Community Experience Distilled

Learning Unreal Engine Android Game Development

Tap into the power of Unreal Engine 4 and create exciting games for the Android platform

Nitish Misra

Learning Unreal Engine Android Game Development

Tap into the power of Unreal Engine 4 and create exciting games for the Android platform

Nitish Misra

BIRMINGHAM - MUMBAI

Learning Unreal Engine Android Game Development

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book 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 book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author 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 book.

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

First published: June 2015

Production reference: 1250615

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78439-436-3

www.packtpub.com

Credits

Author
Nitish Misra

Reviewers
David Pol
Steve Santello
Nicola Valcasara

Commissioning Editor
Edward Bowkett

Acquisition Editors
Ruchita Bhansali
Rebecca Youé

Content Development Editor
Shweta Pant

Technical Editor
Namrata Patil

Copy Editors
Puja Lalwani
Merilyn Pereira

Project Coordinator
Sanjeet Rao

Proofreader
Safis Editing

Indexer
Tejal Soni

Production Coordinator
Melwyn D'sa

Cover Work
Melwyn D'sa

About the Author

Nitish Misra wanted a career in the game industry ever since he was a child. In late 2014, he was one step closer to realizing that dream after he completed his BA (Hons) in game design and production management from Abertay University. He is currently working in a start-up based in New Delhi, designing games on mobiles and working toward breaking into the game industry.

His areas of interest include concept development, systems design, and level design. With 3 plus years of experience in Unreal Development Kit, Unreal Engine 4 was the most logical go-to option for him, and he has been using it for the past 1 year, experimenting with the various features offered by UE4, to see what can be achieved using it.

First of all, I would like to thank my parents for supporting me in writing this book. Juggling both work and the book was no easy task, and their support really motivated me. I would like to thank my friends who helped me while writing the book.

I would like to thank my tutors from the university. It was their guidance that gave me the confidence and knowledge to write this book.

Finally, I would like to thank Packt Publishing for giving me the opportunity to author this guide. Writing this book was a great experience.

About the Reviewers

David Pol is a seasoned game programmer who started working in the industry in 2008. What began as a hobby, when he was a little kid programming games in his bedroom, became a full-time job even before he graduated with a master's degree in computer science. While working with several game studios, he has shipped over a dozen games on many platforms.

He enjoys sharing his knowledge with the community whenever possible. He is currently working as a freelance programmer, helping studios and individuals realize their digital dreams.

You can follow David at `http://www.davidpol.com`.

Steve Santello is a well-seasoned educator and veteran of the game industry from Chicago, IL, USA. His cross-disciplinary study in art, design, programming, and project management has allowed him to explore every facet of game development. As an educator, he has used his passion to teach others about game development, which has populated the game industry with many talented developers over the years. Steve has worked on over 15 different titles as a 2D and 3D artist for the Chicago game developer, Babaroga. With Babaroga, he worked as one of four artists on many mobile titles published by EA, such as *Spore Origins*, *The Godfather Game*, and *Pictionary: The Game of Quick Draw*. He has also worked on many mobile titles published by Disney Interactive, such as *Hannah Montana In Action* and *Meet the Robinsons*. He and his development team have developed a number of original IPs, such as Babaroga Eats Children and BEES!

Steve has been a university professor since 2006. He helped pioneer the Digital Entertainment and Game Design program at ITT Technical Institute in St. Rose, LA, USA, as well as the Game and Simulation Programming program at DeVry University in Addison, IL, USA. He has served as an adjunct professor at the Illinois Institute of Art, Chicago, where he taught game prototyping as a team manager and acting producer. On the side, Steve develops game and simulation prototypes and is planning to release his first two independent games, which are being developed with Unreal Engine 4, sometime in 2015. Steve has made contributions to *Game Development Essentials: Game Interface Design, 2nd Edition*. In it, he wrote about the past, present, and future of user interface in games, which included breaking down the HUD in games, such as *Deadspace*, and talked about how he and his team designed the user interface in Spore Origins.

Steve is currently working toward tenure as a CIS Gaming Instructor at the College of DuPage in Glen Ellyn, IL, USA. Although he is very proud of his success, he knows that all of his combined experiences played a major role to where he is today.

I would like to thank the Illinois Institute of Art in Chicago, where I earned a bachelor of fine arts degree in game art and design, and the talented professors who helped me break into the industry as an artist. I would also like to thank DePaul University, where I earned a master's degree in computer science and the professors who helped him become a great programmer and independent developer. Last but not least, I would like to thank Babaroga for immersing me in the world of game development and giving me the strong work ethic that I have today.

Nicola Valcasara is a freelance game developer and cofounder of Deuxality Games Ltd. He is an expert programmer, specializing in mobile development, with a strong passion for games and technology. He started working in the game industry in 2012, after winning the first prize at the Microsoft *Rapid2D* competition for young developers.

He works at Deuxality Games Ltd., an indie game company that started in 2014 with the dream of becoming the new mobile market success story. Having developed and published five products in less than 8 months, DeuXality is now starting to confirm its place in the industry, participating actively at the 2015 GDC in San Francisco.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface vii

Chapter 1: Getting Started with Unreal 4 1

- **What to expect** 1
- **System requirements** 2
- **Downloading and installing UE4** 3
  - The Windows directory structure 7
    - Windows DirectXRedist 8
    - Launcher 8
    - 4.X folders 8
- **The Engine Launcher** 9
  - News 12
  - Learn 12
  - Marketplace 13
  - Library 14
  - UE4 Links 17
- **Summary** 19

Chapter 2: Launching Unreal Engine 4 21

- **Meet the Editor** 21
- **The Unreal Project Browser** 22
- **The user interface** 26
  - The tab bar and the menu bar 27
  - The toolbar 29
  - Viewport 31
  - Modes 35
  - World Outliner 37
  - Content Browser 39
  - Details 40
- **Hotkeys and controls** 41
- **Summary** 43

Chapter 3: Building the Game – First Steps **45**

Projects **45**
Creating a new project 46
Opening an existing project 47
Project directory structure 47
Bloques **48**
Concept 48
Controls 48
Creating the project for the game 48
BSP brushes **50**
Default BSP brush shapes 51
Editing BSP brushes 53
Blocking out the rooms with BSP brushes **54**
The first room 54
The second room 57
The third room 60
The fourth room 66
Content Browser **69**
Migrating and importing assets 70
Importing assets 71
Migrating assets 72
Placing actors **74**
Materials **77**
The Material Editor 78
The tab and menu bar 79
The toolbar 80
The Palette panel 81
The Stats panel 82
The Details panel 82
The Viewport panel 83
The Graph panel 84
Applying materials 85
Creating the materials 87
Pedestals 87
Doors 90
Key Cubes 91
Decorative assets 93
Lighting **95**
Mobility 99
Lighting up the environment 101
Summary **102**

Chapter 4: Using Actors, Classes, and Volumes 103
Basic classes 104
Adding basic class actors to the game 105
Placing the Player Start actor 105
Adding triggers 106
Visual Effects 111
Adding Visual Effect actors to the game – Post Process Volume 114
Volumes 116
Adding Volumes to the game 118
Lightmass Importance Volume 119
Nav Mesh Bounds Volume 120
All Classes 122
Adding actors from All Classes 124
Camera 124
Matinee actors 125
Target Point 125
Summary 127
Chapter 5: Scripting with Blueprints 129
How Blueprint works 130
The Level Blueprint user interface 134
The tab and menu bars 134
The toolbar 135
The Details panel 136
The Compiler Results panel 137
My Blueprint panel 137
The Event Graph 138
Using Level Blueprint in the game 139
Key cube pickup and placement 139
The Blueprint class 155
Creating a Blueprint class 155
Viewport 158
The Construction Script 159
The Event Graph 161
Scripting basic AI 165
Summary 172
Chapter 6: Using Unreal Matinee 173
What is Unreal Matinee? 173
Adding Matinee actors 174
The Unreal Matinee user interface 174
The tab and menu bar 175
The toolbar 176
The Curve Editor 177

The Tracks panel 180
The Details panel 182
Animating the door **183**
Room 1 183
Room 2 191
A bridge for the AI character 194
Summary **200**
Chapter 7: Finishing, Packaging, and Publishing the Game **201**
Adding the main menu using Unreal Motion Graphics **201**
UMG Editor **202**
The tab and menu bar 202
The toolbar 203
The Graph Editor 204
The Details panel 206
The Palette panel 206
The Hierarchy panel 207
The Animations panel 208
Creating the main menu **208**
Installing the Android SDK **216**
Setting up the Android device **219**
Packaging the project **221**
The Maps & Modes settings 222
The Packaging settings 223
The Android app settings 224
Building a package 226
Developer Console **229**
ALL APPLICATIONS 230
APK 231
Store Listing 232
Content Rating 233
Pricing & Distribution 234
In-app Products 234
Services & APIs 235
GAME SERVICES 236
Game details 236
Linked apps 237
Events 238
Achievements 239
Leaderboards 240
Testing 240
Publishing 241

REPORTS 242
SETTINGS 243
ALERTS 244
Publishing your game 244
Activating Google services 244
Preparing the project for shipping 247
Uploading the game on the Play Store 251
Monetization methods 252
Mobile performance and optimization 254
Summary 256
Appendix: What Next? 257
Learn 257
AnswerHub 265
Forums 267
Summary 268
Index 269

Preface

The first Unreal Engine was launched in 1998 by Epic Games. A great feature about this engine was that, because of UnrealScript, the engine became quite popular with the community, as it made modding quite easy and accessible. Then, in 2002, Epic released their next engine, Unreal Engine 2, which was a great improvement over the first engine. It came with something called UnrealEd 2 (and later, UnrealEd 3), which was essentially a level editor that you could use to create levels for Unreal. This, along with UnrealScript could be used to create entirely new games. The engine offered better rendering, better physics, and better collision than its predecessor. It also supported the then current generation consoles, namely PlayStation 2, Xbox, and GameCube. In 2006, Epic released their next, and, probably, their most popular and widely used engine, Unreal 3. It was a giant leap in terms of technology. This is where Unreal Engine started picking up steam. However, perhaps the most significant feature offered by Unreal Engine 3 was Kismet. Kismet is an extremely powerful visual scripting tool. The way it works is that there are various nodes, which can be connected to form a logical sequence, kind of like a flowchart. The best part about Kismet is that you do not require any programming knowledge. You can make an entire game without writing a single line of code using Kismet. It is an extremely handy tool for artists and designers, since they can make quick prototypes or experiment with a certain feature on their own and not have to rely on programmers.

We now come to a more present time. In February 2012, Epic unveiled Unreal 4, which was finally released on March 19, 2014. Unreal Engine 4 had been in the works since 2003. This Engine was a huge step up from the previous one. For one, it totally removed UnrealScript and replaced it with C++. In earlier versions of Unreal, if you wanted to modify the engine to develop your game, you had to do so using UnrealScript, which meant learning a new language. But now, if you wish to modify the Engine, you can do so with C++. This was a huge improvement for engine programmers, since this meant they could modify and tweak anything they wish using a language they already know and love.

Not only that, the engine's source code is also available for developers and can be downloaded from the GitHub repository. This means that developers have full control over the engine and can tweak virtually anything, including the physics, rendering, and UI.

It also offers something called the Hot Reload function. Normally, when you want to make changes in the code of a game, you have to stop the game, make the desired change, and then run it again to see how it affects the game. However, with the hot reload function, you can make changes in the game without having to stop or pause the game. Any change you make in the game's code is instantly updated and you can see its effect in real time.

You can also develop games for a wide variety of platforms on Unreal 4, including Xbox One, PlayStation 4 (including Project Morpheus), Windows PC, Linux, Mac OS X, HTML 5, iOS, and Android. It also offers support for Oculus Rift.

Another major change made by Epic is the licensing model, aimed at smaller, indie developers. To be more specific, to license Unreal Development Kit (UDK), which was the previous version of Unreal Engine, the developers had to pay a $99 licensing fee and 25 percent of the royalties made after the company had earned at least $50,000 from sales. In Unreal 4, however, they have changed the structure. As of early 2015, Unreal Engine 4 is now completely free to download and use. No licensing fee, no subscription, nothing. You can download it, develop a game on it, and publish it, without spending a single penny on the engine. You only pay 5 percent of the royalties after you have earned more than $3000 in revenue.

Another great feature that Unreal 4 provides is the Marketplace. The Marketplace is a great place to buy and upload assets. These assets can be Animations, 3D Models, Materials, Sound Effects, Premade Games, and so on. This is also good news for aspiring developers, who do not have the resources or manpower to develop these assets. They can simply buy the required assets from the Marketplace and use them in their game. Developers can also upload their own work onto the Marketplace and earn some money.

This book is aimed at beginners who have little to no knowledge regarding Unreal Engine 4, and it covers how to develop games for Android using it. This book will cover all that you need to know in order to get started. From the very basics, such as how to download and install the engine, to the more advanced, such as how to upload your finished product on to the Google Play Store, are going to be covered. We will do this by making a game of our own using UE4, step-by-step. Each chapter can be thought of as a step in making our game. The book has been structured in such a way that each subsequent chapter is a continuation of the previous chapter to give a sense of flow to the readers.

This book will also not use any other software (apart from the Android SDK), so that the readers do not have to first download dozens of software in order to get ready, and focus solely on the engine. All of the assets, such as materials, static meshes, and so on, will be the ones available in the Engine itself.

What this book covers

Chapter 1, *Getting Started with Unreal 4*, teaches you how to download and install Unreal Engine 4 and the launch client.

Chapter 2, *Launching Unreal Engine 4*, discusses what a project is, how to create a new project, what the Editor is, its UI, and some hotkeys you should be aware of.

Chapter 3, *Building the Game – First Steps*, 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.

Chapter 4, *Using Actors, Classes, and Volumes*, looks at the various types of classes offered in the Modes panel. This includes Basic Classes, Visual Classes, and Volumes, and how to implement them into our game.

Chapter 5, *Scripting with Blueprints*, looks at how to script in the game using Blueprint. Scripting is an important aspect of UE4, since a game without interactivity is not really a game. We will also look at the most commonly used types of Blueprint.

Chapter 6, *Using Unreal Matinee*, looks at another power tool offered by UE4, Matinee. We will use Matinee to create cutscenes and other animations essential for the game.

Chapter 7, *Finishing, Packaging, and Publishing the Game*, completes the only thing left after having a fully functional game. That is to finalize the game, package it, and publish it on the Google Play Store.

Appendix, *What Next?*, suggests where to go from here. It deals with various places where you can find documentation and tutorials to further develop your skills in UE4.

What you need for this book

For this book, you require:

- Unreal Engine 4
- The Android Development Kit

The main aim of the book was to use as little 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 book is for

This book is aimed at developers, who have little to no knowledge of how to use Unreal Engine 4, and wish to use it to develop games for Android.

Conventions

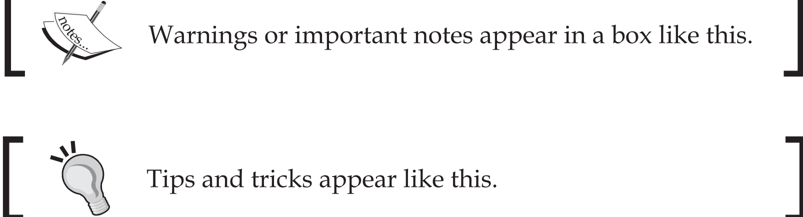
In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: " The next step is finalizing the game, packaging it into an `.apk` file, and publishing it to the Google Play Store."

Any command-line input or output is written as follows:

```
keytool -genkey -v -keystore -*name of your project*.keystore -alias
*alias_name* -keyalg RSA - keysize 2048 -validity 10000
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: " From the home page, click on the **Get Unreal** button on the right of the screen.".



Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via 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 on `www.packtpub.com/authors`.

Customer support

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

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/4363OT_ColorImages.pdf`

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 books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright 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` 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

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

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:

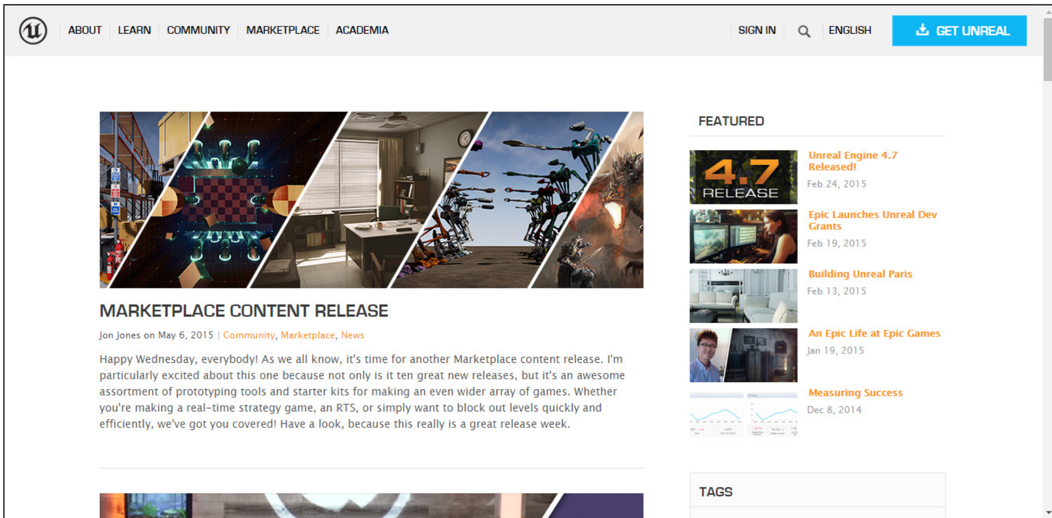
- Windows 7/8, 64-bit (Or Mac OS X 10.9.2 or later)
- .NET 4.0
- DirectX 10 (Mac: OpenGl 3.3)

- 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:

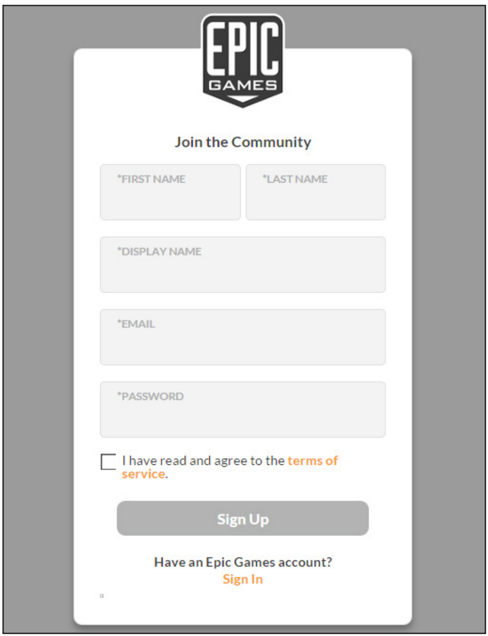


   Everything you need to know regarding UE4, you can find here—including the latest news, the latest version of the engine, blog updates, latest Marketplace entries, and so on. As of 2015, the engine has been made free to download.

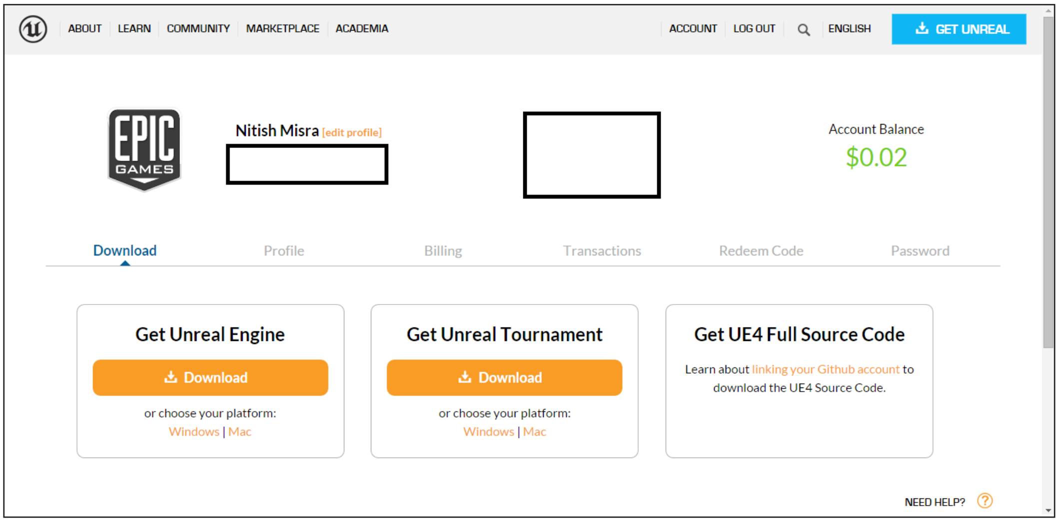
   In addition to the UE4 homepage, it is recommended that you visit `https://docs.unrealengine.com/latest/INT/`. It is full of documentation and video tutorials on how to use UE4. Epic boasts a large, active, and friendly community, always willing to help anyone facing a problem via the forums.

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/`.

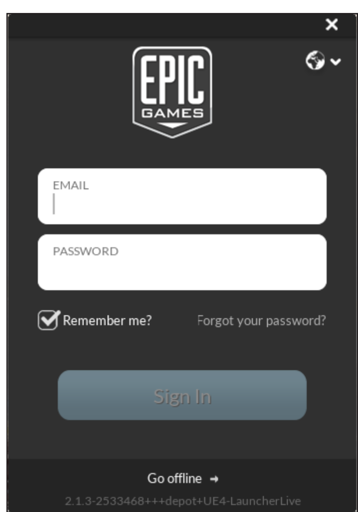
2. From the home page, click on the **GET UNREAL** button on the right of the screen. Clicking on it will bring you to the subscription page, shown here:



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.



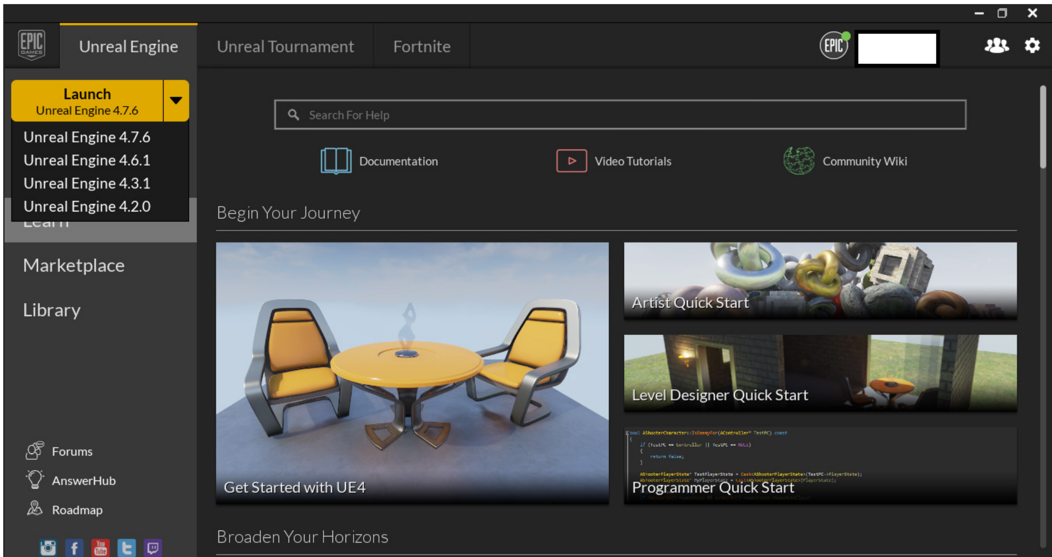
5. Now that you have your account set up, you can download UE4. You can download either the Windows version or the Mac version, depending upon your setup. To download, under **Latest Download**, click on the **Download** button and you will download the Engine Launcher.
6. To run the installer, simply 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.



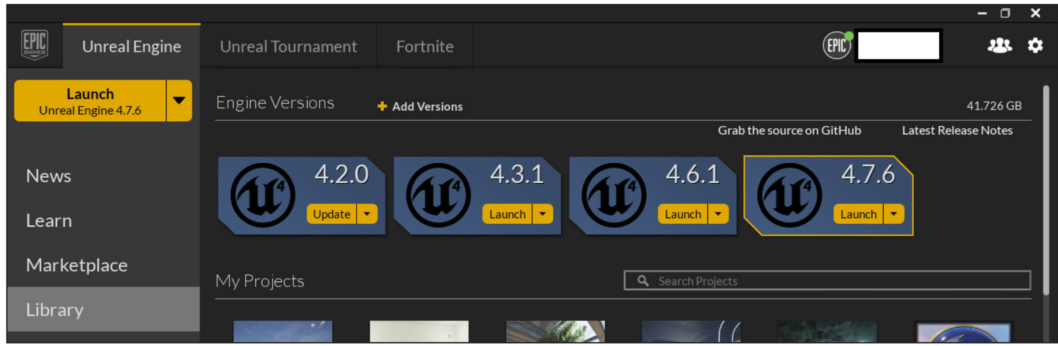
This is the login screen. Just type the e-mail address you used to subscribe and your password, then either click on the arrow button next to **PASSWORD** or hit the *Enter* key, to log in.

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.



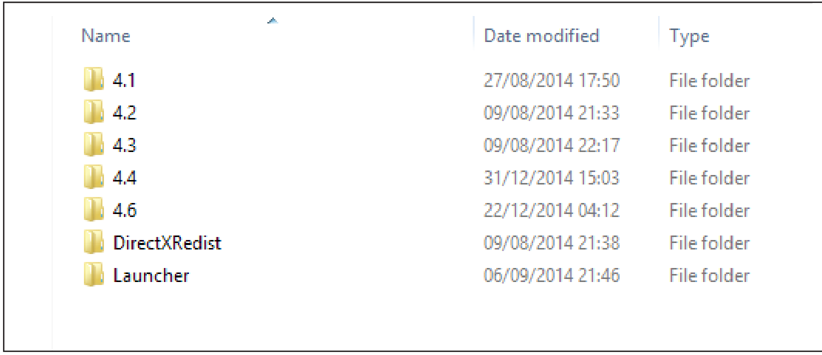
Alternatively, you can also click on the **Library** button, and select which engine to run from there. All of the versions installed on your system will be listed, and you can simply launch any version from the list by clicking on the **Launch** button.



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:



Each version of the Engine gets its very own folder. Apart from that, there are two other folders, namely `DirecXRedist` 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.

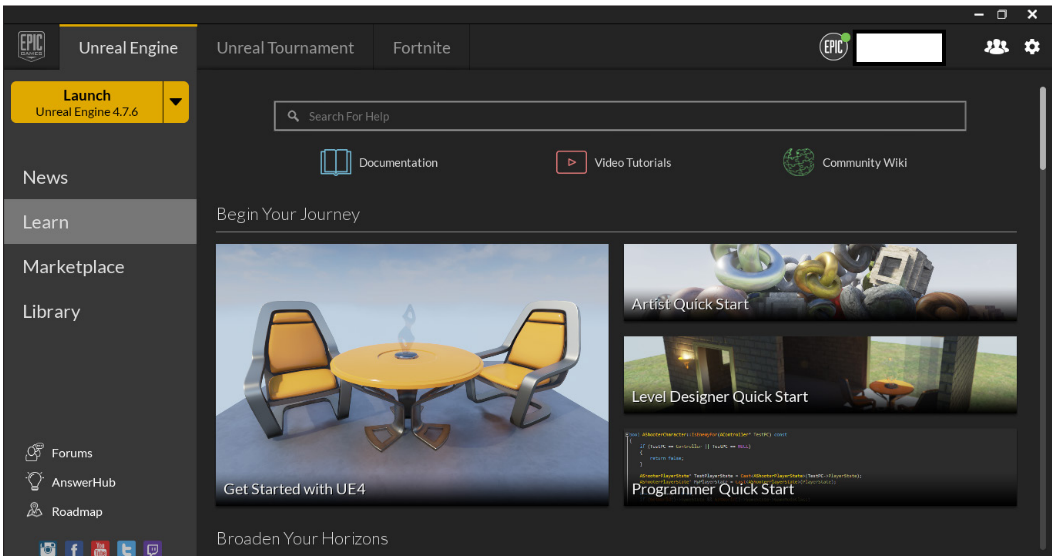
All of the `4.X` folders, although separate, contain the same set of subfolders, hence they are grouped together. The following are the subfolders:

- `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

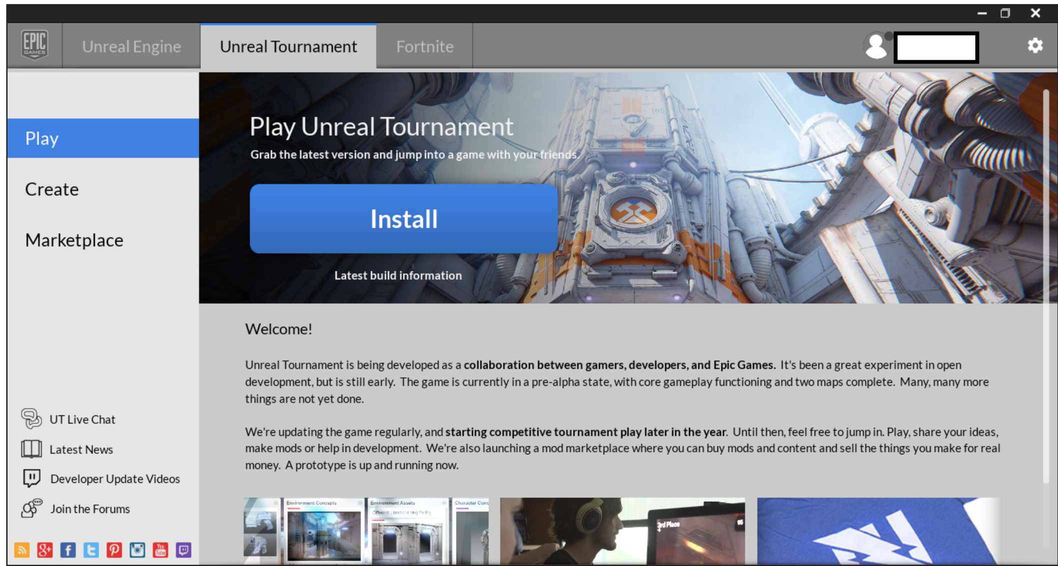
The Engine Launcher is a window that opens up after you run the engine. It is full of features and resources that can prove to be quite useful for you. Firstly, we will look at the Engine Launcher's user interface, its breakdown, where everything is located, its functionalities, and so on.

Upon opening the Engine Launcher, you will see the following window:



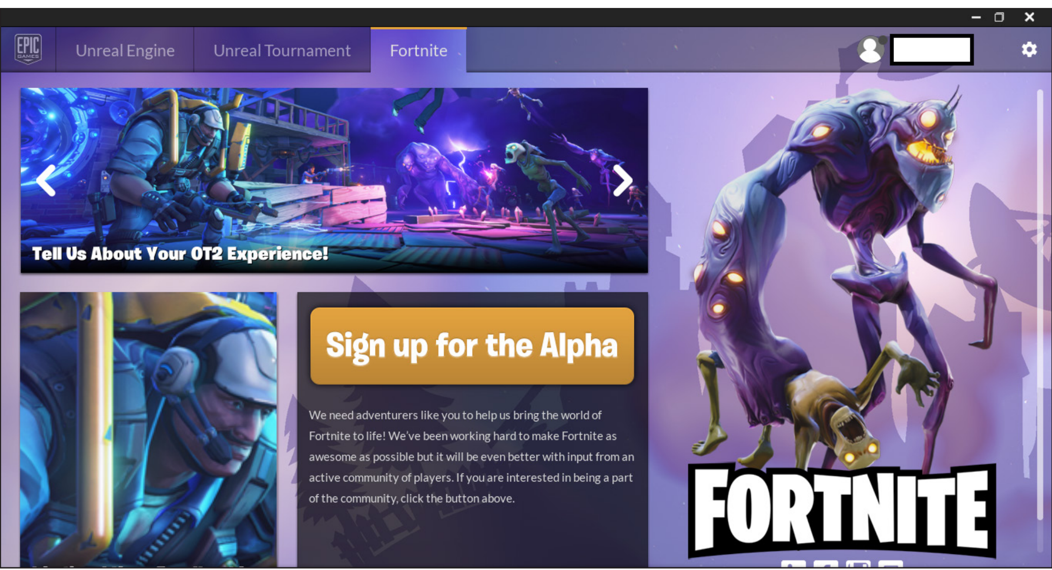
At the top left, there are three tabs, **Unreal Engine**, **Unreal Tournament**, and **Fortnite**. The Unreal Engine tab is open by default, and contains what you see in the preceding screenshot.

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.

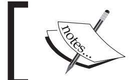
At the top-right corner, on the panel with the tabs, are two buttons, the **friend list** button, and the **Settings** button. When you click on the **friend list** button, it opens a window, where you can manage your friend list, like adding and removing friends, seeing who is online, and so on. You 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.



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.

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

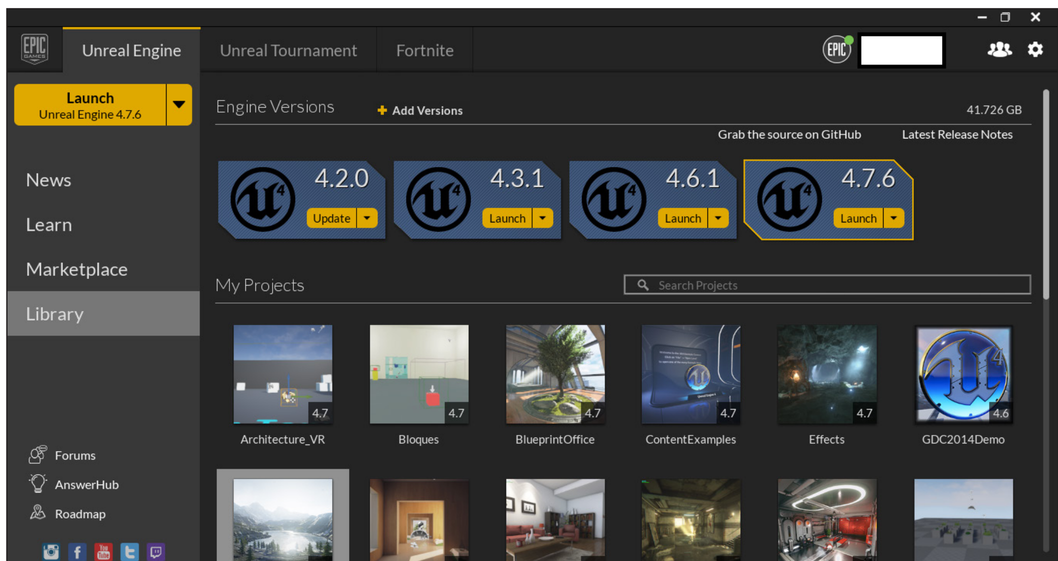
The Marketplace is where developers can purchase assets. Developers who lack the manpower or resources to create assets can purchase and use them in their game. These include meshes, materials, animation sets, rigged characters, audio files, sound effects, projects, and tutorials, to name a few. Certain 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.

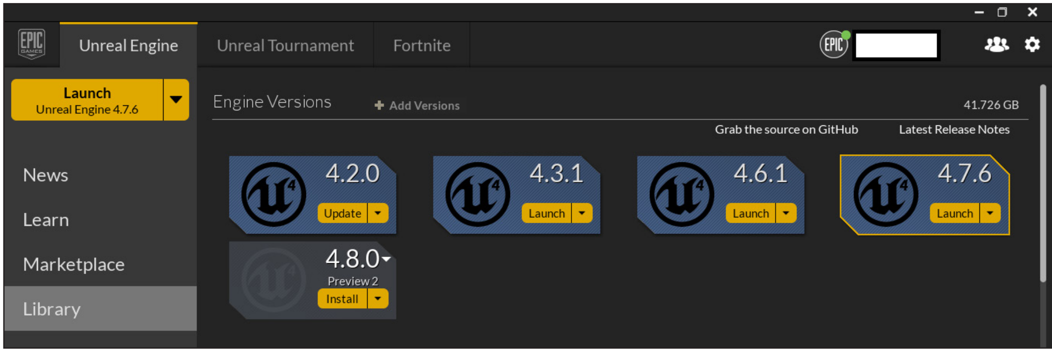
It also has the Marketplace Business Terms, which has all the information regarding things like how the revenue from the sales will be split, how you will get paid, when you will get paid, and so on. It also has the Marketplace Submission Guidelines, which explain things like the submission process, 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.



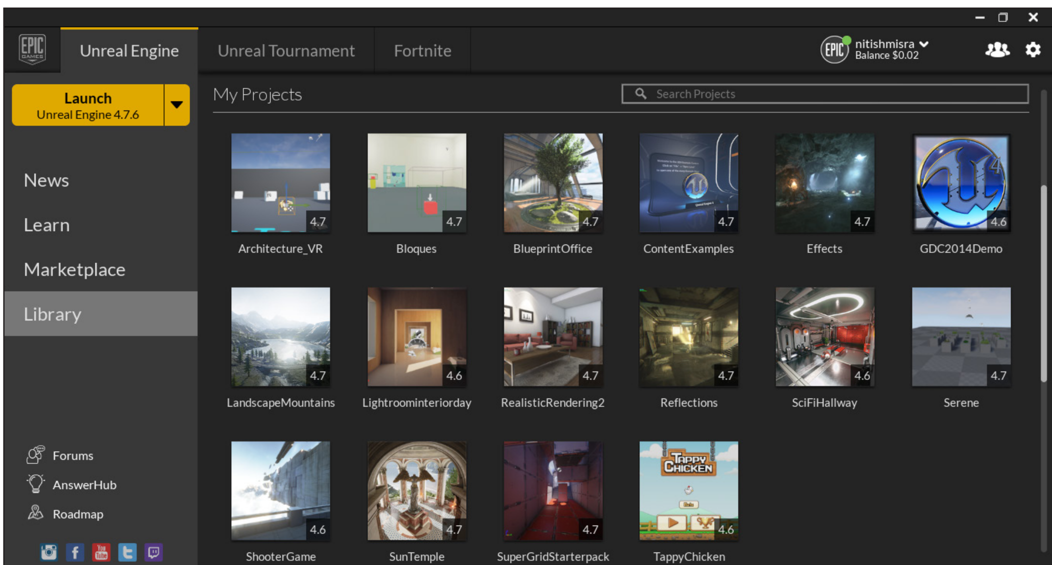
Library has 3 sections, **Engine Versions**, **My Projects**, and **Vault**. The **Engine Versions** section displays all versions of UE4 currently installed on your system. You can launch any version of the Engine listed from here. Additionally, you can also download the latest version or previous versions. 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.

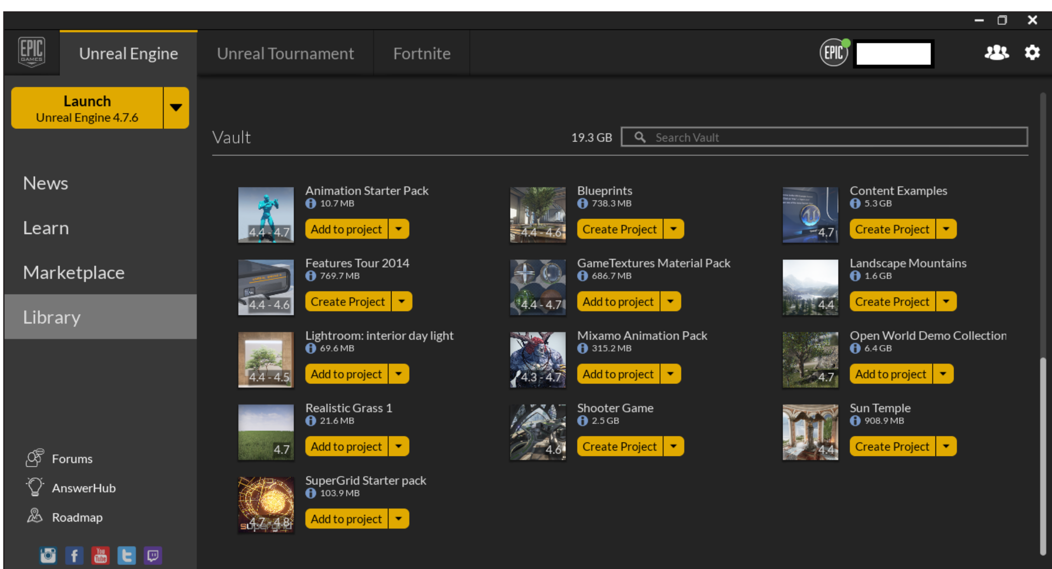


The projects are categorized alphabetically. At the top-right corner is the search bar. In the preceding screenshot, there are relatively few project files; therefore, it is easy to find a specific project. However, if you have lots of projects, it might be harder to find the project you are looking 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**.



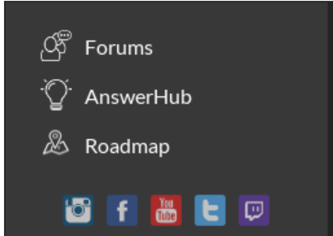
The preceding screenshot demonstrates what **Vault** looks like and how the items are arranged. On the left is the thumbnail of the item, followed by the name of said item. Below the name is the amount of space occupied by that item. The blue **i** icon below the name is information regarding compatibility. Hovering the cursor over **i** 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 forum's 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.

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

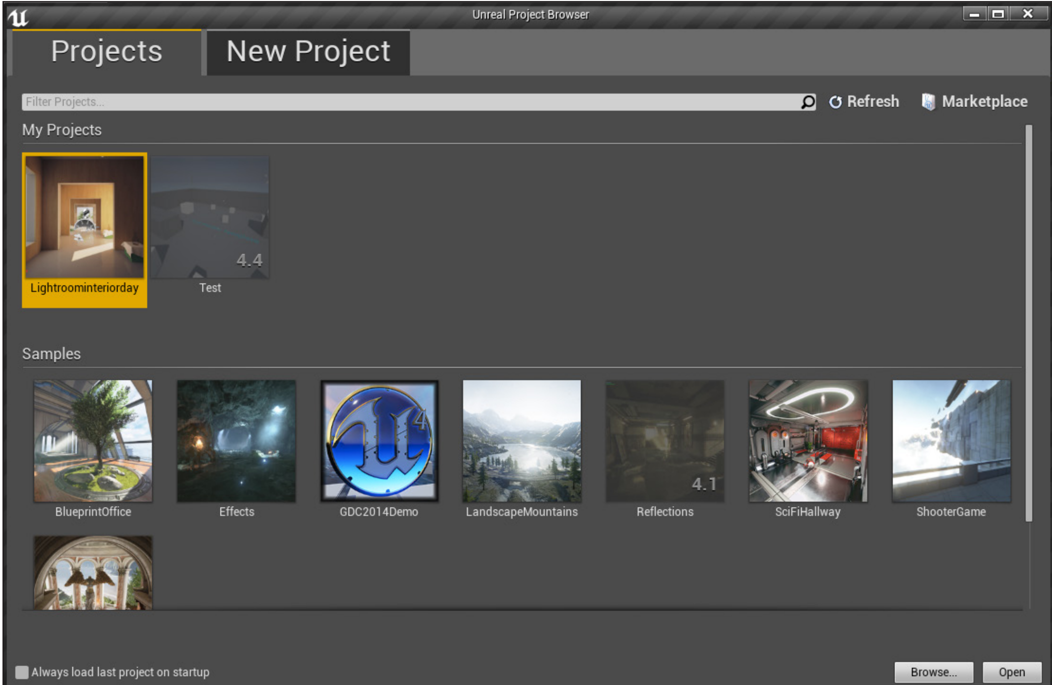
The Editor is where you make the game. All of the assets you create for your game are implemented via the Editor. You set up your environment and the levels in the Editor, and all of the code sequences you create can be tested here; the testing, debugging, and packaging of your game is done here as well.

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.



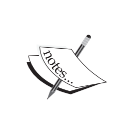
At the top, just below the tab bar, are two tabs, namely **Projects** and **New Project**. Each of these tabs contains certain features that we will go through.

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.

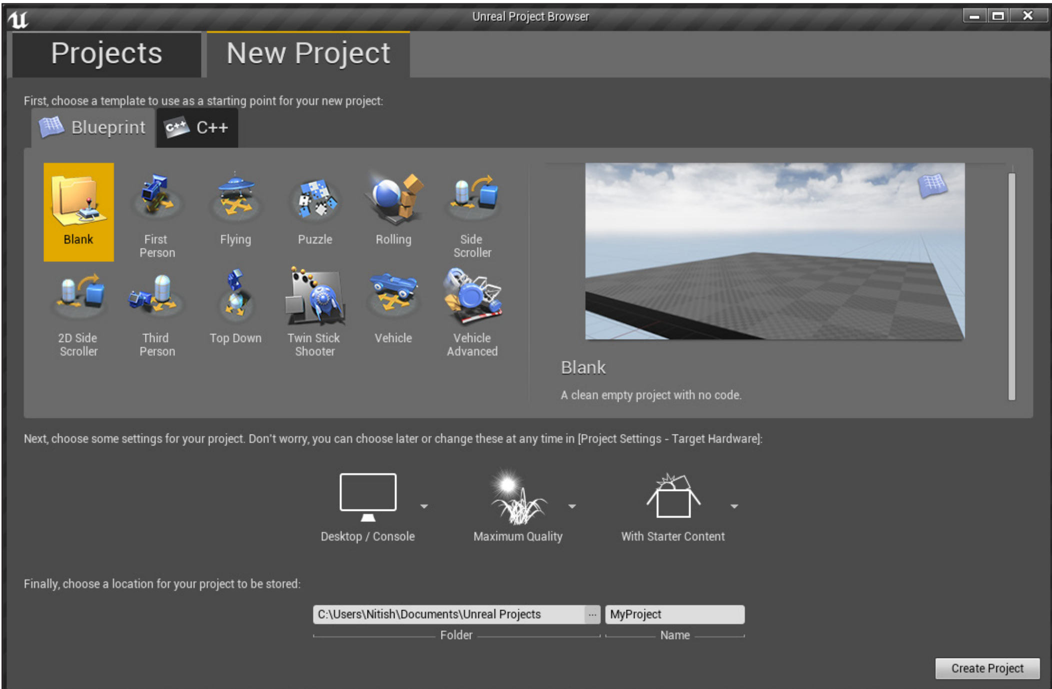


While conversion works well for upgrading projects, downgrading a project (for example, converting a project created in version 4.6 to be compatible with version 4.4) has a few complications. For one, although you will be able to use most of the assets created in the project, you will not be able to 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.

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.

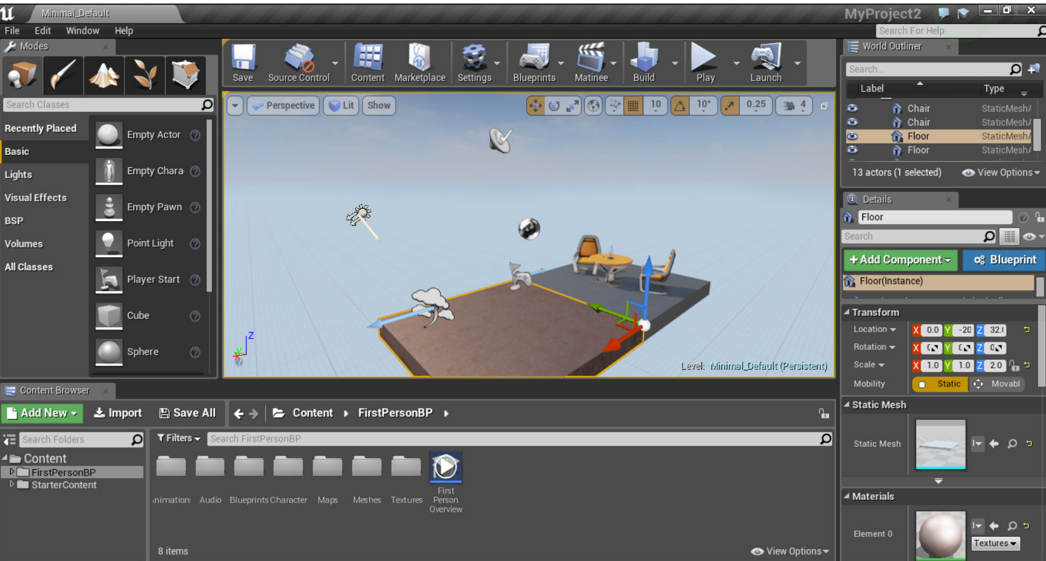
After deciding which template to use, there are a few settings you can select before opening the project. First of all, you can choose which hardware you are going to develop your game on, either PC/console, or mobile/tablet.

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.

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

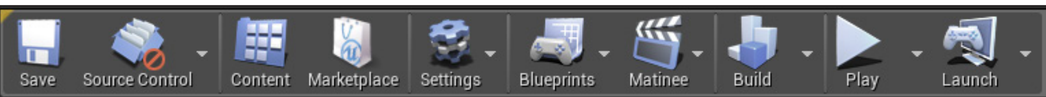
Sometimes you may not be able to find the windows tab. For example, in the preceding screenshot, the Viewport does not have a tab. This is because its tab is hidden. To unhide it, just click on the little yellow arrow located at the top-left corner of the window.

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).

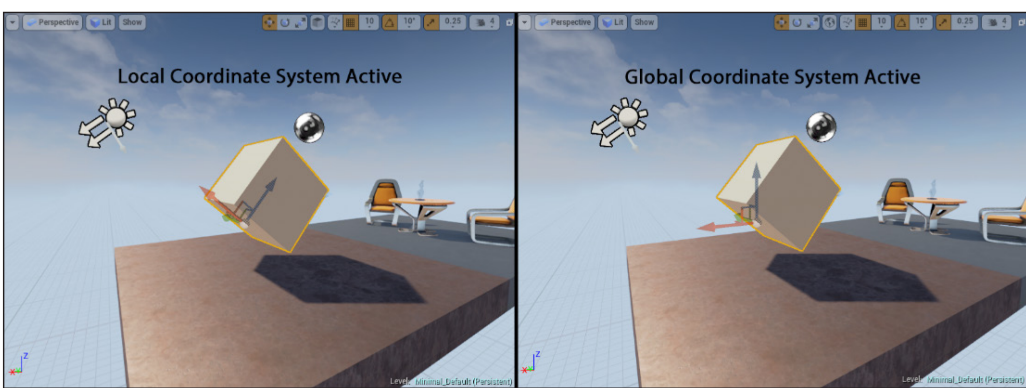
Viewport

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 **View Mode** 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).

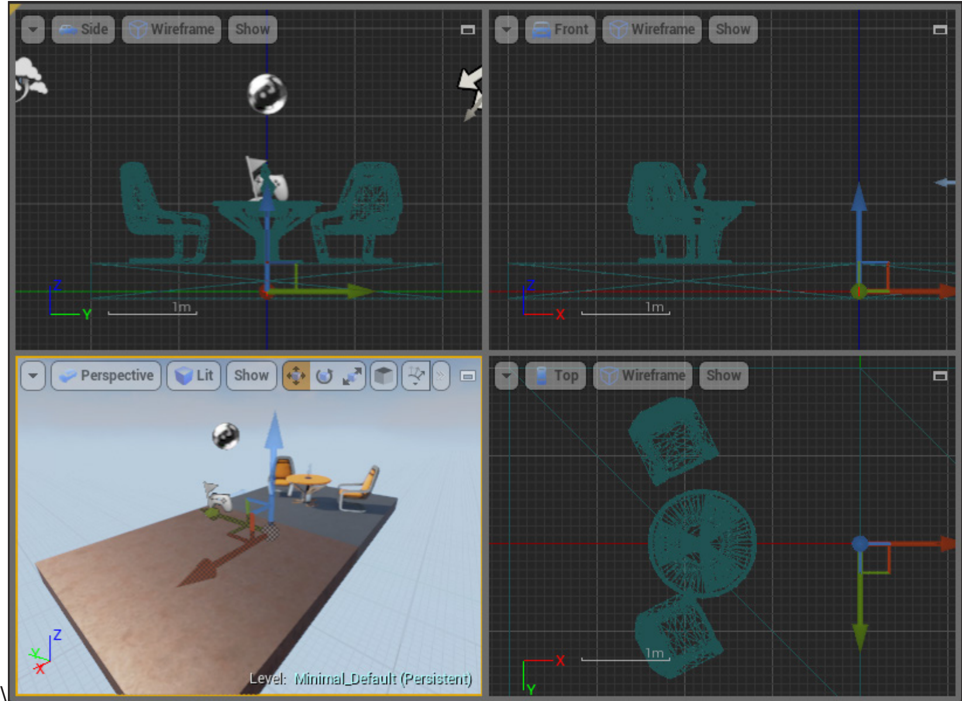
- **Coordinate System**: The next item in the Viewport Toolbar, represented by an icon shaped as a globe, is the Coordinate System. There are two coordinate systems in which any transform action takes place: global, and local. You can click on the button to switch between them. If the global coordinate 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 the 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.

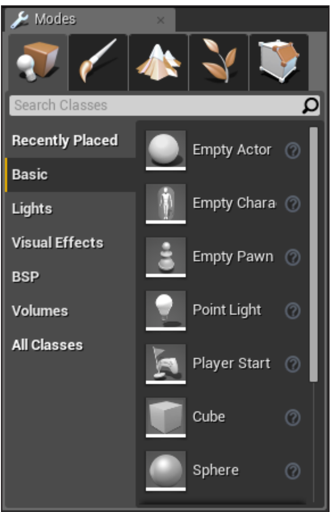
- **Rotation Grid Snapping and Rotation Grid Snap Value**: The next two tools are similar to Grid Snapping, the only difference being that these tools are for the rotation action. When active, the actor will rotate in set values (for example, 10 degrees). This value can be changed in the Rotation Grid 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 comment 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 is 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 games 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.

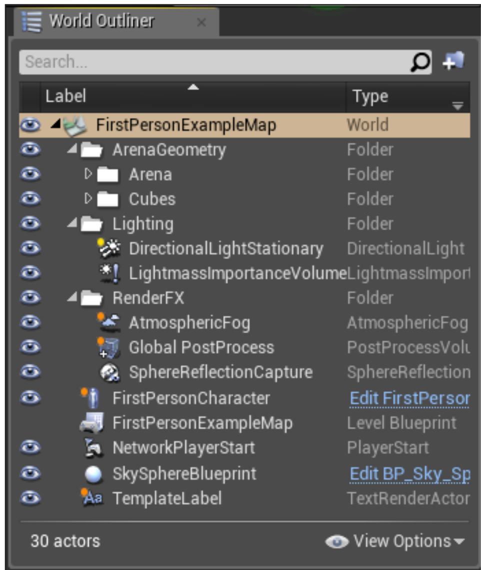
- **Paint Mode**: Paint Mode allows you to paint and adjust colors and textures onto static meshes. Here, you can set the brush size, falloff radius, strength, and more. One thing to note is that you can only paint on the actor that is currently selected to ensure that you only paint on the mesh and 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 **AllDescendants**.

- **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.

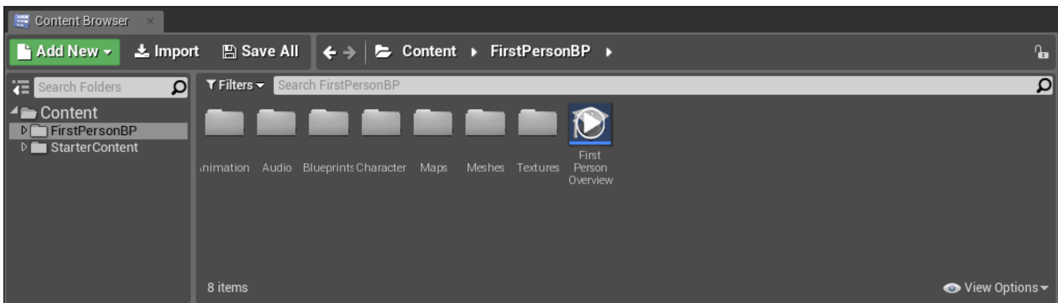
  Attaching forms – a hierarchy: The actor to which you attach other actors becomes somewhat of a parent actor. When you move the parent actor, all of the actors attached to it move as well. However, if you move the attached actor, the parent actor will not move.

  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 games 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

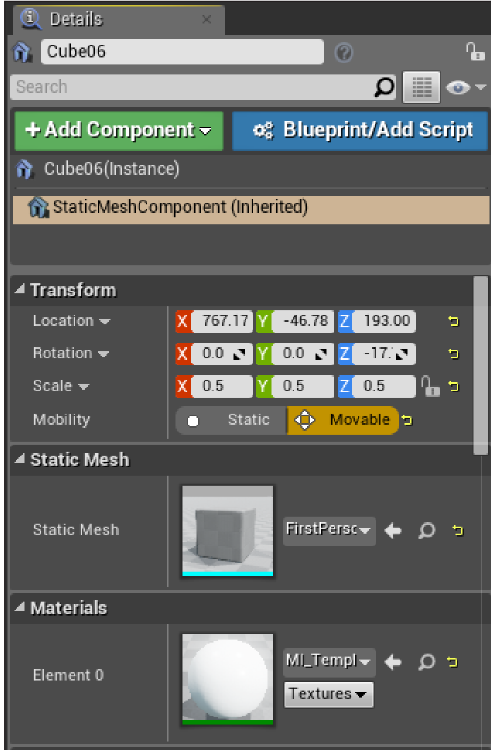
Below these icons, we have the navigation bar. If you have a lot of folders and subfolders in your Content Browser, this will come in handy to help you navigate through them quickly. At the far right corner, there is a small padlock icon, which is, by default, unlocked. If you click on it, it will lock 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 as 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

Control	Action
Scroll down	This moves the camera backward
F	This zooms in and focuses on the selected actor
Arrow keys	This moves the camera forward, backward, left, and right
W	This selects the translate tool
E	This selects the rotation tool
R	This selects the scale tool
W + any mouse button	This moves the camera forward
S + any mouse button	This moves the camera backward
A + any mouse button	This moves the camera left
D + any mouse button	This moves the camera right
E + any mouse button	This moves the camera up
Q + any mouse button	This moves the camera down
Z + any mouse button	This increases the field of view (goes back to the default value when the buttons are released)
C + any mouse button	This decreases the field of view (goes back to default value when the buttons are released)
Ctrl + *S*	This saves the scene
Ctrl + *N*	This creates a new scene
Ctrl + *O*	This opens a saved scene
Ctrl + *Alt* + *S*	This lets you save a scene in a different format
Alt + left-mouse button+ drag	This creates a duplicate of the selected actor
Alt + left-mouse button + drag	This rotates the camera in a complete 360 degrees
Alt + right-mouse button + drag	This moves the camera forward or backward
Alt + *P*	This lets you enter into play mode
Esc (while playing)	This escapes the play mode
F11	This switches to immersive mode
G	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, lets 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.

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:

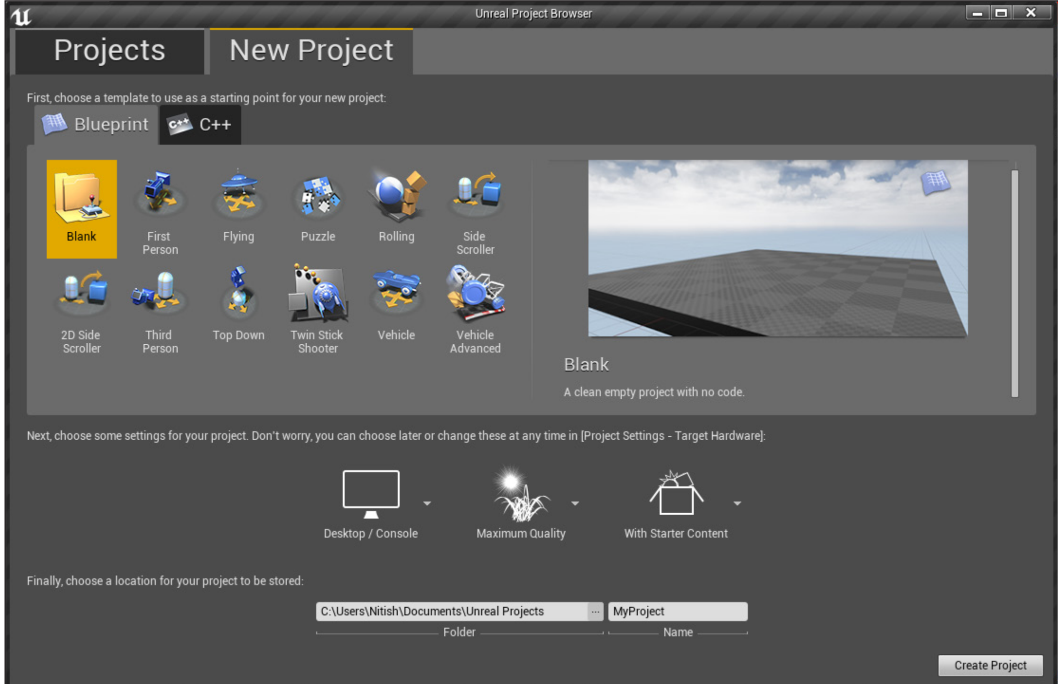
- What a project is, different types of projects offered by UE4, loading, and creating new projects
- Our game's concept, objective, genre, and features
- Geometry and BSP brushes
- Importing assets into the Content Browser, and onto the level
- How to create materials
- Lighting, its types, implementation, 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 Browser** again to better understand what this means:



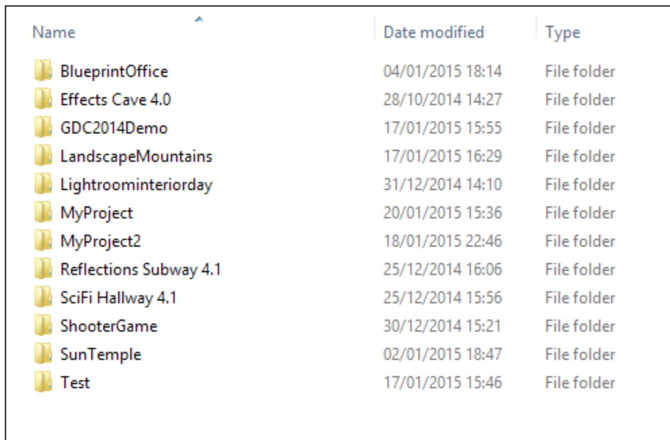
In the preceding screenshot, we see there are various types of templates available. These templates are project files that you can create, which contain the framework for the type of game you need to make. For example, if you want to make a third-person shooter or adventure game, you can choose **Third 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:



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

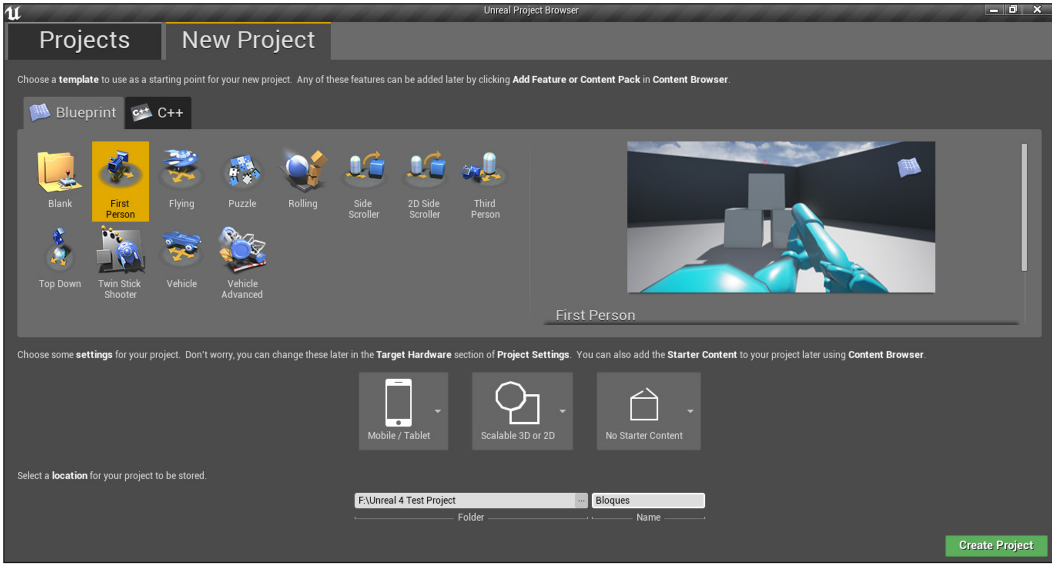
The player controls the character using two virtual joysticks, one for moving and the other for looking. All of the game's interaction, such as picking up objects, opening doors, and so on, will be done via touch.

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`.

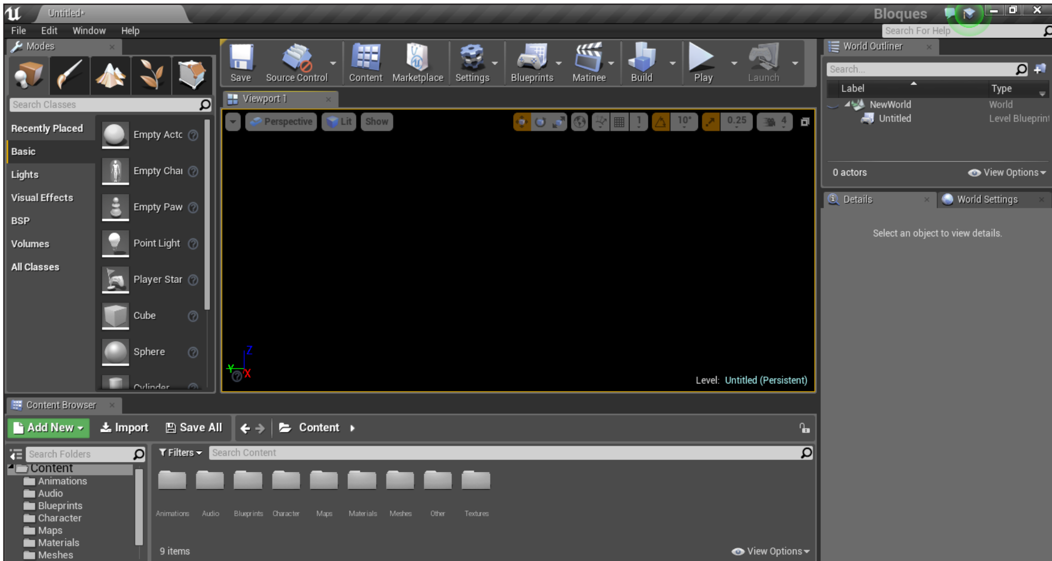
After all of these settings, it should look something like this:



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:

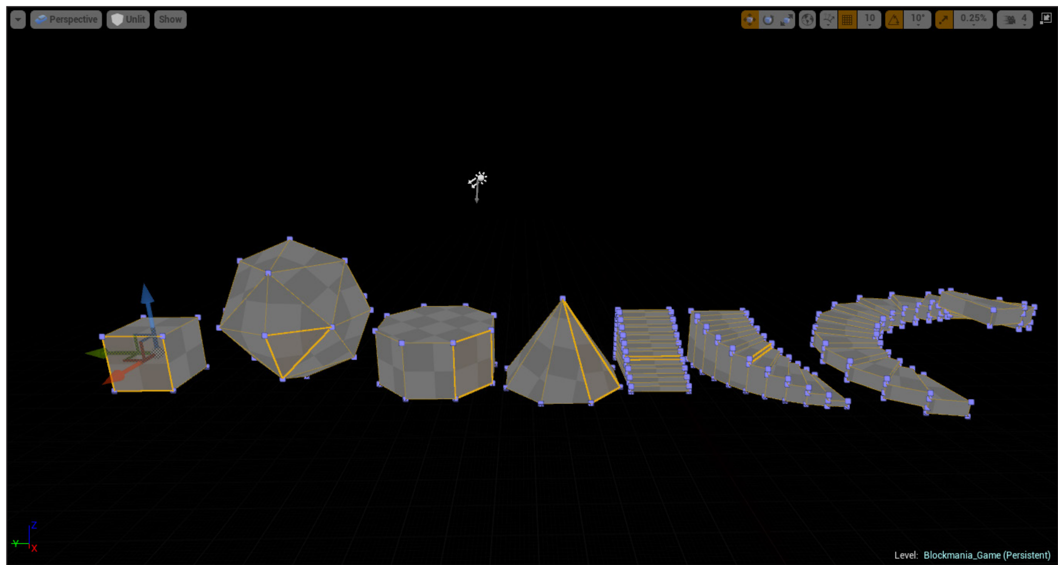


BSP brushes

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.

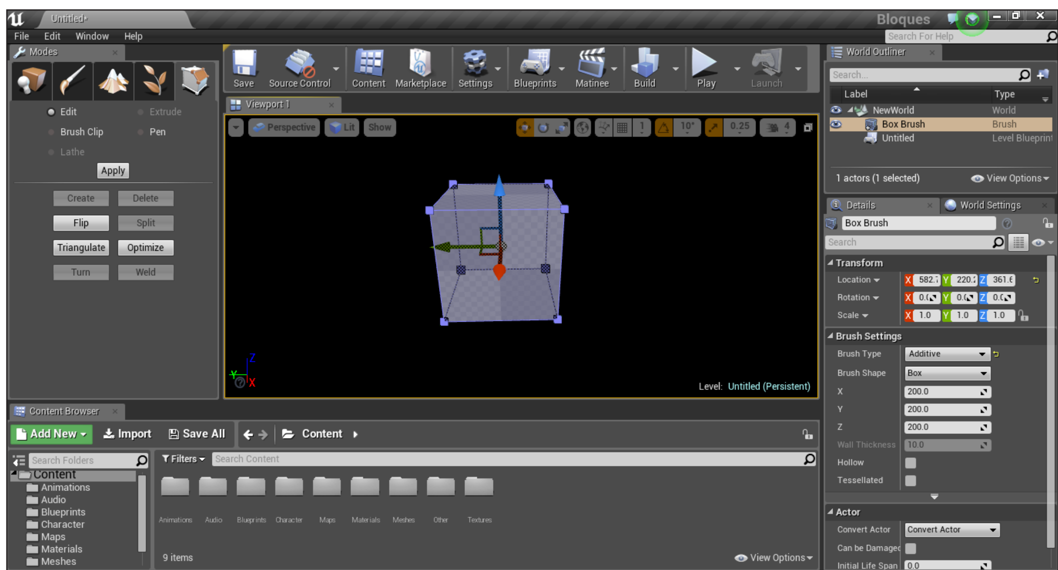
The preceding brush types can be used to create geometry through the use of additive or subtractive brush types through the **Modes** panel under **Place**/**BSP**. When a brush is added to your level and used to create geometry, an additive brush type will add geometry wherever placed. Subtractive brush 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).

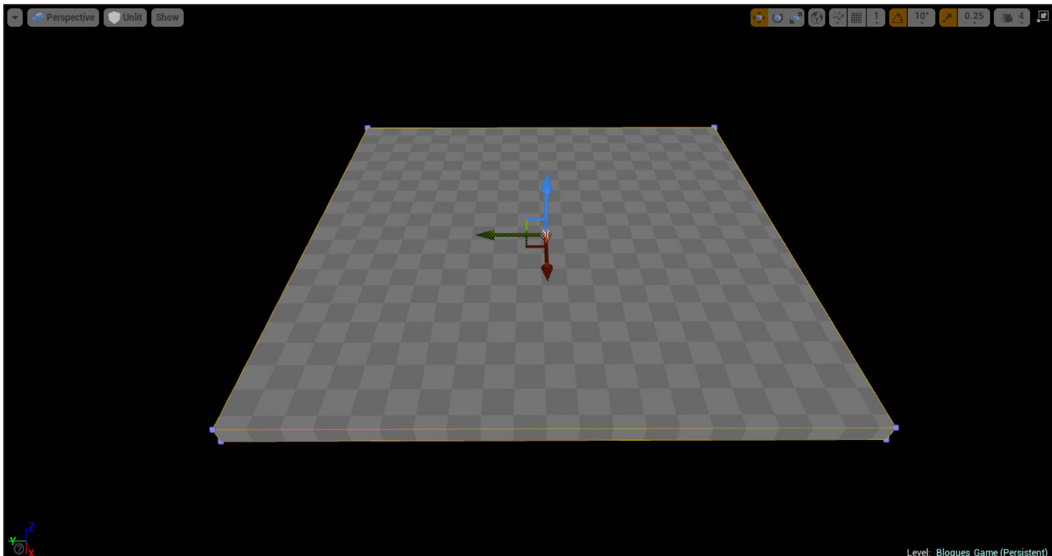
Blocking out the rooms with BSP brushes

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.

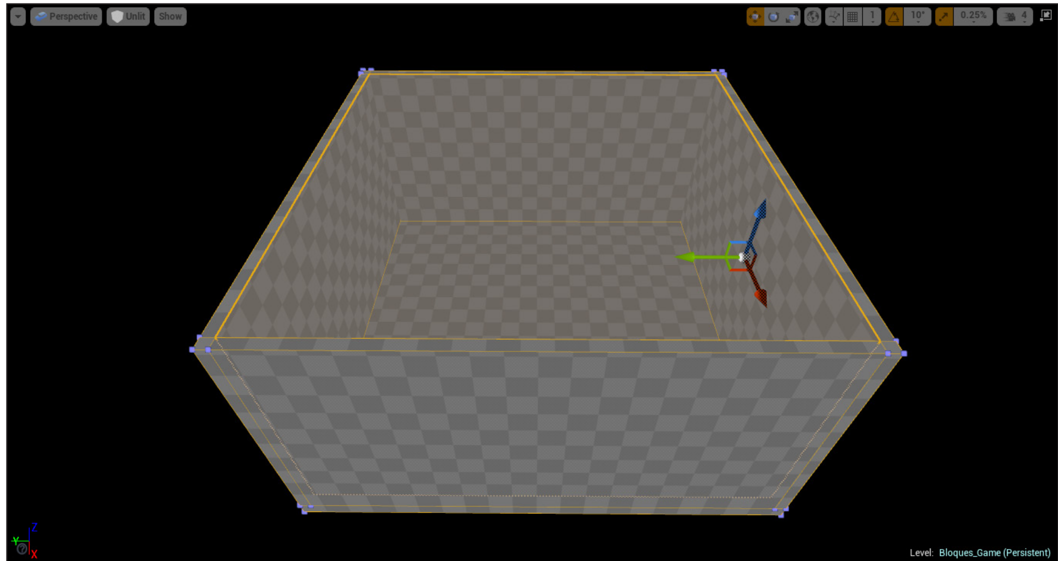
So let's begin by making the floor. We are going to use a **Box Brush**. To add a **Box Brush**, simply click on **Box** in the **Modes** panel under **BSP** and drag it on to the **Viewport**. In the **Details** panel, set the dimensions of the brush as `2048 x 2048 x 64`. We want this room to be relatively 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 x 64 x 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.

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 x 2048 x 1024`. Again, as with the other wall, simply duplicate the BSP brush and move it to the far side of the room.

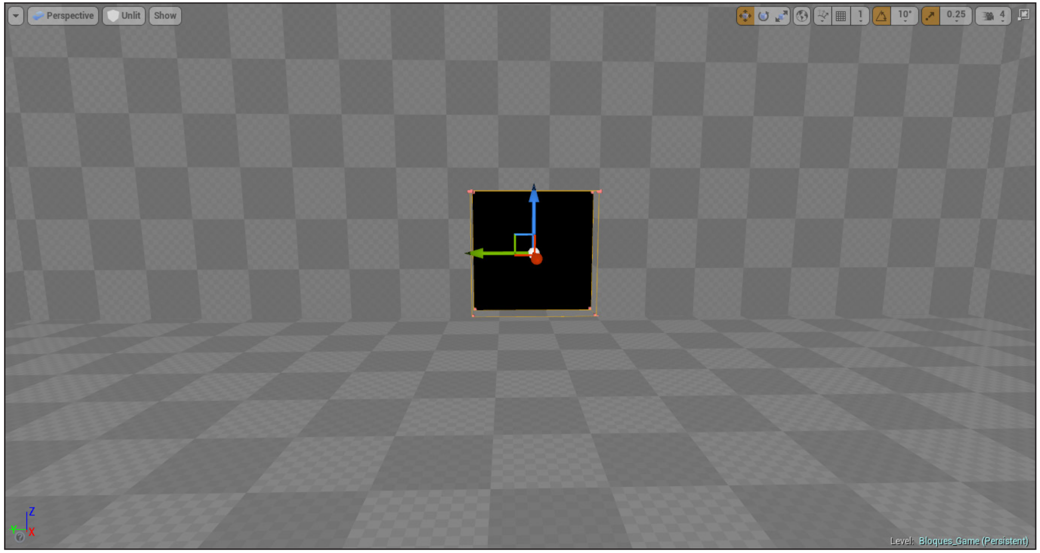


Finally, for the ceiling, simply duplicate the floor and drag it on top of the walls.

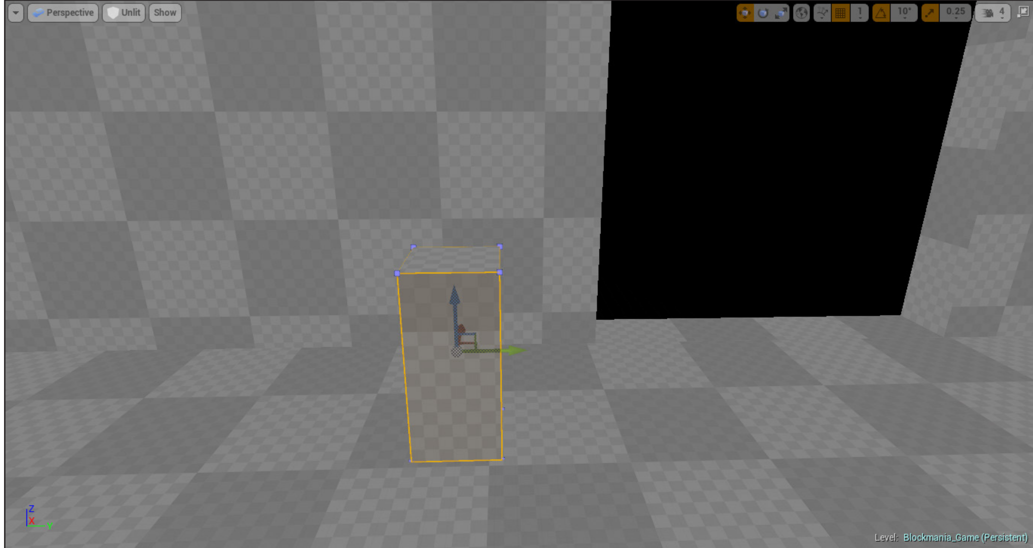
To prevent light bleeding and other complications regarding lighting and rendering, make sure that there is no gap between any of the brushes. Switch to **Top**, **Side**, or **Front** view to make sure all of the walls and ceilings are perfectly lined up.

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 x 256 x 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.



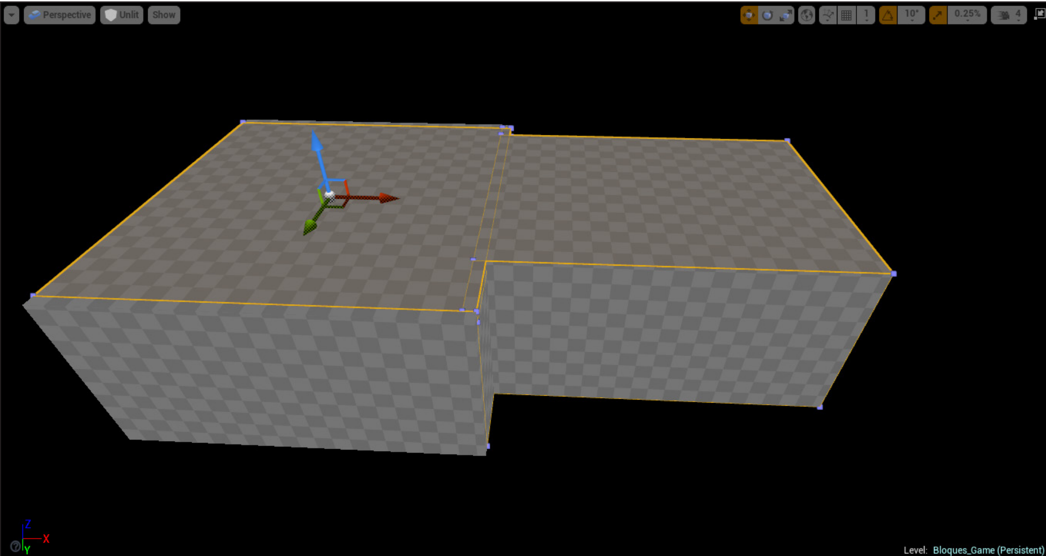
With that, we have now blocked out the first room. Let's move on to the next.

The second room

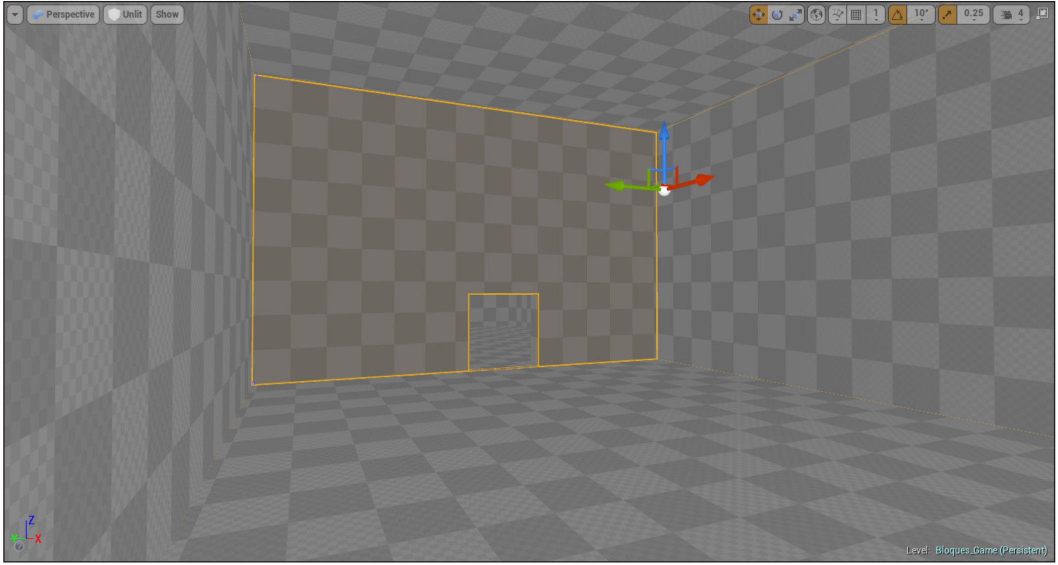
In the second room, we will add a bit of challenge for the player. Upon entering the second room, there will be a large door in the middle. The player can open the door by touching it on the screen. However, as soon as he/she lifts his/her finger or moves away from it, the door closes. To the player's 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 x 1544 x 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:

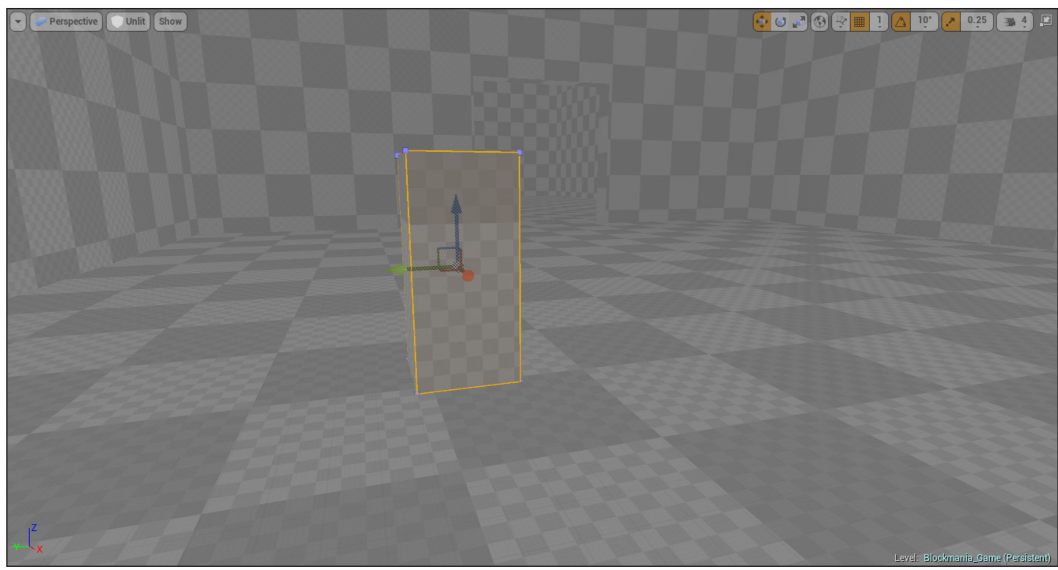


Once positioned, with the brush selected, go to the **Details** panel and tick **Hollow**. As soon as you tick it, a new setting becomes available, **Wall Thickness**. Set its value to **128** (make sure that the subtractive brush we used in the previous room for the door is overlapping both walls; otherwise, you will not be able to see the door).

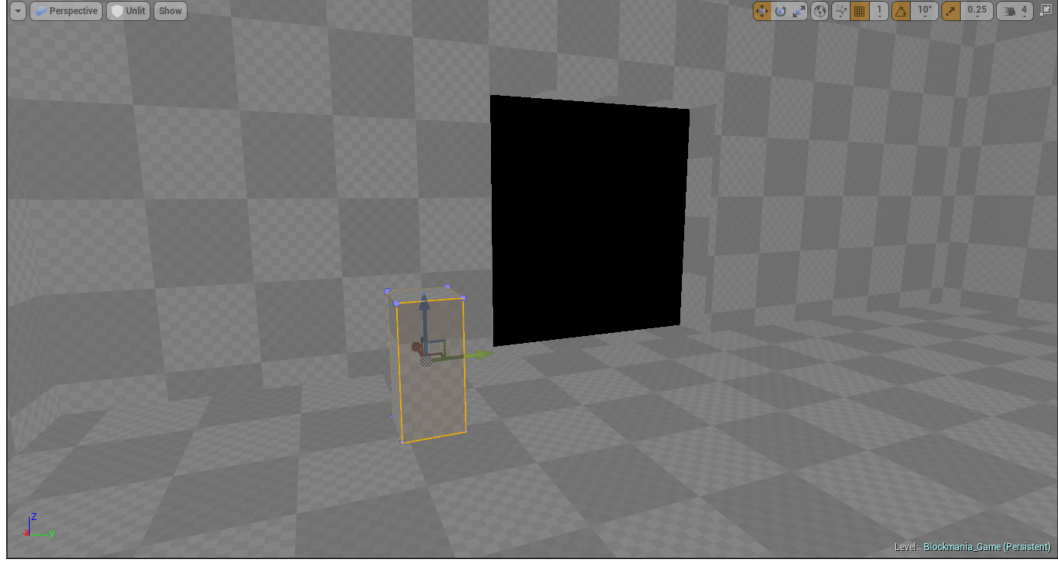


And that is it! We have our second room blocked out without having to spend loads of time placing each part of the room and making sure that they are aligned properly. The only things left to place are the pedestals and the hole for the door that leads to the third room. For the door, simply do what we did for the first room, create a subtractive brush of dimensions `64 x 256 x 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.

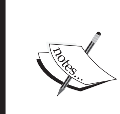
Finally, to finish things off, we will add two pedestals in this room. One pedestal will be near the middle of the room, where the big door will be. The player will have to place the first key cube on this pedestal to unlock the second key cube.



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.



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

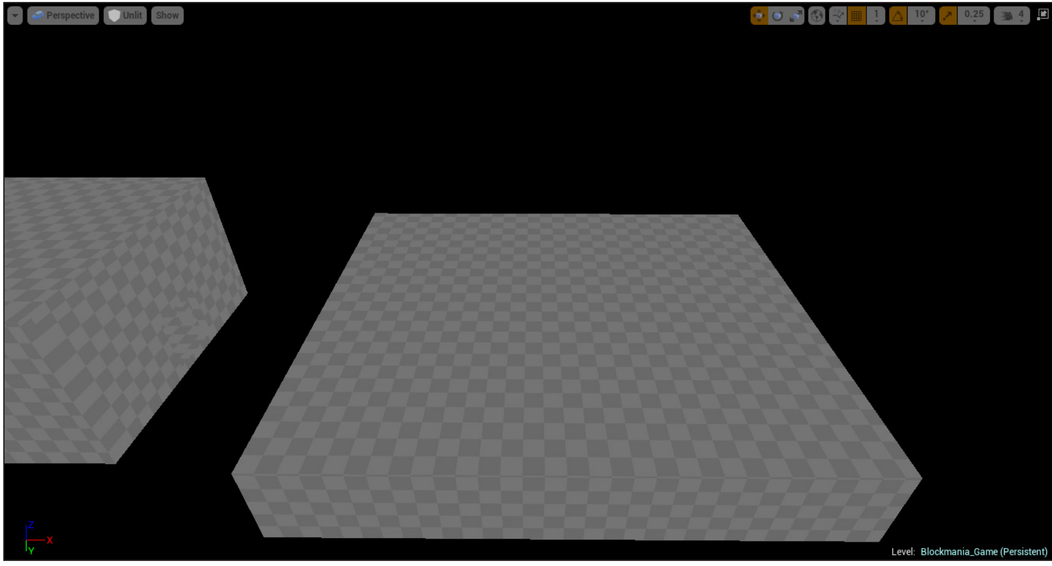
By now, the player would have understood the basic mechanics and controls in the game. Let's now give him/her a bigger challenge in the third room. Upon entering the third room, there will be a pit between the player and the door to the final room. For the player to be able to cross the pit, he/she will 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.

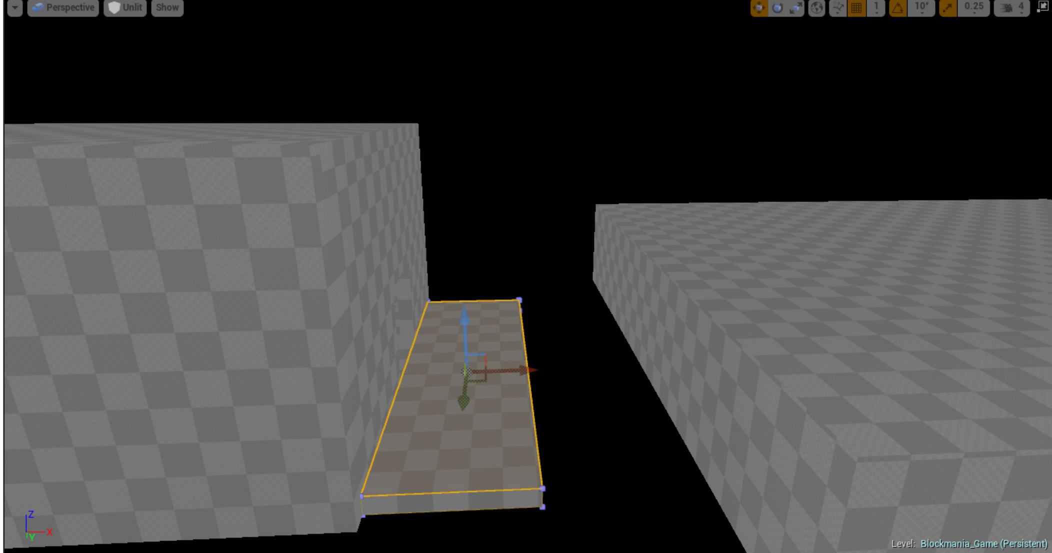
There are two ways of constructing the pit; the first way is to make the room in two parts. The first part would be on one side of the pit, and the second on the other side. After making the two parts, construct the pit, and finally place them. However, this is a bad and time-consuming way of constructing 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.

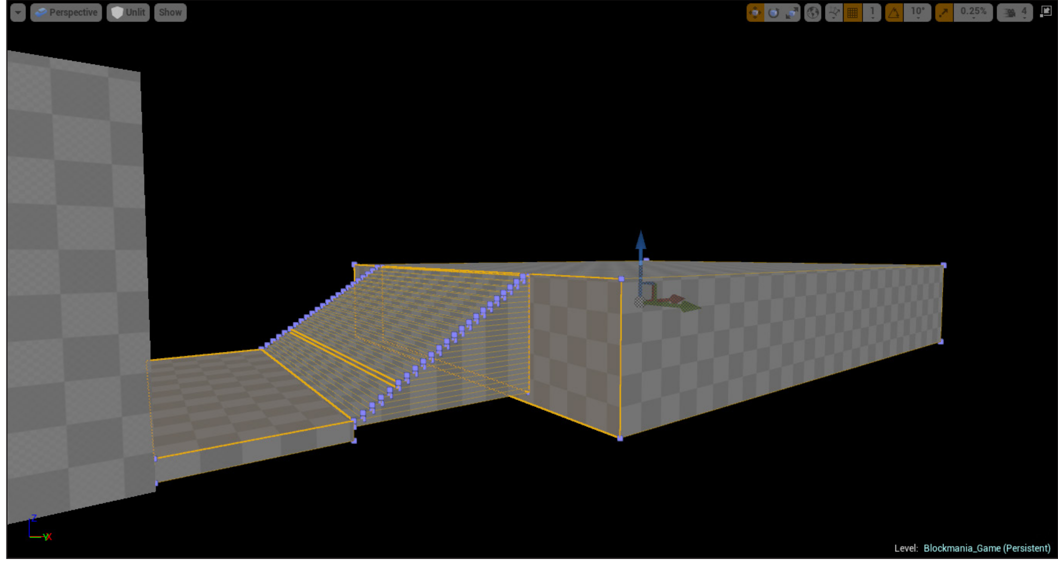
As always, let's begin by making the floor. Now, since we are going to carve out the pit, the floor will have more height than the other rooms. Select the **Box Brush**, and set its dimensions as `4096 x 2048 x 512`. This will be the main area, where the puzzle is going to be.



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 x 1544 x 64` and place it near the door of the previous room.

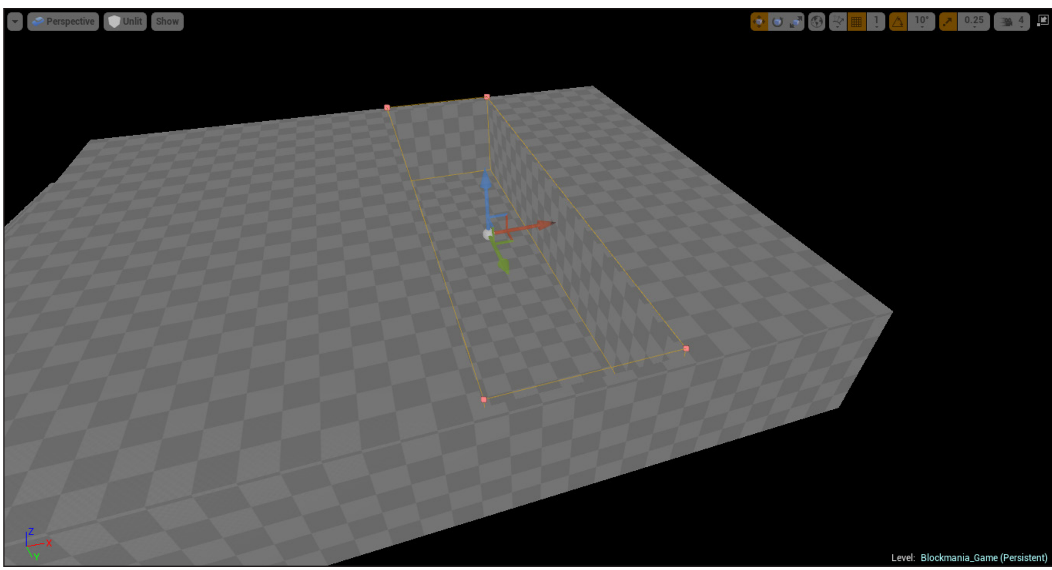


To add the stairs, drag the **Linear Stair Brush** from the **Modes** panel onto the level. Set the width of the stairs as `1544`, and the number of steps to `23`. Place the stairs at the edge of the **Box Brush** we placed earlier; finally, take the floor of the third room and put it adjacent to the stairs.

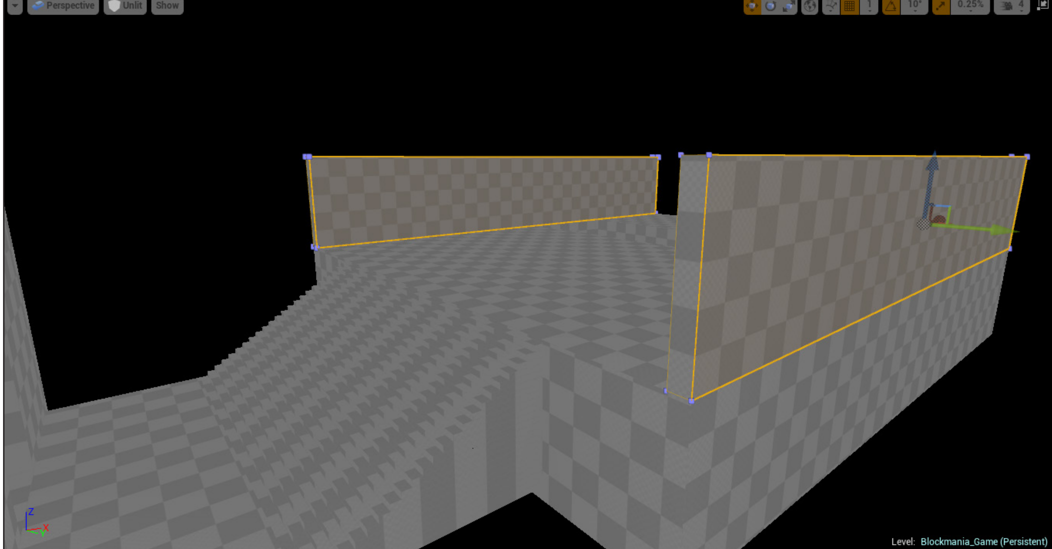


To add the pit, we are going to need a Box subtractive brush. Another way of selecting a subtractive brush is by first selecting the shape of the brush, which in this case is the **Box Brush** and then, in the **Modes** panel, at the bottom, you can set the brush type to be additive or subtractive. Simply select subtractive and drag the brush onto the level.

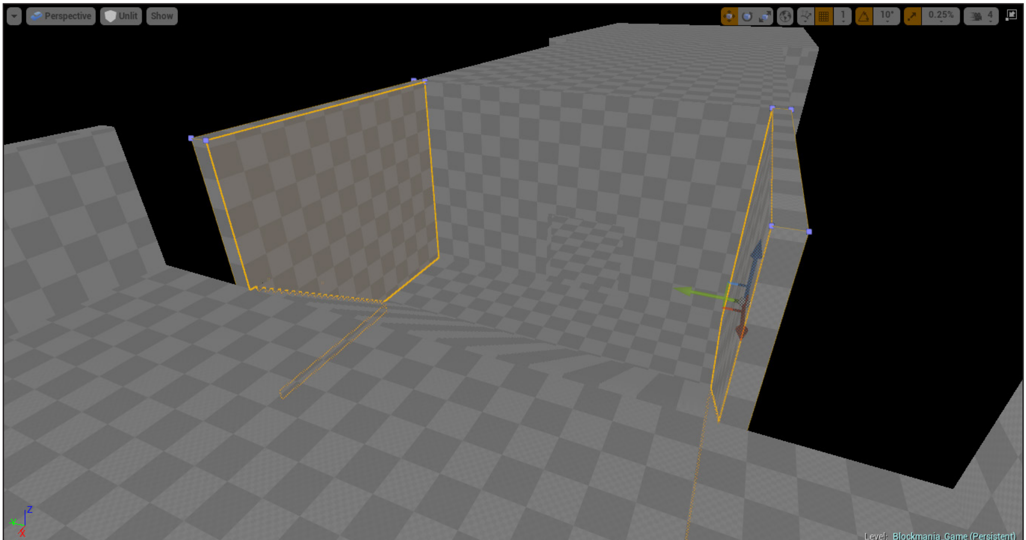
We have to make sure that the pit is wide enough so that the player cannot jump across it and deep enough that the player cannot jump out of should he/she fall into it. Keeping that in mind, set the dimensions of the brush as `640 x 2304 x 512`. Place the pit near the other end of the room.



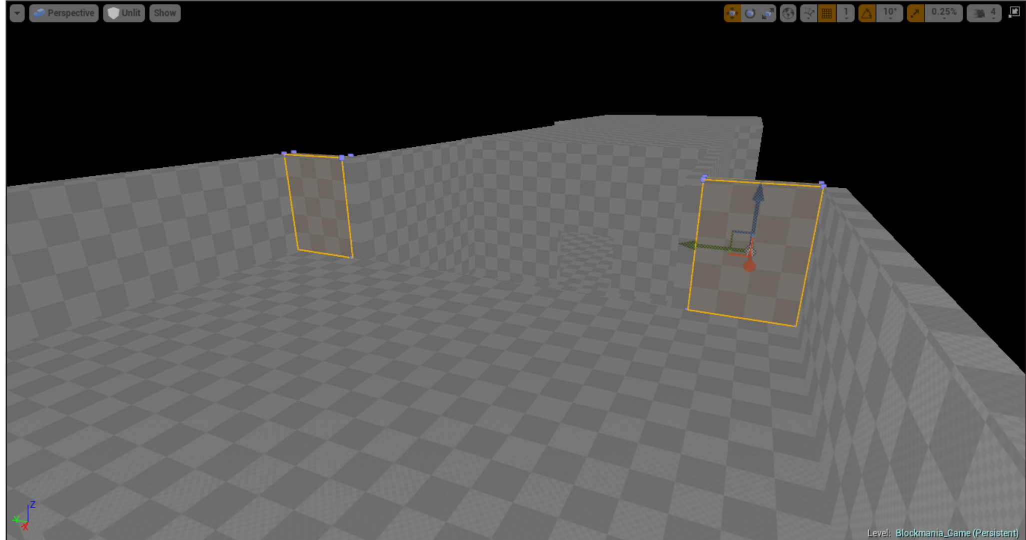
We now have to place the walls. We will add walls to this room in a few steps. First up, select a Box brush, set its dimensions as `4096 x 64 x 512`, and place it along the longer side of the room. Duplicate and place the second wall on the other side.



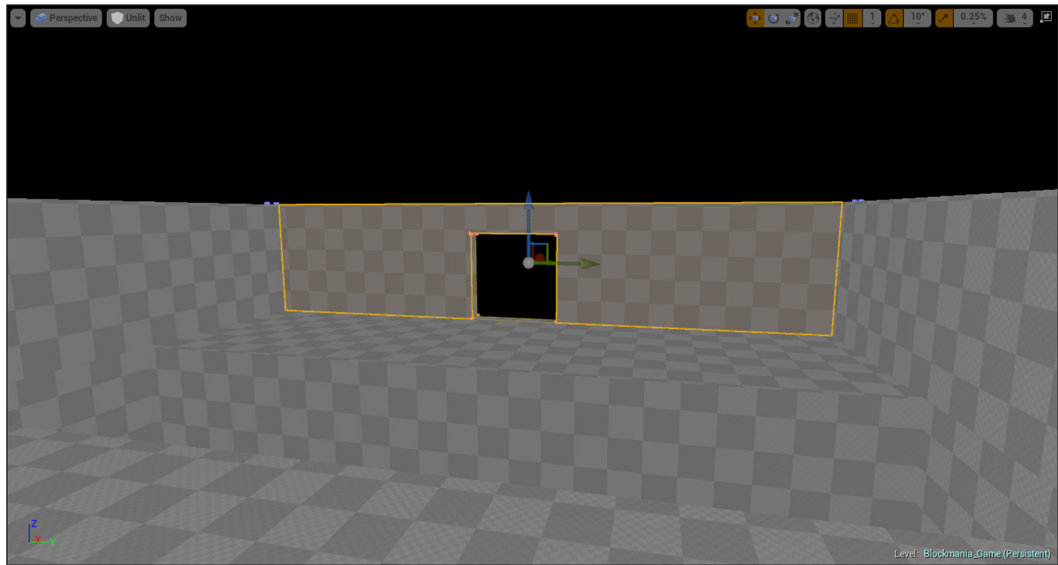
Next, we need to place some walls along the stairs and corridors that lead to the third room. For that, set the dimensions as `1202 x 64 x 1024` and place them on either side of the stairs, ensuring all of them line up properly and there is no gap anywhere between the brushes.



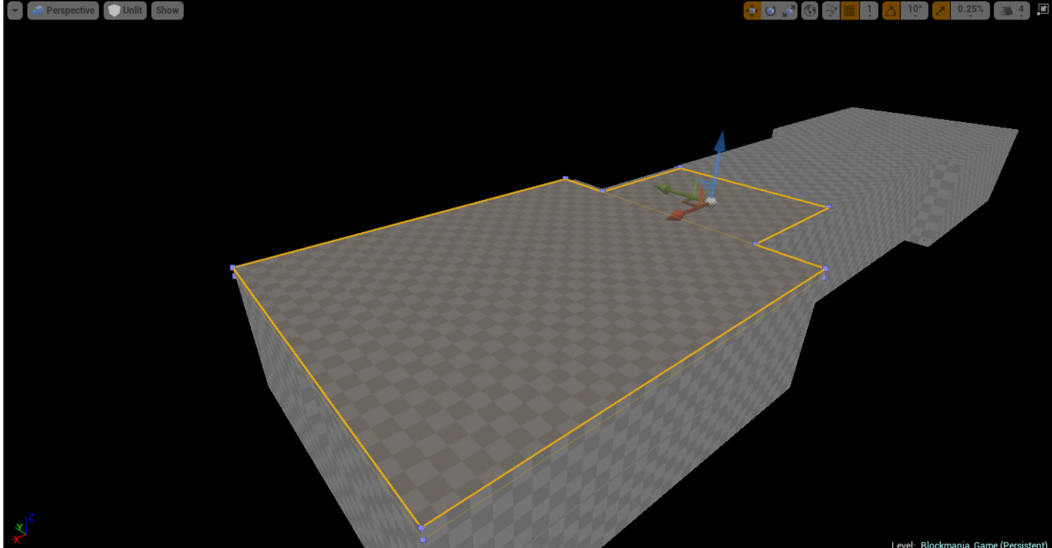
In the preceding screenshot, you may have noticed that there is a gap between the walls of the corridor and the walls along the longer side of the room. Let's fill that in with a Box brush of dimensions `64 x 188 x 512`. Place two of them, one on either side, to fill in the gaps.



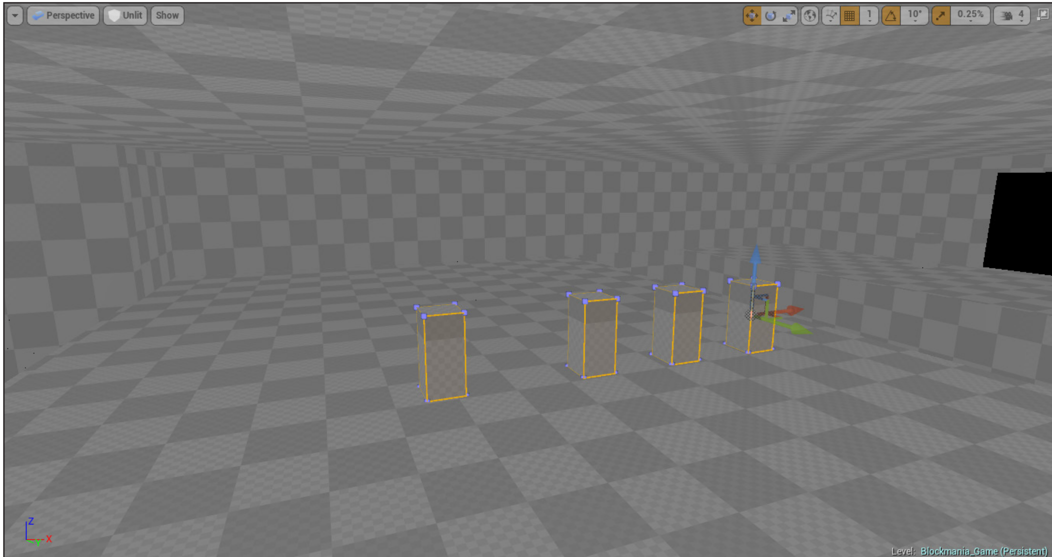
Finally, the wall that we are going to place is on the far side of the room, along the shorter side. Set the dimensions of the **Box Brush** as `64 x 2048 x 512`, and place the final wall with a subtractive brush for the door.



The ceiling for the corridor will be of the dimensions `1202 x 1544 x 64`, and that of the room will be `4096 x 2056 x 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.

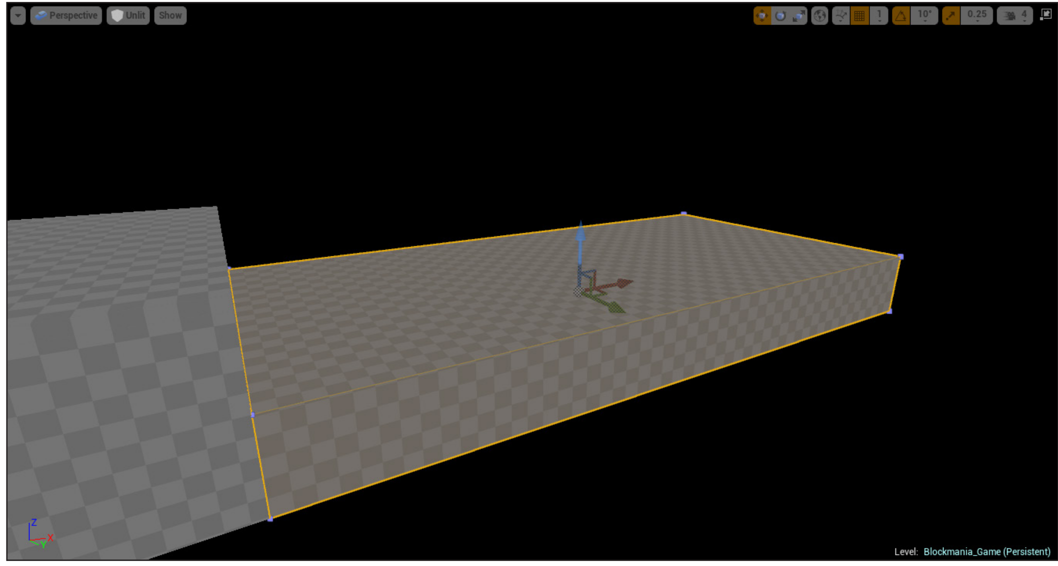


Finally, place a pedestal near the door. With this, we have finished off blocking out the third room. Now, let's block out the fourth and final room.

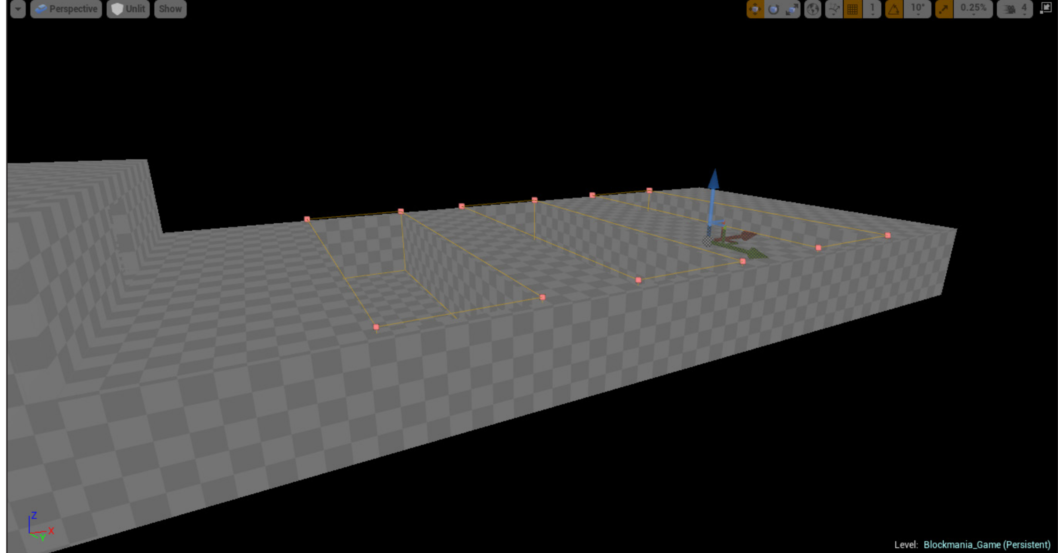
The fourth room

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.

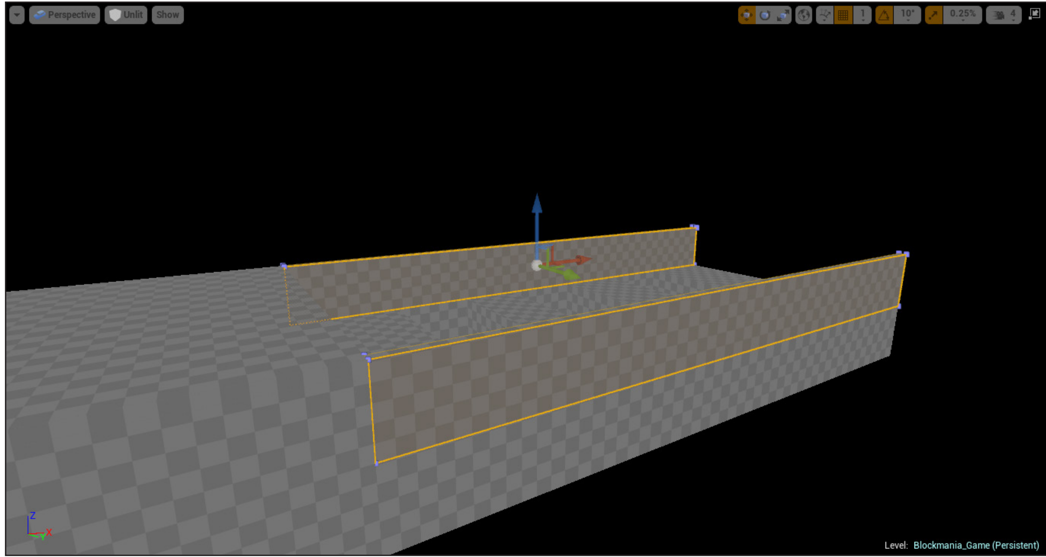
Since we are going to have pits, the height of the room will be similar to the previous ones. Also, this is going to be the biggest room. Keeping that in mind, we will make the floor by selecting a **Box Brush** with the dimensions `5120 x 2048 x 512`.



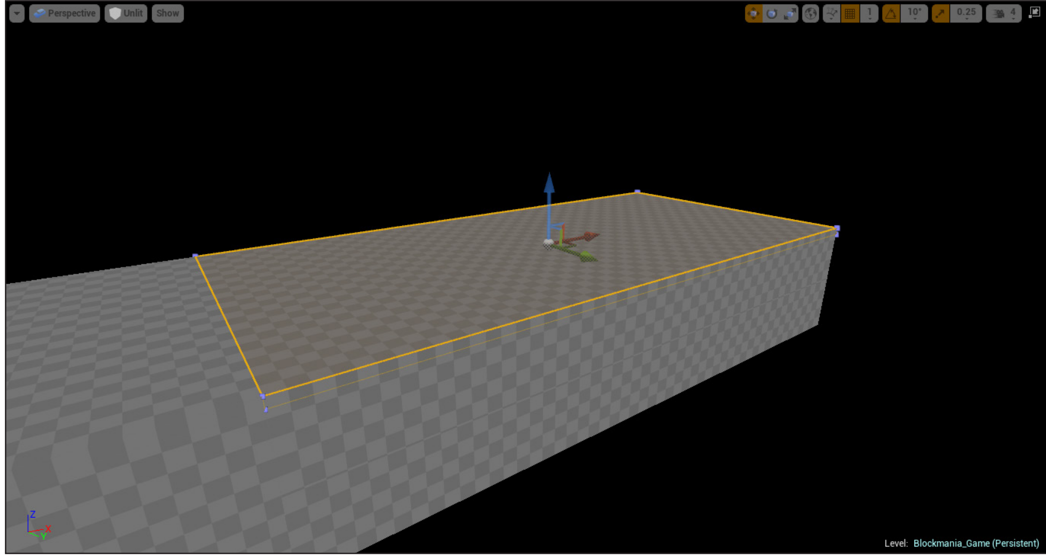
Now that we have the floor, we are going to make some pits. Again, with the subtractive mode selected, create a **Box brush** of dimensions `728 x 1928 x 512`, and place it near the door. In this room, we are going to have three pits, so duplicate the subtractive brush by holding down the *Alt* key, and create two copies and place them along the map.



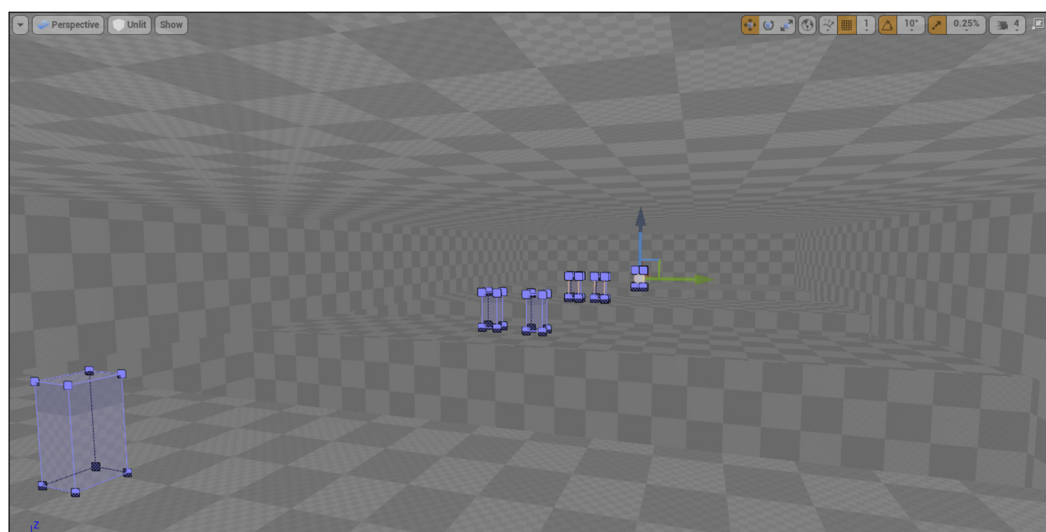
Next, comes the wall. For the longer side, simply duplicate the walls from the third room. In the **Details** panel, set the value of *X* as `5120`, and simply place the wall. Duplicate and place the other wall. For the shorter side, you can do the same—replicate and place the shorter side wall (the 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.

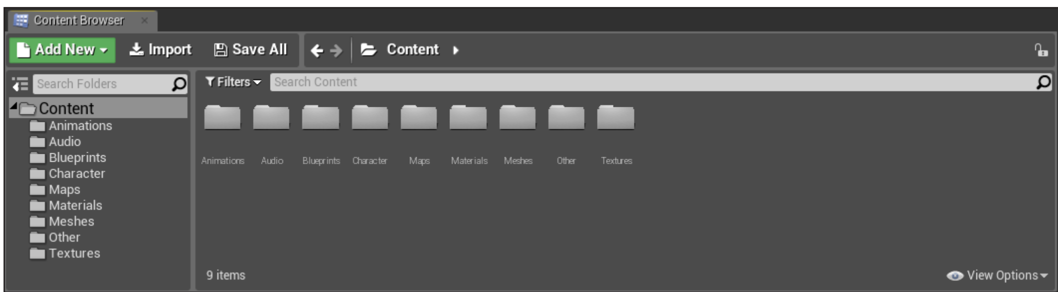


With that, we have now completed blocking out our last room!

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.

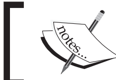
Migrating and importing assets

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

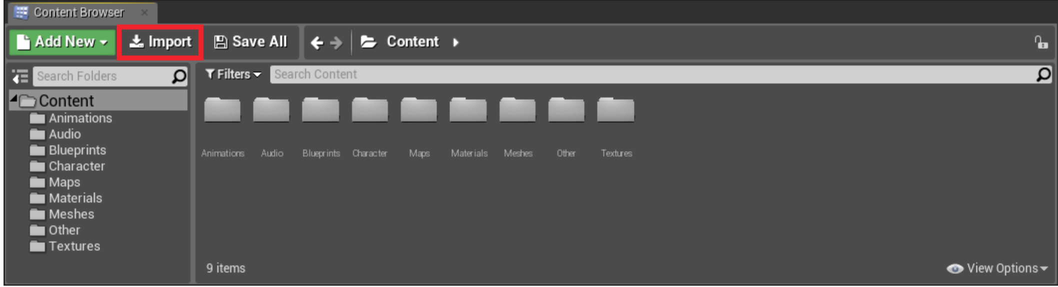
There are two ways of importing assets that you have created for your game in your project file.



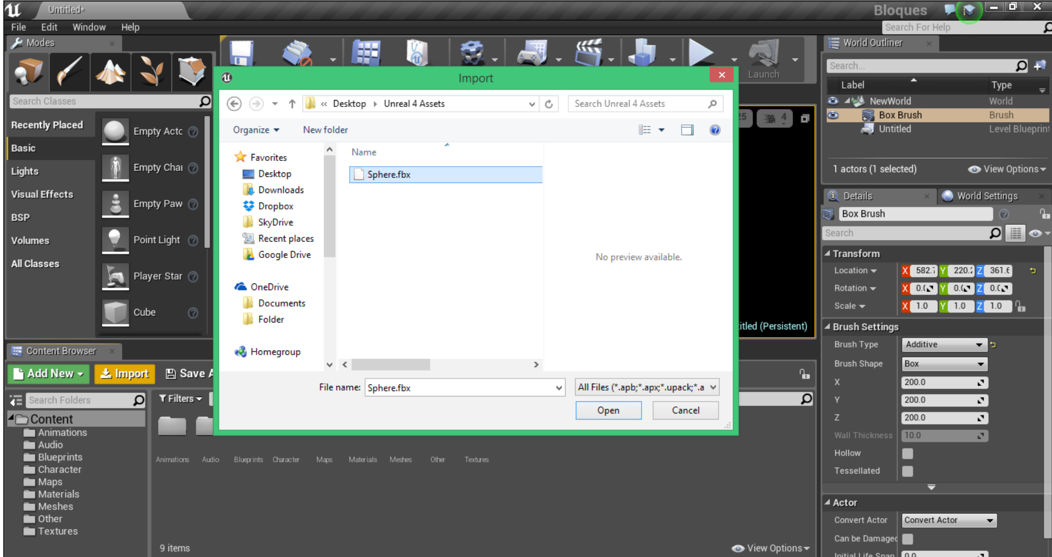
To demonstrate how to import assets, a simple sphere was created in Maya and exported as an FBX file.

The first way is through the **Content Browser**:

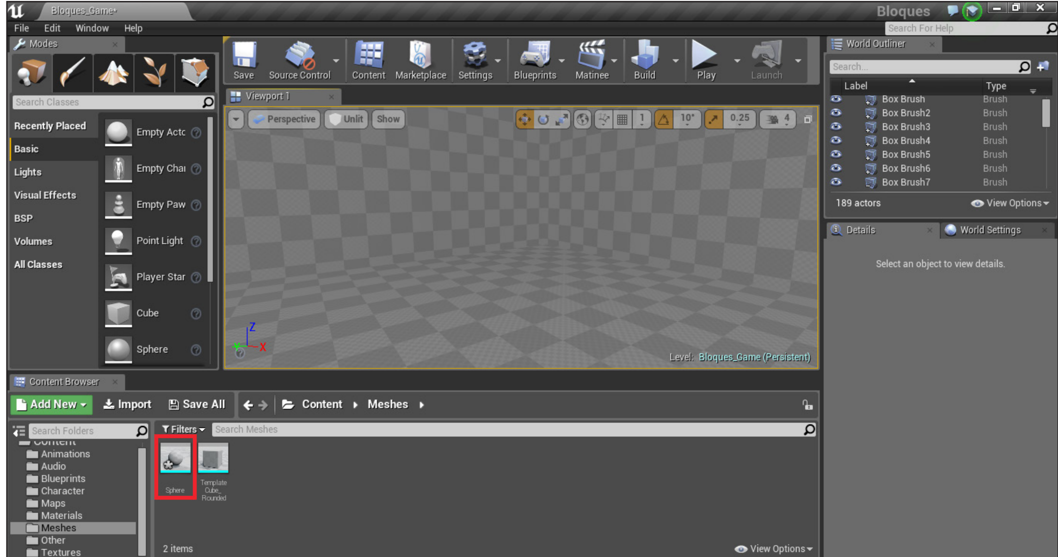
1. Click on the **Import** button located at the top of the **Content Browser**.



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.

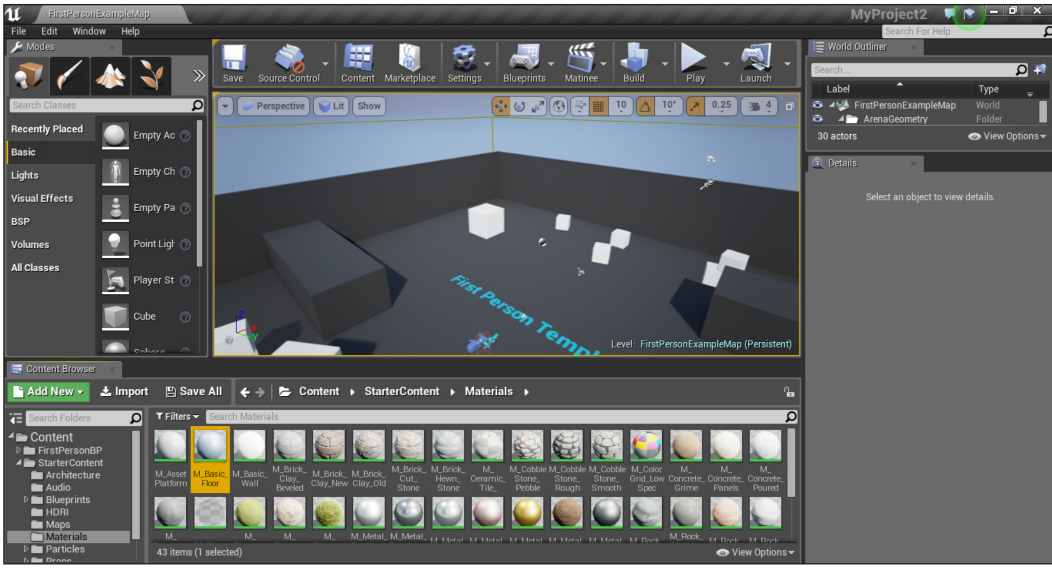


The second method to import assets to your project is by simply dragging the asset you want to import and dropping it in your **Content Browser**. To do so, simply click and drag the asset from wherever it is stored and release the left-mouse button over **Content Browser**. When you release the left-mouse 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

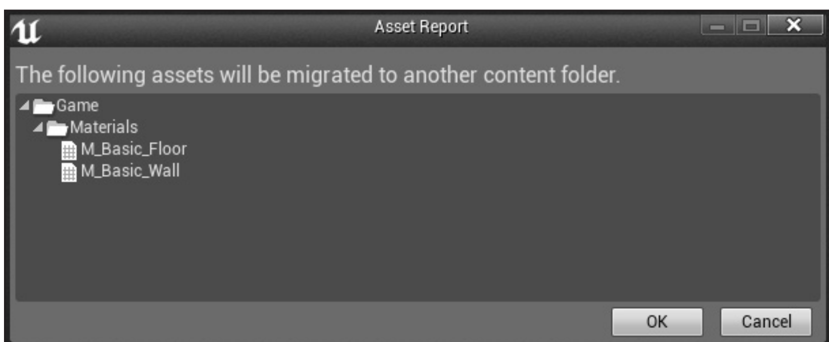
Sometimes, you may need certain assets from a different project file. In such cases, the methods mentioned previously regarding importing assets will not work, since the contents of a project are saved on your system as either a `.uasset` file or `.umap` file. Therefore, UE4 will not be able to import 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.



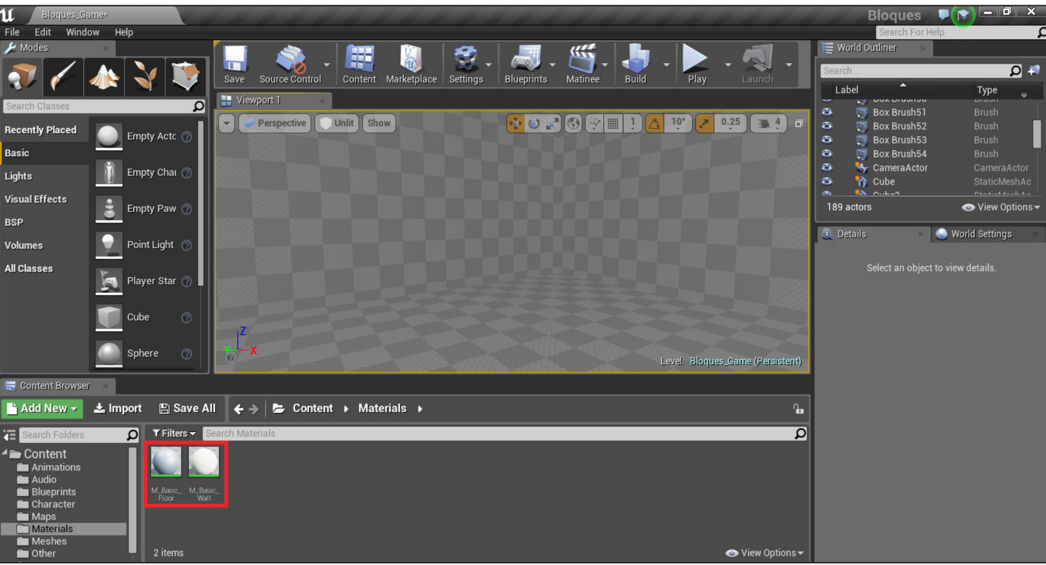
Here is a project with all of the starter content. What we need are two materials, namely **M_Basic_Floor** and **M_Basic_Wall**. Migrate these two materials and follow these steps:

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**.

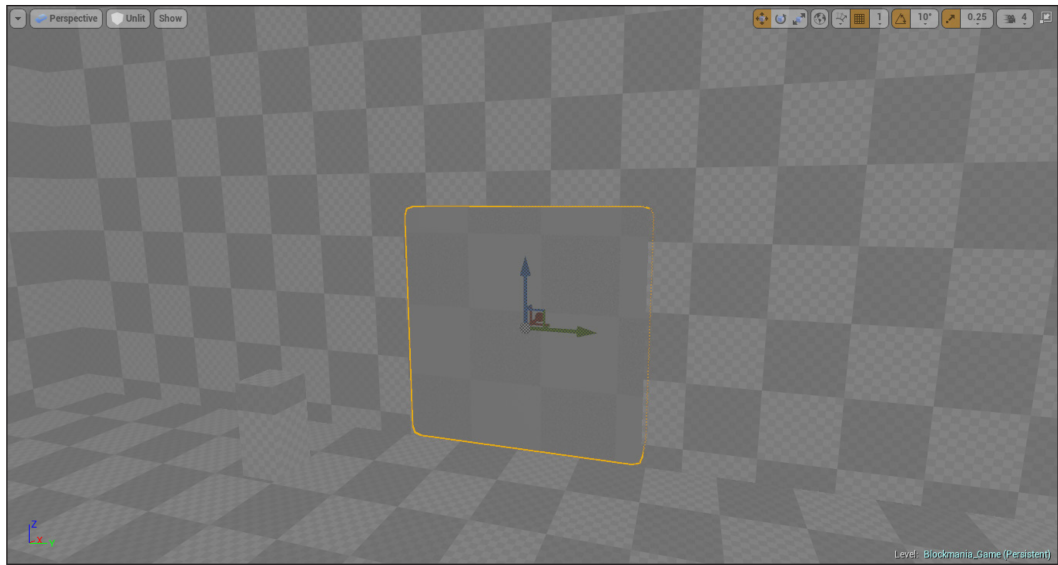
If you now go back to the Blockmania project, you will see **M_Basic_Floor** and **M_Basic_Wall** displayed in **Content Browser** in the **Materials** subfolder.



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.

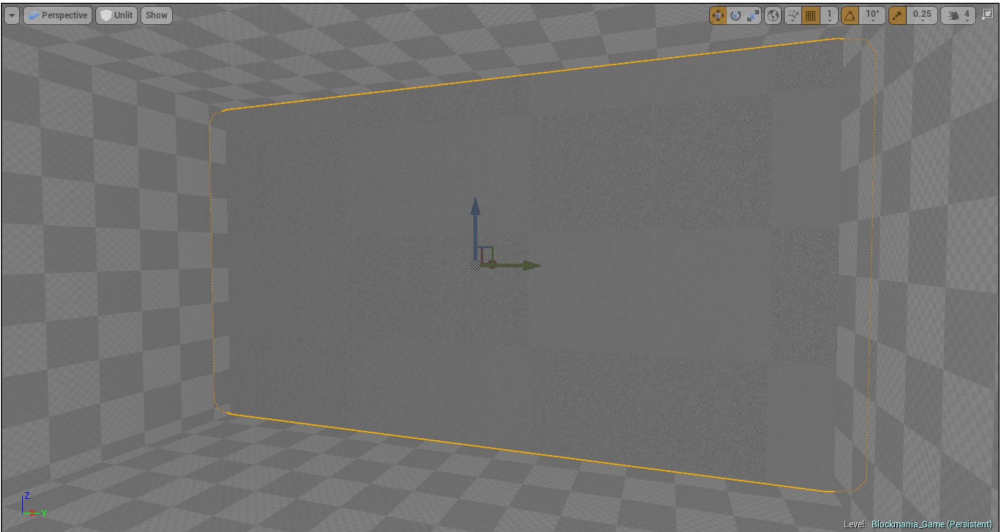
First, let's make the door that opens when you place the key cube on the pedestal. Simply drag the **TemplateCube_Rounded** static mesh from **Content Browser** and place it in the hole where the door is supposed to go. Set its dimensions using the scale tool, so that it perfectly fits.



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**.

[Use different perspectives to set the dimensions and to align the door with the hole.]

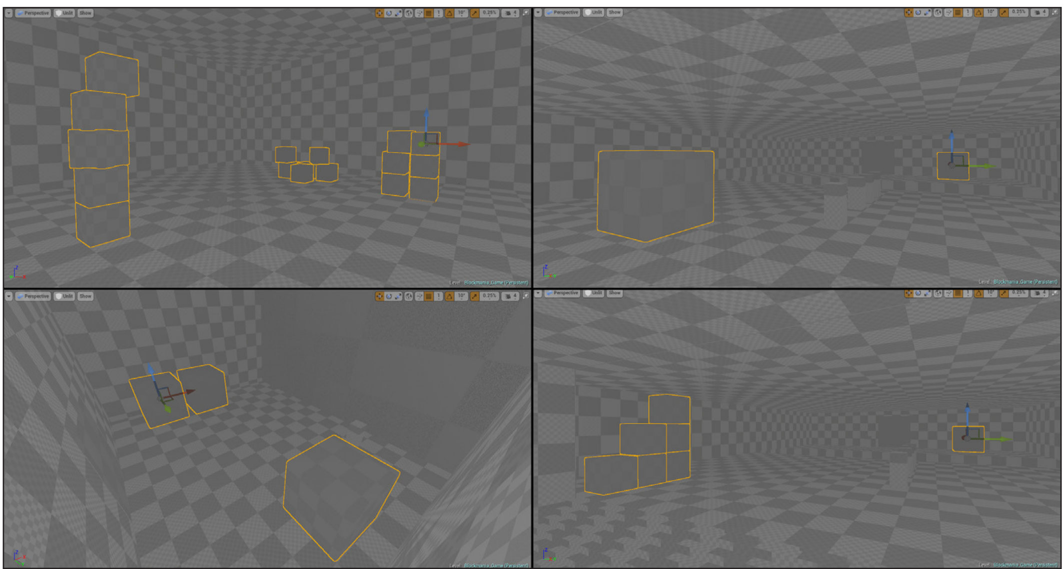
Next, let's place the big doors that the player will encounter in the middle of the rooms. Again, take the **TemplateCube_Rounded** actor from the **Content Browser** and place it in the second room. Set its dimensions so that it blocks the character from moving to the other side of the room. Name this `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

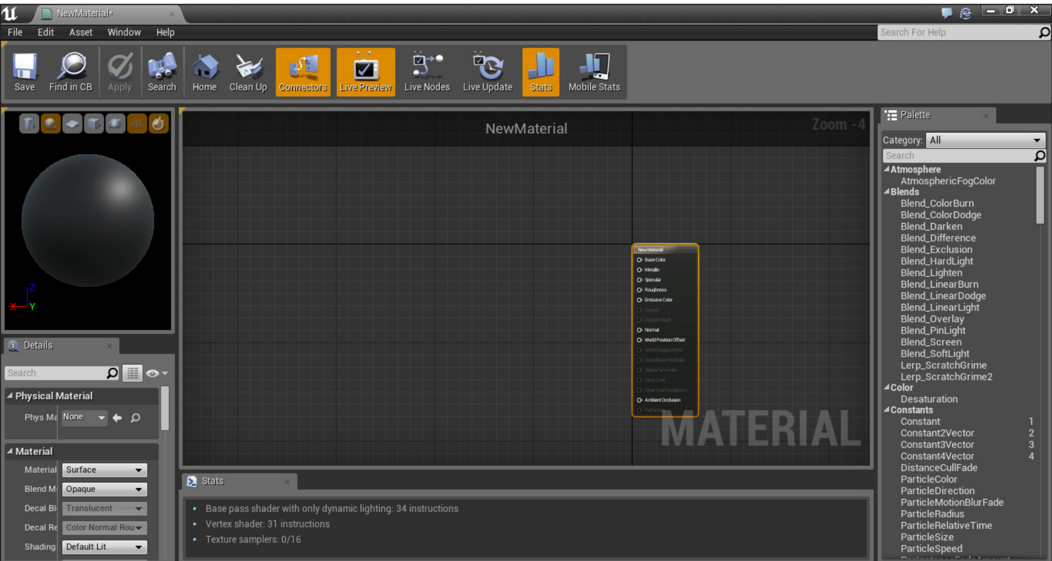
In UE4, before you can apply textures to an object, you first have to create materials. A material is how an object will be rendered in a game, it is a collection of shaders containing many properties that can be applied to objects and rendered in the game. In more technical terms, when light from a 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.

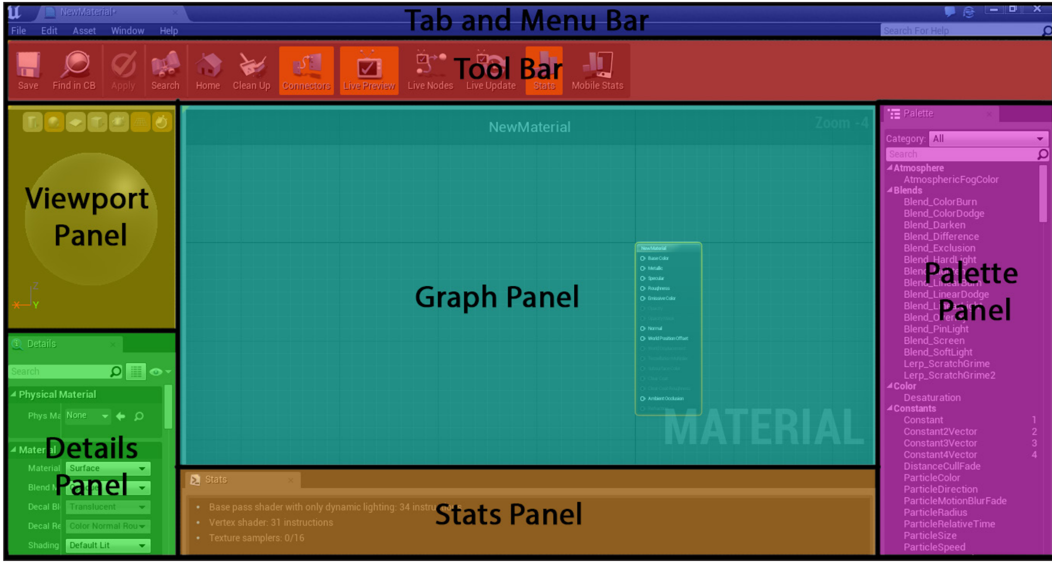
A material is created in what is known as a Material Editor. So, before we start creating our own materials, let's first talk about it and its user interface.

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:

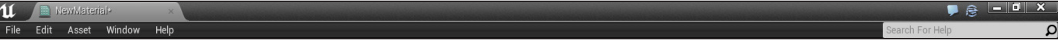


Let's first look at the editor itself and its user interface. As with the Editor, we will divide the Material Editor user interface into sections and go through them individually.



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.

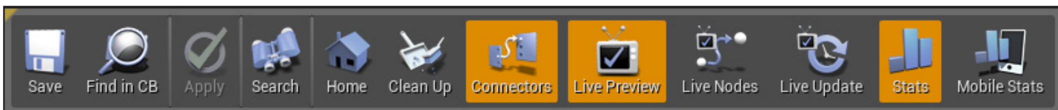


Below the tab bar, we have the menu bar. As with the menu bar in the Editor, it offers all of the general commands and actions. They are described as follows:

- **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.

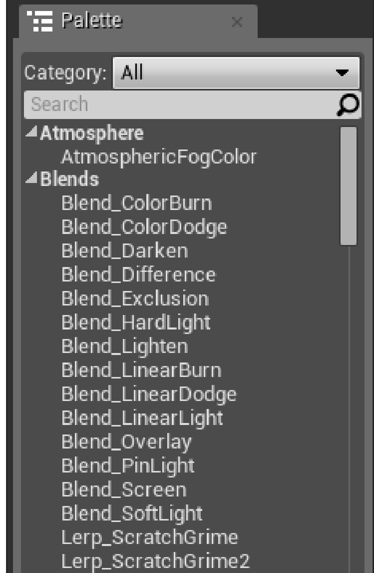


- **Save**: Starting from the left, represented by a floppy disk, is the **Save** button. This is to save any modifications you have made to the material.
- **Find in CB**: **Find in CB** (or Content Browser), depicted by a magnifying glass, locates and highlights the current material you have opened 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.

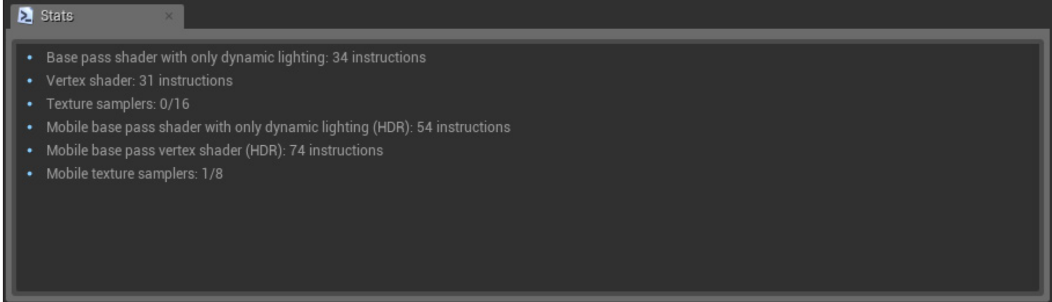


All of the functions, expressions, and so on are categorically listed here. You have the category on the top and you can filter what type of nodes you want displayed using it.

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

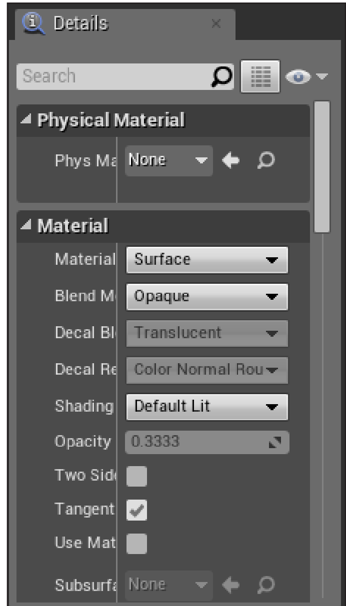
As mentioned previously, the **Stats** panel displays all of the stats regarding the material you have created.



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.

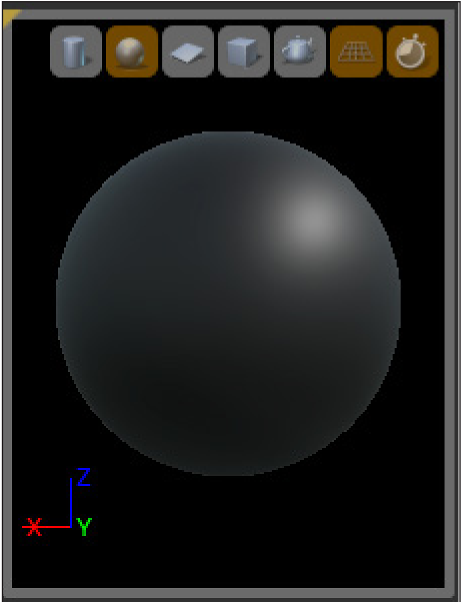


Normal materials, such as the ones you apply to actors, are **Surface** materials. If you want your material to be a **Decal** (which we will discuss in the next chapter), you can switch to **Decal**, and so on.

You can also set properties of any node that you have selected, in this panel.

The Viewport panel

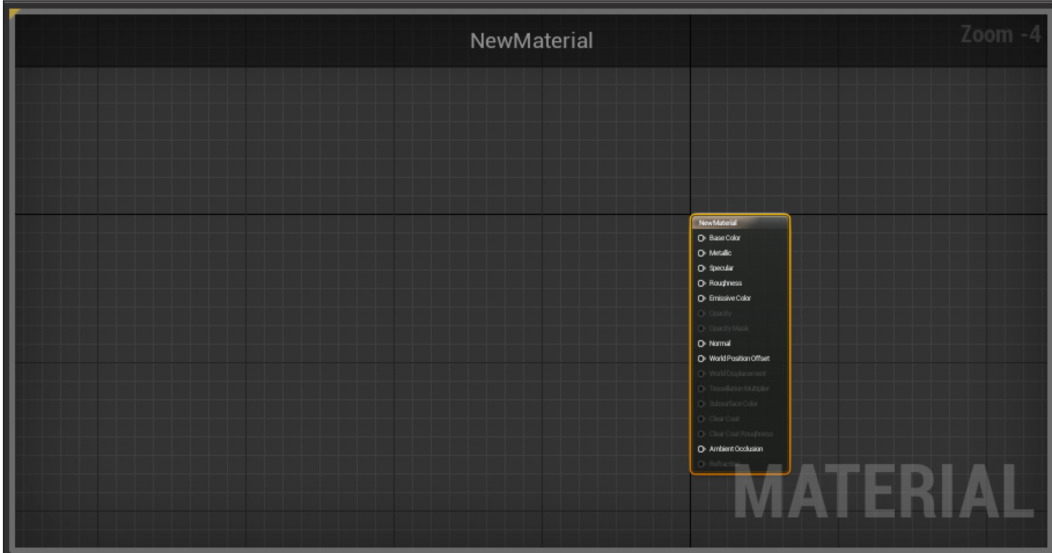
The **Viewport** panel is where you can preview the material you are creating. If the **Live Preview** option is enabled, any modifications you make will be updated in real time and previewed here.



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.

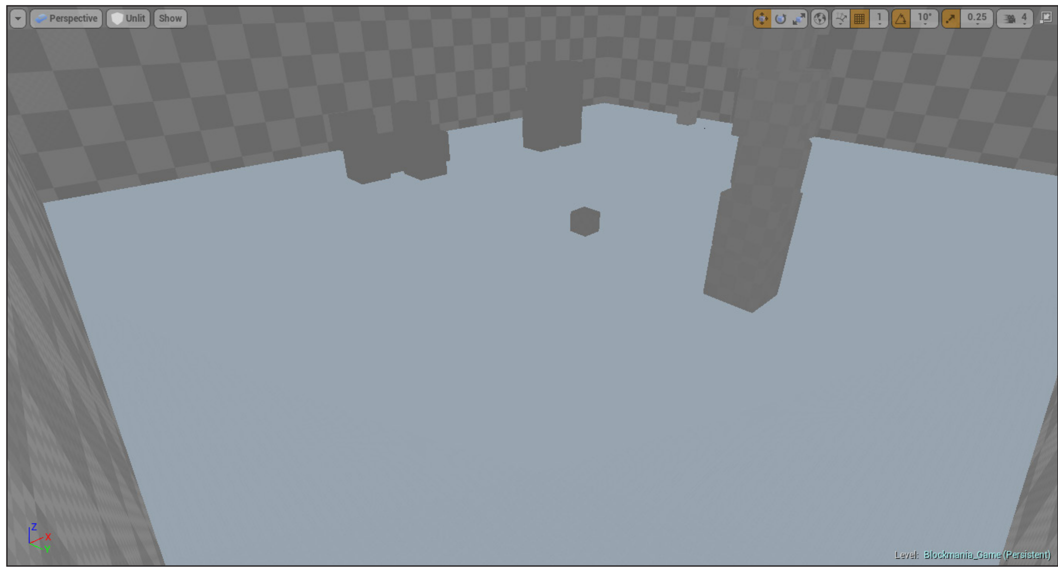
There are other inputs as well, but these are the main inputs that you should know for now.

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:

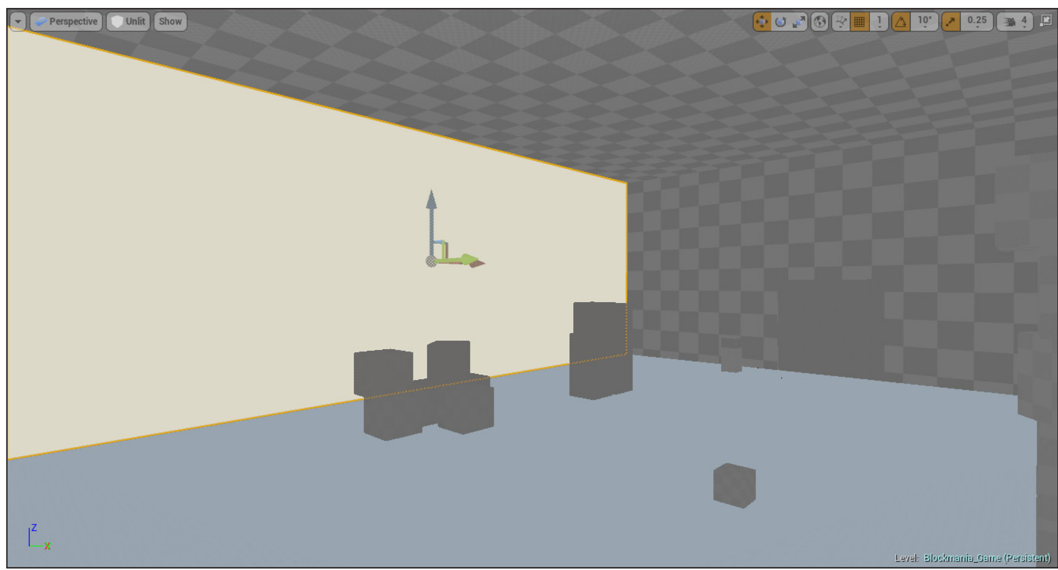
1. Select `M_Basic_Floor` from the Content Browser.
2. Drag it on to the floor surface.

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

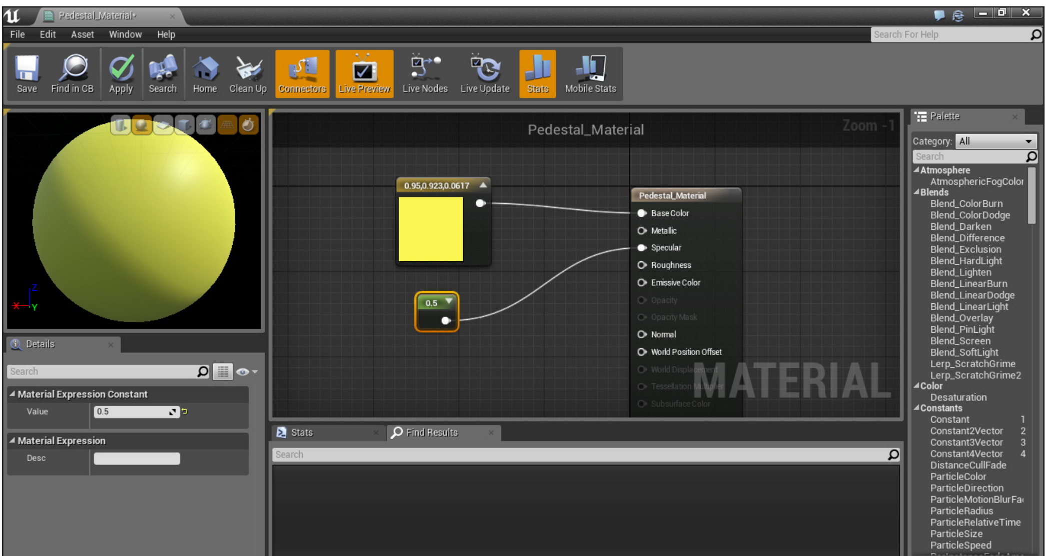
Now that we have covered the Material Editor's user interface and seen how to apply them onto objects, let's go on and create the materials for our level.

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.

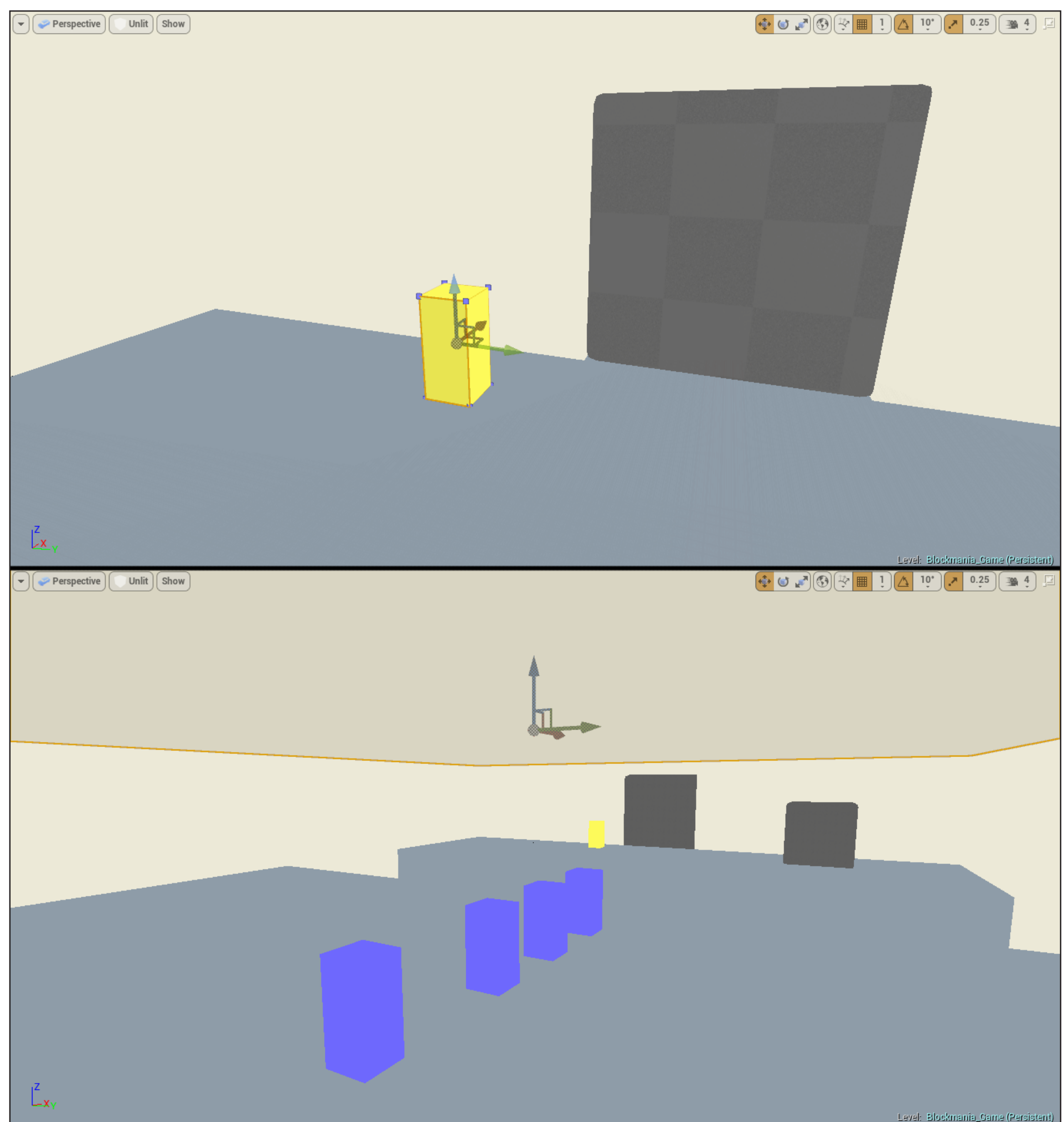
The first thing we are going to do is add something to the **Base Color** input. What we need to create is something called a **Constant3Vector** node. To create it, simply drag the node from the **Palette** panel and place it in the Graph panel. Another much quicker way of creating a **Constant3Vector** node is by holding down *3* 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 *1* 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.

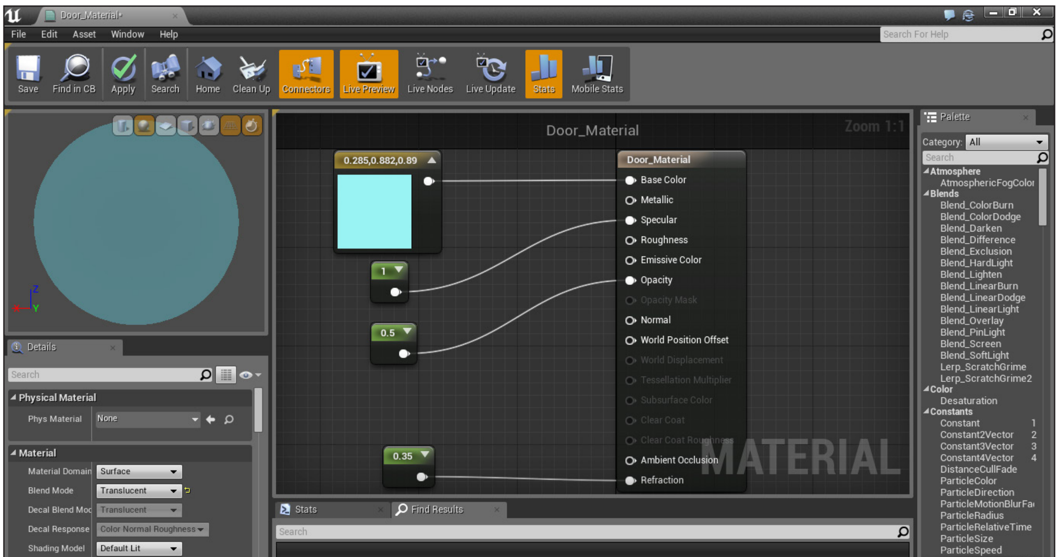


Doors

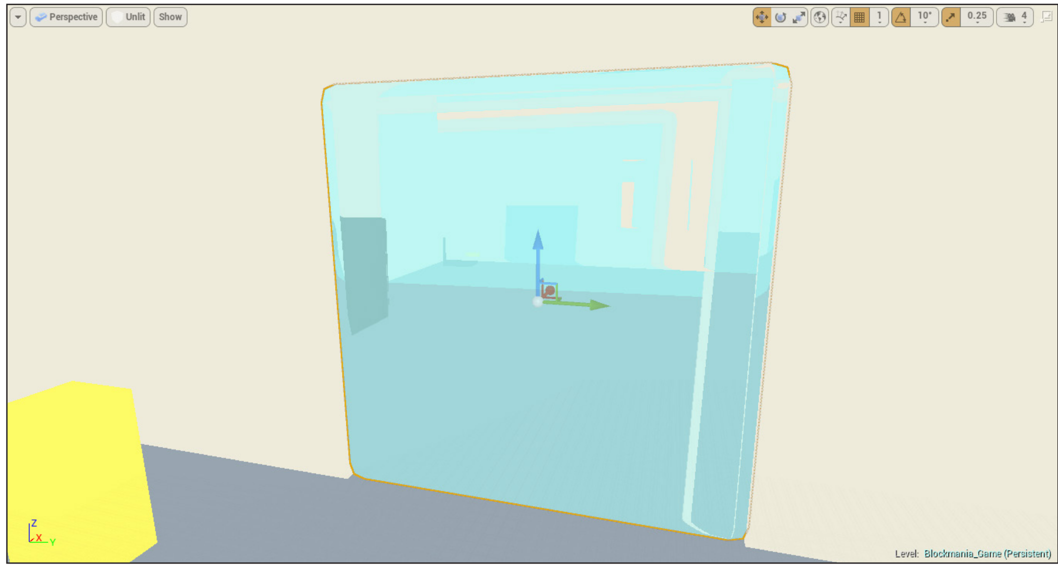
For the doors, we are going to do pretty much the same thing we did when we created the materials for the pedestals. The only difference here is that we will make the doors a bit see-through, and make the material glassy. With that said, create a material for the door, and name it **Door_Material**.

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.

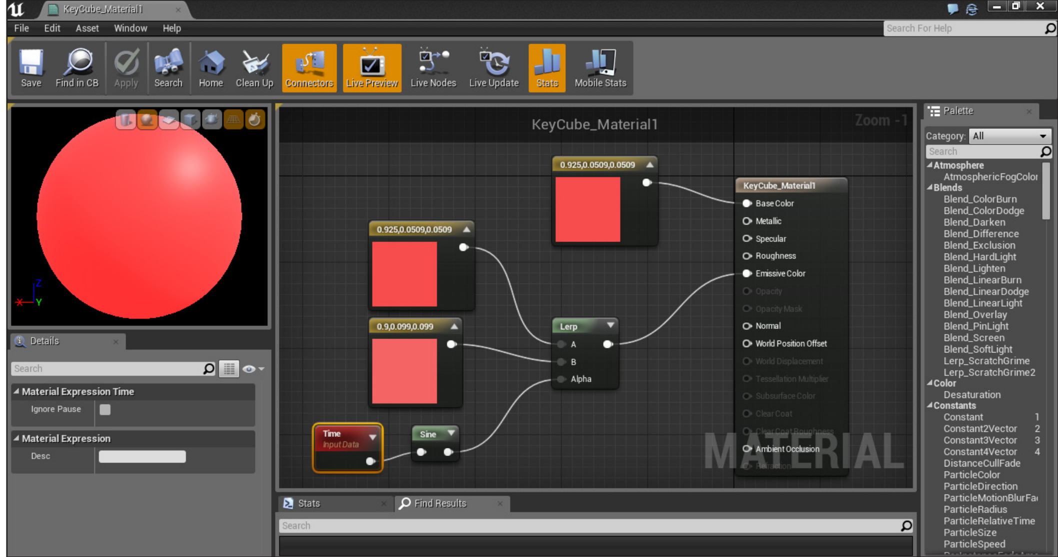


Key Cubes

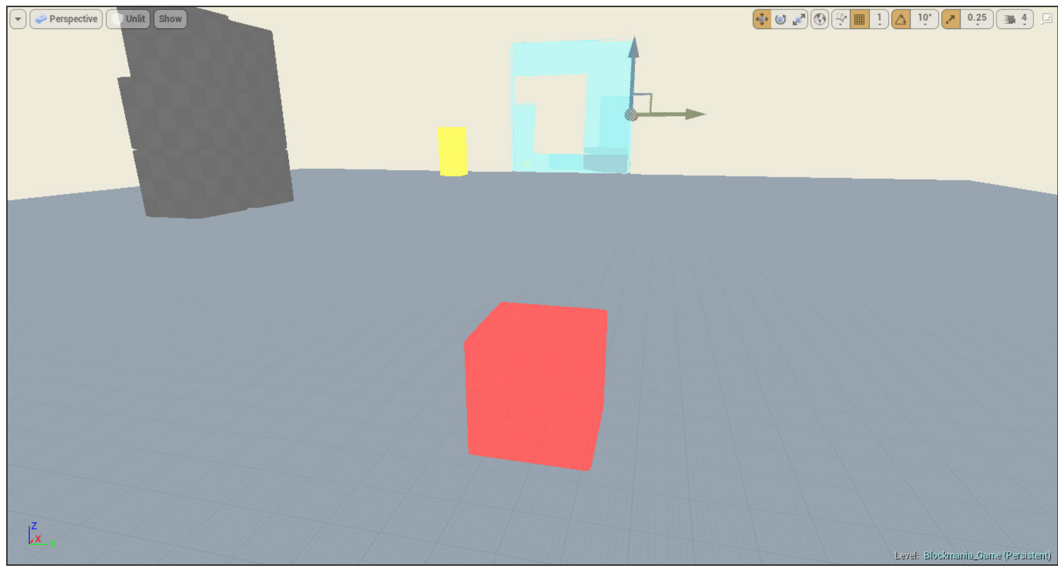
Let's make the set of materials a bit interesting. We are going to make it red. Also, we are going to make it flash so as to grab the player's attention. Create a material and name it **KeyCube_Material**.

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.



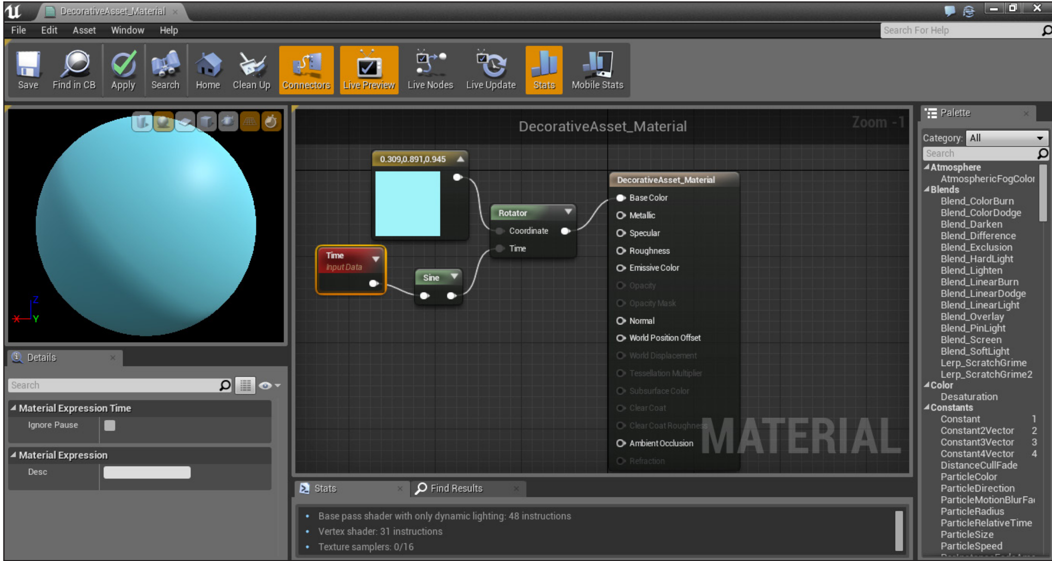
And there you have it. We now have a flashing material for our main key cube. Apply it to all the key cubes.



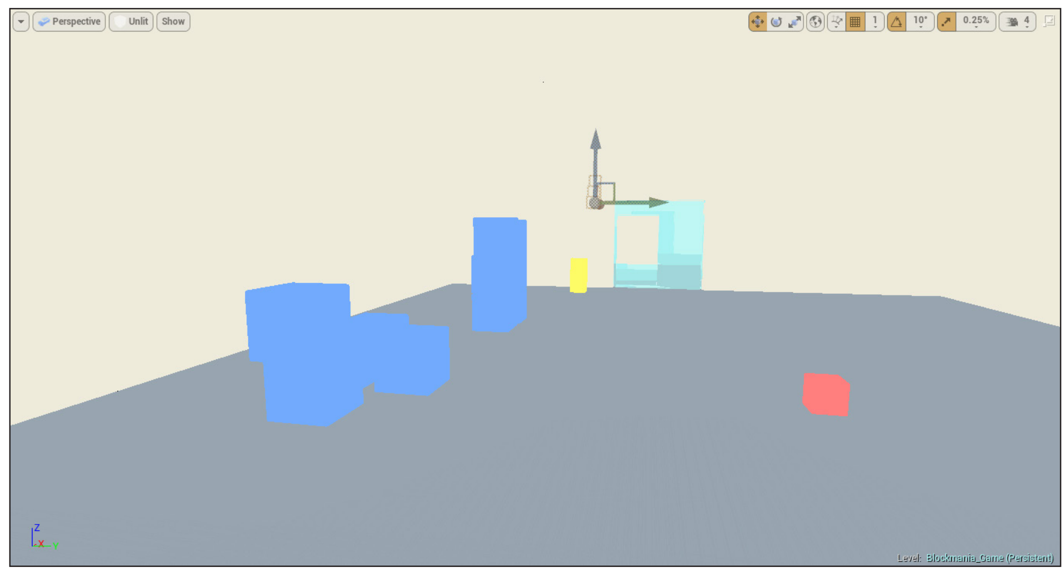
Decorative assets

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.



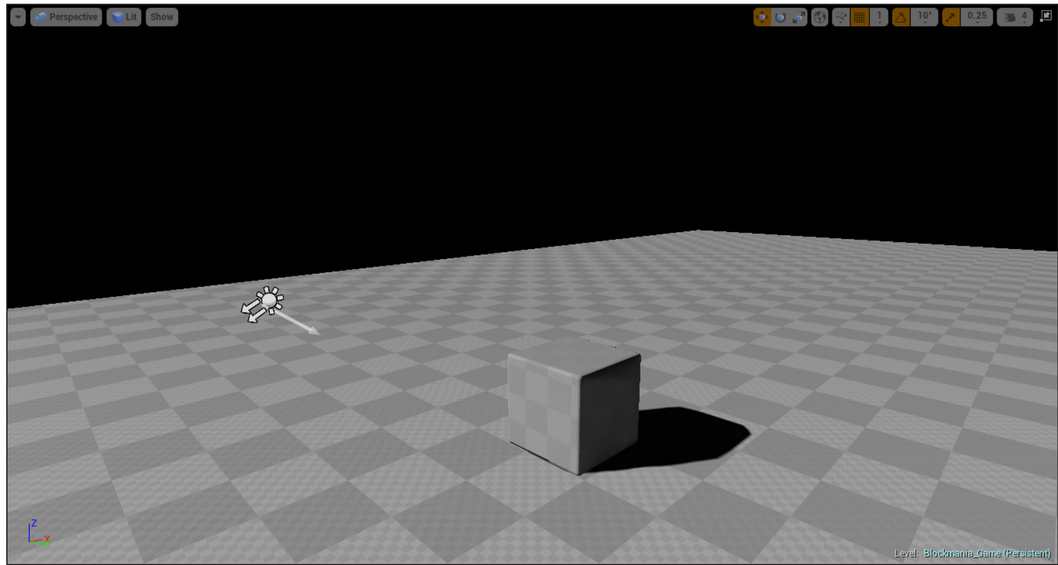
If you test out your level, you would notice that you cannot see anything. This is because our level does not have any lights, therefore nothing is rendered and all you see is a black screen. Let's now move on to lighting.

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:

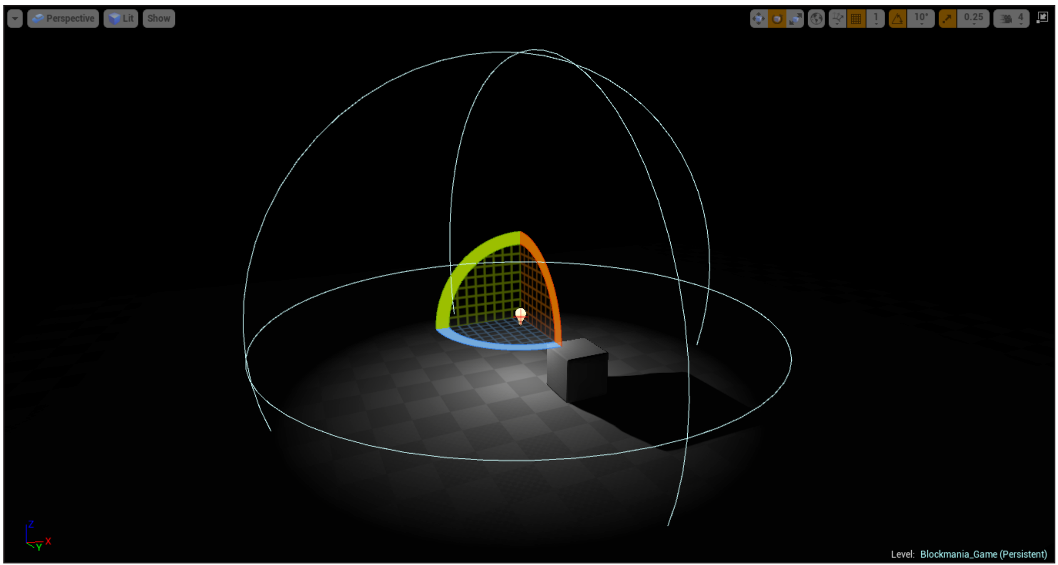
- **Directional light**: Directional light is the ideal type of light when you have an outdoor scene and want to simulate light coming from the sun, since directional light simulates light coming from a source infinitely far away. The shadows formed by directional lights are parallel.

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.



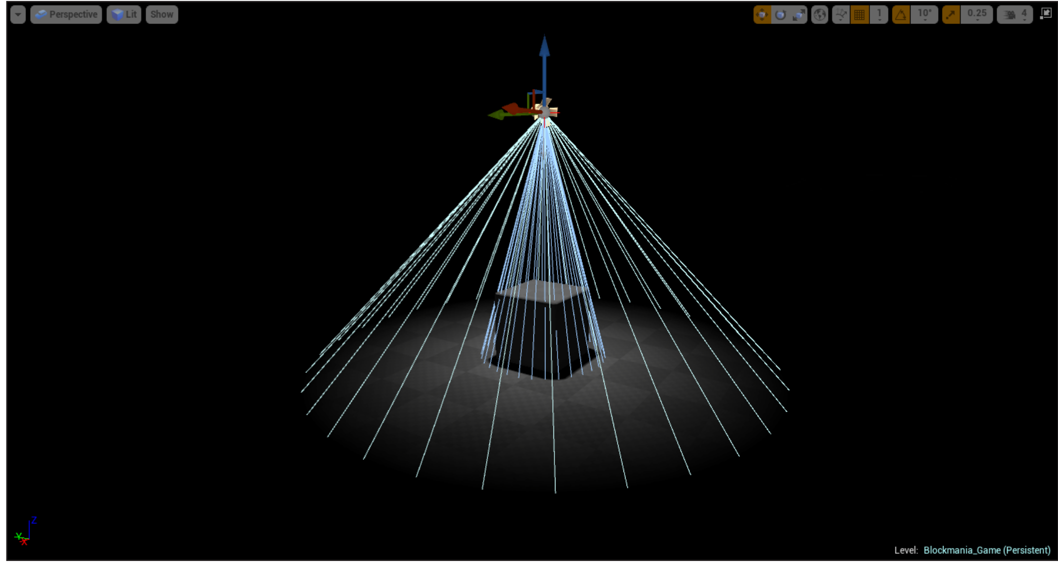
The preceding screenshot shows directional light and how it works in the level. Its icon is depicted by a sun with two parallel arrows coming out of it. The other single arrow shows the direction from where the light is coming. You can set the direction by using the rotation tool.

- **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.



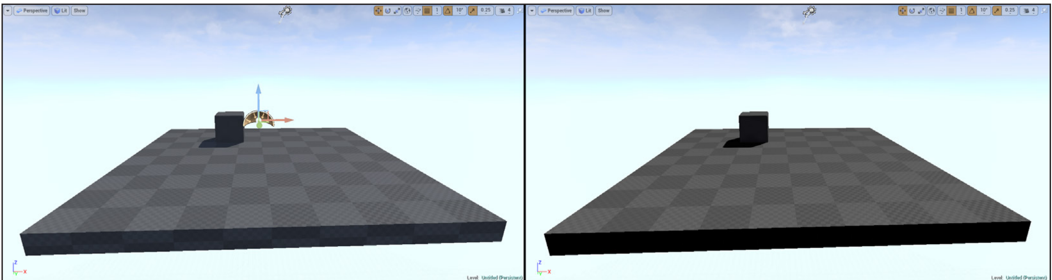
As you can see, the point light actor is represented by a light bulb. The sphere you see around it is the *Attenuation Radius*. It is represented by a sphere wireframe, and shows the volume where the light source has direct influence. You can set things such as the intensity, the color, the attenuation 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.



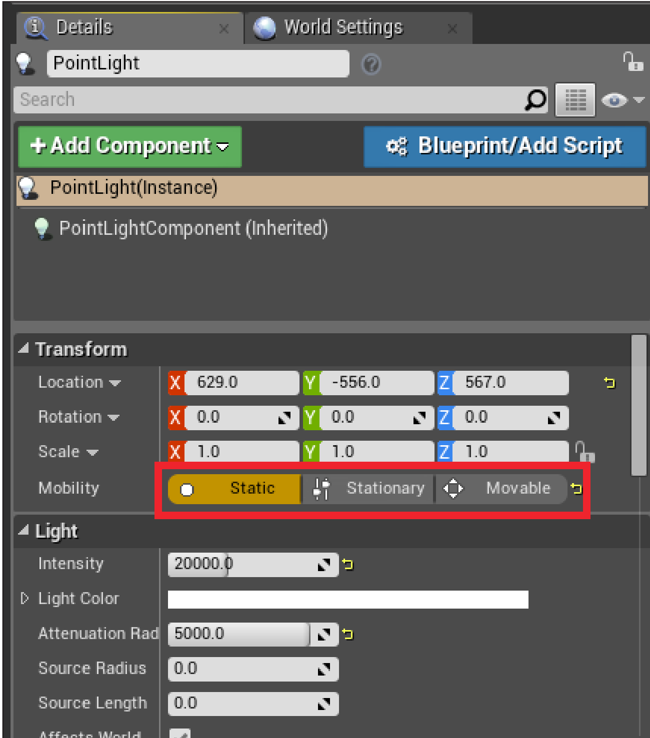
Sky light is quite subtle in terms of effects. With that in mind, in the preceding screenshot, the left is a scene with sky light and to the right, without Sky light. You can see the effect of Sky light when the two are juxtaposed.

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

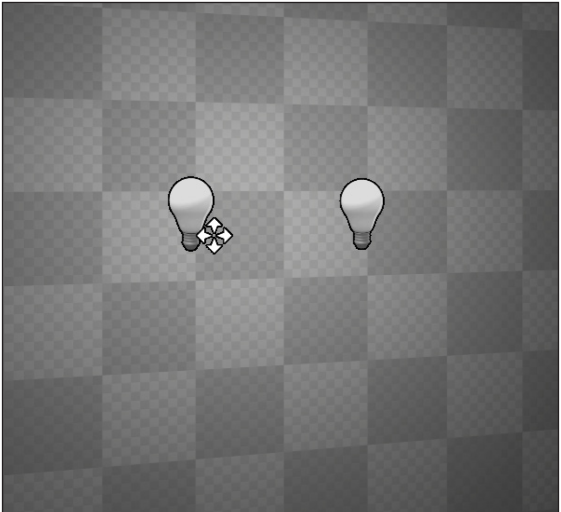
All the types of lights have three types of mobility. There are certain settings of light characters that you can modify while the game is running, should you require to. The mobility section allows you to do just that.

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.

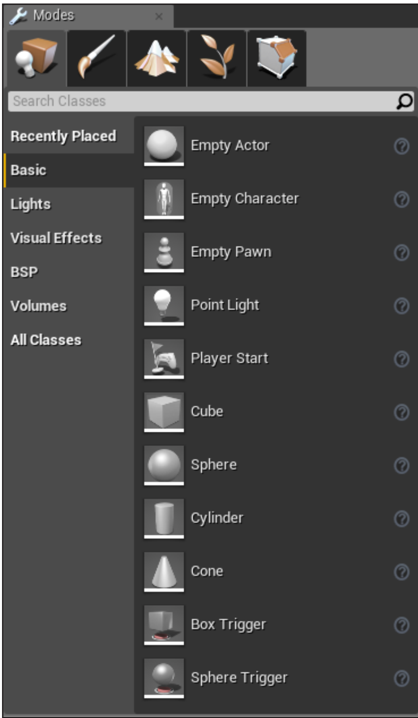
4
Using Actors, Classes, and Volumes

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`).

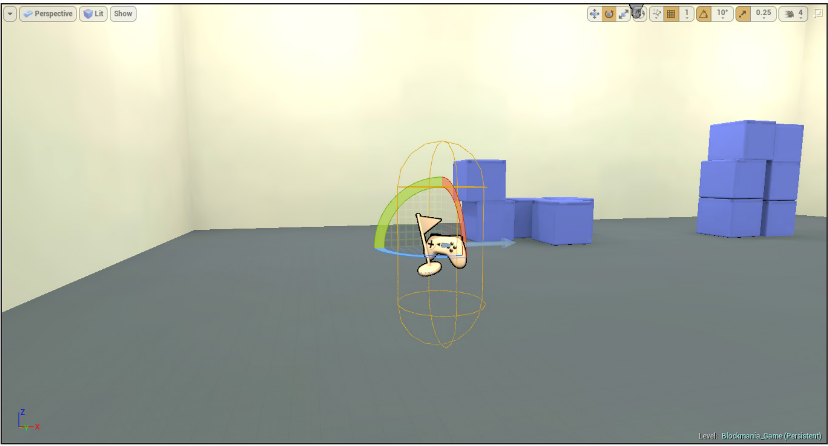
- **Point Light**: As mentioned in the previous chapter, **Point Light** is the most basic and most widely used source of lighting. You can also add the Point Light actor from here.
- **Cube**: This adds a cube primitive (static mesh) to the game.
- **Sphere**: This adds a sphere primitive to the game.
- **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

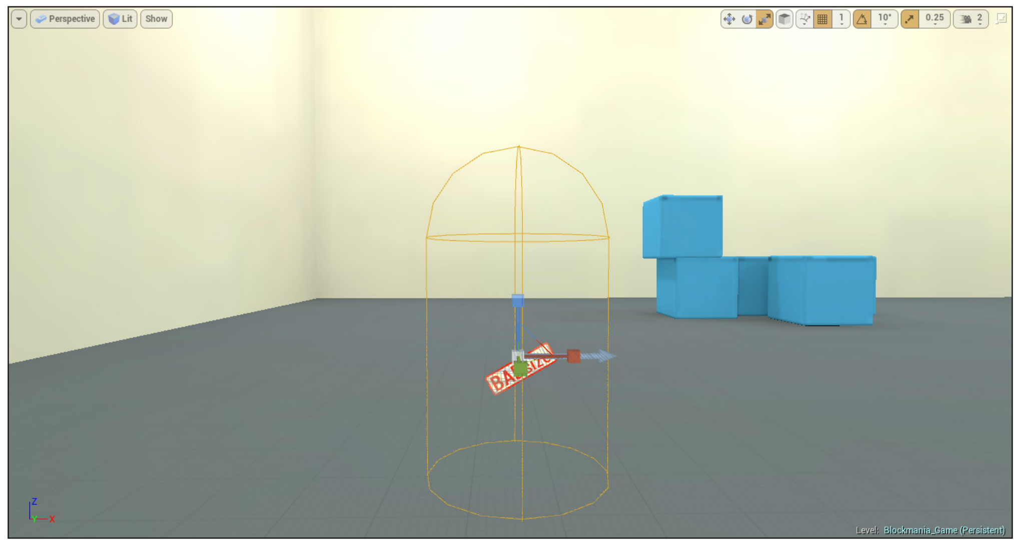
Now that you are acquainted with the basic classes, let us go ahead and add some to our level.

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

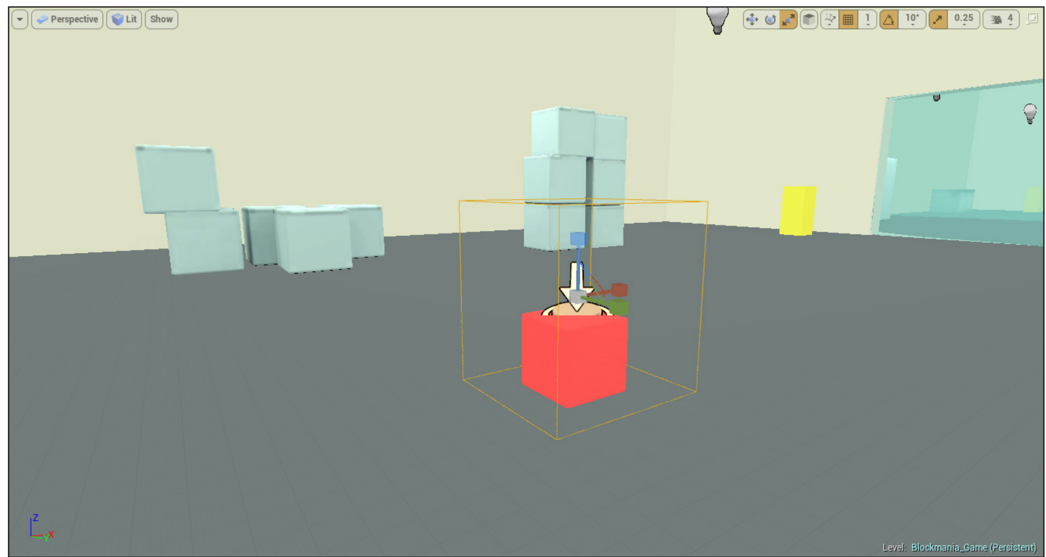
Next, we are going to add triggers. We are going to keep things simple by using only Box triggers. To place a Box trigger, simply drag it from the **Modes** panel and place it on the level. What we will do with the triggers we will cover in the next chapter. For now, you can simply place the triggers 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

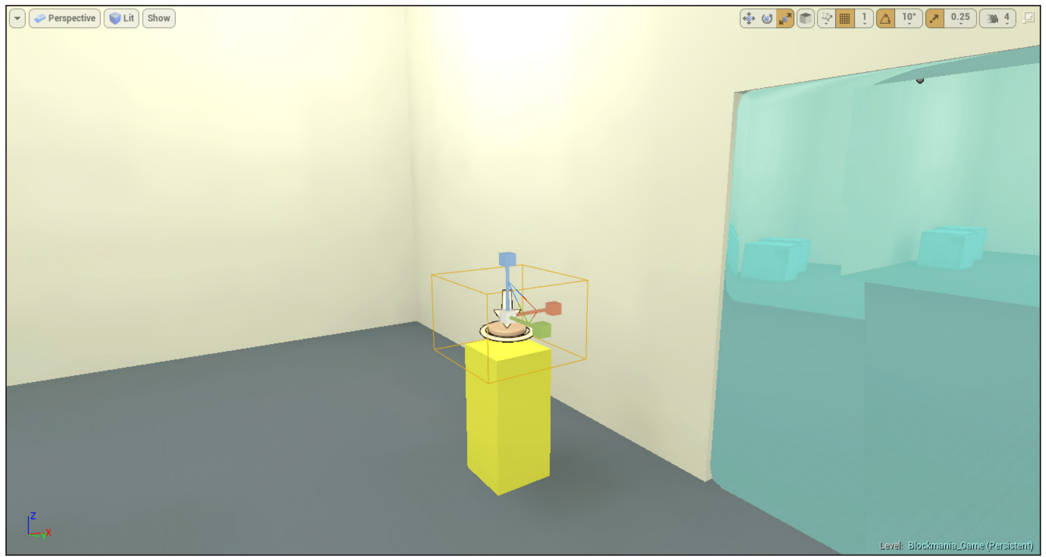
In the first room, place the first trigger near the key cube. This trigger will be used to interact with the key cube. The placement and dimensions of this trigger (or any trigger for that matter) are important, since they determine how far the player can interact with the key cube. For example, if the 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.

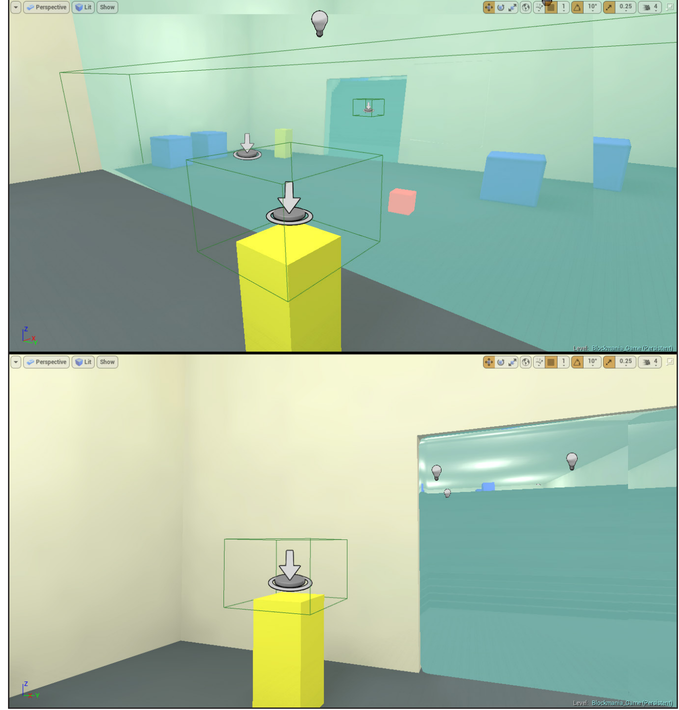


To check the placement of the trigger and whether its position and dimensions are correctly set, you can first unhide the trigger actor by unchecking the **Actor Hidden in Game** option—found in the trigger's **Details** panel and then play the level. For now, this will do in terms of trigger placement. 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.



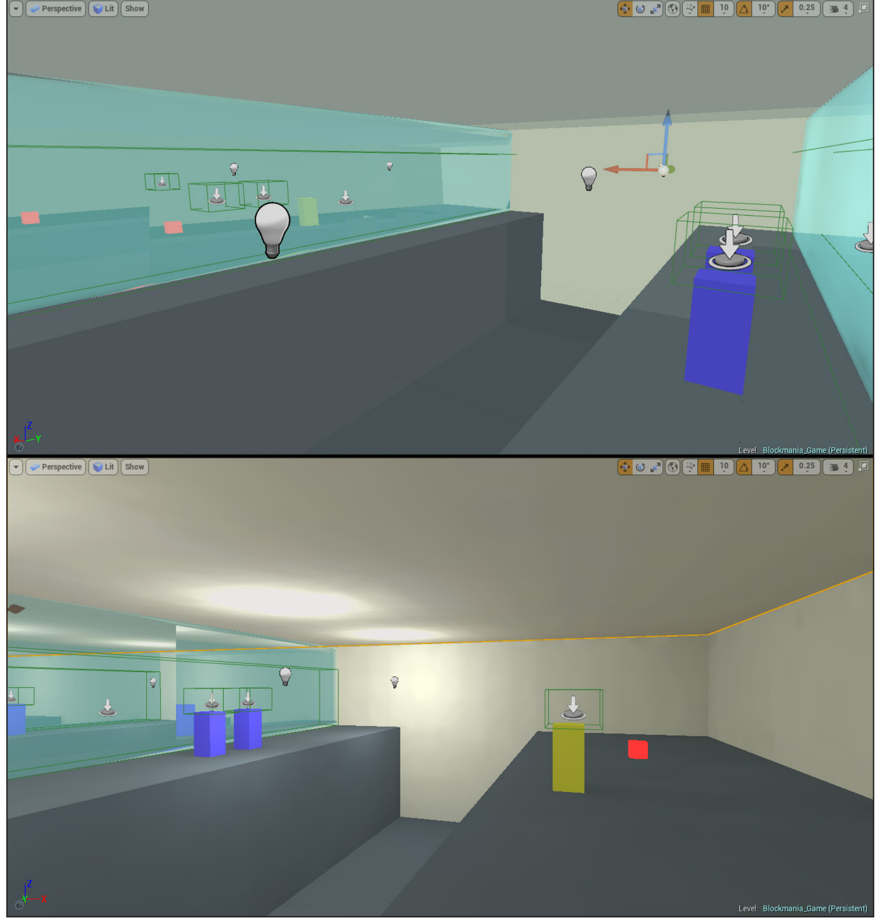
We are not going to place any triggers on the key cubes as we did in the previous room. The reason behind this will be explained in the next chapter.

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.

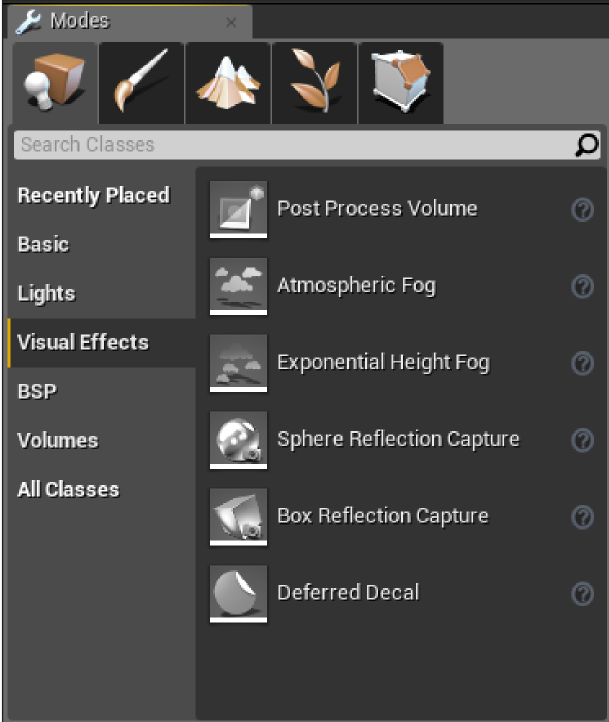


We have now placed most of the triggers we need for our game. We will add more later. And with that, we are done placing Basic classes into our level. Let us now move on to the Visual Effects class and see how it affects our game.

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.



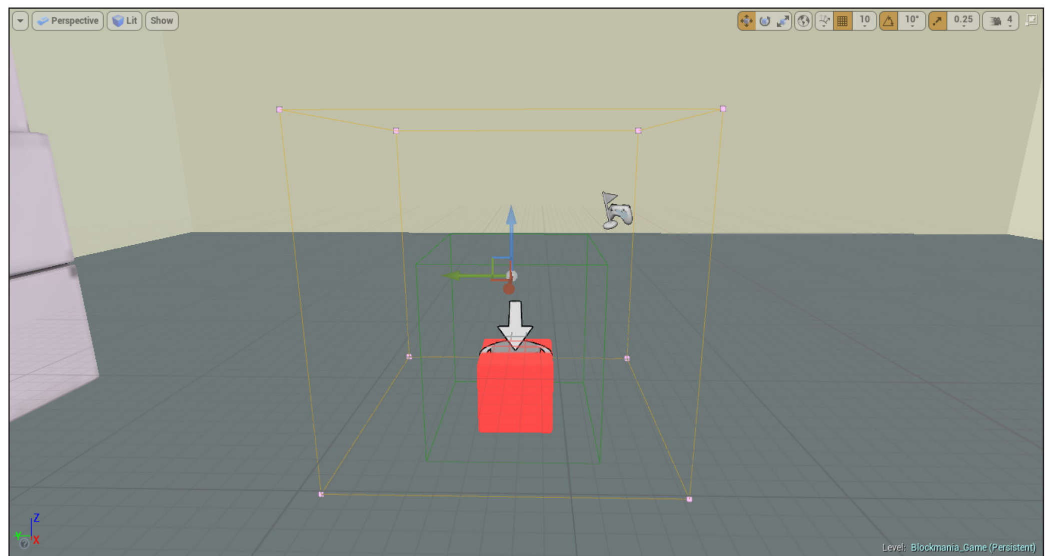
There are various actors in Visual Effects classes:

- **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.

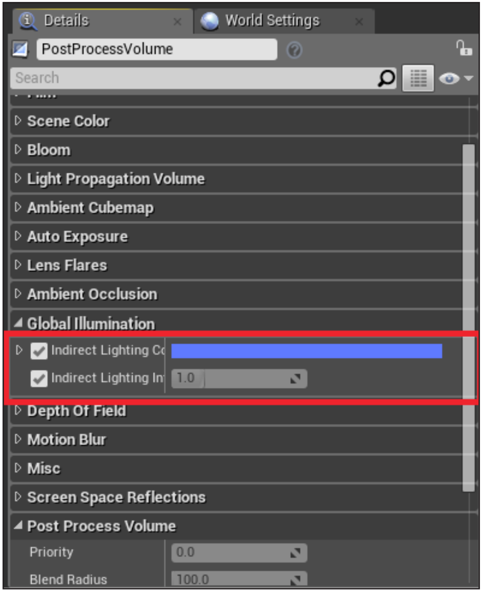
Adding Visual Effect actors to the game – Post Process Volume

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.



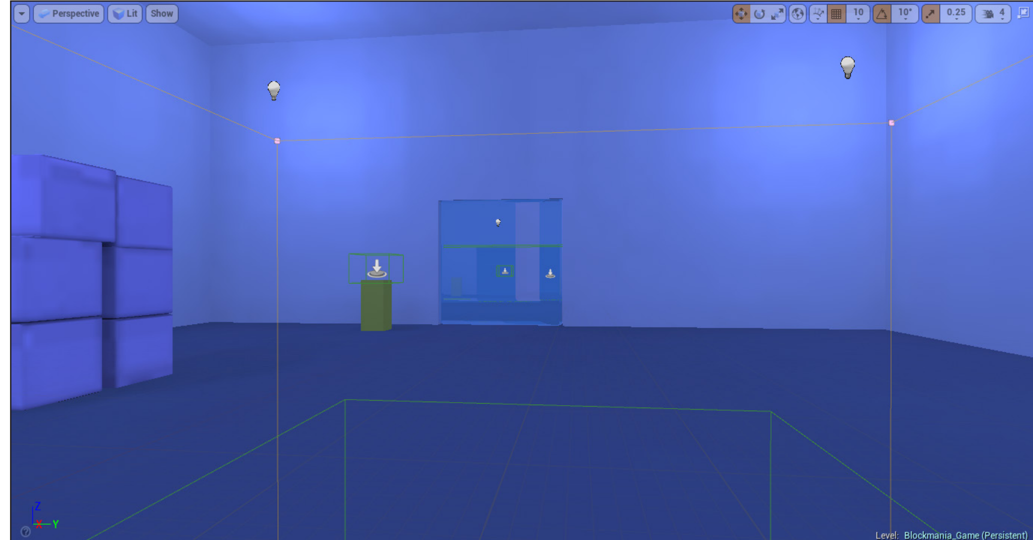
After the Post Process Volume has been placed, let us set some properties in the **Details** panel. In this panel, you will see quite a lot of options. All of them are categorized based on the type of effect they create. One thing to note is that some of the effects are not available on mobile.

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:

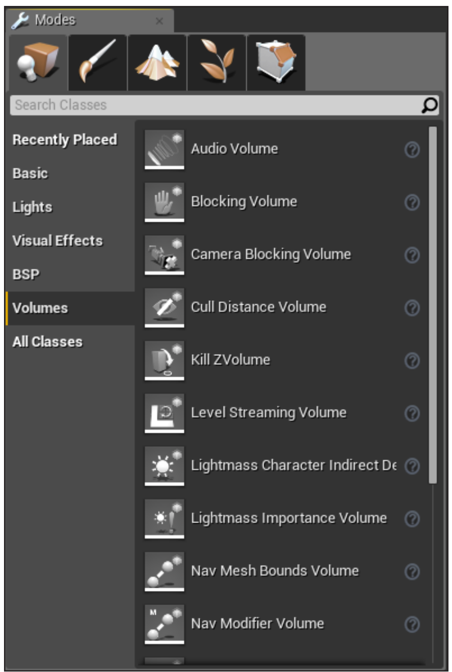


Now, if you were to test the game and walk inside the cube, everything would turn blue and remain blue unless you stepped out of the volume. We do not want that. We only want the screen to turn blue for a brief moment—when the player picks up the key cube—and then fade away. We need to change another 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 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 that while trigger actors come in a predefined shape, you can alter the shape of a trigger Volume using the **Edit Geometry** Mode.

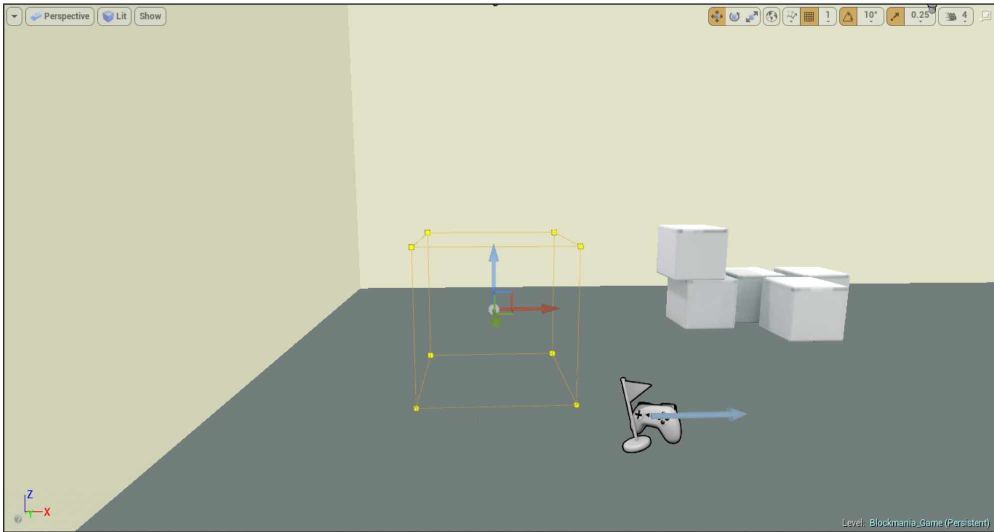
One more thing that you should know about volumes is that just like with BSP brushes, you can edit their shapes to create your own custom shaped volume. The way to edit is the same as that of BSP brushes. In the **Modes** panel, clicking on the **Edit Geometry** mode will switch to the editing mode. Once done, you can click on any volume you want in order to change and make the desired changes.

Adding Volumes to the game

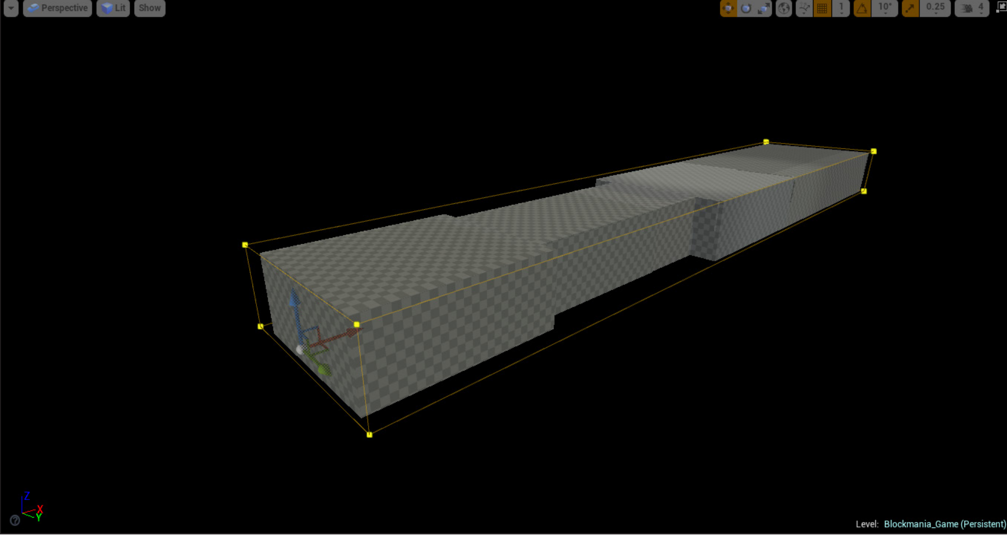
Now that we know what Volumes are and the types of Volumes available, let us go ahead and add a few of them to our level.

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

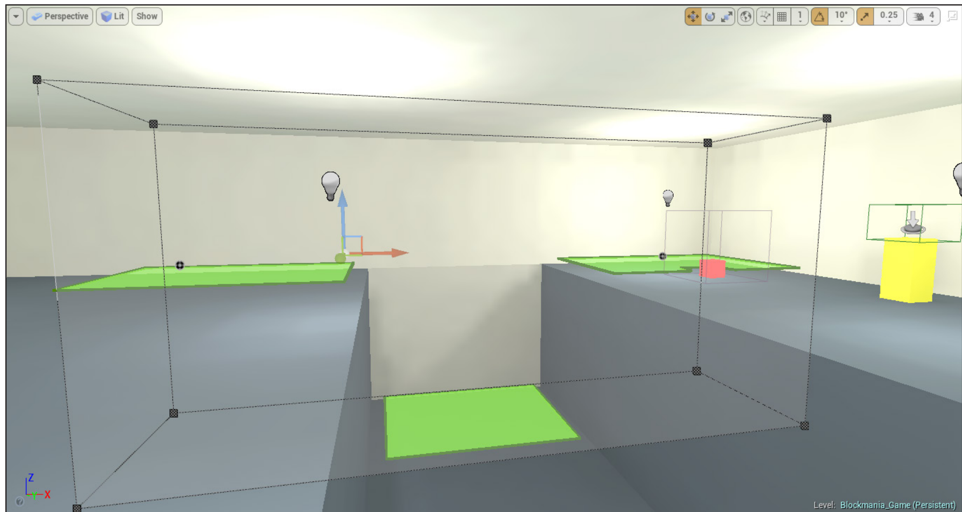
Next, add the **Nav Mesh Bounds Volume**. If you wish to have an AI character in your game, a **Nav Mesh Bounds Volume** is an important component. As mentioned previously, this volume is basically one within which the AI character moves and interacts with the world. When designing your game, you should 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.

Room 3

Place the **Nav Mesh Bounds Volume** on and near the pit, opposite the pedestals with the switches.

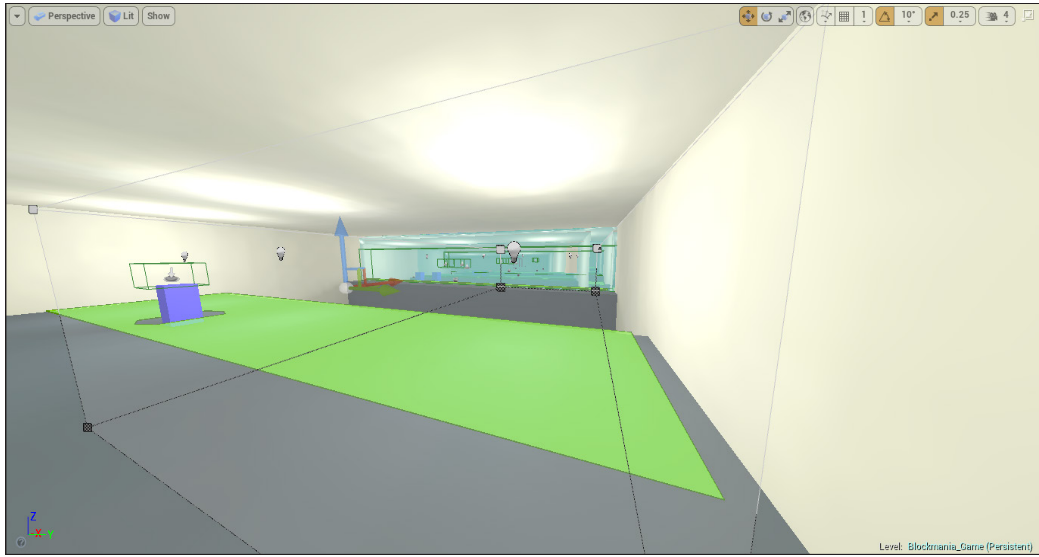
When placing this volume, it is advisable to turn on **Navigation** in the **Show** menu. In the **Viewport Toolbar**, click on **Show** and check the **Navigation option** to have it appear in the Viewport. Once toggled, you will see that any surface, actor, or any other physical object (or parts of 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

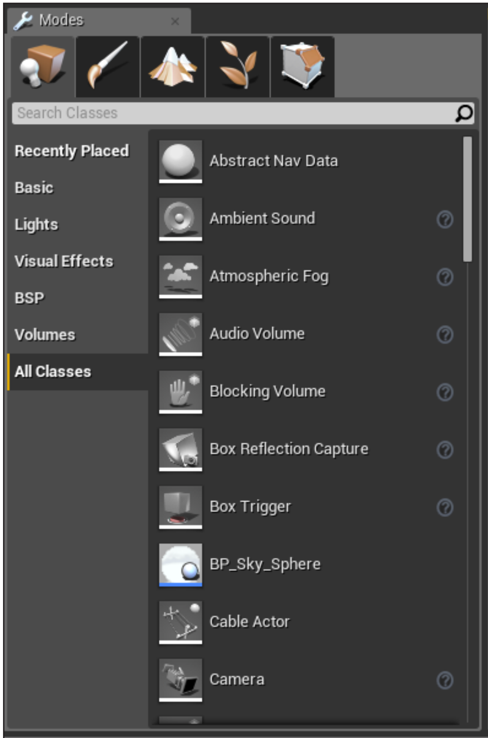
In room 4, we are going to need a big Nav Mesh Bounds Volume, one that covers almost the whole room. Place the Volume and move its sides using the Geometry Editing mode. After setting the dimensions, it should look something like the following screenshot:



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.

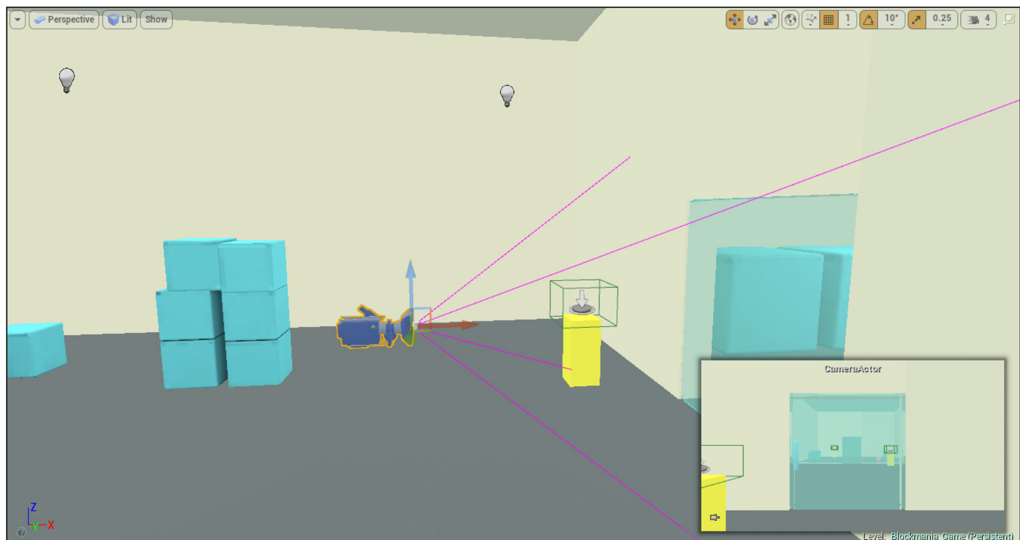
Adding actors from All Classes

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.

> 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.

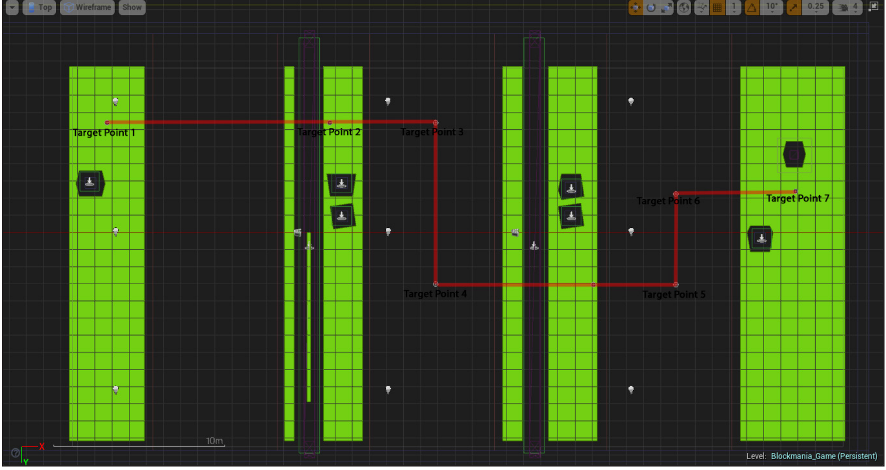
Room 3

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.

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:

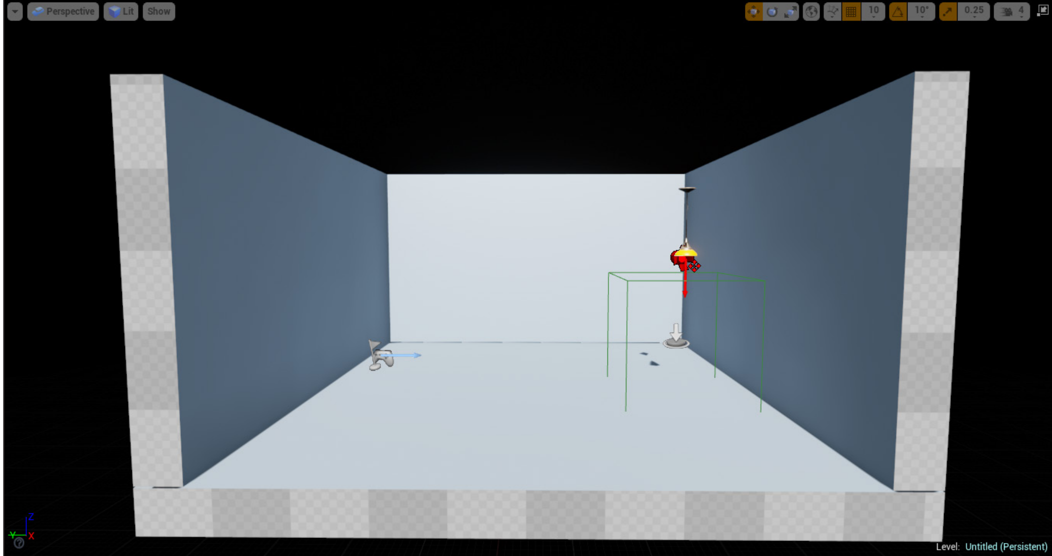
- What Blueprints are and how they work
- What Level Blueprints are and how to script using Level Blueprints
- Level Blueprint user interface
- What a Blueprint class is and how to use it in the game
- How to Script basic AI

How Blueprint works

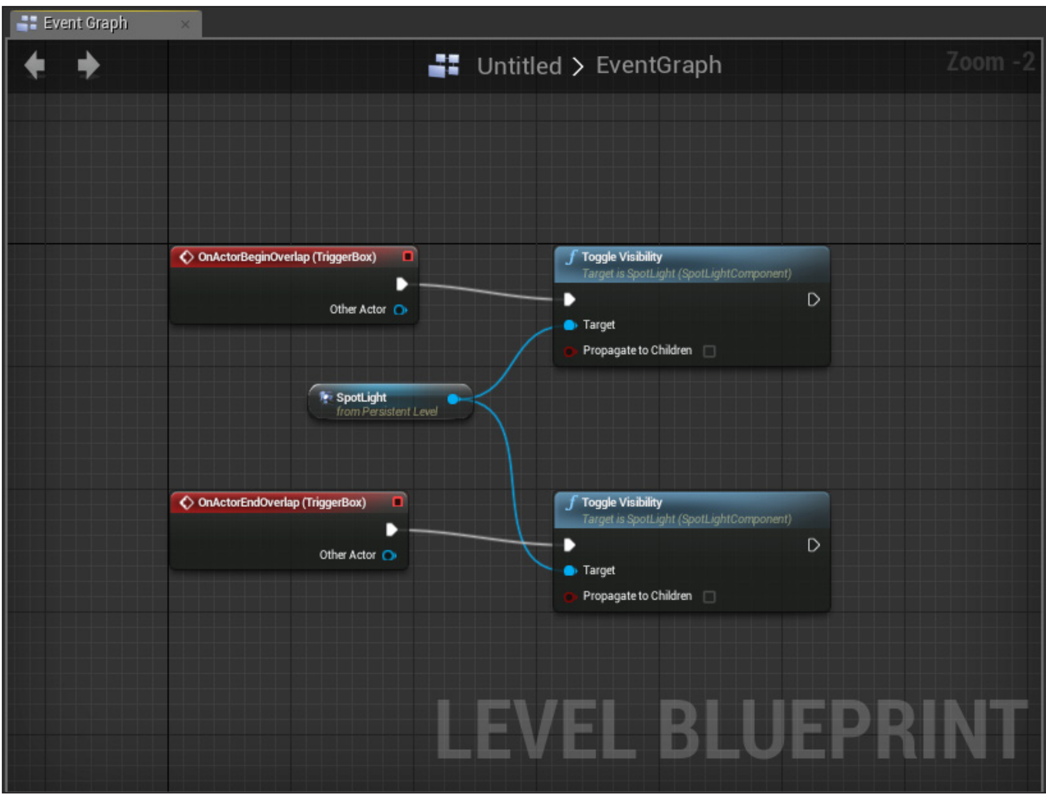
Scripting in Blueprint is similar to creating a flowchart. At your disposal are various nodes, which you can connect to create an action sequence. In order to properly script in Blueprint, you should first know the logic behind the desired sequence. For example, say you want to implement a lamp that 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 notes. 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).

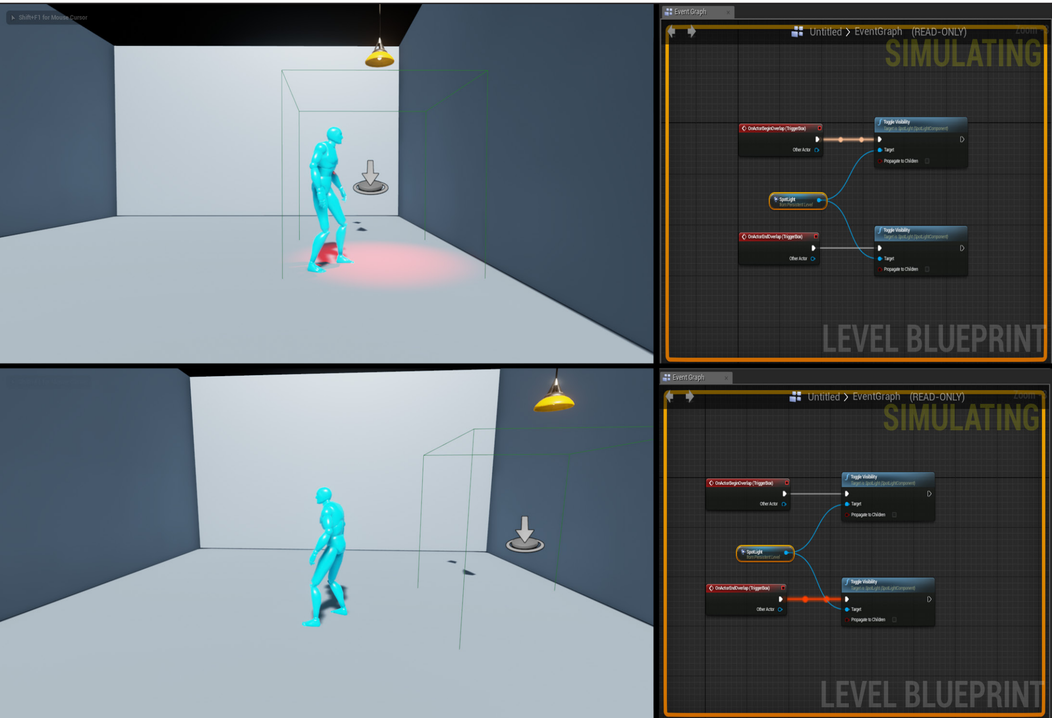
- **Function Nodes**: The nodes with the blue top are function nodes. They usually have a function symbol **f** at the top left corner, as you can see in the preceding screenshot. These nodes perform a specific action on an actor or player.

  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 on.

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.

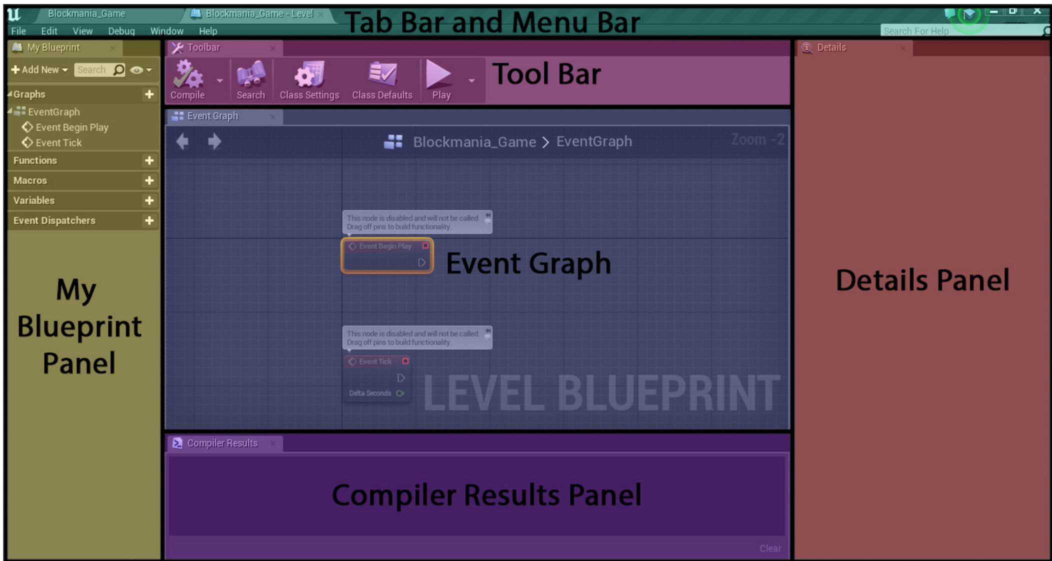
Level Blueprints is to UE4 what Kismet was to UE3. Each level or map that you create in your project file will have its own unique Level Blueprint. You can control everything level related using Level Blueprints, such as playing cutscenes, editing the properties of actors (visibility, location, and so 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.

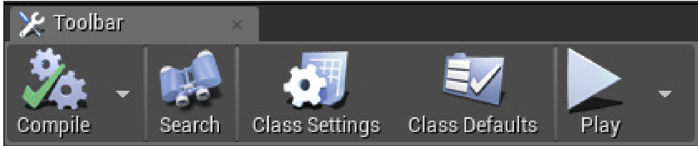
The menu bar is where you can access all the general commands and actions that you would need.

- **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:



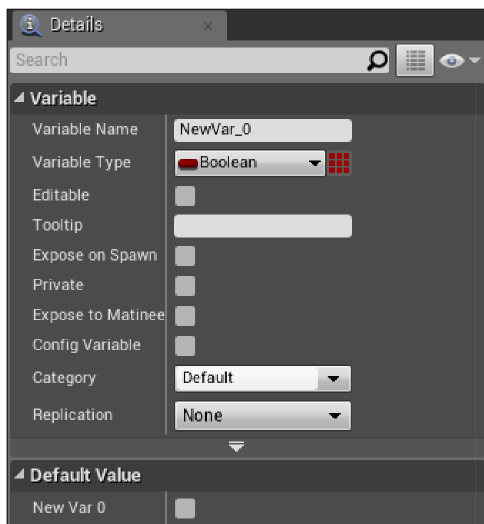
The most commonly used actions are explained as follows:

- **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.

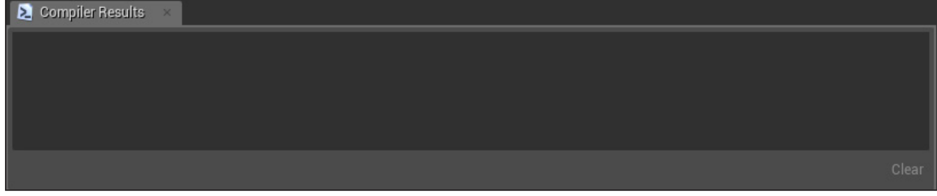
The Details panel

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

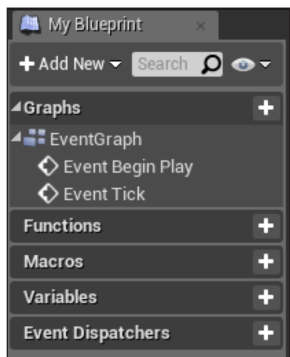
Anyone who has any programming knowledge will know what compilation is. It occurs when the code that you have written is converted from the language you wrote the code in into machine language that the computer understands, so that it can be executed. This is usually handled by a compiler. (If you do 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.

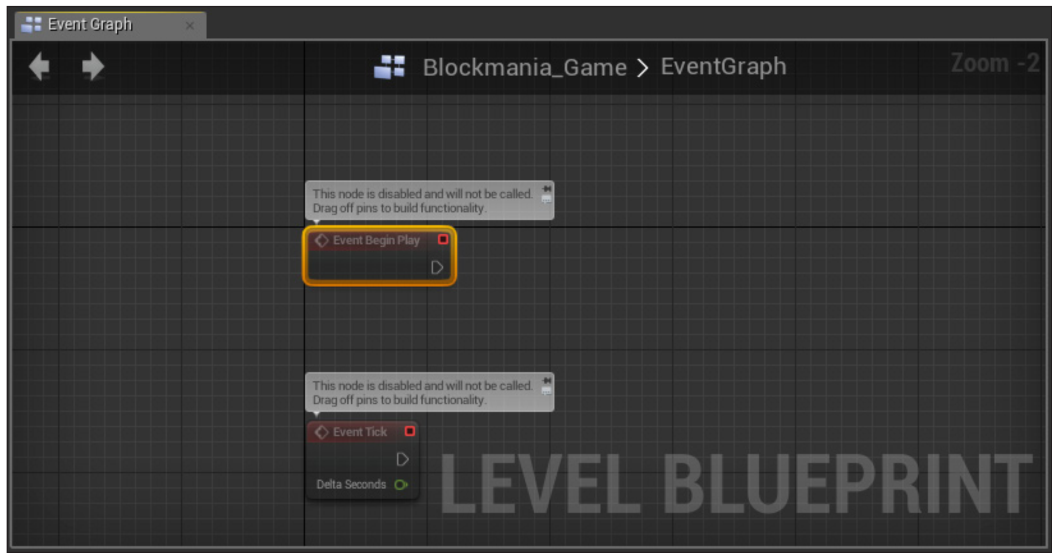
My Blueprint panel

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 our 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)

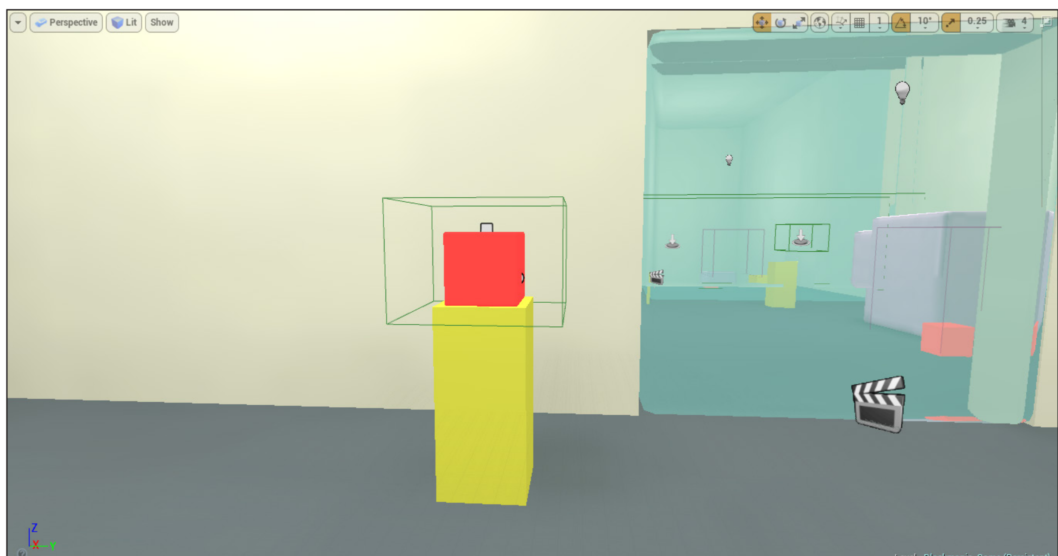
> Even though it is visual scripting, it can still get pretty messy when you script using Blueprints. So, to avoid confusion and keep everything organized, it is advisable to create comment boxes around your nodes.

Using Level Blueprint in the game

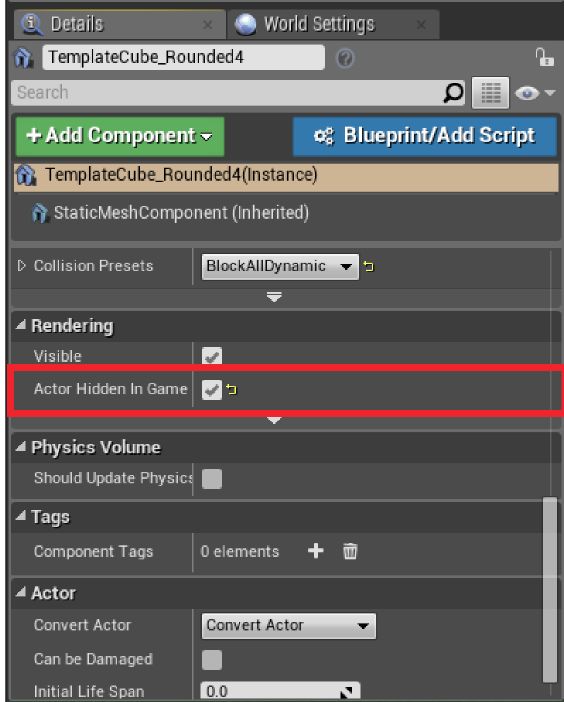
With all the basics out of the way, we can begin scripting our game. One thing you should know is that we might add more triggers and actors as we go along.

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.

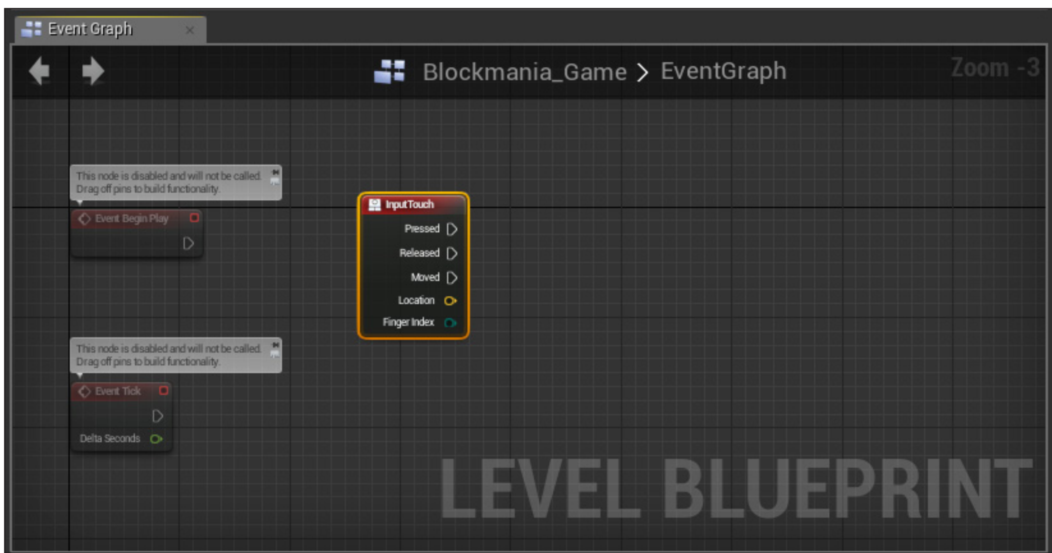


In its **Details** panel, under the **Rendering** section, tick the **Actor Hidden in Game** option. Doing so will hide the game from view during runtime.



Now, let's open the Level Blueprint. To do this, click on **Blueprints** in the **Viewport** toolbar and select **Open Level Blueprint**.

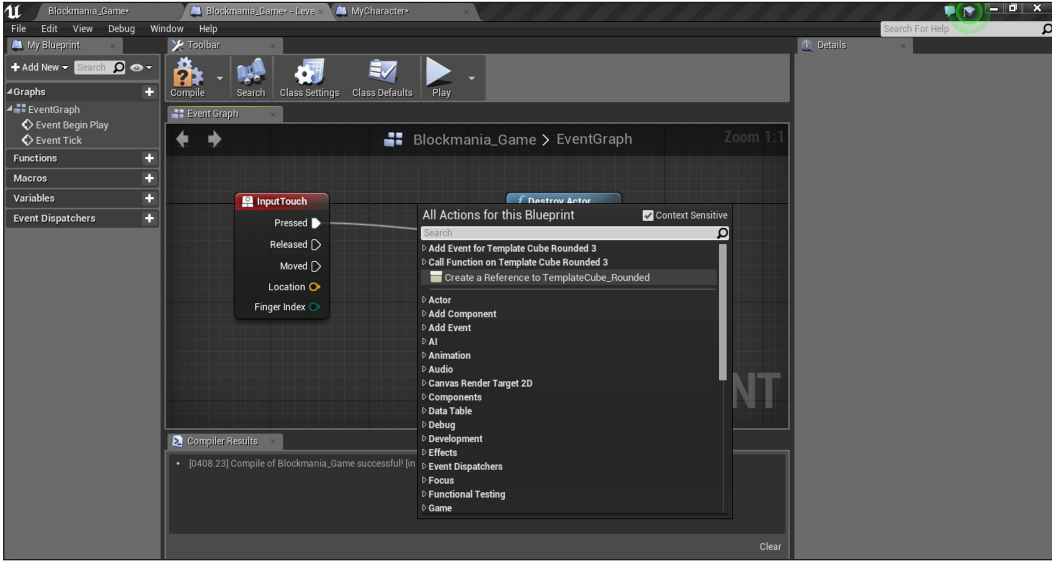
Now, the event in this case would be the player tapping on the screen, or in technical terms, providing a touch input. So in the **Event Graph** window, right-click to open the **Actions** menu and type in `Touch`. This should find and display the **Touch** event node. Click on it to add it to the **Event 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.

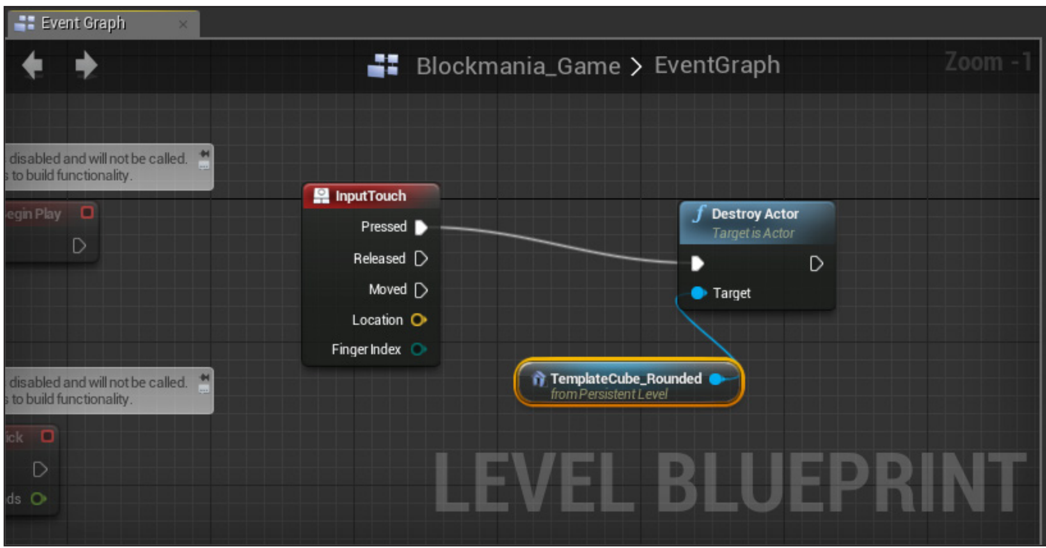
To remove the box from the scene, we are going to use the **Destroy Actor** function node. So, right-click anywhere in the **Graph Editor**, type in `Destroy Actor`, and click on the result (you can also find it manually find under the **Utility** section). With both nodes present, connect the **Pressed** 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.



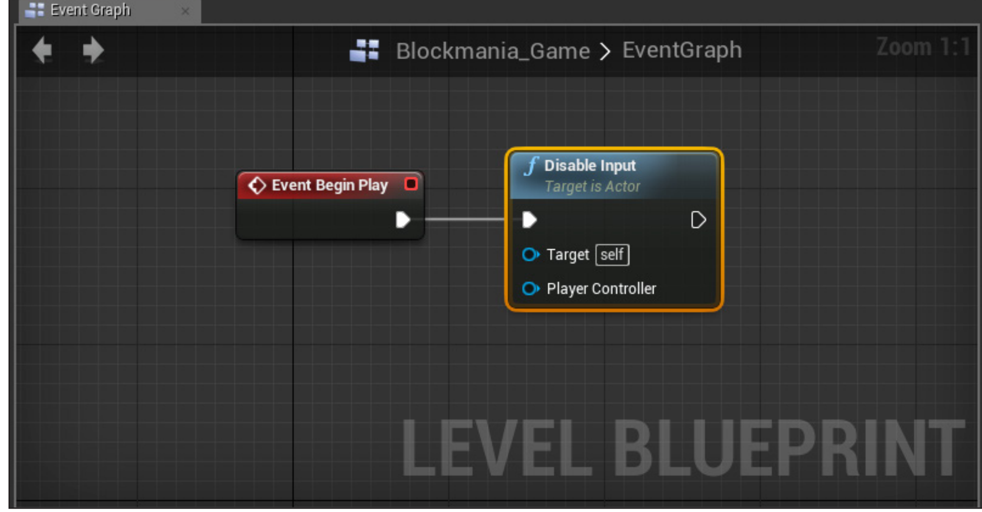
Doing so will create a reference node with the name of the actor written on it. Connect this to the **Destroy Actor** target pin.

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.

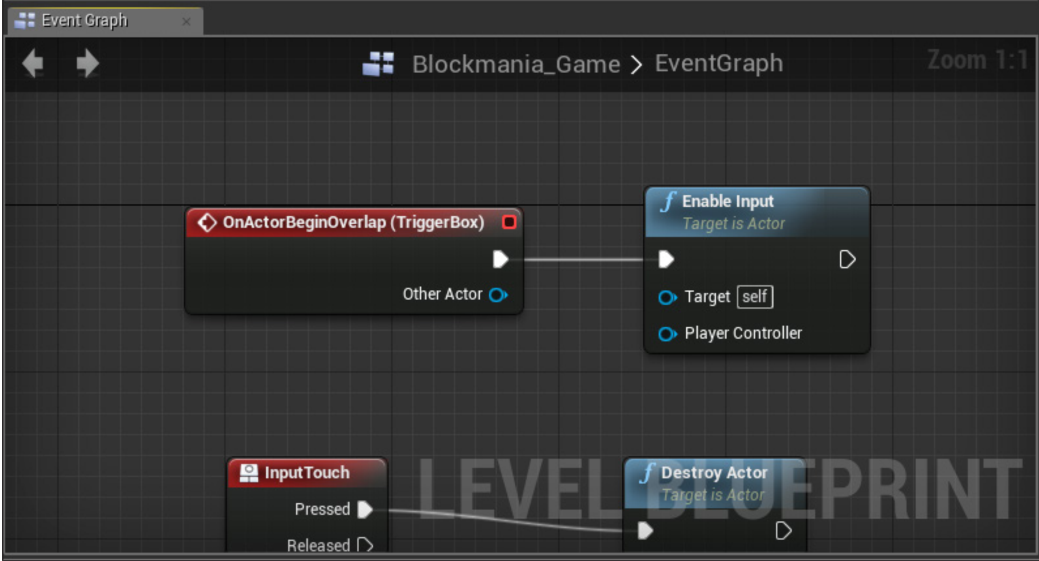
To resolve this issue, the first thing we are going to do is render the input disabled at the start of the game. As you may remember, the event node that is activated when the game begins, **Event Begin Play**, is already present in the **Graph Editor**. To this, we are going to attach a **Disable Input** node. Right-click and find the node, and connect it to the **Event Begin Play**.



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.

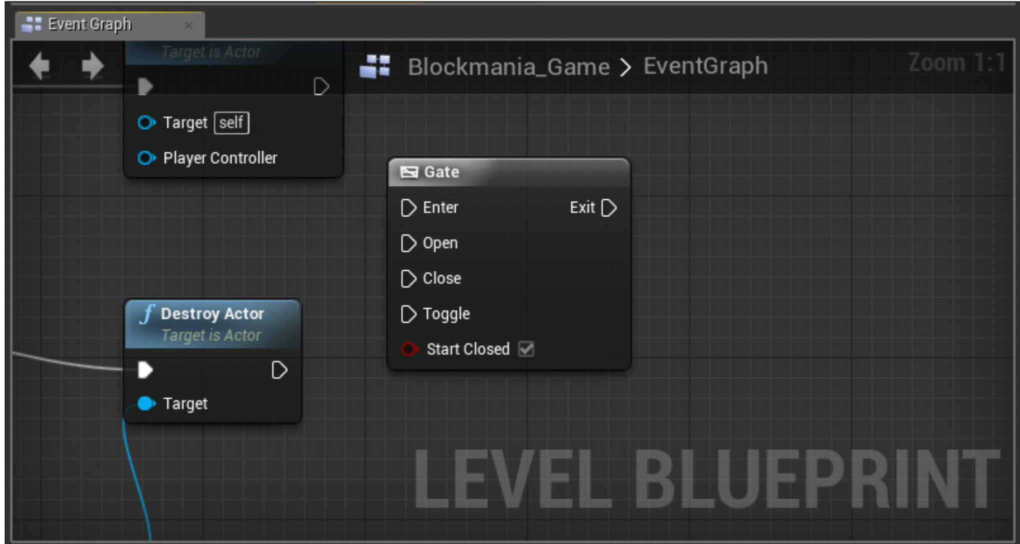
If you test your game now, when you tap on the screen, the key cube is not destroyed (although the character will start shooting projectiles again). Now, going back to our sequence, since we want the player to be only able to pick up the cube when they are at a certain distance, we are going to enable 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.



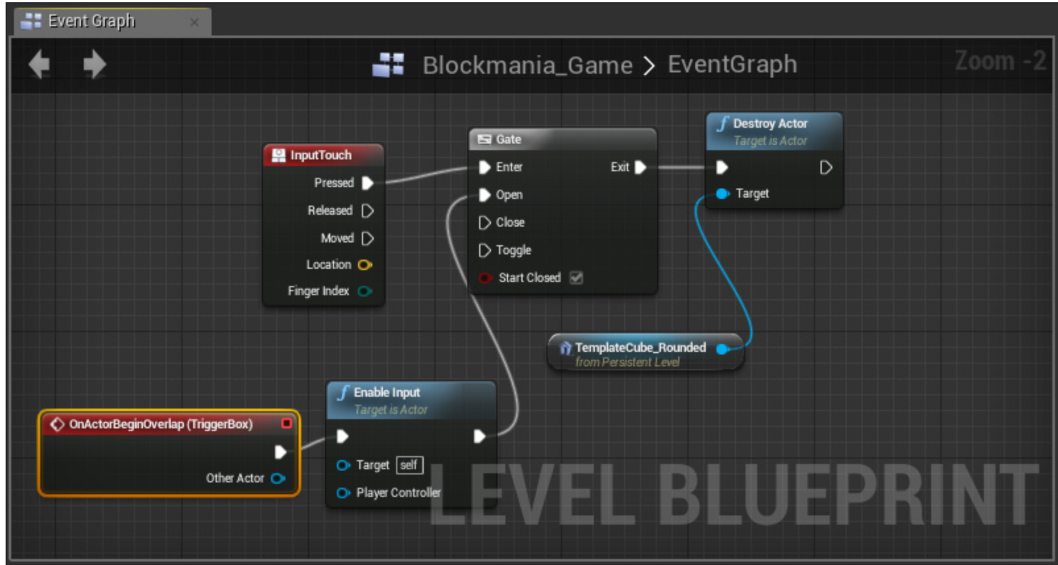
We are now almost done with our setup. If you were to test it now, you would find that it functions the way we intended it to. However, this is still not complete. For one, if you have more than one key cube in your room, both of them will disappear when you click on one. We do not want that; we want 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**.

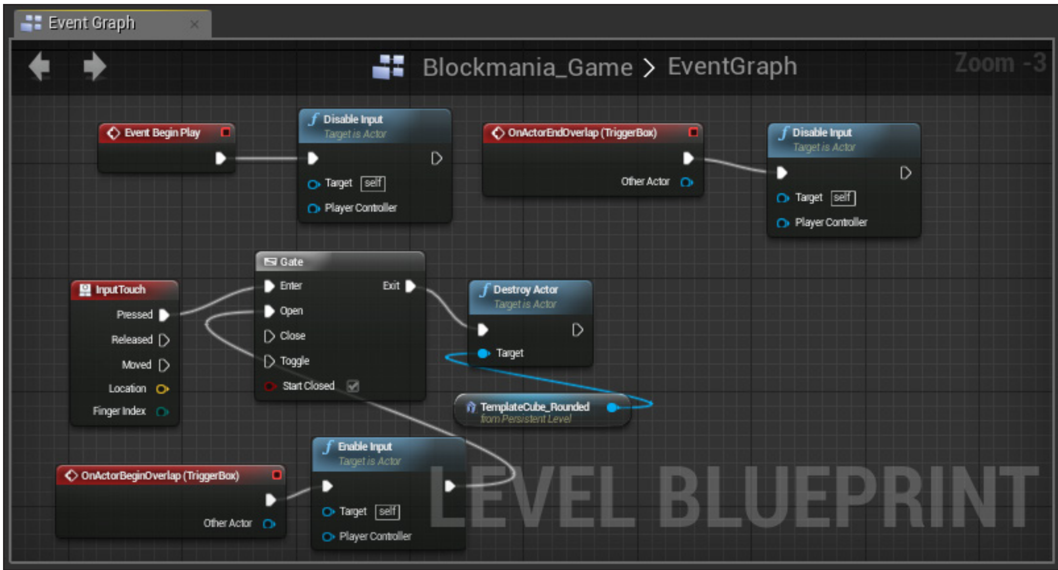


The event you wish to control is connected to the **Gate** node's **Enter** input. The rest of the events are used to control the flow. Connecting the **Close** input to an event will stop anything from passing through the **Gate** when the event has occurred. Similarly, connecting the **Open** input 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:



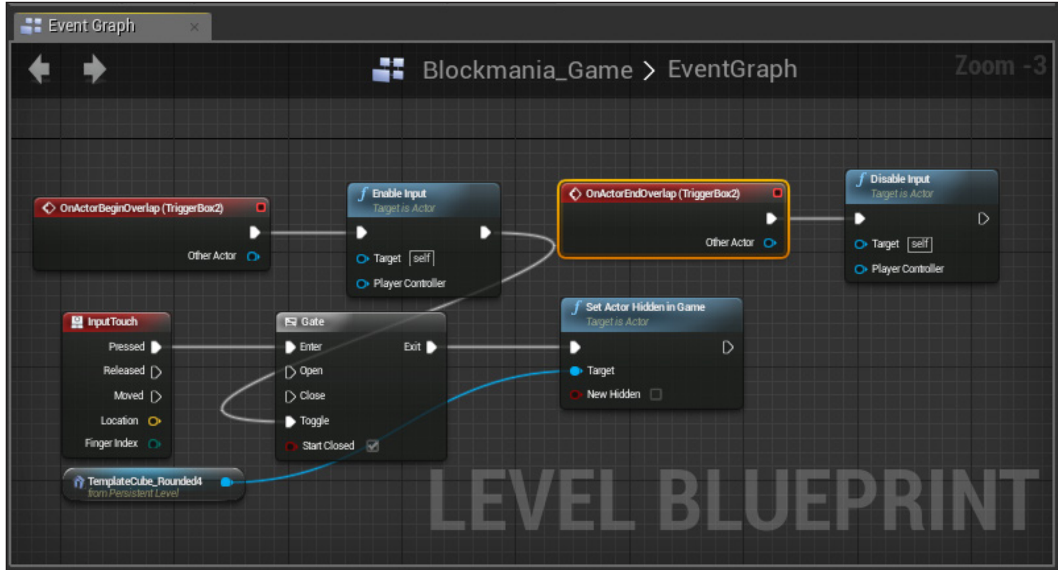
Now, when the player steps out of the trigger, that is, stops overlapping with the trigger, we again want to disable the input. So, with the trigger box selected, create an **OnActorEndOverlap** node. You can type it in the search bar or find it under **Add Event** for <name of the actor> | **Collision** | **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:



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.

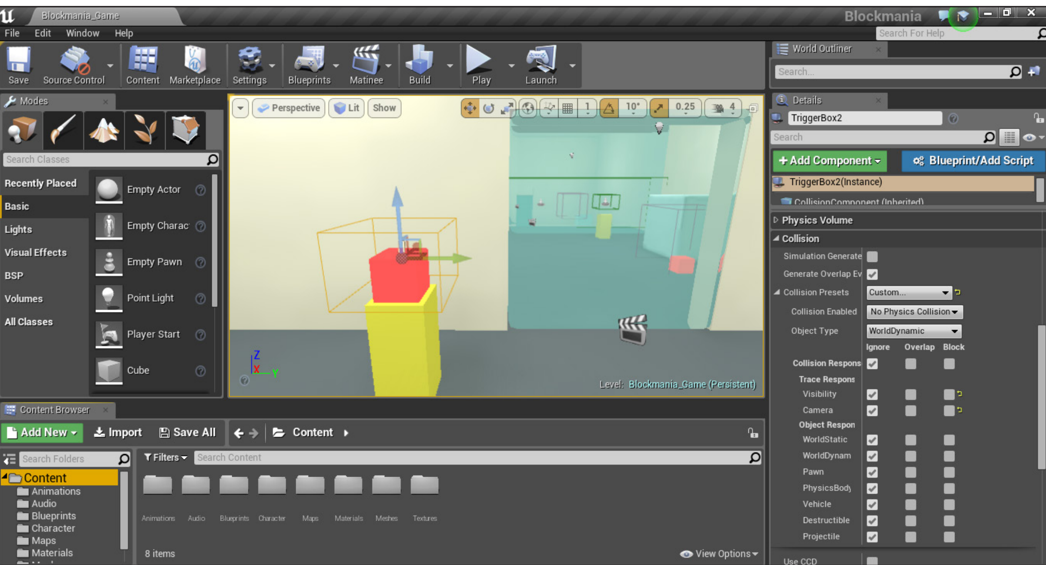
The first thing we need to do is add an overlap event that would enable the player's input. Hook them up the same way as you did with the previous trigger. Next, we are going to add another Touch event node. To this, we are going to connect a **Set Actor Hidden in Game** node. First, select the other key cube (the one that is hidden in the game), then right-click anywhere in the **Graph Editor**, and find it by typing in 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**.

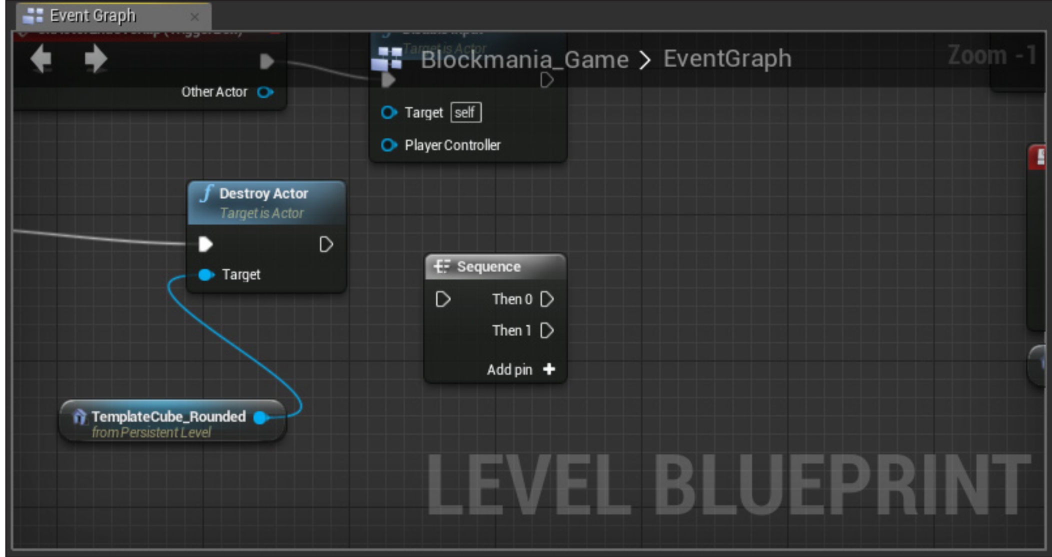


Next, click on the small triangle next to **Collision Presets** in order to open a list of the trigger's collision responses against different types of actors. There are three general responses:

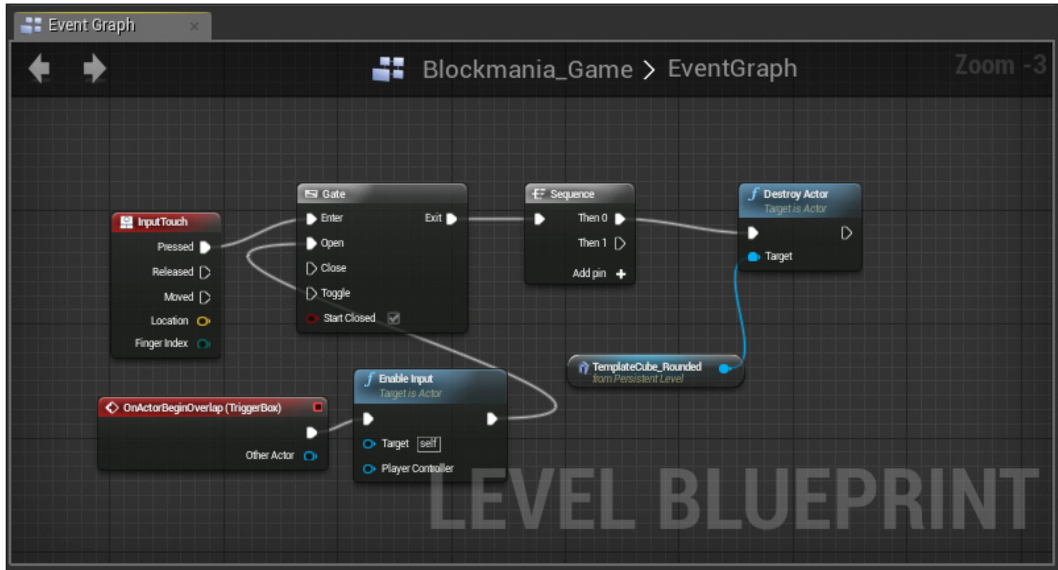
- **Ignore**: The trigger will not register any response to the collision. The actor(s) the trigger is set to ignore will neither be blocked, nor be registered by it.
- **Overlap**: The actor(s) the trigger is set to overlap will not be blocked by it, but the collision will be registered by it. This 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 later. 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.

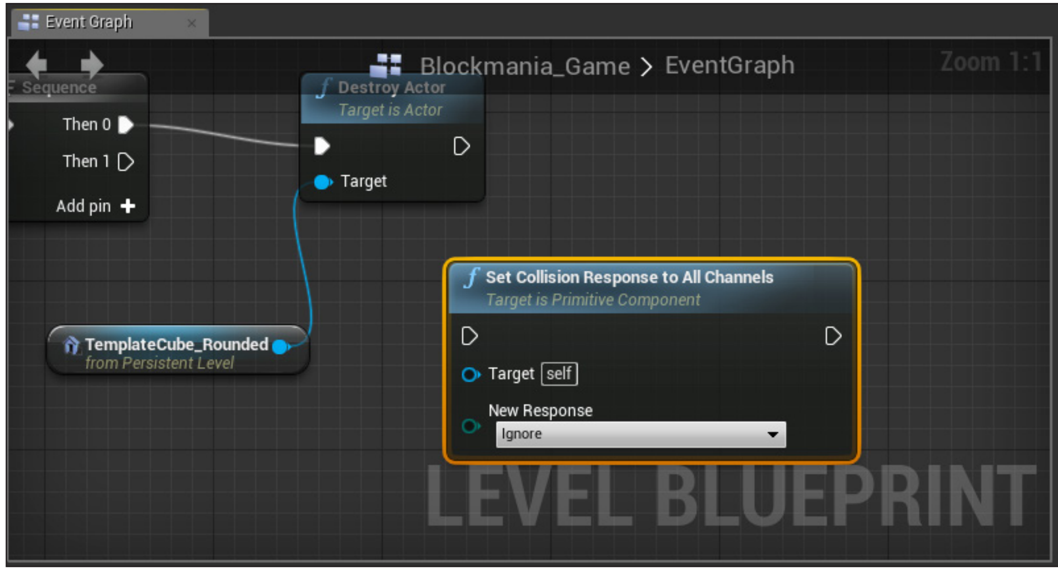


By default, a **Sequence** node has one input pin and two output pins. The first node is fired first, then the next, and so on. If you want more output pins, simply click on **Add Pins**, and it will create another output pin. Now, connect the **Gate** node's output pin to the **Sequence** node's input pin. Then, connect **Then 0** to the **Destroy Actor** node.

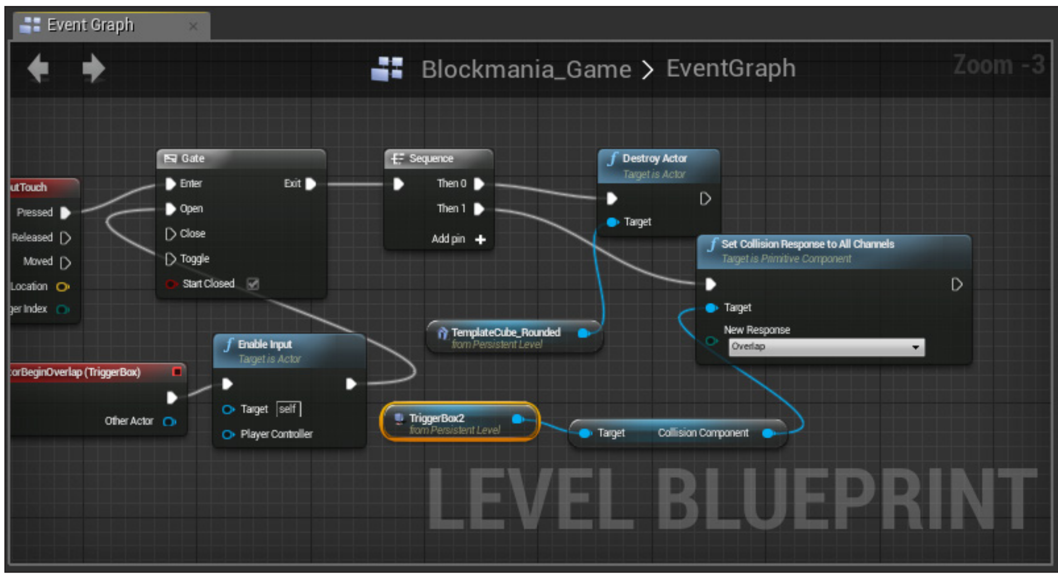


We will now change the **Collision Response** of the trigger to **Overlap** when the player has picked up the cube. With the trigger selected, right-click on it and first uncheck the **Context Sensitive** box, which is located at the top-right corner of the menu. When you have an actor selected in the **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 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.



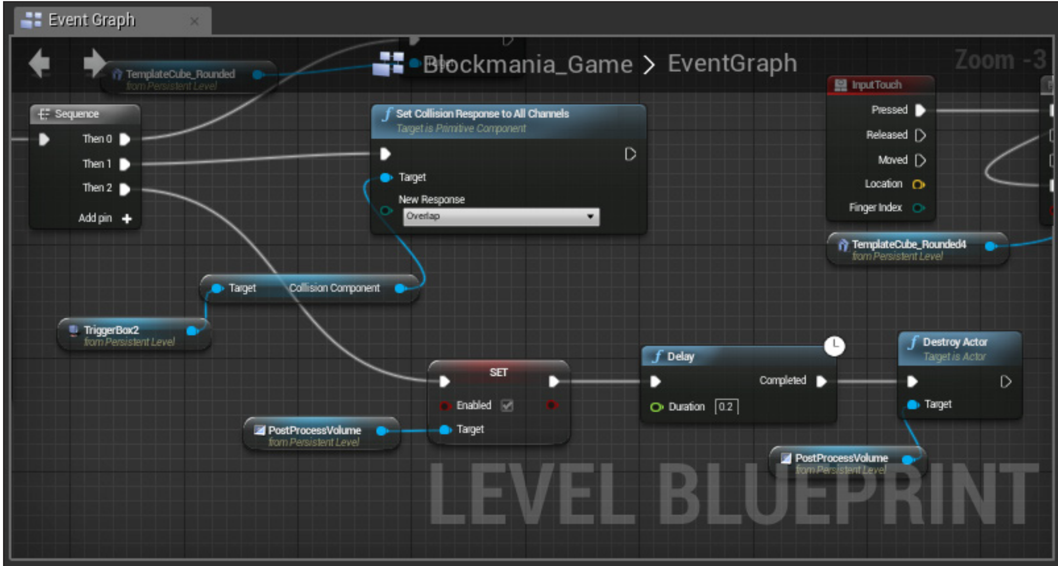
Here, you can set a target actor whose **Collision Response** you wish to change. If you click on the dropdown menu, you will see three settings: **Ignore**, **Overlap**, and **Block**. If you recall, these were the three types of responses in the trigger's **Collision** section. We had set it to **Ignore**, 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 out 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 in 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.



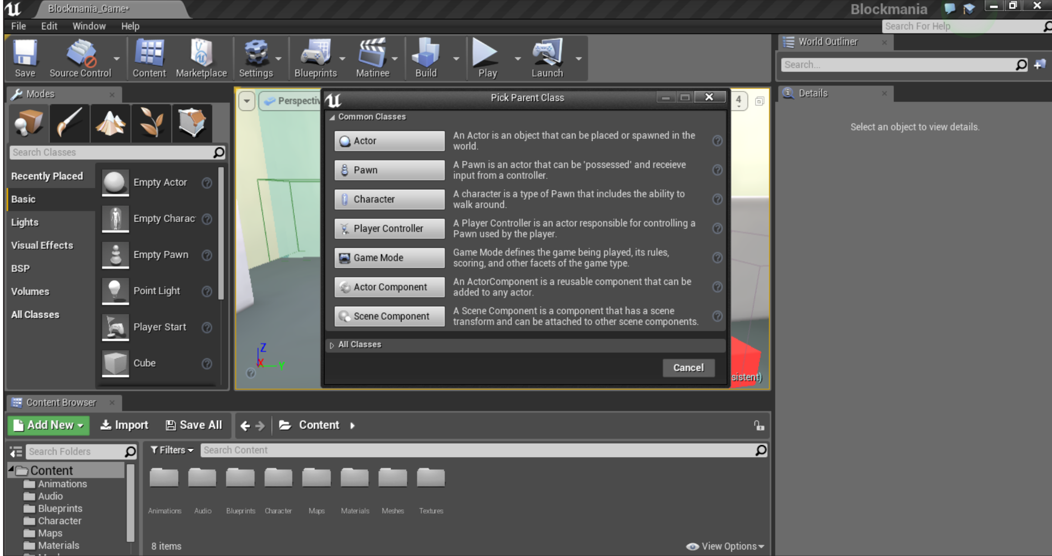
And there you have it! We have a pickup and placement system for our key cube. We also have a visual indicator that we have picked up the cube (we will cover how to script the door opening and closing in the next chapter). Now, though this was a fairly simple and small setup, doing this for every key 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.

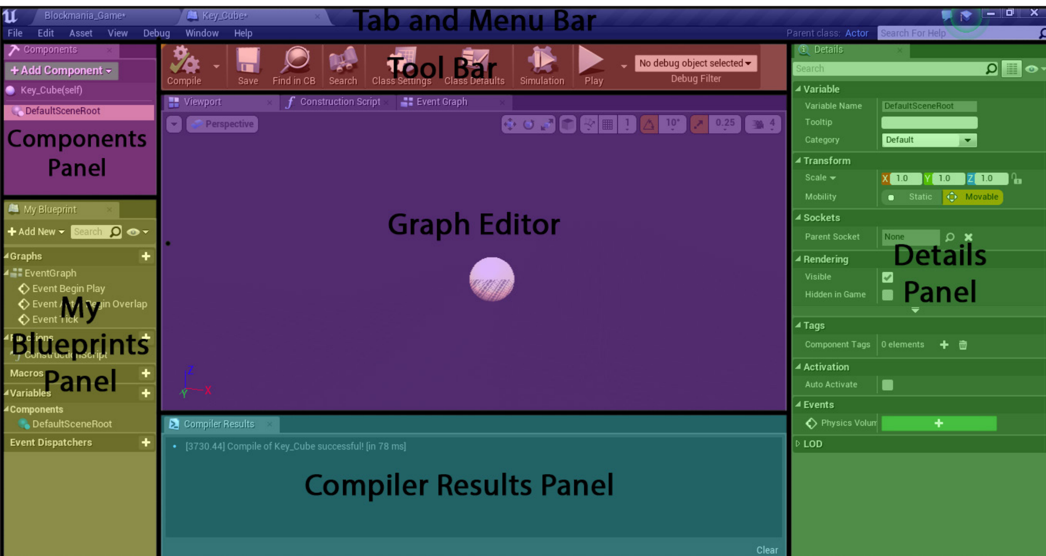


At the top, you will see the most common classes. These are as follows:

- **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:

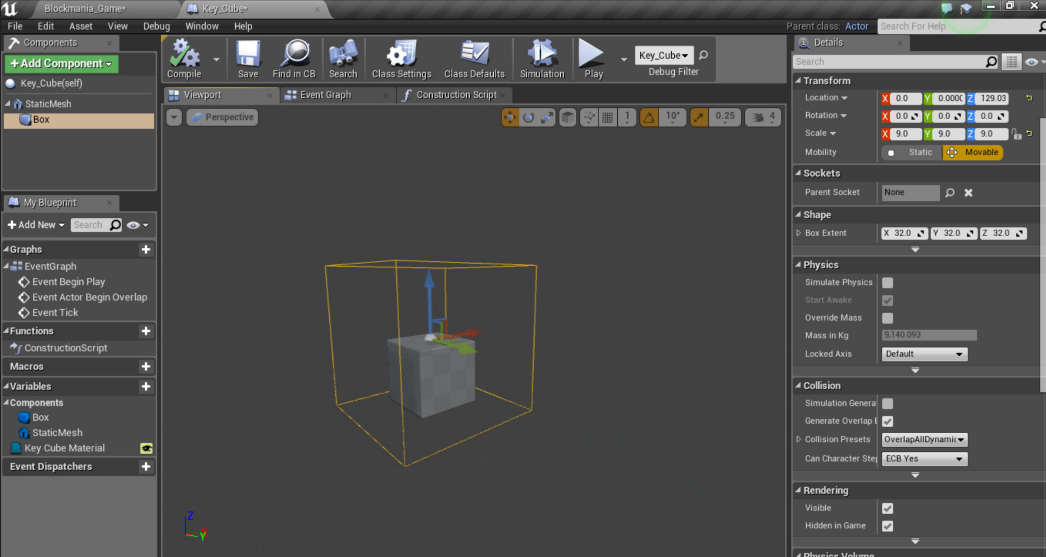
- **Components panel**: The **Components panel** is where you can view, and add components to the Blueprint class. The default component in an empty Blueprint class is **DefaultSceneRoot**. It cannot be renamed, copied, or removed. However, as soon as you add a component, it will replace it. Similarly, 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

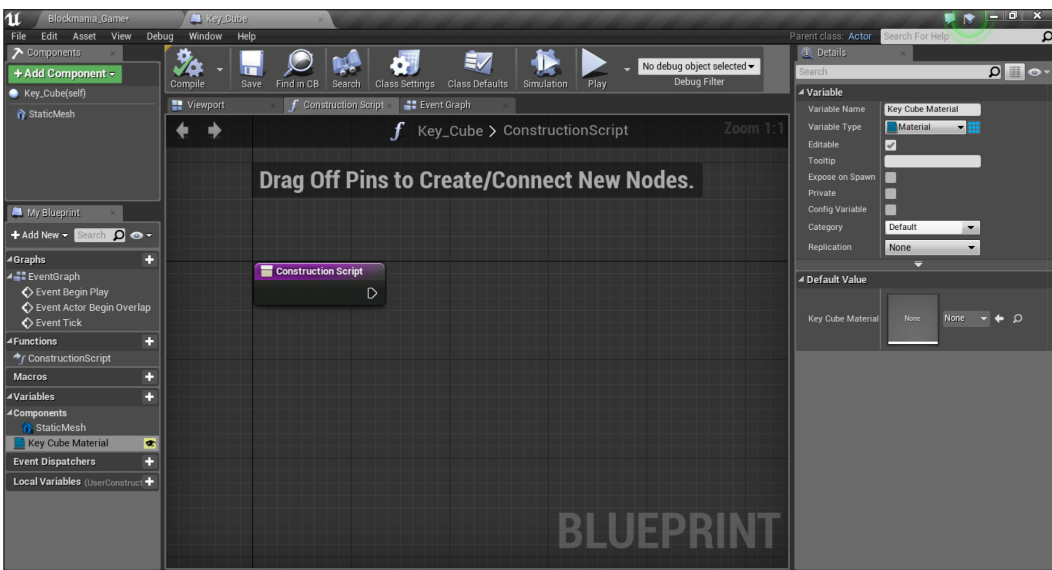
The first thing we need are the components. We require three components: a cube, a trigger box, and a **PostProcessVolume**. In the **Viewport**, click on the **Add Components** button, and under **Rendering**, select **Static Mesh**. It will add a **Static Mesh** component to the class. You now need 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.



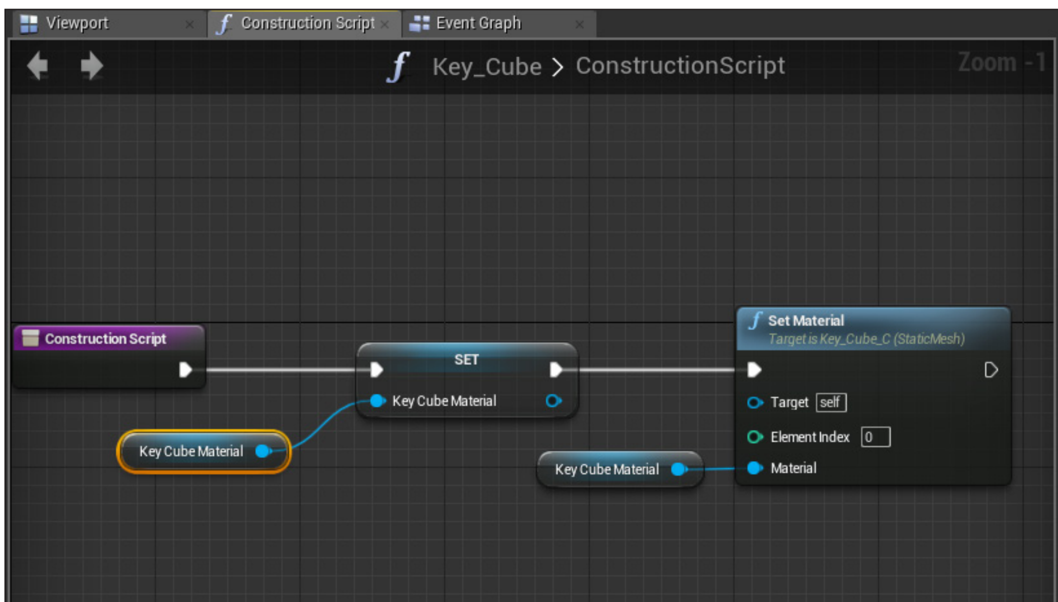
The Construction Script

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.

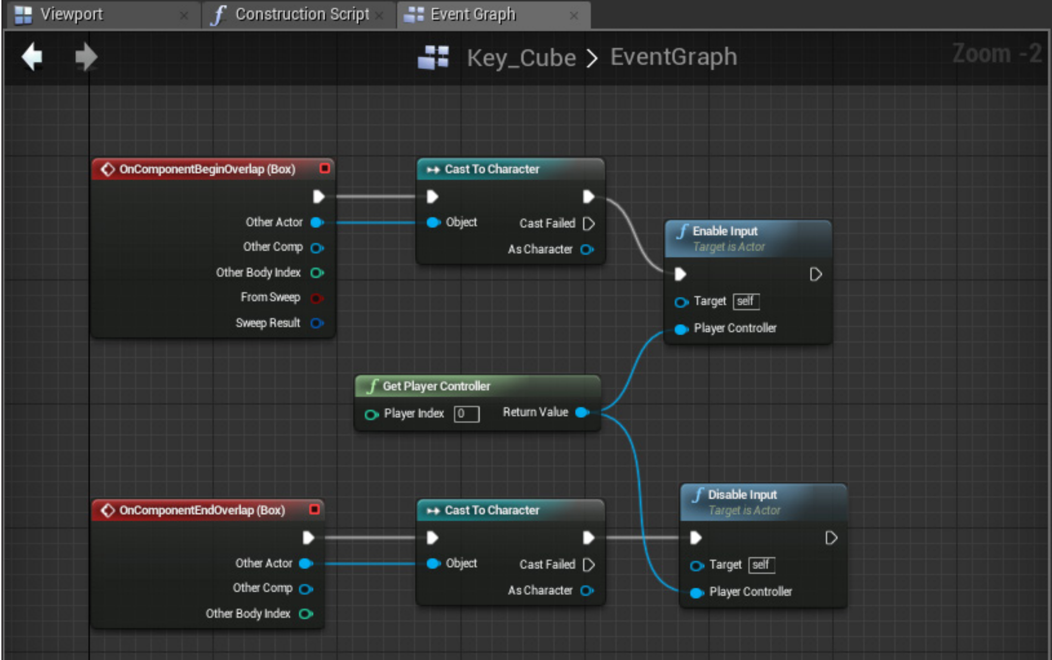


Let's test it out. Add the Blueprint class to the level. You will see a **TemplateCube_Rounded** actor added to the scene. In its **Details** panel, you will see a **Key Cube Material** option under the **Default** section. This is the variable we created inside our **Construction Script**. Any material 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.

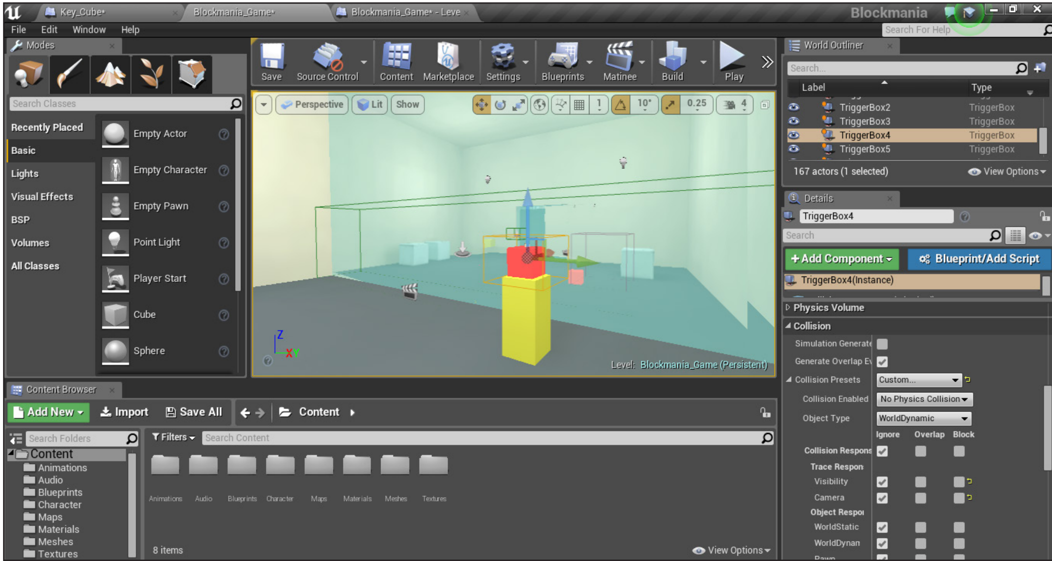
The Event Graph

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:



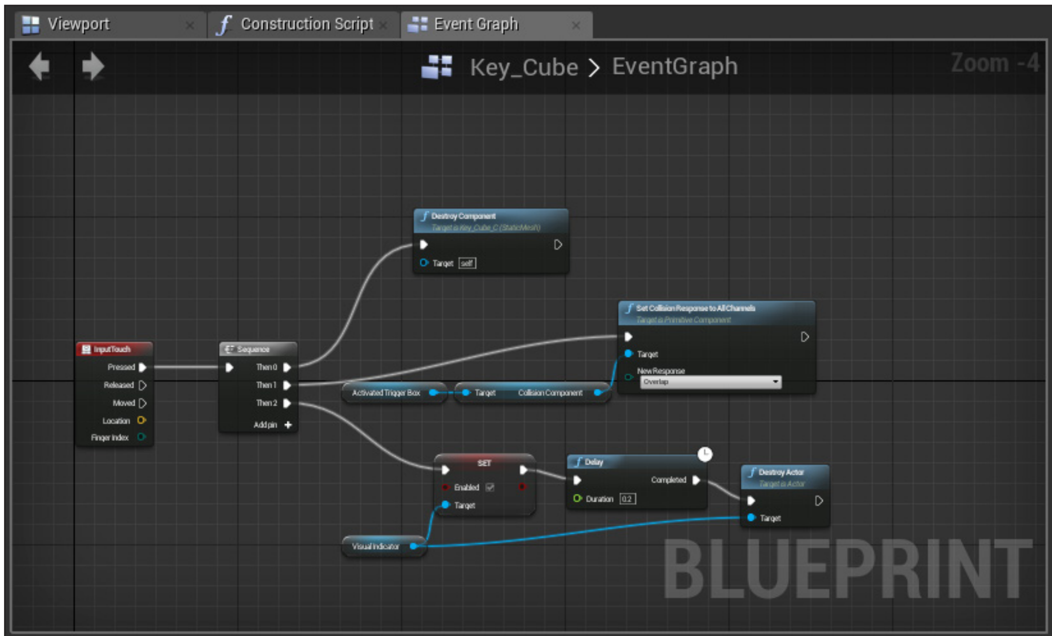
You can move the key cube we had placed earlier on top of the pedestal, set it to hidden, and put the key cube Blueprint class in its place. Also, make sure that you set the collision response of the trigger actor to **Ignore**.



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:

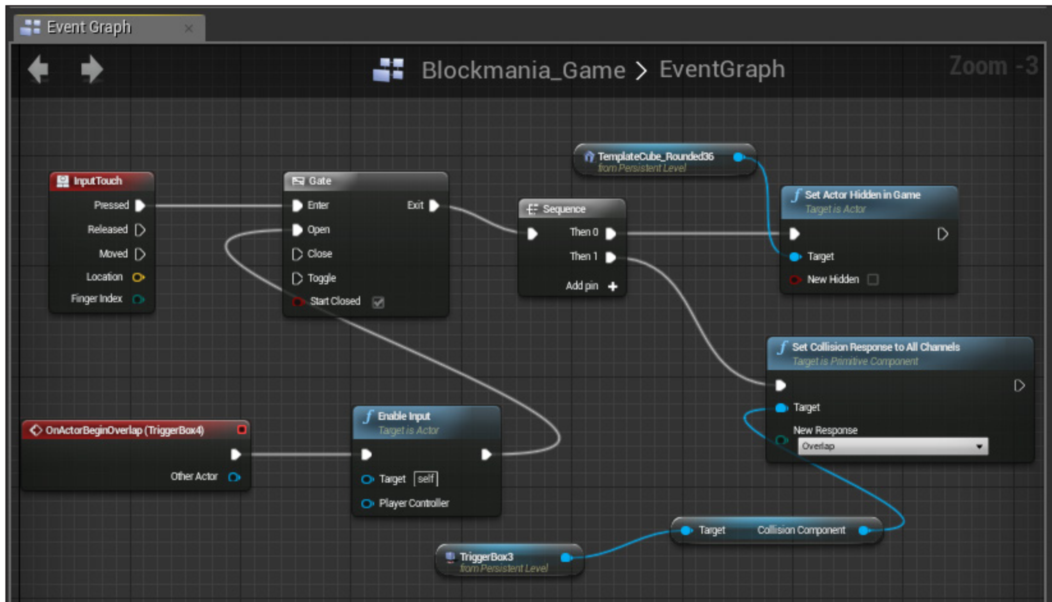


Back in the **Viewport**, you will find that under the **Default** section of the Blueprint class actor, two more options have appeared: **Activated Trigger Box** and **Visual Indicator** (the variables we had created). Using this, you can assign which particular trigger box's collision response you 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.



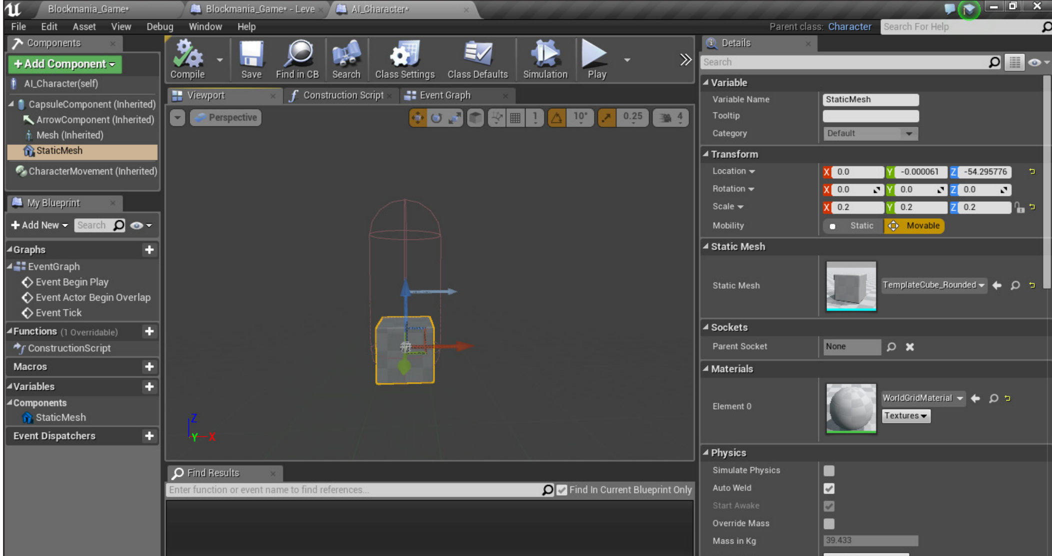
Test it out! You will find that everything is working as expected. Now, do the same with the remaining key cubes. Pick which trigger box and which post process volume to activate when you touch on the screen. Then, in the Level Blueprint, script in which key cube to unhide, and so on (place the key cubes 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.

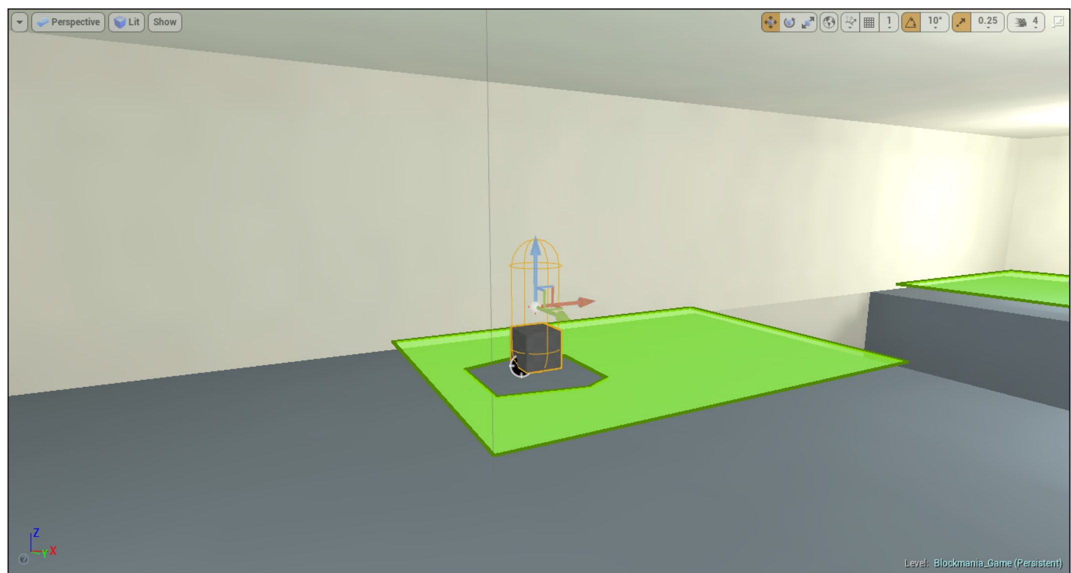
Scripting basic AI

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.



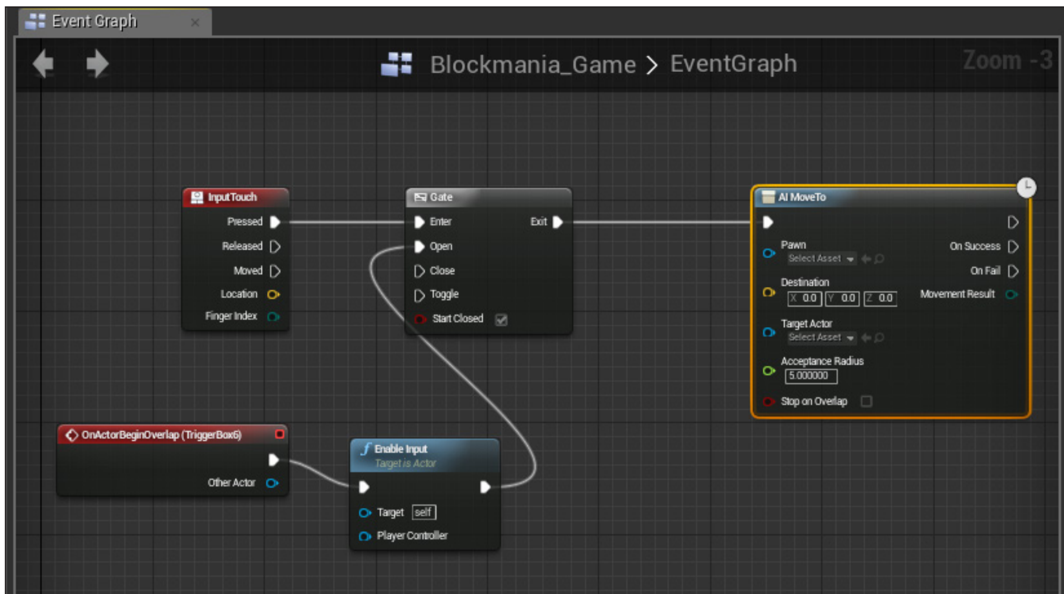
This is all we require for our AI character. The rest we will handle in Level Blueprints. Next, place **AI_Character** into the scene on the Player side of the pit, with all of the switches. Place it directly over the Target Point actor.



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.

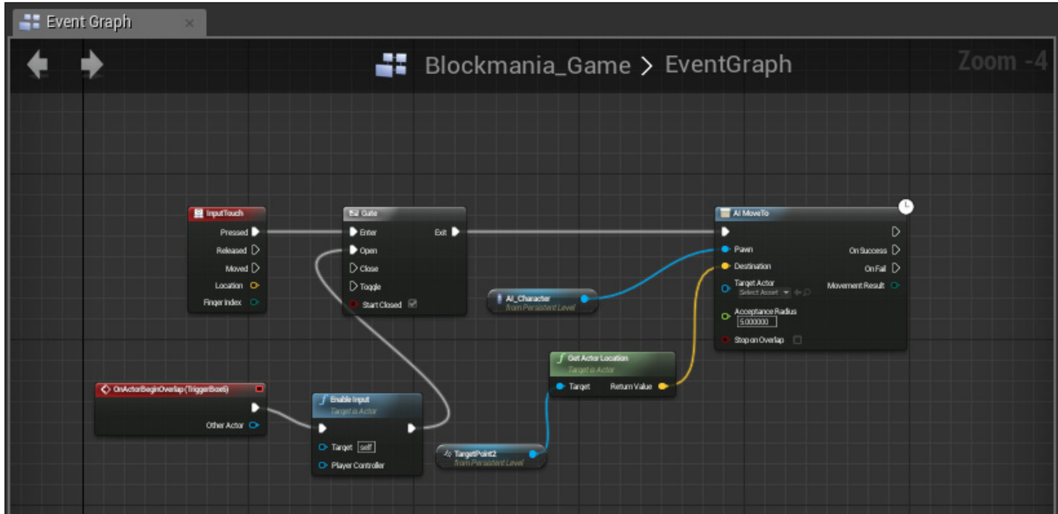
To move the AI character, we will need an **AI Move To** node. The first thing we need is an overlapping event for the trigger over the first switch, which will enable the input, otherwise the AI character will start moving whenever the player touches the screen, which we do not want. Set up an **Overlap** 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.

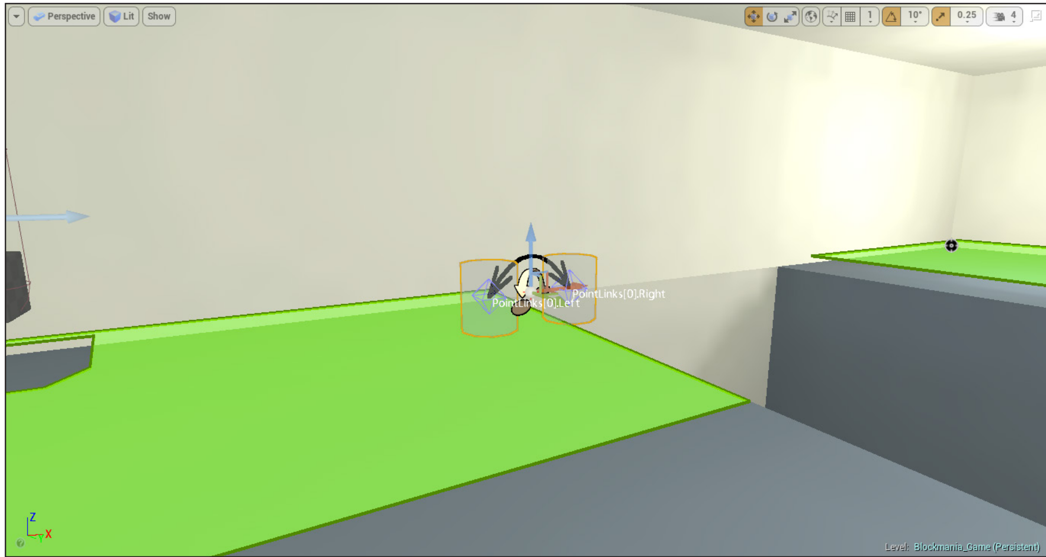


We now need to specify to the node which character we want to move, and where it should move to. To specify which character we want to move, select the AI character in the **Viewport**, and in the Level Blueprint's **Graph Editor**, right-click and create a reference for it. Connect it to the **Pawn** 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.

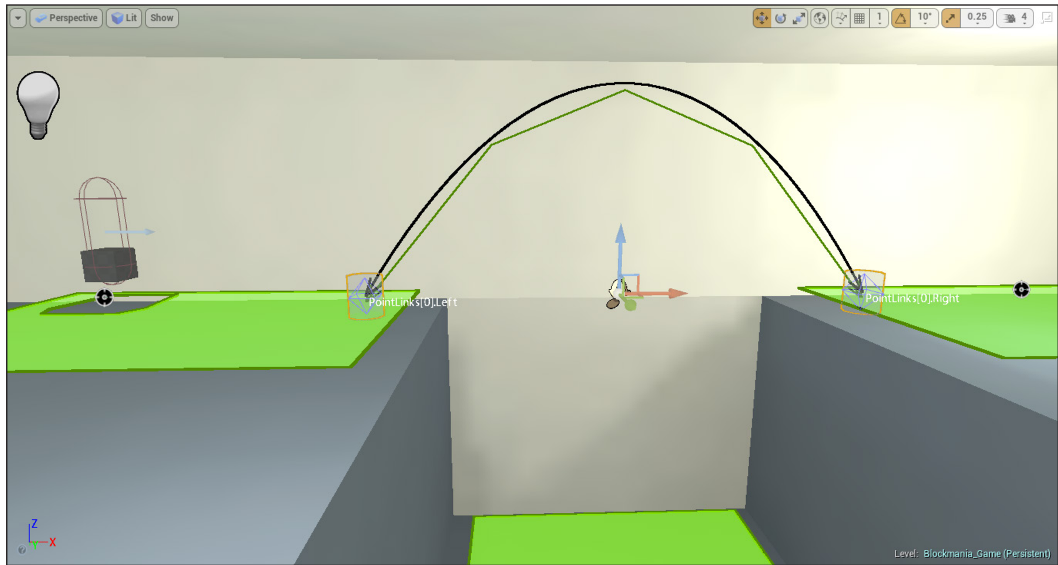
Finally, connect its **Return Value** to the **Destination** input of the **AI Move To** node.



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.



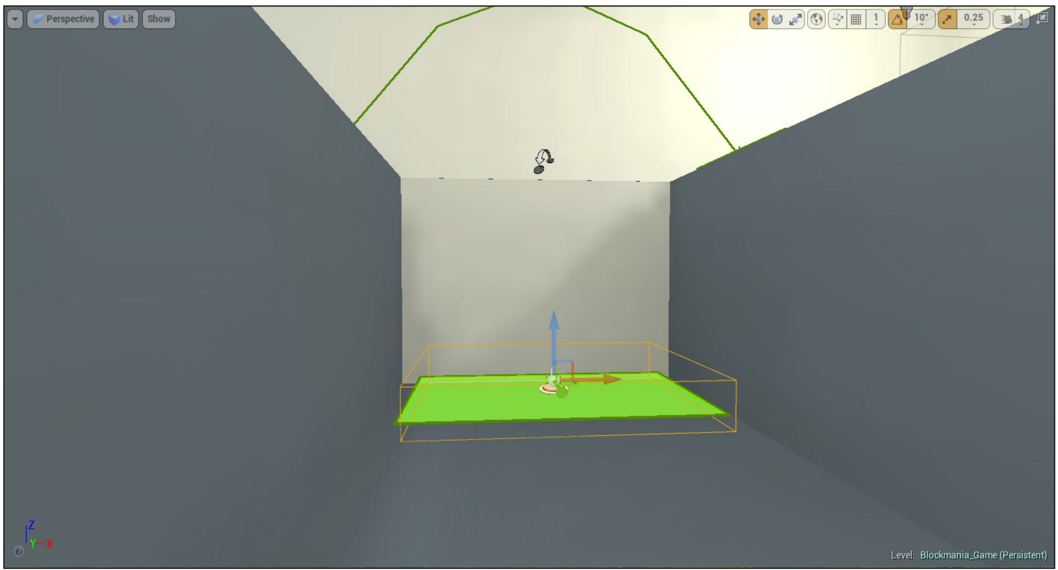
The actor is depicted by two cylinders with a curved arrow connecting them. We want the first cylinder to be on one side of the pit and the other cylinder on the other side. Using the **Scale** tool, increase the size of the **Nav Proxy Link** actor.



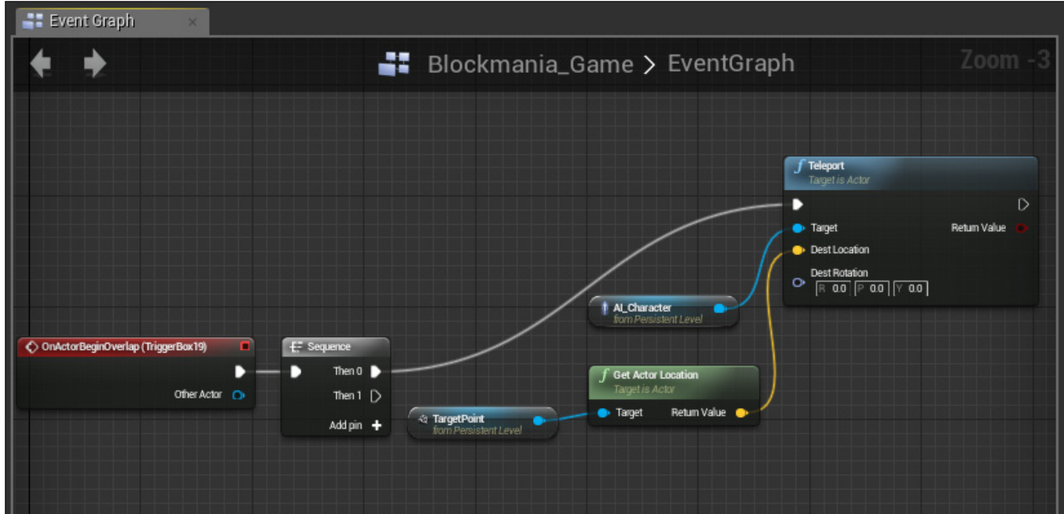
When placing the **Nav Proxy Link** actor, keep two things in mind:

- 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

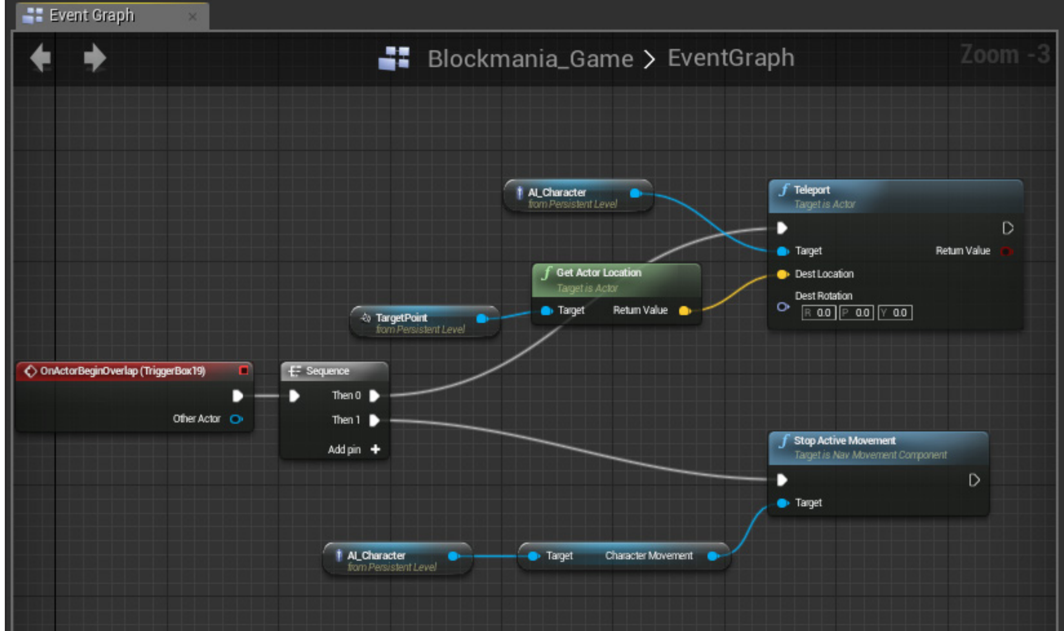
Once placed, you will see that the AI character falls off when it reaches the edge of the pit. We are not done yet. We need to bring the AI character back to its starting position so that the player can start over (or else the player will not be able to progress). For that, we need to first place a trigger 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.

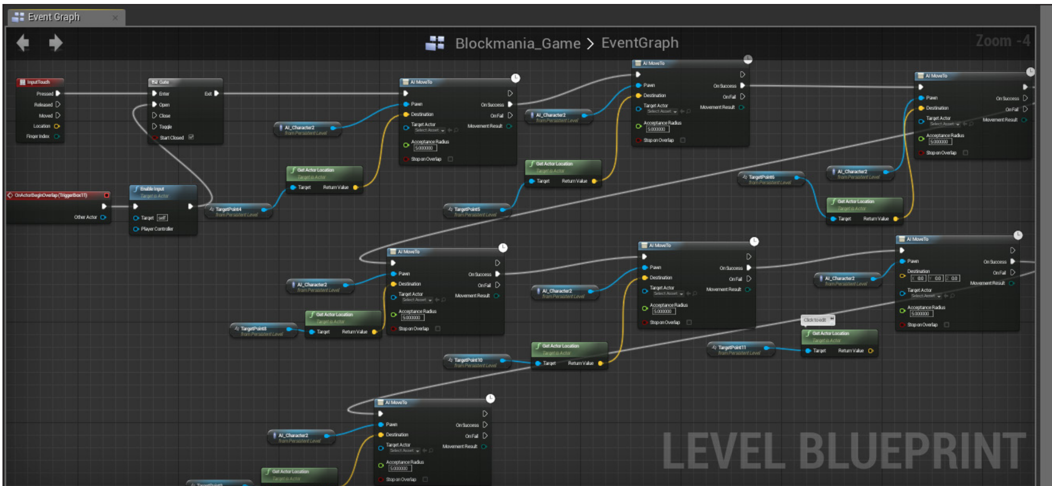


To stop the AI character's movement, right-click anywhere in the **Graph Editor**, and first uncheck the **Context Sensitive** box, since we cannot use this function directly on our AI character. What we need is a **Stop Active Movement** node. Type it into the search bar and create it. Connect this 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.

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:

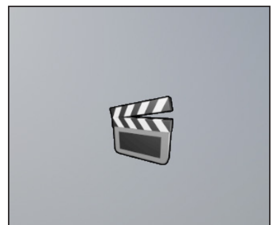
- What is Unreal Matinee?
- Unreal Matinee user interface
- Using Unreal Matinee in our game

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:

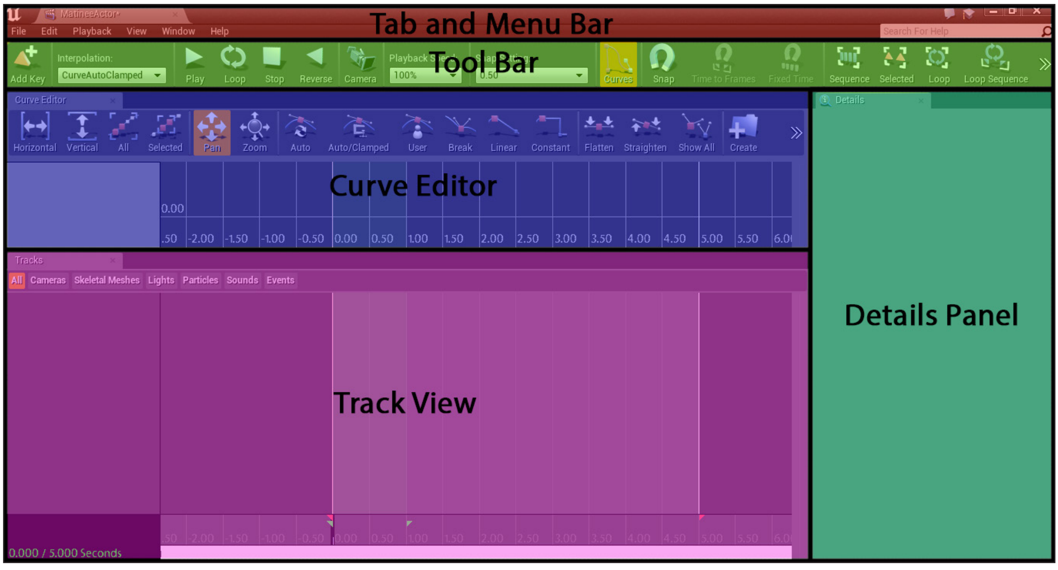


We have already discussed one way of adding Matinee actors in *Chapter 4*, *Using Actors, Classes, and Volumes*, through the Modes panel. Another way of adding Matinee actors while keeping a check on how many of them are placed in the scene is by clicking on the **Matinee** button in the Viewport toolbar, 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.

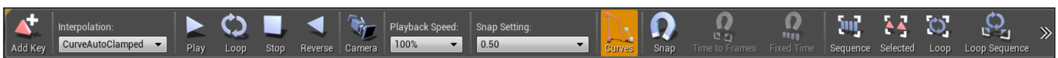
Below that is the menu bar, containing the following actions and functions:

- **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:

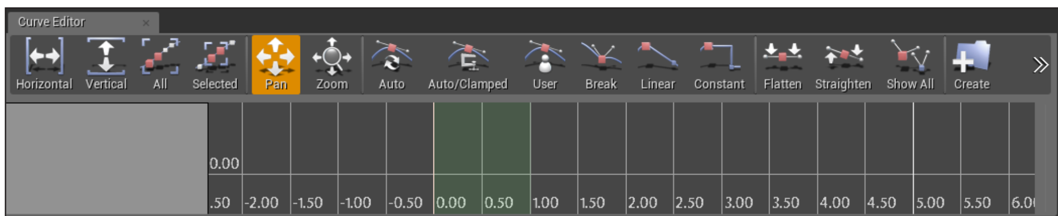
- **Add Key**: This adds a key frame on the current position of the animation track.
- **Interpolation**: Interpolation means finding a point between two other points. In key frame animation, interpolation is an important tool, since it makes the animation smooth. With this, you can set the initial interpolation 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.



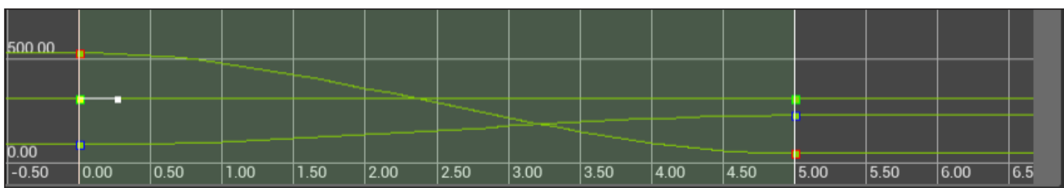
At the top is the toolbar of the Curve Editor. Let's look at its options:

- **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 it.
- **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.

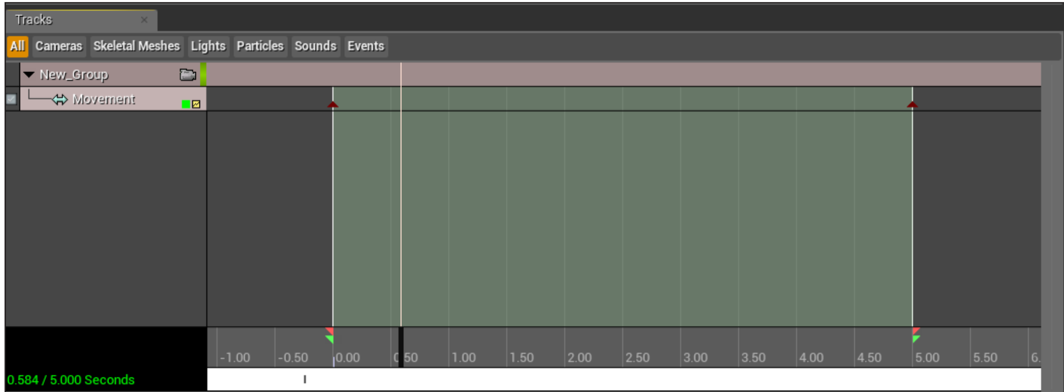
To its right, we have the Graph Editor, where you can see and tweak the curves.



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.

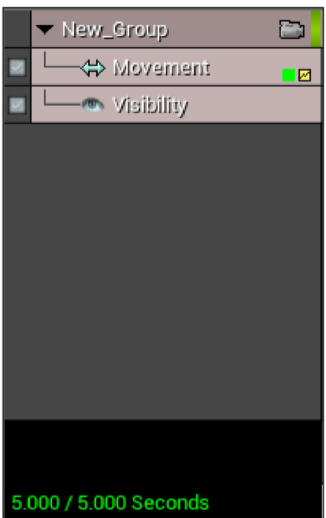
The Tracks panel

In the **Tracks** panel, you can see all of the tracks, folders, and groups in your animation sequence.



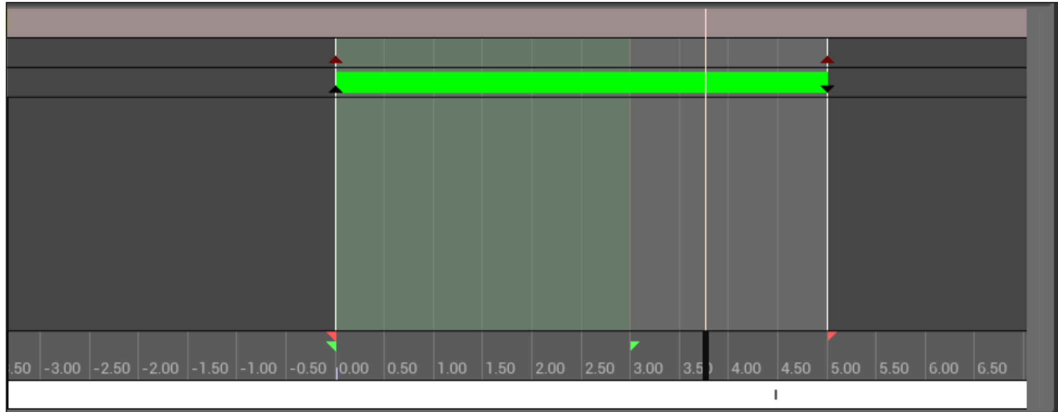
Here, you can see the timeline specifying how long the animation is, the groups you have created, all the components that are involved in the animation, and so on. At the top of the panel is the tab bar. Below it is the **Group** tab. For simple animations, you do not really need this, but if you have 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.

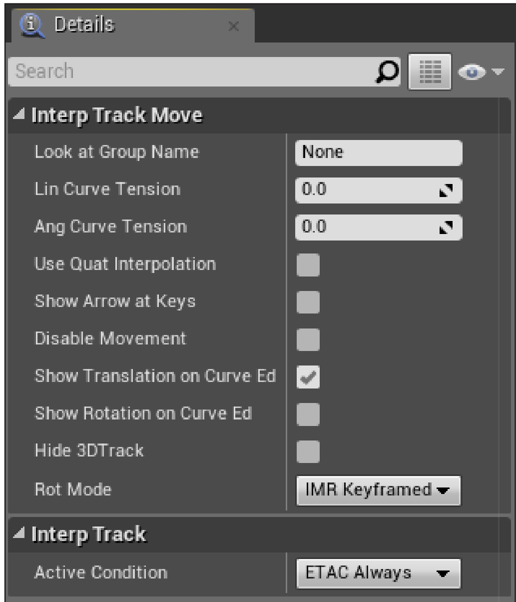
To its right is the animation timeline.



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.

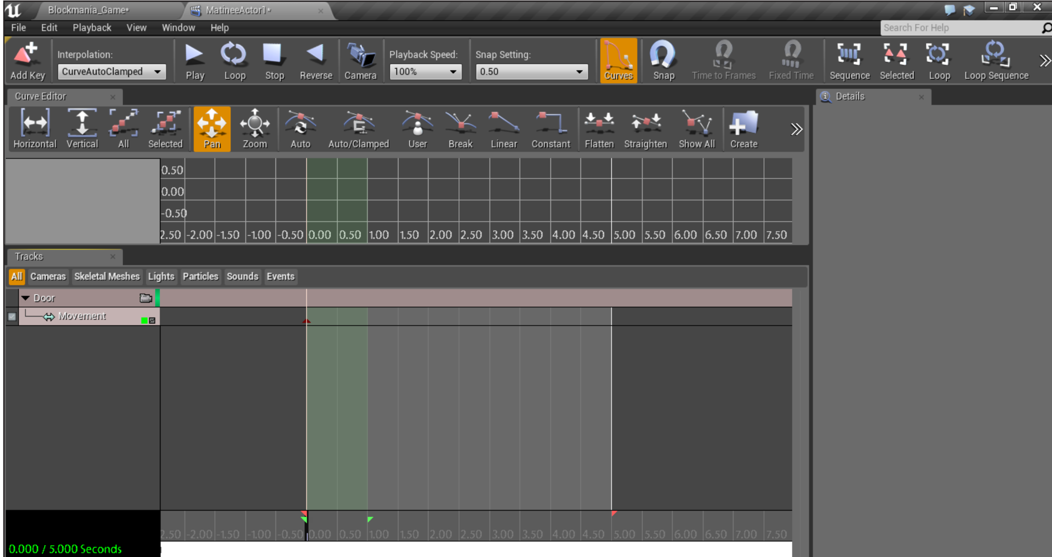
Animating the door

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.

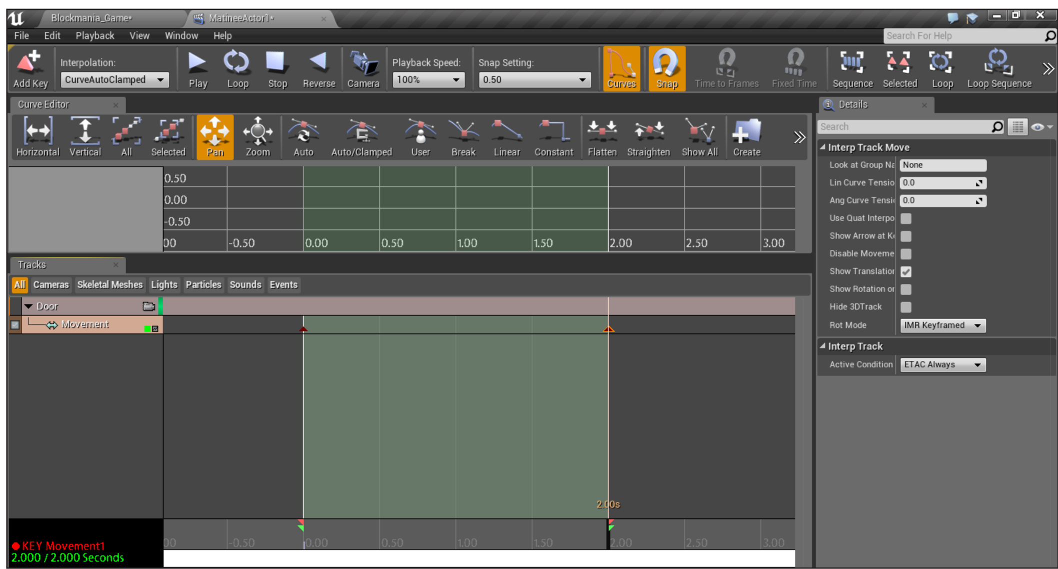
Room 1

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.

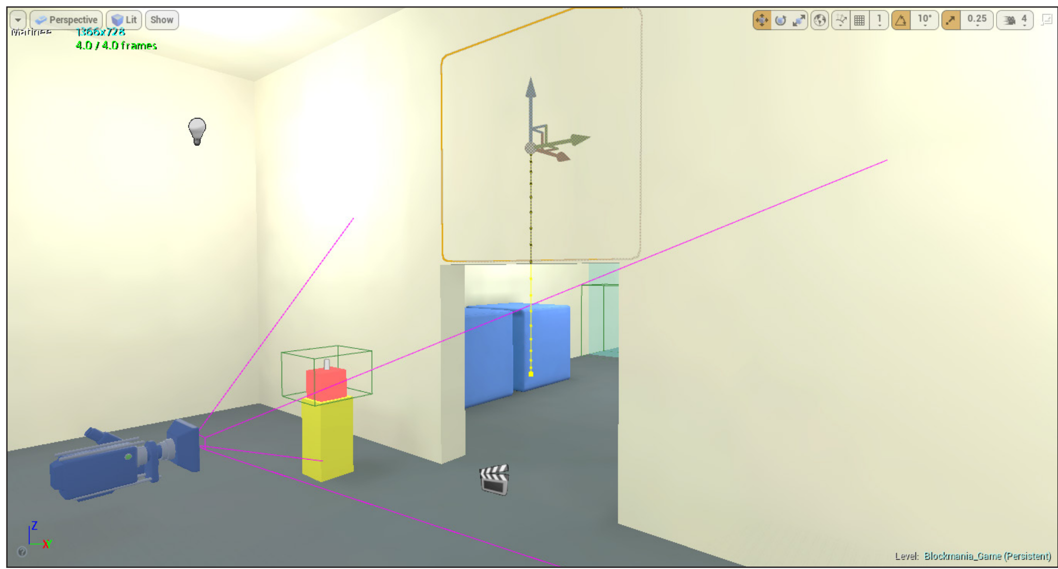


In the timeline, before the Movement track, you will see a small maroon arrow over the 0-second mark. This is a key frame. Since we have a simple animation, we only need two key frames: one at the beginning of the animation sequence and the other at the end (as shown in the preceding screenshot). To 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.

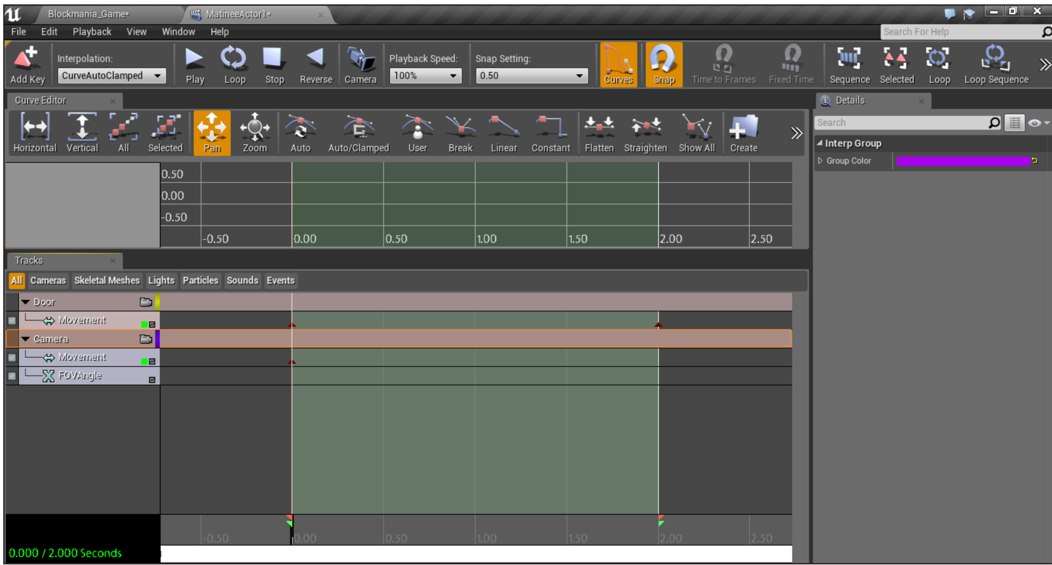


You will see a vertical yellow line whenever you move the door. This line shows the path the door will follow when the animation is played. To view the animation, go back to Matinee and click on the **Play** button in the toolbar.

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:

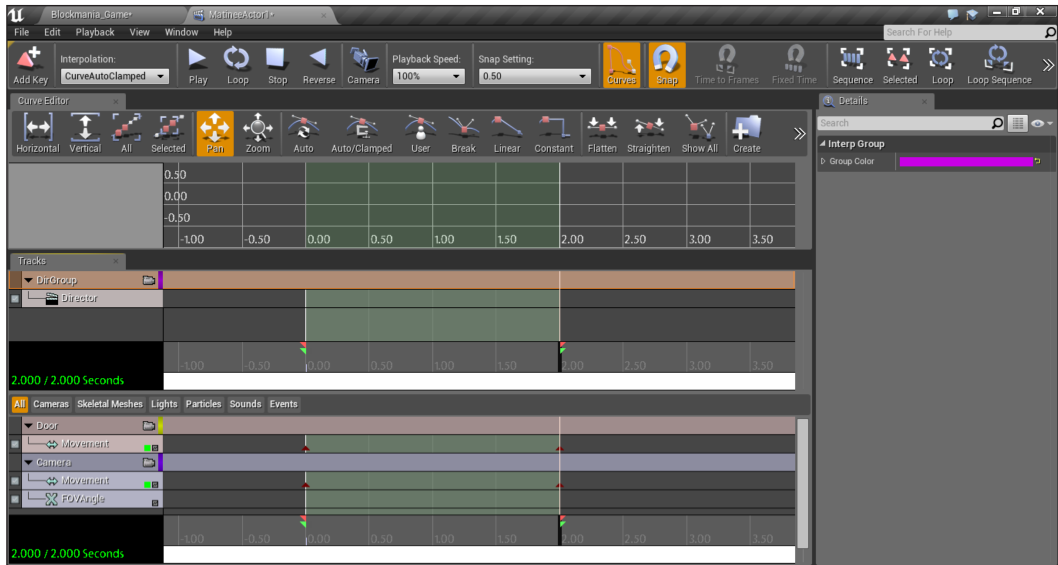
1. Take the control away from the player.
2. Switch from the main camera to the camera in front of the door.
3. Play the animation.
4. Give back the player control over the character.

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**.

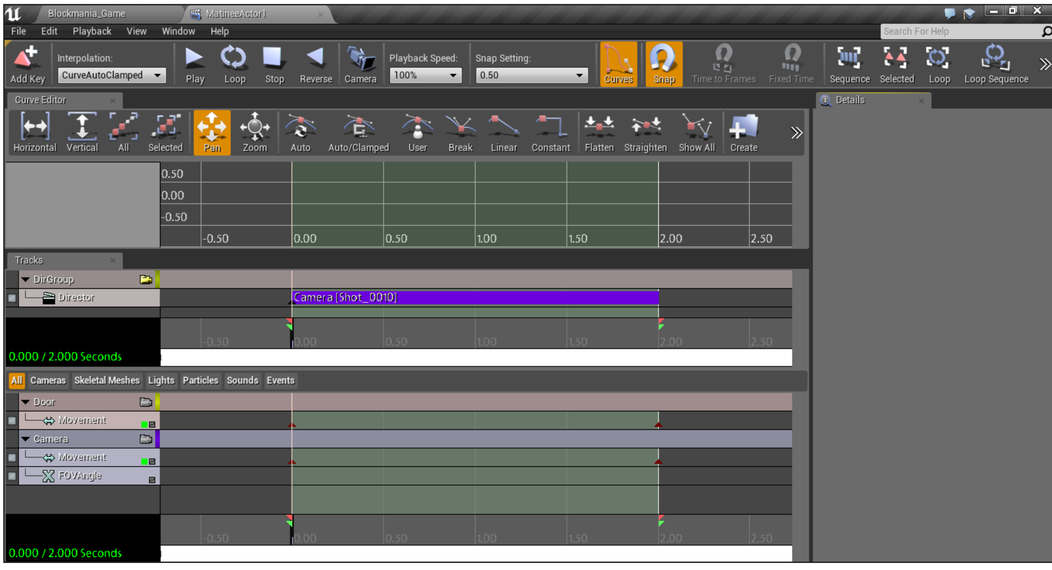


A Camera group has two tracks: a **Movement** track and a **FOVAngle** angle track. You can use this to move the camera and set its field of view when the Matinee is playing. In this case, we want the camera to be stationary. So, take the timeline slider to the end of the timeline and hit *Enter* in the **Movement** track of the camera group to create a key frame.

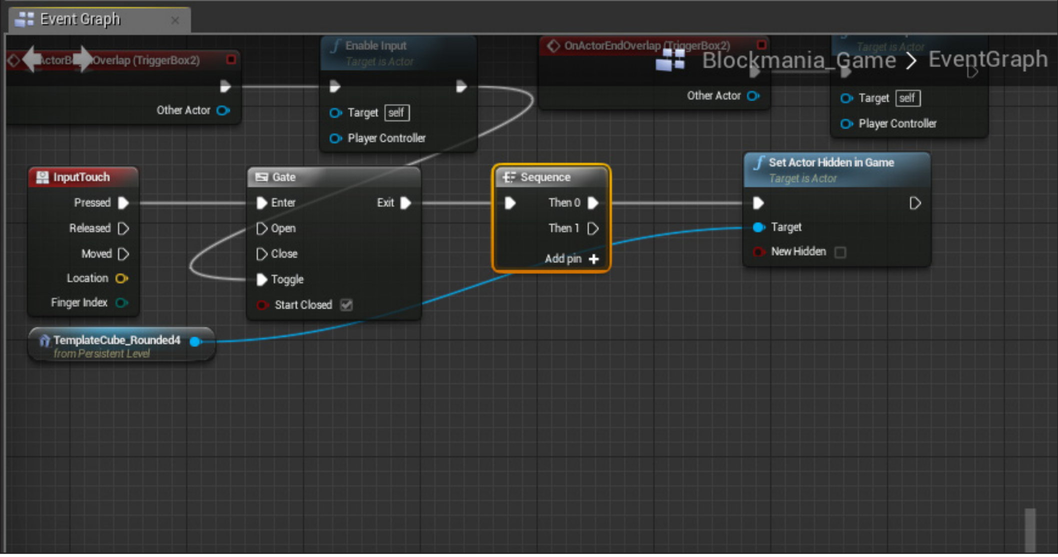
The next thing we need is what is called a **Director** Group. One of its uses is to assign which camera to switch to when playing the animation. Again, right-click on the **Tracks** panel and select **Create New Director Group**. A new, separate track will be created above the **Track** panel for the **Director** Group.



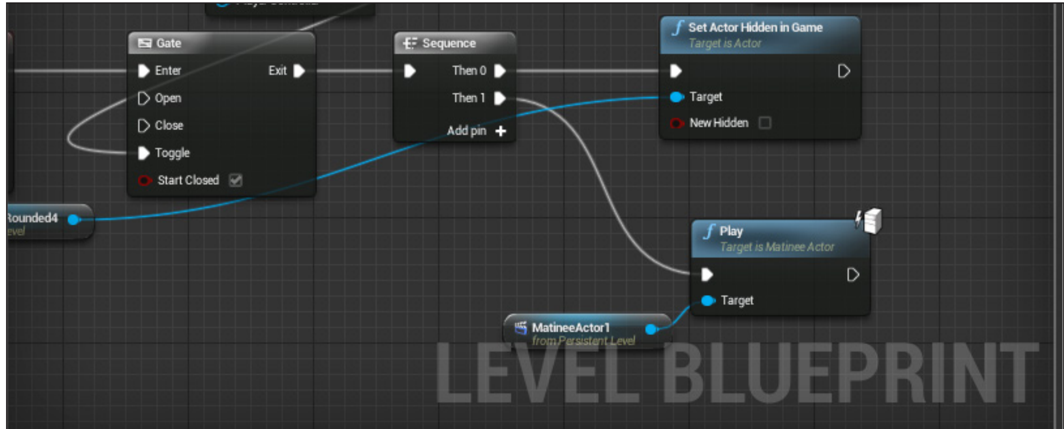
Now that we have our **Director** group, we need to tell it which camera to switch to when the animation is playing. To do so, select it and press *Enter*. This will open a small window asking you which group to cut to. This window enlists all the groups we have created in our Matinee. Here, click on 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.

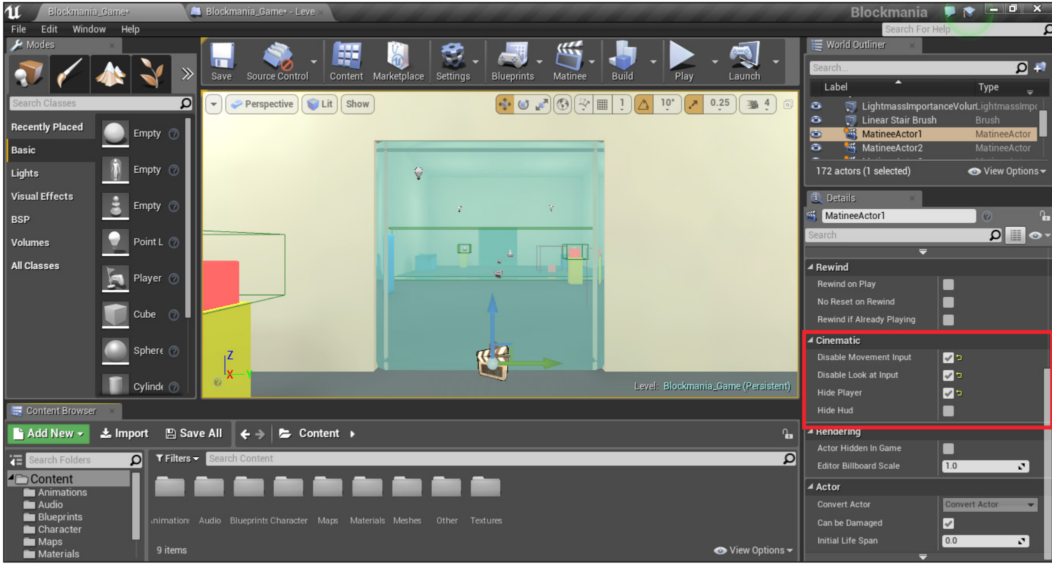


Next, with the **Matinee** actor selected in the Viewport, right-click and create its reference. Then, drag the pin of this node and type in `Play` when the menu opens to create a **Play** node. Attach this to the Then 1 pin of the **Sequence** node.



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.

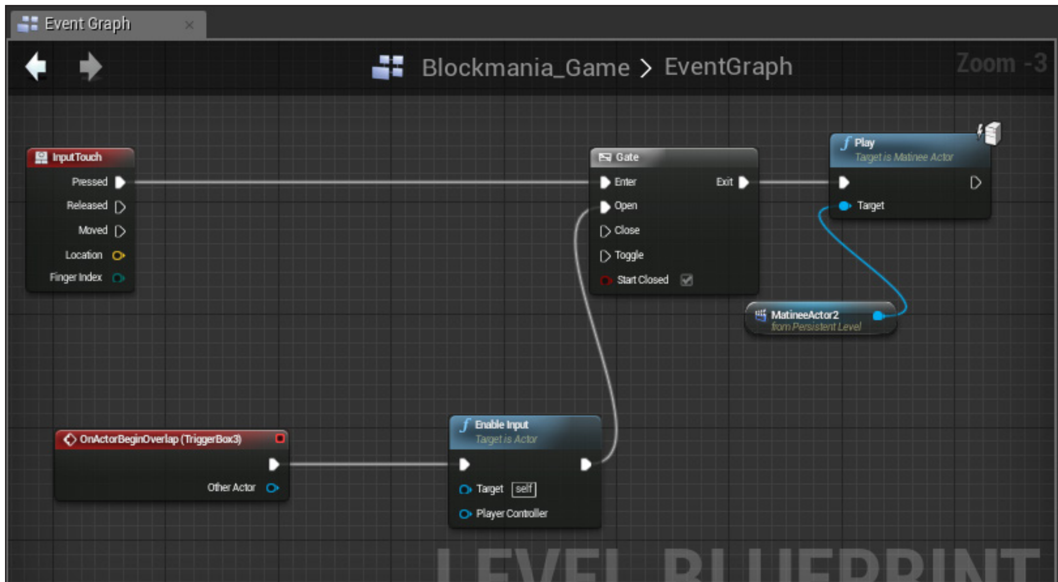
Room 2

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.

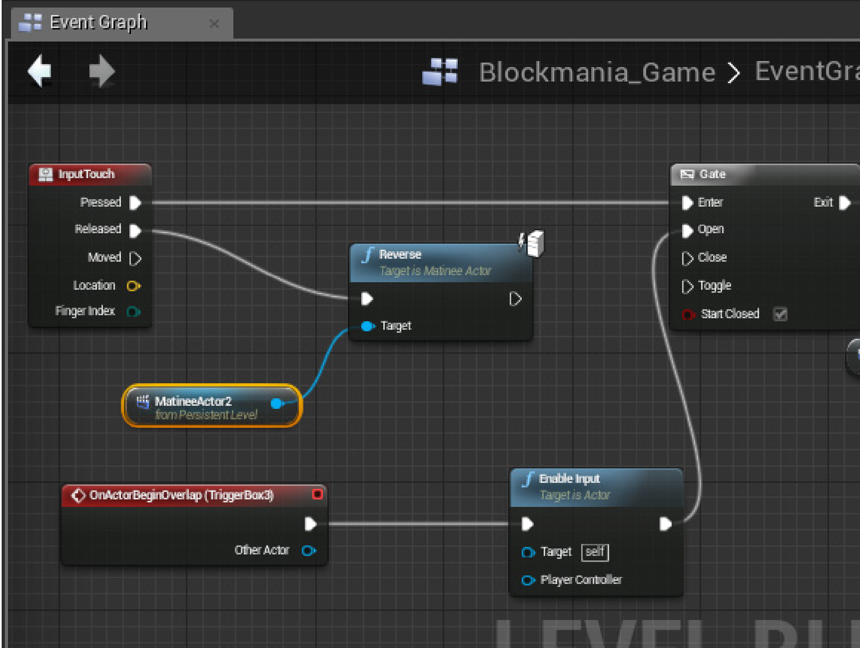
Here, the Matinee part will be similar; we will create a Movement track 2 seconds long and animate the door going upwards. Here, since we are not creating a cut-scene, we do not need a **Camera** group or a **Director** group, neither do we need to disable the player input while the matinee is running. 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 the screen. Here, when the player stops touching the screen, the door will close, that is, the **Matinee** will play in reverse.

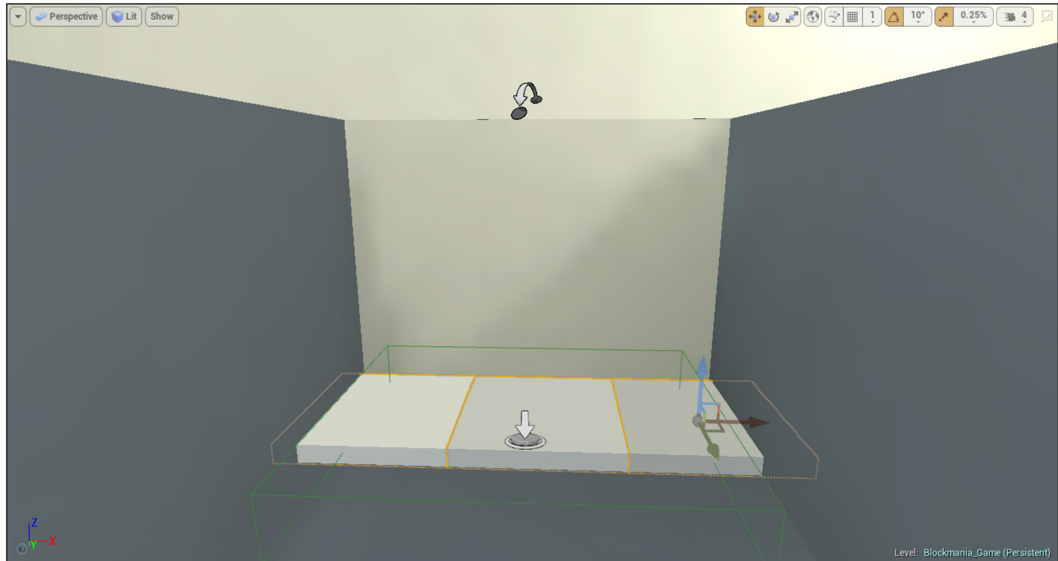
To do so, create another reference of the **Matinee** actor (or just duplicate the one already there), create a **Reverse** node, and attach the **Matinee** actor to its **Target** input.



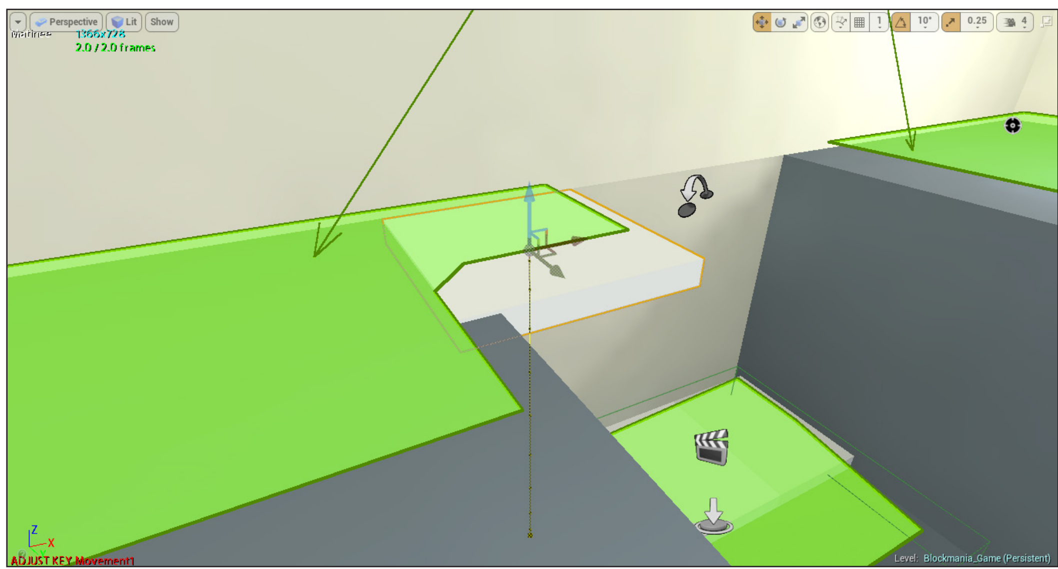
Upon testing the game now, you will see that when you place the key cube on the first pedestal and click on the door (when playing in the Editor, the mouse button acts as a touch input), the door opens; as soon as you let go, it closes again. Do the same with the doors in room 4, but you do not need 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

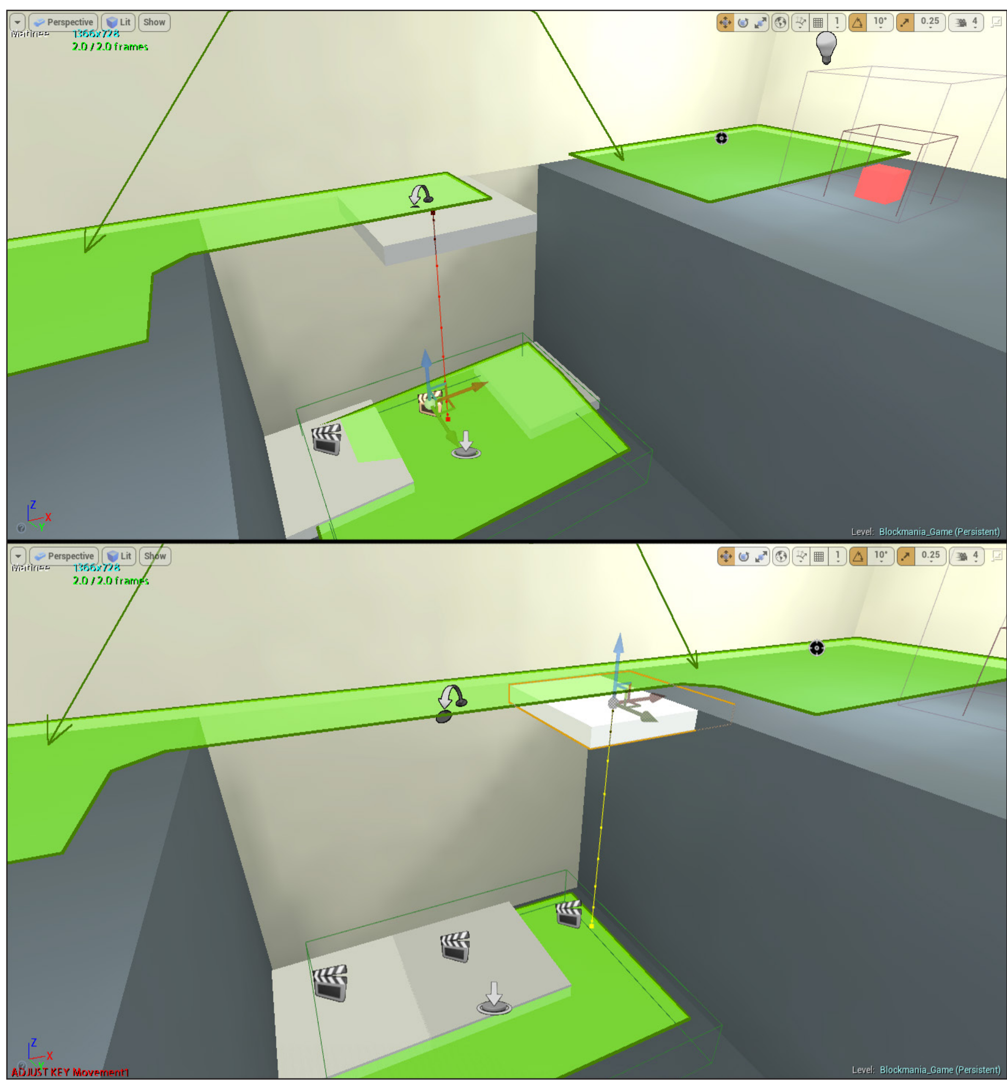
In the room with the AI character, the player has to make a bridge for it, so that it can cross the pit and unlock the key cube. There are switches that draw a segment of the bridge. The player has to quickly draw all of the segments of the bridge before the AI character falls it into the pit. In the 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**.



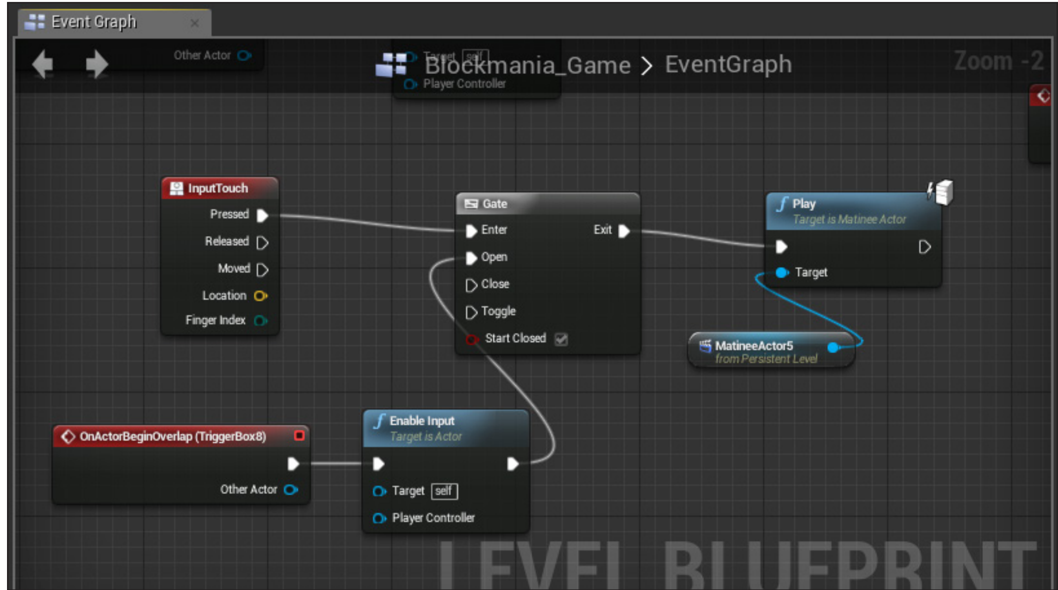
After doing this, add three Matinee actors, one for each section. Then, in Matinee, set the animation time as 1 second and animate the section coming up by moving it so that it is aligned with the ground. Make sure that the section is perfectly aligned. If it is too high, the AI character will not be 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:



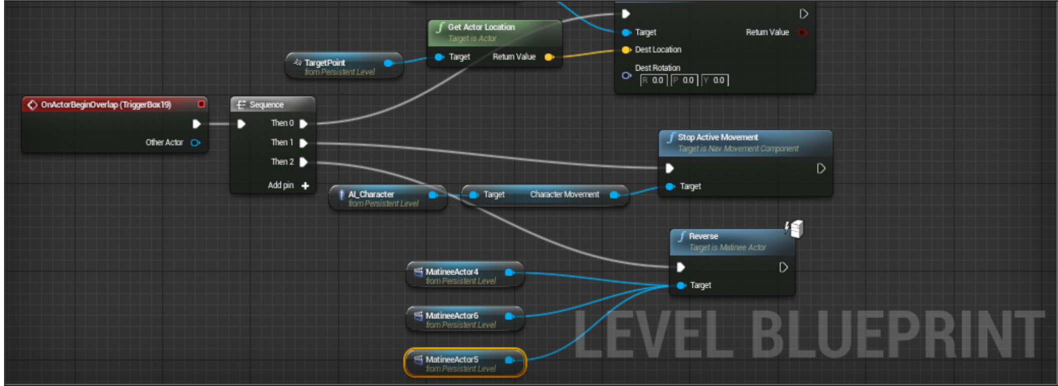
Do the same with the remaining two sections, ensuring that they fit perfectly when they are drawn.



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.

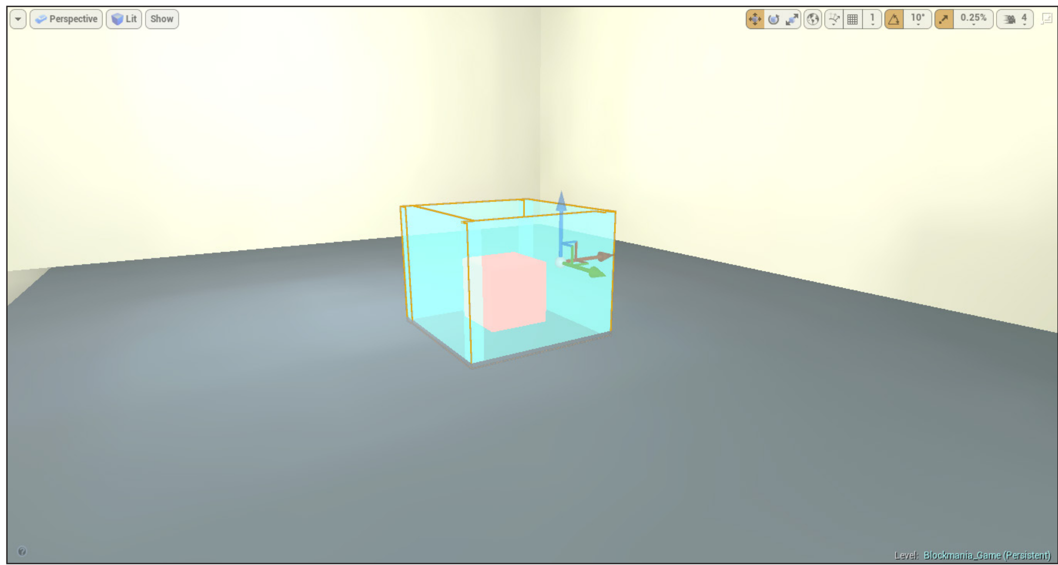


When the AI character falls into the pit, we want the sections of the bridge to go back down to the bottom of the pit, so that when the player starts over, they have to press the switches again to draw the sections. Go back to the trigger box which teleports the AI character back to its starting position. 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.

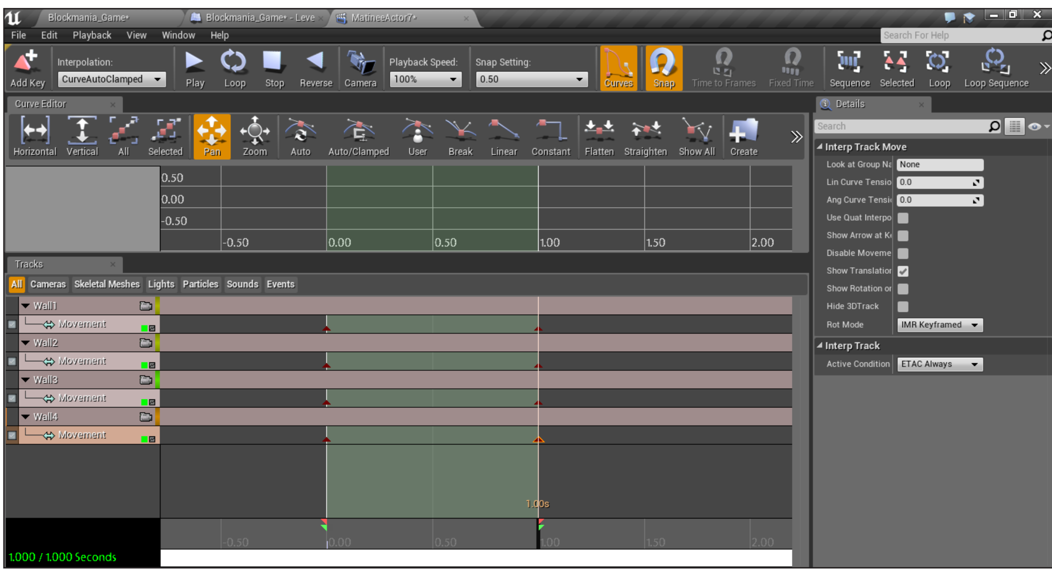


Now if the AI character falls into the pit, apart from being teleported back to its starting position, the sections of the bridge will also reset to their original position.

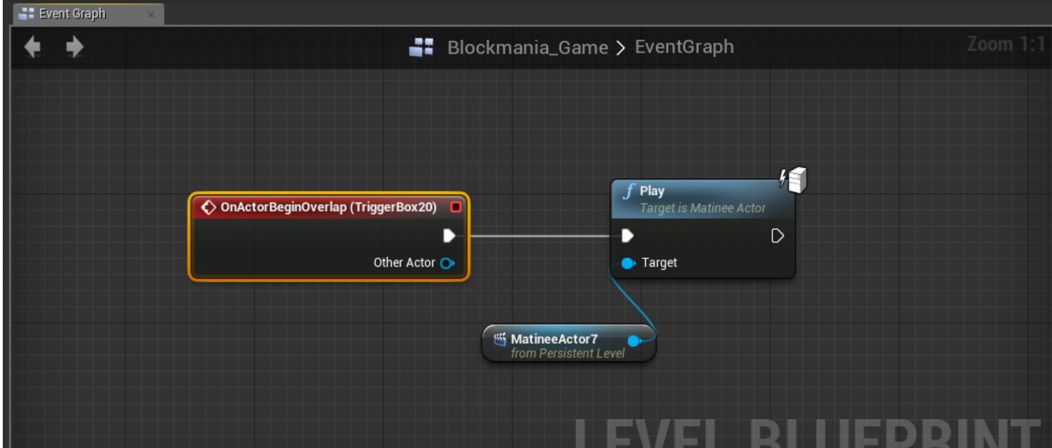
We have our puzzle set up. However, currently the puzzle has no payoff. We had mentioned earlier that when the AI character crosses the pit, it unlocks the key cube. But right now, our key cube does not seem to be locked. What we can do is encage our key cube, which opens when the AI character successfully 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.

7

Finishing, Packaging, and Publishing the Game

In the last few chapters, we have been building a 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.

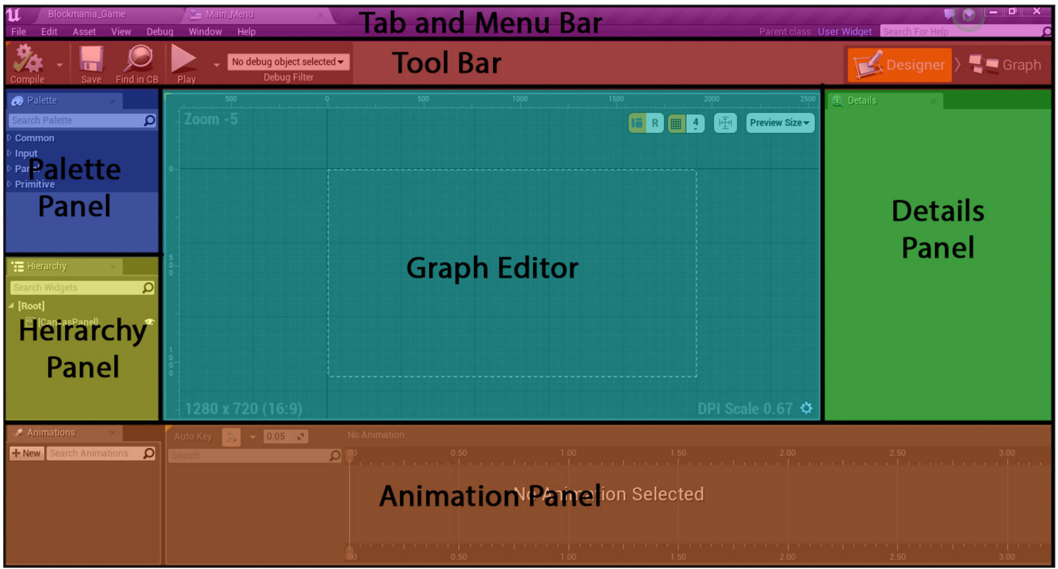
We now have a functioning game. The next step is finalizing the game, packaging it into an `.apk` file, and publishing it to the Google Play Store.

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.

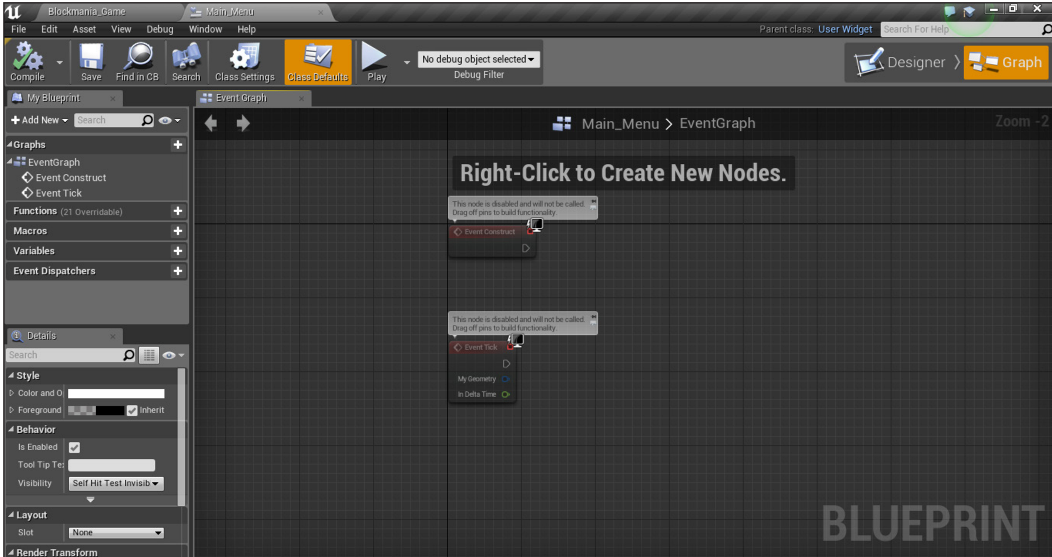
- **Edit**: Here, you can undo or redo the last action and open the **Editor** and **Project Preferences**.
- **Asset**: From here, you can find any asset you have selected in the **Content Browser** as well as find said asset's references in the widget.
- **View**: Here, you can choose to hide/unhide 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

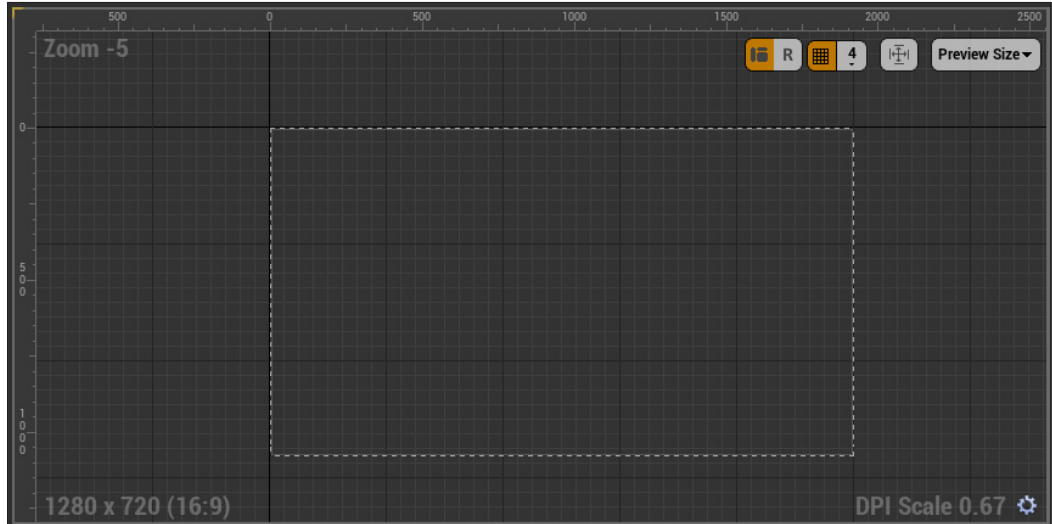
There are two windows in the UMG Editor: **Designer** and **Graph**. The **Designer** window is the window you see when you open the widget class. This is where you design your user interface. This includes adding the UI elements, arranging them in the Graph Editor, setting their properties, and so on. 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.

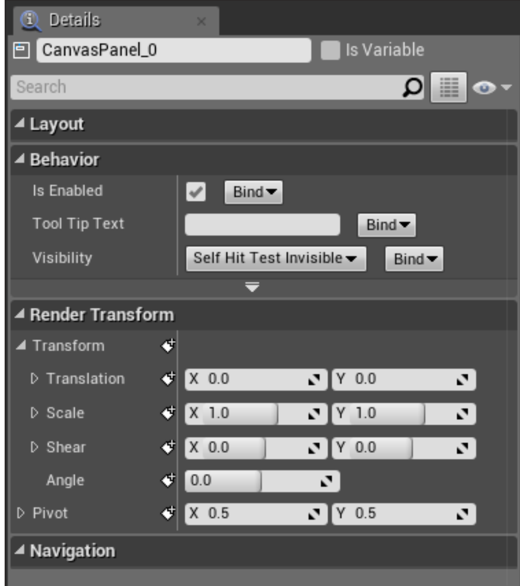
At the bottom-left corner of the Graph Editor, you can see the resolution of the screen. At the bottom-right corner of the screen is where the **Dots per inch** (**DPI**) scale is displayed. You can change this by clicking the gear icon next to it.

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

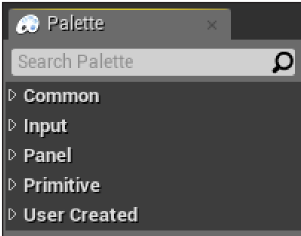
The **Details** panel is where you can set the properties of the selected component.



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.

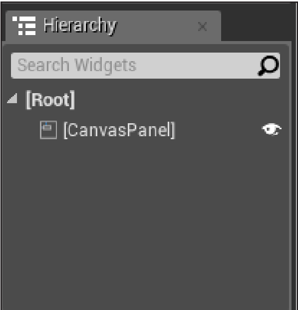


The widget components are categorized into four groups, namely **Common**, **Input**, **Panel**, and **Primitive**:

- **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

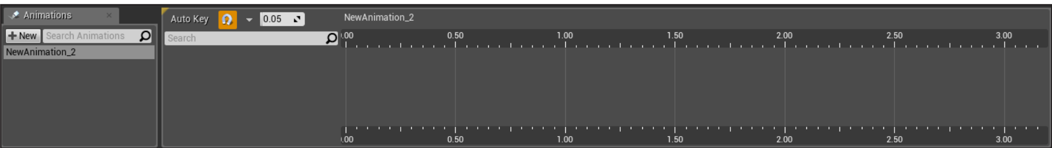
The **Hierarchy** panel shows the hierarchy of all of the components of the widgets.



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.

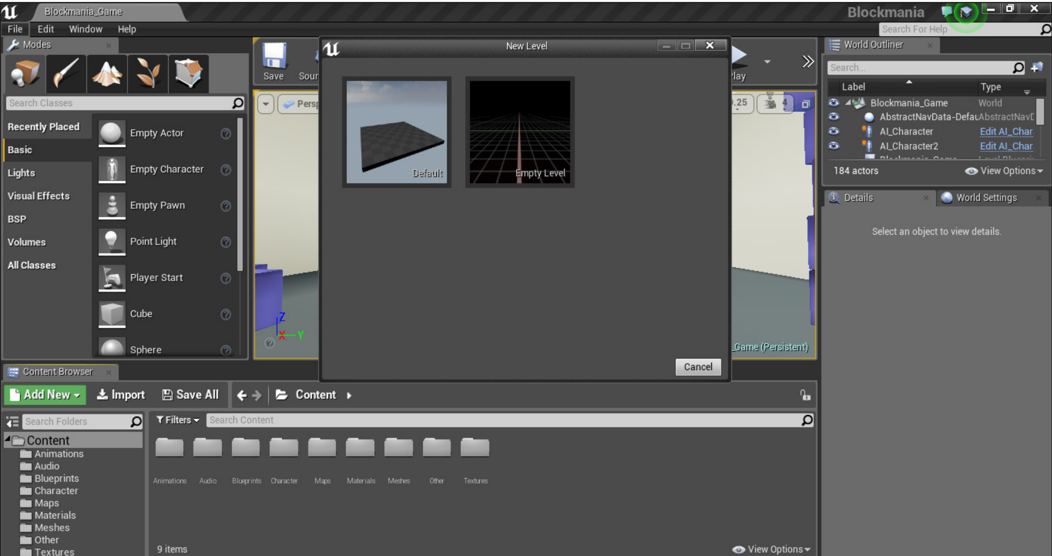


UMG allows the user to create animations. You can animate the Widget components to suit your preference. The **Animation** panel works almost the same way as Unreal Matinee. On the left is the **Tracks** panel. This is where you can add, remove, or move animation tracks. To add an animation track, click on the **+ New** button.

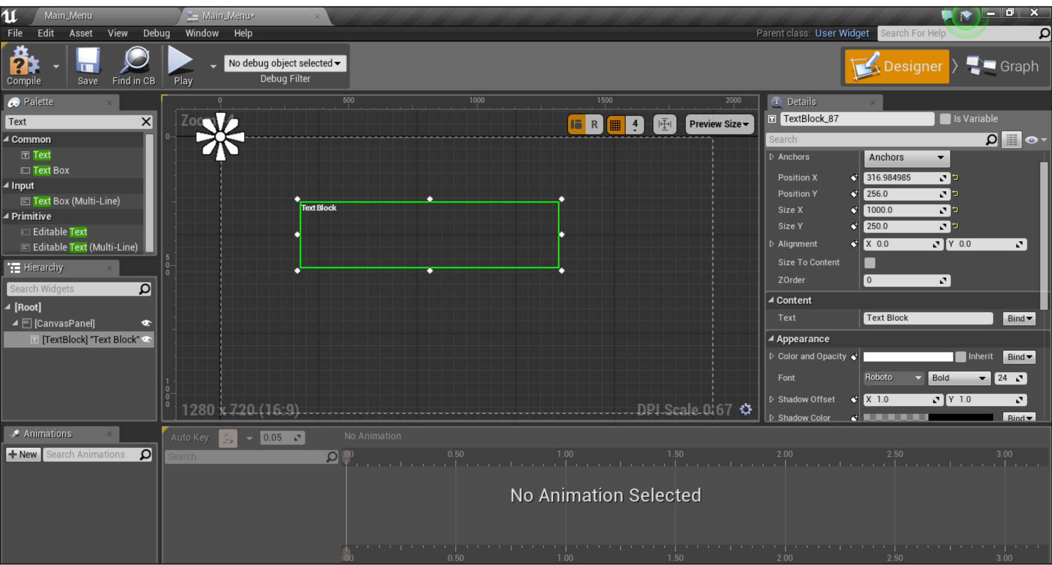
On the right is the animation timeline. This is where you create your animation using key frames. At the top-left corner are some settings, such as enabling/disabling grid snapping, the grid snap value, and so on.

Creating the main menu

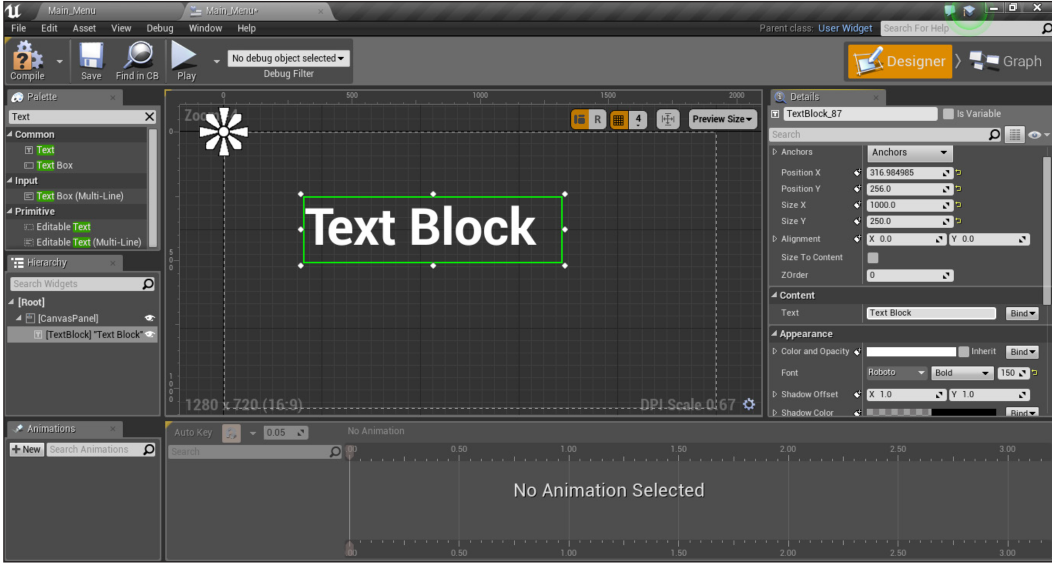
The first step to creating our main menu is to create a new level. In the Editor, click on **File** and select **New Level**. When the Level Type window opens, select **Empty Level**.



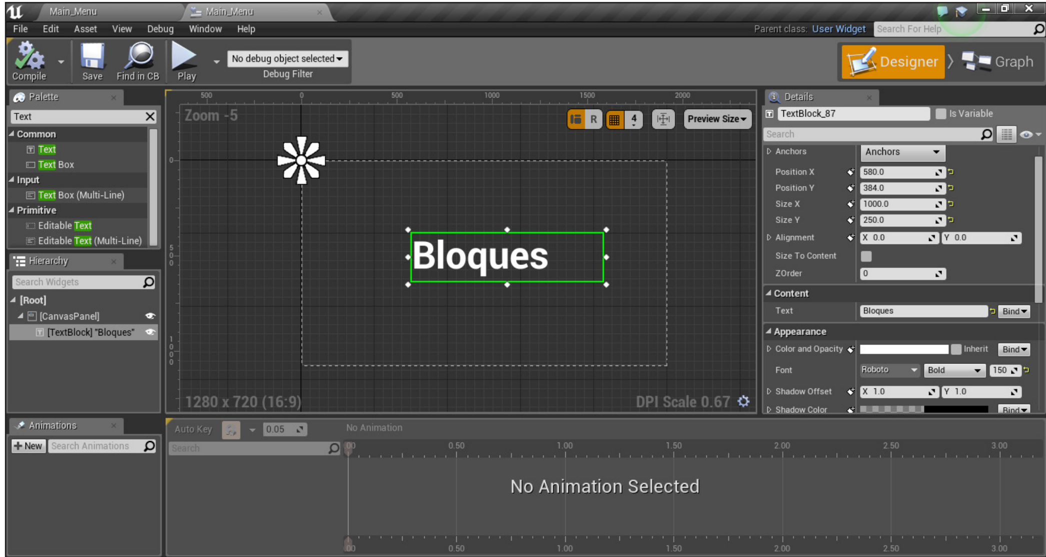
Name this level as **Main_Menu**. Once created, open the **Main_Menu** widget class in the **Content Browser**. In the UMG Editor, add a **Text** widget to the canvas. Make the text big. First, increase the size of the text slot panel to 1300 x 200. There are two ways of doing this: the first way is 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`.



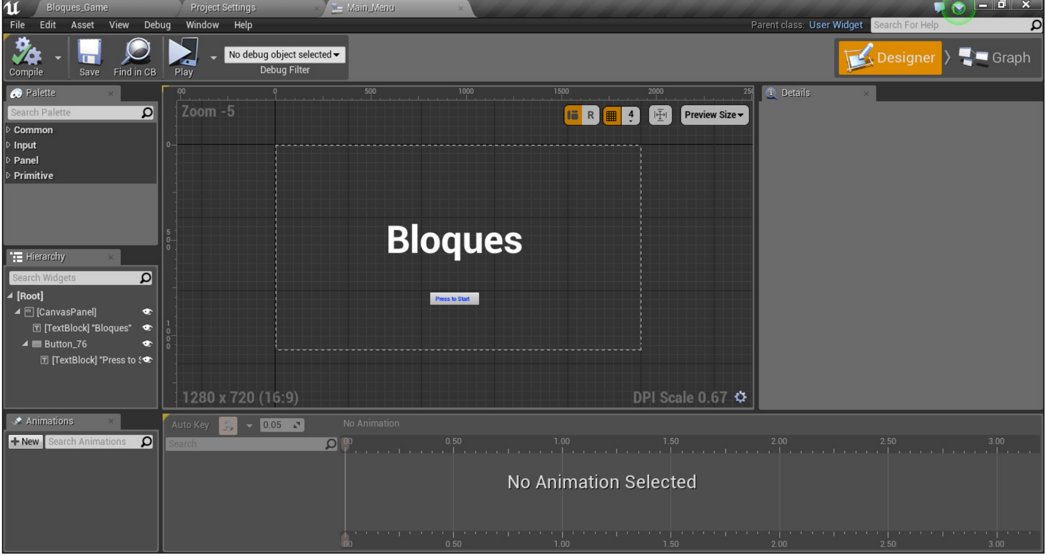
The text slot is now set. However, the actual size of the text itself is quite small. The next thing we need to do is to increase the size of the text. Again in the **Details** panel, under the **Appearance** section, you will find the **Font** option. At the far right, you can set the value of the font size. By default, this value is set as `24`. Increase this to `150`.



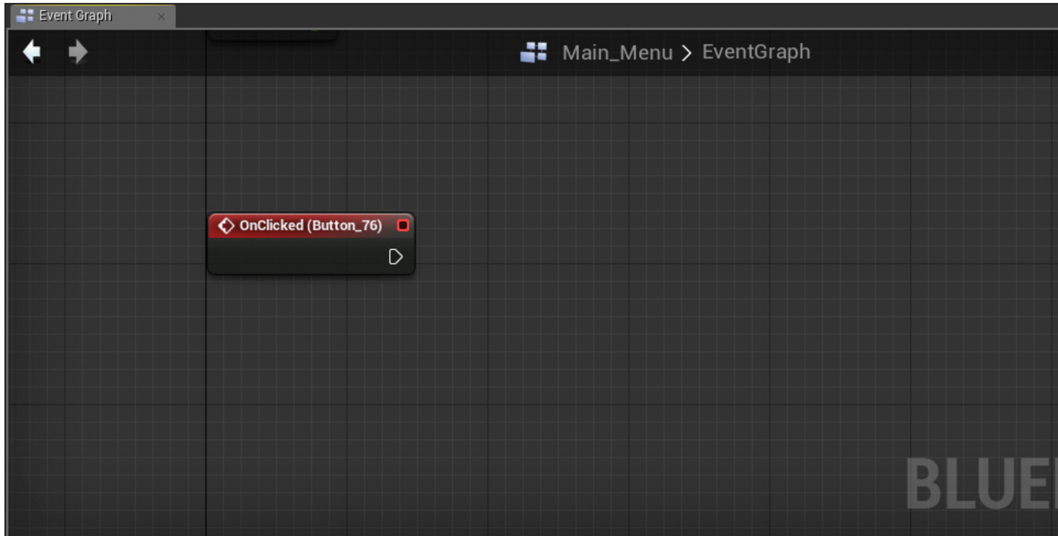
This is where we will display the name of the game. In the **Content** section, you can set what text you want printed on the screen. Whatever you write is displayed in the Graph Editor. Remove **Text Block**, and write the name of the game: `Bloques`. Finally, adjust the position of the **Text** panel so that the name appears in the center of the screen.



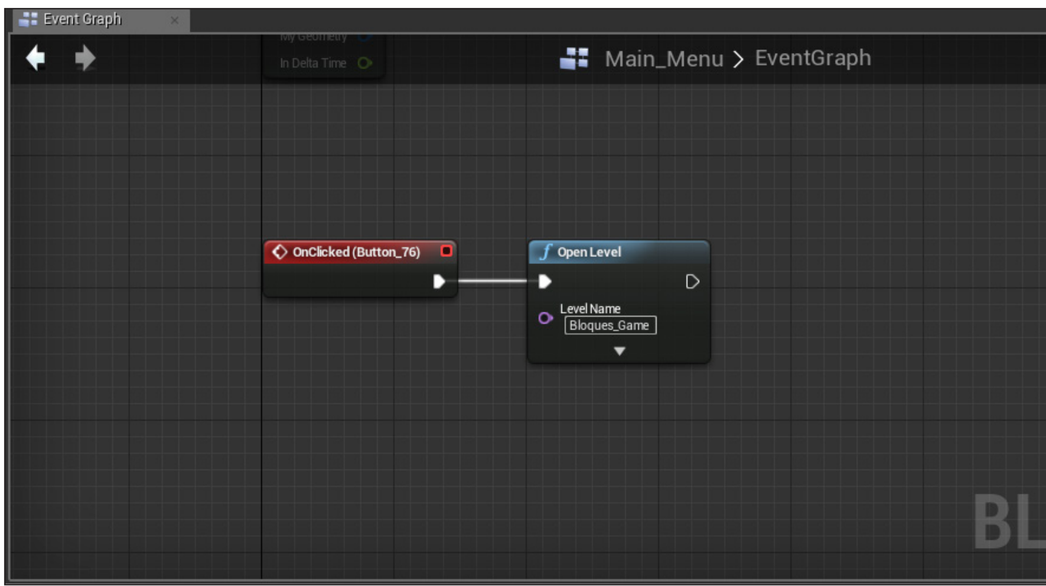
The next thing we are going to add is a button which, when the player clicks, starts the game. To do this, add a **Button** widget from the **Common** section in the **Palette** panel. Place it below the name of the game in the center. Set its dimensions as `256 x 64`. We will also need some text to 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.



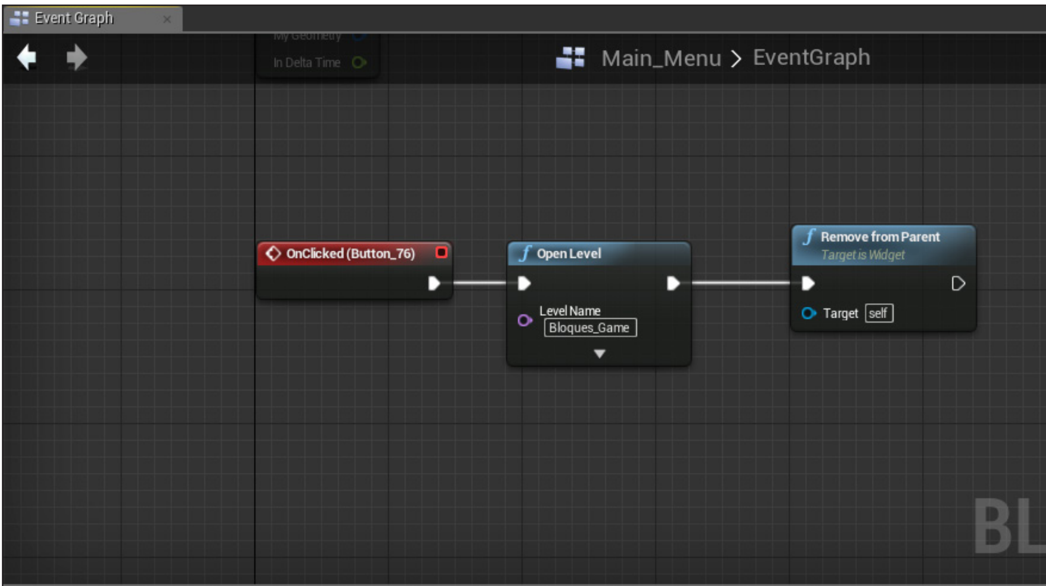
We have our button in place but it does not do anything. We need to script its functionality in. With the button selected, go to the **Details** panel and under the **Events** section, click on the button next to **OnClicked**. The window will switch to the **Graph** window with the **OnClicked** button 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.

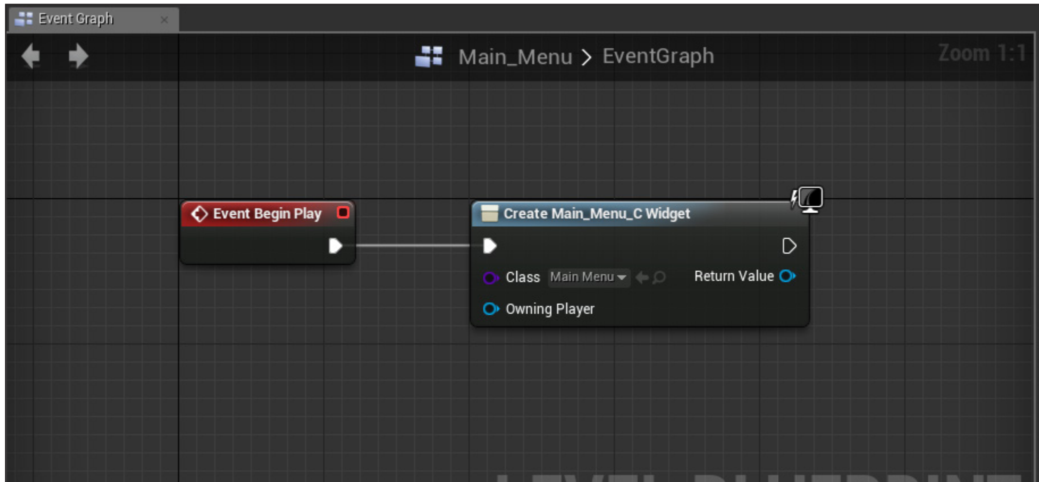


Finally, we also need to remove the widget; otherwise, it will still be there when the level loads. Right-click and select the **Remove from Parent** node and connect it to the output pin of the **Open Level** node.

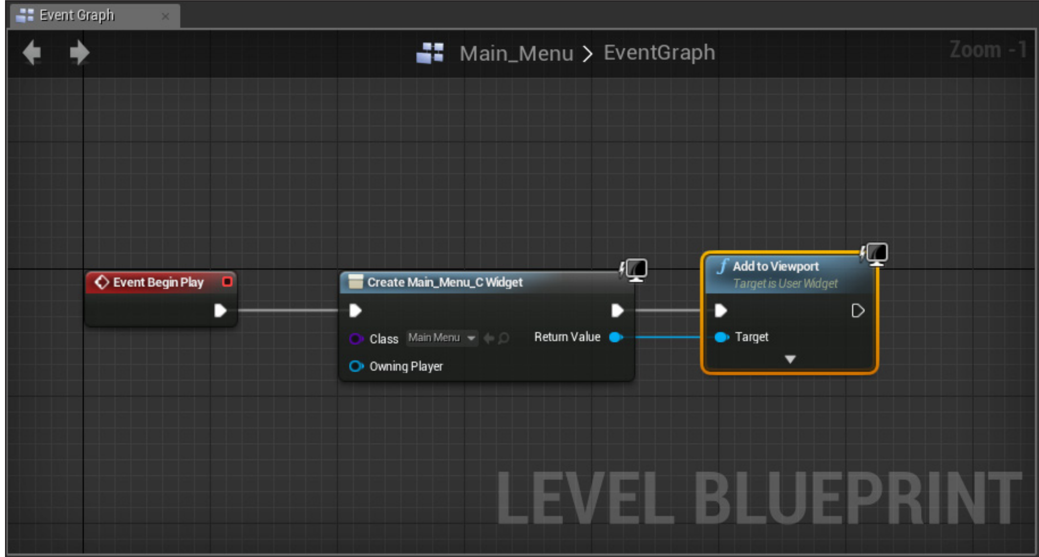


That is it. It really is that simple to load levels via Blueprints.

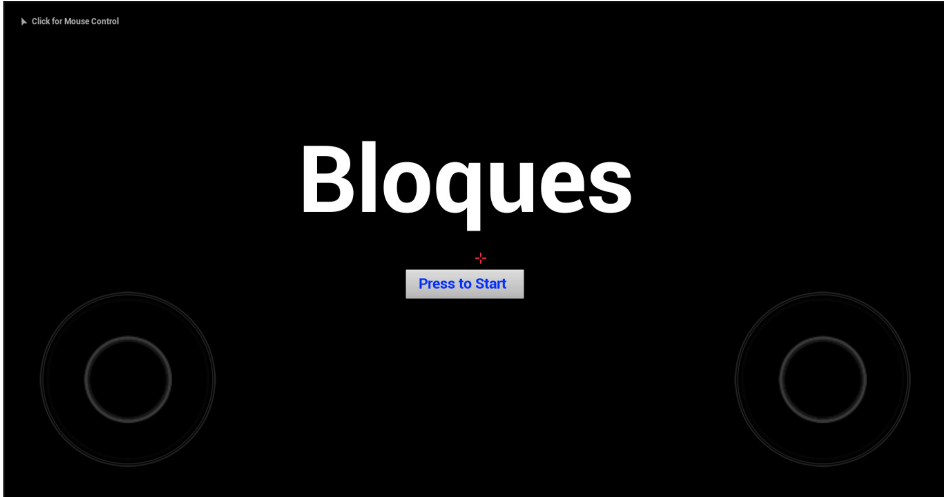
We have our Main Menu set up. We have to implement it into our scene. For this, open **Level Blueprint**. Before we can display the widget, we first need to create it. In the **Event Begin Play** section, right-click and type `create widget` to find the Create Widget node and connect them. Next, in the **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.

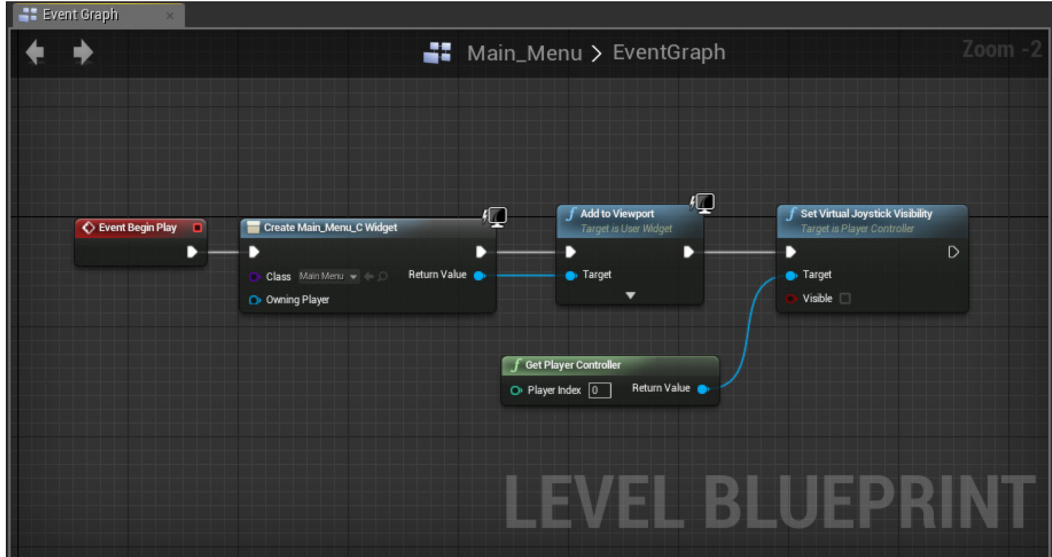


You were to test out the game, you would see that as soon as you run the game, the name of the game and the button appear on the screen.

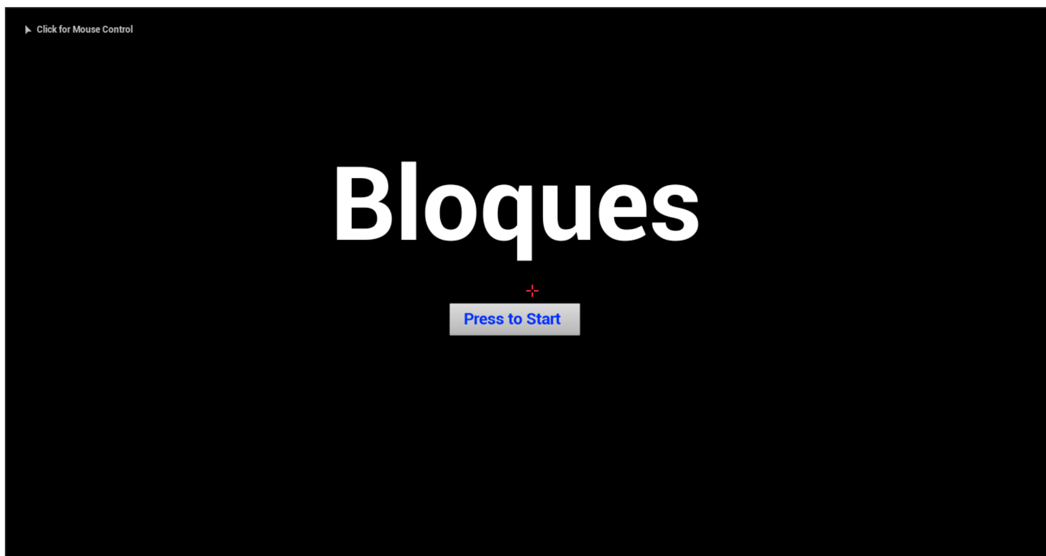


However, there are still a few things we can improve. For starters, we can hide the virtual joysticks, since they are not required.

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.



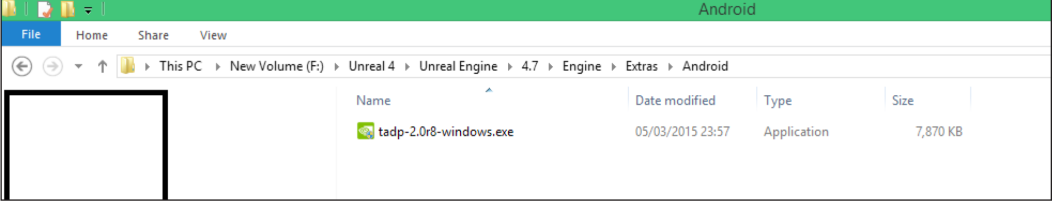
Now run the game. The virtual joysticks will not be visible during runtime.



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

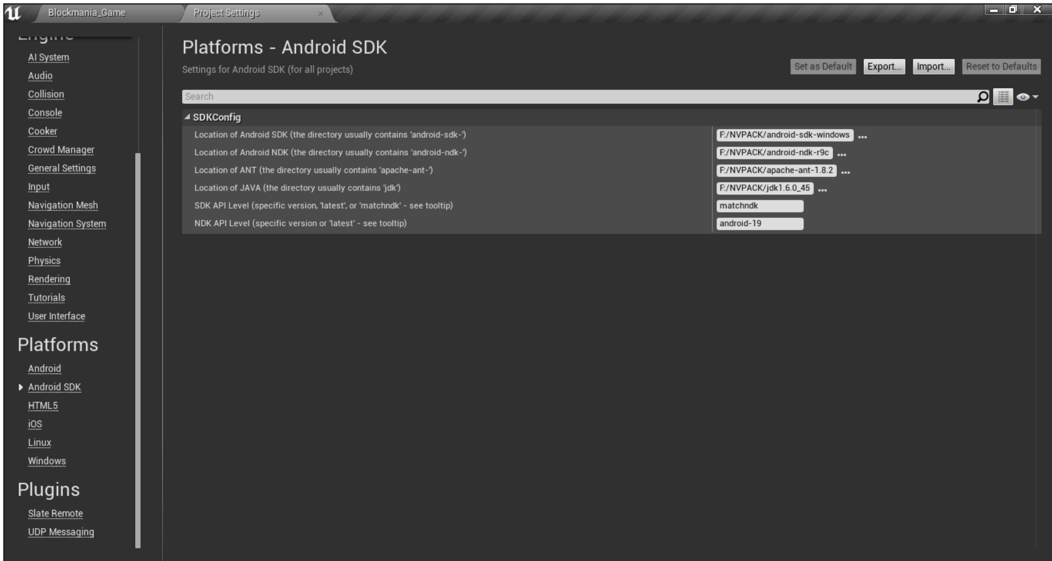
Before we can cook and package our game, we first need to install the Android SDK. Luckily, you do not need to find it on the Internet and download it; its installation file is available with UE4. You can find it in **<Location>\ Unreal Engine\ 4.7\ Engine\ Extras \ Android**. The file we need is `tadp-2.0r8-windows`.



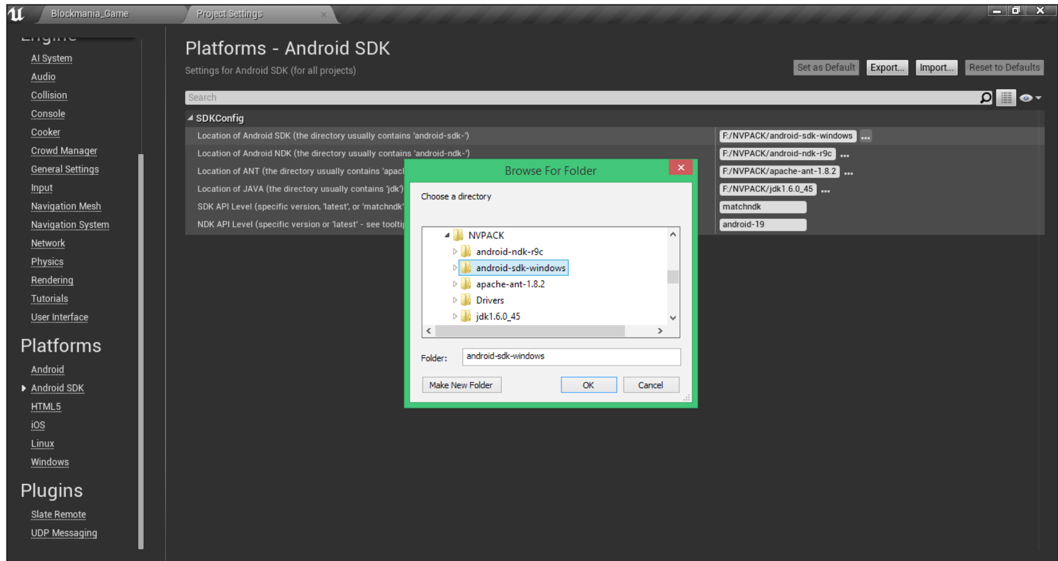
Once you find it, double-click on it to run the installation. Once the setup opens, all you need to do is to follow the onscreen instructions and let it install.

> 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.

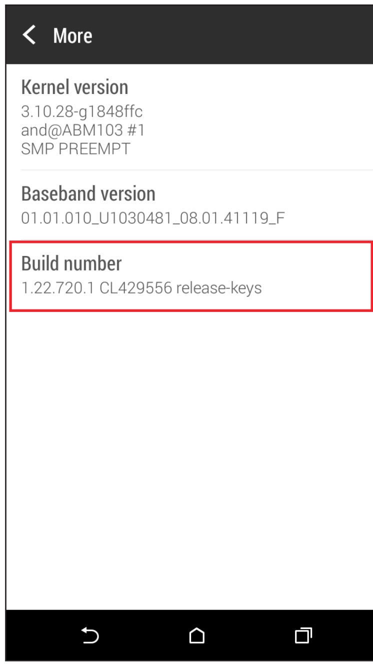
You can have different folder names; you just need to make sure that the appropriate folder is currently assigned.

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.

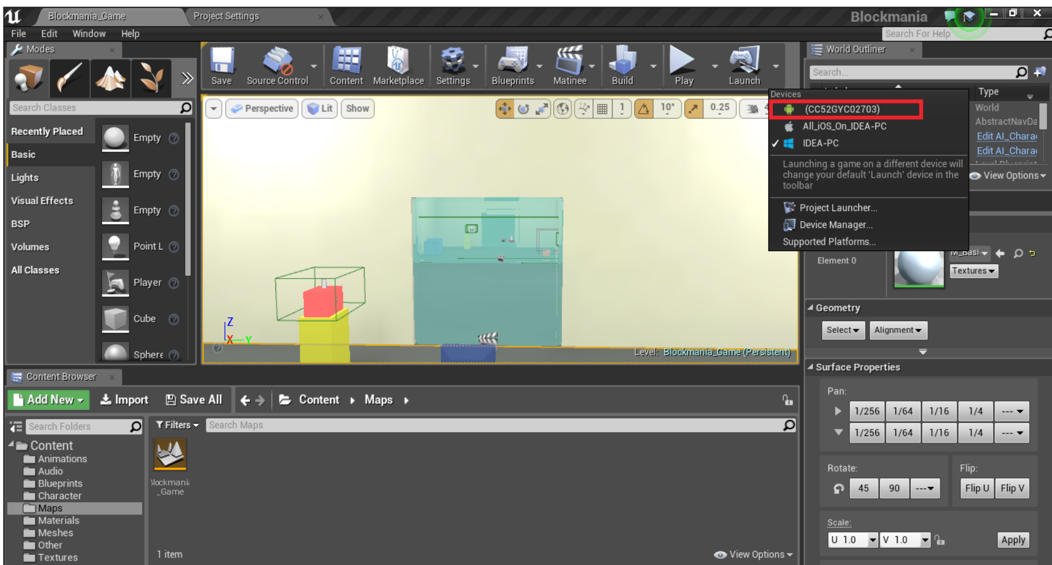
Setting up the Android device

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.

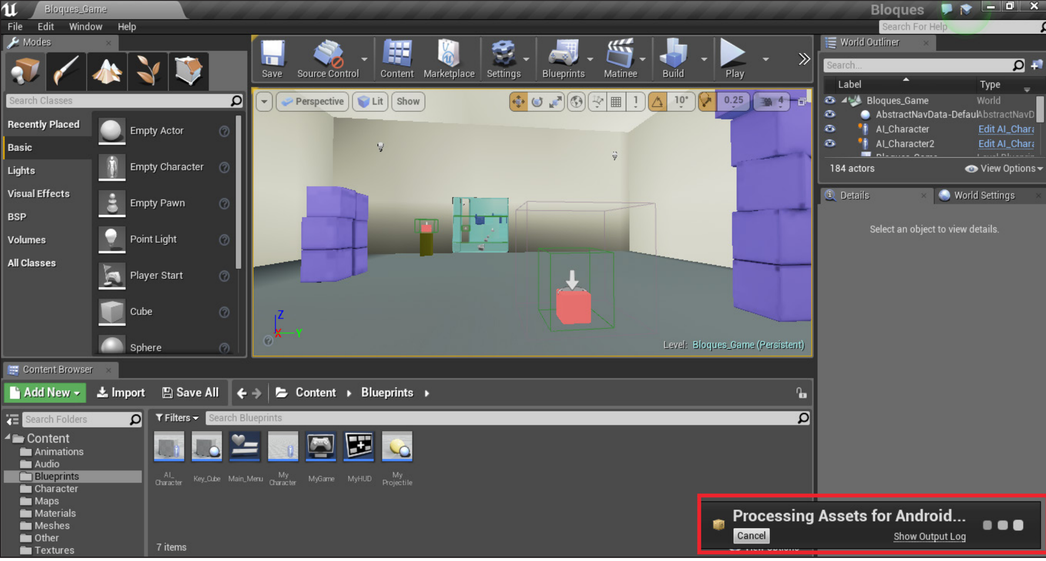
1. To do this, the first thing is to enable the Developer mode on your device. This can usually be found in **Settings** | **About phone**. Here, you need to find **Build number** and tap on it several times. If done correctly, you will get a prompt saying **You are now a developer** (in some devices 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.



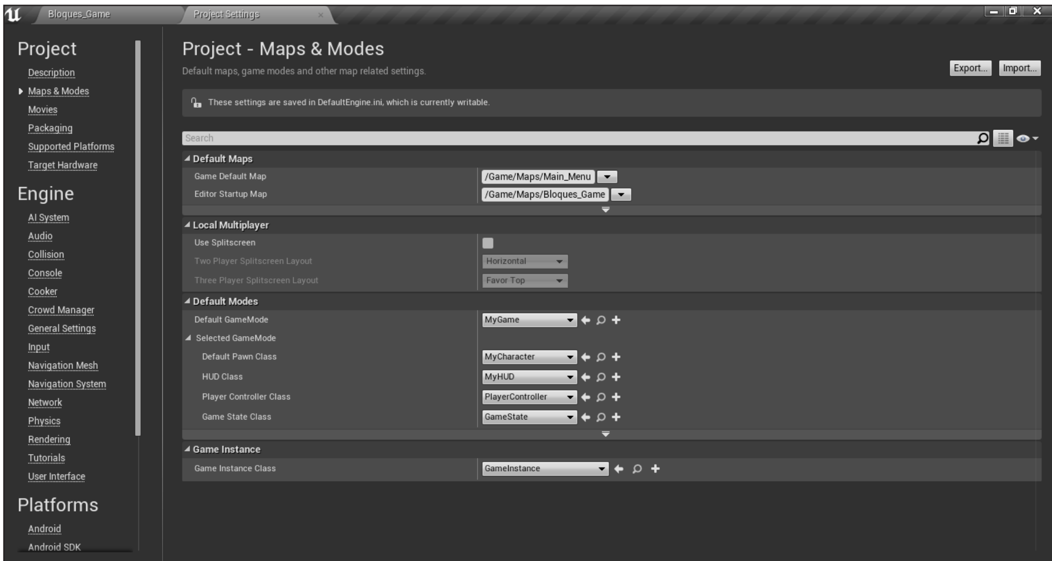
4. To stop the build process, click on the **Cancel** button. To see the output log of the building process, click on **Show Output Log** which opens the log window, where you can see what is being executed. You can also see where the error has happened if one occurs. Once the build process is over, 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**.

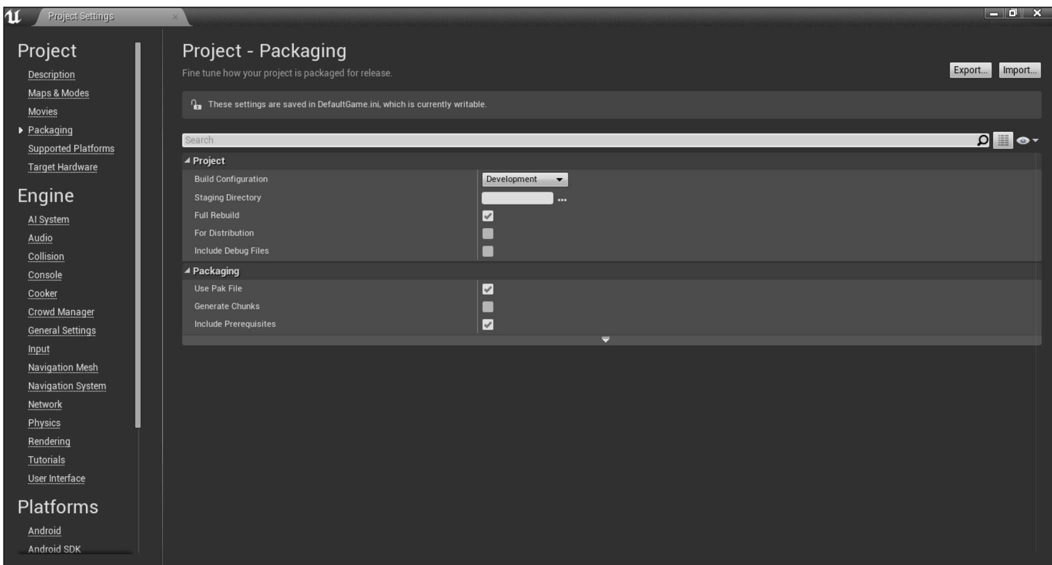


Let's go over the various sections:

- **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**.



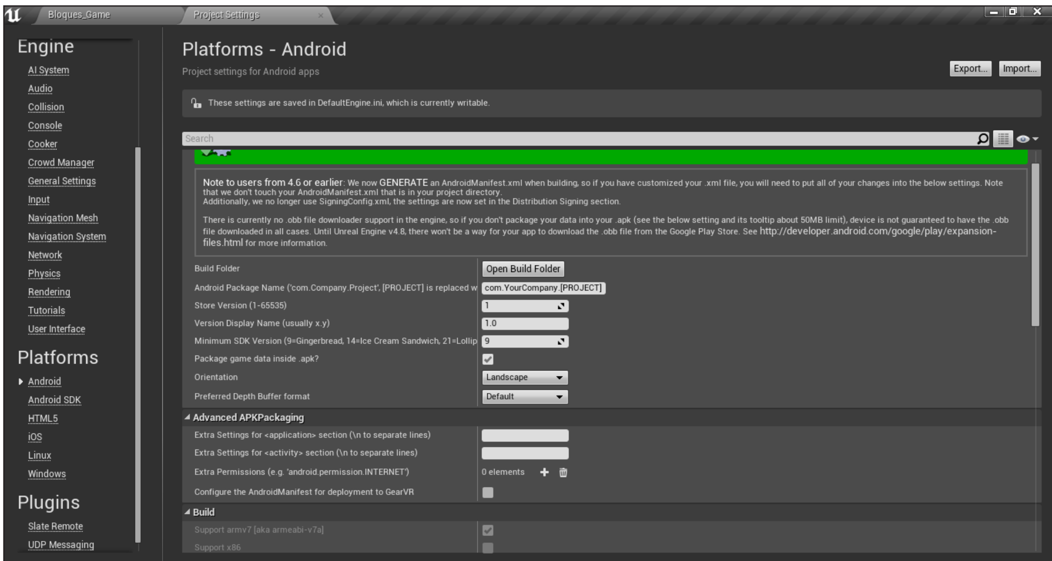
Here, you can view and tweak the general settings related to packaging the project file. There are two sections: **Project** and **Packaging**. Under the **Project** section, you can set options such as the directory of the packaged project, the build configuration to either debug, development, or shipping, 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.

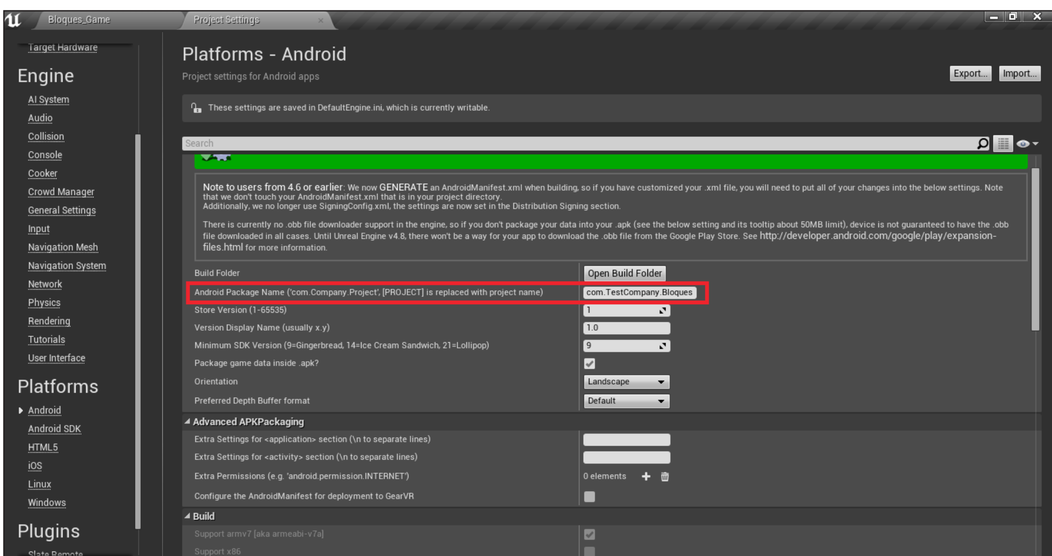


Here you will find all the settings and properties you need to package your game. At the top the first thing you should do is configure your project for Android. If your project is not configured, it will prompt you to do so (since version 4.7, UE4 automatically creates the `androidmanifest.xml` file 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.

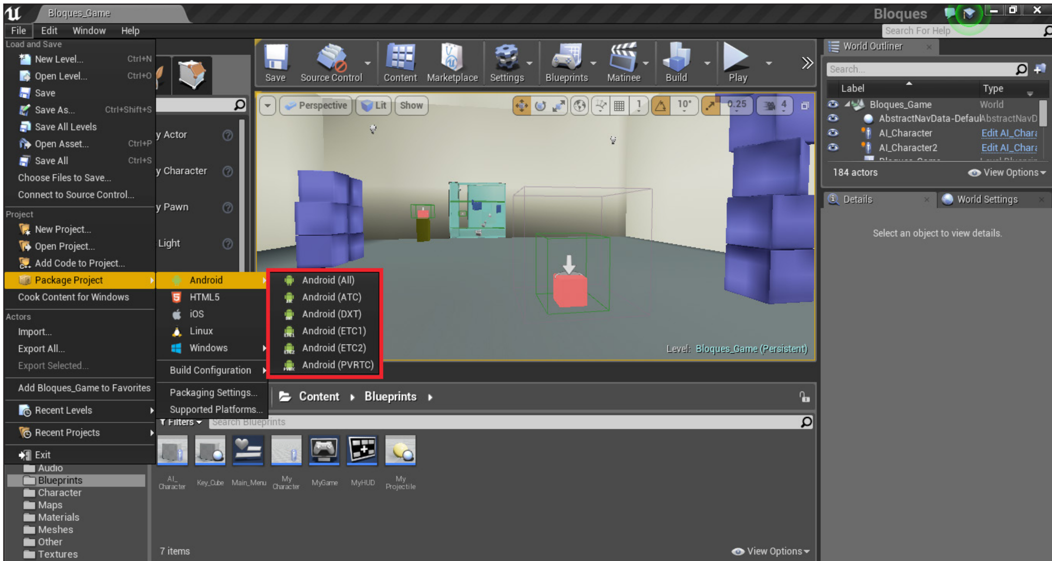
- **Distribution Signing**: This section deals with signing your app. It is a requirement on Android that all apps have a digital signature. This is so that Android can identify the developers of the app. You can learn more about digital signatures by clicking on the hyperlink at the top of the section. 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

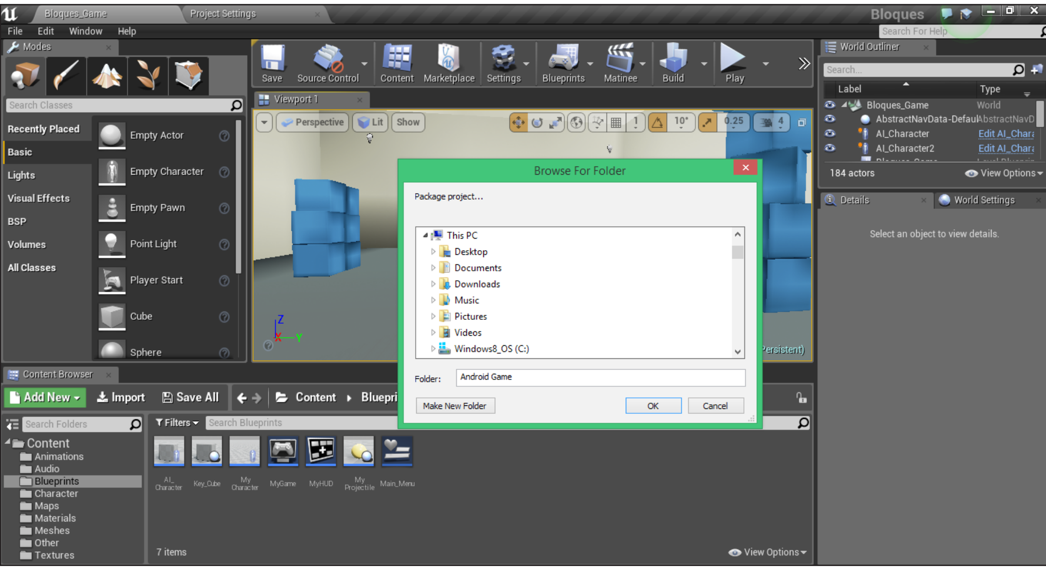
To package your project, in the Editor go to **File** | **Package Project** | **Android**.



You will see different types of formats to package the project in. These are as follows:

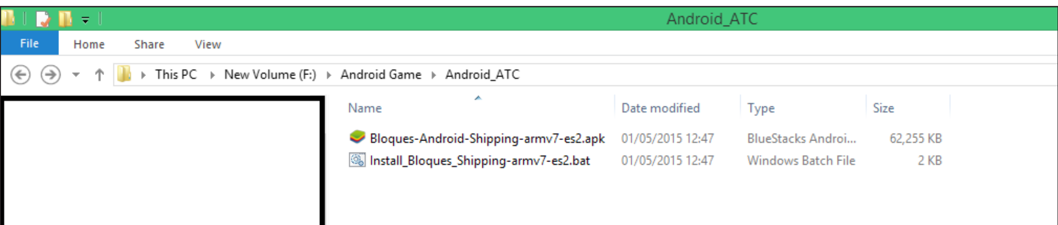
- **ATC**: Use this format if you have a device that has a Qualcomm Snapdragon processor.
- **DXT**: Use this format if your device has a Tegra graphical processing unit (GPU).
- **ETC1**: You can use this for any device. However, this format does not accept textures with alpha channels. Those textures will be uncompressed, making your game requiring more space.
- **ETC2**: Use this format is 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.



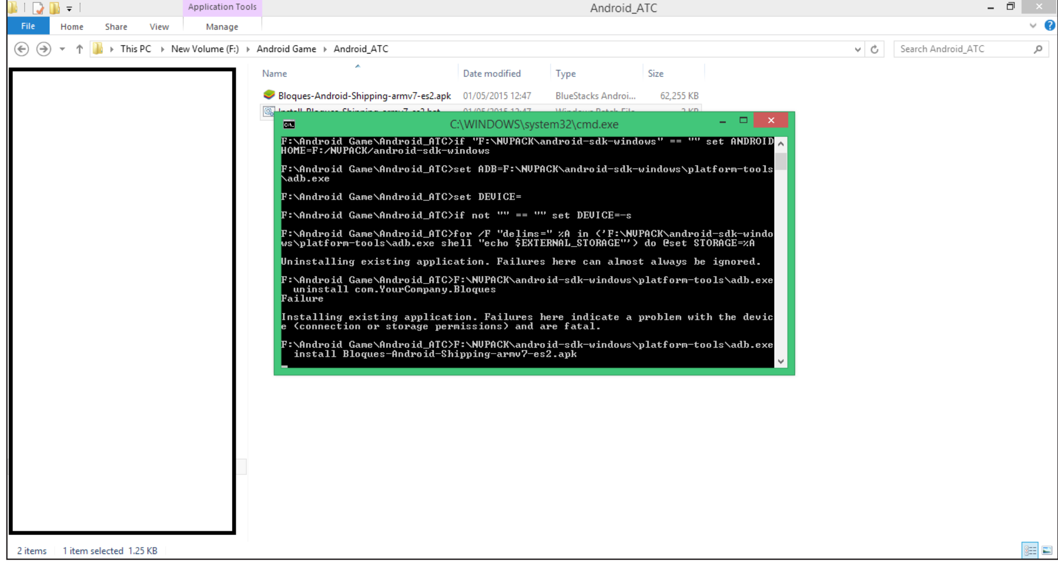
Once you have decided where to store the package file, click **OK** and the build process will commence. When started, just like with launching the project, a small window will pop up at the bottom-right corner of the screen notifying the user that the build process has begun. You can open the output 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.

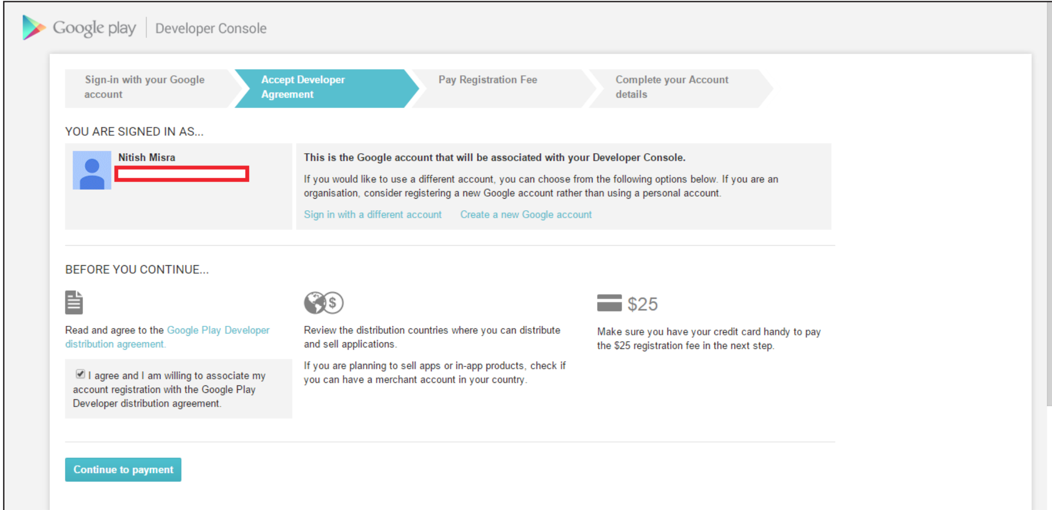
To use the `.apk` file, you will have to do things a bit differently. An `.apk` file installs the game when it is on the device. For that, you need to perform the following steps:

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.

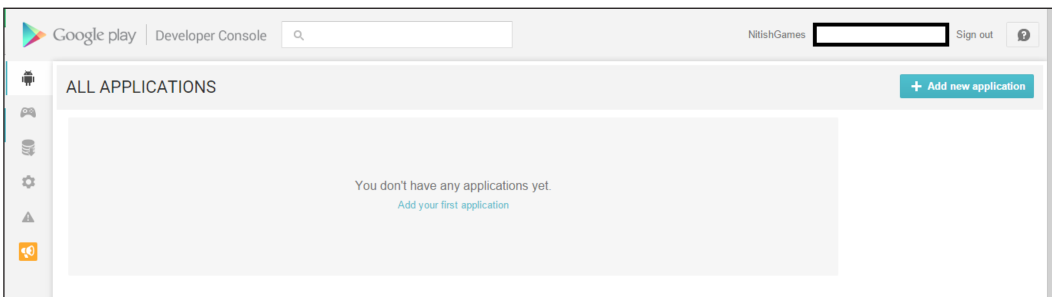
The installation process will begin. Once completed, you can play the game.

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.

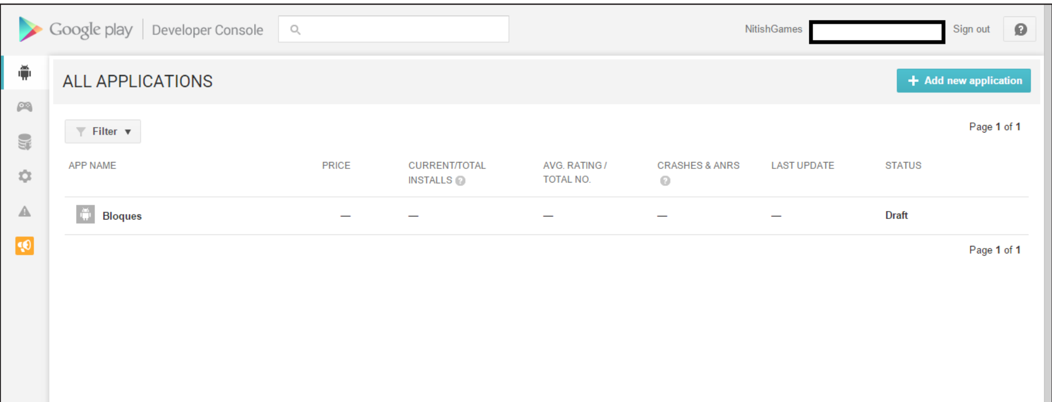


Here, click on the **Add new application button** located at the top-right corner. When clicked, a window will open where you can set the language and set the name of the game as `Bloques`. Once set, click on **Upload APK**. Once clicked, you will see several options regarding the app.

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

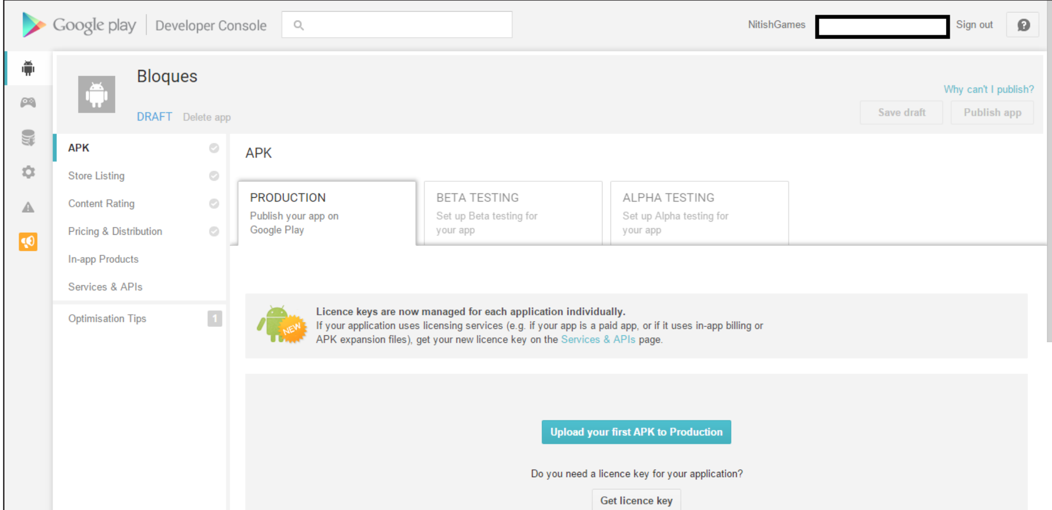
Here is where all the apps and/or games you have published, or are in the process of publishing, are displayed.



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/installs 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

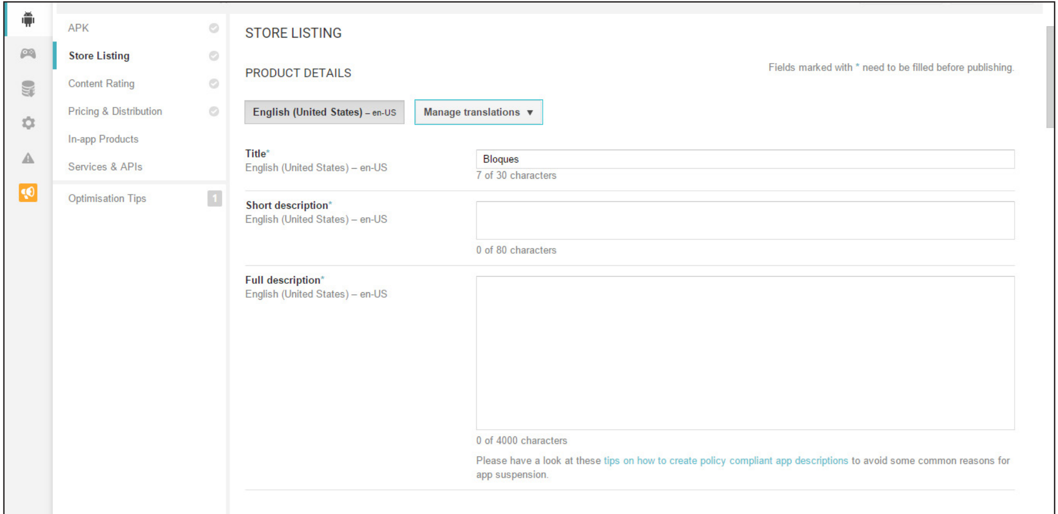
The **APK** panel is opened by default. Here you can choose whether you want to publish the game on the Play Store.



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.



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.

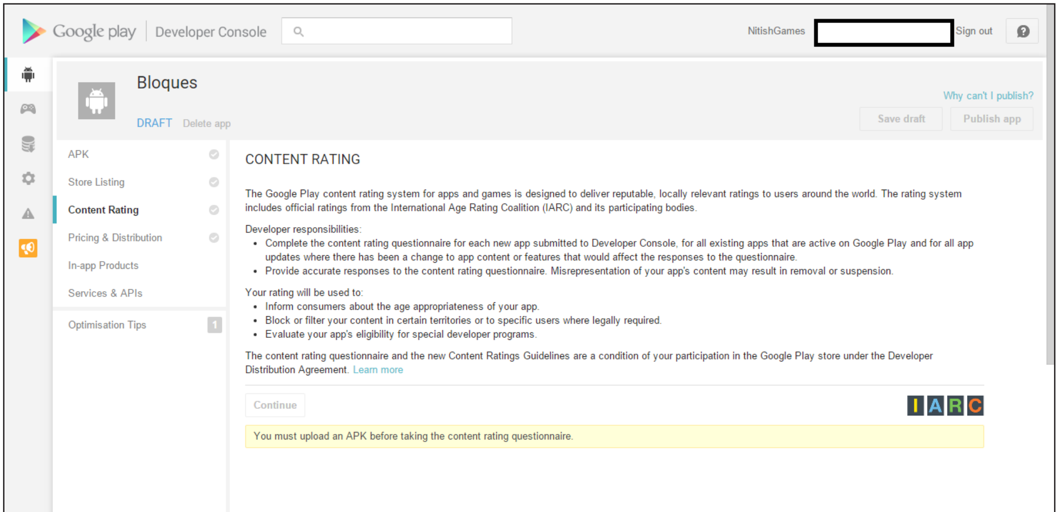
Following **Graphic Assets** is the **Categorization** panel. This deals with how your app will be categorized in the Play Store. Here you need to specify whether your product is an app or a game. After setting that, you will need to specify in what category your product lies; if you have picked **Applications**, 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

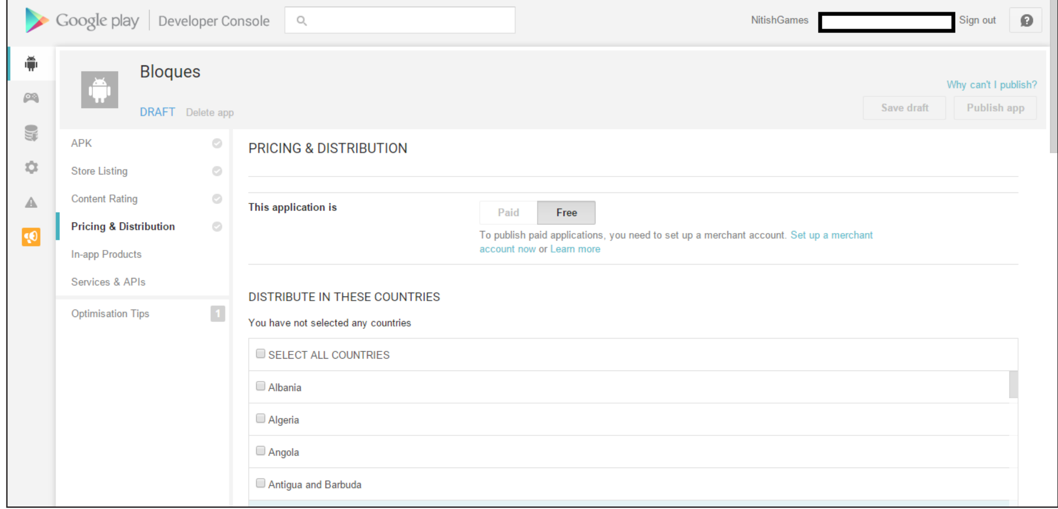
Content rating is an important aspect of any game or app. Google Play also offers its own content rating.



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.

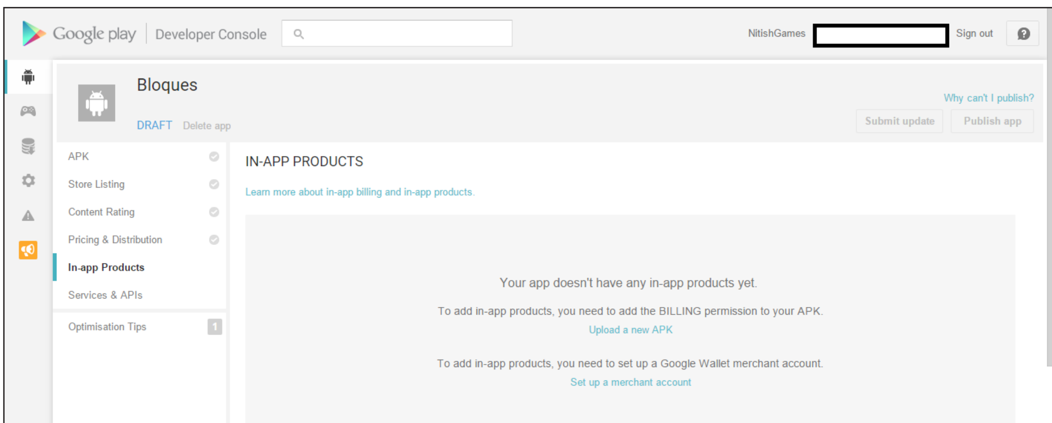


At the top, you can set whether your app is **Free** or **Paid**. If you want your app to be paid, you will first have to set up a merchant account.

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

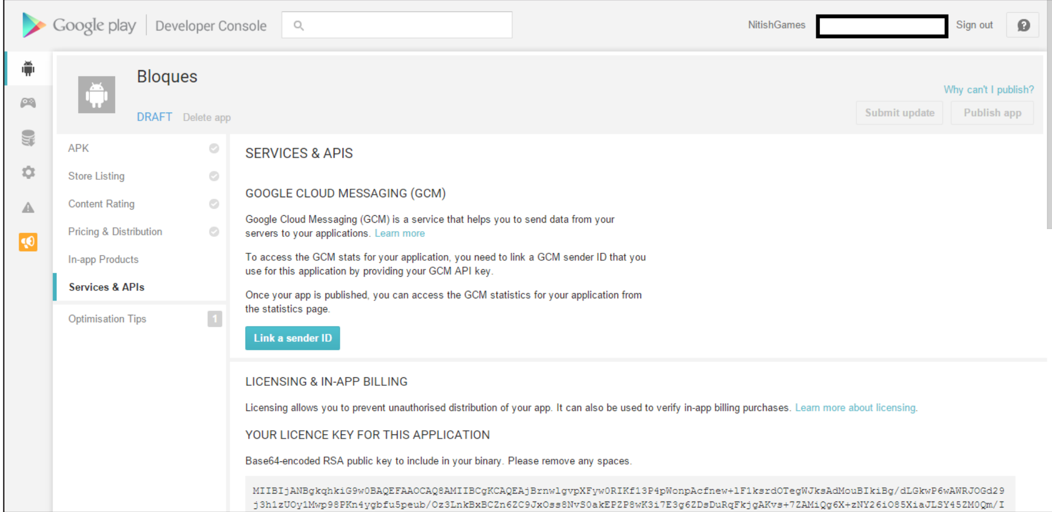
This section contains settings regarding in-app products.



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

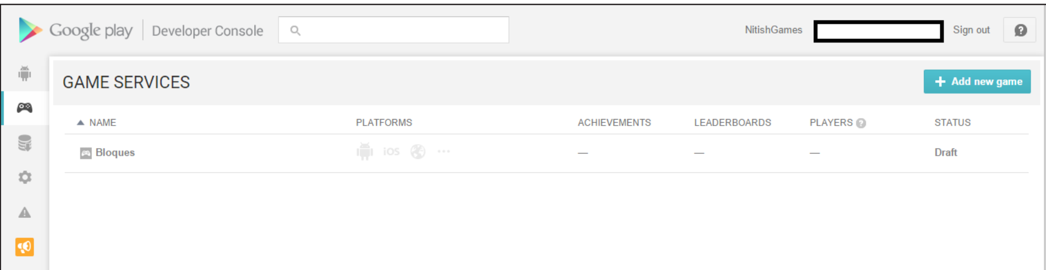
This is where you can see what services offered by Google are currently active in your app/game.



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.

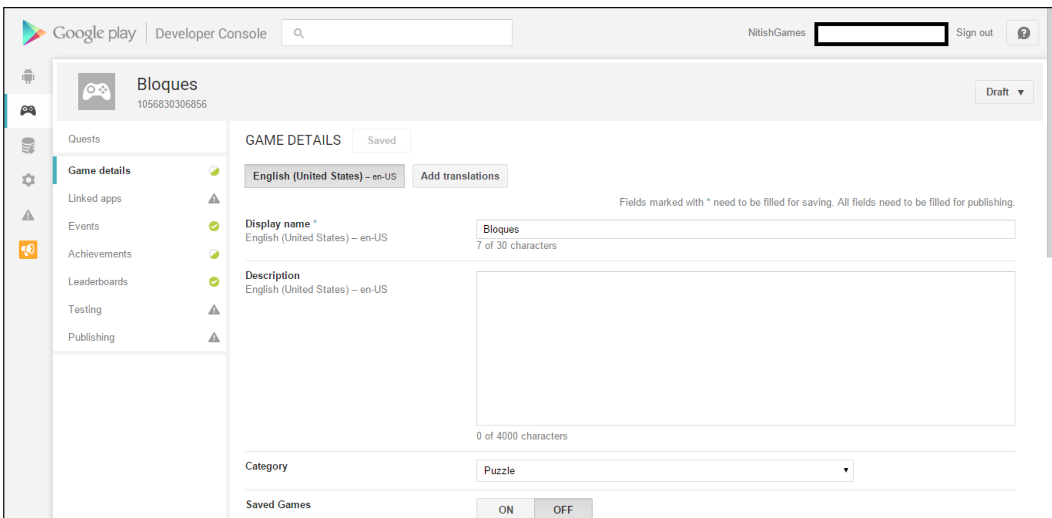


When you go to this panel, you will see a list of apps that you have produced, similar to that in the **All Applications** panel. Here, you can view what platforms your app or game is on, what achievements you have in your game, your leaderboards, and how many unique players have signed in to your game 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

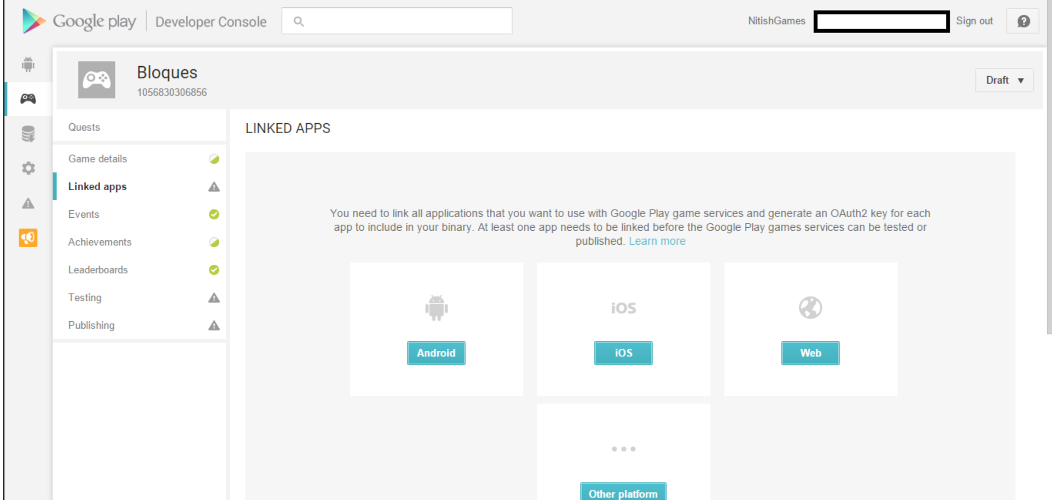
Here you can set the general details regarding your app/game.



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

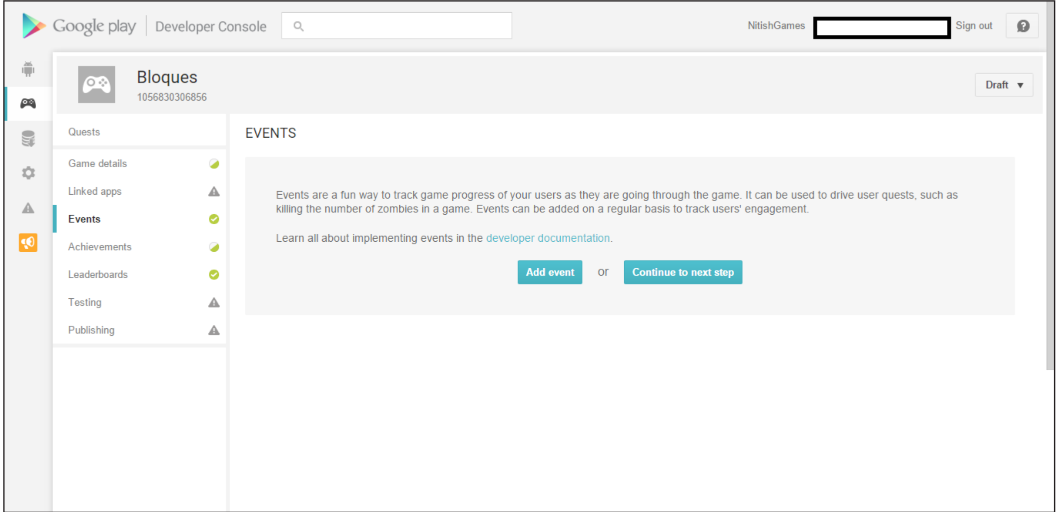
To be able to use Google Services, you will first have to link your app.



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

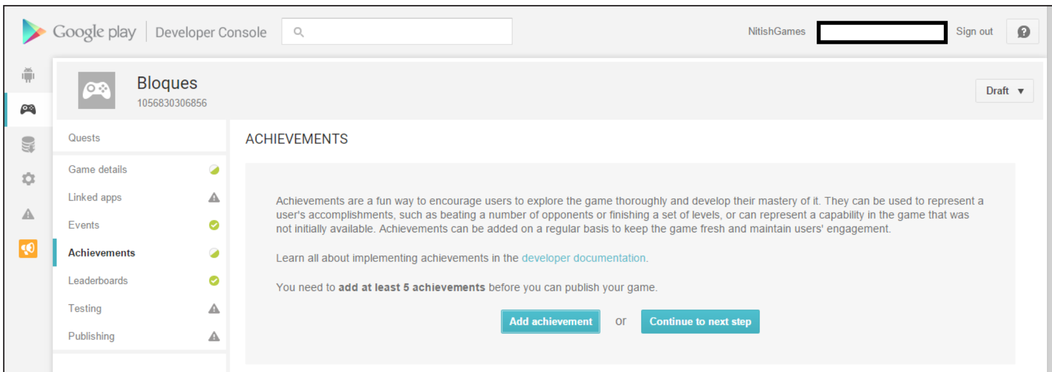
Getting users to download your app is one thing. To ensure that the users who have downloaded your app or game keep using them, in other words user retention, is a whole different area of expertise and equally important, if not more. The next three panels are for just that. The first is the **Events** panel.



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

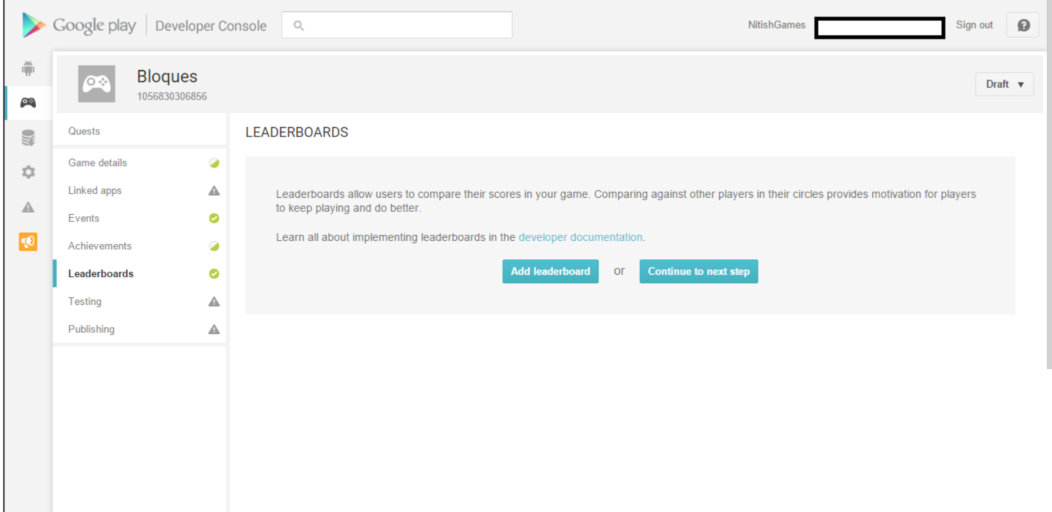
Another way of retaining users is creating achievements. Achievements are goals or tasks that the player has to perform in the game in order to unlock the respective achievement. This could include things like playing the game *X* number of times, killing *Y* number of enemies, and so on. Usually, when 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.



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

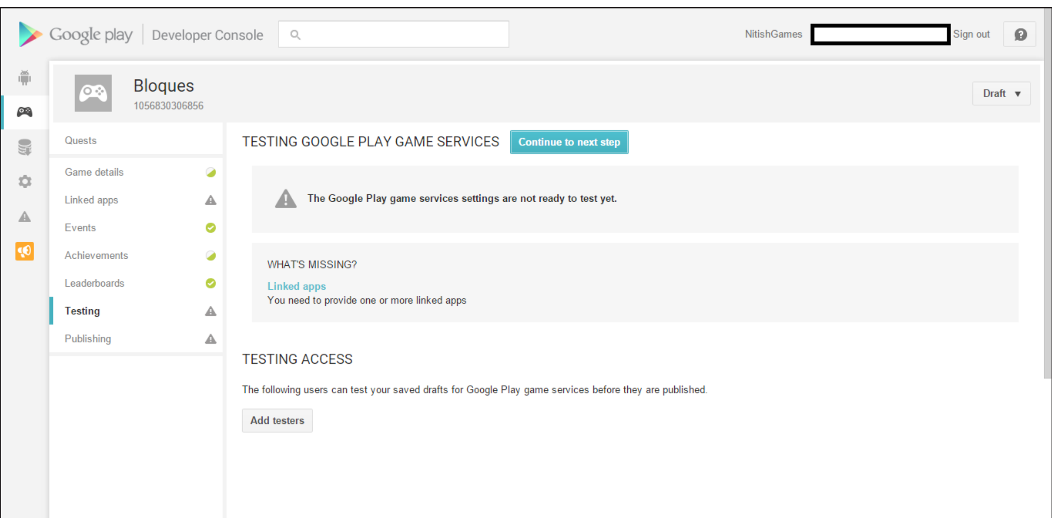
Last, but not least, we have **Leaderboards**. Leaderboards are yet another way of retaining players, especially competitive players who play to get the highest score. A leaderboard is sort of a list where the names of players who have scored the highest points in your game are displayed in descending order.



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

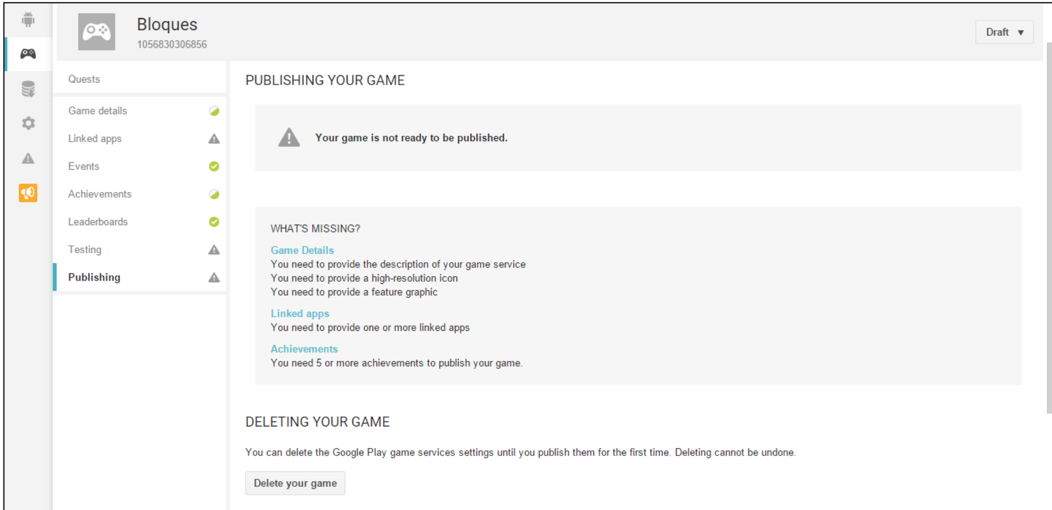
This section contains various options regarding testing the Google Play services on your app/game. Before you publish your game, it is advisable that you first test your app to ensure that everything is working properly and if a problem does arise, fix it before releasing.



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

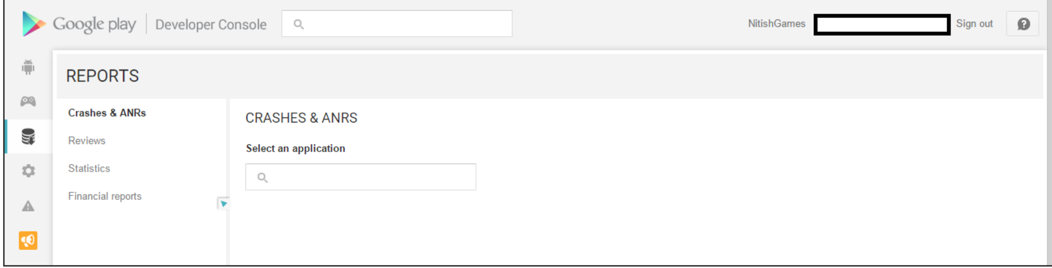
Finally, there is the **Publishing** section. Once you have filled out all of the details, added screenshots, set up the Google Play services, tested them and are confident that your product is ready to be published, you can publish it.



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.



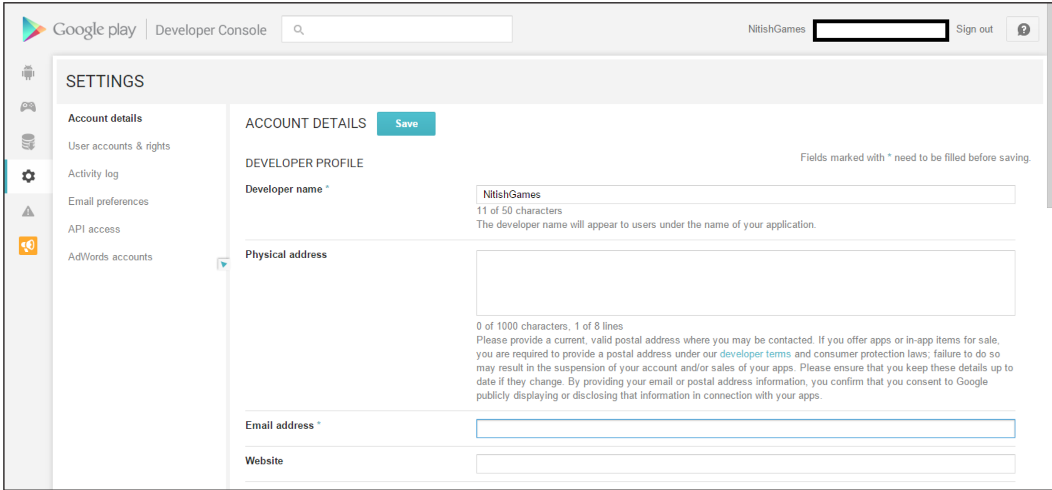
Here too are various sections, all containing different reports:

- **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.



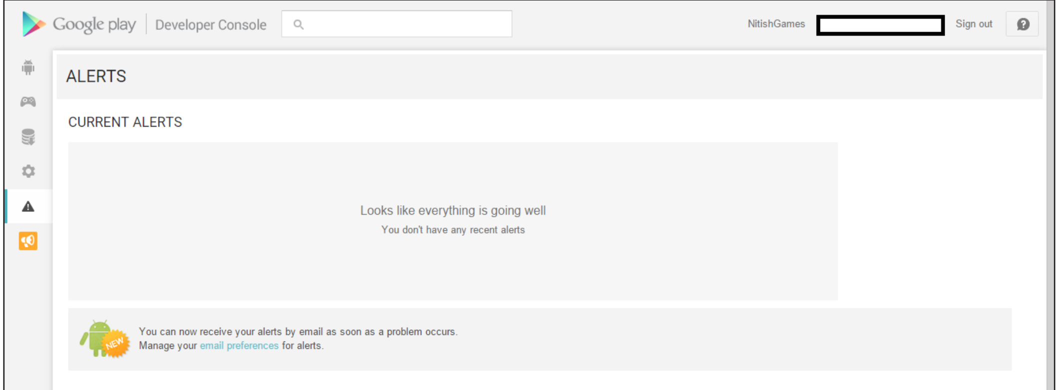
Here too are various sections, each categorized based on the type of options that are as follows:

- **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.

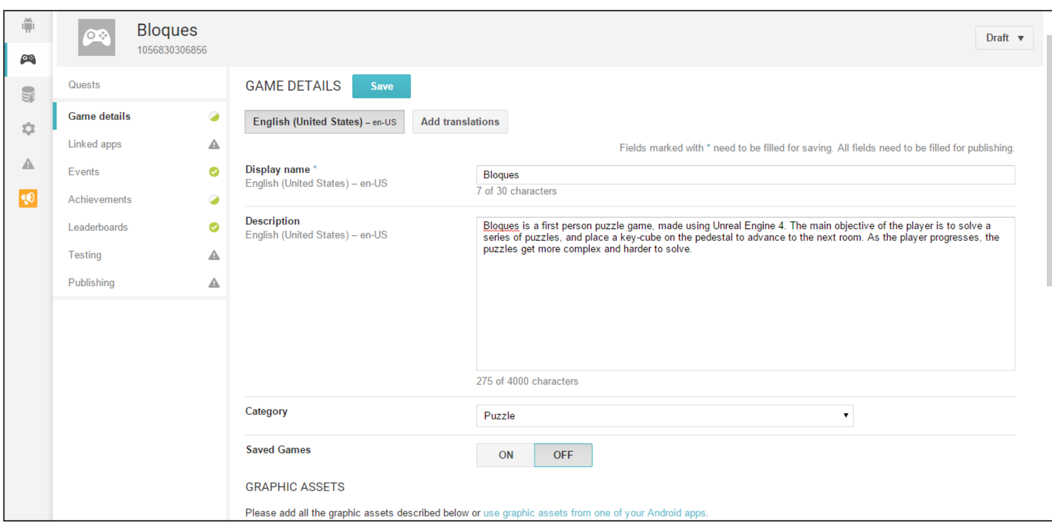


Publishing your game

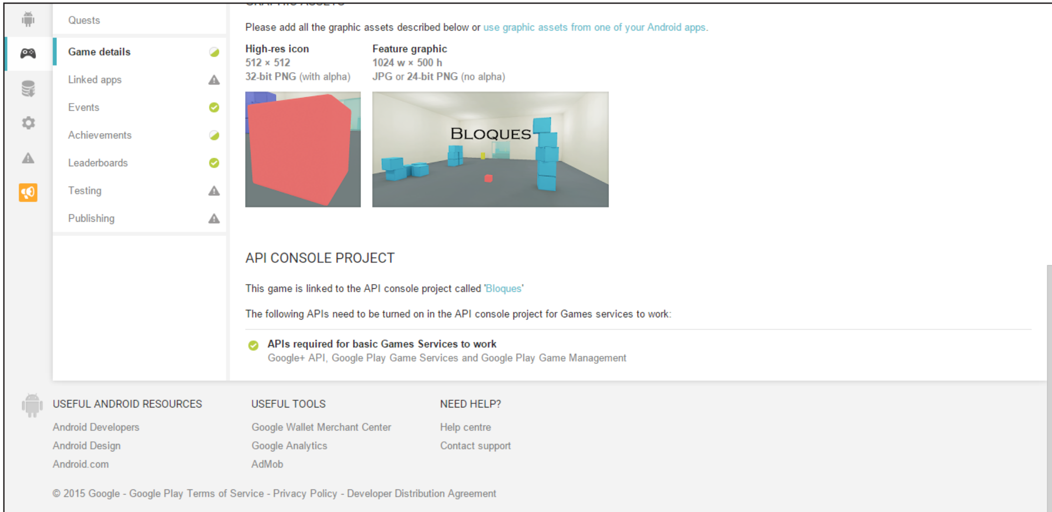
Now that we have discussed the Developer Console, we can go ahead and publish our game to the Google Play Store.

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.



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 x 512` and the resolution required for the feature graphic is `1024 x 500`. You can take high-resolution screenshots of your scene and edit them in Photoshop to create your image.



Once you have everything set up, click on **Save** at the top of the screen.

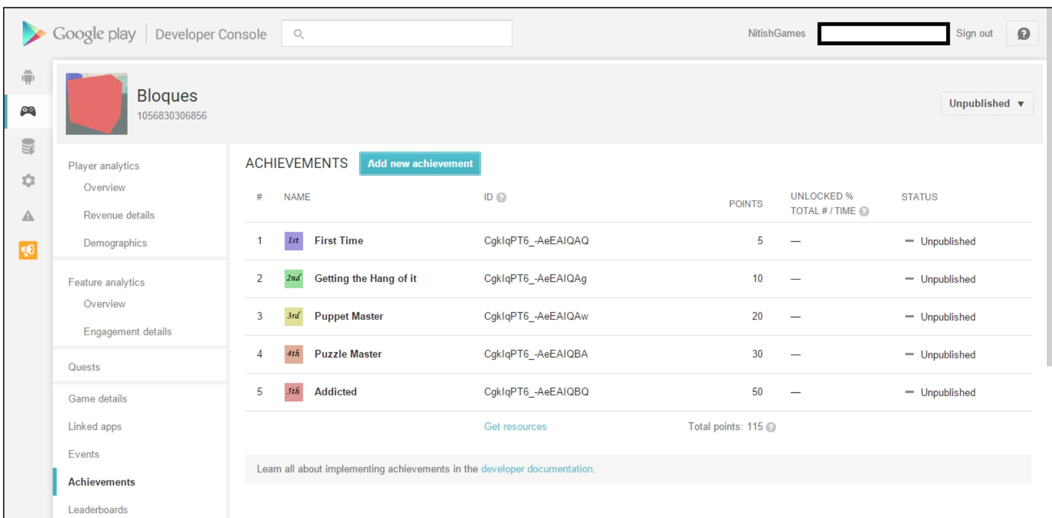
After you have filled in the details go to **Linked apps**, fill in the required fields and click on **Continue** at the top. Once clicked, you will have to authorize your app. Click on **Authorize your app now** button. This will open a menu where you will be asked to fill in the name of the product, 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:

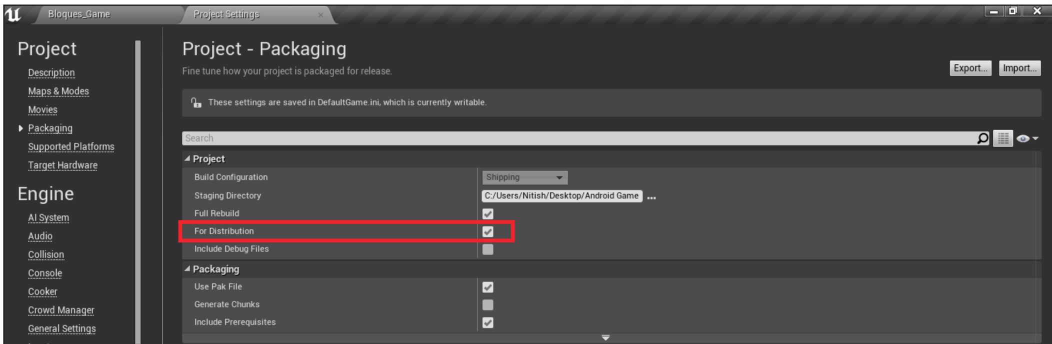
- **First Time**: This is when the player plays the game for the first time
- **Getting the Hang of it**: This is when the player clears the second room
- **Puppet Master**: This is when the player clears the third room
- **Puzzle Master**: This is when the player clears the fourth room
- **Addicted**: This is when the player plays the game five times



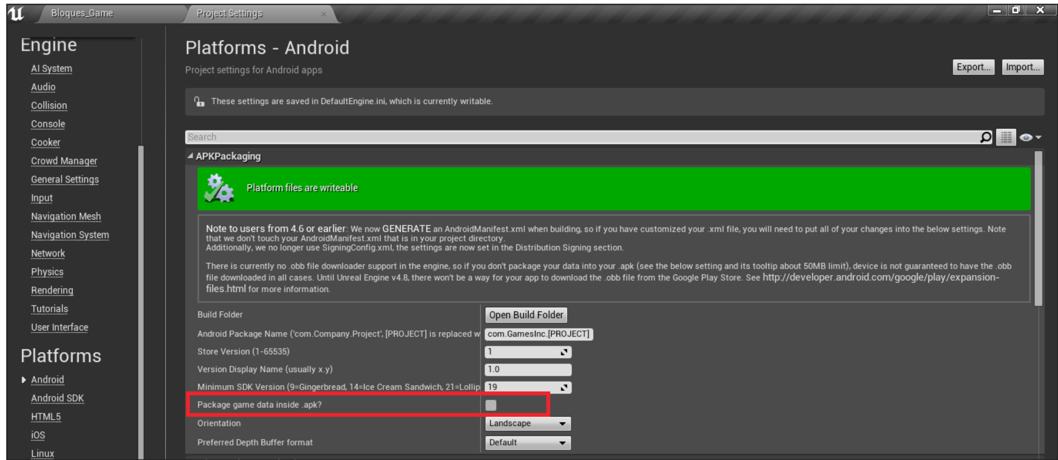
You can assign any number of points you see fit. We do not require leaderboards, since our game does not have points or scores.

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).



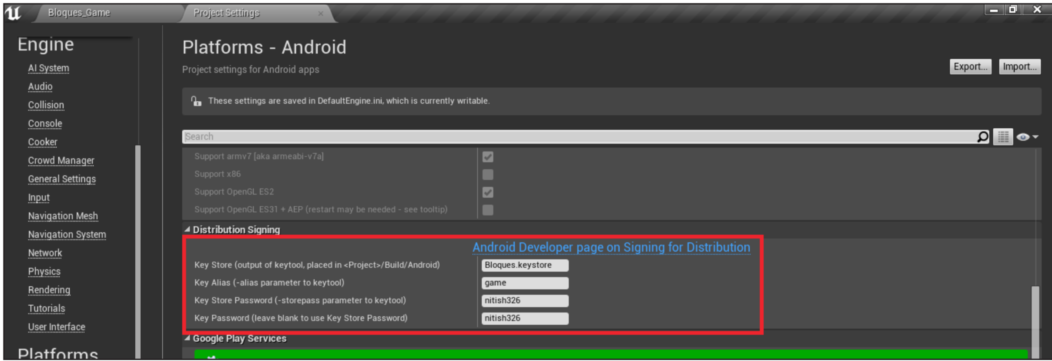
Next, go to **Distribution Signing**. This section is necessary if we are to package our game for shipping. The first thing we need is to create a `keystore` file. A keystore file is a binary file that has a set of keys. You can think of it as a digital signature which is used to identify and authenticate 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
```

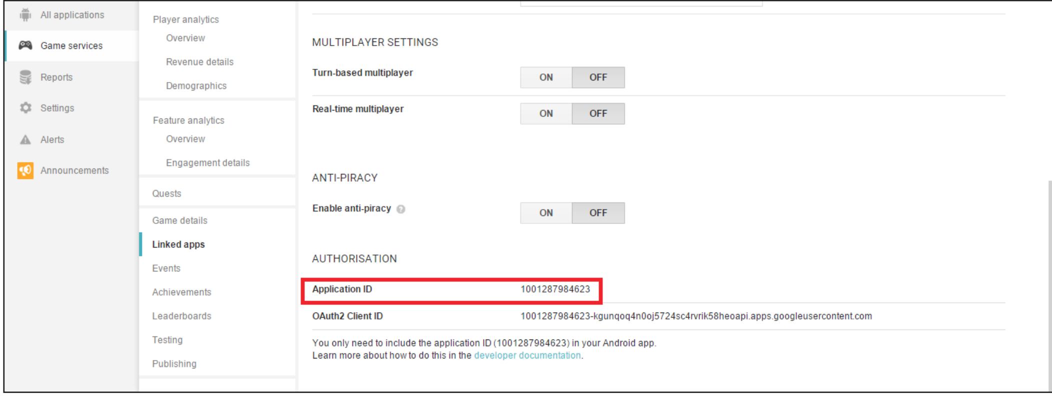
Replace `*name of your project*` with Bloques (or whatever you see fit) and `*alias_name*` with Game (or whatever you see fit).

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`.

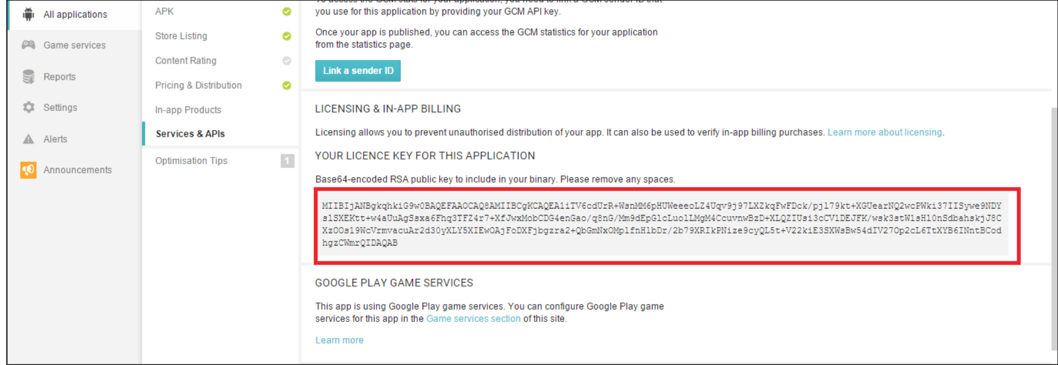
Once done, go back to **Project Preference**, and under **Distribution Signing**, in **Key Store**, enter the name of the `keystore` file you created, along with the extension (for example, if the name of the keystore is Bloques, then enter Bloques.keystore). Then, in **Key Alias**, enter the alias name (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.



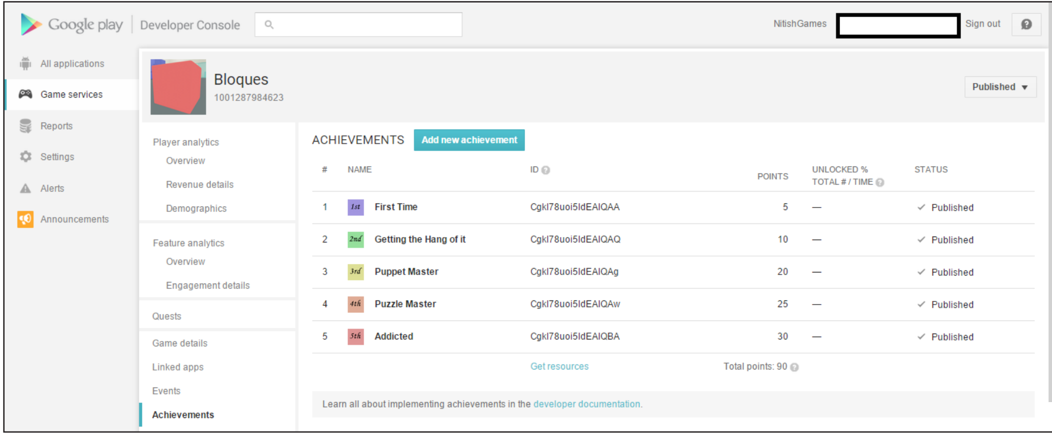
Next, go to **Google Play Services**. Here the first thing to do is to check **Enable Google Play Support**. This will enable Google Play services in the game. In the **Game App ID** field, fill in the Application ID you got when you linked your app in the Developer Console. You can find this ID in the Developer Console under the **Authorization** section of **Linked apps** in the **Game services** panel.



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.

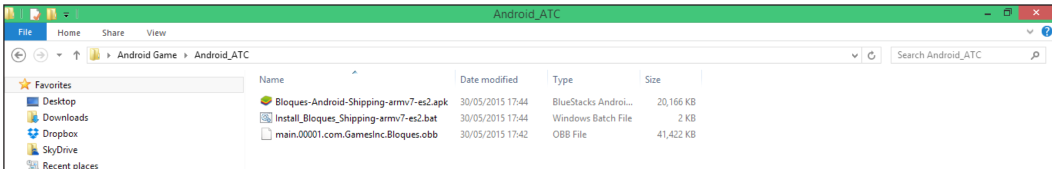


This extremely long string of seemingly random characters is our License Key. Copy the entire thing and paste it in the **Google Play License Key** section. Next, in the Achievement Map, fill in the name and the ID of the achievements you created earlier.



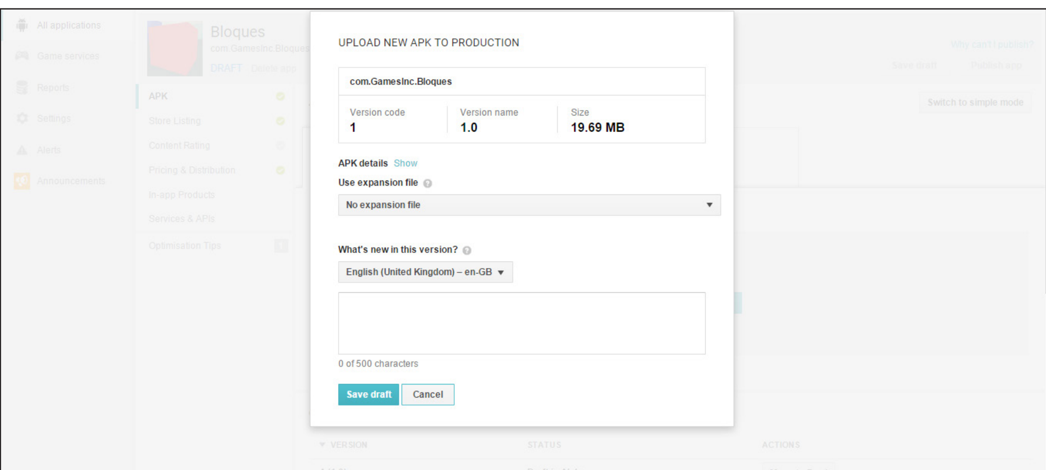
Just copy and paste the name along with the ID here. If you have icons that you would like to add you can do so here as well, in the **Icons** section. We are now ready to package our game for distribution. So, in the Editor, go to **File** | **Package Project** | **Android** and choose any format and 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.



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 paywall 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 this can cause them to stop using or uninstall the app. So be careful how and when you use them.

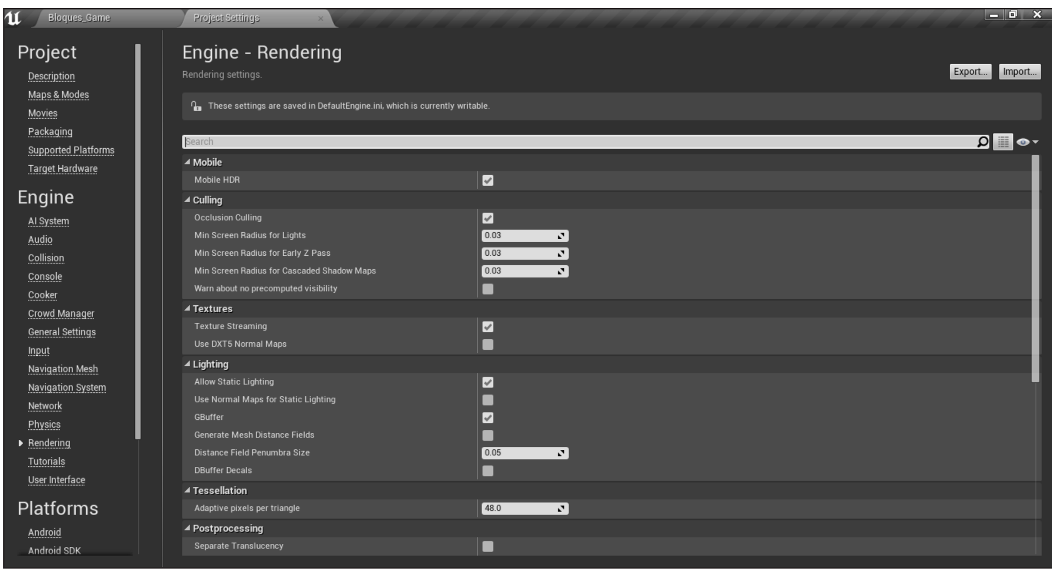
- **Paid apps**: As the name suggests, in this model, the app is not free; users will have to purchase the app in order to use it. The advantages of using this model are that you get upfront revenue from every download along with better user retention (since they paid to download the app, they are more 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.

It is recommended that you visit `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..

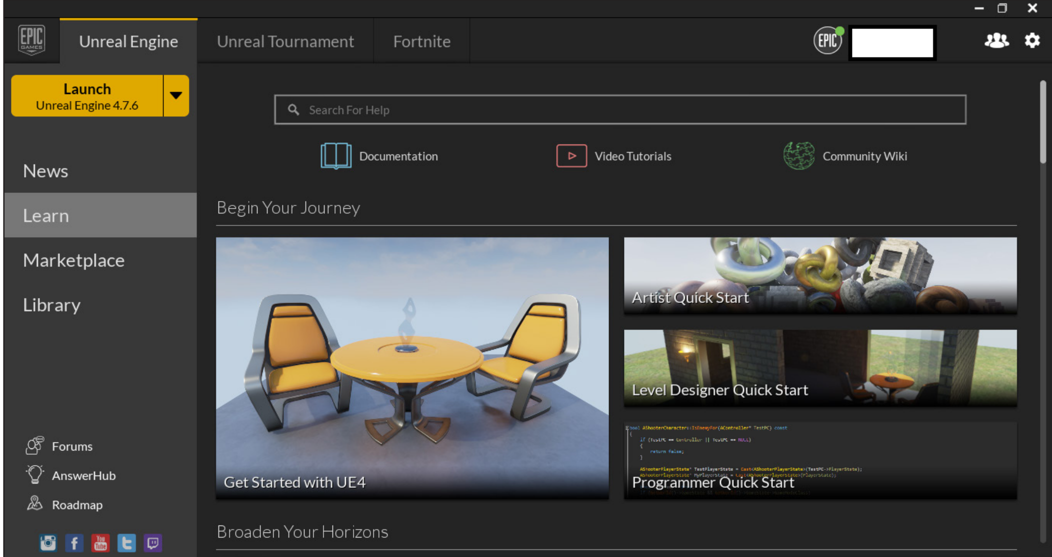
You are now equipped with all that you need to know to get started on making that game you always dreamed of. From here, the only thing you can do to get better at UE4 is practice.

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.

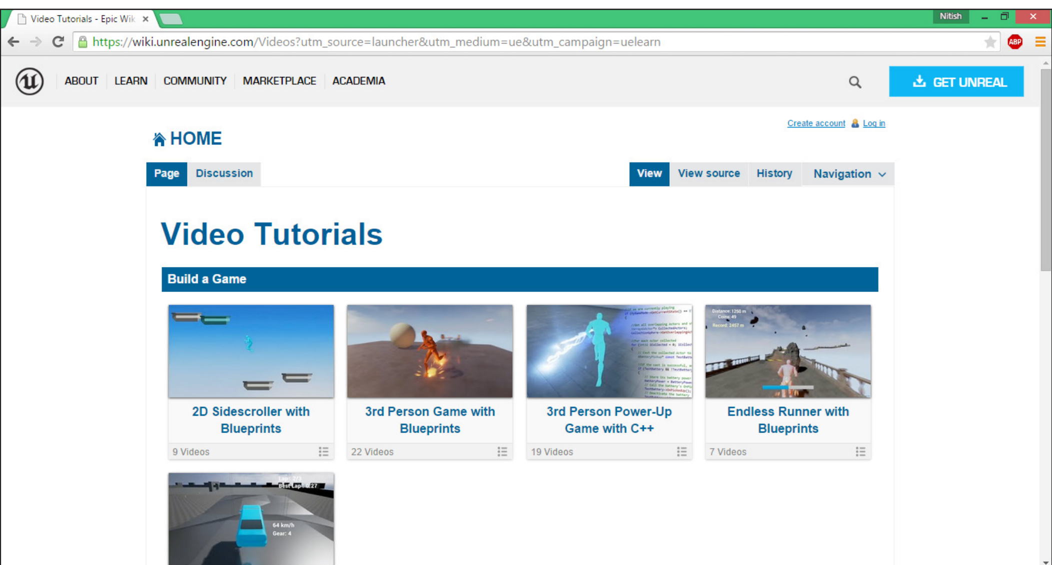
Learn

We briefly covered the **Learn** section in the UE4 Launcher in the first chapter. Let's talk a bit more about it.



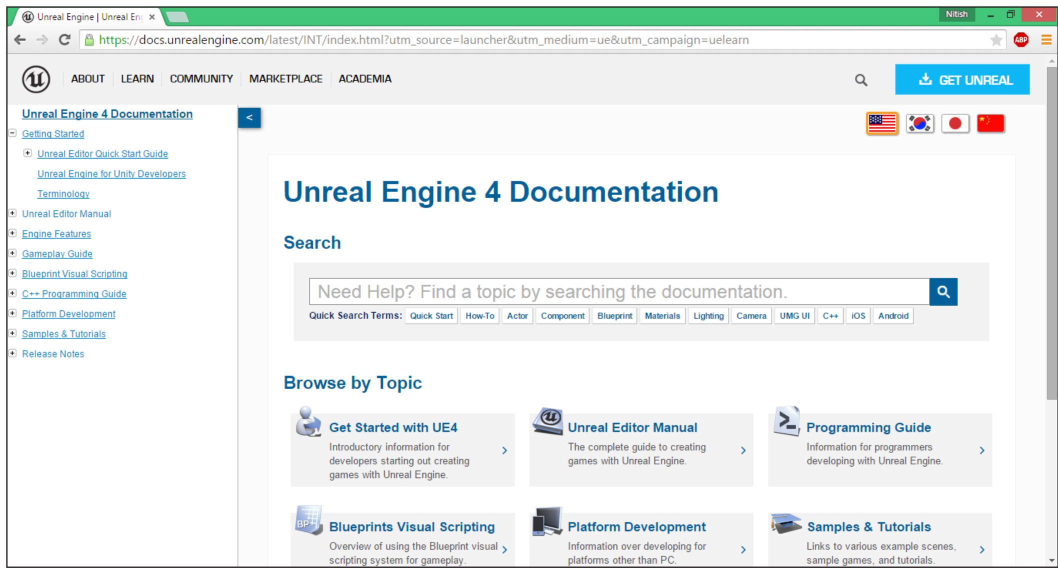
This panel contains tutorials covering a wide variety of topics, including how to create materials, blueprints, and so on. The tutorials available here are available in various formats as follows.

- **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.



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.

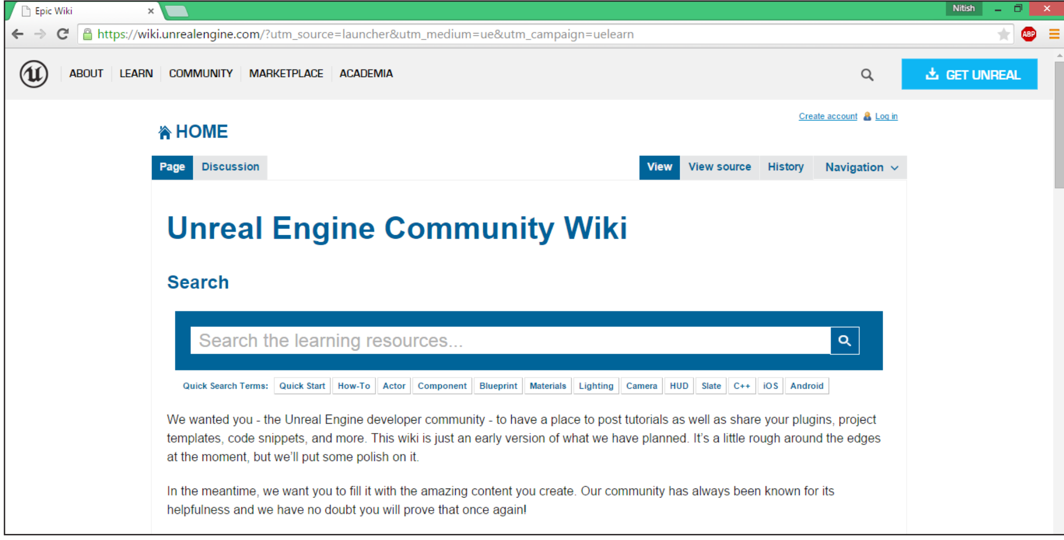
- **Documentation**: Then there is the Epic's official documentation. To access it, click on the **Documentation** hyperlink, and it will take you to the Epic's official documentation page.



The 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.

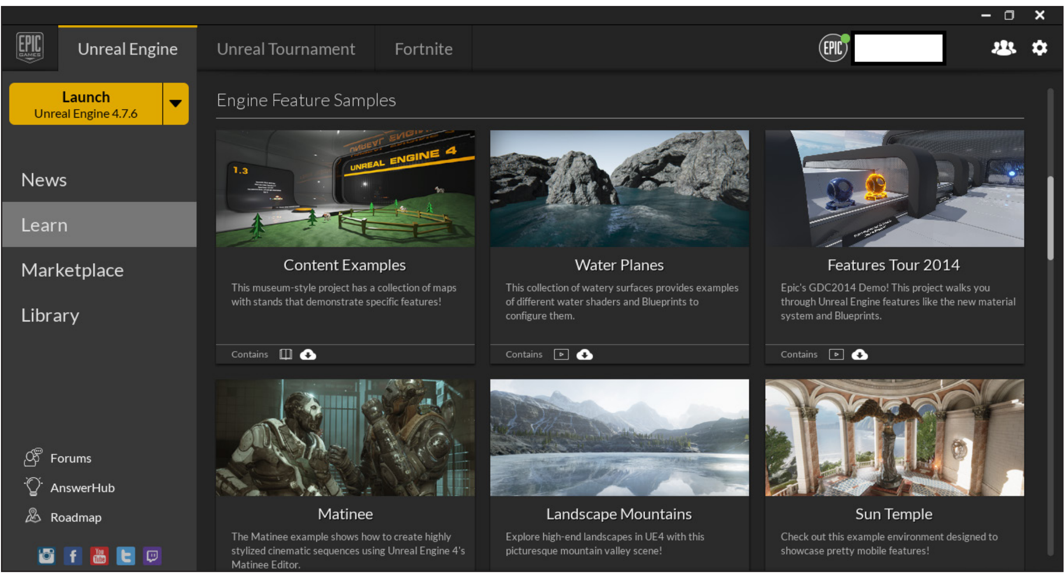
If you are used to making games on Unity and want to move to Unreal 4, Epic has provided a documentation, which compares both engines' Viewport, terminology, and so on, so that you can translate your skills from Unity to Unreal 4. You can find it under the **Broaden Your Horizons** section

- **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.



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.

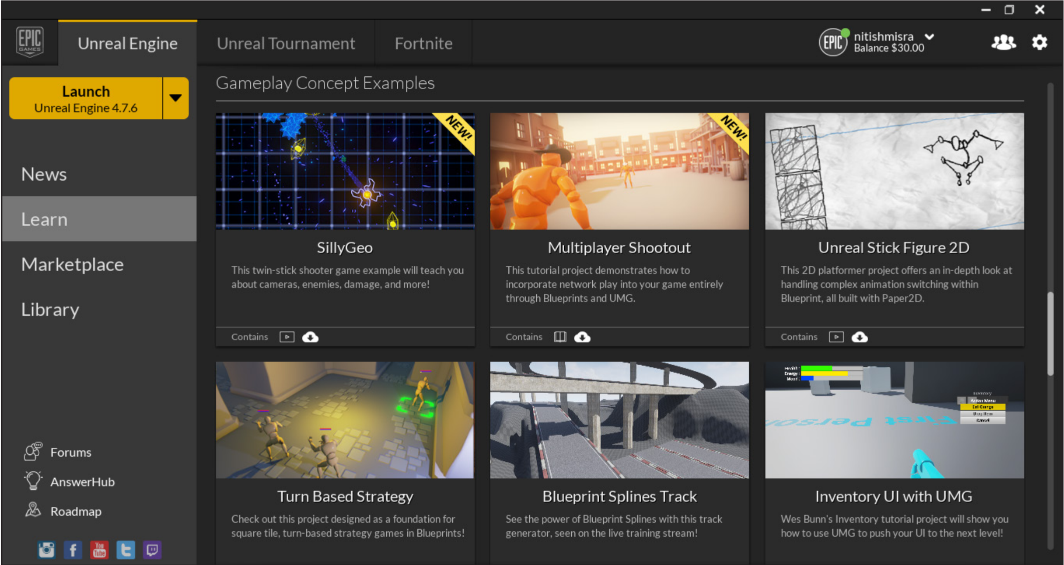
- **Engine Feature Samples**: This section contains project files, each containing a feature or various features offered by UE4.



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.

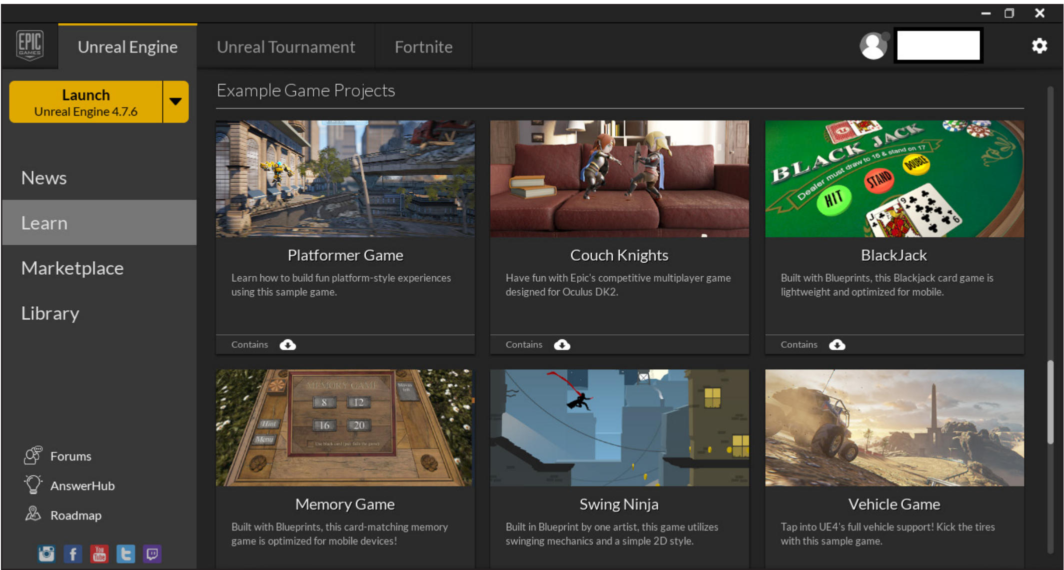
> It is highly advised that you download the **Content Examples** project. This project file contains a collection of the features offered by UE4, all in separate levels, in a museum style manner. For instance, there is a level, which showcases the Material Editor and what can be done with it, there 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.



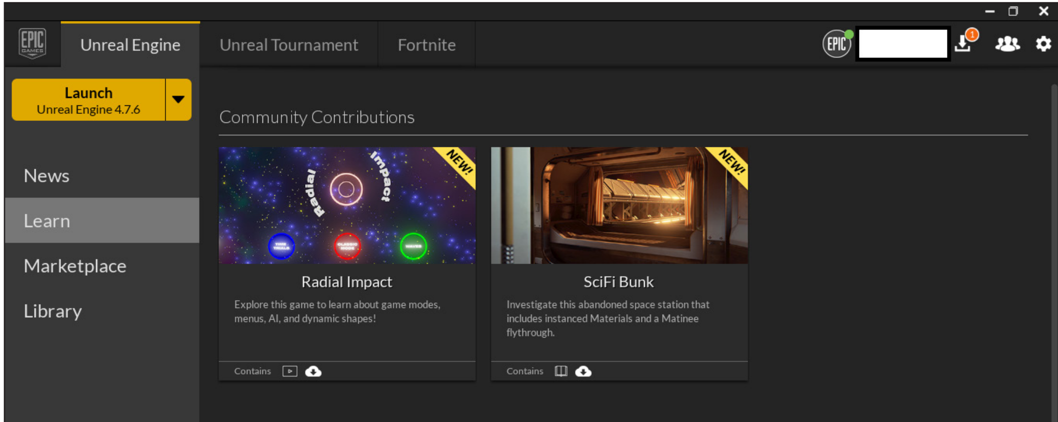
These projects mostly contain a blueprint, each with a mechanic or features usually found in games. They provide the framework, which you can use to build your game upon. For instance, you can find a project file, which has turn-based mechanics setup, there is a project that demonstrates the inventory 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.



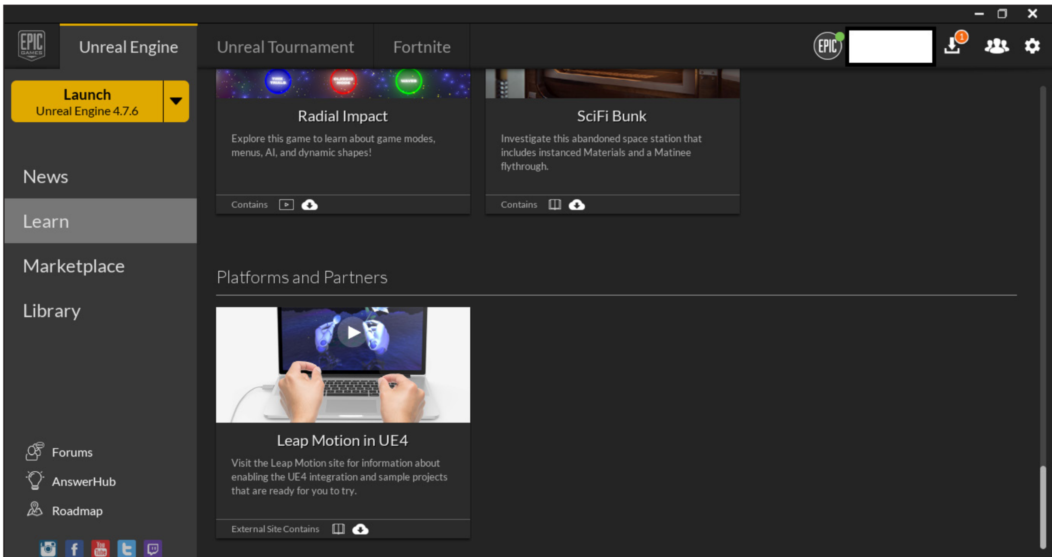
Sample game projects are available with assets and levels set up for you to explore, learn, and utilize. There are a variety of genres you can choose from, for example, 2D platformer, 3D platformer, FPS, and so on.

- **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.



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 UE4s rendering techniques.

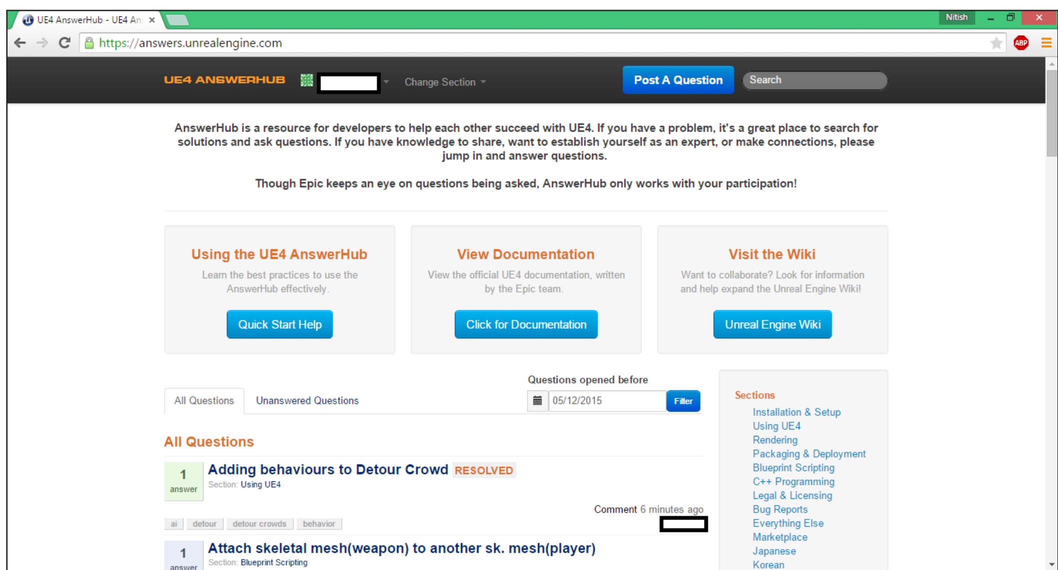
- **Platforms and Partners**: Finally, there's the **Platforms and Partners** section, which has sample projects with technology and peripherals that UE4 currently supports.



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.

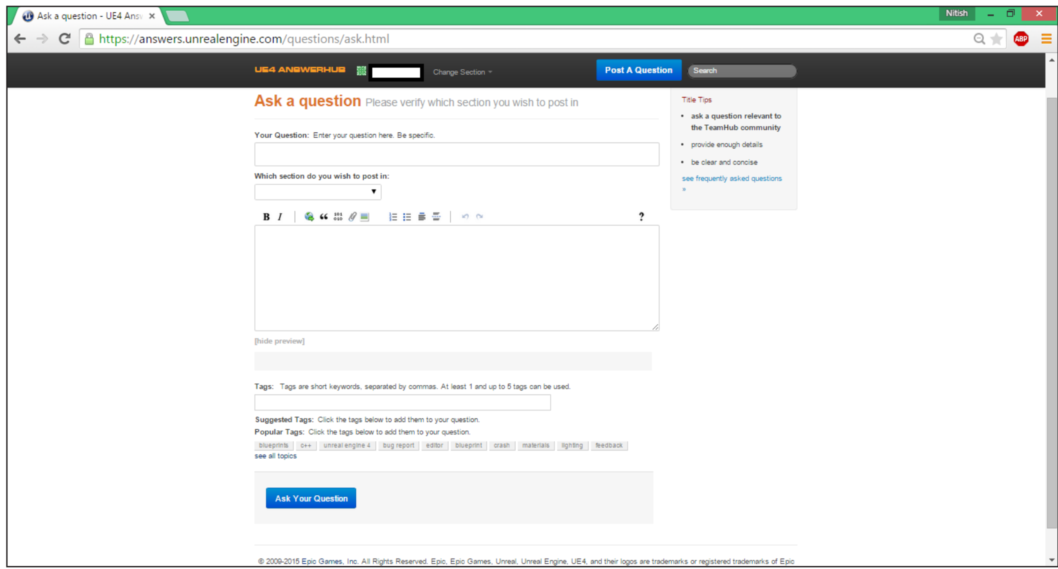
AnswerHub

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.

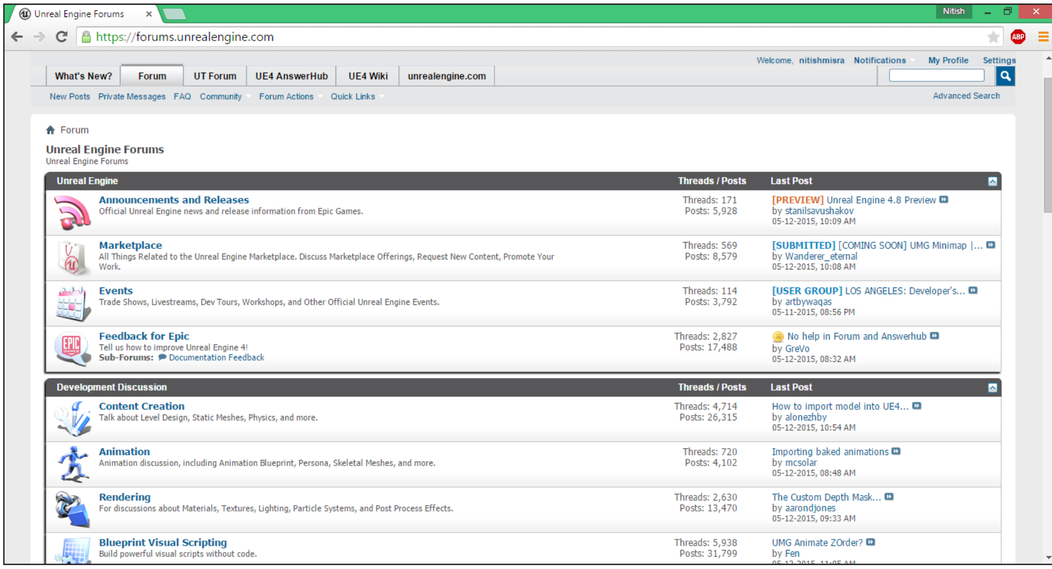
Below this is where you write the details of your issue. This is where you elaborate the question you posted. Again, try to remain brief, concise, and clear in your details. If possible, post screenshots, log files, and so on, so that it is easier for people to understand your problem.

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.



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.

Index

Symbols

4.X folders, Windows directory structure
about 9
Engine 9
Samples 9
Templates 9

A

actors
adding, from All Classes section 124
placing 74-76
ALERTS section, Developer Console 244
ALL APPLICATIONS, Developer Console
about 230
APK panel 231
Content Rating 233
In-app Products 234, 235
Pricing and Distribution 234
Services & APIs 235, 236
Store Listing panel 232, 233
All Classes section, actors
about 122-124
Ambient Sound 122
Camera 122
Class Blueprints 123
Default Pawn 123
Landscape 123
Level Bounds 123
Matinee 123
Nav Link Proxy 123
Target Point 123
Text Renderer 123
Android device
setting up 219-221
Android SDK
installing 216-218
Animations panel, UMG Editor 208
AnswerHub section, UE4 Launcher
about 265, 266
URL 4
assets
importing 70-72
migrating 70-74
assets, UE4
Audio Files 70
Cubemap Texture 70
IES Light Profiles 70
Script Files 70
Static Meshes/Skeletal Meshes 70
Texture Files 70
True Type Fonts 70

B

Basic AI
scripting 165-172
basic class actors
adding, to game 105
basic classes
about 104
Box trigger / Sphere trigger 105
Cone 105
Cube 105
Cylinder 105
Empty Actor 104
Empty Character 104
Empty Pawn 104
Player Start 104
Point Light 105
Sphere 105

Basic Lighting 255
Bloques
about 2, 48
concept 48
controls 48
project, creating for game 48, 49
Blueprint
working 130-133
Blueprint Class
about 133, 155
creating 155, 156
Blueprint Class user interface
Components Panel 157
Construction Script 157-160
Event Graph 157-164
Graph Editor 157
Viewport 158
box brush 51
BSP brushes
about 50
default BSP brushes shapes 51
editing 53
rooms, blocking out with 54-69

C

Camera Actor
adding, from All Classes section 124
Compiler Results panel, Level Blueprint user interface 137
cone brush 52
Content Browser 69, 70
Content Browser, User Interface 39
controls 41, 42
Curved Stair Brush 52
Curve Editor, Unreal Matinee user interface
about 177
options 178, 179
cylinder brush 51

D

decorative assets
materials, creating for 93-95
default BSP brushes shapes
about 51
box brush 51
cone brush 52
curved stair brush 52
cylinder brush 51
linear stair brush 52
sphere brush 51
spiral stair brush 52
Details panel, Level Blueprint user interface 136
Details panel, UMG Editor 206
Details panel, Unreal Matinee user interface 182
Details panel, User Interface 40
Developer Console
about 229, 230
ALERTS 244
ALL APPLICATIONS 230-236
GAME SERVICES 236-242
REPORTS panel 242
SETTINGS section 243
device compatibility, Android
URL 256
directional light 95, 96
Director Group 188
DirectXRedist 8
door, animating
about 183
bridge for AI character 194-199
room 1 183-191
room 2 191-193
Dots per inch (DPI) 205

E

Editor 21
Engine Launcher
about 9, 10
Learn section 12
library 14-17
Marketplace 13
News panel 12
UE4 Links 17
environment
lighting up 101
Event Graph, Level Blueprint user interface 138, 139
Event Nodes 131

F

Facebook, UE4
URL 18
Field of View (FOV) Angle track 187
forums, UE4 Launcher
about 267
community 267
development discussion 267
International 268
UE4 for Schools 268
Unreal Engine 267
URL 4
Freemium model 252
Full HDR Lighting 255
Function Nodes 131

G

game
basic class actors, adding to 105
high-level breakdown feature 2
Level Blueprint, using in 139
uploading, on Play Store 251, 252
Visual Effect actors, adding to 114-116
volumes, adding to 118
game, Level Blueprint
key cube, picking up 139-154
key cube, placing 139-154
game, publishing
about 244
Google services, activating 244-247
project, preparing for shipping 247-250
GAME SERVICES, Developer Console 236-242
Graph Editor, UMG Editor
about 204
Grid Snapping 205
Grid Snap Value 205
Preview Size 205
Widget Layout Transformation 205
Widget Render Transformation 205
Zoom to Fit 205

H

heads-up display (HUD) 201
Hierarchy panel, UMG Editor 207
hotkeys 41, 42

I

in-app advertisement 253
Instagram, UE4
URL 18
installing
UE4 3-6

K

Key Cubes material
creating 91-93
KillZ Volume 36
Kismet 129

L

Launcher folder, Windows directory structure
about 8
Backup 8
Engine 8
PatchStaging 8
VaultCache 8
Learn section, UE4 Launcher
about 12, 257
Community Contributions 263, 264
Documentation 259
Engine Feature Samples 260, 261
Example Game Project 262, 263
Gameplay Content Examples 262
Platforms and Partners 264
Unreal Wiki 260
Video Tutorials 258
Level Blueprint
about 133
using, in game 139
Level Blueprint user interface
about 134
Compiler Results panel 137
Details panel 136

Event Graph 138, 139
menu bar 134
My Blueprint panel 137
tab bar 134
toolbar 135
library, Engine Launcher 14-17
lighting 95
Lightmass Importance Volume
using 119, 120
lights
mobility 99, 100
Linear Stair Brush 52
Low Dynamic Range (LDR) 255

M

main menu
adding, UMG Editor used 201
creating 208-216
Marketplace, Engine Launcher 13
Material Editor
about 78
Details panel 82, 83
Graph panel 84, 85
menu bar 79
Palette panel 81
Stats panel 82
tab bar 79
toolbar 80, 81
Viewport panel 83
materials
about 77
applying 85-87
creating 87
creating, for decorative assets 93-95
creating, for doors 90, 91
creating, for pedestals 87, 88
Matinee actor
adding, from All Classes section 125
menu bar, Level Blueprint user interface
Debug 135
Edit 134
File 134
Help 135
View 135
Window 135
menu bar, UMG Editor 202, 203
menu bar, Unreal Matinee user interface
actions and functions 175, 176
menu bar, User Interface
Edit 28
Editor Window 28
File 28
Help 29
mobile optimization 254
mobile performance 254
mobility, lights
movable 100
static 100
stationary 100
Modes window, User Interface
about 35
Foliage Mode 36
Geometry Editing Mode 37
Landscape Mode 36
Paint Mode 36
PlaceMode 35
monetization methods 252
monetization models
Freeminum model 252
in-app advertisement 253
In-App purchases 253
paid apps 253
Movement track 187
My Blueprint panel, Level Blueprint user interface 137

N

Nav Mesh Bounds Volume
about 120
placing 120, 121
News panel, Engine Launcher 12
nodes
Event Nodes 131
Function Nodes 131
Reference Nodes 132
Variable Nodes 132

P

package
building, for project 226-228
paid apps 253

Palette panel, UMG Editor 206
pedestals
materials, creating for 87, 88
PlaceMode, classes
about 35
Basic 35
BSP 36
Lights 36
Visual 36
Volumes 36
Player Start actor
placing 105, 106
Play Store
game, uploading on 251, 252
point light 97
Prefab 41
project
about 45
creating 46
directory structure 47
existing project, opening 47
project, packaging
about 221
Android app settings 224, 225
Maps & Modes settings 222
Packaging settings 223

R

Reference Nodes 132
REPORTS panel, Developer Console 242
rooms
blocking out, with BSP brushes 54-69

S

Scalable 25
Sci-Fi Hallway project 45
SETTINGS section, Developer Console 243
Sky Distance Threshold 99
sky light 98, 99
social icons, Epic
Facebook 18
Instagram 18
Twitch Stream 18
Twitter 18
YouTube 18
sphere brush 51
Spiral Stair Brush 52
spot light 98

T

tab bar, UMG Editor 202
Target Point actors
adding, from All Classes section 125-127
toolbar, Level Blueprint user interface
about 135
Class Defaults 136
Class Settings 135
Compile 135
Play 136
Search 135
toolbar, UMG Editor 203, 204
toolbar, Unreal Matinee user interface
about 176
actions 176, 177
toolbar, User Interface
Blueprints 30
Build 30
Content button 30
Eject button 31
Launch button 31
Marketplace 30
Matinee 30
Pause button 31
Play button 30
Save 29
Settings 30
Source Control 29
Stop button 31
Tracks panel, Unreal Matinee user interface 180-182
triggers
adding, to rooms 106-111
Twitch Stream 18

U

UE4
downloading 3-6
installing 3-6
system requisites 2
URL, for homepage 3

UE4 Launcher
AnswerHub section 265, 266
forums 267, 268
Learn section 257-264
UE4 Links, Engine Launcher
about 17
AnswerHub 18
forums 17
Roadmap 18
UMG Editor
about 202
Animations panel 208
Details panel 206
Graph Editor 204, 205
Hierarchy panel 207
menu bar 202
Palette panel 206
tab bar 202
toolbar 203, 204
used, for adding main menu 201
Unreal
URL, for official site 3
Unreal Development Kit (UDK) 174
Unreal Matinee
about 173
door, animating 183
Matinee actors, adding 174
Unreal Matinee user interface
about 174
Curve Editor 177-179
Details panel 182
menu bar 175, 176
tab bar 175
toolbar 176, 177
Tracks panel 180-182
Unreal Motion Graphics. *See* **UMG Editor**
Unreal Project Browser 22-25
User Interface
about 26
Content Browser 39
Details panel 40, 41
menu bar 27-29
Modes window 35-37
tab bar 27
toolbar 29-31
Viewport 31-34
World Outliner 37, 38

V

Variable Nodes 132
Viewport, User Interface
about 31
Camera Speed 34
Coordinate System 33
Grid Snapping 33
Grid Snap Value 33
Maximize or Restore Viewport 34
Rotation Grid Snapping 34
Rotation Grid Snap Value 34
Scale Grid Snapping 34
Scale Grid Snap Value 34
Show 32
Surface Snapping 33
Transform Tools 32
View Mode 32
Viewport Options 32
Viewport Type 32
Visual Effect actors
adding, to game 114-116
Visual Effects class
about 111
actors 112
Atmospheric Fog actor 112
Box Reflection Capture actor 113
Deferred Decal actor 113
Exponential Height Fog actor 113
Post Process Volume actor 112
Sphere Reflection actor 113
Volumes
about 116
adding, to game 118
Audio Volume 117
Blocking Volume 117
Camera Blocking Volume 117
Cull Distance Volume 117
Kill ZVolume 117
Lightmass Importance Volume 118
Lightness Character Indirect Detail Volume 118
Nav Mesh Bounds Volume 118
Nav Modifier Volume 118
Pain Causing Volume 118
Physics Volume 118
Post Process Volume 118

Precomputed Visibility Override Volume 118
Precomputed Visibility Volume 118
Trigger Volume 118
types 116-118

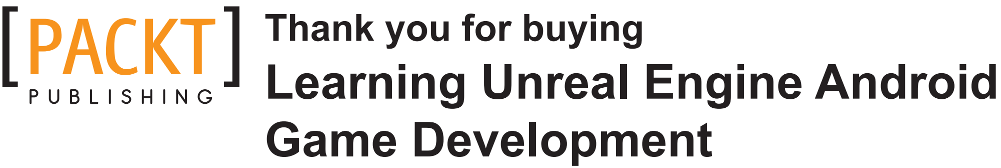
W

Windows directory structure
4.X folders 8
about 7
DirectXRedist 8
Launcher folder 8

World Outliner, User Interface
about 37
Attach Actors 38
Create Folders 38
Hide Function 38

Y

YouTube, UE4
URL 18



Thank you for buying

Learning Unreal Engine Android Game Development

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

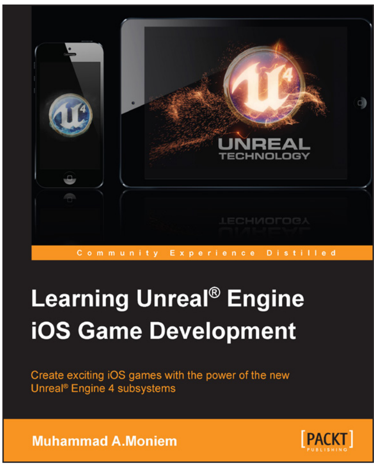
Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

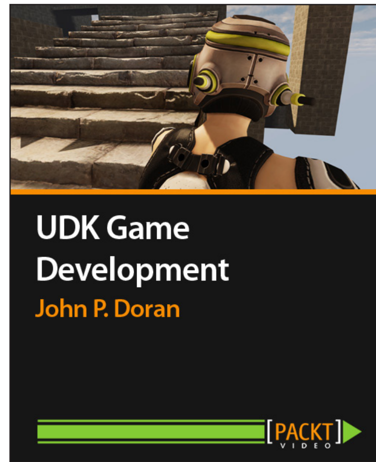


Learning Unreal® Engine iOS Game Development

ISBN: 978-1-78439-771-5 Paperback: 212 pages

Create exciting iOS games with the power of the new Unreal® Engine 4 subsystems

1. Learn about the entire iOS pipeline, from game creation to game submission.
2. Develop exciting iOS games with the Unreal Engine 4.x toolset.
3. Step-by-step tutorials to build optimized iOS games.



UDK Game Development [Video]

ISBN: 978-1-84969-618-0 Duration: 03:37 hours

Create a complete, feature-filled 3D platformer in UDK using Kismet and UnrealScript

1. Develop and customize your very own game quickly and easily in UDK.
2. Enhance the game environment by using Kismet and UnrealScript to create a compelling gameplay experience.
3. An engaging course with step-by-step instructions, making it ideal for newcomers to UDK.

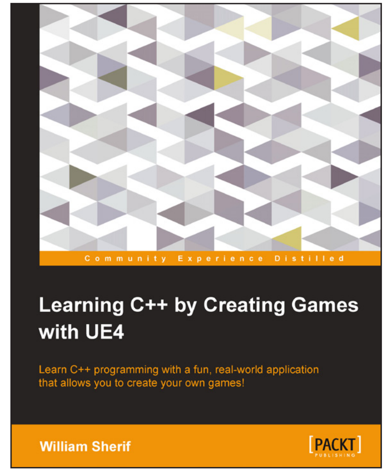
Please check **www.PacktPub.com** for information on our titles

Unreal Development Kit Game Programming with UnrealScript [Video]

ISBN: 978-1-84969-632-6 Duration: 02:23 hours

Kick-start your career in game development with UnrealScript and the UDK

1. Step-by-step guide on how to set up the UDK and create a game with UnrealScript.
2. Explore core UnrealScript features and configurations.
3. Ideal for newcomers to UnrealScript.



Learning C++ by Creating Games with UE4

ISBN: 978-1-78439-657-2 Paperback: 342 pages

Learn C++ programming with a fun, real-world application that allows you to create your own games!

1. Be a top programmer by being able to visualize programming concepts; how data is saved in computer memory, and how a program flows.
2. Keep track of player inventory, create monsters, and keep those monsters at bay with basic spell casting by using your C++ programming skills within Unreal Engine 4.
3. Understand the C++ programming concepts to create your own games.

Please check **www.PacktPub.com** for information on our titles