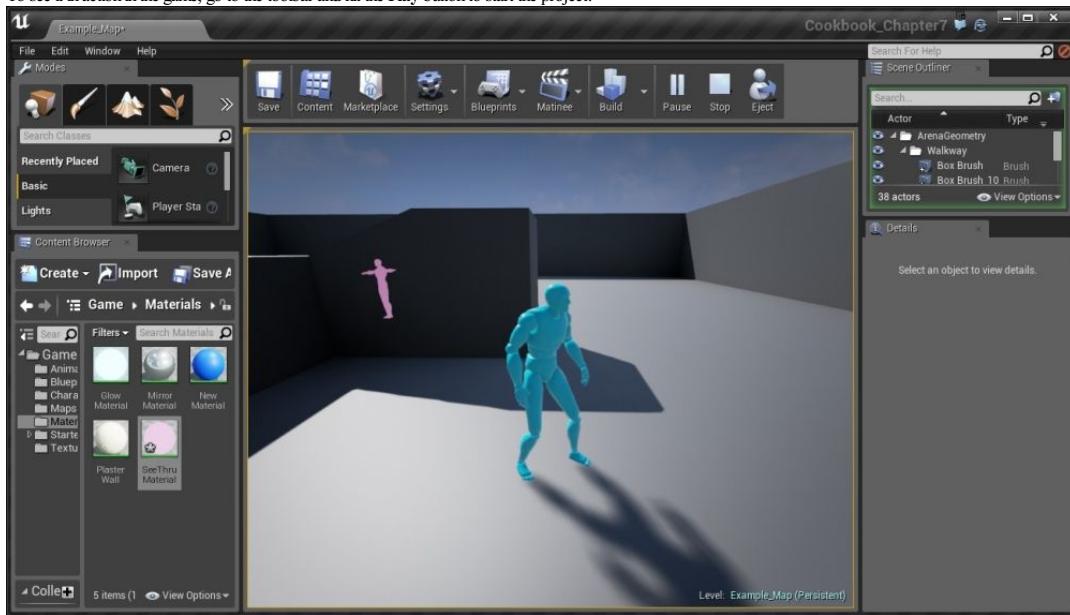
*Setting the translucence value*

6. Once you're done, hit the `Apply` button and then close the editor. Then, apply the newly created material to the enemy and move around the camera to see the effect that we generated!



Applying translucence to the enemy

- To see it in action in the game, go to the toolbar and hit the Play button to start the project!



With this, we can see the object no matter what! Later on, you can use this recipe with the knowledge of blueprints to toggle the effect.

#### Note

Tom Looman also came up with another implementation using the Render Custom Depth flag that looks similar to the game *Evolve* with outlines. It can be found at <http://www.tomlooman.com/ue4-evolves-outline-post-effect/>.

#### See also

Now, there is much more that exists in the Material editor, especially when you add more complex interactions and/or apply them to meshes to create some interesting effects. Here are some additional tutorials that may be useful to you going further:

- Much like working with brushes, you can also modify the UVs of the materials. Learn how to do this at <https://www.youtube.com/watch?v=YcoHWSxFEk0>.
- You can use a process called instancing using parameters to change the material as the game continues. There's a good tutorial on this at <https://www.youtube.com/watch?v=NX-NNvGV3oQ>.
- Sjoerd De Jong (also known as Hourenses) also has some amazing resources for you to look at, with an Example Project of his with materials and meshes that you can take apart, some videos of the project. I originally started learning Unreal Engine 3 years ago from his tutorials, and I'm really glad to hear that he's still creating awesome stuff. Check it out at <http://www.hourenses.com/thesolusproject/>.
- While this book is focused on using Unreal Engine 4, it may be useful to see a brief tutorial talking about building a level from scratch using a 3D modeling program and bringing it into the engine. A fairly good example tutorial series by The 3D Tutor on the subject can be found at <https://www.youtube.com/watch?v=cJ0z6GlgQ&list=PLK3nRt7ToxJokxAocxwIKvpM1f3X5JB2>.
- Once you feel comfortable creating levels, you may also want to build out a character with animations. An introduction to UE4's animation tools can be found at [https://www.youtube.com/watch?v=knbZ\\_g8Hgk&list=PLZlv\\_N0\\_O1gb2ZoKzTaphv3LvhaxJ9eg](https://www.youtube.com/watch?v=knbZ_g8Hgk&list=PLZlv_N0_O1gb2ZoKzTaphv3LvhaxJ9eg).

## Chapter 8. Blueprint Scripting – Level Effects

In this chapter, we'll cover the following recipes:

- Building a flickering light
- Converting from Level to Class Blueprints
- Using Trigger Volumes – opening a door using Matinee
- Adding to an existing Blueprint – flashlight, part 2
- Creating a Health/Damage system, part 1 – taking damage

## Introduction

Introduced in Unreal Engine 4, **Blueprints** is a visual scripting language that is built into the engine. By using visual scripting, instead of writing code from scratch inside Visual Studio or some other IDE (which we will cover in the next chapter), we can use predefined actions and connect them together, similar to drawing a graph. This is often a great starting point for artists and game designers as it is much more visually oriented and easier to grasp than just plain code.

There are two types of blueprints you can create:

- **Level Blueprint** : This works similarly to how UE3's Kismet system worked as events and actions that will occur in just this particular level. This is good for things such as triggering enemies to spawn or moving platforms.
- **Class Blueprint** : Introduced in UE4, this can be put into any level. They just work using their predefined behavior that we create beforehand, similar to prefabs in Unity.

It may take some time to get used to it, but we will dive into some simple systems first.

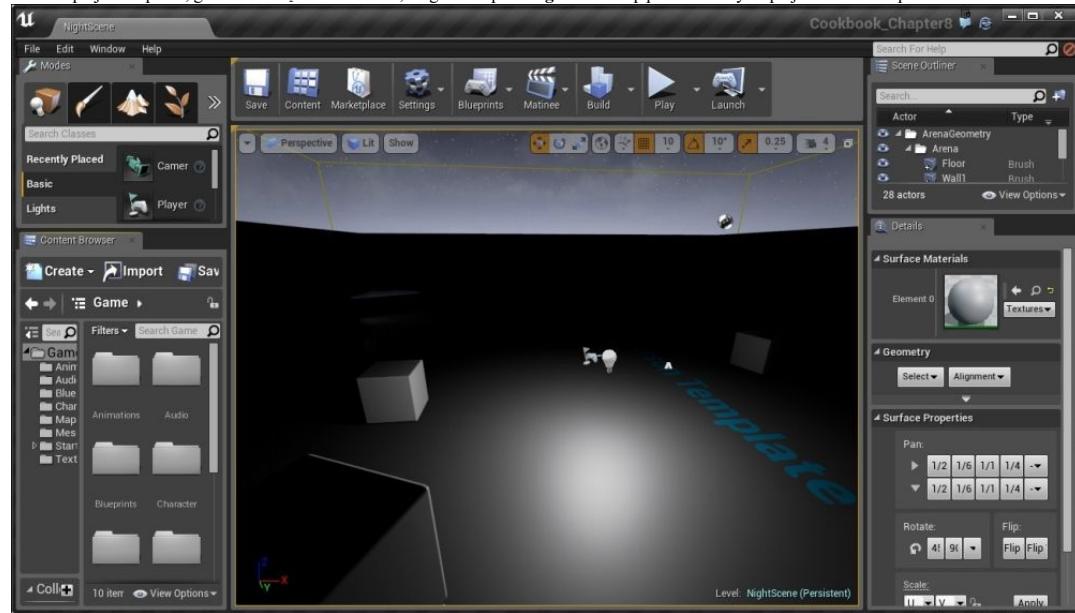
## Building a flickering light

For our first Blueprint, let's make use of the Level Blueprint system to create a simple action of a flickering light. Often a staple of horror games, lights that flicker can cause fear in players of not being able to see certain things for a random period of time.

### Getting ready

Before we start working within the Unreal Editor, we will need to have a project to work with:

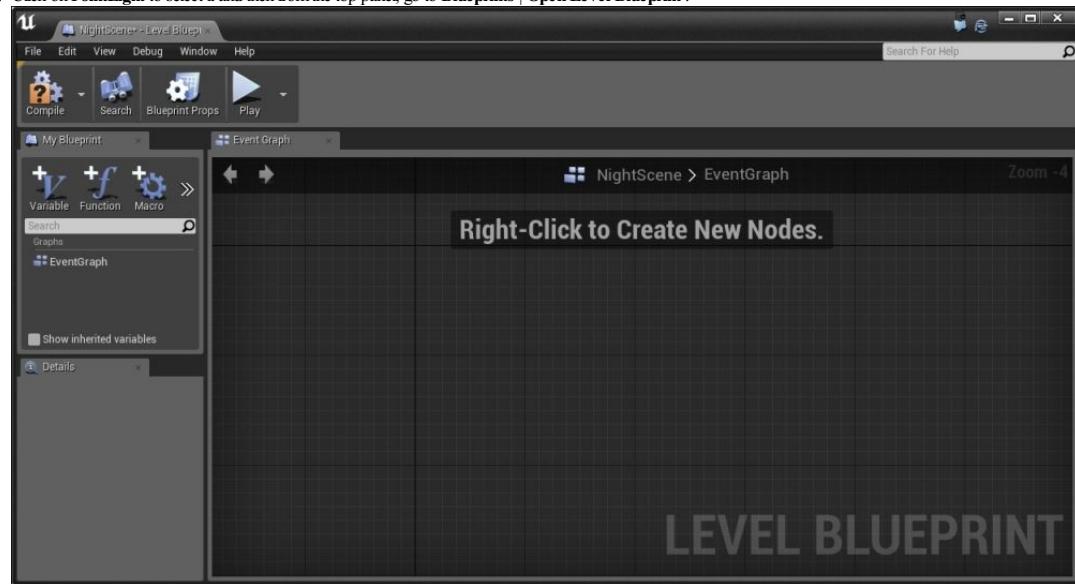
1. Open up the Unreal Editor by clicking on the **Launch** button from the Unreal Engine Launcher.
2. Start a new project from the **Project Browser** tab by selecting the **New Project** tab. Select **First person** and make sure that **With Starter Content** is selected and give the project a Name (**Cookbook\_Chapter8**). Once you are finished, click on **Create Project**.
3. After the project is opened, go to the **Example Code** folder, drag and drop the **NightScene** map provided into your project folder and open it.



### How to do it...

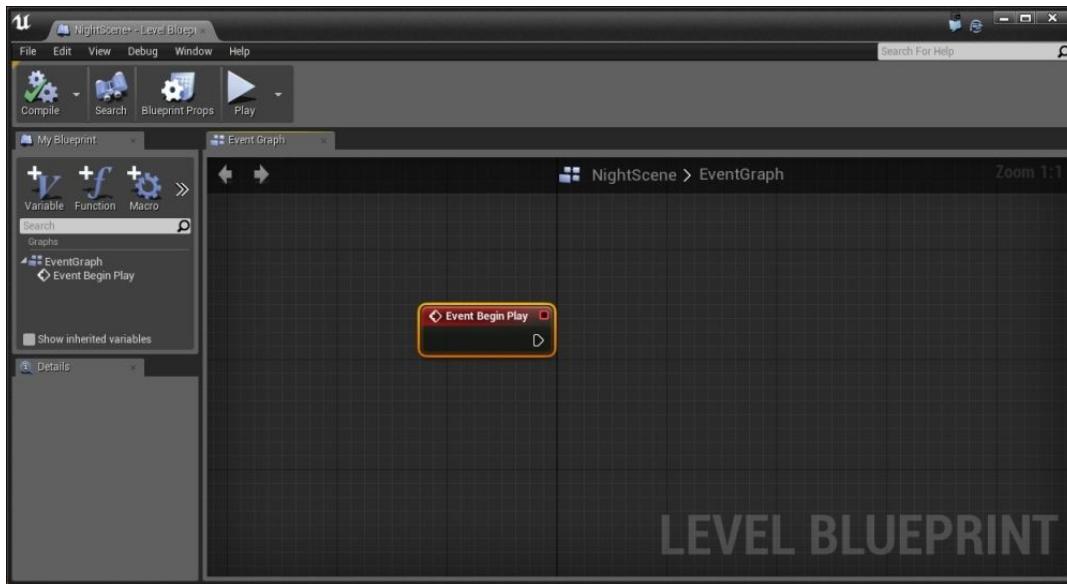
In the middle of the map, you'll notice that there is a light. We're going to make this randomly start to flicker for a time:

1. For a light to change and work correctly with lighting, the first thing we will need to do is change its **Mobility** to **Movable** on the far right. While this may imply the object will move, this is also a way of saying we want the ability to make properties (such as the Light's Intensity) modifiable while the game is running.
2. After this, move down and under **Light**, notice the value of the **Intensity** property. Note that when modified, it makes the light brighter or darker.
3. Click on **PointLight** to select it and then from the top panel, go to **Blueprints | Open Level Blueprint**.



This panel consists of a number of parts that will look familiar to those who have worked with UE3's Kismet system. The **EventGraph** on the right-hand side will contain the **events** and **functions** (which I'll sometimes also refer to as **actions**), which are called inside the game. Think of this in terms of when something happens (the events), we will do something (the functions).

4. From here, when we right-click within the main graph area in the center of the **Event Graph** tab, we will see a context menu pop up. In the search bar, search for **Begin Play** and select the **Event Begin Play** event. This event occurs whenever the game starts, so as soon as the player is spawned, whatever functions are called from it, will happen.

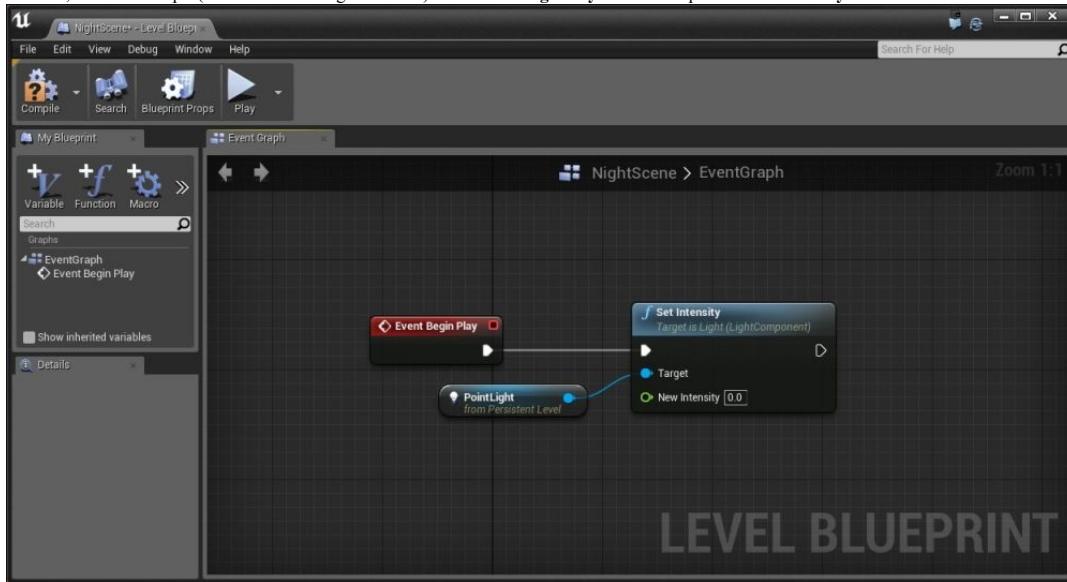


*Adding the Event Begin Play event to the level blueprint*

5. Next, we want to modify the light. To do this, we will right-click and select **Create a Reference to PointLight**.

This will create a variable that is a reference to the object in the scene. We can use this variable in the function's parameters.

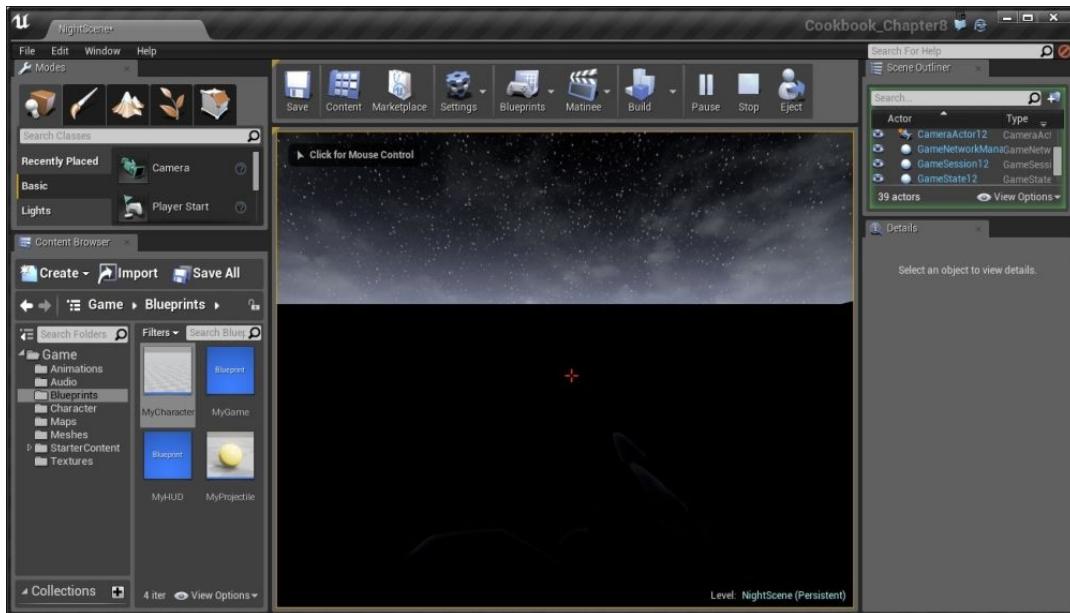
6. Click and drag the blue wire from the right-side of the point light and drag out to place a new node. The menu that pops up will only show the actions, which can use the point light. From here, type in **Intensity** and select the **Set Intensity (Light Component)** function.
7. After this, connect the output (white arrow on the right-hand side) of the **Event Begin Play** node to the input of the **Set Intensity** function.



*Connecting the output of the Event Begin Play node to the input of the Set Intensity function*

You'll notice that the function (note the fancy f icon on the left-hand side) has two properties, the **Target** and the **New Intensity**. This will change the **PointLight** (**Target**) to whatever the value on **New Intensity** is.

8. Click on the **Compile** button to commit our changes and then, back in the editor, hit the **Play** button.



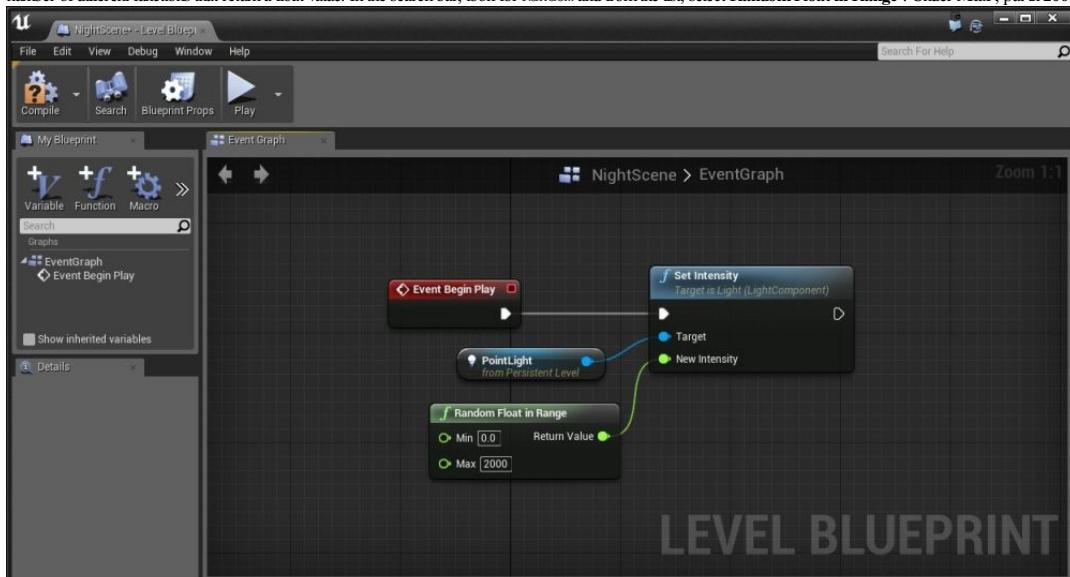
*Starting the game after compiling*

You'll notice that now the world is pitch black. This is because the value of **Set Intensity** is now set to **0**. If you have the **Level Blueprint** window open and then hit the **Play** button again, you'll actually see the action being executed:



*Execution in the level blueprint*

9. Of course, we don't want it to be set to **0**. We want it to flicker with random values, so let's create a new variable to be our new intensity. Drag out from the **New Intensity** arrow toward the left, and you'll see a number of different functions that return a float value. In the search bar, look for **Random** and from the list, select **Random Float in Range**. Under **Max**, put in **2000**.



**Note**

To move around the **Level Blueprint's Event Graph** window, right-click and hold, then drag. To zoom in and out, use the mouse wheel. To drag individual elements around left-click and drag them. For more hotkeys and information on moving around Blueprints, refer to <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/CheatSheet/index.html>.

Now if you were to play the game, you should notice that the light has a different brightness every time we play it. However, if we want the light to flicker, we should change the intensity value a lot of times as long as the game is running.

10. To the right of the **Set Intensity** function, right-click and create a **Delay** function node. Then, connect the exec output of **Set Intensity** to the input (left-hand side) of the **Delay** action. Then connect the **Completed** output to the input of the **Set Intensity** function.

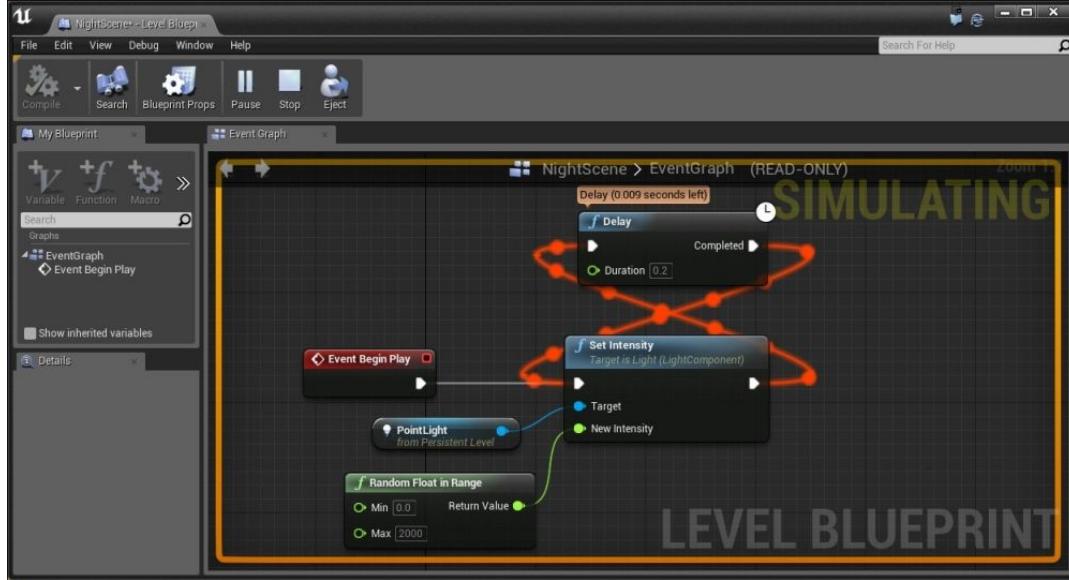
**Tip**

You can also hold the *D* key and then click to create a **Delay** function within the Blueprints Editor.



*Creating a loop for the Set Intensity function to be called every .2 seconds*

This is effectively creating a loop, causing the **Set Intensity** function to be called after every .2 seconds. If you hit the play button to see the actions executed, you'll notice that the actions are continuously happening, and in the game, you'll see that the values are indeed changing, but we notice that it will always change at a fixed rate which is predictable. A flickering light would be nicer to not be as static in terms of when the value changes so players are unsure as to when it will happen.



*Executing the flickering effect*

11. For the sake of readability, move the **Delay** action to the right of the **Set Intensity** function. Doing the same steps as before, create another **Random Float in Range** for the **Delay** action's **Duration** variable with a value between 0 and 0.2 .
12. When working on scripting like this, it is also a good idea to comment your code. This will help people looking at your code (including your future self) to find things more easily.
13. To do this, do a marquee selection around the entire blueprint by clicking slightly on the top-left of the **Event Begin Play** action. Then, drag to the right and down until all of the actions are selected and right-click on one of the functions. From here, select **Create Comment from Selection**. Name the box that was created, **Flashing Light**, and press *Enter*.

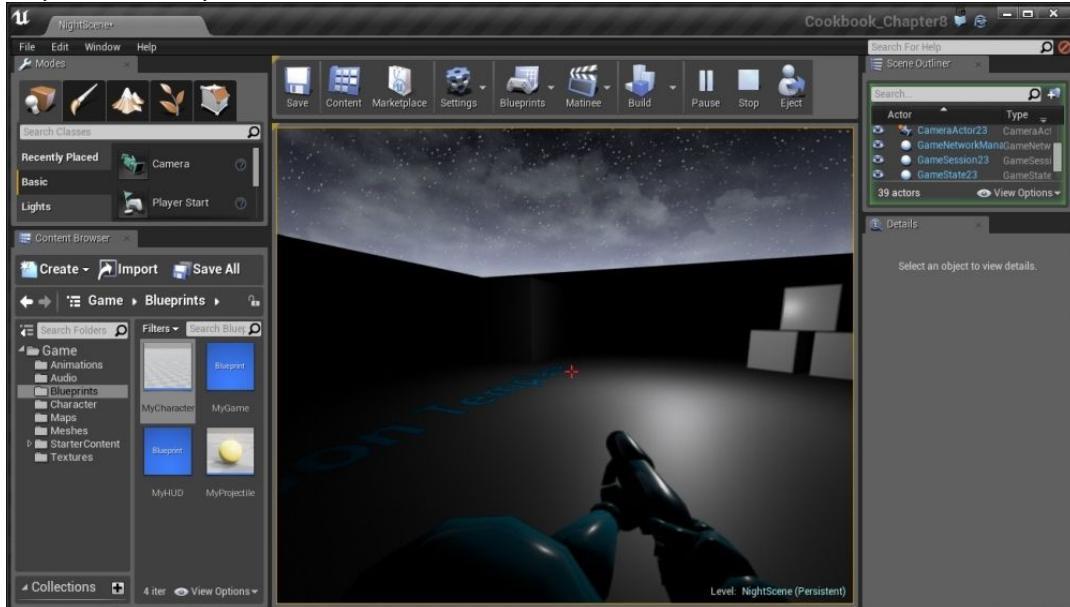


*Adding comments to the script*

### Tip

You can also press the **C** key to create a comment around whatever you have currently selected.

14. Save your level and hit the **Play** button!



With that, our flickering light is working correctly and it's looking good!

## Converting from Level to Class Blueprints

In the previous recipe, we used the **Level Blueprint** system in order to create a prototype of a feature we may want to have. In this recipe, we will see how we can convert this prototyped feature into a **Class Blueprint** which can be reused in every level.

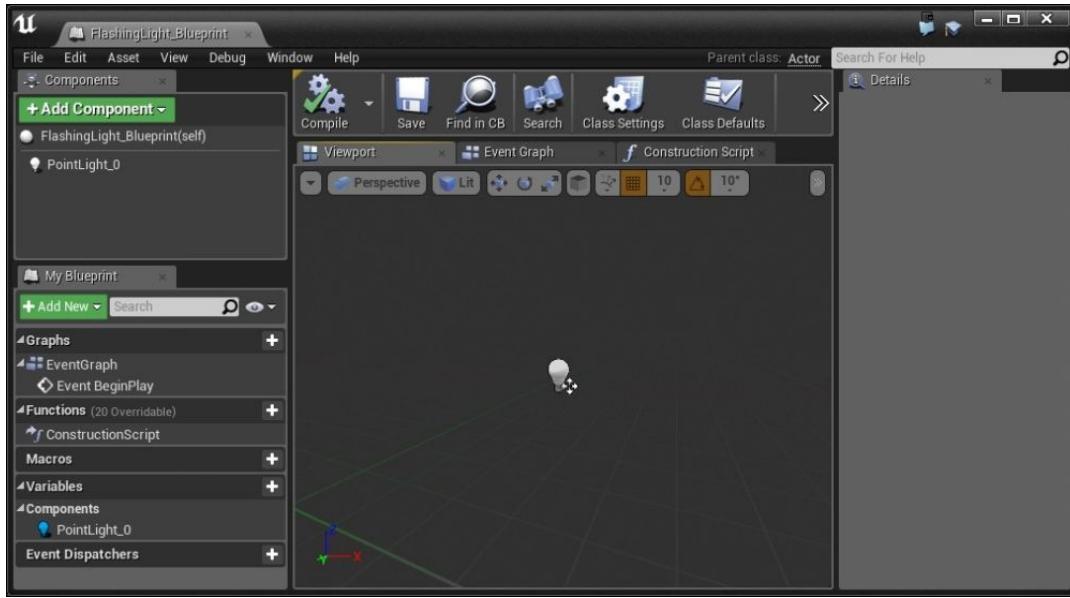
### Getting ready

Before we start working on this, we need to have a level blueprint to convert. Follow the previous recipe or open up the `FlashingLight` level provided in the example code.

### How to do it...

The **Level Blueprint** is great, but it's limited in the fact that everything created is stuck within that level, and we'll generally have to copy/paste a lot to move things between levels (this was a problem back in the UE3 days with Kismet). To solve this issue, we have **Class Blueprint**, which we will create now:

1. With our **Point Light** selected, go to the **Details** tab and scroll down to the **Blueprint** section. Then, click on the **Replace With Composited Blueprint** option. When the **Select Path** dialog comes up, select the **Blueprints** folder and enter the name `FlashingLight_Blueprint`, then press **Create Blueprint**.



You'll then be brought to the **Class Blueprint** for our newly created action and it has the Point Light, which we created earlier, added. We are currently on the **Components** tab, which is where all of the pieces that make this blueprint reside. We'll look over this in another recipe, but for now, let's move over to the **Event Graph** tab.

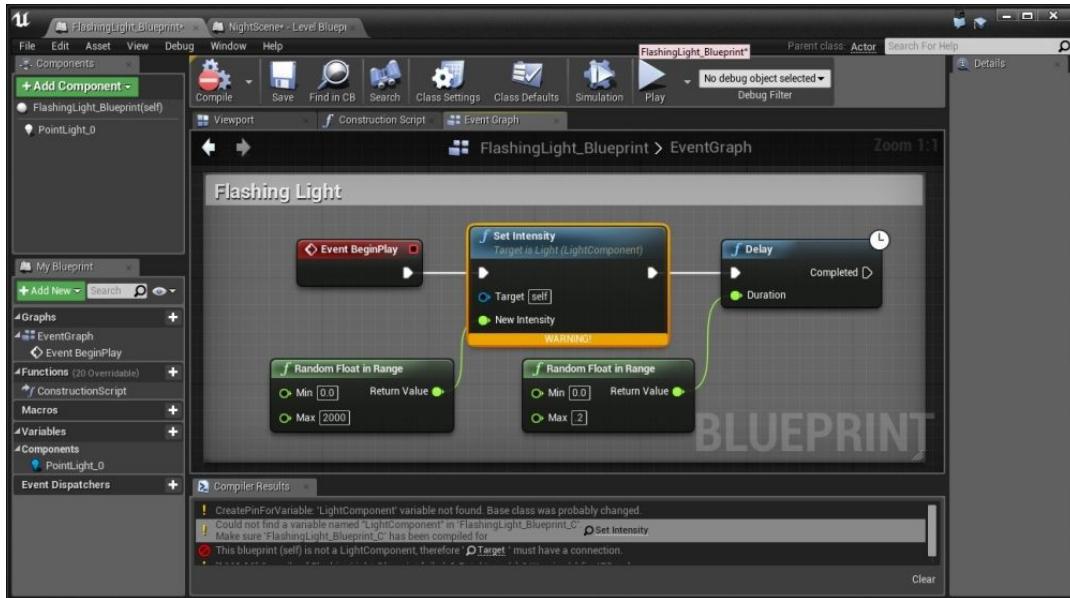
2. Click on the **Event Graph** tab. Just like before, we are given an empty blueprint to work with. However, this time you'll notice that the **My Blueprint** tab now has the **PointLight\_0** object as well as any other objects that may be part of the **Components** tab.



3. Move over to our **Level Blueprint** and select all of the nodes we created. Hit **Ctrl + C** to copy the nodes.
4. After this, move back to the Graph of FlashingLight\_Blueprint again, select the **Event Graph** tab and then paste (**Ctrl + V**).

You'll note that all of our previous content moved over nicely, except for the Point Light. This is because we were previously using a reference to something in the level. When working with blueprints, you are only able to access things inside the blueprint itself (unless you can get a reference to it).

5. Delete the **Unknown** element by selecting it and then hitting the **Del** key.

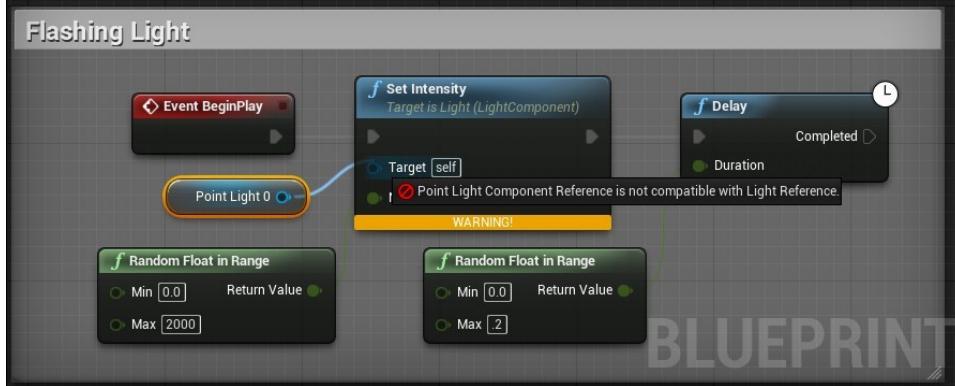


*Deleting the Unknown element causes an error*

This causes an error because we do not have something set for **Target**, but we will use our Blueprint's component to work with it.

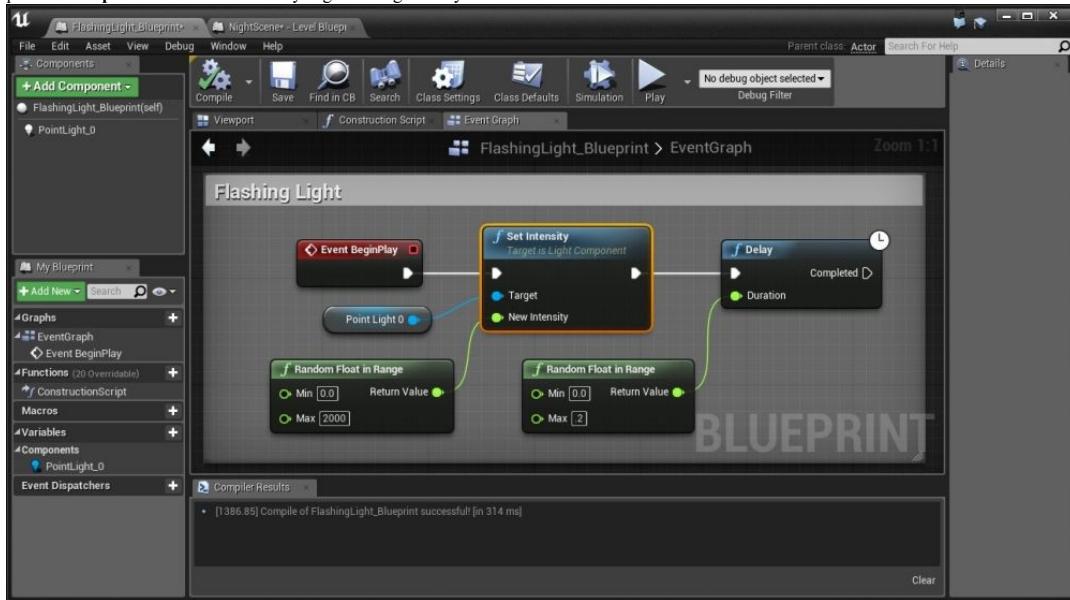
- From the **My Blueprint** tab, drag and drop the `PointLight_0` object into the Event Graph. You'll then have the two options of whether we want to modify the value of it (**Set**) or obtain the value to use (**Get**). In this instance, we will use **Get**.

However, at this point, if we try to drag and drop the point light into the **Target**, you'll notice that there is a conflict due to the types not being the same.



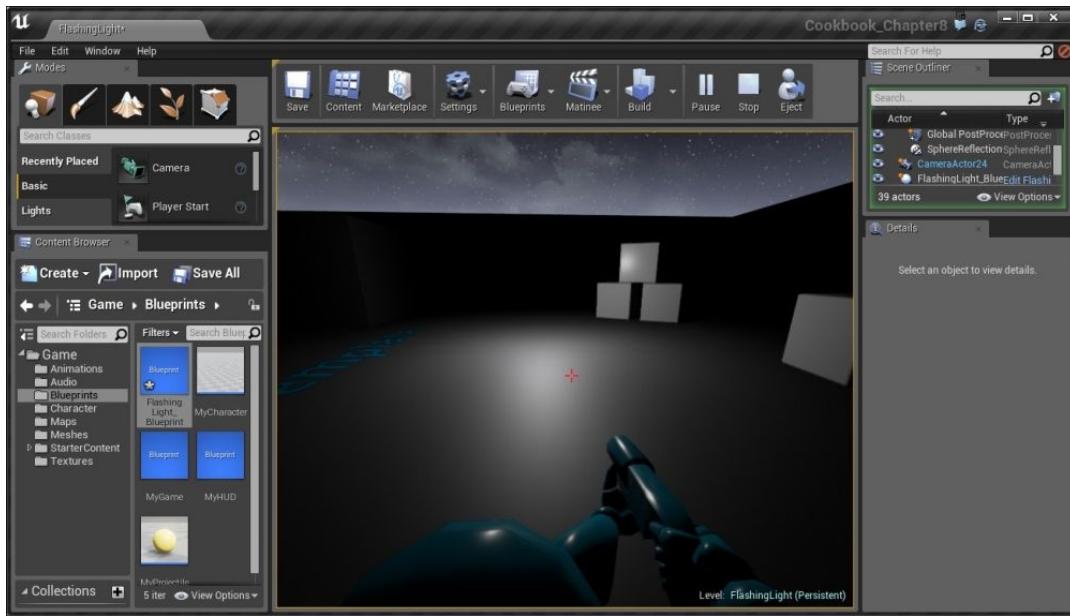
This is easily solved, but I wanted to point out the similar problems like this may happen on occasion, so let's fix it.

- Delete the **Set Intensity** action and drag from `PointLight_0` and create a new **Set Intensity** function using the correct component. Then, connect everything back the way it was in the previous version and then press the **Compile** button to make sure everything is working correctly.



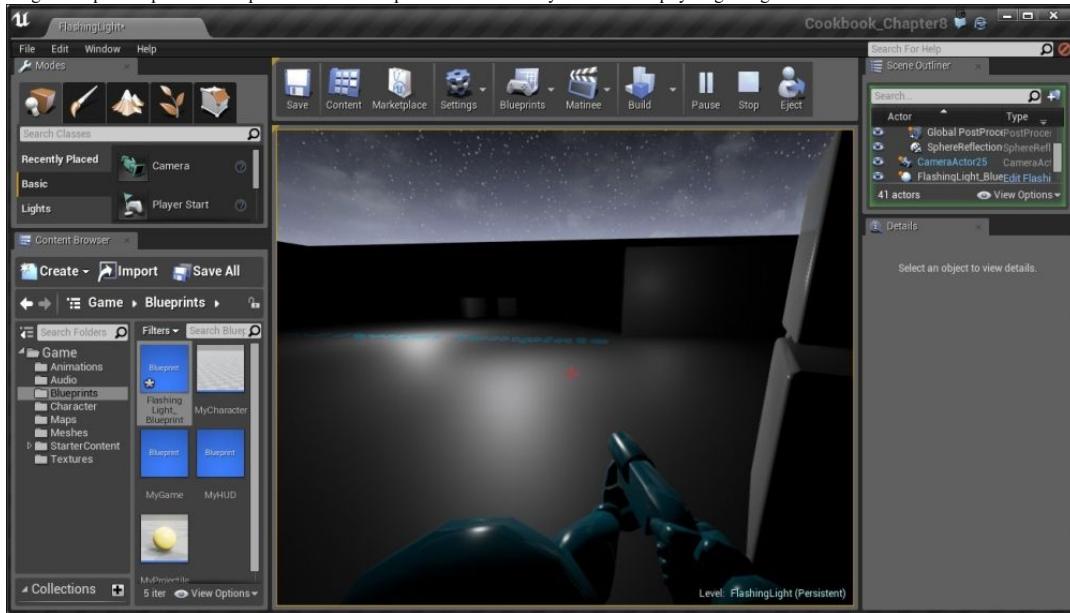
*Creating new Set Intensity function*

- Lastly, move into the **Level Blueprint** window and delete our previously created actions. We won't need them anymore (note the warning that exists now because the object is now a blueprint).
- Save your blueprint and the level and then hit the **Play** button.



Everything is the same as before! However, now that our light is a blueprint, we can use it as many times as we want without having to copy/paste any actions.

10. Drag and drop two copies of the blueprint into the level and position them wherever you'd like to and play the game again.



*Playing the game with two copies of the blueprint*

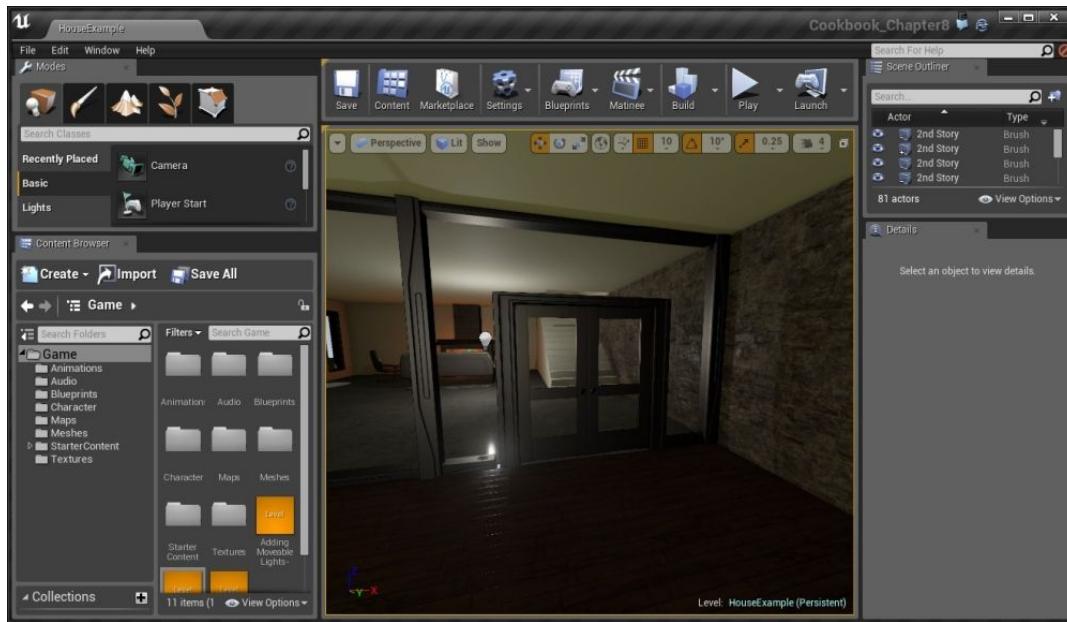
Now we have multiple lights, reacting in their own independent ways while learning about class blueprints! Also, since you have learned how to modify a property, you can also modify any of the other properties of the light, such as its color, position, and so on!

## Using Trigger Volumes – opening a door using Matinee

Of course, there are many other things that we can do inside Blueprints instead of flickering a light. Another example would be to open a door when we get near it making use of Matinee.

### Getting ready

Before we start working within the Unreal Editor, we will need to have a project to work with which contains the Starter Content that Epic provides. After the project is opened, go to the `Example Code` folder and drag and drop the `HouseExample` map provided into your project folder and open it.



## How to do it...

In order to have our doors open, we will want them to be animated. To do this, we will use Unreal's Matinee system.

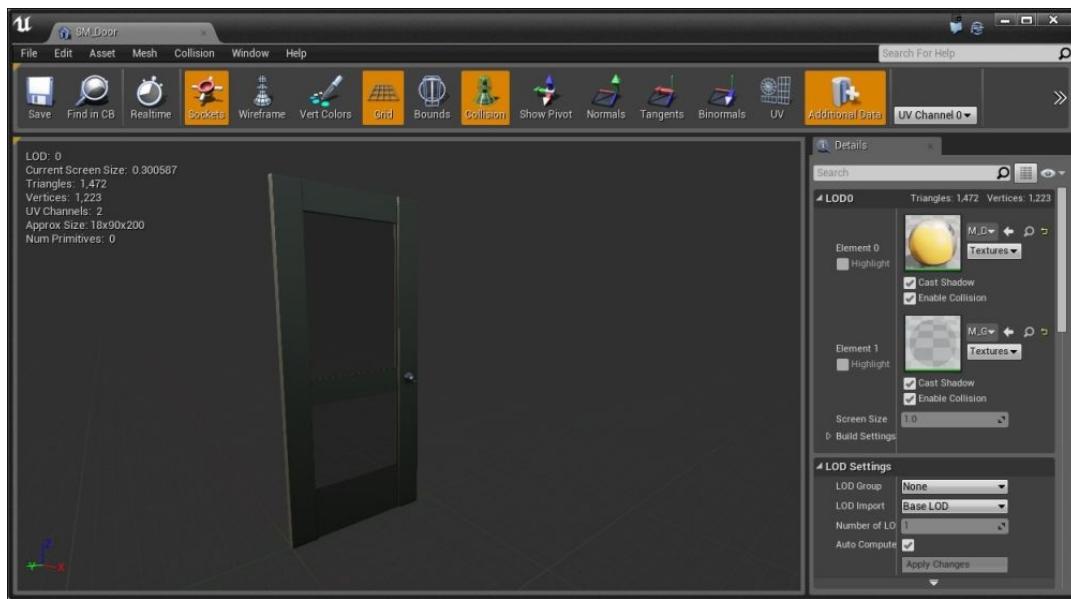
1. Select both doors and ensure that the **Mobility** section is set to **Movable**. If this is not done, you may have errors in Matinee when we say we want to move them.

Currently, our doors do not have collision. Let's fix this by looking at the common reasons for why things cannot be working correctly.

2. Back in the **Details** tab, go into **Collision** and note that **Collision Presets** is set to **BlockAll**. This means that if the object has collision data, anything will collide with it.
3. The key thing to note is that the object will only collide *if* there is collision data. Let's check this now. Scroll up to the **Static Mesh** component and double-click on the door to open up the **Static Mesh Editor**.

### Tip

Alternatively, you can also press *Ctrl + B* when the doors are selected in the scene to automatically find the asset in the **Content Browser** tab and then double-click from there.



*Opening the Static Mesh Editor*

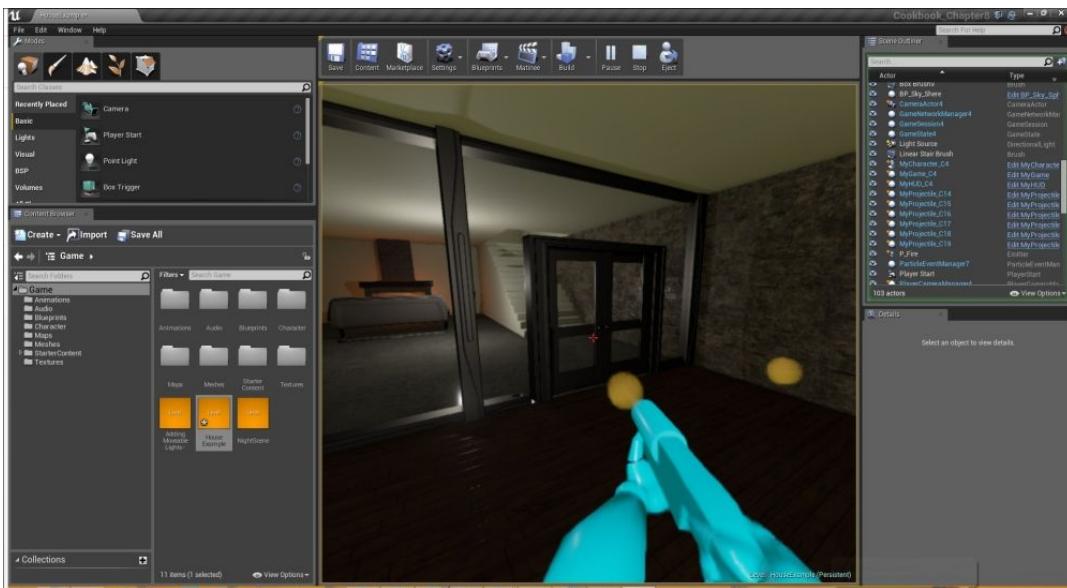
Inside this editor, you can modify any properties that are part of this mesh, including the materials used and the way it reacts to shadows.

### Note

For more information on the Static Mesh Editor, refer to <https://docs.unrealengine.com/latest/INT/Engine/Content/Types/StaticMeshes/Editor/index.html>.

4. There's currently no collision on this object, so let's add it. Go to **Collision** and select **Add Box Simplified Collision**, and with that, you'll notice that there are green lines over the edge, which is correct enough for a simple model like this.
5. Click on the **Save** button and exit the Static Mesh Editor.

Now, if we play the game, you'll notice that we can no longer walk through it!



*Playing the game after adding collision to it*

- Now that the doors are ready, let's learn how to animate them. From the main toolbar, navigate to **Matinee | Add Matinee**.



*Opening up the Matinee Editor*

Once this is created, there is a new Actor added to our **SceneOutliner** called `MatineeActor1`, and the Matinee Editor opens up.

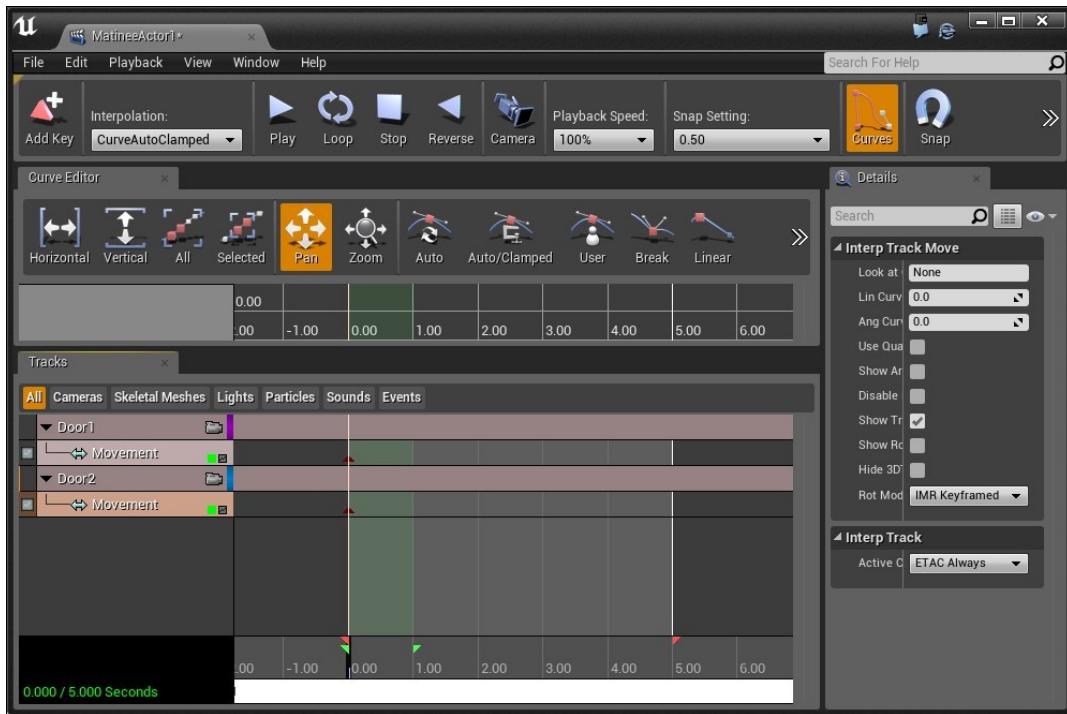
#### Note

For information on the Matinee Editor and what each section is for, refer to <https://docs.unrealengine.com/latest/TNT/Engine/Matinee/UI/index.html>.

- Back in the main editor, click on the left door and then open up the Matinee Window again. From the **Tracks** tab, right-click and select **Add New Empty Group** and give it a name—`Door1`. Once this is created, right-click on `Door1` and select **Add New Movement Track**.

This is letting Matinee know that we want to modify the movement (position, rotation, and scale) of the door object during our animation. It's also worthwhile to note that if the door is not a moveable object, Unreal will automatically change it to one at this point.

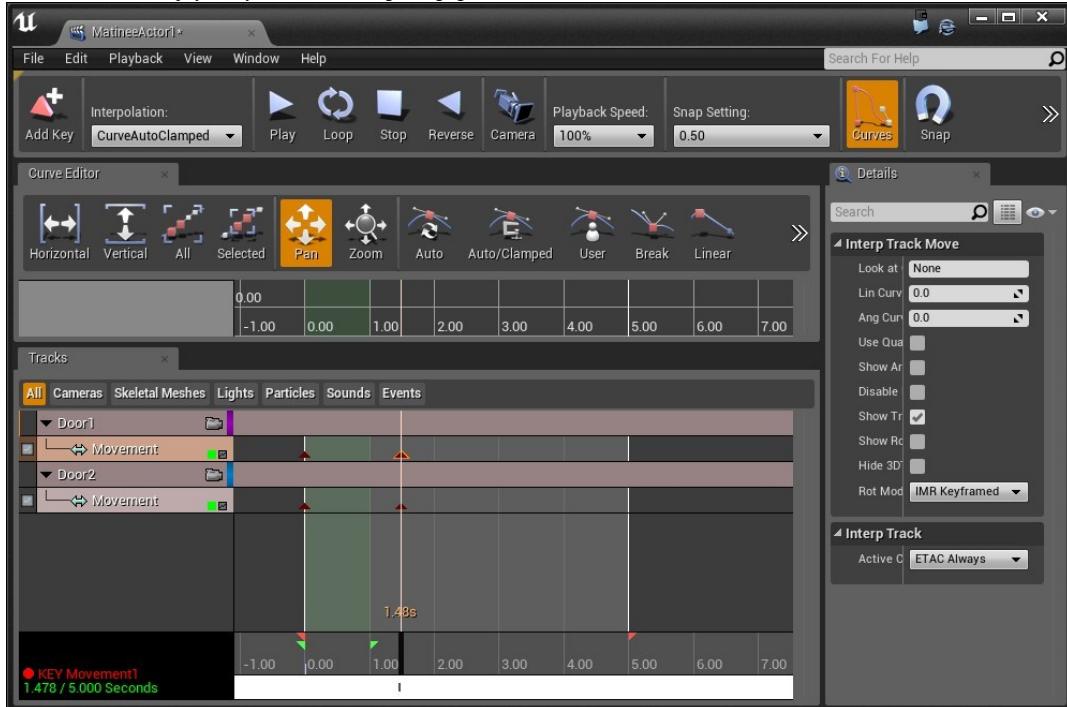
- Next, we want to do the same thing with the right door. Select it in the main editor and then back in the Matinee Editor, right-click and create a new group with a movement track.



*Adding the movement track*

Animations are done in terms of keyframes. You can already see one created on each Movement track with those red triangles. This current animation lasts for 5 seconds (which you can see with the **5.00** value where the edges are). Use the mouse wheel to zoom in and out).

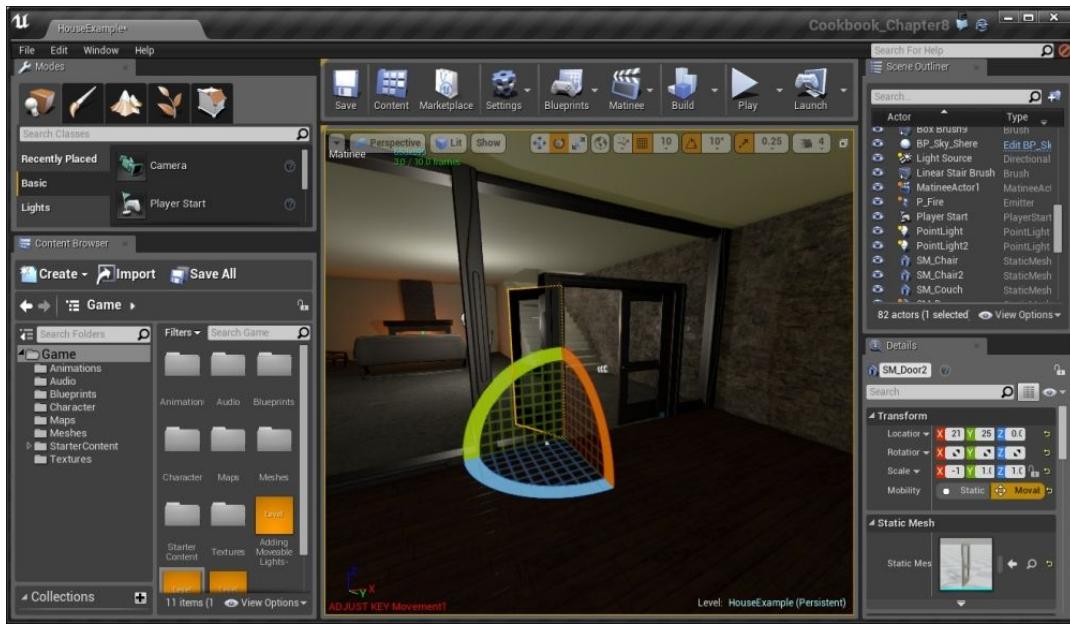
- Click on the gray area where the time is to show a spot on the timeline. Move it to around **1.5** seconds. Then, select each **Movement** track individually and click on **Add Key** on the top-left (or press *Enter*). To see the animation with the playback system, also move the green highlighted timeline to **1.5** seconds.



*Setting a new keyframe at 1.5 seconds*

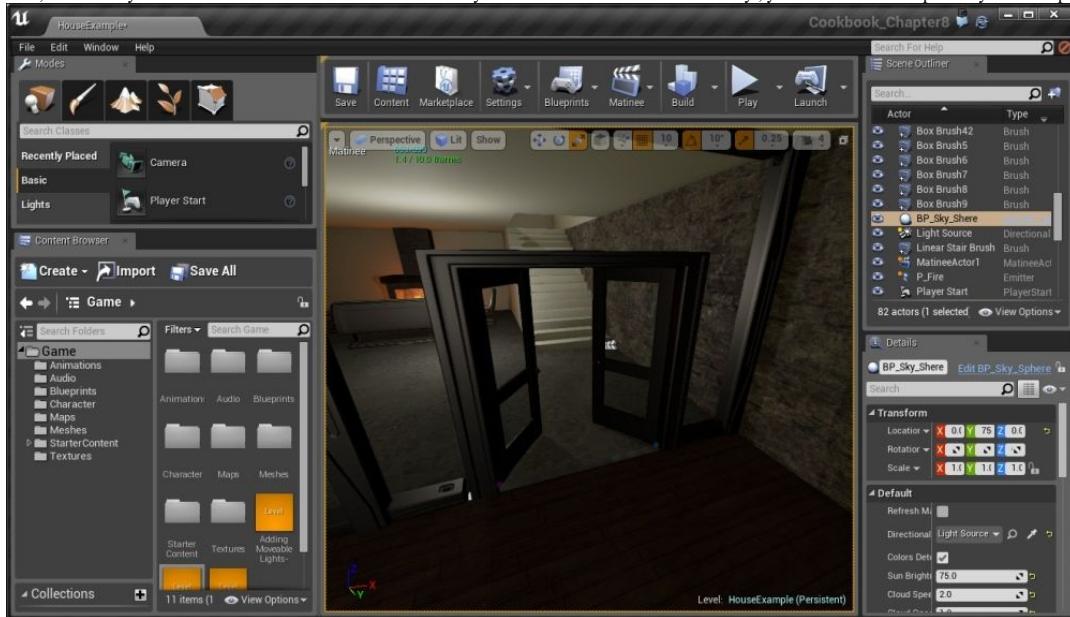
This is stating that in 1.5 seconds, I want these objects to be somewhere else and I want the computer to intelligently move between the two values (also known as a **tween** ).

- Now, select the key on the **Door1 Movement** and move back into the game. Switch to the **Rotation** tool (**E**) and then rotate the door to open **120** degrees.



*Rotating a door*

11. Next, select the key on **Door2** and do the same in the other direction. If you move the timebar between the two keys, you should notice that it parts way between opening the door.



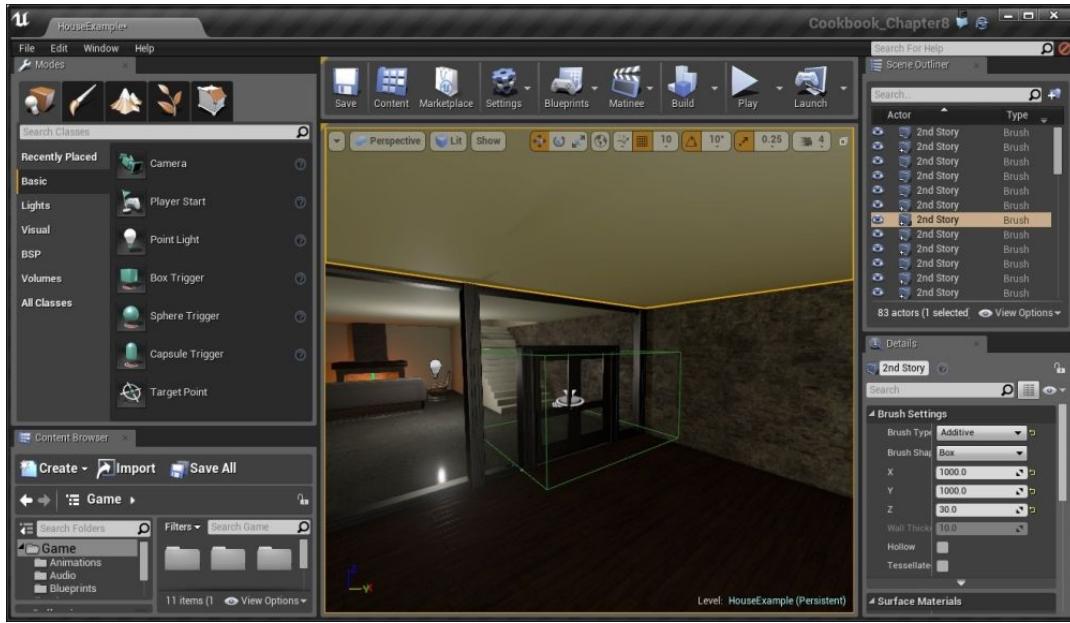
*Rotating the other door*

12. When our Matinee is completed, close the editor.

We now want this animation to play whenever we get near the door, so with that in mind, we can use **Box Trigger** to detect when we are near.

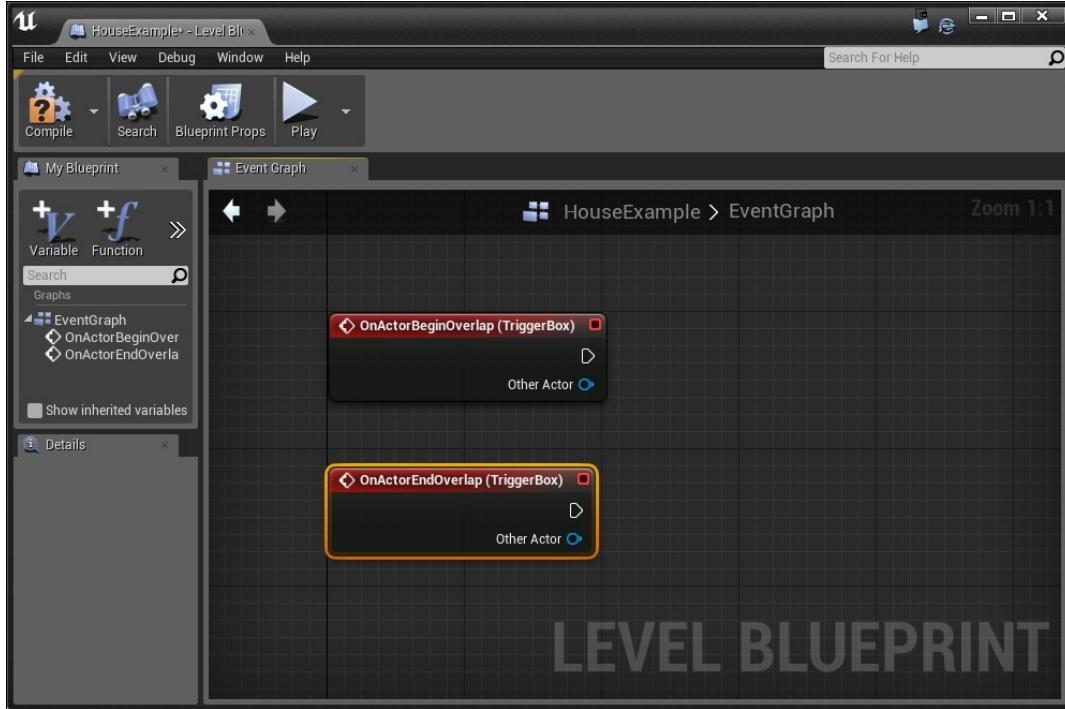
A **trigger** is something that detects collision, but unlike other things, with collider, it does not attempt to block the player from entering it. This is used often in games for the events that happen when something enters an area. If you've played a first person shooter, you may notice a trigger being used whenever you see a group of enemies spawning to attack the player because you entered an area.

13. Go to the **Modes** tab and then select **Basic** and drag and drop **Box Trigger** into the scene where the door is. Scale and translate it until it covers the door and the space ahead and behind it.

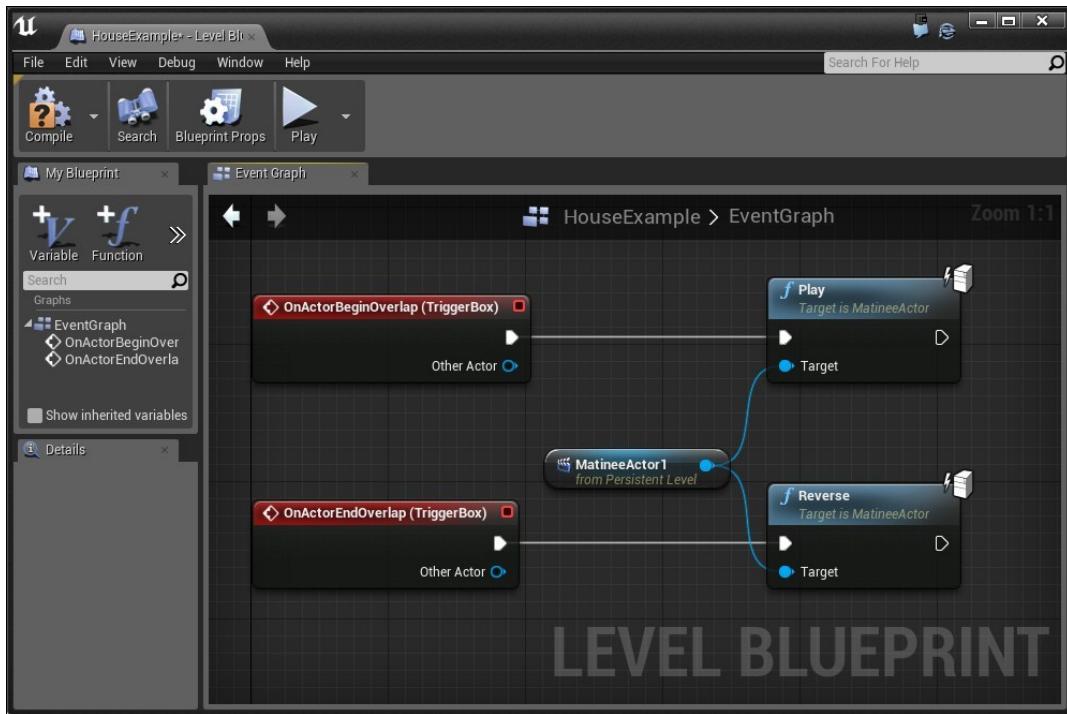


*Setting a box trigger at the doors*

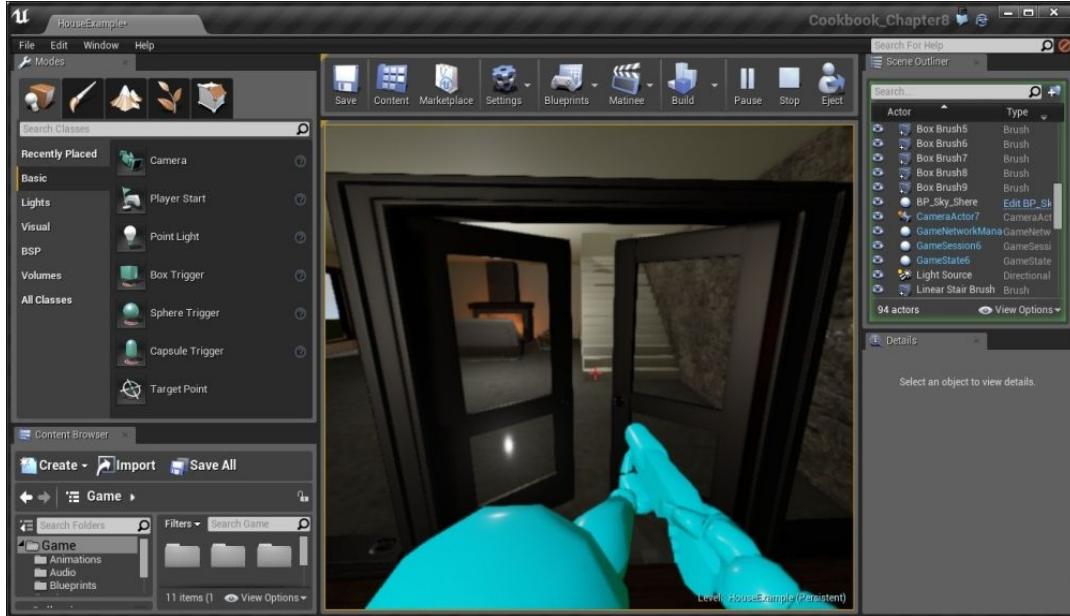
14. In the **Scene Outliner** tab, select our **TriggerBox** actor and scroll down to Blueprint. From here, select **Add Level Events for TriggerBox** and from the popup, select **Add OnActorBeginOverlap**. This will be called whenever our **TriggerBox** has been overlapped by another object.
15. While we are here, also add an event for **OnActorEndOverlap**.



16. We want to play our Matinee when we collide with the trigger, so from the **Scene Outliner** tab, go to **MatineeActor** and drag and drop it into the **Blueprints** window. Then, drag from its blue line to bring up the possible functions that use it and search for **Play**. Connect the output from **BeginOverlap** to the input of **Play**.
17. With that in mind, extend from the Matinee actor once again and search for **Reverse**. Connect the output from **EndOverlap** to the input of **Reverse**.



18. Hit the **Compile** button and play the game.



With this, you have learned to use Triggers to modify things that happen in the game!

#### Note

For simple animations, such as the one done here, it's also possible to make use of UE4's Timelines. For a tutorial on that, visit <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Timelines/Examples/OpeningDoors/index.html>.

For additional examples on how to use Matinee to do different things, refer to <https://docs.unrealengine.com/latest/INT/Engine/Matinee/HowTo/index.html>.

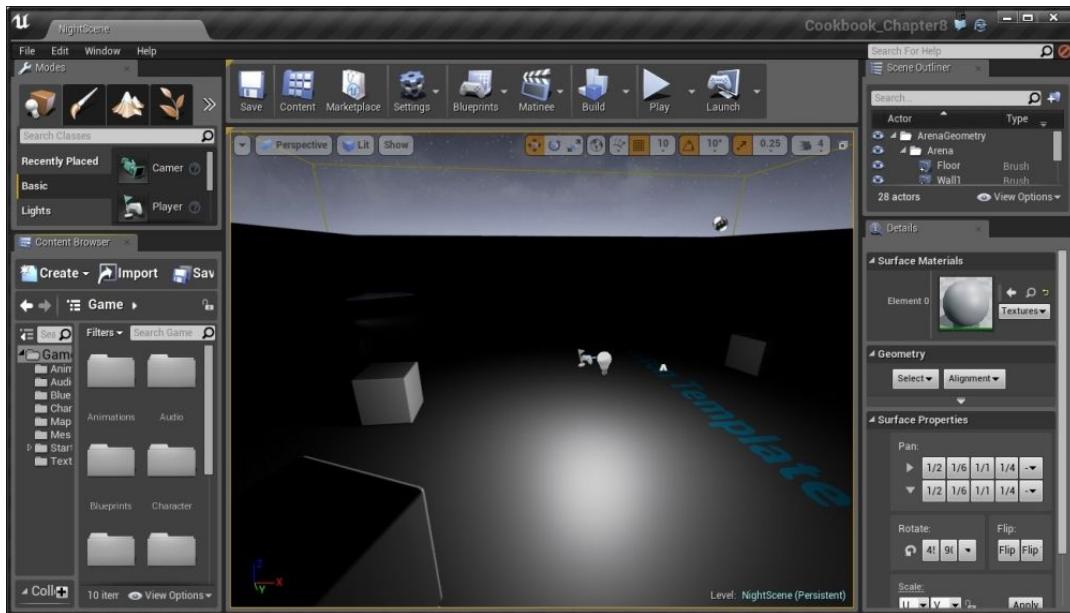
## Adding to an existing Blueprint – flashlight, part 2

Now that we have some experience working with Blueprints, let's modify one that's already been created for us. In this section, we will cover how to add a flashlight to our game's character.

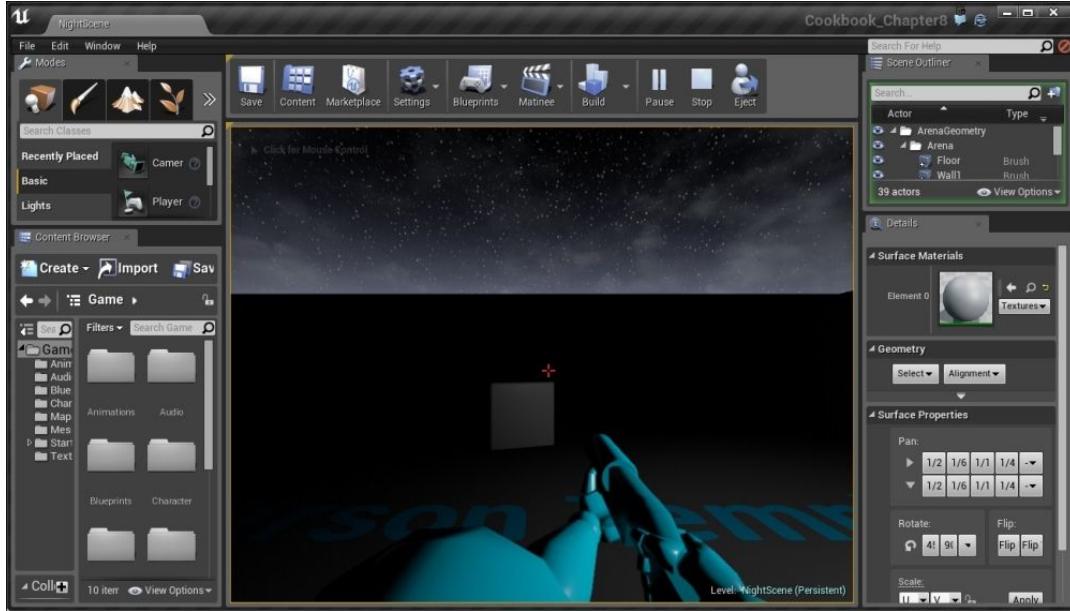
#### Getting ready

Before we start working within the Unreal Editor, we will need to have a project to work with. Follow these steps:

1. First, open up the Unreal Editor by clicking on the **Launch** button from the Unreal Engine Launcher.
2. Start a new project from the **Project Browser** tab by selecting the **New Project** tab. Select **First person** and make sure that **With Starter Content** is selected. Give the project a Name (**Cookbook\_Chapter8**). Once you are done, click on **Create Project**.
3. After the project is opened, go to the **Example Code** folder and drag and drop the **NightScene** map provided in your project folder and open it.



At this point, if you were to play the game, it would be incredibly dark, as shown in the following screenshot:

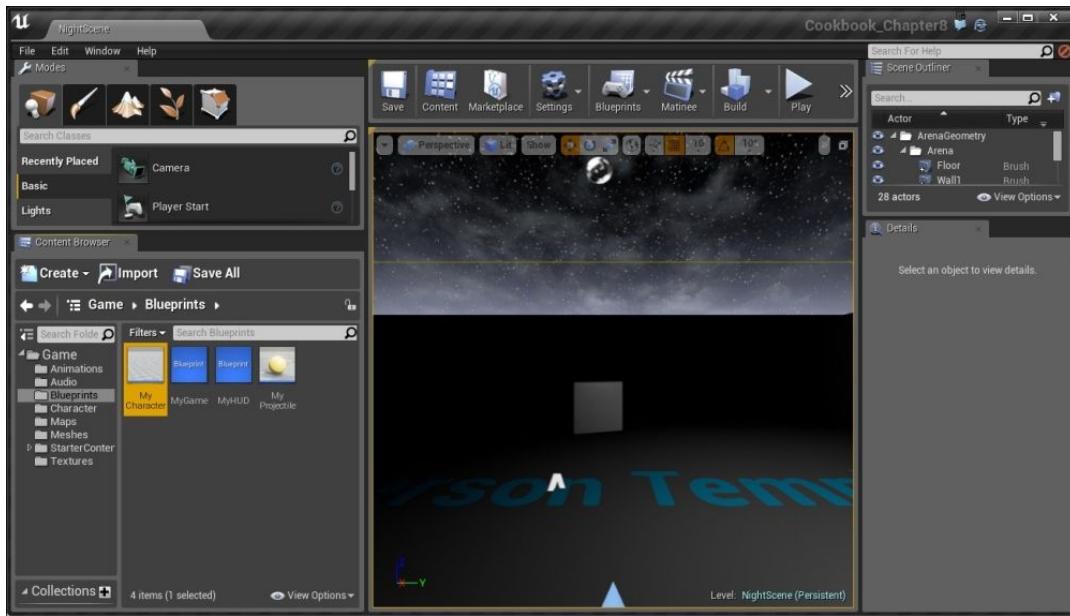


## How to do it...

For those who have gone through [Chapter 6, Lighting and Shadows](#), this should look quite familiar. In that chapter, we actually created a flashlight and then attached it to our player at runtime.

However, we can modify the character's blueprint to make it so that our player will always have a flashlight on. Follow these steps:

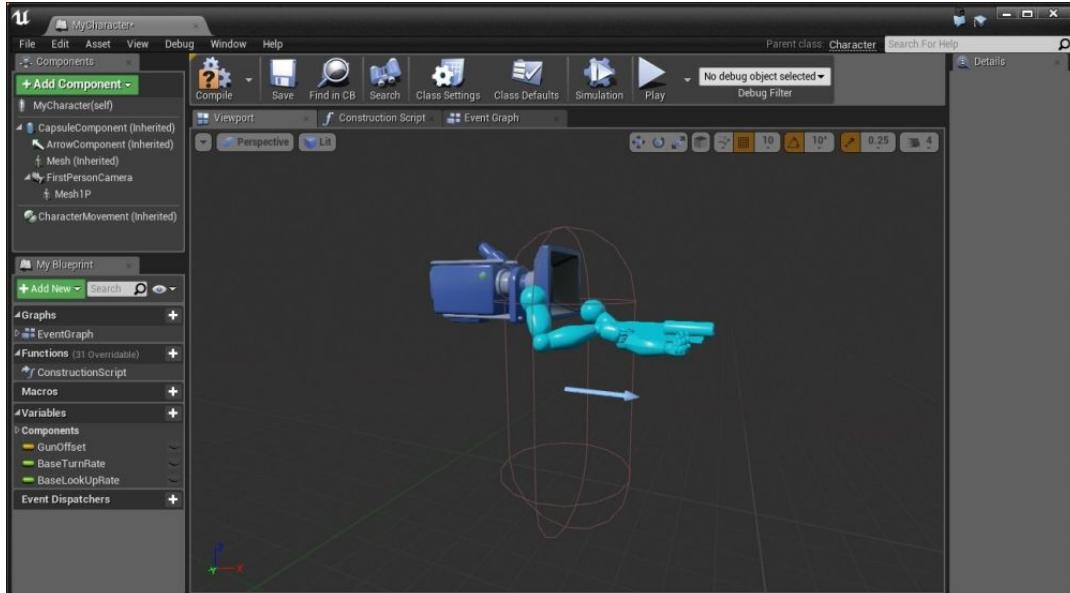
1. Go to the **Content Browser** tab and then to the **Blueprints** folder. Find the **MyCharacter** file.



Selecting the *My Character* blueprint

This blueprint is what's spawned when the game is started. It contains all of the logic for movement, the camera, and the shooting behaviour.

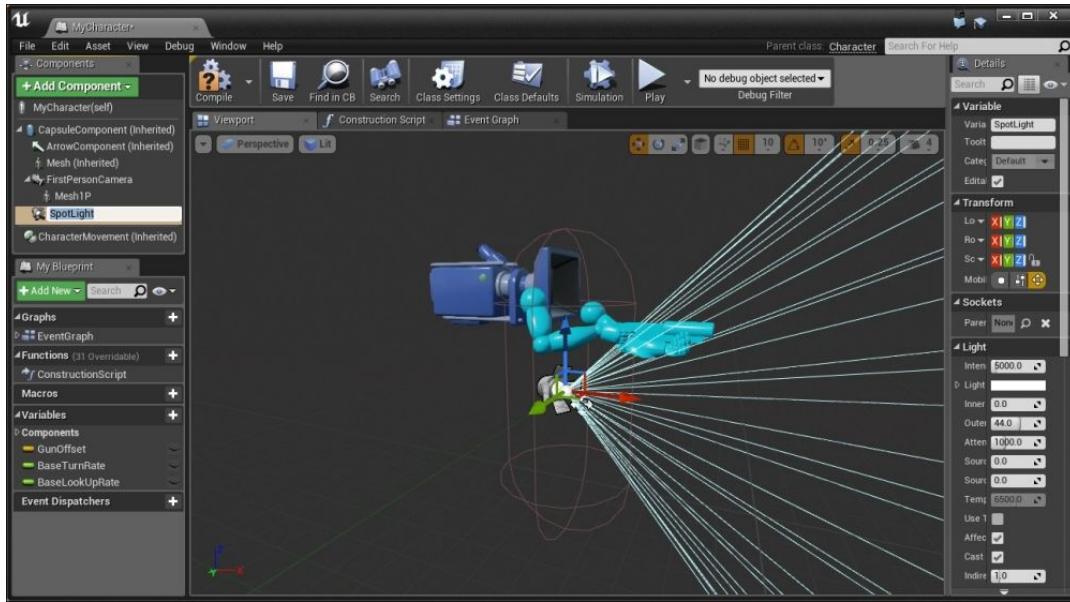
2. Next, double-click on the file to open up the **Blueprints** for it. Initially, it will be on the **Event Graph** tab, which will have a lot of scripts on it, very similar to a level blueprint. But for now, click on the **Viewport** tab and zoom the camera out if needed.



The Viewport tab view in the blueprint editor

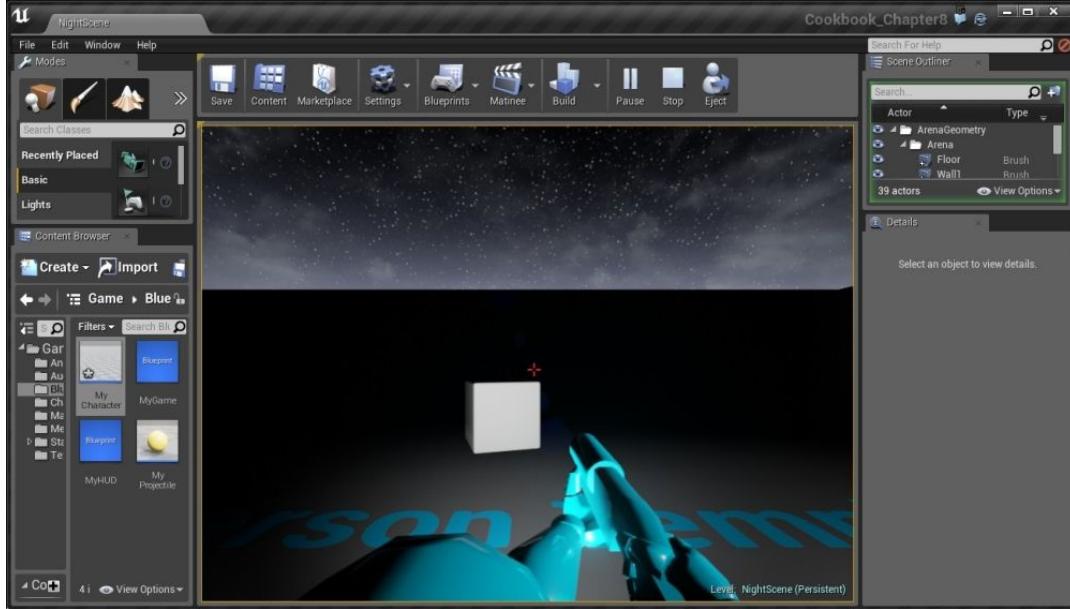
This section is similar to our level in that it has objects and they're placed in the *world* of our object, but it's apart from the level. You can move around the viewport on the right-hand side in exactly the same way as the main game. Zoom out so that you can see everything.

3. To the left, you'll see a tab marked **Components**. Under this, it has the mesh, camera, and everything else that is to do with the physical presence of the class. We can add to this using the **Add Component** option. With this in mind, click on **Add Component** and select **Spot Light** (if you can't see it, use the search bar at the top).

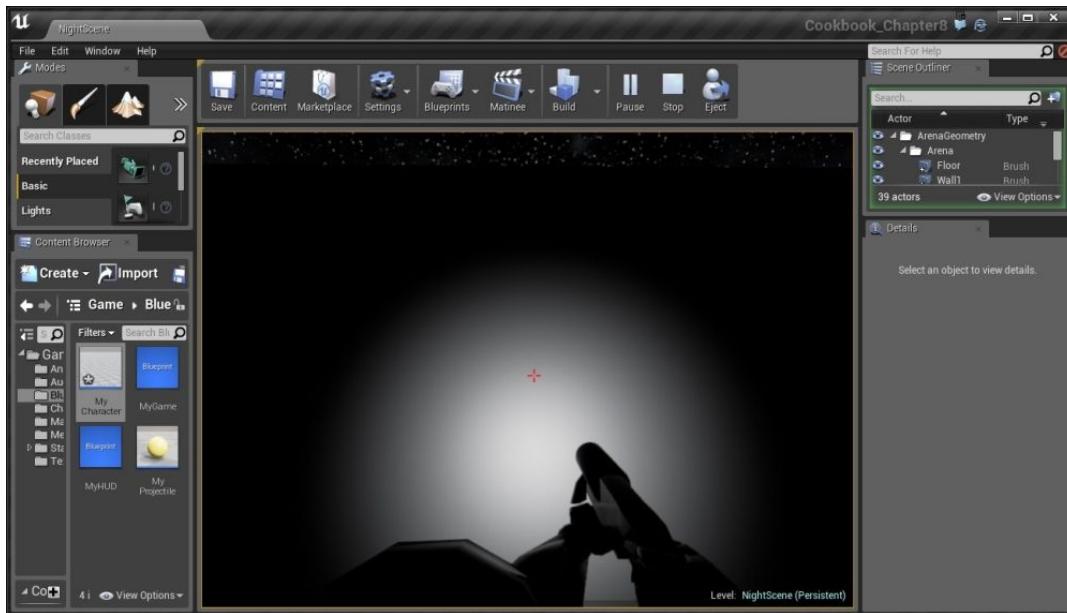


*Adding a Spot Light to our character*

You can see that the spotlight is already added for us.  
4. Go back into the main editor and play the game again.



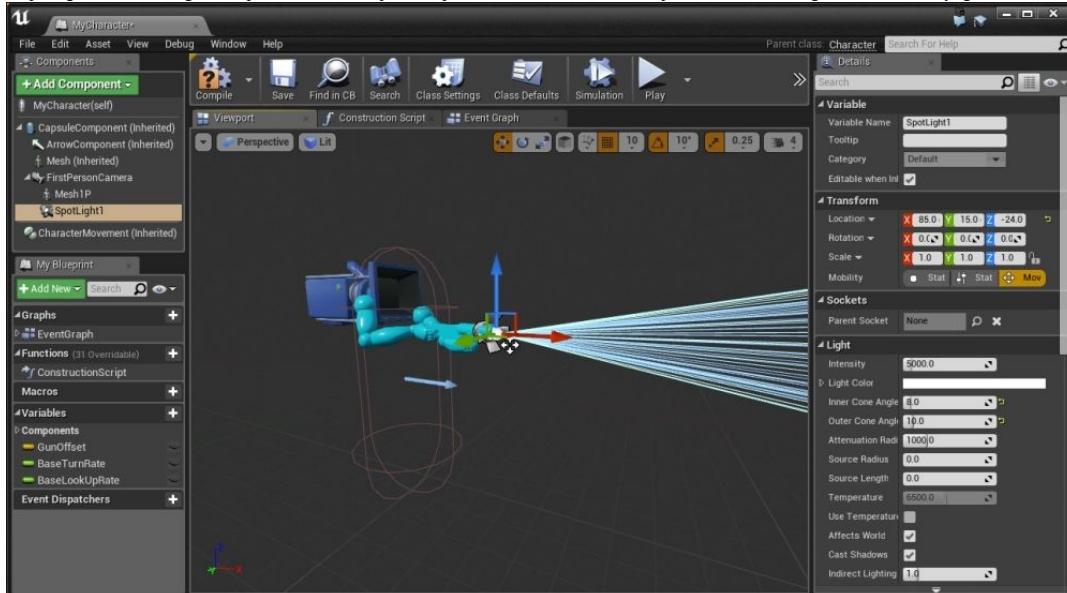
You'll notice a couple of issues. First of all, the light is lighting the player in addition to what's in front of it. Also, if we look up and down, it does not move with us.  
5. With the game still playing, go back to the `MyCharacter` blueprint and select our `SpotLight` object and move it (if the `Move` tool isn't there, press `W`). Notice that when we move it in the blueprint, it modifies what is done in the game. Move the light until it is in front of the barrel of the gun.



*Moving the spot light in game*

This solves the issue of lighting the player, but moving up and down is not working. To do this, we can make the **SpotLight** object a child of the camera. That way, whenever it moves, the light will move as well. Sadly, we cannot change this while the game is running.

- Stop the game and then drag and drop the **SpotLight** object on top of the **FirstPersonCamera** object. You should see a green checkmark, saying that we can attach it.



*Attaching spotlight to the FirstPersonCamera object*

- After this, go into the **SpotLight** objects and set the same parameters that we did in the *Adding moveable lights – flashlight, part 1* recipe of [Chapter 6, Lighting and Shadows](#) or translate the object so that it is in front of the camera with **Inner Cone Angle** of 8 and **Outer Cone Angle** of 10.
- Click on the **Compile** button to save all the changes we've made before and confirm that it all works correctly with our project.
- Lastly, let's add in the ability to turn the flashlight on and off using an input event. Click on **Event Graph** and move it over until you have some empty space.

#### Tip

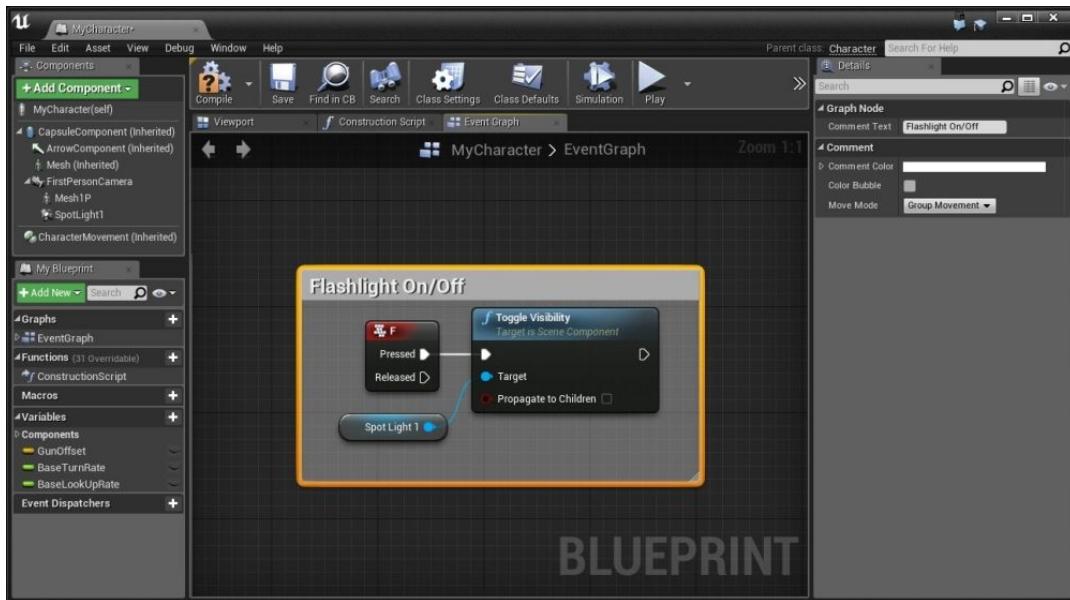
While in **Event Graph**, it may be beneficial to take a look at the actions that are actually part of the player class, giving you an idea of what can be done with blueprints and other actions that you can use!

- Right-click on the empty space and type in **keyboard events**. This will display all of the keyboard keys and allow us to trigger events based on when they are pressed or released.
- Select **F**, and you'll see an event is created. To the right of it, right-click and create a **Toggle Visibility (SpotLight1)** action. Toggle means that it will turn it off if it's on and on if it's off.

#### Tip

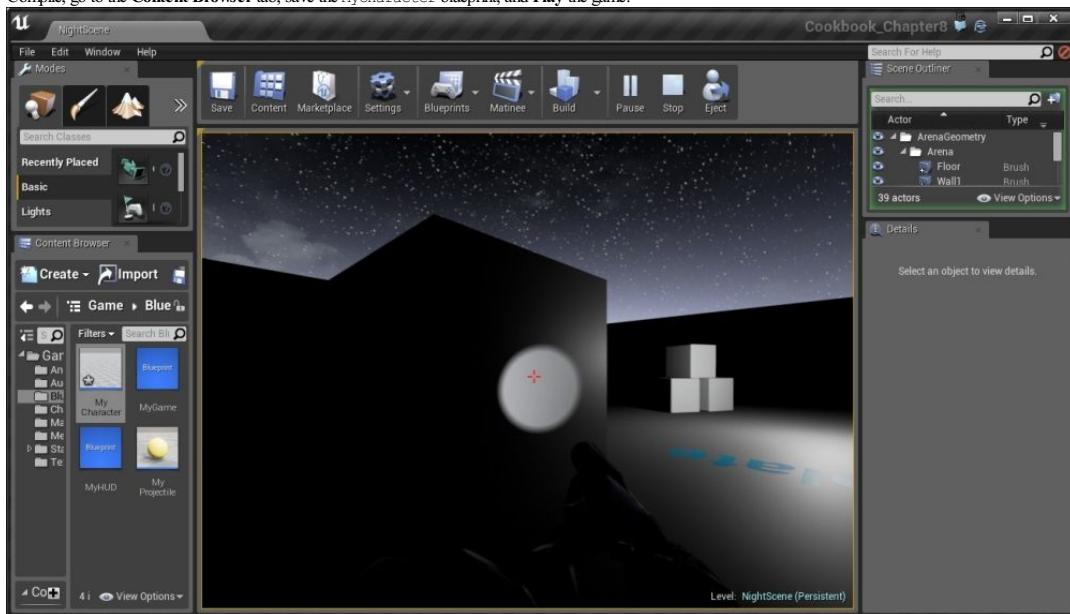
If you want to specifically set a property, use the **Set Visibility** option using a variable to tell what it should do.

- Next, connect the **Pressed** output of the **F** event to the **input** of the **Toggle Visibility** action.



*Connecting the Pressed output to the input of the Toggle Visibility action*

13. Compile, go to the **Content Browser** tab, save the `MyCharacter` blueprint, and **Play** the game!



With this, we now have a flashlight that will follow our player's camera in all directions and exist in every single level we have in the game! Here, you learned how to modify the already existing blueprints and added in some simple player input using the `F` key to toggle visibility.

## Creating a Health/Damage system, part 1 – taking damage

Games will often need the ability to give health to players and/or enemies and players need to have the ability to gain and/or lose the health. In this recipe, it being the first of a two-part recipe, we will create the ability to take and heal damage, whereas later on in [Chapter 10, User Interface](#), we will use the UI tools in Unreal to display this information.

### Getting ready

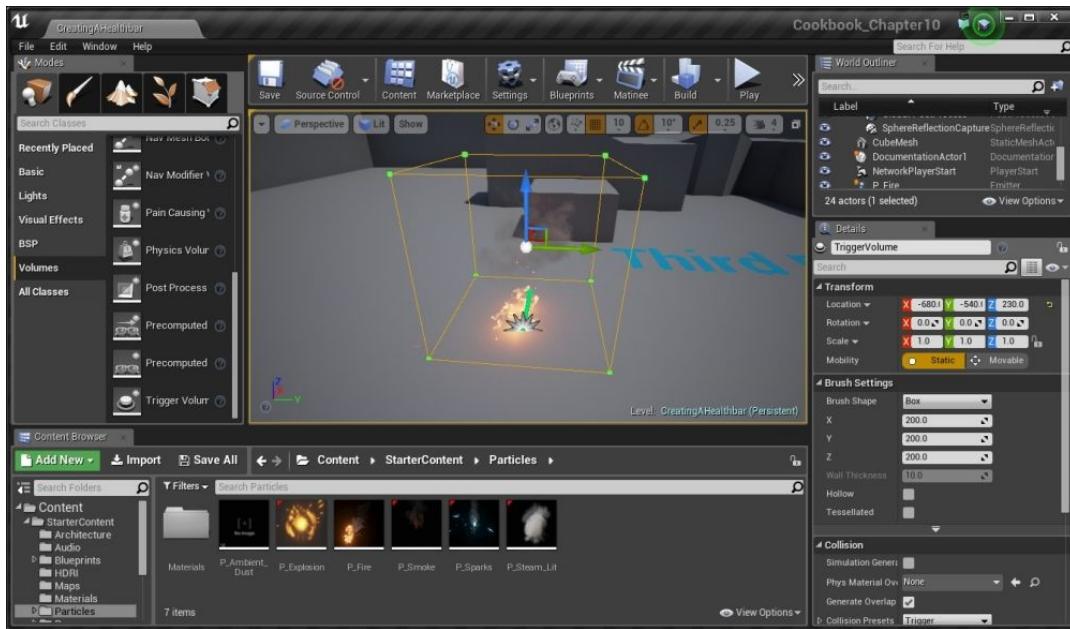
Before we start working on this, we need to have a project created and set up for our character to actually have health and a way to damage it. Follow these steps:

1. First, open up the Unreal Editor by clicking on the **Launch** button from the Unreal Engine Launcher.
2. Start a new project from the **Project Browser** tab by selecting the **New Project** tab. Select **Third person** and make sure that **With Starter Content** is selected. Give the project a Name (`Cookbook_Chapter8`). Once you are done, click on **Create Project**.

### How to do it...

Now, once the project is created, the first step here will be to create some action to damage the player. In this instance, I'm going to create a trigger. When the player touches the trigger, it will get damaged. To make it easier to see, I'm going to add a fire particle system. Follow these steps:

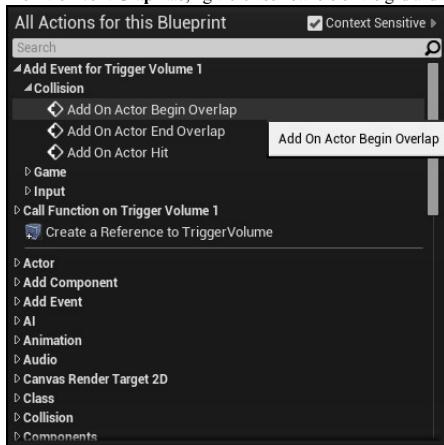
1. From the **Content Browser** tab, open up the `StarterContent/Particles` folder and drag and drop the `P_Fire` particles onto the floor of the starting level.
2. Next, from the **Modes** tab, go to the **Place** mode (if you weren't there already) and select the **Volumes** section. From the options, scroll all the way down until you see **Trigger Volume** and then drag and drop that into the level so that the fire overlaps it.



*Creating a trigger to damage the player*

Now that we have the trigger in place, we need to create a blueprint to do the damage for us.

3. With the **Trigger Volume** object selected, go to **Blueprints | Open Level Blueprint**.
4. From the **Event Graph tab**, right-click somewhere on the grid and then navigate to **Add Event for Trigger Volume 1 | Collision | Add On Actor Begin Overlap**.

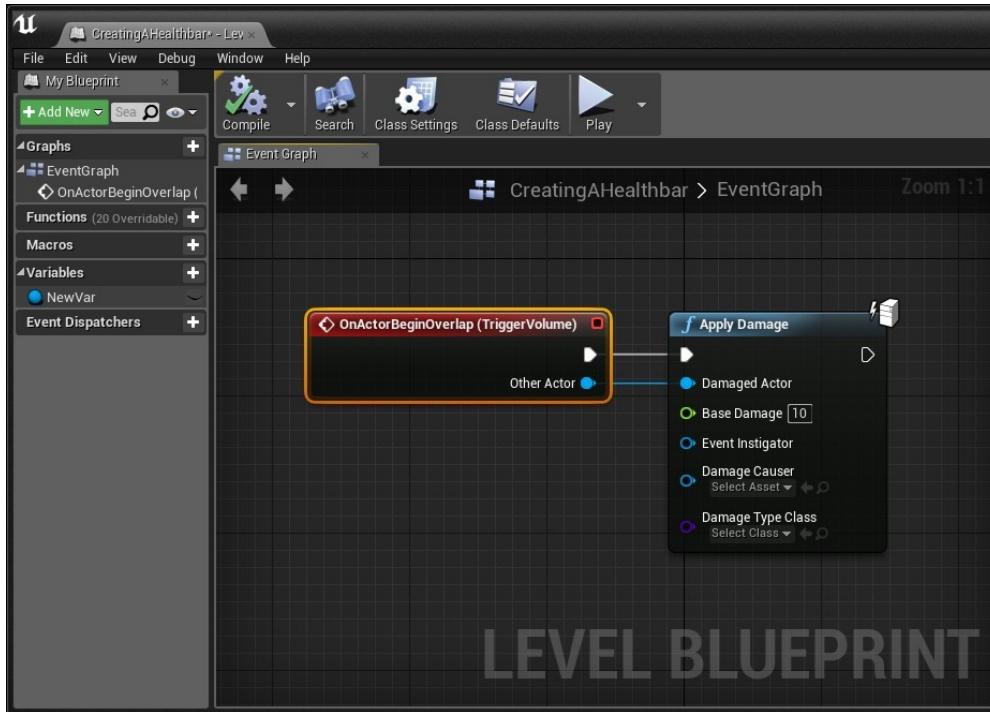


This event will be called whenever any other actor begins to overlap this actor.

#### Note

For more information on **OnActorBeginOverlap** and all of the other Collision Responses in UE4, refer to <https://docs.unrealengine.com/latest/INT/Engine/Physics/Collision/index.html>.

5. Next, when we collide with the trigger, we want to damage the player. Thankfully, Unreal comes with an **Apply Damage** action that we can use. To the right of the **OnActorBeginOverlap** action, right-click and start typing **Apply** and when **Apply Damage** is selected, press the **Enter** key.
6. Connect the output from the **OnActorBeginOverlap** action to the input of the **Apply Damage** action. Then, connect **Other Actor of OnActorBeginOverlap(TriggerVolume)** to **Damaged Actor of Apply Damage**. Finally, under **Base Damage**, type in **10**.

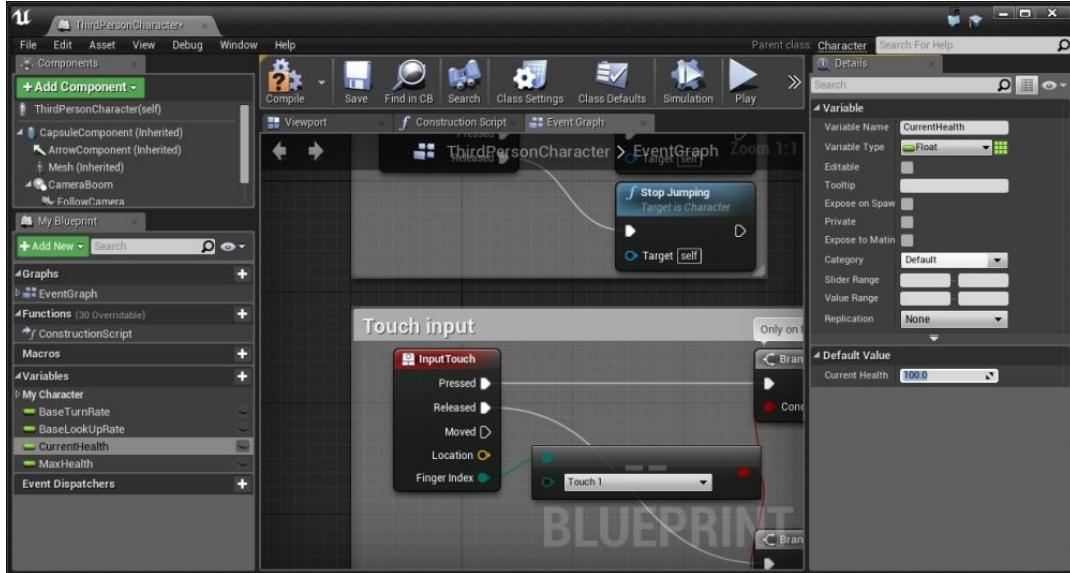


*Blueprint to cause damage event when an object touches the trigger volume*

What this will do is whenever an object touches the trigger volume, it will call a generic damage event on that object with a damage parameter of 10. Note that this will only do something if someone has created an AnyDamage event, which we will learn how to do later on in this recipe.

Now that we have something that can damage the character, we need to actually give our character a way to take damage:

1. Exit out of the level blueprint and from the **Content Browser** tab, open up the `ThirdPersonBP/Blueprints` folder and double-click on the `ThirdPerson Character` object to open it up.
2. From the **MyBlueprint** tab, you'll see a section called **Variables**; extend it out (if it's not already extended) and press the **+** button to create a new variable. When it's created, give it a name (`CurrentHealth`) and then from the **Details** tab, change **Variable Type** value to `Float`.
3. After this, create another float variable called `MaxHealth`.
4. Next, click on the **Compile** button to verify everything is working correctly; then, we can assign a **Default Value** of 100 to each of the two numbers.



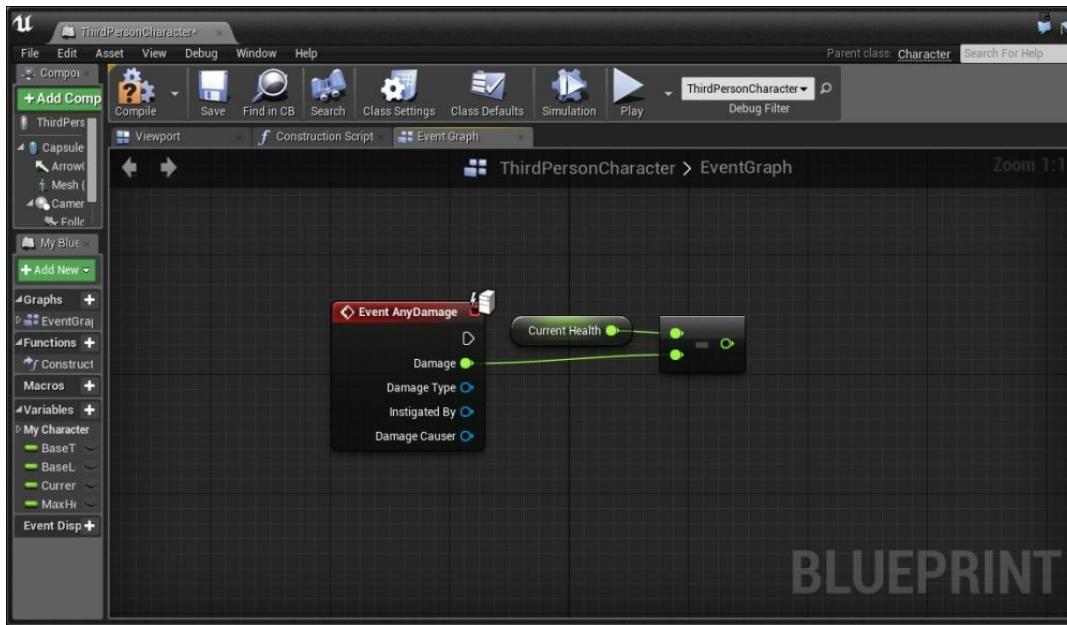
*Assigning Default Value to the numbers*

5. Now that we have the health variables, we can use them. Move down the **Event Graph** until you reach some empty space and right-click to create a new **Event AnyDamage** event (**Add Event | Game | Damage**).

This event is called if the `ApplyDamage` function is called on an object. The default character controllers do not have it, so we need to put it in.

First, we will subtract the damage from the current player's health and then, set the current health to this new value.

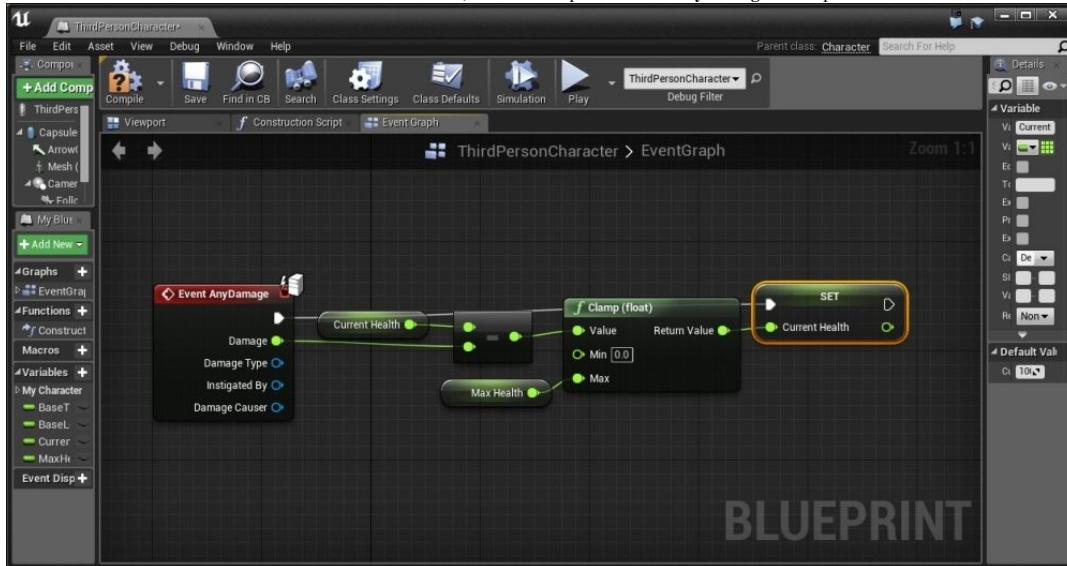
6. Drag and drop the `CurrentHealth` variable from the **Variables** section of the **MyBlueprints** tab into **Event Graph**. From the options, select **Get**.
7. Next, create a `float - float` action (which will subtract a float from another float). On the top, connect the `Current Health` variable. For the bottom, connect the `Damage` value from the **Event AnyDamage** action.



Setting up the final behavior for the damage event

Next, we want to ensure that the value that we get from this, will always be between 0 and our **Max Health** value (we can have someone later on create a trigger with a negative **Damage** value for **Apply Damage**, which will then heal our player). To do this, we can use a **Clamp** action that will make it so that when the number is less than 0, it will set it to 0 or if it is larger than **Max Health**, it'll come down to **Max Health**.

8. Right-click and create a **Clamp(float)** action. Connect **Value** to the output of our subtraction action. Leave the **Min** value at its default of 0, but then for **Max**, connect our **MaxHealth** variable (drag and drop and select **Get**).
9. Now that we have the valid value, we need to actually set this value. To the right of the **Clamp** action, drag and drop a **Current Health** variable, but this time, select the **Set** action. Connect the **Clamp** action's **Return Value** to the **Current Health** variable of the **Set** action. Then, connect the output from **Event Any Damage** to the input of the **Set** action.



Now we have the ability to take or heal damage to our player. Later on in [Chapter 10, User Interface](#), we have the second part of this recipe in which we use this recipe and then create a healthbar to show the player exactly where their health is!

## See also

We've covered quite a lot, but we have only touched the tip of the iceberg, that is, scripting. To give yourself a little exposure to what else can be done with blueprints, take a look at the following tutorials:

- If you're working in teams, having a set of practices will make it much easier for you to work together and perform well. Some recommended best practices for blueprints can be found at <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/BestPractices/index.html>.
- Scott McCutchen has created a tutorial for having a character swap their mesh and animations during gameplay, making it seem like one character poofs out of existence with another taking its place. Check it out at <https://www.youtube.com/watch?v=5R1zrytu7Y>.
- Another more advanced use of blueprints is in the creation of an inventory system. Tom Looman has come up with a tutorial on one that you can check out at <http://www.tomlooman.com/tutorial-basic-inventory-system-in-blueprint/>.
- This tutorial is a bit dated, but Joel Shapiro's blueprints tutorial does go through the creation of a 2D platformer from scratch, so I think it's worth checking it out at <http://www.raywenderlich.com/97058/unreal-engine-tutorial-for-beginners-part-1>.
- Finally, we also have a tutorial from Peter Newton where they build an AI, making use of NavMeshes. While it is a long tutorial, it does go into a number of different aspects of building an AI Bot. Check this out at [https://wiki.unrealengine.com/AI\\_Bot\\_Blueprint\\_Scripting\\_Playlist](https://wiki.unrealengine.com/AI_Bot_Blueprint_Scripting_Playlist).

## Chapter 9. C++ Programming – Gameplay

In this chapter, we'll cover the following recipes:

- Setting up your development environment
- Displaying text during runtime
- Networking 101 – creating collectables with networking
- Saving or loading games and keyboard input with C++
- Creating custom blueprint nodes

# Introduction

So far, we've been using blueprints in order to generate gameplay. This has been great and has worked well, but one of the main advantages that Unreal has over its competition, such as Unity and Cry Engine, is the fact that you can get access to the full source code of the engine and rework it to fit exactly what you're looking for.

It would be quite easy for me to write an entire book just about programming, and within these few pages, I can't possibly cover everything you need to know to write code. Rather, in this chapter, we are going to cover how to set up your development environment and some of the possibilities of programming in C++ for UE4.

## Tip

If you are interested in taking this further and learning how to program C++ using UE4, read *Learning C++ by Creating Games with UE4* by Packt Publishing.

## When to use C++/Blueprints

One of the things that I often hear people asking is whether to use C++ or Blueprints within their projects, or which one is *better*. As it currently stands, there's not really a straightforward answer to it. If you aren't a programmer at all or feel that you're more of a designer, then blueprints may be better for you due to the fact that it's more visually oriented. However, if you've been coding for years, you may be frustrated with having to use so many nodes to do some actions that you could write in a line of code. Factors such as the number of teammates and the project itself can also be taken under consideration.

It's often a lot easier to transition into C++ after you've been using blueprints and Unreal for a while as you'll be used to how Unreal names certain things, but we'll get some exposure in this chapter as well. That being said, you'll often switch between the two within the same project, with level-specific stuff being done in blueprint and content that will be reused in C++.

In previous versions of the Unreal Engine, runtime performance was something you also needed to consider, but it's less of a problem now. Don't worry about it unless something done in blueprint in your game is going to happen lots of times or is causing your game to slow down a lot. If this does happen, check out the guide on improving performance at <https://www.unrealengine.com/blog/how-to-improve-game-thread-cpu-performance>.

Often in game studios, designers will prototype a game mechanic so that it feels right and if it's decided to be used in the game, it's revisited and if it needs to be revised, a programmer can then reimplement it knowing exactly what the designer had in mind while making it as efficient as possible. Lionhead Studios is currently doing this during the development of their new game *Fable Legends*. For more information on that, refer to <https://www.unrealengine.com/showcase/fable-legends>.

# Setting up your development environment

One of the first things that we'll need to do when working in Unreal Engine 4 using C++ is having our **Integrated Developers Environment (IDE)** set up and making it in such a way so that we can run our new code. Let's see how we can do this now.

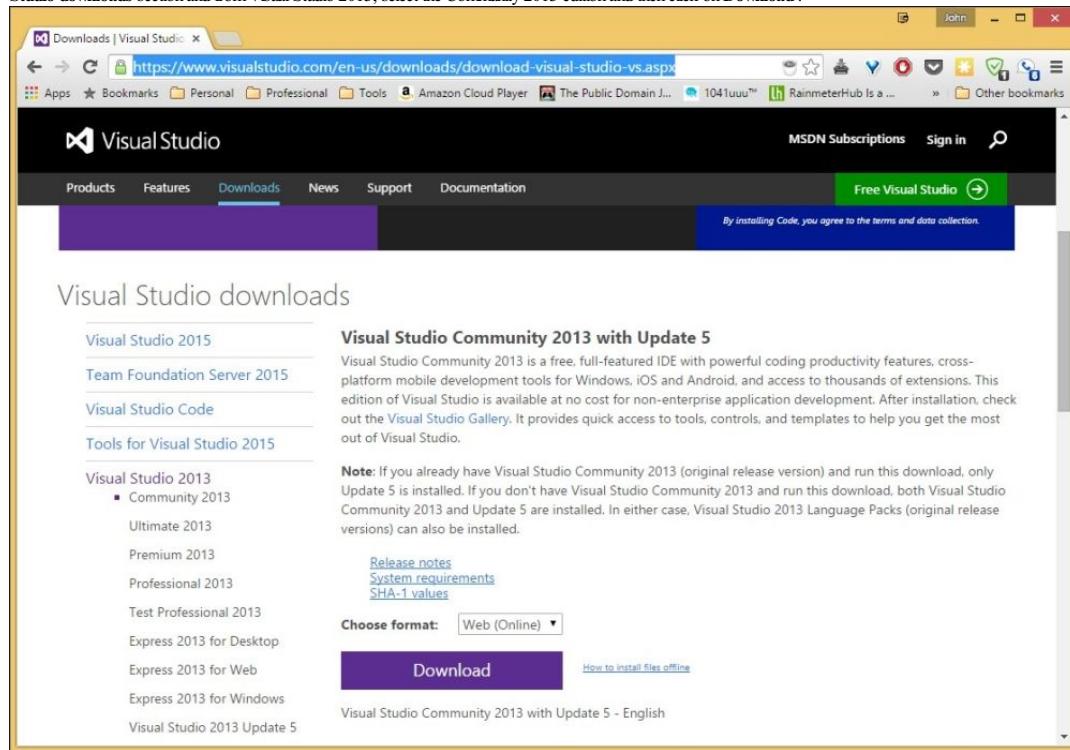
## Getting ready

You'll need to install Visual Studio 2013 (at the time of writing, the 2015 RC version is not supported, so you'll need to install 2013) on Windows or Xcode on Mac to work with this chapter.

## How to do it...

Since I'm working in Windows, I'll be using Visual Studio in which there is a Community edition that is free for students, open source projects, and teams of less than five people. Follow these steps:

1. Open up your web browser and go to the Visual Studio Community's page at <https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx>. Once the page is loaded, scroll down to the **Visual Studio downloads** section and from Visual Studio 2013, select the Community 2013 edition and then click on **Download**.



## Note

Xcode can be downloaded from the Mac App Store for free at <https://itunes.apple.com/us/app/xcode/id497799835>.

2. Once you've got the installer installed, check the agree checkbox and then click on the newly-appeared **Next** button.



- Just keep the initial default settings to install and then click on the **Install** button and wait for it to download the software. This may take a while, so feel free to take a rest and come back later (I had it go overnight, but depending on your Internet connection, it may be completed sooner).

#### Note

For more instructions on setting up Visual Studio to build a project from scratch, visit <https://docs.unrealengine.com/latest/INT/Programming/Development/VisualStudioSetup/index.html>.

- Once Visual Studio is installed, we can then open up the Unreal Editor by clicking on the **Launch** button from the Unreal Engine Launcher.
- Start a new project from the **Project Browser** tab by selecting the **New Project** tab. Click on the C++ tab and then select **Third person** and make sure that **With Starter Content** is selected. Give the project a Name (`Cookbook_Chapter9`). Once you are done, click on **Create Project** and wait for it to finish compiling the project.

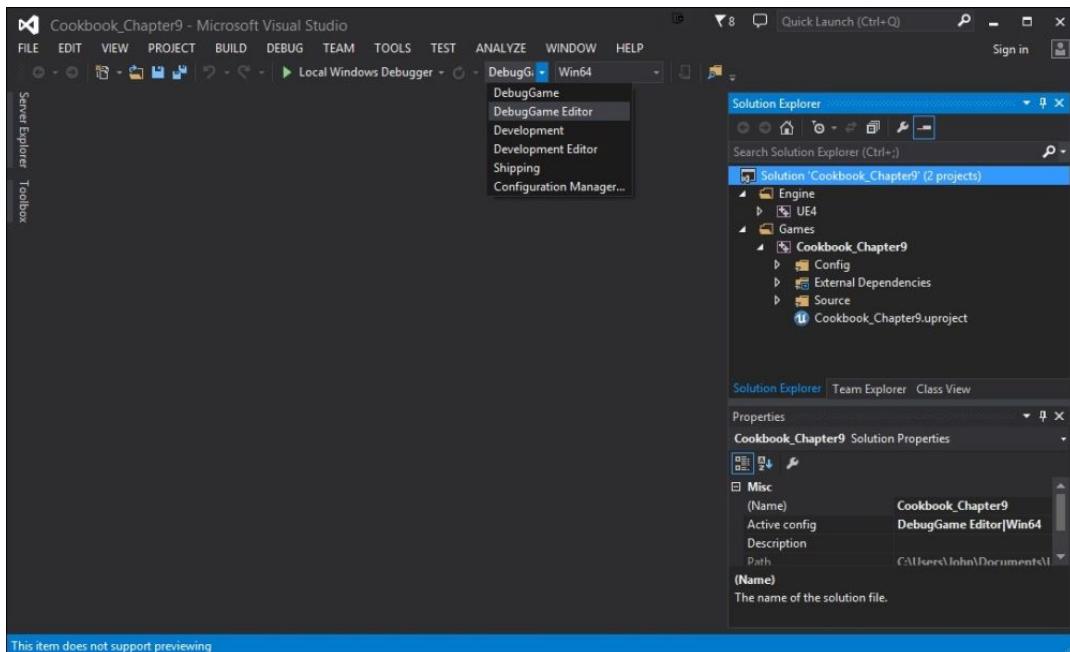
#### Tip

If, on the bottom of your screen, you see a red box saying there was no compiler found, you'll need to install Visual Studio 2013. If that's the case, just follow the same instructions I mentioned earlier.

- Now once we create the project, we'll see that unlike before when the editor opened up by itself, this time Visual Studio or XCode has opened up for us as well.
- If it is your first time opening Visual Studio, you may be asked to sign in. Do so if you'd like to, or click on the **Not now, maybe later** button. Afterward, you may be asked to choose a color theme (I picked **Dark**). Then, click on **Start Visual Studio** and wait for it to finish preparing the program for use.
- Now that we know what's in here, let's set up our project to actually compile correctly. There are two steps that we'll need to look into. Firstly, at the top bar, you should set the **Solution Configurations** property (if you hover your mouse over a part of the toolbar for a short time it will display what property it is) and secondly, under **Solution Configurations**, select **DebugGame Editor**. To see where the property and dropdown are, take a look at the following screenshot:

#### Tip

If you do not see the **Solutions Configurations** property, on the far right, click on the dropdown button and make sure that the property is checked.

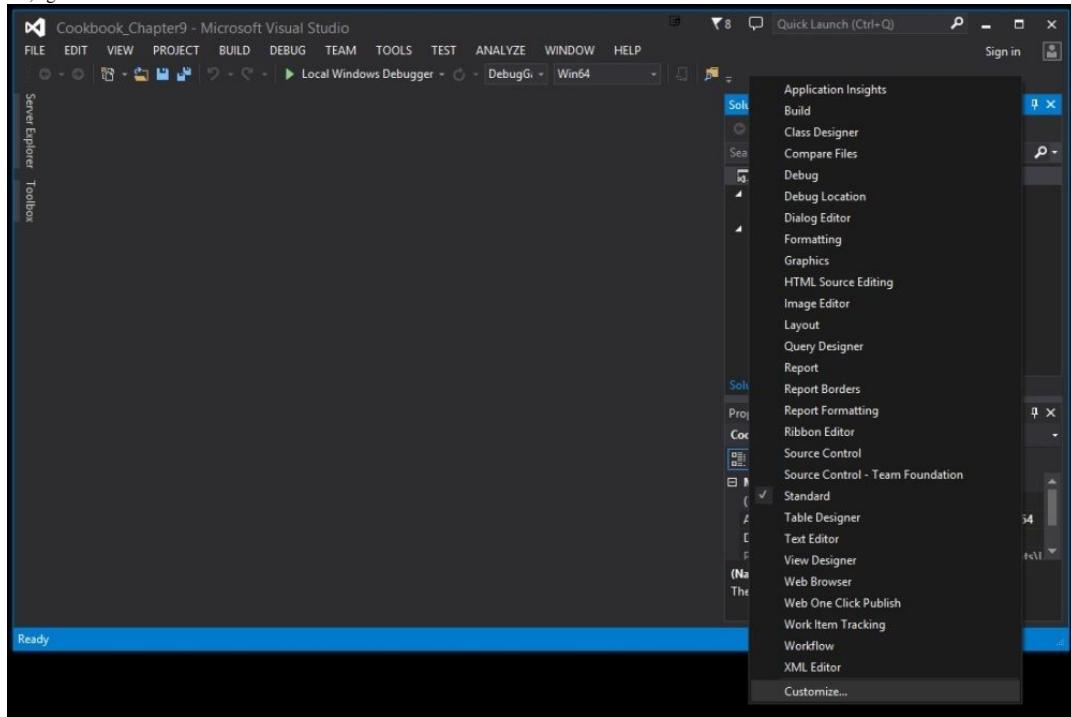


The **Debug Game Editor** mode builds the project with debug symbols enabled for the `Game` code but not the `Editor`. This means that when the game is running, we'll be able to debug it; you can learn more about this at <https://msdn.microsoft.com/en-us/library/k0k771bt.aspx>.

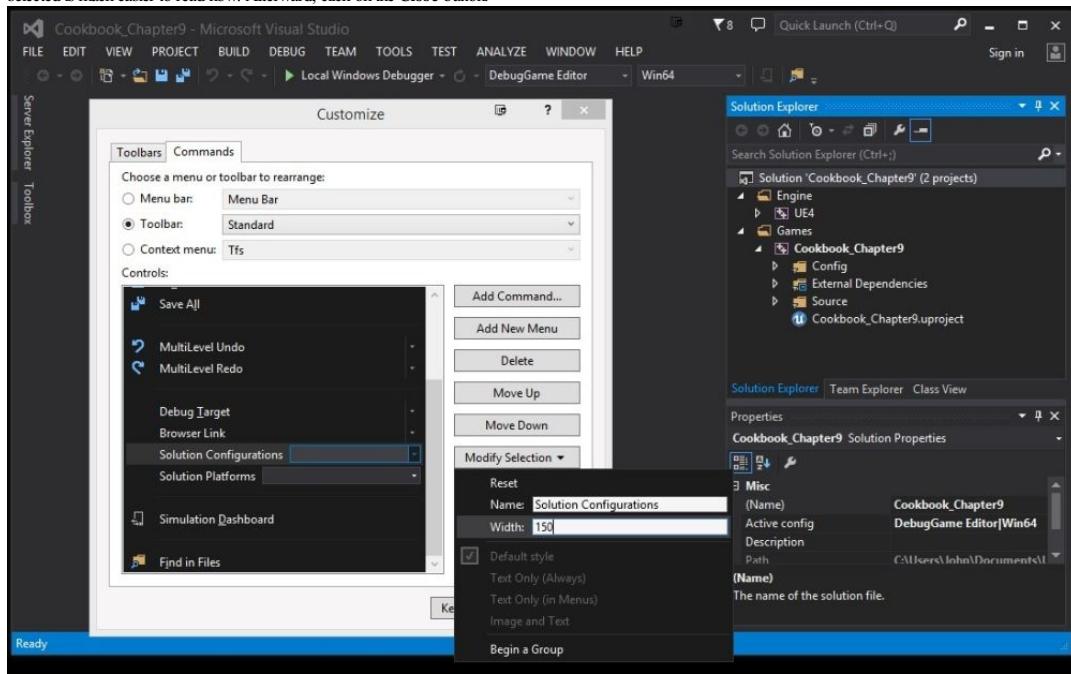
#### Note

For more information on all of the different kinds of configurations, refer to <https://docs.unrealengine.com/latest/INT/Programming/Development/CompilingProjects/index.html#buildconfiguration>.

9. That being said, it's hard for us to tell what that setting is in the default way that Visual Studio is set up, but thankfully, the program is extremely configurable, so let's increase the width of that dropdown menu. To do this, right-click on the toolbar and select **Customize**.



10. From the menu that pops up, select **Commands** and then click on the toggle by **Toolbar**.
11. Select the dropdown next to **Toolbar** and from there, select **Standard**.
12. From the **Controls**: list that pops up from the bottom-right side, scroll down the list until you arrive at the **Solutions Configurations** option and select it.
13. Then, click on the **Modify Selection** dropdown on the right-hand side. From the window that pops up, under **Width:**, type in the value as **150**. You should notice that the area where **DebugGame Editor** is selected is much easier to read now. Afterward, click on the **Close** button.



14. Now, let's run the editor to make sure everything's working correctly. In the **Solution Explorer** tab on the right-hand side, right-click on the project name (**Cookbook\_Chapter9**) and navigate to **Debug | Start New Instance**.
15. It's going to ask whether you want to build the project, so select **Yes**, and our project will be built.

The first thing that's happening here is that the **compiler** is going to compile the code. When compiling the code, it's taking all of the files that are part of the project and translating them from the source code that we've written and converting it into object code (something that computers can understand). This is then taken by a **linker**, which is the tool that will combine all of the modules we've created to produce an executable file.

#### Note

For more information on compiling and linking, refer to <http://www.cprogramming.com/compilingandlinking.html>.

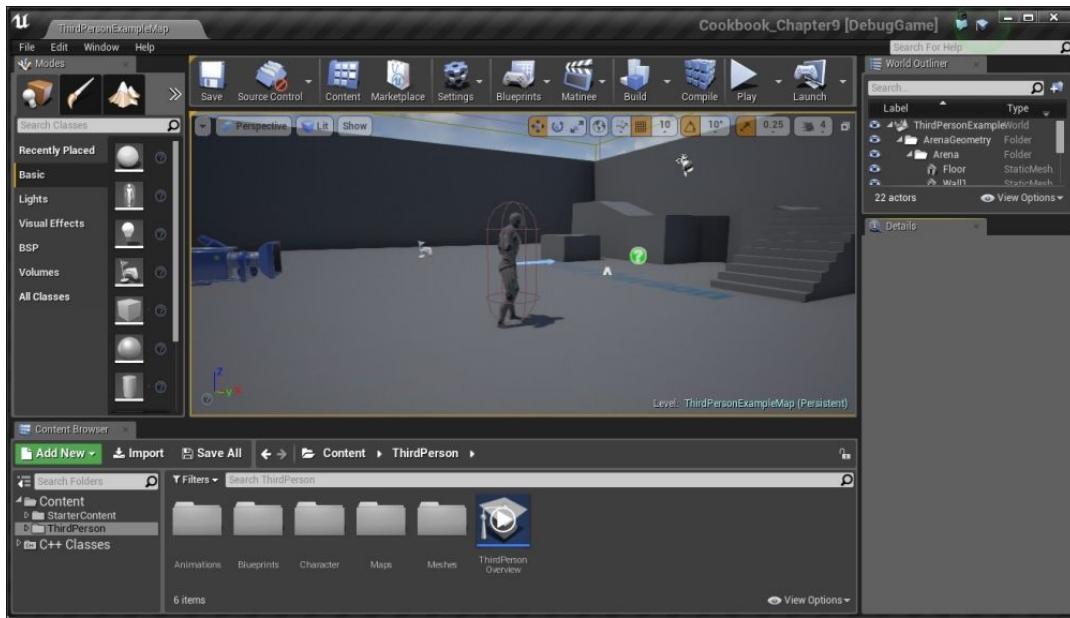
This will take some time when we build a project for the first time, but in future this will be much quicker because it will only compile the files that have changed since the last time you compiled.

#### Note

If you are using Xcode on the Mac side, navigating to **Product | Run** from the menu will launch the UE4 Editor in the **Debug** mode and all the custom C++ code will be available in the editor.

To configure Xcode properly for UE4, make sure the UE4 Editor is selected as the current scheme—`Xcode scheme selector`.

With this, we should see our project open up!



16. Close the editor and return to Visual Studio.

Now before we go on, I want to talk a little bit about the folders and content that are created automatically for us. On the right-hand side, you'll see a tab called **Solution Explorer** and under that, you'll see two folders—**Engine** and **Games**. **Engine** contains the UE4 Solution, which is the Unreal Engine 4 itself in its full glory. For our purposes, we can use it as a reference, but we will not be modifying it, we will rather be extending from it. The next folder, **Games**, contains the games that we are creating using UE4, and in my case, I'll see another solution in bold called **Cookbook\_Chapter9**. Being bolded means that if we click on the Local Windows Debugger button, that'll be the project that'll be built when we tell Visual Studio to compile or run our project. If you extend the solution folder, you'll find the following folders:

- **Config** : This is where all of the configurations that make our game unique will exist, and where settings, input, and so on will live. These files normally have a **.ini** extension.
- **External Dependencies** : This contains all of the headers files for the Unreal Engine. Also known as include files sometimes, header files hold declarations for other files to use without having to see the source code. These files normally have a **.h** extension to them.
- **Source** : Inside it will be a folder with the same name as your project. This is where we can modify how different modules will work with our projects and add additional files for us to work with. These contain headers (**.h**) and source files (**.cpp**).

With this, we now have our environment created to be able to start working in code!

## Displaying text during runtime

While you're creating your project, one of the things that can be valuable to know is how to give yourself information while the game is going on. This way, you can check the order of the things being called and/or what values data have.

Since it was first written 40 years ago, it has been a tradition for beginning programmers to write a function that displays "Hello World!" on the screen. Let's do that now!

### Getting ready

Before we start working on this we need to have a project created and set up. So, follow the previous recipe all the way to completion.

### How to do it...

In this recipe, we will extend and customize the built-in **GameMode** class to do just that:

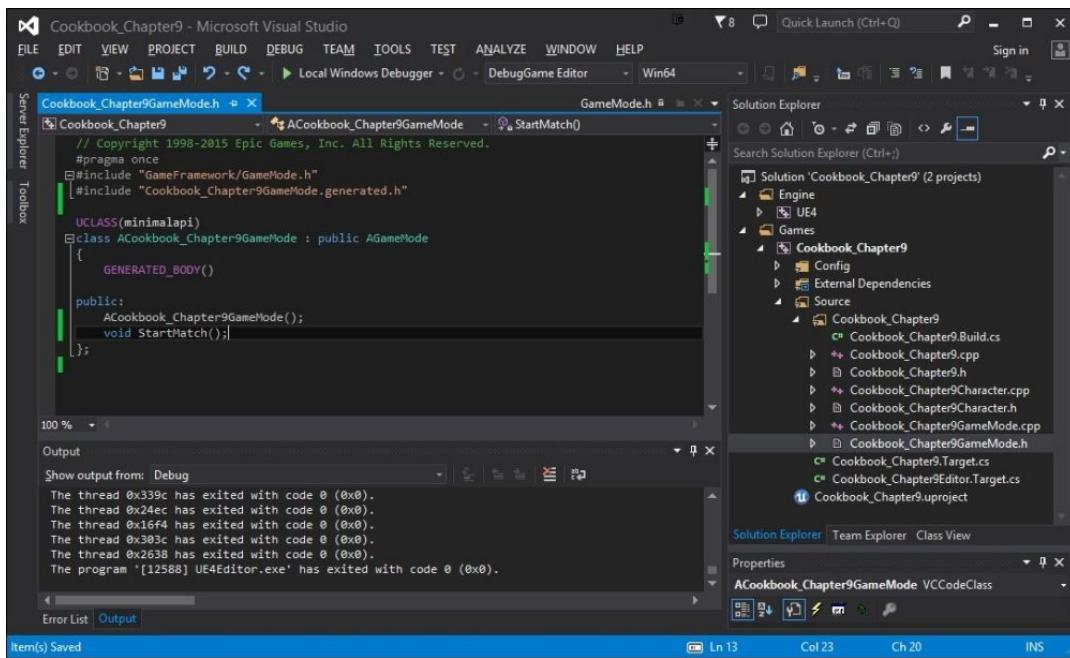
1. With Visual Studio opened, go over to the **Solution Explorer** tab and open up the **Games/Cookbook\_Chapter9/Source/Cookbook\_Chapter9** folder, and you should see a number of files. Double-click on the **Cookbook\_Chapter9GameMode.h** file to open it.

This file contains the information for a class called **ACookbook\_Chapter9GameMode**. A class is a container that holds variables (data) and functions (series of instructions). Think of this class as a blueprint full of information about what this object is.

This is a child class of **AGameMode** which means that it contains everything that **AGameMode** has, and we can add additional information to it as well as extend functions that were created previously, which is what we are going to do.

2. Underneath the line **ACookbook\_Chapter9GameMode();**, add the following line:

```
void StartMatch();
```



This is what's known as a function declaration and it is saying that somewhere else, there is the implementation of this `StartMatch` function, which takes in no parameters and returns nothing (`void`).

3. Next, we need to write the function definition. To do this, from the **Solution Explorer** tab, double-click on the `Cookbook_Chapter9GameMode.cpp` file to open it.
4. Then we need to write the `StartMatch` function, which will look similar to the following:

```
void ACookbook_Chapter9GameMode::StartMatch()
{
    Super::StartMatch();

    if (GEngine)
    {
        GEngine->AddOnScreenDebugMessage(-1, 10.0f,
                                         FColor::Yellow,
                                         "Hello World!");
    }
}
```

This code is first doing what the parent's version did (by using `Super`). Afterward, it is checking whether the game engine exists (`if (GEngine)`), and if it does, it is adding a debug message on the screen that lasts for 10 seconds on the screen and says **Hello World!**. The `f` at the end of the `10.0` stands for float, as in floating point number which means that it can use a decimal value. The `FColor::Yellow` states that the color of the text will be yellow and the `-1` value lets the function know that we don't want to replace a message if one is already there, we just want to create a new one.

#### Note

For more information on the `AddOnScreenDebugMessage` function, refer to <https://docs.unrealengine.com/latest/INT/API/Runtime/Engine/Engine/UEngine/AddOnScreenDebugMessage/index.html>.

After this, you may notice a red squiggly line underneath the `GEngine` part of our code. This is because `GEngine` doesn't exist at this point, or rather it does, but we don't know where it is. To find that variable, we will need to include the `Engine.h` file in our script.

5. At the very top below the other `#include` directives, add the following line:

```
#include "Engine.h" //GEngine
```

The entire file should look similar to the following:

```
// Copyright 1998-2015 Epic Games, Inc. All Rights Reserved.

#include "Cookbook_Chapter9.h"
#include "Cookbook_Chapter9GameMode.h"
#include "Cookbook_Chapter9Character.h"
#include "Engine.h" //GEngine

ACookbook_Chapter9GameMode::ACookbook_Chapter9GameMode()
{
    // set default pawn class to our Blueprinted character
    static ConstructorHelpers::FClassFinder<APawn> PlayerPawnBPClass(TEXT("/Game/ThirdPerson/Blueprints/ThirdPersonCharacter"));
    if (PlayerPawnBPClass.Class != NULL)
    {
        DefaultPawnClass = PlayerPawnBPClass.Class;
    }
}

void ACookbook_Chapter9GameMode::StartMatch()
{
    Super::StartMatch();

    if (GEngine)
    {
        GEngine->AddOnScreenDebugMessage(-1, 10.0f, FColor::Yellow, "Hello World!");
    }
}
```

```

// Copyright 1998-2015 Epic Games, Inc. All Rights Reserved.

#include "Cookbook_Chapter9.h"
#include "Cookbook_Chapter9GameMode.h"
#include "Cookbook_Chapter9Character.h"
#include "Engine.h" //GEngine

ACookbook_Chapter9GameMode::ACookbook_Chapter9GameMode()
{
    // set default pawn class to our Blueprinted character
    static ConstructorHelpers::FClassFinder<APawn> PlayerPawnBPClass(TEXT("/Game/ThirdPerson"));
    if (PlayerPawnBPClass.Class != NULL)
    {
        DefaultPawnClass = PlayerPawnBPClass.Class;
    }
}

void ACookbook_Chapter9GameMode::StartMatch()
{
    Super::StartMatch();

    if (GEngine)
    {
        GEngine->AddOnScreenDebugMessage(-1, 10.0f, FColor::Yellow, "Hello World!");
    }
}

```

### Note

The lines with `//` at the beginning are referred to as **comments**. These are the texts that don't do anything to the code, but are used to help people reading the code understand exactly what something below it is doing. Comments can also be created by placing a `/*`, and until you see a `*/`, whatever is in between is commented.

- After this, we need to compile our code to see any changes. Follow the method we used previously or click on the **Local Windows Debugger** button to start it, saying **Yes** when it asks you to build.
- Once the program opens up, again click on the **Play** button.



Just as we thought, when the game starts, it displays the text "Hello World!". Congratulations on entering the world of programming!

### Note

In addition to printing on the screen, it's often useful to write things to a log file so that you have access to the information wherever you are. For information on that, there is an excellent guide at [https://wiki.unrealengine.com/Logs,\\_Printing\\_Messages\\_To\\_Yourself\\_During\\_Runtime](https://wiki.unrealengine.com/Logs,_Printing_Messages_To_Yourself_During_Runtime).

## Networking 101 – creating collectables with networking

Networking is one of the more complex things you can do as a programmer. Unreal uses a client-server model for communication between multiple computers. In this case, the server is the person who started the game and the clients are those who are playing the game with the first person. In order for things happening on everyone's game to work correctly, we need to call certain code at certain times to certain people.

For example, when a client wants to shoot his/her gun, they send a message to the server, which will then determine whether you hit anything and then tell all the clients what happened using replication. This can be important because some things, such as the Game Mode, only exist on the server.

### Note

For more information on the client-server model, refer to [https://en.wikipedia.org/wiki/Client%20server\\_model](https://en.wikipedia.org/wiki/Client%20server_model).

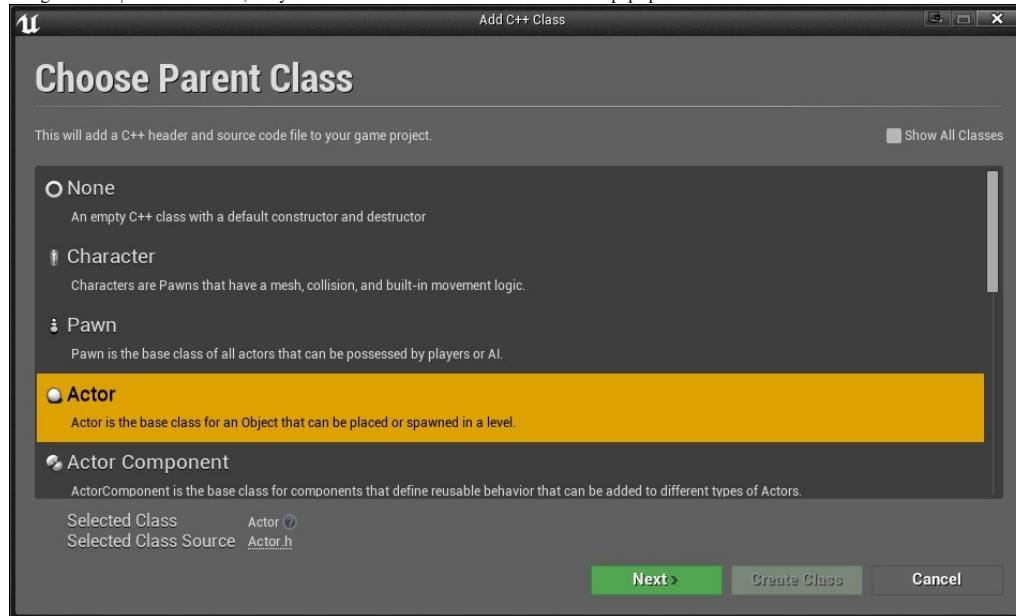
### Getting ready

Before we start working on this, we need to have a project created and set up. Follow the *Setting up your development environment* recipe all the way to completion.

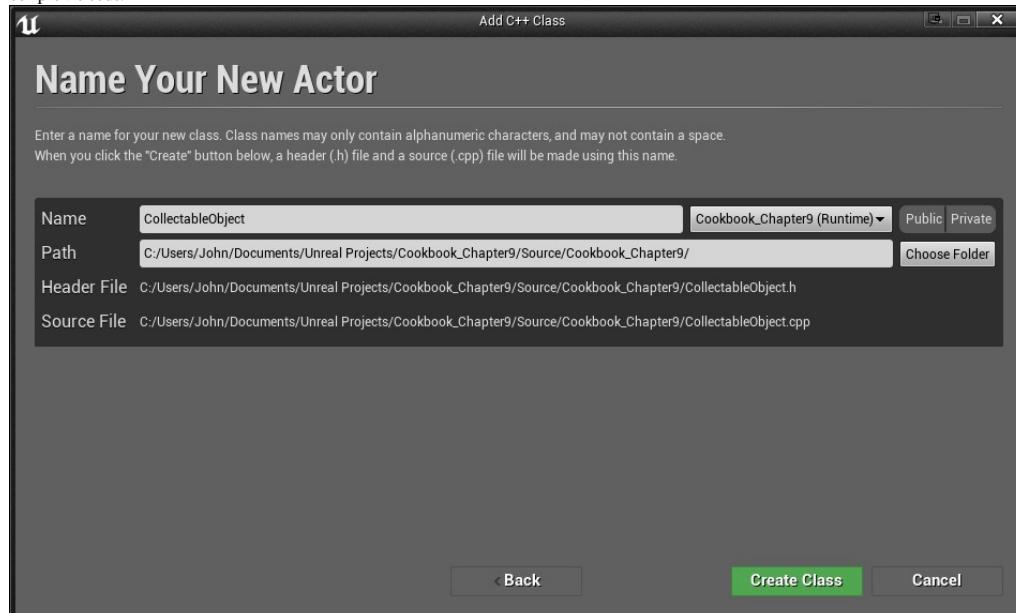
### How to do it...

To give you an idea of how it works, let's do a simple example of a coin collectable:

1. Navigate to **File | New C++ Class**, and you should see the **Choose Parent Class** window pop up. Select **Actor** and then click on **Next**.



2. From there, we then want to set **Name** of our object to something that makes sense; in this case, I named it `CollectableObject`. Once done, click on the **Create Class** button to add it to the project and compile the code.



3. Now, we should see two new files being created inside our Visual Studio project: `CollectableObject.cpp` and `CollectableObject.h`. Open up `CollectableObject.h` and put in the following code:

```
#pragma once

#include "GameFramework/Actor.h"
#include "CollectableObject.generated.h"

UCLASS()
class COOKBOOK_CHAPTER9_API ACollectableObject : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    ACollectableObject(const FObjectInitializer& ObjectInitializer);

    // Event called when something starts to overlap the
    // sphere collider
    UFUNCTION()
    void OnBeginOverlap(class AActor* OtherActor,
                        class UPrimitiveComponent* OtherComp,
                        int32 OtherBodyIndex, bool bFromSweep,
                        const FHitResult& SweepResult);

    // Our server function to update the score.
    UFUNCTION(Reliable, Server, WithValidation)
    void UpdateScore(int32 Amount);
    void UpdateScore_Implementation(int32 Amount);
    bool UpdateScore_Validate(int32 Amount);

};
```

#### Note

Note that there is the `COOKBOOK_CHAPTER9_API` text before `ACollectableObject`. This automatically-generated part of the code uses the same name as our project. If you did not name your project `Cookbook_Chapter9`, the name will be different.

We've done a number of things in this new snippet of code. We've replaced the original constructor with a new one that takes in a `FObjectInitializer` object as a parameter. We've done this so that we can add a new component to the object. The next thing we've done is that we have got rid of the `Tick` and `BeginPlay` functions as we won't be using them.

After that, we've added a new function called `OnBeginOverlap`. We will be talking more about what it does later, but for now, it will be a function that'll be called whenever the object will be overlapped with another object (such as the player). Lastly, we have three functions—`UpdateScore`, `UpdateScore_Implementation`, and `UpdateScore_Validate`—which are used to call, implement, and validate the replication properties as we specified in the `FUNCTION` above its declaration.

## Note

For more information on replication and how the functions are named in their certain way, refer to <https://wiki.unrealengine.com/Replication>.

4. Next, open up the `CollectibleObject.cpp` file and put in the following code:

```
#include "Cookbook_Chapter9.h"
#include "CollectableObject.h"
#include "Engine.h" //GEngine

// Sets default values for this actor's properties
ACollectableObject::ACollectableObject(const FObjectInitializer& ObjectInitializer) : Super(ObjectInitializer)
{
    // Must be true for an Actor to replicate anything
    bReplicates = true;

    // Create a sphere collider for players to hit
    USphereComponent * SphereCollider = ObjectInitializer.CreateDefaultSubobject<USphereComponent>(this, TEXT("SphereComponent"));

    // Sets the size of our collider to have a radius of
    // 64 units
    SphereCollider->InitSphereRadius(64.0f);

    // Sets the root of our object to be the sphere collider
    RootComponent = SphereCollider;

    // Makes it so that OnBeginOverlap will be called
    // whenever something hits this.
    SphereCollider->OnComponentBeginOverlap.AddDynamic(this,
        &ACollectableObject::OnBeginOverlap);
}

// Event called when something starts to overlaps the
// sphere collider
void ACollectableObject::OnBeginOverlap(
    class AActor* OtherActor,
    class UPrimitiveComponent* OtherComp,
    int32 OtherBodyIndex,
    bool bFromSweep,
    const FHitResult& SweepResult)
{
    // If I am the server
    if (Role == ROLE_Authority)
    {
        // Then a coin will be gained!
        UpdateScore(1);
        Destroy();
    }
}

// Do something here that modifies game state.
void ACollectableObject::UpdateScore_Implementation(int32 Amount)
{
    if (GEngine)
    {
        GEngine->AddOnScreenDebugMessage(-1, 5.0f,
            FColor::Green,
            "Collected!");
    }
}

// Optionally validate the request and return false if the
// function should not be run.
bool ACollectableObject::UpdateScore_Validate(int32 Amount)
{
    return true;
}
```

## Note

Again, as before, if you have a different name for your project, the `#include "Cookbook_Chapter9.h"` line will also be different.

This code does a number of different things. The first function has the same name as our object's class and has no return type. This special type of function is known as a `constructor`. This function is special because it's the first thing that gets called when an object is created. In this case, we make sure that our object is going to be replicated, and then after that, we create a sphere collider that we tell (via a listener) to call the `OnBeginOverlap` function when it collides with another object it using the `OnComponentBeginOverlap` function.

## Note

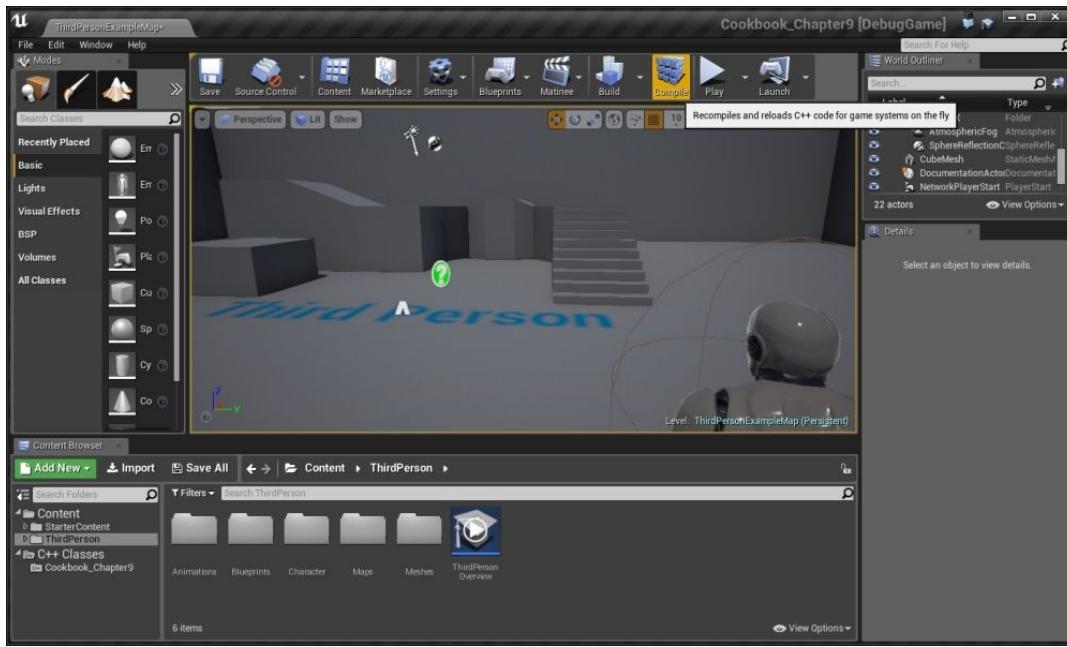
For more information on the `OnComponentBeginOverlap` function and the function needed to be given to it, refer to

<https://docs.unrealengine.com/latest/INT/API/Runtime/Engine/Components/UPrimitiveComponent/OnComponentBeginOverlap/index.html>.

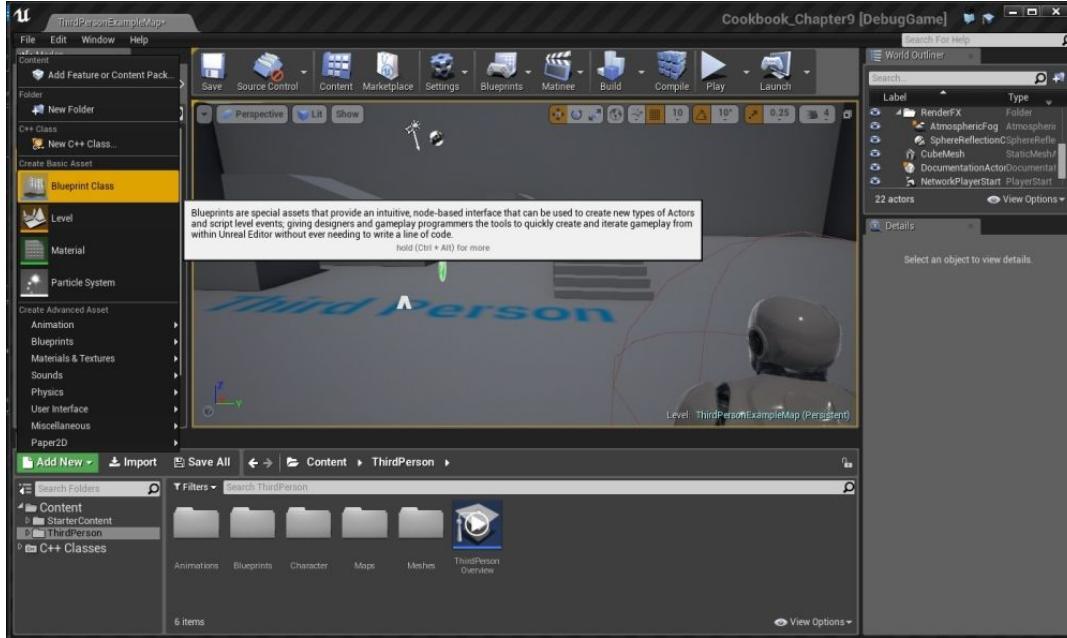
After this, inside our `OnBeginOverlap` function, we first check if we are currently on the server. We don't want things to get called multiple times, and we want the server to be the one that tells the other clients that we've increased our score. We also call the `UpdateScore` function, which will, in turn, call the `UpdateScore_Implementation` function that we created and it will display a message, saying that we've collected the object via printing out some text like we used earlier.

Finally, the `UpdateScore_Validate` function is required for us to have and just tells the game that we should always run the implementation for the `UpdateScore` function.

5. Save both files and go back into the Unreal Editor. From there, click on the `Compile` button in order to update the game with the changes we've made without having to close the game.



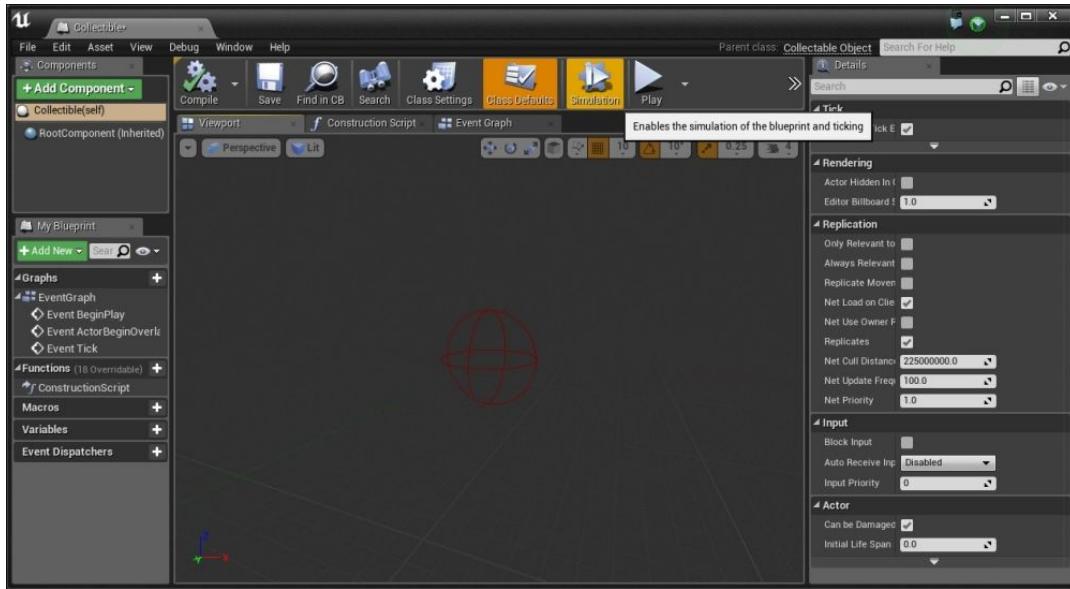
6. Once you see a message at the bottom-right of the screen that says the compilation is complete with no errors, we can then actually add our collectable objects to the scene. From the **Content Browser** tab, select the **ThirdPerson** folder and then click on the green **Add New** button and select **Blueprint Class**.



7. From the popup asking you to select the parent class, go to the **All Classes** section at the bottom and start typing in `CollectableObject`, then select it from the list. Once selected, click on the **Select** button.



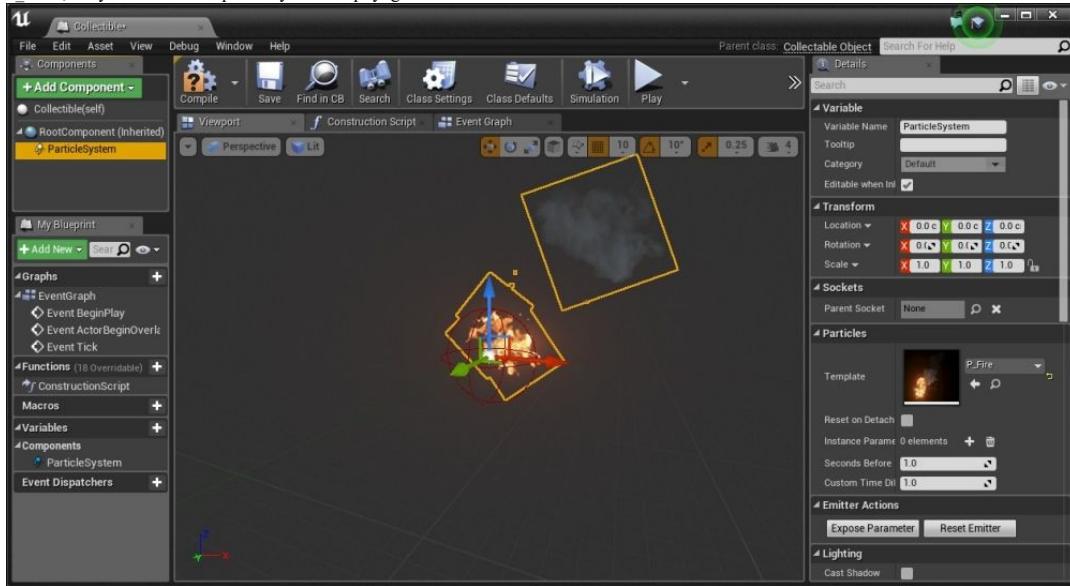
8. After this, it will have the newly-created object selected, which will need a name. Type in `Collectible` and then press *Enter*.  
 9. Double-click on our newly-created `Collectible` object to open up the Blueprints Editor.



Blueprint editor for the Collectible object

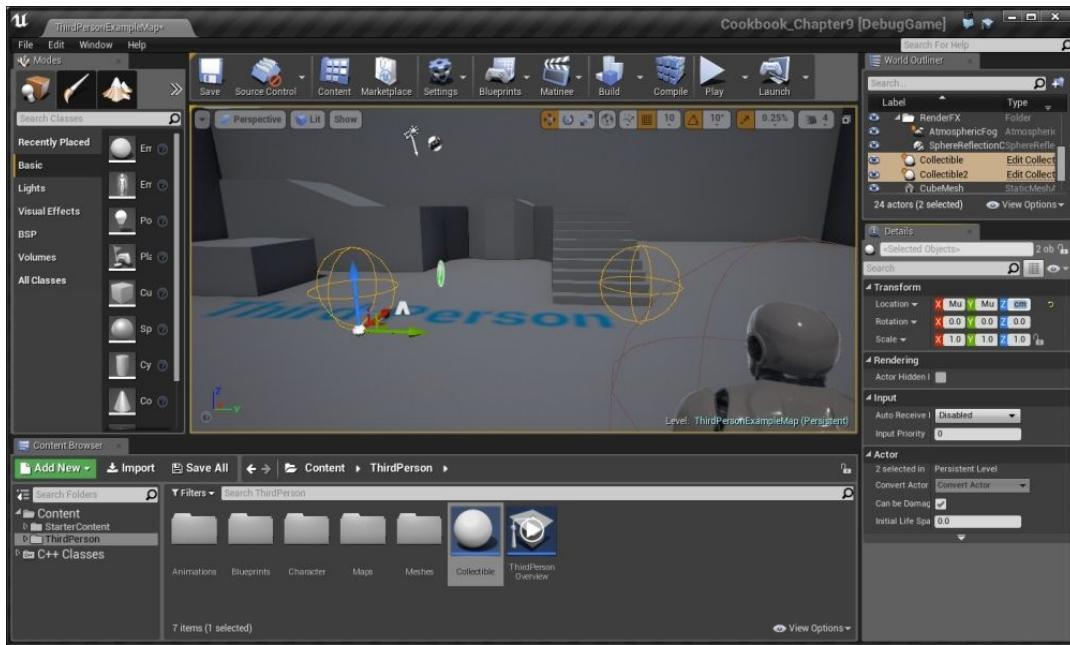
You'll notice that there is a red sphere in the level, just like we wanted to be created in our class file! Right now though, it'll almost be impossible for us to hit it because we can't see anything. Let's add a particle system to make it easy to see.

- From the **Components** tab, select **Add Component** and then **Particle System**. From the **Details** tab under the **Particles** section, select the dropdown to the right of the **Template** property and then select **P\_Fire**, and you should see the particle system start playing.



Adding a fire particle system to the collectible

- Hit the **Compile** and **Save** buttons and then close the Blueprint Editor. After this, drag and drop two of the objects in the world and set their **Z Location** to 200 so that you can see the whole thing.



The level view after compilation

12. Lastly, we'll need to have a way to play two versions of the project at once to see what happens. Click on the dropdown to the right of the Play button and change **Number of Players** to 2 and then click on the **New Editor Window** option, which will open two windows with players.



Opening two simultaneous windows

And with this, you can see the message is replicated from the server to the client!

#### Note

If you're interested in seeing another example of using networking and replication, refer to <https://wiki.unrealengine.com/Networking/Replication>.

In addition, you can also check out the Shooter Game example project included with Unreal Engine 4 and read the files to get a feeling for how it's used in a complete example.

## Saving or loading games and keyboard input with C++

As games get more and more complex and longer and longer, players will often need to play a game within multiple sessions. However, by default, players will need to start over from scratch. In this recipe, we will be going over how to save a variable and load it at runtime.

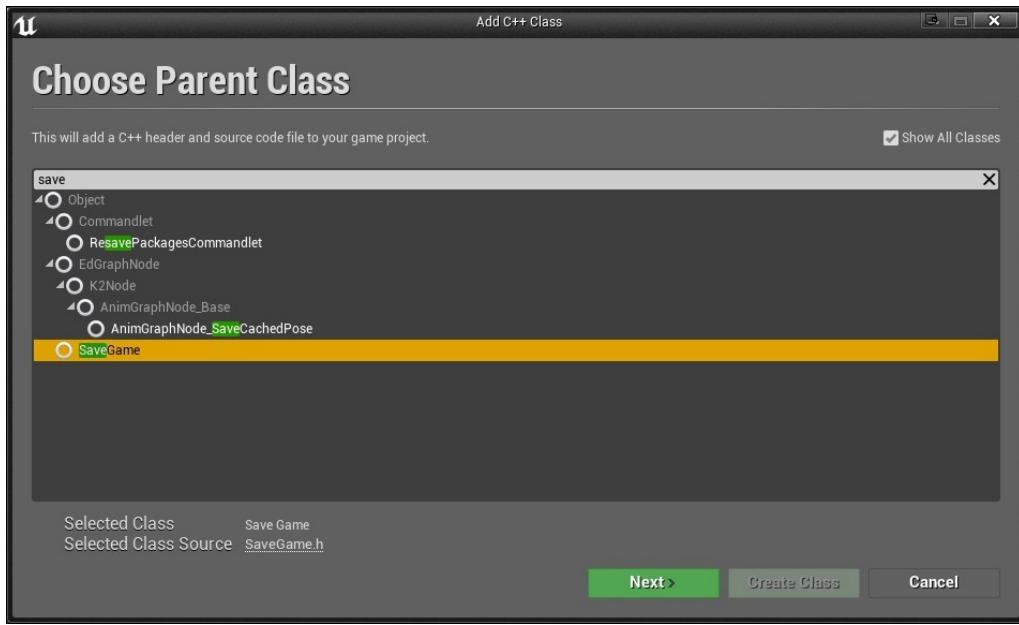
### Getting ready

Before we start working on this, we need to have a project created and set up. Follow the *Setting up your development environment* recipe all the way to completion.

### How to do it...

To give you an idea of how it works, let's do a simple example of saving a player's position and rotation, which we can return to using keyboard input:

1. Navigate to **File | New C++ Class**, and you should see the **Choose Parent Class** window pop up. Check the **Show All Classes** option, select **SaveGame**, and click on **Next**.



2. Next, it'll ask you for the name of your save class. In this instance, I'm going to leave it as `MySaveGame` and click on **Create Class** and wait for it to finish compiling.
3. In the header file for the object, we'll need to add any variables that we want to save. In this case, I want to save my player's position and rotation, so I'm going to do the following:

```
#pragma once

#include "GameFramework/SaveGame.h"
#include "MySaveGame.generated.h"

/** 
 */
UCLASS()
class COOKBOOK_CHAPTER9_API UMySaveGame : public USaveGame
{
    GENERATED_BODY()

public:

    UPROPERTY(VisibleAnywhere, Category = Basic)
    FVector PlayerPosition;

    UPROPERTY(VisibleAnywhere, Category = Basic)
    FRotator PlayerRotation;

};
```

The `PlayerPosition` variable is of type `FVector`, more commonly referred to as a vector. A **vector** is a series of three values, X, Y, and Z, which we've been using already for every object's position, and scaled in the appropriate axis. The rotation of an object is stored in a special type called `FRotator`.

4. We aren't doing anything special inside the .cpp file, but that would be where we would have set any default values. Instead, open up the `Cookbook_Chapter9Character.h` file. Under the `public` section of the class, add in the following bolded code:

```
// Copyright 1998-2015 Epic Games, Inc. All Rights Reserved.
#pragma once
#include "GameFramework/Character.h"
#include "Cookbook_Chapter9Character.generated.h"

UCLASS(config=Game)
class ACookbook_Chapter9Character : public ACharacter
{
    GENERATED_BODY()

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

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

public:
    ACookbook_Chapter9Character();

    /** Base turn rate, in deg/sec. Other scaling may affect
     final turn rate. */
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category=Camera)
    float BaseTurnRate;

    /** Base look up/down rate, in deg/sec. Other scaling may
     affect final rate. */
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category=Camera)
    float BaseLookUpRate;

    // Saves the game
```

```
void SaveMyGameFile();
```

```
// Loads the game
```

```

void LoadMyGameFile();

// Called every frame

void Tick(float DeltaTime);

protected:

/** Called for forwards/backward input */
void MoveForward(float Value);

// Other declarations below

```

5. Next, we need to implement the functionality in the `MySaveGame.cpp` file. At the top, we need to add in some includes, so below the others, add the following:

```

#include "MySaveGame.h"

#include "Kismet/GameplayStatics.h"

#include "Engine.h" //GEngine

```

This will allow us to use the `MySaveGame` class and `GEngine` inside of the file.

6. After that we need to enable the ability for the `Tick` function to be called. To do this, go to the constructor of the `Character` class (`ACookbook_Chapter9Character`) and add the following bold line:

```

ACookbook_Chapter9Character::ACookbook_Chapter9Character()
{
    PrimaryActorTick.bCanEverTick = true;

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

    // Other code below
}

```

7. Now we can implement the `Tick` function:

```

void ACookbook_Chapter9Character::Tick(float DeltaTime)
{
    APlayerController * PController =
        Cast<APlayerController>(Controller);

    // Any time we cast, we need to check if the variable is valid
    if (PController != NULL)
    {
        // If Q is pressed, save the game
        if (PController->WasInputKeyJustPressed(EKeys::Q))
        {
            SaveMyGameFile();
        }
        // Otherwise, if E is pressed we will load
        else if (PController->WasInputKeyJustPressed(EKeys::E))
        {
            LoadMyGameFile();
        }
    }
}

```

The `Tick` function is called in every frame in the game (at 60 FPS, the game is running 60 frames per second), so it'll happen quite often, which is important when it comes to checking input. If the player presses the `Q` or `E` key, we will call the `Save` and/or `Load` functions, respectively.

8. After that, let's implement the function that will save the game:

```

void ACookbook_Chapter9Character::SaveMyGameFile()
{
    // Create a save object for us to store values
    UMySaveGame* SaveGameInstance = Cast<UMySaveGame>(UGameplayStatics::CreateSaveGameObject(UMySaveGame::StaticClass()));

    // Set the player's current location and rotation
    SaveGameInstance->PlayerPosition = GetActorLocation();
    SaveGameInstance->PlayerRotation = GetActorRotation();

    //Saves the new save file into the first save slot (0) with
    // a name of "SaveSlot"
    UGameplayStatics::SaveGameToSlot(SaveGameInstance, "SaveSlot", 0);

    if (GEngine)
    {
        // Notify the player that we saved
        GEngine->AddOnScreenDebugMessage(-1, 10.0f,
            FColor::Yellow, "Game Saved!");
    }
}

```

9. Finally, we need to implement the way to load the game:

```

void ACookbook_Chapter9Character::LoadMyGameFile()
{
    // Create a save object for us to store values to load
    UMySaveGame* LoadGameInstance = Cast<UMySaveGame>(UGameplayStatics::CreateSaveGameObject(UMySaveGame::StaticClass()));

    // Loads the save slot we created previously
    LoadGameInstance = Cast<UMySaveGame>(UGameplayStatics::LoadGameFromSlot("SaveSlot", 0));

    //Set the player's location and rotation to what we saved
    SetActorLocationAndRotation(LoadGameInstance->PlayerPosition,
        LoadGameInstance->PlayerRotation);
}

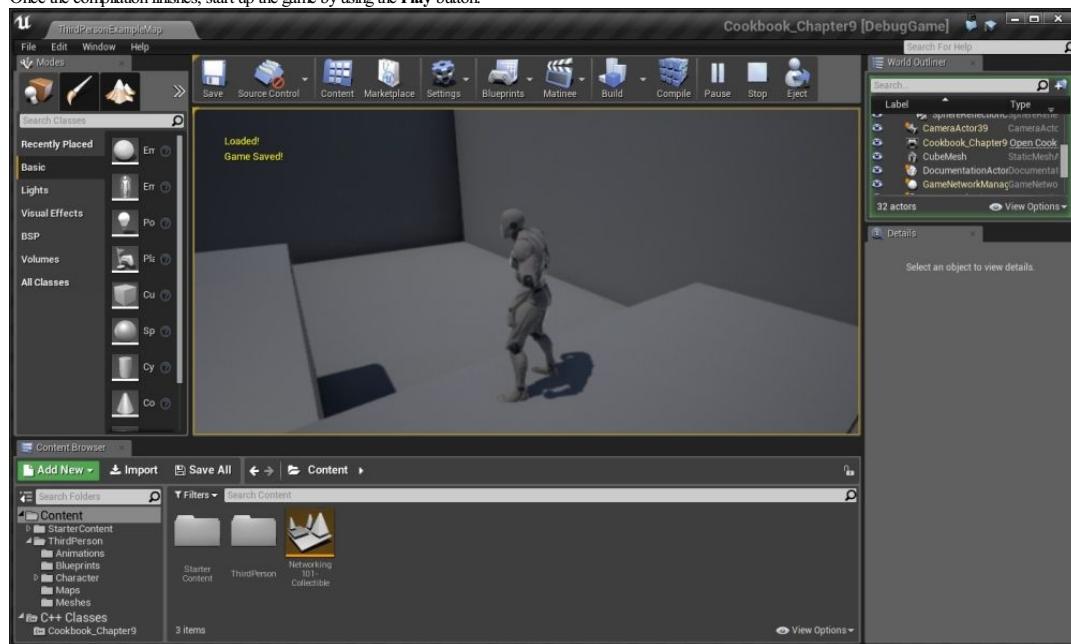
```

```

if (GEngine)
{
    // Notify the player that we loaded
    GEngine->AddOnScreenDebugMessage(-1, 10.0f,
        FColor::Yellow, "Loaded!");
}

```

10. With all that done, save all of the files, go back into the Unreal Editor and then hit the **Compile** button.
11. Once the compilation finishes, start up the game by using the **Play** button.



*Starting the game with Play button*

Now, whenever you press the *Q* button, your position and rotation will be saved and if you walk around and then hit the *E* key, you'll be brought exactly to where you last left it!

#### Note

If you're interested in taking this to the next level, there is a tutorial on how to do file management, being able to create and modify files on the player's hard drive at [https://wiki.unrealengine.com/File\\_Management,\\_Create\\_Folders,\\_Delete\\_Files,\\_and\\_More](https://wiki.unrealengine.com/File_Management,_Create_Folders,_Delete_Files,_and_More).

## Creating custom blueprint nodes

Blueprint is very powerful and has access to most of the functions that are created for a particular class, but there may be certain pieces of code that you want to be always available for use inside Blueprints. Thankfully, we can make use of a custom class extending from Blueprint Function Library to do so.

### Getting ready

Before we start working on this, we need to have a project created and set up. Follow the *Setting up your development environment* recipe all the way to completion.

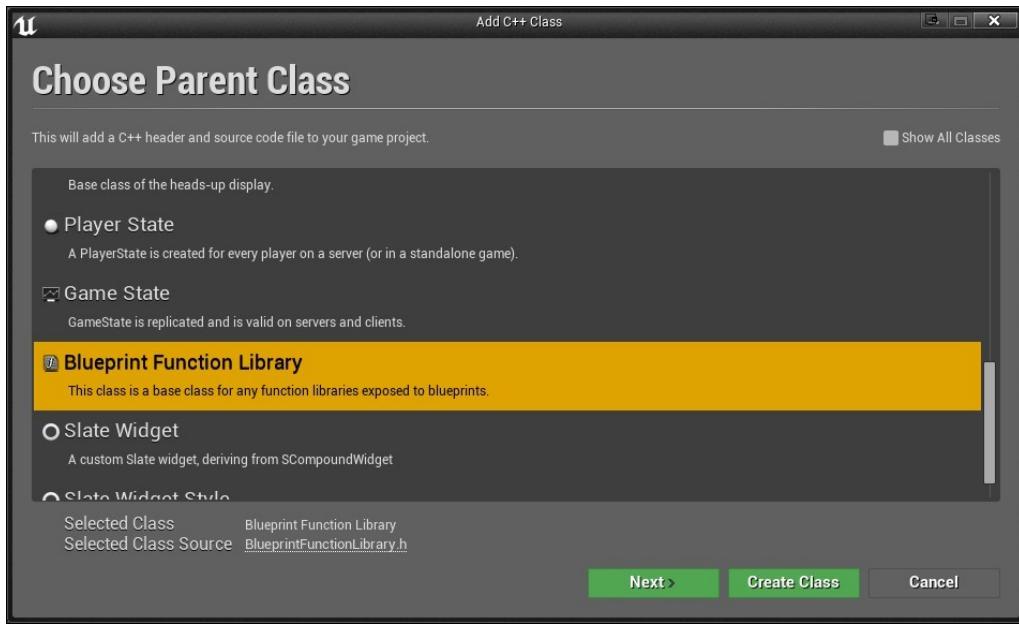
### How to do it...

Often games will display a copyright notice on the main menu. We also do this at the beginning of every class that we create at the header with it, by default, displaying this:

```
//Fill out your copyright notice in the Description page of
//Project Settings.
```

Rather than typing it in every time we want to show it, let's create a note that will get it for us and in the process, teach us how we can create custom blueprint nodes that can do whatever actions we'd like in the future:

1. Navigate to **File | New C++ Class**, and you should see the **Choose Parent Class** window pop up. Scroll down and select **Blueprint Function Library** and then click on **Next**.



2. Select the name you'd like for this class and then click on the **Create Class** button and wait for it to compile.
3. Once compiled, go into the .h file and set it to the following code:

```
#pragma once

#include "Kismet/BlueprintFunctionLibrary.h"
#include "MyBlueprintFunctionLibrary.generated.h"

UCLASS()
class COOKBOOK_CHAPTER9_API UMyBlueprintFunctionLibrary : public UBlueprintFunctionLibrary
{
    GENERATED_BODY()

public:
    // Create the Blueprint node and give it a name and category.
    // The next line is the function that is associated with this
    UFUNCTION(BlueprintPure, meta = (DisplayName = "Copyright Notice", CompactNodeTitle = "Copyright Notice"), Category = "Project Settings")
    static FString GetCopyrightNotice();

};
```

4. Now we need to implement the function in the .cpp file:

```
#include "Cookbook_Chapter9.h"
#include "MyBlueprintFunctionLibrary.h"

FString UMyBlueprintFunctionLibrary::GetCopyrightNotice()
{
    FString CopyrightNotice;

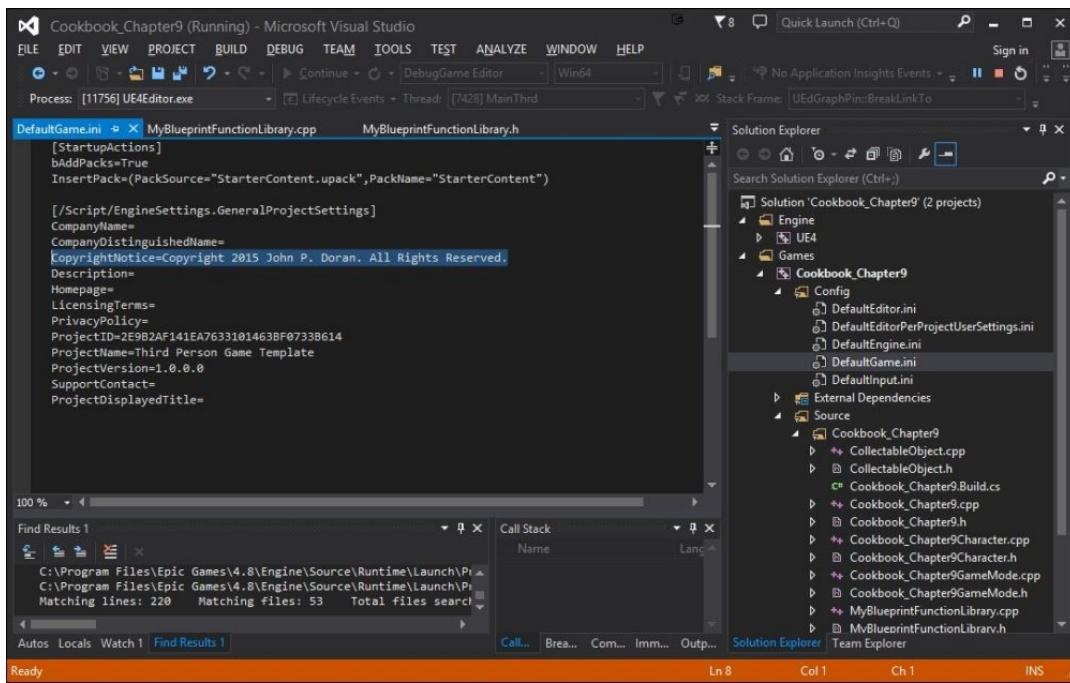
    // Get a string from a configuration file.
    GConfig->GetString(
        // Location of the string you want.
        TEXT("//Script/EngineSettings.GeneralProjectSettings"),
        // Name of the string you want.
        TEXT("CopyrightNotice"),

        // Variable you want to put the string in.
        CopyrightNotice,
        // Which ini file do you want to grab from
        GGameini
    );

    // Return the value of the variable ProjectVersion you've
    // just set.
    return CopyrightNotice;
}
```

5. Go to **Edit | Project Settings**, and you should have a window pop up that has a number of different properties you can assign. For our purposes, scroll down to **Legal** and under **Copyright Notice**, put in what you'd like your copyright to say. In my case, I'll change it to **Copyright 2015 John P. Doran. All Rights Reserved.**

Once you leave the window, these properties are actually saved to the **DefaultGame.ini** file, which is located in the **Config** folder in the **Solution Explorer** tab.



As you can see, the property name is `CopyrightNotice` and it's located in the `/Script/EngineSettings.GeneralProjectSettings` section, just as we did in the code we created earlier.

#### Note

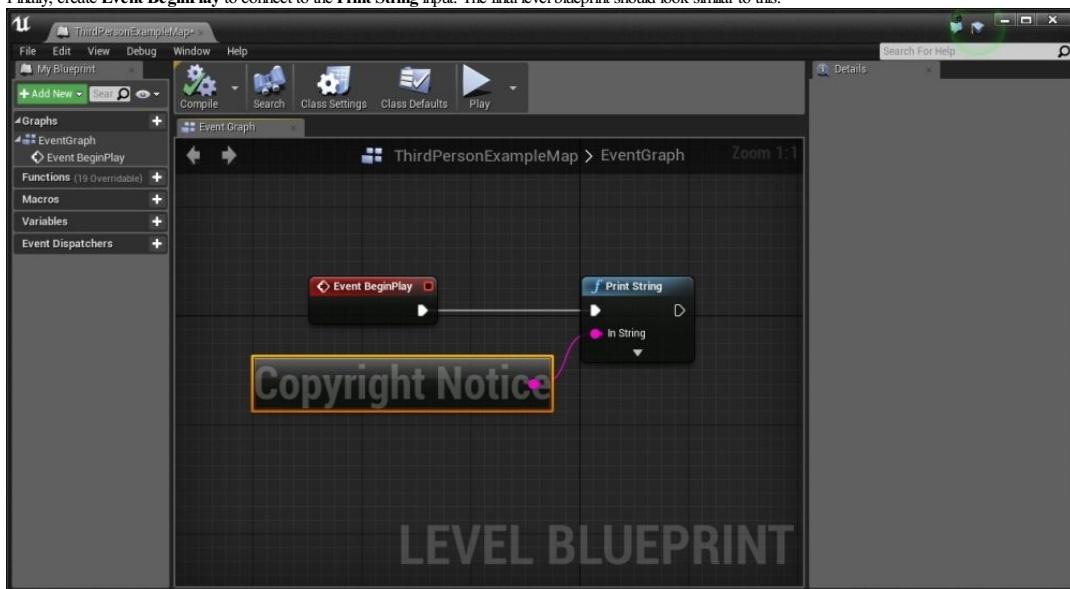
For more information on reading and writing values from config files, refer to [https://wiki.unrealengine.com/Config\\_Files,\\_Read\\_%26\\_Write\\_to\\_Config\\_Files](https://wiki.unrealengine.com/Config_Files,_Read_%26_Write_to_Config_Files).

6. Now that our code is complete, save all of the files and compile the project.
7. After it compiles correctly, let's use our newly-created blueprint node. Click on the **Blueprints** dropdown and select **Open Level Blueprint**.
8. Right-click and search for `Copyright Notice`, and you'll see that there is a node located in our custom-made **Project Settings** category called `Copyright Notice`. Select it, and you'll see it added to the screen with some large text displayed.

#### Note

This text displayed is the content of the `CompactNodeTitle` property. If left blank, it will show up like a normal node. Also notice that the text displayed when you highlight the node is the comment above the `FUNCTION` line. Keep that in mind in the future when you want to provide details to users about what your action does.

9. Let's display this string. So, to do this, right-click on the right-hand side of the `Copyright Notice` node and select a **Print String** action. Connect the output of the `Copyright Notice` to the **In String** variable of the **Print String** node.
10. Finally, create **Event BeginPlay** to connect to the **Print String** input. The final level blueprint should look similar to this:



11. Compile and close the level blueprint and from there, start the game!



*Game view after compilation of level blueprint*

With this, our copyright is displayed!

#### Note

For more examples of how to create custom blueprint notes, refer to [https://wiki.unrealengine.com/Custom\\_Blueprint\\_Node\\_Creation](https://wiki.unrealengine.com/Custom_Blueprint_Node_Creation).

#### See also

Programming is a way of life, and you can spend years becoming better and better at it. Once you feel comfortable with the content in this chapter, take a look at some additional tutorials and/or resources, which will hopefully be useful for you in the future:

- Epic's official programming guide goes through a number of simple examples, introducing content about the API as well as the way the architecture of the engine is built. Check it out at <https://docs.unrealengine.com/latest/INT/Programming/index.html>.
- It may be outdated in terms of images, but the content of this tutorial goes through the steps of creating an FPS game from scratch. It can be found at [https://wiki.unrealengine.com/First\\_Person\\_Shooter\\_C%2B%2B\\_Tutorial](https://wiki.unrealengine.com/First_Person_Shooter_C%2B%2B_Tutorial).
- A number of official C++ Programming tutorials can be found at <https://docs.unrealengine.com/latest/INT/Programming/Tutorials/index.html>.
- For those wanting to take their networking game further, there is another tutorial by Tom Looman about creating a survival game in the same vein as *Day Z*. You can find this at <http://www.tomlooman.com/survival-sample-game-for-u4/>.

## Chapter 10. User Interface

In this chapter, we'll cover the following recipes:

- Create a Health/Damage system, part 2 – creating a healthbar
- Dynamic enemy healthbars
- Creating a main menu
- Animating a menu

## Introduction

In order to create a good game project, you need to be able to communicate information to the player. To do this, we need to create a **user interface ( UI )**, which will allow us to display information such as the player's health, inventory, and others.

Inside Unreal 4, we use the Slate UI framework to create user interfaces, however, it's a very complex system. To make things easier for end users, Unreal also released the **Unreal Motion Graphics ( UMG )** UI Designer, which is a visual UI authoring tool with a much easier workflow. This is what we will be using in this chapter.

#### Note

For more information on Slate, refer to <https://docs.unrealengine.com/latest/INT/Programming/Slate/index.html>.

## Creating a Health/Damage system, part 2 – creating a healthbar

Something that you'll be using often as a game developer is some kind of healthbar. In this recipe, we will learn how to display a healthbar on the screen that will update based off of the values of variables inside of blueprints.

### Getting ready

Before we start working on this, we need to have a project created and set up for our character. To do this, complete the *Creating a Health/Damage system, part 1 – taking damage* recipe in [Chapter 8, Blueprint Scripting – Level Effects](#).

### How to do it...

Let's see how we can use the UMG editor to display a healthbar:

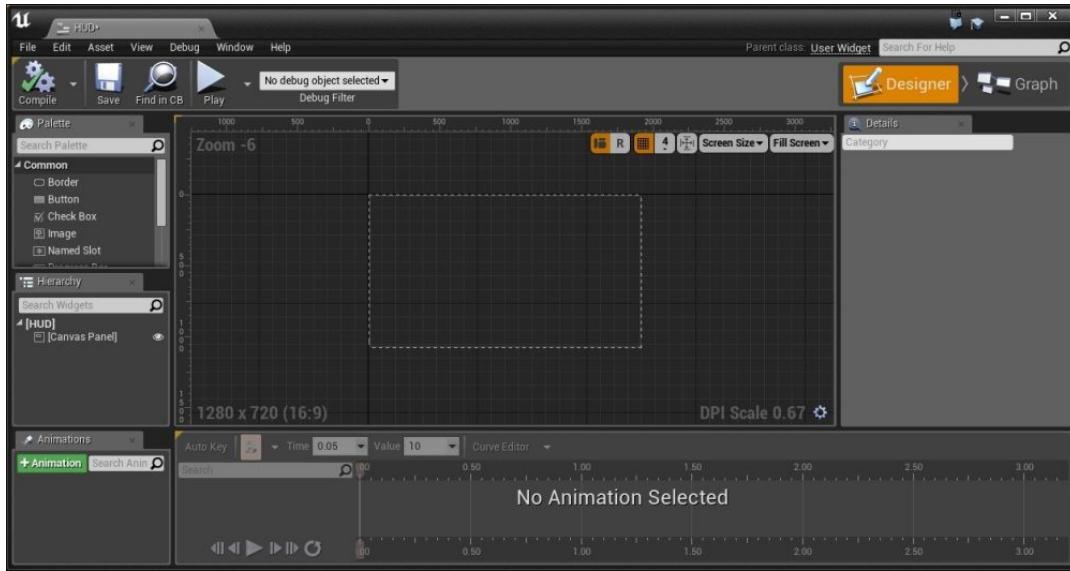
1. From the **Content Browser** tab, select the **Content** folder and then click on the **Add New** button and select **New Folder**. Name this new folder **UI**.
2. Click on the **Add New** button once again and then go to **User Interface | Widget Blueprint**. When it comes up, name it **HUD** (short for Heads Up Display).

The **Widget Blueprint** is a tool that can be used to place all of our UI elements into a scene.

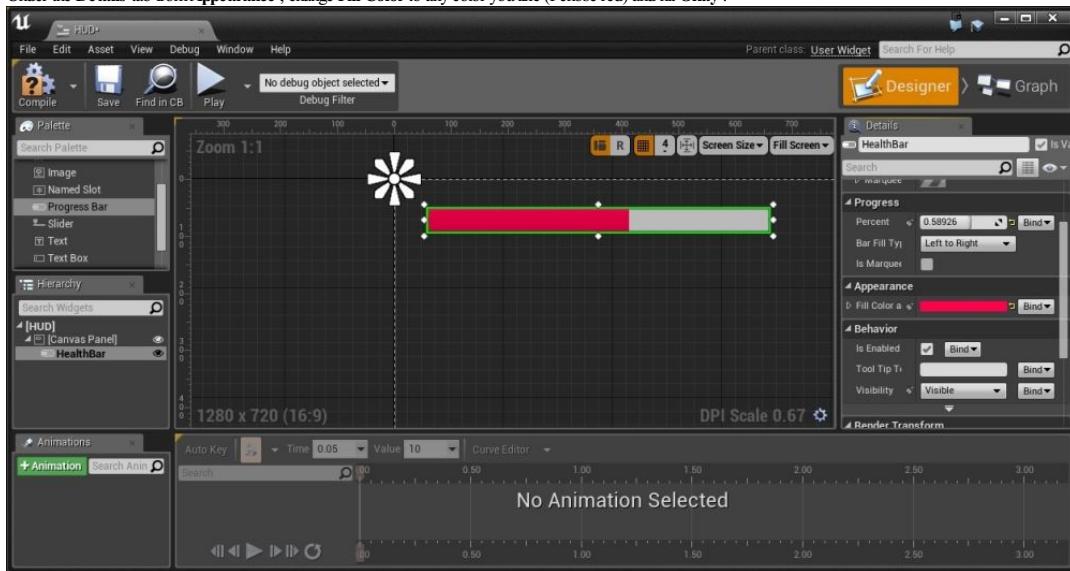
#### Note

For more information on **Widget Blueprints**, refer to <https://docs.unrealengine.com/latest/INT/Engine/UMG/UserGuide/WidgetBlueprints/index.html>.

3. Double-click on the **HUD** to open it up inside the UMG Editor.



4. First things first! We are going to display a bar for our health, so from the **Palette** tab on the left-hand side, open **Common** and then drag and drop **Progress Bar** onto the middle screen.
5. With the newly-created **Progress Bar** selected, in the **Details** tab on the right-hand side, rename **Progress Bar** to **HealthBar** and under **Slot**, click on the **Anchors** dropdown and then click on the top-left side to have the bar move according to our liking. I selected it and changed **Size X** to **600**. Under **Progress**, you can move the **Percent** property to see how the bar changes or how the value moves (note that it is a value from 0 to 1).
6. Under the **Details** tab from **Appearance**, change **Fill Color** to any color you like (I chose red) and hit **Okay**.



Now that we have the visuals down, let's get to the implementation of our elements:

1. From the **Details** tab under the **Progress** component's **Percent** property, click on the **Bind** button and then select **Create Binding**.



This will take us to the object's **Graph** mode with the appropriate node being created.

2. Let's first rename this function to `GetHealthPercentage` by double-clicking on its name from the **Function** dropdown.

### Tip

Alternatively, you can right-click on its name and choose **Rename** from the menu, or simply click on it and then press *F2*.

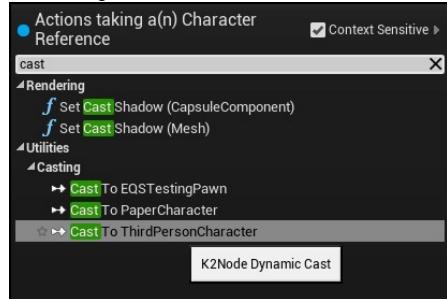
3. After this, let's set the correct value of our character. From the graph, right-click and select **Get Player Character**.

This will give us `Character Reference`, but this is the default version of the character, not the one we created in our blueprint, so we are going to need to cast the object to our class (`ThirdPersonCharacter`).

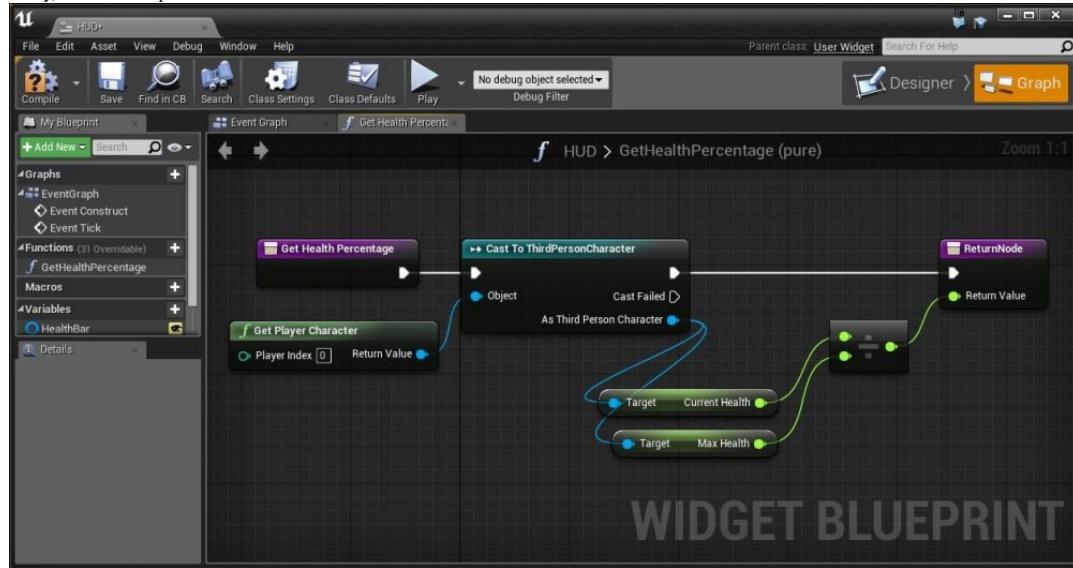
### Note

Computers, by default, want to protect you from making mistakes, so when you get a certain class, such as `Character Reference`, it will only let you do things with it that `Character Reference` can do. The object we're creating when we start the game is a child class of `Character Reference`, which means it contains everything in `Character Reference` blueprint plus more. **Casting** is a programmer's way of saying that you know what the object is, and that you'll take responsibility if the object doesn't. (This is usually done by the game crashing, so don't cast unless you know what you're doing!)

4. Click and drag from the **Return Value** variable and from the action, select **Cast to ThirdPersonCharacter**.



5. Now that we have the cast, delete the connection from `Get Health Percentage` and `Return Node` and then connect the output from `Get Health Percentage` to the input of `Cast to Third Person Character`. Then, connect the output of the `Cast to Third Person Character` action to the input of `ReturnNode`.
6. We still need to set the correct value, so from the `As Third Person Character` output, this time create two `Get` actions—one for `Current Health` and the other for `Max Health`.
7. After this, create a `float / float` action (divide a float by another float) and connect `CurrentHealth` to the top and `Max Health` to the bottom.
8. Finally, connect the output of the division to `Return Value` of `Return Node`.



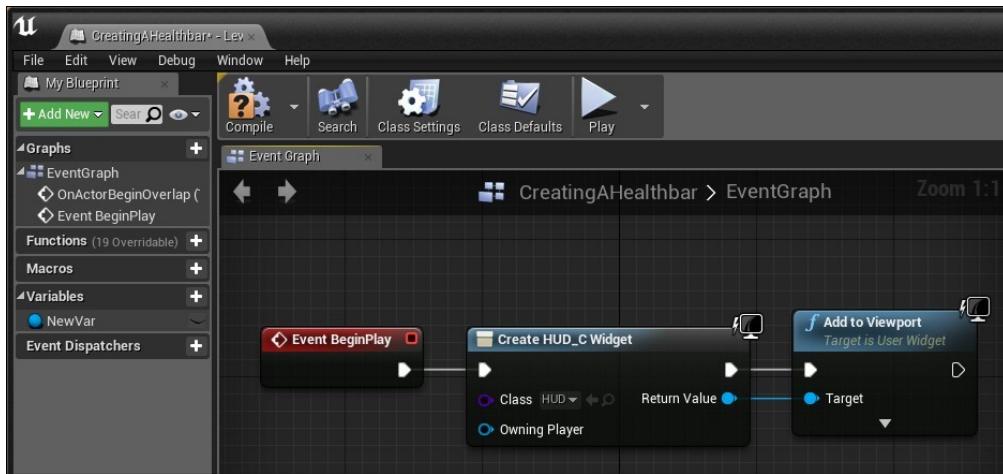
9. Compile, save, and then exit the editor.

10. Next, we need to tell the widget to display on the screen. Go to **Blueprints | Open Level Blueprint** and this time, create an **EventBeginPlay** event.

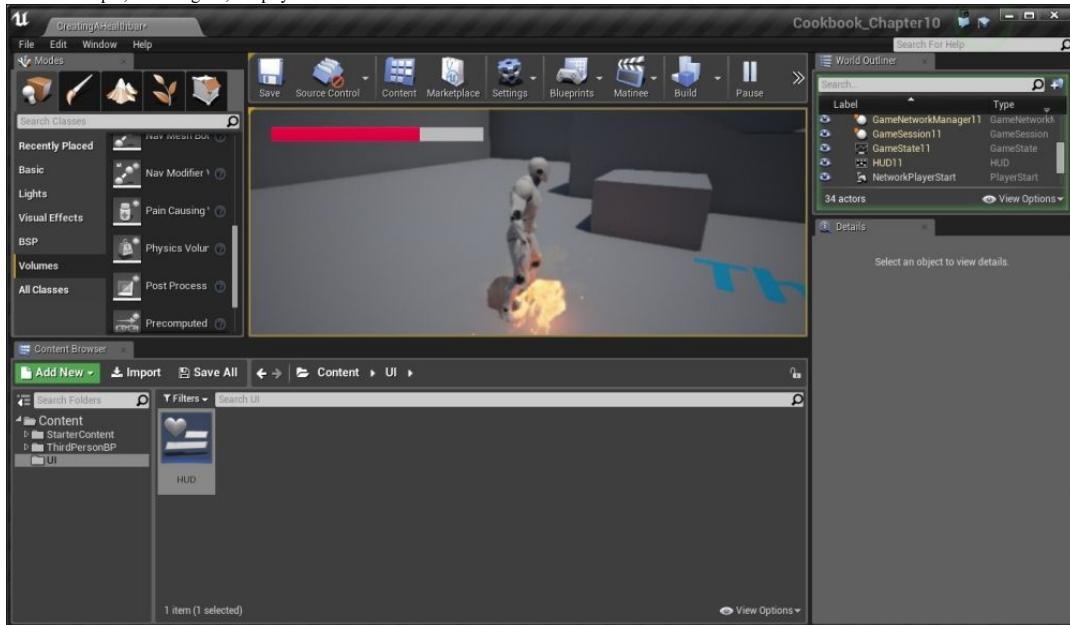
### Tip

Note that it is also possible to put the following blueprint actions into the character's blueprint. For more information on converting from a level to a class blueprint, refer to [Chapter 8, Blueprint Scripting – Level Effects](#).

11. Then, to the right of this, right-click and create a `Create Widget` action. Under **Class** from the dropdown, select `HUD` and connect the arrow from `Event Begin Play` to the input of `Create HUD_C Widget` action.
12. After this, click and drag from the output arrow and create an `Add to Viewport` event. Then, connect the `Return Value` variable of our `Create Widget` action to the `Target` variable of the `Add to Viewport` action.



13. Now let's compile, save our game, and play the level!



With this, we now have a simple healthbar system working for us! With this example in place, you can make much more complex actions dealing with any kind of variable being displayed!

## Dynamic enemy healthbars

Another thing that you'll often see in games are healthbars for enemies that will appear over their heads, changing as the game is being played. In this recipe, we will learn how we can do what we've learned from the previous recipe and apply it here.

### Getting ready

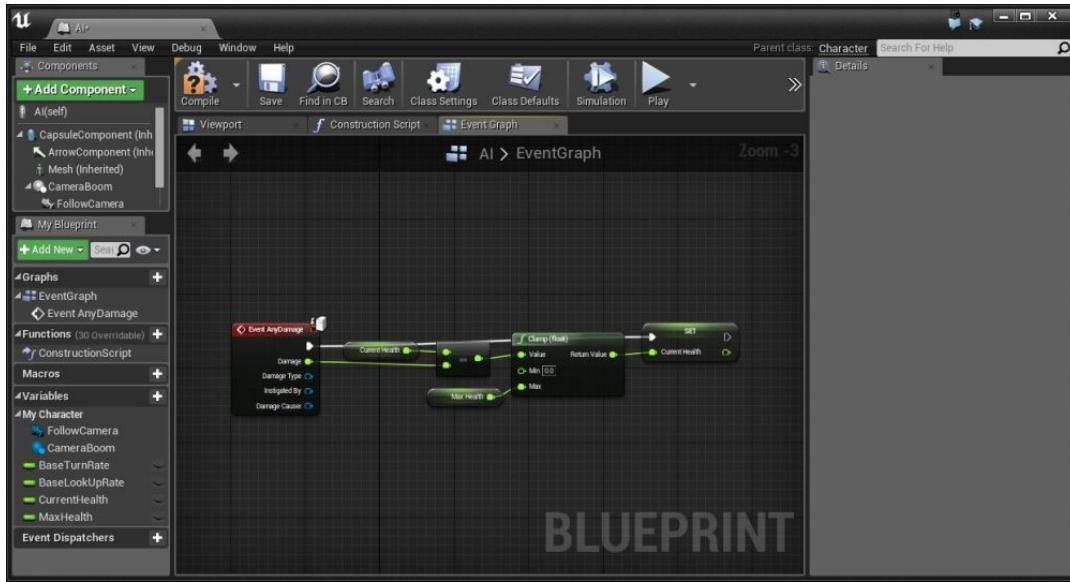
Before we start working on this, we need to have a project created and set up. Do the previous recipe all the way to completion.

### How to do it...

Of course, to actually have enemy healthbars, we will need to actually have an enemy, so let's get that implemented first:

1. Go to the **Content Browser** tab and then to the **ThirdPersonBP/Blueprints** folder. Right-click on the **ThirdPersonCharacter** object and then select **Duplicate** and name the duplicated object **AI**.
2. Double-click on the blueprint to open up its **Event Graph**. We actually don't need any of the previously-created stuff in **Event Graph**, so select all of the Unreal-created content and then delete it. Do note that we do have the **CurrentHealth** and **MaxHealth** properties and **Event AnyDamage** that we created earlier.

We won't be using the **Event AnyDamage** action in this recipe, but it will be good for you to use it in your own internal testing.



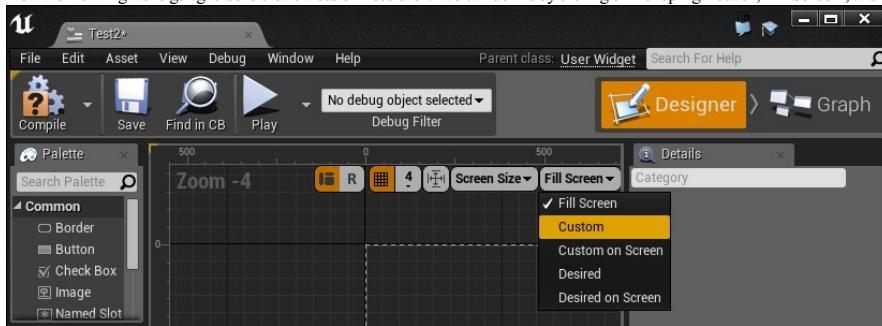
*Blueprint of the new AI object*

3. Close out of the blueprint for right now. We're going to create our enemy healthbar.
4. From the **Content Browser** tab, open the **UI** folder and navigate to **Add New | User Interface | Widget Blueprint**. Give it a name (**EnemyHealthbar**). Double-click on it to open up its editor.

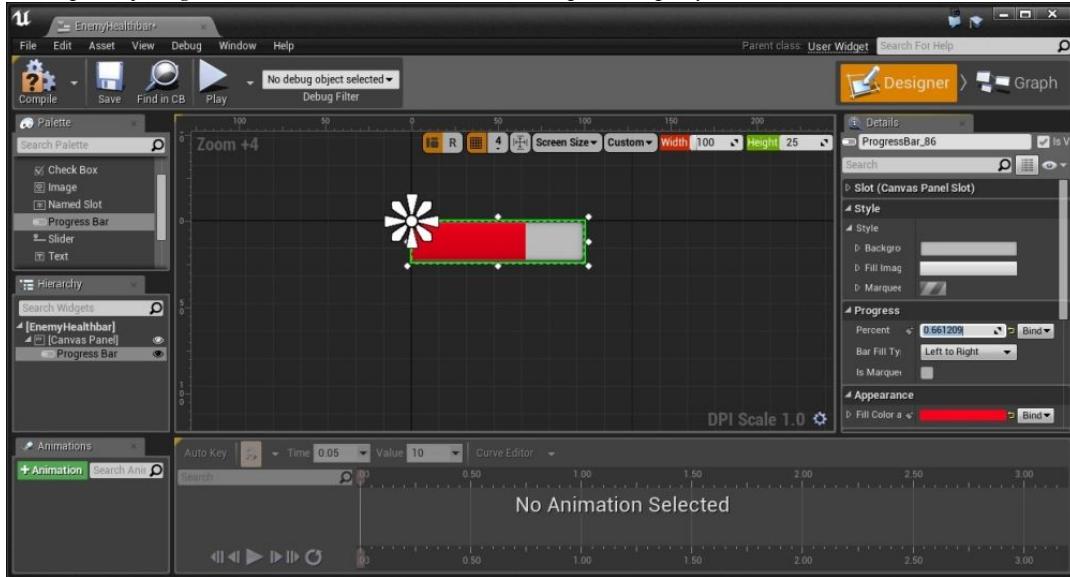
#### Note

It's important to note the lack of spaces between words when naming anything in Unreal Engine 4. Either use `_` or just capitalize the next word's first letter.

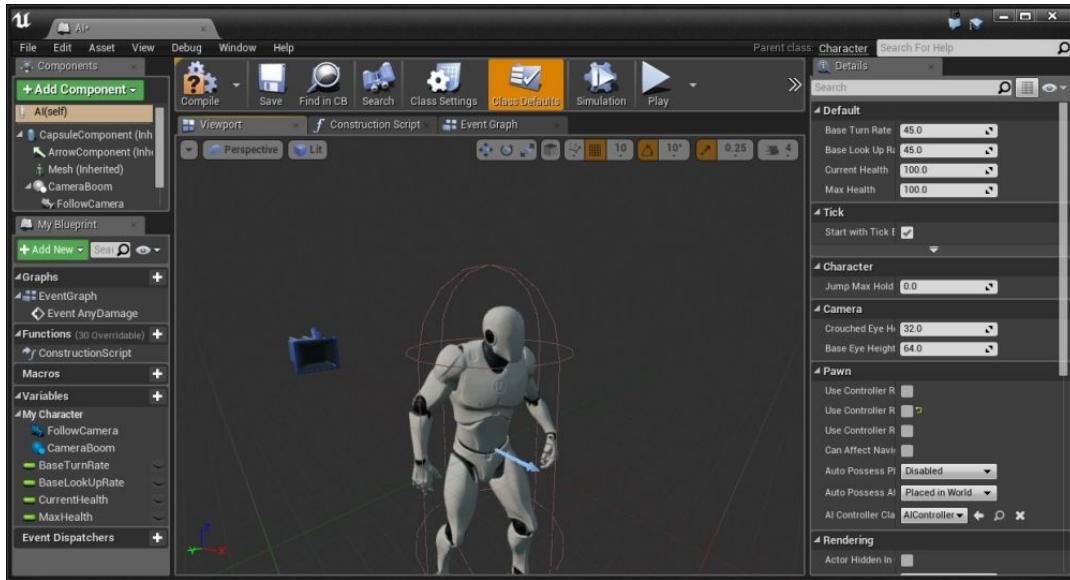
5. Now the first thing we're going to do is create a custom resolution. We can do this by clicking on the top-right button, **Fill Screen**, and then selecting **Custom**.



6. From here, let's keep the **Width** value at 100 and change the **Height** value to 25.
7. Next, drag and drop a **Progress Bar** into the level and scale it to fit into the box. Change the color again if you want to.

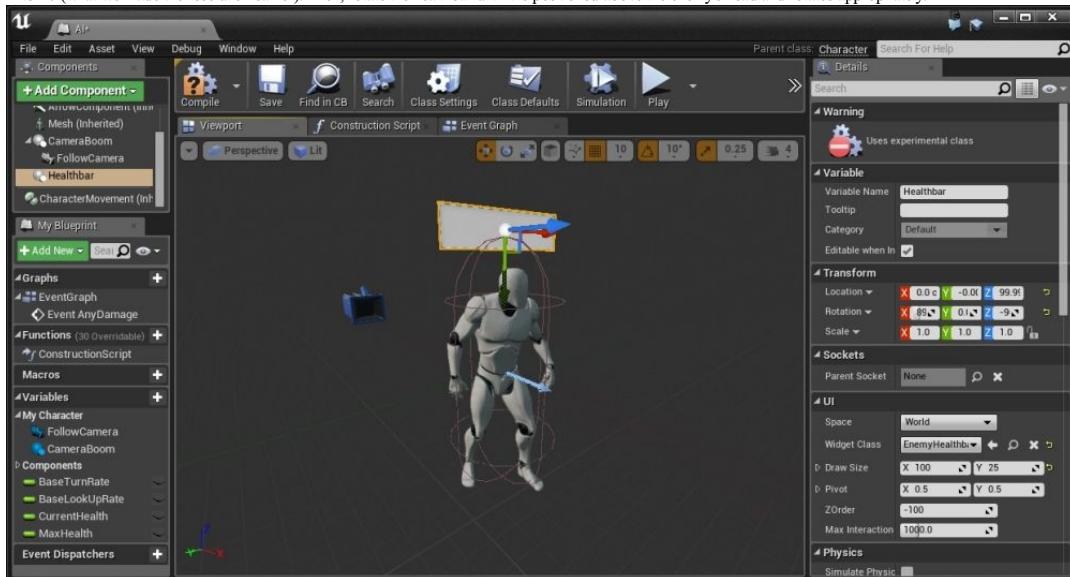


8. Next, rename the progress bar to something a little easier to read, such as `HealthProgressBar`.
9. Compile and save the widget and exit out of the editor. Next, open our **AI** blueprint again and this time, go into the **Viewport** tab.



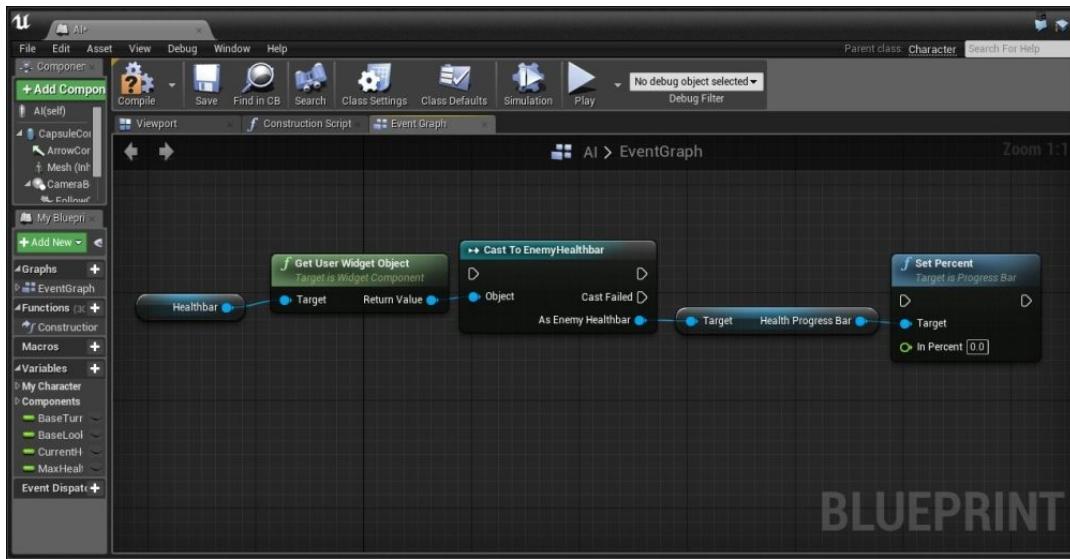
*Viewport tab of the AI blueprint*

10. Go into **Add Component** and select **Widget**. Once created, rename it to **Healthbar**. Under the **UI** component, set the **Widget Class** property to **EnemyHealthbar**. Then, under **Draw Size**, set **X** to 100 and **Y** to 25 (what we made the resolution earlier). Then, rotate the healthbar until it is positioned above the enemy's head and rotates appropriately.

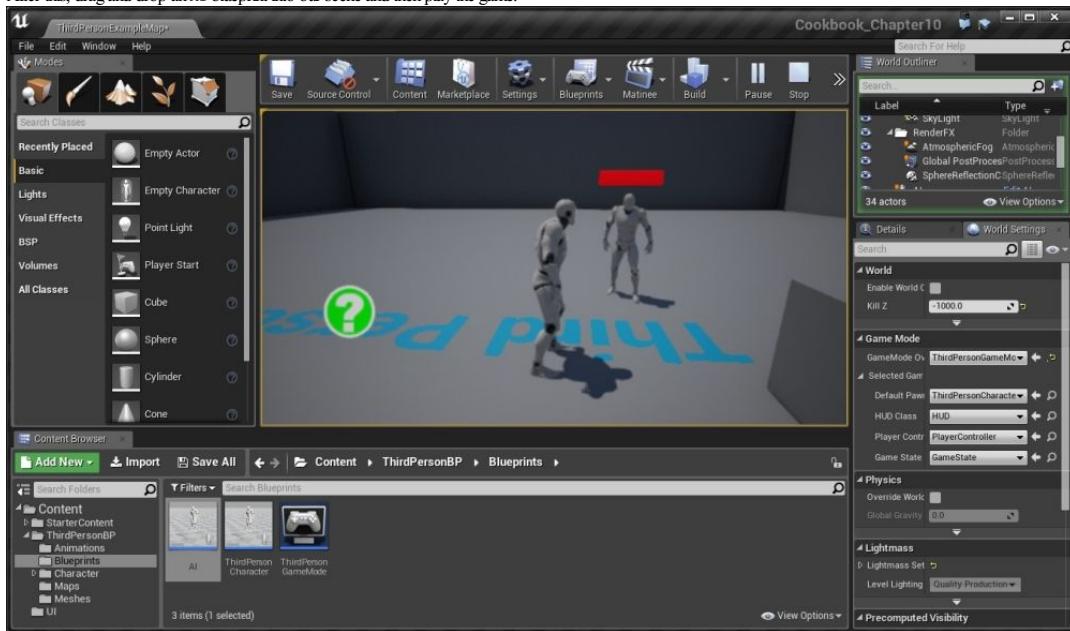


Now, unlike our previous version of the healthbar where we only had to worry about having one player, we can have multiple enemies on the screen. So, instead of having the widget assign its appropriate value, we will use the enemy's blueprint:

1. Select the **Event Graph** tab and then drag and drop the **Healthbar** object from the **Components** tab into **Event Graph**.
2. From the output of the **Healthbar** object, drag out and then create a **Get User Widget Object** action.
3. From the **Return Value** output of the **Get User Widget Object** action, drag out and then create a **CastToEnemyHealthbar** action.
4. Next, from the **As Enemy Healthbar** output, create a **Get Health Progress Bar** action.
5. Finally, drag out **Health Progress Bar** to select the **Set Percent** action.

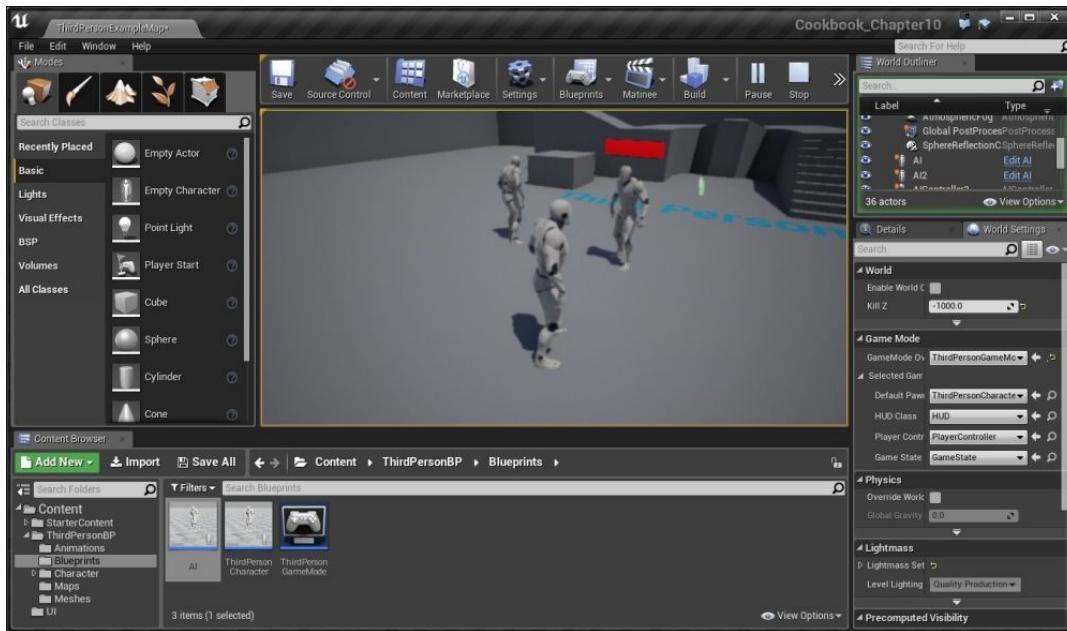


6. Next, we need to divide **Current Health** by **Max Health** once again. Drag and drop the **Current Health** and **Max Health** objects from the **Variables** section and select **Get** both times.
7. To the right of these, create a **float / float** action and connect **Current Health** to the top and **Max Health** on the bottom
8. Finally, create an **Event Tick** event and connect its output to the input of the **Cast to EnemyHealthbar** action and then connect the output of the **Cast to EnemyHealthbar** action to the input of the **Set Percent** action.
9. Compile and save the **Blueprint** and then exit the editor.
10. After this, drag and drop an **AI** blueprint into our scene and then play the game.



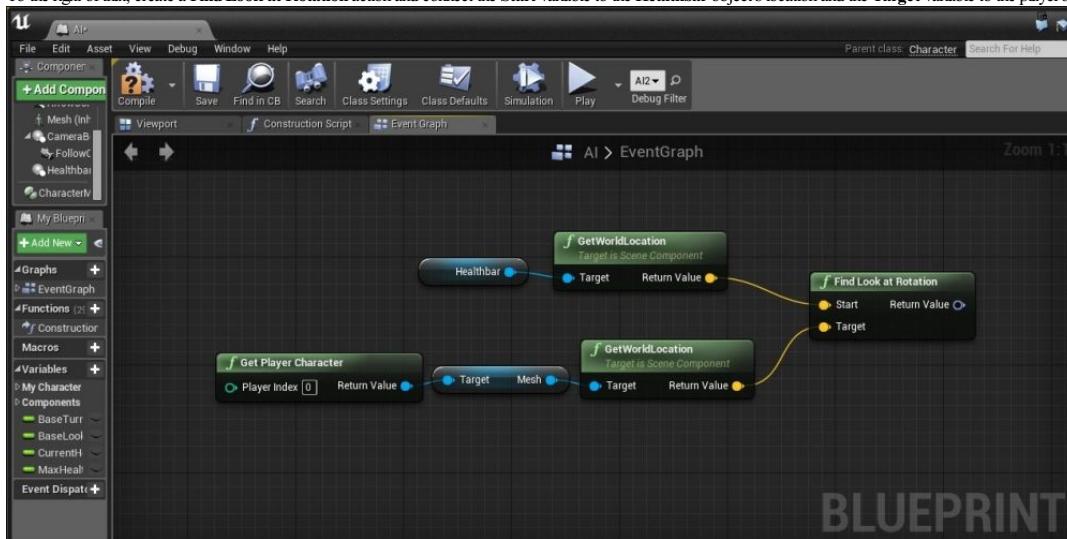
When we are facing the enemy, we can now see the widget clearly, and we can see that it's updated using its health values!

However, we will want to fix an issue, that is, if we aren't facing the enemy, we can't see the UI for it, as you can see in the following screenshot with two enemies, each facing in the opposite direction:



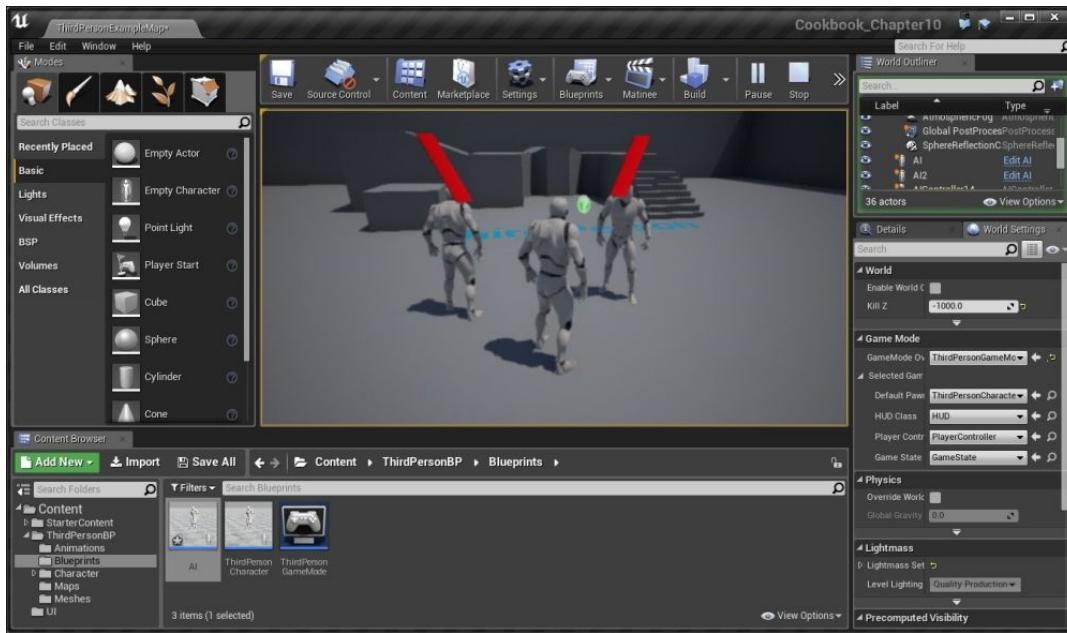
To fix this, we will have the healthbar rotate toward the player.

11. Open up the **AI** blueprint once again. We need to get a couple of variables, namely the player's position and the healthbars. Drag and drop a **Healthbar** variable from the **Components** tab and to the right of it, create a **Get World Location** action.
12. Below this, create a **Get Player Character** action and to the right of that, create another **GetWorldLocation (Mesh)** to get the information for the **Mesh** component.
13. To the right of that, create a **Find Look at Rotation** action and connect the **Start** variable to the **Healthbar** object's location and the **Target** variable to the player's.



*Finding the Look at Rotation of the healthbar to the player*

The rotational value that this will contain will make the object rotate toward us, but it will look strange if we just set it because the bar will move up and down as we move around it.



This is due to the pivot point of the object that it's being rotated to being in the top-left corner rather than in the center. To fix this, we can create a new object as a parent that will work correctly.

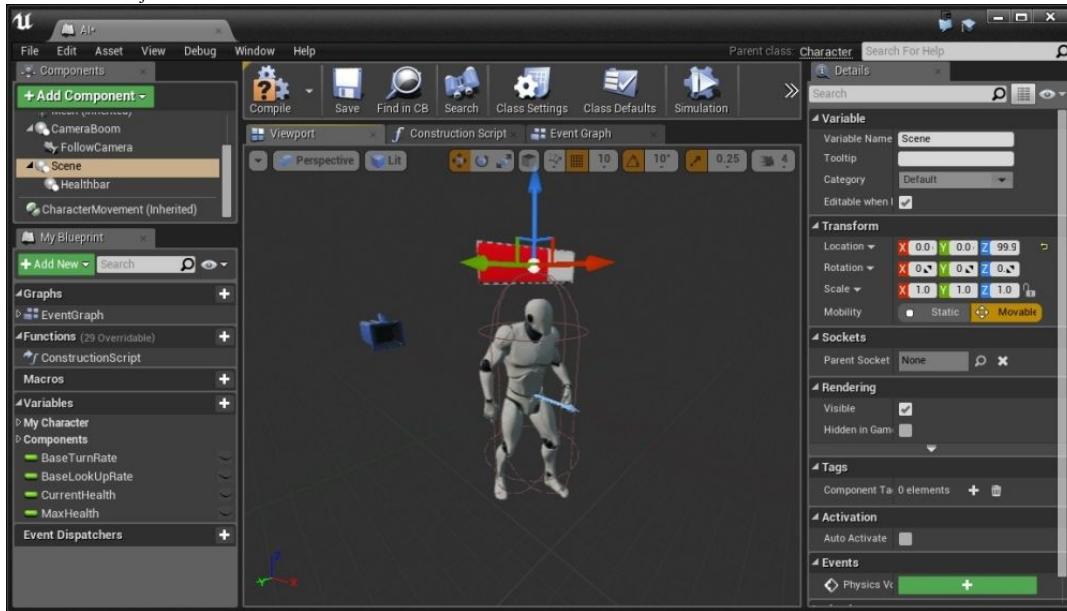
- Open up the **AI Blueprint** and with the **Viewport** tab being selected, go to the **Components** tab and navigate to **Add Component | Scene**.

A scene component is a kind of an empty slate, but it has a **Transform** component so that we can use it as holder or custom pivot, as in this case.

#### Note

For more information on the Scene component, (also referred to as a USceneComponent) refer to <https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Actors/Components/index.html#scenecomponents>.

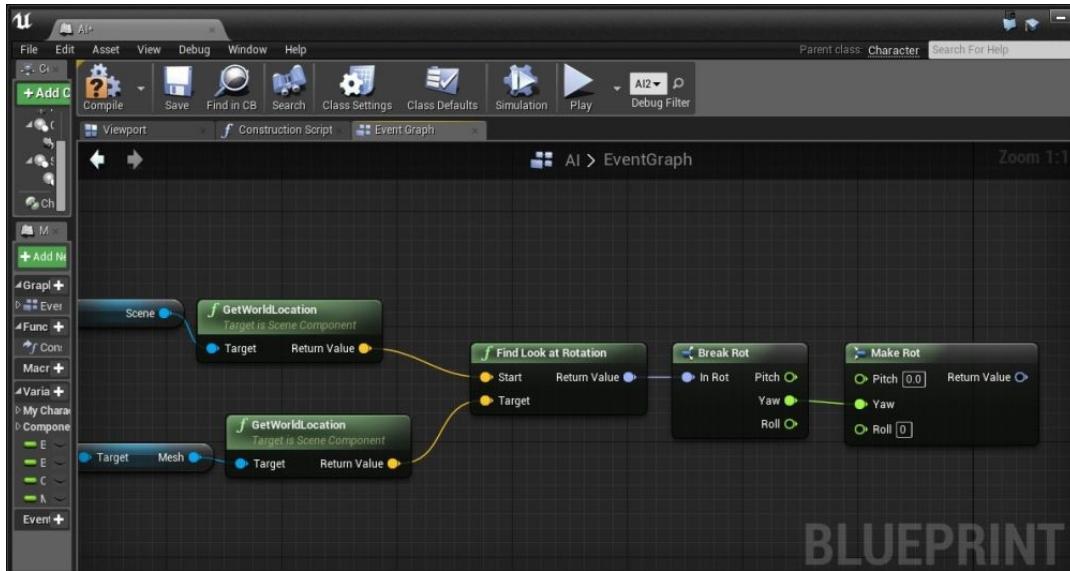
- Move the **Scene** object until its centered around the **Healthbar** object and then from the **Components** tab, drag and drop the **Healthbar** object on top of the **Scene** object to make the **Scene** object the parent of the **Healthbar** object.



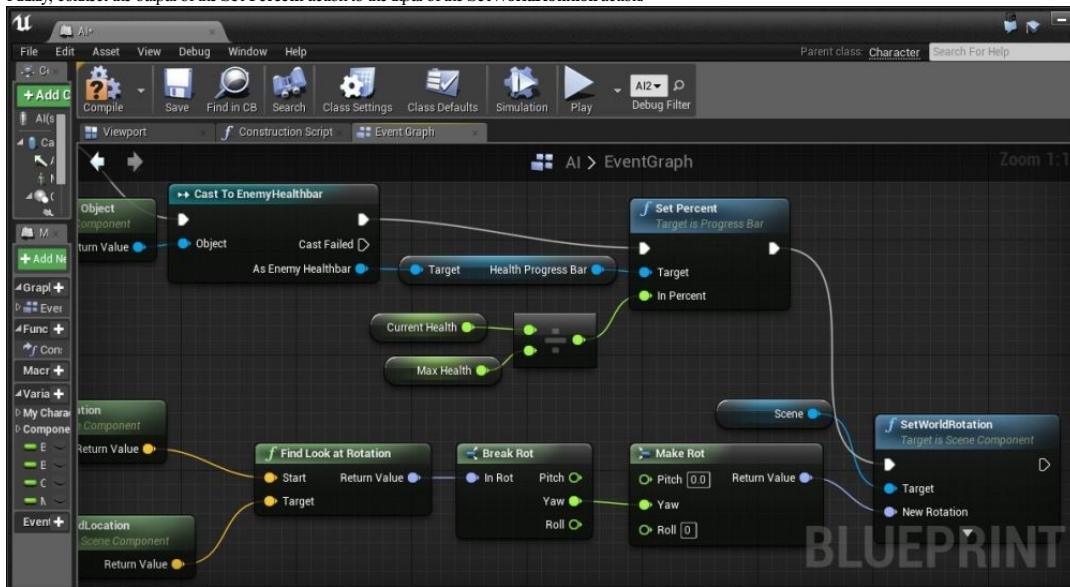
- Next, we will need to replace the **Healthbar** variable with the **Scene** variable for the **GetWorldLocation** action (select the **Healthbar** variable, press **Delete** and then move **Scene** in and connect it).

Now, this is much better, but there's still some issues when jumping or getting too near. This is because in reality, we only want **Yaw** to move.

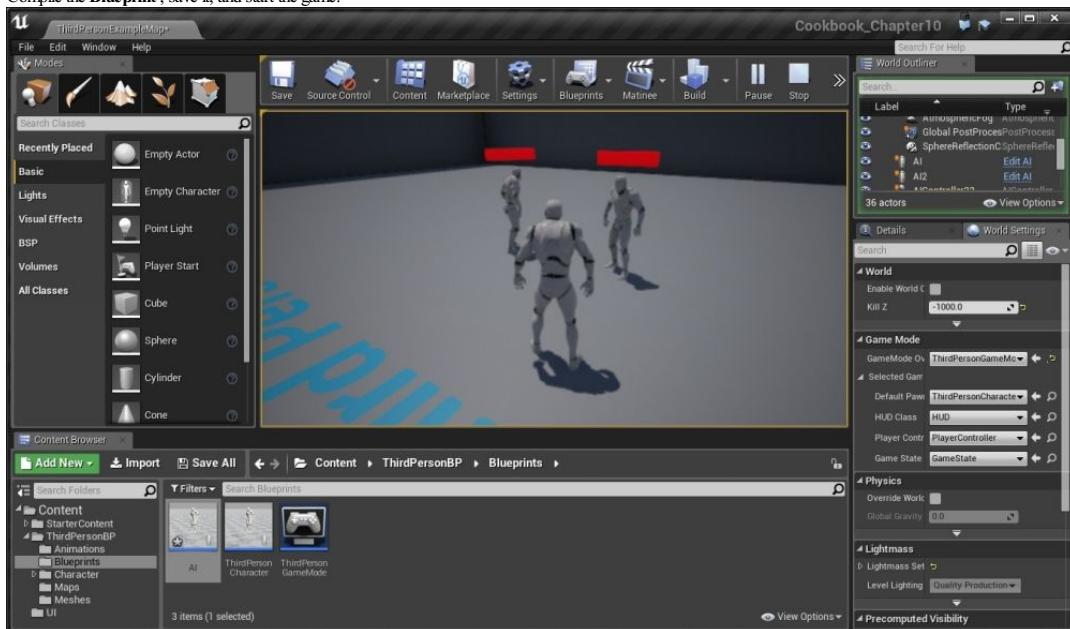
- From **Return Value** of the **Find Look at Rotation** action, drag out and select **Break Rot**, which will give us each part of the rotation by itself.
- To the right of this newly-created rotation, right-click and select **Make Rot** and connect **Yaw** of the **Break Rot** action to **Yaw** of the **Make Rot** action.



19. Now, we need to right-click and select a **Set World Rotation (Scene)** action and connect **Return Value** to **New Rotation**.  
 20. Finally, connect the output of the **Set Percent** action to the input of the **SetWorldRotation** action.



21. Compile the **Blueprint**, save it, and start the game!



Now our enemy's healthbars will rotate toward our player at all times!

# Creating a main menu

A main menu can serve as an introduction to your game and is a great place for us to discuss some additional things that UMG has, such as texts and buttons. We'll also learn how we can make buttons do things. Let's spend some time to see just how easy it is to create one!

## Note

For more information on the client-server model, refer to [https://en.wikipedia.org/wiki/Client%20server\\_model](https://en.wikipedia.org/wiki/Client%20server_model).

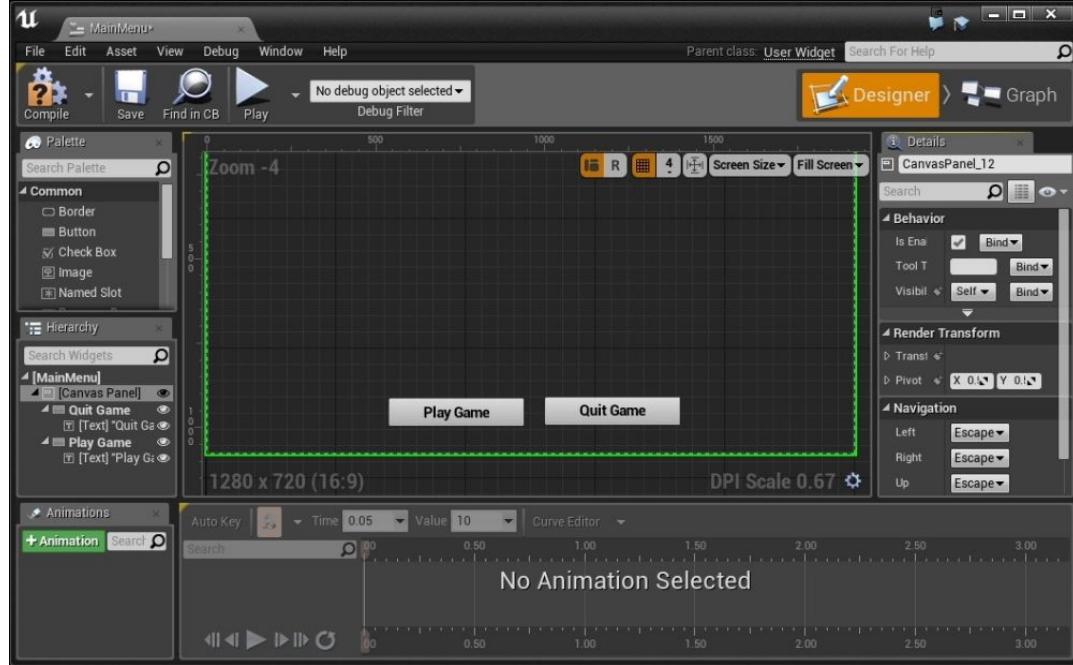
## How to do it...

To give you an idea of how it works, let's create an empty level to hold our menu.

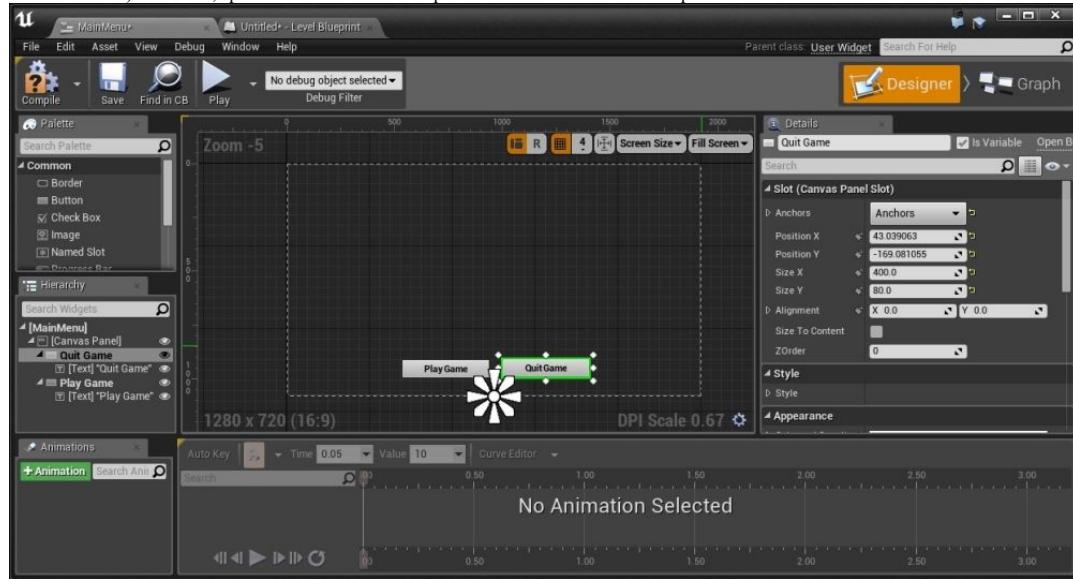
1. Create a new level by going to **File | New Level** and select **Empty Level**.
2. Next, inside the **Content Browser** tab, go to our **UI** folder and navigate to **Add New | User Interface | Widget Blueprint** and give it a name (**MainMenu**). Double-click on it to open up the editor.

In this menu, we are going to have the title of the game, and then a series of buttons the player can press.

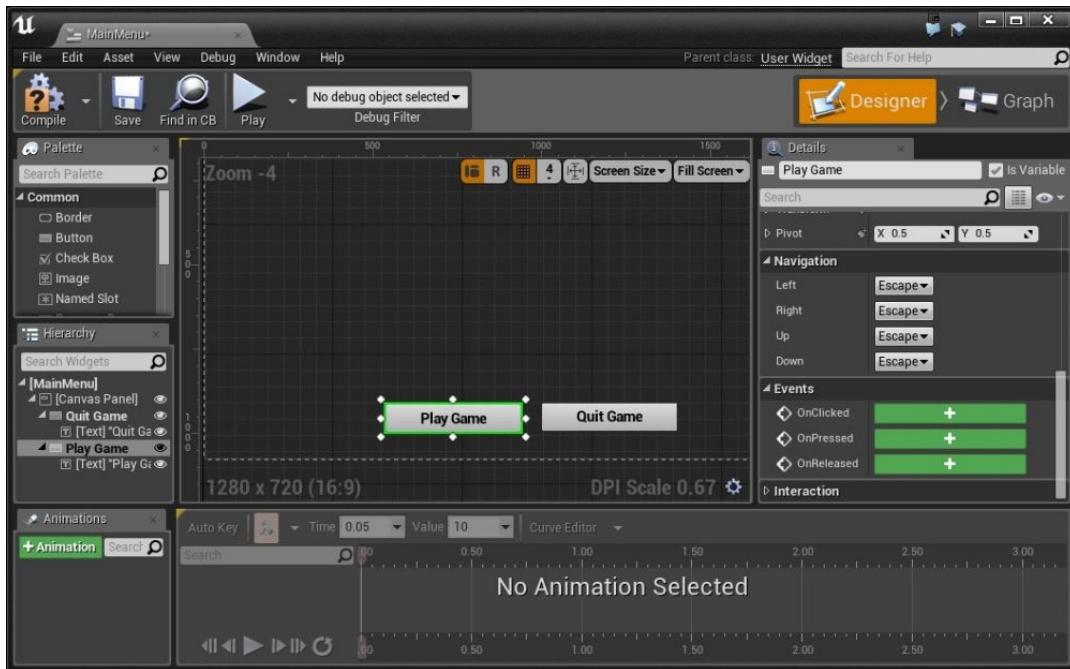
3. From the **Palette** tab, open up the **Common** section and drag and drop a **Button** onto the middle of the screen.
4. Select the button and change its **Size X** to **400** and **Size Y** to **80**. We will also rename the button to **Play Game**.
5. Drag and drop a **Text** object onto the **Play Game** button, and you should see it snap onto the button as a child. Under **Content**, change **Text** to **Play Game**. Under **Appearance**, change the color of the button to black and change the **Font** size to **32**.
6. From the **Hierarchy** tab, select the **Play Game** button and copy and paste it to create a duplicate. Move the button down, rename it to **Quit Game**, and change the text Content as well.
7. Move both of the objects so that they're on the bottom part of the HUD, slightly above and side by side, as shown in the following image:



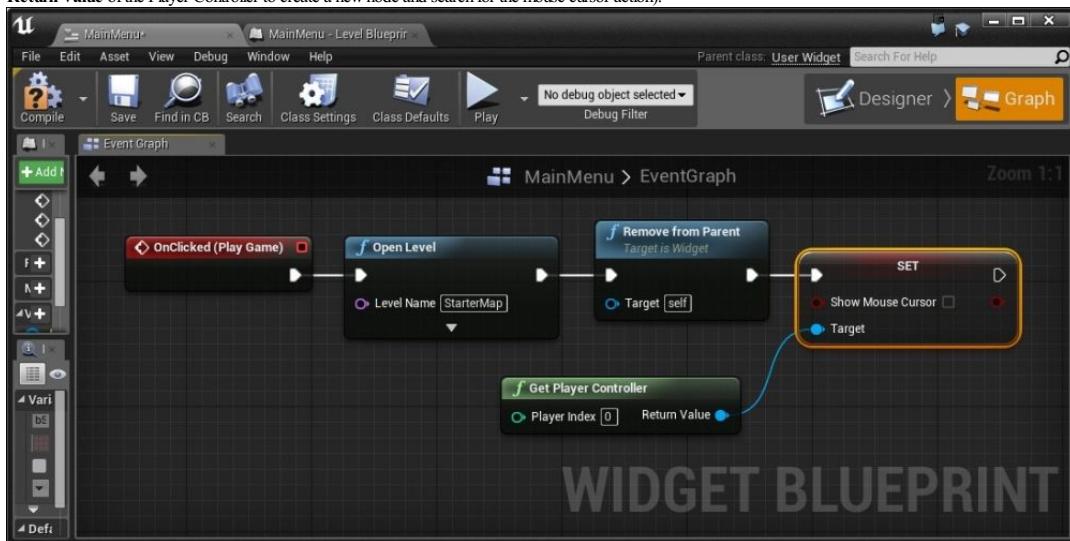
8. Lastly, we'll want to set our pivots and anchors accordingly. When you select either the **Quit Game** or **Play Game** buttons, you may notice a sun-like widget that displays the **Anchors** of the object (known as the Anchor Medallion). In our case, open **Anchors** from the **Details** panel and click on the bottom center option.



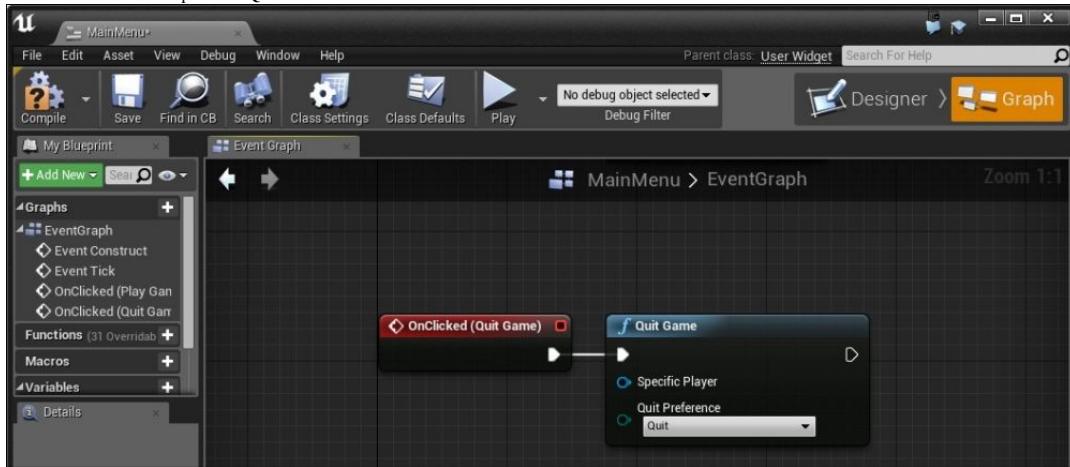
9. Now that we have the buttons created, we want them to actually do something when we click on them. Select the **Play Game** button and from the **Details** tab, scroll down until you see the **Events** component. There should be a series of big green + buttons. Click on the green button beside **OnClicked**.



10. Next, it will take us to **Event Graph** with the appropriate event created for us. To the right of the event, right-click and create an **Open Level** action. Under **Level Name**, put in whatever level you like (for example, `StarterMap`) and then connect the output of the **OnClicked** action to the input of the **Open Level** action.
11. To the right of that, create a **Remove from Parent** action to make sure that when we leave it, the menu doesn't stay.
12. Finally, create a **Get Player Controller** action and to the right of it, a **Set Show Mouse Cursor** action disabled so that the mouse will no longer be visible as we want to see the mouse in the menu (drag from **Return Value** of the Player Controller to create a new node and search for the mouse cursor action).

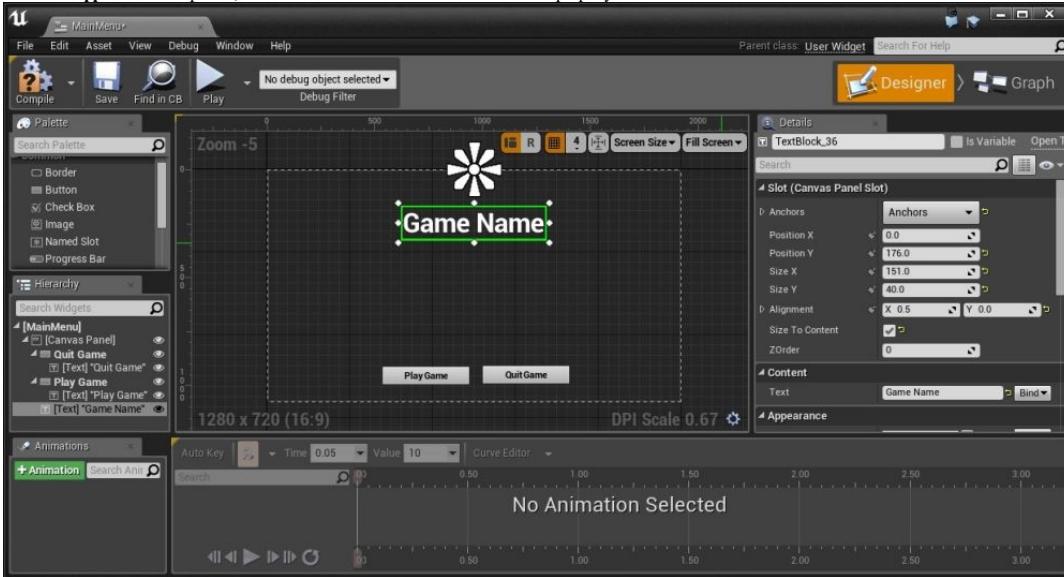


13. Now, go back to the **Designer** button and then select the **Quit Game** button. Click on the **OnClicked** button as well and to the right of this one, create a **Quit Game** action and connect the output of the **OnClicked** action to the input of the **Quit Game** action.



14. Lastly, as a bit of polish, let's add our game title to the screen. Drag and drop another **Text** object onto the scene, this time with **Anchor** at the top of center. From here, change **Position X** to **0** and **Position Y** to **176**.
15. Change the **Alignment** value of the **X** axis to **.5** and check the **Size to Content** option for it to automatically resize.
16. Set the **Content** component's **Text** property to the game's name (in my case, `Game Name`).

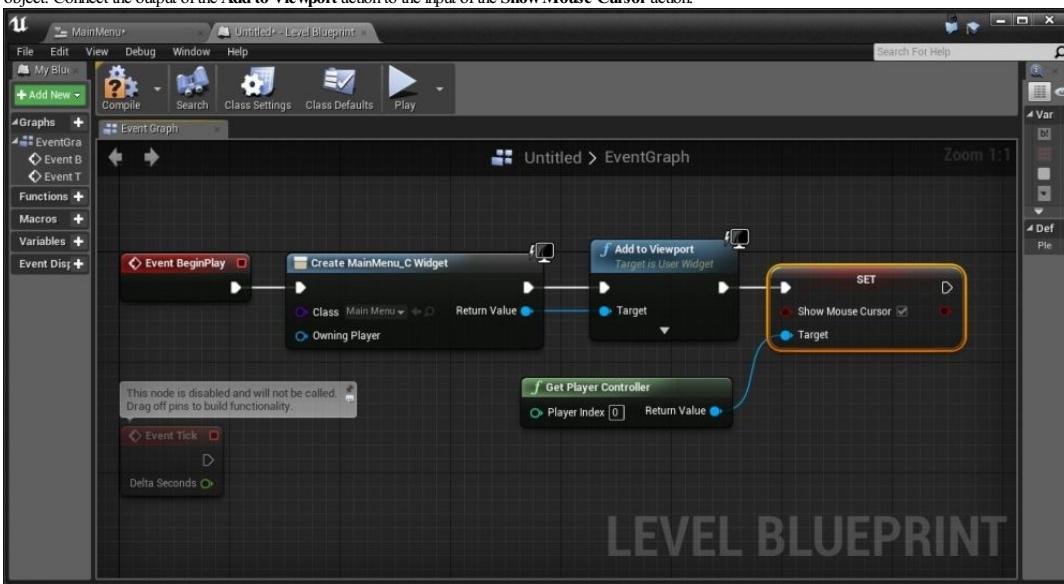
17. Under the **Appearance** component, set the **Font** size to **93** and set the **Justification** property to **Center**.



#### Note

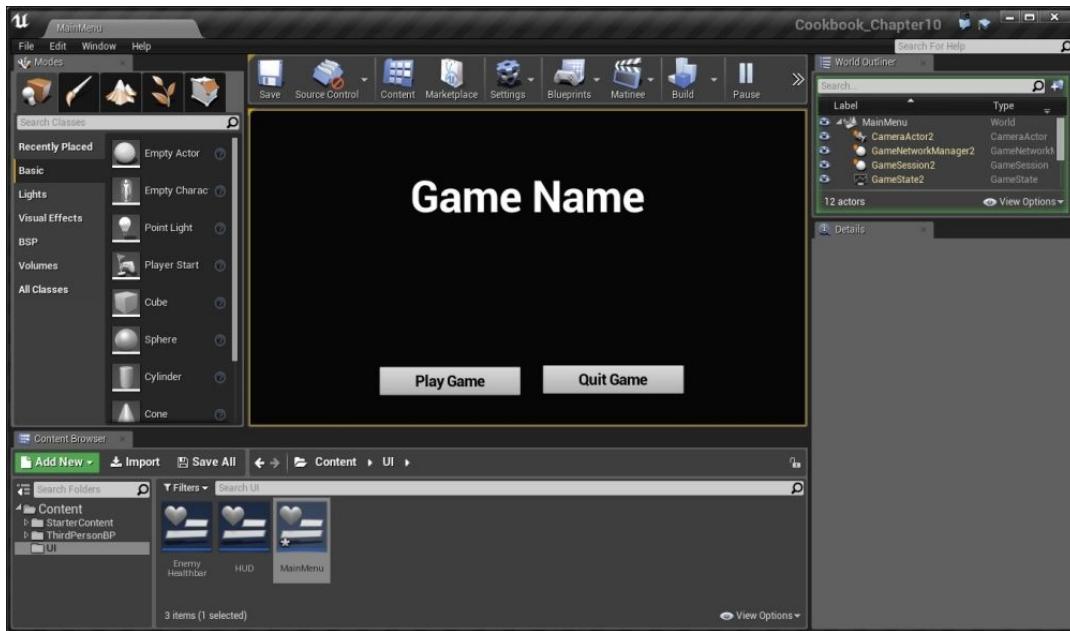
There are a number of other styling options that you may wish to use when developing your HUDs. For more information on that, refer to <https://docs.unrealengine.com/latest/INT/Engine/UMG/UserGuide/Styling/index.html>.

18. Compile the menu, and save it, and now we need to actually have the widget show up. To do so, we need to take the same steps as we did previously.
19. Open up **Level Blueprint** by going to **Blueprints | Open Level Blueprint** and create an **EventBeginPlay** event.
20. Then, to the right of that, right-click and create a **Create Widget** action. Under **Class**, from the dropdown, select **MainMenu** and connect the arrow from **Event Begin Play** to the input of **Create MainMenu\_C Widget**.
21. After this click and drag from the output arrow and create an **Add to Viewport** event. Then, connect **Return Value** of our **Create Widget** action to **Target** of the **Add to Viewport** action.
22. Lastly, we also want to display the player's cursor on the screen to show buttons. To do that, right-click and select **Get Player Controller**. Then, from **Return Value** of that, create a **Set Show Mouse Cursor** object. Connect the output of the **Add to Viewport** action to the input of the **Show Mouse Cursor** action.



*Blueprint to show player's cursor on the screen*

23. Compile, save, and run the project!



With this our menu is completed! We can quit the game with no problems, and pressing the **Play Game** button will start our level!

## Animating a menu

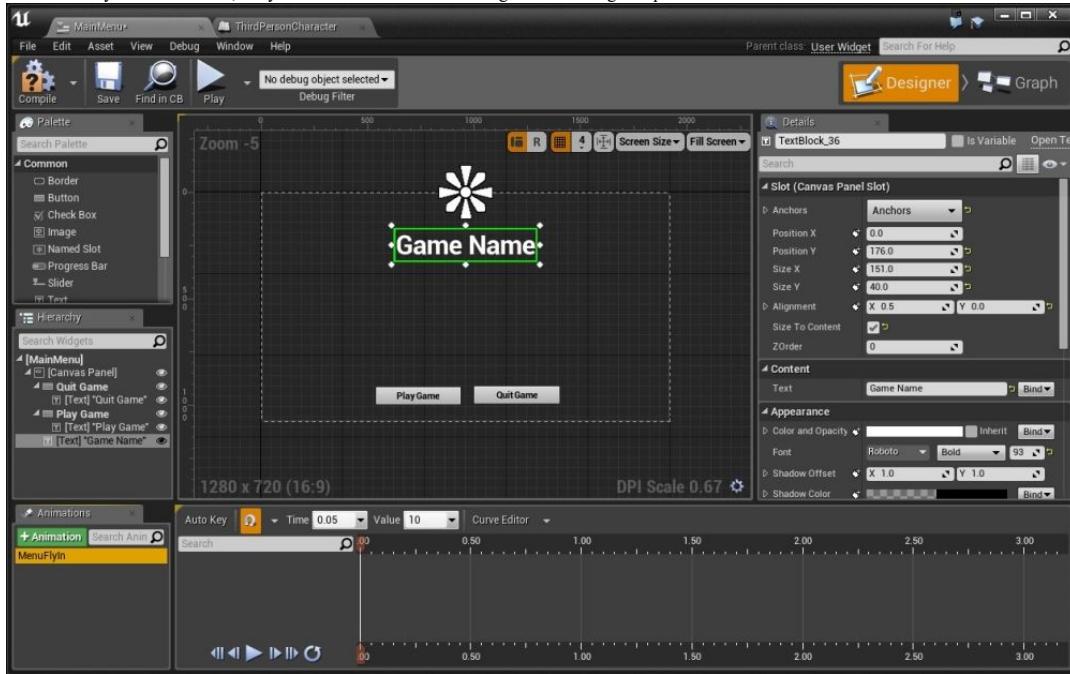
You may have created a menu or UI element at some point, but rather than having it static and non-moving, let's spend some time looking at how we can animate the menus by having them fly in and out or animate in some way. This will help add polish to the title as well as enable players to notice things easier as they move in.

### Getting ready

Before we start working on this, we need to have a project created and set up. Follow the previous recipe all the way to completion.

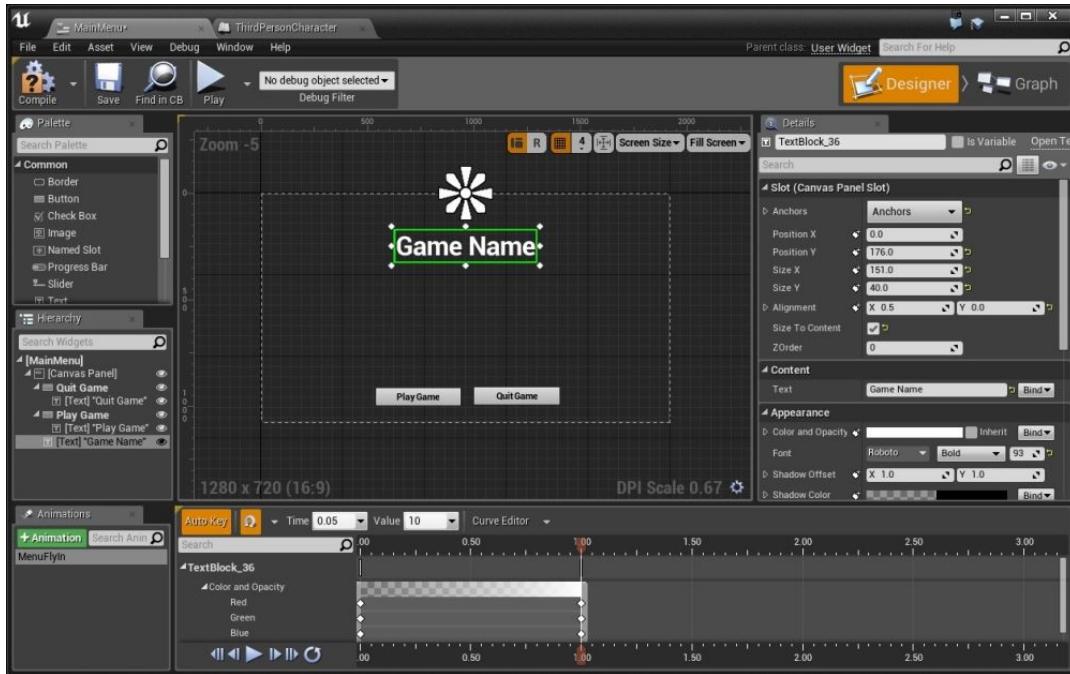
### How to do it...

1. Open up the **MainMenu** blueprint once more and from the bottom-left in the **Animations** tab, click on the **+Animation** button and give the new animation a name (**MenuFlyIn**).
2. Select the newly-created animation, and you should see the window on the right-hand side brighten up.



*Creating a new animation*

3. Next, click on the **Auto Key** toggle to have the animation editor automatically set keys that are appropriate for our implementation.
4. If it's not there already, move the **timeline** bar (the white line with two orange ends on the top and bottom) to the **0.00** mark on the animation timeline.
5. Next, select the **Game Name** object and under the **Color and Opacity** option, open it up and change the **A** (alpha) value to **0**.
6. Now, move the timeline bar to the **1.00** mark and then open the color again and set the **A** value to **1**.



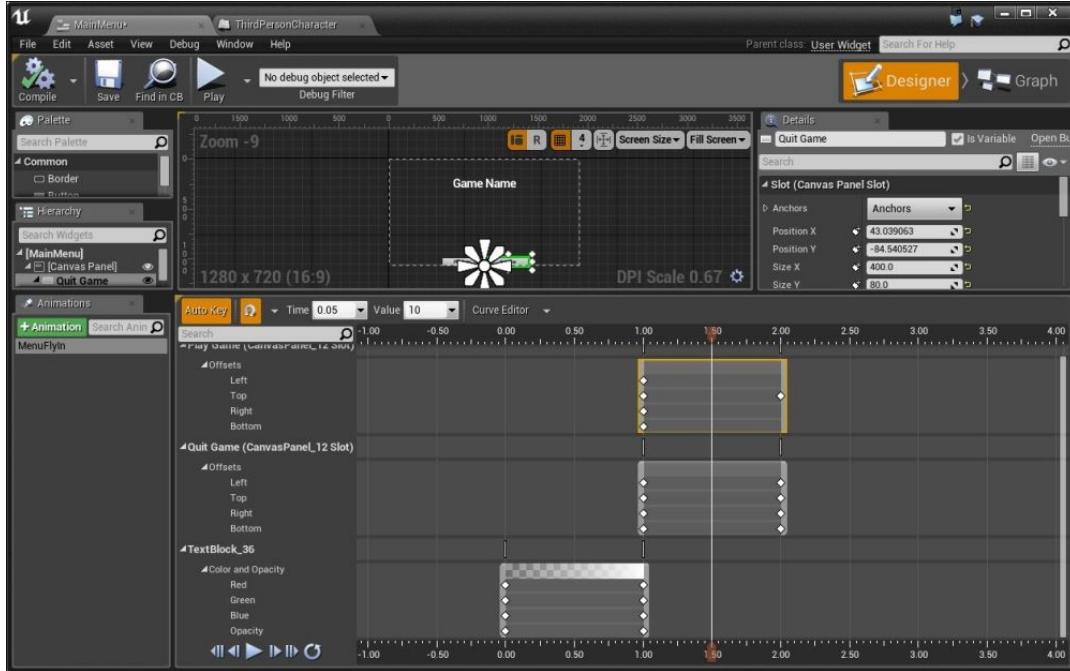
You'll notice that now there is a transition going from a completely transparent text to a fully shown one. This is a good start. Let's have the buttons fly in after the text appears.

7. Next, move the timeline bar to the 2.00 mark and select the **Play Game** button. Now from the **Details** tab, you'll notice that under the variables there are new + icons to the left of variables. This will save the value for use in the animations. Click on the + icon by the **Position Y** value.

#### Tip

If you use your scroll wheel while inside the dark gray portion of the timeline, (where the keyframe numbers are displayed) it zooms in and out. This can be quite useful as you create more complex animations.

8. Now move the timeline bar to the 1.00 mark and move the **Play Game** button off the screen. By doing the animation in this way, we are saving where we want it to be at the end, and then we will go back and do the animations.
9. Do the same animation for the **Quit Game** button.



*Animation for the Quit Game button*

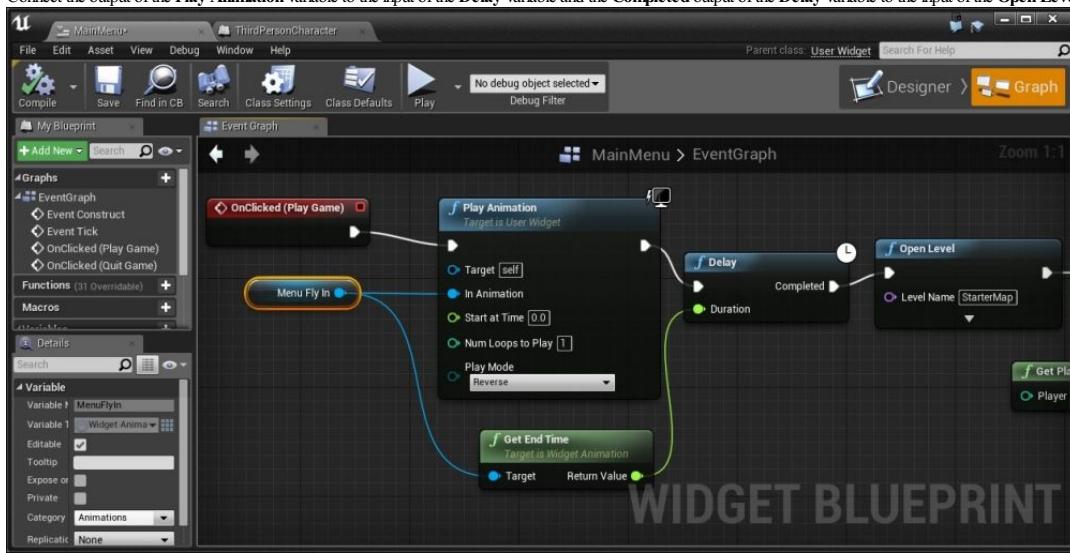
Now that our animation is created, let's make it so that when the object starts, it plays this animation:

1. Click on the **Graph** button and from the **MyBlueprint** tab under the **Graphs** section, double-click on the **Event Construct** event, which is called as soon as we add the menu to the scene. Grab the pin on the end of it and create a **Play Animation** action.
2. Drag and drop a **MenuFlyIn** animation into the scene and select **Get**. Connect its output pin to the **In Animation** property of the **Play Animation** action.

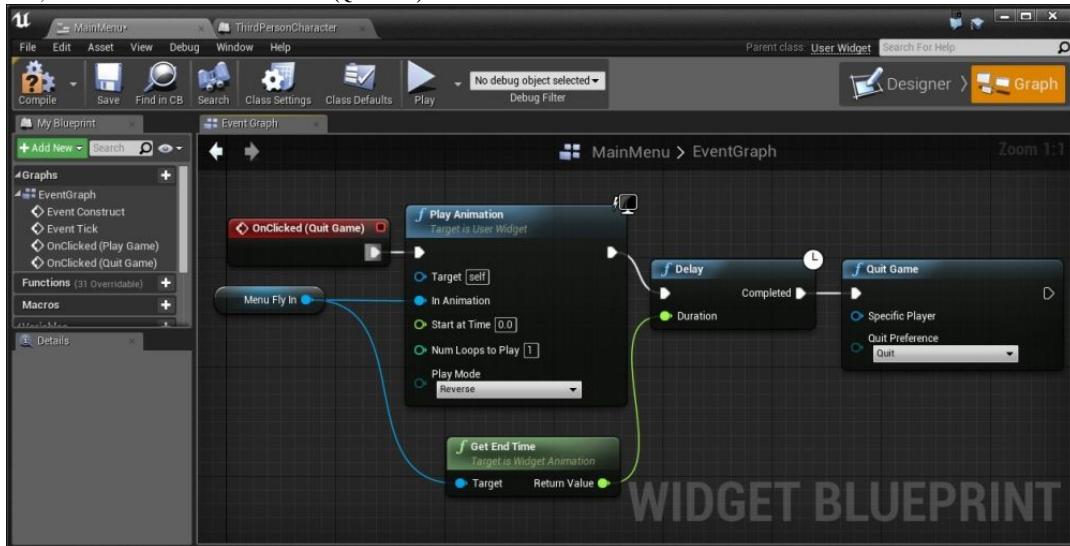


Now that we have the animation working for the menu, let's have it play when we leave the menu.

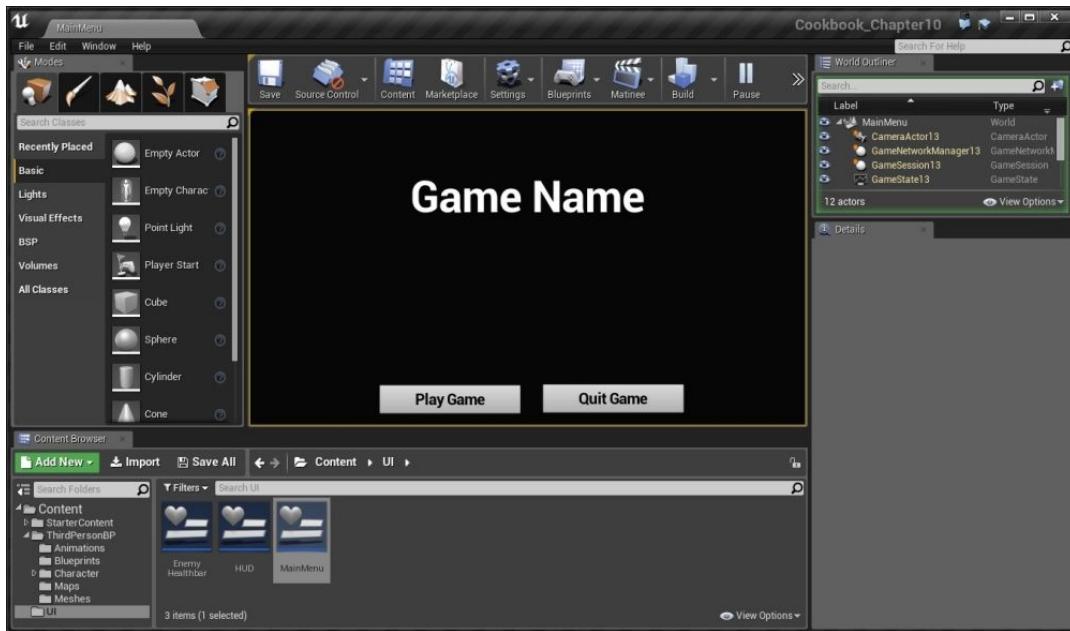
1. Select the **Play Animation** and **Menu Fly In** variables and copy them. Then, move to the **OnClicked (Play Game)** action, drag the **OnClicked** event over to the left, and remove its original connection to the **Open Level** action by holding down *Alt* and clicking.
2. Paste (*Ctrl + V*) the new objects and connect the output of **OnClicked (Play Game)** to the input of **Play Animation**. Now change the **Play Mode** to **Reverse**.
3. To the right of that, create a **Delay** action. For the **Duration** variable, we want to wait as long as the animation is, so from the **Menu Fly In** variable, create another pin and a **Get End Time** action. Connect **Return Value** of the **Get End Time** variable to the input of the **Delay** action.
4. Connect the output of the **Play Animation** variable to the input of the **Delay** variable and the **Completed** output of the **Delay** variable to the input of the **Open Level** action.



5. Now, we need to do the same for the **OnClicked (Quit Game)** event.



6. Now compile, save, and run the game!



Our menu is now completed! You learned about how animation works inside UMG.

#### Note

For more examples of using UMG for animation, refer to <https://docs.unrealengine.com/latest/INT/Engine/UMG/UserGuide/Animation/index.html>.

#### See also

I would love to write an entire book about creating UIs for games, but there just isn't enough space in this book to dive deeply into it. However, for those interested in doing more complex UI, here are some additional tutorials, which may prove to be beneficial:

- Basic Inventory System in Blueprint : <http://www.tomlooman.com/tutorial-basic-inventory-system-in-blueprint/>
- Creating / Scripting an In-Game Pause Menu : <https://docs.unrealengine.com/latest/INT/Engine/UMG/QuickStart/5/index.html>
- Level Menus (UMG) : <https://www.youtube.com/watch?v=wPzIvBP8-PA>
- UMG Using a Gamepad : <https://www.youtube.com/watch?v=tCXuNu4RETs>
- UMG, Create Scrollable List of Clickable Buttons From Dynamic Array : [https://wiki.unrealengine.com/UMG\\_Create\\_Scrollable\\_List\\_of\\_Clickable\\_Buttons\\_From\\_Dynamic\\_Array](https://wiki.unrealengine.com/UMG_Create_Scrollable_List_of_Clickable_Buttons_From_Dynamic_Array)

## Chapter 11. Publishing and Deployment

In this chapter, we'll cover the following recipes:

- Packaging your project
- Creating an installer for Windows

### Introduction

At this point, you should have created an entire project, but taking the time to get the projects out into the world is just as important. Playing the game inside the editor is nice and all, but playing the game as a standalone title has a special feel to it that you can't duplicate.

After exporting it to the computer, it is possible for you to just zip it up and send someone that file, but you spent quality time on your project, and so, give it the respect it deserves. With that in mind, people notice the polish that is put into games and the little things, such as an installer, can help get people into the mood of your project early on and see your game as a professional title.

### Packaging your project

Before we can export our project and share it with others, we first need to package it. Packaging is the name for a series of different steps, including compiling your project, then cooking content into a format that the platform understands before finally putting the project in a certain format. This will allow you to test/play your full game (instead of a single map) in the same way that it would be when published.

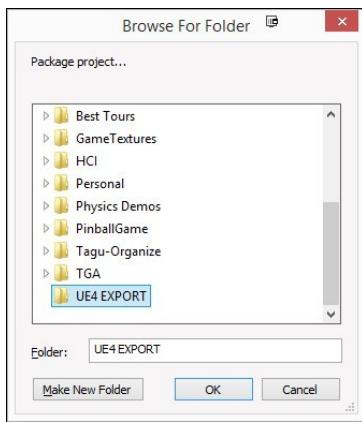
#### Getting ready

Before we start working on this, we need to have a completed project. In my example, I created a new blank project, but any of the things we worked on in the book should work fine.

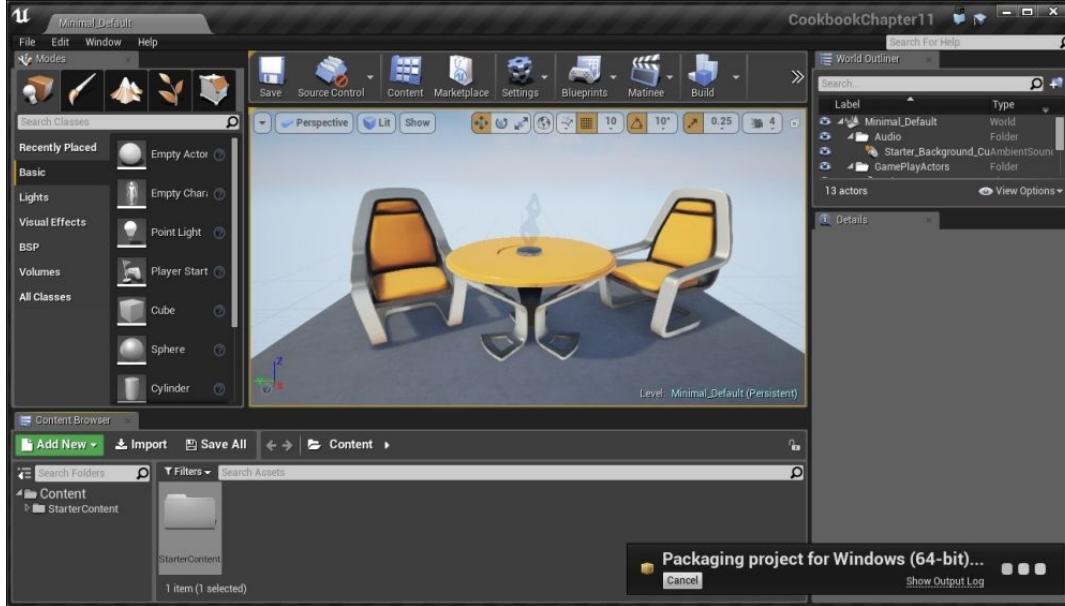
#### How to do it...

Let's see just how the packaging works! Follow these steps:

1. From the **File** menu, navigate to **Package Project | Windows | Windows (64 Bit)**.
2. From there, you should see a **Browse For Folder** dialog window pop up. Here, we will want to place the files for the game. In my case, I selected **Desktop** and then selected **Make New Folder**. From here, I renamed this new folder to **UE4\_EXPORT**:



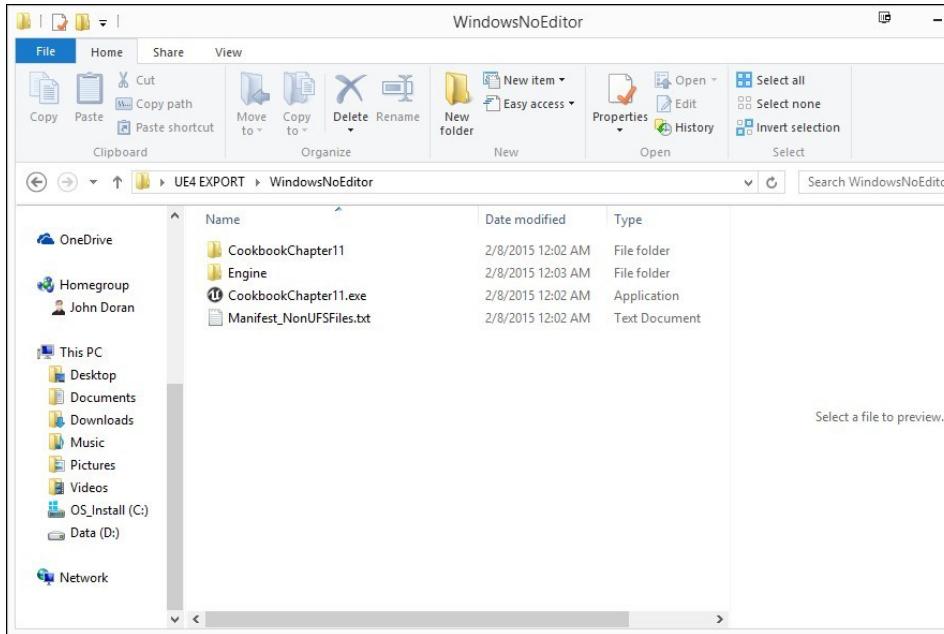
3. After you've selected your folder, click on the **OK** button and you'll notice a new message on the bottom-right of your screen, saying that the project is currently being packaged.



This is done as a background process as it will often take some time, so you can still make changes and work with the editor; note that the changes made during this period will not be reflected in your packaged project.

It is also possible to display additional information about what's going on by clicking the **Show Log** button. The output log can be useful in case something went wrong:

- Once you see the message disappear, you should go back to the exported folder, and once there, you should see a folder called `WindowsNoEditor`. Double-click on that folder and you should see a number of files.



- Double-click on the `.exe` file and you should see your game started!



With this, we've now exported our project for others to play!

### Tip

If, like this generic example, you do not have functionality for exiting the game, feel free to press *Alt + Tab* to exit the game and then just simply click on the **X** in the top-right.

### Note

For more information on exporting projects to different platforms, such as iOS and Android, visit <https://docs.unrealengine.com/latest/INT/Engine/Basics/Projects/Packaging/index.html#distribution>.

If you're specifically interested in exporting to iOS, refer to <https://docs.unrealengine.com/latest/INT/Platforms/iOS/QuickStart/index.html>.

In addition, if you're specifically interested in exporting to Android, visit <https://docs.unrealengine.com/latest/INT/Platforms/Android/GettingStarted/index.html>.

For specifics on working with mobile projects, read *Learning Unreal Engine Android Game Development* and *Learning Unreal® Engine iOS Game Development*, both by Packt Publishing.

## Creating an installer for Windows

As I mentioned previously, having multiple folders that need to be included with our .exe is somewhat of a pain. Rather than giving people a .zip file and hoping that they will extract it all and then keep everything in the same folder, I'd rather have the process be automatic and give the person an opportunity to have it installed, just like a professional game. With that in mind, I'm going to go over a free way to create a Windows installer.

### Getting ready

Before we start working on this, we need to have an exported project. If you do not have that already, follow the previous recipe all the way to completion.

### How to do it...

The first thing we need to do is get our setup program. For our demonstration, I will be using Jordan Russell's Inno setup. Let's get started by first downloading it:

1. Go to <http://jrsoftware.org/isinfo.php> and from there, click on the **Download Inno Setup link**.

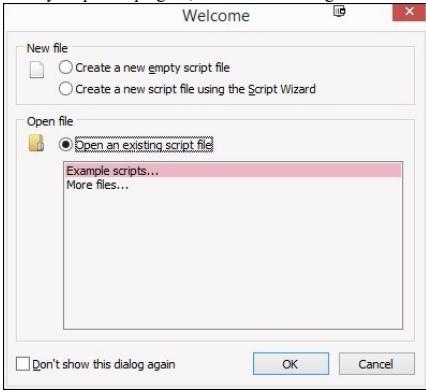
2. From here, click on the **Stable Release** button and select the **isetup-5.5.6.exe** file. Once it's finished, double-click on the executable to open it, clicking on the **Run** button if it shows a Security Warning and **Yes** to allow changes.

3. Under the **Select Setup Language** window, leave it as English and then click on **Ok**.



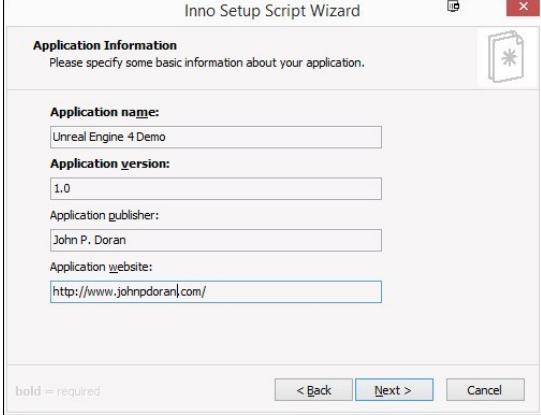
4. From here, run through the installation, making sure to uncheck the **Install Inno Setup Preprocessor** option since we won't be using it. Upon finishing, make sure that **Launch Inno Setup** is checked and then press the **Finish** button.

When you open the program, it will look something similar to this:



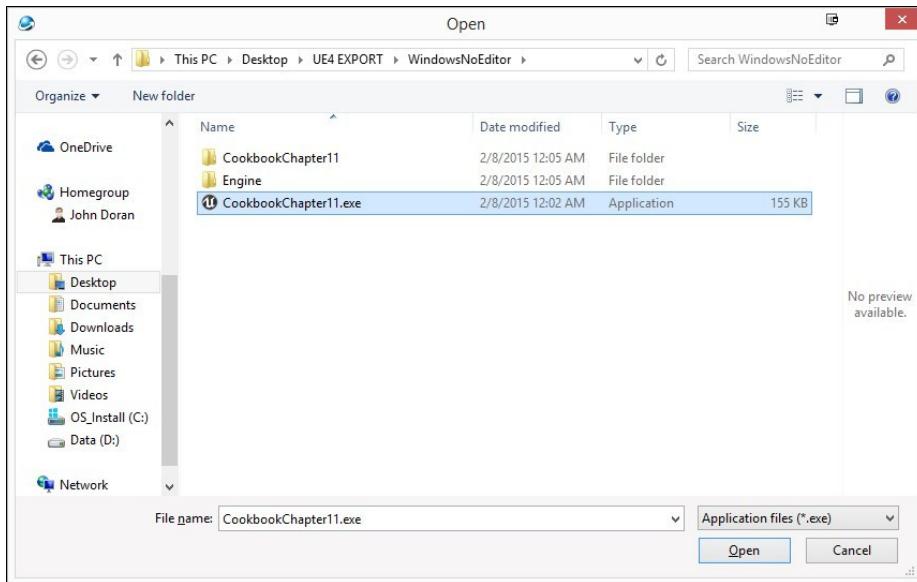
5. From here, choose **Create a new script file using the Script Wizard** and then select **OK**.

6. Now click on the **Next** button, and you'll come to the **Application Information** section. Fill in your information and then click on **Next**.

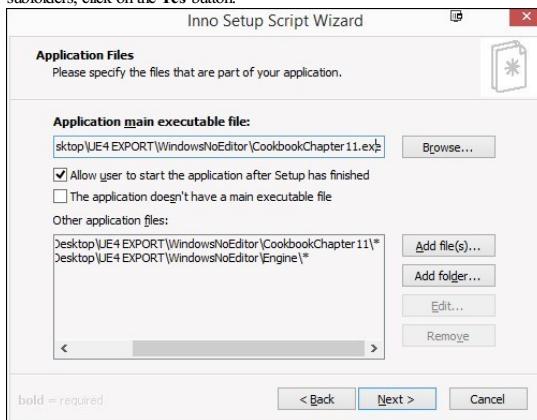


7. Next, you'll come up to some information about the **Application folder**. In general, you will not want to change this information, so I'd click **Next**.

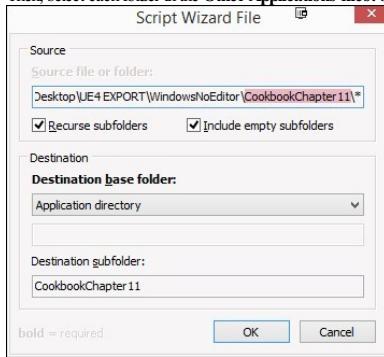
8. From here, we'll be brought to the **Application Files** section, where we need to specify the files we want to install. Under **Application main executable file:**, select **Browse** to go to the location of your **Export** folder, select the **.exe** file, and then click on **Open**.



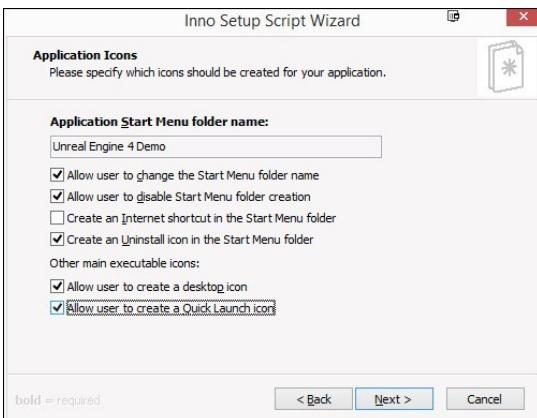
9. Now, we need to add in the project's folder (the same name as the .exe file). Click on the **Add Folder...** button, select the project's name folder, and then click on **OK**. In the popup that appears and asks whether you should include files in the subfolders, click on **Yes**.
10. After that, we need to also add in the `Engine` folder. Click on the **Add Folder...** button, select the `Engine` folder, and then click on **OK**. In the popup that appears and asks whether you should include files in the subfolders, click on the **Yes** button.



11. Then, select each folder in the **Other Applications files:** section and click on the **Edit** button. From here, set the **Destination subfolder** property to the same name as the folder and then click on **OK**.



12. Make sure that you do the same with the `Engine` folder as well and then click on the **Next** button.
13. In the next menu, check whichever options you'd like, and then click on **Next**.



14. Now, you'll have an option to include a license file, such as a EULA or whatever your publisher may require, and any personal stuff you want to tell your users before or after installation. The program accepts .txt and .rtf files. Once you're ready, click on the **Next** button.
15. Next, they'll allow you to specify what languages you want the installation to work for. I'll just go for English, but you can add more. Afterward, click on **Next**.
16. We need to set where we want the setup to be placed as well as the icon for it or a password. I created a new folder on my desktop called UE4DemoSetup and used it. Then click on **Next**!



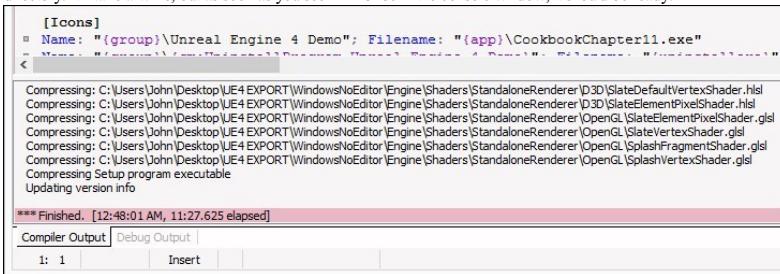
#### Note

If you want to include a custom icon but don't have a .ico file, you can visit <http://www.icoconverter.com/> to create one from an image file.

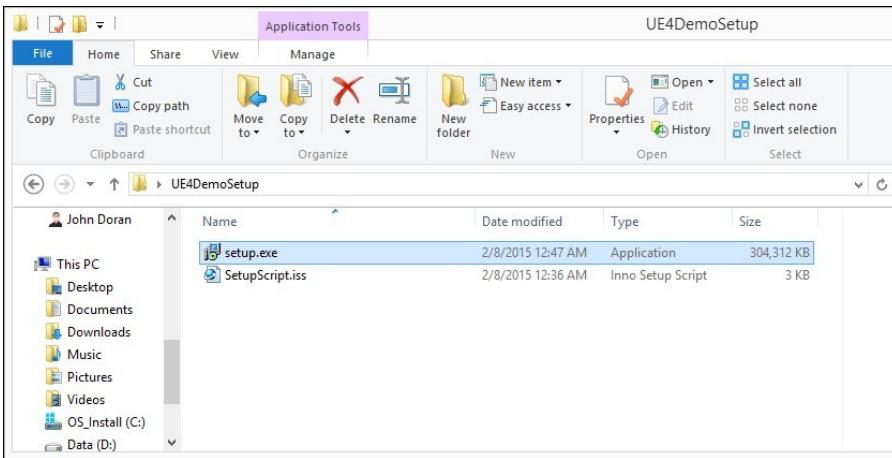
17. Next, you'll be brought to the **Successfully Completed Script Wizard** screen. After this, click on **Finish**!



18. Now it will ask you whether you'd like to compile the script immediately. Select **Yes**. It'll also ask you whether you want to save your script. Again, click on **Yes**. I saved it to the same folder as my exporting directory. It'll take a while, but as soon as you see **Finished** in the console window, it should be ready!



19. If you go to the same place as your export folder, you should see your installer!



20. If you run the project, you should see something similar to this:



With this, we now have a working installer for our game and a single file that we can share with anyone. This will install everything correctly!

## Part 3. Module 3

### *Learning Unreal Engine Android Game Development*

*Tap into the power of Unreal Engine 4 and create exciting games for the Android platform*

## Chapter 1. Getting Started with Unreal 4

Greetings! If you have picked up this book, chances are you are interested in developing games on Android devices using UE4. This book explains all that you need to know in order to get started using UE4. From the very basics, such as downloading and installing the software, to the more advanced, such as packaging your finished game and porting it onto your android device, everything will be covered in this guide. You can develop games for a wide variety of platforms on UE4, but the one we will be focusing on is Android.

In this introductory chapter, we will cover the following topics:

- UE4 and the features it provides
- Downloading and installing UE4
- The Engine Launcher and its user interface
- What to expect from the guide

## What to expect

Learning how to use a game engine can be a daunting task; you just do not know where to begin, and UE4 is no exception. However, once you get the hang of it, you will quickly find out how extremely powerful and intuitive it really is. And what better to teach you how to use a game engine than by actually making a game in it? This book will teach you all that you need to know for you to be able to develop games on Android platforms using UE4, and make an actual functional game in the process. The reason behind this is simple; just talking about the features offered by UE4 and demonstrating them one at a time is not very effective when learning how to develop a game. However, if one were to explain those very features by implementing them in a game, it would be much more effective, since you would get a better understanding of how each feature affects the game and each other.

The game we are going to make in this guide is called **Bloques**, which is a first person puzzle game, wherein the main objective of the player is to solve a series of puzzles in order to progress. As the player progresses, the puzzles get progressively more complex and complicated to solve. As for the scope of the game, it will contain four rooms, each with a puzzle that the player has to solve in order to progress to the next room.

The rationale behind picking a puzzle game is that puzzle games have more complicated systems, in terms of scripting, and level design. To put it in the context of the guide, things such as scripting with blueprints and level design will be much better demonstrated through a puzzle game. Although the game will be explained thoroughly in the subsequent chapters, a high-level breakdown of the game's features are as follows:

- A fully rendered playable 3D environment, with four rooms.
- Interactive environmental elements.
- The player has to solve a series of puzzles in each of the rooms in order to progress to the next. As the player progresses, the puzzles get more complex and harder to solve.
- The game will be optimized and ported to Android.

This guide aims to set the foundation of UE4, upon which you can build your knowledge further and be in a position to actually develop that game you always wanted to make!

A final word of advice is practice! Tutorials and guides can only do so much. The rest is up to you. The only way to truly master UE4, or anything for that matter, is practice. Keep experimenting, keep making small prototypes, keep yourself up-to-date with the latest developments and news, and keep interacting with the community.

## System requirements

Before you jump in and download UE4, you first need to ensure that you have a system capable of running it in the first place! UE4 works on both Windows and Mac OS X. The following are the system requirements for each:

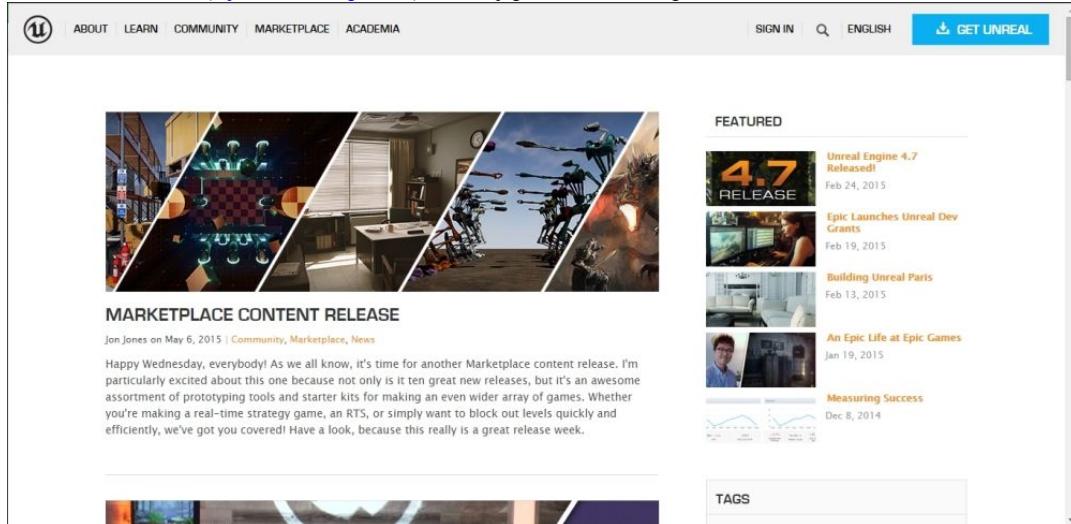
- Windows 7/8, 64-bit (Or Mac OS X 10.9.2 or later)

- .NET 4.0
- DirectX 10 (Mac: OpenGL 3.3)
- 8 GB of RAM
- Quad-core Intel or AMD, 2.5 GHz or faster
- NVIDIA GeForce 470 GTX or AMD Radeon 6870 HD series or higher
- At least 9 GB Hard Disk Space (8 GB for Mac OS X)

## Downloading and installing UE4

The process of downloading and installing UE4 is pretty straightforward; just follow these steps:

1. Go to Unreal's official website (<https://www.unrealengine.com/>). The home page looks like the following screenshot:



Everything you need to know regarding UE4, you can find here—including the latest news, the latest version of the engine, blog updates, latest Marketplace entries, and so on. As of 2015, the engine has been made free to download.

In addition to the UE4 homepage, it is recommended that you visit <https://docs.unrealengine.com/latest/INT/>. It is full of documentation and video tutorials on how to use UE4. Epic boasts a large, active, and friendly community, always willing to help anyone facing a problem via the forums.

### Note

You can access the forums by hovering over the **COMMUNITY** tab on the home page until the menu drops down, and then clicking on **Forums**, or you can simply visit <https://forums.unrealengine.com/>. Alternatively, you can also seek help via **AnswerHub** by visiting <https://answers.unrealengine.com/>.

2. From the home page, click on the **GET UNREAL** button on the right of the screen. Clicking on it will bring you to the subscription page, shown here:

The screenshot shows the 'Join the Community' sign-up form for Epic Games. The form consists of several input fields: 'FIRST NAME' and 'LAST NAME' in separate boxes, a 'DISPLAY NAME' field, an 'EMAIL' field, and a 'PASSWORD' field. Below these fields is a checkbox labeled 'I have read and agree to the terms of service.' followed by a 'Sign Up' button. At the bottom of the form, there is a link 'Have an Epic Games account? Sign In'.

3. In order to download and install UE4, you have to create an account. To create an account with Epic Games, just fill in the required information, and follow the instructions.
4. To download the Engine Launcher, simply sign in. On your account page, you can access your profile, billing history, previous transactions, and so on.

The screenshot shows the Epic Games account dashboard. At the top, there are links for ABOUT, LEARN, COMMUNITY, MARKETPLACE, ACADEMIA, ACCOUNT, LOG OUT, and ENGLISH, along with a 'GET UNREAL' button. The user's name, Nitish Misra, is displayed with an edit profile link. Below the user info, there is a placeholder for a profile picture. To the right, the account balance is shown as \$0.02. A navigation bar below the user info includes links for Download, Profile, Billing, Transactions, Redeem Code, and Password. Under the Download section, three main options are listed: 'Get Unreal Engine' (with a 'Download' button and a note about choosing platform: Windows | Mac), 'Get Unreal Tournament' (with a 'Download' button and a note about choosing platform: Windows | Mac), and 'Get UE4 Full Source Code' (with a note about linking GitHub). A 'NEED HELP?' link is at the bottom right.

5. Now that you have your account set up, you can download UE4. You can download either the Windows version or the Mac version, depending upon your setup. To download, under **Latest Download**, click on the **Download** button and you will download the Engine Launcher.
6. To run the installer, simply double-click on **UnrealEngineInstaller-\*version number\*.msi** if you are using Windows or **UnrealEngineInstaller-\*version number\*.dmg** if you are using a Mac. Follow the steps to install the Engine Launcher.
7. After the installation is complete, run the Launcher. You should encounter the following screen.

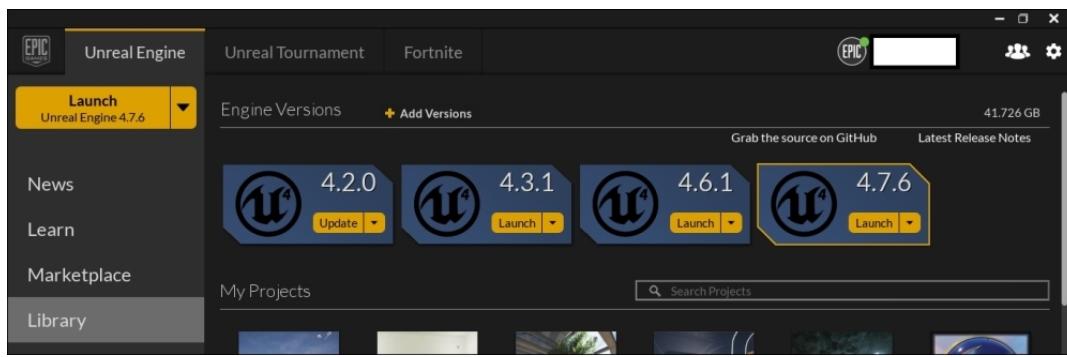


This is the login screen. Just type the e-mail address you used to subscribe and your password, then either click on the arrow button next to **PASSWORD** or hit the *Enter* key, to log in.  
8. Logging in will open the Engine Launcher. We will discuss it and its functionalities in detail later on, but for now, all you need to do is click on **Library** and click on the **Add Versions** button next to **Engines**. Doing so will create a slot. You can select a version number using the version dropdown in the version slot you added, then you can click on the **Install** button and the version of UE4 that you selected will begin downloading.

That is it! You have now downloaded and installed UE4 on your PC (or Mac). To launch the engine, simply click on the **Launch** button on the top-left corner of the Launcher, below the account name, and you are good to go. You can also launch previous versions of the engine if you require. Clicking on the downward arrow next to the **Launch** button will open a menu, with all of the versions of the engine listed, and to launch them, simply click on the version which you wish to run.

The screenshot shows the Unreal Engine Launcher window. The left sidebar has tabs for 'Unreal Engine', 'Unreal Tournament', and 'Fortnite'. The 'Unreal Engine' tab is active and shows a dropdown menu with 'Launch' and 'Unreal Engine 4.7.6'. Other versions listed are 4.7.6, 4.6.1, 4.3.1, and 4.2.0. Below this is a 'Learn' section with links for Forums, AnswerHub, and Roadmap, along with social media icons. The main content area has a search bar 'Search For Help'. It features sections for 'Begin Your Journey' (with 'Get Started with UE4' showing a 3D rendering of two chairs around a table), 'Artist Quick Start' (showing a 3D scene with various objects), 'Level Designer Quick Start' (showing a 3D scene with a character), and 'Programmer Quick Start' (showing a code editor with C++ code). At the bottom, there is a 'Broaden Your Horizons' section.

Alternatively, you can also click on the **Library** button, and select which engine to run from there. All of the versions installed on your system will be listed, and you can simply launch any version from the list by clicking on the **Launch** button.



But hold on! We have a few more things to discuss before we are ready to start using UE4. Let's take a quick look at the directory structure.

## The Windows directory structure

The default location where UE4 is installed is `C:\Program Files\Unreal Engine`. You can change this if you wish, during the installation process. Upon opening the directory, you will find that each version of the Engine has its own separate folder. Say, you have versions 4.1, 4.2, and 4.3 of UE4 installed on your system. You will find 3 separate folders for all three versions, namely 4.1, 4.2, and 4.3. The following screenshot will give you a better idea:

Name	Date modified	Type
4.1	27/08/2014 17:50	File folder
4.2	09/08/2014 21:33	File folder
4.3	09/08/2014 22:17	File folder
4.4	31/12/2014 15:03	File folder
4.6	22/12/2014 04:12	File folder
DirectXRedist	09/08/2014 21:38	File folder
Launcher	06/09/2014 21:46	File folder

Each version of the Engine gets its very own folder. Apart from that, there are two other folders, namely `DirectXRedist` and `Launcher`.

### Windows DirectXRedist

`DirectXRedist` is where the `DirectX` files are located. The folder also contains the installation file, from which you can install `DirectX`.

### Launcher

The `Launcher` folder contains all the files for the Engine Launcher. The `Launcher` folder contains the following subfolders:

- `Backup` : UE4 has an excellent feature that lets you create backups of your work. Should a developer make an unfixable or difficult-to-fix mistake or if the Engine crashes mid development, instead of having him/her do all the work all over again, a backup of their work will be stored in the `Backup` folder, so they can pick up where they left off.
- `Engine` : This folder contains all of the code, libraries, and content that makes up the engine.
- `PatchStaging` : Every now and then, Epic will release a new version of UE4. As of 2015, the latest version out is 4.7.6. (The preview version of 4.8 is available at the time of writing). When you are in the process of download, all of the data of the currently downloading version/versions of UE4 gets stored in the `PatchStaging` folder.
- `VaultCache` : As will be explained later in the chapter, all that you need to know right now is that everything you purchase in the Marketplace is contained in the `Vault`. The `VaultCache` contains all of the purchased items' cache files.

### 4.X folders

Before we talk about the `4.X` folders, you should know all versions of UE4 (4.1, 4.2, 4.3, and so on) work independent of each other. This means you do not require the previous version to run the later versions. For example, if you wish to run version 4.4, then you do not need to download versions 4.0, 4.1, 4.2, and 4.3 in order to run it. You can simply download version 4.4 and use it without any problems. This is the reason why there is a separate folder for every version of Unreal 4, each version is treated like a separate entity.

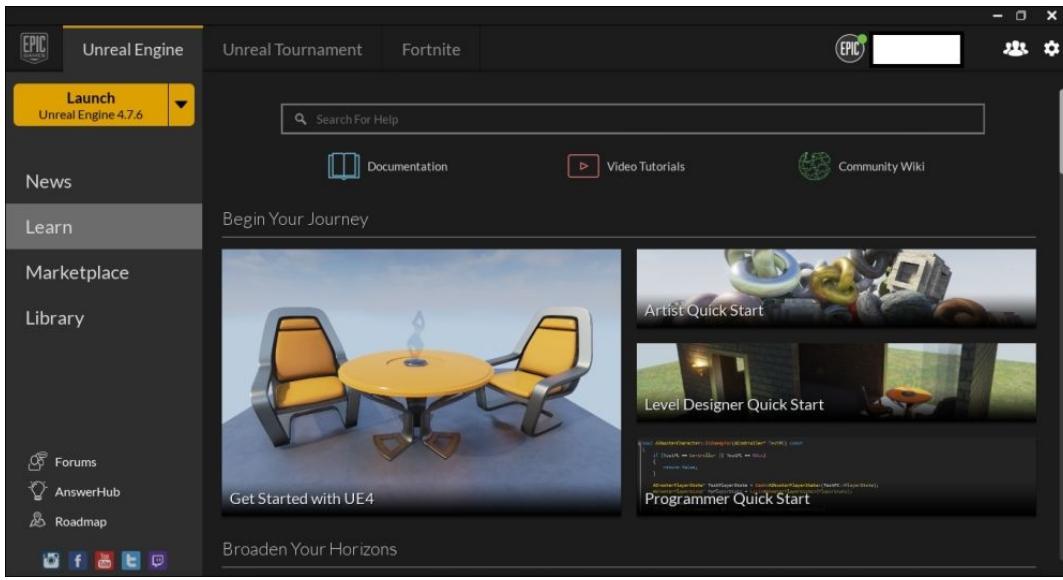
All of the `4.X` folders, although separate, contain the same set of subfolders, hence they are grouped together. The following are the subfolders:

- `Engine` : Similar to the Launcher's `Engine` folder, this contains all of the source code, libraries, assets, map file, and more that make up the Engine.
- `Samples` : UE4 has two sample maps, Minimal Default, and Starter Map. This folder contains all the content including assets, blueprints, and more.
- `Templates` : UE4 offers templates for various genres of games, for example first person, third person, 2D side scroller, top down, and many more. All of the content for each of these genres and the source code are contained here.

## The Engine Launcher

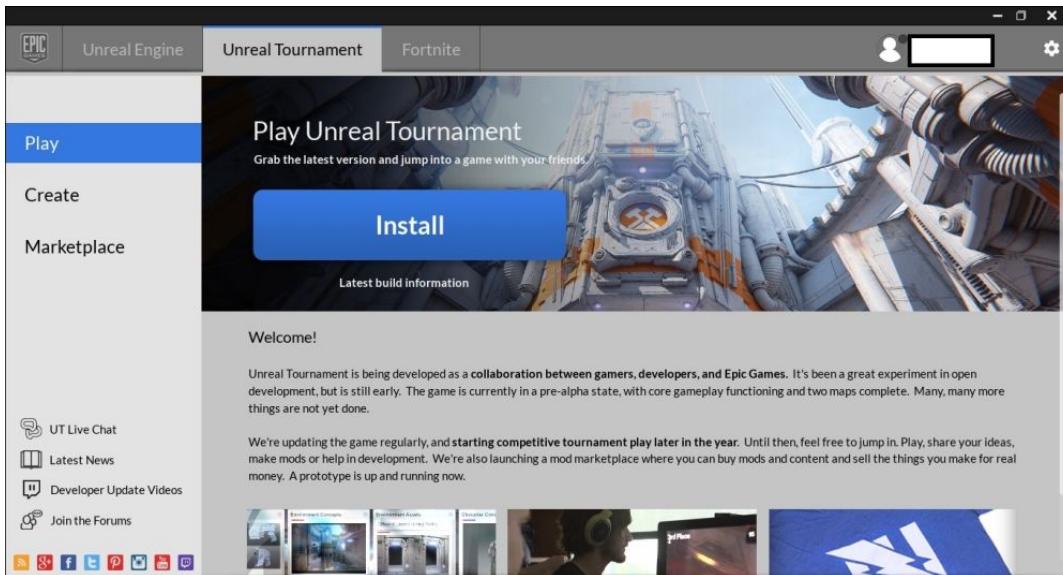
The Engine Launcher is a window that opens up after you run the engine. It is full of features and resources that can prove to be quite useful for you. Firstly, we will look at the Engine Launcher's user interface, its breakdown, where everything is located, its functionalities, and so on.

Upon opening the Engine Launcher, you will see the following window:



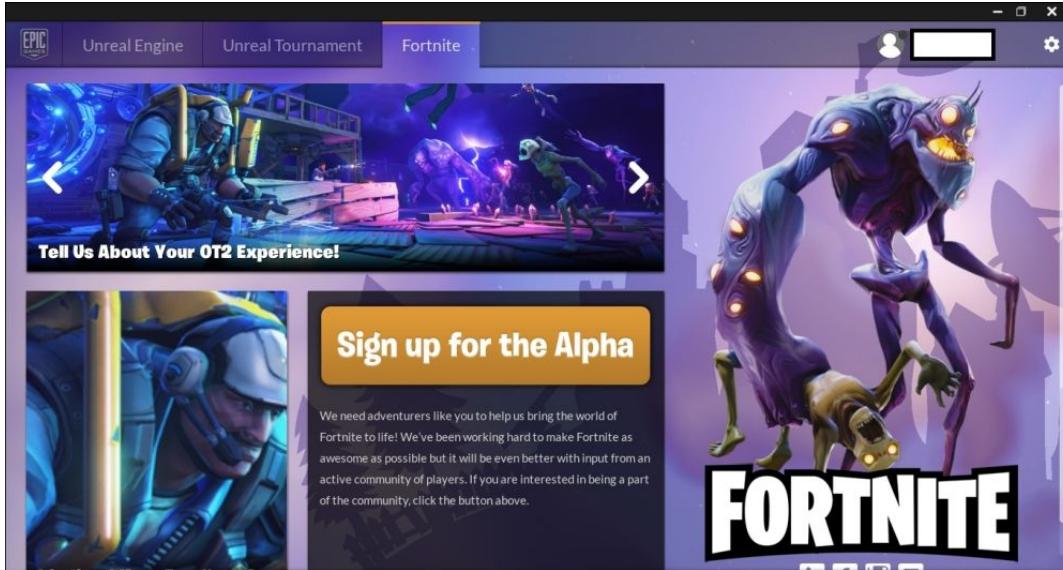
At the top left, there are three tabs, **Unreal Engine**, **Unreal Tournament**, and **Fortnite**. The Unreal Engine tab is open by default, and contains what you see in the preceding screenshot.

The **Unreal Tournament** tab is where you find information and links regarding the latest Unreal Tournament game.



As mentioned previously, Epic's latest project, Unreal Tournament is a project in which Epic accepts and encourages content from the community, such as weapon skins, player skins, levels, and so on. From here, you can download the latest Unreal Tournament, purchase content created by other community members and also get all the latest news and updates regarding Unreal Tournament.

The last tab is the **Fortnite** tab. Epic is currently working on another project, namely Fortnite.



At the time of writing, the Alpha version is available. You can sign up for it, give the developers feedback, and access the official Facebook page, Twitter page, Instagram account, and Twitch streams from this tab.

At the top-right corner, on the panel with the tabs, are two buttons, the **friend list** button, and the **Settings** button. When you click on the **friend list** button, it opens a window, where you can manage your friend list, like adding and removing friends, seeing who is online, and so on. You can also set your status to either **Online** or **Away**.

The next button is the **Settings** button, wherein you can find certain options regarding the Engine Launcher, such as accessing the support page, viewing the launcher logs, exiting the launcher, and so on.

On the top left, is the **Launch** button which, as previously discussed, launches the engine. Below it are several panels; each containing something different. Let's look at each of these panels individually.

## News

The **News** panel contains all the latest news and updates regarding UE4 and Epic in general. From here, you can access the latest articles regarding the current/newest version of UE4, the latest content that has been released in the Marketplace, the latest tutorial series that are out regarding a specific topic, Twitch recaps, and much more. This is the place to be, to stay up to date on the events surrounding Epic and UE4.

## Note

The news section is updated regularly, so checking the news section every once in a while is highly advisable.

## Learn

As the name suggests, this is where you can find all the tutorials and documentation regarding UE4. The **Learn** section offers video tutorials, such as how to use Blueprint, written tutorials, which have step-by-step instructions on how to use UE4, and finally there are Gameplay Content Examples, which are project files with everything already set up, such as the level, lighting, assets, as well as the Blueprint scripts so that you can personally see what does what and can experiment.

At the top of the **Learn** section are three buttons, namely, **Documentation**, **Video Tutorials**, and **Community Wiki**. Clicking on **Documentation** will send you to Epic's official Unreal Engine 4 Documentation page, covering various topics such as how to use the Editor, Blueprint, Matinee, and so on.

The **Video Tutorials** button will take you to Epic's video tutorial page, where everything is neatly categorized. Each category has a certain number of series. A series contains a set of video tutorials covering a certain topic. For instance, the Blueprint category currently has six series, including introduction, how to create an inventory, third person game creation, and so on.

Finally, the **Community Wiki** is a living, breathing wiki page, where the people can post tutorials, code, projects, plugins, and more. It is a great way of getting user content and finding tutorials created by other developers. It is also worth mentioning here that Epic is currently in the process of developing their latest project, Unreal Tournament. A great thing about this title is that they are also accepting and implementing content created by the community. This includes developing the core game functionality, levels, characters, guns, HUD graphics, and so on.

## Note

Should you be interested in contributing to the project or are interested in the whole thing, just go on the Unreal Tournament section of Wiki and it will give you all that you need to know regarding the project. Below this, are the various tutorials, categorized based on their types, that you can view/download and learn more about UE4 and its features.

## Marketplace

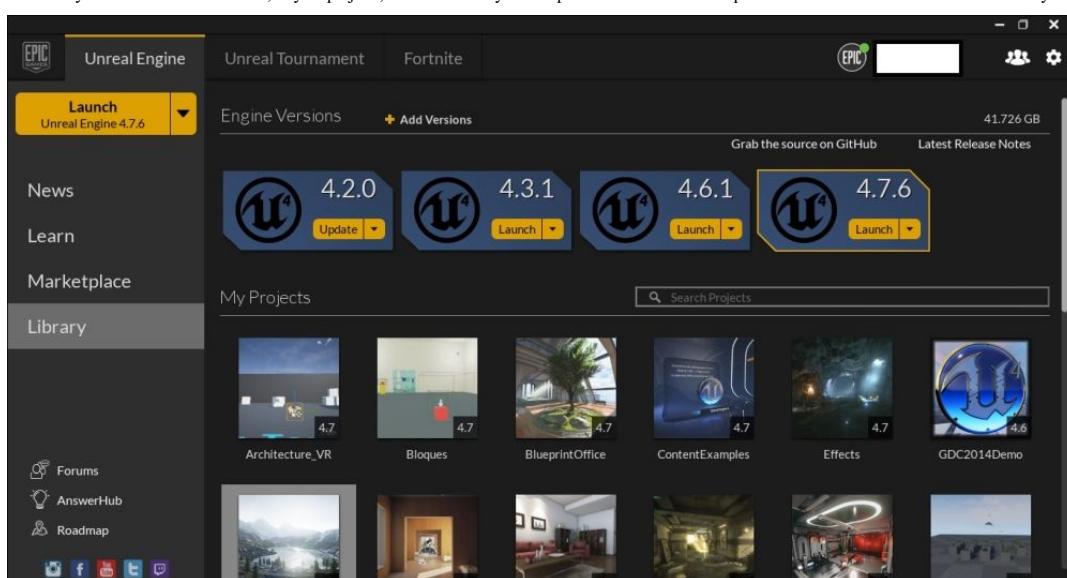
The Marketplace is where developers can purchase assets. Developers who lack the manpower or resources to create assets can purchase and use them in their game. These include meshes, materials, animation sets, rigged characters, audio files, sound effects, projects, and tutorials, to name a few. Certain items in the Marketplace, such as the ones by Epic themselves are free. They are mostly tutorial project files, with a sample level already setup to showcase various features offered by UE4. These project files also have all of the level blueprints set up and implemented, so that users can see them, and experiment with them until they get the hang of it. Other items in the Marketplace, created by users, cost money. The assets you can purchase are neatly categorized, based on the type of content, for your convenience.

Apart from buying assets, you can also submit your own content in the Marketplace and earn some money from it. Clicking on the **Submit your content** hyperlink on the top-right corner of the Marketplace screen will open up Epic's Call for Submission page. The Call for Submission page has all the information regarding submitting content.

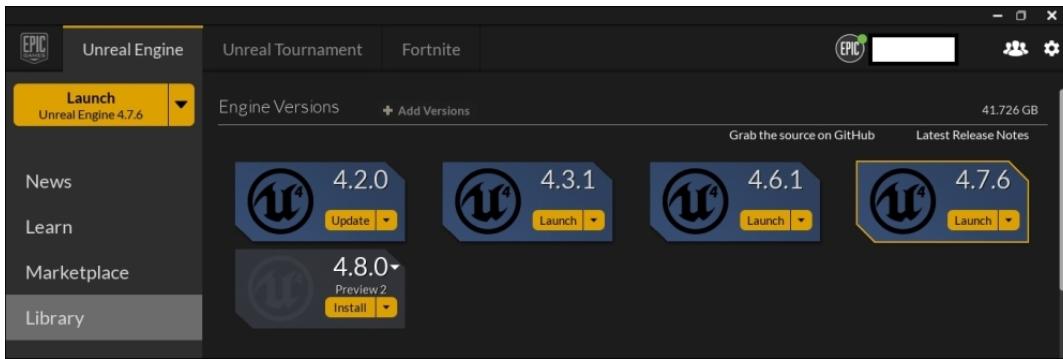
It also has the Marketplace Business Terms, which has all the information regarding things like how the revenue from the sales will be split, how you will get paid, when you will get paid, and so on. It also has the Marketplace Submission Guidelines, which explain things like the submission process, what you need to submit, the resolution for the screenshots, and more. You can also get more information on the submission process and get feedback on your content by posting on the Forums.

## Library

The Library is where all versions of UE4, all your projects, and all the items you have purchased from the Marketplace are listed. Let's look at it a bit more closely.



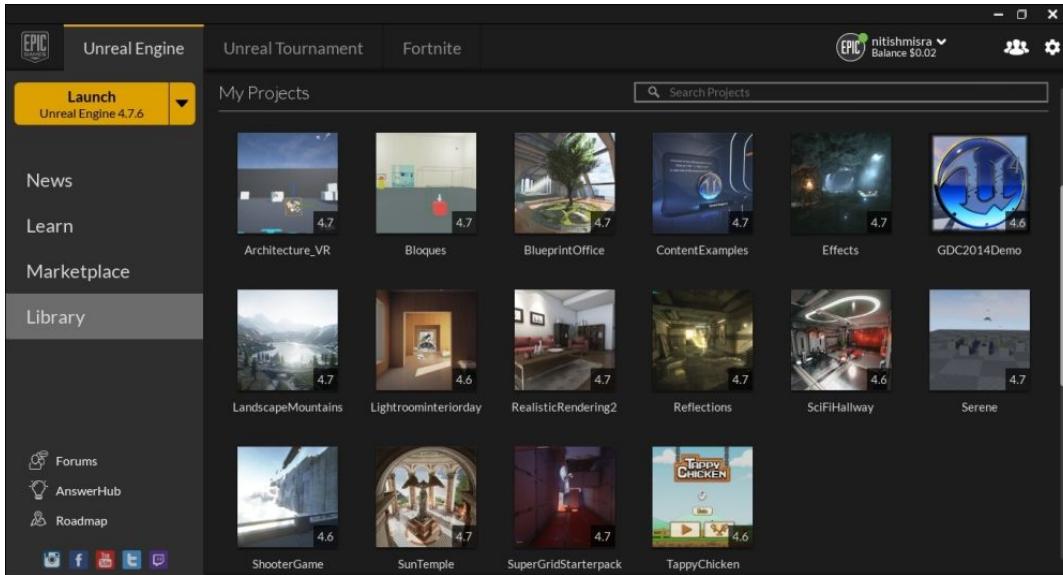
Library has 3 sections, **Engine Versions**, **My Projects**, and **Vault**. The **Engine Versions** section displays all versions of UE4 currently installed on your system. You can launch any version of the Engine listed from here. Additionally, you can also download the latest version or previous versions. To do so, simply click on **Add Versions** at the top of the panel, right next to **Engines Versions**. Clicking on it will create a slot for the version you wish to download.



As you can see in the previous screenshot, clicking on the **Add Versions** button created a slot for the latest version of Unreal 4, which in this case, is 4.8.0 (although it is only the preview version). To download, simply click on the **Download** button and it will start downloading.

Additionally, you can remove versions of UE4 that you do not require. For example, if you have the latest version, it would be understandable if you wish to remove previous or older versions of the engine to make space on your hard drive. To do so, simply hover your cursor over the top-left corner of the version slot until you see an **x**. Once you see the **x**, simply click on it and the corresponding version of UE4 will be uninstalled. Another way of uninstalling is by clicking on the downward arrow button next to **Launch**, which opens a drop-down menu; from this, select **Remove** and the Engine Launcher will uninstall that version.

The second part of Library is the **My Projects** section. In this section, all the projects you have created are displayed.

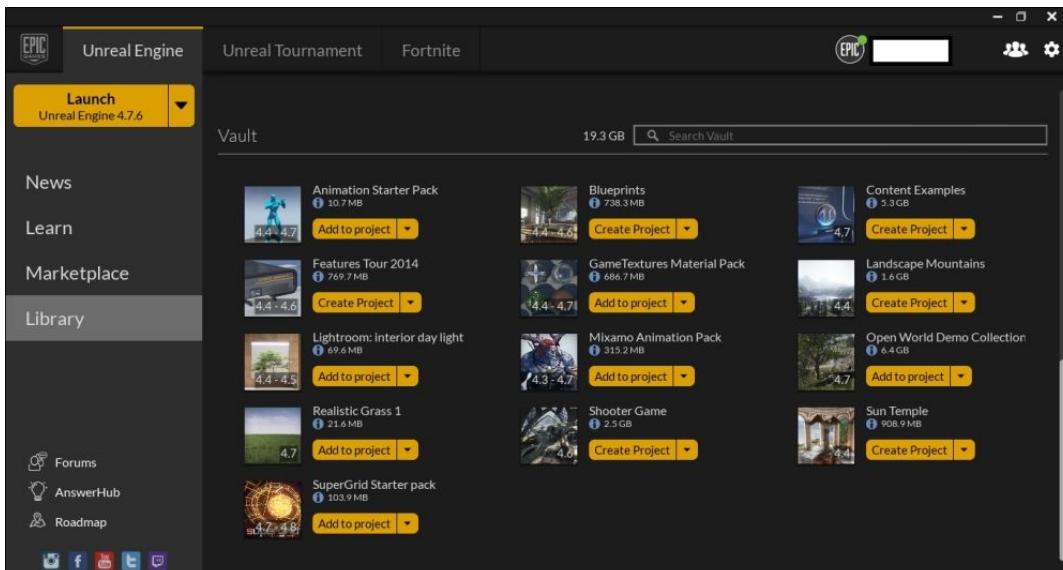


The projects are categorized alphabetically. At the top-right corner is the search bar. In the preceding screenshot, there are relatively few project files; therefore, it is easy to find a specific project. However, if you have lots of projects, it might be harder to find the project you are looking for. In that case, you can type the name of the project you require in the **Search Projects** tab and it will find it for you.

At the bottom-right corner of a project's thumbnail, you can see in which version of the engine the project was created. For instance, in the previous screenshot, the project **Effects** was created with version 4.0 of UE4. If you open that project file, the Launcher will launch the version 4.0 of UE4. If, however, you do not have the corresponding version, then upon launching the project, you will be asked to select which installed version you wish to launch the project file in. After you have made your choice, it will then convert the project to be compatible with the version you selected and launch it. However, always be careful when converting a project, as some unexpected issues might occur. It is advisable to create a backup copy of the project before you convert it.

To launch a project, double-click on the thumbnail. Apart from opening a project, there are other operations you can perform with the projects. Right-clicking on the thumbnail opens a menu. Clicking on **Delete** will delete the respective project. Clicking on **Clone** will create a copy of the project file, and clicking on **Show in Folder** will open the folder where all the project files are stored on your system.

Finally, there is the **Vault**. All of the items you have purchased in the Marketplace are contained in the **Vault**.



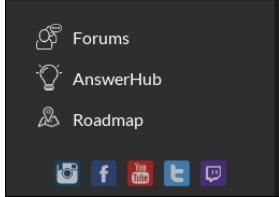
The preceding screenshot demonstrates what **Vault** looks like and how the items are arranged. On the left is the thumbnail of the item, followed by the name of said item. Below the name is the amount of space occupied by that item. The blue **i** icon below the name is information regarding compatibility. Hovering the cursor over it will show you which versions of UE4 that item is compatible with.

The compatibility is also displayed at the bottom-right corner of the thumbnail, similar to **My Projects**. Let's look at the first item in the Vault as an example, that is, the Animation Starter Pack. On the thumbnail, **4.4-4.6** is written. This means that the Animation Starter Pack is compatible with versions 4.4, 4.5, and 4.6.

You may have noticed that certain items have the **Add to Project** option, while others have the **Create Project** option. Items such as animation packs, assets, materials, and audio files can be added to any project you have already created, and you can use them in your level. Projects and Showcases have the **Create Project** option. Once you click on it, it will create a project and will be displayed in **My Projects**, from where you can open it. Additionally, you can verify or remove any item by clicking on the downward arrow, and clicking on the corresponding option from the drop-down menu.

## UE4 Links

The final element in Launcher's user interface is the UE4 Links, located at the bottom-left corner. UE4 Links unsurprisingly contains hyperlinks to different web pages.



Let's look at each of them closely:

- **Forums** : UE4 has a large and active community. The forum is a great place to meet other developers, share your ideas, show your work in progress and get feedback, team up with other members and develop a project, and so on.

The forums discussion board is neatly categorized, based on the topic you wish to discuss. There is the Development Discussion section, where you can talk about Blueprints, Animation, Rendering, C++ Gameplay Programming, and so on. Then, there is the Community section where you can showcase your work in progress and get feedback, and also see other people's work and give them feedback. Following that is the UE4 for Schools section, which is dedicated to students and teachers to discuss UE4 and the education program. Finally, there is the International section, where you can interact with developers from your demographic. Recruiting and teaming up with people for a project is easier and much more convenient since almost all of the members are from the same general area.

- **AnswerHub** : Sometimes, you may face a problem or issue, or have a very specific question that needs to be answered, which you would not find in any documentation or tutorial. In such a scenario, the best course of action is to seek help from others and/or the Epic staff themselves. AnswerHub is a great forum wherein you can resolve any issues, technical or otherwise, with the help of the UE4 community or from the Epic staff. To do so, simply login, post your question, and wait for someone to reply.

Alternatively, if you are feeling generous, you give back to the community by helping others resolve any issues that they might be facing, and build a strong reputation in the process.

- **Roadmap** : The community is an important part of UE4. The developers at Epic wanted to include the community as much as possible and be transparent with their development process. Nowhere is this more evident than in the Roadmap. The Roadmap lists out features that are in the process of development and gives an estimation as to when these features will be deployed.

Epic's social icons are at the very bottom. From left to right, they are as follows:

- **Instagram** : You can follow Epic's Instagram profile, where they post photos and videos regarding UE4, such as environments, events, materials, and so on. Their Instagram link is <https://instagram.com/UnrealEngine/>.
- **Facebook** : Clicking on this will take you to Unreal Engine's official Facebook page, where, as with Instagram, all of the updates regarding UE4 and Epic are posted. The link to their Facebook page is <https://www.facebook.com/UnrealEngine>.
- **YouTube** : This will take you to the official Unreal Engine YouTube page, where you have access to all of the previous Twitch streams, Tutorials, and so on. The link to their YouTube page is <https://www.youtube.com/user/UnrealDevelopmentKit>.
- **Twitter** : This will take you to the official Twitter page, should you want to follow them on Twitter. The official Unreal Engine Twitter handle is @UnrealEngine
- **Twitch Stream** : Every Thursday at 2 pm EST (at the time of writing), the Unreal team has a Twitch stream where they discuss the latest news, talk about the latest version of UE4, what features have been added or have been amended, and answer any questions asked by the viewers watching the stream

## Summary

You have now taken the first step towards becoming an UE4 Android developer. This chapter was just the tip of the iceberg; there are still plenty of things to cover.

In this introductory chapter, we covered what UE4 is and the features UE4 provides. You also learnt how to download and install UE4. Now you're well versed with the Engine Launcher, its UI, and functionality.

All these topics provide a nice segue to our next chapter, where we will be covering the Editor. Before we start using it, it is important that you know and understand what it is, how to navigate through it, and its UI and functionality. The next chapter is dedicated to just that. So, without further ado, let's move on to the next chapter.

## Chapter 2. Launching Unreal Engine 4

Now that you know how to download and install UE4, you should have everything setup and be ready to begin making games. However, there is an important topic that needs to be covered before we start making our game, and that is the **Editor**. The Editor is where all the magic happens. It is where you make the game. So, it is important that you know about the Editor, its functionalities, the user interface, and how to navigate through it before we go any further. So, this chapter is devoted to taking you through it.

We will be covering the following topics:

- What the Editor is
- Its user interface
- Navigating through the Editor
- Hotkeys and controls

## Meet the Editor

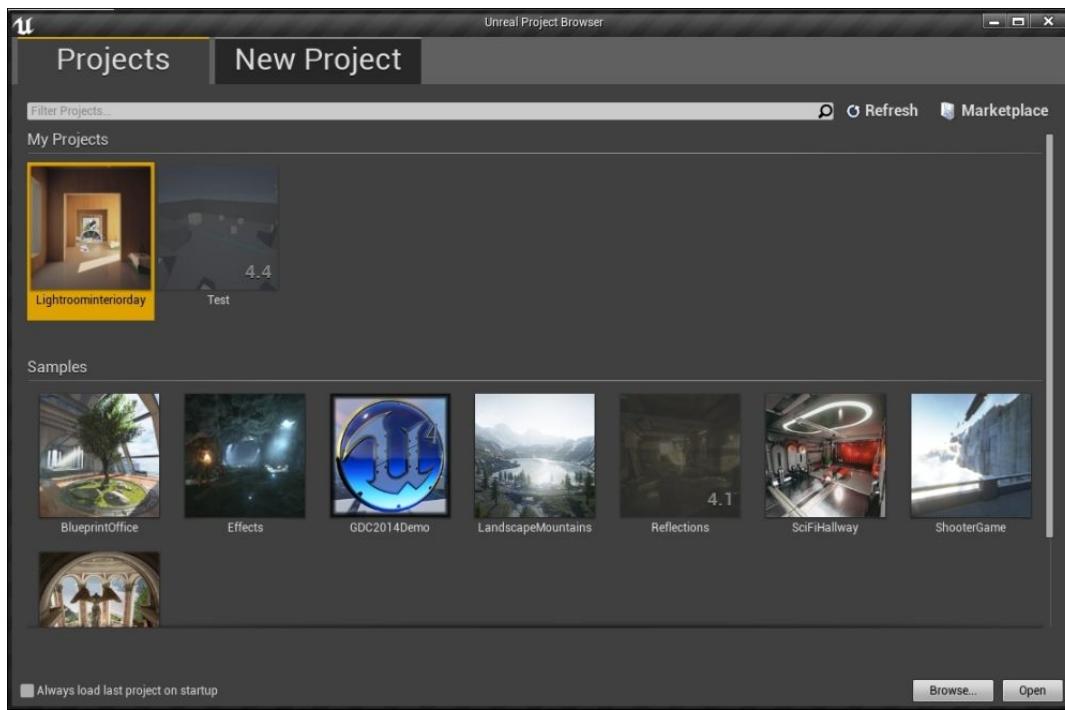
The Editor is where you make the game. All of the assets you create for your game are implemented via the Editor. You set up your environment and the levels in the Editor, and all of the code sequences you create can be tested here; the testing, debugging, and packaging of your game is done here as well.

Needless to say, it is important you understand what the Editor is, get familiar with its user interface, and know how to navigate through it before we can go any further. Finally, to improve your workflow, you should also be familiar with some hotkeys.

## The Unreal Project Browser

When you launch UE4 via the Engine Launcher, unless you have opened a project directly from the library, the Unreal Project Browser will open. In the Unreal Project Browser, you can see a list of all the Unreal projects that you have already created, and you can open whichever one you wish. You can also create a new project from here.

In the next chapter, we will cover what a project is; for now, we will only focus on the **Unreal Project Browser** and its user interface, as shown in the following screenshot.



At the top, just below the tab bar, are two tabs, namely **Projects** and **New Project**. Each of these tabs contains certain features that we will go through.

In the **Projects** tab, you can see and open any project you have stored on your system. At the top is the search bar. If you have a lot of projects and have difficulty finding a particular one, you can simply type in the name of the project you wish to open in the search bar and it will display projects that match the name you have entered.

To the right of the search bar is the **Refresh** button. If you have made any purchases from the marketplace, it will not reflect in the browser. To update the project list, click on the **Refresh** button. Next to the **Refresh** button is the **Marketplace** button; clicking on this will take you to the Engine Launchers Marketplace panel.

Below the search bar is the **My Projects** section; all the projects you have created are displayed here. Below **My Projects** is the **Samples** section. Any gameplay feature samples or engine feature samples that you can purchase from the Marketplace are displayed here.

As you can see in the preceding screenshot, there are currently two projects displayed. You might have also noticed that one of the projects, **Test**, is quite dark and has 4.4 written next to it on the bottom-right corner of the thumbnail. This is because the project **Test** was created using version 4.4, and we have launched version 4.6.1. If we try to launch this project, we will get a prompt saying that this project was built with a different engine version and will be given the option to convert the project to be compatible with the version of Unreal 4 that we are currently running. Once you have converted the project, it will be compatible with your current engine version.

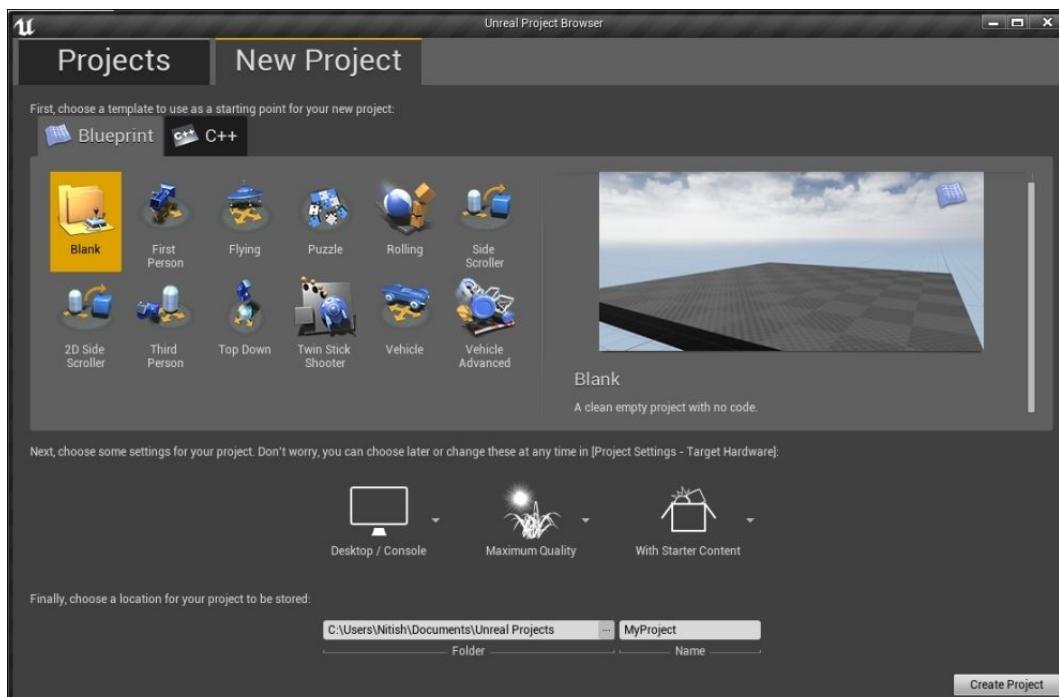
#### Note

While conversion works well for upgrading projects, downgrading a project (for example, converting a project created in version 4.6 to be compatible with version 4.4) has a few complications. For one, although you will be able to use most of the assets created in the project, you will not be able to load the map created in it and will start with a completely empty scene.

Next we have the **Samples** section. Anything you download from Engine Feature Samples, Example Game Projects, and such are displayed here. The upgrading and downgrading process we discussed earlier regarding projects applies to these project files as well.

To open a project, simply click on it and the selected project will be highlighted yellow. After selecting it, click on the **Open** button to launch it. If you have to open a project not listed in the **My Projects** section, simply click on the **Browse** button, search for the project file, and run it. Finally, at the bottom-left corner of the window is a small tick box, which reads **Always load last project on startup**. If the box is ticked, it will automatically open the last project you had opened when you next launch UE4.

Moving over to the **New Project** panel, we see the following screen:



As we can see in the preceding screenshot, there are a number of types of templates to choose from, depending on what type of game you want to make. For example, there are templates for first person games, puzzle games, side scroller games, and vehicle games. There are two types of templates you can choose from, Blueprint and C++. Blueprint projects do not require the user to have prior programming experience. All of the games features can be implemented using Blueprint, and the template also provides the basic set of blueprints required for specific game modes, such as camera, controls, physics, and so on. C++ projects, however, require the developers to know C++. These projects provide the basic framework for that particular template, upon which the developers can make the game. Picking Blueprint projects is beneficial for developers who lack programming knowledge, providing ease of development. However, though Blueprint is a great tool, it is still not as versatile as coding. With coding, you have more control over the engine, and can even modify it to suit your requirements. It also means better optimization of your game. Developers can use Blueprint to implement features, if they so require.

## Note

To create C++ projects, it is recommended that you have Visual Studio 2013 or higher installed on your system. If you do not, you will first have to download and install Visual Studio before you can create these kinds of projects. You can, however, use Visual Studio 2012 provided that you download the source code from GitHub, and then compile the entire engine in Visual Studio 2012. But, since the Engine Launcher is built with Visual Studio 2013, it is advised that you have the 2013 version. If you are on Mac, you will need Xcode 5 or higher.

To the right of the project list is a screenshot of the highlighted project, below which is a description of it.

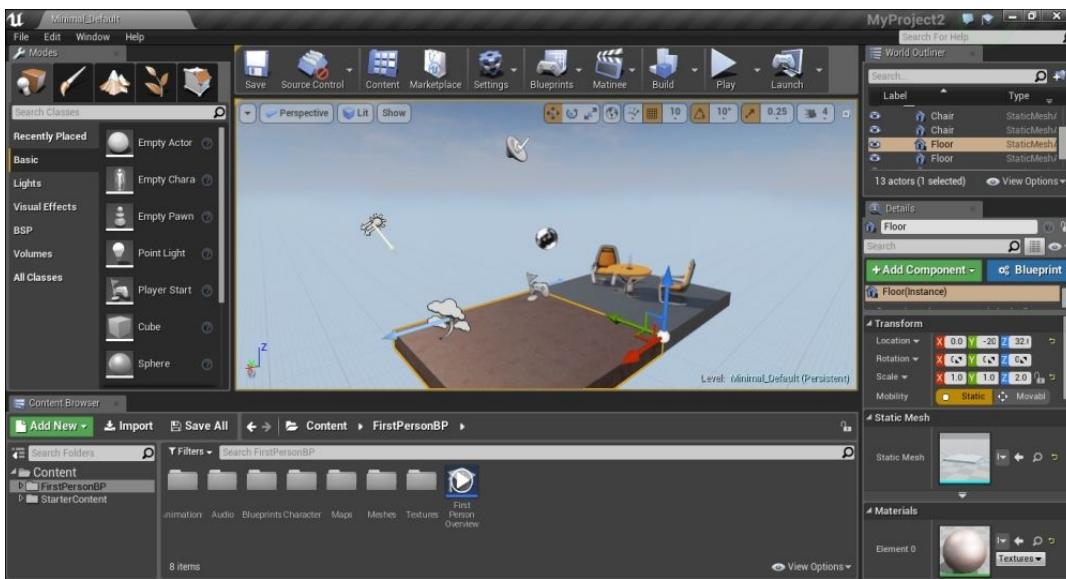
After deciding which template to use, there are a few settings you can select before opening the project. First of all, you can choose which hardware you are going to develop your game on, either PC/console, or mobile/tablet.

This will set things like controls when you launch the project. After that, you can set the quality of rendering in the game. You have the option to either pick maximum quality, or scalable. Now, although it is understandable that you would want to make your game look amazing, you do have to keep in mind that mobile devices have limitations. When developing for mobile devices, it is advisable that you pick **Scalable**. The difference between Scalable and Maximum Quality is that in Scalable, the engine config files are optimized to give the best performance. This means that some of the more costly features, such as anti-aliasing, and motion blur are, by default, turned off. Finally, you can choose whether you want to create the project with the starter content (starter content contains materials, props, audio files, textures, and so on) or not.

Once you have set these options, you can set the location where your project will be stored. The left bar shows the location and name of the folder where the project will be stored. You can set it to anything you wish. In the right bar, you can set the name of the project. Finally, after you have set all of these options, simply click on the **Create Project** button to open the Editor.

## The user interface

After you have opened or created a new project, the Editor will open. Once opened, you will see the following screen:



The Editor is where you create your game. As you can see, even though the Editors user interface is neatly categorized, there are still quite a lot of buttons, menus, and panels. To make explaining the user interface easier, we will divide the Editor into various parts, and then go over each section individually.



## The tab bar and the menu bar

At the top, we have the tab bar. Just like in an internet browser, you can see tabs of all the map files that are currently opened, and you can dock several viewports.



On the right side of the tab bar, you can see the name of the project written in light grey, **MyProject2**, in this case. Next to the projects name is the **Send Quick Feedback** button, which looks like a chat bubble. Want to give some feedback to Epic regarding the Editor, positive or negative? You can do so with this button. Simply click on the icon to open a menu from which you can choose either to send positive or negative feedback, as well as ask questions of Epic. After you have made your choice, a window opens, in which you can select what the feedback is about, followed by your thoughts. After you have written your feedback, click on **Send** and it will be sent to Epic.

Next, we have the **Show Available Tutorials** button, which, when clicked, opens a window, from which you can select what tutorial you would like to see. There are tutorials available inside the Editor itself. When you click on the button, a new window opens up, showing you all of the tutorials available.

Below the tab bar, is the menu bar. It offers all of the general commands and tools offered by all applications.

- **File** : Here, you can create, open, and save levels/maps and you can also create or open projects from here (when you create or open a new project, the current project closes and the Editor reopens). You also package your game from here. To do so, simply click on **File**, hover over **Package Product**, which will open another menu, choose which platform to package your game on, and then follow the instructions to complete the process. There are various settings and build configurations that you can set here, which we will cover in later chapters.
- **Edit** : From the **Edit** menu, you can do things like undoing or redoing the last action, cut, copy, paste, and/or duplicate whatever object (or group of objects) you have selected. You can also access the Editor preferences from here. To do so, simply click on **Editor Preferences** in the **Edit** menu. Doing so will open a new window, where you can set options such as toggling autosaves on/off or setting the frequency of autosaves, changing or assigning hotkeys, and changing the measuring units (centimeters, meters, or kilometers). There are more settings available in preferences, so feel free to explore, and tweak them to suit your game.

Finally, you can also set and change settings of the currently open project. Just below **Editor Preferences** is the **Project Settings** option; clicking on this opens the projects settings window. In the **Project Settings** window, you can set things like the projects description (this includes adding a thumbnail for your project, adding a description, and a project ID), how the game will be packaged, and what platforms will the project support.

### Note

There are a lot of settings that you can change and tweak to suit your requirements both in the **Editor Preferences** and **Project Settings**, so it is advised that you go through them thoroughly.

- **Window** : The Editor window is fully customizable. Apart from the tab and menu bar, all the other windows can be customized in the Editor to suit your preferences. The screenshot in the preceding section is the default layout of the Editor. You can add, remove, and move around any window you like.

To do so, simply follow these steps:

1. Move your cursor to the tab of the window you wish to move
2. Hold down the left-mouse button on the tab
3. Move the window to wherever you wish to move it
4. Release the left-mouse button and the window will be set

Sometimes you may not be able to find the windows tab. For example, in the preceding screenshot, the Viewport does not have a tab. This is because its tab is hidden. To unhide it, just click on the little yellow arrow located at the top-left corner of the window.

Keeping the above in mind, the Windows menu is for just that. If you wish to add another window in your Editor, you can simply open the Windows menu, select which window to add, and click on it. When you do so, the window will open, which you can then move and set wherever you want. If you are using a dual screen, then the Window menu may come in handy, since you have space to add more windows.

- **Help** : Epic has tried their best to ensure that the tutorials provided by them and the community are easily accessible at all times, either on their website, the Engine Launcher, or the Editor itself. The Help menu is similar to the **Learn** section of the Engine Launcher (described in [Chapter 1, Getting Started with Unreal 4](#)); it has links to all of the tutorials and documentation regarding UE4. On the far right corner of the menu bar is the **Search for Help** bar. Do you need to find a specific tutorial, without having to go through the entire Help section? In that case, simply type in the name of the topic you wish to find tutorials on and it will show you matching results.

### The toolbar

Next up, we have the toolbar, located directly below the tab and menu bar.



It provides quick access to the most commonly used commands and operations.

- **Save** : On the very left, is the **Save** button. Any developer knows how important this function is. One crash, and all your work goes down the drain; hence, it is available on the toolbar, so that you have quick access to it.
- **Source Control** : From here, you can either enable or disable source control, which is, by default, disabled. Source control is an important tool when working in a team. It is a method of keeping track of any changes made to a file and controlling the version of the software. When any modification has been made to the file, the team can check the modified files, and if they made any changes, post it for others.

To enable it, click on the button, which will open a dropdown menu, and select **Connect to Source**. A new window will open, asking you to choose the provider. Select the one you want and click on **Accept Settings**. Once enabled, you can check any modifications other team members have made, and post any modification you have made yourself.

- **Content** : The **Content** button opens up the Content Browser. This is similar to the Content Browser in Unreal Engine 3. So, if you are familiar with it, you should have no problem with the new version. For those who have not used Unreal Engine 3, the Content Browser is where all of the assets, code, levels, and everything can be found.
- **Marketplace** : You've suddenly realized that you require an asset or assets for your game; instead of opening the Engine Launcher again, simply click on this button to go to the **Marketplace** section of the Engine Launcher, where you can browse and buy the required item or items.
- **Settings** : This is similar to the **Info** settings in Unreal Engine 3. It lists out the most commonly used settings for the Editor. Things such as toggling on/off actor snapping, allowing/disallowing selection of translucent objects, allowing/disallowing group selection, and more can be changed here. It is also worth mentioning that the Engines visual settings, such as resolution, texture rendering quality, anti-aliasing, and more can also be changed here.
- **Blueprints** : We have a whole chapter dedicated to Blueprint; therefore, it should be no surprise that it is an important and one of the most commonly used features in UE4. You can access the Blueprint Editor from here.
- **Matinee** : This is yet another important and commonly used feature offered by UE4 using which you can create cinematics and so on in Unreal Matinee. You can open Unreal Matinee from here.
- **Build** : Build is a very important function of UE4. When you build your level, the Engine precomputes lighting and visibility data and generates navigation networks, and updates geometry.
- **Play** : When you click on the **Play** button, the game starts normally in the viewport for you to test your level and to see whether everything is functioning as intended. When the game starts, the **Play** button gets replaced by the three other buttons.



- **Pause** : This button pauses the game session. When paused, you can resume, or skip a frame.
- **Stop** : This button stops the game session and takes you back to the editing mode.
- **Eject** : When you click on **Play**, you take control of a character. If you click **Eject**, you stop taking control of it and can move it around in the Viewport.

There are other options you can set for **Play**, by clicking on the downward facing arrow next to the button, which opens the **Play** menu.

- **Launch** : When you believe that your game is finished and ready to be ported, clicking on the **Launch** button will cook, package, and deploy your project into an executable application file (depending upon what platform you want to deploy your game on).

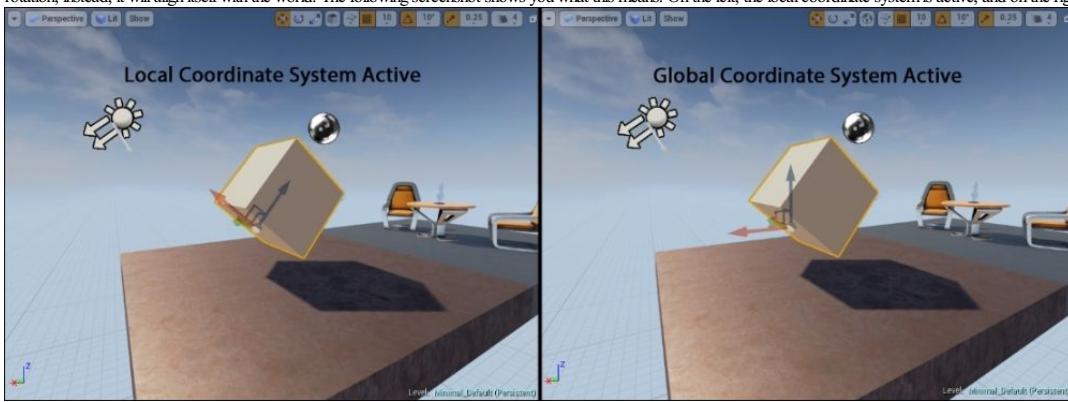
### Viewport

Located in the center of the Editor, the Viewport is where you create and view your game. All of your assets are placed and assembled here to create your world.



Let's look at the Viewport closely. To move around, hold the left or right mouse button, and use the *W*, *S*, *A*, and *D* keys to move around. To select an object, left click on it. At the top is the Viewports toolbar, some on the left, and some on the right. Lets examine these tools individually:

- Viewport Options** : At the far left of the toolbar, represented by the downward facing arrow, is the Viewport Options. When clicked, it opens a menu, which contains options for viewing the Viewport, and what you want to see in it. For example, you can switch to something called **Game Mode**, which displays the scene as it would appear in the game. This means, things such as volumes, hidden actors, and actor icons (for example, in the preceding screenshot, there are four actor icons), all get hidden. There is also something called **Immersive Mode**, which makes the Viewport go full screen. There are other options that can be set in the Viewport Options menu, so have a look around!
- Viewport Type** : Next, we have the **Viewport Type** menu. There are two types of Viewports, perspective and orthographic. The perspective view is the full 3D view, in which you can see the scene in three dimensions. Orthographic view enables you to view the world in two dimensions, either along the *XZ* plane (front), the *YZ* plane (side), or the *XY* plane (top).
- View Mode** : There are various modes in which you can choose to view your world. By clicking on the **ViewMode** button, you can check out all the various view modes offered by UE4. By default, the mode set is **Lit**. In this mode, you can see the levels rendered with all of the light actors placed in the scene. You can switch to **Unlit**, which, as the name suggests, shows the scene without any lighting. Another mode you can switch to is **Wireframe**, which only shows the wireframes of the actors placed in the scene.
- Show** : From here, you can select what types of actors you want to view or hide in your scene. When you open the menu, you can see a list of items with tick boxes. If the box is ticked, those types of actors are visible in the scene. If the box is unticked, those types of actors are hidden.
- Transform Tools** : Lets move to the right side of the toolbar. First, we have the transform tools, with three icons. There are three transform actions that can be performed on an actor. The first action is translate, which is changing the position (or coordinates) of an actor in your world. The second action is rotate, which is rotating an actor about the *x*, *y*, or *z* axis. The third action is scale, which is increasing or decreasing the size of an object. You can choose which action to perform by selecting any one of them from the Transform Tools (you can also do this in the Details window).
- Coordinate System** : The next item in the Viewport Toolbar, represented by an icon shaped as a globe, is the Coordinate System. There are two coordinate systems in which any transform action takes place: global, and local. You can click on the button to switch between them. If the global coordinate system is active, the icon will be of a globe. When the local coordinate system is selected, the icon will be a cube. When the local coordinate system is active, the axes about which you perform a transform action will align itself to the actors rotation. When the global coordinate system is active, it will not align itself with the actors rotation; instead, it will align itself with the world. The following screenshot shows you what this means. On the left, the local coordinate system is active, and on the right the global coordinate system is active.

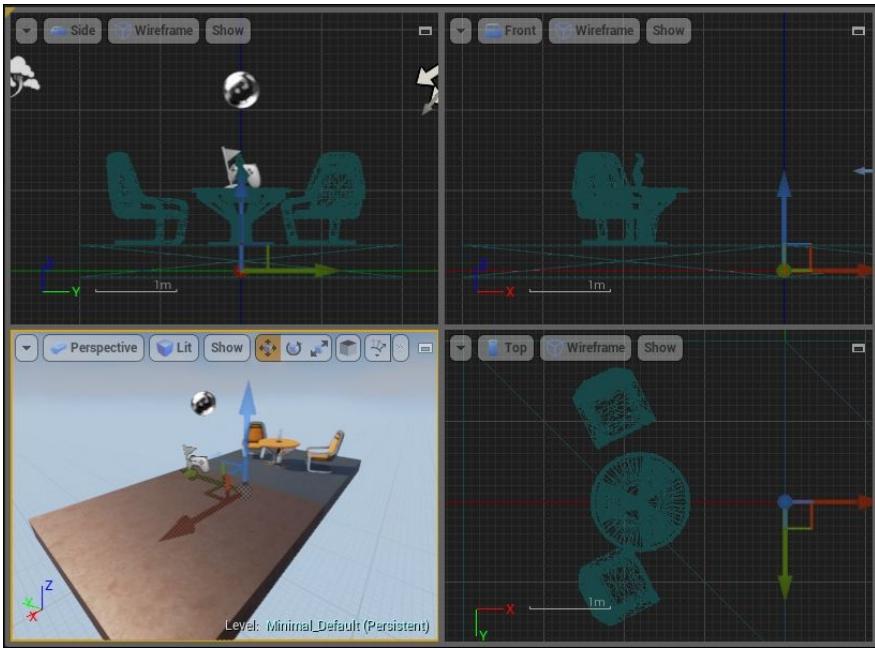


- Surface Snapping, Grid Snapping, and Grid Snap Value** : The next three tools all relate to the translate action; hence, they are grouped together. Surface snapping tool, represented by an icon in the shape of a curved line with an arrow perpendicular to it, when active, causes the actor to snap to surfaces (BSPs, other Actors surfaces, and so on) when translated. This is handy when you want to place actors on the ground. Just make sure that the actors pivot is at the bottom, since its pivot point that snaps onto the surfaces.

Grid Snapping, represented by a grid-shaped icon, when active, causes the actor to move in specific values, when translated. Think of the world as a grid, with each cell in that grid a certain size. When active, if you translate an actor, it will snap to this grid. This is especially handy in level design, when you want precise placement of actors with everything properly spaced or aligned. The value by which these actors will move can be set in the Grid Snap Value menu.

- Rotation Grid Snapping and Rotation Grid Snap Value** : The next two tools are similar to Grid Snapping, the only difference being that these tools are for the rotation action. When active, the actor will rotate in set values (for example, 10 degrees). This value can be changed in the Rotation Grid Snap Value menu.
- Scale Grid Snapping and Scale Grid Snap Value** : The final member in the grid snap group is the Scale Grid Snapping. This applies when you wish to scale objects. When active, the actor will scale up or scale down in specific increments, which can be set in the Scale Grid Snap Value menu.
- Camera Speed** : Moving on from Scale Grid Snapping, we have the Camera Speed. You can move the camera around using the arrow keys. You can set how fast the camera will move, by setting its speed in the Camera Speed menu.
- Maximize or Restore Viewport** : The last item in the Viewport toolbar, at the far right corner, is the Maximize or Restore Viewport button. As mentioned previously, there are four Viewport types that you can switch to: perspective, front, side, and top. When clicked, the Viewport is divided into 4 segments, with each Viewport type in each segment. The following screenshot shows what this looks like:

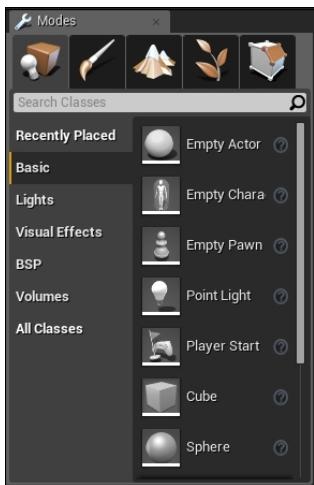
\



At the top left, is the Side view; at the top right, is the Front view; at the bottom left, is the Perspective view; and at the bottom right, is the Top view. Each window has its own Viewport toolbar. You can maximize any viewport type, by clicking on the Maximize or Restore Viewport button in the Viewport you wish to maximize. This viewport setting is most common when designing levels, and placing assets, since you want to make sure they are properly aligned from all directions. So, be ready to switch between those settings frequently.

## Modes

The Modes window contains various modes present in the Editor. Based on what task you wish to perform in the Editor, you can choose which mode to switch to from this panel.



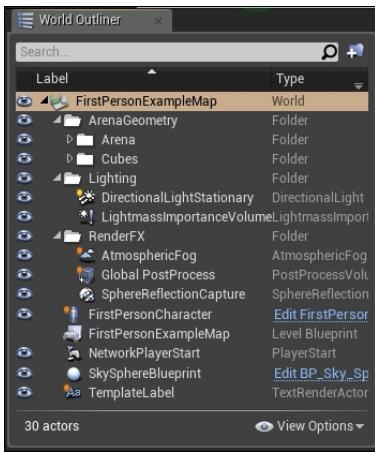
There are five modes, represented by five different icons that you can switch to. These are:

- **PlaceMode** : This is the default mode. It is used for placing actors onto your level. An actor is anything you place in your game; this includes things like static meshes, lights, triggers, volumes, and so on. This is similar to *entities* or *objects*, which is used in other game engines. There are various types of actors that can be placed in the level; especially, actors common to all types of projects. All of these actors have been categorized based on their type, called classes. There are 5 types of classes. They are as follows:
  - **Basic** : This contains the very basic actors, and ones that are used in pretty much any game you make. These include triggers, camera, player start, and so on.
  - **Lights** : The lights panel contains the different types of light actors available. For instance, you have point lights, which act as a normal light bulb, emitting light equally from a point source in all directions; and directional lights, which emit lights from an infinitely far away source, like the sun, and so on.
  - **Visual** : This class contains all of the actors that affect the game's visuals, such as post processing volumes, atmospheric fog, decals, and so on.
  - **BSP** : BSP or Binary Space Partitioning, contains BSP brushes, which are the basic building blocks for creating in-game geometry. The class contains BSP brushes of different shapes, such as box, cone, spiral staircase, and more.
  - **Volumes** : The volumes class contains different volumes, each with a different property. For example, you have something called the **KillZ Volume**, which destroys any actor that enters it, including the Player actor. This is useful when you are making pitfalls, and areas where the place can fall off.
- **Paint Mode** : Paint Mode allows you to paint and adjust colors and textures onto static meshes. Here, you can set the brush size, falloff radius, strength, and more. One thing to note is that you can only paint on the actor that is currently selected to ensure that you only paint on the mesh and not anywhere else.
- **Landscape Mode** : If you have a natural outdoor environment, instead of creating the entire landscape first in a 3D modeling software and then importing it in the Engine, you can create it in the Engine itself with the help of the Landscape tool! When you enable the landscape tool, a huge green plane appears in the viewport. This shows you what the dimensions of the landscape will be, once created. You can set the dimensions and other settings in the window. When you are satisfied, just click on the **Create** button at the bottom of the modes window, and it will create the landscape plane. Once created, you can sculpt and paint the plane to create your landscape. To delete the plane, simply click on **Place Mode** to enable the place mode, select the plane, and hit **Delete**, to remove it.
- **Foliage Mode** : In this mode, you can quickly paint static meshes using paint selection (place) and erase static meshes on landscape planes and other static meshes. This is an extremely handy tool if you are placing things like trees, plants, bushes, rocks, and so on, hence the name Foliage Mode. Instead of painstakingly placing each tree, rock, and bush in your level one at a time, you can simply use this tool to place them. You can set the density of the mesh you want to place, the brush size, and what actor or actors to place in the Foliage Mode.
- **Geometry Editing Mode** : Finally, we have the Geometry Editing Mode. As mentioned previously, BSP brushes are the basic building blocks for your in-game geometry and are extremely useful. However, the BSP brushes provided to you come in specific shapes. If you require the BSP brush to be of a different shape, you can switch to the Geometry Editing Mode, and then manually customize your BSP brush.

Finally, if you want to find a specific actor, you can type in its name in the search bar at the top, and it will show you the actors which match the name you entered.

## World Outliner

The **World Outliner** displays all of the actors that are in your level in a hierarchical format. You can select and modify actors from the **Scene Outliner** window. It is a great way to keep track of which actors are in the scene. When making a relatively large level, it is a common occurrence for the developers to forget to remove some actors, that they do not need anymore, from the scene. As a result, these actors stay in the scene and take up unnecessary memory when the game is running. The World Outliner is one way to prevent this from happening.



Some of the operations you can perform in the **World Outliner** are as follows:

- **Create Folders** : You have the option to create a folder and put actors into it. For example, in the preceding screenshot, there is a folder titled `Lighting`, which contains all of the light actors that are present in the scene. This makes keeping track of your assets even easier. It also makes things look neat, tidy, and organized!
- Grouping actors into a folder is also really handy when you want to move certain actors, without disturbing their relative position from each other. For example, say you have made an indoor scene and you want to move it to a different location. Instead of moving all of the assets individually, or group selecting them, you can group all of the assets that are in the room into a folder. If you want to move the room, simply click on the folder and all of the assets in the room will be selected, and you can move all of them simultaneously, without disturbing their relative position. To select all of the objects in a folder, right click on the folder to open a menu. Then, move the cursor over to `Select`, and then click on `AllDescendents`.
- **Hide Function** : You may have noticed an icon shaped as an eye on the left of every actor/folder. This is the hide function. If you click on it, the corresponding actor will be hidden in the scene. If it is a folder, then all of the actors in that folder will be hidden.
- **Attach Actors** : You can attach two or more actors. This is another, and relatively quicker way of moving a group of actors without disturbing their relative position from each other. To do so, in the Scene Outliner window, simply select the actor, and drag it over to the actor you wish to attach it to, and then release it when you see a popup saying `Attach *actor name* to *other actor name*`. You can also attach multiple actors to another actor.

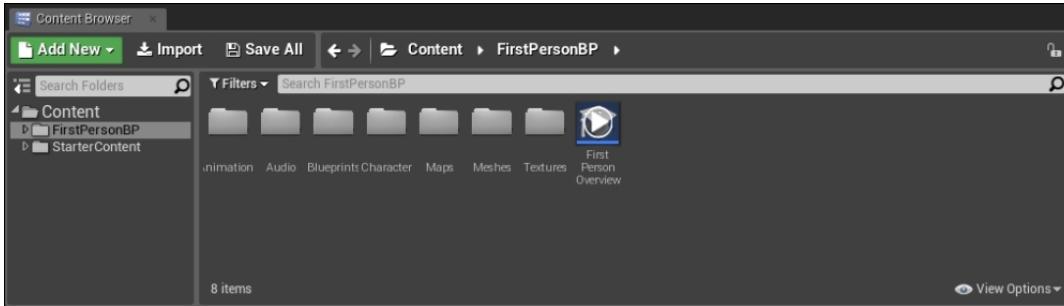
**Attaching forms – a hierarchy** : The actor to which you attach other actors becomes somewhat of a parent actor. When you move the parent actor, all of the actors attached to it move as well. However, if you move the attached actor, the parent actor will not move.

One thing to note is that to move the attached actors together, you have to select the parent actor from the Scene Outliner and then move it. If you select the parent actor from the Viewport, only it will move, and the other attached actors will not.

At the bottom left, you can see a number of actors in your scene. At the bottom-right corner, is something called **View Options**, from where you can choose what actors you want to see based on filters in the **View Options** menu.

## Content Browser

All of your game assets, such as static meshes, materials, textures, blueprints, audio files, and so on are displayed in the Content Browser. It is where you import, organize, view, and create your assets.



At the top are three icons, **Add New**, **Import**, and **Save All**. They are described as follows:

- **Add New** : Using this button, you can create a new asset, such as a material, a particle system, blueprint, and so on
- **Import** : If you want to import content into your project file, you can do so using the import function
- **Save All** : If you have created or modified an asset in the Content Browser, click on **Save All** to save all modified or created assets

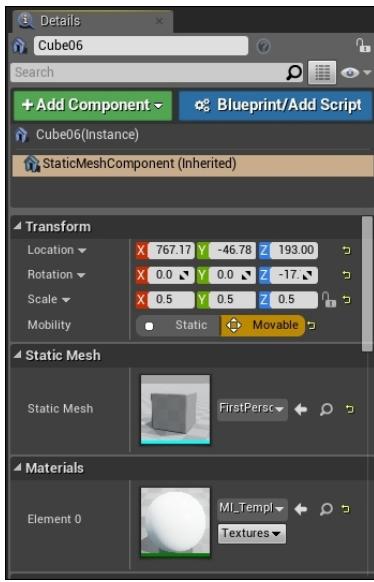
Below these icons, we have the navigation bar. If you have a lot of folders and subfolders in your Content Browser, this will come in handy to help you navigate through them quickly. At the far right corner, there is a small padlock icon, which is, by default, unlocked. If you click on it, it will lock all **Find in Content Browser** requests. In the Viewport, when you right-click on an actor, it opens up a menu. Inside it, is a function called **Find in Content Browser**. When you click on it, the Content Browser shows you where the asset is located. If locked, when you click on **Find in Content Browser**, it will not show you where that asset is located. Instead, it will open a new Content Browser window, showing you where the actor is located.

Underneath the navigation bar, on the left, is the **Sources Panel**. The **Sources Panel** contains all of the folders and collections you have in your project. On the right, is the **Asset View**; this shows all of the assets and subfolders contained within the selected folder in the Sources Panel. At the top is the **Filters** menu. If you only want to see a certain type of asset, say, if you only want to see what material assets are contained within the folder you have selected, then you can do so with the help of the **Filters** menu. To its right, is the Search Bar, which you can use to find a specific asset in the selected folder.

At the bottom of the Asset View, you can see the total number of items, including assets and folders inside the selected folder. On the bottom right, is the **View Options** menu, from where you can set how you want the items in the Asset View to be displayed. For example, you can set whether you want to see the items as tiles, as a list, or as columns.

## Details

In the **Details** panels, you can see and modify the properties of the currently selected actor.



At the top, you can see the name of the selected actor (which, in this case, is `Cube06`). This is the name box, where you can set the name of the actor to whatever you like. On the far right, is the lock button. It is by default, unlocked. When locked, the Details panel will only display the properties or details, of that actor, even if you have selected a different actor.

Below this is the search bar, which you can use to filter what properties you wish to see. Next to it, is the **Property Matrix** button which opens the Property Editor window. On the far right, is the **Display Filter** button, which you can use to do things like collapsing/expanding all of the categories, only displaying modified properties, and showing all of the advanced properties in the **Details** window.

Below the name, there are two buttons, **Add Component**, and **Blueprint/Add Script**. The Add Component, as the name suggests, allows you to add a component to the selected actor. These components include static meshes, shape primitives (cube, sphere, cylinder, and cone), light actors, and so on. This is similar to the Attach Actors function in the World Outliner. The actors get attached in a hierarchy, with the selected actor as the parent.

Apart from attaching components, you can also convert the selected actor into a Blueprint Class. A Blueprint Class is an actor which has components, as well as some code in it. (In other engines, the equivalent term would be **Prefab**). We will be covering this in great detail in the later chapters.

Finally, at the bottom, is the Properties Area, which displays all of the selected actors properties, such as location, rotation, scale, what material is currently on it, adding and removing materials, and so on, which you can modify.

## Hotkeys and controls

We will end our discussion on the Editor, by listing some controls and hotkeys for windows that you should know. Memorize them! It will make navigating through the Editor much easier and more efficient. Here are the essential controls that you should know:

Control	Action
Left-mouse button	This selects whichever actor is under the cursor
Left-mouse button + mouse drag	This moves the camera forward and backward and rotates it left and right
Right-mouse button	This selects the actor under the cursor and opens an options menu for the actor
Right-mouse button + drag	This rotates the camera in the direction you drag the mouse
Left-mouse button + right mouse button+ drag	This moves the camera up, down, left, and right, depending upon where you move your mouse
Middle-mouse button + drag	This moves the camera up, down, left, and right, depending upon where you move your mouse
Scroll up	This moves the camera forward
Scroll down	This moves the camera backward
<i>F</i>	This zooms in and focuses on the selected actor
Arrow keys	This moves the camera forward, backward, left, and right
<i>W</i>	This selects the translate tool
<i>E</i>	This selects the rotation tool
<i>R</i>	This selects the scale tool
<i>W</i> + any mouse button	This moves the camera forward
<i>S</i> + any mouse button	This moves the camera backward
<i>A</i> + any mouse button	This moves the camera left
<i>D</i> + any mouse button	This moves the camera right

<i>E</i> + any mouse button	This moves the camera up
<i>Control Q</i> + any mouse button	Action This moves the camera down
<i>Z</i> + any mouse button	This increases the field of view (goes back to the default value when the buttons are released)
<i>C</i> + any mouse button	This decreases the field of view (goes back to default value when the buttons are released)
<i>Ctrl + S</i>	This saves the scene
<i>Ctrl + N</i>	This creates a new scene
<i>Ctrl + O</i>	This opens a saved scene
<i>Ctrl + Alt + S</i>	This lets you save a scene in a different format
<i>Alt + left-mouse button+ drag</i>	This creates a duplicate of the selected actor
<i>Alt + left-mouse button + drag</i>	This rotates the camera in a complete 360 degrees
<i>Alt + right-mouse button + drag</i>	This moves the camera forward or backward
<i>Alt + P</i>	This lets you enter into play mode
<i>Esc</i> (while playing)	This escapes the play mode
<i>F11</i>	This switches to immersive mode
<i>G</i>	This switches to game mode

## Summary

In this chapter, we looked closely at the Project Brower, Editor, its UI, how to navigate through it, and some important controls and hotkeys that you should be aware of. We have now covered everything you need to know to actually start using UE4 to make games. So, without further ado, let's start making our game.

In the next chapter, we will take you through what a project is, the different types of projects you can create, how to use BSP brushes, importing and implementing assets in the game, lighting, and so on.

## Chapter 3. Building the Game – First Steps

By now, you hopefully have everything set up. You now know enough for us to finally get started with what you are really here for, to make games using UE4. A great feature about UE4 is that it is easy to get into, yet hard to master, since there is so much you can do with this powerful engine. We will start by making the core elements of the game, namely the level, lighting, and materials.

In this chapter, we will cover the following topics:

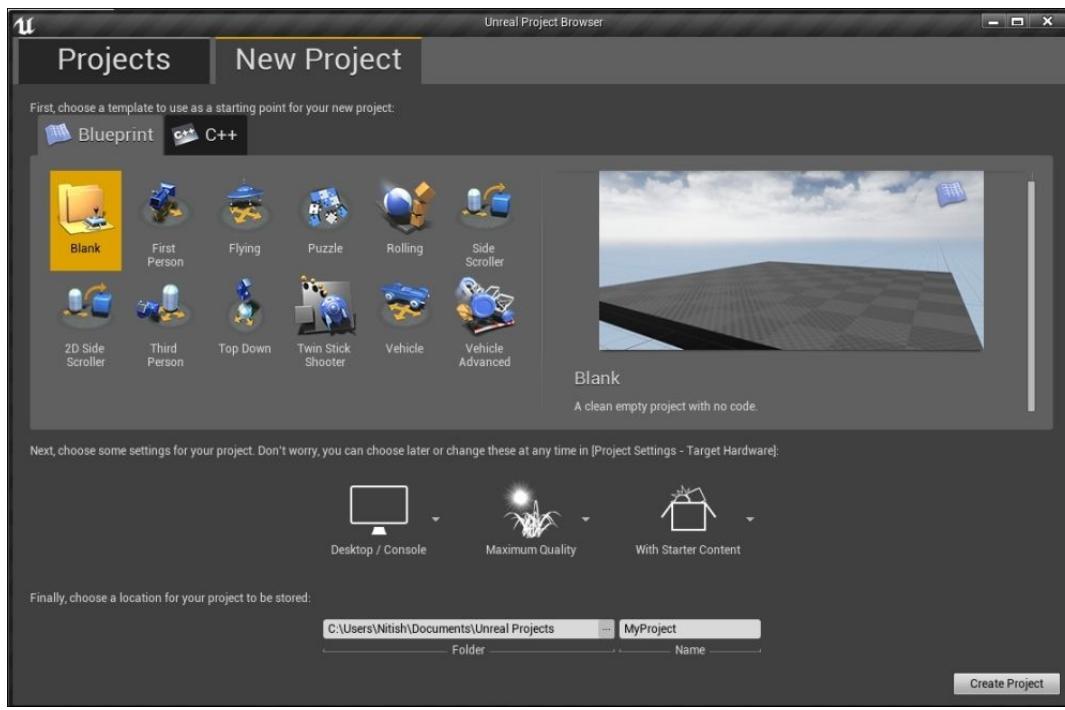
- What a project is, different types of projects offered by UE4, loading, and creating new projects
- Our game's concept, objective, genre, and features
- Geometry and BSP brushes
- Importing assets into the Content Browser, and onto the level
- How to create materials
- Lighting, its types, implementation, and building lights

## Projects

A project is an entity that holds all of the assets, maps, and code that make up your game. Once created, you can create multiple levels, or scenes, within that project. You can create, and purchase your own project files and use them. Mostly, projects that you can purchase come as a theme with assets and levels made according to that theme. For instance, you can download the **Sci-Fi Hallway** project for free from the Marketplace. This project file contains various objects, materials, and an example level setup of a futuristic hallway.

### Creating a new project

When creating a new project, UE4 offers a number of templates that you can choose from, depending upon what type of game you wish to make. Let's look at **Unreal Project Brower** again to better understand what this means:



In the preceding screenshot, we see there are various types of templates available. These templates are project files that you can create, which contain the framework for the type of game you need to make. For example, if you want to make a third-person shooter or adventure game, you can choose **Third Person**, which contains things like the camera, characters, and the basic mechanics scripted. It also contains a sample map, where you can test the controls and the mechanics. To create a new project, simply highlight the type of game you wish to create, and click on **Create Project**.

## Opening an existing project

There are several ways to open a project. One way is through **My Projects**, in the **Engine Launcher Library** section. The second way is in **Unreal Project Browser**, under the **Projects** panel. The third way to load a project is in the Editor itself. To do so, simply click on the **File** in the menu bar to open the **File** menu, select **Open Project** and simply select which project you wish to open. Doing so will close the current project and reopen the Editor.

## Project directory structure

By default, any projects you create are stored in `C:\Users\*account name*\Documents\Unreal Projects`. On opening this folder, you will see something similar to the following screenshot:

Name	Date modified	Type
BlueprintOffice	04/01/2015 18:14	File folder
Effects Cave 4.0	28/10/2014 14:27	File folder
GDC2014Demo	17/01/2015 15:55	File folder
LandscapeMountains	17/01/2015 16:29	File folder
Lightroominteriorday	31/12/2014 14:10	File folder
MyProject	20/01/2015 15:36	File folder
MyProject2	18/01/2015 22:46	File folder
Reflections Subway 4.1	25/12/2014 16:06	File folder
SciFi Hallway 4.1	25/12/2014 15:56	File folder
ShooterGame	30/12/2014 15:21	File folder
SunTemple	02/01/2015 18:47	File folder
Test	17/01/2015 15:46	File folder

In the preceding screenshot, each project has its own separate folder. Each folder contains files and folders related to that project, such as the assets, maps, the project file or `.uproject` files, and so on. Have a look around, see which folder contains what and the role each of them plays. To delete any project, simply delete the folder of the project you wish to remove.

## Bloques

We have our project setup. We can start making our game. Let's first talk about what the game is.

### Concept

The game we are going to make in the guide is Bloques, which is a first person puzzle game designed for Android. The main objective of the game is to solve a series of puzzles in each room to progress to the next. The game we are going to make is going to have four rooms; with each progressive level, the puzzle gets more complex.

### Controls

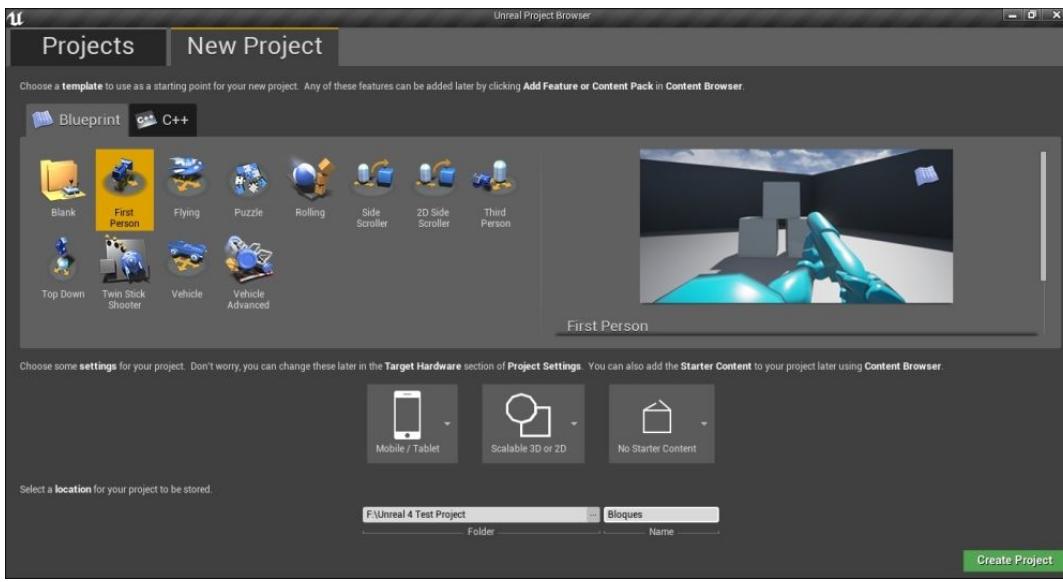
The player controls the character using two virtual joysticks, one for moving and the other for looking. All of the game's interaction, such as picking up objects, opening doors, and so on, will be done via touch.

### Creating the project for the game

The first thing we need to do is set up a project. In the Engine Launcher, launch the Editor through the **Launch** button. The version used to make this game is 4.7.6. After the Unreal Project Browser has opened, open the **New Project** panel and follow these steps:

1. Select **First Person** from the templates section, and select the template in the **Blueprint** tab.
2. In the **Target Hardware** options, pick **Mobile/Tablet**.
3. In the **Quality Settings**, you have two options, **Maximum Quality** and **Scalable 2D or 3D**. As mentioned before, you should only pick **Maximum Quality** if you are making a game for PC or Console and **Scalable 2D or 3D** for mobile or tab. Keeping that in mind, select **Scalable 2D or 3D**.
4. In the menu which asks you whether you want to start with or without starter content, select **No Starter Content**.
5. Finally, set the name of the project as **Bloques**.

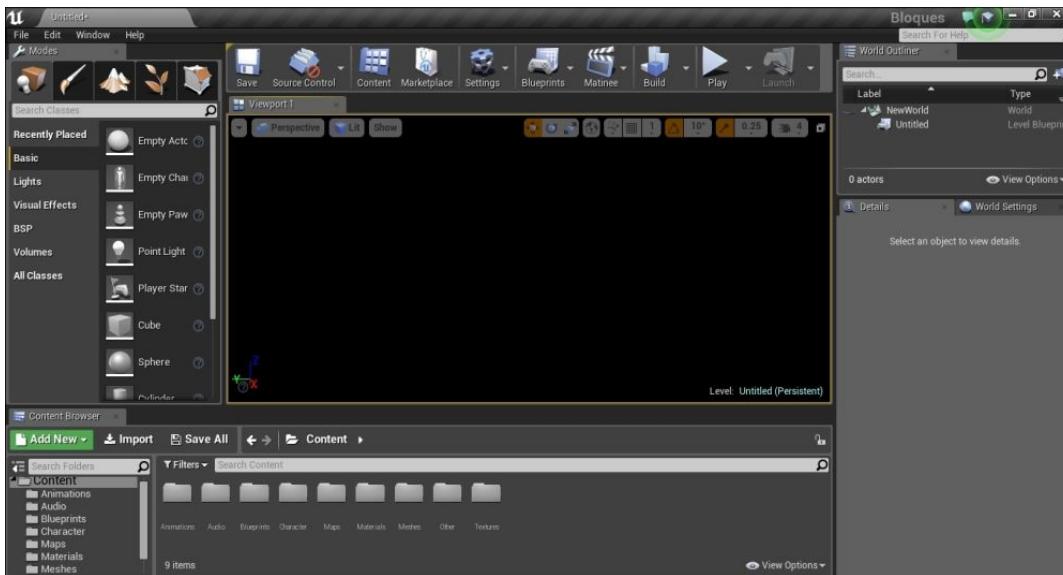
After all of these settings, it should look something like this:



Next, simply click on **Create Project**. We have now set up our project. After the Editor has opened, you will see a test level already set up. The test level is just to showcase the basic functionalities and mechanics that the template you have chosen, provides. In the **First Person** template, the player will be able to move, jump, and shoot. Another great feature is that, when you select **Mobile/Tablet** as your target hardware, UE4 automatically provides two virtual game-pads, one for moving and the other for looking. This takes a lot of work out of having to script in the controls.

However, we do not want to work on this example map. We would want to work on a new map. To do so, simply click on **File** to open the menu, and click on **New Level**. Clicking on it will open up the **New Level** window, which offers two types of levels, **Default** and **Empty**. A **Default Level** has the very basic components, such as a skybox and a player start actor already set up.

**Empty Level**, however, as the name suggests, contains absolutely nothing set up. If you wish to make your game from scratch, you should pick **Empty Level**. Since our game is going to take place indoors, we do not really need a skybox. Therefore, choose **Empty Level**. We have now set up our level where we are going to make our game. Let's save this level as **Bloques\_Game**. Your **Viewport** will now look like this:

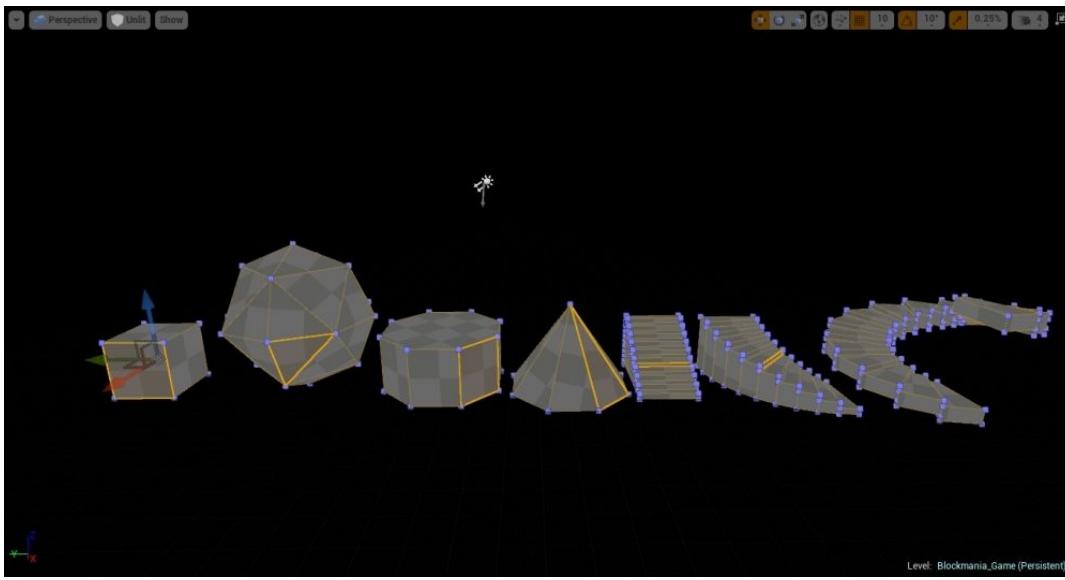


## BSP brushes

The first thing we need to do is build our level. We will do this with the help of BSP brushes. We talked briefly about BSP brushes in the second chapter, but now we will talk about them in a bit more detail. BSP brushes create volumes and surfaces for your level. It provides a quick and easy way to block out your level and to make quick prototypes. You can even create the entire level itself using BSP brushes. If you do not have access to 3D modelling software, such as Maya or 3DS Max, to create assets for your level (such as walls, ceilings, and so on), then you can use BSP brushes instead to create your level. The BSP brushes can be selected in the **Modes** panel, in the **Place Mode**.

### Default BSP brush shapes

As mentioned in the previous chapter, there are a total of seven default brushes offered by UE4. The following is a screenshot of the geometry created by brushes:



From left to right, you have:

- Box Brush**: This creates a cube-shaped brush. You can set the length, width, and height of the box. You can also set whether you want the cube to be hollow or not. If so, you have the option to set the thickness of the walls.
- Sphere Brush**: This creates a spherical brush. You can set the number of tessellations. Increasing the number of tessellations will make it smoother and more like a proper sphere. However, keep in mind that increasing the tessellations will increase the number of surfaces and therefore will require more memory to render. Keeping that and the technical limitations of mobile devices in mind, it is better to have a low-polygon geometry with a good texture, than a high-polygon geometry with a bad texture.
- Cylinder Brush**: This creates a cylindrical brush. You can set its radius and height. You can also increase or decrease the number of sides. As with the Sphere Brush, increasing the number of sides will increase the number of surfaces along the length, making it smoother, but will require more memory to render.
- Cone Brush**: This creates a conical brush. You can set properties such as the height, and the radius of the base. You can also set the number of surfaces in the brush.
- Linear Stair Brush**: This allows you to create linear or straight stairs. Instead of having to model, unwrap, and import stairs into your level, you can create it in the engine itself. You can set properties such as the length, width, and height of each step, the number of steps, and the distance below the first step.
- Curved Stair Brush**: You can also create curved stairs using the Curved Stair Brush. You can set properties such as the inner radius of the curve, the angle of the curve (the angle of curve means how much the stair will curve. You can set it to any value between 0 to 360 degrees), the number of steps, and the distance below the first step. Finally, you can also set whether you want the stairs to curve clockwise or counter-clockwise.
- Spiral Stair Brush**: Finally, we have the Spiral Stair Brush. The difference between Spiral and Curved Stairs is that Spiral Stairs can repeatedly wrap over itself, while Curved Stairs cannot. You can set things like the inner radius, the width, height, and thickness of each step, number of steps, and number of steps in one whole spiral. Finally, you can also set options such as whether you want the underside and/or the surface of the stairs to be sloped or stepped and whether you want the spiral to be clockwise or counter-clockwise.

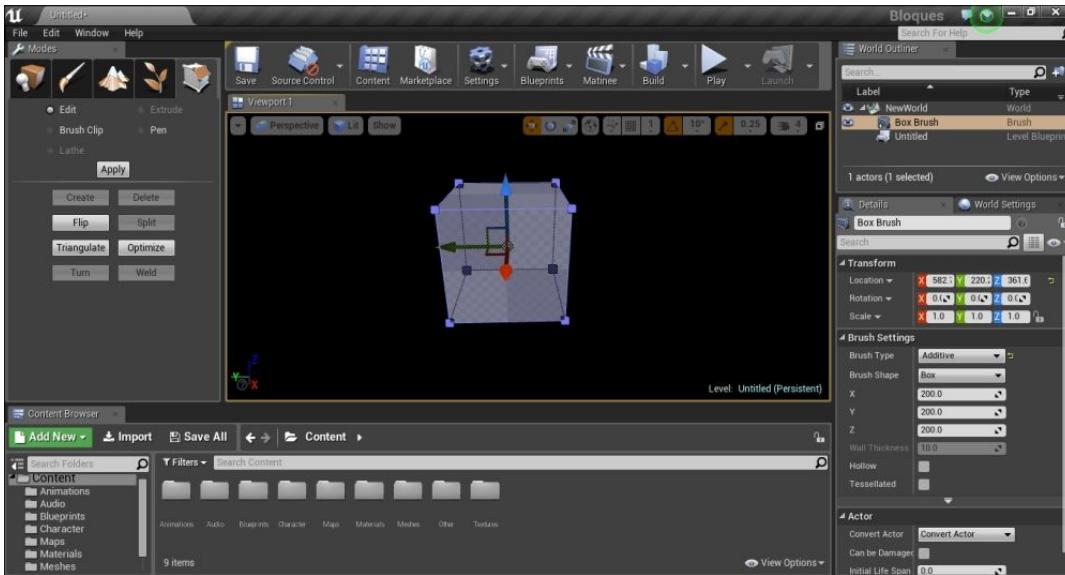
The preceding brush types can be used to create geometry through the use of additive or subtractive brush types through the **Modes** panel under **Place / BSP**. When a brush is added to your level and used to create geometry, an additive brush type will add geometry wherever placed. Subtractive brush types will remove any geometry that is overlapping additive geometry.

Apart from these settings, you can also set the properties of each surface of the geometry, such as panning, rotating, flipping, and scaling the *U* and/or *V* coordinates. You can see their effects when you apply materials to them.

Finally, you also have the option to use brushes to create volumes, such as trigger volumes, blocking volumes, pain-causing volumes, and so on.

## Editing BSP brushes

Say, you want to create geometry using BSP brushes, but the shape that you require is not one of the seven default shapes. In that case, you can create your own brush using the **Geometry Edit** mode. It is located on the far right of the **Modes** panel. Click on it to switch to the **Geometry Edit** mode.



In the preceding screenshot, you can see that when you switch to the **Geometry Edit** mode, all of the vertices, faces, and edges in the geometry. You also may have noticed that the size of the vertices have increased. In this mode, you can select either the whole brush, a face, a vertex, or an edge of the geometry.

In the modes panel, you can see several operations, such as **Create**, **Delete**, **Flip**, and so on. Some of them you can perform, while the others you cannot. What operation you can and cannot perform depends upon what you have selected (such as a vertex, edge, or face).

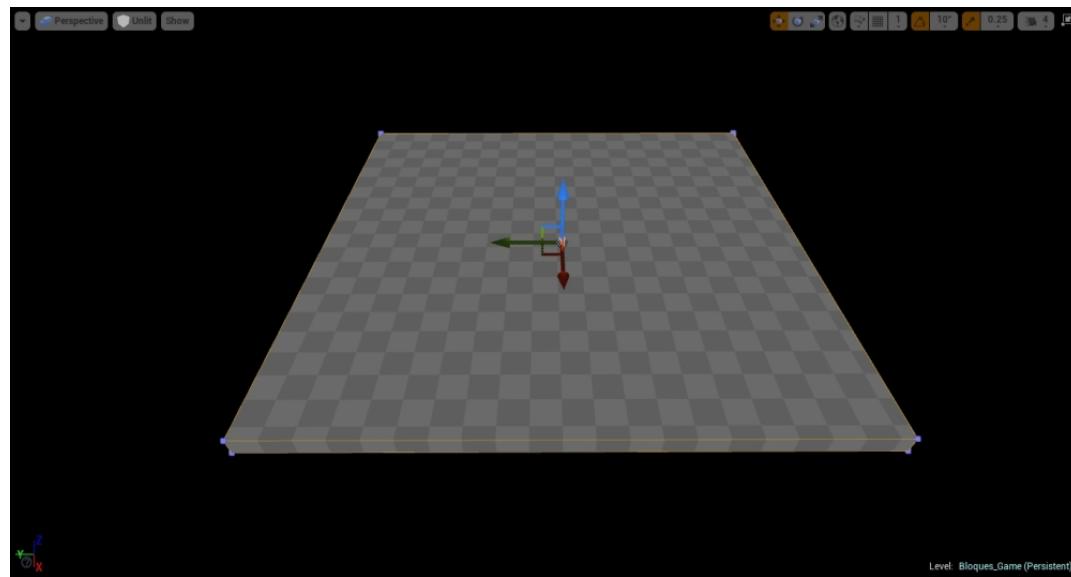
## Blocking out the rooms with BSP brushes

We will now design the environment for our game. Make four rooms and keep them blocked out so the player has to solve the puzzle in one room to get to the next.

## The first room

The first room is going to be relatively straightforward. The room is going to be cuboid. Since this is where the player starts, he/she will be introduced to the mechanics in this room, such as moving, looking, picking up and placing objects. The player simply has to pick up the key cube and place it on the pedestal to open the door. The player will also know, through this simple task, the main objective in each room.

So let's begin by making the floor. We are going to use a **Box Brush**. To add a **Box Brush**, simply click on **Box** in the **Modes** panel under **BSP** and drag it on to the **Viewport**. In the **Details** panel, set the dimensions of the brush as  $2048 \times 2048 \times 64$ . We want this room to be relatively small, since the puzzle is simple and also, to avoid unnecessary walking, as the player might get bored.

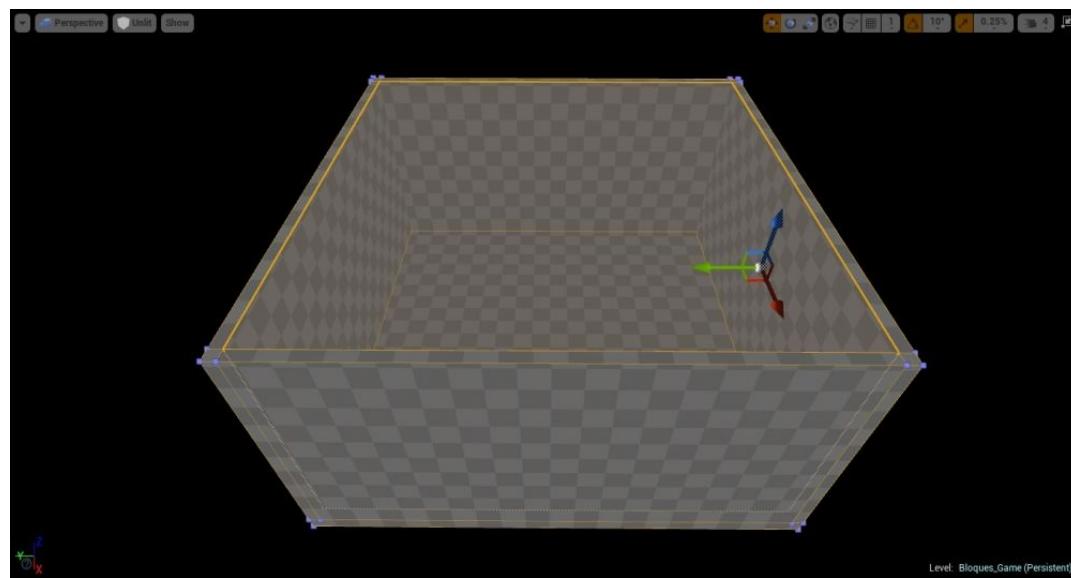


After that, let's now make the walls. Again, we will use a Box BSP brush to make them. Set the dimensions as  $2048 \times 64 \times 1024$ . After you have made one wall, simply click and hold the *Alt* key and move the wall to create a duplicate, which can be placed on the other side of the room.

### Note

When using BSP brushes, switch the View mode to **Unlit**. Otherwise, you will not be able to see the surfaces and would have to build the lighting every time you introduce a surface.

For the walls along the adjacent side, let's set the dimensions as  $64 \times 2048 \times 1024$ . Again, as with the other wall, simply duplicate the BSP brush and move it to the far side of the room.



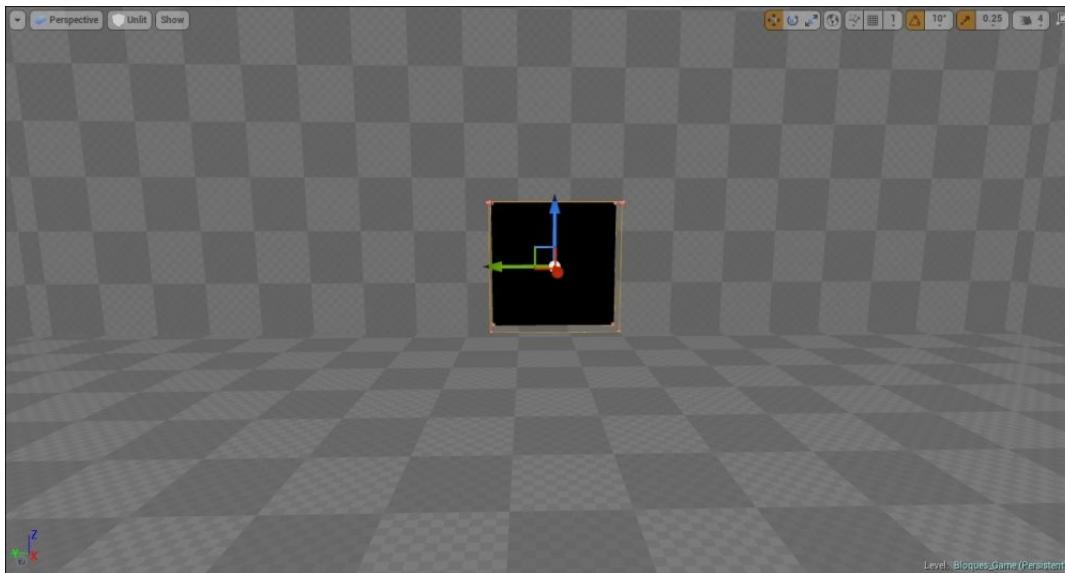
Finally, for the ceiling, simply duplicate the floor and drag it on top of the walls.

### Note

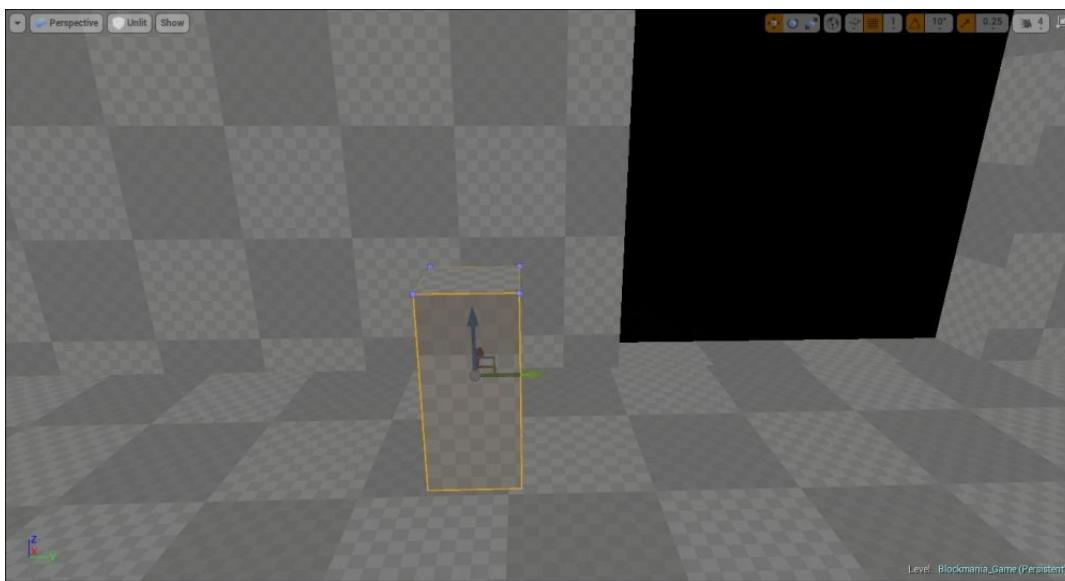
To prevent light bleeding and other complications regarding lighting and rendering, make sure that there is no gap between any of the brushes. Switch to **Top**, **Side**, or **Front** view to make sure all of the walls and ceilings are perfectly lined up.

What this room now needs is a hole for the door. Otherwise, the player would be stuck in the first room and would not be able to advance to the next. We are going to do so with the help of a subtractive BSP brush.

To create a subtractive brush, drag the **Box Brush** onto the level, and in the **Details** panel, set the **Brush Type** to **Subtractive**. Set the dimensions of this as  $64 \times 256 \times 256$ . Place this subtractive brush on any of the walls along the shorter side of the room.



Finally, let's add a pedestal near the door, where the player has to place the key cube in order to open the door.



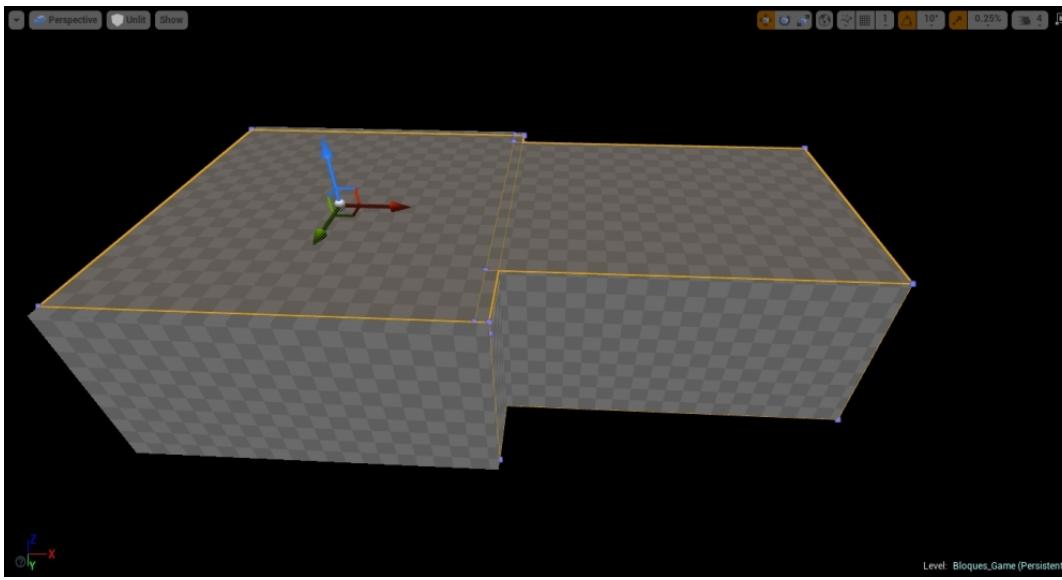
With that, we have now blocked out the first room. Let's move on to the next.

## The second room

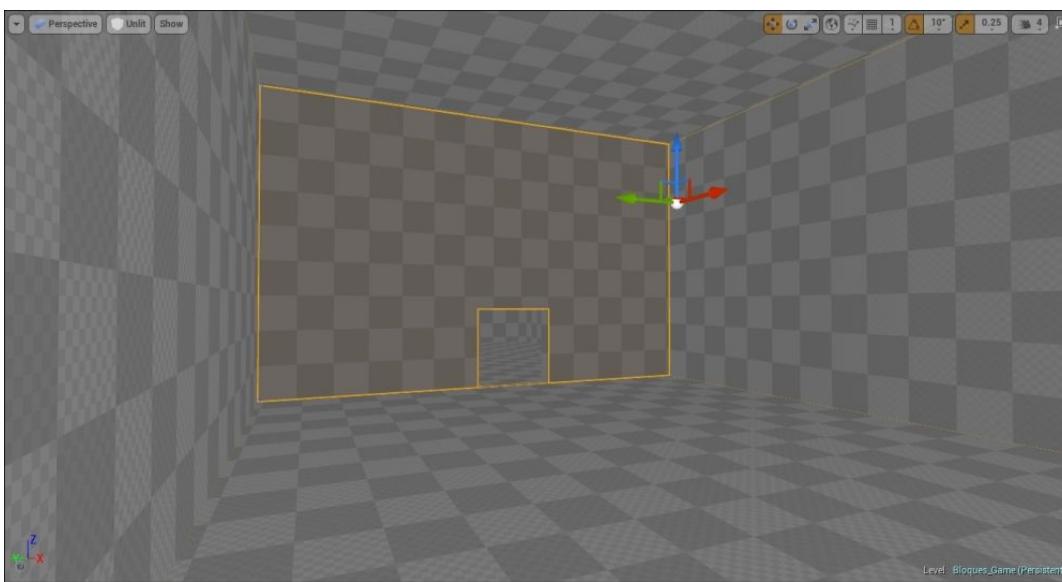
In the second room, we will add a bit of challenge for the player. Upon entering the second room, there will be a large door in the middle. The player can open the door by touching it on the screen. However, as soon as he/she lifts his/her finger or moves away from it, the door closes. To the player's right will be the key cube; however, it is trapped. In order for the player to unlock the key cube, he/she will have to go past the door, retrieve another cube, place it on a platform near the door, to unlock the key cube, place it on the pedestal, and advance to the next room.

In the first room, we had created each surface (walls, floor, and ceiling) individually. There is an alternate method we can use to create our second room. For that, we will make use of the **Hollow** property in the brush's Details panel.

With that said, select the **Box Brush**, and drag it on to the scene. Next, set its dimensions as  $2048 \times 1544 \times 1024$ , and position it right next to the first room (remember to position it next to the wall with the door). Position it as in the following screenshot:

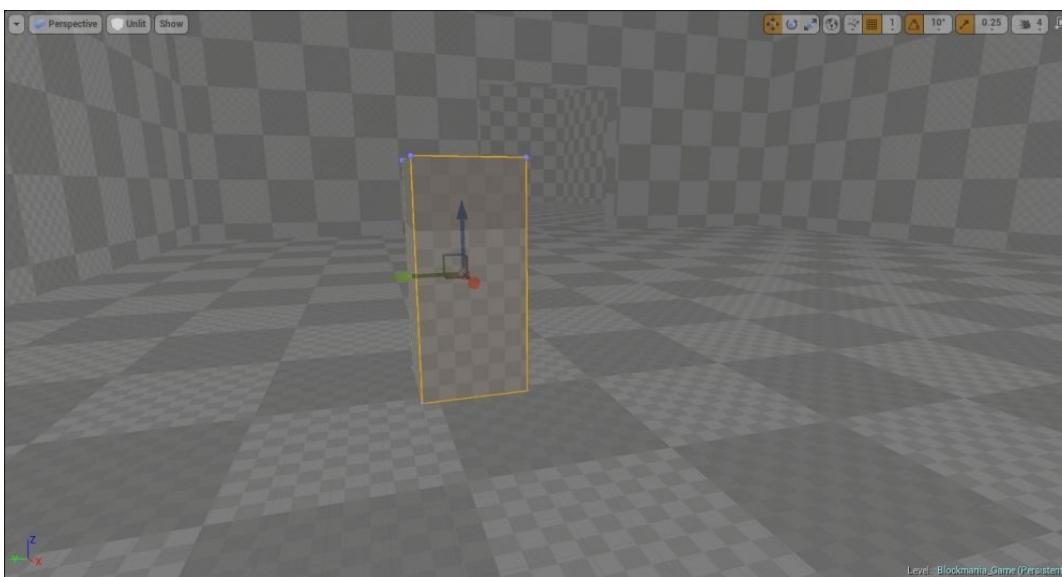


Once positioned, with the brush selected, go to the **Details** panel and tick **Hollow**. As soon as you tick it, a new setting becomes available, **Wall Thickness**. Set its value to **128** (make sure that the subtractive brush we used in the previous room for the door is overlapping both walls; otherwise, you will not be able to see the door).

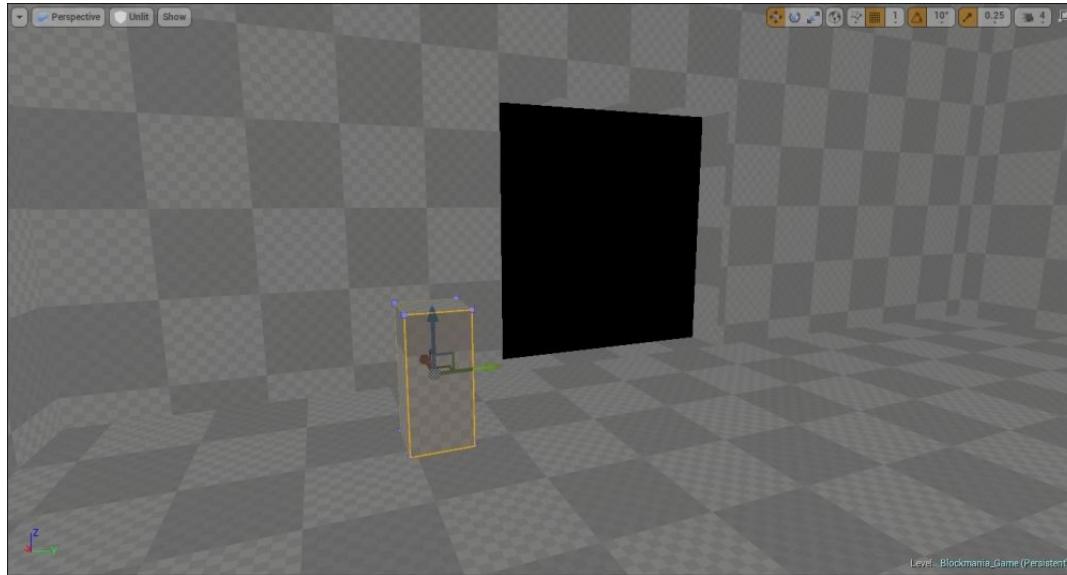


And that is it! We have our second room blocked out without having to spend loads of time placing each part of the room and making sure that they are aligned properly. The only things left to place are the pedestals and the hole for the door that leads to the third room. For the door, simply do what we did for the first room, create a subtractive brush of dimensions  $64 \times 256 \times 256$ , and position it on the other side of the room. Alternatively, duplicate the subtractive brush for the first room and then move its copy to the far side of the room.

Finally, to finish things off, we will add two pedestals in this room. One pedestal will be near the middle of the room, where the big door will be. The player will have to place the first key cube on this pedestal to unlock the second key cube.



The second pedestal goes on the other side of the room, where the player has to place the key cube to open the door to advance on to the third room.



With this, we have finished blocking out the second room. Let's now move on to the third, where things start to get interesting.

#### Note

Remember to keep saving, so that you do not lose your work should the Engine suddenly crash or any other technical issue arise. To save, just click on *Ctrl + S*, or click on the **Save** button on the **Viewport** toolbar.

### The third room

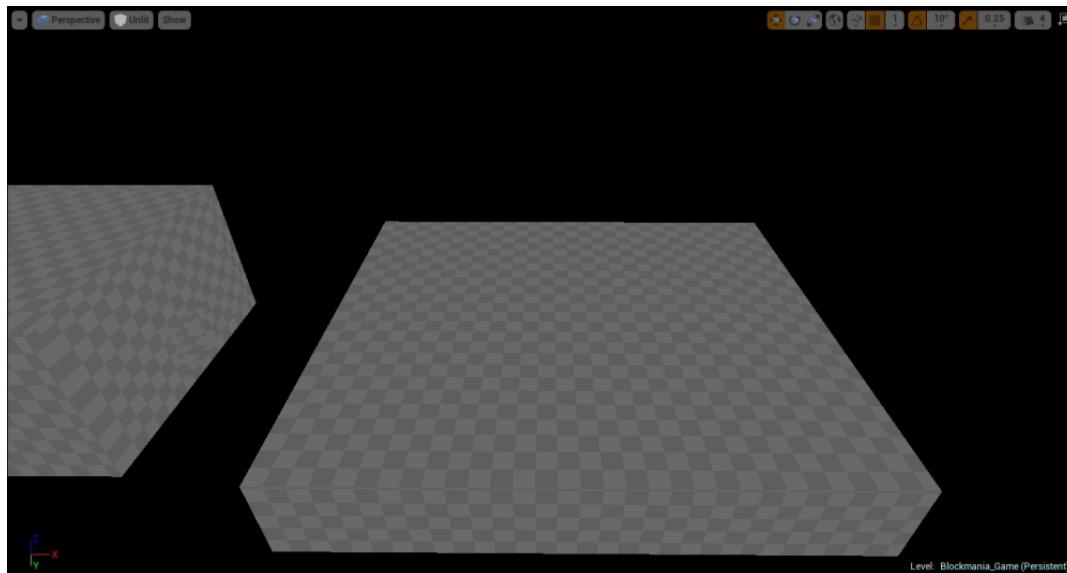
By now, the player would have understood the basic mechanics and controls in the game. Let's now give him/her a bigger challenge in the third room. Upon entering the third room, there will be a pit between the player and the door to the final room. For the player to be able to cross the pit, he/she will require a bridge.

To draw the bridge, the player will have to direct an AI controlled object on to a switch. The object will move along a path. However, parts of the bits are missing. The player can fill in the gaps with the help of switches placed in the level. Here, the challenge lies in determining which switch to press and when to press it. It also relies on proper timing. After the object has reached its destination, the bridge will be drawn, through which the player can cross, grab the key cube, open the door, and advance to the next level.

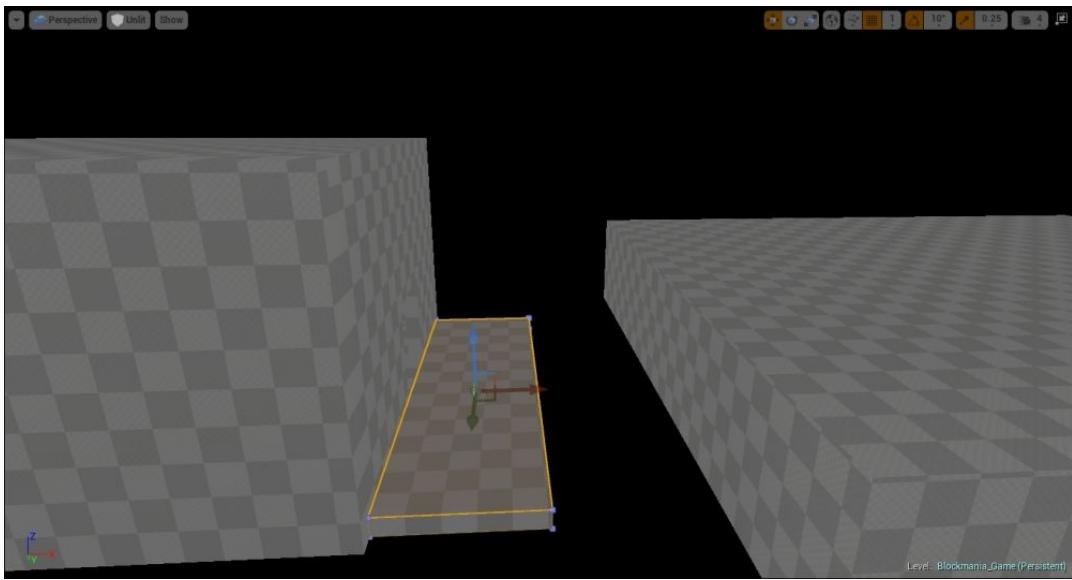
There are two ways of constructing the pit; the first way is to make the room in two parts. The first part would be on one side of the pit, and the second on the other side. After making the two parts, construct the pit, and finally place them. However, this is a bad and time-consuming way of constructing the room and the pit. We would also have extra surfaces, meaning more memory usage when rendering the level. Also, you would have to painstakingly line all of the parts up properly to ensure there are no gaps.

The second way, which we will use instead, is that we will create the whole room and with the help of a subtractive brush, carve out the pit. This way, we would not have to worry about extra faces aligning parts of the room, and we can also easily set the dimensions of the pit by moving and/or editing the subtractive brush.

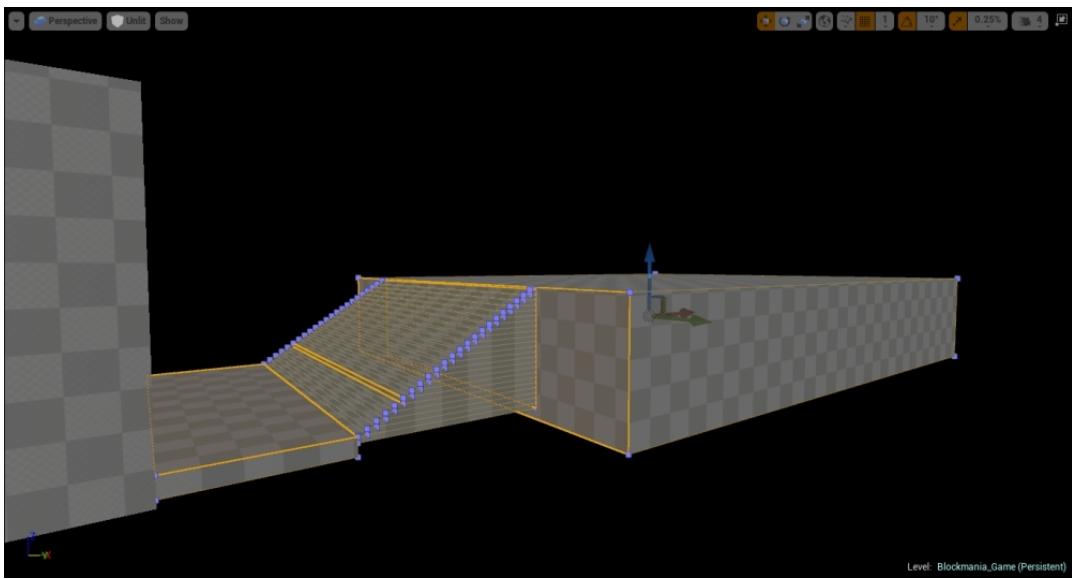
As always, let's begin by making the floor. Now, since we are going to carve out the pit, the floor will have more height than the other rooms. Select the **Box Brush**, and set its dimensions as  $4096 \times 2048 \times 512$ . This will be the main area, where the puzzle is going to be.



Since this room is higher than the previous rooms, we are going to need some stairs so that the player can reach the third room. First, add a small **Box Brush**, set its dimensions as  $512 \times 1544 \times 64$  and place it near the door of the previous room.

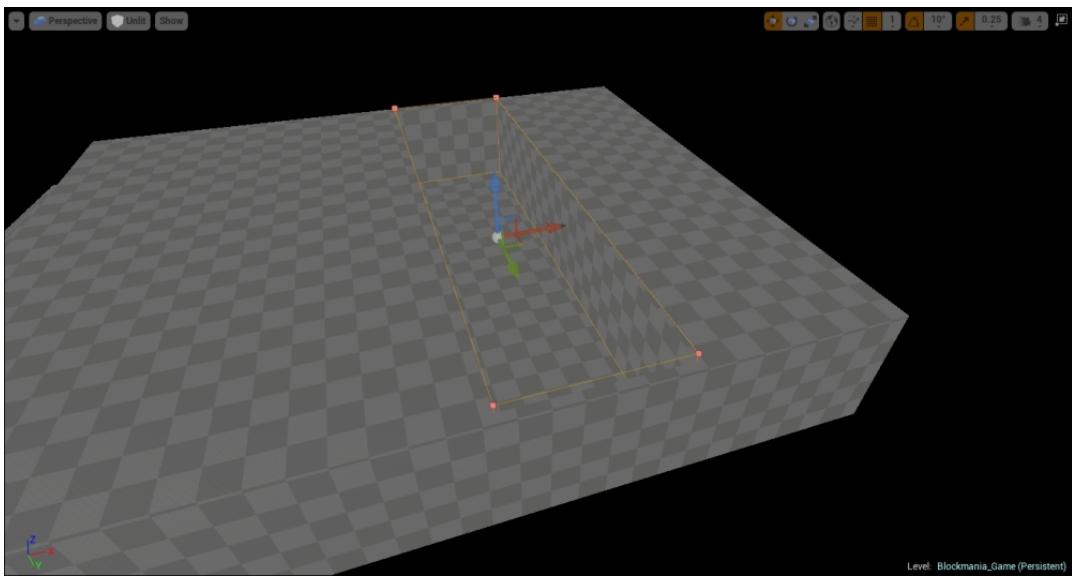


To add the stairs, drag the **Linear Stair Brush** from the **Modes** panel onto the level. Set the width of the stairs as `1544`, and the number of steps to `23`. Place the stairs at the edge of the **Box Brush** we placed earlier; finally, take the floor of the third room and put it adjacent to the stairs.

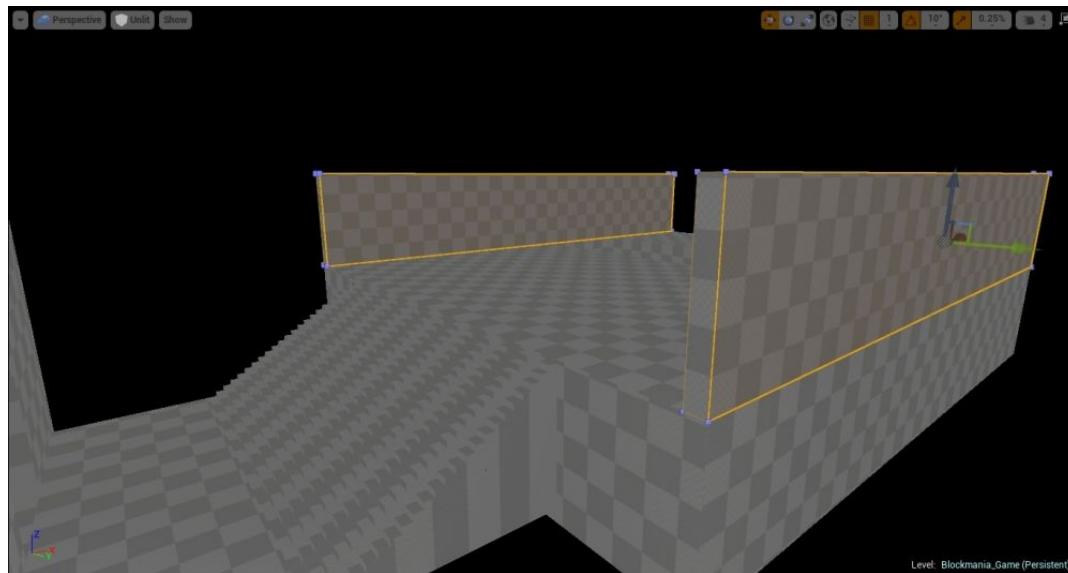


To add the pit, we are going to need a Box subtractive brush. Another way of selecting a subtractive brush is by first selecting the shape of the brush, which in this case is the **Box Brush** and then, in the **Modes** panel, at the bottom, you can set the brush type to be additive or subtractive. Simply select subtractive and drag the brush onto the level.

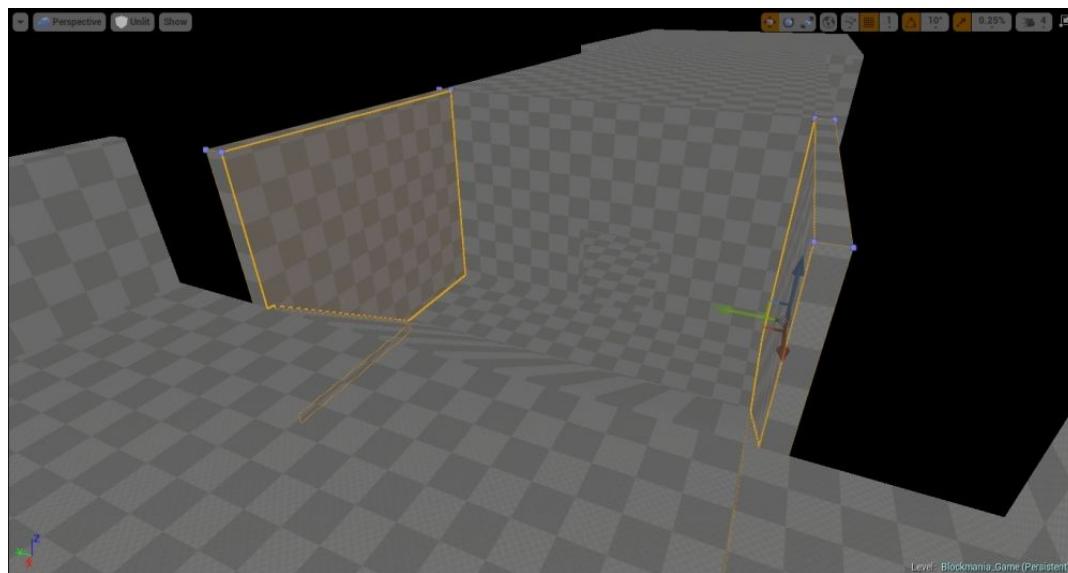
We have to make sure that the pit is wide enough so that the player cannot jump across it and deep enough that the player cannot jump out of should he/she fall into it. Keeping that in mind, set the dimensions of the brush as `640 x 2304 x 512`. Place the pit near the other end of the room.



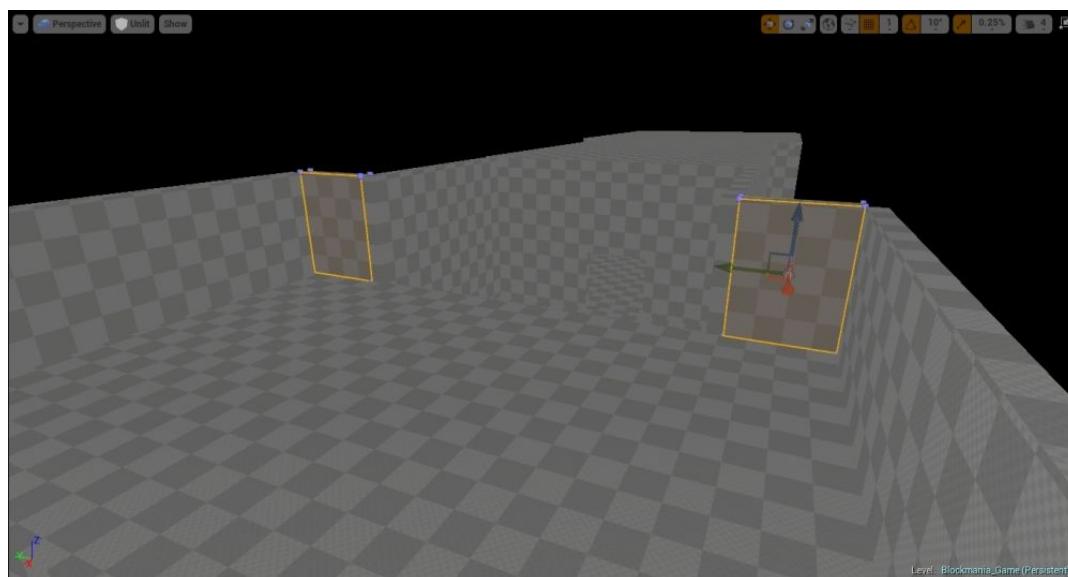
We now have to place the walls. We will add walls to this room in a few steps. First up, select a Box brush, set its dimensions as  $4096 \times 64 \times 512$ , and place it along the longer side of the room. Duplicate and place the second wall on the other side.



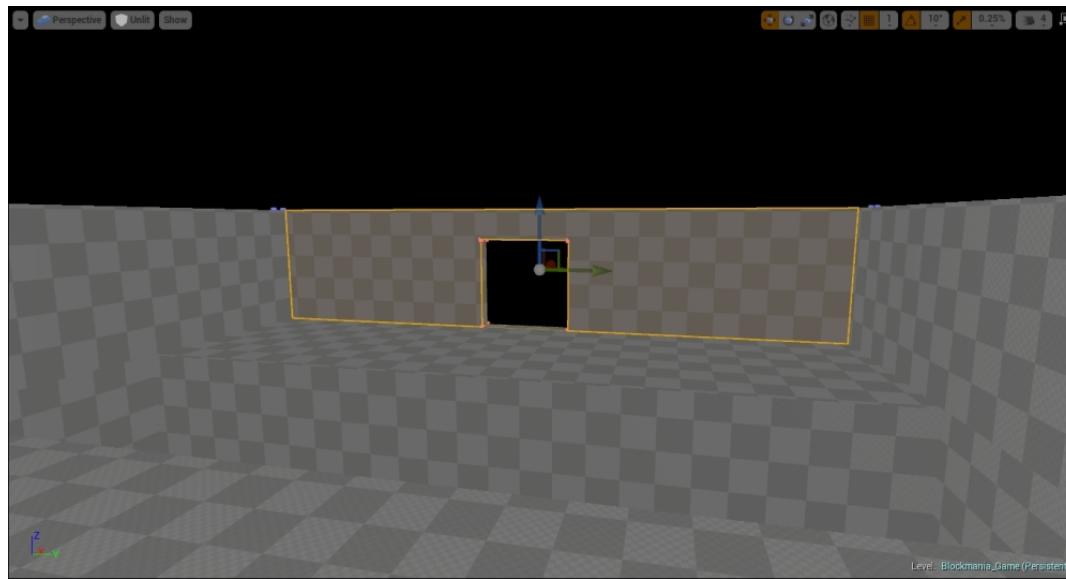
Next, we need to place some walls along the stairs and corridors that lead to the third room. For that, set the dimensions as  $1202 \times 64 \times 1024$  and place them on either side of the stairs, ensuring all of them line up properly and there is no gap anywhere between the brushes.



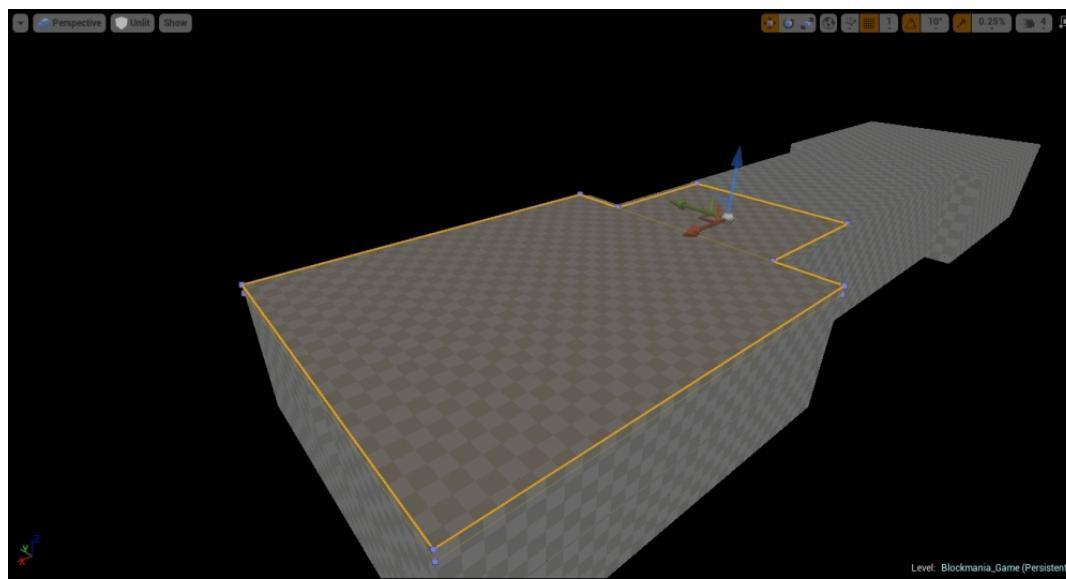
In the preceding screenshot, you may have noticed that there is a gap between the walls of the corridor and the walls along the longer side of the room. Let's fill that in with a Box brush of dimensions  $64 \times 188 \times 512$ . Place two of them, one on either side, to fill in the gaps.



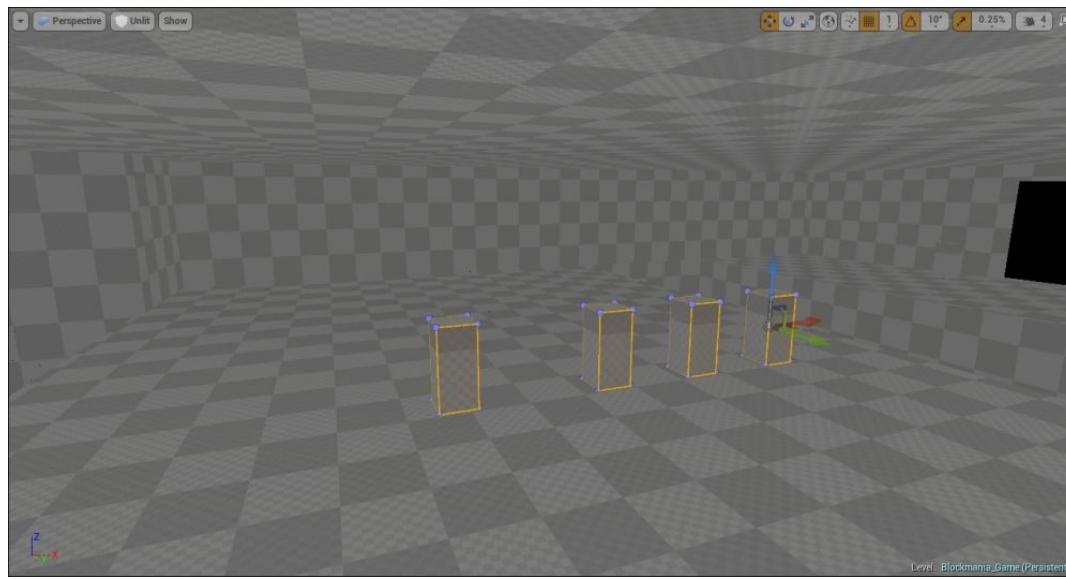
Finally, the wall that we are going to place is on the far side of the room, along the shorter side. Set the dimensions of the **Box Brush** as  $64 \times 2048 \times 512$ , and place the final wall with a subtractive brush for the door.



The ceiling for the corridor will be of the dimensions  $1202 \times 1544 \times 64$ , and that of the room will be  $4096 \times 2056 \times 64$ .



Now that we have blocked out the room, we need to add a few more things before we move on to the fourth and final room. First, we are going to place some panels on which there are switches, which the player can press to direct the object across the pit. Just duplicate the pedestals from the previous rooms and place them in this one to create the panels. Place them near any of the longer walls, and place them in such a way that the player can see the other wall.

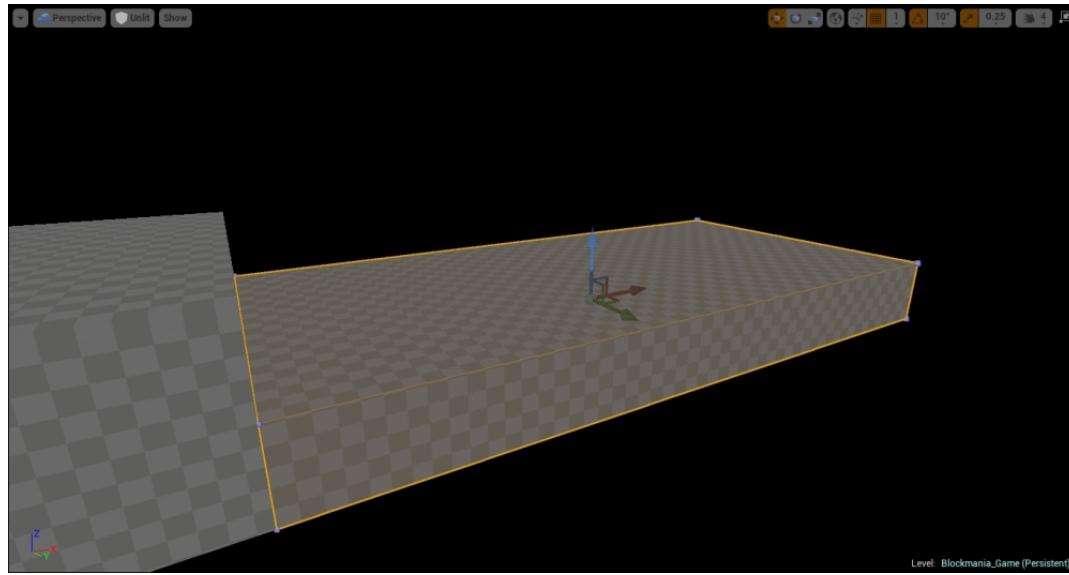


Finally, place a pedestal near the door. With this, we have finished off blocking out the third room. Now, let's block out the fourth and final room.

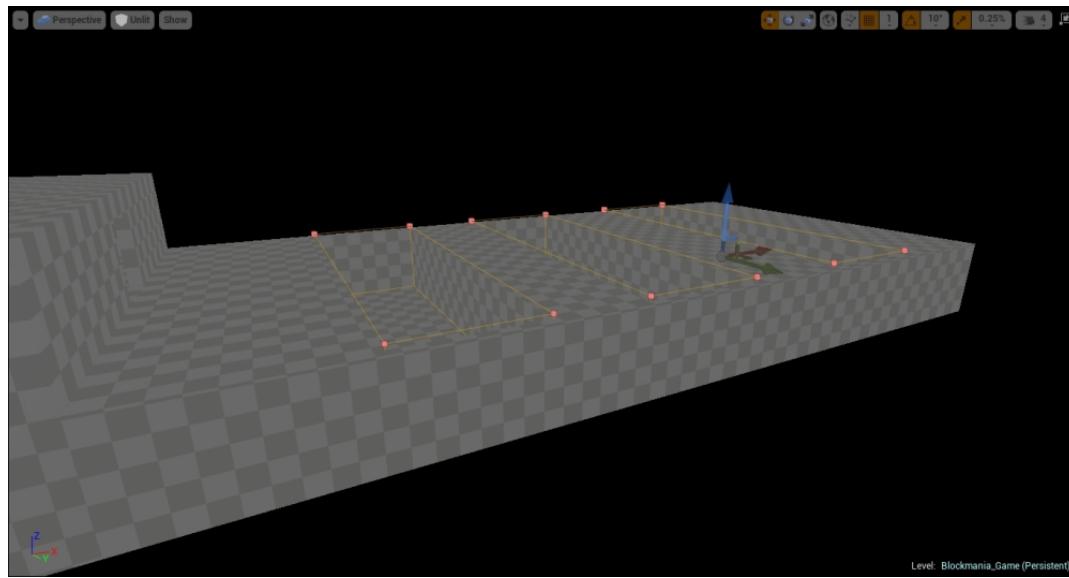
## The fourth room

We only have one more room to block out. In the previous three rooms, we had different puzzles and objectives. In the fourth room, we are going to combine all of the puzzles from the previous room. In this room, the objective of the player is similar to that in the previous room, direct an AI controlled object through obstacles such as doors and pits towards the other side of the room. The object will travel in a predefined path and keep on moving until it reaches its target. If it hits a door or falls down a pit, it resets from its starting position.

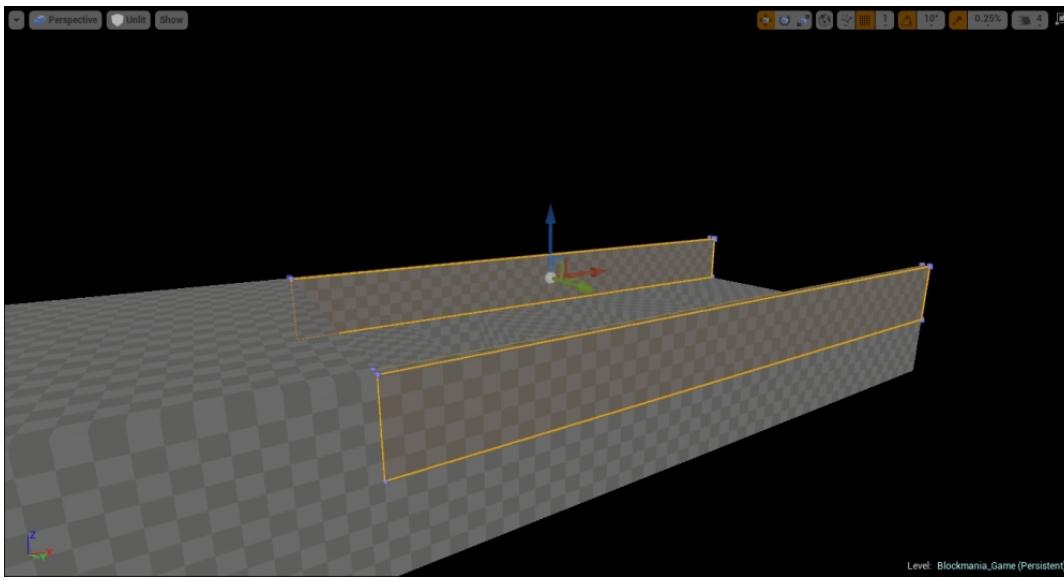
Since we are going to have pits, the height of the room will be similar to the previous ones. Also, this is going to be the biggest room. Keeping that in mind, we will make the floor by selecting a **Box Brush** with the dimensions  $5120 \times 2048 \times 512$ .



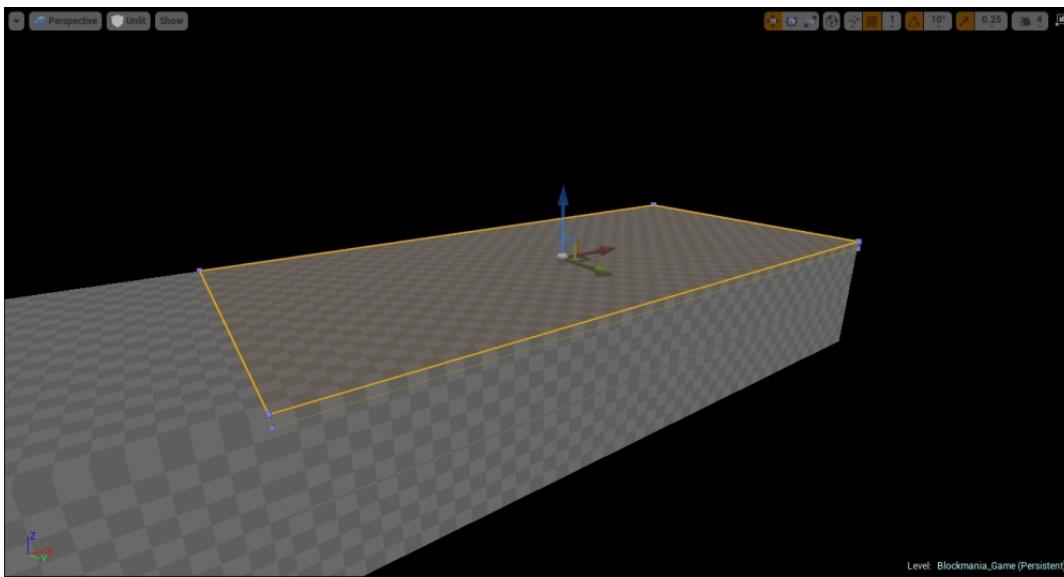
Now that we have the floor, we are going to make some pits. Again, with the subtractive mode selected, create a **Box brush** of dimensions  $728 \times 1928 \times 512$ , and place it near the door. In this room, we are going to have three pits, so duplicate the subtractive brush by holding down the *Alt* key, and create two copies and place them along the map.



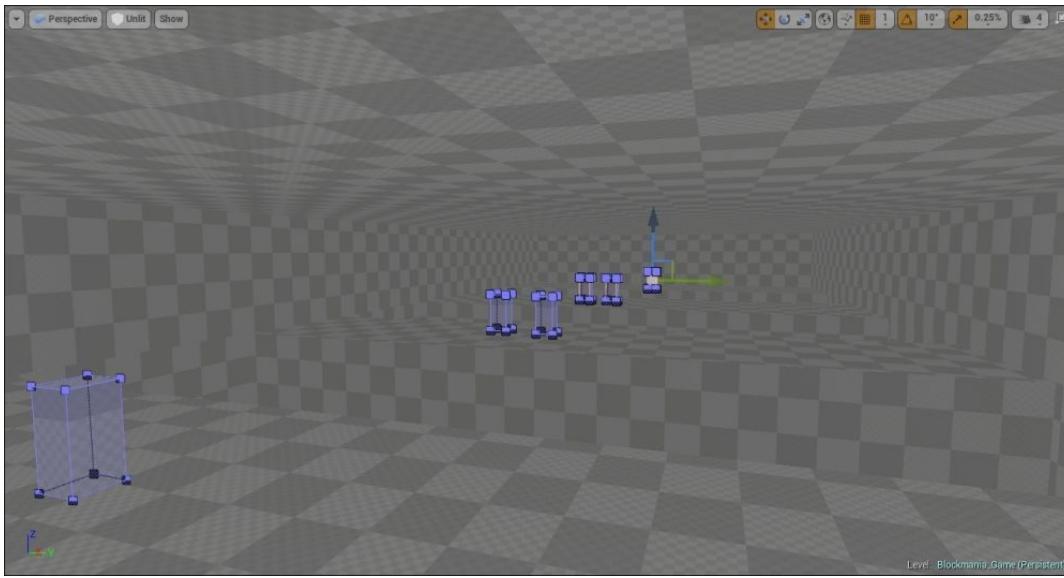
Next, comes the wall. For the longer side, simply duplicate the walls from the third room. In the **Details** panel, set the value of *X* as  $5120$ , and simply place the wall. Duplicate and place the other wall. For the shorter side, you can do the same—replicate and place the shorter side wall (the one with the door) at the other end of the room.



Next, duplicate, set the Z value of the brush to `64`, and drag it upwards to form the roof.



Finally, we are going to place a few pedestals where the switches are going to be.

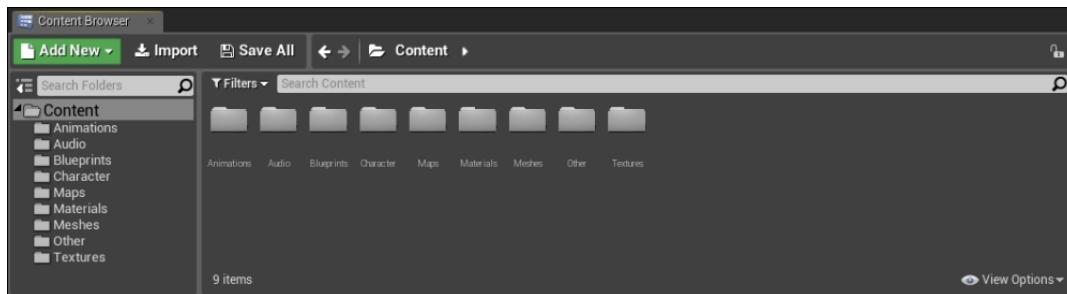


With that, we have now completed blocking out our last room!

Now that we have blocked out our rooms, let's place some assets, create materials, and apply them onto our level.

# Content Browser

We briefly talked about **Content Browser** when we were discussing the Editor's user interface in the previous chapter. Let's talk about it a bit more. The **Content Browser** is where all of your assets of a project are stored and displayed. These assets include **Meshes**, **Textures**, **Materials**, **Skeletal Meshes**, **Blueprints**, **Map Files**, **Audio files**, and so on.



In the preceding screenshot, you can see different folders, all named, based on what is contained within each of them. It is considered a good practice, and also prevents confusion later on if your project has a lot of assets, that you organize your assets based on their type.

## Migrating and importing assets

When creating a game, the main thing you need are assets, as without them your game would just be a blank map. You would want to import the assets you created for the game in your Project. For that, you need to import them first to your project file before you can use them. Currently, UE4 accepts the following assets:

- **Texture files** : These are 2D images, which can be imported as .jpg, .png, .bmp, and .tga files.
- **Static Meshes/Skeletal Meshes** : Static Meshes are 3D objects created using a 3D software, such as Maya, Max, Blender, and so on. A Skeletal Mesh is a mesh that can be animated. You can import them as .obj files, as well as .fbx files.
- **Audio files** : You can import audio files, such as music, sound effects, dialogues, and so on (mono), as a .wav file. You can also import audio files with more than one channel (stereo).
- **Script files** : You can also import script files into UE4 as .lua files.
- **IES Light Profiles** : IES Light Profiles files define the intensity of light in an arc. This is usually used to make the light appear more realistic. These too can be imported into UE4 as .ies files.
- **Cubemap Texture** : Cubemap Texture is imported as an .hdr file. These files are used to map out the environment, especially if your game has outdoor scenes. These files contain information regarding color and brightness across a range.
- **True Type Fonts** : You can also import various types of fonts for your game. Fonts are used either in huds, UI, and such, and can be imported as a .ttf file.

## Importing assets

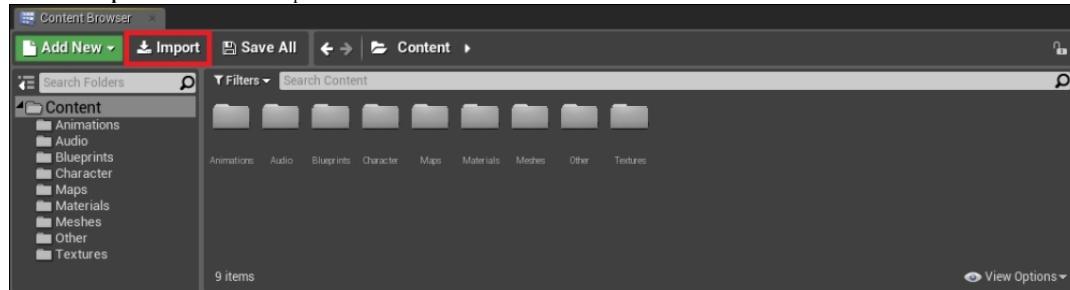
There are two ways of importing assets that you have created for your game in your project file.

### Note

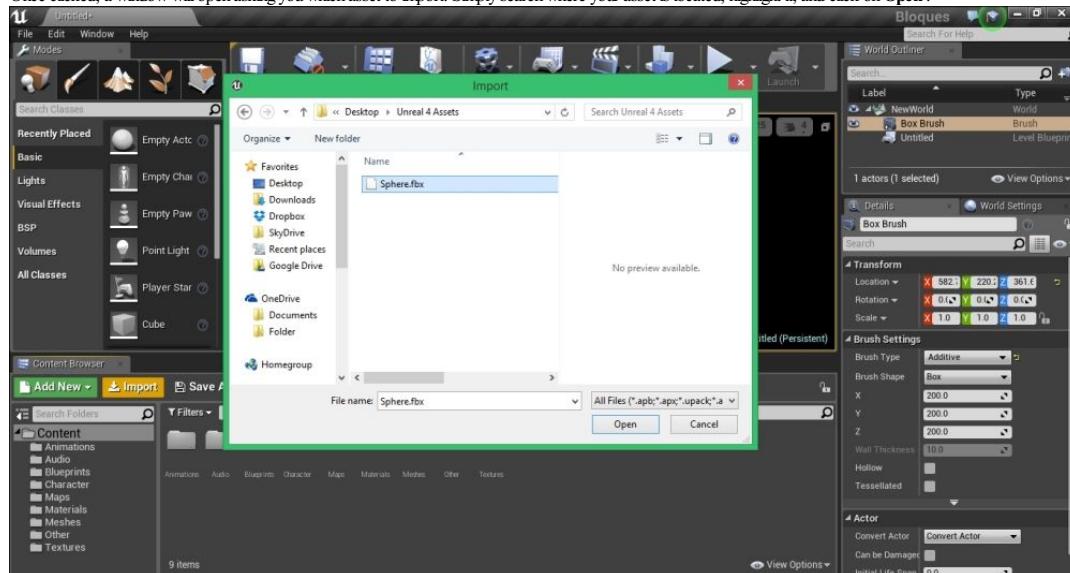
To demonstrate how to import assets, a simple sphere was created in Maya and exported as an FBX file.

The first way is through the **Content Browser**:

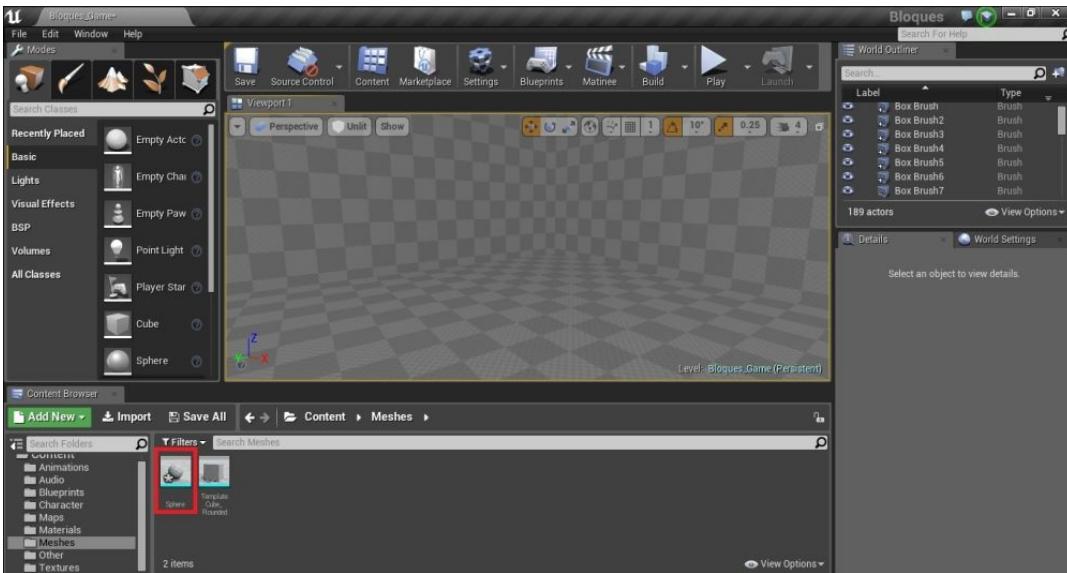
1. Click on the **Import** button located at the top of the **Content Browser**.



2. Once clicked, a window will open asking you which asset to import. Simply search where your asset is located, highlight it, and click on **Open**.



3. When you click on **Open**, the **Import Options** window will open up, which has different import options, depending upon the type of asset you have imported. Once satisfied with the import settings, click on **Import** and the asset will be imported to your project file.

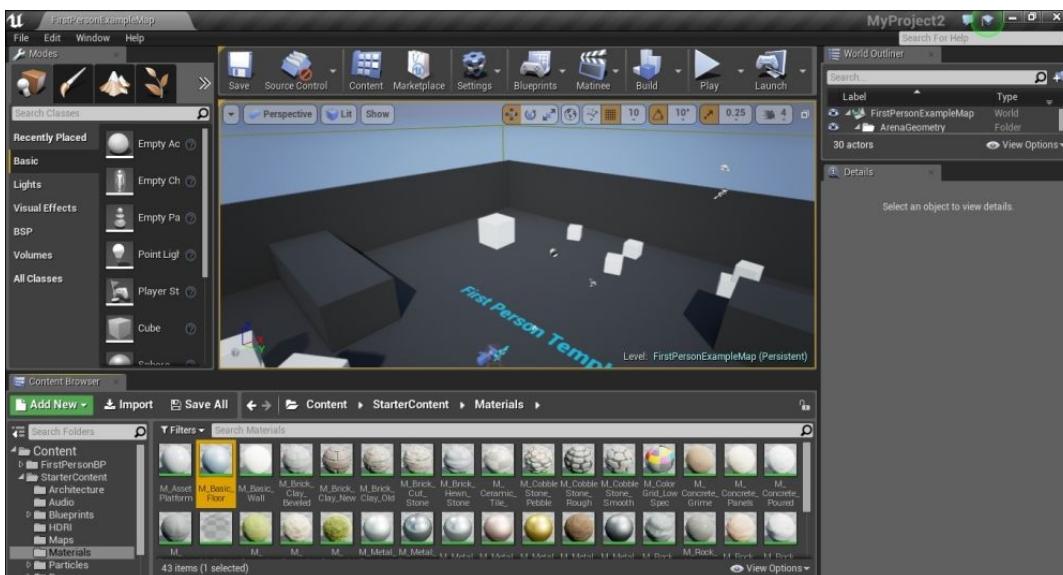


The second method to import assets to your project is by simply dragging the asset you want to import and dropping it in your **Content Browser**. To do so, simply click and drag the asset from wherever it is stored and release the left-mouse button over **Content Browser**. When you release the left-mouse button, the **Import Options** window will open, similar to the one mentioned in the first method. Again, once satisfied with the settings, click on **Import** and the asset will get imported.

### Migrating assets

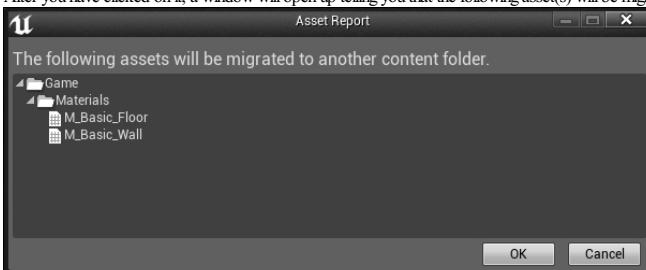
Sometimes, you may need certain assets from a different project file. In such cases, the methods mentioned previously regarding importing assets will not work, since the contents of a project are saved on your system as either a **.uasset** file or **.umap** file. Therefore, UE4 will not be able to import them. You, therefore, will have to perform the **Migrate Asset** action.

When we create our project, we select **No Starter Content** since we do not require all of them. It would eat up unnecessary space. We do, however, require a few assets, particularly wall and floor materials. To demonstrate this, a new project with all of the starter content has been set up.



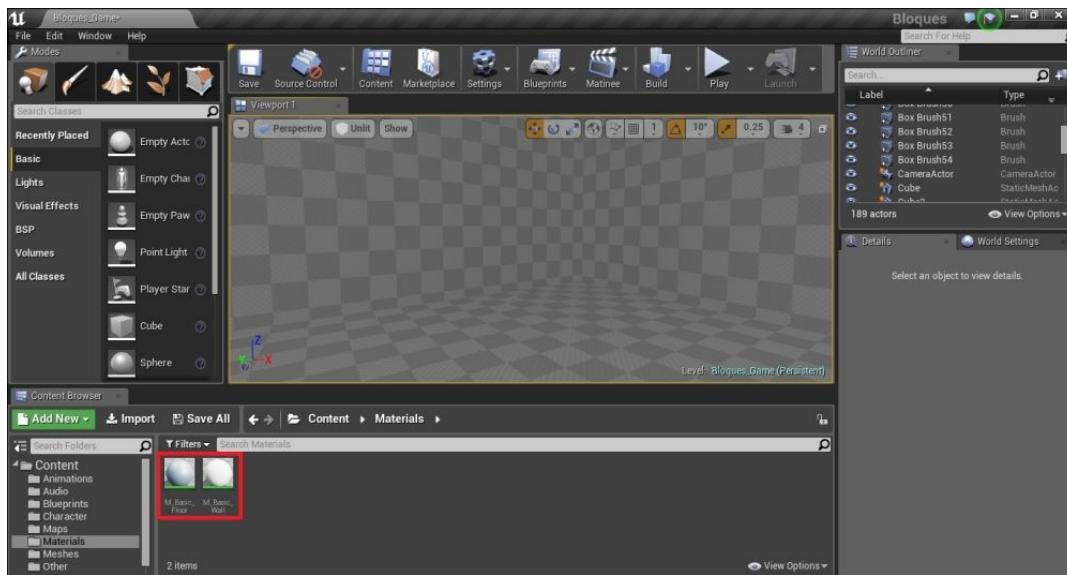
Here is a project with all of the starter content. What we need are two materials, namely **M\_Basic\_Floor** and **M\_Basic\_Wall**. Migrate these two materials and follow these steps:

1. Highlight both **M\_Basic\_Floor** and **M\_Basic\_Wall** and right-click to open a menu. In the menu, hover the cursor over **Asset Actions**, and click on **Migrate**.
2. After you have clicked on it, a window will open up telling you that the following asset(s) will be migrated to another project.



3. Once you click on **OK**, another window will open up asking you where you want to move the asset. You would want to store them in your project's **Content** folder. When this opens up, find your project folder (which, if you recall, is in **My Documents**), and store it in the **Content** subfolder.
4. After clicking on it, you will get a prompt saying that the **asset has been successfully migrated**.

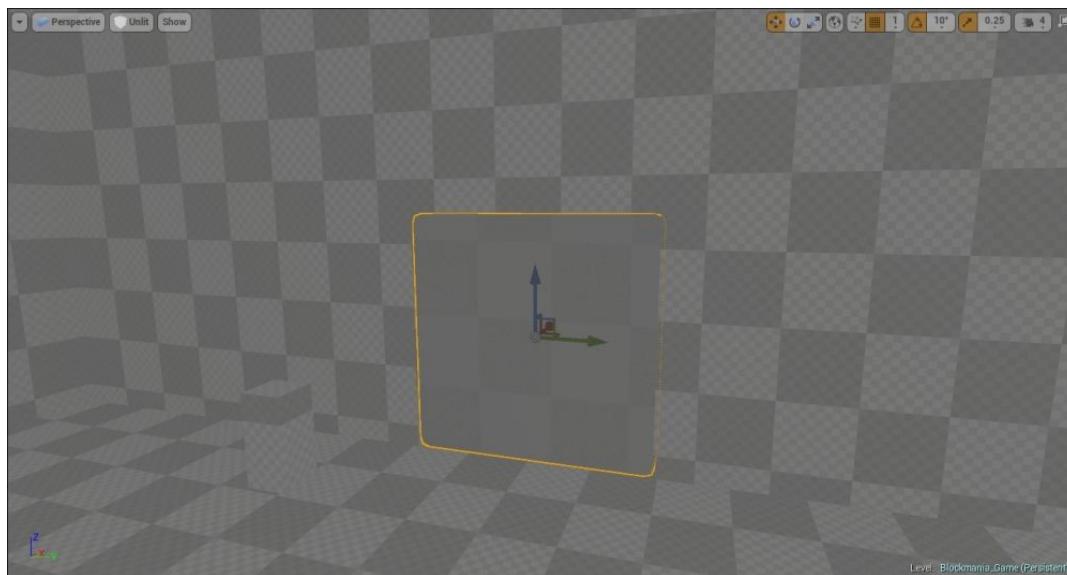
If you now go back to the Blockmania project, you will see **M\_Basic\_Floor** and **M\_Basic\_Wall** displayed in **Content Browser** in the **Materials** subfolder.



## Placing actors

Once you have all of your assets in your **Content Browser**, the next step is to place them in your game. We are going to use the cube mesh in the **Content Browser** to create objects in our game, such as the key cube, the AI-controlled object, the doors, and so on.

First, let's make the door that opens when you place the key cube on the pedestal. Simply drag the **TemplateCube\_Rounded** static mesh from **Content Browser** and place it in the hole where the door is supposed to go. Set its dimensions using the scale tool, so that it perfectly fits.

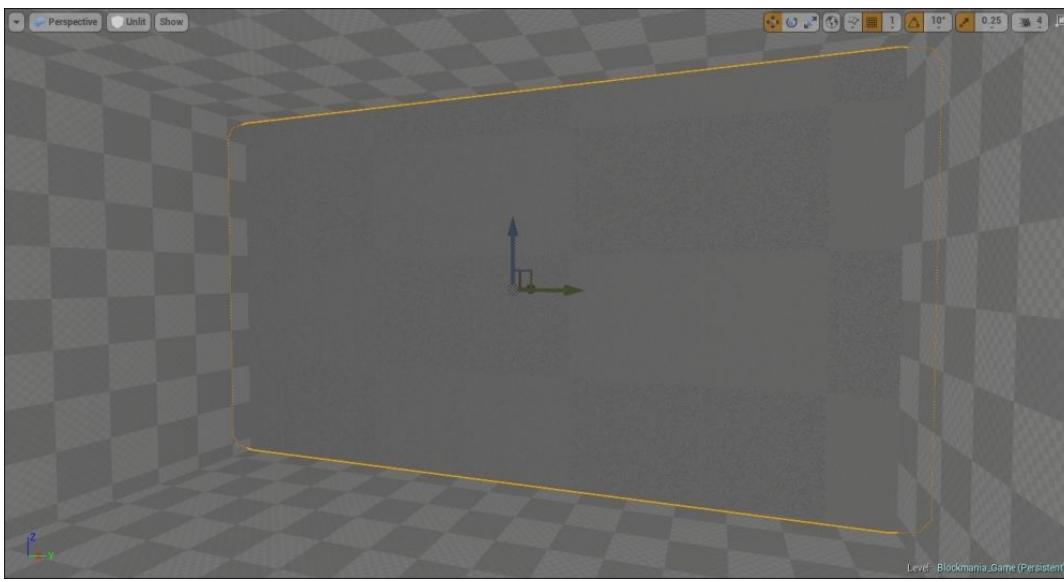


To make things easier and convenient, you can change the name of the cube to **Door01** in the **Details** panel. For the rest of the door, since the dimensions of the hole were the same, you can simply duplicate this actor and place the copies. Finally, since these will be moving and will not be stationary in the game, set its mobility type to **Movable**.

### Tip

Use different perspectives to set the dimensions and to align the door with the hole.

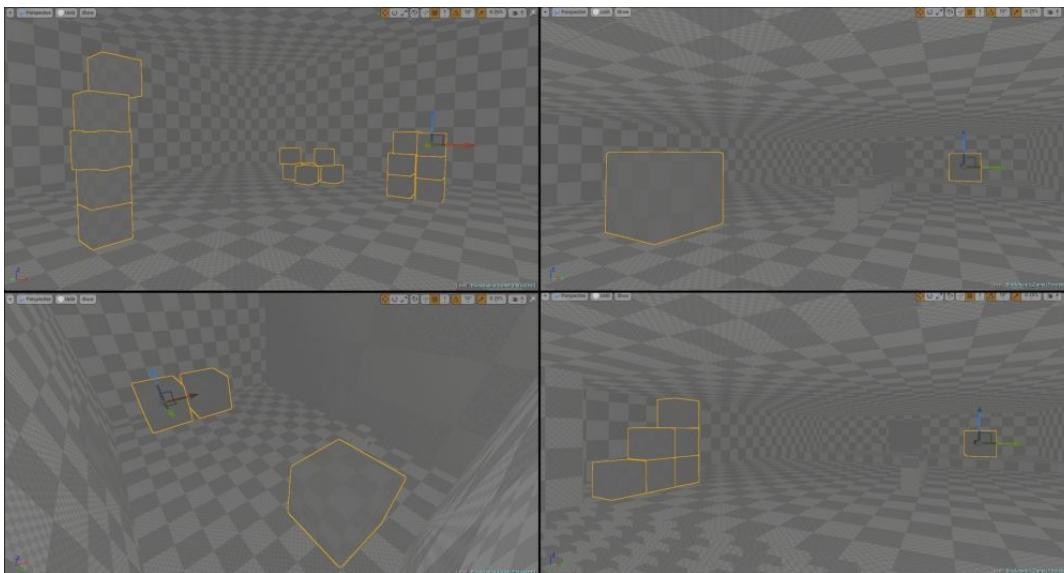
Next, let's place the big doors that the player will encounter in the middle of the rooms. Again, take the **TemplateCube\_Rounded** actor from the **Content Browser** and place it in the second room. Set its dimensions so that it blocks the character from moving to the other side of the room. Name this **Room\_Door01**. Similarly, duplicate and place it in the fourth room, where the ledges are. However, you will have to modify the dimensions of the doors that go in the fourth room, since the dimensions of the fourth room are a little different from the second room.



Next, we have to place the key cubes, which the player has to collect and place in order to progress to the next room, or to unlock other key cubes. We will, again, use **TemplateCube\_Rounded**. Set its scale to 0.15 along all three coordinates and place them in our level. Here is a quick rundown of where the key cubes will be placed in the four rooms:

- **First room**: In the first room, we will only require one key cube. You can place it anywhere near the middle of the room.
- **Second room**: In the second room, we will require two key cubes. Place the first cube on one side of the big door and place the second cube on the other side. The first cube will be the locked one and the second cube is what the player will have to place on the pedestal to unlock it.
- **Third room**: In the third room, we will require only one key cube, which the AI object will unlock upon hitting the target. So, place it on the other side of the pit near the door that leads to the fourth room.
- **Fourth room**: In the fourth room, place the key cube on the other side of the room. This will also be unlocked once the AI object hits the target.

We have placed all of the essential assets in our game. Let's add some decorative cubes in our level. Since decorative assets are going to remain stationary, and not move in the game, you should set their mobility to **Static**. Just place cubes around the map, in different patterns and arrange them as you see fit.



## Materials

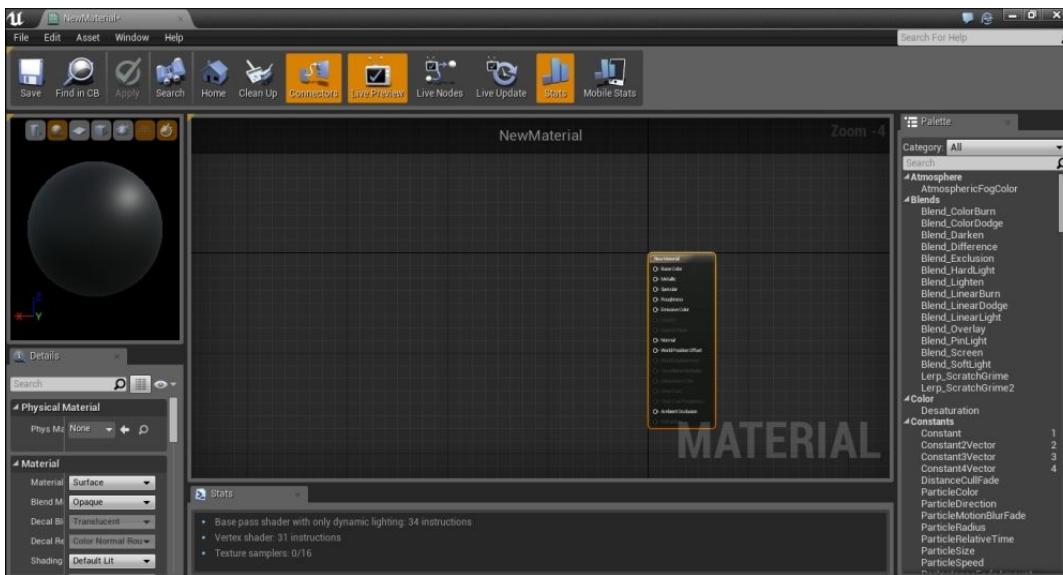
In UE4, before you can apply textures to an object, you first have to create materials. A material is how an object will be rendered in a game, it is a collection of shaders containing many properties that can be applied to objects and rendered in the game. In more technical terms, when light from a light sources fall on a surface or object, the material is what determines how the light will interact with said surface or object (the color, texture, how rough or smooth the surface is, how metallic it is, and so on).

UE4 uses *Physically-based Shading*. In earlier versions of Unreal Engine, the material had some arbitrary properties, such as DiffusePower, Custom Lighting Diffuse, and so on. In UE4, the materials have more relatable properties, such as base color, metallic, roughness, and so on, making the process easier to understand.

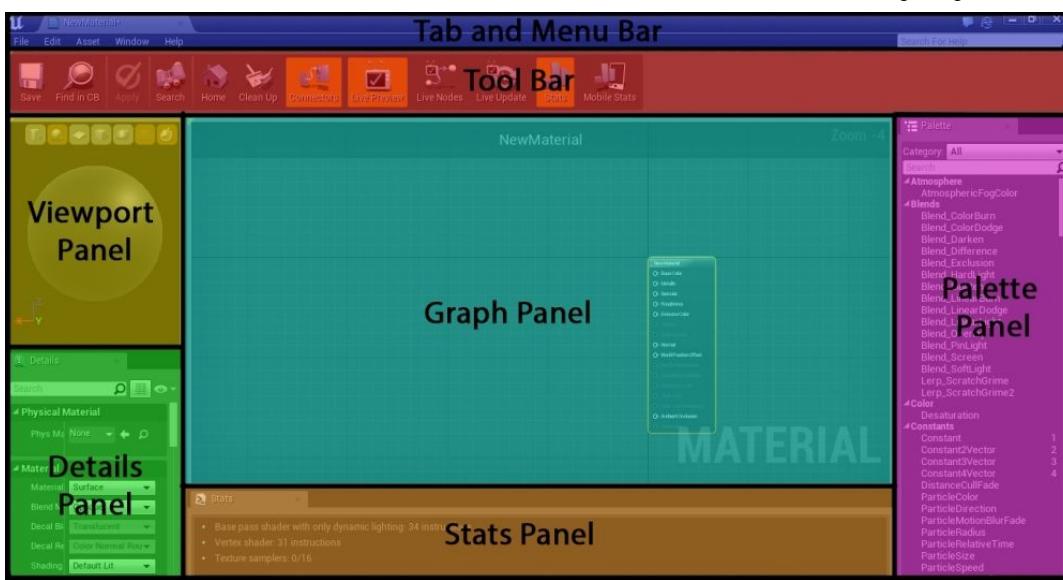
A material is created in what is known as a Material Editor. So, before we start creating our own materials, let's first talk about it and its user interface.

### The Material Editor

The Material Editor is a simple, yet extremely powerful tool that you can use to create materials for your objects. For example, you can apply a texture file to the diffuse channel of your material and then use it on your assets. For our game, we are going to create our own materials from scratch. To access the Material Editor, you first need to create a new material, or double-click on an existing material. To create a new material, click on the **Add New** button located at the top-left corner of **Content Browser** and select **Material**, which will be under the **Create Basic Asset** section. Once done, you will see a new material created. Double-click on it to open the Material Editor. Once opened, you should see this window:



Let's first look at the editor itself and its user interface. As with the Editor, we will divide the Material Editor user interface into sections and go through them individually.



### The tab and menu bar

At the top, we have the tab and menu bar. The tab bar is similar to that of the Editor.



Below the tab bar, we have the menu bar. As with the menu bar in the Editor, it offers all of the general commands and actions. They are described as follows:

- **File** : Clicking on this will open up the **File** menu. From here, you can perform actions such as saving the material you have created, open an asset from the **Content Browser**, and so on.
- **Edit** : This will open up the **Edit** menu. Here, you can undo or redo any actions you might have performed. You can also access the Editor and project settings from here.
- **Asset** : From the **Asset** menu, you can find the material you are creating in the **Content Browser** and open the current material's reference viewer window.
- **Window** : In this menu, you can choose what panel you want in your Material Editor window, search for a particular node in the **Graph** panel, access the plugins window, and more.
- **Help** : Finally, if you want to learn more about the Material Editor or need to find a solution to a problem, you can access the wiki page and the official Epic documentation from here.

### The toolbar

Once again making the comparison with the toolbar in the Editor, the toolbar here displays all of the most commonly used actions that you may perform while making your material.

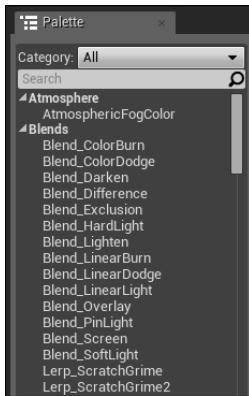


- **Save** : Starting from the left, represented by a floppy disk, is the **Save** button. This is to save any modifications you have made to the material.
- **Find in CB** : **Find in CB** (or Content Browser), depicted by a magnifying glass, locates and highlights the current material you have opened in the Material Editor in the **Content Browser**.
- **Apply** : Next, we have the **Apply** button. If you have already applied the current material to an object in the game, once you have made any modification to it, clicking on **Apply** will update the material in the current scene.
- **Search** : Need to find a node or connection? You can do so by clicking on **Search** and typing in whatever you wish to find. When you click on it, the **Search** panel opens up at the bottom of the material editor. Simply type in whatever it is you want to find, and it will give you the results in the panel.
- **Home** : The **Home** button refocuses the Graph panel to the material input description (the node that you first see in the Material Editor, with all those inputs). This comes in handy when you are creating a very complex material, since you will have a lot of nodes connected to each other and might lose sight of the material description.
- **Clean Up** : The **Clean Up** button deletes any nodes that are not connected to the material inputs. This is very handy since even unconnected nodes will make the material unnecessarily heavy in terms of memory usage.

- Connectors**: This is, by default, turned on. When turned off, any unused input or output pins (pins that are not connected to anything), are hidden. This is a good way of keeping your workspace clean and organized as it removes clutter from view.
- Live Preview**: This, too, is by default turned on. When turned on, any modifications you make to the material are updated in real time in the **Viewport** panel.
- Live Nodes**: When enabled, it updates the material in each node in real time.
- Live Update**: When you toggle this on, it compiles the shaders and all of the nodes and expressions in real time, and it does so every time you change, add, remove a node or change a value of a parameter.
- Stats**: When turned on, you can see the stats of the material you have created at the bottom of the Material Editor. This can give you an idea of just how heavy your material is.
- Mobile Stats**: This is similar to **Stats**, but gives you the material stats and any errors for mobile devices.

### The Palette panel

The Palette panel contains all of the nodes that can be used in the Material Editor.

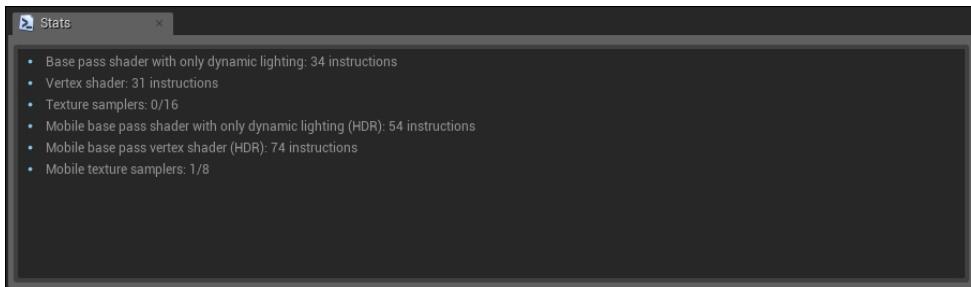


All of the functions, expressions, and so on are categorically listed here. You have the category on the top and you can filter what type of nodes you want displayed using it.

If you wish to find a specific node, you can search for it in the **Search** bar, located below the **Category** bar.

### The Stats panel

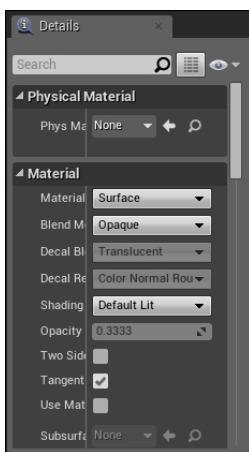
As mentioned previously, the **Stats** panel displays all of the stats regarding the material you have created.



If you have turned on the **Mobile Stats** in the toolbar, you will also see the stats for mobile devices displayed here. This basically shows you how many instructions and shaders are in your material. This will give you an idea as to how big and heavy your material is.

### The Details panel

The **Details** panels displays the general properties of the material, such as **Material Domain**, which means what type of material you want to create.

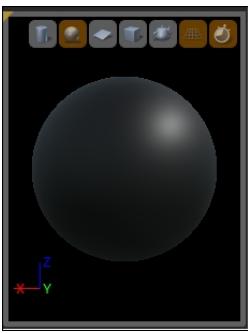


Normal materials, such as the ones you apply to actors, are **Surface** materials. If you want your material to be a **Decal** (which we will discuss in the next chapter), you can switch to **Decal**, and so on.

You can also set properties of any node that you have selected, in this panel.

### The Viewport panel

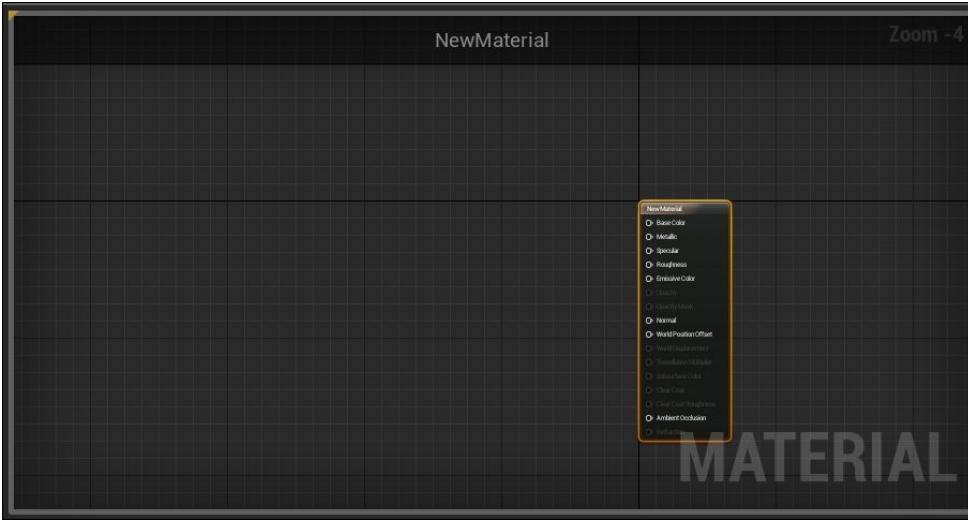
The **Viewport** panel is where you can preview the material you are creating. If the **Live Preview** option is enabled, any modifications you make will be updated in real time and previewed here.



By default, the shape primitive is a sphere. You can switch to either a cylinder, a plane, a cube, or a specific mesh from the Material Editor using the buttons at the top of the Viewport Panel. You can also toggle the grid on/off, and also toggle the Live Preview on/off from here.

### The Graph panel

The last element of the Material Editor's user interface is the Graph panel. This is where you create your material.



Before we continue, there are a few terms and expressions you should be aware of since we will be using them quite a lot in this book. Firstly, we have something called a node. A node is anything that you connect to either make your material or script in blueprint. It could be a variable, an expression, and so on. In the preceding screenshot, the node here would be the long panel you see.

Now, every node has either an input, an output, or both. Input is any value or expression a node takes, and is located on the left side of the node as white pins. Coming back to the screenshot above, the node accepts quite a number of inputs, each input corresponding to a different property of the material.

The output pin is what the node returns (a value, expression, and so on), and it is also represented by white circular pins. They are located on the right side of the node. When connected to two or more nodes, you connect the output pin of the first node to the input pin of the second node, and so on, to create a chain of nodes.

In the preceding screenshot, you can see a long node with a lot of inputs. This is the material input. Each input has a different effect on the material. You might also have noticed that some of the inputs are white, while the rest are darkened. The darkened nodes are the ones that are currently disabled and cannot be used. Which input is enabled and which is disabled depends upon the material's **Blend Mode**, which can be found and changed in the **Details** panel. For example, look at the screenshot, the **Opacity** node is currently disabled. This is because the **Blend Mode** set for the material is **Opaque**, so naturally, you cannot set the opacity of the material. To enable it, you will have to change the **Blend Mode** to **Translucent**.

Let's go over the most commonly used material inputs:

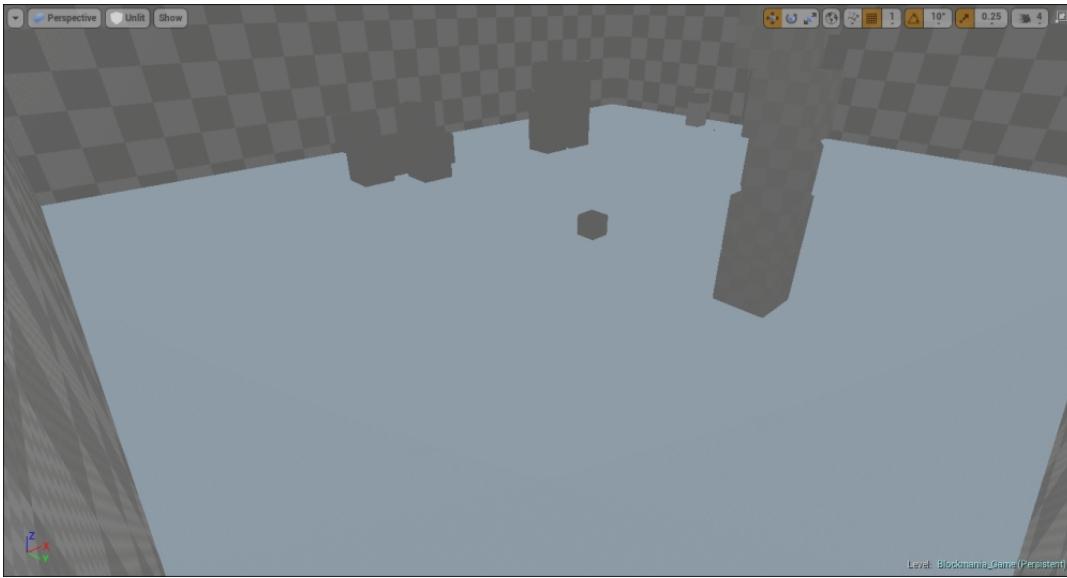
- **Base Color** : This has to do with the color of the material. What the underlying color will be.
- **Metallic** : This deals with how much metallic luster you want in your material.
- **Specular** : Should you want to make your material smooth and shiny, this is the node for you. This deals with how much light is reflected off the material.
- **Roughness** : The opposite of **Specular**, if you want your material to be unsmooth or rugged, this input is what you need.
- **Emissive Color** : If you want your asset to glow, you can set it here, in the Emissive Color input.
- **Opacity** : This deals with how translucent you want your object to be.

There are other inputs as well, but these are the main inputs that you should know for now.

### Applying materials

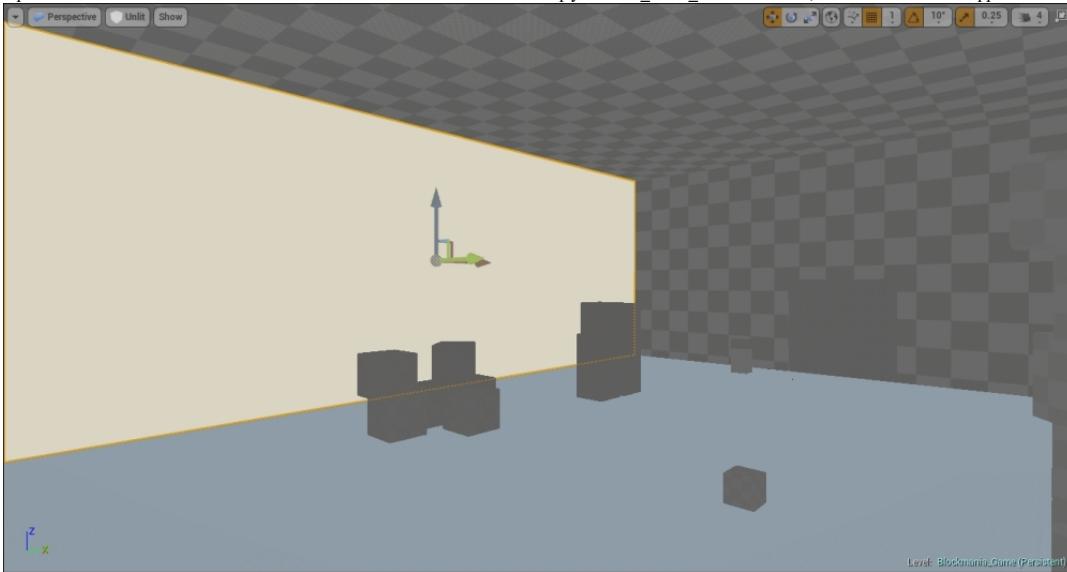
There are two ways of applying materials to objects. The first way to apply materials is simple and straightforward. Let's use the first method to apply the floor material. We will apply **M\_Basic\_Floor**. To apply the material:

1. Select **M\_Basic\_Floor** from the Content Browser.
2. Drag it on to the floor surface.
3. Release the left-mouse button, and the material will be applied to the floor.



We will use the second method of applying materials for the walls. We will apply `M_Basic_Wall` to our walls. To do so, follow these steps:

1. Select any of the wall surfaces.
2. In the **Details** panel, under the **Surface Materials** section, you will see something called **Element 0**. Next to it, there will be a menu button with **None** written on it.
3. Open the menu. You will see all of the materials in the Content Browser listed out. Simply select `M_Basic_Wall` from the list, and the material will be applied to the wall.



Using any of the two methods, apply the materials to all of the walls and floors in the game.

## Creating the materials

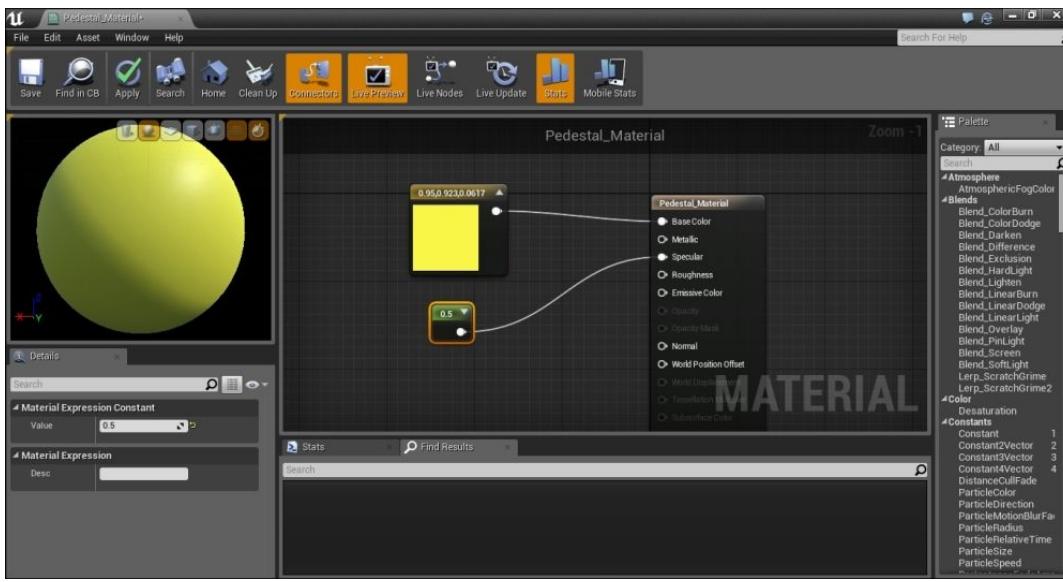
Now that we have covered the Material Editor's user interface and seen how to apply them onto objects, let's go on and create the materials for our level.

### Pedestals

First, we are going to create a material for the pedestals where the player has to place the cubes. We are going to create a fairly simple material. The pedestal we will create will be yellow in color and will be a bit shiny. First, create a new material in **Content Browser** and name it `Pedestal_Material1`. Then double-click on it to open the Material Editor.

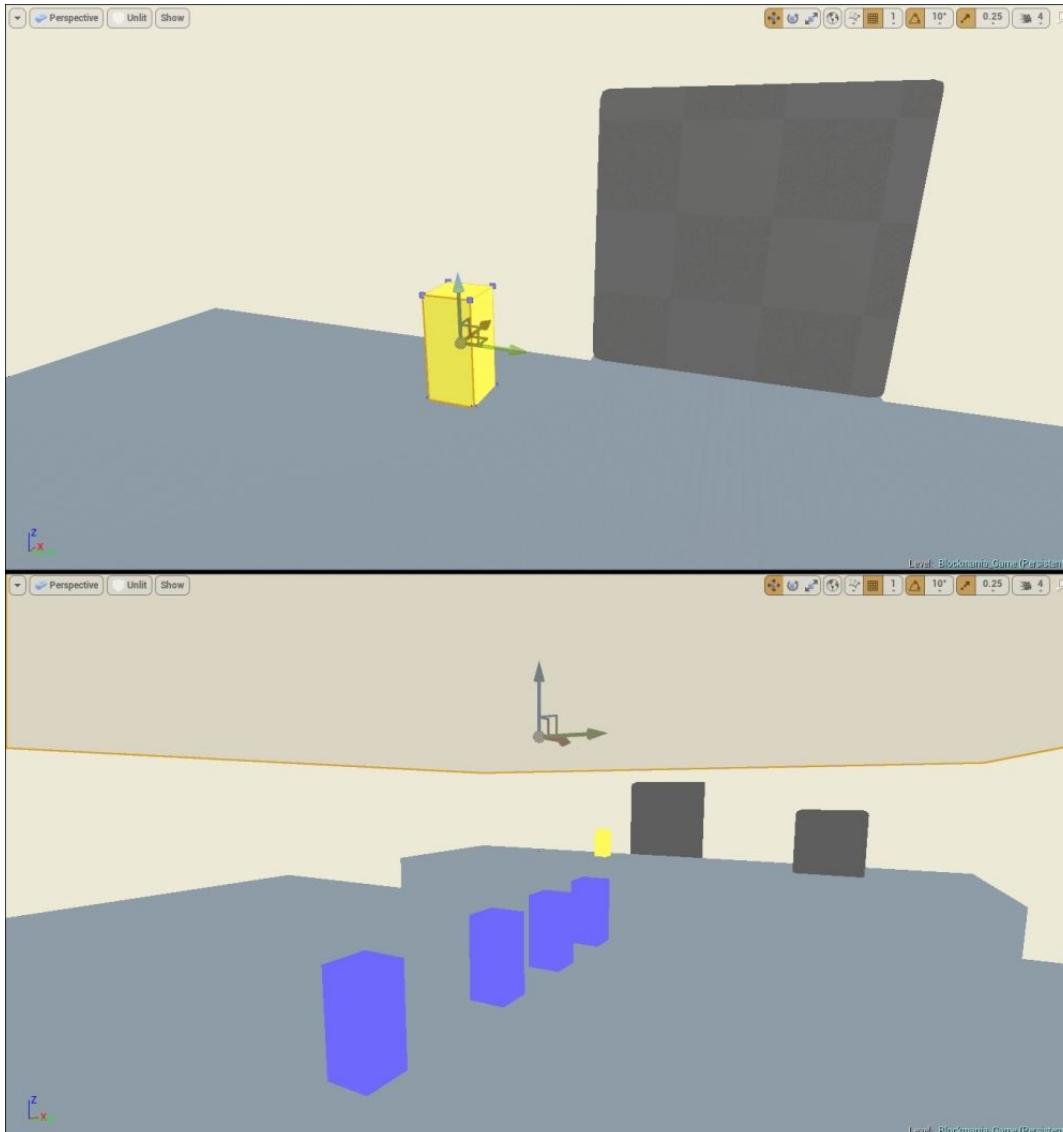
The first thing we are going to do is add something to the **Base Color** input. What we need to create is something called a **Constant3Vector** node. To create it, simply drag the node from the **Palette** panel and place it in the **Graph** panel. Another much quicker way of creating a **Constant3Vector** node is by holding down **Shift** and left-clicking anywhere in the **Graph** panel. The node will be formed wherever you click. Once formed, connect it to the **Base Color** input. You can set the color to yellow. To do this, simply select the **Constant3Vector** node and in the **Details** panel, you will see a property called **Constant**, with a black bar next to it. Click on the black bar, which will open the **ColorPicker** window. Set the color to yellow and click on **OK**. Now connect it to the **Base Color** input. You will see the **Viewport** panel updated with the change you have made.

Now that we have the base color, we are going to add the shine. For this, we are going to create a **Constant** node. Again, you can either drag it from the palette panel or hold **Shift** and click anywhere on the **Graph** panel. In the **Details** panel, set its value to **0.5**, then connect it to the **Specular** input. Your **Graph** editor will look something like this:



Now that you have created the material, click on **Save**, then apply it to the pedestals.

For the pedestals where the buttons will be, create a copy of **Pedestal\_Material1** by right-clicking on it and selecting **Duplicate**, rename it to **Pedestal\_Material2**, and in the material editor, simply change the base color to blue, save it, and apply it.

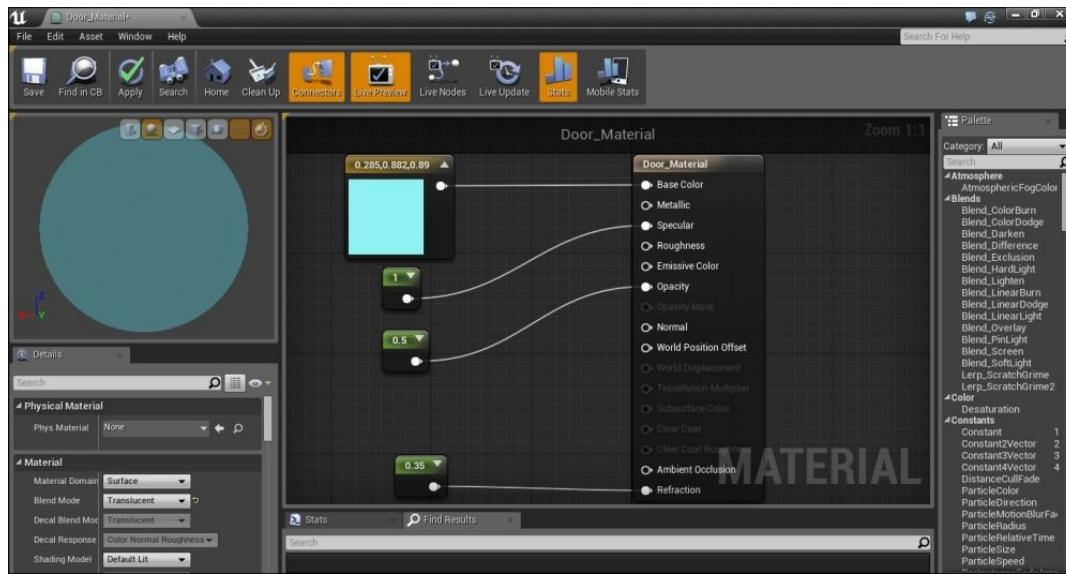


## Doors

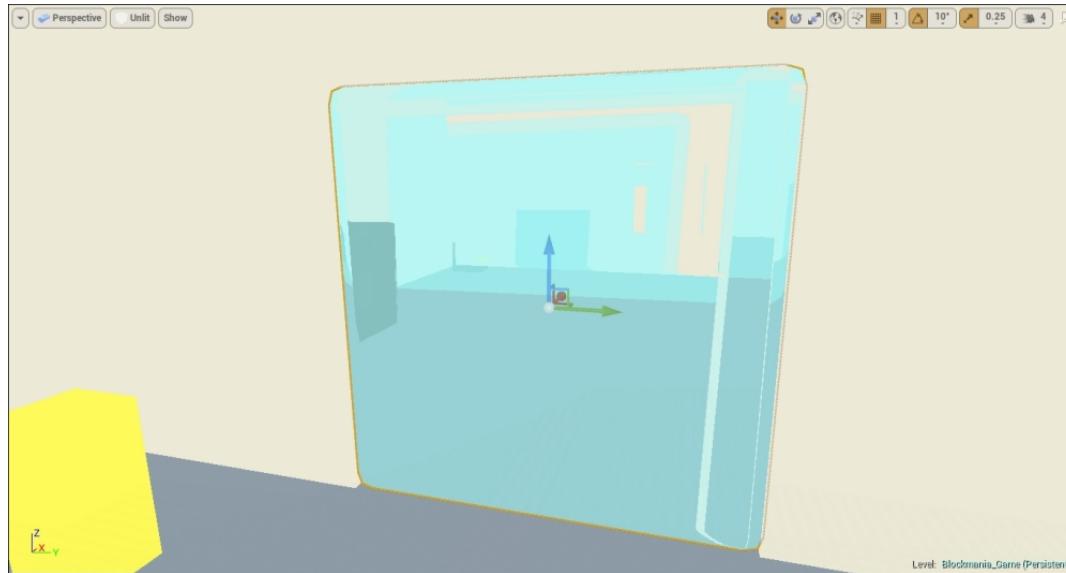
For the doors, we are going to do pretty much the same thing we did when we created the materials for the pedestals. The only difference here is that we will make the doors a bit see-through, and make the material glassy. With that said, create a material for the door, and name it **Door\_Material**.

In the Material Editor, the first thing you need to do is set the material's blend mode to **Translucent**. After that, create a **Constant3Vector**, pick a light blue color, and connect it to the **Base Color** input.

Now, create three **Constant** nodes. Set the value of the first node as **1**, and connect it to the **Specular** input. Set the value of the **Second** node as **0.5**, and connect it to the **Opacity** input. Our glass material is now translucent. However, there is still something left, a very fundamental property of glass—refraction! The refraction input is located at the bottom of the input panel. Set the value of the third **Constant** node to **0.35**, and connect it to the **Refraction** input.



We have created a basic glass material for our door. Save this and apply it to the doors.

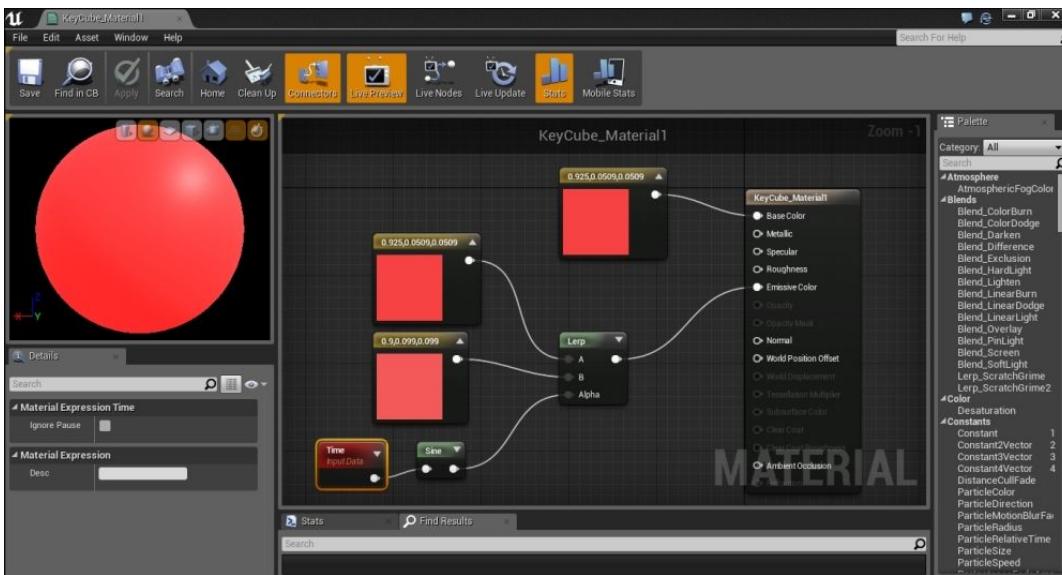


## Key Cubes

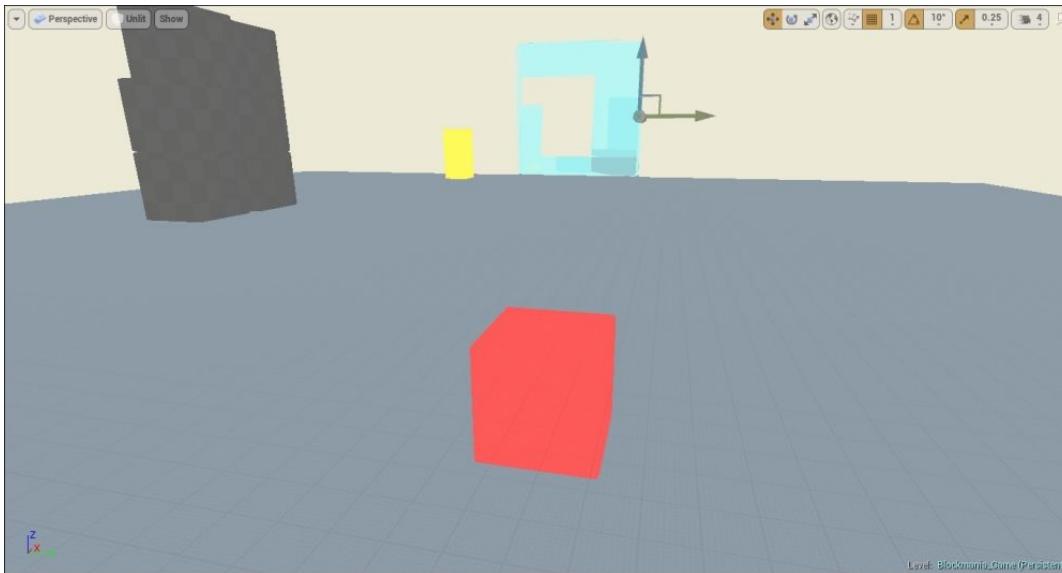
Let's make the set of materials a bit interesting. We are going to make it red. Also, we are going to make it flash so as to grab the player's attention. Create a material and name it **KeyCube\_Material**.

First, create a **Constant3Vector** node, set the color to red, and connect it to the **Base Color** input. This will be our base color. For the glowing effect, we are going to use a **Linear Interpolate** function. A **Linear Interpolate** takes in three inputs. It blends the first two inputs (**A** and **B**) and the third input is used as a mask (Alpha). For input **A**, create a copy of the **Constant3Vector** node you created for the base color and connect it. For input **B**, create another **Constant3Vector** node, set the color to a lighter shade of red and connect it.

We have our inputs. We now need to create something for the **Alpha** input. Since the key cube will be flashing in regular intervals, we can use the **Sine** or **Cosine** function. You can find any of the two in the palette panel. Place it in the Graph panel, and connect it to the **Alpha** input. Now, for the **Sine** or **Cosine** function to actually work, it needs some input data. We are going to create a **Time input data** node. Simply type in **Time** in the Palette panel, place the node in the Graph panel, and connect it to the **Sine** or **Cosine** node. Finally, after doing all that, connect the **Linear Interpolate** to the **Emissive** input.



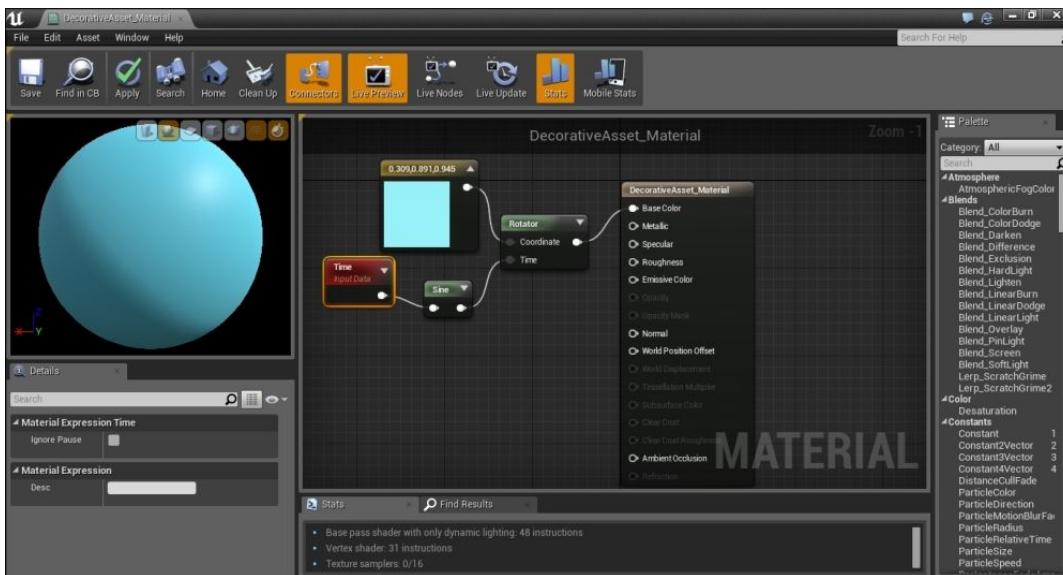
And there you have it. We now have a flashing material for our main key cube. Apply it to all the key cubes.



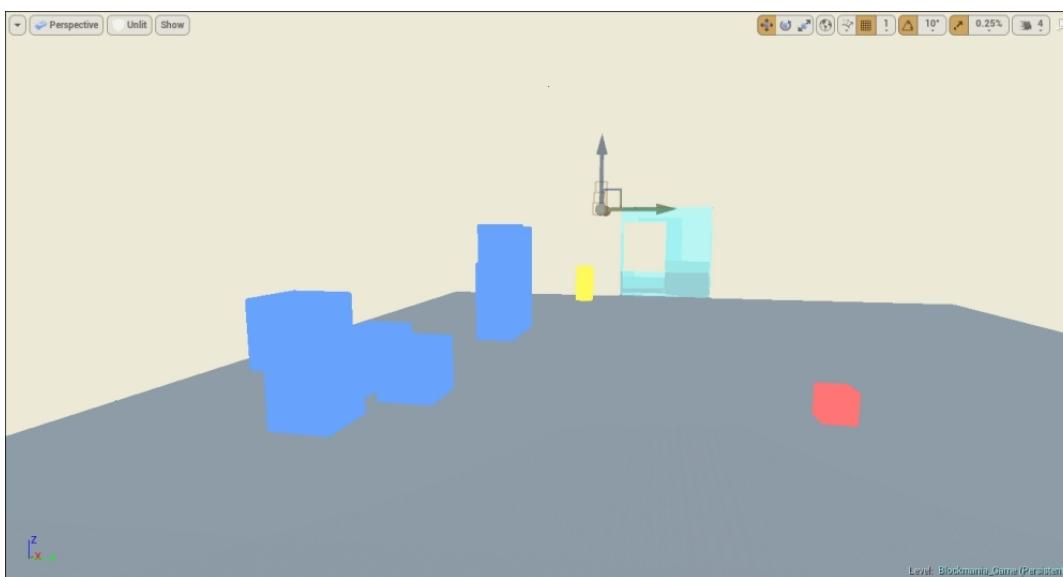
## Decorative assets

For our decorative assets, we are going to create a material that changes colors periodically. First up, create a material and name it **DecorativeAsset\_Material**. For the changing effect, we are going to use a **Rotator** node. If you have a proper texture, and wish to have it rotate, you can do so using this. It takes the *UV* coordinates of the **Textures** and a **Time** function and uses that to rotate the texture. We are going to use it to have our material change color. As always, create a **Constant3Vector**, and set its color to a pale blue.

Now, instead of connecting it directly to the base color input, we are going to connect it to the **Rotator** node. Find the **Rotator** node in the palette panel, and place it in the graph panel. The **Rotator** node takes in two inputs, **Coordinate** and **Time**. Connect the **Constant3Vector** to the **Coordinate** input. Create a **Sine** function, connect a time input data to it, and connect it to the **Time** input. Finally, connect the **Rotator** node to the **Base Color** input. That is all we need to do. You will now notice the material changing color periodically. You can set the speed at which it changes color by changing the **Speed** setting in the **Rotator** node's Details panel.



Apply this material to all of the decorative assets in the game. And with that, we have textured our environment.



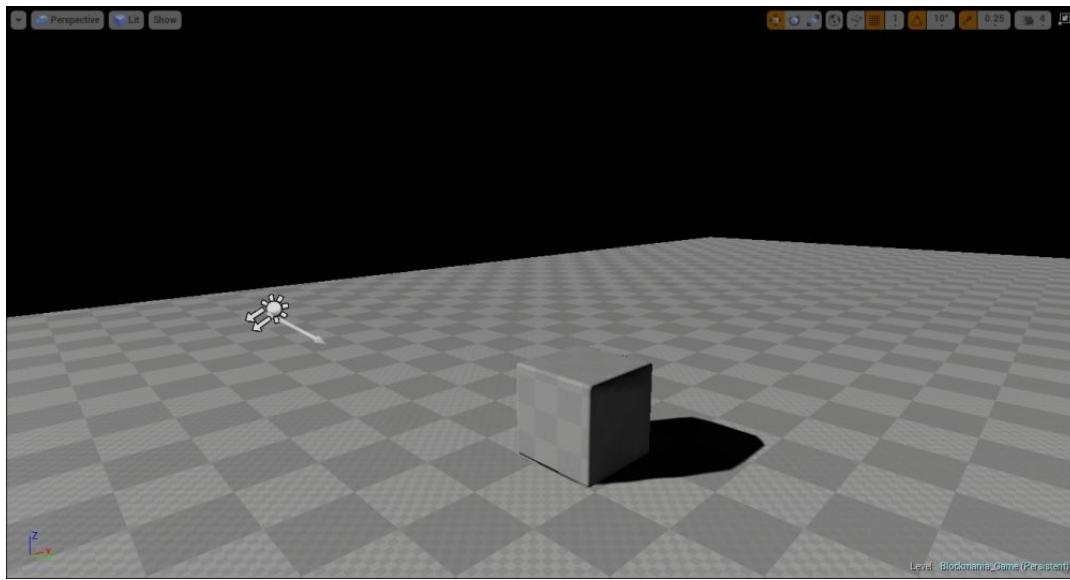
If you test out your level, you would notice that you cannot see anything. This is because our level does not have any lights, therefore nothing is rendered and all you see is a black screen. Let's now move on to **lighting**.

## Lighting

Lighting is a very important element of any game, as without it you cannot see the world around you. UE4 offers four types of lighting, namely: Directional light, Point light, Spot light, and Sky light. Let's go over each of them individually and discuss some of their properties that you can set in the Details panel:

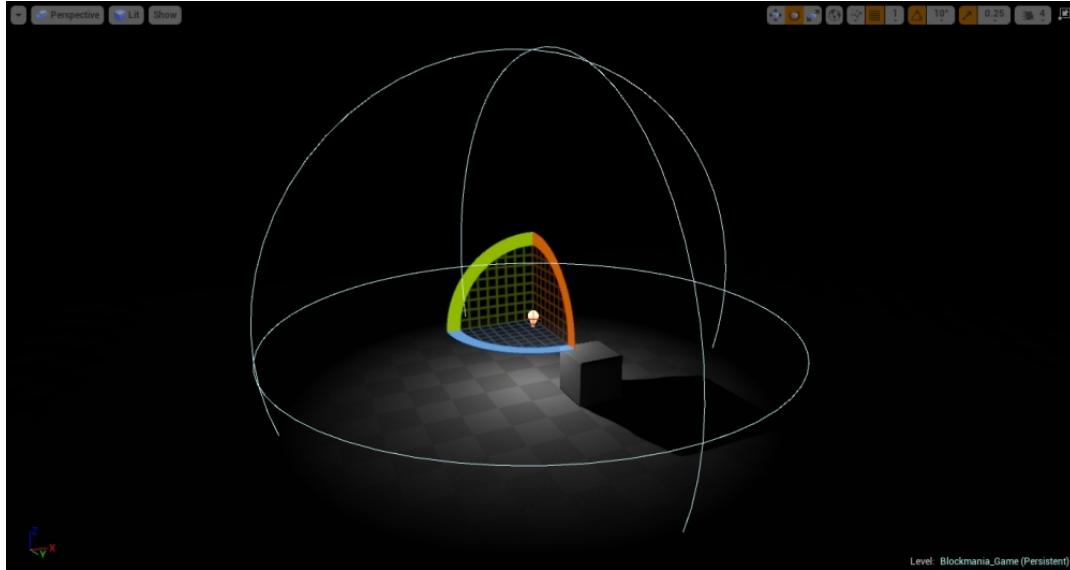
- **Directional light**: Directional light is the ideal type of light when you have an outdoor scene and want to simulate light coming from the sun, since directional light simulates light coming from a source infinitely far away. The shadows formed by directional lights are parallel.

You can set things such as the intensity, color, whether it affects the world, the intensity of the indirect lights, and more in the Details panel.



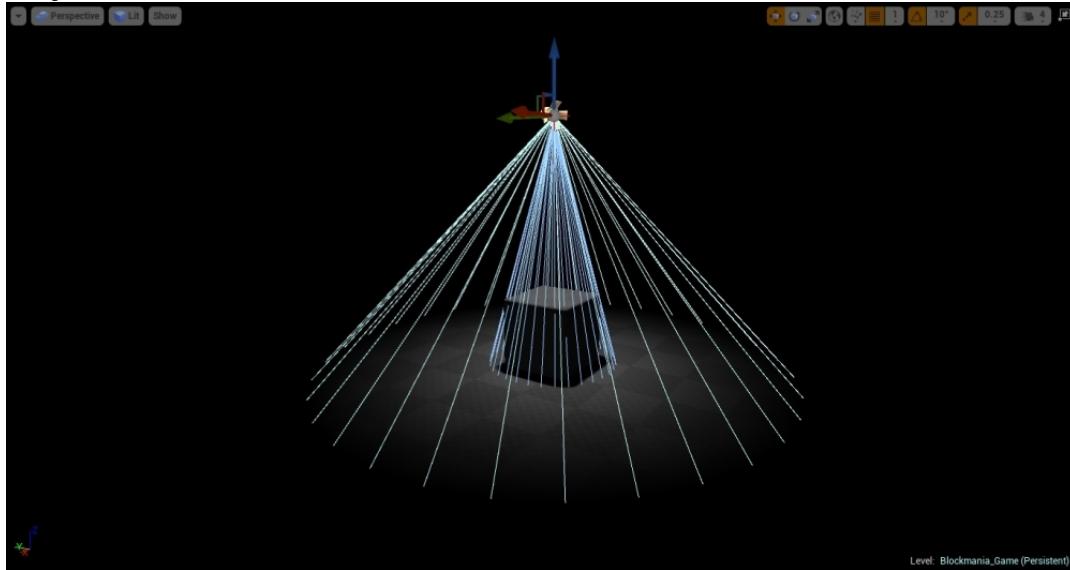
The preceding screenshot shows directional light and how it works in the level. Its icon is depicted by a sun with two parallel arrows coming out of it. The other single arrow shows the direction from where the light is coming. You can set the direction by using the rotation tool.

- **Point light** : A point simulates light coming from a single source of light, emitting light uniformly in all directions, much like a bulb. It is the light source we will be using most in the game. In an indoor scene, this will be your primary actor for lighting.



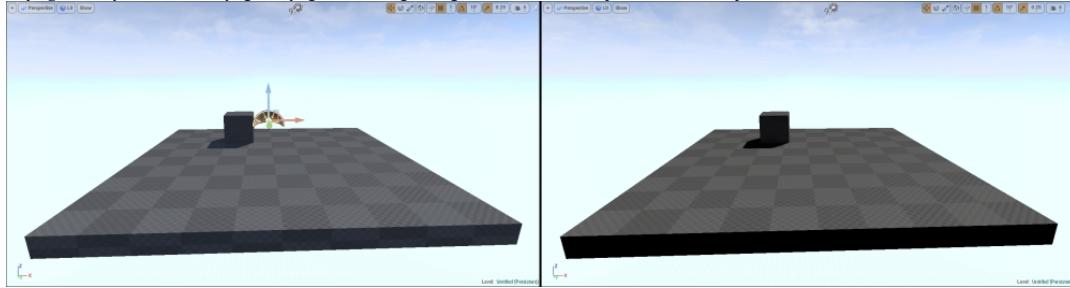
As you can see, the point light actor is represented by a light bulb. The sphere you see around it is the *Attenuation Radius*. It is represented by a sphere wireframe, and shows the volume where the light source has direct influence. You can set things such as the intensity, the color, the attenuation radius, whether the light source affects the world or not, the intensity of the indirect lighting, and so on. If you feel like you do not want a single point light source, you can set a radius or length of the light source, depending on your requirement.

- **Spot light** : Spot lights are similar as point light, in that, they both originate from a single source. The difference between the two, however, is that while a point light emits lights uniformly in all directions, spot light emits lights in one direction, in a cone.



There are two cones that you may have noticed. The inner cone is where the light is the brightest. As you move radially outwards, towards the outer cone, the light becomes less bright and fall-off takes place. You can set the radius of these two cones, along with the intensity, color, length, and radius of the source, and so on.

- **Sky light :** Lastly, we have the sky light. Sky light simulates light being reflected off the atmosphere and distant objects.



Sky light is quite subtle in terms of effects. With that in mind, in the preceding screenshot, the left is a scene with sky light and to the right, without Sky light. You can see the effect of Sky light when the two are juxtaposed.

Sky light settings are a bit different than the rest of the types of lights. For one, you can set something called the **Sky Distance Threshold**. This is the distance from the Sky light actor at which any actor will be treated as part of the sky. You also have the **Lower Hemisphere is Black** option, which, when toggled, will ignore the light coming from the lower hemisphere of the scene. You can also use your own custom cube map to get your own type of Sky lights. Apart from that, you can set the color of the light, the intensity, and so on.

## Mobility

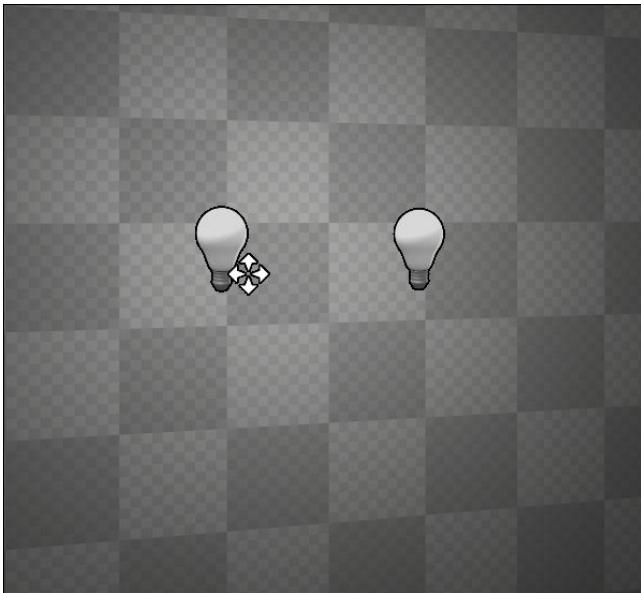
All the types of lights have three types of mobility. There are certain settings of light characters that you can modify while the game is running, should you require to. The mobility section allows you to do just that.

The mobility options are located in the **Transform** section of the light actor's Details panel.



There are a total of three types to choose from

- **Static** : When you set the light as **Static**, the properties set while making the game remain constant and cannot be modified with triggers or during gameplay. Once the lighting has been built, the light information and the shadows baked by that light remain static and cannot be changed. For lights that do no move or toggle on/off during the game, this is the most apt mobility type, since it requires the least amount of memory to render.
- **Stationary** : When a light actor's mobility is set to **Stationary**, the only properties that can be altered with triggers or during gameplay are things like intensity, color, and so on. The light, however, cannot move or translate under this setting. This requires more memory than static lights, and can be used for things like flickering lights.
- **Movable** : Under this setting, the light actor is totally dynamic and along with altering things like intensity and color, it can also move, rotate, and scale during gameplay. Keep in mind, however, that this type of light takes the most memory to render, since every time its property is altered within the game, it has to recalculate the shadows and lighting information in real time. Therefore, unless it is really required in your game, avoid using movable lights when developing games on mobile devices. It is better if you use static lights, for best performance. When you switch the mobility to **Movable**, the light actor's icon in the game changes. You will see four arrows underneath the light actor.



On the left, is a light actor with the mobility set to **Movable**, and to the right is a light actor with the mobility set to either **Static** or **Stationary**.

## Lighting up the environment

Now that you are familiar with light actors, their types, and mobility types, let's place some lights onto our level.

For now, we are just going to place Static point lights in our level. We will cover stationary and movable lights later on. With that said, place a point light in the level. Set `intensity` to 20000, and `attenuationradius` to 5000. Also, set the mobility to `Static`. Create multiple copies of the point light and place them, keeping in mind that there should be sufficient lighting for the player. You should also switch to `Lit view mode`, so that you can figure out if there is enough lighting in the rooms. Finally, build the level. You can set the quality of the lighting build. But remember, the higher the quality of light you want, the longer it will take to build, but the lighting will be closer to reflecting production quality at these higher settings.



You can now see all of the materials and textures properly rendered.

## Summary

In this chapter, we talked about projects, what they are, what types of projects UE4 offers, and how to create and load projects. Following that, we covered what BSP brushes are, the default types of BSP brushes and how we can edit them to create our own geometry. We returned and took another look at Content Browser, how to import and migrate assets, and how to place actors from Content Browser onto the level. We covered the concept of materials, the material editor, its user interface, and how to create materials. Finally, we ended the chapter by talking about lighting, the different types of lights, and some of the important settings.

Using this knowledge, we created our environment, hence taking our first steps towards building our game. In the next chapter, we will discuss the concept of volumes, what they are, the different types of volumes, and how we can apply them in our game. So, without further ado, let us move on to the next chapter.

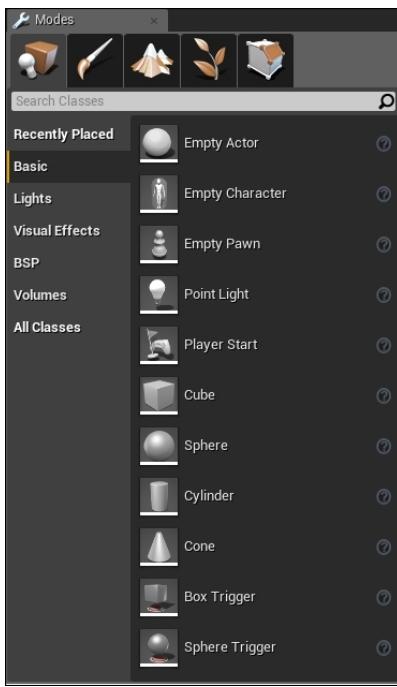
## Chapter 4. Using Actors, Classes, and Volumes

We have our environment set up with all of the essential actors and objects placed. There are other equally, if not more, important types of objects (or actors), without which the game would not be complete. These are Visual and Basic classes and Volumes, which are actors with special properties. Some are vital to the game, while some add special features. In this chapter, we will be looking at some of these volumes and classes and how they affect the game. We will cover the following topics:

- Basic classes
- Visual Effects
- Volumes
- All Classes

### Basic classes

We will kick off with an introduction to basic classes. These can be accessed in the **Modes** panel under the **Basic** section.



This contains the most basic classes that are essential to every almost all games, regardless of their type or genre. Let us go over them.

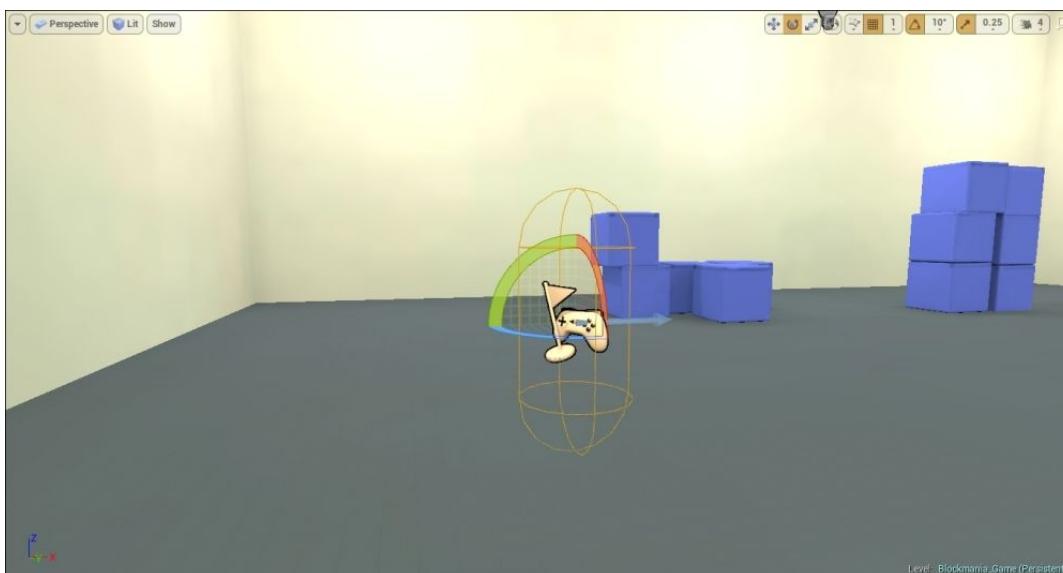
- **Empty Actor** : An Actor is any object that is placed in the game world. All objects, lights, cameras, volumes, and so on are actors. An empty actor is an empty entity that you can place in your level. It does not have any inherent properties.
- **Empty Character** : An **Empty Character** does not have a mesh or any animations—just a collision capsule.
- **Empty Pawn** : A **Pawn** is an actor that can be possessed (in other words, controlled) by a player or the AI. The game's characters, all of the enemies, allies, and NPCs in the game are all pawns.
- **Player Start** : As the name suggests, **Player Start** is where the player spawns when playing the game. If there is no Player Start actor in the level, the player will spawn at the origin of the world ( $0, 0, 0$ ).
- **Point Light** : As mentioned in the previous chapter, **Point Light** is the most basic and most widely used source of lighting. You can also add the Point Light actor from here.
- **Cube** : This adds a cube primitive (static mesh) to the game.
- **Sphere** : This adds a sphere primitive to the game.
- **Cylinder** : This adds a cylinder primitive to the game.
- **Cone** : This adds a cone primitive to the game.
- **Box trigger / Sphere trigger** : The next two actors have been clubbed together since they serve the same purpose; the only difference is their shapes. Triggers, simply put, add interactivity to the game. You can add an event for the trigger (for instance, if the player touches it, hits a specific key, and so on), which, when fulfilled, carries out a specific action as set by the developer. For example, you can have a trigger, which when the player touches, turns on a light, and so on. UE4 offers two default shapes: box and sphere. You can also create custom shape triggers, but more on that later.

## Adding basic class actors to the game

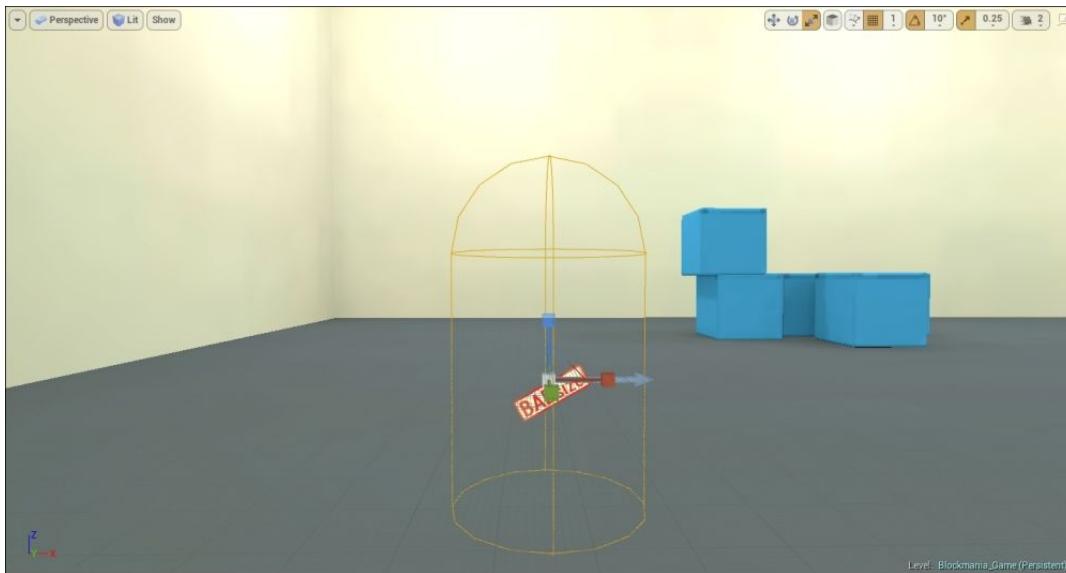
Now that you are acquainted with the basic classes, let us go ahead and add some to our level.

### Placing the Player Start actor

The first thing that we are going to place is the Player Start actor. Its placement is vital and should be decided beforehand. In our game, we would want our player to start in the first room. Keeping that in mind, drag the Player Start actor and place it in the first room, away from the door.



The actor is represented by a gamepad with a flag next to it. As you can see, there is a capsule-shaped volume around it. This capsule is there to give you an idea about the size of the character as well as its placement when the game starts. So make sure that the capsule does not overlap any other actor or surface, as doing so will change the icon into a sign saying **Bad Size**. You can resize this capsule by using the **Scale** tool so that it fits with the character.



You may have also noticed a blue arrow along with the icon. The direction in which the arrow points is where the character will face when the game starts (remember, *W* is for the translate tool, *E* is for the rotation tool, and *R* is for the scaling tool). You can change this direction with the help of the rotation action. If you were to click on the **Play** button, the character would spawn where the actor is placed, facing the direction of the arrow.

### Adding triggers

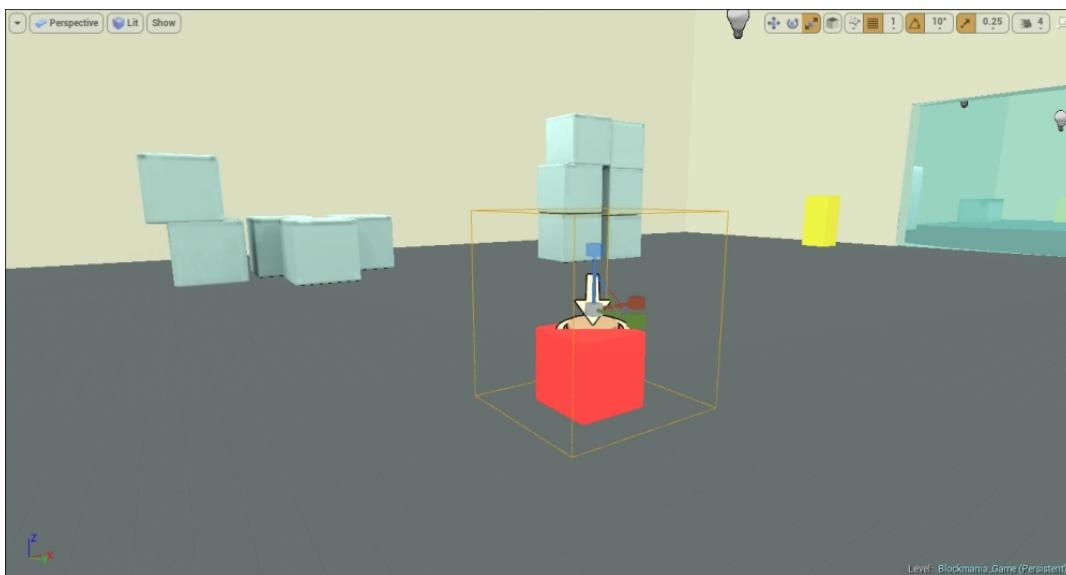
Next, we are going to add triggers. We are going to keep things simple by using only Box triggers. To place a Box trigger, simply drag it from the **Modes** panel and place it on the level. What we will do with the triggers we will cover in the next chapter. For now, you can simply place the triggers in the locations mentioned in the following sections.

Always name your actors, be it triggers, lights, characters, or so on. It is not only considered good practice but it will make your project easier to read and will keep everything organized and is easier to track.

#### Room 1

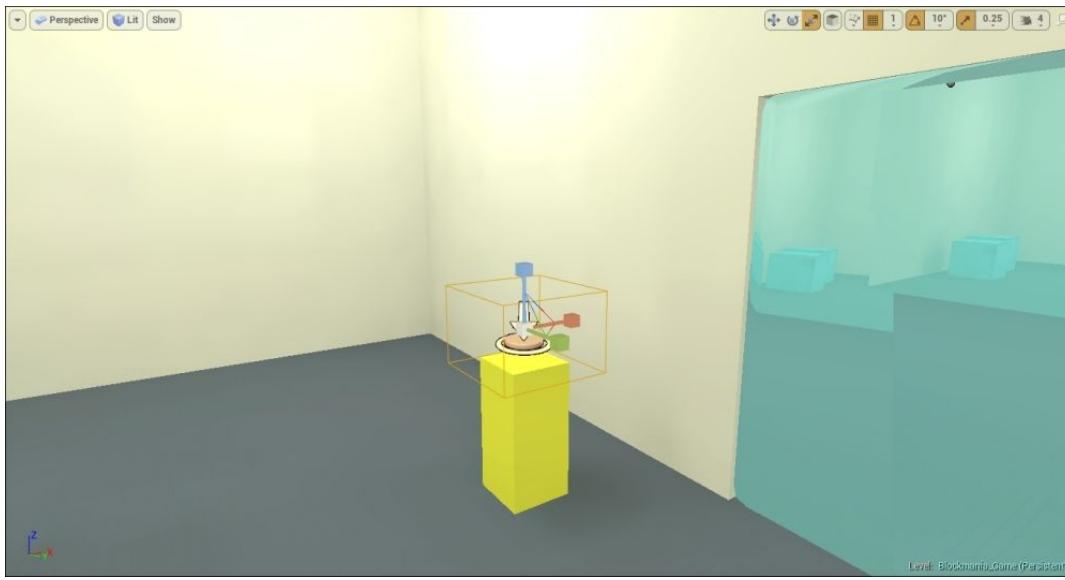
In the first room, place the first trigger near the key cube. This trigger will be used to interact with the key cube. The placement and dimensions of this trigger (or any trigger for that matter) are important, since they determine how far the player can interact with the key cube. For example, if the trigger is too large, the player will be able to pick up the key cube from afar, which is not what we want. We want the player to be relatively close to or adjacent to the key cube before they can pick it up.

Keeping that in mind, place the trigger and set the dimensions such that it encapsulates the entire key cube (so that the player can pick it up from any direction), and make it taller (so that the player does not have to look directly at the key cube to pick it up; otherwise, it will get annoying). Finally, make the trigger bigger than the key cube. Once set, it should look something like the following screenshot:



Add the next trigger in this room on the pedestal. This is where the player will have to place the key cube in order to open the door.

Once again, drag and drop a Box Trigger, and place it on top of the pedestal. Again, as we did with the trigger for the key cube, set this trigger's dimensions such that the player can interact with it from any direction and does not have to be standing right next to it in order to interact with it.



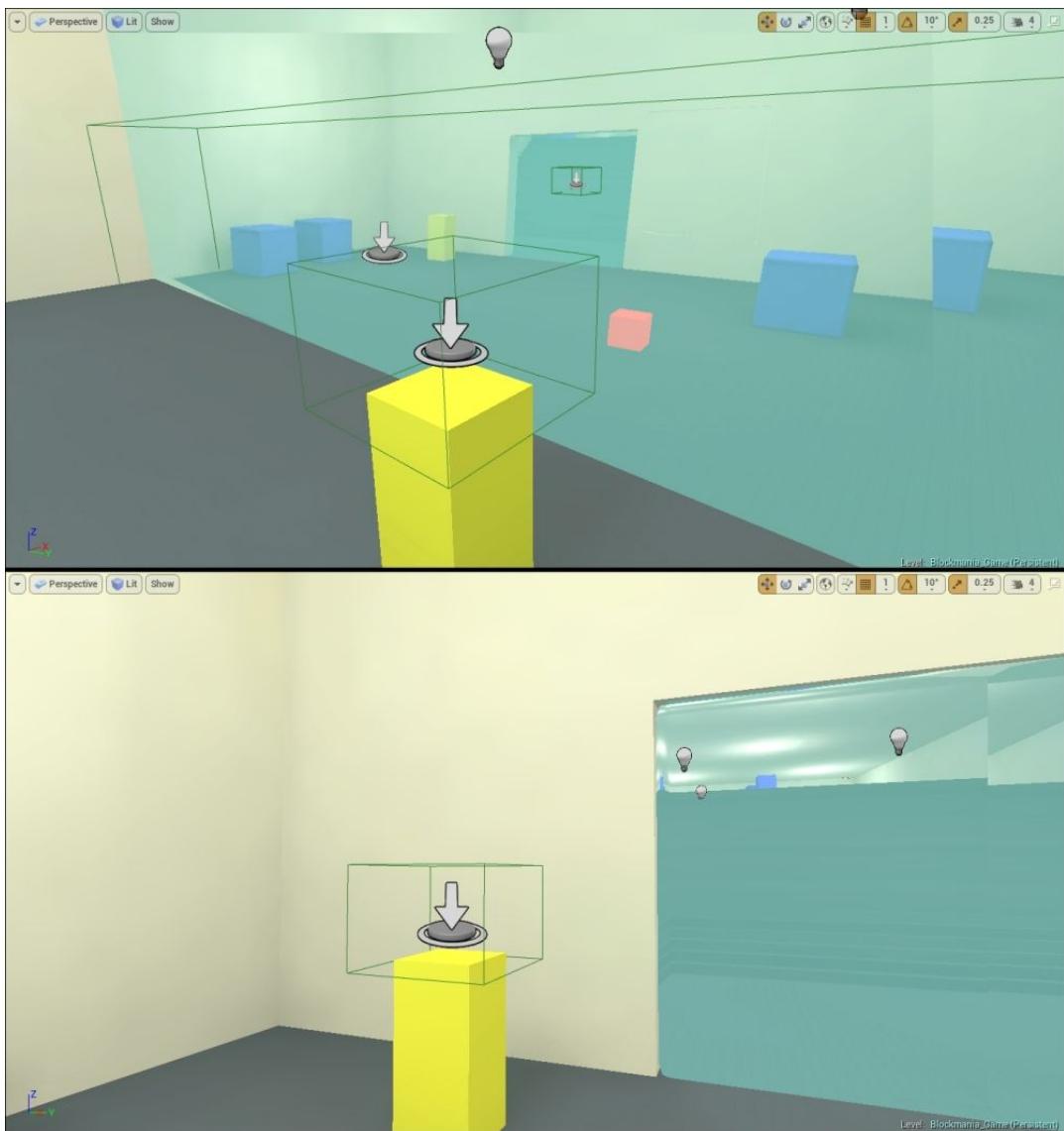
To check the placement of the trigger and whether its position and dimensions are correctly set, you can first unhide the trigger actor by unchecking the **Actor Hidden in Game** option—found in the trigger's **Details** panel and then play the level. For now, this will do in terms of trigger placement. Let us now move on to the second room.

#### Room 2

In the second room, place the first trigger near the large door in the middle of the room. Now, we would want the player to be able to open the door from anywhere. Keeping that in mind, place the trigger such that it covers the entire door lengthwise and widthwise. Again, adjust the width, keeping in mind how close you want the player to be in order to interact with the door.



Next, place the next two triggers that on the pedestal, similar to that in the other rooms. Since the pedestals have similar dimensions, you can place the triggers by duplicating them in the previous room.



We are not going to place any triggers on the key cubes as we did in the previous room. The reason behind this will be explained in the next chapter.

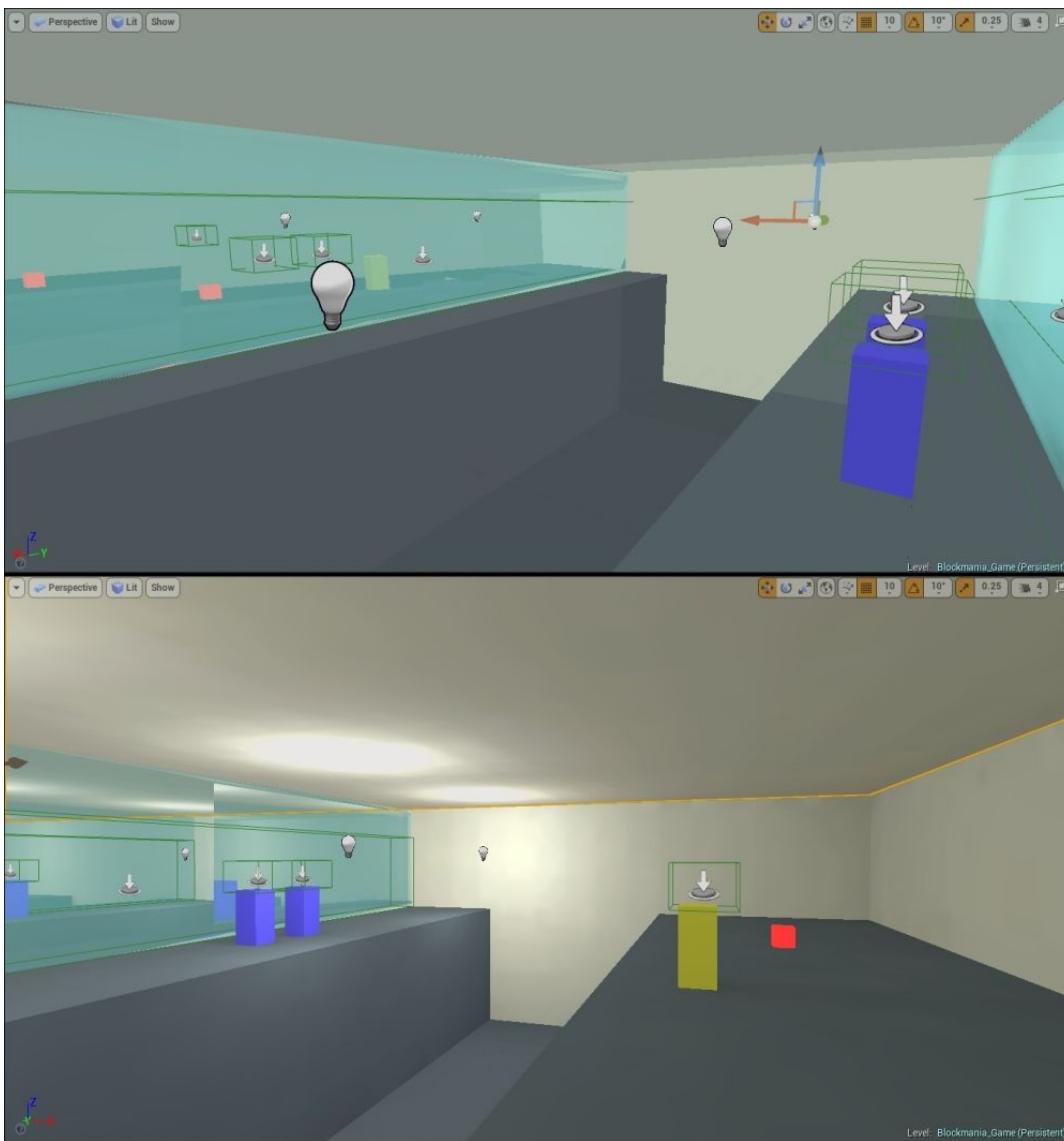
### Room 3

In the third room, we have a couple of pedestals upon which there will be buttons. Therefore, we will require triggers for interactivity. Again, do not place triggers on the key cube.



#### Room 4

Finally, in the fourth room, as with the previous rooms, place triggers on all of the pedestals and doors.

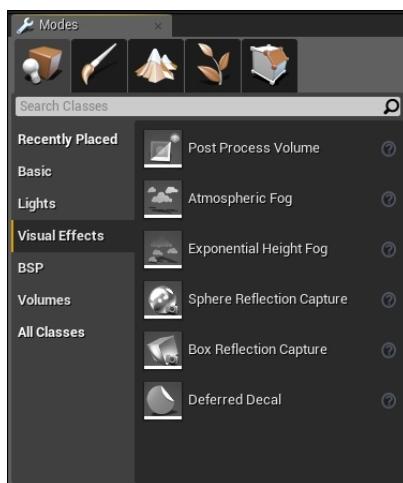


We have now placed most of the triggers we need for our game. We will add more later. And with that, we are done placing Basic classes into our level. Let us now move on to the Visual Effects class and see how it affects our game.

## Visual Effects

Since we have already covered the Light class in the previous chapter, we are going to skip it and move straight to the Visual Effects class. The Visual Effects class contains actors that affect the visuals of the game. Although not necessary components of a game, they help improve its overall quality.

Moreover, they do not require a lot of memory.



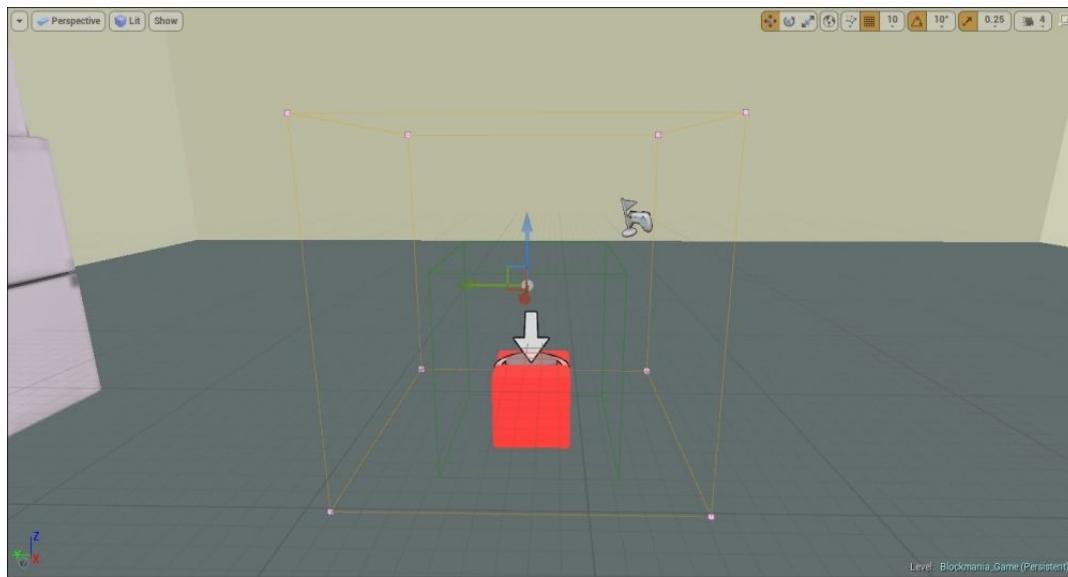
There are various actors in Visual Effects classes:

- **Post Process Volume** : This is an actor that can be used to manipulate the look and feel of the game. The effects will take place while the player is in the volume. There are many effects available. Some examples include Anti-Aliasing (removes hard edges of actors, giving them a smoother finish), Bloom (can be seen in real life when looking at a bright object against a darker background), Depth of Field (blurs objects based on their distance from a focal point), and much more. There are several effects that you can add to your game by using **Post Process Volume** ; so experiment with the volume to see everything you can do with it.

- **Atmospheric Fog** : In an outdoor level, just having a skylight is not enough to provide a realistic outdoor scene. In reality, the light coming from the sun scatters and spreads because of the earth's atmosphere. To have that effect in the game, you need to add the **Atmospheric Fog** actor to the level. You can set properties such as **Sun Multiplier** (to brighten the fog as well as the sky), **Default Brightness** (to set the brightness of the **Fog** ), **Default Light Color** (to set the color of the atmosphere), and more.
- **Exponential Height Fog** : You can use this actor to add fog and mist to your level. This is also used mostly in outdoor scenes. You can set properties such as the **Fog density**, the color of the **Fog** , the **Fog Height Falloff** (how the density of the fog decreases as we go up), and so on.
- **Sphere Reflection Capture** : This is another useful tool. The **Sphere Reflection Capture** actor takes the lighting information and provides a realistic reflective effect, giving materials a glossy finish. Metallic materials and similar rely on this actor to provide a realistic finish. In the actor's setting panel, you will see something called the **Influence Radius** , which is the volume in which the actor has influence. You can increase or decrease it. Below that are the **Brightness** settings, which you can use to set how bright you want the reflections to be. Keep in mind that if you change the lighting in the level (by moving it, changing the brightness, changing the color, and so on) or move the actors around, the **Sphere Reflection Capture** actor will not update. You will have to update it manually, which you can do with the help of the **Update Capture** button that is located above the **Influence Radius** option.
- **Box Reflection Capture** : This is similar to **Sphere Reflection Capture** . The only difference is that while the **Sphere Reflection Capture** actor has a spherical influence area, the **Box Reflection Capture** actor has a cubical area of influence, making it relatively less effective than a **Sphere Reflection Capture** actor. This actor is best used in hallways or cubical rooms. Its settings are the same as the **Sphere Reflection Capture** actor, only instead of an **Influence Radius** , it has a **Box Transition Distance** , which can be used to either increase or decrease its area of influence. Again, as with the **Sphere Reflection Capture** actor, if you change the lighting or the objects in the game, you will have to click on the **Update Captures** button to update the reflections.
- **Deferred Decal** : The **Deferred Decal** actor provides an easy and inexpensive way of adding decals onto objects. It is a great way of adding effects such as blood splatter. You can pick which material you want for the decal and place it on the level.

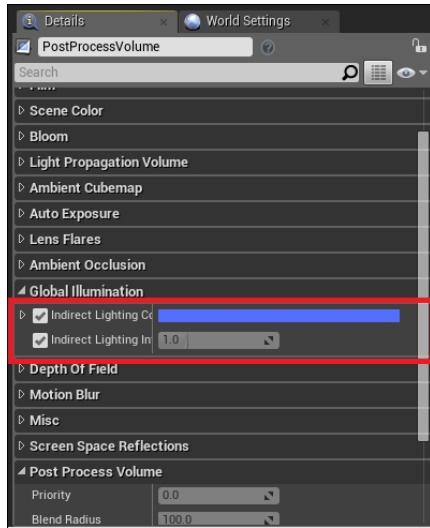
## Adding Visual Effect actors to the game – Post Process Volume

The first thing we are going to add to the level is a Post Process Volume. When the player picks up a key cube, we want to give them a visual indicator that they have picked it up. The visual indicator, in this case, is a flash on the screen. To add the actor, simply drag it from the panel and drop it on the level, over the key cube. The post process volume is represented by a light pink cube.



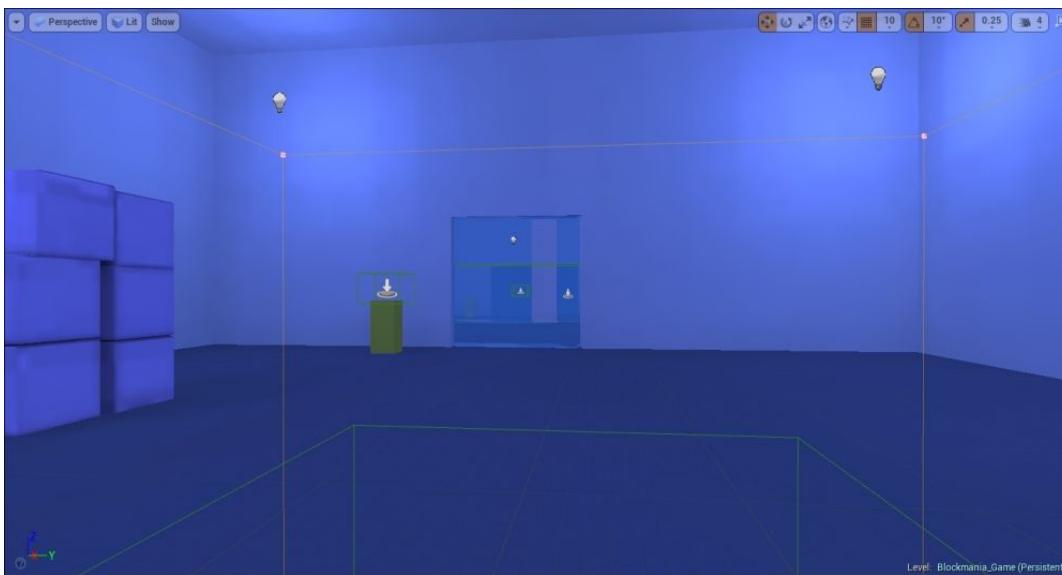
After the Post Process Volume has been placed, let us set some properties in the **Details** panel. In this panel, you will see quite a lot of options. All of them are categorized based on the type of effect they create. One thing to note is that some of the effects are not available on mobile.

We are just going to tweak the **Global Illumination** setting. In the **Details** panel, go to the **Global Illumination** section, where you will see two settings: **Indirect Lighting Color** and **Indirect Lighting Intensity** .



First, enable both effects by selecting them. Then, in the **Indirect Lighting Color** option, set the color to anything you like. In our case, we are going to set the color to blue. You can also set the **Indirect Lighting Intensity** option to anything you want, but we are just going to leave it at 1 .

Once set, if you move inside the volume, you will find everything turned blue. This is going to be our effect for when we pick up our cube:

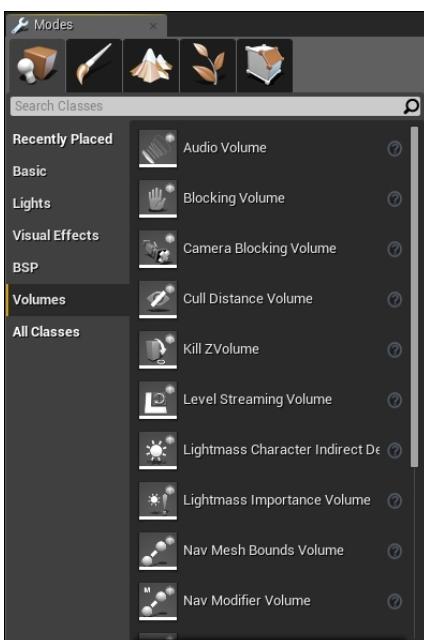


Now, if you were to test the game and walk inside the cube, everything would turn blue and remain blue unless you stepped out of the volume. We do not want that. We only want the screen to turn blue for a brief moment—when the player picks up the key cube—and then fade away. We need to change another setting. We want the volume to be disabled when the game starts and only be triggered when the player picks up the key cube. We will cover how to enable this setting in the next chapter. For now, go to the **Details** panel, and in the **Post Process Volume** section, you will find the **Enabled** option checked. Simply uncheck it; this will disable the **Post Process Volume**. If you were to check it now, you would see that the screen no longer turns blue. Finally, duplicate and place **Post Process Volumes** over all the key cubes in the level.

## Volumes

Volumes are actors that have special properties. They can be seen as invisible triggers, each doing something different (depending on the type of volume) when the player enters them. There are various types of **Volume** actors available; each has a different property upon entering it. Volumes are only visible in the **Editor** mode and not in the actual game itself. Therefore, they are usually accompanied by another actor. For example, a Volume called **Pain Causing Volume**, as the name suggests, causes the player to take damage when it is entered into. It is obvious that developers would use this volume when the player walks through something hazardous, such as fire, electricity, and so on. Therefore, the volume would be placed around it. The fire would act as a visual cue indicating that the area is unsafe to go through, and the **Pain Causing Volume** would take care of the rest (cause damage to the player).

There are different types of volumes available to users. Let us take a look at them.



- **Audio Volume** : Audio Volume allows you to control the audio within the game by tweaking its settings.
  - **Blocking Volume** : The **Blocking Volume** acts as an invisible wall, which prevents certain types of actors from going through it. You can set what types of actors can and cannot pass through in the **Details** panel.
  - **Camera Blocking Volume** : This prevents camera actors from passing through it.
  - **Cull Distance Volume** : This is an optimizing tool that does not render objects smaller or equal to a set value (set by the developer), based on their distance from the camera. This is an important tool, especially if you have a vast outdoor scene, since it will not render objects far away, therefore saving memory.
  - **Kill ZVolume** : This destroys any actor that enters it, including the player. This can be used in cases when the player falls off the edge of a cliff, into a pit, and so on.
  - **Level Streaming Volume** : This is another optimizing tool that you can use to set the part(s) of the level you want to be visible to the player. This is really useful when you have huge levels. You can use this volume to hide parts of the level that the player cannot see from his/her current location or parts that are far away from him/her.
  - **Lightmass Character Indirect Detail Volume** : This takes the lighting information and generates indirect light maps inside the volume.
  - **Lightmass Importance Volume** : Yet another optimizing tool, the Lightmass Importance Volume is used to generate lighting information within it (indirect lighting, shadows, and so on). It is advisable to place a Lightmass Importance Volume around your game level for faster light building.
  - **Nav Mesh Bounds Volume** and **Nav Modifier Volume** : The Nav Mesh Bounds Volume is used for the AI to move around in the level. When placed, the AI character will move anywhere within the volume (provided the area is accessible in the first place).
- A **Nav Modifier Volume** is used to modify **Nav Mesh Bounds Volume**. You can set it so that a certain area inside the **Nav Mesh Volume** can be blocked off and the AI character will not be able to traverse through it.
- **Pain Causing Volume** : This causes damage to any player that enters it.
  - **Physics Volume** : This is in which certain physical properties of a physics object can be altered. For example, you can enable/disable a setting called **Water Volume** : This, when enabled, simulates the character moving through a watery area, such as a swamp.
  - **Post Process Volume** : This is the same as the volume found in the **Visual Effects** section.

- **Precomputed Visibility Override Volume** : Using this volume, you can manually override the visibility of the actors in the game.
- **Precomputed Visibility Volume** : This volume has a similar function as the **Precomputed Visibility Override Volume**, the only difference being that this volume automatically stores the visibility of the actors in the game.
- **Trigger Volume** : This is the same as the trigger actors, which were discussed earlier. One of the differences is that while trigger actors come in a predefined shape, you can alter the shape of a trigger Volume using the **Edit Geometry Mode**.

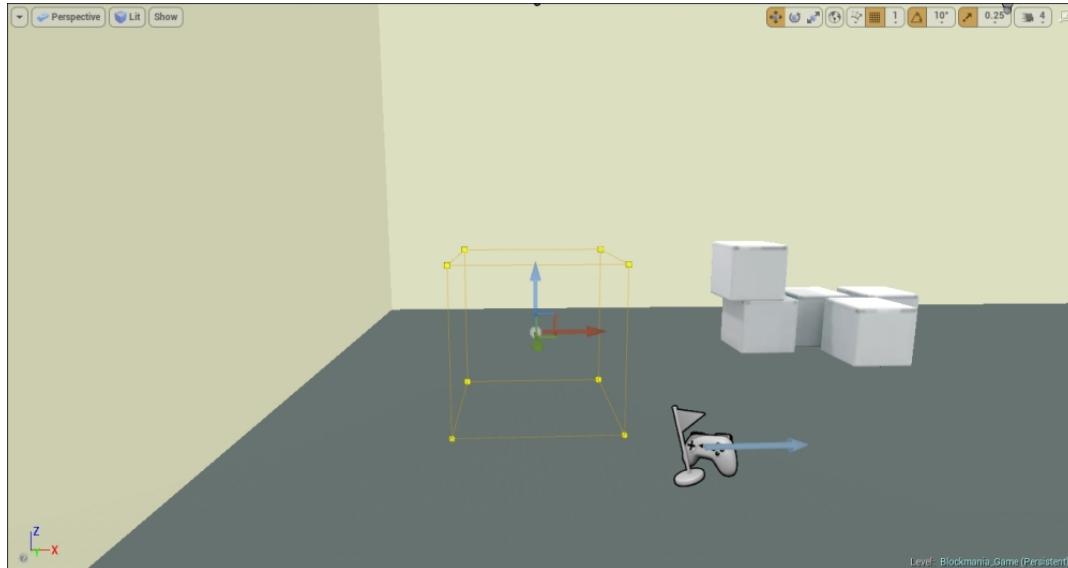
One more thing that you should know about volumes is that just like with BSP brushes, you can edit their shapes to create your own custom shaped volume. The way to edit is the same as that of BSP brushes. In the **Modes** panel, clicking on the **Edit Geometry** mode will switch to the editing mode. Once done, you can click on any volume you want in order to change and make the desired changes.

## Adding Volumes to the game

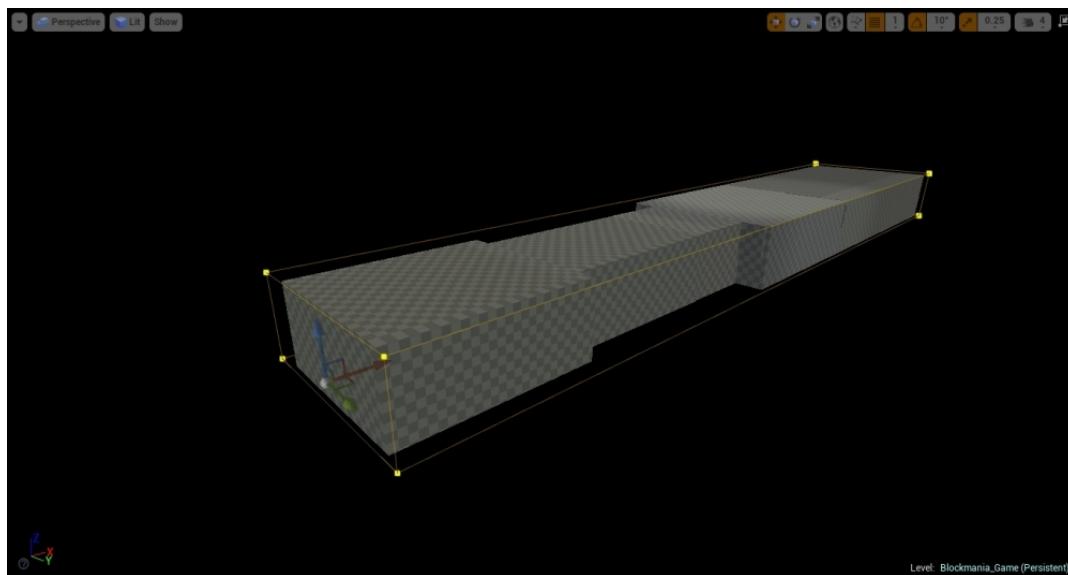
Now that we know what Volumes are and the types of Volumes available, let us go ahead and add a few of them to our level.

### Lightmass Importance Volume

It is advisable to use the Lightmass Importance Volume, since it reduces the rendering time (light building time). We are going to set its dimensions so that it encapsulates all of the four rooms in the game. Simply drag the actor from the **Modes** panel and drop it on the screen.



The Lightmass Importance Volume is represented by a yellow colored cube. Set its dimensions such that it encapsulates all of the four rooms in the game.



If you were to build the lighting now, you would notice that the building process takes relatively less time, and the areas outside the rooms are now completely dark. The engine now focuses mainly on what is inside the volume to produce high-quality lighting, and anything outside of it will be of lower quality, in terms of lighting.

### Nav Mesh Bounds Volume

Next, add the **Nav Mesh Bounds Volume**. If you wish to have an AI character in your game, a **Nav Mesh Bounds Volume** is an important component. As mentioned previously, this volume is basically one within which the AI character moves and interacts with the world. When designing your game, you should know where all the AI-controlled characters will move around in the game and what objects they can interact with, and then place your volume accordingly.

Grab the **Nav Mesh Bounds Volume** and drag it into the screen. We are going to require the volume in room 3 and room 4. The volume is represented by a gray cube.

### Room3

Place the **Nav Mesh Bounds Volume** on and near the pit, opposite the pedestals with the switches.

When placing this volume, it is advisable to turn on **Navigation** in the **Show** menu. In the **Viewport Toolbar**, click on **Show** and check the **Navigation option** to have it appear in the Viewport. Once toggled, you will see that any surface, actor, or any other physical object (or parts of them) inside the volume will have a bright green color on them. This is a visual indicator of where the AI will be active. It will ignore anything outside of itself.



As mentioned earlier, any part of a surface or object that the volume overlaps with will be green. Let us move on to room 4.

#### Room 4

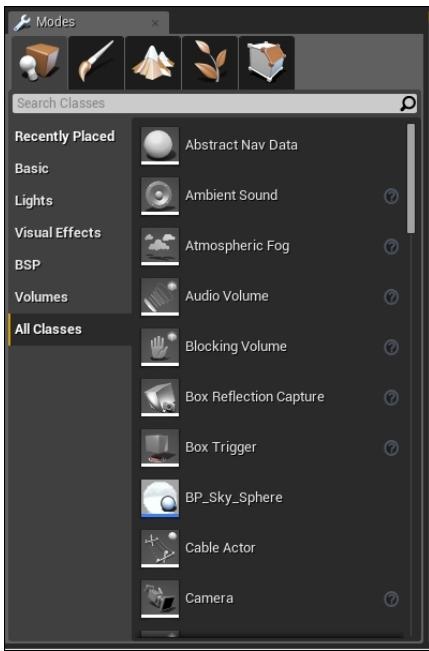
In room 4, we are going to need a big Nav Mesh Bounds Volume, one that covers almost the whole room. Place the Volume and move its sides using the Geometry Editing mode. After setting the dimensions, it should look something like the following screenshot:



Our AI character will now know where to move about in the level (The area highlighted in green).

## All Classes

Lastly, we have the **All Classes** section, in which all of the classes and volumes we have discussed so far are listed. Additionally, there are certain actors that are not displayed in the previous three sections and are only accessible via the **All Classes** section.



There are many such actors, some of which are beyond the scope of this guide. We will only talk about some of the actors that are in the **All Classes** section:

- Ambient Sound Actor** : This is an actor that you can use to play audio or sound effects in your game. It also emulates real-world sounds, in that the closer you are to the source, the louder the sound will get, and conversely, the further you are from the source, the fainter it will get.
- Camera** : A **Camera** actor is one through which you see the virtual world. By default, your character class already has a camera, but if you want to import your own character, a **Camera** actor is an essential component. It is also used in cutscenes, and so on.
- Default Pawn** : A **Default Pawn** actor is a simple spherical actor with built-in flying mechanics, static mesh, spherical collision, and so on, which you can use for simple AI.
- Landscape** : A different way of switching to the landscape mode can be found here. If you drag the landscape actor from here and drop it on the screen, the mode will change to Landscape mode, wherein you can place your terrain.
- Level Bounds** : A **Level Bounds** actor, when placed in the level, automatically updates and resizes to encapsulate the entire world. It can be used to calculate the size of the level and the world. Just keep in mind that if your level has a skybox or a skydome, the volume will resize and include that as well.
- Matinee Actor** : Matinee is a powerful tool used to create cinematics, set pieces, and so on. There are two ways of adding a Matinee actor. The first way is through the Viewport Toolbar. Simply click on **Matinee**, and select **Add Matinee** when the menu opens. The other way is with this option. You can drag the actor from the **Modes** panel and drop it on the level.
- Nav Link Proxy** : The **Nav Link Proxy** actor is used if an AI character has to perform actions, such as dropping off or jumping off a ledge, jumping between gaps, and so on. It allows the AI character to leave the Nav Mesh temporarily. (We will return to this in the next chapter, when we talk about AI.)
- Target Point** : **Target Point** actors are used to get the coordinates of a particular point in the level. They can also be used for AI characters. If you want to have your AI character follow a particular path, or have it patrol a certain area, you should use target points. Also, keep in mind that Target Points should be placed inside the Nav Mesh Bounds Volume, otherwise the AI actor will ignore them, even if you have it scripted to move towards the said Target Point. Target points can also be used if you want your character to teleport to a particular destination.
- Text Render** : The **Text Render** actor, as the name might suggest, is used to render text in the game. If you want your game to have popup text (for tutorials or something similar), this is what you should use. You can import your own font and create your own text render.
- Class Blueprints** : Although not a single specific actor per se, all of the Class Blueprints that you create in your project are displayed here. What a Class Blueprint exactly is will be discussed in the next chapter. For now, the only thing you need to know and remember is that all of the classes you create are also accessible from here.

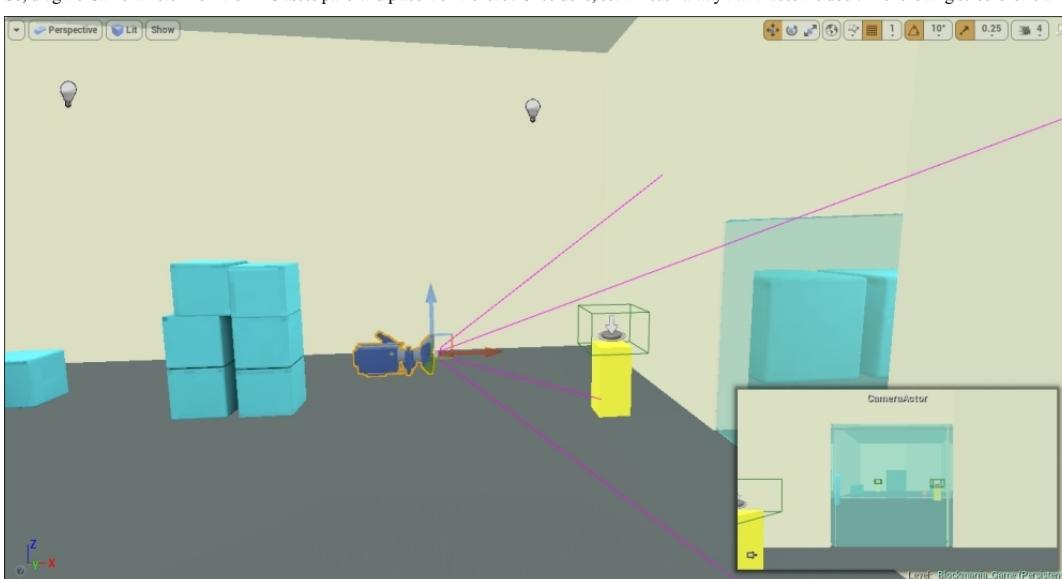
## Adding actors from All Classes

Now that we have discussed some of the actors found in All Classes, let us go ahead and add a few of them to our game.

### Camera

The first thing we are going to add is a Camera Actor. As a part of the **tutorial**, we are going to have a small cut-scene in which we will show the door opening when the player has placed the key cube on the pedestal (so that the player understands what the key cube is for and what it does).

So, drag the **CameraActor** from the All Classes panel and place it on the level. Once done, set it in such a way that it faces the door. The following screenshot is an example of where you can place your camera:



The small window on the bottom-right corner of the screen shows the view from the second camera, which you can use to properly adjust its position until you can see the door clearly. Any time you select any **CameraActor**, the window opens, showing you the view from the selected **CameraActor**. We are going to need this when we create our cut-scene in later chapters.

## Matinee actors

We are also going to place several Matinee actors. Instead of moving our doors and platforms for our AI characters via scripting (or in this case, blueprint), we are going to implement said features with the help of Matinee. We will animate the opening of the door and the drawing of the animation in **Unreal Matinee** and call it whenever the player interacts with the appropriate trigger.

To add a Matinee actor, drag it from the **Modes** panel and place it near the door.

### Note

Although where you place the Matinee actor does not matter, to prevent confusion and to make things more convenient, you should place it near the actor(s) you want to move or edit using Matinee.



Just place the Matinee actors like this for all the doors. We still have to place more Matinee actors, but we will do so later.

## Target Point

Next up, we are going to place a couple of Target Point actors. As mentioned previously, Target Points are useful for moving AI characters. We will have a fairly simple AI, one which moves in a specified path, stops when it hits a switch, and respawns if it falls in the pit.

### Room 3

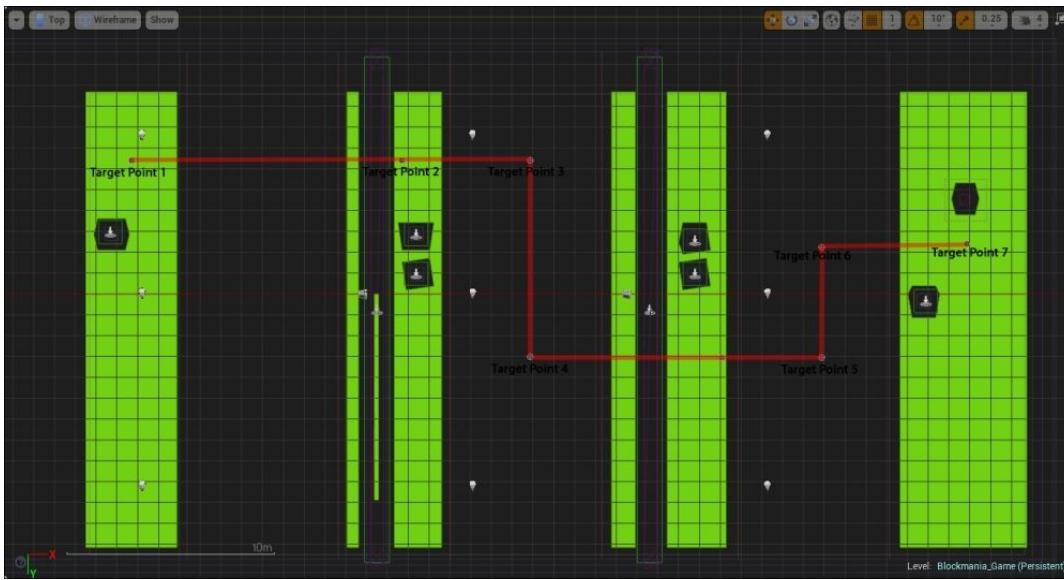
In room 3, place two target points: one where the AI character will be initially and the other where the switch will be. When the AI character falls into the pit, it will respawn at the first Target Point. When the player activates it, the AI character will go to the second Target Point.



The actors are depicted by a small target icon. Let us now place them in room 4.

### Room 4

In room 4, we are going to place a couple of Target Point actors, since the AI character will take a more nonlinear route.



The preceding screenshot is the top view of room 4. The player will start from the left-hand side and will have to direct the AI character towards the right. Also, as you can see, we have placed seven Target Point actors. The path the AI character will take is shown by the red line. With that done, we have fully placed our Target Point actors. With that, we have placed all of the classes, volumes, and other actors in our level.

## Summary

In this chapter, we looked at some of the actors present in the Modes panel that are vital to the functionality of the game, and some that enhance the overall experience.

Apart from talking about them, we also placed some of them in our level. We are now close to making our game. In the next chapter, we are going to talk about Blueprints—probably the most important topic we will be covering in our guide. Without them, there would be no interactivity in the game. So, let us start scripting in the next chapter.

## Chapter 5. Scripting with Blueprints

We now come to one of the most important aspects of the game: interactivity. Without it, our game will just be an environment that the player can move around in. Those who have used UDK before may already be familiar with the concept of visual scripting. UDK had what is called Kismet, a powerful visual scripting tool. A really attractive feature of this tool is that anyone can use it without any prior programming knowledge. All you need to know is the logic behind the event you wish to implement. You can create a full game without even writing a single line of code.

In Unreal 4, we have Blueprints, which is sort of an upgrade to Kismet. The basic setup is the same: you have various nodes and expressions, which you can use to script in-game events, actions, and so on. The interface is simple to understand, easy to use, and yet extremely powerful. Once you get the hang of it, you can create complex sequences and events.

However, even though Blueprint is an easy-to-learn and a great tool, it is still limited in terms of what it can do. In that respect, C++ is much more versatile and flexible than Blueprint. C++ is great for implementing complex interactions and mechanics, which Blueprint may or may not offer. Another difference between Blueprint and C++ is that Blueprint is much slower to execute than C++ code, but this is only noticeable if you have a lot of Blueprint script in your game. There are several other differences between Blueprint and C++; but it all boils down to personal preference about the features, mechanics, and functionalities you want in your game. You can choose to use either one or both.

In this chapter, we will be covering the following topics:

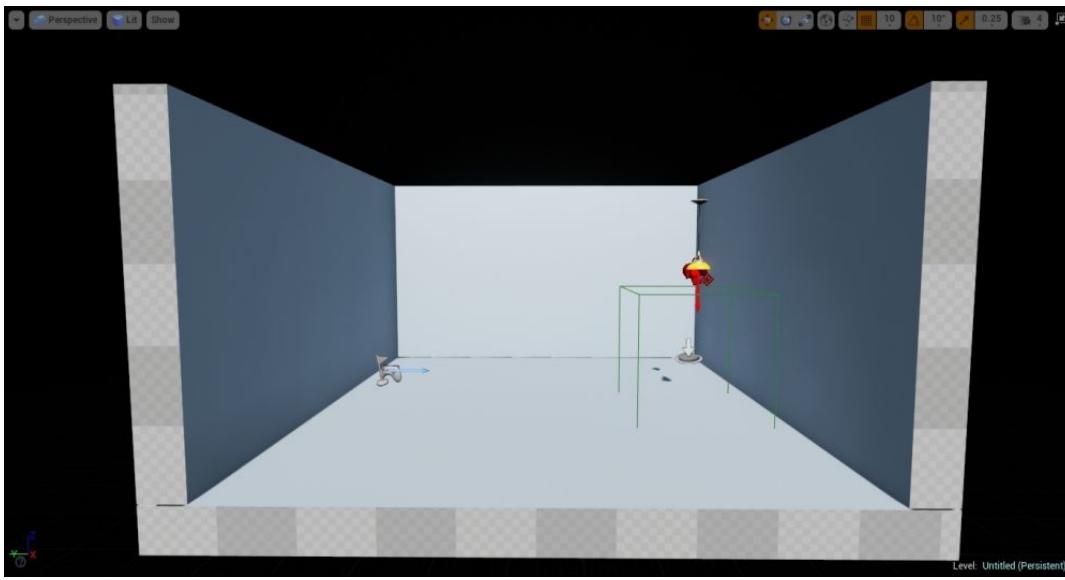
- What Blueprints are and how they work
- What Level Blueprints are and how to script using Level Blueprints
- Level Blueprint user interface
- What a Blueprint class is and how to use it in the game
- How to Script basic AI

## How Blueprint works

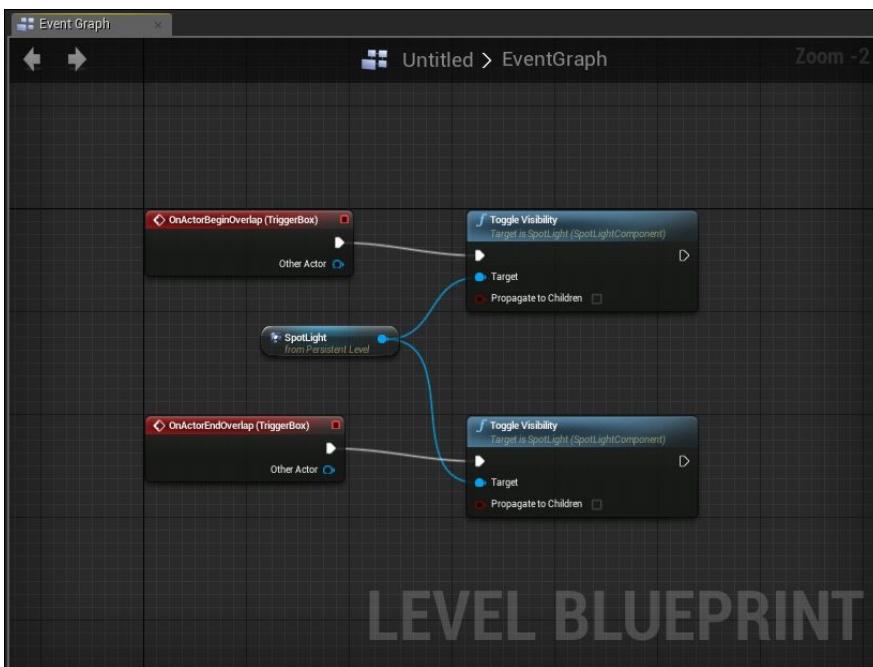
Scripting in Blueprint is similar to creating a flowchart. At your disposal are various nodes, which you can connect to create an action sequence. In order to properly script in Blueprint, you should first know the logic behind the desired sequence. For example, say you want to implement a lamp that is switched on when you go near it and switched off when you move away from it. In such a case, you would first place a trigger around the lamp. Then, the logic behind this would be:

1. If the player is overlapping the trigger, the light will be switched on.
2. If the player is not overlapping the trigger, the light will be switched off.

Now that we have figured out how to carry out this action, we can proceed to set up our nodes. Say your setup looks something like this:



When the character walks into the box trigger, the ceiling light (the spotlight) is switched on. The Blueprint to toggle the light on/off will look something like this:



There are various types of nodes that you should know about:

- **Event Nodes** : The nodes with the red bar are event nodes. These are activated when the corresponding event takes place. They usually have a rightward facing arrow at the top left corner.  
You also have Input Event Nodes, which fire off when the player gives the corresponding input (for example, firing a weapon when the player presses the left mouse button).
- **Function Nodes** : The nodes with the blue top are function nodes. They usually have a function symbol **f** at the top left corner, as you can see in the preceding screenshot. These nodes perform a specific action on an actor or player.

There are two types of function nodes: those that act upon an actor, and those that return a specific value. These types of nodes have green tops (not displayed in the screenshot). This includes things such as returning an actor's location in the world, returning the actor's velocity, and so on.

- **Reference or Variable Nodes** : The node in the center of the screenshot is a variable node (or reference). When you want a function node to act upon a specific actor in the scene, or if you want to "get" some of its property or properties, you need to create its reference in the Level Blueprint. The same goes with variable nodes. It should come as no surprise that variables play a very important role when scripting or coding. So naturally, you will have to create variables when you are using Blueprint.

When scripting using Blueprint, keep in mind that the fewer nodes you have, the better the performance. It also makes your workspace more organized and easier to read.

When a particular event occurs, the corresponding event node fires off a pulse to whichever node it is attached. In our case, when the player overlaps the trigger, it will activate the Spotlight's **Toggle Visibility** function (since the light is off by default, it will toggle it on). And when the player stops overlapping the trigger (walks out of the trigger), it will fire the **Toggle Visibility** function once again (the light will be switched off).



The preceding screenshot demonstrates how blueprints function when an event has taken place. When the player has overlapped the trigger (top left), the corresponding event fires off a pulse to the **Toggle Visibility** function, which toggles the light on. When the player moves out of the trigger, or stops overlapping, the corresponding event again fires a pulse to the **Toggle Visibility** function node, which toggles the light off.

You can actually see the pulse being fired whenever a node is activated. This makes debugging very easy, as you can see which node is causing the problem and fix it accordingly.

Apart from knowing the logic, you should also know which nodes are available and what they can be used to do. This comes with time and practice.

There are two types of Blueprints in Unreal 4: Level Blueprint and Blueprint class.

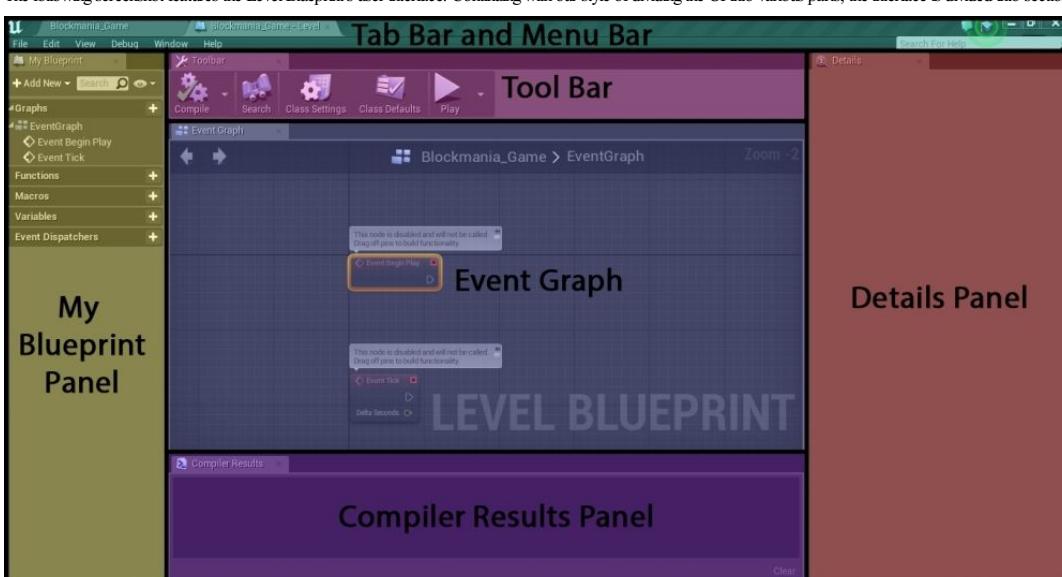
Level Blueprints is to UE4 what Kismet was to UE3. Each level or map that you create in your project file will have its own unique Level Blueprint. You can control everything level related using Level Blueprints, such as playing cutscenes, editing the properties of actors (visibility, location, and so on) in your level, and so on.

Blueprint classes, on the other hand, are special actors that contain various components as well as scripts. These components include things such as static or skeletal meshes, camera, collision component, triggers, and audio components. By using scripts, you can set their properties and determine how they interact with the world. Blueprint classes are not unique to a particular level; therefore, you can use them in any map or level you have made in your project. You can export them to other projects as well.

Let us move on to the Level Blueprint's user interface.

## The Level Blueprint user interface

The following screenshot features the Level Blueprint's user interface. Continuing with our style of dividing the UI into various parts, the interface is divided into sections, which we will go through individually.



### The tab and menu bars

The tab bar is the same as that seen in all the other windows. Just like in the case of web browsers, you can see which windows are open, swap between them, and close any window you want from there.

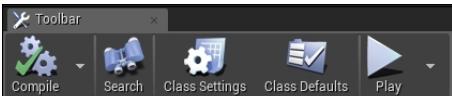


The menu bar is where you can access all the general commands and actions that you would need.

- **File** : In this menu, you can save your level, open any asset in your Content Browser, compile your blueprint, enable source control, and so on.
- **Edit** : From this menu, you can perform actions such as undoing the last action, redoing it, searching for a specific node or expression, and so on.
- **View** : Here, you can hide/unhide unused pins and unconnected pins (through which you connect nodes), zoom in, and zoom out in the event graph.
- **Debug** : If there is a problem in your Blueprint sequence that you cannot figure out, you can use the options available in the Debug menu to find and resolve them. These include adding breaking points, watching the value of a particular variable at a particular point, and so on.
- **Window** : Here, you can set what windows you want to be visible and what windows you do not want to be visible. You can customize the layout and save it from the Window menu.
- **Help** : From here, you can access Epic's official documentation regarding Blueprints. You can also go to the Wiki page, the forums, and the Answer Hub from here.

## The toolbar

Toolbar has the most commonly used actions:

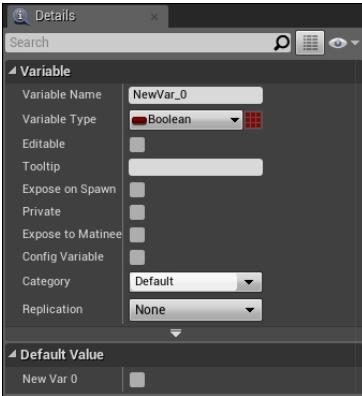


The most commonly used actions are explained as follows:

- **Compile** : Whenever you add, remove, or edit any node in the event graph, be sure to click on the **Compile** button. It compiles all the nodes and sequences, and if there is any error or warning, it will notify you in the **Compiler Results** panel, which you can then fix. Also, if you have a variable node, in order to set its default value, you first need to compile.
- **Search** : When you have large and complex sequences with several connected nodes, trying to find a specific node or variable can be a tedious and time-consuming task. In order to avoid that, you can click on the **Search** button, and type in the name of whatever it is you wish to find. The results will, by default, be displayed at the bottom of the screen, where the **Compiler Results** panel is (it will open a new tab called **Find Results** ).
- **Class Settings** : Clicking on this will open up the Blueprint settings in the **Details panel**, where you can set certain options such as adding a description, category, and so on.
- **Class Defaults** : Here, you can set the default or initial values of your Blueprint class.
- **Play** : Similar to the **Play** button in the **Viewport** toolbar, this opens a new window where you can test your game. While the game is running, clicking the *Esc* button will close the game and return to the Editor. You may notice a small downward-facing arrow next to the button. This opens a menu where you can set options such as how you want to preview your game, where the player should start, and so on.

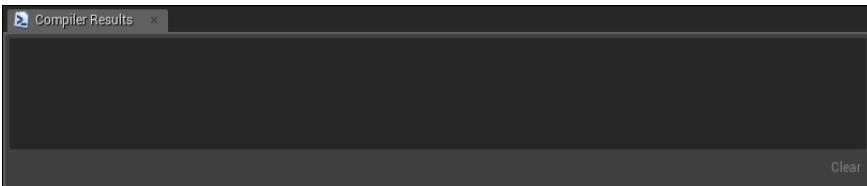
## The Details panel

In the **Details** panel, you can set the properties of various nodes and variables. It offers settings such as the type of variable you want (Boolean, float, integer, and so on), the name of the variable, and more.



## The Compiler Results panel

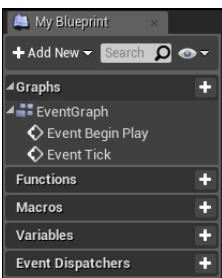
Anyone who has any programming knowledge will know what compilation is. It occurs when the code that you have written is converted from the language you wrote the code in into machine language that the computer understands, so that it can be executed. This is usually handled by a compiler. (If you do not see the **Compiler Results** panel, in Level Blueprint, go to Window, and click on **Compiler Results** .)



In the **Compiler Results** panel, you can see the output log of the compiler. How long it took to compile, any errors found during compilations, any warnings, and so on, all are displayed here.

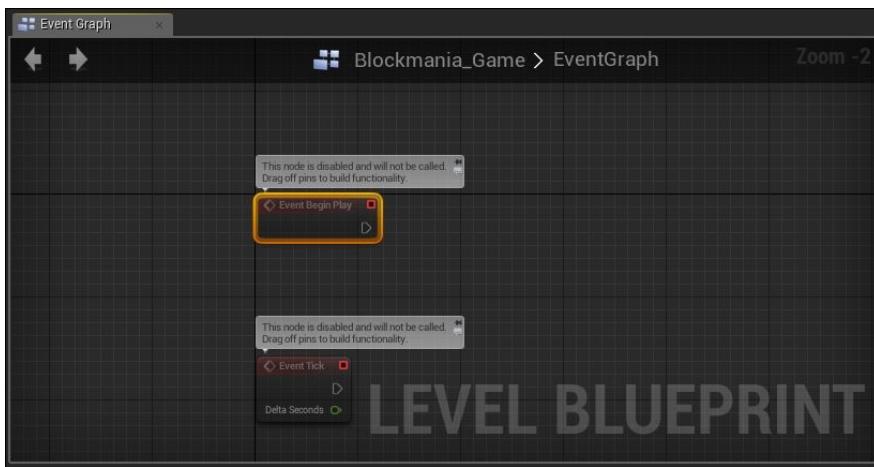
## My Blueprint panel

In this panel, you can see a list of all of the events, variables, and event dispatchers that you have created. Whenever you create an event node, it is displayed here, under **EventGraph**. Apart from that, you can also create various functions, macros, variables, and so on. To do so, you can click on the **Add New** button and select whatever it is you want to create, or by clicking on the + button in front of the names.



## The Event Graph

Located at the center of the screen, the **Event Graph** (also referred to as **Graph Editor**) is where you set up your nodes and sequences. By default, there are two event nodes already set up, namely **Event Begin Play**, which is activated when the game begins, and **Event Tick**, which is activated at every frame. These two events do not require any trigger to be activated.



At the top, you can see the tab. Below the tab, at the top-left corner, are two arrows. You can use them to switch between graphs. At the center, you can see the hierarchy and the blueprint structure. At the extreme right, you can see the zoom ratio—in other words, how much you have zoomed in or zoomed out.

The following table lists the controls of **Event Graph** that you should know and memorize:

Control	Action
Left-click on mouse	Selects nodes
Left-click + drag	Creates selection box
Right-click	Opens the Action Menu
Right-click + drag	Pans the Graph Editor
Scroll wheel up	Zooms in
Scroll wheel down	Zooms out
C	Creates a comment box around selected node(s)

#### Note

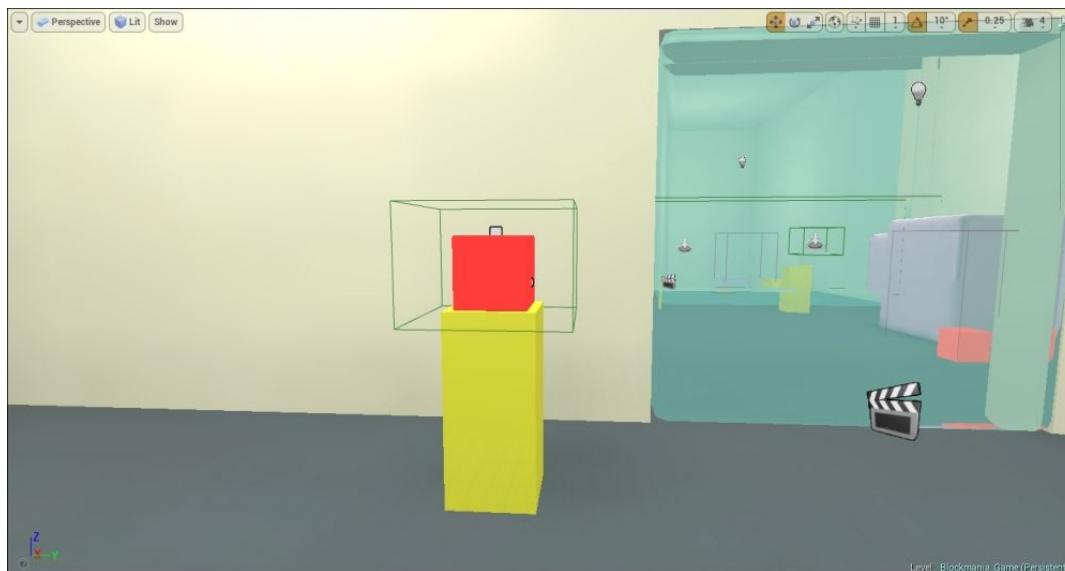
Even though it is visual scripting, it can still get pretty messy when you script using Blueprints. So, to avoid confusion and keep everything organized, it is advisable to create comment boxes around your nodes.

## Using Level Blueprint in the game

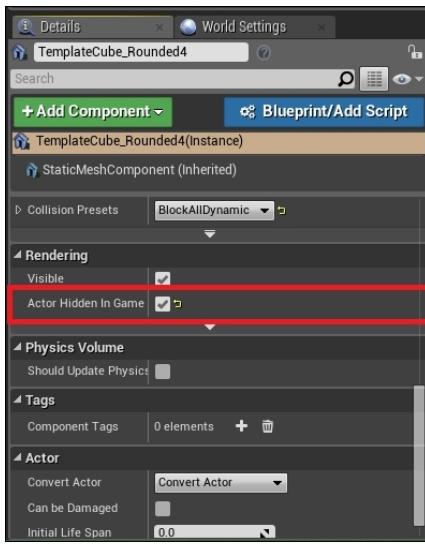
With all the basics out of the way, we can begin scripting our game. One thing you should know is that we might add more triggers and actors as we go along.

### Key cube pickup and placement

The first thing that we are going to script is the player picking up the key cube. Now, when the player is close enough to the key cube and taps on it on their screen, they will be able to pick up the cube. In our case, to give the illusion that the player has picked up the key, we are going to destroy the actor when the player taps on it on the screen. Also, we will place a replica of the key cube on top of the pedestal at the very beginning and keep it hidden at the start of the game. When the player has picked up the cube and is close enough to the pedestal, tapping on the screen will unhide the cube from the game, giving the impression that the character has placed the key on the pedestal. So, let's set that up. Firstly, with the key cube selected, hold down the *Alt* button, and with the help of the Transform tool, drag out a duplicate. Place this duplicate on top of the pedestal.

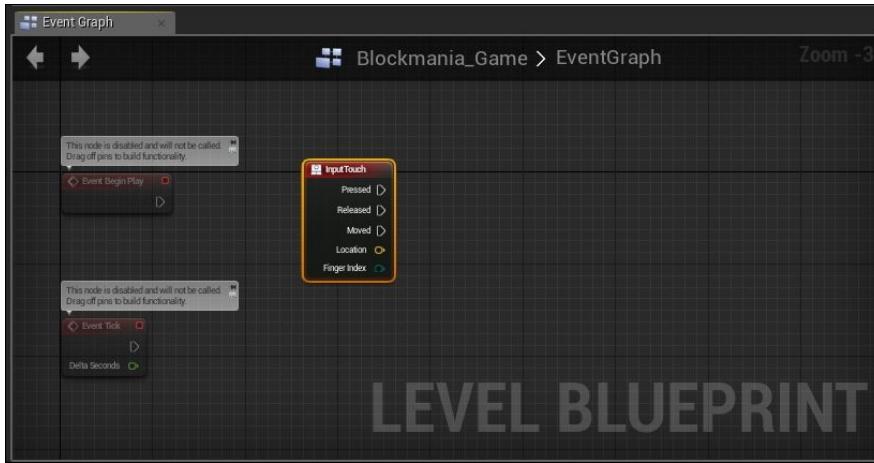


In its **Details** panel, under the **Rendering** section, tick the **Actor Hidden in Game** option. Doing so will hide the game from view during runtime.



Now, let's open the Level Blueprint. To do this, click on **Blueprints** in the **Viewport** toolbar and select **Open Level Blueprint**.

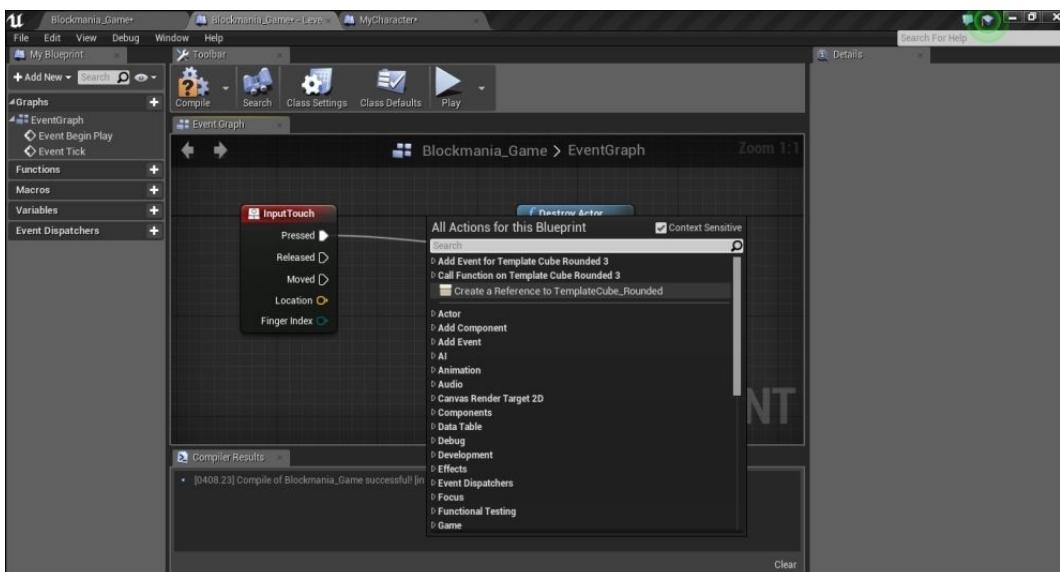
Now, the event in this case would be the player tapping on the screen, or in technical terms, providing a touch input. So in the **Event Graph** window, right-click to open the **Actions** menu and type in **Touch**. This should find and display the **Touch** event node. Click on it to add it to the **Event Graph**. You may find various types of **Touch** nodes—the one you need is the node that simply says **Touch**.



the **InputTouch** event node, we are only concerned with the **Pressed** output pin. The **Pressed** pin will be activated when the player presses anywhere on the screen.

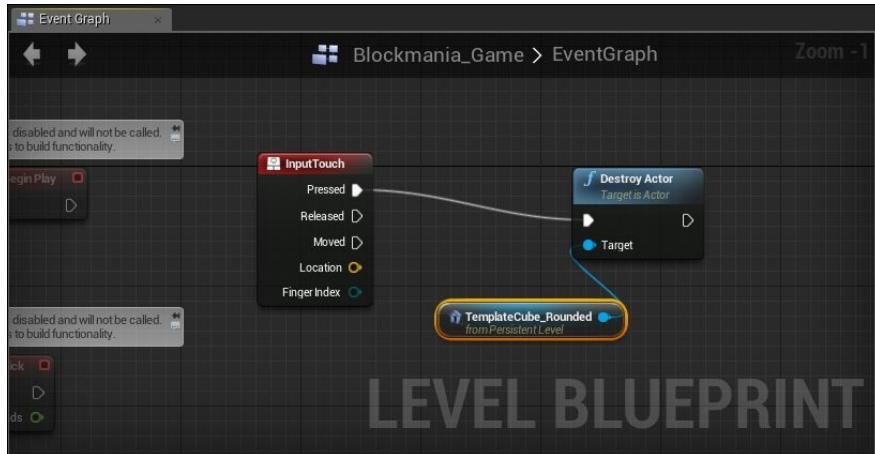
To remove the box from the scene, we are going to use the **Destroy Actor** function node. So, right-click anywhere in the **Graph Editor**, type in **Destroy Actor**, and click on the result (you can also find it manually find under the **Utility** section). With both nodes present, connect the **Pressed** output pin to the **Destroy Actor** input pin. Now, the function does not know itself which actor it has to destroy. We have to specify to it which actor we want to get rid of. In Blueprint terms, we have to create a reference to the actor we wish to apply the function to. So, in the **Viewport**, select the key cube. Then, in the **Graph Editor**, right-click and select **Create a Reference to TemplateCube\_Rounded**.

If your object has a different name, instead of **TemplateCube\_Rounded**, you will see the name of the actor.



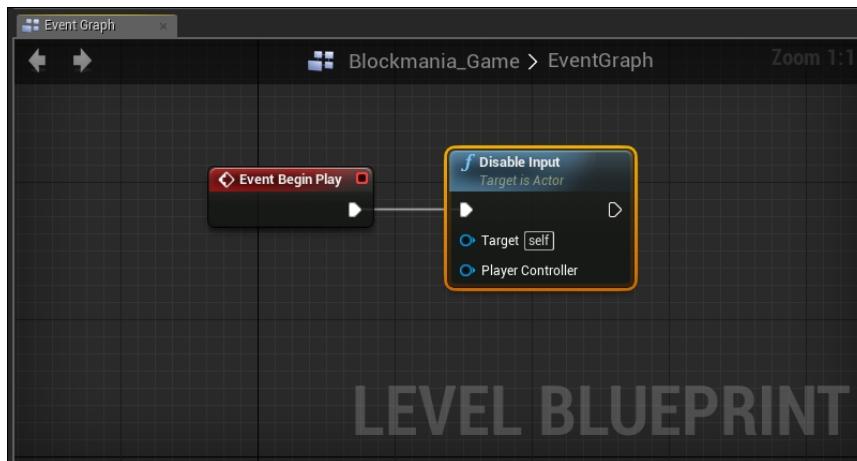
Doing so will create a reference node with the name of the actor written on it. Connect this to the **Destroy Actor** target pin.

Another way of doing this is to drag the reference node's output pin and release it anywhere in the **Event Graph**. Doing so will open a menu, from where you can select the **Destroy Actor** node. Once created, the reference node will automatically be connected to it. The setup so far should look like this:



However, there is a problem here. If you were to test your game, you would notice that the cube is destroyed when you tap on the screen, no matter where you are. We only want the cube to be destroyed when the player is close enough to it.

To resolve this issue, the first thing we are going to do is render the input disabled at the start of the game. As you may remember, the event node that is activated when the game begins, **Event Begin Play**, is already present in the **Graph Editor**. To this, we are going to attach a **Disable Input** node. Right-click and find the node, and connect it to the **Event Begin Play**.

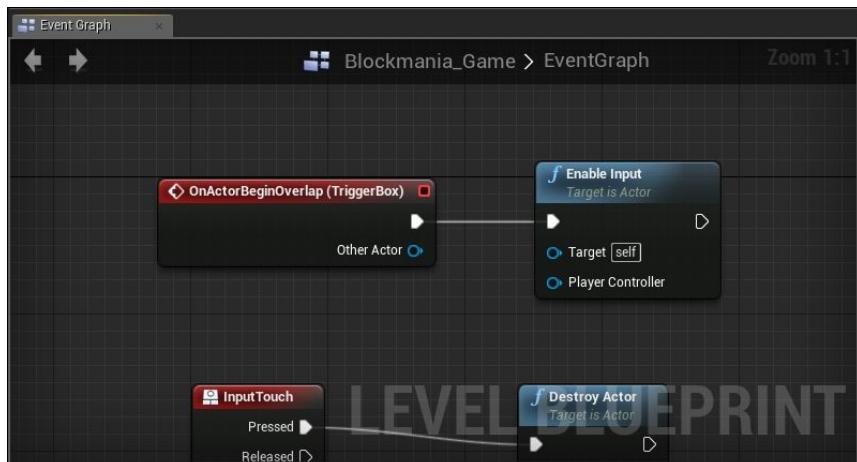


## Note

Although not required right now, if your game has several players or controllers, you will have to specify which player's controller you wish to disable. This can be done by first creating a **Get Player Controller** node and attaching it to **Player Controller** in the **Disable Input** node. By default, **Player Index** of the character you play as is 0.

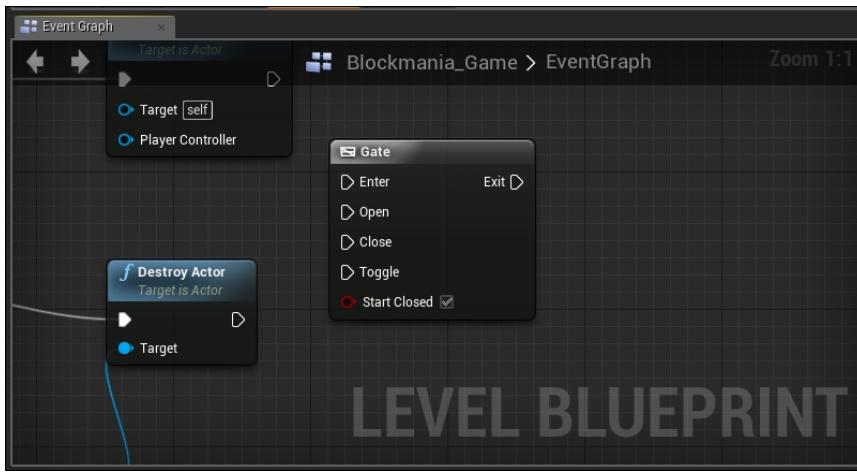
If you test your game now, when you tap on the screen, the key cube is not destroyed (although the character will start shooting projectiles again). Now, going back to our sequence, since we want the player to be only able to pick up the cube when they are at a certain distance, we are going to enable their input when they are overlapping with the trigger we had placed around the key cube.

With the trigger selected, create an **EventBeginOverlap** node. After creating it, create an **Enable Input** node and attach it to the **EventBeginOverlap** node's output pin.



We are now almost done with our setup. If you were to test it now, you would find that it functions the way we intended it to. However, this is still not complete. For one, if you have more than one key cube in your room, both of them will disappear when you click on one. We do not want that; we want only the key cube the player picks up to disappear. To do this, we are going to use a **Gate** node. A **Gate** node is used to control pulses going through the node based on certain conditions that you can set. For example, you can set it to **Open** when a certain event has taken place, and so on. For instance, when the player has overlapped with the trigger, it will open the gate, allowing pulses to go through it.

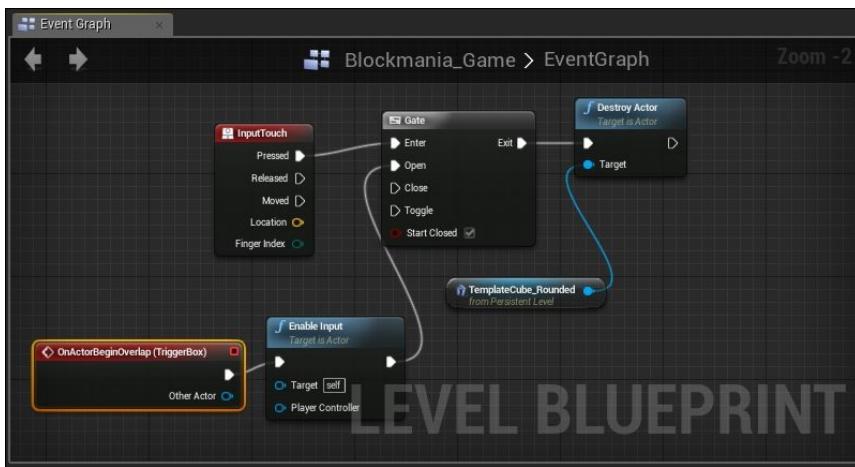
Right-click anywhere in the **Event Graph** and type **Gate**. Then, click on it to place it. You can also find it under **Utilities | Flow Control | Gate**.



## LEVEL BLUEPRINT

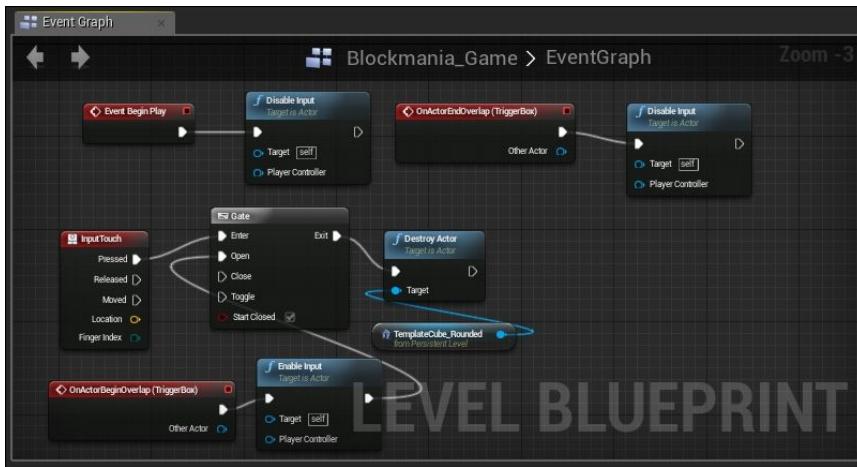
The event you wish to control is connected to the **Gate** node's **Enter** input. The rest of the events are used to control the flow. Connecting the **Close** input to an event will stop anything from passing through the **Gate** when the event has occurred. Similarly, connecting the **Open** input to an event or function will allow pulses to pass through the node. The **Toggle** node will either open the **Gate** node if it was initially closed or vice versa. Finally, you have a **Start Closed** pin, which sets the initial state of the **Gate** node. If checked, it will be closed initially, and when unchecked, it will be open initially.

First, disconnect the **Pressed** pin from the **InputTouch** node and connect it, instead, to the **Gate** node's **Enter** pin. Then, connect the **Exit** pin to the **Destroy Actor** function's **Input** pin. Once that is done, we need an event that will open the **Gate** node. This event would be when the player overlaps with the trigger. So, take the output pin of the **Enable Input** node and connect it to the **Open** input pin of the **Gate** node (make sure that **Start Closed** is checked). Your setup should look like this:



## LEVEL BLUEPRINT

Now, when the player steps out of the trigger, that is, stops overlapping with the trigger, we again want to disable the input. So, with the trigger box selected, create an **OnActorEndOverlap** node. You can type it in the search bar or find it under **Add Event** for <name of the actor> | **Collision** | **Add On Actor End Overlap**. Also, create a **Disable Input** node. Again, you can type it in or find under **Input** | **Disable Input**. With everything set up, here is what we should end up with:

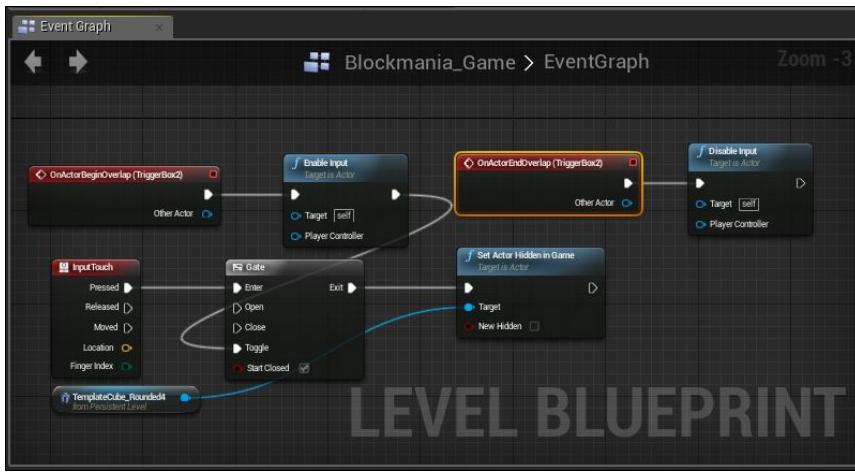


## LEVEL BLUEPRINT

We now have a simple pickup action. But we have more to do before we are finished. We have only just scripted the picking up of the cube. We still have to script in for when the player places the key cube. This is straightforward.

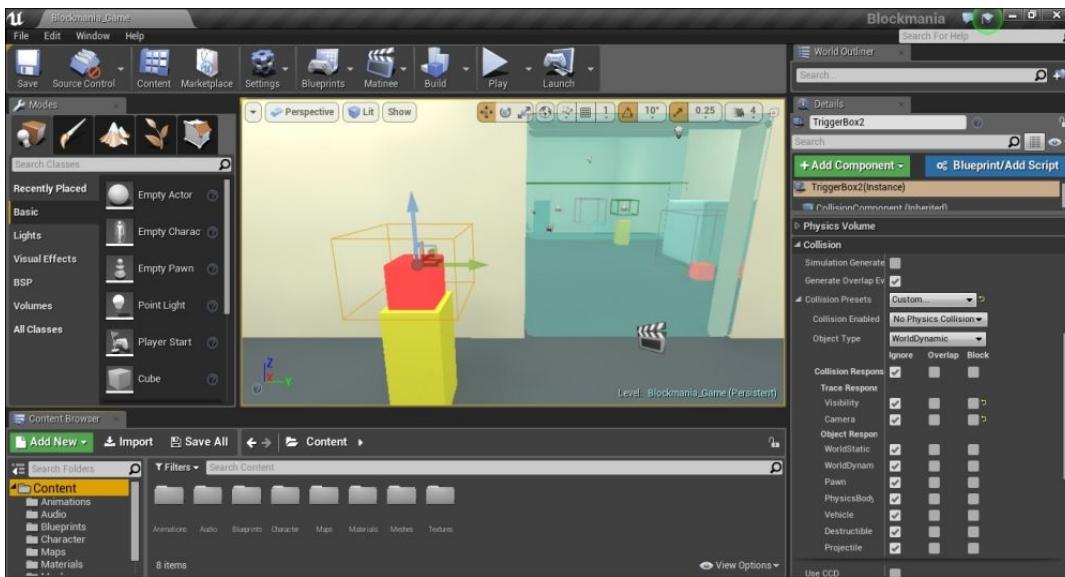
The first thing we need to do is add an overlap event that would enable the player's input. Hook them up the same way as you did with the previous trigger. Next, we are going to add another Touch event node. To this, we are going to connect a **Set Actor Hidden in Game** node. First, select the other key cube (the one that is hidden in the game), then right-click anywhere in the **Graph Editor**, and find it by typing its name in the search bar. You can also find it in **Call Function** on <name of the selected actor> | **Rendering** | **Set Actor Hidden in Game**. You will notice that when you create this node, a reference node for the key cube will automatically be created with it, and connect to the **Target** input. Next, connect the **Pressed** output pin to the **Set Actor Hidden in Game** node's **input** pin. After having done that, add a **Gate** node with its **Toggle** input connected to the output of the **Enable Output** node.

Finally, create an **OnActorEndOverlap** node and connect a **Disable Input** node to it, just like with the previous trigger.



If you were to test it out, you would find it working perfectly. However, there is a problem here: we have not set a condition as to when the player can place or unhide the key cube on the pedestal. In other words, if the player simply walks up to the pedestal, without picking up the first key cube, they would still be able to unhide the other key cube, since that would mean that the player can progress through the room without picking up the key cube.

To fix that, we are first going to change a few properties of the trigger on the pedestal. We need to first turn off the trigger's collision. We will have to set it up so that initially, the trigger on the pedestal ignores all types of overlap events—in other words, toggle it off. To do so, select the trigger on the pedestal, and in the Details panel, go to the Collision section. Under this section, you will see an option called **Collision Presets**. By default, it will be set to **Trigger**. Click on the bar to open the preset menu. From here, select **Custom**.

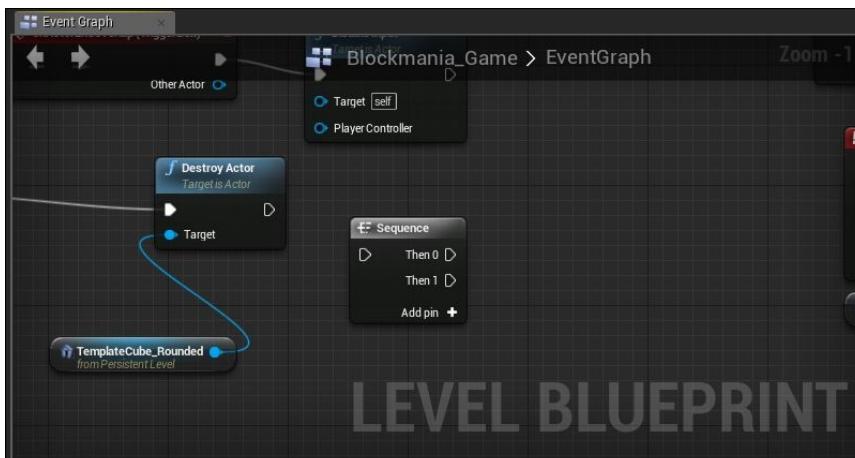


Next, click on the small triangle next to **Collision Presets** in order to open a list of the trigger's collision responses against different types of actors. There are three general responses:

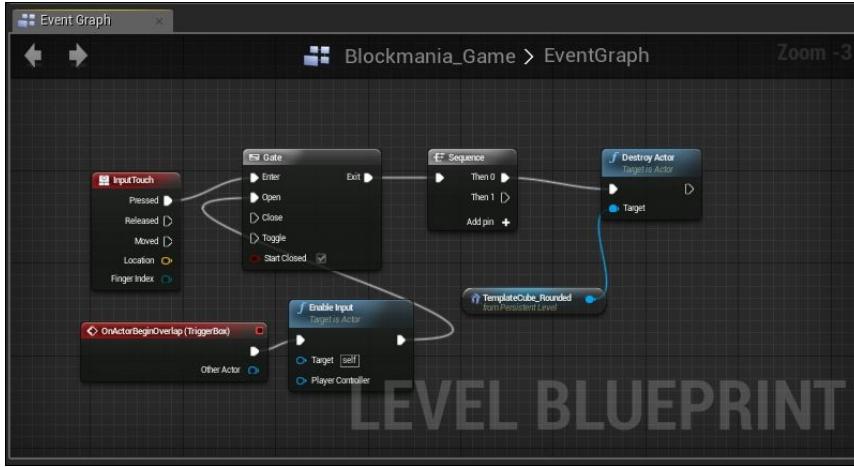
- Ignore**: The trigger will not register any response to the collision. The actor(s) the trigger is set to ignore will neither be blocked, nor be registered by it.
- Overlap**: The actor(s) the trigger is set to overlap will not be blocked by it, but the collision will be registered by it. This collision registration is how we are able to script things such as switching on the light when the player overlaps the trigger.
- Block**: The actor(s) the trigger is set to block will not be able to pass through it. It will act like a wall.

You can either set what response you want from each actor individually, or choose the general response you want from all actors. We will be doing the latter. At the very top, there is an option called **Collision Response**. In front of that, you have three boxes, one for each type of response. Simply check the **Ignore** box, and everything below it will be set to ignore. This is what we want. We want the trigger to ignore collision for all actors initially. We will change its collision response when the player has picked up the key cube.

Coming back to the pickup setup we had made, we will need to add a few more nodes to it. Right-click anywhere in the **Event Graph**, and in the **Flow Control** section, you will find something called **Sequence**. Select and create it. A **Sequence** node takes one input and has multiple outputs. If you want a particular node to activate or set off various different events or functions, you should use a **Sequence** node.

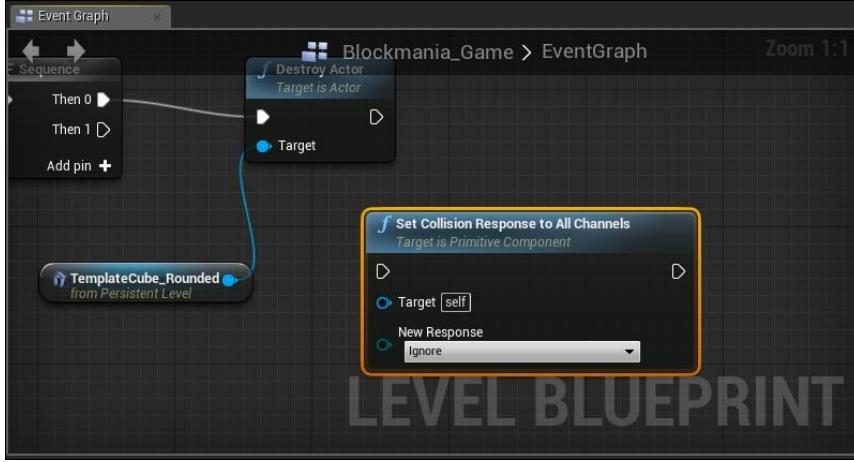


By default, a **Sequence** node has one input pin and two output pins. The first node is fired first, then the next, and so on. If you want more output pins, simply click on **Add Pins**, and it will create another output pin. Now, connect the **Gate** node's output pin to the **Sequence** node's input pin. Then, connect **Then 0** to the **Destroy Actor** node.

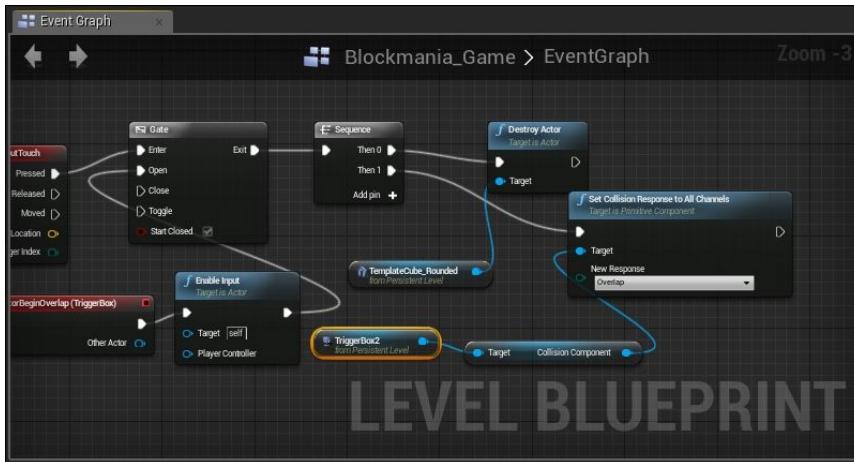


We will now change the **Collision Response** of the trigger to **Overlap** when the player has picked up the cube. With the trigger selected, right-click on it and first uncheck the **Context Sensitive** box, which is located at the top-right corner of the menu. When you have an actor selected in the **Viewport** and when you right-click in the **Graph Editor**, the event nodes and the function nodes you can see are usually correlated to the selected actor. They only display the functions and expressions that can be applied directly to the actor. Otherwise, Blueprint offers quite a few nodes, but some of them cannot be used either directly or at all on the selected actor.

The node that we need here is one that can be applied to the trigger, but not directly. What we need is a **Set Collision Response to All Channels** node. This node can be used to change the collision response of actors during runtime. Right-click in the **Event Graph** and type in the name of the node and create it.



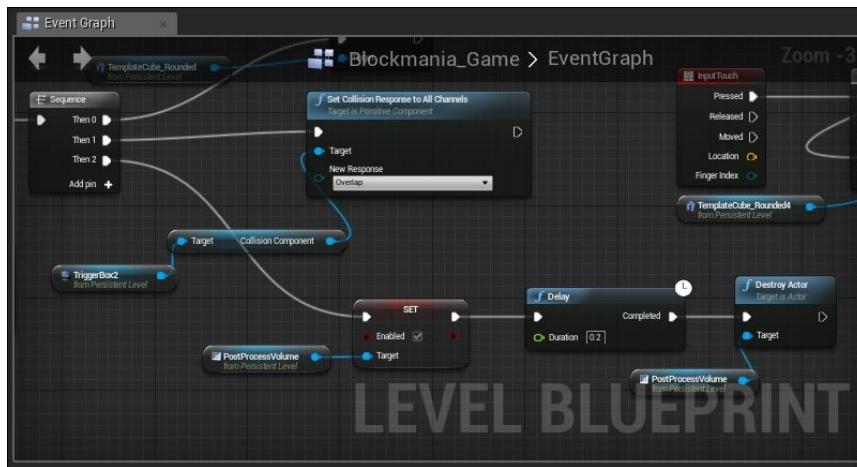
Here, you can set a target actor whose **Collision Response** you wish to change. If you click on the dropdown menu, you will see three settings: **Ignore**, **Overlap**, and **Block**. If you recall, these were the three types of responses in the trigger's **Collision** section. We had set it to **Ignore**, but we want to change it to **Overlap** once the player has picked up the key cube. So, select **Overlap** in the dropdown menu. Next, connect the **Then 1** pin to this node's input. Finally, with the trigger selected in the **Viewport**, right-click in the **Graph Editor**, check the **Context Sensitive** box, and select **Create Reference** for <actor name>. Connect this to the **Target** input of the **Set Collision Response to All Channels** node. When you do this, you will find that it does not directly connect to the **Target** input. Instead, a new node is created, which takes the trigger's reference as its input and has **Collision Component** as its output. What this does is take the trigger box and convert it into something called a **Primitive Component Reference**. It is the only way you can connect it to this node.



If we were to test the game now, we would find things working the way we want. We would not be able to place the key cube without first picking it up.

We are almost done with our setup here. Remember we had placed a **PostProcessVolume** around our key cube, which would act as a visual indicator that the player has picked up the key cube? We need to script that in as well. We had initially set it to be disabled. We will enable it via **Blueprint**, and then destroy it after a very brief moment. First, add a new pin to the **Sequence** node. Next, with the **PostProcessVolume** selected, right-click in the **Graph Editor** and create a reference for it. Then, click on its output pin, drag it out, and release the left mouse button to open the menu. Here, you can type in **Set Enabled**, which would create a **Bool** node (Bool nodes are red in color). Once created, you will see a tick box, which says **Enabled**. Tick that box, and connect it to the **Then 2** output pin of the **Sequence** node.

Next, we need to destroy the actor after a brief moment. For that, we will need to create a **Delay** node. A **Delay** node takes an input and fires off a pulse after a certain time (which you can set). Right-click in the **Graph Editor** and type in **Delay** and create it. You can also find it under **Utilities** | **Flow Control** | **Delay**. You can see an option called **Duration** in the node. Here, you can set how long before you want the pulse to be fired. For now, leave it at the default value (which is 0.2 seconds), and connect it to the **Set Enabled** node. Finally, create a **Destroy Actor** node, connect it to the **Delay** node, and set the **PostProcessVolume** as its target. You can simply select the post process reference you had created for the **Set Enabled** node by clicking **Ctrl + C**, and then **Ctrl + V**) to create a copy and connect this duplicate to the **Target** input of the **Destroy Actor** node.



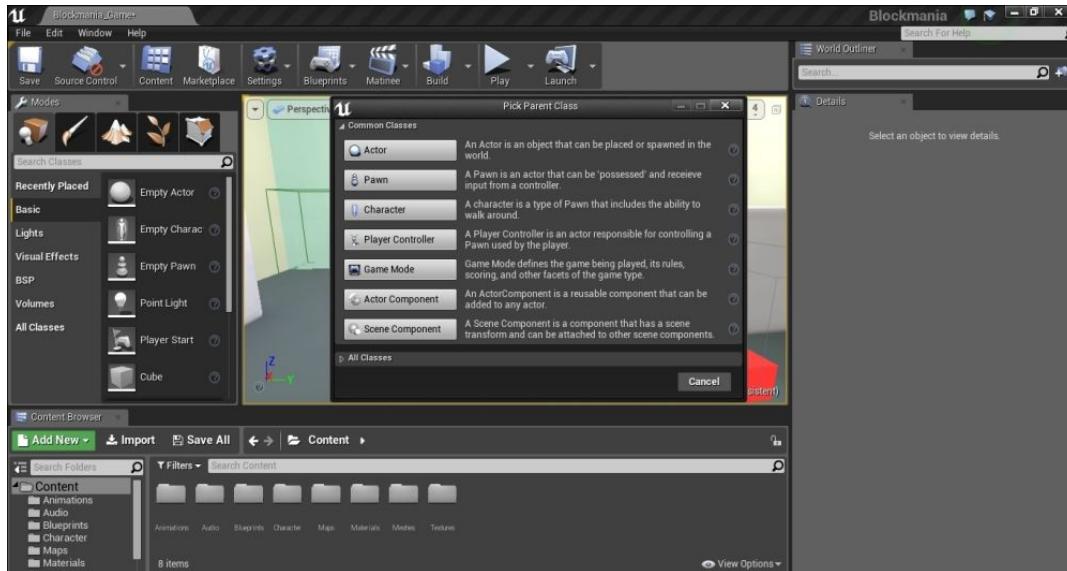
And there you have it! We have a pickup and placement system for our key cube. We also have a visual indicator that we have picked up the cube (we will cover how to script the door opening and closing in the next chapter). Now, though this was a fairly simple and small setup, doing this for every key cube would be a tedious job. Imagine if your game had 10...20...50...100 rooms! You would have to script for each key cube in the game and waste loads of time. Thankfully, UE4 offers something to get around such a scenario: a **Blueprint** class.

## The Blueprint class

As already mentioned, scripting for each key cube would just be a tedious and time-consuming task. With a Blueprint class, you would need to do all the scripting and everything else only once. A Blueprint class is an entity that contains actors (static meshes, volumes, camera classes, trigger box, and so on) and functionalities scripted in it. Looking at our example once again of the lamp turning on/off, say you want to place 10 such lamps. With a Blueprint class, you would just have to create and script once, save it, and duplicate it. This is really an amazing feature offered by UE4.

### Creating a Blueprint class

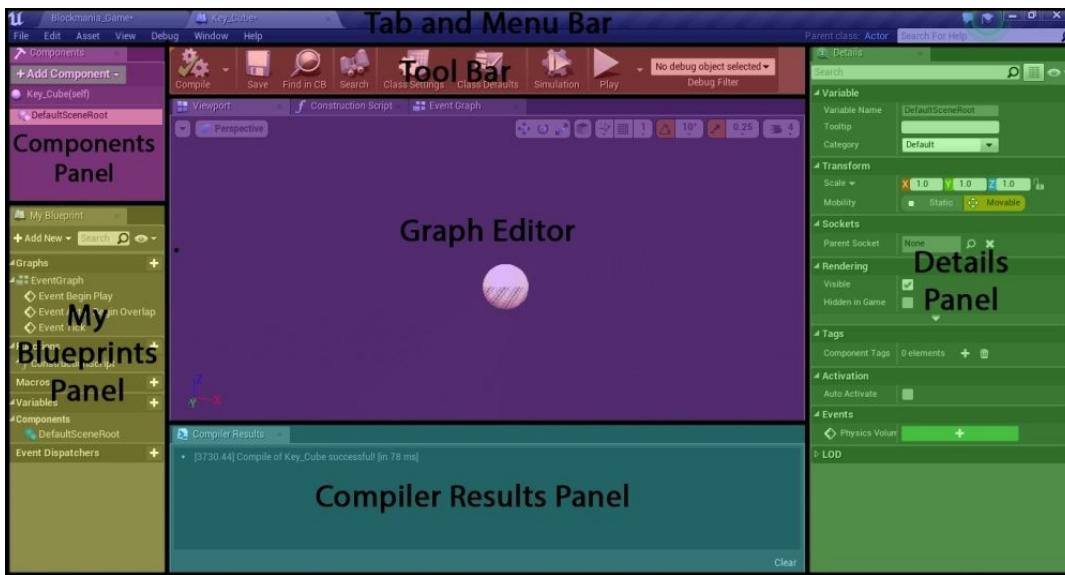
To create a Blueprint class, click on the **Blueprints** button in the **Viewport** toolbar, and in the dropdown menu, select **New Empty Blueprint Class**. A window will then open, asking you to pick your parent class, indicating the kind of Blueprint class you wish to create.



At the top, you will see the most common classes. These are as follows:

- **Actor** : An **Actor** , as already discussed, is an object that can be placed in the world (static meshes, triggers, cameras, volumes, and so on, all count as actors)
- **Pawn** : A **Pawn** is an actor that can be controlled by the player or the computer
- **Character** : This is similar to a **Pawn** , but has the ability to walk around
- **Player Controller** : This is responsible for giving the **Pawn** or **Character** inputs in the game, or controlling it
- **Game Mode** : This is responsible for all of the rules of gameplay
- **Actor Component** : You can create a component using this and add it to any actor
- **Scene Component** : You can create components that you can attach to other scene components

Apart from these, there are other classes that you can choose from. To see them, click on **All Classes** , which will open a menu listing all the classes you can create a Blueprint with. For our key cube, we will need to create an **Actor Blueprint Class** . Select **Actor** , which will then open another window, asking you where you wish to save it and what to name it. Name it **Key\_Cube** , and save it in the **Blueprint** folder. After you are satisfied, click on **OK** and the **Actor Blueprint Class** window will open.



The Blueprint class user interface is similar to that of Level Blueprint, but with a few differences. It has some extra windows and panels, which have been described as follows:

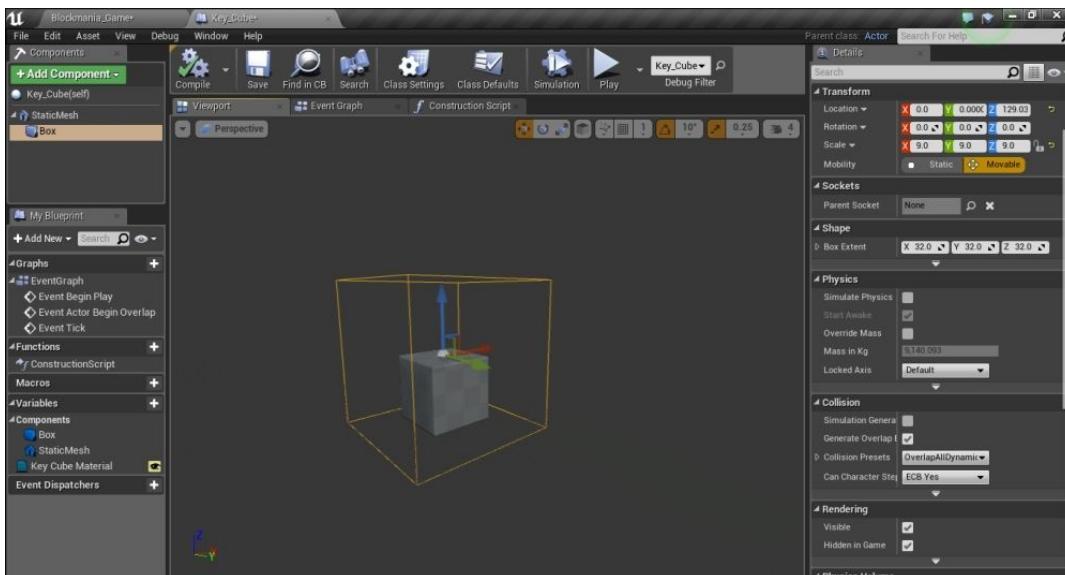
- Components panel :** The **Components panel** is where you can view, and add components to the Blueprint class. The default component in an empty Blueprint class is **DefaultSceneRoot**. It cannot be renamed, copied, or removed. However, as soon as you add a component, it will replace it. Similarly, if you were to delete all of the components, it will come back. To add a component, click on the **Add Component** button, which will open a menu, from where you can choose which component to add. Alternatively, you can drag an asset from the Content Browser and drop it in either the **Graph Editor** or the **Components panel**, and it will be added to the Blueprint class as a component. Components include actors such as static or skeletal meshes, light actors, camera, audio actors, trigger boxes, volumes, particle systems, to name a few. When you place a component, it can be seen in the **Graph Editor**, where you can set its properties, such as size, position, mobility, material (if it is a static mesh or a skeletal mesh), and so on, in the **Details panel**.
- Graph Editor :** The **Graph Editor** is also slightly different from that of Level Blueprint, in that there are additional windows and editors in a Blueprint class. The first window is the **Viewport**, which is the same as that in the Editor. It is mainly used to place actors and set their positions, properties, and so on. Most of the tools you will find in the main **Viewport** (the editor's **Viewport**) toolbar are present here as well.
- Event Graph :** The next window is the **Event Graph** window, which is the same as a Level Blueprint window. Here, you can script the components that you added in the **Viewport** and their functionalities (for example, scripting the toggling of the lamp on/off when the player is in proximity and moves away respectively). Keep in mind that you can script the functionalities of the components only present within the Blueprint class. You cannot use it directly to script the functionalities of any actor that is not a component of the Class.
- Construction Script :** Lastly, there is the **Construction Script** window. This is also similar to the **Event Graph**, as in you can set up and connect nodes, just like in the **Event Graph**. The difference here is that these nodes are activated when you are constructing the Blueprint class. They do not work during runtime, since that is when the **Event Graph** scripts work. You can use the **Construction Script** to set properties, create and add your own property of any of the components you wish to alter during the construction, and so on.

Let's begin creating the Blueprint class for our key cubes.

### Viewport

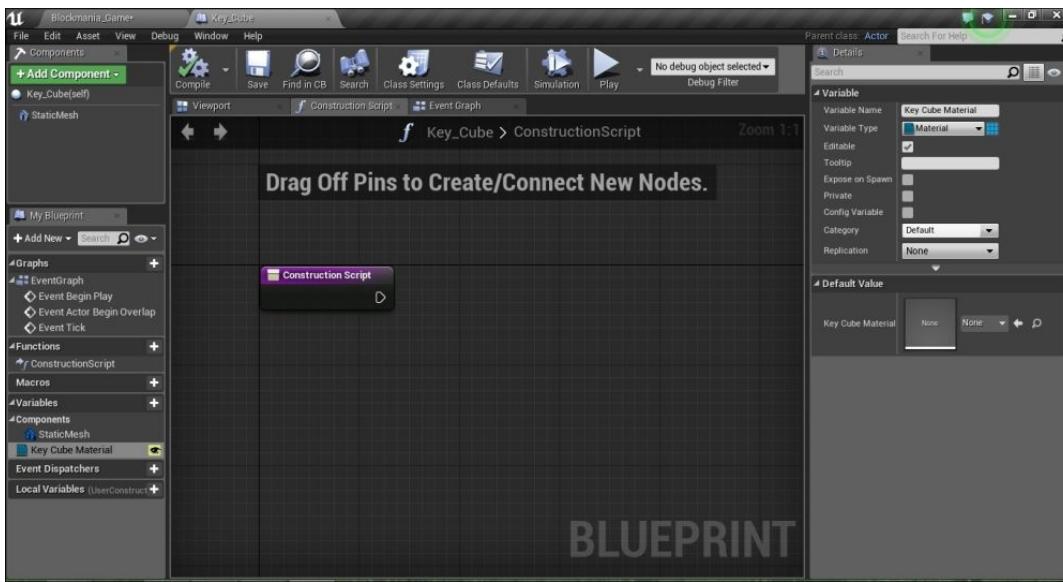
The first thing we need are the components. We require three components: a cube, a trigger box, and a **PostProcessVolume**. In the **Viewport**, click on the **Add Components** button, and under **Rendering**, select **Static Mesh**. It will add a **Static Mesh** component to the class. You now need to specify which **Static Mesh** you want to add to the class. With the **Static Mesh** actor selected in the **Components panel**, in the actor's **Details** panel, under the **Static Mesh** section, click on the **None** button and select **TemplateCube\_Rounded**. As soon as you set the mesh, it will appear in the **Viewport**. With the cube selected, decrease its scale (located in the **Details** panel) from 1 to 0.2 along all three axes.

The next thing we need is a trigger box. Click on the **Add Component** button and select **Box Collision** in the **Collision** section. Once added, increase its scale from 1 to 9 along all three axes, and place it in such a way that its bottom is in line with the bottom of the cube.



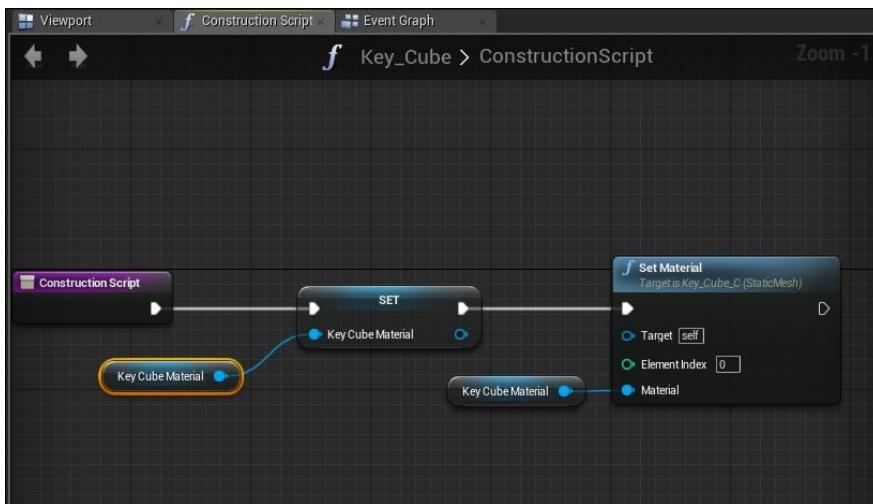
### The Construction Script

You could set its material in the **Details** panel itself by clicking on the **Override Materials** button in the **Rendering** section, and selecting the key cube material. However, we are going to assign its material using **Construction Script**. Switch to the **Construction Script** tab. You will see a node called **Construction Script**, which is present by default. You cannot delete this node; this is where the script starts. However, before we can script it in, we will need to create a variable of the type **Material**. In the **My Blueprint** section, click on **Add New** and select **Variable** in the dropdown menu. Name this variable **Key Cube Material**, and change its type from **Bool** (which is the default variable type) to **Material** in the **Details** panel. Also, be sure to check the **Editable** box so that we can edit it from outside the Blueprint class.



Next, drag the **Key Cube Material** variable from the **My Blueprint** panel, drop it in the **Graph Editor**, and select **Set** when the window opens up. Connect this to the output pin of the **Construction Script** node. Repeat this process, only this time, select **Get** and connect it to the input pin of **Key Cube Material**.

Right-click in the **Graph Editor** window and type in **Set Material** in the search bar. You should see **Set Material (Static Mesh)**. Click on it and add it to the scene. This node already has a reference of the Static Mesh actor (**TemplateCube\_Rounded**), so we will not have to create a reference node. Connect this to the **Set** node. Finally, drag **Key Cube Material** from **My Blueprint**, drop it in the **Graph Editor**, select **Get**, and connect it to the **Material** input pin. After you are done, hit **Compile**. We will now be able to set the cube's material outside of the Blueprint class.

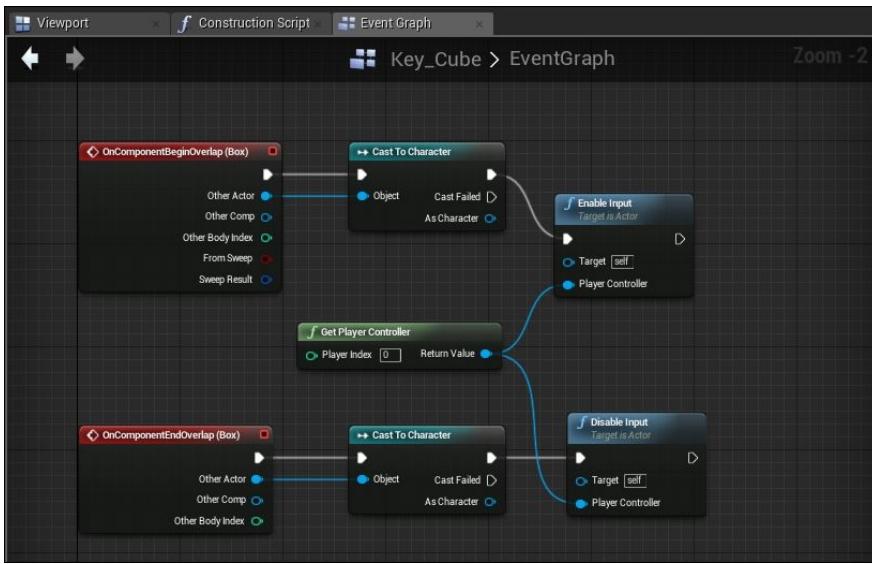


Let's test it out. Add the Blueprint class to the level. You will see a **TemplateCube\_Rounded** actor added to the scene. In its **Details** panel, you will see a **Key Cube Material** option under the **Default** section. This is the variable we created inside our **Construction Script**. Any material we add here will be added to the cube. So, click on **None** and select **KeyCube\_Material**. As soon as you select it, you will see the material on the cube. This is one of the many things you can do using **Construction Script**. For now, only this will do.

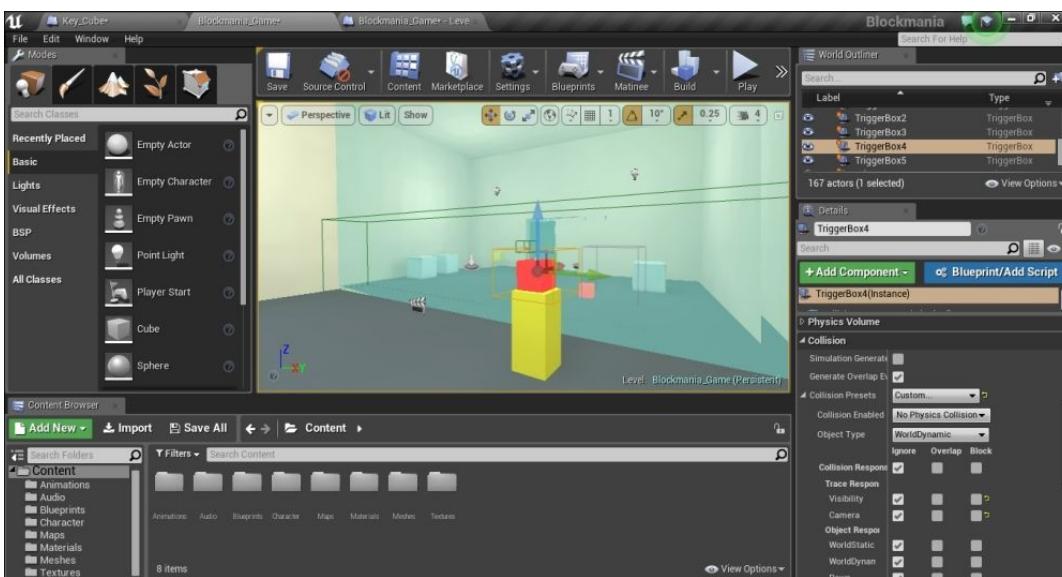
### The Event Graph

We now need to script the key cube's functionalities. This is more or less the same as what we did in the Level Blueprint with our first key cube, with some small differences. In the **Event Graph** panel, the first thing we are going to script is enabling and disabling input when the player overlaps and stops overlapping the trigger box respectively. In the **Components** section, right-click on **Box**. This will open a menu. Mouse over **Add Event** and select **Add OnComponentBeginOverlap**. This will add a **Begin Overlap** node to the **Graph Editor**. Next, we are going to need a **Cast** node. A **Cast** node is used to specify which actor you want to use. Right-click in the **Graph Editor** and add a **Cast to Character** node. Connect this to the **OnComponentBeginOverlap** node and connect the other actor pin to the **Object** pin of the **Cast to Character** node. Finally, add an **Enable Input** node and a **Get Player Controller** node and connect them as we did in the Level Blueprint.

Next, we are going to add an event for when the player stops overlapping the box. Again, right-click on **Box** and add an **OnComponentEndOverlap** node. Do the exact same thing you did with the **OnComponentBeginOverlap** node; only here, instead of adding an **Enable Input** node, add a **Disable Input** node. The setup should look something like this:



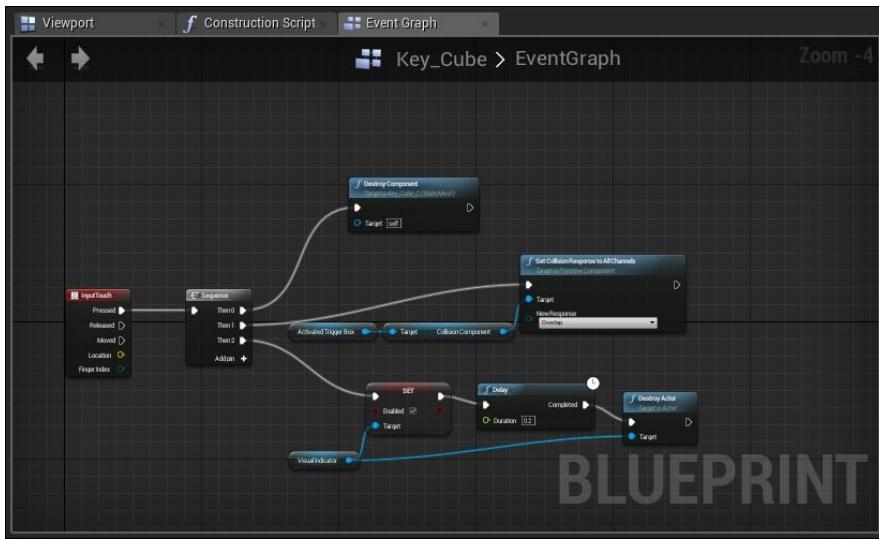
You can move the key cube we had placed earlier on top of the pedestal, set it to hidden, and put the key cube Blueprint class in its place. Also, make sure that you set the collision response of the trigger actor to **Ignore**.



The next step is scripting the destruction of the key cube when the player touches the screen. This, too, is similar to what we had done in Level Blueprint, with a few differences. Firstly, add a **Touch** node and a **Sequence** node, and connect them to each other. Next, we need a **Destroy Component** node, which you can find under **Components** | **Destroy Component** (Static Mesh). This node already has a reference to the key cube (Static Mesh) inside it, so you do not have to create an external reference and connect it to the node. Connect this to the **Then 0** node.

We also need to activate the trigger after the player has picked up the key cube. Now, since we cannot call functions on actors outside the Blueprint class directly (like we could in Level Blueprint), we need to create a variable. This variable will be of the type **Trigger Box**. The way this works is, when you have created a **Trigger Box** variable, you can assign it to any trigger in the level, and it will call that function to that particular trigger. With that in mind, in the **My Blueprint** panel, click on **Add New** and create a variable. Name this variable **Activated Trigger Box**, and set its type to **Trigger Box**. Finally, make sure you tick on the **Editable** box; otherwise, you will not be able to assign any trigger to it. After doing that, create a **Set Collision Response to All Channels** node (uncheck the **Context Sensitive** box), and set the **New Response** option to **Overlap**. For the target, drag the **Activated Trigger Box** variable, drop it in the **Graph Editor**, select **Get**, and connect it to the **Target** input.

Finally, for the Post Process Volume, we will need to create another variable of the type **PostProcessVolume**. You can name this variable **Visual Indicator**, again, while ensuring that the **Editable** box is checked. Add this variable to the **Graph Editor** as well. Next, click on its pin, drag it out, and release it, which will open the actions menu. Here, type in **Enabled**, select **Set Enabled**, and check **Enabled**. Finally, add a **Delay** node and a **Destroy Actor** and connect them to the **Set Enabled** node, in that order. Your setup should look something like this:

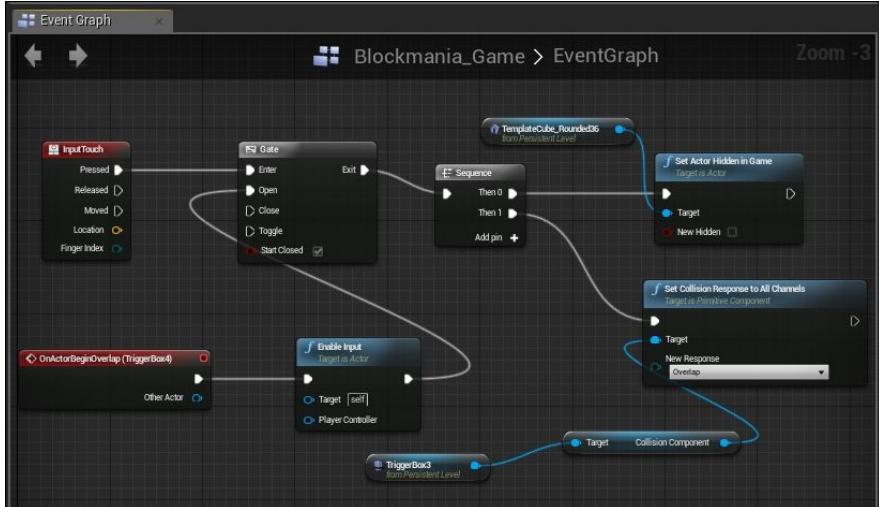


Back in the **Viewport**, you will find that under the **Default** section of the Blueprint class actor, two more options have appeared: **Activated Trigger Box** and **Visual Indicator** (the variables we had created). Using this, you can assign which particular trigger box's collision response you want to change, and which exact post process volume you want to activate and destroy. In front of both variables, you will see a small icon in the shape of an eye dropper. You can use this to choose which external actor you wish to assign the corresponding variable. Anything you scripted using those variables will take effect on the actor you assigned in the scene. This is one of the many amazing features offered by the Blueprint class. All we need to do now for the remaining key cubes is:

- Place them in the level
- Using the eye dropper icon that is located next to the name of the variables, pick the trigger to activate once the player has picked up the key cube, and which post process volume to activate and destroy.

In the second room, we have two key cubes: one to activate the large door and the other to activate the door leading to the third room. The first key cube will be placed on the pedestal near the big door. So, with the first key cube selected, using the eye dropper, select the trigger box on the pedestal near the big door for the **Activated Trigger Box** variable. Then, pick the post process volume inside which the key cube is placed for the **Visual Indicator** variable.

The next thing we need to do is to open **Level Blueprint** and script in what happens when the player places the key cube on the pedestal near the big door. Doing what we did in the previous room, we set up nodes that will unhide the hidden key cube on the pedestal, and change the collision response of the trigger box around the big door to **Overlap**, ensuring that it was set to **Ignore** initially.



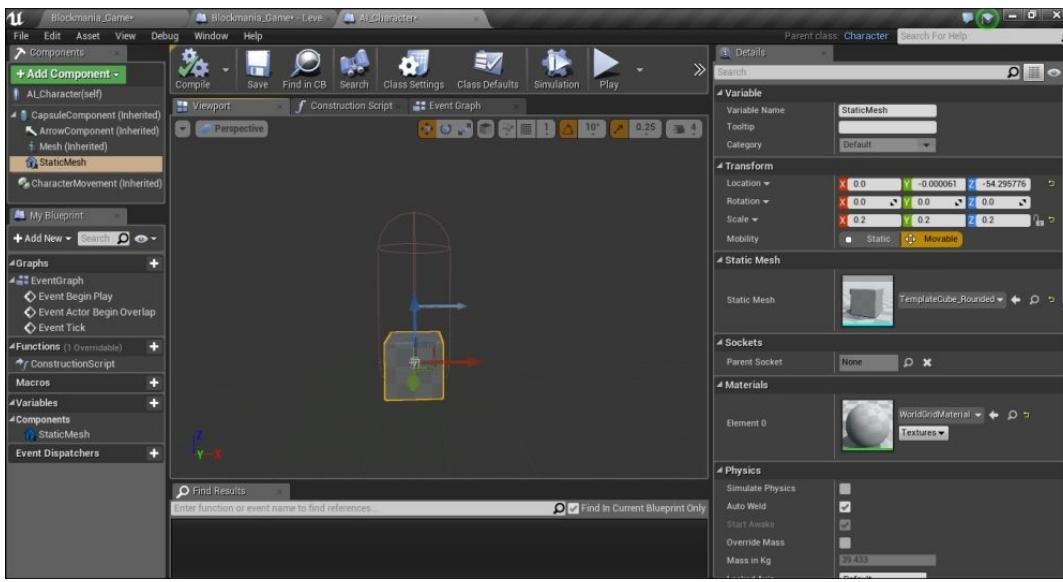
Test it out! You will find that everything is working as expected. Now, do the same with the remaining key cubes. Pick which trigger box and which post process volume to activate when you touch on the screen. Then, in the Level Blueprint, script in which key cube to unhide, and so on (place the key cubes we had placed earlier on the pedestals and set it to **Hidden**), and place the Blueprint class key cube in its place.

This is one of the many ways you can use Blueprint class. You can see it takes a lot of work and hassle. Let us now move on to Artificial intelligence.

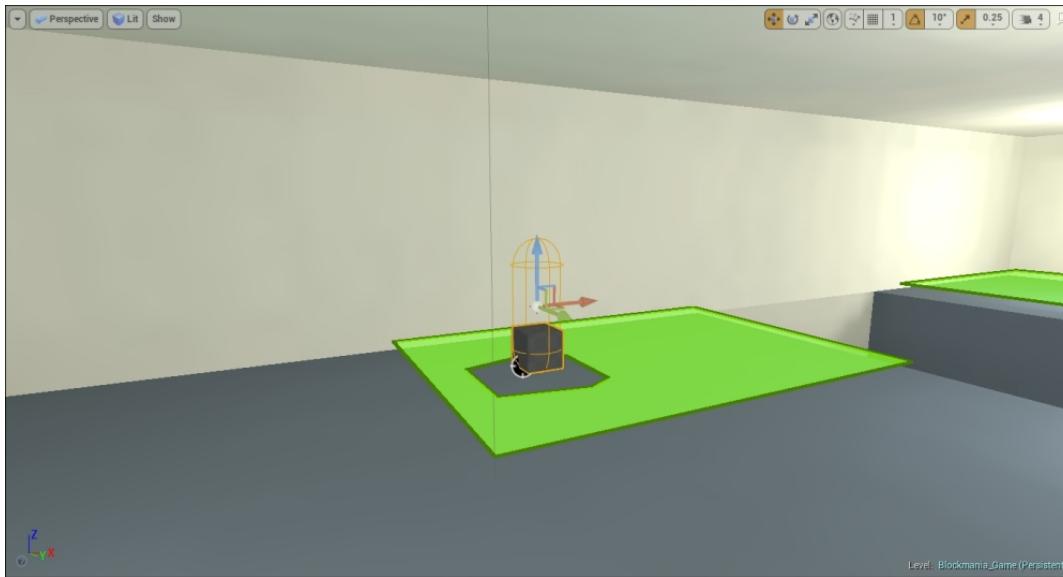
## Scripting basic AI

Coming back to the third room, we are now going to implement AI in our game. We have an AI character in the third room which, when activated, moves. The main objective is to make a path for it with the help of switches and prevent it from falling. When the AI character reaches its destination, it will unlock the key cube, which the player can then pick up and place on the pedestal. We first need to create another Blueprint class of the type **Character**, and name it **AI\_Character**. When created, double-click on it to open it. You will see a few components already set up in the **Viewport**. These are the **CapsuleComponent** (which is mainly used for collision), **ArrowComponent** (to specify which side is the front of the character, and which side is the back), **Mesh** (used for character animation), and **CharacterMovement**. All four are there by default, and cannot be removed. The only thing we need to do here is add a **StaticMesh** for our character, which will be **TemplateCube\_Rounded**.

Click on **Add Components**, add a **StaticMesh**, and assign it **TemplateCube\_Rounded** (in its **Details** panel). Next, scale this cube to **0.2** along all three axes and move it towards the bottom of the **CapsuleComponent**, so that it does not float in midair.



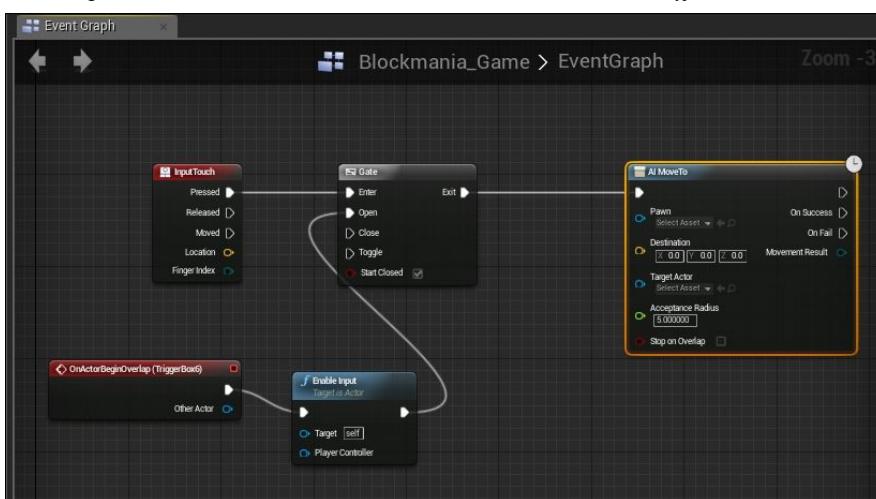
This is all we require for our AI character. The rest we will handle in Level Blueprints. Next, place **AI\_Character** into the scene on the Player side of the pit, with all of the switches. Place it directly over the Target Point actor.



Next, open up Level Blueprint, and let's begin scripting it. The left-most switch will be used to activate the AI character, and the remaining three will be used to draw the parts of a path on which it will walk to reach the other side.

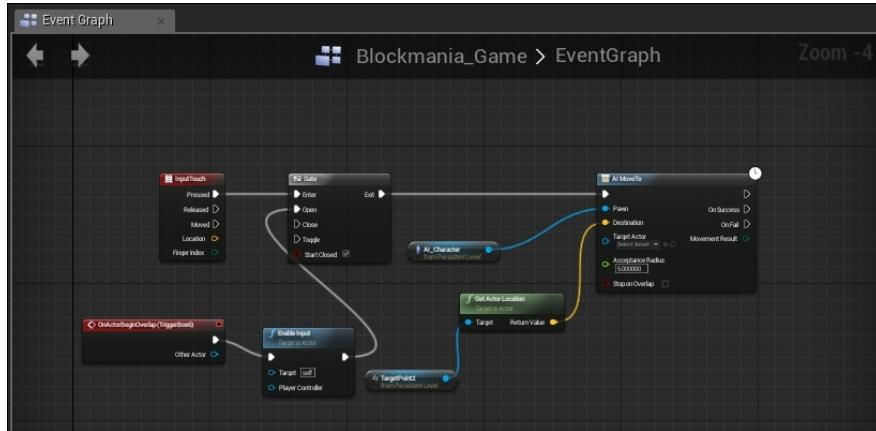
To move the AI character, we will need an **AI Move To** node. The first thing we need is an overlapping event for the trigger over the first switch, which will enable the input, otherwise the AI character will start moving whenever the player touches the screen, which we do not want. Set up an **Overlap** event, an **Enable Input** node, and a **Gate** event. Connect the **Overlap** event to the **Enable Input** event, and then to the **Gate** node's **Open** input.

The next thing is to create a **Touch** node. To this, we will attach an **AI Move To** node. You can either type it in or find it under the AI section. Once created, attach it to the **Gate** node's **Exit** pin.

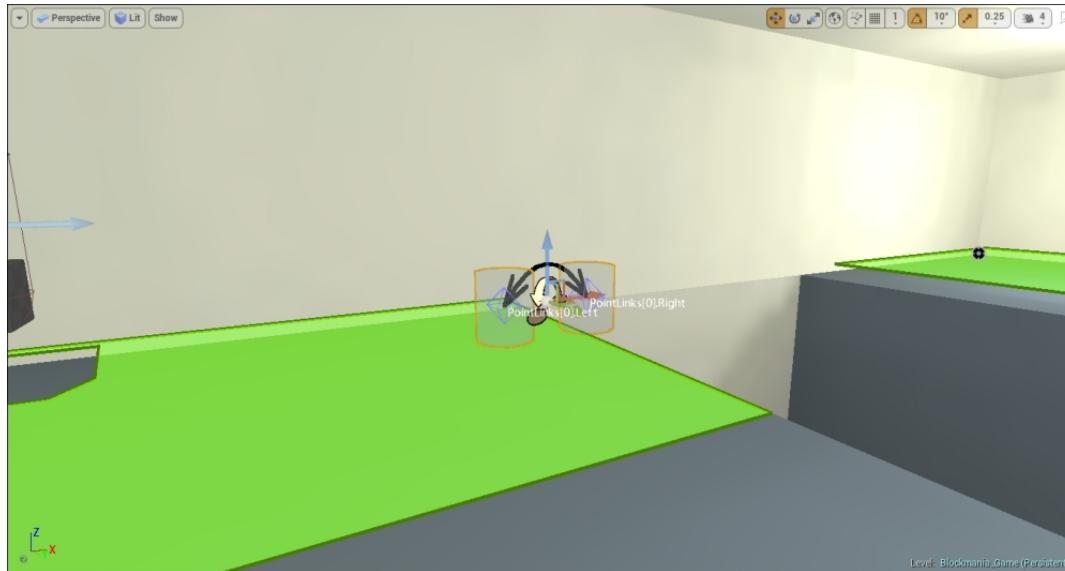


We now need to specify to the node which character we want to move, and where it should move to. To specify which character we want to move, select the AI character in the **Viewport**, and in the Level Blueprint's **Graph Editor**, right-click and create a reference for it. Connect it to the **Pawn** input pin. Next, for the location, we want the AI character to move towards the second **Target Point** actor, located on the other side of the pit. But first, we need to get its location in the world. With it selected, right-click in the **Graph Editor**, and type in `Get Actor Location`. This node returns an actor's location (coordinates) in the world (the one connected to it). This will create a **Get Actor Location**, with the **Target Point** actor connect to its input pin.

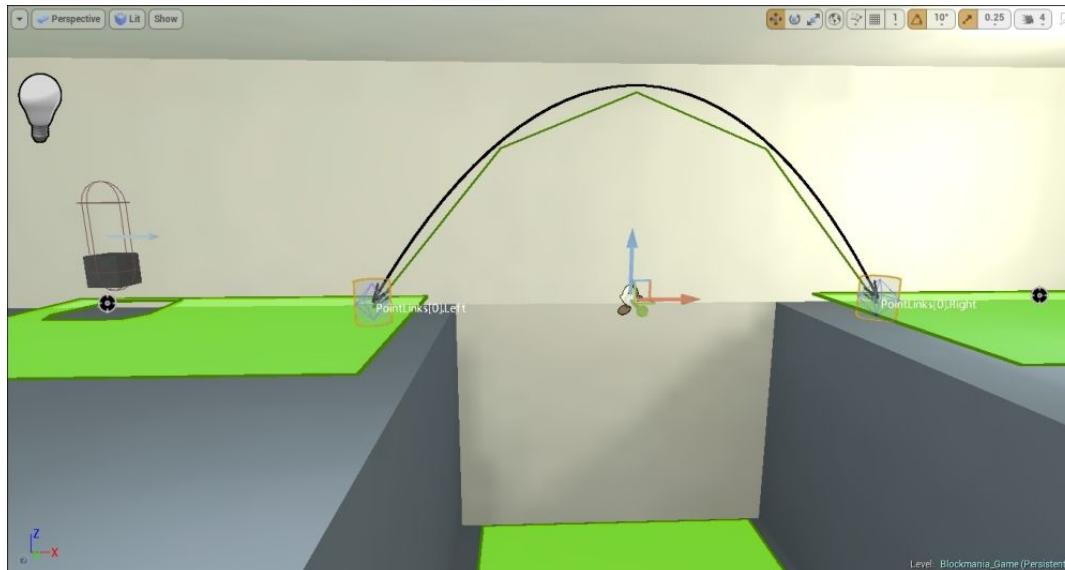
Finally, connect its **Return Value** to the **Destination** input of the **AI Move To** node.



If you were to test it out, you would find that it works fine, except for one thing the AI character stops when it reaches the edge of the pit. We want it to fall off the pit if there is no path. For that, we will need a **Nav Proxy Link** actor. As discussed in the previous chapter, a **Nav Proxy Link** actor is used when an AI character has to step outside the Nav Mesh temporarily (for example, jump between ledges). We will need this if we want our AI character to fall off the ledge. You can find it in the **All Classes** section in the **Modes** panel. Place it in the level.



The actor is depicted by two cylinders with a curved arrow connecting them. We want the first cylinder to be on one side of the pit and the other cylinder on the other side. Using the **Scale** tool, increase the size of the **Nav Proxy Link** actor.



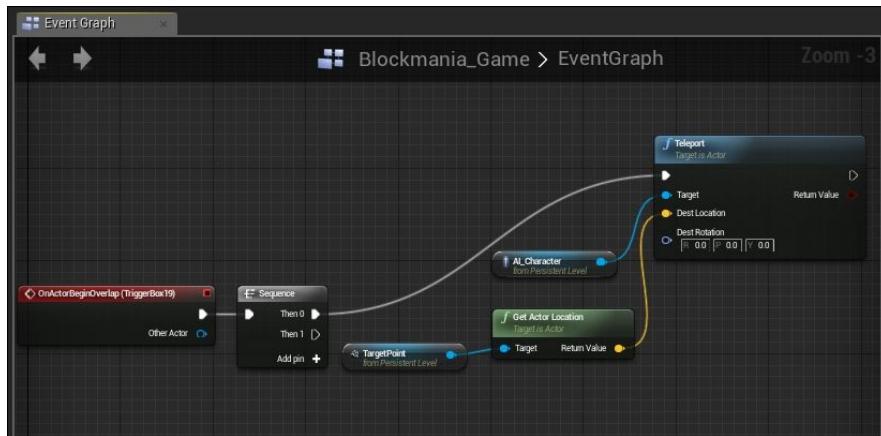
When placing the **Nav Proxy Link** actor, keep two things in mind:

- Make sure that both cylinders intersect in the green area; otherwise, the actor will not work
- Ensure that both cylinders are in line with the AI character; otherwise, it will not move in a straight line but instead to where the cylinder is located

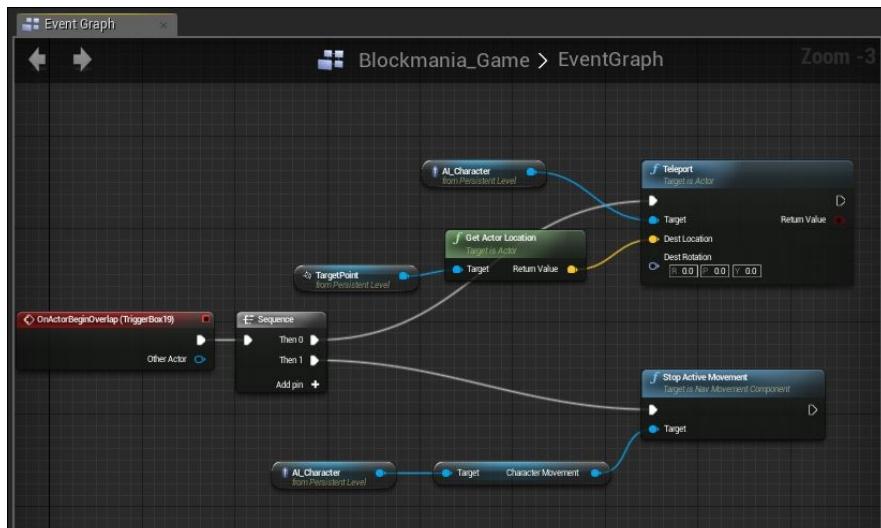
Once placed, you will see that the AI character falls off when it reaches the edge of the pit. We are not done yet. We need to bring the AI character back to its starting position so that the player can start over (or else the player will not be able to progress). For that, we need to first place a trigger at the bottom of the pit, making sure that if the AI character does fall into it, it overlaps the trigger. This trigger will perform two actions: first, it will teleport the AI character to its initial location (with the help of the first Target Point); second, it will stop the **AI Move To** node, or it will keep moving even after it has been teleported.



After placing the trigger, open Level Blueprint and create an Overlap event for the trigger box. To this, we will add a **Sequence** node, since we are calling two separate functions for when the player overlaps the trigger. The first node we are going to create is a **Teleport** node. Here, we can specify which actor to teleport, and where. The actor we want to teleport is the AI character, so create a reference for it and connect it to the **Target** input pin. As for the destination, first use the **Get Actor Location** function to get the location of the first Target Point actor (upon which the AI character is initially placed), and connect it to the **Dest Location** input.

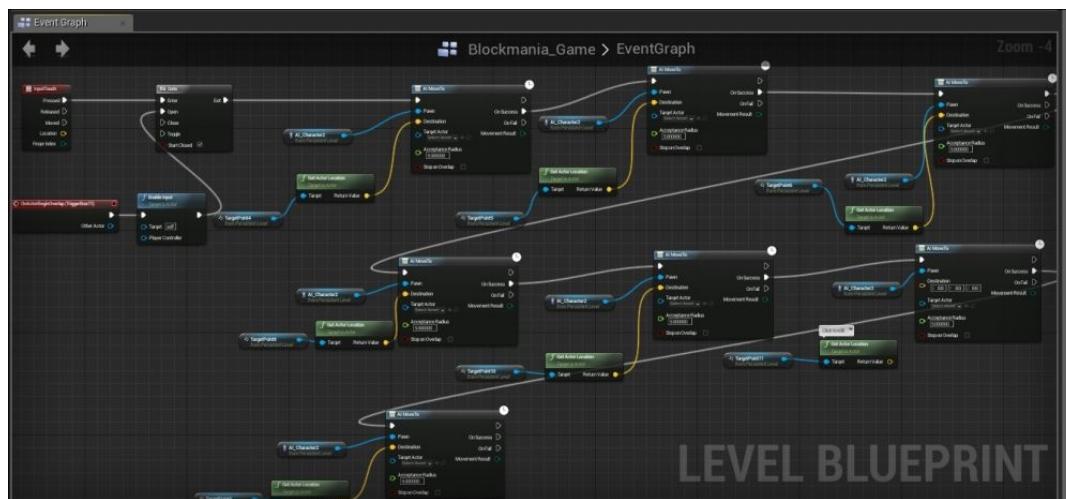


To stop the AI character's movement, right-click anywhere in the **Graph Editor**, and first uncheck the **Context Sensitive** box, since we cannot use this function directly on our AI character. What we need is a **Stop Active Movement** node. Type it into the search bar and create it. Connect this to the **Then 1** output node, and attach a reference of the AI character to it. It will automatically convert from a **Character Reference** into **Character Movement** component reference.



This is all that we need to script for our AI in the third room. There is one more thing left: how to unlock the key cube. But we will cover this in the next chapter since it involves Matinee.

In the fourth room, we are going to use the same principle. Here, we are going to make a chain of **AI Move To** nodes, each connected to the previous one's **On Success** output pin. This means that when the AI character has successfully reached the destination (Target Point actor), it should move to the next, and so on. Using this, and what we have just discussed about AI, script the path that the AI will follow (recall the previous chapter, where we lined out the path the AI character would take in the fourth room).



## Summary

In this chapter, we covered with Blueprints and discussed how they work. We also discussed Level Blueprints and the Blueprint class, and covered how to script AI. We still have a few more things to script, but first we will have to cover the topic of Unreal Matinee. In the next chapter, we will be doing just that.

## Chapter 6. Using Unreal Matinee

Unreal Matinee is yet another powerful tool offered by Unreal Engine 4. Unreal Matinee is used to create cinematics, cutscenes, animations, set pieces, and so on. It is also easy to learn, and you can use it to create amazing things. Usually, Matinee is used in conjunction with Blueprints. In the previous chapter, we had left out scripting a few things in our game, which we will be covering now. The following topics will be covered in this chapter:

- What is Unreal Matinee?
- Unreal Matinee user interface
- Using Unreal Matinee in our game

## What is Unreal Matinee?

Unreal Matinee is an animating tool. It provides tools which you can use to animate the properties of actors during gameplay or create cinematics, cutscenes, set pieces, and so on. With the help of curves and key frames, you can use this tool to animate actors in the game, just like any other video editing software that professionals use. You can also use Matinee to set up Matinee events.

### Adding Matinee actors

Before you can use Matinee, the first thing you need to do is to add a Matinee actor into the scene. A Matinee actor is depicted by a clapper, like the ones you see on movie sets, as shown in the following screenshot:

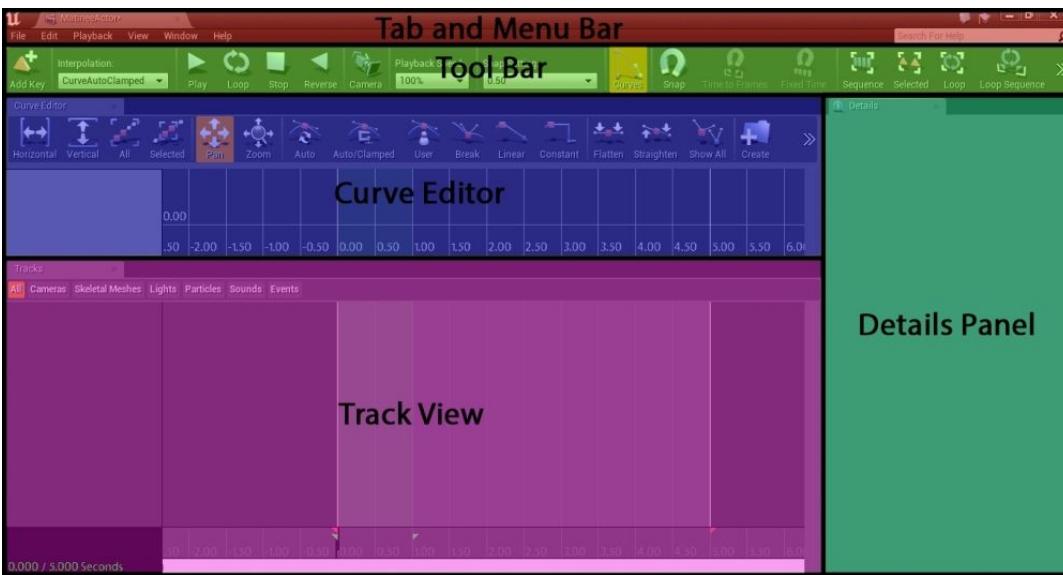


We have already discussed one way of adding Matinee actors in [Chapter 4 , Using Actors, Classes, and Volumes](#), through the Modes panel. Another way of adding Matinee actors while keeping a check on how many of them are placed in the scene is by clicking on the **Matinee** button in the Viewport toolbar, which opens a drop-down menu. Here, at the top of the menu, under the **New** section, you have the option to create a new Matinee actor by clicking on **Add Matinee**. Underneath this, in the **Edit Existing Matinee** section, you can find a list of all the Matinee actors placed in the list. Double-click on any of them to edit.

When you create a new Matinee actor or double-click on an existing one, a new window opens up. Let us take a closer look at it.

## The Unreal Matinee user interface

Anyone who has used Matinee in **Unreal Development Kit (UDK)** will find that the layout and the user interface are quite similar.



## The tab and menu bar

Just as in a web browser, in the tab bar, you can see how many windows are currently open, swap between them, rearrange them, and close any one of them.

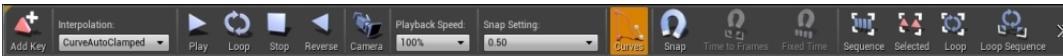


Below that is the menu bar, containing the following actions and functions:

- File** : From here, you have the option to either import sequences from FBX files (since FBX files, along with the mesh, also store the animation of the object, importing it will import the animation sequence of that FBX file to Matinee), export an animation sequence, save a sequence, and so on
- Edit** : With this tab, you can undo or redo previous actions, add or remove key frames, edit sections, and so on
- Playback** : This gives you the options to play, pause, stop, loop, and reverse your animation sequence
- View** : Here, you can set what you wish to view, enable grid snapping, fit the entire sequence in the track view panel, and so on
- Window** : This tab allows you to customize your Matinee UI by setting what panels and windows you want displayed in the UI, and so on
- Help** : You can access document and tutorials on Matinee from here

## The toolbar

Below the menu bar is the toolbar. Here, you can access the most commonly used actions.



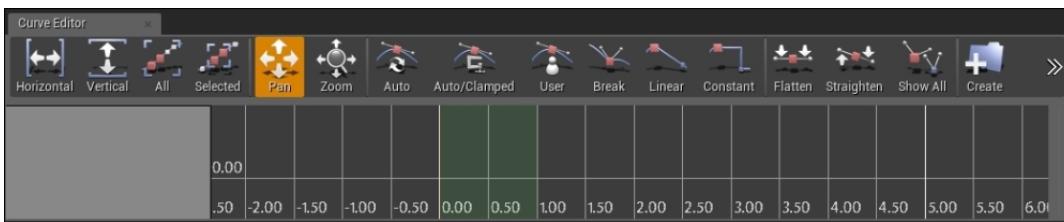
From left to right, the icons are as follows:

- Add Key** : This adds a key frame on the current position of the animation track.
- Interpolation** : Interpolation means finding a point between two other points. In key frame animation, interpolation is an important tool, since it makes the animation smooth. With this, you can set the initial interpolation mode of the keys and curves. You can choose from linear, constant, curve auto clamped, curved auto, and so on.
- Play** : The play button plays the animation sequence you create. You can see the animation in the Editor Viewport. The animation only plays once.
- Loop** : If you want to view the animation multiple times, you can click on the **Loop** button, and it will loop the animation once it has finished.
- Stop** : Clicking on this will stop the animation.
- Reverse** : This plays the animation in reverse.
- Camera** : Clicking on this will enable you to create a **Camera** group.
- Playback Speed** : With this option, you can set the speed at which you want the animation to play. You can choose from 100%, 50%, 25%, 10%, and 1% of the normal play speed.
- Snap Settings** : If you have grid snapping enabled, you can set the snap size from here.
- Curve** : Enabled by default, you can use this to open/close the Curve Editor window.
- Snap** : Represented by a magnet, this toggles grid snapping.
- Time to Frames** : This snaps the timeline cursor to the snap size set in **Snap Setting**. It is enabled only if the snap size is in frames per second.
- Fixed Time** : Using this, you can fix the playback speed to the framerate chosen in the **Snap Setting** menu. It only works if the snap setting is in frames per second.
- Sequence** : Click on this to zoom to fit the entire sequence you created in the Tracks panel.
- Selected** : This zooms to fit the selected key frames in the Tracks panel.
- Loop** : This zooms the timeline to fit the loop sections in the animation sequence.
- Loop Sequence** : This sets the starting point of the loop section to the start of the animation sequence, and the end to the end of the sequence. In other words, clicking on this will resize and fit the loop section to the entire animation sequence you created.
- End** : If you click on the arrow at the far right corner of the toolbar, you will find three more icons. The first of them is **End**. This moves the Tracks timeline to the end of the sequence.
- Recorder** : This opens the **Matinee Recorder** window, which you can use to record sequences for your Matinee.
- Movie** : Finally, we have the **Movie** option. You can use this to create a movie from the animation sequence you created.

## The Curve Editor

The next window we have is the **Curve Editor**. As with any animation software, you can edit your animations using curves. It is also used if you want to have particle systems in your animation sequence. There are other numerous uses of curves, but these are the most common uses. The x axis represents time, and the y axis is the value that you are changing in your Matinee over time. The curve in the Curve Editor shows that the value is changing over time. For instance, if the value is changing at a constant rate, the curve would be linear; if the value is changing exponentially, the curve would be an exponential curve, and so on.

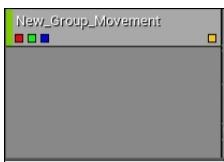
To view and edit curves, you first have to toggle them in the **Tracks** Panel. Once done, you will see the curve of the corresponding animation sequence, which you can then edit in order to further edit or fine-tune your animation.



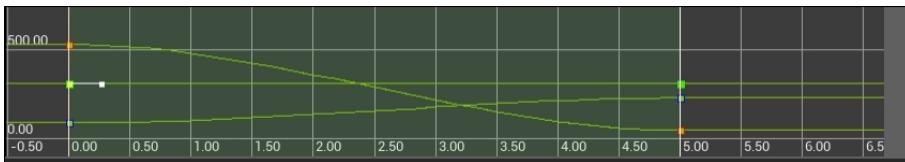
At the top is the toolbar of the Curve Editor. Let's look at its options:

- **Horizontal** : This option enables zooming to fit the selected curve group horizontally.
- **Vertical** : This is the same as the **Horizontal** option, but it fits the selected curve group vertically.
- **All** : It fits the curve both horizontally as well as vertically so that the whole curve is visible in the Curve Editor.
- **Selected** : This performs the same function as **All**, but only for the selected points.
- **Pan** : This switches the **Curve Editor** to pan mode, meaning you can move around in the graph editor by holding the left mouse button and dragging the mouse.
- **Zoom** : This switches to the zoom mode. You can zoom in and out by holding the left mouse button and dragging the mouse.
- **Auto** : This changes the selected curves interpolation to **Auto**.
- **Auto/Clamped** : Set the selected curves interpolation to **Auto/Clamped**.
- **User** : This option changes the interpolation to **User** (if you make any change to the curve while it is set to **Auto** or **Auto/Clamped**, it will change the interpolation to **User**).
- **Break** : This changes the interpolation to **Break**.
- **Linear** : This changes the interpolation to **Linear**. This means that the curve between the two points will be linear, that is, a straight line between the two key frames.
- **Constant** : This changes the interpolation to **Constant**. This means that the curve between the selected key frames will be a straight horizontal line. In other words, the value along the **Y** axis will remain constant over time.
- **Flatten** : Clicking on this will flatten the selected tangent of the curve, making it flat.
- **Straighten** : This option removes any irregularities from the selected tangent of the curve and makes it straight.
- **Show All** : This shows all the tangents of all the curves present.
- **Create** : Finally, we have the **Create** tab button. You have the option to create multiple tabs. You can switch between tabs by clicking on the arrow button at the extreme right corner of the toolbar and selecting which tab to go to using the **Current** tab menu. You can also delete the current tab.

Below the toolbar, on the left-hand side, is the track list. Here, you can see all of the tracks in the current tab. At the top is the name of the track, and at the bottom are buttons that you can toggle on/off. In this example, we have a movement track, so there are three buttons representing each axis: red represents the **X** axis, green the **Y** axis, and blue the **Z** axis. At the far right is a yellow button, which can be used to toggle the entire track on/off.



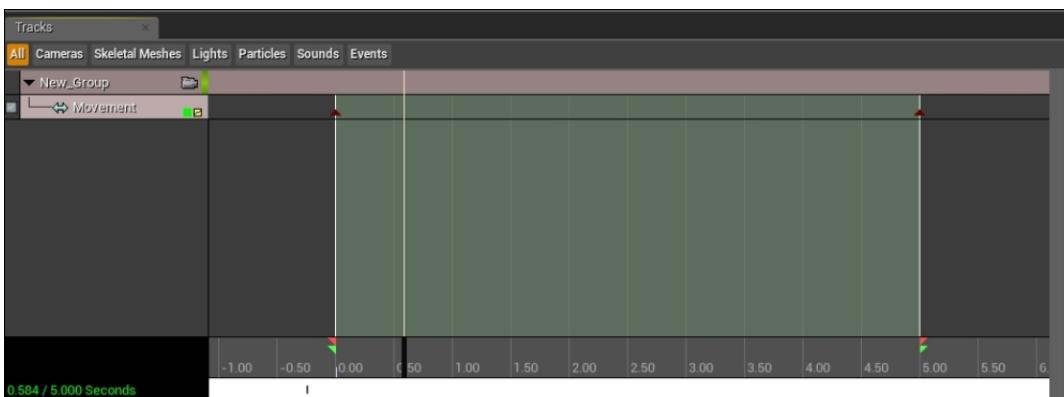
To its right, we have the Graph Editor, where you can see and tweak the curves.



Here, you can see that there are three separate curves—each representing the buttons we just discussed. You can tweak each of the curves here to get the desired result. Select a point on the curve and change the tangent to alter the shape of the curve.

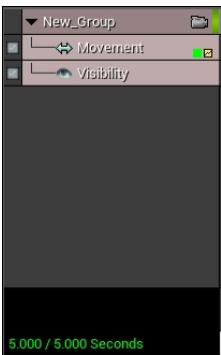
## The Tracks panel

In the **Tracks** panel, you can see all of the tracks, folders, and groups in your animation sequence.



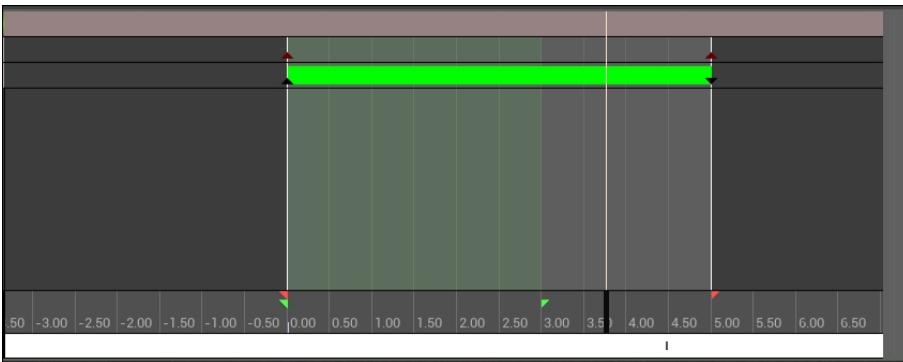
Here, you can see the timeline specifying how long the animation is, the groups you have created, all the components that are involved in the animation, and so on. At the top of the panel is the tab bar. Below it is the **Group** tab. For simple animations, you do not really need this, but if you have a complex animation sequence (such as a cinematic or a cutscene), you may want to make use of this feature to keep your workstation neat and organized. You can put similar actors in their respective tabs. For example, if you have cameras, you can place them in the **Camera** group. To create a group or a folder, right click on it; this will open a menu, from which you can create a group or a folder.

Below the **Group** tab, on the left, is the Group and Track list. All of the groups and tracks in your animation sequence or in the current tab are listed. You can also add or remove groups here.



As you can see, there are various tracks under a group. Each individual track has its own animation which you can edit. You can also toggle an entire track on/off by clicking on the gray box on the left of each track. On the bottom-right corner are two more boxes. The button on the left enables or disables the respective track from playing in the animation sequence. The one on the right enables or disables the Curve Editor for that track. Also, some tracks do not have these two buttons, indicating that those tracks do not have a curve.

To its right is the animation timeline.



At the bottom, you can see the animation timescale. Above it, the translucent white box with the red edges depicts the total duration of the animation time. You can click on the edges and drag them left or right to decrease or increase the duration of the animation respectively. The light green box shows the duration of the loop sequence. This means that only the section inside the green segment will loop, the rest will be ignored. You can also increase or decrease its size by clicking and dragging the green arrow right or left. There is also the timeline slider, depicted by the white line, which you can use to jump to any frame. Finally, you can click anywhere on the time bar to jump to any frame. The timeline slider will jump to wherever you click.

## The Details panel

Finally, there is the **Details** panel, wherein you can set the properties of certain tracks, groups, and so on.



The preceding screenshot shows the properties of the key frame of a movement track. You can toggle the movement track from playing in the Viewport, toggle the rotation on/off, set whether you want to see the track in the Editor Viewport, and so on.

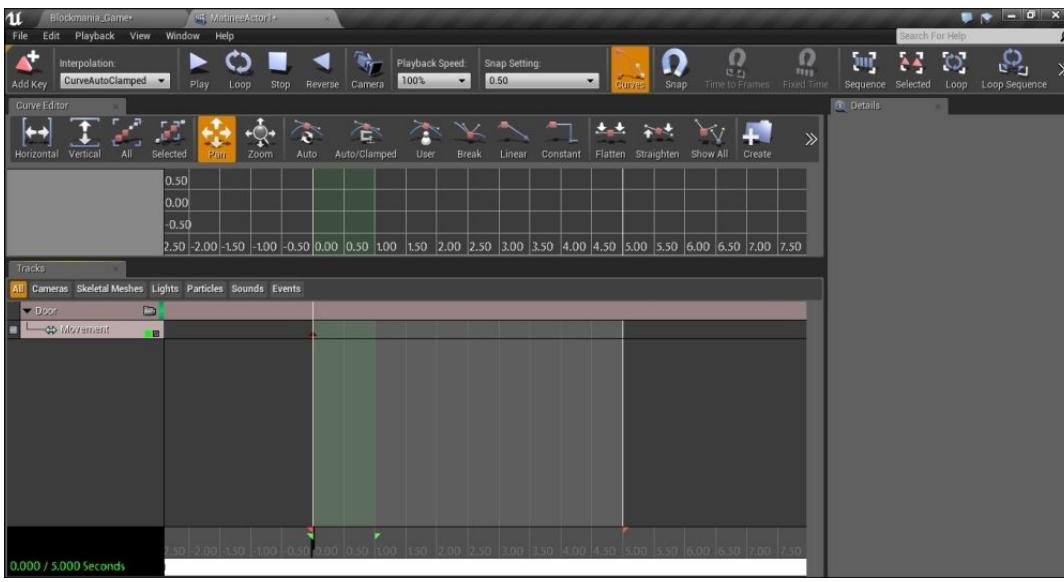
## Animating the door

Now that you are acquainted with Unreal Matinee and its user interface, we can go ahead and add a few animations to our game. This includes animations for the doors opening, a small cutscene in the first room, and drawing a bridge for the AI character.

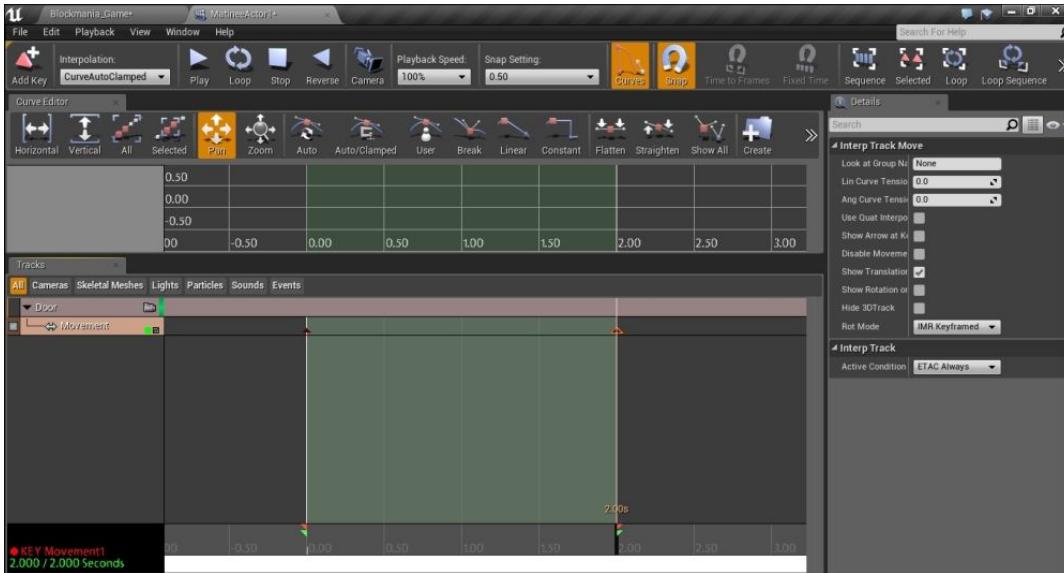
### Room 1

It is now time to actually use Matinee in our game. Select the Matinee actor near the door in the first room and click on **Open Matinee** in the **Details** panel. Once opened, the first step is to add a group. Before making a group, be sure that the actor or actors you want to animate are selected in the **Editor Viewport**; otherwise, the group will not have any actor reference in it, and you will get an error saying **Nothing to keyframe**, or **Selected object cant be keyframed on this type of track**.

Click on the door to select it, right-click on the **Track list** in the **Tracks** panel, and select **Add New Empty Group**. When asked to name the group, name it **Door** and press **Enter**. The door will be attached to this group. This will create an empty group in which you can place multiple tracks (movement, visibility, particles, and so on). First, we will need a movement track. Right-click on **Door** and select **Add New Movement Track**. Once created, you will see it in the **Tracks** panel.

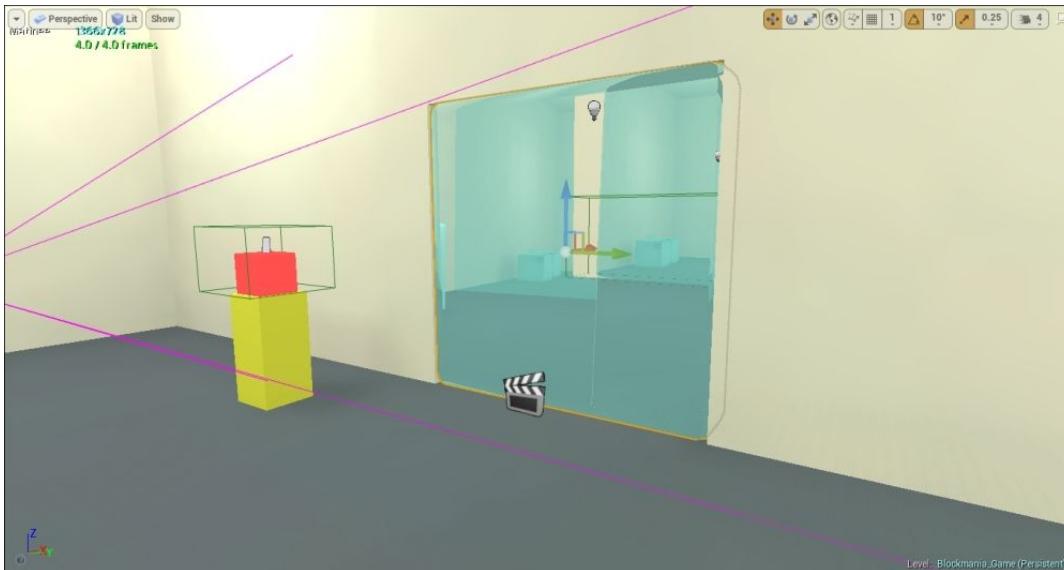


Now, we do not want the animation to last for 5 seconds, since it is too long for a simple door opening animation and would unnecessarily slow the game down. So, decrease the animation duration to 2 seconds by left-clicking on the red arrow on the extreme right (which is above the 5-second mark) and move it back over to the 2-second mark. If you want, you can also increase the loop sequence duration from 1 second to 2 seconds by doing the same with the green arrow on the right edge of the green box.



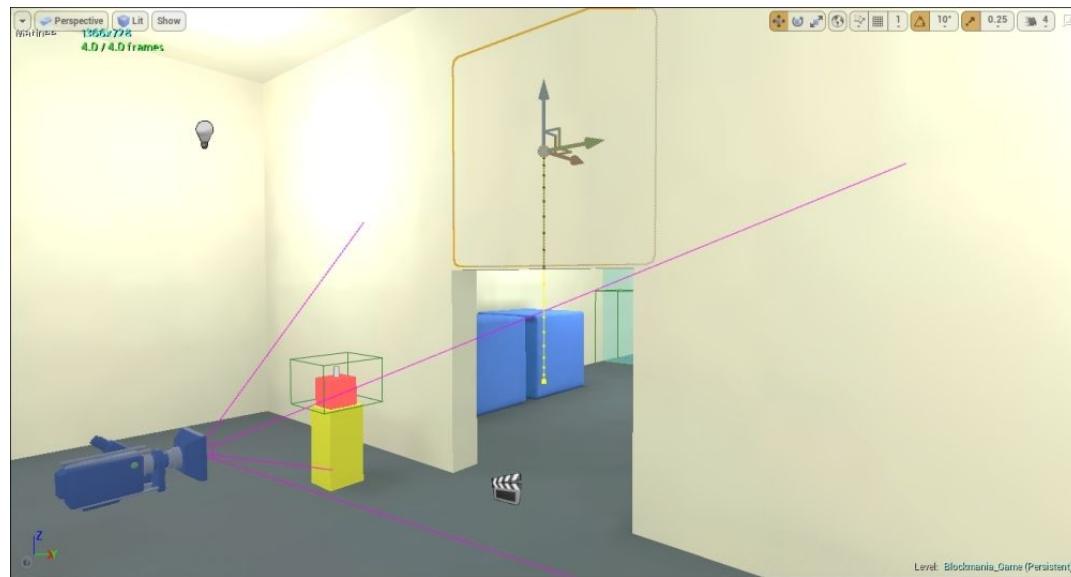
In the timeline, before the Movement track, you will see a small maroon arrow over the 0-second mark. This is a key frame. Since we have a simple animation, we only need two key frames: one at the beginning of the animation sequence and the other at the end (as shown in the preceding screenshot). To add the second key frame, move the animation slider to the end of the animation timeline and press *Enter*.

Now that we have the key frames set up, go back to the Editor.



You will notice something on the top-left corner of the Viewport. Whenever you open Matinee, the Viewport changes to preview the corresponding Matinee animation. At the top left, written in white text is **Matinee**. This is basically notifying that the Viewport is currently set to the preview mode. Next to it, written in teal text is the resolution of the viewport. Below it, written in green text, is the number of frames in the corresponding Matinee animation. At the top-left corner, it reads **Matinee**. Next to it, you can see the resolution, and below it, the total number of frames in the animation. In this mode, you cannot save, open, or load a scene, nor can you play the game. In order to perform any of these actions, you must first close Matinee.

Now, making sure that the timeline slider is at the end of the animation timeline, using the transform tool, move the Door upwards or along the z axis. Move it until it is out of the way.

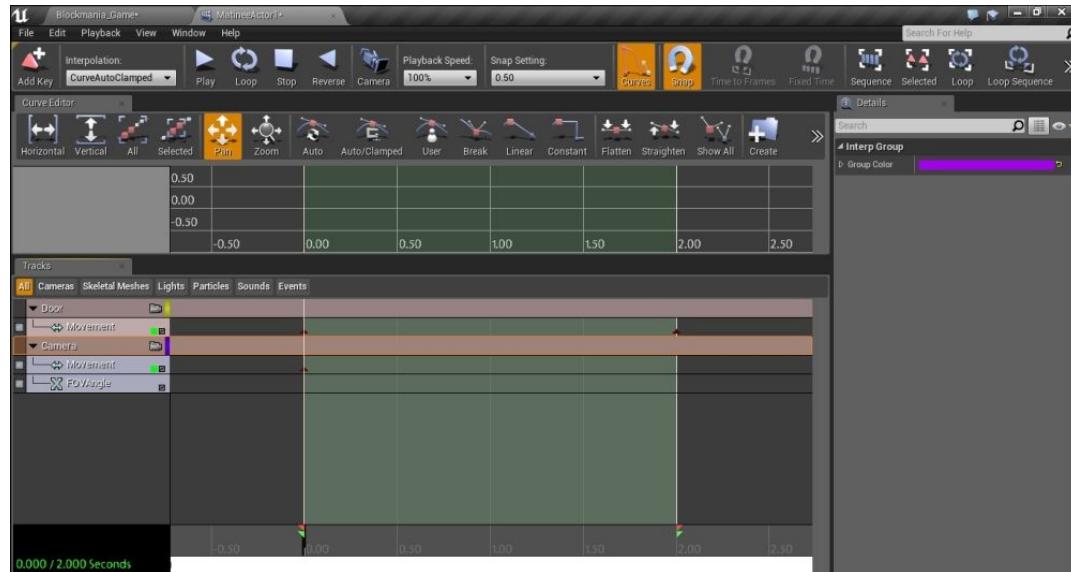


You will see a vertical yellow line whenever you move the door. This line shows the path the door will follow when the animation is played. To view the animation, go back to Matinee and click on the **Play** button in the toolbar.

The next thing we need is to make use of the camera we added earlier. When the player places the key cube on the pedestal, we want to play a small cutscene showing what effect it has, so that the player knows what the main objective is in the rooms. In this cutscene, we will:

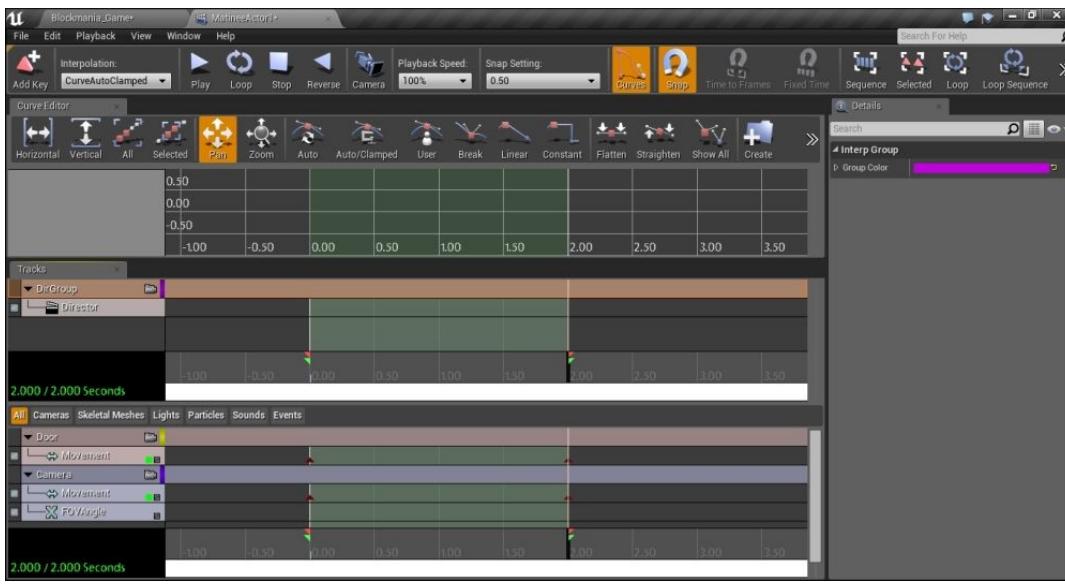
1. Take the control away from the player.
2. Switch from the main camera to the camera in front of the door.
3. Play the animation.
4. Give back the player control over the character.

For this, we will need to add a camera group. Going back to the Matinee window, right-click on the **Tracks** panel and click on **Add New Camera Group** (again, make sure that the camera is selected in the Viewport). Name this group **Camera**.

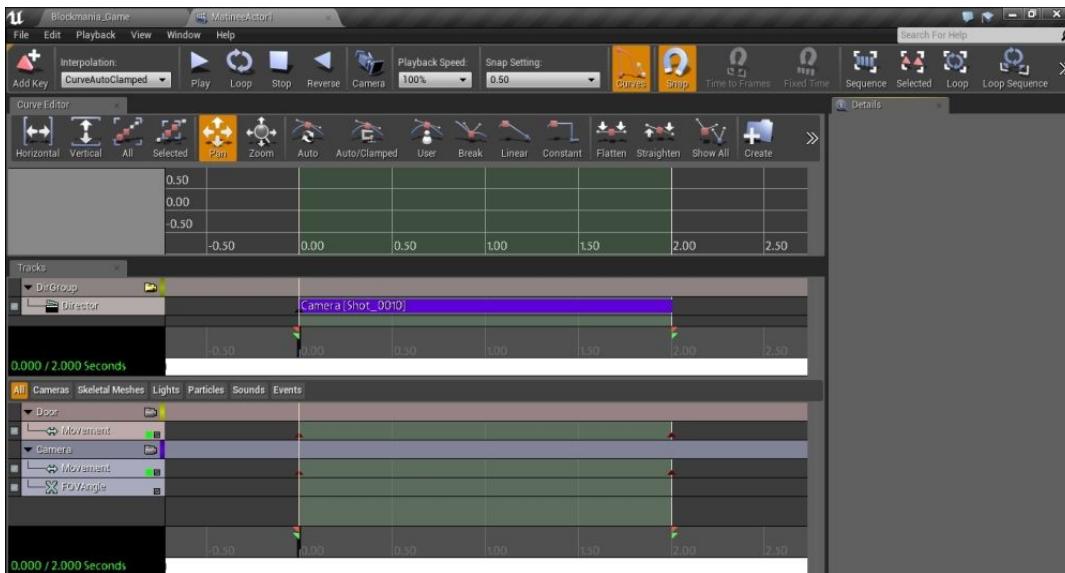


A Camera group has two tracks: a **Movement** track and a **FOVAngle** angle track. You can use this to move the camera and set its field of view when the Matinee is playing. In this case, we want the camera to be stationary. So, take the timeline slider to the end of the timeline and hit **Enter** in the **Movement** track of the camera group to create a key frame.

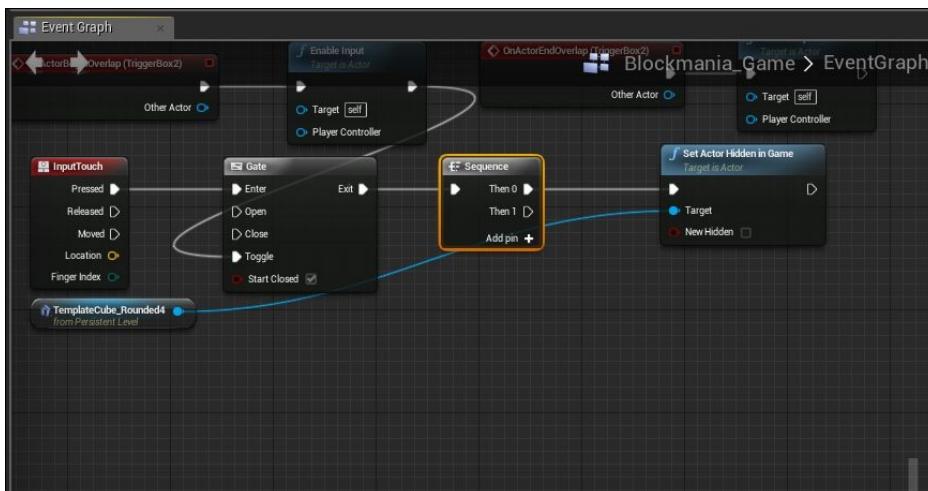
The next thing we need is what is called a **Director** Group. One of its uses is to assign which camera to switch to when playing the animation. Again, right-click on the **Tracks** panel and select **Create New Director Group**. A new, separate track will be created above the **Track** panel for the **Director** Group.



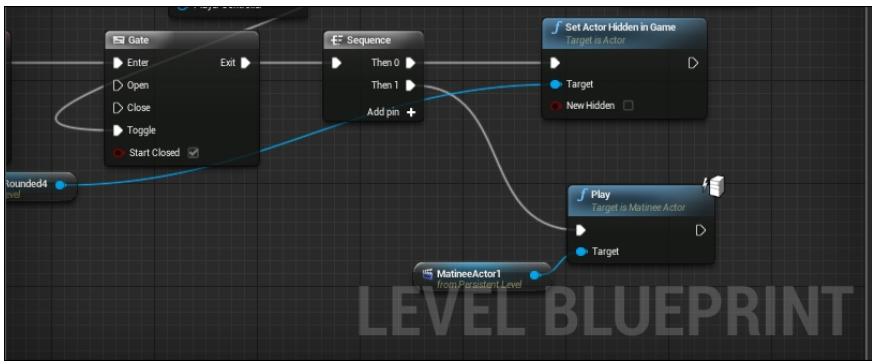
Now that we have our **Director** group, we need to tell it which camera to switch to when the animation is playing. To do so, select it and press *Enter*. This will open a small window asking you which group to cut to. This window enlists all the groups we have created in our Matinee. Here, click on the dropdown menu, select **Camera**, and click **OK** (Be sure that the timeline slider is at the 0-second mark). Once done, the **Camera** group will be assigned to the **Director** group.



Now if you play the Matinee, you will see that in the Viewport, the camera switches to the one we placed in front of the door. We have our Matinee set up. Now we need to script in when it should play. We want it to play when the player places the key cube on the pedestal. So, open Blueprint, and in the sequence we created for the placement of the key cube, add a **Sequence** node. Attach this to the **Gate** node. Next, attach the **Then 0** output pin to the **Set Actor Hidden in Game** node.

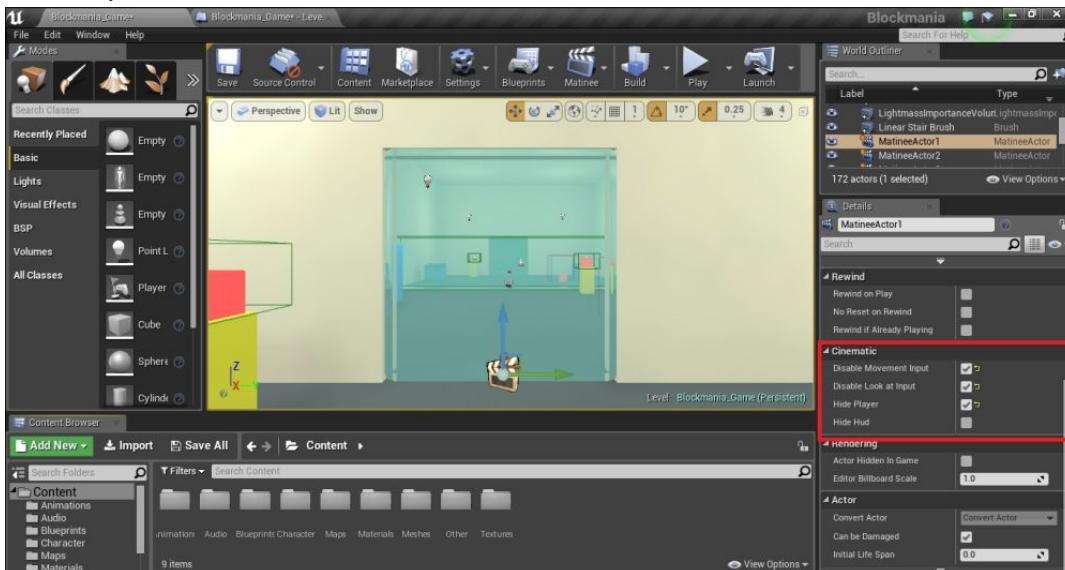


Next, with the **Matinee** actor selected in the Viewport, right-click and create its reference. Then, drag the pin of this node and type in **Play** when the menu opens to create a **Play** node. Attach this to the **Then 1** pin of the **Sequence** node.



If you test out the level, you will see that everything is working as intended; when the player places the key cube on the pedestal, the camera will switch to the one in front of the door, and the animation of the door will start playing. However, there is still something left to do. You can still control the player (move around and shoot) while the cutscene is playing. We want the player to be unable to move or shoot while the animation is playing. To do so, select the **Matinee** actor, and in the **Details** panel, under the **Cinematic** section, check the following:

- Disable Movement Input
- Disable Look at Input
- Hide Player



And there we have it: we have just created our very own cutscene. Let us move on to the large door in the second room.

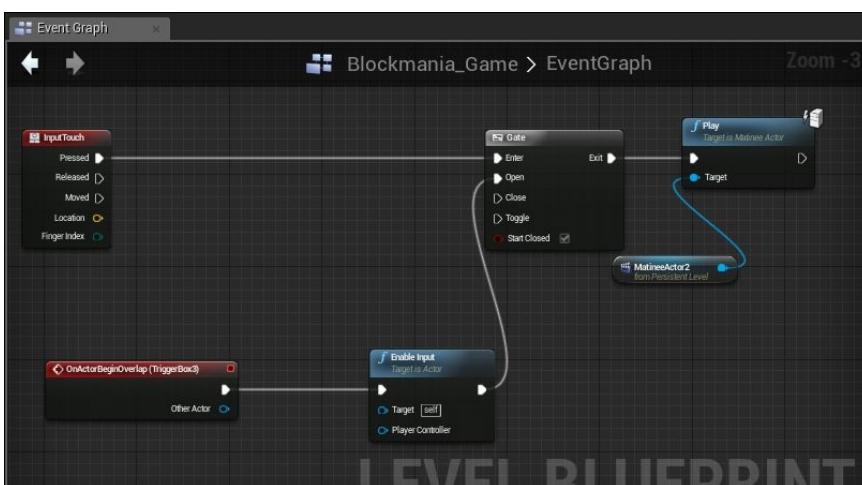
## Room 2

In the middle of the second room, we have a large door. We want the player to be able to use it when they place the first key cube upon the pedestal. So far, it is quite similar to the previous door. The only differences here are as follows:

1. The door opens when the player touches it on the screen (provided it is unlocked first).
2. The door closes when the player releases their finger from the screen.

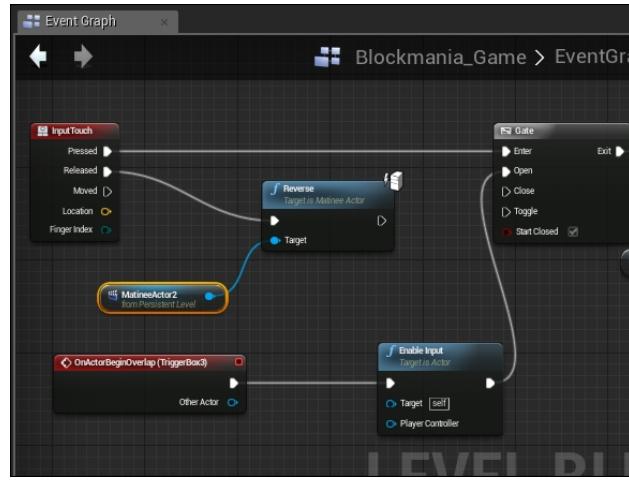
Here, the Matinee part will be similar; we will create a Movement track 2 seconds long and animate the door going upwards. Here, since we are not creating a cut-scene, we do not need a **Camera** group or a **Director** group, neither do we need to disable the player input while the matinee is running. With that in mind, select the **Matinee** actor near the large door and click **Open Matinee** in the **Details** panel. Animate the door using the **Movement** track, as we did with the previous door.

Once done, open Level Blueprint. Here, instead of the cube opening the door, we will make use of the Touch input node. As with the previous triggers, add an overlapping event for the big trigger around the door. Set up the nodes as shown in the following screenshot:



Here, when the player overlaps the trigger, it enables the player input and opens the **Gate** node. When the player now touches the screen, it will play the **Matinee**. The only thing we need to script is what happens when the player releases touching the screen. Here, when the player stops touching the screen, the door will close, that is, the **Matinee** will play in reverse.

To do so, create another reference of the **Matinee** actor (or just duplicate the one already there), create a **Reverse** node, and attach the **Matinee** actor to its **Target** input.



Upon testing the game now, you will see that when you place the key cube on the first pedestal and click on the door (when playing in the Editor, the mouse button acts as a touch input), the door opens; as soon as you let go, it closes again. Do the same with the doors in room 4, but you do not need to unlock the door first; you can simply script it so that it opens when the player is close to it and touches on the screen, and closes when the player releases their finger from the screen. Let us move on to the platforms for the bridge.

### A bridge for the AI character

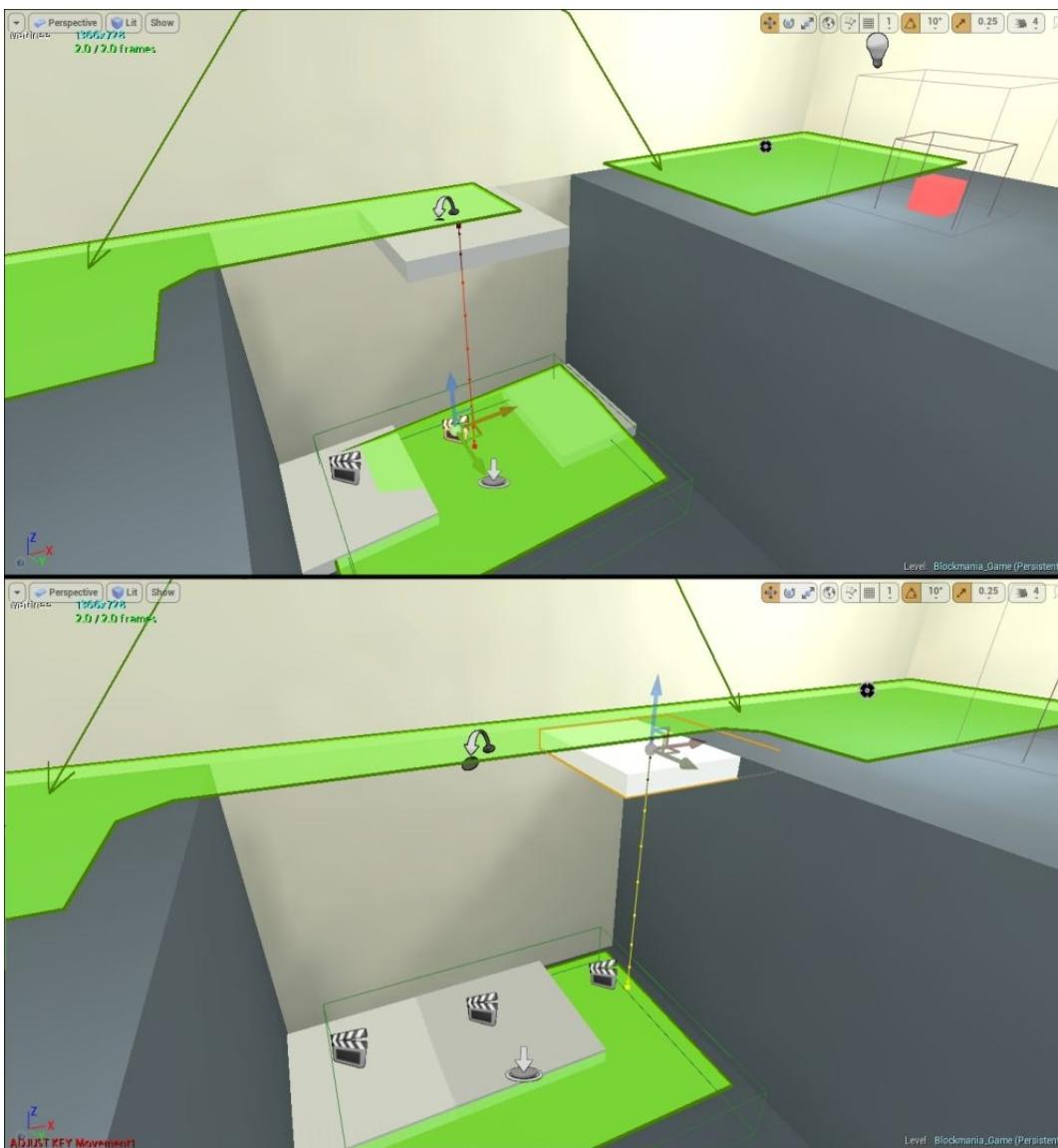
In the room with the AI character, the player has to make a bridge for it, so that it can cross the pit and unlock the key cube. There are switches that draw a segment of the bridge. The player has to quickly draw all of the segments of the bridge before the AI character falls into the pit. In the third room, the bridge will have three parts, each part drawn by the switches on the pedestal. For our bridge, we will use the Cube primitive actor. Add three Cube primitive actors into the scene, set their scale as **2.2**, **2.5**, **0.3** respectively, and place them at the bottom of the pit. Make sure to set their mobility to **Movable**.



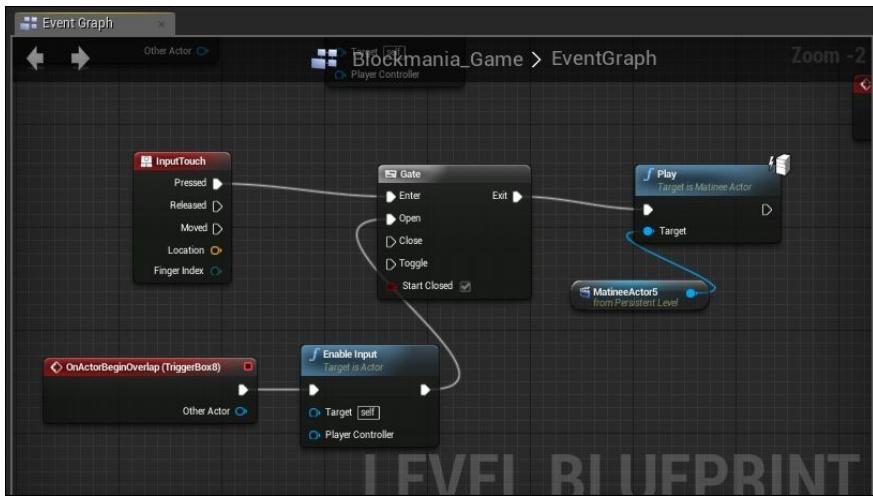
After doing this, add three **Matinee** actors, one for each section. Then, in **Matinee**, set the animation time as 1 second and animate the section coming up by moving it so that it is aligned with the ground. Make sure that the section is perfectly aligned. If it is too high, the AI character will not be able to get on it, and if it is too low, the AI character will not be able to get to the other side. If you see that the upper surface of the section is green (because of the **Nav Mesh Bounds Volume**), it means that the section is walkable for the AI character, as shown in the following screenshot:



Do the same with the remaining two sections, ensuring that they fit perfectly when they are drawn.

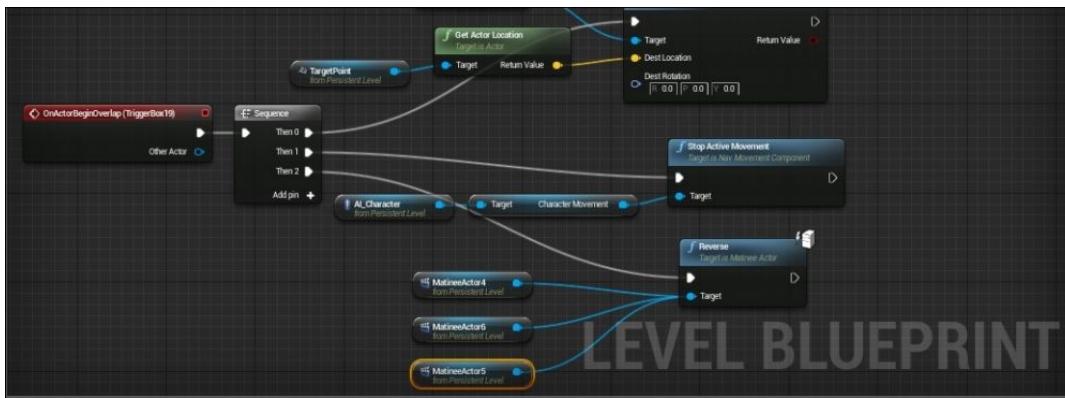


The next thing we need to do is script the Matinee into our game. We have four pedestals in the third room. From left to right, the first one activates the AI character, the second one draws the first section of the bridge, the third one draws the second section, and the fourth one draws the third section. So, open up Level Blueprint, and just as with the large door in the second room, script it in so that the Matinee plays when the player is close enough to the switch and touches on the screen. Do this for all three trigger boxes playing their respective Matinee.



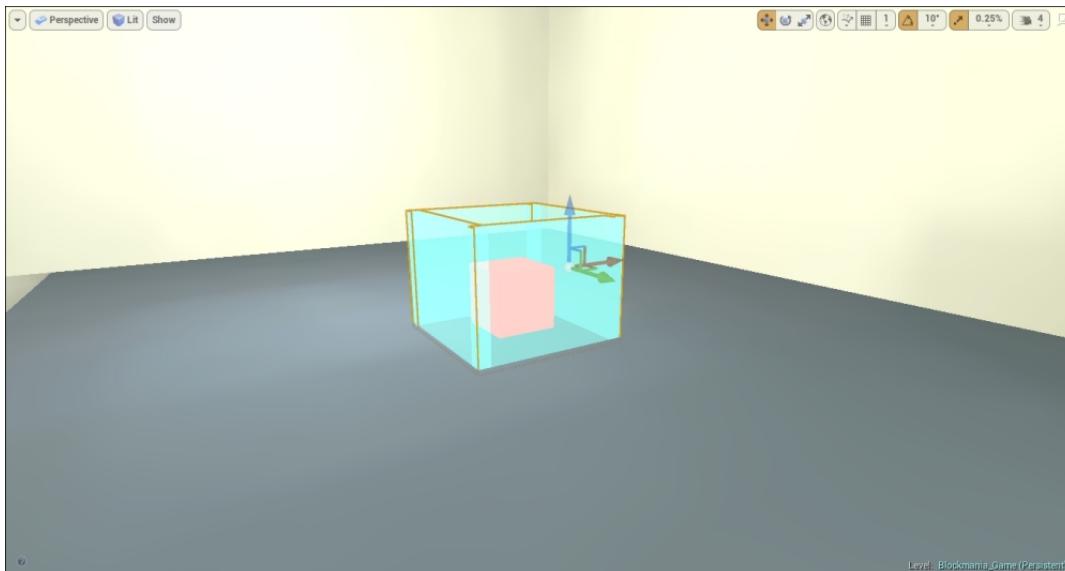
# I FVFI RI UUPRINT

When the AI character falls into the pit, we want the sections of the bridge to go back down to the bottom of the pit, so that when the player starts over, they have to press the switches again to draw the sections. Go back to the trigger box which teleports the AI character back to its starting position. To the **Sequence** node, add a pin, creating a reference for all three Matinee actors. Create a **Reverse** node and connect all three actors to it.

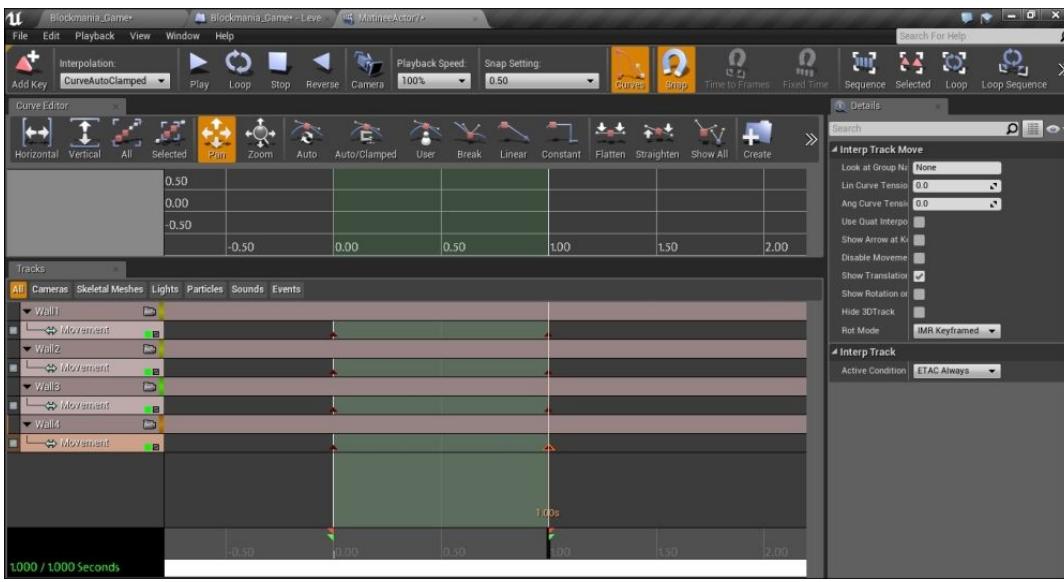


Now if the AI character falls into the pit, apart from being teleported back to its starting position, the sections of the bridge will also reset to their original position.

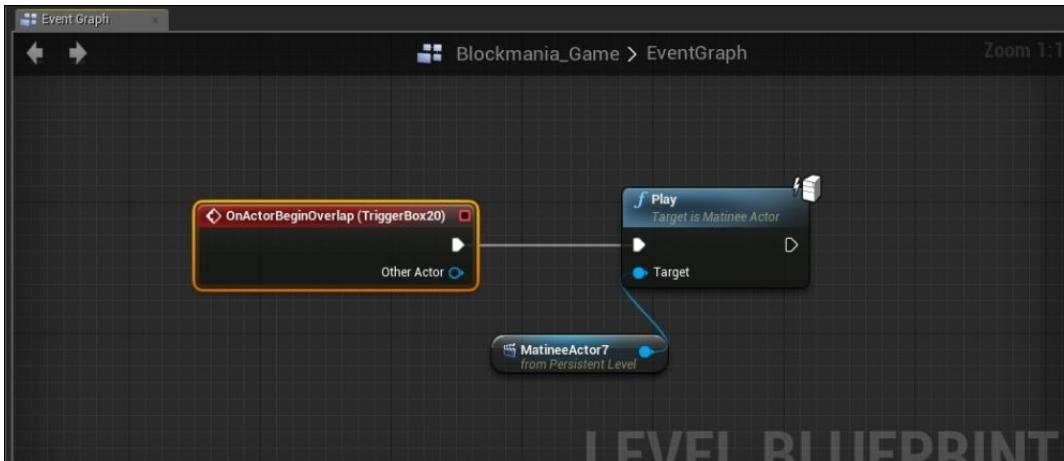
We have our puzzle set up. However, currently the puzzle has no payoff. We had mentioned earlier that when the AI character crosses the pit, it unlocks the key cube. But right now, our key cube does not seem to be locked. What we can do is enclose our key cube, which opens when the AI character successfully crosses the pit. For this, place four cubes around the key cube (which act as walls of the cage), create a Matinee of them opening, and set up a trigger box, which plays the Matinee when the AI character overlaps the trigger box. So, place four cubes around the key cube enclosing it, and apply the door material to them (if you wish, you can duplicate **Door\_Material**, and in the material editor, remove the connections to the **Refraction** expression and apply that to the walls, so that the key cube is properly visible from the other side of the pit). When placing these, ensure that the trigger box around the key cube is also enclosed within the walls.



Now add a Matinee near this cage and place a trigger box over the Target Point actor. Open up Matinee so that we can create our animation. Here, we have four separate objects to animate. However, since they all animate together, it would be wasteful to create four different Matinee actors for each wall. Instead, we will animate all four walls in the same Matinee. For this, we will create four different groups—one for each wall—each with a movement track. Select the first wall, create an empty group (name it **Wall1**) with a Movement track, and animate it going downwards. Set the animation time to 1-second. Then, create another empty group (name it **Wall2**) with a Movement track for the second wall, and animate it going downwards. Do the same for the remaining two walls.



We now have four separate animated objects using the same Matinee actor. The last thing left is to set the overlapping event for the trigger box, which plays the Matinee.



From what you have learned, apply the same method when making the bridges in the fourth room, making sure that the AI character can cross them when they are drawn.

## Summary

In this chapter, we looked at Unreal Matinee: what it is, its UI, and what can be done with it. We used it to create cutscenes and animate doors and bridges. And with this, we have completed our little game, which demonstrates the various features and tools offered by UE4. The next step is finalizing the game (adding in a main menu, polishing the game, and so on), packaging it into an .apk file, porting it to an Android device, and testing it there.

## Chapter 7. Finishing, Packaging, and Publishing the Game

In the last few chapters, we have been building the game bit by bit. We started by building the world using BSP brushes, adding lights and static meshes, creating materials, and applying them to the meshes. We then added the various classes and volumes that were required in order to develop the game and to enhance gameplay. After adding all of the actors, we went on to script our game using Blueprints, Level Blueprints, and Blueprint Classes. Finally, we made use of Matinee to create cut-scenes and animations.

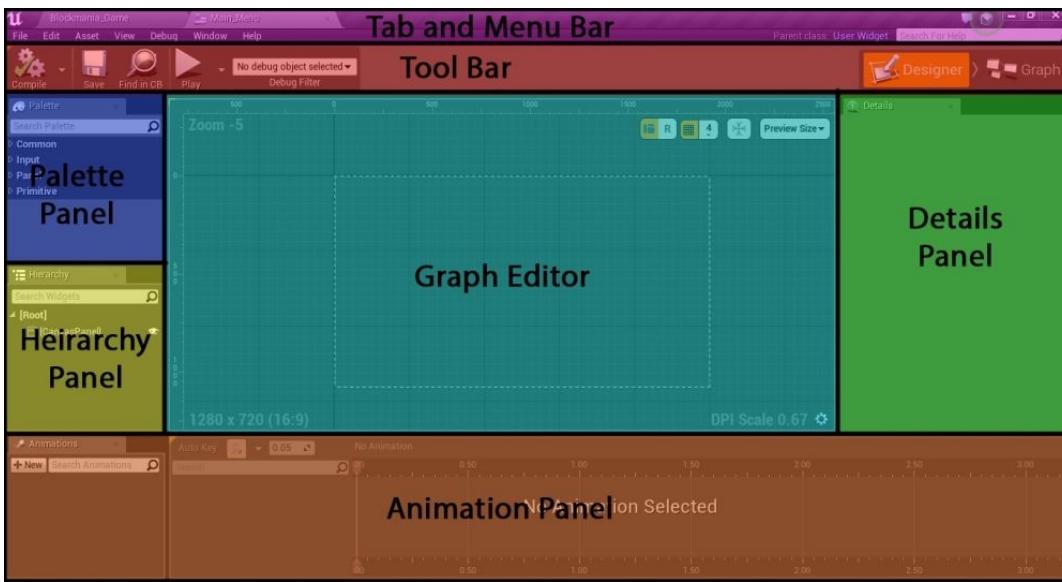
We now have a functioning game. The next step is finalizing the game, packaging it into an .apk file, and publishing it to the Google Play Store.

## Adding the main menu using Unreal Motion Graphics

If you have played any sort of game, you will find that all of them have a main menu. Let's add one to our game as well. Our main menu will be fairly simple. We will have the name of the game in the center. The game will load when the player taps below the name. The game's menus, UI, heads-up display ( HUD ), and so on—along with their functionalities—can be easily created using **Unreal Motion Graphics ( UMG )**. UMG is an easy-to-use tool with a simple and intuitive editor, which you can quickly learn to make your game's menus and such.

## UMG Editor

To access the UMG Editor and make your user interface, you first need to create a **Widget Blueprint** class. Right-click in the **Content Browser**, and under **User Interface**, select **Widget Blueprint**. Name it **Main\_Menu**, and double-click on it to open the **UMG Editor**.



## The tab and menu bar

The tab bar shows the currently open tabs. You can switch, close, rearrange, and move any tab from here.



The menu bar is where you can find the actions you would use:

- **File** : From here, you can save your widget blueprint, compile the blueprint you created, open a selected asset, and so on.
- **Edit** : Here, you can undo or redo the last action and open the **Editor and Project Preferences**.
- **Asset** : From here, you can find any asset you have selected in the **Content Browser** as well as find said asset's references in the widget.
- **View** : Here, you can choose to hide/unhide any unused pins that are present in the blueprint.
- **Debug** : If you have created breakpoints in your blueprint (to debug errors), then you can enable/disable them and remove all of them from here.
- **Window** : From here, you can choose which panel or window you wish to see and which you do not.
- **Help** : Finally, should you need any tutorials or documentation regarding widgets, you can access them from here.

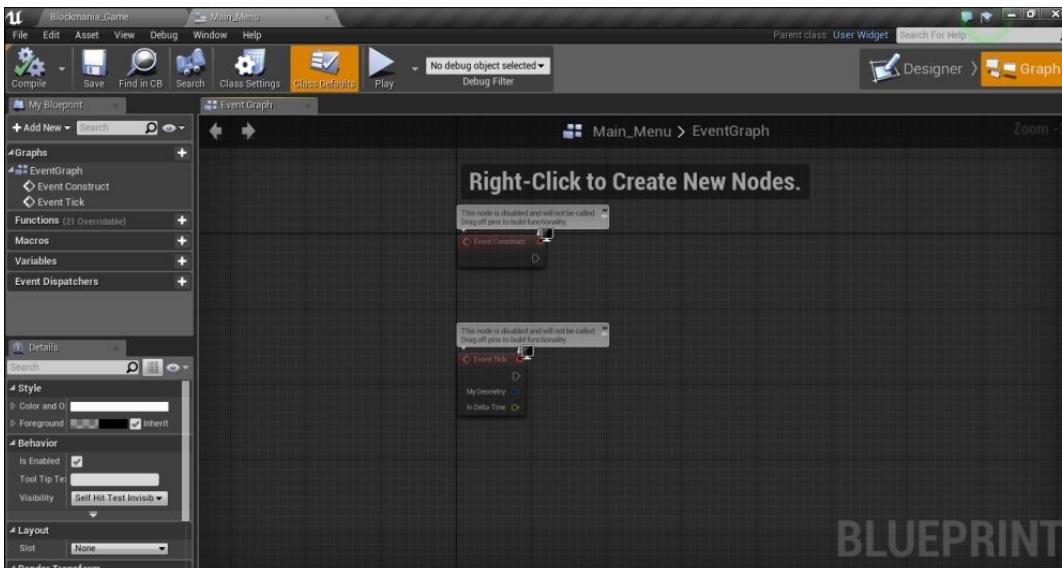
## The toolbar

Below the menu bar is the toolbar. Here, all of the most commonly used actions are listed.



- **Compile** : Clicking on this option compiles the class and notifies the user of any errors and/or warnings
- **Save** : This saves all of the modifications you have made to the widget class
- **Find in CB** : This option locates the selected object in the Content Browser
- **Play** : This plays the game in the Editor Viewport
- **Debug Filter** : All of the variables and/or nodes you have set to debug are listed here

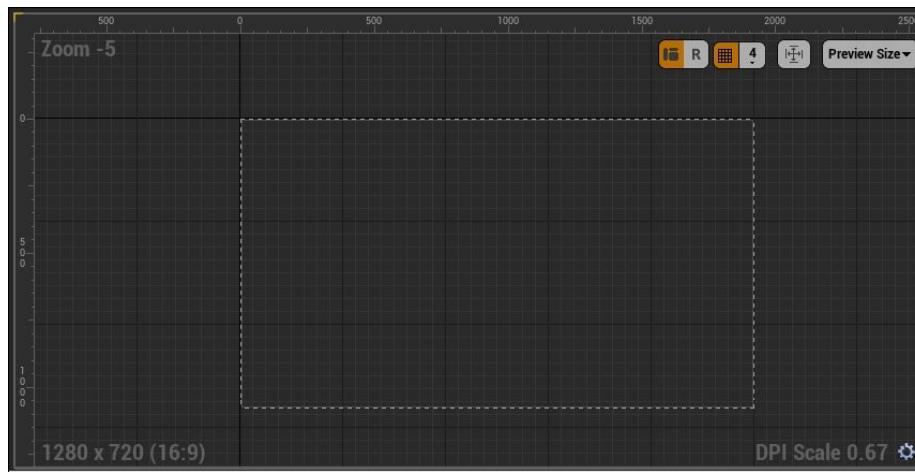
There are two windows in the UMG Editor: **Designer** and **Graph**. The **Designer** window is the window you see when you open the widget class. This is where you design your user interface. This includes adding the UI elements, arranging them in the Graph Editor, setting their properties, and so on. The other window is the **Graph** Window, which looks like the **Level Blueprint** window. This is where you do all of the visual scripting for the user interface.



All of the menus have the same actions that you would find in any other Blueprint window.

## The Graph Editor

In the middle of the screen is the Graph Editor. Here, you make and design your UI.



At the center, you can see a dotted rectangle. This is called the canvas and it represents the game screen. All of the UI elements you want in your widget go in here. This shows where the elements will be, how they will look on the screen, and where they will be arranged on the game screen.

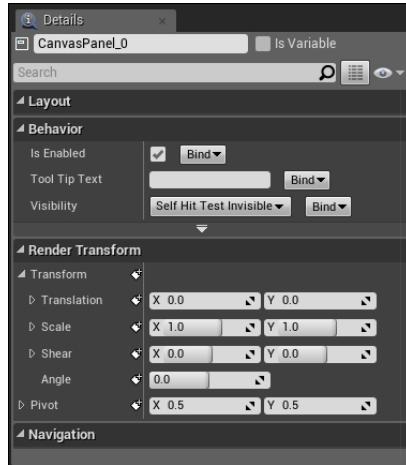
At the bottom-left corner of the Graph Editor, you can see the resolution of the screen. At the bottom-right corner of the screen is where the **Dots per inch (DPI)** scale is displayed. You can change this by clicking the gear icon next to it.

At the top-left corner, you can see the zoom scale. Finally, at the top-right corner are a few buttons. From left to right, they have been enlisted as follows:

- **Widget Layout Transformation** : This is used to transform and set the layout of the widgets you created for your UI.
- **Widget Render Transformation** : This is to transform the entire UI itself.
- **Grid Snapping** : This toggles the grid snapping on/off.
- **Grid Snap Value** : If enabled, you can set the snap value from here.
- **Zoom to Fit** : This pans and adjusts the Graph Editor to fit the entire canvas within the Graph Editor.
- **Preview Size** : Here, you can set and see how the UI would look on different devices, each with a different screen size, DPI, and screen resolution. It is really handy when you are developing games for different devices.

## The Details panel

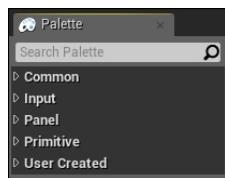
The **Details** panel is where you can set the properties of the selected component.



Here, you can set properties such as transformation and the pivot of the widget, add events when the widget is pressed or hovered over, set the visibility, and so on.

## The Palette panel

There are various widget elements that go into making a UI. You can find all of them in the **Palette** panel. Just drag and drop what you need from the panel to the Graph Editor in order to add that element.



The widget components are categorized into four groups, namely **Common**, **Input**, **Panel**, and **Primitive** :

- **Common** : This group contains the most frequently used widgets, such as buttons, image box, sliders, progress bar, and so on.
- **Input** : This group contains elements that can take in input from the player, for instance, a text box, spin box, and combo box.
- **Panel** : This group contains elements useful for laying out widgets and for controlling when placing widgets.
- **Primitive** : This group contains things such as Trobbers, Editable Texts, and so on.
- **User Created** : This section contains any Widget Blueprint that you have created or have in your Project file. You can add them from here.

## The Hierarchy panel

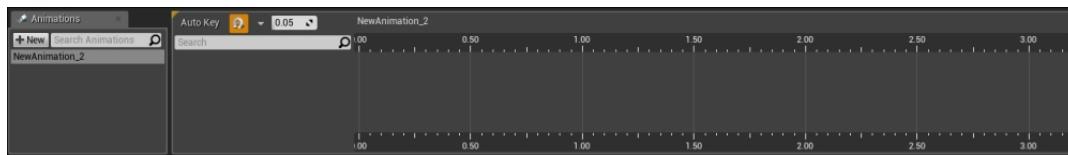
The **Hierarchy** panel shows the hierarchy of all of the components of the widgets.



At the top is the **CanvasPanel** which acts as the foundation of the UI. Whenever you add a component, it is added to the hierarchy as well. If you drag and drop a component on a Widget Component in the **Hierarchy** panel, it is added to it with the component upon which you dropped the widget component acting as the parent and the component you added acting as the child. They also get attached together in the Graph Editor.

## The Animations panel

The final component in the UMG's user interface is the **Animations** panel, located at the bottom of the window.

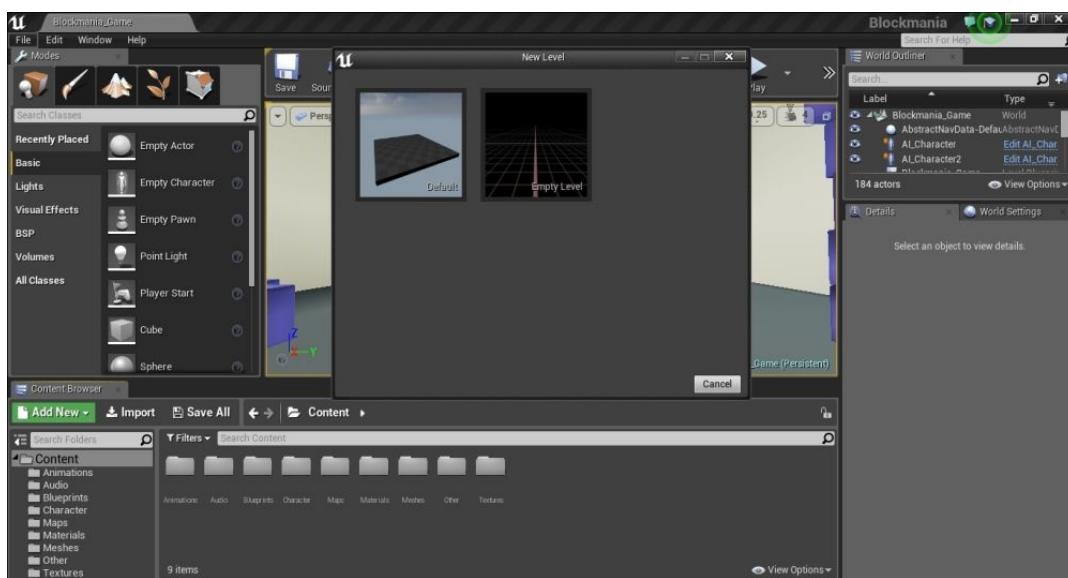


UMG allows the user to create animations. You can animate the Widget components to suit your preference. The **Animation** panel works almost the same way as Unreal Matinee. On the left is the **Tracks** panel. This is where you can add, remove, or move animation tracks. To add an animation track, click on the **+ New** button.

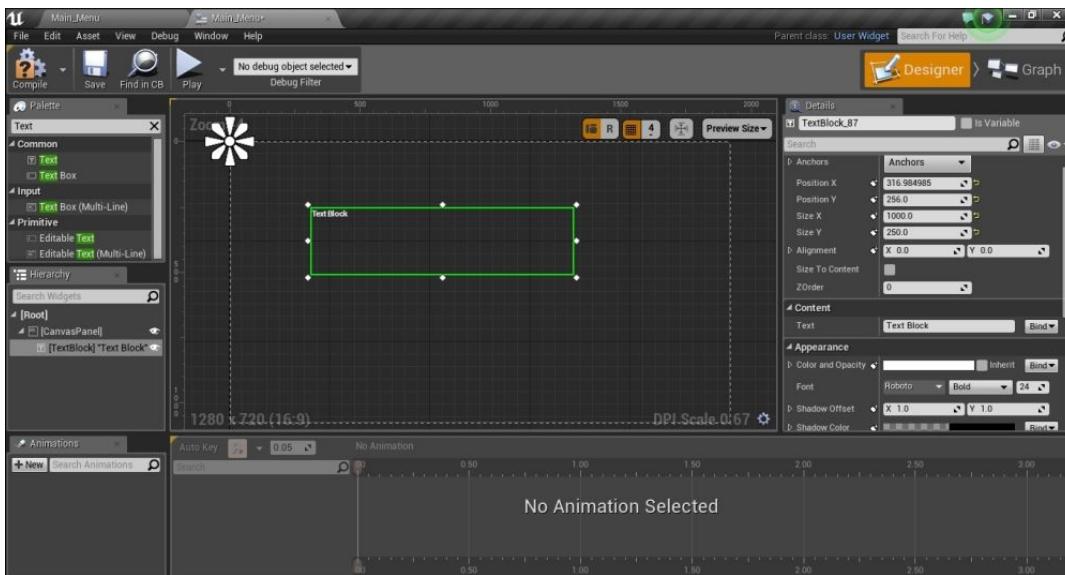
On the right is the animation timeline. This is where you create your animation using key frames. At the top-left corner are some settings, such as enabling/disabling grid snapping, the grid snap value, and so on.

## Creating the main menu

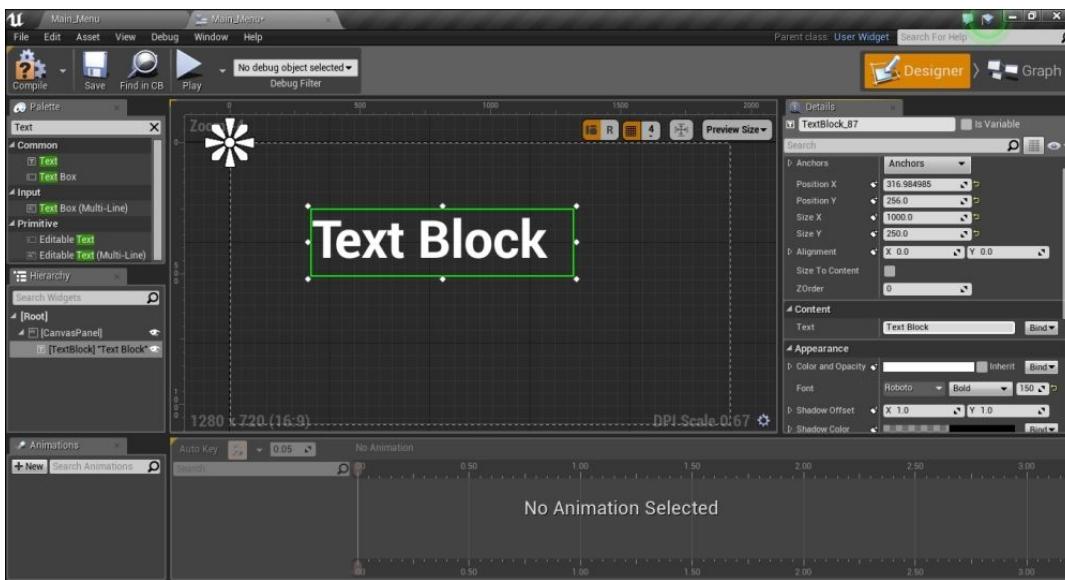
The first step to creating our main menu is to create a new level. In the Editor, click on **File** and select **New Level**. When the Level Type window opens, select **Empty Level**.



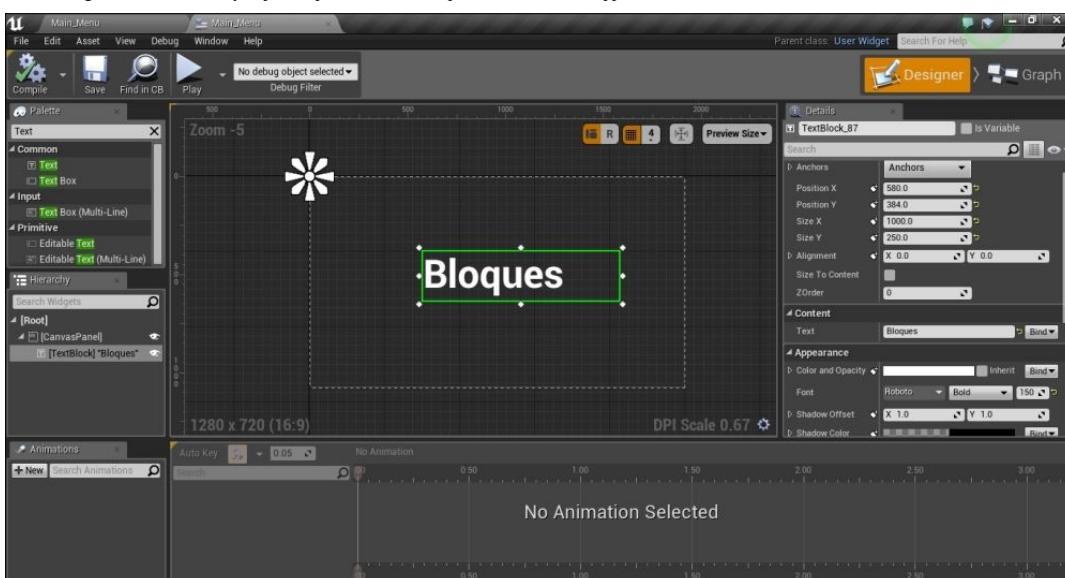
Name this level as **Main\_Menu**. Once created, open the **Main\_Menu** widget class in the **Content Browser**. In the UMG Editor, add a **Text** widget to the canvas. Make the text big. First, increase the size of the text slot panel to 1300 x 200. There are two ways of doing this: the first way is to click on the edge and drag it to increase the size. The second way is in the **Details** panel, where you can set **Size X** to 1000 and **Size Y** to 250.



The text slot is now set. However, the actual size of the text itself is quite small. The next thing we need to do is to increase the size of the text. Again in the **Details** panel, under the **Appearance** section, you will find the **Font** option. At the far right, you can set the value of the font size. By default, this value is set as 24. Increase this to 150.

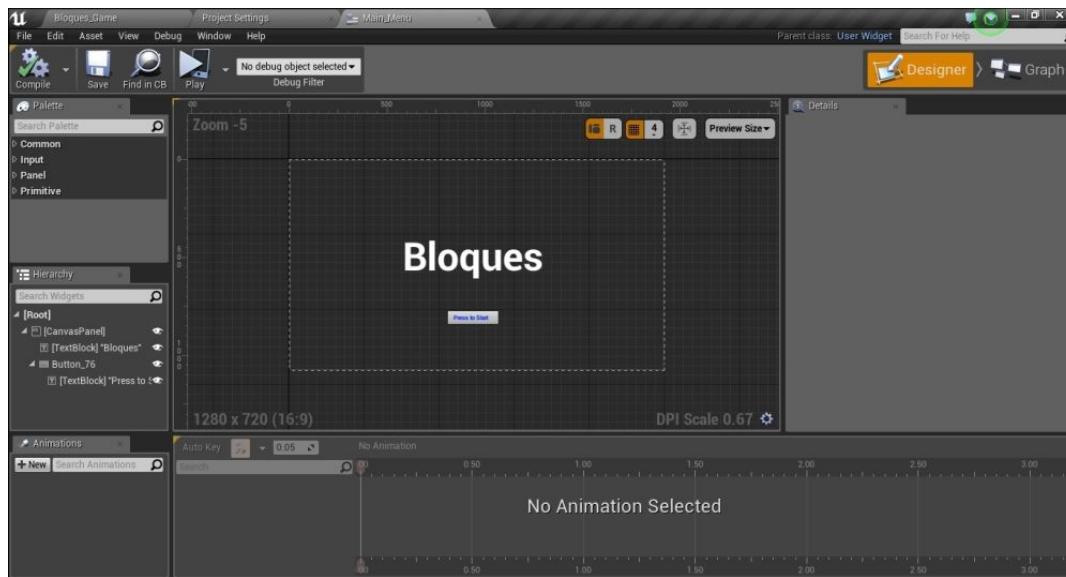


This is where we will display the name of the game. In the **Content** section, you can set what text you want printed on the screen. Whatever you write is displayed in the Graph Editor. Remove **Text Block**, and write the name of the game: **Bloques**. Finally, adjust the position of the **Text** panel so that the name appears in the center of the screen.

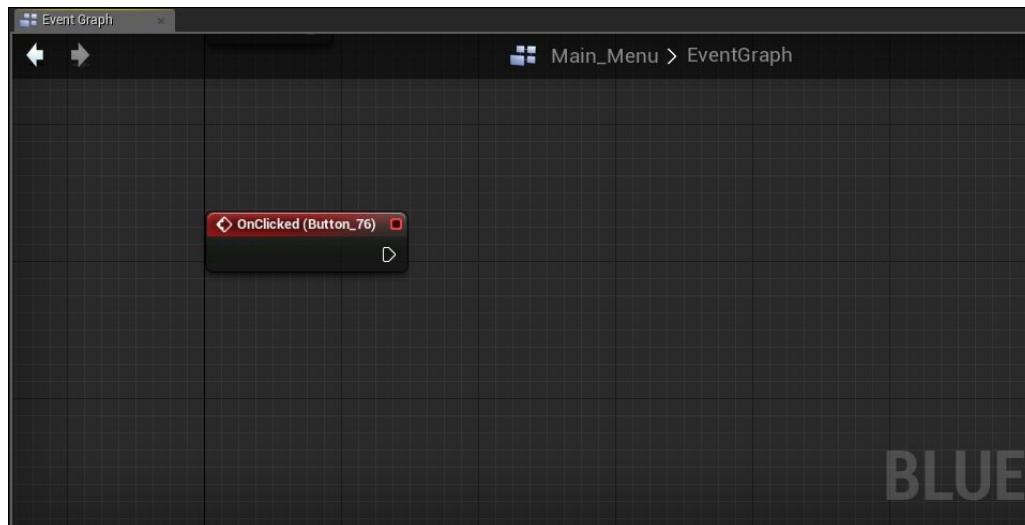


The next thing we are going to add is a button which, when the player clicks, starts the game. To do this, add a **Button** widget from the **Common** section in the **Palette** panel. Place it below the name of the game in the center. Set its dimensions as 256 x 64. We will also need some text to go on the button, so drag a **Text** component from the **Palette** panel and drop it over the button. The text will get attached to the **Button** widget.

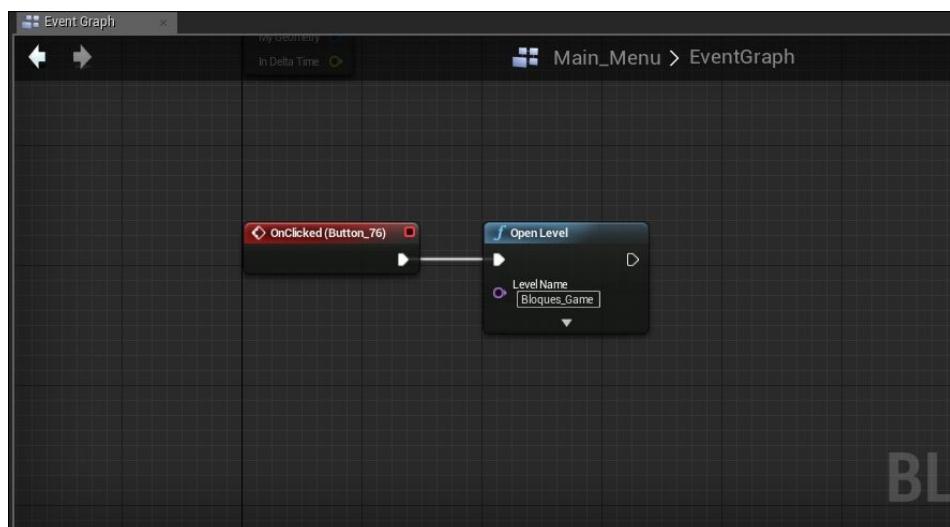
You can see in the **Hierarchy** panel that the **Text** widget is under the **Button** widget as a component. Whenever you move the **Button** widget, the **Text** widget moves along with it. Change the color of the font from white to blue so that it is clearly visible.



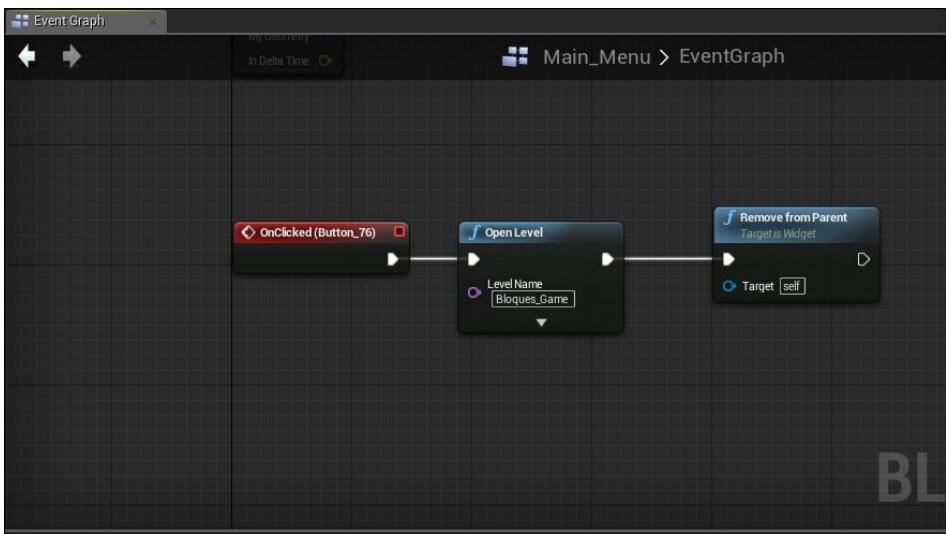
We have our button in place but it does not do anything. We need to script its functionality in. With the button selected, go to the **Details** panel and under the **Events** section, click on the button next to **OnClicked**. The window will switch to the **Graph** window with the **OnClicked** button event node already set up.



To this node, we will attach an **OpenLevel** node. Right-click and search for it by typing it in the search bar and attach it to the **OnClicked** node. In the **Level Name** input, write in the name of the level you want to open which in our case is **Bloques\_Game**. Make sure you copy the name of the level properly; otherwise, it will not open.

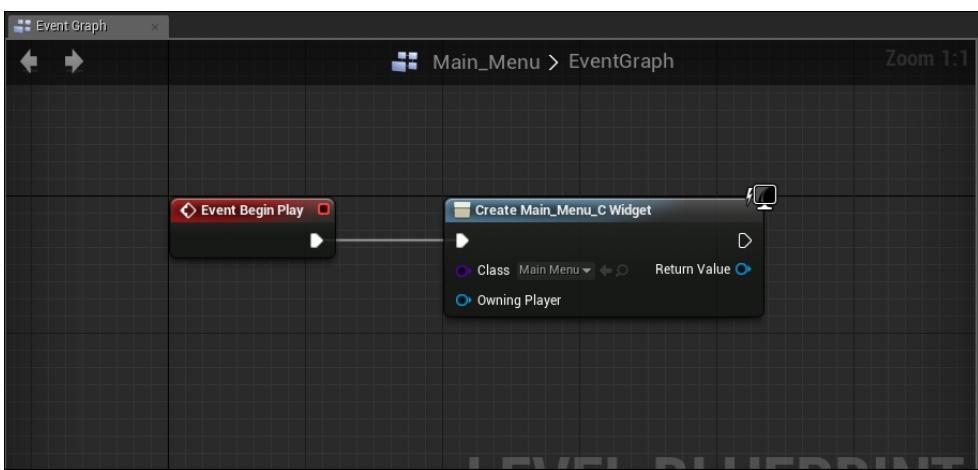


Finally, we also need to remove the widget; otherwise, it will still be there when the level loads. Right-click and select the **Remove from Parent** node and connect it to the output pin of the **Open Level** node.

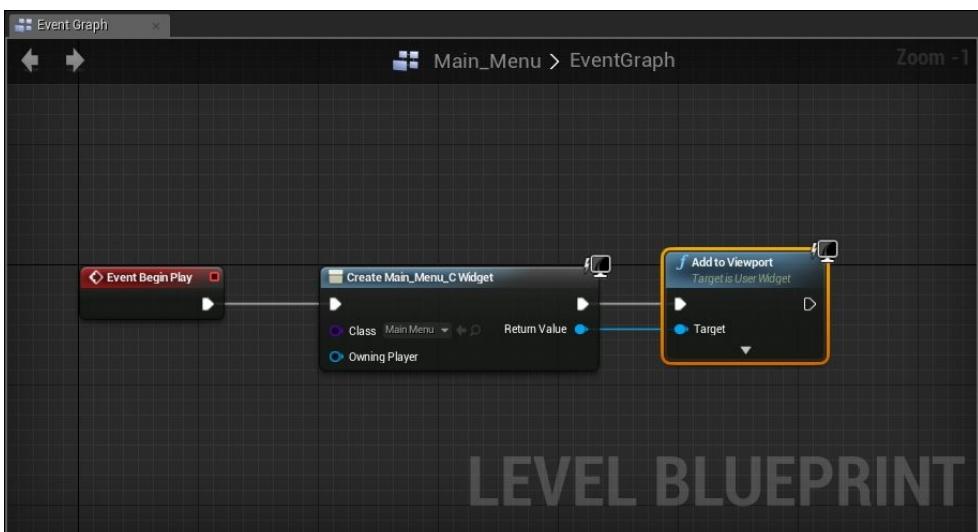


That is it. It really is that simple to load levels via Blueprints.

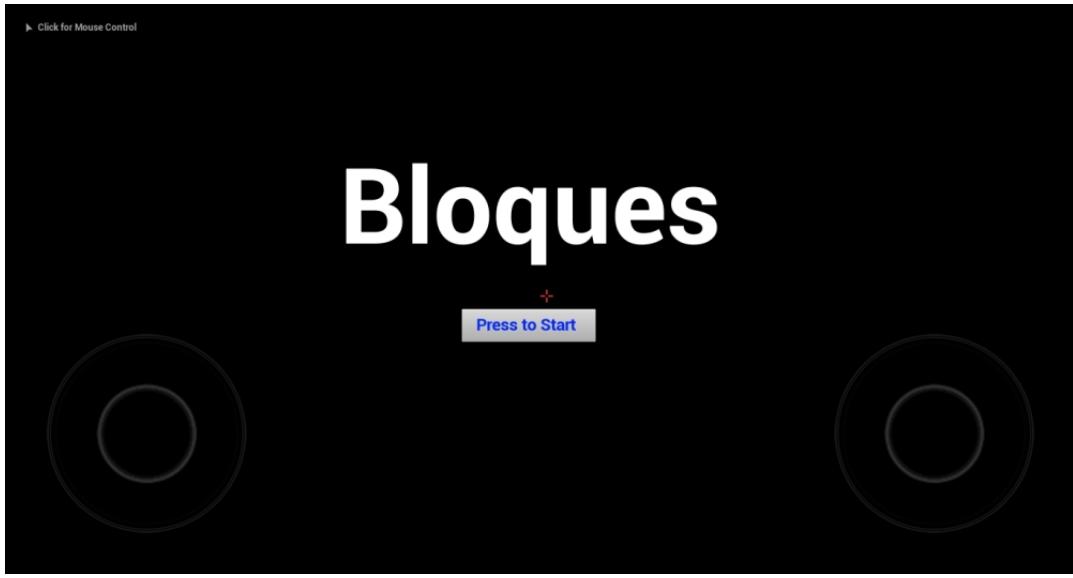
We have our Main Menu set up. We have to implement it into our scene. For this, open **Level Blueprint**. Before we can display the widget, we first need to create it. In the **Event Begin Play** section, right-click and type `create widget` to find the Create Widget node and connect them. Next, in the **Class** input, click on the dropdown menu and select **Main\_Menu**.



When the level begins, the **Main\_Menu** widget will be created. Once created, we need to add it to the Viewport so that the player can see it. For this, we need an **Add to Viewport** node. Drag the **Return Value** out and release it anywhere in the Graph Editor. When the menu opens, type `in Add to Viewport`, find it, and add it.

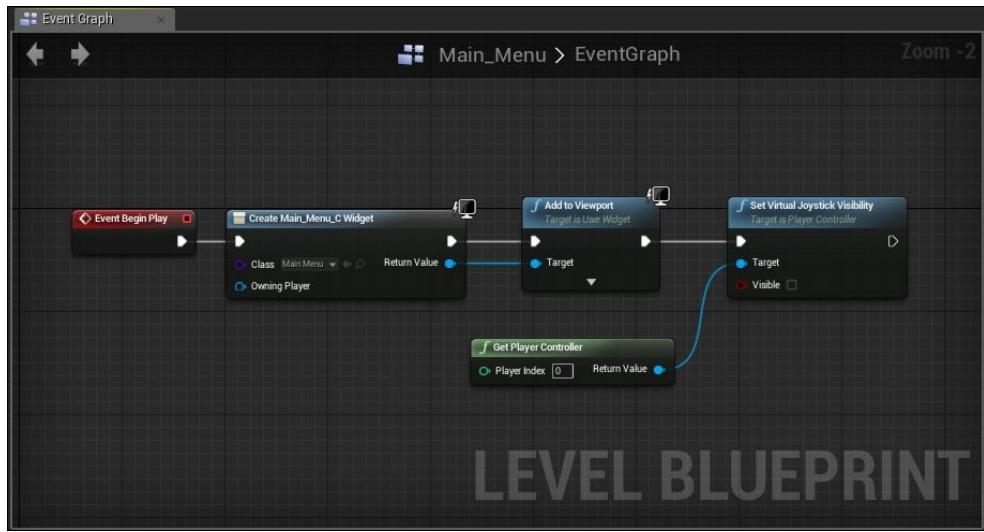


You were to test out the game, you would see that as soon as you run the game, the name of the game and the button appear on the screen.

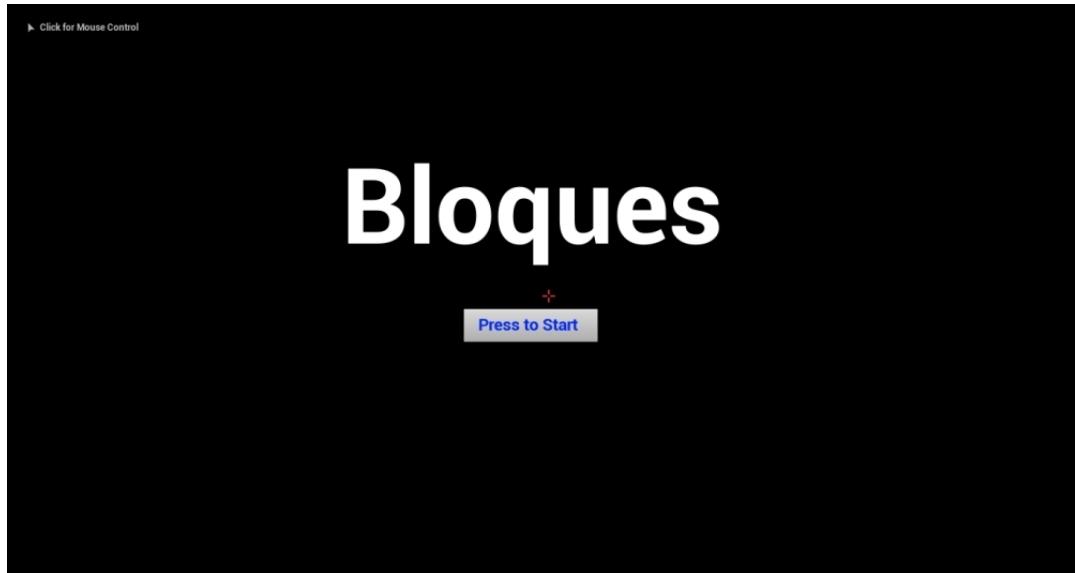


However, there are still a few things we can improve. For starters, we can hide the virtual joysticks, since they are not required.

Right-click anywhere in the Graph Editor; turn off **Context Sensitive**; find the **Set Virtual Joystick Visibility** node; add it to the Graph Editor; and connect it to the **Add to Viewport** node. For the **Target** input, you need to create a **Get Player Controller** and attach it to the **Target** input pin.



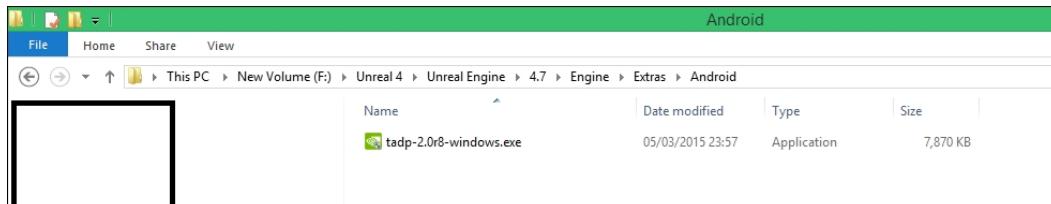
Now run the game. The virtual joysticks will not be visible during runtime.



If you press on the button, it will load and take you to the game, and as soon as the level loads the UI will disappear. This is how you add a main menu to your game.

## Installing the Android SDK

Before we can cook and package our game, we first need to install the Android SDK. Luckily, you do not need to find it on the Internet and download it; its installation file is available with UE4. You can find it in <Location>\Unreal Engine\4.7\Engine\Extras\Android. The file we need is `tadp-2.0r8-windows`.

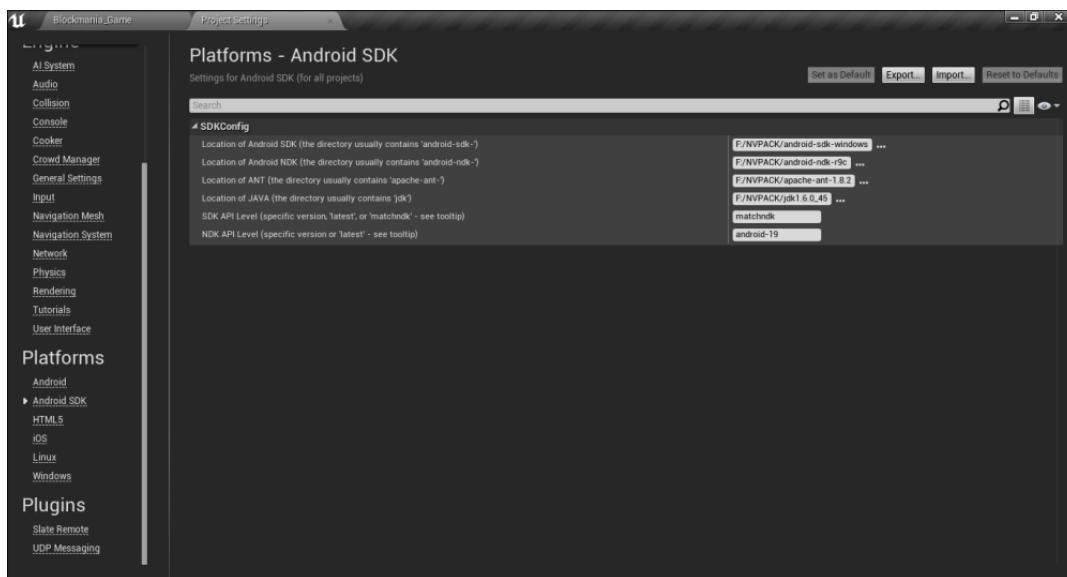


Once you find it, double-click on it to run the installation. Once the setup opens, all you need to do is to follow the onscreen instructions and let it install.

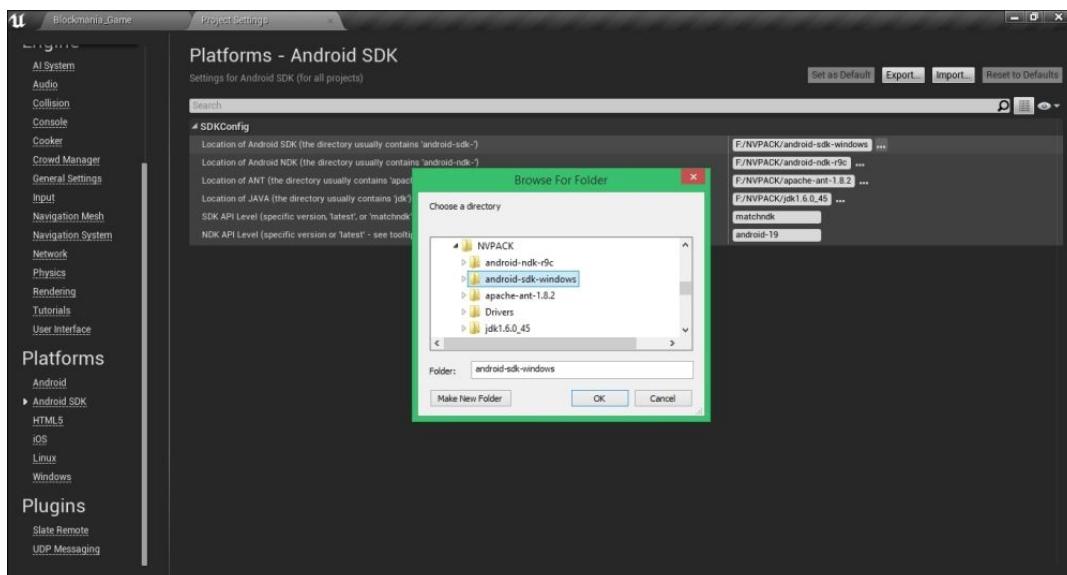
#### Note

One thing to note is that the setup needs an Internet connection since it downloads the build tools; so make sure that you are connected to the Internet when installing the SDK tools.

Once you have successfully installed the Android SDK files, the next thing you need to do is to tell the engine where all the files are (especially if you have installed the SDK files in some other location than the default location). To do this, you first need to open **Project Settings** (which can be found in the Viewport Menu Bar under **Edit**). In the **Project Settings** window under the **Platforms** section, select **Android SDK**.



Here, you can tell the engine where all of the folders are located. On the left is a list of all the files required to build for Android. On the right, in the panel, is where the directory for the corresponding files is set. There are two ways to change this. You can either manually type in the location of the folder in the panel, or click on the ... next to the panel. When you click on it, the **Browse For Folder** window opens up where you can specify the folder.



- The first option is for the Android SDK folder. Find the `android-sdk-windows` folder and set it.
- The second option is for the android NDK files. Find the `android-ndk-r9c` folder and set it.
- The third option is for the ANT files. Find the `apache-ant-1.8.2` folder and set it.
- The fourth option is for the Java files. Find the `jdk1.6.0_45` folder and set it. For developers who already have the Java SDK installed, you need to check the system environment variables and that the paths are correct.

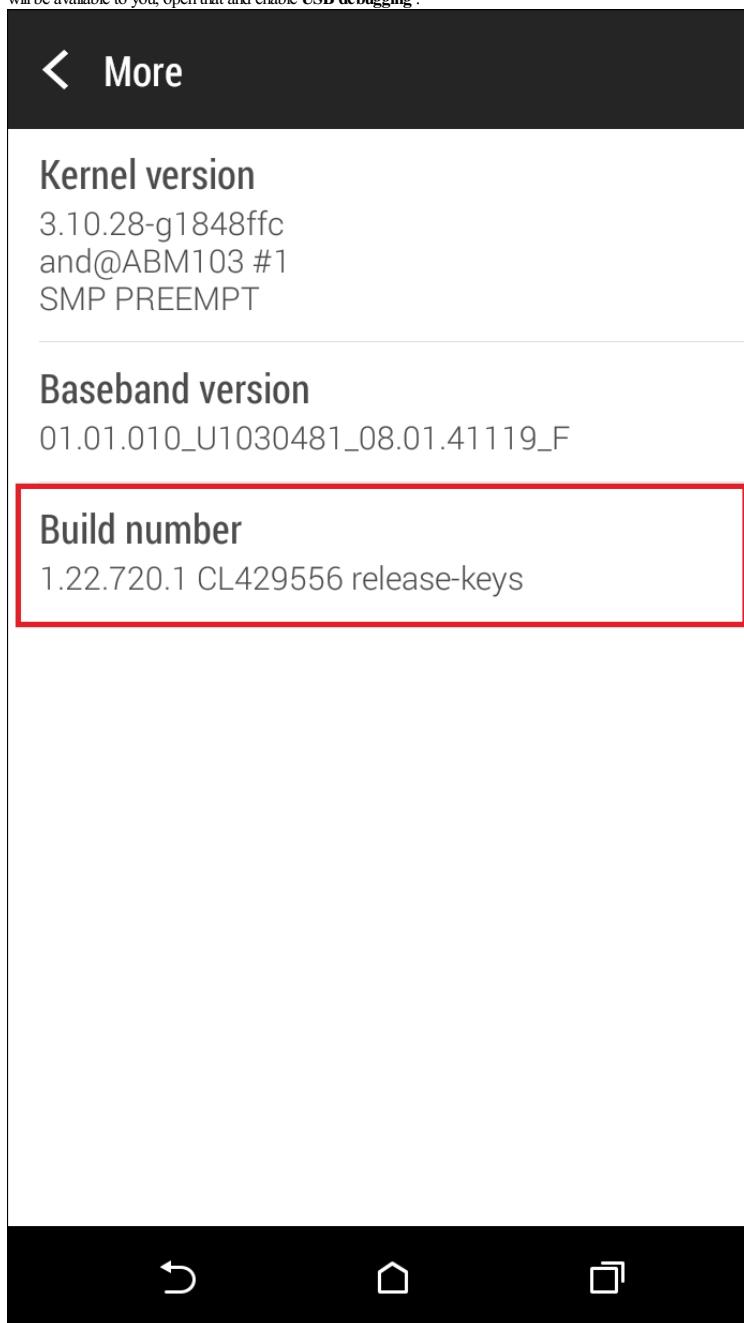
You can have different folder names; you just need to make sure that the appropriate folder is currently assigned.

The next two options are to specify which version of the SDK and NDK tools to use. The Android SDK files have now been installed and set up. The next thing to do is set up your Android device for testing.

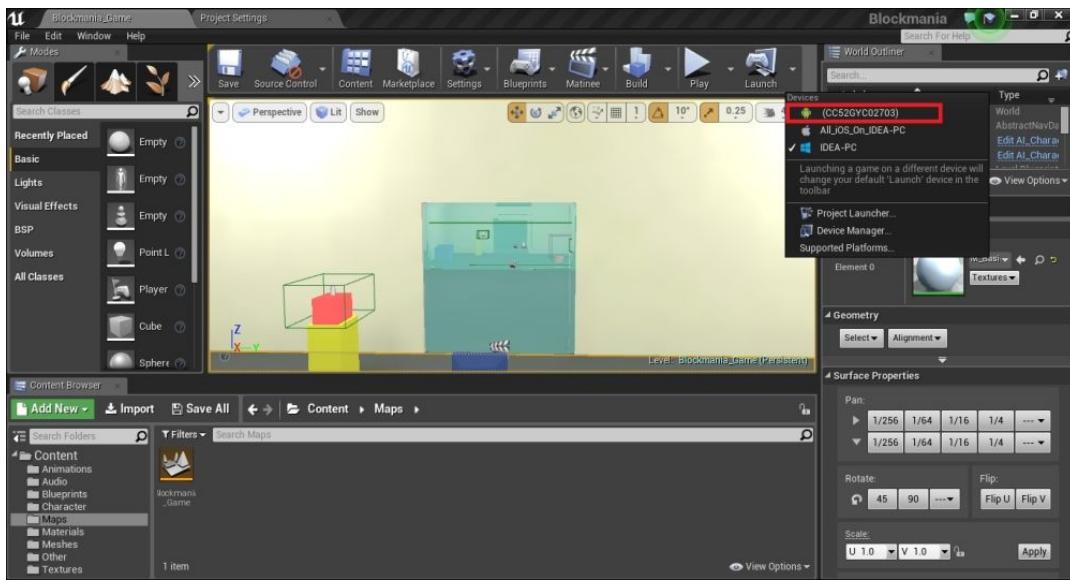
## Setting up the Android device

When developing games, it is natural that you would want to test your game from time to time to see whether it is working smoothly and properly on the device. Instead of having to first package your game, put it on your device, and then test it, you have the option to directly build and run the game on Android devices.

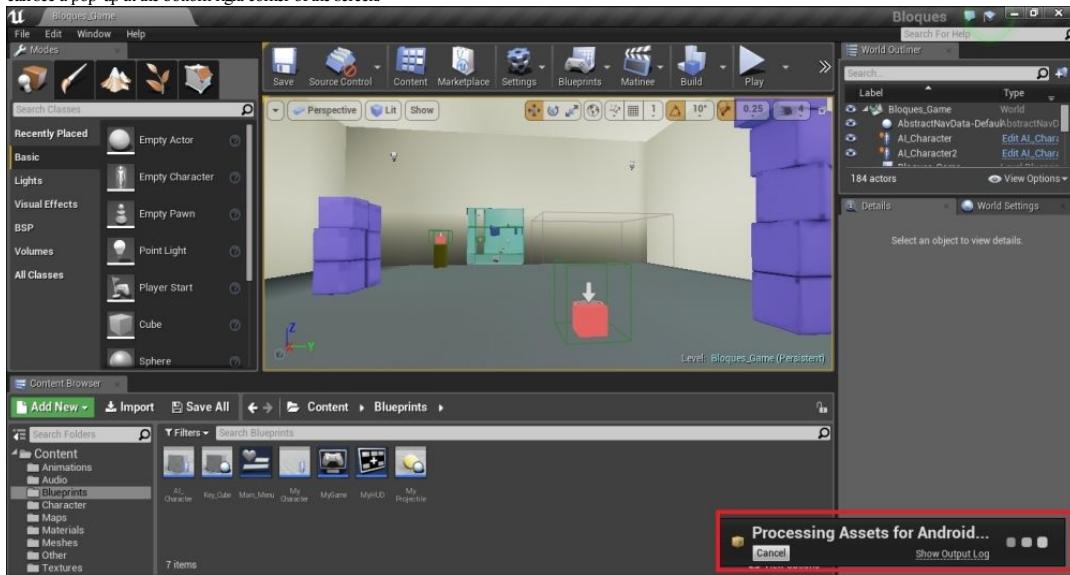
1. To do this, the first thing is to enable the Developer mode on your device. This can usually be found in **Settings | About phone**. Here, you need to find **Build number** and tap on it several times. If done correctly, you will get a prompt saying **You are now a developer** (in some devices or versions of Android, the **Build number** option is located within **More**). Once that is done, a new section called **Developer Options** will be available to you; open that and enable **USB debugging**.



2. Once done, connect your device to the system and let it install the drivers. Your device is now ready for testing. To check whether it has been properly set up, in the **Editor Viewport Toolbar** click on the little arrow next to **Launch** and see whether your Android device is listed under **Devices**. If it is, that means that it has been properly set up. Some devices do not automatically install the driver. In such cases, it is advised that you download the respective driver from the company's website.



3. Once set up, we can now test our game on our Android device. To do so, click on the Android device listed under **Devices**. Once clicked, UE4 will start packaging the game and deploying it for the device. You can see a pop-up at the bottom-right corner of the screen.



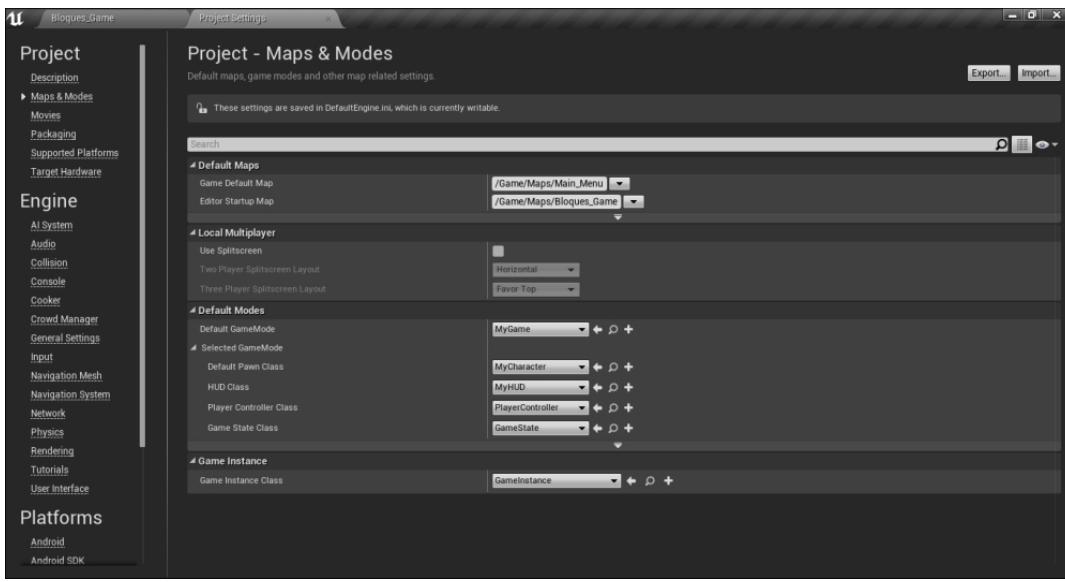
4. To stop the build process, click on the **Cancel** button. To see the output log of the building process, click on **Show Output Log** which opens the log window, where you can see what is being executed. You can also see where the error has happened if one occurs. Once the build process is over, it will automatically start on your device. The .apk file will also be installed.

## Packaging the project

Another way of packaging the game and testing it on your device is to first package the game, import it to the device, install it, and then play it. But first, we should discuss some settings regarding packaging, and packaging for Android.

### The Maps & Modes settings

These settings deal with the maps (scenes) and the game mode of the final game. In the Editor, click on **Edit** and select **Project** settings. In the **Project** settings, **Project** category, select **Maps & Modes**.

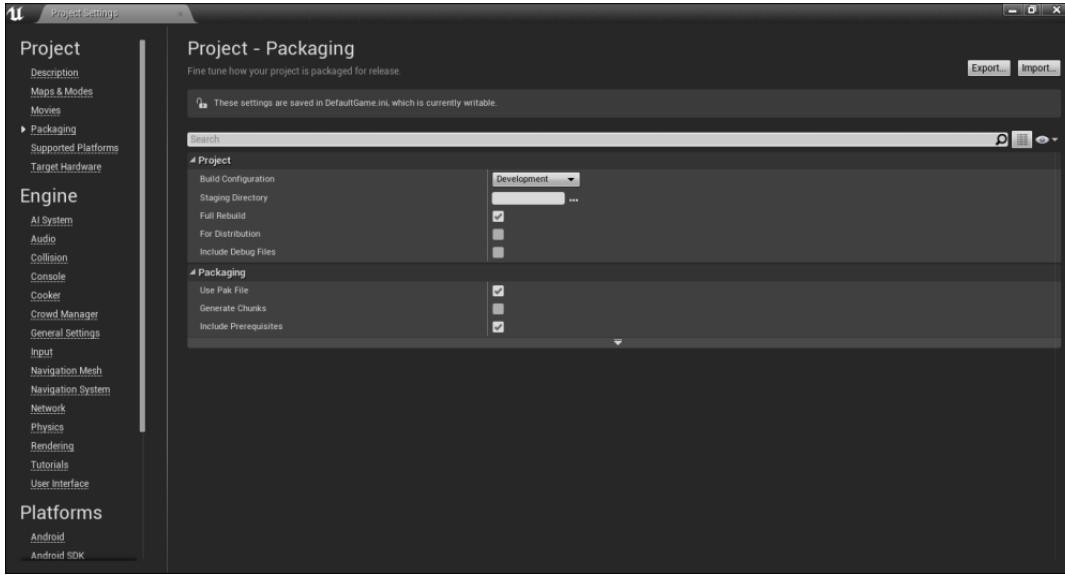


Let's go over the various sections:

- **Default Maps** : Here, you can set which map the Editor should open when you open the Project. You can also set which map the game should open when it is run. The first thing you need to change is the main menu map we had created. To do this, click on the downward arrow next to **Game Default Map** and select **Main\_Menu**.
- **Local Multiplayer** : If your game has local multiplayer, you can alter a few settings regarding whether the game should have a split screen. If so, you can set what the layout should be for two and three players.
- **Default Modes** : In this section, you can set the default game mode the game should run with. The game mode includes things such as the **Default Pawn class**, **HUD class**, **Controller class**, and the **Game State Class**. For our game, we will stick to **MyGame**.
- **Game Instance** : Here, you can set the default **Game Instance Class**.

## The Packaging settings

There are settings you can tweak when packaging your game. To access those settings, first go to **Edit** and open the **Project** settings window. Once opened, under the **Project** section click on **Packaging**.



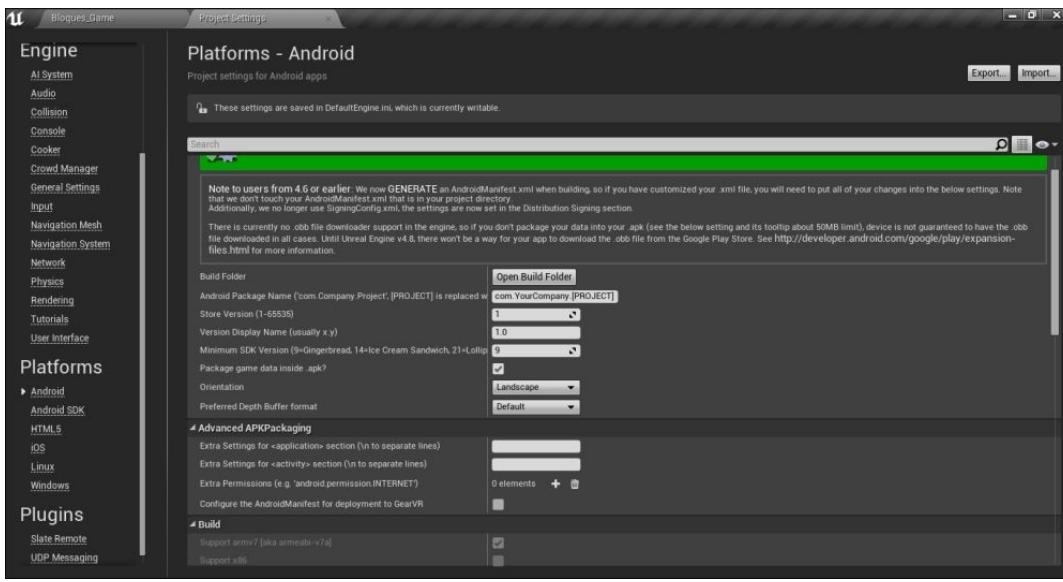
Here, you can view and tweak the general settings related to packaging the project file. There are two sections: **Project** and **Packaging**. Under the **Project** section, you can set options such as the directory of the packaged project, the build configuration to either debug, development, or shipping, whether you want UE4 to build the whole project from scratch every time you build, or only build the modified files and assets, and so on.

Under the **Packaging** settings, you can set things such as whether you want all files to be under one **.pak** file instead of many individual files, whether you want those **.pak** files in chunks, and so on. Clicking on the downward arrow will open the advanced settings.

Here, since we are packaging our game for distribution check the **For Distribution** checkbox.

## The Android app settings

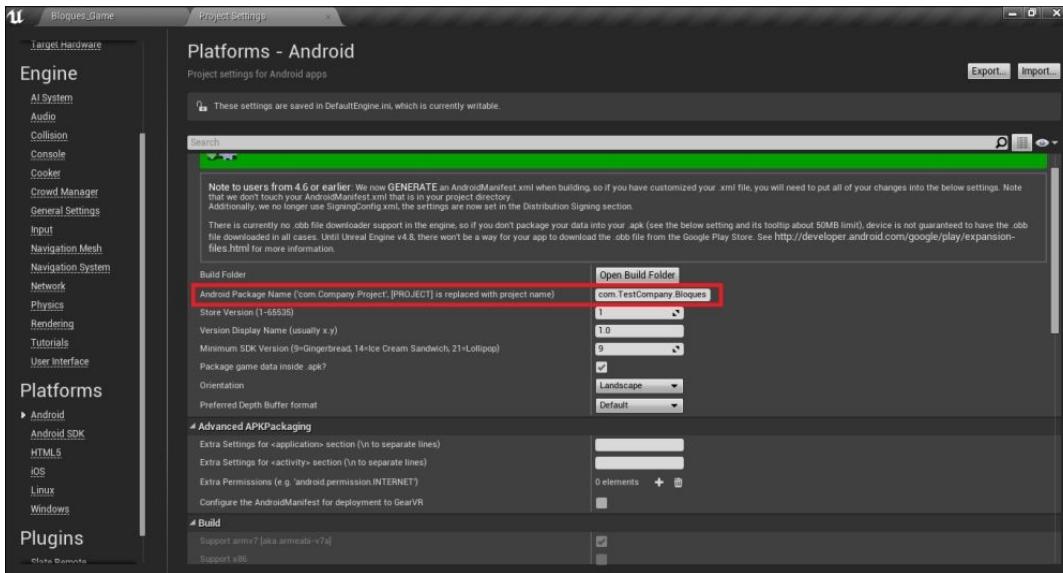
In the preceding section, we talked about the general packaging settings. We will now talk about settings specific to Android apps. This can be found in Project Settings, under the **Platforms** section. In this section, click on **Android** to open the Android app settings.



Here you will find all the settings and properties you need to package your game. At the top the first thing you should do is configure your project for Android. If your project is not configured, it will prompt you to do so (since version 4.7, UE4 automatically creates the `androidmanifest.xml` file for you). Do this before you do anything else. Here you have various sections. These are:

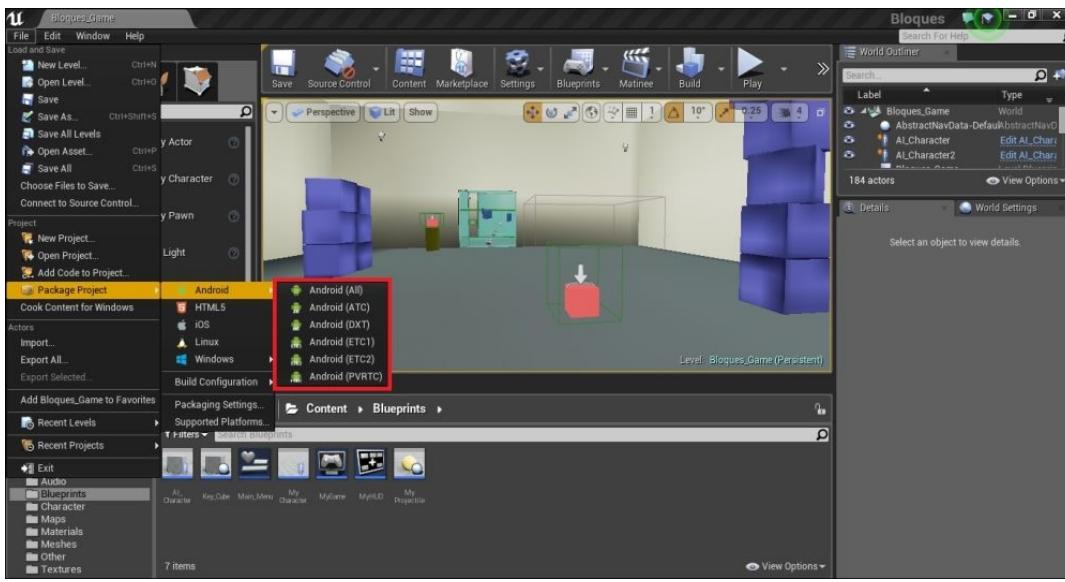
- APKPackaging** : In this section, you can find options such as opening the folder where all of the build files are located, setting the package's name, setting the version number, what the default orientation of the game should be, and so on.
- Advanced APKPackaging** : This section contains more advanced packaging options, such as one to add extra settings to the `.apk` files.
- Build** : To tweak settings in the **Build** section, you first need the source code which is available from GitHub. Here, you can set things like whether you want the build to support x86, OpenGL ES2, and so on.
- Distribution Signing** : This section deals with signing your app. It is a requirement on Android that all apps have a digital signature. This is so that Android can identify the developers of the app. You can learn more about digital signatures by clicking on the hyperlink at the top of the section. When you generate the key for your app, be sure to keep it in a safe and secure place since if you lose it you will not be able to modify or update your app on Google Play.
- Google Play Service** : Android apps are downloaded via the Google Play store. This section deals with things such as enabling/disabling Google Play support, setting your app's ID, the Google Play license key, and so on.
- Icons** : In this section, you can set your game's icons. You can set various sizes of icons depending upon the screen density of the device you are aiming to develop on. You can get more information about icons by click on the hyperlink at the top of the section.
- Data Cooker** : Finally, in this section, you can set how you want the audio in the game to be encoded.

For our game, the first thing you need to set is the **Android Package Name** which is found in the **APKPackaging** section. The format of the naming is `com.YourCompany.[PROJECT]`. Here, replace `YourCompany` with the name of the company and `[PROJECT]` with the name of your project.



## Building a package

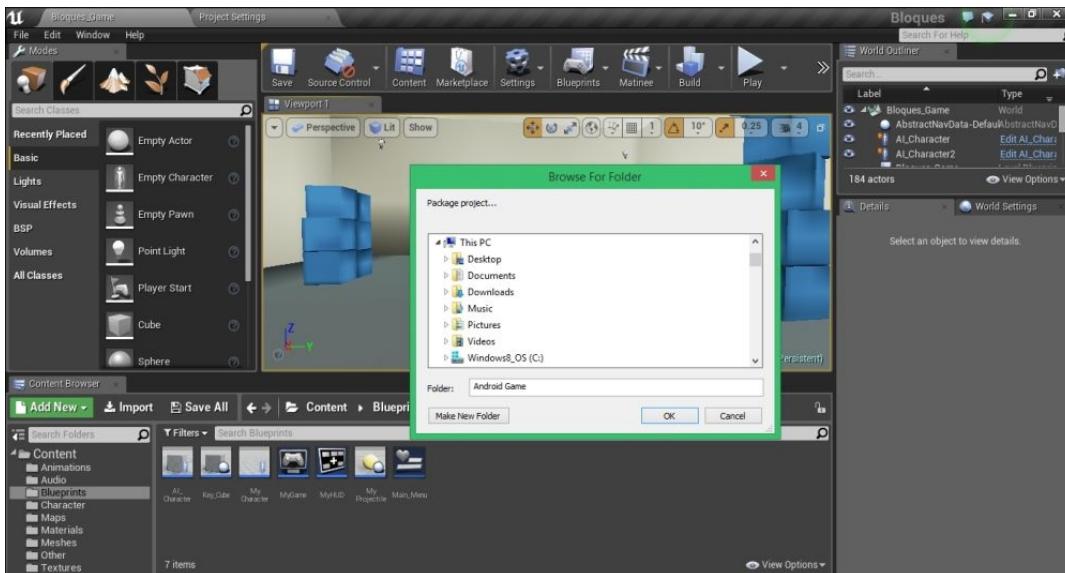
To package your project, in the Editor go to **File | Package Project | Android**.



You will see different types of formats to package the project in. These are as follows:

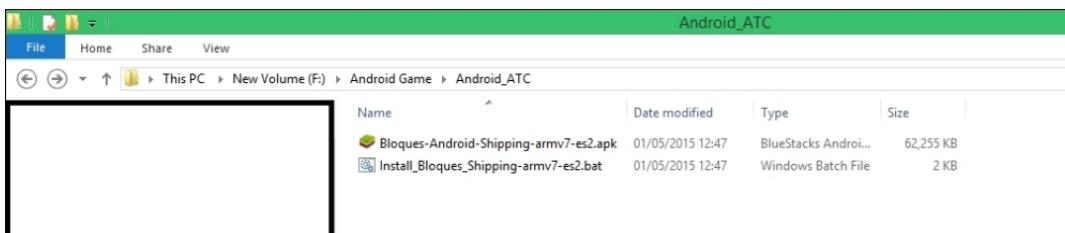
- **ATC** : Use this format if you have a device that has a Qualcomm Snapdragon processor.
- **DXT** : Use this format if your device has a Tegra graphical processing unit (GPU).
- **ETC1** : You can use this for any device. However, this format does not accept textures with alpha channels. Those textures will be uncompressed, making your game requiring more space.
- **ETC2** : Use this format if you have a Mali-based device.
- **PVRTC** : Use this format if you have a device with a PowerVR GPU.

Once you have decided upon which format to use, click on it to begin the packaging process. A window will open up asking you to specify which folder you want the package to be stored in.



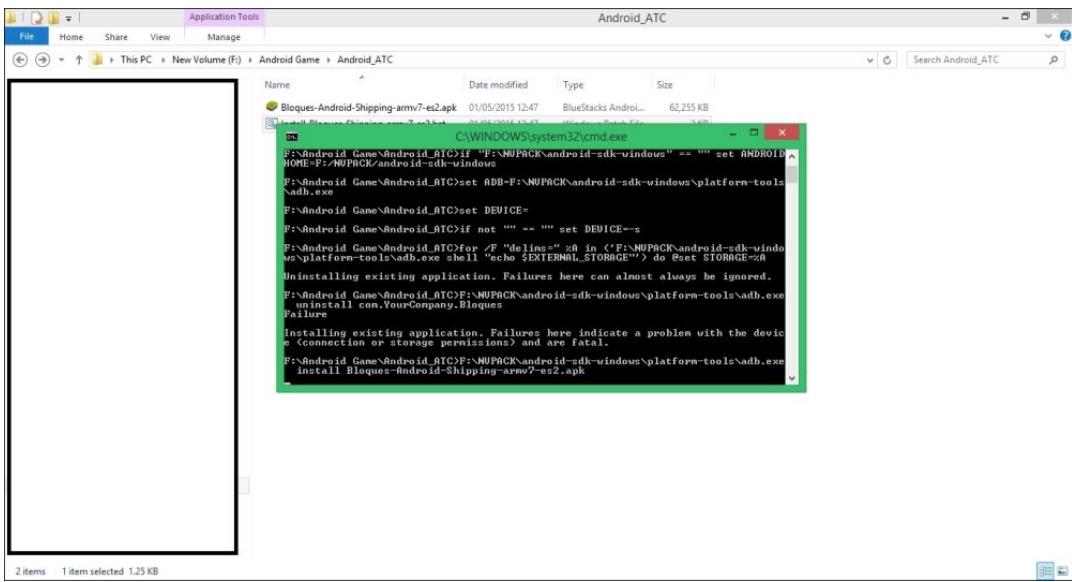
Once you have decided where to store the package file, click **OK** and the build process will commence. When started, just like with launching the project, a small window will pop up at the bottom-right corner of the screen notifying the user that the build process has begun. You can open the output log and cancel the build process.

Once the build process is complete, go the folder you set.



You will find a **.bat** file of the game. Providing you have checked the packaged game data inside **.apk?** option (which is located in the Project settings in the Android category under the **APKPackaging** section), you will also find an **.apk** file of the game.

The **.bat** file directly installs the game from the system onto your device. To do so, first connect your device to the system. Then double-click on the **.bat** file. This will open a command prompt window.



Once it has opened, you do not need to do anything. Just wait until the installation process finishes. Once the installation is done, the game will be on your device ready to be executed.

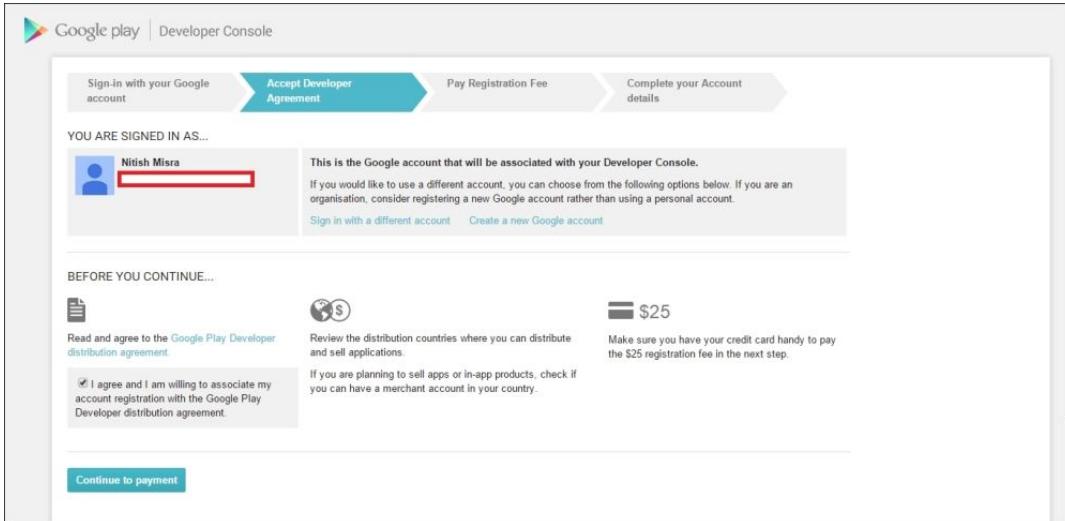
To use the .apk file, you will have to do things a bit differently. An .apk file installs the game when it is on the device. For that, you need to perform the following steps:

1. Connect the device.
2. Create a copy of the .apk file.
3. Paste it in the device's storage.
4. Execute the .apk file from the device.

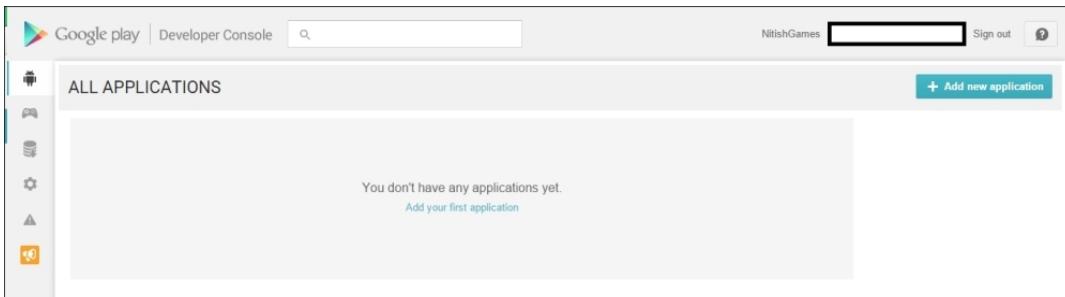
The installation process will begin. Once completed, you can play the game.

## Developer Console

Before we talk about how to publish the game on the Google Play Store, we first need to talk about the **Developer Console** from where you manage your app. This includes uploading and removing the app to and from the Play Store, filling in the app's description, setting the price of the app, and so on. However, before you can access the **Developer Console**, you first need to register. The cost to register is \$25. The link to the signup page is <https://play.google.com/apps/publish/signup/>.



Once registered, go to the **Developer Console** page.



Here, click on the **Add new application button** located at the top-right corner. When clicked, a window will open where you can set the language and set the name of the game as **Bloques**. Once set, click on **Upload APK**. Once clicked, you will see several options regarding the app.

At the top, you can see the name of the app, **Bloques**. Below it on the left-hand side are several panels. At the center are the available settings or options in the corresponding panel.

## ALL APPLICATIONS

Here is where all the apps and/or games you have published, or are in the process of publishing, are displayed.

APP NAME	PRICE	CURRENT/TOTAL INSTALLS	AVG. RATING / TOTAL NO.	CRASHES & ANRS	LAST UPDATE	STATUS
Bloques	—	—	—	—	—	Draft

The applications that are not published yet and/or in the process of being uploaded to the Play Store are saved as Drafts. You can also check things such as how many download/installations your app has had, the price, when the app was last updated, and so on. Clicking on any of the apps listed will open the app page, wherein you can set various properties and options for it.

## APK

The APK panel is opened by default. Here you can choose whether you want to publish the game on the Play Store.

APK

Beta Testing

PRODUCTION Publish your app on Google Play

BETA TESTING Set up Beta testing for your app

ALPHA TESTING Set up Alpha testing for your app

Licence keys are now managed for each application individually.  
If your application uses licensing services (e.g. if your app is a paid app, or if it uses in-app billing or APK expansion files), get your new licence key on the Services & APIs page.

Upload your first APK to Production

If you are in the process of developing your app or game and want to test your product in the market, as well as get some feedback before you publish the final product, you can set up a beta test by uploading your app from the **BETA TESTING** tab. If you wish to have more controlled testing of your app, for instance only being able to download if you have an invite or a key code, then you can set up an alpha test by uploading your product from the **ALPHA TESTING** tab.

## Store Listing

The next step panel is the **Store Listing** panel. This panel contains all of the options related to how the app is going to be displayed in the Google Play.

STORE LISTING

PRODUCT DETAILS

Title\* English (United States) – en-US Manage translations ▾

Bloques 7 of 30 characters

Short description\* English (United States) – en-US

0 of 80 characters

Full description\* English (United States) – en-US

0 of 4000 characters

Please have a look at these tips on how to create policy compliant app descriptions to avoid some common reasons for app suspension.

Here you need to fill out the details regarding your app. These details will be displayed in the Play Store when the user clicks on it. There are five sections in this panel.

The first section is **PRODUCT DETAILS**. Here you can set the name of the app, a short description about your app (which should be short and interesting enough to get the users' attention), and a full description about

your app (which contains a full description of what the app is, its feature set, and so on).

The next section is **Graphic Assets**. Here you need to post screenshots of your game. Since there are various Android devices, each with different screen sizes and screen resolutions, you need to add several screenshots of various resolutions so that the screenshots are not stretched or compressed on any device. You need to add a minimum of two screenshots; the maximum number of screenshots you can add is eight. The required sizes for various devices is given, so you can check and set them up accordingly. Apart from adding screenshots, you can also add a video which could be a trailer or a gameplay video of your app.

Following **Graphic Assets** is the **Categorization** panel. This deals with how your app will be categorized in the Play Store. Here you need to specify whether your product is an app or a game. After setting that, you will need to specify in what category your product lies; if you have picked **Applications**, the options you get under category would be **Books and References**, **Education**, **Business**, **Finance**, **Lifestyle**, and so on. If you have picked Games, you will have to set its genre under Board Game, Puzzle Game, Casual Game, and so on. Finally, you also need to set the **Content Rating** for your app.

Next, we have the **Contact Details** wherein you can post your website or your company's website, contact email address, and phone number.

Finally, in the **Privacy Policy** section you provide a privacy policy for your app.

## Content Rating

Content rating is an important aspect of any game or app. Google Play also offers its own content rating.

The screenshot shows the 'Content Rating' page in the Google Play Developer Console. The left sidebar lists sections: APK, Store Listing, Content Rating (selected), Pricing & Distribution, In-app Products, Services & APIs, and Optimisation Tips. The main content area is titled 'CONTENT RATING'. It includes a note about the IARC rating system, developer responsibilities (like completing the questionnaire for changes), and how ratings are used (informing consumers, blocking/filtering content). It also mentions the IARC logo and a note that an APK must be uploaded before taking the questionnaire. Buttons for 'Save draft' and 'Publish app' are at the top right.

Even though there is a **Content Rating** section in **Store Listing**, this is a much more comprehensive and extensive rating system. With the help of a questionnaire with various questions regarding the app/game and its content, Google will rate your product. But to get the questionnaire you will first have to upload your app.

## Pricing & Distribution

This section contains everything related to the cost of your app and where your app will be available for download.

The screenshot shows the 'Pricing & Distribution' page in the Google Play Developer Console. The left sidebar shows the same sections as the previous screenshot. The main content area is titled 'PRICING & DISTRIBUTION'. It has a section for 'This application is' with 'Paid' and 'Free' options ('Free' is selected). A note says to set up a merchant account for paid apps. Below is a section for 'DISTRIBUTE IN THESE COUNTRIES' with a note that no countries are selected. A checkbox for 'SELECT ALL COUNTRIES' is checked, followed by a list of countries: Albania, Algeria, Angola, and Antigua and Barbuda.

At the top, you can set whether your app is **Free** or **Paid**. If you want your app to be paid, you will first have to set up a merchant account.

Below that is a list of countries. You can choose in which country you want your app to be available by clicking on the checkbox. If you want your app to be available worldwide, check **SELECT ALL COUNTRIES** and all of the tick boxes will be checked.

## In-app Products

This section contains settings regarding in-app products.

The screenshot shows the 'In-app Products' section of the developer console. On the left, there's a sidebar with icons for APK, Store Listing, Content Rating, Pricing & Distribution, In-app Products (which is selected), Services & APIs, and Optimisation Tips. The main area has tabs for 'IN-APP PRODUCTS' and 'Why can't I publish?'. It displays a message: 'Your app doesn't have any in-app products yet.' Below it, it says 'To add in-app products, you need to add the BILLING permission to your APK.' and provides a link to 'Upload a new APK'. At the bottom, it says 'To add in-app products, you need to set up a Google Wallet merchant account.' and a link to 'Set up a merchant account'.

A popular monetization method used by developers is in-app purchases. These are virtual goods that can be bought using real currency. Here, you can manage your in-app products and set items such as price, what is available to purchase, and so on. However, if you want to have in-app purchases in your game or app, you will first have to set up a merchant account.

## Services & APIs

This is where you can see what services offered by Google are currently active in your app/game.

The screenshot shows the 'Services & APIs' section of the developer console. The sidebar shows 'Services & APIs' is selected. The main area has a tab for 'SERVICES & APIs'. It lists 'GOOGLE CLOUD MESSAGING (GCM)' with a note that it helps send data from servers to applications. It also notes that to access GCM stats, a GCM sender ID needs to be linked. Below that is a 'LICENSING & IN-APP BILLING' section with a note about preventing unauthorized distribution and verifying purchases. At the bottom, there's a 'YOUR LICENCE KEY FOR THIS APPLICATION' section containing a long Base64-encoded RSA public key.

These services include Google Cloud Messaging, which you can use to send information and data from the servers to your app or game; licensing, which prevents piracy and unauthorized distribution of your product and is used to verify in-app purchases; and Google Play services, which includes leaderboards, achievements, push notifications, and many more.

## GAME SERVICES

As previously discussed, Google offers various services for app developers. This is where you can set what services you want on your app/game.

The screenshot shows the 'Game Services' panel of the developer console. The sidebar shows 'Game Services' is selected. The main area has a table with columns for NAME, PLATFORMS, ACHIEVEMENTS, LEADERBOARDS, PLAYERS, and STATUS. There is one entry for 'Bloques' with platforms listed as Android, iOS, and others. The status is 'Draft'.

When you go to this panel, you will see a list of apps that you have produced, similar to that in the All Applications panel. Here, you can view what platforms your app or game is on, what achievements you have in your game, your leaderboards, and how many unique players have signed in to your game using their Google account. As with All Applications, clicking on the name of the app will open a page wherein you have several options regarding these services.

Let's discuss these sections individually.

### Game details

Here you can set the general details regarding your app/game.

The screenshot shows the 'Game details' section of the Google Play Developer Console. On the left, a sidebar lists various settings: Quests, Game details (selected), Linked apps, Events, Achievements, Leaderboards, Testing, and Publishing. The main panel displays 'GAME DETAILS' for the English (United States) - en-US locale. It includes fields for 'Display name' (set to 'Bloques'), 'Description' (empty), 'Category' (set to 'Puzzle'), and 'Saved Games' (set to 'ON'). A note at the top right states: 'Fields marked with \* need to be filled for saving. All fields need to be filled for publishing.'

These general details include your app's display name, its description, category, graphic assets, and so on. You can change the settings for these here as well.

## Linked apps

To be able to use Google Services, you will first have to link your app.

The screenshot shows the 'LINKED APPS' section of the Google Play Developer Console. The sidebar shows 'Game details' (selected) and other options like 'Events', 'Achievements', etc. The main panel has a message: 'You need to link all applications that you want to use with Google Play game services and generate an OAuth2 key for each app to include in your binary. At least one app needs to be linked before the Google Play games services can be tested or published.' Below this are buttons for 'Android', 'iOS', and 'Web', with an 'Other platform' link at the bottom.

Linking your app will generate an OAuth2 key, which you will need to include in your app's binary before you can make use of these services. Here you can choose different platforms for your app, namely Android, iOS, Web, and others. Clicking on any one of them will open a page wherein you have to fill in certain details before you can link your app. We will come back to this in the later sections.

## Events

Getting users to download your app is one thing. To ensure that the users who have downloaded your app or game keep using them, in other words user retention, is a whole different area of expertise and equally important, if not more. The next three panels are for just that. The first is the **Events** panel.

The screenshot shows the 'EVENTS' section of the Google Play Developer Console. The sidebar shows 'Events' (selected) and other options like 'Quests', 'Game details', etc. The main panel has a message: 'Events are a fun way to track game progress of your users as they are going through the game. It can be used to drive user quests, such as killing the number of zombies in a game. Events can be added on a regular basis to track users' engagement.' Below this is a link to 'Learn all about implementing events in the developer documentation.' At the bottom are buttons for 'Add event' and 'Continue to next step'.

Once you have a sizable number of users, you can start having periodic events and rewards for users who participate in or win this event. Events can include weekly contests, discounts on in-game products, and so on. This is a great way to keep the users engaged in your game. To add an event, click on **Add Event** which will take you to the **Event** page wherein you can fill in the details regarding the event.

## Achievements

Another way of retaining users is creating achievements. Achievements are goals or tasks that the player has to perform in the game in order to unlock the respective achievement. This could include things like playing the game  $X$  number of times, killing  $Y$  number of enemies, and so on. Usually, when the player unlocks an achievement, they are rewarded with some in-game currency or something similar. Achievements are ideal for retaining perfectionist players who play to finish the game completely. This means finishing the game (providing it has an ending), unlocking anything and everything that is unlockable, and unlocking every achievement in the game. It is also a necessary component of your game since you need at least five achievements before you can publish your game.

The screenshot shows the Google Play Developer Console interface. On the left, there's a sidebar with icons for Quests, Game details, Linked apps, Events, Achievements (which is selected and highlighted in orange), Leaderboards, Testing, and Publishing. The main content area is titled "ACHIEVEMENTS". It contains a brief description of what achievements are and how they can be used. Below that, it says "Learn all about implementing achievements in the developer documentation." and "You need to add at least 5 achievements before you can publish your game." At the bottom, there are two buttons: "Add achievement" and "Continue to next step".

To add an achievement, click on the **Add achievement** button which will take you to the **achievement details** page. Here, you can set all of your achievement details, such as the name of the achievement, a description about how it can be unlocked, an icon, and so on.

## Leaderboards

Last, but not least, we have **Leaderboards**. Leaderboards are yet another way of retaining players, especially competitive players who play to get the highest score. A leaderboard is sort of a list where the names of players who have scored the highest points in your game are displayed in descending order.

The screenshot shows the Google Play Developer Console interface. The sidebar is identical to the previous one, with the "Achievements" tab still selected. The main content area is titled "LEADERBOARDS". It contains a brief description of what leaderboards are and how they encourage players to keep playing. Below that, it says "Learn all about implementing leaderboards in the developer documentation." At the bottom, there are two buttons: "Add leaderboard" and "Continue to next step".

To add a leaderboard, click on the **Add leaderboard** button, which will take you to the **leaderboard details** page. On this page, you can set various options, such as the name, the format, the upper and lower limit of the leaderboard, and so on.

## Testing

This section contains various options regarding testing the Google Play services on your app/game. Before you publish your game, it is advisable that you first test your app to ensure that everything is working properly and if a problem does arise, fix it before releasing.

The screenshot shows the Google Play Developer Console interface. The sidebar includes "Quests", "Game details", "Linked apps", "Events", "Achievements", "Leaderboards", "Testing" (which is selected and highlighted in blue), and "Publishing". The main content area is divided into two sections: "TESTING GOOGLE PLAY GAME SERVICES" and "TESTING ACCESS". The "TESTING GOOGLE PLAY GAME SERVICES" section has a warning message: "The Google Play game services settings are not ready to test yet." It also lists "WHAT'S MISSING?" under "Linked apps" with the note "You need to provide one or more linked apps". The "TESTING ACCESS" section shows a list of users who can test the game: "The following users can test your saved drafts for Google Play game services before they are published." There is a button labeled "Add testers".

Here, you can see whether the services are ready to test or not. If not, you will be told what is required before the testing can begin. You can also invite other people to test your app/game by clicking on the **Add testers** button and entering in their respective Google account details.

## Publishing

Finally, there is the **Publishing** section. Once you have filled out all of the details, added screenshots, set up the Google Play services, tested them and are confident that your product is ready to be published, you can publish it.

The screenshot shows the 'Publishing' section of the Google Play Developer Console. A prominent warning message states: "Your game is not ready to be published." Below this, a section titled "WHAT'S MISSING?" lists requirements for publishing:

- Game Details**: You need to provide the description of your game service, a high-resolution icon, and a feature graphic.
- Linked apps**: You need to provide one or more linked apps.
- Achievements**: You need 5 or more achievements to publish your game.

At the bottom, there is a "DELETING YOUR GAME" section with a "Delete your game" button.

Here, you can see whether your app/game is ready to be published or not. If it is not ready, you can see what is missing and what you need to do before you publish. If your game is already published and you want to remove it from the Play Store, you can do so by clicking on the **Delete your game** button.

## REPORTS

In the **REPORTS** panel, you can view things such as reviews, **Crashes & ANRs**, **Statistics**, and so on.

The screenshot shows the "REPORTS" section of the Google Play Developer Console. On the left, there is a sidebar with options: Crashes & ANRs, Reviews, Statistics, and Financial reports. The main area is titled "CRASHES & ANRS" and contains a "Select an application" dropdown with a search bar.

Here too are various sections, all containing different reports:

- Crashes & ANRs** : In this section, you can view reports related to game crashes and ANRs. Game crashes are a frequent occurrence. Even if your game is properly optimized, there are still some unforeseen issues that might occur—some minor and some major. To retain your users, you will have to fix these crashes; otherwise, they will get frustrated and delete your game. Here you can view crash reports for all of the games and apps you have published and accordingly work to debug them. ANRs appear when your app stops responding on a device similar to the **Not Responding** prompt you get in Windows and Mac. This is different from crashes since when a game crashes the application stops. When an ANR occurs, the application still runs on the device, but does not respond. This is equally important to resolve.
- Reviews** : A good way to see how well your game or app is doing in the market is by reading its reviews. It is also a great way of learning the shortcomings of your game or app. You can use this information when creating a sequel or planning to develop an update. You can also get information about any bugs that are present in the game and use that knowledge to resolve them.
- Statistics** : This section shows you how many people are playing the game, how many downloads your game has gotten, where your game is most popular (location), how many active users your game has, and so on.
- Financial reports** : Finally, there is the **Financial Reports** section wherein you can keep track of the revenue your product is generating and use it to update your game or app accordingly.

## SETTINGS

Here, you have various options regarding things such as account details, rights regarding who can access the developer console, and so on, which you can view and set.

The screenshot shows the "SETTINGS" section of the Google Play Developer Console. On the left, there is a sidebar with options: Account details, User accounts & rights, Activity log, Email preferences, API access, and AdWords accounts. The main area is titled "ACCOUNT DETAILS" and contains fields for "Developer name" (NitishGames, 11 of 50 characters), "Physical address" (a large text input field), "Email address" (Email address \*), and "Website" (a text input field).

Here too are various sections, each categorized based on the type of options that are as follows:

- **Account Details** : This section contains all the general options regarding your developer console account, such as the name of the developer, your address, your email ID, website URL, and so on.
- **User accounts & rights** : From here, you can set who can access your developer console. When working in a team, it is understandable that you would want all of them to have access. You can set that in this section.
- **Activity log** : Here you can see all of the changes you have made to your app in the developer console, along with a time-stamp as to when the particular change was made. This is a great way to keep track of any changes made to the app.
- **Email preferences** : If you want or do not want to receive alerts about your apps via email, you can set it here.
- **API Access** : API is an important aspect of apps. It allows you to manage things like in-app purchases, authenticating transactions, and so on. However, before you can use them, you will have to link your app first. You can link your app here, in the **API Access** section.
- **AdWords accounts** : Here you can link your account to an **AdWords account**. AdWords is a service offered by Google which allows you to promote your app by using advanced targeting techniques to make sure that the right people get to see your advertisement or promotion for your app or game.

## ALERTS

In the Settings section, we discussed how you can turn on/off email alerts. If you have turned on email alerts, you will receive all alerts regarding your app via email. If you have turned it off, you can view these alerts here in the **ALERTS** panel.

## Publishing your game

Now that we have discussed the Developer Console, we can go ahead and publish our game to the Google Play Store.

### Activating Google services

Now, since the app is more than 50 MB, you will have to make use of the Google Play service, namely the APK expansion files. Go to the **Game services** panel, and then the **Game details** section. Here, fill out all the fields since all of them are the basic requirements that the app should have before you can publish your services. This includes setting the **Display name**, which you can set as **Bloques**. In the **Description** section just write what the game is, what the objective of the game is, and so on.

The screenshot shows the 'GAME DETAILS' section of the developer console. The 'Display name' is set to 'Bloques'. The 'Description' field contains a detailed description of the game. The 'Category' is set to 'Puzzle'. The 'Saved Games' option is currently set to 'OFF'.

Since this is a puzzle game, in the **Category** field, choose **Puzzle**. Further, since we do not require saved games, you can set it to **Off**. For the **Graphic Assets**, you will need a minimum of two: one for the icon and one feature graphic. The resolution required for the screenshot is  $512 \times 512$  and the resolution required for the feature graphic is  $1024 \times 500$ . You can take high-resolution screenshots of your scene and edit them in Photoshop to create your image.

The screenshot shows the Google Play Developer Console interface. On the left, there's a sidebar with icons for Quests, Game details, Linked apps, Events, Achievements, Leaderboards, Testing, and Publishing. The main area has tabs for 'Game details' and 'API CONSOLE PROJECT'. Under 'Game details', there are sections for 'High-res icon' (512 x 512, 32-bit PNG with alpha) and 'Feature graphic' (1024 w x 500 h, JPG or 24-bit PNG no alpha). A preview image of the game 'BLOQUES' is shown. Below this, under 'API CONSOLE PROJECT', it says 'This game is linked to the API console project called "Bloques"' and lists 'APIs required for basic Games Services to work' (Google+ API, Google Play Game Services, Google Play Game Management). At the bottom, there are links for 'USEFUL ANDROID RESOURCES' (Android Developers, Android Design, Android.com), 'USEFUL TOOLS' (Google Wallet Merchant Center, Google Analytics, AdMob), and 'NEED HELP?' (Help centre, Contact support). A footer at the bottom includes a copyright notice: '© 2015 Google - Google Play Terms of Service - Privacy Policy - Developer Distribution Agreement'.

Once you have everything set up, click on **Save** at the top of the screen.

After you have filled in the details go to **Linked apps**, fill in the required fields and click on **Continue** at the top. Once clicked, you will have to authorize your app. Click on **Authorize your app now** button. This will open a menu where you will be asked to fill in the name of the product, logo, and so on. However, since we already filled in those details earlier, we can leave it as it is (unless you want to make any changes). Click on **Continue**. Before we can move any further, we will first have to create a Client ID. To find out what the Client ID for your app is, follow these steps:

1. Open Command Prompt. Make sure you run as administrator.
2. Run the following command:

```
keystore -exportcert -alias androiddebugkey -keystore C:\Users\*Your Username*.android\debug.keystore -list -v
```

3. When it asks for a password, enter the default password: `android`.
4. Copy the SHA1 signature and paste it in the Signing certificate fingerprint (SHA1) field.
5. Next, click on **Create ID** which will create a client ID for you.

Once done, Google will create a unique client ID for your game as well as an Application ID.

The next thing we need is **ACHIEVEMENTS**. In order to publish our game, we need a minimum of five achievements. You need a name for the achievement, a description, and an icon to go along with it. You can create your own achievements but for now there are five achievements as follows:

- **First Time** : This is when the player plays the game for the first time
- **Getting the Hang of it** : This is when the player clears the second room
- **Puppet Master** : This is when the player clears the third room
- **Puzzle Master** : This is when the player clears the fourth room
- **Addicted** : This is when the player plays the game five times

The screenshot shows the Google Play Developer Console under the 'Achievements' tab for the game 'Bloques'. The sidebar on the left includes links for Player analytics, Feature analytics, Quests, Game details, Linked apps, Events, and Achievements (which is currently selected). The main area displays a table of achievements with columns for #, NAME, ID, POINTS, UNLOCKED %, and STATUS. There are five achievements listed:

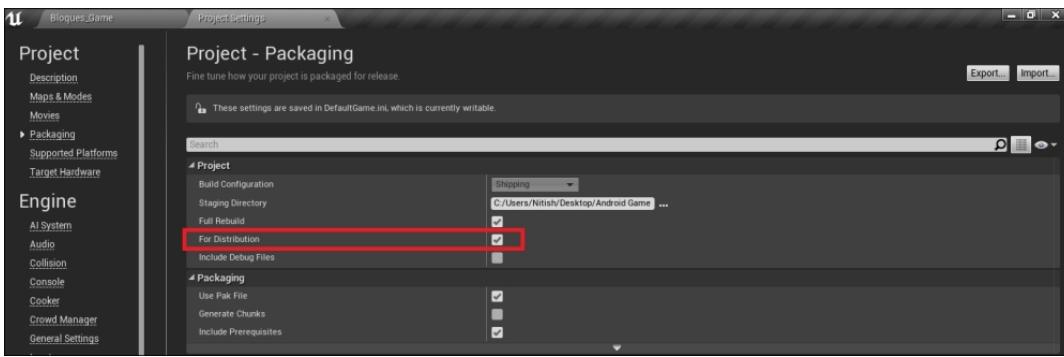
#	NAME	ID	POINTS	UNLOCKED %	STATUS
1	1st First Time	CgkIqPT6_AeEAIQAA	5	—	— Unpublished
2	2nd Getting the Hang of it	CgkIqPT6_AeEAIQAg	10	—	— Unpublished
3	3rd Puppet Master	CgkIqPT6_AeEAIQAw	20	—	— Unpublished
4	4th Puzzle Master	CgkIqPT6_AeEAIQBA	30	—	— Unpublished
5	5th Addicted	CgkIqPT6_AeEAIQBQ	50	—	— Unpublished

At the bottom, there are buttons for 'Get resources' and 'Total points: 115'. A note at the bottom says 'Learn all about implementing achievements in the developer documentation.'

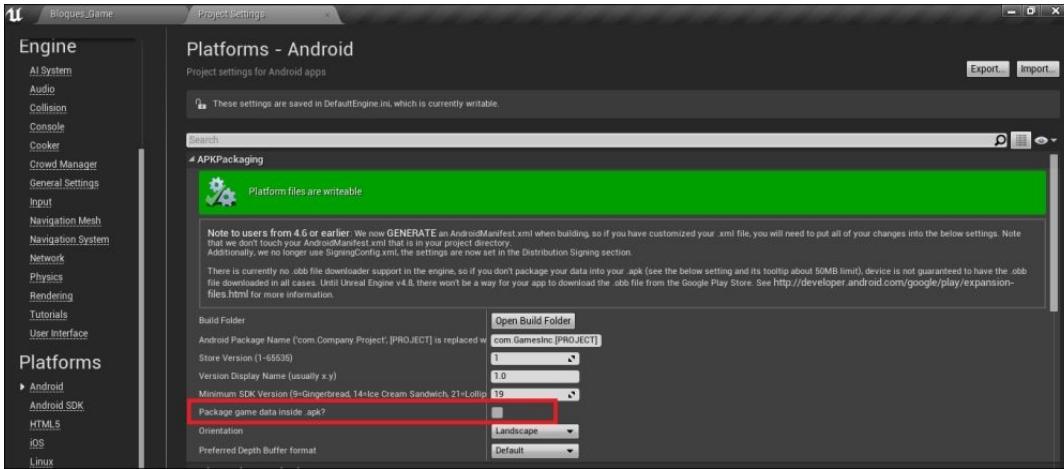
You can assign any number of points you see fit. We do not require leaderboards, since our game does not have points or scores.

## Preparing the project for shipping

Now we need to go back to the **Engine** and define a few settings in the Project. First, we need to set what type of build we are going to make. To do this, go to **Project Settings** and under **Project**, in the **Packaging** section, check the **For Distribution** box. You will notice that the **Build Configuration** option will be set to **Shipping** and the option will be locked.



The next thing we are going to do is copy some of the values from the developer console to our project. Go to **Project Settings | Platforms | Android**. First, in the **APKPackaging** section, uncheck **Package game data inside .apk?**. Since our game is more than 50 MB, we cannot upload the .apk file directly. What we need is a .obb file, an extension file, which we will be uploading in addition to the .apk file. With this option unchecked, UE4 will automatically create both a .apk file (which will be smaller than 50 MB) along with an .obb file (the extension file).



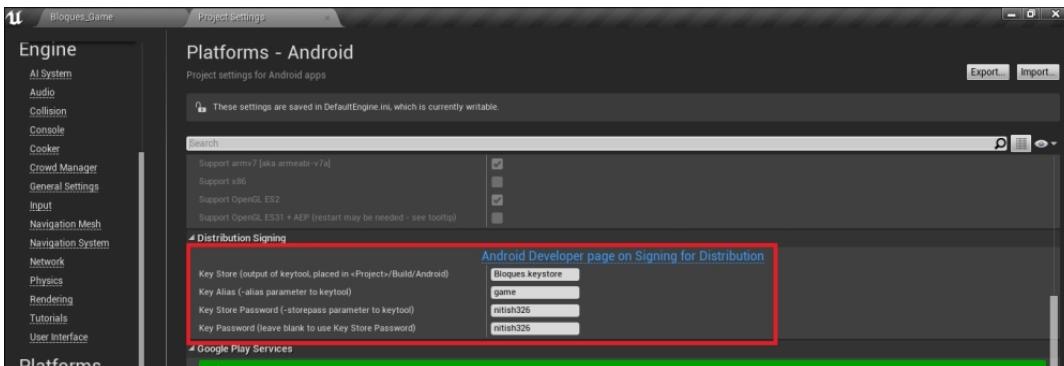
Next, go to **Distribution Signing**. This section is necessary if we are to package our game for shipping. The first thing we need is to create a **keystore** file. A keystore file is a binary file that has a set of keys. You can think of it as a digital signature which is used to identify and authenticate your app on the app store. There are various ways to create a keystore: one of them is with the help of Command Prompt. So, run Command Prompt as administrator and enter the following command:

```
keytool -genkey -v -keystore -*name of your project*.keystore -alias *alias_name* -keyalg RSA - keysize 2048 -validity 10000
```

Replace `*name of your project*` with Bloques (or whatever you see fit) and `*alias_name*` with Game (or whatever you see fit).

After you have entered the command, you will be asked to fill in some other information, such as the password for your keystore, your name, your company's information, and so on. Once you fill those in, the keystore will be created. Locate it and place it in `*Project Directory*/Build/Android`.

Once done, go back to **Project Preference**, and under **Distribution Signing**, in **Key Store**, enter the name of the **keystore** file you created, along with the extension (for example, if the name of the keystore is Bloques, then enter Bloques.keystore). Then, in **Key Alias**, enter the alias name (Game). Finally, in the **Key Store Password** and **Key Password** fields, enter in the password you had defined when you were making the **keystore** file.



Next, go to **Google Play Services**. Here the first thing to do is to check **Enable Google Play Support**. This will enable Google Play services in the game. In the **Game App ID** field, fill in the Application ID you got when you linked your app in the Developer Console. You can find this ID in the Developer Console under the **Authorization** section of **Linked apps** in the **Game services** panel.

MULTIPLAYER SETTINGS

Turn-based multiplayer

Real-time multiplayer

ANTI-PIRACY

Enable anti-piracy

AUTHORISATION

Application ID **1001287984623**

OAuth2 Client ID **1001287984623-kgunqq4n0ej5724sc4rvnk58heoapi.apps.googleusercontent.com**

You only need to include the application ID (1001287984623) in your Android app.  
Learn more about how to do this in the developer documentation.

Next, we are going to need the Google Play License Key. This can be found in the **Developer Console**, under the **LICENSING & IN-APP BILLING** section of **Services and APIs** in the All applications panel.

LICENSING & IN-APP BILLING

Licensing allows you to prevent unauthorised distribution of your app. It can also be used to verify in-app billing purchases. Learn more about licensing.

YOUR LICENCE KEY FOR THIS APPLICATION

Base64-encoded RSA public key to include in your binary. Please remove any spaces.

**MIIBIjANBgkqhkiG9w0BAQEFAACQgA8M11Bc9yKCAQEA1iT7cdURk+Mnb0MpHCUweeoLZ4Uqv937LN1qFwTck/pj179kt+XGDeaIQ2wvFWk137IIBye9NDYi1SXKt+w4aJuhg8xax8ffnq3Tf4z+7Xf3xhobC0D4en5o/o/gn0/ym9dfpG1clu01LNg4Covun8zD+XlQ2IUs13CV1DEJFK/wax3stK1s10nSdbahkj3C9XaOo19Ng7rmvacuAs2d30yX125X12sQajfocDXFbgrira2+2b79X1kPnize9cyQLst+V2k1E3SXKsBv54d1V27Op2L4TzXYB6IHntBCodhgzCmhrQ1DAQAB**

GOOGLE PLAY GAME SERVICES

This app is using Google Play game services. You can configure Google Play game services for this app in the [Game services](#) section of this site.

[Learn more](#)

This extremely long string of seemingly random characters is our License Key. Copy the entire thing and paste it in the **Google Play License Key** section. Next, in the Achievement Map, fill in the name and the ID of the achievements you created earlier.

#	NAME	ID	POINTS	UNLOCKED %	TOTAL # / TIME	STATUS
1	First Time	CgkI78uo15IdEAiQAA	5	—	✓ Published	
2	Getting the Hang of it	CgkI78uo15IdEAiQAQ	10	—	✓ Published	
3	Puppet Master	CgkI78uo15IdEAiQAg	20	—	✓ Published	
4	Puzzle Master	CgkI78uo15IdEAiQAw	25	—	✓ Published	
5	Addicted	CgkI78uo15IdEAiQBA	30	—	✓ Published	

Total points: 90

[Get resources](#)

Learn all about implementing achievements in the [developer documentation](#).

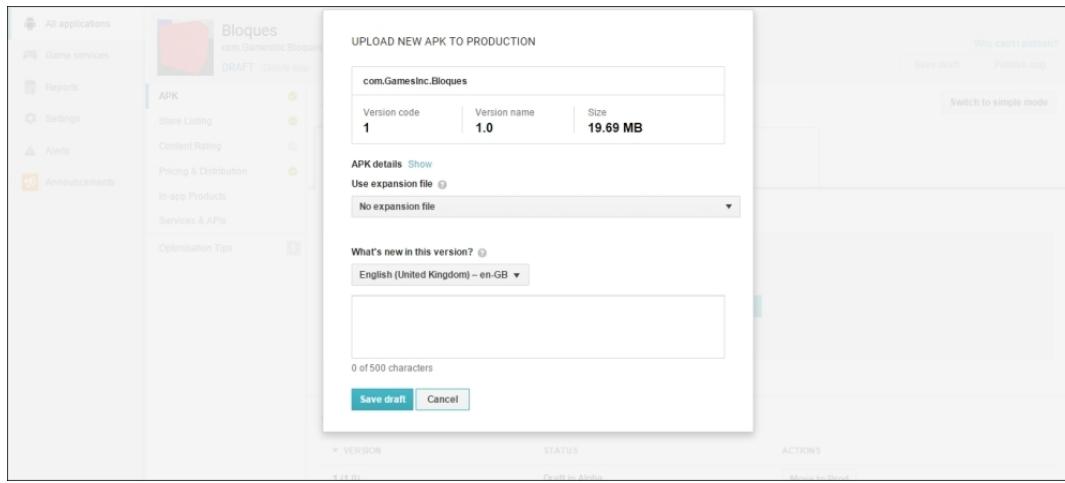
Just copy and paste the name along with the ID here. If you have icons that you would like to add you can do so here as well, in the **Icons** section. We are now ready to package our game for distribution. So, in the Editor, go to **File | Package Project | Android** and choose any format and package the game.

Once the build is complete, go to the folder you set to store the package. You will see a **.apk** file, a **.bat** file, and a **.obb** file of the project. We will only require the **.apk** and **.obb** file.

Name	Date modified	Type	Size
Bloques-Android-Shipping-armv7-es2.apk	30/05/2015 17:44	BlueStacks Android	20,166 KB
Install_Bloques_Shipping-armv7-es2.bat	30/05/2015 17:44	Windows Batch File	2 KB
main.00001.com.GamesInc.Bloques.obb	30/05/2015 17:42	OBb File	41,422 KB

## Uploading the game on the Play Store

The last step is to upload the package onto the Play Store and fill out a few more details. Go back to the **Developer Console** and go to **All Applications**. Within that, go to **APK**. Here is where we upload our **.apk** and **.obb** files. Go to the **Production** tab and click on **Upload APK**; then select the **.apk** file. Once the upload has finished, we will need to upload the extension or the **.obb** file.



To upload the .obb file, click on **No expansion file** to open a dropdown menu and select **Upload a new file**. From there, select the .obb file. After the upload is complete, click on **Save Draft**.

The next thing to do is fill the sections in the **Store Listing** section. This is similar to what we filled in in the **Game Details** sections in the **Game Services** panel. You can copy-paste the fields. You also require a minimum of two screenshots here.

Next, go to **Content Rating**. Here, once you have uploaded the APK you will be asked to fill in a questionnaire which will properly rate your app. These questions mostly relate to the content of your game. Fill them out and save the questionnaire. The **Save** button is located at the bottom. Next, click on **Calculate Rating** and it will generate an appropriate rating for your game. Once you have seen your rating, click on **Apply Rating** located at the bottom of the page.

The last thing we need to fill out is the **Pricing and Distribution** form. Here, you will need to specify in which countries you want the game to be available, as well as tick read and agree to guidelines regarding export laws, and such.

You are now ready to publish your app. To do so, click on **Publish App** located at the top-right corner of the **Developer Console** and your app will be added to the Google Play Store for the world to download, play, and enjoy.

## Monetization methods

It is one thing to make a good game. However, the most important thing is for your game to generate revenue because, after all, we all have bills to pay. Ever since mobile games came into the scene, several monetization models have emerged. Here are the four most popular and widely used monetization models that developers use to generate revenue:

- **Freemium model** : This is a widely used business model made popular by games such as Angry Birds, Cut the Rope, and so on. Here, you make certain features of the game free while other features are locked—which cost money to unlock. The main goal of this model is to attract as many people as possible and give them a preview of the app and its features without giving away too much, so that the users become interested enough in purchasing the whole app. This is similar to the Demo version of games for the PC and Consoles.

The advantages of using this model are that users get to try the product before they purchase it, making them more likely to purchase the app later on. It is also an easy way to build a large user base, since the app has no upfront charge and people love free things.

On the flipside, some of the disadvantages of using this model are that if you offer too few features, the users will not be engaged enough to purchase the whole product. On the other hand, if you offer too many features, the users will not have a good enough reason to purchase the whole product.

- **In-app advertisement** : Another popular business model is making the product completely free for users. The way developers earn revenue is through in-app advertisements. You may have seen such apps. The game or app is free to download and you have ad banners at the top or bottom of the screen.

The advantages of using this model are that you completely remove any payroll or barriers between the user and the features offered by the app, making it more desirable. You can also gather data based on their behavior and use this for target advertising.

However, using this model means sacrificing the already limited screen space for ads, which means sacrificing the user experience. Another thing to keep in mind is that people can and will get annoyed with the ads and thus can cause them to stop using or uninstall the app. So be careful how and when you use them.

- **Paid apps** : As the name suggests, in this model, the app is not free; users will have to purchase the app in order to use it. The advantages of using this model are that you get upfront revenue from every download along with better user retention (since they paid to download the app, they are more likely to use it frequently). However, with that said, unless you have a good reputation in the app market, it is hard to convince users to pay upfront for your product. It also means that users would have high expectations of your product.

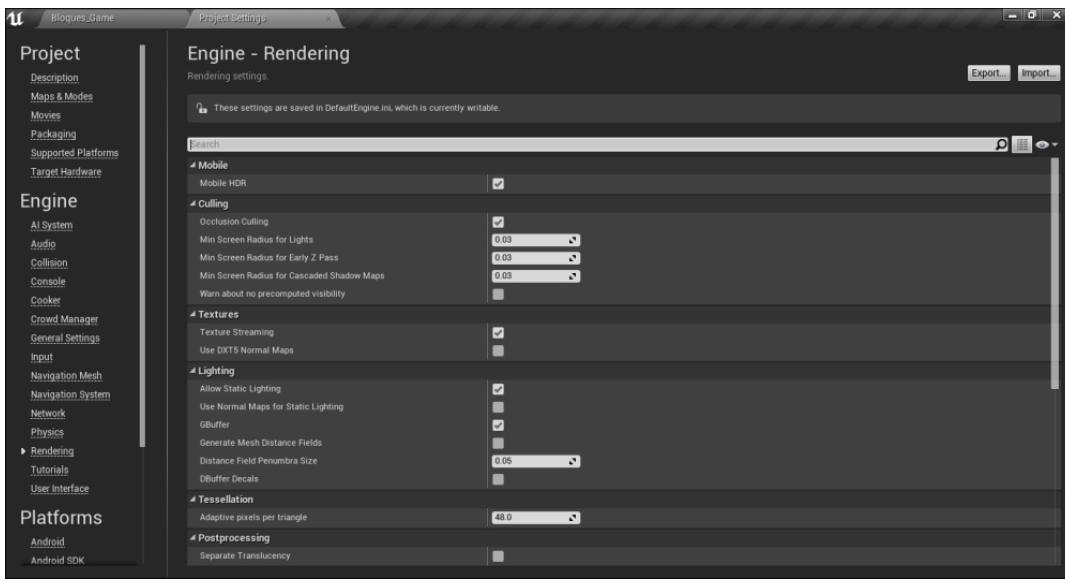
• **In-App purchases** : This model, made popular by games such as Candy Crush Saga, Clash of Clans, and so on is similar to the Freemium model in that the product is made available free. However, in this case, no features are blocked for the users until they make a purchase. Instead, they get all of the features right from the start. The way that the developers earn revenue is by in-app purchases. These are virtual goods, such as lives, power-ups, virtual currency, and so on. The advantages of using this model are that it increases user engagement and therefore, user retention. Another thing is that it has a low level of risk, since it does not require a lot of investment. However, keep in mind not to get carried away with this. One fatal mistake developers make is building a game which is virtually unplayable unless the users pay.

## Mobile performance and optimization

It is every developer's desire to make their game nice and beautiful with various types of post-processes, complex shaders, lighting, and so on. While this is perfectly fine when making games on PC/Consoles, you have to keep in mind that mobile platforms have certain technological limitations which can cause your game to perform poorly, thus resulting in poor sales. Here are a few tips and tricks on how you can optimize your game to achieve the best performance.

- The draw calls for your entire scene should be less than 700 to achieve the best performance.
- Although dynamic lights make the game look good, it is also heavy on the technical side. Thus, unless you absolutely require them avoid using dynamic lights. Also, it is recommended that you build the lighting before you port the game on your device.
- The triangle count for your entire scene should ideally be 500,000 or less.
- Unless you absolutely need to, you should turn off **Mobile High Definition Rendering ( HDR )**. This turns off lighting features and greatly improves the performance on mobile.
- Again, unless required, you should turn off post-process features for better performance.
- When it comes to textures, to prevent memory wastage, their resolution should be of powers of 2 (256 x 256, 512 x 512, 1024 x 1024, and so on).

You can set some of the **Rendering** options in the **Project Settings** in the **Engine** category, under the **Rendering** section.



You can turn on/off different features such as lighting, post-process, textures, and so on from here, including the last two points mentioned above.

When it comes to performance, you should first be aware of some performance tiers. They all have to do with the lighting features in the game. When developing games for mobile, depending upon your requirements and your target platform, you should keep these in mind:

- **Low Dynamic Range ( LDR )** : This mode has the highest performance and least load on the memory and processor. This mode is recommended for games that do not need lighting features and/or post-processing in their game.
- **Basic Lighting** : This is the next best mode in terms of performance. In this mode, you have access to the basic lighting features offered by UE4, such as global illumination, material shading, and so on.
- **Full HDR Lighting** : This mode is on the other end of the spectrum. It has the lowest performance and the highest load on the memory and processor. In this mode, you make use of most of the lighting and post-processing features offered by UE4. It is not recommended for use in mobile games.

#### Tip

It is recommended that you visit [https://wiki.unrealengine.com/Android\\_Device\\_Compatibility](https://wiki.unrealengine.com/Android_Device_Compatibility). This is a community driven page, wherein various community members have tested UE4 on various Android-based devices using different lighting modes. It is really helpful when developing games for Android. If you have tested UE4 on a device that has not been listed, you can post it yourself for the benefit of other community members.

## Summary

In this chapter, we discussed UMG and its user interface. Using it, we created a main menu for our game (which contains the name of the game and a button that starts the game when the player touches it). Having done that, we were then ready to package our game for Android devices.

The first step in doing that was installing the Android SDK which contains all of the build files required to package our game into a .apk file. Once installed, the next step was to specify to the engine where all of the files are located. Then, we discussed how to set up the Android device to test the game directly without having to package the game every time. Finally, we discussed how to package the final product and upload the game onto the Google Play Store for people to download.

The chapter ended with details on the few monetization models that are widely used by mobile game developers to earn revenue, and a few tips on how to properly optimize your game for mobile devices..

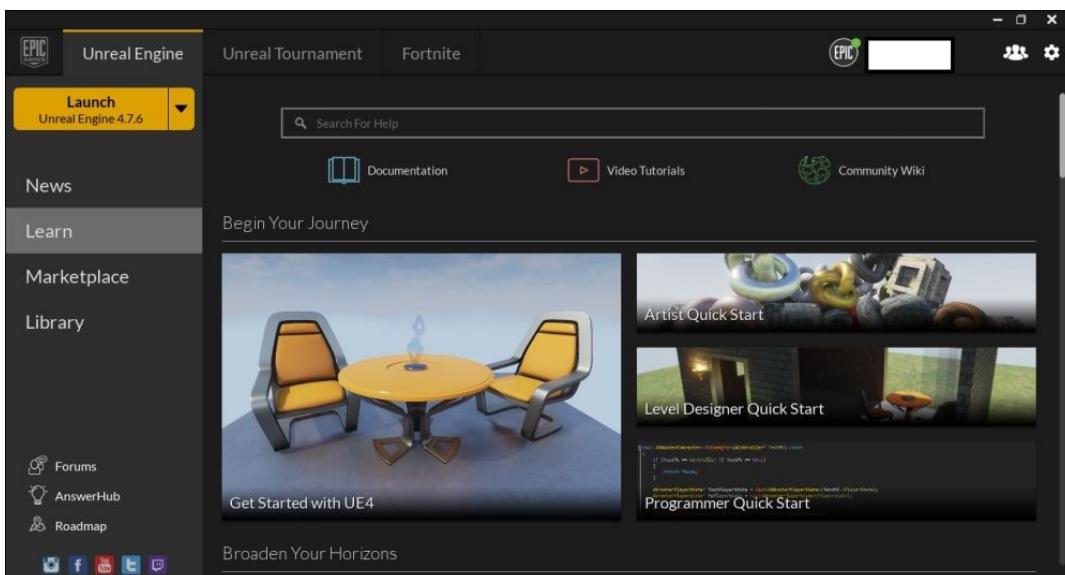
You are now equipped with all that you need to know to get started on making that game you always dreamed of. From here, the only thing you can do to get better at UE4 is practice.

## Appendix A. What Next?

In this book, we covered a whole range of topics, such as how to download and install UE4, how to create a project, how to create materials, adding/importing/migrating assets from other projects, scripting with Blueprints, and so on. Hopefully by now, you know enough about UE4 to get started. The next step is taking what you have learned, and to keep practicing.

## Learn

We briefly covered the **Learn** section in the UE4 Launcher in the first chapter. Let's talk a bit more about it.



This panel contains tutorials covering a wide variety of topics, including how to create materials, blueprints, and so on. The tutorials available here are available in various formats as follows.

- **Video Tutorials** : First off, there are video tutorials offered by Epic. To access them, click on the **Video Tutorials** button located at the top of the page, which will take you to the Unreal website.

The screenshot shows the 'Video Tutorials' section of the Unreal Wiki. It features four cards, each representing a different game project:

- 2D Sidescroller with Blueprints**: 9 Videos
- 3rd Person Game with Blueprints**: 22 Videos
- 3rd Person Power-Up Game with C++**: 19 Videos
- Endless Runner with Blueprints**: 7 Videos

Here, you can find various video tutorial series, all categorized neatly for your convenience. If you scroll down the page, you will find the browse section, where all of the topics are listed, and how many video series are there for each of them. Click on any of the topics, and you will be shown all the tutorial series available for that particular topic.

- **Documentation** : Then there is the Epic's official documentation. To access it, click on the **Documentation** hyperlink, and it will take you to the Epic's official documentation page.

The screenshot shows the 'Unreal Engine 4 Documentation' page. On the left is a navigation sidebar with categories like Getting Started, Unreal Editor Manual, Engine Features, etc. The main area has a search bar and a 'Browse by Topic' section with links to various guides:

- Get Started with UE4
- Unreal Editor Manual
- Programming Guide
- Blueprints Visual Scripting
- C++ Programming Guide
- Platform Development
- Samples & Tutorials
- Release Notes

The documentation page, like the video tutorials page, contains tutorials on various topics, but they are more in-depth and cover more topics. On the left-hand side, is the navigation panel. All of the topics are listed there. Clicking on any topic with a + sign opens up more sub-topics, from which you can choose what you want to read up on. When you click on a topic, it will open up on the right-hand side of the screen.

#### Note

If you are used to making games on Unity and want to move to Unreal 4, Epic has provided a documentation, which compares both engines' Viewport, terminology, and so on, so that you can translate your skills from Unity to Unreal 4. You can find it under the **Broaden Your Horizons** section

- **Unreal Wiki** : Unreal Wiki, accessible by clicking on the **Wiki** hyperlink at the top, is UE4's official Wiki page, made by the community, and is constantly updated by them

The screenshot shows the homepage of the Unreal Engine Community Wiki at <https://wiki.unrealengine.com/>. The page features a search bar, navigation tabs (Page, Discussion), and a main content area with a heading "Unreal Engine Community Wiki". Below the heading is a search bar with placeholder text "Search the learning resources..." and a quick search dropdown menu containing terms like Quick Start, How-To, Actor, Component, Blueprint, Materials, Lighting, Camera, HUD, Slate, C++, iOS, and Android. A message from Epic states: "We wanted you - the Unreal Engine developer community - to have a place to post tutorials as well as share your plugins, project templates, code snippets, and more. This wiki is just an early version of what we have planned. It's a little rough around the edges at the moment, but we'll put some polish on it." Another message encourages users to contribute: "In the meantime, we want you to fill it with the amazing content you create. Our community has always been known for its helpfulness and we have no doubt you will prove that once again!"

Here, you can find tutorials, plugins, code, and games, all created by the community, to help you develop your skills. You can also submit your own content or tutorials onto the wiki page for others to see. Finally, you can see a list of games that people are developing on UE4. You can also submit your game to get some publicity.

- **Engine Feature Samples** : This section contains project files, each containing a feature or various features offered by UE4.

The screenshot shows the "Engine Feature Samples" section on the Unreal Engine website. On the left is a sidebar with links to News, Learn, Marketplace, Library, Forums, AnswerHub, and Roadmap. The main content area displays six project cards:
 

- Content Examples**: A museum-style project with stands demonstrating specific features.
- Water Planes**: A collection of watery surfaces showing different water shaders and Blueprints.
- Features Tour 2014**: Epic's GDC2014 Demo! It walks you through Unreal Engine features like the new material system and Blueprints.
- Matinee**: Shows how to create highly stylized cinematic sequences using the Matinee Editor.
- Landscape Mountains**: Explore high-end landscapes in UE4 with a picturesque mountain valley scene.
- Sun Temple**: A project designed to showcase pretty mobile features.

For example, there is a project that showcases Matinee, then there is a project that demonstrates the landscape tool offered by UE4, and many more. Once you pick and download a project file, you can open it and check it out. Everything related to that feature is already set up for you, and you can see for yourself how everything works, how things are hooked up, and so on. It is also a great way of getting assets (materials, textures, blueprint classes, and so on) for your own project.

#### Note

It is highly advised that you download the **Content Examples** project. This project file contains a collection of the features offered by UE4, all in separate levels, in a museum style manner. For instance, there is a level, which showcases the Material Editor and what can be done with it, there is a level that demonstrates the animation features, and many more.

- **Gameplay Content Examples** : This section contains project files that showcase features, similar to what you would find in the Engine Feature Samples, but geared towards games specifically.

**Gameplay Concept Examples**

- SillyGeo**: This twin-stick shooter game example will teach you about cameras, enemies, damage, and more!
- Multiplayer Shootout**: This tutorial project demonstrates how to incorporate network play into your game entirely through Blueprints and UMG.
- Unreal Stick Figure 2D**: This 2D platformer project offers an in-depth look at handling complex animation switching within Blueprint, all built with Paper2D.
- Turn Based Strategy**: Check out this project designed as a foundation for square tile, turn-based strategy games in Blueprints!
- Blueprint Splines Track**: See the power of Blueprint Splines with this track generator, seen on the live training stream!
- Inventory UI with UMG**: Wes Bunn's Inventory tutorial project will show you how to use UMG to push your UI to the next level!

These projects mostly contain a blueprint, each with a mechanic or features usually found in games. They provide the framework, which you can use to build your game upon. For instance, you can find a project file, which has turn-based mechanics setup, there is a project that demonstrates the inventory UI, and so on.

- **Example Game Project** : This section is similar to the **Gameplay Content Examples** section, the only difference being that this section contains project files with a sample game already set up.

**Example Game Projects**

- Platformer Game**: Learn how to build fun platform-style experiences using this sample game.
- Couch Knights**: Have fun with Epic's competitive multiplayer game designed for Oculus DK2.
- BlackJack**: Built with Blueprints, this Blackjack card game is lightweight and optimized for mobile.
- Memory Game**: Built with Blueprints, this card-matching memory game is optimized for mobile devices!
- Swing Ninja**: Built in Blueprint by one artist, this game utilizes swinging mechanics and a simple 2D style.
- Vehicle Game**: Tap into UE4's full vehicle support! Kick the tires with this sample game.

Sample game projects are available with assets and levels set up for you to explore, learn, and utilize. There are a variety of genres you can choose from, for example, 2D platformer, 3D platformer, FPS, and so on.

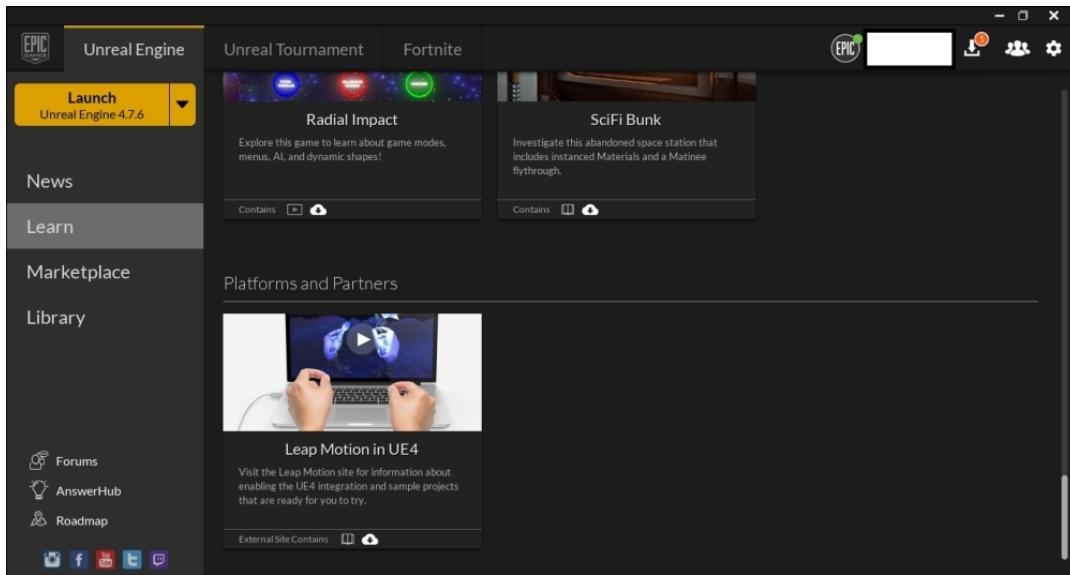
- **Community Contributions** : This section also contains project files with sample games and environments, but, unlike the project files in the other section, the content here is created by the community, hand-picked by Epic.

**Community Contributions**

- Radial Impact**: Explore this game to learn about game modes, menus, AI, and dynamic shapes!
- SciFi Bunker**: Investigate this abandoned space station that includes instanced Materials and a Matinee flythrough.

This section, at the time of writing, contains only two project files, namely Radial Impact, which demonstrates the saving menu, and so on, and SciFi Bunker, which showcases UE4's rendering techniques.

- **Platforms and Partners** : Finally, there's the **Platforms and Partners** section, which has sample projects with technology and peripherals that UE4 currently supports.



At the time of writing, there is only one sample project available, which is that of Leap Motion. Clicking on the thumbnail will take you to the official page, where you can download an experiment with Leap Motion.

## AnswerHub

**AnswerHub**, another topic briefly covered in [Chapter 1, Getting Started with Unreal 4](#), is a place where you can post questions regarding a particular topic you are having issues with, or if you require assistance with something. Clicking on the **AnswerHub** hyperlink located on the bottom-left corner of the Launch Client will take you to the AnswerHub page.

On the front page, you can see a list of questions that other members have posted. Now, it is quite likely that the problem that you are facing is not unique. So, before posting a question, it is advisable that you first search AnswerHub for the same or similar questions. Only when you cannot find a suitable solution or any solution, should you post on it.

To post a question, click on the **Post A Question** button at the top-right corner of the page, which will bring you to the [Ask a question](#) page.

You put the question at the top. Make sure that it is clear, concise, and written in such a way that a reader is able to get an idea after reading it. Below it, you can choose which section the problem is related to. For instance, if you are having difficulties in packaging your game, then in the section, you should choose **Packaging & Deployment**, and so on.

Below this is where you write the details of your issue. This is where you elaborate the question you posted. Again, try to remain brief, concise, and clear in your details. If possible, post screenshots, log files, and so on, so that it is easier for people to understand your problem.

Finally, you need to set Tags to your question. Tags make it easier for users to search for questions. When you search for a question, the search engine compares what you have written with the tags, so also make sure you properly tag your question.

It is always nice to help others. With that said, if you feel that you can help someone who has a problem, you should help him/her out by answering questions posted by other members. It is a great way to engage with the community, and you might also learn something new in the process. It is also a great way to build your karma points. If you are quite active on AnswerHub, and your answers are useful, you will get upvoted, and in turn, increase your karma points (something like Reddit).

## Forums

And finally, we have our good old-fashioned forums. Clicking on the **Forum** hyperlink at the Engine Launcher will bring you to the forums page.

Unreal Engine Forums > https://forums.unrealengine.com

What's New? Forum UT Forum UE4 AnswerHub UE4 Wiki unrealengine.com

New Posts Private Messages FAQ Community Forum Actions Quick Links

Welcome, nitishmisra Notifications My Profile Settings Advanced Search

Forum

## Unreal Engine Forums

Unreal Engine Forums

	Threads / Posts	Last Post
 Announcements and Releases Official Unreal Engine news and release information from Epic Games.	Threads: 171 Posts: 5,928	[PREVIEW] Unreal Engine 4.8 Preview by iStanislavShakov 05-12-2015, 10:09 AM
 Marketplace All Things Related to the Unreal Engine Marketplace. Discuss Marketplace Offerings, Request New Content, Promote Your Work.	Threads: 569 Posts: 8,579	[SUBMITTED] [COMING SOON] UMG Minimap [...] by Wandering_etal 05-12-2015, 10:08 AM
 Events Trade Shows, Livestreams, Dev Tours, Workshops, and Other Official Unreal Engine Events.	Threads: 114 Posts: 3,792	[USER GROUP] LOS ANGELES: Developer's... by arbyrosas 05-11-2015, 08:56 PM
 Feedback for Epic Tell us how to improve Unreal Engine 4! Sub-Forums: Documentation Feedback	Threads: 2,827 Posts: 17,488	⚠ No help in Forum and Answerhub by GreV 05-12-2015, 08:32 AM
<hr/>		
Development Discussion		
 Content Creation Talk about Level Design, Static Meshes, Physics, and more.	Threads: 4,714 Posts: 26,315	How to import model into UE4... by alonehdby 05-12-2015, 10:54 AM
 Animation Animation discussion, including Animation Blueprint, Persona, Skeletal Meshes, and more.	Threads: 720 Posts: 4,102	Importing baked animations by mcsolar 05-12-2015, 08:48 AM
 Rendering For discussions about Materials, Textures, Lighting, Particle Systems, and Post Process Effects.	Threads: 2,630 Posts: 13,470	The Custom Depth Mask... by jordandjones 05-12-2015, 09:33 AM
 Blueprint Visual Scripting Build powerful visual scripts without code.	Threads: 5,938 Posts: 31,799	UMG Animate ZOrder... by Fen 05-12-2015, 09:33 AM

The forums are neatly categorized, based on various topics, so that it is easier for you to find the relevant thread to post on. The sections are as follows:

- **Unreal Engine** : This is where all of the topics related to UE4 in general are posted, such as the latest announcements, updates, the marketplace, and so on.
  - **Development discussion** : This section contains topics related to developing games or other projects in UE4. These topics range from things such as rendering, animation, content creation, architectural visualization, to more technical topics, such as C++ programming, Android development, iOS development, and so on.
  - **Community** : This is where you can engage with the community more directly. For instance, if you are interested in making a game or product, but lack the talent, or if you are interested in joining a team to make a game, you can go to the [Got Skill? Looking for Talent?](#) section, and post your requirement or your profile there, and hopefully, get some responses. Otherwise, if you are already working on a project, and wanted to get some feedback, you can post it on the [Work in Progress](#) section, and get some constructive feedback from other community members. It is also a great way to get your game noticed.
  - **UE4 for Schools** : This section contains topics and discussion threads related to academia. Here, tutors can discuss the curriculum and how to go about teaching UE4 to students, and also give feedback to other educators. If you are a student, you can discuss UE4 with other fellow students, and get some support.
  - **International** : Finally, if you are interested in engaging with community members near your geographical location, you can post a discussion thread in the relevant subsection. For instance, if you want to make a game, but prefer that your team members are nearby, you can post a thread related to that.

## Summary

Here, we looked at the various ways you can learn and build upon your skills regarding UE4 that you have acquired after reading this guide.

The good news is that UE4 itself has a huge arsenal of tutorials in almost every format available, be it written, video, project files, live stream and so on. You will learn a lot about the Engine just by going over them alone.

If you still need some help regarding a particular aspect or feature or if you have an issue that you cannot seem to resolve, or just need some assistance, you can always post it on AnswerHub and have it resolved by other community members, and/or by the Epic staff themselves. Also, if you find a question, posted by some other community member, that you think you know the answer to, why not just give them a hand? It will improve your karma and keep the community active.

Finally, there are the forums, which you can also use to get some help, discuss the latest updates, and marketplace additions, show your work and get some feedback, give feedback on other peoples work, recruit people, or be a part of a team, and so on.

## Appendix A. Bibliography

This course is packaged up keeping your journey in mind. It includes content from the following Packt products:

- *Learning Unreal Engine Game Development*, Joanna Lee
- *Unreal Engine Game Development Cookbook*, John P. Doran
- *Learning Unreal Engine Android Game Development*, Nitish Misra

## Index

### A

- actors
  - placing / [Placing actors](#)
  - adding, from All Classes section / [Adding actors from All Classes](#)
- AddOnScreenDebugMessage function
  - reference / [How to do it...](#)
- adjusted camera functions
  - about / [Adjusted camera functions](#)
  - Zoom / [Zoom](#)
  - Field of view (FOV) / [Field of view](#)
  - Depth of field (DOF) / [Depth of field](#)
- AI
  - implementing, in games / [Implementing AI in games](#)
- AIController
  - linking, to Character Blueprint / [Linking AIController to the Character Blueprint](#)
  - configuring / [Configuring AIController](#)
  - movement speed, adjusting / [Adjusting movement speed](#)
- AI logic
  - setting up / [Tutorial – setting up AI logic](#)
  - basic animation, adding / [Adding basic animation](#)
- Alerts section, Developer Console / [ALERTS](#)
- ALL APPLICATIONS, Developer Console
  - about / [ALL APPLICATIONS](#)
  - APK panel / [APK](#)
  - Store Listing panel / [Store Listing](#)
  - content rating / [Content Rating](#)
  - Pricing and Distribution / [Pricing](#)
  - In-app Products / [In-app Products](#)
  - Services & APIs / [Services](#)
- All Classes section
  - about / [All Classes](#)
  - actors / [All Classes](#)
  - Ambient Sound Actor / [All Classes](#)
  - Camera actor / [All Classes](#)
  - Default Pawn actor / [All Classes](#)
  - Landscape actor / [All Classes](#)
  - Level Bounds actor / [All Classes](#)
  - Matinee Actor / [All Classes](#)
  - Nav Link Proxy actor / [All Classes](#)
  - Target Point actor / [All Classes](#)
  - Text Renderer actor / [All Classes](#)
  - Class Blueprint actor / [All Classes](#)
  - actors, adding from / [Adding actors from All Classes](#)
  - Camera Actor, adding from / [Camera](#)
  - Matinee actors, adding from / [Matinee actors](#)
  - Target Point actors, adding from / [Target Point , Room3 , Room4](#)
- Ambient Sound Actor
  - about / [Unreal sound formats and terminologies](#)
- Anchor Medallion / [How to do it...](#)
- Android device
  - setting up / [Setting up the Android device](#)
- Android SDK
  - installing / [Installing the Android SDK](#)
- animation
  - about / [What is animation?](#)
  - stop-motion animation / [What is animation?](#)
  - computer animation / [What is animation?](#)
  - 3D model, preparing for / [Preparing before animation](#)
  - creating / [How is animation created?](#)
  - importing, from Maya / [Importing animation from Maya/3ds Max](#)
  - importing, from 3ds Max / [Importing animation from Maya/3ds Max](#)
- animation, in Unreal Engine 4 / [What Unreal Engine 4 offers for animation in games](#)
- Animation and Rigging Toolset (ART)
  - about / [What Unreal Engine 4 offers for animation in games](#)
- Animation Blueprint
  - setting up, for Blend Animation usage / [Tutorial – setting up the Animation Blueprint to use a Blend Animation](#)
- animation pack
  - importing, from Marketplace / [Tutorial – importing the animation pack from Marketplace](#)
- animations
  - blending / [Why do we need to blend animations?](#)
- Animations panel, UMG Editor / [The Animations panel](#)
- AnimGraph
  - about / [AnimGraph](#)
- AnswerHub section, UE4 Launcher
  - URL / [Downloading and installing UE4](#)
  - about / [AnswerHub](#)
- application programming interfaces (APIs) / [APIs – DirectX and OpenGL](#)
- artificial intelligence
  - about / [Artificial intelligence](#)
- assets
  - migrating / [Migrating and importing assets](#) , [Migrating assets](#)
  - importing / [Migrating and importing assets](#) , [Importing assets](#)
- assets, UE4
  - Texture Files / [Migrating and importing assets](#)

- Static Meshes/Skeletal Meshes / [Migrating and importing assets](#)
- Audio Files / [Migrating and importing assets](#)
- Script Files / [Migrating and importing assets](#)
- IES Light Profiles / [Migrating and importing assets](#)
- Cubemap Texture / [Migrating and importing assets](#)
- True Type Fonts / [Migrating and importing assets](#)
- audio, obtaining into Unreal
  - about / [Getting audio into Unreal](#)
  - audio format / [The audio format](#)
  - sampling rate / [The sampling rate](#)
  - bit depth / [Bit depth](#)
  - supported sound channels / [Supported sound channels](#)
- audio quality
  - about / [Audio quality](#)
- Audio Volume
  - about / [Audio Volume](#)

## B

- baking
  - about / [A special form of texture maps – Normal Maps](#)
- Basic AI
  - scripting / [Scripting basic AI](#)
- basic class actors
  - adding to game / [Adding basic class actors to the game](#)
- basic classes
  - about / [Basic classes](#)
  - Empty Actor / [Basic classes](#)
  - Empty Character / [Basic classes](#)
  - Empty Pawn / [Basic classes](#)
  - Player Start / [Basic classes](#)
  - Point Light / [Basic classes](#)
  - Cube / [Basic classes](#)
  - Sphere / [Basic classes](#)
  - Cylinder / [Basic classes](#)
  - Cone / [Basic classes](#)
  - Box trigger / Sphere trigger / [Basic classes](#)
- Basic Lighting / [Mobile performance and optimization](#)
- beginners guide, Unreal Editor
  - start menu / [The start menu](#)
  - Project Browser / [Project Browser](#)
  - Content Browser / [Content Browser](#)
  - Toolbar / [Toolbar](#)
  - Viewport / [Viewport](#)
  - Scene Outliner / [Scene Outliner](#)
  - Modes window / [Modes](#)
- Behavior Tree
  - about / [Understanding a Behavior Tree](#)
  - logic, designing of / [Exercise – designing the logic of a Behavior Tree](#)
  - implementing in Unreal Engine 4 / [How to implement a Behavior Tree in Unreal Engine 4](#)
  - creating / [Creating a Behavior Tree](#)
  - using / [Using the Behavior Tree](#)
  - custom task, creating for / [Creating a custom task for the Behavior Tree](#)
  - PickTargetLocation custom task, using in / [Using the PickTargetLocation custom task in BT](#)
- Blackboard
  - about / [How to implement a Behavior Tree in Unreal Engine 4](#)
- BlackBoardData
  - creating / [Creating the BlackBoardData](#)
  - variable, adding into / [Adding a variable into BlackBoardData](#)
- blank map
  - level, creating from / [Creating a level from a new blank map](#)
- Blend Animation
  - about / [Why do we need to blend animations?](#)
  - creating / [Tutorial – creating a Blend Animation](#)
  - Animation Blueprint, setting up for usage of / [Tutorial – setting up the Animation Blueprint to use a Blend Animation](#)
- Blocking Volume
  - about / [Blocking Volume](#)
- Bloques
  - about / [What to expect , Bloques](#)
  - concept / [Concept](#)
  - controls / [Controls](#)
  - project, creating for game / [Creating the project for the game](#)
- Blueprint / [The history of Unreal Engine](#)
  - about / [Introducing Blueprint](#)
  - Level Blueprint / [Level Blueprint](#)
  - working / [How Blueprint works](#)
- Blueprint AIController
  - creating / [Creating the Blueprint AIController](#)
- Blueprint character
  - creating / [Creating the Blueprint character](#)
- Blueprint Class
  - about / [How Blueprint works , The Blueprint class](#)
  - creating / [Creating a Blueprint class](#)
- Blueprint Class user interface
  - Components Panel / [Creating a Blueprint class](#)
  - Graph Editor / [Creating a Blueprint class](#)
  - Event Graph / [Creating a Blueprint class , The Event Graph](#)
  - Construction Script / [Creating a Blueprint class , The Construction Script](#)
  - Viewport / [Viewport](#)
- Blueprint Editor / [How to do it...](#)
- Blueprint First Person Project
  - creating / [Creating a new project](#)
- Blueprints
  - about / [Introduction](#)
  - Level Blueprint / [Introduction](#)
  - Class Blueprints / [Introduction](#)
  - using / [When to use C++/Blueprints](#)
- Blueprint visual scripting system

- about / [The Blueprint visual scripting system](#)
- bots / [The history of Unreal Engine](#)
- box brush / [Default BSP brush shapes](#)
- brushes
  - converting, to static meshes/volumes / [Converting brushes to static meshes or volumes](#) , [Getting ready](#) , [How to do it...](#)
- BSP
  - about / [Background](#)
- BSP Box brush
  - used, for creating ground / [Creating the ground using the BSP Box brush](#)
- BSP Brush
  - about / [BSP Brush](#)
  - background / [Background](#)
  - type / [Brush type](#)
  - solidity / [Brush solidity](#)
  - versus Static Mesh / [BSP Brush versus Static Mesh](#)
- BSP brushes
  - about / [BSP brushes](#)
  - default BSP brushes shapes / [Default BSP brush shapes](#)
  - editing / [Editing BSP brushes](#)
  - rooms, blocking out with / [Blocking out the rooms with BSP brushes](#) , [The first room](#) , [The second room](#) , [The third room](#) , [The fourth room](#)

## C

- C++
  - using / [When to use C++/Blueprints](#)
- Camera Actor
  - adding, from All Classes section / [Camera](#)
- Camera Blocking Volume
  - about / [Camera Blocking Volume](#)
- camera movement
  - about / [Camera movement](#)
  - tilt / [Tilt](#)
  - pan / [Pan](#)
  - dolly/track/truck / [Dolly/track/truck](#)
  - pedestal / [Pedestal](#)
- casting
  - about / [How to do it...](#)
- central processing unit (CPU) / [Rendering pipeline](#)
- Character Blueprint
  - Mesh, adding to / [Adding and configuring Mesh to a Character Blueprint](#)
  - Mesh, configuring to / [Adding and configuring Mesh to a Character Blueprint](#)
  - AIController, linking to / [Linking AIController to the Character Blueprint](#)
- cinematics
  - about / [Introducing cinematics](#)
- cinematic techniques
  - about / [Introducing cinematics](#) , [Cinematic techniques](#)
  - adjusted camera functions / [Adjusted camera functions](#)
  - camera movement / [Camera movement](#)
  - scene, capturing / [Capturing a scene](#)
- Class Blueprints
  - about / [Introduction](#)
  - Level Blueprint, converting to / [How to do it...](#)
- client-server model
  - reference / [Creating a main menu](#)
- Close Up Shot (CU) / [Shot types](#)
- collectables
  - collecting with networking / [Networking 101 – creating collectables with networking](#) , [Getting ready](#) , [How to do it...](#)
- collision configuration properties
  - about / [Collision configuration properties](#)
  - Simulation Generates Hit Events / [Simulation Generates Hit Events](#)
  - Generate Overlap Events / [Generate Overlap Events](#)
  - Collision Presets / [Collision Presets](#)
  - Collision Enabled / [Collision Enabled](#)
  - Object Type / [Object Type](#)
  - Collision Responses / [Collision Responses](#)
  - collision hulls / [Collision hulls](#)
- Collision Responses option
  - about / [Collision Responses](#)
  - Trace Responses / [Trace Responses](#)
  - Object Responses / [Object Responses](#)
- collisions
  - about / [Collisions](#)
- comments / [How to do it...](#)
- compiler / [How to do it...](#)
- Compiler Results panel, Level Blueprint user interface / [The Compiler Results panel](#)
- computer animation
  - about / [What is animation?](#)
- conditional / [How to implement a Behavior Tree in Unreal Engine 4](#)
- cone brush / [Default BSP brush shapes](#)
- config files
  - reference / [How to do it...](#)
- Constructive Solid Geometry (CSG)
  - about / [Background](#)
- constructor / [How to do it...](#)
- content
  - importing / [Importing your own content](#) , [Getting ready](#) , [How to do it...](#)
- Content Browser
  - about / [The Content Browser overview](#) , [How to do it...](#) , [Content Browser](#)
- Content Browser, User Interface
  - about / [Content Browser](#)
- Content Browser UI
  - reference / [How to do it...](#)
- control keys
  - about / [Control keys](#)
- controls
  - about / [Hotkeys and controls](#)
- Cooper Industries
  - reference link, for downloading IES Light Profiles / [Downloading IES Light Profiles](#)

- Cull Distance Volume
  - about / [Cull Distance Volume](#)
- Curved Stair Brush / [Default BSP brush shapes](#)
- Curve Editor, Unreal Matinee user interface
  - about / [The Curve Editor](#)
  - options / [The Curve Editor](#)
- custom blueprint nodes
  - creating / [Creating custom blueprint nodes](#), [How to do it...](#)
  - reference / [How to do it...](#)
- custom material
  - creating / [Creating a custom material](#), [Getting ready](#), [How to do it...](#)
- custom materials
  - creating, simple texture used / [Creating custom material using simple textures](#)
  - used, for transforming level / [Using custom materials to transform the level](#)
- custom sounds
  - adding to level / [Exercise – adding custom sounds to a level](#)
- custom task
  - creating, for Behavior Tree / [Creating a custom task for the Behavior Tree](#)
- cut scenes
  - about / [Introducing cinematics](#)
  - need for / [Why do we need cut scenes?](#)
- cylinder brush / [Default BSP brush shapes](#)

## D

- 3D model
  - animating / [Understanding how to animate a 3D model](#)
  - preparing for animation / [Preparing before animation](#)
- 3ds Max
  - animation, importing from / [Importing animation from Maya/3ds Max](#)
  - [Introduction](#)
- day night cycle
  - creating / [Creating a Day/Night cycle](#), [How to do it...](#), [See also](#)
- Debug Game Editor mode / [How to do it...](#)
- Deca Tree
  - about / [How are sounds recorded?](#)
- decorative assets
  - materials, creating for / [Decorative assets](#)
- default BSP brushes shapes
  - about / [Default BSP brush shapes](#)
  - box brush / [Default BSP brush shapes](#)
  - sphere brush / [Default BSP brush shapes](#)
  - cylinder brush / [Default BSP brush shapes](#)
  - cone brush / [Default BSP brush shapes](#)
  - linear stair brush / [Default BSP brush shapes](#)
  - curved stair brush / [Default BSP brush shapes](#)
  - spiral stair brush / [Default BSP brush shapes](#)
- Delta Seconds / [How to do it...](#)
- Depth of field (DOF)
  - about / [Depth of field](#)
- design principles, particle system
  - about / [The design principles of a particle system](#)
  - research / [Research](#)
  - iterative creative process / [The iterative creative process](#)
- Details panel, Level Blueprint user interface / [The Details panel](#)
- Details panel, UMG Editor / [The Details panel](#)
- Details panel, Unreal Matinee user interface
  - about / [The Details panel](#)
- Details panel, User Interface
  - about / [Details](#)
- Developer Console
  - about / [Developer Console](#)
  - ALL APPLICATIONS / [ALL APPLICATIONS](#), [APK](#), [Store Listing](#), [Content Rating](#), [Pricing](#), [In-app Products](#), [Services](#)
  - game services / [GAME SERVICES](#), [Game details](#), [Linked apps](#), [Achievements](#), [Leaderboards](#), [Testing](#), [Publishing](#)
  - Reports panel / [REPORTS](#)
  - Settings section / [SETTINGS](#)
  - Alerts section / [ALERTS](#)
- device compatibility, Android
  - URL / [Mobile performance and optimization](#)
- Direct3D 11 graphics pipeline
  - reference link / [Shaders](#)
- directional light
  - about / [Lighting](#)
- Directional Light
  - adding / [Adding and configuring a Directional Light](#)
  - configuring / [Adding and configuring a Directional Light](#)
- Directional Lights
  - about / [How to do it...](#)
  - reference / [How to do it...](#)
- Direct Light
  - about / [Common light/shadow definitions](#)
- director group / [How to do it...](#)
- Director Group / [Room 1](#)
- DirectX
  - about / [APIs – DirectX and OpenGL](#), [DirectX](#)
- DirectX 12
  - reference link / [DirectX12](#)
- DirectX12
  - about / [DirectX12](#)
  - pipeline state transformation / [Pipeline state representation](#)
  - work submission / [Work submission](#)
  - resource access / [Resource access](#)
- DirectXRedist
  - about / [Windows DirectXRedist](#)
- domain-shader stage / [Shaders](#)
- door
  - opening, creating for / [Creating an opening for a door](#)
- door, animating

- about / [Animating the door](#)
- room 1 / [Room 1](#)
- room 2 / [Room 2](#)
- bridge for AI character / [A bridge for the AI character](#)
- doors
  - materials, creating for / [Doors](#)
- Dots per inch (DPI) / [The Graph Editor](#)
- dynamic enemy healthbars
  - implementing / [Dynamic enemy healthbars](#) , [How to do it...](#)

## E

- Editor
  - about / [Meet the Editor](#)
- Emissive property
  - reference / [How to do it...](#)
- Engine Launcher
  - about / [The Engine Launcher](#)
  - News panel / [News](#)
  - Learn section / [Learn](#)
  - Marketplace / [Marketplace](#)
  - library / [Library](#)
  - UE4 Links / [UE4 Links](#)
- environment
  - lighting up / [Lighting up the environment](#)
- environment artist
  - about / [Introduction](#)
- Event
  - playing, via Blueprints / [How to do it...](#)
- EventGraph
  - about / [EventGraph](#)
  - nodes, adding in / [Nodes to add in EventGraph](#)
- Event Graph, Level Blueprint user interface / [The Event Graph](#)
- Event Nodes / [How Blueprint works](#)
- example map
  - meshing / [Meshing an example map](#) , [How to do it...](#)
- existing animation
  - assigning, to Persona / [Tutorial – assigning existing animation to a Pawn](#)
- exterior level
  - building, Sculpt mode used / [Building an exterior level using the Sculpt mode](#) , [How to do it...](#)
- Extreme Close Up Shot (ECU) / [Shot types](#)
- Extreme Long Shot (ELS) / [Shot types](#)
- Extreme Wide Shot (EWS) / [Shot types](#)

## F

- Facebook, UE4
  - URL / [UE4 Links](#)
- Field of view (FOV)
  - about / [Field of view](#)
- field of view (FOV) / [How to do it...](#)
- Field of View (FOV) Angle track / [Room 1](#)
- fill light / [How to do it...](#)
- finishing touches
  - applying, to room / [Applying finishing touches to a room](#)
- fireplace particle system
  - creating / [Example – creating a fireplace particle system](#)
  - P\_Fireplace, crafting / [Crafting P\\_Fireplace](#)
  - solo emitters, observing of / [Observing the solo emitters of the system](#)
  - nonessential emitters, deleting / [Deleting non-essential emitters](#)
  - Flame emitter, editing / [Focusing on editing the Flame emitter](#)
  - viewing / [Looking at the complete particle system](#)
- First-Person Shooter (FPS) game / [How to do it...](#)
- Flame emitter
  - editing / [Focusing on editing the Flame emitter](#)
- flashlight
  - creating / [Getting ready](#) , [How to do it...](#)
  - adding, to existing Blueprint / [Adding to an existing Blueprint – flashlight, part 2](#) , [Getting ready](#) , [How to do it...](#)
- Flatten tool
  - about / [Creating rivers with the Flatten tool](#)
  - for creating rivers / [Getting ready](#) , [How to do it...](#)
- flickering light
  - building, Level Blueprint used / [Building a flickering light](#) , [How to do it...](#)
- float / [How to do it...](#)
- Folder Hierarchy
  - about / [How to do it...](#)
- Foliage tool
  - for placing trees and rocks / [Placing trees and rocks using the Foliage tool](#) , [How to do it...](#)
- forums, UE4 Launcher
  - URL / [Downloading and installing UE4](#)
  - about / [Forums](#)
  - Unreal Engine / [Forums](#)
  - development discussion / [Forums](#)
  - community / [Forums](#)
  - UE4 for Schools / [Forums](#)
  - International / [Forums](#)
- framing
  - about / [Framing](#)
  - rules / [Some framing rules](#)
- Freemium model / [Monetization methods](#)
- Full HDR Lighting / [Mobile performance and optimization](#)
- Function Nodes / [How Blueprint works](#)

## G

- game
  - requisites / [What goes into a game?](#)
  - basic class actors, adding to / [Adding basic class actors to the game](#)
  - Visual Effect actors, adding to / [Adding Visual Effect actors to the game – Post Process Volume](#)
  - volumes, adding to / [Adding Volumes to the game](#)

- Level Blueprint, using in / [Using Level Blueprint in the game](#)
- uploading, on Play Store / [Uploading the game on the Play Store](#)
- game, Level Blueprint
  - key cube, placing / [Key cube pickup and placement](#)
  - key cube, picking up / [Key cube pickup and placement](#)
- game, publishing
  - about / [Publishing your game](#)
  - Google services, activating / [Activating Google services](#)
  - project, preparing for shipping / [Preparing the project for shipping](#)
- game development
  - about / [Game development](#)
  - artists / [Artists](#)
  - cinematic creators / [Cinematic creators](#)
  - sound designers / [Sound designers](#)
  - game designers / [Game designers](#)
  - programmers / [Programmers](#)
- game engine
  - about / [What is a game engine?](#)
- game features
  - high-level breakdown / [What to expect](#)
- game level
  - extending / [Exercise – extending your game level \(optional\)](#)
  - useful tips / [Useful tips](#)
  - guidelines / [Guidelines](#)
- gameplay
  - development environment, setting up / [Setting up your development environment](#), [How to do it...](#)
  - text, displaying during runtime / [Displaying text during runtime](#), [How to do it...](#)
- games
  - AI, implementing in / [Implementing AI in games](#)
  - music, producing for / [How do we produce sound and music for games?](#)
  - sound, producing for / [How do we produce sound and music for games?](#)
  - saving, with C++ / [Saving or loading games and keyboard input with C++](#), [How to do it...](#)
  - loading, with C++ / [Saving or loading games and keyboard input with C++](#), [How to do it...](#)
- game services, Developer Console
  - about / [GAME SERVICES](#)
  - Game Details / [Game details](#)
  - Linked Apps / [Linked apps](#)
  - Events / [Events](#)
  - Achievements / [Achievements](#)
  - Leaderboards / [Leaderboards](#)
  - Testing / [Testing](#)
  - Publishing / [Publishing](#)
- geometry-shader stage / [Shaders](#)
- geometry brushes
  - materials, applying / [Getting ready](#), [How to do it...](#)
- Geometry Brushes (BSP) / [Getting ready](#)
- Gimp / [Introduction](#)
- Graph Editor, UMG Editor
  - about / [The Graph Editor](#)
  - Widget Layout Transformation / [The Graph Editor](#)
  - Widget Render Transformation / [The Graph Editor](#)
  - Grid Snapping / [The Graph Editor](#)
  - Grid Snap Value / [The Graph Editor](#)
  - Zoom to Fit / [The Graph Editor](#)
  - Preview Size / [The Graph Editor](#)
- graphic programmers
  - about / [Materials](#)
- graphics
  - example flow / [Shaders](#)
- graphics processing unit (GPU) / [Rendering pipeline](#)
- grid snapping / [How to do it...](#)
- ground
  - creating, BSP Box brush used / [Creating the ground using the BSP Box brush](#)
  - material, adding to / [Adding material to the ground](#)
- Grouping
  - reference / [How to do it...](#)
- Groups
  - about / [Using Groups](#)
  - using / [Getting ready](#), [How to do it...](#)
- guidelines, game level extension
  - area expansion / [Area expansion](#), [Part 2 – creating a big room \(living and kitchen area\)](#), [Part 3 – creating a small room along the walkway](#)
  - windows, creating / [Creating windows and doors](#), [Part 2 – creating an open window for the window seat](#), [Part 3 – creating windows for the room](#), [Part 4 – creating the main door area](#)
  - doors, creating / [Creating windows and doors](#), [Part 2 – creating an open window for the window seat](#), [Part 3 – creating windows for the room](#), [Part 4 – creating the main door area](#)
  - basic furniture creation, placing / [Creating basic furniture](#), [Part 3 – creating the window seat area](#), [Part 5 – creating the kitchen cabinet area](#)

## H

- heads-up display (HUD)
  - about / [Adding the main menu using Unreal Motion Graphics](#)
- Health/Damage system
  - creating / [Creating a Health/Damage system](#), [part 1 – taking damage](#), [How to do it...](#)
- height map
  - reference link / [Importing height maps and layers](#)
- Hierarchy panel, UMG Editor / [The Hierarchy panel](#)
- High Level Shading Language (HLSL)
  - about / [High Level Shading Language](#)
- hills
  - creating, Landscape tool used / [Exercise – creating hills using the Landscape tool](#)
- hotkeys
  - about / [Hotkeys and controls](#)
- Hot Reload function
  - about / [Unreal programming](#)
- hull-shader stage / [Shaders](#)

## I

- IES Light Profiles
  - downloading / [Downloading IES Light Profiles](#)
- IES Profile

- using / [Using the IES Profile](#)
- IES Profiles
  - importing, into Unreal Engine Editor / [Importing IES Profiles into the Unreal Engine Editor](#)
  - using / [Using IES Profiles](#)
- in-app advertisement / [Monetization methods](#)
- In-App purchases / [Monetization methods](#)
- Indirect Light
  - about / [Common light/shadow definitions](#)
- input-assembler stage / [Shaders](#)
- Instagram, UE4
  - URL / [UE4 Links](#)
- installer, for Windows
  - creating / [Creating an installer for Windows](#), [How to do it...](#)
- installing
  - UE4 / [Downloading and installing UE4](#)
- Integrated Developers Environment (IDE) / [Setting up your development environment](#)
- interactions
  - about / [Interactions](#)
- items
  - seeing, through walls / [Seeing through walls](#), [Getting ready](#), [How to do it...](#)

## K

- keyboard input
  - using / [How to do it...](#)
- keyboard tips, for building level / [Some keyboard tips](#)
- Key Cubes material
  - creating / [Key Cubes](#)
- Kill Z Volume
  - about / [Kill Z Volume](#)
- KillZ Volume / [Modes](#)
- Kismet / [The history of Unreal Engine](#)
- Kismet system
  - about / [Introducing Blueprint](#)

## L

- landscape
  - creating / [Creating a landscape](#), [How to do it...](#)
- landscape creation options
  - about / [Landscape creation options](#)
  - multiple landscapes / [Multiple landscapes](#)
  - custom material, using / [Using custom material](#)
  - height maps, importing / [Importing height maps and layers](#)
  - layers, importing / [Importing height maps and layers](#)
  - Scale settings / [Scale](#)
  - number of components / [The number of components](#)
  - Section Size / [Section Size](#)
- Landscape Splines tool
  - reference / [How to do it...](#)
- Landscape tool
  - about / [Introducing terrain manipulation](#)
  - used, for creating hills / [Exercise – creating hills using the Landscape tool](#)
- Launcher folder, Windows directory structure
  - about / [Launcher](#)
  - Backup / [Launcher](#)
  - Engine / [Launcher](#)
  - PatchStaging / [Launcher](#)
  - VaultCache / [Launcher](#)
- Learn section, Engine Launcher / [Learn](#)
- Learn section, UE4 Launcher
  - about / [Learn](#)
  - Video Tutorials / [Learn](#)
  - Documentation / [Learn](#)
  - Unreal Wiki / [Learn](#)
  - Engine Feature Samples / [Learn](#)
  - Gameplay Content Examples / [Learn](#)
  - Example Game Project / [Learn](#)
  - Community Contributions / [Learn](#)
  - Platforms and Partners / [Learn](#)
- level
  - creating, from blank map / [Creating a level from a new blank map](#)
  - light, adding to / [Adding light to a level](#)
  - objects, positioning in / [Useful tip – positioning objects in a level](#)
  - sky, adding to / [Adding the sky to a level](#)
  - objects, rotating in / [Useful tip – rotating objects in a level](#)
  - viewing / [Viewing a level that's been created](#)
  - saving / [Saving a level](#)
  - transforming, custom materials used / [Using custom materials to transform the level](#)
  - custom sounds, adding to / [Exercise – adding custom sounds to a level](#)
  - building / [Building out a level](#), [Getting ready](#), [How to do it...](#)
  - elements, duplicating / [Seeing double – duplicating elements](#)
- Level Blueprint
  - about / [Level Blueprint](#), [Introduction](#), [How Blueprint works](#)
  - for building flickering light / [Building a flickering light](#), [How to do it...](#)
  - converting, to Class Blueprints / [How to do it...](#)
  - using, in game / [Using Level Blueprint in the game](#)
- Level Blueprint user interface
  - about / [The Level Blueprint user interface](#)
  - tab bar / [The tab and menu bars](#)
  - menu bar / [The tab and menu bars](#)
  - toolbar / [The toolbar](#)
  - Details panel / [The Details panel](#)
  - Compiler Results panel / [The Compiler Results panel](#)
  - My Blueprint panel / [My Blueprint panel](#)
  - Event Graph / [The Event Graph](#)
- level designer
  - about / [Introduction](#)
- level of detail (LOD) system / [Landscape – building large outdoor worlds and foliage](#)

- levels
  - streaming / [Streaming levels](#) , [How to do it...](#)
- Level Streaming Volume
  - about / [Level Streaming Volume](#) , [Cull Distance Volume](#)
- library, Engine Launcher / [Library](#)
- light
  - adding, to level / [Adding light to a level](#)
- lighting
  - about / [Lighting](#) , [Introduction](#) , [Lighting](#)
  - reference / [How to do it...](#)
  - overview / [Lighting overview – learning the types of lights](#)
  - types of lights / [Lighting overview – learning the types of lights](#) , [Getting ready](#)
- Light Map
  - about / [Common light/shadow definitions](#)
- Lightmass Importance Volume / [How to do it...](#)
  - adding / [Adding Lightmass Importance Volume](#)
  - about / [Lightmass Importance Volume](#)
  - using / [Lightmass Importance Volume](#)
- lights
  - about / [Lights](#)
  - Point Light / [Configuring a Point Light with more settings](#)
  - Spot Light / [Adding and configuring a Spot Light](#)
  - Directional Light / [Adding and configuring a Directional Light](#)
  - Sky Light / [Example – adding and configuring a Sky light](#)
  - mobility / [Mobility](#)
- Linear Stai Brush / [Default BSP brush shapes](#)
- linker / [How to do it...](#)
- Lithonia
  - reference link, for downloading IES Light Profiles / [Downloading IES Light Profiles](#)
- Lit mode / [How to do it...](#)
- LOD
  - about / [Level of detail](#)
- logic
  - designing, of Behavior Tree / [Exercise – designing the logic of a Behavior Tree](#)
- Long Shot (LS) / [Shot types](#)
- Low Dynamic Range (LDR) / [Mobile performance and optimization](#)

## M

- main menu
  - about / [Creating a main menu](#)
  - creating / [How to do it...](#) , [Creating the main menu](#)
  - adding, UMG Editor used / [Adding the main menu using Unreal Motion Graphics](#)
- map
  - configuring, as start level / [Configuring a map as a start level](#)
- Marketplace
  - animation pack, importing from / [Tutorial – importing the animation pack from Marketplace](#)
- Marketplace, Engine Launcher / [Marketplace](#)
- marquee selection / [Some keyboard tips](#)
- massively online games (MMOG) / [The history of Unreal Engine](#)
- Material
  - creating, in Unreal / [Creating a Material in Unreal](#)
  - versus Texture / [Materials versus Textures](#)
- material
  - adding, to ground / [Adding material to the ground](#)
  - adding, to walls / [Adding materials to the walls](#)
- Material Editor
  - about / [Material Editor](#) , [The Material Editor](#) , [The Material Editor](#)
  - Cascade particle system / [The Cascade particle system](#)
  - Persona skeletal mesh animation / [The Persona skeletal mesh animation](#)
  - landscape / [Landscape – building large outdoor worlds and foliage](#)
  - tab bar / [The tab and menu bar](#)
  - menu bar / [The tab and menu bar](#)
  - toolbar / [The toolbar](#)
  - Palette panel / [The Palette panel](#)
  - Stats panel / [The Stats panel](#)
  - Details panel / [The Details panel](#)
  - Viewport panel / [The Viewport panel](#)
  - Graph panel / [The Graph panel](#)
- material manipulation
  - about / [Materials](#)
- materials
  - about / [Materials](#) , [Creating a custom material](#) , [Materials](#)
  - applying, to geometry brushes / [Applying materials to geometry brushes](#) , [How to do it...](#)
  - applying / [Applying materials](#)
  - creating / [Creating the materials](#)
  - creating, for pedestals / [Pedestals](#)
  - creating, for doors / [Doors](#)
  - creating, for decorative assets / [Decorative assets](#)
- Matinee
  - creating / [An introduction to Matinee](#) , [Getting ready](#) , [How to do it...](#)
  - playing, via Blueprints / [Playing a Matinee via Blueprints](#) , [How to do it...](#)
- Matinee actor
  - adding, from All Classes section / [Matinee actors](#)
- Matinee Editor
  - about / [Matinee Editor](#)
- Matinee Editor window
  - about / [How to do it...](#)
  - toolbar / [How to do it...](#)
  - Curve Editor / [How to do it...](#)
  - Details tab / [How to do it...](#)
  - tracks / [How to do it...](#)
- Matinee tool
  - about / [An introduction to Matinee](#)
- Maya
  - animation, importing from / [Importing animation from Maya/3ds Max](#)
- Medium Shot (MS) / [Shot types](#)

- menu
  - animating / [Animating a menu](#) , [How to do it...](#)
- menu bar, Level Blueprint user interface
  - File / [The tab and menu bars](#)
  - Edit / [The tab and menu bars](#)
  - View / [The tab and menu bars](#)
  - Debug / [The tab and menu bars](#)
  - Window / [The tab and menu bars](#)
  - Help / [The tab and menu bars](#)
- menu bar, UMG Editor / [The tab and menu bar](#)
- menu bar, Unreal Matinee user interface
  - actions and functions / [The tab and menu bar](#)
- menu bar, User Interface
  - File / [The tab bar and the menu bar](#)
  - Edit / [The tab bar and the menu bar](#)
  - Editor Window / [The tab bar and the menu bar](#)
  - Help / [The tab bar and the menu bar](#)
- Mesh
  - adding to Character Blueprint / [Adding and configuring Mesh to a Character Blueprint](#)
  - configuring to Character Blueprint / [Adding and configuring Mesh to a Character Blueprint](#)
- mesher / [Introduction](#)
- Metal Shading Language / [Shaders](#)
- mirror material
  - creating / [Creating a mirror material](#) , [How to do it...](#)
- mobile optimization / [Mobile performance and optimization](#)
- mobile performance / [Mobile performance and optimization](#)
- mobility, lights
  - static / [Mobility](#)
  - stationary / [Mobility](#)
  - movable / [Mobility](#)
- Modes window, User Interface
  - about / [Modes](#)
  - PlaceMode / [Modes](#)
  - Paint Mode / [Modes](#)
  - Landscape Mode / [Modes](#)
  - Foliage Mode / [Modes](#)
  - Geometry Editing Mode / [Modes](#)
- modules
  - about / [Modules](#)
  - reference link, for documentation / [Modules](#)
- monetization methods
  - about / [Monetization methods](#)
- monetization models
  - Freemium model / [Monetization methods](#)
  - in-app advertisement / [Monetization methods](#)
  - paid apps / [Monetization methods](#)
  - In-App purchases / [Monetization methods](#)
- motion capture
  - about / [What is animation?](#)
- Movable Light
  - about / [Static, stationary, or movable lights](#) , [Movable Light](#)
- moveable lights
  - adding / [Adding moveable lights – flashlight, part 1](#)
  - reference / [How to do it...](#)
- Movement track / [Room 1](#)
- Move To node
  - Wait task, replacing with / [Replacing the Wait task with Move To](#)
- multitexturing
  - about / [Multitexturing](#)
- music
  - about / [Sound and music](#)
  - producing, for games / [How do we produce sound and music for games?](#)
- My Blueprint panel, Level Blueprint user interface / [My Blueprint panel](#)

## N

- Navigation Mesh
  - about / [Navigation Mesh](#)
  - creating / [Tutorial – creating a Navigation Mesh](#)
- Nav Mesh Bounds Volume
  - about / [Nav Mesh Bounds Volume](#) , [Nav Mesh Bounds Volume](#)
  - placing / [Room 3](#) , [Room 4](#)
- networking
  - about / [Networking 101 – creating collectables with networking](#)
  - collectables, collecting with / [Networking 101 – creating collectables with networking](#) , [Getting ready](#) , [How to do it...](#)
- New Landscape properties
  - reference / [How to do it...](#)
- News panel, Engine Launcher / [News](#)
- nodes
  - adding, in EventGraph / [Nodes to add in EventGraph](#)
  - Event Nodes / [How Blueprint works](#)
  - Function Nodes / [How Blueprint works](#)
  - Variable Nodes / [How Blueprint works](#)
  - Reference Nodes / [How Blueprint works](#)
- non-player characters (NPCs) / [The history of Unreal Engine](#)
- normal maps
  - using, with materials / [Using Textures and normal maps with Materials](#) , [How to do it...](#)
  - reference / [How to do it...](#)
- Normal Maps
  - about / [A special form of texture maps – Normal Maps](#)
  - reference link / [A special form of texture maps – Normal Maps](#)

## O

- objects
  - selecting / [Useful tip – selecting an object easily](#)
  - positioning, in level / [Useful tip – positioning objects in a level](#)
  - rotating, in level / [Useful tip – rotating objects in a level](#)
- OnComponentBeginOverlap function

- reference / [How to do it...](#)
- OpenGL
  - about / [APIs – DirectX and OpenGL](#)
- OpenGL Shading Language (GLSL) / [Shaders](#)
- opening
  - creating, for door / [Creating an opening for a door](#)
- opening cutscene
  - creating / [Creating an opening cutscene](#), [How to do it...](#)
- output-merger stage / [Shaders](#)

## P

- package
  - building, for project / [Building a package](#)
- paid apps / [Monetization methods](#)
- Pain Causing Volume
  - about / [Pain Causing Volume](#)
- Palette panel, UMG Editor / [The Palette panel](#)
- panning
  - about / [Pan](#)
- particle effect / [What goes into a game?](#)
- particle system
  - about / [What is a particle system?](#)
  - exploring / [Exploring an existing particle system](#)
  - components / [The main components of a particle system](#)
  - design principles / [The design principles of a particle system](#)
  - placing / [Placing a particle system](#), [How to do it...](#)
- pedestals
  - materials, creating for / [Pedestals](#)
- performance capture
  - about / [What is animation?](#)
- Persona
  - about / [What can you do with Persona?](#)
  - existing animation, assigning to / [Tutorial – assigning existing animation to a Pawn](#)
- Philips
  - reference link, for downloading IES Light Profiles / [Downloading IES Light Profiles](#)
- Photoshop / [Introduction](#)
- Physical Based Shading Model (PBSP)
  - about / [Physical Based Shading Model](#)
- Physics Asset Tool (PhAT) / [The Persona skeletal mesh animation](#)
- Physics Volume
  - about / [Physics Volume](#)
  - Pain Causing Volume / [Pain Causing Volume](#)
  - Kill Z Volume / [Kill Z Volume](#)
  - Level Streaming Volume / [Level Streaming Volume](#), [Cull Distance Volume](#)
  - Cull Distance Volume / [Cull Distance Volume](#)
  - Audio Volume / [Audio Volume](#)
  - PostProcess Volume / [PostProcess Volume](#)
  - Lightmass Importance Volume / [Lightmass Importance Volume](#)
- PickTargetLocation custom task
  - using, in Behavior Tree / [Using the PickTargetLocation custom task in BT](#)
- pipeline state object (PSO) / [Pipeline state representation](#)
- pixel-shader stage / [Shaders](#)
- PlaceMode, classes
  - about / [Modes](#)
  - Basic / [Modes](#)
  - Lights / [Modes](#)
  - Visual / [Modes](#)
  - BSP / [Modes](#)
  - Volumes / [Modes](#)
- player
  - preventing, from moving via cinematic mode / [Preventing a player from moving via cinematic mode](#), [How to do it...](#), [See also](#)
- Player Start
  - adding / [Adding Player Start](#)
- Player Start actor
  - placing / [Placing the Player Start actor](#)
- Play Store
  - game, uploading on / [Uploading the game on the Play Store](#)
- point light
  - about / [Lighting](#)
- Point Light
  - configuring / [Configuring a Point Light with more settings](#)
  - Attenuation Radius setting / [Attenuation Radius](#)
  - Intensity setting / [Intensity](#)
  - Use Inverse Squared Falloff setting / [Use Inverse Squared Falloff](#)
  - Color setting / [Color](#)
- Point Lights
  - about / [How to do it...](#)
  - reference / [How to do it...](#)
- PostProcess Volume
  - about / [PostProcess Volume](#)
- preconfigured levels
  - exploring / [Exploring preconfigured levels](#)
- Prefab / [Details](#)
- project
  - creating / [Creating a new project](#), [Creating a new project](#)
  - packaging / [Packaging your project](#), [How to do it...](#)
  - about / [Projects](#)
  - existing project, opening / [Opening an existing project](#)
  - directory structure / [Project directory structure](#)
- project, packaging
  - about / [Packaging the project](#)
  - Maps & Modes settings / [The Maps](#)
  - Packaging settings / [The Packaging settings](#)
  - Android app settings / [The Android app settings](#)
- props
  - adding, to room / [Adding props or a static mesh to the room](#)
- Pulse Code Modulation (PCM)

- about / [How are sounds recorded?](#)

## R

- rasterizer stage / [Shaders](#)
- Reference Nodes / [How Blueprint works](#)
- rendering
  - reference link / [Shaders](#)
- rendering pipeline
  - about / [Rendering pipeline](#)
- rendering system
  - about / [The rendering system](#)
- replication
  - reference / [How to do it...](#)
- Reports panel, Developer Console / [REPORTS](#)
- rig / [Preparing before animation](#)
- rigging / [Preparing before animation](#)
- rivers
  - creating, with Flatten tool / [Creating rivers with the Flatten tool](#), [How to do it...](#)
- room
  - sealing / [Sealing a room](#)
  - static mesh, adding to / [Adding props or a static mesh to the room](#)
  - props, adding to / [Adding props or a static mesh to the room](#)
  - finishing touches, applying to / [Applying finishing touches to a room](#)
  - building / [Building a room](#), [How to do it...](#)
- rooms
  - blocking out, with BSP brushes / [Blocking out the rooms with BSP brushes](#), [The first room](#), [The second room](#), [The third room](#), [The fourth room](#)

## S

- Scalable
  - about / [The Unreal Project Browser](#)
- Sci-Fi Hallway project
  - about / [Projects](#)
- Sculpt mode
  - for building exterior level / [Building an exterior level using the Sculpt mode](#), [How to do it...](#)
- Sculpt mode tools
  - reference / [How to do it...](#)
- Settings section, Developer Console / [SETTINGS](#)
- shaders
  - about / [Shaders](#)
- Shading Models
  - reference / [How to do it...](#)
- Shadow Map
  - about / [Common light/shadow definitions](#)
- shot plan
  - about / [Shot plan](#)
- shots
  - about / [Shot types](#)
  - Extreme Wide Shot (EWS) / [Shot types](#)
  - Extreme Long Shot (ELS) / [Shot types](#)
  - Wide Shot (WS) / [Shot types](#)
  - Long Shot (LS) / [Shot types](#)
  - Medium Shot (MS) / [Shot types](#)
  - Close Up Shot (CU) / [Shot types](#)
  - Extreme Close Up Shot (ECU) / [Shot types](#)
- Signal to Quantization Noise Ratio (SQNR)
  - about / [Bit depth](#)
- simple Behavior Tree
  - creating / [Example – creating a simple Behavior Tree](#)
  - creating, Wait task used / [Creating a simple BT using a Wait task](#)
- simple custom material
  - creating / [Creating a simple custom material](#)
- simple healthbar
  - creating / [Creating a Health/Damage system, part 2 – creating a healthbar](#), [Getting ready](#), [How to do it...](#)
- simple matinee sequence
  - creating / [Exercise – creating a simple matinee sequence](#)
- simple texture
  - used, for creating custom materials / [Creating custom material using simple textures](#)
- sky
  - adding, to level / [Adding the sky to a level](#)
- Sky Distance Threshold / [Lighting](#)
- Sky Light
  - adding / [Example – adding and configuring a Sky light](#)
  - configuring / [Example – adding and configuring a Sky light](#)
  - about / [How to do it...](#)
  - reference / [How to do it...](#)
- Sky light / [Lighting](#)
- Slate
  - reference / [Introduction](#)
- snap grids
  - using / [Useful tip – using the drag snap grid](#)
- social icons, Epic
  - Instagram / [UE4 Links](#)
  - Facebook / [UE4 Links](#)
  - YouTube / [UE4 Links](#)
  - Twitter / [UE4 Links](#)
  - Twitch Stream / [UE4 Links](#)
- solo emitters
  - observing, of fireplace particle system / [Observing the solo emitters of the system](#)
- sound
  - about / [Sound and music](#)
  - producing, for games / [How do we produce sound and music for games?](#)
  - recording / [How are sounds recorded?](#)
  - importing, into Unreal Editor / [Exercise – importing a sound into the Unreal Editor](#)
- Sound Cue Editor
  - about / [Sound Cue Editor](#), [The Sound Cue Editor](#)
  - opening / [How to open the Sound Cue Editor](#)
  - configuring / [Configuring the Sound Cue Editor](#)

- Sound Cues
  - about / [Sound Cue Editor](#)
- sound cues
  - about / [Unreal sound formats and terminologies](#)
- sound waves
  - about / [Unreal sound formats and terminologies](#)
- SpeedTree
  - reference / [How to do it...](#)
- sphere brush / [Default BSP brush shapes](#)
- Spiral Stair Brush / [Default BSP brush shapes](#)
- Spot Light
  - adding / [Adding and configuring a Spot Light](#)
  - configuring / [Adding and configuring a Spot Light](#)
  - inner cone angle / [Inner cone and outer cone angle](#)
  - outer cone angle / [Inner cone and outer cone angle](#)
- spot light
  - about / [Lighting](#)
- Spot Lights
  - reference / [How to do it...](#)
  - about / [How to do it...](#)
- start level
  - map, configuring as / [Configuring a map as a start level](#)
- static emissive lighting (glow)
  - creating / [Creating glowing materials with static emissive lighting](#), [How to do it...](#)
- Static Light
  - about / [Static, stationary, or movable lights](#), [Static Light](#)
- static mesh
  - adding to room / [Adding props or a static mesh to the room](#)
- Static Mesh
  - about / [Static Mesh](#)
  - versus BSP Brush / [BSP Brush versus Static Mesh](#)
  - movable, making / [Making Static Mesh movable](#), [Materials](#)
  - reference link / [Static Mesh creation pipeline](#)
- Static Mesh creation pipeline
  - about / [Static Mesh creation pipeline](#)
- Static Mesh Editor
  - about / [How to do it...](#)
  - reference / [How to do it...](#)
- static meshes
  - placing / [Placing static meshes](#), [Getting ready](#), [How to do it...](#)
  - life, adding to / [Adding life to static meshes](#), [How to do it...](#)
- Stationary Light
  - about / [Static, stationary, or movable lights](#), [Stationary Light](#)
- stop-motion animation
  - about / [What is animation?](#)
- stream-output stage / [Shaders](#)
- subtractive brush / [How to do it...](#)
- survey-accurate visual simulations
  - reference link / [Field of view](#)

## T

- tab bar, UMG Editor / [The tab and menu bar](#)
- Target Point actors
  - adding from All Classes section / [Target Point](#), [Room 3](#), [Room 4](#)
- terrain manipulation
  - about / [Introducing terrain manipulation](#)
- tessellator stage / [Shaders](#)
- text
  - displaying during runtime / [Displaying text during runtime](#), [How to do it...](#)
- texture
  - about / [Materials](#)
- Texture
  - versus Material / [Materials versus Textures](#)
- Texture Map
  - creating / [How to create and use a Texture Map](#)
  - using / [How to create and use a Texture Map](#)
- Texture Mapping
  - about / [Texture/UV mapping](#)
- textures
  - using with materials / [Using Textures and normal maps with Materials](#), [How to do it...](#)
- toolbar, Level Blueprint user interface
  - about / [The toolbar](#)
  - Compile / [The toolbar](#)
  - Search / [The toolbar](#)
  - Class Settings / [The toolbar](#)
  - Class Defaults / [The toolbar](#)
  - Play / [The toolbar](#)
- toolbar, UMG Editor / [The toolbar](#)
- toolbar, Unreal Matinee user interface
  - about / [The toolbar](#)
  - actions / [The toolbar](#)
- tool bar, User Interface
  - Save / [The toolbar](#)
  - Source Control / [The toolbar](#)
  - Content button / [The toolbar](#)
  - Marketplace / [The toolbar](#)
  - Settings / [The toolbar](#)
  - Blueprints / [The toolbar](#)
  - Matinee / [The toolbar](#)
  - Build / [The toolbar](#)
  - Play button / [The toolbar](#)
  - Pause button / [The toolbar](#)
  - Stop button / [The toolbar](#)
  - Eject button / [The toolbar](#)
  - Launch button / [The toolbar](#)
- Tracks panel, Unreal Matinee user interface
  - about / [The Tracks panel](#)

- Transparency
  - reference / [How to do it...](#)
- trees and rocks
  - placing, with Foliage tool / [Getting ready](#), [How to do it...](#)
- trigger / [How to do it...](#)
- triggers
  - adding to rooms / [Adding triggers](#), [Room 1](#), [Room 2](#), [Room 4](#)
- Trigger Volume
  - about / [Trigger Volume](#)
  - used, for turning on light / [Using the Trigger Volume to turn on/off light](#)
  - used, for turning off light / [Using the Trigger Volume to turn on/off light](#)
  - used, for toggling light on/off / [Using Trigger Volume to toggle light on/off \(optional\)](#)
- Trigger Volumes
  - for animating door / [Using Trigger Volumes – opening a door using Matinee](#), [How to do it...](#)
- tutorial
  - about / [Camera](#)
- tween / [How to do it...](#)
- Twitch Stream / [UE4 Links](#)
- types of lights
  - Point Lights / [How to do it...](#)
  - Spot Lights / [How to do it...](#)
  - Directional Lights / [How to do it...](#)
  - Sky Light / [How to do it...](#)

## U

- UE4
  - about / [Introduction](#)
  - installing / [Installing UE4 and folder structure](#), [How to do it...](#), [Downloading and installing UE4](#)
  - folder structure / [Installing UE4 and folder structure](#)
  - system requirements / [System requirements](#)
  - downloading / [Downloading and installing UE4](#)
- UE4 homepage
  - URL / [Downloading and installing UE4](#)
- UE4 Launcher
  - Learn section / [Learn](#)
  - AnswerHub section / [AnswerHub](#)
  - forums / [Forums](#)
- UE4 Links, Engine Launcher
  - about / [UE4 Links](#)
  - forums / [UE4 Links](#)
  - AnswerHub / [UE4 Links](#)
  - Roadmap / [UE4 Links](#)
- UI overview / [UI overview](#), [How to do it...](#)
- UMG Editor
  - used, for adding main menu / [Adding the main menu using Unreal Motion Graphics](#)
  - about / [UMG Editor](#)
  - tab bar / [The tab and menu bar](#)
  - menu bar / [The tab and menu bar](#)
  - toolbar / [The toolbar](#)
  - Graph Editor / [The Graph Editor](#)
  - Details panel / [The Details panel](#)
  - Palette panel / [The Palette panel](#)
  - Hierarchy panel / [The Hierarchy panel](#)
  - Animations panel / [The Animations panel](#)
- Unlit mode / [How to do it...](#)
- Unreal
  - Material, creating in / [Creating a Material in Unreal](#)
  - URL, for official site / [Downloading and installing UE4](#)
- Unreal audio system
  - about / [The Unreal audio system](#)
- Unreal Development Kit (UDK) / [The Unreal Matinee user interface](#)
- Unreal Editor
  - about / [Unreal Editor](#)
  - beginners guide / [A beginner's guide to the Unreal Editor](#)
  - sound, importing into / [Exercise – importing a sound into the Unreal Editor](#)
- Unreal Engine
  - history / [The history of Unreal Engine](#)
  - editors / [Unreal Engine and its powerful editors](#)
  - Unreal Editor / [Unreal Editor](#)
  - Material Editor / [Material Editor](#)
  - Sound Cue Editor / [Sound Cue Editor](#)
  - Matinee Editor / [Matinee Editor](#)
  - Blueprint visual scripting system / [The Blueprint visual scripting system](#)
- Unreal Engine 4
  - components / [The components of Unreal Engine 4](#)
  - sound engine / [The sound engine](#)
  - physics engine / [The physics engine](#)
  - graphics engine / [The graphics engine](#)
  - Gameplay framework / [Input and the Gameplay framework](#)
  - input system / [Input and the Gameplay framework](#)
  - light / [Light and shadow](#)
  - shadow / [Light and shadow](#)
  - post-process effects / [Post-process effects](#)
  - artificial intelligence (AI) / [Artificial intelligence](#)
  - online and multiplatform capabilities / [Online and multiplatform capabilities](#)
  - Behavior Tree, implementing in / [How to implement a Behavior Tree in Unreal Engine 4](#)
  - about / [Introduction](#)
- Unreal Engine Editor
  - IES Profiles, importing into / [Importing IES Profiles into the Unreal Engine Editor](#)
- Unreal Landscaping tool
  - about / [Introducing terrain manipulation](#)
- Unreal Matinee
  - about / [What is Unreal Matinee?](#)
  - Matinee actors, adding / [Adding Matinee actors](#)
  - door, animating / [Animating the door](#)
- Unreal Matinee Editor
  - about / [Getting familiar with the Unreal Matinee Editor](#)

- Unreal Matinee user interface
  - about / [The Unreal Matinee user interface](#)
  - tab bar / [The tab and menu bar](#)
  - menu bar / [The tab and menu bar](#)
  - toolbar / [The toolbar](#)
  - Curve Editor / [The Curve Editor](#)
  - Tracks panel / [The Tracks panel](#)
  - Details panel / [The Details panel](#)
- Unreal Motion Graphics (UMG) UI Designer / [Introduction](#)
- Unreal objects
  - about / [Unreal objects](#)
- Unreal programming
  - about / [Unreal programming](#)
- Unreal Project Browser
  - about / [The Unreal Project Browser](#)
- User Interface
  - about / [The user interface](#)
  - tab bar / [The tab bar and the menu bar](#)
  - menu bar / [The tab bar and the menu bar](#)
  - tool bar / [The toolbar](#)
  - Viewport / [Viewport](#)
  - Modes window / [Modes](#)
  - World Outliner / [World Outliner](#)
  - Content Browser / [Content Browser](#)
  - Details panel / [Details](#)
- user interface (UI) / [Introduction](#)
- UV texture map
  - about / [Texture/UV mapping](#)

## V

- variable
  - adding into BlackBoardData / [Adding a variable into BlackBoardData](#)
- Variable Nodes / [How Blueprint works](#)
- vector / [How to do it...](#)
- vertex-shader stage / [Shaders](#)
- Video Games Live
  - reference link / [How do we produce sound and music for games?](#)
- View Mode
  - changing / [Useful tip – changing View Mode to aid visuals](#)
- viewport
  - navigating / [Navigating the viewport](#), [Navigating the viewport](#), [How to do it...](#)
- Viewport, User Interface
  - about / [Viewport](#)
  - Viewport Options / [Viewport](#)
  - Viewport Type / [Viewport](#)
  - View Mode / [Viewport](#)
  - Show / [Viewport](#)
  - Transform Tools / [Viewport](#)
  - Coordinate System / [Viewport](#)
  - Surface Snapping / [Viewport](#)
  - Grid Snapping / [Viewport](#)
  - Grid Snap Value / [Viewport](#)
  - Rotation Grid Snapping / [Viewport](#)
  - Rotation Grid Snap Value / [Viewport](#)
  - Scale Grid Snapping / [Viewport](#)
  - Scale Grid Snap Value / [Viewport](#)
  - Camera Speed / [Viewport](#)
  - Maximize or Restore Viewport / [Viewport](#)
- views
  - about / [Views](#)
- Visual Effect actors
  - adding to game / [Adding Visual Effect actors to the game – Post Process Volume](#)
- Visual Effects class
  - about / [Visual Effects](#)
  - actors / [Visual Effects](#)
  - Post Process Volume actor / [Visual Effects](#)
  - Atmospheric Fog actor / [Visual Effects](#)
  - Exponential Height Fog actor / [Visual Effects](#)
  - Sphere Reflection actor / [Visual Effects](#)
  - Box Reflection Capture actor / [Visual Effects](#)
  - Deferred Decal actor / [Visual Effects](#)
- Vivaldi
  - URL / [Exercise – importing a sound into the Unreal Editor](#)
- volumes
  - about / [Introducing volumes](#), [Volumes](#)
  - Blocking Volume / [Blocking Volume](#), [Volumes](#)
  - Camera Blocking Volume / [Camera Blocking Volume](#), [Volumes](#)
  - Trigger Volume / [Trigger Volume](#), [Volumes](#)
  - Nav Mesh Bounds Volume / [Nav Mesh Bounds Volume](#), [Volumes](#)
  - Physics Volume / [Physics Volume](#), [Volumes](#)
  - types / [Volumes](#)
  - Audio Volume / [Volumes](#)
  - Cull Distance Volume / [Volumes](#)
  - Kill ZVolume / [Volumes](#)
  - Lightness Character Indirect Detail Volume / [Volumes](#)
  - Lightmass Importance Volume / [Volumes](#)
  - Nav Modifier Volume / [Volumes](#)
  - Pain Causing Volume / [Volumes](#)
  - Post Process Volume / [Volumes](#)
  - Precomputed Visibility Override Volume / [Volumes](#)
  - Precomputed Visibility Volume / [Volumes](#)
- Vorbis
  - URL / [Exercise – importing a sound into the Unreal Editor](#)
- Vulkan / [Shaders](#)

## W

- Wait task

- used, for creating simple Behavior Tree / [Creating a simple BT using a Wait task](#)
- replacing, with Move To node / [Replacing the Wait task with Move To](#)
- wall
  - adding / [Adding a wall](#)
  - duplicating / [Duplicating a wall](#)
  - materials, adding to / [Adding materials to the walls](#)
- Wids Shot (WS) / [Shot types](#)
- Widget Blueprint
  - about / [How to do it...](#)
  - reference / [How to do it...](#)
- Windows directory structure
  - about / [The Windows directory structure](#)
  - DirectXRedist / [Windows DirectXRedist](#)
  - Launcher folder / [Launcher](#)
  - 4.X folders / [4.X folders](#)
- World Outliner , User Interface
  - Create Folders / [World Outliner](#)
- World Outliner, User Interface
  - about / [World Outliner](#)
  - Hide Function / [World Outliner](#)
  - Attach Actors / [World Outliner](#)

## X

- 4.X folders, Windows directory structure
  - about / [4.X folders](#)
  - Engine / [4.X folders](#)
  - Samples / [4.X folders](#)
  - Templates / [4.X folders](#)

## Y

- YouTube, UE4
  - URL / [UE4 Links](#)