

EXPERT INSIGHT

Early
Access

Hands-On Unity Game Development

Unlock the power of Unity 2023 and build
your dream game

Fourth Edition



Nicolas Alejandro Borromeo
Juan Gabriel Gomila Salas

<packt>

Hands-On Unity Game Development

Copyright © 2023 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.

Early Access Publication: Hands-On Unity Game Development

Early Access Production Reference: B21361

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-83508-571-4

www.packt.com

Table of Contents

1. [Hands-On Unity Game Development, Fourth Edition: Unlock the power of Unity 2023 and build your dream game](#)
2. [1 Embark on Your Unity Journey](#)
 - I. [Join our book community on Discord](#)
 - II. [Installing Unity](#)
 - i. [Unity's technical requirements](#)
 - ii. [Unity versions](#)
 - iii. [Installing Unity with Unity Hub](#)
 - III. [Creating projects](#)
 - i. [Creating a project](#)
 - ii. [Project structure](#)
 - IV. [Summary](#)
3. [2 Crafting Scenes and Game Elements](#)
 - I. [Join our book community on Discord](#)
 - II. [Manipulating scenes](#)
 - i. [The purpose of a scene](#)
 - ii. [The Scene view](#)
 - iii. [Adding our first GameObject to the scene](#)
 - iv. [Navigating the Scene view](#)
 - v. [Manipulating GameObjects](#)
 - III. [GameObjects and components](#)
 - i. [Understanding components](#)
 - ii. [Manipulating components](#)
 - IV. [Understanding object Hierarchies](#)
 - i. [Parenting of objects](#)
 - ii. [Possible uses](#)
 - V. [Managing GameObjects using Prefabs](#)
 - i. [Creating Prefabs](#)
 - ii. [Prefab-instance relationship](#)
 - iii. [Prefab variants](#)
 - VI. [Saving scenes and projects](#)

VII. Summary

4. 3 From Blueprint to Reality: Building with Terrain and ProBuilder

- I. Join our book community on Discord
- II. Defining our game concept
- III. Creating a landscape with Terrain
 - i. Discussing Height Maps
 - ii. Creating and configuring Height Maps
 - iii. Authoring Height Maps
 - iv. Adding Height Map details
 - v. Creating shapes with ProBuilder
 - vi. Installing ProBuilder
 - vii. Creating a shape
 - viii. Manipulating the mesh
 - ix. Adding details

IV. Summary

5. 4 Seamless Integration: Importing and Integrating Assets

- I. Join our book community on Discord
- II. Importing assets
 - i. Importing assets from the internet
 - ii. Importing assets from the Asset Store
 - iii. Importing assets from Unity packages
- III. Integrating assets
 - i. Integrating terrain textures
 - ii. Integrating meshes
 - iii. Integrating textures
- IV. Configuring assets
 - i. Configuring meshes
 - ii. Configuring textures
 - iii. Assembling the scene

V. Summary

6. 5 Unleashing the Power of C# and Visual Scripting

- I. Join our book community on Discord
- II. Introducing scripting
- III. Creating scripts
 - i. Initial setup

- ii. [Creating a C# script](#)
- iii. [Adding fields](#)
- iv. [Creating a Visual Script](#)

IV. [Using events and instructions](#)

- i. [Events and instructions in C#](#)
- ii. [Events and instructions in Visual Scripting](#)
- iii. [Using fields in instructions](#)

V. [Common beginner C# script errors](#)

VI. [Summary](#)

7. [6 Dynamic Motion: Implementing Movement and Spawning](#)

- I. [Join our book community on Discord](#)
- II. [Implementing movement](#)
 - i. [Moving objects through Transform](#)
 - ii. [Using Input](#)
 - iii. [Understanding Delta Time](#)
- III. [Implementing spawning](#)
 - i. [Spawning objects](#)
 - ii. [Timing actions](#)
 - iii. [Destroying objects](#)

IV. [Using the new Input System](#)

- i. [Installing the new Input System](#)
- ii. [Creating Input Mappings](#)
- iii. [Using Mappings in our scripts](#)

V. [Summary](#)

8. [7 Collisions and Health: Detecting Collisions Accurately](#)

- I. [Join our book community on Discord](#)
- II. [Configuring physics](#)
 - i. [Setting shapes](#)
 - ii. [Physics object types](#)
 - iii. [Filtering collisions](#)
- III. [Detecting collisions](#)
 - i. [Detecting Trigger events](#)
 - ii. [Modifying the other object](#)
- IV. [Moving with physics](#)
 - i. [Applying forces](#)
 - ii. [Tweaking physics](#)

V. Summary

9. 8 Victory or Defeat: Win and Lose Conditions

I. Join our book community on Discord

II. Creating object managers

i. Sharing variables with the Singleton design pattern

ii. Sharing variables with Visual Scripting

iii. Creating managers

iv. Creating Game Modes

III. Improving our code with events

IV. Summary

10. 9 Starting your AI Journey: Building Intelligent Enemies for your Game

I. Join our book community on Discord

II. Gathering information with sensors

i. Creating three-filter sensors with C#

ii. Creating Three-Filters sensors with Visual Scripting

iii. Debugging with gizmos

III. Making decisions with FSMs

i. Creating the FSM in C#

ii. Creating transitions

iii. Creating the FSM in Visual Scripting

IV. Executing FSM actions

i. Calculating our scene's NavMesh

ii. Using Pathfinding

V. Adding the final details

VI. Summary

11. 10 Material Alchemy: URP and Shader Graph for Stunning Visuals

I. Join our book community on Discord

II. Introducing shaders and URP

i. Shader Pipeline

ii. Render Pipeline and URP

iii. URP built-in shaders

III. Creating shaders with Shader Graph

i. Creating our first Shader Graph

IV. Using Textures

V. [Combining Textures](#)

VI. [Applying transparency](#)

VII. [Creating Vertex Effects](#)

VIII. [Summary](#)

Hands-On Unity Game Development, Fourth Edition: Unlock the power of Unity 2023 and build your dream game

Welcome to Packt Early Access. We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time. You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book.

1. Chapter 1: Embark on Your Unity Journey
2. Chapter 2: Crafting Scenes and Game Elements
3. Chapter 3: From Blueprint to Reality: Building with Terrain and ProBuilder
4. Chapter 4: Seamless Integration: Importing and Incorporating Assets
5. Chapter 5: Unleashing the Power of C# and Visual Scripting
6. Chapter 6: Dynamic Motion: Implementing Object Movement and Spawning
7. Chapter 7: Collisions and Health: Detecting Collisions Accurately
8. Chapter 8: Victory or Defeat: Designing Win and Lose Conditions

9. Chapter 9: Starting your AI Journey: Building Intelligent Enemies for Your Game
10. Chapter 10: Material Alchemy: URP and Shader Graph for Stunning Visuals
11. Chapter 11: Captivating Visual Effects: Harnessing Particle Systems and VFX Graph
12. Chapter 12: Enlightening Worlds: Illuminating Scenes with the Universal Render Pipeline
13. Chapter 13: Immersive Realism: Achieving Fullscreen Effects with Postprocessing
14. Chapter 14: Harmonious Soundscapes: Integrating Audio and Music
15. Chapter 15: Interface Brilliance: Designing User-Friendly UI
16. Chapter 16: Next-Gen UI: Creating Dynamic Interfaces with the UI Toolkit
17. Chapter 17: Animated Realities: Crafting Animations with Animator, Cinemachine, and Timelin
18. Chapter 18: Performance Wizardry: Optimizing Your Game with Profiler Tools
19. Chapter 19: From Prototype to Executable: Generating and Debugging Your Game
20. Chapter 20: AR/VR

1 Embark on Your Unity Journey

Join our book community on Discord

<https://packt.link/unitydev>



In this chapter, we will learn how to install Unity and create a project with Unity Hub, a tool that manages different Unity versions and projects, among other tasks. Unity Hub gives easy access to community blogs, forums, resources, and learning portals; it also manages your licenses and allows managing different installs and projects. Specifically, we will examine the following topics in this chapter:

- Installing Unity
- Creating projects

Let's start by talking about how to get Unity up and running. If you already know how to install Unity, feel free to skip ahead to *Chapter 2, Editing Scenes and Game Objects*. If you are already familiar with Unity's editor, you can jump to *Chapter 3, Grayboxing with Terrain and ProBuilder*, where we start creating the book's project.

Installing Unity

We'll begin with a simple but necessary first step: installing Unity. It seems like a straightforward first step, but we can discuss the

proper ways to do this. In this section, we will be looking at the following concepts:

- Unity's technical requirements
- Unity versioning
- Installing Unity with Unity Hub

First, we will discuss what is necessary to run Unity on our computers.

Unity's technical requirements

To run the Unity 2023 editor, your computer will need to meet the requirements specified here:

<https://docs.unity.cn/ru/2021.1/Manual/system-requirements.html#editor>

Here is a summary of what's specified in the link:

- If you use Windows, you need Windows 10.7 version 1909 or greater, Windows 10, or Windows 11. Unity will run only on 64-bit versions of those systems; there is no 32-bit support unless you are willing to work with Unity versions before 2017.x, but that's outside the scope of this book.
- For Mac, you need Big Sur 11.0 to run Apple silicon versions of the editor. In any other case, you can run Intel versions of the editor from Mojave 10.14 or superior.
- For Linux, you need exactly Ubuntu 20.04, Ubuntu 18.04, or CentOS 7.

Regarding the CPU, these are the requirements:

- Your CPU needs to support 64 bits
- Your CPU needs to support SSE2 (most CPUs support it)
- In the case of Macs with Apple silicon, M1 or above is needed

Finally, regarding graphics cards, these are the supported ones:

- On Windows, we need a graphics card with DirectX 10, 11, or 12 support (most modern GPUs support it)
- On Mac, any Metal-capable Intel or AMD GPU will be enough
- On Linux, OpenGL 3.2 or any superior version, or a Vulkan-compatible card from Nvidia and AMD is supported

Note that these are not the requirements for a user to play your game, but for you to use the editor. For the requirements for a user to play your game, please read the following documentation:

<https://docs.unity.cn/ru/2021.1/Manual/system-requirements.html#player>

Now that we know the requirements, let's discuss the Unity installation management system.

Unity versions

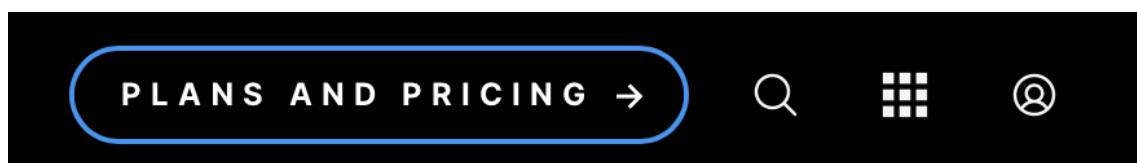
Unity releases a new major version each year—at the time of writing, 2023.1—and during that year it receives an update with new features, which is planned to be 2023.2 at the time of writing this book. Near the end of the year or during the beginning of the next one, an **LTS (long-term support)** version is released, which will be 2023.3 for this edition of the book, marking the end of new features being added to that year's version of the engine. After that, the next year's edition of the engine is released, and the cycle repeats. LTS versions have the benefit that they are planned to be updated bi-weekly with bug fixes for 2 years, while new major versions of Unity release. That's the reason most companies stick to LTS versions of the engine: because of its stability and long-term support. In this book, we will be using 2023.1 just to explore the new features of the engine, but consider sticking to LTS versions when developing commercial game titles. Considering this, you may need to have several versions of Unity installed in case you work on different projects made with different versions. You may be wondering why you can't just use the latest version of Unity for

every project, but there are some problems with that. In newer versions of Unity, there are usually lots of changes to how the engine works, so you may need to rework lots of pieces of the game to upgrade it, including third-party plugins. It can take lots of time to upgrade the whole project, and that can push the release date back. Maybe you need a specific feature that comes with an update that will help you. In such a case, the cost of upgrading may be worthwhile. For projects that are maintained and updated for several years, developers are used to updates only to the latest LTS versions of the editor, although this policy may vary from case to case. Managing different projects made with different Unity versions, and installing and updating new Unity releases, all used to be a huge hassle. Thus, **Unity Hub** was created to help us with this, and it has become the default way to install Unity. Despite this, it is not necessary for installing Unity, but we will keep things simple for now and use it. Let's look closer into it.

Installing Unity with Unity Hub

Unity Hub is a small piece of software that we will install before installing Unity. It centralizes the management of all your Unity projects and installations. You can get it from the official Unity website. The steps to download it change frequently, but at the time of writing this book, you need to do the following:

1. Go to unity.com.
2. Click on the **PLANS AND PRICING** button, as shown in the following screenshot:



*Figure 1.1: The **PLANS AND PRICING** button on Unity's website*

3. Click on the **Student and hobbyist** tab; then, under the **Personal** section, click on the **Get started** button, as illustrated in the following screenshot:

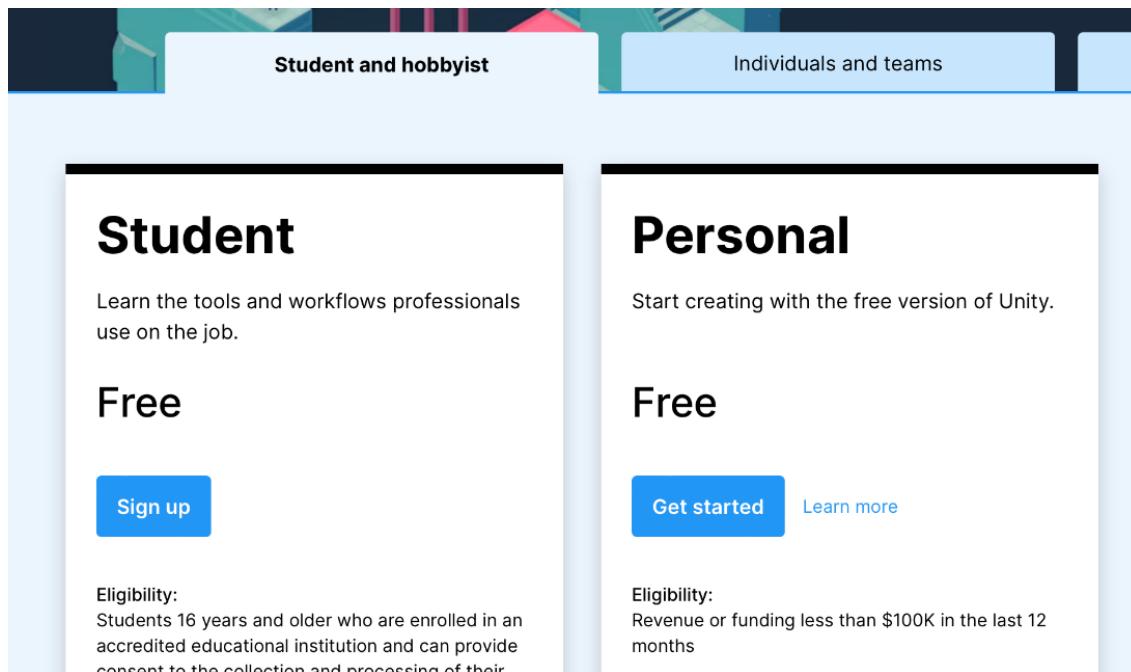


Figure 1.2: Choosing an individual/free license

4. Scroll down to the section saying **1. Download the Unity Hub** and click on the **Download** button according to your operating system. For Windows, click **Download for Windows**, and for Mac, click on **Download for Mac**. For Linux, there is an **Instructions for Linux** button with further info about how to install on that platform, but we won't be covering Unity in Linux in this book:

1. Download the Unity Hub

Follow the instructions onscreen for guidance through the installation process and setup.

[Download for Windows](#)

[Download for Mac](#)

[Instructions for Linux](#)

Figure 1.3: Starting the download

5. Execute the downloaded installer.
6. Follow the instructions of the installer, which will mostly involve clicking **Next** all the way to the end.

Now that we have Unity Hub installed, we must use it to install a specific Unity version. You can do this with the following steps:

1. Start Unity Hub.
2. If prompted to install a Unity version and/or create a license, please skip these steps with the corresponding Skip buttons (which may vary according to the Unity Hub version). This way to install Unity and licenses is only available the first time you run Unity Hub, but we are going to learn an alternative approach that works after the initial setup.
3. Log in to your account by clicking on the "person" icon in the top-left part of the window and selecting Sign in:

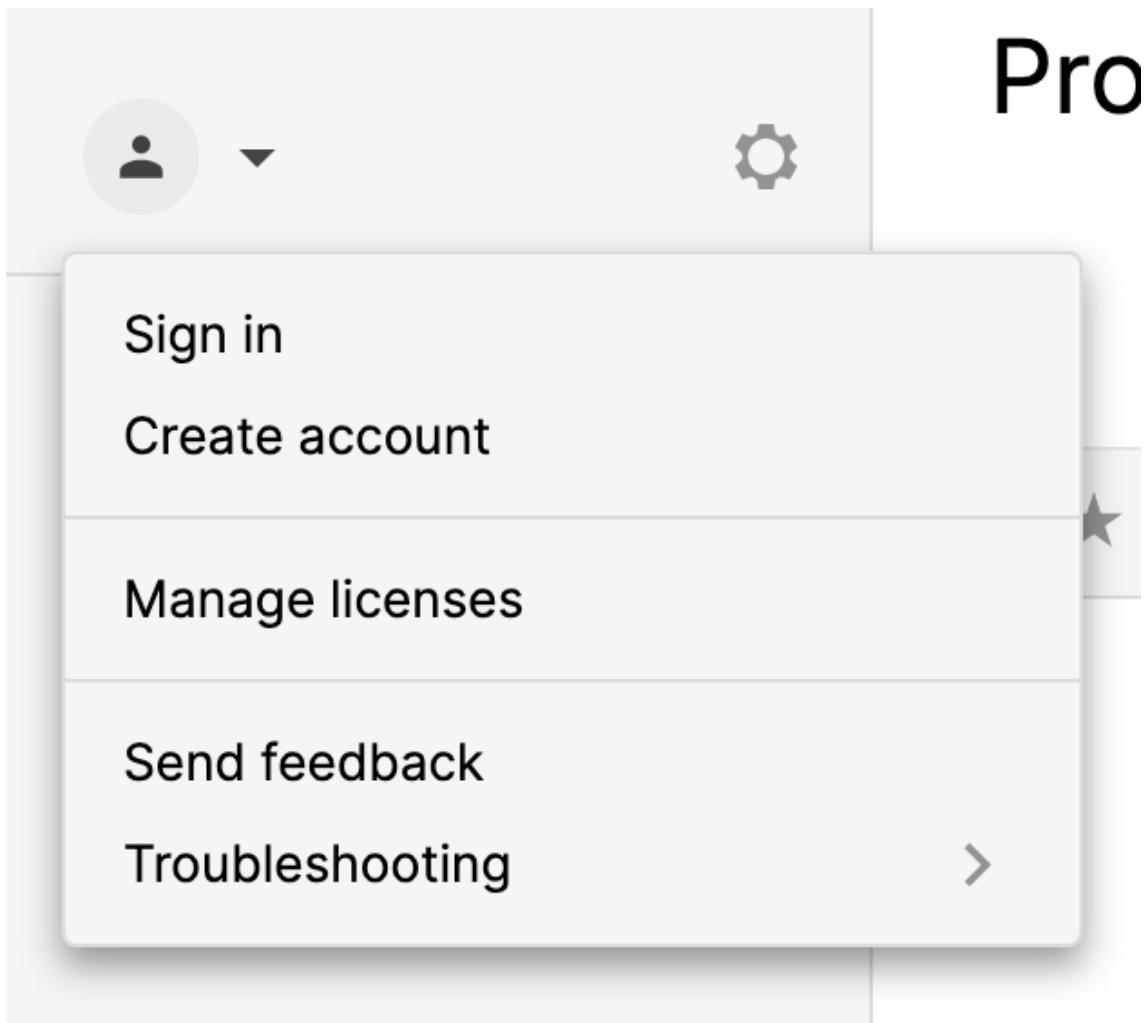


Figure 1.4: Signing in to Unity Hub

4. Here, you also have the option to create a Unity account if you haven't already, as illustrated in the link labeled **create one** that appears in the Unity login prompt in the following screenshot:

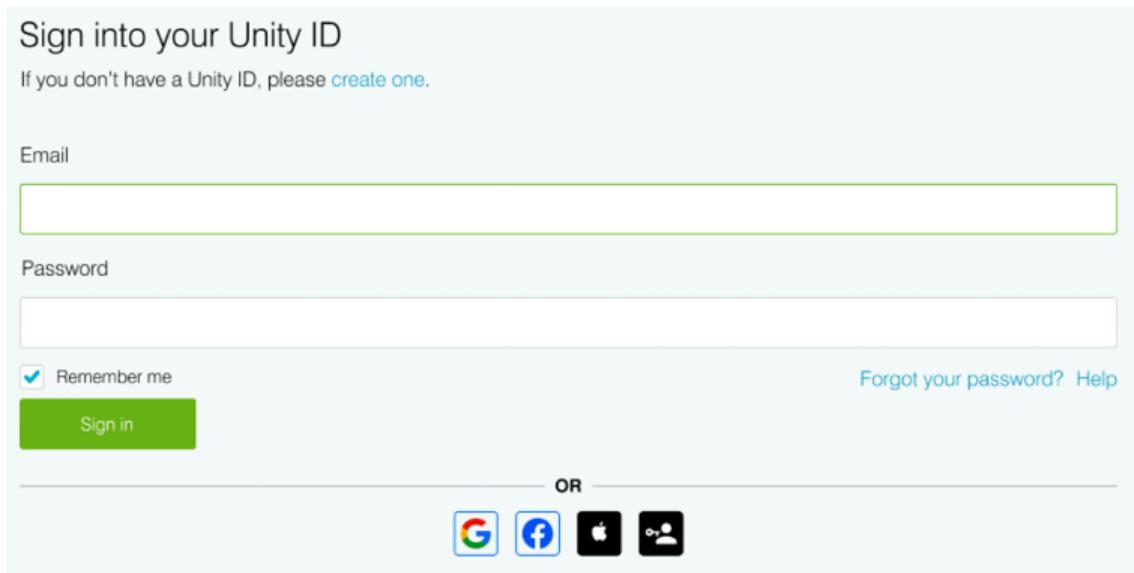


Figure 1.5: Logging in to Unity Hub

5. Follow the steps on the installer and then you should see a screen like the one in the next image. If it is not the same, try clicking the **Learn** button in the top-left part of the screen:

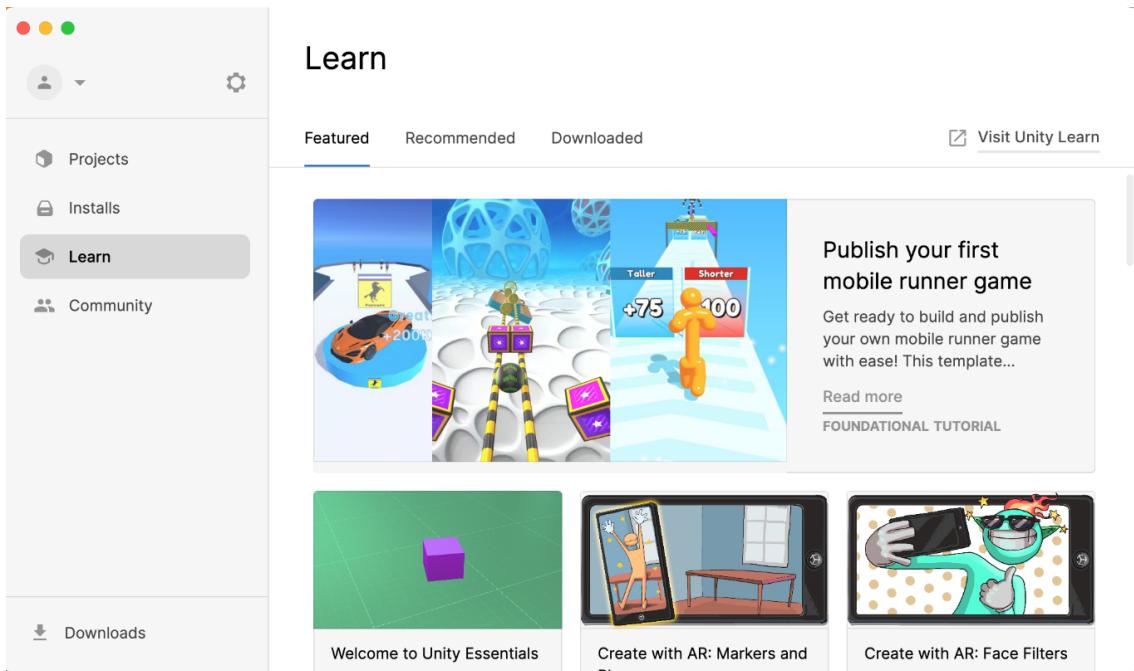


Figure 1.6: The Unity Hub window

6. Click on the **Installs** button and check if you have **Unity 2023** listed there.
7. If not, press the **Install Editor** button in the top-right corner. This will show a list of Unity versions that can be installed from here:

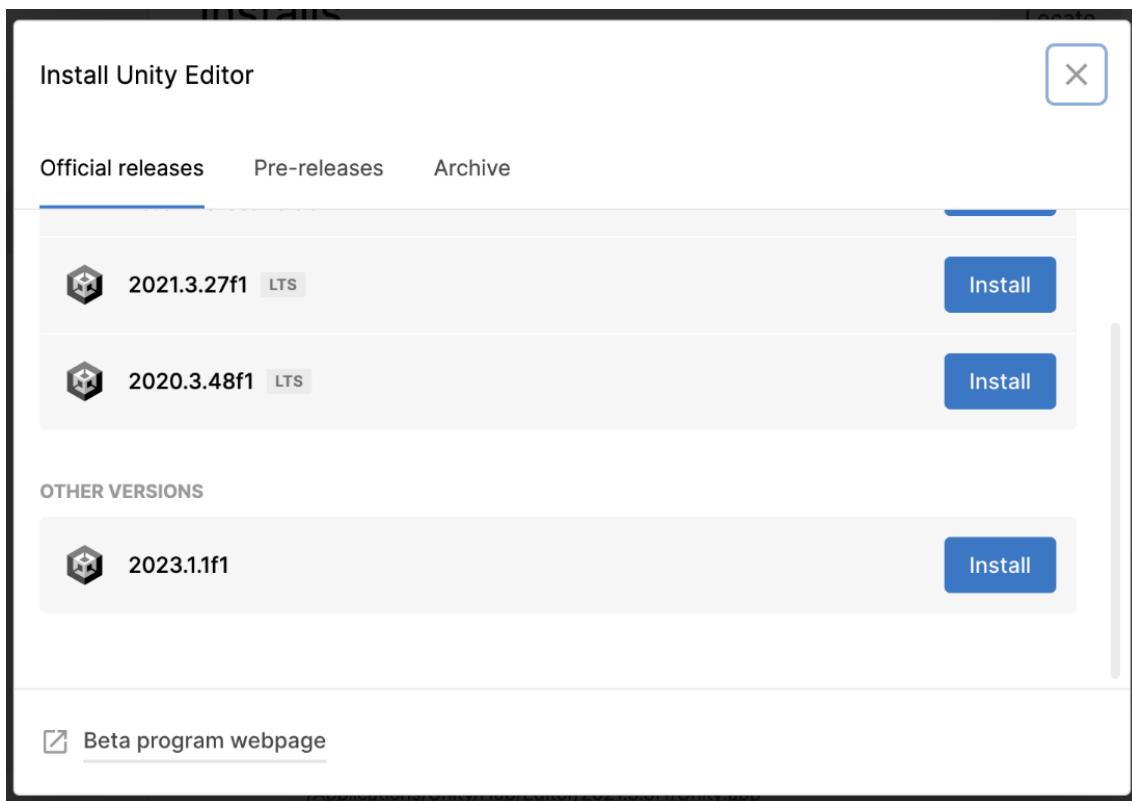


Figure 1.7: Unity versions available to install

8. You will see three tabs here. **Official releases** contains the latest versions of each major release already released. **Pre-releases** contains alpha and beta releases of Unity, so you can participate in these programs and test new features before they are officially released. **Archive** contains a link to the **Unity Download Archive**, which contains every single Unity version released. For example, the official release at the moment of writing this is 2023.1.q, but the project is being

developed in 2023.1.1, so you can install the correct version from the archive.

9. Locate Unity 2023.1 in the **Official releases** tab (or, if you can't find it, in the **Archives** tab).
10. Click on the **Install** button at the right of Unity 2023.1.XXf1, where XX will vary according to the latest available version. At the moment of writing this book, we are using 2023.1.14f1. You might need to scroll down to find this version. If not present, install the latest 2023 version available (for example, 2023.2.XX or 2023.3.XX). Newer versions might vary from what is seen in the book. If you find the images in the book too different, consider looking for Unity 2023.1.14 in the archive.
11. A modules selection window will show up. Make sure the **Visual Studio** feature is checked. While this program is not needed to work in Unity, we will be using it later in the book. If you already have a C# IDE installed, feel free to skip it.
12. Now, click the **Continue** button:

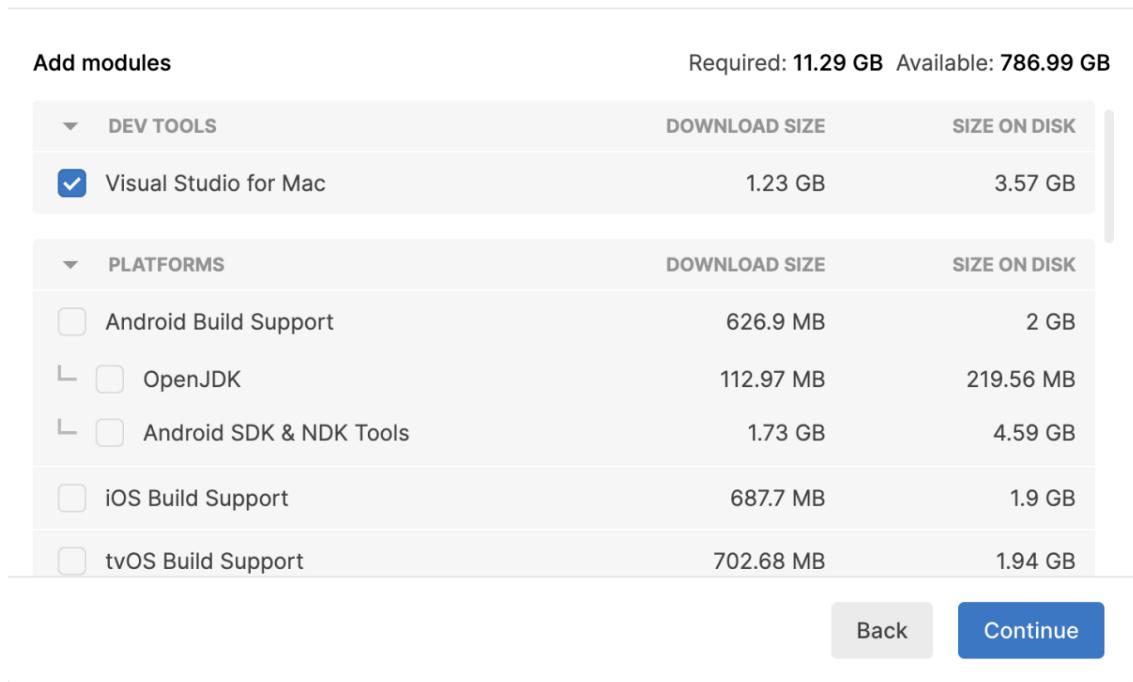


Figure 1.8: Selecting Visual Studio

13. Accept Visual Studio's terms and conditions and then click **Install**:

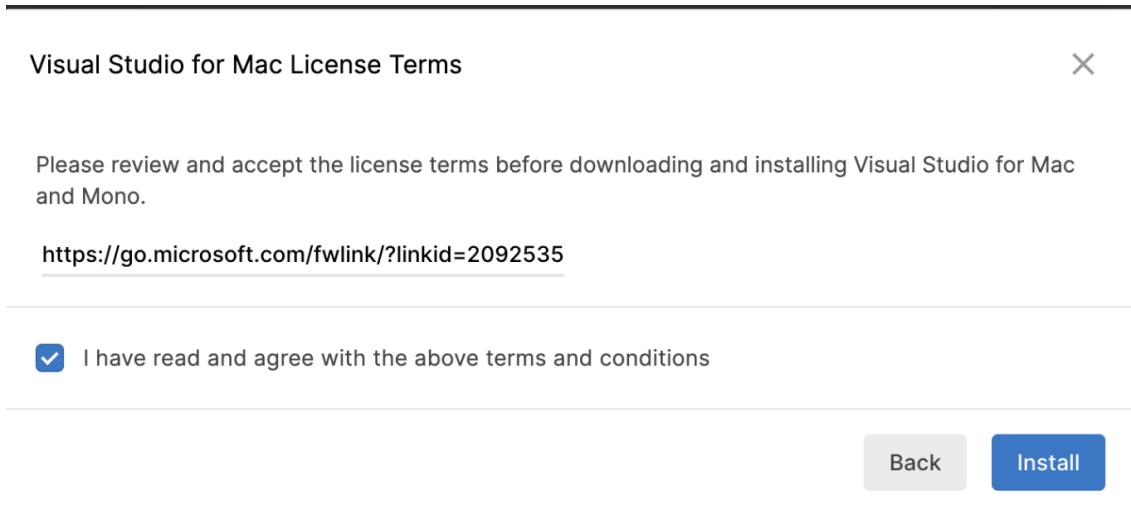


Figure 1.9: Accepting Visual Studio's terms and conditions

It is important to note that Visual Studio is the program we will use in *Chapter 5, Unleashing the Power of C# and Visual Scripting*, to create our code. We do not need the other Unity features right now, but you can go back later and install them if you need them.

14. You will see the selected Unity version downloading and installing. Wait for this to finish. If you don't see it, click the **Downloads** button to reopen it:

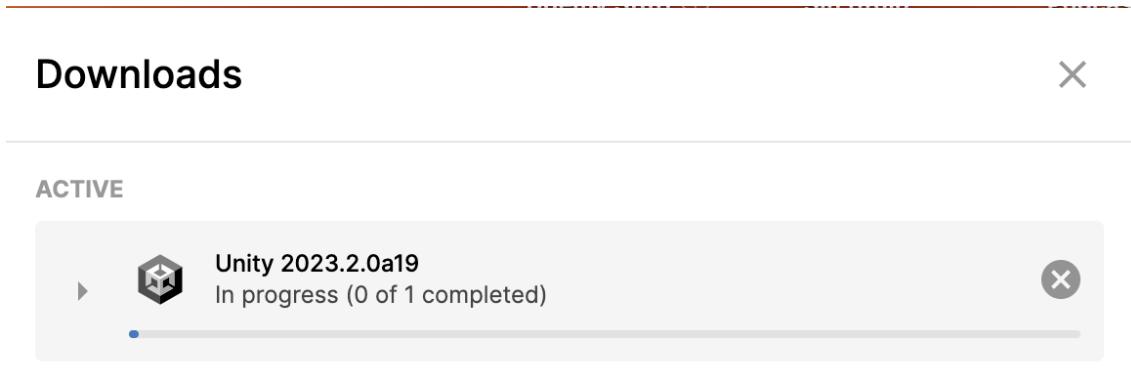


Figure 1.10: Currently active Unity Hub downloads

15. If you decided to install Visual Studio, after Unity has finished installing, the Visual Studio Installer will automatically execute. It will download an installer that will download and install Visual Studio Community:

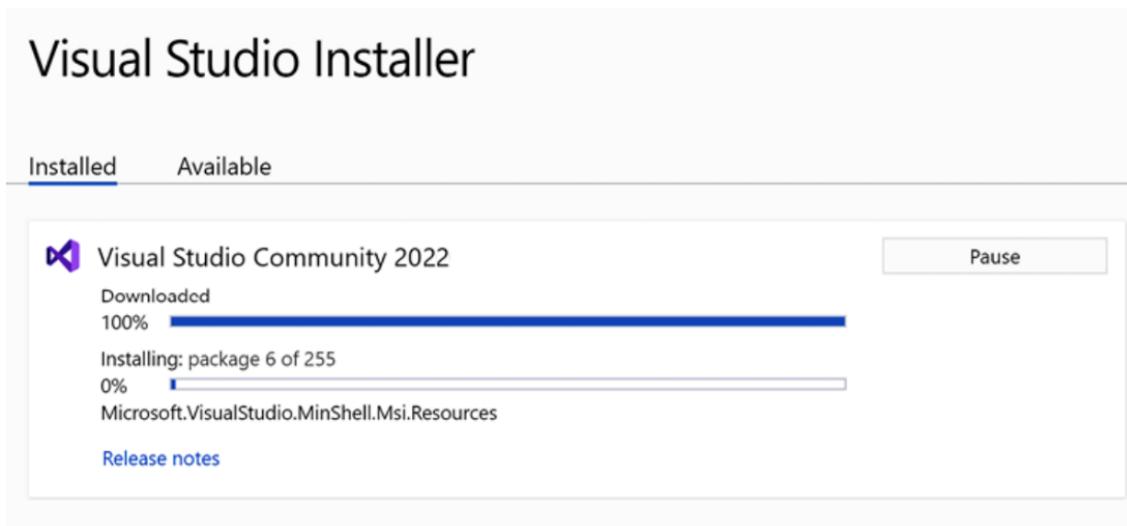


Figure 1.11: Installing Visual Studio

16. To confirm everything worked, you must see the selected Unity version in the list of **Installs** of Unity Hub:

Installs

All Official releases Pre-releases

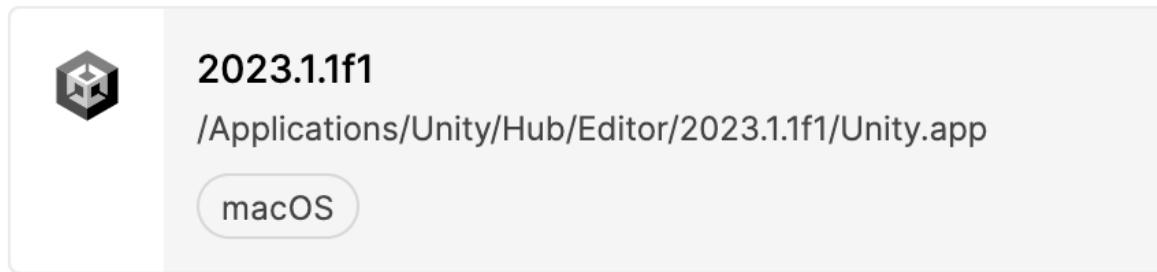


Figure 1.12: Available Unity versions

Now that we have installed Unity and Visual Studio through Unity Hub on our computer, before using Unity, we need to acquire and install a free license to make it work by doing the following:

1. Click the **Manage licenses** button in the top-right corner of the Unity Hub. If you don't see it, click your account icon in the top-left corner and click **Add licenses** there:

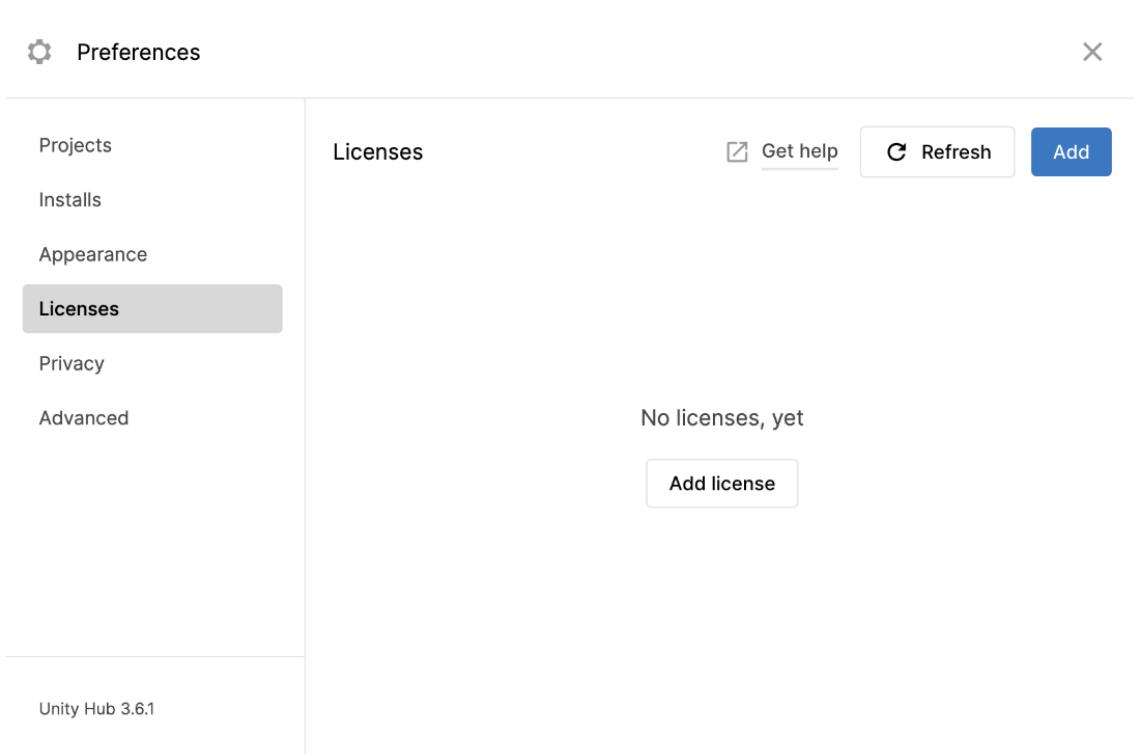


Figure 1.13: The Add licenses button to press in order to acquire a free license

2. Click the **Add** button in the **Licenses** list window:

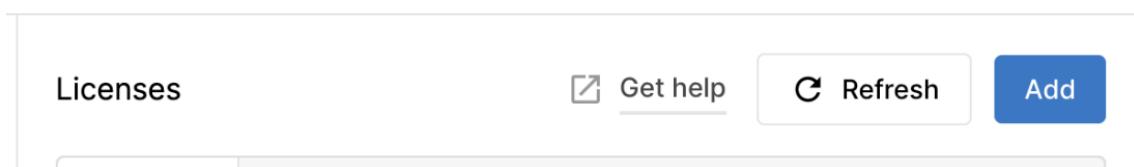


Figure 1.14: The Licenses list window's Add button

3. Click the Get a free personal license button:

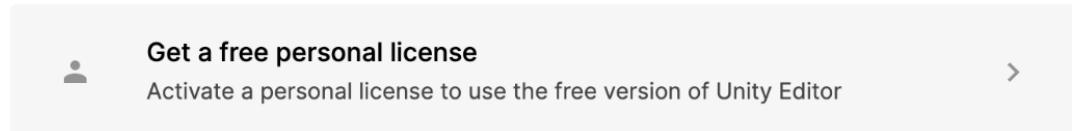


Figure 1.15: Option to get a free personal license

4. Read and accept the terms and conditions if you agree with them by clicking the **Agree and get personal edition license** button:

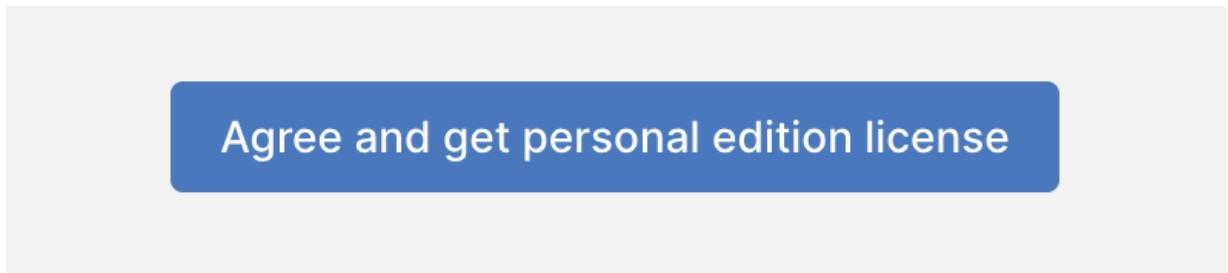


Figure 1.16: The button to accept the terms and conditions

With that, we now have a valid license to use Unity in our account. Remember that the preceding steps may be different in new Unity Hub versions, so just try to follow the flow that Unity designed—most of the time, it is intuitive. Now it is time to create a project using Unity.

Creating projects

Now that we have Unity installed, we can start creating our game. To do so, we first need to create a project, which is basically a folder containing all the files that your game will be composed of. These files are called **assets** and there are different types of them, such as images, audio, 3D models, script files, and so on. In this section, we will see how to manage a project, addressing the following concepts:

- Creating a project
- Project structure

Let's learn first how to create a blank project to start developing our first project within the book.

Creating a project

As with Unity installations, we will use the Unity Hub to manage projects. We need to follow these next steps to create one:

1. Open Unity Hub and click on the **Projects** button, and then click on **New project**:



Figure 1.17: Creating a new project in Unity Hub

2. Note that if you have more than one version of Unity installed through Unity Hub, you may need to select the appropriate version from the dropdown menu at the top of the UI to make sure you use the 2023.1 version you installed before.

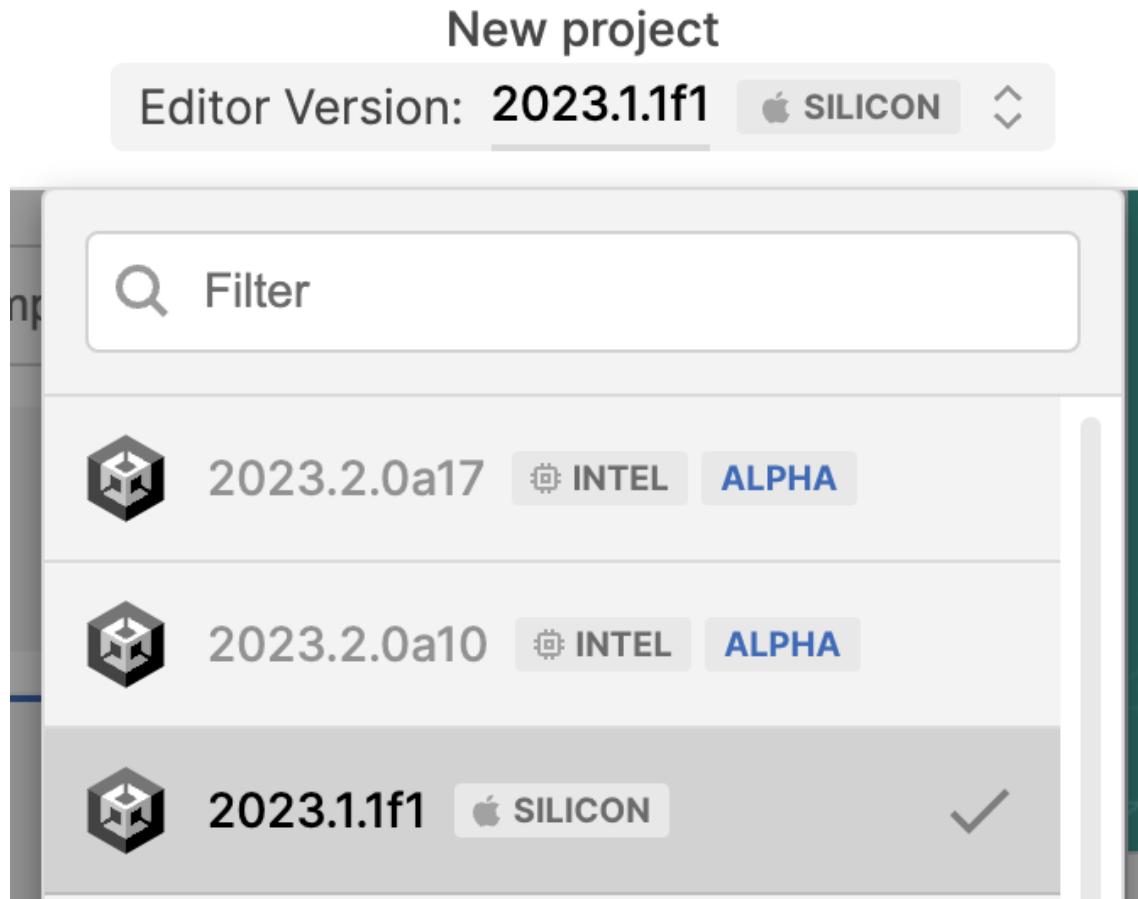


Figure 1.18: Picking the proper Unity version

3. Pick the **3D (URP)** template as we will be creating a 3D game with simple graphics, prepared to run on every device Unity can be executed on, so the **Universal Render Pipeline** (or **URP**) is the better choice for that. In *Chapter 10, Material Alchemy: URP and Shader Graph for Stunning Visuals*, we will be discussing exactly why.
4. If you see a **Download template** button, click it; if not, that means you already have the template:

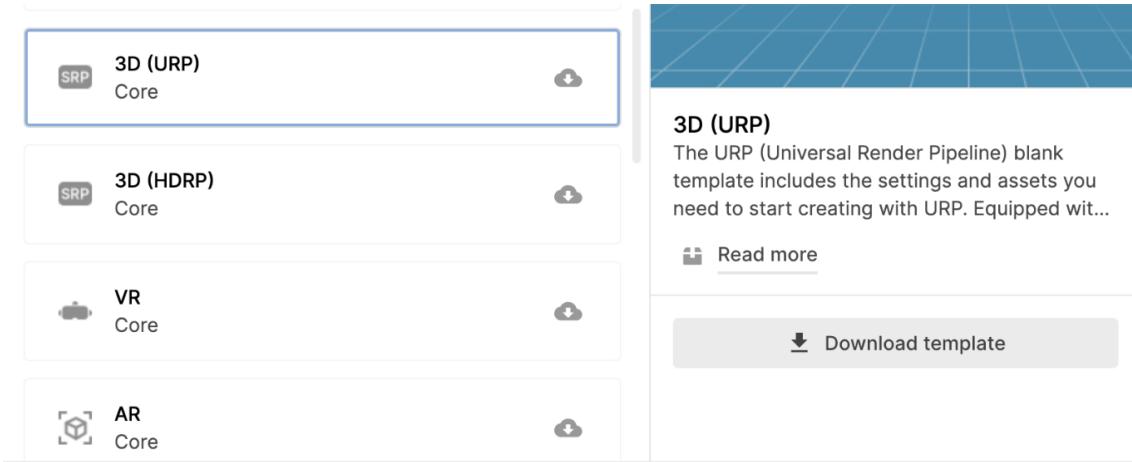


Figure 1.19: Downloading the 3D URP template

Memory

I still remember the first template project I saw in Unity 2.6 back in 2009, before the Unity version numbers matched the release year. It was an island that showcased the Terrain and Water systems. It had flamingos that avoided you when you were close, and it was so fun to walk around. Sadly, the template project we chose won't be as memorable as that one, but it's still a good one to start.

5. Choose a Project name and a Location, and click Create project:

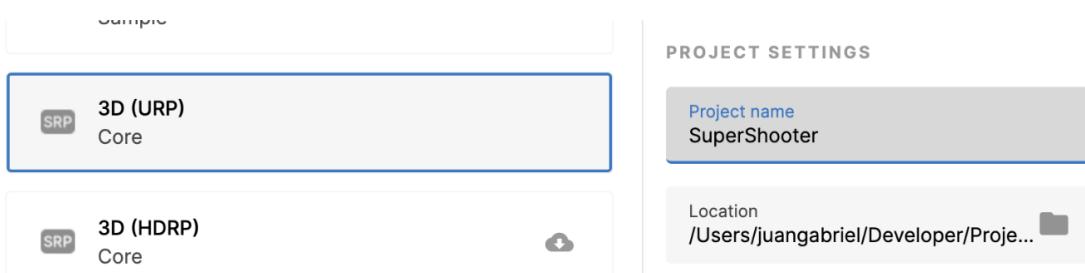


Figure 1.20: Selecting the Universal Render Pipeline template

6. Unity will create and automatically open the project. This can take a while, but after that you will see a window similar to the

one in the following image. You might see the dark-themed editor instead, but for better clarity, we will use the light theme throughout the book. Feel free to keep the dark theme:

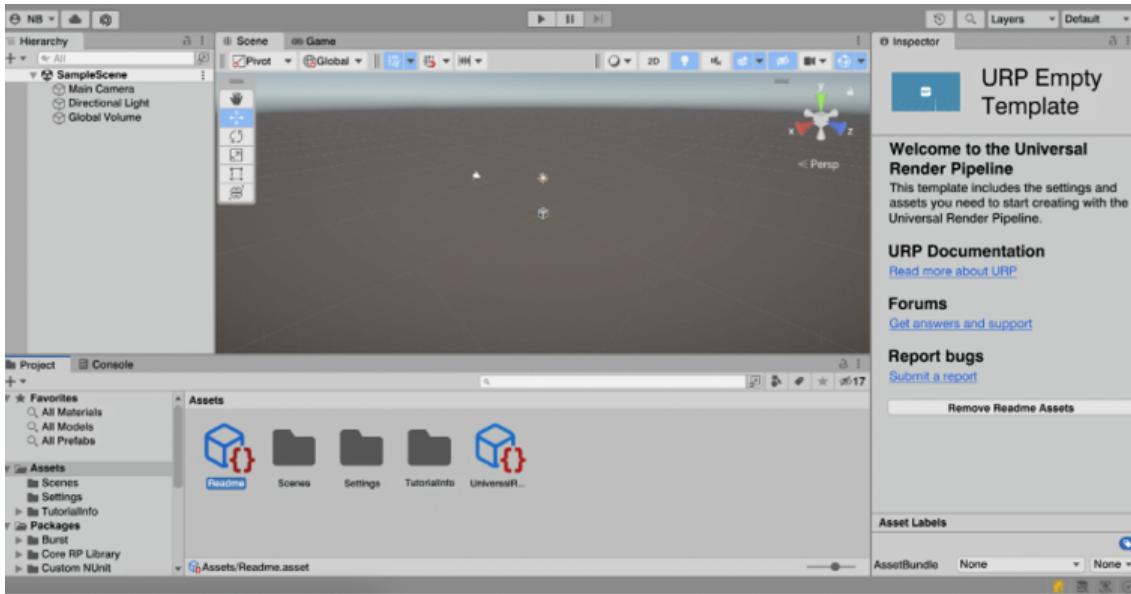
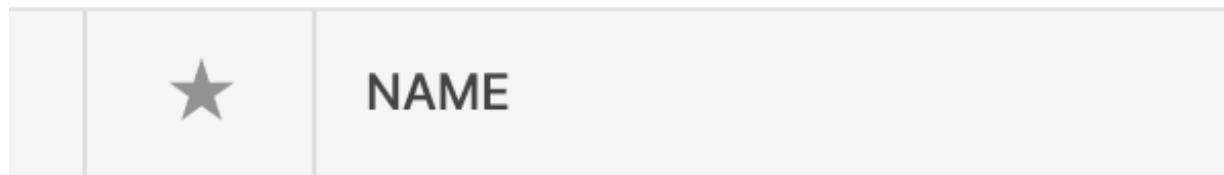


Figure 1.21: The Unity Editor window

7. Close the window, then go back to Unity Hub and pick the project from the list to open it again:

Projects



★ **SuperShooter**
/Users/juangabriel/Developer/Projects

Figure 1.22: Reopening the project

Now that we have created the project, let's explore its structure.

Project structure

We have just opened Unity, but we won't start using it until the next chapter. Now, it's time to see how the project folder structure is composed. To do so, we need to open the folder in which we created the project. If you don't remember where this is, you can do the following:

1. Right-click the **Assets** folder in the **Project** panel, located at the bottom part of the editor.

2. Click the **Show in Explorer** option (if you are using a Mac, the option is called **Reveal in Finder**). The following screenshot illustrates this:

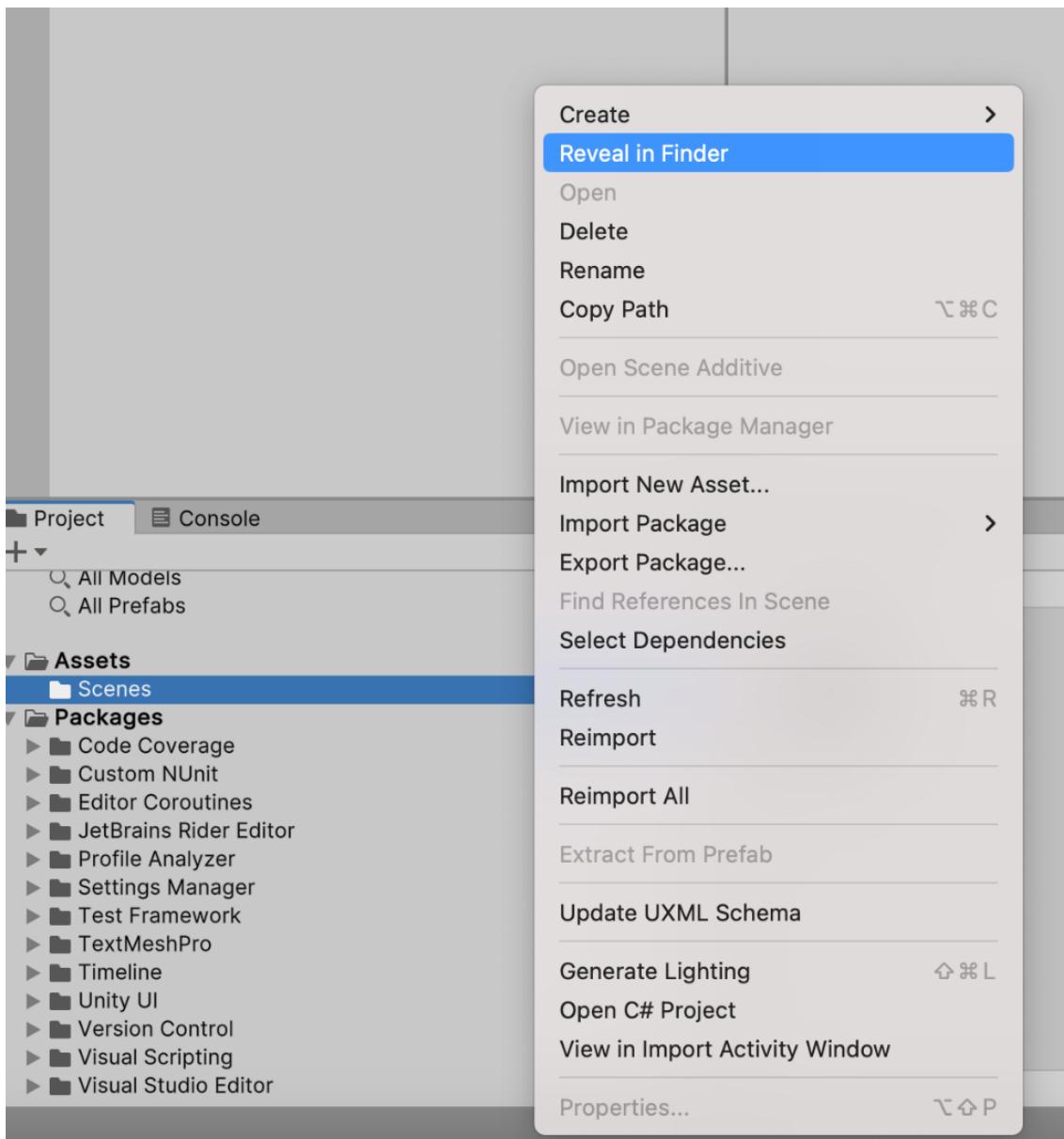


Figure 1.23: Opening the project folder in Explorer

3. Then, you will see a folder structure similar to this one (some files or folders may vary):

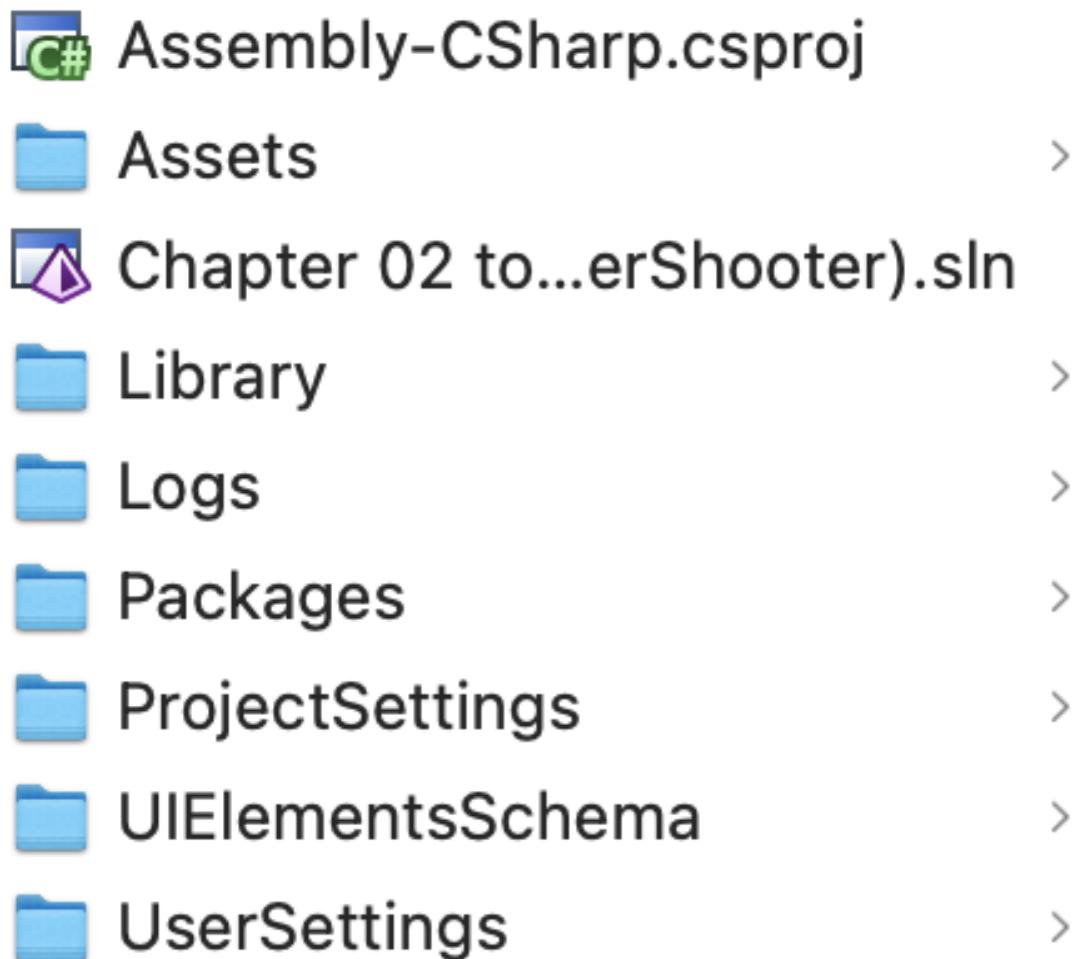


Figure 1.24: Unity project folder structure

Now that we have created and opened our first URP project using Unity Hub, we can find it again in the future in the **Projects** tab of our Unity Hub. From there, we can open it again at any time. If you want to move this project to another PC or send it to a colleague, you can just compress all those files and send them as a ZIP file, but not all the folders are necessary all of the time. The important

folders are `Assets`, `Packages`, and `ProjectSettings`. `Assets` will hold all the files we will create and use for our game, so this is a must. We will also configure different Unity systems to tailor the engine to our game; all the settings related to this are in the `ProjectSettings` and `UserSettings` folders. Finally, we will install different Unity modules or packages to expand its functionality, so the `Packages` folder will hold which ones we are using. It's not necessary to copy the rest of the folders if you need to move the project elsewhere or add it to any versioning system, but let's at least discuss what the `Library` folder is, especially considering it's usually a huge size. Unity needs to convert the files we will use to its own format in order to operate; an example is audio and graphics. Unity supports **MPEG Audio Layer 3 (MP3)**, **Waveform Audio File Format (WAV)**, **Portable Network Graphics (PNG)**, and **Joint Photographic Experts Group (JPG)** files (and much more), but prior to using them, they need to be converted to Unity's internal formats, a process called **Importing Assets**. Those converted files will be in the `Library` folder. If you copy the project without that folder, Unity will simply take the original files in the `Assets` folder and recreate the `Library` folder entirely. This process can take time, and the bigger the project, the more time involved. Keep in mind that you want to have all the folders Unity created while you are working on the project, so don't delete any of them while you work on it, but if you need to move an entire project, you now know exactly what you need to take with you.

Summary

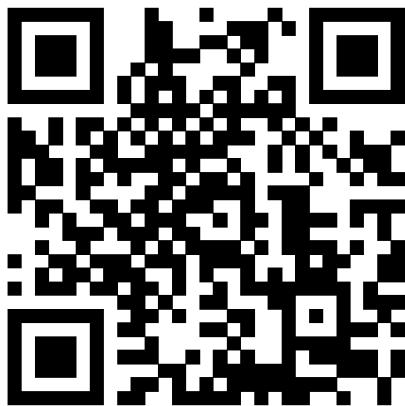
In this chapter, we reviewed how the Unity versioning system works. We also saw how to install and manage different Unity versions using Unity Hub. Finally, we created and managed multiple projects with the same tool. We will use Unity Hub a lot, so it is important to know how to use it initially. Now, we are prepared to

dive into the Unity Editor. In the next chapter, we will start learning the basic Unity tools to author our first level prototype.

2 Crafting Scenes and Game Elements

Join our book community on Discord

<https://packt.link/unitydev>



In this chapter, we will develop some base knowledge of Unity in order to edit a project, and we will learn how to use several Unity editor windows to manipulate our first scene and its objects. We will also learn how an object, or GameObject, is created and composed and how to manage complex scenes with multiple objects, using Hierarchies and Prefabs. Finally, we will review how we can properly save all our work to continue working on it later. Specifically, we will examine the following concepts in this chapter:

- Manipulating scenes
- GameObjects and components
- Understanding object Hierarchies
- Managing GameObjects using Prefabs
- Saving scenes and projects

Manipulating scenes

A **scene** is one of the several types of files (also known as **assets**) in our project. Other types of files include scripts for code, audio files, 3D models, and textures, among others. A "scene" can be used

for different things according to the type of project, but the most common use case is to separate your game into whole sections, the most common ones being the following:

- The main menu
- Level 1, Level 2, Level 3, etc.
- A victory screen and a lose screen
- A splash screen and a loading screen

In this section, we will cover the following concepts related to scenes:

- The purpose of a scene
- The Scene view
- Adding our first GameObject to a scene
- Navigating the Scene view
- Manipulating GameObjects

So, let's take a look at each of these concepts.

The purpose of a scene

The idea of separating your game into scenes is so that Unity can process and load just the data needed for the scene. Let's say you are in the main menu; in such a case, you will have only the textures, music, and objects that the main menu needs to be loaded in **random-access memory (RAM)**, the device's main memory. In that case, there's no need for your game to have loaded the Level 10 boss if you don't need it right now. That's why loading screens exist, just to fill the time between unloading the assets needed in one scene and loading the assets needed in another. Maybe you are thinking that open-world games such as Grand Theft Auto don't have loading screens while you roam around in the world, but they are actually loading and unloading chunks of the world in the background as you move, and those chunks are different scenes that are designed to be connected to each other. The difference between

the main menu and a regular level scene is the objects (also known as **GameObjects** in Unity lingo) they have. In a menu, you will find objects such as backgrounds, music, buttons, and logos, and in a level, you will have the player, enemies, platforms, health boxes, and so on. So, the meaning of your scene depends on what **GameObjects** are put into it. But how can we create a scene? Let's start with the Scene view.

The Scene view

When you open a Unity project, you will see the Unity editor. It will be composed of several **windows** or **panels**, each one helping you to change different aspects of your game. In this chapter, we will look at the windows that help you create scenes. The Unity editor is shown in the following screenshot:

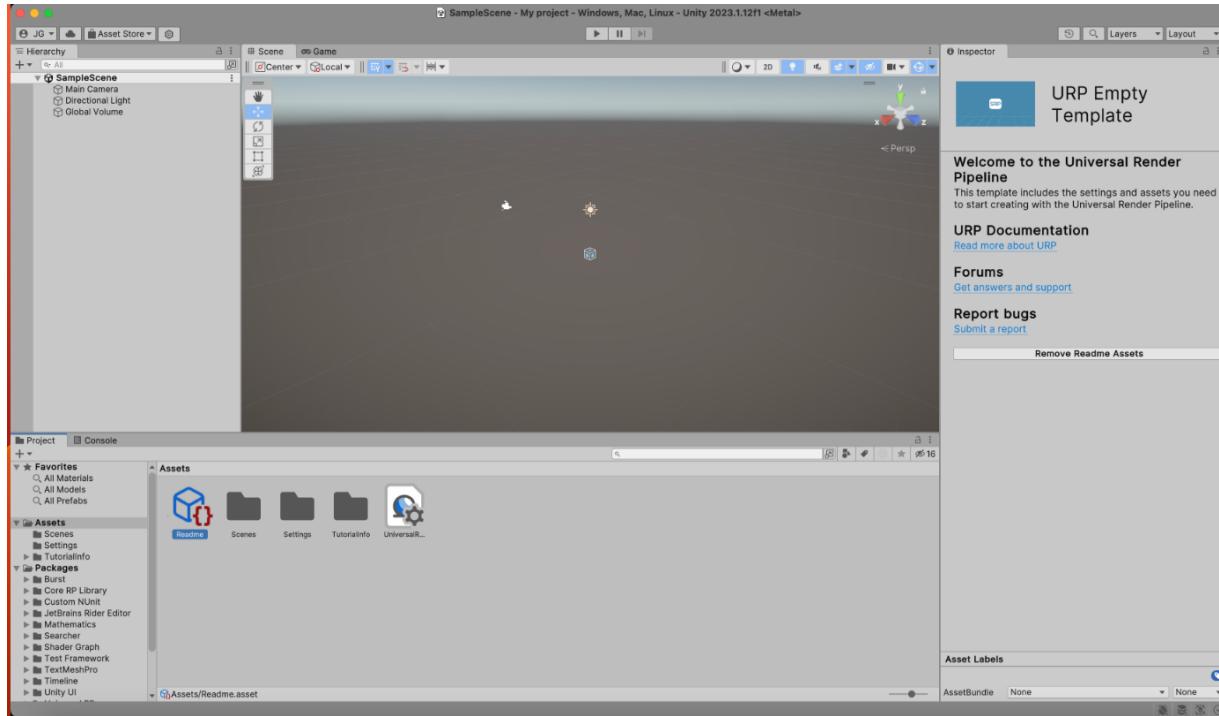


Figure 2.1: The Unity editor

If you have ever programmed any kind of application before, you are probably used to having a starting function such as `Main`, where you

start writing code to create several objects needed for your app. If you were developing a game, you probably create all the objects for the scene there. The problem with this approach is that in order to ensure all objects are created properly, you will need to run the program to see the results, and if something is misplaced, you will need to manually change the coordinates of the object, which is a slow and painful process. Luckily, in Unity, we have the Scene view, an example of which is shown in the following screenshot:

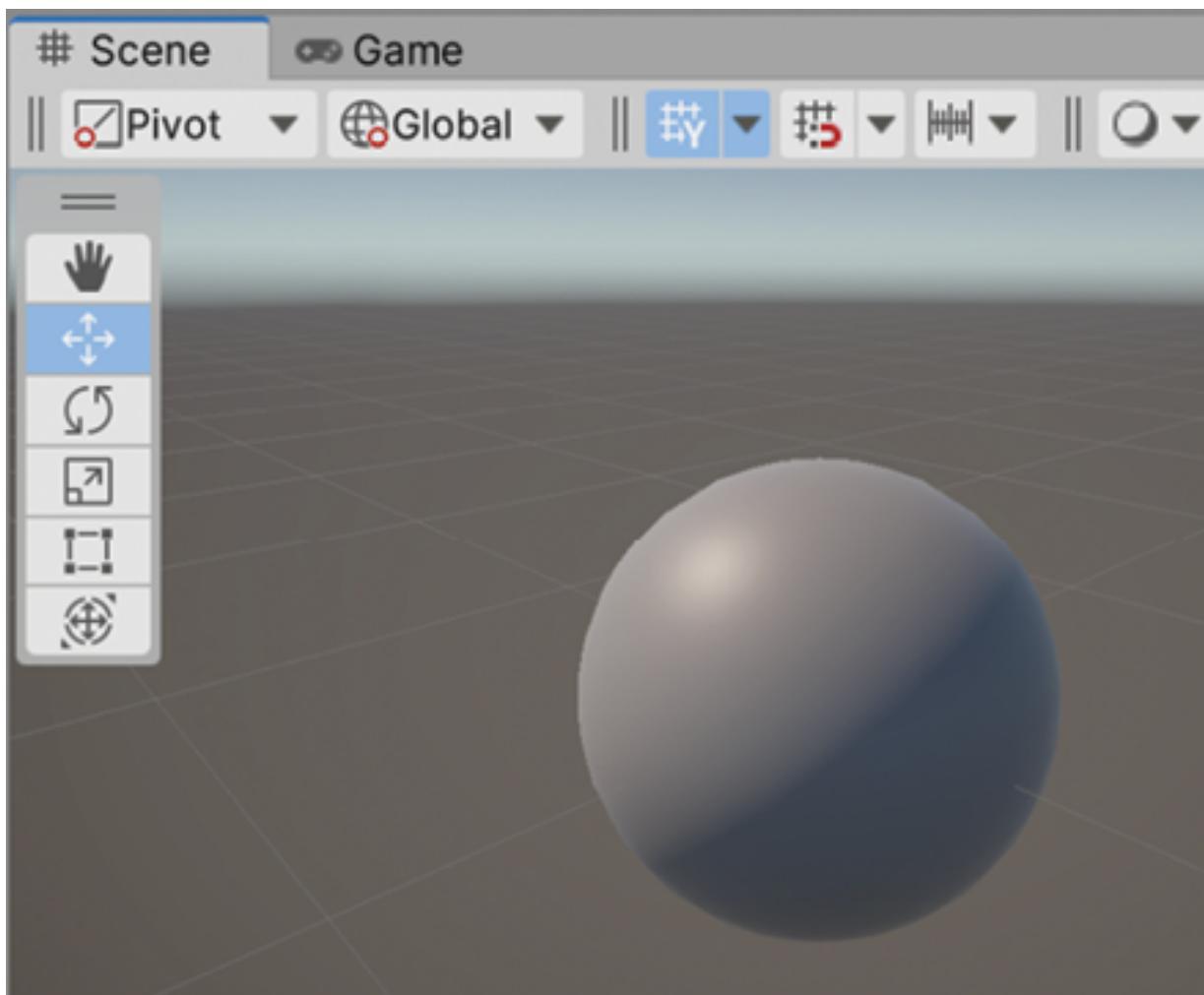


Figure 2.2: The Scene view

This window is an implementation of the classic **WYSIWYG** (**What You See Is What You Get**) concept. Here, you can create objects and place them all over the scene, all through a scene

previsualization where you can see how the scene will look when players play the game. However, before learning how to use this scene, we need to have an object in the scene, so let's create our first object.

Adding our first GameObject to the scene

The project template we choose when creating the project comes with a blank scene ready to work with, but let's create our own empty scene to see how to do it ourselves. To do that, you can simply use the **File | New Scene** menu to create an empty new scene, as illustrated in the following screenshot:

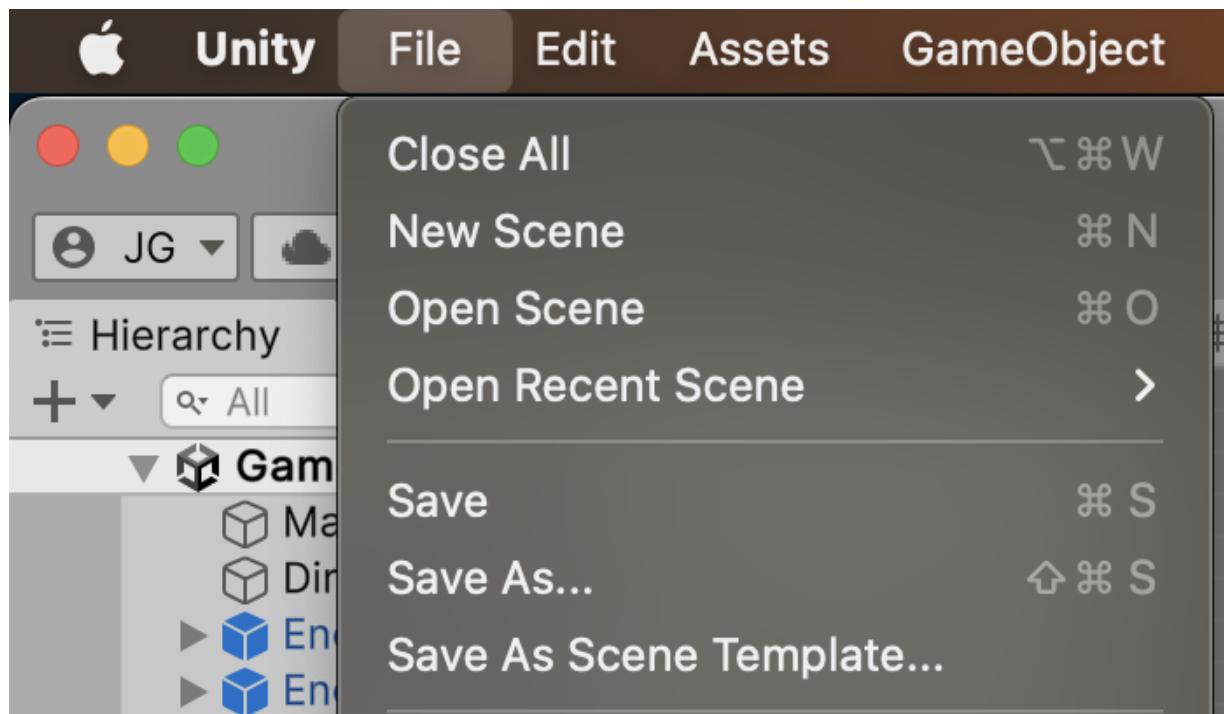


Figure 2.3: Creating a new scene

After clicking **New Scene**, you will see a window to pick a scene template; here, select the **Basic (URP)** template. A template defines which objects the new scene will have, and in this case, our template will come with a basic light and a camera, which will be

useful for the scene we want to create. Once selected, just click the **Create** button:

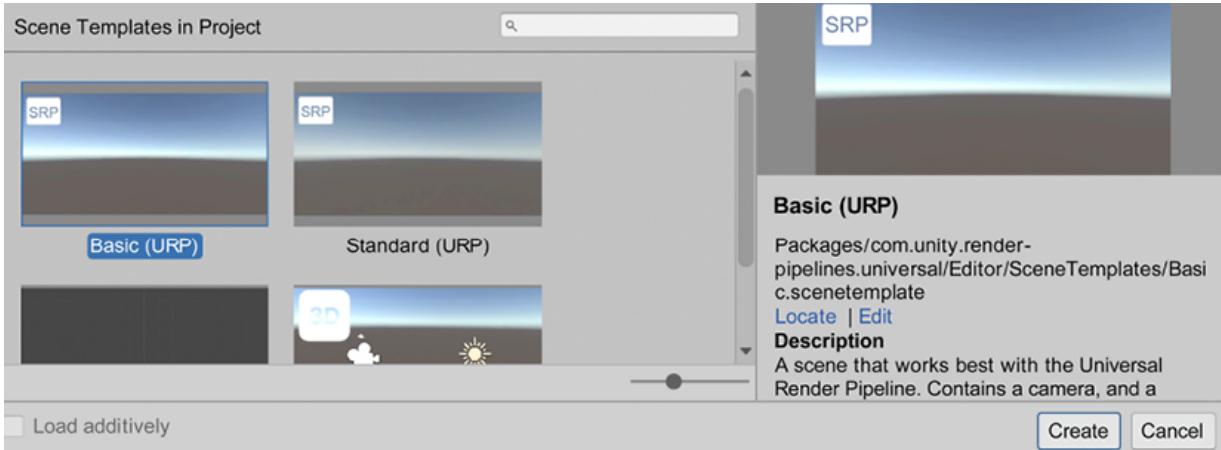


Figure 2.4: Selecting the scene template

Now that we have our empty scene, let's add GameObjects to it. We will learn several ways of creating GameObjects throughout the book, but for now, let's start using some basic templates Unity provides us. In order to create them, we will need to open the **GameObject** menu at the top of the Unity window, and it will show us several template categories, such as **3D Object**, **2D Object**, **Effects**, and so on, as illustrated in the following screenshot:

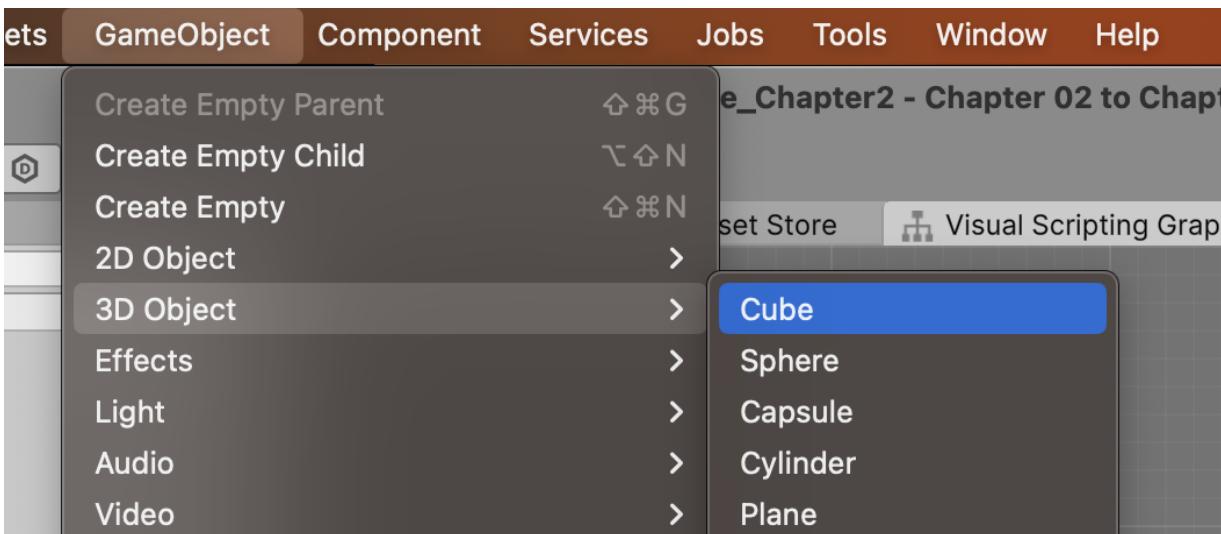


Figure 2.5: Creating a cube

Under the **3D Object** category, we will see several 3D primitives such as **Cube**, **Sphere**, **Cylinder**, and so on, and while using them is not as exciting as using beautiful, downloaded 3D models, remember that we are only prototyping our level at the moment. This is called **gray-boxing**, which means that we will use lots of prototyping primitive shapes to model our level so that we can quickly test it, seeing if our idea is good enough to start the complex work of converting it to a final version. I recommend you pick the **Cube** object to start because it is a versatile shape that can represent lots of objects. So, now that we have a scene with an object to edit, the first thing we need to learn to do with the Scene view is to navigate through the scene.

Navigating the Scene view

In order to manipulate a scene, we need to learn how to move through it to view the results from different perspectives. There are several ways to navigate the scene, so let's start with the most common one, the first-person view. This view allows you to move through a scene using first-person-shooter-like navigation, using the mouse and the WASD keys. To navigate like this, you will need to press and hold the right mouse button, and while doing so, you can:

- Move the mouse to rotate the camera around its current position
- Press the WASD keys to move the position of the camera, always holding the right click
- You can also press Shift to move faster
- Press the Q and E keys to move up and down

Another common way of moving is to click an object to select it (the selected object will have an orange outline), and then press the F key to focus on it, making the Scene view camera immediately move to a position where we can look at that object more closely. After that, we can press and hold the left Alt key on Windows, or Option

on a Mac, along with the left mouse click, to finally start moving the mouse and "orbit" around the object. This will allow you to see the focused object from different angles to check if every part of it is properly placed, as demonstrated in the following screenshot:

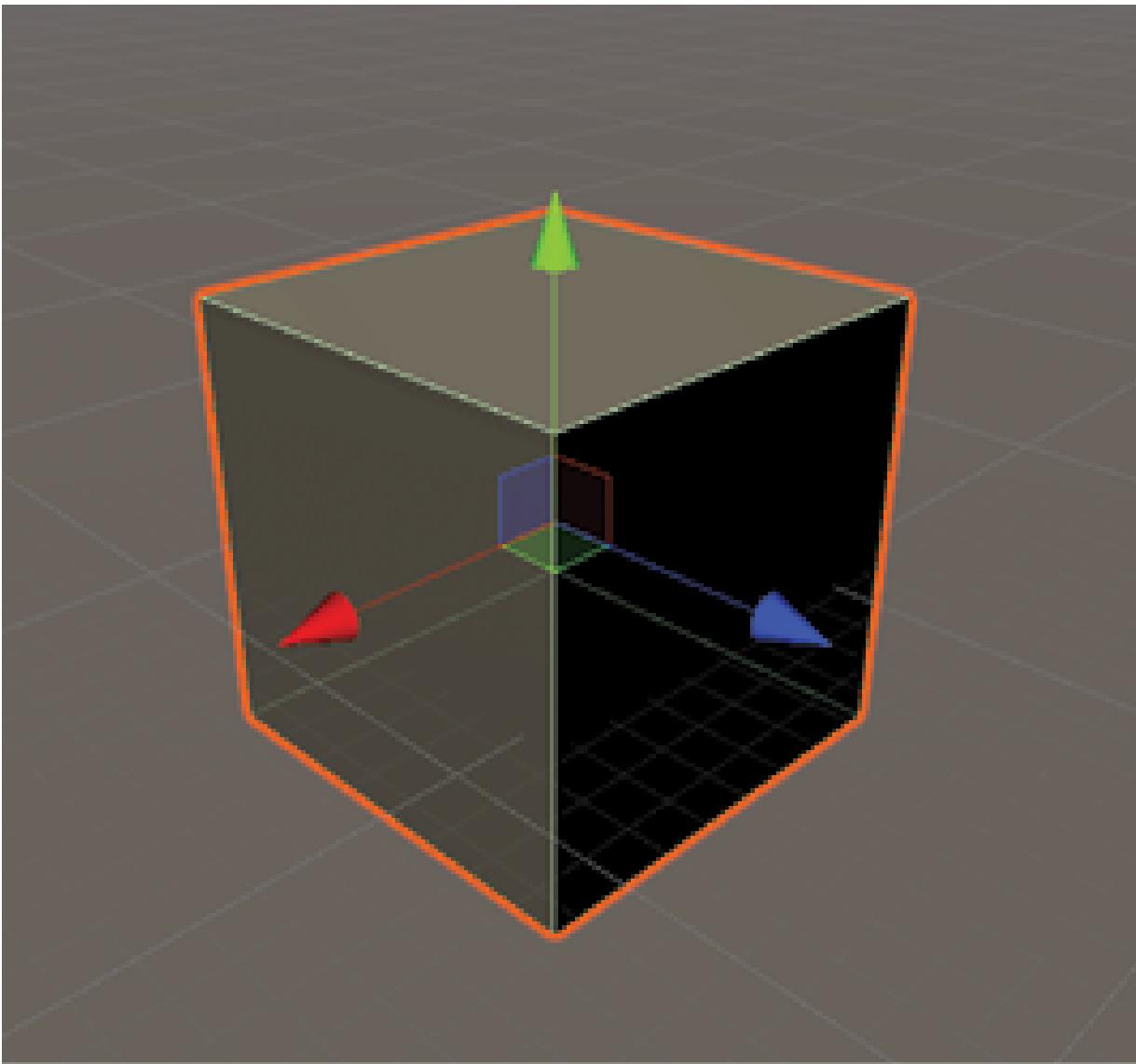


Figure 2.6: Selecting an object

Now that we can move freely through the scene, we can start using the Scene view to manipulate GameObjects.

Manipulating GameObjects

Another use of the Scene view is to manipulate the locations of the objects. In order to do so, we first need to select an object, and then press the **Transform** tool in the top-left corner of the Scene view. You can also press the Y key on the keyboard once an object is selected to activate the same Transform tool:





Figure 2.7: The transformation tool

This will show what is called the **Transform** gizmo over the selected object. A **gizmo** is a visual tool overlaid on top of the selected object, used to modify different aspects of it. In the case of the **Transform Gizmo**, it allows us to change the position, rotation, and scale of the object, as illustrated in Figure 2.8. Don't worry if you don't see the cube-shaped arrows outside the sphere—we will enable them in a moment:

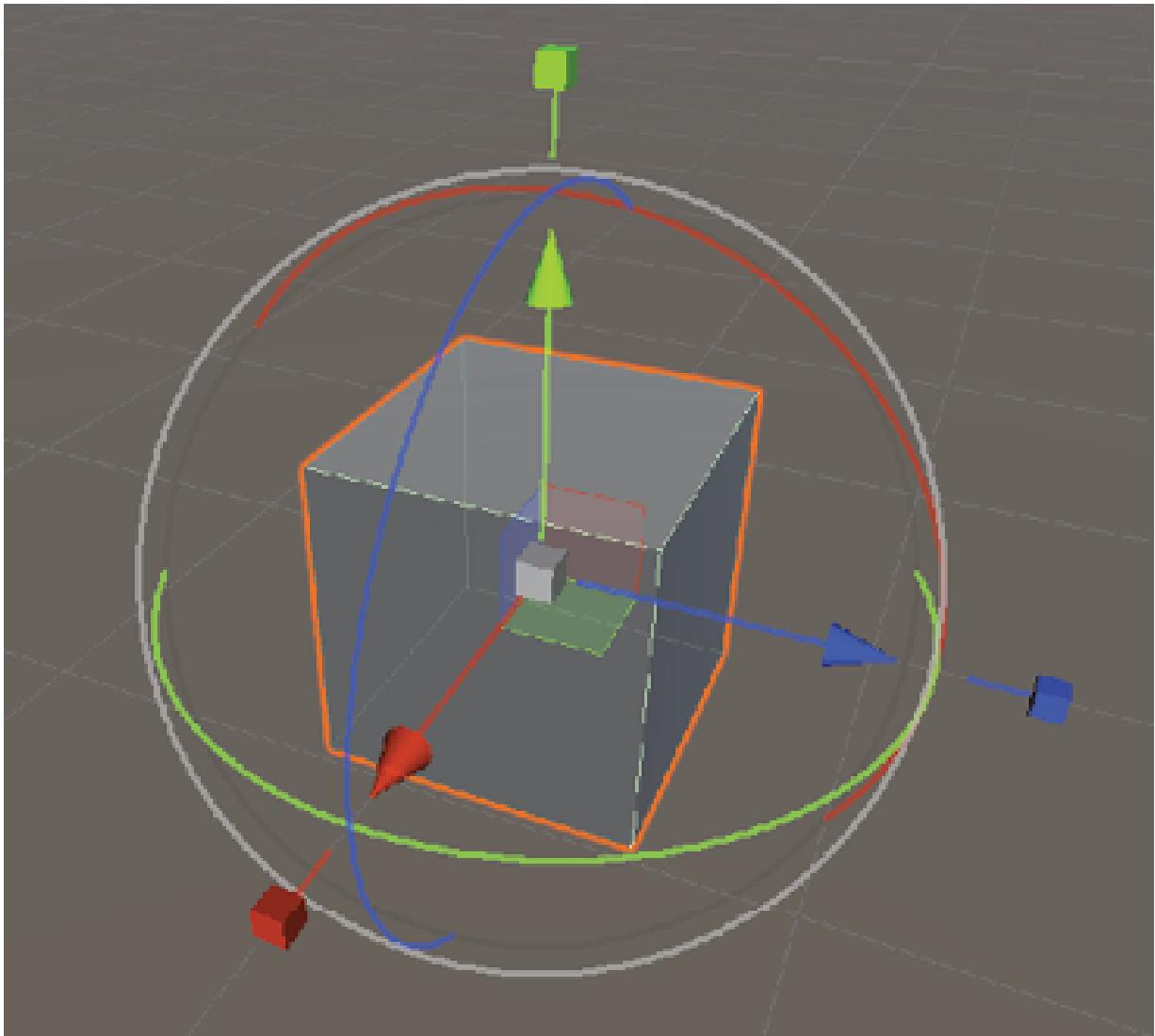


Figure 2.8: The Transform gizmo

Let's start translating the object, which is accomplished by dragging the red, green, and blue arrows inside the gizmo's sphere. While you do this, the object will move along the selected axis. An interesting concept to explore here is the meaning of the colors of these arrows. If you pay attention to the top-right area of the Scene view, you will see an axis gizmo that serves as a reminder of those colors' meaning, as illustrated in the following screenshot:

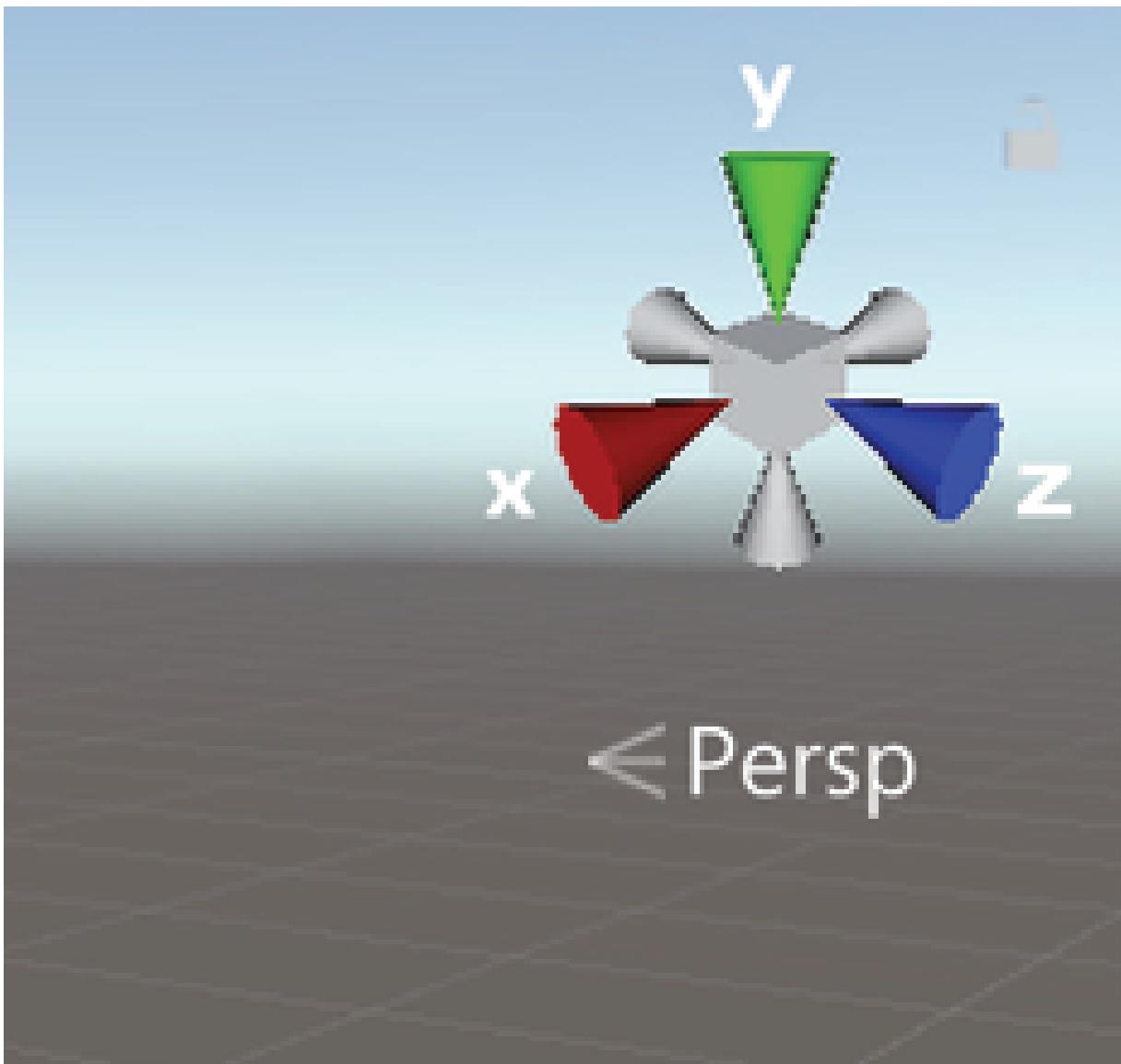


Figure 2.9: The axis gizmo

Computer graphics use the classic 3D **Cartesian coordinate system** to represent objects' locations. The red color is associated with the x-axis of the object, green with the y axis, and blue with the z axis. But what does each axis mean? If you are used to another 3D authoring program like Maya, Blender or 3DS Max, this can be different, but in Unity, the z axis represents the **Forward Vector**, which means that the arrow points along the front of the object; the x axis is the **Right Vector**, and the y axis represents the **Up Vector**. These directions are known as **local** coordinates, and that's

because every object can be rotated differently, meaning each object can point its forward, up, and right vectors elsewhere according to its orientation. The local coordinates will make more sense when used later in the Object Hierarchies section of the chapter, so bear with me on that, but it's worth discussing **global** coordinates now. The idea is to have a single origin point (the zero point) with a single set of forward, right, and up axes that are common across the scene. This way, when we say an object has a global position of $5, 0, 0$, we know that we are referring to a position 5 meters along the global x-axis, starting from the global zero position. The global axes are the ones you see in the top-right axis gizmos previously mentioned.

We just mentioned that a position $(5,0,0)$ would mean 5 meters along the *x*-axis, which implies that the Unity unit system is the meter. While several Unity systems like Physics and Audio follow this assumption (1 unit = 1 meter), it is not necessarily the mandatory unit system. There are several ways to change this and scale the world for convenience, but given that it is only necessary in specific scenarios, in this book we will stick to the meter measuring system.

In order to be sure that we work with local coordinates, meaning we will move the object along its local axes, make sure the **Local** mode is activated in the Scene view, as shown in the following screenshot:

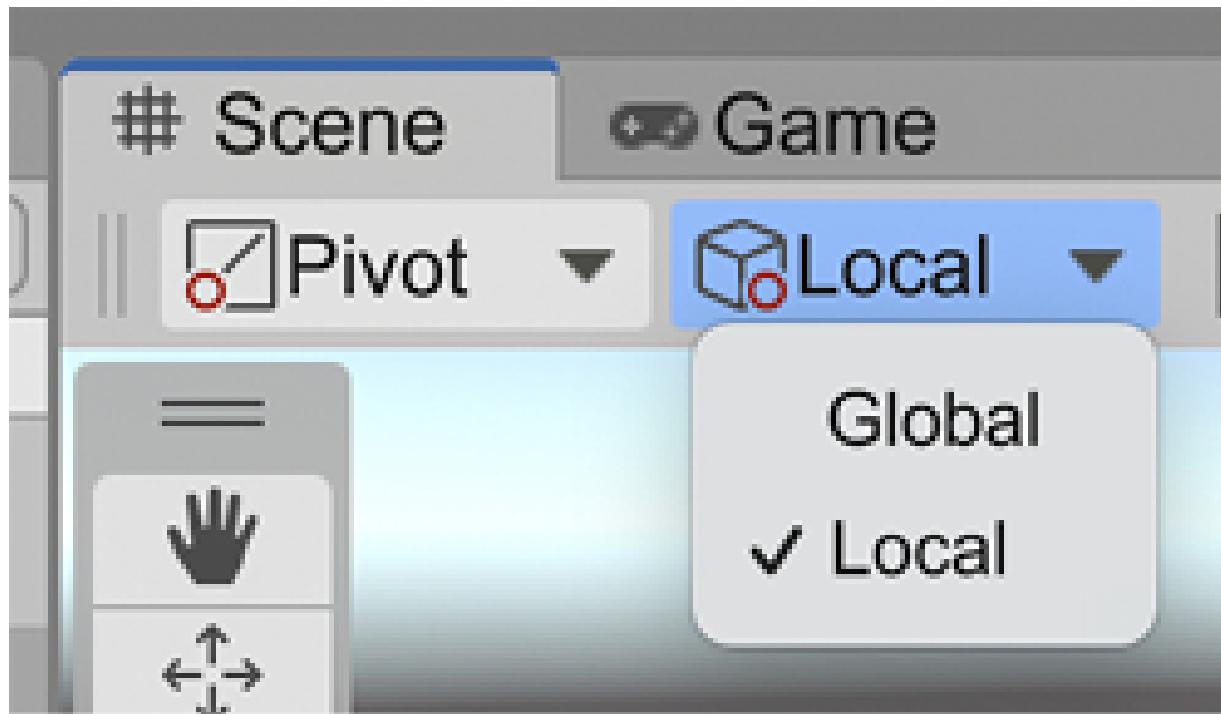


Figure 2.10: Switching pivot and local coordinates

If the right button says **Global** instead of **Local**, just click it and select **Local** from the drop-down options. By the way, try to keep the left button as **Pivot**. If it says **Center**, click and select **Pivot**. The pivot of the object is not necessarily its center, and that depends entirely on the 3D model we use, and where the author of it specifies the object rotation center is located. For example, a car could have its pivot in the middle of its back wheels, so when we rotate, it will respect the real car's rotation center. Editing based on the object's pivot will simplify our understanding of how rotating via C# scripts will work later in Chapter 6. Also, now that we have enabled **Local** coordinates, you should see the cube-shaped arrows seen in Figure 2.8; we will use them in a moment to scale the cube. I know—we are editing a cube, so there is no clear front or right side, but when you work with real 3D models such as cars and characters, they will certainly have those sides, and they must be properly aligned with those axes. If, by any chance in the future, you import a car into Unity and the front of the car points along the x-axis, you will need to align that model along the z axis because the code we

will create to move our object will rely on that convention. Now, let's use this **Transform** gizmo to rotate the object, using the three colored circles around it. If you click and drag, for example, the red circle, you will rotate the object along the x-axis. If you want to rotate the object horizontally, based on the color coding we previously discussed, you will probably pick the x-axis—the one that is used to move horizontally—but, sadly, that's wrong. A good way to look at the rotation is like the accelerator of a motorcycle: you need to take it and roll it. If you rotate the x-axis like this, you will rotate the object up and down. So, in order to rotate horizontally, you would need to use the green circle or the y axis. The process is illustrated in the following screenshot:

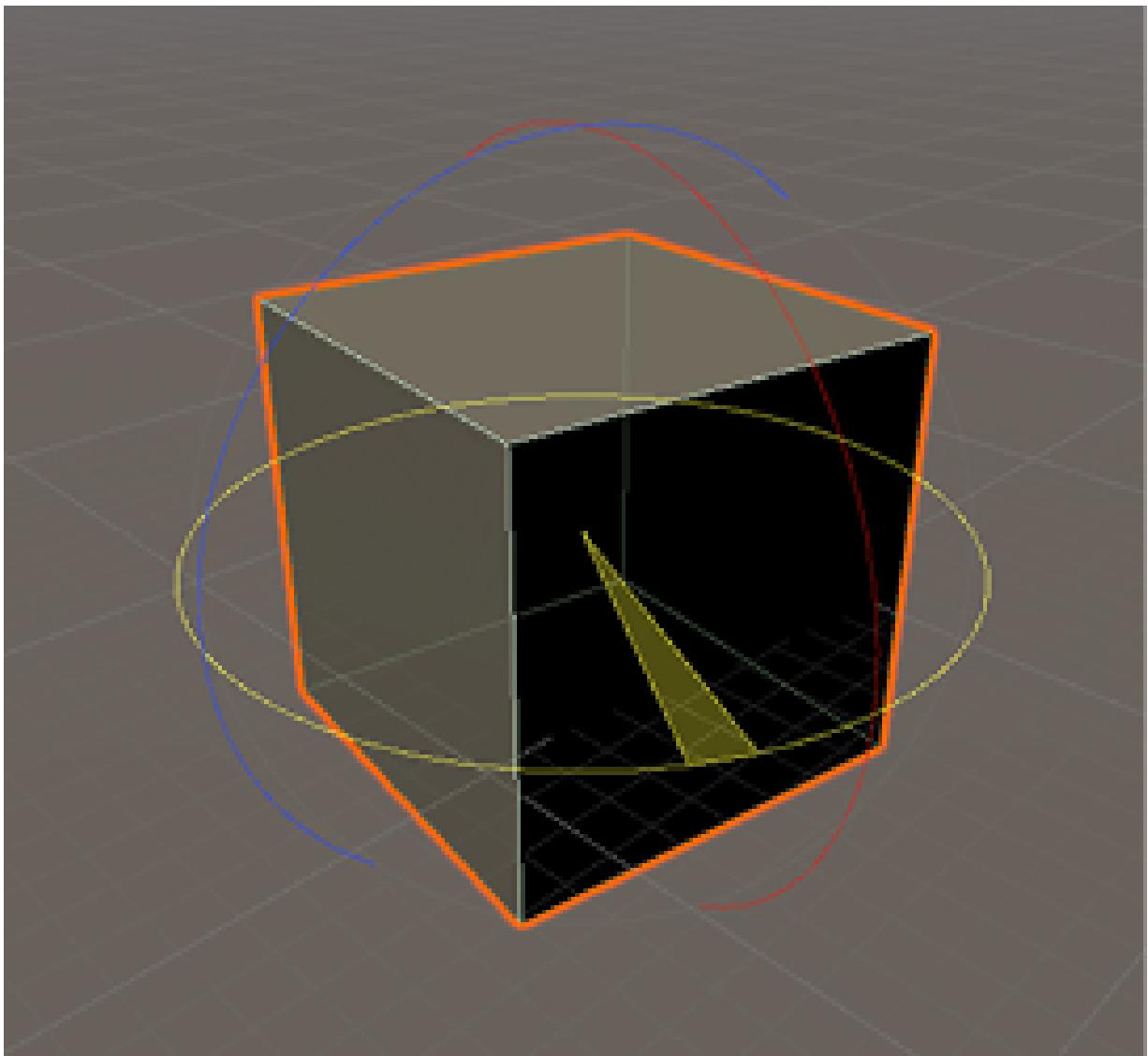


Figure 2.11: Rotating an object

Finally, we have scaling, and we have two ways to accomplish that, one of them being through the gray cube at the center of the **Transform** gizmo shown in Figure 2.8. This allows us to change the size of the object by clicking and dragging that cube. Now, as we want to prototype a simple level, sometimes we want to stretch the cube to create, for example, a column or a flat floor, and here's where the second way comes in. If you click and drag the colored cubes in front of the translation arrows instead of the gray one in the center, you will see how our cube stretches over those axes,

allowing you to change the shape of the object. If you don't see those cube-shaped arrows, remember to enable **Local** coordinates, as stated earlier in this section. The process of stretching is illustrated in the following screenshot:

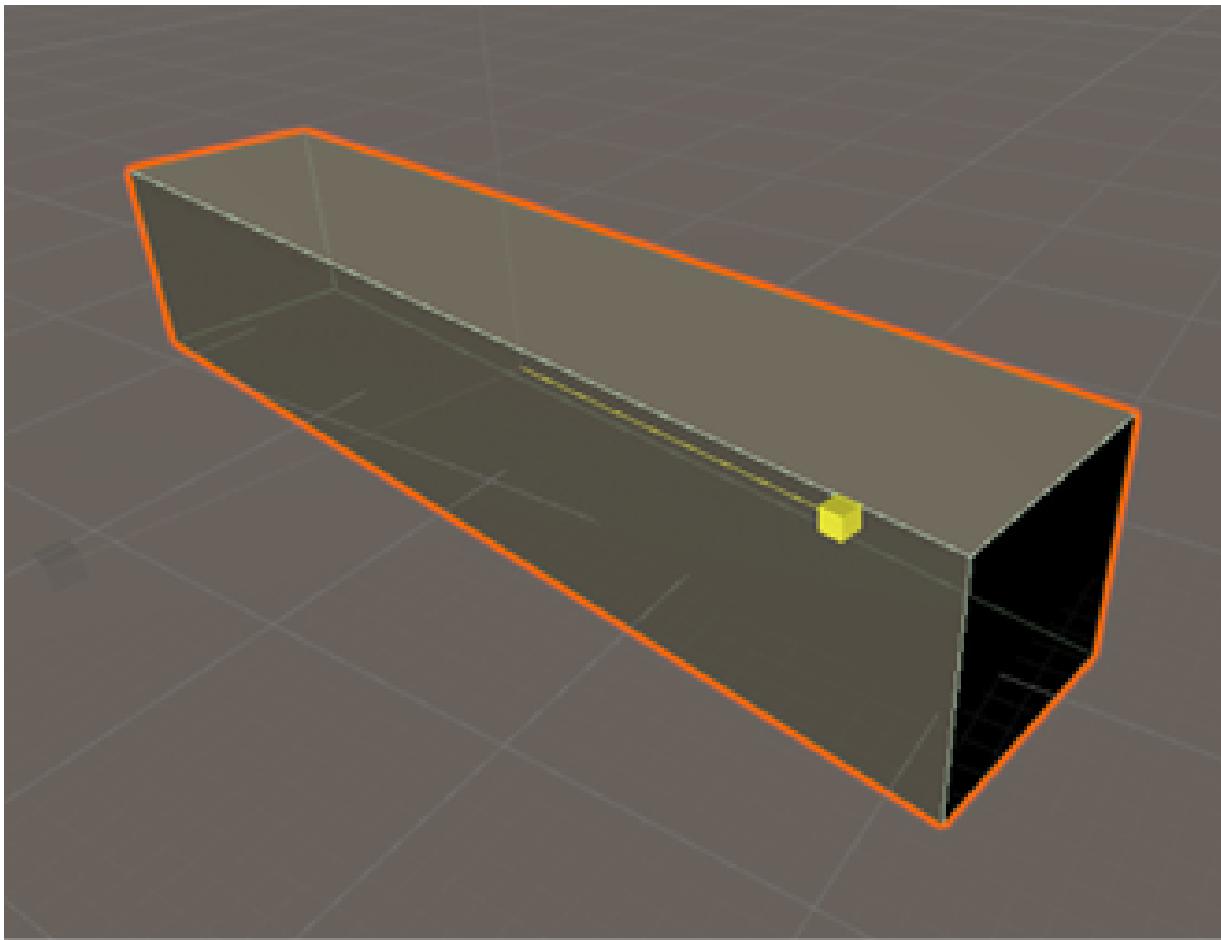


Figure 2.12: Scaling an object

Remember that you can also use the same gray cube we had in the **Transform** gizmo in the middle to scale all axes at the same time if desired. This is known as **uniform scaling**. Finally, something to consider here is that several objects can have the same scale values but different sizes, given how they were originally designed. **Scale** is a multiplier we can apply to the original size of an object, so a building and a car both with scale 1 make perfect sense; the relative size of one against the other seems correct. The

main takeaway here is that scale is not size but a way to multiply it. Consider that scaling objects is usually a bad practice in many cases. In the final versions of your scene, you will use models with the proper size and scale, and they will be designed in a modular way so that you can plug them into each other. If you scale them, several bad things can happen, such as textures being stretched and becoming pixelated, and modules that no longer plug properly. There are some exceptions to this rule, such as placing lots of instances of the same tree in a forest and changing its scale slightly to simulate variation. Also, in the case of gray-boxing, it is perfectly fine to take cubes and change the scale to create floors, walls, ceilings, columns, and so on because, ultimately, those cubes will be replaced with real 3D models. Here's a challenge! Create a room composed of a floor, three regular walls, and the fourth wall with a hole for a door (three cubes), and no need for a roof. In the next image, you can see how it should look:

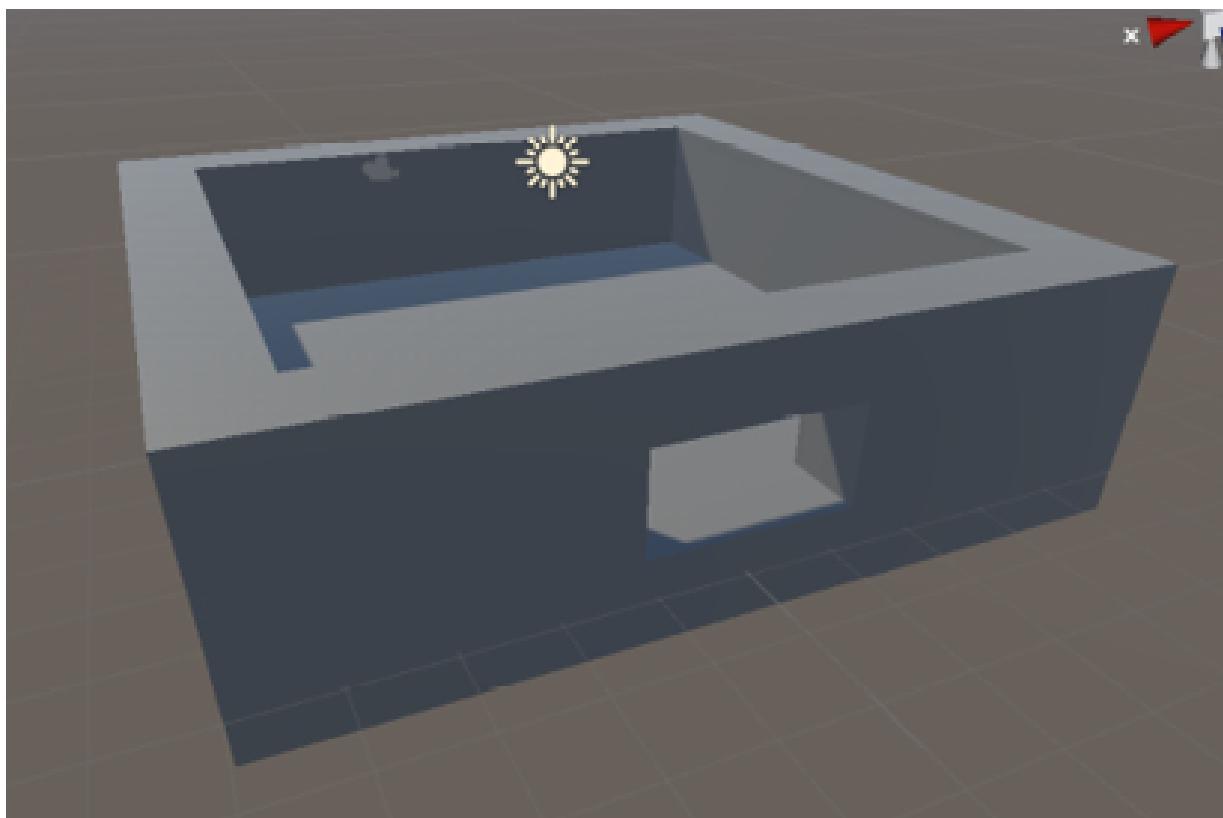


Figure 2.13: Room task finished

Memory

I remember when I started making games, I did most of my scenes with just boxes. While we will certainly do better in this book, there are still incredibly clever ways to use just that simple shape. Take, for example, the game *Thomas Was Alone*, which uses just boxes with clever lighting effects and looks beautiful.

Now that we can edit an object's location, let's see how we can edit all its other aspects.

GameObjects and components

We talked about our project being composed of `assets` (the project's files) and that a scene (which is a specific type of asset) is composed of GameObjects; so, how can we create an object? Through a composition of **components**. In this section, we will cover the following concepts related to components:

- Understanding components
- Manipulating components

Let's start by discussing what a component is.

Understanding components

A **component** is one of several pieces that make up a GameObject; each one is in charge of different features of the object. There are several components that Unity already includes that solve different tasks, such as playing a sound, rendering a mesh, applying physics, and so on; however, even though Unity has a large number of components, we will eventually need to create custom components sooner or later. In the next image, you can see what Unity shows us when we select a GameObject:

Inspector

Cube Static

Tag Untagged Layer Default

Transform

Position	X <input type="text" value="82.186"/>	Y <input type="text" value="53.663"/>	Z <input type="text" value="80.539"/>
Rotation	X <input type="text" value="0"/>	Y <input type="text" value="0"/>	Z <input type="text" value="0"/>
Scale	X <input type="text" value="1"/>	Y <input type="text" value="1"/>	Z <input type="text" value="1"/>

Cube (Mesh Filter)

Mesh

Mesh Renderer

Materials 1

Lighting

Cast Shadows	<input type="checkbox"/> On
Static Shadow Cas	<input type="checkbox"/>
Contribute Global	<input type="checkbox"/>
Receive Global Illu	<input type="button" value="Light Probes"/>

Probes

Light Probes	<input type="button" value="Blend Probes"/>
Anchor Override	<input type="button" value="None (Transform)"/>

Additional Settings

Motion Vectors	<input type="button" value="Per Object Motion"/>
Dynamic Occlusion	<input checked="" type="checkbox"/>
Rendering Layer	<input type="button" value="Light Layer default"/>

Box Collider

Edit Collider

Is Trigger

Provides Contacts

Material

Center X Y Z



Figure 2.14: The Inspector panel

If we needed to guess what the **Inspector** panel in the preceding screenshot does right now, we could say it shows all the properties of objects selected, either via the Hierarchy, the menu where you can see all the objects that have already been placed within the current scene, or the Scene view, and allows us to configure those options to change the behavior of the object (i.e., the position and rotation, whether it will project shadows or not, and so on). That is true, but we are missing a key element: those properties don't belong to the object; they belong to the components of the object. We can see some titles in bold before a group of properties, such as **Transform** and **Box Collider**, and so on. Those are the components of the object. In this case, our object has a **Transform**, a **Mesh Filter**, a **Mesh Renderer**, and a **Box Collider** component, so let's review each one of those. **Transform** just holds the position, rotation, and scale of the object, and by itself it does nothing—it's just a point in our game—but as we add components to the object, that position starts to have more meaning. That's because some components will interact with **Transform** and other components, each one affecting the other. An example of these different components interacting with each other would be the case of **Mesh Filter** and **Mesh Renderer**, both of those being in charge of rendering a 3D model. **Mesh Renderer** will render the 3D model, also known as mesh, specified by the **Mesh Filter** in the position specified in the **Transform** component, so **Mesh Renderer** needs to get data from those other components and can't work without them. Another example would be the **Box Collider**. This represents the physics shape of the object, so when the physics calculates collisions between objects, it checks if that shape collides with other shapes, based on the position specified in the **Transform** component. We will explore rendering and physics later in the book, but the takeaway from this section is that a **GameObject** is a collection of components, each component adding a specific behavior to our object, and each one interacting with the

others to accomplish the desired task. To further reinforce this, let's see how we can convert a cube into a sphere that will fall, due to gravity applied via physics.

Manipulating components

The tool to edit an object's components is the **Inspector**. It not only allows us to change the properties of our components but also lets us add and remove components. In this case, we want to convert a cube to a sphere, so we need to change several aspects of those components. We can start by changing the visual shape of the object, so we need to change the rendered model or **mesh**. The component that specifies the mesh to be rendered is the **Mesh Filter** component. If we look at it in the following figure, we can see a **Mesh** property that says **Cube**, with a little circle and a dot on its right:



Figure 2.15: The Mesh filter component

If you don't see a particular property, such as the **Mesh** we just mentioned, try to click the triangle at the left of the component's name. Doing this will expand and collapse all the component's properties. If we click the button with a circle and a dot inside, the one at the right of the **Mesh** property, the **Select Mesh** window will pop up, allowing us to pick several **mesh** options. In this case, select the **Sphere** mesh. In the future, we will add more 3D models to our project so that the window will have more options. The mesh selector is shown in the following screenshot:

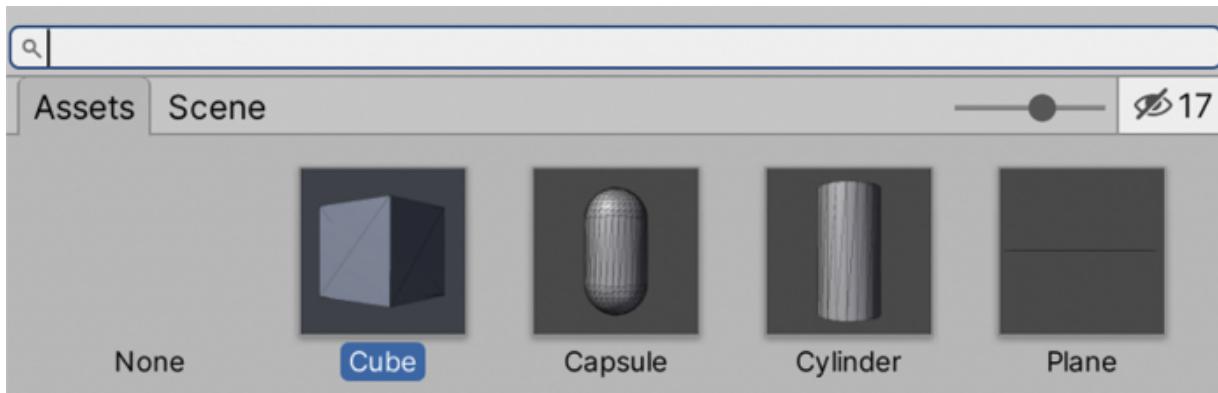


Figure 2.16: The mesh selector

Okay—the object now looks like a sphere, but will it behave like a sphere? Let's find out. In order to do so, we can add a component named **Rigidbody** to our sphere, which will add physics to it. We will talk more about Rigidbody and physics later in *Chapter 7, Collisions and Health: Detecting Collisions Accurately* but for now, let's stick to the basics. To add Rigidbody to our sphere, we need to click the **Add Component** button at the bottom of the Inspector. It will show a **Component Selector** window with lots of categories; in this case, we need to click on the **Physics** category. The window will show all the **Physics** components, and there we can find the **Rigidbody**. Another option would be to type `Rigidbody` in the search box at the top of the window. The following screenshot illustrates how to add a component:

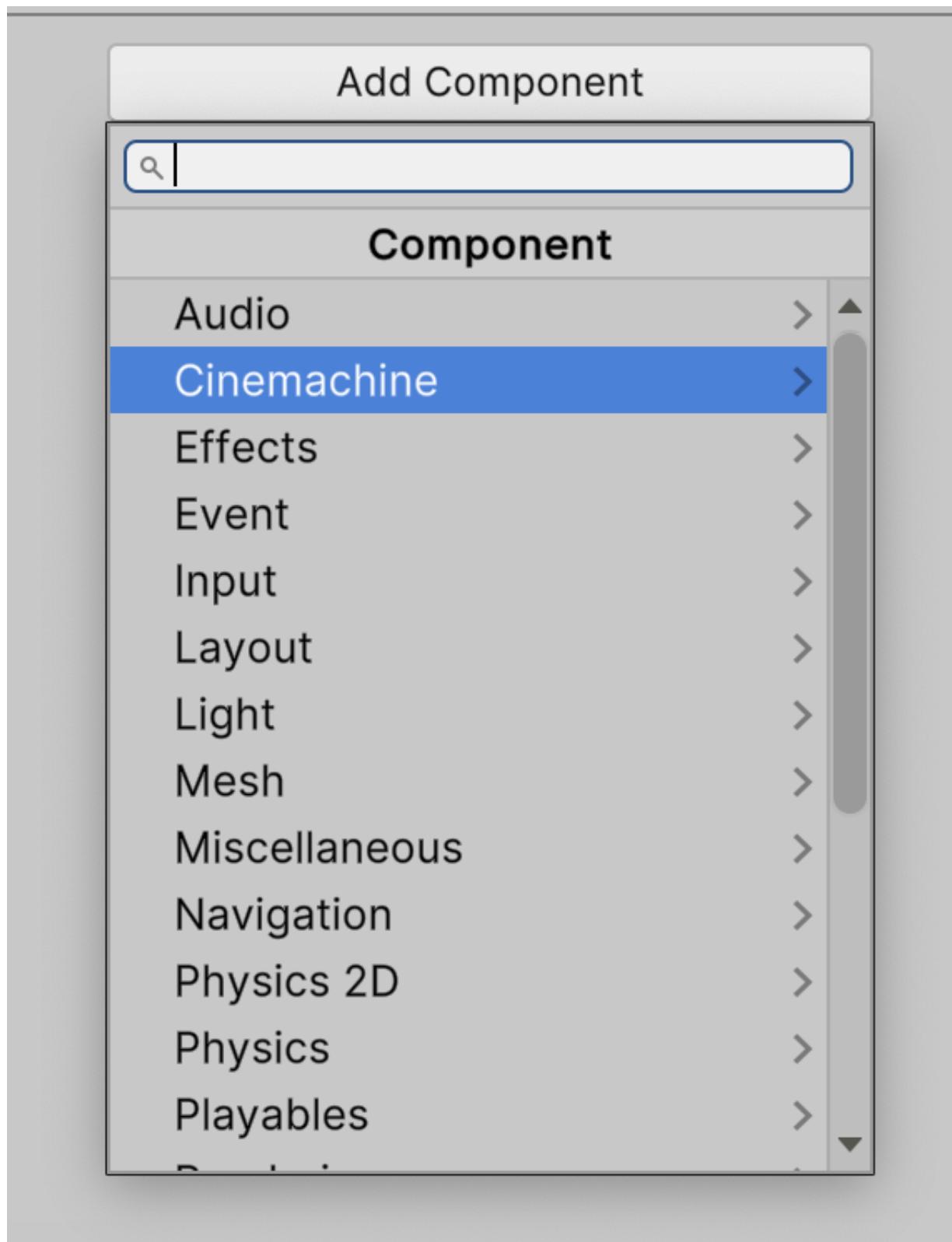


Figure 2.17: Adding components

If you hit the **Play** button in the top-middle part of the editor, you can test your sphere physics using the **Game** panel. That panel will be automatically focused when you hit **Play** and will show you how the player will see the game. The playback controls are shown in the following screenshot:

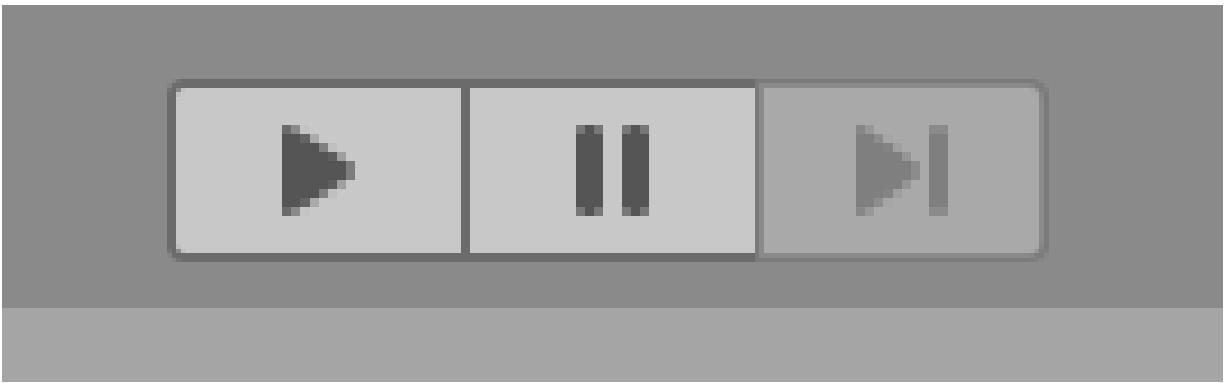


Figure 2.18: Playback controls

Here, you can just use the **Transform** gizmo to rotate and position your camera in such a way that it looks at our sphere. This is important, as one problem that can happen is that you might not see anything during **Play** mode, which can happen if the game camera does not point to where our sphere is located. While you are moving, you can check the little preview in the bottom-right part of the scene window to check out the new camera perspective. That is the expected behavior if you have selected a camera. Another alternative would be to select the camera in the **Hierarchy** and use the shortcut *Ctrl + Shift + F* (or *Command + Shift + F* on a Mac).

The camera preview is shown in the following screenshot:

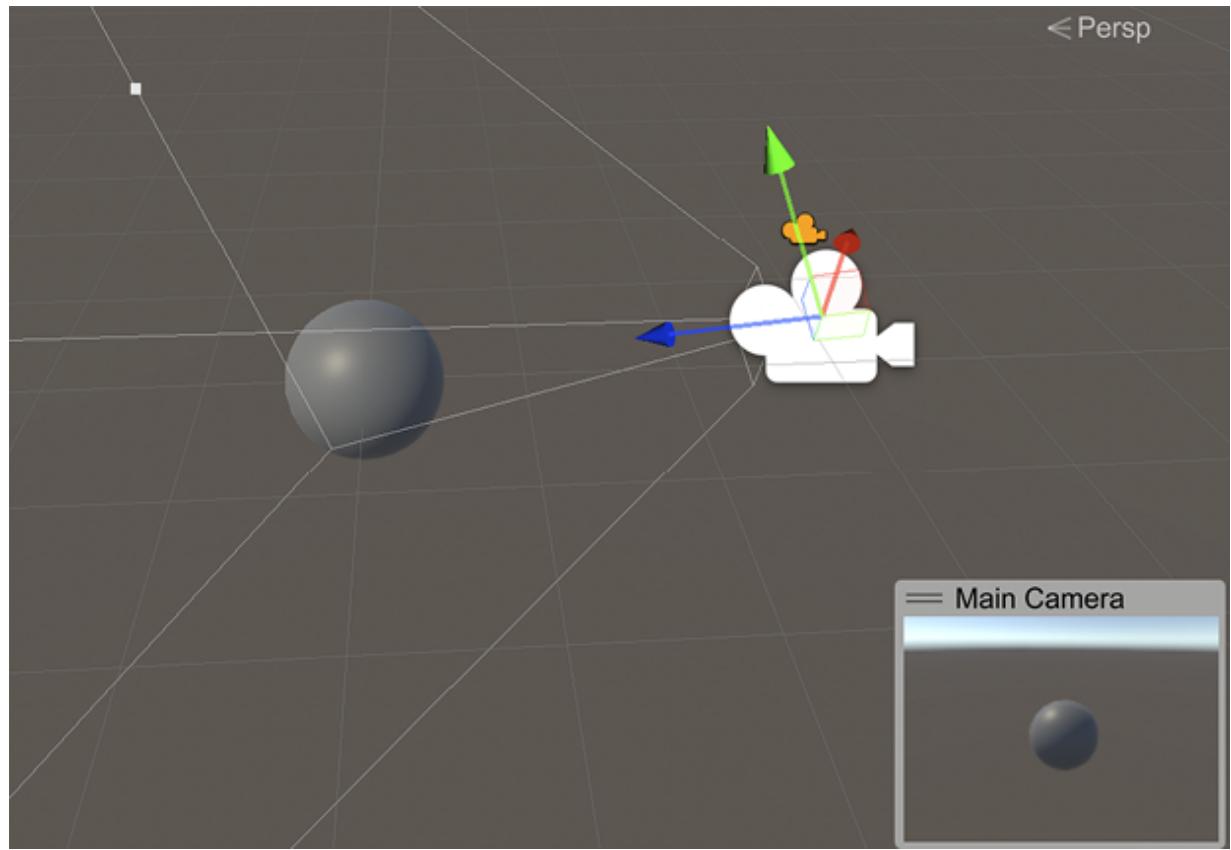


Figure 2.19: The camera preview

Now, to test if physics collisions are executing properly, let's create a cube, scale it until it has the shape of a ramp, and put that ramp below our sphere, as shown here:

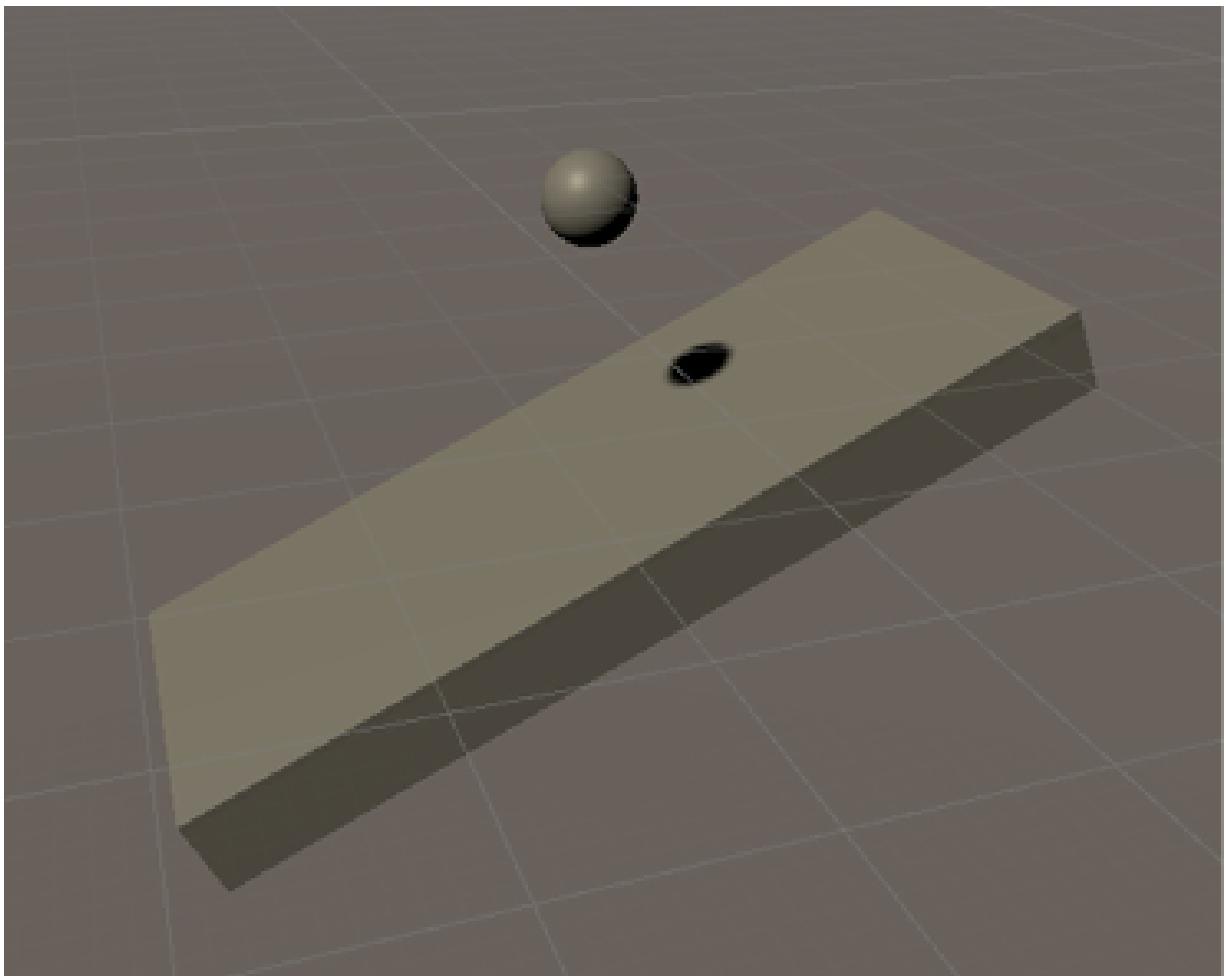


Figure 2.20: Ball and ramp objects

If you hit **Play** now, you will see the sphere colliding with our ramp, but in a strange way. It looks like it's bouncing, but that's not the case. If you expand the **Box Collider** component of our sphere, you will see that even if our object looks like a sphere, the green box gizmo shows us that our sphere is actually a box in the physics world, as illustrated in the following screenshot:

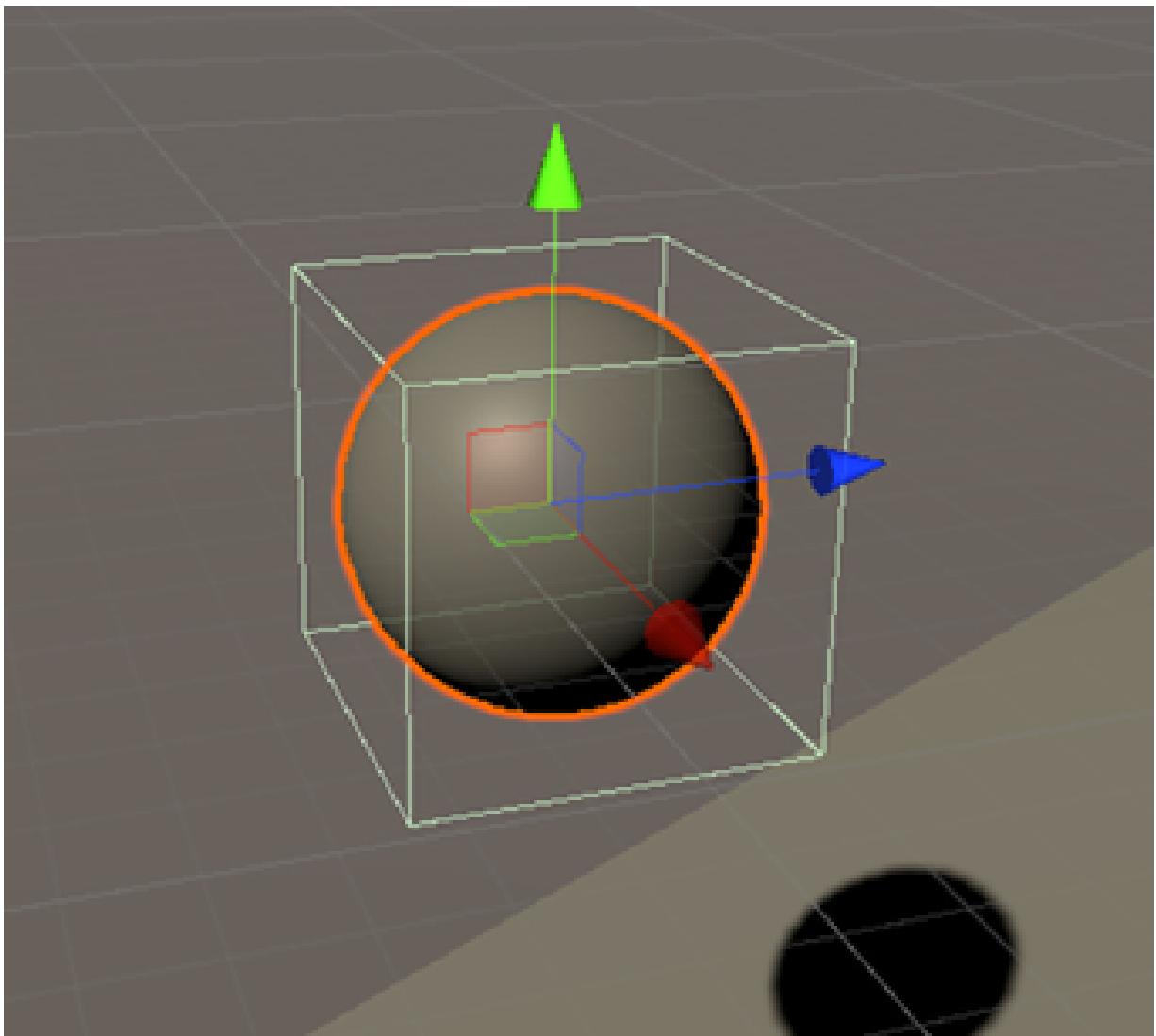


Figure 2.21: Object with a sphere graphic and box collider

Nowadays, video cards (GPUs) can handle rendering highly detailed models (models with a high polygon count), but the physics system is executed in the **central processing unit (CPU)**, and it needs to do complex calculations in order to detect collisions. To get a decent performance in our game, it needs to run at least **30 frames per second (FPS)**, the minimum accepted by the industry to provide a smooth experience. The physics system considers that and, hence, works using simplified collision shapes that may differ from the actual shape a player sees on the screen. That's why we have **Mesh Filter** and the different types of **Collider** components separated—

one handles the visual shape and the other the physics shape. Again, the idea of this section is not to deep-dive into those Unity systems, so let's just move on for now. How can we solve the issue of our sphere appearing as a box? Simple: by modifying our components! In this case, the **Box Collider** component already present in our cube GameObject can just represent a box physics shape, unlike **Mesh Filter**, which supports any rendering shape. So, first, we need to remove it by right-clicking the component's title and selecting the **Remove Component** option, as illustrated in the following screenshot:

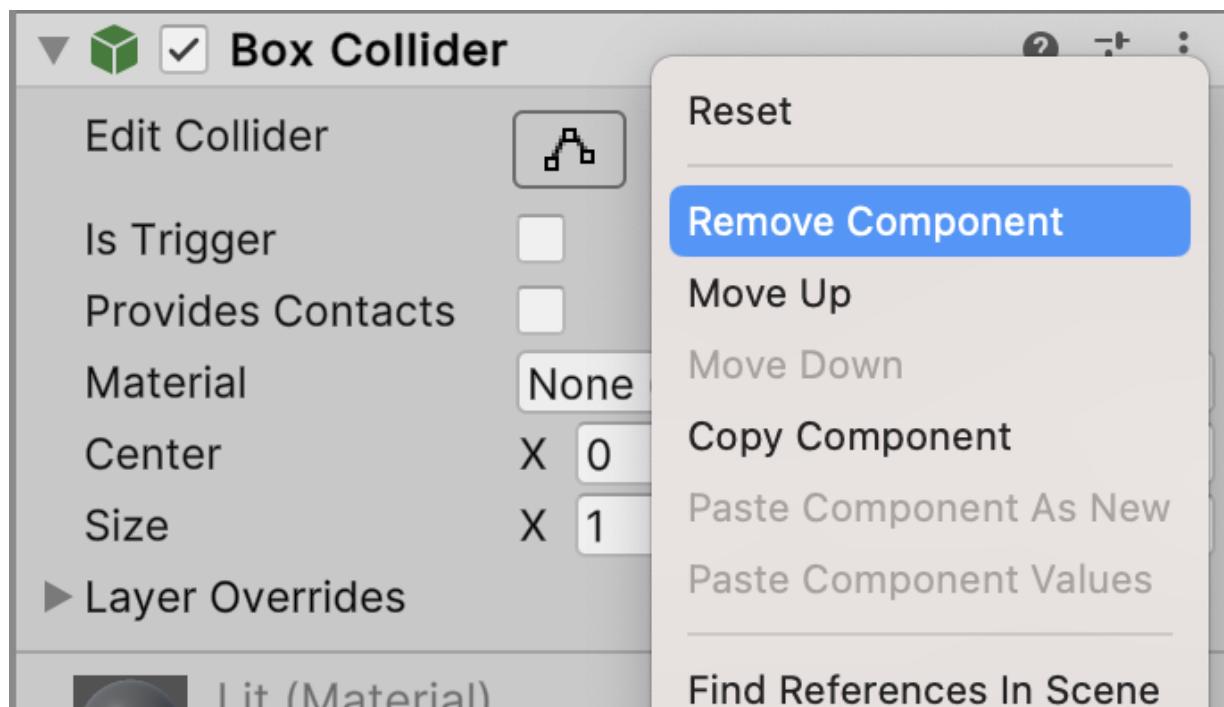


Figure 2.22: Removing components

Now, we can again use the **Add Component** menu to select a **Physics** component, this time selecting the **Sphere Collider** component. If you look at the **Physics** components, you will see other types of colliders that can be used to represent other shapes, but we will look at them later in Chapter 7. The **Sphere Collider** component can be seen in the following screenshot:

Add Component



Physics

- Character Controller
- Character Joint
- Cloth
- Configurable Joint
- Constant Force
- Fixed Joint
- Hinge Joint
- Mesh Collider
- Rigidbody
- Sphere Collider
- Spring Joint
- Terrain Collider
- Wheel Collider

Figure 2.23: Adding a Sphere Collider component

So, if you hit **Play** now, you will see that our sphere not only looks like a sphere but also behaves like one. Remember: the main idea of this section of the book is understanding that, in Unity, you can create whatever object you want just by adding, removing, and modifying components, and we will be doing a lot of this throughout the book. Now, components are not the only thing needed in order to create objects. Complex objects may be composed of several sub-objects, so let's see how that works.

Understanding object Hierarchies

Some complex objects may need to be separated into sub-objects, each one with its own components. Those sub-objects need to be somehow attached to the main object and work together to create the necessary object behavior. In this section, we will cover the following concepts related to objects:

- Parenting of objects
- Possible uses

Let's start by discovering how to create a parent-child relationship between objects.

Parenting of objects

Parenting consists of making an object the child of another, meaning that those objects will be related to each other. One type of relationship that happens is a **Transform relationship**, meaning that a child object will be affected by the parent's Transform. In simple terms, the child object will follow the parent, as if it is attached to it. For example, imagine a player with a hat on their head. The hat can be a child of the player's head, making the hat follow the head while they are attached. In order to try this, let's create a capsule that represents an enemy and a cube that

represents the weapon of the enemy. Remember that in order to do so, you can use the **GameObject** | **3D Object** | **Capsule** and **Cube** options and then use the **Transform** tool to modify them. An example of a capsule and a cube can be seen in the following screenshot:

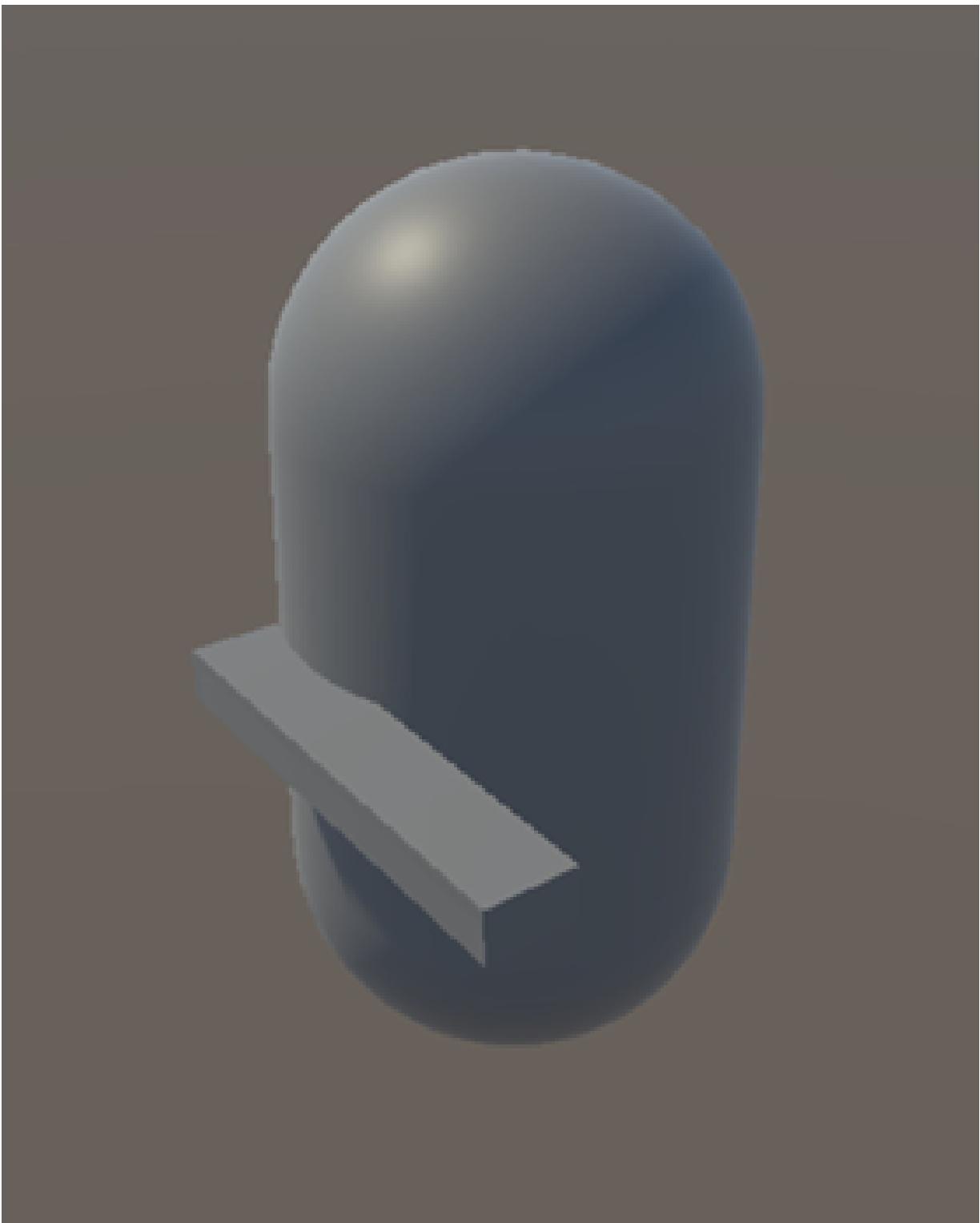


Figure 2.24: A capsule and a cube representing an enemy and a weapon

If you move the enemy object (the capsule), the weapon (the cube) will keep its position, not following our enemy. So, to prevent that, we can simply drag the weapon to the enemy object in the **Hierarchy** window, as illustrated in the following screenshot:

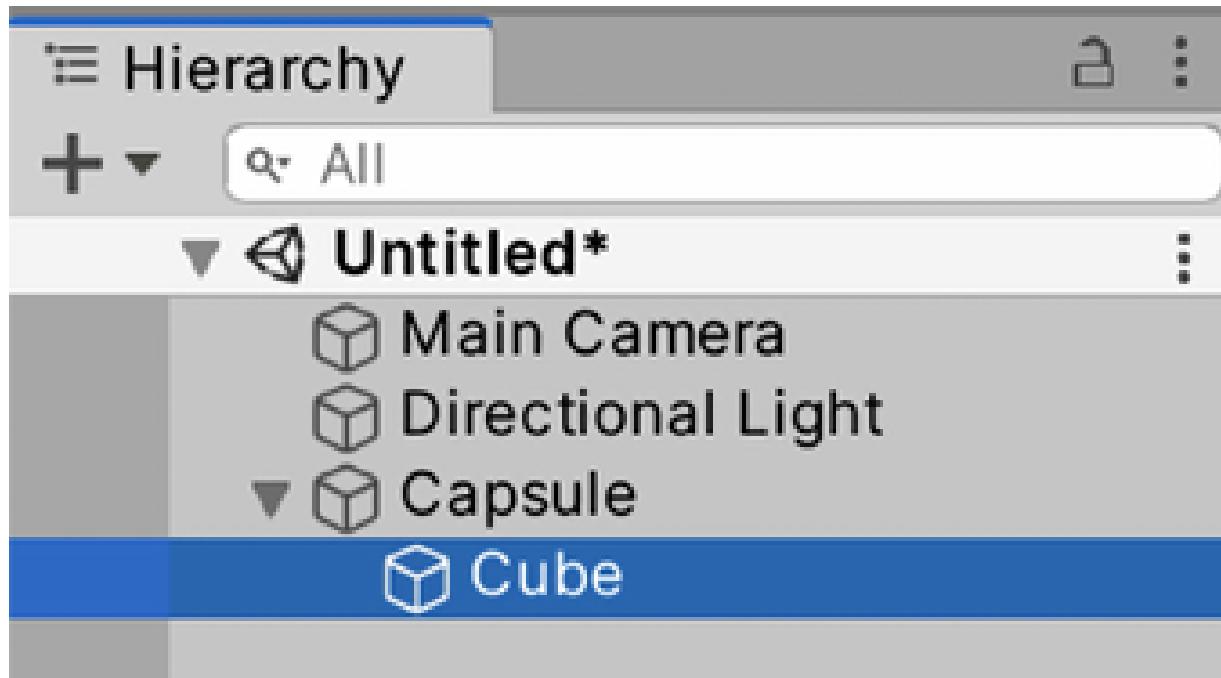


Figure 2.25: Parenting the cube weapon to the capsule character

Now, if you move the enemy, you will see the gun moving, rotating, and being scaled along with it. So, basically, the gun Transform also has the effects of the enemy Transform component. Now that we have done some basic parenting, let's explore other possible uses.

Possible uses

There are some other uses of parenting aside from creating complex objects. Another common usage for it is to organize the project Hierarchy. Right now, our scene is simple, but in time it will grow, so keeping track of all the objects will become difficult. To prevent this, we can create empty GameObjects (in **GameObject | Create Empty**) that only have the Transform component to act as containers, with objects put into them just to organize our scene.

Try to use this with caution because this has a performance cost if you abuse it. Generally, having one or two levels of parenting when organizing a scene is fine, but more than that can have a performance hit. Consider that you can—and will—have deeper parenting for the creation of complex objects; the proposed limit is just for scene organization. To keep improving on our previous example, duplicate the enemy a couple of times all around the scene, create an empty GameObject, name it `Enemies`, and drag all the enemies into it so that it will act as a container. This is illustrated in the following screenshot:

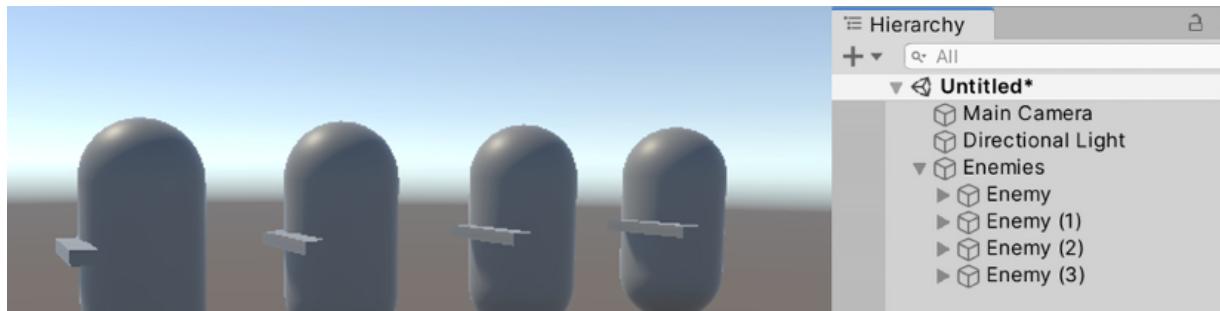


Figure 2.26: Grouping enemies in a parent object

Another common usage of parenting is to change the **pivot** (or center) of an object. Right now, if we try to rotate our gun with the **Transform** gizmo, it will rotate around its center because the creator of that cube decided to put the center there. Normally, that's okay, but let's consider a case where we need to make the weapon aim at the point where our enemy is looking. In this case, we need to rotate the weapon around the weapon handle; so, in the case of this cube weapon, it would be the part of the handle that is closest to the enemy. The problem here is that we cannot change the center of an object, so one solution would be to create another "weapon" 3D model or mesh with another center, which will lead to lots of duplicated versions of the weapon if we consider other possible gameplay requirements such as a rotating weapon pickup. We can fix this easily using parenting. The idea is to create an empty GameObject and place it where we want the new pivot of our object

to be. After that, we can simply drag our weapon inside this empty GameObject, and, from now on, consider the empty object as the actual weapon. If you rotate or scale this weapon container, you will see that the weapon mesh will apply those transformations around this container, so we can say the pivot of the weapon has changed (actually, it hasn't, but our container simulates the change). The process is illustrated in the following screenshot:

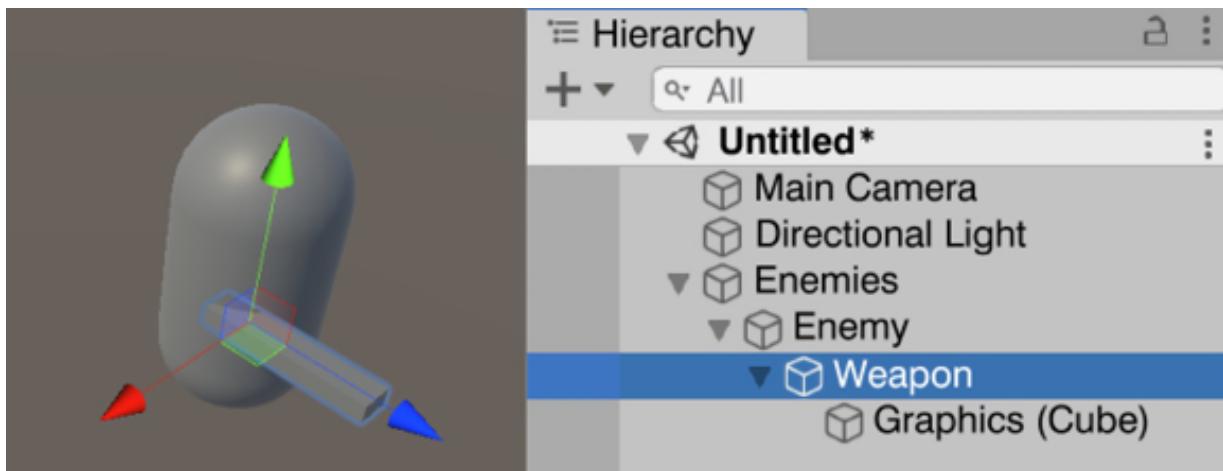


Figure 2.27: Changing the weapon pivot

Now, let's continue seeing different ways of managing GameObjects, using Prefabs this time.

Managing GameObjects using Prefabs

In the previous example, we created lots of copies of our enemy around the scene, but in doing so, we created a new problem. Let's imagine we need to change our enemy and add a **Rigidbody** component to it, but because we have several copies of the same object, we need to take them one by one and add the same component to all of them. Maybe later, we will need to change the mass of each enemy, so again, we need to go over each one of the enemies and make the change, and here we can start to see a pattern. One solution could be to select all the enemies using the Ctrl key (Command on a Mac) and modify all of them at once, but

that solution won't be of any use if we have enemy copies in other scenes. So, here is where Prefabs come in. In this section, we will cover the following concepts related to Prefabs:

- Creating Prefabs
- Prefab-instance relationship
- Prefab variants

Let's start by discussing how to create and use Prefabs.

Creating Prefabs

Prefabs are a Unity tool that allows us to convert custom-made objects, such as our enemy, into an asset that defines how they can be created. We can use them to create new copies of our custom object easily, without needing to create its components and sub-objects all over again. In order to create a Prefab, you can simply drag your custom object from the **Hierarchy** window to the project window, and after doing that, you will see a new asset in your project files. The project window is where you can navigate and explore all your project files; so, in this case, your Prefab is the first asset you ever created. Now, you can simply drag the Prefab from the project window into the scene to easily create new Prefab copies, as illustrated in the following screenshot:

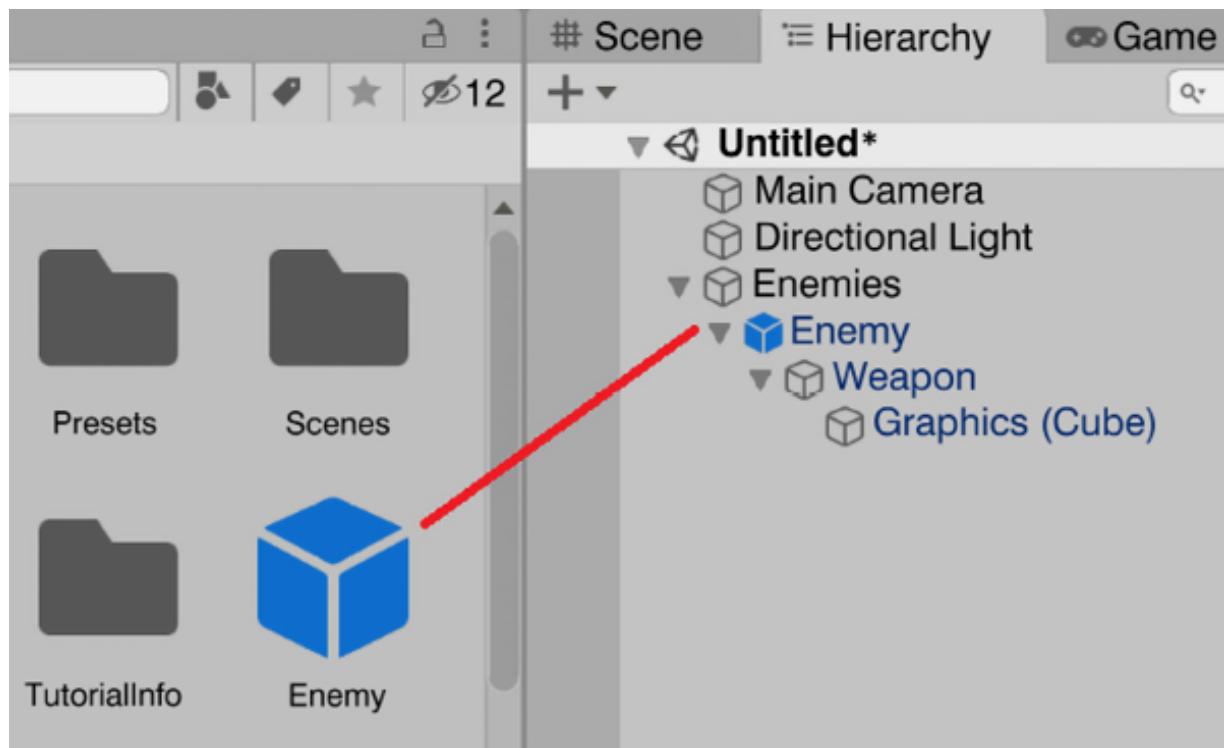


Figure 2.28: Creating a Prefab

Now, we have a little problem here. If you pay attention to the Hierarchy window, you will see the original Prefab objects and all the new copies with their names in the color blue, while the enemies created before the Prefab will have their names in black. The blue color in a name means that the object is an **instance** of a Prefab, meaning that the object was created based on a Prefab. We can select those blue-named objects and click the **Select** button in the **Inspector**, selecting the original Prefab that created that object. This is illustrated in the following screenshot:

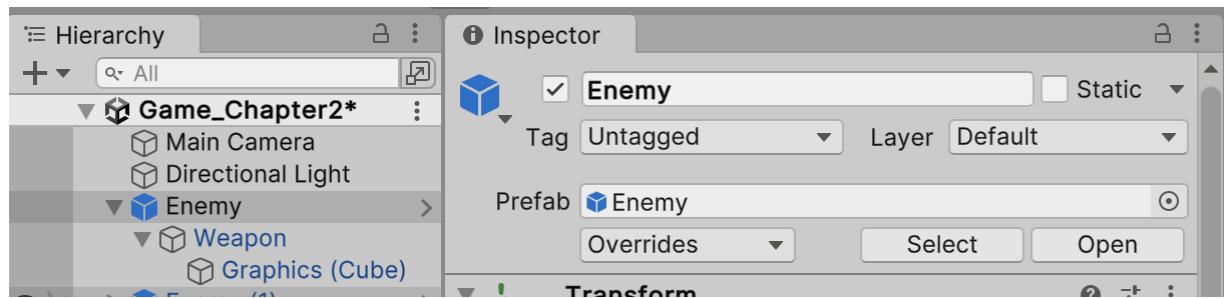


Figure 2.29: Detecting prefabs in the Hierarchy

The problem here is that the non-blue copies of the enemy are not instances of the Prefab we just created. We can fix this by selecting all enemies, by clicking them in the Hierarchy window while pressing the *Ctrl* key (*CMD* on a Mac), and then right-clicking on them to select the **Prefab | Reconnect Prefab** option. Finally, select the Enemy Prefab in the **Select GameObject** window that appeared to convert them to Prefab instances. This is a new feature available in the latest Unity versions.



Figure 2.30: Converting regular GameObjects to prefab instances

You can also replace the Prefab of Prefab instances and convert a Prefab instance back to a regular GameObject. For more details, see the video at <https://youtu.be/WOJzHz4sRyU> and the Unity documentation:

<https://docs.unity3d.com/Manual/UnpackingPrefabInstances.html>

Not having all enemy copies as Prefab instances didn't seem to be a problem, but we will see why it was an issue in the next section of this chapter, where we will explore the relationship between the Prefabs and their instances.

Prefab-instance relationship

An instance of a Prefab, the GameObject that was created when dragging the Prefab to the scene, has a binding to it that helps to revert and apply changes easily between the Prefab and the instance. If you take a Prefab and make some modifications to it,

those changes will be automatically applied to all instances across all the scenes in the project, so we can easily create a first version of the Prefab, use it all around the project, and then experiment with changes. To practice this, let's say we want to add a **Rigidbody** component to the enemies so that they can fall. In order to do so, we can simply double-click the Prefab file in the **Project** panel and enter the **Prefab Edit Mode**, where we can edit the Prefab isolated from the rest of the scene. Here, we can simply take the Prefab root object (**Enemy**, in our case) and add the **Rigidbody** component to it. After that, we can simply click on the **Scenes** button in the top-left part of the scene window to get back to the scene we were editing, and now, we can see that all the Prefab instances of the enemy have a **Rigidbody** component, as illustrated in the following screenshot:

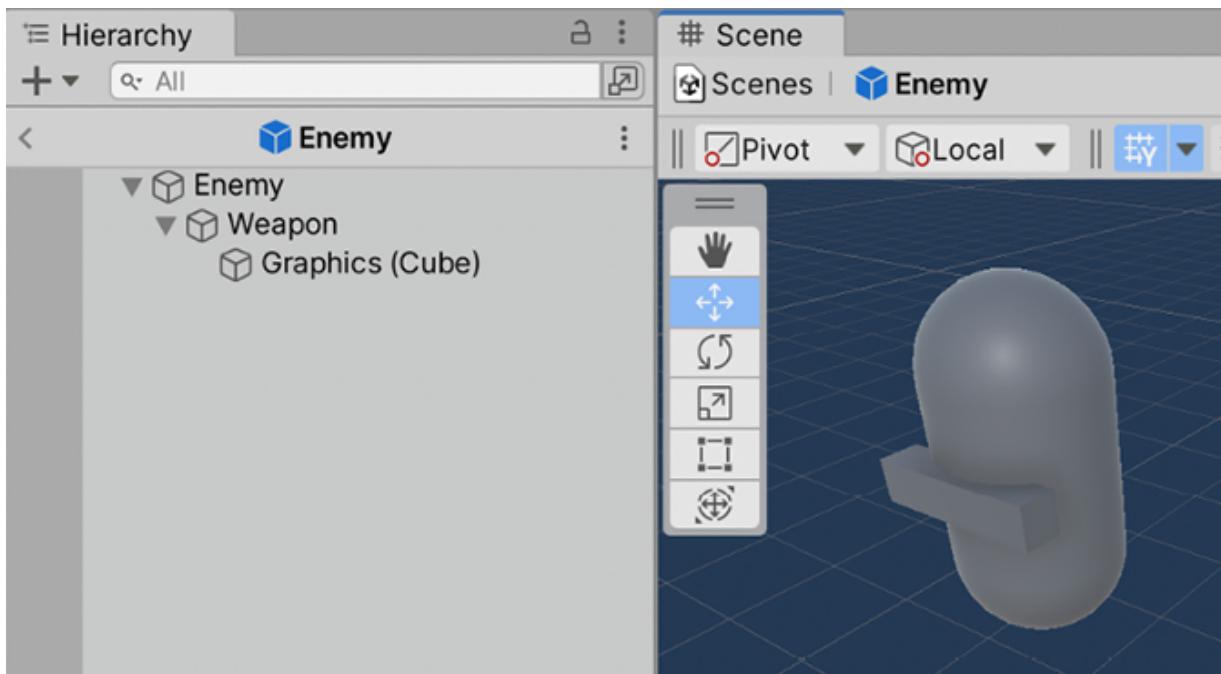


Figure 2.31: Prefab Edit Mode

Now, what happens if we change a Prefab instance (the one in the scene) instead? Let's say we want one specific enemy to fly, so they won't suffer the effect of gravity. We can do that by simply selecting the specific Prefab and unchecking the **Use Gravity** checkbox in

the **Rigidbody** component. After doing that, if we play the game, we will see that only that specific instance will float. That's because changes to an instance of a Prefab become an **override**, a set of differences the instance has compared to the original Prefab. We can see how the **Use Gravity** property is bold in the Inspector, and also has a blue bar displayed to its left, meaning it's an override of the original Prefab value. Let's take another object and change its **Scale** property to make it bigger. Again, we will see how the **Scale** property becomes bold, and a blue bar will appear to its left. The **Use Gravity** checkbox can be seen in the following screenshot:

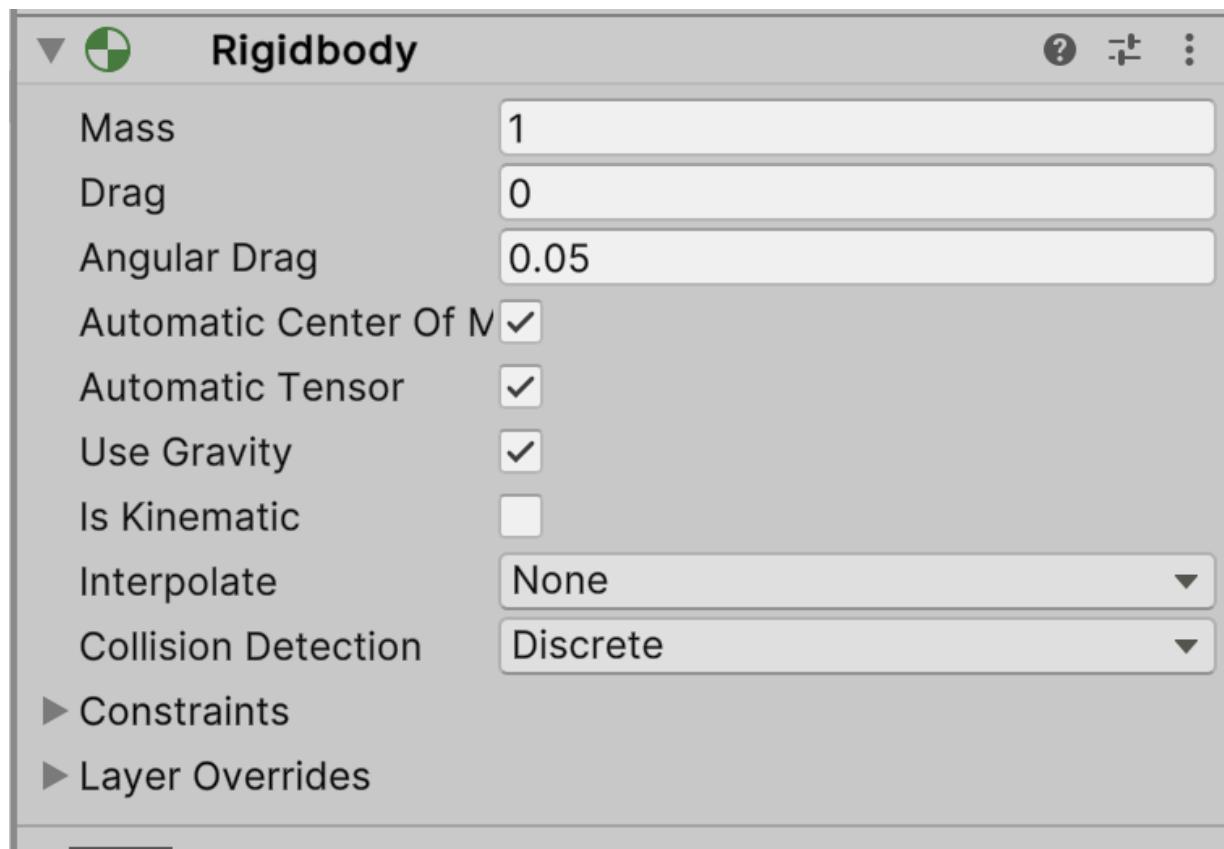


Figure 2.32: Use Gravity being highlighted as an override

The overrides have precedence over the Prefab, so if we change the scale of the original Prefab, the one that has a scale override won't change, keeping its own version of the scale, as illustrated in the following screenshot:

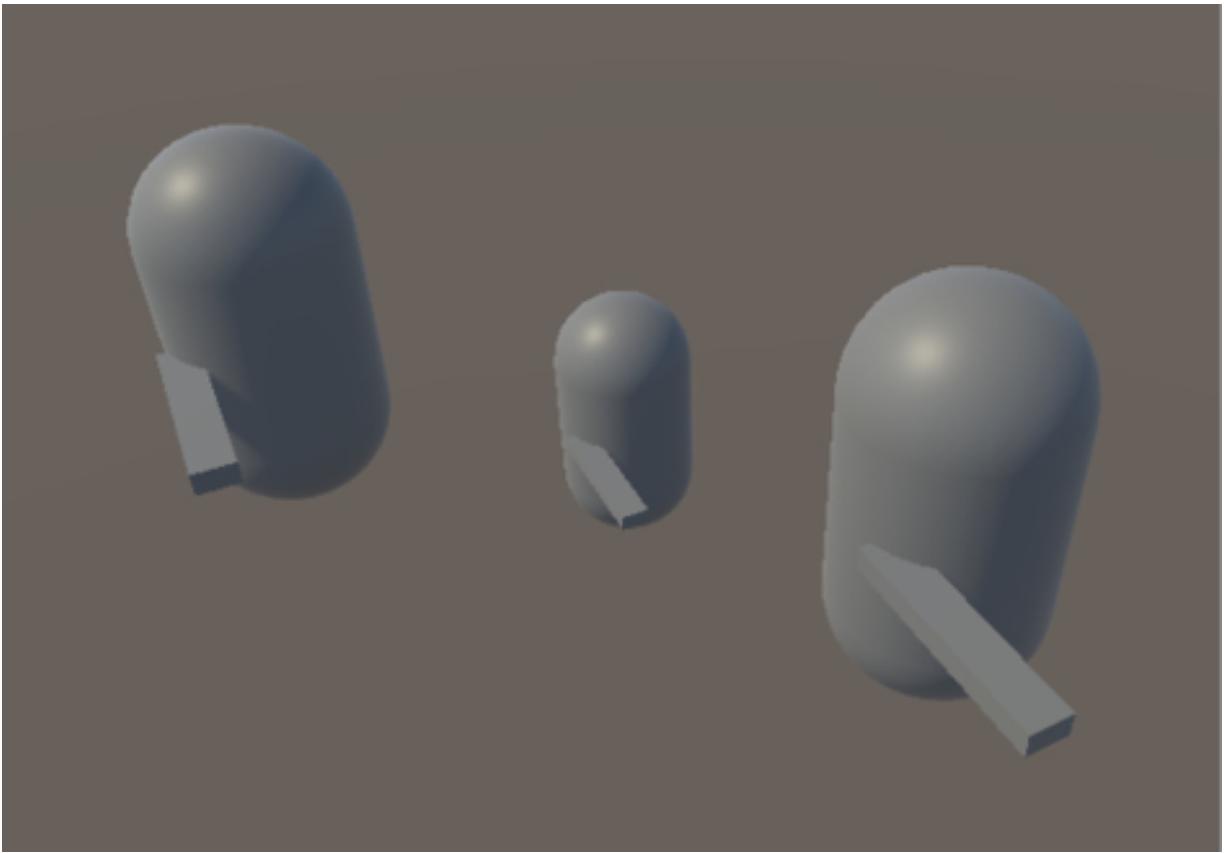


Figure 2.33: One Prefab instance with a scale override

We can easily locate all overrides of an instance using the **Overrides** dropdown in the Inspector after selecting the Prefab instance (the one in the scene, outside **Prefab Edit Mode**) in the Hierarchy, locating all the changes our object has. It not only allows us to see all the overrides but also reverts any override we don't want, applying the ones we do want. Let's say we regretted the lack of gravity of that specific Prefab—no problem! We can just locate the override and revert it using the **Revert all** button after clicking on the component with the override. The process is illustrated in the following screenshot:

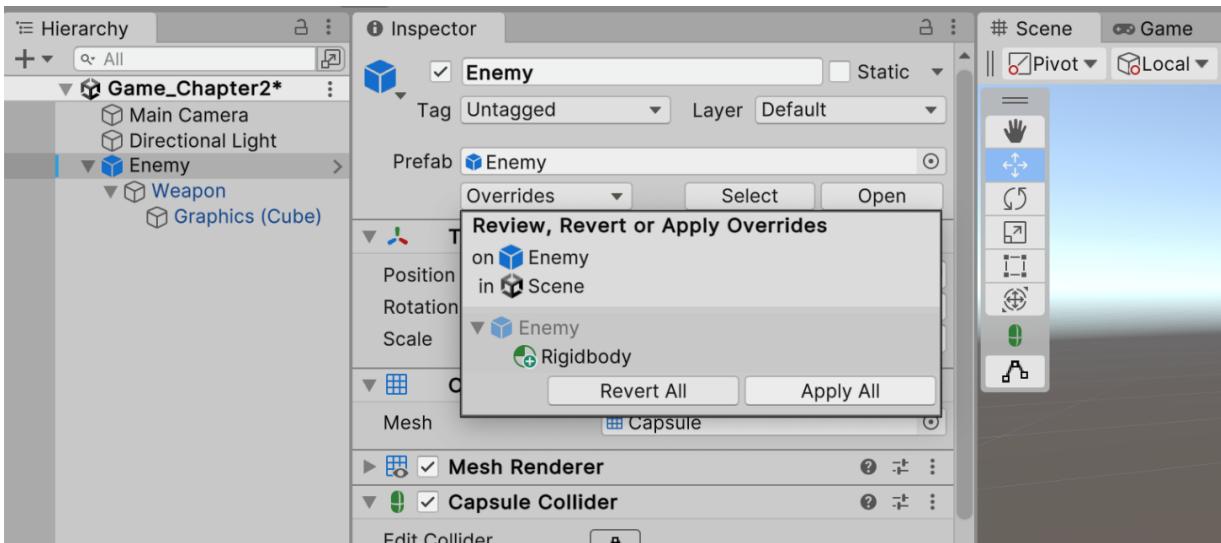


Figure 2.34: Reverting a single override

Also, let's imagine that we really liked the new scale of that instance, so we want all instances to have that scale—great! We can simply select the specific override, hit the **Apply** button, and then the **Apply All** option; now, all instances will have that scale (except the ones with an override), as illustrated in the following screenshot:

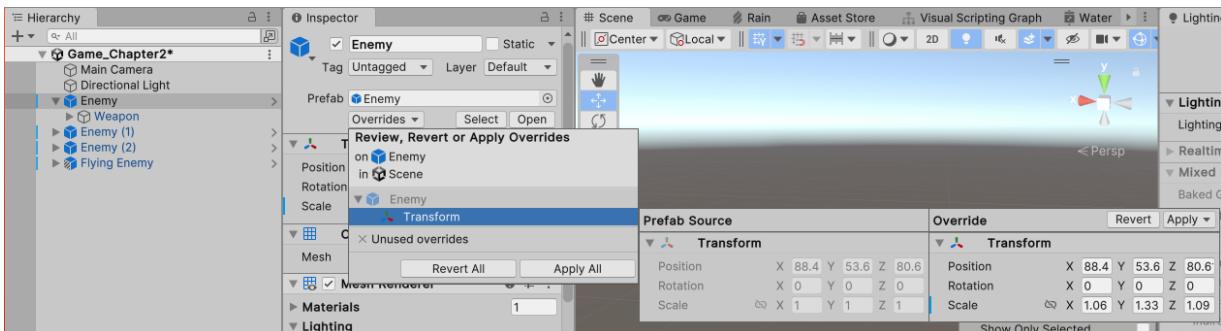


Figure 2.35: The **Apply** button

Also, we have the **Revert All** and **Apply All** buttons, but use them with caution, because you can easily revert and apply changes that you are not aware of. So, as you can see, Prefabs are a really useful Unity tool to keep track of all similar objects and apply changes to all of them, and they also have specific instances with few

variations. Talking about variations, there are other cases where you will want to have several instances of a Prefab with the same set of variations—for example, flying enemies and grounded enemies—but if you think about that, we will have the same problem we had when we didn't use Prefabs, so we need to manually update those varied versions one by one. Here, we have two options: one is to create a brand new Prefab just to have another version with that variation. This leads to the problem that if we want all types of enemies to undergo changes, we need to manually apply the changes to each possible Prefab. The second option is to create a Prefab variant. Let's review the latter.

Prefab variants

A **Prefab Variant** is a new Prefab that is created based on an existing one, so the new one **inherits** the features of the base Prefab. This means that our new Prefab can have differences from the base one, but the features that they have in common are still connected. To illustrate this, let's create a variation of the enemy Prefab that can fly: the flying enemy Prefab. In order to do that, we can select an existing enemy Prefab instance in the Hierarchy window, name it `Flying Enemy`, and drag it again to the project window, and this time we will see a prompt, asking which kind of Prefab we want to create. This time, we need to choose **Prefab Variant**, as illustrated in the following screenshot:

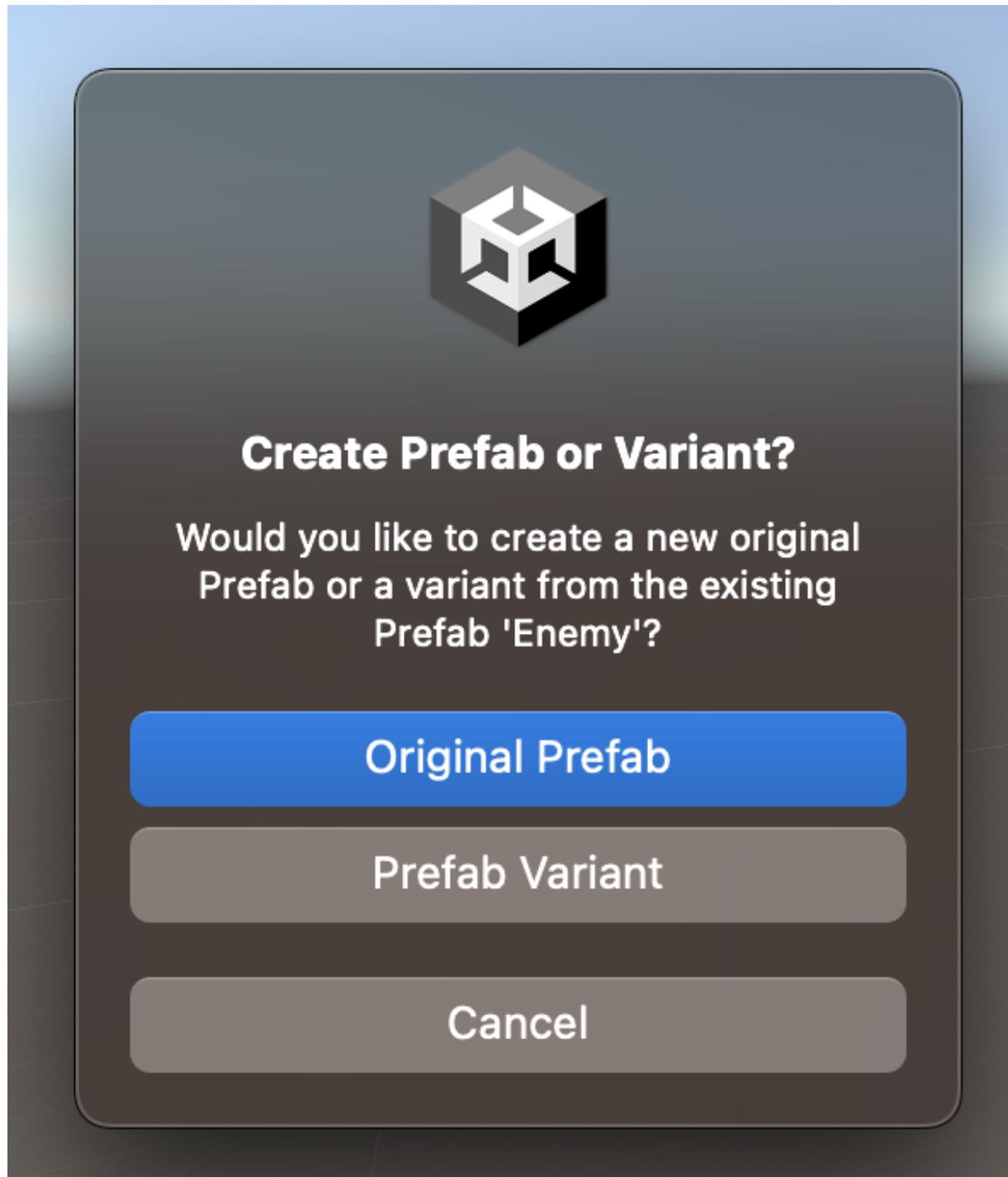


Figure 2.36: Creating Prefab Variants

Now, we can enter the **Prefab Edit Mode** of the variant by double-clicking the new Prefab file created in the project panel. Then, add a cube as the jetpack of our enemy, and also uncheck the **Use**

Gravity property for the enemy. If we return to the scene, we will see that the variant instance has changed, and the base enemies haven't changed. You can see this in the following screenshot:

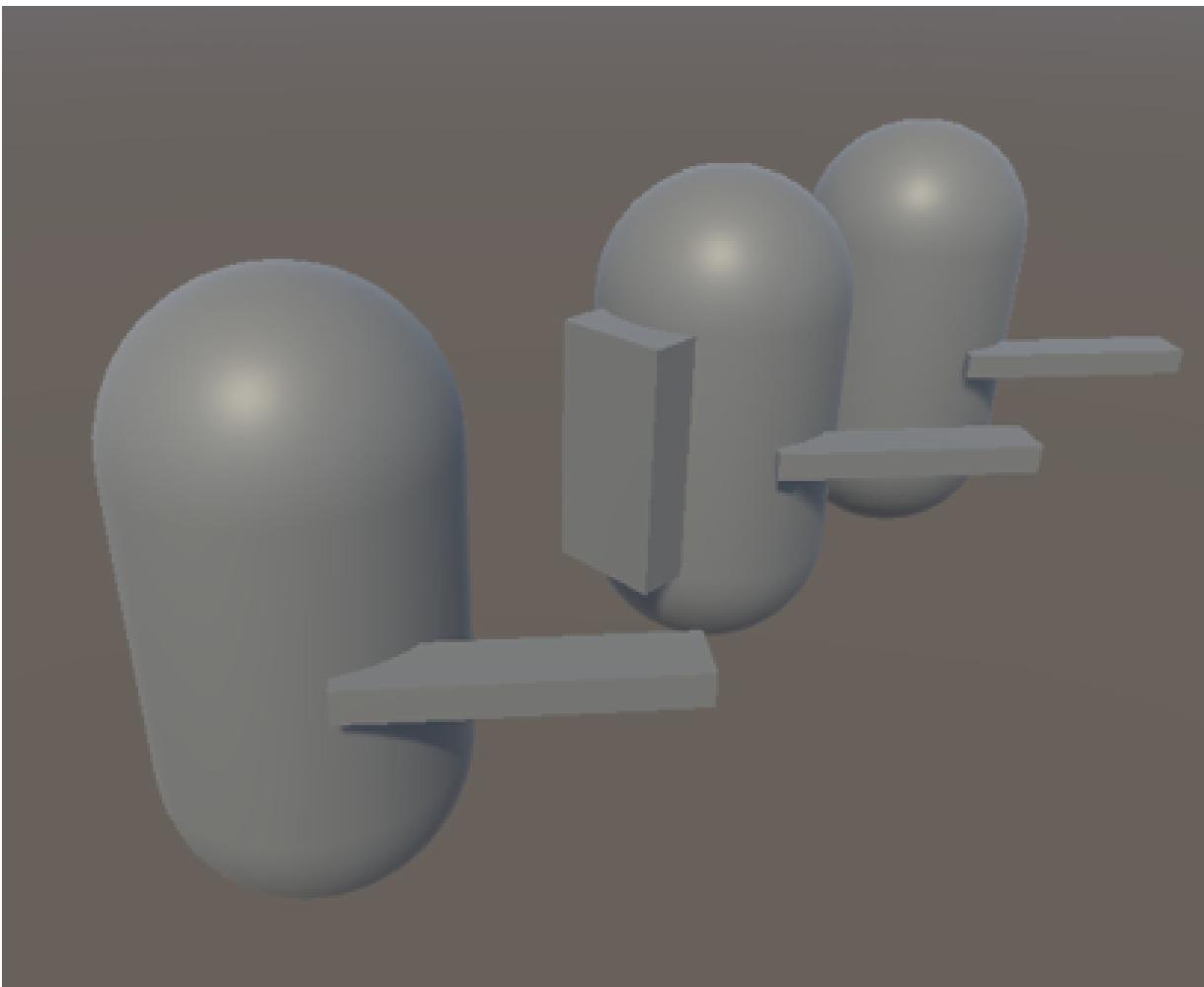


Figure 2.37 A Prefab variant instance

Now, imagine you want to add a hat to all our types of enemies. We can simply enter the **Prefab Edit Mode** of the base enemy Prefab by double-clicking it and adding a cube as a hat. Now, we will see that change applied to all the enemies, because remember: the **Flying Enemy** Prefab is a variant of the base enemy Prefab, meaning that it will inherit all the changes of that one.

There's also the concept of Nested Prefabs, which allows you to use prefabs inside prefabs to cleverly reuse prefabs' pieces. For more info, see the Unity documentation here:
<https://docs.unity3d.com/Manual/NestedPrefabs.html>

We have created lots of content so far, but if our PC turns off for some reason, we will certainly lose it all, so let's see how we can save our progress.

Saving scenes and projects

As in any other program, we need to save our progress. The difference here is that we don't have just one giant file with all the project assets but also several files for each asset. Let's start saving our progress by saving the scene, which is pretty straightforward. We can simply go to **File | Save** or press **Ctrl + S** (**Command + S** on a Mac). The first time we save our scene, a window will ask us where we want to save our file, and you can save it wherever you want inside the `Assets` folder of our project, but never outside that folder; otherwise, Unity will not be capable of finding it as an asset in the project. That will generate a new asset in the project window: a scene file. In the following screenshot, you can see how I saved the scene, naming it `test`, and now it shows up in the **Project** panel:

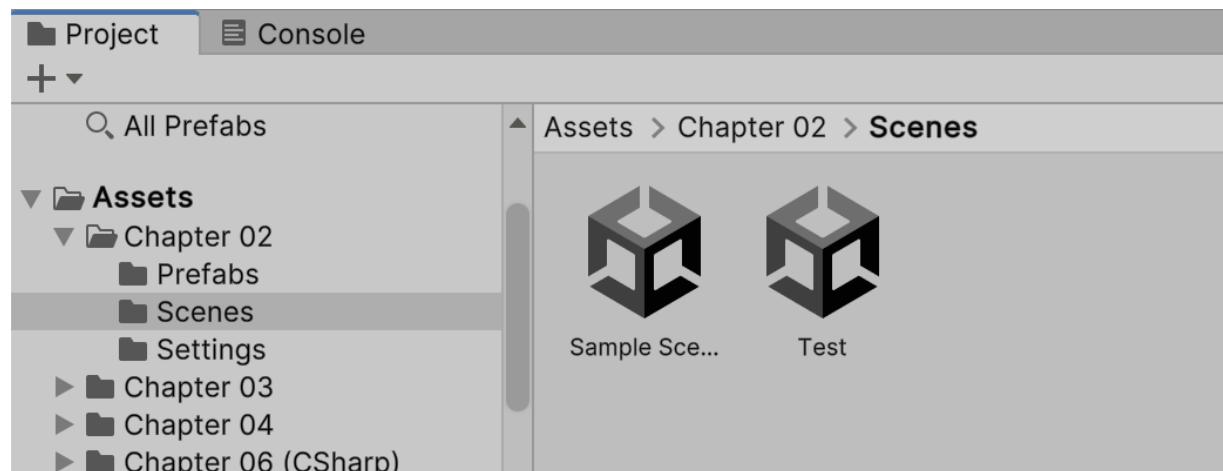


Figure 2.38: Scene files

We can create a folder to save our scene in the save dialog, or, if you already saved the scene, you can create a folder using the **plus (+)** icon in the project window, and then click the **Folder** option.

Finally, drag the created scene to that folder. Now, if you create another scene with the **File | New Scene** menu option, you can get back to the previous scene just by double-clicking the scene asset in the project window. Try it! This only saved the scene, but any change in Prefabs and other kinds of assets are not saved with that option. Instead, if you want to save every change of the assets except scenes, you can use the **File | Save Project** option. It can be a little confusing, but if you want to save all your changes, you need to both save the scenes and the project, as saving just the project won't save the changes to the scenes. Sometimes, the best way to be sure everything is saved is just by closing Unity, which is recommended when you try to move your project between computers or folders. This will show you a prompt to save the changes in the scene, and it will automatically save any change made to other assets, like Prefabs.

Summary

In this chapter, we had a quick introduction to essential Unity concepts. We reviewed the basic Unity windows and how we can use all of them to edit a full scene, from navigating it and creating premade objects (Prefabs) to manipulating them to create our own types of objects, using GameObjects and components. We also discussed how to use the Hierarchy window to parent GameObjects to create complex object Hierarchies, as well as how to create Prefabs to reutilize and manipulate large amounts of the same type of objects. Finally, we discussed how we can save our progress. In the next chapter, we will learn different tools like the Terrain system and ProBuilder to create the first prototype of our game's

level. This prototype will serve as a preview of where our scene will head, testing some ideas before going into full production.

3 From Blueprint to Reality: Building with Terrain and ProBuilder

Join our book community on Discord

<https://packt.link/unitydev>



Now that we've grasped all the necessary concepts to use Unity, let's start designing our first level of the game. The idea in this chapter is to learn how to use Terrain Tools to create the landscape of our game and then use ProBuilder to create the 3D mesh of the base with greater detail than using cubes. By the end of the chapter, you will be able to create a prototype of any kind of scene and try out your idea before actually implementing it with the final graphics. Specifically, we will examine the following concepts in this chapter:

- Defining our game concept
- Creating a landscape with Terrain Tools
- Creating shapes with ProBuilder

Let's start by talking about our game concept, which will help us draft the first-level environment.

Defining our game concept

Before even adding the first cube to our scene, it is good to have an idea of what we are going to create, as we will need to understand

the basic concept of our game to start designing the first level. Throughout this book, we will be creating a shooter game, in which the player will be fighting against waves of enemies trying to destroy the player's base. This base will be a complex in a (not so) secret location bordered by mountains:



Figure 3.1: Our finished game

We will be defining the mechanics of our game as we progress through the book, but with this basic high-level concept of the game, we can start thinking about how to create a mountainous landscape and a placeholder player's base. With that in mind, in the next section of this chapter, we will learn how to use Unity's Terrain Tools to create our scene's landscape.

Creating a landscape with Terrain

So far, we have used cubes to generate our level prototype, but we also learned that cubes sometimes cannot represent all possible objects we could need. Imagine something irregular, such as a full

terrain with hills, canyons, and rivers. This would be a nightmare to create using cubes, given the irregular shapes you find in the terrain. Another option would be to use 3D modeling software, but the problem with that is that the generated model will be so big and so detailed that it won't perform well, even on high-end PCs. In this scenario, we need to learn how to use Unity's Terrain system, which we will do in this first section of the chapter. In this section, we will cover the following concepts related to terrains:

- Discussing Height Maps
- Creating and configuring Height Maps
- Authoring Height Maps
- Adding Height Map details

Let's start by talking about Height Maps, whose textures help us define the heights of our terrain.

A `texture` is an image that is applied to different parts of 3D models to give them details. The concept is analogous to the sticker sheet in the toys of Kinder eggs, where you paste them on different parts of the toys to give them eyes or smiles. We will talk more about textures in *Chapter 4, Seamless Integration: Importing and Integrating Assets*.

Discussing Height Maps

If we create a giant area of the game with hills, canyons, craters, valleys, and rivers using regular 3D modeling tools, we will have a problem in that we will use fully detailed models for objects at all possible distances, thus wasting resources on rendering details that we won't see when the object is far away. In several instances in a game, players will need to view various parts of the terrain from considerable distances, making efficient resource management a serious issue. Unity Terrain Tools uses a technique called Height Maps to generate the terrain in a performant and dynamic way. Instead of generating large 3D models for the whole terrain, it uses

an image called a **Height Map**, which looks like a top-down black-and-white photo of the terrain. In the following image, you can see a black-and-white top-down view of a region of Scotland, with white being higher and black being lower:

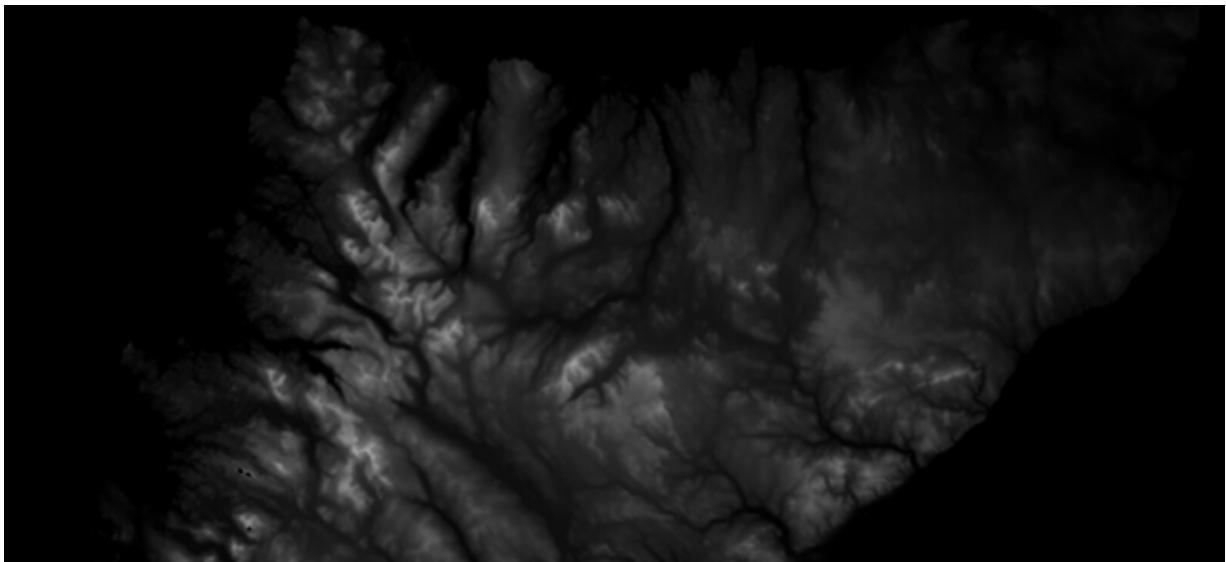


Figure 3.2: Scotland's Height Map

In the preceding image, you can easily spot the peaks of the mountains by looking for the whitest areas of the image. Everything below sea level becomes black, while anything in the middle uses gradients of gray, representing different heights between the minimum and maximum heights. The idea is that each pixel of the image determines the height of that specific area of the terrain. Unity Terrain Tools can automatically generate a 3D mesh from that image, saving us the hard drive space of having full 3D models of that terrain. Also, Unity will create the terrain as we move, generating high-detail models for nearby areas and low-detail models for faraway areas, making it a performant solution. In the following image, you can see the mesh that was generated for the terrain. You can appreciate that the nearer parts of the terrain have more polygons than the parts further away:

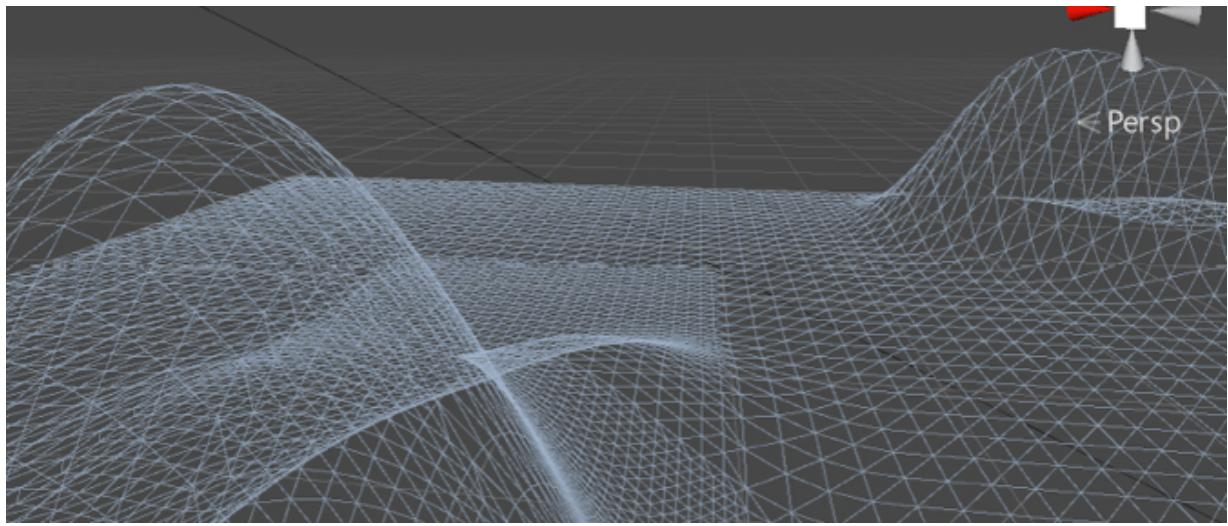


Figure 3.3: Height Map generated mesh

Take into account that this technology has its cons, such as the time it takes for Unity to generate those 3D models while we play and the inability to create caves. For now, however, that's not a problem for us. Now that we know what a Height Map is, let's see how we can use Unity Terrain Tools to create our own Height Maps.

Creating and configuring Height Maps

If you click on **GameObject | 3D Object | Terrain**, you will see a giant plane appear on your scene, and a **Terrain** object appears in your Hierarchy window. That's our terrain, and it is plain because its Height Map starts all black, so no height whatsoever is in its initial state. In the following image, you can see what a brand-new **Terrain** object looks like:

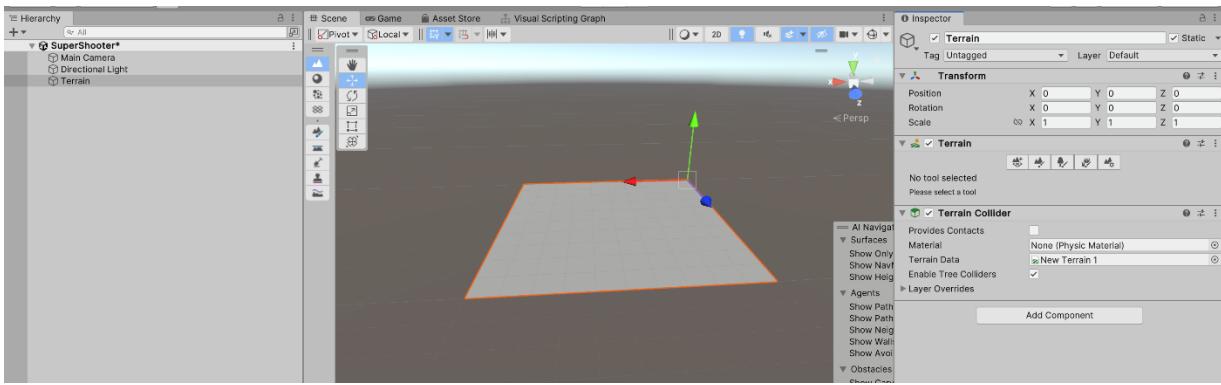


Figure 3.4: Terrain with no heights painted yet

Before you start editing this **Terrain**, you must configure different settings such as the size and resolution of the Terrain's Height Map, and that depends on what you are going to do with it. This is not the same as generating a whole world. Our game will feature the player's base, which they will defend, so the terrain will be small. In this case, an area that's 200 x 200 meters in size surrounded by mountains will be enough. In order to configure our terrain for those requirements, we need to do the following:

1. Select **Terrain** from the **Hierarchy** or **Scene** window.
2. Look at the **Inspector** for the **Terrain** component and expand it if it is collapsed.
3. Click on the mountain and gear icon (the furthest right option) to switch to configuration mode. In the following screenshot, you can see where that button is located:

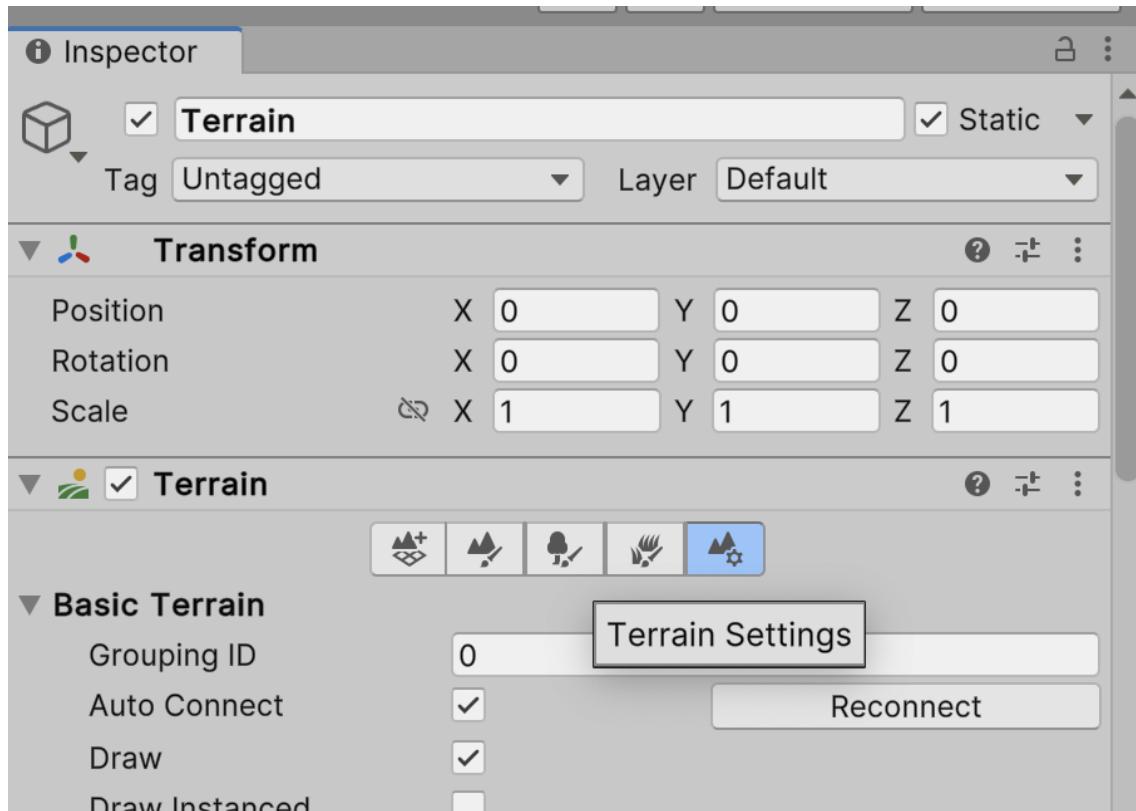


Figure 3.5: Terrain Settings button

4. Look for the **Mesh Resolution** (On Terrain Data) section.
5. Change the **Terrain Width** and **Terrain Length** to `200` in both settings. This will say that the size of our terrain is going to be `200 x 200` meters.
6. **Terrain Height** determines the maximum height possible. The white areas of our Height Map are going to be that size. We can reduce it to `500` just to limit the maximum peak of our mountains:

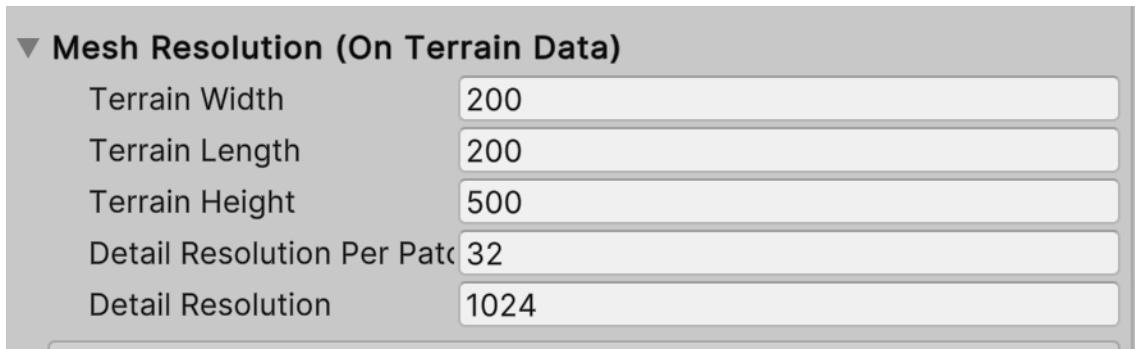


Figure 3.6: Terrain resolution settings

7. Look for the **Texture Resolutions (On Terrain Data)** section.
8. Change **Heightmap Resolution** to 257 x 257:

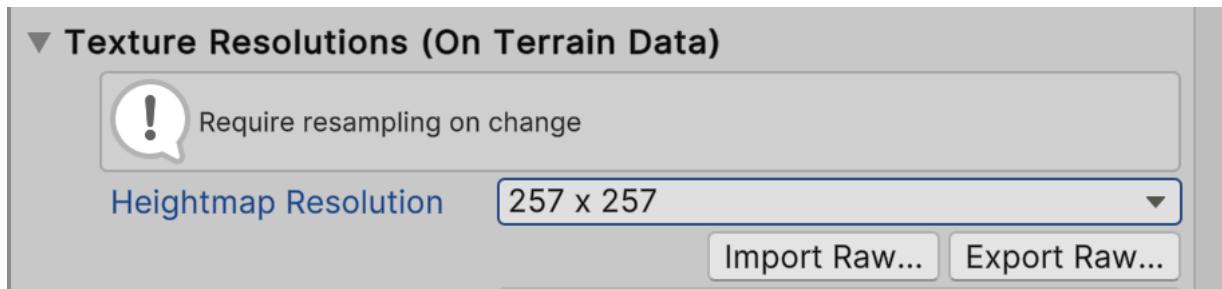


Figure 3.7: Height Map resolution settings

Heightmap Resolution is the size of the Height Map image that will hold the heights of the different parts of the terrain. Using a resolution of 257 x 257 in our 200 x 200-meter terrain means that each square meter in the terrain will be covered by a little bit more than 1 pixel of the Height Map. The higher the resolution per square meter, the greater detail you can draw in that area size. Usually, terrain features are big, so having more than 1 pixel per square meter is generally a waste of resources. Find the smallest resolution you can have that allows you to create the details you need. Another initial setting you will want to set is the initial terrain height. By default, this is 0, so you can start painting heights from the bottom part, but this way, you can't make holes in the terrain because it's already at its lowest point. Setting up a small initial height allows

you to paint river paths and pits in case you need them. In order to do so, do the following:

1. Select our **Terrain** in the **Hierarchy** panel.
2. Click on the **Paint Terrain** button (the second button).
3. Set the dropdown to **Set Height** if it's not already there.
4. Set the **Height** property to `50`. This will indicate that we want all the terrain to start at `50` meters in height, allowing us to make holes with a maximum depth of `50` meters:

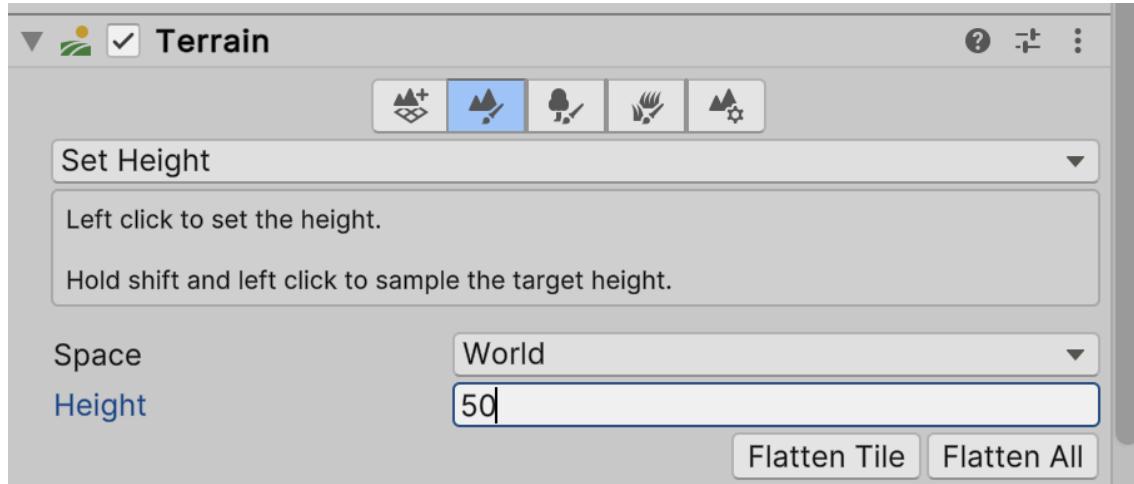


Figure 3.8: Set Height Terrain tool location

5. Click the **Flatten All** button. You will see all the terrain has been raised to the `50` meters we specified. This leaves us with `450` more meters to go up, based on the maximum of `500` meters we specified earlier.

Now that we have properly configured our Height Map, let's start editing it.

Authoring Height Maps

Remember that the Height Map is just an image of the heights, so in order to edit it, we need to paint the heights in that image.

Luckily, Unity has tools that allow us to edit the terrain directly in the editor and see the results of the modified heights directly. In order to do this, we must follow these steps:

1. Select our **Terrain** in the **Hierarchy** panel.
2. Click the **Paint Terrain** button (the second button, the same as in the previous section).
3. Set the dropdown to **Raise or Lower Terrain**:

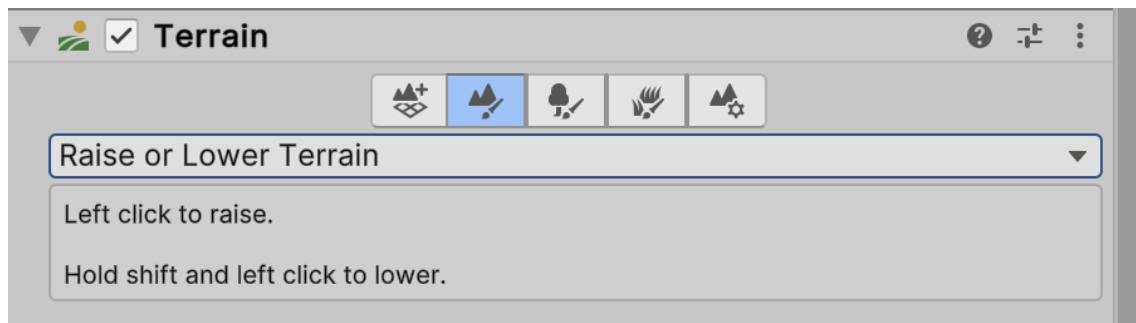


Figure 3.9: Raise or Lower Terrain tool location

4. Select the second brush in the **Brushes** selector. This brush has blurred borders to allow us to create softer heights.
5. Set the **Brush Size** to `30` so that we can create heights that span 30-meter areas. If you want to create subtler details, you can reduce this number.
6. Set **Opacity** to `10` to reduce the amount of height we paint per second, or click:

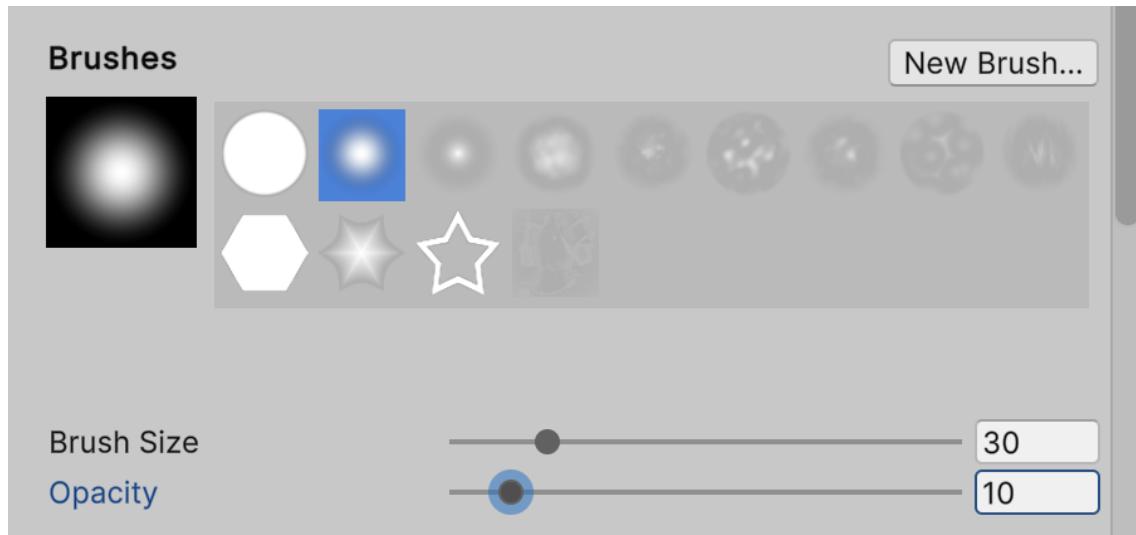


Figure 3.10: Smooth edges brush

7. Now, if you move the mouse in the **Scene** view, you will see a little preview of the height you will paint if you click on that area. You may need to navigate closer to the terrain to see it in detail:

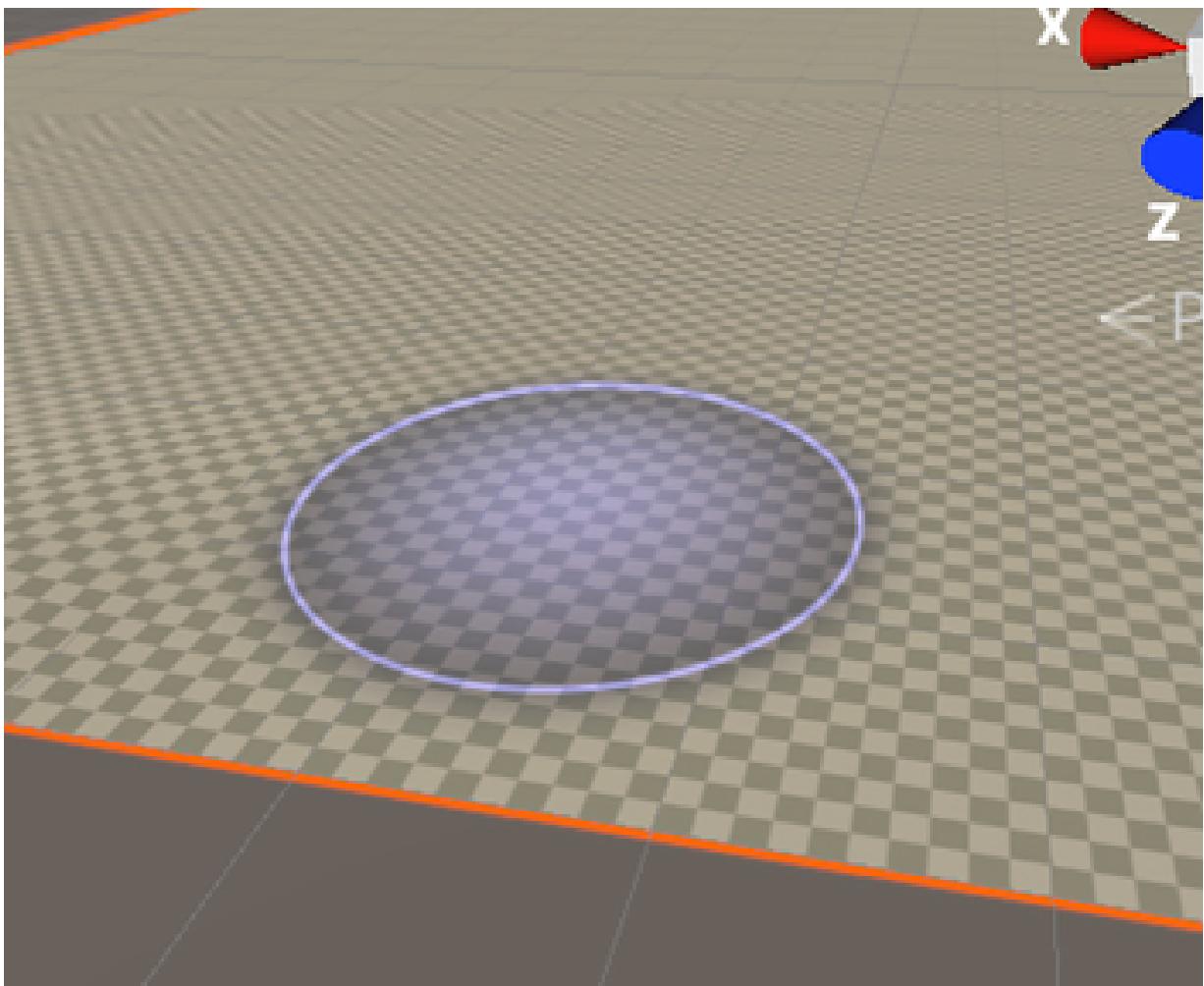


Figure 3.11: Previsualization of the area to raise the terrain

That checked pattern you can see allows you to see the actual size of the objects you are editing. Each cell represents a square meter. Remember that having a reference to see the actual size of the objects you are editing helps to prevent you from creating terrain features that are too big or too small. You could also put in other kinds of references, such as a big cube with accurate sizes representing a building to get a notion of the size of the mountain or lake you are creating. Remember that the cube has a default size of $1 \times 1 \times 1$ meters, so scaling to $10, 10, 10$ will give you a cube of $10 \times 10 \times 10$ meters.

1. Hold, left-click, and drag the cursor over the terrain to start painting your terrain heights. Remember that you can press Ctrl + Z (Command + Z on Mac) to reverse any undesired change.
2. Try to paint the mountains all around the borders of our area, which will represent the background hills of our base:

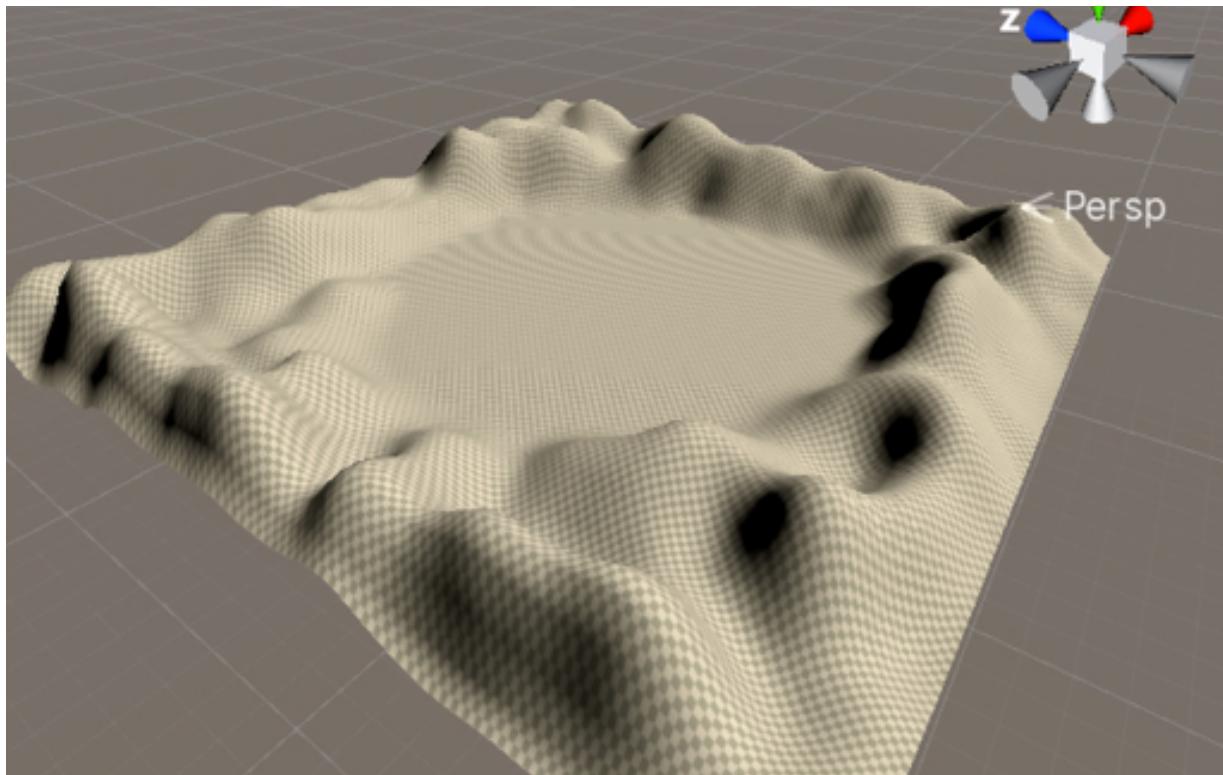


Figure 3.12: Painted mountains around the edges of the terrain

We now have decent starter hills around our future base. We can also draw a moat around our future base. To do so, follow these steps:

1. Place a cube with a scale of $50, 10, 50$ in the middle of the terrain. This will act as a placeholder for the base we are going to create:

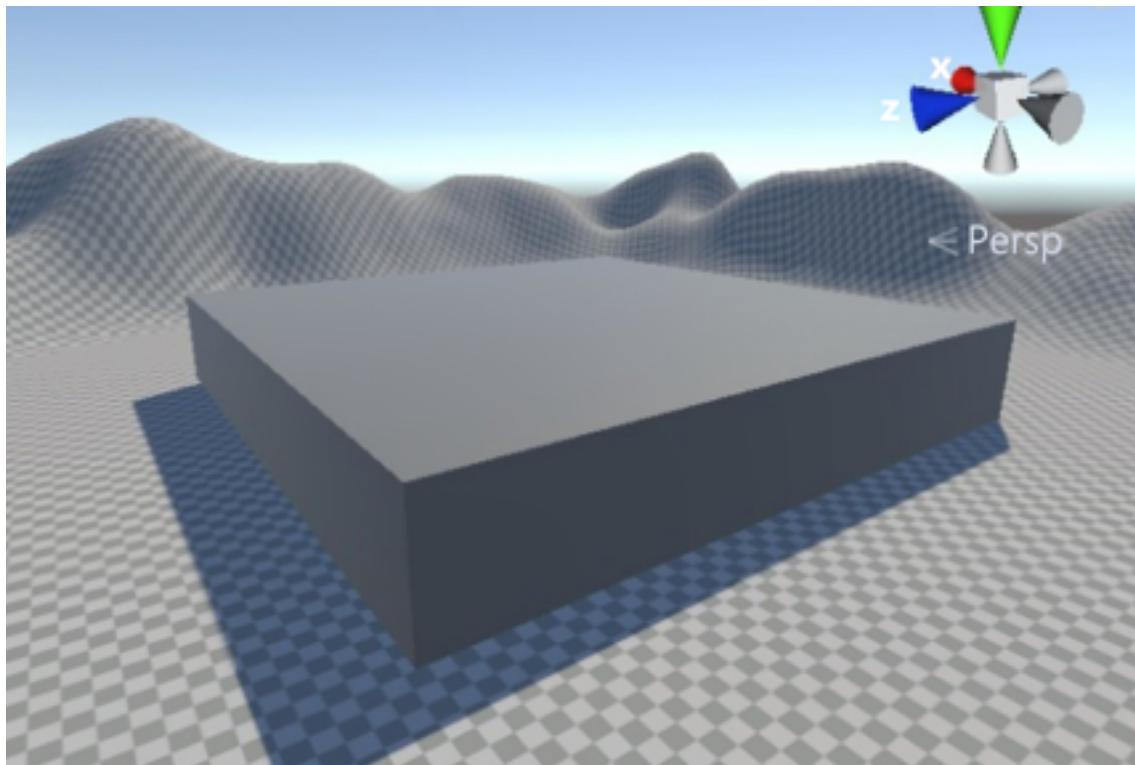


Figure 3.13: Placeholder cube for the base area

2. Select **Terrain** and the **Brush** button once more.
3. Reduce **Brush Size** to `10`.
4. Holding the Shift key, left-click and drag the mouse over the terrain to paint the basin around our base placeholder. Doing this will lower the terrain instead of raising it:

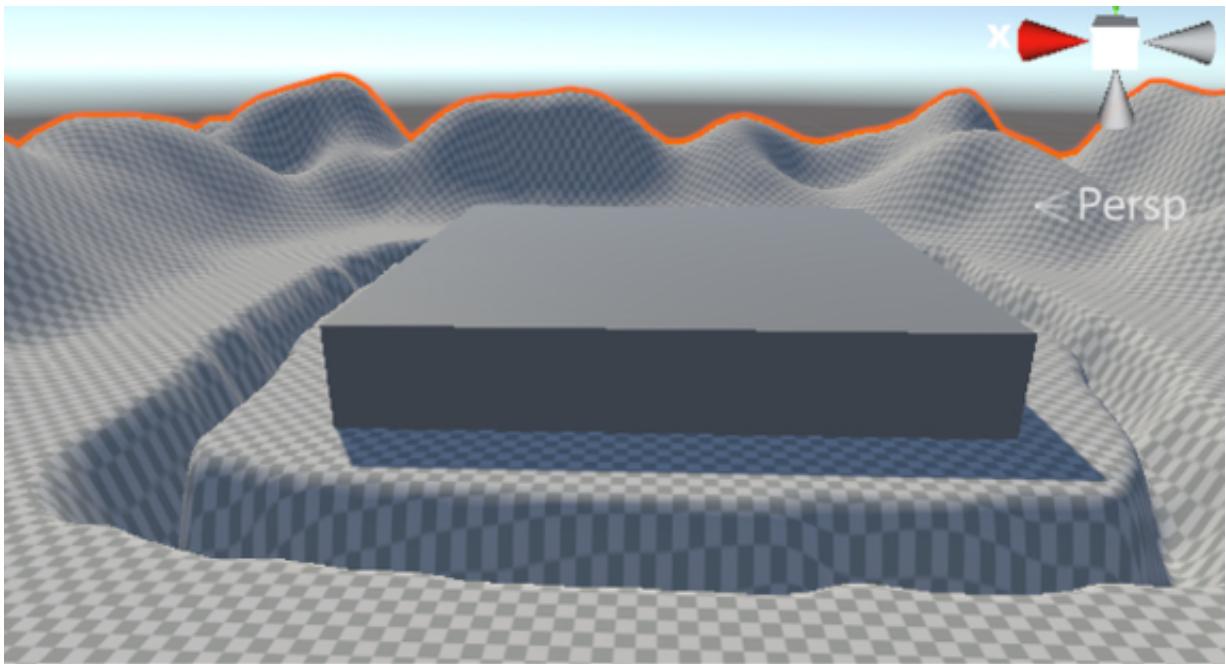


Figure 3.14: Moat around our placeholder base

Now, we have a simple but good starter terrain that gives us a basic idea of how our base and its surroundings will look. Before moving on, we will apply some finer details to make our terrain look a little bit better. In the next section, we will discuss how to simulate terrain erosion with different tools.

Memory

Before learning Unity, I was making games using DirectX, a low-level graphics library. While it was a challenge, I really enjoyed learning the algorithms needed to generate my own terrain system. While an engine provides a practical way to make games, making your own tools can also be a great way to better understand how those engines work, learn their capabilities and limits, and how to sort them.

Adding Height Map details

In the previous section, we created a rough outline of the terrain. If you want to make it look a little bit more realistic, then you need to start painting lots of tiny details here and there. Usually, this is done later in the level design process, but let's take a look now since we are exploring Terrain Tools. Right now, our mountains look very smooth. In real life, they are generally sharper, so let's improve that:

1. Select the **Terrain** and click the **Brush** button as in the previous sections.
2. Set the dropdown to **Raise or Lower Terrain** if it's not already set.
3. Pick the fifth brush, as shown in *Figure 3.15*. This brush has an irregular shape so that we can paint a little bit of noise here and there.
4. Set the **Brush Size to 50** so that we can cover a greater area:

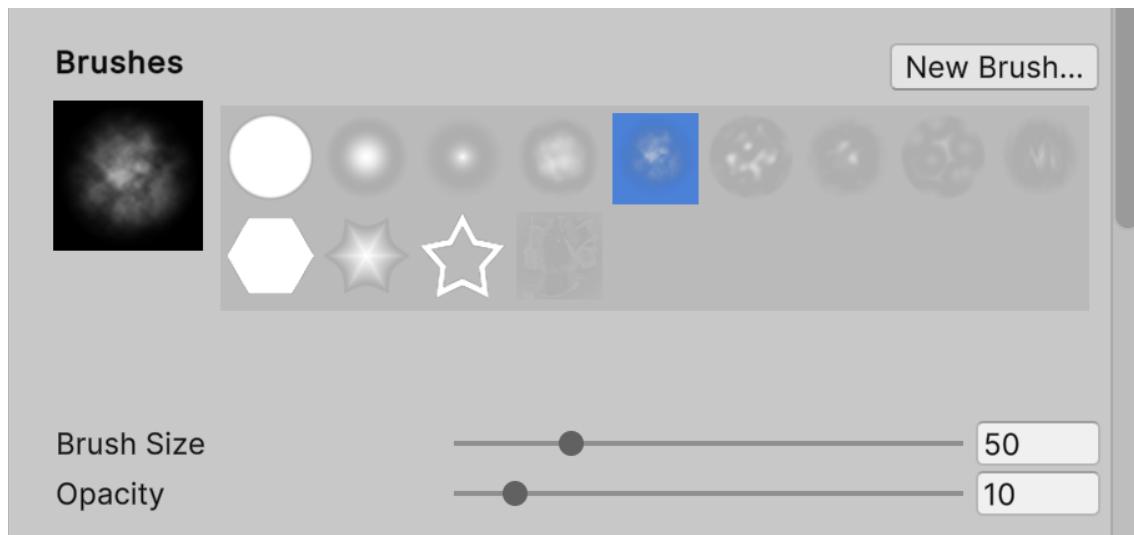


Figure 3.15: Cloud pattern brush for randomness

5. Hold Shift and do small clicks over the hills of the terrain without dragging the mouse. Remember to zoom into the areas you are applying finer details to because they can't be seen at great distances:

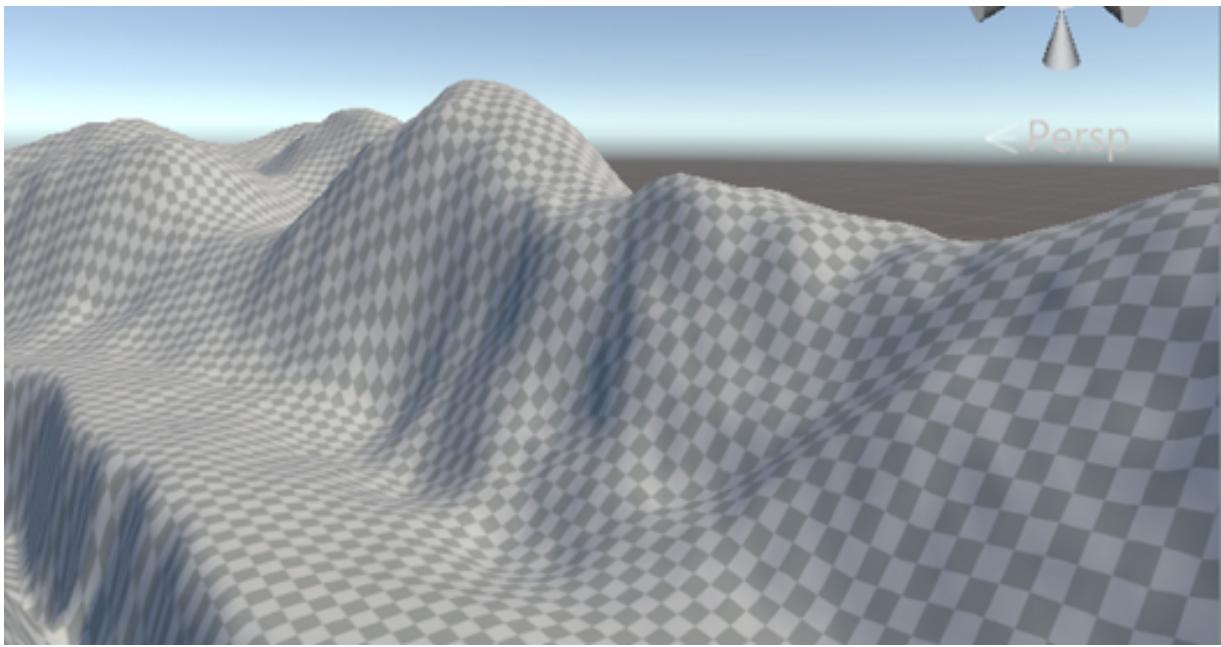


Figure 3.16: Erosion generated with the aforementioned brush

This has added some irregularity to our hills. Now, let's imagine we want to have a flat area on the hills to put a decorative observatory or antenna. Follow these steps to do so:

1. Select **Terrain**, **Brush Tool**, and **Set Height** from the dropdown.
2. Set **Height** to 60 .
3. Select the full-circle brush (the first one).
4. Paint an area over the hills. You will see that the terrain will rise if it's lower than 60 meters or drops in areas higher than 60 meters:

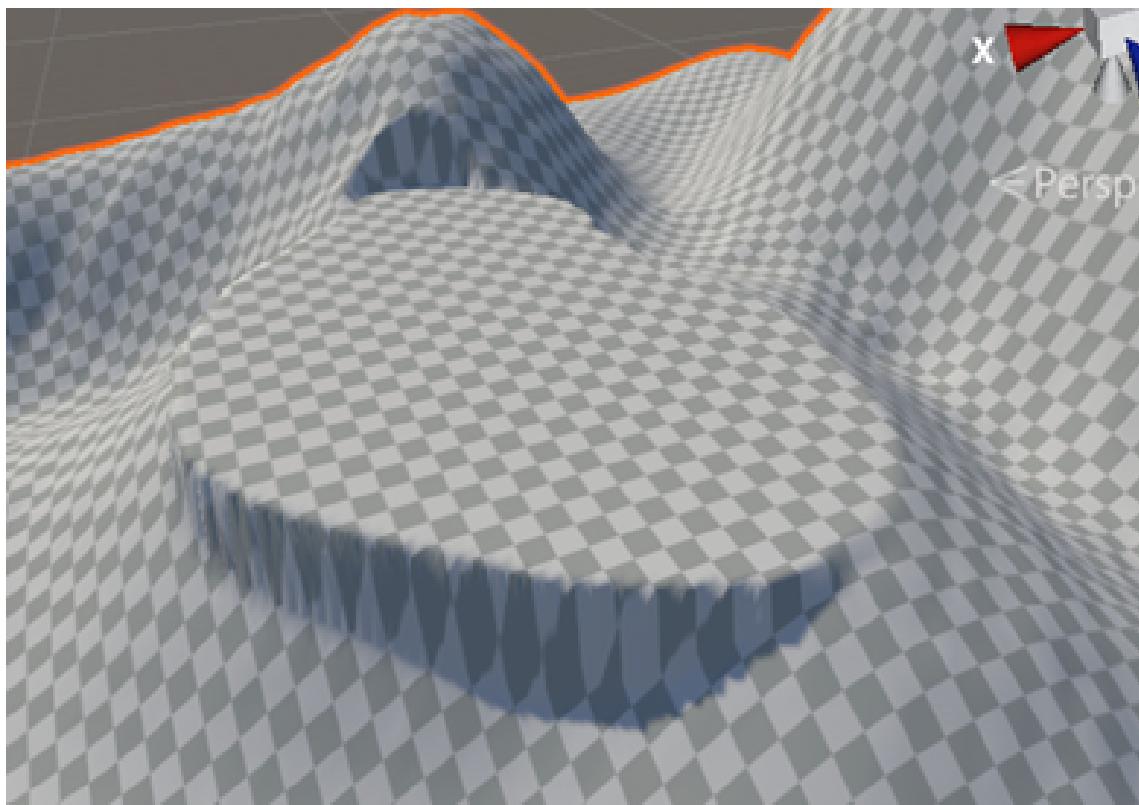


Figure 3.17: Flattened hill

5. You can see that the borders have some rough corners that need to be smoothed:

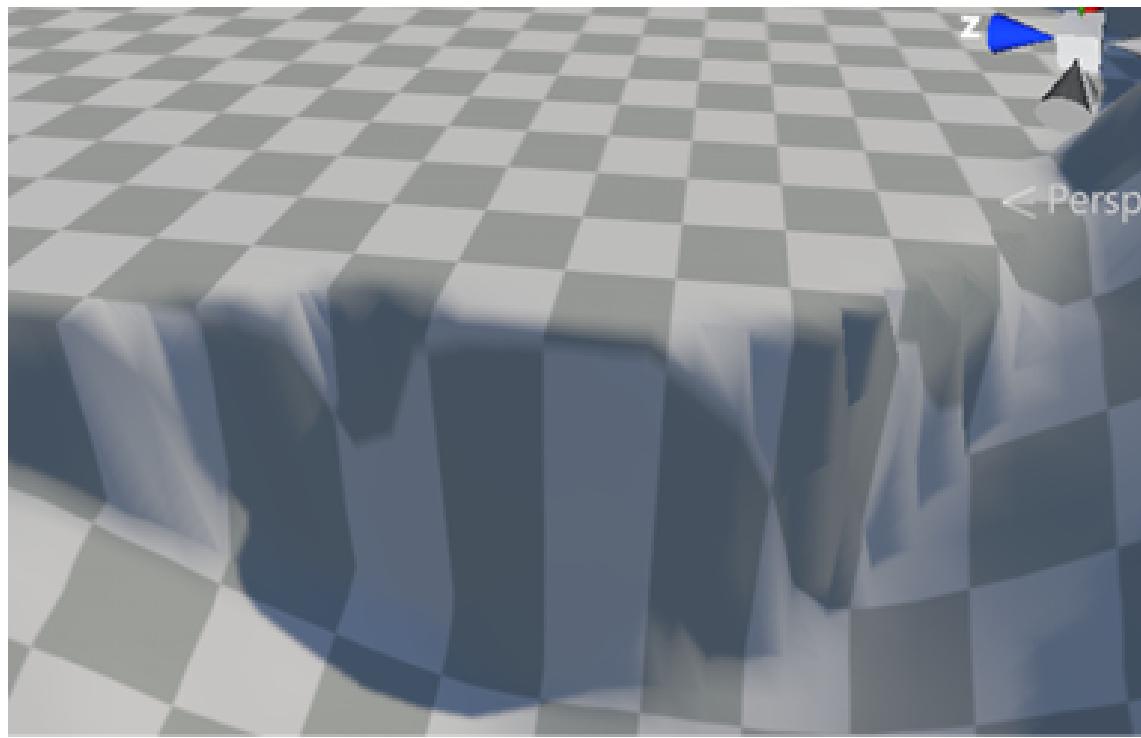


Figure 3.18: Non-smoothed terrain edges

6. Change the dropdown to **Smooth Height**.
7. Select the second brush, as shown in *Figure 3.19*, with a size of 5 and an opacity of 10 :

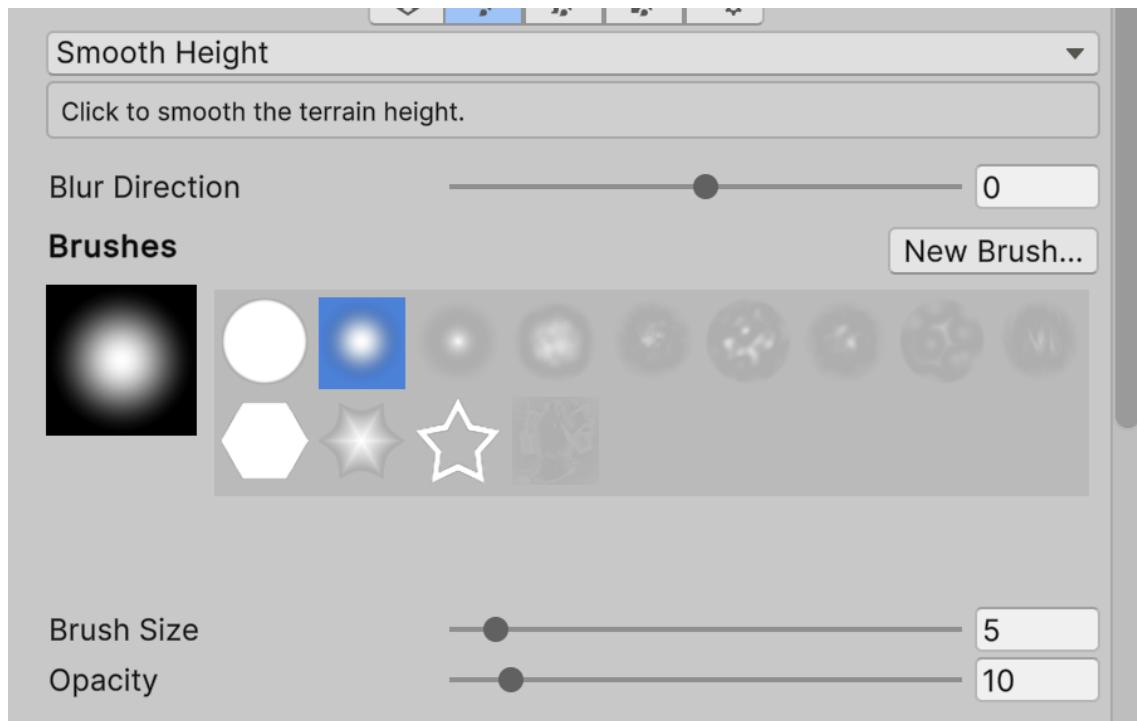


Figure 3.19: Smooth Height brush selected

8. Click and drag over the borders of our flat area to make them smoother:

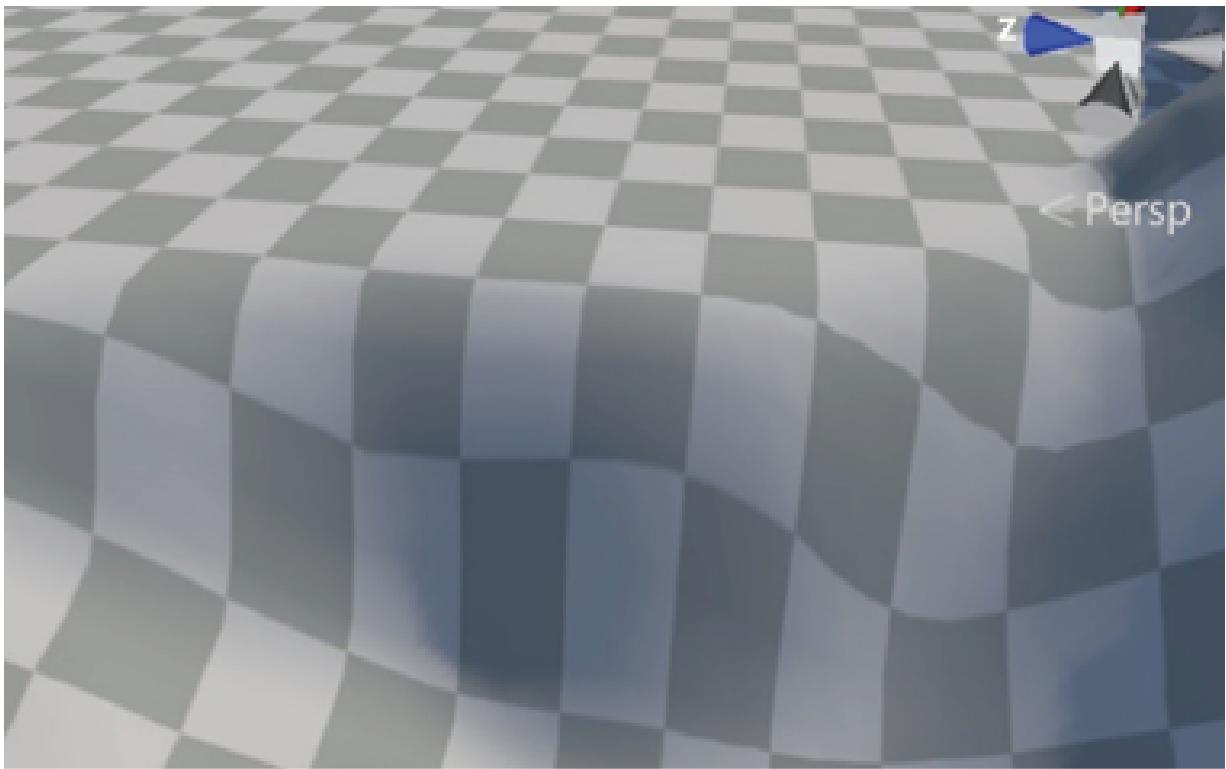


Figure 3.20: Smoothed terrain edges

If you want to dive deep into Terrain, there's the Terrain Tools extension package, which adds more tools to sculpt it with finer details. Check this documentation:

<https://docs.unity3d.com/Packages/com.unity.terrain-tools@5.1> and also this video:

<https://www.youtube.com/watch?v=smnLYvF4Os4> for more info.

We could keep adding details here and there, but we can settle with this for now. The next step is to create our player's base, but first, let's explore ProBuilder in order to generate our geometry.

Creating shapes with ProBuilder

So far, we have created simple scenes using cubes and primitive shapes, and that's enough for most of the prototypes you will create, but sometimes, you will have tricky areas of the game that would be

difficult to model with regular cubes, or maybe you want to have some deeper details in certain parts of your game to get an idea of how the player will experience that area. In this case, we can use any 3D modeling tool for this, such as 3D Studio Max, Maya, or Blender, but they can be difficult to learn, and you probably won't need all your power at this stage in your development. Luckily, Unity has a simple 3D model creator called ProBuilder, so let's explore it. In this section, we will cover the following concepts related to ProBuilder:

- Installing ProBuilder
- Creating a shape
- Manipulating the mesh
- Adding details

ProBuilder is not included by default in our Unity project, so let's start by learning how to install it.

Installing ProBuilder

Unity is a powerful engine full of features, but adding all those tools to our project if we are not using all of them can make the engine run more slowly, so we need to manually specify which Unity tools we are using. To do so, we will use **Package Manager**, a tool that we can use to select which Unity packages we are going to need. As you may recall, earlier, we talked about the `Packages` folder. This is basically what the Package Manager is modifying. In order to install ProBuilder in our project with this tool, we need to do the following:

1. Click the **Window | Package Manager** option:

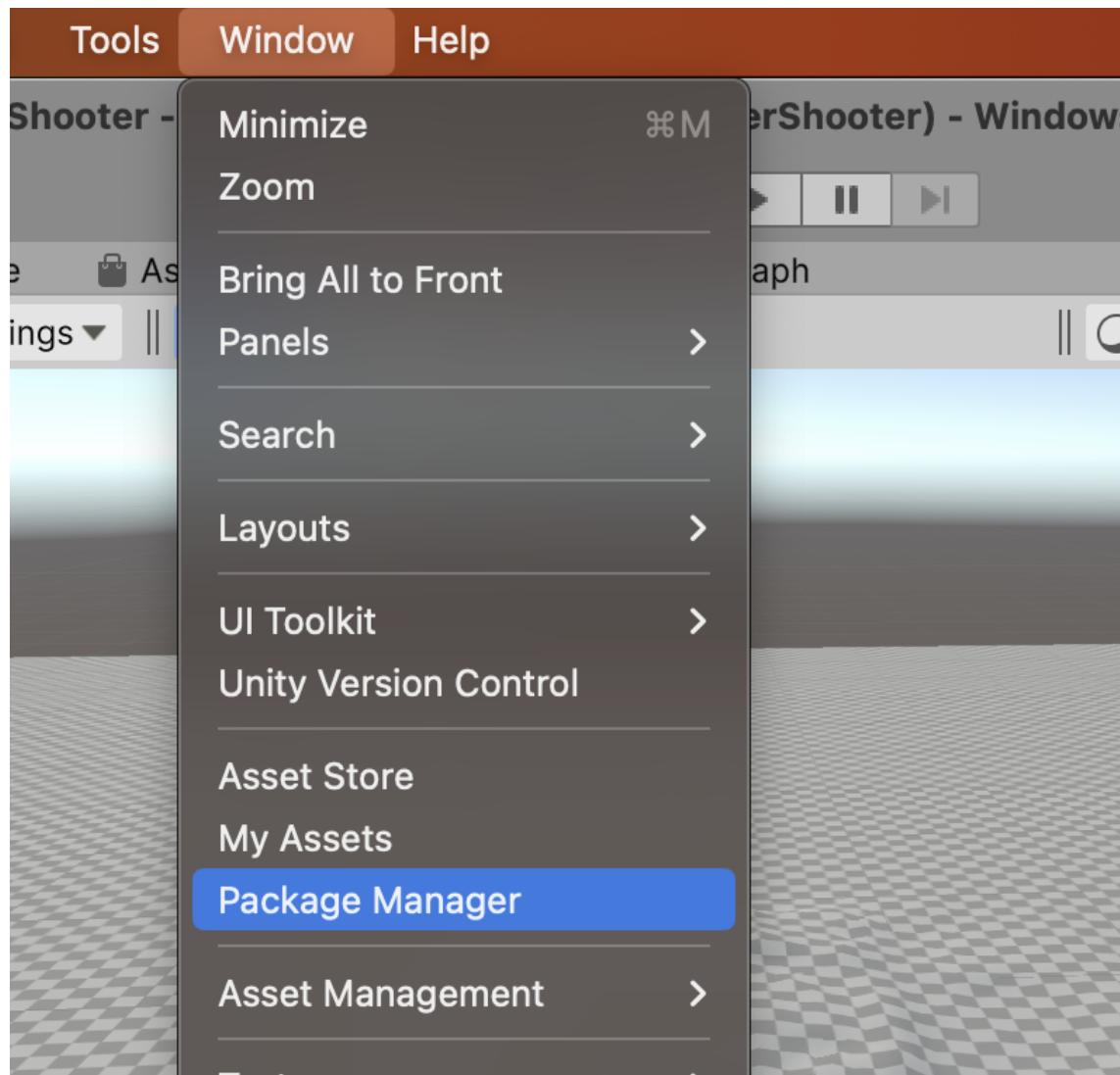


Figure 3.21: Package Manager option

2. In the window that just opened, ensure the **Packages** mode is in **Unity Registry** mode by clicking on the button saying **Packages** in the top-left part of the window and selecting **Unity Registry**. Unlike the **In Project** option, which will show only the packages our project already has, **Unity Registry** will show all the official Unity packages you can install:

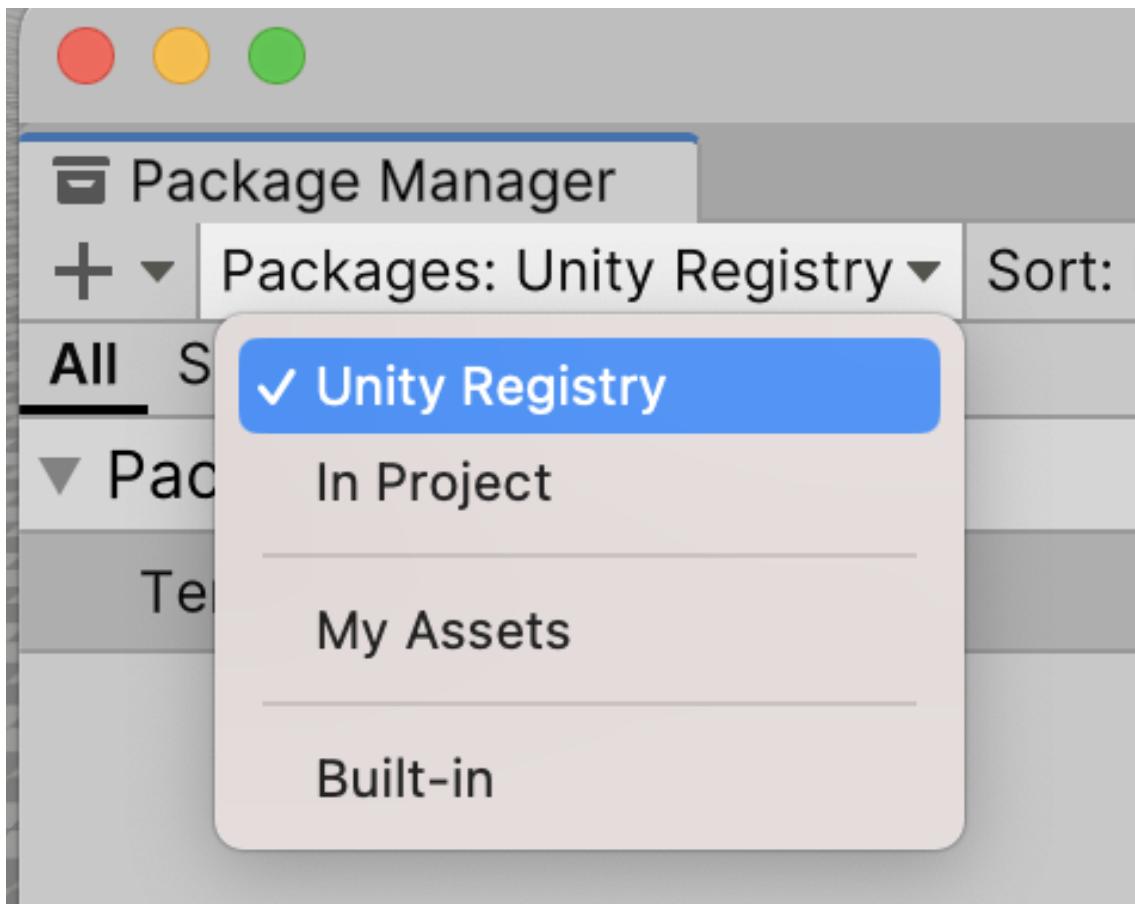


Figure 3.22: Showing all packages

3. Wait a moment for the list of packages left to fill. Make sure you are connected to the internet to download and install the packages.
4. Look at the **ProBuilder** package in that list and select it. You can also use the search box in the top-right corner of the **Package Manager** window:

A screenshot of the Unity Package Manager interface. On the left, there's a sidebar with tabs for 'FROM' (selected), 'com.unity.*', 'Docu...', 'Descr...', 'Build, level...', and 'Advanced...'. The main area shows a list of packages:

Polybrush	1.1.4
Post Processing	3.3.0
ProBuilder	5.0.7 ✓
Profile Analyzer	1.2.2
Python Scripting	7.0.0
Recorder	4.0.1
Replay	1.0.5

Figure 3.23: ProBuilder in the packages list

I'm using ProBuilder version 5.0.7, the newest version available at the time of writing this book. While you can use a newer version, the process of using it may differ. You can look at older versions using the arrow to the left of the title.

1. Click on the **Install** button in the top-right-hand corner of the **Package Manager**:

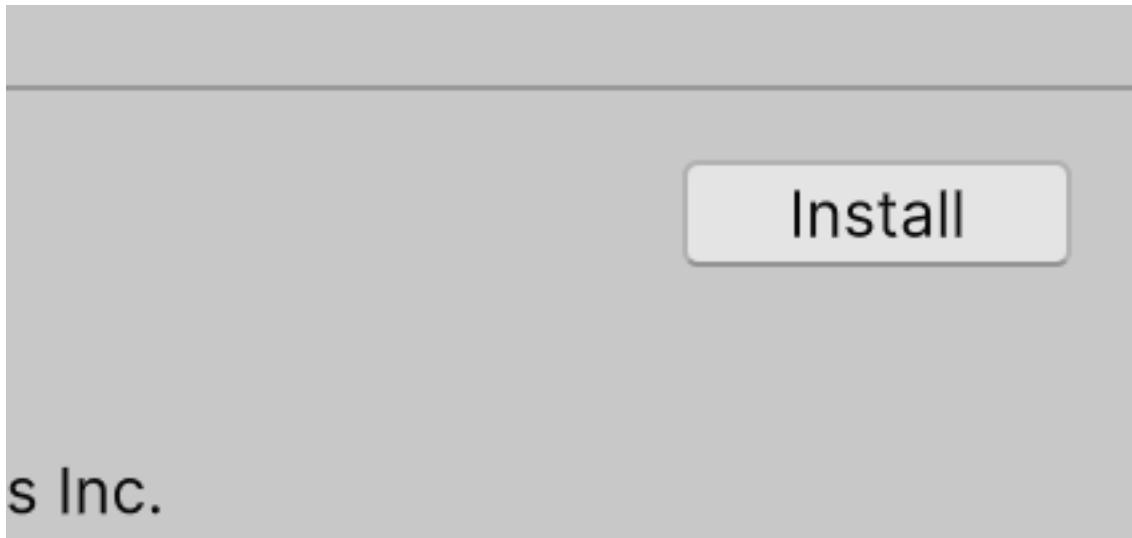


Figure 3.24: Install button

2. Wait for the package to install; this can take a while. You can tell that the process has ended when the **Install** button has been replaced with the **Remove** label after the **Importing** popup finishes. If, for some reason, Unity freezes or takes more than 10 minutes, feel free to restart it.
3. Go to **Edit | Preferences** on Windows (**Unity | Preferences** on Mac).
4. Select the **ProBuilder** option from the left list.
5. Set **Vertex Size** to **2** and **Line Size** to **1**. This will help you to better visualize the 3D model we are going to create while editing its different parts:

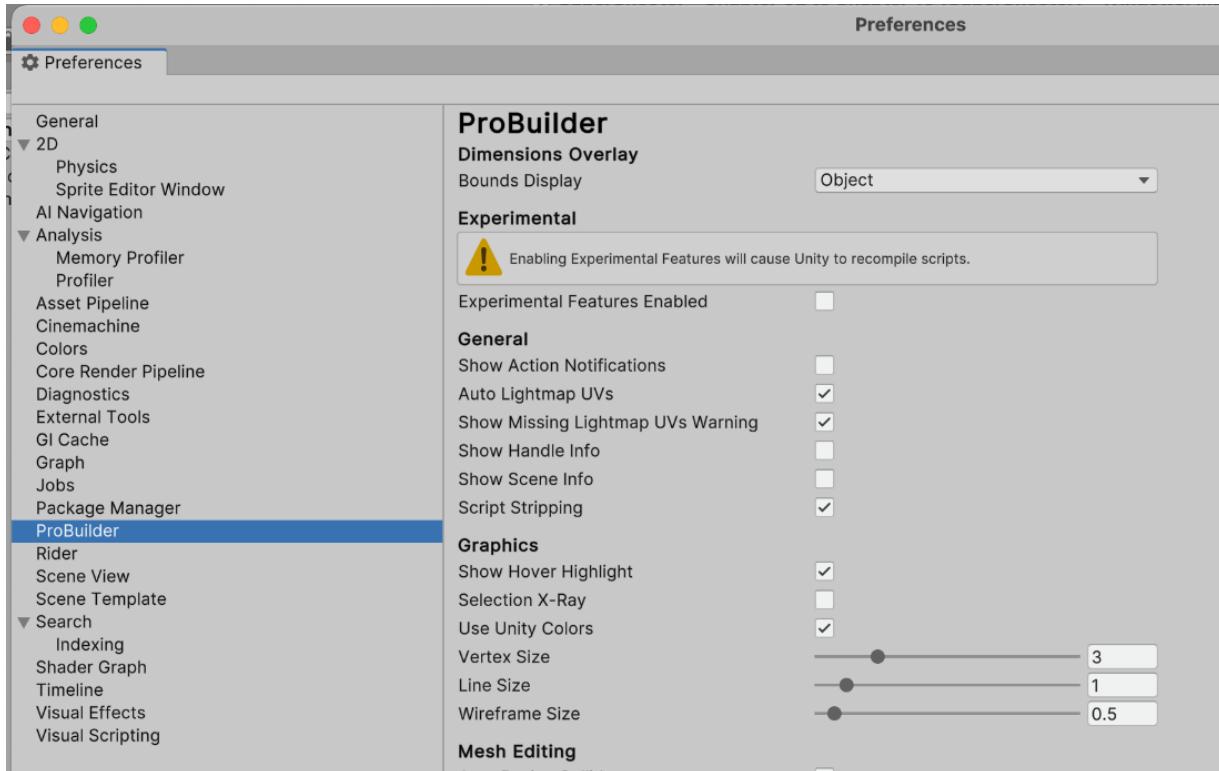


Figure 3.25: Configuring ProBuilder

The **Vertex Size** and **Line Size** values are big (**2** and **1** meter, respectively) due to the fact that we are not going to edit little details of a model but big features like walls. You might want to modify it later, depending on what you are editing.

Although this is all we need to know about **Package Manager** to install ProBuilder, if you want to know more about it, you can review its documentation here:

<https://docs.unity3d.com/Manual/upm-ui.html>

Now that we have installed ProBuilder in our project, let's use it!

Creating a shape

We will start the player's base by creating a plane for our floor. We will do this by doing the following:

1. Delete the cube we placed as the base placeholder. You can do that by right-clicking on the cube in the Hierarchy and then pressing **Delete**.
2. Open ProBuilder and go to **Tools | ProBuilder | ProBuilder Window**:

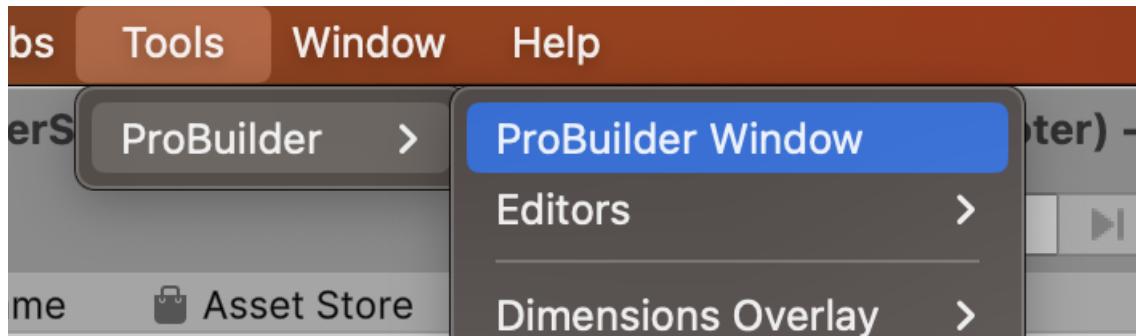


Figure 3.26: *ProBuilder Window* option

3. In the window that has opened, click the **New Shape** button:

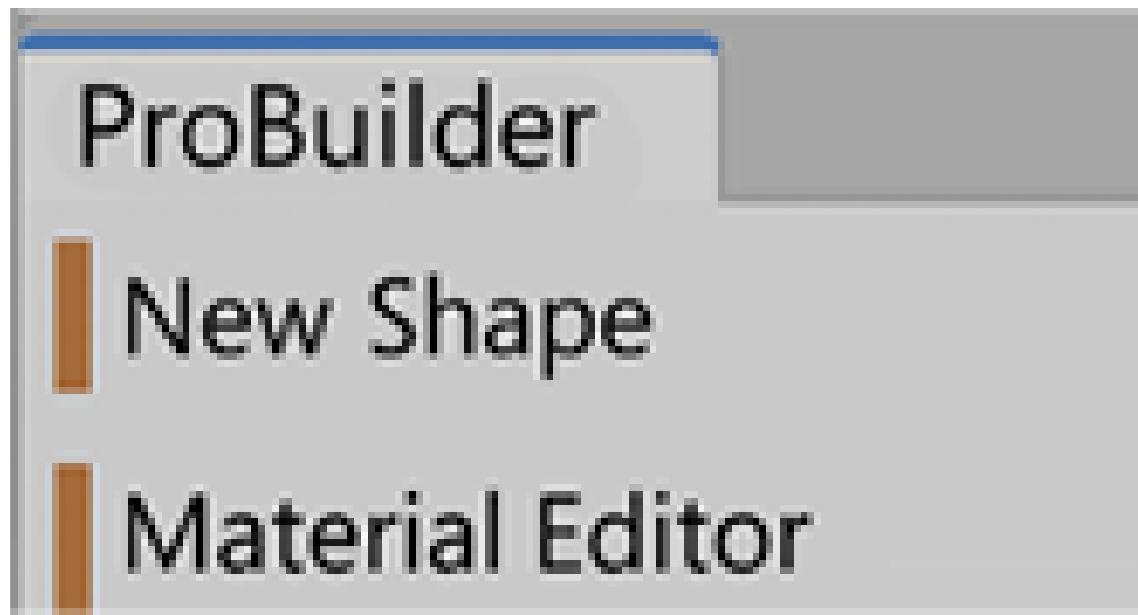


Figure 3.27: New Shape option

4. In the **Create Shape** panel that appears in the bottom-right corner of the Scene View, select the **Plane** icon (the second icon on the first row).

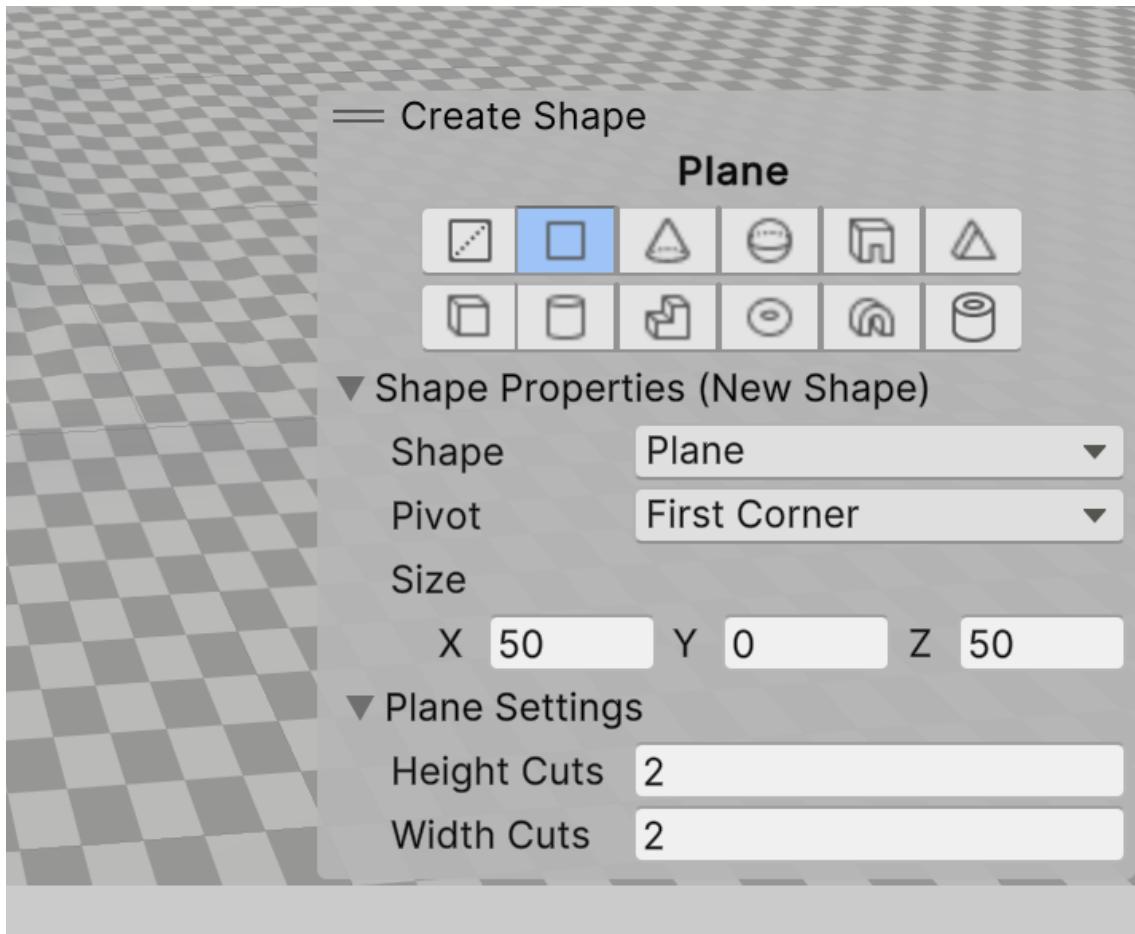


Figure 3.28: New shape created

5. Expand **Shape Properties** and **Plane Settings**.
6. Set **Width Cuts** and **Height Cuts** to 2. We will need those subdivisions later.
7. Click and drag over the terrain to draw the plane. While you do that, check how the **Size** value in the **Create Shape** panel changes, and adjust the values **x** and **z** to 50.
8. Release the mouse button and see the resulting plane.
9. Select the newly created **Plane** object in the Hierarchy and drag it slightly upward using the **Transform** tool.

We needed to move the plane upward because it was created at exactly the same height as the terrain. That caused an effect called **Z-Fighting**, where the pixels that are positioned in the same

position are fighting to determine which one will be rendered and which won't. Now that we have created the floor, let's learn how we can manipulate its vertices to change its shape.

Manipulating the mesh

If you select the plane, you will see that it is subdivided into a 3×3 grid because we set up the width and height cuts to $\frac{1}{2}$. We did that because we will use the outer cells to create our walls, thus raising them. The idea is to modify the size of those cells to outline the wall length and width before creating the walls. In order to do so, we will do the following:

1. Select the plane in the Hierarchy.
2. Open ProBuilder if it's not already open, and go to the **Tools | ProBuilder | ProBuilder Window** option.
3. Select the second button (vertex) from the four new buttons that appear in the Scene View:



Figure 3.29: Select vertices tool

4. Click the **Select Hidden** option until it says **On**, as shown in the following image. This will make selecting vertices easier:

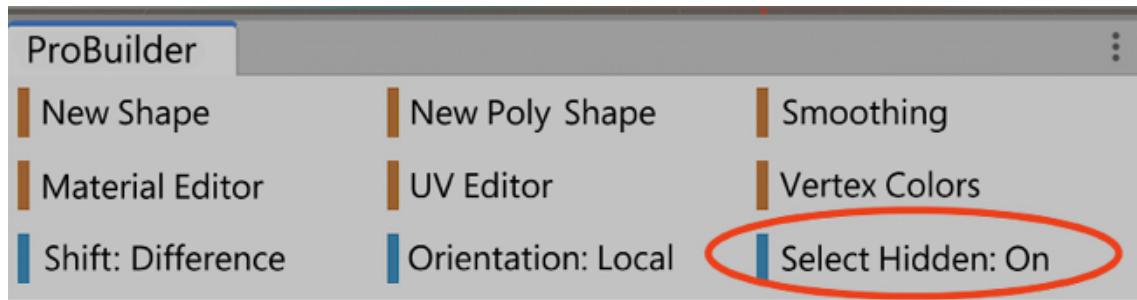


Figure 3.30: Enabling Select Hidden

5. Click and drag the mouse to create a selection box that picks the four vertices on the second row of vertices:

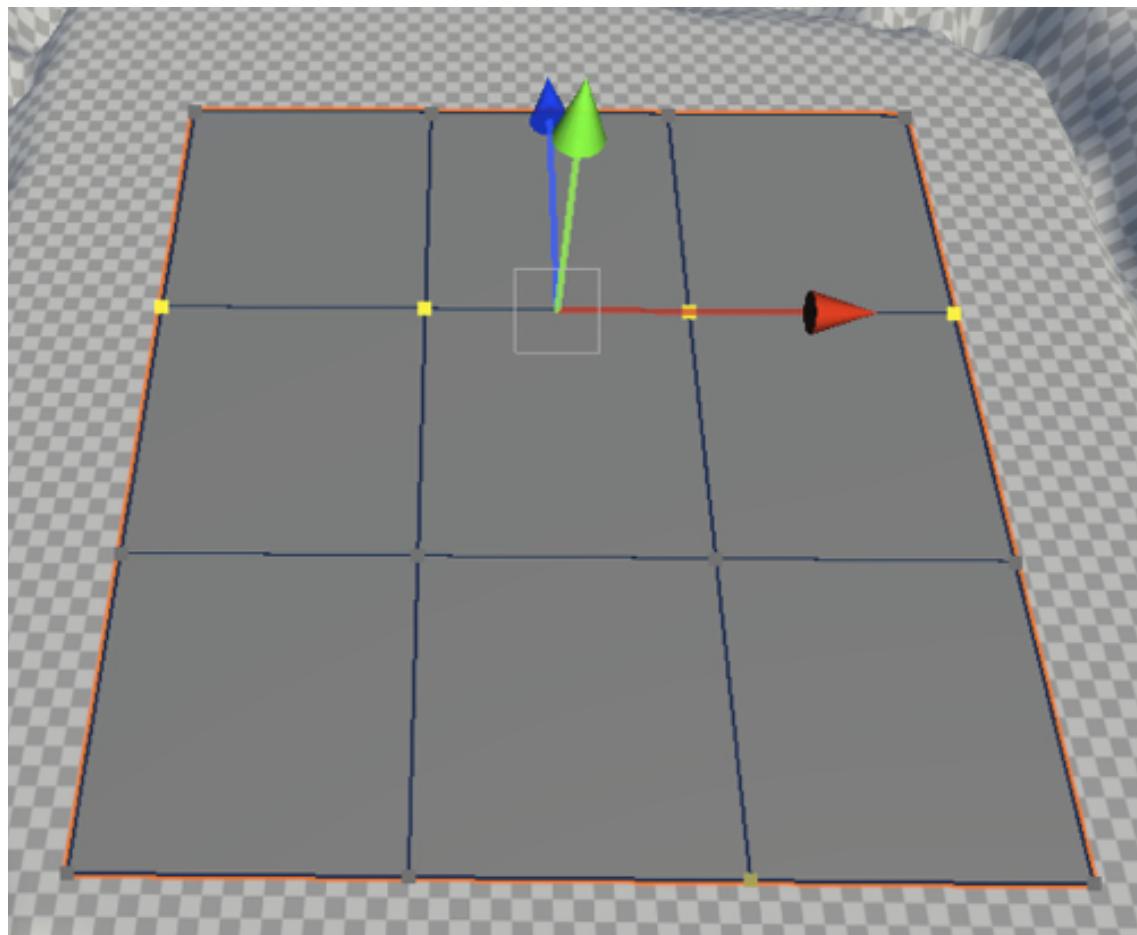


Figure 3.31: Vertex selection

6. Click on the second button in the top-left of the buttons of the Unity Editor to enable the **Move Tool**, which will allow us to move vertices. Like the **Transform Tool**, this can be used to move any object, but to move vertices, this is our only option. Remember to do this once you have selected the vertices. You can also press the W key to enable the **Move Tool**.



Figure 3.32: Move Tool

7. Move the row of vertices to make the subdivision of the plane thinner. You can use the checker pattern on the terrain to get a notion of the size of the wall in meters (remember, each square is one square meter):

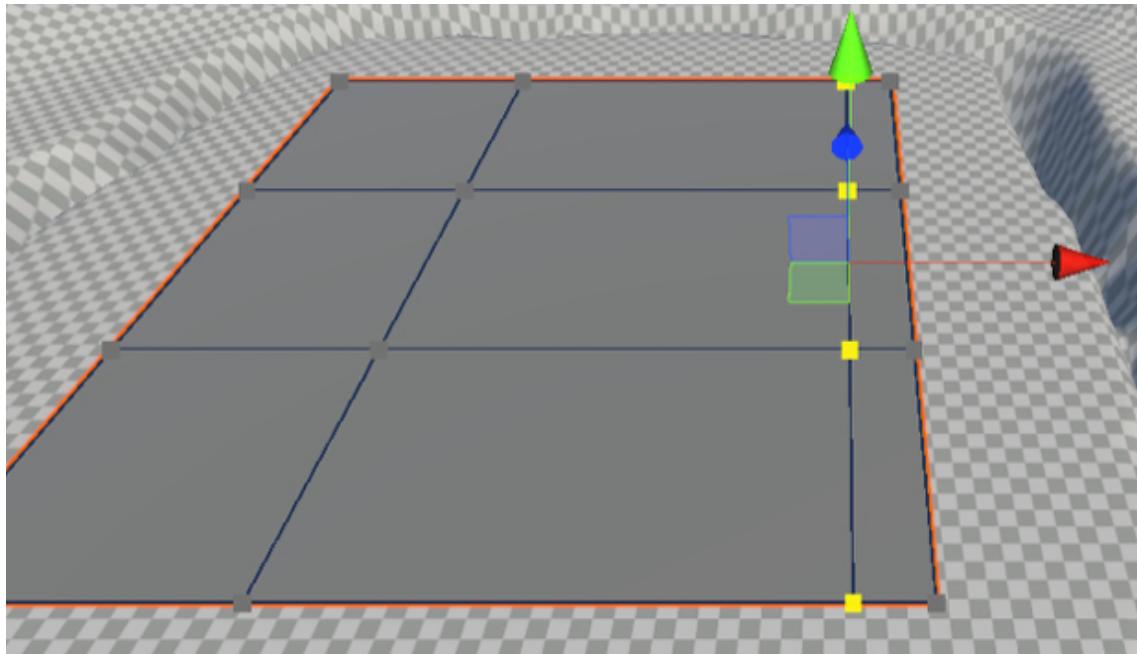


Figure 3.33: Moved vertices

8. Repeat *steps 3 to 5* for each row of vertices until you get wall outlines with similar sizes:

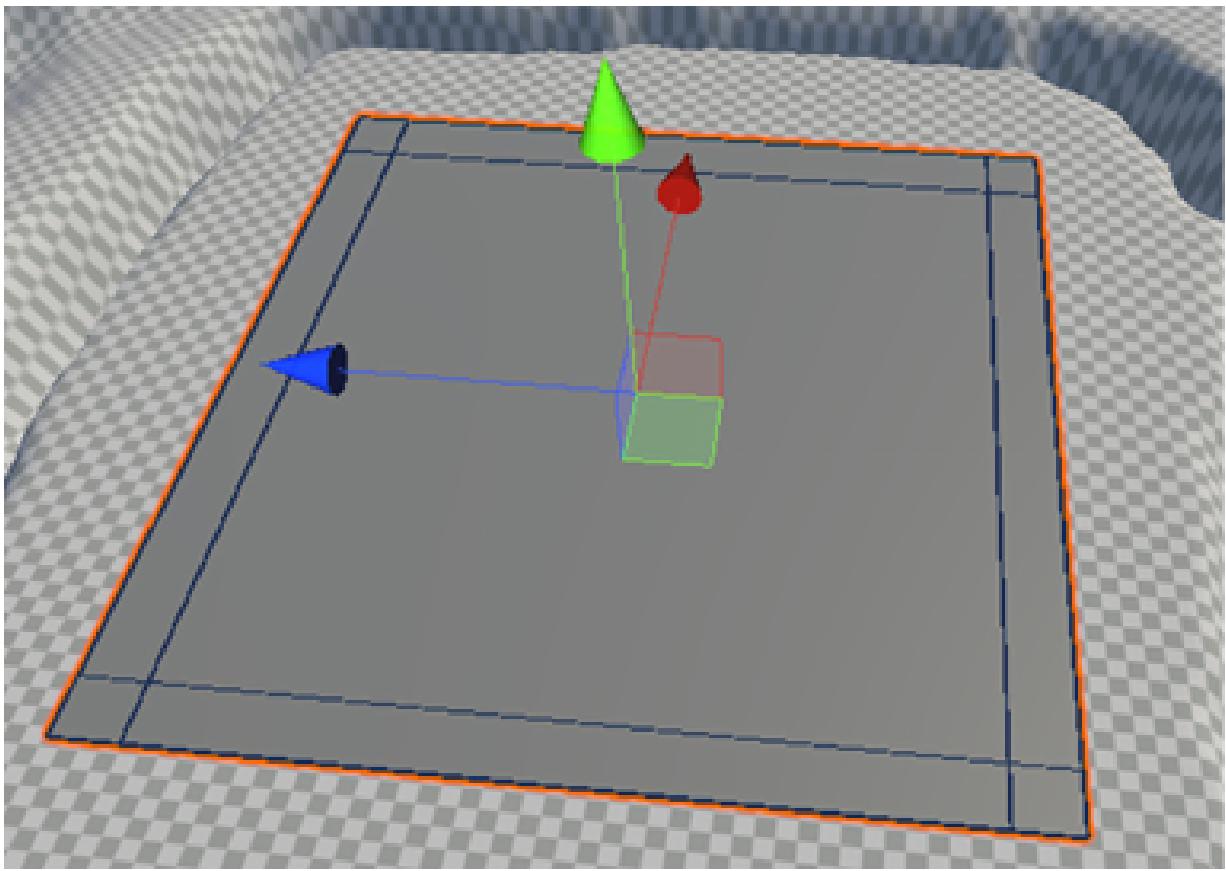


Figure 3.34: Moved vertices to reduce edges cell width

Now that we have created the outline for our walls, let's add new faces to our mesh to create them. In order to use the subdivisions or **faces**, we have created to make our walls, we must pick and extrude them. Follow these steps to do so:

1. Select the plane.
2. Select the fourth button of the **ProBuilder** buttons in the Scene View:



Figure 3.35: Select the Face tool

3. While holding Ctrl (Command on Mac), click on each of the faces of the wall outlines:

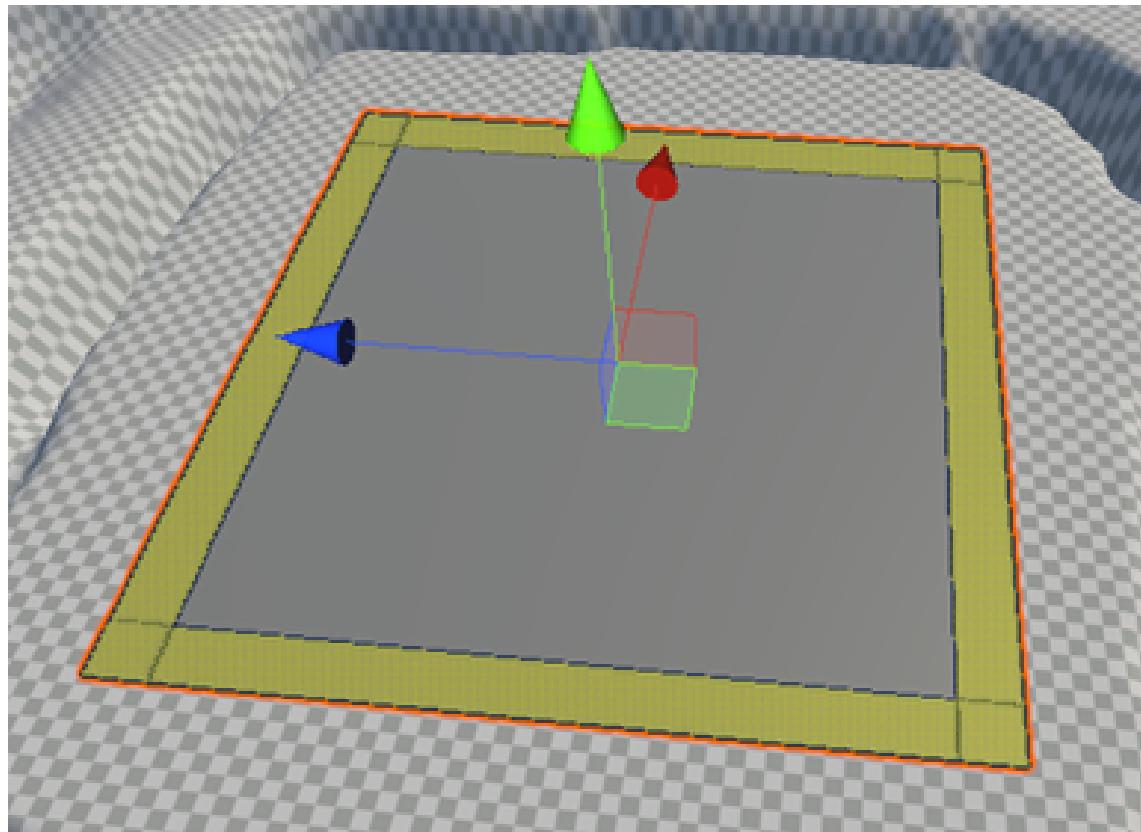


Figure 3.36: Edge faces being selected

4. In the **ProBuilder** window, look for the **plus (+)** icon to the right of the **Extrude Faces** button. It is located in the red section of the window:



Figure 3.37: Extrude Faces option

5. Set the **Distance** to **5** in the window that appears after we click the **+** button.
6. Click the **Extrude Faces** button in that window:

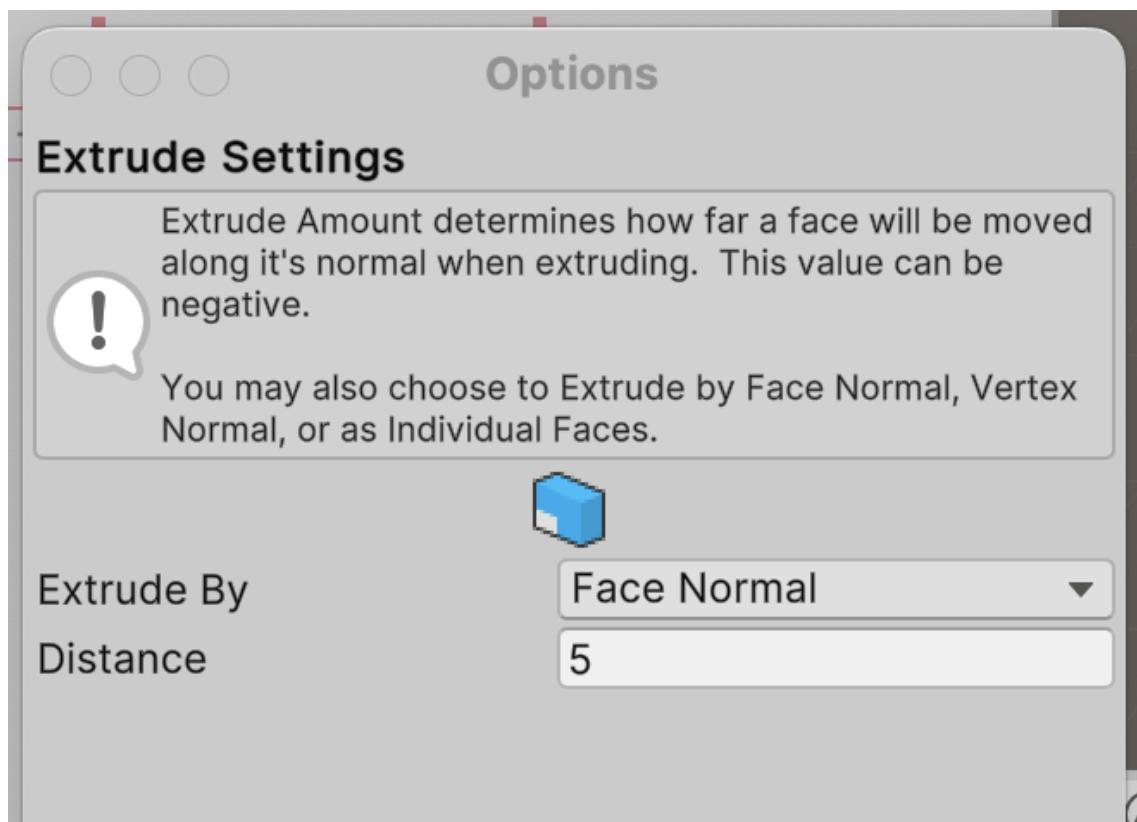


Figure 3.38: Extrude distance option

7. Now, you should see that the outline of the walls has just raised from the ground:

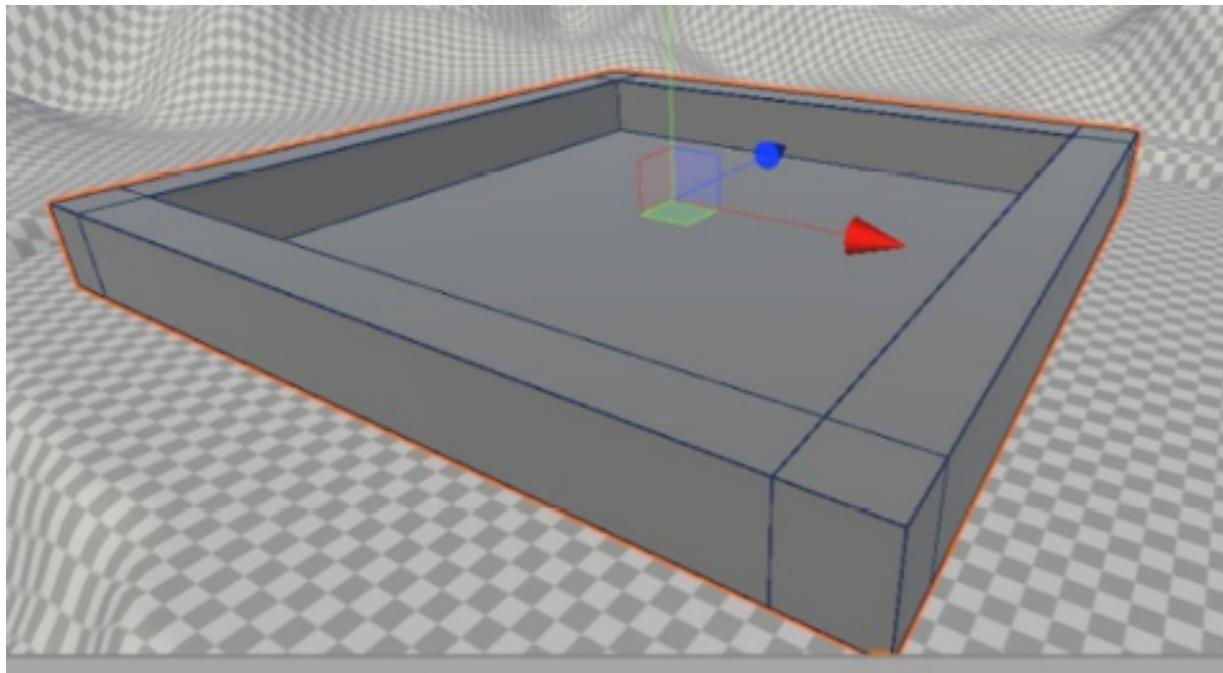


Figure 3.39: Extruded grid edges

Now, if you pay attention to how the base floor and walls touch the terrain, there's a little gap. We can try to move the base downward, but the floor will probably disappear because it will be buried under the terrain. A little trick we can do here is to push the walls downward, without moving the floor, so that the walls will be buried in the terrain, but our floor will stay a little distance from it. You can see an example of how it would look in the following image:



Figure 3.40: Slice of the expected result

In order to do this, we need to do the following:

1. Select the third **ProBuilder** button in the Scene View to enable edge selection:



Figure 3.41: Select edges tool

2. While holding Ctrl (Command on Mac), select all the bottom edges of the walls.
3. If you selected undesired edges, just click them again while holding Ctrl (Command on Mac) to deselect them while keeping the current selection:

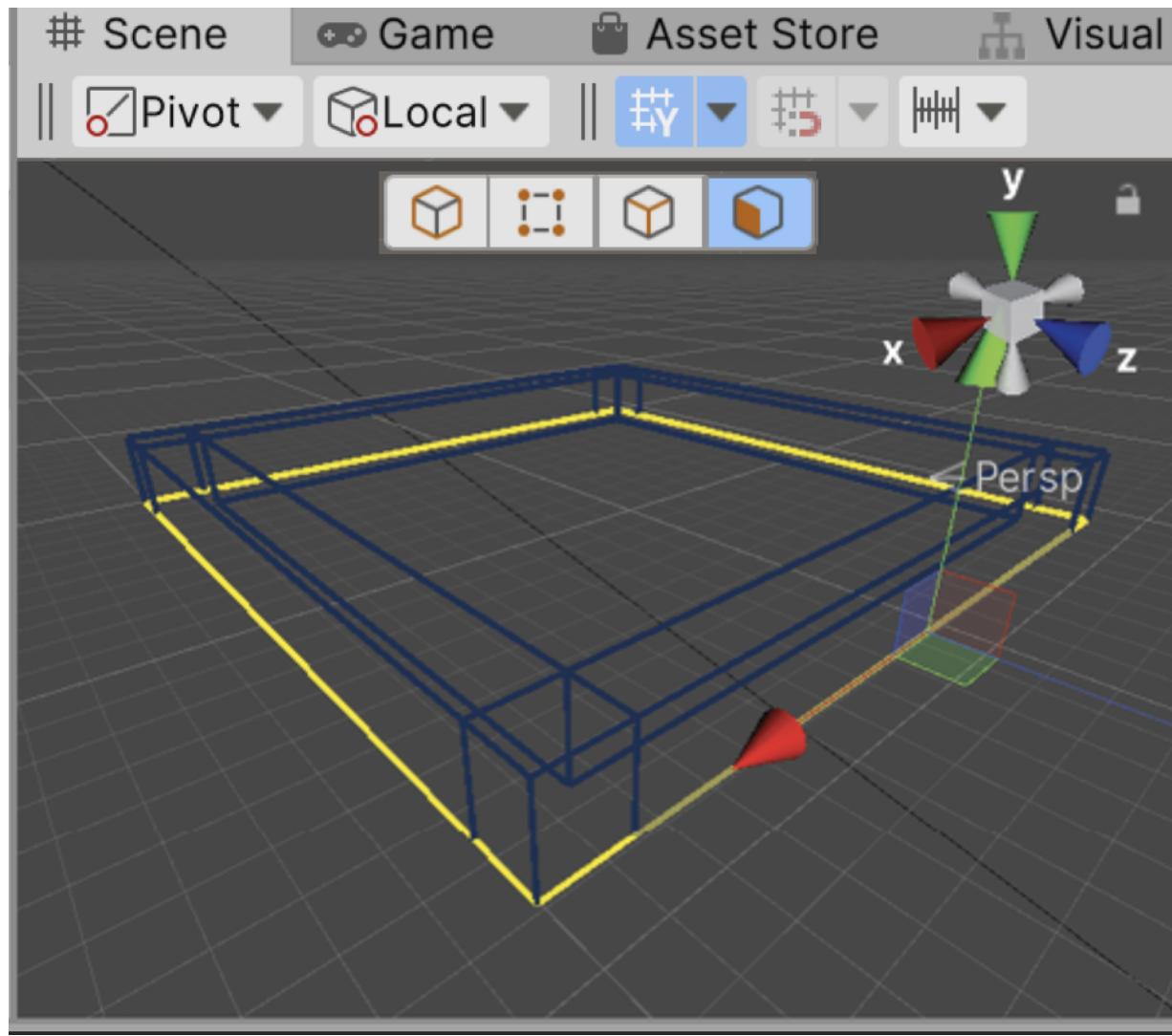


Figure 3.42: Selecting floor edges

If you want to use **Wireframe** mode in the **sphere** icon, go to the left of the 2D button in the top-right corner of the Scene View and select the **Wireframe** option from the dropdown menu, as shown in the following image. You can get back to normal by selecting **Shaded**.

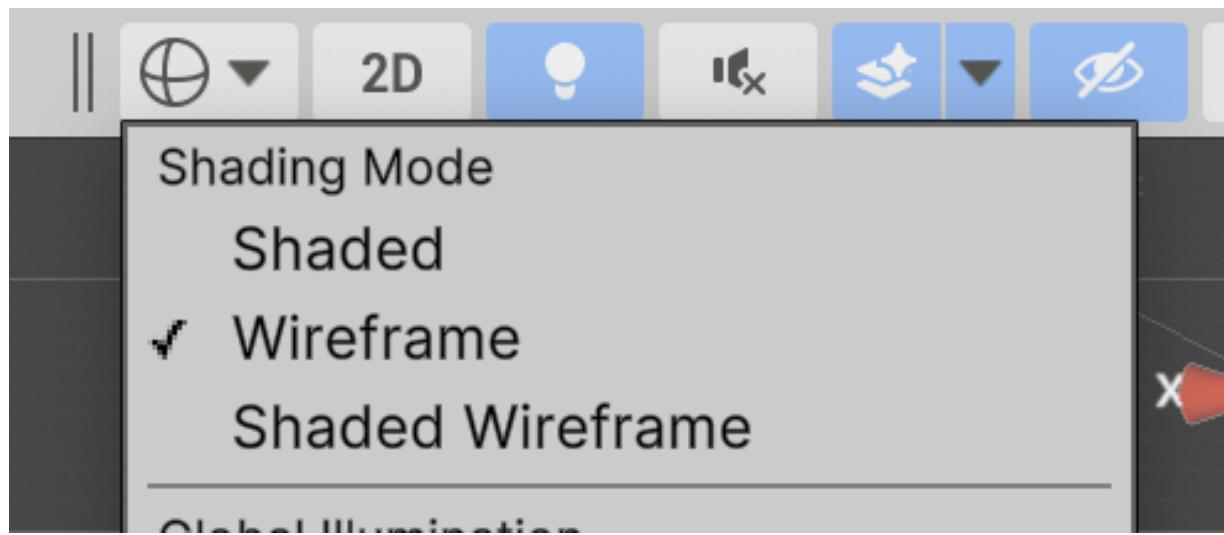
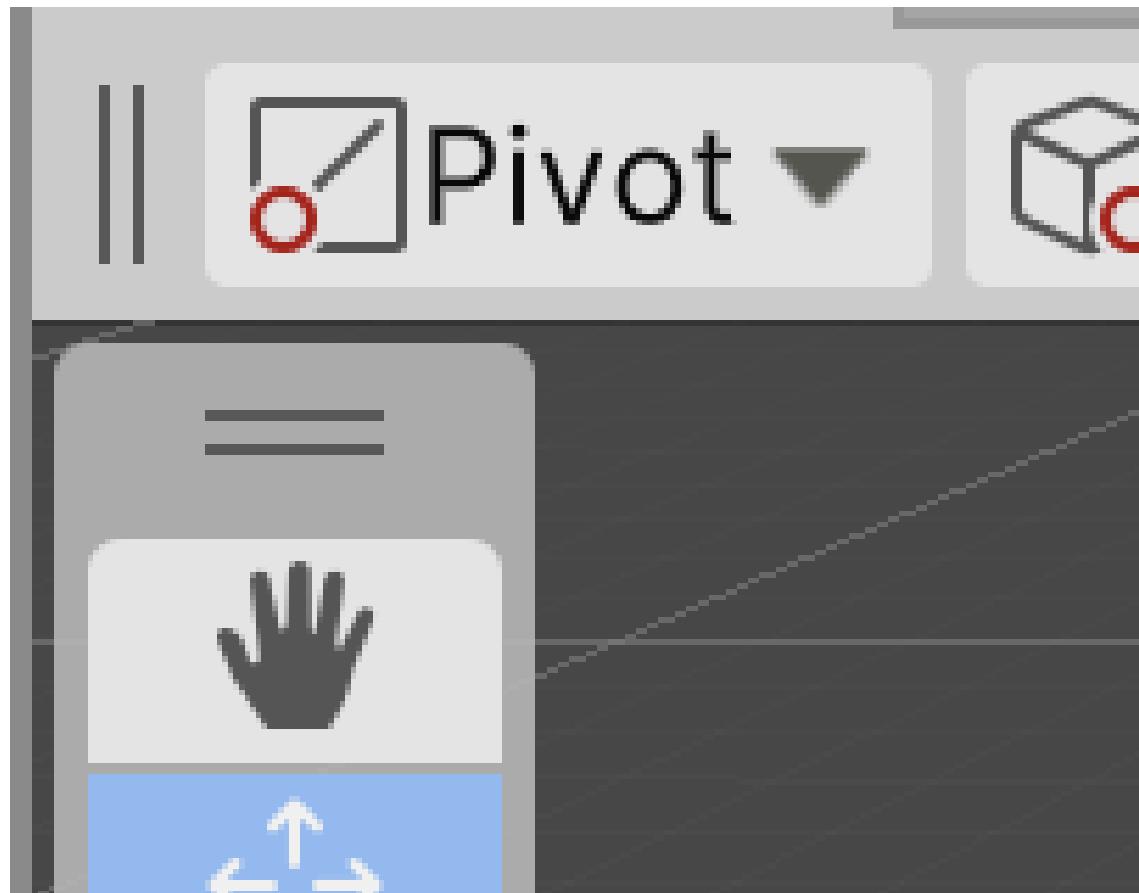


Figure 3.43: Enabling Wireframe mode

1. Enable the **Move** tool by pressing the second button (or the W key on the keyboard) in the top-left corner of the **Scene** panel:



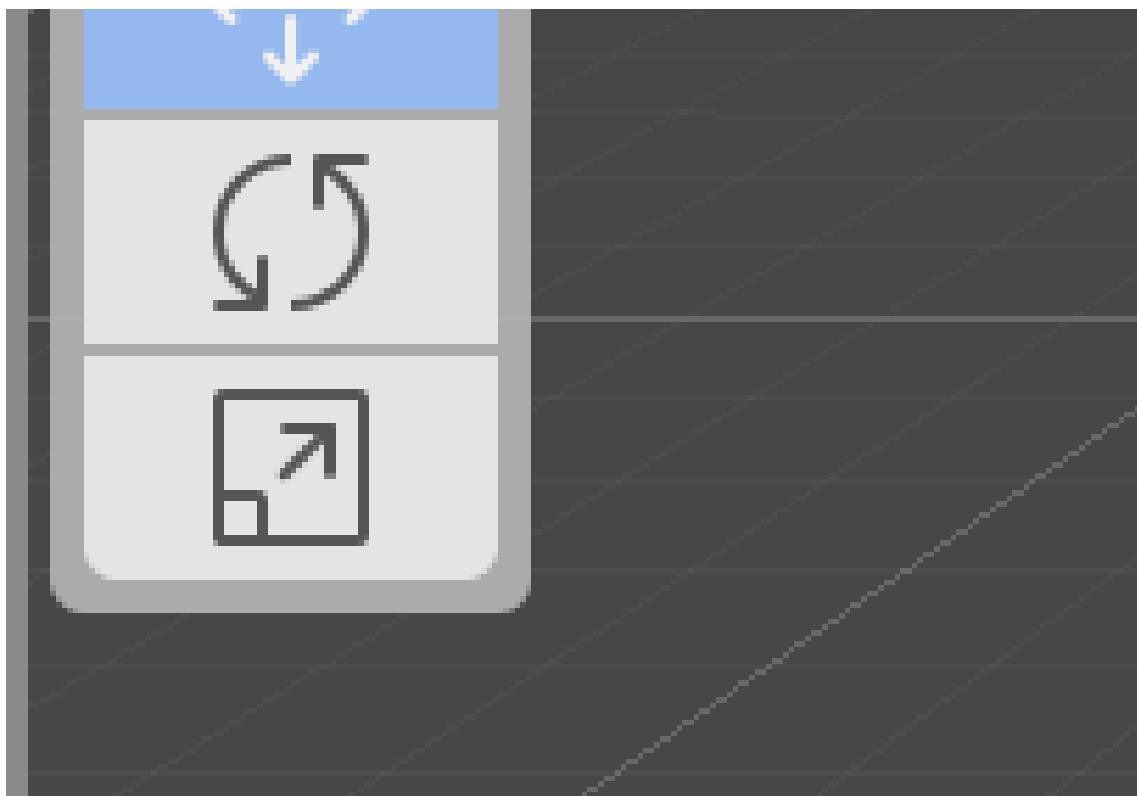


Figure 3.44: Object Move tool

2. Move the edges down until they are fully buried in the terrain:

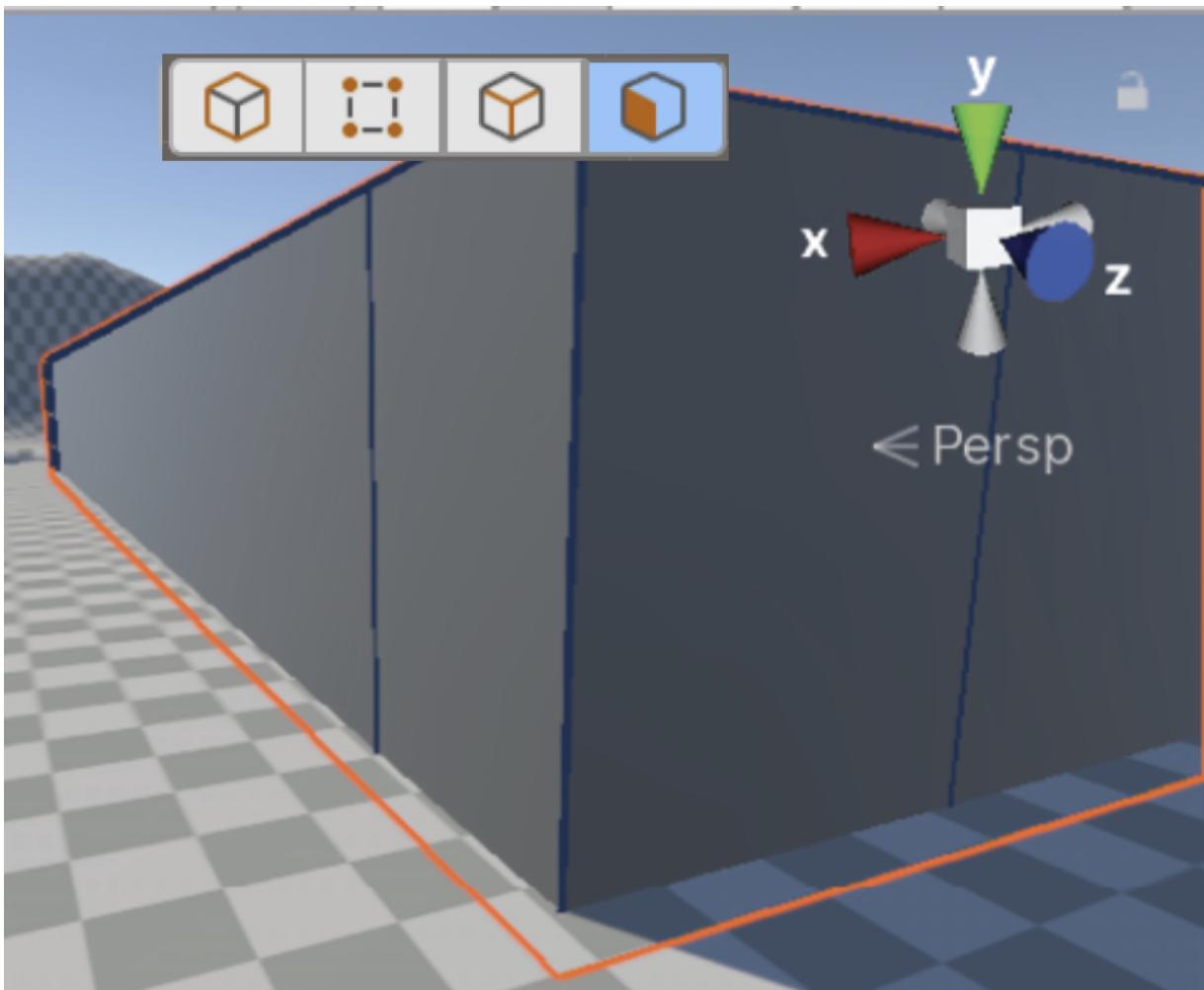


Figure 3.45: Overlapping faces

Now that we have a base mesh, we can start adding details to it using several other ProBuilder tools.

Memory

This section reminds me of the time I learned my first 3D authoring software, Maya. I was using a version before Autodesk acquired it, so imagine how old it was. I enjoyed learning the techniques to sculpt solids based on actual blueprints of the objects. I remember creating a Stargate and the F302 Tau'ri vessel between my first models. I'm in

my third run of the entire series (including Atlantis and Universe) as I write this.

Adding details

Let's start adding details to the base by applying a little bevel to the walls and a little cut in the corners so they are not so sharp. To do so, follow these steps:

1. Using the Edge Selection Tool (the third of the **ProBuilder** buttons), select the top edges of our model:

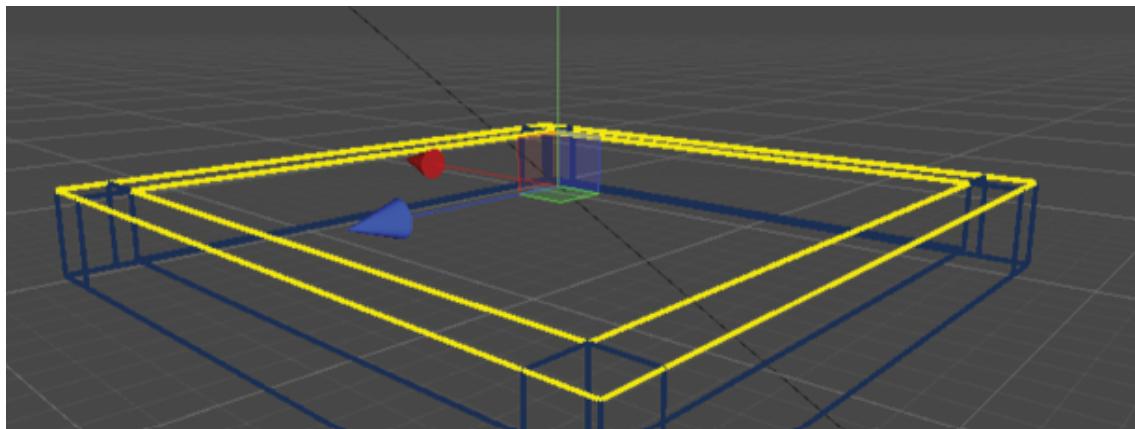


Figure 3.46: Top wall edges being selected

2. In the **ProBuilder** window, press the + icon to the right of the **Bevel** button.
3. Set a distance of 0.5 :

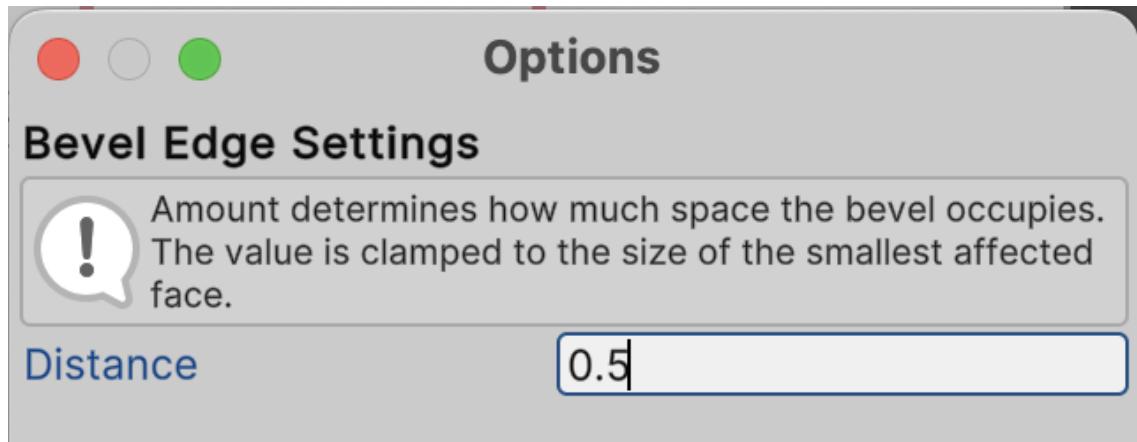


Figure 3.47: Bevel distance to generate

4. Click on **Bevel Edges**. Now you can see the top parts of our walls have a little bevel:



Figure 3.48: Result of the bevel process

5. Optionally, you can do that with the bottom part of the inner walls:

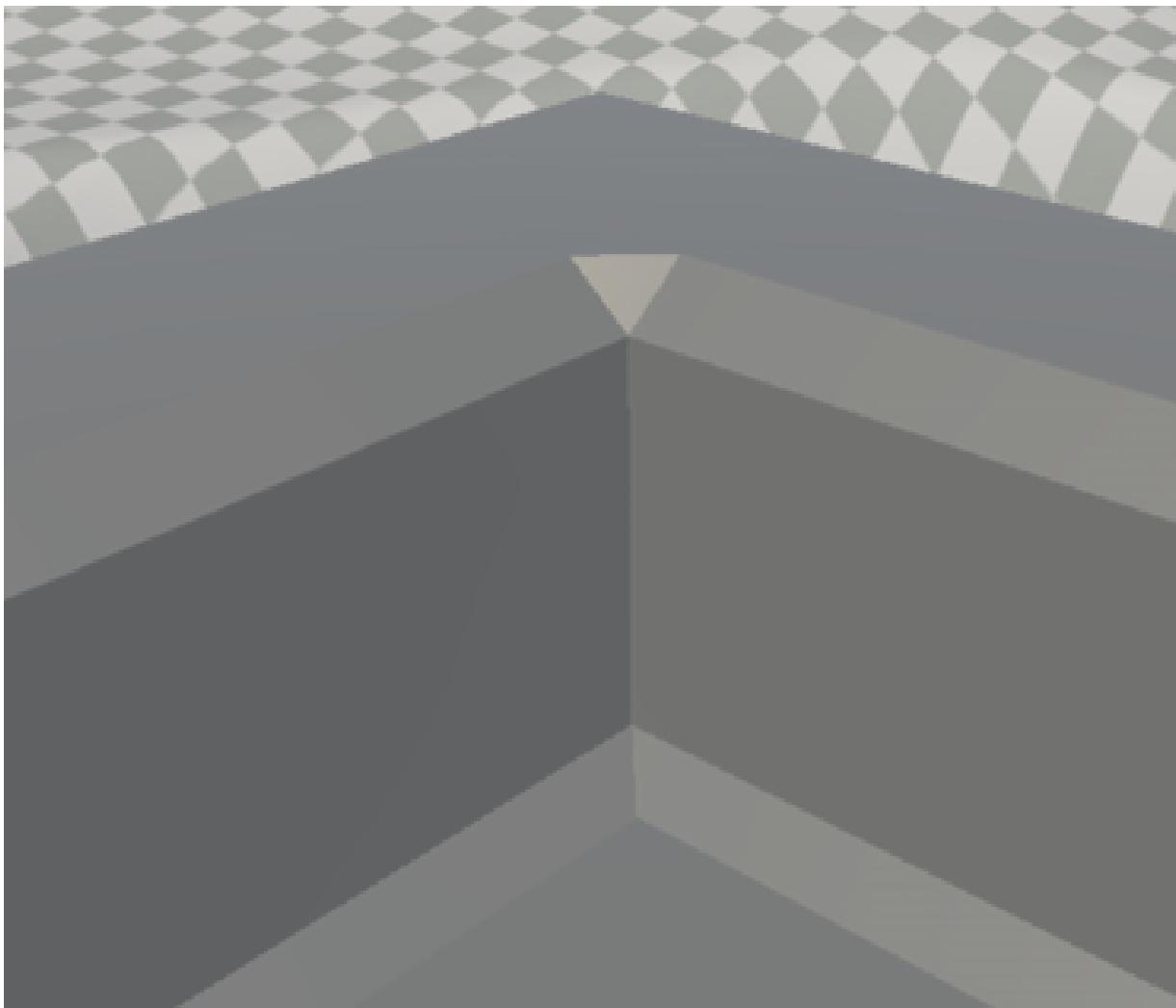


Figure 3.49: Bevel being applied to floor-wall edges

Another detail to add could be a pit in the middle of the ground as a hazard we need to avoid falling into and to make the enemies avoid it using AI. In order to do that, follow these steps:

1. Enable the **Face** selection mode by clicking the fourth **ProBuilder** Scene View button.
2. Select the floor.

3. Click the **Subdivide Faces** option in the **ProBuilder** window.
You will end up with the floor split into four.
4. Click that button again to end up with a 4 x 4 grid:

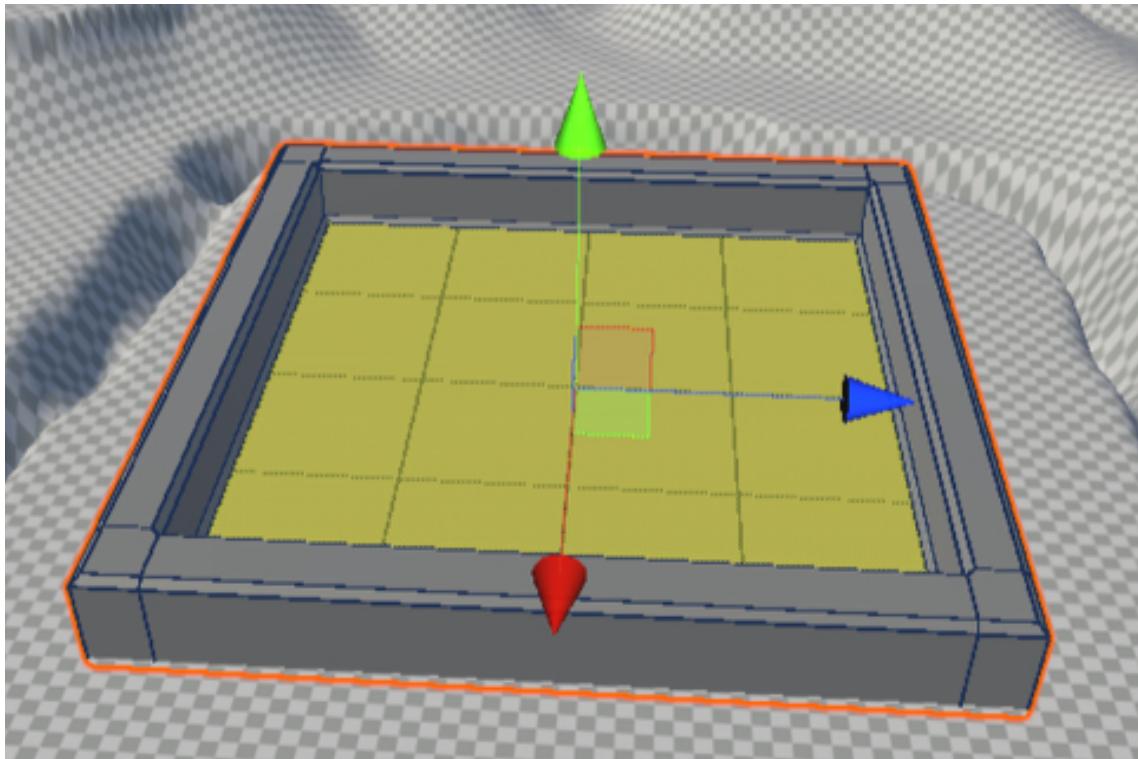


Figure 3.50: Subdividing the floor

5. Select the four inner floor tiles while holding Ctrl (Command on Mac) using the **Select Face** tool (the third of the ProBuilder buttons in the top part of the Scene View).
6. Enable the **Scale** tool by clicking the fourth button in the top-left part of the Scene View or pressing the R key on the keyboard. Make sure that the tool handle position is set to **Center** (and not to **Pivot**), so the object is scaled from the center of the object itself. As with the **Move** tool, this can be used to scale any object, not only vertices:

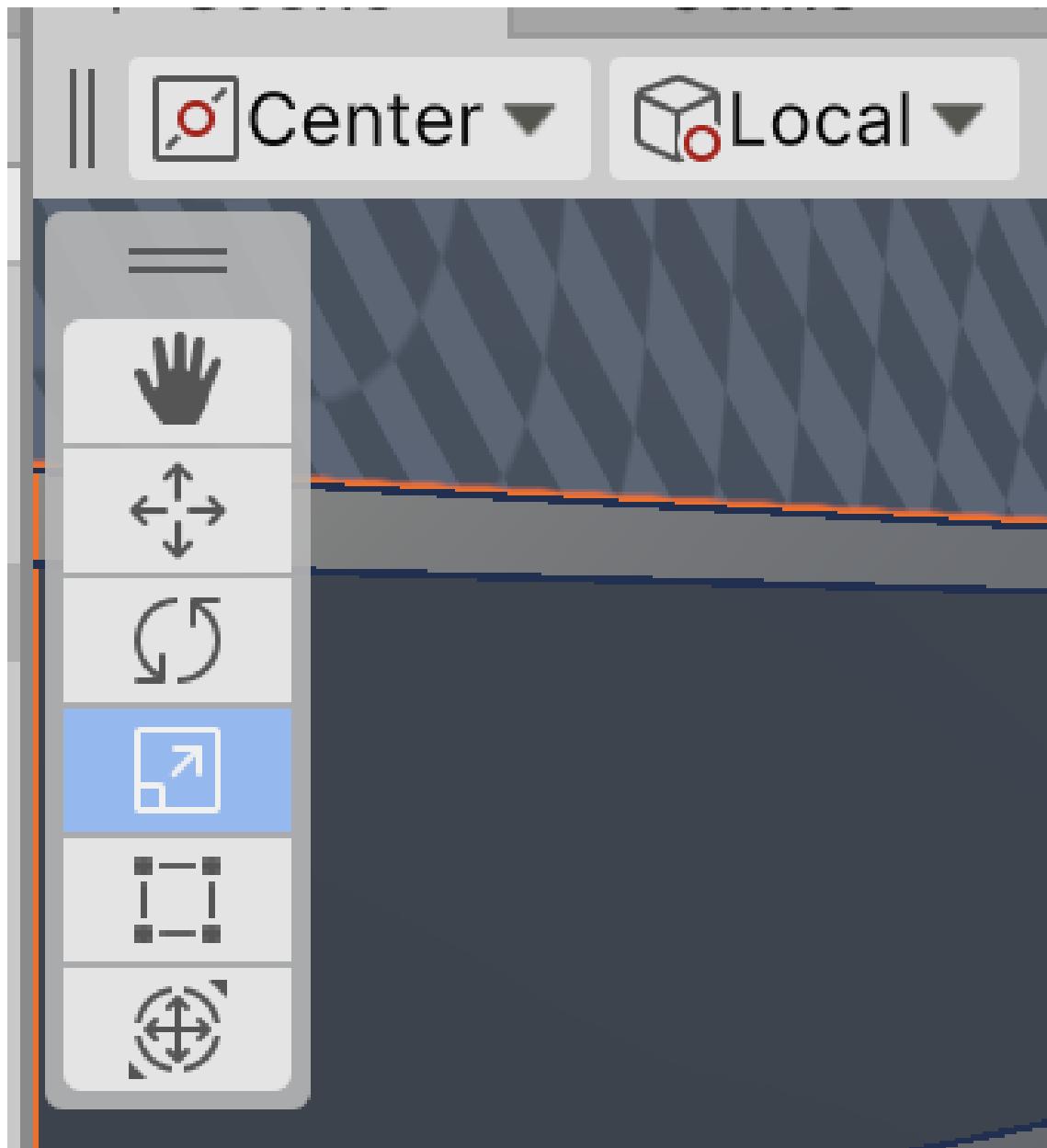


Figure 3.51: Scale tool

7. Using the gray cube at the center of the gizmo, scale down the center tiles:

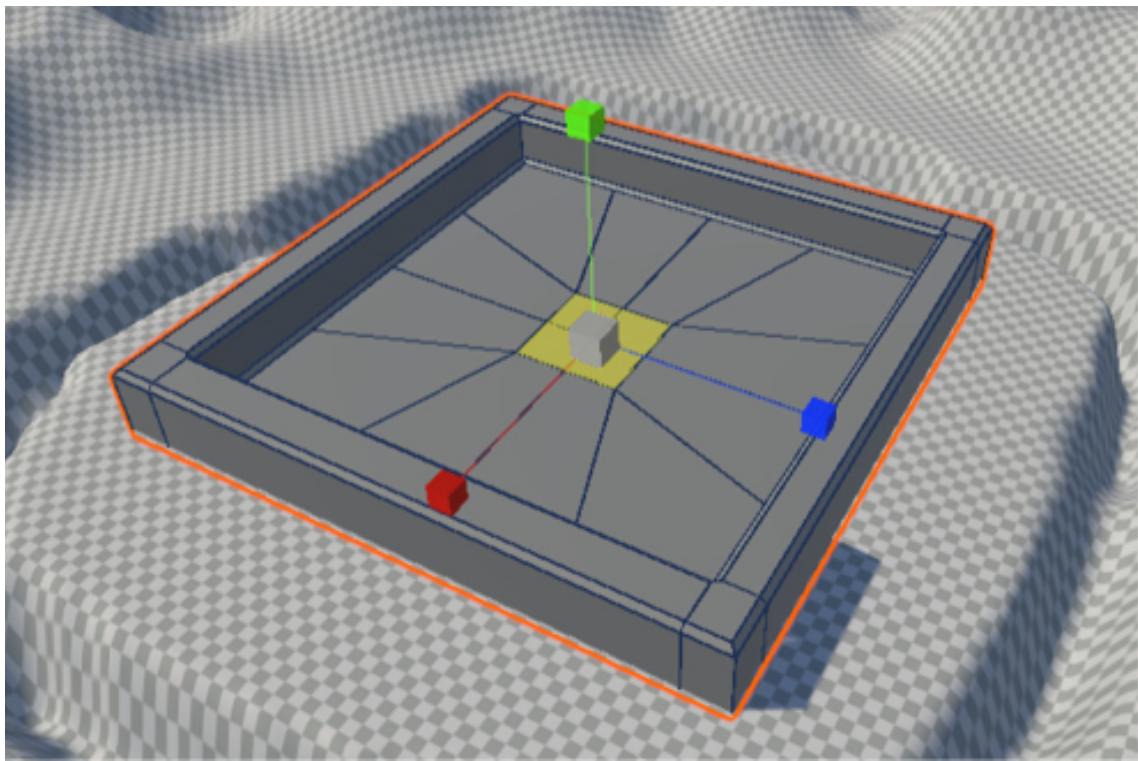


Figure 3.52: Inner cells being scaled down

8. Click the **Extrude Faces** button in the **ProBuilder** window.
9. Push the extruded faces downward with the **Move Tool**.
10. Right-click on the **ProBuilder** window tab and select **Close Tab**. We need to get back to terrain editing, and having **ProBuilder** open won't allow us to do that comfortably:

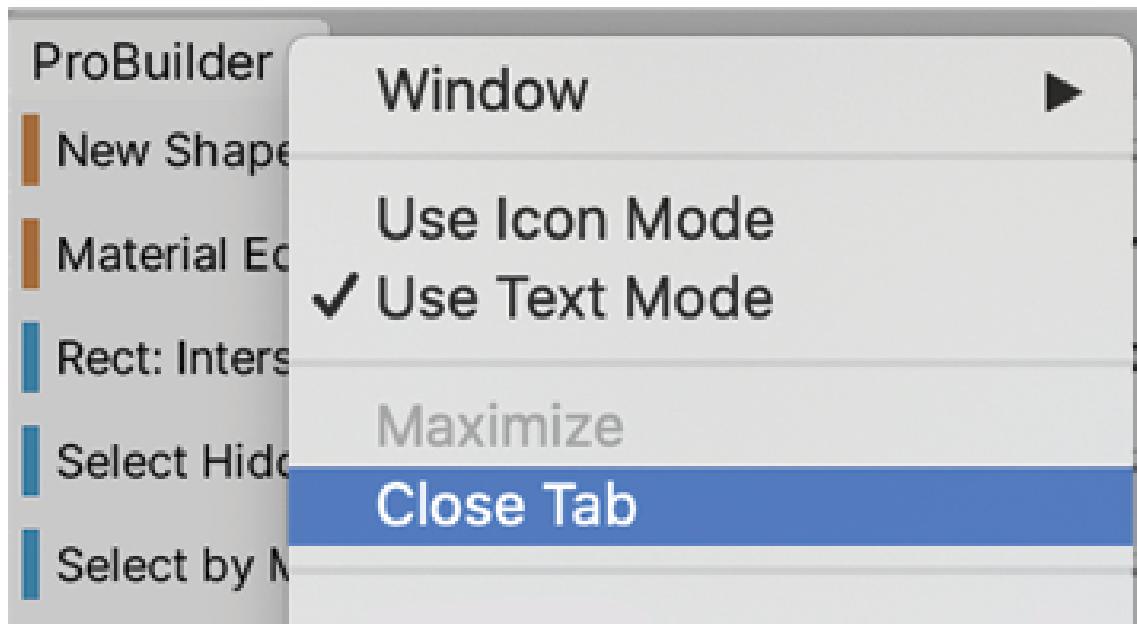


Figure 3.53: Close Tab option

11. Select the terrain and lower it so that we can see the pit:

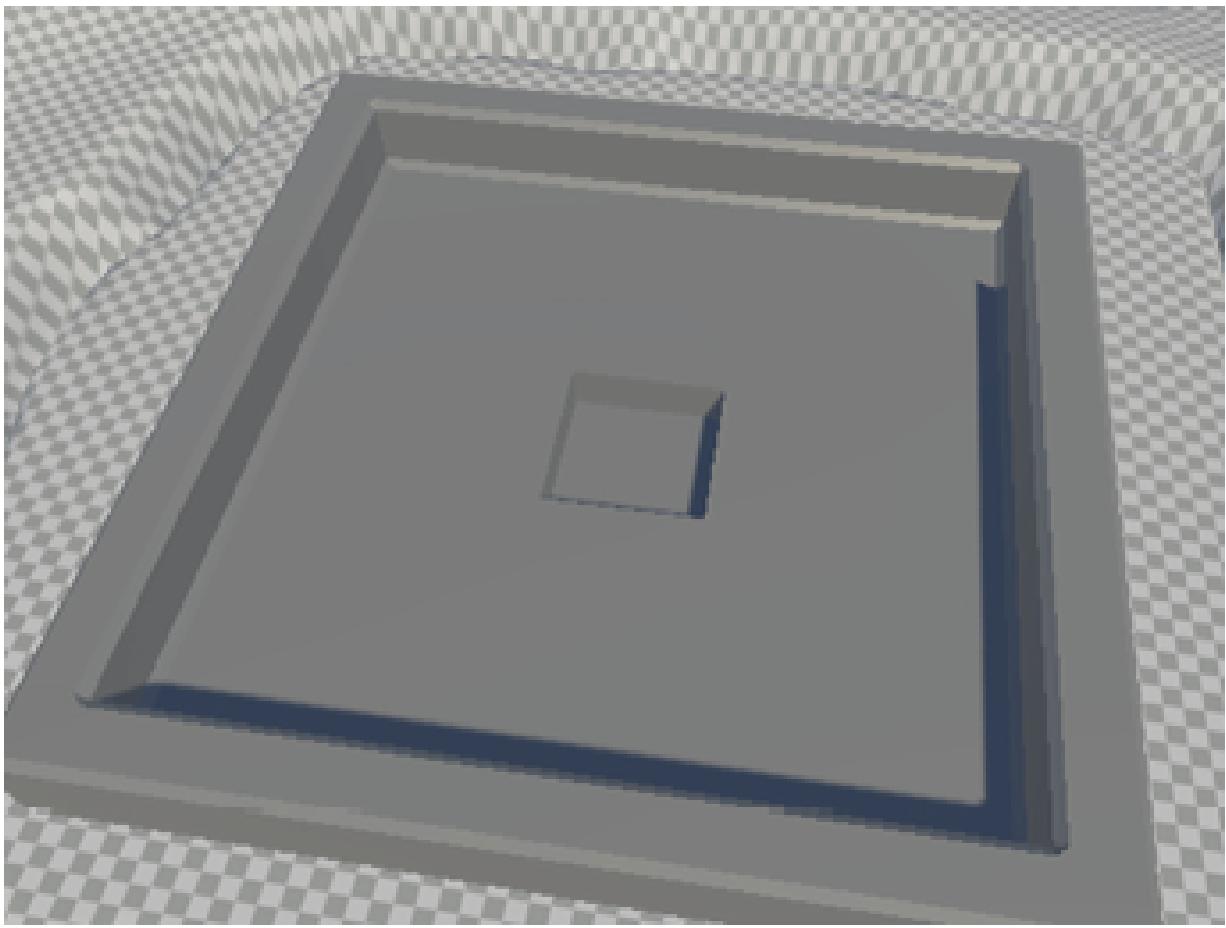


Figure 3.54: Terrain being lowered for the pit to be visible

With this, we have seen how to use different ProBuilder tools like Extrude and Bevel to create simple meshes to prototype the layout of our level. Sometimes, doing so is not easy with plain cubes.

Summary

In this chapter, we learned how to create large terrain meshes using Height Maps and Unity Terrain Tools such as **Paint Height** and **Set Height** to create hills and rivers. Also, we saw how to create our own 3D meshes using ProBuilder, as well as how to manipulate the vertices, edges, and faces of a model to create a prototype base model for our game. We didn't discuss any performance optimizations we can apply to our meshes or advanced 3D modeling

concepts as that would require entire chapters and is beyond the scope of this book. Right now, our main focus is prototyping, so we are fine with our level's current status. In the next chapter, we will learn how to download and replace these prototyping models with final art by integrating assets (files) we have created with external tools. This is the first step to improving the graphics quality of our game, which we will finish by the end of *Part 3, Improving Graphics*.

4 Seamless Integration: Importing and Integrating Assets

Join our book community on Discord

<https://packt.link/unitydev>



In the previous chapter, we created the prototype of our level. Now, let's suppose that we have coded the game and tested it, confirming the game idea is fun. With that, it's time to replace the prototype art with the real finished art. We are going to actually code the game in the next chapter, *Chapter 5, Unleashing the Power of C# and Visual Scripting*, but for learning purposes, let's just skip that part for now. In order to use the final assets, we need to learn how to get them (images, 3D models, and so on), how to import them into Unity, and how to integrate them into our scene. In this chapter, we will examine the following topics:

- Importing assets
- Integrating assets
- Configuring assets

Let's start by learning how to get assets in Unity, such as 3D models and textures.

Importing assets

We have different sources of assets we can use in our project. We can simply receive a file from our artist, download them from different free and paid asset sites, or we can use the **Asset Store**, Unity's official asset virtual store, where we can get free and paid assets ready to use with Unity. We will use a mix of downloading assets from the internet and from the Asset Store, just to use all possible resources. In this section, we will cover the following concepts related to importing assets:

- Importing assets from the internet
- Importing assets from the Asset Store
- Importing assets from Unity packages

Let's start by exploring the first source of assets, the internet.

Importing assets from the internet

In terms of getting art assets for our project, let's start with our terrain textures. Remember that we have our terrain painted with a grid pattern, so the idea is to replace that with grass, mud, rock, and other kinds of textures. To do that, we need images. In this case, these kinds of images are usually top-down views of different terrain patterns, and they have the requirement of being "tileable," meaning you can repeat them with no noticeable pattern in their connections. You can see an example of this in the following image:

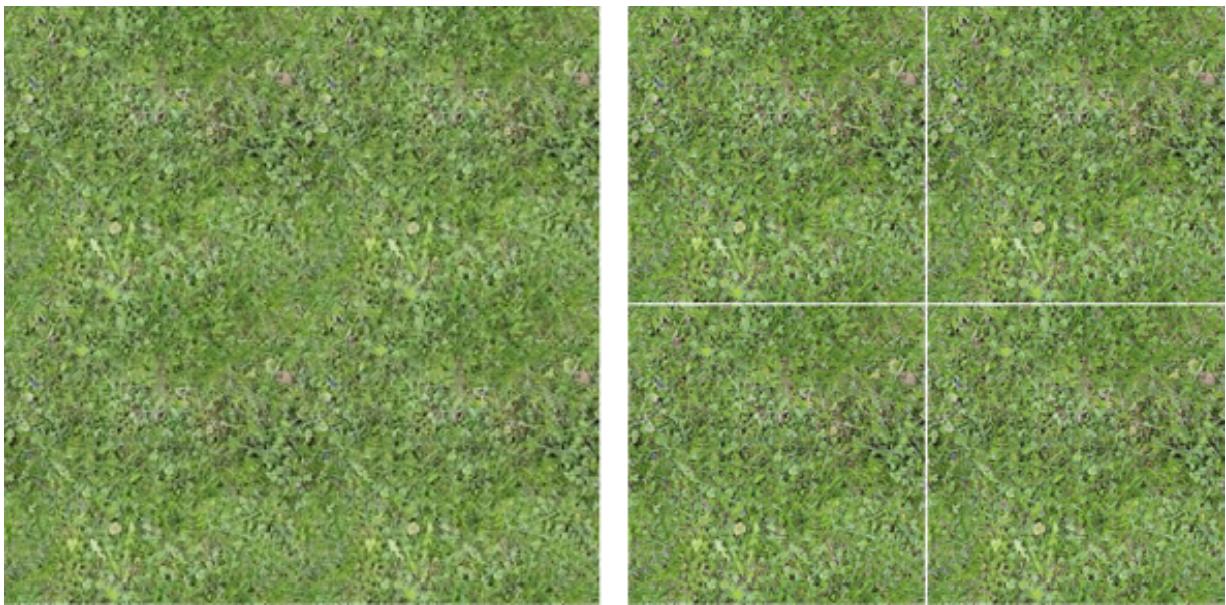


Figure 4.1: Left – grass patch; Right – the same grass patch separated to highlight the texture tiling

The grass on the left seems to be one single big image, but if you pay attention, you should be able to see some patterns repeating themselves. In this case, this grass is just a single image repeated four times in a grid, like the one on the right. This way, you can cover large areas by repeating a single small image, saving lots of RAM on the user's computer. The idea is to get these kinds of images to paint our terrain. You can get them from several places, but the easiest way is to use *Google Images* or any image search engine. Always check for copyright permissions before using something from these sources. Use the keywords “PATTERN seamless tileable texture” when searching for the texture, replacing “PATTERN” with the kind of terrain you are looking for, such as “grass tileable texture” or “mud tileable texture.” In this case, I would type “grass tileable texture.” Once you have downloaded the image, you can add it to your project in several ways. The simplest one would be doing the following:

1. Locate your image using **File Explorer (Finder on Mac)**.

2. Locate or create the `Textures` folder in the **Project** window in Unity.
3. Put **File Explorer** and the **Unity Project** window next to each other.
4. Drag the file from **File Explorer** to the `Textures` folder in the **Unity Project** window:

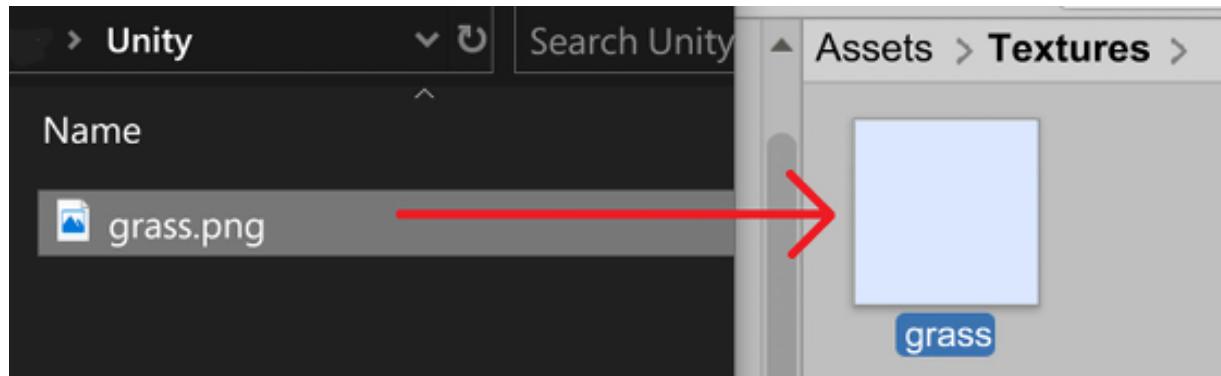


Figure 4.4: Texture being dragged from Windows File Explorer to Unity's Project view

For simple textures like the ones in the previous figure, any search engine can be helpful, but if you want to replace the player's base geometry with detailed walls and doors or place enemies in your scene, you need to get 3D models. If you search for those in any search engine using keywords such as “free zombie 3D model,” you will find endless free and paid 3D model sites such as TurboSquid and Mixamo, but those sites can be problematic because those meshes are usually not prepared for being used in Unity, or even games. You will find models with very high polygon counts, incorrect sizes or orientations, unoptimized textures, and so on. To prevent those problems, we'll want to use a better source, and in this case, we will use Unity's Asset Store, so let's explore it.

Importing assets from the Asset Store

The Asset Store is Unity's official asset marketplace where you can find lots of models, textures, sounds, and even entire Unity plugins

to extend the capabilities of the engine. In this case, we will limit ourselves to downloading 3D models to replace the player's base prototype. You will want to get 3D models with a modular design, meaning that you will get several pieces, such as walls, floors, corners, and so on. You can connect them to create any kind of scenario. In order to do that, you must follow these steps:

1. Click on **Window | Asset Store** in Unity, which will open your web browser on the site <https://assetstore.unity.com>. In previous versions of Unity, you could see the Asset Store directly inside the editor, but now, it is mandatory to open it in a regular web browser, so just click the **Search Online** button, which will open the site <https://assetstore.unity.com/> in your preferred browser. Also, you can check **Always open in browser from menu** to directly open the page whenever you click on **Window | Asset Store**:



Figure 4.5: Asset Store moved message

2. In the top menu, click on the **3D** category to browse 3D assets:

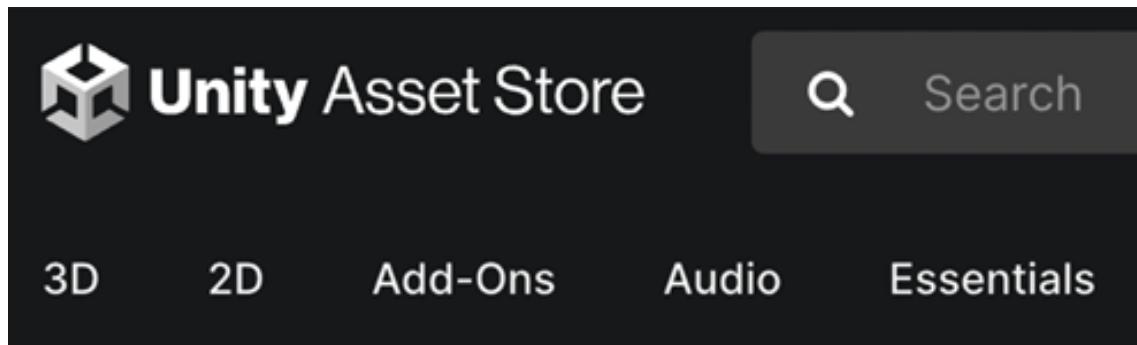


Figure 4.6: 3D assets menu

3. On the recently opened page, click the arrow to the right of the **3D** category in the **All Categories** panel on the right, and then open **Environments** and check the **Sci-Fi** mark, as we will make a future-themed game:

All Categories

- 3D (37191) ^
- Animations (723) ^
- Characters (10524) ▼
- Environments (7576) ^
- Dungeons (319)
- Fantasy (1128)

Figure 4.7: 3D assets menu

As you can see, there are several categories for finding different types of assets, and you can pick another one if you want to. In **Environments**, you will find 3D models that can be used to generate the scenery for your game.

1. If you need to, you can pay for an asset, but let's hide the paid ones for now. You can do that by checking the **Free Assets** checkbox from the **Pricing** dropdown on the right side:

Pricing

Free Assets (5)

Figure 4.8: Free Assets option

2. In the search area, find any asset that seems to have the aesthetic you are looking for and click it. Remember to look out for outdoor assets, because most environment packs are usually interiors only. In my case, I have picked one called **Sci-Fi Styled Modular Pack**, which serves both interiors and exteriors. Take into account that that package might not exist by the time you are reading this, so you might need to choose another one. If you don't find a suitable package, you can download and pick the asset files we used in the GitHub repository at
3. <https://github.com/PacktPublishing/Hands-On-Unity-2023-Game-Development-Fourth-Edition>.

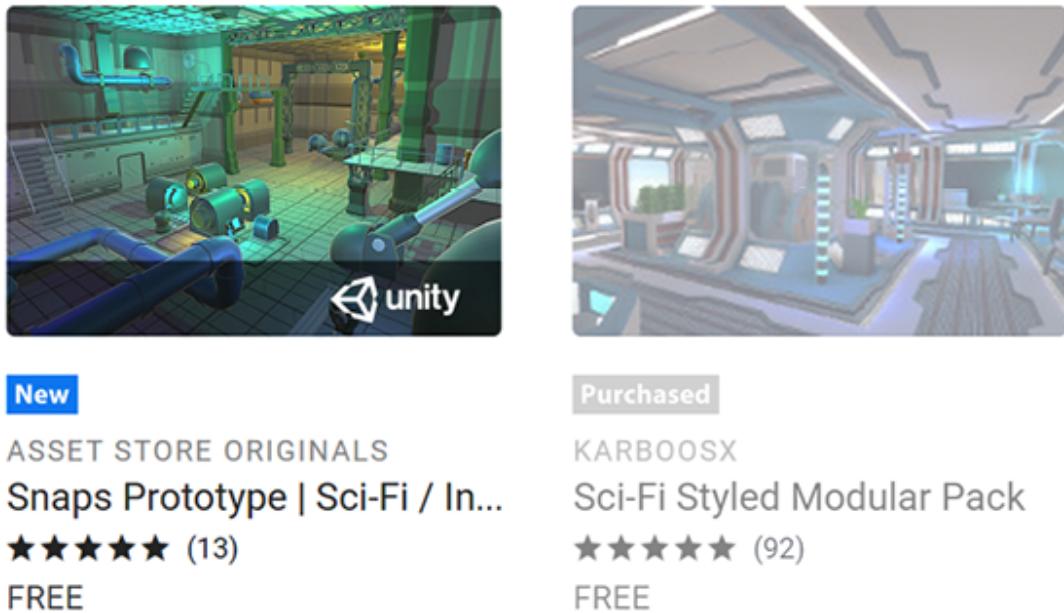


Figure 4.9: Preview of Asset Store searched packages

4. Now, you will see the package details in the **Asset Store** window. Here, you can find information regarding the package's description, videos/images, the package's contents, and the most important part – the reviews, where you can see whether the package is worth getting:

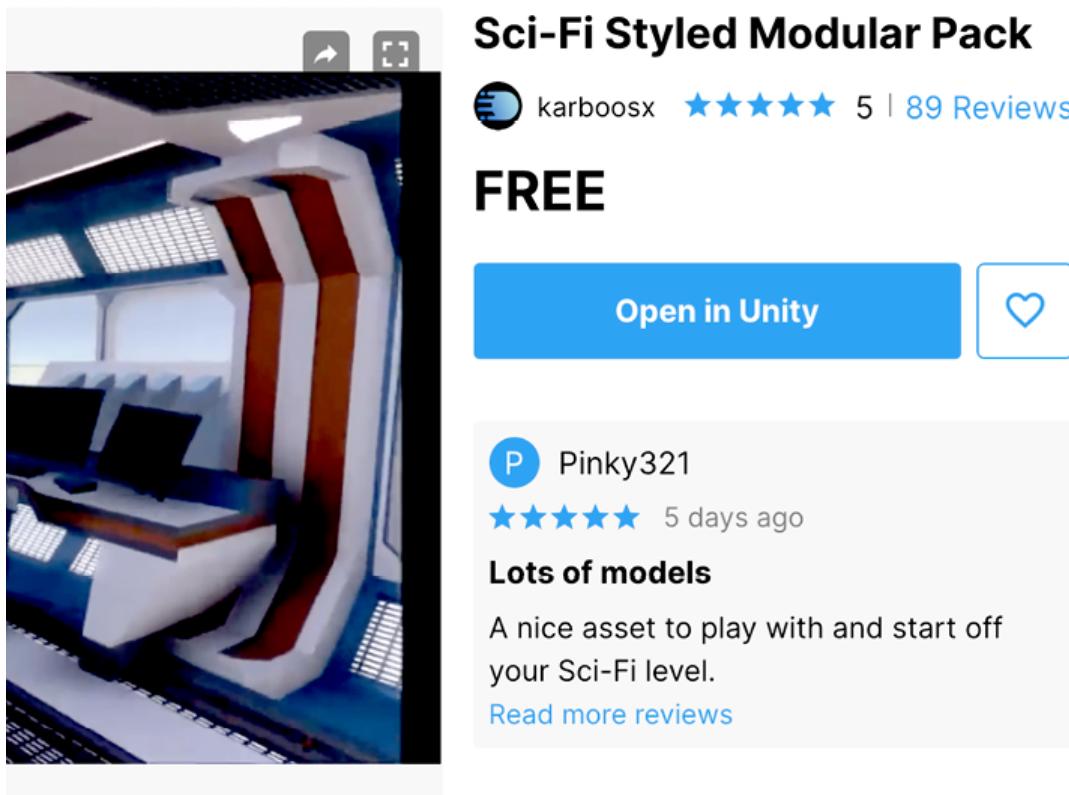


Figure 4.10: Asset Store package details

5. If you are OK with this package, click the **Add To My Assets** button, log in to Unity if requested, and then click the **Open In Unity** button. You might be prompted to accept the browser to open Unity; if so, just accept:

Open Unity.app?

<https://assetstore.unity.com> wants to open this application.

Always allow assetstore.unity.com to open links of this type in the associated app

[Cancel](#) [Open Unity.app](#)

Figure 4.11: Switching apps

6. This will open the **Package Manager** again, but this time in the **My Assets** mode, you should see a list of all assets you have ever downloaded from the Asset Store, and the one you just selected in the list:

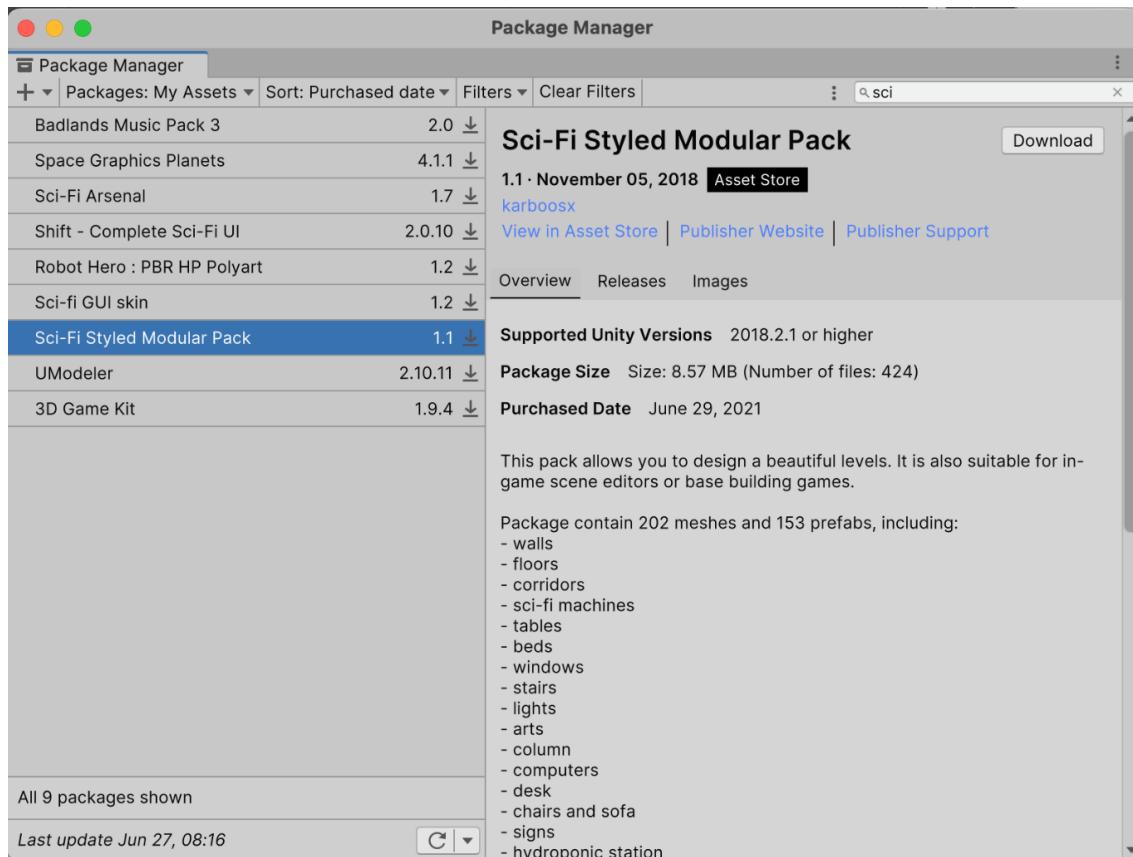


Figure 4.12: Package Manager showing assets

7. Click on Download in the bottom-right part of the window and wait for it to finish. Then hit Import.
8. After a while, the Package Contents window will show up, allowing you to select exactly which assets of the package you want in your project. For now, leave it as is and click Import:

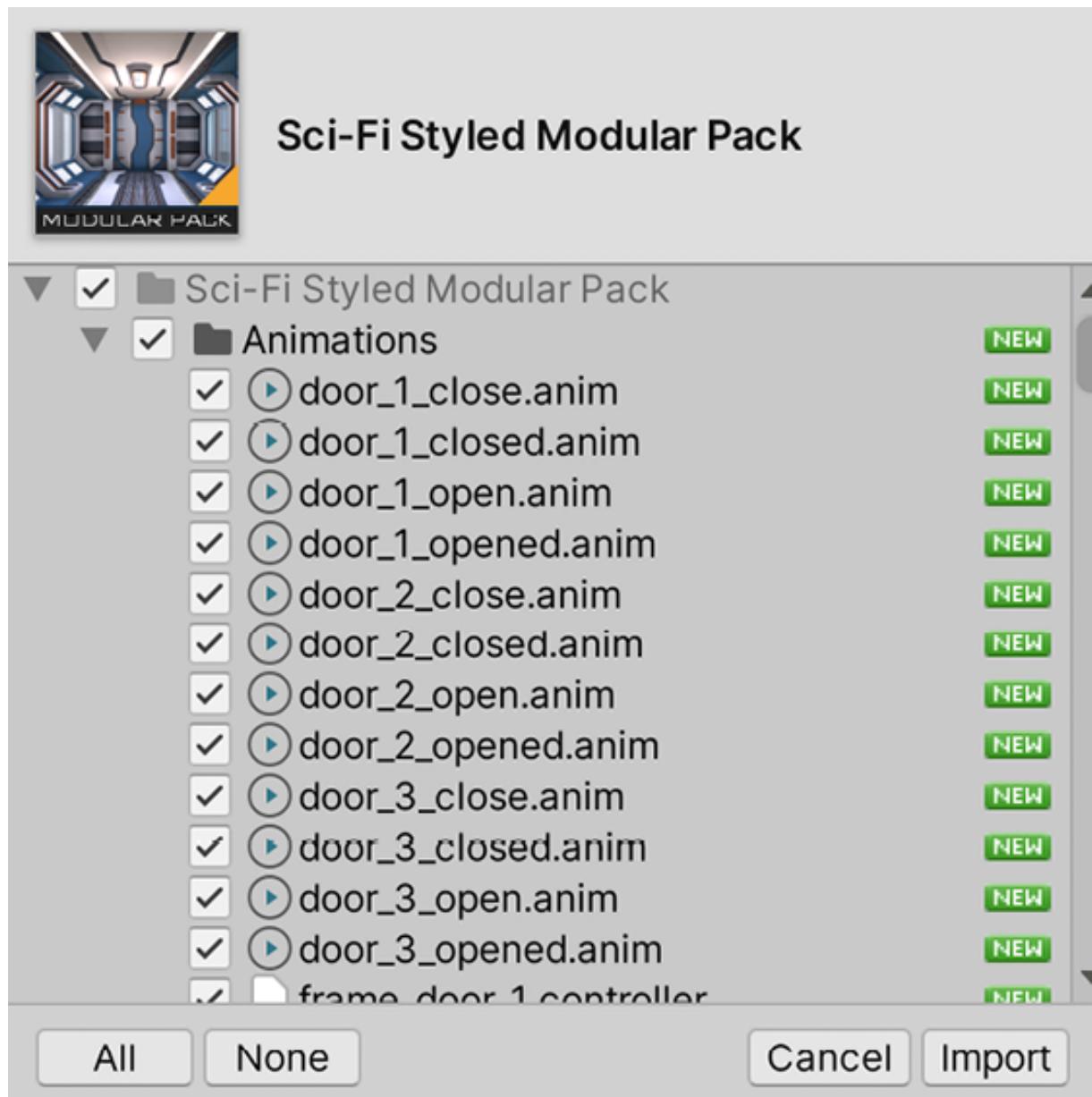


Figure 4.13: Assets to import selection

After some importing time, you will see all the package files in your **Project** window.

Memory

Having something like the Asset Store is really a great help. When I worked on other engines or lesser-known game development frameworks, getting content for the game was

a challenge. There are tons of pages to get 3D models and textures, but they were not always optimized for video games, or even compatible with Unity. Of course, it is still a great skill to know what to do if the Asset Store doesn't have what you need, so I recommend you also explore other possible sources of assets and see the kind of challenges you will face working with those.

Take into account that importing lots of full packages will increase your project's size considerably, and that, later, you will probably want to remove the assets that you didn't use. Also, if you import assets that generate errors that prevent you from playing the scene, just delete all the `.cs` files that come with the package. They are usually in folders called `Scripts`. Those are code files that might not be compatible with your Unity version:

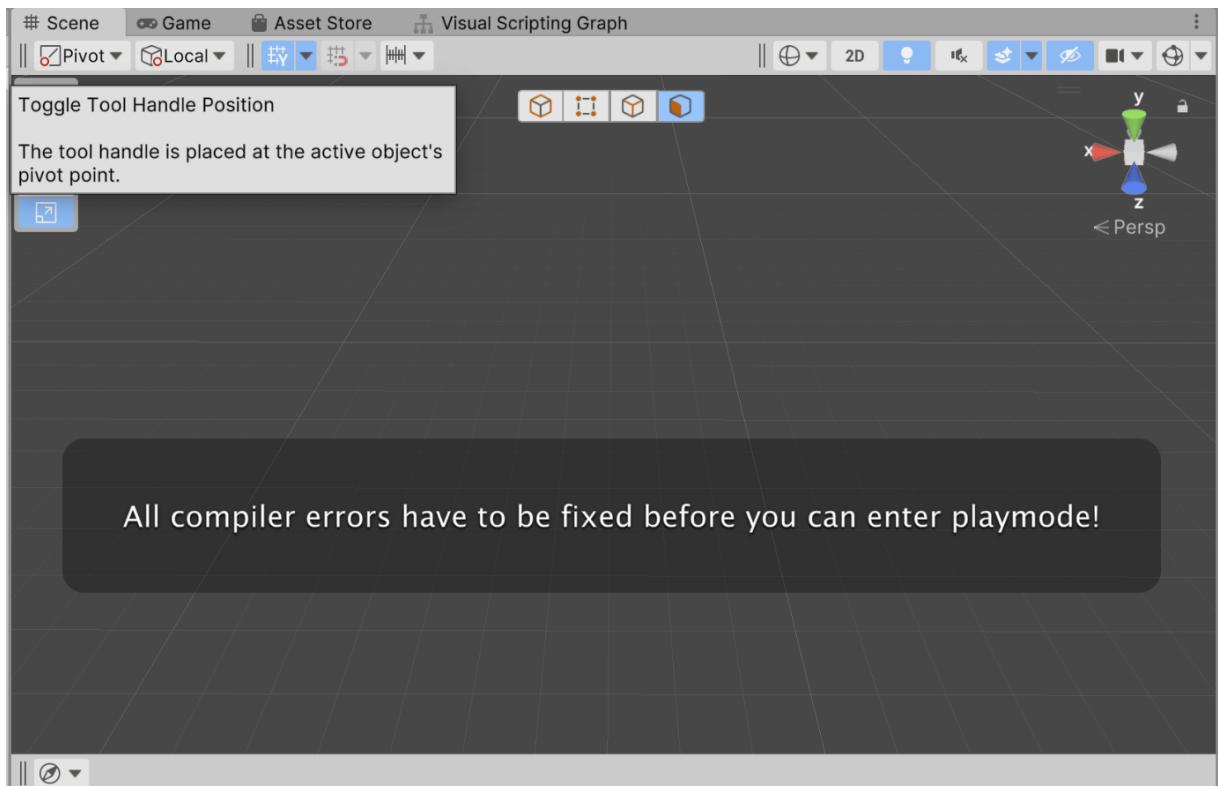


Figure 4.14: Code error warning when hitting play

Before you continue with this chapter, try to download a character 3D model using the Asset Store, following the previous steps. In order to do this, you must complete the same steps as we did with the level environment pack but look in the **3D | Characters | Humanoid** category of the Asset Store. In my case, I picked the **Robot Hero: PBR HP Polyart** package:



DUNGEON MASON
Robot Hero : PBR HP Poly...
★★★★★ (43) | ❤ (2664)
FREE [Add to My Assets](#)

Figure 4.15: Character package used in our game

Now, let's explore yet another source of Unity Assets: **Unity packages**.

Importing assets from Unity packages

The Asset Store is not the only source of asset packages; you can get .unitypackage files from the internet, or maybe from a coworker who wants to share assets with you.

If you want to create your own asset packages to share your assets with other developers, check the documentation at <https://docs.unity3d.com/Manual/AssetPackagesCreate.html>

In order to import a .unitypackage file, you need to do the following:

1. Go to the Assets | Import Package | Custom Package option:

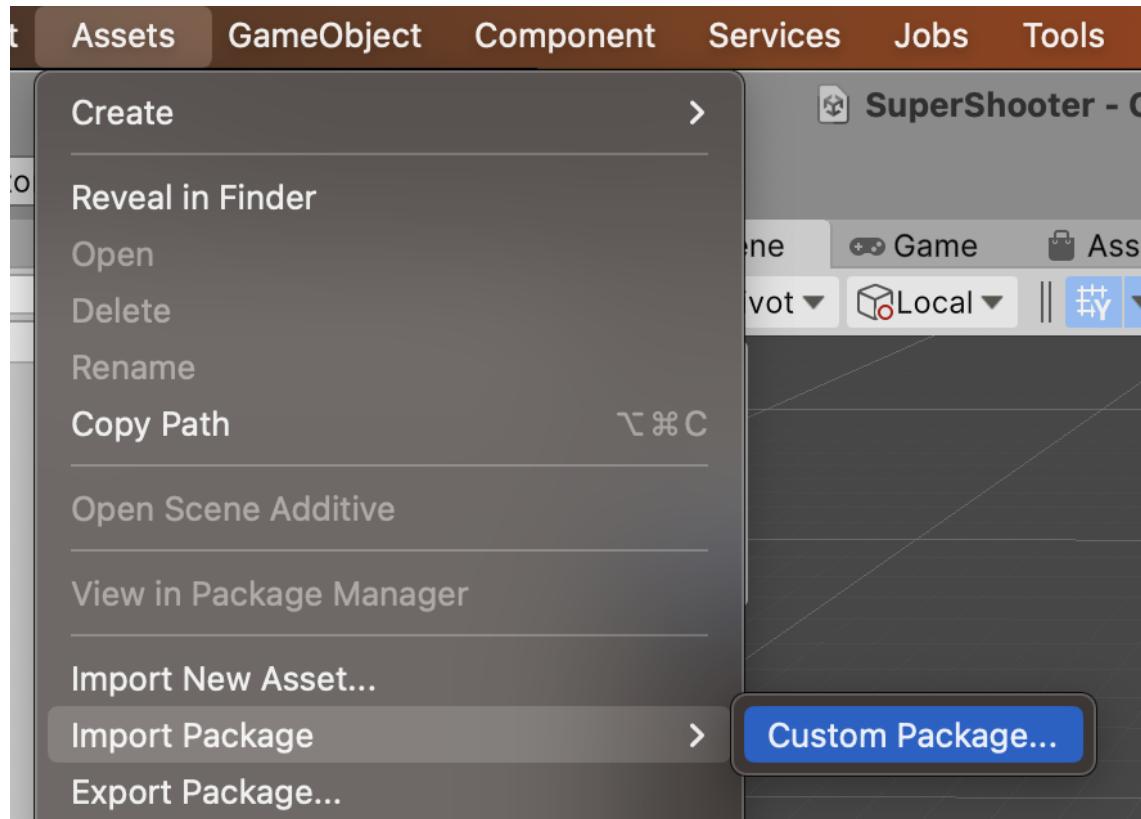


Figure 4.16: Importing custom packages

2. Search for the .unitypackage file in the displayed dialog.
3. Click the **Import** option in the **Import Unity Package** window that appears – the one we saw earlier, in the Asset

Store section.

Now that we have imported lots of art assets, let's learn how to use them in our scene.

Integrating assets

We have just imported lots of files that can be used in several ways, so the idea of this section is to see how Unity integrates those assets with the GameObjects and components that need them. In this section, we will cover the following concepts related to importing assets:

- Integrating terrain textures
- Integrating meshes
- Integrating materials

Let's start by using tileable textures to cover the terrain.

Integrating terrain textures

In order to apply textures to our terrain, do the following:

1. Select the **Terrain** object.
2. In the **Inspector**, click the brush icon of the **Terrain** component (second button).
3. From the drop-down menu, select **Paint Texture**:

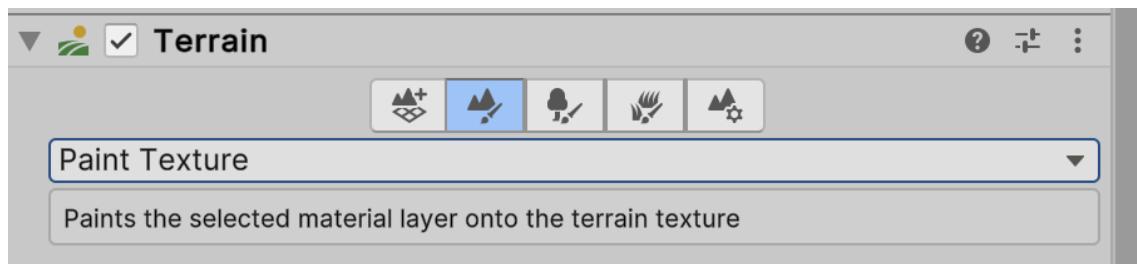
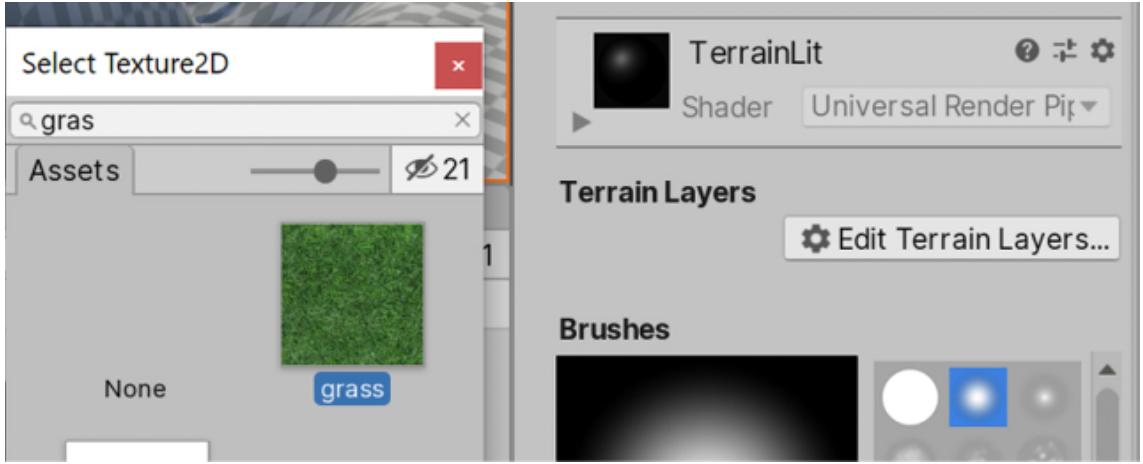


Figure 4.17: Terrain Paint Texture option

4. Click the Edit Terrain Layers... | Create Layer option.
 5. Find and double-click the terrain texture you downloaded previously in the Texture Picker window that appears:
- 
- Figure 4.18: Texture to paint picker*
6. You will see how the texture will be immediately applied to the whole terrain.
 7. Repeat steps 4 and 5 to add other textures. This time, you will see that that texture is not immediately applied.
 8. In the **Terrain Layers** section, select the new texture you have created to start painting with that. I used a mud texture in my case.
 9. Just like when you edited the terrain, in the **Brushes** section, you can select and configure a brush to paint the terrain.
 10. In the **Scene** view, paint the areas you want to have that texture applied to.
 11. If your texture patterns are too obvious, open the **New Layer N** section on top of the **Brushes** section, where N is a number that depends on the layer you have created. Each time you add a texture to the terrain, you will see that a new asset called **New Layer N** is created in the **Project** view. It holds data on the terrain layer you have created, and you can use it on other terrains if you need to. You can also rename that asset to give it

a meaningful name or reorganize those assets in their own folder for organization purposes.

12. Open the section using the triangle to its left and increase the **Size** property in the **Tiling Settings** section until you find a suitable size where the pattern is not that obvious:

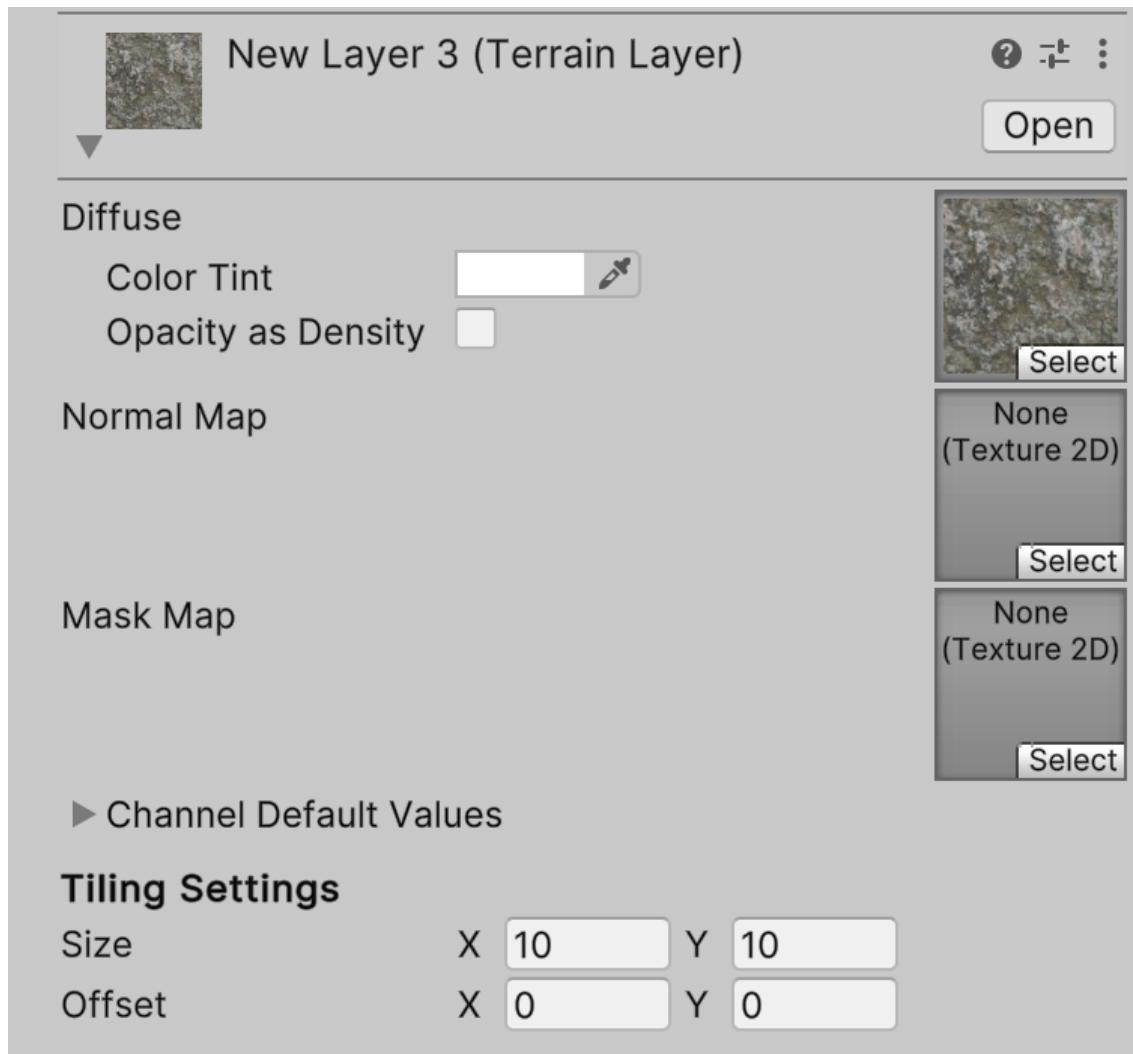


Figure 4.19: Painting texture options

13. Repeat steps 4 to 12 until you have applied all the textures you wanted to add to your terrain. In my case, I've applied the mud texture to the river basin and used a rock texture for the hills. For the texture of the rocks, I reduced the opacity property of

the brush to blend it better with the grass in the mountains. You can try to add a layer of snow at the top just for fun:

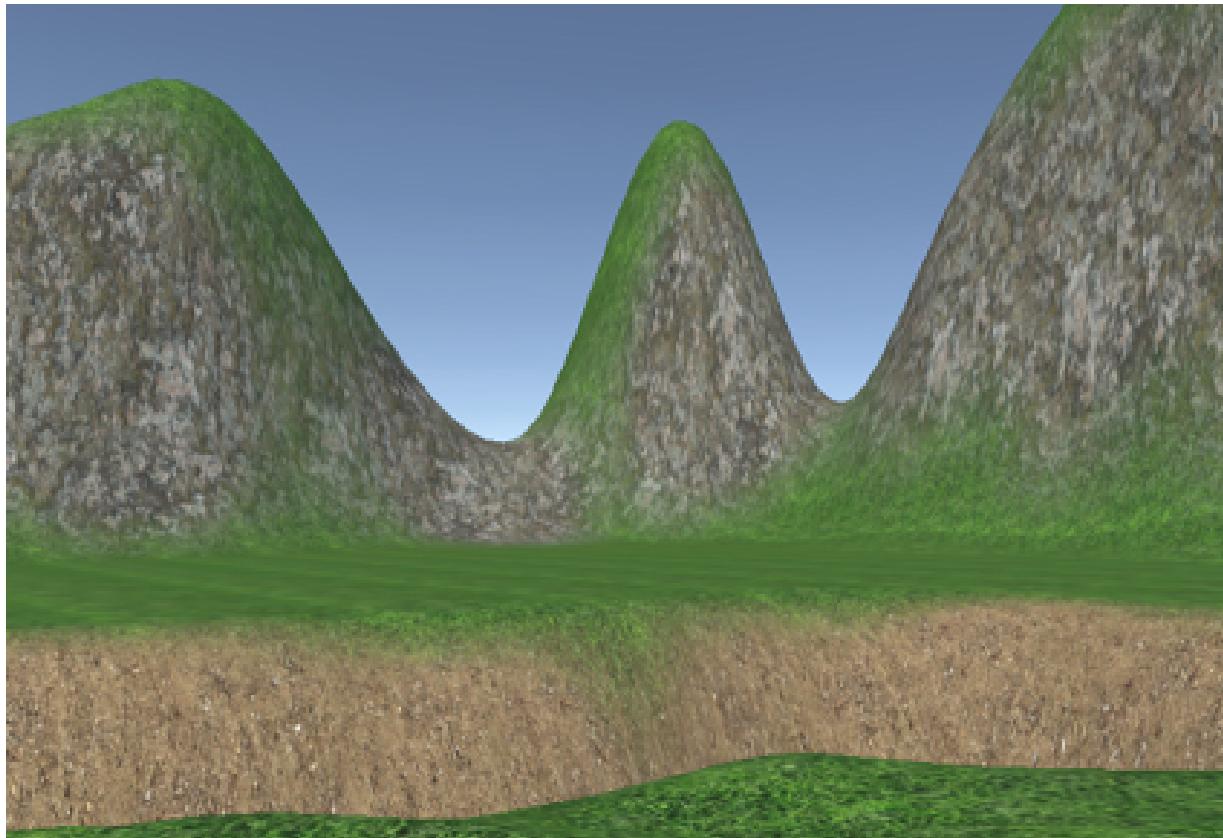


Figure 4.20: Results of painting our terrain with three different textures

Of course, we can improve this significantly using several of the advanced tools of the system, but let's just keep things simple for now. Next, let's see how we can integrate the 3D models into our game.

Integrating meshes

If you select one of the 3D assets we downloaded previously and click the arrow to its right, one or more sub-assets will appear in the **Project** window. This means that the 3D model files we

downloaded from the Asset Store (the FBX files) are containers of assets that define the 3D model:

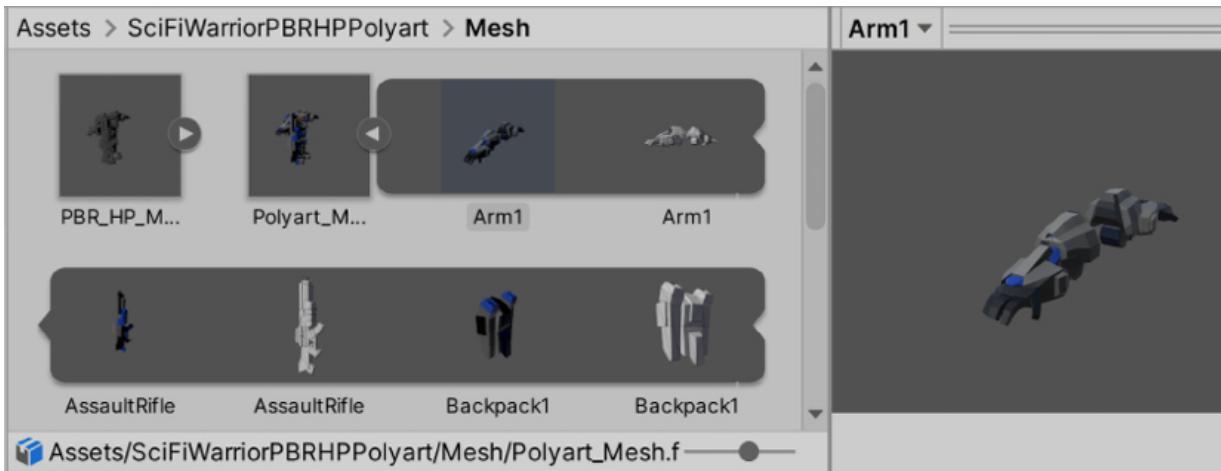


Figure 4.21: Mesh picker

Some of those sub-assets are meshes, which are a collection of triangles that define the geometry of your model. You can find at least one of these mesh sub-assets inside the file, but you can also find several, and that can happen if your model is composed of lots of pieces. For example, a car can be a single rigid mesh, but that won't allow you to rotate its wheels or open its doors; it will be just a static car, and that can be enough if the car is only a prop in the scene, but if the player will be able to control it, you will probably need to modify it. The idea is that all pieces of your car are different GameObjects parented to one another in such a way that if you move one, all of them will move, but you can still rotate its pieces independently. When you drag the 3D model file to the scene (not the sub-asset), Unity will automatically create all the objects for each piece and its proper parenting based on how the artist created those. You can select the object in the Hierarchy and explore all its children to see this:



Figure 4.22: Sub-object selection

Also, you will find that each of those objects may have its own `Mesh Filter` and `Mesh Renderer` components, each one rendering just that piece of the model. Remember that the `Mesh Filter` is a component that has a reference to the mesh asset to render, so the `Mesh Filter` is the one using those mesh sub-assets we talked about previously. In the case of animated characters, you will find the `Skinned Mesh Renderer` component instead, but we will discuss that component later, in *Part 3, Improving Graphics*. Now, when you drag the 3D model file into the scene, you will get a similar result as if the model were a Prefab and you were instancing it, but 3D model files are more limited than Prefabs because you can't apply changes to the model. If you've dragged the object onto the scene and edited it to have the behavior you want, I suggest that you create a Prefab to get all the benefits we discussed in *Chapter 2, Crafting Scenes and Game Elements*, such as applying changes to all the instances of the Prefab and so on. Never create lots of instances of a model from its model file—always create them from the Prefab you created based on that file to allow you to add extra behavior to it. That's the basic usage of 3D meshes. Now, let's explore the texture integration process, which will give our 3D models more detail.

Integrating textures

Maybe your model already has the texture applied but has a magenta color applied to all of it. If this is the case, that means the asset wasn't prepared to work with the **Universal Render Pipeline (URP)** template you selected when creating the project. Some assets in the Asset Store are created by third-party editors and could be meant to be used in older versions of Unity:

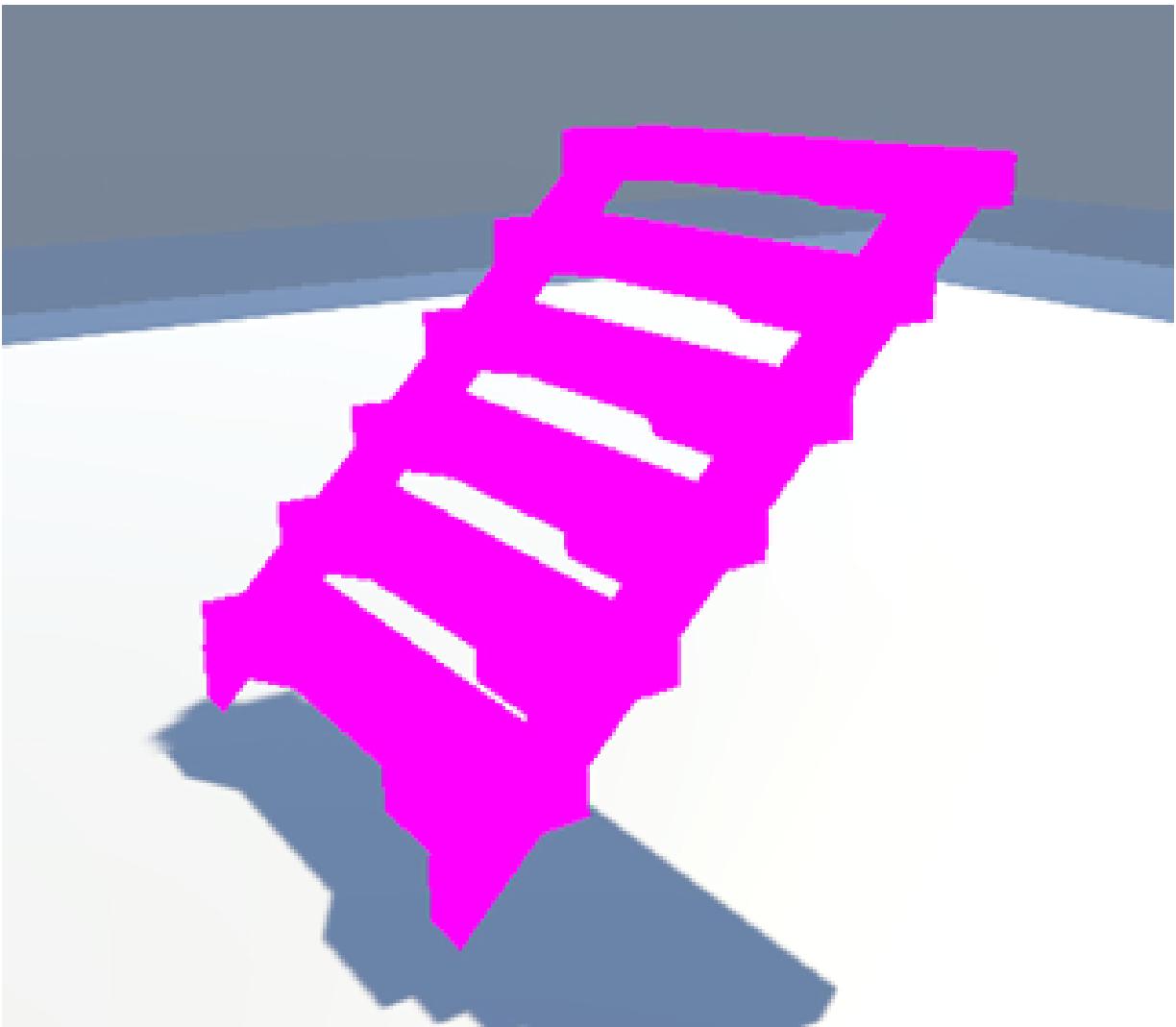


Figure 4.23: Mesh being rendered with erroneous material or no material at all

One option to fix magenta assets is using the **Render Pipeline Converter**, a tool that will find them and reconfigure them (if possible) to work with URP. To do so, perform the following steps every time you import an asset that looks magenta:

1. Go to Window | Rendering | Render Pipeline Converter.
2. Select the Built-in to URP option from the dropdown:

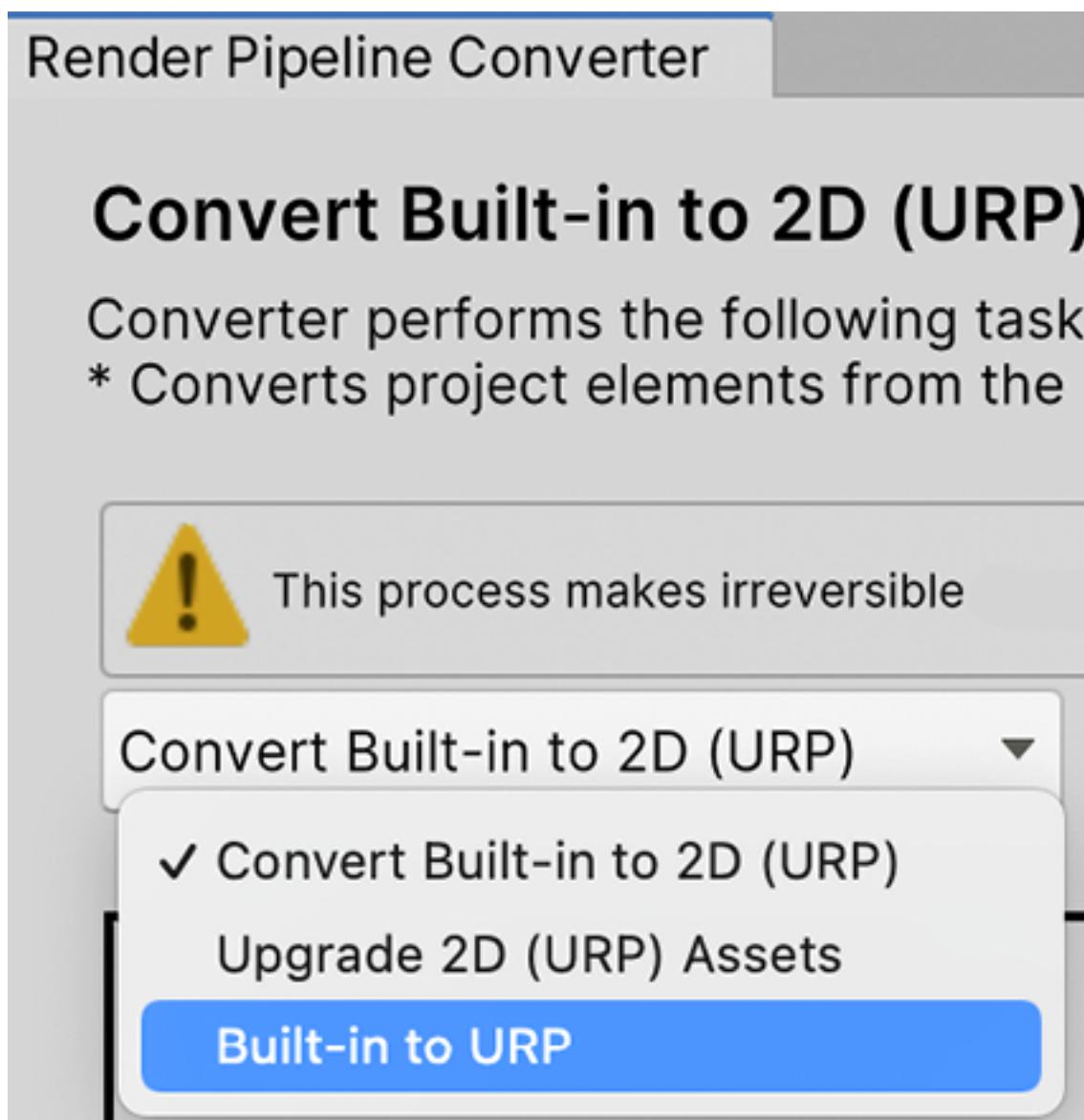


Figure 4.24: Upgrading older assets to URP

3. Scroll until you see the **Material Upgrade** checkbox and check it.
4. Click the **Initialize Converters** button in the bottom-left corner. This will display a list of all the materials that need to be upgraded. We will discuss materials more later:

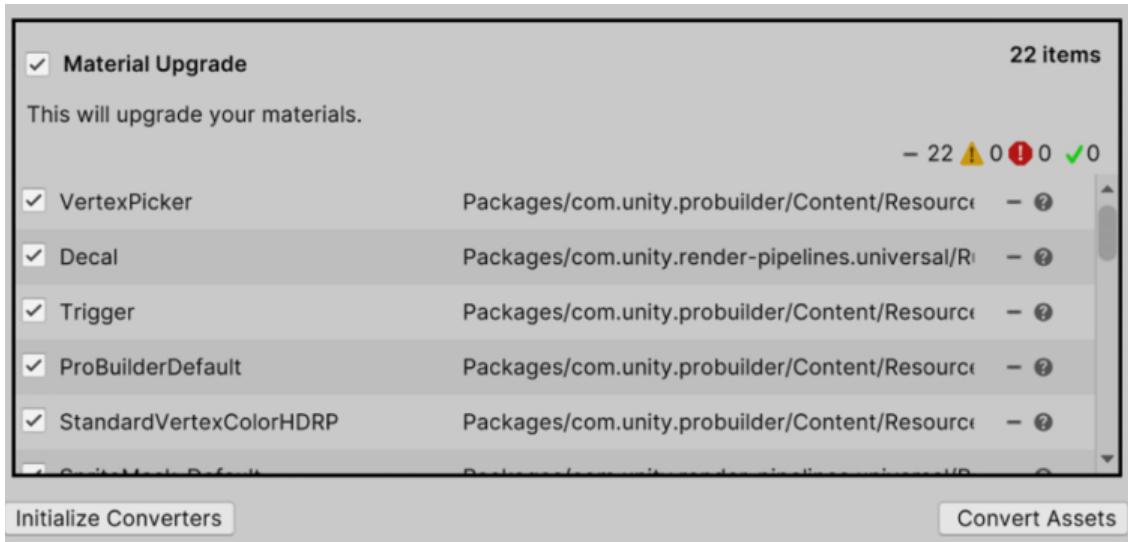


Figure 4.25: Fixing material to work with URP

5. Click the Convert Assets button and see if the model was fixed.

You will need to close the window for it to detect new magenta assets that weren't there before opening it. The con of this method is that, sometimes, it won't upgrade the material properly. Luckily, we can fix this by reapplying the textures of the objects manually. Even if your assets work just fine, I suggest that you reapply your textures anyway, just to learn more about the concept of materials. A texture is not applied directly to the object. That's because the texture is just one single configuration of all the ones that control the aspect of your model. In order to change the appearance of a model, you must create a **Material**. A material is a separate asset that contains lots of settings about how Unity should render your object. You can apply that asset to several objects that share the same graphics settings, and if you change the settings of the

material, it will affect all the objects that are using it. It works like a graphics profile. In order to create a material to apply the textures of your object, you need to follow these steps:

1. In the **Project Window**, click the plus (+) button in the top-left part of the window.
2. Click the **Material** option in that menu.
3. Name your material. This is usually the name of the asset we will be applying the material to (for example, `Car`, `Ship`, `Character`, and so on).
4. Drag the created material to the model instance on your scene. If you move the mouse with the dragged asset over the object, you will be able to see a preview of how it will look with that material, which would be white in the case of a new material. We will change that in the following steps.
5. Apply the material by releasing the mouse.
6. If your object has several parts, you will need to drag the material to each part. Dragging the material will change the material's property of the `MeshRenderer` component of the object you have dragged.
7. Select the material and click the circle to the left of the **Base Map** property (see *Figure 4.23*).
8. In the **Texture Selector**, click on the texture of your model. It can be complicated to locate the texture just by looking at it. Usually, the name of the texture will match the model's name. If not, you will need to try different textures until you see one that fits your object. Also, you may find several textures with the same name as your model. Just pick the one that seems to have the proper colors instead of the ones that look black and white or light blue; we will use those later:

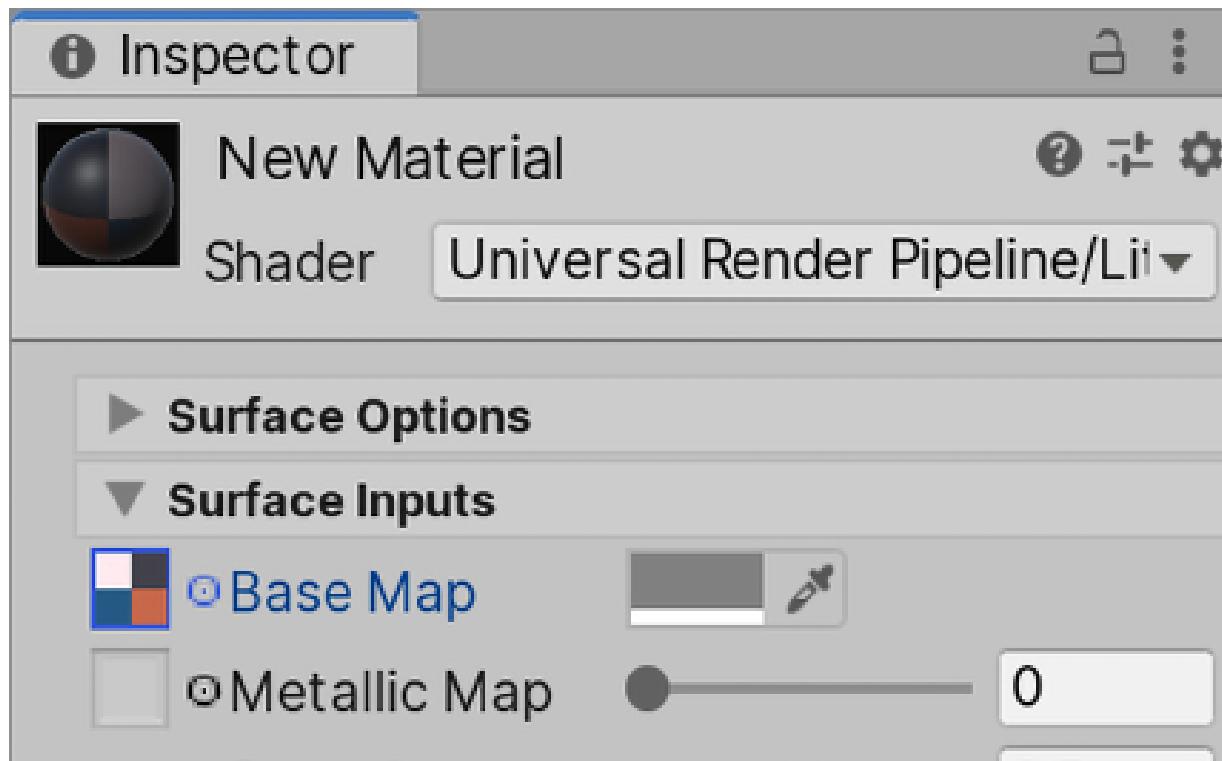


Figure 4.26: Base Map property of URP materials

With this, you have successfully applied the texture to the object through a material. For each object that uses the same texture, just drag the same material.

Materials have a concept like Prefab Variants, called Material Variants. It consists of the same idea of creating a base material and then alternative versions of it with small changes. For more information check the following documentation:
<https://docs.unity3d.com/2022.2/Documentation/Manual/materialvariant-landingpage.html>, and the following blog post:
<https://blog.unity.com/engine-platform/material-variants-the-solution-for-managing-complex-material-libraries>

Now that we have a basic understanding of how to apply the model textures, let's learn how to properly configure the import settings before spreading models all over the scene.

Configuring assets

As we mentioned earlier, artists are used to creating art assets outside Unity, and that can cause differences between how an asset is seen from that tool and how Unity will import it. As an example, 3D Studio Max can work in centimeters, inches, and so on, while Unity works in meters. We have just downloaded and used lots of assets, but we have skipped the configuration step to solve those discrepancies, so let's take a look at this now. In this section, we will cover the following concepts related to importing assets:

- Configuring meshes
- Configuring textures

Let's start by discussing how to configure 3D meshes.

Configuring meshes

In order to change the model's import settings, you need to locate the model file you have downloaded. There are several file extensions that contain 3D models, with the most common one being the `.fbx` file, but you can encounter others such as `.obj`, `.3ds`, `.blender`, `.mb`, and so on. You can identify whether the file is a 3D mesh via its extension:

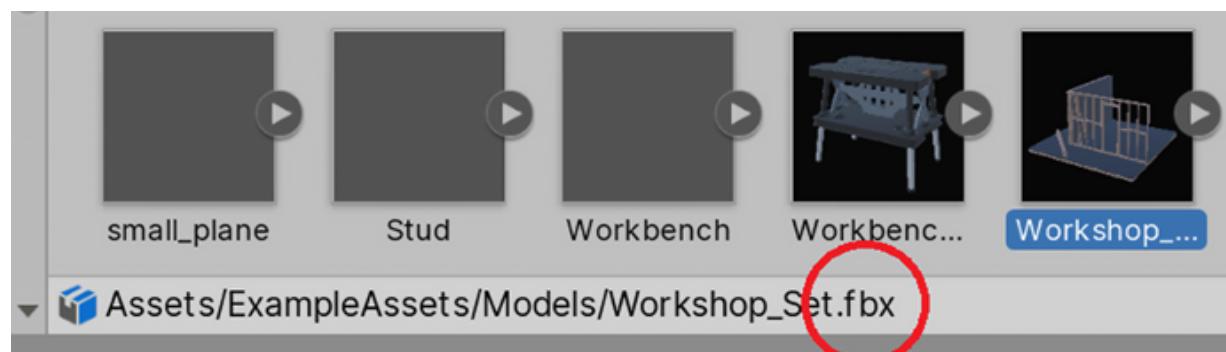


Figure 4.27: Selected asset path extension

Also, you can click the **asset** and check in the **Inspector** for the tabs you can see in the following screenshot:

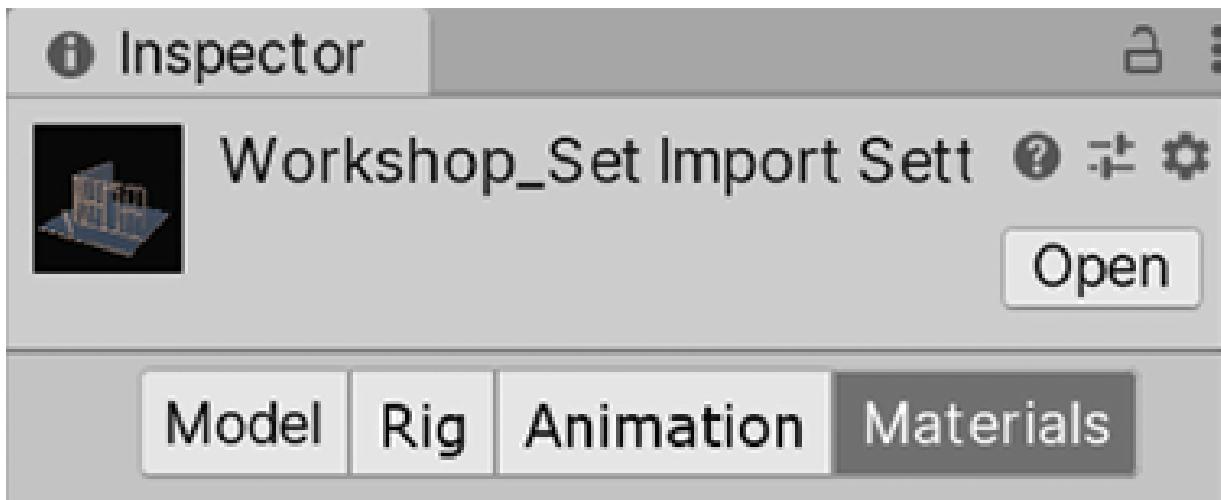


Figure 4.28: Mesh materials settings

Now that you have located the 3D mesh files, you can configure them properly. Right now, the only thing we should take into account is the proper scale of the model. Artists are used to working with different software with different setups; maybe one artist created the model using meters as its metric unit, while other artists used inches, feet, and so on. When importing assets that have been created in different units, they will probably be unproportioned, which means we will get results such as humans being bigger than buildings and so on. The best solution is to just ask the artist to fix that. If all the assets were authored in your company, or if you used an external asset, you could ask the artist to fix it to the way your company works, but right now, you are probably a single developer learning Unity by yourself. Luckily, Unity has a setting that allows you to rescale the original asset before using it in Unity. In order to change the "**Scale Factor**" of an object, you must do the following:

1. Locate the 3D mesh in your **Project Window**.
2. Drag it to the scene. You will see that an object will appear in your scene.

3. Create a capsule using the **GameObject | 3D Object | Capsule** option.
4. Put the capsule next to the model you dragged into the editor. See if the scale makes sense. The idea is that the capsule represents a human being (2 meters tall) so that you have a reference for the scale:



Figure 4.29: Using a capsule as reference for scale

5. If the model is bigger or smaller than expected, select the mesh again in the Project window (not the GameObject instance you dragged to the editor) and you will see some import settings in the Inspector. In the image, we can see that the model has a good relative size, but just for learning purposes, I recommend proceeding with the next steps.
6. Look for the **Scale Factor** property and modify it, increasing it if your model is smaller than expected, or reducing it in the opposite case:

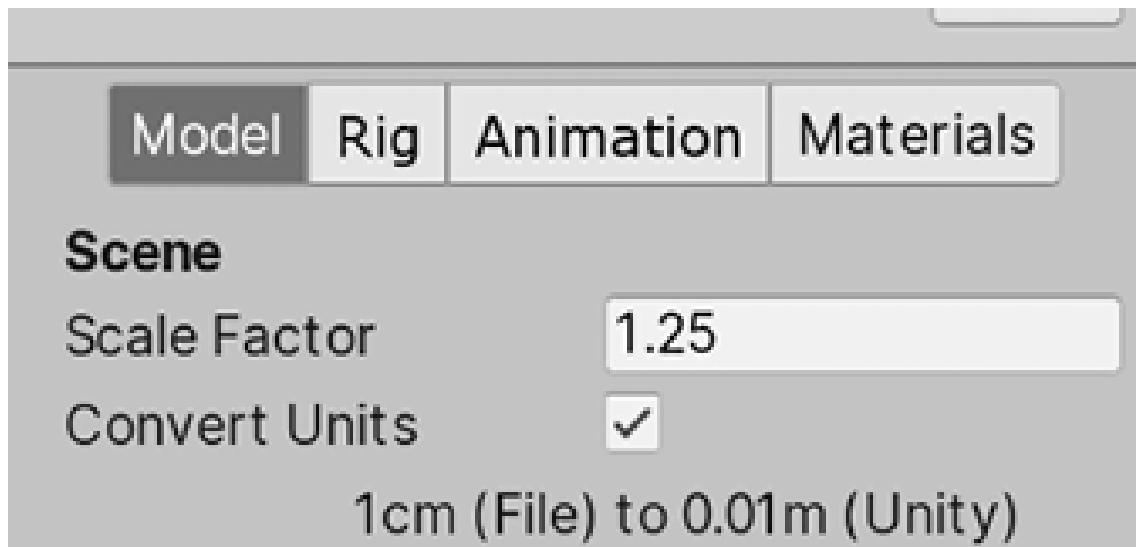


Figure 4.30: Model mesh options

7. Click the **Apply** button at the bottom of the Inspector.
8. Repeat *Steps 6 and 7* until you get the desired result.

There are plenty of other options to configure, but let's stop here for now. Next, let's discuss how to properly configure the textures of our models.

Configuring textures

Again, there are several settings to configure here, but let's focus on the **Texture Size** for now. The idea is to use the size that best fits the usage of that texture, and that depends on lots of factors. The first factor to take into account is the distance from the object to the camera. If you are creating a first-person game, you will probably encounter many objects up close – enough to justify the use of a big texture. However, if you have several distant objects, such as billboards at the top of buildings, which you will never be near enough to see the details of, you can use smaller textures for that. Another thing to take into account is the importance of the object. If you are creating a racing game, you will probably have lots of 3D models that will be onscreen for a few seconds and the player

will never focus on them; they will be paying attention to the road and other cars. In this case, an object such as a trash can on the street could have a little texture and a low polygon model and the user will never notice that (unless they stop to appreciate the scenery), but that's acceptable. Finally, you can have a game with a top-down view that will never zoom in on the scene, so the same object that has a big texture in first-person games will have a less detailed texture here. In the following images, you can see that the smaller ship could use a smaller texture:



Figure 4.31: The same model seen at different distances

The ideal size of the texture is relative. The usual way to determine the right size is by changing the dimensions until you find the smallest possible size exhibiting decent quality when the object is seen from the nearest possible position in the game. This is a trial-and-error method and you can do the following:

1. Locate the 3D model and put it into the scene.
2. Put the Scene view camera in a position that shows the object at its largest possible in-game size. As an example, in a **first-person-shooter (FPS)** game, the camera can be almost right next to the object, while in a top-down game, it would be a few

meters above the object. Again, that depends on your game. Remember our game is a third-person shooter.

3. Find and select the texture that the object is using in the folders that were imported with the package or from the material you created previously. They usually have `.png`, `.jpg`, or `.tif` extensions.
4. In the Inspector, look at the **Max Size** property and reduce it, trying the next smaller value. For example, if the texture is **2048**, try **1024**.
5. Click **Apply** and check the Scene view to see if the quality has decreased dramatically or if the change is unnoticeable. You will be surprised.
6. Repeat *Steps 4 to 5* until you get a bad-quality result. Once you do, just increase the previous resolution to get an acceptable quality. Of course, if you are targeting PC games, you can expect higher resolutions than mobile games.

Now that you have imported, integrated, and configured your objects, let's create our player's base with those assets.

Assembling the scene

Let's start replacing our prototype base using the environment pack we have downloaded. To do that, you must do the following:

1. In the **Environment** pack we imported before, locate the folder that contains all the models for the different pieces of the scene and try to find a corner. You can use the search bar in the **Project Window** to search for the `corner` keyword:

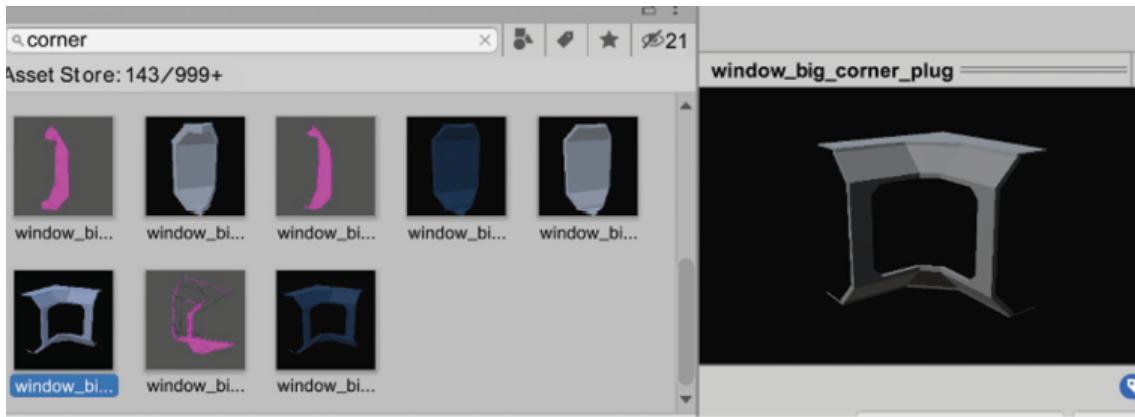


Figure 4.32: Mesh picker

2. In my specific case, I have the outer and inner sides of the corner as separate models, so I need to put them together.
3. Place it in the same position as any corner of your prototype base:

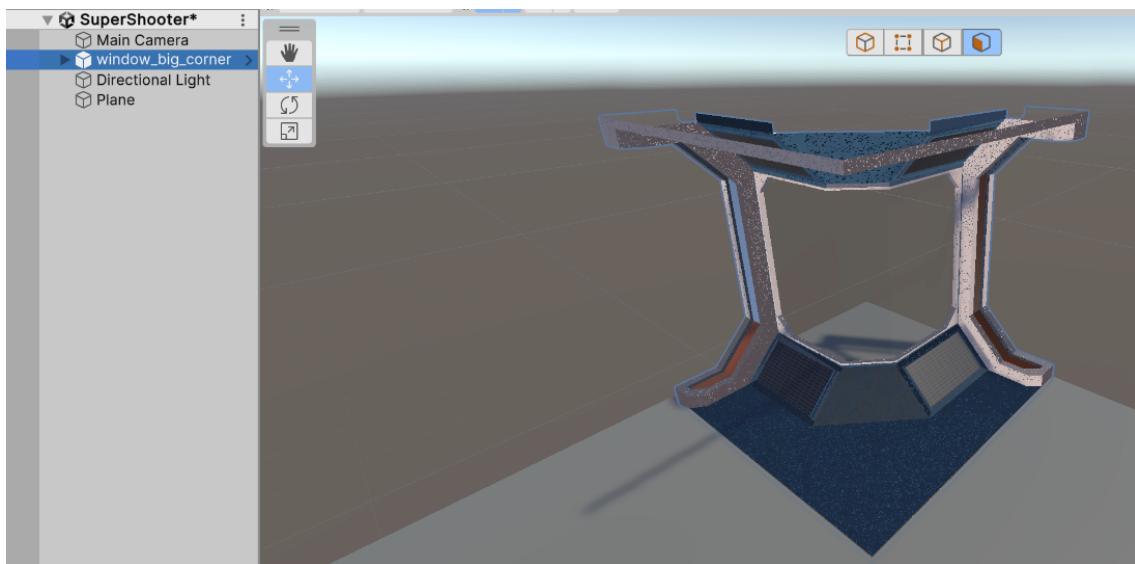


Figure 4.33: Positioning the mesh on a placeholder for replacement

4. Find the proper model that will connect with that corner to create walls. Again, you can try searching for the `wall` keyword in the **Project** window.

5. Instance it and position it so that it's connected to the corner.
Don't worry if it doesn't fit perfectly; you will go over the scene when necessary later.

You can select an object and press the V key to select a vertex of the selected object. Then you can drag it, click on the rectangle in the middle of the translate gizmo, and direct it to a vertex of another object. This is called **Vertex Snapping**. It allows you to connect two pieces of the scene exactly as intended.

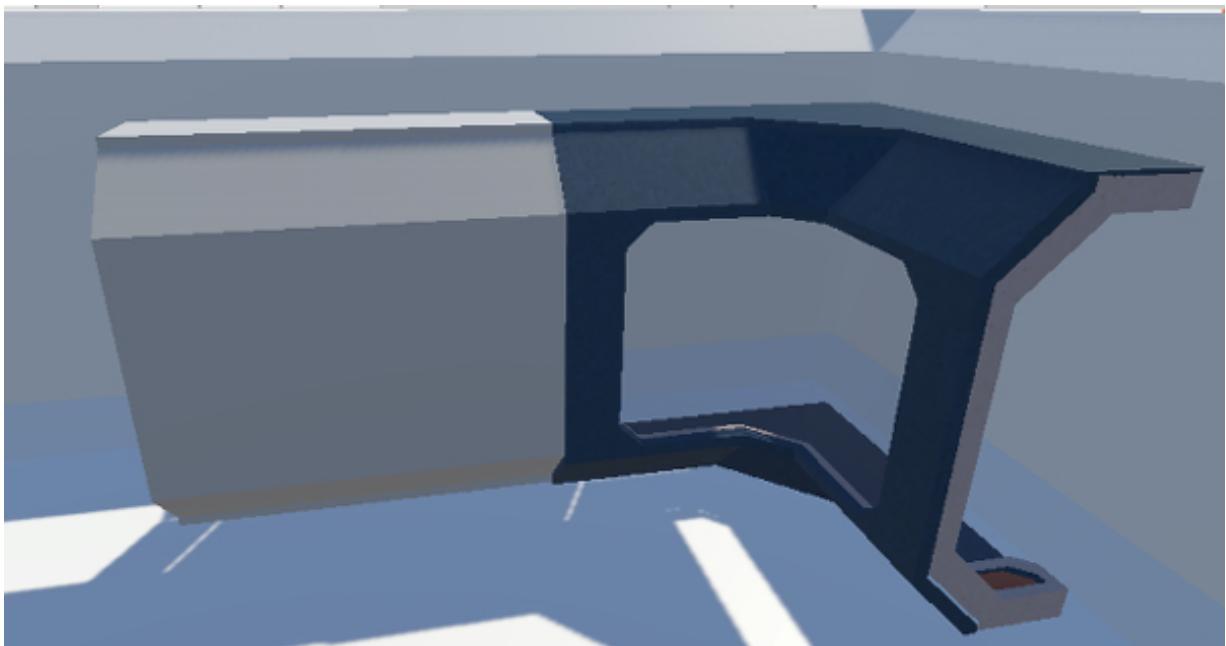


Figure 4.34: Connecting two modules

1. Repeat the walls until you reach the other end of the player base and position another corner. You might get a wall that's a little bit larger or smaller than the original prototype, but that's fine:



Figure 4.35: Chain of connected modules

You can move an object while pressing the Ctrl key (Command on Mac) to snap the object's position so that the clones of the wall can be easily located right next to the others. Another option is to manually set the `Position` property of the `Transform` component in the Inspector.

1. Complete the rest of the walls and destroy the prototype cube we made in ProBuilder. Remember that this process is slow and you will need to be patient.
2. Add floors by looking for floor tiles and repeating them all over the surface:

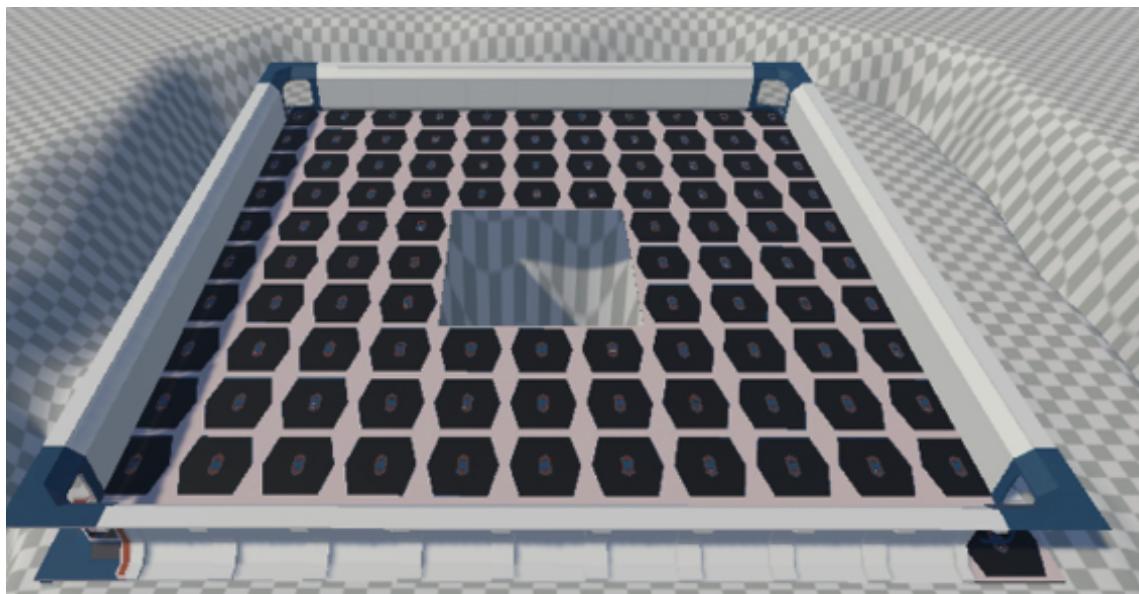


Figure 4.36: Floor modules with a hole for the pit

3. Add whatever details you want to add with other modular pieces in the package.
4. Put all those pieces in a container object called `Base`.
Remember to create an empty object and drag the base pieces into it:

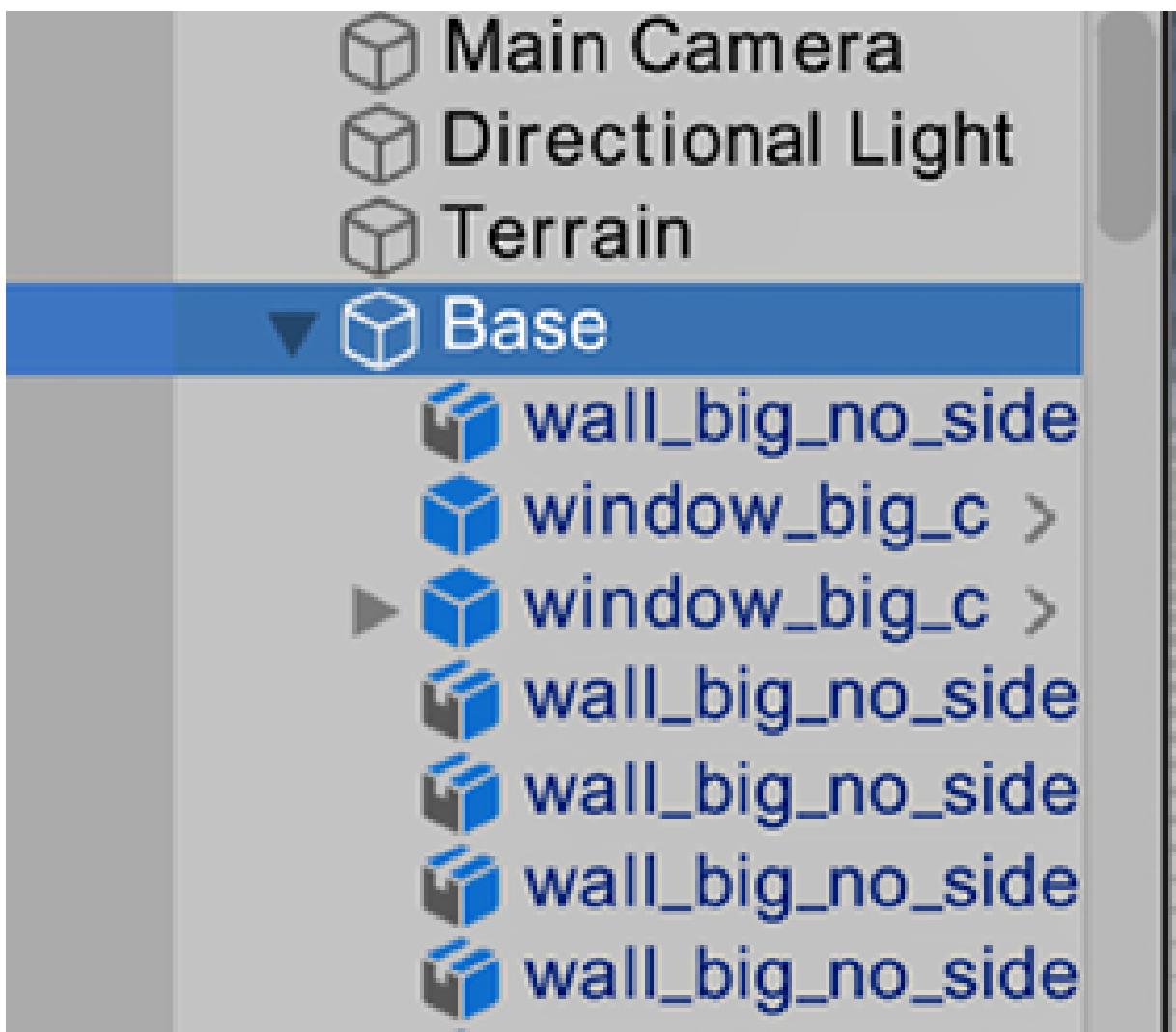


Figure 4.37: Mesh sub-assets

With this, we learned how to create a scene easily by using a module approach, assembling the different pieces by using Unity's snapping features. After a lot of practice doing this, you will slowly gain experience with the common pitfalls and good practices of modular

scene design. All the packages have a different modular design in mind, so you will need to adapt to them.

Summary

In this chapter, we learned how to import models and textures and integrate them into our scene. We discussed how to apply textures to the terrain, how to replace our prototype mesh with modular models, how to apply textures to those, and how to properly configure the assets, all while taking several criteria into account according to the usage of the object. With this, we have finished *Part 1* of this book and discussed several basic Unity concepts we will use throughout the book. In *Part 2*, we will start coding the gameplay of our game, like the player's movement and the health system. We will start learning how to create our own components to add behavior to our objects and the basic anatomy of a script.

5 Unleashing the Power of C# and Visual Scripting

Join our book community on Discord

<https://packt.link/unitydev>



Unity has a lot of great built-in tools to solve the most common problems in game development, such as the ones we have seen so far. Even two games of the same genre have their own little differences that make each game unique, and Unity cannot foresee that, so that's why we have scripting. In this chapter, we will introduce the two main Unity scripting options: C# and Visual Scripting. We will discuss their pros and cons and the base knowledge required to start creating gameplay with them. From now on, we will see how to achieve all our scripts using both options. In this chapter, we will examine the following topics:

- Introducing scripting
- Creating scripts
- Using events and instructions
- Common beginner C# script errors

We are going to create our own Unity components, learning the basic structure of a script and the way that we can execute actions and expose properties to be configured, both with C# and Visual Scripting. We are not going to create any of our actual game code in this chapter, just some example scripts to set the groundwork to do

so in the next one. Let's start by discussing Unity's scripting options.

Introducing scripting

Through coding, we can extend Unity's capabilities in several ways to achieve the exact behavior we need, all through a well-known programming language—C#. However, aside from C#, Unity also has **Visual Scripting**, a way to generate code through a node graph tool. This means you can create scripts without writing code but by dragging **nodes**, boxes that represent actions that can be chained:

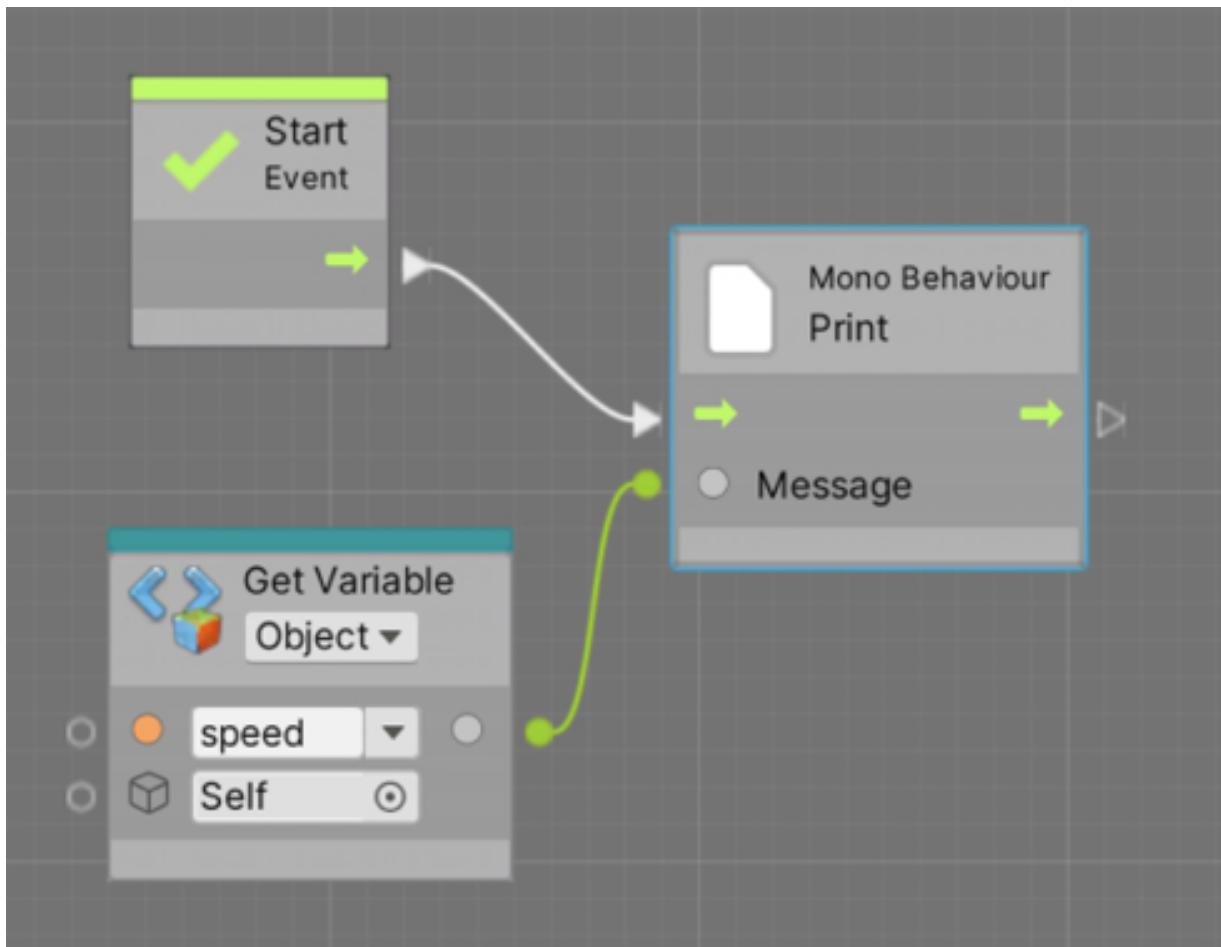


Figure 5.1: Example of a Visual Scripting graph

While essentially both ways can achieve the same result, we can use them for different things. Usually, the core logic of a game is written in C# due to it being usually huge and very performance-sensitive. However, sometimes using visual scripts instead allows non-programmer team members, like artists or game designers, to have more freedom to edit minor changes in the game, especially regarding balancing or visual effects. Another example would be game designers prototyping ideas through visual scripts that programmers would later convert to C# scripts when the idea is approved. Also, C# programmers can create nodes for Visual Script programmers to use. The way to mix these tools varies widely between teams, so while, in the next chapters, we are going to focus mainly on C#, we are also going to see the Visual Scripting equivalent version of the scripts we are going to create. This way, you will have the opportunity to experiment when convenient to use one or the other, according to your team structure. Now, let's continue by discussing the basics of script creation.

Creating scripts

The first step to creating behavior is to create script assets; these are files that will contain the logic behind the behavior of our components. Both C# and Visual Scripting have their own type of asset to achieve that, so let's explore how to do that in both tools. Having some programming knowledge is required in this book. However, in this first section, we are going to discuss a basic script structure to make sure you have a strong foundation to follow when we code the behaviors of our game in the following chapters. Even if you are familiar with C#, try not to skip this section because we will cover Unity-specific structures of code. In this section, we will examine the following script creation concepts:

- Initial setup
- Creating a C# script
- Adding fields

- Creating a Visual Script graph

We are going to create our first script, which will serve to create our component, discussing the tools needed to do so and exploring how to expose our class fields to the editor. Let's start with the basics of script creation.

Initial setup

Support for Visual Scripting is added by installing the **Visual Scripting** package in the **Package Manager** as we did with other packages in previous chapters, but as Unity does that automatically for us when we create the project, we don't need any further setup. That means the rest of this section will take care of setting up the tools needed to work with C#. One thing to consider before creating our first C# script is how Unity compiles the code. While coding, we are used to having an **Integrated Development Environment (IDE)**, which is a program to create our code and compile or execute it. In Unity, we will just use an IDE as a tool to create the scripts easily with coloring and auto-completion because Unity doesn't have a custom code editor (if you have never coded before, these are valuable tools for beginners). The scripts will be created inside the Unity project and Unity will detect and compile them if any changes are made, so you won't compile them in the IDE. Don't worry, even if not compiling and running the code in the IDE, it is possible to debug, add breakpoints, and check the data on the variables and structures using the IDE and Unity together. We can use Visual Studio, Visual Studio Code, Rider, or whatever C# IDE you'd like to use, but when you install Unity, you will probably see an option to install Visual Studio automatically, which allows you to have a default IDE. This installs the free version of Visual Studio, so don't worry about the licenses here. If you don't have an IDE on your computer and didn't check the Visual Studio option while installing Unity, you can do the following:

1. Open **Unity Hub** and go to the **Installs** section.

2. Click on the wheel button in the top-right area of the Unity version you are using and click on **Add Modules**:

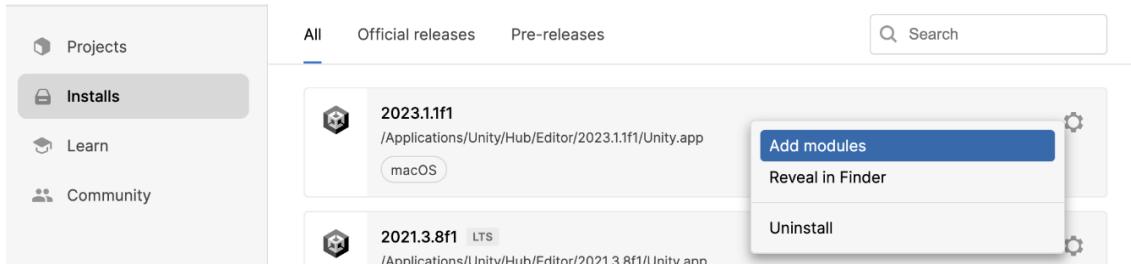


Figure 5.2: Adding a module to the Unity installation

3. Check the option that says **Visual Studio**; the description of the option will vary, depending on the version of Unity and the platform you are using.
4. Hit the **Continue** button at the bottom-right:

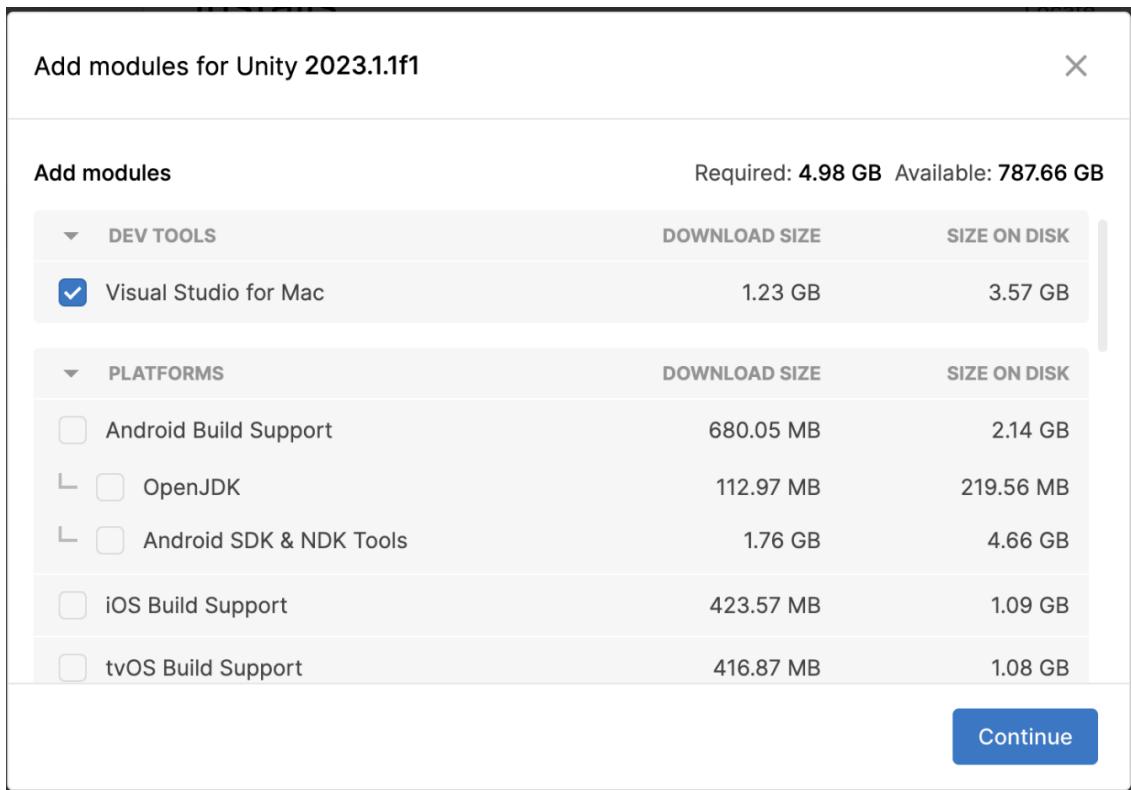


Figure 5.3: Installing Visual Studio

5. Check that you accept the terms and conditions, and click **Install**:

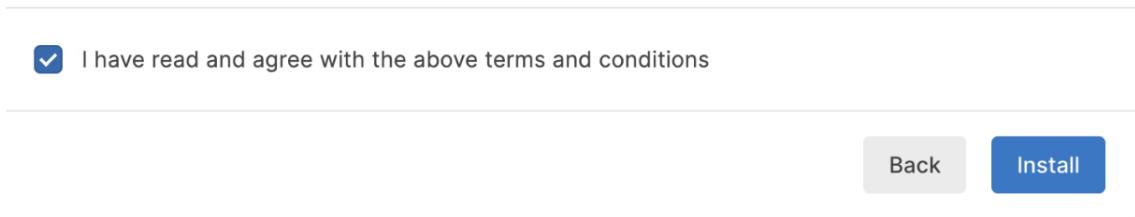


Figure 5.4: Accepting the terms and conditions

6. Wait for the operation to end. This might take a few minutes. There may be additional Visual Studio steps that vary between platform and version; if so, just follow them.

If you have a preferred IDE, you can install it yourself and configure Unity to use it. If you can afford it or you are a teacher or a student (as it is free in these cases), I recommend Rider. It is a great IDE with lots of C# and Unity features that you will love; however, it is not vital for this book. In order to set up Unity to use a custom IDE, do the following:

1. Open the project and go to **Edit | Preferences** in the top menu of the editor (**Unity | Preferences** on a Mac).
2. Select the **External Tools** menu from the left panel.
3. From the external script editor, select your preferred IDE; Unity will automatically detect the supported IDEs:

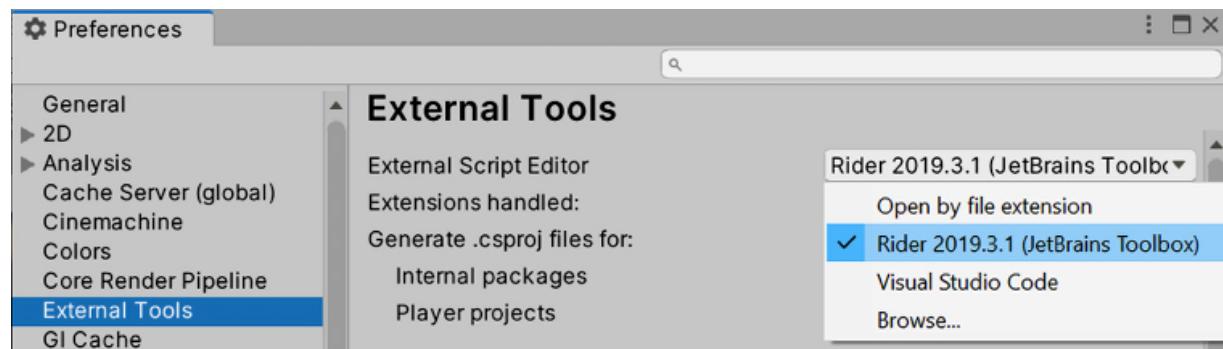


Figure 5.5: Selecting a custom IDE

- If you don't find your IDE in the list, you can use the **Browse...** option. Note that, usually, IDEs that require you to use this option are not very well supported—but it's worth a shot.
1. Press **Regenerate project files** after selecting your IDE. That will recompile all the necessary files of the project so you don't run into any issues if some of the project files don't exist.

Finally, some IDEs, such as Visual Studio, Visual Studio Code, and Rider, have Unity integration tools that you need to install in your project, which is optional but can be useful. Usually, Unity installs these automatically, but if you want to be sure that they are installed, follow these steps:

1. Open Package Manager (Window | Package Manager).
2. Set the Packages dropdown to Unity Registry mode:

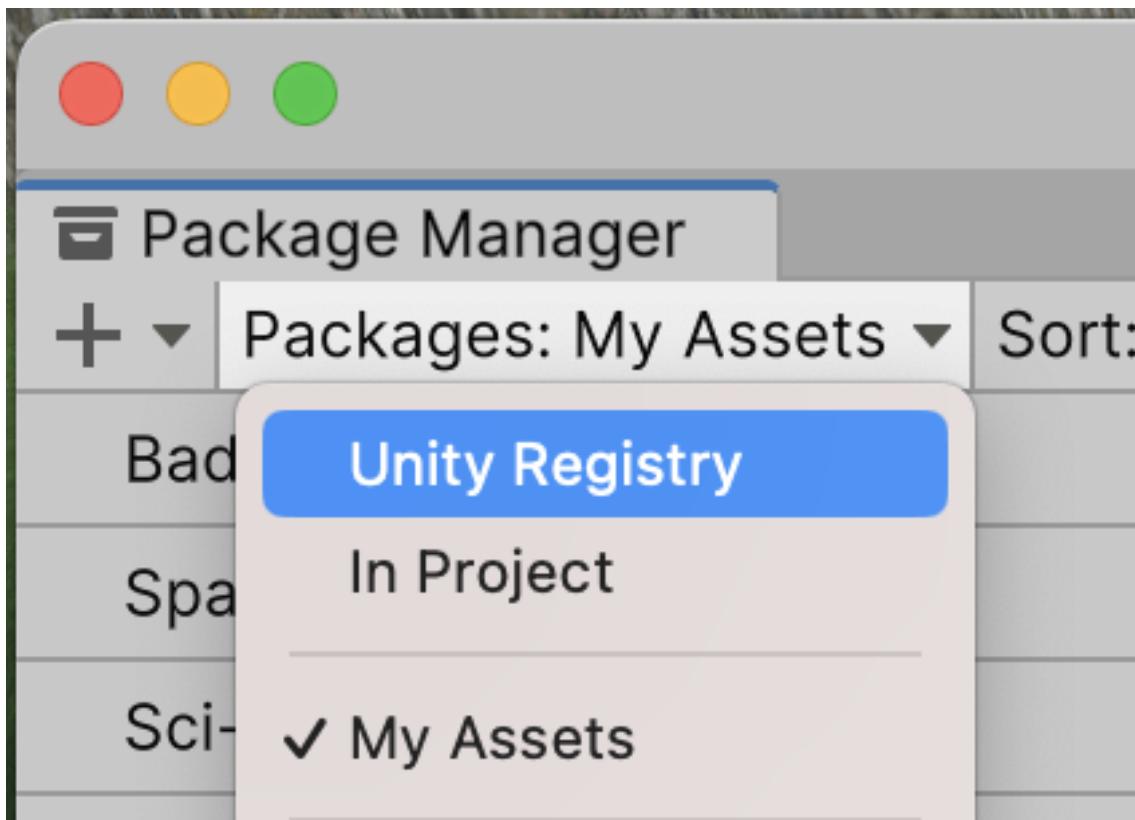


Figure 5.6: Enabling Unity Registry mode

3. Search the list for your IDE or filter the list by using the search bar. In my case, I used Rider, and I can find a package called **JetBrains Rider Editor**:

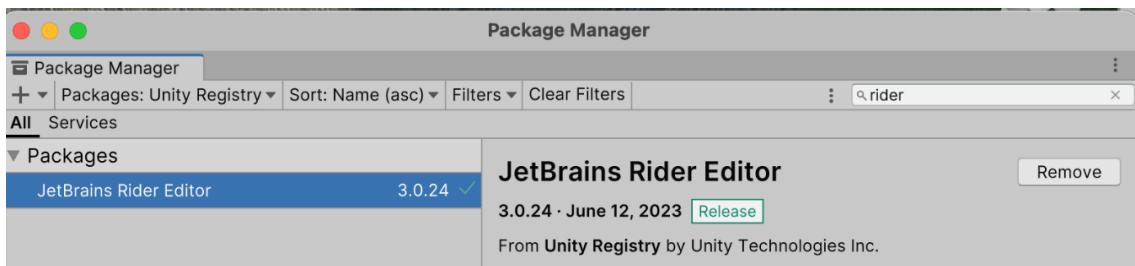


Figure 5.7: Custom IDE editor extension installation—in this case, the Rider one

4. Check whether your IDE integration package is installed by looking at the buttons at the bottom-right part of the package

manager. If you see an **Install** or **Update** button, click on it; but if it says **Installed**, everything is set up.

Now that we have an IDE configured, let's create our first script.

Creating a C# script

C# is an object-oriented language. Any time we want to extend Unity, we need to create our own class—a script with the instructions we want to add to Unity. If we want to create custom components, we need to create a class that inherits from `MonoBehaviour`, the base class of every custom component. We can create C# script files directly within the Unity project using the editor, and you can arrange them in folders right next to other `assets` folders. The easiest way to create a script is by following these steps:

1. Select any `GameObject` that you want to have the component we are going to create. As we are just testing this out, select any object.
2. Click on the **Add Component** button at the bottom of the Inspector, and look for the **New script** option at the bottom of the list, displayed after clicking on **Add Component**:

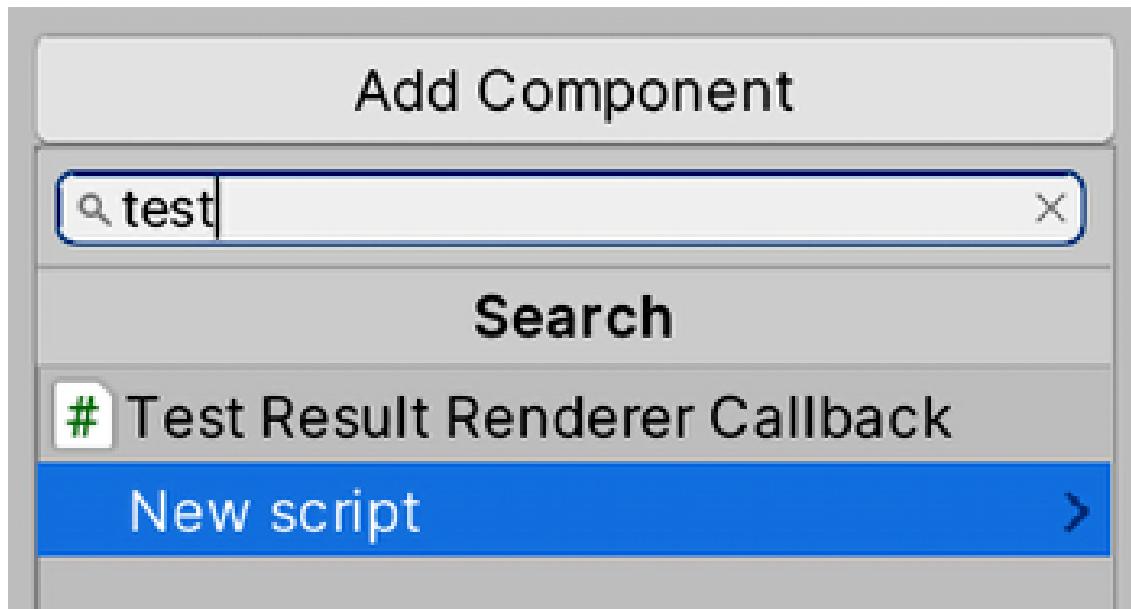


Figure 5.8: The New script option

3. In the **Name** field, enter the desired script name, and then click **Create and Add**. In my case, I will call it `MyFirstScript`, but for the scripts that you will use for your game, try to enter descriptive names, regardless of the length:

A small, light gray arrow pointing to the left, indicating a back or cancel action.

New script

Name

A text input field containing the text "MyFirstScript". The field has a blue border and is the active element.

Create and Add

Figure 5.9: Naming the script

It is recommended that you use Pascal case for script naming. In Pascal case, a script for the player's shooting functionality would be called `PlayerShoot`. The first letter of each word of the name is in uppercase, and you can't use spaces.

1. You can check how a new asset with the same name as your script is created in **Project View**. Remember that each component has its own asset, and I suggest you put each component in a `Scripts` folder:



Figure 5.10: Script asset

2. Now, you will also see that your GameObject has a new component in the Inspector window, which has the same name as your script. So you have now created your first `component` class:

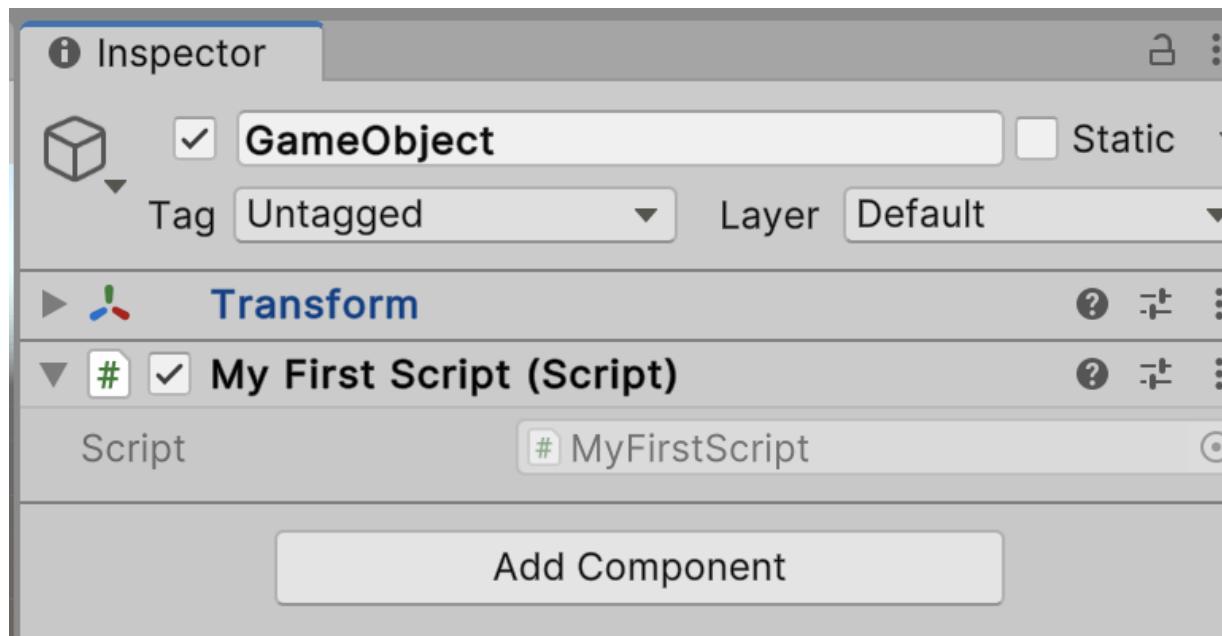


Figure 5.11: Our script added to a GameObject

Now that we have created a `component` class, remember that a class is not the component itself. It is a description of what the component should be—a blueprint of how a component should work. To actually use the component, we need to instantiate it by creating a component based on the class. Each time we add a component to an object using the editor, Unity is instantiating it for us. If you are familiar with object-oriented programming languages, you may recall that when we use a programming language like C#, we need to code the instantiation of the object using one specific keyword inside a script: `new`. In Unity, there is no need to do that; generally, we don't instantiate components using the new C# keyword but, instead, by using the editor or specialized functions. Now, you can add your new empty component to other objects as you would any other component, by using the **Add Component** button in the Inspector window. Then, you can look for the component in the **Scripts** category or search for it by name:

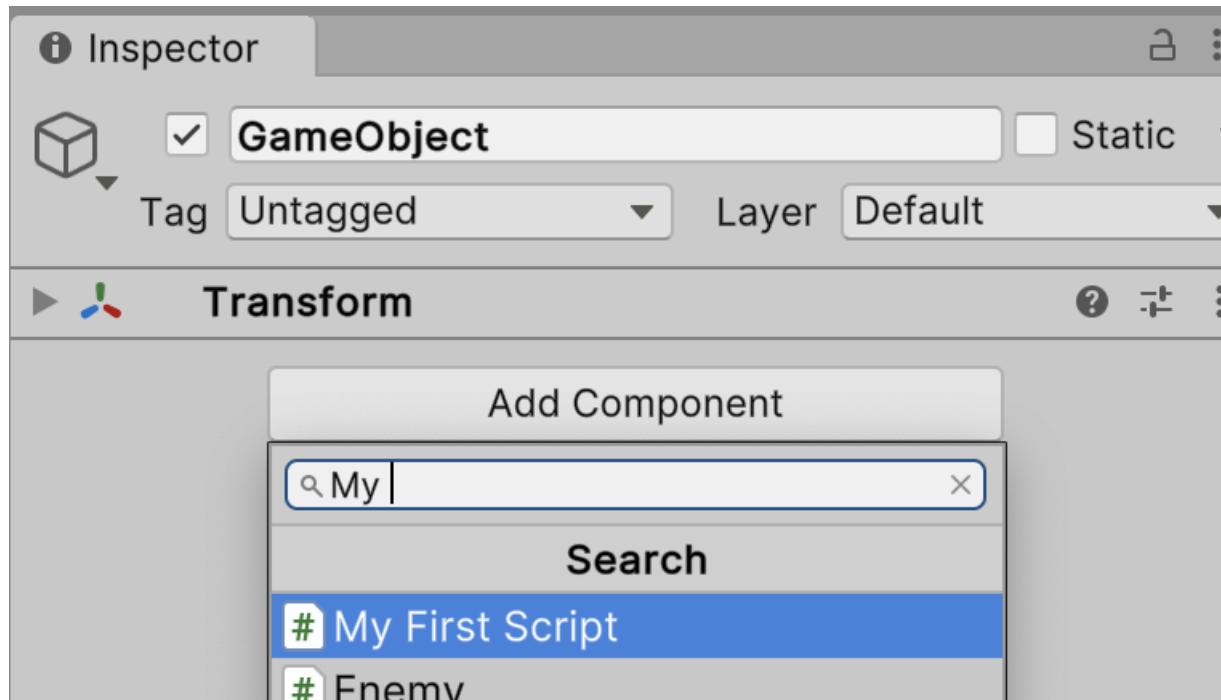


Figure 5.12: Adding a custom component in the Scripts category

Something that you need to consider here is that we can add the same component to several GameObjects. We don't need to create a class for each GameObject that uses the component. I know this is basic programmers' knowledge, but remember that we are trying to recap the basics here. Now that we have our component, let's explore how it looks and carry out a class structure recap by following these steps:

1. Locate the script asset in **Project View** and double-click on it. Remember that it should be in the `Scripts` folder you created previously.
2. Wait for the IDE to open; this can take a while. You will know that the IDE has finished the initialization when you see your script code and its keywords properly colored like in the following figure, which varies according to the desired IDE. In Rider, it looks like what is shown in *Figure 5.13*. In my case, I

knew that Rider had finished initializing because the `MonoBehaviour` type and the script name are colored the same:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MyFirstScript : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
    }
}
```

Figure 5.13: A new script opened in the Rider IDE

3. The first three lines in the preceding screenshot—the ones that start with the `using` keyword—include common namespaces. **Namespaces** are like code containers, which are, in this case, code created by others (such as Unity, C# creators, and so on). We will be using namespaces quite often to simplify our tasks; they already contain solved algorithms that we will use. We will add and remove the `using` component as we need; in my case, Rider suggests that the first two `using` components are not necessary because I am not using any code inside them, and so they are grayed out. But keep them for now, as you will use them in later chapters of this book. Remember, they should always be at the beginning of a file:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

Figure 5.14: The using sections

4. The next line, the one that starts with `public class`, is where we declare that we are creating a new class that inherits from `MonoBehaviour`, the base class of every custom component. We know this because it ends with `: MonoBehaviour`. You can see how the rest of the code is located inside brackets right below that line, meaning that the code inside them belongs to the component:

```
public class MyFirstScript : MonoBehaviour
```

Figure 5.15: The MyFirstScript class definition inherits from MonoBehaviour

Now that we have our C# script, let's add fields to configure it.

Adding fields

In previous chapters, when we added components such as `Rigidbody` or different kinds of colliders, adding the components wasn't enough. We needed to properly configure them to achieve the exact behavior that we needed. For example, `Rigidbody` has the `Mass` property to control an object's weight, and the colliders have the `Size` property to control their shape. This way, we can reuse the same component for different scenarios, preventing the duplication of similar components. With a `Box` collider, we can represent a cube or rectangular box just by changing the size properties. Our

components are no exception; if we have a component that moves an object and we want two objects to move at different speeds, we can use the same component with different configurations. Each configuration is a **field** or **variable** where we can hold a parameter's value. We can create class fields that can be edited in the editor in two ways:

- By marking the field as `public`, but breaking the encapsulation principle
- By making a private field and exposing it with an attribute

Now, we are going to cover both methods, but if you are not familiar with **Object-Oriented Programming (OOP)** concepts, such as encapsulation, I recommend you use the first method. Suppose we are creating a movement script. We will add an editable number field representing the velocity using the first method—that is, by adding the `public` field. We will do this by following these steps:

1. Open the script by double-clicking on it, as we did before.
2. Inside the class brackets, but outside any brackets within them, add the following code:

```
public class MyFirstScript : MonoBehaviour
{
    public float speed;
```

Figure 5.16: Creating a speed field in our component

The `public` keyword specifies that the variable can be seen and edited beyond the scope of the class. The `float` part of the code says that the variable uses the decimal number type, and `speed` is the name we chose for our field—although this can be whatever you want. You can use other value types to represent other kinds of data,

such as `bool` to represent checkboxes or `Booleans` and `string` to represent text.

- To apply the changes, just save the file in the IDE (usually by pressing `Ctrl + S` or `Command + S`) and return to Unity. When you do this, you will notice a little loading wheel at the bottom-right part of the editor, indicating that Unity is compiling the code. You can't test the changes until the wheel stops turning:



Figure 5.17: The loading wheel

Remember that Unity will compile the code; don't compile it in the IDE.

1. After the compilation is finished, you can see your component in the Inspector window and the **Speed** variable should be there, allowing you to set the speed you want. Of course, right now, the variables do nothing. Unity doesn't recognize your intention by the name of the variable; we need to set it for use in some way, but we will do that later:

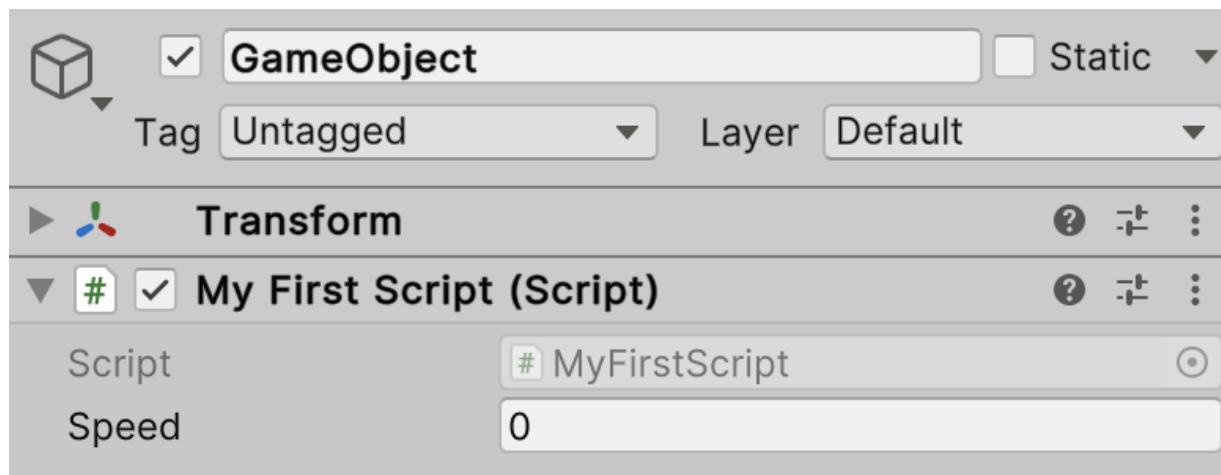


Figure 5.18: A public field to edit data that the component will use later

If you don't see the speed variable, please check the section at the end of this chapter called *Common beginner C# script errors*, which will give you tips about how to troubleshoot compilation errors.

1. Try adding the same component to other objects, and set a different speed. This will show you how components in different GameObjects are independent, allowing you to change some of their behaviors via different settings.
2. Instead of using the `public` keyword as we did in the previous steps to define available properties in the inspector, we create a `private` field, encouraging encapsulation and exposing it using

the `SerializeField` attribute, as shown in the following screenshot.



The image shows a portion of a C# script. It starts with the text "[SerializeField]" in purple, followed by "private float speed;" in blue. This indicates that the `speed` variable is being exposed for inspection in the Unity Editor.

Figure 5.19: Exposing private attributes in the Inspector window

If you are not familiar with the OOP concept of encapsulation, just use the first approach, using the `public` keyword, to expose the variable on the Inspector, which is more flexible for beginners. If you create a `private` field, it won't be accessible to other scripts because the `SerializeField` attribute only exposes the variable to the editor. Remember that Unity won't allow you to use constructors, so the only way to set initial data and inject dependencies is via serialized `private` fields or `public` fields and setting them in the editor (or using a dependency injection framework, but that is beyond the scope of this book). For simplicity, we will use the first method in most of the exercises in this book. If you want, try to create other types of variables, and check how they look in the Inspector. Try replacing `float` for `bool` or `string`, as previously suggested. Remember that not every possible C# type is recognized by Unity; through this book, we will learn the most commonly supported ones. Now that we know how to configure our components through data, let's use that data to create some behavior.

Even if we are using C#, which is still a very fast language, Unity has a feature called IL2CPP, which automatically converts our scripts to optimized C++ code. Check this documentation to get more info:

<https://docs.unity3d.com/Manual/IL2CPP.html>. However, IL2CPP is not always necessary, as the code you will write while reading this book will still be fast enough. We won't do a massive simulation with thousands of GameObjects, but I still recommend experimenting with IL2CPP, especially on mobile devices, where the performance boost is going to be significant.

Now that we have our C# script, let's see how to do the same in Visual Scripting.

Creating a Visual Script

As we need to create a script asset for C# scripts, we need to create the Visual Scripting equivalent of it called **Script Graph** and also attach it to our GameObject, although using a different approach this time. Before continuing, it is worth noticing that our objects must only have C# or the Visual Scripting version, but not both, or the behavior will be applied twice, once per version. Essentially, only do the steps for the version you want to try or do both steps in different objects if you want to experiment. Let's create a Visual Script doing the following:

1. Create a new GameObject to which we will add the Visual Script.
2. Add the **Script Machine** component to it. This component will execute the **Visual Script Graph** we will be creating shortly:

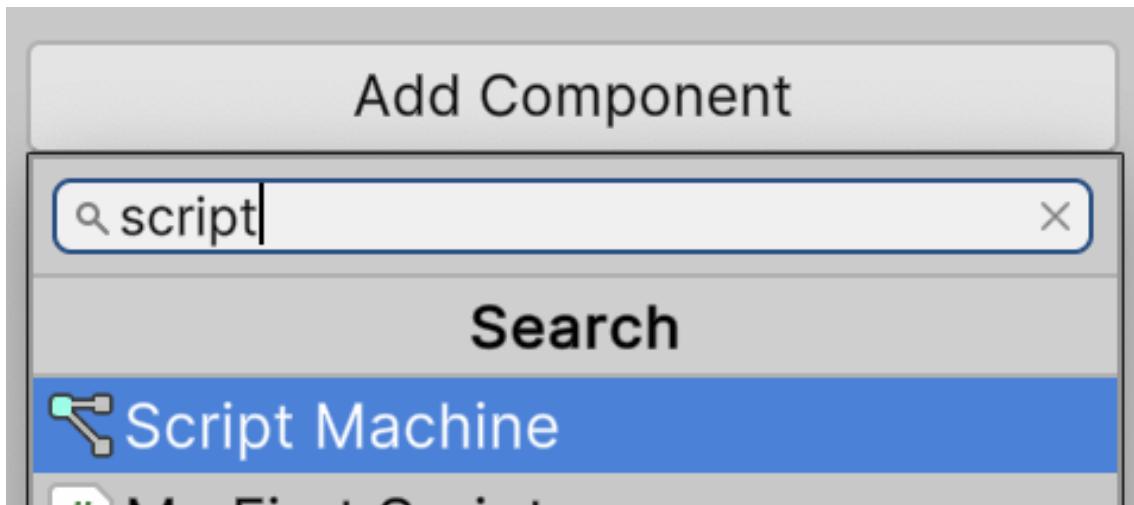


Figure 5.20: Adding a Script Machine component

3. In the **Script Machine** component, click the **New** button, and select a folder and a name to save the **Visual Script Graph** asset. This asset will contain the instructions of our script, and the **Script Machine** component will execute those:

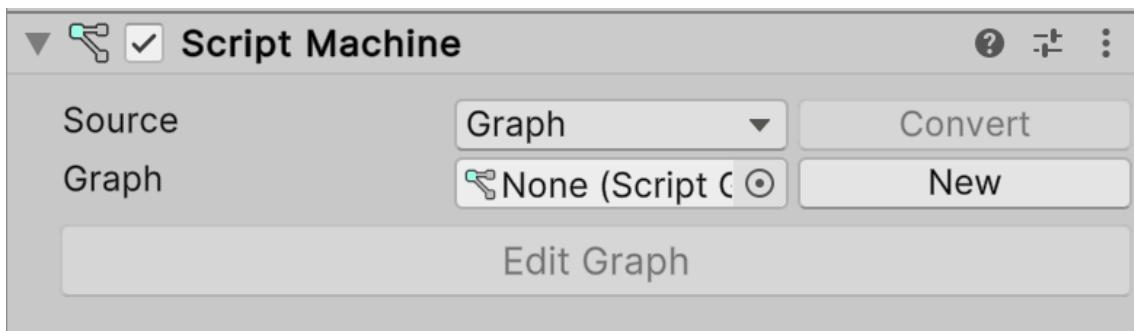


Figure 5.21: Using the New button to create a Visual Scripting Graph asset

4. If a warning appears, click the **Change now** option. This will prevent those changes on the script from affecting the game while it's running because, as the warning says, it can cause instability in code. Always stop the game, change the code, and then play again.

- Click the **Edit Graph** Button to open the Visual Script editor window. You can drag the **Script Graph** tab to any part of the editor to merge that window:

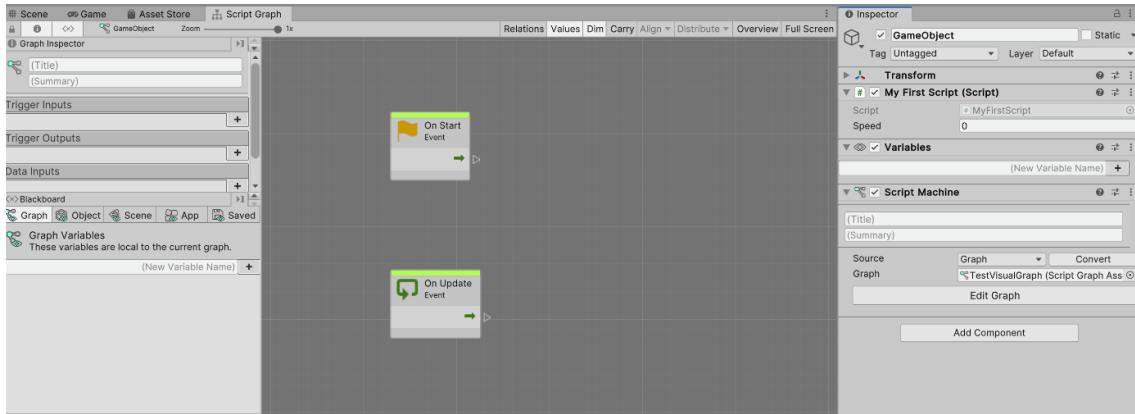


Figure 5.22: Visual Scripting asset editor

- Put the mouse in an empty area in the grid of the **Visual Script** editor, and while holding the middle mouse button, move the mouse to scroll through the graph. On MacBooks and Apple Magic Mouses, you can scroll using two fingers on the trackpad.

What we did is create the **Visual Graph** asset that will contain the code of our script, and attach it to a GameObject through the **Script Machine** component. Unlike C# scripts, we can't attach the **Graph Asset** directly; that's why we need the **Script Machine** to run the component for us. Regarding fields, the ones we created in the C# scripts are contained in the script itself, but for **Visual Graph**, they work a little bit differently. When we added the **Script Machine** component, another one was added: the **Variables** component. This will hold all the variables for all the **Visual Script Graph** that a GameObject can contain. That means that all graphs we add to our object will share those variables. You can create graph-specific variables if you want, but they won't be exposed in the Inspector, and this way also simplifies the access of variables from other objects' scripts. Also remember, you will want to add several graphs

to the object, given that each graph will take care of different behaviors, in a way in which we can mix and match them according to our needs. In order to add a variable to our GameObject that can be used by our graph, let's do the following:

1. Select a GameObject with a **Visual Script** added (with the **Script Machine** component) and look at the **Variables** component.
2. Click the input field that says **(New Variable Name)** and type the name of the variable. In my case, this is `speed`. If you don't see that option, click the triangle at the left of the **Variables** component name.
3. Click the **Plus (+)** button of the **Variables** component.
4. In the **Type** dropdown, select **Float**.
5. Optionally, you can set an initial value in the **Value** field:



Figure 5.23: Creating variables for the Visual Graph

We created a `speed` variable that we can configure in the GameObject to alter the way all **Visual Scripts Graphs** attached to our GameObject will work, or at least the ones that use that `Variable` value. Consider that maybe you will have different kinds of speed, like movement and rotational speed, so in real cases you might want to be a bit more specific with the variable name. The `Variables` component used in Visual Scripting is also called **Blackboard**, a common programming technique. This Blackboard

is a container of several values of our object, like a memory or database, that several other components of our object will then query and use. C# scripts usually contain their own variables inside instead. With our scripts created and ready to be configured, let's see how to make both of them do something.

Using events and instructions

Now that we have a script, we are ready to do something with it. We won't implement anything useful in this chapter, but we will settle the base concepts to add interesting behavior to the scripts we are going to create in the next chapters. In this section, we are going to cover the following concepts:

- Events and instructions in C#
- Events and instructions in Visual Scripting
- Using fields in instructions

We are going to explore the **Unity event system**, which will allow us to respond to different situations by executing instructions. These instructions will also be affected by the value of the editor. Finally, we are going to discuss common scripting errors and how to solve them. Let's start by introducing the concept of Unity events in C#.

Events and instructions in C#

Unity allows us to create behavior in a cause-effect fashion, which is usually called an **event system**. An event is a situation that Unity is monitoring—for example, when two objects collide or are destroyed, Unity tells us about this situation, allowing us to react according to our needs. As an example, we can reduce the life of a player when it collides with a bullet. Here, we will explore how to listen to these events and test them using some simple actions. If you are used to event systems, you will know that they usually require us to

subscribe to some kind of listener or delegate, but in Unity, there is a simpler method available. For C# scripts, we just need to write a function with the exact same name as the event we want to use—and I mean *exact*. If a letter of the name doesn't have the correct casing, it won't execute, and no warning will be raised. This is the most common beginner's error that is made, so pay attention. For Visual Scripting, we will add a special kind of node, but we will discuss that after the C# version. There are lots of events or messages to listen to in Unity, so let's start with the most common one—`Update`. This event will tell you when Unity wants you to update your object, depending on the purpose of your behavior; some don't need them. The `Update` logic is usually something that needs to be executed constantly—to be more precise, in every frame. Remember that every game is like a movie—a sequence of images that your screen switches through fast enough to look like we have continuous motion. A common action to perform in the `Update` event is to move objects a little bit, and by doing this, every frame will make your object constantly move. We will learn about the sorts of things we can do with `Update` and other events or messages later. Now, let's focus on how to make our component at least listen to this event. Actually, the base script already comes with two event functions that are ready to use, one being `Update` and the other one `Start`. If you are not familiar with the concept of methods in C#, we are referring to the snippet of code in the following screenshot, which is already included in our script. Try to find it in yours:

```
// Update is called once per frame
void Update()
{
|
}
```

Figure 5.24: A function called *Update*, which will be executed with every frame

You will notice a (usually) green line of text (depending on the IDE) above the `void Update()` line—this is called a **comment**. These are basically ignored by the compiler. They are just notes that you can leave to yourself and must always begin with `//`, preventing Unity from trying to execute them and failing. We will use this to temporarily disable lines of code later. Now, to test whether the `Update` method actually works, let's add an instruction to be executed all the time. There's no better test function than `print`. This is a simple instruction that tells Unity to print a message to the console, where all kinds of messages can be seen by the developers to check whether everything is properly working. The user will never see these messages. They are similar to the classic log files that developers sometimes ask you for when something goes wrong in the game and you are reporting an issue. In order to test events in C# using functions, follow these steps:

1. Open the script by double-clicking on it.
2. To test, add `print("test");` within the event function. In the following screenshot, you can see an example of how to do that

in the `Update` event. Remember to write the instruction exactly, including the correct casing, spaces, and quote symbols:

```
void Update()
{
    print("test");
}
```

Figure 5.25: Printing a message in all the frames

3. Save the file, go to Unity, and play the game.

Remember to save the file before switching back to Unity from the IDE. This is the only way that Unity knows your file has changed. Some IDEs, such as Rider, save the file automatically for you, but I don't recommend you use auto-save, at least in big projects. You don't want accidental recompilations of unfinished work—that takes too long in projects with lots of scripts.

4. Look for the **Console** tab and select it. This is usually found next to the **Project View** tab. If you can't find it, go to **Window | General | Console**, or press **Ctrl + Shift + C** (**Command + Shift + C** on macOS).
5. You will see a new printed message, saying `"test"`, in every frame on the **Console** tab. If you don't see this, remember to

save the script file before playing the game.

6. You might see a single message but with a number increasing to its right; that means the same message appears several times. Try clicking the **Collapse** button of the **Console** to change that behavior.
7. Let's also test the `Start` function. Add `print("test Start");` to it, save the file, and play the game. The full script should look as follows:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MyFirstScript : MonoBehaviour
{
    [SerializeField]
    private float speed;

    void Start()
    {
        print("test Start");
    }

    void Update()
    {
        print("test");
    }
}
```

Figure 5.26: The script that tests the Start and Update functions

If you check the console now and scroll all the way up, you will see a single "test Start" message and lots of "test" messages following it. As you can guess, the `Start` event tells you that the `GameObject` is created and allows you to execute the code that needs to happen just once at the beginning of its lifetime. For the `void Update()` syntax, we will say to Unity that whatever is contained within the brackets below this line is a function that will be executed in all the frames. It is important to put the `print` instruction *inside* the `Update` brackets (the ones inside the brackets of the class). Also, the `print` function expects to receive a value to print inside its parenthesis, called an argument or parameter. In our example, we want to print simple text, and in C#, it must be enclosed with quotation marks. Finally, all instructions inside functions such as **Update** or **Start** *must* end with a semicolon. Here, I challenge you to try to add another event called `OnDestroy`, using a `print` to discover when it executes. A small suggestion is to play and stop the game and look at the bottom of the console to test this one. For advanced users, you can also use breakpoints if your IDE allows you to do that. **Breakpoints** allow you to freeze Unity completely before executing a specific code line to see how our field's data changes over time and to detect errors. Here, I will show you the steps to use breakpoints in Rider, but the Visual Studio version should be similar:

1. Install the Unity package belonging to your IDE if not already installed. Check the **Package Manager** for the **JetBrains Rider Editor** package. In the case of Visual Studio, install the **Visual Studio Editor** package.
2. Click on the vertical bar at the left of the line where you want to add the breakpoint:



Figure 5.27: A breakpoint in the print instruction

3. Go to Run | Attach to Unity Process. If you are using Visual Studio, go to Debug | Attach Unity Debugger:

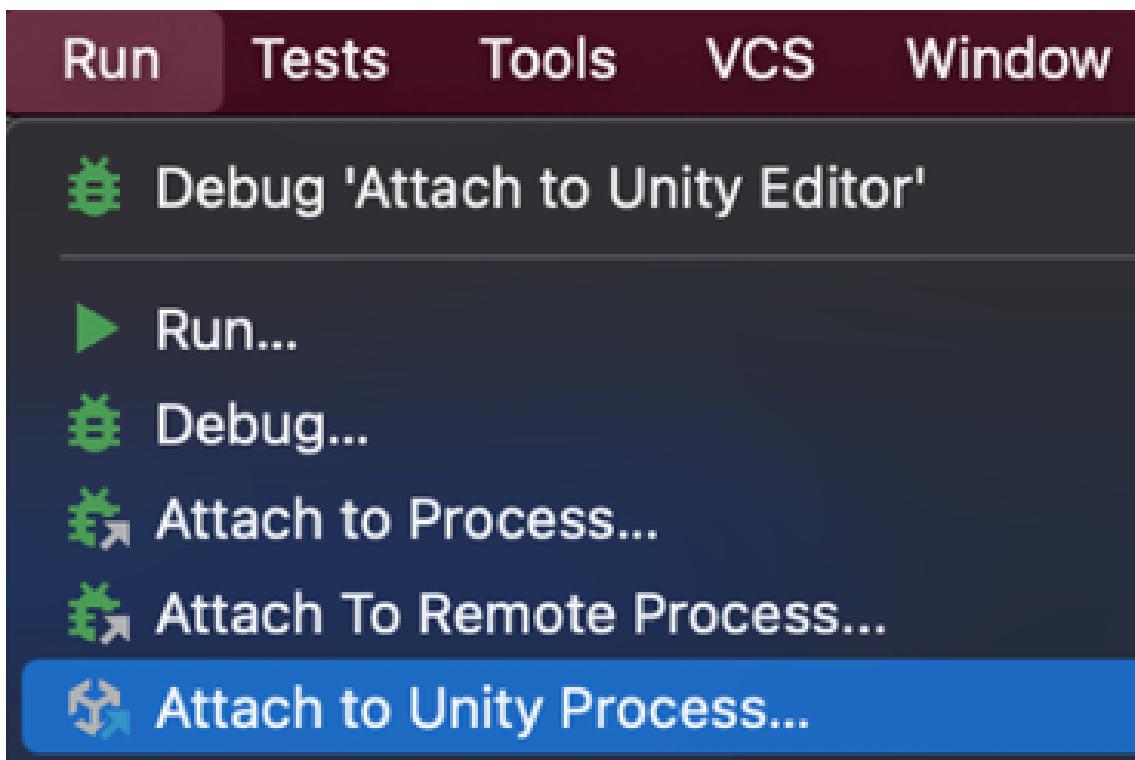


Figure 5.28: Attacking our IDE with a Unity process

4. From the list, look for the specific Unity instance you want to test. The list will show other opened editors or running debugging builds if there are any.
5. If this doesn't work, check if the editor is in debug mode by looking at the bug icon at the bottom-right part of the editor. If

the bug looks blue with a checkbox, then it is OK, but if it looks gray and crossed out, click it and click **Switch to debug mode**:

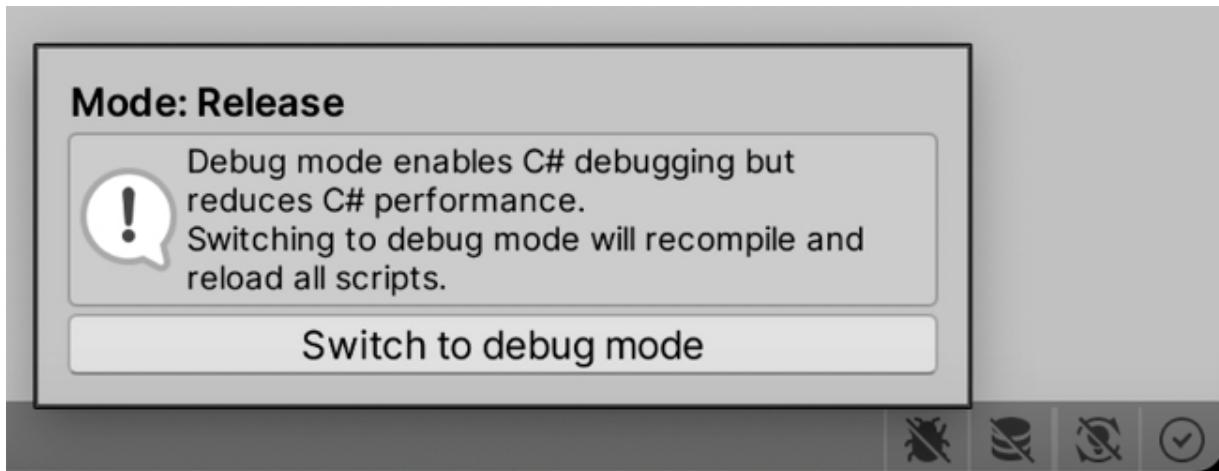


Figure 5.29: Changing from release mode to debug mode

Stopping the debugging process won't close Unity. It will just detach the IDE from the editor. Remember you can press the **Continue** button in Visual Studio (and the equivalent in other IDEs) to continue the game execution without detaching the debugger. Now, let's explore the Visual Scripting equivalent of using events and instructions.

Events and instructions in Visual Scripting

The same concept of events and instructions remains in Visual Scripting, but of course, this will be done with nodes in the graph. Remember that a node represents an instruction of the graph, and we can connect them to chain the effects of each instruction. In order to add events and the print instruction on our graph, do the following:

1. Open the **Visual Script Graph** (double-click the Visual Script asset).

2. Right-click the **On Start** and **On Update** nodes that are created by default, and then click **Delete**. Even if those events are the ones we need, I want you to see how to create them from scratch:

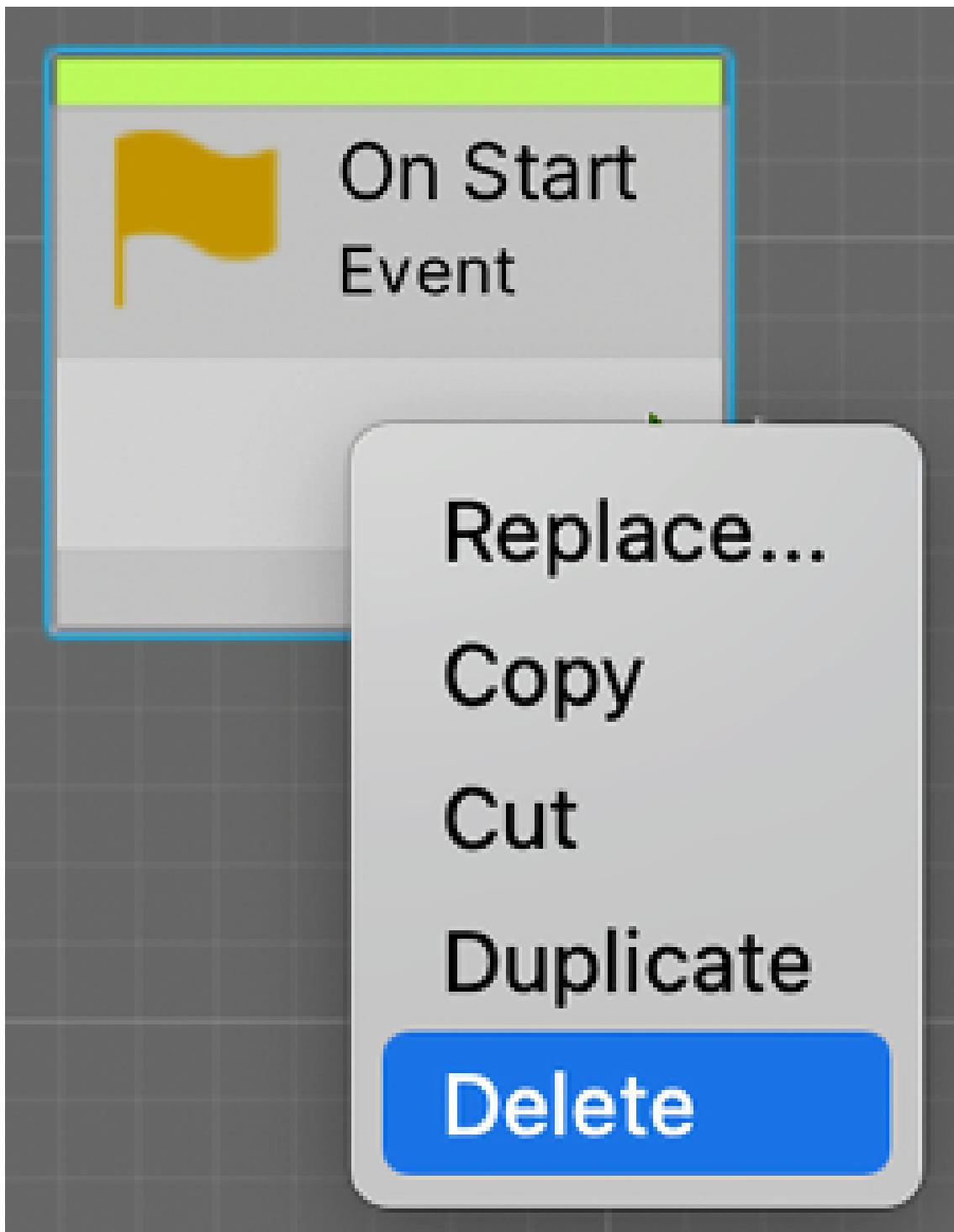


Figure 5.30: Deleting nodes

3. Right-click in any empty space of the **Graph** and type `start` inside the **Search** box. It can take a while the first time.

4. Select the **On Start** element in the list with the green checkbox to its left. In this case, I knew this was an event because I was aware of it, but usually, you will recognize it as an event because it won't have input pins (more on that in the next steps):

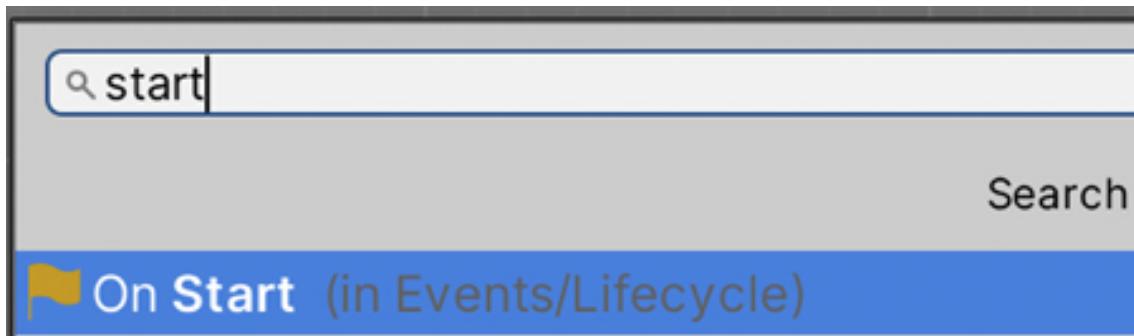


Figure 5.31: Searching the *On Start* event node

5. Drag the white arrow at the right of the event node, also known as the Output Flow Pin, and release the mouse button in any empty space.
6. In the **Search** box, search for the `print` node, and select the one that says **Mono Behaviour:Print**. This means that when the **On Start** event happens, the connected node will be executed—in this case, **print**. This is how we start to chain instructions to events:

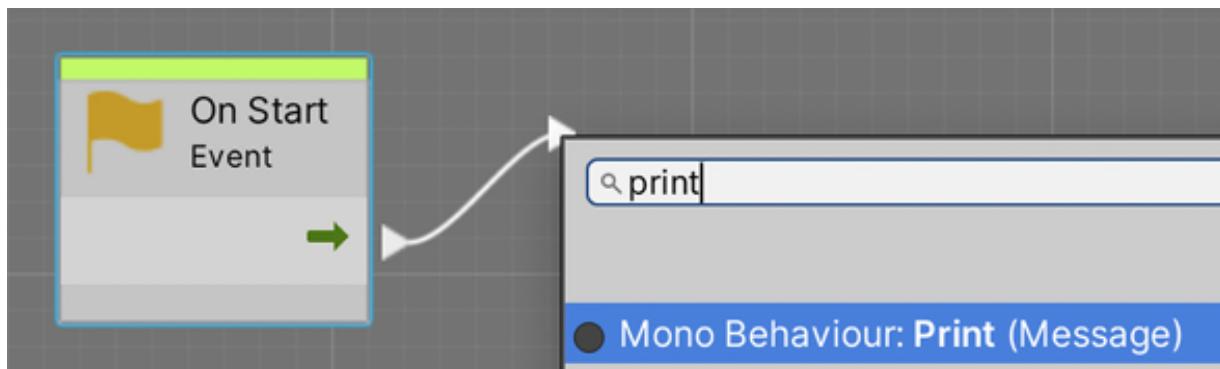


Figure 5.32: Creating a `print` node connected to the event

- Drag the empty circle at the left of the **Message** input pin of the **Print** node, and release it in any empty space. This pin is marked with a circle, indicating that it is a parameter pin, containing data that will be used when executing the pin. The flow pins, the ones with a green arrow, represent the order in which the nodes will be executed.
- Select the **String Literal** option, which will create a node to allow us to specify the message to print:

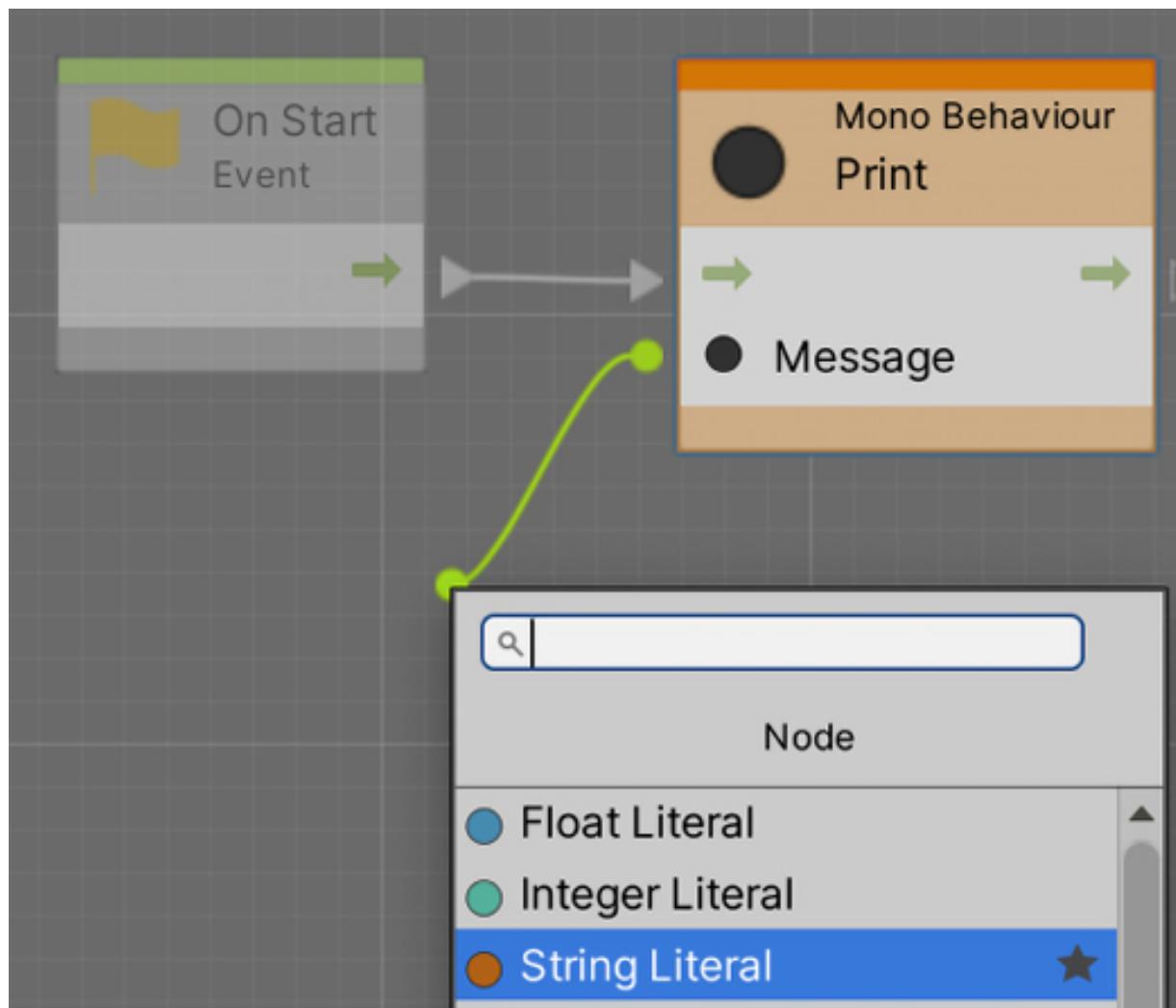


Figure 5.33: Creating a string literal node

1. In the empty white box, write the message to be printed:

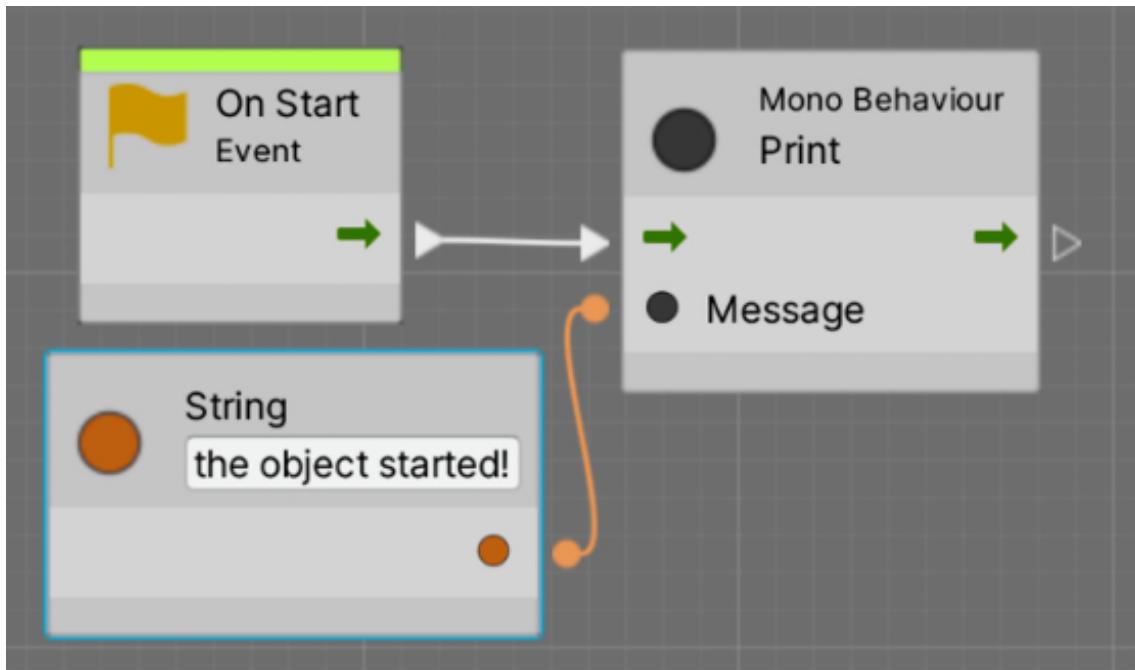


Figure 5.34: Specifying the message to print

2. Play the game, and see the message printed in the console. Be sure you have only the Visual Scripting version in the scene to avoid confusing the message in the console with the C# version. You can also use different message texts in the Visual Scripts to be sure which ones are really executing.

Now we have the same behavior we previously coded using C#, but now using the Visual Scripting tool in Unity. You can chain more actions to the **On Start** by dragging the pin to the right (Flow Output Pin) of the **Print** node, and chaining new nodes, but we will do that later. Now that we have our scripts doing something, let's make the instructions use the fields we created so that the scripts use their configurations.

Using fields in instructions

We have created fields to configure our components' behavior, but we have not used them so far. We will create meaningful components in the next chapter, but one thing we will often need is

to use the fields we have created to change the behavior of the object. So far, we have no real use of the `speed` field that we created. However, following the idea of testing whether our code is working (also known as debugging), we can learn how to use the data inside a field with a function to test whether the value is the expected one, changing the output of `print` in the console according to the field's value. In our current C# script, our `speed` value doesn't change during runtime. However, as an example, if you are creating a life system with shield damage absorption and you want to test whether the reduced damage calculation is working properly, you might want to print the calculation values to the console and check whether they are correct. The idea here is to replace the fixed message inside the `print` functions with a field. When you do that, `print` will show the field's value in the console. So if you set a value of `5` in `speed` and you print it, you will see lots of messages saying `5` in the console, and the output of the `print` function is governed by the field. To test this, your `print` message within the `Update` function should look as follows:

```
[SerializeField]  
private float speed;  
  
void Update()  
{  
    print(speed);  
}
```

Figure 5.35: Using a field as a print function parameter

As you can see, we just put the name of the field without quotation marks. If you use quotation marks, you will print a "speed" message. In other scenarios, you can use this `speed` value within some moving functions to control how fast the movement will be, or you can perhaps create a field called "fireRate" (fields use **camel case** instead of Pascal case, with the first letter being in lowercase) to control the cooldown time between one bullet and the next:

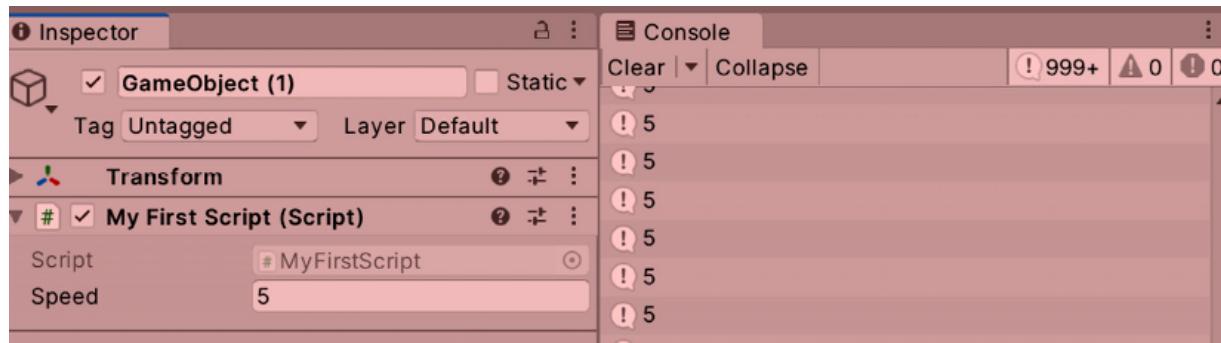


Figure 5.36: Printing the current speed

Now, to make the Visual Script graph print the value of the **speed** variable we created in the **Variables** component, let's do the following.

1. Open the Visual Scripting graph asset (double-click it).
2. In the panel on the left, select the **Object** tab to display all the variables our object has—essentially the ones we defined in the **Variables** component previously.
3. Drag the **speed** variable, using the two lines to the left of the variable box, to any empty area of the graph. This will create a **GetVariable** node in the graph to represent the variable.

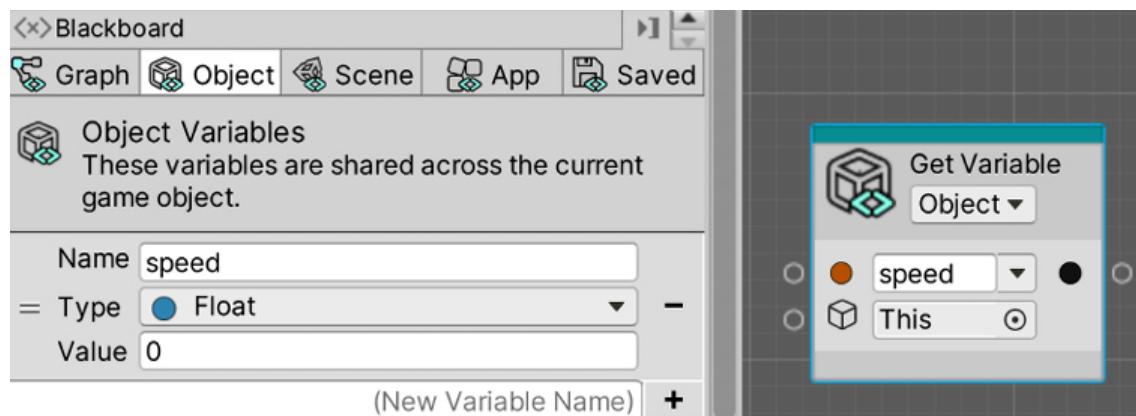


Figure 5.37: Dragging variables to the graph to be used in the nodes

4. Drag the empty circle at the right of the **Get Variable** node to the circle at the left of the **Message** input pin of the **Print** node. This will replace the previous connection to the **String Literal** node. This node doesn't have **Input** or **Output** flow nodes (the green arrow ones), as they are data-only nodes that provide data to other nodes. In this case, when `Print` needs to execute, it will execute `Get Variable` to get the text to read:

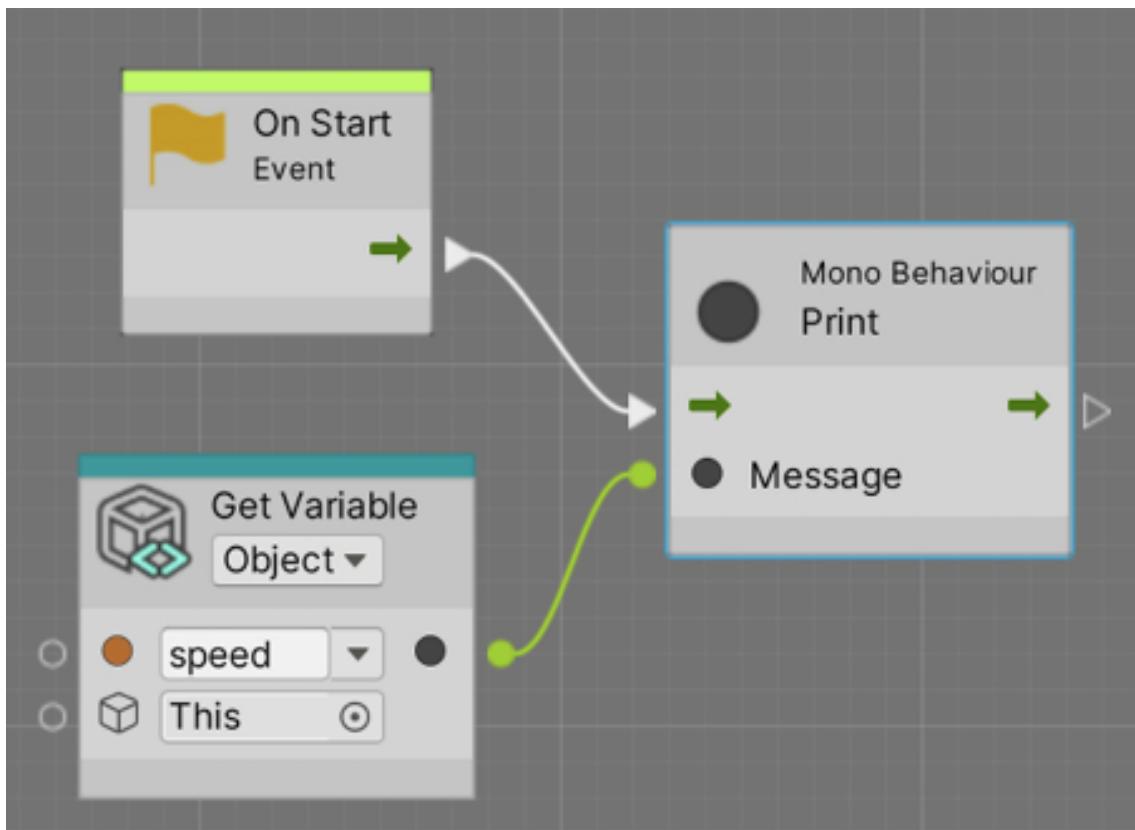


Figure 5.38: Connecting the speed variable to the print node

5. Right-click on the **String Literal** node and delete it.
6. Play the game and observe.

With all this, we now have the necessary tools to start creating actual components. Before moving on, let's recap some of the common errors that you will likely encounter if this is your first time creating scripts in C#.

Common beginner C# script errors

The Visual Scripting scripts are prepared in a way in which you make fewer errors, not allowing you to write incorrect syntax like C# script does. If you are an experienced programmer, I bet you are quite familiar with them, but let's recap the common errors that will make you lose lots of time when you start out with C# scripting. Most of them are caused by not copying the shown code *exactly*. If you have an error in the code, Unity will show a red message in the console and won't allow you to run the game, even if you are not using the script. So never leave anything unfinished. Let's start with a classic error, a missing semicolon, which has resulted in many programmer memes and jokes. All fields and most instructions inside functions (such as `print`), when called, need to have a semicolon at the end. If you don't add a semicolon, Unity will show an error, such as the one in the screenshot on the left in *Figure 5.39*, in the console. You will also notice that this also has an example of bad code, where the IDE shows a red icon, suggesting something is wrong in that place:

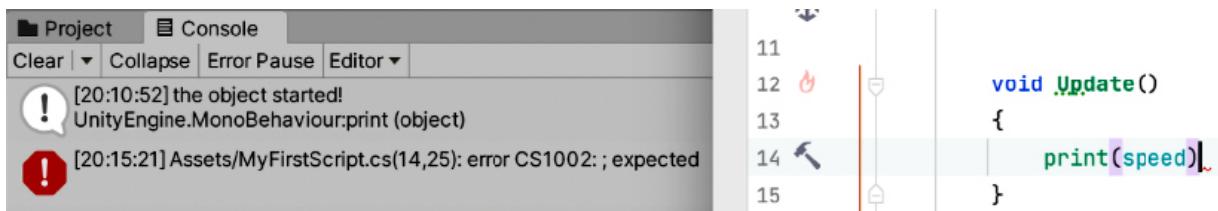


Figure 5.39: An error in the print line hinted by the IDE and the Unity console

You will notice that the error shows the exact script (`MyFirstScript.cs`), the exact line of code (14, in this case), and usually, a descriptive message—in this case, `; expected`—as a way to specify that the instruction ends there, so the compiler can process the next instruction as a separate one. You can simply double-click the error and Unity will open the IDE highlighting the problematic line. You can even click on the links in the stack to jump to the line

of the stack that you want. I already mentioned why it is important to use the *exact* case for every letter of the instruction. However, based on my experience of teaching beginners, I need to stress this particular aspect more. The first scenario where this can happen is in instructions. In the following screenshots, you can see how a badly written `print` function looks—that is, the error that the console will display and how the IDE will suggest that there is something wrong. First, in the case of Rider, the instruction is colored red, saying that the instruction is not recognized (in Visual Studio, it will show a red line instead). Then, the error message says that `Print` does not exist in the current context, meaning that Unity (or C#, actually) does not recognize any instruction named `Print`. In another type of script, `Print` in uppercase may be valid, but not in regular components, which is why the “**in the current context**” clarification exists:

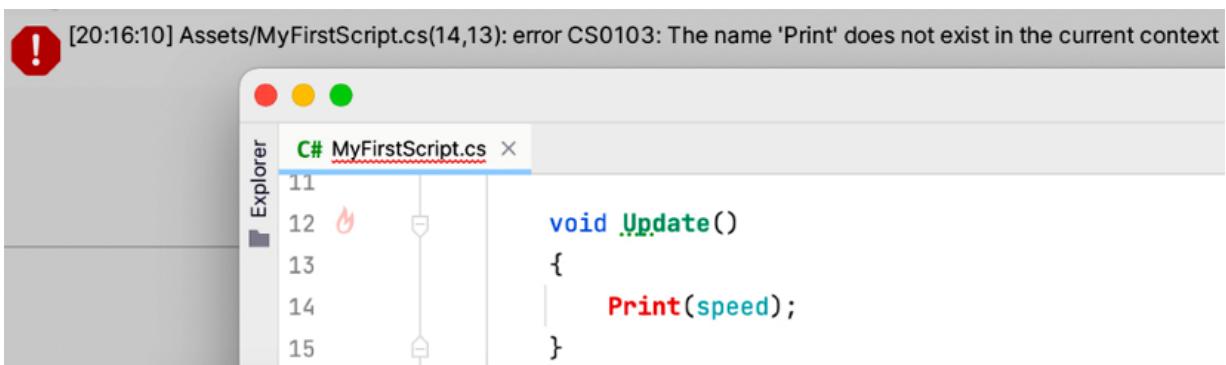


Figure 5.40: Error hints when writing an instruction incorrectly

Now, if you write an event with the wrong casing, the situation is worse. You can create functions such as `Start` and `Update` with whatever name you want for other purposes. Writing `update` or `start` is perfectly valid, as C# will think that you are going to use those functions not as events but as regular functions. So no error will be shown, and your code will just not work. Try to write `update` instead of `Update` and see what happens:

```

void update()
{
    print(speed);
}

```

Figure 5.41: The wrong casing in the Update function will compile the function but won't execute it

Another error is to put instructions outside the function brackets, such as inside the brackets of the class or outside them. Doing this will give no hint to the function as to when it needs to execute. So a `print` function outside an `Event` function makes no sense, and it will show an error such as the ones in the following *Figures 5.42* and *5.43*. This time, the error is not super-descriptive. C# expects you to create a function or a field—the kind of structure that can be put directly inside a class:

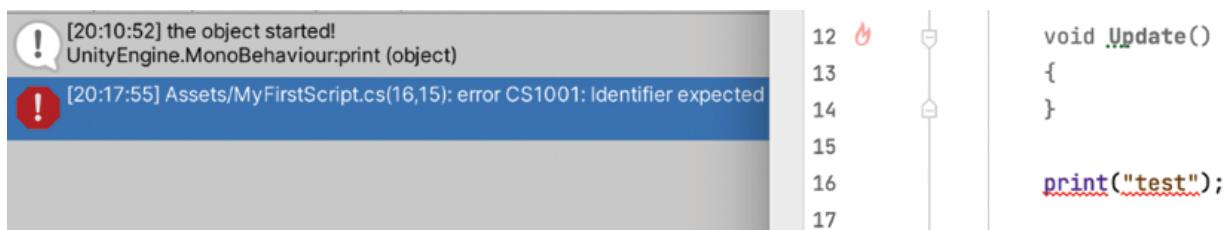
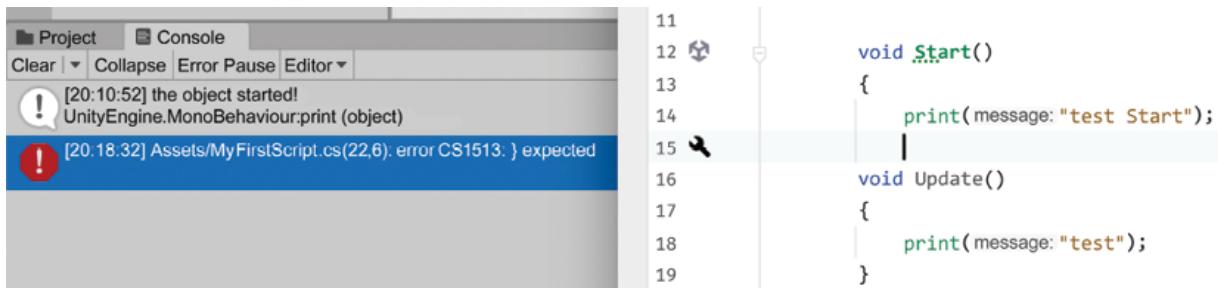


Figure 5.42: Misplaced instruction or function call

Finally, another classic mistake is to forget to close open brackets. If you don't close a bracket, C# won't know where a function finishes and another starts or where the class function ends. This may sound redundant, but C# needs that to be perfectly defined. In the following screenshots, you can see how this would look:



The screenshot shows the Unity Editor's IDE interface. The top bar has tabs for 'Project' and 'Console'. Below that is a toolbar with 'Clear', 'Collapse', 'Error Pause', and 'Editor'. The 'Console' tab is selected. The console window displays two messages: '[20:10:52] the object started!' and 'UnityEngine.MonoBehaviour:print (object)'. Below the console is a blue status bar with a red exclamation mark icon and the text '[20:18:32] Assets/MyFirstScript.cs(22,6). error CS1513. } expected'. The code editor on the right shows the following C# script:

```
11
12 void Start()
13 {
14     print(message: "test Start");
15 }
16 void Update()
17 {
18     print(message: "test");
19 }
```

Figure 5.43: Missing closed brackets

Memory

So much time is lost looking for that missing semicolon at the end of the line. At first, it was extremely frustrating trying to figure out what was wrong in my code but, at the same time, very satisfying to find the solution and have it all working again. That frustration-satisfaction loop still happens to me, maybe with more complex issues, but it's all part of being a developer. The sooner you embrace it, the faster you will see improvement.

This one is a little bit difficult to catch because the error in the code is shown way after the actual error. This is caused by the fact that C# allows you to put functions inside functions (not used often), and so C# will detect the error later, asking you to add a closing bracket. However, as we don't want to put `Update` inside `Start`, we need to fix the error beforehand, at the end of `Start`. The error message will be descriptive in the console, but again, don't put the close bracket where the message suggests you do unless you are 100% sure that position is correct. You will likely face lots of errors aside from these ones, but they all work the same. The IDE will

show you a hint, and the console will display a message; you will learn how to resolve them with time. Just have patience, as every programmer experiences this. There are other kinds of errors, such as runtime errors, code that compiles but fails when being executed due to some misconfiguration, or the worst, logic errors, where your code compiles and executes with no error but doesn't do what you intended.

Summary

In this chapter, we explored the basic concepts that you will use while creating scripts. We discussed the concept of a script's assets and how the C# ones must inherit from `MonoBehaviour` to be accepted by Unity to create our own scripts. We also saw how to mix events and instructions to add behavior to an object and how to use fields in instructions to customize what they do. All of this was done using both C# and Visual Scripting. We just explored the basics of scripting to ensure that everyone is on the same page. However, from now on, we will assume that you have basic coding experience in some programming language, and you know how to use structures such as `if`, `for`, `array`, and so on. If not, you can still read through this book and try to complement the areas you don't understand with a C# introduction book, as you need. In the next chapter, we are going to start seeing how we can use what we have learned to create movement and spawning scripts.

6 Dynamic Motion: Implementing Movement and Spawning

Join our book community on Discord

<https://packt.link/unitydev>



In the previous chapter, we learned about the basics of scripting, so now let's create the first script for our game. We will see the basics of how to move objects through scripting using the `Transform` component, which will be applied to the movement of our player with the keyboard keys, the constant movement of bullets, and other object movements. Also, we will see how to create and destroy objects during the game, such as the bullets our player and enemy shoot and the enemy waves that will be generated during the game (also called enemy spawners). These actions can be used in several other scenarios, so we will explore a few in this chapter. In this chapter, we will examine the following scripting concepts:

- Implementing movement
- Implementing spawning
- Using the new Input System

We will start by scripting components to move our character through the keyboard, and then we will make our player shoot bullets. Something to consider is that we are going to first see the C# version and then show the Visual Scripting equivalent in each section.

Implementing movement

Almost every object in a game moves one way or another: the player character with the keyboard; the enemies through AI; the bullets that simply move forward; and so on. There are several ways of moving objects in Unity, so we will start with the simplest one, that is, through the `Transform` component. In this section, we will examine the following movement concepts:

- Moving objects through `Transform`
- Using input
- Understanding Delta Time

First, we will explore how to access the `Transform` component in our script to drive the player movement, to later apply movement based on the player's keyboard input. Finally, we are going to explore the concept of Delta Time to make sure the movement speeds are consistent on every computer. We are going to start learning about the `Transform` API to make a simple movement script.

Moving objects through Transform

`Transform` is the component that holds the translation, rotation, and scale of the object, so every movement system such as physics or pathfinding will affect this component. Sometimes, we want to move the object in a specific way according to our game by creating our own script, which will handle the movement calculations we need and modify `Transform` to apply them. One concept implied here is that components can alter other components. The main way of coding in Unity is to create components that interact with other components. Here, the idea is to create one that accesses another and tells it to do something: in this case, to move. To create a script that tells `Transform` to move, do the following:

1. Create and add a script called `PlayerMovement` to our character, like we did in the previous chapter. In this case, it would be the animated 3D model we downloaded previously named **Polyart_Mesh** (drag the 3D asset from the **Project** view to the scene). Remember to move the script to the **Scripts** folder after creation:

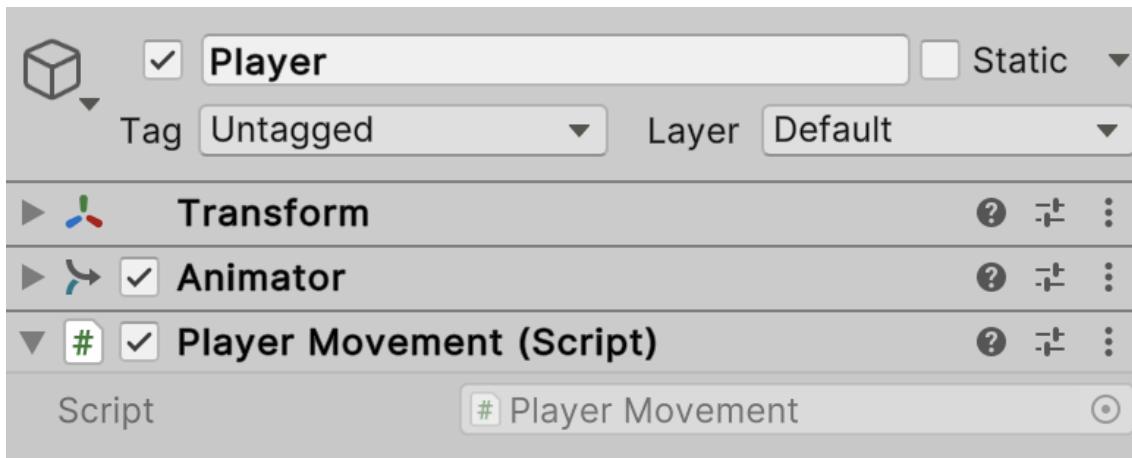


Figure 6.1: Creating a Player Movement script in the character

2. Double-click the created script asset to open an IDE to edit the code.
3. We are moving, and the movement is applied to every frame. So this script will use only the `Update` function or method, and we can remove `Start` (it is a good practice to remove the unused functions):

```
using UnityEngine;

public class PlayerMovement : MonoBehaviour
{
    void Update()
    {
    }
}
```

Figure 6.2: A component with just the Update event function

4. To move our object along its local forward axis (z axis), add the `transform.Translate(0, 0, 1);` line to the `Update` function, as shown in *Figure 6.3*:

Every component has access to a `Transform` field (to be specific, a getter) that is a reference to the `Transform` of the **GameObject** the component is placed in. Through this field, we can access the `Translate` function of the `Transform`, which will receive the offset to apply to the x, y, and z local coordinates.

```
using UnityEngine;

public class PlayerMovement : MonoBehaviour
{
    void Update()
    {
        transform.Translate(0, 0, 1);
    }
}
```

Figure 6.3: A simple Move Forward script

1. Save the file and play the game to see the movement. Ensure the camera is pointing at the character to properly see the effect of the script. To do that, remember you can select the Camera in the hierarchy and move and rotate it until the character falls inside the frustum.

Now that we have implemented a simple movement for the player, you will notice that the player is moving too fast. That's because we are using a fixed speed of 1 meter, and because `Update` is executing all frames, we are moving 1 meter per frame. In a standard 30 FPS game, the player will move 30 meters per second, which is too much, but our computer is perhaps running the game with way more FPS than that. We can control the player's speed by adding a `speed` field and using the value set in the editor instead of the fixed value of 1. You can see one way to do this in *Figure 6.4*, but remember the other options we discussed in *Chapter 5, Introduction to C# and Visual Scripting*:

```

public float speed;

void Update()
{
    transform.Translate(0, 0, speed);
}

```

Figure 6.4: Creating a speed field and using it as the z speed of the movement script

Now if you save the script to apply the changes and set the **Speed** of the player in the Editor, you can play the game and see the results. In my case, I used `0.1`, but you might need another value (more on this in the *Understanding Delta Time* section):



Figure 6.5: Setting speed of 0.1 meters per frame

Now, for the Visual Scripting version, first remember to not mix the C# and Visual Scripting versions of our scripts, not because it is not possible, but because we want to keep things simple for now. So, you can either delete the script from the player object and add the Visual Scripting version, or you can create two player objects and

enable and disable them to try both versions. I recommend creating one project for the C# version of the scripts and then creating a second project to experiment with the Visual Script version. The Visual Scripting Graph of this script will look like the following image:

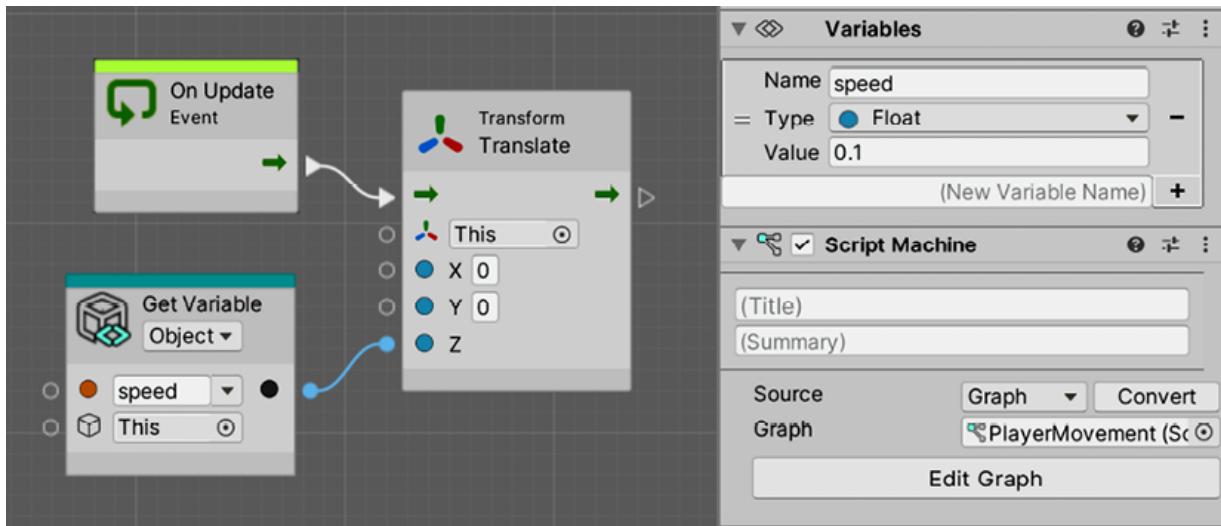


Figure 6.6: Setting a speed of 0.1 meters per frame

As you can see, we added a **Script Machine** component to our `Player` GameObject. Then, we pressed the **New** button in the **Script Machine** component to create a new **Graph** called `PlayerMovement`. We also created a **Float** variable called `speed` with the value of `0.1`. In the **Graph**, we added the **On Update** event node and attached it to the **Translate (X,Y,Z)** node of the **Transform**, which, similar to the C# version, will move along the local axes of the object. Finally, we connected the **Z** parameter pin of **Translate** to the `GetVariable` node representing the speed we created in the GameObject. If you compare this **Graph** with the code we used in the C# version, they are essentially the same `Update` method and `Translate` function. If you don't remember how to create this **Graph**, you can go back to *Chapter 5, Introduction to C# and Visual Scripting*, to recap the process. You will notice that the player will move automatically. Now let's see

how to execute the movement based on **player input** such as the keyboard and mouse.

Using Input

Unlike NPCs (non playable characters in a videogame), we want the player's movement to be driven by the users input, based on which keys they press when they play, the mouse movement, and so on. To know whether a certain key has been pressed, such as the *Up* arrow, we can use the `Input.GetKeyDown(KeyCode.W)` line, which will return a Boolean, indicating whether the key specified in the `KeyCode` enum is pressed, which is *W* in this case. This is usually the main setup for keyboard controllers in 3D videogames. We can combine the `GetKey` function with an `If` statement to make the translation execute when the key is pressed. Let's start by implementing the keyboard movement by following these steps:

1. Make the forward movement execute only when the W key is pressed with the code, as shown in the next screenshot:

```
void Update()
{
    if (Input.GetKey(KeyCode.W))
    {
        transform.Translate(0, 0, speed);
    }
}
```

Figure 6.7: Conditioning the movement until the W key is pressed

2. We can add other movement directions with more `if` statements to move backward and A and D to move left and right, as shown in the following screenshot. Notice how we used the minus sign to inverse the speed when we needed to move in the opposite axis direction:

```
if (Input.GetKey(KeyCode.W)) { transform.Translate(0, 0, speed); }
if (Input.GetKey(KeyCode.S)) { transform.Translate(0, 0, -speed); }
if (Input.GetKey(KeyCode.A)) { transform.Translate(-speed, 0, 0); }
if (Input.GetKey(KeyCode.D)) { transform.Translate(speed, 0, 0); }
```

Figure 6.8: Checking the W, A, S, and D key pressure

3. In case you also want to consider the arrow keys, you can use an `OR` inside `if`, as shown in the following screenshot:

```
if (Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.UpArrow)) { transform.Tra
if (Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.DownArrow)) { transform.T
if (Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.LeftArrow)) { transform.T
if (Input.GetKey(KeyCode.D) || Input.GetKey(KeyCode.RightArrow)) { transform.
```

Figure 6.9: Checking the W, A, S, D, and arrow key pressure

4. Save the changes and test the movement in **Play** mode.

With this lines of code, we have implemented basic movement using WASD keys. Something to take into account is that, first, we have another way to map several keys to a single action by configuring the Input Manager—a place where action mappings can be created. Second, at the time of writing this, Unity has released a new Input System that is more extensible than this one. For now, we will use this one because it is simple enough to make our introduction to scripting with Unity easier, but in games with complex input, it is recommended to look for more advanced tools. Now, for the Visual Scripting version, the graph will look like this:

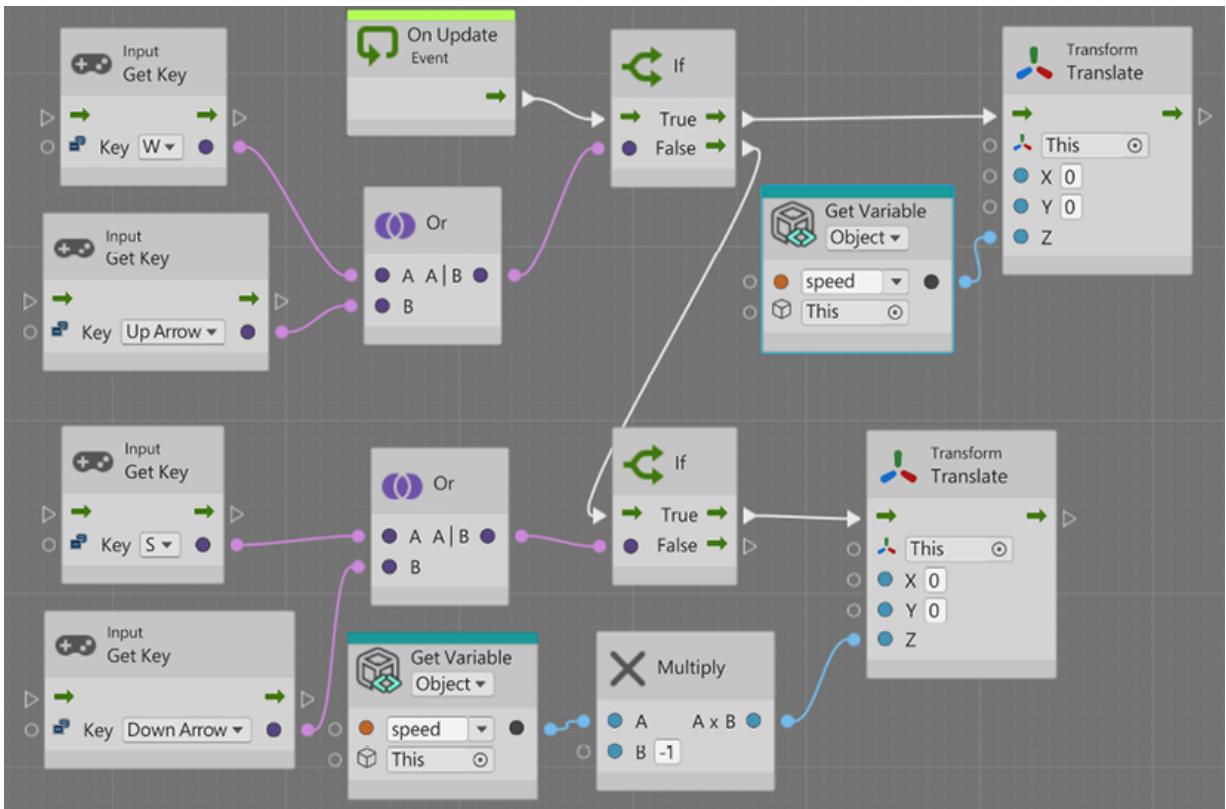


Figure 6.10: Input movement in Visual Scripting

As you can see, the graph has grown in size considerably compared to the C# version, which serves as an example of why developers prefer to code instead of using visual tools. Of course, we have several ways to split this graph into smaller chunks and make it more readable. Additionally, I needed to squeeze the nodes together to be in the same image. In the graph, we only see the example graph to move forward and backward, but you can easily extrapolate the necessary steps for lateral movement based on this one. As usual, you can also check the GitHub repository of the project to see the completed files. Notice all the similarities to the C# version; we chained **If** nodes to the **On Update** event node in a way that if the first **If** node condition is true, it will execute the **Translate** in the player's forward direction. If that condition is false, we chained the **False** output node to another **If** that checks the pressure of the other keys, and in that case, we moved backward using the **Multiply** (Scalar) node to inverse the speed. You can see nodes like

`If` that have more than one **Flow Output** pin to branch the execution of the code. You can also notice the usage of the **GetKey (Key)** node, the Visual Scripting version of the same **GetKey** function we used previously. When looking at this node in the **Search** box, you will see all the versions of the function, and in this case, we selected the **GetKey(Key)** version; the one that receives a name (string) works differently, and we are not covering that one:

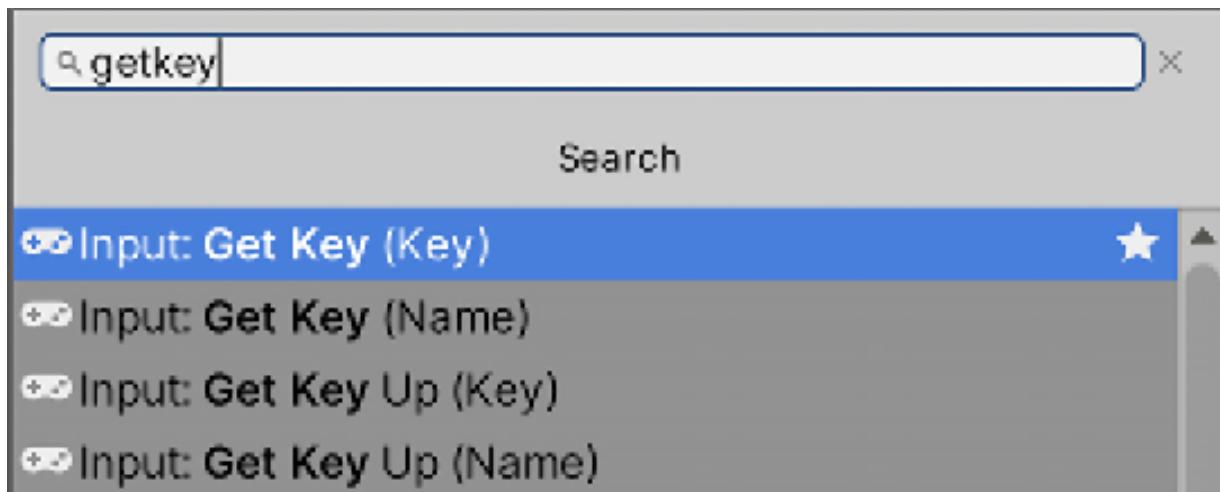


Figure 6.11: All versions of Input GetKey

We also used the `or` node to combine the two **GetKey (Key)** functions into one condition to give to the `If`. These conditional operators can be found in the **Logic** category of the **Search** box:

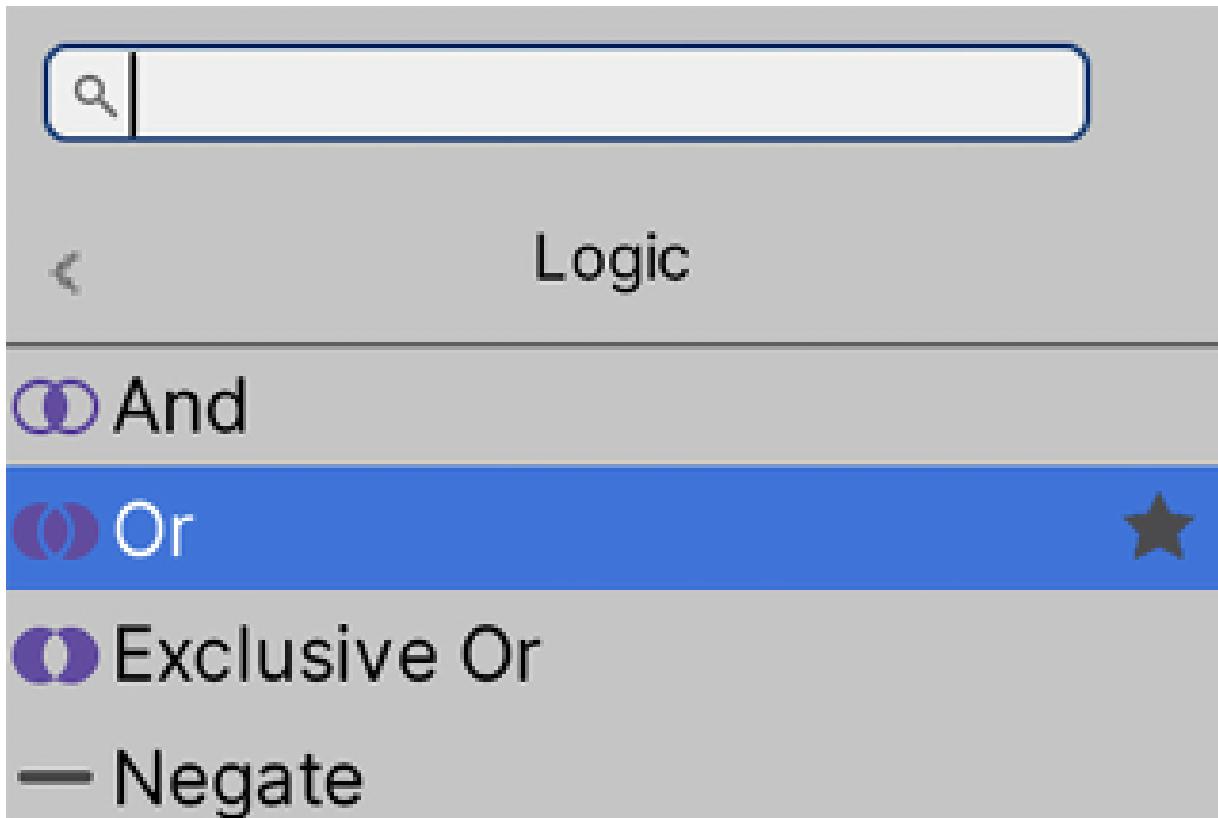


Figure 6.12: The Boolean Logic operators

One thing to highlight is the usage of the **Multiply** node to multiply the value of the speed variable by -1 . We needed to create a **Float Literal** node to represent the -1 value. Next, surely all programmers will notice some limitations regarding how we used the `If` node's `True` and `False` output pins, but we will address that in a moment. Finally, consider that this implementation has the problem of blocking the second input read if the first is successful; we will discuss a way to fix this when we add rotation to our character later in this section. Now, let's implement the mouse controls. In this section, we will only cover rotation with mouse movement; we will shoot bullets in the next section: *Implementing spawning*. In the case of mouse movement, we can get a value saying how much the mouse has moved both horizontally and vertically. This value isn't a Boolean but a number: a type of input usually known as an **axis**. The value of an axis will indicate the intensity of the movement, and the sign of that number will indicate

the direction. For example, if Unity's "Mouse X" axis says `0.5`, it means that the mouse moved to the right with a moderate speed, but if it says `-1`, it moved quickly to the left, and if there is no movement, it will say `0`. The same goes for sticks in gamepads; the axis named **Horizontal** represents the horizontal movement of the left stick in common joysticks, so if the player pulls the stick fully to the left, it will say `-1`. We can create our own axes to map other common joystick pressure-based controls, but for our game, the default ones are enough. To detect mouse movement, follow these steps:

1. Use the `Input.GetAxis` function inside `Update`, next to the movement `if` statements, as shown in the following screenshot, to store the value of this frame's mouse movement into a variable:

```
float mouseX = Input.GetAxis("Mouse X");
```

Figure 6.13: Getting the horizontal movement of the mouse

2. Use the `transform.Rotate` function to rotate the character. This function receives the degrees to rotate in the x, y, and z axes. In this case, we need to rotate horizontally, so we will use the mouse movement value as the y-axis rotation, as shown in the next screenshot:

```
float mouseX = Input.GetAxis("Mouse X");
transform.Rotate(0, mouseX, 0);
```

Figure 6.14: Rotating the object horizontally based on mouse movement

3. If you save and test this, you will notice that the character will rotate but very quickly or slowly, depending on your computer. Remember, this kind of value needs to be configurable, so let's create a `rotationSpeed` field to configure the speed of the player in the editor:

```
public float speed;  
public float rotationSpeed;
```

Figure 6.15: The speed and rotation speed fields

4. Now we need to multiply the mouse movement value by the speed, so, depending on the `rotationSpeed`, we can increase or reduce the rotation amount. As an example, if we set a value of `0.5` in the rotation speed, multiplying that value by the mouse movement will make the object rotate at half the previous speed, as shown in the following screenshot:

```
float mouseX = Input.GetAxis("Mouse X");  
transform.Rotate(0, mouseX * rotationSpeed, 0);
```

Figure 6.16: Multiplying the mouse movement by the rotation speed

5. Save the code and go back to the editor to set the rotation speed value. If you don't do this, the object won't rotate because the default value of the float type fields is `0`:



Figure 6.17: Setting the rotation speed

The Visual Scripting additions to achieve rotation will look like this:

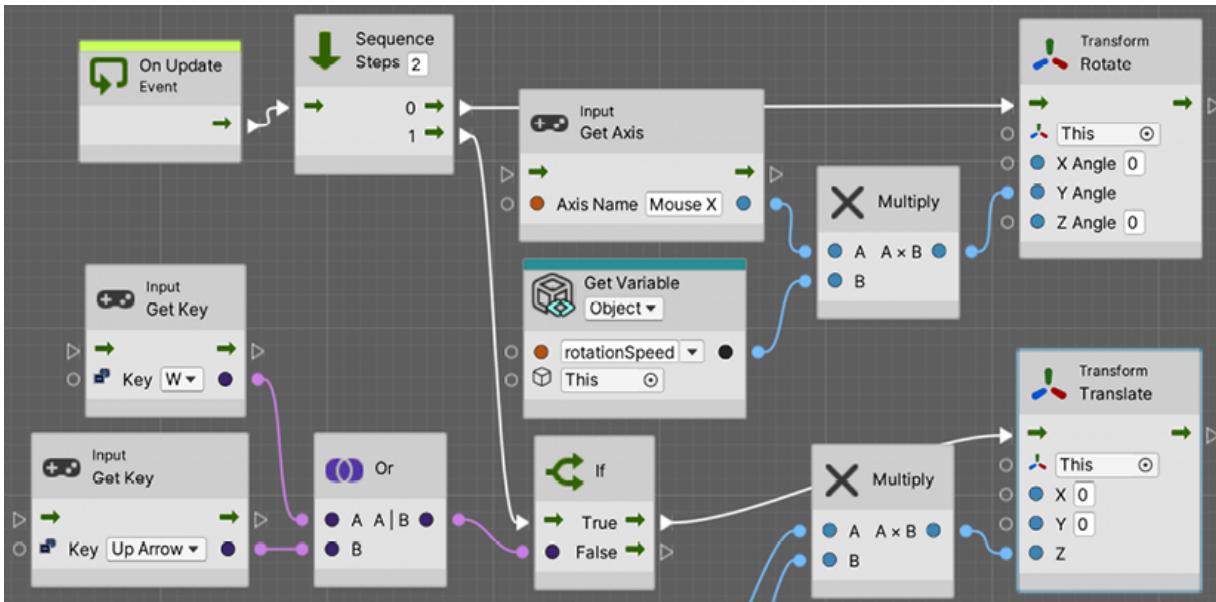


Figure 6.18: Rotating in Visual Scripting

The first thing to notice here is the usage of the **Sequence** node. An output pin can only be attached to one other node, but in this case, **On Update** needs to do two different things, to rotate and to move, each one being independent of the other. **Sequence** is a node that will execute all its output pins one after the other, regardless of the results of each one. You can specify the number of output pins in the **Steps** input box; in this example, two is enough. In the output

pin 0, the first one, we added the rotation code, which is pretty self-explanatory given it is essentially the same as the movement code with slightly different nodes (**Rotate** (X, Y, Z) and **GetAxis**). Then, to Output Pin 1, we attached the `If` that checks the movement input—the one we did at the beginning of this section. This will cause the rotation to be executed first and the movement second. Regarding the limitation we mentioned before, it's basically the fact that we cannot execute both **Forward** and **Backward** movements, given that if the forward movement keys are pressed, the first `If` will be true. Because the backward key movement is checked in the false output pin, they won't be checked in such cases. Of course, as our first movement script, it might be enough but consider the lateral movement. If we continue the `If` chaining using `True` and `False` output pins, we will have a scenario where we can only move in one direction. So we cannot combine, for example, `Forward` and `Right` to move diagonally. A simple solution to this issue is to put the `If` nodes in the sequence instead of chaining them, so all the `If` nodes are checked, as the original C# did. You can see an example of this in the next image:

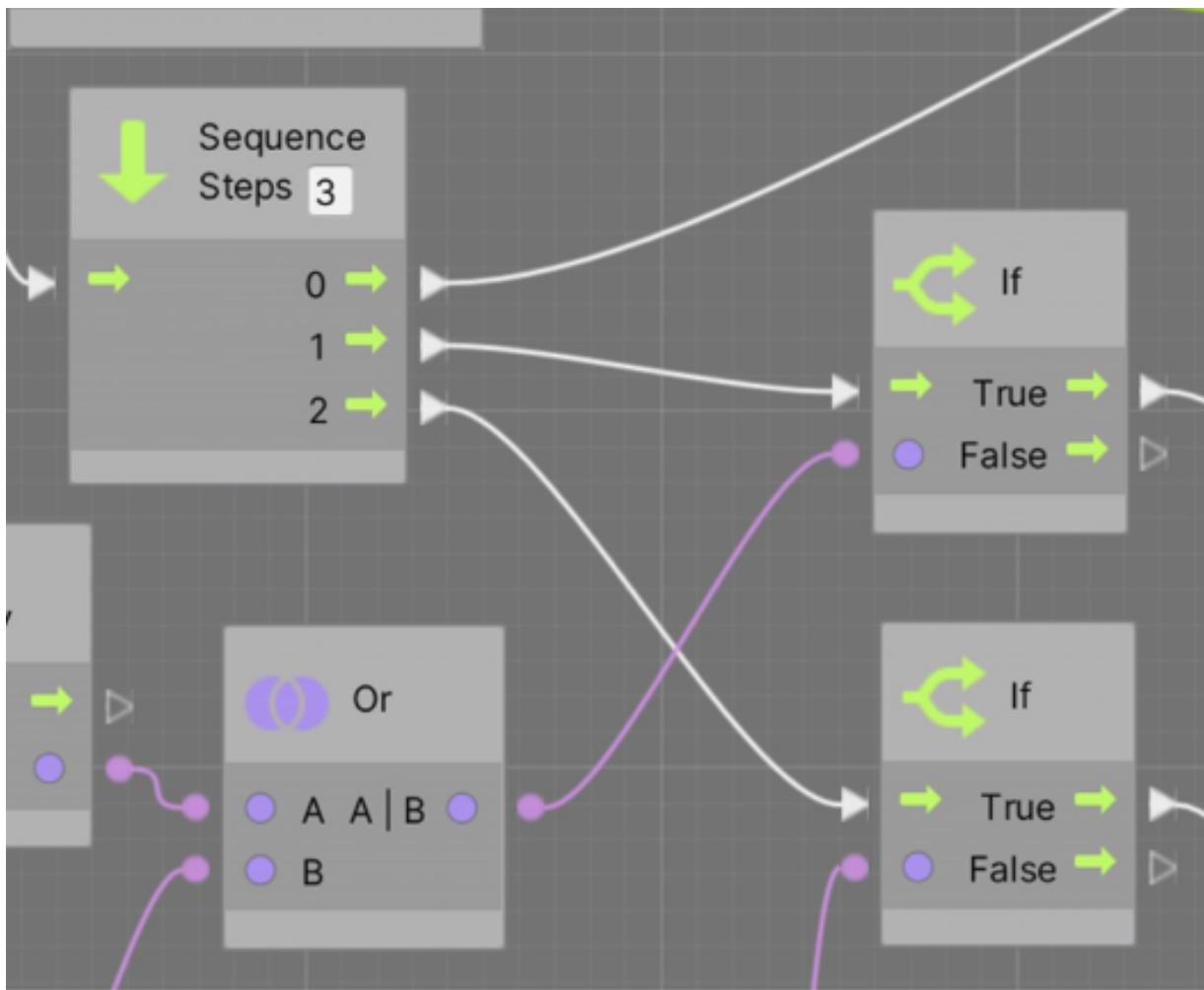


Figure 6.19: Sequencing Ifs

Something to consider here is that the connection of the `If` nodes and any kind of node can be removed by right-clicking the circle pins on both ends of the line that connects them.

Memory

Making basic input scripting tends to be easy, but the most difficult thing to do is to make intuitive and engaging input. Aside from adhering to the standards for the user to quickly adapt to your game, like jumping with the Space key or the A button in a gamepad, I recommend experimenting with the full expressivity of controls. That

will allow to better understand when it feels natural to use things like drag-and-drop, charging a punch when holding the key and execute the punch when releasing it, or using the triggers to control progressive things like the acceleration/braking of a car. This is even more interesting when talking about AR/VR experiences, where the possibilities are endless.

Now that we have completed our movement script, we need to refine it to work in every machine by exploring the concept of Delta Time.

Understanding Delta Time

Unity's **Update** loop executes as fast as the computer can. While you can set in Unity the desired frame rate, achieving it depends exclusively on your computer's capabilities, which are influenced by various factors, not just hardware. This means you can't always guarantee a consistent FPS. You must code your scripts to handle every possible scenario. Our current script is moving at a certain speed per frame, and the *per frame* part is important here. We have set the movement speed to 0.1, so if my computer runs the game at 120 FPS, the player will move 12 meters per second. Now, what happens on a computer where the game runs at 60 FPS? As you may guess, it will move only 6 meters per second, making our game have inconsistent behavior across different computers. This is where Delta Time saves the day. **Delta Time** is a value that tells us how much time has passed since the previous frame. This time depends a lot on our game's graphics, number of entities, physics bodies, audio, and countless aspects that will dictate how fast your computer can process a frame. As an example, if your game runs at 10 FPS, it means that, in a second, your computer can process the `Update` loop 10 times, meaning that each loop takes approximately 0.1 seconds; in the frame, Delta Time will provide that value. In the following diagram, you can see an example of 4 frames taking different times to process, which can happen in real-life cases:

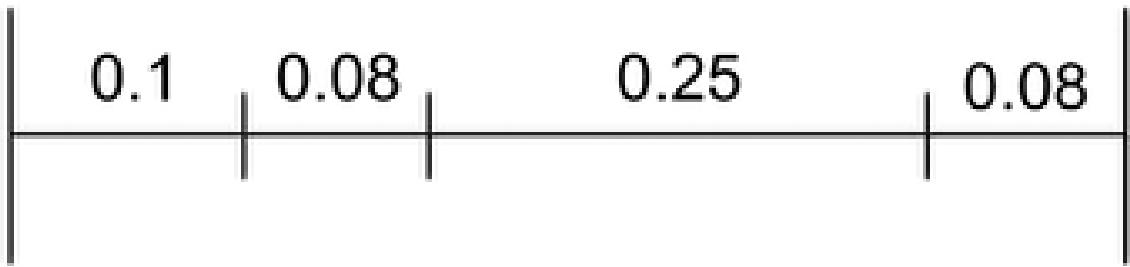


Figure 6.20: Delta Time values varying in different frames of the game

Here, we need to code in such a way as to change the *per frame* part of the movement to *per second*; we need to have consistent movement per second across different computers. One way to do that is to move proportionally to the Delta Time: the higher the Delta Time value, the longer that frame is, and the further the movement should be to match the real time that has passed since the last update. We can think about our `speed` field's current value in terms of `0.1` meters per second; our Delta Time saying `0.5` means that half a second has passed, so we should move half the speed, `0.05`. After two frames a second have passed, the sum of the movements of the frames (2×0.05) matches the target speed, `0.1`. Delta Time can be interpreted as the percentage of a second that has passed. To make the Delta Time affect our movement, we should simply multiply our speed by Delta Time every frame because the Delta Time can be different every frame. So let's do that:

1. We access Delta Time using `Time.deltaTime`. We can start affecting the movement by multiplying the Delta Time in every Translate:

```

if (Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.UpArrow))
{ transform.Translate(0, 0, speed * Time.deltaTime); }

if (Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.DownArrow))
{ transform.Translate(0, 0, -speed * Time.deltaTime); }

if (Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.LeftArrow))
{ transform.Translate(-speed * Time.deltaTime, 0, 0); }

if (Input.GetKey(KeyCode.D) || Input.GetKey(KeyCode.RightArrow))
{ transform.Translate(speed * Time.deltaTime, 0, 0); }

```

Figure 6.21: Multiplying speed by Delta Time

2. We can do the same with the rotation speed, by chaining the mouse and speed multiplications:

```

float mouseX = Input.GetAxis("Mouse X");
transform.Rotate(0, mouseX * rotationSpeed * Time.deltaTime, 0);

```

Figure 6.22: Applying Delta Time to the rotation code

If you save and play the game, you will notice that the movement will be slower than before. That's because now `0.1` is the movement per second, meaning `10` centimeters per second, which is pretty slow; try raising those values. In my case, `10` for speed and `180` for rotation speed was enough, but the rotation speed depends on the player's preferred sensibility, which can be configurable, but let's keep that for another time. The Visual Scripting change for rotation will look like this:

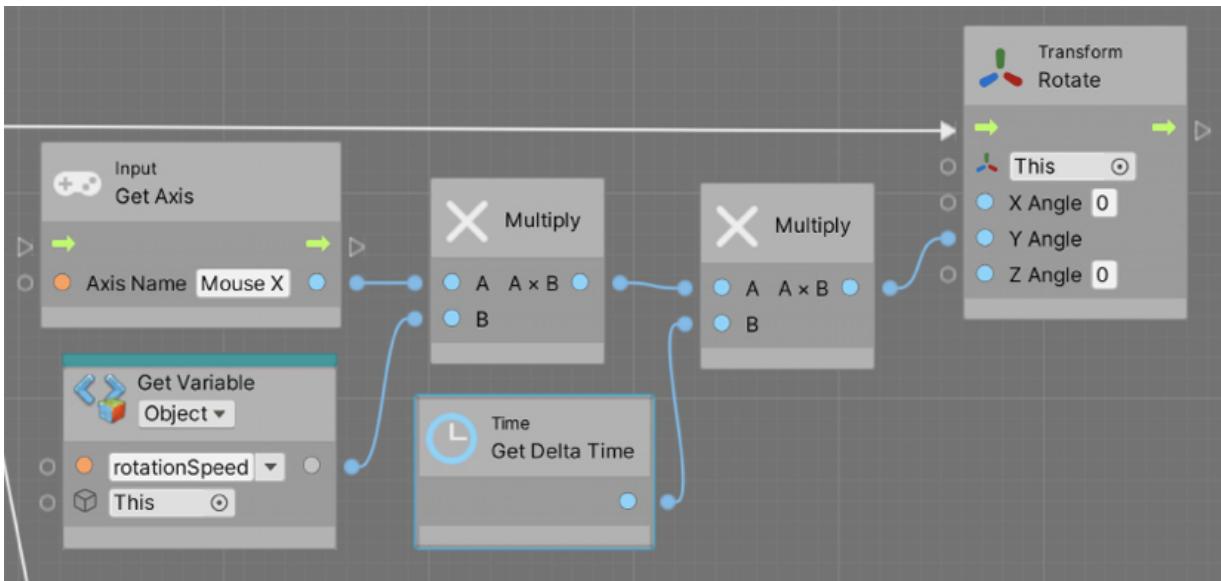


Figure 6.23: Applying Delta Time to Rotate Visual Script

For movement, you can easily extrapolate from this example or remember to check the project on <https://github.com/PacktPublishing/Hands-On-Unity-2023-Game-Development-Fourth-Edition>. We simply chained another **Multiply** node with **Get Delta Time**. We just learned how to mix the Input System of Unity, which tells us about the state of the keyboard, mouse, and other input devices, with the basic `Transform` movement functions. This way, we can start making our game feel more dynamic. Now that we have finished the player's movement, let's discuss how to make the player shoot bullets using `Instantiate` functions.

Implementing spawning

We have created lots of objects in the editor that define our level, but once the game begins, and according to the player's actions, new objects must be created to better fit the scenarios generated by player interaction. Enemies might need to appear after a while, or bullets must be created according to the player's input; even when enemies die, there's a chance of spawning a power-up. This means

that we cannot create all the necessary objects beforehand but should create them dynamically, and that's done through scripting. In this section, we will examine the following spawning concepts:

- Spawning objects
- Timing actions
- Destroying objects

We will start seeing the Unity `Instantiate` function, which allows us to create instances of Prefabs at runtime, such as when pressing a key, or in a time-based fashion, such as making our enemy spawn bullets once every certain amount of time. Also, we will learn how to destroy these objects to prevent our scene from starting to perform badly due to too many objects being processed. Let's start with how to shoot bullets according to the player's input.

Spawning objects

To spawn an object in runtime or **Play** mode, we need a description of the object, which components it has, and its settings and possible sub-objects. You might be thinking about Prefabs here, and you are right; we will use an instruction that will tell Unity to create an instance of a Prefab via scripting. Remember that an instance of a Prefab is an object created based on the Prefab—basically a clone of the original one. We will start with shooting player's bullets, so first let's create the bullet Prefab by following these steps:

1. Create a sphere in **GameObject | 3D Object | Sphere**. You can replace the sphere mesh with another bullet model if you want, but we will keep the sphere in this example for now.
2. Rename the sphere `Bullet`.
3. Create a material by clicking on the `+` button of the **Project** window, choosing the option **Material**, and calling it `Bullet`. Remember to place it inside the `Materials` folder.

4. Check the **Emission** checkbox in the material and set the **Emission Map** and **Base Map** colors to red.:

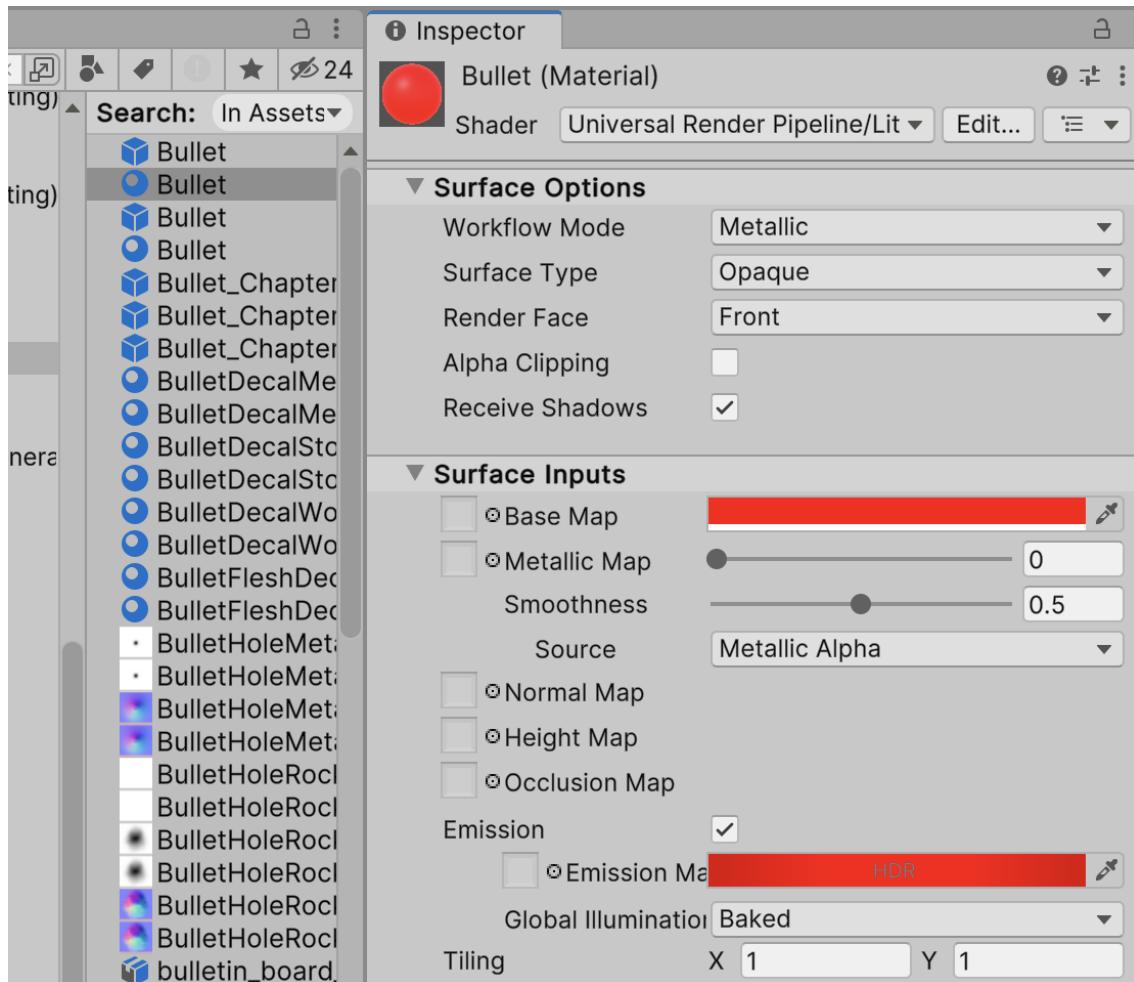


Figure 6.24: Creating a red bullet material with emission color

5. Apply the **Material** to the **Sphere** by dragging the material to it.
6. Set the Scale to a smaller value—`0.3, 0.3, 0.3` worked in my case.
7. Create a script called `ForwardMovement` to make the bullet constantly move forward at a fixed speed. You can create it both with C# and Visual Scripting, but for simplicity, we are only going to use C# in this case.

I suggest you try to solve this by yourself first and look at the screenshot in the next step with the solution later as a little challenge to recap the movement concepts we saw previously. If you don't recall how to create a script, please look at *Chapter 5, Introduction to C# and Visual Scripting*, and check the previous section to see how to move objects.

8. The next screenshot shows you what the script should look like:

```
using UnityEngine;

public class ForwardMovement : MonoBehaviour
{
    public float speed;

    void Update()
    {
        transform.Translate(0, 0, speed * Time.deltaTime);
    }
}
```

Figure 6.25: A simple ForwardMovement script

9. Add the script (if not already there) to the bullet and set the speed to a value you see fit. Usually, bullets are faster than the player but that depends on the game experience you want to get. In my case, 20 worked fine. Test it by placing the bullet near the player and playing the game:

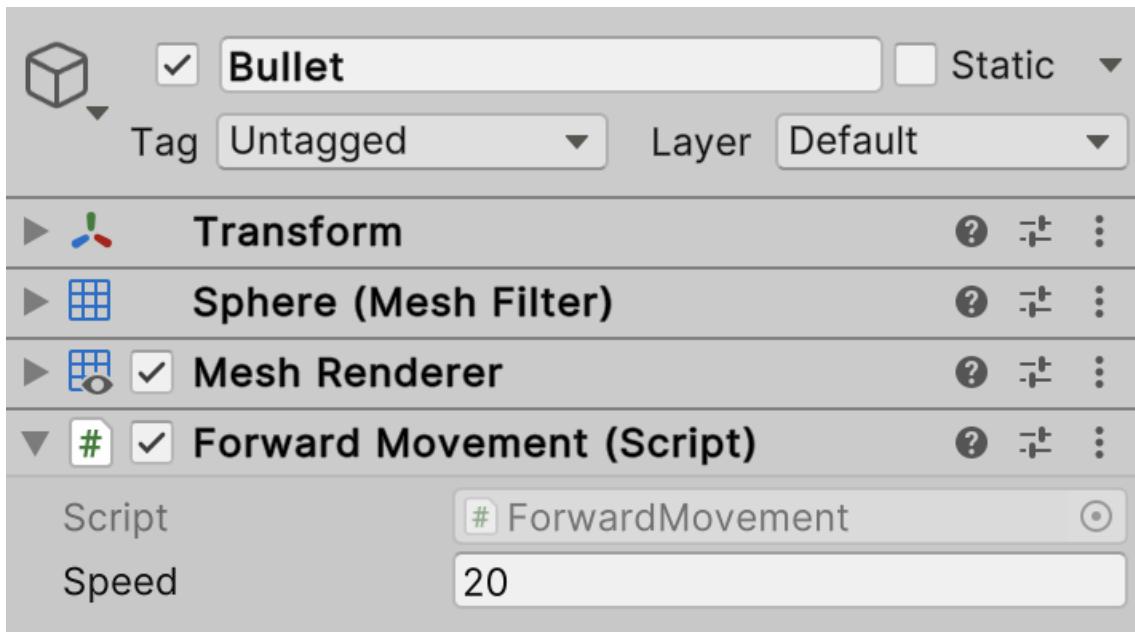


Figure 6.26: A `ForwardMovement` script in the bullet

10. Drag the bullet `GameObject` instance to the `Prefabs` folder to create a **Bullet** Prefab. Remember that the Prefab is an asset that has a description of the created bullet, like a blueprint of how to create a bullet:

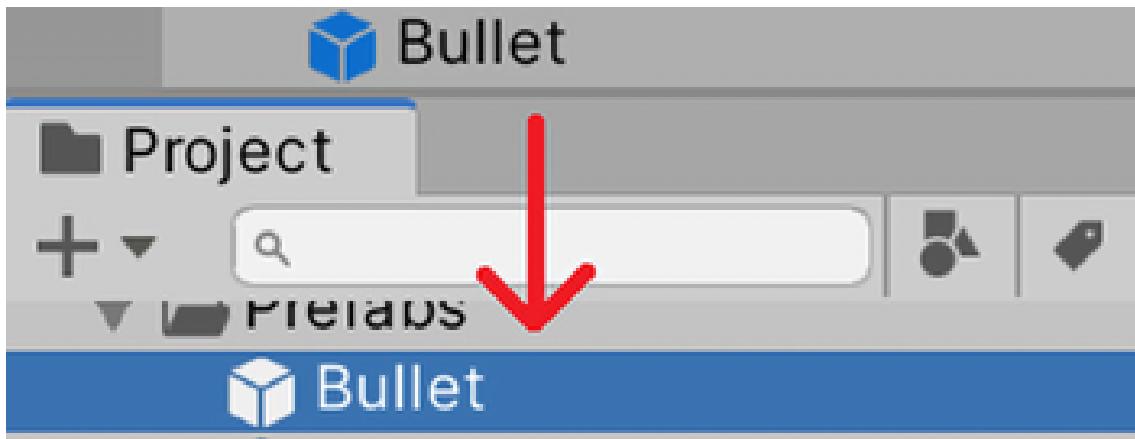


Figure 6.27: Creating a Prefab

11. Remove the original bullet from the scene; we will use the Prefab to create bullets when the player presses a key (if ever).

Now that we have our bullet Prefab, it is time to instantiate it (clone it) when the player presses a key. To do that, follow these steps:

1. Create and add a script to the player's `GameObject` called `PlayerShooting` and open it.
2. We need a way for the script to have access to the Prefab to know which Prefab to use from probably the dozens we will have in our project. All of the data our script needs, which depends on the desired game experience, is in the form of a field, such as the speed field used so far. So in this case, we need a field of the `GameObject` type—a field that can reference or point to a specific Prefab, which can be set using the editor.
3. Adding the field code would look like this:

```
using UnityEngine;

public class PlayerShooting : MonoBehaviour
{
    public GameObject prefab;
}
```

Figure 6.28: The Prefab reference field

As you might have guessed, we can use the `GameObject` type to not only reference Prefabs but also other objects. Imagine an enemy AI needing a reference to the player object to get its position, using `GameObject` to link the two objects. The trick here is considering that Prefabs are just regular GameObjects that live outside the scene; you cannot see them, but they are in memory, ready to be copied or instantiated. You will only see them through copies or instances that are placed in the scene with scripting or via the editor, as we have done so far.

1. In the editor, click on the circle toward the right of the property and select the `Bullet` Prefab. Another option is to just drag the `Bullet` Prefab to the property. This way, we tell our script that the bullet to shoot will be that particular one. Remember to drag the Prefab and not the bullet in the scene (the one in the scene should be deleted by now):

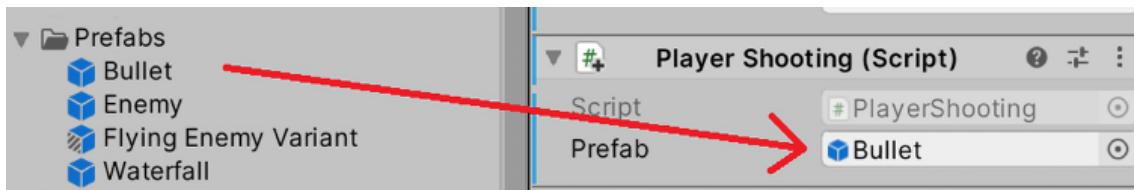


Figure 6.29: Setting the Prefab reference to point the bullet

2. We will shoot the bullet when the player presses the left mouse button, so place the proper `if` statement to handle that in the `Update` event function, like the one shown in the next screenshot:

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Mouse0)) { }
}
```

Figure 6.30: Detecting the pressure of the left mouse button

3. You will notice that this time, we used `GetKeyDown` instead of `GetKey`, the former being a way to detect the exact frame the pressure of the key started; this `if` statement will execute its code only in that frame, and until the key is released and re-pressed, it won't enter again. This is one way to prevent bullets from spawning in every frame, but just for fun, you can try using `GetKey` instead to check how it would behave. Also, `KeyCode.Mouse0` is the mouse button number that belongs to

left-click, `KeyCode.Mouse1` is the right-click, and `KeyCode.Mouse2` is the middle click.

4. Use the `Instantiate` function to clone the Prefab, passing the reference to it as the first parameter. This will create a clone of the previously mentioned Prefab that will be placed in the scene:

```
if (Input.GetKeyDown(KeyCode.Mouse0))  
{  
    Instantiate(prefab);  
}
```

Figure 6.31: Instantiating the Prefab

If you save the script and play the game, you will notice that when you press the mouse, a bullet will be spawning, but probably not in the place you are expecting. If you don't see it, try to check the Hierarchy for new objects; it will be there. The problem here is that we didn't specify the desired spawn position, and we have two ways of setting that, which we will see in the next steps:

1. The first way is to use the `transform.position` and `transform.rotation` inherited fields from `MonoBehaviour`, which will tell us our current position and rotation. We can pass them as the second and third parameters of the `Instantiate` function, which will understand that this is the place we want our bullet to appear. Remember that it is important to set the rotation to make the bullet face the same direction as the player, so it will move that way:

```
Instantiate(prefab, transform.position, transform.rotation);
```

Figure 6.32: Instantiating the Prefab in our position and rotation

2. The second way is by using the previous version of `Instantiate`, but saving the reference returned by the function, which will be pointing to the clone of the Prefab. This allows us to change whatever we want from it. In this case, we will need the following three lines; the first will instantiate and capture the clone reference, the second will set the position, and the third the rotation. We will also use the `transform.position` field of the clone, but this time to change its value by using the `=` (assignment) operator:

```
GameObject clone = Instantiate(prefab);
clone.transform.position = transform.position;
clone.transform.rotation = transform.rotation;
```

Figure 6.33: The longer version of instantiating a Prefab in a specific position

Remember that you can check the project's GitHub repository linked in the *Preface* to see the full script finished. Now you can save the file with one of the versions and try to shoot. If you try the script so far, you should see the bullet spawn in the player's position, but in our case, it will probably be the floor. The problem here is that the player's character pivot is there, and usually, every humanoid character has the pivot in their feet. We have several ways to fix that. The most flexible one is to create a **Shoot Point**, an empty GameObject child of the player, placed in the position we want the bullet to spawn. We can use the position of that object instead of the player's position by following these steps:

1. Create an empty `GameObject` in **GameObject | Create Empty**. Rename it to `ShootPoint`.

2. Make it a child of the player's GameObject and place it where you want the bullet to appear, probably a little higher and further forward:

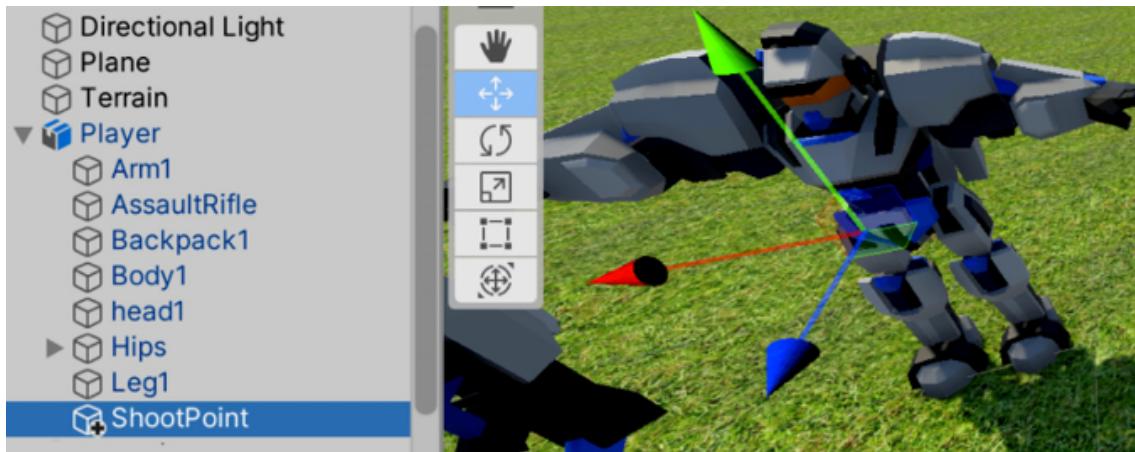


Figure 6.34: An empty ShootPoint object placed inside the character

3. As usual, to access the data of another object, we need a reference to it, such as the Prefab reference, but this time it needs to point to our shoot point. We can create another `GameObject` type field, but this time drag `ShootPoint` instead of the Prefab. The script and the object set would look as follows:

```
public GameObject prefab;  
public GameObject shootPoint;
```

Figure 6.35: The Prefab and ShootPoint fields and how they are set in the editor

4. We can access the position of the `ShootPoint` by using the `transform.position` field of it again, as shown in the following screenshot:

```
clone.transform.position = shootPoint.transform.position;  
clone.transform.rotation = shootPoint.transform.rotation;
```

Figure 6.36: The Prefab and ShootPoint fields and how they are set in the editor

The Visual Scripting version of **ForwardMovement** will look like this:

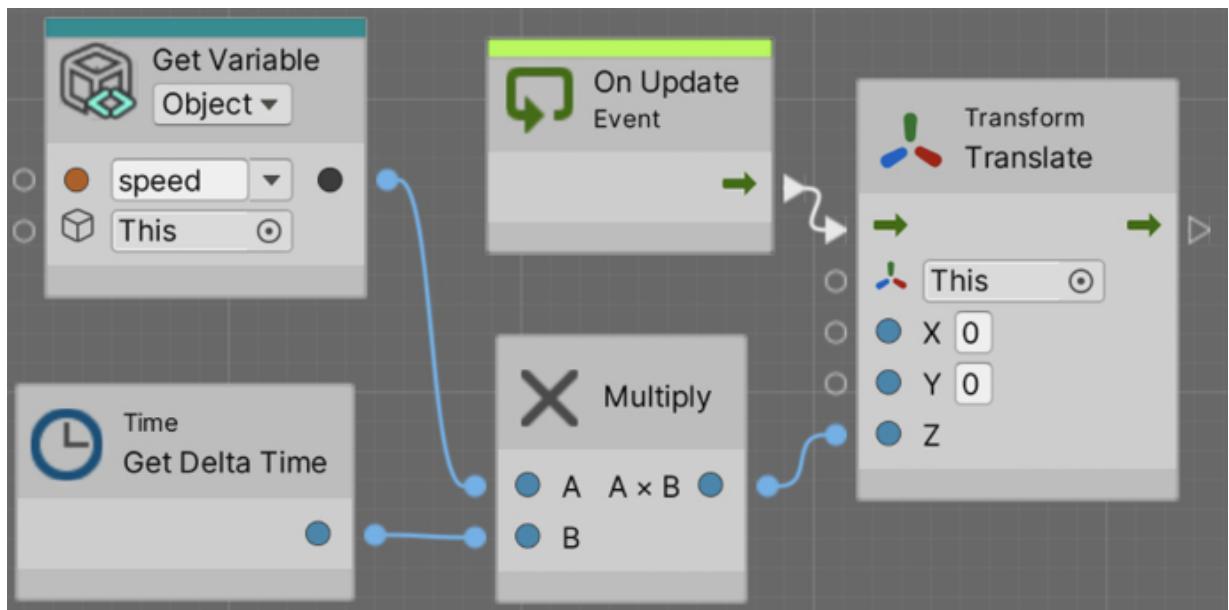


Figure 6.37: ForwardMovement with Visual Scripting

And `PlayerShooting` will look like this:

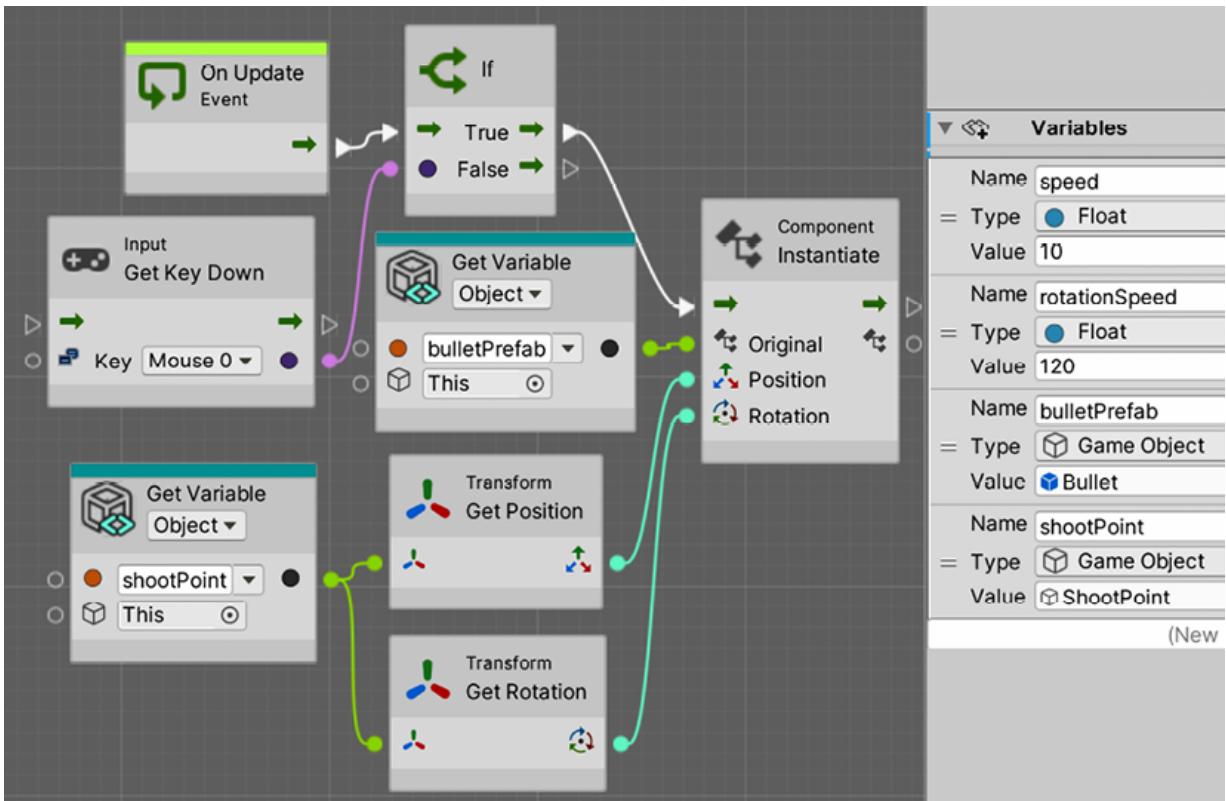


Figure 6.38: Instantiating in the PlayerShooting Visual Script

As you can see, we added a second **Script Machine** component with a new graph called **Player Shooting**. We also added a new variable, `bulletPrefab`, of type `GameObject` and dragged the **Bullet** Prefab to it, and a second `GameObject` typed variable called `shootPoint`, to have the reference to the bullet's spawn position. The rest of the script is essentially the counterpart of the C# version without any major differences. Something to highlight here is how we connected the `Transform GetPosition` and `Transform GetRotation` nodes to the `GetVariable` node belonging to the `shootPoint`; this way, we are accessing the position and rotation of the shooting point. If you don't specify that, it will use the player's position and rotation, which in the case of our model is in the player's character's feet. You will notice that now shooting and rotating with the mouse has a problem; when moving the mouse to rotate, the pointer will fall outside the **Game View**, and when clicking, you will accidentally click the editor, losing the focus on

the **Game** View, so you will need to click the **Game** View again to regain focus and use input again. A way to prevent this is to disable the cursor while playing. To do this, follow these steps:

1. Add a `Start` event function to our Player Movement Script.
2. Add the two lines you can see in the following screenshot to your script. The first one will make the cursor visible, and the second one will lock it in the middle of the screen, so it will never abandon the **Game** View. Consider the latter; you will need to reenable the cursor when you switch back to the main menu or the pause menu, to allow the mouse to click the UI buttons:

```
void Start()
{
    Cursor.visible = false;
    Cursor.lockState = CursorLockMode.Locked;
}
```

Figure 6.39: Disabling the mouse cursor

3. Save and test this. If you want to stop the game, you could either press Ctrl + Shift + P (Command + Shift + P on Mac) or press the Esc key to reenable the mouse. Both options only work in the editor; in the real game, you will need to reset `Cursor.visible` to `true` and `Cursor.lockState` to `CursorLockMode.None`.
4. The Visual Scripting equivalent will look like this:

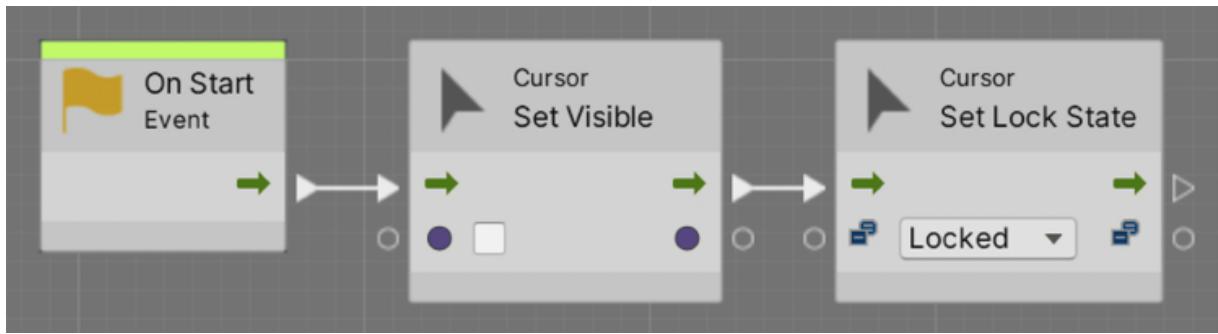


Figure 6.40: Disabling the mouse cursor in Visual Scripting

Now that we have covered the basics of object spawning, let's see an advanced example by combining it with timers.

Timing actions

Not entirely related to spawning, but usually used together, timing actions is a common task in video games. The idea is to schedule something to happen later; maybe we want the bullet to be destroyed after a while to prevent memory overflow, or we want to control the spawn rate of enemies or when they should spawn. That's exactly what we are going to do in this section, starting with implementing the **enemy waves**. The idea is that we want to spawn enemies at a certain rate in different moments of the game; maybe we want to spawn enemies from second 1 to 5 at a rate of 2 per second, getting 10 enemies, and giving the player up to 20 seconds to finish them, programming another wave starting at 25 seconds. Of course, this depends a lot on the exact game you want, and you can start with an idea like this one and modify it after some testing to find the exact way you want the wave system to work. In our case, we will apply timing by implementing a simple wave system. First of all, we need an enemy, and for now, we will simply use the same 3D model we used for the player, but add a Forward Movement script to simply make it move forward; later in this book, we will add AI behavior to our enemies. I suggest you try to create this Prefab by yourself and look at the following steps once you have tried it, to check the correct answer: Drag the downloaded Character FBX

model to the scene to create another instance of it, but rename it to `Enemy` this time:

1. Add the `ForwardMovement` script created for the bullets but this time to `Enemy`, and set it at a speed of `10` for now.
2. Drag the `Enemy` GameObject to the Project to create a Prefab based on that one; we will need to spawn it later. Remember to choose **Prefab Variant**, which will keep the Prefab linked with the original model to make the changes applied to the model automatically apply to the Prefab.
3. Remember also to destroy the original `Enemy` from the scene.

Now, to schedule actions, we will use the `Invoke` functions to create timers. They are basic but enough for our requirements. Let's use them by following these steps:

1. Create an empty GameObject at one end of the base and call it `Wave1a`.
2. Create and add a script called `WaveSpawner` to it.
3. Our spawner will need four fields: the `Enemy` Prefab to spawn, the `startTime` of the wave, the `endTime`, and the spawn rate of the enemies (how much time should be between each spawn). The script and the settings will look like the following screenshot:

```
public GameObject prefab;  
public float startTime;  
public float endTime;  
public float spawnRate;
```

Figure 6.41: The fields of the wave spawner script

We will use the `InvokeRepeating` function to schedule a custom function to repeat periodically. You will need to schedule the repetition just once; Unity will remember that, so don't do it for every frame. This is a good reason to use the `Start` event function instead. The first argument of the function is a string (text between the quotation marks) with the name of the other function to execute periodically, and unlike `Start` or `Update`, you can name the function whatever you want. The second argument is the time to start repeating, our `startTime` field, in this case. Finally, the third argument is the repetition rate of the function—how much time needs to pass between each repetition—this being the `spawnRate` field. You can find how to call that function in the next screenshot, along with the custom `Spawn` function:

```

void Start()
{
    InvokeRepeating("Spawn", startTime, spawnRate);
}

void Spawn() { }

```

Figure 6.42: Scheduling a Spawn function to repeat

1. Inside the `Spawn` function, we can put the spawning code as we know, using the `Instantiate` function. The idea is to call this function at a certain rate to spawn one enemy per call. This time, the spawn position will be in the same position as the spawner, so place it carefully:

```

void Spawn()
{
    Instantiate(prefab, transform.position, transform.rotation);
}

```

Figure 6.43: Instantiating in the Spawn function

If you test this script by setting the Prefab `startTime` and `spawnRate` fields to some values greater than 0, you will notice that the enemies will start spawning but never stop, and you can see that we didn't use the `endTime` field so far. The idea is to call the `CancelInvoke` function, the one function that will cancel all the `InvokeRepeating` calls we made, but after a while. We will delay the execution of `CancelInvoke` using the `Invoke` function, which works similarly to `InvokeRepeating`, but this one executes just once. In the next screenshot, you can see how we added an `Invoke` call to the `CancelInvoke` function in `Start`, using the `endTime` field as the

time to execute `CancelInvoke`. This will execute `CancelInvoke` after a while, canceling the first `InvokeRepeating` call that spawns the Prefab:

```
InvokeRepeating("Spawn", startTime, spawnRate);
Invoke("CancelInvoke", endTime);
```

Figure 6.44: Scheduling a Spawn repetition but canceling after a while with `CancelInvoke`

This time, we used `Invoke` to delay the call to `CancelInvoke`. We didn't create a custom function because `CancelInvoke` doesn't receive arguments. If you need to schedule a function with arguments, you will need to create a wrapper function without parameters that calls the desired one and schedules it, as we did with `Spawn`, where the only intention is to call `Instantiate` with specific arguments.

1. Now you can save and set some real values to our spawner. In my case, I used the ones shown in the following screenshot:

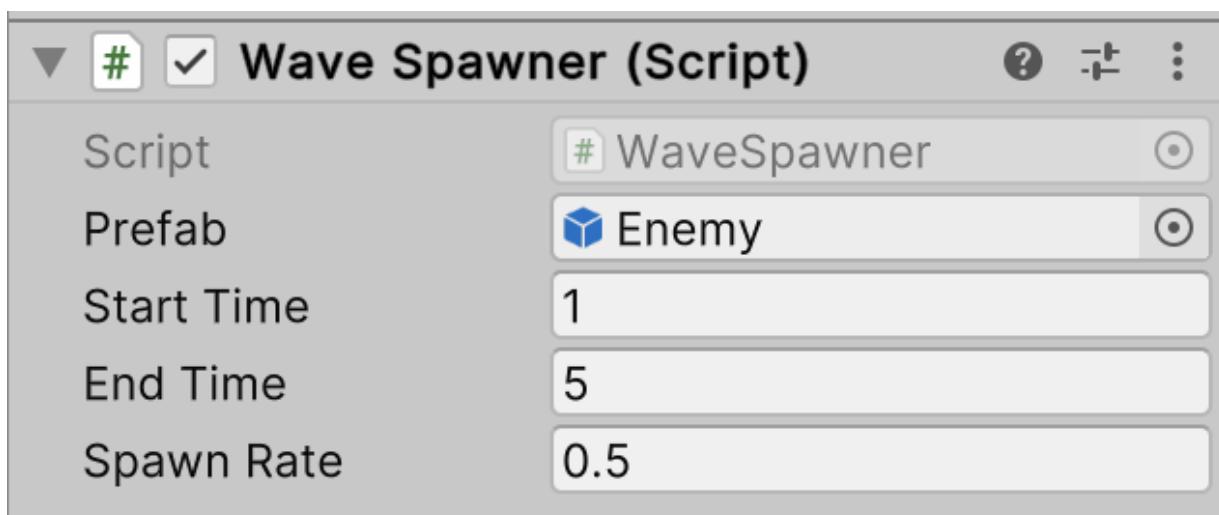


Figure 6.45: Spawning enemies from second 1 to 5 of the gameplay every 0.5 seconds, 2 per second

You should see the enemies being spawned one next to the other, and because they move forward, they will form a row of enemies. This behavior will change later when we make use of the AI features of Unity in a future chapter. Now, the Visual Scripting version will look like this:

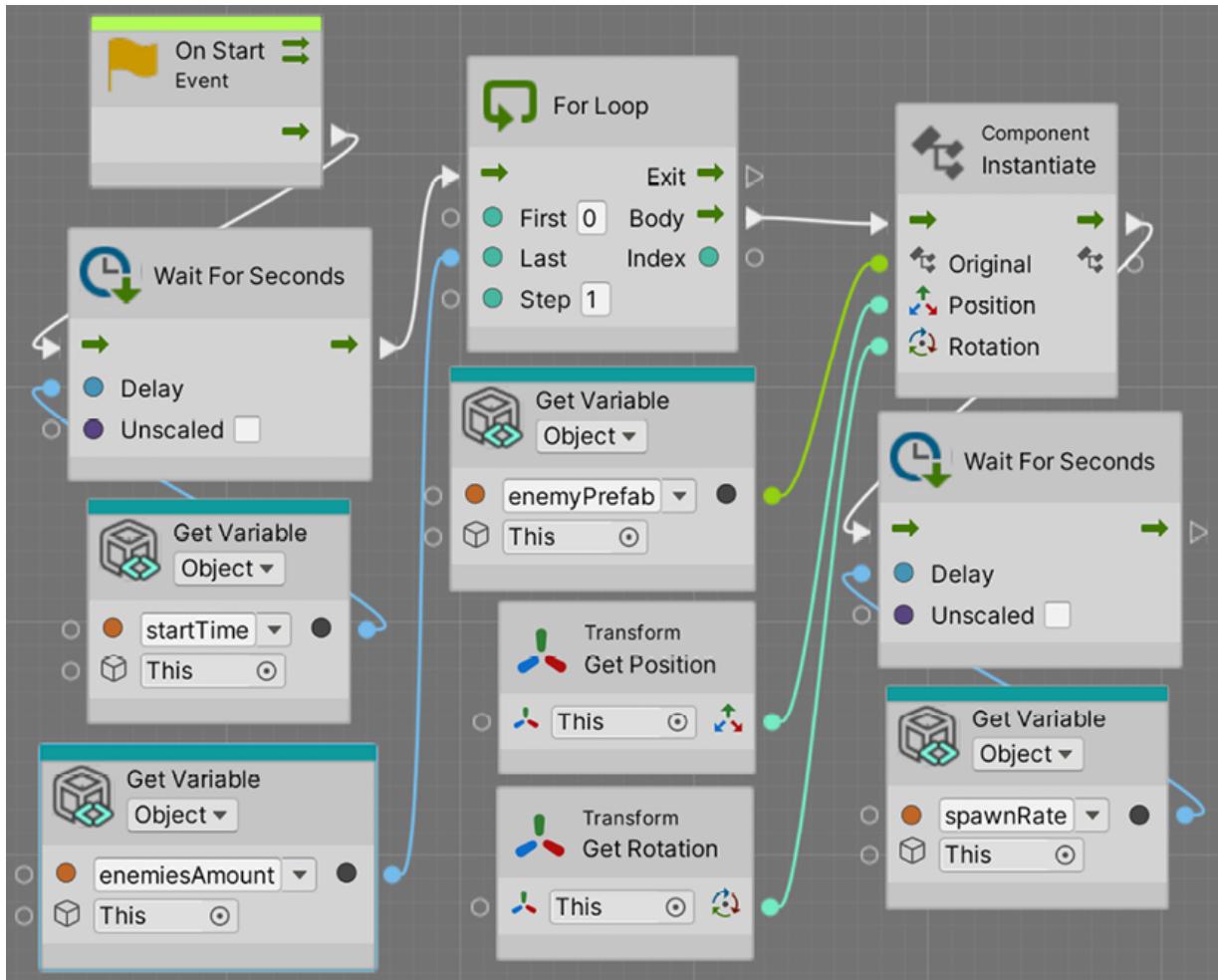


Figure 6.46: Spawning enemies in Visual Scripting

While we could use the `InvokeRepeating` approach in Visual Scripting, here we can see some benefits of the Visual approach, given it sometimes has more flexibility than coding. In this case, we used the `Wait For Seconds` node at the beginning of the `Start`, a

node that basically will hold the execution of the flow for a couple of seconds. This will create the initial delay we had in the original script; that's why we used the `startTime` as the amount of `Delay`. Now, after the wait, we used a `For` loop; for this example, we changed the concept of the script, as we want to spawn a specific number of enemies instead of spawning during a time. The `For` loop is essentially a classic `For` that will repeat whatever is connected to the `Body` output pin the number of times specified by the `Last` input pin. We connected that pin to a variable to control the number of enemies we want to spawn. Then, we connected an `Instantiate` to the `Body` output pin of the `For` loop to instantiate our enemies, and then a `Wait For Seconds` node, to stop the flow for a time before the loop can continue spawning enemies. Something interesting is that if you play the game now, you will receive an error in the console that will look like this:

! [20:23:45] InvalidOperationException: Port 'enter' on 'WaitForSecondsUnit#ed5f4...' can only be triggered in a coroutine.
Unity.VisualScripting.Flow.InvokeDelegate (Unity.VisualScripting.ControlInput input) (at Library/PackageCache/com.unity

Figure 6.47: Error when using Wait nodes

You can even go back to the graph editor and see that the conflicting node will be highlighted in red:

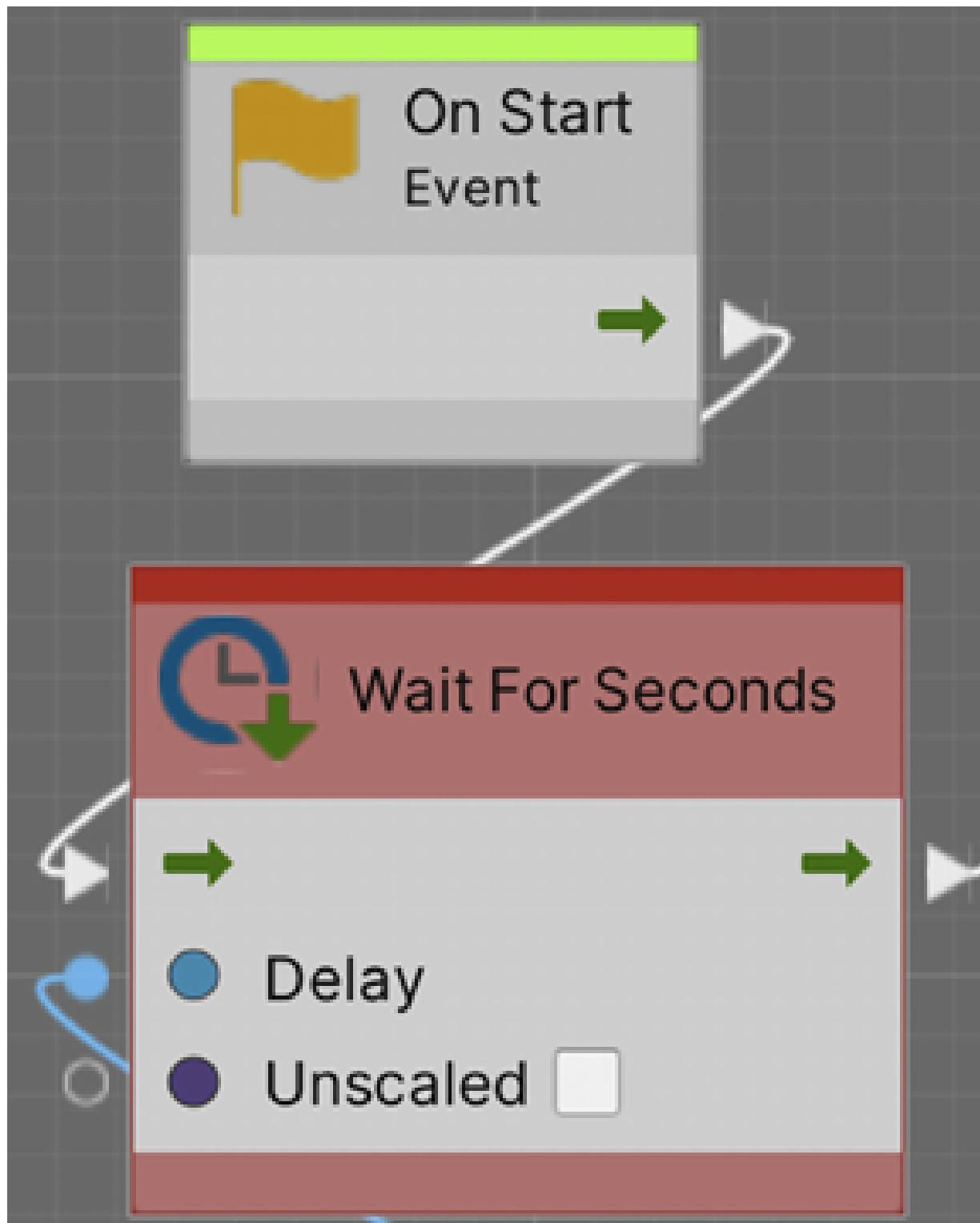


Figure 6.48: Node causing the error

The issue here is that in order for the `Wait For Seconds` nodes to work, you need to mark the `Start` event as a **Coroutine**. This will basically allow the event to be paused for an amount of time and be resumed later.

The concept of coroutines exists in C#, but as it is simpler to implement here in Visual Scripting than in C#, we decided to go with this approach here. If you want more info about them, please check this documentation:

<https://docs.unity3d.com/Manual/Coroutines.html>

To solve this error, just select the `On Start` event node and check the **Coroutine** checkbox in the **Graph Inspector** pane on the left of the **Script Graph** editor. If you don't see it, consider clicking the **Info** button (circle with *i*) in the top-left part of the editor. A coroutine is a function that can be paused and resumed later, and that's exactly what the `Wait` node does. Coroutines also exist in `MonoBehaviours`, but let's keep things simple for now.

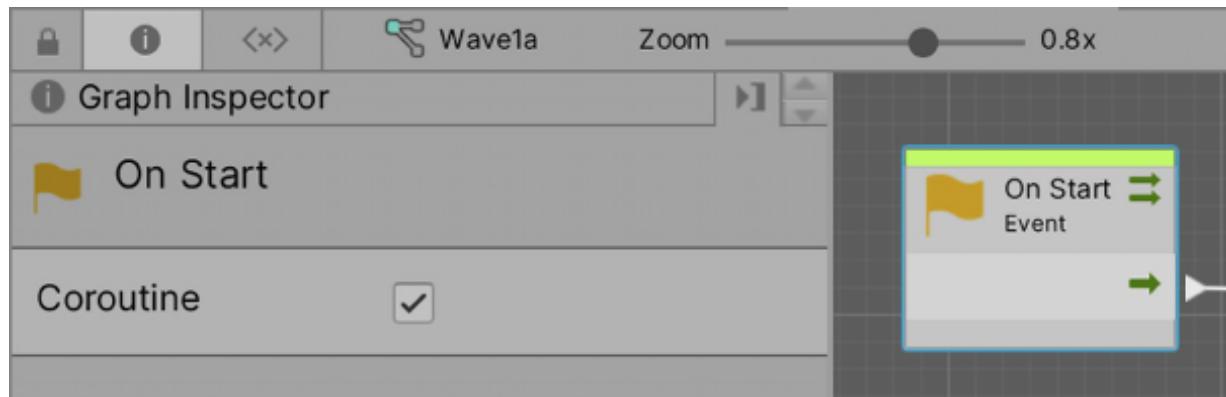


Figure 6.49: Marking Start as a coroutine

Now that we have discussed timing and spawn, let's discuss timing and `Destroy` to prevent our bullets from living forever in the memory.

Destroying objects

We can use the `Destroy` function to destroy object instances. The idea is to make the bullets have a script that schedules their own auto-destruction after a while to prevent them from living forever. We will create the script by following these steps:

1. Select the Prefab of `Bullet` and add a script called `Autodestroy` to it, as you did with other objects using the **Add Component | New Script** option. This time, the script will be added to the Prefab, and each instance of the Prefab you spawn will have it.
2. You can use the `Destroy` function, as shown in the next screenshot, to destroy the object just once in `Start`:

```
void Start()
{
    Destroy(gameObject);
}
```

Figure 6.50: Destroying an object when it starts

The `Destroy` function expects the object to destroy as the first argument, and here, we are using the `gameObject` reference; a way to point to the `GameObject` our script is placed into to destroy it. If you use the `this` pointer instead of `GameObject`, we will be destroying only the `Autodestroy` component we are creating. Of course, we don't want the bullet to be destroyed as soon as it is spawned, so we need to delay the destruction. You may be thinking about using `Invoke`, but unlike most functions in Unity, `Destroy`

can receive a second argument, which is the time to wait until destruction.

1. Create a delay field to use as the second argument of `Destroy`, as shown in the next screenshot:

```
public float delay;  
  
void Start()  
{  
    Destroy(gameObject, delay);  
}
```

Figure 6.51: Using a field to configure the delay to destroy the object

2. Set the `delay` field to a proper value; in my case, 5 was enough. Now check how the bullets despawn (removed) after a while by looking at them being removed from the Hierarchy.
3. The Visual Scripting equivalent will look like this:

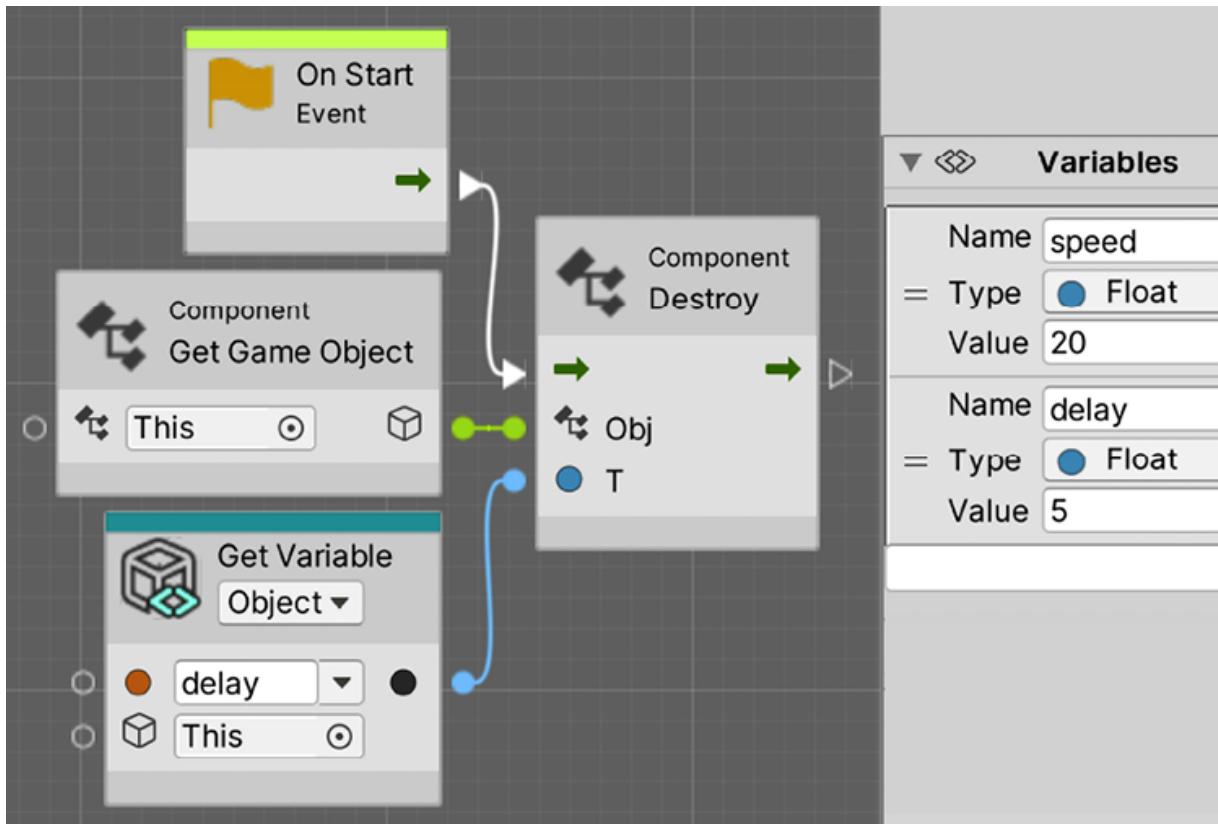


Figure 6.52: Destroying in Visual Scripting

Regarding this version, notice how we use the **Component Destroy (Obj, T)** version of the `Destroy` node, which includes the delay time. Additionally, look for the `Object Pool` concept, which is a way to recycle objects instead of creating them constantly; you will learn that sometimes creating and destroying objects is not that performant. Now, we can create and destroy objects at will, which is something very common in Unity scripting. In the next section, we will discuss how to modify the scripts we have created so far to support the new Unity Input System.

Using the new Input System

We have been using the **Input** class to detect the buttons and axes being pressed, and for our simple usage that is more than enough. But the default Unity input system has its limitations regarding

extensibility to support new input hardware and mappings. In this section, we will explore the following concepts:

- Installing the new Input System
- Creating Input Mappings
- Using Mappings in scripts

Let's start exploring how to install the new Input System.

Installing the new Input System

To start using the new Input System, it needs to be installed like any other package we have installed so far, using the **Package Manager**. The package is just called **Input System**, so go ahead and install it as usual. In this case, we are using version 1.6.1, but a newer one may be available when you read this chapter.

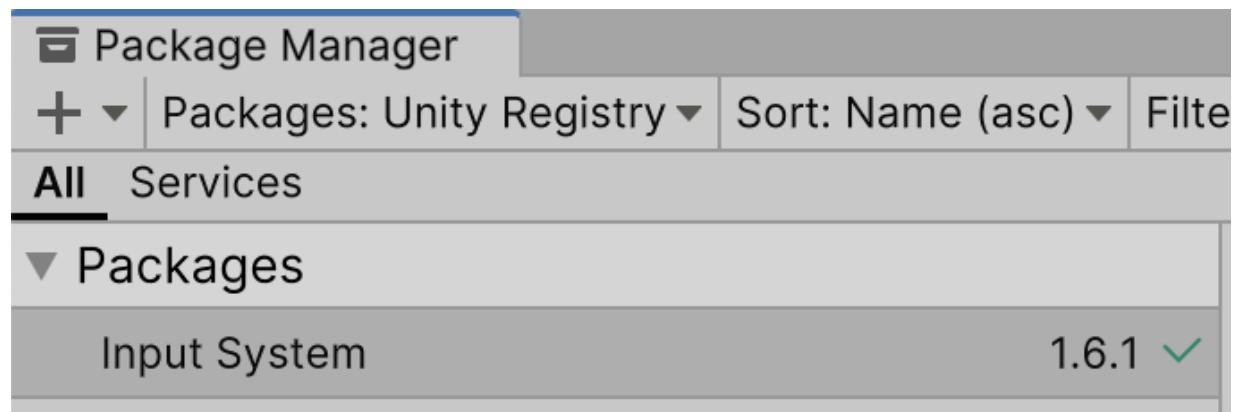


Figure 6.53: Installing the new Input System package

By default, when you install the Input System, it will prompt you to enable the new Input System with a window like the one in the following image. If that appears, just click **Yes** and wait for Unity to restart:



Warning

This project is using the new input system package but the native platform backends for the new input system are not enabled in the player settings. This means that no input from native devices will come through.

Do you want to enable the backends?
Doing so will ***RESTART*** the editor and
will ***DISABLE*** the old UnityEngine.Input
APIs.

No

Yes

Figure 6.54: Switching the active Input System

If for some reason that didn't appear, the other alternative is going to **Edit | Project Settings** and then going to **Player | Other Settings | Configuration** to set the **Active Input Handling** property to **Input System Package (New)**. There's an option called **Both** to keep both enabled, but let's stick with just one.

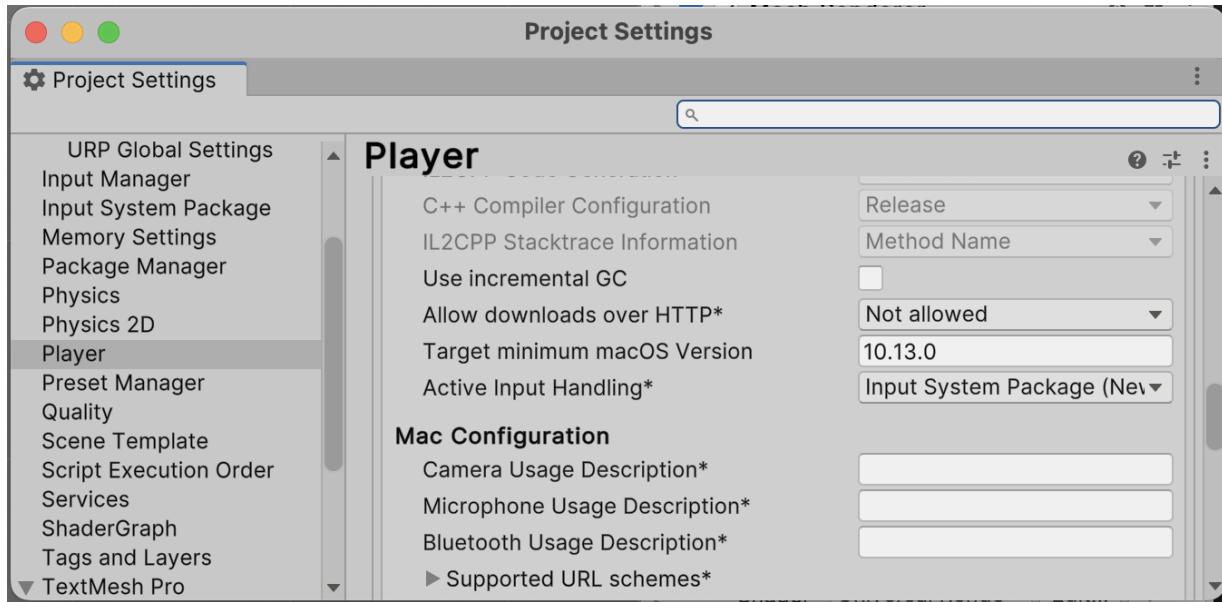


Figure 6.55: Switching the active Input System

Now that we have the system installed and set up, let's explore how to create the Input Mappings needed.

Creating Input Mappings

The new system has a way to directly request the current state of a button or thumbstick to the gamepad, mouse, keyboard, or whatever other device we have, like what we did so far with the previous Input System. But using this method would prevent us from using one of the best features of the system, Input Mappings. The idea of an Input Mapping is to abstract the Input Actions from the Physical Input. Instead of thinking about the space bar, the left thumbstick of a gamepad, or the right click of a mouse, you think in terms of actions, like move, shoot, or jump. In code,

you will ask if the `shoot` button has been pressed, or the current value of the `move` axes, like we did with the mouse axes rotation. While the previous system supported a certain degree of Input Mapping, the one in the new Input System is way more powerful and easier to configure.

Action	Mappings
Shoot	Left Mouse Button, Left Control, X button of the gamepad
Jump	Space, Y button of gamepad
Horizontal Movement	A and D keys, Left and Right arrows, gamepad Left Stick

Table 6.01: Example of the Input Mapping tableThe power of this idea is that the actual keys or buttons that will trigger these actions are configurable in the Unity editor, allowing any game designed to alter the exact keys to control the entire game without changing the code. We can even map more than one button to the same action, even from different devices, so we can make the mouse, keyboard, and gamepad trigger the same action, greatly simplifying our code. Another benefit is that the user can also rebind the keys with some custom UI we can add to our game, which is very common in PC games. The easiest way to start creating an Input Mapping is through the **Player Input** component. This component, as the name suggests, represents the input of a particular player, allowing us to have one of those on each player in our game to support split-screen multiplayer, but let's focus on single-player. Adding this script to our player will allow us to use the **Create Actions...** button to create a default Input Mapping asset. This asset, as a material, can be used by several players, so we modify it and it will affect all of them (for example, adding the `Jump` Input Mapping):

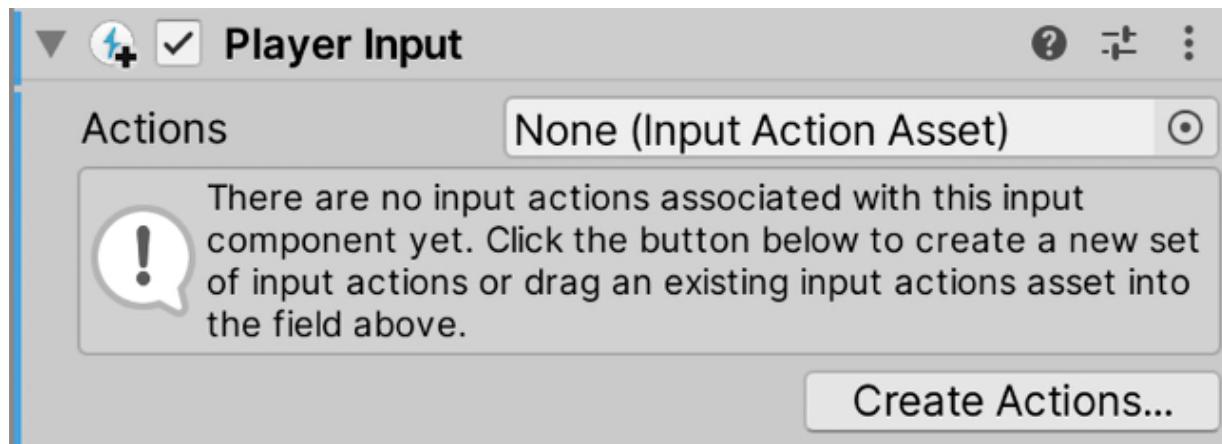


Figure 6.56: Creating Input Action assets using the Player Input component

After clicking that button and saving the asset location in the save prompt, you will see the following screen:

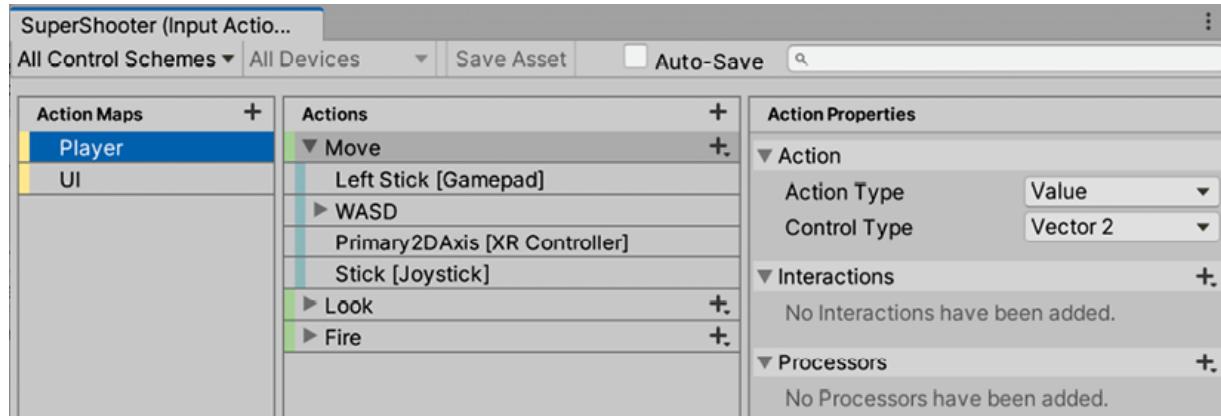


Figure 6.57: The default Input Mapping file

The first part to understand from this asset is the **Action Maps** section (left panel). This allows us to create separate Action Maps for different situations, for example, for driving and on-foot controls in games like GTA. By default, **Player** and **UI** mappings are created, to separate the mappings for the player controlling and navigating through the UI. If you check the **Player Input** component again, you will see that the **Default Map** property is set to **Player**, which means that we will only care for the player controlling the Input

Mappings in this GameObject; any UI action pressed won't be considered. We can switch the active map in runtime at will, for example, to disable the character controller input when we are in the pause menu, or switch to the driving mappings while in a car, using the same buttons but for other purposes. If you select an Action Map in the left panel, you will see all the actions it contains in the **Actions** list in the middle panel. In the case of the **Player**, we have the **Move**, **Look**, and **Fire** mappings, which are exactly the inputs we will use in our game. Bear in mind you can add more if you need to use the + button, but for now, let's stick with the default ones. When you select any action from the list, you will see their configurations in the **Action Properties** panel, the one on the right:

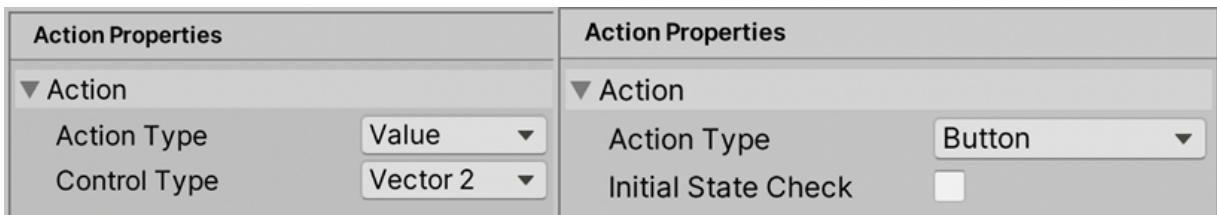


Figure 6.58: The Move (left) and Fire (right) action configurations

As you can see, there's a property called **Action Type** that will dictate which kind of input we are talking about. If you select **Move** in the middle panel, you can see it's a **Value** action type with **Control Type** being `Vector2`, meaning it will return the x and y axis values, the horizontal and vertical values—the kind we expect from any thumbstick in a gamepad. In the previous system, we got those values from separated 1D axes, like the **Mouse X** and **Mouse Y** axes, but here they are combined into a single variable for convenience. On the other hand, the **Fire** action is of type **Button**, which has the capacity not only to check its current state (pressed or released) but also do checks like if it has just been pressed or just released, the equivalents to `GetKey`, `GetKeyDown`, and `GetKeyUp` from the previous system. Now that we understand which actions we have and of which type each one is, let's discuss how the Physical

Input will trigger them. You can click the arrow on the left of each action in the middle panel to see its physical mappings. Let's start exploring the **Move** Action Mappings. In this case, we have 4 mappings:

- **Left Stick [Gamepad]**: The left stick of the gamepad
- **Primary 2D Axis [XR Controller]**: The main stick of the VR controllers
- **Stick [Joystick]**: Main stick for arcade-like joysticks or even flight sticks
- **WASD**: A composite input simulating a stick through the W, A, S, and D keys

If you select any of them, you can check their configurations; let's compare the left stick and WASD as an example:

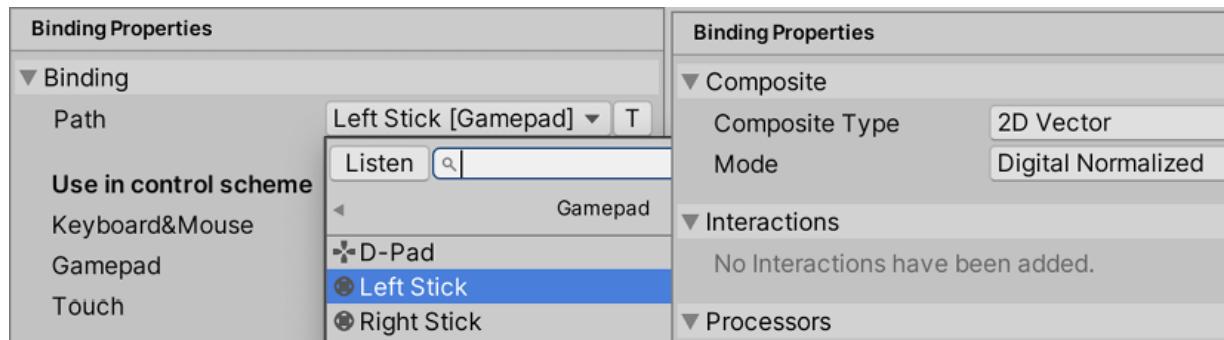


Figure 6.59: The left stick mapping (left) and the WASD key mapping (right)

In the case of the **Left Stick**, you can see the **Path** property that allows you to pick all the possible hardware physical controls that provide `Vector2` values (the x and y axes). In the case of the **WASD** key mapping, you can see it is a composite binding of type **2D Vector**, which, as stated previously, allows us to simulate a 2D Axis with other inputs—keys in this case. If you expand the **WASD** Input Mappings in the middle panel, you can see all inputs that are being composited for this 2D axis, and see their configurations by selecting them:

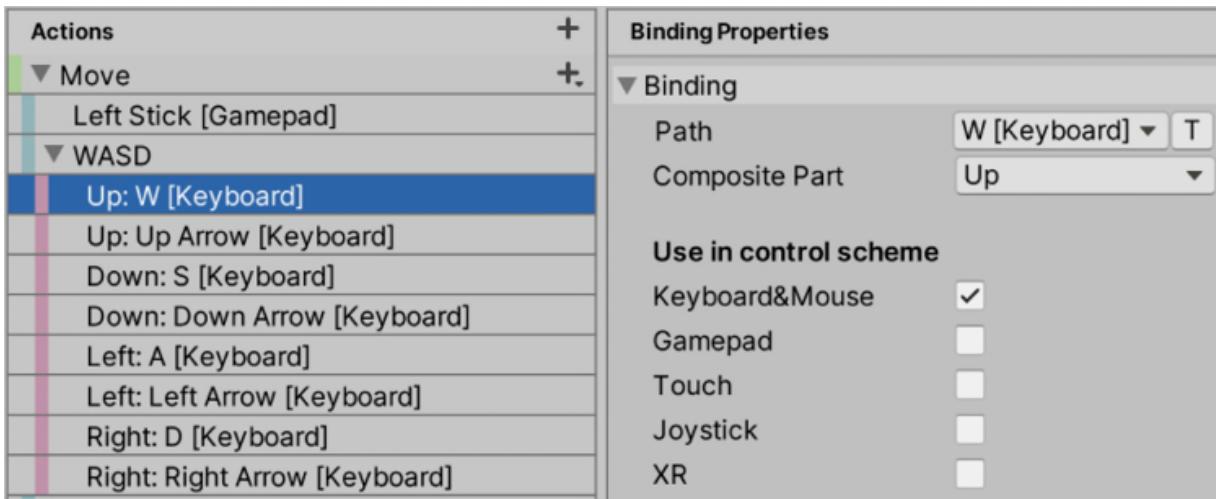


Figure 6.60: The inputs considered for the WASD composite 2D axis

In the preceding case, it maps not only the W, A, S, and D buttons but also the 4 keyboard arrows. Each one of those mappings has a path to select the physical button, but also the **Composite Part** setting, allowing us to specify which direction this input will pull the simulated stick. And with this, we have just scratched the surface of what this system is capable of, but for now, let's keep things simple and use these settings as they are. Remember a new asset was created with the same name as our game (*SuperShooter*, in our case) in the root of the project. You can reopen this Action Mapping window by double-clicking it whenever you want. Now let's see how we can use these inputs in our code.

There's much more this system can do for us. One example is Interactions, which allow us to make things like making the input trigger the action if it's pressed for X amount of time, or also Composite, which triggers the action if a combination of keys is pressed. Check the Input System package documentation here for more information:

<https://docs.unity3d.com/Packages/com.unity.inputsystem@1.6>

Using Mappings in our scripts

This Input System provides several ways to detect the input state. The **Player Input** component has a **Behavior** property to switch between some of the available modes. The simplest one is the one called **Send Messages**, the one that we will use, which will execute methods in our code when the keys are pressed. In this mode, each action in the mappings will have its own event, and you can see all of them in the tooltip at the bottom of the component. As you add mappings, more will appear.



Figure 6.61: All the input events for the default mapping Info

For more information about the other *PlayerInput* behaviour modes, check its documentation here:

<https://docs.unity3d.com/Packages/com.unity.inputsystem@1.8/manual/PlayerInput.html#notification-behaviors> From the list, we will need three, `OnMove`, `OnLook`, and `OnFire`. We can modify our `PlayerMovement` script like in the following screenshot to use them:

```
using UnityEngine;
using UnityEngine.InputSystem;

public class PlayerMovement : MonoBehaviour
{
```

```

public float speed;
public float rotationSpeed;
private Vector2 movementValue;
private float lookValue;

private void Awake()
{
    Cursor.visible = false;
    Cursor.lockState = CursorLockMode.Locked;
}

public void OnMove(InputValue value)
{
    movementValue = value.Get<Vector2>() * speed;
}

public void OnLook(InputValue value)
{
    lookValue = value.Get<Vector2>().x * rotationSpeed;
}

void Update()
{
    transform.Translate(
        movementValue.x * Time.deltaTime,
        0,
        movementValue.y * Time.deltaTime);

    transform.Rotate(0, lookValue * Time.deltaTime, 0);
}

```

Figure 6.62: Player movement with the new Input System

The first difference you will notice is that we don't request the status of the input in the `Update` method like we did before.

Instead, we listen to the `OnMove` and `OnLook` events, which provide us with an `InputValue` parameter containing the current state of those axes. The idea is that every time these axes change value, these events will execute, and if the values didn't change, like when the player keeps pushing the stick all the way to the right, they won't be executed. That's why we need to store the current value in the `movementValue` and `lookValue` variables, to use the latest value of the axis later in the `Update` and apply the movement in every frame. Consider those are private, meaning they won't appear in the editor, but that's fine for our purposes. Also, observe that we added the `using UnityEngine.InputSystem` line at the top of the file to enable the usage of the new Input System in our script. In this version of the `PlayerMovement` script, we used the axis input type like we did with the mouse before but also for movement, unlike the previous version that used buttons. This is the preferred option most of the time, so we will stick with that version. Observe how we use a single `transform.Translate` to move; we need to use the x axis of `movementValue` to move the x axis of our player but use the y axis of `movementValue` to move the z axis of our player. We don't want to move our player vertically, so that's why we needed to split the axis this way. The `InputValue` parameter has the `Get<Vector2>()` method, which will give us the current value of both axes, given `Vector2` is a variable that contains the x and y properties. Then, we multiply the vector by the movement or rotation speed according to the case. You will notice that we don't multiply by `Time.deltaTime` in the axis events, but we do that in the `Update`. That's because `Time.deltaTime` can change between frames, so storing the movement value considering the `Time.deltaTime` of the last time we moved the stick won't be useful for us. Also, notice how `movementValue` is a `Vector2`, just a combination of the x and y axes, while `lookValue` is a simple float. We did it this way because we will rotate our character only following the lateral movement of the mouse; we don't want to rotate it up and down. Check that we do `value.Get<Vector2>().x`, with emphasis on the `.x` part, where we extract just the horizontal

part of the axis for our calculations. Regarding the `PlayerShooting` component, we need to change it to this:

```
public void OnFire()
{
    GameObject clone = Instantiate(prefab);

    clone.transform.position = shootPoint.transform.position;
    clone.transform.rotation = shootPoint.transform.rotation;
}
```

Figure 6.63: PlayerShooting script using the new Input System

This case is simpler, as we don't need to execute the shooting behavior each frame, we only need to execute something at the very same moment the input is pressed, which is exactly when the `OnFire` event will be executed. If you need to also detect when the key was released, you can add the `InputValue` parameter as we did with `OnMove` and `OnLook`, and consult the `isPressed` property:

```
public void OnFire(InputValue value)
{
    if (value.isPressed)
    {
        GameObject clone = Instantiate(prefab);

        clone.transform.position = shootPoint.transform.position;
        clone.transform.rotation = shootPoint.transform.rotation;
    }
}
```

Figure 6.64: Getting the state of the button

Regarding the Visual Script Machine version of our scripts, first, you will need to refresh the **Visual Script Node Library** by going to **Edit | Project Settings | Visual Scripting** and clicking the **Regenerate Nodes** button. If you don't do this, you won't see the new Input System nodes:

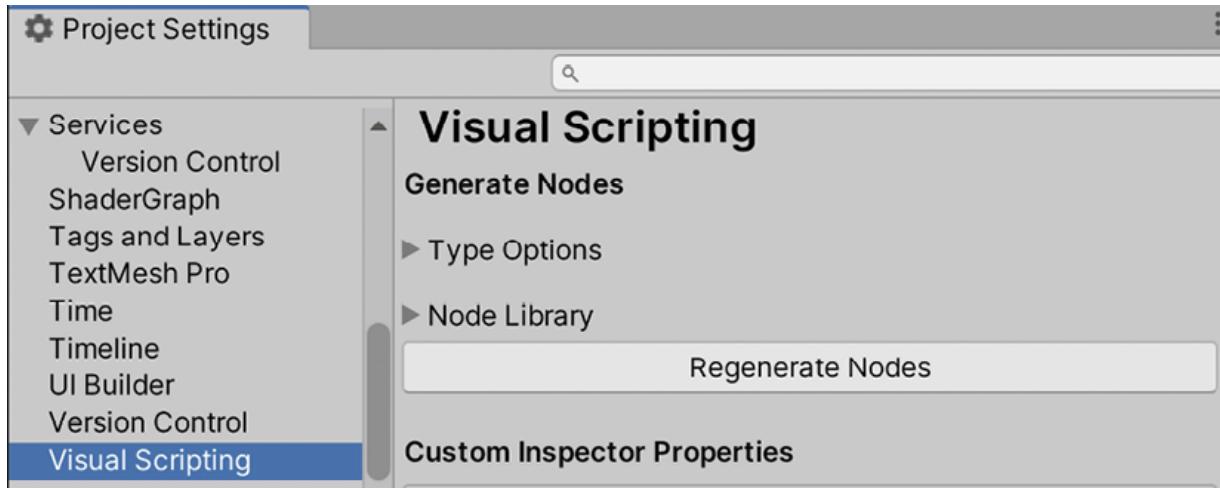


Figure 6.65: Regenerating Visual Scripting nodes to support the new Input System

Now, the `PlayerShooting` visual script would look like this:

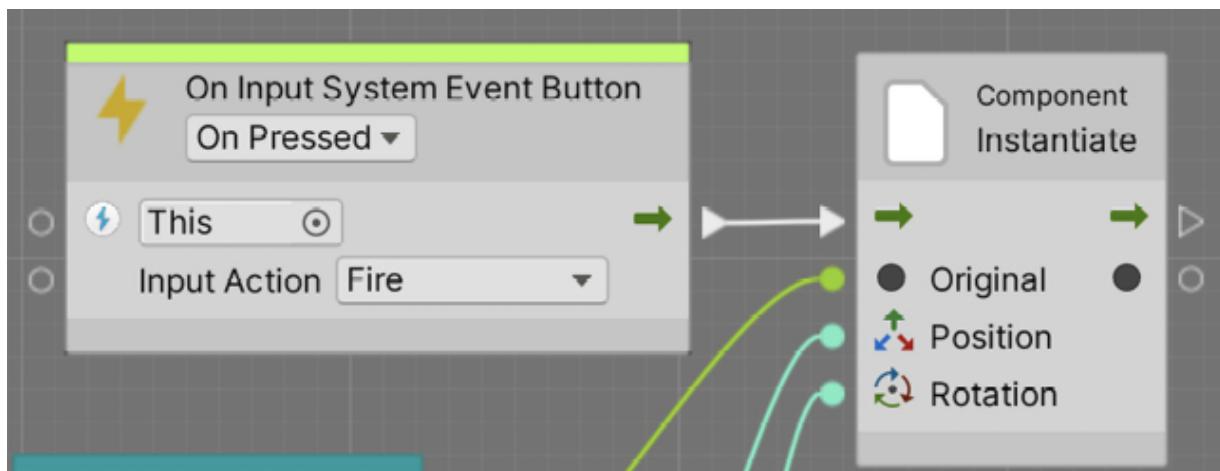


Figure 6.66: Instantiating bullets with the new input system

The new **On Input System Event Button** node allows us to detect when an action button has been pressed and react accordingly. You can pick the specific action in the **Input Action** parameter, and you can even make the node react to the pressure, release, or hold states of the button with the option right below the node's title. There is a bug where the **Input Action** property might not show any option; in such cases, try removing and adding the node again in the graph, and check that you added the `ScriptMachine` component to the same `GameObject` that has the `PlayerInput` component. Also, ensure you have selected the Player `GameObject` in the hierarchy. Regarding movement, it can be achieved this way:

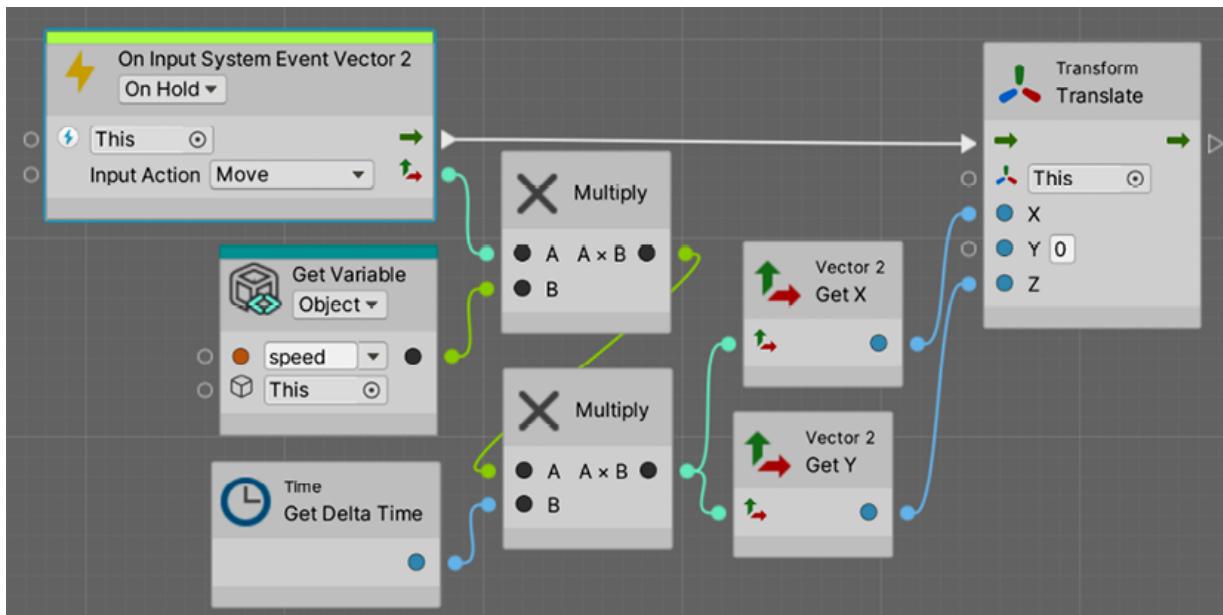


Figure 6.67: Moving with the new Input System

In this case, we used the **On Input System Event Vector2** node. This time, we used the `OnHold` mode, which means that, unlike the C# version, it won't execute just when the axis changes, but all the frames when the axis is pressed act like an `Update`; that, however, will only execute when the user is pressing the stick. The output pin of the node is the `Vector2` value, so we multiply it by the `speed` variable (declared in the `variables` component of our player) and

by `DeltaTime`. Finally, we use the `Vector2 GetX` and `Vector2 GetY` nodes to translate over the x and z axes. You may have trouble when rewiring the **Multiply** nodes with the new **Input System** node, given the return type is different compared to the previously used node (a `Vector2` instead of a single float). I recommend just deleting all nodes in this graph and redoing it to be sure everything is fine. This way we made our scripts react to the input messages from the new Unity's Input System.

Summary

We created our first real scripts in this chapter, which provide useful behavior. We discussed how to move a `GameObject` based on input and instantiate Prefabs via scripting to create objects at will according to the game situation. Also, we saw how to schedule actions, in this case, spawning, but this can be used to schedule anything. We saw how to destroy the created objects, to avoid increasing the number of objects to an unmanageable level. Finally, we explored the new Input System to provide maximum flexibility to customize our game's input. We will be using these actions to create other kinds of objects, such as sounds and effects, later in this book. Now you are able to create any type of movement or spawning logic your objects will need and make sure those objects are destroyed when needed. You might think that all games move and create shooting systems the same way, and while they are similar, being able to create your own movement and shooting scripts allows you to customize those aspects of the game to behave as intended and create the exact experience you are looking for. In the next chapter, we will be discussing how to detect collisions to prevent the player and bullets from passing through walls and much more.

7 Collisions and Health: Detecting Collisions Accurately

Join our book community on Discord

<https://packt.link/unitydev>



As games try to simulate real-world behaviors, one important aspect to simulate is physics, which dictates how objects move and how they collide with each other, such as in the collision of players and walls, or bullets and enemies. Physics can be difficult to control due to the myriad of reactions that can happen after a collision, so we will learn how to properly configure our game to create physics as accurately as we can. This will generate the desired arcade movement feeling but get realistic collisions working—after all, sometimes, real life is not as interesting as video games! In this chapter, we will examine the following collision concepts:

- Configuring physics
- Detecting collisions
- Moving with physics

First, we will learn how to properly configure physics, a step needed for the collisions between objects to be detected by our scripts, using new Unity events that we are also going to learn. All of this is needed in order to detect when our bullets touch our enemies and damage them. Then, we are going to discuss the difference between moving with `Transform`, as we have done so far, as well as moving

with Rigidbody and the pros and cons of each of these two methods. Both methods will be used to experiment with different ways of moving our player and let you decide which one you will want to use. Let's start by discussing physics settings.

Configuring physics

Unity's physics system is prepared to cover a great range of possible gameplay applications, so properly configuring it is important to get the desired result. In this section, we will examine the following physics settings concepts:

- Setting shapes
- Physics object types
- Filtering collisions

We are going to start by learning about the different kinds of colliders that Unity offers, and then learn about different ways to configure those to detect different kinds of physics reactions (**collisions** and **triggers**). Finally, we will discuss how to ignore collisions between specific objects to prevent situations such as the player's bullets damaging the player.

Setting shapes

At the beginning of this book, we learned that objects usually have two shapes, the visual shape—which is basically the 3D mesh—and the physical one, the collider—the one that the physics system will use to calculate collisions. Remember that the idea of this is to allow you to have a highly detailed visual model while having a simplified physics shape to increase the performance. Unity has several types of colliders, so here we will recap the common ones, starting with the primitive types, that is, **Box**, **Sphere**, and **Capsule**. These shapes are the cheapest ones (in terms of performance) to detect collisions due to the fact that the collisions

between them are done via mathematical formulae, unlike other colliders such as the **Mesh Collider**, which allows you to use any mesh as the physics body of the object, but with a higher performance cost and some limitations. The idea is that you should use a primitive type to represent your objects or a combination of them, for example, an airplane could be done with two Box colliders, one for the body and the other one for the wings. You can find an example of this in the following screenshot, where you can see a weapons collider made from primitives:

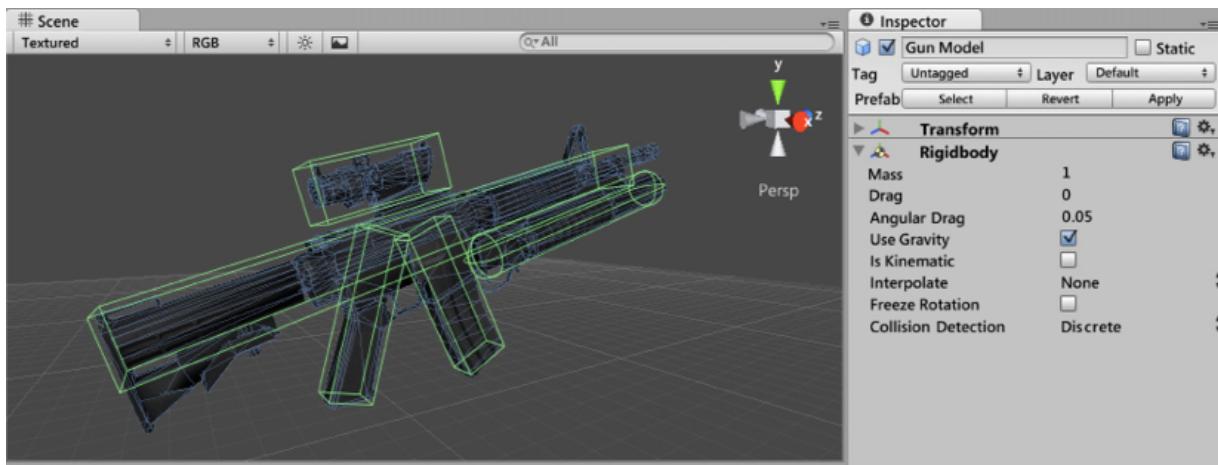


Figure 7.1: Compound colliders

Anyway, this is not always necessary; if we want the weapon to just fall to the ground, maybe a Box collider covering the entire weapon can be enough, considering those kinds of collisions don't need to be accurate, thereby increasing performance. Also, some shapes cannot be represented even with a combination of primitive shapes, such as ramps or pyramids, where your only solution is to use a Mesh collider, which asks for a 3D mesh to use for collisions, but we won't use them in this book given their high-performance impact; we will solve all of our physics colliders with primitives. Now, let's add the necessary colliders to our scene to prepare it to calculate collisions properly. Consider that if you used an Asset Store environment package other than mine, you may already have the scene modules with colliders; I will be showing the

work I needed to do in my case, but try to extrapolate the main ideas here to your scene. To add the colliders, follow these steps:

1. Select a wall in the base and check the object and possible child objects for collider components; in my case, I have no colliders. If you detect any Mesh collider, you can leave it if you want, but I would suggest you remove it and replace it with another option in the next step. The idea is to add the collider to it, but the problem I detected here is that, due to the fact my wall is not an instance of a Prefab, I need to add a collider to every wall in the scene.
2. One option is to create a Prefab and replace all of the walls with instances of the Prefab (the recommended solution) or to just select all walls in the Hierarchy (by clicking them while pressing *Ctrl* or *Cmd* on Mac) and, with them selected, use the **Add Component** button to add a collider to all of them. In my case, I will use the `Box Collider` component, which will adapt the size of the collider to the mesh. If it doesn't adapt, you can just change the **Size** and **Center** properties of the **Box Collider** to cover the entire wall:

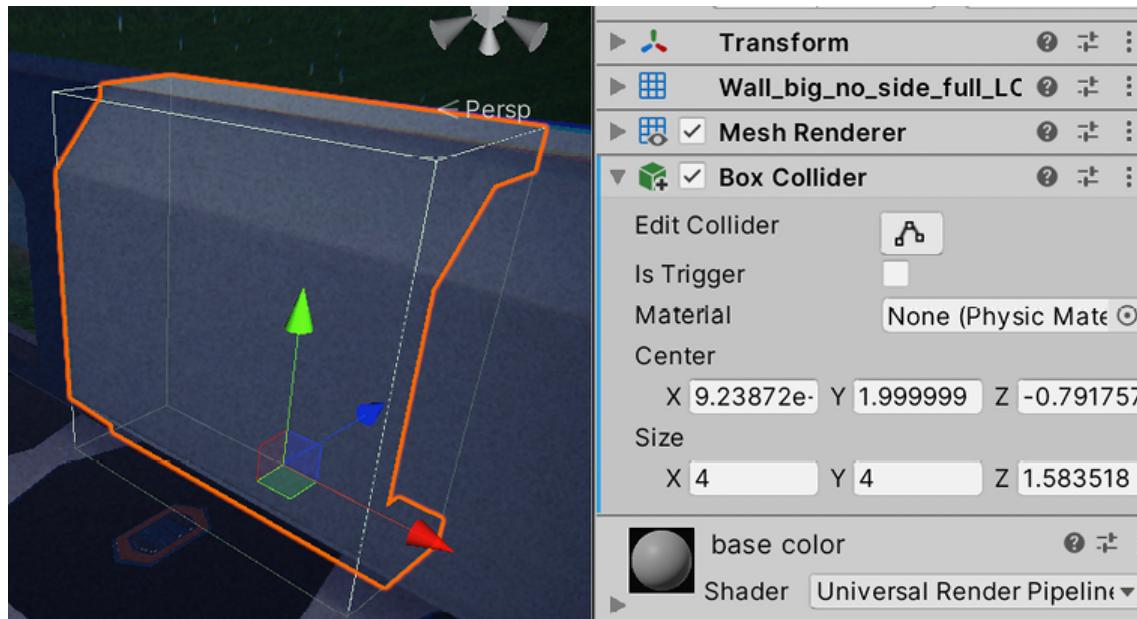


Figure 7.2: A Box Collider added to a wall

3. Repeat *steps 1 and 2* for the corners, floor tiles, and any other obstacle that will block player and enemy movement.

Now that we have added the needed colliders to the walls and floor, we can continue with the player and enemy. We will be adding the **Capsule Collider** to them, the usual collider to use in movable characters due to the fact that the rounded bottom will allow the object to smoothly climb ramps. Being horizontally rounded allows the object to easily rotate in corners without getting stuck, along with other conveniences of that shape. You might want to create an enemy Prefab based on one of the characters we downloaded before, so you can add the collider to that Prefab. Our player is a simple GameObject in the scene, so you will need to add the collider to that one, but do consider creating a Prefab for the player for convenience. You may be tempted to add several Box colliders to the bones of the character to create a realistic shape of the object. While we can we can use this approach to vary the damage based on where the enemies were shot on the body, note that we are primarily creating movement colliders, and using a capsule collider is sufficient for this purpose. In advanced damage systems, both capsule and Bone colliders will coexist, one for the movement and the other for damage detection; but we will simplify this in our game. Also, sometimes the collider won't adapt well to the visual shape of the object, and in my case, the Capsule collider didn't fit the character very well. I needed to fix its shape to match the character by setting its values as shown in the following screenshot: **Center** to `0, 1, 0`, **Radius** to `0.5`, and **Height** to `2`:

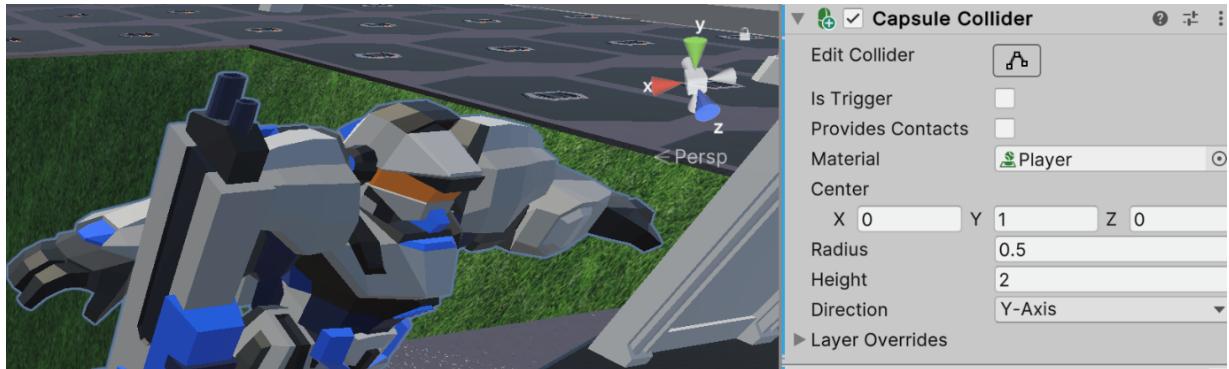


Figure 7.3: Character collider

The bullet we created with the Sphere already had a Sphere collider, but if you replaced the mesh of the bullet with another one, you might want to change the collider. For the time being, we don't need other objects in our game, so now that everything has its proper collider, let's see how to set the different physics settings to each object to enable proper collision detection. If you check the Terrain's components, you will see that it has its own kind of collider, the **Terrain Collider**. For Terrains, that's the only collider to use.

Physics object types

Now that we have added colliders to every object by making the objects have a presence in the physics simulation, it is time to configure them to have the exact physics behavior we want. We have a myriad of possible combinations of settings, but we will discuss a set of common profiles that cover most situations. Remember, besides colliders, we saw the Rigidbody component at the beginning of this book, which is the one that applies physics to the object. The following profiles are created with a combination of colliders and Rigidbody settings:

- **Static Collider:** As the name suggests, this kind of collider is not supposed to move, aside from some specific exceptions. Most of the environment objects fall into this category, such as

walls, floors, obstacles, and the terrain. These kinds of colliders are just colliders with no `Rigidbody` component, so they have a presence in the physics simulation but don't have any physics applied to them; they cannot be moved by other objects' collisions, they won't have physics, and they will be fixed in their position no matter what. Take into account that this has nothing to do with the **Static** checkbox at the top-right part of the editor; those are for systems we will explore later in several chapters (such as *Chapter 12, Enlightening Worlds: Illuminating Scenes with the Universal Render Pipeline*), so you can have a Static Collider with that checkbox unchecked if needed.

- **Physics Collider:** These are colliders with a `Rigidbody` component, like the example of the falling ball we did in the first part of this book. These are fully physics-driven objects that have gravity and can be moved through forces; other objects can push them and they perform every other physics reaction you can expect. You can use this for the player, grenade movement, falling crates, or in all objects in heavily physics-based games such as *The Incredible Machine*.
- **Kinematic Collider:** These are colliders that have a `Rigidbody` component but have the **Is Kinematic** checkbox checked. These don't have physics reactions to collisions and forces like **Static Colliders**, but they are expected to move, allowing **Physics Colliders** to properly handle collisions against them when moving. These can be used in objects that need to move using animations or custom scripting movements such as moving platforms.
- **Trigger Static Collider:** This is a regular Static Collider but with the **Is Trigger** checkbox of the collider checked. The difference is that kinematic and physics objects pass through it but by generating a `Trigger` event, an event that can be captured via scripting, which tells us that something is inside the collider. This event can be used to create buttons or trigger objects, in areas of the game when the player passes through something happening, such as a wave of enemies being

spawned, a door being opened, or winning the game in case that area is the goal of the player. Note that regular Static Colliders won't generate a trigger event when passing through this type because those aren't supposed to move.

- **Trigger Kinematic Collider:** Kinematic Colliders don't generate collisions, so they will pass through any other object, but they will generate `Trigger` events, so we can react via scripting. This can be used to create moveable power-ups that, when touched, disappear and give us points, or bullets that move with custom scripting movements and no physics, just straight like our bullets, but damage other objects when they touch them.

Of course, other profiles can exist aside from the specified ones to use in some games with specific gameplay requirements, but it's down to you to experiment with all possible combinations of physics settings to see whether they are useful for your case; the described profiles will cover 99% of cases. To recap the previous scenarios, I leave you with the following table showing the reaction of contact between all the types of colliders. You will find a row per each profile that can move; remember that static profiles aren't supposed to move. Each column represents the reaction when they collide with the other types: `Nothing` meaning the object will pass through with no effect, `Trigger` meaning the object will pass through but raise `Trigger` events, and `Collision` meaning the first object won't be able to pass through the second object:

| Collides with |
|---------------|---------------|---------------|---------------|---------------|
| Dynamic | Dynamic | Kinematic | Trigger | Static |
| Dynamic | Collision | Collision | Collision | Trigger |
| Kinematic | Nothing | Collision | Nothing | Trigger |
| Trigger | Trigger | Trigger | Trigger | Trigger |
| Kinematic | | | | |

Table 7.1: Collision Reaction Matrix

Considering this, let's start configuring the physics of our scene's objects. The walls, corners, floor tiles, and obstacles should use the Static Collider profile, so no `Rigidbody` component on them, and their colliders will have the **Is Trigger** checkbox unchecked:

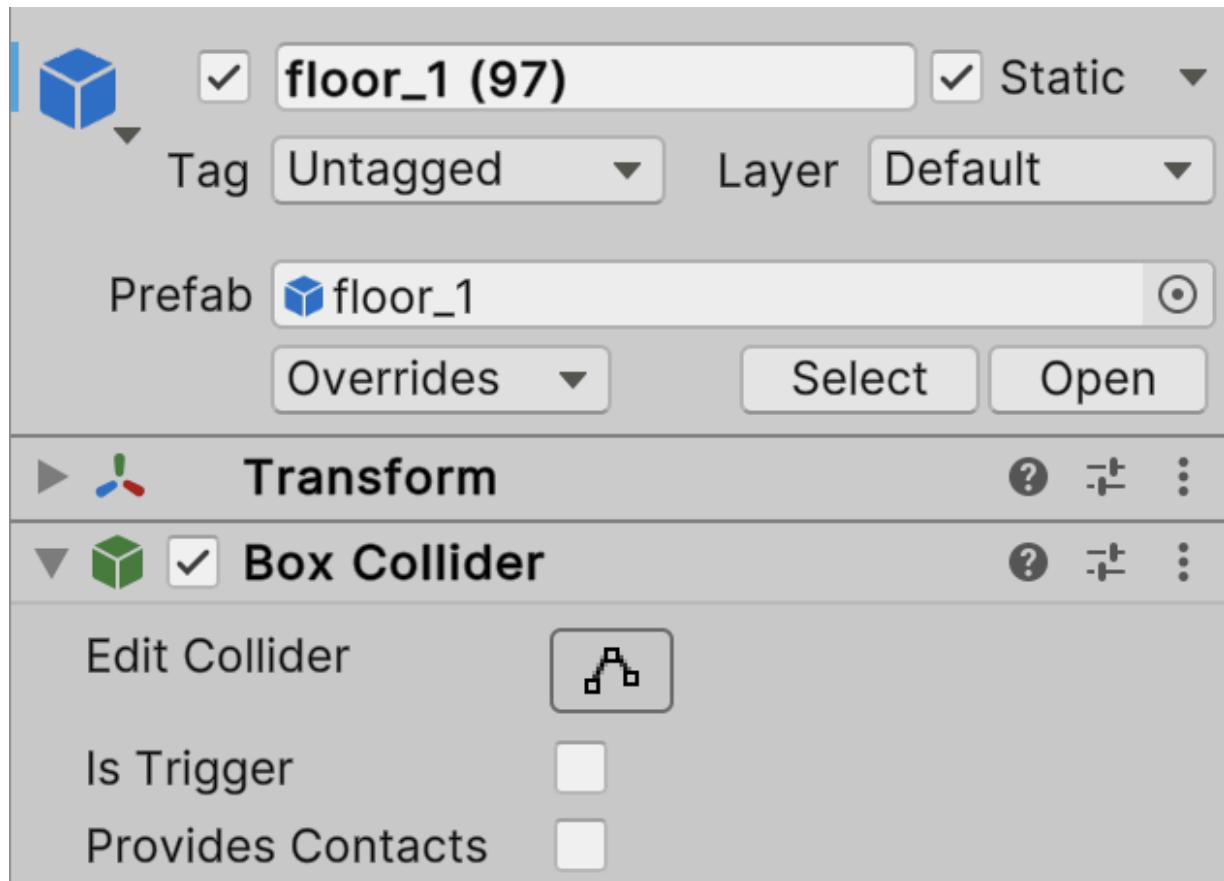


Figure 7.4: Configuration for floor tiles; remember the Static checkbox is for lighting only

The player should move and generate collisions against objects, so we need it to have a **Dynamic** profile. This profile will generate a funny behavior with our current movement script (which I encourage you to test), especially when colliding against walls, so it won't behave as you expected. We will deal with this later in this chapter:

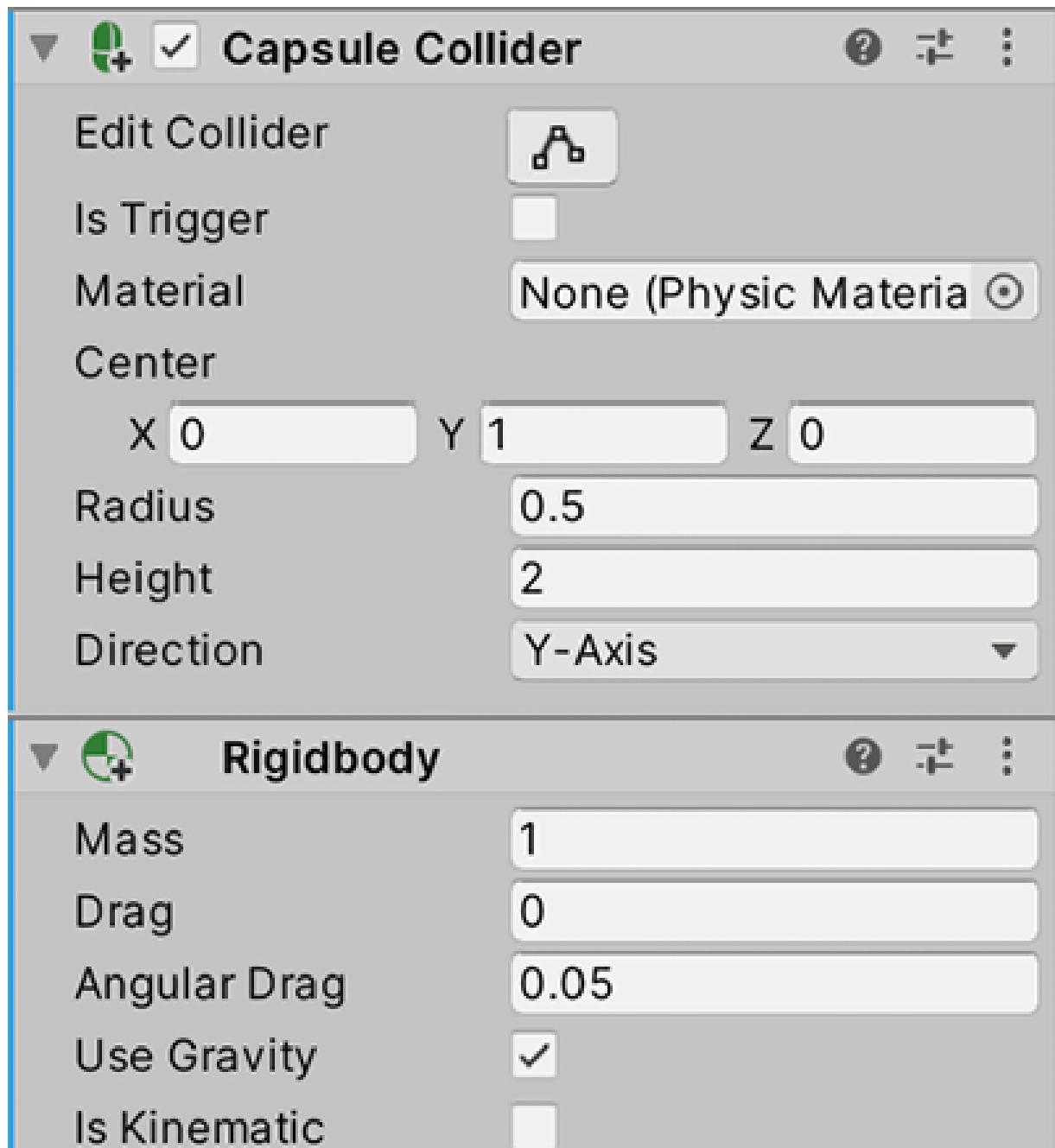


Figure 7.5: Dynamic settings on the player

The `Enemy` Prefab we suggested you create previously will be using the Kinematic profile because we will be moving this object with Unity's AI systems later in the book, so we don't need physics here, and as we want the player to collide against them, we need a collision reaction there, so there's no `Trigger` here:

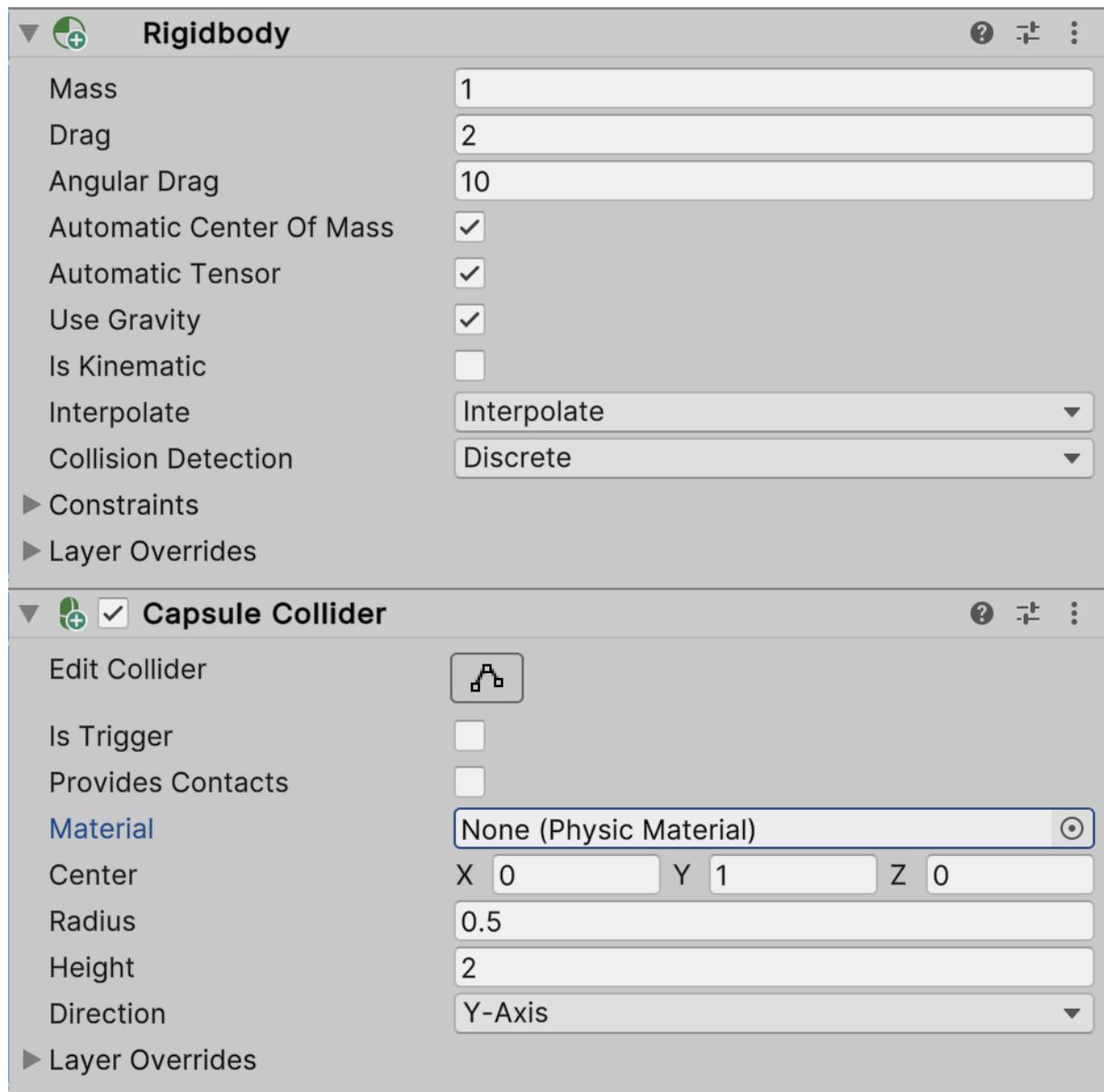


Figure 7.6: Kinematic setting for the enemy

For the `Bullet` Prefab, it moves with simplistic movement via scripting (it just moves forward), and not physics. We don't need collisions; we will code the bullet to destroy itself as soon as it touches something and will damage the collided object (if possible), so a Kinematic Trigger profile is enough for this one. We will use the `Trigger` event to script the contact reactions:

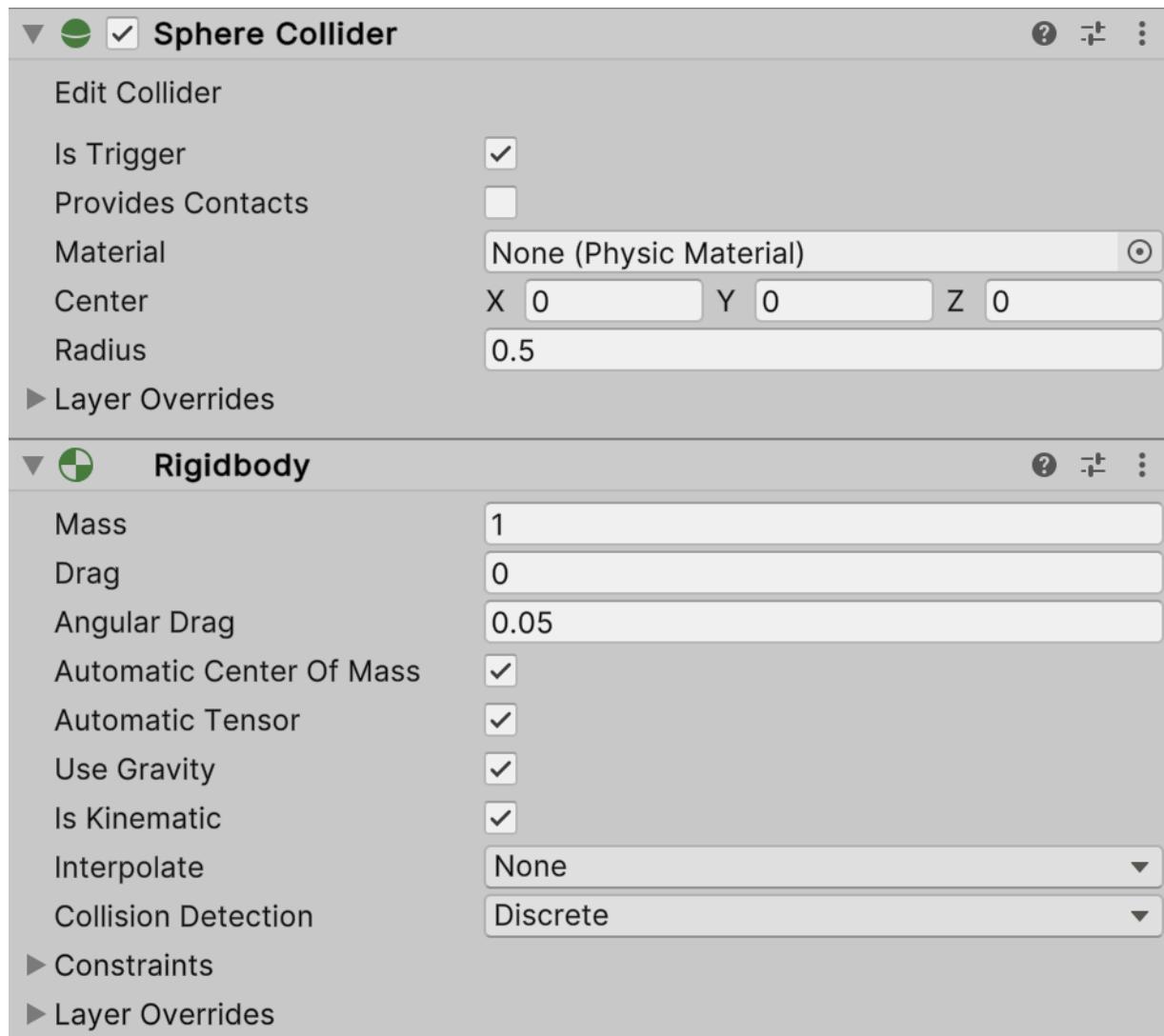


Figure 7.7: The Kinematic Trigger setting for our bullet; Is Trigger and Is Kinematic are checked

Now that we have properly configured the objects, let's check how to filter undesired collisions between certain object types.

Filtering collisions

Sometimes we want certain objects to ignore each other, like the bullets shot by the player, which shouldn't collide with the player itself. We can always filter that with an `if` statement in the C# script, checking whether the hit object is from the opposite team or

whatever filtering logic you want, but by then, it is too late; the physics system wasted resources by checking a collision between objects that were never meant to collide. Here is where the Layer Collision Matrix can help us. The **Layer Collision Matrix** sounds scary, but it is a simple setting of the physics system that allows us to specify which groups of objects should collide with other groups. For example, the player's bullets should collide with enemies, and enemy bullets should collide with the player. In this case, the enemies' bullets will pass through enemies, but this is desired in our case. The idea is to create those groups and put our objects inside them; in Unity, those groups are called **layers**. We can create layers and set the layer property of the GameObject (the top part of the Inspector) to assign the object to that group or layer. Note that you have a limited number of layers, so try to use them wisely. We can achieve this by doing the following:

1. Go to Edit | Project Settings and, inside it, look for the Tags and Layers option from the left pane:

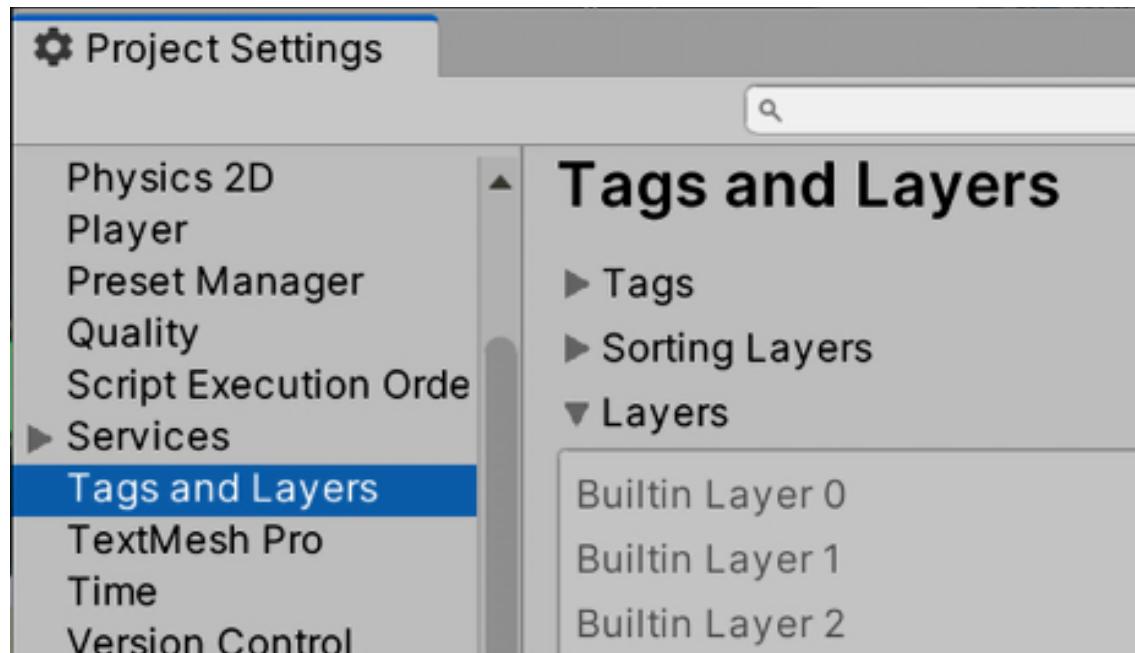


Figure 7.8: The Tags and Layers settings

2. From the **Layers** section, fill the empty spaces to create layers. We will use this for the bullet scenario, so we need four layers: `Player`, `Enemy`, `PlayerBullet`, and `EnemyBullet`:



Figure 7.9: Creating layers

3. Select the `Player` GameObject in the Hierarchy and, from the top part of the Inspector, change the **Layer** property to `Player`. Also, change the `Enemy` Prefab to have the `Enemy` layer. A window will show, asking you whether you want to also change the child objects; select **Yes**:

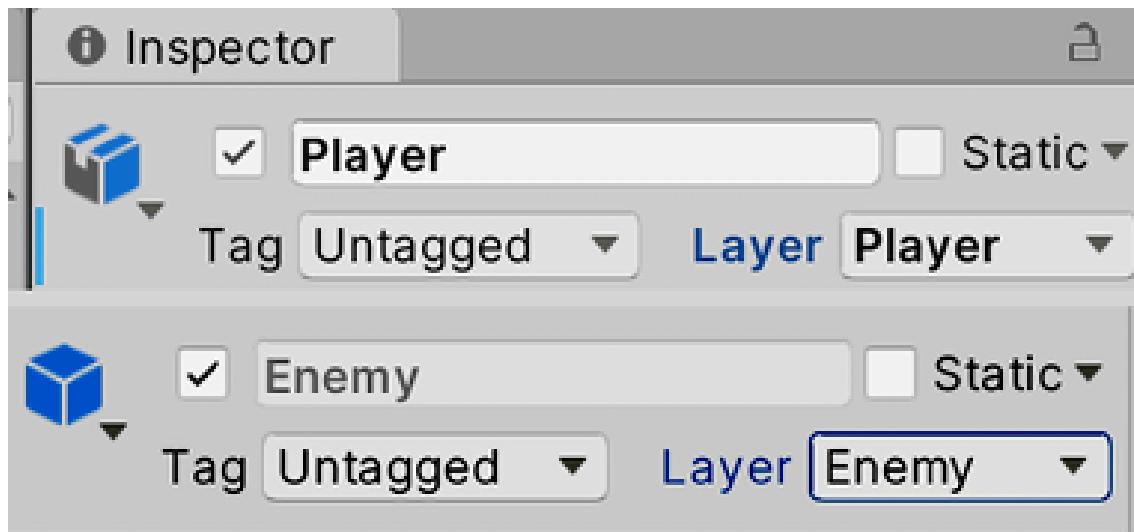


Figure 7.10: Changing the layers of the player and the enemy Prefab

4. In the case of the bullet, we have a problem; we have one Prefab but two layers, and a Prefab can only have one layer. We

have two options: changing the layer according to the shooter via scripting, or having two bullet Prefabs with different layers. For simplicity, I will choose the latter, also taking the chance to apply another material to the enemy bullet to make it look different.

5. We will be creating a Prefab **Variant** of the player bullet. Remember that a Variant is a Prefab that is based on an original one like class inheritance. When the original Prefab changes, the Variant will change, but the Variant can have differences, which will make it unique.
6. Drop a bullet Prefab into the scene to create an instance.
7. Drag the instance again to the `Prefabs` folder, this time selecting the **Prefab Variant** option in the window that will appear.
8. Rename it `Enemy Bullet`.
9. Destroy the Prefab instance in the scene.
10. Create a second material similar to the player bullet with a different color and put it on the enemy bullet Prefab Variant.
11. Select the enemy bullet Prefab, set its layer to `EnemyBullet`, and do the same for the original Prefab (`PlayerBullet`). Even if you changed the original Prefab layer, as the Variant modified it, the modified version (or override) will prevail, allowing each Prefab to have its own layer.

Now that we have configured the layers, let's configure the physics system to use them:

1. Go to **Edit | Project Settings** and look for the **Physics** settings (not **Physics 2D**).
2. Scroll down until you see the **Layer Collision Matrix**, a half grid of checkboxes. You will notice that each column and row is labeled with the names of the layers, so each checkbox in the cross of a row and column will allow us to specify whether these two should collide. In our case, we configured it as shown in the following screenshot so that player bullets do not hit the

player or other player bullets, and enemy bullets do not hit enemies or other enemy bullets:

	Default	TransparentFX	Ignore Raycast	Water	UI	Player	Enemy	PlayerBullet	EnemyBullet
Default	✓	✓	✓	✓	✓	✓	✓	✓	✓
TransparentFX	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ignore Raycast	✓	✓	✓	✓	✓	✓	✓	✓	✓
Water	✓	✓	✓	✓	✓	✓	✓	✓	✓
UI	✓	✓	✓	✓	✓	✓	✓	✓	✓
Player	✓								
Enemy		✓							
PlayerBullet									
EnemyBullet									

Figure 7.11: Making player bullets collide with enemies and enemy bullets with the player

It is worth noticing that sometimes filtering logic won't be that rigid or predictable. For example, it might involve only hit objects that have a certain amount of life, objects that don't have an invisibility

temporal buff, or conditions that can change during the game and are difficult to generate for all possible layers for all possible groups. So, in these cases, we should rely on manual filtering after the **Trigger** or **Collision** event. Now that we have filtered collisions, let's check whether our settings are working properly by reacting to collisions in the next section.

Detecting collisions

As you can see, proper physics settings can be complicated and very important, but now that we have tackled that, let's perform some task with those settings by reacting to the contact in different ways and creating a **health system** in the process. In this section, we will examine the following collision concepts:

- Detecting Trigger events
- Modifying the other object

First, we are going to explore the different collision and trigger events Unity offers to react to contact between two objects through the Unity collision events. This allows us to execute any reaction code we want to place, but we are going to explore how to modify the contacted object components using the `GetComponent` function.

Detecting Trigger events

If objects are properly configured, as previously discussed, we can get two reactions: collisions or triggers. The **Collision** reaction has a default effect that blocks the movement of the objects, but we can add custom behavior on top of that using scripting; but with a **Trigger**, unless we add custom behavior, it won't produce any noticeable effect. Either way, we can script reactions to both possible scenarios such as adding a score, reducing health, and losing the game. To do so, we can use the suite of **Physics events**. These events are split into two groups, **Collision events**

and **Trigger events**, so according to your object setting, you will need to pick the appropriate group. Both groups have three main events, **Enter**, **Stay**, and **Exit**, telling us when a collision or trigger began (*Enter*), whether they are still happening or are still in contact (*Stay*), and when they stopped contacting (*Exit*). For example, we can script a behavior such as playing a sound when two objects first make contact in the Enter event, such as a friction sound, and stop it when the contact ends, in the Exit event. Let's test this by creating our first contact behavior: the bullet being destroyed when coming into contact with something. Remember that the bullets are configured to be triggers, so they will generate `Trigger` events on contact with anything. You can do this with the following steps:

1. Create and add a script called `ContactDestroyer` on the **Player Bullet** Prefab; as the **Enemy Bullet** Prefab is a Variant of it, it will also have the same script.
2. To detect when a trigger happens, such as with **Start** and **Update**, create an event function named `OnTriggerEnter`.
3. Inside the event, use the `Destroy(gameObject);` line to make the bullet destroy itself when touching something:

```
public class ContactDestroyer : MonoBehaviour
{
    void OnTriggerEnter()
    {
        Destroy(gameObject);
    }
}
```

Figure 7.12: Auto-destroying on contact with something

4. Save the script and shoot the bullets against the walls to see how they disappear instead of passing through them. Here we don't have a collision, but a trigger that destroys the bullet on contact. So, this way, we are sure that the bullet will never pass through anything, but we are still not implementing a physics-based movement.

After enabling these components, now, we won't need the other Collision events, but if you need them, they will work similarly; just create a function called `OnCollisionEnter` instead. Now, let's explore another version of the same function. We'll configure it to not only tell us that we hit something but also what we came into contact with. We will use this to make our **Contact Destroyer** also destroy the other object. To do this, follow these steps:

1. Replace the `OnTriggerEnter` method signature with the one in the following screenshot. This one receives a parameter of the `Collider` type, indicating the exact collider that hit us:

```
void OnTriggerEnter(Collider other)
```

Figure 7.13: Version of the trigger event that tells us which object we collided with

2. We can access the `GameObject` of that collider using the `gameObject` property. We can use this to destroy the other one as well, as shown in the following screenshot. If we just use the `Destroy` function by passing the `other` variable, it will only destroy the `Collider` component:

```

void OnTriggerEnter(Collider other)
{
    Destroy(gameObject);
    Destroy(other.gameObject);
}

```

Figure 7.14: Destroying both objects

3. Save and test the script. You will notice that the bullet will destroy everything it touches. Remember to verify that your enemy has a Capsule collider for the bullet to detect collisions against it.

The equivalent version in Visual Scripting would be like the following figure:

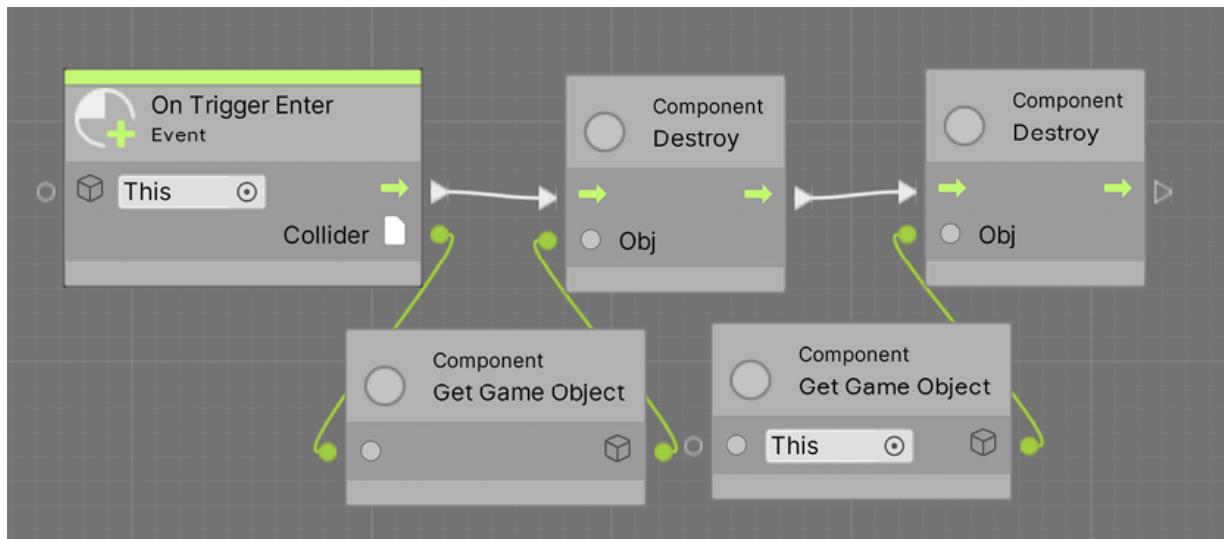


Figure 7.15: Destroying both objects with Visual Scripting

As you can see, we created an **On Trigger Enter** node and chained it to two **Destroy** nodes. To specify which object each **Destroy**

node will destroy, we used the **Component: Get GameObject** node twice. The right one was created with no node connected to its left input pin, which means it will return the GameObject that is currently executing this script (hence, the **This** label in the node left pin), in this case, the bullet. For the second one we needed to connect the **Collider** output pin at the right of the **OnTriggerEnter** node to the **Get GameObject** node; this way we specify that we want to obtain the GameObject that contains the collider our bullet collided with. Now, in our game we don't want the bullet to destroy everything on contact; instead, we will make the enemies and the player have a life amount; the bullets will reduce that life amount until it reaches 0, so let's find out how to do that.

Modifying the other object

For the bullet to damage the collided object, we will need to access a `Life` component to change its amount, so we will need to create this `Life` component to hold a float field with the amount of life. Every object with this component will be considered a damageable object. To access the `Life` component from our bullet scripts, we will need the `GetComponent` function to help us. If you have a reference to a `GameObject` or component, you can use `GetComponent` to access a specific component if the object contains it (if not, it will return `null`). Let's see how to use that function to make the bullet lower the amount of life of the other object:

1. Create and add a `Life` component with a `public float` field called `amount` to both the player and enemy Prefabs. Remember to set the value `100` (or whatever life amount you want to give them) in the **Amount** field for both in the Inspector:

```

public class Life : MonoBehaviour
{
    public float amount;
}

```

Figure 7.16: The Life component

2. Remove the `ContactDestroyer` component from the player bullet, which will also remove it from the Enemy Bullet Variant.
3. Add a new script called `ContactDamager` to both the enemy and player.
4. Add an `OnTriggerEnter` event that receives the `other` collider as a parameter and just add the `Destroy` function call that auto-destroys itself, not the one that destroys the other object; our script won't be responsible for destroying it, just reducing its life.
5. Add a float field called `damage`, so we can configure the amount of damage to inflict on the other object. Remember to save the file and set a value before continuing.
6. Use `GetComponent` on the reference to the other collider to get a reference to its `Life` component and save it in a variable:

```

Life life = other.GetComponent<Life>();

```

Figure 7.17: Accessing the collided object's Life component

7. Before reducing the life of the object, we must check that the `Life` reference isn't `null`, which would happen if the other object doesn't have the `Life` component, as in the case of walls

and obstacles. The idea is that the bullet will destroy itself when anything collides with it and reduce the life of the other object if it is a damageable object that contains the `Life` component.

In the following screenshot, you will find the full script:

```
public class ContactDamager : MonoBehaviour
{
    public float damage;

    void OnTriggerEnter(Collider other)
    {
        Destroy(gameObject);

        Life life = other.GetComponent<Life>();

        if (life != null)
        {
            life.amount -= damage;
        }
    }
}
```

Figure 7.18: Reducing the life of the collided object

1. Place an enemy in the scene and set its speed to `0` to prevent it from moving.

2. Select it in the Hierarchy before hitting **Play** and start shooting at it.

You can see how the life value reduces in the Inspector. You can also press the Esc key to regain control of the mouse and select the object while in **Play** mode to see the life field change during the runtime in the editor. Now, you will notice that life is decreasing, but it will become negative; we want the object to destroy itself when life is below 0 instead. We can do this in two ways: one is to add an `Update` to the `Life` component, which will check all of the frames to see whether life is below 0, destroying itself when that happens. The second way is by encapsulating the life field and checking its value inside the setter to prevent all frames from being checked. I would prefer the second way, but we will implement the first one to make our scripts as simple as possible for beginners. To do this, follow these steps:

1. Add `Update` to the `Life` component.
2. Add `If` to check whether the amount field is below or equals 0.
3. Add `Destroy` in case the `if` condition is `true`.
4. The full `Life` script will look like the following screenshot:

```
public class Life : MonoBehaviour
{
    public float amount;

    void Update()
    {
        if (amount <= 0)
        {
            Destroy(gameObject);
        }
    }
}
```

Figure 7.19: The Life component

5. Save and see how the object is destroyed once `Life` becomes `0`.

The Visual Scripting version for the `Life` component would look like this:

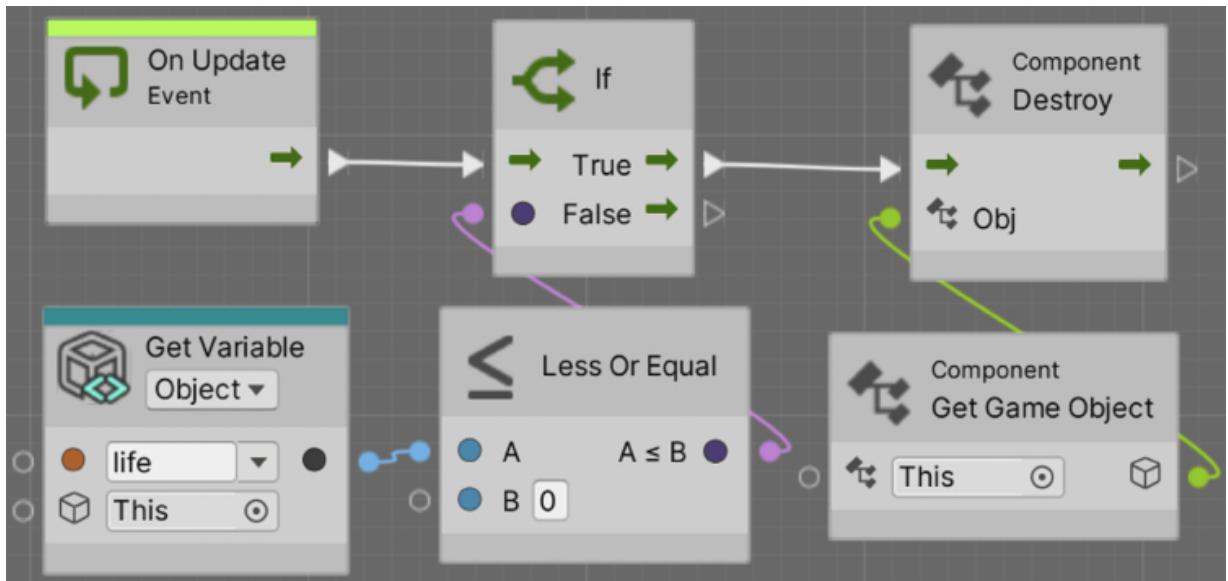


Figure 7.20: The Life component in Visual Scripting

The script is pretty straightforward—we check if our `Life` variable is less than 0 and then destroy ourselves as we did previously. Now, let's check the **Damager** script:

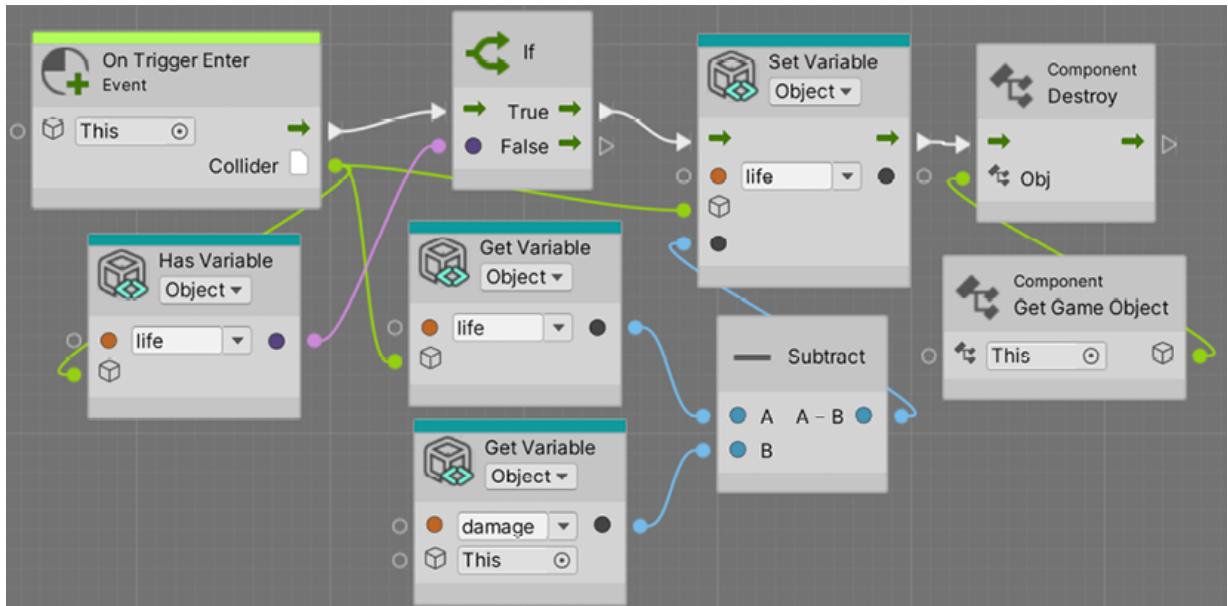


Figure 7.21: The Damager component in Visual Scripting

This version is a little bit different from our C# counterpart. At first glance it looks the same: we use **Get Variable** as before to read the life, then we use the **Subtract** node to subtract **damage** from **life**, and the result of that calculation becomes the new value of **life**, with the **Set Variable** node used to alter the current value of that variable. The first difference we can see here is the absence of any `GetComponent` node. In C# we used that instruction to get the `Life` component on the collided object in order to read and alter its **amount** variable, reducing the remaining life. But as in Visual Scripting, our node graphs don't have variables, so we don't need to access the component to read them. Instead, knowing that the enemy has a `Life` variable in its **Variables** component, we use the **Get Variable** node, connecting it to the collider we hit (the **Collider** output pin of **On Trigger Enter**), so essentially we are reading the value of the `Life` variable of the collided object. The same goes for changing its value: we use the **Set Value** node, connecting it to the collider, specifying we want to alter the value of the `Life` variable of the collider object, not ours (as bullets don't have a `Life` variable). Note that this can raise an error if the collided object doesn't have the `Life` variable, which is why we added the **Object Has Variable** node, which checks if the object has a variable called `Life`. If it doesn't, we just do nothing, which is useful when we collide with walls or other non-destructible objects. Finally, we make the **Damager** (the bullet in this case) auto-destroy itself.

You can instantiate an object when this happens such as a sound, a particle, or maybe a power-up. I will leave this as a challenge for you. By using a similar script, you can implement a life power-up that increases the life value or a speed power-up that accesses the PlayerMovement script and increases the Speed field; from now on, use your imagination to create exciting behaviors using the previous acquired knowledge.

Now that we have explored how to detect collisions and react to them, let's explore how to fix the player falling when hitting a wall.

Moving with physics

So far, the player, the only object that moves with the **Dynamic Collider Profile** and the one that will move with physics, is actually moving through custom scripting using the Transform API. Every dynamic object should instead move using the Rigidbody API functions in a way the physics system understands better. As such, here we will explore how to move objects, this time through the Rigidbody component. In this section, we will examine the following physics movement concepts:

- Applying forces
- Tweaking physics

We will start by seeing how to move objects the correct physical way, through forces, and we will apply this concept to the movement of our player. Then, we will explore why real physics is not always fun, and how we can tweak the physics properties of our objects to have a more responsive and appealing behavior.

Applying forces

The physically accurate way of moving an object is through forces, which affect the object's velocity. To apply force, we need to access `Rigidbody` instead of `Transform` and use the `AddForce` and `AddTorque` functions to move and rotate respectively. These are functions where you can specify the amount of force to apply to each axis of position and rotation. This method of movement will have full physics reactions; the forces will accumulate on the velocity to start moving and will suffer drag effects that will make the speed slowly decrease, and the most important aspect here is that they will collide against walls, blocking the object's way. To get this kind of movement, we can do the following:

1. Create a `Rigidbody` field in the `PlayerMovement` script, but this time, make it `private`, meaning, do not write the `public`

keyword in the field, which will make it disappear in the editor; we will get the reference another way:

A screenshot of a Unity code editor showing a script with a single line of code: `private Rigidbody rb;`. The word "private" is in blue, "Rigidbody" is in purple, and "rb" is in green. A yellow squiggly underline is under "rb", indicating it is a potential typo or error.

Figure 7.22: The private Rigidbody reference field

2. Note that we named this variable `rb` just to prevent our scripts from being too wide, making the screenshots of the code in the book too small. It's recommended to call the variable properly in your scripts—in this case, it would be named `rigidbody`.
3. Using `GetComponent` in the `Start` event function, get our `Rigidbody` and save it in the field. We will use this field to cache the result of the `GetComponent` function; calling that function every frame to access the `Rigidbody` is not performant. Also, you can notice here that the `GetComponent` function can be used to retrieve not only components from other objects (like the collision example) but also your own:

```
private void Awake()
{
    Cursor.visible = false;
    Cursor.lockState = CursorLockMode.Locked;

    rb = GetComponent<Rigidbody>();
}
```

Figure 7.23: Caching the Rigidbody reference for future usage

4. Replace the `transform.Translate` calls with `rb.AddRelativeForce`. This will call the add force functions of

the Rigidbody, specifically the relative ones, which will consider the current rotation of the object. For example, if you specify a force in the z-axis (the third parameter), the object will apply its force along with its forward vector.

5. Replace the `transform.Rotate` calls with `rb.AddRelativeTorque`, which will apply rotation forces:

```
void Update()
{
    rb.AddRelativeForce(
        movementValue.x * Time.deltaTime,
        0,
        movementValue.y * Time.deltaTime);

    rb.AddRelativeTorque(0, lookValue* Time.deltaTime, 0);
}
```

Figure 7.24: Using the Rigidbody forces API

6. Check that the player GameObject capsule collider is not intersecting with the floor, but just a little bit over it. If the player is intersecting, the movement won't work properly. If this is the case, move it upward.

If you used Unity before, you might find odd to use `Update` instead of `FixedUpdate` to apply physics forces. `FixedUpdate` is a special update that runs at a fixed rate, regardless of the actual game's FPS, and here is where the Physics system executes. It is configured by default to run 50 times per frame. This means that if the game is running at 200 FPS, the `FixedUpdate` will execute every 4 frames, but if the game is running at 25 FPS, the fixed update will execute twice per frame. This is done this way to enhance physics calculations stability, given their complexity. While it would be correct to call any Rigidbody method that applies forces and torque in the

FixedUpdate, is not necessarily wrong to do that in the Update method. For simplicity we kept our code in the Update method, given FixedUpdate can be tricky to use for beginners, as it could execute more than once per frame or even skip some frames. One example is checking if a key is pressed using methods like Input.GetKeyDown, given the key pressure happens in specific frames. If you call that method in the FixedUpdate, and FixedUpdate skipped the frame where the key was pressed, the key pressure won't detected, making the game feel unresponsive. A classic fix would be to detect key presses in the Update method and store if they were pressed in boolean variables, to later check them in the FixedUpdate. But again, due to simplicity we decided to leave it as is. In the Visual Scripting version, the change is the same; replace the **Transform** and **Rotate** nodes with **Add Relative Force** and **Add Relative Torque** nodes. An example of **Add Relative Force** would be the following one:

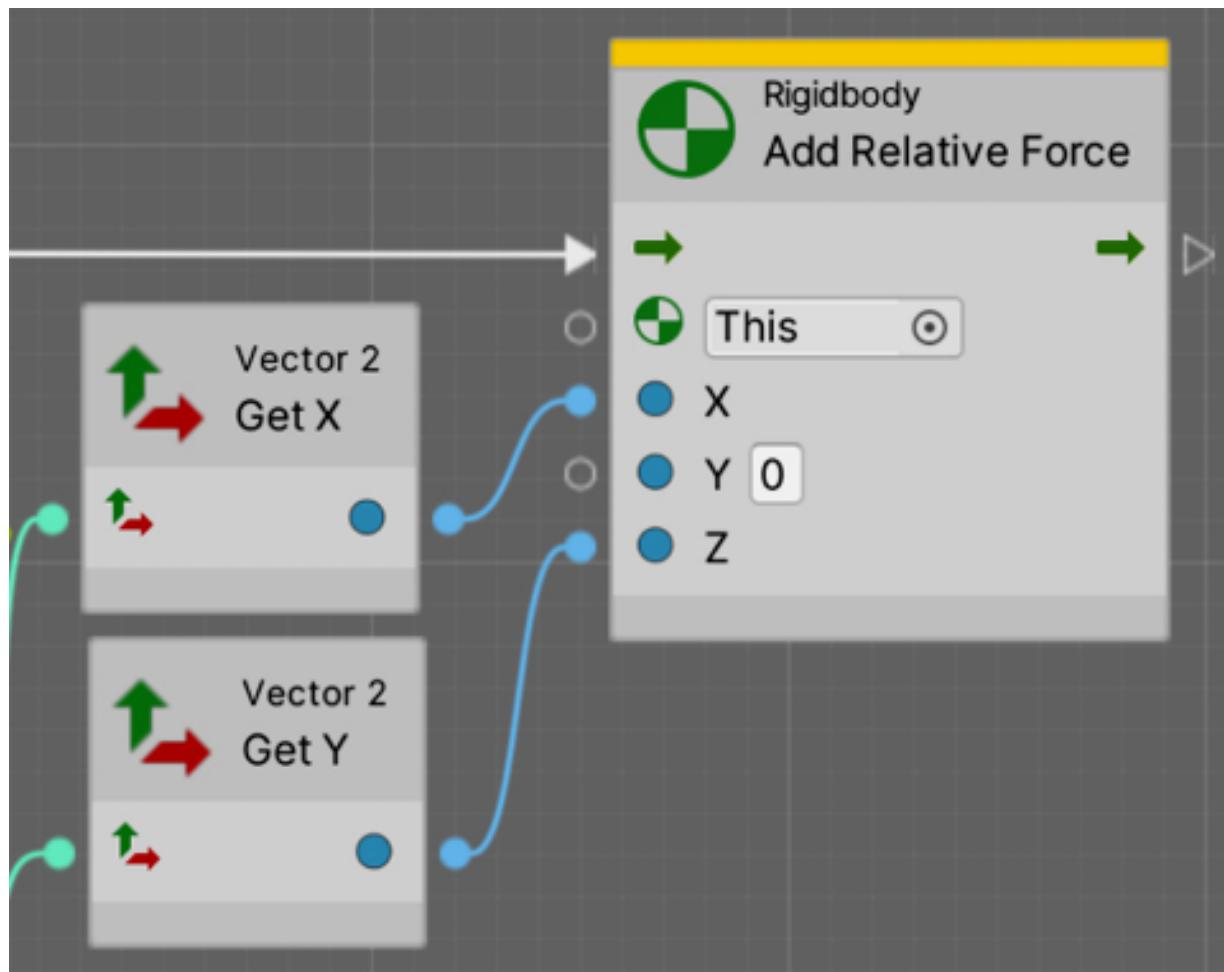


Figure 7.25: Using the Rigidbody forces API

And for rotation like this:

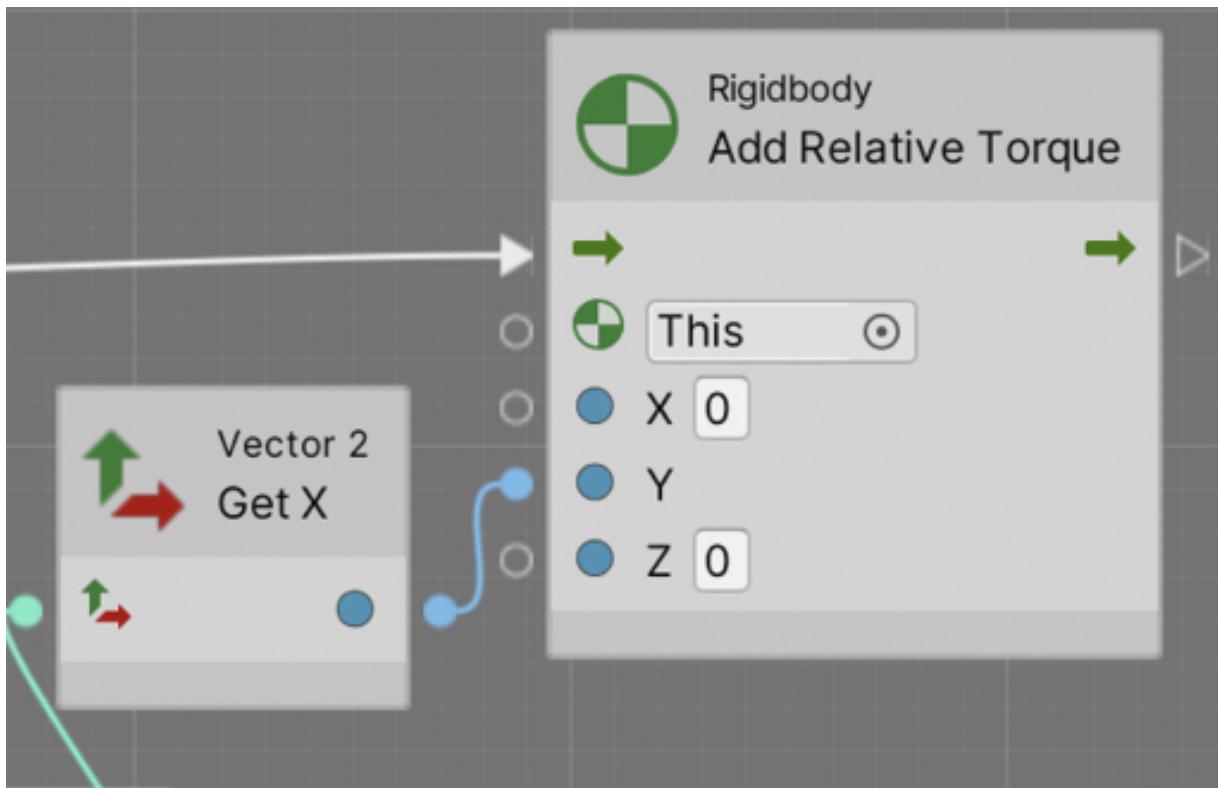


Figure 7.26: Using the Rigidbody torque API

You can see that we don't need to use the **GetComponent** nodes here either, given that just using the **Add Relative Force** or **Torque** nodes makes Visual Scripting understand that we want to apply those actions on our own Rigidbody component (explaining again the **This** label). If in any other case we need to call those functions on a Rigidbody other than ours, we would need the **GetComponent** node there, but let's explore that later. Now, if you save and test the results, you will probably find the player falling and that's because now we are using real physics, which contains floor friction, and due to the force being applied at the center of gravity, it will make the object fall. Remember that, in terms of physics, you are a capsule; you don't have legs to move, and here is where standard physics is not suitable for our game. The solution is to tweak physics to emulate the kind of behavior we need.

Tweaking physics

To make our player move like in a regular platformer game, we will need to freeze certain axes to prevent the object from falling. Remove the friction to the ground and increase the air friction (drag) to make the player reduce its speed automatically when releasing the keys. To do this, follow these steps:

1. In the `Rigidbody` component, look at the **Constraints** section at the bottom and check the **X** and **Z** axes of the **Freeze Rotation** property:

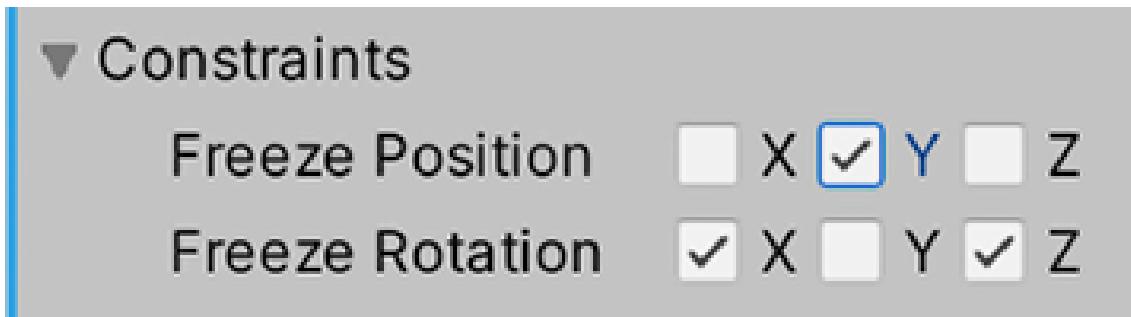


Figure 7.27: Freezing rotation axes

2. This will prevent the object from falling sideways but will allow the object to rotate horizontally. You might also freeze the *y*-axis of the **Freeze Position** property if you don't want the player to jump, preventing some undesired vertical movement on collisions.
3. You will probably need to change the speed values because you changed from a meters-per-second value to newtons-per-second, the expected value of the **Add Force** and **Add Torque** functions. Using 1,000 in speed and 160 in rotation speed was enough for me.
4. Now, you will probably notice that the speed will increase a lot over time, as will the rotation. Remember that you are using forces, which affects your velocity. When you stop applying forces, the velocity is preserved, and that's why the player kill keeps rotating even if you are not moving the mouse. The fix to this is to increase the **Drag** and **Angular Drag**, which

emulates air friction, and will reduce the movement and rotation respectively when no force is applied. Experiment with values that you see suitable; in my case, I used `2` for **Drag** and `10` for **Angular Drag**, needing to increase **Rotation Speed** to `150` to compensate for the drag increase:

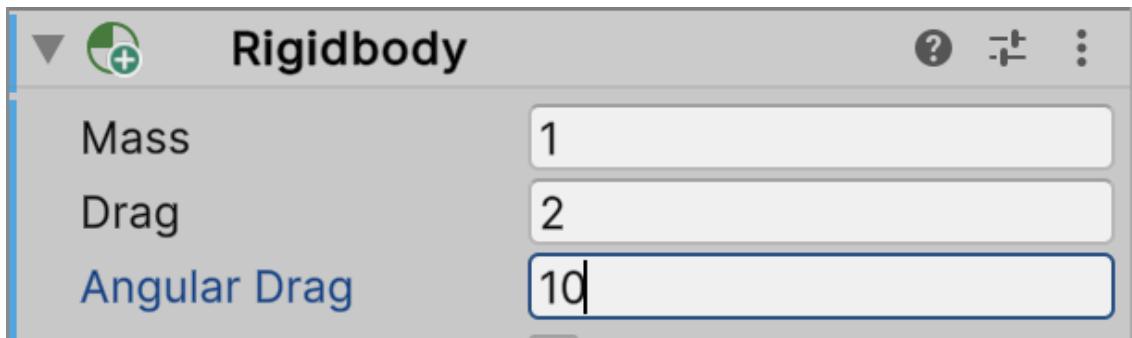


Figure 7.28: Setting air friction for rotation and movement

5. Now, if you move while touching the wall, instead of sliding, like in most games, your Player will stick to the obstacles due to contact friction. We can remove this by creating a `Physics Material`, an asset that can be assigned to the colliders to control how they react in those scenarios.
6. Start creating one by clicking on the `+` button from the **Project** window and selecting **Physics Material** (not the 2D version). Call it `Player` and remember to put it in a folder for those kinds of assets.
7. Select it and set **Static Friction** and **Dynamic Friction** to `0`, and **Friction Combine** to `Minimum`, which will make the **Physics** system pick the minimum friction of the two colliding objects, which is always the minimum—in our case, zero:

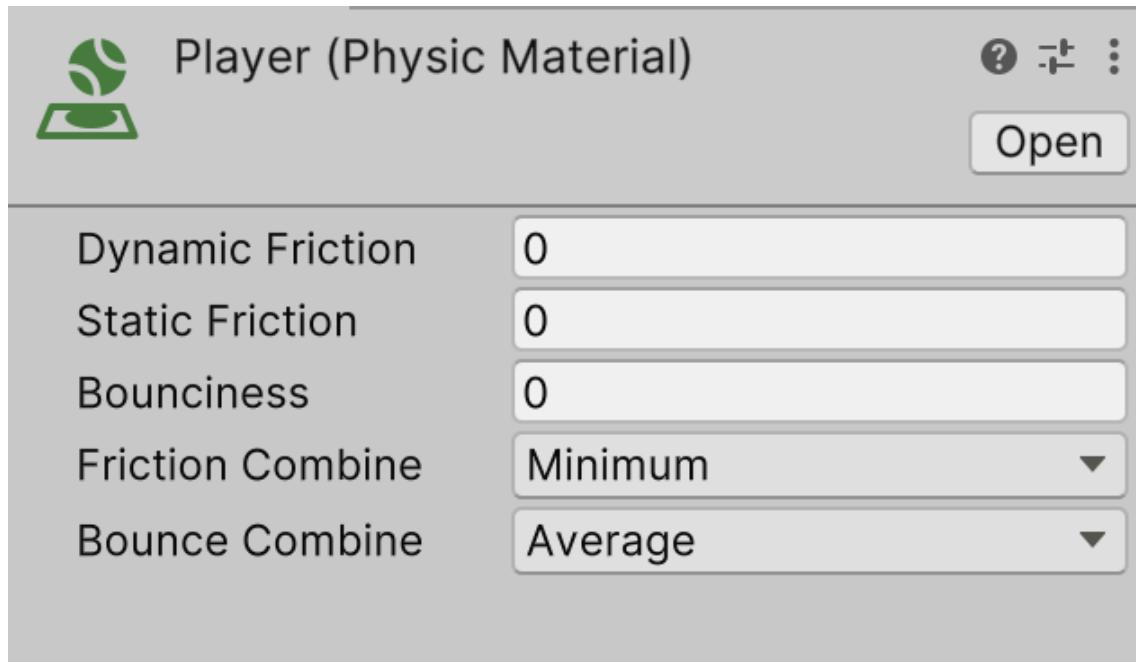


Figure 7.29: Creating a physics material

8. Select the player and drag this asset to the **Material** property of the **Capsule Collider**:

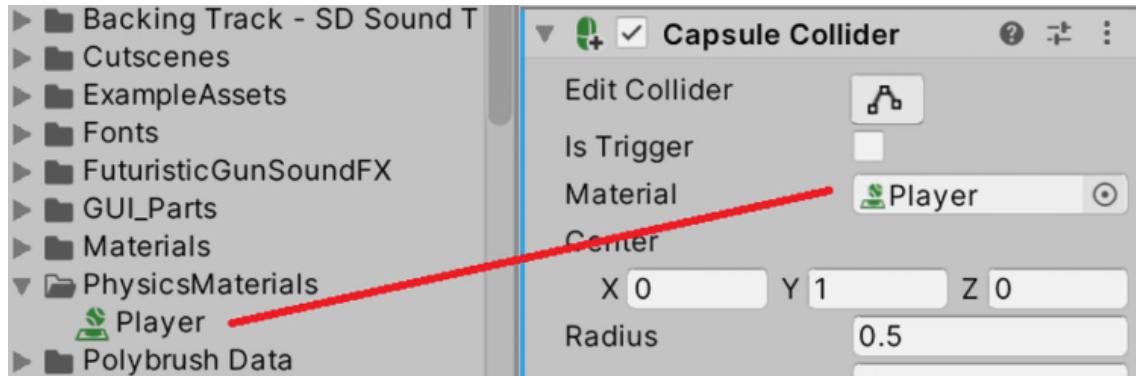


Figure 7.30: Setting the physics material of the player

9. If you play the game now, you may notice that the player will move faster than before because now we don't have any kind of friction on the floor, so you may need to reduce the movement force.

As you can see, we needed to bend the physics rules to allow a responsive player movement. You can get more responsiveness by increasing drags and forces, so the speeds are applied faster and reduced faster, but that depends, again, on the experience you want your game to have.

Tip

Some games want an immediate response with no velocity interpolation, going from 0 to full speed and vice versa from one frame to the other, and in these cases, you can override the velocity and rotation vectors of the player directly at your will or even use other systems instead of physics, such as the Character Controller component, which have special physics for platformer characters. You can read more about of them here:

<https://docs.unity3d.com/Manual/CharacterControllers.html>

Summary

Every game has physics in some way or another, for movement, collision detection, or both. In this chapter, we learned how to use the physics system for both, being aware of proper settings to make the system work properly, reacting to collisions to generate gameplay systems, and moving the player in such a way that it collides with obstacles, keeping its physically inaccurate movement. We used these concepts to create our player and bullet movement and make our bullets damage the enemies, but we can reuse the knowledge to create a myriad of other possible gameplay requirements, so I suggest you play a little bit with the physics concepts seen here; you can discover a lot of interesting use cases. In the next chapter, we will be discussing how to program the visual aspects of the game, such as effects, and make the UI react to the input.

8 Victory or Defeat: Win and Lose Conditions

Join our book community on Discord

<https://packt.link/unitydev>



Now that we have a basic gameplay experience, it's time to make the game end with the outcomes of winning or losing. One common way to implement this is through separated components with the responsibility of overseeing a set of objects to detect certain situations that need to happen, such as the player's life becoming 0 or all of the waves being cleared. We will implement this through the concept of **managers**, components that will manage and monitor several objects. In this chapter, we will examine the following manager concepts:

- Creating object managers
- Creating game modes
- Improving our code with events

With this knowledge, you will be able to not only create the victory and lose conditions of the game, but also do this in a properly structured way using design patterns such as **Singleton** and **Event Listeners**. These skills are not only useful for creating the winning and losing code of the game but any code in general. First, let's begin by creating managers to represent concepts such as score or game rules.

Creating object managers

Not every object in your Scene should be something that can be seen, heard, or collided with. Some objects can also exist with a conceptual meaning, not something tangible. For example, imagine you need to keep a count of the number of enemies: where do you save that? You also need someplace to save the current score of the player, and you may be thinking it could be on the player itself, but what happens if the player dies and respawns? The data would be lost! In such scenarios, the concept of a **manager** can be a useful way of solving this in our first games, so let's explore it. In this section, we are going to see the following object manager concepts:

- Sharing variables with the Singleton design pattern
- Sharing variables in Visual Scripting
- Creating managers

We will start by discussing what the Singleton design pattern is and how it helps us simplify the communication of objects. With it, we will create manager objects that allow us to centralize information about a group of objects, among other things. Let's start by discussing the Singleton design pattern.

Sharing variables with the Singleton design pattern

Design patterns are usually described as common solutions to common problems. There are several coding design decisions you will have to make while you code your game, but luckily, the ways to tackle the most common situations are well known and documented. In this section, we are going to discuss one of the most common design patterns, the **Singleton**, a convenient one to implement in simple projects. A Singleton pattern is used when we need a single instance of an object, meaning that there shouldn't be more than one instance of a class and that we want it to be easily accessible (not necessarily, but useful in our scenario). We have

plenty of cases in our game where this can be applied, for example, `ScoreManager`, a component that will hold the current score. In this case, we will never have more than one score, so we can take advantage of the benefits of the Singleton manager here. One benefit is being sure that we won't have duplicated scores, which makes our code less error prone. Also, so far, we have needed to create public references and drag objects via the editor to connect two objects, or look for them using `GetComponent`; with this pattern, however, we will have global access to our Singleton component, meaning you can just write the name of the component in your script and you will access it. In the end, there's just one `ScoreManager` component, so specifying which one via the editor is redundant. This is similar to `Time.deltaTime`, the class responsible for managing time—we have just one time. If you are an advanced programmer, you may be thinking about code testing and dependency injection now, and you are right, but remember, we are trying to write simple code so far, so we will stick to this simple solution. Let's create a **Score Manager** object, responsible for handling the score, to show an example of a Singleton by doing the following:

- Create an empty GameObject (**GameObject | Create Empty**) and call it `ScoreManager`; usually, managers are put in empty objects, separated from the rest of the scene's objects.
- Add a script called **ScoreManager** to this object with an **int** field called **amount** that will hold the current score.
- Add a field of the **ScoreManager** type called **instance**, but add the **static** keyword to it; this will make the variable global, meaning it can be accessed anywhere by just writing its name:

```
public class ScoreManager : MonoBehaviour
{
    public static ScoreManager instance;

    public int amount;
}
```

Figure 8.1: A static field that can be accessed anywhere in the code

- In **Awake**, check whether the **instance** field is not `null`, and in that case, set this **ScoreManager** instance as the instance reference using the **this** reference.
- In the `else` clause of the `null` checking `if` statement, print a message indicating that there's a second **ScoreManager** instance that must be destroyed:

```
void Awake()
{
    if (instance == null)
    {
        instance = this;
    }
    else
    {
        print("Duplicated ScoreManager, ignoring this one");
    }
}
```

Figure 8.2: Checking whether there's only one Singleton instance

The idea is to save the reference to the only **ScoreManager** instance in the instance static field, but if by mistake the user creates two objects with the **ScoreManager** component, this `if` statement will detect it and inform the user of the error, asking them to take action. In this scenario, the first **ScoreManager** instance to execute **Awake** will find that there's no instance set (the field is `null`) so it will set itself as the current instance, while the second **ScoreManager** instance will find the instance is already set and will print the message. Remember that `instance` is a static field, shared between all classes, unlike regular reference fields, where each component will have its own reference, so in this case, we have two `ScoreManager` instances added to the scene, and they will share the same instance field. To improve the example a little bit, it would be ideal to have a simple way to find the second `ScoreManager` in the game. It will be hidden somewhere in the Hierarchy and it may be difficult to find, but we fix this by doing the following:

- Replace `print` with `Debug.Log`. `Debug.Log` is similar to `print` but has a second argument that expects an object to be highlighted when the message is clicked in the console. In this case, we will pass the `gameObject` reference to allow the console to highlight the duplicated object:

```
Debug.Log("Duplicated ScoreManager, ignoring this one", gameObject);
```

Figure 8.3: Printing messages in the console with Debug.Log

After clicking the log message, the `GameObject` containing the duplicated `ScoreManager` will be highlighted in the Hierarchy:

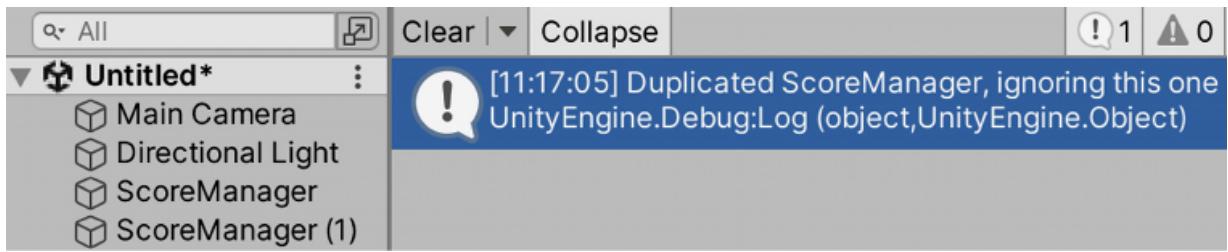


Figure 8.4: The highlighted object after clicking the message

- Finally, a little improvement can be made here by replacing `Debug.Log` with `Debug.LogError`, which will also print the message but with an error icon. In a real game, you will have lots of messages in the console, and highlighting the errors over the information messages will help us to identify them quickly:

Debug.LogError("Duplicated

Figure 8.5: Using LogError to print an error message

- Try the code and observe the error message in the console:

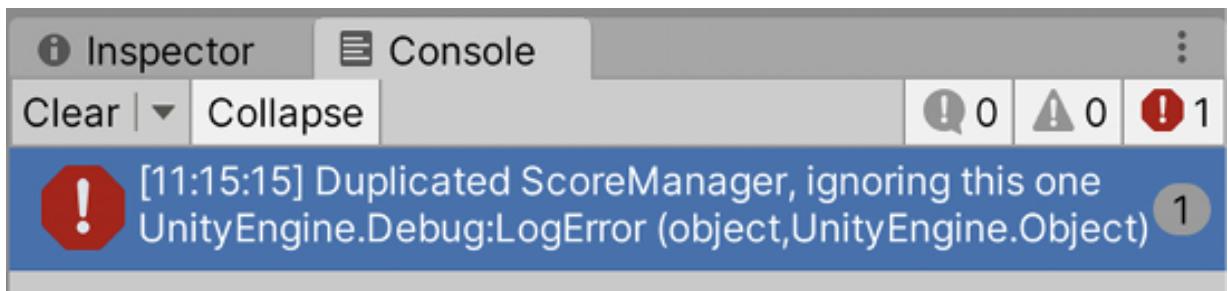


Figure 8.6: An error message in the console

The next step would be to use this Singleton somewhere, so in this case, we will make the enemies give points when they are killed by doing the following:

- Add a script to the `Enemy` Prefab called `ScoreOnDeath` with an `int` field called `amount`, which will indicate the number of

points the enemy will give when killed. Remember to set the value to something other than `0` in the editor for the Prefab.

- Create the `OnDestroy` event function, which will be automatically called by Unity when this object is destroyed, in our case, the enemy:

```
void OnDestroy()  
{  
}  
}
```

Figure 8.7: The `OnDestroy` event function

Consider that the `OnDestroy` function is also called when we change scenes or the game is quitting, so in this scenario, we might get points when changing scenes, which is not correct. So far, this is not a problem in our case, but later in this chapter, we will see a way to prevent this.

- Access the Singleton reference in the `OnDestroy` function by writing `ScoreManager.instance`, and add the `amount` field of our script to the `amount` field of the Singleton to increase the score when an enemy is killed:

```
public class ScoreOnDeath : MonoBehaviour
{
    public int amount;

    void OnDestroy()
    {
        ScoreManager.instance.amount += amount;
    }
}
```

Figure 8.8: Full ScoreOnDeath component class contents

- Select the `ScoreManager` in the Hierarchy, hit **Play**, and kill some enemies to see the score rise with every kill. Remember to set the `amount` field of the `ScoreOnDeath` component of the Prefab.

As you can see, the Singleton simplified a lot the way to access `ScoreManager` and have security measures to prevent duplicates of itself, which will help us to reduce errors in our code. Something to take into account is that now you will be tempted to just make everything a Singleton, such as the player's life or player's bullets and just to make your life easier when creating gameplay mechanics such as power-ups. While that will totally work, remember that your game will change, and I mean change a lot; any real project will experience constant change. Maybe today, the game has just one player, but maybe in the future, you want to add a second player or an AI companion, and you want the power-ups to affect them too, so if you abuse the Singleton pattern, you will have trouble handling those scenarios and many more. Maybe a future player companion will try to get the health pickup but the main player will be healed instead! The point here is to try to use the pattern as few times as

you can, in case you don't have any other way to solve the problem. To be honest, there are always ways to solve problems without Singleton, but they are a little bit more difficult to implement for beginners, so I prefer to simplify your life a little bit to keep you motivated. With enough practice, you will reach a point where you will be ready to improve your coding standards.

There are lots of design patterns out there to help you design your game. Once you get comfortable with Unity scripting, we recommend reading the following Unity's official game programming patterns book:

<https://resources.unity.com/games/level-up-your-code-with-game-programming-patterns>. This book also includes an advanced implementation of singleton.

Now, let's discuss how to achieve this in Visual Scripting, which deserves its own section given that it will be a little bit different. You may consider skipping the following section if you are not interested in the Visual Scripting side of these scripts.

Sharing variables with Visual Scripting

Visual Scripting has a mechanism that replaces Singleton as a holder of variables to be shared between objects: the **scene variables**. If you check the left panel in the **Script Graph** editor (the window where we edit the nodes of a script) under the **Blackboard** panel (the panel that shows the variables of our object), you will notice it will have many tabs: **Graph**, **Object**, **Scene**, **App** and **Saved**. If you don't see **Blackboard** panel, click the third button from left to right at the top-left part of the window, the button at the right of the **i** (information) button:

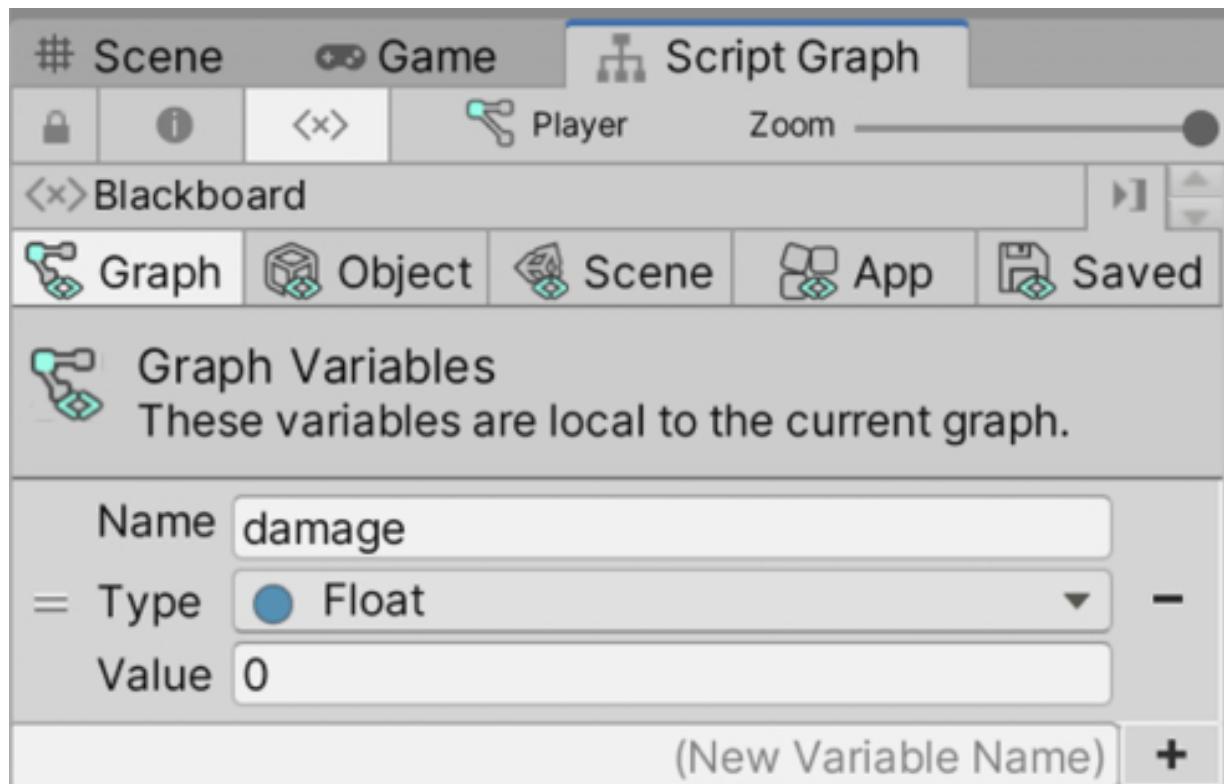


Figure 8.9: Blackboard (variables) editor in Script Graph

So far, when we created a variable in the **Variables** component of any object, we were actually creating **Object Variables**: variables that belongs to an object and are shared between all Visual Scripts in that one, but that's not the only scope a variable can have. Here's a list of the remaining scopes:

- **Graph**: Variables that can only be accessed by our current graph. No other script can read or write that variable. This is useful to save internal state, like private variables in C#.
- **Scene**: Variables that can be accessed by all objects in the current scene. When we change the scene, those variables are lost.
- **App**: Variables that can be accessed in any part of the game at any time. This is useful to move values from one scene to the other. For example, you can increase the score in one level and keep increasing it in the next, instead of restarting the score from 0.

- **Saved:** Variables whose values are kept between game runs. You can save persistent data such as the **Player Level** or **Inventory** to continue the quest, or simpler things such as the sound volume as set by the user in the **Options** menu (if you created one).

In this case, the **Scene** scope is the one we want, as the score we intend to increase will be accessed by several objects in the scene (more on that later) and we don't want it to persist if we reset the level to play again; it will need to be set again to 0 in each run of the level and game. To create scene variables, you can simply select the **Scene** tab in the **Blackboard** pane of the **Script Graph** editor, while you are editing any **Script Graph**, or you can also use the **Scene Variables** GameObject that was created automatically when you started editing any graph. That object is the one that really holds the variables and must not be deleted. You will notice that it will have a **Variables** component as we have used before, but it will also have the **Scene Variables** component, indicating those variables are scene variables. In the following screenshot, you can see how we have simply added the **Score** variable to the **Scene Variables** tab to make it accessible in any of our Script Graphs.

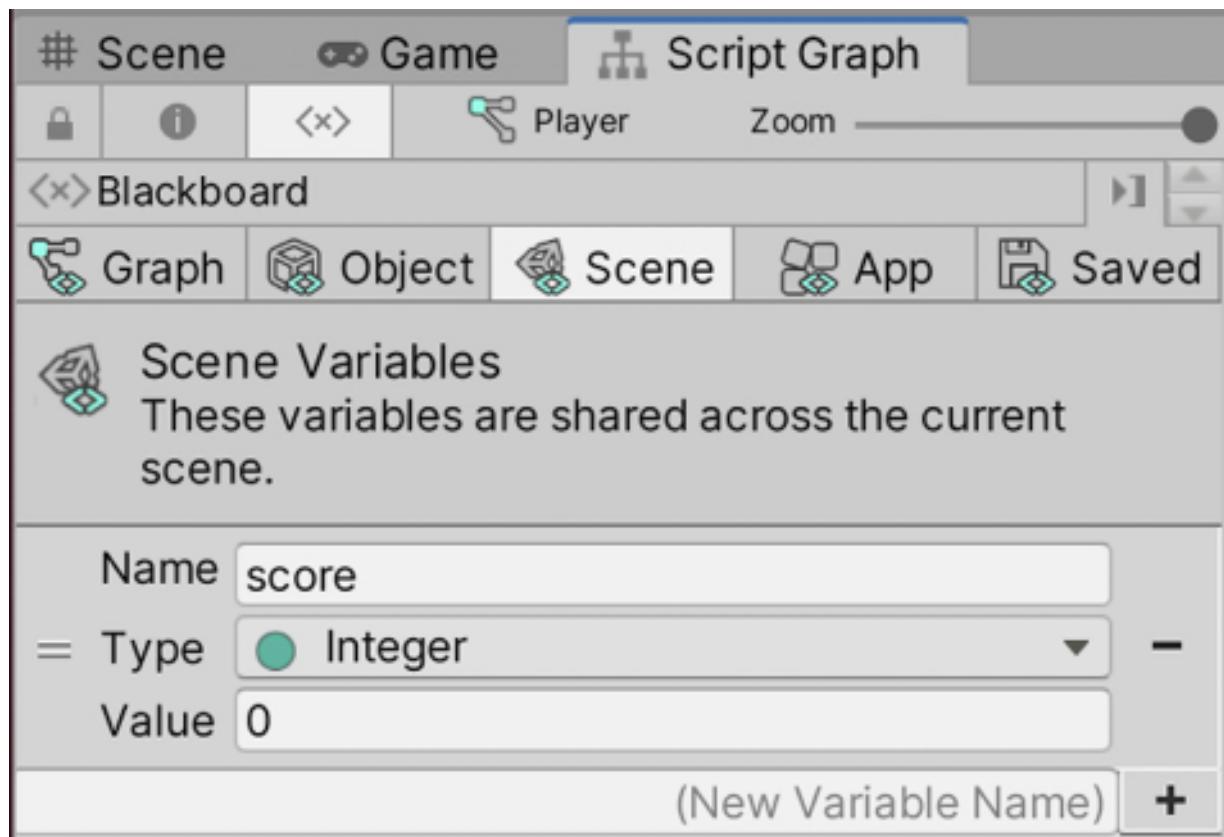


Figure 8.10: Adding scene variables to our game

Finally, for the score-increasing behavior, we can add the following graph to our enemy. Remember, as usual, to have the C# or the Visual Scripting version of the scripts, not both.

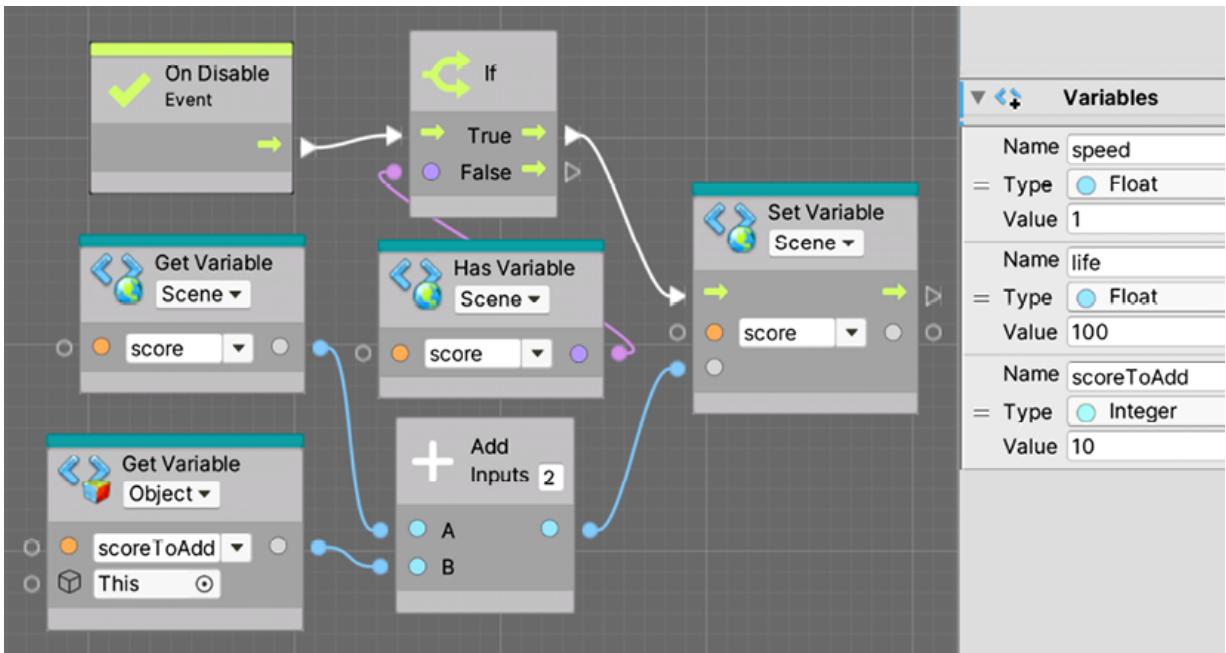


Figure 8.11: Adding score when this object is destroyed

At first, this script seems pretty similar to our C# version; we add our `coreToAdd` variable of our object (**Object** scope) and then we add it to the whole scene's `score` variable, as specified in the node. The main difference you can see is that here we are using the **OnDisable** event instead of **OnDestroy**. Actually, **OnDestroy** is the correct one, but in the current version of Visual Scripting there is a bug that prevents it from working properly, so I replaced it for now. The problem with **OnDisable** is that it executes whenever the object is disabled, and while the object is disabled before it is destroyed, it can also be disabled in other circumstances (for example, using **Object Pooling**, a way to recycle objects instead of destroying and instancing them constantly), but so far it is enough for us. Please consider trying first with **OnDestroy** when you try this graph to see if it runs properly in your Unity or Visual Scripting package version. Something to highlight is the usage of the **Has Variable** node to check if the **score variable** exists. This is done because **OnDisable** can be executed either at the moment of the enemy being destroyed, or when the scene changes, which we will do later this chapter with the lose/win screens. If we try to get a

scene variable at that moment, we risk getting an error if the **Scene Variables** object is destroyed before the **GameMode** object, given the change of scene involves destroying every object in the scene first. As you may have noticed by now, even if Visual Scripting is mostly extremely similar to C#, one has concepts to solve certain scenarios that the other doesn't. Now that we know how to share variables, let's finish some other managers that we will need later in the game.

Creating managers

Sometimes, we need a place to put together information about a group of similar objects, for example, `EnemyManager`, to check the number of enemies and potentially access an array of them to iterate over them and do any extra gameplay implementation we want to, for instance make our `MissionManager`, to have access to all of the active missions in our game. Again, these cases can be considered Singletons, single objects that won't be repeated (in our current game design), so let's create the ones we will need in our game, that is, `EnemyManager` and `WaveManager`. In our game, `EnemyManager` and `WaveManager` will just be used as places to save an array of references to the existing enemies and waves in our game, just as a way to know their current amount. There are ways to search all objects of a certain type to calculate their count, but those functions are expensive and not recommended for use unless you really know what you are doing. So, having a Singleton with a separate updated list of references to the target object type will require more code but will perform better. Also, as the game features increase, these managers will have more functionality and helper functions to interact with those objects. Let's start with the enemies manager by doing the following:

- Add a script called `Enemy` to the **Enemy** Prefab; this will be the script that will connect this object with `EnemyManager` in a moment.

- Create an empty `GameObject` called `EnemyManager` and add a script to it called `EnemiesManager`.
- Create a `public static` field of the `EnemiesManager` type called `instance` inside the script and add the Singleton repetition check in `Awake` as we did in `ScoreManager`.
- Create a public field of the `List<Enemy>` type called `enemies`:

```
public List<Enemy> enemies;
```

Figure 8.12: List of Enemy components

A list in C# represents a dynamic array, an array capable of adding and removing objects. You will see that you can add and remove elements to this list in the editor, but keep the list empty; we will add enemies another way. Take into account that `List` is in the `System.Collections.Generic` namespace; you will find the `using` sentence at the beginning of our script. Also, consider that you can make the list private and expose it to the code via a getter instead of making it a public field; but as usual, we will make our code as simple as possible for now:

```
using System;  
using System.Collections.Generic;  
using UnityEngine;
```

Figure 8.13: Using needed to use the List class, inside System.Collections.Generic

Consider that `List` is a class type, so it must be instantiated, but as this type has exposing support in the editor, Unity will automatically instantiate it. You must use the **new keyword** to

instantiate it in cases where you want a non-editor-exposed list, such as a private one or a list in a regular non-component C# class. The C# list internally is implemented as an array. If you need a linked list, use the **LinkedList** collection type instead.

- In the `Start` function of the `Enemy` script, access the `EnemyManager` Singleton and using the `Add` function of the enemies list, add this object to the list. This will "register" this enemy as active in the manager, so other objects can access the manager and check for the current enemies. The `Start` function is called after all of the `Awake` function calls, and this is important because we need to be sure that the `Awake` function of the manager is executed prior to the `Start` function of the enemy to ensure that there is a manager set as the instance.

The problem we solved with the `Start` function is called a race condition, that is, when two pieces of code are not guaranteed to be executed in the same order, whereas `Awake` execution order can change due to different reasons. There are plenty of situations in code that this will happen, so pay attention to the possible race conditions in your code. Also, you might consider using more advanced solutions such as `lazy initialization` here, which can give you better stability, but again, for the sake of simplicity and exploring the Unity API, we will use the `Start` function approach for now.

- In the `OnDestroy` function, remove the enemy from the list to keep the list updated with just the active ones:

```
public class Enemy : MonoBehaviour
{
    void Start()
    {
        EnemyManager.instance.enemies.Add(this);
    }

    void OnDestroy()
    {
        EnemyManager.instance.enemies.Remove(this);
    }
}
```

Figure 8.14: The enemy script to register ourselves as an active enemy

With this, now we have a centralized place to access all of the active enemies in a simple but efficient way. I challenge you to do the same with the waves, using **WaveManager**, which will have the collection of all active waves to later check whether all waves finished their work to consider the game as won. Take some time to solve this; you will find the solution in the following screenshots, starting with **WavesManager**:

```
using System.Collections.Generic;
using UnityEngine;

public class WavesManager : MonoBehaviour
{
    public static WavesManager instance;

    public List<WaveSpawner> waves;

    private void Awake()
    {
        if (instance == null)
            instance = this;
        else
            Debug.LogError("Duplicated WavesManager", gameObject);
    }
}
```

Figure 8.15: The full WavesManager script

You will also need the `WaveSpawner` script:

```

public class WaveSpawner : MonoBehaviour
{
    public GameObject prefab;
    public float startTime;
    public float endTime;
    public float spawnRate;

    private void Start()
    {
        WavesManager.instance.waves.Add(this);
        InvokeRepeating("Spawn", startTime, spawnRate);
        Invoke("EndSpawner", endTime);
    }

    void Spawn()
    {
        Instantiate(prefab, transform.position, transform.rotation);
    }

    void EndSpawner()
    {
        WavesManager.instance.waves.Remove(this);
        CancelInvoke();
    }
}

```

Figure 8.16: The modified WaveSpawner script to support WavesManager

As you can see, `WaveManager` is created the same way `EnemyManager` was, just a Singleton with a list of `WaveSpawner` references, but `WaveSpawner` is different. We execute the `Add` function of the list in the `Start` event of `WaveSpawner` to register the wave as an active one, but the `Remove` function needs more work. The idea is to deregister the wave from the active waves list when the spawner

finishes its work. Before this modification, we used `Invoke` to call the `CancelInvoke` function after a while to stop the spawning, but now we need to do more after the end time. Instead of calling `CancelInvoke` after the specified wave end time, we will call a custom function called `EndSpawner`, which will call `CancelInvoke` to stop the spawner, `Invoke Repeating`, but also will call the `remove-from-WavesManager`-list function to make sure the removing-from-the-list function is called exactly when `WaveSpawner` finishes its work. Regarding the Visual Scripting version, we can add two lists of `GameObject` type to the scene variables to hold the references to the existing waves and enemies so we can keep track of them. Just search "List of `GameObject`" in the search bar of the variable type selector and you will find it. In this case, the lists contain only `GameObjects` given that the Visual Scripting versions of **WaveSpawner** and enemy scripts are not types we can reference like C# ones. If you did both C# and Visual Scripting versions of these you will see you can reference the C# versions, but we are not going to mix C# and Visual Scripting as it is out of the scope of the book, so ignore them. Anyway, given how the **Variables** system of Visual Scripting works, we can still access variables inside if needed using the **GetVariable** node—remember the variables are not in the Visual Scripts but in the **Variables** node:

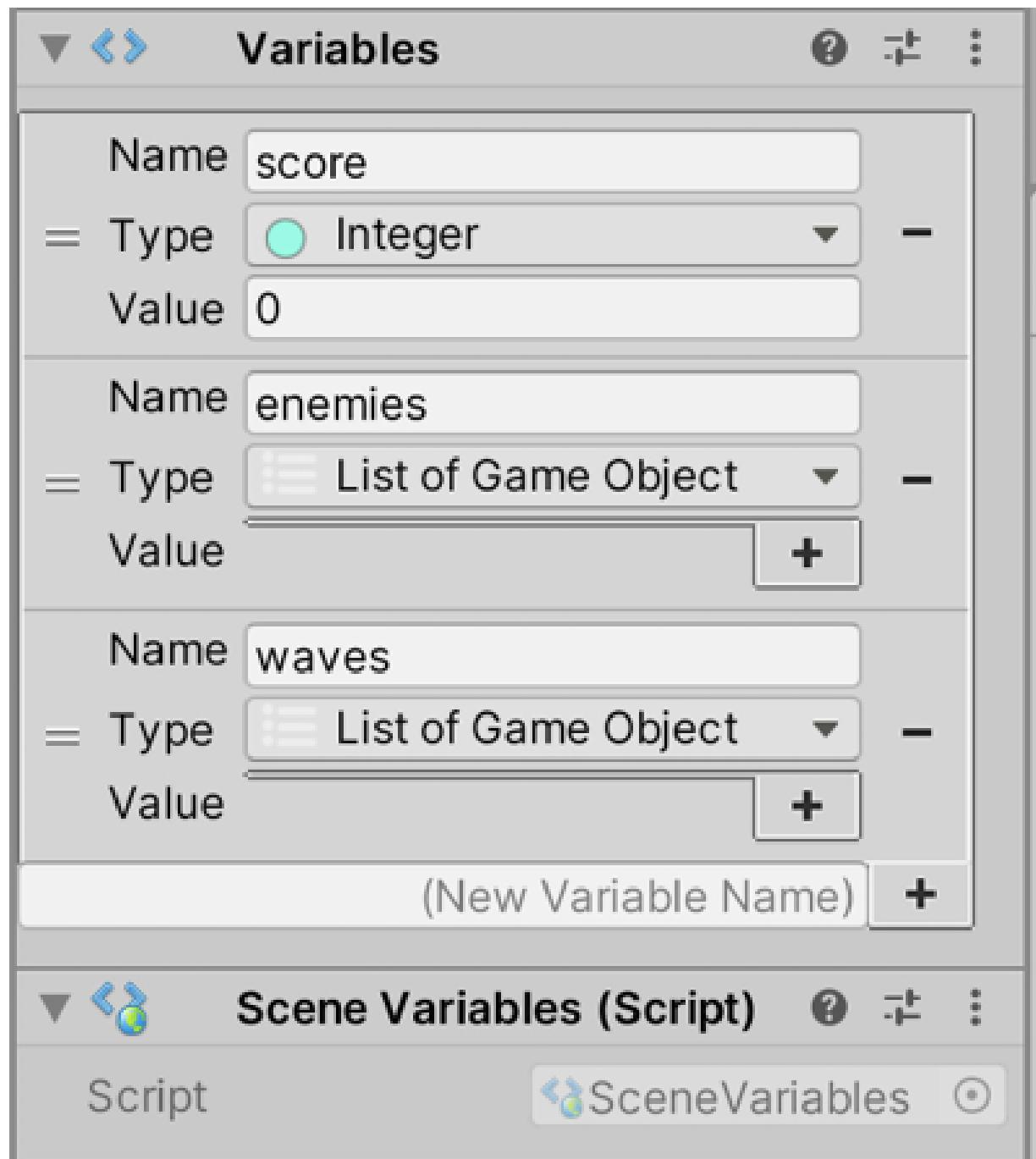


Figure 8.17: Adding lists to the Scene variables

Then, we can add the following to the **WaveSpawner** graph:

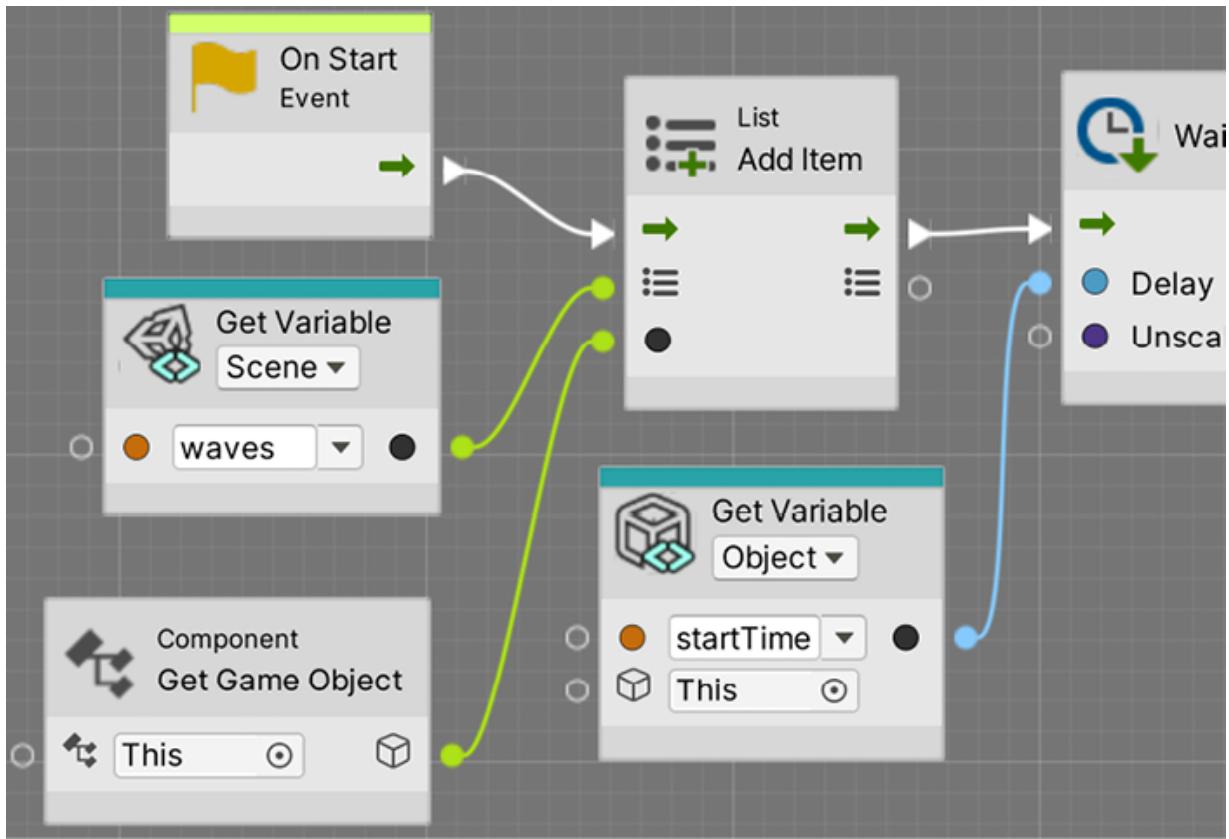


Figure 8.18: Adding elements to List

We used the **Add List Item** node to add our `GameObject` to the **waves** variable. We did this as the first thing to do in the **On Start** event node before anything. And to remove that wave from the active ones you will need to make the following change:

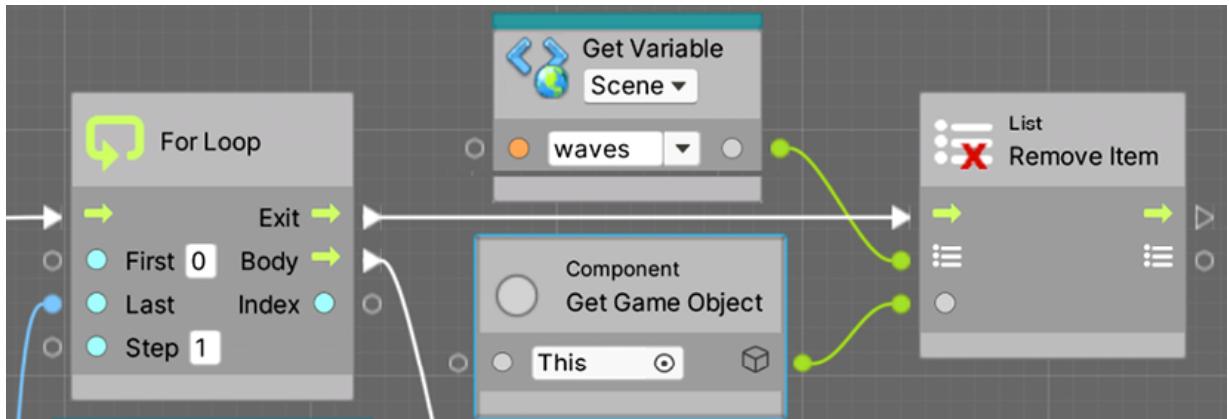


Figure 8.19: Removing elements from the List

We remove this spawner from the list using the **Exit** flow output pin of the **For Loop**, which is executed when the `for` loop finishes iterating. Finally, regarding **Enemy**, you will need to create a new **Enemy Script** graph that will look similar:

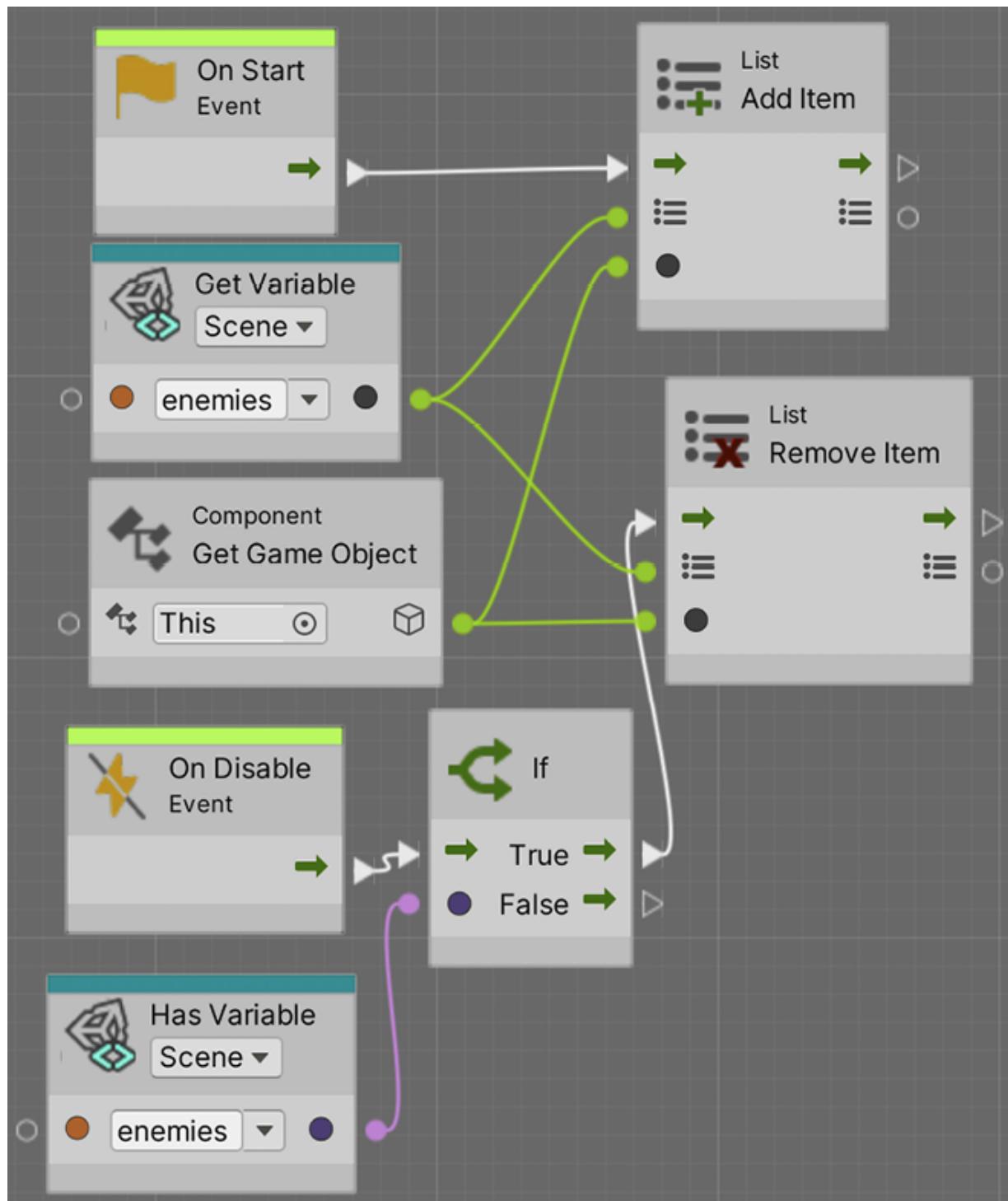


Figure 8.20: Enemy Adding and removing itself from the Lists

As you can see, we simply add the enemy on **OnStart** and remove it in **OnDisable**. Remember to try first using **OnDestroy** instead of

OnDisable due to the bug we mentioned previously. You can check these changes by playing the game while having the **Scene Variables** GameObject selected and seeing how its value changes. Also remember the need to use the **Has Variable** node in case we are changing scenes. Using Object managers, we now have centralized information about a group of objects, and we can add all sorts of object group logic here. We created the `EnemiesManager`, `WavesManager`, and `ScoreManager` as centralized places to store several game systems' information, such as the enemies and waves present in the scene, and the score as well. We also saw the Visual Scripting version, centralizing that data in the Scene Variables object, so all Visual Scripts can read that data. But aside from having this information for updating the UI (which we will do in the next chapter), we can use this information to detect whether the victory and lose conditions of our game are met, creating a **Game Mode** object to detect that.

Creating Game Modes

We have created objects to simulate lots of gameplay aspects of our game, but the game needs to end sometime, whether we win or lose. As always, the question is where to put this logic, and that leads us to further questions. The main questions would be, will we always win or lose the game the same way? Will we have a special level with different criteria than "kill all of the waves," such as a timed survival? Only you know the answer to those questions, but if right now the answer is no, it doesn't mean that it won't change later, so it is advisable to prepare our code to adapt seamlessly to changes.

To be honest, preparing code to adapt seamlessly to changes is almost impossible; there's no way to have code that considers every possible case, and we will always need to rewrite some code sooner or later. We will try to make the code as generic as possible to adapt to changes, but we need to find a balance between necessary and unnecessary adaptability. Creating generic code tends to generate complex codebases and takes

more time, and while a certain degree of complexity is certainly necessary, I saw many times programmers go beyond what's needed, taking a huge amount of time to solve simple cases, creating tools that ended up being under-utilized.

To do this, we will separate the Victory and Lose conditions' logic in its own object, which I like to call the "Game Mode" (not necessarily an industry standard). This will be a component that will oversee the game, checking conditions that need to be met in order to consider the game over. It will be like the referee of our game. The Game Mode will constantly check the information in the object managers and maybe other sources of information to detect the needed conditions. Having this object separated from other objects allows us to create different levels with different Game Modes; just use another Game Mode script in that level and that's all. In our case, we will have a single Game Mode for now, which will check whether the number of waves and enemies becomes 0, meaning that we have killed all of the possible enemies and the game is won. Also, it will check whether the life of the player reaches 0, considering the game as lost in that situation. Let's create it by doing the following:

- Create an empty `GameMode` object and add a `WavesGameMode` script to it. As you can see, we gave the script a descriptive name considering that we can add other Game Modes.
- In its `Update` function, check whether the number of enemies and waves has reached 0 by using the `Enemy` and `Wave` managers; in that case, just `print` a message in the console for now. All lists have a `count` property, which will tell you the number of elements stored inside.
- Add a `public` field of the `Life` type called `PlayerLife` and drag the player to that one; the idea is to also detect the lose condition here.
- In `Update`, add another check to detect whether the life amount of the `playerLife` reference reached 0, and in that case, `print`

a lose message in the console:

```
[SerializeField] Life playerLife;

void Update()
{
    if (EnemyManager.instance.enemies.Count <= 0 &&
        WavesManager.instance.waves.Count <= 0)
    {
        SceneManager.LoadScene("You win!");
    }

    if (playerLife.amount <= 0)
    {
        SceneManager.LoadScene("You lose!");
    }
}
```

Figure 8.21: Win and lose condition checks in WavesGameMode

- Play the game and test both cases, whether the player life reaches 0 or whether you have killed all enemies and waves.

Now, it is time to replace the messages with something more interesting. For now, we will just change the current scene to a **Win Scene** or **Lose Scene**, which will only have a UI with a win or lose message and a button to play again. In the future, you can add a Main Menu scene and have an option to get back to it. Let's implement this by doing the following:

- Create a new scene (**File | New Scene**) and save it, calling it `WinScreen`.
- Add something to indicate that this is the win screen, such as simply a sphere with the camera pointing to it. This way we know when we changed to the win screen.
- Select the scene in the **Project View** and press **Ctrl + D** (**Cmd + D** on Mac) to duplicate the scene. Rename it `LoseScreen`.
- Double-click the `LoseScreen` scene to open it and change the sphere to something different, maybe a cube.
- Go to **File | Build Settings** to open the **Scenes in Build** list inside this window.

The idea is that Unity needs you to explicitly declare all scenes that must be included in the game. You might have test scenes or scenes that you don't want to release yet, so that's why we need to do this. In our case, our game will have `WinScreen`, `LoseScreen`, and the scene we have created so far with the game scenario, which I called `Game`, so just drag those scenes from the **Project View** to the list of the **Build Settings** window; we will need this to make the Game Mode script change between scenes properly. Also, consider that the first scene in this list will be the first scene to be opened when we play the game in its final version (known as the build), so you may want to rearrange the list according to that:



Figure 8.22: Registering the scenes to be included in the build of the game

- In `WavesGameMode`, add a `using` statement for the `UnityEngine.SceneManagement` namespace to enable the scene

changing functions in this script.

- Replace the console `print` messages with calls to the `SceneManager.LoadScene` function, which will receive a string with the name of the scene to load; in this case, it would be `WinScreen` and `LoseScreen`. You just need the scene name, not the entire path to the file.

If you want to chain different levels, you can create a `public` `string` field to allow you to specify via editor which scenes to load.

Remember to have the scenes added to the **Build Settings**, if not, you will receive an error message in the console when you try to change the scenes:

```

using UnityEngine;
using UnityEngine.SceneManagement;

public class WavesGameMode : MonoBehaviour
{
    [SerializeField] Life playerLife;

    void Update()
    {
        if (EnemyManager.instance.enemies.Count <= 0 &&
            WavesManager.instance.waves.Count <= 0)
        {
            SceneManager.LoadScene("WinScreen");
        }

        if (playerLife.amount <= 0)
        {
            SceneManager.LoadScene("LoseScreen");
        }
    }
}

```

Figure 8.23: Changing scenes with SceneManager

- Play the game and check whether the scenes change properly.

Right now, we picked the simplest way to show whether we lost or won, but in the future, you may want something gentler than a sudden change of scene, such as maybe waiting a few moments with `Invoke` to delay that change or directly show the winning message

inside the game without changing scenes. Bear this in mind when testing the game with people and checking whether they understood what happened when they were playing—game feedback is important to keep the player aware of what is happening and is not an easy task to tackle. Regarding the Visual Scripting version, we added a new Script Graph to a separated object. Let's examine it piece by piece to see it clearly. Let's start with the win condition:

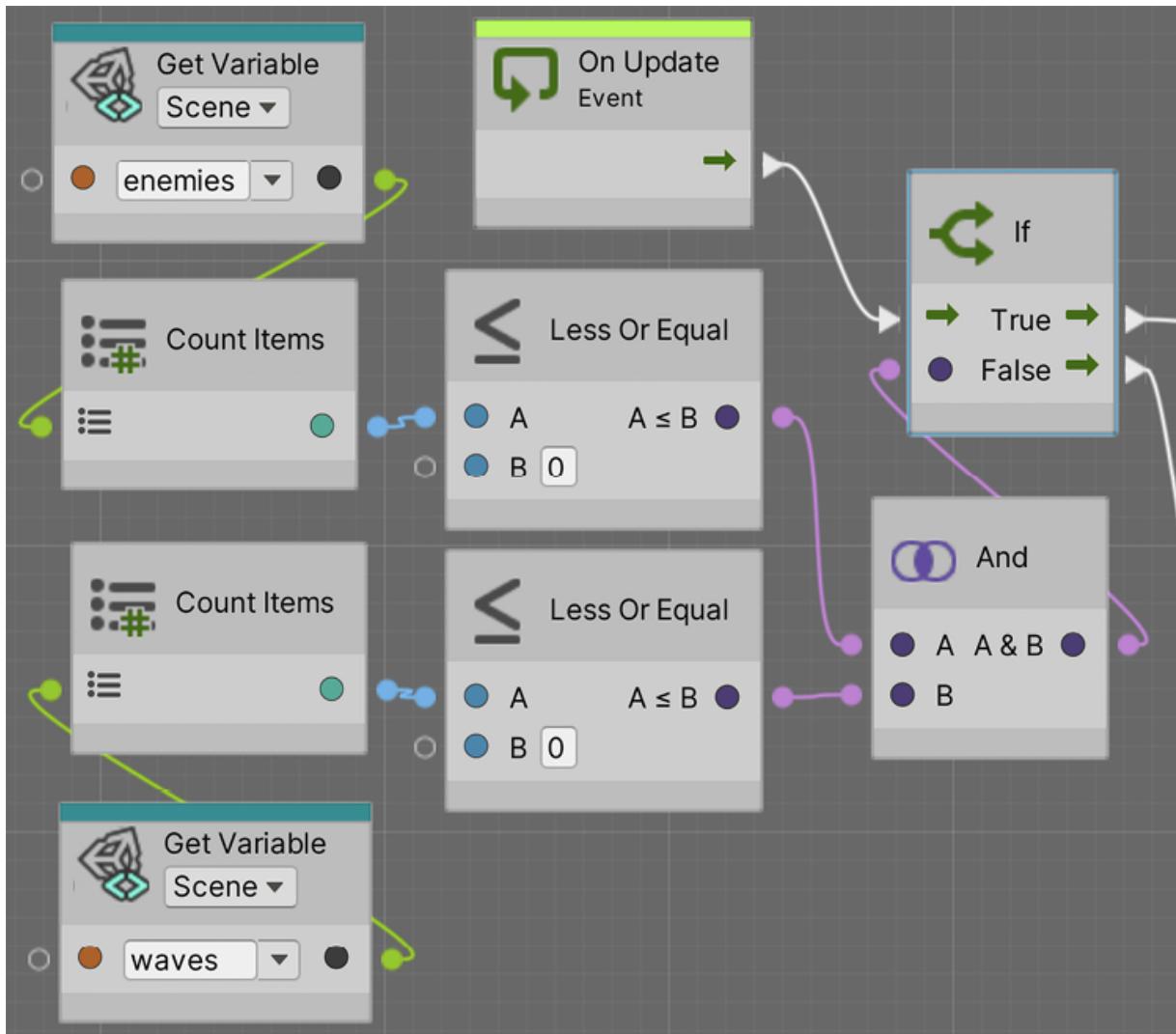


Figure 8.24: Win condition in Visual Scripting

Here, we are getting the **Enemies** list from the scene context (**GetVariable** node), and knowing that it contains a List, we are using the **Count Items** node to check how many enemies remain

in this list. Remember we have a script that adds the enemy to the list when it's spawned and removes it when it is destroyed. We do the same for the waves, so combine the conditions with an **And** node and connect it with an **If** to then do something (more on that in a moment). Now let's examine the Lose condition:

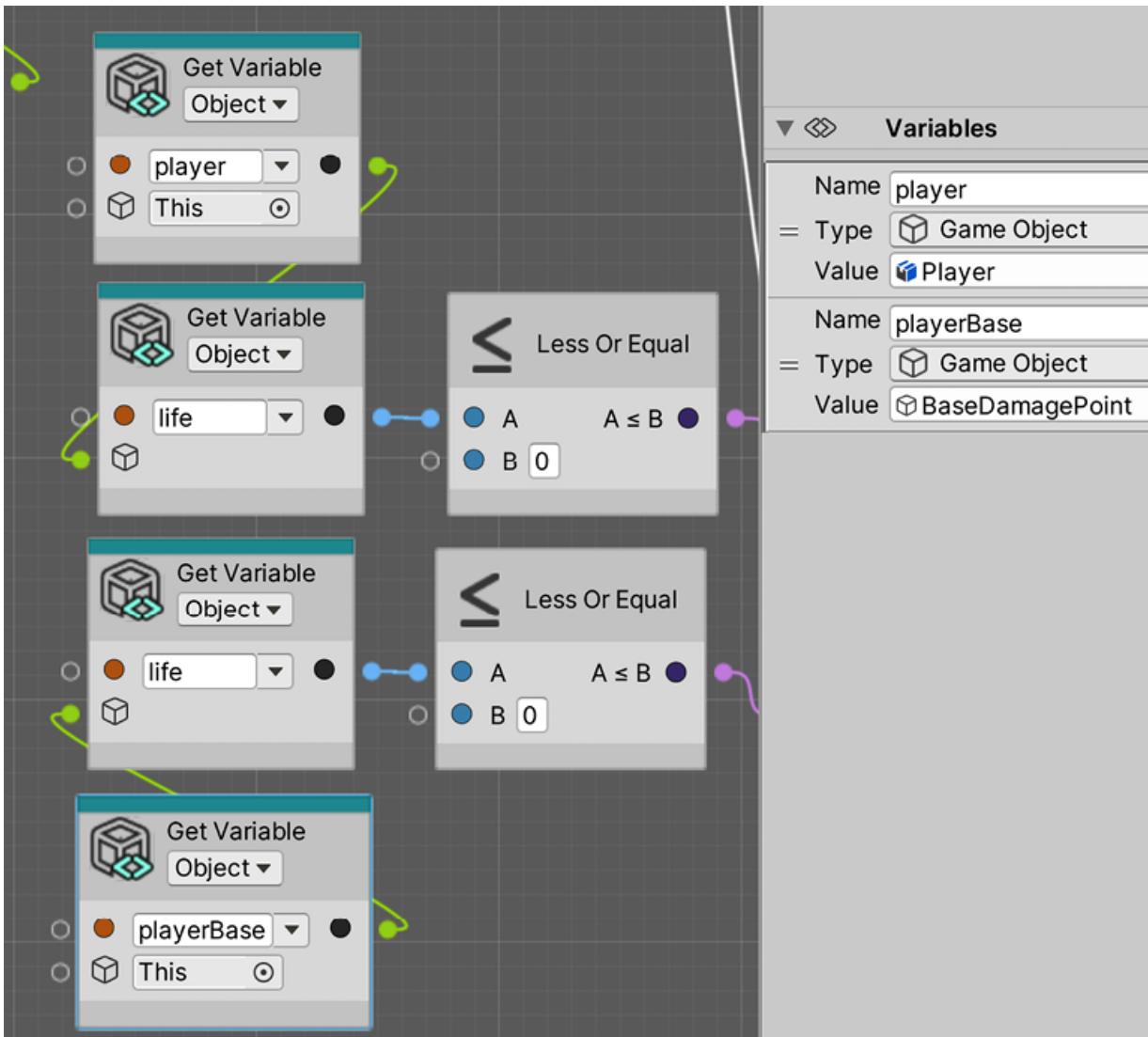


Figure 8.25: Lose condition in Visual Scripting

As the player's life is not in the scene context (and shouldn't be), and the player is a different GameObject from the one called `GameMode` (the one we created specifically for this script), we need a variable of type GameObject called **player** to reference it. As you can

see, we dragged our player to it in the **Variables** component. Finally, we used a **GetVariable** to access our player reference in the graph, and then another **GetVariable** to extract the life from it. We accomplished that by connecting the player reference to the **GetVariable** node of the life variable. Then we repeated this for the player's base. Finally, we load the scenes by doing the following:

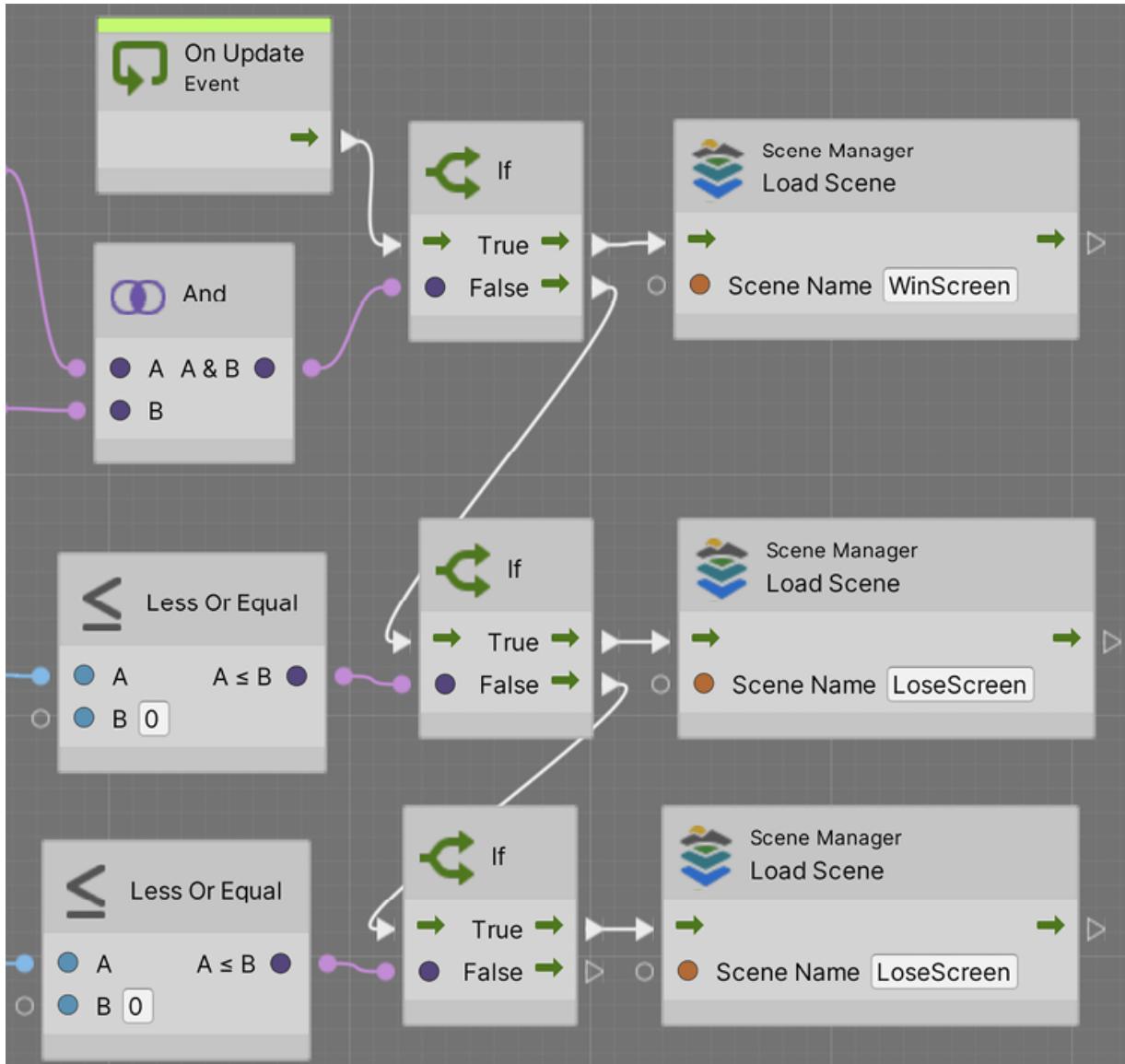


Figure 8.26: Loading scenes in Visual Scripting

As you can see, we use the **SceneManager LoadScene** (**SceneName**) node to load the scenes. Notice how we load both

scenes we created before (**Win Scene** and **Lose Scene**).

Remember that this two scenes need to be added to the Scenes in Build section inside the Build Settings like we did before, in order to be available to be leaded by the Scene Manager here. Now we have a fully functional simple game, with mechanics and win and lose conditions, and while this is enough to start developing other aspects of our game, I want to discuss some issues with our current manager approach and how to solve them with events.

Improving our code with events

So far, we used Unity event functions to detect situations that can happen in the game such as `Awake` and `Update`. There are other similar functions that Unity uses to allow components to communicate with each other, as in the case of `OnTriggerEnter`, which is a way for the Rigidbody to inform other components in the GameObject that a collision has happened. In our case, we are using `if` statements inside the `Update` method to detect changes on other components, such as `GameMode` checking whether the number of enemies has reached 0. But we can improve this if we are informed by the Enemy manager when something has changed, and just do the check at that moment, such as with the Rigidbody telling us when collisions occur instead of checking for collisions every frame. Also, sometimes, we rely on Unity events to execute logic, such as the score being given in the `OnDestroy` event, which informs us when the object is destroyed, but due to the nature of the event, it can be called in situations we don't want to add to the score, such as when the scene is changed, or the game is closed. Objects are destroyed in those cases, but not because the player killed the enemy, leading to the score increasing when it shouldn't. In this case, it would be great to have an event that tells us that life reached 0 to execute this logic, instead of relying on the general-purpose `OnDestroy` event. The idea of events is to improve the model of communication between our objects, with the assurance that at the exact moment something happens, the relevant parts in

that situation are notified to react accordingly. Unity has lots of events, but we can create ones specific to our gameplay logic. Let's start by applying this in the score scenario we discussed earlier; the idea is to make the `Life` component have an event to communicate to the other components that the object was destroyed because life reached 0. There are several ways to implement this, and we will use a little bit of a different approach than the **Awake** and **Update** methods; we will use the `UnityEvent` field type. This is a field type capable of holding references to functions to be executed when we want to, like C# delegates, but with other benefits, such as better Unity editor integration. To implement this, do the following:

- In the `Life` component, create a `public` field of the `UnityEvent` type called `onDeath`. This field will represent an event where other classes can subscribe to it to be made aware when `Life` reaches 0:

```
public class Life : MonoBehaviour
{
    public float amount;
    public UnityEvent onDeath;
```

Figure 8.27: Creating a custom event field

- If you save the script and go to the editor, you can see the event in the Inspector. Unity events support being subscribed to methods in the editor so we can connect two objects together. We will use this in the UI scripting chapter, so let's just ignore this for now:

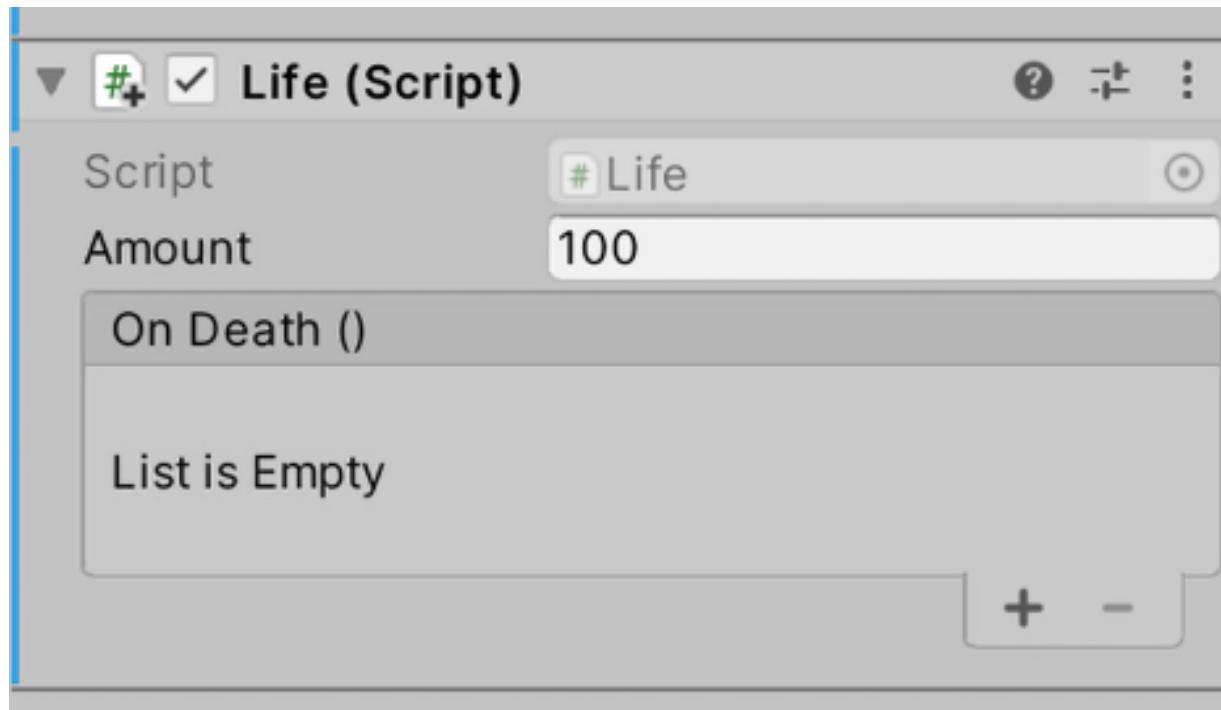


Figure 8.28: *UnityEvents showing up in the Inspector*

You can use the generic delegate action or a custom delegate to create events instead of using `UnityEvent`, and aside from certain performance aspects, the only noticeable difference is that `UnityEvent` will show up in the editor, as demonstrated in step 2.

- When life reaches `0`, call the `Invoke` function of the event. This way, we will be telling any script interested in the event that it has happened:

```

public float amount;
public UnityEvent onDeath;

void Update()
{
    if (amount <= 0)
    {
        onDeath.Invoke();
        Destroy(gameObject);
    }
}

```

Figure 8.29: Executing the event

- In `ScoreOnDeath`, rename the `OnDestroy` function to `GivePoints` or whatever name you prefer; the idea here is to stop giving points in the `OnDestroy` event.
- In the `Awake` function of the `ScoreOnDeath` script, get the `Life` component using `GetComponent` and save it in a local variable.
- Call the `AddListener` function of the `onDeath` field of the `Life` reference and pass the `GivePoints` function as the first

argument. This is known as **subscribing** our `listener` method `GivePoints` to the event `onDeath`. The idea is to tell `Life` to execute `GivePoints` when the `onDeath` event is invoked. This way, `Life` informs us about that situation. Remember that you don't need to call `GivePoints`, but just pass the function as a field:

```
private void Awake()
{
    var life = GetComponent<Life>();
    life.onDeath.AddListener(GivePoints);
}

void GivePoints()
{
    ScoreManager.instance.amount += amount;
}
```

Figure 8.30: Subscribing to the OnDeath event to give points in that scenario

Consider calling `RemoveListener` in `OnDestroy`; as usual, it is convenient to unsubscribe listeners when possible to prevent any memory leak (reference preventing the GC to deallocate memory). In this scenario, it is not entirely necessary because both the `Life` and `ScoreOnDeath` components will be destroyed at the same time, but try to get used to this as a good practice.

- Save, select `ScoreManager` in the editor, and hit **Play** to test this. Try deleting an enemy from the Hierarchy while in **Play** mode to check that the score doesn't rise because the enemy was destroyed for a reason other than their life becoming 0; you must destroy an enemy by shooting at them to see the score increase.

Now that `Life` has an `onDeath` event, we can also replace the player's `Life` check from the `WavesGameMode` to use the event by doing the following:

- Create an `OnPlayerDied` function on the `WavesGameMode` script and move the loading of the `LoseScreen` scene from `Update` to this function. You will be removing the `if` that checks the life from the `Update` method, given that the event version will replace it.
- In `Awake`, add this new function to the `onDeath` event of the player's `Life` component reference, called `playerLife` in our script:

```
void Awake()
{
    playerLife.onDeath.AddListener(OnPlayerDied);
}

void OnPlayerDied()
{
    SceneManager.LoadScene("LoseScreen");
}
```

Figure 8.31: Checking the lose condition with events

As you can see, creating custom events allows you to detect more specific situations other than the defaults in Unity, and keeps your code clean, without needing to constantly ask conditions in the `Update` function, which is not necessarily bad, but the event approach generates clearer code. Remember that we can lose our game also by the player's base `Life` reaching 0, so let's create a cube that represents the object that enemies will attack to reduce the base `Life`. Taking this into account, I challenge you to add this second lose condition (player's base life reaching 0) to our script. When you finish, you can check the solution in the following screenshot:

```
[SerializeField] private Life playerLife;
[SerializeField] private Life playerBaseLife;

void Awake()
{
    playerLife.onDeath.AddListener(OnPlayerOrBaseDied);
    playerBaseLife.onDeath.AddListener(OnPlayerOrBaseDied);
}

void OnPlayerOrBaseDied()
{
    SceneManager.LoadScene("LoseScreen");
}
```

Figure 8.32: Complete WavesGameMode lose condition

As you can see, we just repeated the `life` event subscription, remember to create an object to represent the player's base damage point, add a `Life` script to it, and drag that one as the player base `Life` reference of the Waves Game Mode. Something interesting here is that we subscribed the same function called

`OnPlayerOrBaseDied` to both `player Life` and `base Life onDeath` events, given that we want the same result in both situations. Now, let's keep illustrating this concept by applying it to the managers to prevent the Game Mode from checking conditions every frame:

- Add a `UnityEvent` field to `EnemyManager` called `onChanged`. This event will be executed whenever an enemy is added or removed from the list.
- Create two functions, `AddEnemy` and `RemoveEnemy`, both receiving a parameter of the `Enemy` type. The idea is that instead of `Enemy` adding and removing itself from the list directly, it should use these functions.
- Inside these two functions, invoke the `onChanged` event to inform others that the enemies list has been updated. The idea is that anyone who wants to add or remove enemies from the list needs to use these functions:

```
public List<Enemy> enemies;
public UnityEvent onChanged;

public void AddEnemy(Enemy enemy)
{
    enemies.Add(enemy);
    onChanged.Invoke();
}

public void RemoveEnemy(Enemy enemy)
{
    enemies.Remove(enemy);
    onChanged.Invoke();
}
```

Figure 8.33: Calling events when enemies are added or removed

Here, we have the problem that nothing stops us from bypassing those two functions and using the list directly. You can solve that by making the list private and exposing it using the `IReadOnlyList` interface. Remember that this way, the list won't be visible in the editor for debugging purposes.

- Change the `Enemy` script to use these functions:

```
public class Enemy : MonoBehaviour
{
    void Start()
    {
        EnemyManager.instance.AddEnemy(this);
    }

    void OnDestroy()
    {
        EnemyManager.instance.RemoveEnemy(this);
    }
}
```

Figure 8.34: Making the `Enemy` use the add and remove functions

- Repeat the same process for `WaveManager` and `WaveSpawner`, create an `onChanged` event, and create the `AddWave` and `RemoveWave` functions and call them in `WaveSpawner` instead of directly accessing the list. This way, we are sure the event is called when necessary as we did with `EnemyManager`. Try to solve this step by yourself and then check the solution in the following screenshot, starting with `WavesManager`:

```

public class WavesManager : MonoBehaviour
{
    public static WavesManager instance;

    public List<WaveSpawner> waves;
    public UnityEvent onChanged;

    private void Awake()
    {
        if (instance == null)
            instance = this;
        else
            Debug.LogError("Duplicated WavesManager", gameObject);
    }

    public void AddWave(WaveSpawner wave)
    {
        waves.Add(wave);
        onChanged.Invoke();
    }

    public void RemoveWave(WaveSpawner wave)
    {
        waves.Remove(wave);
        onChanged.Invoke();
    }
}

```

Figure 8.35: WaveManager OnChanged event implementation

- Also, `WavesSpawner` needed the following changes:

```

public class WaveSpawner : MonoBehaviour
{
    public GameObject prefab;
    public float startTime;
    public float endTime;
    public float spawnRate;

    private void Start()
    {
        WavesManager.instance.AddWave(this);
        InvokeRepeating("Spawn", startTime, spawnRate);
        Invoke("EndSpawner", endTime);
    }

    void Spawn()
    {
        Instantiate(prefab, transform.position, transform.rotation);
    }

    void EndSpawner()
    {
        WavesManager.instance.RemoveWave(this);
        CancelInvoke();
    }
}

```

Figure 8.36: Implementing the AddWave and RemoveWave functions

- In WavesGameMode, rename Update to CheckWinCondition and subscribe this function to the onChanged event of EnemyManager and the onChanged event of WavesManager. The idea is to check for the number of enemies and waves being changed only when necessary. Remember to do the subscription to the events in the Start function due to the Singletons being initialized in Awake :

```

void Start()
{
    playerLife.onDeath.AddListener(OnPlayerOrBaseDied);
    playerBaseLife.onDeath.AddListener(OnPlayerOrBaseDied);
    EnemyManager.instance.onChanged.AddListener(CheckWinCondition);
    WavesManager.instance.onChanged.AddListener(CheckWinCondition);
}

void OnPlayerOrBaseDied()
{
    SceneManager.LoadScene("LoseScreen");
}

void CheckWinCondition()
{
    if (EnemyManager.instance.enemies.Count <= 0 && WavesManager.instance.waves.Count <= 0)
    {
        SceneManager.LoadScene("WinScreen");
    }
}

```

Figure 8.37: Checking the win condition when the enemies or waves amount is changed

Regarding the Visual Scripting version, let's start checking the lose condition with events, checking first some changes needed in the **Life Script Graph**:

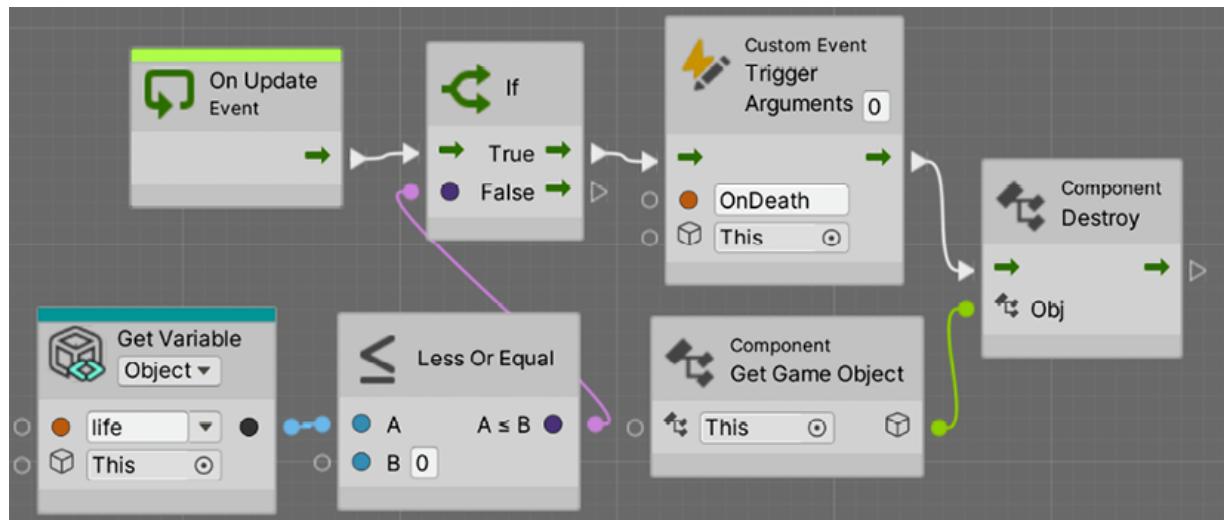


Figure 8.38: Triggering a Custom Event in our Life graph

First, after destroying the object when life reaches 0, we use the **Trigger Custom Event** node, specifying the name of our event is `OnDeath`. This will tell anyone waiting for the execution of the `OnDeath` event that we did. Remember, this is our **Life Script Graph**. Be sure to call `destroy` after triggering the event—while most of the time the order doesn't matter, given that the `destroy` action doesn't actually happen until the end of the frame, sometimes it can cause issues, so better be safe here. In this case, Game Mode should listen to the player's `OnDeath` event, so let's make the following change in our **Game Mode Graph**:

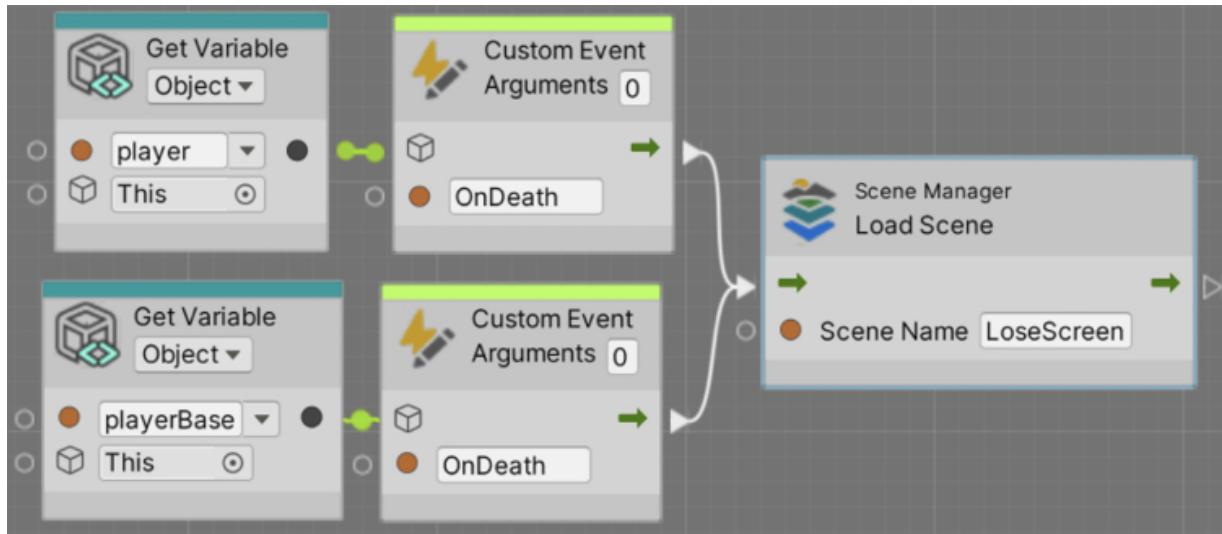


Figure 8.39: Listening to the `OnDeath` event of Player in Visual Scripting

We used the **CustomEvent** node connecting it to the player reference of our `GameMode`. This way we are specifying that if that player executes that event, we will execute the **Load Scene** node. Remember that the player reference is crucial to specify from whom we want to execute the `OnDeath` event, and remember that the **Life Visual Graph** will also be present in the enemies and we are not interested in them here. Also, remember to remove the `If` node and the condition nodes we used previously to detect this – the only

If our Game Mode will have is the one for the win condition. Essentially, we made any object with the `Life` script have an `OnDeath` event, and we made the GameMode listen to the `OnDeath` event of the player specifically. We could also do events for enemies and waves, but that would complicate our graphs somewhat, given that we don't have `WaveManager` or `EnemyManager` in the Visual Scripting versions. We could certainly create those to accomplish this, but sometimes the point of using Visual Scripting is to create simple logic, and these kinds of changes tend to make a graph grow quite a bit. Another possible solution is to make the enemy and wave directly inform the Game Mode. We could use **Trigger Custom Event** in the enemies and waves, connecting that node to the Game Mode, to finally let the Game Mode have a **Custom Event** node from which to listen. The issue is that that would violate the correct dependencies between our objects; lower-level objects such as enemies and waves shouldn't communicate with higher-level objects such as Game Mode. Essentially, Game Mode was supposed to be an overseer. If we apply the solution described in this paragraph, we won't be able to have an enemy in another scene or game without having a Game Mode. So, for simplicity and code decoupling purposes, let's keep the other conditions as they are—the more complex logic such as this will be probably handled in C# in full production projects. Yes, using events means that we have to write more code than before, and in terms of functionality, we didn't obtain anything new, but in bigger projects, managing conditions through `Update` checks will lead to different kinds of problems as previously discussed, such as race conditions and performance issues. Having a scalable code base sometimes requires more code, and this is one of those cases. Before we finish, something to consider is that Unity events are not the only way to create this kind of event communication in Unity; you will find a similar approach called **Action**, the native C# version of events, which I recommend you look into if you want to see all of the options out there.

We explored some programming patterns in this chapter, but there are plenty. You can learn more about them here:
<https://gameprogrammingpatterns.com/>

Summary

In this chapter, we finished an important part of the game: the ending, both by victory and by defeat. We discussed a simple but powerful way to separate the different layers of responsibilities by using managers created through Singletons, to guarantee that there isn't more than one instance of every kind of manager and simplifying the connections between them through static access. Also, we visited the concept of events to streamline communication between objects to prevent problems and create more meaningful communication between objects. With this knowledge, you are now able not only to detect the victory and lose conditions of the game but can also do it in a better-structured way. These patterns can be useful to improve our game code in general, and I recommend you try to apply them in other relevant scenarios. In the next chapter, we are going to start *Part 3* of the book, where we are going to see different Unity systems to improve the graphics and audio aspects of our game, starting by seeing how we can create materials to modify aspects of our objects, and create shaders with Shader Graph.

9 Starting your AI Journey: Building Intelligent Enemies for your Game

Join our book community on Discord

<https://packt.link/unitydev>



What is a game if not a great challenge to the player, who needs to use their character's abilities to tackle different scenarios? Each game imposes different kinds of obstacles for the player, but in our game, the primary challenge comes from the enemies. Imagine an enemy that can sense your presence and strategically plan its attack - this is what we aim to achieve through these AI techniques.

Creating challenging and believable enemies can be complex; they must behave like real characters, smart enough to present challenges yet not so formidable as to be invincible. We are going to use basic but sufficient AI techniques to make an AI capable of sensing its surroundings and, based on that information, making decisions on what to do, using **FSMs** or **Finite State Machines**, along with other techniques. Those decisions will be executed using **intelligent pathfinding**. In this chapter, we will examine the following AI concepts:

- Gathering information with sensors
- Making decisions with FSMs
- Executing FSM actions

These components are crucial in creating enemies that are not only reactive but also exhibit a semblance of intelligence and strategy. By the end of the chapter, you will have a fully functional enemy capable of detecting the player and attacking them, so let's start by seeing first how to make the sensor systems.

Gathering information with sensors

AI in games works in a three-step process: gathering information, analysing it, and executing actions based on such analysis. This is important as it reflects a simple version of how the human mind works, making the resulting AI more realistic. As you can see, we cannot do anything without information, so let's start with that part. There are several sources of information our AI can use, such as data about itself (life and bullets) or maybe some game state (winning condition or remaining enemies), which can easily be found with the code we've seen so far. One important source of information, however, is also sensors such as sight and hearing. In our case, sight will be enough, so let's learn how to code that. In this section, we will examine the following sensor concepts:

- Creating three-filter sensors with C#
- Creating three-filter sensors with Visual Scripting
- Debugging with gizmos

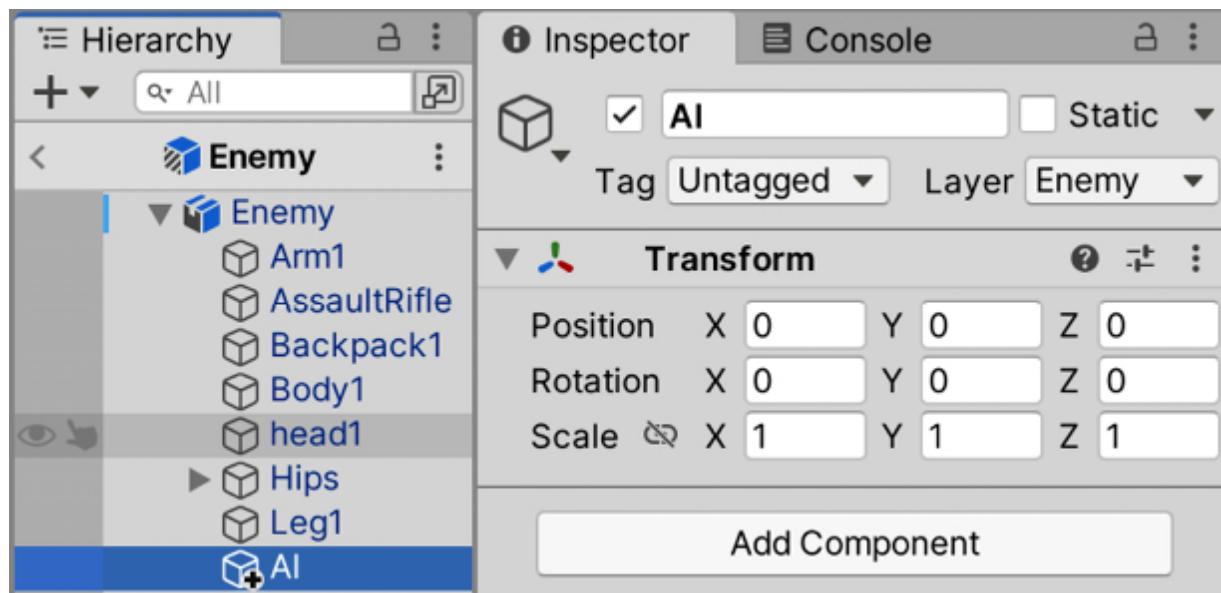
Let's start by seeing how to create a sensor with the three-filters approach.

Creating three-filter sensors with C#

The common way to code senses is through a three-filters approach to discard enemies out of sight. Each filter will discard objects that fall outside our sensors. The first filter is a distance filter, which will discard enemies too far away to be seen, then the second filter would be the angle check, which will check enemies inside our

viewing cone, and finally, the third filter is a raycast check, which will discard enemies that are being occluded by obstacles such as walls. Before starting, a word of advice: we will be using vector mathematics here, and covering those topics in-depth is outside the scope of this book. If you don't understand something, feel free to just search online for the code in the screenshots. Let's code sensors in the following way:

1. Create an empty `GameObject` called `AI` as a child of the **Enemy** Prefab. You need to first open the Prefab to modify its children (double-click the Prefab). Remember to set the transform of this `GameObject` to **Position (0, 1.75, 0)**, **Rotation (0, 0, 0)**, and **Scale (1, 1, 1)** so it will be aligned with the enemy's eyes. This is done this way for the future sight sensors we will do. Consider your enemy prefab might have a different height for the eyes. While we can certainly just put all AI scripts directly in the **Enemy** Prefab root `GameObject`, we did this just for separation and organization:



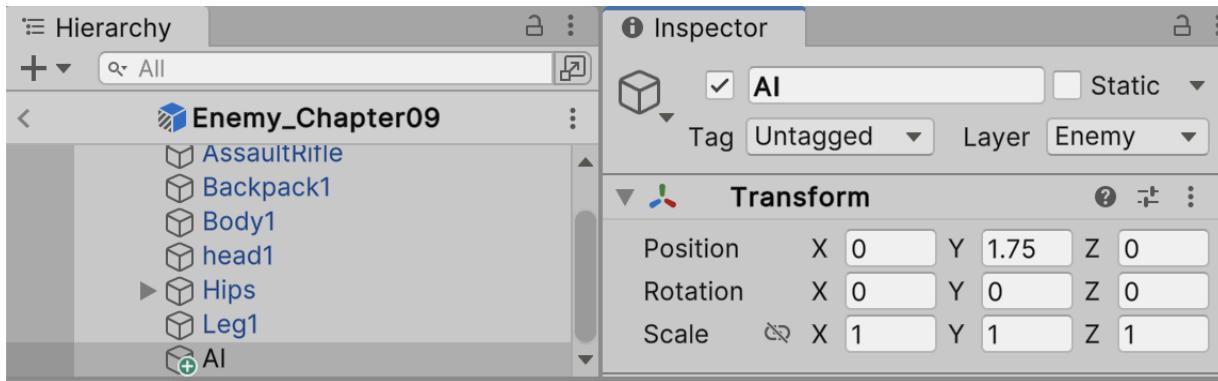


Figure 9.1: AI scripts container

1. Create a script called `Sight` and add it to the `AI` child object.
2. Create two fields of the `float` type called `distance` and `angle`, and another two of the `LayerMask` type called `obstaclesLayers` and `objectsLayers`. `distance` will be used as the vision distance, `angle` will determine the amplitude of the view cone, `obstacleLayers` will be used by our obstacle check to determine which objects are considered obstacles, and `objectsLayers` will be used to determine what types of objects we want the `Sight` component to detect.
3. Info
4. We just want the sight to see enemies; we are not interested in objects such as walls or power-ups. `LayerMask` is a property type that allows us to select one or more layers to use inside code, so we will be filtering objects by layer.

```
using UnityEngine;

public class Sight : MonoBehaviour
{
    public float distance;
    public float angle;
    public LayerMask objectsLayers;
    public LayerMask obstaclesLayers;
}
```

Figure 9.2: Fields to parametrize our sight check

5. In `Update`, call `Physics.OverlapSphere` as in the *Figure 9.3*.

This function creates an imaginary sphere in the place specified by the first parameter (in our case, our position) and with a radius specified in the second parameter (the `distance` property) to detect objects with the layers specified in the third parameter (`ObjectsLayers`). It will return an array with all the colliders found inside the sphere; these functions use physics to carry out the check, so the objects must have at least one collider. This is the method we will be using to find all enemies inside our view distance, and we will be further filtering them in the next steps. Note that we are passing our position to the first parameter, which is not actually the position of the enemy but the position of the `AI` child object, given our script is located there. This highlights the importance of the position of the `AI` object. Another way of accomplishing the first check is to just check the distance from the objects we want to see to the player, or if looking for other kinds of

objects, to a `Manager` component containing a list of them. However, the method we chose is more versatile and can be used for any kind of object. Also, you might want to check the `Physics.OverlapSphereNonAlloc` version of this function, which does the same but is more performant by not allocating an array to return the results.

1. Iterate over the array of objects returned by the function using a `for` loop:

```
private void Update()
{
    Collider[] colliders = Physics.OverlapSphere(
        transform.position, distance, objectsLayers);

    for (int i = 0; i < colliders.Length; i++)
    {
        Collider collider = colliders[i];
    }
}
```

Figure 9.3: Getting all `GameObjects` at a certain distance

2. To detect whether the object falls inside the vision cone, we need to calculate the angle between our viewing direction and the direction from ourselves towards the object itself. If the angle between those two directions is less than our cone angle, we consider that the object falls inside our vision. We will do that in the following steps:

Start calculating the direction toward the object, which can be done by normalizing the difference between the object's position and ours, like in *Figure 9.4*. You might notice we used `bounds.center` instead of `transform.position`; this way, we check the direction to

the center of the object instead of its pivot. Remember that the player's pivot is in the ground and the ray check might collide against it before the player:

```
Vector3 directionToCollider = Vector3.Normalize(  
    collider.bounds.center - transform.position);
```

```
Vector3 directionToController = Vector3.Normalize(  
    collider.bounds.center - transform.position);
```

+Figure 9.4: Calculating direction from our position toward the collider

1. We can use the `Vector3.Angle` function to calculate the angle between two directions. In our case, we can calculate the angle between the direction toward the enemy and our forward vector to see the angle:

```
float angleToCollider = Vector3.Angle(  
    transform.forward, directionToCollider);
```

Figure 9.5: Calculating the angle between two directions

If you want, you can instead use `Vector3.Dot`, which will execute a dot product, a mathematics function to calculate the length of a vector projected to another (search online for more info).

`Vector3.Angle` actually uses that one, but converts the result of the dot product into an angle, which needs to use trigonometry, and that can be time expensive to calculate. But our `Vector3.Angle` approach is simpler and faster to code, and given that we don't require many sensors because we won't have many enemies,

optimizing the sensor using dot products is not necessary now, but consider that for games with larger scale.

1. Now check whether the calculated angle is less than the one specified in the `angle` field. Note that if we set an angle of `90`, it will actually be `180`, because if the `Vector3.Angle` function returns, as an example, `30`, it could be `30` to the left or to the right. If our angle says `90`, it could be both `90` to the left and to the right, so it will detect objects in a 180-degree arc.
2. Use the `Physics.Linecast` function to create an imaginary line between the first and the second parameter (our position and the collider position) to detect objects with the layers specified in the third parameter (the obstacle layers) and return `boolean` indicating whether that ray hit something or not.

The idea is to use the line to detect whether there are any obstacles between ourselves and the detected collider, and if there is no obstacle, this means that we have a direct line of sight toward the object. Observe how we use the `!` or `not` operator in *Figure 9.6* to check if `Physics.Linecast` didn't detect any objects. Again, note that this function depends on the obstacle objects having colliders, which in our case, we have (walls, floor, and so on):

```
if (angleToCollider < angle)
{
    if (!Physics.Linecast(transform.position,
        collider.bounds.center, obstaclesLayers))
    {
    }
}
```

Figure 9.6: Using a Linecast to check obstacles between the sensor and the target object

1. If the object passes the three checks, that means that this is the object we are currently seeing, so we can save it inside a field of the `Collider` type called `detectedObject`, to save that information for later usage by the rest of the `AI` scripts.

Consider using `break` to stop the `for` loop that is iterating the colliders to prevent wasting resources by checking the other objects, and to set `detectedObject` to `null` before `for` to clear the result from the previous frame. So if in this frame, we don't detect anything, it will keep the `null` value so we notice that there is nothing in the sensor:

```
public float distance;
public float angle;
public LayerMask objectsLayers;
public LayerMask obstaclesLayers;
public Collider detectedObject;

private void Update()
{
    Collider[] colliders = Physics.OverlapSphere(
        transform.position, distance, (int) objectsLayers);

    detectedObject = null;
    for (int i = 0; i < colliders.Length; i++)
    {
        Collider collider = colliders[i];

        Vector3 directionToCollider = Vector3.Normalize(
            collider.bounds.center - transform.position);

        float angleToCollider = Vector3.Angle(
            transform.forward, directionToCollider);
    }
}
```

```
if (angleToCollider < angle)
{
    if (!Physics.Linecast(transform.position,
        collider.bounds.center, (int) obstaclesLayers))
    {
        detectedObject = collider;
        break;
    }
}
}
```

Figure 9.7: Full sensor script

In our case, we are using the sensor just to look for the player, the only object the sensor is in charge of looking for, but if you want to make the sensor more advanced, you can just keep a list of detected objects, placing inside it every object that passes the three tests instead of just the first one. In our case, it's not necessary given we have only one player in the game.

1. In the editor, configure the sensor at your will. In this case, we will set `objectsLayer` to `Player` so our sensor will focus its search on objects with that layer, and `obstaclesLayer` to `Default`, the layer we used for walls and floors. Remember the `Sight` script is in the `AI` `GameObject`, which is a child of the `Enemy` `prefab`:

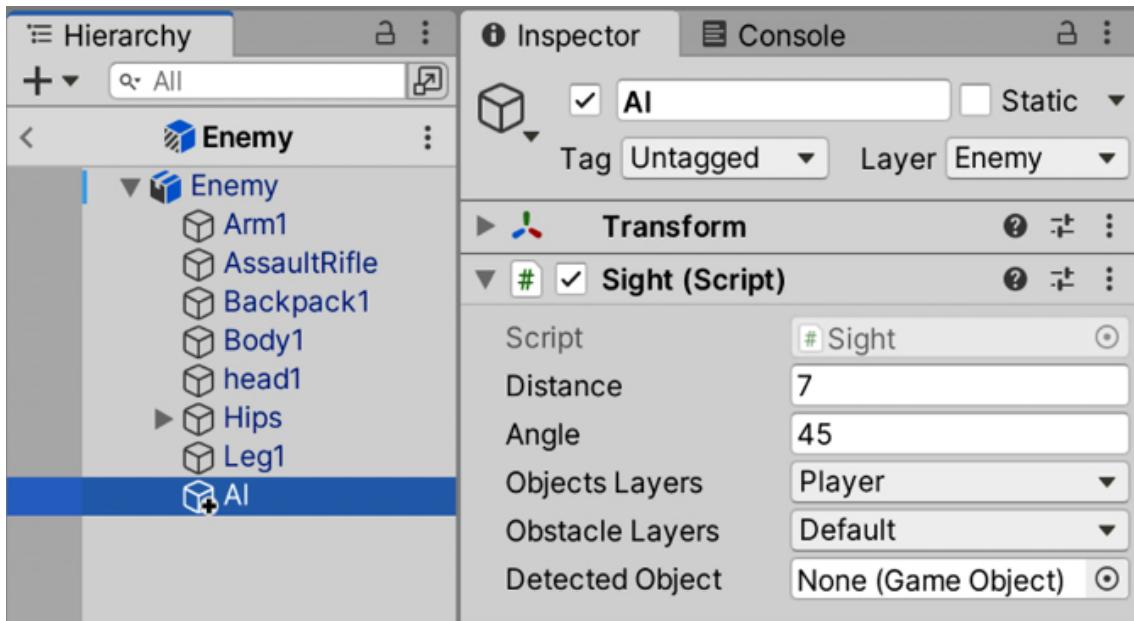


Figure 9.8: Sensor settings

2. To test this, just place an enemy with a movement speed of 0 in front of the player, select its `AI` child object and then play the game to see how the property is set in the Inspector. Also, try putting an obstacle between the two and check that the property says `None (null)`. If you don't get the expected result, double-check your script, its configuration, and whether the player has the `Player` layer, and the obstacles have the `Default` layer. Also, you might need to raise the `AI` object a little bit to prevent the ray from starting below the ground and hitting it. Feel free to make further tests to really understand your code.

Now that we understand how the sensors work in C#, let's see the visual scripting version.

Creating Three-Filters sensors with Visual Scripting

Regarding the Visual Scripting version, let's check it part by part, starting with the **Overlap Sphere**:

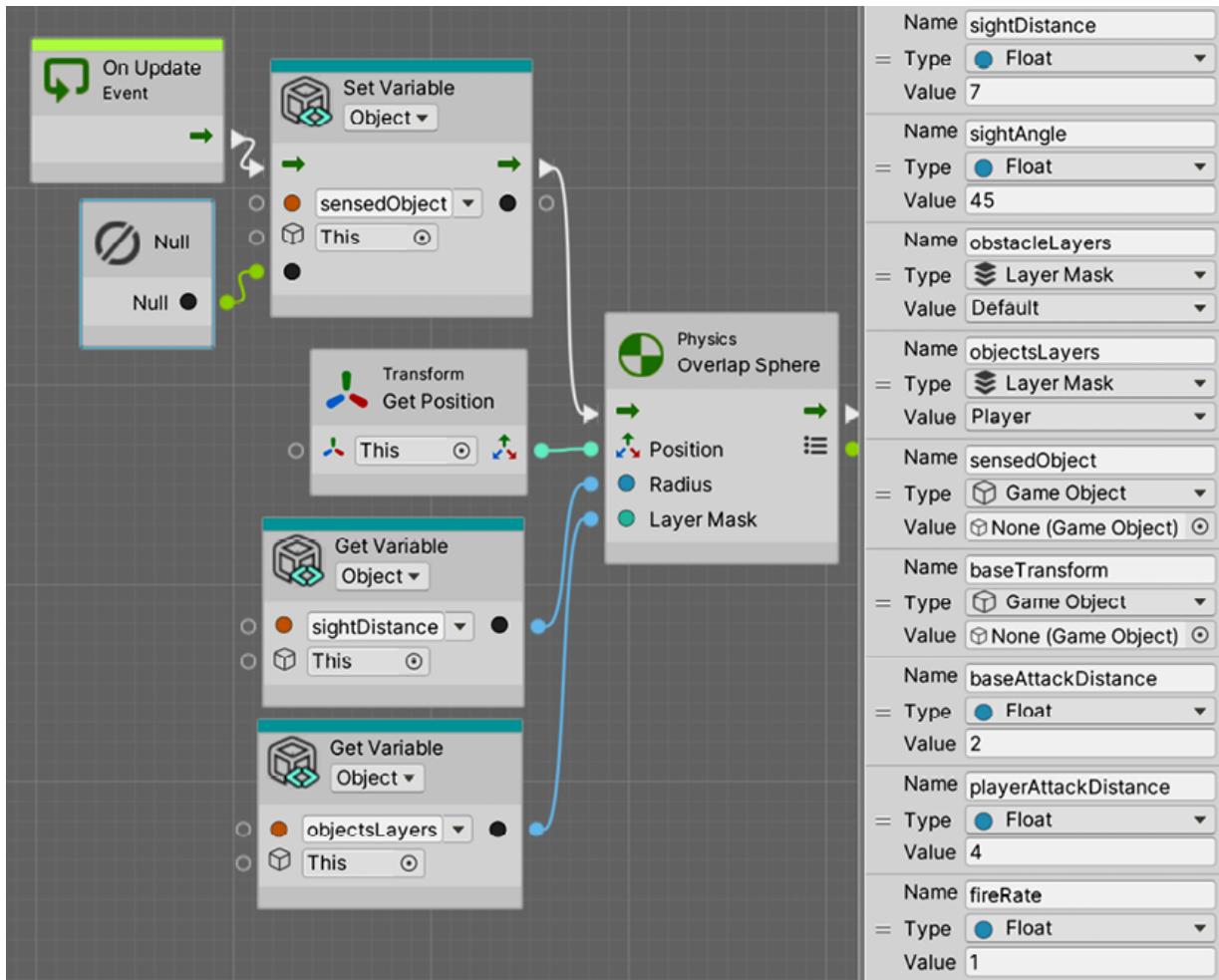


Figure 9.9: Overlap Sphere in Visual Scripting

So far, we just called **Overlap Sphere** after setting the `sensedObject` variable to `null`. The previous image contains not only the variables we need so far, but also the ones that we will use later, so remember to create all of them. Something to consider is how the `sensedObject` variable in the **Variables** component in the Inspector might instead have a `Null` type in your case, which means no type in Visual Scripting). This can't be possible in C#—all variables must have a type—and while we could set the `sensedObject` variable to the proper type (**Collider**), we will keep the variable type to be set later via a script. Even if we set the type now, Visual Scripting tends to forget the type if no value is set, and we cannot set it until we detect something. Don't worry about that

for the moment; when we set the variable through our script it will acquire the proper type. Actually, all variables in Visual Scripting can switch types at runtime according to what we set them to, given how the **Variables** component works. I don't recommend changing a variable's type in runtime, as it will give a different meaning to it. Try to stick with the intended variable type. We just said that all variables in C# must have a type, but that's not entirely true. There are ways to create dynamically typed variables, but it's not a good practice that I'd recommend using unless no other option is present. Another thing to observe is how we set the `sensedObject` variable to `null` at the beginning using the **Null** node, which effectively represents the `null` value. Now, let's explore the **Foreach** part:

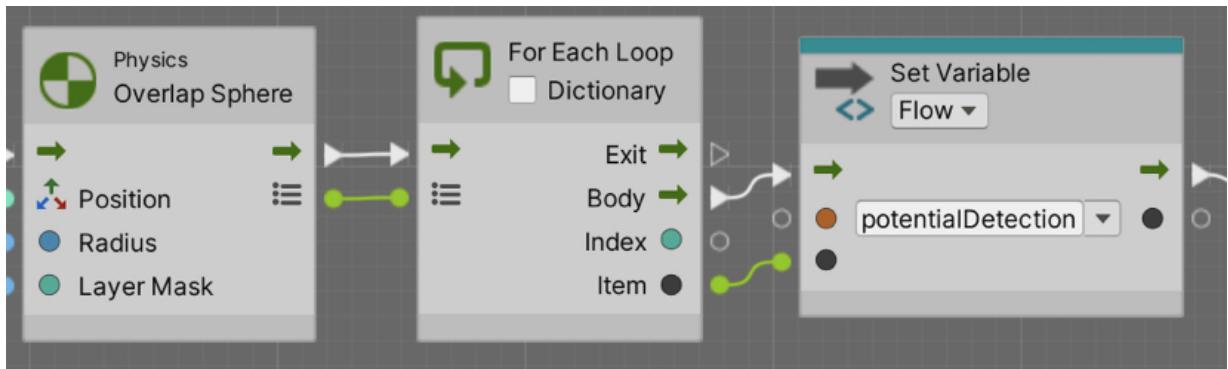


Figure 9.10: Iterating collections in Visual Scripting

We can see that one of the output pins of **Overlap Sphere** has a list icon, which essentially represents the collider array returned by **Overlap Sphere**. We connect that pin to the **For Each Loop** node, which as you might imagine iterates over the elements of the provided collection (array, list, dictionary, etc.). The **Body** pin represents the nodes to execute in each iteration of the loop, and the **Item** output pin represents the item currently being iterated—in our case, one of the colliders detected in **Overlap Sphere**. Finally, we save that item in a **Flow** `potentialDetection` variable, **Flow** variables being the equivalent to local variables in C# functions. To maintain clarity in our visual scripting graph and avoid clutter, we

assign the currently iterated collider to a 'Flow' variable named 'potentialDetection.' This approach eliminates the need for extensive connections across the graph, simplifying the visual layout and subsequent referencing of this collider. Now let's explore the **Angle** check:

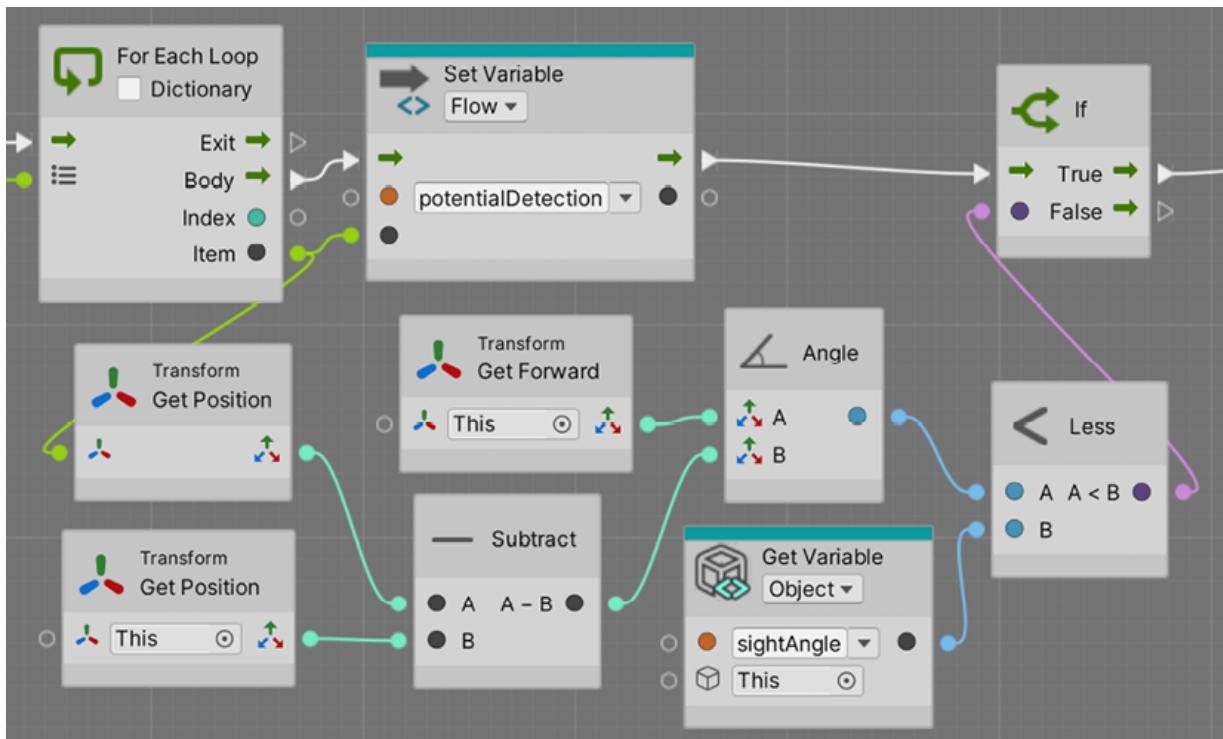


Figure 9.11: Angle check in Visual Scripting

Here, you can see a direct translation of what we did in C# to detect the angle, so it should be pretty self-explanatory. Now, let's explore the **Linecast** part:

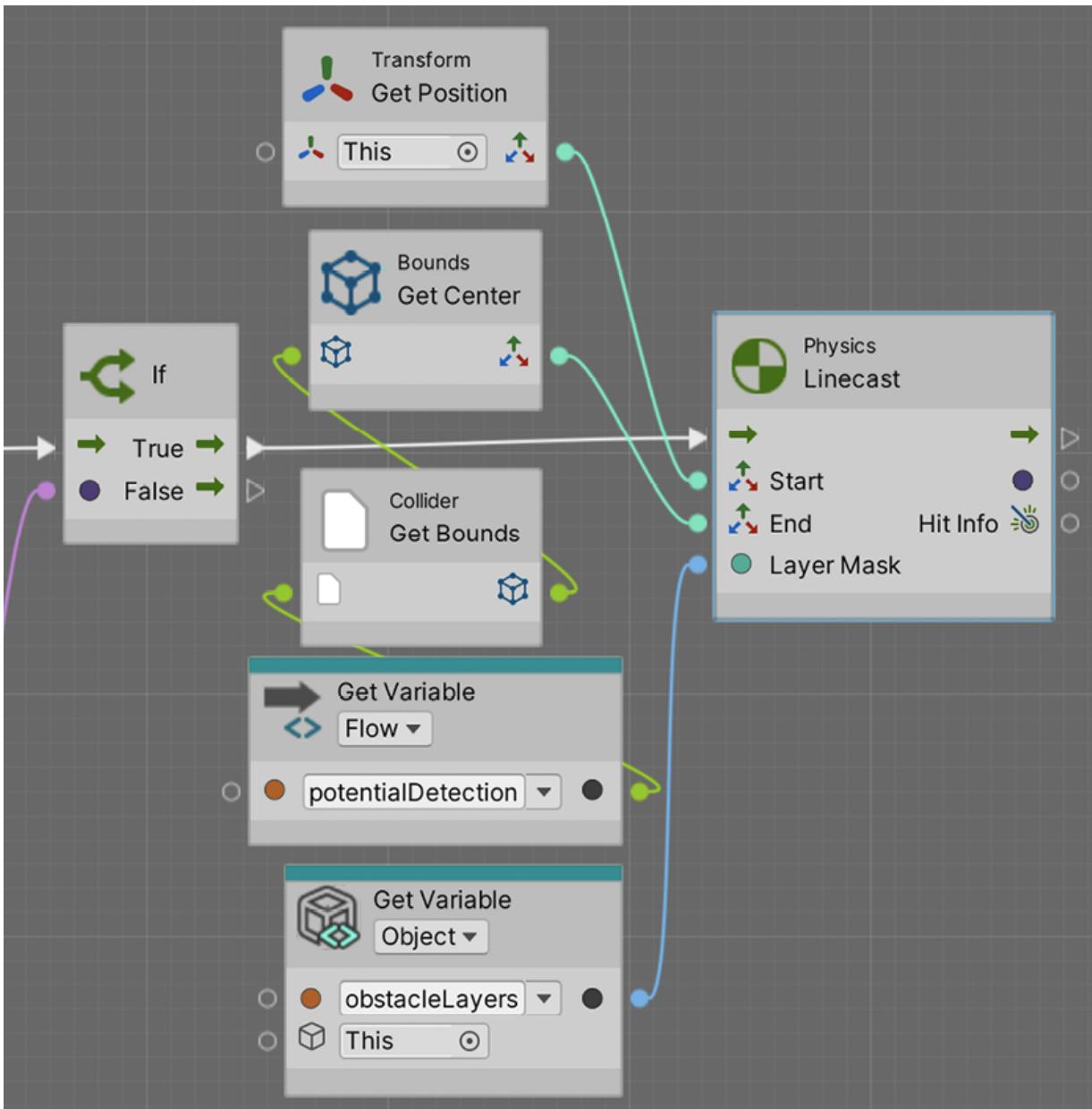


Figure 9.12: Linecast check in Visual Scripting

Again, essentially the same as we did before in C#. The only thing to highlight here is the fact we used the **Flow** variable

`potentialDetection` to again get the position of the current item being iterated, instead of connecting the **Get Position** node all the way to the **Foreach Item** output pin. Now, let's explore the final part:

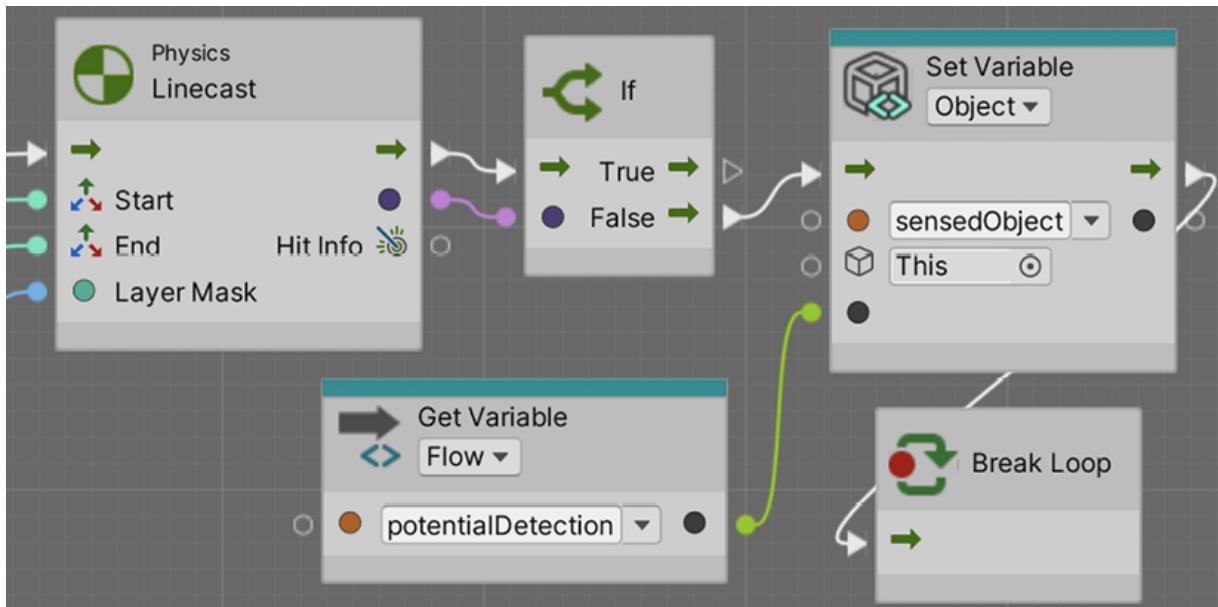


Figure 9.13: Setting the sensedObject

Again, pretty much self-explanatory; if the **Linecast** returns `false`, we set the `potentialDetection` variable (the currently iterated item) as the `sensedObject` variable (the one that will be accessed by other scripts later to query which is the object our AI can see right now). Something to consider here is the usage of the **Break Loop** node, which is the equivalent to the C# `break` keyword; essentially, we are stopping the **ForEach** loop we are currently in. Now, even if we have our sensor working, sometimes checking whether it's working or configured properly requires some visual aids we can create using gizmos.

Debugging with gizmos

As we create our AI, we will start to detect certain errors in edge cases, usually related to misconfigurations. You may think that the player falls within the sight range of the enemy but maybe you cannot see that the line of sight is occluded by an object, especially as the enemies move constantly. A good way to debug those scenarios is through editor-only visual aids known as `Gizmos`, which allow you to visualize invisible data such as the sight distance

or the `Linecasts` executed to detect obstacles. Let's start seeing how to create `Gizmos` drawing a sphere representing the sight distance by doing the following:

1. In the `Sight` script, create an event function called `OnDrawGizmos`. This event is only executed in the editor (not in builds) and is the place to draw any `Gizmos` in Unity.
2. Use the `Gizmos.DrawWireSphere` function, passing our position as the first parameter and the distance as the second parameter to draw a sphere in our position with the radius of our distance. You can check how the size of the `Gizmo` changes as you change the distance field:

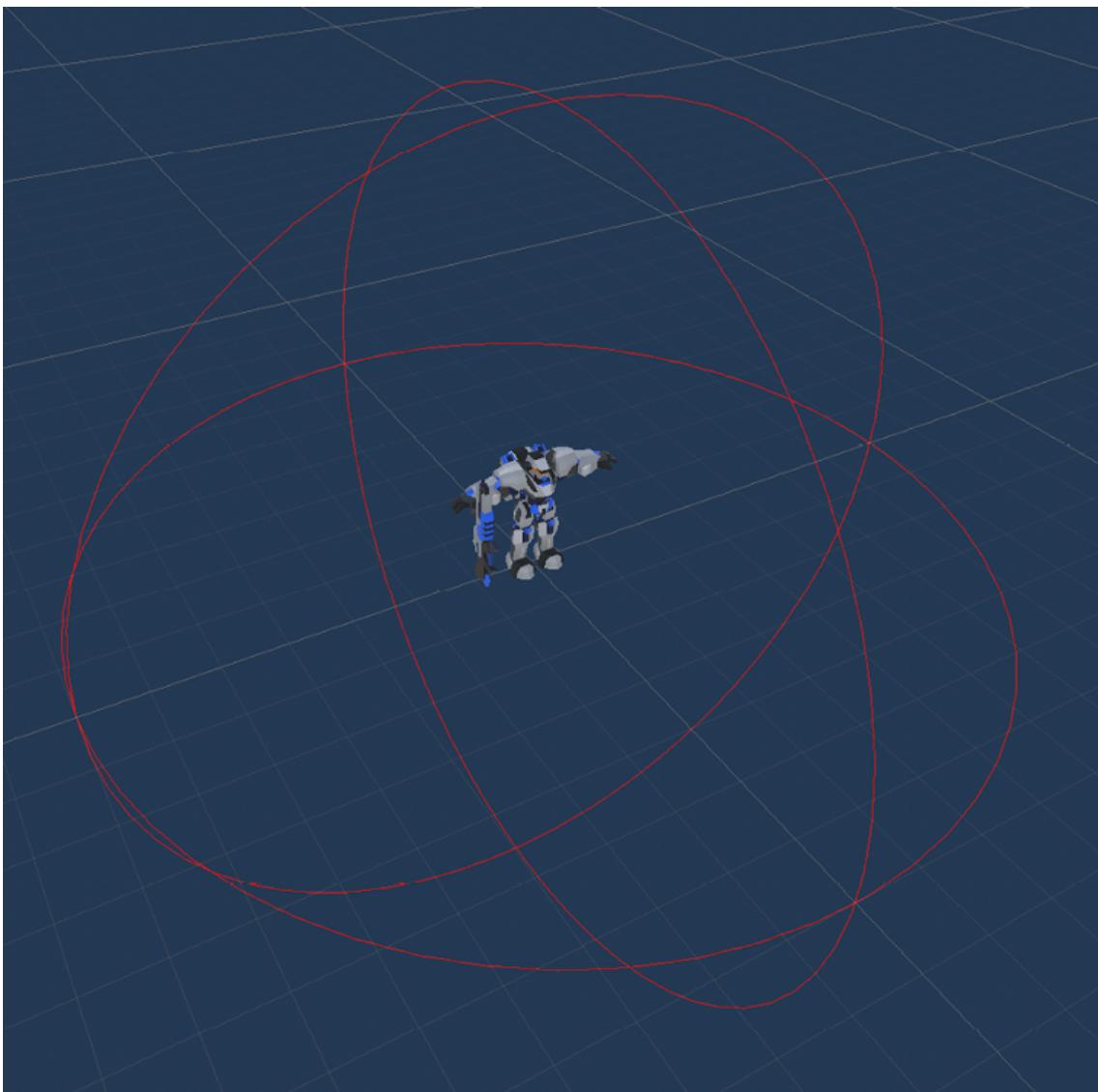


Figure 9.14: Sphere Gizmo

3. Optionally, you can change the color of the gizmo, setting `Gizmos.color` prior to calling the drawing functions:

```

void OnDrawGizmos()
{
    Gizmos.color = Color.red;
    Gizmos.DrawWireSphere(transform.position, distance);
}

```

Figure 9.15: Gizmos drawing code

Now you are drawing `Gizmos` constantly, and if you have lots of enemies, they can pollute the scene view with too many `Gizmos`. In that case, try the `OnDrawGizmosSelected` event function instead, which draws `Gizmos` only if the object is selected.

1. We can draw the lines representing the cone using `Gizmos.DrawRay`, which receives the origin of the line to draw and the direction of the line, which can be multiplied by a certain value to specify the length of the line, as in the following screenshot:

```

Vector3 rightDirection = Quaternion.Euler(0, angle, 0) * transform.forward;
Gizmos.DrawRay(transform.position, rightDirection * distance);

Vector3 leftDirection = Quaternion.Euler(0, -angle, 0) * transform.forward;
Gizmos.DrawRay(transform.position, leftDirection * distance);

```

Figure 9.16: Drawing rotated lines

2. In the screenshot, we used `Quaternion.Euler` to generate a quaternion based on the angles we want to rotate. A quaternion is a mathematical construct to represent rotations; please search for this term for more info on it. If you multiply this quaternion by a direction, we will get the rotated direction. We are taking our forward vector and rotating it according to the angle field to generate our cone vision lines.

Also, we multiply this direction by the sight distance to draw the line as far as our sight can see; you will see how the line matches the end of the sphere this way:

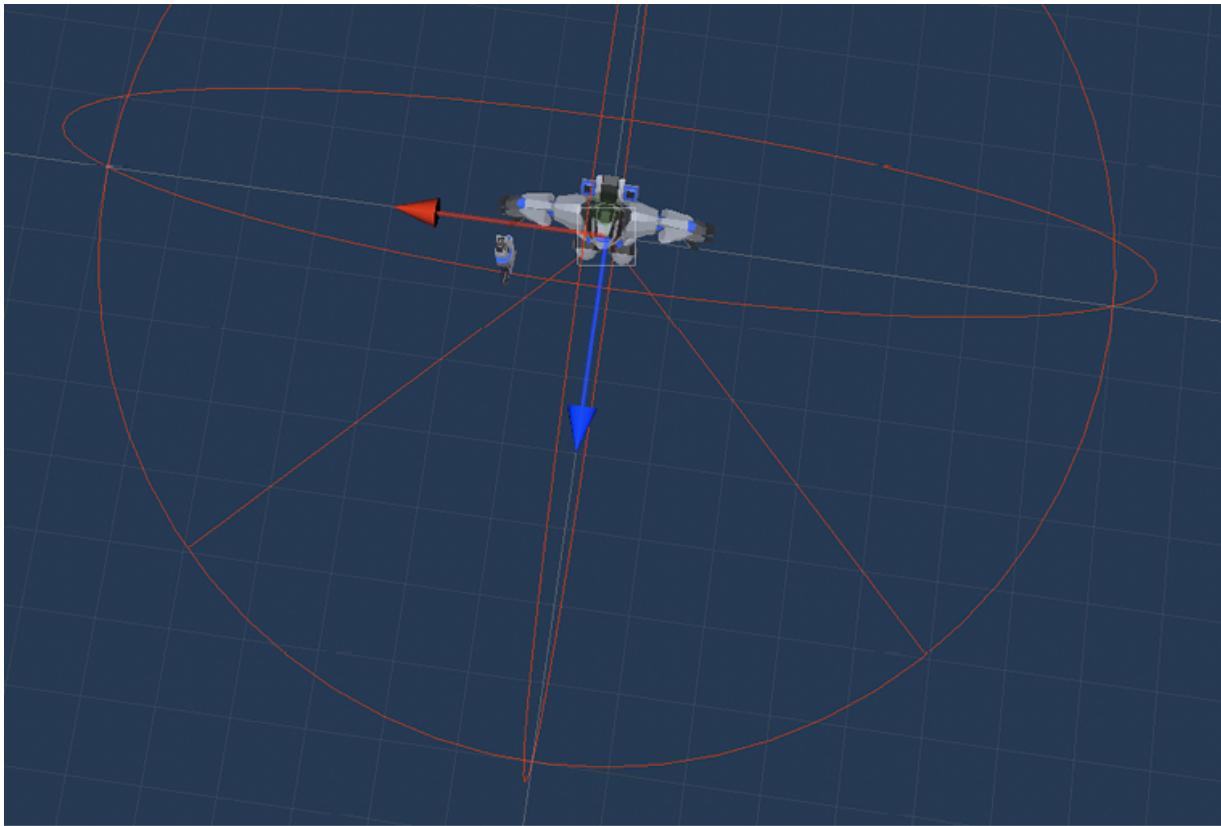


Figure 9.17: Vision angle lines

We can also draw the Linecasts, which check the obstacles, but as those depend on the current situation of the game, such as the objects that pass the first two checks and their positions, we can use `Debug.DrawLine` instead, which can be executed in the `Update` method. This version of `DrawLine` is designed to be used in runtime only. The `Gizmos` we saw also execute in the editor. Let's try them the following way:

1. First, let's debug the scenario where `Linecast` didn't detect any obstacles, so we need to draw a line between our sensor and the object. We can call `Debug.DrawLine` in the `if` statement that calls `Linecast`, as in the following screenshot:

```
if (angleToCollider < angle)
{
    if (!Physics.Linecast(transform.position,
        collider.bounds.center, obstaclesLayers))
    {
        Debug.DrawLine(transform.position,
            collider.bounds.center, Color.green);
        detectedObject = collider;
        break;
    }
}
```

Figure 9.18: Drawing a line in Update

2. In the next screenshot, you can see `DrawLine` in action:

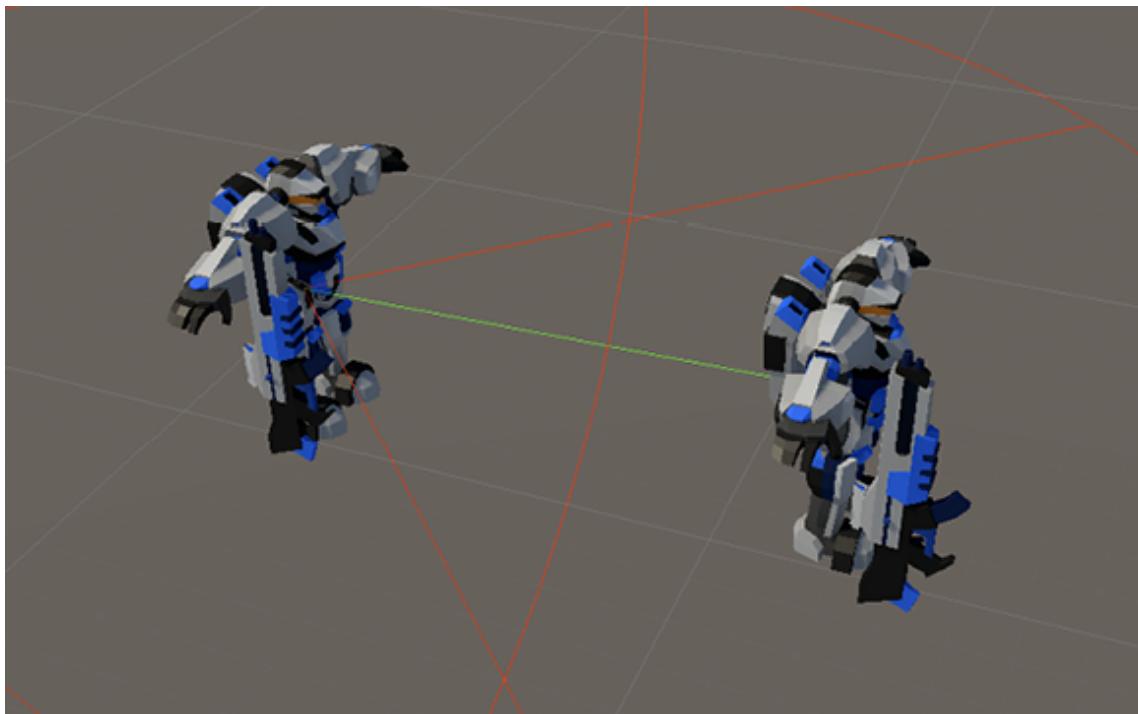


Figure 9.19: Line toward the detected Object

3. We also want to draw a line in red when the sight is occluded by an object. In this case, we need to know where the `Linecast` hit, so we can use an overload of the function, which provides an `out` parameter that gives us more information about what the line collided with, such as the position of the hit and the normal and the collided object, as in the following screenshot:

```
if (!Physics.Linecast(  
    transform.position, collider.bounds.center,  
    out RaycastHit hit, obstaclesLayers))  
{
```

Figure 9.20: Getting information about Linecast

Info Parameters using the `out` keyword allows the method to return data also via parameters. For more info on this check the following link: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out>

Note that `Linecast` doesn't always collide with the nearest obstacle but with the first object it detects in the line, which can vary in order. If you need to detect the nearest obstacle, look for the `Physics.Raycast` version of the function.

1. We can use that information to draw the line from our position to the hit point in `else` of the `if` sentence when the line collides with something:

```
if (!Physics.Linecast(
    transform.position, collider.bounds.center,
    out RaycastHit hit, obstaclesLayers))
{
    Debug.DrawLine(transform.position,
        collider.bounds.center, Color.green);
    detectedObject = collider;
    break;
}
else
{
    Debug.DrawLine(transform.position, hit.point, Color.red);
}
```

Figure 9.21: Drawing a line if we have an obstacle

2. In the next screenshot, you can see the results. Ensure the Gizmos option is on in the Scene view toolbars (the rightmost sphere-shaped icon):

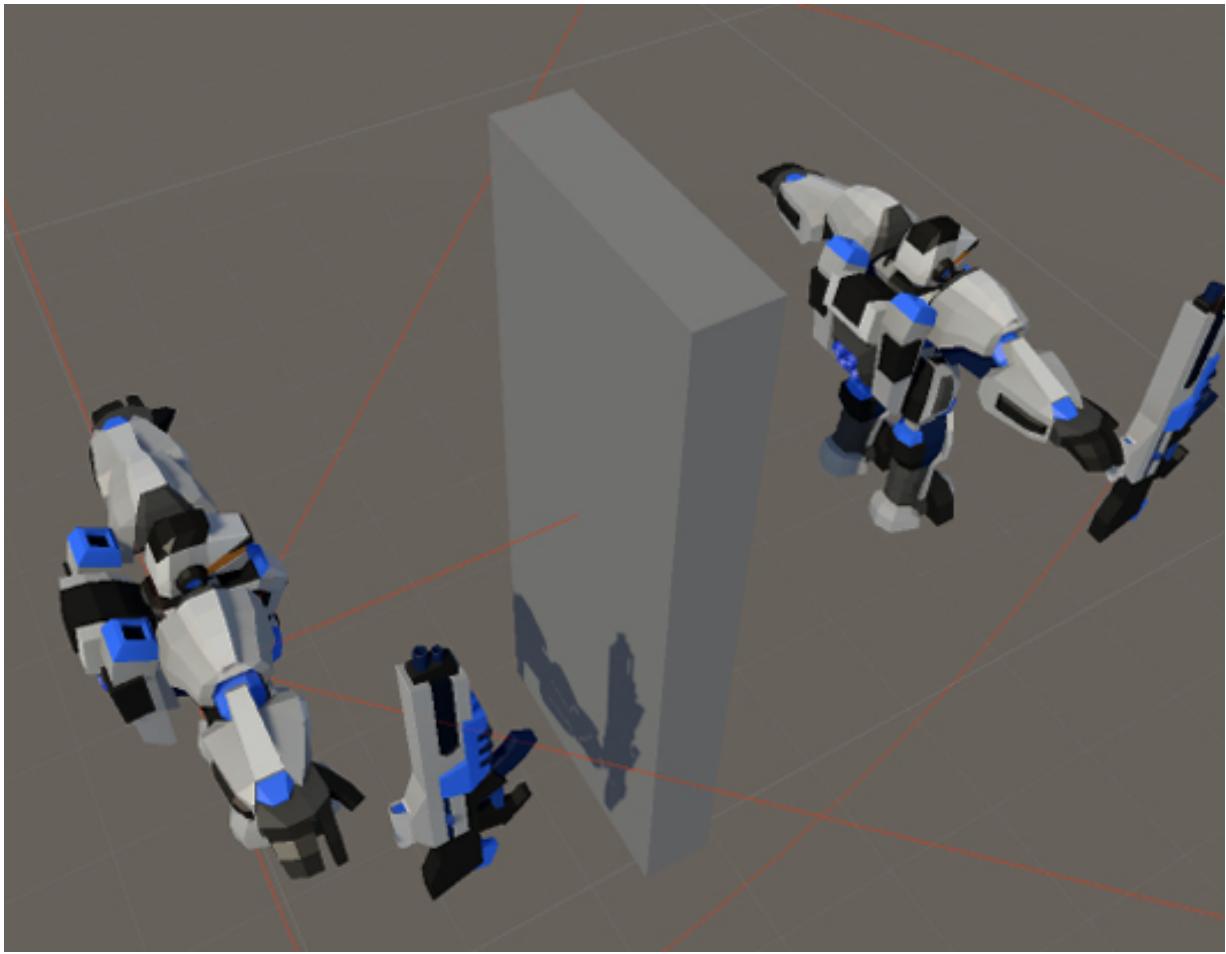


Figure 9.22: Line when an obstacle occludes vision

Regarding the Visual Scripting version, the first part will look like this:

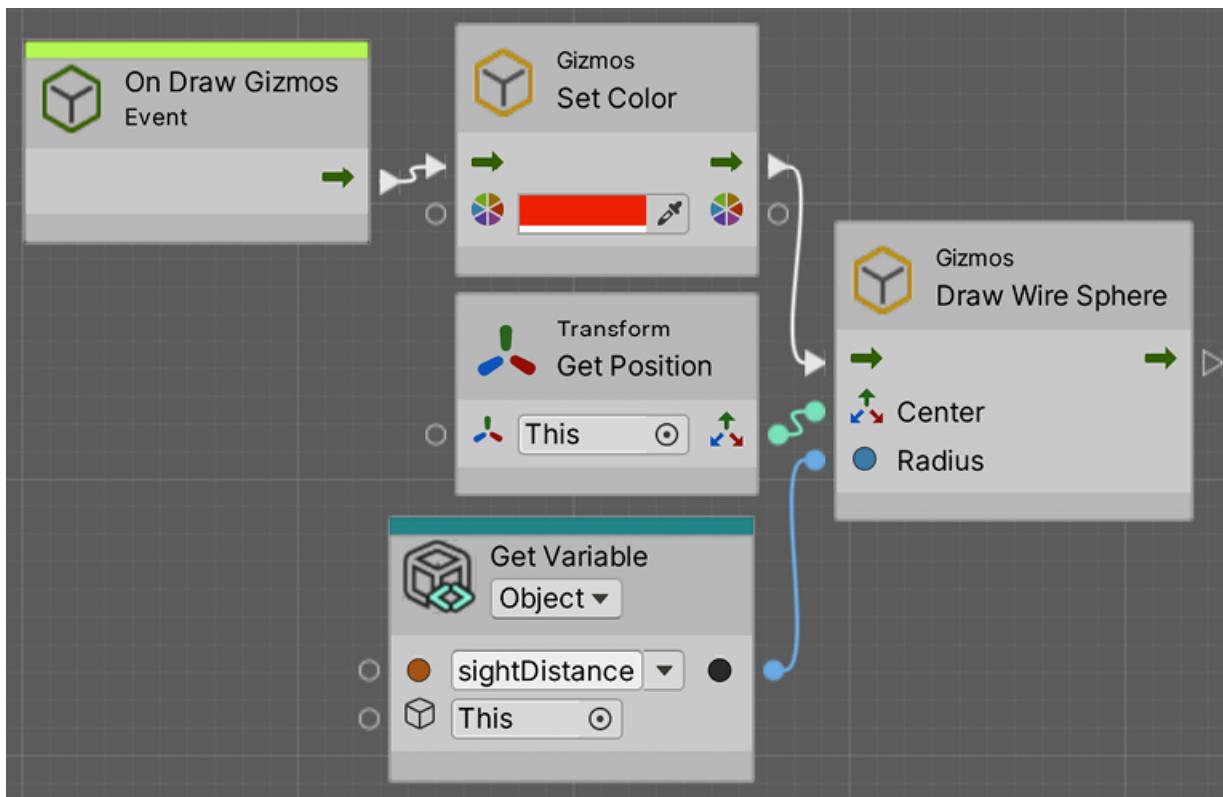


Figure 9.23: Drawing Gizmos with Visual Scripting

Then, the angle lines would look like this:

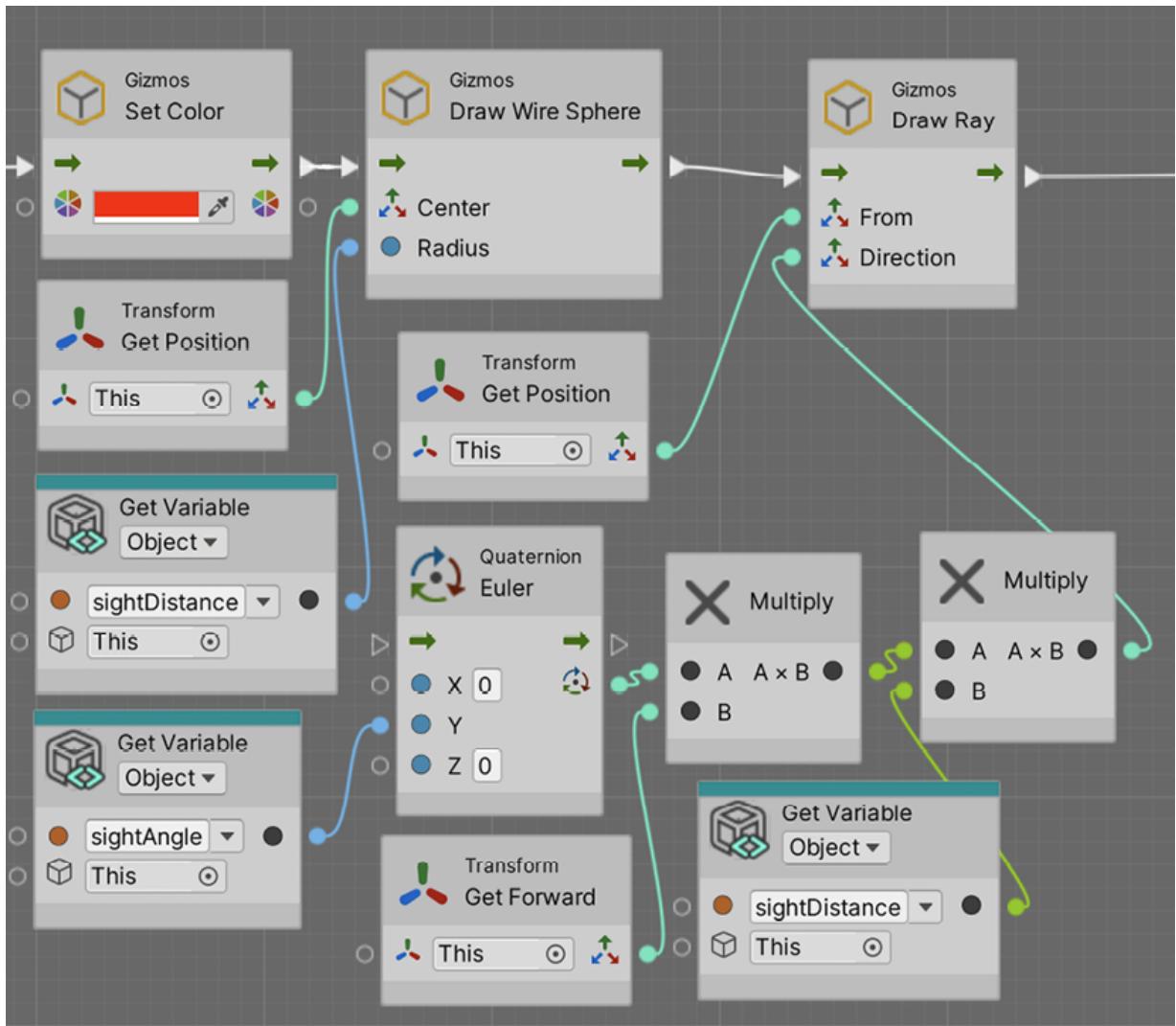


Figure 9.24: Drawing Angle lines of sight in Visual Scripting

Note that, here, we are showing just one, but the other is essentially the same but multiplying the angle by -1. Finally, the red lines towards the detected object and obstacles will look like this:

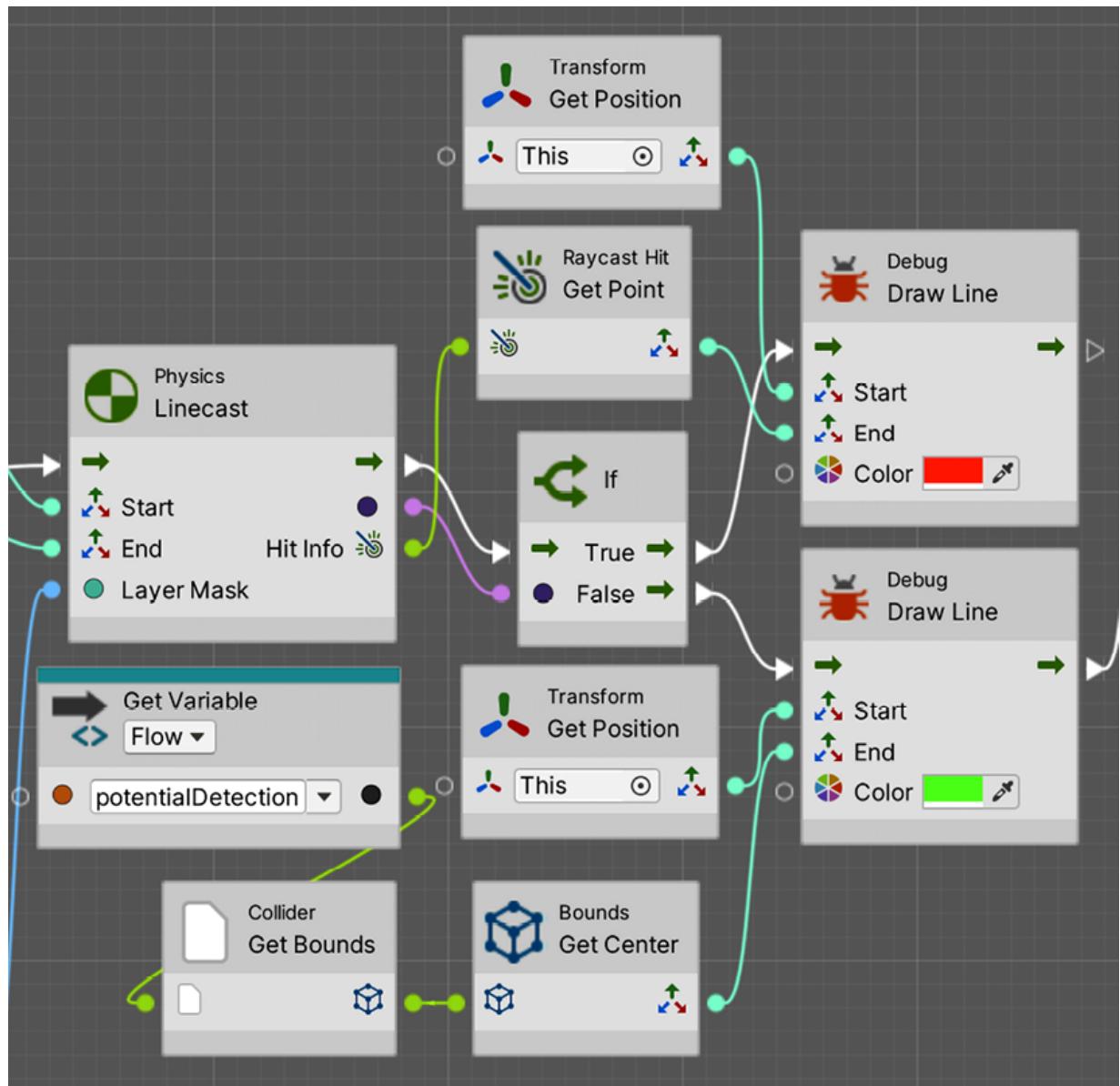


Figure 9.25: Drawing lines towards obstacles or detected objects in Visual Scripting

Note that, to accomplish this last one, we needed to change the previous **Linecast** node for the version that returns **Raycast Hit** info at the end. In this section we created the sensors system that will give sight to our AI and plenty of info about what to do next. Now that we have our sensors completed, let's use the information provided by them to make decisions with FSMs.

Making decisions with FSMs

We explored the concept of **Finite State Machines (FSMs)** in the past when we used them in the `Animator` component. To recap it we recommend reviewing Chapter 17, Animated Realities: Creating Animations with Animator, Cinemachine, and Timeline. We learned that an FSM is a collection of states, each one representing an action that an object can be executing at a time, and a set of transitions that dictates how the states are switched. This concept is not only used in animation but in a myriad of programming scenarios, and one of the common ones is AI. For AI, each state will represent a different possible AI behaviour to be active at a time, and transitions will represent the conditions that need to be met for other AI behaviours to be active. For example, in a shooter game the enemies can have states like being Idle, Patrolling, Attacking, Fleeing, Taking Cover and so on.

Info

To further reinforce the FSM concept, we recommend reviewing this link:

<https://gameprogrammingpatterns.com/state.html>

In this section, we will examine the following AI FSM concepts:

- Creating the FSM in C#
- Creating transitions
- Creating the FSM in Visual Scripting

Let's start implementing this FSM theory by creating a FSM in C#.

Creating the FSM in C#

To create our own FSM, we need to recap some basic concepts. Remember that an FSM can have a state for each possible action it can execute and that only one can be executed at a time. In terms of

AI, for example, we can be patrolling, attacking, fleeing, and so on. Also, remember that there are transitions between states that determine conditions to be met to change from one state to another, and in terms of AI, this can be the user being near the enemy to start attacking or life being low to start fleeing. In the next figure, you can find a simple reminder example of the two possible states of a door:

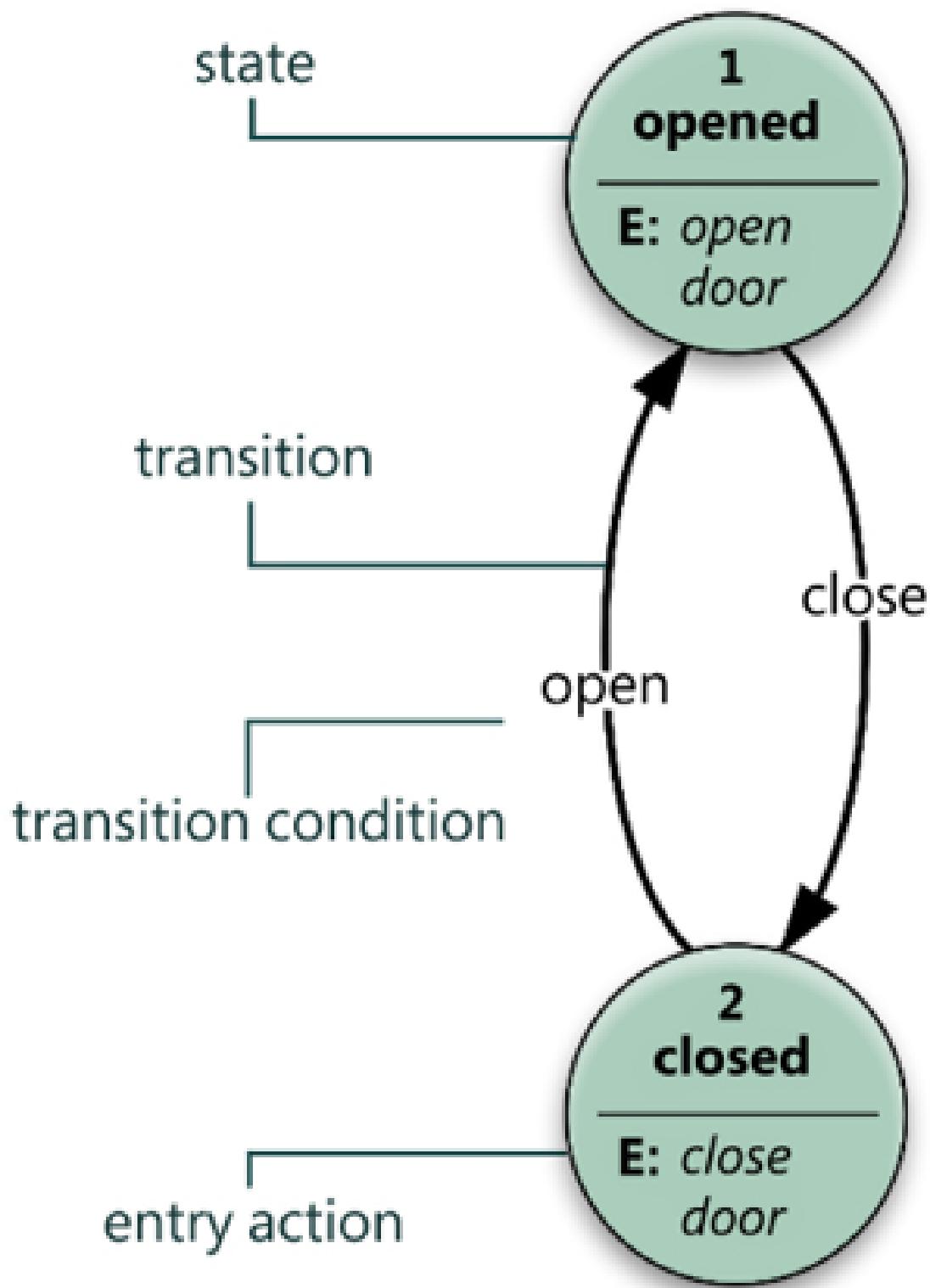


Figure 9.26: FSM skeleton

There are several ways to implement FSMs for AI; you can even use the `Animator` component if you want to or download some FSM system from the Asset Store. In our case, we are going to take the simplest approach possible, a single script with a set of `If` sentences, which can be basic but is still a good start to understanding the concept. Let's implement it by doing the following:

1. Create a script called `EnemyFSM` in the `AI` child object of the enemy.
2. Create an `enum` called `EnemyState` with the `GoToBase`, `AttackBase`, `ChasePlayer`, and `AttackPlayer` values. We are going to have those states in our AI.
3. Create a field of the `EnemyState` type called `currentState`, which will hold the current state of our enemy:

```
public class EnemyFSM : MonoBehaviour
{
    public enum EnemyState { GoToBase, AttackBase, ChasePlayer, AttackPlayer }

    public EnemyState currentState;
}
```

Figure 9.27: EnemyFSM state definition

Info

For more information about how enums work, we recommend checking the following link:

<https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum>

4. Create three functions named after the states we defined.
5. Call those functions in `Update` depending on the current state:

```

void Update()
{
    if (currentState == EnemyState.GoToBase) { GoToBase(); }
    else if (currentState == EnemyState.AttackBase) { AttackBase(); }
    else if (currentState == EnemyState.ChasePlayer) { ChasePlayer(); }
    else { AttackPlayer(); }
}

void GoToBase() { print("GoToBase"); }
void AttackBase() { print("AttackBase"); }
void ChasePlayer() { print("ChasePlayer"); }
void AttackPlayer() { print("AttackPlayer"); }

```

Figure 9.28: If-based FSM

Yes, you can totally use a switch here, but I just prefer the regular `if` syntax for this example.

1. Test in the editor how changing the `currentState` field will change which state is active, seeing the messages being printed in the console:

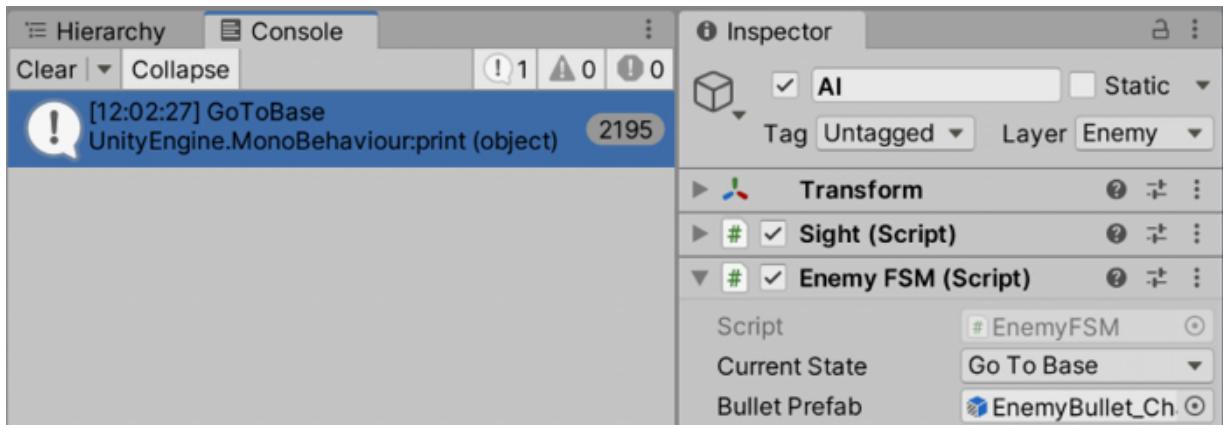


Figure 9.29: State testing

As you can see, it is a pretty simple but totally functional approach. In the future, you could face having to code enemies with many

more states, and this approach will start to scale badly. In such a case, you could use any FSM plugin of the Asset Store you prefer to have more powerful and scalable tools, or even consider advanced techniques like Behavior Trees, but that's outside the scope of this book. Now let's continue with this FSM, creating its transitions.

Creating transitions

If you remember the transitions created in the `Animator Controller`, those were basically a collection of conditions that are checked if the state the transition belongs to is active. In our FSM approach, this translates simply as `If` sentences that detect conditions inside the states. Let's create the transitions between our proposed states as follows:

1. Add a field of the `Sight` type called `sightSensor` in our FSM script, and drag the AI `GameObject` to that field to connect it to the `Sight` component there. As the FSM component is in the same object as `Sight`, we can also use `GetComponent` instead, but in advanced AIs, you might have different sensors that detect different objects, so I prefer to prepare my script for that scenario. You should pick the approach you like the most.
2. In the `GoToBase` function, check whether the detected object of the `Sight` component is not `null`, meaning that something is inside our line of vision. If our AI is going toward the base but detects an object in the way, we must switch to the `Chase` state to pursue the player, so we change the state, as in the following screenshot:

```
public Sight sightSensor;

void GoToBase()
{
    if (sightSensor.detectedObject != null)
    {
        currentState = EnemyState.ChasePlayer;
    }
}
```

Figure 9.30: Creating transitions

3. Also, we must change to `AttackBase` if we are near enough to the object that must be damaged to decrease the base life. We can create a field of the `Transform` type called `baseTransform` and drag the player's base life object we created previously there so we can check the distance. Remember to add a float field called `baseAttackDistance` to make that distance configurable:

```

public Transform baseTransform;
public float baseAttackDistance;

void GoToBase()
{
    if (sightSensor.detectedObject != null)
    {
        currentState = EnemyState.ChasePlayer;
    }

    float distanceToBase = Vector3.Distance(
        transform.position, baseTransform.position);

    if (distanceToBase < baseAttackDistance)
    {
        currentState = EnemyState.AttackBase;
    }
}

```

Figure 9.31: GoToBase transitions

4. In the case of `ChasePlayer`, we need to check whether the player is out of sight to switch back to the `GoToBase` state or whether we are near enough to the player to start attacking it. We will need another distance field called `PlayerAttackDistance`, which determines the distance to attack the player, and we might want different attack distances for those two targets. Consider an early return in the transition to prevent getting `null` reference exceptions if we try to access the position of the sensor detected object when there are not any:

```

public float playerAttackDistance;

void ChasePlayer()
{
    if (sightSensor.detectedObject == null)
    {
        currentState = EnemyState.GoToBase;
        return;
    }

    float distanceToPlayer = Vector3.Distance(transform.position,
        sightSensor.detectedObject.transform.position);

    if (distanceToPlayer <= playerAttackDistance)
    {
        currentState = EnemyState.AttackPlayer;
    }
}

```

Figure 9.32: ChasePlayer transitions

5. For `AttackPlayer`, we need to check whether the player is out of sight to get back to `GoToBase` or whether it is far enough to go back to chasing it. You will notice how we multiplied `playerAttackDistance` to make the stop-attacking distance a little bit greater than the start-attacking distance; this will prevent switching back and forth rapidly between attacking and chasing when the player is near that distance.

You can make it configurable instead of hardcoding 1.1 :

```

void AttackPlayer()
{
    if (sightSensor.detectedObject == null)
    {
        currentState = EnemyState.GoToBase;
        return;
    }

    float distanceToPlayer = Vector3.Distance(transform.position,
        sightSensor.detectedObject.transform.position);

    if (distanceToPlayer > playerAttackDistance * 1.1f)
    {
        currentState = EnemyState.ChasePlayer;
    }
}

```

Figure 9.33: AttackPlayer transitions

1. In our case, `AttackBase` won't have any transition. Once the enemy is near enough to the base to attack it, it will stay like that, even if the player starts shooting at it. Its only objective once there is to destroy the base.
2. Remember you can use `Gizmos` to draw the distances:

```

private void OnDrawGizmos()
{
    Gizmos.color = Color.blue;
    Gizmos.DrawWireSphere(transform.position, playerAttackDistance);

    Gizmos.color = Color.yellow;
    Gizmos.DrawWireSphere(transform.position, baseAttackDistance);
}

```

Figure 9.34: FSM Gizmos

3. Test the script by selecting the AI Object prior to clicking play and then move the player around, checking how the states change in the inspector. You can also keep the original `print` messages in each state to see them changing in the console. Remember to set the attack distances and the references to the objects. In the screenshot, you can see the settings we use:

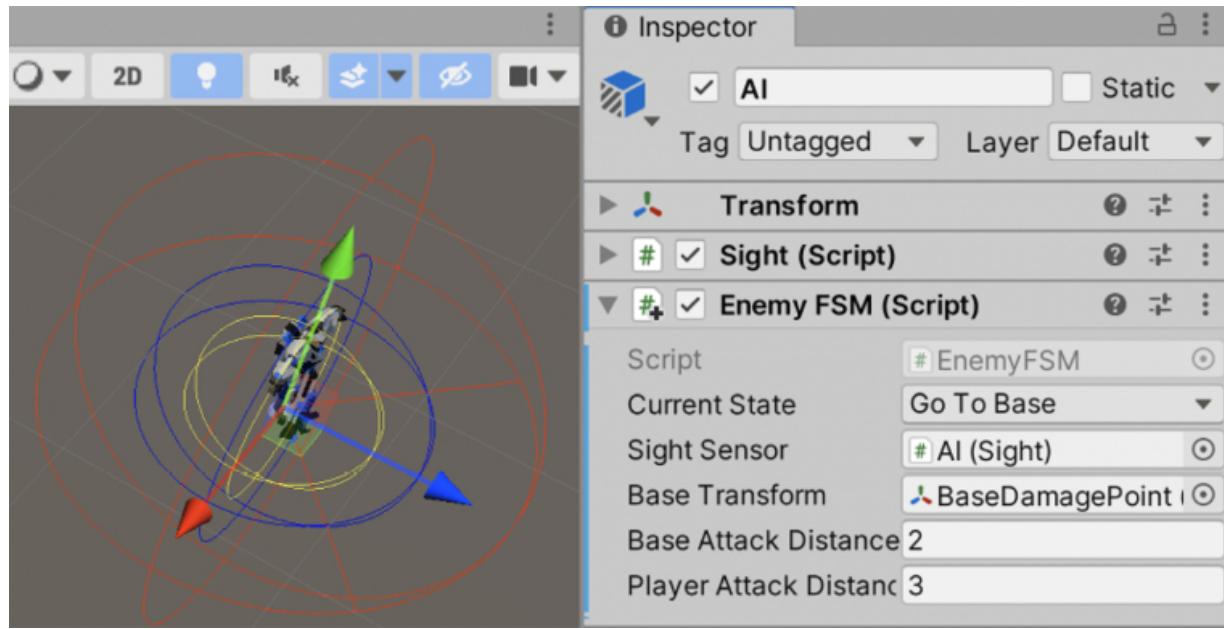


Figure 9.35: Enemy FSM settings

A little problem that we will have now is that the spawned enemies won't have the needed references to make the distance calculations

to the player's base transform. You will notice that if you try to apply the changes on the enemy of the scene to the Prefab (**Overrides -> Apply All**), the **Base Transform** variable will say `None`. Remember that Prefabs cannot contain references to objects in the scene, which complicates our work here. One alternative would be to create `BaseManager`, a Singleton that holds the reference to the damage position, so our `EnemyFSM` can access it. Another one could be to make use of functions such as

`GameObject.Find` to find our object. In this case, we will see the latter. Even though it can be less performant than the `Manager` version, I want to show you how to use it to expand your Unity toolset. In this case, just set the `baseTransform` field in `Awake` to the return of `GameObject.Find`, using `BaseDamagePoint` as the first parameter, which will look for an object with the same name, as in the following screenshot. You will see that now our wave-spawned enemies will change states:

```
private Transform baseTransform;  
  
private void Awake()  
{  
    baseTransform = GameObject.Find("BaseDamagePoint").transform;  
}
```

Figure 9.36: Searching for an object in the scene by name

Memory

When I started learning about AI for games, I thought I was going to create Skynet, using complex algorithms like deep learning. As you can see, we are far away from that, and the reason is that AI for games doesn't need to be intelligent, it needs to be fun. Making it so requires careful design to generate the exact desired experience, which

could be difficult to achieve with cutting edge AI technology.

Having said that, there are other AI techniques, like Behaviour Trees, which you can learn about it in this Halo developers' article:

<https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai>. Other alternative is called GOAP, and I recommend reading this paper from the F.E.A.R. developers:

https://alumni.media.mit.edu/~jorkin/gdc2006_orkin_jef_f_fear.pdf. Finally, there's the Game AI Pro book series which collects several game AI related papers, which you can learn more about it here: <http://www.gameapro.com/>

In this section we made our FSM to properly switch states according to the data provided by sensors and other sources, laying the foundation to start scripting the actual behaviour of each state. Now that our FSM states are coded and execute transitions properly, let's see how to do the same in Visual Scripting. Feel free to skip the following section if you are only interested in the C# version.

Creating the FSM in Visual Scripting

So far, most scripts in Visual Scripting were almost a mirror of the C# version with some differences in some nodes. While regarding state machines we could do the same, instead, we are going to use the **State Machine** system of Visual Scripting. The concept is the same, you have states and can switch them, but how the states are organized and when the transitions trigger is managed visually, in a similar way as the Animator system does. So, let's see how we can use the system by creating our first State Machine Graph and some states . Follow these steps:

1. Add the **State Machine** component to our enemy. Remember it is called **State Machine** and not **Script Machine**, the latter

being the component for regular Visual Scripts.

2. Click the **New** button in the component and select a place to save the `fixed` asset in a similar way to what we have done so far for regular Visual Scripts. In my case, I called it `EnemyFSM`.

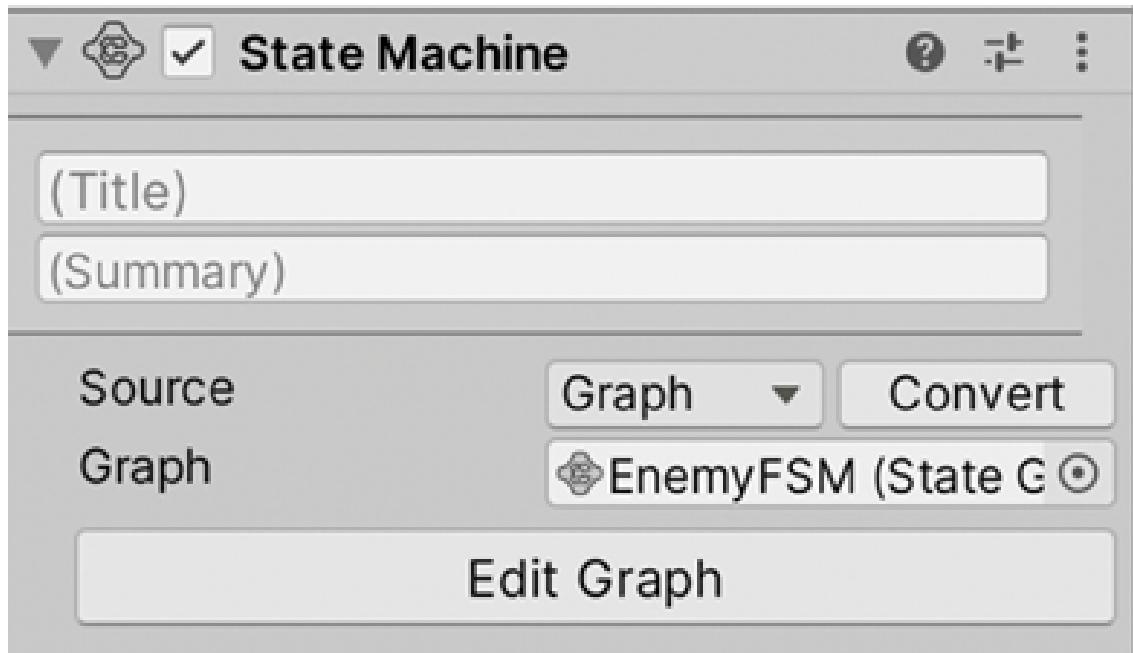


Figure 9.37: Creating a Visual State Machine

3. Double-click **State Machine Graph** to edit it as usual.
4. Right-click in any empty area of the **Graph** editor and select **Create Script State** in order to create a new state:

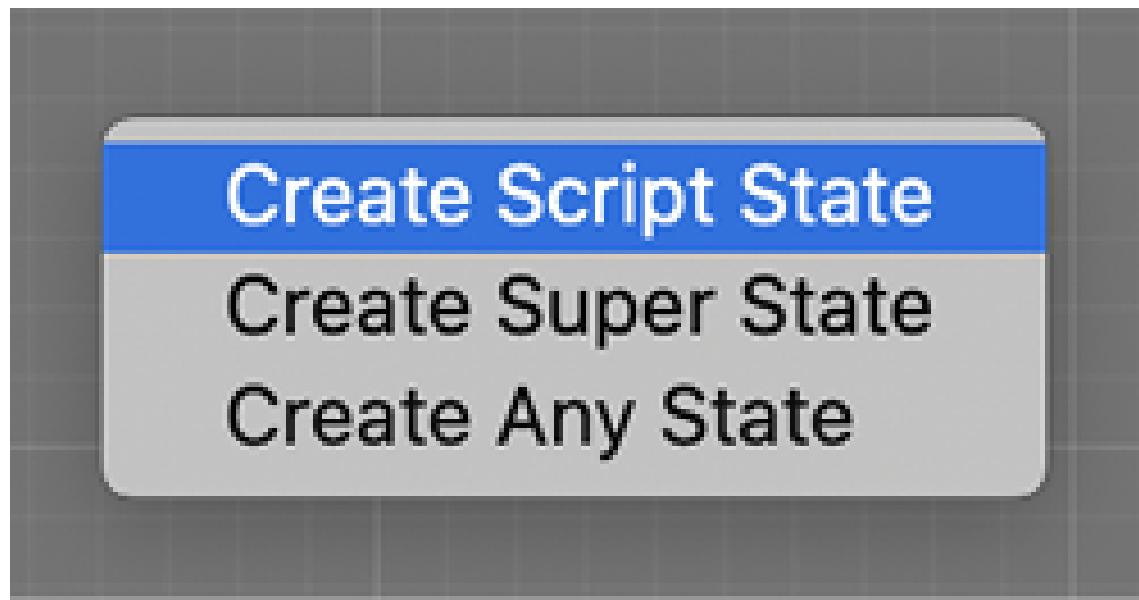


Figure 9.38: Creating our first Visual State Machine State

5. Repeat *step 4* until you end up having 4 states:

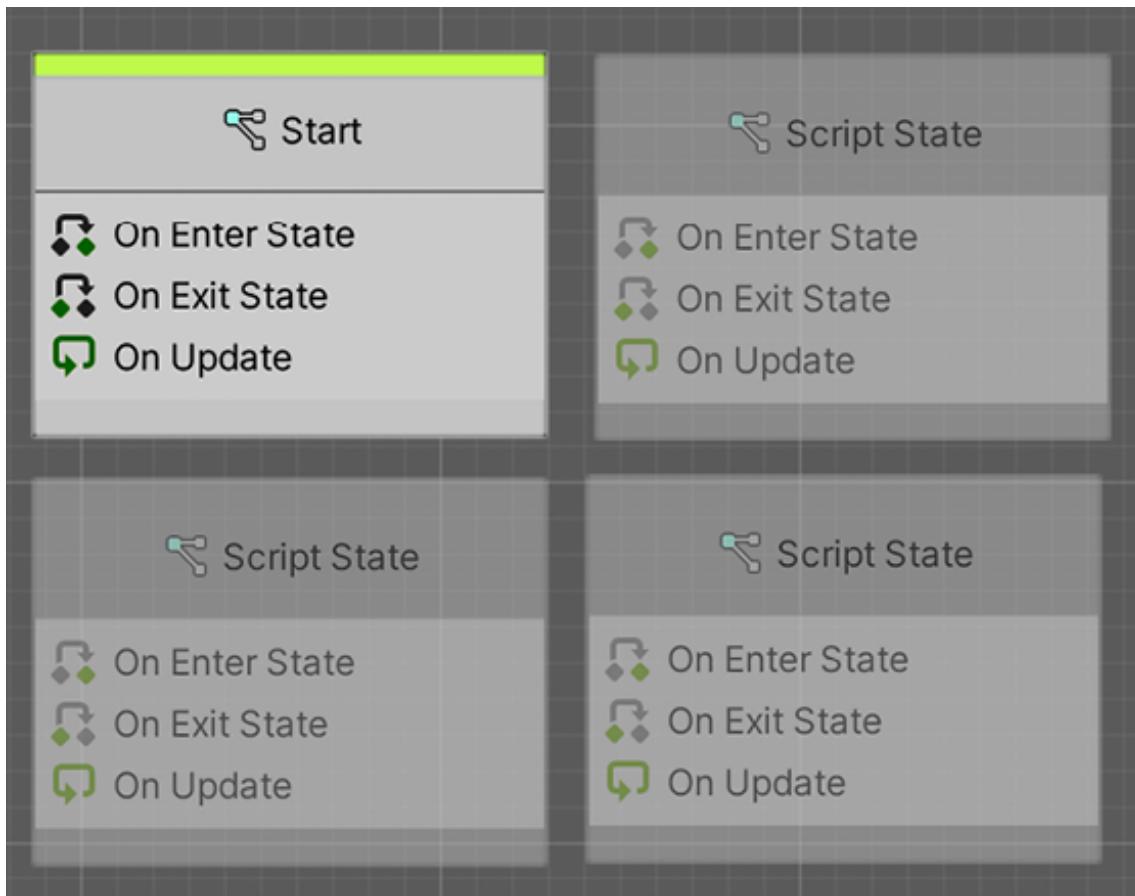


Figure 9.39: Visual states

6. Select any of them and in the **Info** panel on the left, fill the **Title** field (the first one) with the name of any of the states we created before (`GoToBase`, `AttackBase`, `ChasePlayer`, and `AttackPlayer`). If you don't see the **Info** panel, click the button with the **i** in the middle to display it:

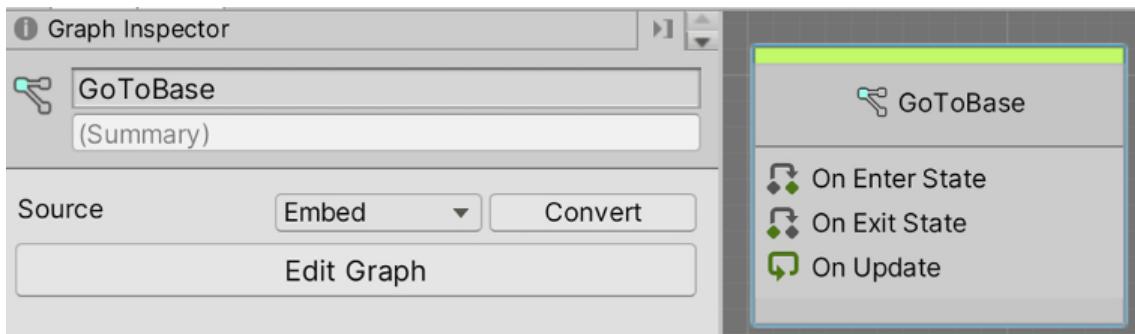


Figure 9.40: Renaming a Visual State

7. Repeat that for the rest of the state nodes until you have each node named after each state created in the *Creating the FSM in C#* section of this chapter:

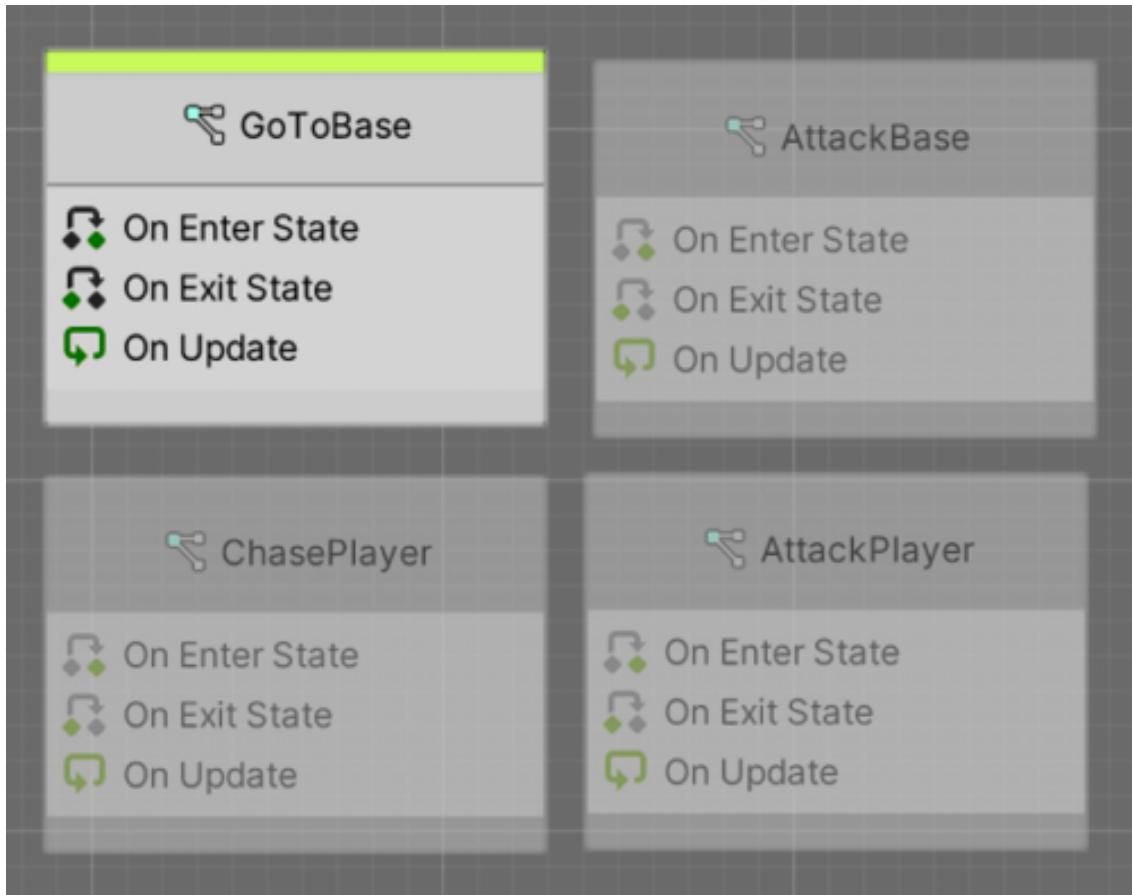


Figure 9.41: All needed states

8. You can see one of the states has a green bar at the top, which represents which node is supposed to be the first one. I renamed that initial state `GoToBase` as that's the one I prefer to be first. If you don't have that one as the starting one, right-click the node that currently has the green bar in your state machine, select **Toggle Start** to remove the green bar from it, and then repeat for the node that you want to be the first one (`GoToBase` in our scenario), adding the green bar to that one.

Something to consider is that you can have more than one start state in Visual Scripting, meaning you can have multiple states running at the same time and transitioning. If possible, I recommend avoiding having more than one state active at a time to make things simple.

1. Double-click `GoToBase` to enter the edit mode for these states. Connect a **String** node to the **print Message** input pin in the **OnUpdate** event node to print a message saying `GoToBase`:

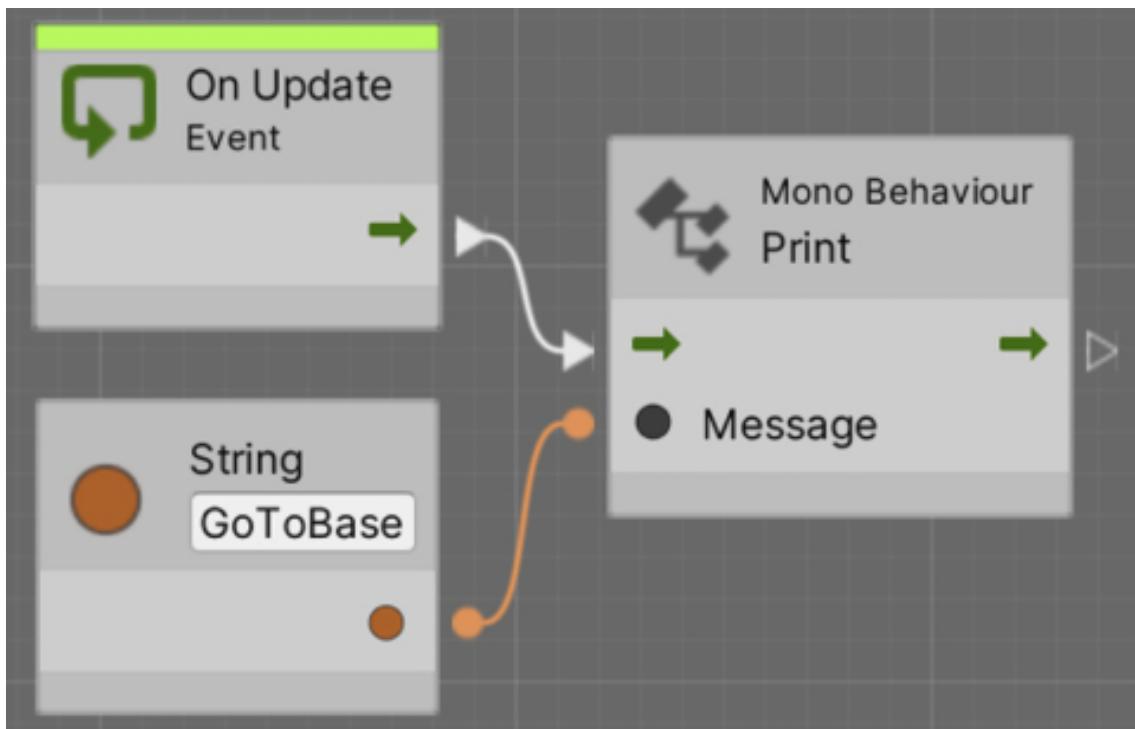


Figure 9.42: Our first state machine logic

2. In the top bar, click the **EnemyFSM** label at the left of **GoToBase** in order to return to the whole State Machine view. If you don't see it, click any text label at the right of the third button (the one that looks like `<x>`):

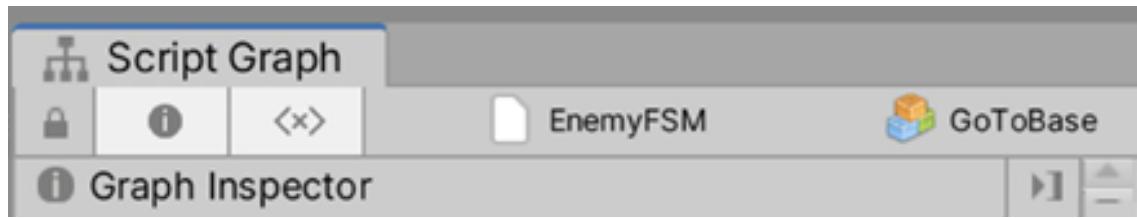


Figure 9.43: Returning to the State Machine editor mode

3. Feel free to delete the other event nodes if you are not planning to use them.
4. Repeat *steps 9-11* for each state until all of them print their names.

With this, we have created the nodes representing the possible states of our AI. In the next section, we will be adding logic for them to something meaningful, but before that, we need to create the transitions between the states and the conditions that need to be met to trigger them by doing the following:

1. Create variables in the **Variables** component of the enemy called `baseTransform`, `baseAttackDistance`, and `playerAttackDistance` as we are going to need them to do the transitions.
2. Don't set any type to `baseTransform` as we will fill it later via code, but regarding `baseAttackDistance`, make it using the **Float** type and put a value of `2`, and finally for `playerAttackDistance`, also use **Float** and a value of `3`. Feel free to change those values if you prefer:

=	Name	baseTransform	-
=	Type	(Null)	▼
	Name	baseAttackDistance	
=	Type	Float	▼ -
	Value	2	
	Name	playerAttackDistance	
=	Type	Float	▼ -
	Value	3	

Figure 9.44: Variables needed for our transitions

3. Right-click the `GoToBase` node and select the **Make Transition** option, and then click the `ChasePlayer` node. This will create a transition between the two states:



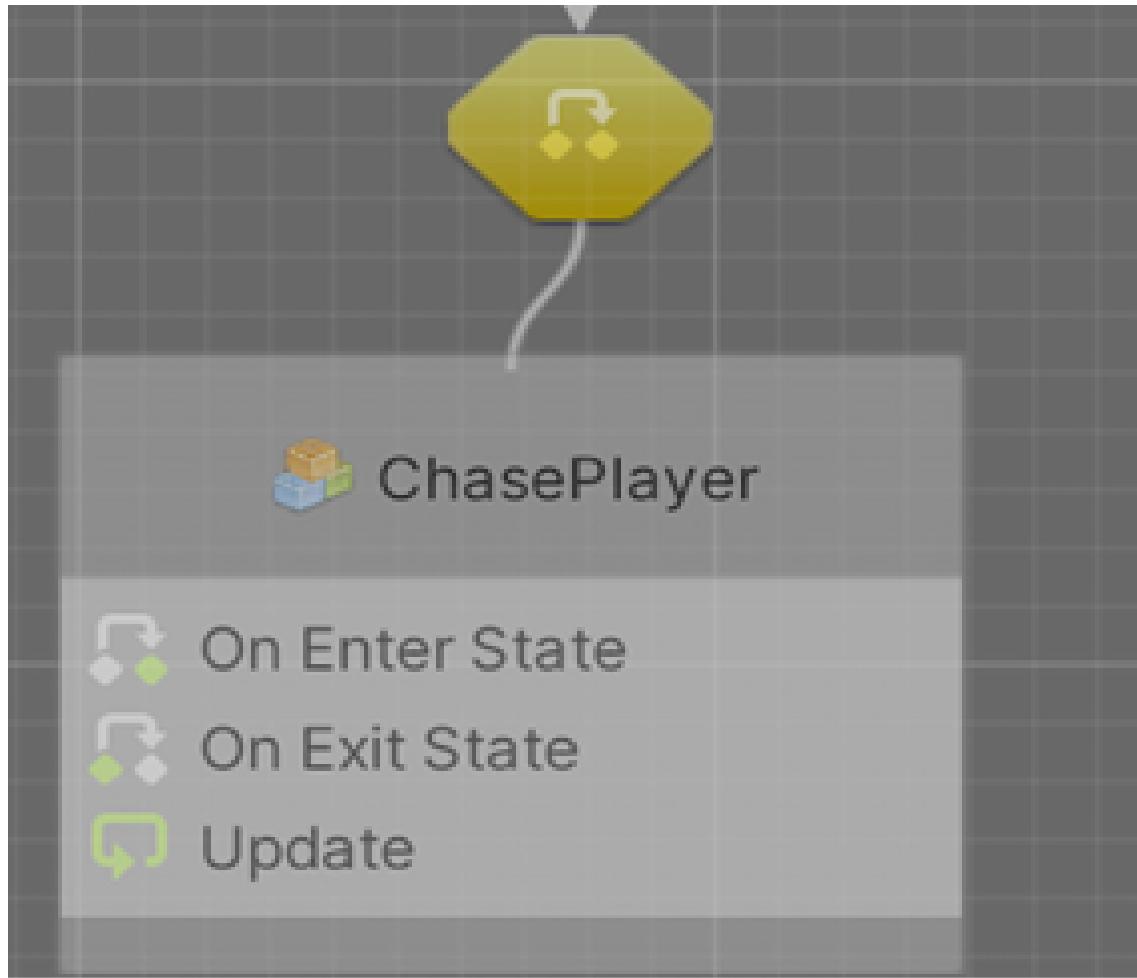


Figure 9.45: A transition between two states

4. Repeat *step 3* for each transition we created in the C# version.
The State Machine Graph will need to look like the following screenshot:

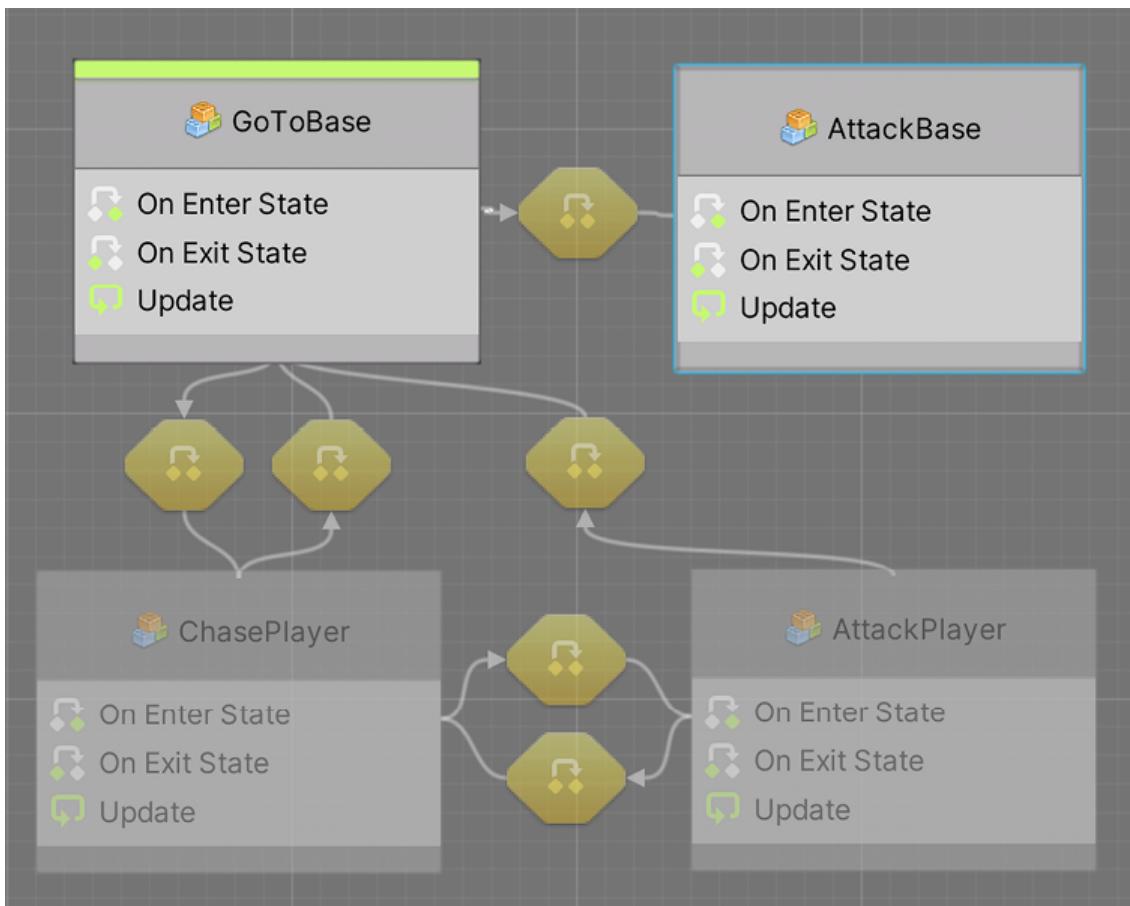


Figure 9.46: All the needed transitions

5. Double-click the yellow shape in the middle of the transition between **GoToBase** and **ChasePlayer** to enter the **Transition** mode. Here, you will be able to specify the condition that will trigger that transition (instead of using an `If` node during the state logic). Remember you have two yellow shapes, one for each transition direction, so check you are double-clicking the correct one based on the white arrows connecting them.
6. Modify the graph in order to check if the `sensedObject` variable is not `null`. It should look like this:

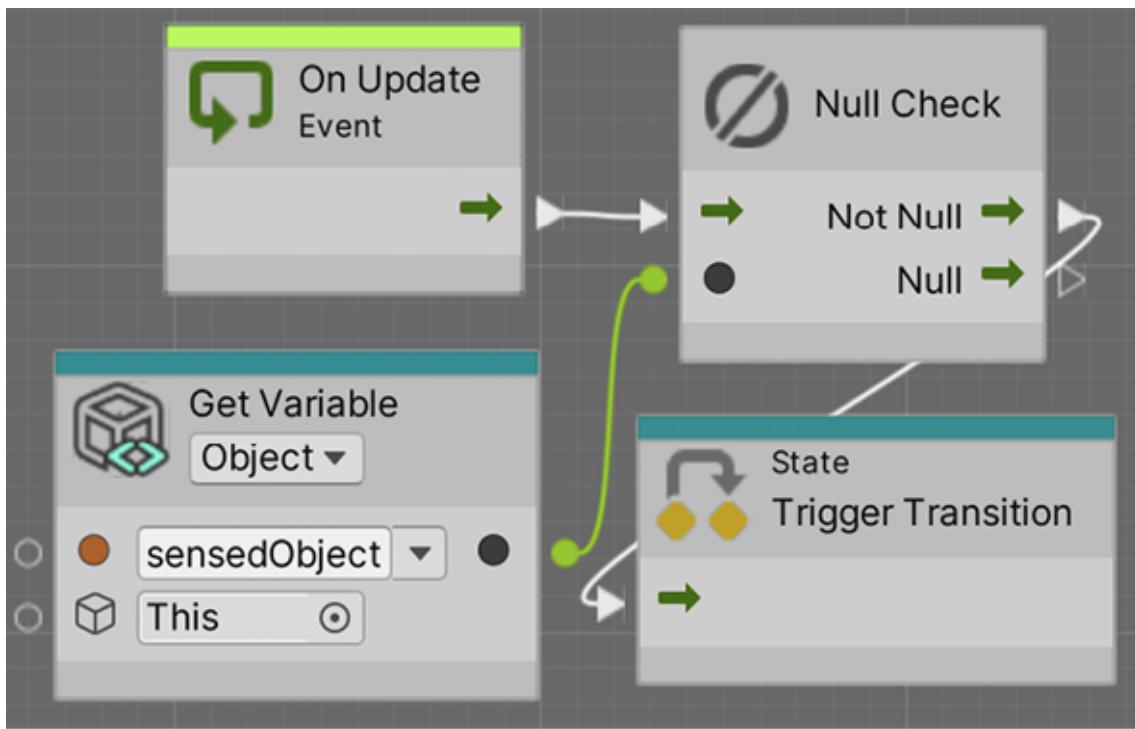


Figure 9.47: Adding a transition condition

7. The transition between **GoToBase** and **AttackBase** should look like this:

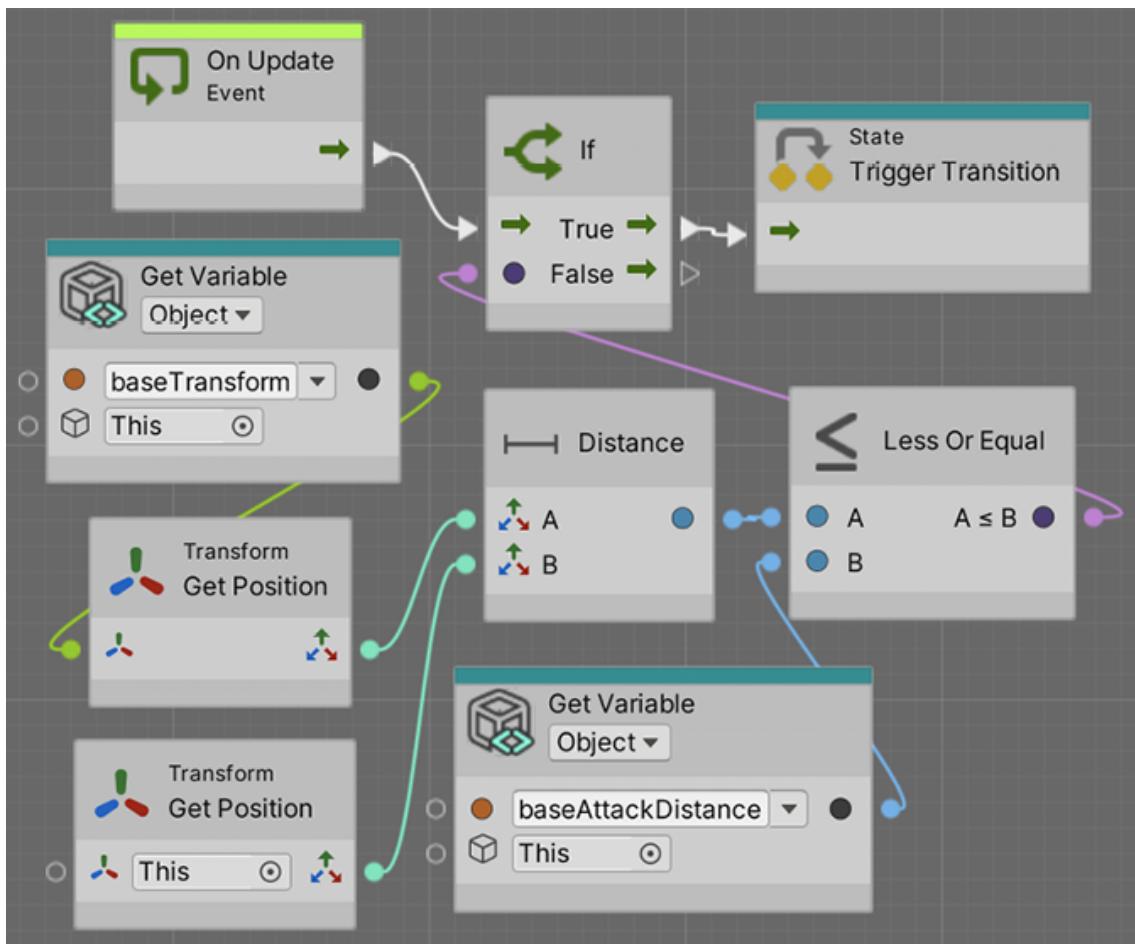


Figure 9.48: GoToBase to AttackBase transition condition

8. Now, ChasePlayer to GoToBase should be as follows:

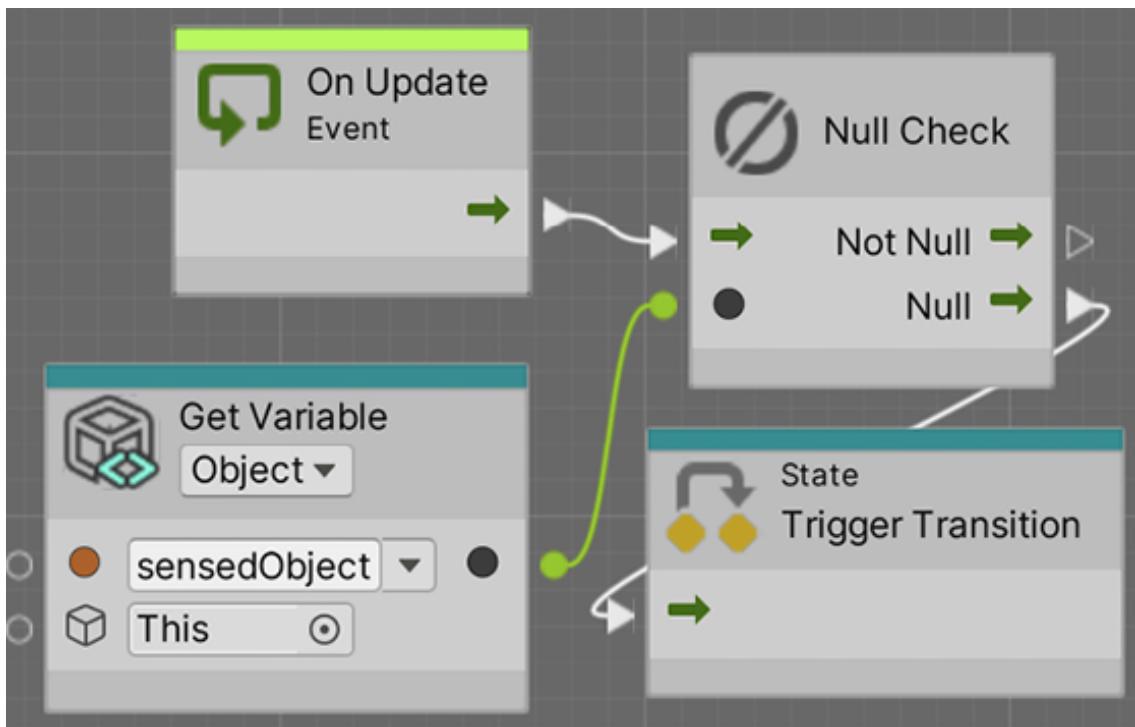


Figure 9.49: *ChasePlayer* to *GoToBase* transition condition

9. For the **ChasePlayer** to **AttackPlayer** transition, do as in *Figure 9.50*. This is essentially the same as **GoToBase** and **AttackBase**, a distance check, but with different targets:

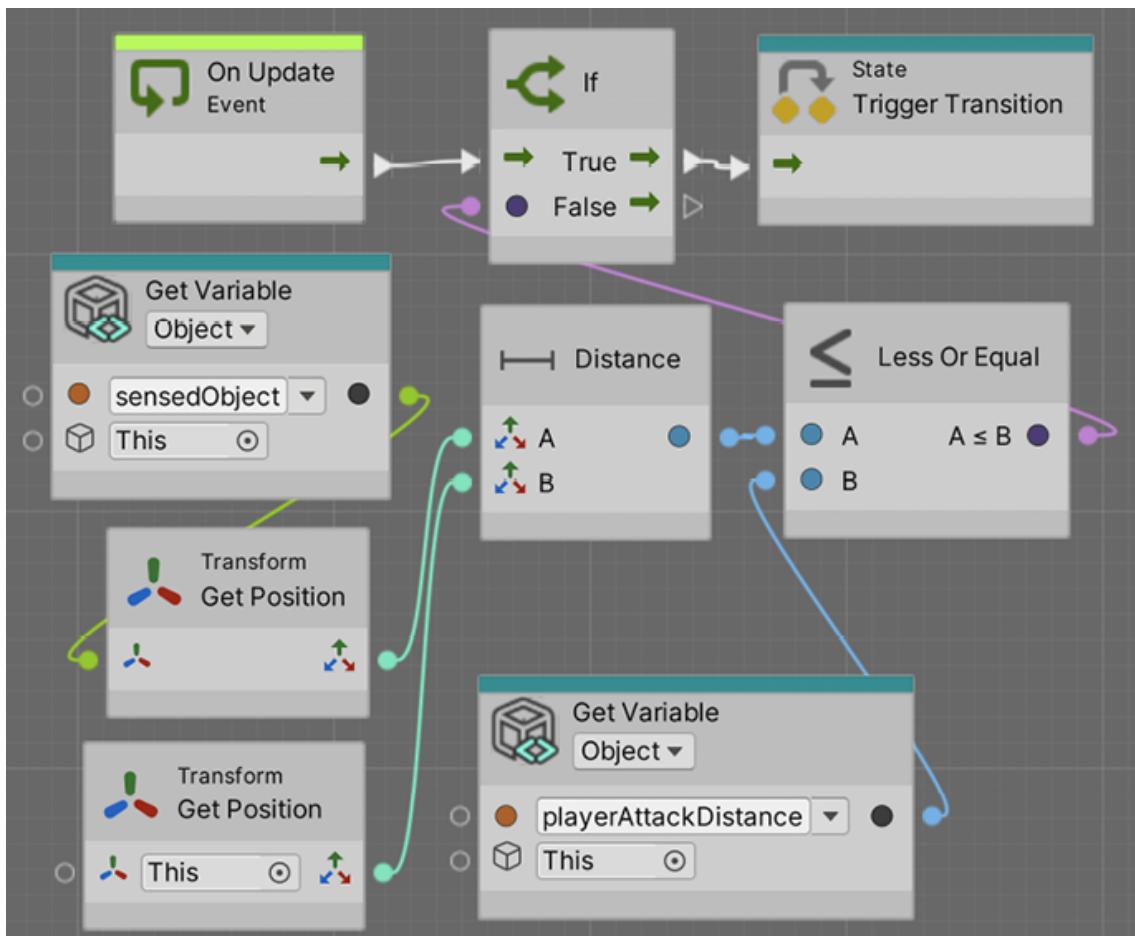


Figure 9.50: *ChasePlayer* to *AttackPlayer* transition condition

10. For the **AttackPlayer** to **ChasePlayer** transition, do as in *Figure 9.51*. This is another distance check but is now checking if the distance is greater and multiplying the distance by 1.1 (to prevent transition jittering as we explained in the C# version):

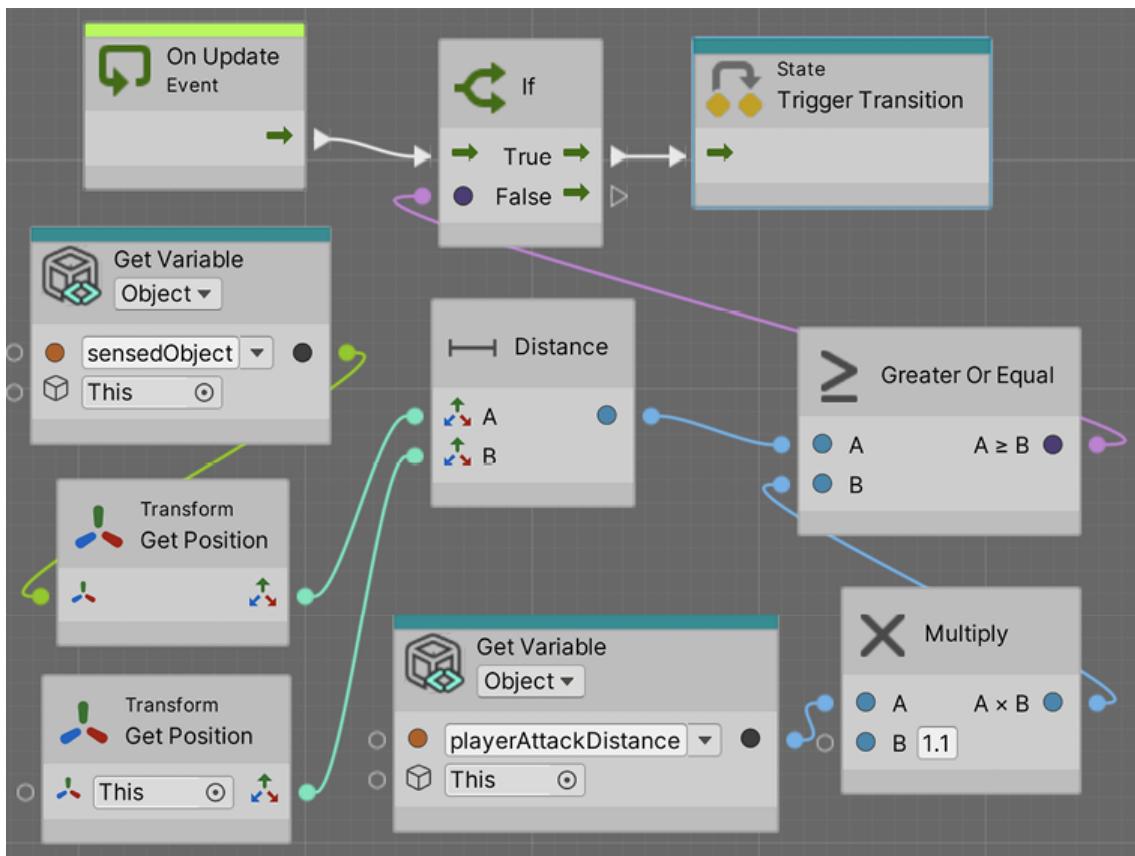


Figure 9.51: *AttackPlayer* to *ChasePlayer* transition condition

11. Finally, for **AttackPlayer** to **GoToBase** this is the expected graph:

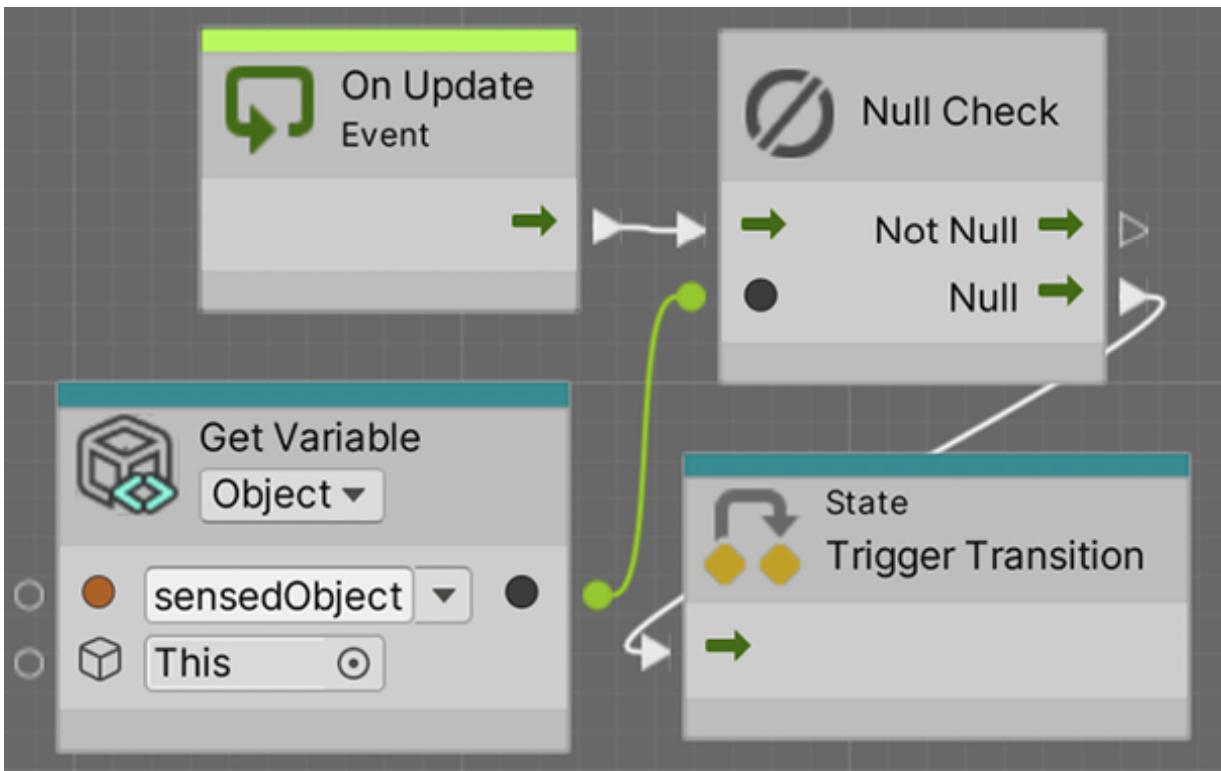


Figure 9.52: AttackPlayer to GoToBase transition condition

A little detail we need to tackle before moving on is the fact that we still don't have any value set in the `baseTransform` variable. The idea is to fill it via code as we did in the C# version. But something to consider here is that we cannot add an `Awake` event node to the whole state machine, but just to the states. In this scenario, we could use the **OnEnterState** event, which is an exclusive event node for state machines. It will execute as soon as the state becomes active, which is useful for state initializations. We could add the logic to initialize the `baseTransform` variable in the **OnEnterState** event node of the **GoToBase** state, given it is the first state we execute. This way, **GoToBase** logic will look as in *Figure 9.53*. Remember to double-click the state node to edit it:

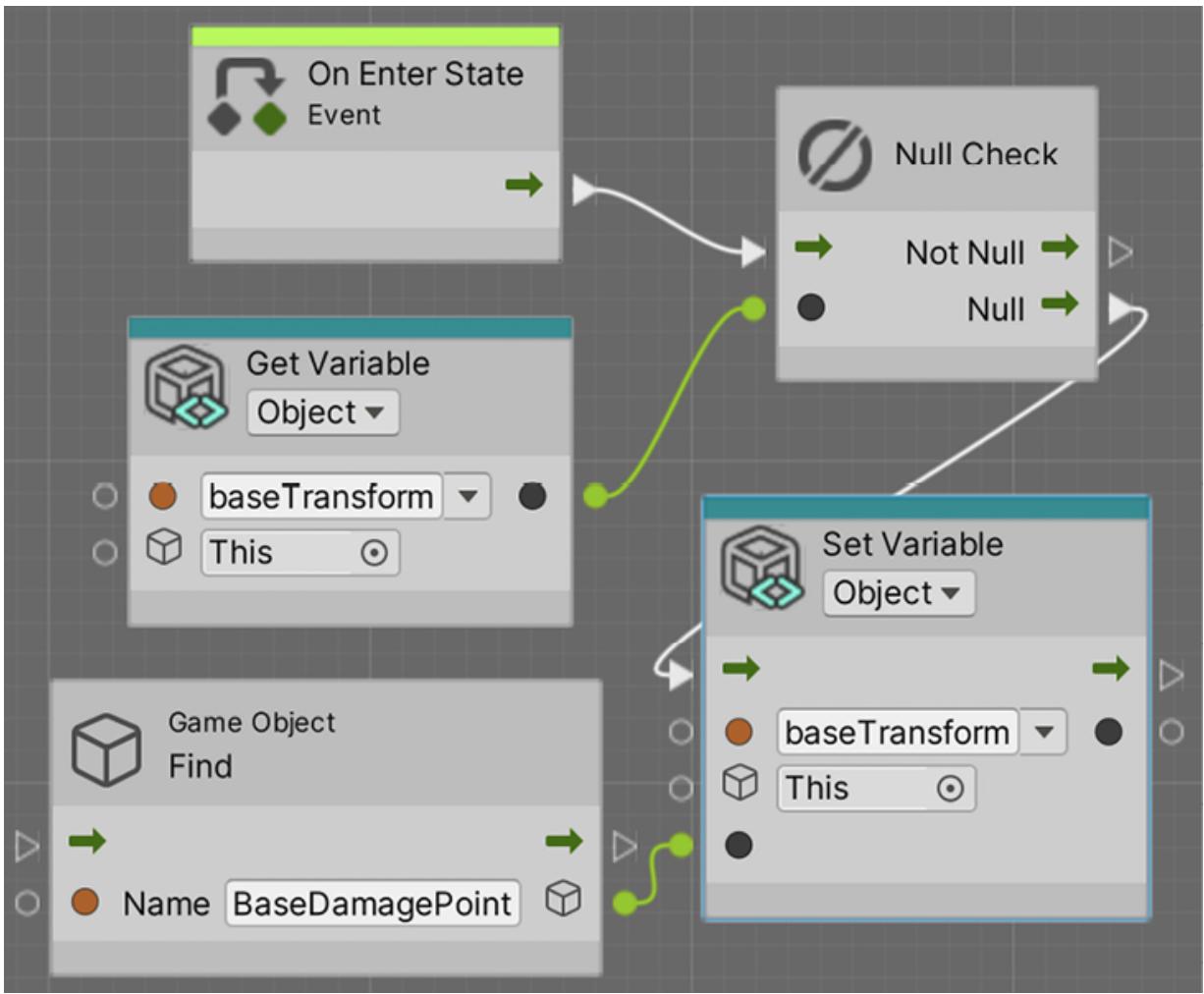


Figure 9.53: GoToBase initialization logic

Notice how, here, we set the result of the **Find** node into the variable only on the **Null** pin of **Null Check**. What **Null Check** does is check if our **baseTransform** variable is set, going through the `Not Null` pin if it is, and `Null` if it isn't. This way we avoid executing **GameObject.Find** every time we enter the **GoToBase** state, but only the first time. Also, note that in this case, we will be executing the **Set Variable** node not only when the object initializes, but also each time **GoToBase** becomes the current state. If, in any case, that results in unexpected behavior, other options could be to create a new initial state that initializes everything and then transitions to the rest of the states, or maybe do a classic Visual Script graph that initializes those variables in the **On Start**

event node. With all this, we learned how to create a decision-making system for our AI through FSMs. It will make decisions based on the info gathered via sensors and other systems. Now that our FSM states are coded and transition properly, let's make them do something.

Executing FSM actions

Now we need to complete the last step—make the FSM do something interesting. Here, we can do a lot of things such as shoot the base or the player and move the enemy toward its target (the base or the player). We will be handling movement with the Unity Pathfinding system called `NavMesh`, a tool that allows our AI to calculate and traverse paths between two points while avoiding obstacles, which needs some preparation to work properly. In this section, we will examine the following FSM action concepts:

- Calculating our scene's NavMesh
- Using Pathfinding
- Adding final details

Let's start by preparing our scene for movement with Pathfinding.

Calculating our scene's NavMesh

Pathfinding algorithms rely on simplified versions of the scene. Analyzing the full geometry of a complex scene is almost impossible to do in real time. There are several ways to represent Pathfinding information extracted from a scene, such as Graphs and `NavMesh` geometries. Unity uses the latter—a simplified mesh similar to a 3D model that spans all areas that Unity determines are walkable. In the next screenshot, you can find an example of `NavMesh` generated in a scene, that is, the light blue geometry:

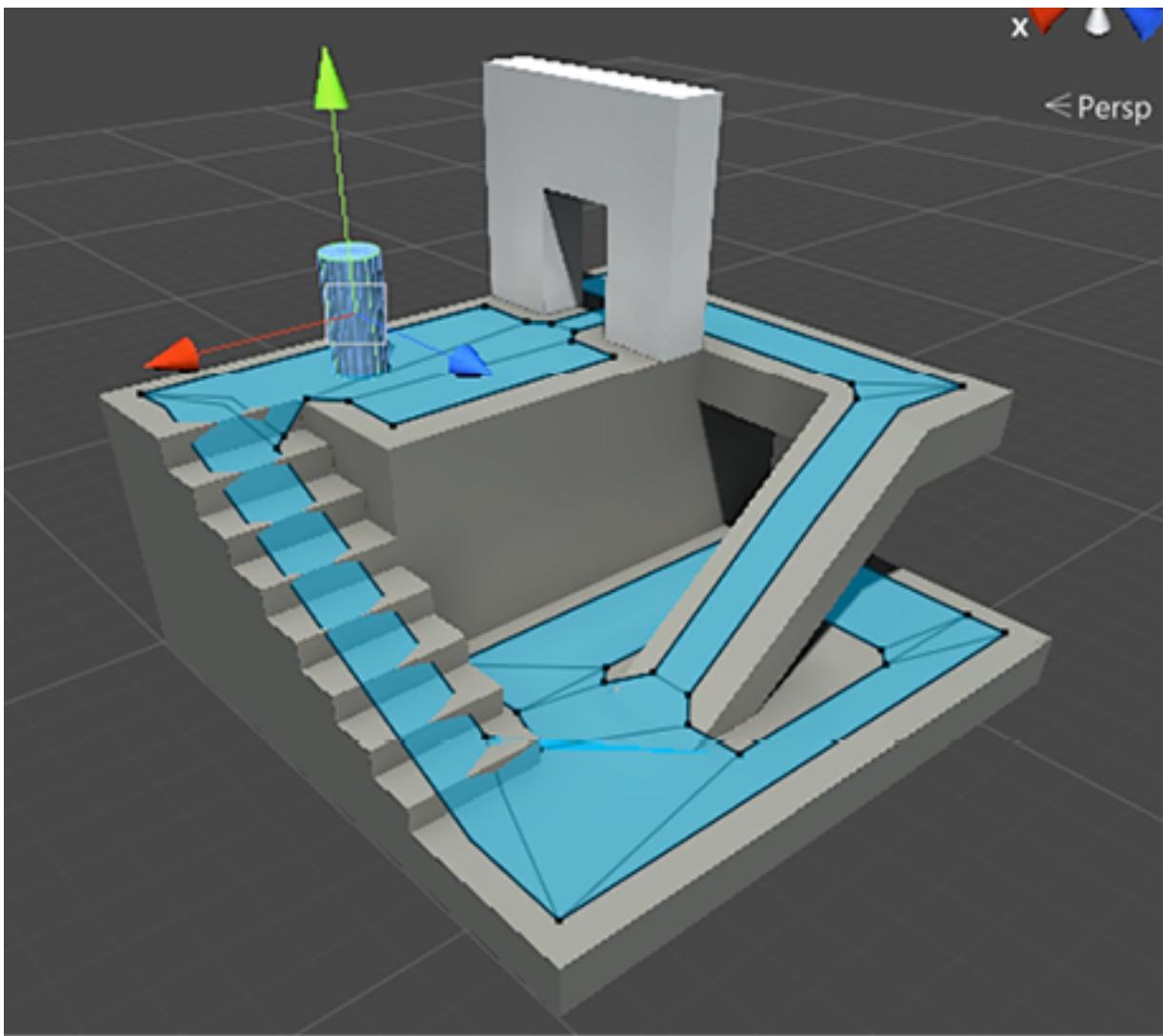


Figure 9.54: NavMesh of walkable areas in the scene

Generating `NavMesh` can take from seconds to minutes depending on the size of the scene. That's why Unity's Pathfinding system calculates the `NavMesh` once in the editor, so when we distribute our game, the user will use the pre-generated `NavMesh`. In previous Unity versions, like Lightmapping, `NavMesh` used to be baked into a file for later usage. That meant that Game Objects that contributed to the `NavMesh` surface used to be static, and that they couldn't suffer any modifications of the scene during runtime. The main advantage of the new AI Navigation system in Unity is that `NavMesh` objects can now change during runtime. If you destroy or move a

floor tile, the AI will still adapt its behaviour to walk, stay or fall over that area. This means if a floor tile is destroyed during gameplay, the NavMesh dynamically updates to reflect this change, showing the AI where it can no longer walk.. We will install and use the AINavigation package to add this behaviour to our game. To generate `NavMesh` for our scene, do the following:

1. Open Package Manager (Window | Package Manager).
2. Set the Packages dropdown to Unity Registry mode:
3. Search the list for a package called AI Navigation. This package will allow us to have access to new components that will help us define which surfaces are walkable and which agents can walk on top of them. At the moment of writing the book, the current version of this package is 1.1.4

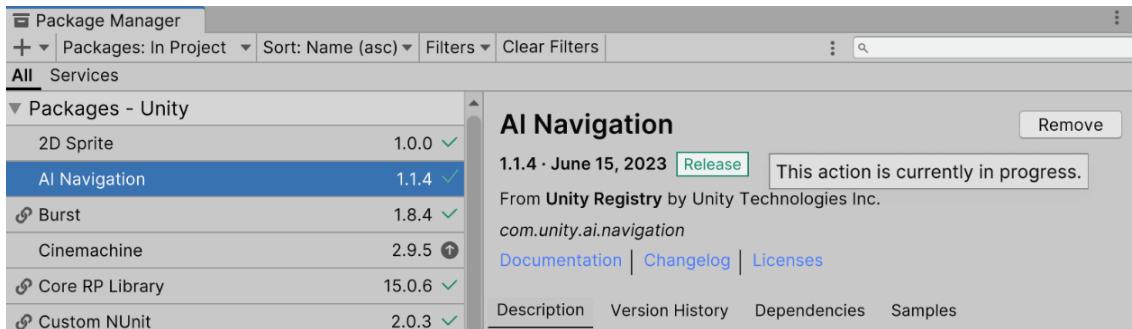


Figure 9.55: Installing the AI Navigation Package

4. Add a `NavMeshSurface` component to the walkable surface
5. From the recently added component, click on the **Bake** button at the bottom of the window, and check the generated `NavMesh`:

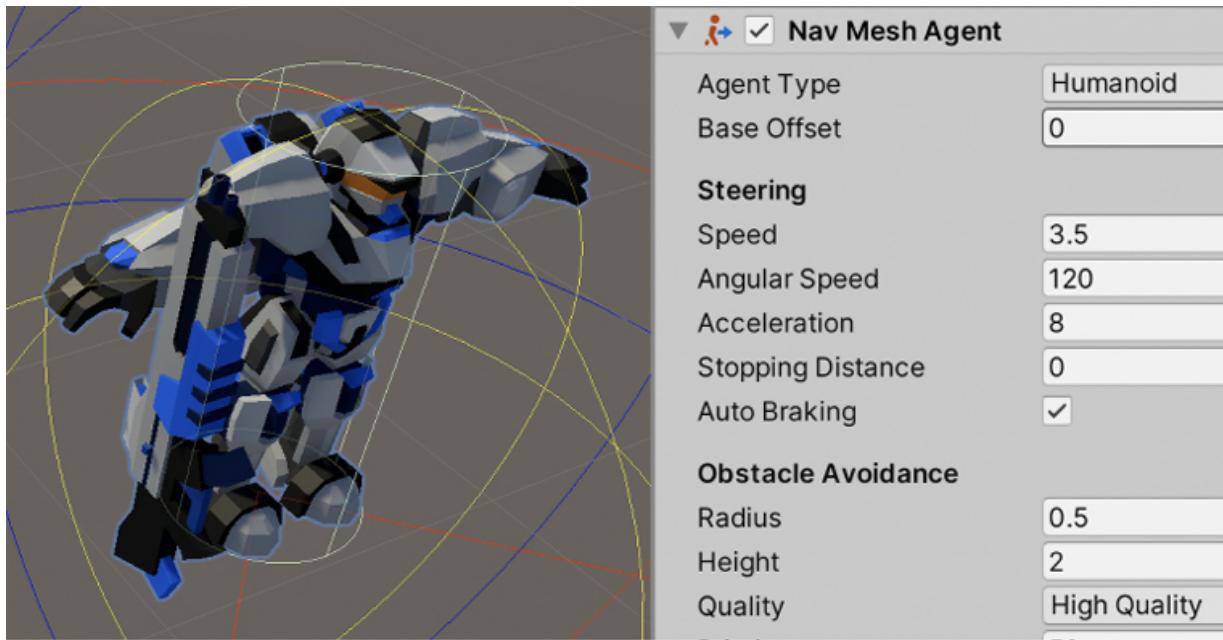


Figure 9.56: Generating a NavMesh

And that's pretty much everything you need to do. Of course, there are lots of settings you can fiddle around with in this component, such as **Max Slope**, which indicates the maximum angle of slopes the AI will be able to climb, or **Step Height**, which will determine whether the AI can climb stairs, connecting the floors between the steps in `NavMesh`, but as we have a plain and simple scene, the default settings will suffice. If you want to play around with them, you can go to the menu bar and select `Window > AI > Navigation`. From there, you will be able to adjust all these parameters and re-bake the `NavMeshSurface` to adjust the walkable areas depending on the size of the AI agents. With our scene's NavMesh set up, we've laid the groundwork for sophisticated AI movement. Let's see this in action as we program our AI to navigate the game world.

Using Pathfinding

For making an AI object that moves with `NavMesh`, Unity provides the `NavMeshAgent` component, which will make our AI stick to `NavMesh`, preventing the object from going outside it. It will not

only calculate the path to a specified destination automatically but also will move the object through the path with the use of Steering behavior algorithms that mimic the way a human would move through the path, slowing down on corners and turning with interpolations instead of instantaneously. This component also ensures AI characters avoid each other. It prevents crowding by steering each character away from others, maintaining a natural flow in the game. Let's use this powerful component by doing the following:

1. Select the **Enemy** Prefab and add the `NavMeshAgent` component to it. Add it to the root object, the one called `Enemy`, not the AI child—we want the whole object to move. You will see a cylinder around the object representing the area the object will occupy in `NavMesh`. Note that this isn't a collider, so it won't be used for physical collisions:

```
private NavMeshAgent agent;

private void Awake()
{
    baseTransform = GameObject.Find("BaseDamagePoint").transform;
    agent = GetComponentInParent<NavMeshAgent>();
}
```

Figure 9.57: The NavMeshAgent component

2. Remove the `ForwardMovement` component; from now on, we will drive the movement of our enemy with `NavMeshAgent`.
3. In the `Awake` event function of the `EnemyFSM` script, use the `GetComponentInParent` function to cache the reference of `NavMeshAgent` into a new private variable. This will work similarly to `GetComponent`—it will look for a component in our `GameObject`, but if the component is not there, this version will try to look for that component in all parents. Remember to add

the `using UnityEngine.AI` line to use the `NavMeshAgent` class in this script:

```
void GoToBase()
{
    agent.SetDestination(baseTransform.position);
```

Figure 9.58: Caching a parent component reference

As you can imagine, there is also the `GetComponentInChildren` method, which searches components in `GameObject` first and then in all its children if necessary.

1. In the `GoToBase` state function, call the `SetDestination` function of the `NavMeshAgent` reference, passing the position of the base object as the target:

```
void AttackBase()
{
    agent.isStopped = true;
}
```

Figure 9.59: Setting a destination for our AI

2. Save the script and test this with a few enemies in the scene or with the enemies spawned by the waves. You will see the problem where the enemies will never stop going toward the target position, entering inside the object, if necessary, even if the current state of their FSMs changes when they are near enough. That's because we never tell `NavMeshAgent` to stop,

which we can do by setting the `isStopped` field of the agent to `true`.

You might want to tweak the base attack distance to make the enemy stop a little bit closer or further away:

```
void GoToBase()
{
    agent.isStopped = false;
    agent.SetDestination(baseTransform.position);
```

Figure 9.60: Stopping agent movement

1. We can do the same for `ChasePlayer` and `AttackPlayer`. In `ChasePlayer`, we can set the destination of the agent to the player's position, and in `AttackPlayer`, we can stop the movement. In this scenario, Attack Player can go back again to `GoToBase` or `ChasePlayer`, so you need to set the `isStopped` agent field to `false` in those states or before doing the transition. We will pick the former, as that version will cover other states that also stop the agent without extra code. We will start with the `GoToBase` state:

```
void ChasePlayer()
{
    agent.isStopped = false;

    if (sightSensor.detectedObject == null)
    {
        currentState = EnemyState.GoToBase;
        return;
    }

    agent.SetDestination(sightSensor.detectedObject.transform.position);
```

Figure 9.61: Reactivating the agent

2. Then, continue with `ChasePlayer`:

```
void AttackPlayer()
{
    agent.isStopped = true;
```

Figure 9.62: Reactivating the agent and chasing the player

3. And finally, continue with `AttackPlayer`:

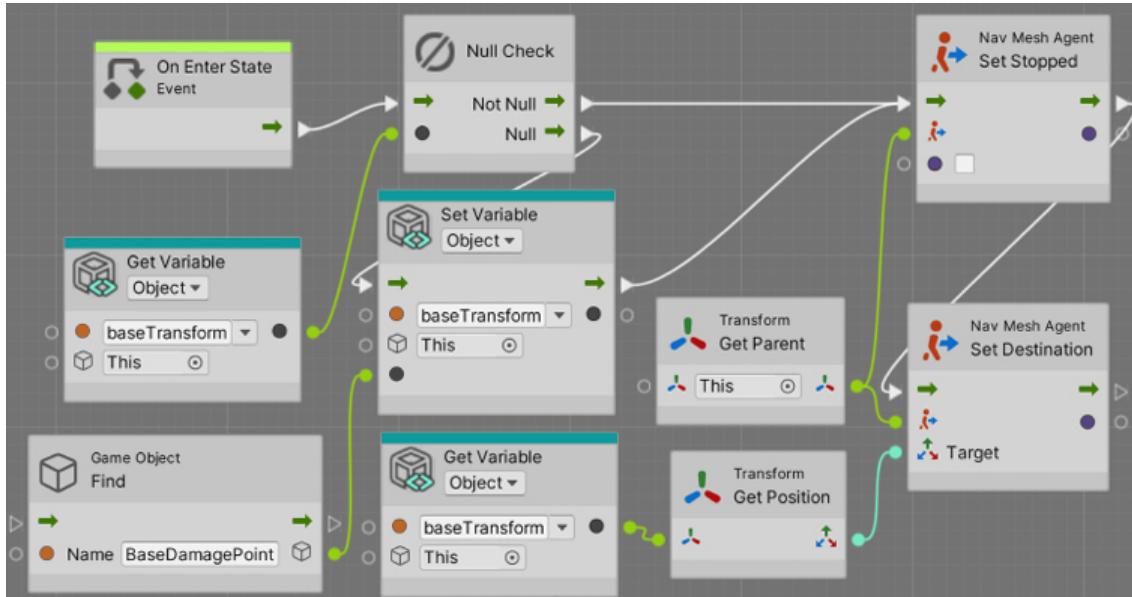


Figure 9.63: Stopping the movement

4. You can tweak the **Acceleration**, **Speed**, and **Angular Speed** properties of `NavMeshAgent` to control how fast the enemy will move. Balance these settings to make sure the AI moves in a way that makes sense in your game. Also, remember to apply

the changes to the Prefab for the spawned enemies to be affected.

5. Regarding the Visual Scripting versions, `GoToBase` will look like the following screenshot:

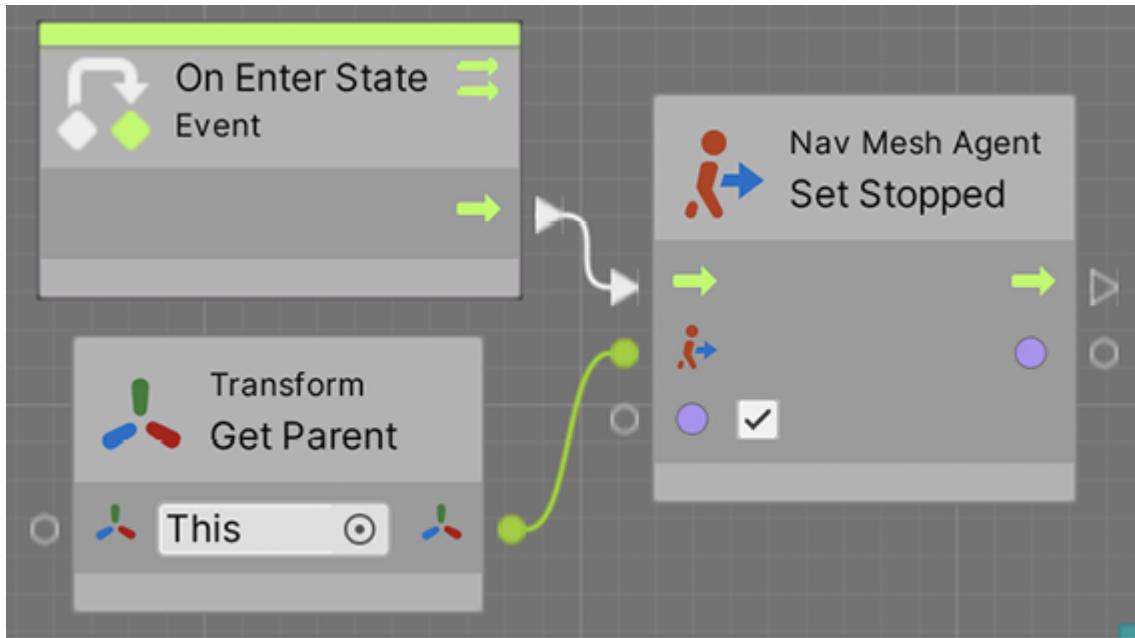


Figure 9.64: Making our agent move

6. We deleted the **OnUpdate** event node printing a message as we don't need it anymore. Also, we called the **Set Destination** node after setting the variable if `if` was `null`, and also when the variable wasn't `null` (**Not Null** pin of **Null Check**). Note that all of this happens in the **On Enter State** event, so we just need to do it once. We do it every frame in the C# version for simplicity but that's actually not necessary, so we will take advantage of the **OnEnterState** event. We can emulate that behavior in the C# version if we want, executing these actions at the moment we change the state (inside the `If` statements that check the transition conditions), instead of using the **Update** function. Finally, notice how we needed to use the **GetParent** node in order to access the `NavMeshAgent`

component in the enemy's root object? This is needed because we are currently in the **AI** child object instead.

7. Now, the AttackBase state will look like this:

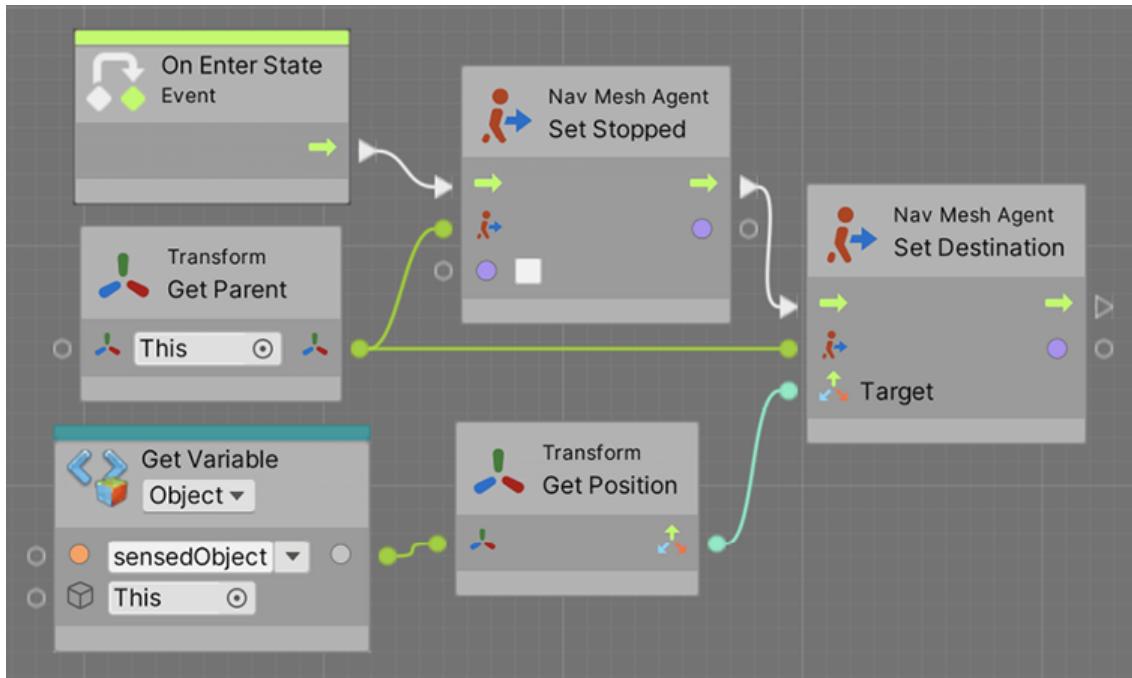


Figure 9.65: Making our agent stop

8. The ChasePlayer state will look like this:

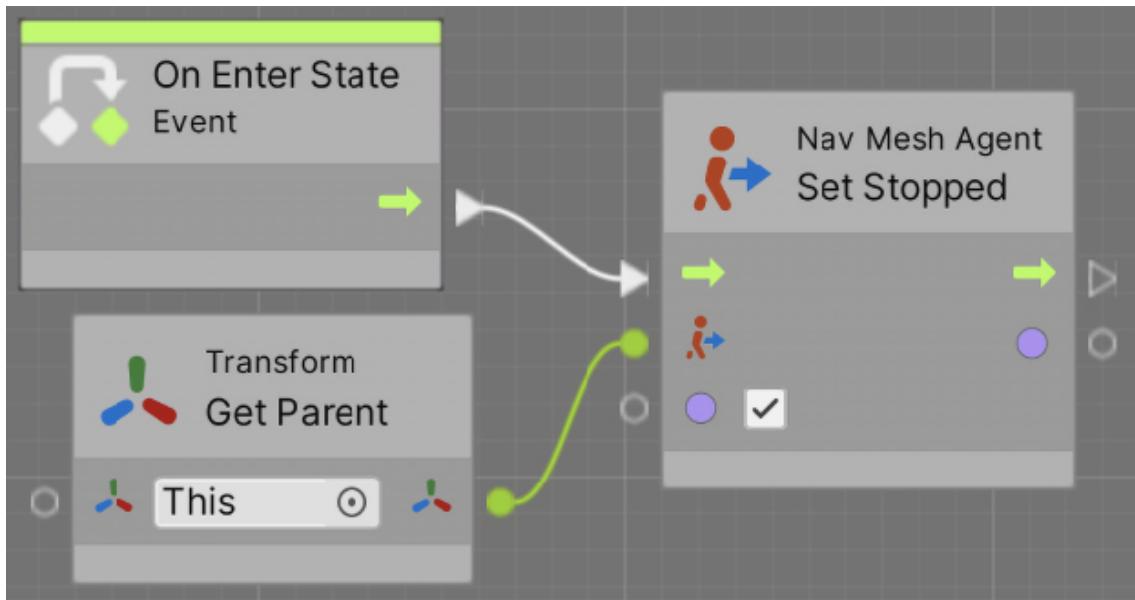


Figure 9.66: *ChasePlayer* logic

9. And finally, *AttackPlayer* like this:

```

void AttackPlayer()
{
    agent.isStopped = true;

    if (sightSensor.detectedObject == null)
    {
        currentState = EnemyState.GoToBase;
        return;
    }

    Shoot();

    float distanceToPlayer = Vector3.Distance(transform.position,
                                                sightSensor.detectedObject.transform.position);

    if (distanceToPlayer > playerAttackDistance * 1.1f)
    {
        currentState = EnemyState.ChasePlayer;
    }
}

void AttackBase()
{
    agent.isStopped = true;
    Shoot();
}

void Shoot()
{
}

```

Figure 9.67: AttackPlayer logic

Tip

While Unity has its own pathfinding system, it is not the only one, and it might not suit more advanced games. I recommend learning the basics of pathfinding, like learning about the BFS, Dijkstra and A* algorithms. If you want to deep dive, you can learn more advanced techniques, like the tactical pathfinding explained in this Killzone's developer's paper:

http://cse.unl.edu/~choueiry/Documents/straatman_remco_killzone_ai.pdf, or this presentation of Left 4 Dead AI systems: https://steamcdn-a.akamaihd.net/apps/valve/2009/ai_systems_of_l4d_mike_booth.pdf

With our AI now capable of navigating the game world, we're close to having a fully functional enemy. Next, we'll add the finishing touches, including shooting mechanics and animations, to complete our AI's behavior.

Adding the final details

We have two things missing here: the enemy is not shooting any bullets, and it doesn't have animations. Let's start with fixing the shooting by doing the following:

1. Add a `bulletPrefab` field of the `GameObject` type to our `EnemyFSM` script and a `float` field called `fireRate`.
2. Create a function called `Shoot` and call it inside `AttackBase` and `AttackPlayer`:

```

public float lastShootTime;
public GameObject bulletPrefab;
public float fireRate;

void Shoot()
{
    var timeSinceLastShoot = Time.time - lastShootTime;
    if (timeSinceLastShoot > fireRate)
    {
        lastShootTime = Time.time;
        Instantiate(bulletPrefab,
                    transform.position, transform.rotation);
    }
}

```

Figure 9.68: Shooting function calls

3. In the `Shoot` function, put similar code as that used in the `PlayerShooting` script to shoot bullets at a specific fire rate, as in *Figure 9.68*. Remember to set the **Enemy** layer in your **Enemy** Prefab, if you didn't before, to prevent the bullet from damaging the enemy itself. You might also want to raise the AI `GameObject` position a little bit to shoot bullets from a position other than the ground or, better, add a `shootPoint` `transform` field and create an empty object in the enemy to use as a spawn position. If you do that, consider making the empty object not be rotated so the enemy rotation affects the direction of the bullet properly:

```

        LookTo(sightSensor.detectedObject.transform.position);
        Shoot();

        float distanceToPlayer = Vector3.Distance(transform.position,
            sightSensor.detectedObject.transform.position);

        if (distanceToPlayer > playerAttackDistance * 1.1f)
        {
            currentState = EnemyState.ChasePlayer;
        }
    }

    void AttackBase()
    {
        agent.isStopped = true;
        LookTo(baseTransform.position);
        Shoot();
    }

    void LookTo(Vector3 targetPosition)
    {
    }

```

Figure 9.69: Shoot function code

Here, you find some duplicated shooting behavior between `PlayerShooting` and `EnemyFSM`. You can fix that by creating a **Weapon** behavior with a function called `Shoot` that instantiates bullets and takes into account the fire rate and call it inside both components to re-utilize it.

1. When the agent is stopped, not only does the movement stop but also the rotation. If the player moves while the enemy is

being attacked, we still need the enemy to face the player to shoot bullets in its direction. We can create a `LookTo` function that receives the target position to look at and call it in `AttackPlayer` and `AttackBase`, passing the target to shoot at:

```
void LookTo(Vector3 targetPosition)
{
    Vector3 directionToPosition = Vector3.Normalize(
        targetPosition - transform.parent.position);
    directionToPosition.y = 0;
    transform.parent.forward = directionToPosition;
}
```

Figure 9.70: LookTo function calls

2. Complete the `LookTo` function by calculating the direction of our parent to the target position. We access our parent with `transform.parent` because, remember, we are the child AI object—the object that will move is our parent. Then, we set the `y` component of the direction to `0` to prevent the direction from pointing upward or downward—we don't want our enemy to rotate vertically. Finally, we set the forward vector of our parent to that direction so it will face the target position immediately. You can replace that with interpolation through quaternions to have a smoother rotation if you want to, but let's keep things as simple as possible for now:

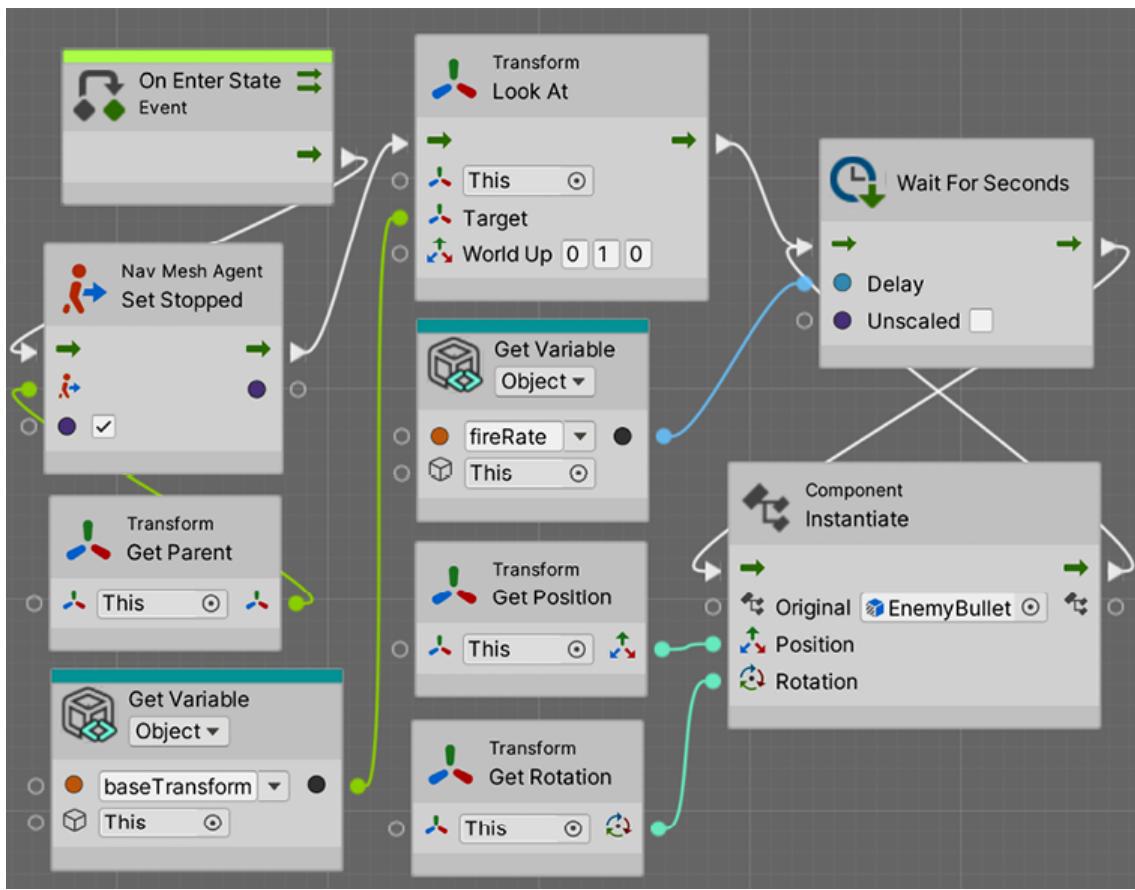


Figure 9.71: Looking toward a target

3. Regarding the Visual Scripting version, **AttackBase** actions look like this:

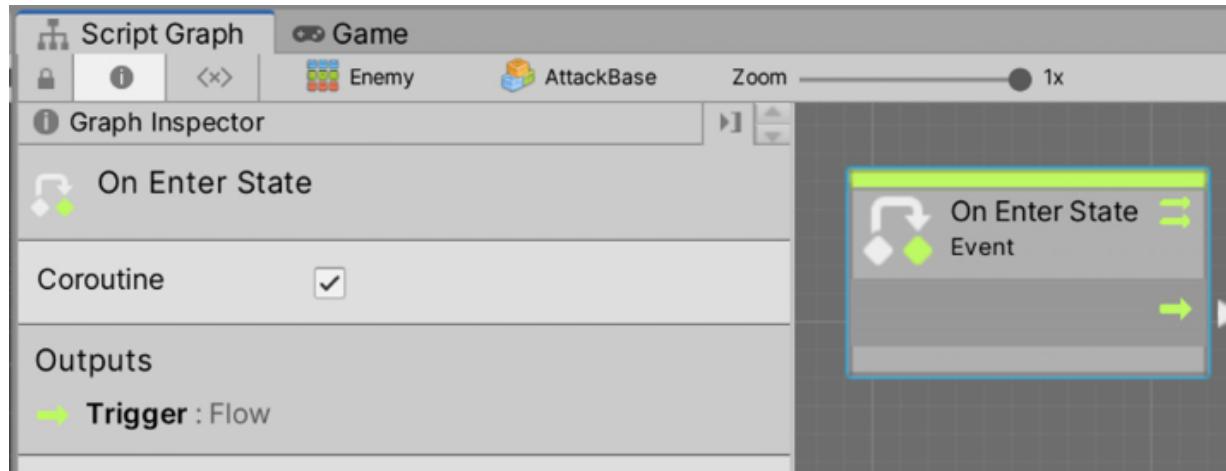


Figure 9.72: AttackBase state

In this state, we have some things to highlight. First, we are using the **LookAt** node in the **OnEnterState** event node after the **SetStopped** node. As you might imagine, this does the same as we did with math in C#. We specify a target to look at (our base transform) and then we specify that the **World Up** parameter is a vector pointing upwards $0, 1, 0$. This will make our object look at the base but maintain its up vector pointing to the sky, meaning our object will not look at the floor if the target is lower than him. We can use this exact function in C# if we want to (`transform.LookAt`); the idea was just to show you all the options. Also note that we execute `LookAt` only when the state becomes active—as the base doesn't move, we don't need to constantly update our orientation. The second thing to highlight is that we used coroutines to shoot, the same idea we used in the `Enemy Spawner` to constantly spawn enemies. Essentially, we make an infinite loop between **Wait For Seconds** and **Instantiate**. We took this approach here because it was convenient given it takes fewer nodes in Visual Scripting. Remember to select the **OnEnterState** node and check the **Coroutine** checkbox as we did before. Also, we need a new Float type variable called `fireRate` in the Enemy's AI child object:

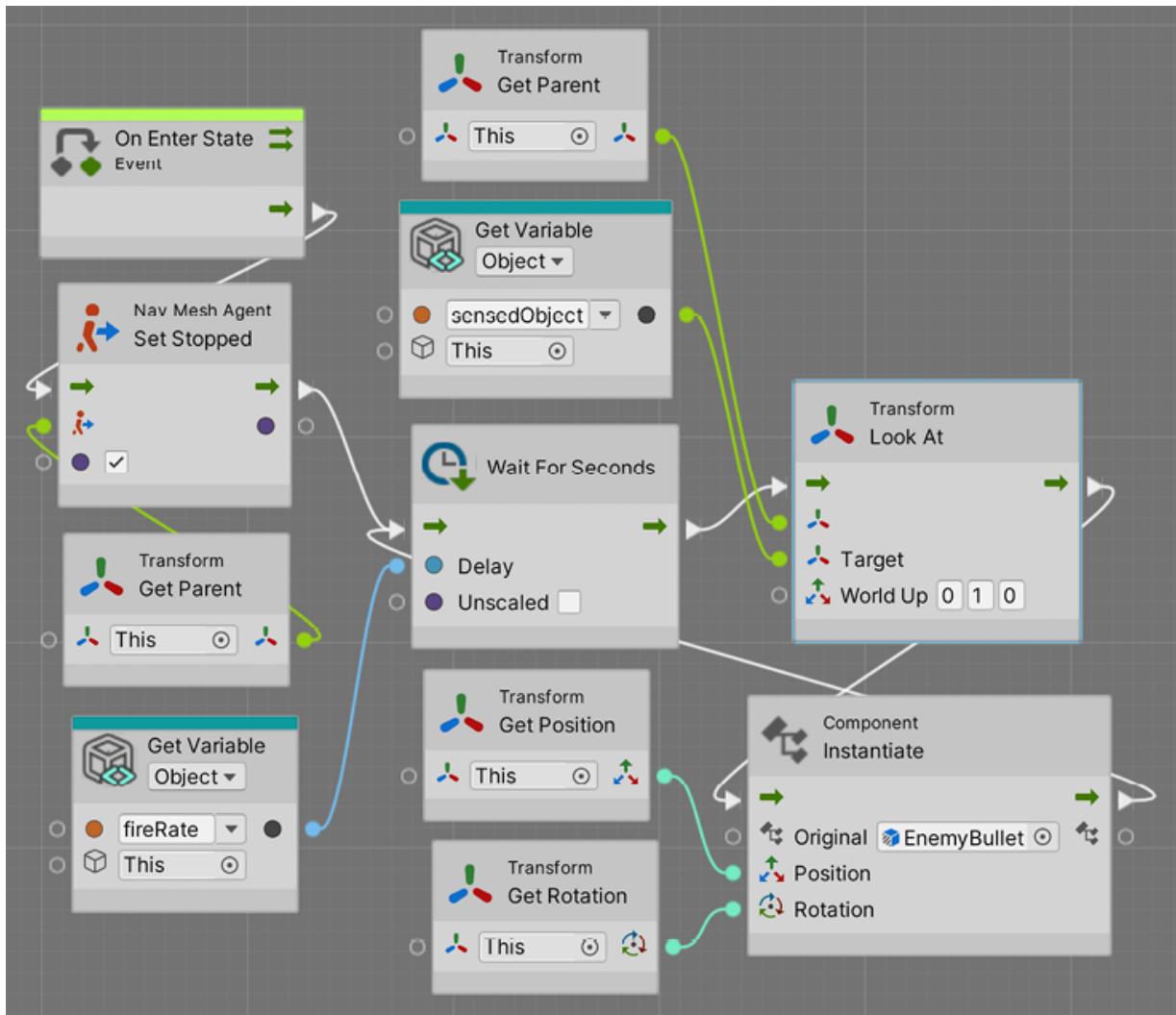


Figure 9.73: Coroutines

Then, **AttackPlayer** will look like this:

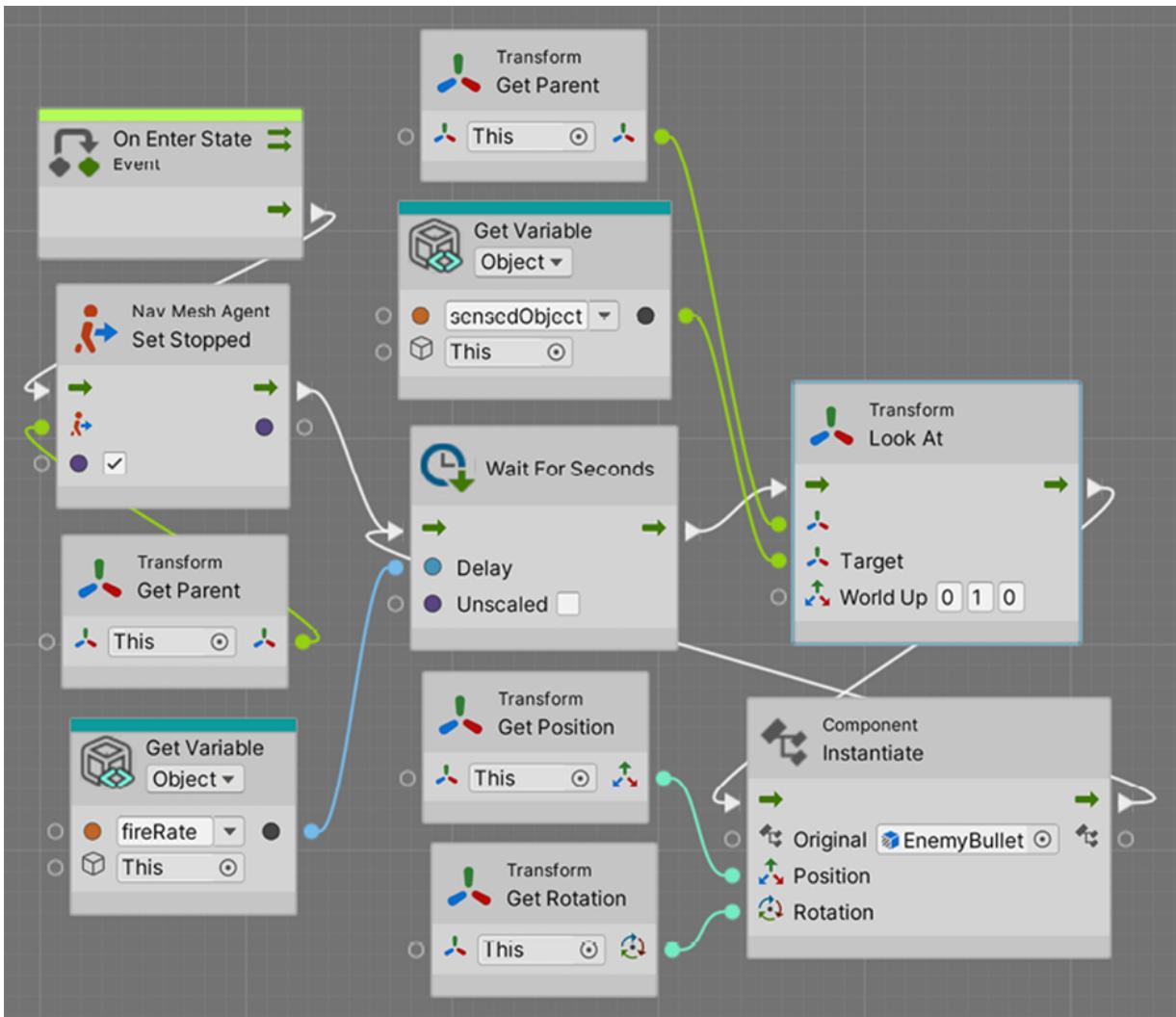


Figure 9.74: *AttackPlayer* state

Essentially it is the same as **AttackBase**, but that looks towards the `sensedObject` instead toward the player's base, and we also made the **LookAt** node part of the infinite loop, to correct the enemy's heading before shooting to target the player. With that, we have finished all AI behaviors. Of course, these scripts/graphs are big enough to deserve some rework and splitting in the future, but with this, we have prototyped our AI, and we can test it until we are happy with it, and then we can improve this code.

Summary

I'm pretty sure AI is not what you imagined; you are not creating Skynet here, but we have accomplished a simple but interesting AI to challenge our players, which we can iterate and tweak to tailor to our game's expected behavior. We saw how to gather our surrounding information through sensors to make decisions on what action to execute using FSMs and using different Unity systems such as Pathfinding to make the AI execute those actions. We used those systems to diagram a State Machine capable of detecting the player, running to them, and attacking them, and if the player is not there, just going to the base to accomplish its task to destroy it. As we move into the next chapter, we'll shift our focus to another vital aspect of game development: enhancing the graphics and audio. Get ready to dive into creating materials and shaders that will bring your game world to life.

10 Material Alchemy: URP and Shader Graph for Stunning Visuals

Join our book community on Discord

<https://packt.link/unitydev>



Welcome to the first chapter of *Part 3*. Here, we will dive deep into the different graphics and audio systems of Unity to dramatically improve the look and feel of the game. Let's begin our journey into the world of shaders, the artists behind the scenes in every Unity game, and learn how to craft our own from Scratch. We will start by discussing what a shader is and how to create our own to achieve several custom effects that couldn't be accomplished using the default Unity Shaders. We will be creating a simple water animation effect using Shader Graph, a visual shader editor included in the Universal Render Pipeline, the preferred option for creators to be able to launch their games into a wide variety of devices, including web or mobile among others. Also known as URP, this is one of the different rendering pipelines available in Unity, which provides rendering features oriented toward performance. We will be discussing some of its capabilities in this chapter. In this chapter, we will examine the following shader concepts:

- Introducing shaders and URP
- Creating shaders with Shader Graph

Introducing shaders and URP

Remember the glowing orb material we created in Part 1? Let's explore how its shader property manipulates light to create that glow effect. In this first section of this chapter, we will be exploring the concept of a shader as a way to program the video card to achieve custom visual effects. We will also be discussing how URP works with those shaders, and the default shaders it provides. In this section, we will cover the following concepts related to shaders:

- Shader Pipeline
- Render Pipeline and URP
- URP built-in shaders

Let's start by discussing how a shader modifies the Shader Pipeline to achieve effects.

Shader Pipeline

Whenever a video card renders a 3D model, it needs different information to process, such as a **Mesh**, **Textures**, the transform of the object (position, rotation, and scale), and lights that affect that object. With that data, the video card must output the pixels of the object into the **back-buffer**, an image where the video card will be drawing our objects, but the user won't see this yet. This is done to prevent the user from seeing unfinished results, given we can still be drawing at the time the monitor refreshes. That image will be shown when Unity finishes rendering all objects (and some effects) to display the finished scene, swapping the **Back-buffer** with the **front-buffer**, the image that the user actually sees. You can imagine this as having a page with an image that is being shown to the user while you draw a new image, and when you finish the new drawing, you just swap the pages and start drawing again on the page the user is not seeing, repeating this with every frame. That's the usual way to render an object, but what happens between the input of the data and the output of the pixels can be handled in a myriad of different ways and techniques that depend on how you want your object to look; maybe you want it to be

realistic or look like a hologram, maybe the object needs a disintegration effect or a toon effect—there are endless possibilities. The way to specify how our video card will handle the render of the object is through a shader. A **shader** is a program coded in specific video card languages, such as:

- **HLSL**: The DirectX shading language, DirectX being a graphics library.
- **GLSL**: The OpenGL shading language, OpenGL also being a graphics library.
- **CG**: A language that can output either HLSL or GLSL, depending on which graphics library we use in our game.
- **Shader Graph**: A visual language that will be automatically converted into one of the previously mentioned languages according to our needs. This is the one we will be using given its simplicity (more on that later).

Any of those languages can be used to configure different stages of the render process necessary to render a given object, sometimes not only configuring them but also replacing them with completely custom code to achieve the exact effect we want. All of the stages to render an object make up what we call the Shader Pipeline, a chain of modifications applied to the input data until it is transformed into pixels. Each stage of the pipeline is in charge of different modifications and depending on the video card shader model, this pipeline can vary a lot. In the next diagram, you can find a simplified Render Pipeline, skipping advanced/optional stages that are not important right now:



Figure 10.1: Common Shader Pipeline

Think of the Shader Pipeline like an assembly line in a factory, where each stage represents a different worker specializing in a specific task, collectively contributing to the final product. Let's discuss each of the stages:

- **Input Assembler:** Here is where all of the mesh data, such as vertex position, UVs, and normals, is assembled to be prepared for the next stage.
- **Vertex Shader:** This stage used to be limited to applying the transformation of the object, the position and perspective of the camera, and simple lighting calculations. In modern GPUs, you are in charge of doing whatever you want. This stage receives each one of the vertices of the object to render and outputs a modified one. You have the chance to modify the geometry of the object here. The usual code here is applying the transform of the object, but you can also apply several effects such as inflating the object along its normals to apply the old toon effect technique or apply distortion adding random offsets to each vertex to recreate a hologram. There's also the opportunity to calculate data for the next stages.
- **Primitive Culling:** Most of the models you are going to render have the particularity that you will never see the back side of a model face. In a cube, there's no way to look at its inner sides. Given that, rendering both sides of each face of the cube makes no sense, and this stage takes care of that. Primitive Culling will determine whether the face needs to be rendered based on the orientation of the face, saving lots of pixel calculation of occluded faces. You can change this to behave differently for specific cases; as an example, we can create a glass box that needs to be transparent to see all sides of the box. Don't confuse this with other types of culling, like Frustum Culling. This other type of culling filter objects outside the camera view area before they are even sent to the shader pipeline.

- **Rasterizer:** Now that we have the modified and visible geometry of our model calculated, it's time to convert it into pixels. The rasterizer creates all pixels for the triangles of our mesh. Lots of things happen here but again, we have very little control of that; the usual way to rasterize is just to create all pixels inside the edges of the mesh triangles. We have other modes that just render the pixels on the edges to see a wireframe effect, but this is usually used for debugging purposes:

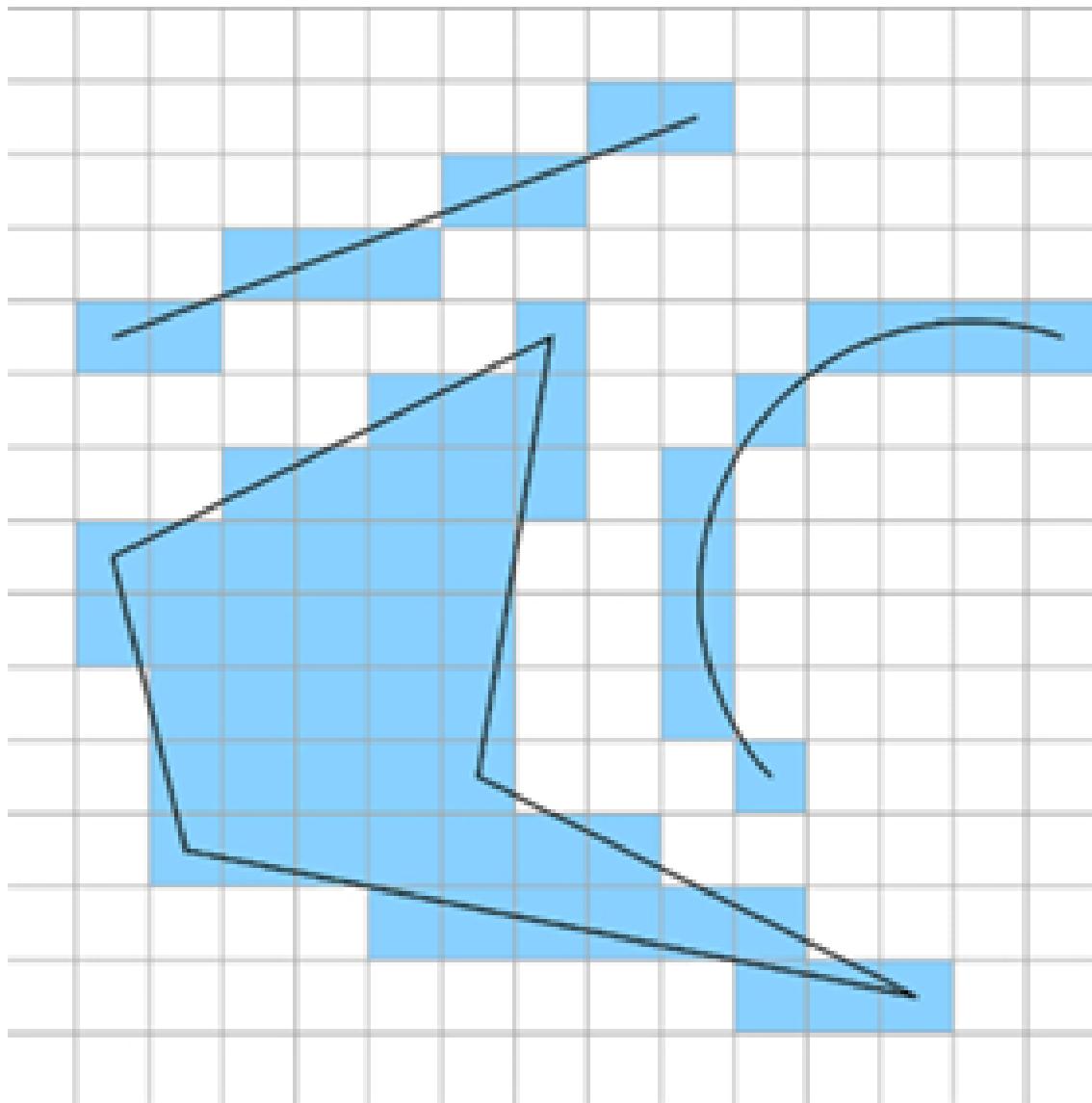


Figure 10.2: Example of figures being rasterized

- **Fragment Shader:** This is one of the most customizable stages of all. Its purpose is simple: just determine the color of each one of the fragments (pixels) that the rasterizer has generated. Here, lots of things can happen, from simply outputting a plain color or sampling a texture to applying complex lighting calculations such as normal mapping and PBR. Also, you can use this stage to create special effects such as water animations, holograms, distortions, disintegrations, and any special effects that require you to modify what the pixels look like. We will explore how we can use this stage in the next sections of this chapter.
- **Depth Testing:** Before showing a pixel on the screen, we need to check whether it can be seen. This stage checks whether the pixel's depth is behind or in front of the previous pixel rendered in the same position, guaranteeing that regardless of the rendering order of the objects, the nearest pixels to the camera are always being drawn on top of others. Again, usually, this stage is left in its default state, prioritizing pixels that are nearer to the camera, but some effects require different behavior. Also, nowadays we have **Early-Z testing**, which does this same test but before the Fragment shader, but let's keep things simple for now. As an example, in the next screenshot, you can see an effect that allows you to see objects that are behind other objects, like the one used in *Age of Empires* when a unit is behind a building:

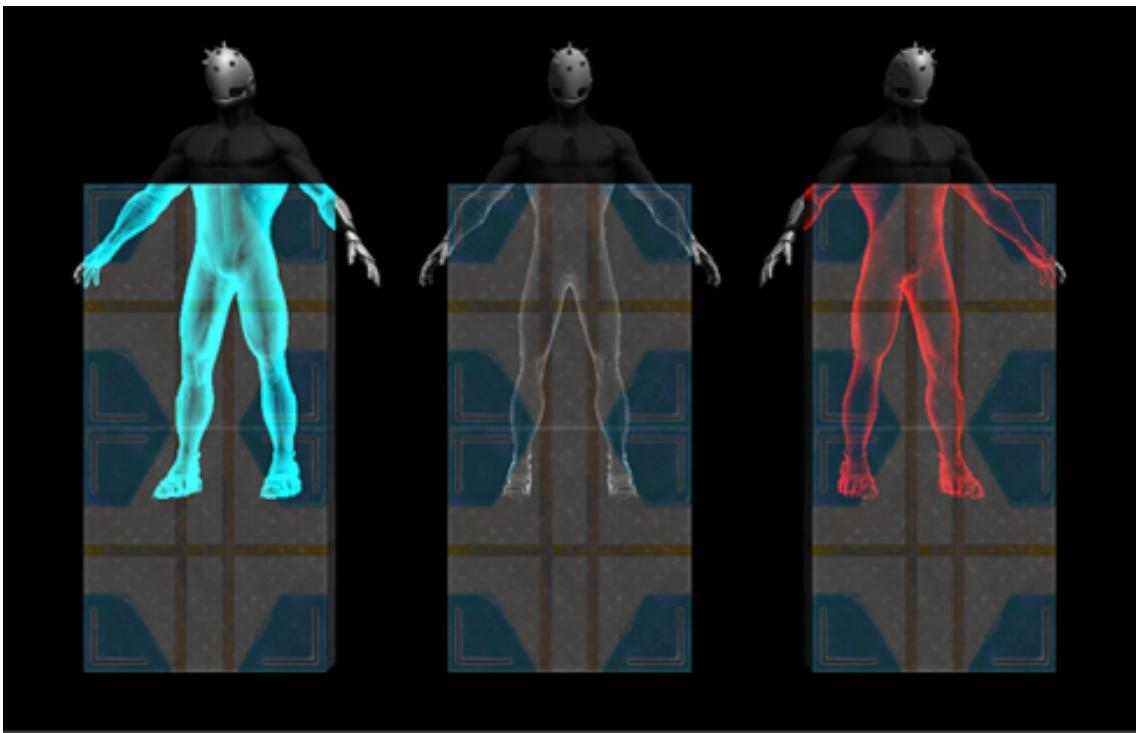


Figure 10.3: Rendering the occluded parts of the character

- **Blending:** Once the color of the pixel is determined and we are sure the pixel is not occluded by a previous pixel, the final step is to put it in the back-buffer (the frame or image you are drawing). Usually, we just override whatever pixel was in that position (because our pixel is nearer to the camera), but if you think about transparent objects, we need to combine our pixel with the previous one to make the transparent effect. Transparency has other things to take into account aside from the blending, but the main idea is that blending controls exactly how the pixel will be combined with the previously rendered pixel in the back-buffer.

Shader Pipelines is a subject that would require an entire book, but for the scope of this book, the previous description will give you a good idea of what a shader does, and the possible effects that it can achieve. Now that we have discussed how a shader renders a single object, it is worth discussing how Unity renders all of the objects using Render Pipelines.

Info

For more information about shaders, you can start reading the following link:

<https://docs.unity3d.com/Manual/shader-writing.html>

Render Pipeline and URP

We have covered how the video card renders an object, but Unity is in charge of asking the video card to execute its Shader Pipeline per object. To do so, Unity needs to do lots of preparations and calculations to determine exactly how and when each shader needs to be executed. The responsibility of doing this is with what Unity calls the Render Pipeline. Think of a Unity's Render pipeline like a film director orchestrating how each scene (object) is presented, with URP as one of its advanced cameras, optimizing how each shot is captured. Also, think of the Render Pipeline as the stage crew of a theater, setting the scene and lighting for each object (actor) to ensure they look the best under spotlight. A Render Pipeline is a way to draw the objects of the scene. At first, it sounds like there should be just one simple way of doing this, for example, iterating over all objects in the scene and executing the Shader Pipeline with the shader specified in each object's Material, but it can be more complex than that. Usually, the main difference between one Render Pipeline and another is the way in which lighting and some advanced effects are calculated, but they can differ in other ways. In previous Unity versions, there was just one single Render Pipeline, which is now called the **Built-in Renderer Pipeline** (also known as **BIRP**). It was a pipeline that had all of the possible features you would need for all kinds of projects, from mobile 2D graphics and simple 3D to cutting-edge 3D like the ones you can find in consoles or high-end PCs. This sounds ideal, but actually, it isn't. Having one single giant renderer that needs to be highly customizable to adapt to all possible scenarios generates lots of overhead and limitations that cause more headaches than creating a custom Render Pipeline. Luckily, the last versions of Unity introduced **Scriptable Render**

Pipeline (SRP), a way to create Render Pipeline adapted for your project. Luckily, Unity doesn't want you to create your own Render Pipeline for each project (which is a complex task), so it has created two custom pipelines for you that are ready to use: **URP** (formerly called LWRP, or Light Weight Render Pipeline), which stands for **Universal Render Pipeline**, and **HDRP**, which stands for **High Definition Render Pipeline**. The idea is that you must choose one or the other based on your project's requirements (unless you really need to create your own). URP, the one we selected when creating the project for our game, is a Render Pipeline suitable for most games that don't require lots of advanced graphics features, such as mobile games or simple PC games, while HDRP is packed with lots of advanced rendering features for high-quality games. The latter requires high-end hardware to run, while URP runs in almost every relevant target device. It is worth mentioning that you can swap between Built-in Renderer, HDRP, and URP whenever you want, including after creating the project (but this is not recommended):

New project

Editor Version: 2023.1.1f1 ▾

 **3D Mobile**
Core

 **2D Mobile**
Core

 **3D (URP)**
Core

 **3D (HDRP)**
Core

 **VR**
Core

 **AR**
Core



3D (URP)

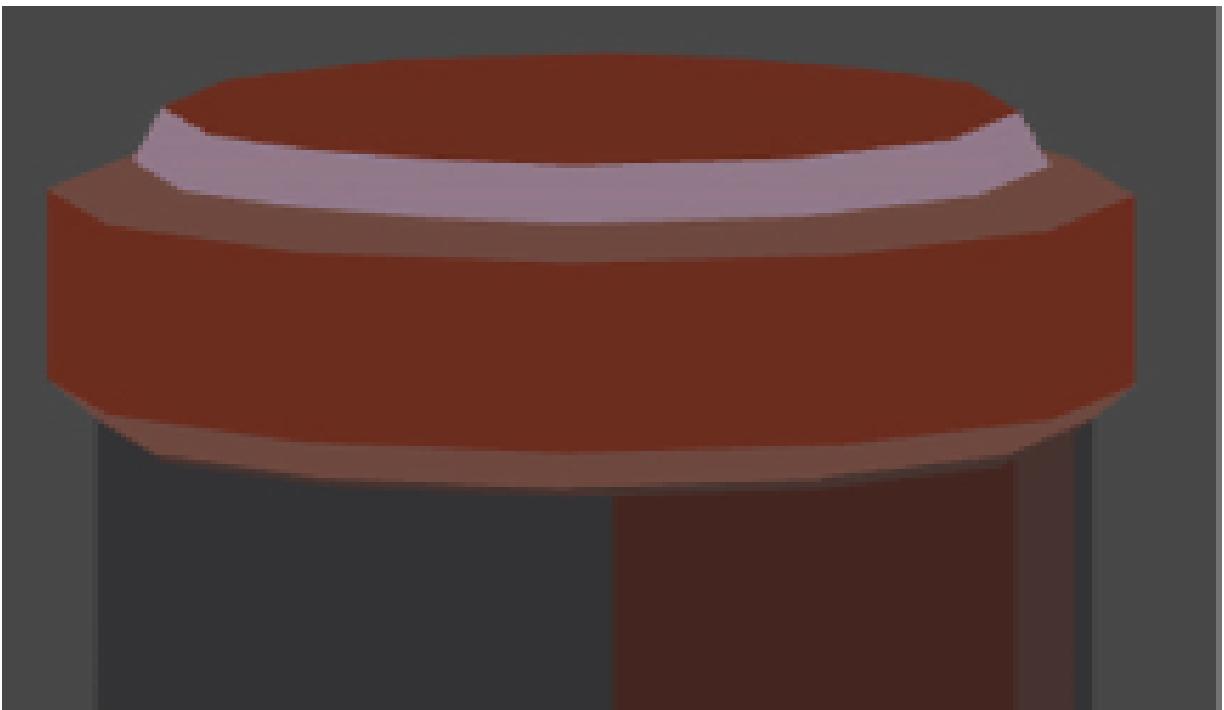
The URP (Universal Render Pipeline) blank template includes the settings and assets you need to start creating with URP. Equipped with...

 [Read more](#)

 [Download template](#)

[Cancel](#)

[Create project](#)



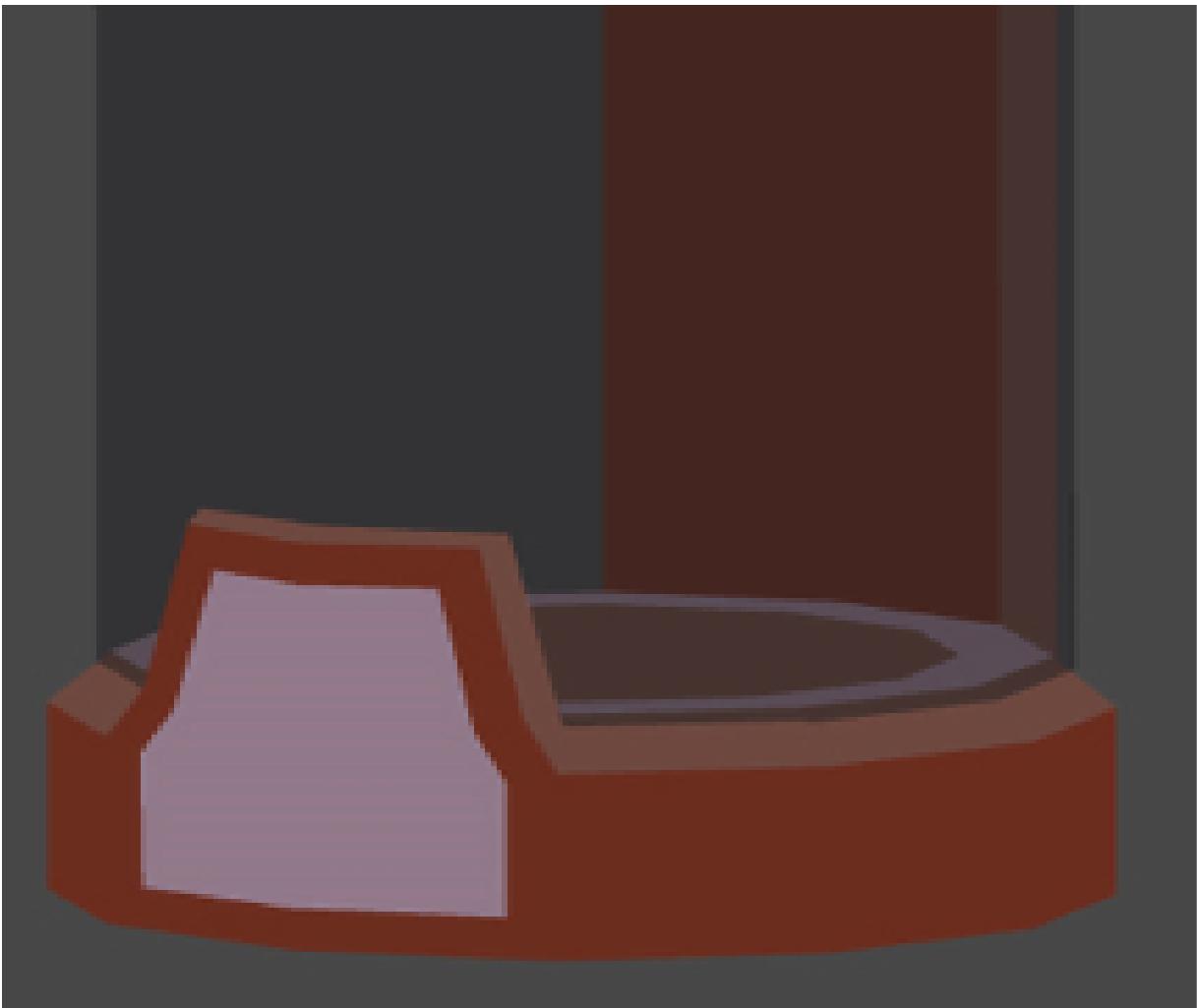


Figure 10.4: Project wizard showing HDRP and URP templates

We can discuss how each one is implemented and the differences between each, but again, this could fill entire chapters; right now, the idea of this section is for you to know why we picked URP when we created our project because it has some restrictions we will encounter throughout this book that we will need to take into account, so it is good to know why we accepted those limitations (to run our game on every relevant hardware). Also, we need to know that we have chosen URP because it has support for Shader Graph, the Unity tool that we will be using in this chapter to create custom effects. Previous Unity built-in pipelines didn't provide us with such a tool (aside from third-party plugins). Finally, another reason to introduce the concept of URP is that it comes with lots of built-in

shaders that we will need to know about before creating our own to prevent reinventing the wheel. This will allow us to get used to those shaders, because if you came from previous versions of Unity, the shaders you already know won't work here; actually, this is exactly what we are going to discuss in the next section of this chapter: the difference between the different URP built-in shaders.

URP built-in shaders

Now that we know the difference between URP and other pipelines, let's discuss which shaders come integrated into URP. Let's briefly describe the three most important shaders in this pipeline:

- **Lit:** This is the replacement of the old Standard Shader. This shader is useful for creating all kinds of realistic physics materials such as wood, rubber, metal, skin, and combinations of them (such as a character with skin and metal armor). It supports features like Normal Mapping, Occlusion, different lighting workflows like Metallic and Specular, and transparencies.
- **Simple Lit:** This is the replacement of the old Mobile/Diffuse Shader. As the name suggests, this shader is a simpler version of Lit, meaning that its lighting calculations are simpler approximations of how light works, getting fewer features than its counterpart. Basically, when you have simple graphics without realistic lighting effects, this is the best choice.
- **Unlit:** This is the replacement of the old Unlit/Texture Shader. Sometimes, you need objects with no lighting whatsoever, and in that case, this is the shader for you. No lighting doesn't mean an absence of light or complete darkness; it actually means that the object has no shadows at all, and it's fully visible without any shade. Some simplistic graphics can work with this, relying on shadowing being baked in the texture, meaning that the texture comes with the shadow.

This is extremely performant, especially for low-end devices such as mobile phones. Also, you have other cases such as light tubes or screens, objects that can't receive shadows because they emit light, so they will be seen at their full color even in complete darkness. In the following screenshot, you can see a 3D model using an Unlit Shader. It looks like it's being lit, but it's just the texture of the model that applied lighter and darker colors in different parts of the object:

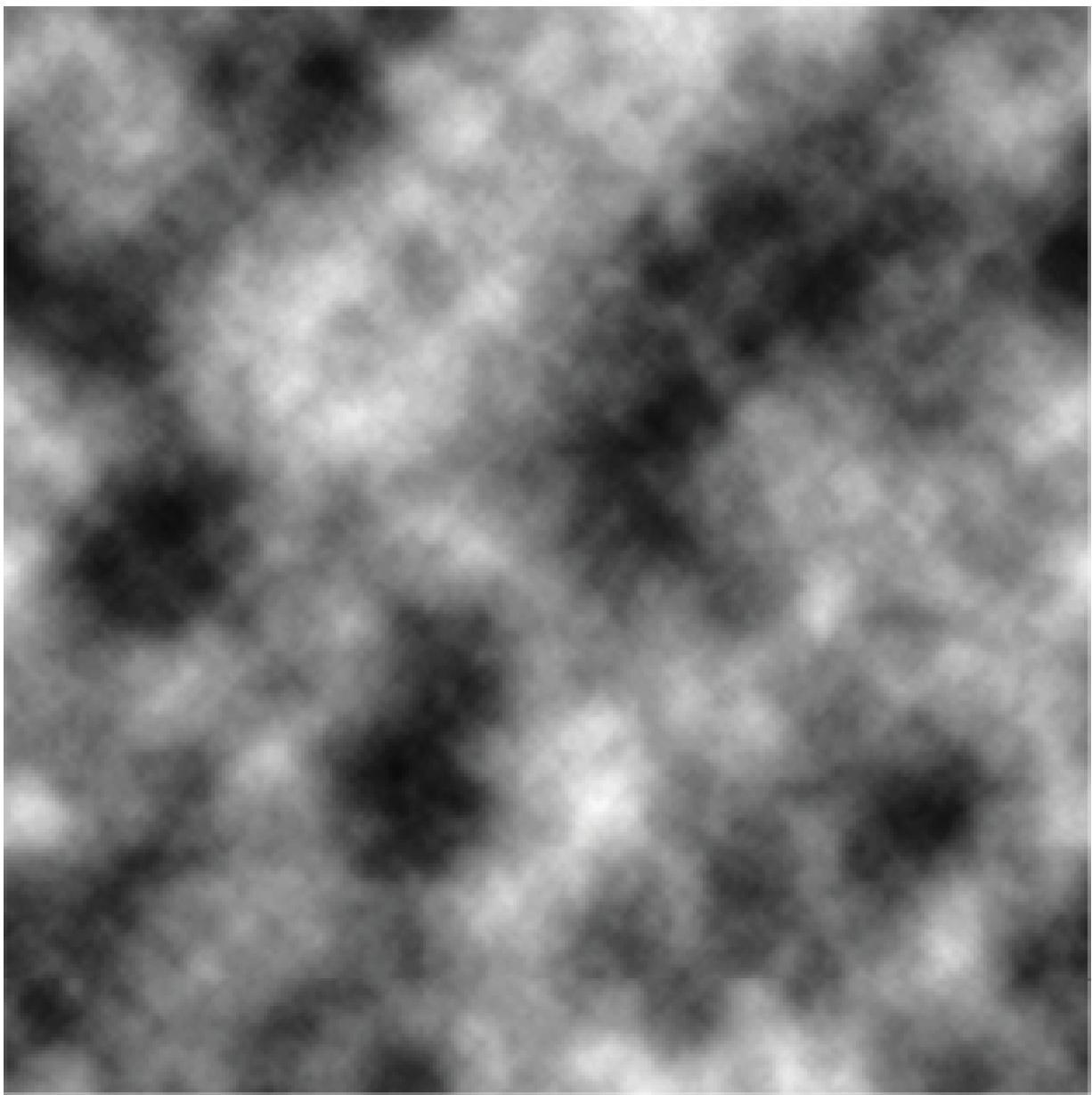


Figure 10.5: Pod using an Unlit effect to simulate cheap lighting

Let's do an interesting disintegration effect with the Simple Lit Shader to demonstrate its capabilities. You must do the following:

1. Begin by sourcing a **Cloud Noise** texture. You can find suitable textures on various free asset websites. Ensure the texture's resolution and format are compatible with Unity for optimal results:

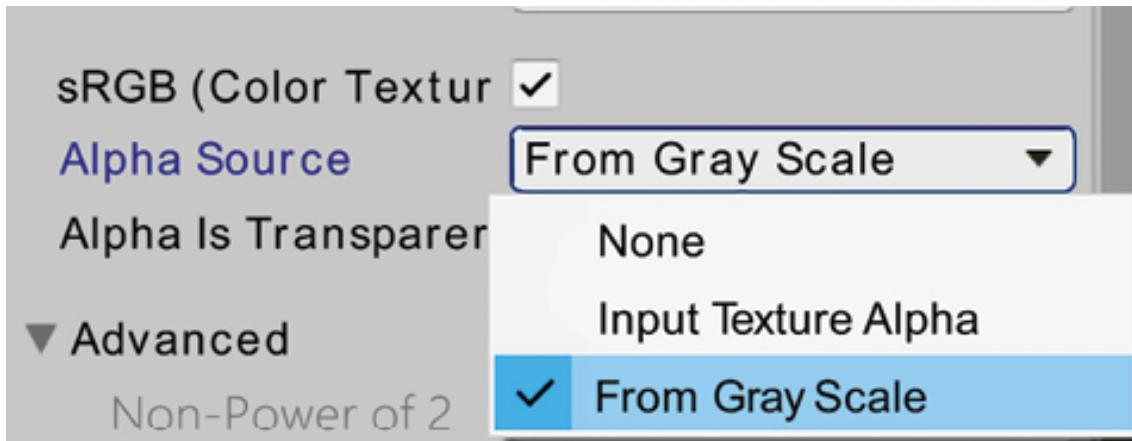


Figure 10.6: Noise texture

2. Select the recently imported texture in the **Project** panel.
3. In the Inspector, set the **Alpha Source** property to **From Gray Scale**. This will make the alpha channel of the texture be calculated based on the grayscale of the image. We will use the calculated alpha value to determine which pixels need to be deintegrated first (the darker ones first):

Prefab Variant

Audio Mixer

Material

Material Variant

Lens Flare

Render Texture

Lightmap Parameters

Lighting Settings

Figure 10.7: Generate Alpha From Gray Scale texture setting

The Alpha channel of a color is often associated with transparency, but you will notice that our object won't be transparent. The Alpha channel is extra color data that can be used for several purposes when creating effects. In this case, we will use it to determine which pixels are being disintegrated first.

1. Click the + icon in the Project view and select **Material**. You can rename it giving it a proper name, so it's easier to find it later:

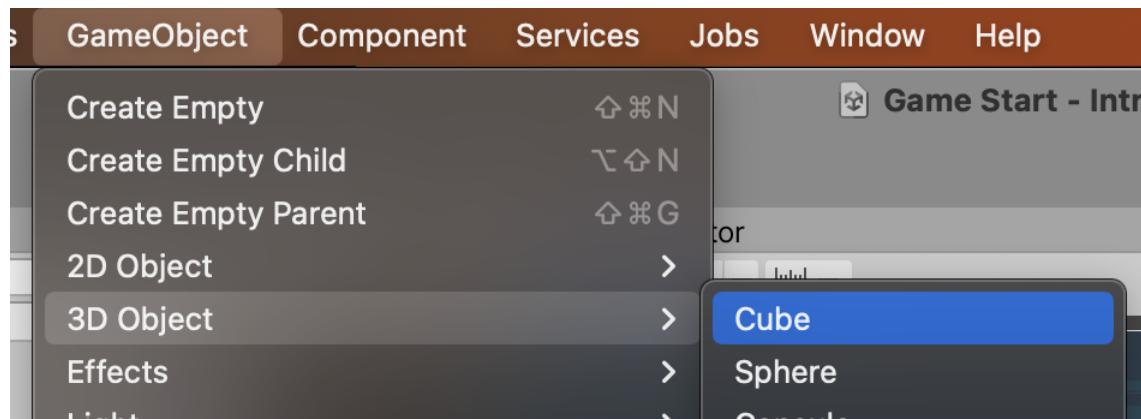


Figure 10.8: Material creation button

2. Create a cube by going to the top menu and selecting **GameObject | 3D Object | Cube:**

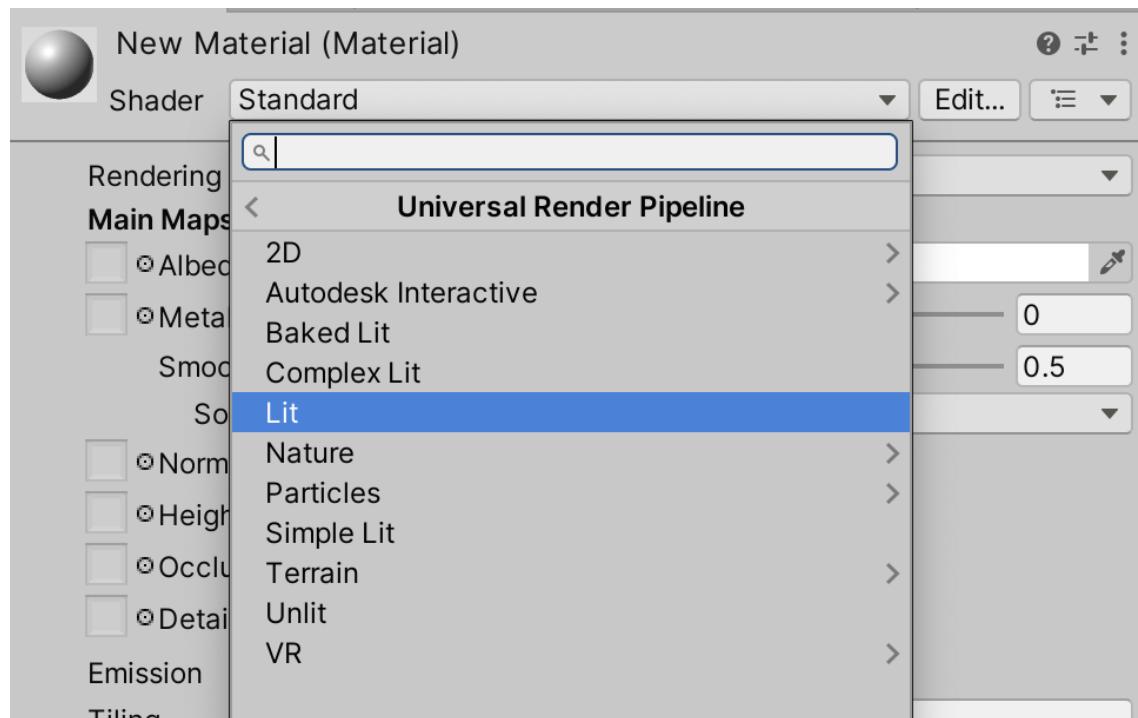


Figure 10.9: Cube primitive creation

3. Drag the Material from the Project window to the cube in the Scene window.

4. Click in the drop-down menu at the right of the **Shader** property in the Inspector and look for the **Universal Render Pipeline | Simple Lit** option. We could also work with the default shader (**Lit**), but **Simple Lit** is going to be easier on performance and we won't use the advanced features of **Lit**:

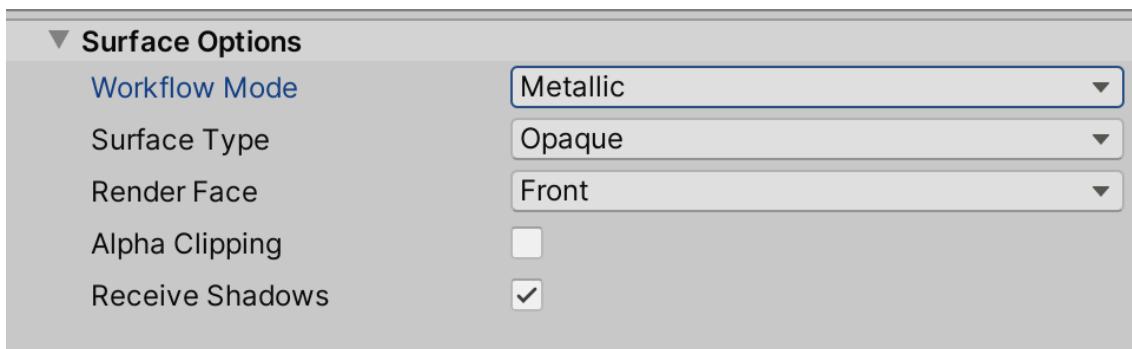


Figure 10.10: Simple Lit Shader selection

5. Next, select the newly created **Material** in your project. Drag and drop the downloaded Cloud Noise Texture into the **Base Map** section. This step visually binds your texture to the shader, enabling the disintegration effect.
6. Enable the **Alpha Clipping** option and adjust the **Threshold** slider to `0.5`. Alpha Clipping plays a critical role in how the shader interprets texture transparency, influencing the disintegration effect's appearance.

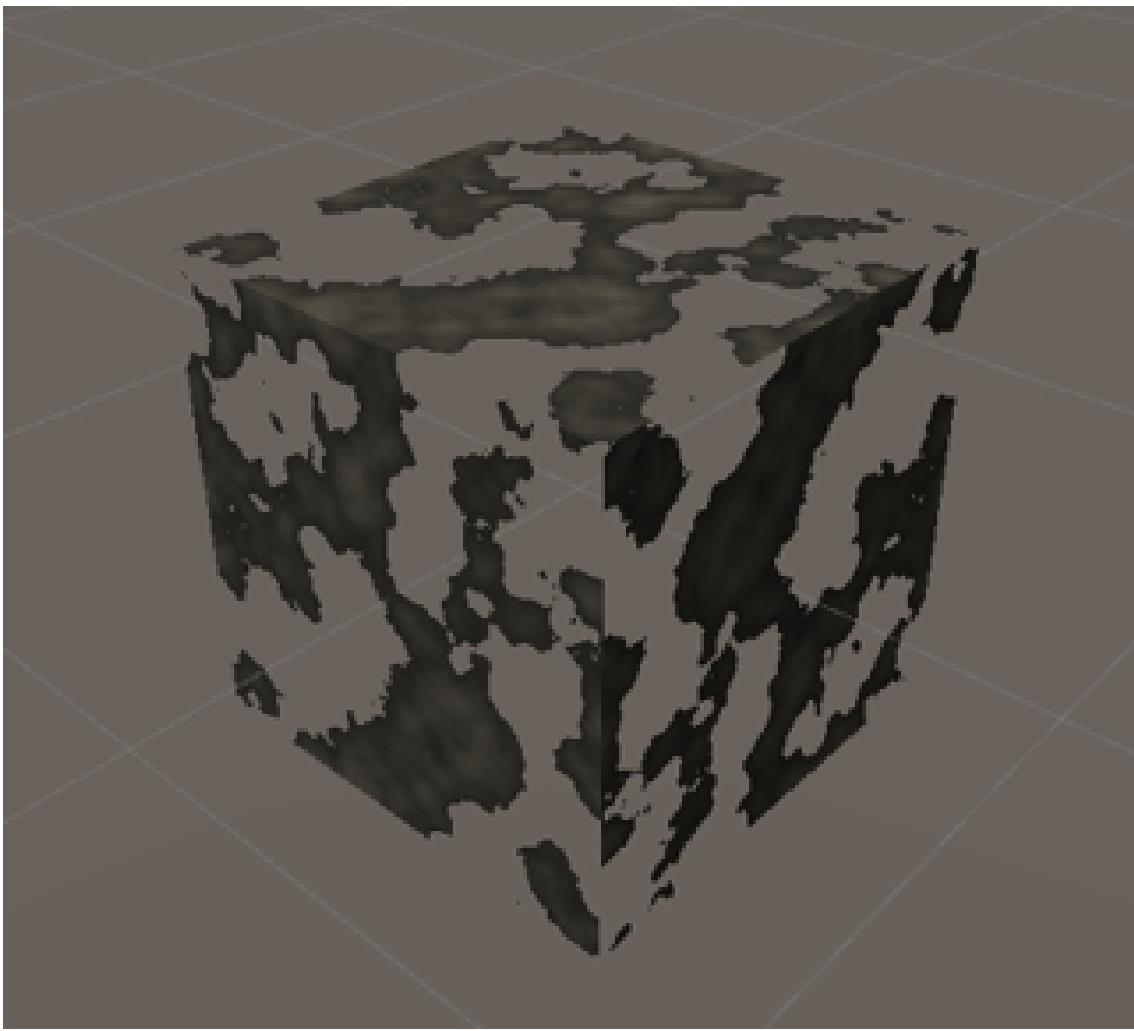


Figure 10.11: Alpha Clipping Threshold Material slider

7. As you move the **Threshold** slider, the object will start to disintegrate. **Alpha Clipping** discards pixels that have less Alpha intensity than the **Threshold** value:

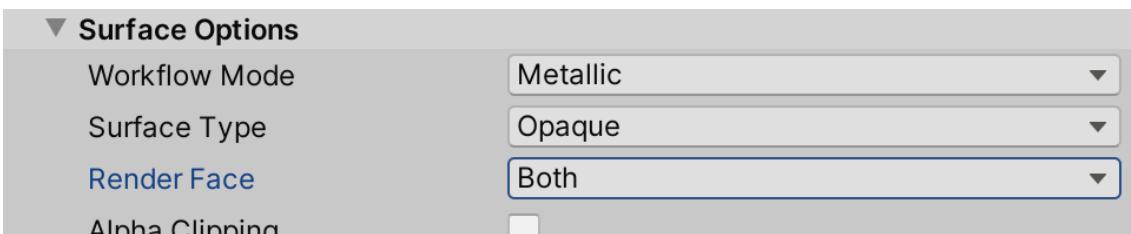


Figure 10.12: Disintegration effect with Alpha Clipping

- Finally, set Render Face to Both to see both sides of the cube's faces:

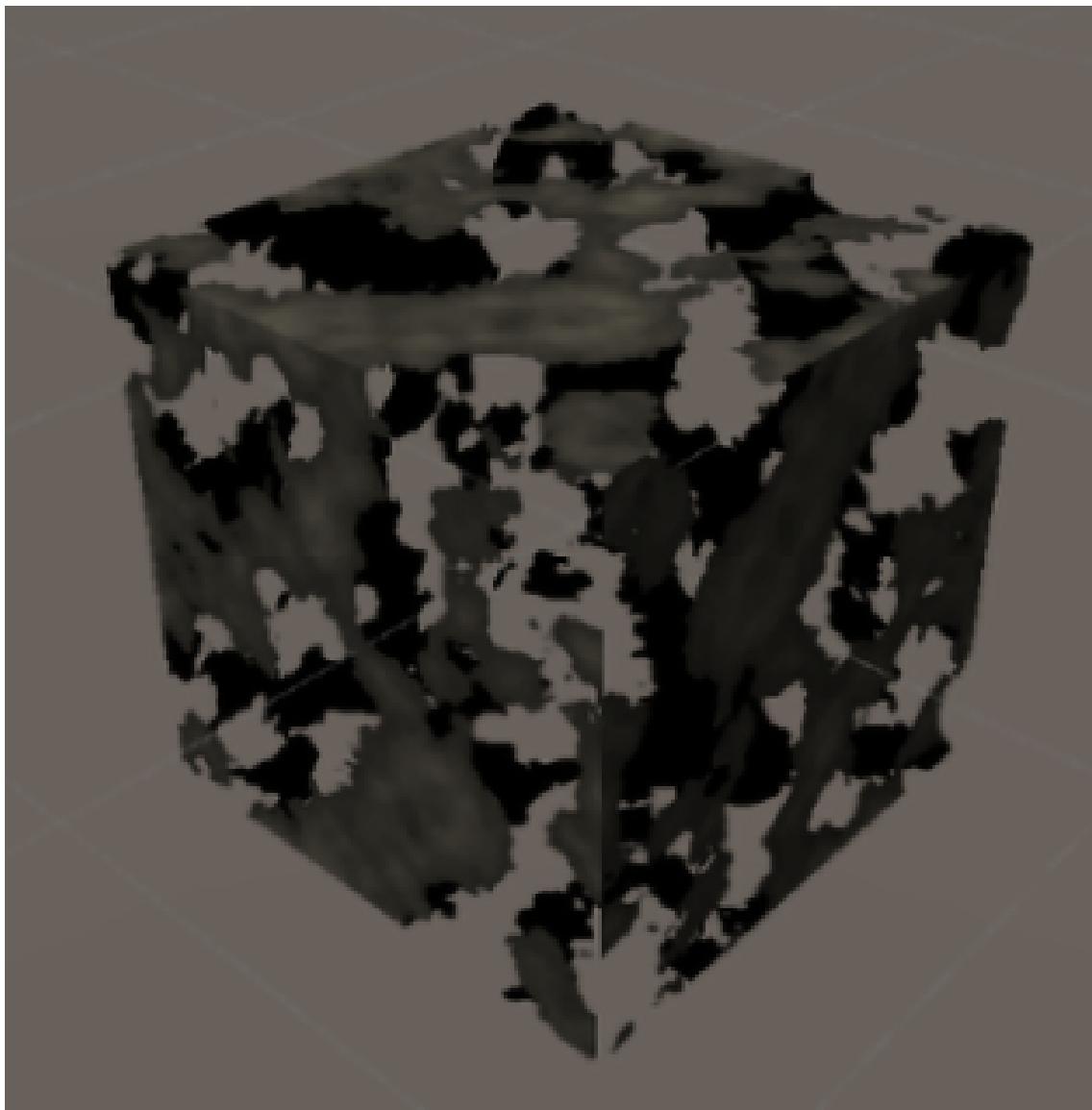


Figure 10.13: Double-sided render face

- Take into account that the artist that creates the texture can configure the Alpha channel manually instead of calculating it from the grayscale, just to control exactly how the disintegration effect must look regardless of the texture's color distribution:

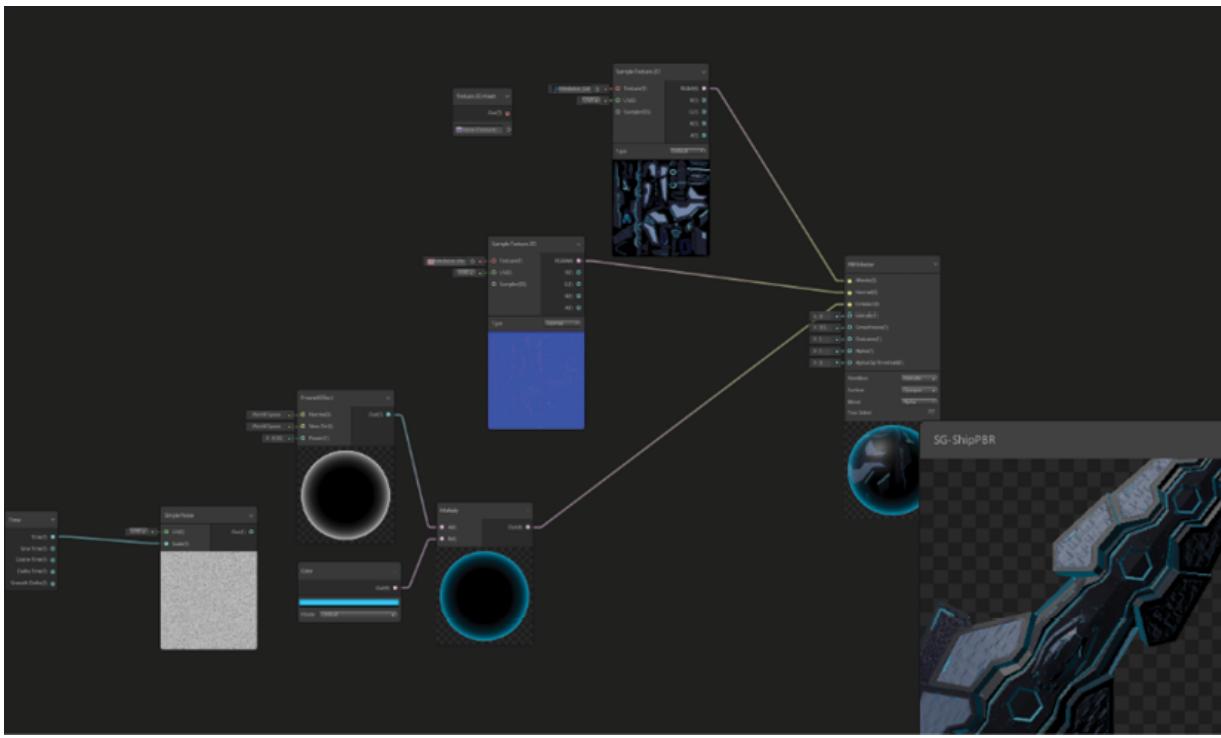


Figure 10.14: Double-sided Alpha Clipping

The idea of this section is not to give a comprehensive guide of all of the properties of all URP shaders, but to give you an idea of what a shader can do when properly configured and when to use each one of the integrated shaders. Sometimes, you can achieve the effect you need just by using existing shaders, probably in 99% of cases in simple games, so try to stick to them as much as you can. But if you really need to create a custom shader to create a very specific effect, the next section will teach you how to use the URP tool called **Shader Graph**.

Creating shaders with Shader Graph

Now that we know how shaders work and the existing shaders in URP, we have a basic notion of when it is necessary to create a custom shader and when it is not necessary. In case you really need to create one, this section will cover the basics of effects creation with **Shader Graph**, a tool to create effects using a visual node-based

editor. This is an easy tool to use when you are not used to coding. In this section, we will discuss the following concepts of the Shader Graph:

- Creating our first Shader Graph
- Using textures
- Combining textures
- Applying transparency
- Creating Vertex effects

Let's start by seeing how we can create and use a Shader Graph.

Creating our first Shader Graph

Shader Graph is a tool that allows us to create custom effects using a node-based system. An effect in the Shader Graph can look like in the following screenshot:



Figure 10.15: *Shader Graph with nodes to create a custom effect*

We will discuss later what those nodes do and we will be creating an example effect step by step, but in the screenshot, you can see how the author created and connected several nodes—the interconnected boxes—with each one executing a specific process to achieve the effect. The idea of creating effects with Shader Graph is to learn which specific nodes you need and how to connect them properly. This is similar to the way we code the gameplay of the game, but this Shader Graph is adapted and simplified just for effect purposes. To create and edit our first Shader Graph, do the following:

1. In the Project window, click the + icon and find the **Shader Graph | URP | Lit Shader Graph** option. This will create a Shader Graph using the PBR mode, meaning that this shader will support lighting effects (unlike Unlit Graphs):

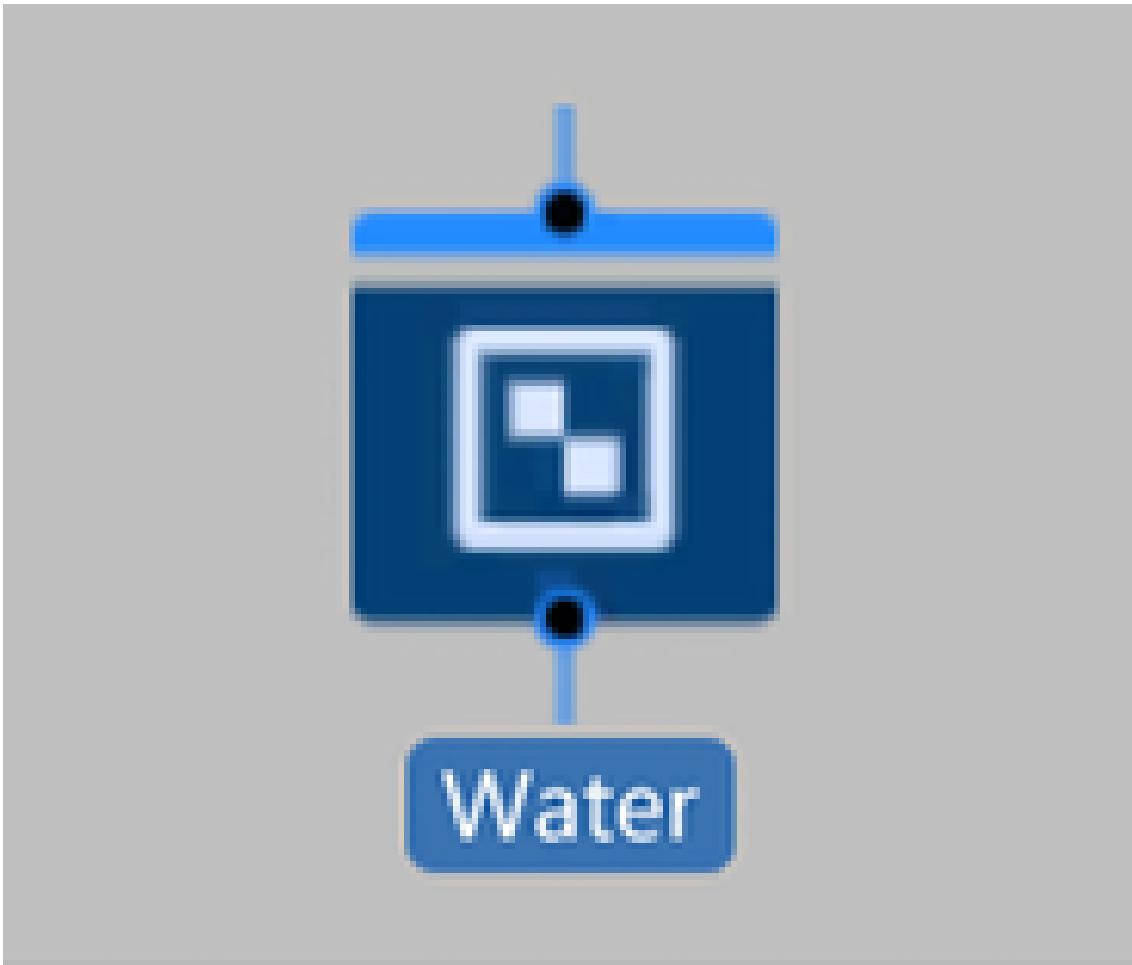


Figure 10.16: PBR Shader Graph creation

2. Name it `Water`. If you want the opportunity to rename the asset, remember that you can select the asset, right-click, and select **Rename**:

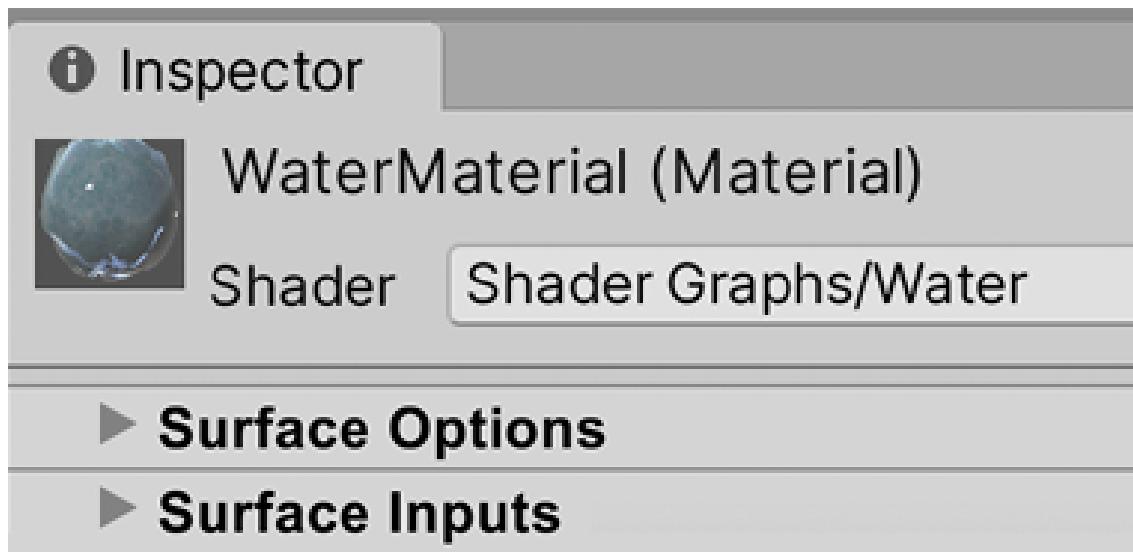
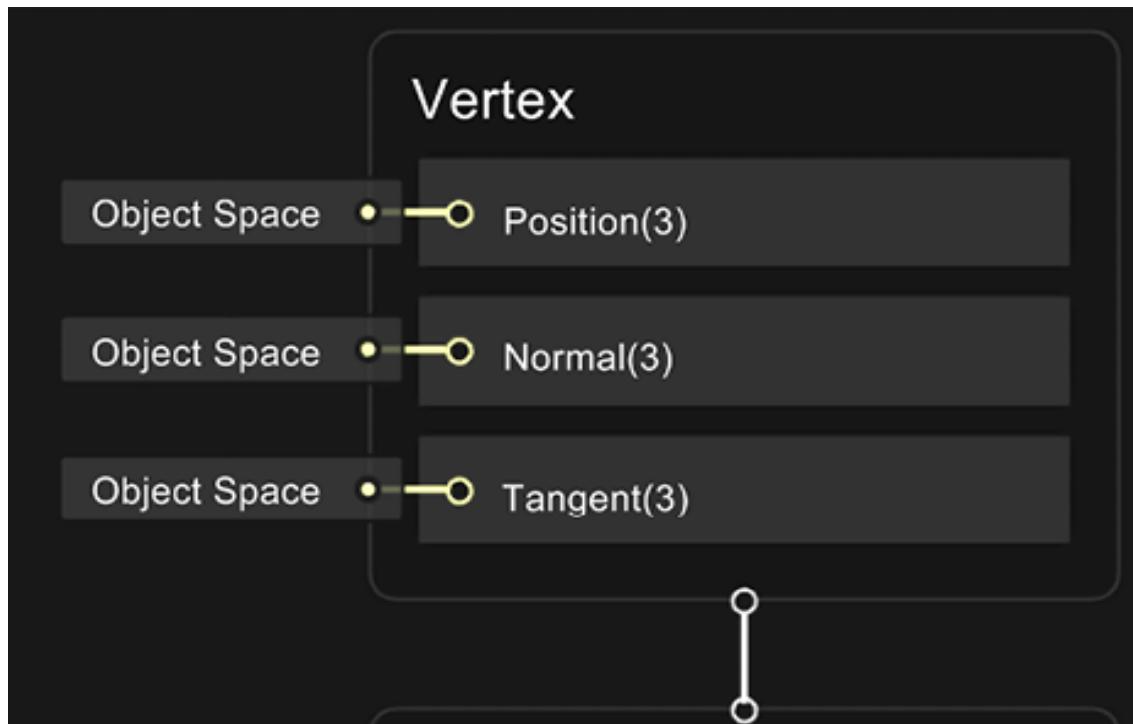


Figure 10.17: Shader Graph Asset

3. Create a new Material called `WaterMaterial` and set **Shader** to **Shader Graphs/Water**. If for some reason Unity doesn't allow you to do that, try right-clicking on the **Water Graph** and clicking **Reimport**. As you can see, the created Shader Graph now appears as a shader in the Material:



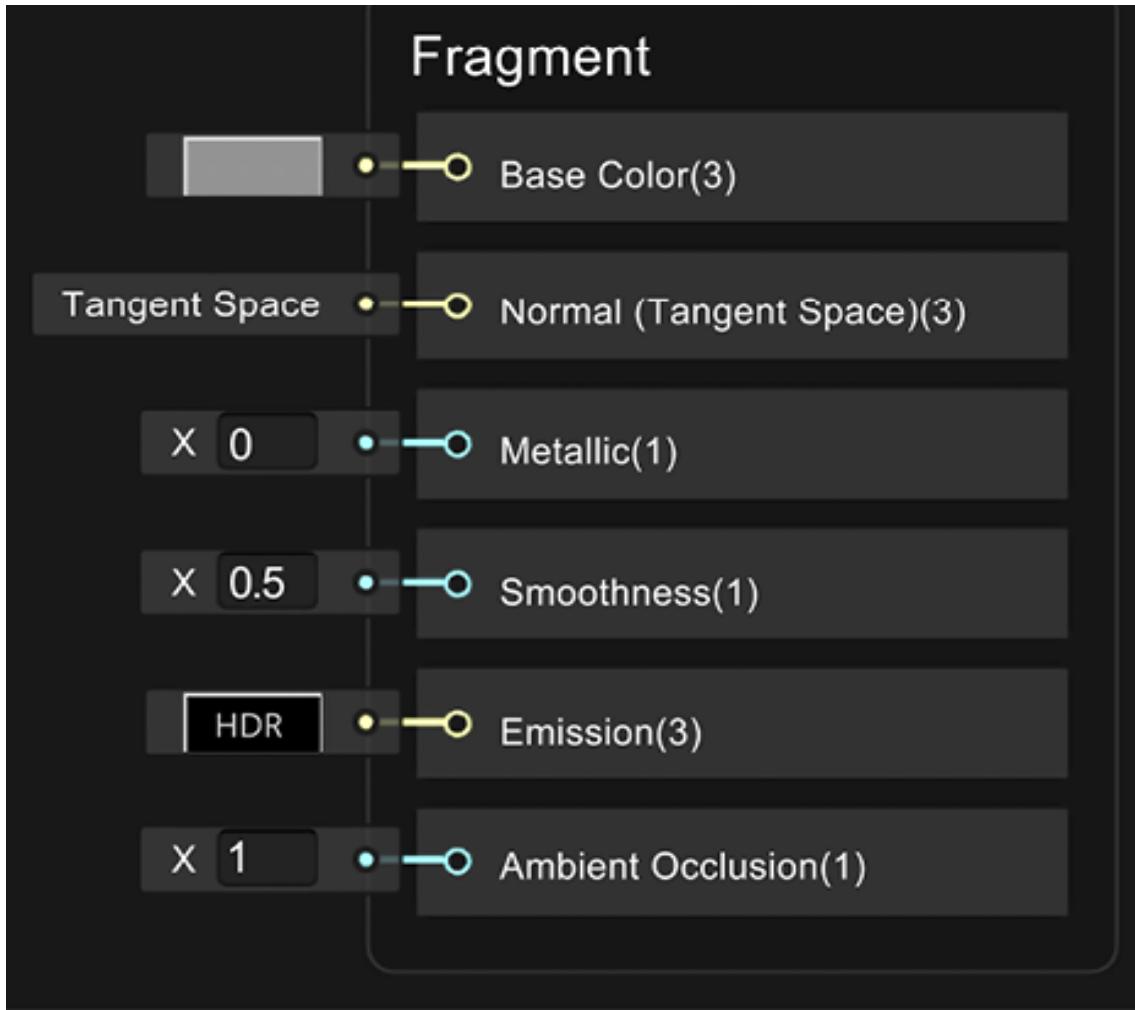


Figure 10.18: Setting a Shader Graph as a Material Shader

4. Create a plane with the **GameObject | 3D Object | Plane** option by right clicking from the Hierarchy window.
5. Drag the **Material** to the **Plane** to apply it.

Now, you have created your first custom shader and applied it to a Material. So far, it doesn't look interesting at all—it's just a gray effect—but now it's time to edit the graph to unlock its full potential. As the name of the graph suggests, we will be creating a water effect in this chapter to illustrate several nodes of the Shader Graph toolset and how to connect them, so let's start by discussing the Master node. When you open the graph by double-clicking the shader asset, you will see the following:



Figure 10.19: Master node with all of the properties needed to calculate object appearance

All nodes will have input pins, the data needed to work, and output pins, the results of its process. As an example, in a sum operation, we will have two input numbers and an output number, the result of the sum. In this case, you can see that the Master node only contains inputs, and that's because all data that enters the Master node will be used by Unity to calculate the rendering and lighting of the object, things such as the desired object color or texture (**Base Color** input pin), how smooth it is (**Smoothness** input pin), or how much metal it contains (**Metallic** input pin), properties that will affect how the lighting will be applied to the object. You can see that the Master node is split between a **Vertex** section and a **Fragment** section. The first is capable of changing the mesh of the object we are modifying to deform it, animate it, etc., while the latter will change how it will look, which textures to use, how it will be illuminated, etc. Let's start exploring how we can change that data in the **Fragment** section by doing the following:

1. Double-click the **Shader Graph** asset in Project View to open its editor.
2. Click in the gray rectangle at the left of the **Base Color** input pin:

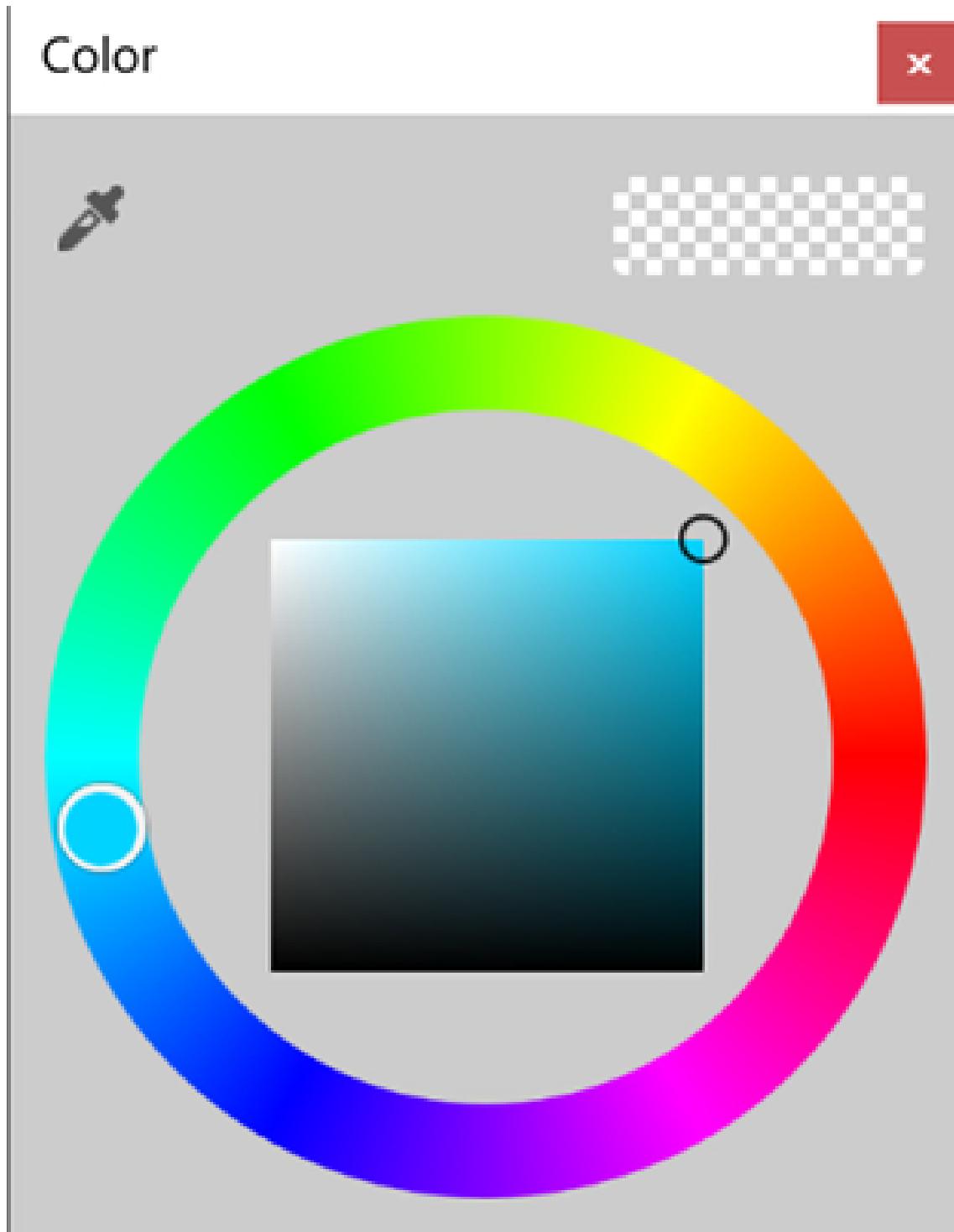


Figure 10.20: Base Color node input pin

3. In the color picker, select a light blue color, like water. Select the bluish part of the circle and then a shade of that color in the

middle rectangle:



Figure 10.21: Color picker

4. Set **Smoothness** to `0.9`, which will make the object almost completely smooth (90% of the total smoothness possible). This will make our water reflect the sky almost completely:



Figure 10.22: Smoothness PBR Master node input pin

5. Click the **Save Asset** button at the top left of the window:

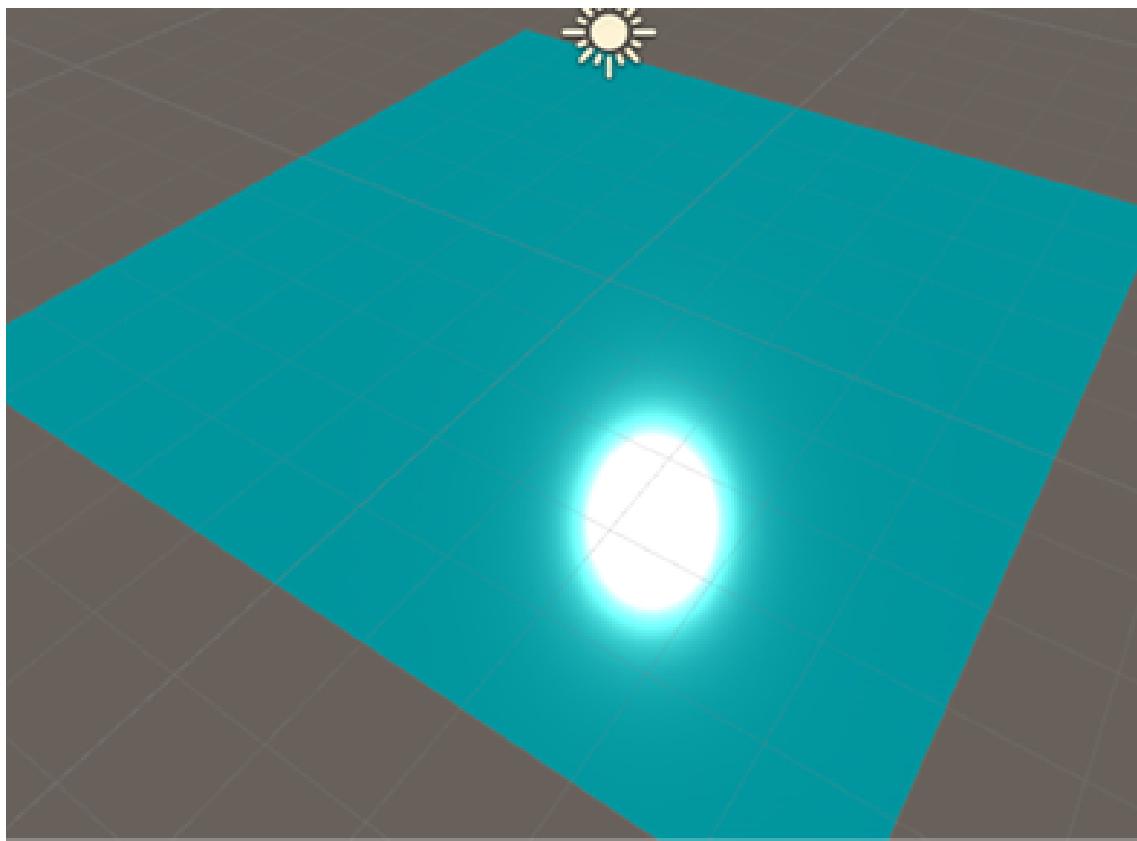


Figure 10.23: Shader Graph saving options

6. Go back to the Scene View and check the plane is light blue with the sun reflected on it:

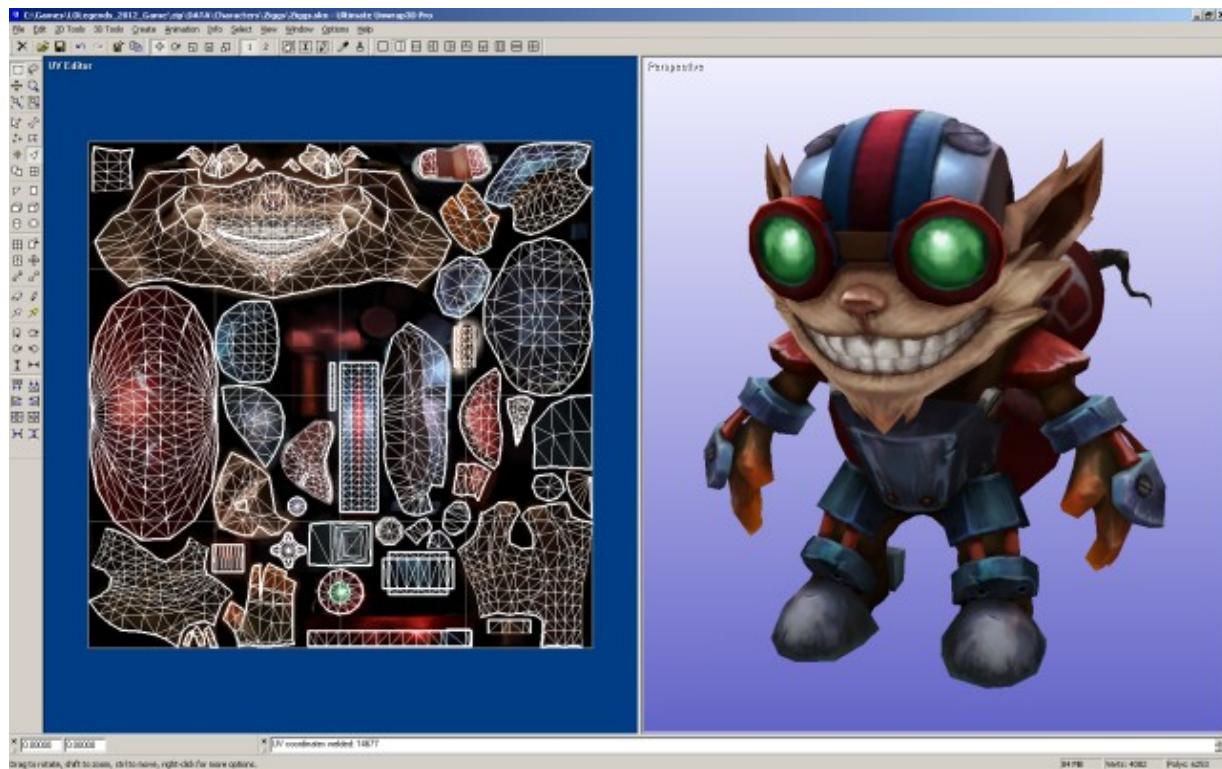


Figure 10.24: Initial Shader Graph results

As you can see, the behavior of the shader varies according to the properties you set in the **Master** node, but so far, doing this is no different than creating an Unlit Shader and setting up its properties; the real power of Shader Graph is when you use nodes that do specific calculations as inputs of the Master node. We will start looking at the texturing nodes, which allow us to apply Textures to our model.

Using Textures

The idea of using Textures is to have an image applied to the model in a way that we can paint different parts of the models with different colors. Remember that the model has a UV map, which allows Unity to know which part of the Texture will be applied to which part of the model:

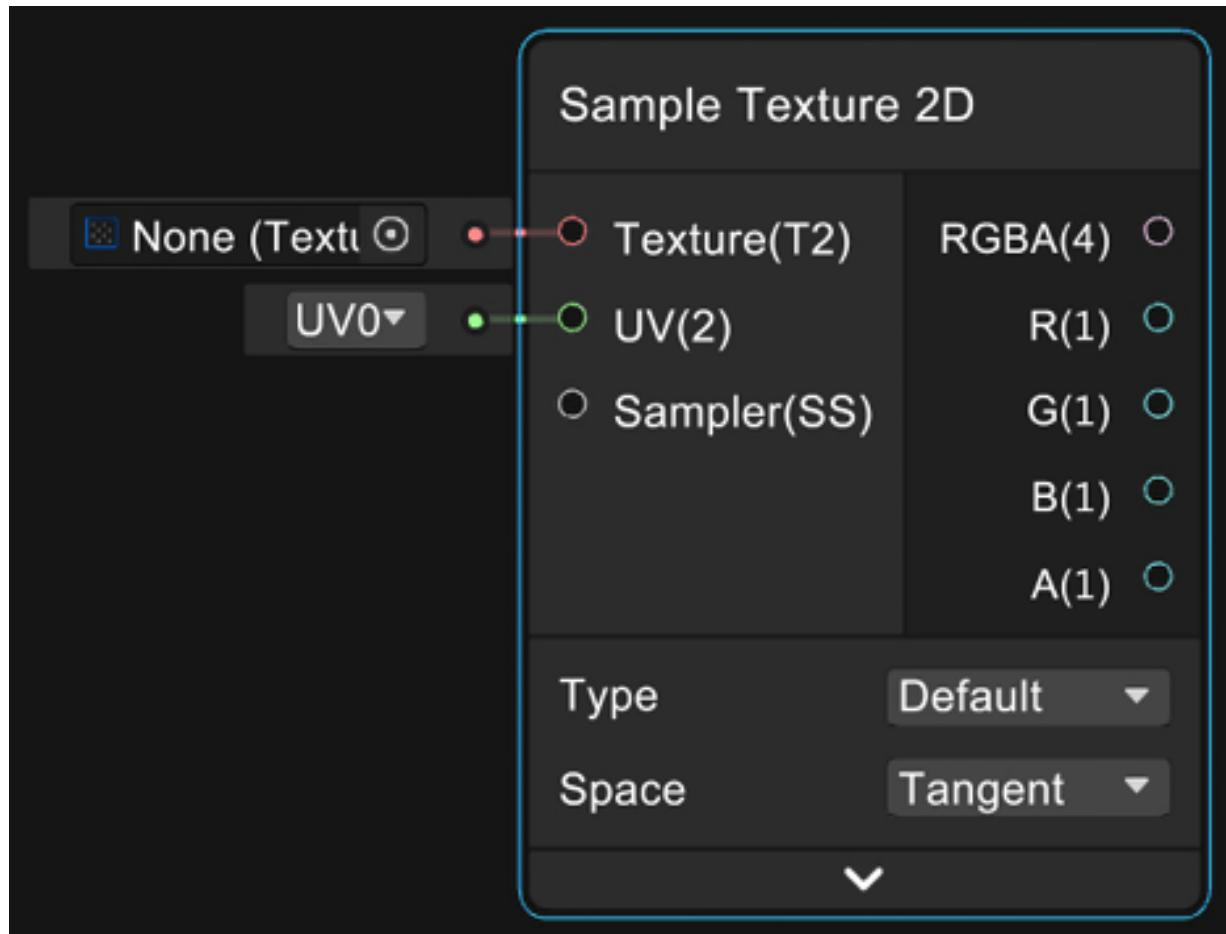


Figure 10.25: On the left, a face Texture; on the right, the same texture applied to a face mesh

Like in Visual Scripting, we will use interconnected nodes in our shader graph that will execute specific shader operations. We have several nodes to do this task, one of them being Sample Texture 2D, a node that has two main inputs. First, it asks us for the texture to sample or apply to the model, and then for the UV. You can see it in the following screenshot:



Figure 10.26: Sample Texture 2D node

As you can see, the default value of the **Texture** input node is **None**, so there's no texture by default, and we need to manually specify that. For **UV**, the default value is `UV0`, meaning that, by default, the node will use the main UV channel of the model, and yes, a model can have several UVs set. For now, we will stick with the main one. If you are not sure what that means, `UV0` is the safest option. Let's try this node, doing the following:

1. Download and import a **tileable water texture** from the internet:



Figure 10.27: Water tileable Texture

2. Select the Texture and be sure that the **Wrap Mode** property of the Texture is set to **Repeat**, which will allow us to repeat the Texture as we did in the terrain because the idea is to use this shader to cover large water areas:

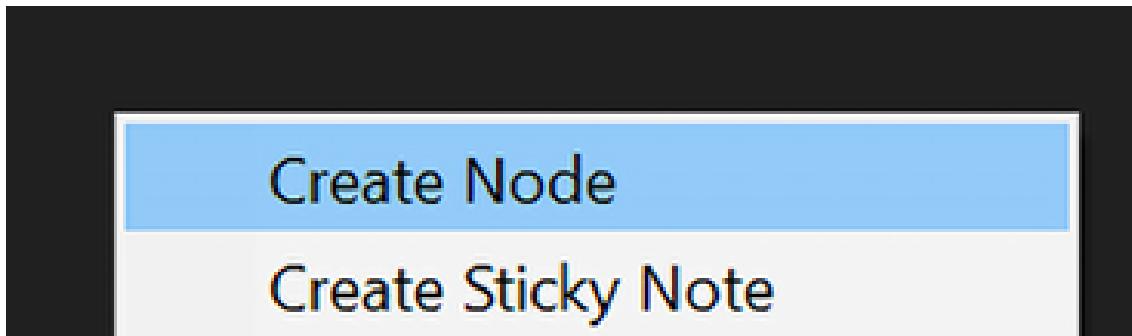


Figure 10.28: Texture Repeat mode

3. In the **Water Shader Graph**, right-click in an empty area of the **Shader Graph** and select **Create Node**:

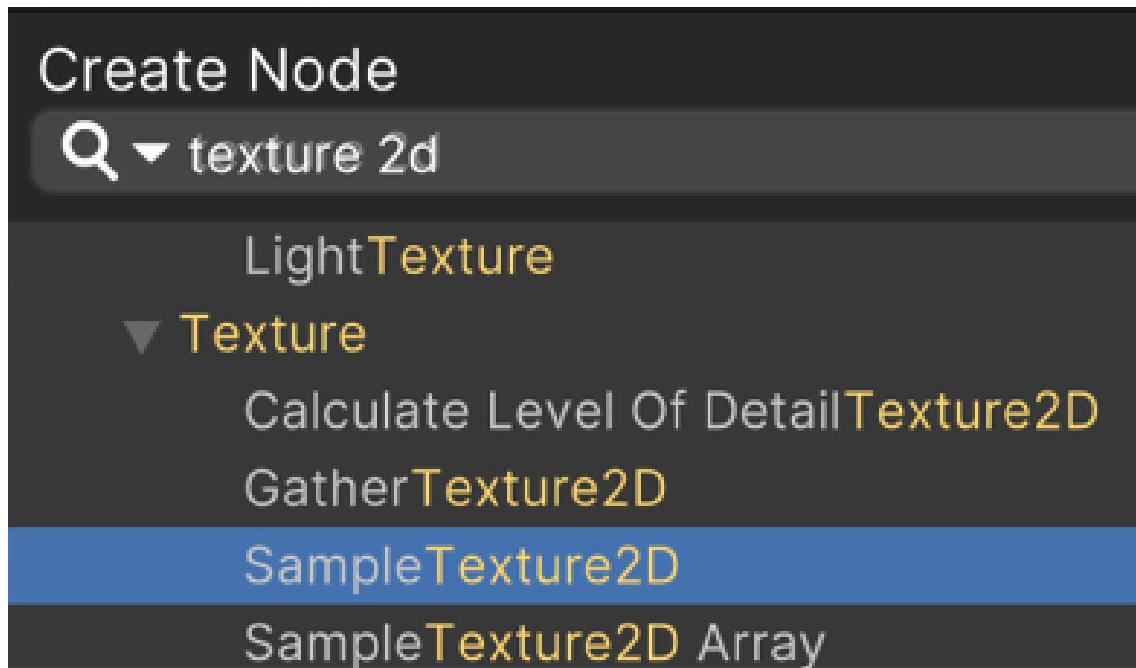


Figure 10.29: Shader Graph Create Node option

4. In the **Search** box, write `Sample texture` and all of the sampler nodes will show up. Double-click **Sample Texture 2D**. If for some reason you can't double-click the option, right-click on it first and then try again. There is a known bug on this tool and this is the workaround:

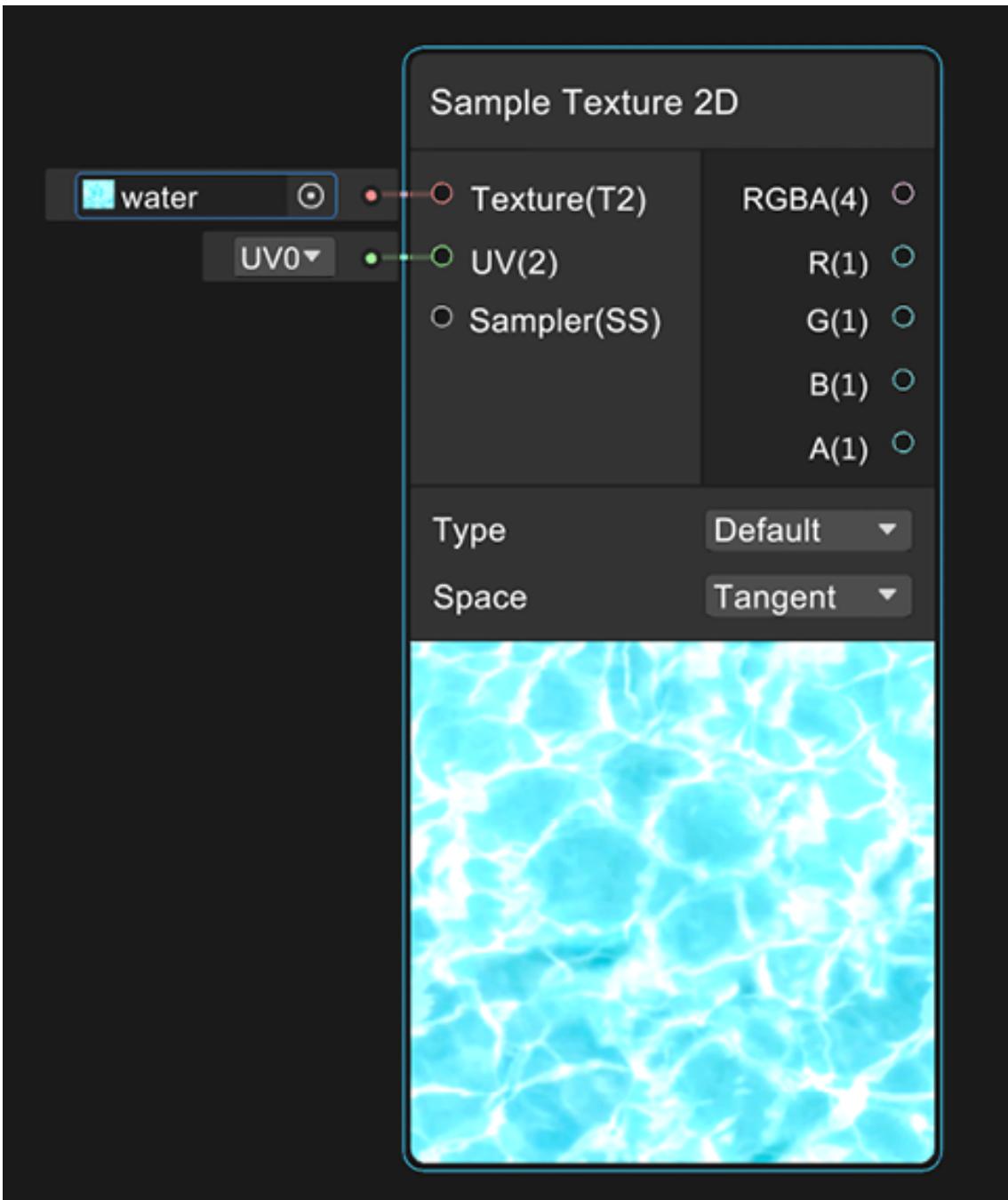


Figure 10.30: Sample texture node search

5. Click in the circle to the left of the **Texture** input pin of the **Sample Texture 2D** node. It will allow us to pick a Texture to sample—just select the water one. You can see that the Texture can be previewed in the bottom part of the node:

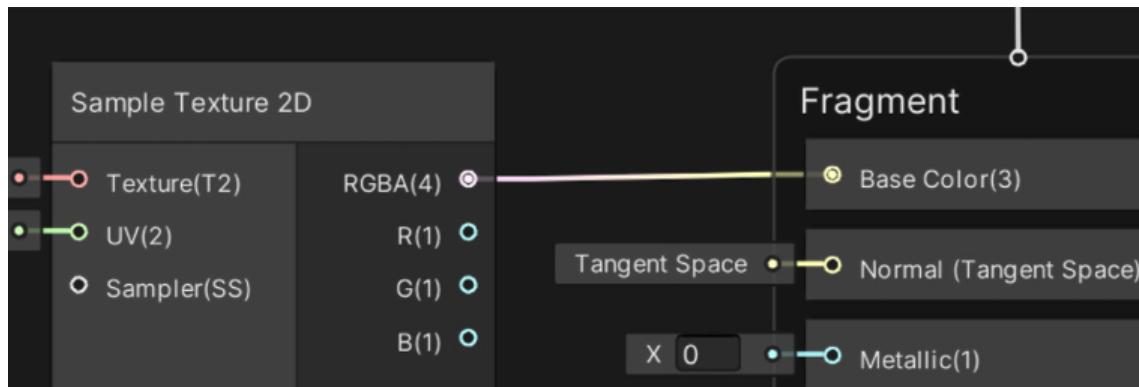


Figure 10.31: Sample Texture node with a Texture in its input pin

6. Drag the output pin **RGBA** from the **Sample Texture 2D** node to the **Base Color** input pin of the Master node:

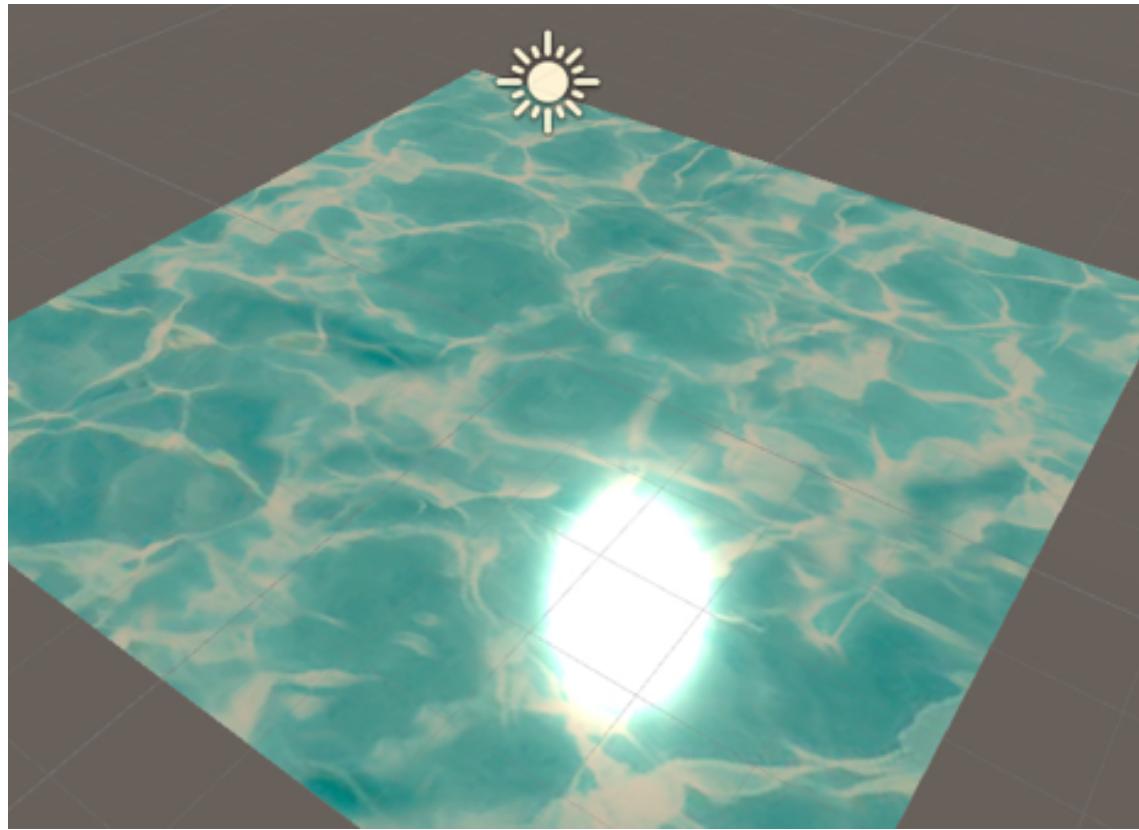


Figure 10.32: Connecting the results of a Texture sampling with the Base Color pin of the Master node

7. Click the **Save Asset** button at the top-left part of the Shader Graph editor and see the changes in the Scene view:

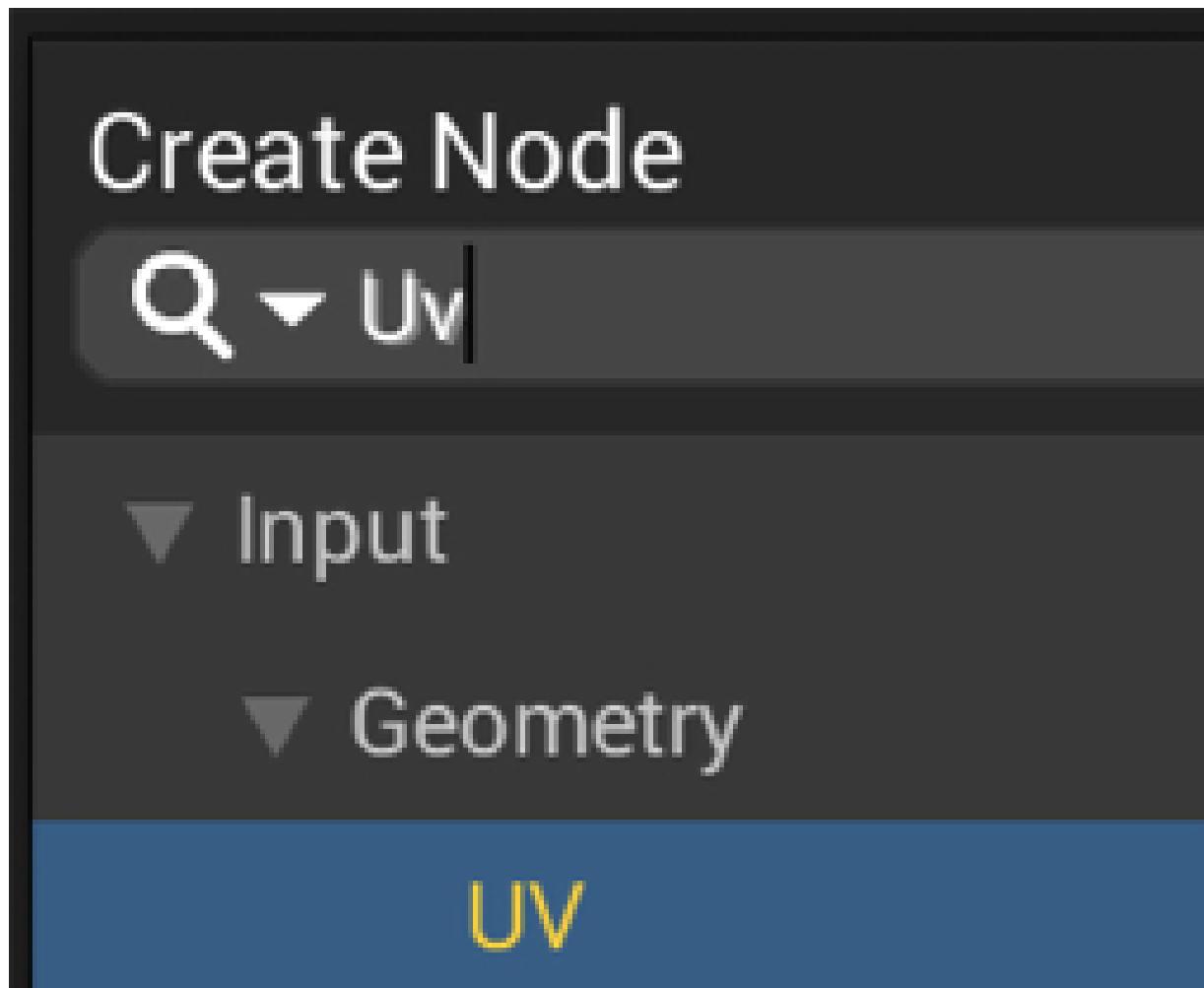


Figure 10.33: Results of applying a Texture in our Shader Graph

As you can see, the Texture is properly applied to the model, but if you take into account that the default plane has a size of 10x10 meters, the ripples of the water seem too big. So, let's tile the Texture! To do this, we need to change the UVs of the model, making them bigger. You may imagine that bigger UVs mean the Texture should also get bigger, but take into account that we are not making the object bigger; we are just modifying the UV. In the same object area, we will display more of the texture area, meaning that in the

bigger texture sample area (achieved by bigger UVs), repetitions of the texture may appear. To do so, follow the next steps:

1. Right-click in any empty space and click **New Node** to search for the UV node:



Figure 10.34: Searching for the UV node

2. Using the same method, create a **Multiply** node.
3. Drag the **Out** pin of the **UV** node to the **A** pin of the **Multiply** node to connect them.
4. Set the **B** pin input value of **Multiply** to $4, 4, 4, 4$:

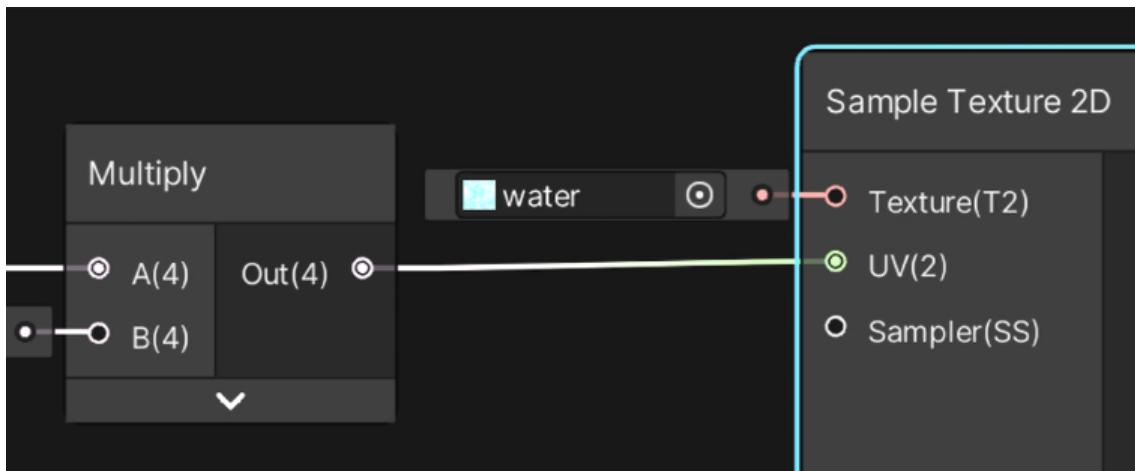


Figure 10.35: Multiplying the UVs by 4

5. Drag the **Out** pin of the **Multiply** node to the **UV** of the **Sample Texture 2D** node to connect them:

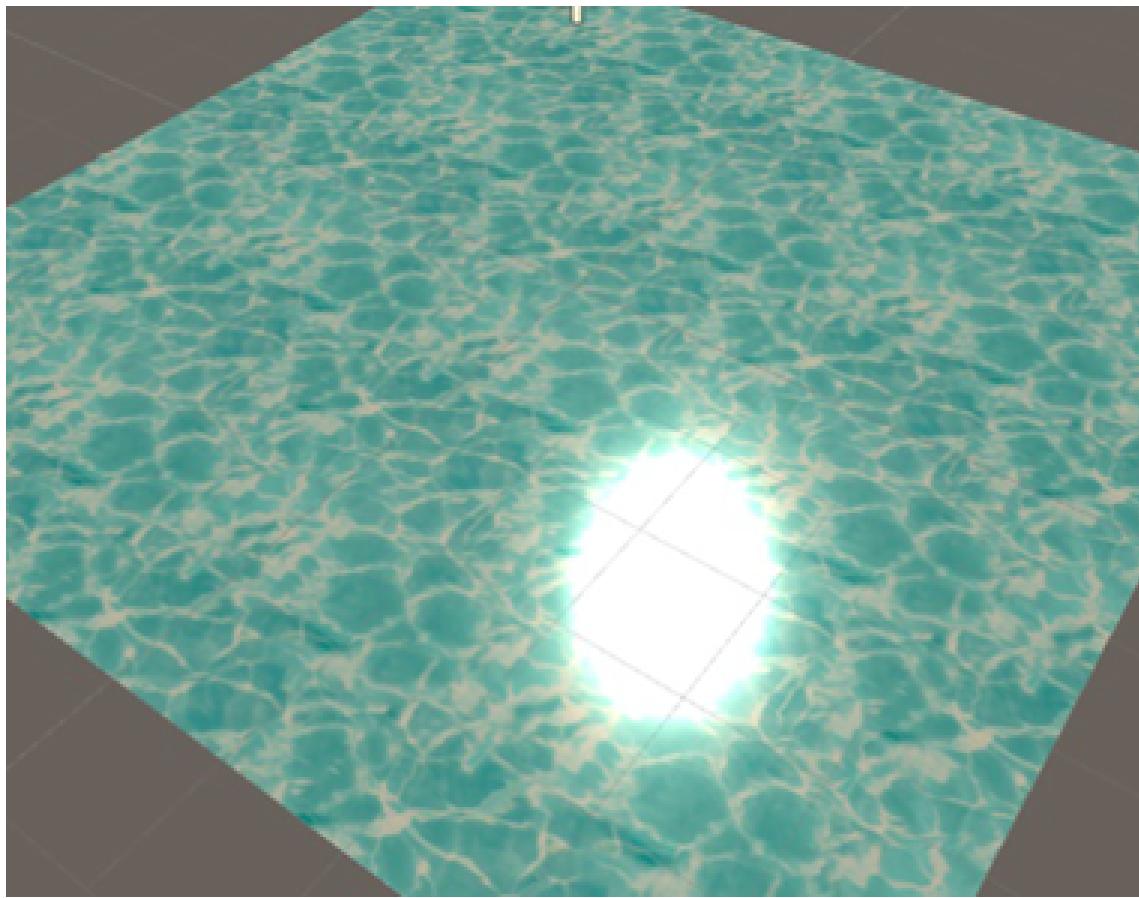


Figure 10.36: Using the multiplied UVs to sample the Texture

6. If you save the graph and go back to the Scene view, you can see that now the ripples are smaller, because we have tiled the UVs of our model. You can also see that in the preview of the **Sampler Texture 2D** node:

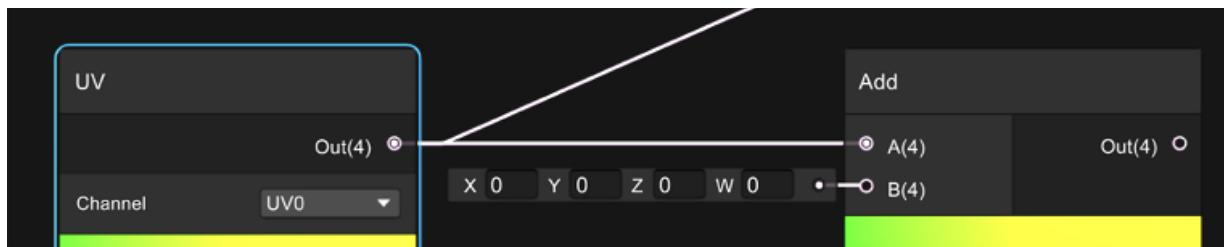


Figure 10.37: Results of the model's UV multiplication

Another interesting effect we can do now is to apply an offset to the Texture to move it. The idea is that even if the plane is not actually moving, we will simulate the flow of the water through it, moving just the Texture. Remember, the responsibility of determining the part of the Texture to apply to each part of the model belongs to the UV, so if we add values to the UV coordinates, we will be moving them, generating a Texture sliding effect. To do so, let's do the following:

1. Create an **Add** node to the right of the **UV** node.
2. Connect the **Out** pin of the **UV** to the **A** pin of the **Add** node:

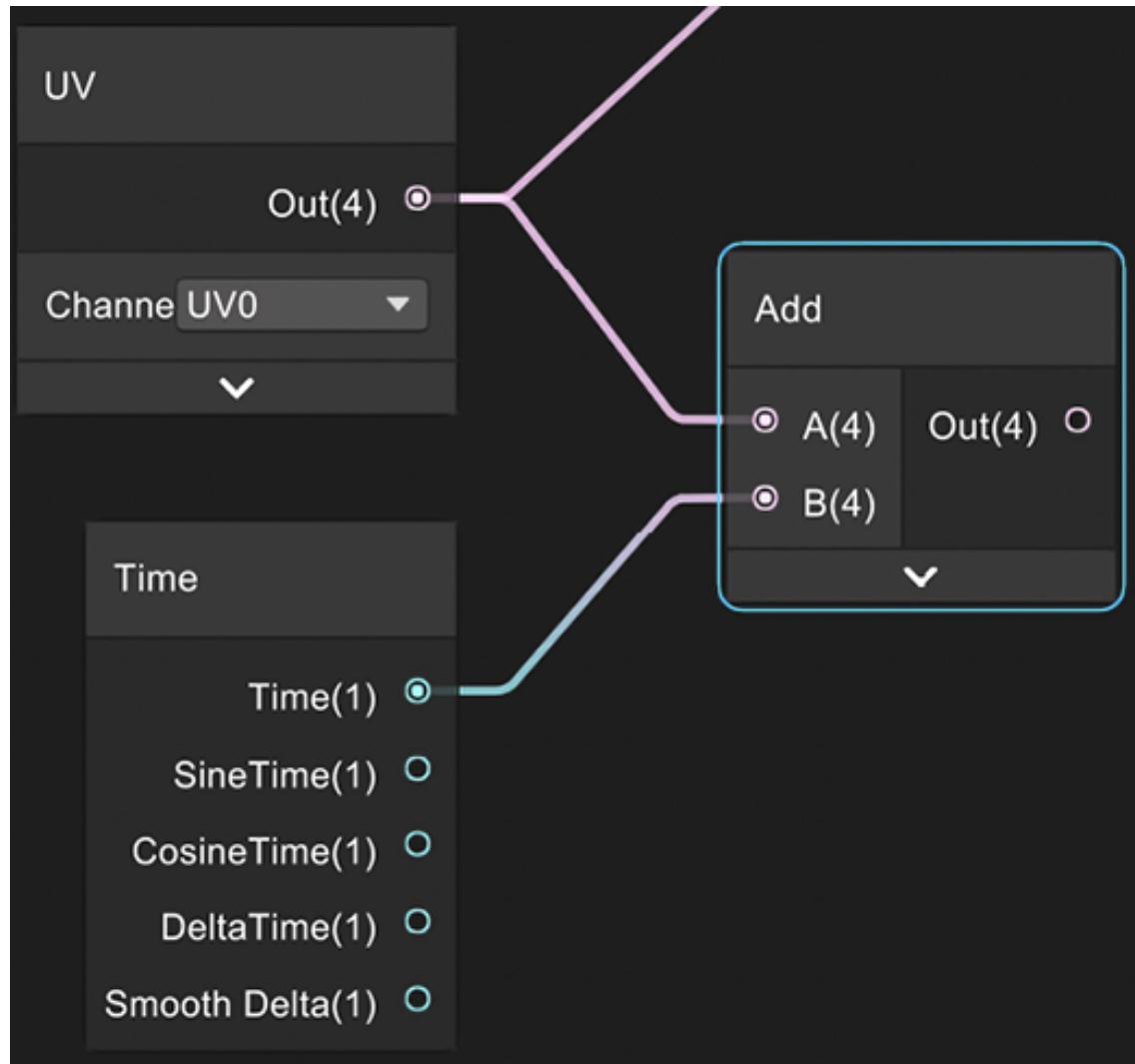


Figure 10.38: Adding values to the UVs

3. Create a **Time** node at the left of the **Add** node.
4. Connect the **Time** node to the **B** pin of the **Add** node:

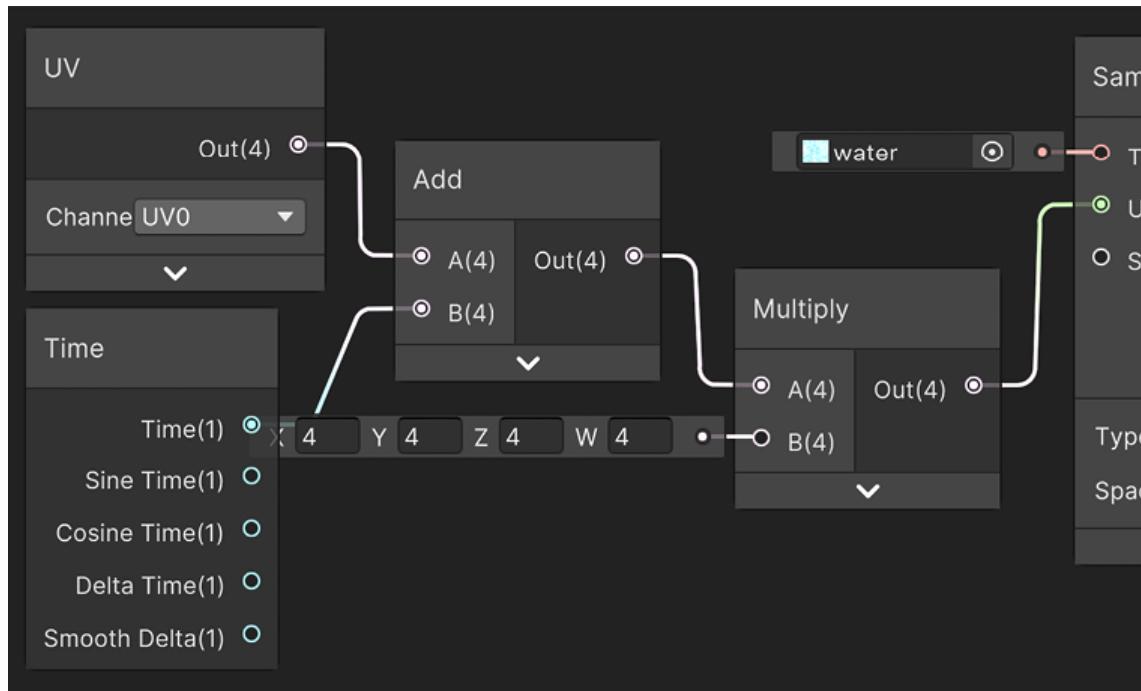


Figure 10.39: Adding time to the UVs

5. Connect the Out pin of the Add node to the A input pin of the Multiply node:

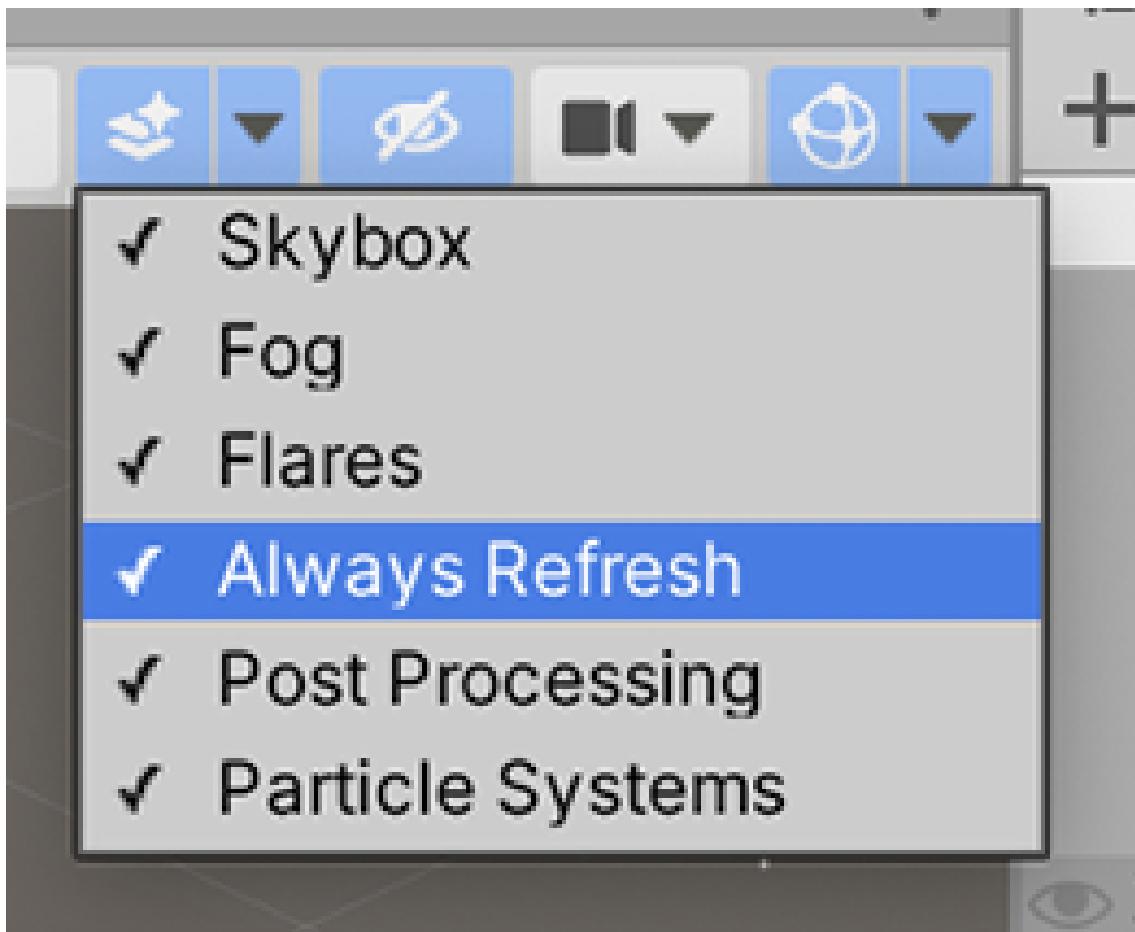


Figure 10.40: Added and multiplied UVs as an input of the sample Texture

6. Save and see the water moving in the Scene view. If you don't see it moving, click the layers icon in the top bar of the scene and check **Always Refresh**:

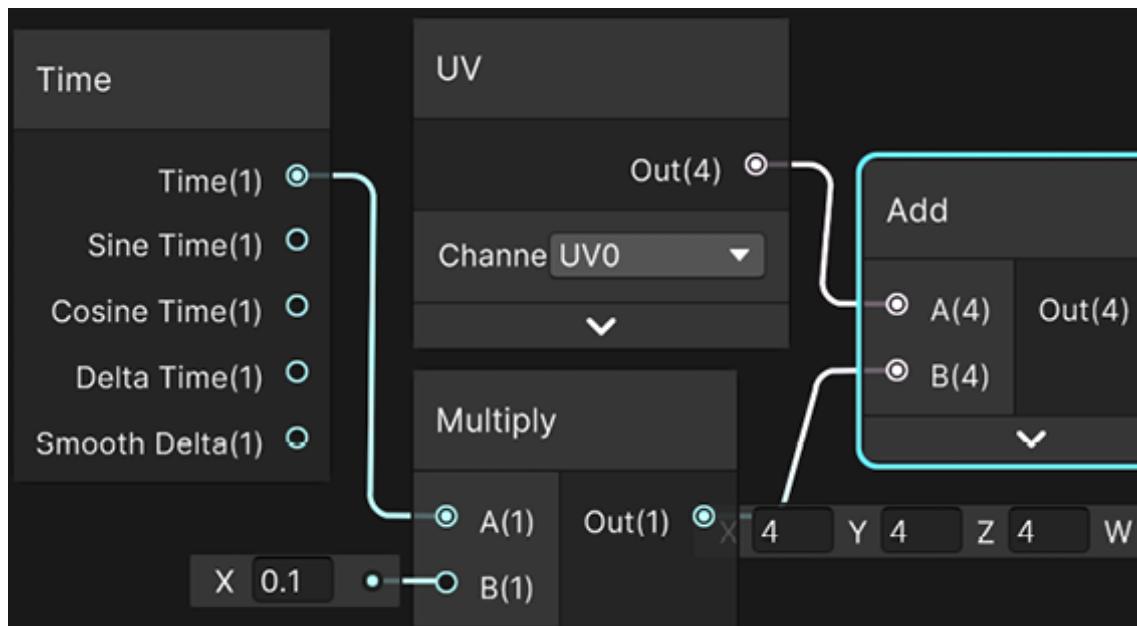


Figure 10.41: Enabling Always Refresh to preview the effect

7. If you feel the water is moving too fast, try using the multiplication node to make the time a smaller value. I recommend you try it by yourself before looking at the next screenshot, which has the answer:



Figure 10.42: Multiplication of time to move the texture slower

8. If you feel the graph is too big, try to hide some of the node previews by clicking on the **up (^)** arrow that appears on the preview when you move the mouse over it:

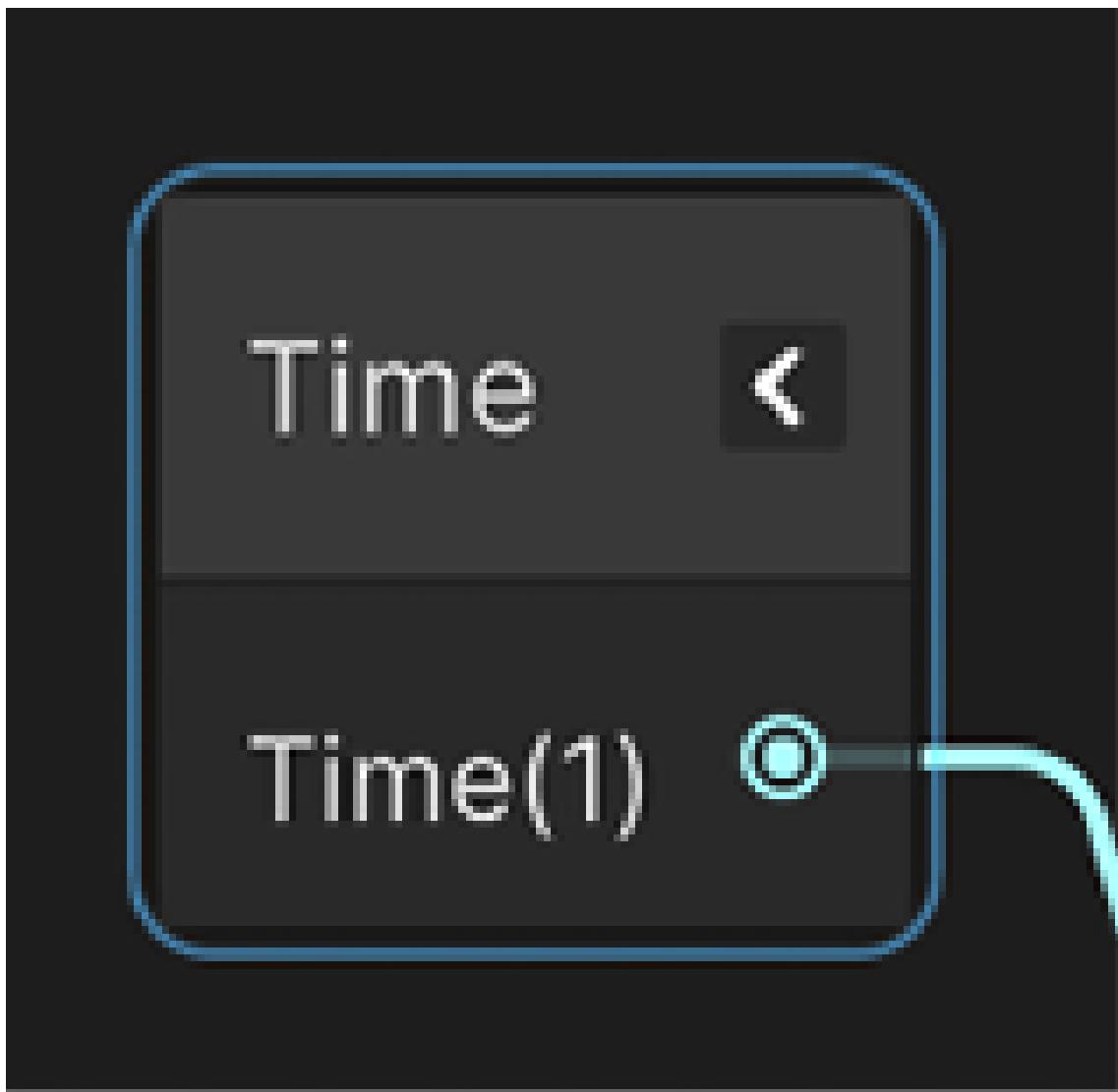


Figure 10.43: Hiding the preview from the graph nodes

9. Also, you can hide unused pins by selecting the node and clicking the arrow at its top right:

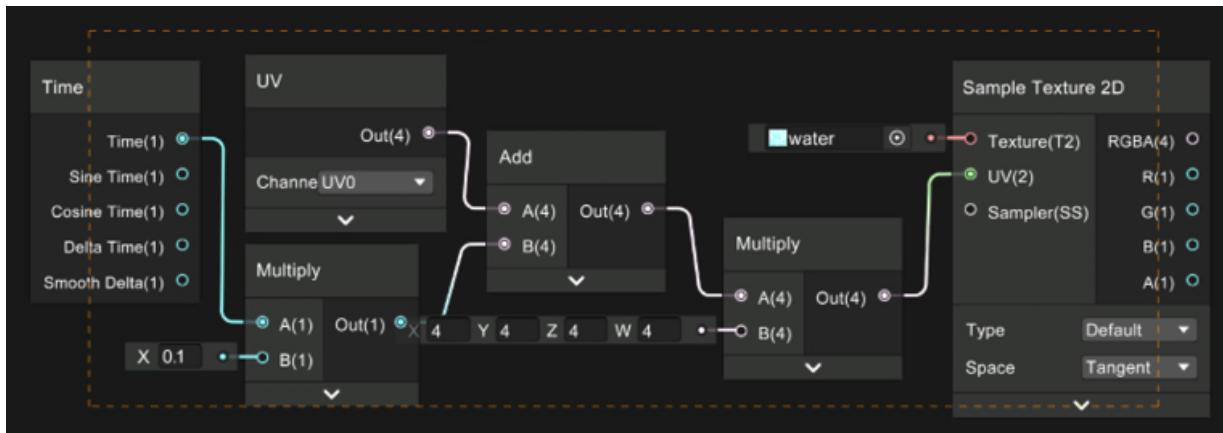


Figure 10.44: Hiding unused pins from the graph nodes

So, to recap, first we added the time to the UV to move it and then multiplied the result of the moved UV to make it bigger to tile the Texture. It is worth mentioning that there's a **Tiling and Offset** node that does all of this process for us, but I wanted to show you how a simple multiplication to scale the UV and an add operation to move it generates a nice effect; you can't imagine all of the possible effects you can achieve with other simple mathematical nodes! Actually, let's explore other usages of mathematical nodes to combine Textures in the next section.

Info

When learning DirectX, making shaders was harder given you needed to learn a less user-friendly language called HLSL. While for most of the cases Shader Graph is all you need, I don't regret at all learning such shader languages, as they have access to more advanced features that usually node-based shading languages doesn't, and they gave you a deeper understanding of the internals of the GPU. For more info about how to create code-based shaders in URP you can check this:

<https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@15.0/manual/writing-custom-shaders-upr.html>

Combining Textures

Even though we have used nodes, we haven't created anything that can't be created using regular shaders, but that's about to change. So far, we can see the water moving but it stills look static, and that's because the ripples are always the same. We have several techniques to generate ripples, and the simplest one would be to combine two water Textures moving in different directions to mix their ripples, and actually, we can simply use the same Texture just flipped to save some memory. To combine the Textures, we will sum them and then divide them by 2, so basically, we are calculating the average of the textures! Let's do that by doing the following:

1. Select all of the nodes between **Time** and **Sampler 2D** (including them) creating a selection rectangle by clicking in any empty space in the graph, holding and dragging the click, and then releasing when all target nodes are covered:

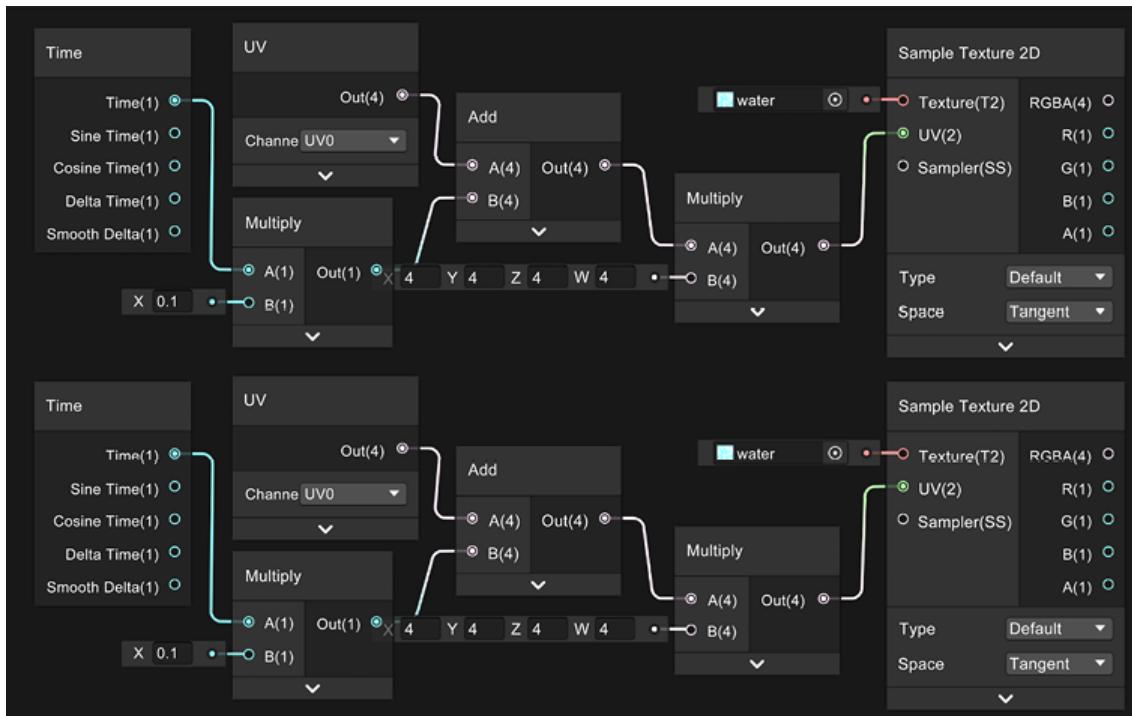


Figure 10.45: Selecting several nodes

2. Right-click and select **Copy**, and then again right-click and select **Paste**, or use the classic *Ctrl + C, Ctrl + V* commands (*Command + C, Command + V* on Mac).

3. Move the copied nodes below the original ones:

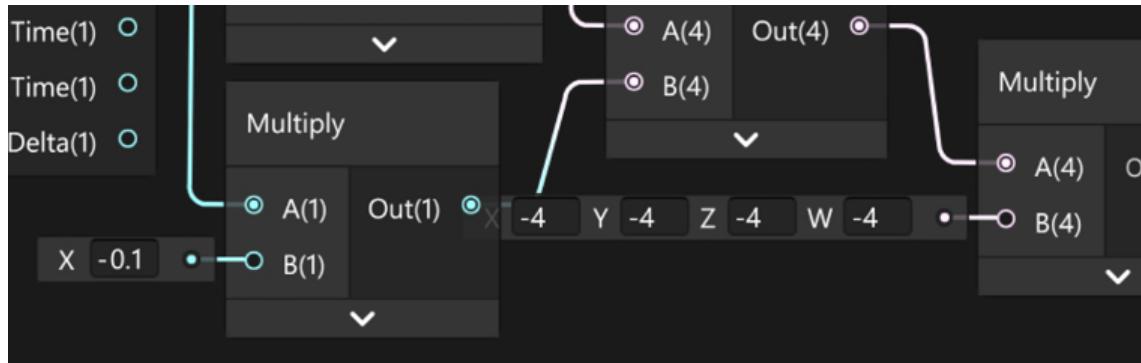


Figure 10.46: Duplication of nodes

4. For the copied nodes, set the **B** pin of the **Multiply** node connected to **Sample Texture 2D** to $-4, -4, -4, -4$. You can see that that flipped the texture.

5. Also, set the **B** pin of the **Multiply** node connected to the **Time** node to -0.1 :

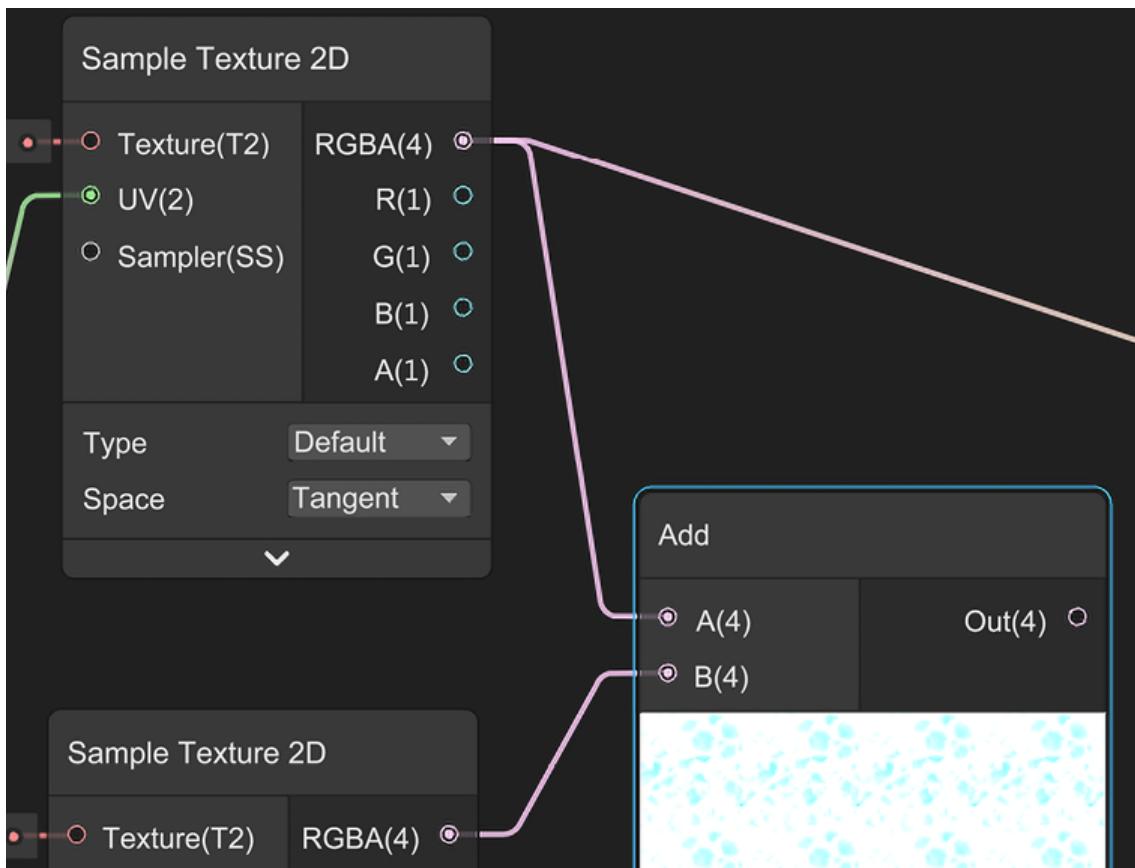


Figure 10.47: Multiplication of values

6. Create an **Add** node at the right of both **Sampler Texture 2D** nodes and connect the outputs of those nodes to the **A** and **B** input pins of the **Add** node:

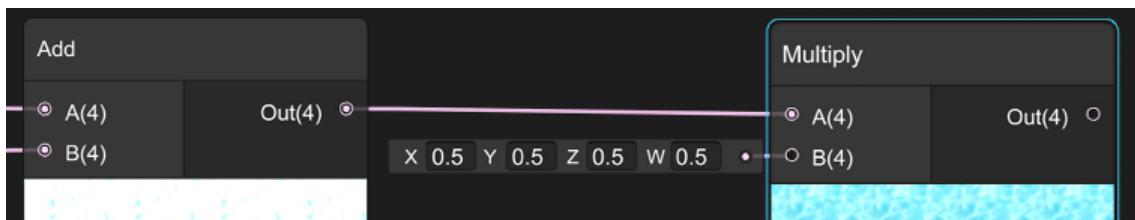


Figure 10.48: Adding two Textures

7. You can see that the resulting combination is too bright because we have summed up the intensity of both textures, so let's fix that by multiplying the **Out** of the **Add** node by

`0.5, 0.5, 0.5, 0.5`, which will divide each resulting color channel by 2, averaging the color. You can also experiment with what happens when you set different values to each channel if you want, but for our purposes, `0.5` is the proper value for each channel:

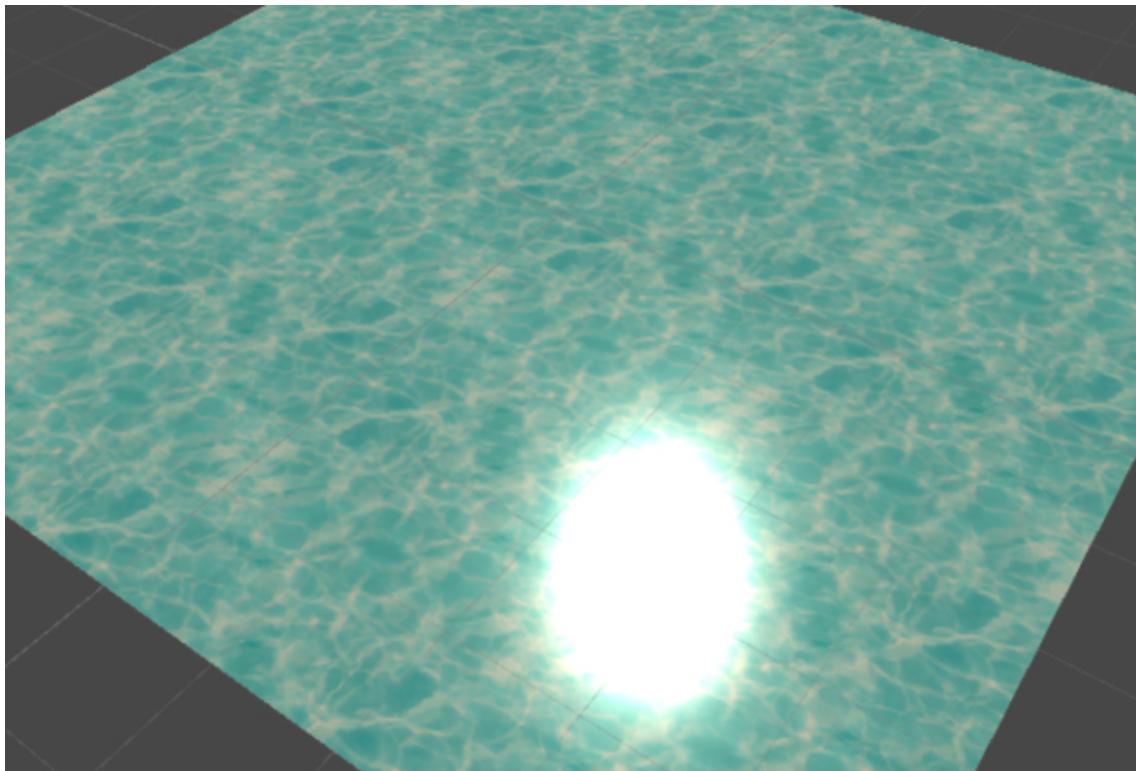


Figure 10.49: Dividing the sum of two Textures to get the average

8. Connect the **Out** pin of the **Multiply** node to the **Base Color** pin of the **Master** node to apply all of those calculations to the color of the object.
9. Save the **Asset** and see the results in the Scene view:

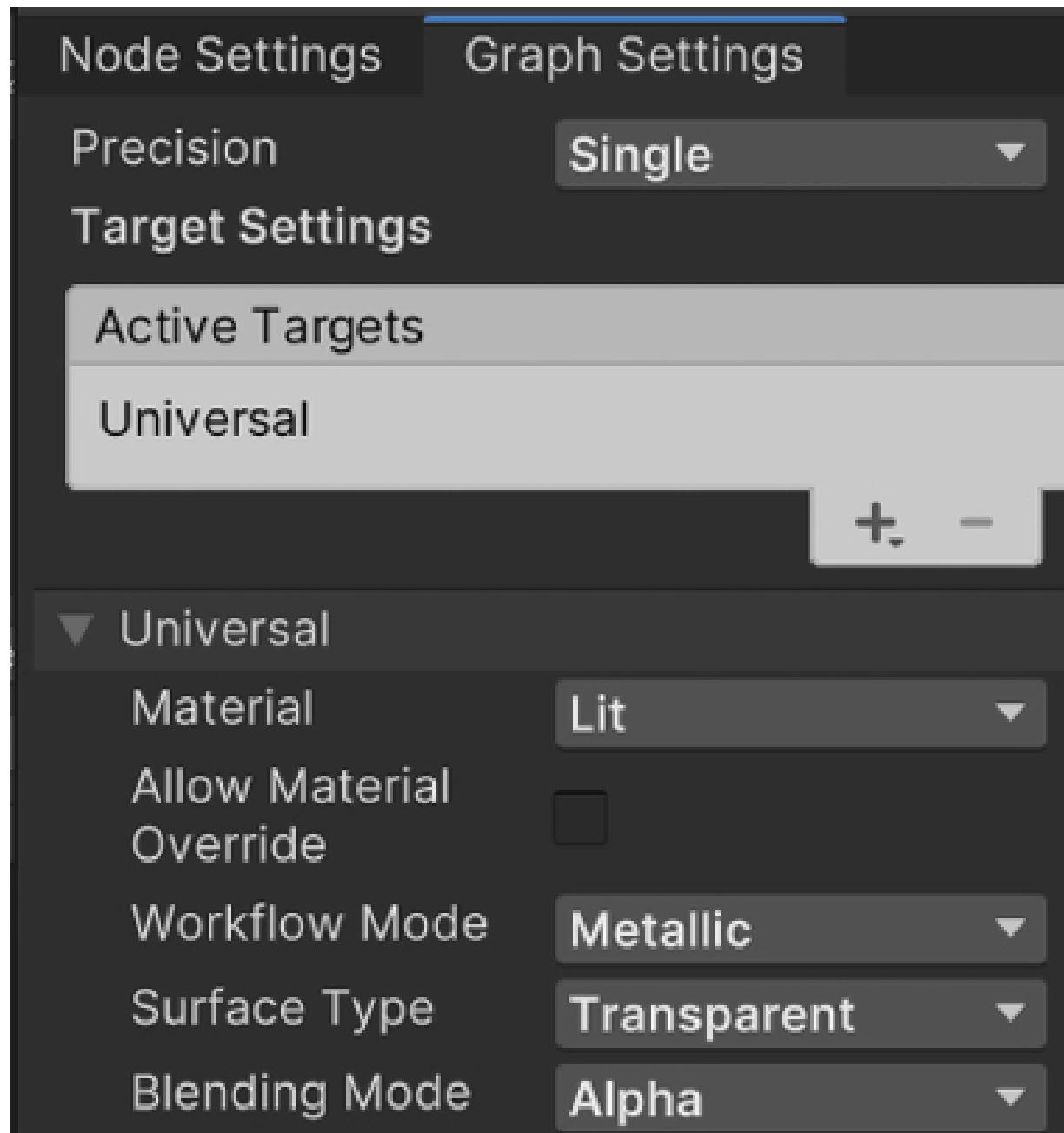


Figure 10.50: Results of texture blending

You can keep adding nodes to make the effect more diverse, such as using **Sine** nodes (which will execute the trigonometry sine function) to apply non-linear movements and so on, but I will let you learn that by experimenting with this by yourself. For now, we will stop here. As always, this topic deserves a full book, and the intention of this chapter is to give you a small taste of this powerful

Unity tool. I recommend you look for other Shader Graph examples on the internet to learn other usages of the same nodes and, of course, new nodes. One thing to consider here is that everything we just did is basically applied to the Fragment Shader stage of the Shader Pipeline we discussed earlier. Now, let's use the Blending Shader stage to apply some transparency to the water.

Info

For more examples of shader graphs, i recommend checking the following link:

<https://docs.unity3d.com/Packages/com.unity.shadergraph@17.0/manual/ShaderGraph-Samples.html>

Applying transparency

Before declaring our effect finished, a little addition we can do is to make the water a little bit transparent. Remember that the Shader Pipeline has a blending stage, which has the responsibility of blending each pixel of our model into the image being rendered in this frame. The idea is to make our Shader Graph modify that stage to apply **Alpha Blending**, a blending mode that combines our model and the previously rendered models based on the Alpha value of our model. To get that effect, take the following steps:

1. Look for the **Graph Inspector** window floating around. If you don't see it, click the **Graph Inspector** button at the top-right part of the Shader Graph editor. Try also expanding the shader graph window to display it in case its hidden behind the right bound of the window.
2. Click the **Graph Settings** tab.
3. Set the **Surface Type** property to **Transparent**.
4. Set the **Blending Mode** property to **Alpha** if it isn't already at that value:



Figure 10.51: Graph Inspector Transparency settings

5. Set the Alpha input pin of the Master to `0.5`.

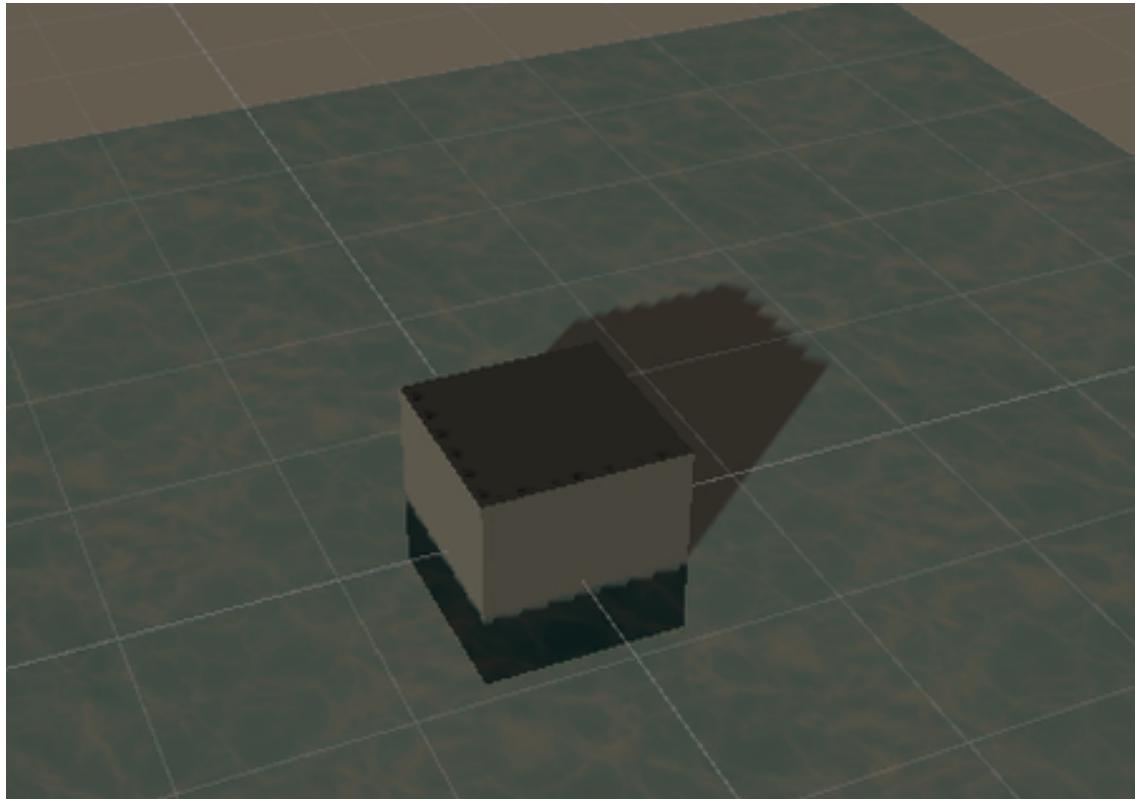


Figure 10.52: Setting Alpha of the Master node

6. Save the Shader Graph and see the transparency being applied in the Scene view. If you can't see the effect, just put a cube into the water to make the effect more evident:

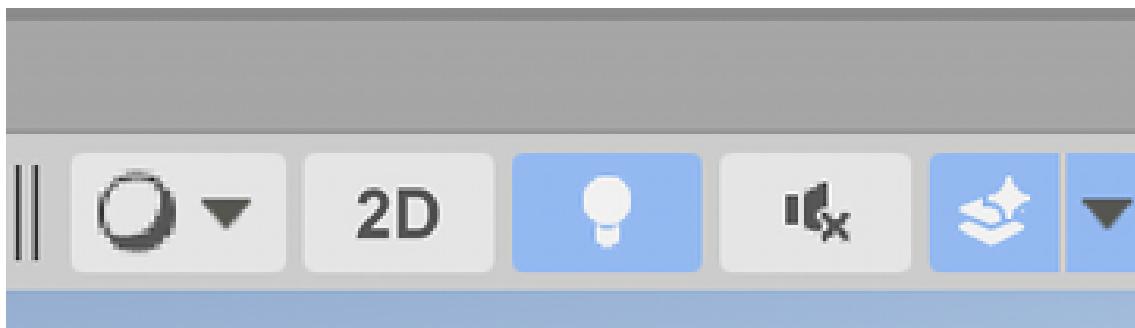


Figure 10.53: Shadows from the water being applied to a cube

7. You can see the shadows that the water is casting on our cube because Unity doesn't know the object is transparent and hence casts shadows. Click on the water plane and look for the Mesh Renderer component in the Inspector. If you don't see the shadow, click the lightbulb at the top of the Scene view.

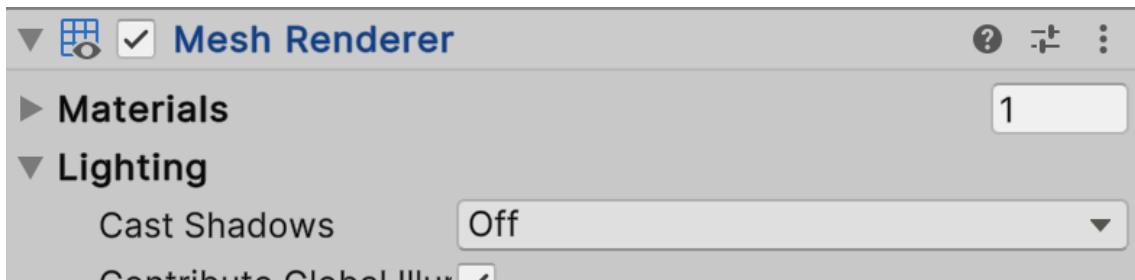


Figure 10.54: Enabling lights in the Scene View

8. In the **Lighting** section, set **Cast Shadows** to **Off**; this will disable shadow casting from the plane on the parts of the cube that are underwater:

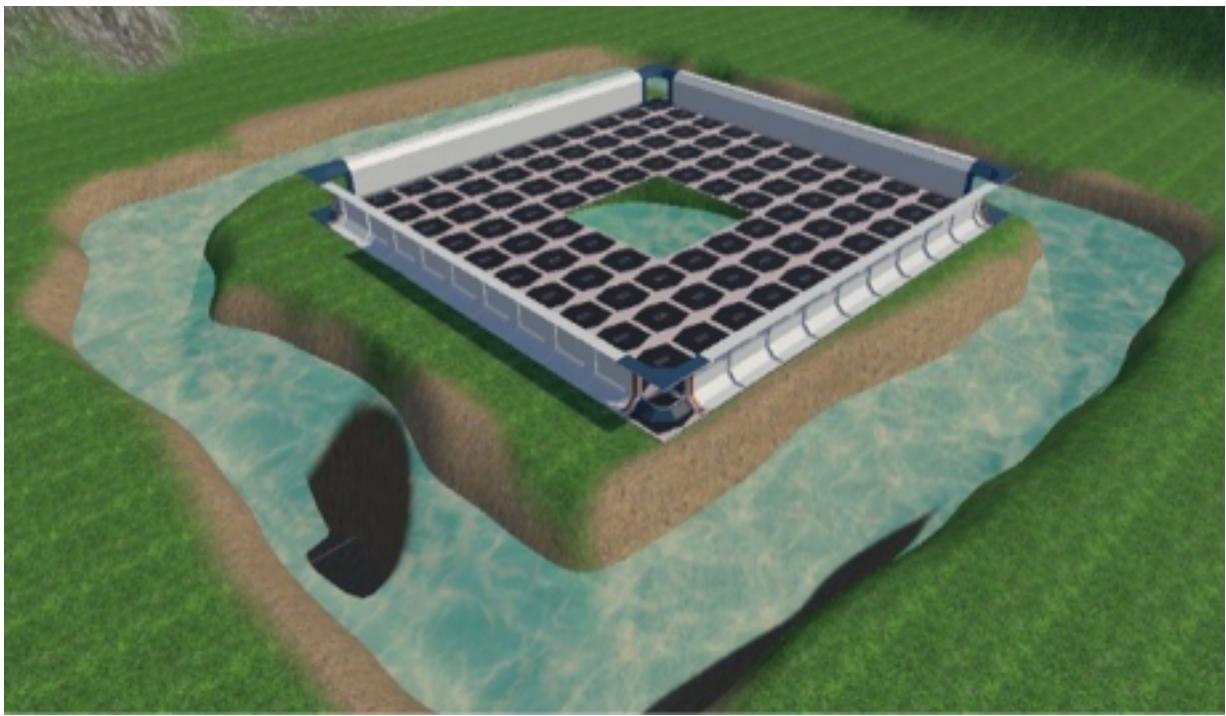


Figure 10.55: Disabling shadow casting

Adding transparency is a simple process but it has its caveats, like the shadow problem, and in more complex scenarios, it can have other problems, like increasing overdraw, meaning the same pixel needs to be drawn several times (the pixel belonging to the transparent object, and one of the objects behind). I would suggest you avoid using transparency unless it is necessary. Actually, our water can live without transparency, especially when we apply this water to the river basin around the base because we don't need to see the part under the water, but the idea is for you to know all of your options. In the next screenshot, you can see how we have put a giant plane with this effect below our base, big enough to cover the entire basin:

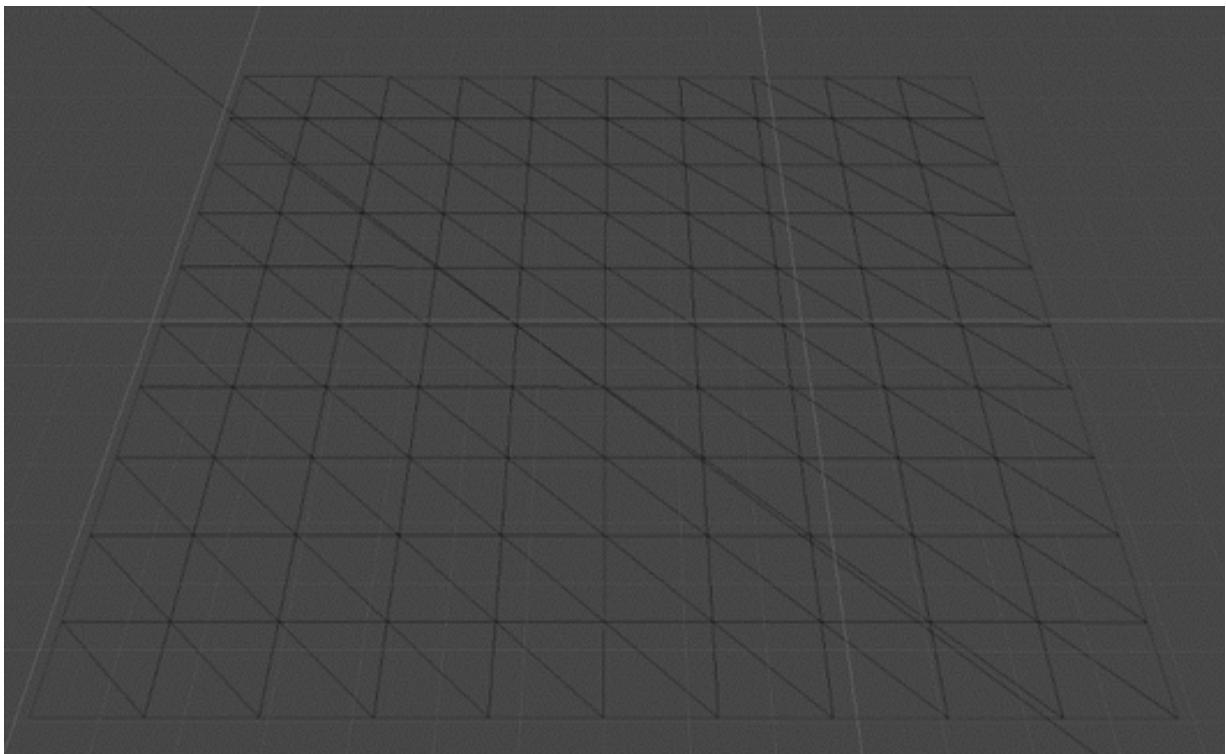


Figure 10.56: Using our water in the main scene

Memory

I have a friend that always jokes about me saying that everything can be solved with a shader, but leaving jokes aside, it is an extremely useful tool if used cleverly. In the past, developers used shaders to do non-related graphics processing, like mathematics simulations, reading the generated pixels as the needed results. That lead to what today is known as Compute Shaders, which is essentially running custom programs on a GPU to do calculations, leveraging the power of the GPU. Of course, Unity supports Compute Shaders, you can learn more about it here:

<https://docs.unity3d.com/Manual/class-ComputeShader.html>

From now on we can do plenty of things with our shader. We can think about simulating water foam for the pixels that are higher

than certain height, leveraging the vertex animation we added. We could also change the water scrolling direction via scripting or using sine nodes, the sky is the limit! Now that we have modified how the object looks through the **Fragment** node section, let's discuss how to use the **Vertex** section to apply a mesh animation to our water.

Creating Vertex Effects

So far, we have applied water textures to our water, but it's still a flat plane. We can go further than that and make the ripples not only via textures but also by animating the mesh. To do so, we will apply the noise texture we used at the beginning of the chapter in the shader, but instead of using it as another color to add to the **Base Color** of the shader, we will instead use it to offset the **Y** position of the vertexes of our plane. Due to the chaotic nature of the noise texture, the idea is that we will apply a vertical offset to different parts of the model, so we can emulate the ripples:

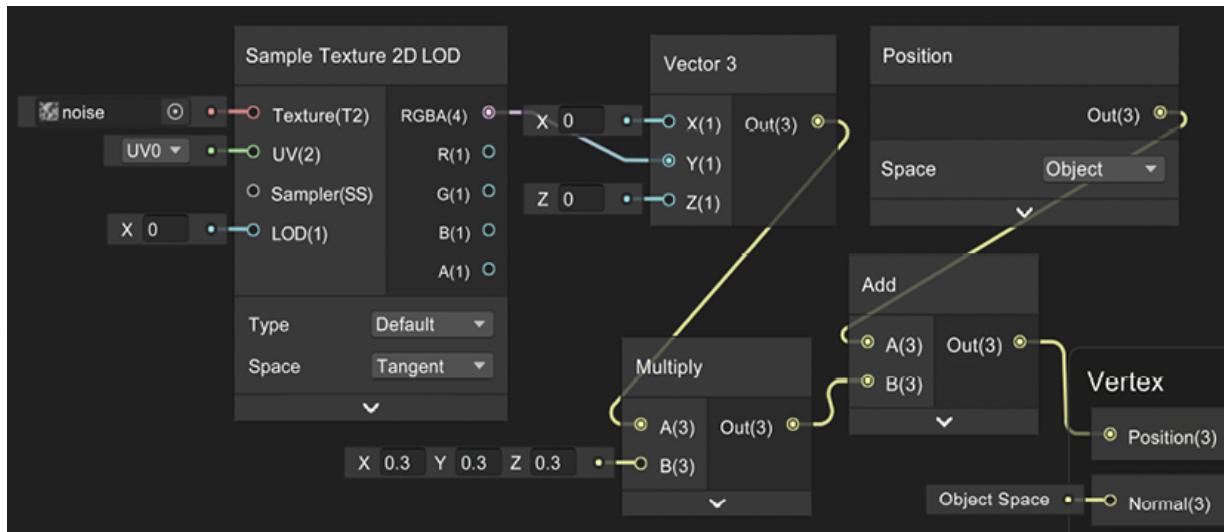


Figure 10.57: Default plane mesh subdivided into a grid of 10x10 with no offset

To accomplish something like this, you can modify the **Vertex** section of your shader to look like the following:

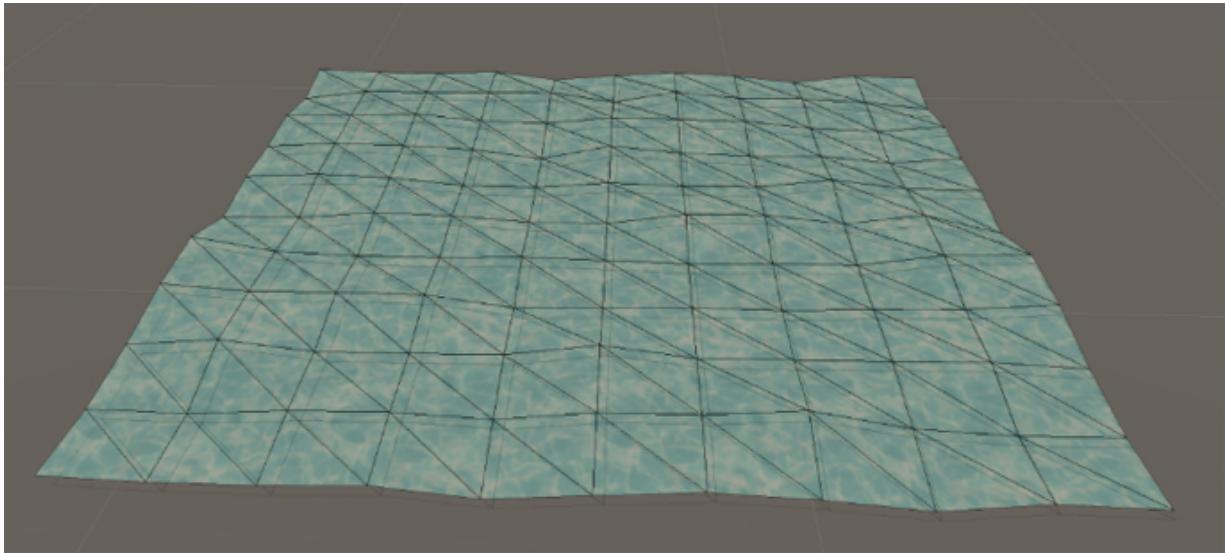


Figure 10.58: Ripples vertex effect

In the graph, you can see how we are creating a **Vector** whose *y* axis depends on the noise Texture we downloaded at the beginning of the chapter. The idea behind that is to create a **Vector** pointing upward whose length is proportional to the grayscale factor of the texture; the whiter the pixel of the texture, the longer the offset. This texture has an irregular yet smooth pattern so it can emulate the behavior of the tide. Please notice that here we used **Sample Texture 2D LOD** instead of **Sample Texture 2D**; the latter does not work in the **Vertex** section, so keep that in mind. Then we multiply the result by *0.3* to reduce the height of the offset to add, and then we add the result to the **Position** node. See that the **Space** property of the **Position** node is set to **Object** mode. We need that mode to work with the **Vertex** section of the Shader Graph (we discussed World and Local spaces before in *Chapter 2, Editing Scenes and GameObjects* but you can also search [Object vs World Space](#) on the internet for more info about this). Finally, the result is connected to the **Position** node of the **Vertex** section. If you save, you will see something like the following image:

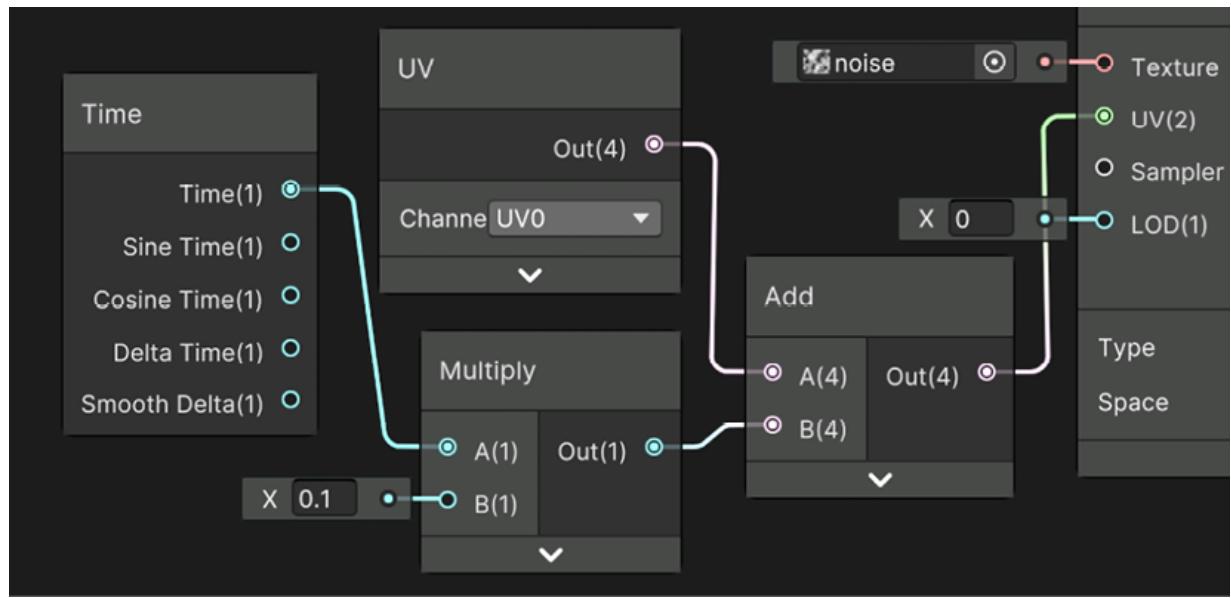


Figure 10.59: Ripples vertex effect applied

Of course, in this case, the ripples are static because we didn't add any time offset to the UV as we did before. In the following screenshot, you can see how to add that, but before looking at it I recommend you try to resolve it first by yourself as a personal challenge:

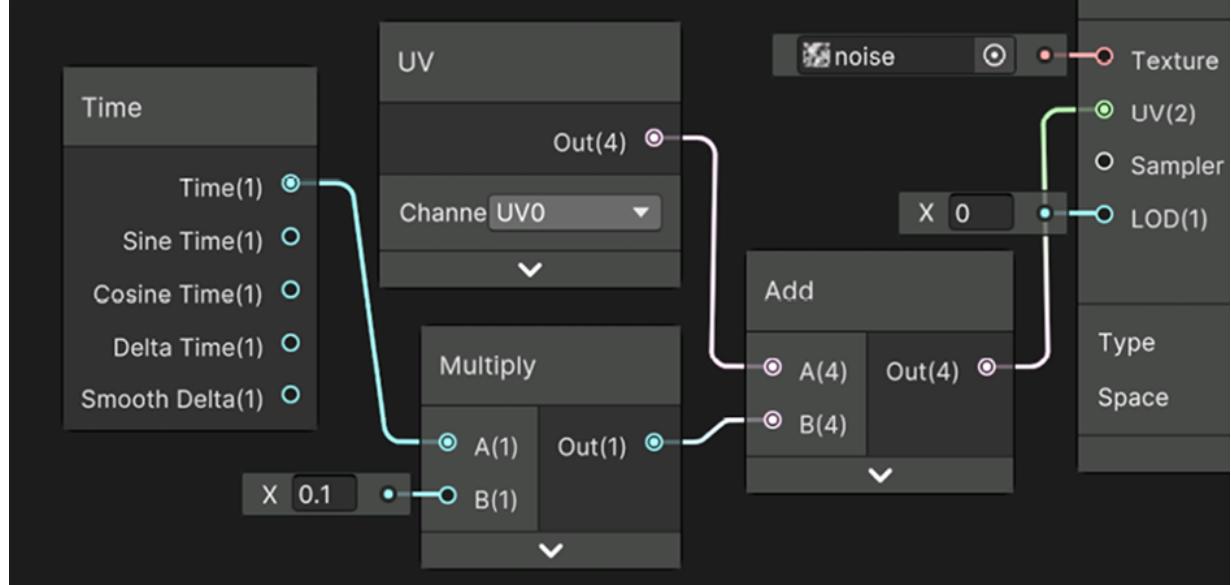


Figure 10.60: Animated ripples vertex effect graph

As you can see, we are again taking the original UV and adding the time multiplied by any factor so it will slowly move, the same as we did previously with our water texture. You can keep playing around with this, changing how this looks with different textures, multiplying the offset to increase or reduce the height of the ripples, applying interesting math functions like sine, and so much more, but for now, let's finish with this.

Summary

In this chapter, we discussed how a shader works in the GPU and how to create our first simple shader to achieve a nice water effect. Working with shaders is a complex and interesting job, and in a team, there is usually one or more people in charge of creating all of these effects, in a position called Technical Artist; so, as you can see, this topic can expand up to a whole career. Remember, the intention of this book is to give you a small taste of all the possible roles you can take in the industry, so if you really liked this role, I suggest you start reading shader-exclusive books. You have a long but super interesting road in front of you. Enough shaders for now! In the next chapter, we will look at how to improve our graphics and create visual effects with particle systems!