

Unity for Human Beings

By

Jesse Glover

Professional Unity Developer

Pablo Farias Navarro

Unity Developer & Founder of Zenva

Renan Oliveira

Game Development Expert

Tim Bonzon

Advanced Unity Developer and Animator

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Before diving into this eBook, why not check out some resources that will supercharge your coding skills:

ACCESS ALL 250+ COMPLETE COURSES



Unlimited access to EVERY course on our platform! Get new courses each month, help from expert mentors, and guided learning paths on popular topics.

GET EVERY COURSE

LEARN PYTHON BY BUILDING A GAME



No experience is required to take this project-based course, which covers variables, functions, conditionals, loops, and object-oriented programming.

LEARN PYTHON

FREE CODING 101 BUNDLE



Courses that will quickly get you coding with the world's most popular languages! Discover Python, web development, game development, VR, AR, & more.

LEARN FOR FREE

BUILD YOUR OWN GAMES WITH UNITY



Learn how to build games with C# and Unity! You'll master popular genres including RPGs, idle games, Platformers, and FPS games.

BUILD GAMES

Table of Contents

How to Get Started with Unity3D – Ultimate Beginners Guide

Frequently Asked Questions:

UI BASICS:

Basic Unity Project

 Part 1: Building the UI

 Part 2: Scripting

How to Build a Complete 2D Platformer in Unity

Introduction

Project Details

Tilemap Editor, Timeline, and Cinemachine

Setting up our project

Creating our environment

Adding the character

Creating the enemy

Scripting our enemy

Cinemachine and Timeline

Outro

Understanding 2D Animations in Unity3D

Requirements

Segment 1: Animation Basics

 Section 1: Setting up the Project

 Section 2: Let's Make an Animation

 Section 3: Controlling animations via Scripting

Segment 2: Player controlled Sprite with Animations

Fundamentals of 3D Development with Unity3D

Section 1: Import a 3D Model into Unity3D

Section 2: 3D Theory

 Why is knowing how the rotation works on the three axis' important?

 How do I use Quaternions in Unity3D?

 Camera in 3D Mode:

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Various 3D Objects built into Unity3D:

Unity 3D First and Third Person View Tutorial

Section 1: First Person View

Section 2: Third Person Perspective

Section 3: Collisions

Box Colliders

Mesh Colliders

Rigidbody

Sphere Collider

Terrain

Basic Collisions:

A Deeper look into the Camera in Unity3D

WHAT WE WILL DISCUSS TODAY:

TUTORIAL SCREENCAST

LET'S GET STARTED

SECTION 1: CAMERA PROPERTIES

SECTION 2: SPLIT SCREEN CAMERA

BUILDING THE EXAMPLES:

CAMERA INSIDE ANOTHER EXAMPLE:

HORIZONTAL SPLIT SCREEN EXAMPLE:

VERTICAL SPIT SCREEN:

SECTION 3: CAMERA OVERLAYS

PAUSE MENU:

How to Script a 2D Tile Map in Unity3D

Prologue: The Back Story

TUTORIAL SOURCE CODE

SECTION 1: THE BORING THEORY SECTION

SECTION 2: PLANNING THE CODE

SECTION 2.5: WRITING THE CODE

SECTION 3: BUILDING THE SCENE IN UNITY3D

How to Build 3D Algorithms with Unity3D

SECTION 1: BUILDING THE UI

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

SECTION 2: THE FIRST ALGORITHM

SECTION 3: ALGORITHM WITH A LIST OF PREFABS

THE WRONG WAY:

THE CORRECT WAY:

SECTION 4: MAKING A 3D CUBE

SECTION 5: RANDOMIZING THE CUBES VARIANT

SECTION 6: TURNING THE CUBE INTO A HALLWAY AND STAIRS

How to make a game in Unity3D – Part 1

PART 1: INITIAL DESIGN PHASE

SECTION 1: BASE GAME IDEA

SECTION 2: EXPANDING UPON THE IDEA

SECTION 3: FLESHING OUT MORE DETAILS

SECTION 4: STARTING THE GAME DESIGN DOCUMENT

SECTION 5: WRITING THE DOCUMENT

PART 2: PROTOTYPING

SECTION 1: CRUDE ART

SECTION 2: CRUDE MUSIC

PART 3: PSEUDOCODE TIME

SECTION 1: PLAN OUT WHAT WE WANT TO DO

SECTION 2: PLANNING OUT BUTTON EVENTS

CONCLUSION:

How to make a game in Unity3D – Part 2

PART 1: SETTING UP THE GAME SKELETON

PART 2: CHOOSING WHERE TO BEGIN

SECTION 1: GAME BOUNDARIES

SECTION 2: PLAYER CONTROLLER

SECTION 3: MISSILE CONTROLLER

SECTION 4: DESTROYING OBJECTS

SECTION 5: ADDING LIFE TO THE BACKGROUND

PART 3: USING UNITY TO BUILD OUT THE GAME

How to make a game in Unity3D – Part 3

Issue Fixing

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Backend Code

READING AND WRITING XML DATA

Timing and timer

Modification for Collision events and writing XML Data

Scene Transitions

Game Start Scene Building

Game Over Scene Building

Finalizing the Game Play Scene

How to Create an RPG Game in Unity - Comprehensive Guide

Source code files

Assets copyright

Title Scene

Background canvas

HUD canvas

Player party

Town Scene

Creating our Tilemap

Player prefab

Starting Battle

Battle Scene

Background and HUD Canvases

Units Animations

Turn-Based Battle System

Attacking Units

Selecting Unit and Action

Finishing the Battle

How to Create a Multiplayer Game in Unity

Creating Project and Importing Assets

Source Code Files

Background canvas

Network Manager

Ship Movement

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Spawn Positions

Shooting Bullets

Spawning Enemies

Taking Damage

Conclusion

How to Import Blender Models into Unity – Your One-Stop Guide

1. Coordinate differences
2. Blender model settings
3. Importing .blend files vs importing .fbx files
 - 3.1. Importing a blend file into Unity
 - 3.2. Importing a fbx file into Unity
4. blend vs fbx? Which one wins?

Intro to Shaders with Unity3D

Intro to Shaders

A deeper look into shaders

Standard Shader

Mobile Bumped Diffuse Shader

Mobile bumped with specular mapping.

Sprites Default shader

Sprites Diffuse shader.

Building our own Shader

Breaking down Shader Syntax:

How to Get Started with Unity3D – Ultimate Beginners Guide

By Jesse Glover

Unity3D is a game development engine originally designed with 3D game making in mind, however, it is also very possible and easy to create 2D games and standard applications using this engine. In this introductory tutorial on Unity3D, we will discuss how to use the UI and attach scripts to components. By the end of this tutorial series, we will have made an application that has dynamically generated buttons that will show a specific image upon being clicked.

I know, it doesn't sound very exciting. But this is an introductory course for using Unity3D and I felt that making a game right off the bat would be extremely confusing, especially for someone that has never used Unity3D before. If you have any experience working with Blend, WPF or Windows Store Apps, this will be similar to what you know.

Frequently Asked Questions:

What do I expect from my readers?

I expect them to want to learn Unity3D and have at the very least a fundamental understanding of C#.

What languages are supported by Unity3D?

Unity3D supports C#, JavaScript (commonly called UnityScript), and Boo.

Does Unity3D support Object-Oriented Programming?

To some degree Unity3D does support Object-Oriented Programming Concepts, however, Unity3D is composed of Component Oriented Programming.

Where do I get Unity3D from?

<https://store.unity.com/#plans-individual>

Now that the FAQ is out of the way. This tutorial assumes the reader has zero knowledge of how Unity3D works, so if you know about Unity3D, please wait for the Intermediate and Advanced tutorials coming soon.

Download the tutorial asset files [here](#).

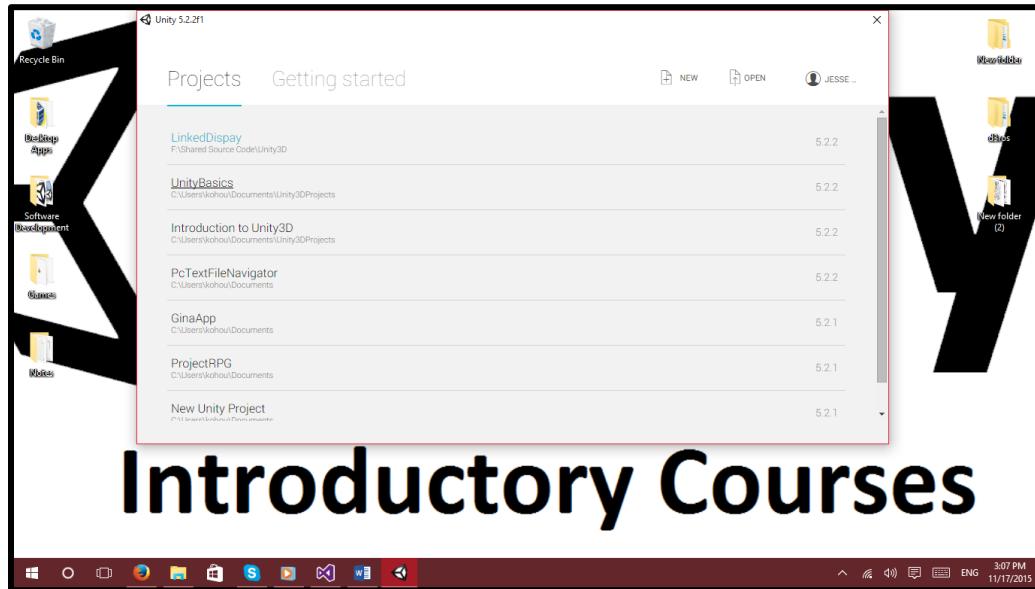
This tutorial is broken down into several sections. First Section is UI basics, the next section is Scripting Basics, finally the last section is Putting it all together.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

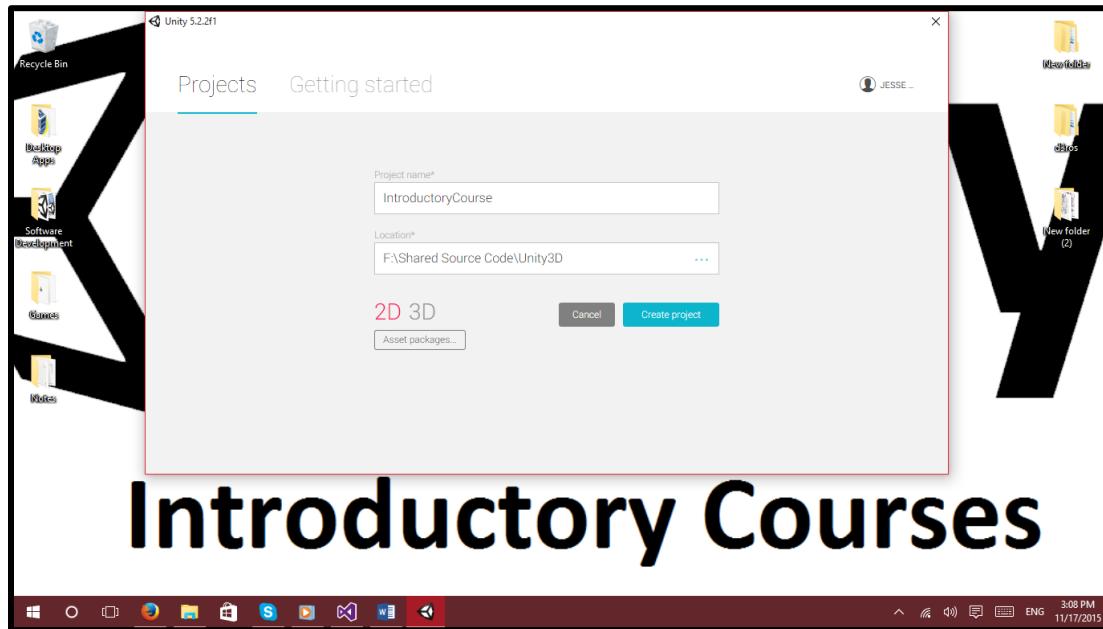
© Zenva Pty Ltd 2021. All rights reserved

UI BASICS:

When you open Unity3D, it will look like the image below. The only difference is that there should be absolutely no projects listed. Select NEW to get started.



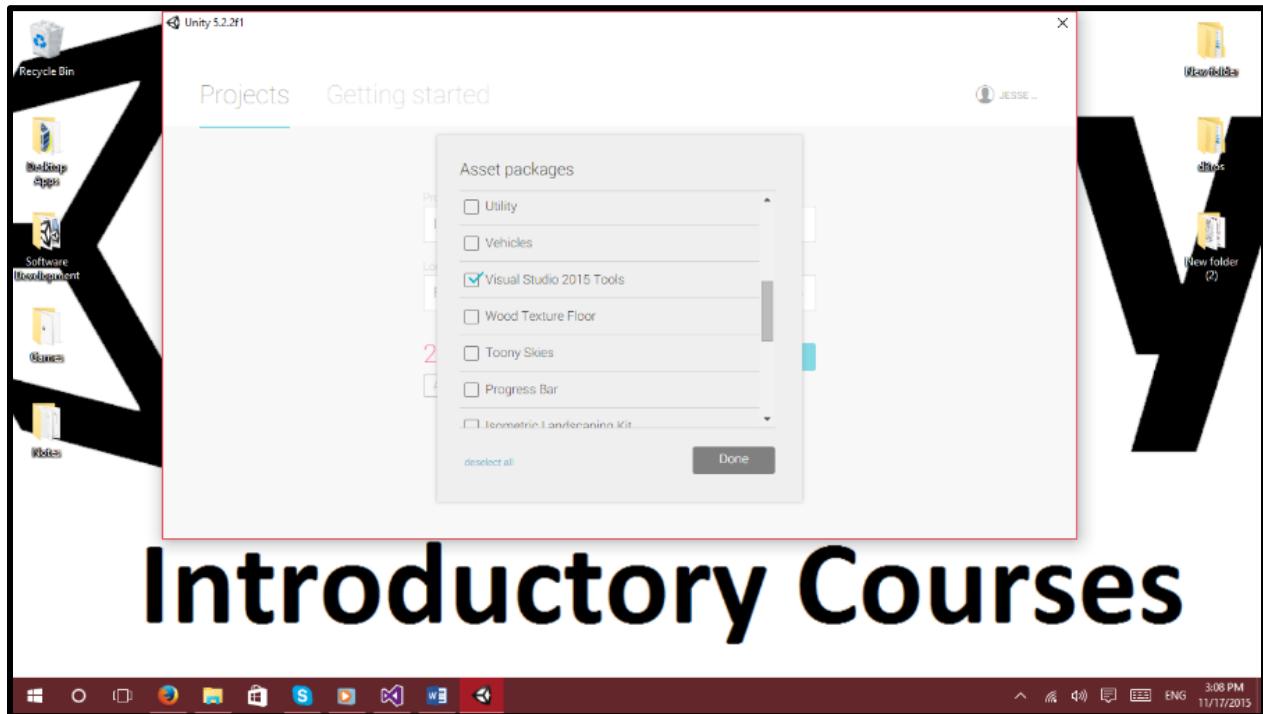
The window will look like this. I have named the project Introductory courses and set the save location to be in my F: drive. You can set it to be anywhere you want. Make sure the project is set to 2D and then select Asset Packages.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

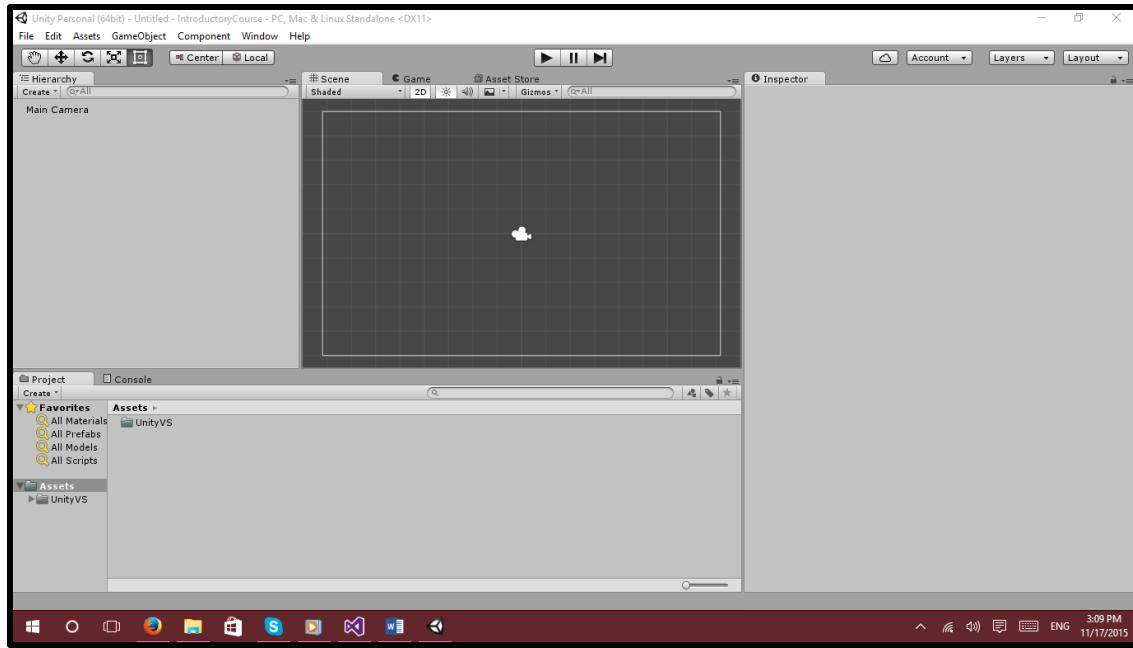
After selecting Asset Packages, the only package we want to import is Visual Studio Tools 2015/ 2013 (Whichever version of Visual Studio you are running). Alternatively, you can skip this step if you would prefer to use MonoDevelop prepackaged with Unity3D.



Click done once you have selected Visual Studio 2015/2013 tools. Now select Create project. This will close the window and Unity will build the base project for you.

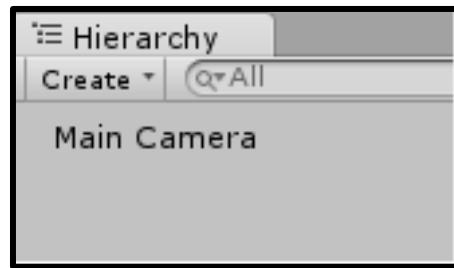
You will see Unity's editor once the packages finish building. So, let's take a moment and get an explanation what we are looking at in further detail.

I should also point out that in any project type you make, it will always have a Camera on the scene by default. If you have any packages added to the current project, they will be in the assets folder as well.

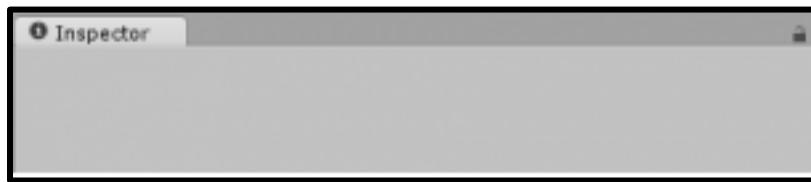


We will only be explaining what I feel are the most important features of the UI that we are looking at. A book could be written about Unity3D's UI Editor alone.

Hierarchy tab is where you will be placing components that will appear in the Scene.



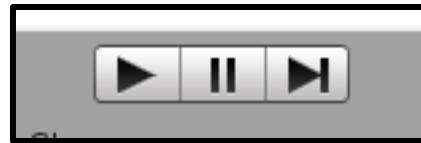
The Inspector tab allows you to manipulate objects on the scene. (We will see this in action soon)



These are the Play, Pause, and Step buttons. They allow you to test your application or game.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

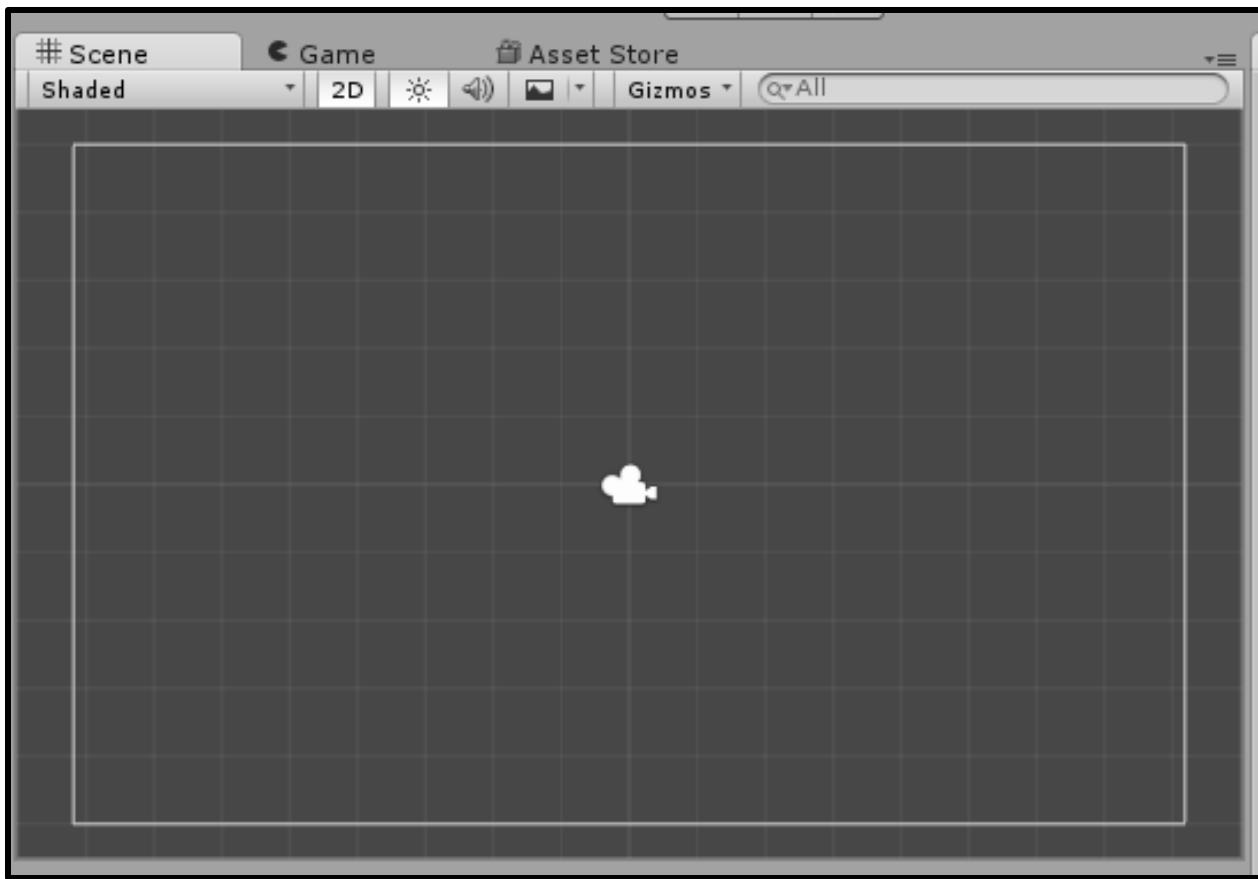
© Zenva Pty Ltd 2021. All rights reserved



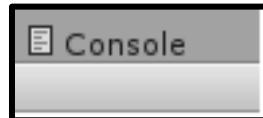
The project tab allows you to see the folders, scripts, images, and other assets you have added to your project.



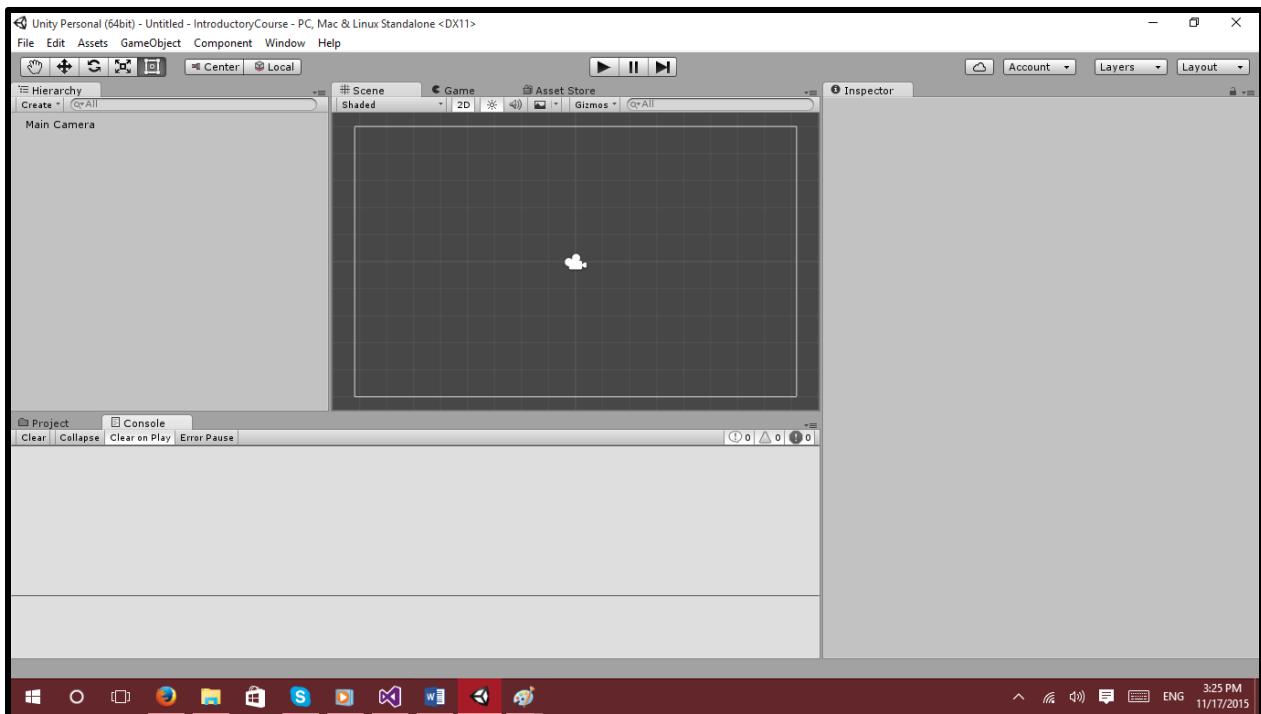
The Scene tab is where you can see the objects you have placed. Game view allows you to see it as you would in your game. Asset store tab allows you to view assets that you could download and install into your project.



There is also a Console Tab. Which is located directly next to your project tab in the default setup.



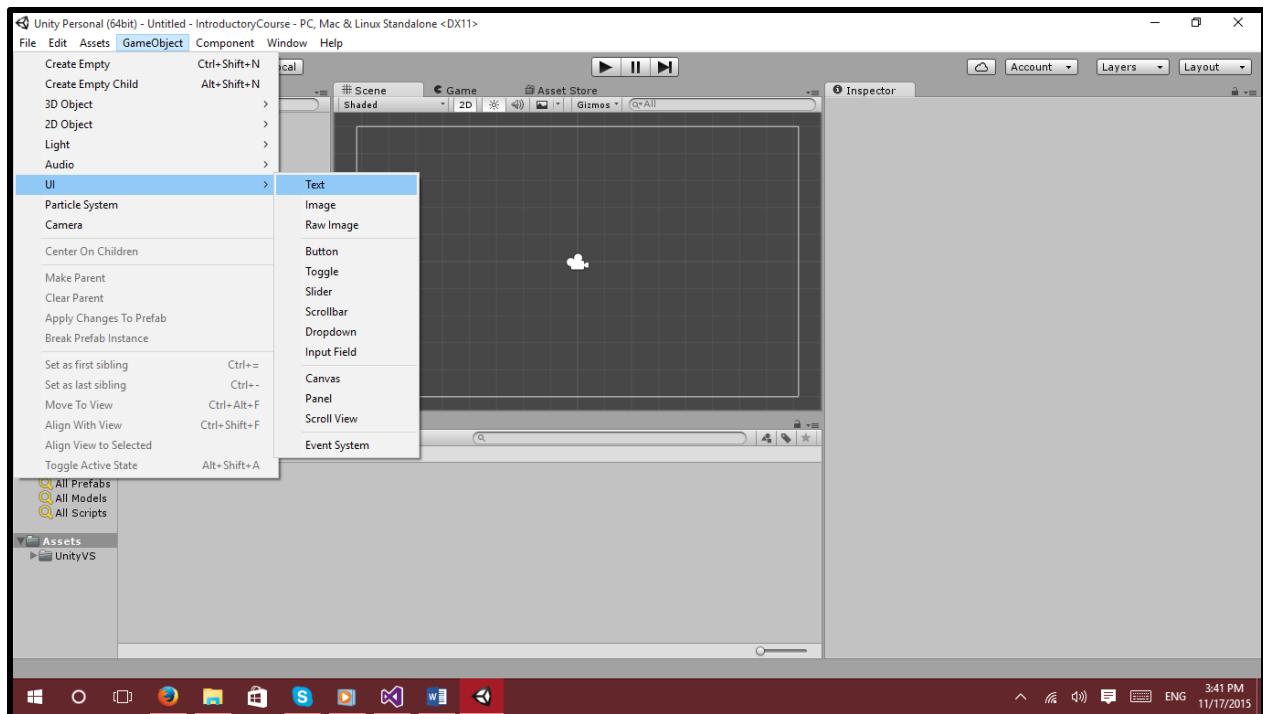
Let's look at the Editor after we have selected the Console tab.



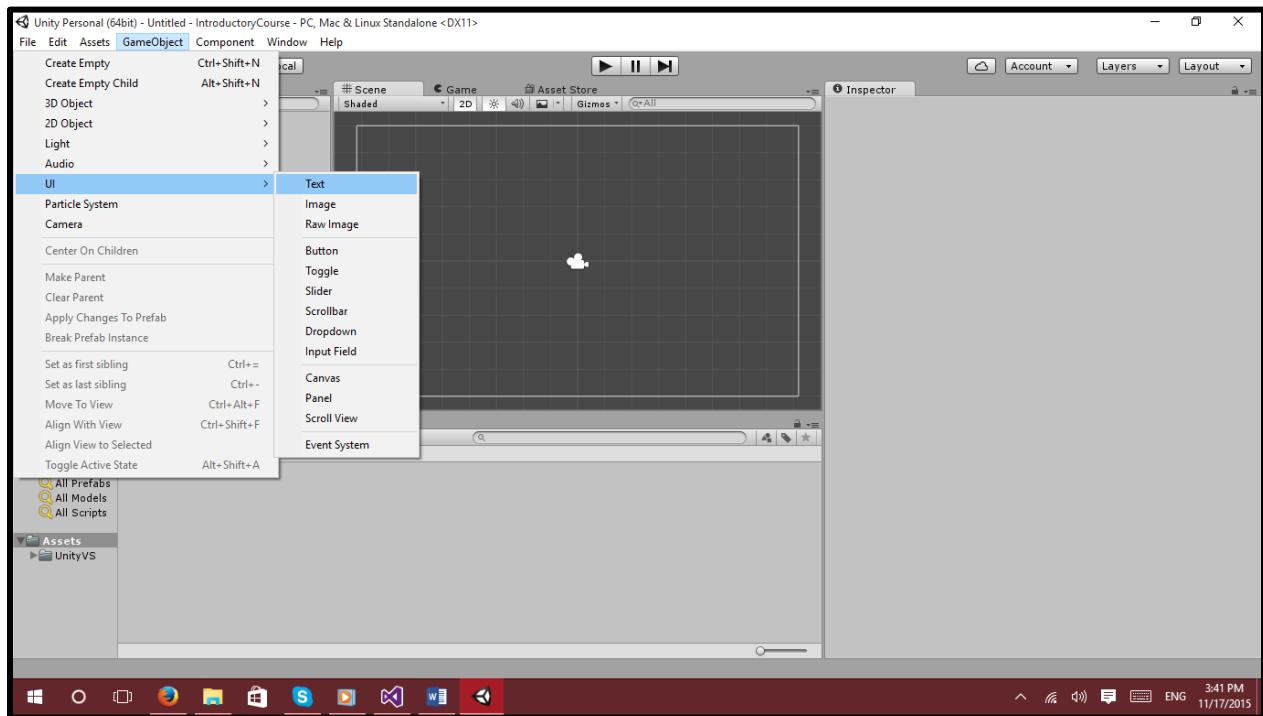
As you can see, it has removed the view from the project tab. This comes very much in handy when you are testing/ debugging your app during play. It will show you any compile errors and warnings.

We will see all of these items in action as we build the application during the last section. Now that you are familiar with the editor. I think it is time we throw some components into the Hierarchy pane and show some deeper understanding of how everything plays out together.

Let's add a text component to the Unity Editor. Select GameObject, UI, Text.

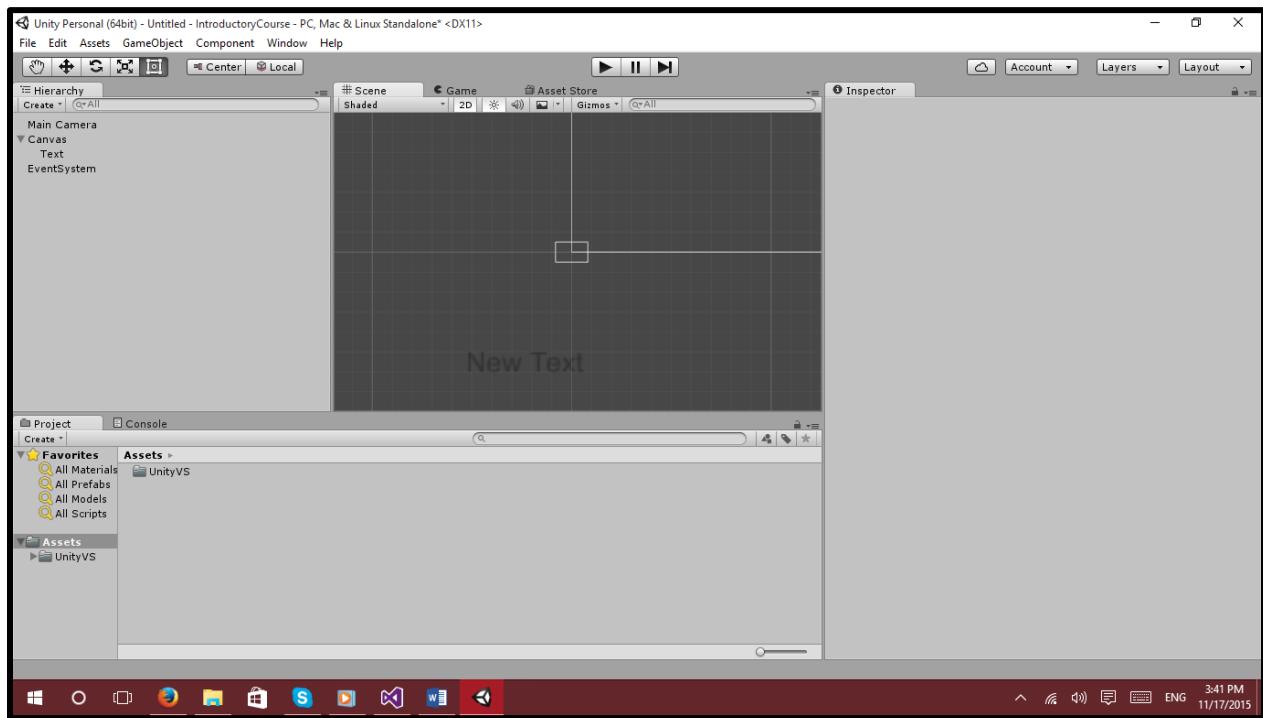


Here are the results. Take a moment and look at the Hierarchy pane. You will notice a few more items.



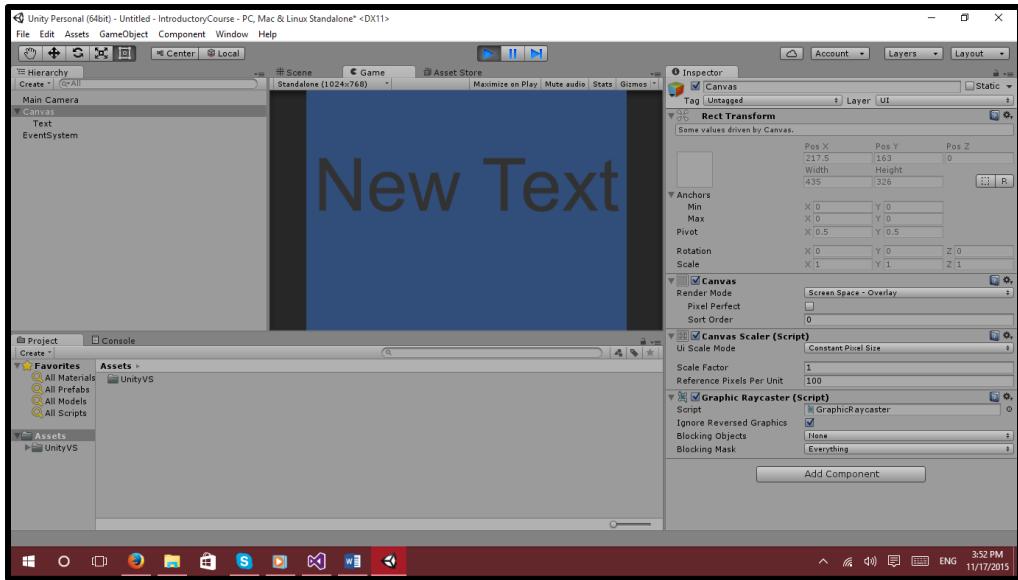
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Most UI controls are directly tied to the Canvas control. So Unity has a smart enough editor to go ahead and add the Canvas component and EventSystem for you. You don't have to do anything with them, as a matter of fact, we could go ahead and run the project right now and everything will display just fine. (Side note, just make sure the text is within the Canvas)



I went ahead and changed a few items within the Inspector pane to make things easier to see. So I feel obliged to explain what I did. Notice that in the Hierarchy pane, I have text selected, that allows me to see the different parameters that I can modify in the Inspector pane. I put a check mark in the Best Fit box and changed the Max Size to be 100.

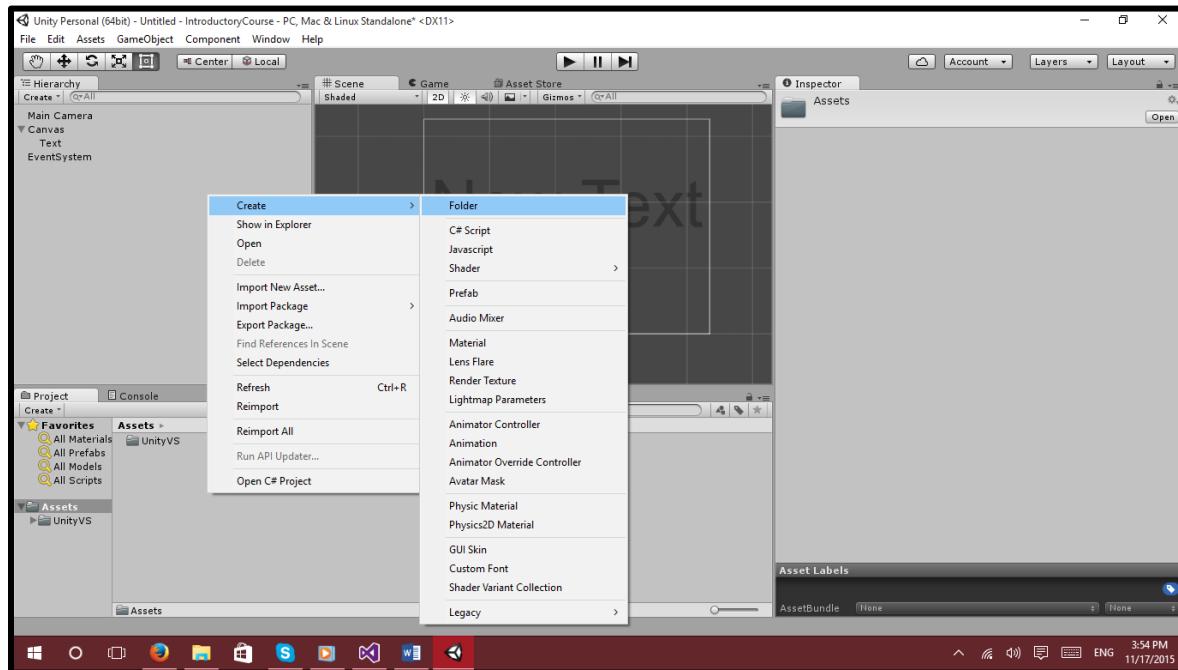
Now, I'll run the project.



It is also extremely important to note that you can still make changes to the editor while it is in play mode, however, those changes are not saved. So be extremely careful about that.

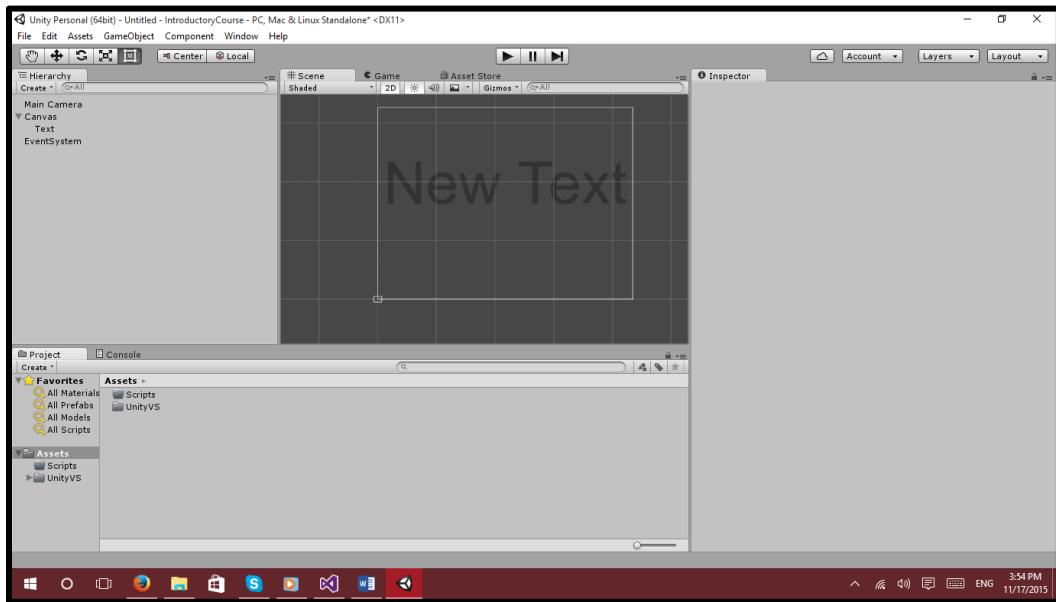
Alright, I think that is about all that we need to talk about for the Basics for the UI. Let us move on to Scripting Basics.

I right clicked on the Assets folder, highlighted Create, and selected Folder.



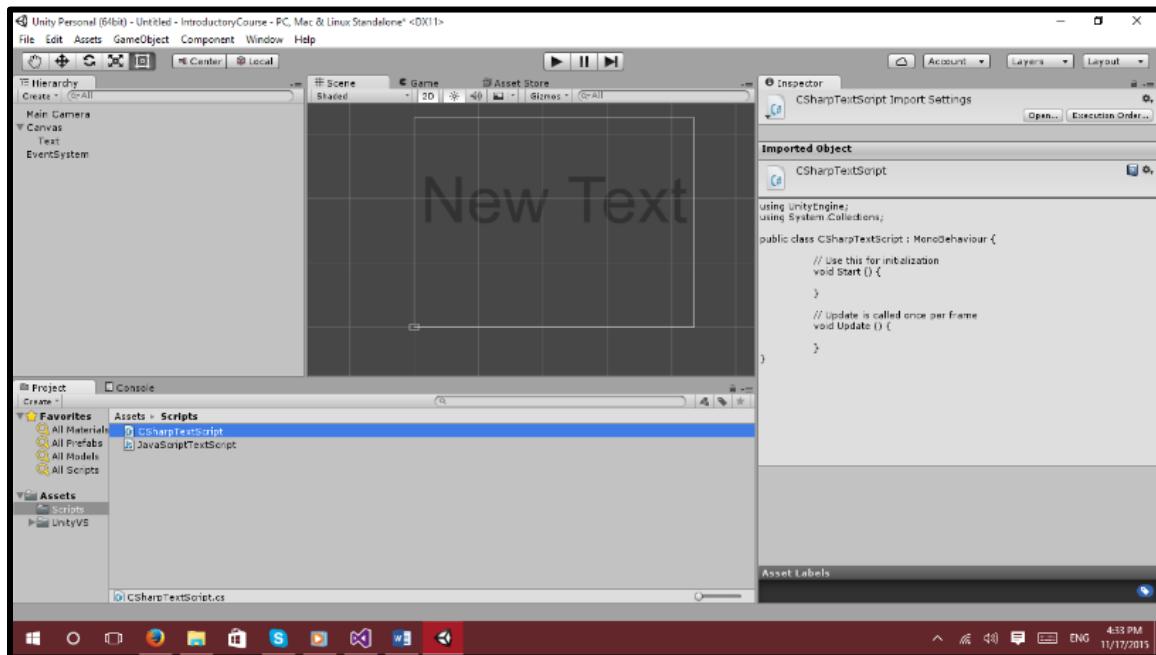
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

This allowed me to create a new folder, which I named Scripts.



Then after making sure I was inside of the Scripts folder, I followed the same pattern. Right clicked inside the Scripts folder, highlighted Create, and selected C# Script.

You can select a script and it will show you what code has been written in the Inspector Pane. C# Code:

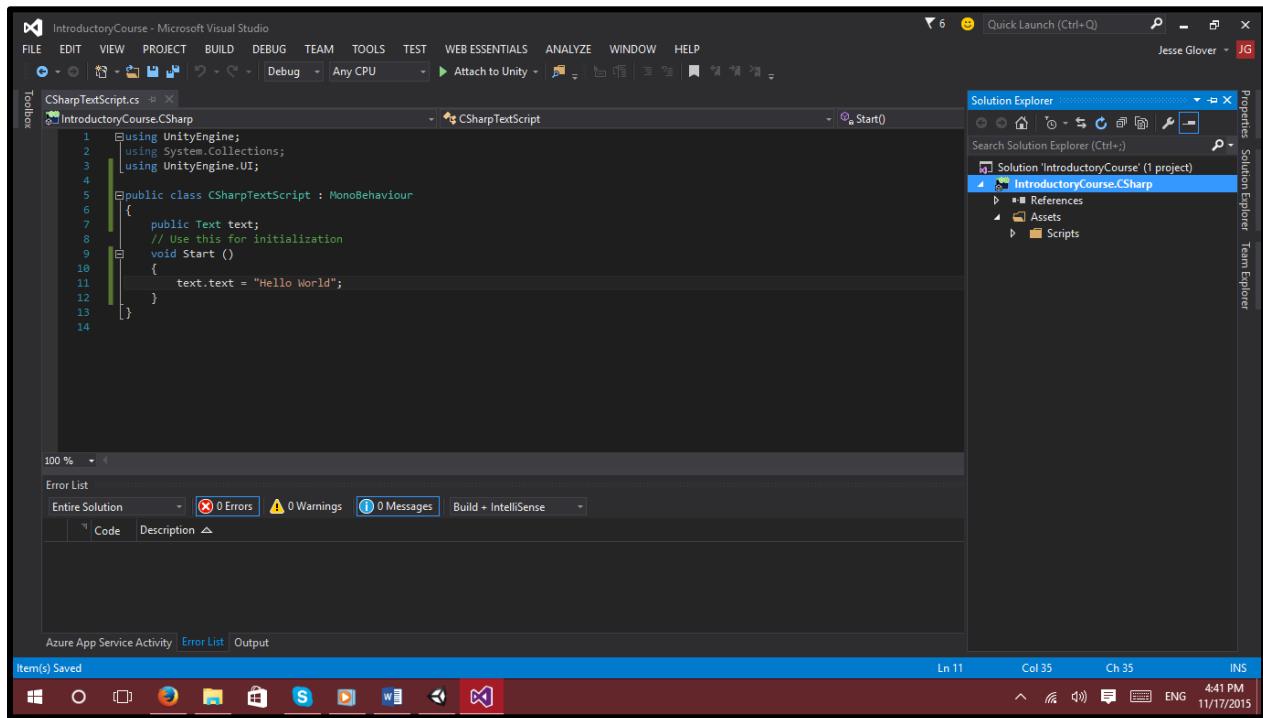


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

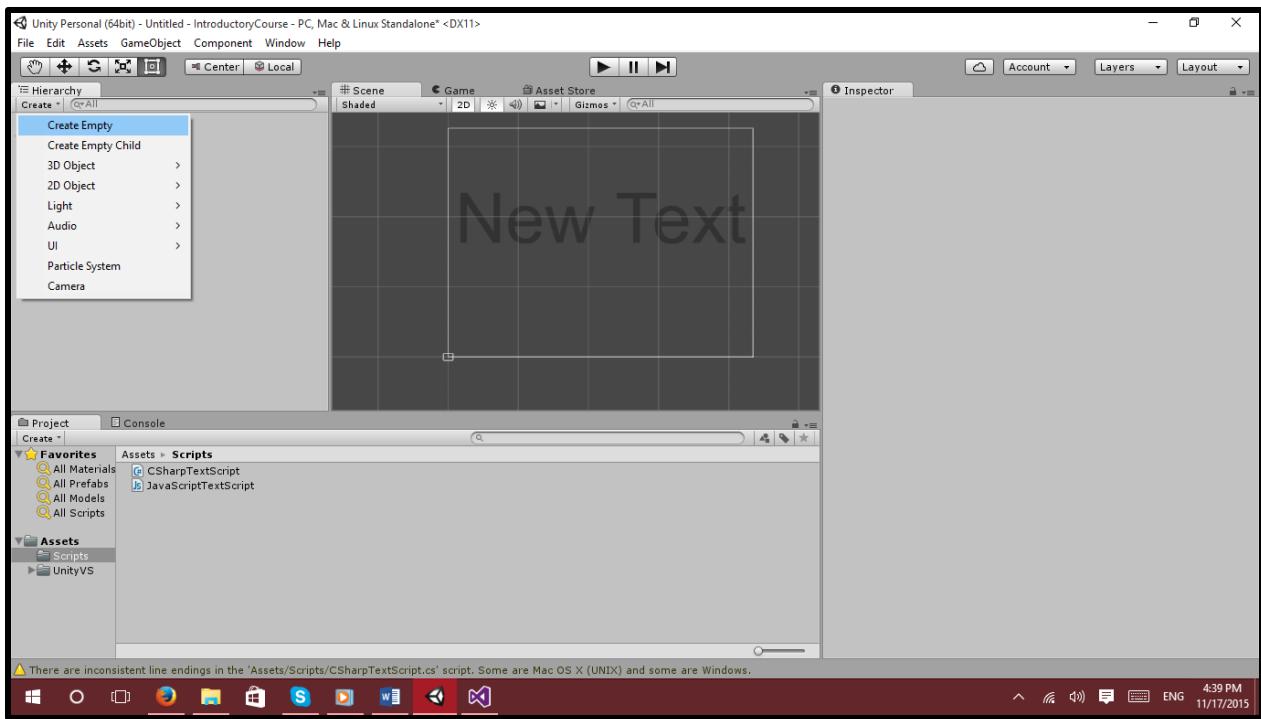
© Zenva Pty Ltd 2021. All rights reserved

Let's add a script to the existing project.

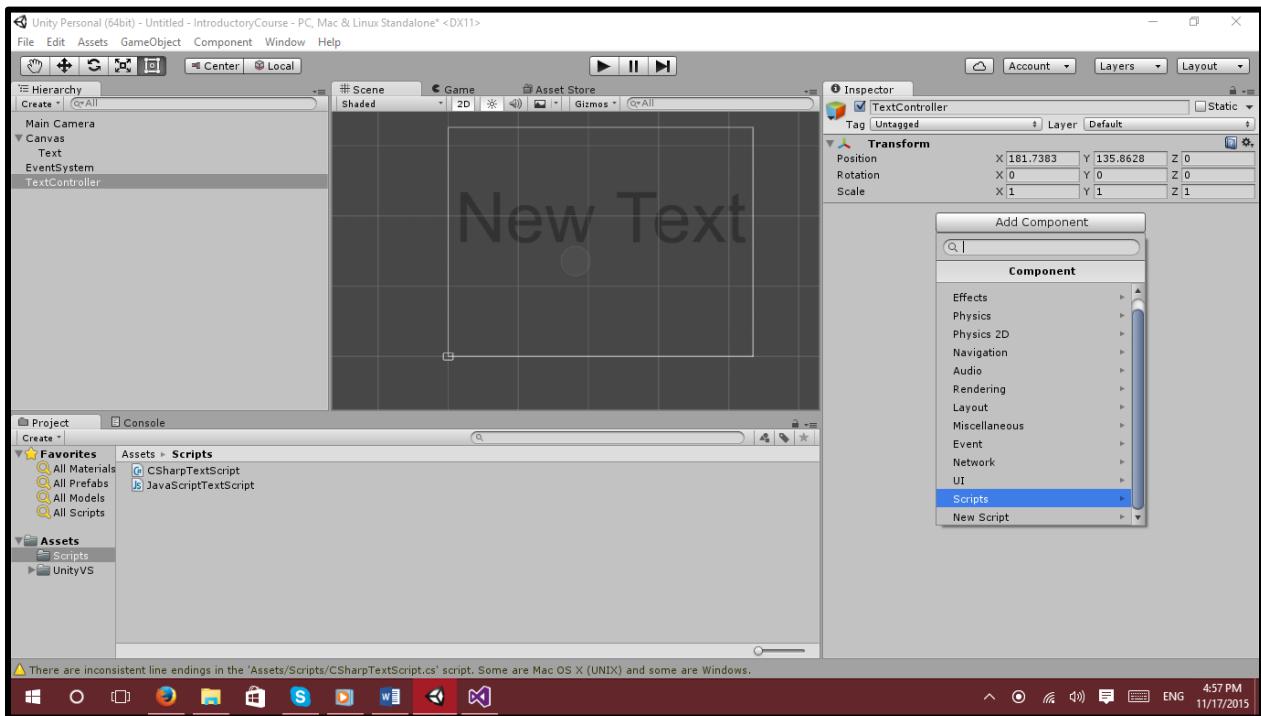
The script will be extremely simple. Change the Text from New Text to Hello World. Since Text is a component within Unity, we have to be able to access it. To do this, we make a public variable that calls it. We also need to import the UnityEngine.UI Namespace. We will put it inside of the Start Method and remove the update method. The reason we want the public variable is so we can see it in the editor and attach components if we need to.



Now, let's add the script to a component. Start off by Creating an empty GameObject in the hierarchy pane. To do this, select Create just under hierarchy and select the Create Empty in the list.

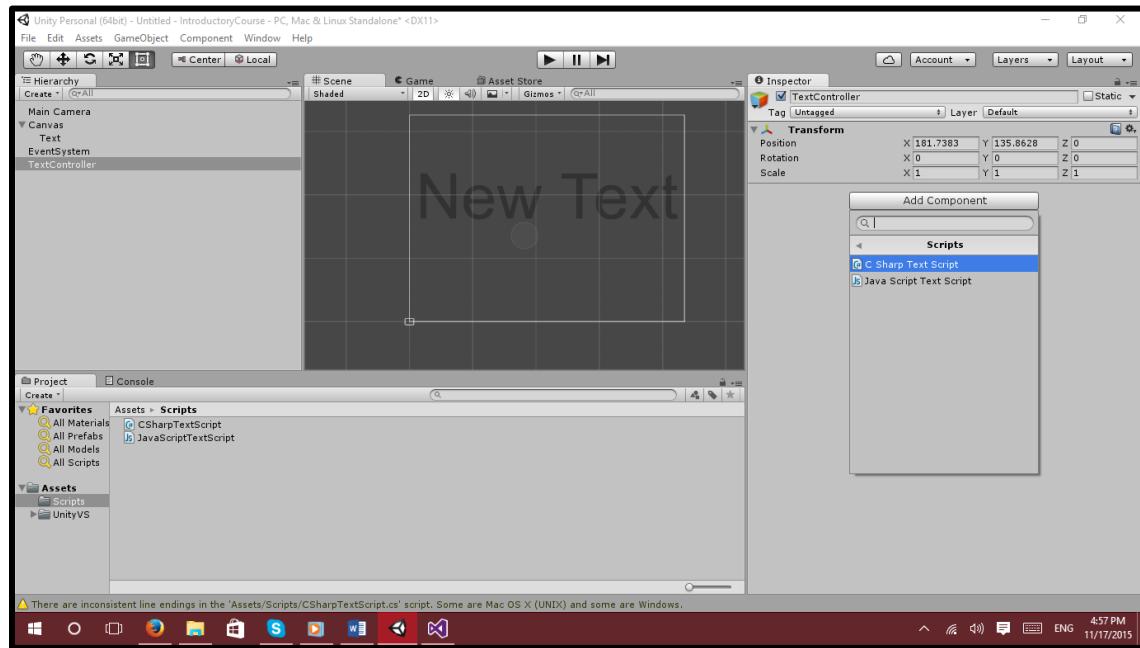


Rename that object by changing it from empty to TextController inside of the Inspector Pane.
Next Select Add Component and select the Scripts item in the list.

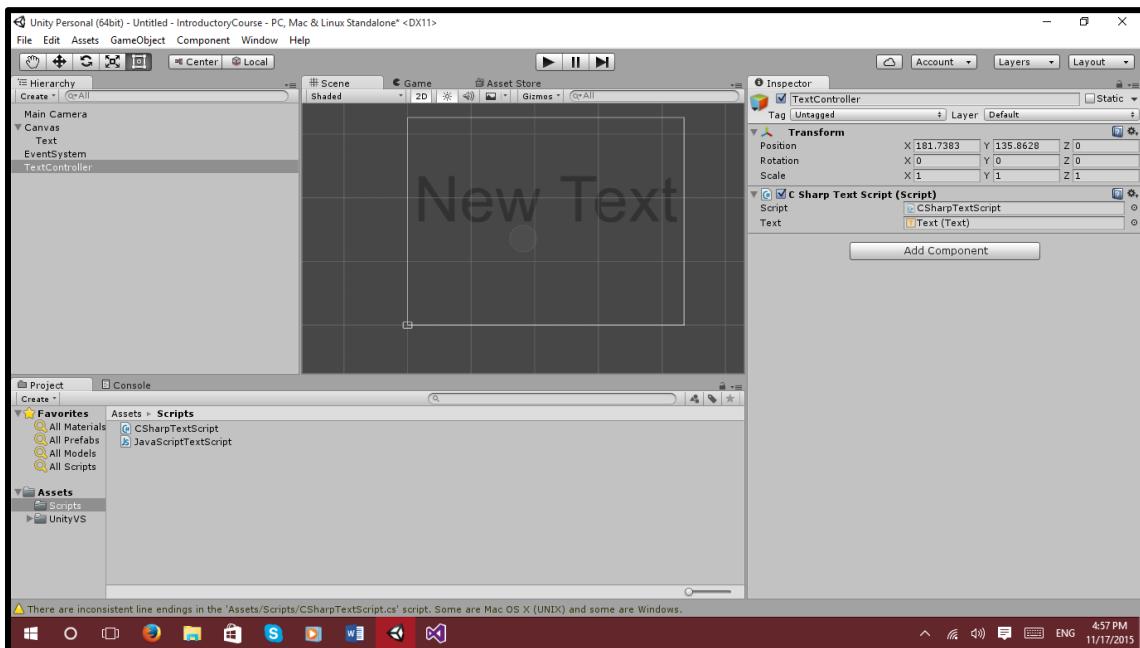


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

After selecting the Scripts. Choose which script you want to use. In this case it will be the C# one.



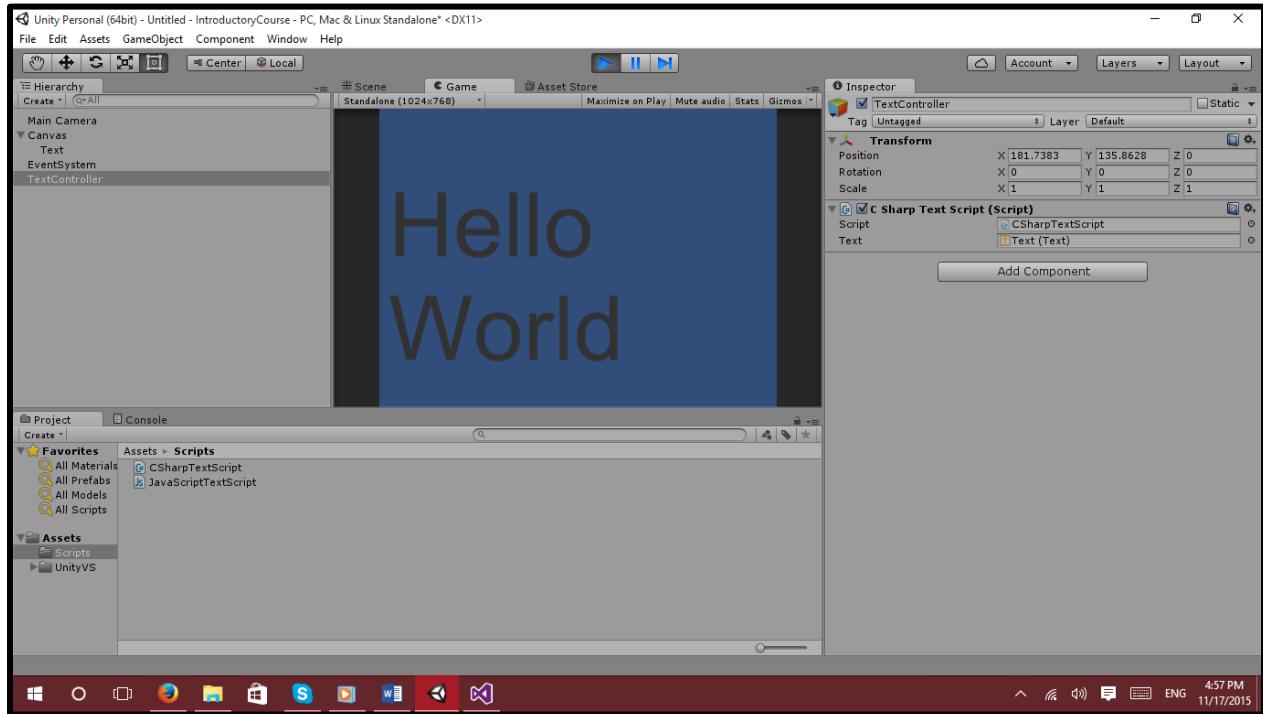
Remember the text we created inside of the canvas? Drag that Text component from the hierarchy pane to the Text field inside of the inspector to attached the object to the script.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Now, let's run the project inside of the Unity Editor to see what happens.



It overwrote the original New Text and displayed Hello World just as intended.

Now that you have the basics for Unity3D's editor and scripting. Let's move along to Section 3 and make a little application with what we have learned as well as expand on it.

Basic Unity Project

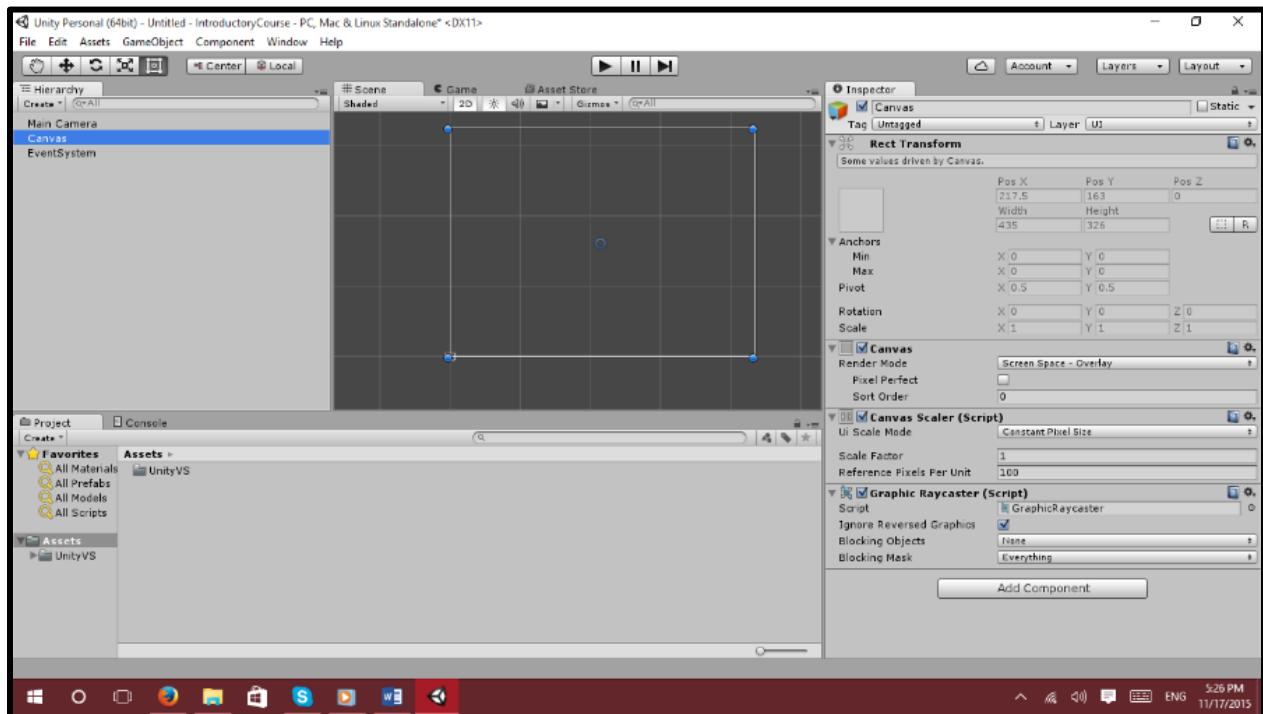
Welcome to Section 3. I'm sorry if you were bored with the previous sections. I hope to make this one much more entertaining. This is the final section where we will take everything we have learned and mash it all together and create an application with it. The scripting is a little bit more involved than with section 2's scripts. So, not only will my code be commented, but I will discuss some parts of it in more detail.

Art assets were downloaded from <http://www.gameart2d.com/cat-and-dog-free-sprites.html> under Creative Common Zero (CC0) a.k.a Public Domain license.

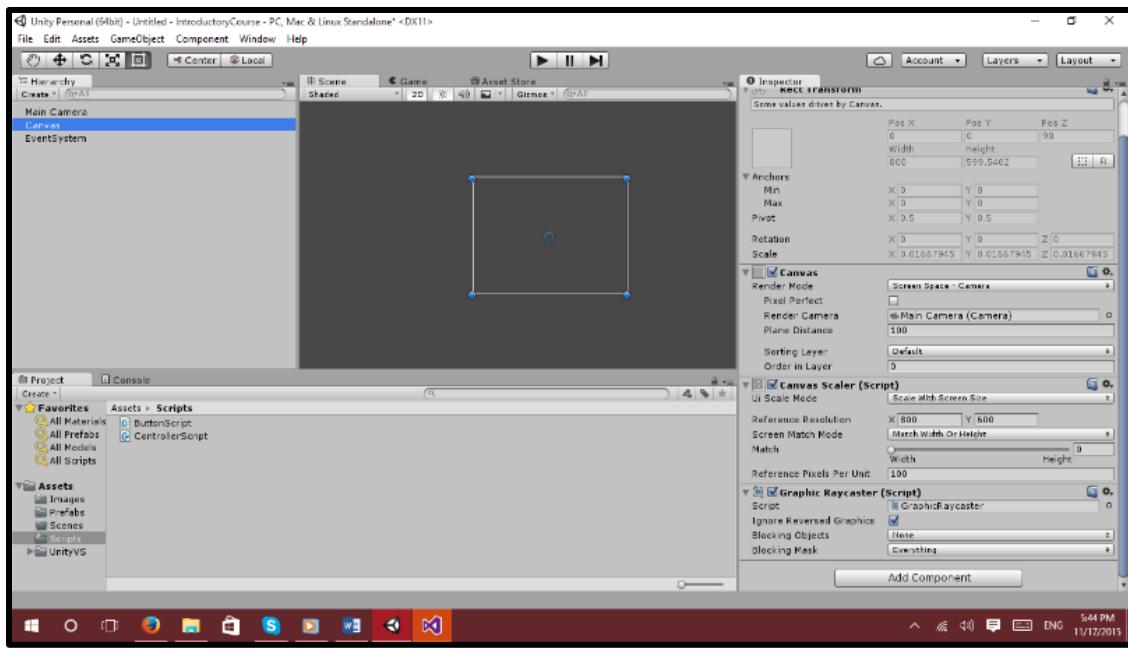
Part 1: Building the UI

You should always start off by making your folders and naming them accordingly. (Scripts, Scenes, Prefabs, Images). Go ahead and add 2 scripts into the script folder. Name one script “ButtonScript”, and the other “ControllerScript”.

We again add a Canvas component to the scene. Only, this time. We will make some changes to the Canvas component.

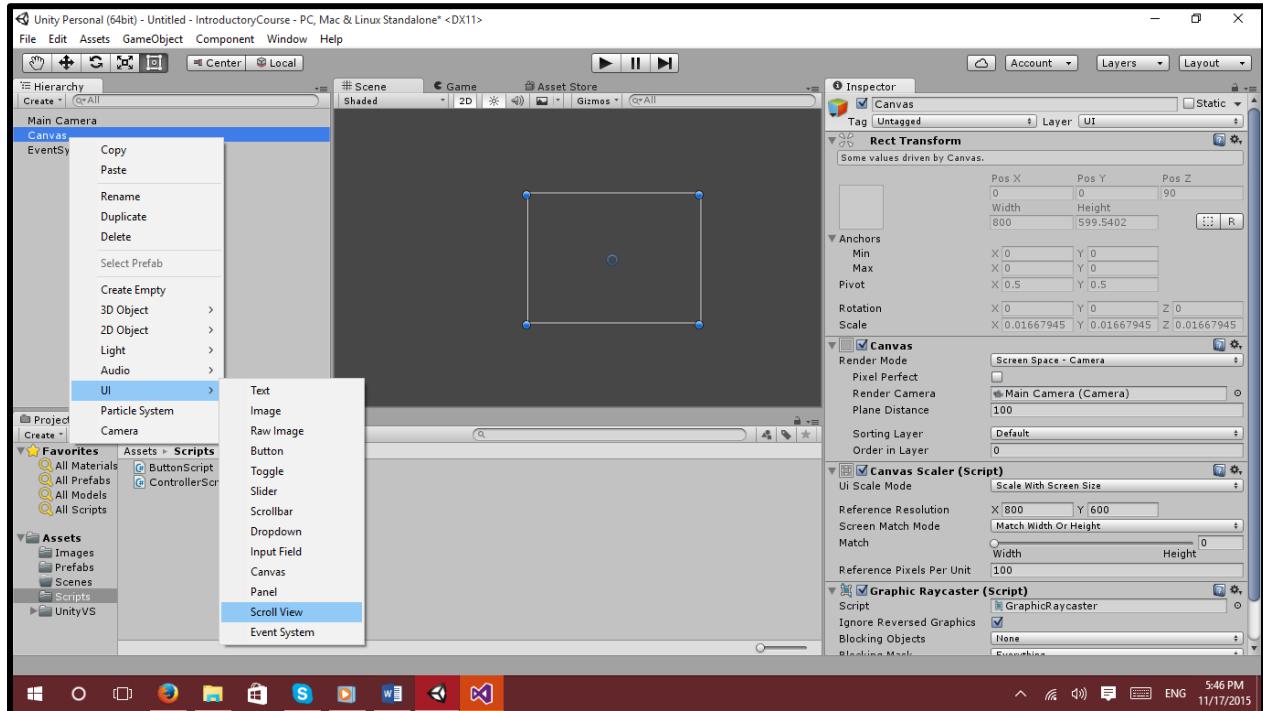


We will make the Canvas component display at the same resolution as the Camera and set screen size parameters.



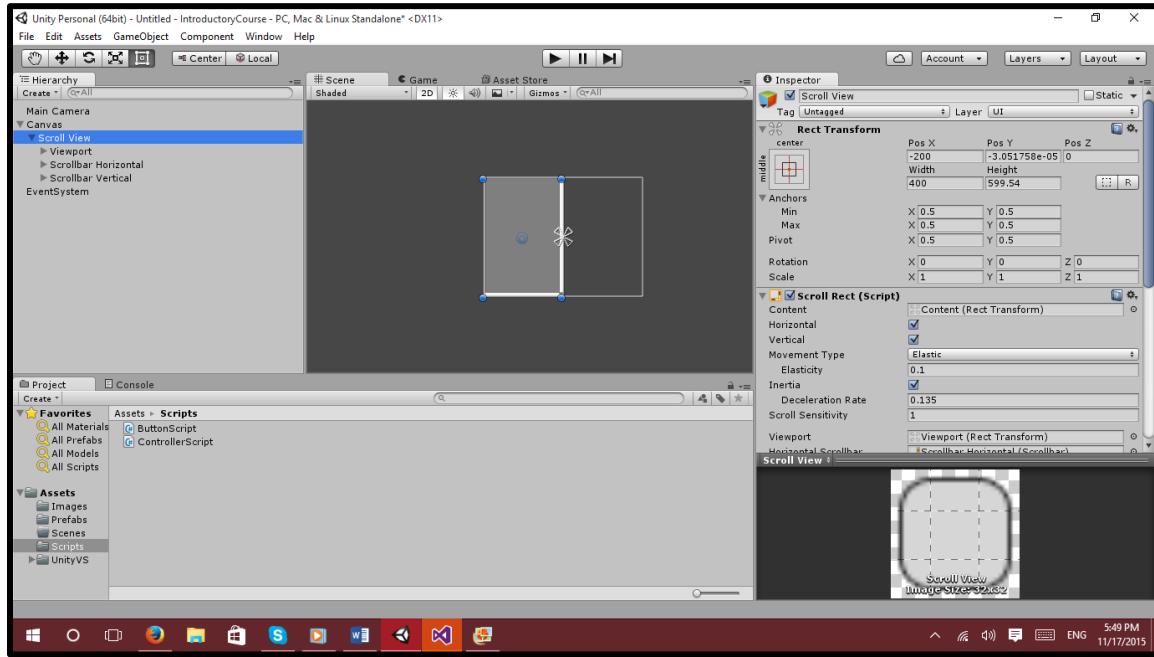
We want the screen resolution to be 800X600 and the screen to made width or height with the slider pulled towards width.

Next up, we want to add a Scroll View element (Which is basically a ListView in Blend/ WPF).

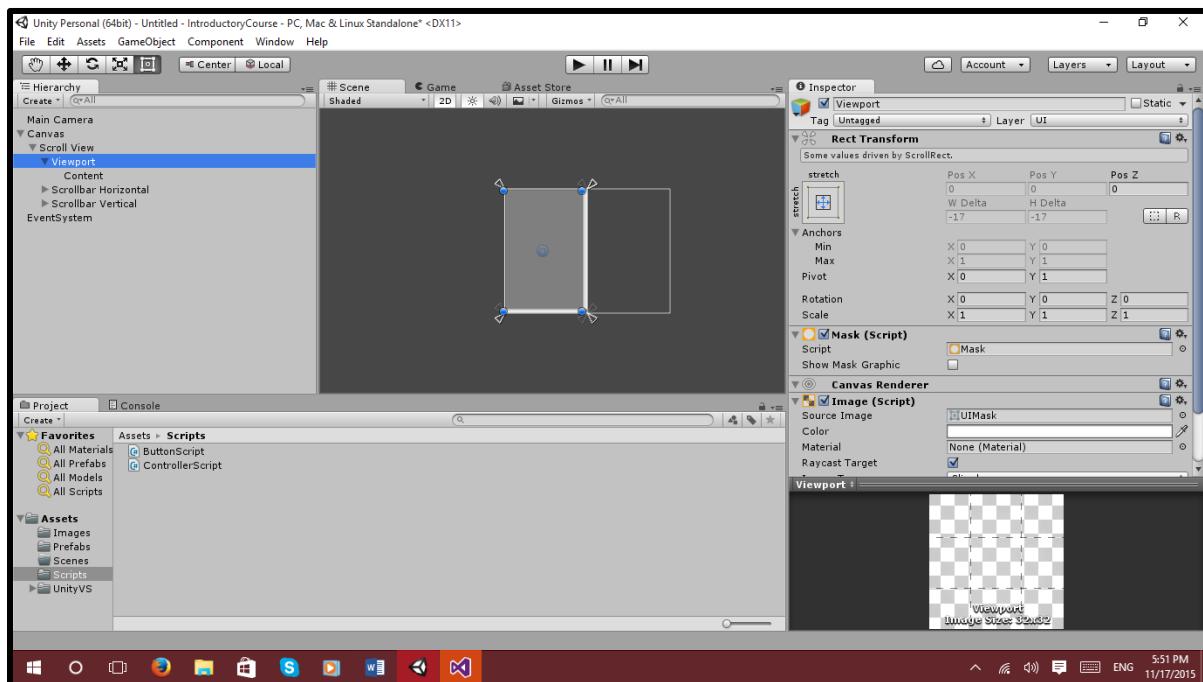


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Let's resize it to take up about half of the screen. Notice that the ScrollView component has a few child elements to it. We only need to worry about the ViewPort Element. Let's open that one up more to see what it has for us to manipulate.



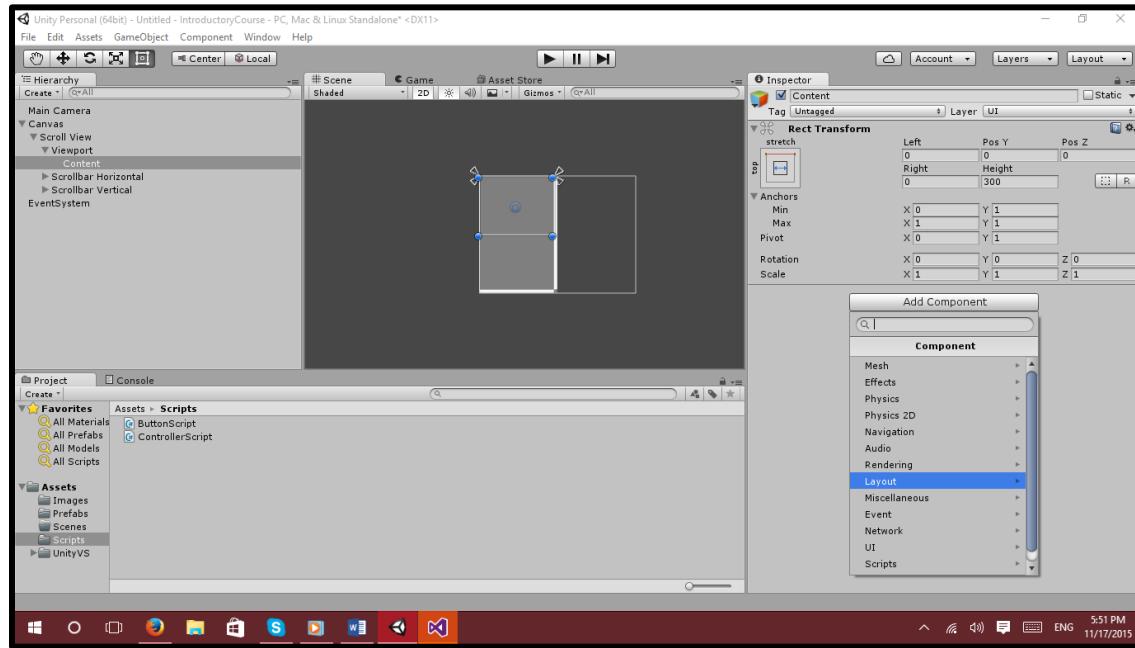
It has a content element. We will make good use out of it by adding a layout element to it.



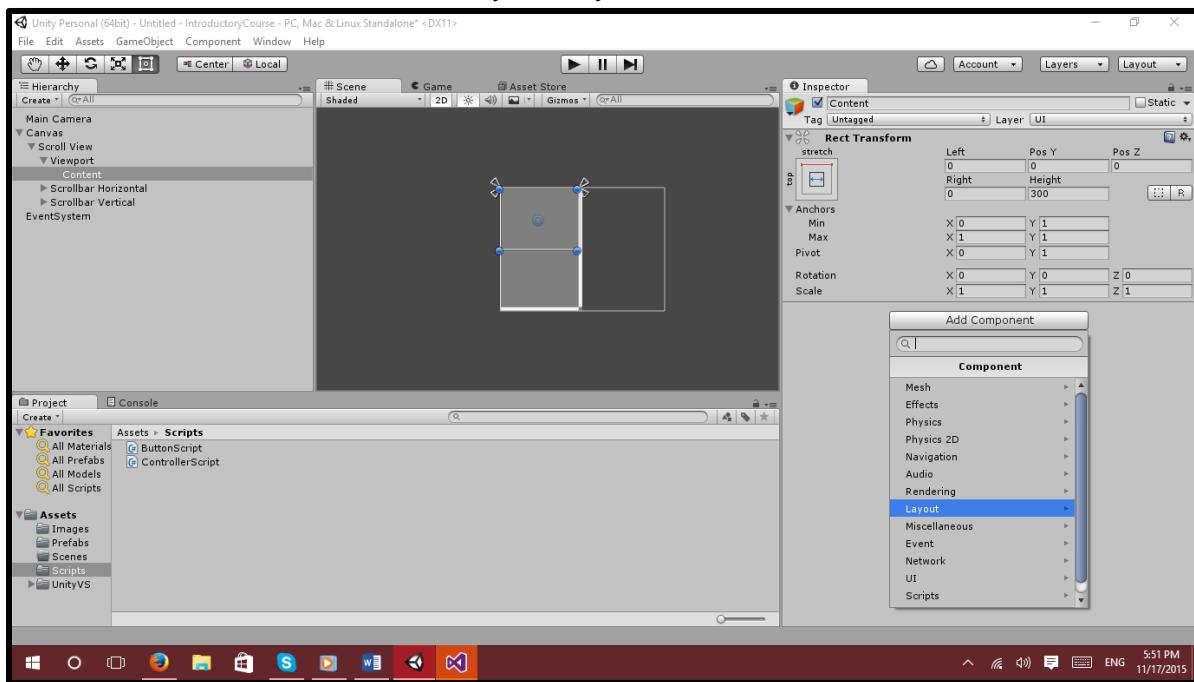
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

We will use a layout element to allow us to display the element in a uniform manner. Click on Content, click add component in the Inspector and click on Layout.



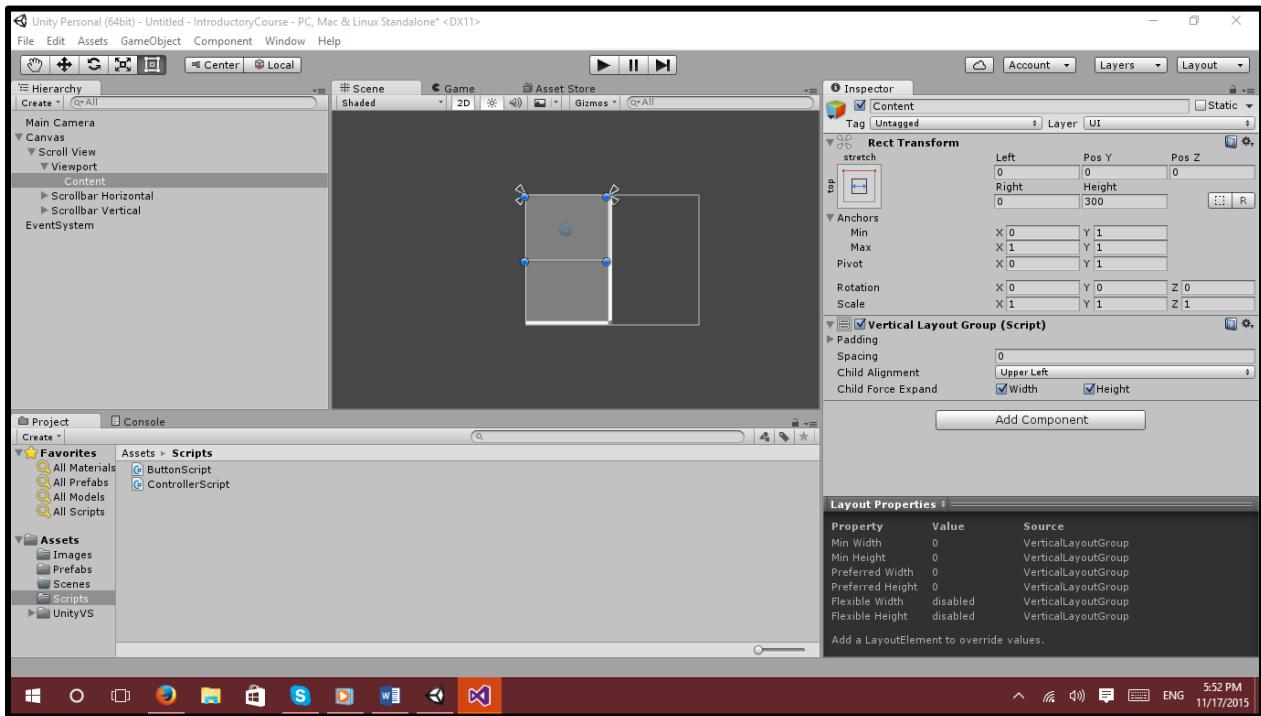
Now it gives us a list of the different Layout options available to us. We want to work with the Vertical Layout option. However, please feel free to play with the other Layout options. On a side note, I like the Grid and Vertical Layouts myself.



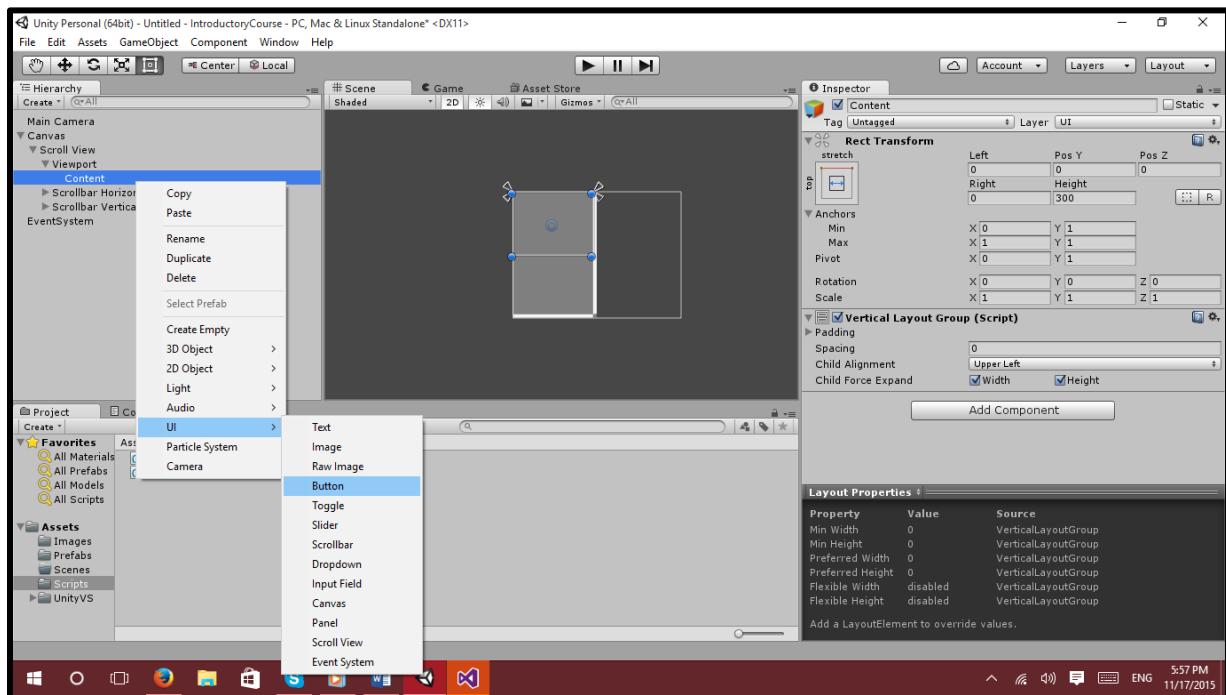
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Your Content Panel should now look like the picture below:



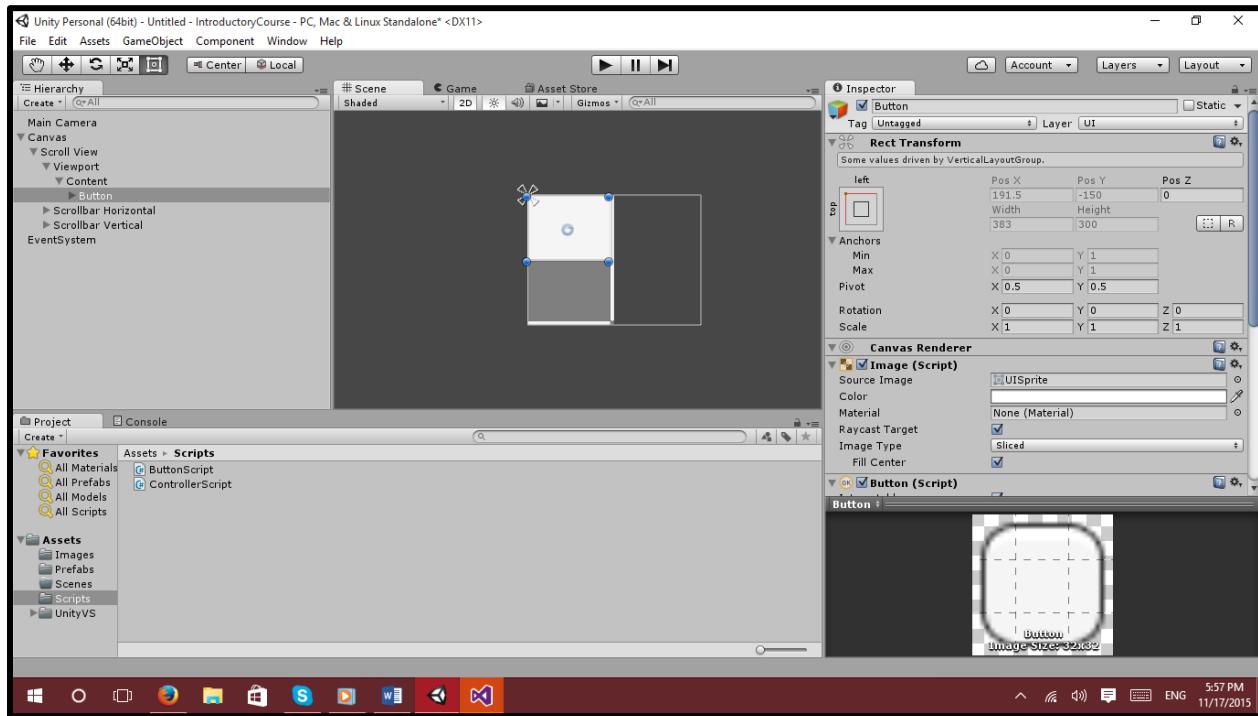
Now, we can add a button to the content panel. We will need this to create a Prefab.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Unity's editor should look like this now. If it doesn't drag the button inside of the Content Panel to reset the button as a child element. (Sometimes Unity's smart UI makes errors).



Freeze frame!

We interrupt your building the UI Section to abruptly switch to the Scripting Section!

Part 2: Scripting

We can't really go any further with the UI at the moment since we will be dynamically generating (Generating through code) most of the remaining steps of the UI.

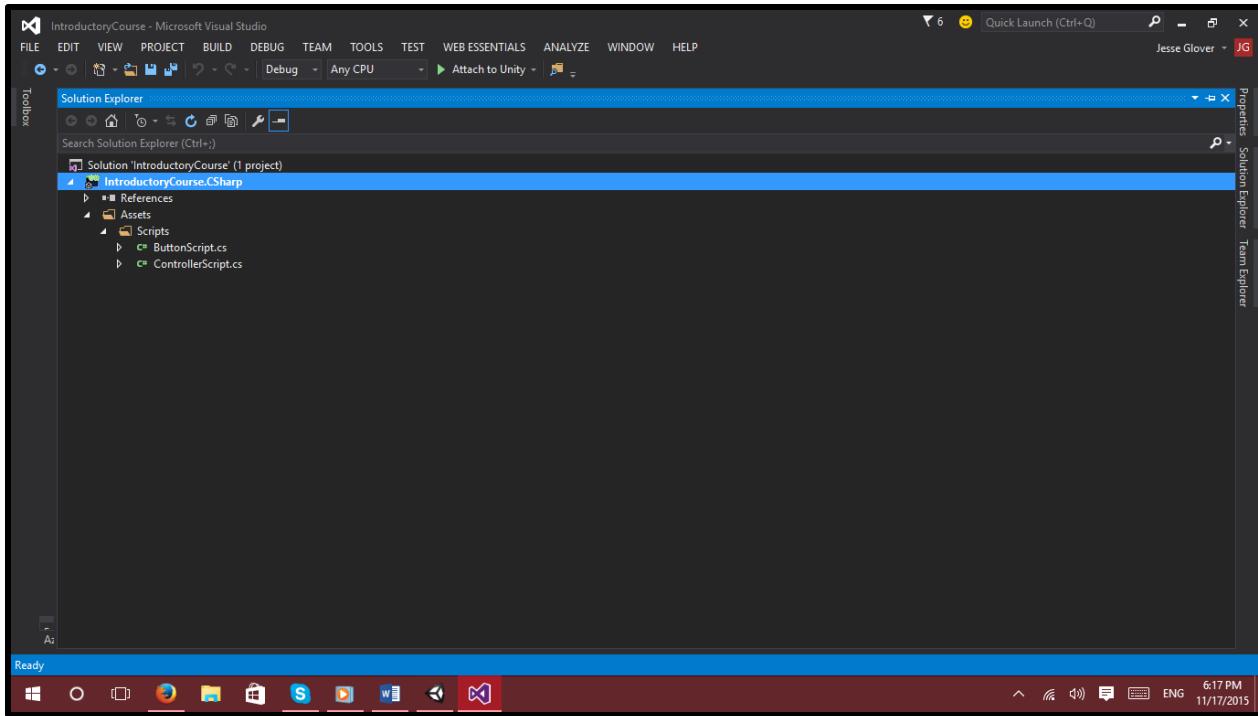
Go ahead and open up the Button Script by double clicking on it. It may take a few minutes to open the IDE for you. Don't worry, I'll wait...

Alright, your IDE of choice should now be open. Let's write some code! (I sense some apprehension about this, don't worry, it will be fun!).

However, I would like to point out in Visual Studio, the solution explorer looks slightly different than it does for a regular solution.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved



It isn't a bad idea to become familiar with how the Solution explorer looks when using a Unity Project. As you can see, it shows the CSProj as normal. References look the same as well (Although if you drill into it, you will see the libraries and dependencies that are used with Unity Projects). The big change is that you see the Assets folder and Scripts only, without any other code. The code you work with will be in the Scripts folder as you can see in the Screenshot above.

We will start with the ButtonScript since it should be the first one that opens.

The screenshot shows the Microsoft Visual Studio interface with the title bar "IntroductoryCourse - Microsoft Visual Studio". The menu bar includes FILE, EDIT, VIEW, PROJECT, BUILD, DEBUG, TEAM, TOOLS, TEST, WEB ESSENTIALS, ANALYZE, WINDOW, and HELP. The toolbar has icons for file operations like Open, Save, and Build. The status bar at the bottom right shows "Ln 46 Col 91 Ch 91 INS" and the date/time "6:15 PM 11/17/2015". The main code editor window displays the "ControllerScript.cs" file under the "IntroductoryCourse.csproj" project. The code uses namespaces UnityEngine, System.Collections, System.Collections.Generic, System, and UnityEngine.UI. It defines a public class ControllerScript : MonoBehaviour with methods Start() and AddButtons(). The AddButtons() method instantiates buttons from a prefab, gets their script components, sets button data, and attaches them to a content panel. The code editor has syntax highlighting and a vertical scrollbar.

```
1  using UnityEngine;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System;
5  using UnityEngine.UI;
6
7  [Public class ControllerScript : MonoBehaviour
8  {
9      // Viewable in Unity
10     public GameObject ButtonPrefab;
11     // Viewable in Unity
12     public GameObject ContentPanel;
13     // Viewable in Unity
14     public List<ButtonData> _data;
15     // Viewable in Unity
16     public Image ImageLoader;
17
18     void Start()
19     {
20         // On Application start, run this method
21         AddButtons();
22     }
23
24     public void AddButtons()
25     {
26         for (int i = 0; i < _data.Count; i++)
27         {
28             GameObject btn = Instantiate(ButtonPrefab) as GameObject; // Instantiates the button from the attached prefab
29             ButtonScript script = btn.GetComponent<ButtonScript>(); // Gets a reference to the script attached to the button
30             script.SetButtonData(_data[i].TextA, _data[i].buttonSprite); // remember we wrote this method in the button script
31             btn.transform.SetParent(ContentPanel.transform, false); // Attaches the button to the content panel as its child
32         }
33     }
34 }
```

We need to make sure we are using `UnityEngine.UI`, `UnityEngine`, and `System.Collections`. We want a public `Text` to make sure we can modify the parent inside the Unity Editor. And we want a private `Sprite`. We want that to occur only when we press on a button.

I should also mention that public and private methods have interesting results with Unity. A private method cannot be called by a component. For example, if we were to change `Clicked()` from a public method to a private method. A button component wouldn't see that method.

I know what you are thinking. That should be obvious, a private method can only be seen within the class itself. And I would respond with you are absolutely correct, however, I figured I should bring this up since I myself tried it and was tickled with the outcome.

Now, let's take a look at the Controller Script.

```
1  using UnityEngine;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System;
5  using UnityEngine.UI;
6
7  public class ControllerScript : MonoBehaviour
8  {
9      // Viewable in Unity
10     public GameObject ButtonPrefab;
11     // Viewable in Unity
12     public GameObject ContentPanel;
13     // Viewable in Unity
14     public List<ButtonData> _data;
15     // Viewable in Unity
16     public Image ImageLoader;
17
18     void Start()
19     {
20         // On Application start, run this method
21         AddButtons();
22     }
23
24     public void AddButtons()
25     {
26         for (int i = 0; i < _data.Count; i++)
27         {
28             GameObject btn = Instantiate(ButtonPrefab) as GameObject; // Instantiates the button from the attached prefab
29             ButtonScript script = btn.GetComponent<ButtonScript>(); // Gets a reference to the script attached to the button
30             script.SetButtonData(_data[i].TextA, _data[i].buttonSprite); // Remember we wrote this method in the button script
31             btn.transform.SetParent(ContentPanel.transform, false); // Attaches the button to the content panel as its child
32         }
33     }
34 }
```

The screenshot shows the Microsoft Visual Studio interface with the following details:

- Title Bar:** IntroductoryCourse - Microsoft Visual Studio
- Menu Bar:** FILE, EDIT, VIEW, PROJECT, BUILD, DEBUG, TEAM, TOOLS, TEST, WEB ESSENTIALS, ANALYZE, WINDOW, HELP
- Toolbar:** Standard icons for file operations.
- Solution Explorer:** Shows the project structure with files like ButtonScript.cs, ControllerScript.cs, and IntroductoryCourse.CSharp.
- Properties:** Shows settings for the selected file.
- Code Editor:** Displays the C# code for ButtonScript.cs. The code includes methods for setting an image and updating the application, as well as a Serializable class for ButtonData containing TextA and buttonSprite properties.

```
33     }
34 
35     public void SetImage(Sprite sprite)
36     {
37         ImageLoader.overrideSprite = sprite; // override the base image with the image we specify
38     }
39 
40     void Update()
41     {
42         if (Input.GetKey(KeyCode.Escape)) Application.Quit(); // << So we can easily close the app when testing on android etc
43     }
44 }
45 
46 // IMPORTANT!!! Making it Serializable allows us to see it in Unity and modify the values.
47 [Serializable]
48 public class ButtonData
49 {
50     public string TextA;
51     public Sprite buttonSprite;
52 }
53 }
```

- Status Bar:** Shows the current zoom level (100%), Azure App Service Activity, Error List, Output, and system status (Ln 46, Col 91, Ch 91, 6:15 PM, 11/17/2015).

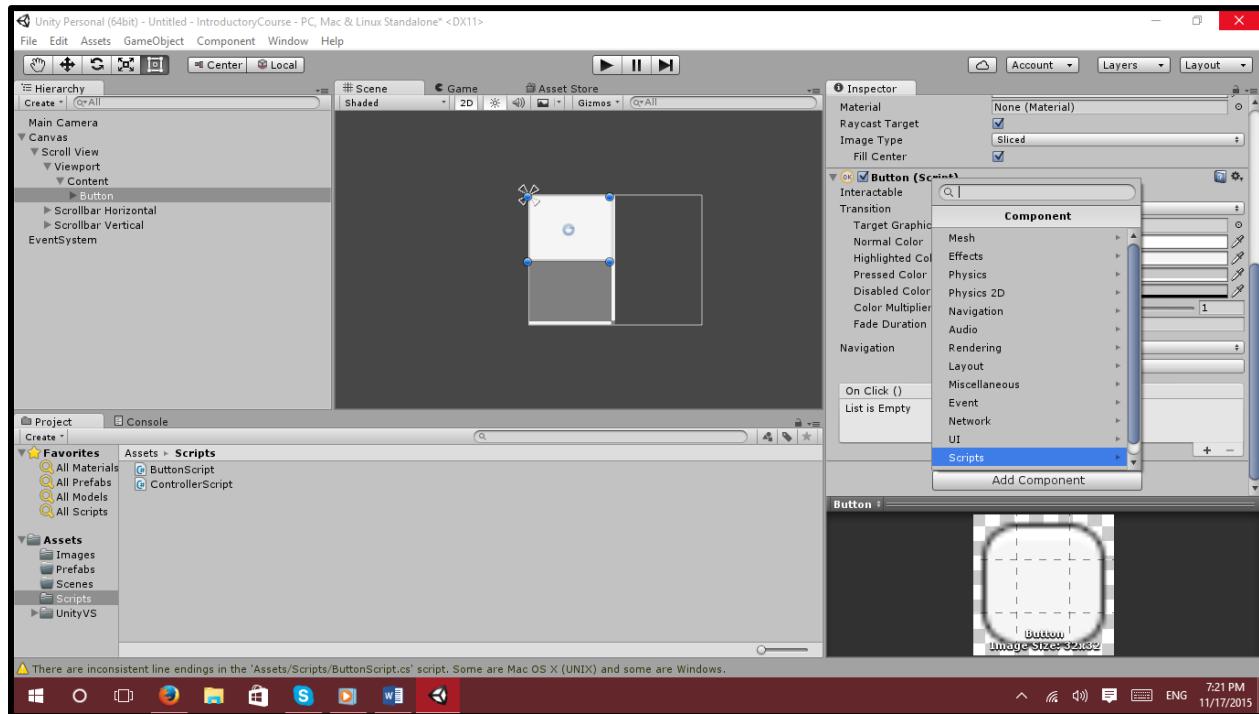
This code should be self-explanatory, so if this code confuses anyone... Please, comment below.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

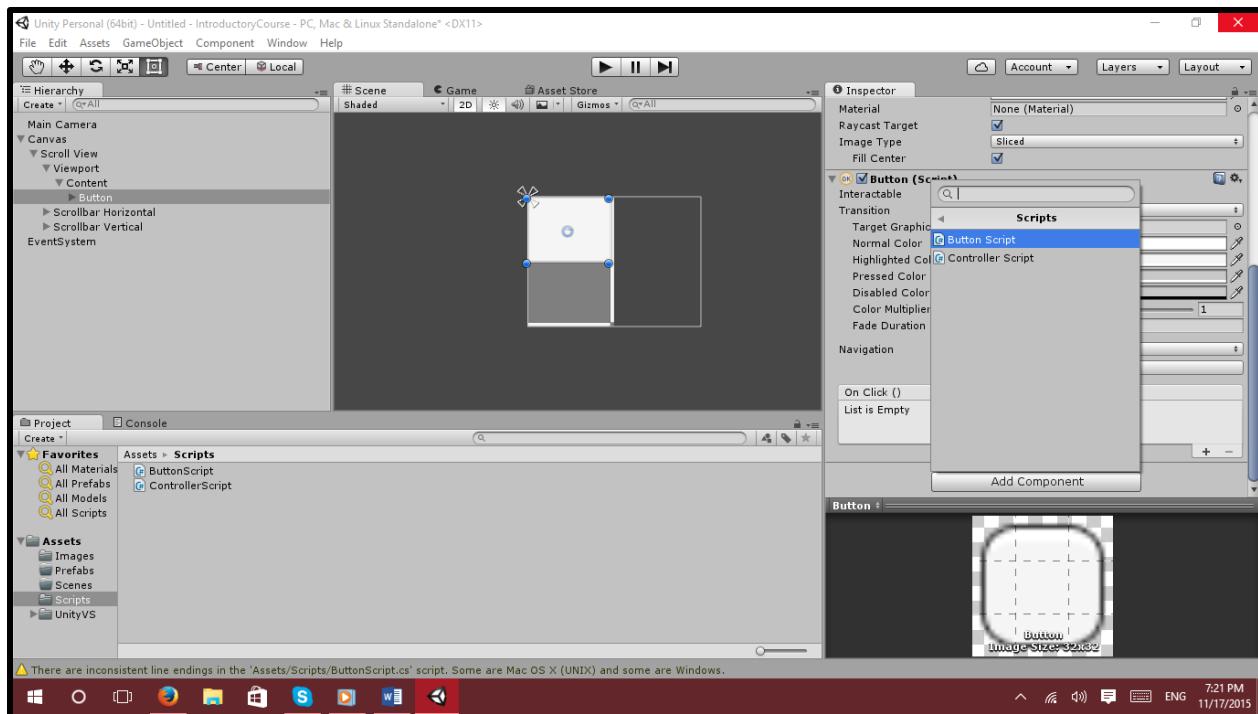
Alright, now that the hard part is out of the way, We can now move on to Part 3: Finishing Touches!

Section 1 left off with creating a button inside of the content element of the ScrollView. Now, we need to attach the Button Script to the Button and create a prefab from it.

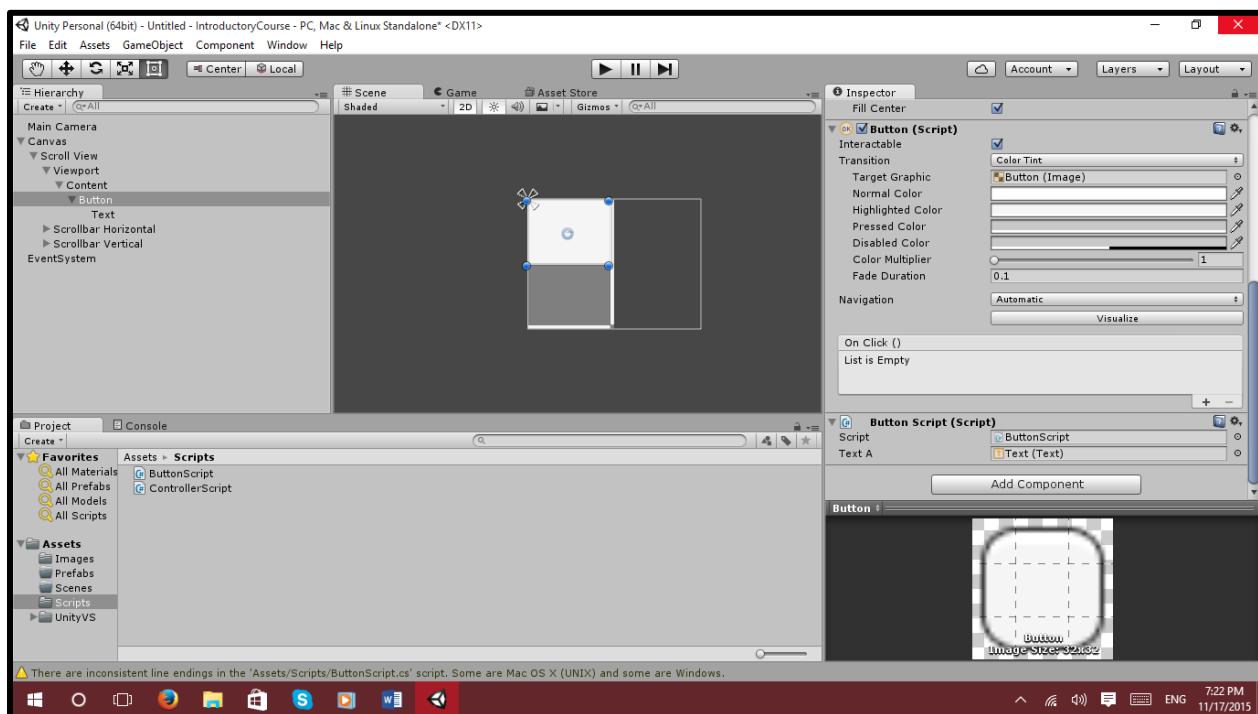
To do this, we first click on the button from the hierarchy pane, and then go to the inspector pane. We select Add Component. We want to select Scripts.



Now we need to select Button Script.



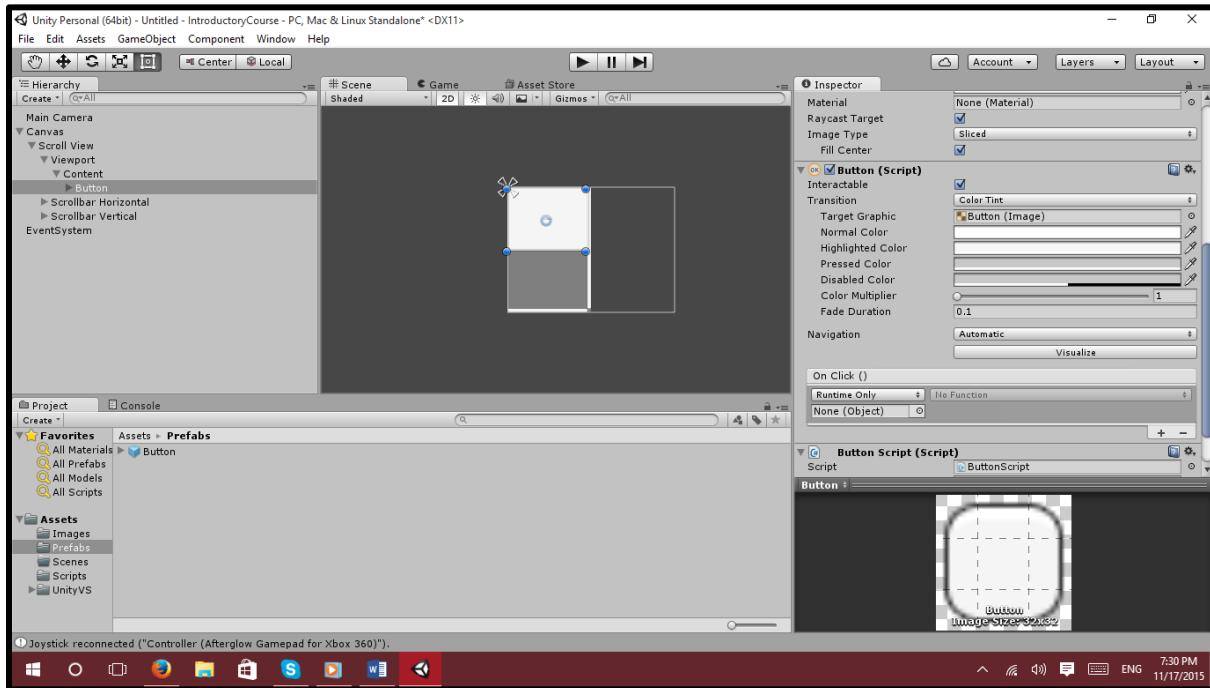
After that, make sure to add the text element from the button to Text A.



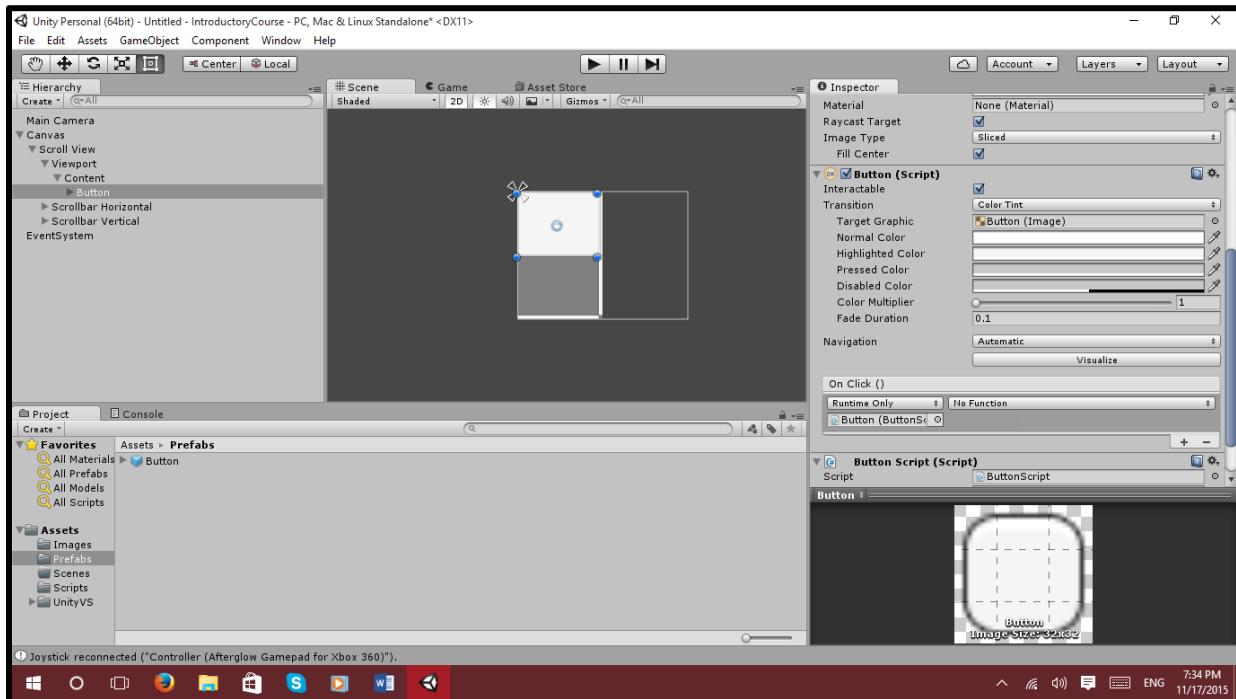
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Click the plus sign at the end of the OnClick section of the Button in the inspector, where it says None (Object), drag the button element from hierarchy pane onto that box.



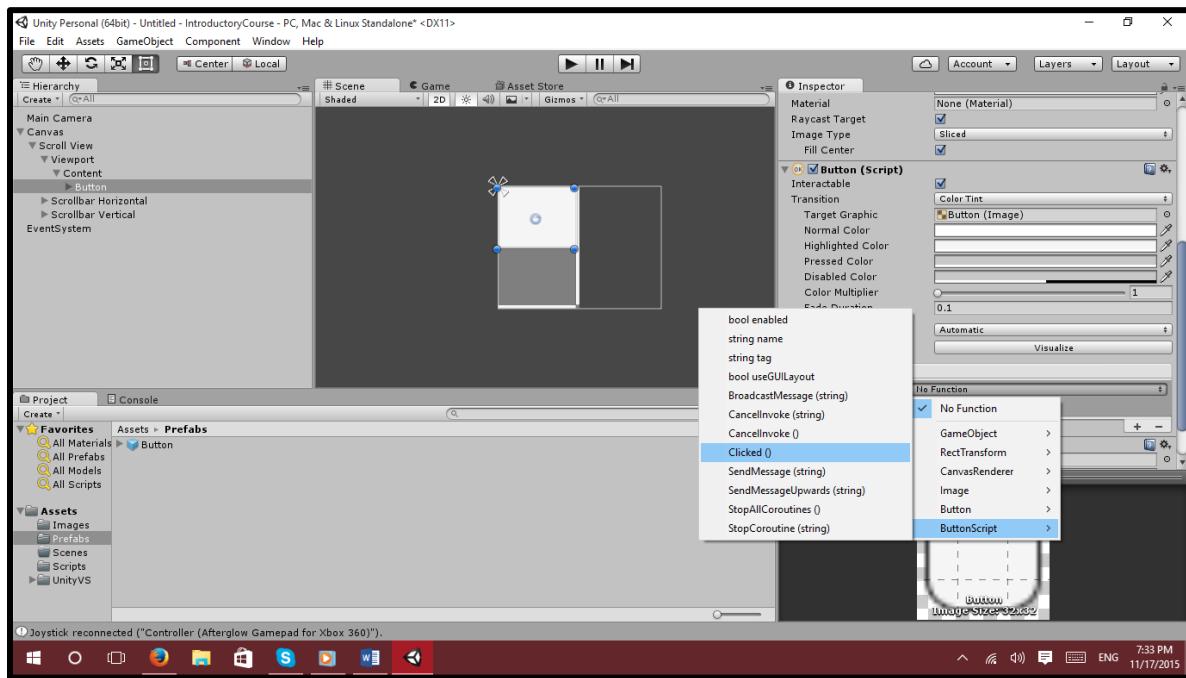
It should look like this:



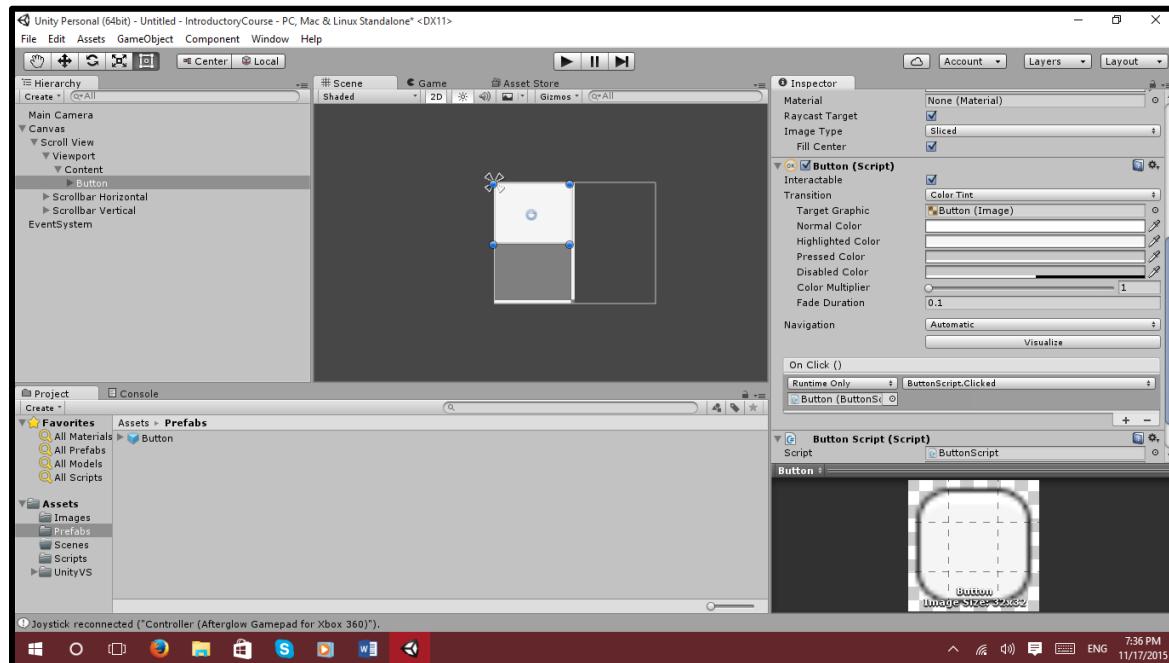
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Now that we have done this, we need to make sure that we select a function for it to execute when the button has been clicked. We want ButtonScript and Clicked(). The same name as the one in the method.

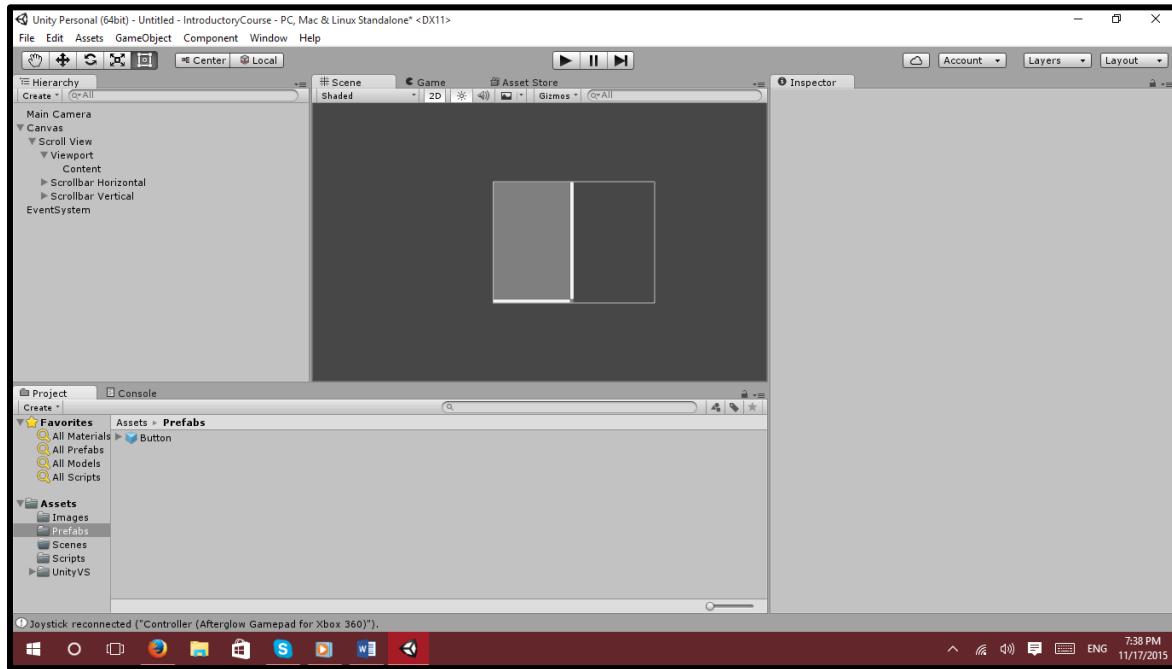


It should now look like this:

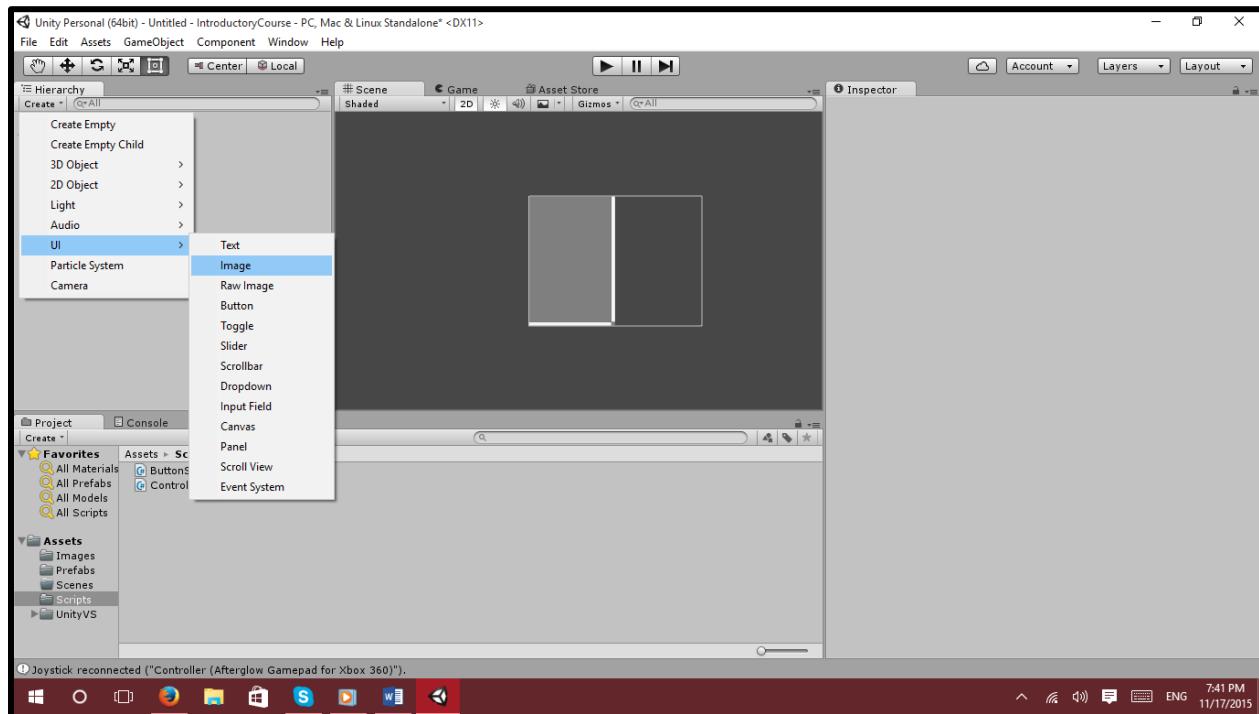


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

We are almost done with the button, Drag the button from the Hierarchy Pane into the prefabs folder. It should turn blue. And then simply delete the button from the Hierarchy Pane.



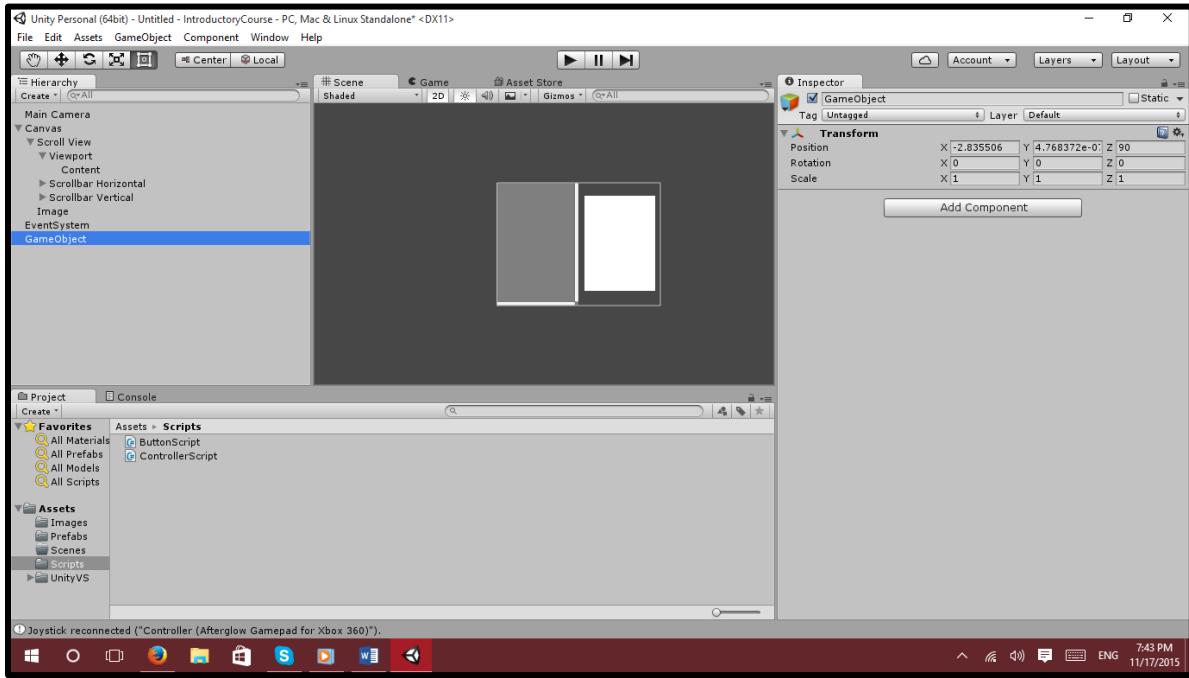
Now we need to add the Image Component to the Canvas



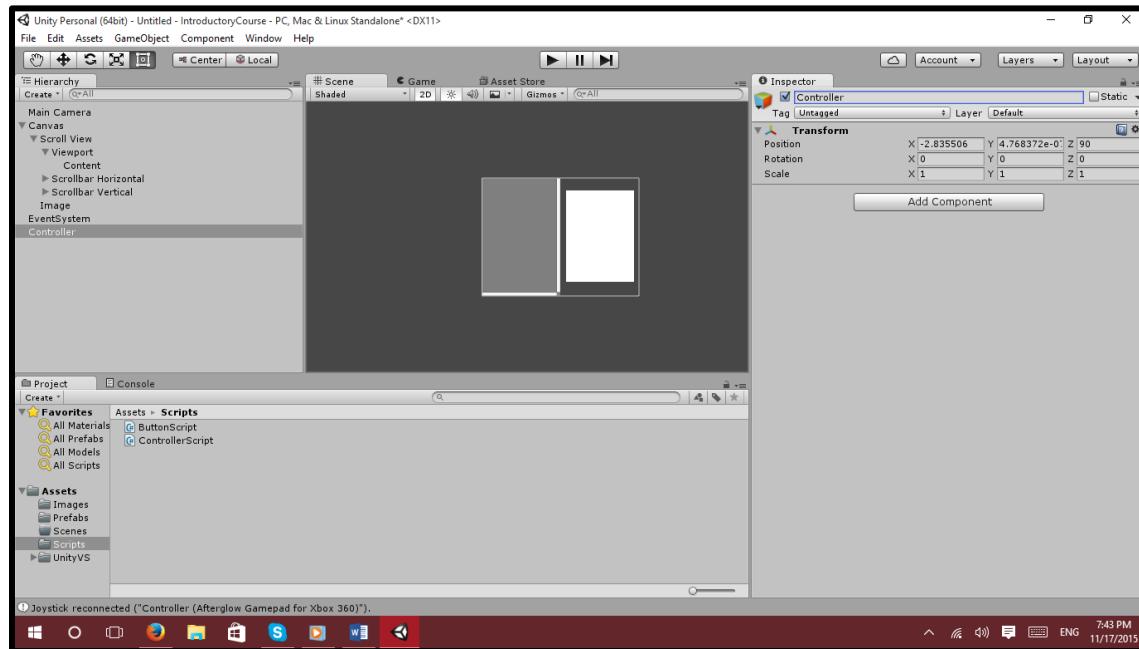
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Next, we need to add an empty object

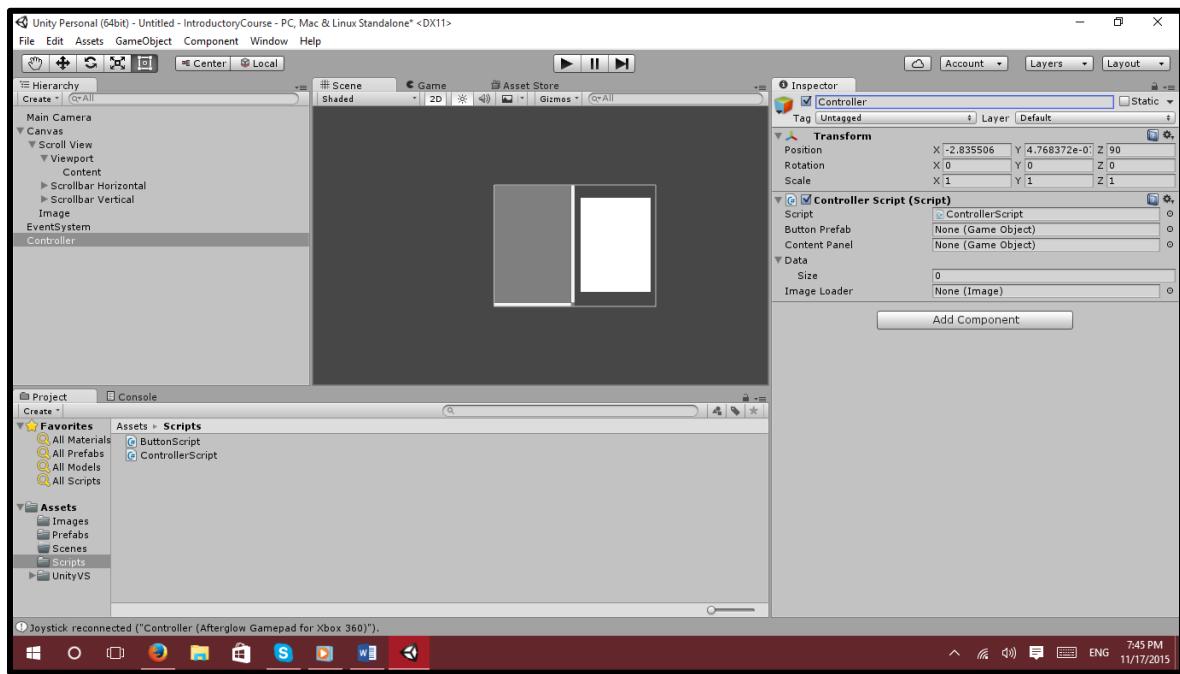


Now to rename it to Controller:



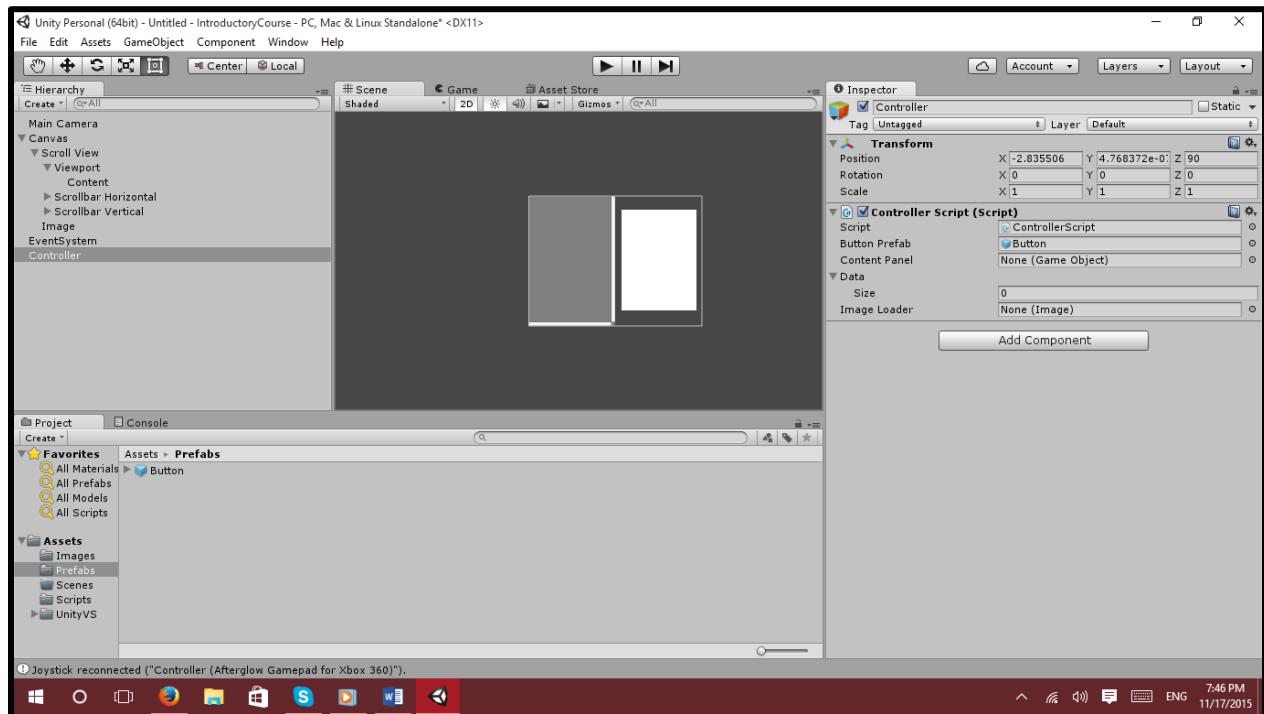
Add component time. Click Add Component, Scripts, Controller Script.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity



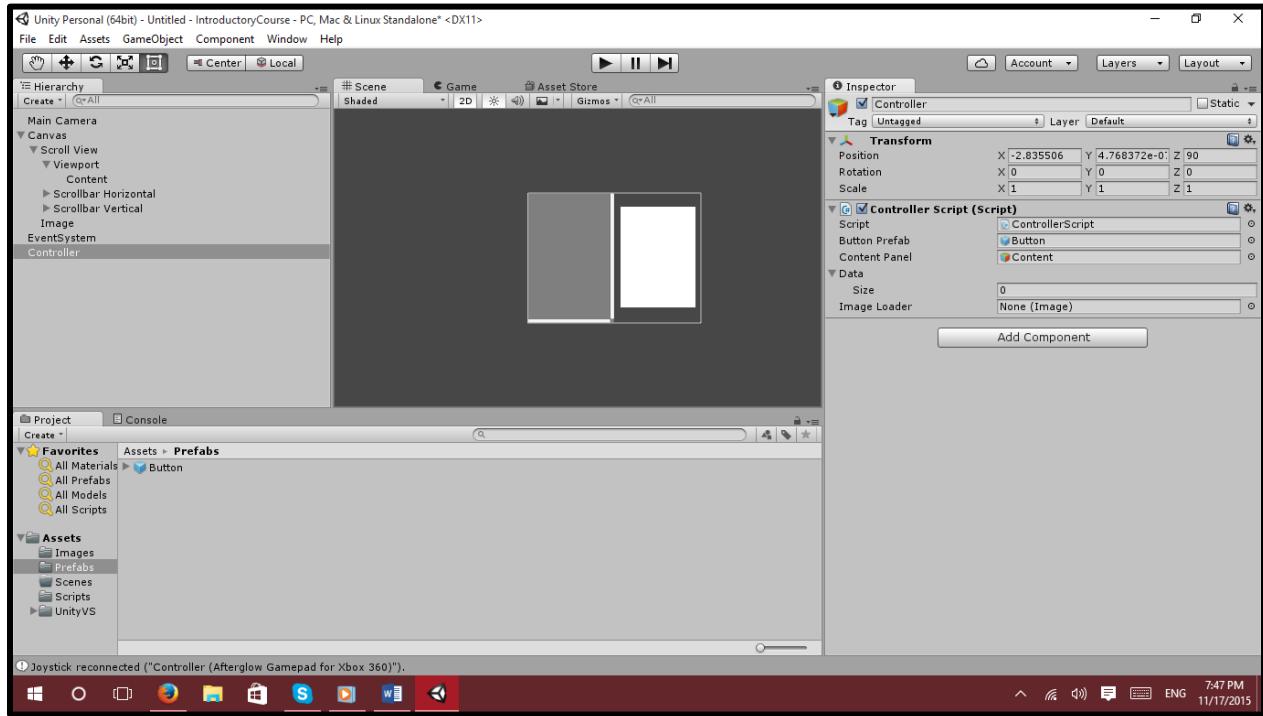
All we have to do now is wire it all up.

Button prefab should have the Button we put into the prefabs folder in it.

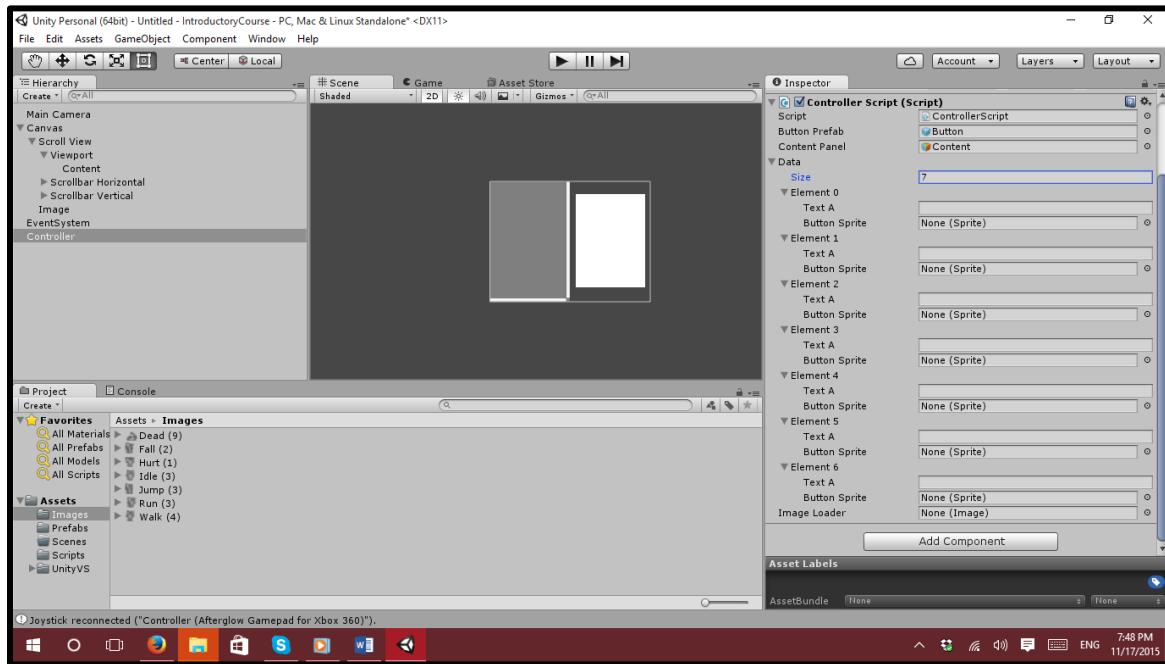


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

The Content Panel should have the Content from the Viewport in it.



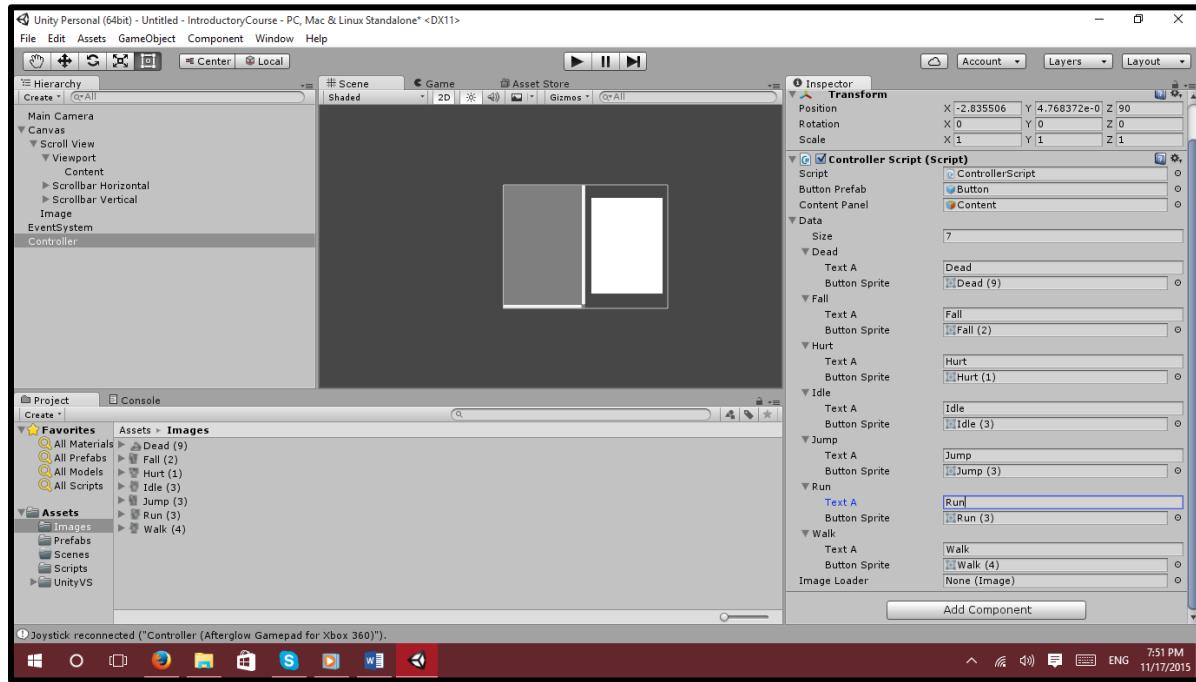
Now, change the size of the Data to be 7.



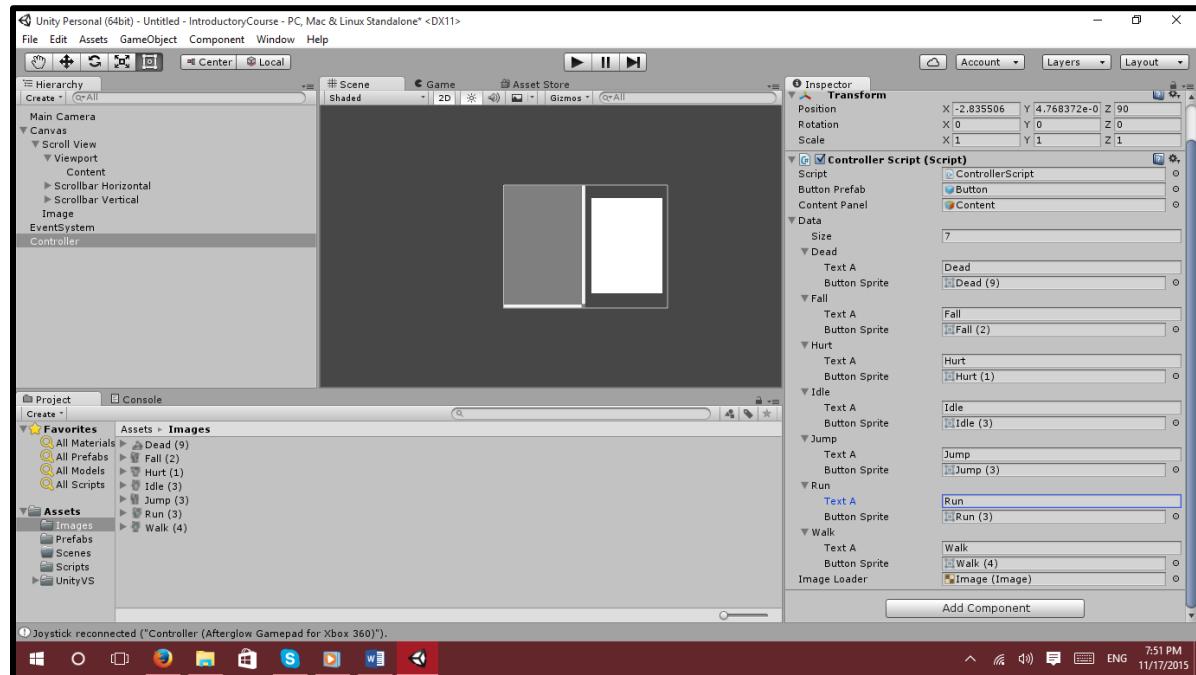
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Take each image and put it into an individual Button Sprite, name it whatever you want. I will name it according to the name of the picture.

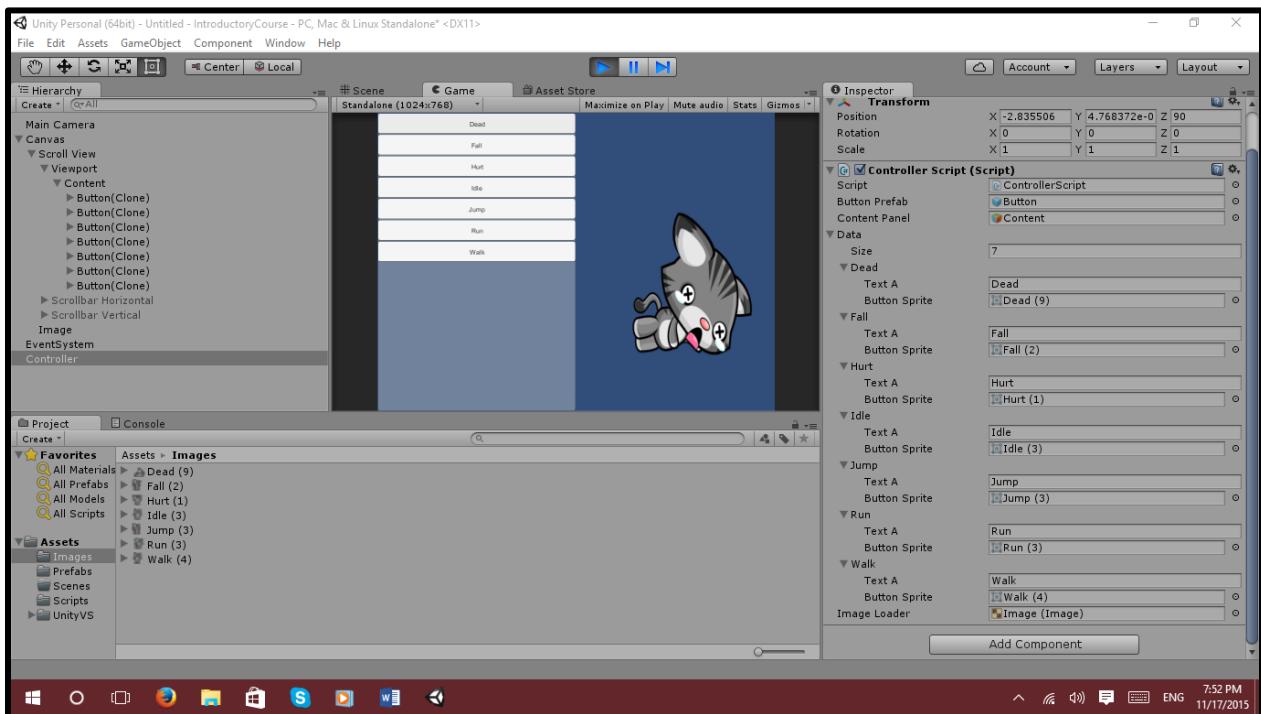
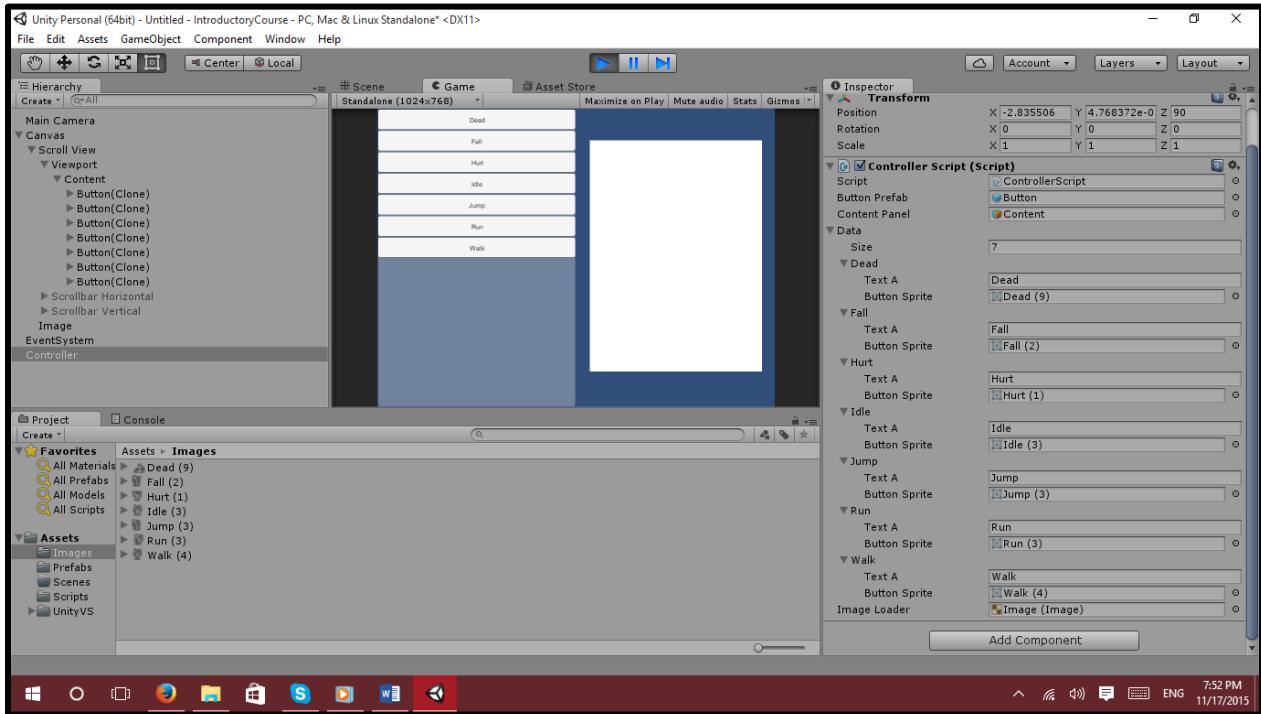


Lastly, the Image Loader needs to have the Image that we added into the Hierarchy Panel in it.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

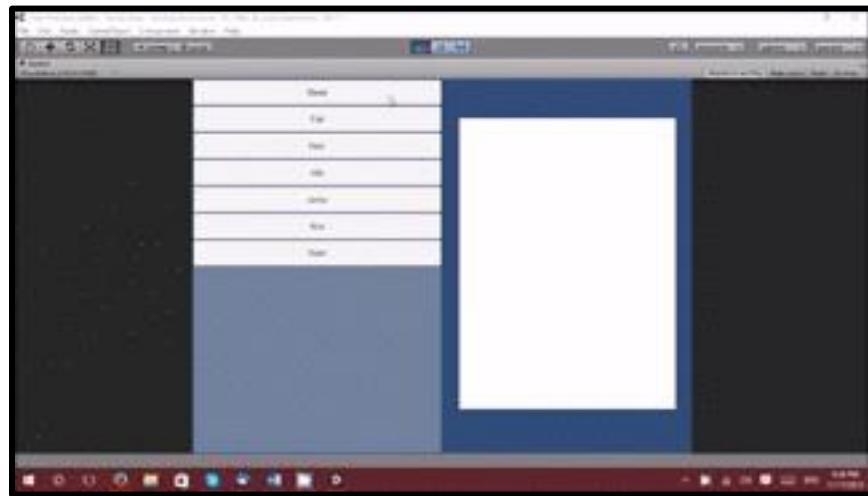
Now, let's run the program.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Here is a gif of what should occur during playback!



You have now completed the Introduction to Unity3D. Give yourself a pat on the back for a job well done!

How to Build a Complete 2D Platformer in Unity

By Tim Bonzon

Introduction

Since the release of Unity 2017.1 in the summer of 2017, Unity Technologies has made making 2D games incredibly easy and fast. In this tutorial, we will create a fully-featured 2D platformer. This project will incorporate a number of key topics including how to make cutscenes, how to quickly build and prototype a 2D level, and how to precisely choreograph game objects using the Timeline Editor. This tutorial can also be thought of as the culmination of several tutorials – posted on Game Dev Academy – about these topics. You can check them out here:

- [Storytelling in Unity – Part 1: Virtual Cameras](#)
- [Storytelling in Unity – Part 2: Animation Tracks](#)
- [Cinemachine and Timeline Editor for Unity 2D Game Development](#)
- [Mastering Unity's New Tilemap Editor: Building 2D Levels](#)

Project Details

Let's think about what we will be making. As said in the introduction, it will be a 2D platformer. We will use the 2D character from the Unity Standard Assets pack. Then we will create an environment using tilemaps and Unity's new Tilemap Editor. Then we will create an enemy that has an Idle/Attack animation (you'll see). Then we will create a cutscene where the camera views the entire level then zooms on the character: here we will use the Timeline Editor and Cinemachine for 2D.

You can download the complete Unity project [here](#).

Tilemap Editor, Timeline, and Cinemachine

When Unity Technologies released the Unity 2017 version cycle, they introduced three pivotal tools designed to make Unity more accessible to artists. These tools were the Tilemap Editor, Timeline, and Cinemachine. The Tilemap Editor, released later in the 2017 cycle, allowed users to "...literally paint directly in Unity" according to Rus Scammell, the project manager of Unity for 2D. The Tilemap Editor gives you the ability to create vast and complicated Tilemaps without having to use a third-party program. The Timeline Editor and Cinemachine were released at the same time, though improvements to Cinemachine were released later. Cinemachine is a suite of cameras that allows you to create cutscenes, specify how the camera tracks a game object, and, in the end, allows you to tell a better story. Combine this with the Timeline Editor, a tool that allows you to choreograph game objects like a movie editor. With these two tools, you can

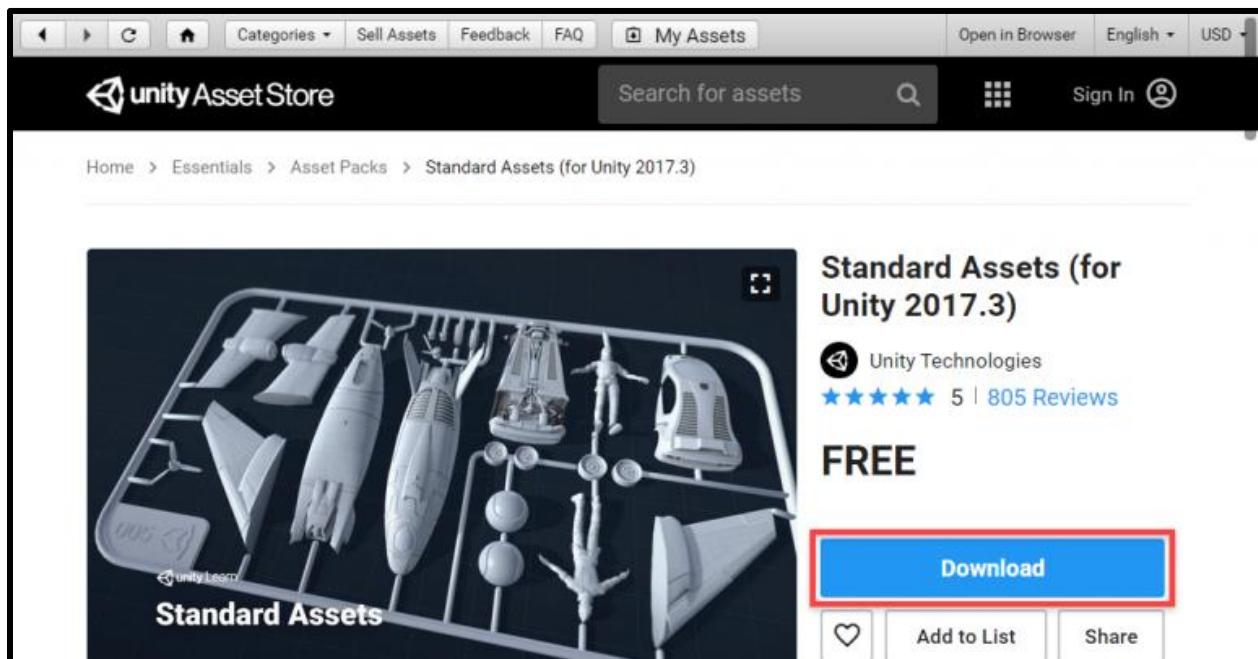
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

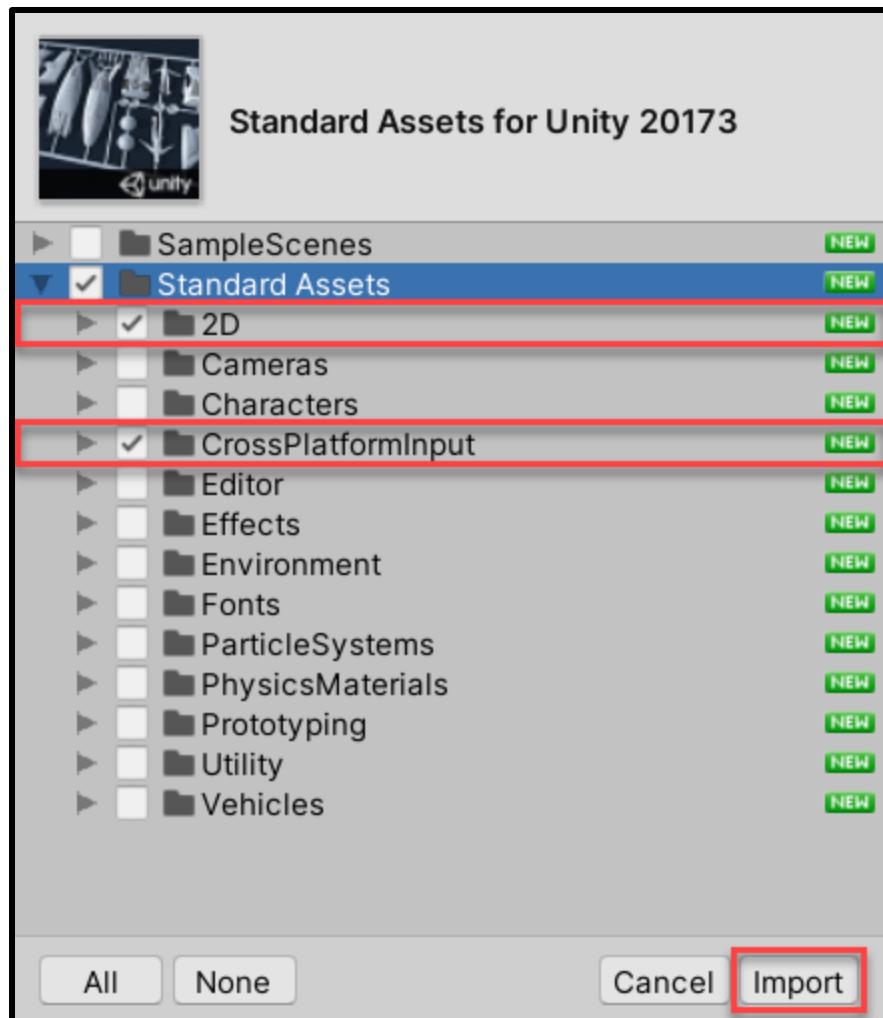
create stunning compositions with having to write any code. Well, that's a summary of the tools that we will be using! This tutorial is by no means exhaustive, for more information about these tools check out the tutorials linked above.

Setting up our project

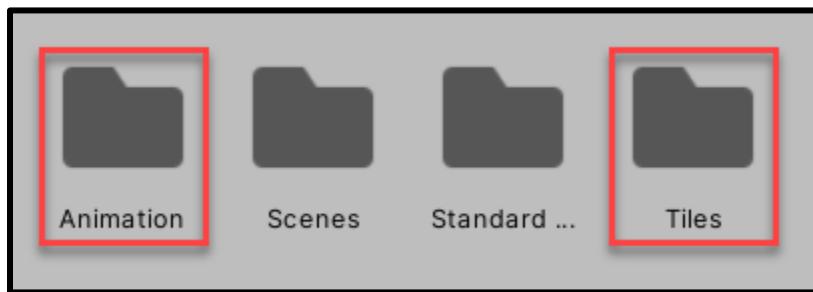
The assets for this project you can get [here](#). Then create a new Unity project. Let's import the 2D Standard Assets package by going to the Asset Store panel. Here, search for "Standard Assets" and download it.



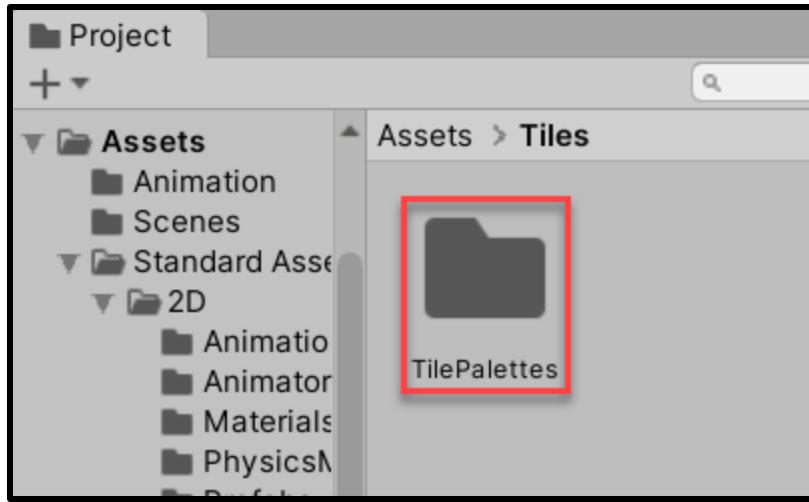
Then click on the Import button to import the asset. When the import window pops up, select the *Standard Assets > 2D* and *StandardAssets > CrossPlatformInput* folders only. Then import those.



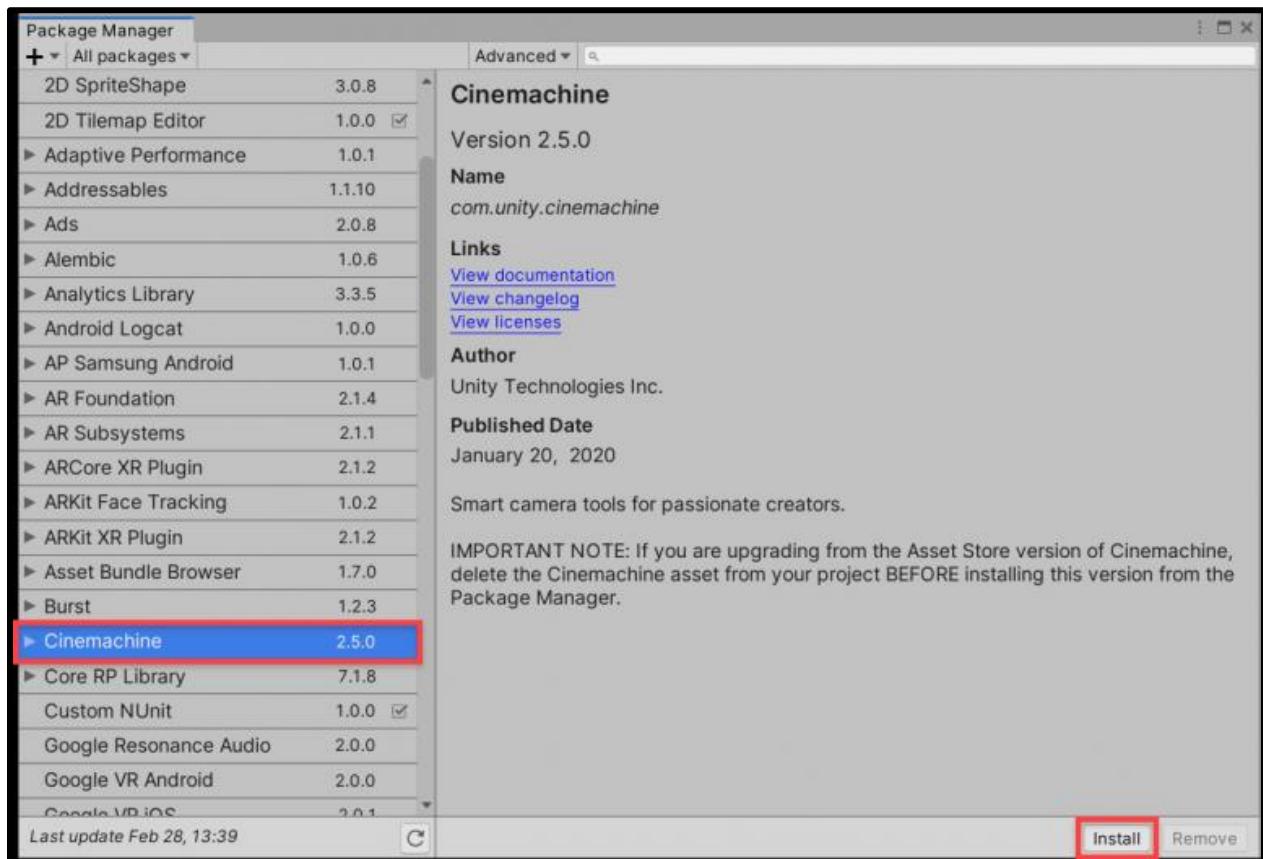
Next, we need to create two new folders. One called “Animations” and the other called “Tiles”.



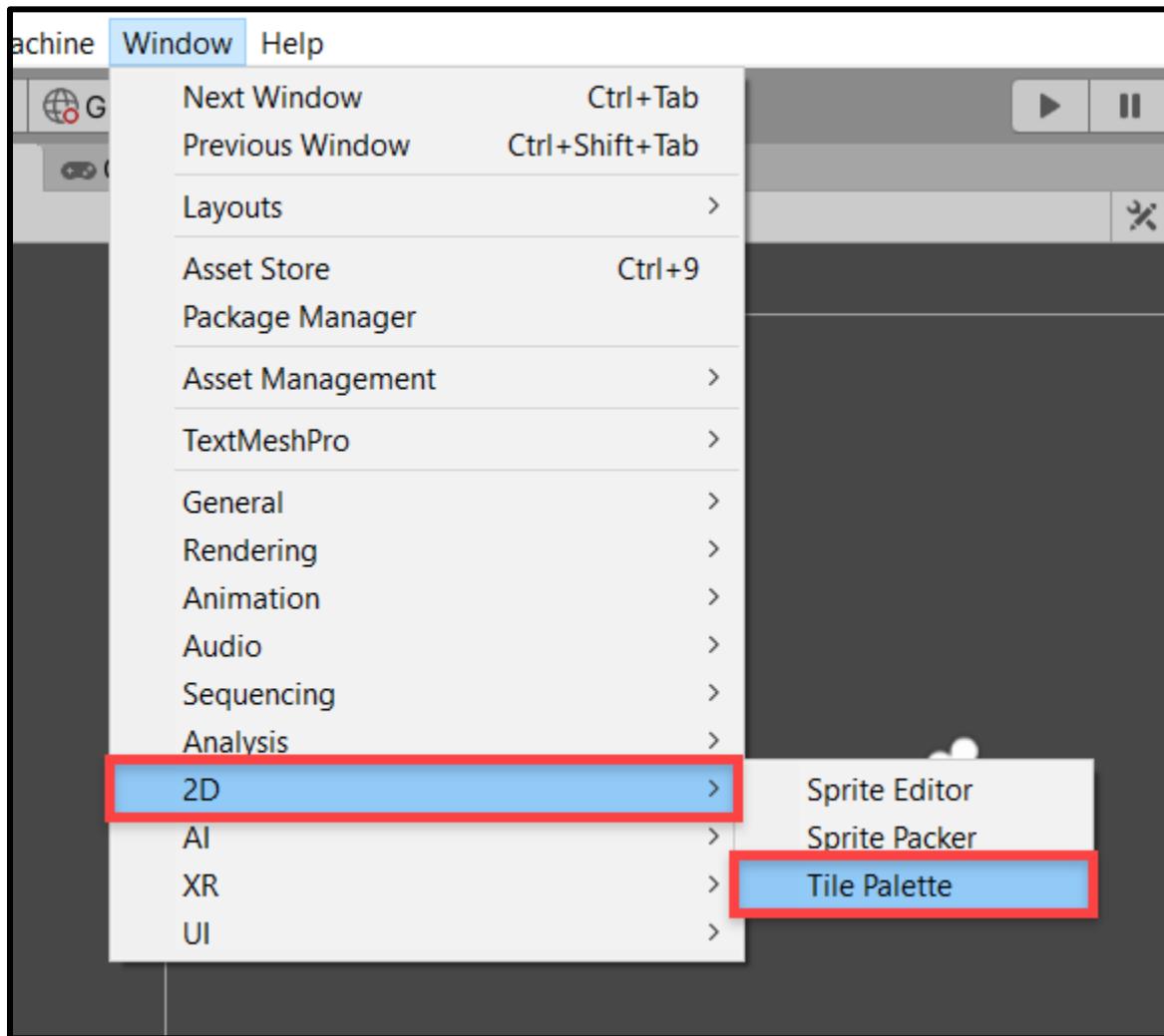
In the Tiles folder, create another folder called “Tile Palettes”.



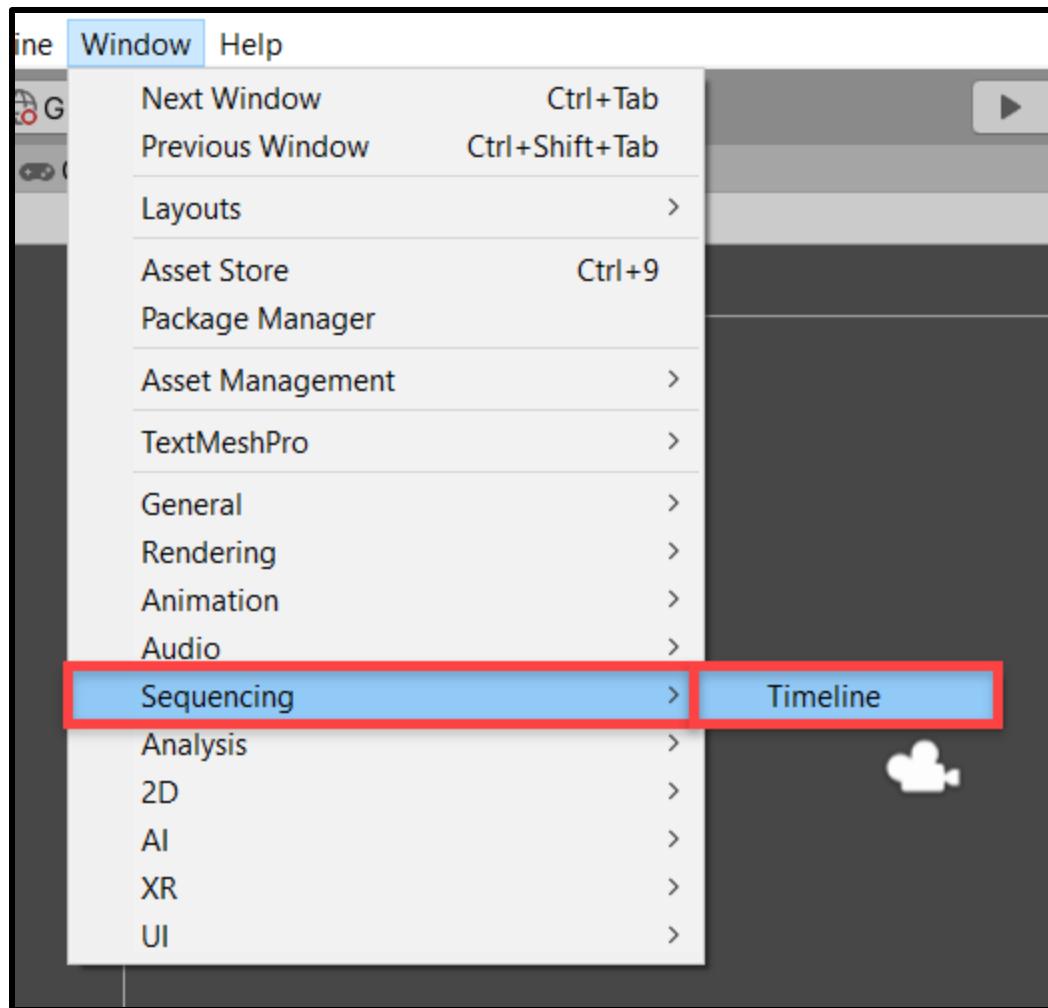
The use of this folder will become apparent later on. Now let's import Cinemachine by going to Package Manager (Window > Package Manager) and installing "Cinemachine".



Now that we have everything imported we can start getting our tools in order. We will start with the Tilemap Editor. Go to *Window > 2D > Tile Palette*.



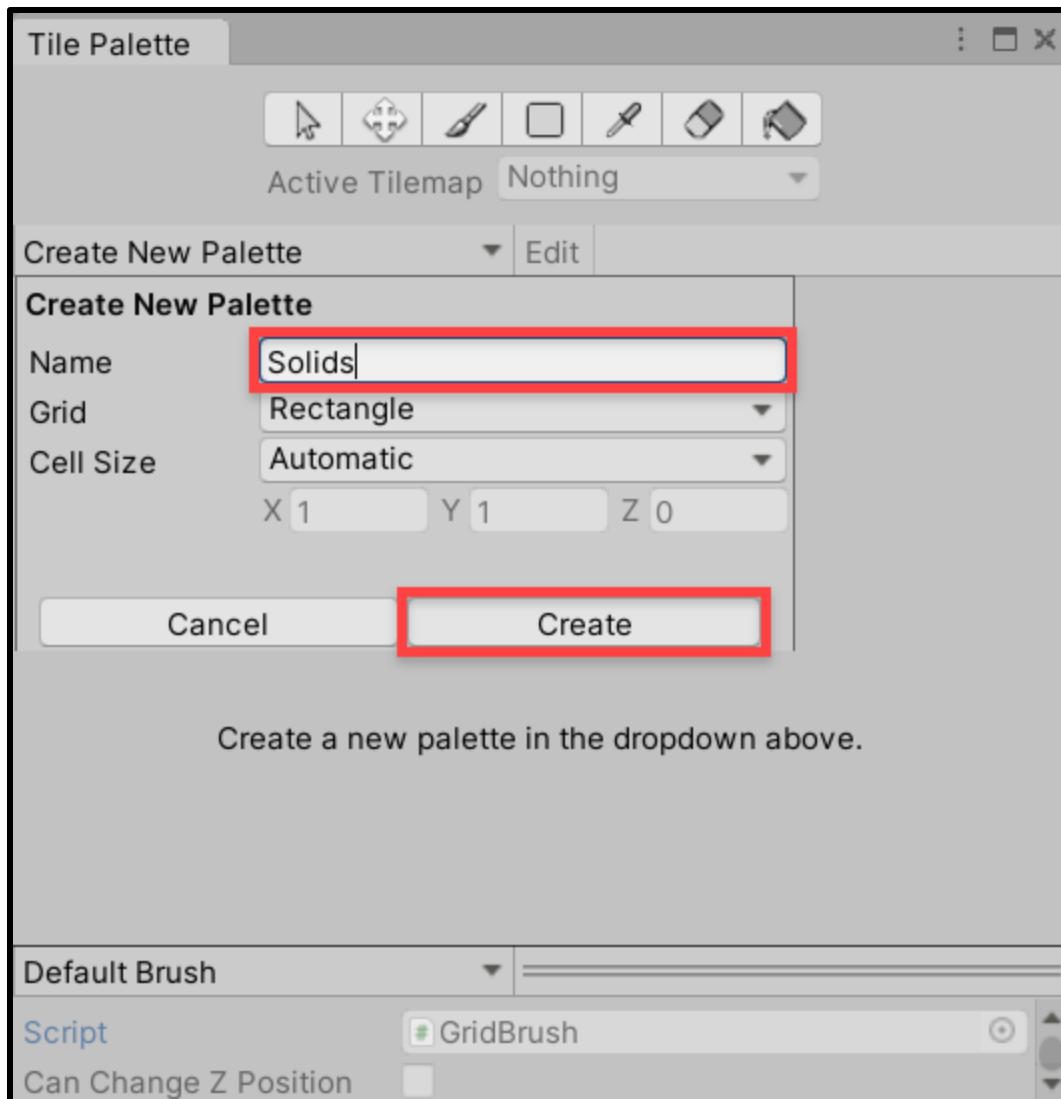
Place it in a sensible place on your workspace. I chose to put it in the space between the inspector and the scene view. Next, we need the Timeline Editor. Go to *Window > Sequencing > Timeline*.



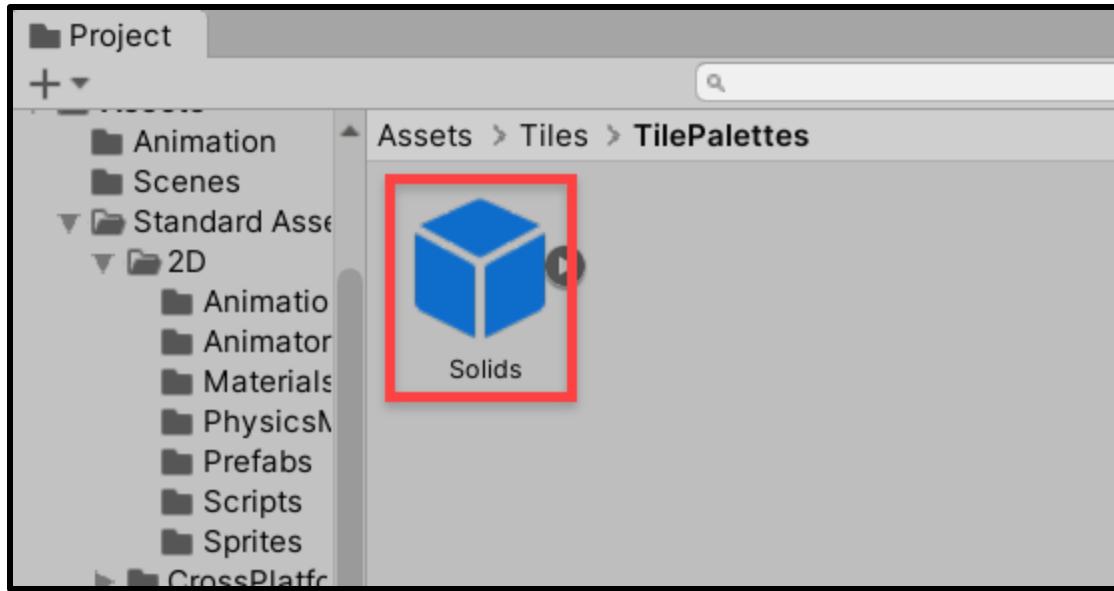
The position of this tab is not set in stone so be prepared to move it around. We now have our workspace in order! Time to start creating!

Creating our environment

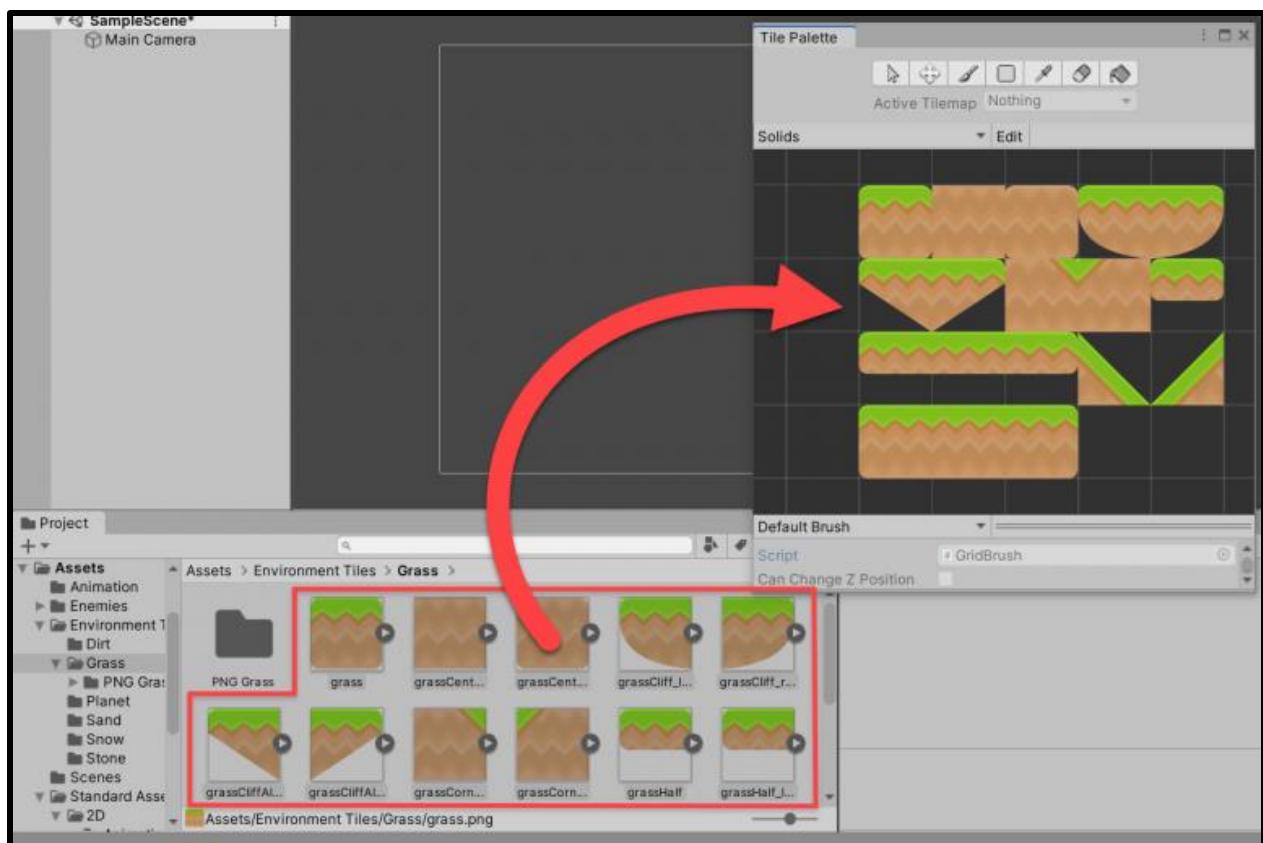
Go to your Tile Palette tab and create a new palette. Call it “Solids” since these are the tiles that the character will not be able to pass through.



A palette works just like you would expect it to based on the name, it is a set of images that we use to “paint” with. Leave all of the settings set to default and save it in the Tile Palettes folder which we created in the Tiles folder.



To setup our tiles, go to the *Environment Tiles* > *Grass* folder and drag those sprites into the *Tile Palette* window (do the same for any other tiles you want). Save the asset files to the *Tiles* folder.

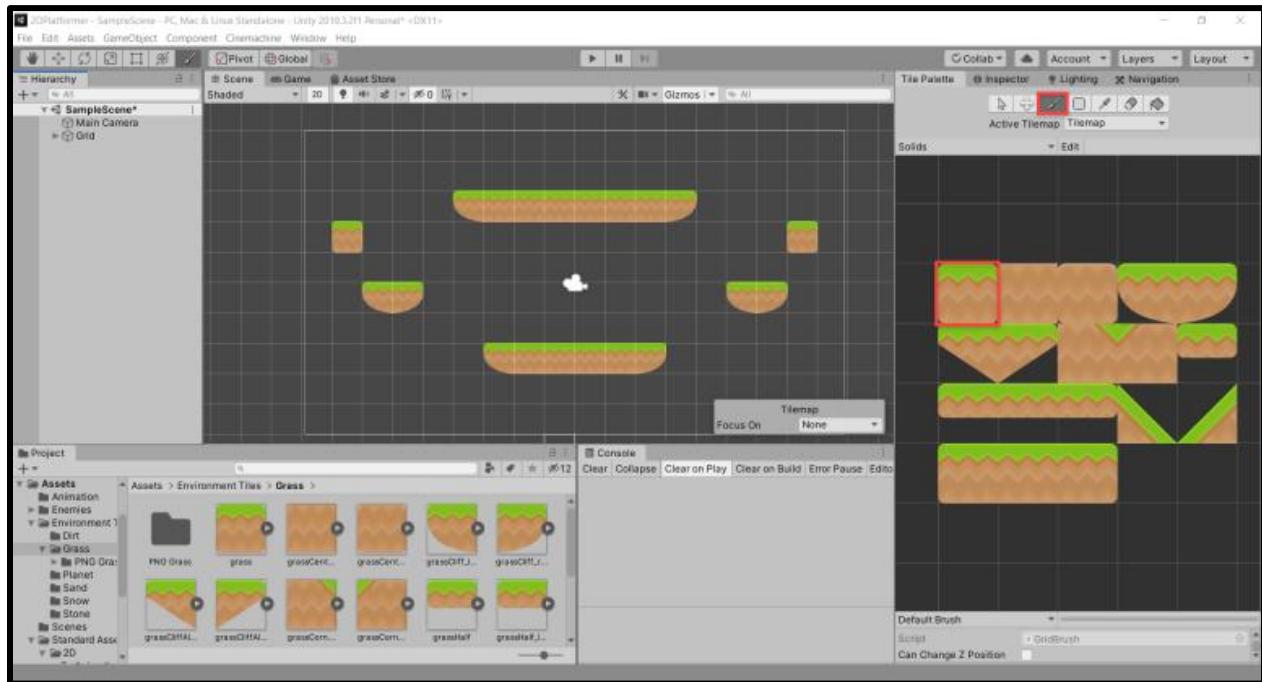


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

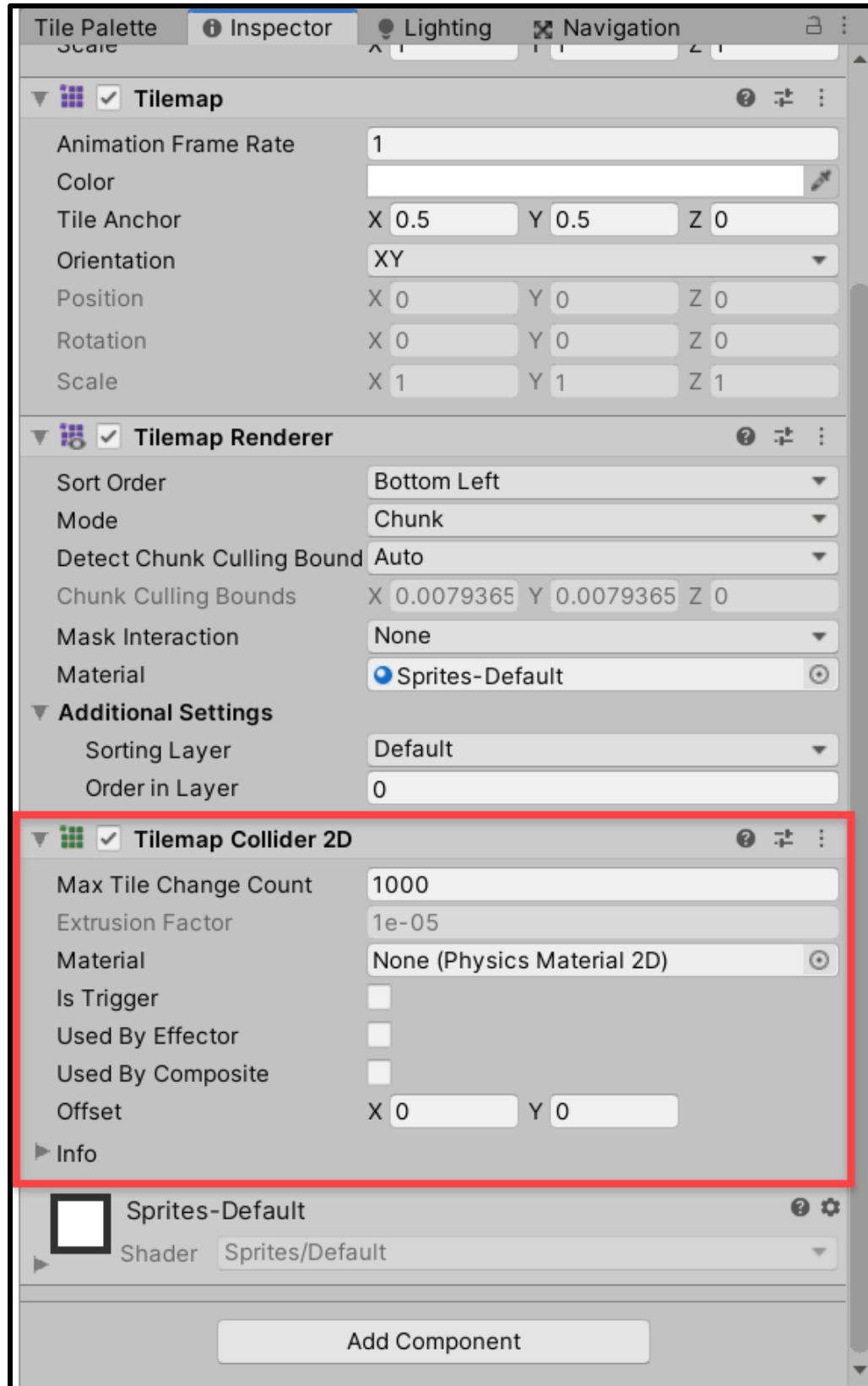
Now that all of our tiles are in order, let's create a "canvas" to paint on. In your hierarchy, right-click and go to *2D Object > Tilemap*.

What it has done is created a grid, and inside that grid is our "canvas", also known as a tilemap. In order to start painting you need to familiarize yourself with the toolbar in the Tile Palette tab. With your rule tile selected, start experimenting with the various types of brushes. Once you feel pretty confident with the tools, start creating your level!



Adding the character

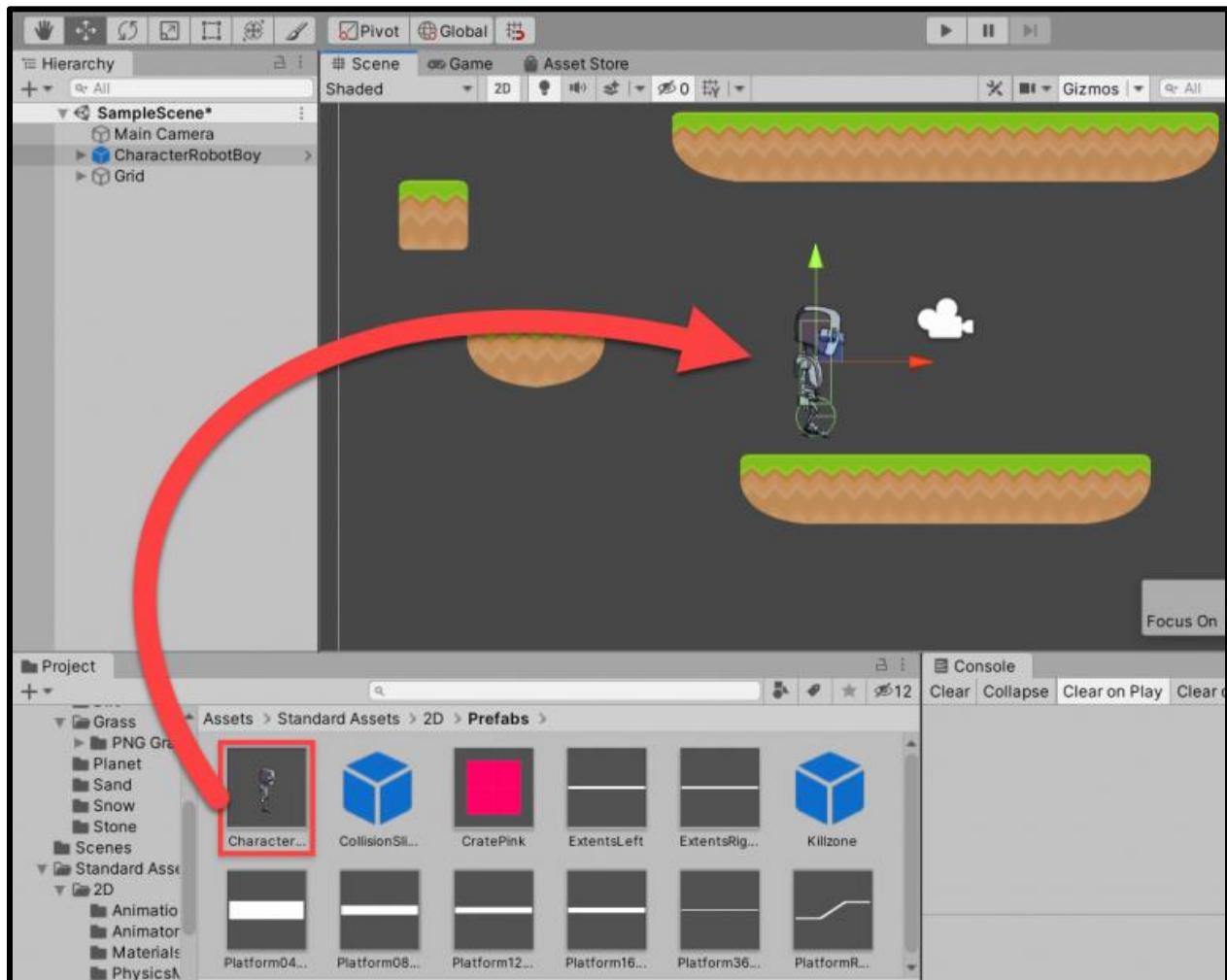
The last thing we have to do to our tilemap is to make it have physics interactions. Right now anything we put on it would just fall through the world. To fix this, Unity Technologies released a new collider component called the Tilemap Collider. This behaves just like you would expect it to based on the title, it creates a collider around each tile. Go to your Tilemap and click Add Component.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Search “CharacterRobotBody” and drag the character into the scene.



We will be using the default character that came with the 2D standard asset pack. You can find the character by either going to Standard Assets -> 2D -> Prefabs and then dragging in the “CharacterRobotBoy”, or, you can just search “CharacterRobotBoy” and access it from there. Once the character is in the scene, you can hit play and move the character through the arrow keys. You may have to reposition the camera in order to see the robot. Great! On to the next paragraph!

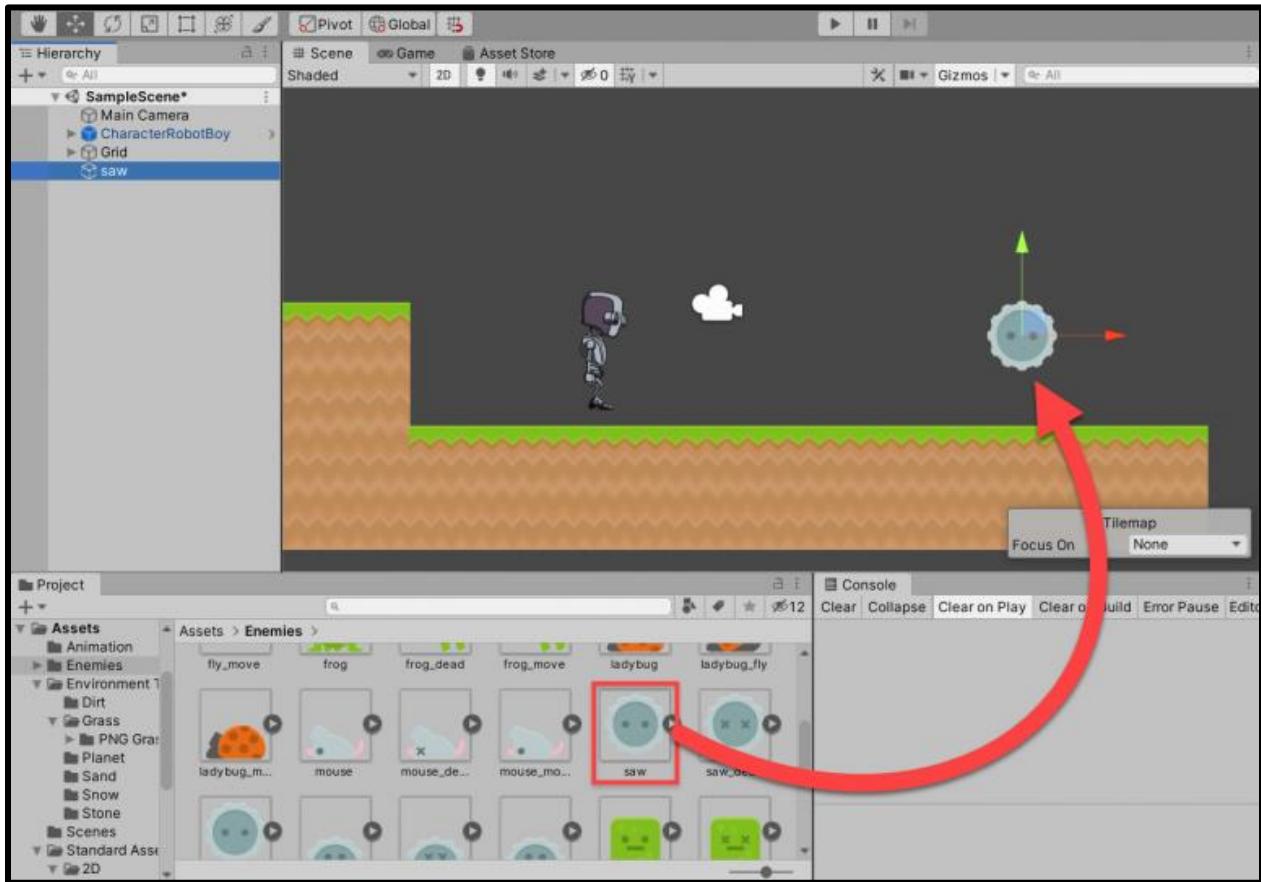
Creating the enemy

In the “Enemies” folder from the asset pack, pick a certain enemy that you think would go well with your scene.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

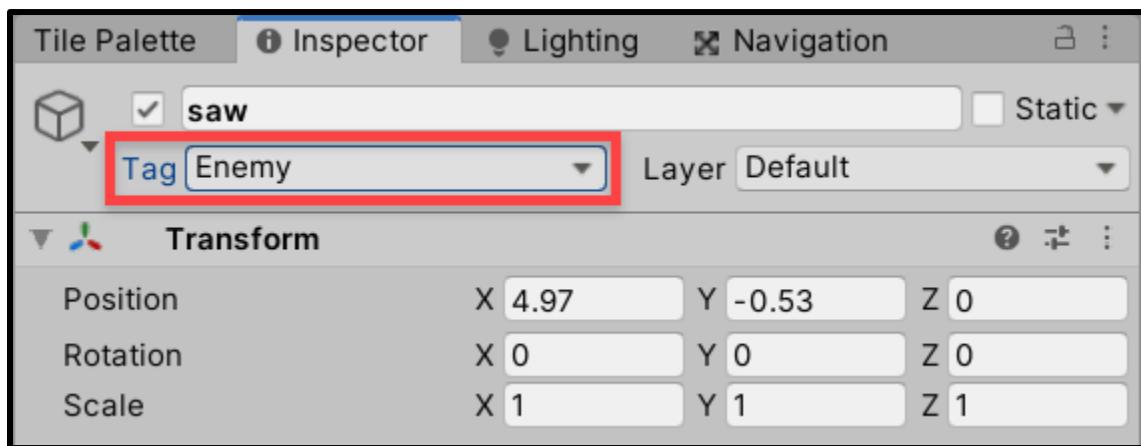
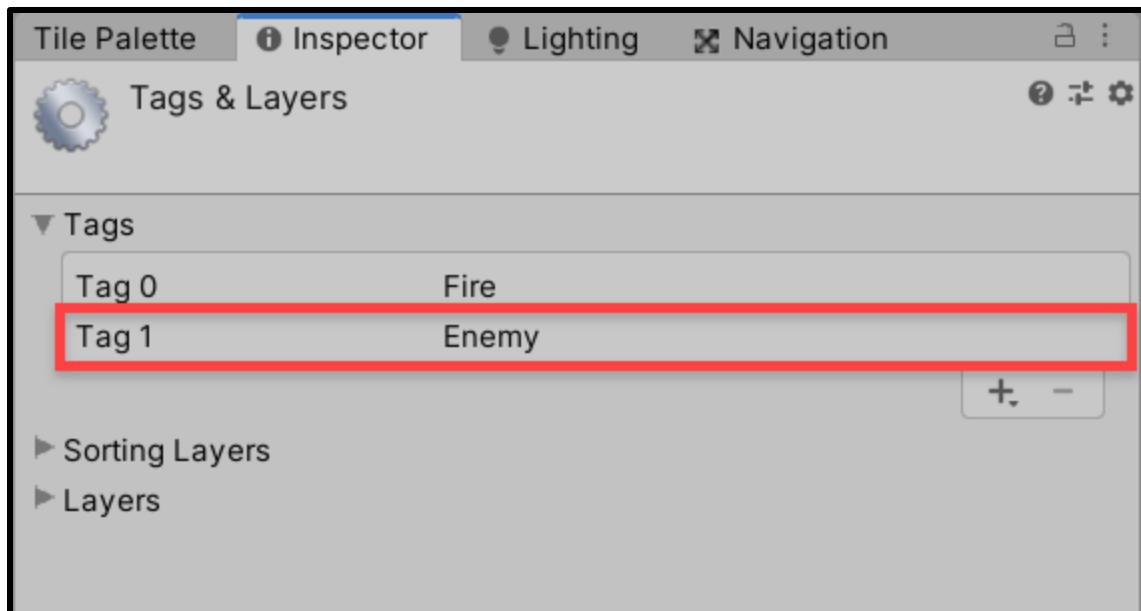


Drag it into your scene and place it in a sensible spot.

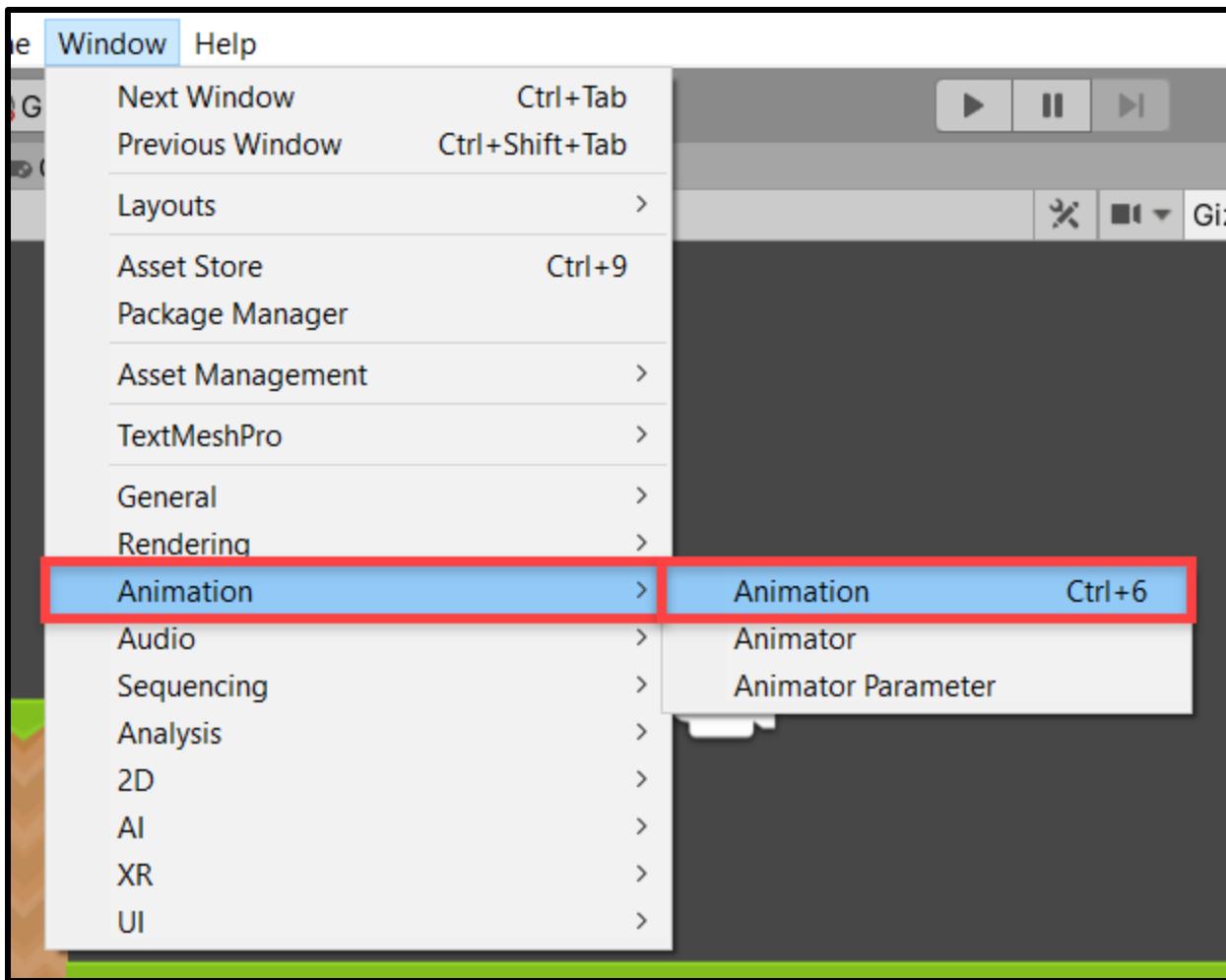


Then create a new tag called “Enemy” and assign it to your enemy.

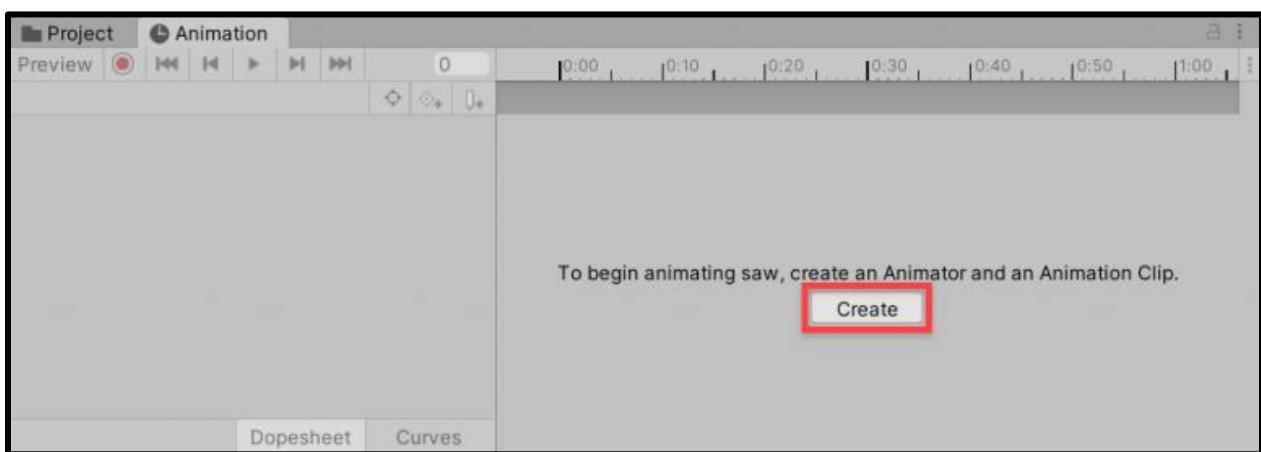
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity



Now we are going to animate this enemy. You should already have the Animation tab open in the lower window of your workspace. If not, go to *Window > Animation > Animation*.



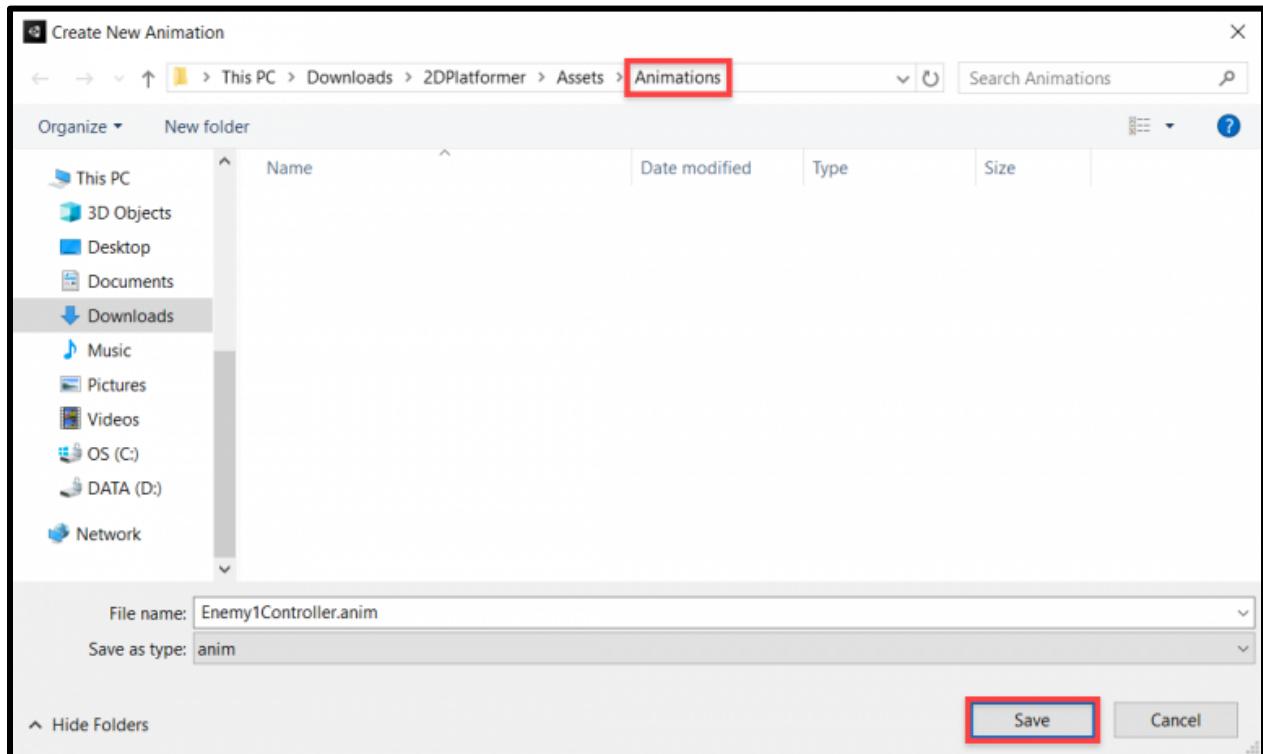
With your enemy selected, click “Create” in the Animation tab.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Name it “Enemy1Controller”.

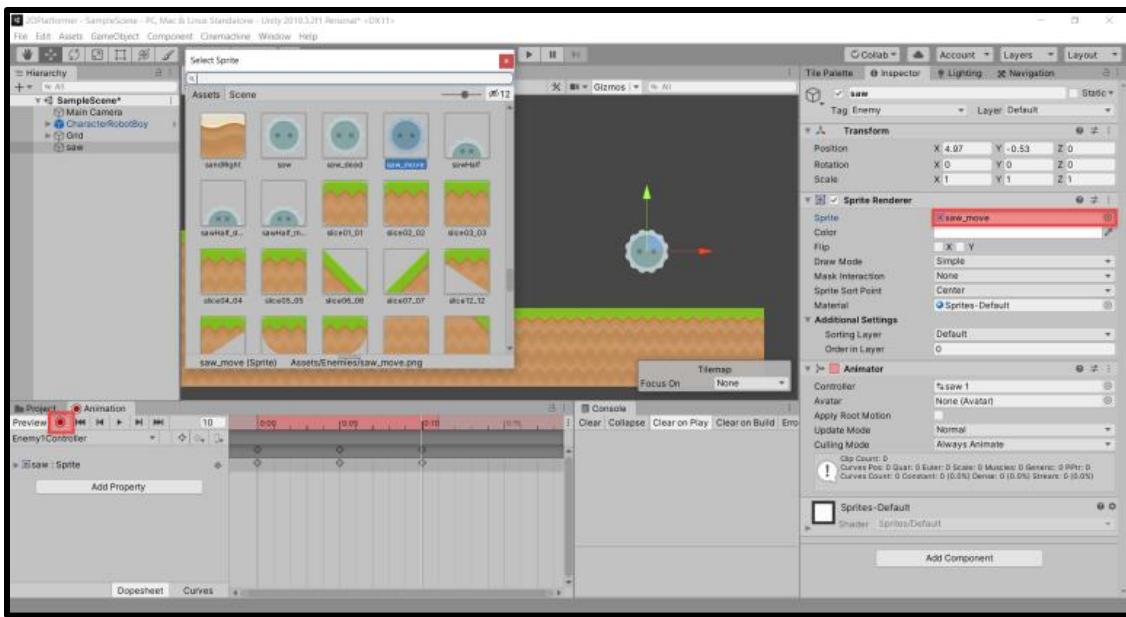


Save the animator controller in the Animations folder we created earlier.

Hit the record button and change the Sprite field to the other image that was provided (in my case it was called Saw_move, it may be similar if you chose a different enemy).

Then move about four frames ahead and change the image back to what it was before.

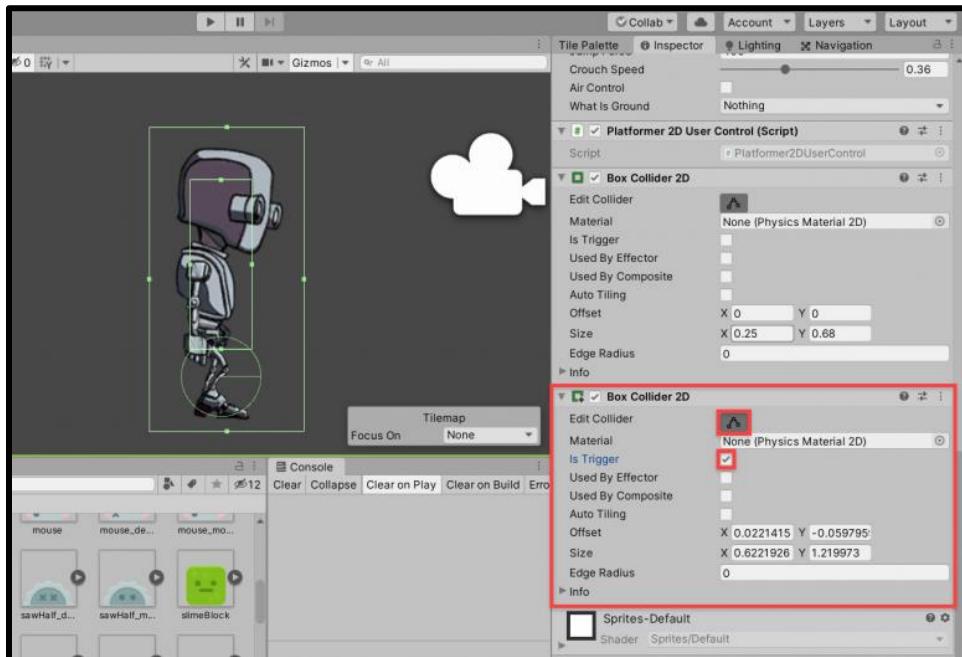
Then move a couple frames forward and change it back to the first image.



Now if you hit play you will see that our enemy is animating! Cool!

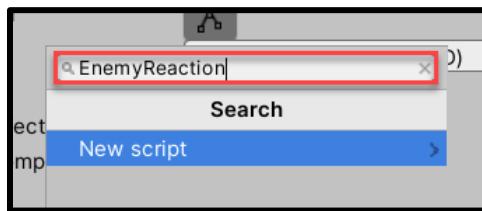
Scripting our enemy

Let's make it so that whenever the character touches the enemy the level restarts. The best way to do this is to actually create a new script and box collider on our robot character.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Set the box collider to Trigger and make sure it liberally covers the character. Name the new script “EnemyReaction”.



Let's create a new folder called “Scripts” to house this component.



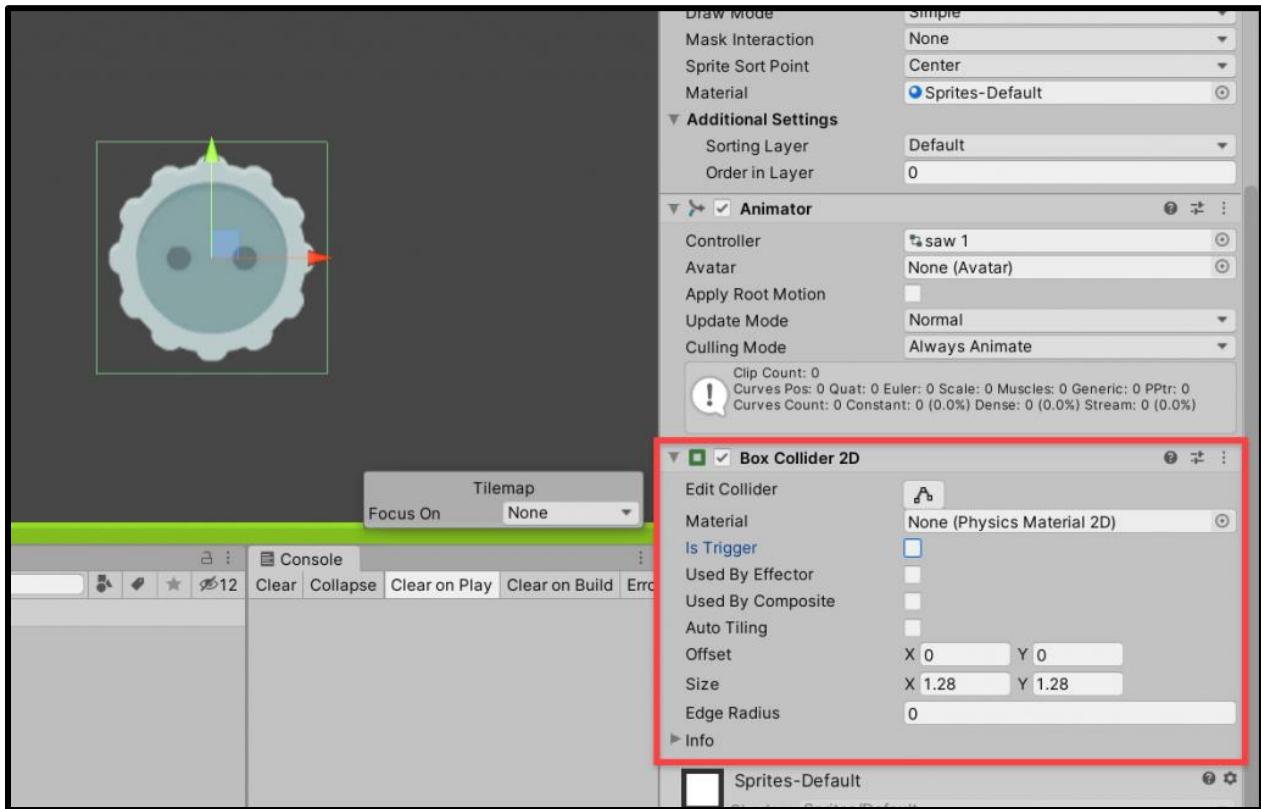
Here is the content of the EnemyReaction script.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.SceneManagement; // This is very important if we want to restart the level
5
6  public class EnemyReaction : MonoBehaviour {
7
8      // Use this for initialization
9      void Start () {
10
11  }
12
13      // Update is called once per frame
14      void Update () {
15
16  }
17      // This function is called every time another collider overlaps the trigger collider
18      void OnTriggerEnter2D (Collider2D other){
19          // Checking if the overlapped collider is an enemy
20          if (other.CompareTag ("Enemy")) {
21              // This scene HAS TO BE IN THE BUILD SETTINGS!!!
22              SceneManager.LoadScene ("scene1");
23          }
24      }
25 }
```

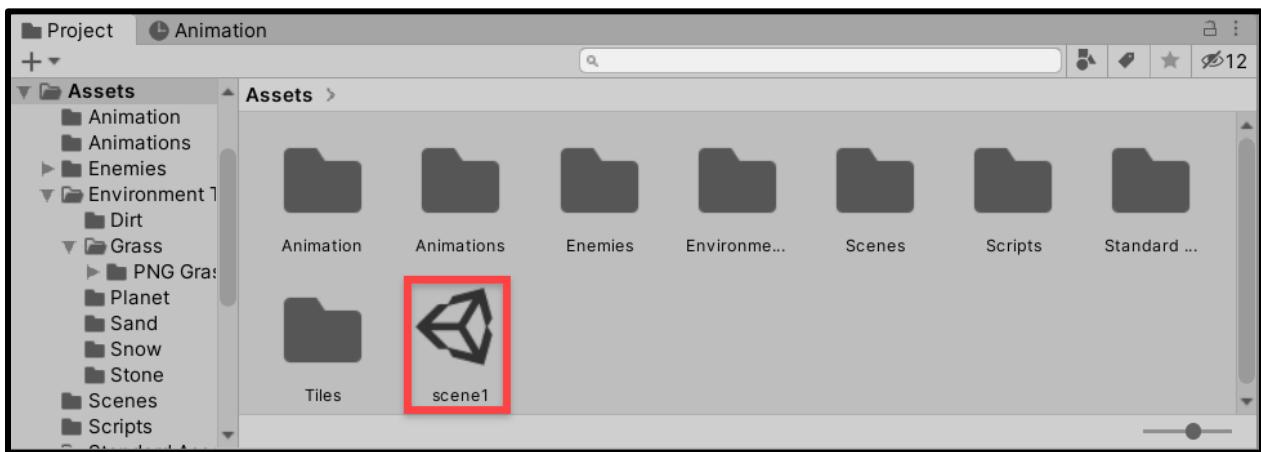
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

In order for this script to work, we need to do a couple of things. First, we need to have a collider on our enemy.



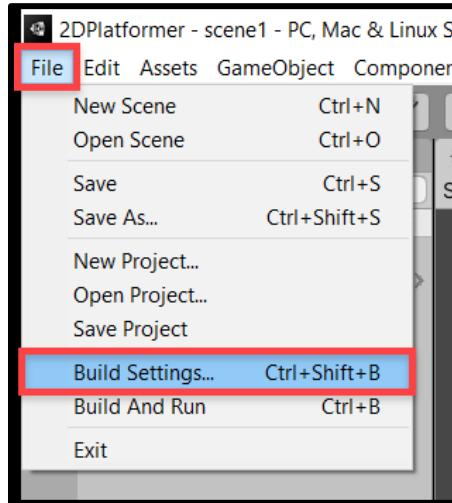
A simple box collider works best for covering all needs. Then, we need to save this scene as "scene1".



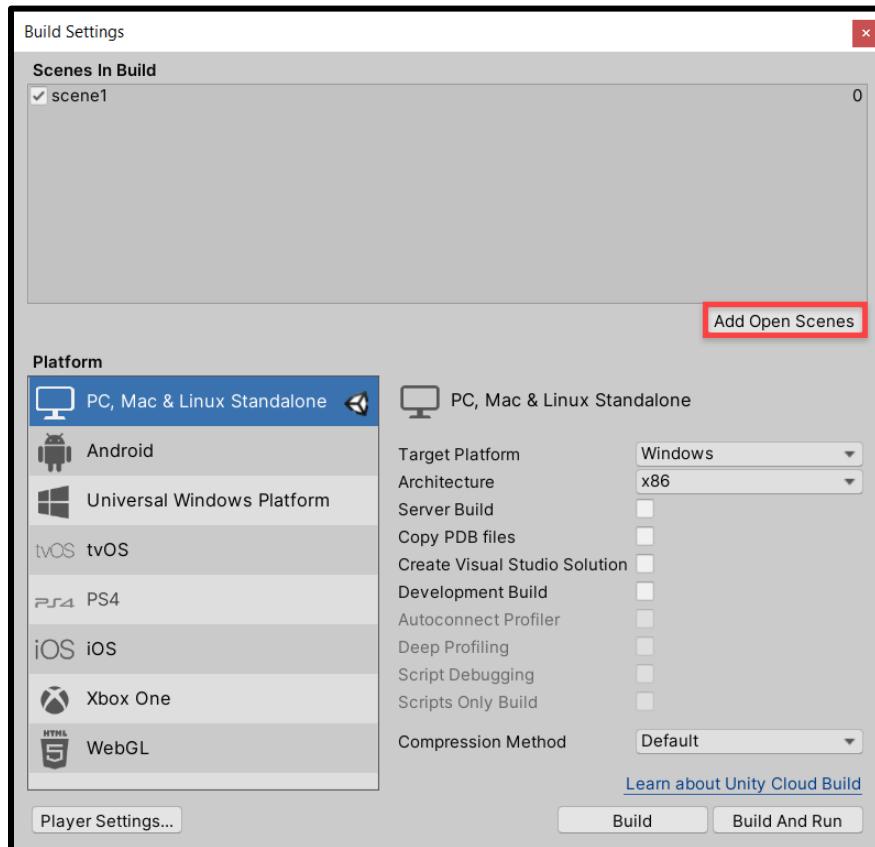
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Just save it in the root folder since we only have one, but if you are planning on having multiple scenes then you should create a new folder. Finally, that scene has to be put in the build settings. To do this, just go to File -> Build Settings



and then click “Add Open Scenes”.



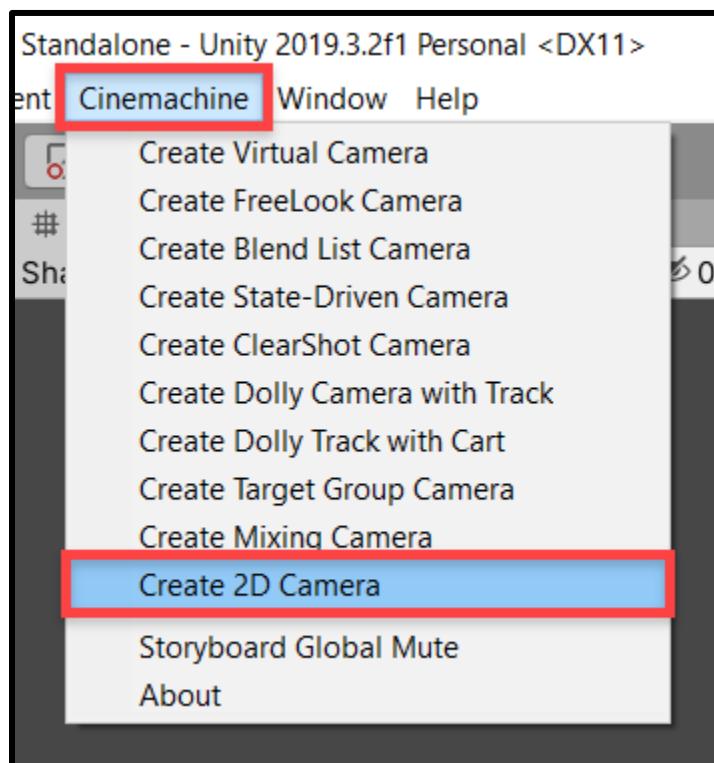
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

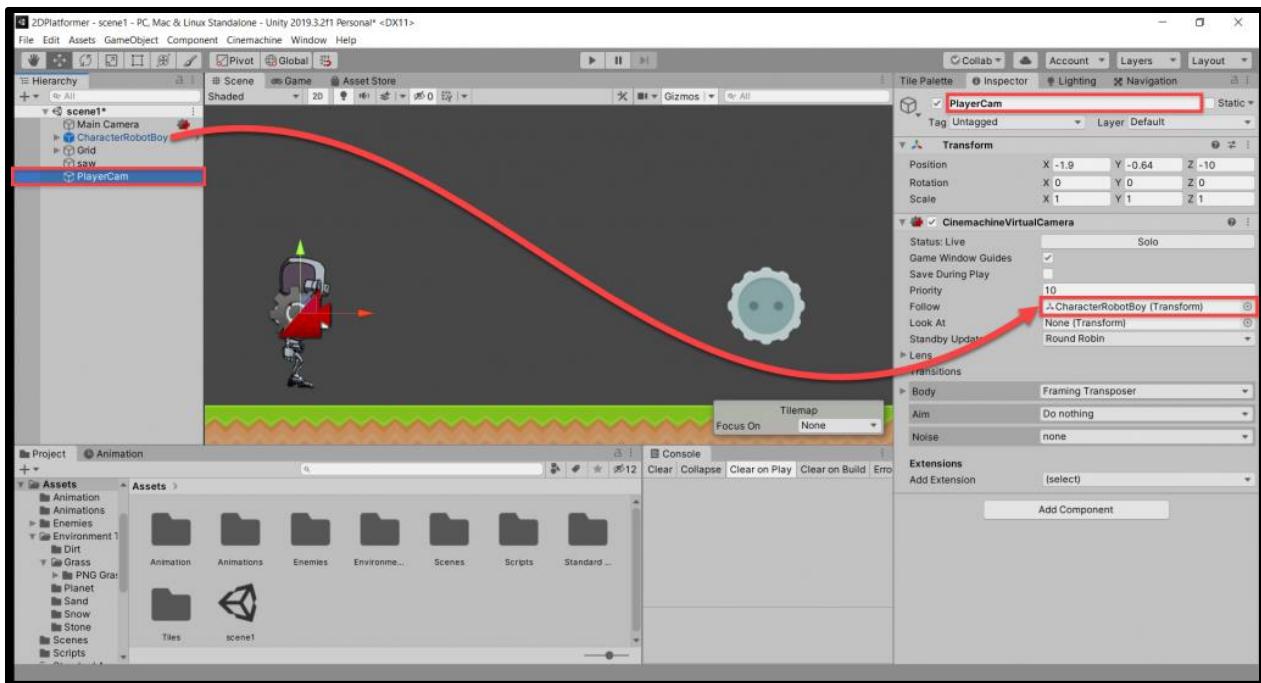
Now everything should work! Hit play and run into the enemy, the level restarts!

Cinemachine and Timeline

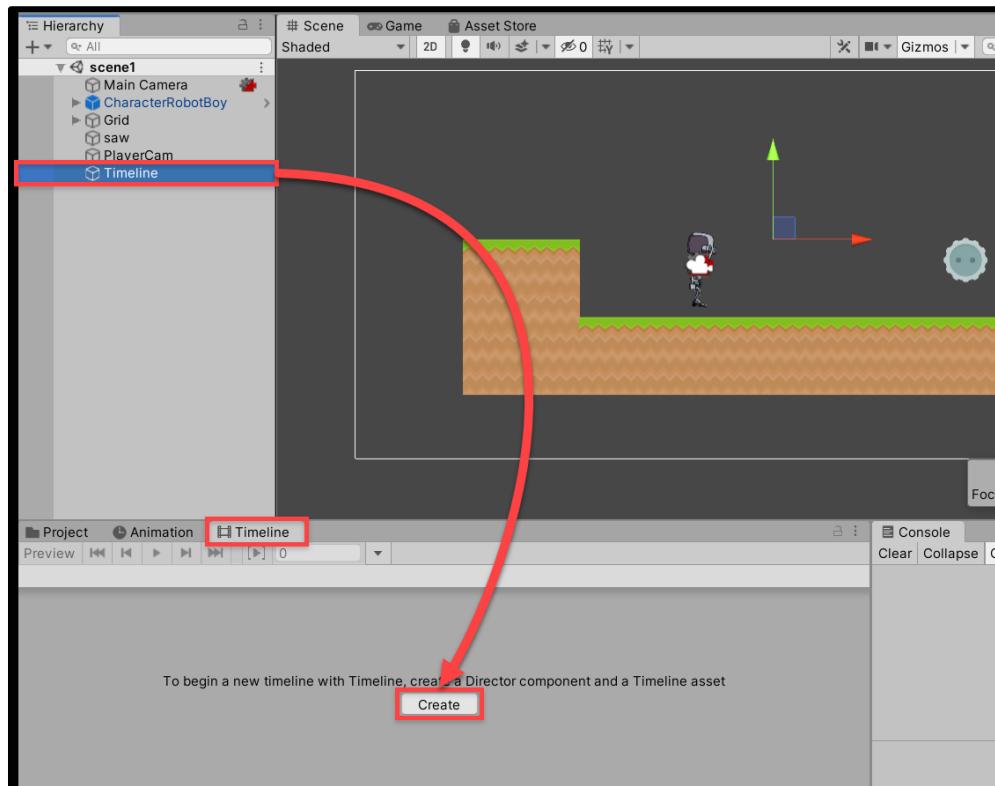
We now come to the last part of this tutorial. In this part, we will be using Cinemachine and the Timeline editor. Let's start with Cinemachine. You'll notice that our camera isn't following the character when he moves around. We can fix this by creating what is known as a Virtual Camera. Navigate to your toolbar and go to Cinemachine -> Create 2D Camera.



Then assign the “follow” field on this camera to be the CharacterRobotBoy. Rename this to “PlayerCam”. Set the Aim to “Do Nothing” and set the Body to “Framing Transposer”.



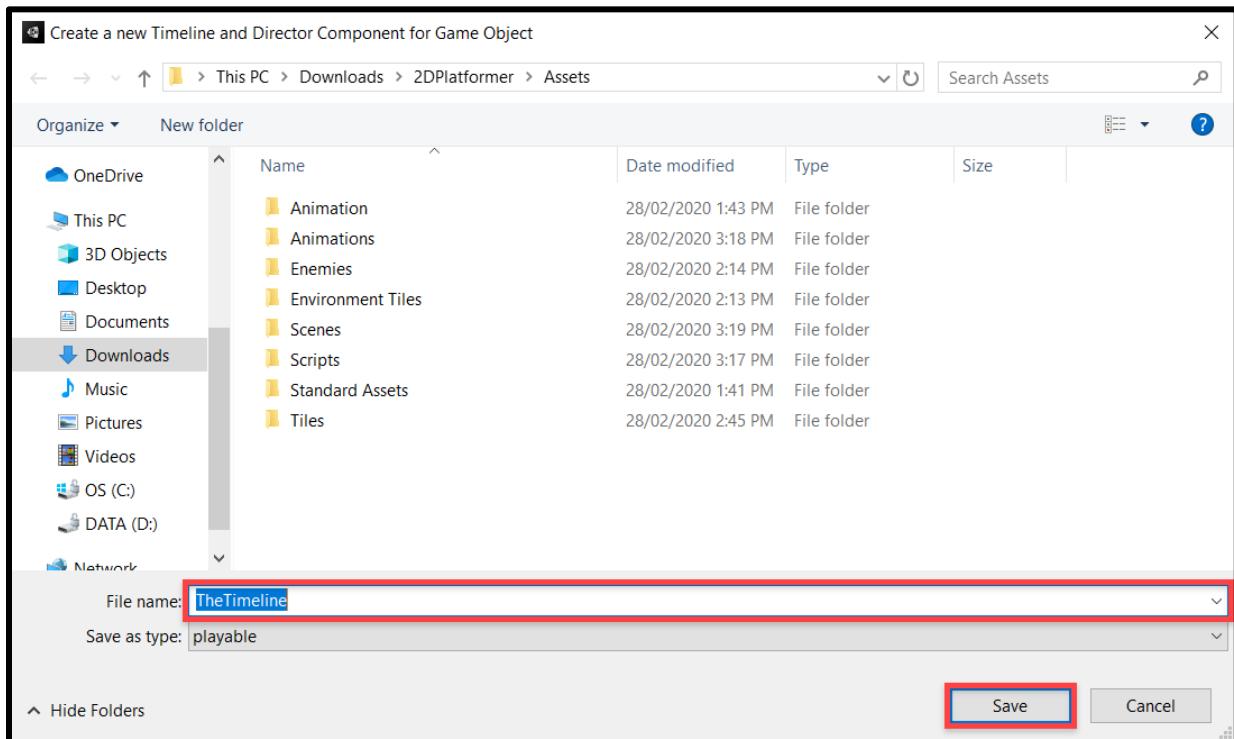
Next, let's have a look at the Timeline Editor. Create a new, empty game object, named "Timeline"...



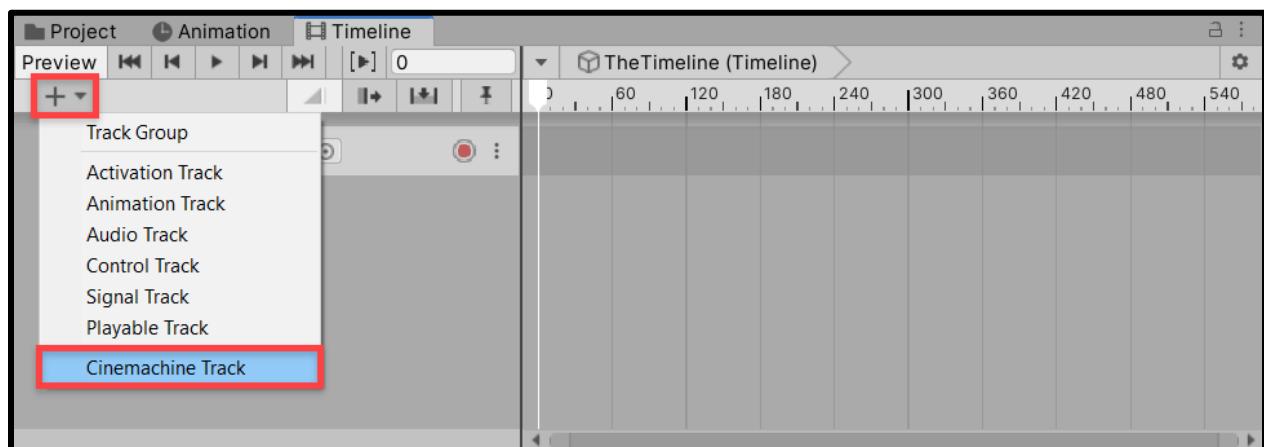
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

...and then click Create in the Timeline Editor. Call it “TheTimeline” and save it in the root folder.

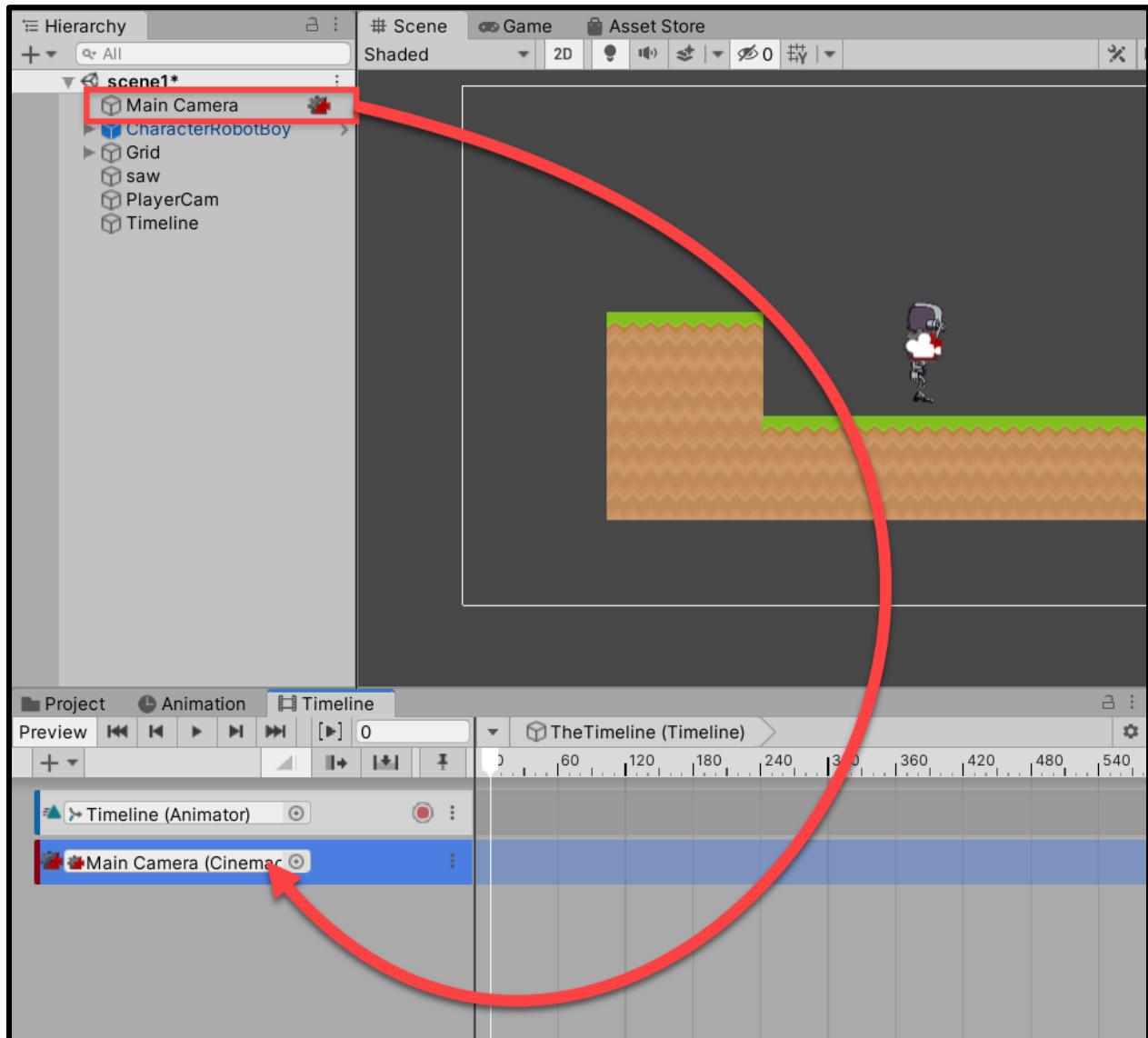


You can have multiple timeline editors in a scene, which just adds to the complexity so we just have one. With this Timeline, we will create a cutscene where the camera views the entire level and then zooms in on the player. This will occur as soon as the scene starts. In order to do this, we need to combine Cinemachine and the Timeline Editor. We can do this through Cinemachine shot clips. In the Timeline, click “Add” and then go to Cinemachine.Timeline -> Cinemachine Track.

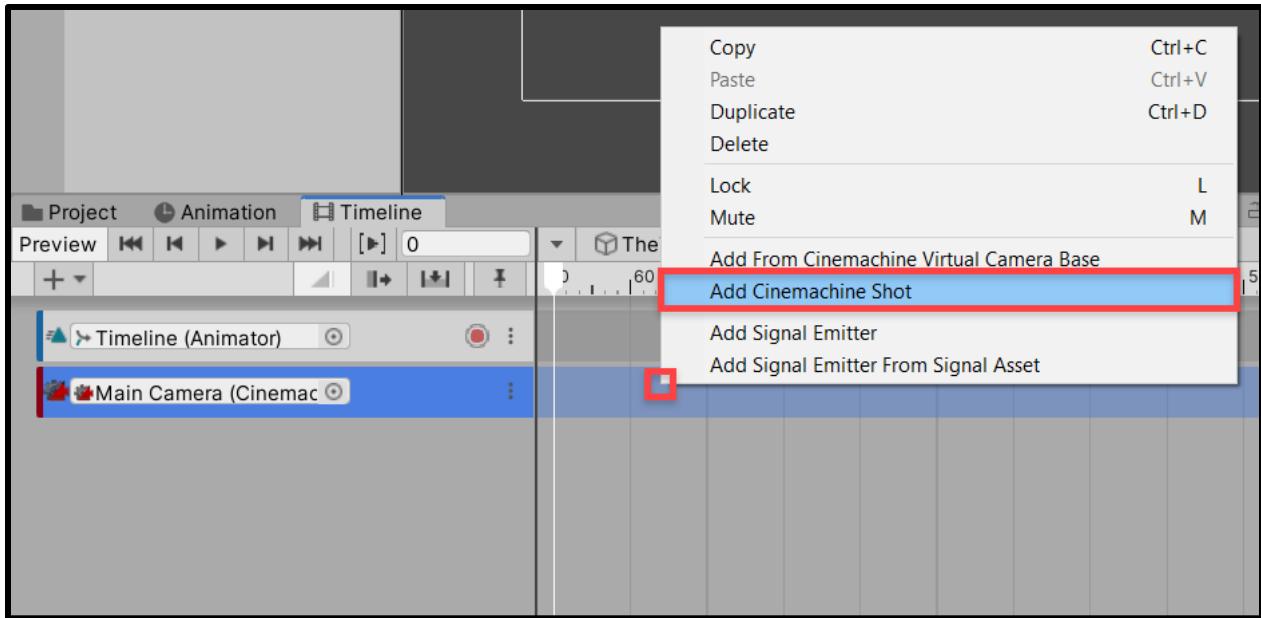


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

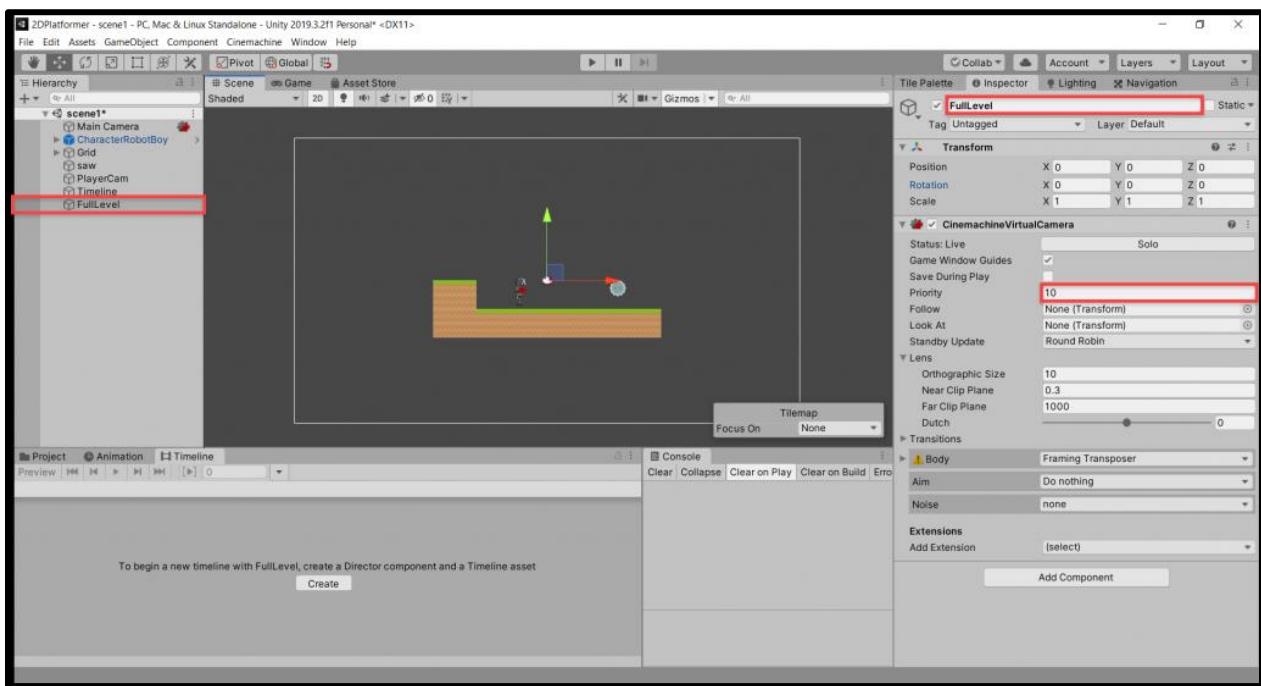
Drag the Main Camera into the field.



Then right-click and go to “Add Cinemachine Shot Clip”.

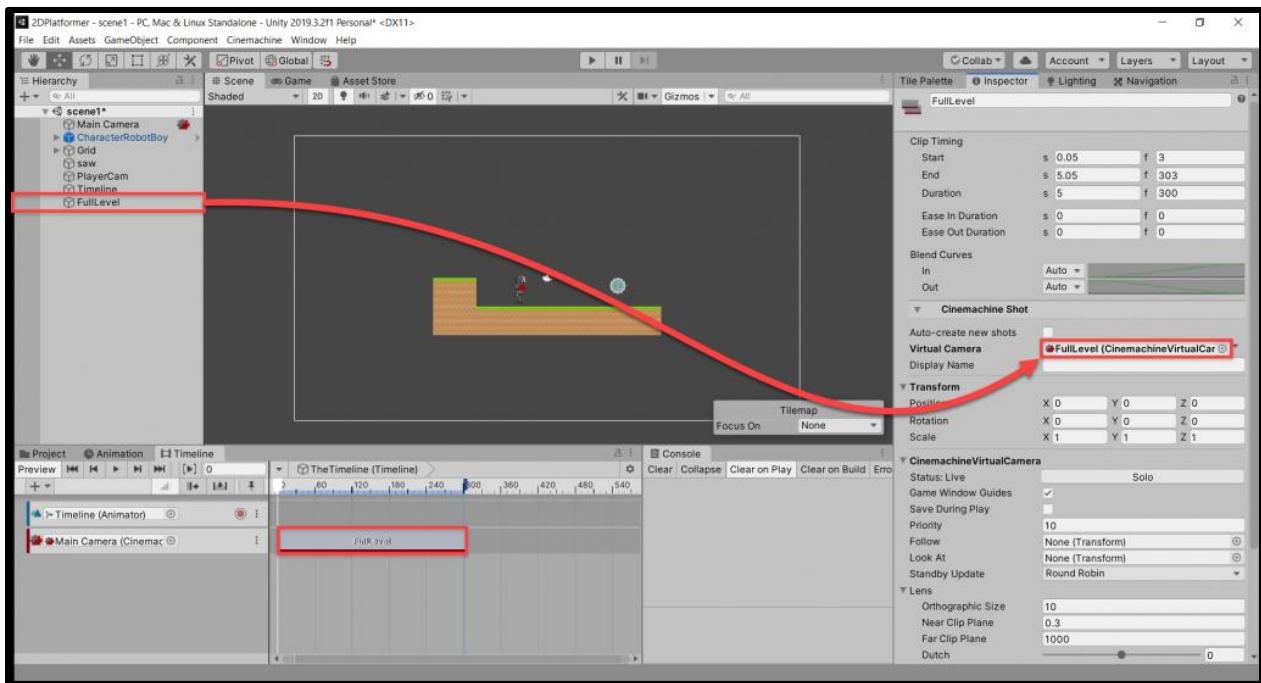


With this track, we can specify how long we want a certain camera to be active. Let's create a new 2D camera and position it so that we can see the entire level. Name this one "FullLevel".

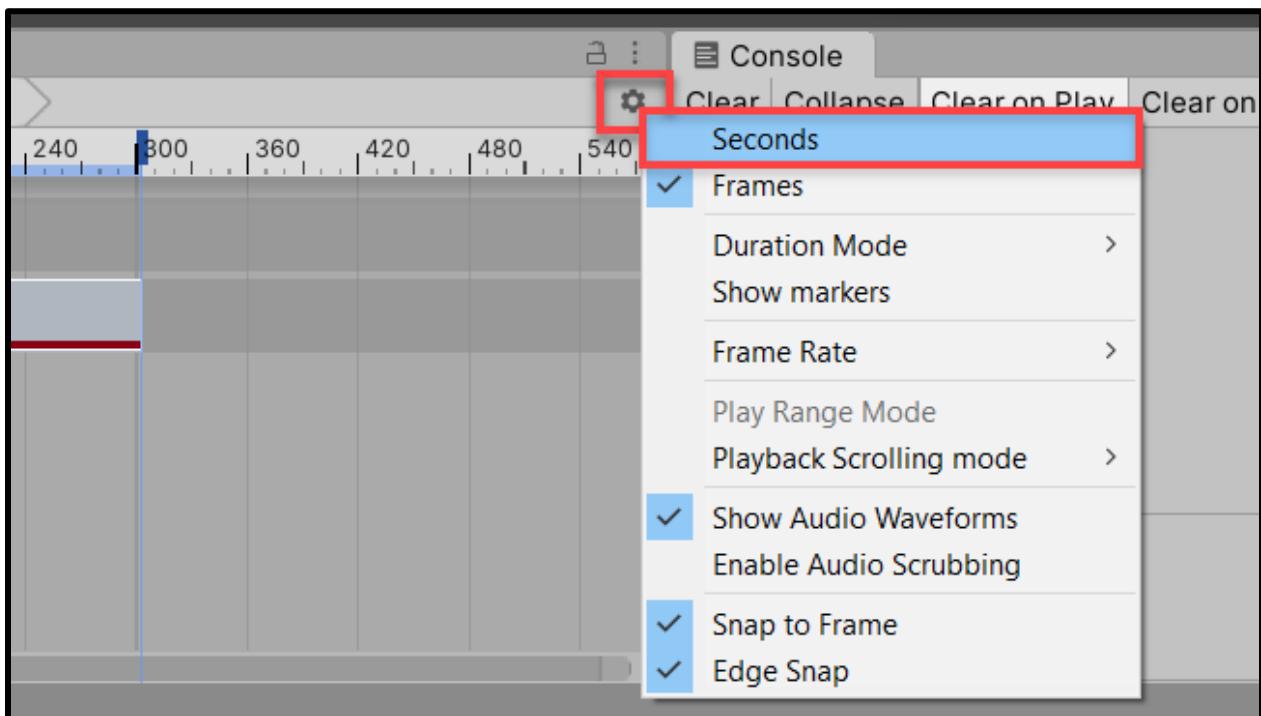


This will be our first camera so select the Cinemachine shot clip and drag this camera into the "Virtual Camera" field.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

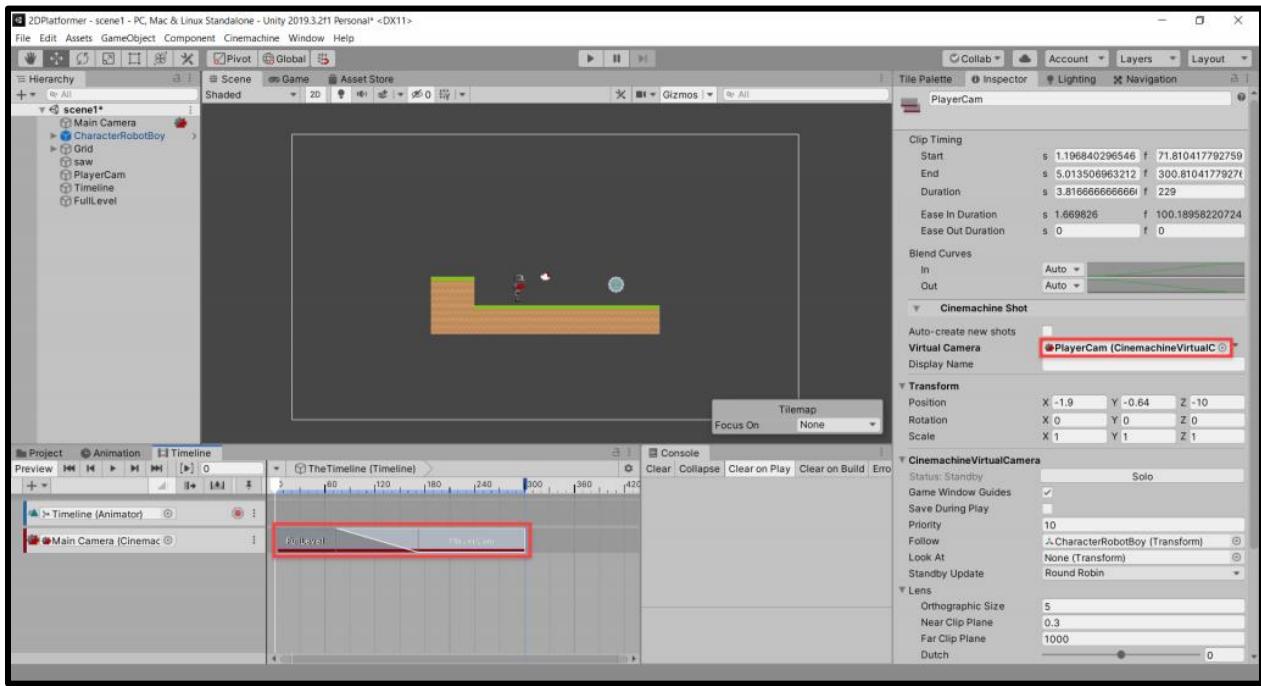


Set how long you want this camera to last. One thing that helps is to set the timescale to be in second and not frames.

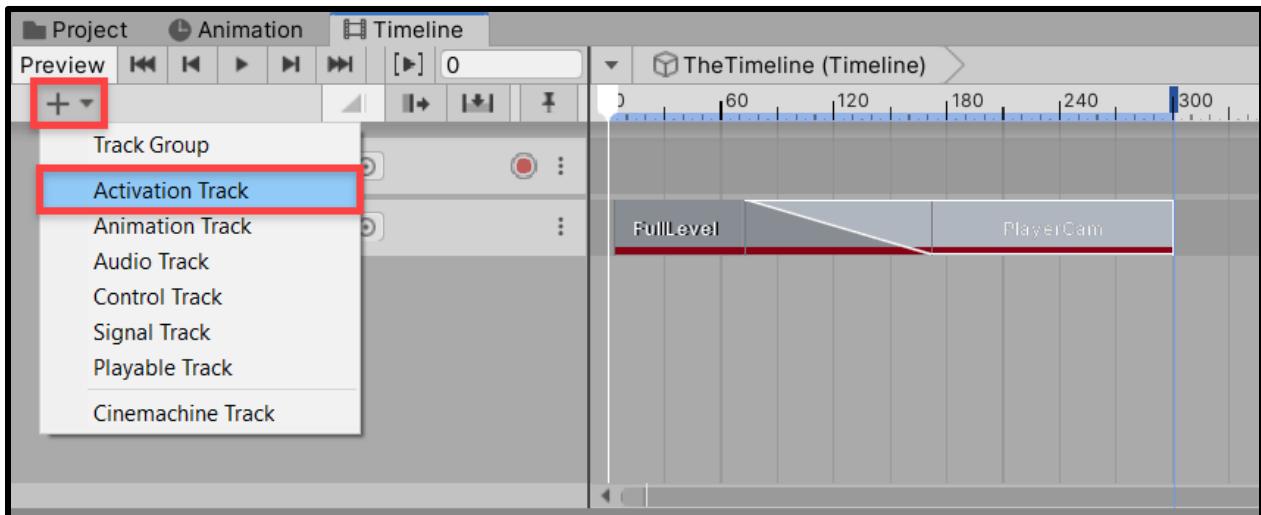


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Next, we need to create another clip for the player camera.



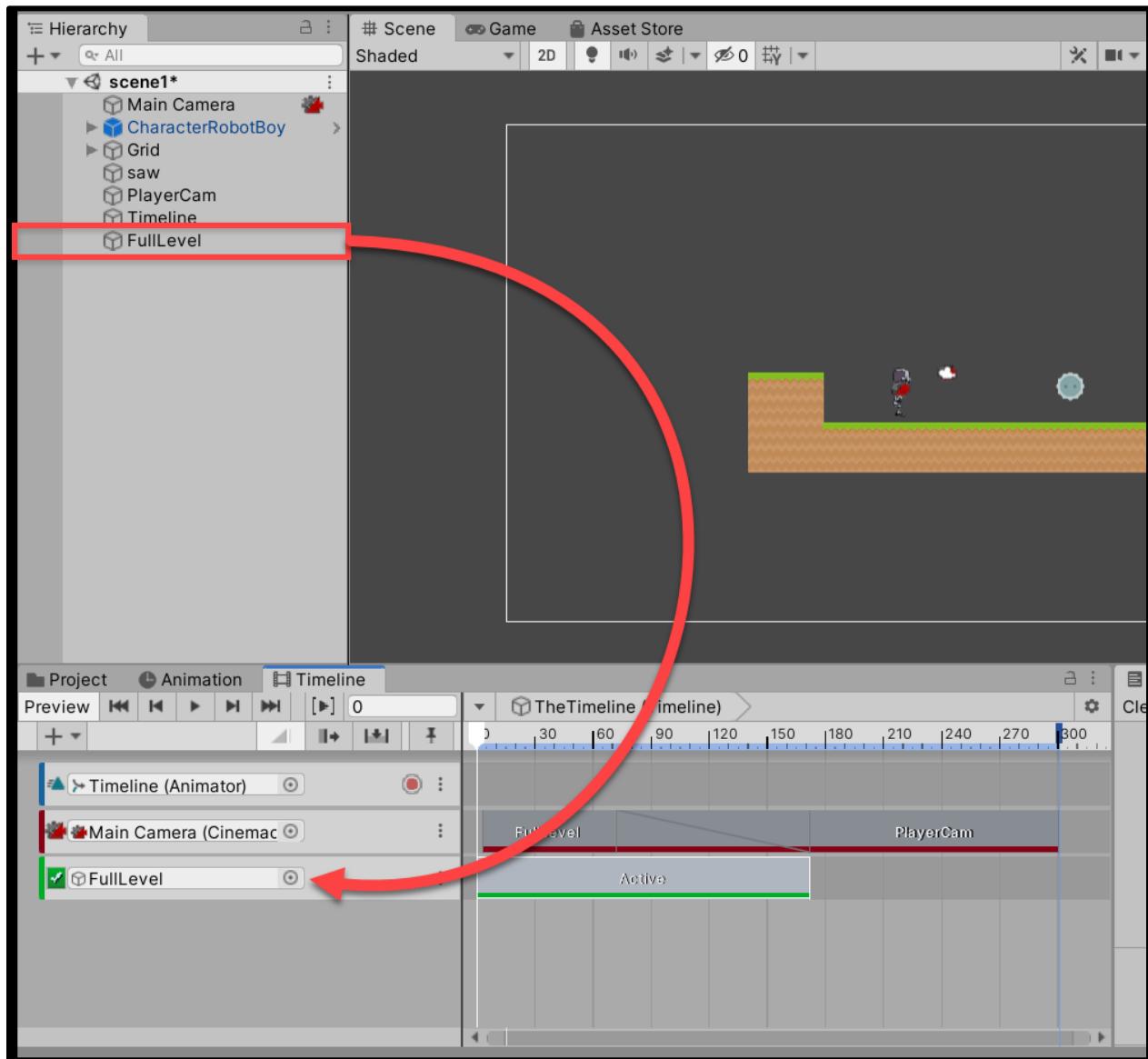
Place this at the end of the other clip. Drag either one of the clips on top of each other in order to smoothly transition. Now if you hit play, we have a pretty neat cutscene! But notice when the last track ends it goes back to the first camera instead of staying on the player. The easiest way to fix this is by deactivating the first camera after we have transitioned into the second. We can do this by using an Activation Track. As its title implies, the specified game object will stay active as long as the track persists. Create an Activation track...



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

...and drag the first camera into its field.



Then set the length to be a little after the transition. Now if you hit play, it all looks good and the player camera persists!

Outro

Congratulations on getting to the end of this tutorial. We now have a 2D platformer in our hands, a perfect addition for any [portfolio](#)!

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

However, as I said before, this is not meant to be exhaustive. In order to find out more about these tools, you can either read the tutorials linked above, or you can explore them on your own! We also recommend you check out our [web class](#) on camera following and our [mini-course](#) on 2D platformers as well, as they're sure to help you cement the knowledge you've learned here!

In either case:

Keep making great games!

Understanding 2D Animations in Unity3D

By Jesse Glover

Welcome to another exciting tutorial on Unity3D. Today's lesson will focus on animations within Unity3D. Since animations occur not only in 2D games but 3D games as well, I've decided to break this up into two separate tutorials. This tutorial will focus on 2D animations, and a subsequent tutorial in the future will discuss 3D animation techniques.

As always, I have broken this tutorial down into two distinct segments. The first segment explaining the basic premise of how animations work within Unity3D, and the second segment will be an actual implementation of animations within a scene that you can control.

Requirements

A basic understanding of the C# language

Basic knowledge of the Unity3D editor

Segment 1: Animation Basics

Unity3D allows for multiple animations to play at the same time in multiple layers or a single animation to play on a base layer. This makes for Unity3D to be extremely flexible in how you can create a game. For example, let's say you wanted to build a game where you have a ship that tilts left and right according to movement trigger and auto fires constantly. Unity3D's animation allows you to do this with relative ease.

Let's explore how to start working with animations in Unity.

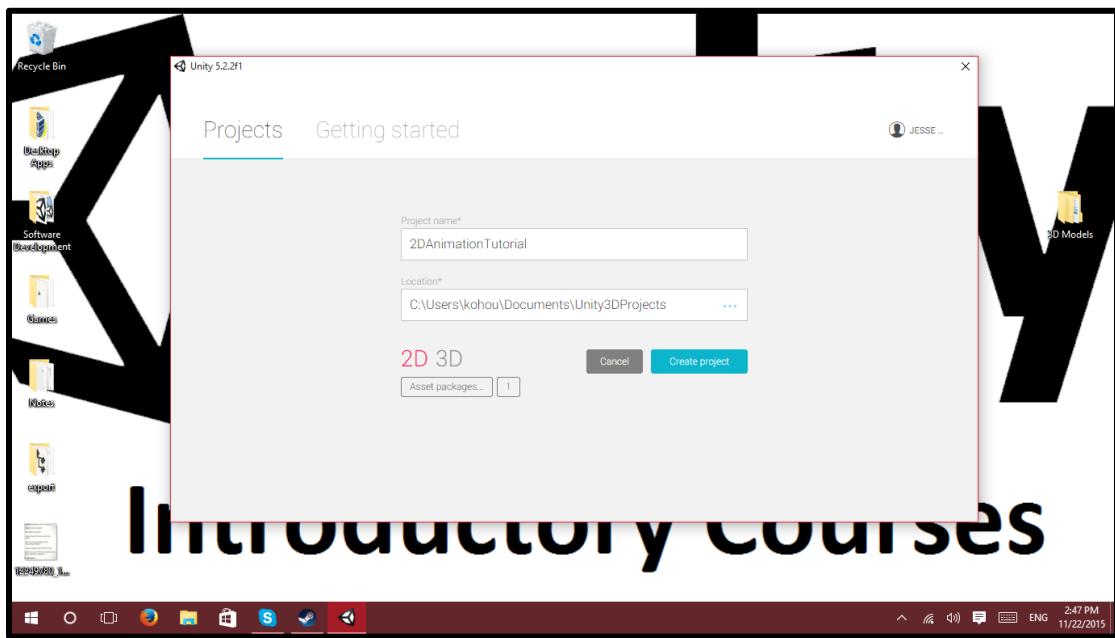
Section 1: Setting up the Project

The art used in this tutorial was from <http://opengameart.org> under the public domain license.

- <http://opengameart.org/content/animated-robot-set-png> – Without SpriteSheet artwork
- <http://opengameart.org/content/ghost-animated> – With SpriteSheet artwork

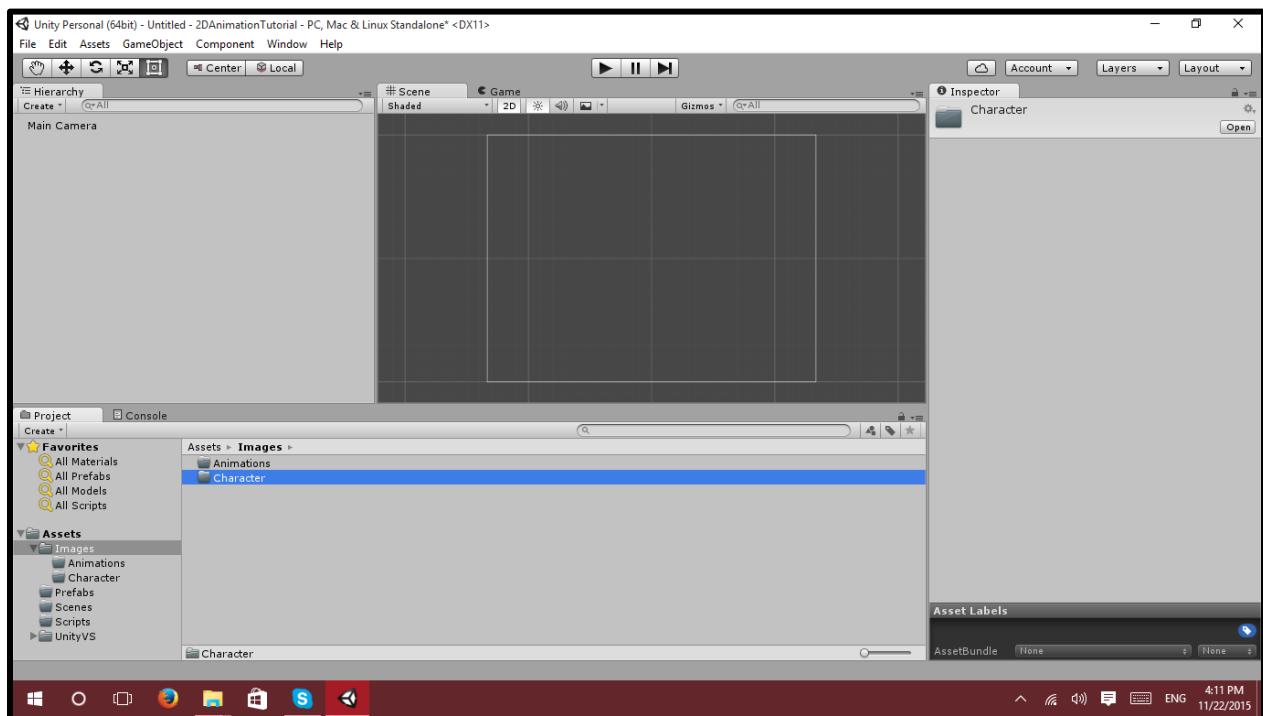
Zip file for the entire project is located [here](#).

To begin, let's start by opening Unity3D and creating a new project. We will name the project "2D Animations Tutorial", make sure the 2D option is selected, and we want the Visual Studio 2015/ 2013 tools asset imported.



In the assets folder, let's make some folders we know we will need from the start. Images, Prefabs, Scenes, and Scripts.

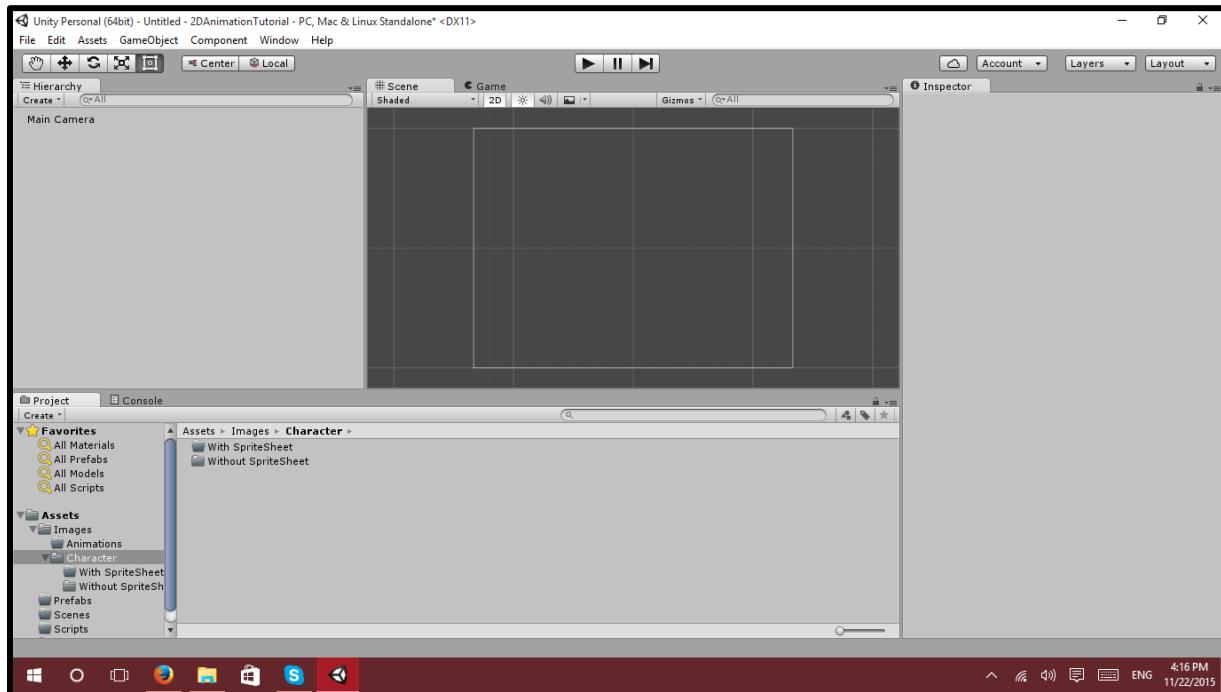
Inside the Images folder, create 2 more folders. Animations and Character.



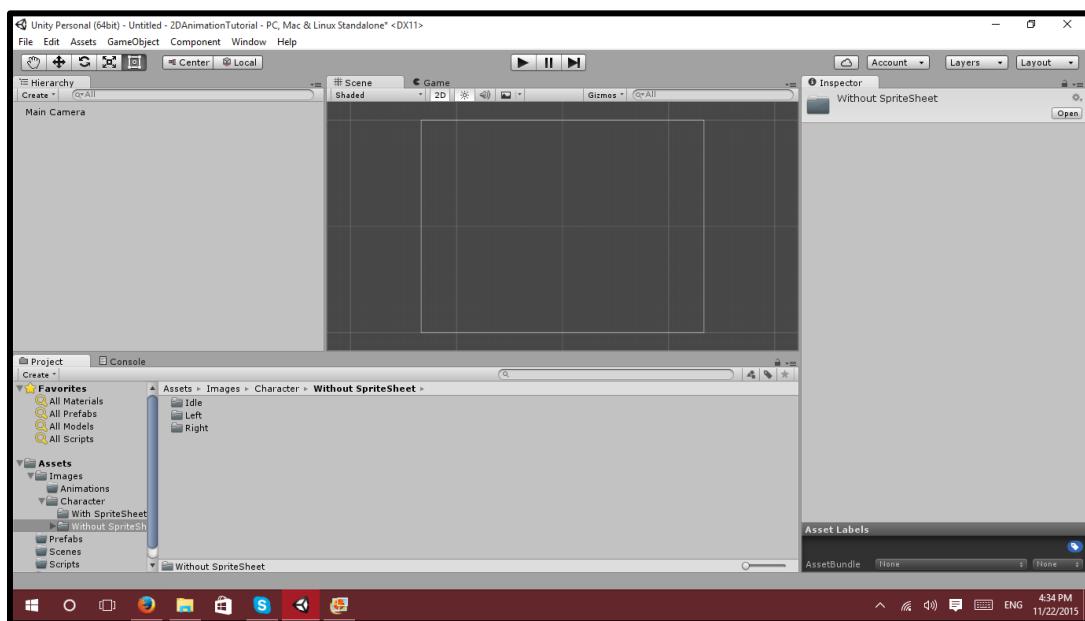
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Just a few more things to add and we will be ready to begin. Inside the Character folder, create 2 more folders. With SpriteSheet and Without SpriteSheet. We will need both for later sections.

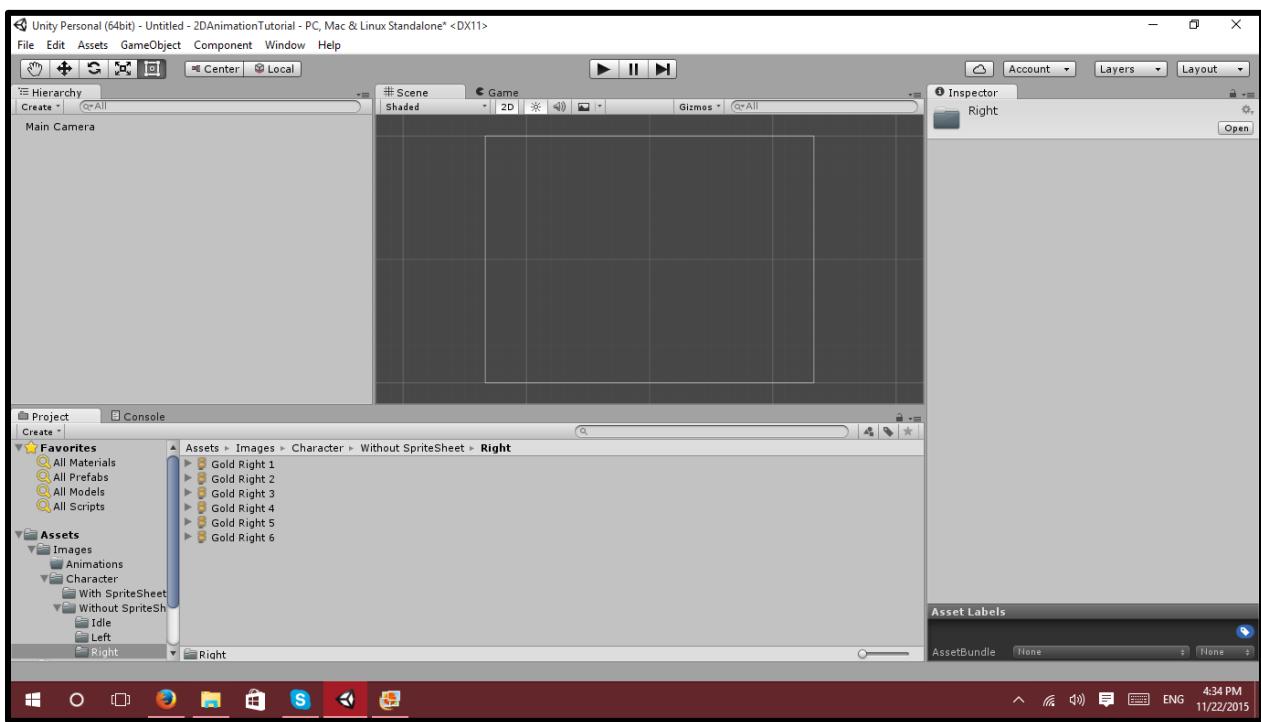
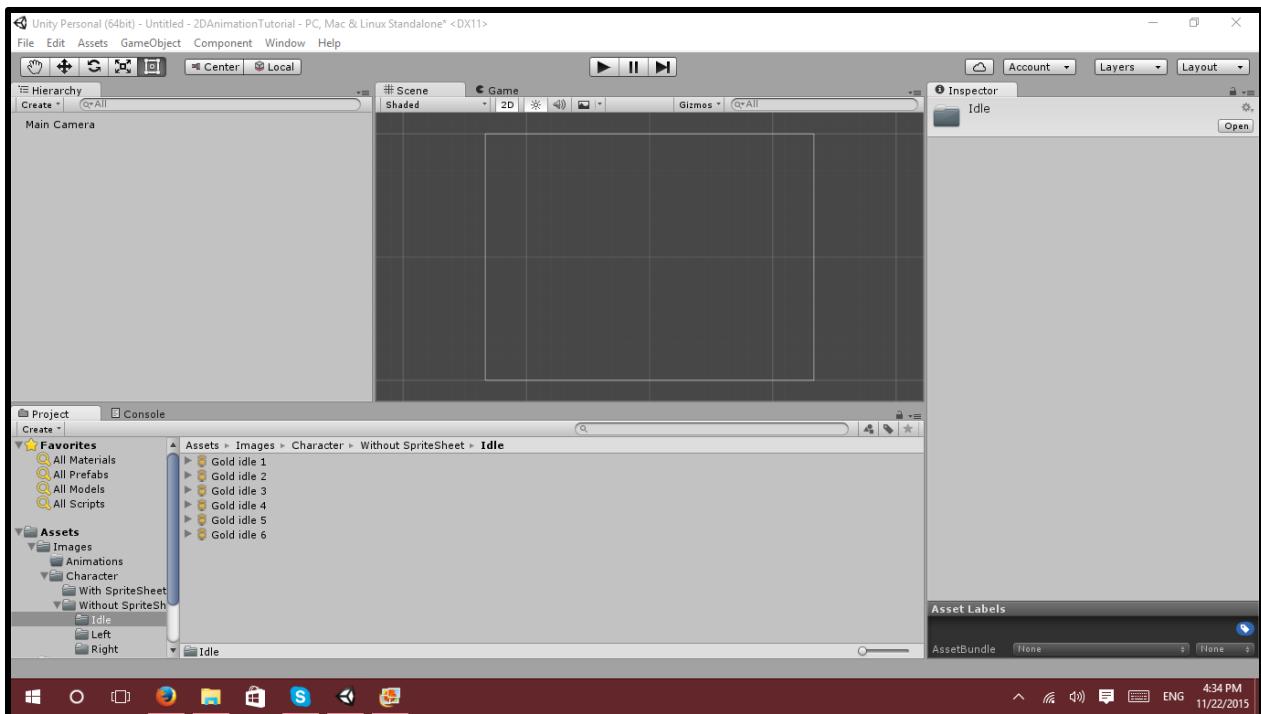


Inside the Without SpriteSheet folder, add the robot images. (I dragged the three folders inside of it to keep things organized).



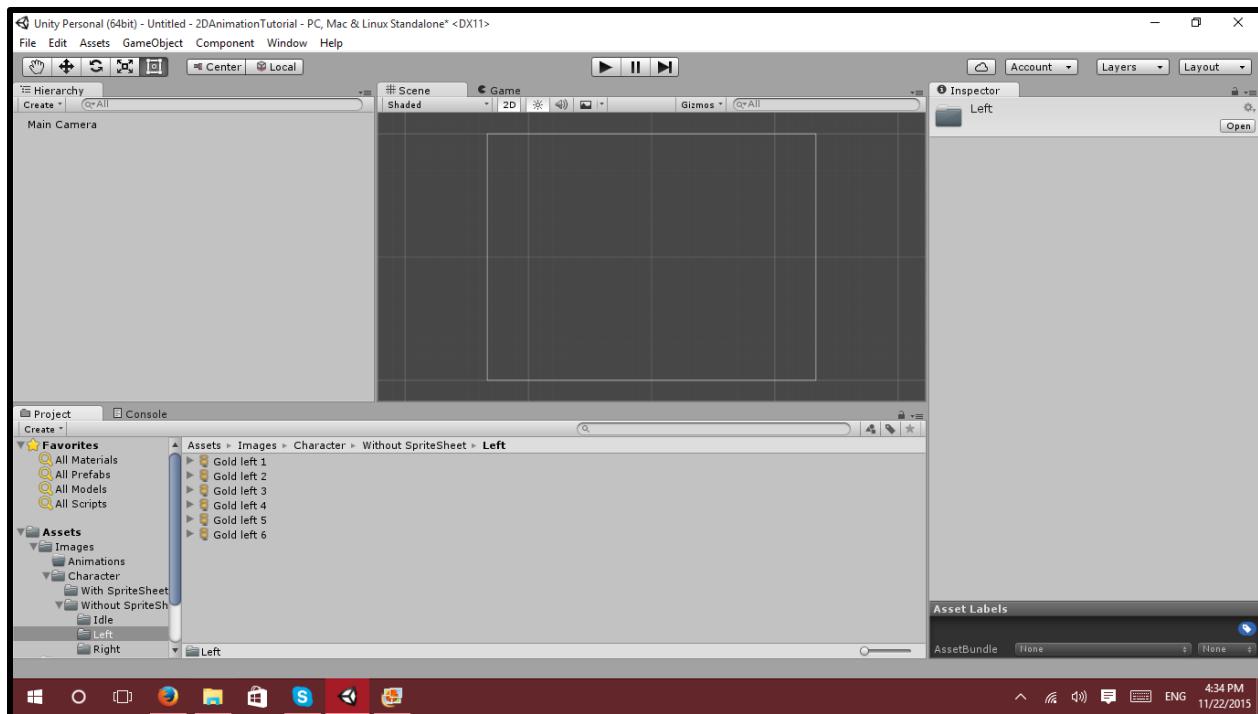
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

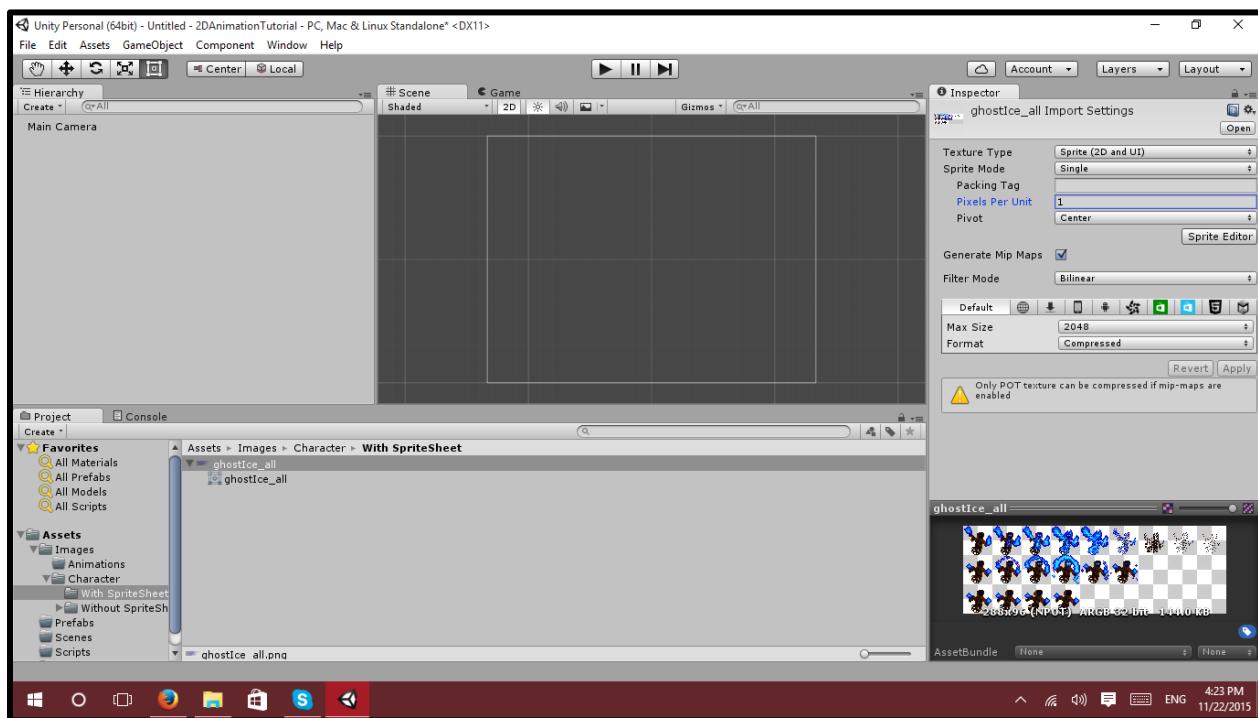


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved



Now, let's select the With Spritesheet folder and add the Ghost spritesheet to it.

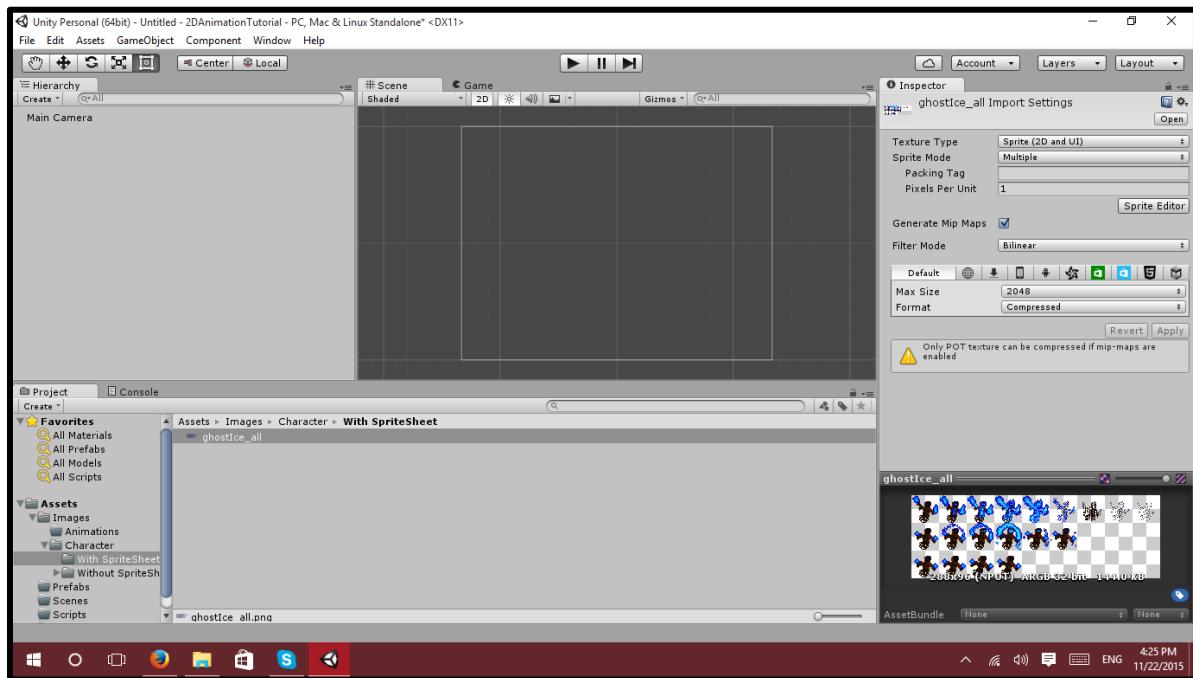


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

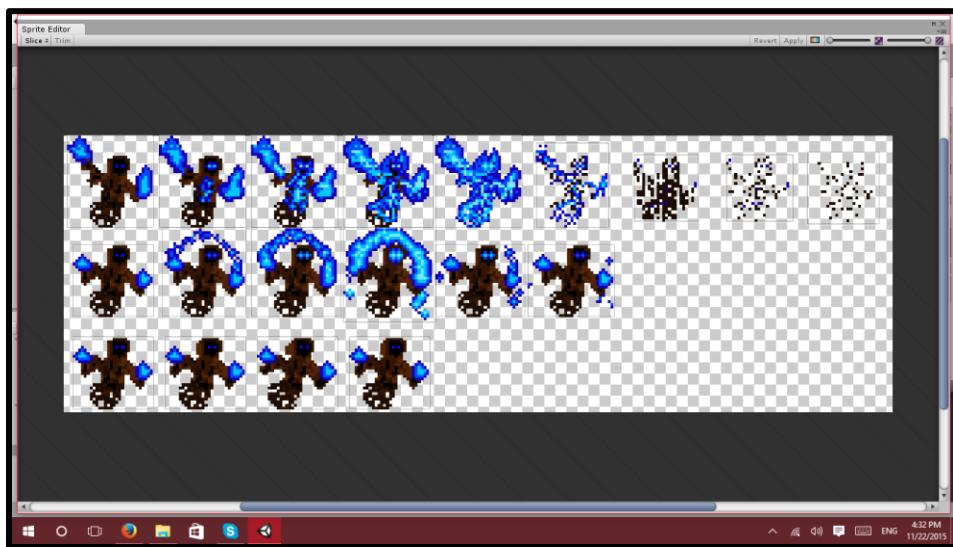
© Zenva Pty Ltd 2021. All rights reserved

We want the Texture Type to be Sprite(2D and UI), SpriteMode to be Multiple, Pixels Per Unit to be 1 instead of 100, Generate Mip Maps to be checked, Filter Mode to be Bilinear, Max Size 2048, and Format to be compressed.

After these have been set, click on Apply.



Now, click on the Sprite Editor option underneath Pixels per unit.



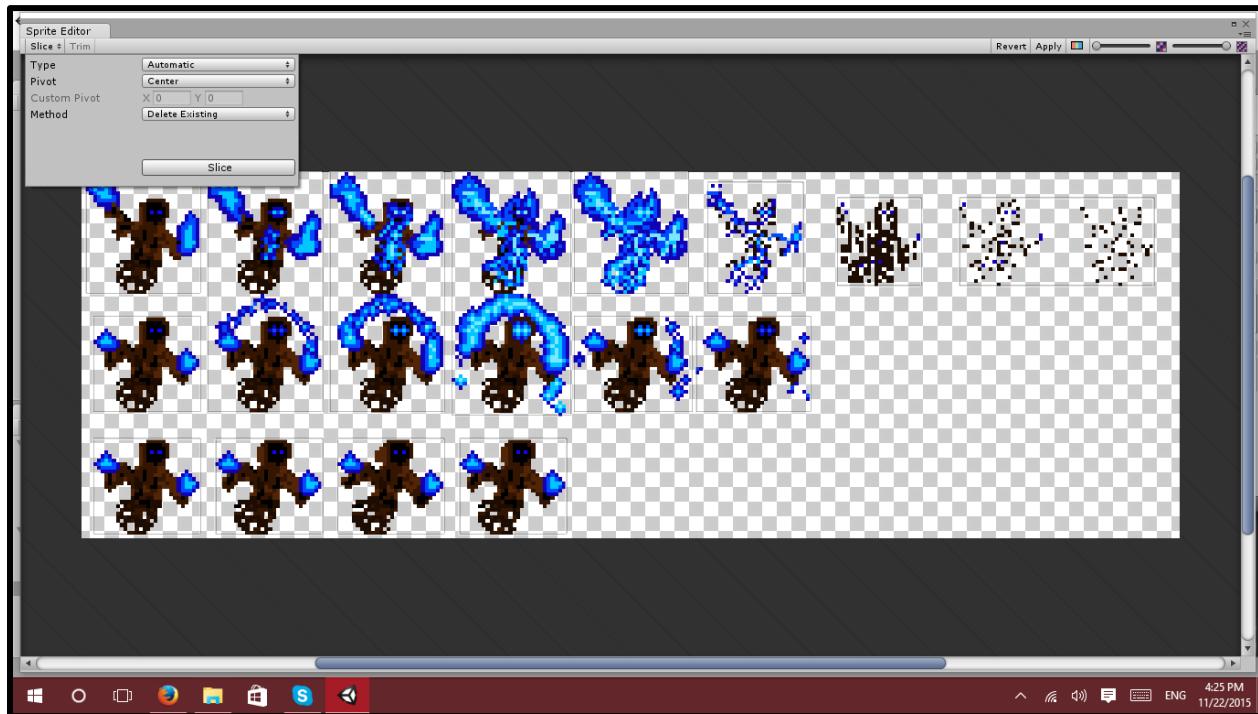
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

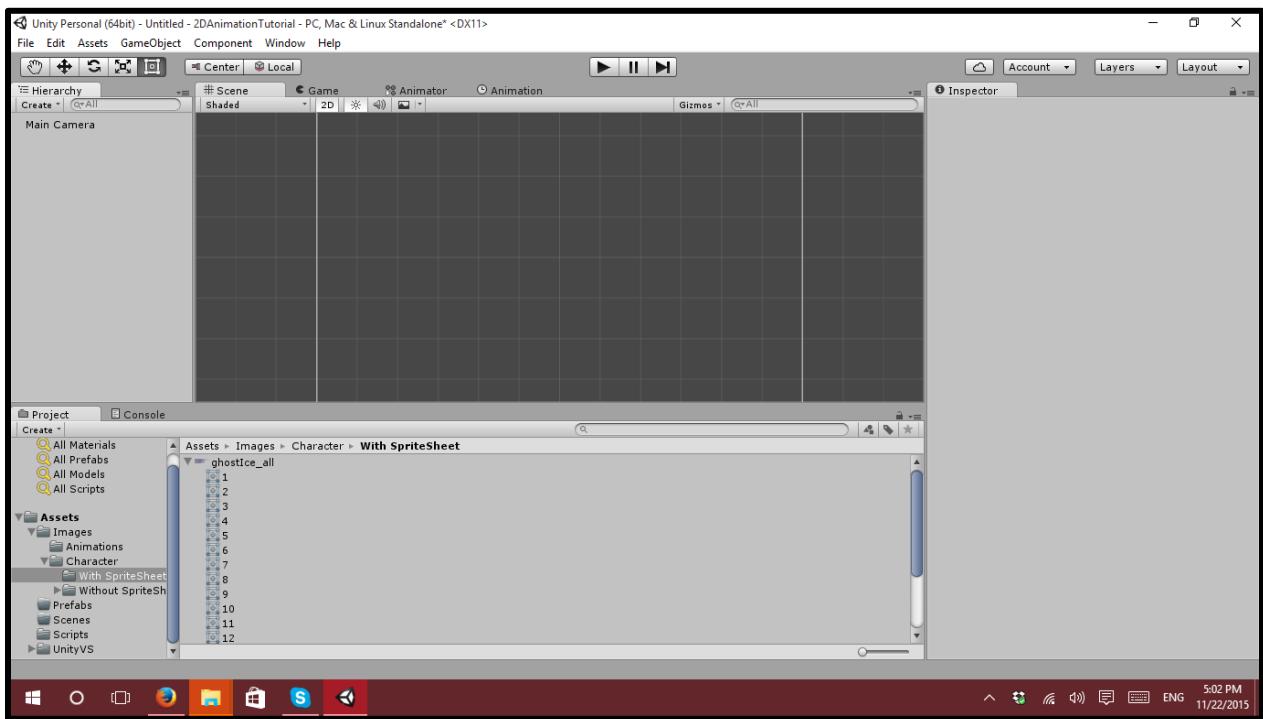
On the top of the Sprite Editor Window, select slice. Type should be Automatic, Pivot center, method is delete existing. Click on the button called Slice.

(Note: If for any reason the slice method did not slice the images into their own sprite block, you can manually do this. Simply select somewhere within the editor box and drag, it will create a box that you can modify in size.)

After you are done, click on apply to apply the changes made.

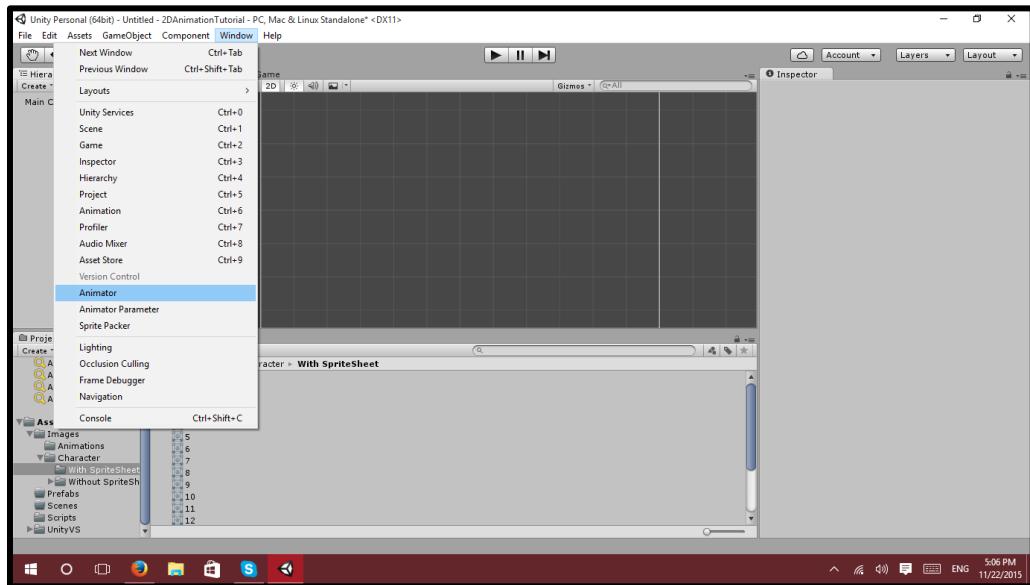


Your Unity3D editor should now look like the image below.



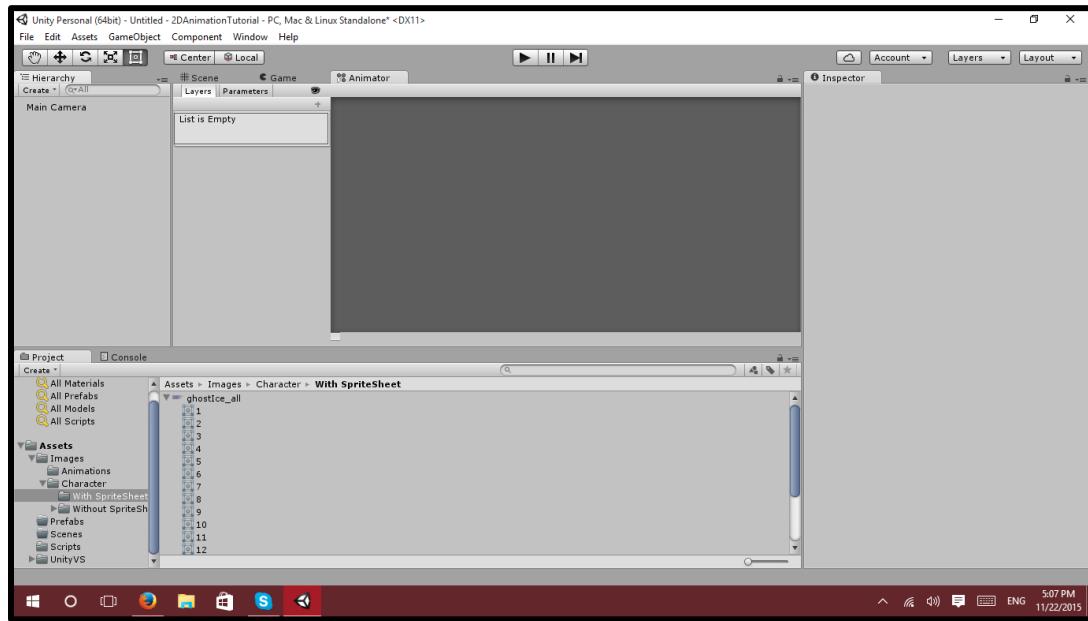
Before we move on to the next section, I want to point out 2 very useful window components for animations within the main Unity3D editor. Animator and Animations.

Click on the Window button and select Animator. You will see why it is important in the next section.

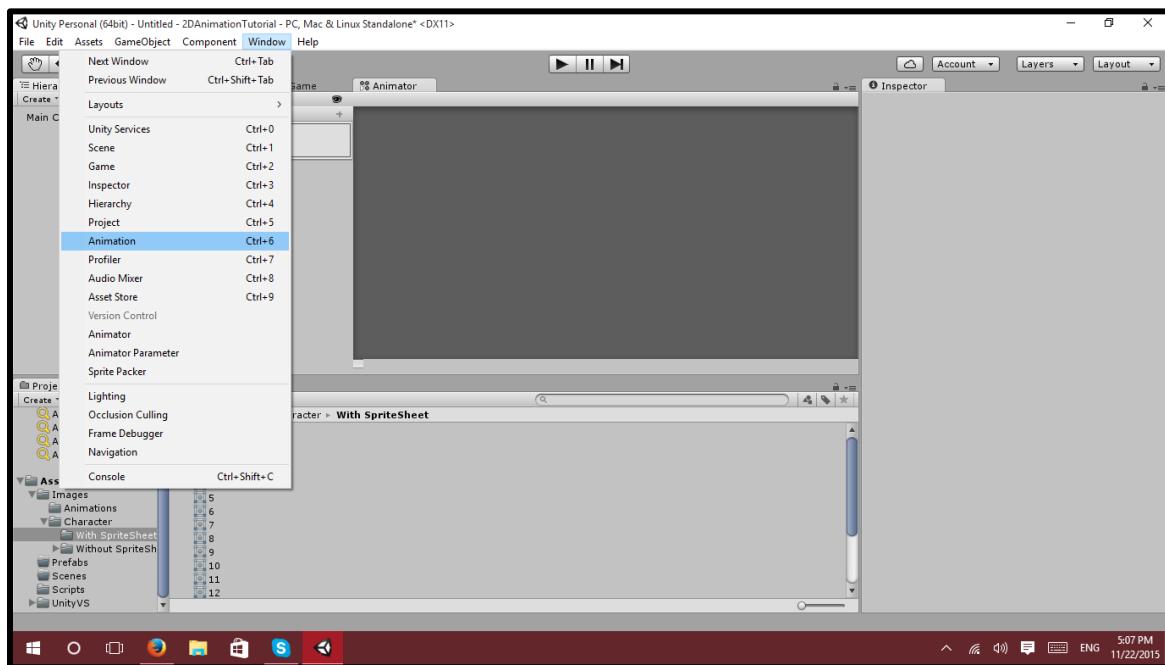


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

It will appear in its own window, click and drag on the tab above with the name Animator in it. I put it next to my game and Scene tab.



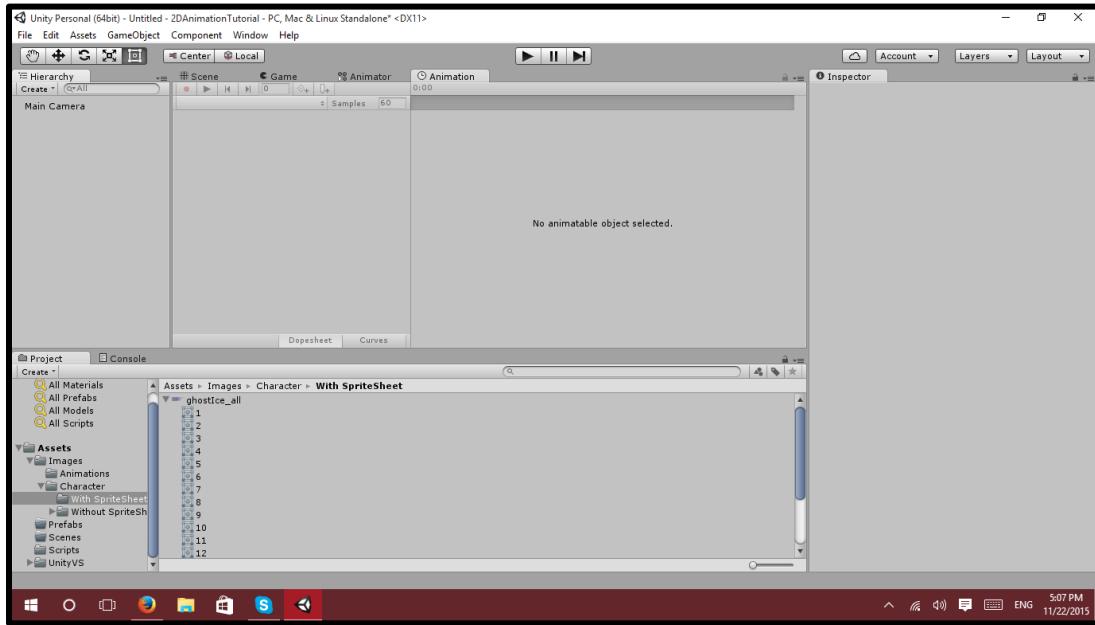
Now, we can do the same thing with the Animation editor view. Click on Window and select the Animation option. Alternatively, you can select Ctrl + 6 / Cmd + 6 (if using Mac OS).



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

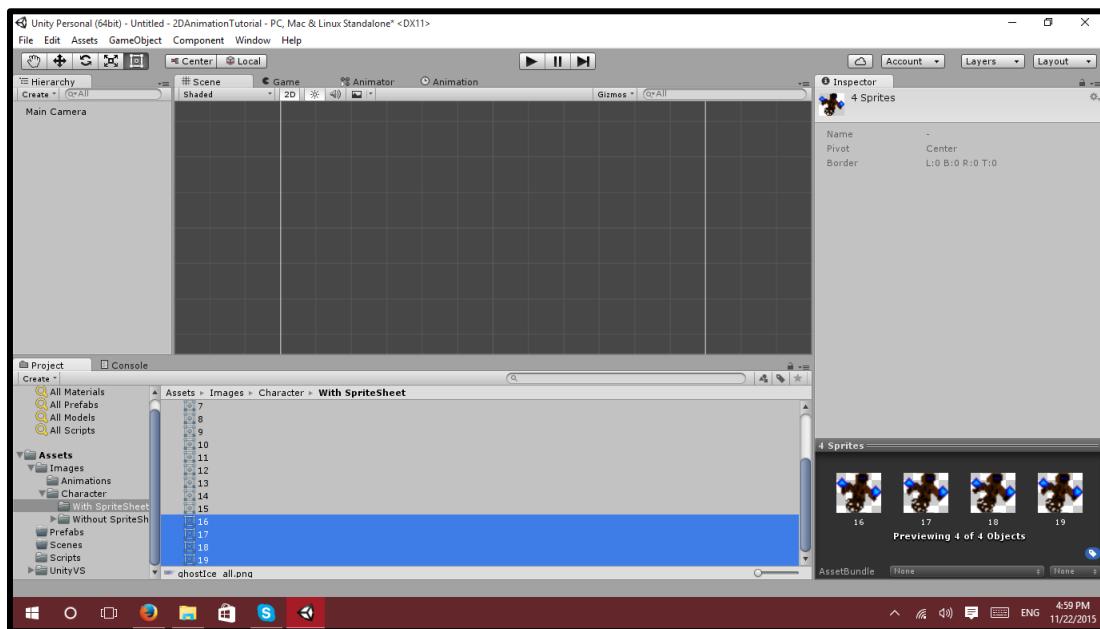
© Zenva Pty Ltd 2021. All rights reserved

Again, click and drag on the tab. I put it next to the animator.



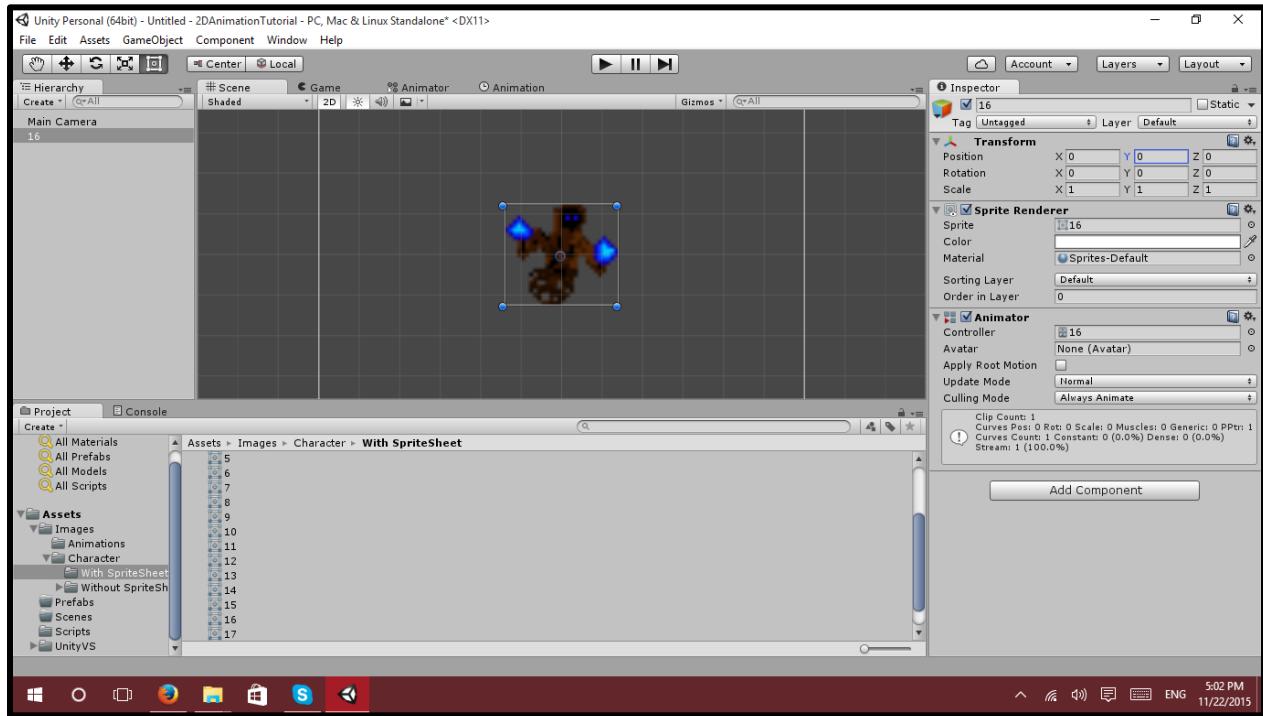
Section 2: Let's Make an Animation

Now we can begin with actually making an animated object within Unity3D. So, let's go back to our With SpriteSheet Folder and minus down the spritesheet image. We want to select images 16 through 19.

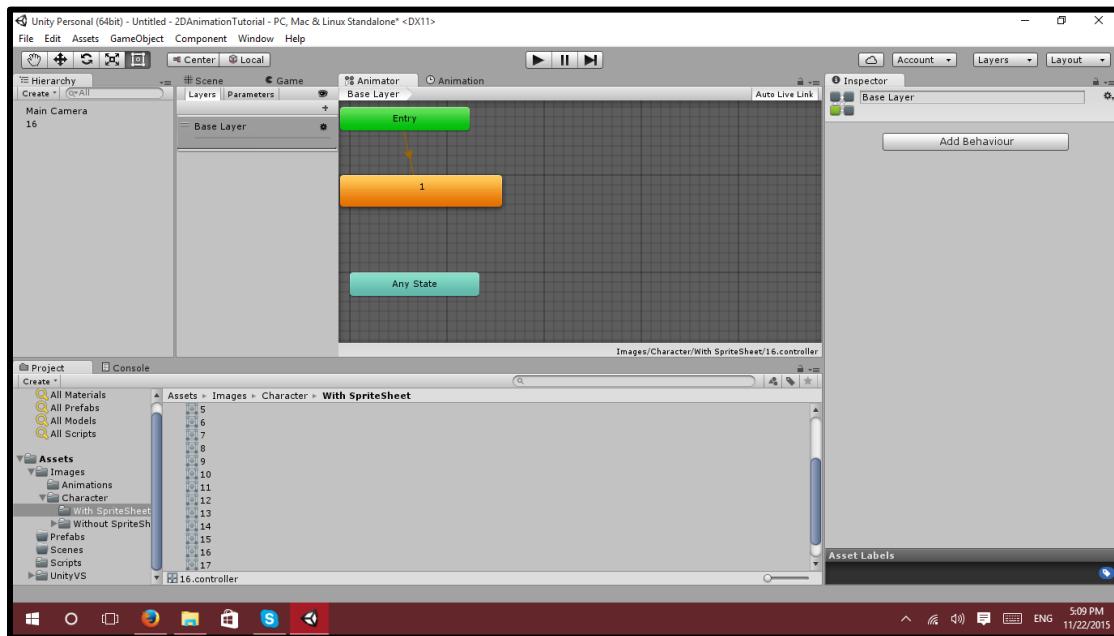


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

And we drag it on the scene view.



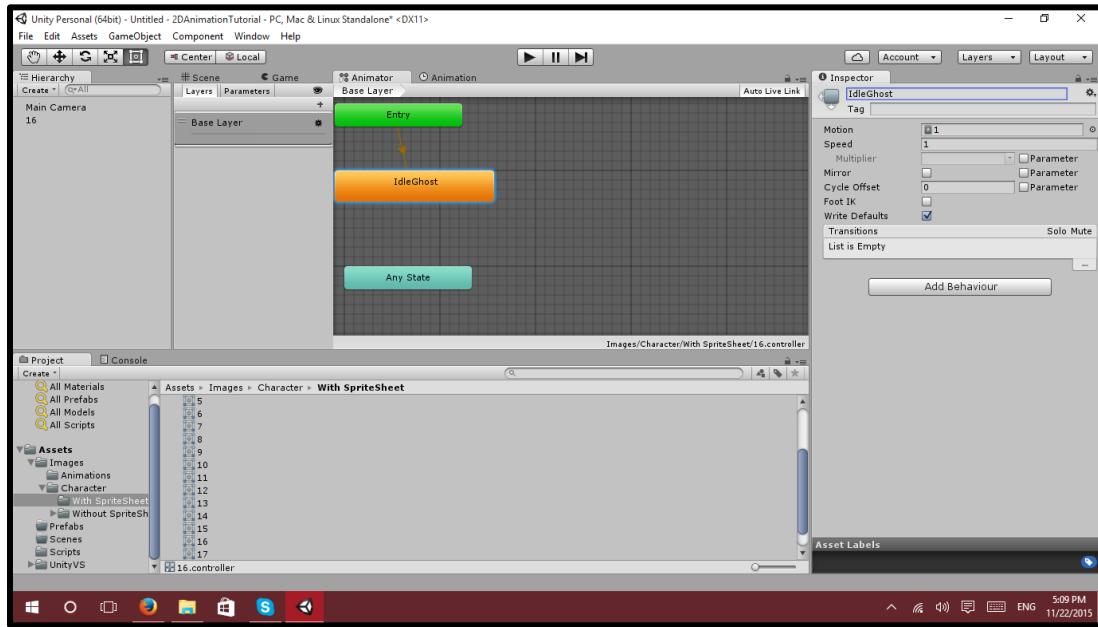
It should have the name 16 on it in the hierarchy view. Make sure to have that selected and click on the Animator. It will show you the beginnings of the Animation State Machine you can create.



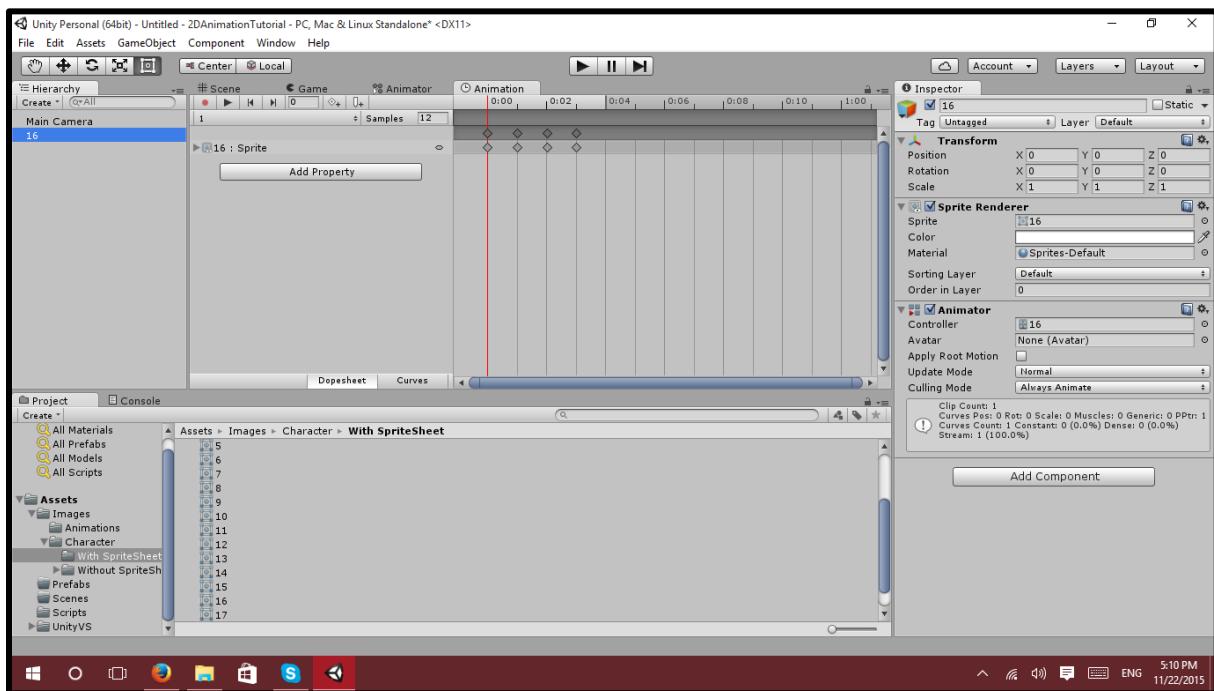
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

For convenience purposes, I rename it from 1 to IdleGhost.



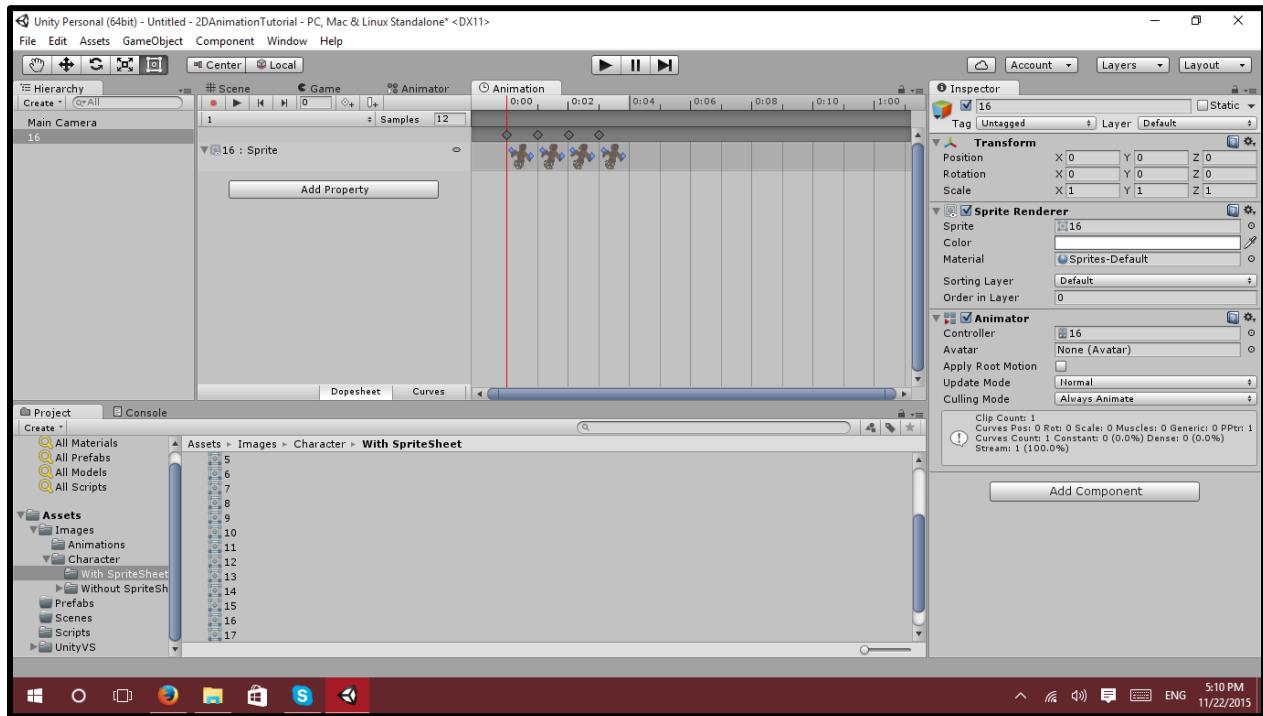
We are done with the Animator tab, I just wanted to show a brief preview of how the Animator is displayed. Don't worry, the next section will deal heavily with using it. Now we can view the Animation tab.



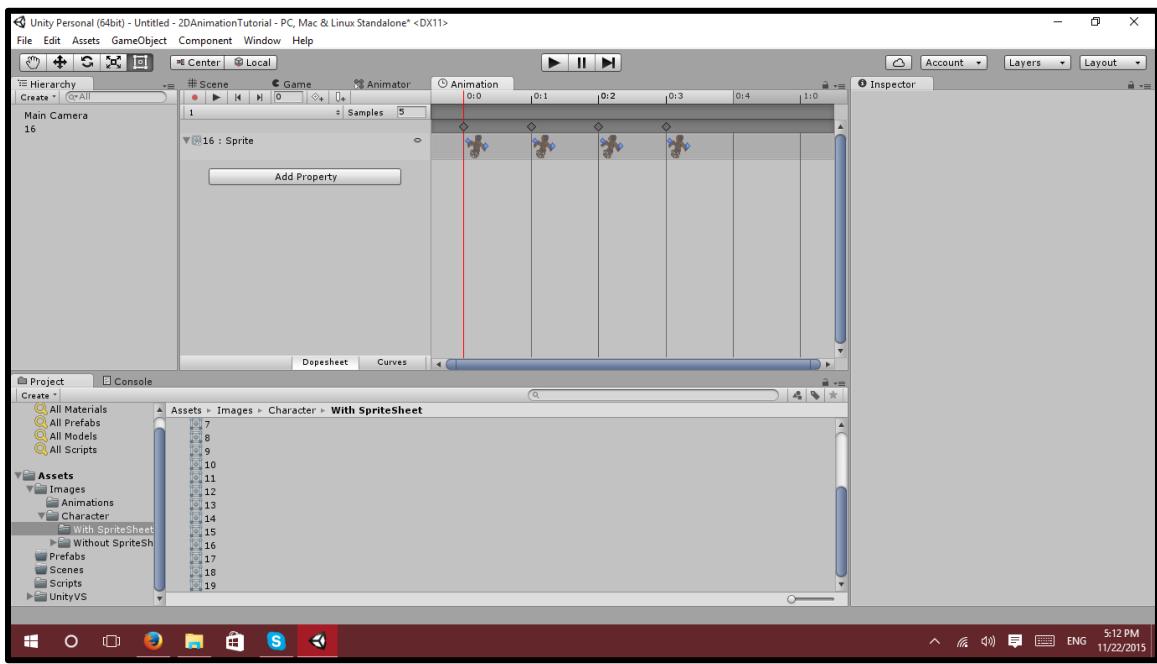
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

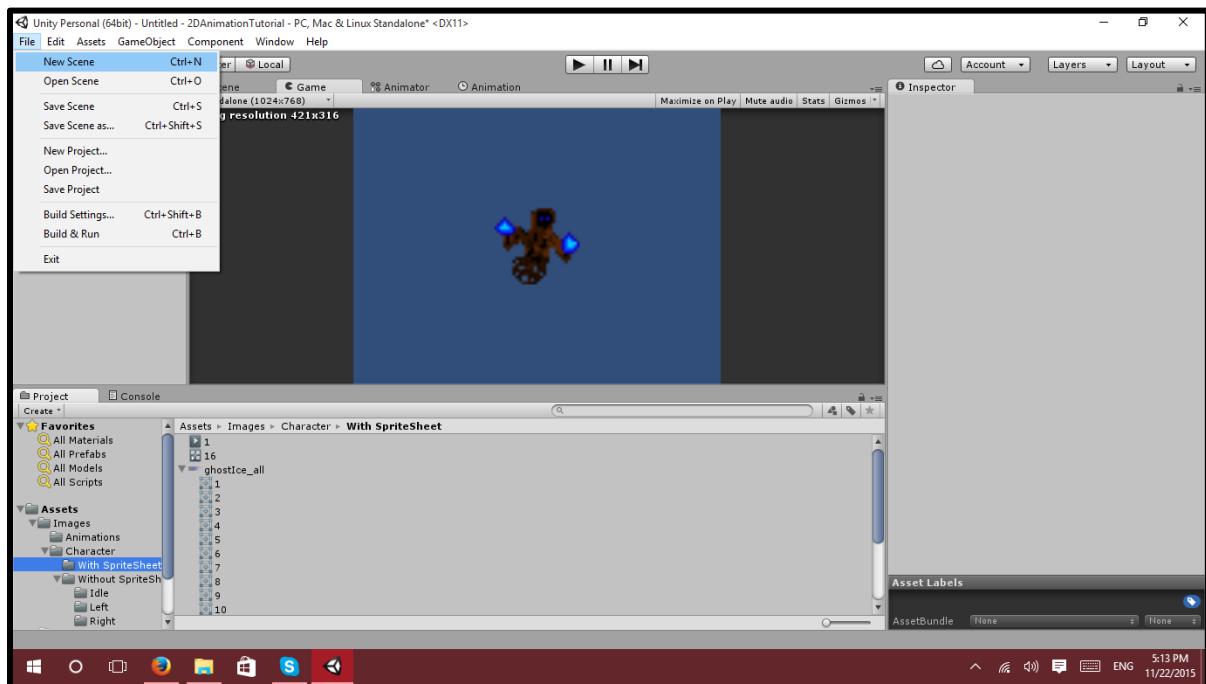
If you click on the right facing arrow next to 16: Sprite, it will give you a brief preview of the actual images used for this animation.



At the top of the Animation tab, you will notice a box that has the name Samples in it. The original sample rate is 12 and if you play the animation, it is going a bit too fast. Let's change the sample rate from 12 to 5.

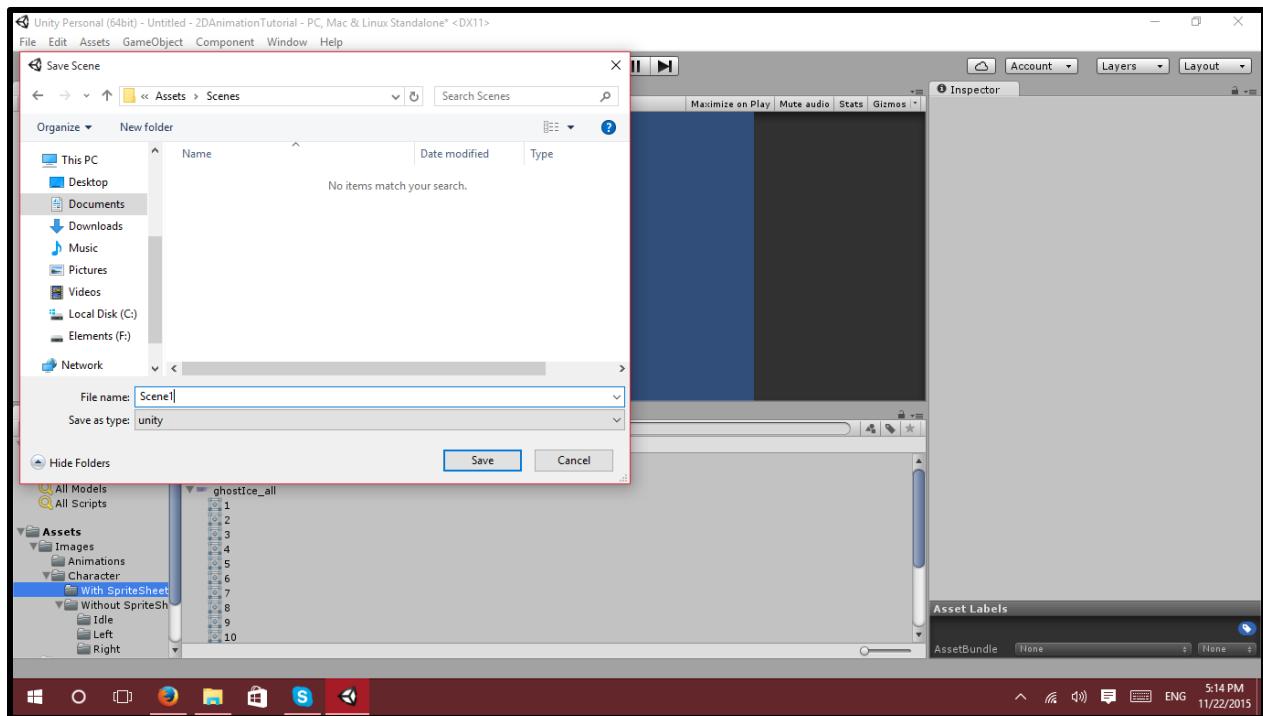
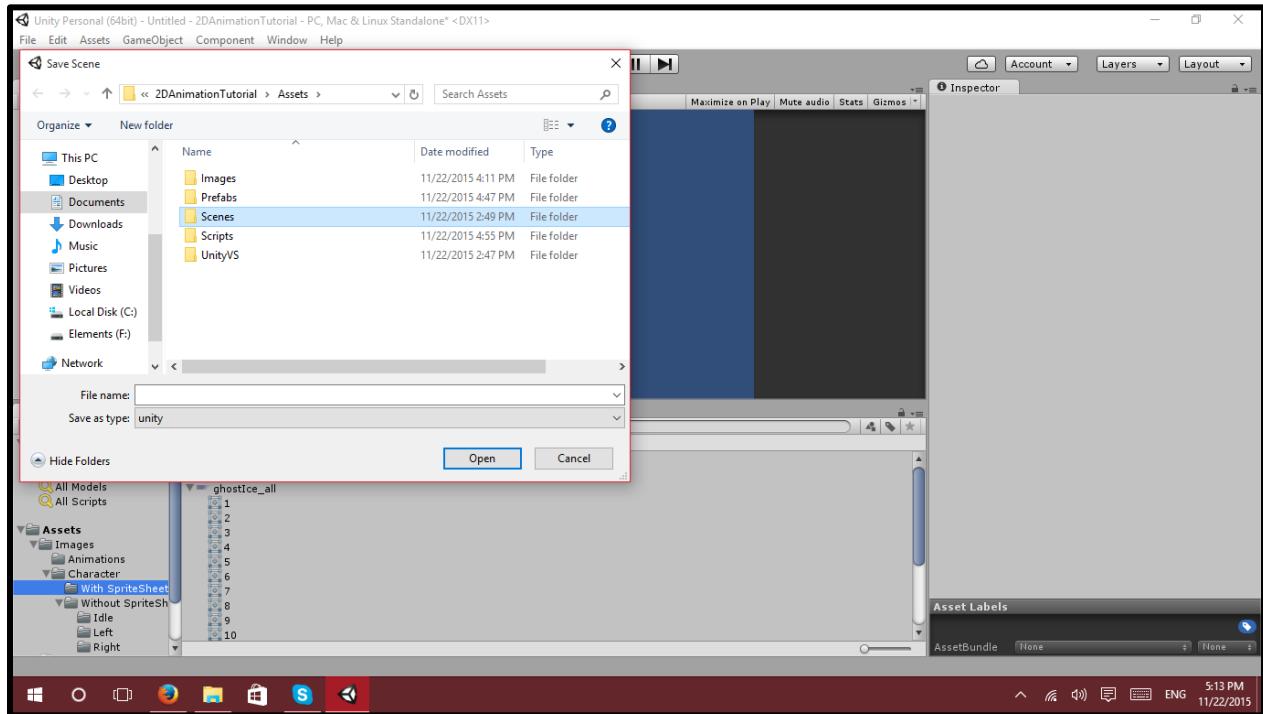


Go back to the scene view and press play. the ghost should now be animated on the screen. That is the overall basics of Animations within Unity3D. Before we go into the next section, let's finish this off to prepare for the next section. Click on file, New Scene. It will ask you if you want to save the old scene.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Save it in the Scenes folder under the name Scene1.



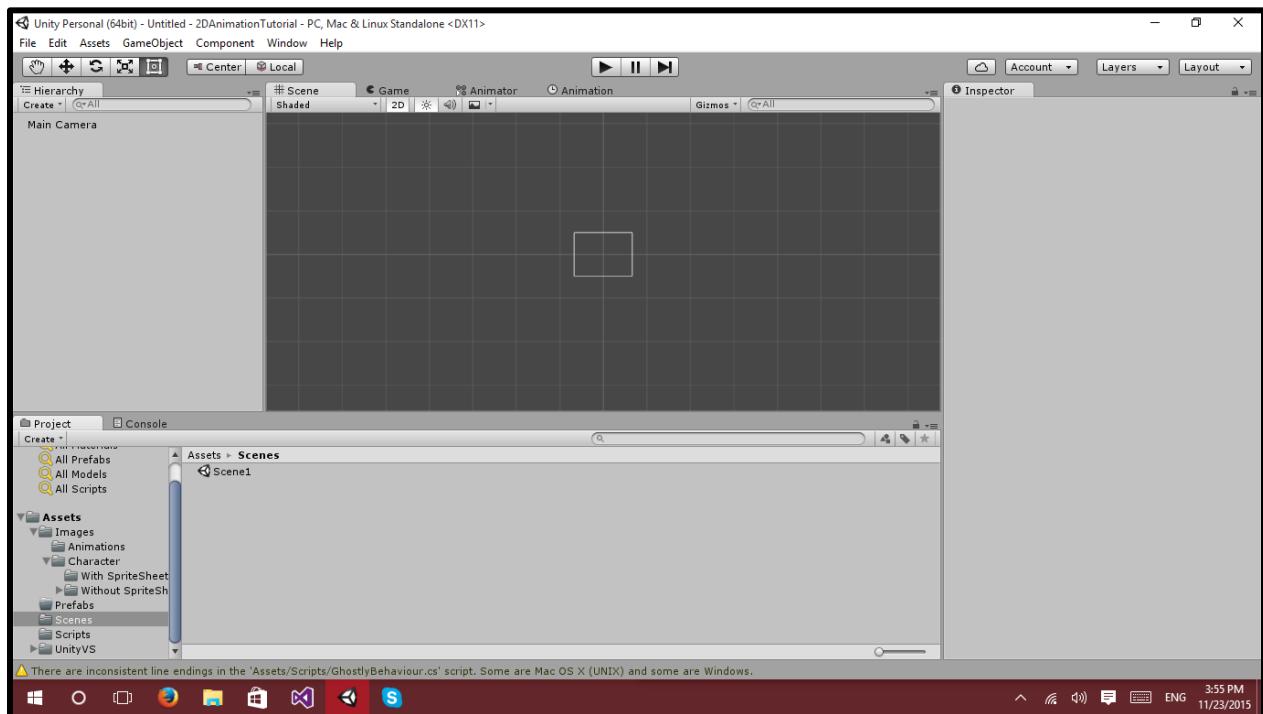
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

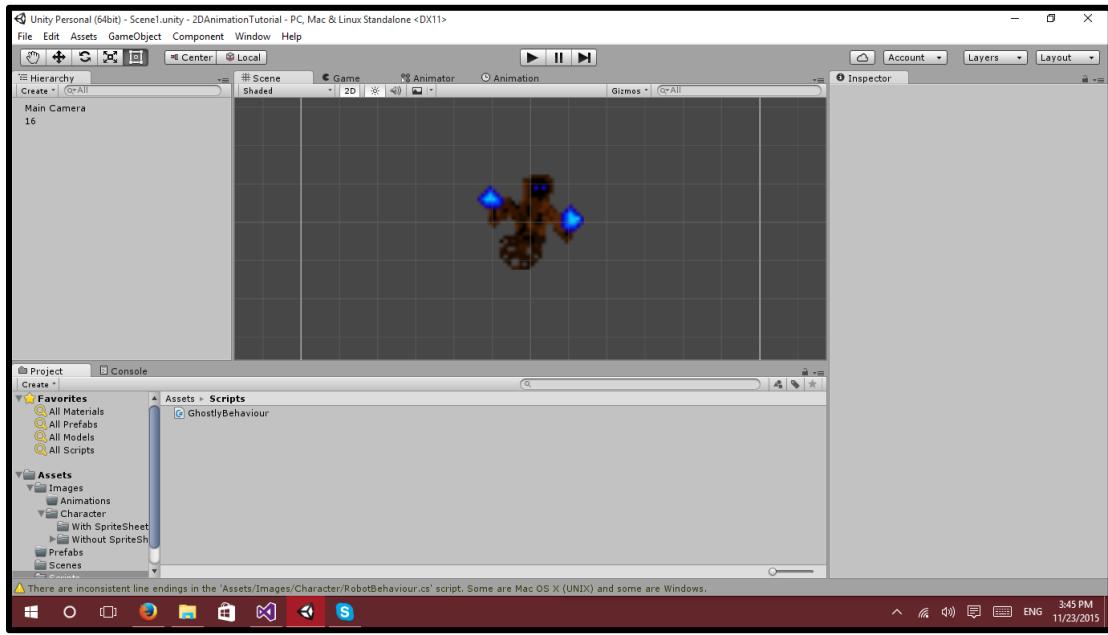
Section 3: Controlling animations via Scripting

I realized that anyone reading this tutorial would say, “Well, I know how to make an animation, but how do I control it using code?”; And this section will explain it all to you, using the scene we already created to show that. This section will be relatively short and to the point, so if you don’t fully understand it, Segment 2 will show it in more detail with more controls.

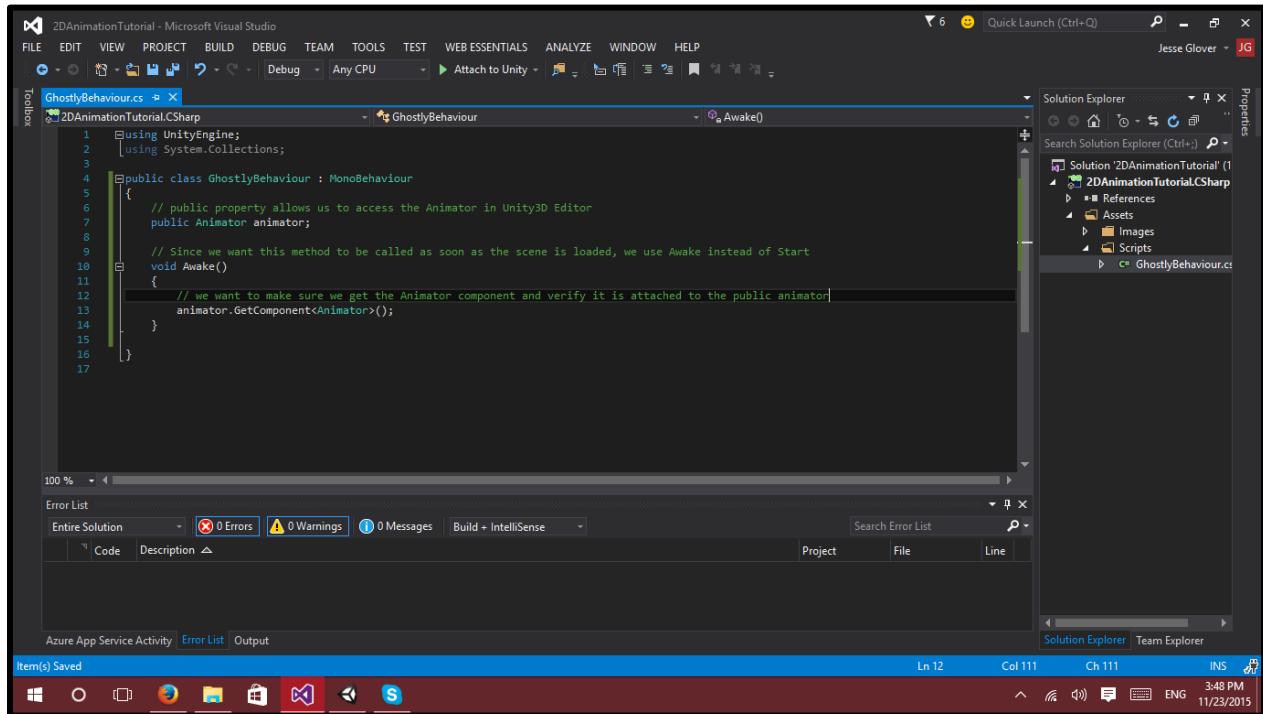
To get started, load Scene1 back up. To do this, click on the scene’s folder and double click on Scene1.



Next up, we create a script inside of the Scripts folder and call it GhostlyBehaviour.



Open up the script inside Visual Studio or MonoDevelop and Remove the Start and Update Methods from the editor. Create a public Animator and call it animator. We also want a void Awake method. The awake method should get the component of the Animator from the animator. Your code editor should look like the image below:



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

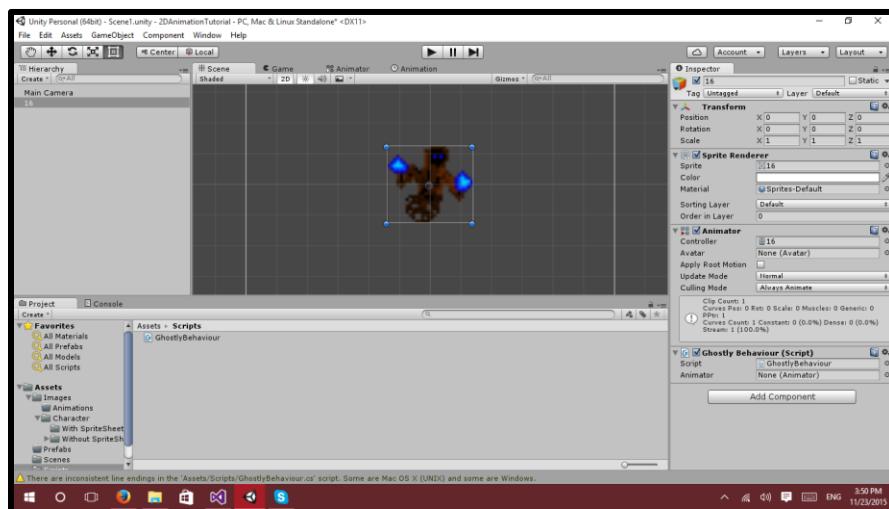
Now we want to add a void Update method. This next part is super simple. We take the animator's transform position which is a Vector 3 location and add to its current position a new Vector 3 to add 1 to its x position. Your code editor should look like this image below.

```

1  //using UnityEngine;
2  using System.Collections;
3
4  public class GhostlyBehaviour : MonoBehaviour
5  {
6      // public property allows us to access the Animator in Unity3D Editor
7      public Animator animator;
8
9      // Since we want this method to be called as soon as the scene is loaded, we use Awake instead of Start
10     void Awake()
11     {
12         // we want to make sure we get the Animator component and verify it is attached to the public animator
13         animator.GetComponent<Animator>();
14     }
15
16     // we want an action to occur every frame, so we use Update
17     void Update()
18     {
19         // we take the animator's position and make it add 1 to it every frame
20         // So, we generate movement to the right from the animation.
21         animator.transform.position += new Vector3(1, 0, 0);
22     }
23 }

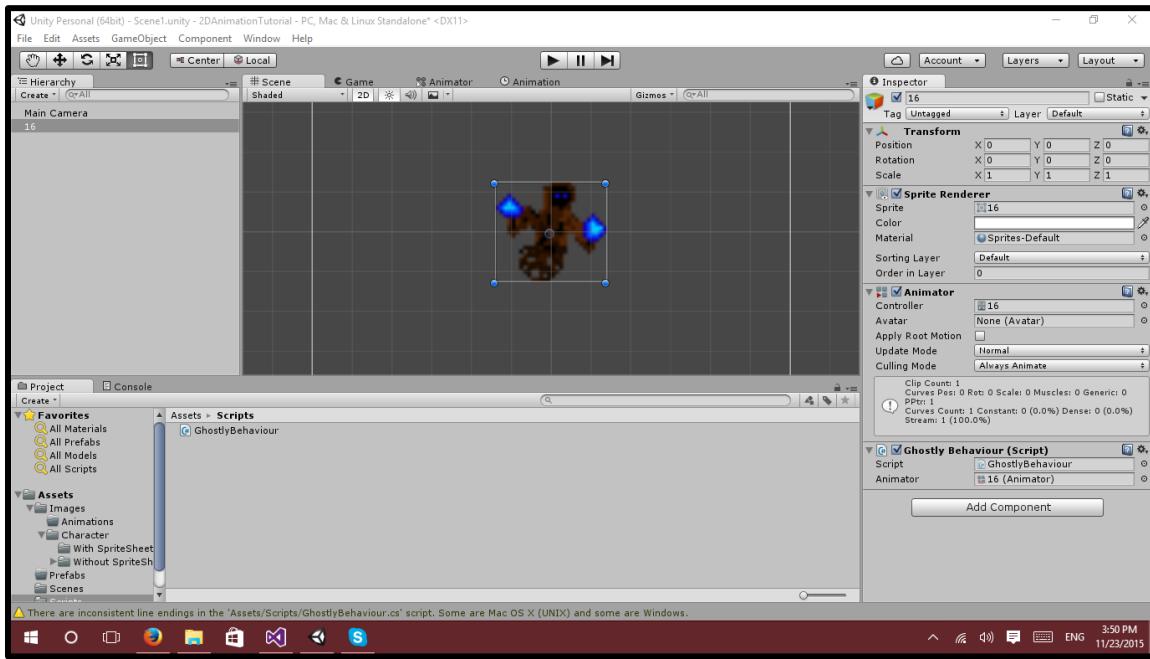
```

Go back into Unity3D, make sure the ghost animation is selected from within the Hierarchy pane. And in the inspector either drag the GhostlyBehaviour script onto the inspector or click Add Component, Scripts, GhostlyBehaviour.

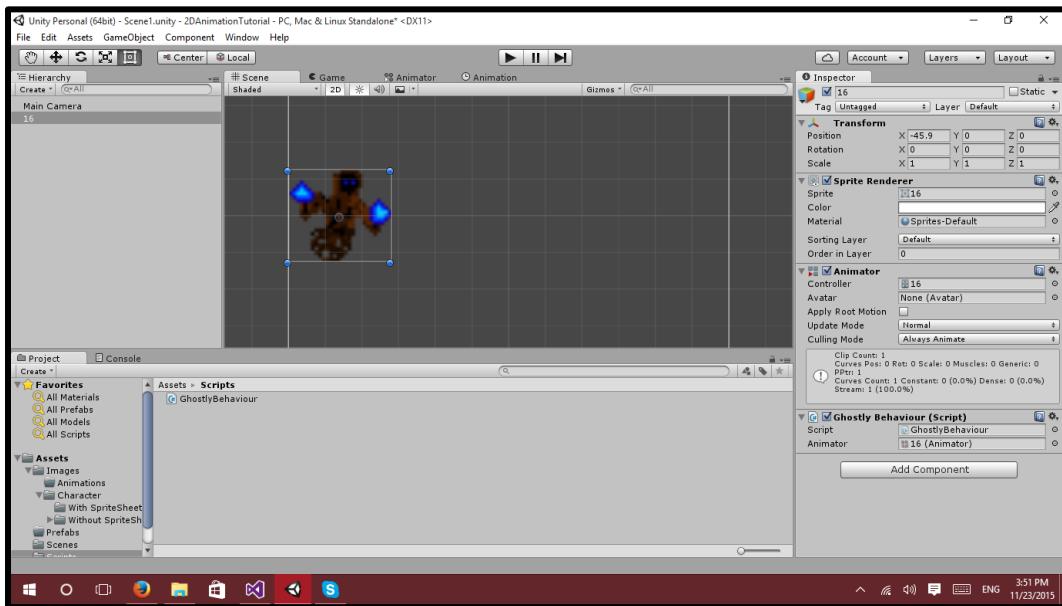


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Drag the Ghost Animation from the Hierarchy pane to the Animator underneath the Ghostly Behaviour Script. This ties the game object to the script and vice versa.



Last step, Change the Ghost's animation location via the transform position inside of the inspector pane. X should be -45.9, Y should be 0, and Z should also be 0.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

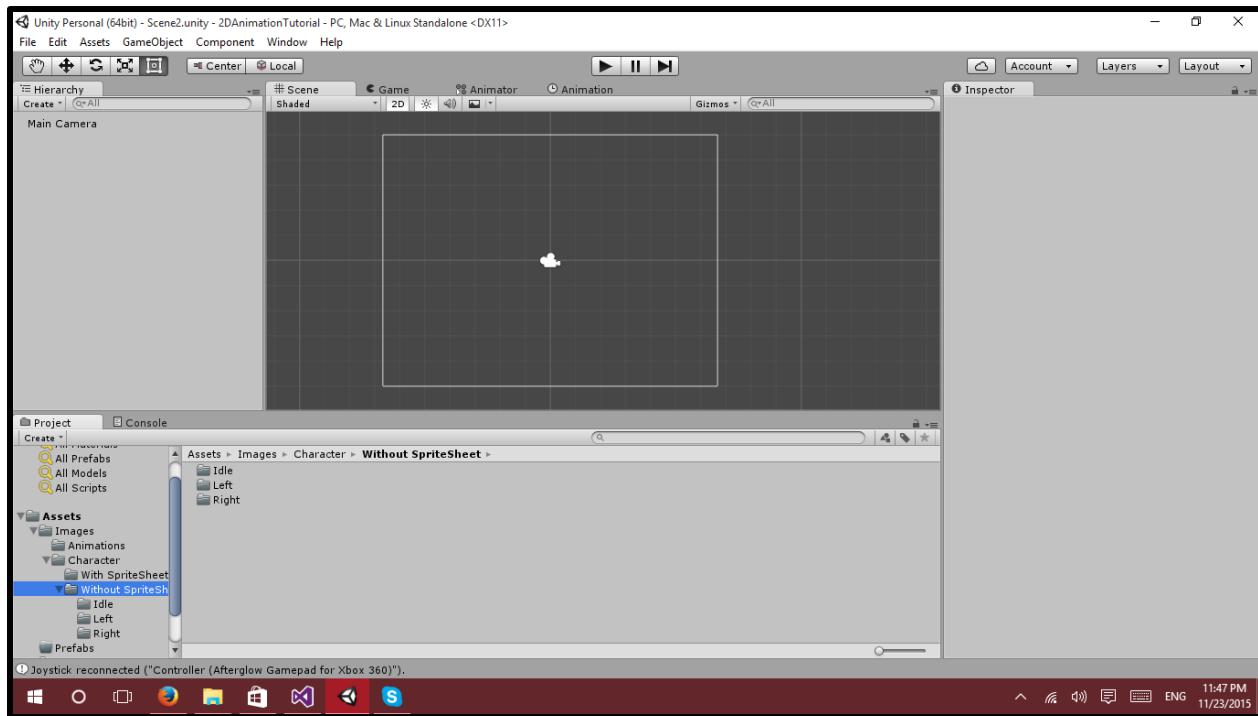
Run the program and your ghost should be animated and move across the screen until it is no longer within view and continue that way forever. Congratulations, you have just created a simple animation and controlled it via scripting.

Segment 2: Player controlled Sprite with Animations

This segment will take what we have learned from Segment 1, expand upon it a little and make a scene where you can control the character with the left and right mouse buttons. Although it doesn't sound very exciting, it will most certainly open doors for creating a full game with the techniques used in this article.

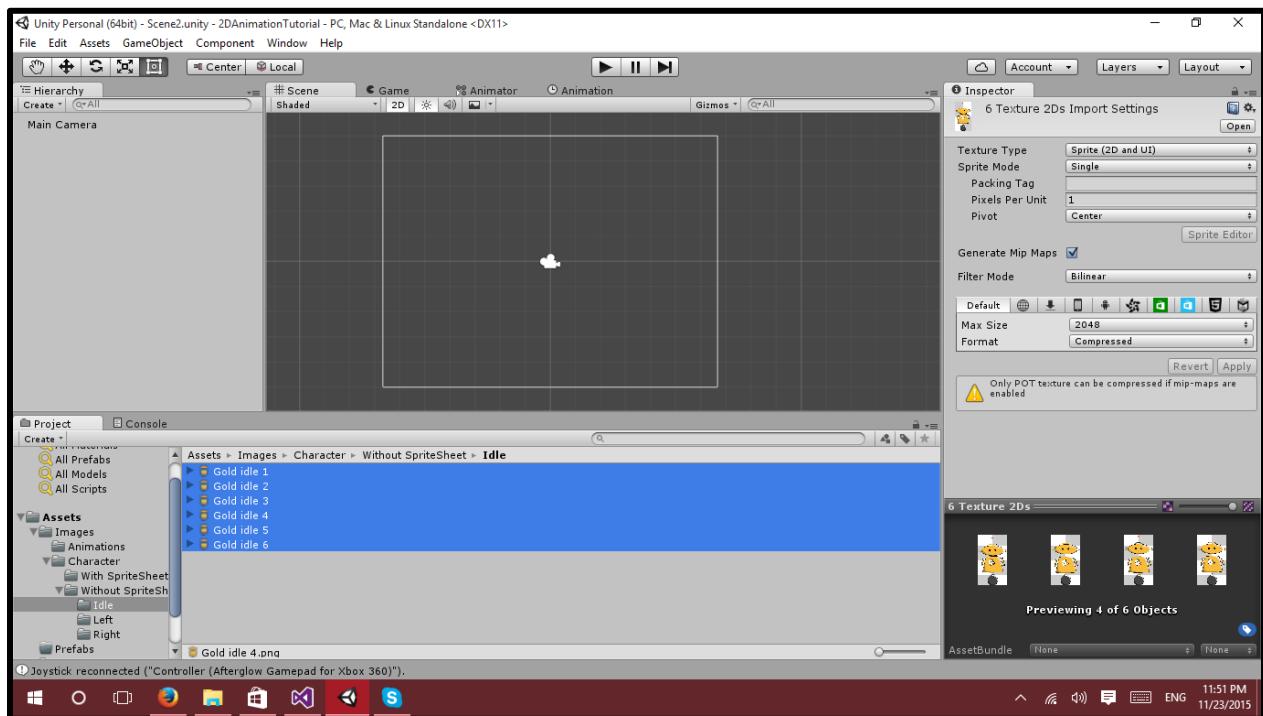
So where do we begin? Well, we always start from the beginning. Remember the new scene we created and then went back to Scene1? If you saved that new scene, go ahead and open that scene up, if not...

Click on new scene to begin. We want to navigate to the Folder WithoutSpriteSheet for this section of the project.

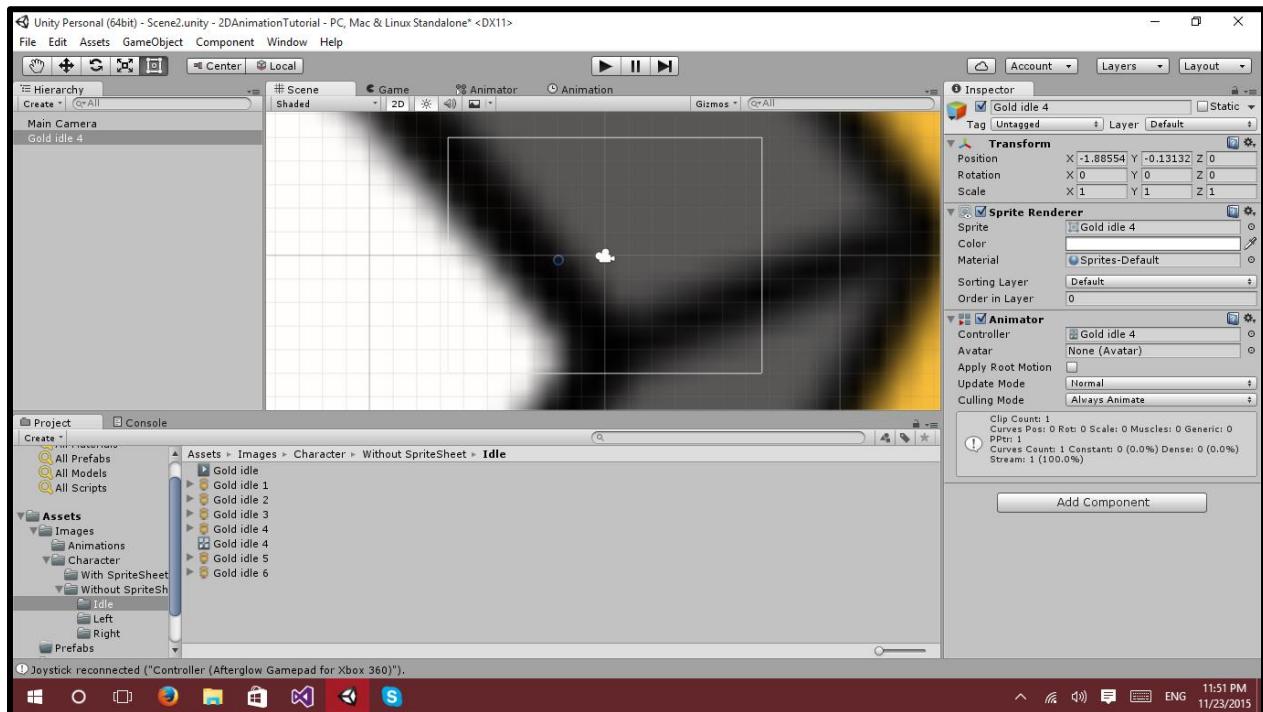


Open the Idle folder and reveal the sprites located within it. Ctrl + A or control click all of the images. Change the Pixels Per Unit to 1 in the Inspector tab. Everything else can stay the same and click apply.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity



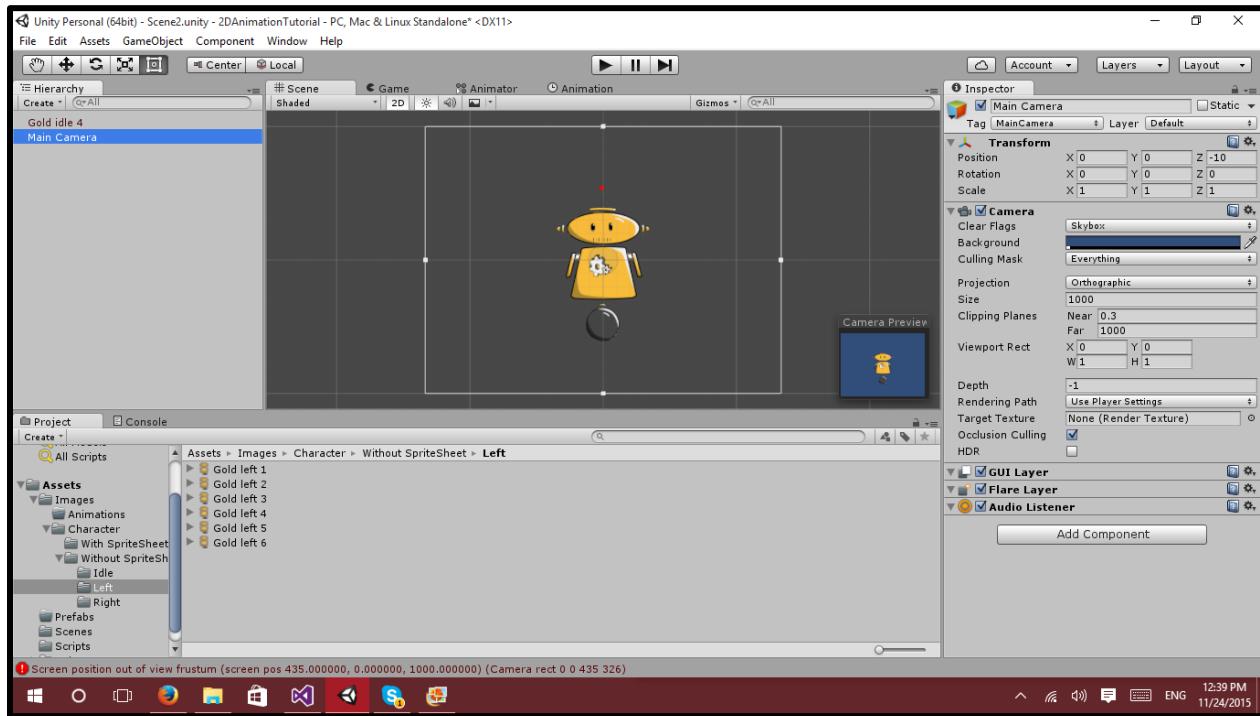
Drag it on the Scene.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

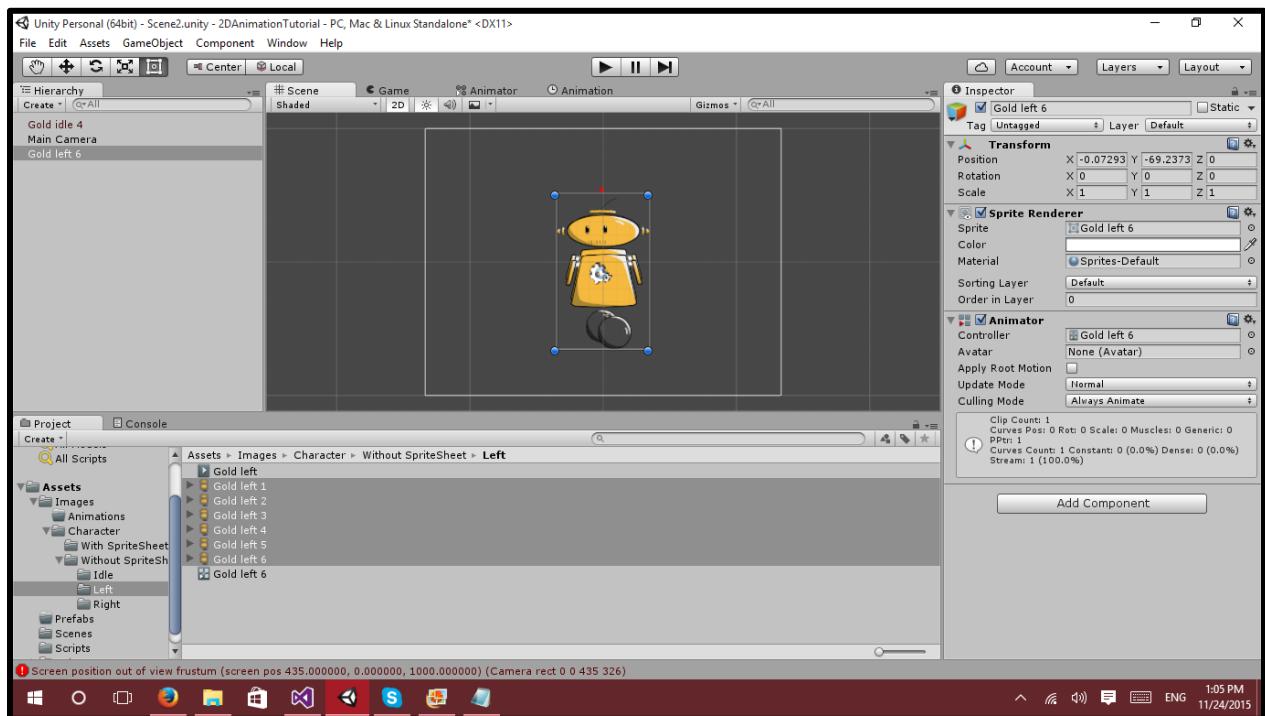
© Zenva Pty Ltd 2021. All rights reserved

Now, you may notice that the sprite is HUGE on the screen, no need to worry about it, we will address that in a moment. Simply click on the Camera on the Hierarchy tab and in the Inspector, change the Size to 1000. Boom, problem solved!

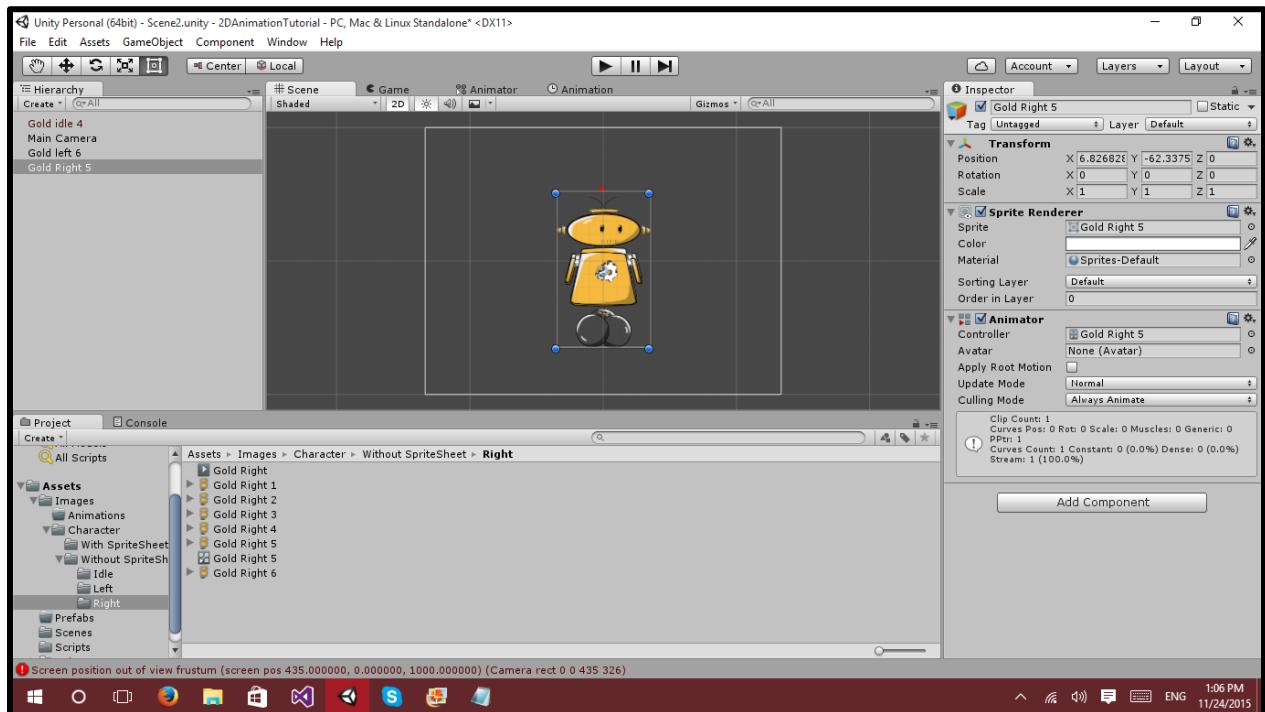


Now, Select the Left folder from within the Without SpriteSheet folder and again, we want to select all of the images and drag them on the scene. (I tend to drag them to the same position as the Idle animation on the scene).

You will see two new files on the screen. One is the Animation Clip and the other is the Animator Controller. We will need these later.



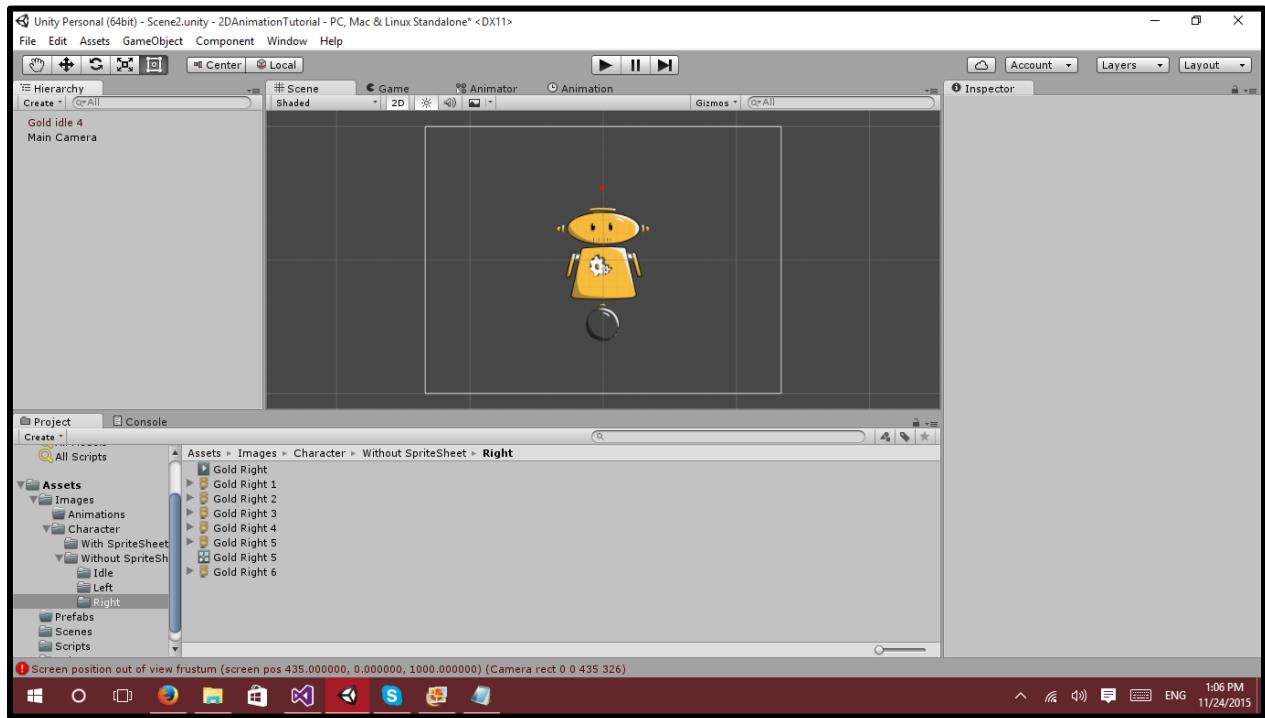
Repeat one last time with the images for Right inside of the Without SpriteSheet folder.



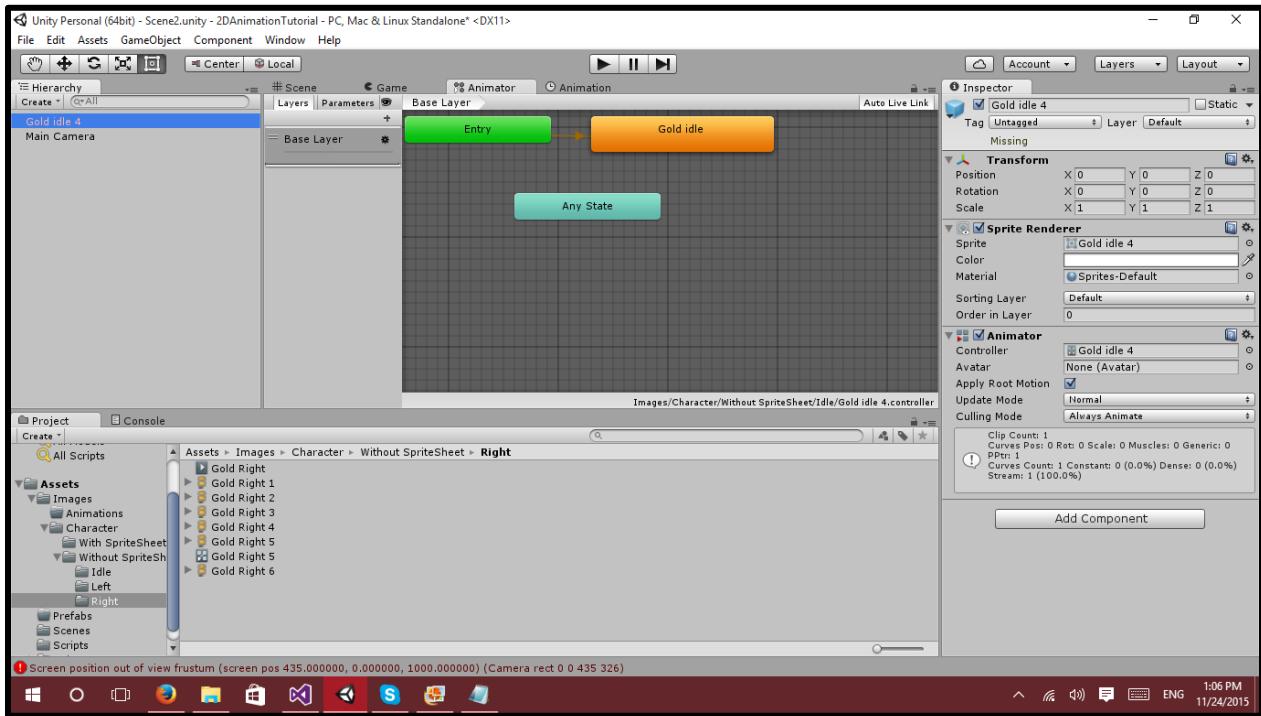
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

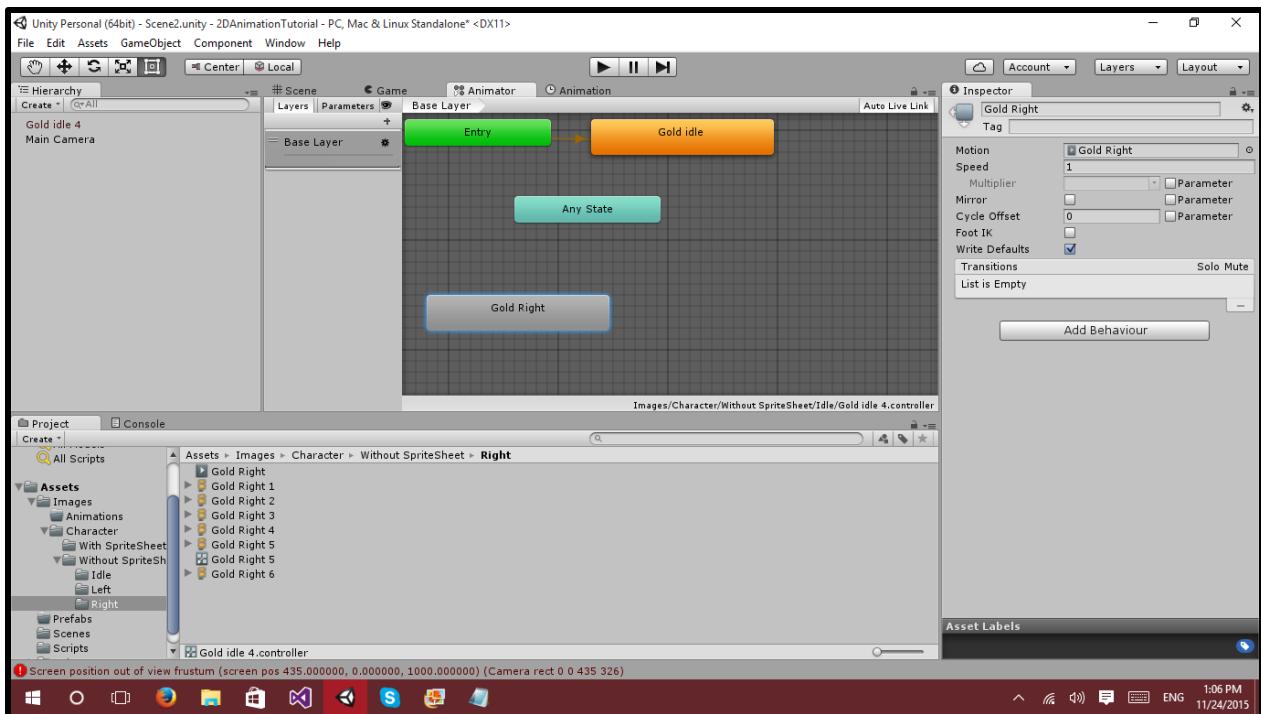
Now we can delete Gold Right and Gold Left from the Hierarchy pane.



Make sure you select Gold Idle 4 and then click on the Animator Pane next to Game. Your screen should roughly look like mine.

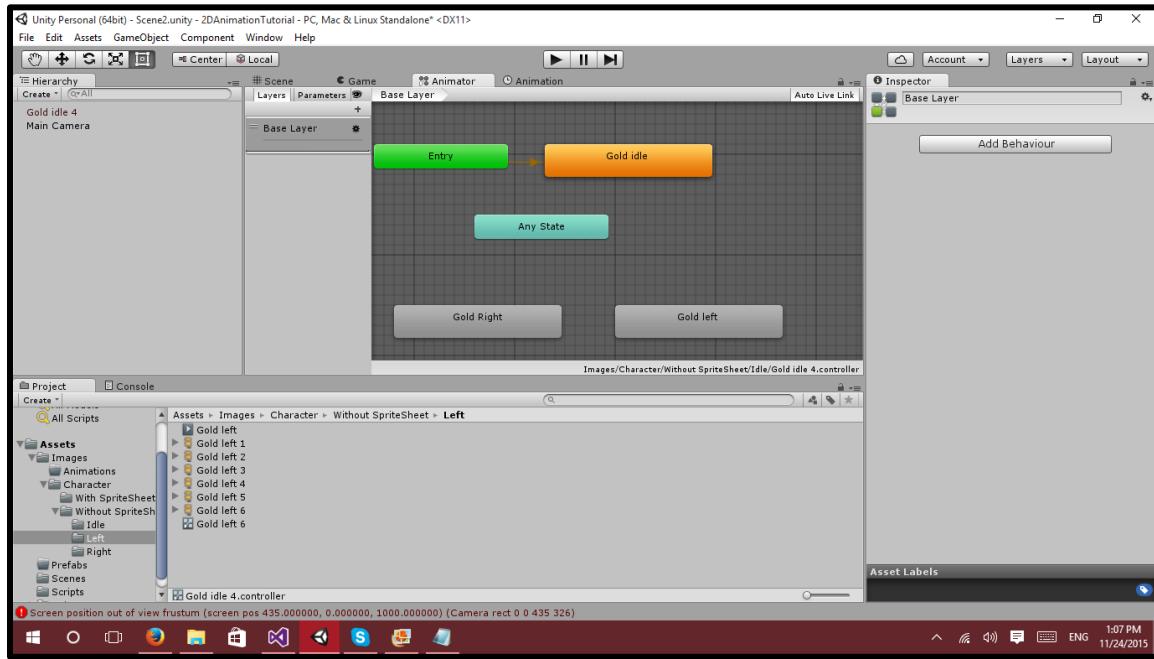


Navigate to the right folder inside of the without Spritesheet folder and drag the first item from the list, which is the Gold Right Animation Clip, to the animator.

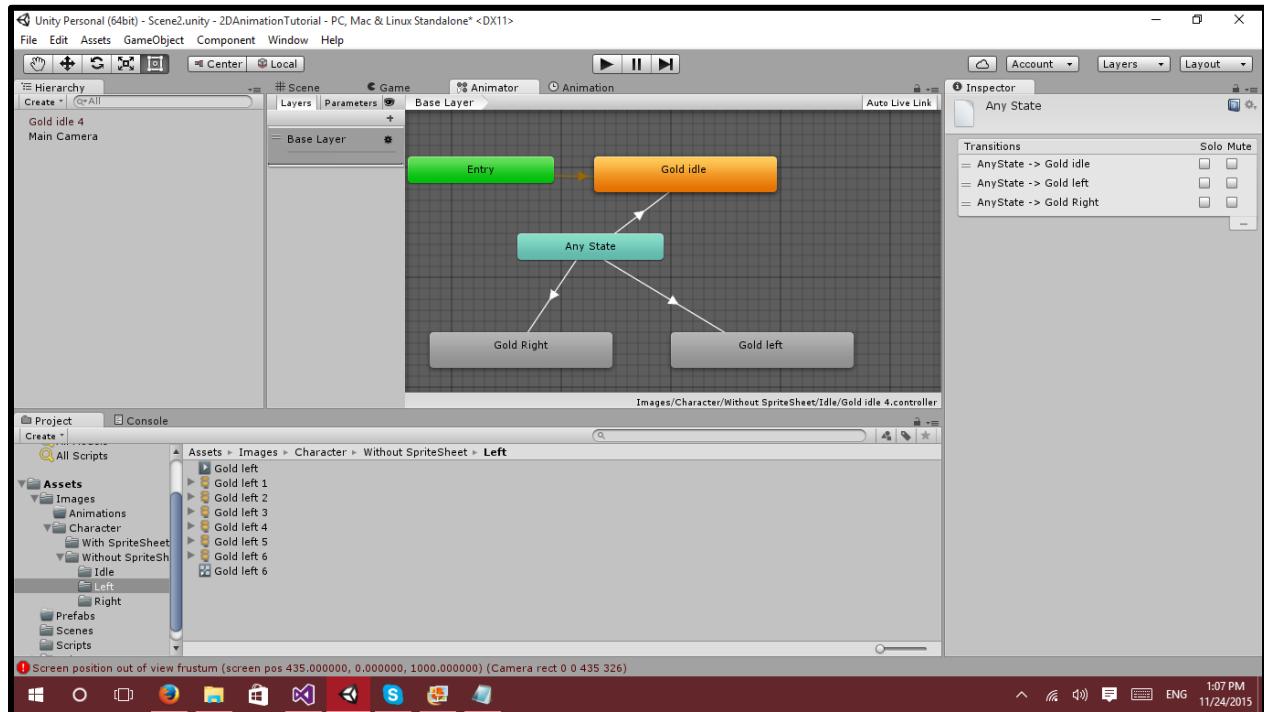


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Repeat for the Left Animation.



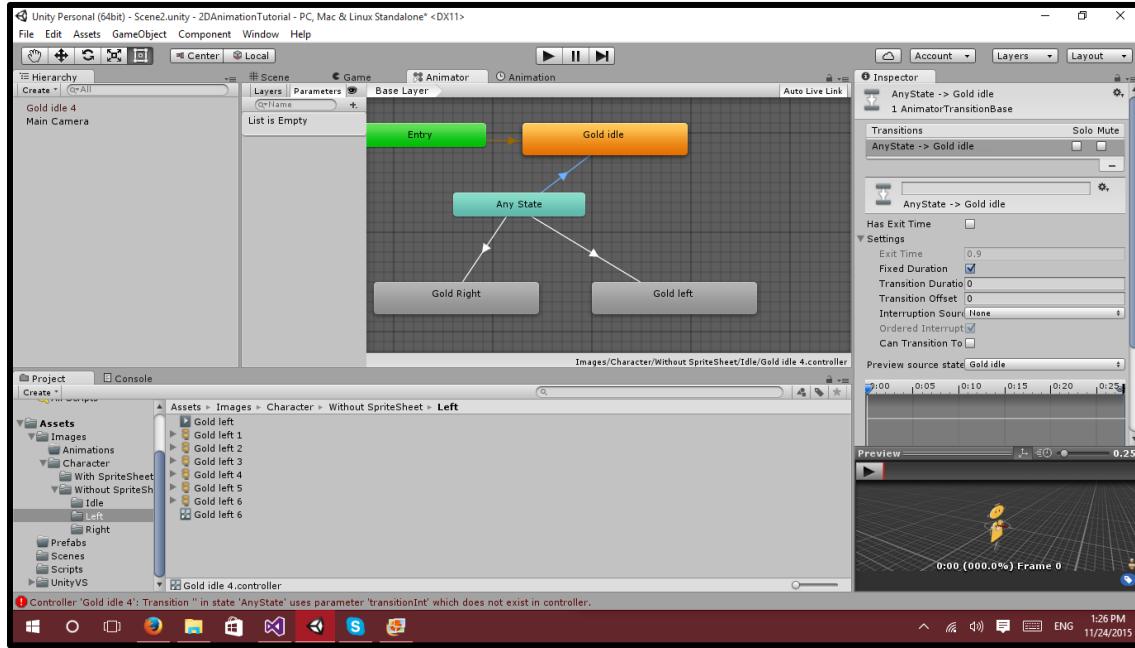
Make a transition from Any State to Gold Idle, Gold Right, and Gold Left.



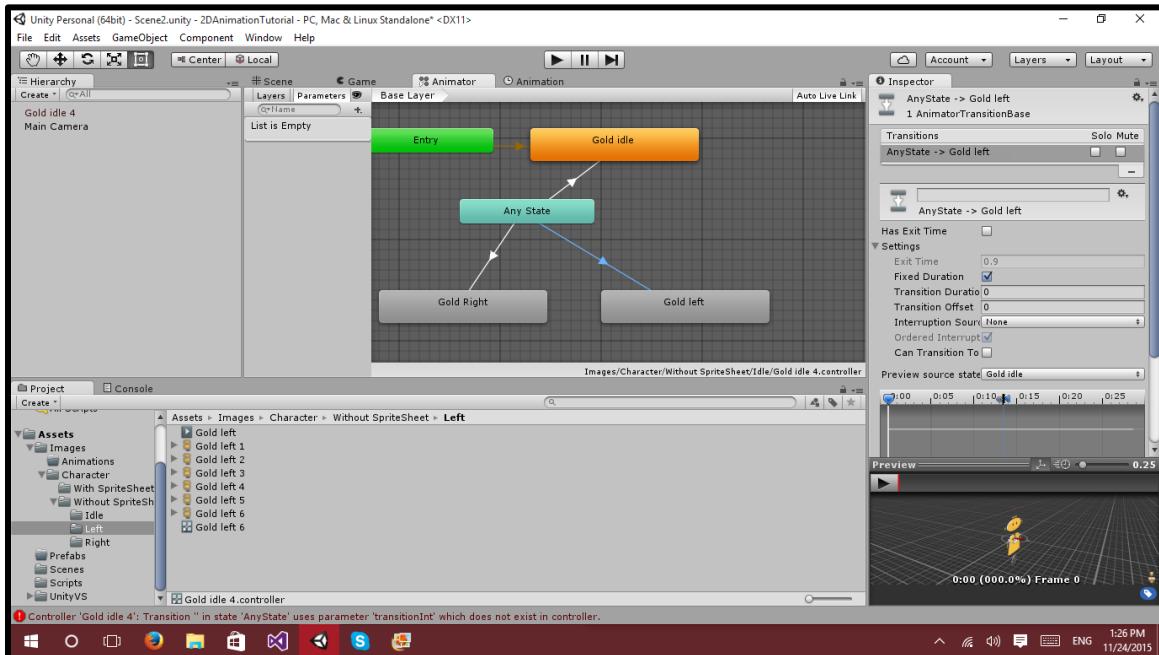
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Click on the transition from Any State to Gold Idle. Open the Settings and lets make a couple changes to it. Transition Duration should be set to 0 and Can Transition to Self check mark should be unchecked.

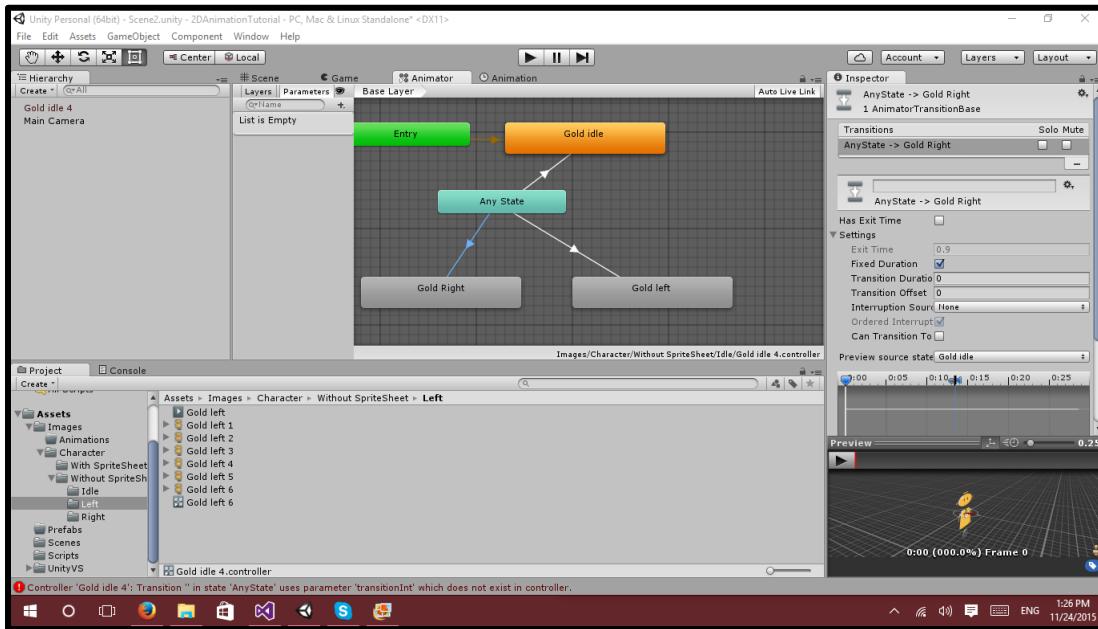


Repeat for transition from Any State to Gold Left



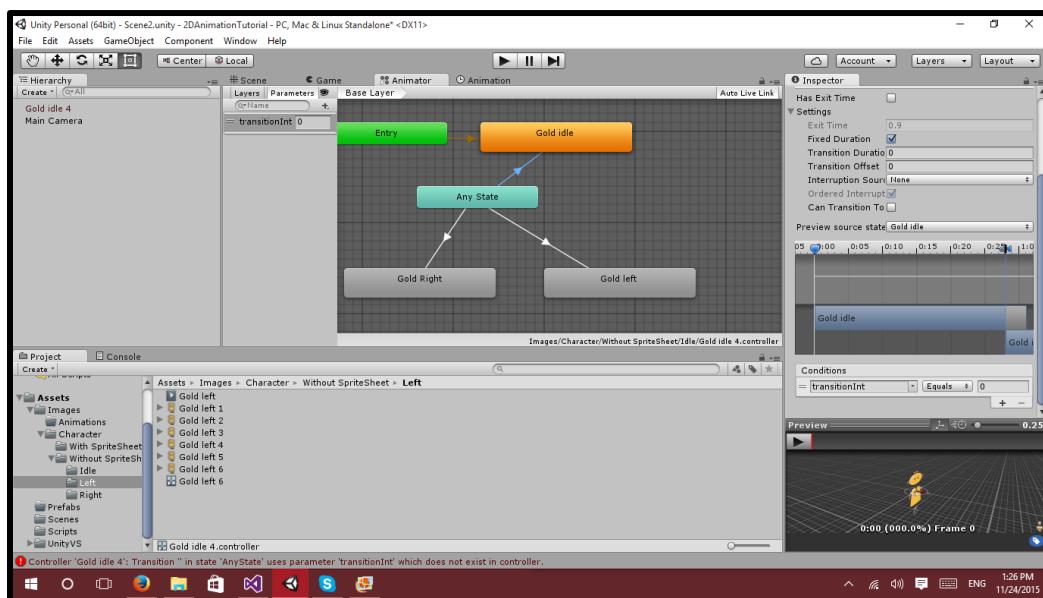
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Last time, repeat for Any State to Gold Right.



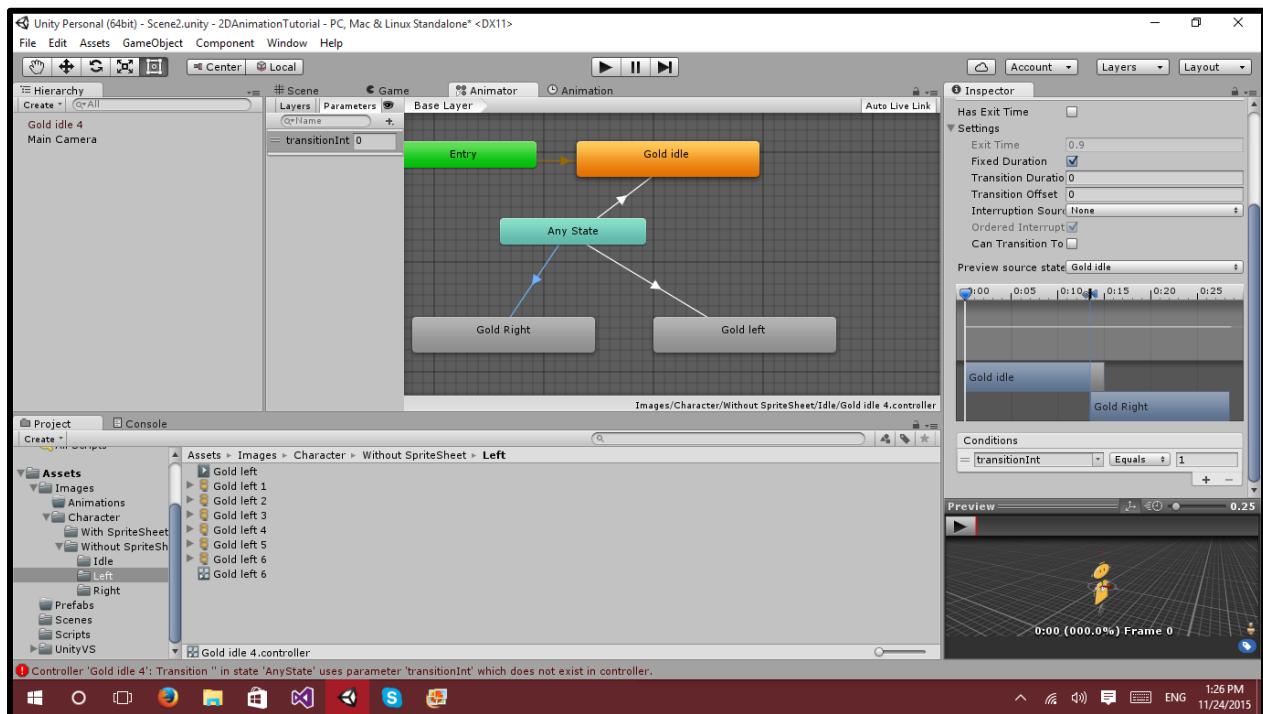
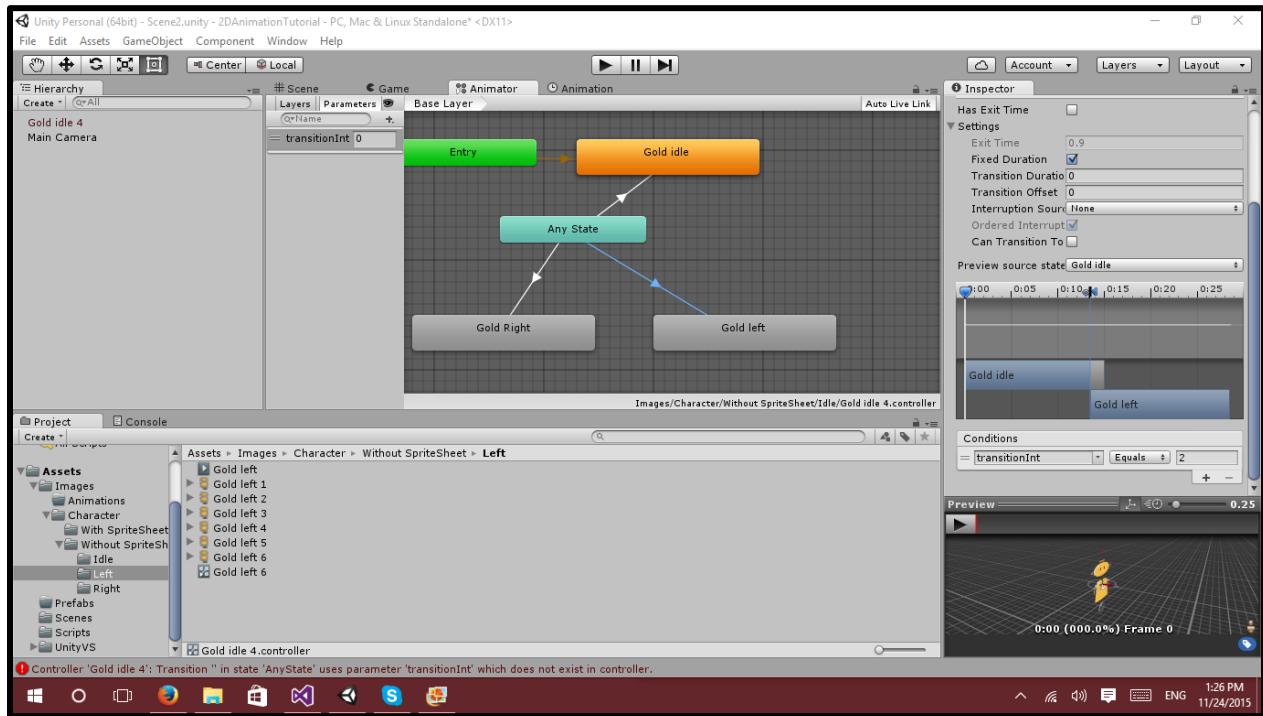
Next to layers, you should see Parameters, click on that. Then Click on the plus icon just underneath that. We want this to be set as an Integer (Int) and we can leave it set to 0 for now.

Once that is set, click on the transition from Any State to Gold Idle and look in the Inspector pane for Conditions. Add the Condition we created called transitionInt and set it to equals 0.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

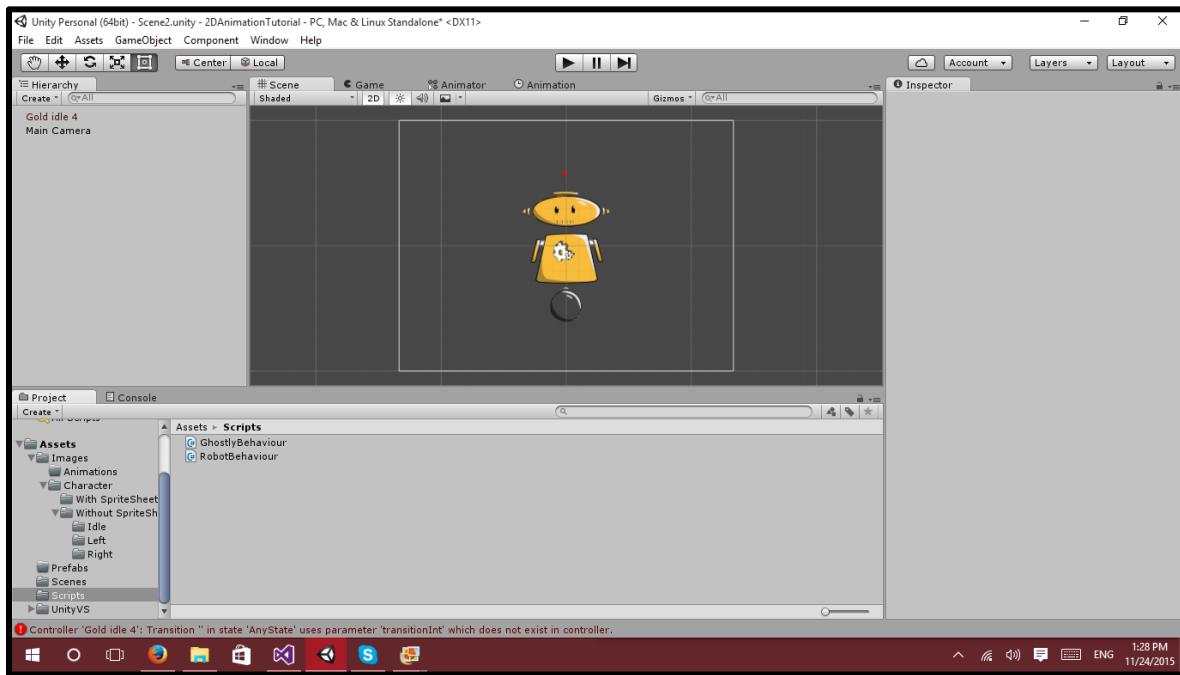
for Gold Left and Gold Right's transition from Any State, we repeat but making sure we change the int values to 1 and 2 respectively.



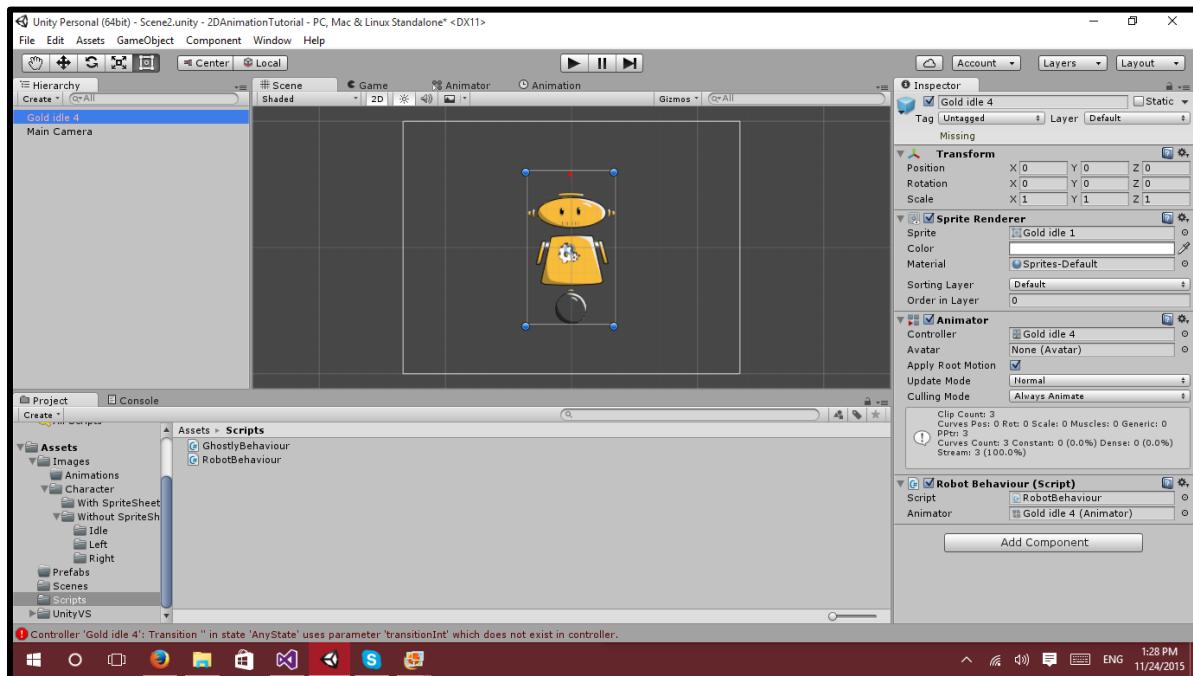
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Go back to Scene view. Now create a new script called RobotBehaviour.



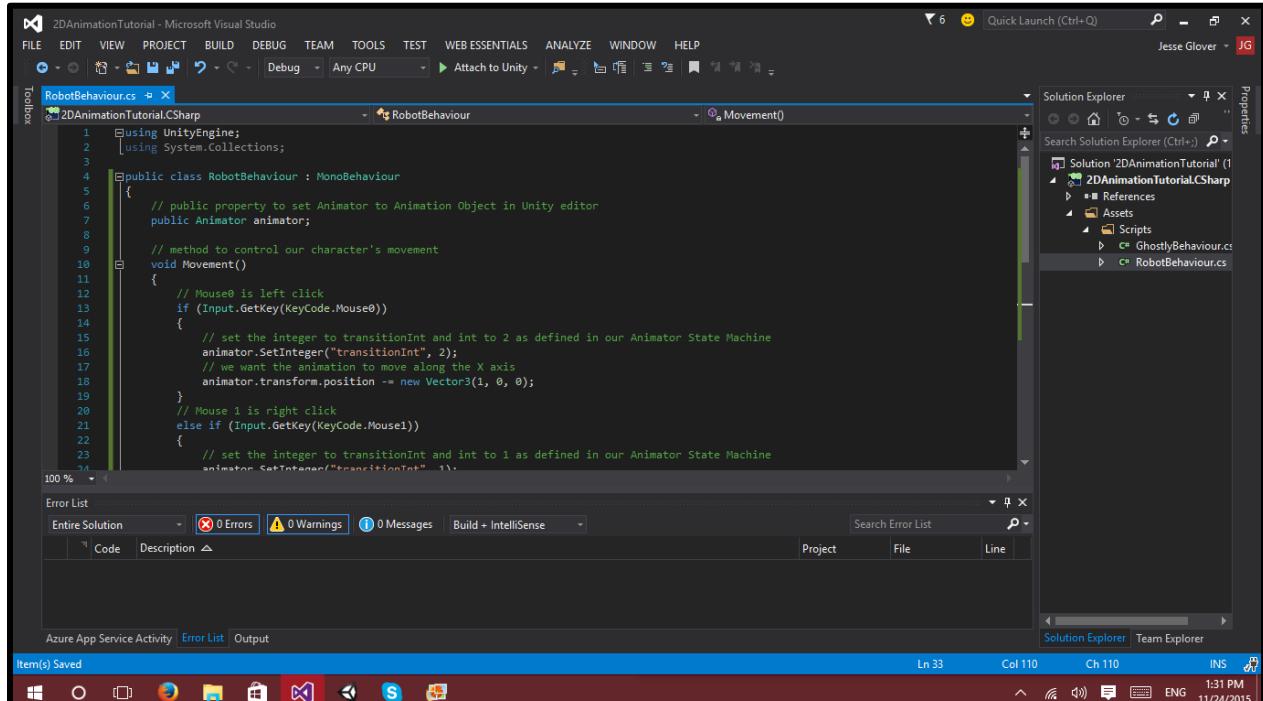
Click on Gold Idle 4 inside of the Hierarchy pane and drag your RobotBehaviour script onto the Inspector pane for Gold idle 4.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

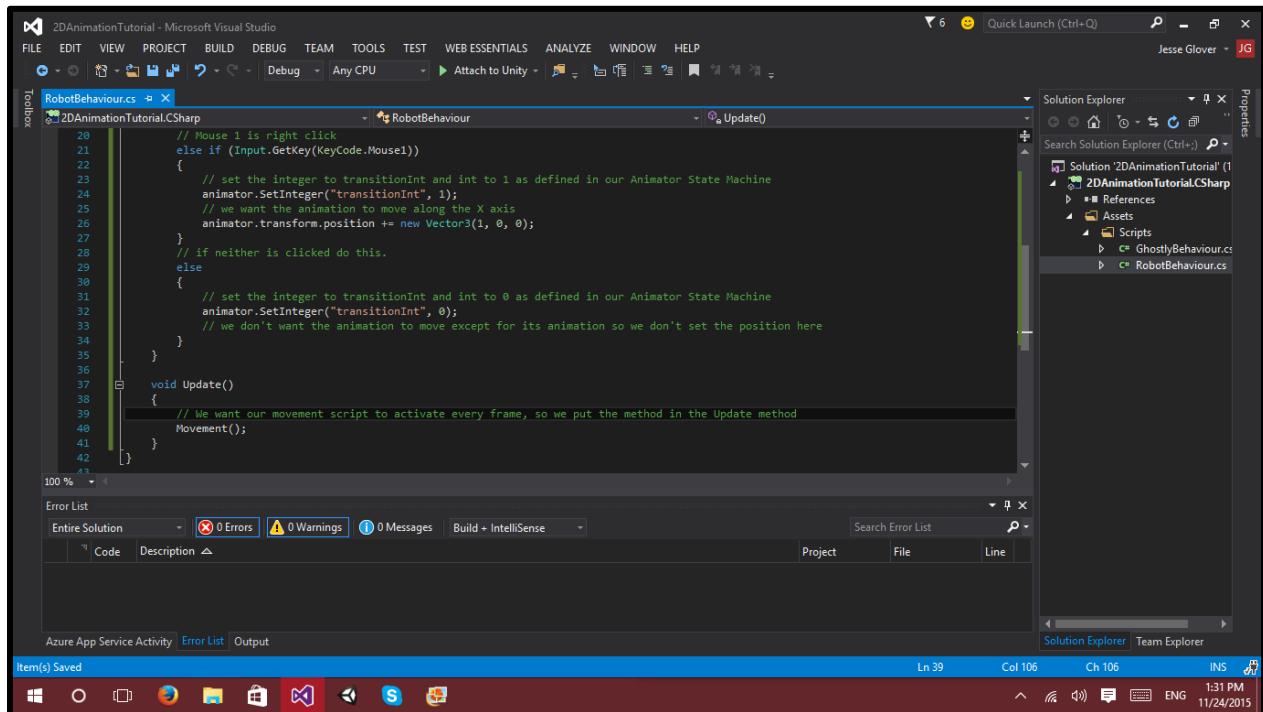
Open up the script inside of Visual Studio or MonoDevelop. Your code should look like the images below.



The screenshot shows the Microsoft Visual Studio interface with the 'RobotBehaviour.cs' file open in the editor. The code implements movement logic based on mouse clicks:

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class RobotBehaviour : MonoBehaviour
5  {
6      // public property to set Animator to Animation Object in Unity editor
7      public Animator animator;
8
9      // method to control our character's movement
10     void Movement()
11     {
12         // Mouse0 is left click
13         if (Input.GetKey(KeyCode.Mouse0))
14         {
15             // set the integer to transitionInt and int to 2 as defined in our Animator State Machine
16             animator.SetInteger("transitionInt", 2);
17             // we want the animation to move along the X axis
18             animator.transform.position += new Vector3(1, 0, 0);
19         }
20         // Mouse 1 is right click
21         else if (Input.GetKey(KeyCode.Mouse1))
22         {
23             // set the integer to transitionInt and int to 1 as defined in our Animator State Machine
24             animator.SetInteger("transitionInt", 1);
25         }
26     }
27
28     void Update()
29     {
30         // Want our movement script to activate every frame, so we put the method in the Update method
31         Movement();
32     }
33 }
```

The Solution Explorer shows the project structure with files 'RobotBehaviour.cs' and 'GhostlyBehaviour.cs' in the 'Scripts' folder.



The screenshot shows the Microsoft Visual Studio interface with the 'RobotBehaviour.cs' file open in the editor. The code has been modified to handle both mouse clicks and key presses:

```
20     // Mouse 1 is right click
21     else if (Input.GetKey(KeyCode.Mouse1))
22     {
23         // set the integer to transitionInt and int to 1 as defined in our Animator State Machine
24         animator.SetInteger("transitionInt", 1);
25         // we want the animation to move along the X axis
26         animator.transform.position += new Vector3(1, 0, 0);
27     }
28     // if neither is clicked do this.
29     else
30     {
31         // set the integer to transitionInt and int to 0 as defined in our Animator State Machine
32         animator.SetInteger("transitionInt", 0);
33         // we don't want the animation to move except for its animation so we don't set the position here
34     }
35 }
36
37 void Update()
38 {
39     // Want our movement script to activate every frame, so we put the method in the Update method
40     Movement();
41 }
```

The Solution Explorer shows the project structure with files 'RobotBehaviour.cs' and 'GhostlyBehaviour.cs' in the 'Scripts' folder.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Let's take a moment and discuss the code above. Whenever we call a component, we always want to make sure we get a reference to it. The most effective way to do so is by using a public property. We could have very easily just put this code inside of the update method, although by convention, you want to avoid this.

Instead of doing a generic left and right arrow or specific buttons on the keyboard, I felt that controlling the sprite by mouse movement would be a bit more fun to code. Input is a utility built directly into MonoDevelop and does not use the built in .NET Framework's capabilities on it.(For those curious about this, they extend upon what was built into the .NET Framework and used their own).

Input.GetKeyDown is a boolean by nature, which is why we can use it inside of an if statement without issue. And we specify which direct boolean we want to compare to, in this case it is KeyCode.Mouse0 (left mouse button) and KeyCode.Mouse1(right mouse button). If neither of these two buttons have been pressed, play the idle animation and do not move. Otherwise, if the right mouse button is pressed, play the right movement animation and move to the right by 1 on the x axis.(Same goes for moving to the left).

Now you can save your scene. Click run and reap the rewards of a job well done! Congratulations, you have now animated a sprite and allowed it to move. You are well on your way to becoming a game developer using Unity3D. Give yourself a pat on the back for a job well done!

As always, if you have any questions or comments about this tutorial please comment below. Alternatively, if you are having any issues learning about a specific portion of Unity3D, comment below and we can make a tutorial on it as soon as possible.

Fundamentals of 3D Development with Unity3D

By Jesse Glover

Today, we will start the deep dive into what most programmers see as the holy grail of game development, 3D games. Unity3D makes it so much simpler than the days of old. You don't have to build your very own physics engine or guess at how things look in 3D space. Whilst Unity3D does make it easier to do, there are still plenty of things to learn before you jump into 3D development.

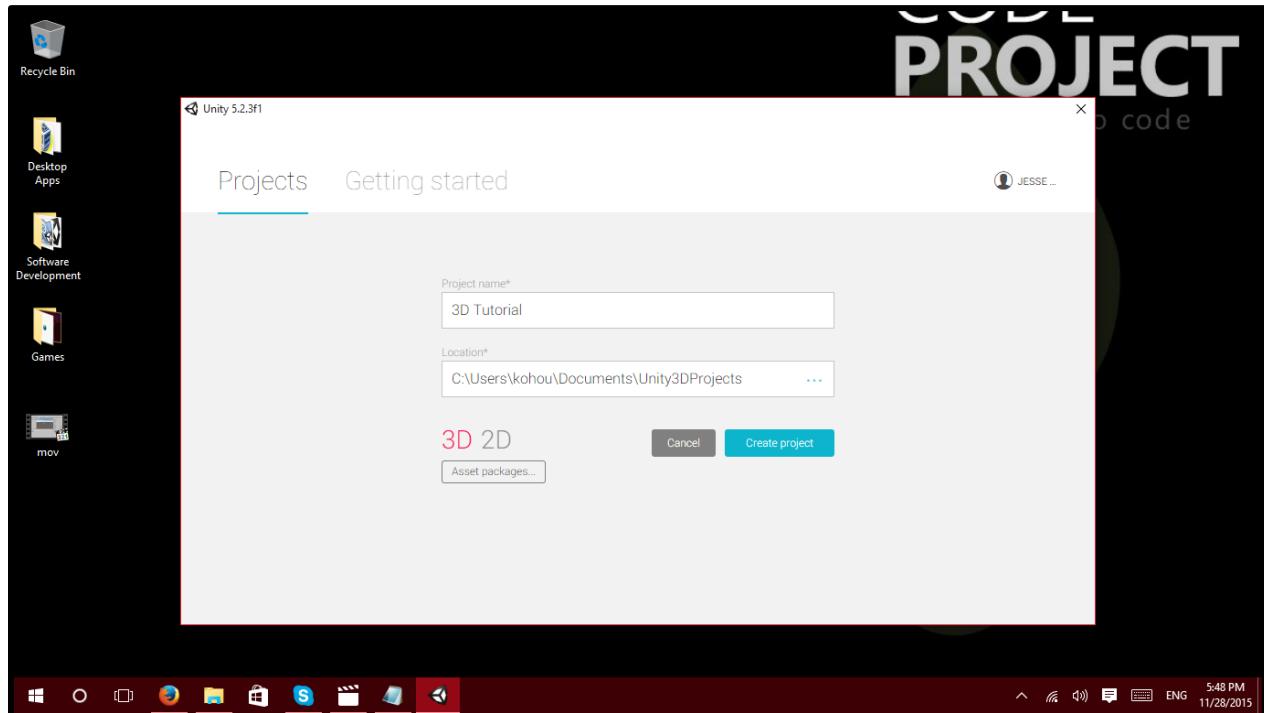
Before we get started. I wanted to point out that the 3D models I showcase today were created inside of a program called MagicaVoxel which can be downloaded from <http://ephtracy.github.io/>

You can download the source for this project [here](#).

[Note: most of my 3D tutorials will be using this program for the assets]

Section 1: Import a 3D Model into Unity3D

I'm glad you are ready to get started with 3D development with Unity3D. First off, we should go ahead and create a new project. I have named this one as 3D Tutorial. Take care to be absolutely certain you have 3D selected instead of the usual 2D.

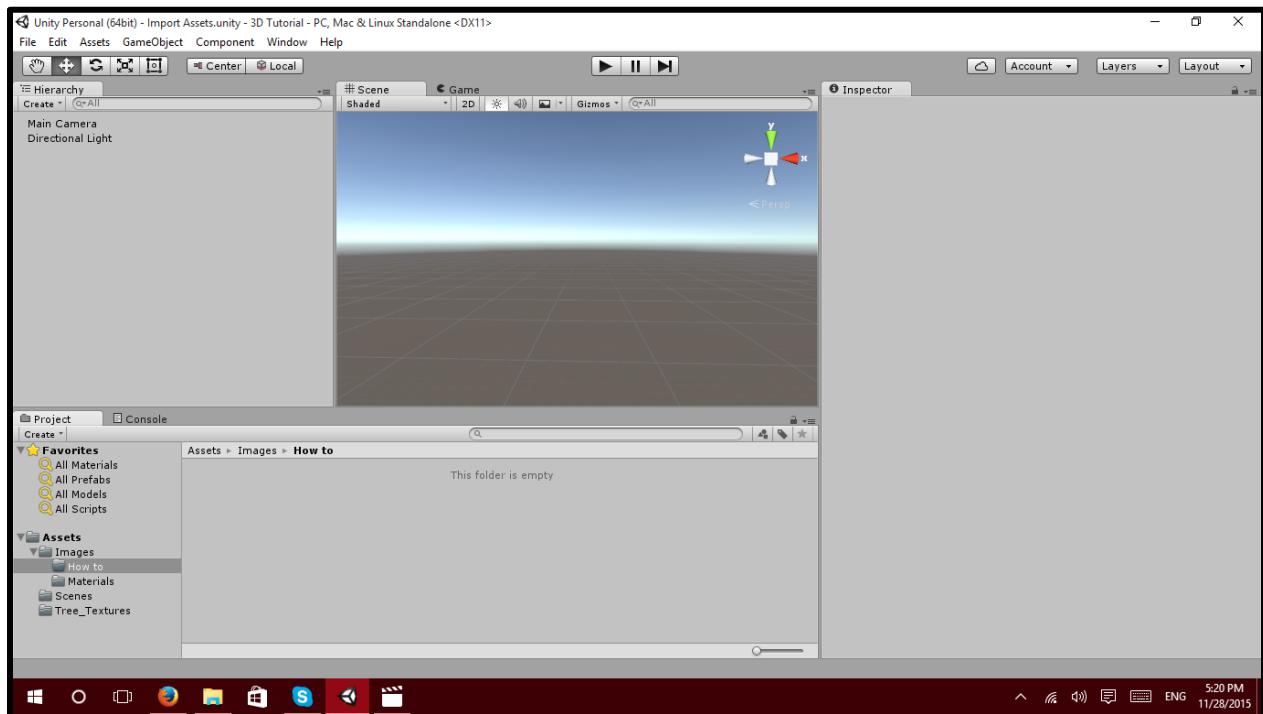


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

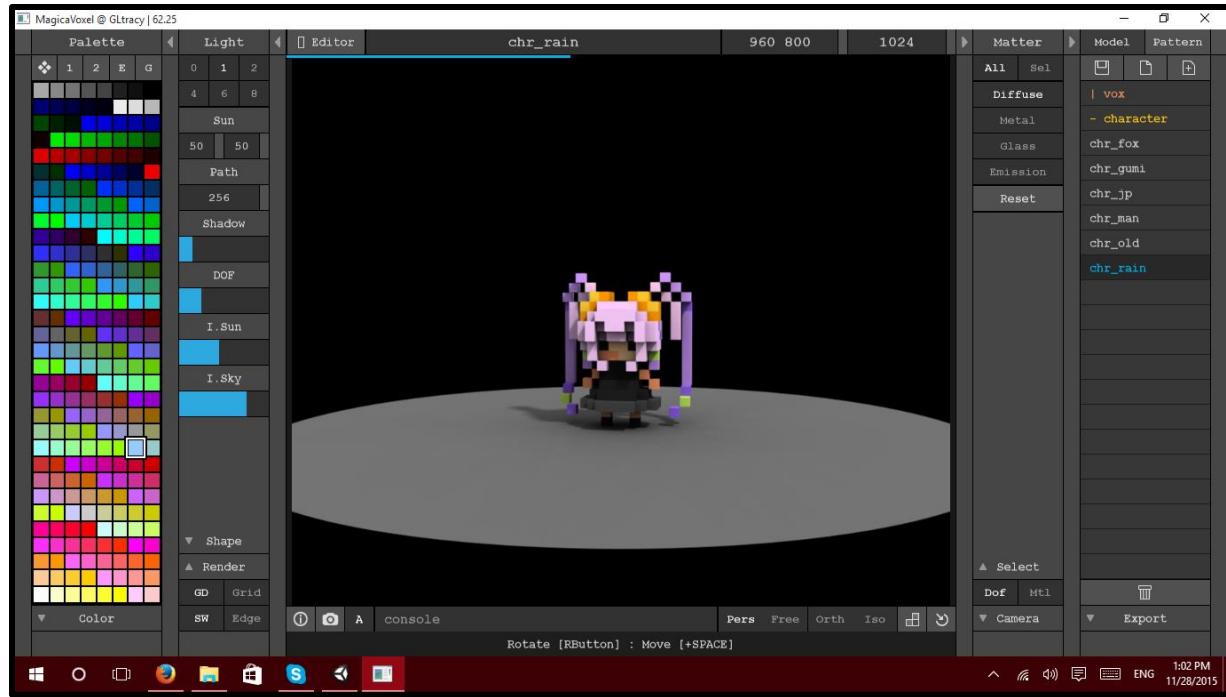
© Zenva Pty Ltd 2021. All rights reserved

Once that is done we need our standard folders inside of the Assets folder. (Scripts, Scenes, Images, Prefab) You can omit Scripts and Prefabs since we will not be using them in this tutorial.

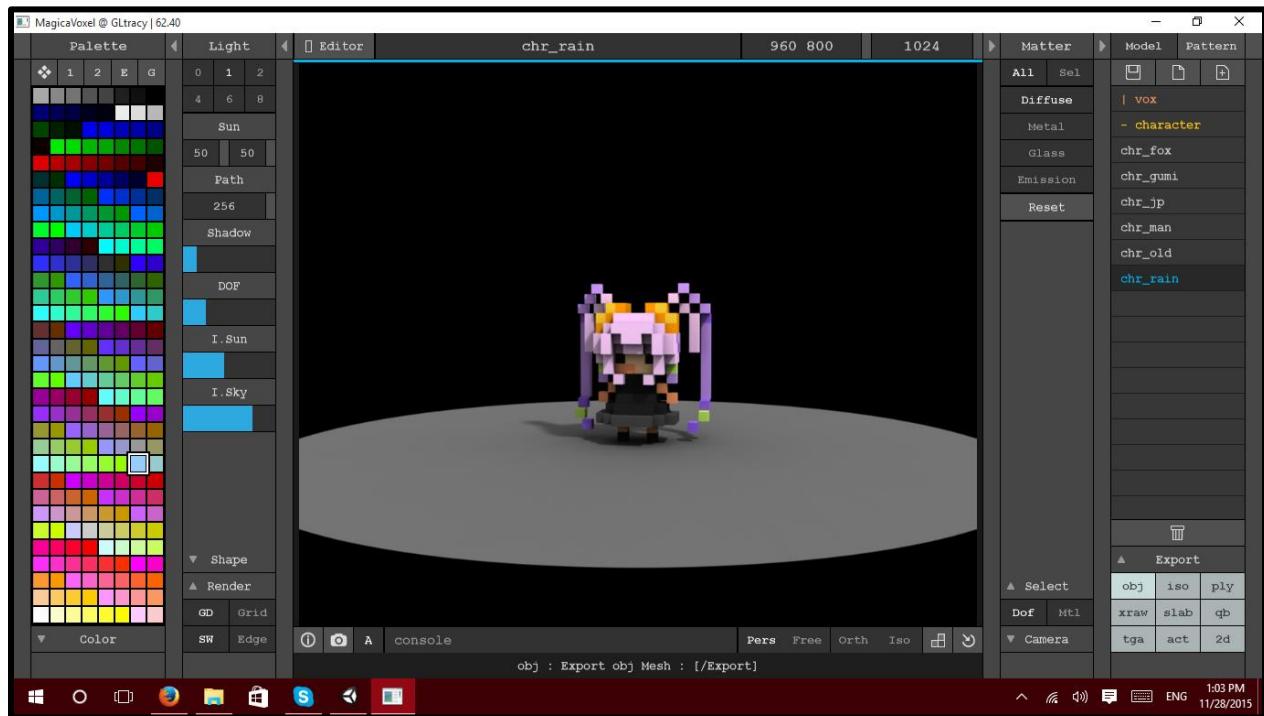
Inside of the images folder, create a folder called “How To”.



If you haven't already installed magicavoxel, go ahead and install it now. Next up, open up MagicaVoxel and create a background Model and character Model.

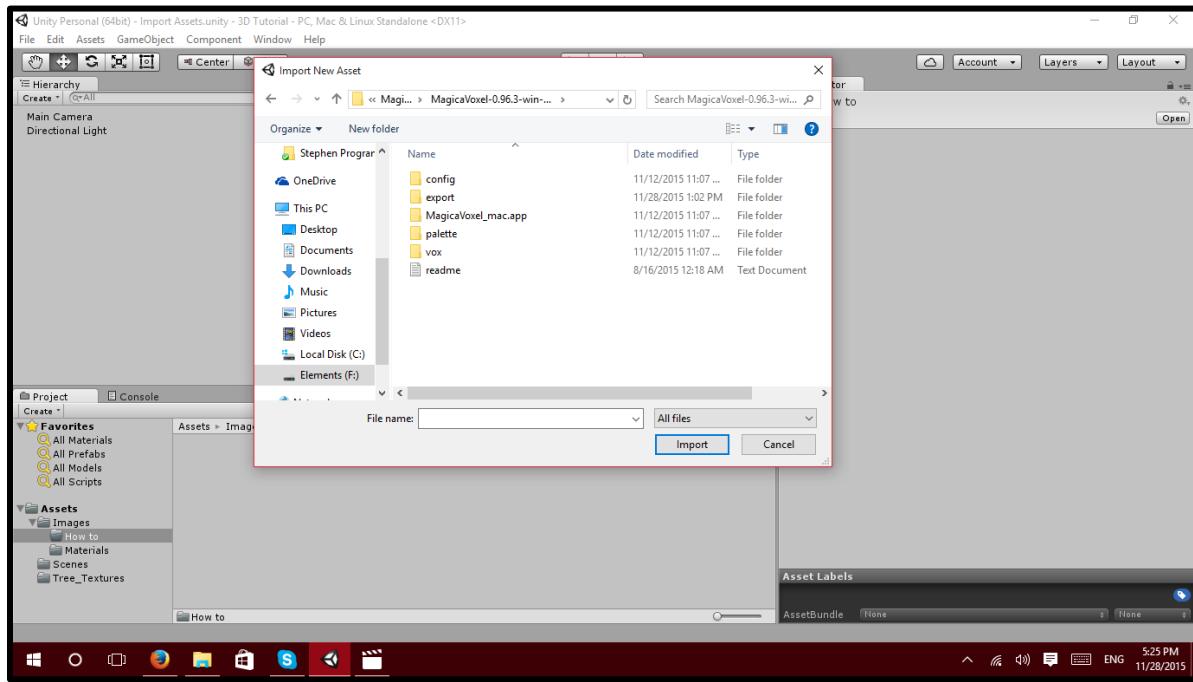


We now need to export the 3D Models. Unity3D works with many formats although a few have some limitations. The best ones to use are .OBJ and .FBX file formats. MagicaVoxel has the ability to export as .OBJ, Click on OBJ to Export.

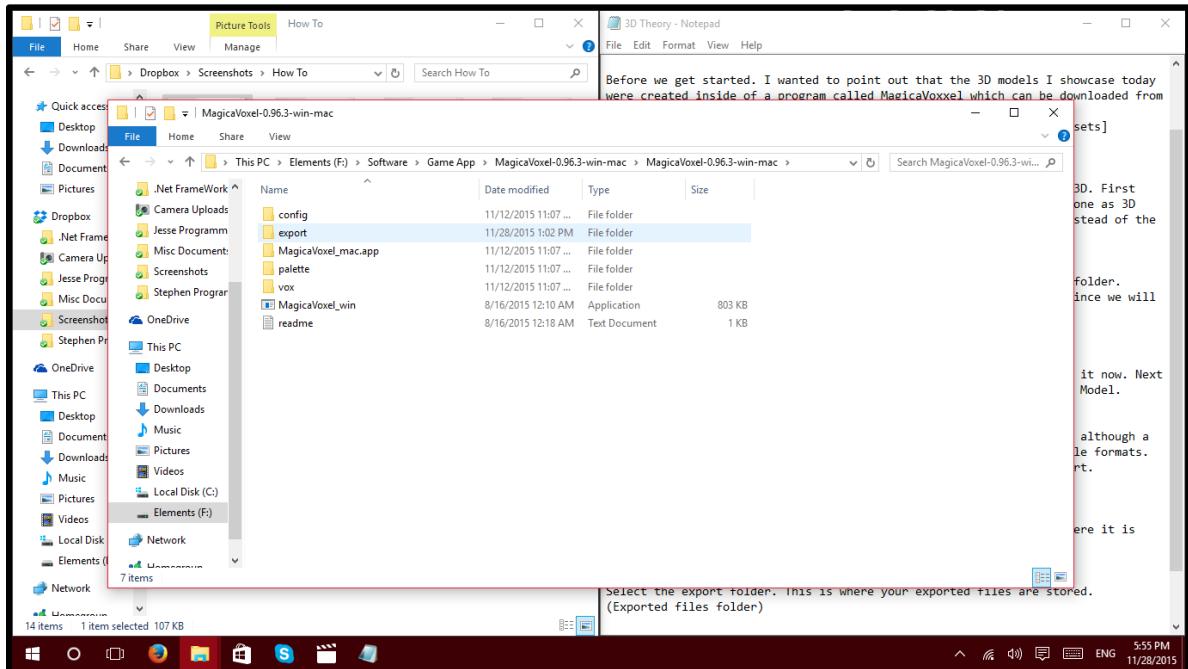


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Navigate to the folder where MagicaVoxel saves the files. [Note: the base location is always within the MagicaVoxel folder where it is executed from]



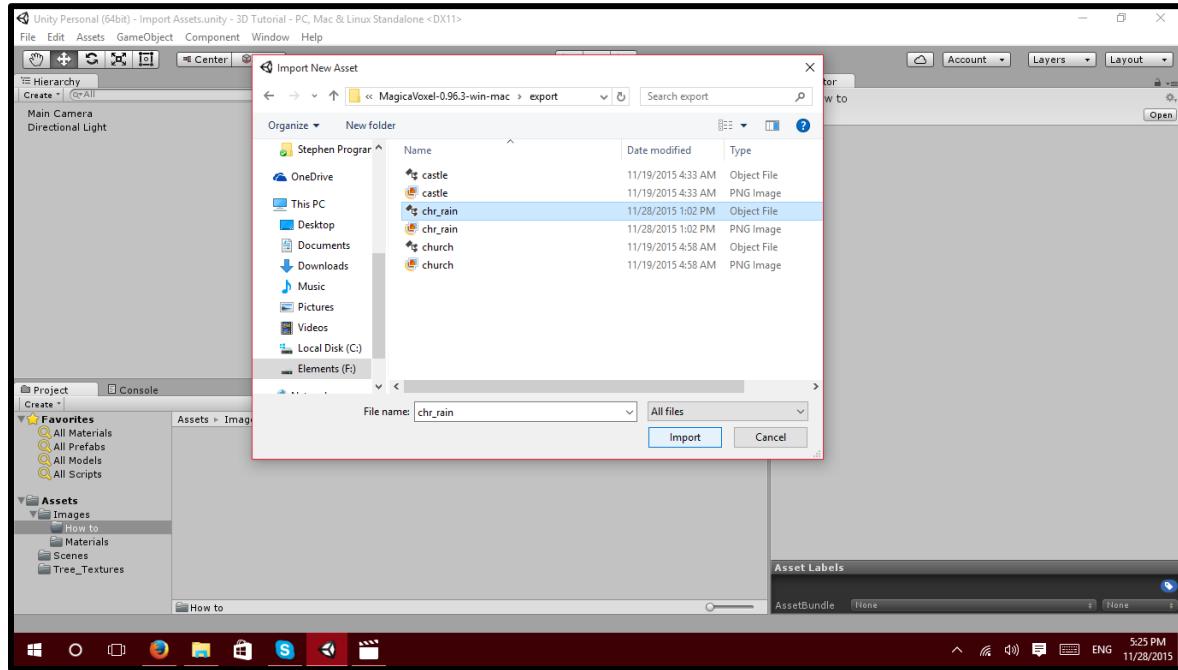
Select the export folder. This is where your exported files are stored.



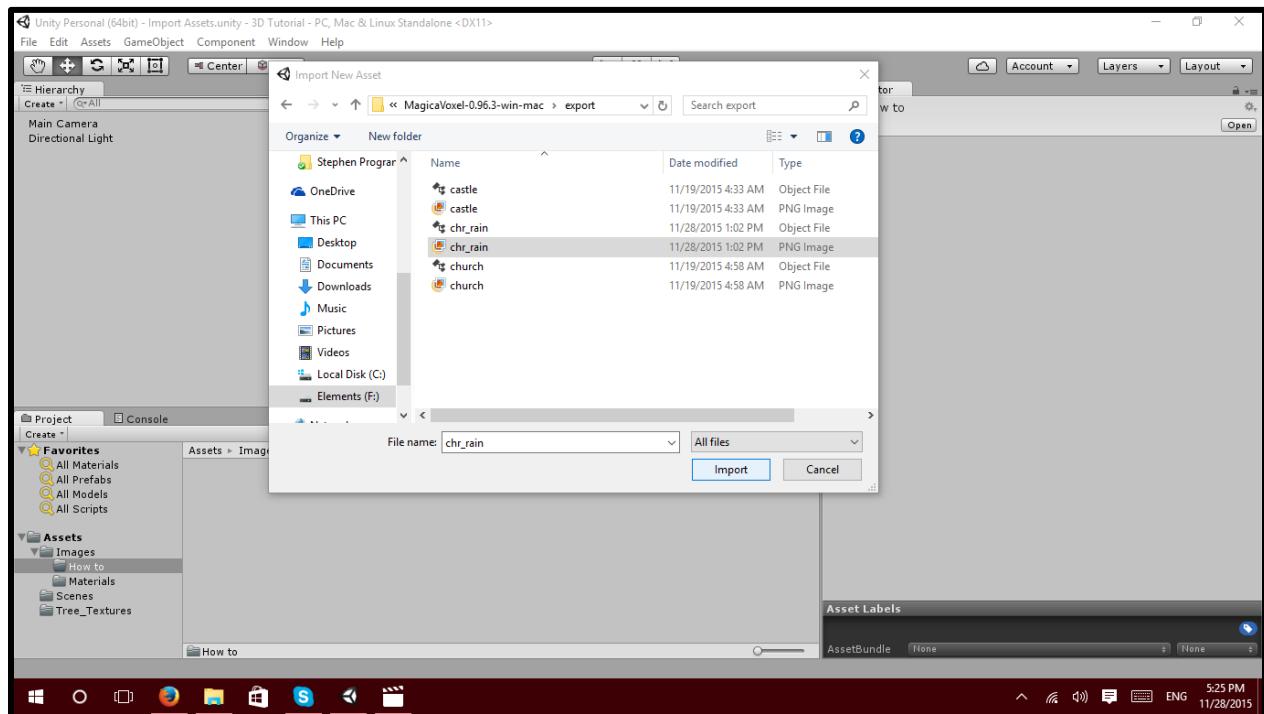
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Select the OBJ file and click import.



We need to do this for the PNG of the same name now.

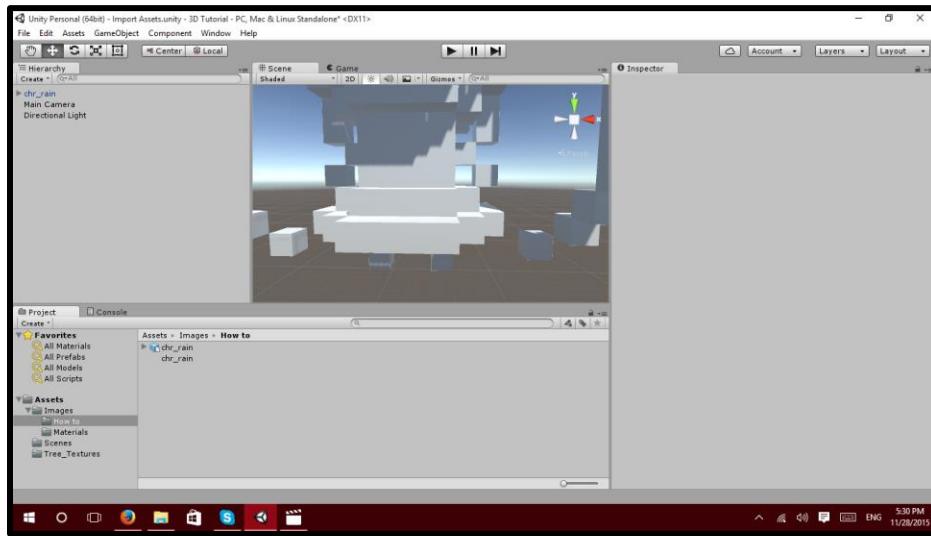


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

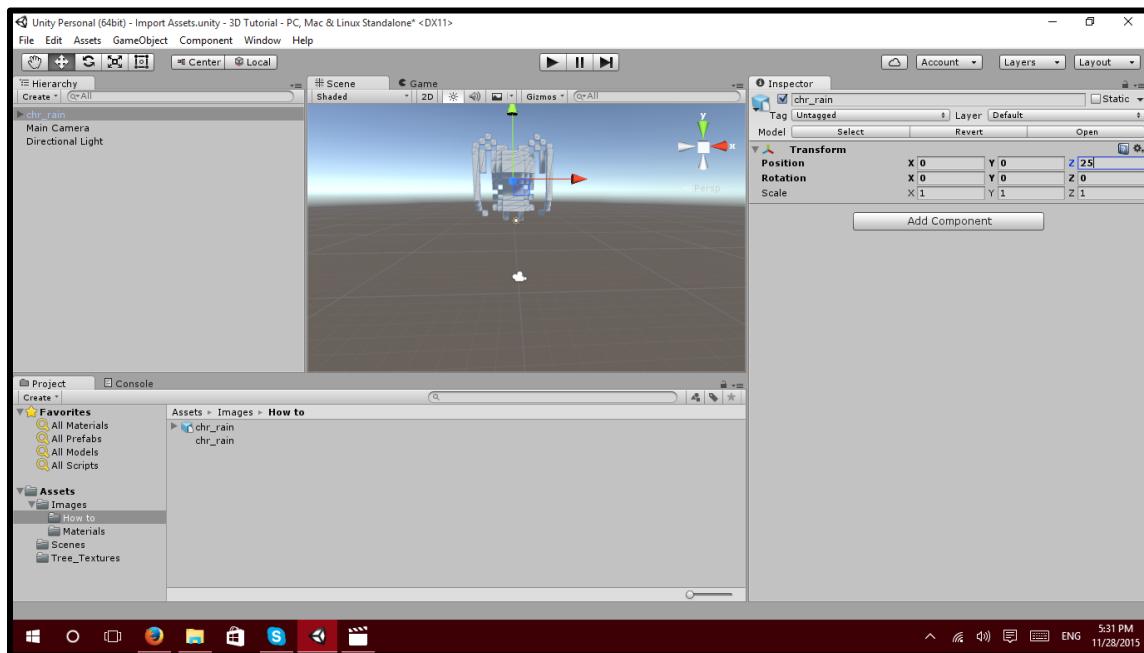
© Zenva Pty Ltd 2021. All rights reserved

You have now successfully imported a 3D model into Unity3D. Let's go further and display the 3D model inside of the Scene.

The OBJ file will have a blue square next to it, click and drag it onto the scene view.



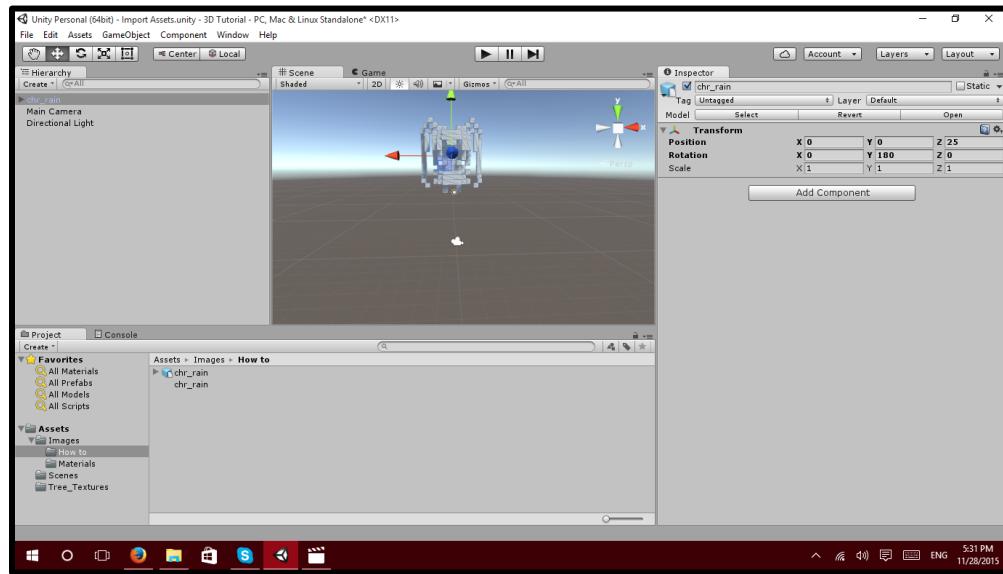
Notice in the Hierarchy Pane, you should now have a chr_rain listed with a minus down box. Click on chr_rain itself. We need to change the position of where she currently stands. We will set Position X to be 0, Position Y to be 0, and Position Z to be 25.



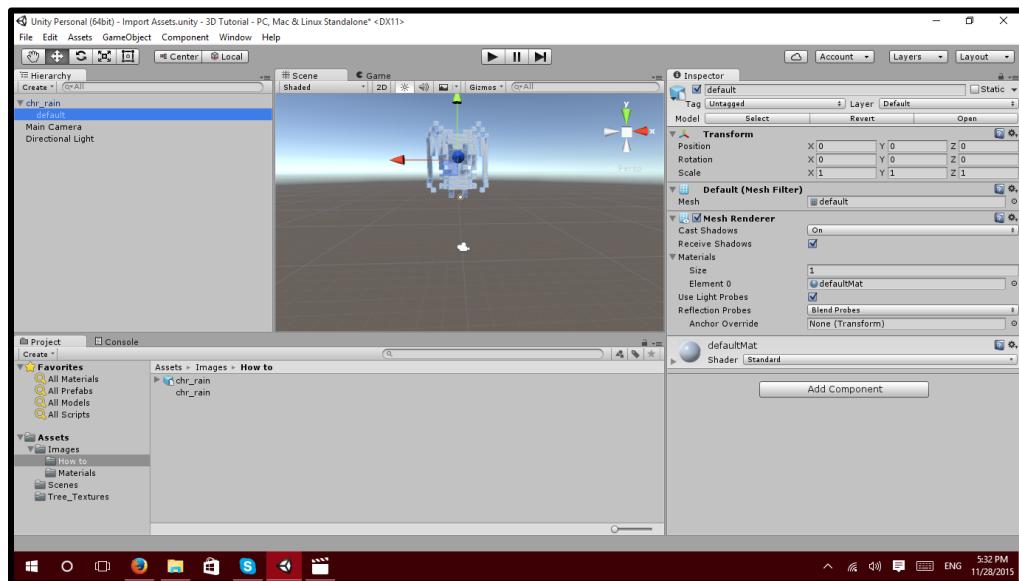
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Now, we should change the rotation of the model as well. We want to have her face us. Rotation X should be 0, Rotation Y should be 180, and Rotation Z should be 0.



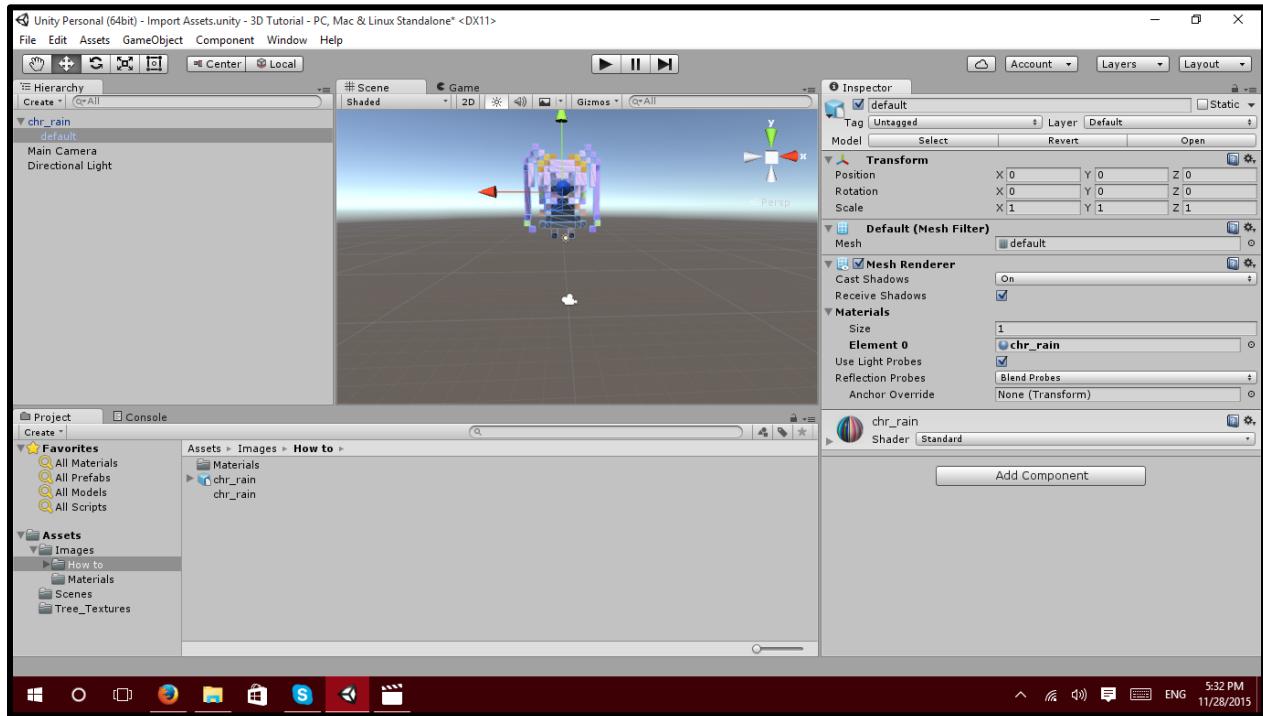
Something still looks off about this 3D model. In MagicaVoxel, she was coloured and in Unity3D she looks like a statue. Remember that PNG we also imported? That is her colour palette. Minus down the chr_rain model in the Hierarchy Pane and select the item inside, which should be called default. Click and drag the chr_rain png file and drag it underneath where it says add component.



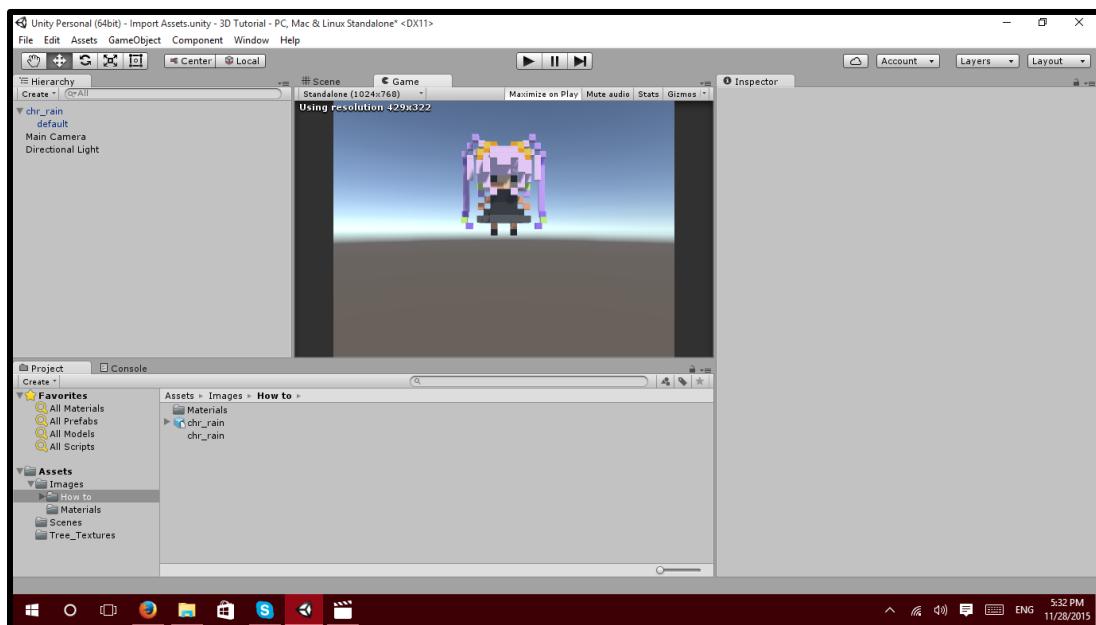
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

She now has her colour scheme on her. Now, we can click on the game view and see how she would look inside of a game.



You have now successfully imported and used a 3D model inside of Unity3D. This is the basis for 3D development within Unity3D.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Section 2: 3D Theory

Yes, it is time for some theory. This section will help you further understand what is actually happening behind the scenes in Unity3D.

Not only should you know how the Unity Editor handles the view change and how different components work in 3D space, you should also have a fair amount of knowledge of what the differences between 2D and 3D cameras are.

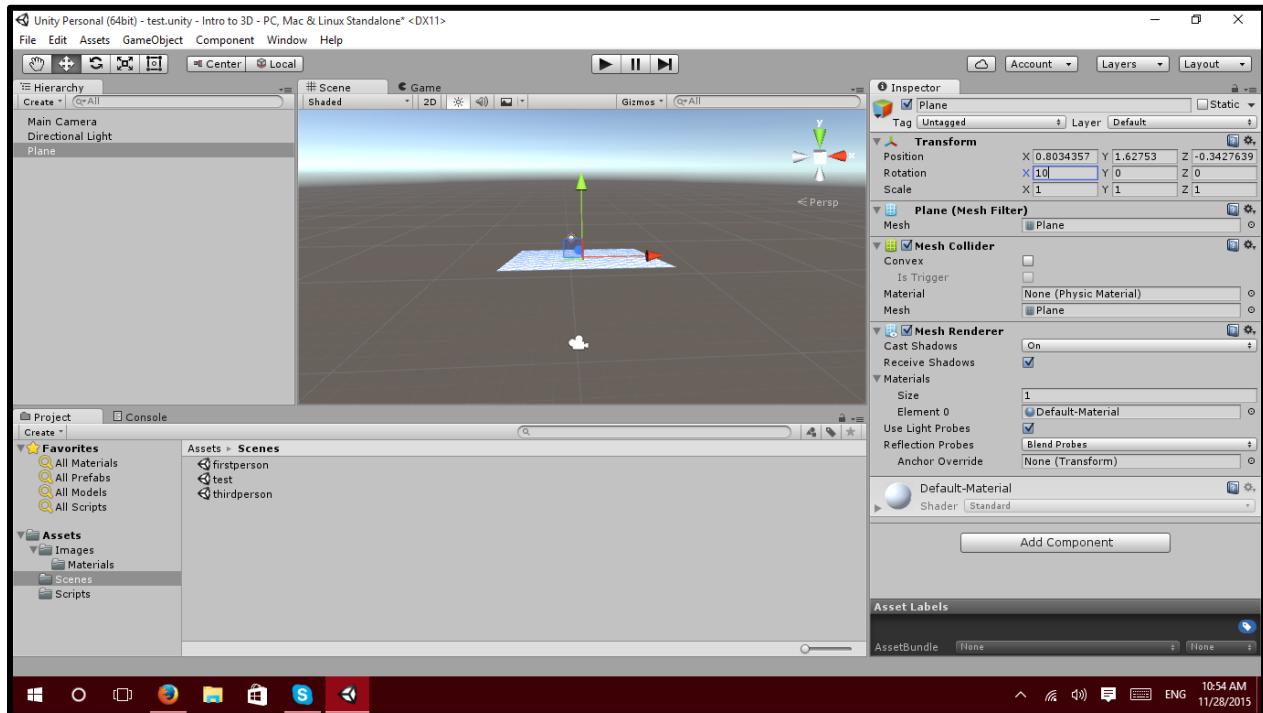
The differences in how the cameras are used in 3D vs 2D is quite possibly the most crucial piece of knowledge that anyone could impart on a budding 3D developer and thus, I feel compelled to start here.

A 2D camera typically uses X and Y coordinates. Most people innately know this fact, however, what people tend to forget about is the Z axis which is still present in 2D. The Z axis controls rotation. Pretty basic stuff here, but what does this have to do with 3D?

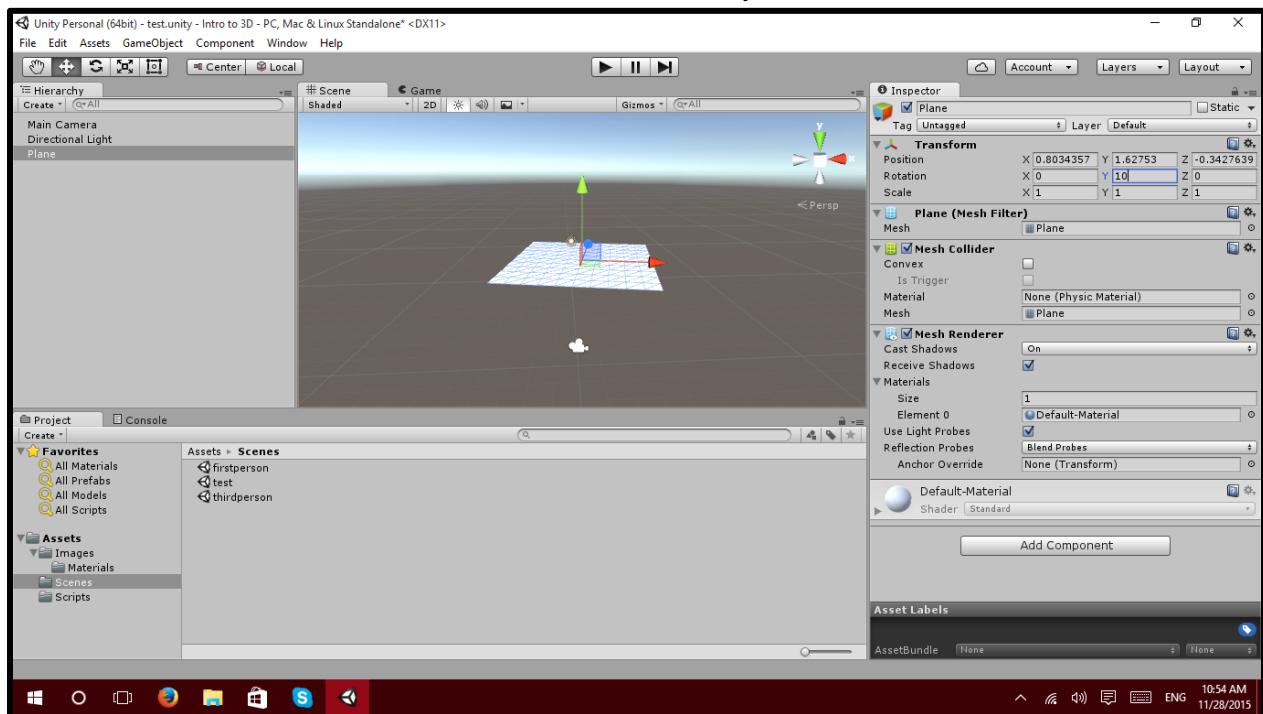
In a 3D camera, it uses the X, Y, and Z axis for quite a few things. X is for left and right movement, Y is for up and down, Z is used for forward and backward movement. To grossly oversimplify, X is horizontal, Y is vertical, Z is depth. However, there is also rotation on all 3 of these coordinate planes.

So, let's start with understanding how the rotation works on each axis.

X axis: tilts the object forward



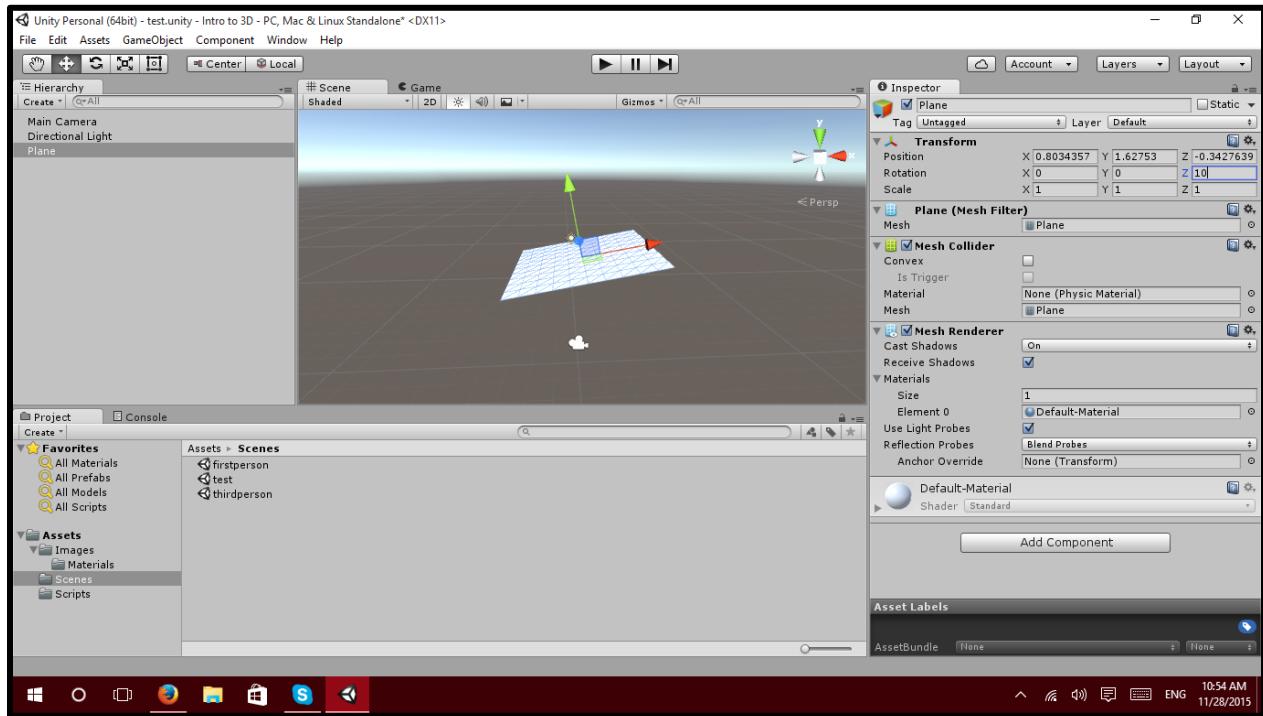
Y axis: rotates the object



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

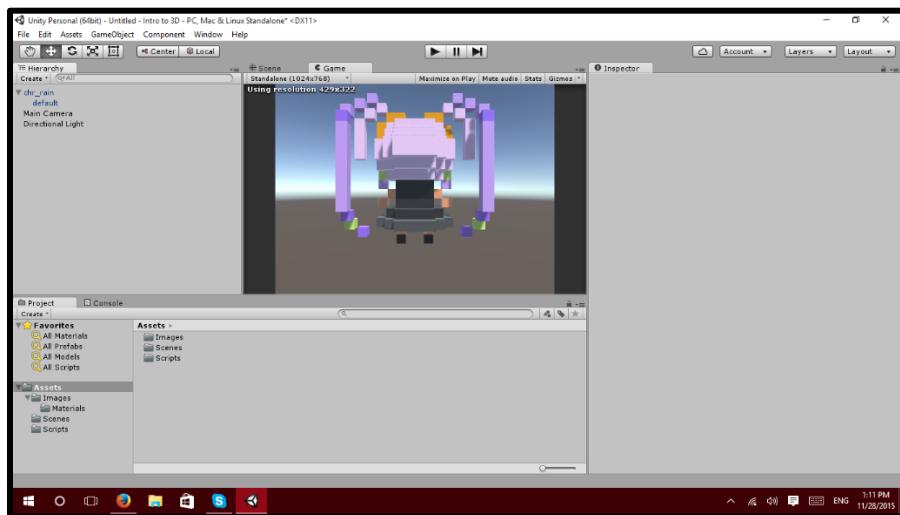
© Zenva Pty Ltd 2021. All rights reserved

Z axis: tilts the object sideways



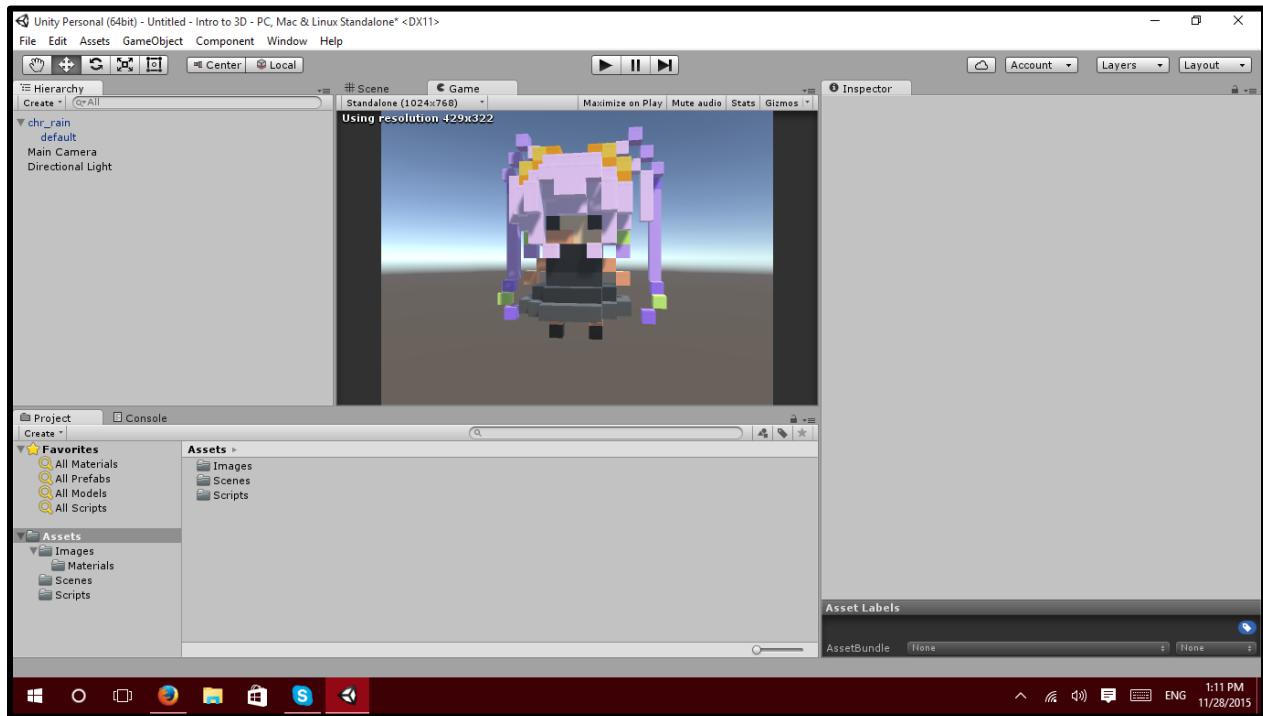
Why is knowing how the rotation works on the three axis' important?

As a programmer, you don't always get to choose how your objects or sprites come to you. Typically, the artist will make it for you and then you have to work with how they gave it to you. This isn't always a bad thing, but suppose the object you get (3D model) is facing away from the camera when you import it. You aren't going to waste time and have the artist redraw it or re-render it. You will just have to fix it in the scene and create a prefabrication of the fixed rotations.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

So, in this scenario, If you want the 3D model to face the camera; It is pretty simple to fix. You just rotate the model by 180 degrees on the X axis.



[Note: Unity stores world space rotations as Quaternions internally.]

This may spark the question, "What is a Quaternion and how is it different from degrees or radian?"

This question does not have a simple answer In fact, all of these elements have a bit of math involved. Instead of a math lesson, we will look at the definitions of each of these words.

Quaternion Definition: a complex number of the form $w + xi + yj + zk$, where w, x, y, z are real numbers and i, j, k are imaginary units that satisfy certain conditions.

Radian Definition: a unit of angle, equal to an angle at the center of a circle whose arc is equal in length to the radius.

Degree Definition: a unit of measurement of angles, one three-hundred-and-sixtieth of the circumference of a circle:

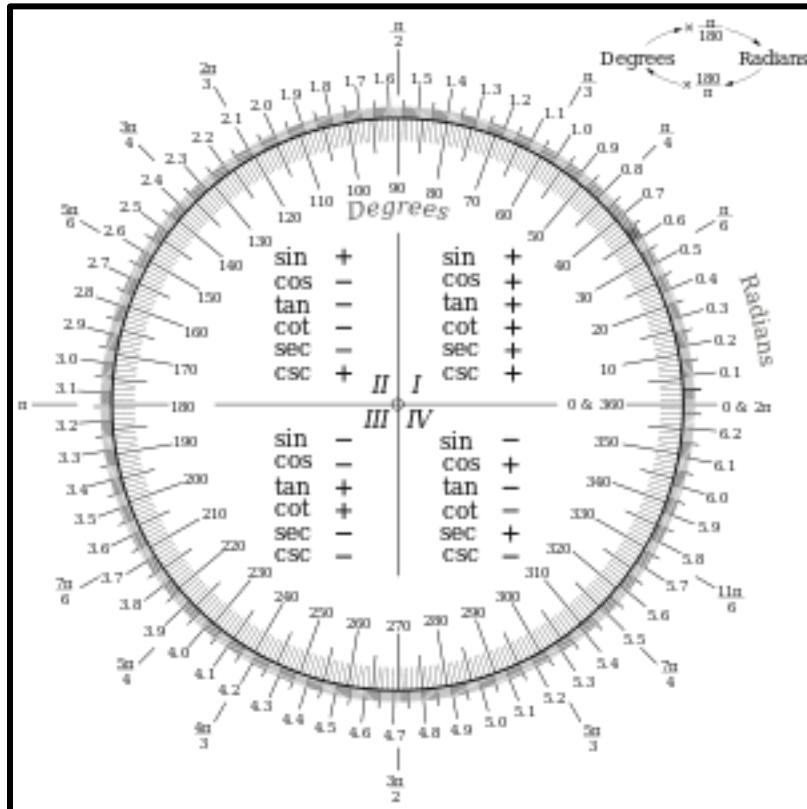
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

In mathematics, A radian is classified to be more accurate than Degrees, and they are both 1 dimensional. A single radian is equal to 57.2958 degrees.

However, you can easily convert between radians and degrees.

- To go from radians to degrees: multiply by 180, divide by π
- To go from degrees to radians: multiply by π , divide by 180
- To make things simpler, here is a conversion chart between degrees and radians.

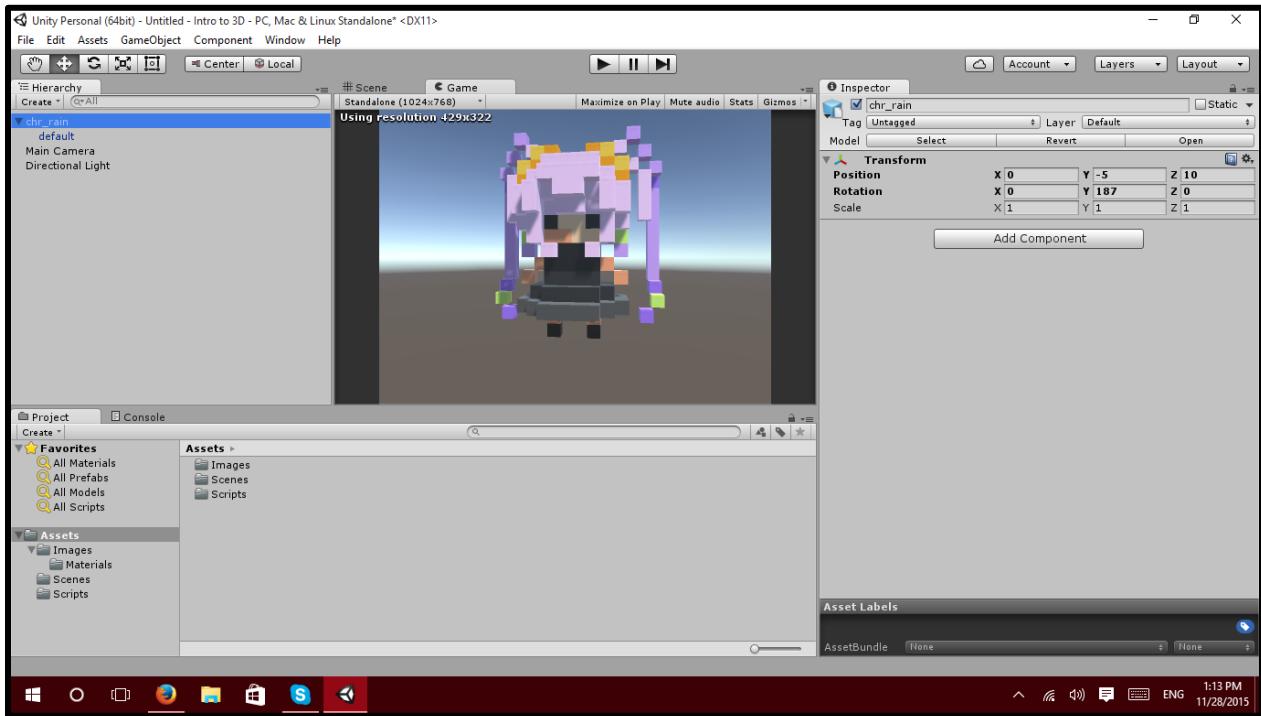


Now that degrees and radians are out of the way. Let's talk more about Quaternions.

You can easily describe a quaternion as 3D quadrant math. If you were to take a cube and divide it into 4 equal planes, you would have created essentially a quaternion out of it. If you are interested in the mathematics behind Quaternions, visit [here](#). He or she does a phenomenal job of explaining the math behind Quaternions.

How do I use Quaternions in Unity3D?

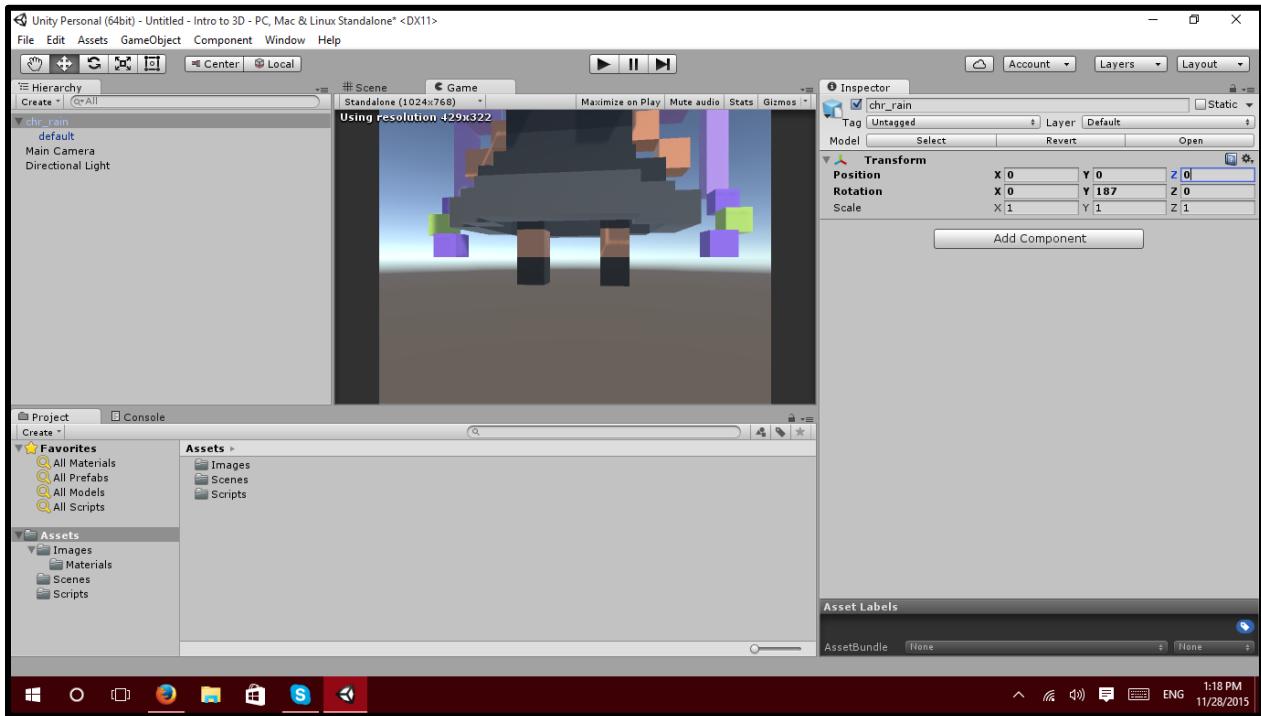
Great question! Let's look at the turned around voxel character in more detail to answer that question.



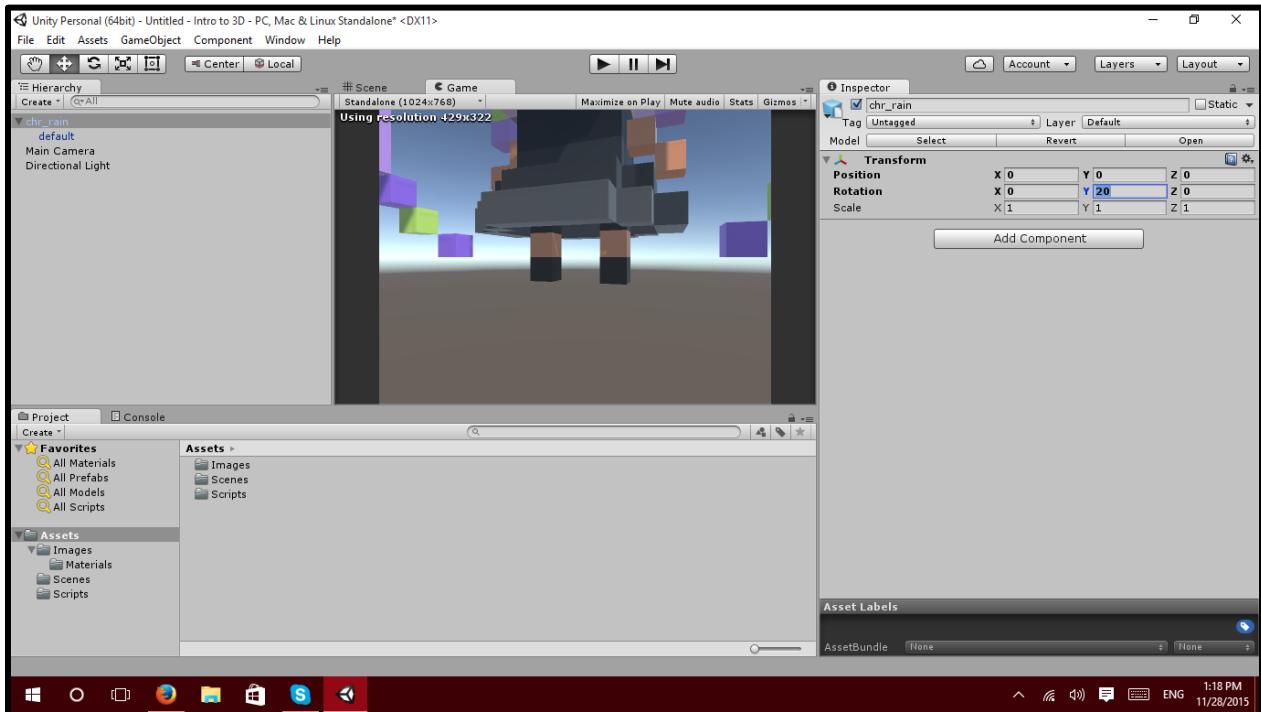
If you look at the Inspector of the voxel character that was imported on the screen, you will notice 3 sets of the X, Y, and Z axis. The first set is for Position. Second set is for Rotation.

Third set is for Scale. To go into greater detail, Position is referencing placement on the screen. These can be set as whole numbers or decimal numbers on both the scale of positive and negatives.

Position sets the actual location on screen of the model as shown below:



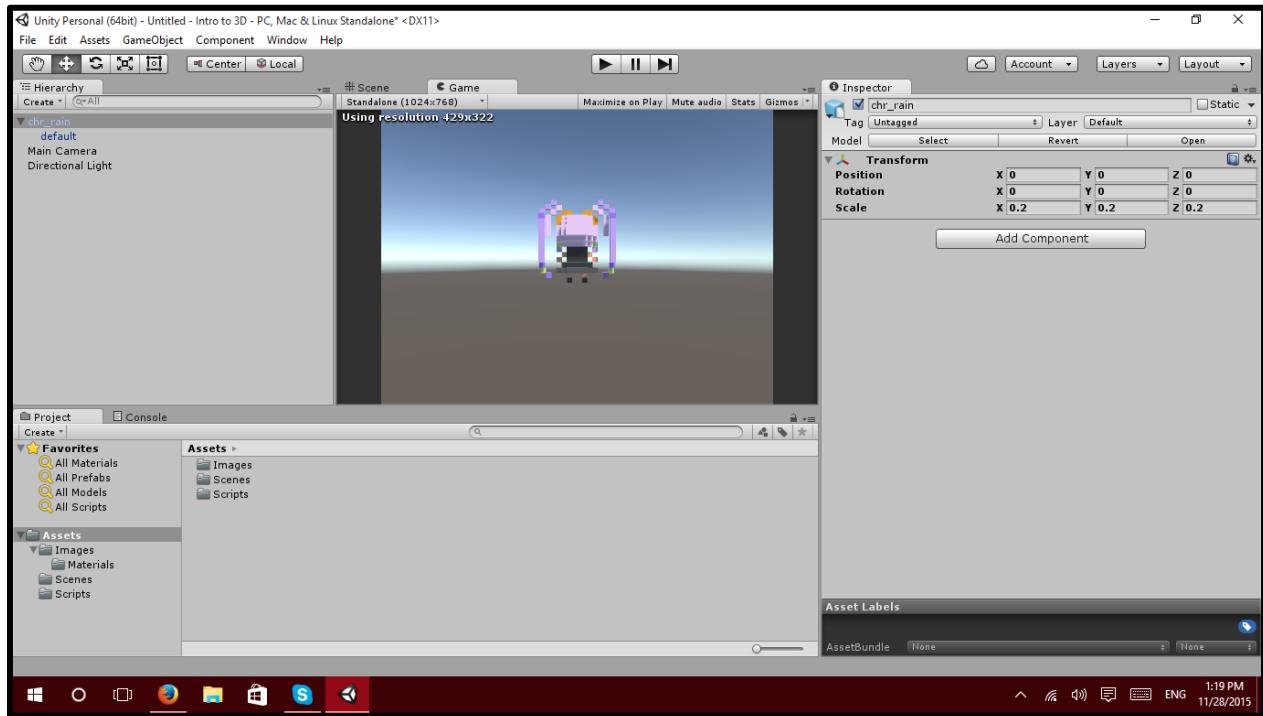
Rotation works the same way, although it only deals with where the model or texture is facing.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Scale also works in the same fashion, however, it deals with sizing of the model.



Hopefully you were able to understand the previous point without too many issues. Because now we are going to look at the Camera Component in Unity3D and look at the key difference between the camera component in 2D mode versus 3D mode.

Camera in 3D Mode:

Select the Camera in the Hierarchy Pane, right away the Scene view should show off the camera. It is giving a visual representation of the field of view of the 3D camera. Pretty neat right? Well, look over at the Inspector Pane and we will see in detail on what's different.

If we look at the projection property, we see it is set to perspective. [Note: If you change it to Orthographic, it will be in 2D mode]

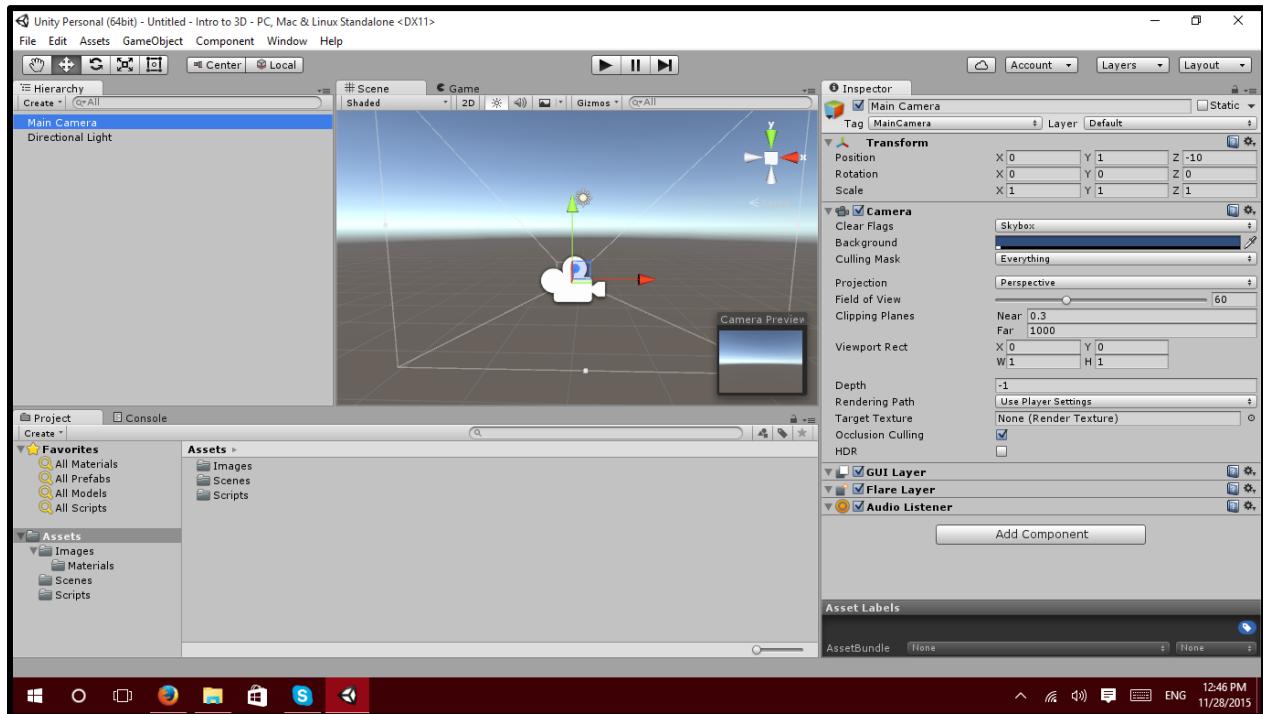
Field of view is defaulted to 60. Play around with it and you will see the camera zoom in or pan out.

Up next we have the Clipping Planes. It has the property of Near and Far. Play around with it if you'd like. Any object you have on the screen would display differently depending on what you

have set. If we change near to be say 5, it will start clipping out portions of the Voxel character. If we change the far to be 10, you will again see portions of the Voxel character be clipped.

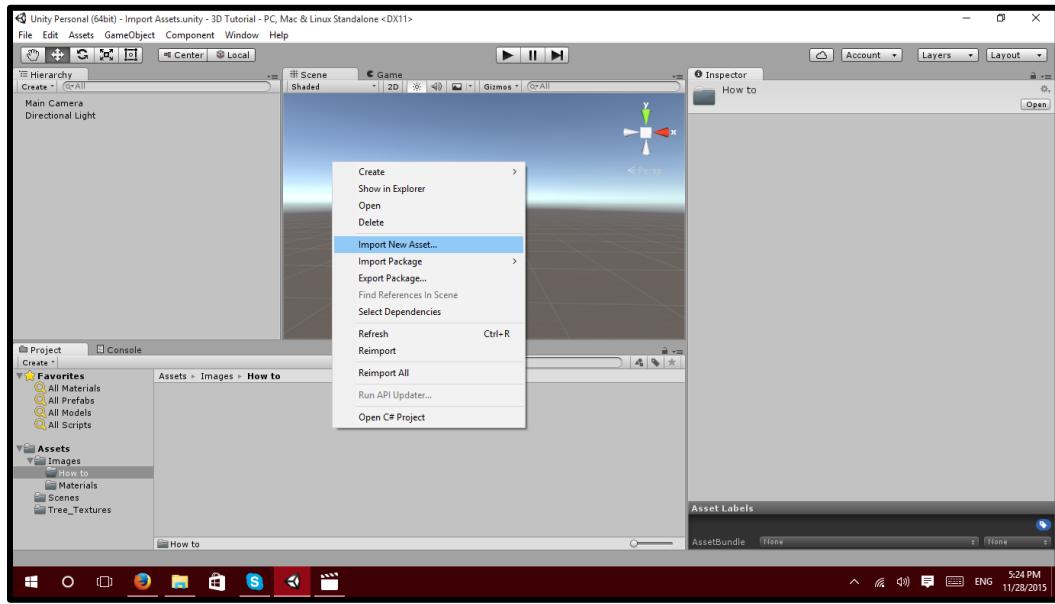
Essentially, Clipping Planes sets the distance the Camera will render something. Viewport Rect has four properties, X, Y, Width, and Height. This is useful when you have 2 cameras on the scene.

Depth by default is set to -1. This again is useful when you have multiple cameras on the scene.



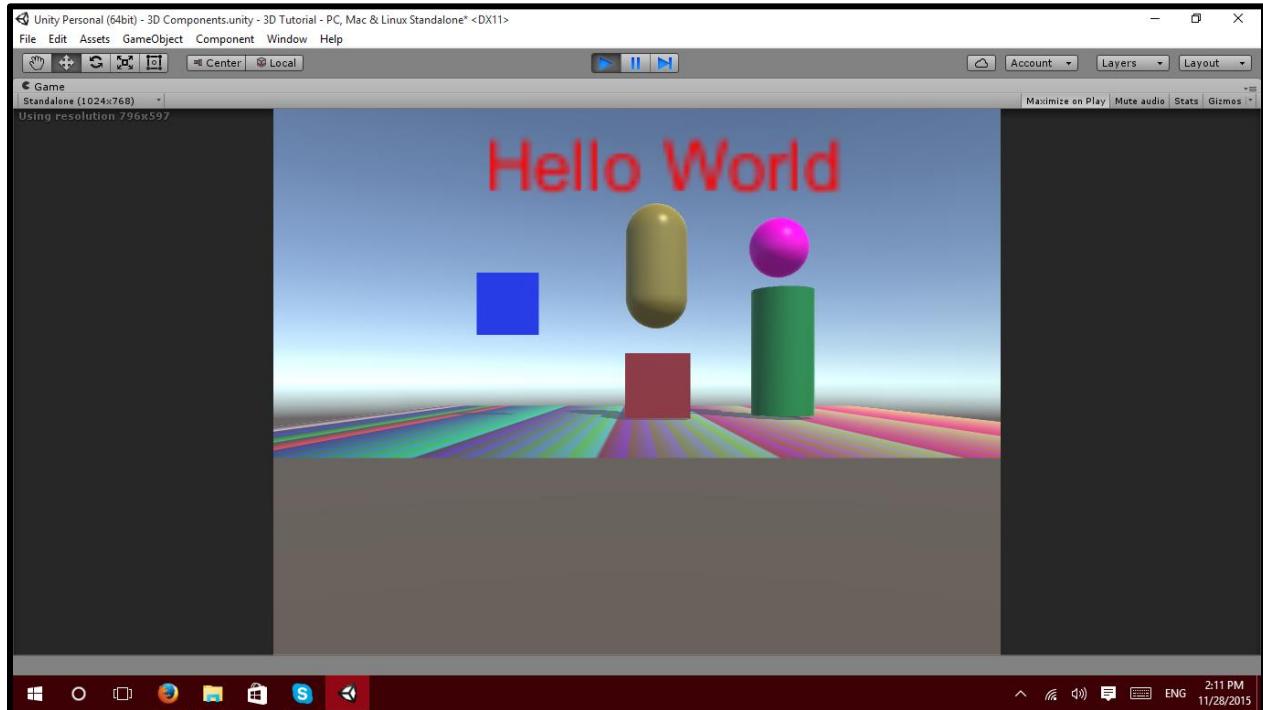
Various 3D Objects built into Unity3D:

Now we can look at the different 3D objects we can create from the Hierarchy Pane.



We have a Cube, Sphere, Capsule, Cylinder, Plane, Quad, Ragdoll, Terrain, Tree, Wind Zone, and 3D Text.

I know I don't need to explain what a Cube, Sphere, Capsule, Cylinder, Plane, 3D Text, or Quad is; However, Here is a screenshot that shows what they all look like.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

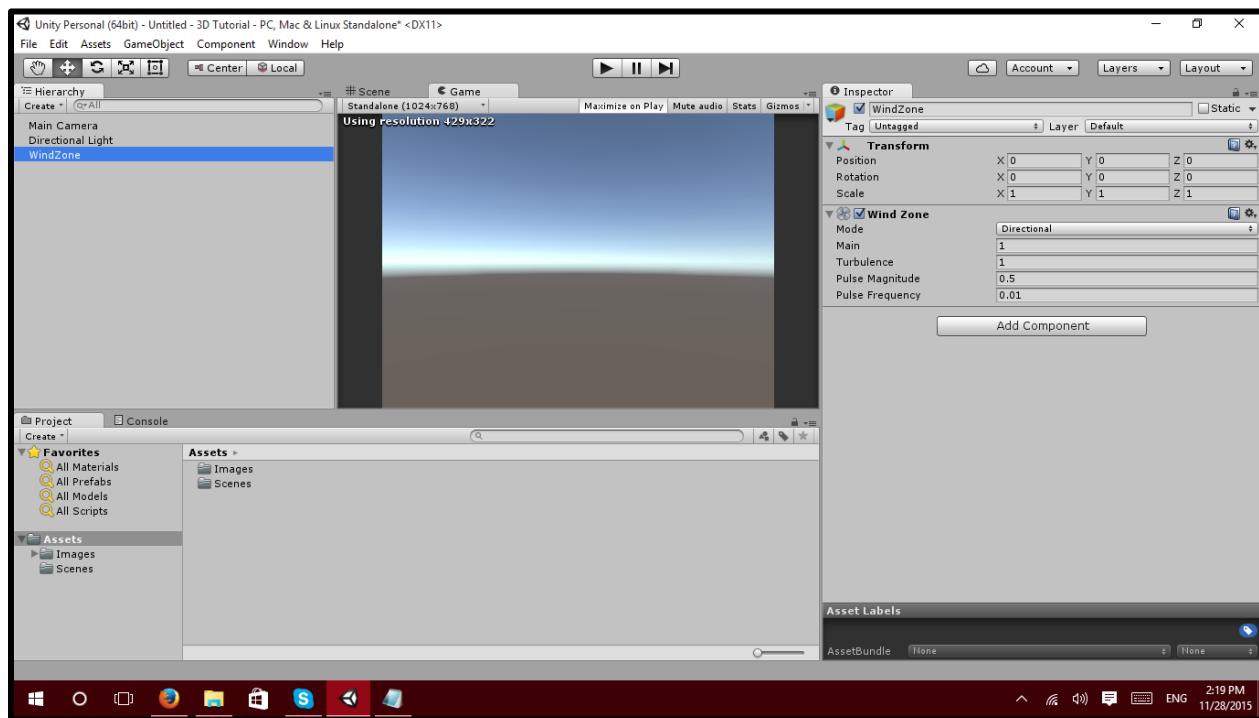
Ragdoll 3D object:

Ragdolls make use of Skinned Meshes, that is a character mesh rigged up with bones in the 3D modeling application. For this reason, you must build ragdoll characters in a 3D package like Maya or Cinema4D.

Sorry, No screenshots for this one. I don't currently have any skinned Mesh or Skeleton Models that I can use for it. However, this will be covered in a future tutorial that covers 3D animation techniques.

Wind Zone 3D object:

A wind zone object can be created directly (menu: GameObject > Create General > Wind Zone) or you can add the component to any suitable object already in the scene (menu: Component > Miscellaneous > Wind Zone). The inspector for the wind zone has a number of options to control its behaviour.



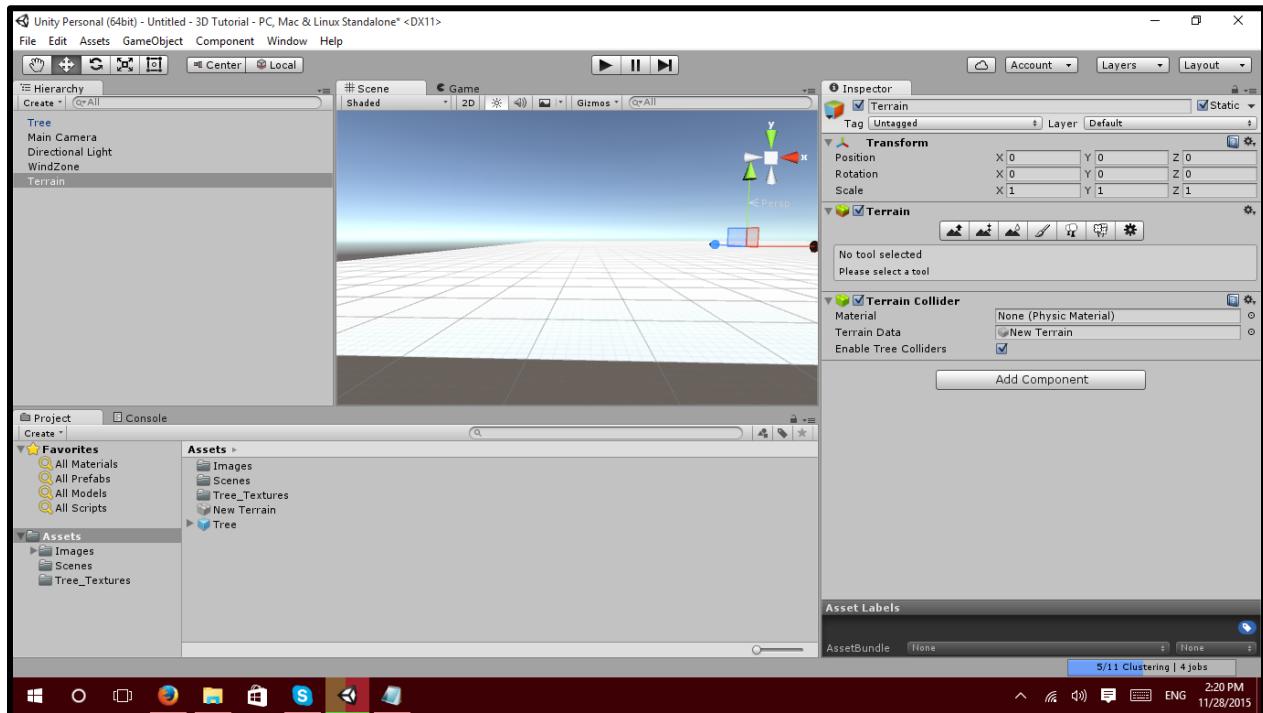
Mode can be set to Directional or Spherical. Directional mode will affect the whole terrain at once. Spherical will blow from within a sphere and defined by the radius property. [Note: Spherical mode is more suitable for special effects such as explosions.]

The Wind Main property is the overall strength of the wind. Wind turbulence gives a bit of a

random variation to Wind Main. Wind Pulse Magnitude is the strength of the Wind pulse and Wind Pulse Frequency is how often the Wind will pulse.

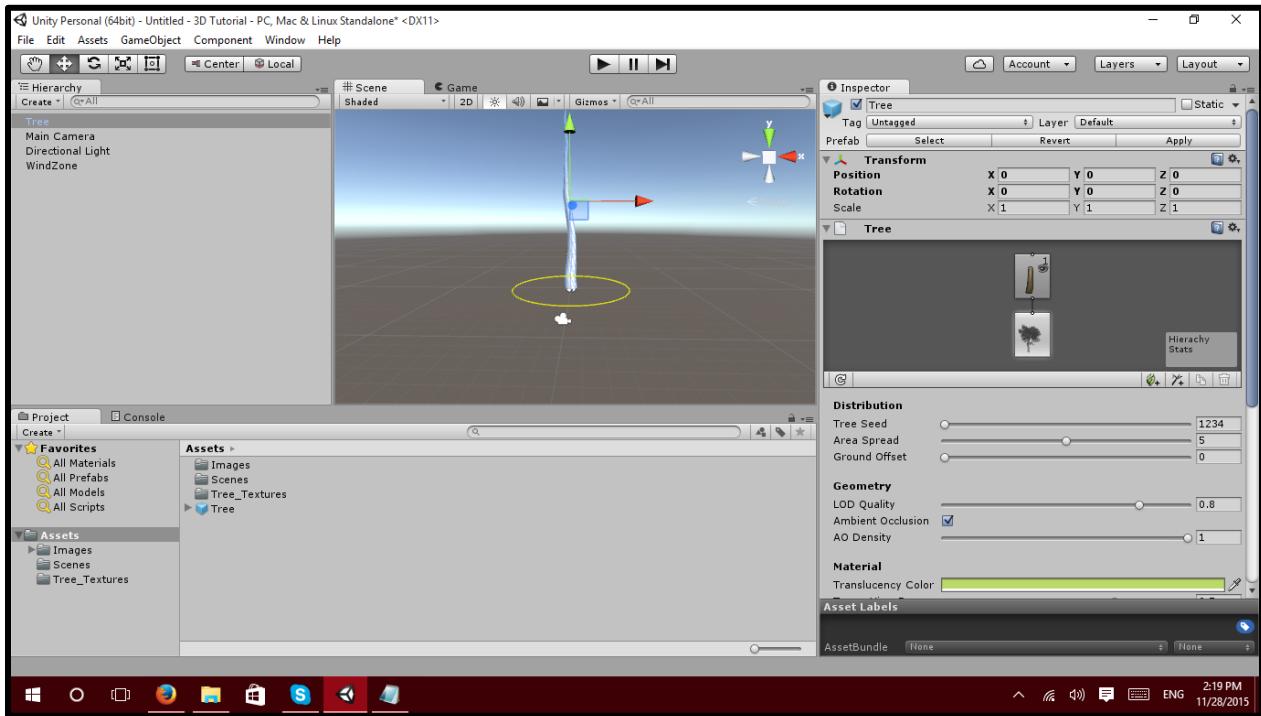
Terrain 3D object:

The Terrain is the landscape of the scene you are creating. This feature is extremely robust and will require a lesson of its own to fully explain.



Tree 3D object:

The tree component allows you to “paint” trees onto the Terrain. Again, this will be fully explained in its own tutorial along with the terrain in the future.



I hope this introductory lesson in the 3D side of Unity3D was beneficial for you. There are plenty more 3D tutorials on the way. If you have any questions, comments, or feedback on this tutorial; Leave a comment below. Thanks for reading and “May your code be robust and bug free!”.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Unity 3D First and Third Person View Tutorial

By Jesse Glover

Now that we have gone through the fundamentals of 3D development with Unity3D, we can now cover the basics. The basics will cover first person view, third person view, and collision detection. As usual, my tutorials will be broken down into several sections. The first section will cover what first person view is and how to achieve it using code. The second section will cover what third person view is and how to achieve it using code. The third and final section is all about collision detection in 3D.

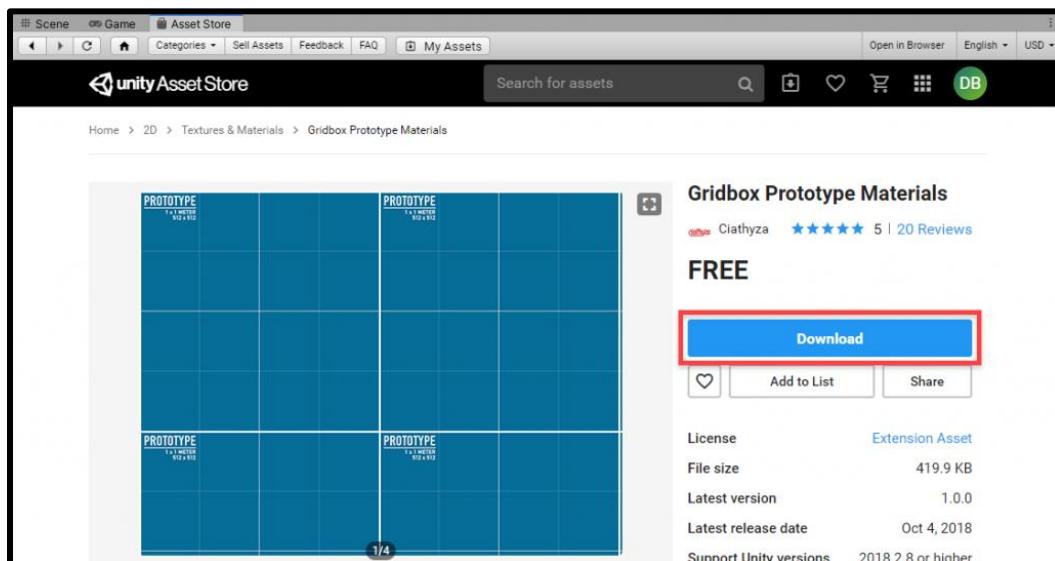
You can download the complete Unity project for this tutorial [here](#).

Section 1: First Person View

First person view is not as complicated as you might think. Rather, it is actually super simple. First person view means that the player IS the camera, or to simplify, The camera IS the player. The implications of this thought process simplifies everything as a designer and as a programmer.

So, how do we begin to utilize this knowledge within Unity3D? We could start by building a 3D scene and setting up the scene for testing. I think that would probably be best considering if we do too much for a basic course, things could get very confusing.

To begin, create a new Unity project and navigate to the Asset Store window (*Window > Asset Store*). Here, we want to search for and download the Grid Prototype Materials asset.



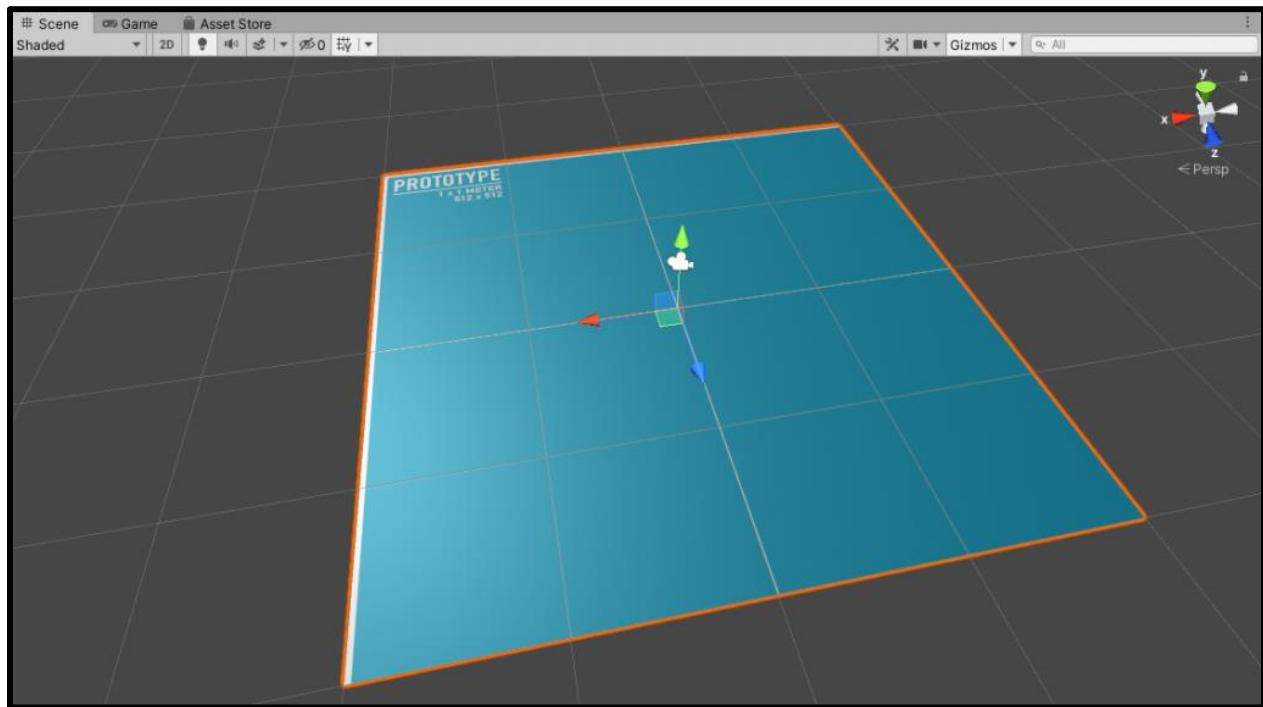
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Now that the Thirdparty folder has been added to your project, go ahead and add the Scene and Scripts folder. Also create a C# script called FirstPersonCamera.

Create a new scene called FirstPerson and create a new plane for the ground.

- Set the Scale to 2
- Assign the Prototype_512x512_Blue1 material

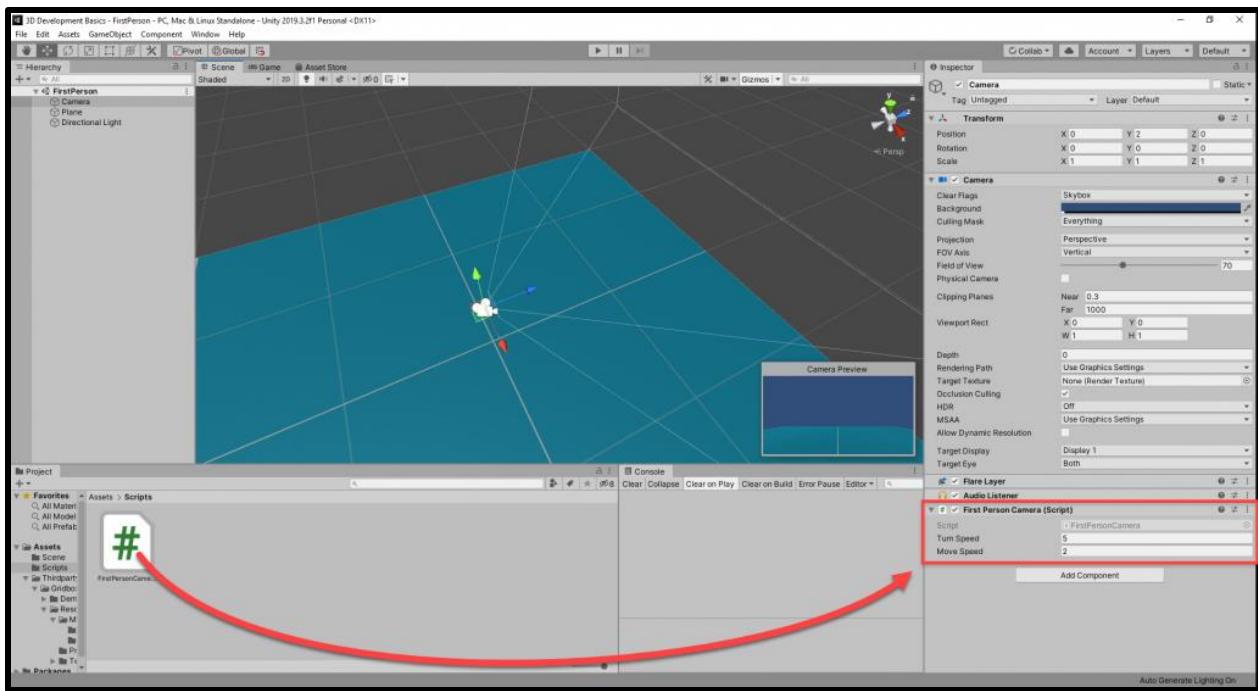


We can now move on to the scripting side. I have tried to do an extremely basic approach to scripting to allow the camera to move with the event of using the mouse and keyboard. Since we plan on attaching the script directly to the camera, we can use the this keyword or omit the this keyword and just write the transform.

```

1 public float turnSpeed = 4.0f;
2 public float moveSpeed = 2.0f;
3
4 public float minTurnAngle = -90.0f;
5 public float maxTurnAngle = 90.0f;
6 private float rotX;
7
8 void Update ()
9 {
10     MouseAiming();
11     KeyboardMovement();
12 }
13
14 void MouseAiming ()
15 {
16     // get the mouse inputs
17     float y = Input.GetAxis("Mouse X") * turnSpeed;
18     rotX += Input.GetAxis("Mouse Y") * turnSpeed;
19
20     // clamp the vertical rotation
21     rotX = Mathf.Clamp(rotX, minTurnAngle, maxTurnAngle);
22
23     // rotate the camera
24     transform.eulerAngles = new Vector3(-rotX, transform.eulerAngles.y + y, 0);
25 }
26
27 void KeyboardMovement ()
28 {
29     Vector3 dir = new Vector3(0, 0, 0);
30
31     dir.x = Input.GetAxis("Horizontal");
32     dir.z = Input.GetAxis("Vertical");
33
34     transform.Translate(dir * moveSpeed * Time.deltaTime);
35 }
```

Now, we can attach the script to the camera by dragging and dropping the first person script onto the camera in the Inspector Pane.



Save the scene in the Scene folder under the name First Person. To do this, click File, Save Scene As.

Now you can go ahead and run the scene. Voila, you now have a scene where the player is the camera. This is the overall basis for a first person view game.

Section 2: Third Person Perspective

It is extremely easy to over complicate the thought process for a third person perspective game and with good reason. My first assumption was to have the camera follow the player similar to the fashion that would occur in a 2D platformer. But, in reality, that doesn't make sense when you think about it from a 3D perspective. So, what does make sense and work?

It can be as simplistic or complicated as you want it to be. We should go over a few examples. There are various methods that you can use for third person perspective.

One method is taking the first person view camera and shifting it over the shoulder or behind the player model. This method's concerns are where the character is facing and collision detection.

Another method is the overhead view. The overhead view does not care whether the player is facing the camera or not.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

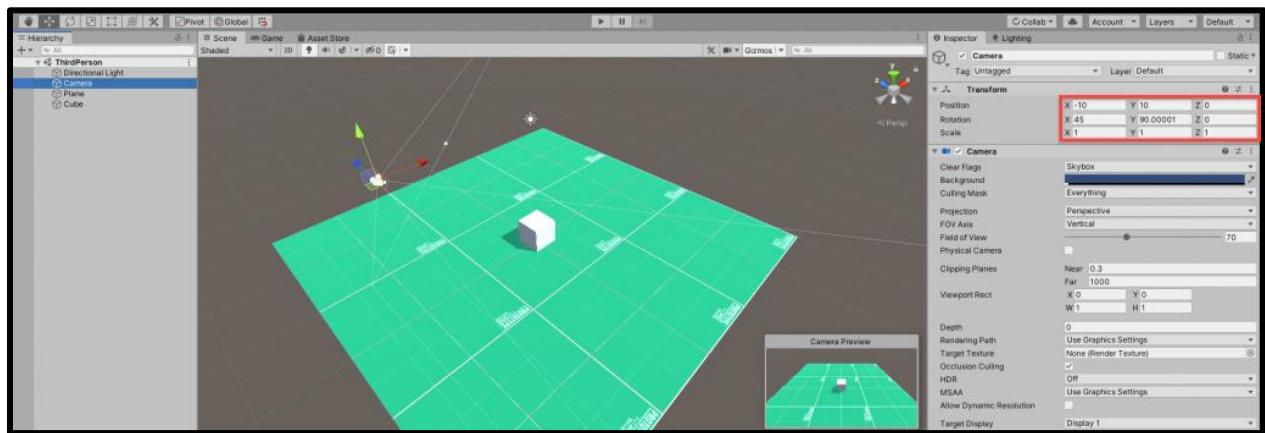
A third method is a fixed angle camera. It doesn't care where the player faces, it only cares about whether or not the player is in the room or not.

The last method I want to mention is a CCTV camera. Think about this as a fixed angle camera that acts like a surveillance camera with a TV showing the player what's going on. This method only cares about whether or not the player is in the room.

There are plenty of other third person view cameras that could be utilized. But the ones listed should give you a base idea of the difference between Third Person and First Person perspectives in addition to triggering some memories of games you have played in the past or currently, and make you think about how they decided to go with that particular game's perspective.

So let's create a new scene and set it up in a similar way to before.

- Create a cube and position it in the middle of the scene
- Move the camera to be in a third person perspective of the cube

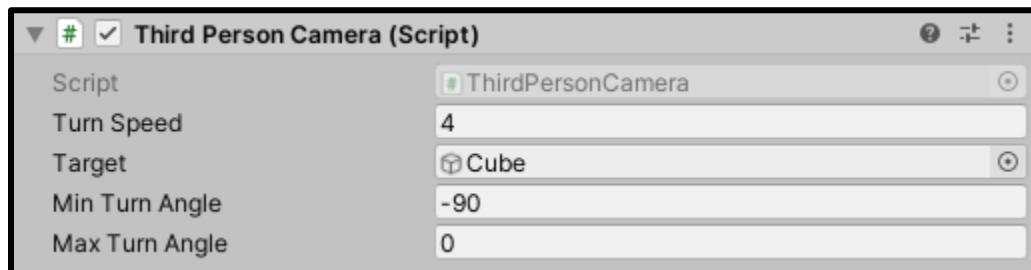


Create a new script called `ThirdPersonCamera` and attach it to the camera.

```

1 public float turnSpeed = 4.0f;
2
3 public GameObject target;
4 private float targetDistance;
5
6 public float minTurnAngle = -90.0f;
7 public float maxTurnAngle = 0.0f;
8 private float rotX;
9
10 void Start ()
11 {
12     targetDistance = Vector3.Distance(transform.position, target.transform.position);
13 }
14
15 void Update ()
16 {
17     // get the mouse inputs
18     float y = Input.GetAxis("Mouse X") * turnSpeed;
19     rotX += Input.GetAxis("Mouse Y") * turnSpeed;
20
21     // clamp the vertical rotation
22     rotX = Mathf.Clamp(rotX, minTurnAngle, maxTurnAngle);
23
24     // rotate the camera
25     transform.eulerAngles = new Vector3(-rotX, transform.eulerAngles.y + y, 0);
26
27     // move the camera position
28     transform.position = target.transform.position - (transform.forward * targetDistance);
29 }
```

Drag the third Person Camera Script onto the Camera. Then the target property should have the cube component dragged onto it.



If you run the game, you should be able to move the mouse in order to orbit around the cube.

Section 3: Collisions

Collision detection has been made a lot easier with Unity3D and other game development engines. What makes collision detection easier within Unity3D is the options available to you. You have Box colliders, mesh colliders, Rigidbody, sphere collider, capsule collider, wheel collider, terrain collider, cloth, hinge joint, fixed joint, spring joint, character joint, configurable joint, and constant force. These are the ones for 3D alone.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

If you want to make a 2.5D or 3D game that also has 2D elements, there are more options available. Rigidbody 2D, Circle Collider 2D, Box Collider 2D, Edge Collider 2D, Polygon Collider 2D, Spring Joint 2D, Distance Joint 2D, Hinge Joint 2D, Slider Joint 2D, Wheel Joint 2D, Constant Force 2D, Area Effector 2D, Point Effector 2D, Platform Effector 2D, and Surface Effector 2D.

Each one of these options has its own particular use and whilst we won't go over each and every one, I will be sure to cover the ones that you will mainly use in basic game development.

The main ones that you will use in a basic 3D game is the Box colliders, Mesh Colliders, Rigidbody, Sphere Collider, and Terrain. I feel like we should talk about these.

Box Colliders

A Box Collider is a basic cube-shaped collision primitive. In other words, it is shaped just like a cube would be. So basic uses for it would be a chest, floor, walls, or if you wanted to use as much processing power you could, each individual part of a 3D model.

Mesh Colliders

A Mesh Collider builds a collider based on the Mesh Asset from a 3D model. It is one of the most accurate collision detection methods and has the ability to collide with other mesh colliders if you choose.

Rigidbody

One of the most used and recognizable collider types. Rigidbody colliders allow you to enable GameObjects to act under the laws of physics within a game environment. I should also note that it can interact with objects through the NVIDIA PhysX engine as well.

Sphere Collider

Sphere Colliders are a basic sphere shaped collision primitive, in other words, shaped like a sphere. These can be used on balls, fist, heads, and et cetera.

Terrain

The Terrain Collider creates a collision surface that holds the same shape as the Terrain object it is attached to.

As I have shown above, there are plenty more physics objects that are used from within Unity3D. Later tutorials will cover these as needed.

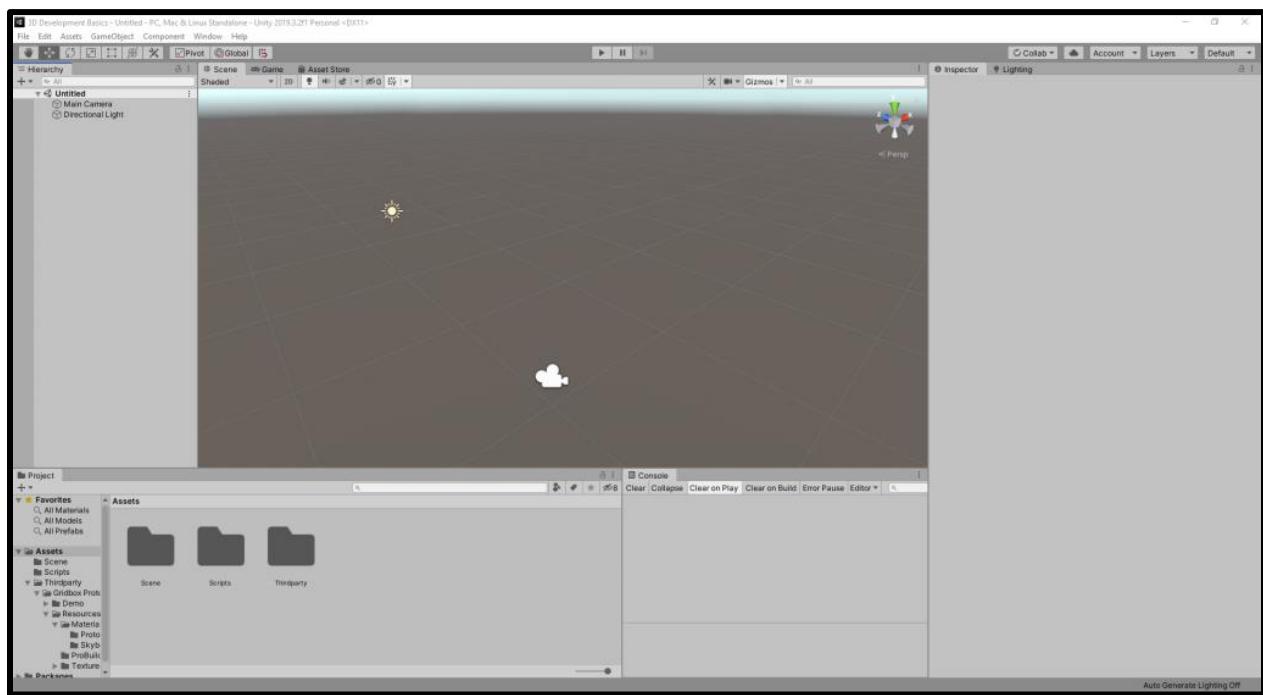
Each of the different physics options have a script attached to them that you cannot access or modify. However, when needed, you can create a controller script that can handle collisions on your own terms if you so choose.

Now that we have gone over some of the basic physics objects to handle collision, it is time for us to actually implement this in a scene.

This section of the tutorial is extremely short and very cool to say the least. Because we are using built in objects within Unity3D, we don't have to do any scripting whatsoever.

Basic Collisions:

To start off, we make a brand new scene.

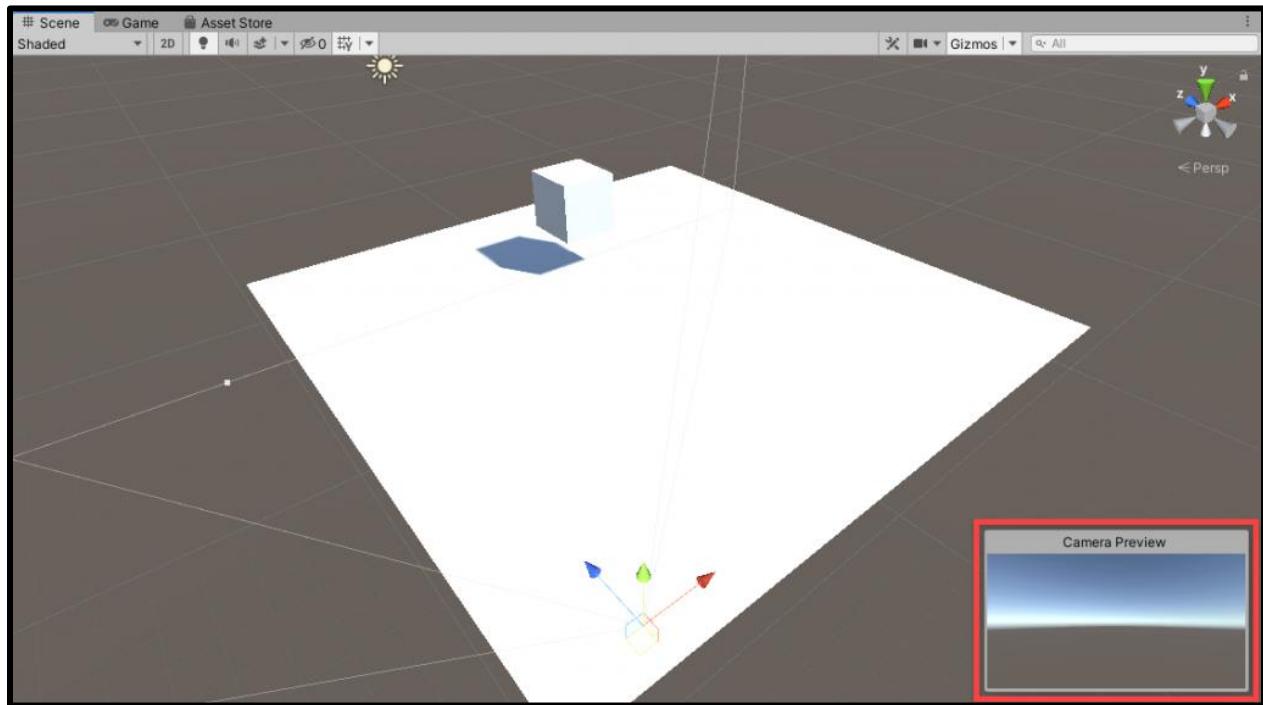


Click on the camera and take note of the X,Y, and Z positions of it. These figures are immensely important.



- Right click on the Hierarchy Pane, highlight 3D object, and select Plane.
- Right click on the Hierarchy Pane again, highlight 3D object, and select Cube.

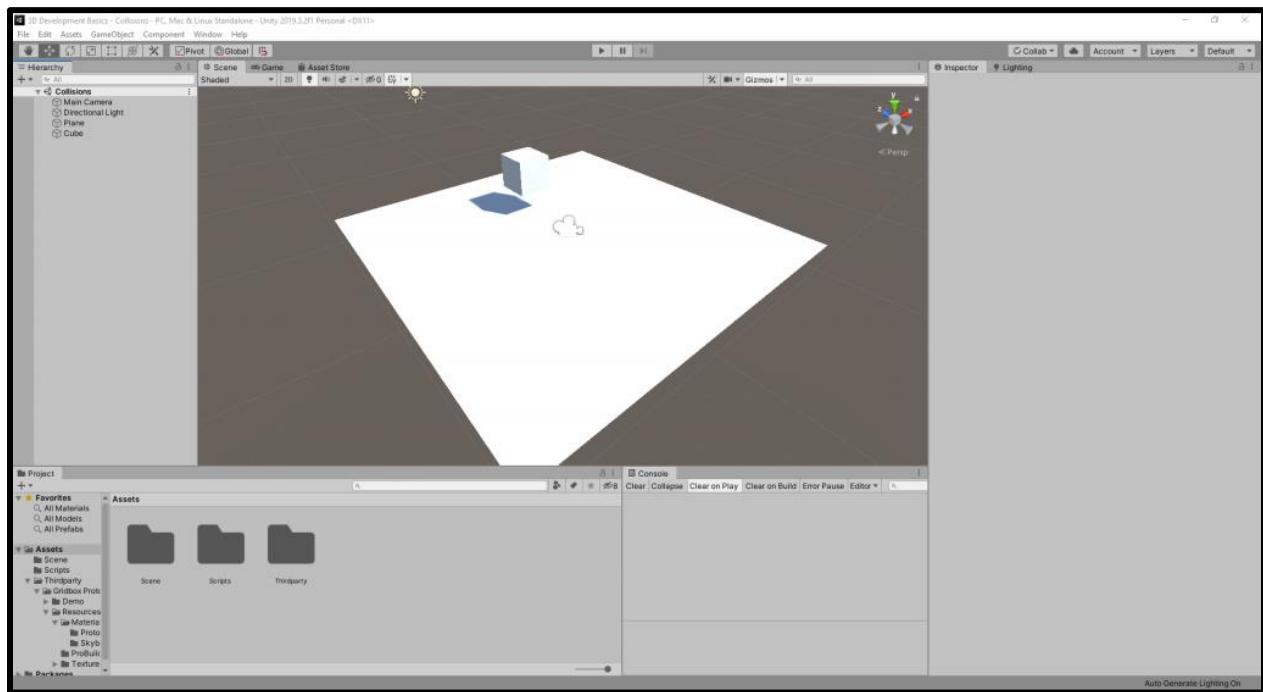
Select the Main camera and look at the Camera Preview in the bottom right of the Scene view. Neither the plane or the cube shows up. We need to fix that.



To make this part easier, I went into the Game view. Select Plane, we need to modify the Y Position of its transform to 0.8.

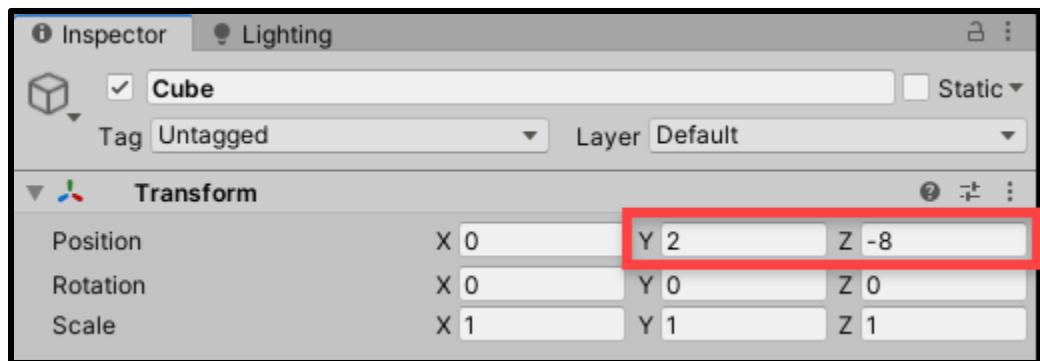
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Notice the Plane has a Plane(Mesh Filter), Mesh Collider, and Mesh Renderer. The Mesh Collider is important with physics implementation.



Select the Cube and in the Inspector Pane, we need to Change the Y and Z position of its transform. Y is set to 2, and Z is set to -8.

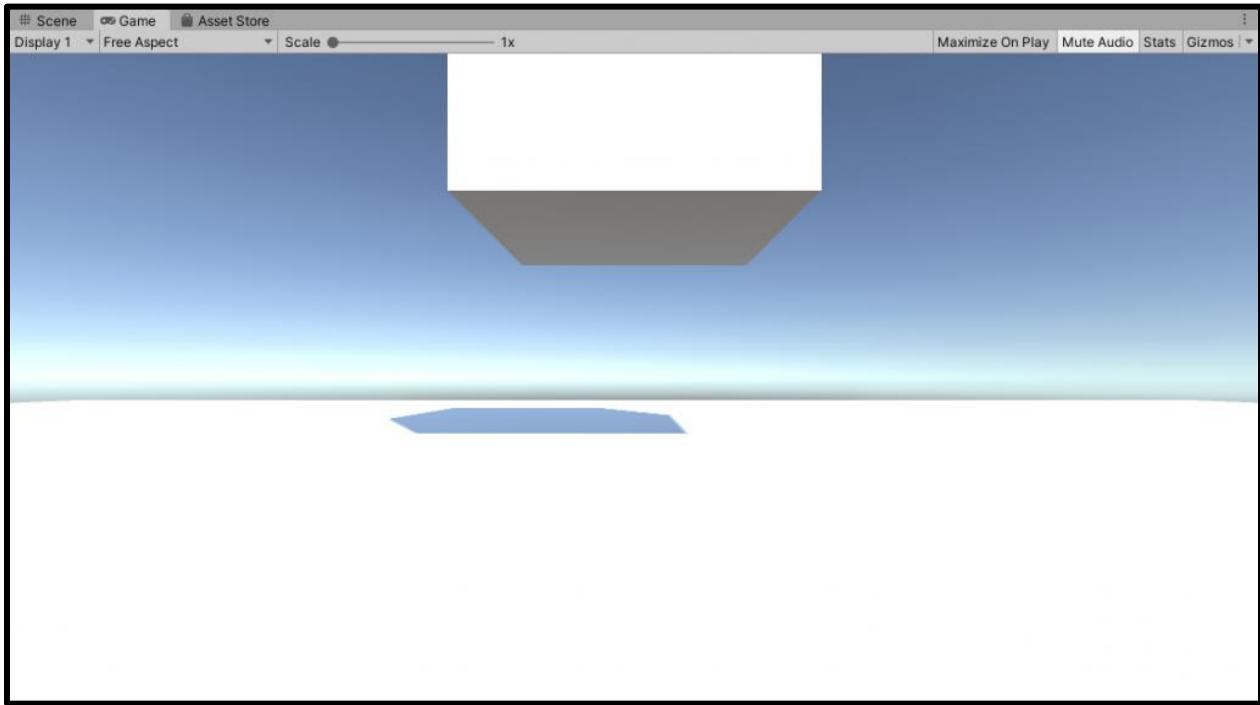
Notice the Cube has a Cube (Mesh Filter), Box Collider, and Mesh Renderer as well.



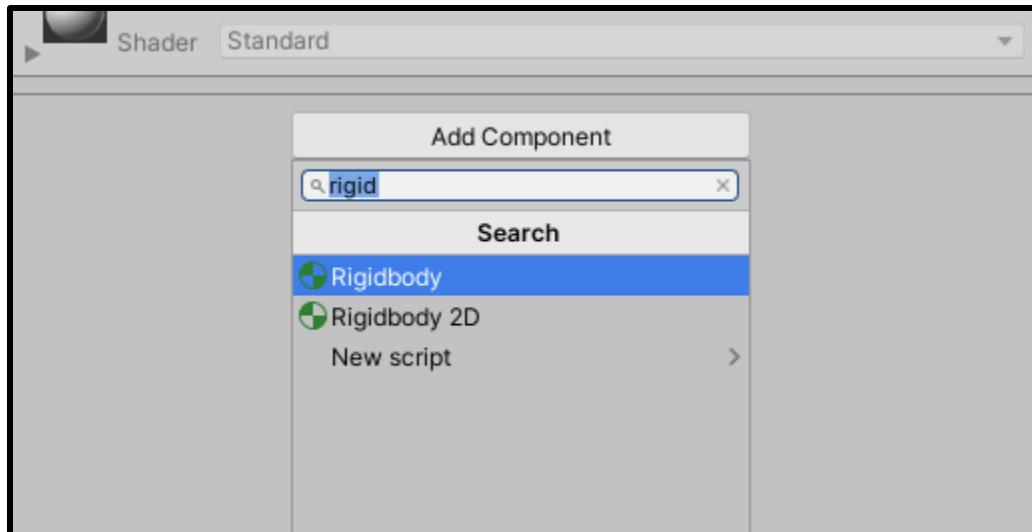
If you were to select the play button right now, you will notice that the box hangs in midair and defies gravity. There is one last step we need to complete before gravitational physics can take place. We need to add a rigidbody to the Cube.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

To do this, click on Add Component in the Inspector Pane of the Cube and select Physics.



Now, select RigidBody component.

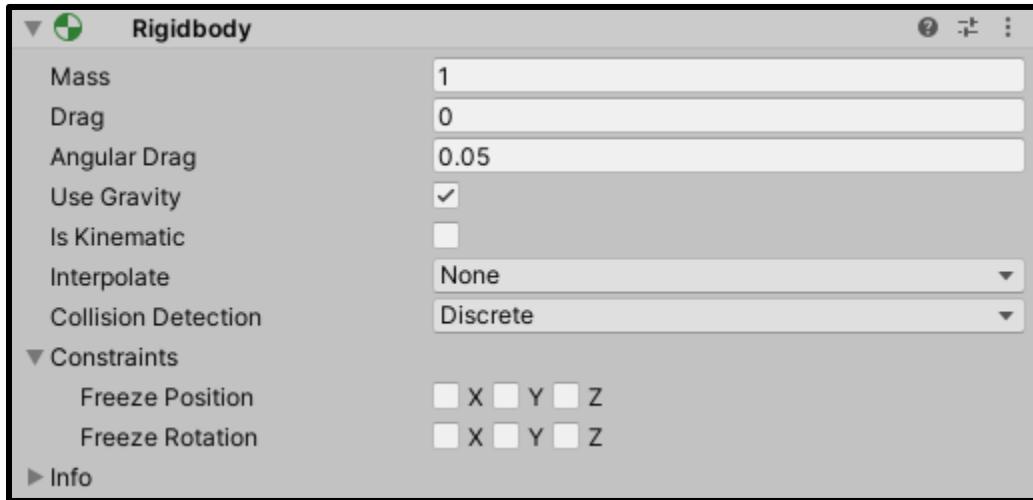


Notice the RigidBody has a few properties that have populated within the Cube in the Inspector Pane. Mass, Drag, Angular Drag, Use Gravity, Is Kinematic, Interpolate, Collision Detection, and Constraints.

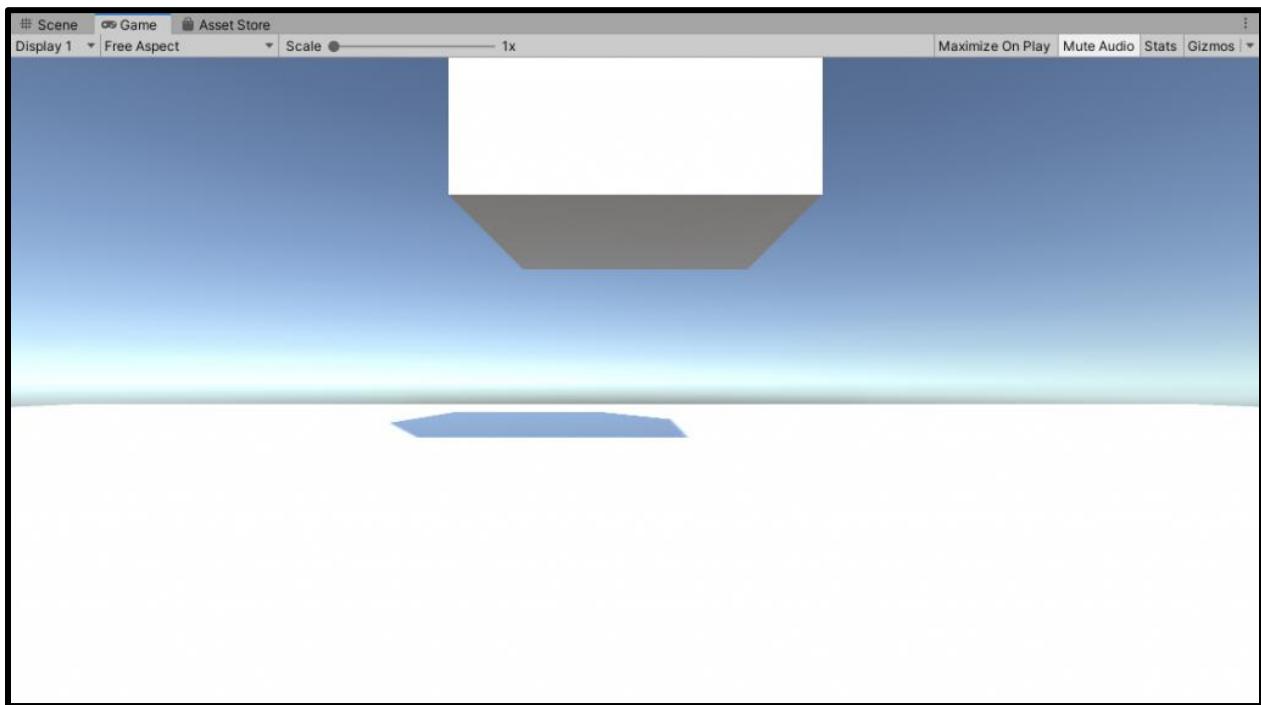
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Mass should be set to 1, drag 0, Angular Drag 0.05, Use Gravity should be checked, Is Kinematic is unchecked, Interpolate should be none, Collision Detection is Discrete, and Constraints should have nothing checked.



If you click the play button now, the box should drop onto the plane as if gravity were affecting it in real life.



Congratulations, you have now added some rudimentary physics to your game!

As you can see, the basics of 3D game development in itself can be a complex and daunting task. In the near future, we will go into much more depth and create a 3D game which houses all of the elements used within each tutorial that I upload. I hope you enjoyed this tutorial and look forward to the next one, until next time... “May your code be robust and bug free.”

A Deeper look into the Camera in Unity3D

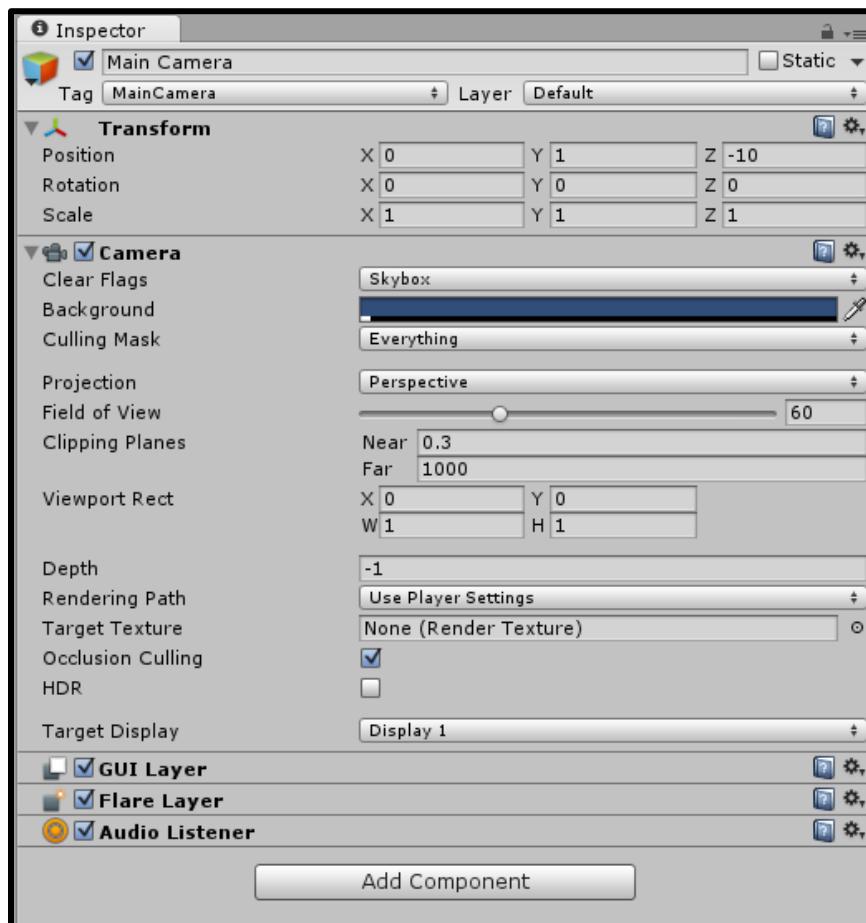
By Jesse Glover

Project files are located [here](#)

WHAT WE WILL DISCUSS TODAY:

Today we are going to take a step back and focus on one of the most important components within Unity3D, the Camera component. Unlike the other components which aren't needed to build a game or application, the Camera component is a pivotal component. You cannot run the application or game if a camera is not present.

The camera component is especially unique because you can use only one or many in an application or game. Before we start getting into the nitty gritty and adding a bunch of cameras to a scene, there are a few things that should be discussed. First off, we should go back over the Camera Component and the available properties for it.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

TUTORIAL SCREENCAST

For the ones with a more visual learning style, I've prepared a screencast showing what we do in this written article [here](#).

LET'S GET STARTED

Make sure you fully understand what you have read from the Camera Component Screenshot within the Inspector of Unity3D and what all of the properties do. This is crucial information that could lead to detrimental failures later on.

As you can see, Unity3D makes it very easy to make a split screen game, a menu, and game overlay screen using the camera component. The tutorial today will also focus more on design concepts and recreating menu's, views, and overlays that are present in various finished video games. Each design concept will be broken into its own section and gone over with a fine tooth comb, so to speak. We will discuss the ideas behind the concept and a rudimentary implementation of the example(s).

I will also note that I am using Unity3D version 5.3.0f4 which is the latest version of Unity3D.

SECTION 1: CAMERA PROPERTIES

This is the boring section, where it is all about the theory and no practical. I find that it is fairly unavoidable in terms of explaining everything. To simplify this section, I have taken a screenshot of the Unity3D manual that has this exact information. You can view the website [here](#).

Property:	Function:
Clear Flags	Determines which parts of the screen will be cleared. This is handy when using multiple Cameras to draw different game elements.
Background	Color applied to the remaining screen after all elements in view have been drawn and there is no skybox.
Culling Mask	Include or omit layers of objects to be rendered by the Camera. Assign layers to your objects in the Inspector.
Projection	Toggles the camera's capability to simulate perspective.
Perspective	Camera will render objects with perspective intact.
Orthographic	Camera will render objects uniformly, with no sense of perspective. NOTE: Deferred rendering is not supported in Orthographic mode. Forward rendering is always used.
Size (when Orthographic is selected)	The viewport size of the Camera when set to Orthographic.
Field of view (when Perspective is selected)	Width of the Camera's view angle, measured in degrees along the local Y axis.
Clipping Planes	Distances from the camera to start and stop rendering.
Near	The closest point relative to the camera that drawing will occur.
Far	The furthest point relative to the camera that drawing will occur.
Normalized View Port Rect	Four values that indicate where on the screen this camera view will be drawn, in Screen Coordinates (values 0–1).
X	The beginning horizontal position that the camera view will be drawn.
Y	The beginning vertical position that the camera view will be drawn.
W (Width)	Width of the camera output on the screen.
H (Height)	Height of the camera output on the screen.
Depth	The camera's position in the draw order. Cameras with a larger value will be drawn on top of cameras with a smaller value.
Rendering Path	Options for defining what rendering methods will be used by the camera.
Use Player Settings	This camera will use whichever Rendering Path is set in the Player Settings.
Vertex Lit	All objects rendered by this camera will be rendered as Vertex-Lit objects.
Forward	All objects will be rendered with one pass per material.
Deferred Lighting	All objects will be drawn once without lighting, then lighting of all objects will be rendered together at the end of the render queue. NOTE: If the camera's projection mode is set to Orthographic, this value is overridden, and the camera will always use Forward rendering.

There are only a few that I want to go over in detail that coincide with this lesson, future lessons will cover more of them as needed. Just because I am not covering them in detail here does not mean that you shouldn't read about them, in fact, I highly encourage it. I am grossly oversimplifying the definitions here, however, it will describe the basic ideas easier.

- **Clear Flags:** Determine what is cleared on the screen by the camera.
- **Projection:** Determines whether you are in 2D or 3D mode with the camera.
- **Perspective:** Displays the camera in 3D mode.
- **Orthographic:** Displays the camera in 2D mode.
- **Field of view:** 2D mode sets the viewport (screen space size). 3D mode sets the viewing angle of the camera.
- **Near:** closest point in the field of view that will be drawn.
- **Far:** furthest point in the field of view that will be drawn.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

- **View Port Rect:** values that describe what will be drawn in screen space.
- **X:** horizontal begin point (float value between 0 and 1).
- **Y:** vertical begin point (float value between 0 and 1).
- **W:** width of the camera on screen.
- **H:** height of the camera on screen.

SECTION 2: SPLIT SCREEN CAMERA

Welcome to section 1, we will discuss the split screen camera ideals here. Chances are, you have seen the split screen camera view in one of their many incarnations. Normally, I would show screen shots of the examples being described, however, they contain images that are trademarked to each respective company. If you are curious, a quick google search will provide these images for you. This section will describe two examples of it. The first example comes from Touhou Project, a game series developed by a single person known as ZUN. Touhou Project is most typically found to be a Bullet Hell game series, and the various incarnations use the split screen archetype differently.

Zun went with a two camera setup that overlap one another. The bottom camera houses the Score, Hi-Score, Lives count, Power (damage dealing capabilities) and Graze (How long you've gone without being hit). The top most camera has the actual game play inside of it.

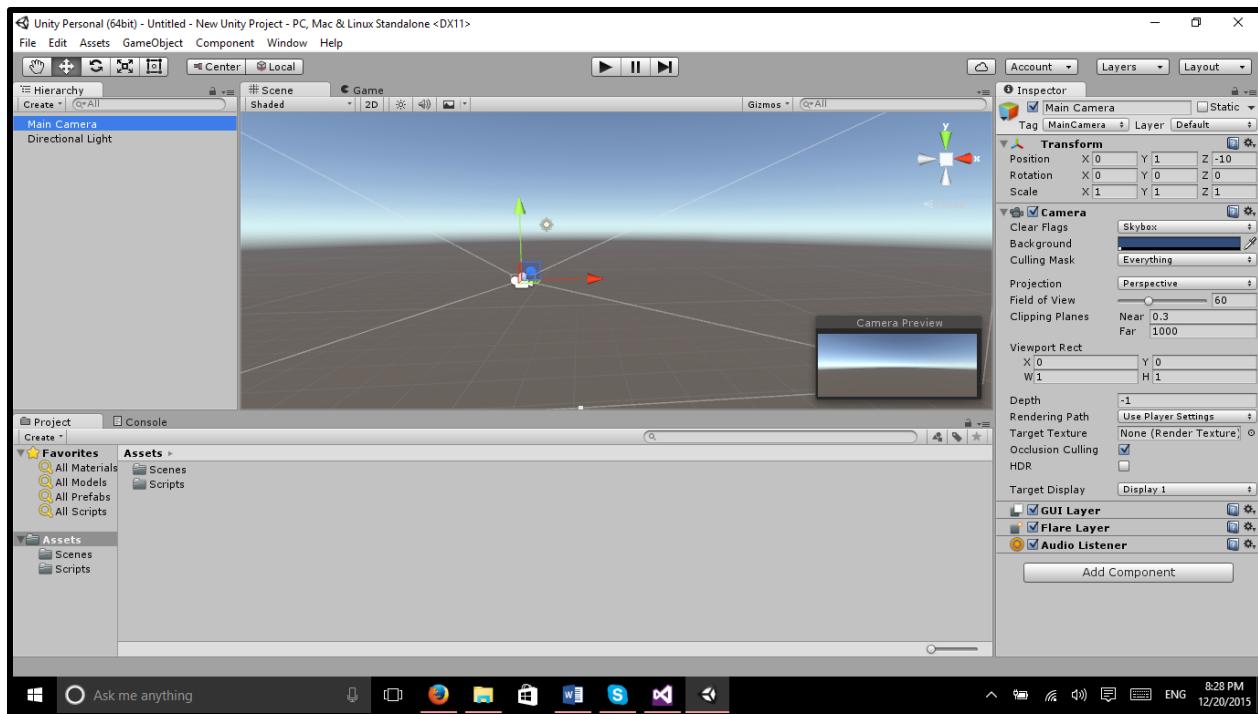
The next example is courteously given to us from the Call of Duty franchise created by Activision and Treyarch. Two cameras that make up the full screen. The top being one player and the bottom having another player. This style has started to slowly disappear from multiplayer games in favor of network only play. None the less, it is and has been a favorite style of play without network capabilities.

BUILDING THE EXAMPLES:

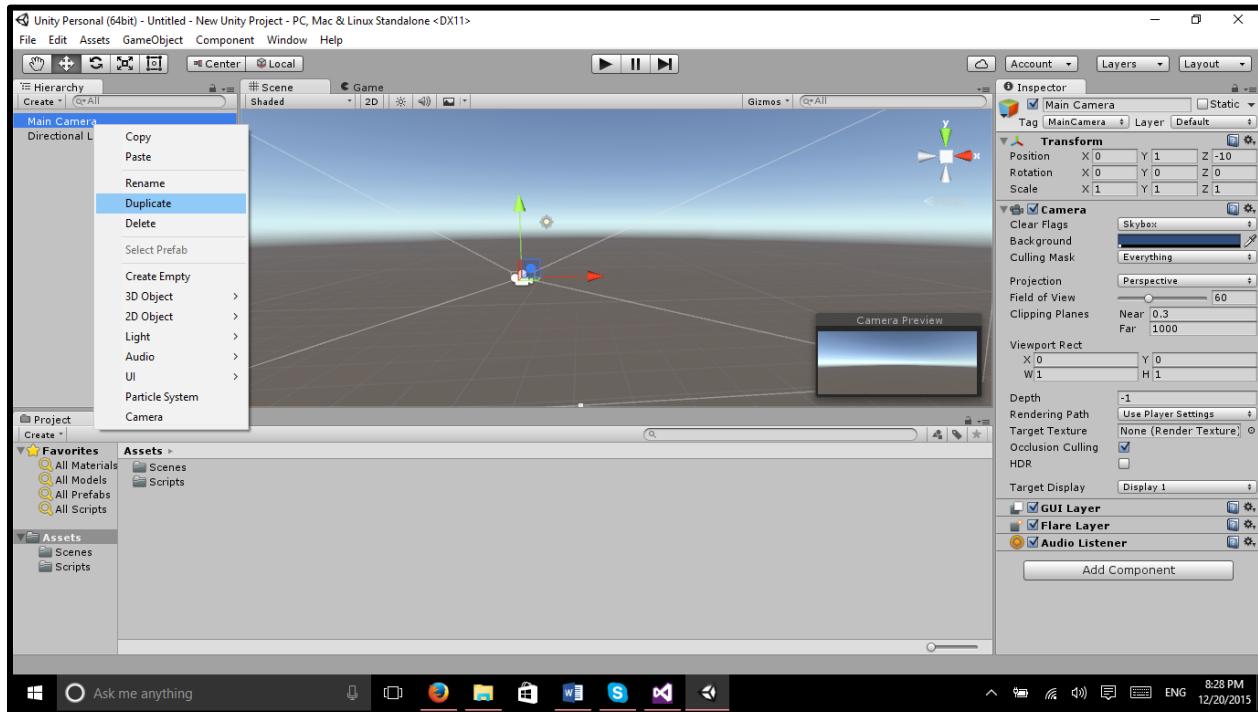
Now building both of these examples will utilize the same concept. It is up to you if you want to use a 2D or 3D camera component or project. Personally, I feel that doing this with a 3D camera is easier. As I stated earlier, they will be rudimentary examples. Essentially, I will use simple objects and components to convey the concept so that it is easier to digest.

CAMERA INSIDE ANOTHER EXAMPLE:

If you haven't already created a new project in 3D, go ahead and make one now. The only components in the hierarchy pane should be your Camera and Directional Light. If the clue left not so subtlety from the course name didn't clue you in, go ahead and select the camera component.



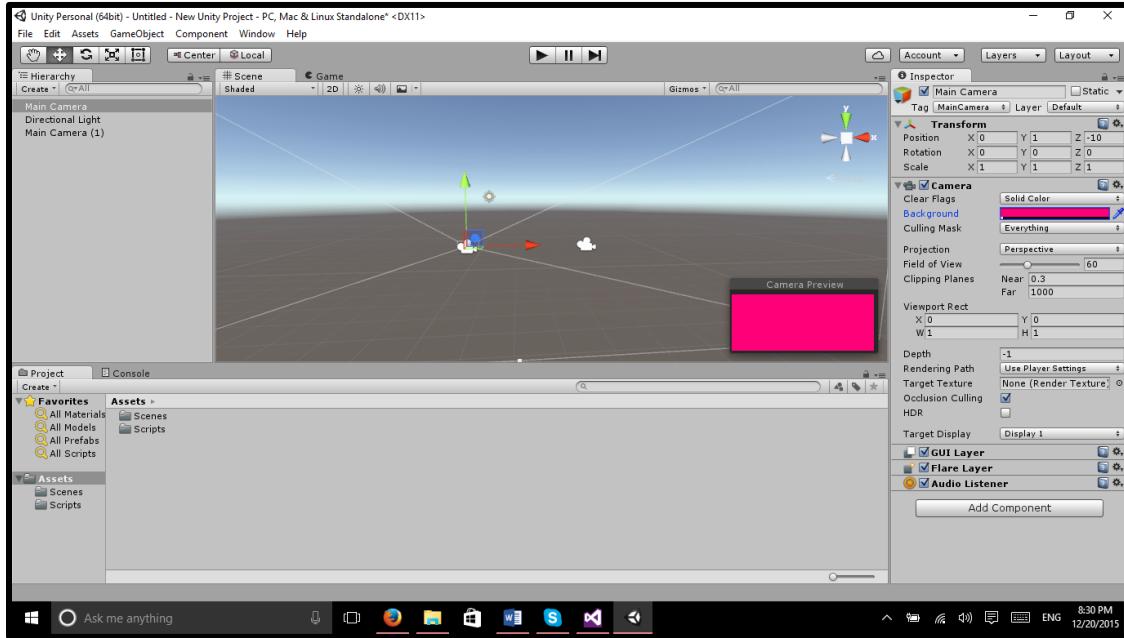
Because we want to use more than one camera here, Right click on the Main Camera and select Duplicate.



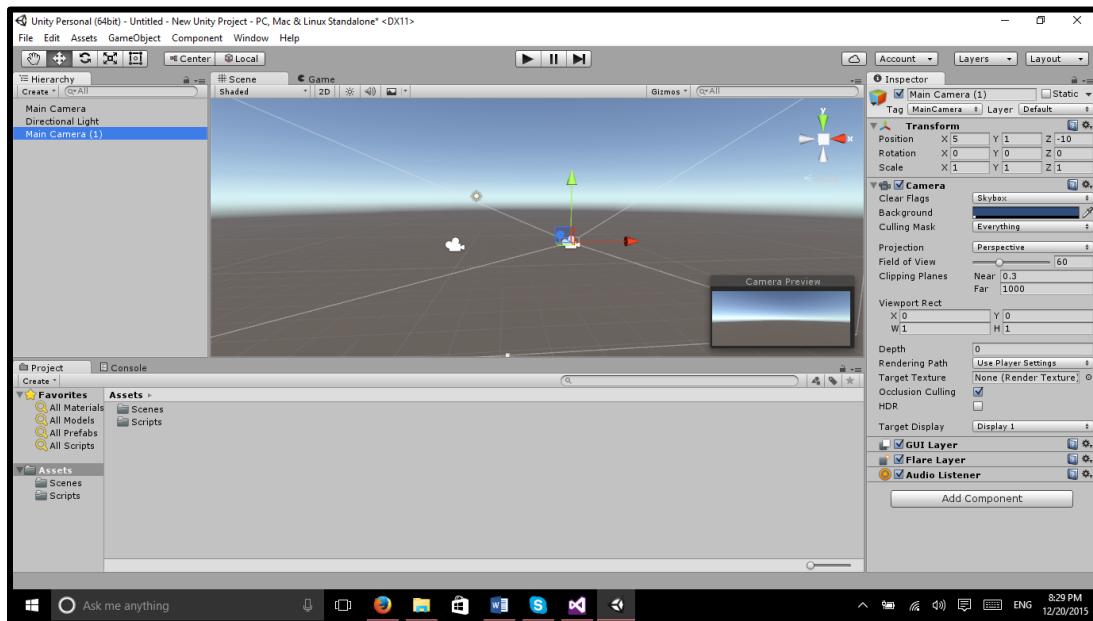
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Go back to the first camera. We have a couple of changes we want to make here. We want the clear flags to be changed to Solid Color. The background property will allow you to change the colour, change it to whatever you want it to be, I went with 255,0,121,5 (a pink). Depth should be set to -1.

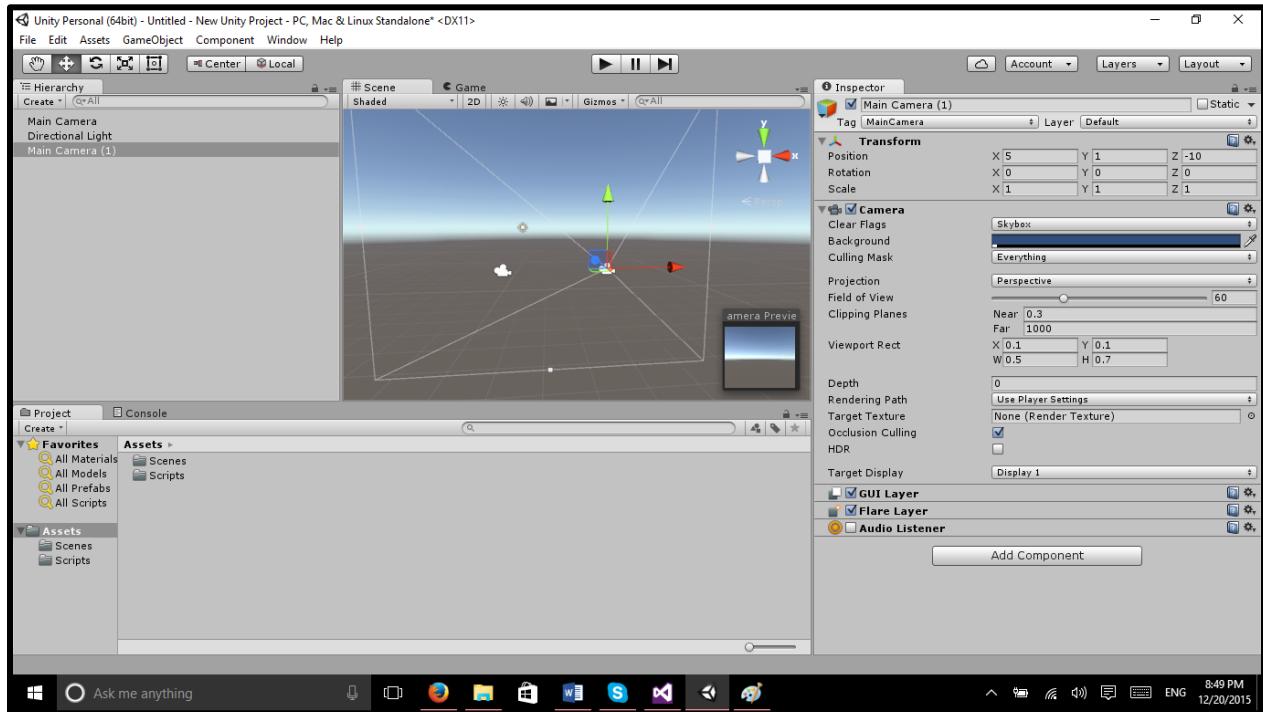


Select the Duplicated Camera, pretty much everything should be default. We need to change the depth to be 0.

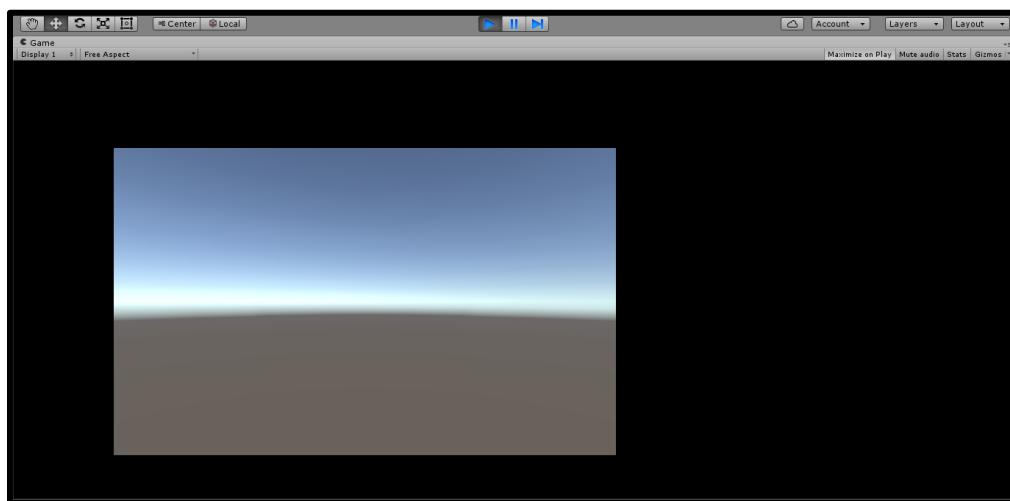


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

There is one last thing we need to do with the Duplicated Camera. We need to modify the viewport rect. We want the X value to be 0.1, Y value to be 0.1, W to be 0.5, and H to be 0.7. We also want to make sure the Audio Listener is Unchecked.



Save and run the scene and it should resemble the basic idea of what is done in the Touhou screenshot. It is really that simple, and an excellent starter point for anyone that wants to build a game with this concept in mind.

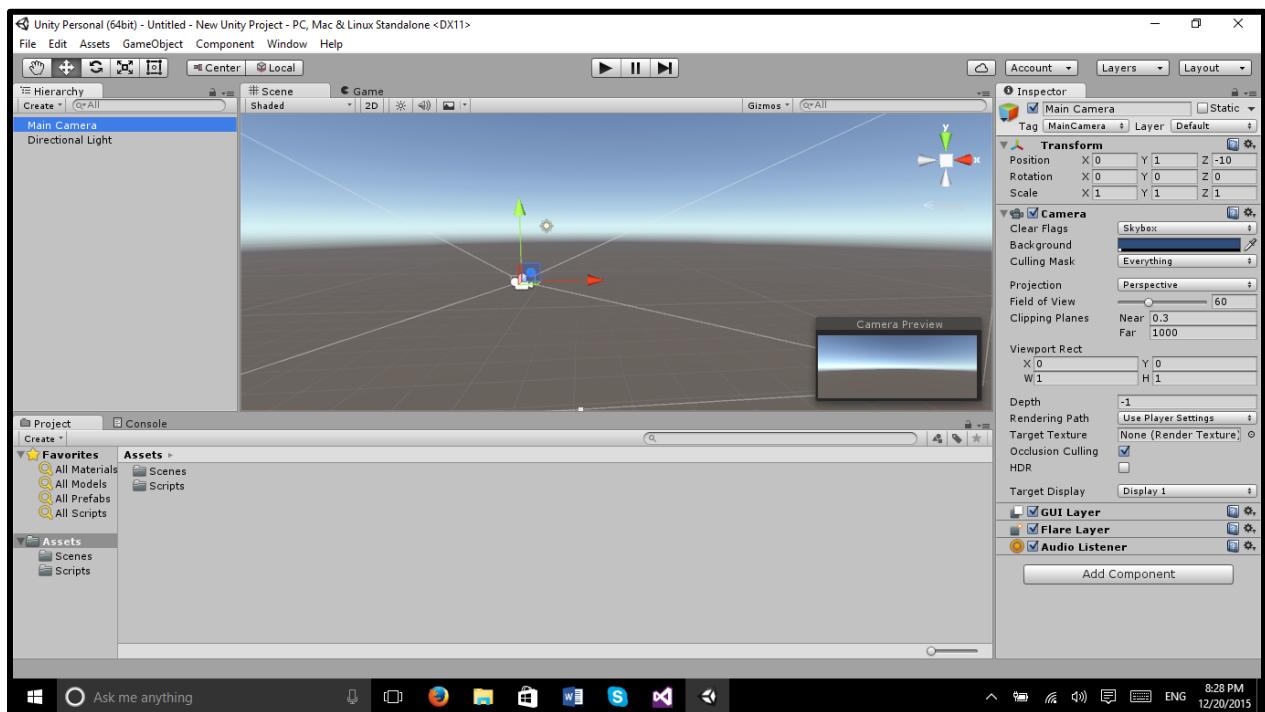


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

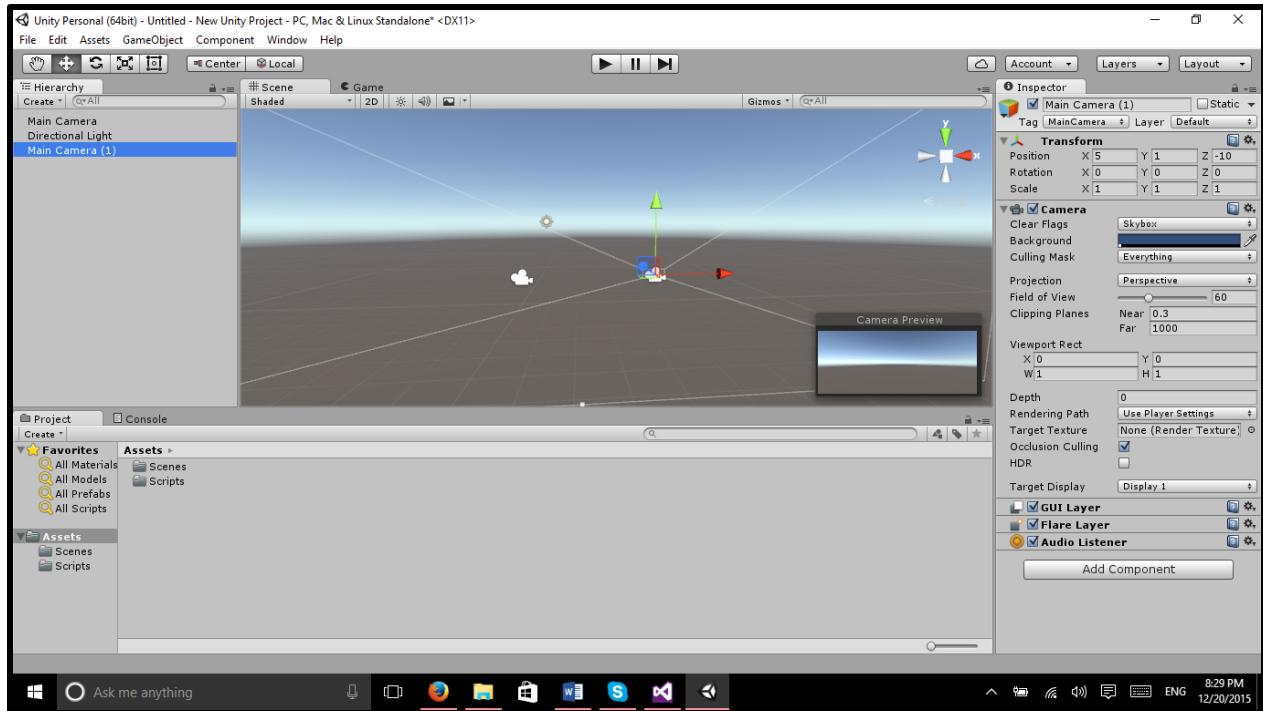
© Zenva Pty Ltd 2021. All rights reserved

HORIZONTAL SPLIT SCREEN EXAMPLE:

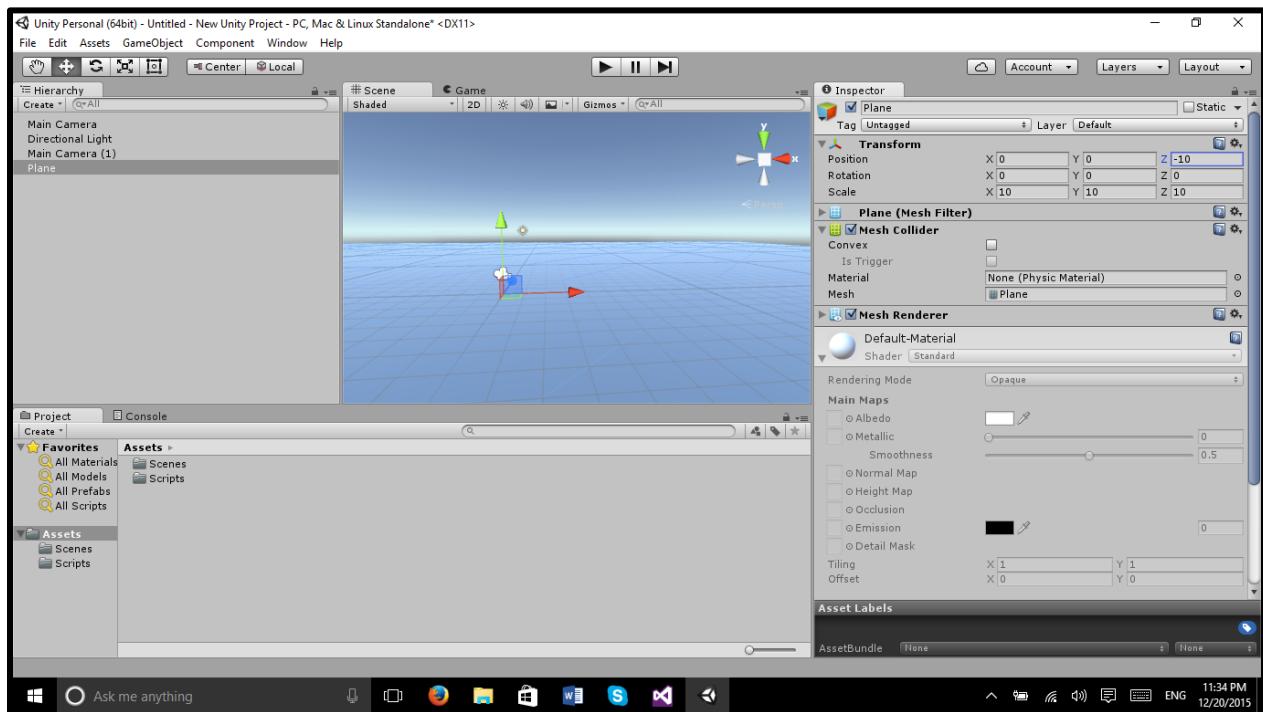
This example utilizes most of the same concepts as the first example, thus I will add a change at the end to show horizontal and vertical versions. If you haven't made a new scene, go ahead and create one now. Make sure to have the x position of the Main Camera is 0, Y is 1, and z is -10.



Again, we want to make a duplicate of the Main Camera.

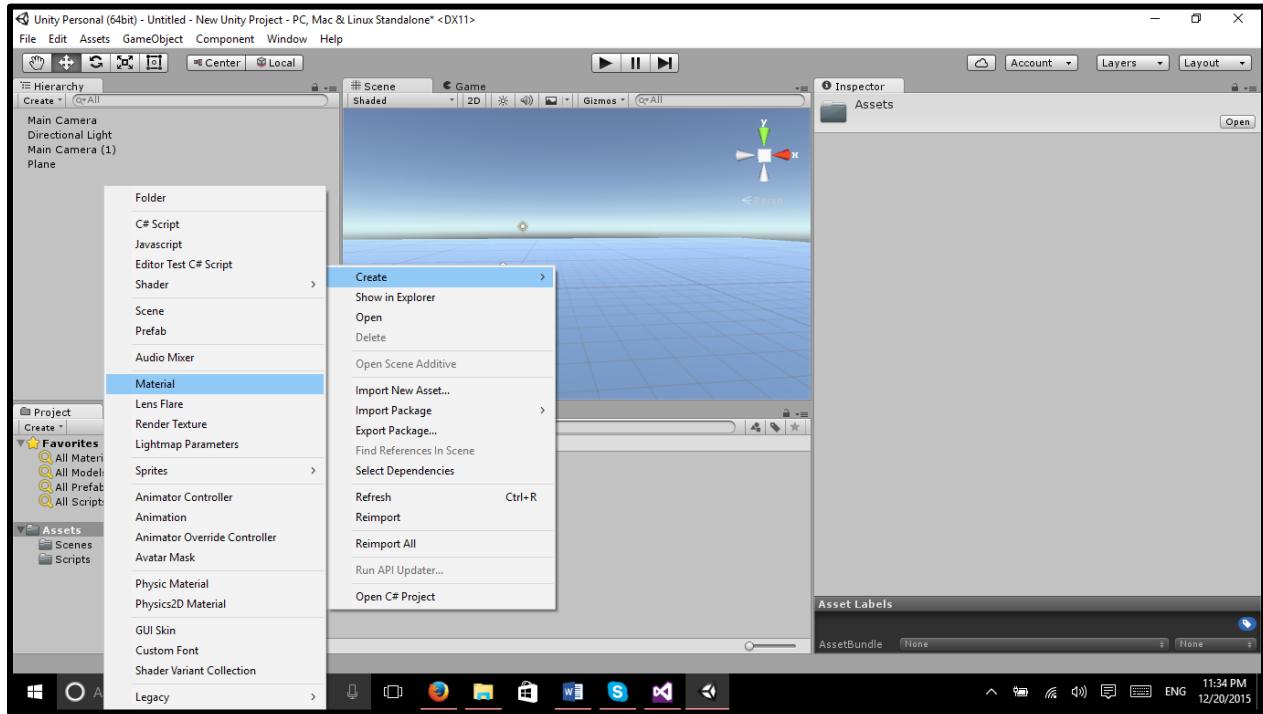


Here is where we will differentiate from the first example's concepts. First, we will create a plane and give it a colour. First off, make a plane and give it an x, y and z scale size of 10. Also, make sure the x and y position is set to 0 and the z is set to -10.



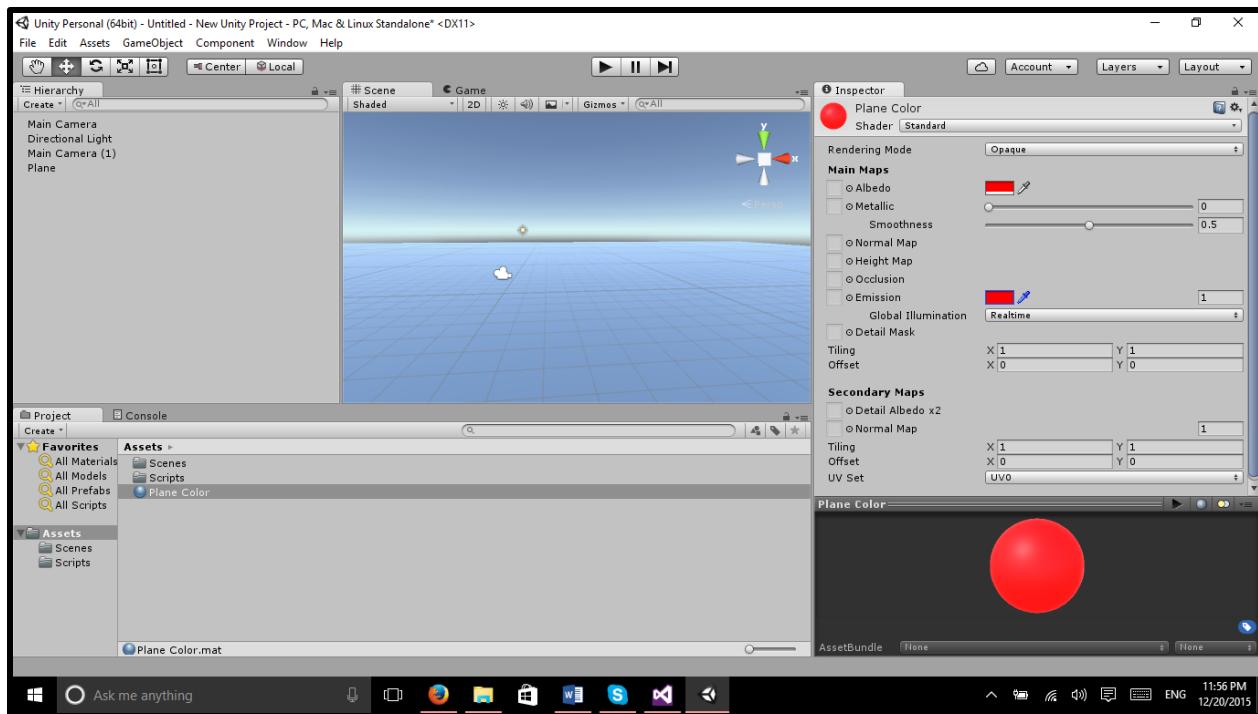
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Right click on the assets folder. Highlight create, and select Material.

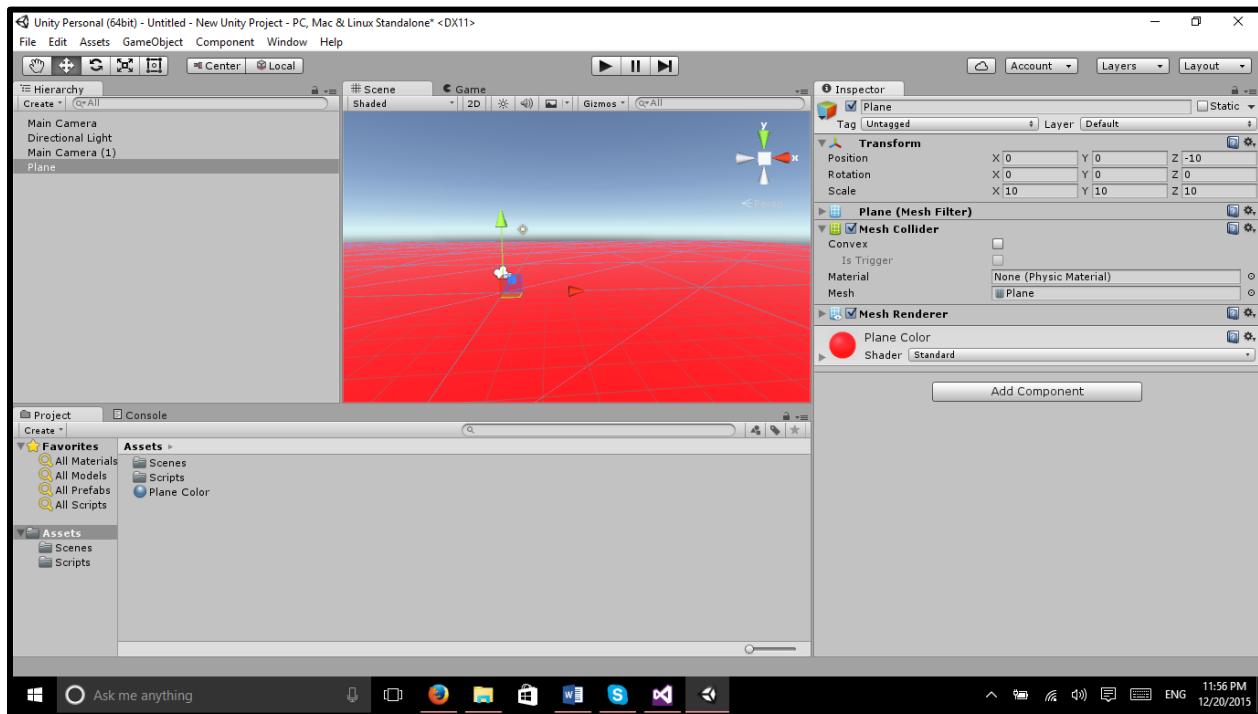


A material is a component that allows you to render a texture or shader. There are quite a few shader scripts prewritten for us to use from within Unity3D. Although you can't see specific shaders when creating a material with this tutorial, it is a good idea to keep this in mind. If you have experience with the shaderlab language, you can create your own.

Alright, now we can apply our own colour to the material. To do this, click on the colour box to select what colour you want for both Albedo and Emission. You can leave Rendering Mode and Global Illumination as is.

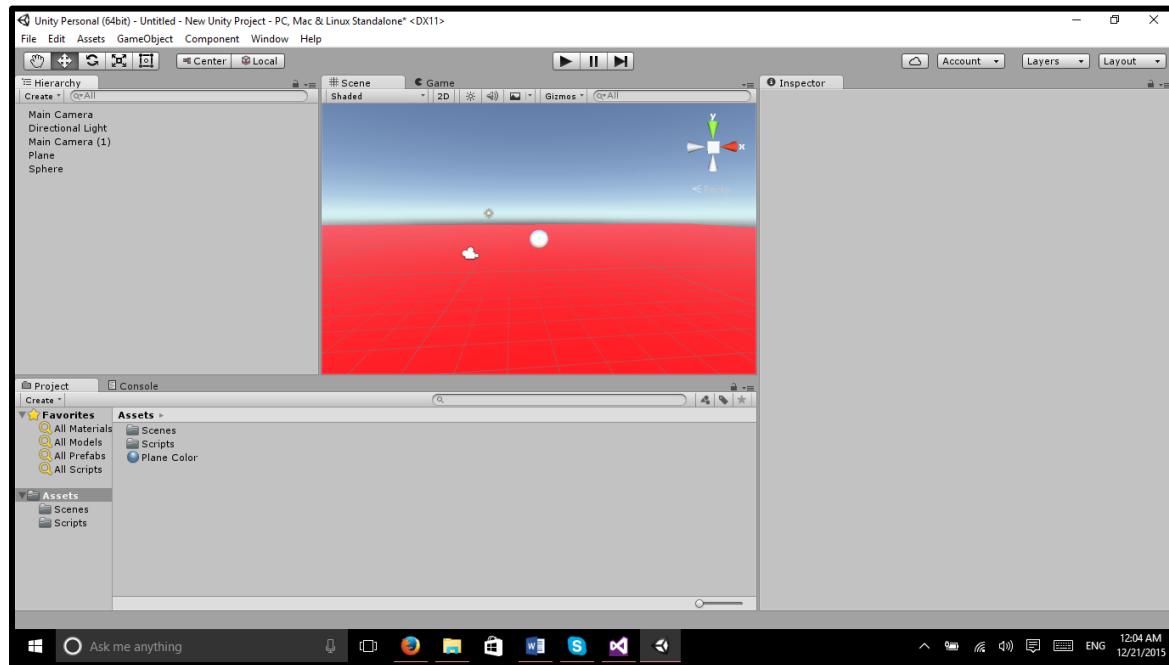


Select the plane in the hierarchy and drag the material on the add component portion of the Inspector Pane. Whatever colour you chose, the plane should now be that colour.

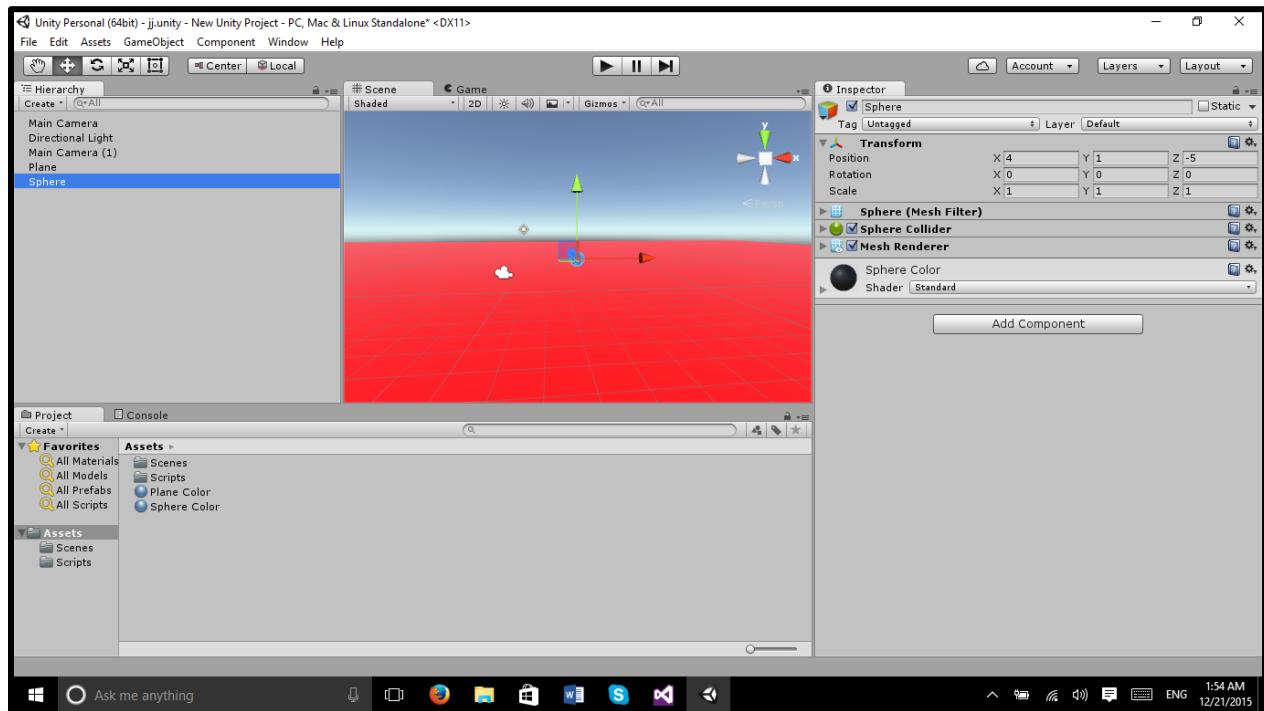


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

The scene is looking a little bland, so we should add a sphere to the scene.



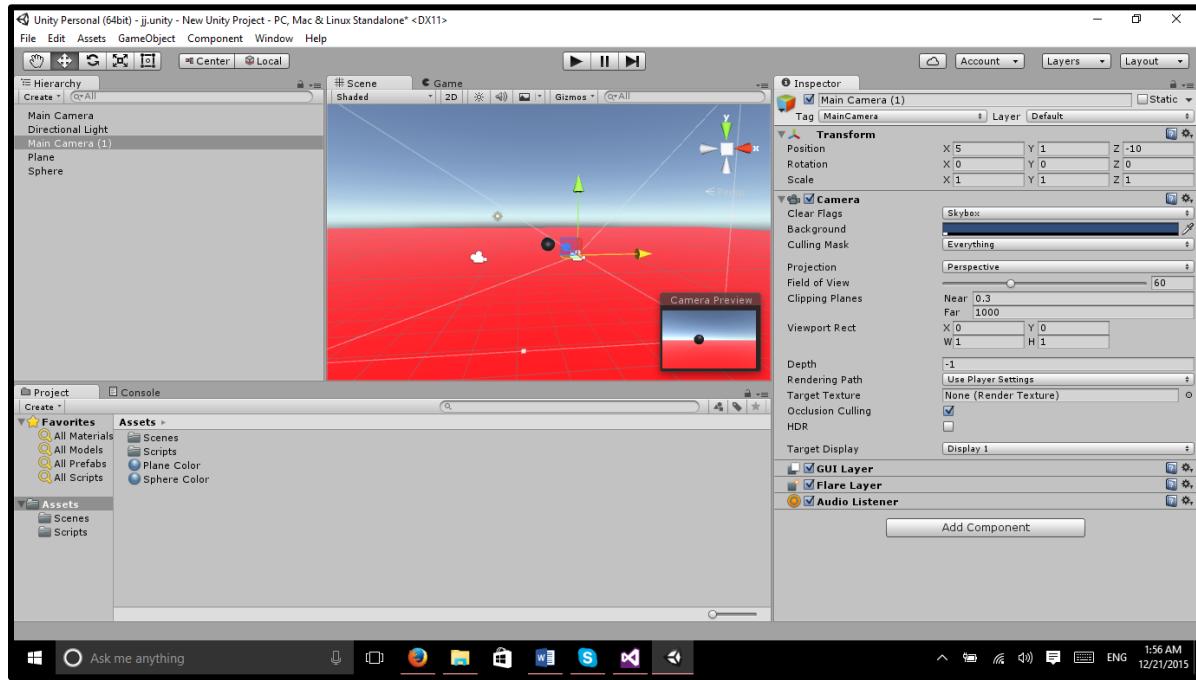
Create a material for the Sphere and attach it to the sphere just as we did for the Plane. Also make the X position of the sphere's transform to be 4, y 1, and z to be -5.



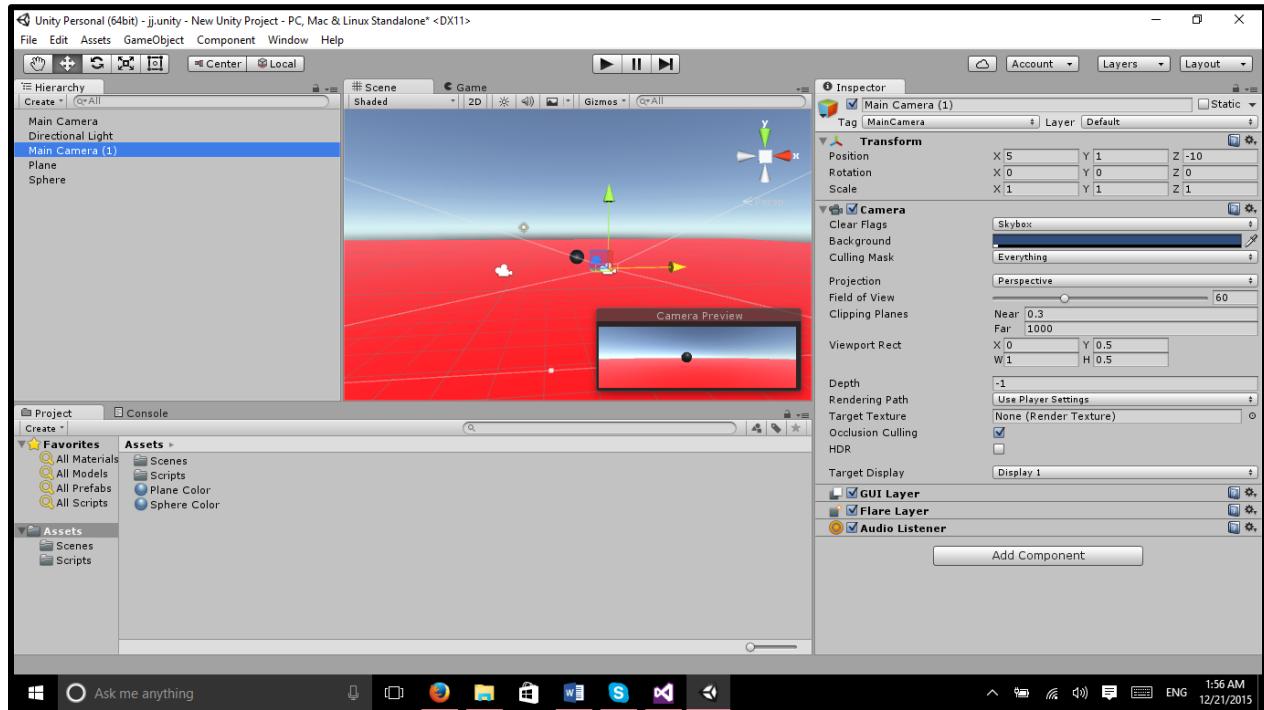
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Let's have some fun and make this look like a split screen game. Select Main Camera 2 and move the x position of its transform to 5, y should be 1, and z is -10.



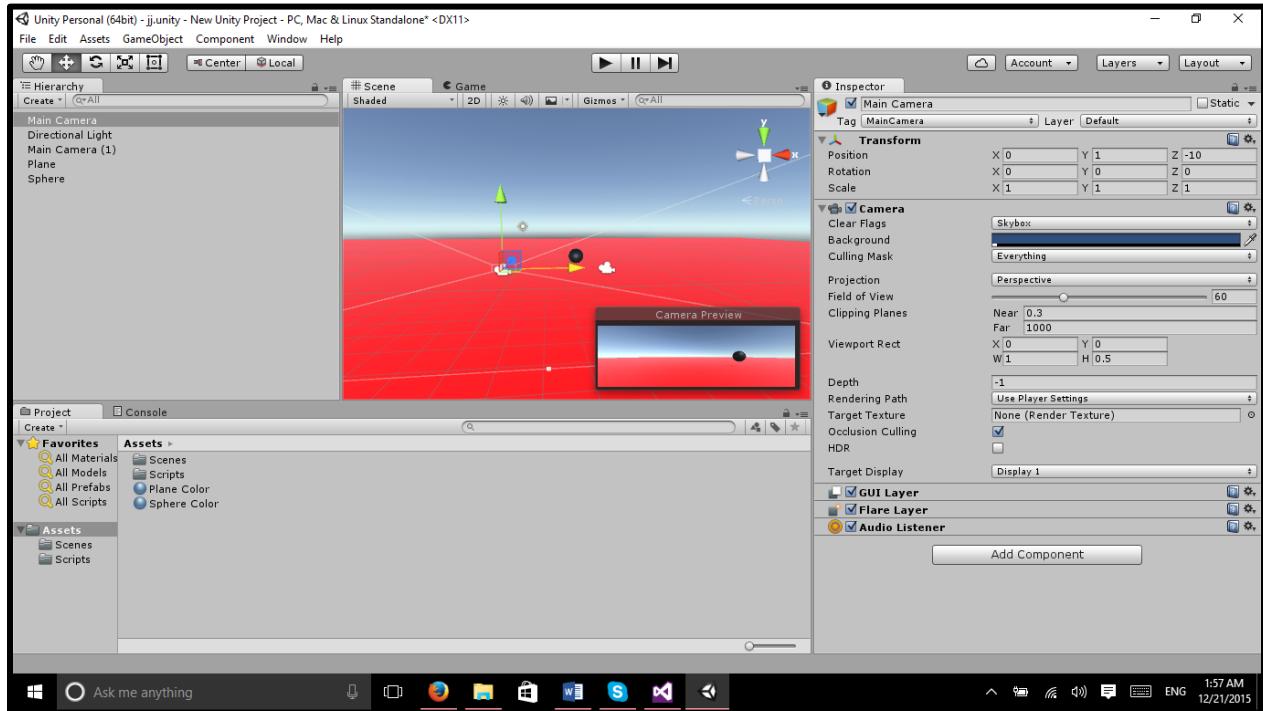
On Main Camera 2, change the Viewport Rect Y to be 0.5 and H to be 0.5



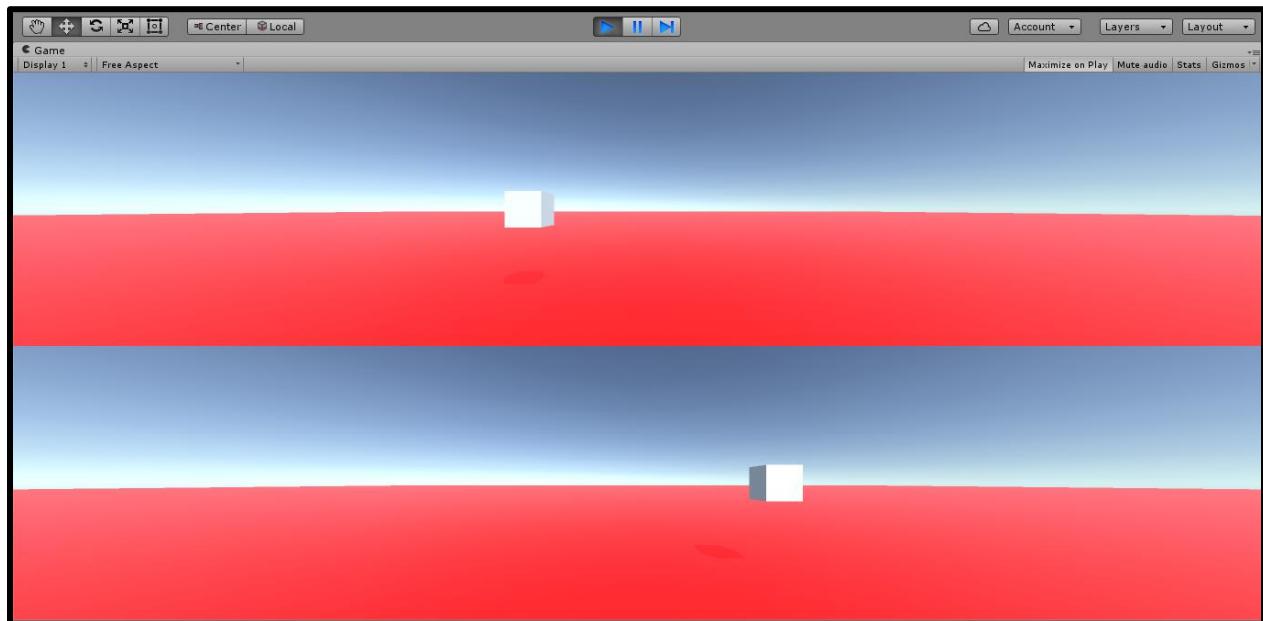
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

On the Main Camera, Viewport Rect Y should be 0 and the H should be changed to be 0.5.



Run the game and you will see a horizontal split screen view. This is basically how you would begin the basics for a split screen view for a game. Congratulations!



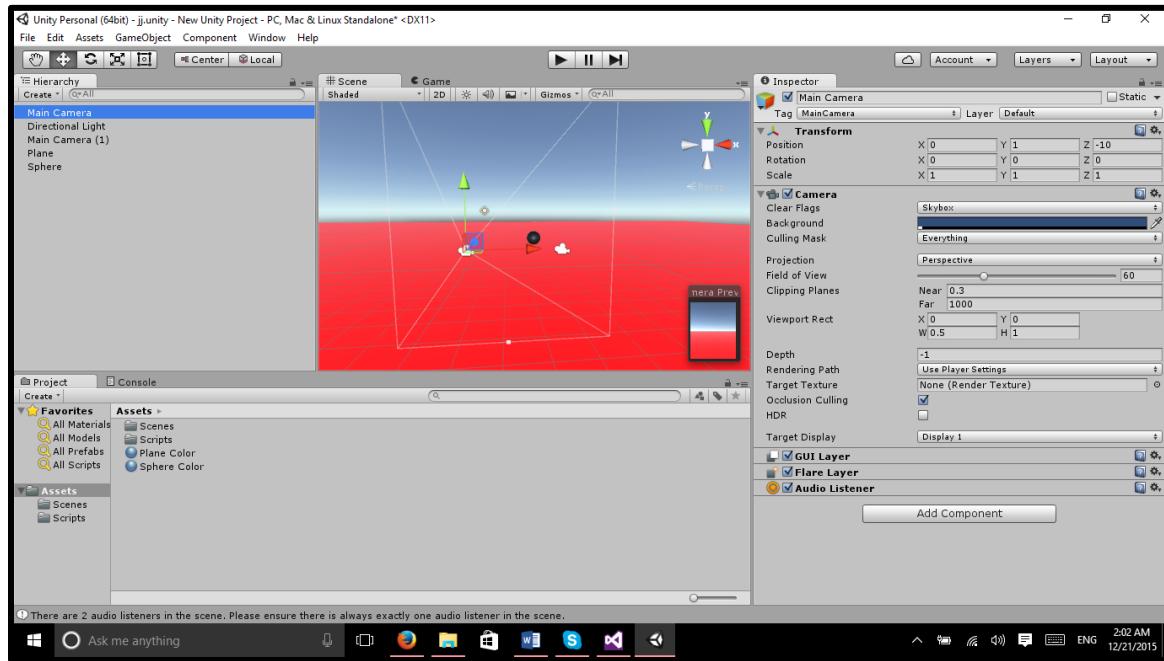
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

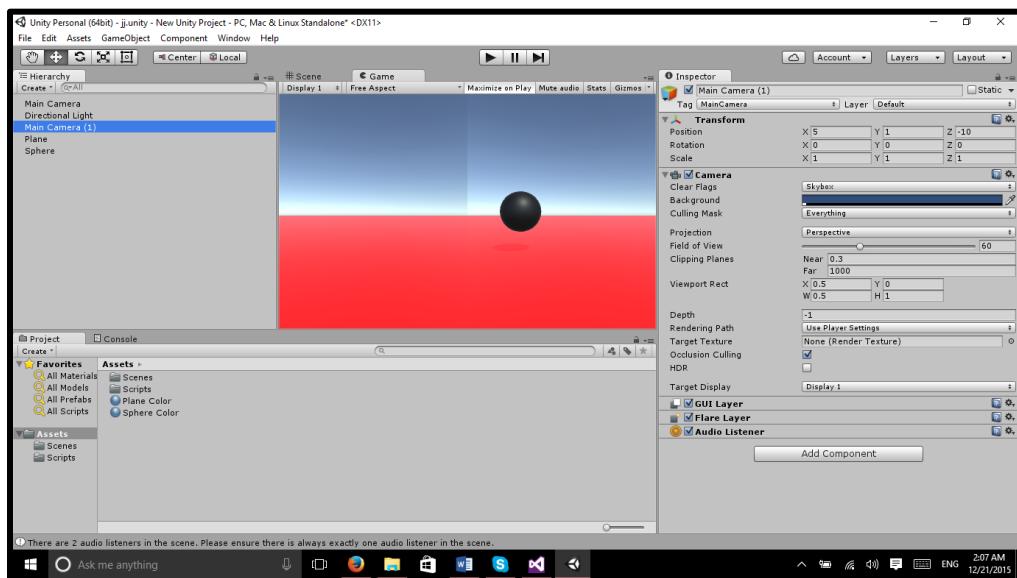
Challenge mode. Time to make it a vertical split screen view. All we need to do is change the Viewport Rect for both of the cameras to make this work appropriately.

VERTICAL SPIT SCREEN:

For the Main Camera, X should be 0, Y is 0, W is 0.5, and H is 1.

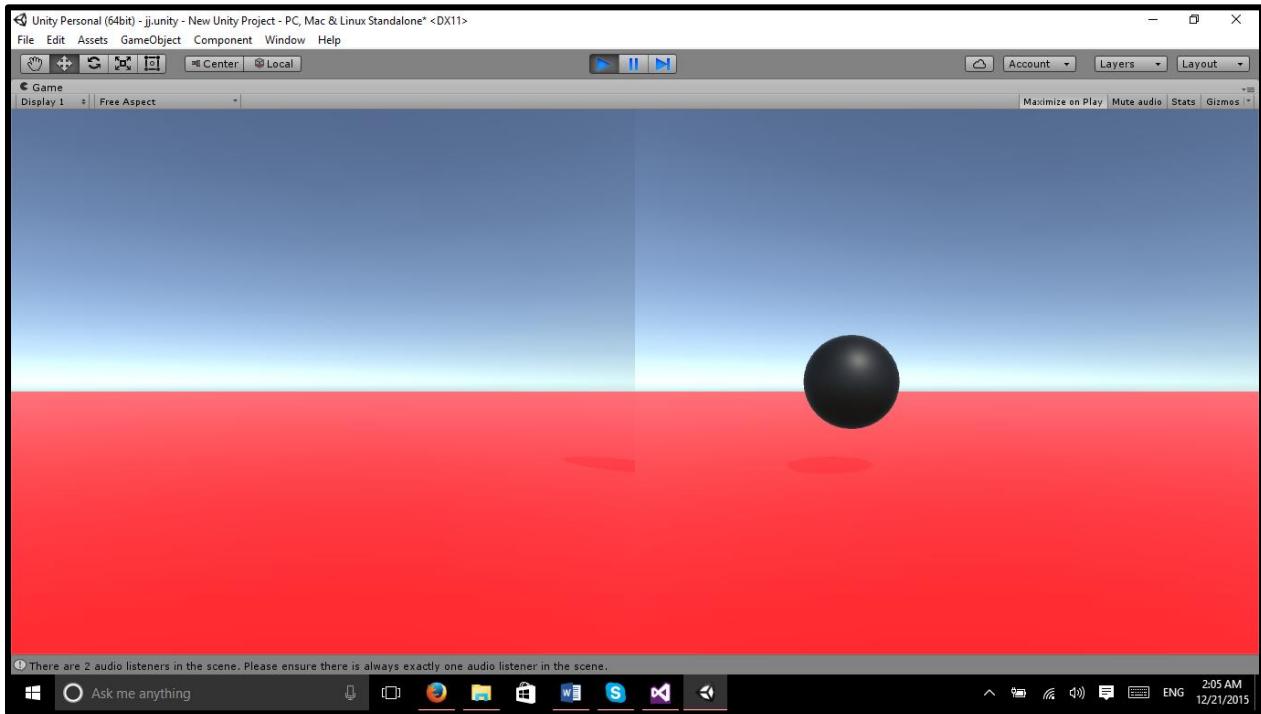


For Main Camera 2, x should be 0.5, Y is 0, W is 0.5, and H is 1.



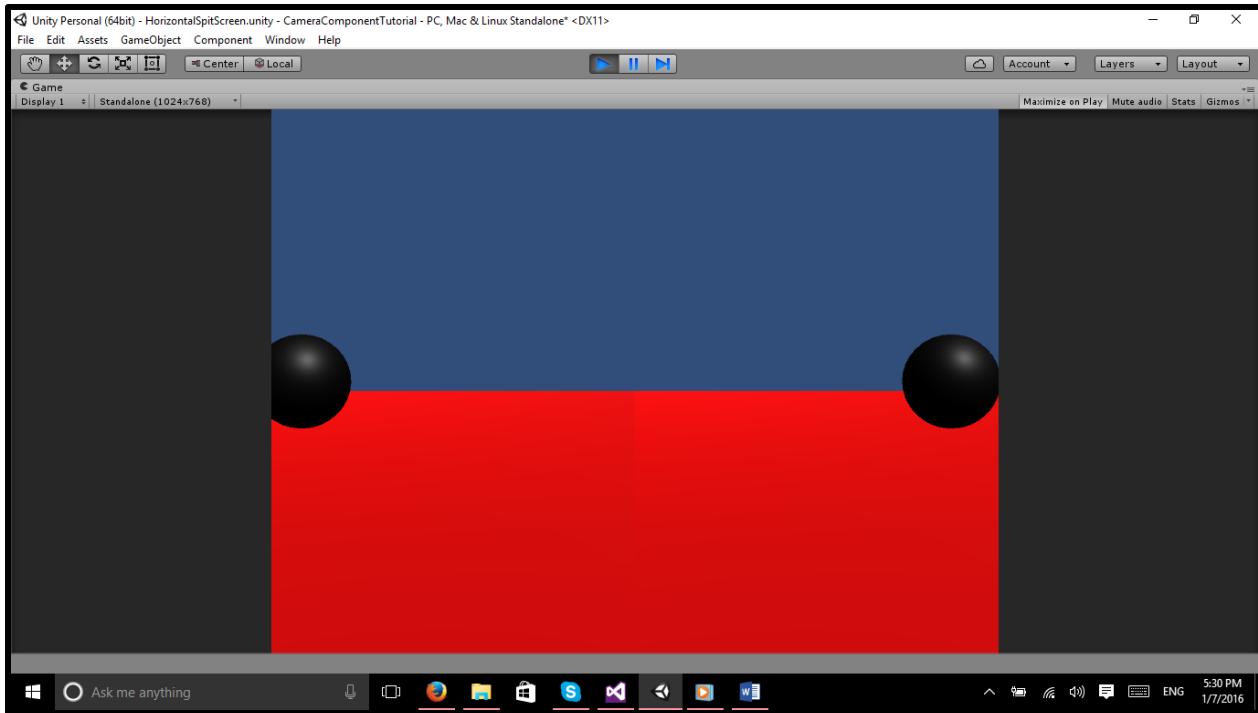
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Run the game and you will see the vertical split screen view. Due to the location of the sphere, it does not show up on the left hand side of the screen.



This is one of the dangers of doing horizontal and vertical split screen views. You want to be sure every asset is aligned correctly to prevent this.

Let's go ahead and fix this by changing the position of the sphere and secondary camera some. Change the X position of the secondary camera to 3 and leave everything else the way it was. Now, change the X position of the Sphere to be 1.41. This should allow you to see the sphere on both cameras on the left and right side respectively.



SECTION 3: CAMERA OVERLAYS

Overlays are another fundamental thing that you can do with multiple cameras. You could make a mini map hud display on the screen, a pause menu, or a score display screen on separate cameras and call them on a button press. These concepts are used with single player and multiplayer games that you see today.

PAUSE MENU:

The pause menu has been used for quite some time and is highly doubtful to be removed anytime in the near future. Pause menus work differently depending on the game. Multiplayer games will put a mark on the player stating that they are in a menu, and display for that player a specific menu or a game paused overlay. The Hud display or Mini map is also frequently used. The design idea for it is to give the player a scale model idea of where they are in the game world. It can also be extended to have objective markers to show the player where they need to go. This is used in the FPS (First Person Shooter), RTS (Real Time Strategy), Action RPG, and MMORPG (Massively Multiplayer Online Role Playing Game) genres. I will call upon one of my favorite development studios to give an example of this, Blizzard Entertainment, with Diablo 3. The last overlay option that I want to discuss today is the Score keeping overlay. This overlay type's design idea comes from not wanting to overload the player without much information that they don't need. This is mostly used in the RTS and FPS genres, although, it can very easily be used with any other game type. Activision is quite known for this sort of overlay with the Call of

Duty Franchise.

While we aren't going to build examples for these, it is important to keep these sort of ideas in mind when developing games and even some types of software. These concepts are instrumental for creating immersive games and a great UX (User Experience). I hope you enjoyed this tutorial and look forward to the next one, until next time... "May your code be robust and bug free."

How to Script a 2D Tile Map in Unity3D

By Jesse Glover

Prologue: The Back Story

A little while ago, a friend and mentor and I were talking about the tutorials I write for this site. I was talking about how I feel I should write a tutorial that deals more with scripting and less with components; and how I couldn't decide what the concept for the tutorial would be. We went back and forth with ideas for a few hours with the pros and cons of each. He then reminded me of a time when he tutored me on a tiling engine and was explaining how algorithms weren't as scary as I thought.

Suddenly, it hit me like a stampede. I should make a tutorial on making a working tile map system while explaining the editor concepts, the ideas behind the code, and ways you could extend it for a top down adventure game.

TUTORIAL SOURCE CODE

Download the source files for the course [here](#).

Link for Visio Alternatives here.

SECTION 1: THE BORING THEORY SECTION

If you are a beginner, you may have never heard of a tile map. A tile map is a procedurally generated level map that is created within a given set of rules. This in itself can be confusing and may draw questions, so we should break that down.

What are map generation rules?

- map is a 5 x 5 grid
- if a tile is marked unset, set a tile in that spot
- if a tile would go outside of grid size, do not set a tile.
- These are the basic rules you could employ for making a tile map system.

What programming keywords or loops will we use for the tile map system?

Enum, nested for loops, foreach loops, constructors, overloading, and if statements

Can this map system be modified to Isometric, Diametric or Hexagonal?

With this question, I am going to say this. If you can understand the rules, the fundamental math behind each style and what was done with the implementation, you could convert it to either an

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Isometric, Diametric or Hexagonal grid. To be completely honest, I have not been able to successfully do Isometric, Diametric, or Hexagonal maps to date.

Can this map system be used in an RTS game?

If you are a beginner, I do not recommend you start off with an RTS game. They are extremely difficult to prototype and fully implement. If you have a mentor or are shadowing a team of developers, I would highly suggest you talk to them and see what they think.

That being said, many of the classic RTS games did use a tiling system similar to this. So it can absolutely be used.

Now that these questions are done, I think we should actually talk about the math involved. The math involved isn't as complicated as you might think, if you are like me and are deathly afraid of math.

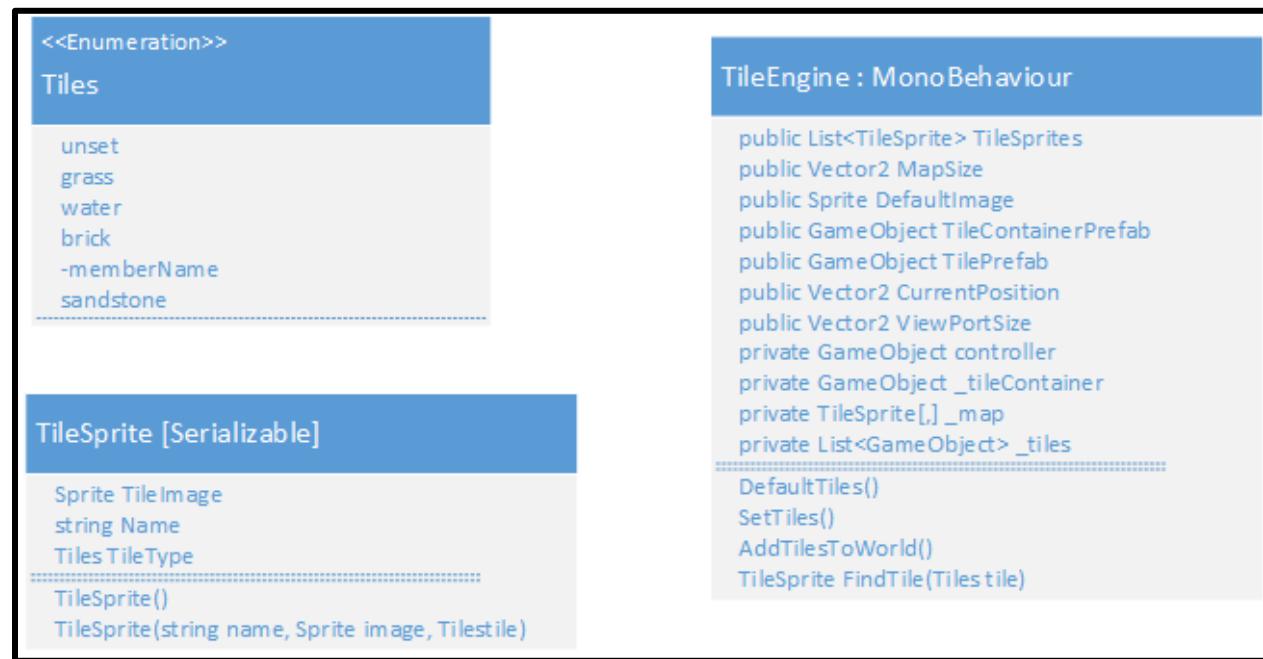
What information do we need mathematically?

Size of images, the size of the grid you want to display, and the screen size. You should also have a good understanding of some linear math, multiplication, and positive / negative number counting.

I think this just about covers all of the theory behind making a tile map system. I've kept it language agnostic so this portion of the tutorial can be applied to whatever language you are learning. What I mean by language agnostic is this section can be applied to any programming language. If you decide to learn Typescript and Phaser, BabylonJs, LimeJS, Quintus, Unreal Engine, Construct2, HTML5, or any other programming language / Game Engine; You could use this portion of the tutorial and it will work!

SECTION 2: PLANNING THE CODE

In this section, we will be dealing with planning out our source code and then writing it. What I mean by this is, we will first create the UML Diagram of how the code should be structured and then actually write the code. It is important that I stress how imperative it is to plan your code first, and this holds true regardless of whether you design games, apps on any platform, and websites. Proper planning and documentation means that you will spend less time trying to figure out what you are trying to implement and how you want to implement it. It also means that you can much more easily add new features to the code and know exactly where it goes, what it inherits from, and if it conflicts with another portion of your code. For planning this out, I am using Microsoft Visio 2015. If you do not have Visio or the funds to purchase it, you can use Lucid Chart or Libra Office Draw instead.



As you can see, we separated everything according to what class it should be in. We made the Enum be in its own class called Tiles, TileSprite class is marked serializable and has no inheritance, and TilingEngine inherits from MonoBehaviour. We also wrote out our properties and methods. We separated the properties from the methods and made sure to add the parentheses to designate methods.

SECTION 2.5: WRITING THE CODE

In this section, we will be writing the actual code. I know you guys are elated to finally see some code and how we can apply it to Unity3D. Since this is with the beginner in mind, I will be explaining the code through self documenting code and with remarks after each code segment regarding it. First up, is the Tiles class.

```
1 using UnityEngine;
2 using System.Collections;
3
4 public enum Tiles
5 {
6     Unset,
7     Grass,
8     Water,
9     Brick,
10    Sandstone
11 }
```

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Here, we are making use of the enum keyword. If you are unfamiliar with the .Net keywords, I would suggest that you stop reading this tutorial and read [this guide](#) on all the C# keywords in the .Net Framework. In the code above, the compiler would see this as Unset = 0, Grass = 1, and so on. You could explicitly set this, however, you don't need to in most cases.

Next up, is the TileSprite Class.

```
1  using System;
2  using UnityEngine;
3  using System.Collections;
4
5  [Serializable]
6  public class TileSprite
7  {
8      public string Name;
9      public Sprite TileImage;
10     public Tiles TileType;
11
12    public TileSprite()
13    {
14        Name = "Unset";
15        TileImage = new Sprite();
16        TileType = Tiles.Unset;
17    }
18
19    public TileSprite(string name, Sprite image, Tiles tile)
20    {
21        Name = name;
22        TileImage = image;
23        TileType = tile;
24    }
25 }
```

We declared a string as name, sprite as tile image, tiles as tile type. In the default constructor, we set the name to unset, the tile image as new sprite, and tile type as tiles.unset. What we are doing here in the default constructor is giving a string value to associate with the sprite and enum's value of unset. Next up, we overload the default constructor with a method signature of

string name, sprite image, and tiles tile. This is done so later on we can modify the name, image, and tile type later on.

Last on the list is our TilingEngine class. I am going to break it down into smaller sections before revealing the full code.

```
1  using UnityEngine;
2  using System.Collections;
3  using System.Collections.Generic;
4  using Lean;
5
6  public class TilingEngine : MonoBehaviour
7  {
8      public List<TileSprite> TileSprites;
9      public Vector2 MapSize;
10     public Sprite DefaultImage;
11     public GameObject TileContainerPrefab;
12     public GameObject TilePrefab;
13     public Vector2 CurrentPosition;
14     public Vector2 ViewPortSize;
15
16     private TileSprite[,] _map;
17     private GameObject controller;
18     private GameObject _tileContainer;
19     private List<GameObject> _tiles = new List<GameObject>();
20
21     private TileSprite FindTile(Tiles tile)
22     {
23         foreach (TileSprite tileSprite in TileSprites)
24         {
25             if (tileSprite.TileType == tile) return tileSprite;
26         }
27         return null;
28     }
29 }
```

Alright, the properties on this one is fairly beefy. I'm sure you noticed a new using statement for Lean. That is because we are using an asset called LeanPool. It is a free asset in the asset store and it is designed to be used for object pooling. Object pooling is recycling objects rather than simply deleting them. It uses less memory in the long run and works fantastically for objects that will be reused extremely often. We have a list that has a parameter of tile sprite called tile sprite, a Vector 2 of MapSize, Sprite of default image, two gameobjects for tile container prefab and tile prefab, a vector 2 for current position, vector 2 for view port size. For

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

the private properties we have; Tile sprite as an array called map, list with a parameter of gameobject called tiles, and 2 gameobject respectively called controller and tile controller.

Each of these properties will be discussed as we get to them in the script. On that note, our first method is Find tile with a parameter of tile. Inside we have a foreach loop that goes through each tile sprite and if it is equal to tile, it returns a tile. If it doesn't equal tile, it returns null. Basic error handling here.

On to the next section of the code. I promise it won't be as big as the first.

```
1  private void DefaultTiles()
2  {
3      for (var y = 0; y < MapSize.y - 1; y++)
4      {
5          for (var x = 0; x < MapSize.x - 1; x++)
6          {
7              _map[x, y] = new TileSprite("unset", DefaultImage, Tiles.Unset);
8          }
9      }
10 }
11
12 private void SetTiles()
13 {
14     var index = 0;
15     for (var y = 0; y < MapSize.y - 1; y++)
16     {
17         for (var x = 0; x < MapSize.x - 1; x++)
18         {
19             _map[x, y] = new TileSprite(TileSprites[index].Name,
20                 TileSprites[index].TileImage, TileSprites[index].TileType);
21             index++;
22             if (index > TileSprites.Count - 1) index = 0;
23         }
24     }
25 }
```

Our first method is Default tile. Here we see a nested for loop, or a for loop inside of another for loop. Here we see Map size x and y in the loops as well as the var keyword. The var keyword means that you create a generic type that the compiler chooses which specific type it is at compile time. What the for loops are saying is pretty straight forward, however, I think it is always best to put it into plain English for full understanding.

Here is a variable of y and it equals 0. If y is less than the map size's y minus 1, add to y. Here is a variable of x and it equals 0. If it is less than the map size's x minus 1, add to x. Each time it loops through, the map's array with an x and y value is given a new Tile sprite and its name should be unset, with the default image, and the enum's value of Unset.

Next up is the Set tiles method. We declare an index that equals 0. The for loop that says, here is a variable of y and it equals 0. If y is less than the map size's y minus 1, add to y. Here is a variable of x and it equals 0. If it is less than the map size's x minus 1, add to x. Each time it loops through, the map's array with an x and y value is given a new Tile sprite with the parameters of Tile sprites with an array of index that has a name, Tile sprites with an array of index of tile image, and Tile sprites with an array of index of tile type. add to index for each loop. Lastly, we have an if statement. If the index is greater than tile sprites' counts minus 1, the index is equal to 0. The if statement is a little bit of error handling sprinkled on.

```
1  private void AddTilesToWorld()
2  {
3      foreach (GameObject o in _tiles)
4      {
5          LeanPool.Despawn(o);
6      }
7      _tiles.Clear();
8      LeanPool.Despawn(_tileContainer);
9      _tileContainer = LeanPool.Spawn(TileContainerPrefab);
10     var tileSize = .64f;
11     var viewOffsetX = ViewPortSize.x/2f;
12     var viewOffsetY = ViewPortSize.y/2f;
13     for (var y = -viewOffsetY; y < viewOffsetY; y++)
14     {
15         for (var x = -viewOffsetX; x < viewOffsetX; x++)
16         {
17             var tX = x*tileSize;
18             var tY = y*tileSize;
19
20             var iX = x + CurrentPosition.x;
21             var iY = y + CurrentPosition.y;
22
23             if (iX < 0) continue;
24             if (iY < 0) continue;
25             if(iX > MapSize.x - 2) continue;
26             if (iY > MapSize.y - 2) continue;
27
28             var t = LeanPool.Spawn(TilePrefab);
29             t.transform.position = new Vector3(tX, tY, 0);
30             t.transform.SetParent(_tileContainer.transform);
31             var renderer = t.GetComponent<SpriteRenderer>();
32             renderer.sprite = _map[(int)x + (int)CurrentPosition.x,
33             (int)y + (int)CurrentPosition.y].TileImage;
34             _tiles.Add(t);
35         }
36     }
37 }
```

Now we have the Add tiles to world method. This one is the heart of most of the code. We have a foreach loop with the parameter of Game object from within tiles. With each loop, lean pool is to despawn an object. Lean pool will despawn the tile container and the tile container is set to Lean pool to spawn the Tile container prefab.

- The variable called tile size is set to 0.64 float.
- The variable called view offset x is set to the view port size of x divided by 2 float.
- The variable called view offset y is set to the view port size of y divided by 2 float.

The nested for loop starts with y instead of x this time. The reason being is because it is considered industry standard to do so. The variable of y is equal to the negative view offset of y. if y is less than the view offset y, add to y. The variable of x is equal to the negative view offset of x. if x is less than the view offset of x, add to x.

We create a variable called tX and it is equal to x multiplied by the tile size. We create a variable called tY and it is equal to Y multiplied by the tile size.

We create a variable called iX and it is equal to x plus the current position of x.

We create a variable called iY and it is equal to y plus the current position of y.

Now for the if statements. if iX is less than 0, continue on. if iY is less than 0, continue on. if iX is greater than the map size's x minus 2, continue on. if iY is greater than the map size's y minus 2, continue on.

We create a variable called t and it is set to Lean pool's spawn for Tile Prefab. t's transform position is equal to a new Vector 3 of tX, tY, and 0. We set the parent of the t's transform to the tile container's transform. We create a variable called renderer and tell t to get the component of Sprite Renderer. then we tell tiles to add t.

```
1  public void Start()
2  {
3      controller = GameObject.Find("Controller");
4      _map = new TileSprite[(int)MapSize.x, (int)MapSize.y];
5
6      DefaultTiles();
7      SetTiles();
8  }
9
10 private void Update()
11 {
12     AddTilesToWorld();
13 }
```

Last 2 methods and then we move on to the next section. We have the start method. We instantiate the controller object by having the Game object's find method to find an object called controller. Now we set the map to be a new Tile sprite, the parameters of it are explicitly converted map size's x and y to integer values to coincide with how the array is set up. We now call the Default tiles method and set tiles method. The update method is now up. Inside Update we simply call the Add tiles to world method.

Finally, let's look at the TilingEngine class as a whole.

```
1  using UnityEngine;
2  using System.Collections;
3  using System.Collections.Generic;
4  using Lean;
5
6  public class TilingEngine : MonoBehaviour
7  {
8      public List<TileSprite> TileSprites;
9      public Vector2 MapSize;
10     public Sprite DefaultImage;
11     public GameObject TileContainerPrefab;
12     public GameObject TilePrefab;
13     public Vector2 CurrentPosition;
14     public Vector2 ViewPortSize;
15
16     private TileSprite[,] _map;
17     private GameObject controller;
18     private GameObject _tileContainer;
19     private List<GameObject> _tiles = new List<GameObject>();
20
21     public void Start()
22     {
23         controller = GameObject.Find("Controller");
24         _map = new TileSprite[(int)MapSize.x, (int)MapSize.y];
25
26         DefaultTiles();
27         SetTiles();
28     }
29
30     private void DefaultTiles()
31     {
32         for (var y = 0; y < MapSize.y - 1; y++)
33         {
34             for (var x = 0; x < MapSize.x - 1; x++)
35             {
36                 _map[x, y] = new TileSprite("unset", DefaultImage, Tiles.Unset);
37             }
38         }
39     }
40 }
```

```
40     private void SetTiles()
41     {
42         var index = 0;
43         for (var y = 0; y < MapSize.y - 1; y++)
44         {
45             for (var x = 0; x < MapSize.x - 1; x++)
46             {
47                 _map[x, y] = new TileSprite(TileSprites[index].Name,
48                     TileSprites[index].TileImage, TileSprites[index].TileType);
49                 index++;
50                 if (index > TileSprites.Count - 1) index = 0;
51             }
52         }
53     }
54
55     private void Update()
56     {
57         AddTilesToWorld();
58     }
59 }
60 }
```

```

61     private void AddTilesToWorld()
62     {
63         foreach (GameObject o in _tiles)
64         {
65             LeanPool.Despawn(o);
66         }
67         _tiles.Clear();
68         LeanPool.Despawn(_tileContainer);
69         _tileContainer = LeanPool.Spawn(TileContainerPrefab);
70         var tileSize = .64f;
71         var viewOffsetX = ViewPortSize.x/2f;
72         var viewOffsetY = ViewPortSize.y/2f;
73         for (var y = -viewOffsetY; y < viewOffsetY; y++)
74         {
75             for (var x = -viewOffsetX; x < viewOffsetX; x++)
76             {
77                 var tX = x*tileSize;
78                 var tY = y*tileSize;
79
80                 var iX = x + CurrentPosition.x;
81                 var iY = y + CurrentPosition.y;
82
83                 if (iX < 0) continue;
84                 if (iY < 0) continue;
85                 if(iX > MapSize.x - 2) continue;
86                 if (iY > MapSize.y - 2) continue;
87
88                 var t = LeanPool.Spawn(TilePrefab);
89                 t.transform.position = new Vector3(tX, tY, 0);
90                 t.transform.SetParent(_tileContainer.transform);
91                 var renderer = t.GetComponent<SpriteRenderer>();
92                 renderer.sprite = _map[(int)x + (int)CurrentPosition.x,
93                                         (int)y + (int)CurrentPosition.y].TileImage;
94                 _tiles.Add(t);
95             }
96         }
97     }

```

```

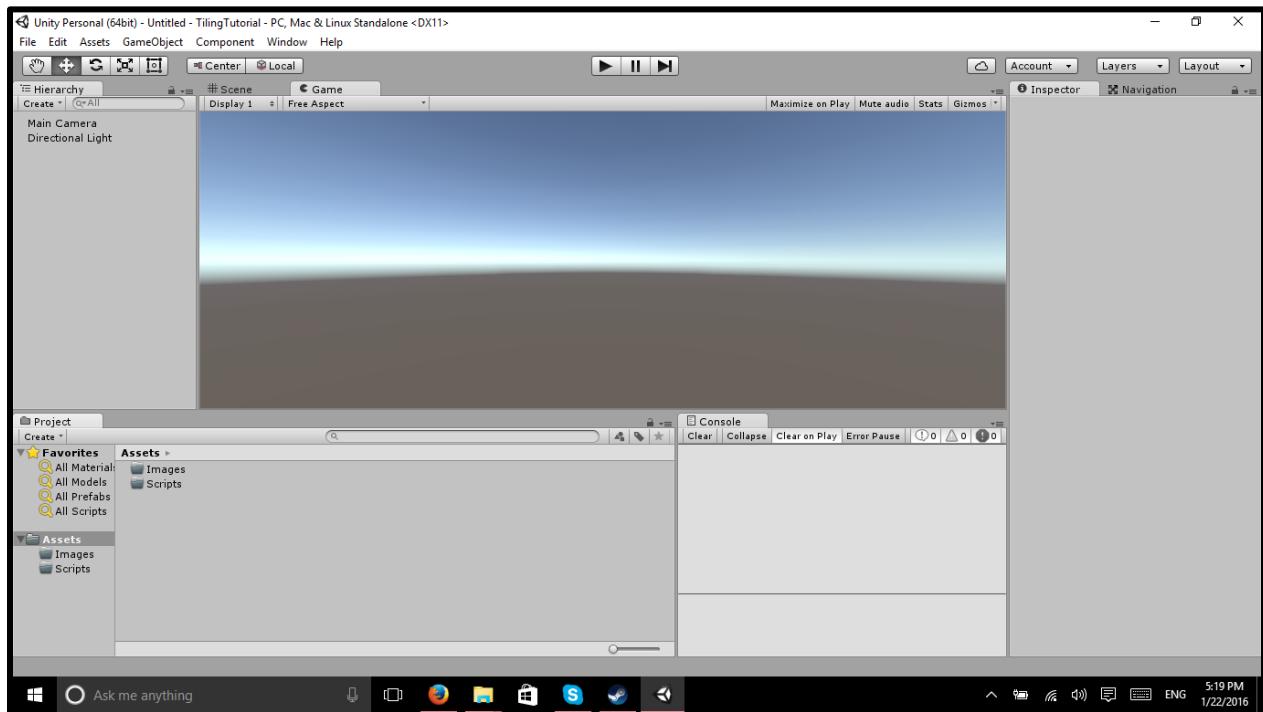
99     private TileSprite FindTile(Tiles tile)
100    {
101        foreach (TileSprite tileSprite in TileSprites)
102        {
103            if (tileSprite.TileType == tile) return tileSprite;
104        }
105        return null;
106    }
107 }

```

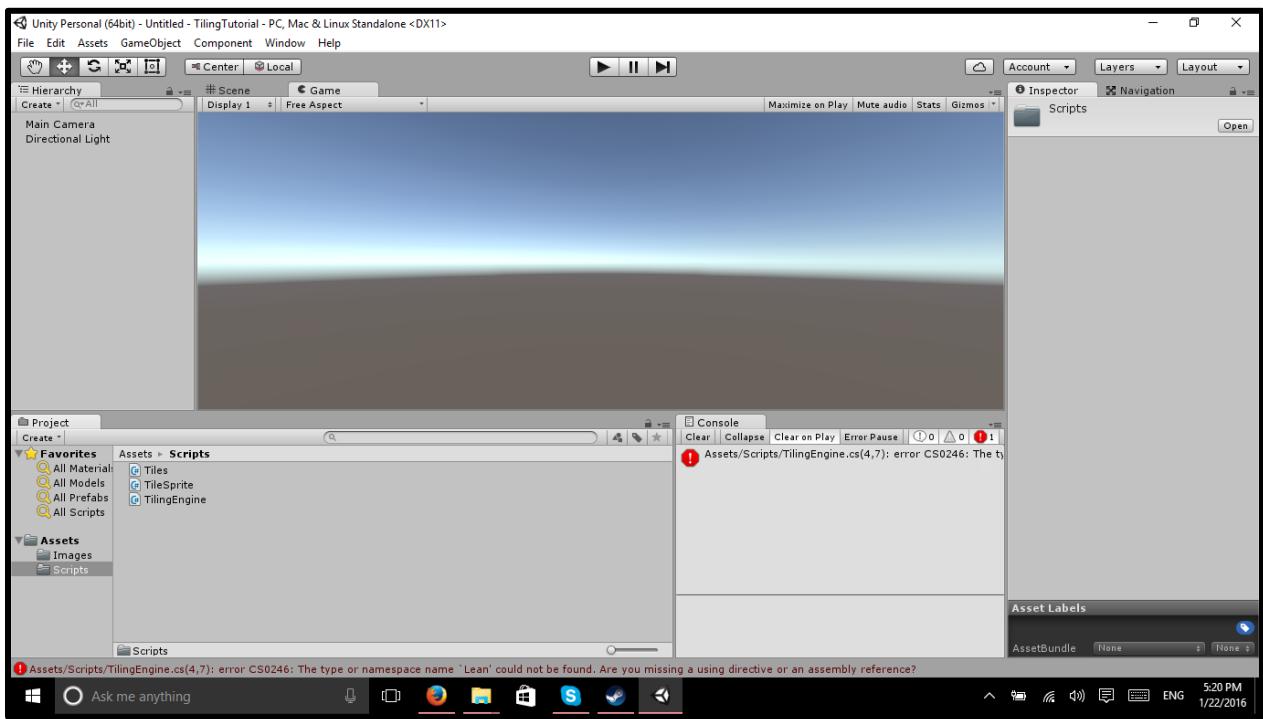
Now that we have created our scripts, we can now go into Unity and set everything up to see it in action.

SECTION 3: BUILDING THE SCENE IN UNITY3D

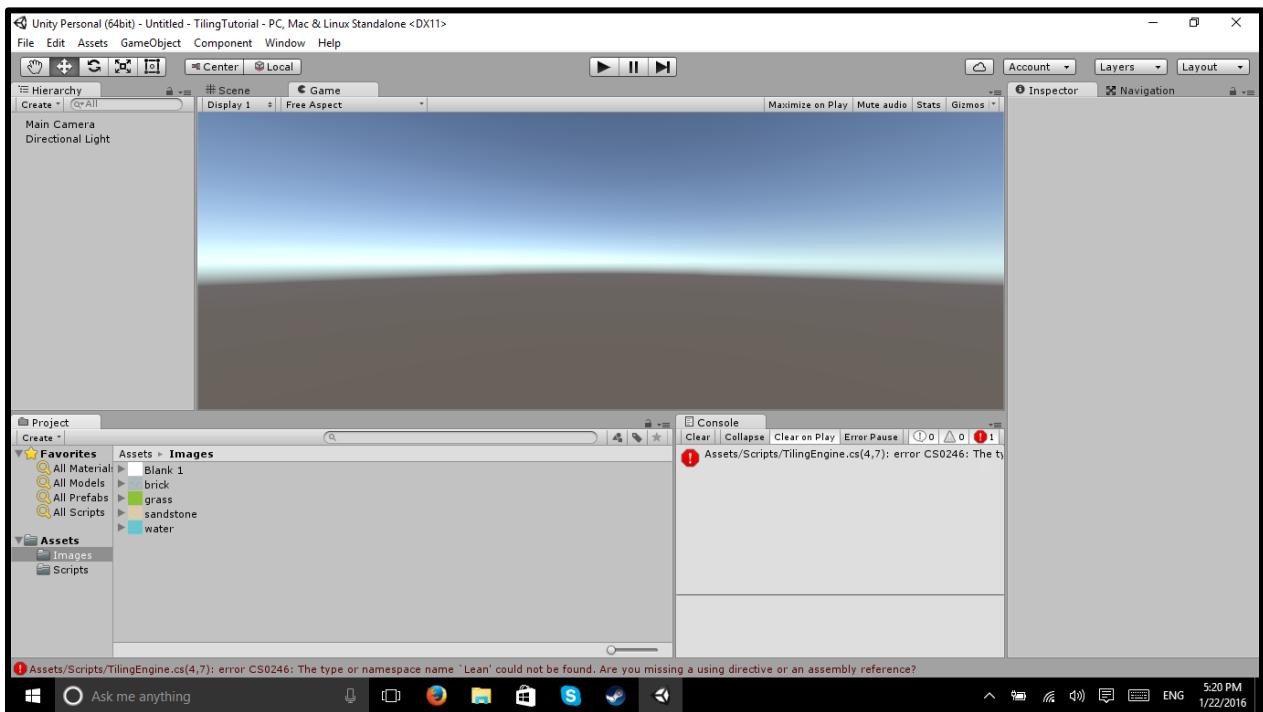
Open a new project in Unity. Set it to be a 2D project since this was designed with 2D in mind. In the assets folder, create 2 folders; One is Images and the other is scripts.



Next up, add your scripts to the scripts folder or create 3 scripts with the same names as the classes spoken about in Section 2. Either write the code by hand if you haven't done so already, or copy and paste the code in if you wrote it out while reading section 2.

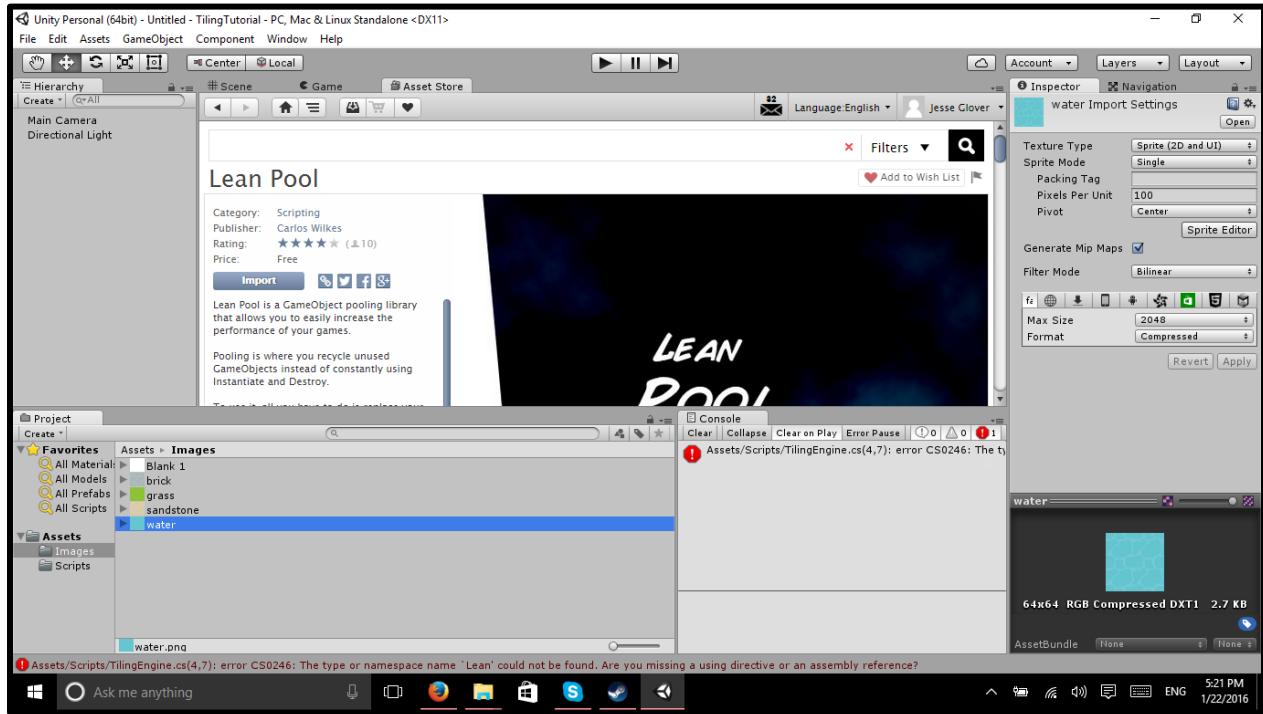


Using Microsoft Paint or whatever image editing / creation program make 5 separate images with the colours (blank, grey, green, beige, and blue) and add them to the Images folder.

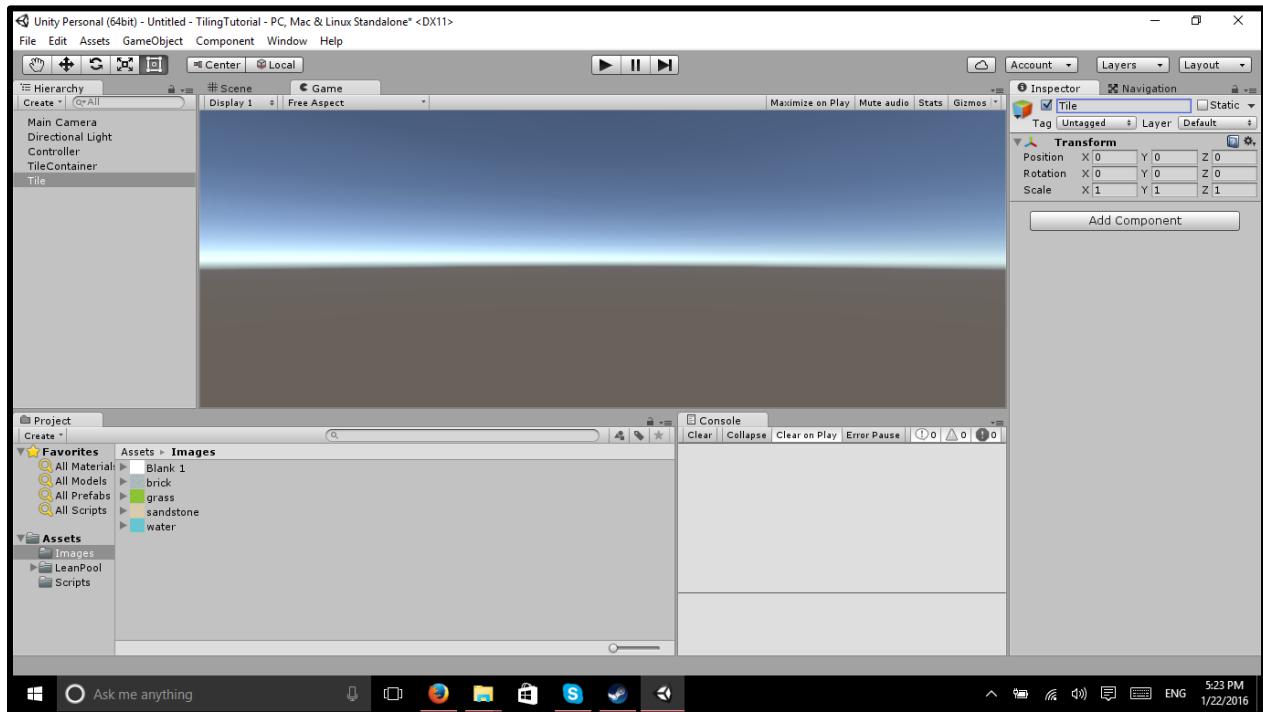


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Open the asset store and search for “Lean Pool”, download and import it into the project.



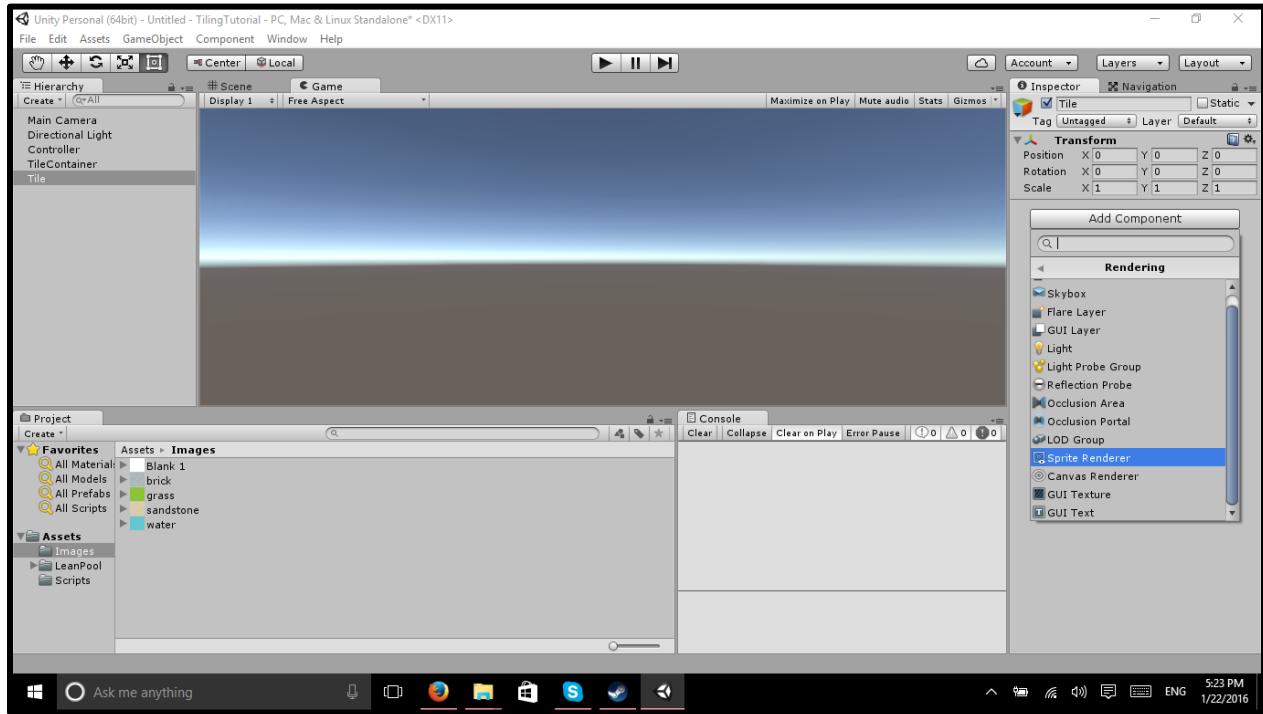
Add 3 empty game objects. Name one controller, another TileContainer, and third one Tile.



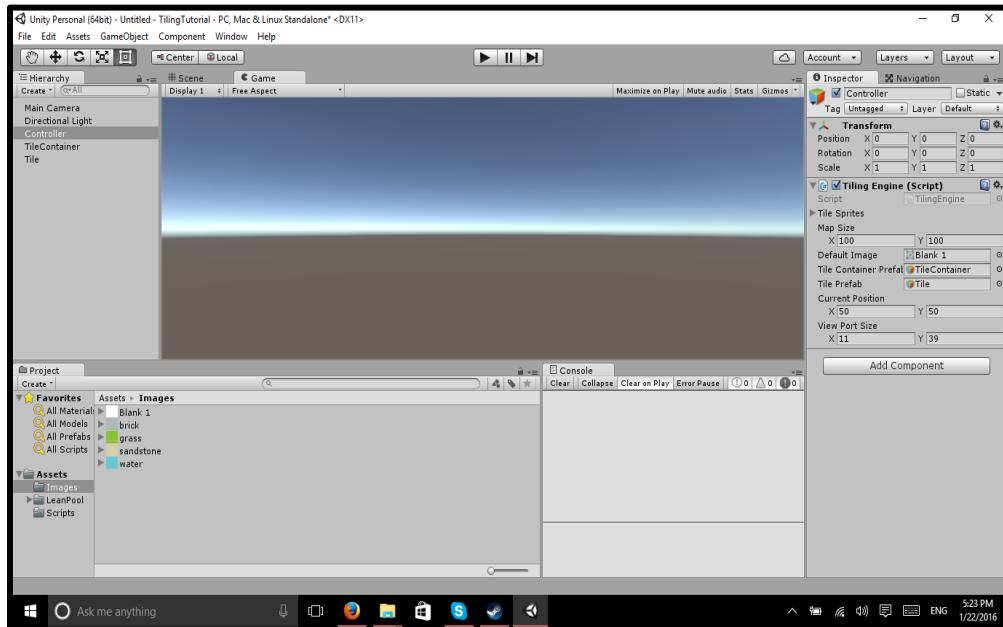
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

On the game object called tile, add a Sprite Renderer component



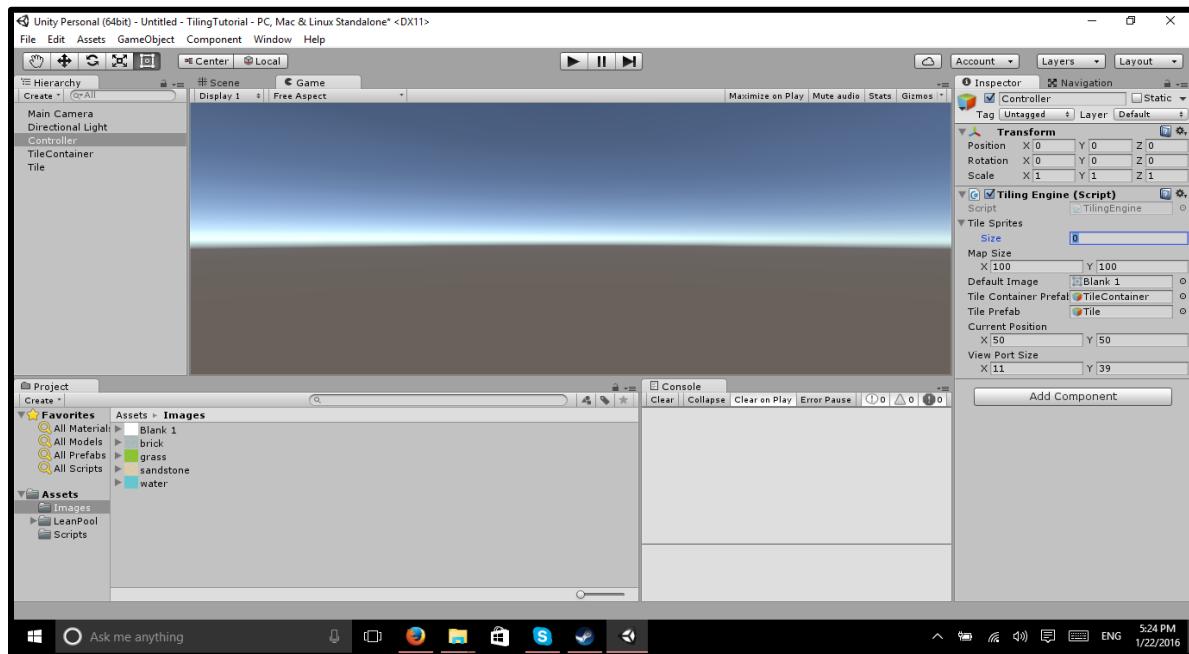
On the controller, add the Tiling Engine script onto it. make the x and y to be 100. Default image is blank 1, Tile Container prefab attach the tile container object, Tile Prefab attach the tile object, current position x and y is 50. View Port size x is 11 and y is 39.



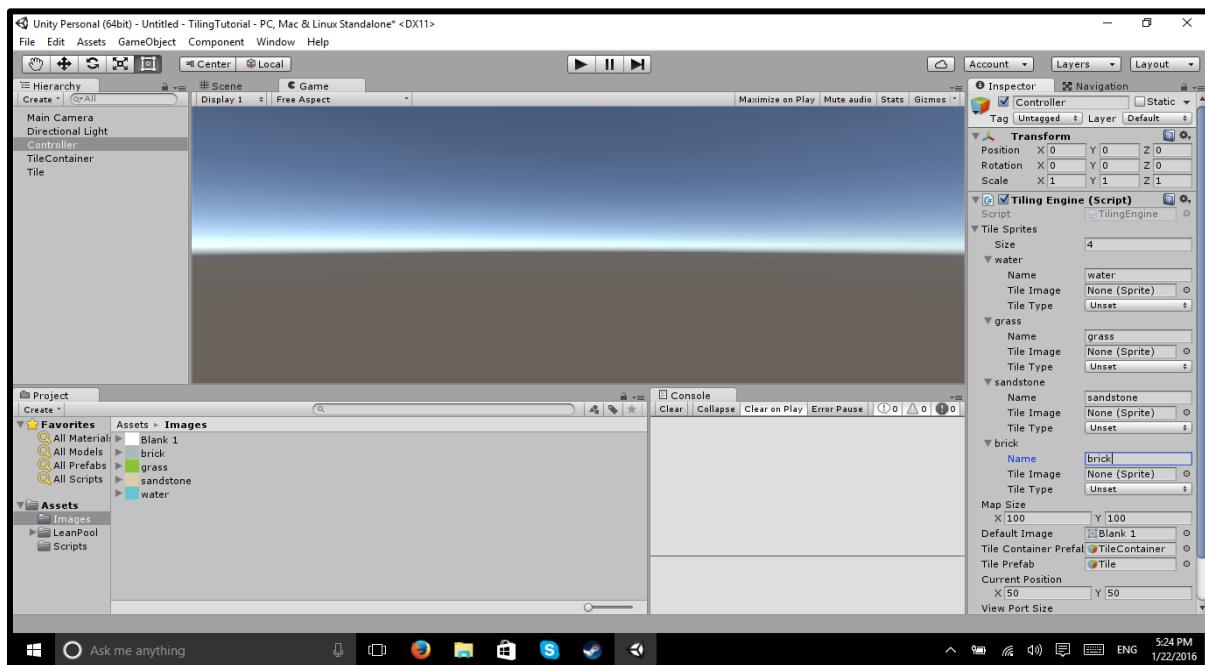
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Minus down the Tile Sprites arrow just under the Tiling Engine script.



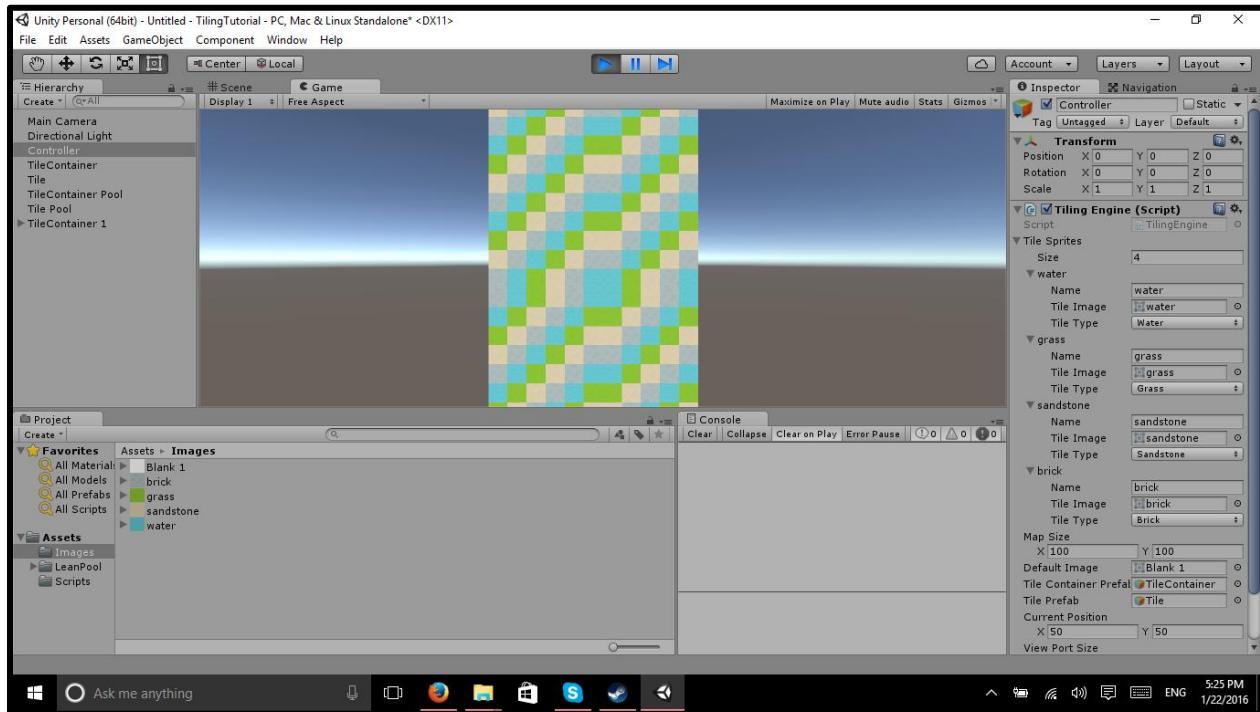
Change the size to 4. Name each one to what it should look like (according to the enums), set the image according to the name, and set the tile type to be what you named it.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Run the project and You now have a 2D top down tile map displayed on the game screen.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

How to Build 3D Algorithms with Unity3D

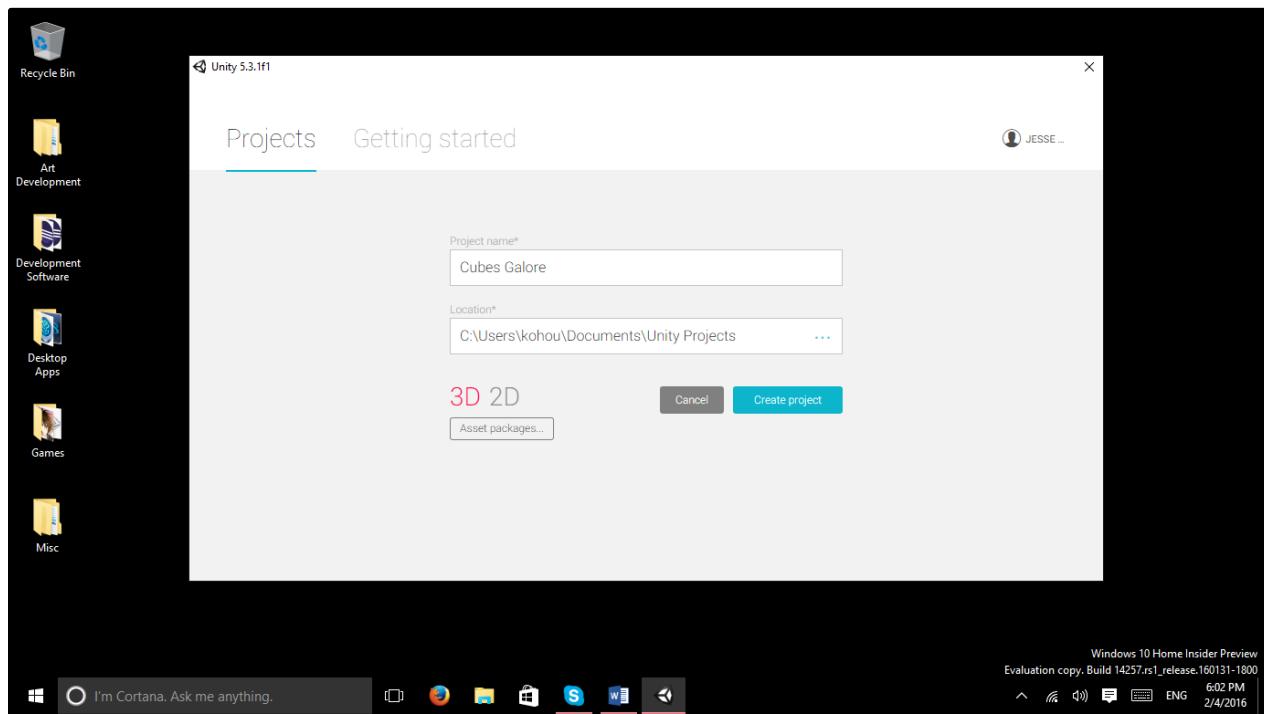
By Jesse Glover

Project download is located [here](#).

This week, we will again talk about algorithms with Unity3D. The main focus will be to take a main idea and build it. We will start with breaking the concept down into smaller, more manageable sections and build up to the main idea. So, the idea we want to have in the end is a cube that has random colors that we can save and load. We also want to dynamically generate these from a single to multiple cube prefabs. I hope you guys are ready because this is going to be a very exciting tutorial.

SECTION 1: BUILDING THE UI

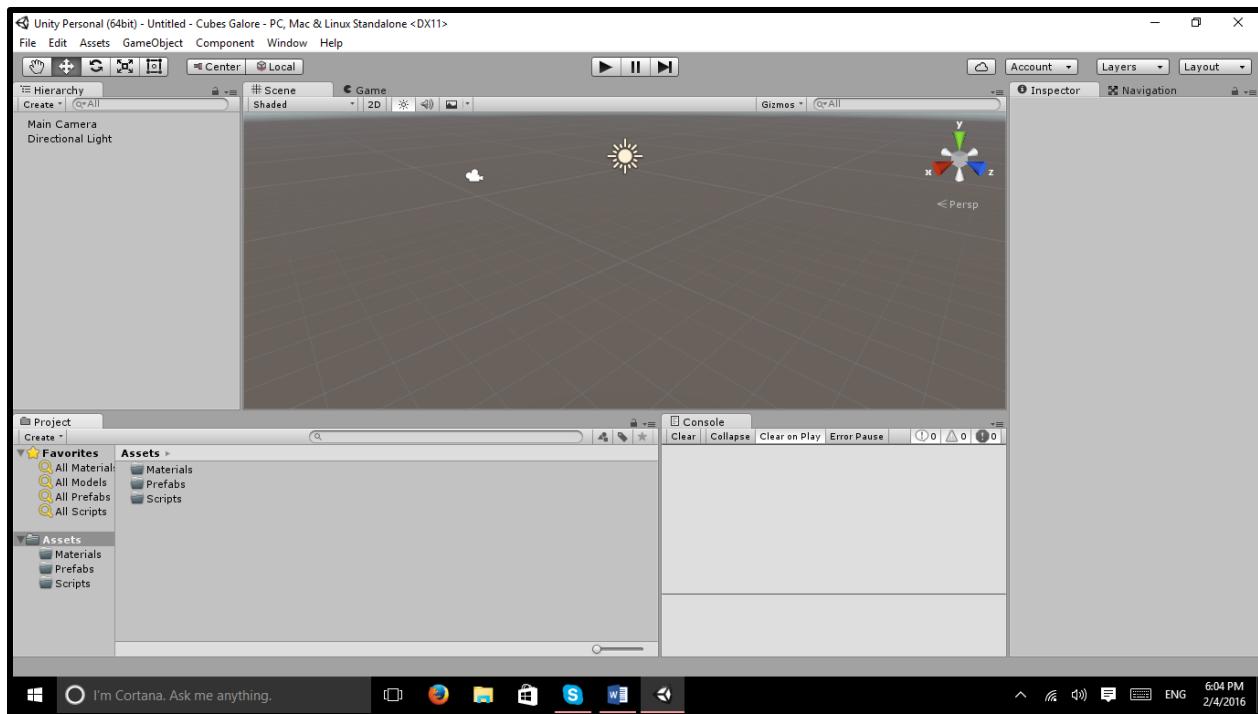
In this section, we will be using the Unity editor to create the basic prefab that has our cube and materials. So, let's start by creating a new project that is set to be in 3D mode and calling it "Cubes Galore".



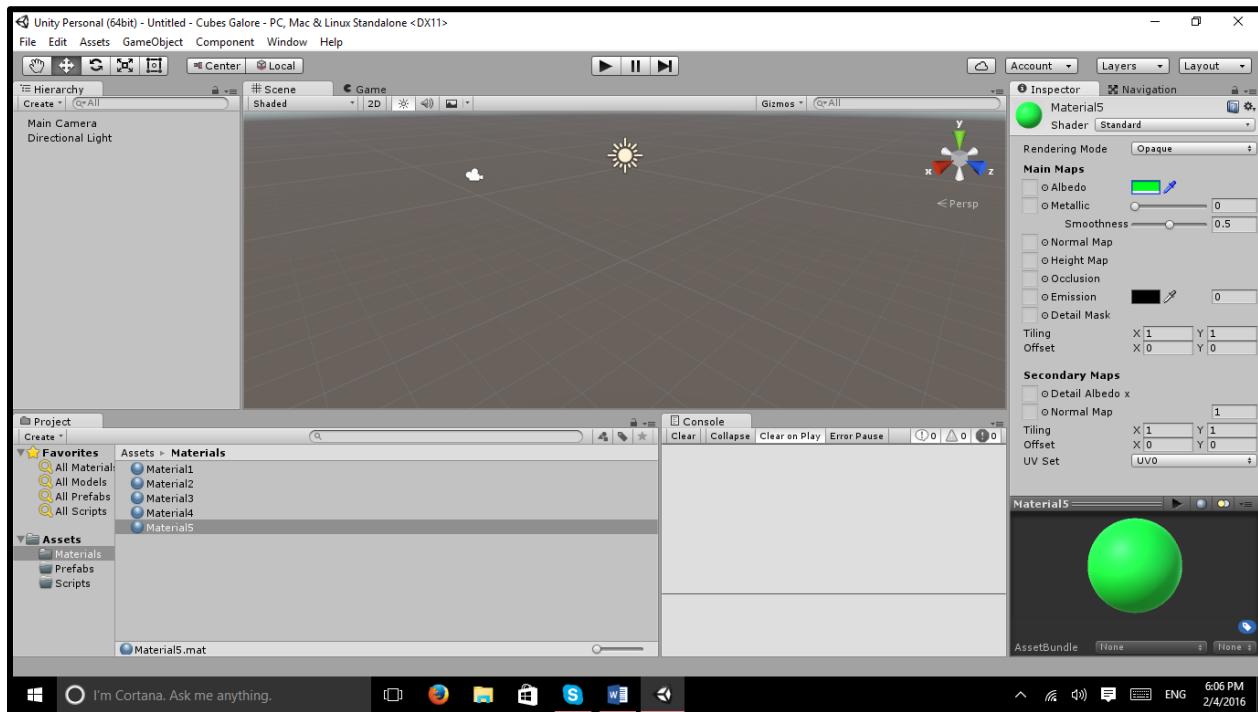
Next up, we need a few folders. Prefabs, Scripts, and Materials are the folder names.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved



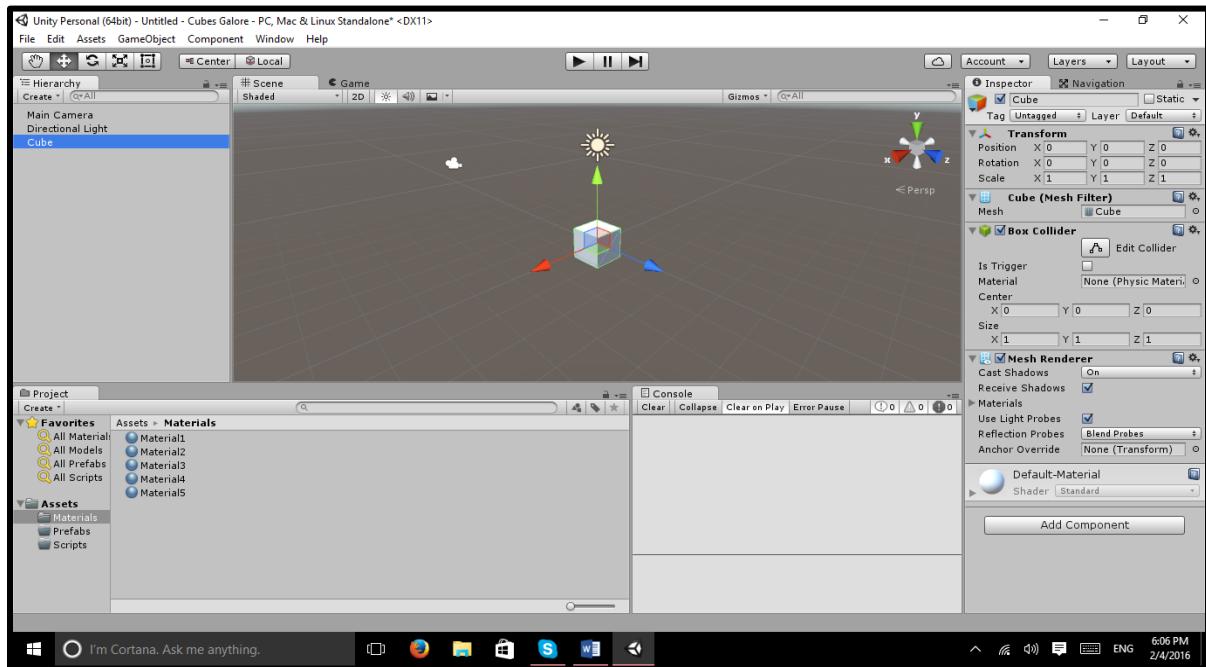
Let's make about 5 different colored materials. You can follow along with the colors I used, or create your own.



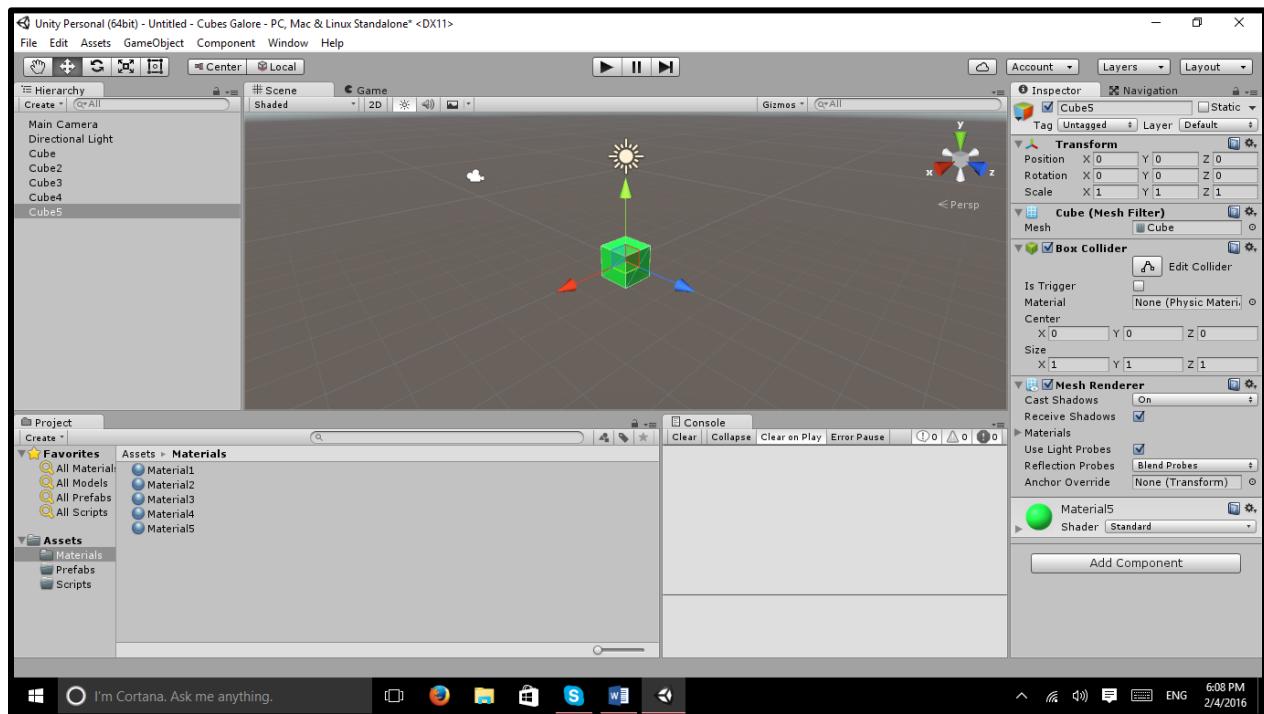
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Now that we have our materials created, lets add a cube to the scene.



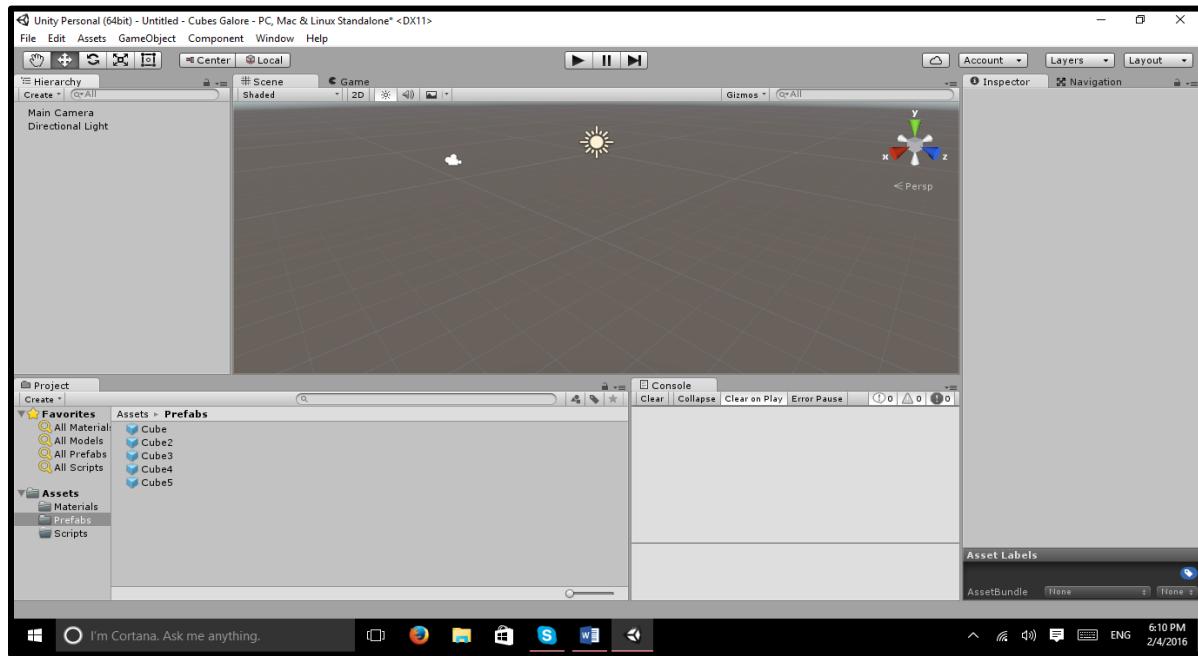
Duplicate the cube 4 times and apply one material to each one.



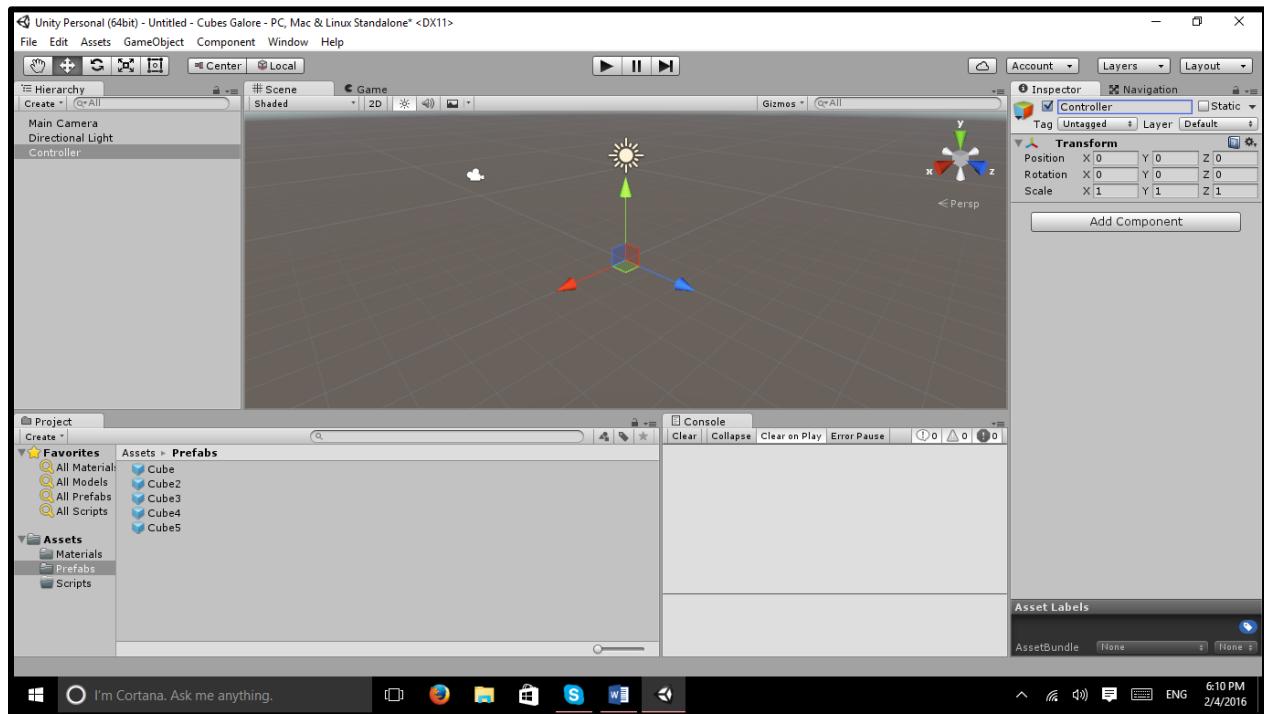
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Now, drag all 5 cubes into the prefabs folder and remove them from the hierarchy pane.



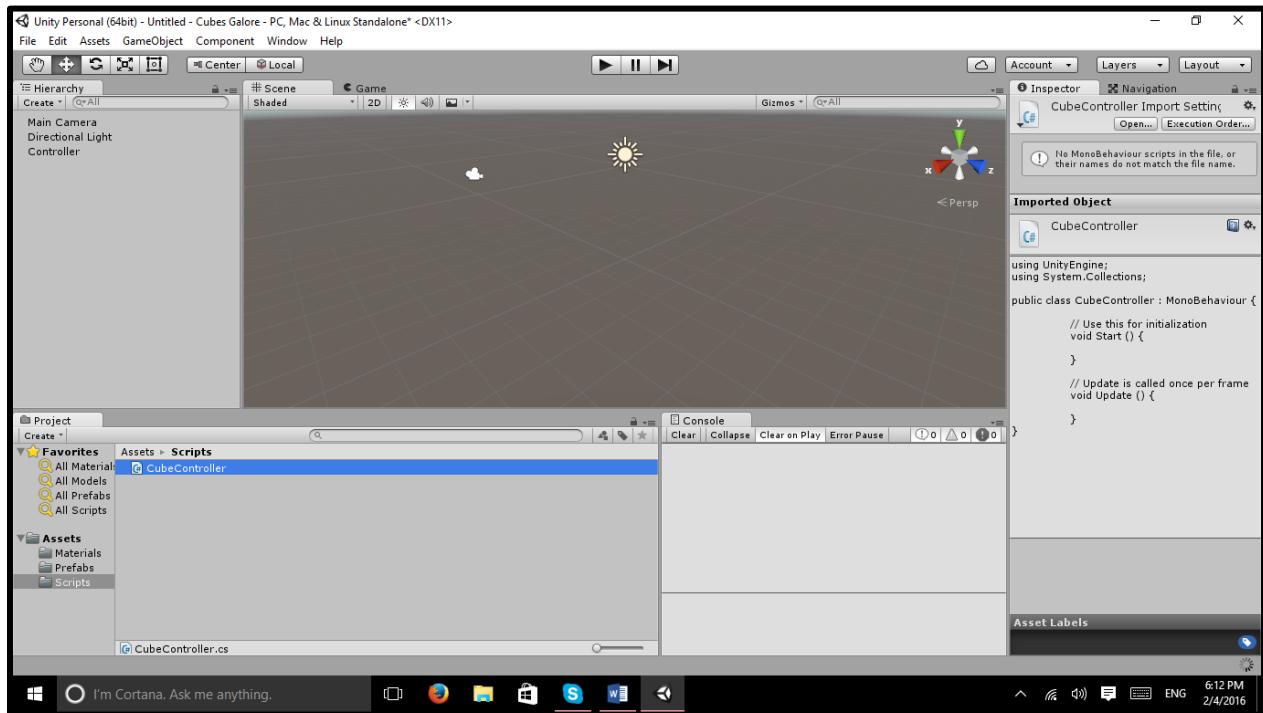
Create an empty game object and call it controller.



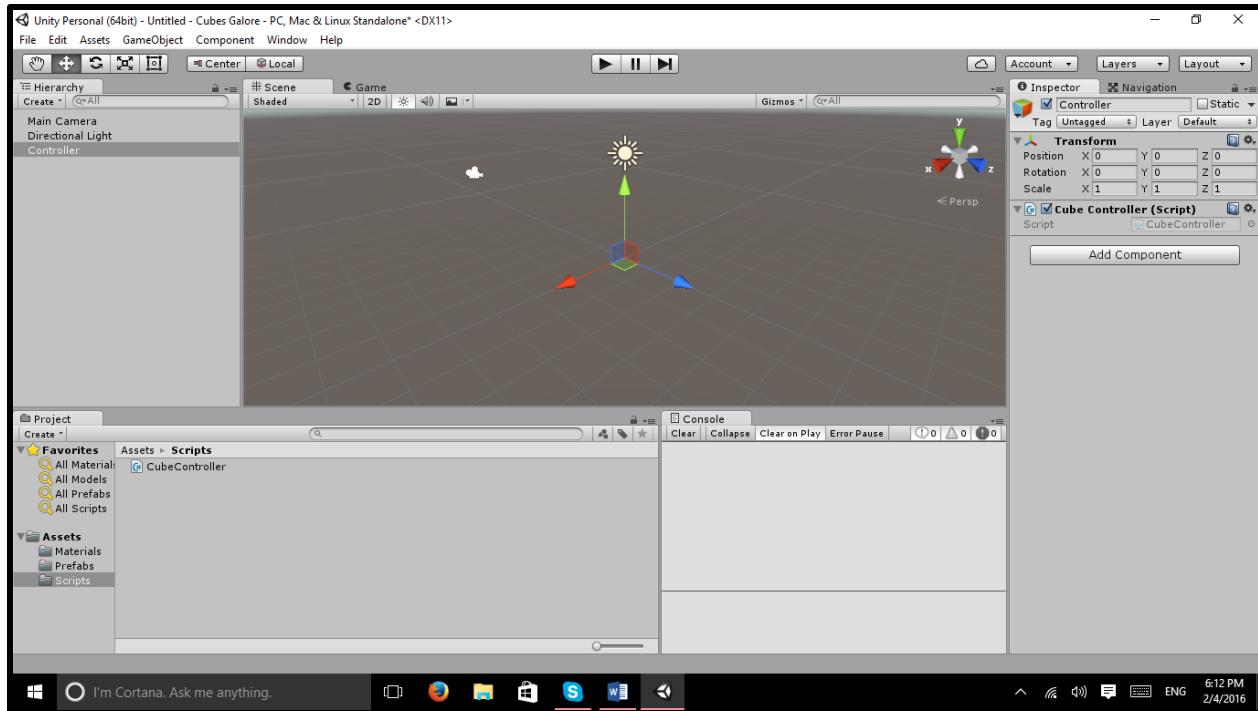
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Now, we need to make a script in the scripts folder and call it "CubeController".



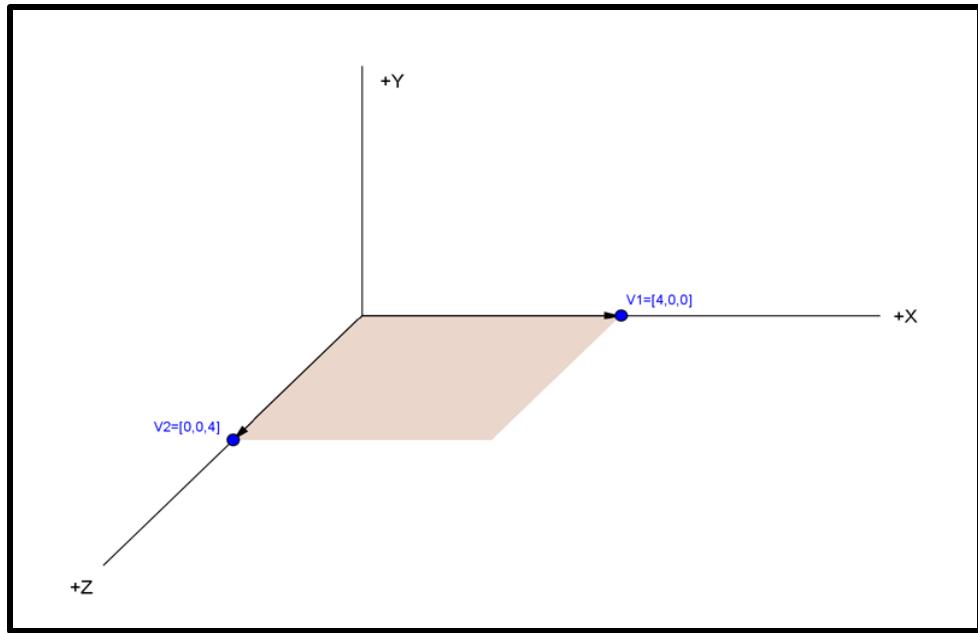
The last thing to do for this section is to go ahead and add the CubeController script onto the Controller object.



Section 1 is now complete; it doesn't look like much but this is most assuredly the basics for what we hope to accomplish.

SECTION 2: THE FIRST ALGORITHM

In this section, we are going to take a single cube prefab and have it duplicate itself multiple times to form a square. To do this, we need a little bit of information. That information would be in terms of positions. What is the Vector 3 position of the left and right sides of the square, as well as the forward and back most position? To answer this, we need a little bit of geometry to answer the question. A square by definition has 4 equal sides.



Now, we want our square's y position to be at 0. The reason for this is because we don't want any height difference for this first square.

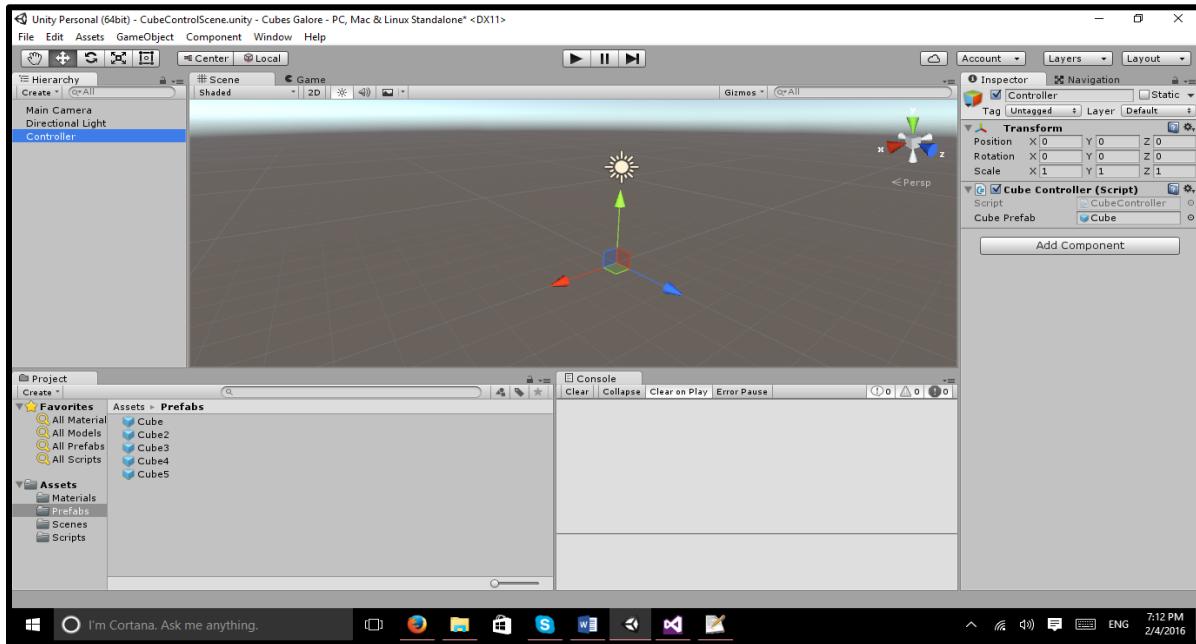
The x position will start at -3 and move to 2. So, this will make the basic square be 5 long. So, we need to do the same for the depth. Which leads us to the Z position. The z position will be from 0 to 5. Now that we have the basic idea of what we need, let's write the algorithm.

```

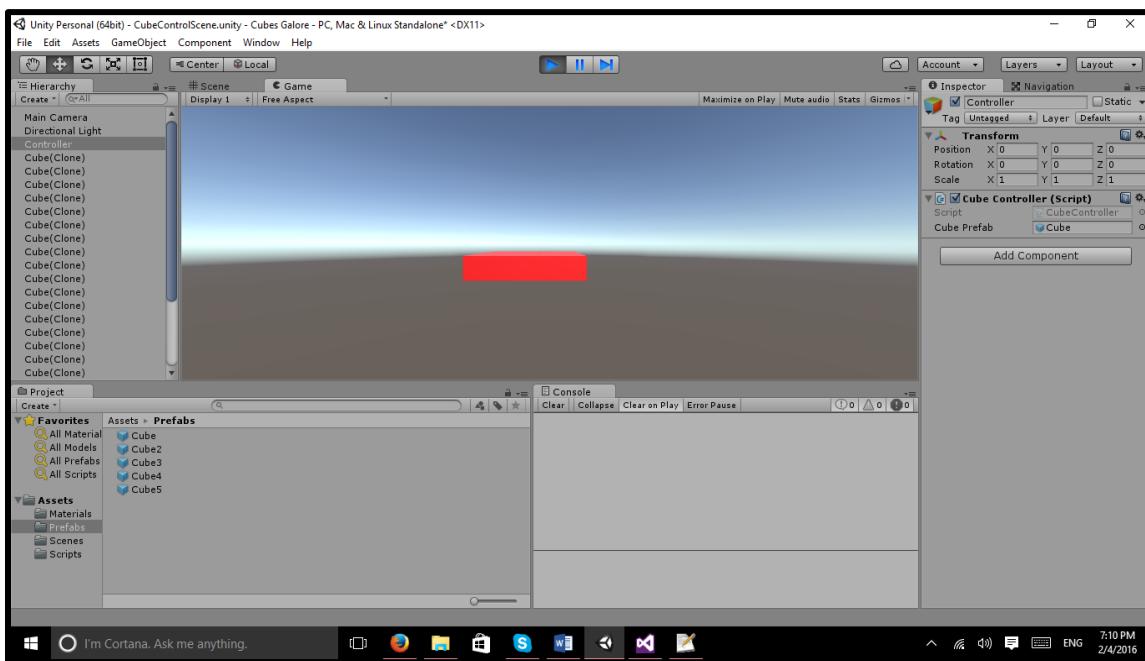
1  using UnityEngine;
2  using System.Collections;
3
4  public class CubeController : MonoBehaviour
5  {
6      public GameObject CubePrefab;
7
8      public void generateCubeSquare()
9      {
10         for (float x = -3; x < 2; x++)
11         {
12             for (float z = 0; z < 5; z++)
13             {
14                 Instantiate(CubePrefab, new Vector3(x, 0, z), Quaternion.identity);
15             }
16         }
17     }
18
19     public void Start()
20     {
21         generateCubeSquare();
22     }
23 }
```

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

With the algorithm, we wrote the algorithm to coincide with the x and z values to make it easier to read. Now, we need to add a cube prefab to the controller.



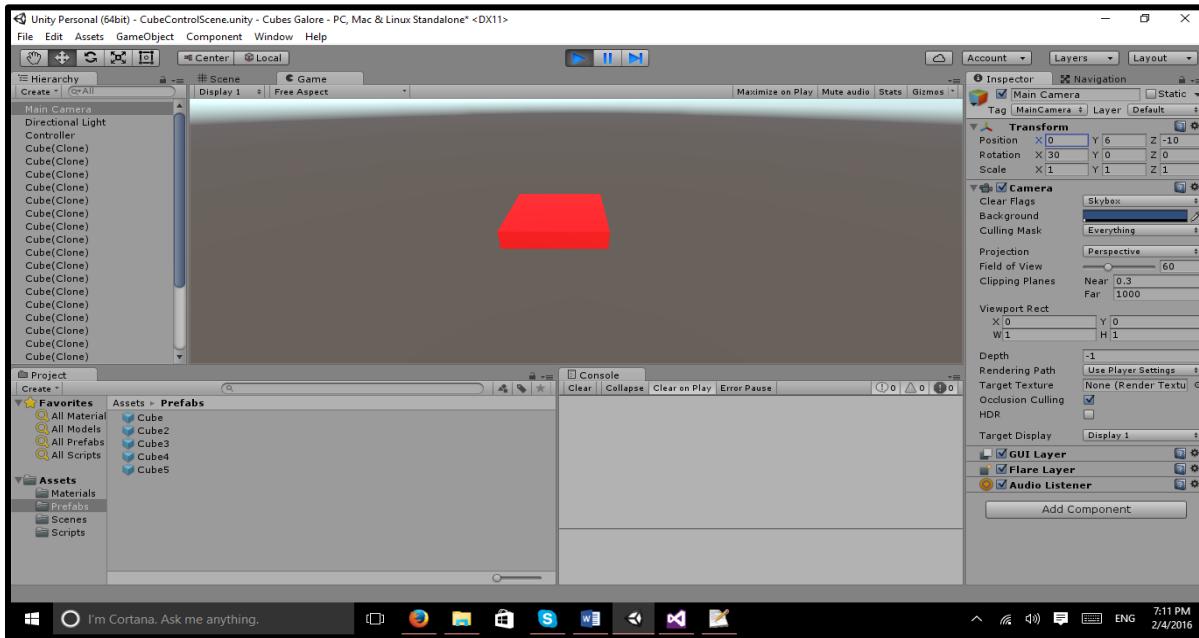
Let's take a look at the results.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Hmm, let's change the camera's view angle so we can see the full results. We will change the Y position of the camera to be 6. Now, let's change the X rotation to be 30.



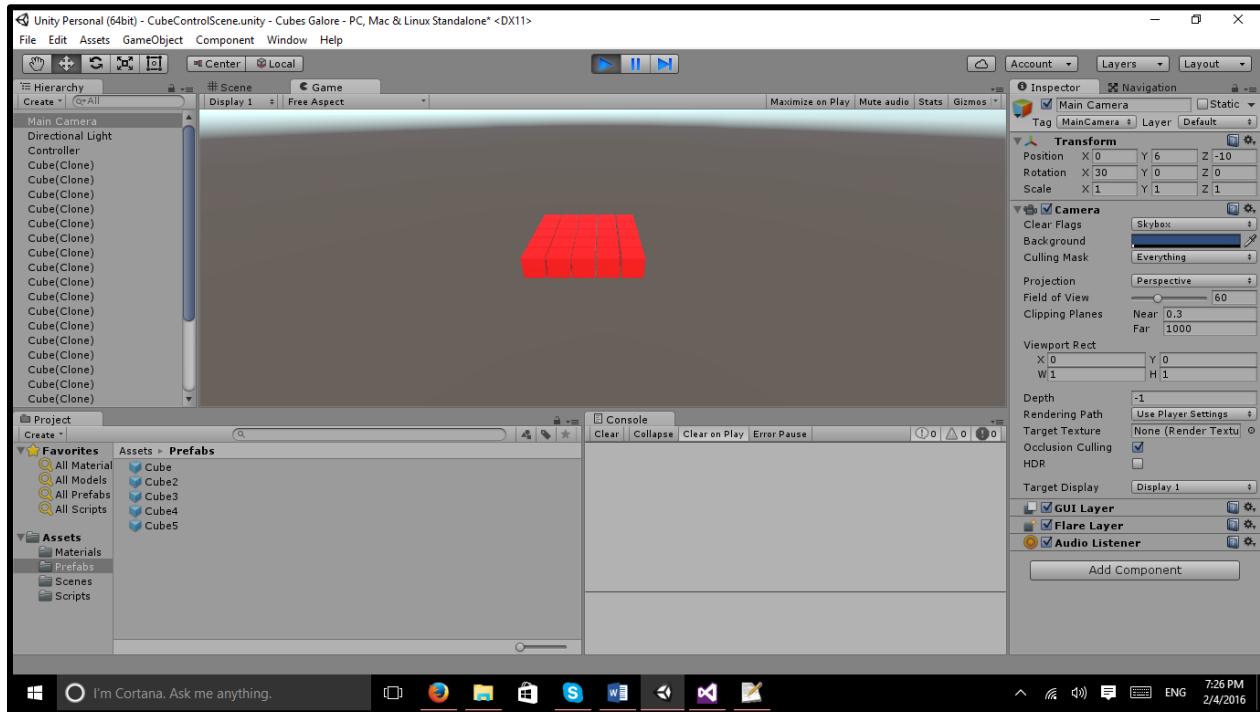
Let's do a slight modification to add a slight modification to this, the modification is to add a small space in between each cube.

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class CubeController : MonoBehaviour
5 {
6     public GameObject CubePrefab;
7
8     public void generateCubeSquare()
9     {
10         for (float x = -3; x < 2; x += 1.08f)
11         {
12             for (float z = 0; z < 5; z += 1.08f)
13             {
14                 Instantiate(CubePrefab, new Vector3(x, 0, z), Quaternion.identity);
15             }
16         }
17     }
18
19     public void Start()
20     {
21         generateCubeSquare();
22     }
23 }
```

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

What we did was changed the increment portion of the for loop. Instead of adding 1 with each loop, we add 1.08 to the current z number.



As you can see, it looks a little bit more like a square grid; Which in turn looks a little more uniform and clean.

SECTION 3: ALGORITHM WITH A LIST OF PREFABS

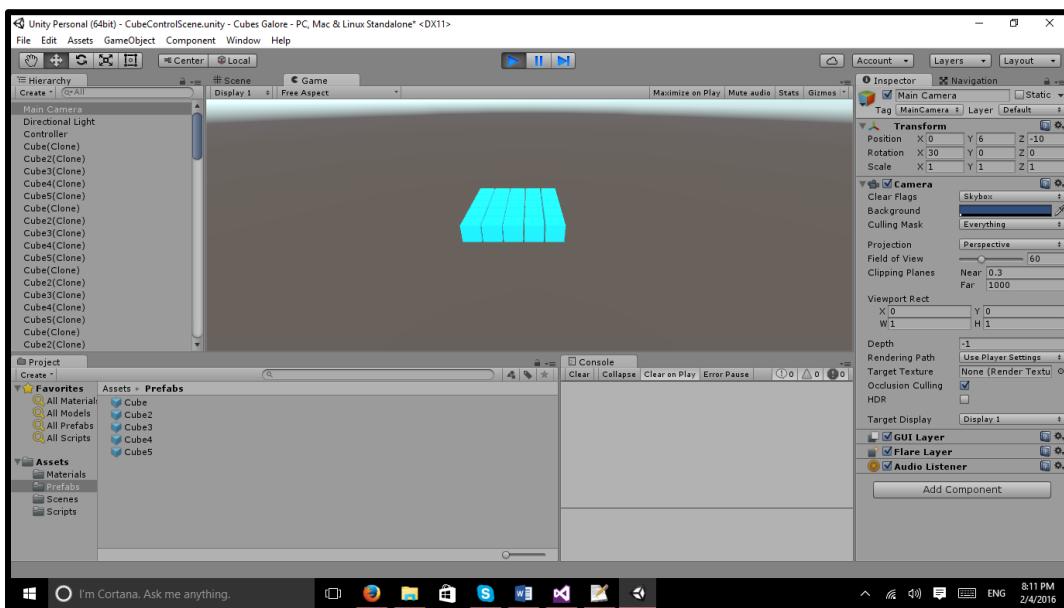
In this section, we will randomize the colors of the cubes in the square. This is most certainly going to cause some frustrations to the beginning developer. So, the way I will do this is show the incorrect way to do it first and explain why it does not work as intended and then show the correct way to do it with an explanation.

THE WRONG WAY:

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

```
1 using UnityEngine;
2 using System.Collections;
3 using System.Collections.Generic;
4
5 public class CubeController : MonoBehaviour
6 {
7     public List<GameObject> CubePrefabList;
8
9     public void generateColoredCubeSquare()
10    {
11        var index = CubePrefabList.Count;
12
13        for (float x = -3; x < 2; x += 1.08f)
14        {
15            for (float z = 0; z < 5; z += 1.08f)
16            {
17                foreach(var item in CubePrefabList)
18                {
19                    Instantiate(item, new Vector3(x, 0, z), Quaternion.identity);
20                }
21            }
22        }
23    }
24
25
26    public void Start()
27    {
28        generateColoredCubeSquare();
29    }
30}
31}
```

Let's take a look at the results.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

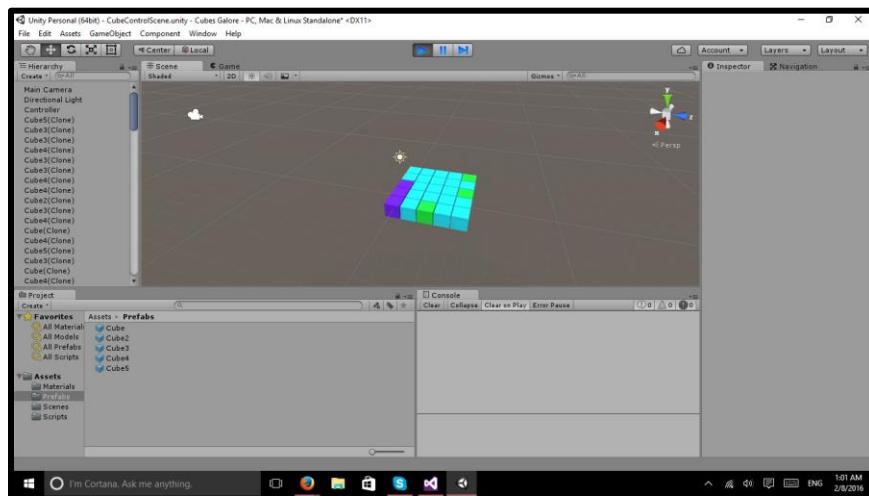
© Zenva Pty Ltd 2021. All rights reserved

The reason why this does not work as intended is because of the way the code is written, it tells the compiler to put a cube in the same spot as ones that are already placed. There are quite a few ways to make this mistake as well as to rectify the problem.

THE CORRECT WAY:

```
1  using UnityEngine;
2  using System.Collections;
3  using System.Collections.Generic;
4
5  public class CubeController : MonoBehaviour
6  {
7      public List<GameObject> CubePrefabList;
8
9      public void generateColoredCubeSquare()
10     {
11         for (float x = -3; x < 2; x += 1.08f)
12         {
13             for (float z = 0; z < 5; z += 1.08f)
14             {
15                 Instantiate(CubePrefabList[Random.Range(0, CubePrefabList.Count)],
16                             new Vector3(x, 0, z), Quaternion.identity);
17             }
18         }
19     }
20
21
22     public void Start()
23     {
24         generateColoredCubeSquare();
25     }
}
```

Let's take a look at the results



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

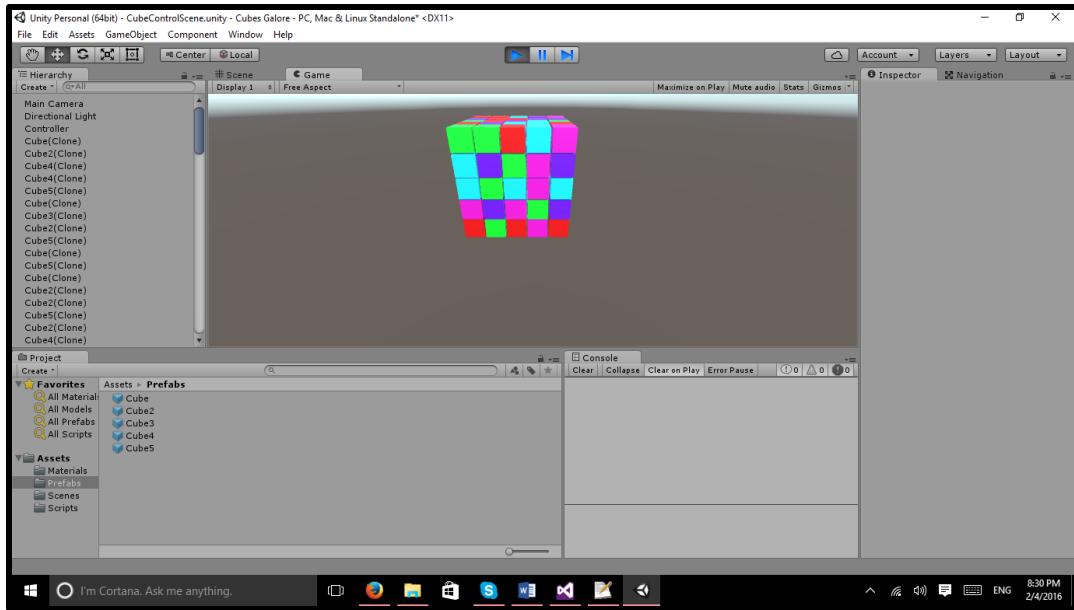
As I said in the wrong way subsection, there are many ways to write this correctly. I chose a way that is simple to use and easy to read. With this method, we tell the compiler to put a cube in every open position and randomize which cube is in each position.

SECTION 4: MAKING A 3D CUBE

This section we will take the existing code and modify it some in order to create a cube. To do this, all we need to do is to apply a little bit more geometry to the existing geometry we have already created. We have our x and z values already done, now we need to do the Y values. Since we know we want to start at 0, that means we need to have it end at 5. This should allow us to have a $5 \times 5 \times 5$ cube, in other words a length of 5, a width of 5, and a depth of 5.

```
1 using UnityEngine;
2 using System.Collections;
3 using System.Collections.Generic;
4
5 public class CubeController : MonoBehaviour
6 {
7     public List<GameObject> CubePrefabList;
8
9     public void generateColoredCubeSquare()
10    {
11        for (float x = -3; x < 2; x += 1.08f)
12        {
13            for (float z = 0; z < 5; z += 1.08f)
14            {
15                for (float y = 0; y < 5; y += 1.08f)
16                {
17                    Instantiate(CubePrefabList[Random.Range(0, CubePrefabList.Count)],
18                                new Vector3(x, y, z), Quaternion.identity);
19                }
20            }
21        }
22    }
23
24    public void Start()
25    {
26        generateColoredCubeSquare();
27    }
28 }
29 }
```

Here are the results



What we did to accomplish this was we nested a for loop 3 times to handle each value we wanted to manipulate. X, Y, and Z values to be exact, and the code itself did not change much at all. The thing to always remember is to keep it as simple as possible when you want to add a new feature or change existing code. And you should always try to have the code to be as readable as possible.

SECTION 5: RANDOMIZING THE CUBES VARIANT

In this section, we will be rebuilding the controller class to do several different things than the last. Rather than hard coding the x, y, and z position of the Vector 3, we will make it so we can change it in the editor. The reason for this is to allow for more tightly controlled testing as well as be able to see the results at runtime without needing to modify our code. The next change will be to create the rgb values and randomize those instead of using material prefabs. So, let's look at the code all the way up to the start method.

```

1 using UnityEngine;
2 using System.Collections;
3 using System.Collections.Generic;
4 using System.IO;
5
6 public class Controller : MonoBehaviour
7 {
8     public GameObject CubePrefab;
9     public Vector3 GridSize;

```

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

We have our public cube prefab which we brought back from our first algorithm and we have a public Vector 3 called GridSize. The CubePrefab was brought back because we don't want to use a list of prefabs since we can use a single one and modify the colors of that one. The GridSize Vector 3 is here because we want to be able to modify the x y and z values in the editor.

```
1  public void Start()
2  {
3      CreateCubeGrid();
4  }
```

The start method has 1 item within it. We are calling a method called CreateCubeGrid. At this point, we should see the compiler complain because we don't have a method for CreateCubeGrid yet.

```
1  /// <summary>Creates a new cube grid if one doesn't exist</summary>
2  public void CreateCubeGrid()
3  {
4      for (var a = 0; a < GridSize.x; a++)
5      {
6          for (var b = 0; b < GridSize.y; b++)
7          {
8              for (var c = 0; c < GridSize.z; c++)
9              {
10                  var cube = (GameObject)Instantiate(CubePrefab, new Vector3(a, b, c),
11                                              Quaternion.identity);
12                  var renderer = cube.GetComponent<MeshRenderer>();
13                  var red = Random.Range(0f, 1f);
14                  var green = Random.Range(0f, 1f);
15                  var blue = Random.Range(0f, 1f);
16                  var color = new Color(red, green, blue);
17                  renderer.material.color = color;
18              }
19          }
20      }
21  }
```

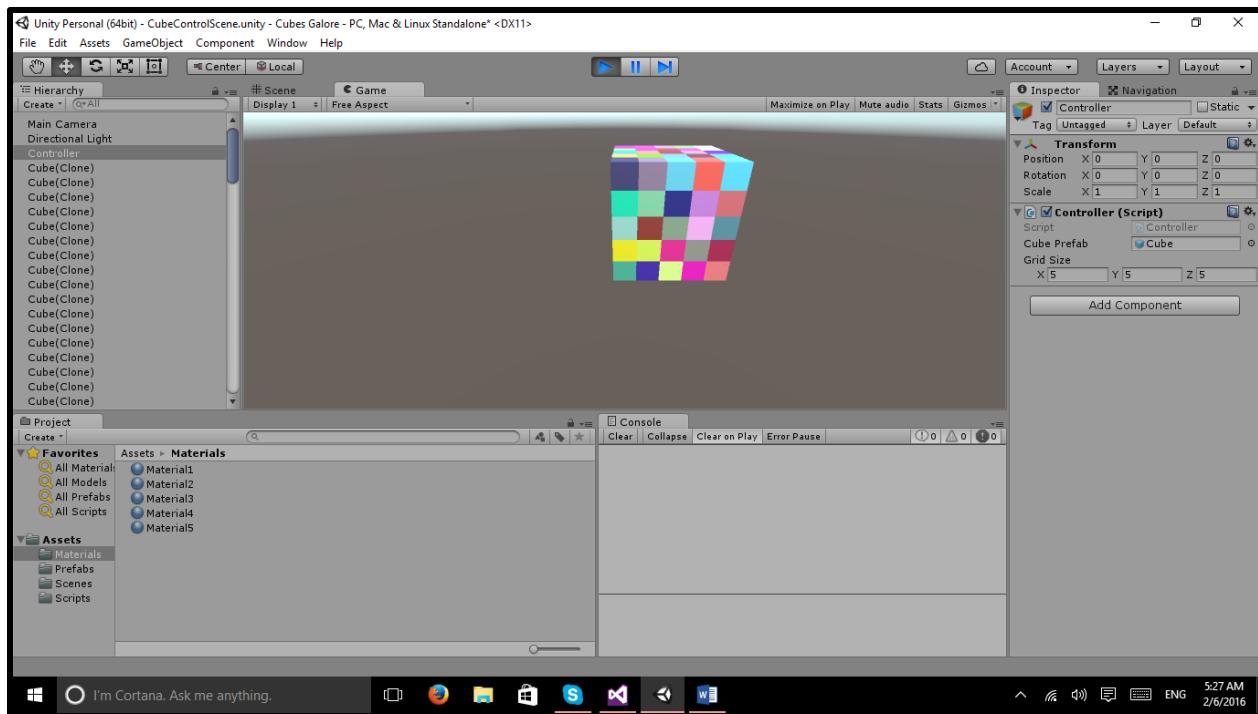
Now we have the CreateCubeGrid method and it is rather lengthy, so let's go over it in small sections.

```
1  /// <summary>Creates a new cube grid if one doesn't exist</summary>
2  public void CreateCubeGrid()
3  {
4      for (var a = 0; a < GridSize.x; a++)
5      {
6          for (var b = 0; b < GridSize.y; b++)
7          {
8              for (var c = 0; c < GridSize.z; c++)
9              {
10             }
11         }
12     }
13 }
```

The for loops here should be fairly straight forward. Var a, b, and c are placeholders for the GridSize's x, y, and z variables and we increment by 1 with each loop.

```
1 var cube = (GameObject)Instantiate(CubePrefab, new Vector3(a, b, c), Quaternion.identity);
2 var renderer = cube.GetComponent<MeshRenderer>();
3 var red = Random.Range(0f, 1f);
4 var green = Random.Range(0f, 1f);
5 var blue = Random.Range(0f, 1f);
6 var color = new Color(red, green, blue);
7 renderer.material.color = color;
```

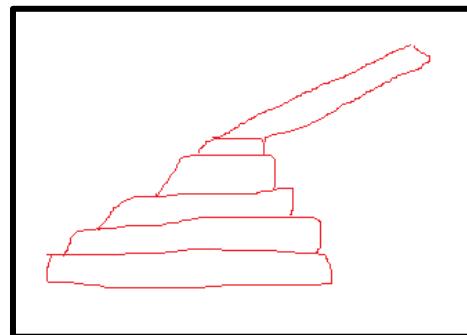
We create a variable called cube to instantiate the cubePrefab with the vector 3 positions of a, b, and c. We set the Quaterion at identity because we don't want any rotations on the cubes. In order to manipulate the colors of the cube, we need a reference to the MeshRender which is what our renderer variable does. Red, Green, and Blue have a random range of 0 to 1, this allows us to specify which color we want to pick. Finally, we have a color variable that instantiates the Color class which we specify as red, green, and blue.



SECTION 6: TURNING THE CUBE INTO A HALLWAY AND STAIRS

In this section, we will go about taking everything we have learned from the other sections and create a little walkway and some stairs leading upwards to another walkway. Keeping with the motif, we will still be using a cube for this.

The other sections were really good for a basic start, but really couldn't be used for too much besides an idea for a puzzle game. So, I think it is time to take what we have learned, and apply it to create something more useful for game development. Let's make a 1 wide, by 5 long corridor that leads to stairs that are 5 tiers high.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Using my super amazing art skills to draw what we kind of want to be the end results, we can see exactly what we need to do. But how should we go about this? Should we put everything into a single method or have 3 separate methods? If we follow the KISS method (Keep It Stupidly Simple), we should have 3 methods that take care of a single task. Let's look at the finished results of the algorithms.

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class StairWayToWalkWay : MonoBehaviour
5  {
6      public GameObject CubePrefab;
7
8      public void Start()
9      {
10         CreateStairs();
11         CreateWalkWay();
12         CreateUpperWalkWay();
13     }
14
15     public void CreateStairs()
16     {
17         int y = 0;
18         int z = 5;
19         for (int x = 5; x > 0; x--)
20         {
21             if (x == 5)
22             {
23                 y = 0;
24                 GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
25             }
26
27             if (x == 4)
28             {
29                 y = 1;
30                 GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
31             }
32         }
33
34         if (x == 3)
35         {
36             y = 2;
37             GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
38         }
39
40         if (x == 2)
41         {
42             y = 3;
43             GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
44         }
45
46         if (x == 1)
47         {
48             y = 4;
49             GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
50         }
51     }
52 }
```

```
33         if (x == 3)
34         {
35             y = 2;
36             GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
37         }
38
39         if (x == 2)
40         {
41             y = 3;
42             GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
43         }
44
45         if (x == 1)
46         {
47             y = 4;
48             GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
49         }
50     }
51 }
```

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

```

51         if (x == 0)
52     {
53         y = 5;
54         GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
55     }
56 }
57 }
58
59 public void CreateWalkWay()
60 {
61     for (int x = 5; x < 10; x++)
62     {
63         GameObject.Instantiate(CubePrefab, new Vector3(x, 0, 5), Quaternion.identity);
64     }
65 }
66
67 public void CreateUpperWalkWay()
68 {
69     for (int x = -5; x < 1; x++)
70     {
71         GameObject.Instantiate(CubePrefab, new Vector3(x, 5, 5), Quaternion.identity);
72     }
73 }
74 }
```

Now, we should go over each method one at a time. To start this off, we have our CreateWalkWay method.

```

1  public void CreateWalkWay()
2  {
3      for (int x = 5; x < 10; x++)
4      {
5          GameObject.Instantiate(CubePrefab, new Vector3(x, 0, 5), Quaternion.identity);
6      }
7 }
```

We are iterating over the x variable starting with x being at 5 and ending at 10. We then create a cube at the Vector 3 positions of x (5 – 10), y being always at 0, and z always being at 5. Pretty straight forward and simple.

Let's take a deeper look at what is going on with the CreateStairs method.

```

1  public void CreateStairs()
2  {
3      int y = 0;
4      int z = 5;
5      for (int x = 5; x > 0; x--)
6      {
7          if (x == 5)
8          {
9              y = 0;
10             GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
11         }
12     }
```

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

```
13     if (x == 4)
14     {
15         y = 1;
16         GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
17     }
18
19     if (x == 3)
20     {
21         y = 2;
22         GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
23     }
24
25     if (x == 2)
26     {
27         y = 3;
28         GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
29     }
30
31     if (x == 1)
32     {
33         y = 4;
34         GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
35     }
36
37     if (x == 0)
38     {
39         y = 5;
40         GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
41     }
42 }
43 }
```

This method is vastly different from how we handled things before. So, let's break this down into smaller and more manageable chunks.

```
1     int y = 0;
2     int z = 5;
3     for (int x = 5; x > 0; x--)
4     {
5 }
```

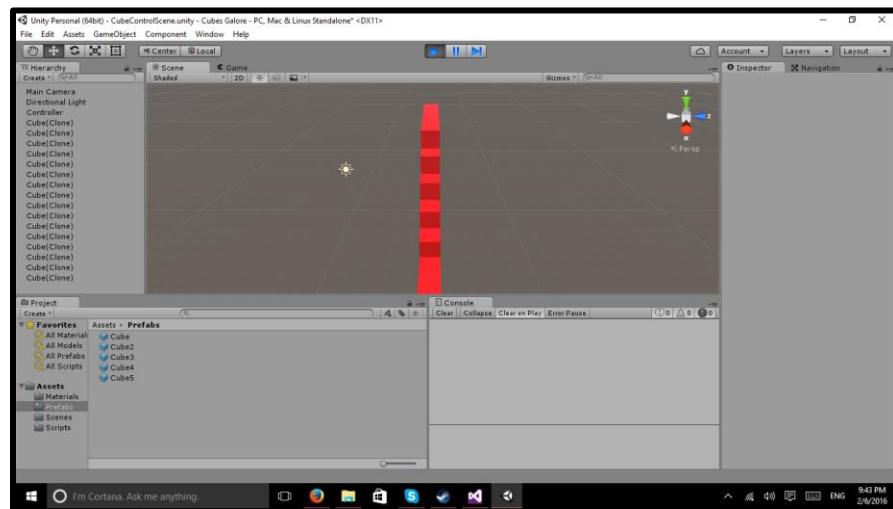
We started off by initializing our y and z variables at 0 and 5 respectively. And we are looping with a for loop that starts at 5 and ends at 0. Instead of increment by 1, we decremented by 1 (Fancy words for counting up and down respectively).

```

1      if (x == 5)
2      {
3          y = 0;
4          GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
5      }
6
7      if (x == 4)
8      {
9          y = 1;
10         GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
11     }
12
13     if (x == 3)
14     {
15         y = 2;
16         GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
17     }
18
19     if (x == 2)
20     {
21         y = 3;
22         GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
23     }
24
25     if (x == 1)
26     {
27         y = 4;
28         GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
29     }
30
31     if (x == 0)
32     {
33         y = 5;
34         GameObject.Instantiate(CubePrefab, new Vector3(x, y, z), Quaternion.identity);
35     }

```

We have a lot of if statements here, which is just pointing to the location of the x variable. Within each if statement we are specifying the y position. Lastly, we set those locations in the Vector 3 portion of the Instantiate clause.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

As you can see, converting algorithms from 2D to 3D is extremely simple and rather fun to do. I hope you learned a lot from this tutorial.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

How to make a game in Unity3D – Part 1

By Jesse Glover

Today, we will finally start developing a game using everything we have learned in Unity3D. I have thought long and hard about how we should go about doing this and have decided to do the proper route. The proper route is we design, prototype, and implement the game. The tutorial today will be all about the design phase. The design phase is writing out what we want in the game with a few images pertaining to a rough draft to how the game will look.

PART 1: INITIAL DESIGN PHASE

This section is the boring wall of text, sadly, there is no way around it although I will try to make it as short as possible while still being informative.

SECTION 1: BASE GAME IDEA

To begin designing the game, we first need a basic idea that we can flesh out later. This section will be extremely short and act as a means to quickly jump into the next phase.

Our base game idea will be a 3D 3rd person view dodge game.

SECTION 2: EXPANDING UPON THE IDEA

Now that we have our base game idea, we need to begin to flesh it out some. The idea is to keep it as non-technically as possible in this phase.

3D 3rd person view dodge game that counts how long someone survives. It will have a leaderboard, a start screen, game play screen, credits screen, and game over screen.

The leaderboard will display the player's name/ nickname and their survival time.

The Start Screen will allow the player to look at the leaderboard, play the game, exit the game, and look at the credits screen.

SECTION 3: FLESHING OUT MORE DETAILS

Now it is time to go into a brief idea of the game mechanics.

As the player plays the game, they must move around on the screen dodging obstacles that will drop down from the top, fly up from the bottom, or fly towards them from the right side of the screen. Player can only dodge the obstacles, no shooting.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

SECTION 4: STARTING THE GAME DESIGN DOCUMENT

We now have the base idea for the game that has been fleshed out some. We can start writing our design document and specifying what features we want to have and would like to have. The main reason for want to have and would like to have is to determine what is critical to releasing the game and what features would expand upon the game that aren't needed to release the game.

Your game design document can be as verbose as you want it with as deep of detail as you want to go. The idea for the design document is to allow for not only yourself to be able to look at it and start prototyping or developing the game, but for any developer that looks at it as well. The game design document will follow (loosely depending on the type of game) this structure.

- Story
- Characters
- Level / environment design
- Gameplay
- Art
- Sound and Music
- User Interface, Game Controls

SECTION 5: WRITING THE DOCUMENT

The Big Story:

Ace pilot Jeffrey B is in deep trouble. He is out of ammo and rockets, deep in enemy skies, injured, and surrounded by pilots firing rockets at him. He called out to the radio tower for help, the radio man has control of the plane and has accessed the camera feed to assist Jeffrey B.

The Characters:

- Jeffrey B. starring as himself
- The player starring as the Radio Man

The Level/ Environment design:

A cloudy sky that has procedurally changing colors with each play through ranging from a sunrise orange, brilliantly sky blue, to a sunset red.

The Gameplay:

Character jet dodges procedurally generated rockets from various angles.

The Art Style:

3D Voxel artwork for simplicity and single person development.

The Sound and Music:

Music composed using an electronic keyboard and guitar. Menus will have a soft rock feel and the gameplay will have more hard rock to metal feel.

The User Interface:

A simple button interface for menus that designate whether it leads to another menu or to gameplay. Either by text or icons.

The Game Controls:

Controls for the menu will be handled via mouse input and game controls will allow for W,A,S,D/ Up arrow, Down arrow, Left arrow, Right arrow, or mouse movement.

PART 2: PROTOTYPING

This section will detail how I go about prototyping, which involves pseudocode, crude pictures for the art design, and crudely written music.

What is Pseudocode?

The beginning programmer may not be familiar with the term, so let me start off by explaining what it is. Pseudocode is where you structure your writing to mimic your programming language.

Why go with crude pictures for the art design?

When doing your prototype, it doesn't need to look like the finished product. Instead, you want to make it as quickly as possible and give an idea of how you want everything to look.

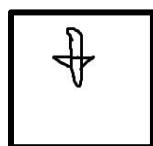
Why crude music?

This one is a personal preference for me. The crude music is the basic chord progressions you will be using with no leads or any real melody. It serves the purpose to see what bpm and feel you want to go for in the finished product.

SECTION 1: CRUDE ART

We know that we will be doing 3D Voxel art in the end, however, we can start off with 2D for the crude art. There are a couple of things we need to make. The Jet and the Missiles. Pretty basic even for the final product.

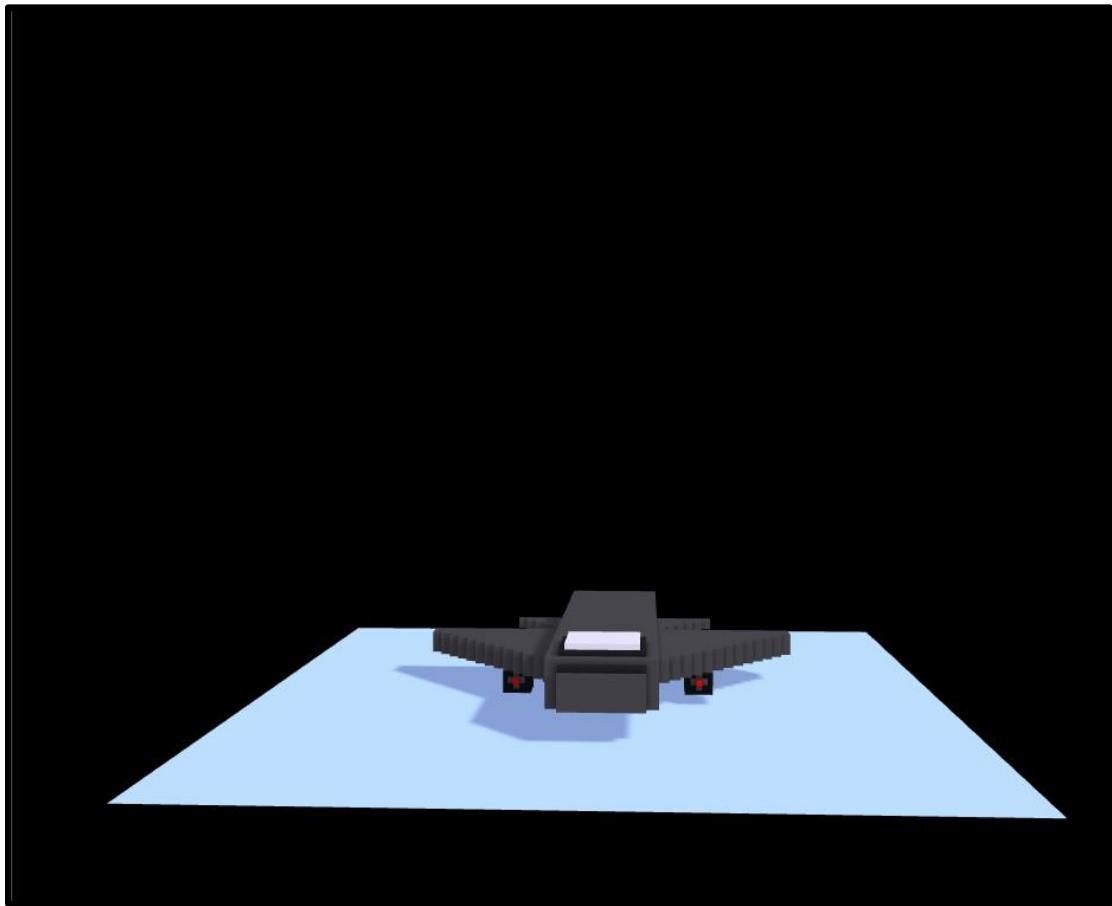
Our first item to make is the crude Jet. Go as simplistic as you can that conveys the basic idea.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

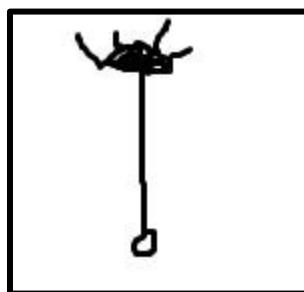
© Zenva Pty Ltd 2021. All rights reserved

Pretty simple right? Now let's look at what the final design looks like.

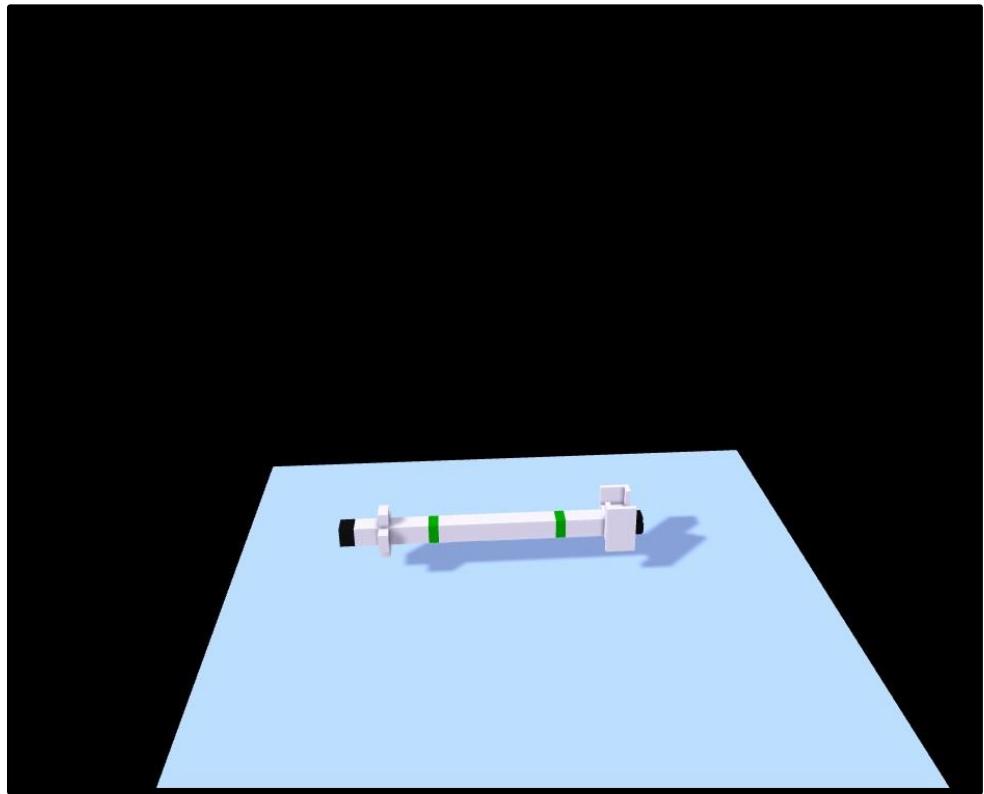


As you can see, the basic idea remains the same. The only difference is we started with 2D and ended with 3D Voxel art.

Let's look at the missile now.

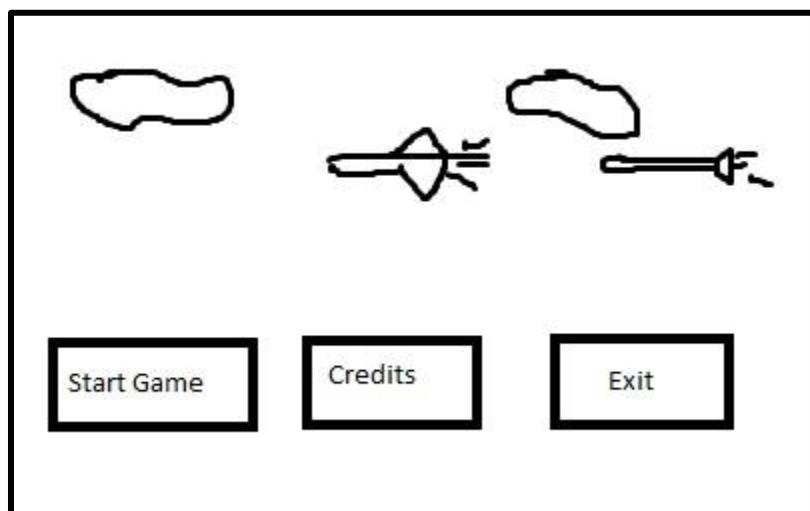


Again, extremely simple. Almost looks like an arrow. Again, we didn't want to go too in depth with details. The artist will deal with that with the final product. Let's look at the 3D version.

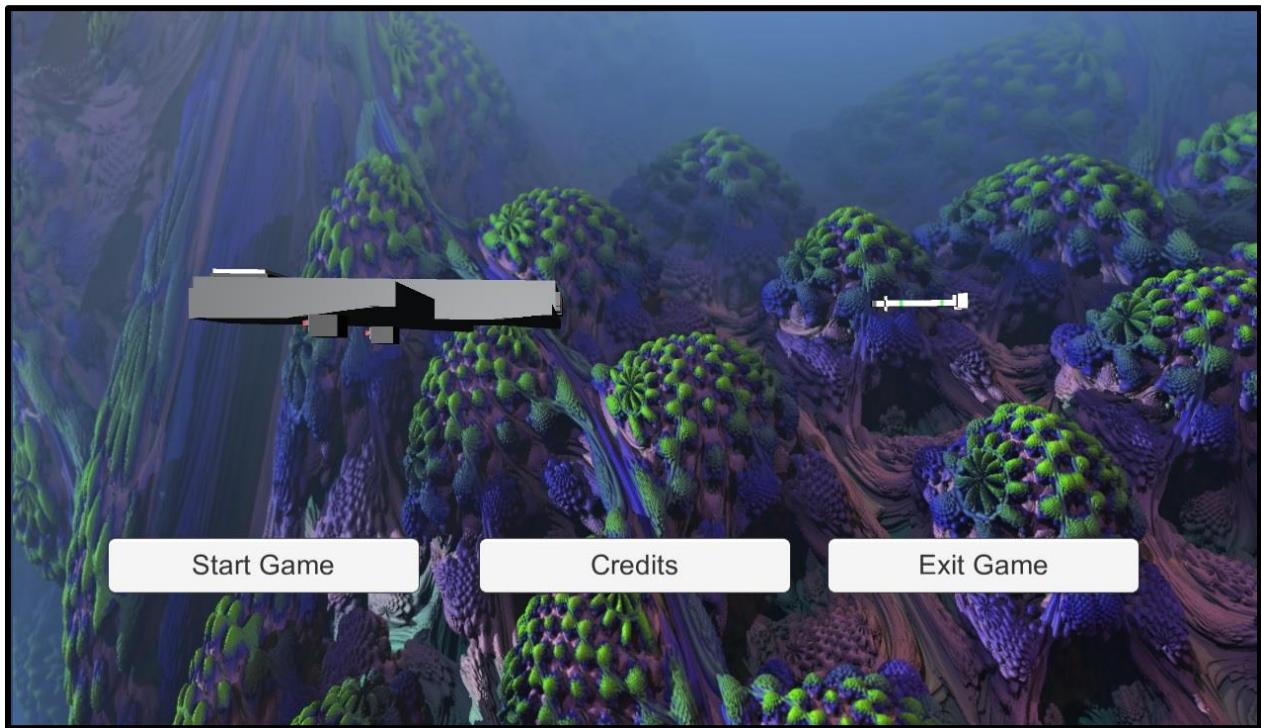


That looks a lot better and follows a design for an actual missile a bit better. On to the basic UI.

Here is how we envision our UI will look for starting the game.



What we are looking at is the jet flying with a missile chasing it. Below, we have 3 buttons (Credits, Start Game, and Exit Game). This shows we want to have a minimalist UI that is easy to navigate. Let's look at the final version from a Unity3D screenshot.



Very cool. It is starting to look a lot more like a game. But we haven't addressed what we want for the background. I have decided that fractals based off the Mandelbrot and Julia sets would fit very well with the whole voxel theme.

Let's take a step back for a moment here and answer a question you may have from my last statement. Fractals based off Mandelbrot and Julia sets? Fractals are images that have forever repeating patterns within it and is a very important and beautiful mathematical find. We are not going to be creating the fractals ourselves because that could be several tutorials of their own, instead, we are going to use fractals that have been premade.

SECTION 2: CRUDE MUSIC

This section is for those that have either used music creation software before or have musical knowledge. You can choose to skip it if you don't have this knowledge or you can read it and hopefully start to learn a little bit about it.

As stated in the design document, we want the UI to have a more soft rock feel using keyboards and guitar. Now, we need to come up with the basic progression that would have that soft rock feel.

I am going to go with the key of C Major for the UI. I'm going with major because it sounds a little bit happier and it is also a lot easier to write. Below, I have the base scale and pentatonic scale written out in note letters. I have the diatonic Triads and chords that we can use written out as well as the Relative key.

The usage of the capital M denotes that it is Major and the lower case m denotes that it is a Minor.

Scale:	C, D, E, F, G, A, B
Pentatonic Scale:	C, D, E, G, A
Diatonic Triads:	1. CM → C, E, G 2. Dm → D, F, A 3. Em → E, G, B 4. FM → F, A, C 5. GM → G, B, D 6. Am → A, C, E 7. Bdim → B, D, F
Relative Key:	A Minor

Now that we have that information, we need to choose the chords we want to use.

CM, Am, GM, and Em are the basic chords we will use (C Major, A minor, G Major, and E minor). I chose these chords for 2 reasons. The first reason being that they are easy to play and very well used. The second reason is because it is very easy to incorporate using the relative key when writing the leads for the song.

Unfortunately, I cannot link the crude version of the song. What I will do is add it to the base project so that you can listen to it on your own time.

PART 3: PSEUDOCODE TIME

We are going to go through this section and specifically do the UI as pseudocode. The reason for this is because it seems we have been doing a focus on the basic UI. Which is perfectly fine because the next tutorial will be making the full game.

SECTION 1: PLAN OUT WHAT WE WANT TO DO

We know we will need a Canvas Component to use certain components. We will also need 3 buttons.

Since this is editor, I will go with the GO structure. Which is an XML visualization of what we need.

```
1 <GOStructure>
2 <Canvas>
3 <Button1>
4 <Text>
5 </Button1>
6 <Button2>
7 <Text>
8 </Button2>
9 <Button3>
10 <Text>
11 </Button3>
12 </Canvas>
13 </GOStructure>
```

Pretty simple to follow right? GOStructure (Game Object Structure) is the root element of the entire XML file. Canvas is the Main Body of the XML. Button1, 2, and 3 all have a child element of Text.

This completes section 1 of the psudocode for our Editor with the Game Object Structure.

SECTION 2: PLANNING OUT BUTTON EVENTS

Now that we have the basic structure, it is time to figure out what each button does and how it does it. Now this is pretty simple in itself as well.

Button 1 is Start Game. Button 2 is the Credits scene. Button 3 is Exit Game.

```
1 Public ButtonController()
2 {
3     Private void StartGame()
4     {
5         Start the game(); // This will utilize the Scene Manager class
6     }
7
8     Private void CreditsScene()
9     {
10        Go to the credits(); // Scene Manager class utilization once again
11    }
12
13    Private void ExitGame()
14    {
15        Exit the game(); // Application controls from System class
16    }
17 }
```

The job of the pseudocode is to plan out the code. Not to write the actual code, although you could write code that “should” work instead. It is easier to go with pseudocode because you aren’t trying to copy and paste what you have just written.

CONCLUSION:

As you can see, the design process can take a vast amount of time to actually do. While it isn’t a bad thing, it could be resource consuming. If you are a hobbyist or a beginner, you could write the design document while you have down time at work, in class, or when you are bored. I would always suggest writing a design document whenever you are making any application or game.

Even if you are just recreating someone else’s game as practice, or making something on your own; the design document will help guide you along with the development cycle. It will also be something that you will need to know when getting a job as a developer. If you can read, write, and understand design documents and UML; it will most certainly be a plus.

I hope you enjoyed this tutorial, next week, we will use everything written in this tutorial as well as prior things I have taught to make this game. May your code be robust and error free, this is Jesse signing off.

How to make a game in Unity3D – Part 2

By Jesse Glover

Fractal artwork used under the GPL 3.0 License from <http://opengameart.org/content/fractal-alien-landscape-pack>

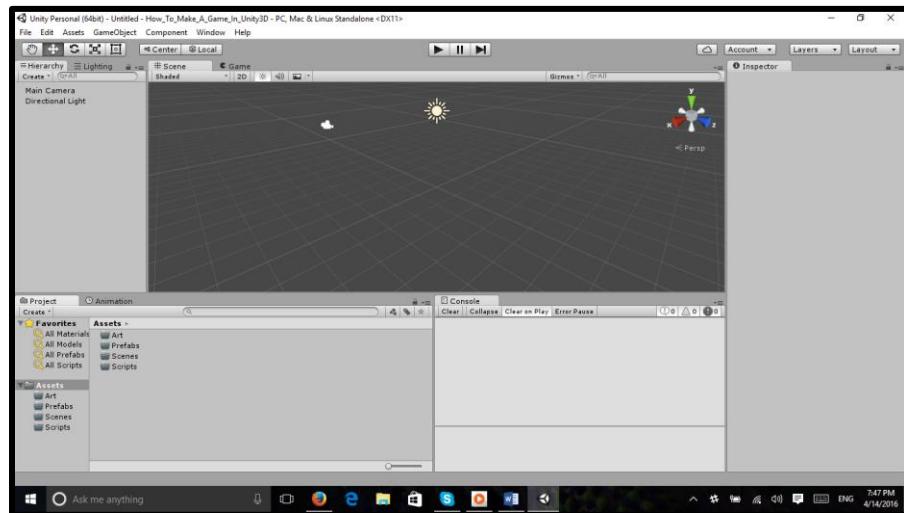
Download link for complete project [here](#).

Today, we will be taking the design concept we made in part 1 and actually be developing the game. Whilst this game will not be of AAA quality, it will show the fundamentals of game design and implementation. We will be using concepts learned from all of my previous tutorials in some way, shape, or form to be creating this game. For example; we could use JSON or XML file format for saving data, algorithms for the missiles flying at the player from random directions, and so much more.

Today, we will focus on the primary game play for this game. That means, we will be working in a single scene. We will also only implement the scripts needed to get the base game to run. I hope you are as excited as I am about this tutorial. It has been quite a few months and tutorials to lead up to this one. So, let's jump right on in and start developing this long awaited game.

PART 1: SETTING UP THE GAME SKELETON

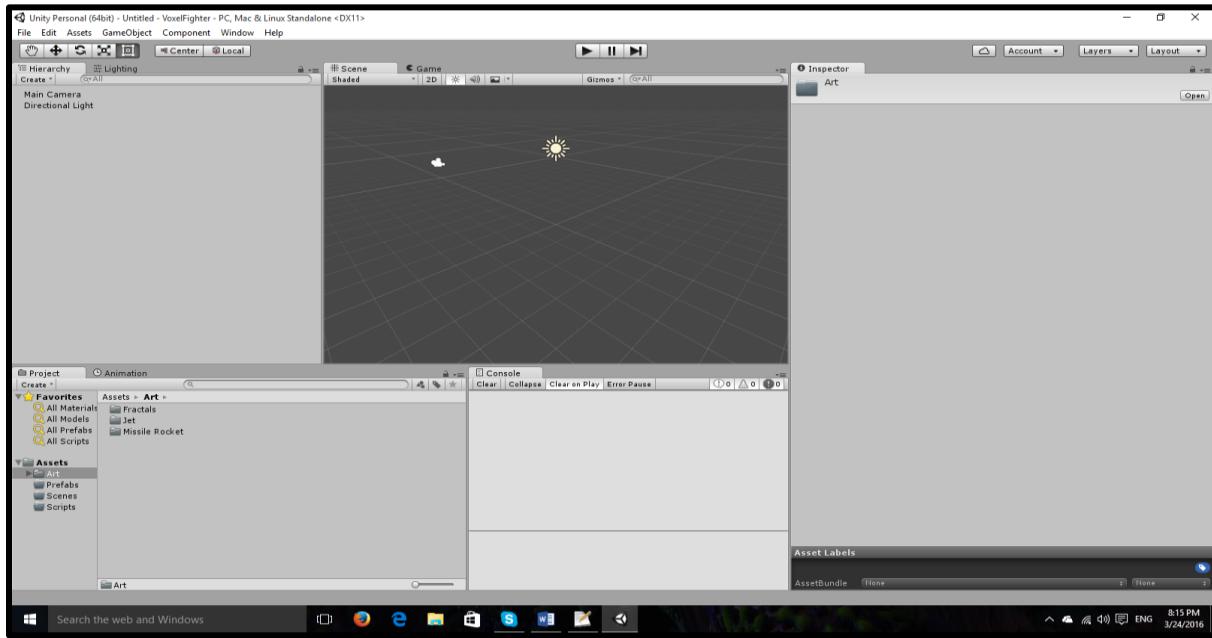
This one should be fairly obvious for all of us now. We need to first set up the skeleton of the whole project within the Unity3D editor. This can be as complete as you want it to be, but I like to start with the bare essentials. The bare essentials are the scripts, art, prefabs, and scene folders.



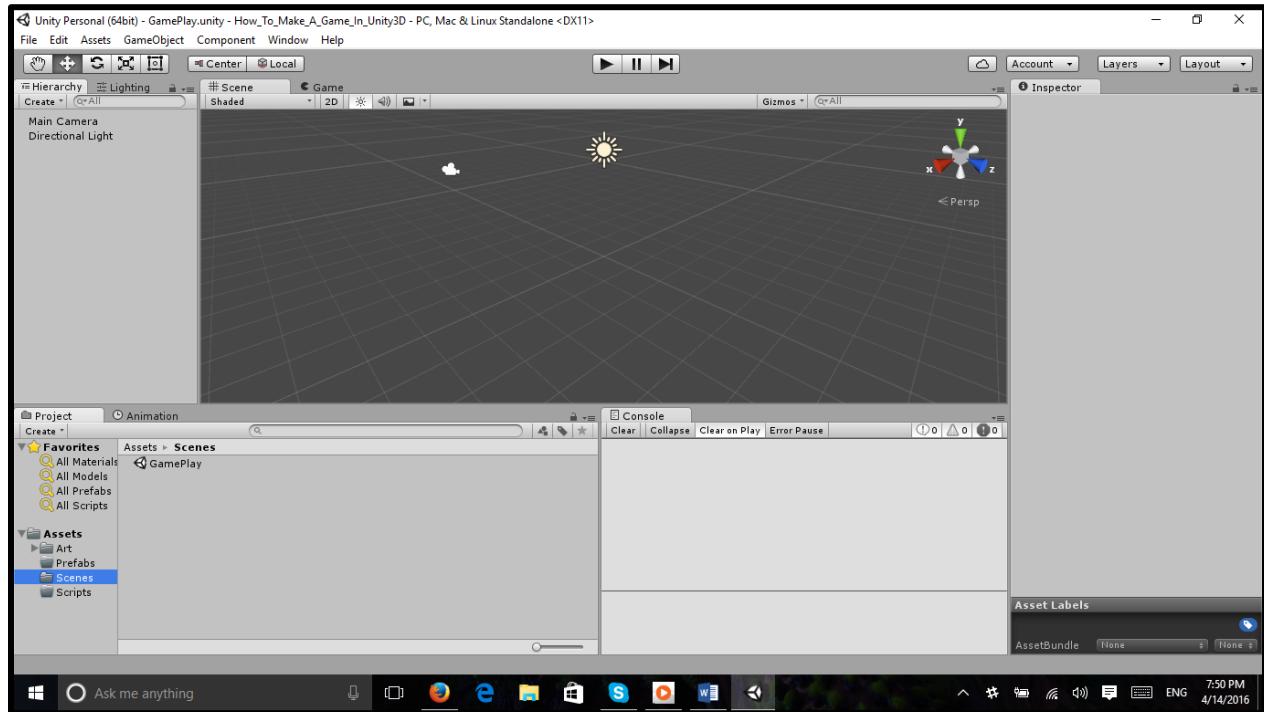
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Next up, we take all of the art that we have gathered / created and place them into the art folder. This includes the voxel jet, voxel missile, and the fractal art from open game art.



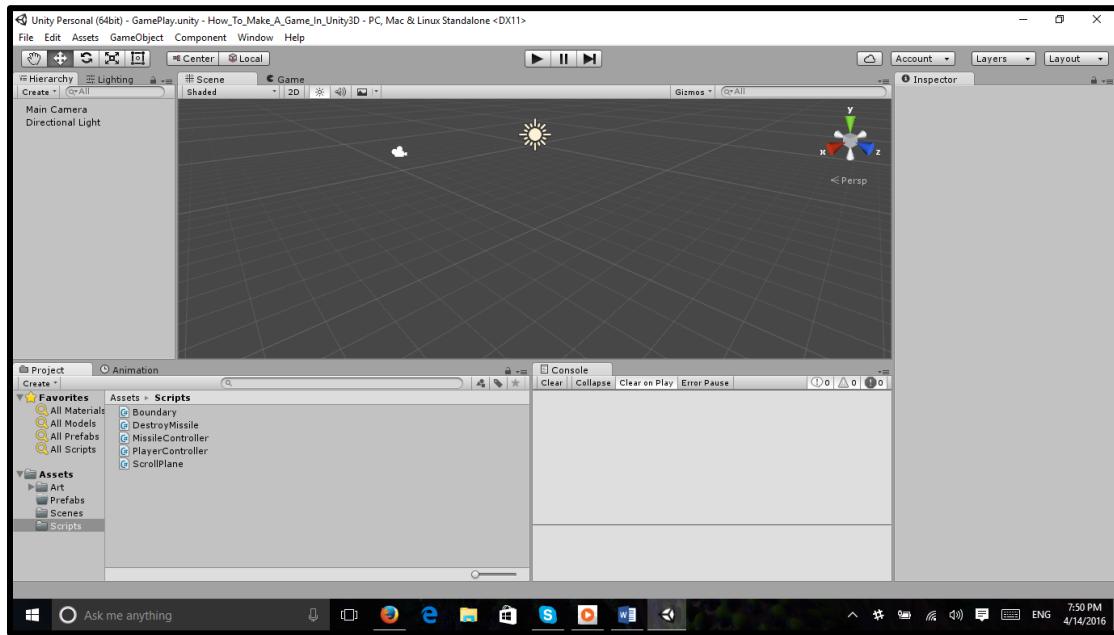
We now save our base scene and call it “Game Play Screen”.



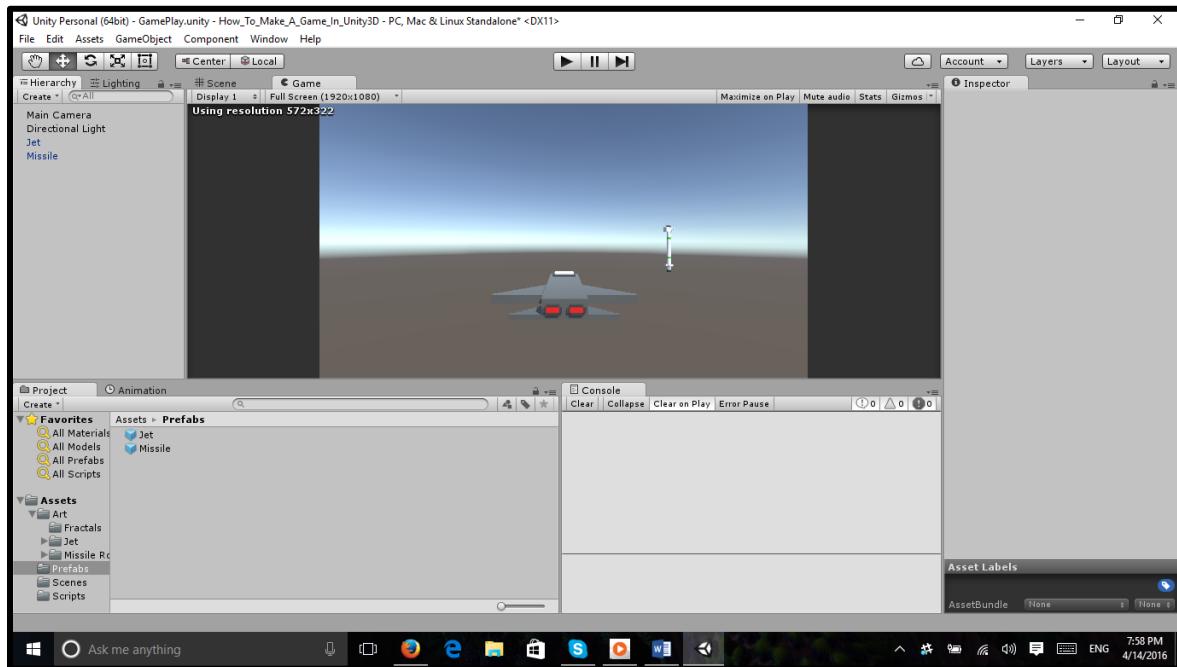
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Okay, now we can set up our scripts skeleton that we may or may not fully utilize when creating the game. The scripts we need are boundary, player controller, missile controller, destroy missile, and scroll plane.



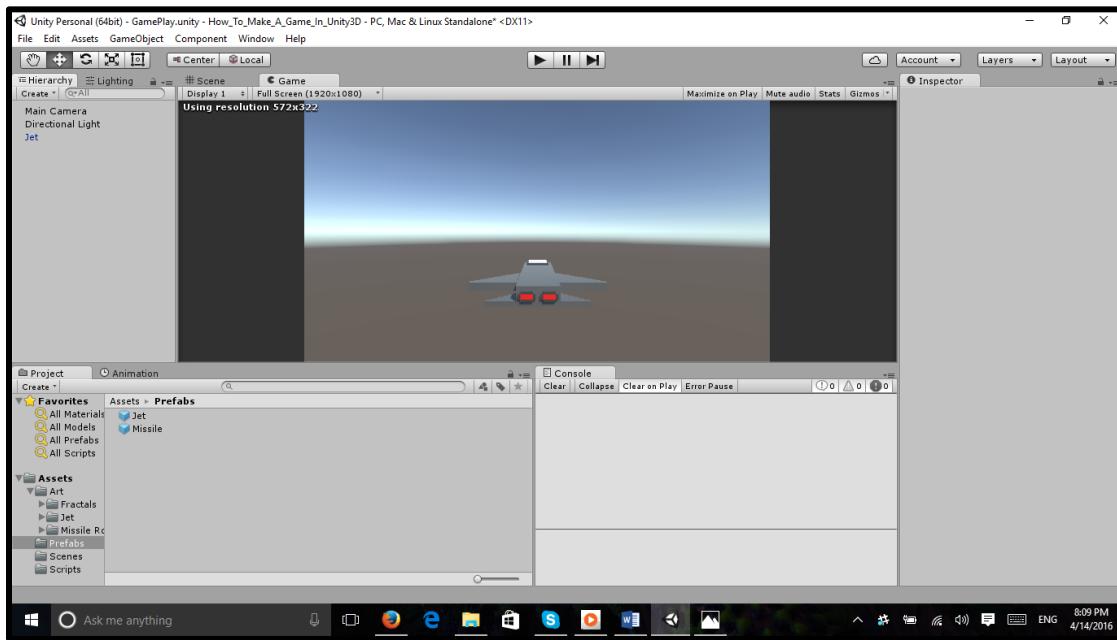
We should now set up our prefabs, our prefabs will consist of the player ship and the missile at the present time. This is how they should be scaled and look.



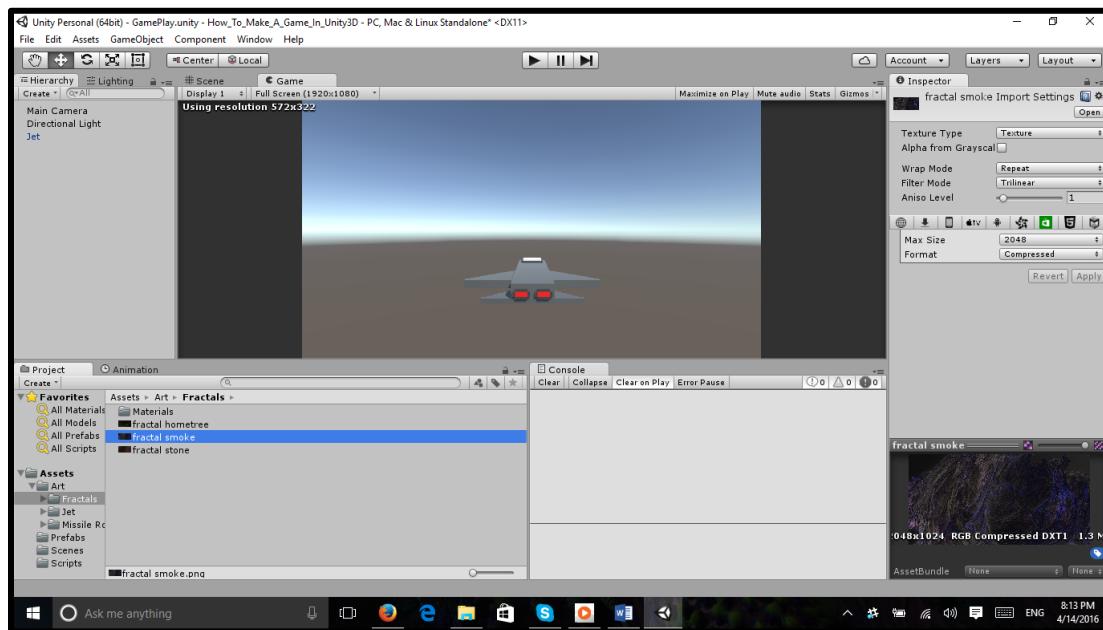
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Now we delete the missile prefab from the Hierarchy pane. It should still be in the prefabs folder, make sure not to delete it.



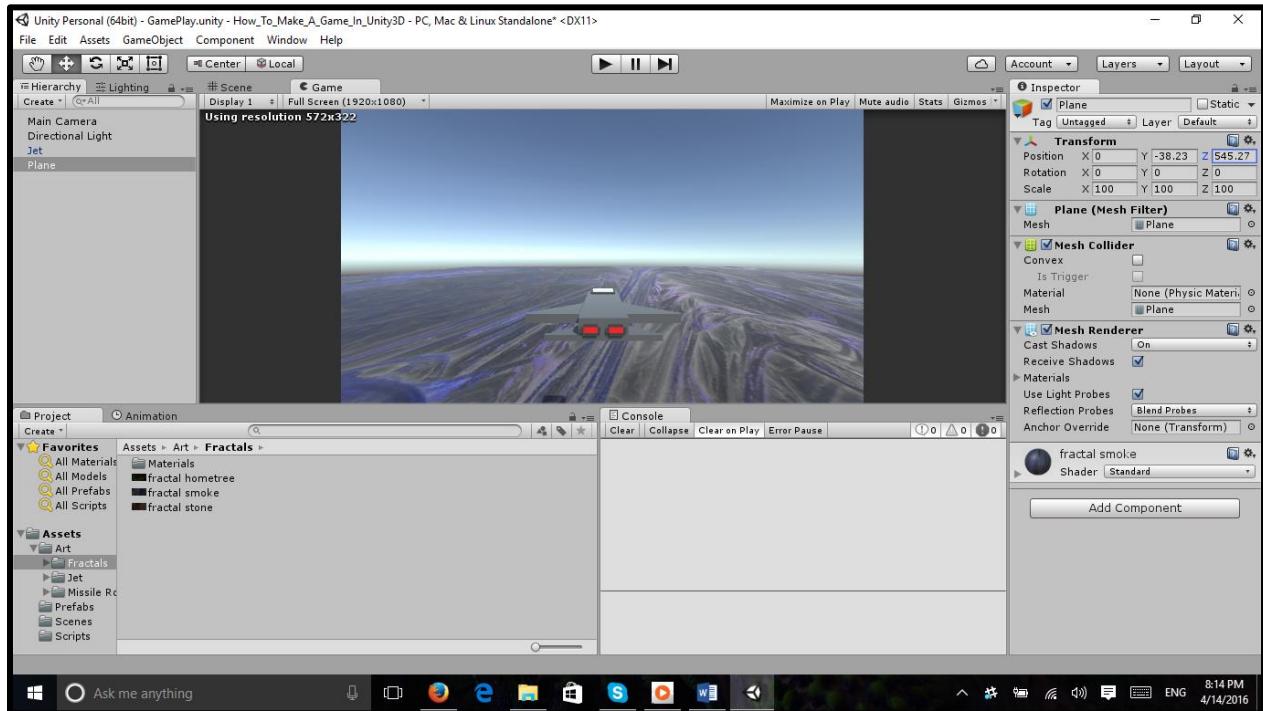
In the art folder, fractal sub directory; Make sure the fractal texture is set to texture, wrap mode of repeat, and a filter mode of trilinear. The name of the texture we will be using is fractal smoke.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Create a plane that is 100 x 100 x 100, y position set at -38.23, z position at 545.27. Attach the fractal image called fractal smoke to the plane.



Our base game skeleton is now complete. We can now begin working on the game itself. This is where the fun begins!

PART 2: CHOOSING WHERE TO BEGIN

Before we can really move forward, we need to decide if the back end or front end is of the most importance. A visually based person may decide to go with the front end first and then work on the back end. A person with a mind and knack for the abstract will go with the back end first. We will go with back end first because it is the industry standard. It is the industry standard because art, music, and mechanics may change throughout the development process; Back end can easily be scaled and modified to fit the new ideas whilst front end will struggle with it.

With this game, the back end will be setting up the game boundaries, the missile controller, the player controller, background controller, and the destruction script. So, before we jump into the code, we should specify this information.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

SECTION 1: GAME BOUNDARIES

Since we know this is an endless flight game, we don't want the player to move too far to the left, right, up, or down but we want them to have enough free movement that they can dodge the obstacles. So we will give these a rough value of 10, we can modify these values as needed when we test the game play later on.

- Left X has a max value of -10.
- Right X has a max value of 10.
- Upper Y has a max value of 10.
- Lower Y has a max value of -10.

Since we know this will be in a class of its own, we also need to make sure the class is set to have the ability to be serialized.

```
1 using UnityEngine;
2 using System.Collections;
3
4 [System.Serializable]
5 public class Boundary
6 {
7     public float xMin, xMax, yMin, yMax;
8 }
```

System.Serializable being attached to this class will allow us to modify the data in the class that calls this one in the Unity editor. More on this later.

SECTION 2: PLAYER CONTROLLER

The player controller only needs to store the methods for moving the player around on the screen.

```

1  using UnityEngine;
2  using System.Collections;
3
4  public class PlayerController : MonoBehaviour
5  {
6      public Boundary boundary;
7      public Rigidbody stiffBody;
8      public float tilt;
9      public float speed;
10
11     void Update()
12     {
13         var horizontal = Input.GetAxis("Horizontal");
14         var vertical = Input.GetAxis("Vertical");
15
16         Vector3 movement = new Vector3(horizontal, vertical, 0f);
17         stiffBody.velocity = movement * speed;
18
19         stiffBody.position = new Vector3
20         (
21             Mathf.Clamp(stiffBody.position.x, boundary.xMin, boundary.xMax),
22             Mathf.Clamp(stiffBody.position.y, boundary.yMin, boundary.yMax)
23         );
24
25         stiffBody.rotation = Quaternion.Euler(0.0f, 0.0f, stiffBody.velocity.x * -tilt);
26     }
27 }
```

As you can see, we called the Boundary class within the player controller, although we have not specified the data directly in the code. You may not be familiar with the implementation of the Vector3 we have in this code.

This allows us to more explicitly state what we want the vector 3 values to be, since I only have 2 items in it, it defaults to Vector3(float x, float y). Mathf.Clamp allows you to clamp a value between a minimum float and maximum float value. Quaternion.Euler is used to return a rotation that rotates z degrees around the z axis, x degrees around the x axis, and y degrees around the y axis (in that order).

SECTION 3: MISSILE CONTROLLER

The missile controller will house the methods and algorithms for when and how the missiles will spawn.

```

1 using UnityEngine;
2 using System.Collections;
3
4 public class MissileController : MonoBehaviour
5 {
6     public GameObject hazard;
7
8     private float spawnWait = 1f;
9
10    void Start()
11    {
12        StartCoroutine(SpawnWaves());
13    }
14
15    IEnumerator SpawnWaves()
16    {
17        while (true)
18        {
19            var xMinMax = Random.Range(-15f, 20f);
20            Vector3 spawnPosition = new Vector3(xMinMax, 10f, 25f);
21            Quaternion spawnRotation = Quaternion.Euler(new Vector3(90f, 0f, 0f));
22            Instantiate(hazard, spawnPosition, spawnRotation);
23            yield return new WaitForSeconds(spawnWait);
24            spawnWait -= 0.01f;
25            if (spawnWait < 0.05f) spawnWait = 0.05f;
26        }
27    }
28}

```

IEnumerator is Unity's version of parallel programming and it is supposed to be used in conjunction with StartCoRoutine.

SECTION 4: DESTROYING OBJECTS

We now need a way to make sure the player ship and the missile that hit the ship is destroyed when the collision has been detected. We also want to make sure the spawned missiles will automatically be destroyed after a certain time has passed.

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class DestroyMissile : MonoBehaviour
5 {
6     public float lifetime;
7
8     void Start()
9     {
10         Destroy(gameObject, lifetime);
11     }
12
13     void OnTriggerEnter(Collider other)
14     {
15         Destroy(other.gameObject);
16         Destroy(gameObject);
17     }
18 }
```

We have 2 separate destroy capabilities in this code. The first one, inside of the Start method will destroy the specified game object at the appropriate time frame which we will specify as 5 seconds. OnTriggerEnter is called when the Collider other enters the trigger. Basically, when one of the specified objects collide with a different object or another object of the same type; The object is destroyed.

SECTION 5: ADDING LIFE TO THE BACKGROUND

We should make it so that our plane with the fractal image scrolls to simulate a flying ship. It adds some life to the game, although it is not 100% needed. There are two ways we could do this, one would be attaching it to player movement or by creating a script to handle it alone. I opted for the standalone script due to the fact that the player is supposed to dodge obstacles and isn't going to constantly be moving.

```

1  using UnityEngine;
2  using System.Collections;
3
4  public class ScrollPlane : MonoBehaviour
5  {
6      public MeshRenderer rend;
7
8      private float scrollSpeed = 0.5F;
9
10     void Start()
11     {
12         rend = GetComponent<MeshRenderer>();
13     }
14     void Update()
15     {
16         float offset = -Time.time * scrollSpeed;
17         rend.material.SetTextureOffset("_MainTex", new Vector2(0, offset));
18     }
19 }
```

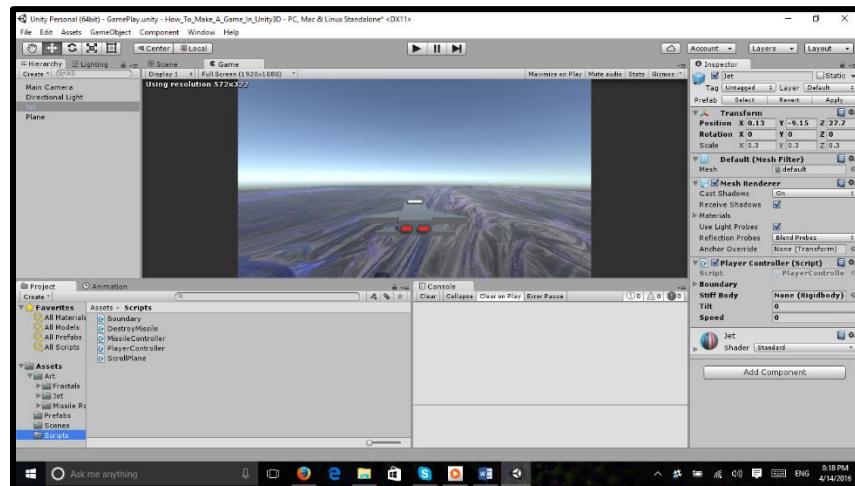
Now that we have some of our back end done, let's start building it with Unity.

PART 3: USING UNITY TO BUILD OUT THE GAME

The fun part is about to begin, we will now attach scripts to their rightful places and test to make sure everything works out according to plan. We all know that bugs can happen or items could be placed in the correct area.

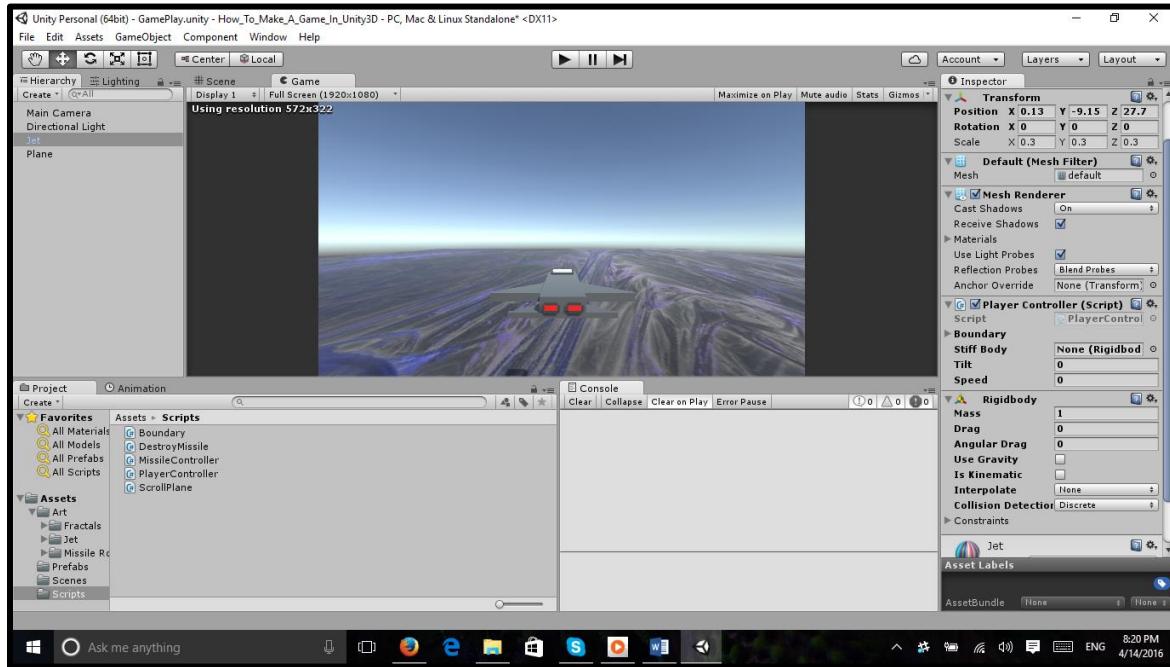
STEP 1:

Attach the Player script to the Jet. We want the player to control the Jet, we also want the Jet to move independently from the camera.

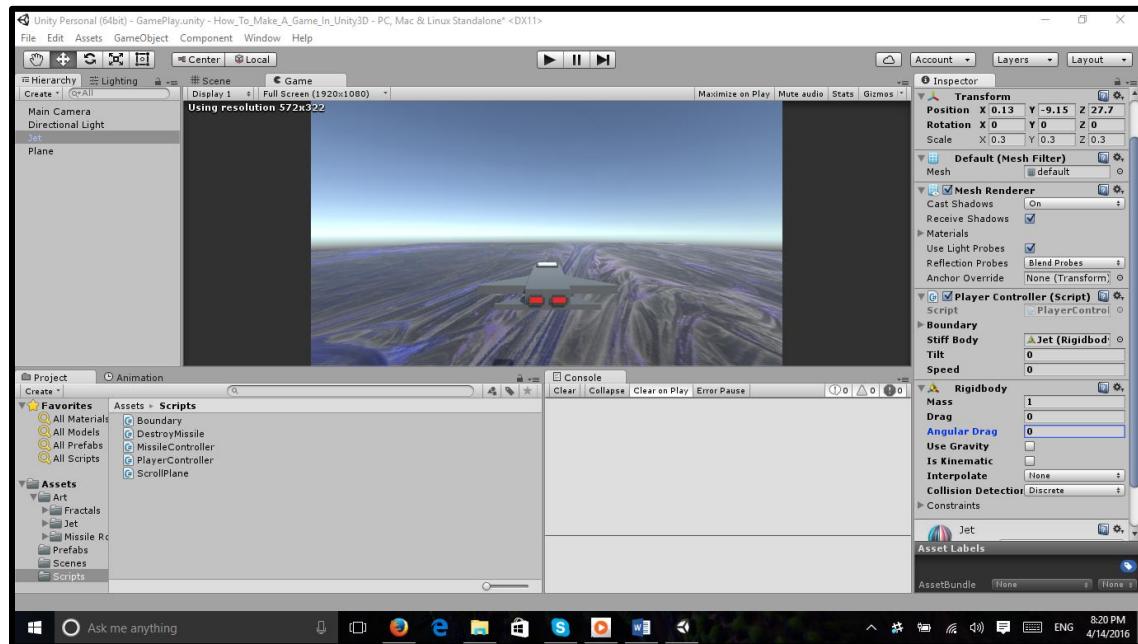


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

The script says we need a rigid body component, so we should add that to our jet via add component, physics, rigid body.



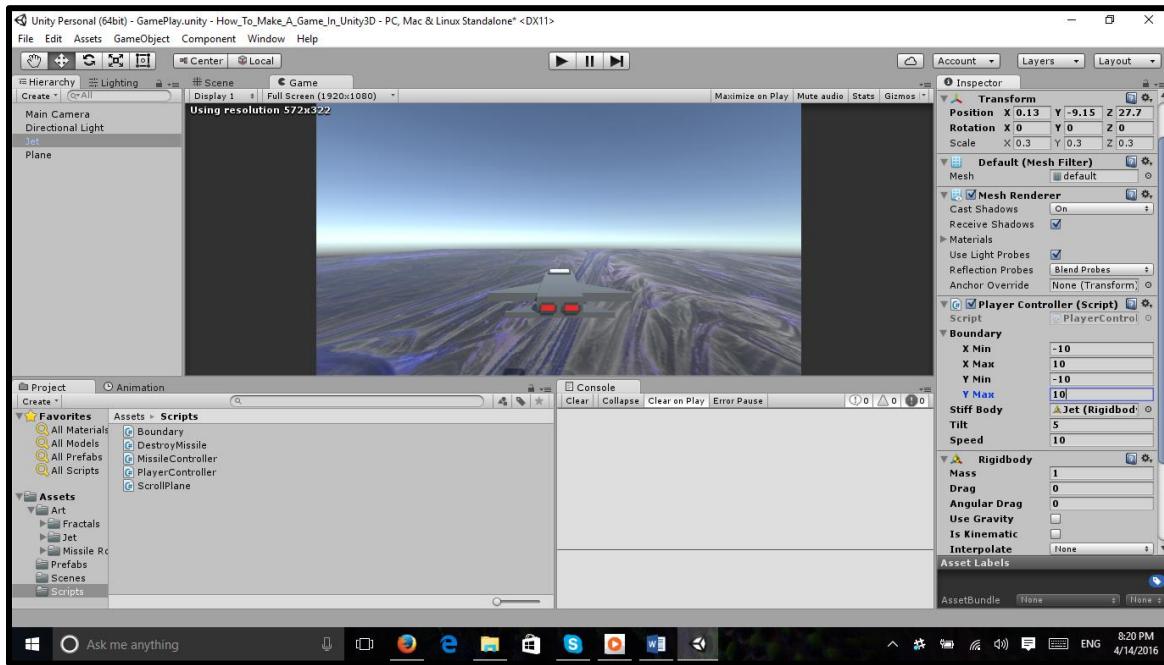
Make sure to set angular drag to 0, and uncheck the use gravity property.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

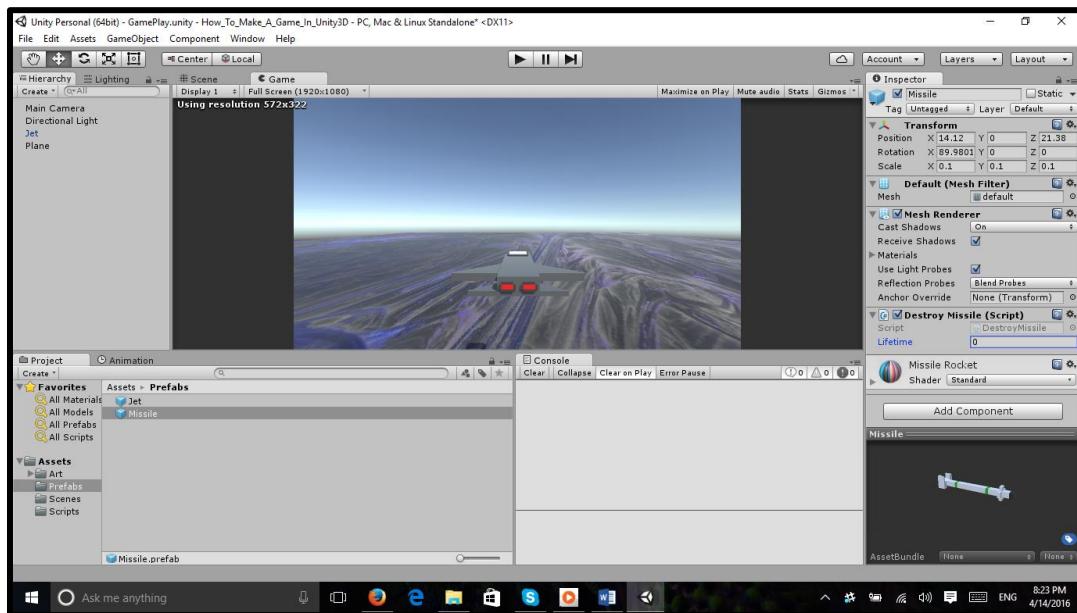
© Zenva Pty Ltd 2021. All rights reserved

Now we can drag the jet game object onto the stiff body in the player controller. We also need to set our x min and y min to be -10. X max and y max should be 10. Tilt should be set at 5, and speed should be 10.



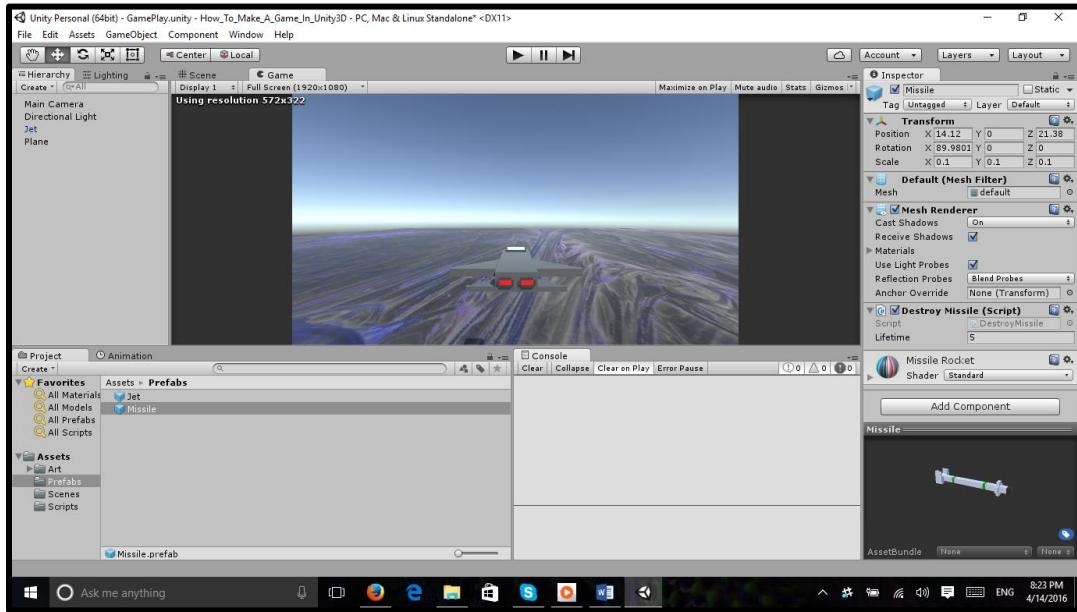
STEP 2:

Moving on to the Missile prefab. We should click on it and add component, script, destroy missile.



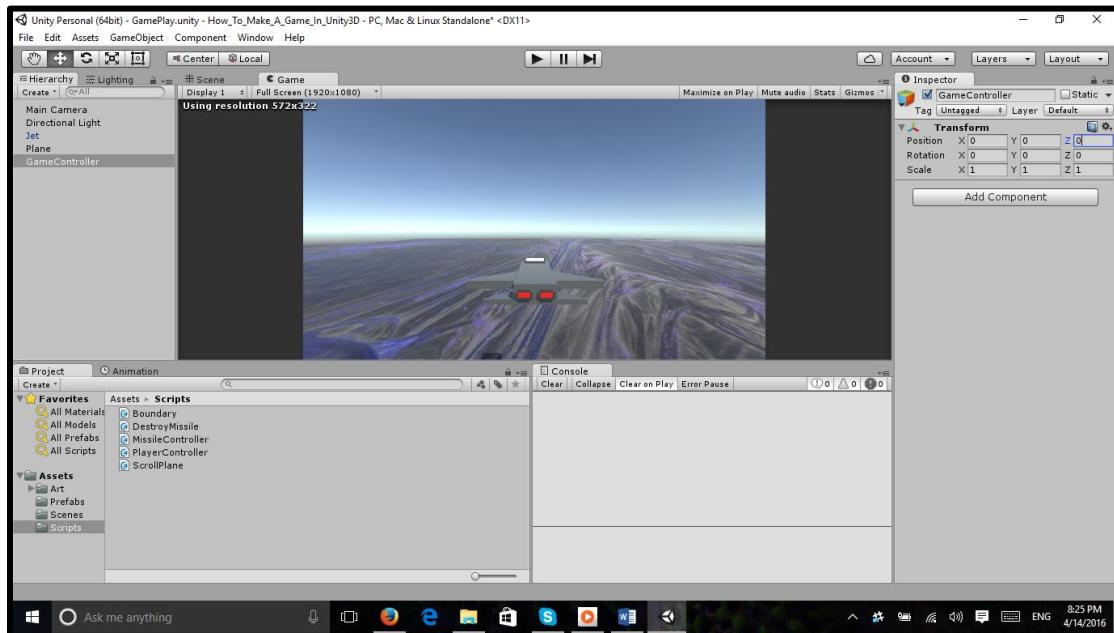
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Change the Lifetime property to be 5. We want the missile to last for 5 seconds before it destroys itself off screen.



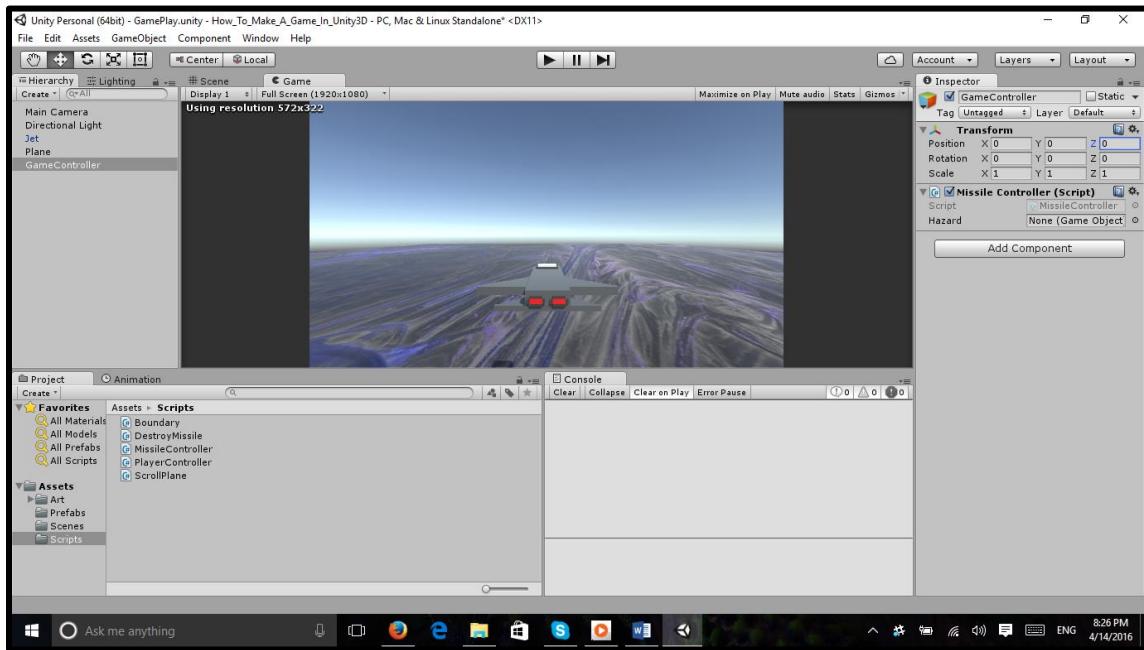
STEP 3:

Create an empty Game Object in the hierarchy pane and rename it to Game Controller. (Note: You don't have to change the x, y, z coordinates for this; I chose to because I am neurotic.)

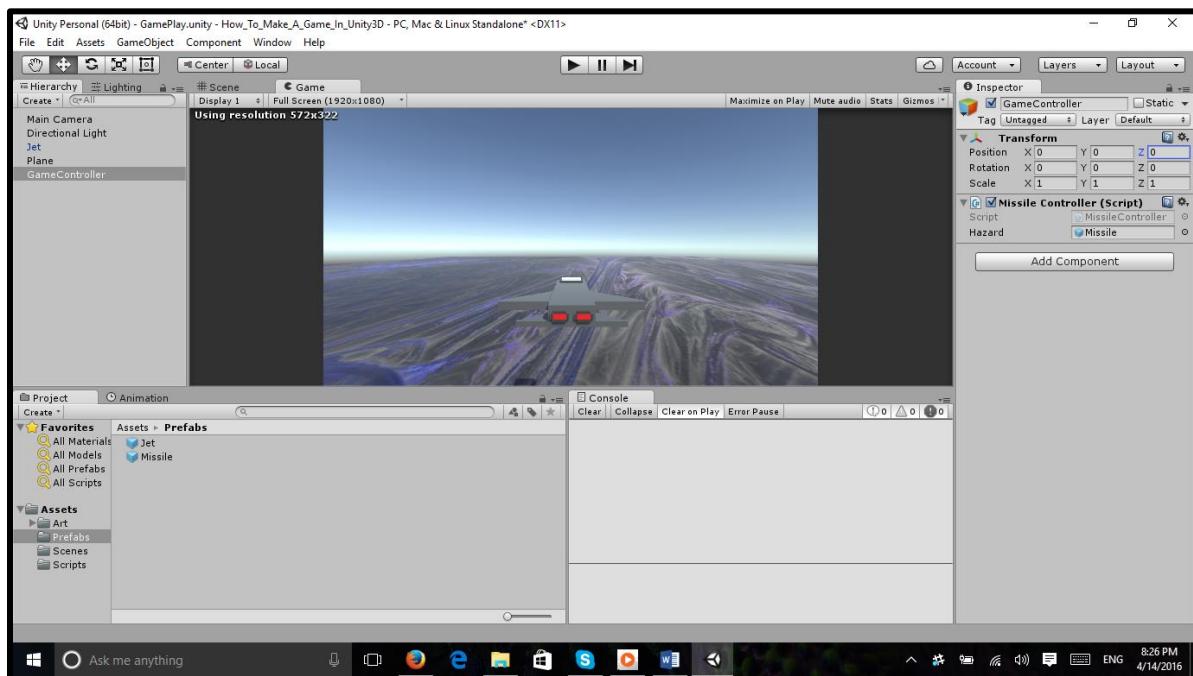


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Add component, script, missile controller or drag and drop the missile controller script onto Game Controller.



Attach the Missile prefab to the Hazard property in the Inspector Pane.

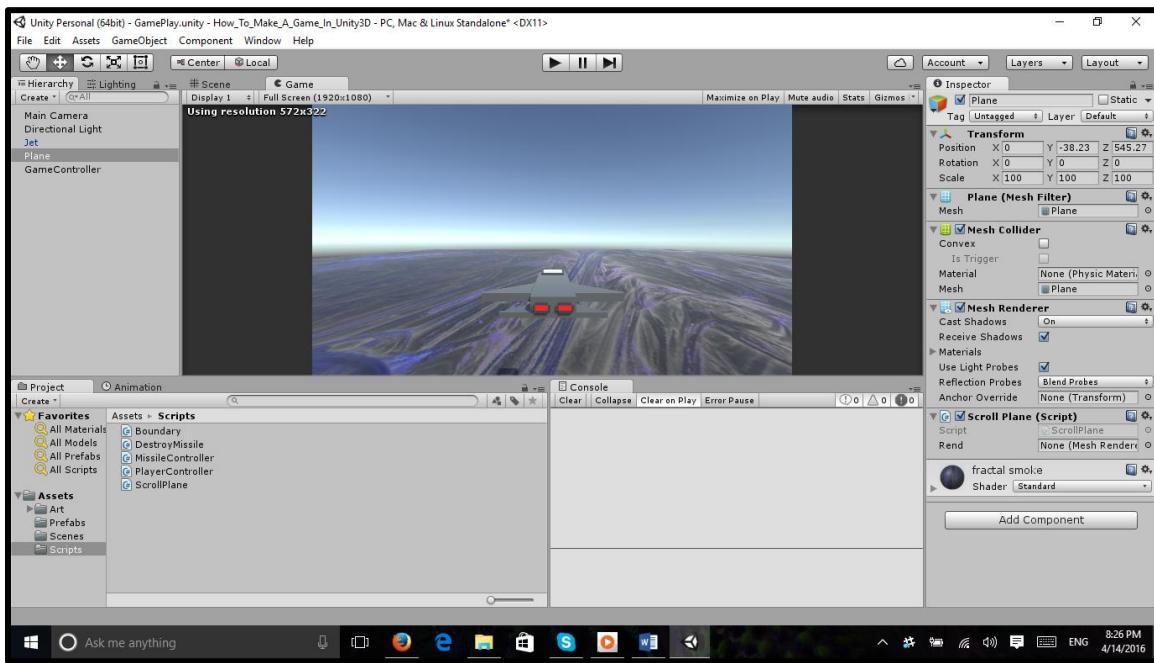


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

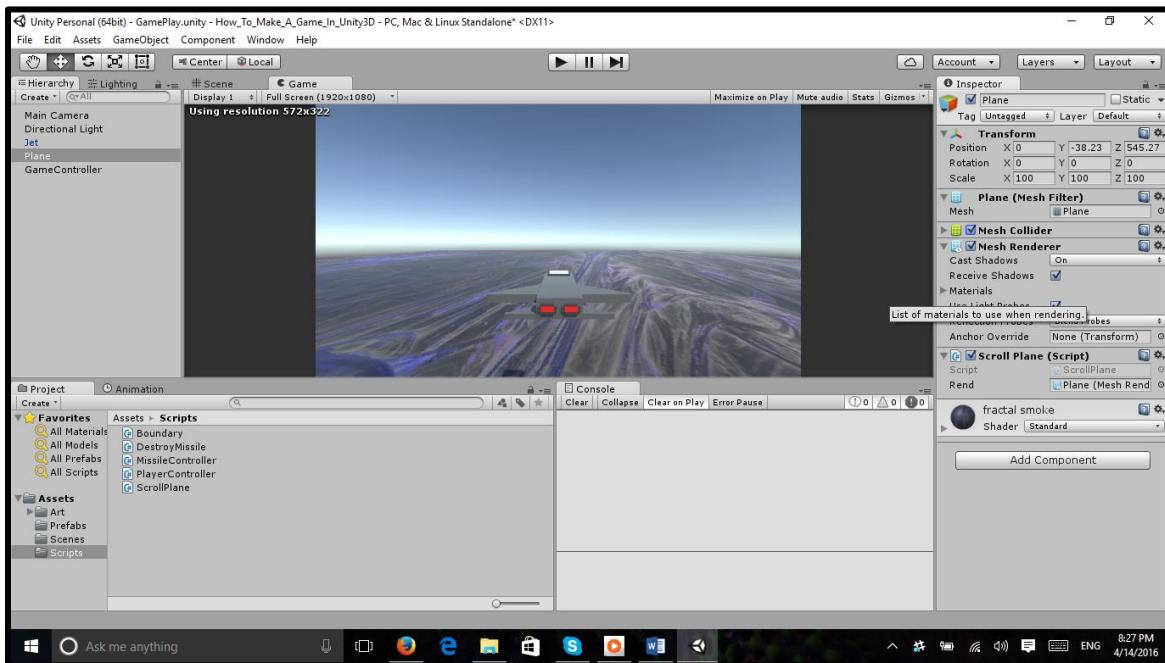
© Zenva Pty Ltd 2021. All rights reserved

STEP 4:

Select the Plane object and add the scroll plane script to it.



Drag the plane from the hierarchy pane to the rend property in the Inspector pane.

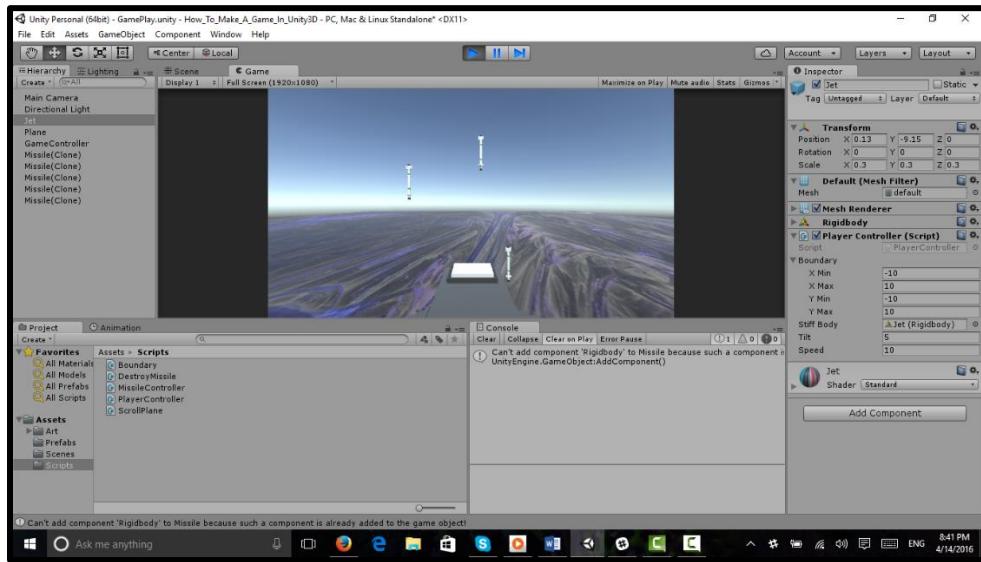


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

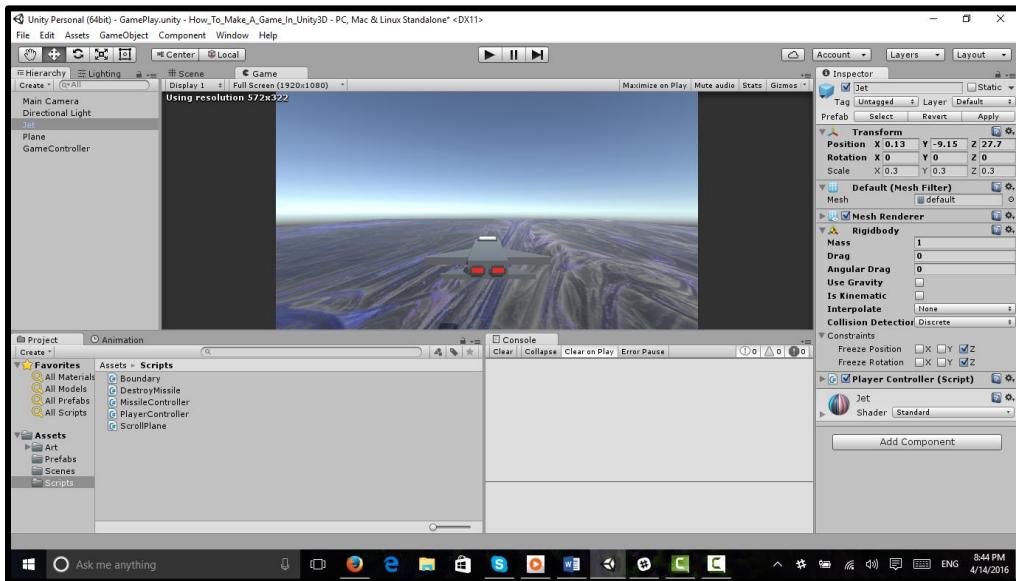
STEP 5:

Run the game and let's see how it goes.



Whoops, looks like something is over writing our jet's z position. This is why testing is important. We all make mistakes, including myself. So, let's go back over the player controller script to see where the problem lies.

The problem does not lie in the code, rather, it lies in the constraints section of the Rigid body component. We need to make sure that we freeze the position and rotation on the z axis.



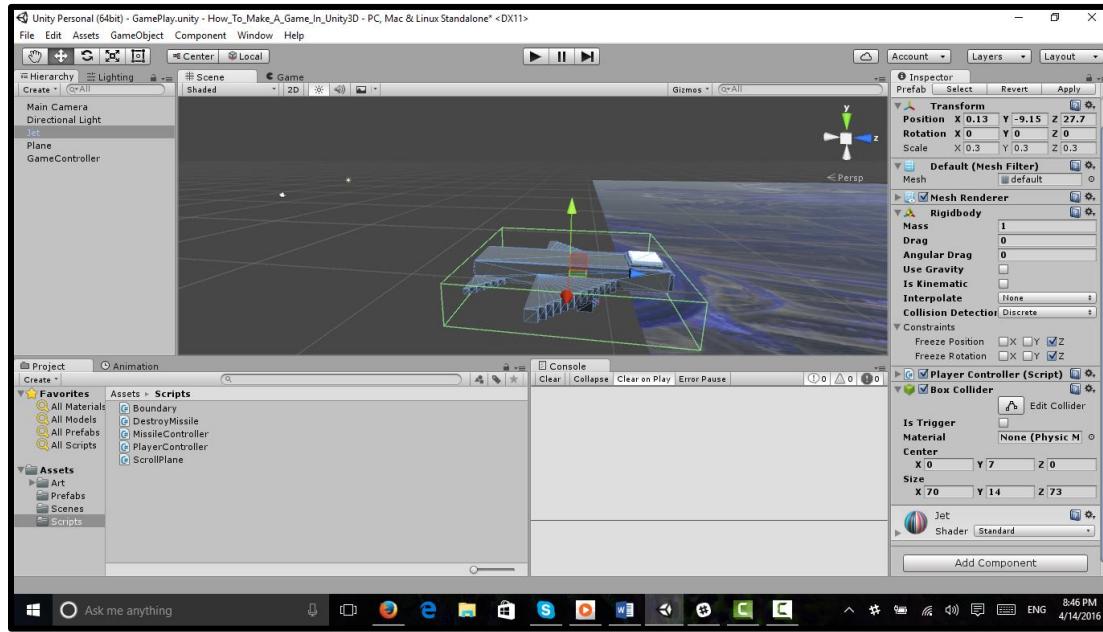
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

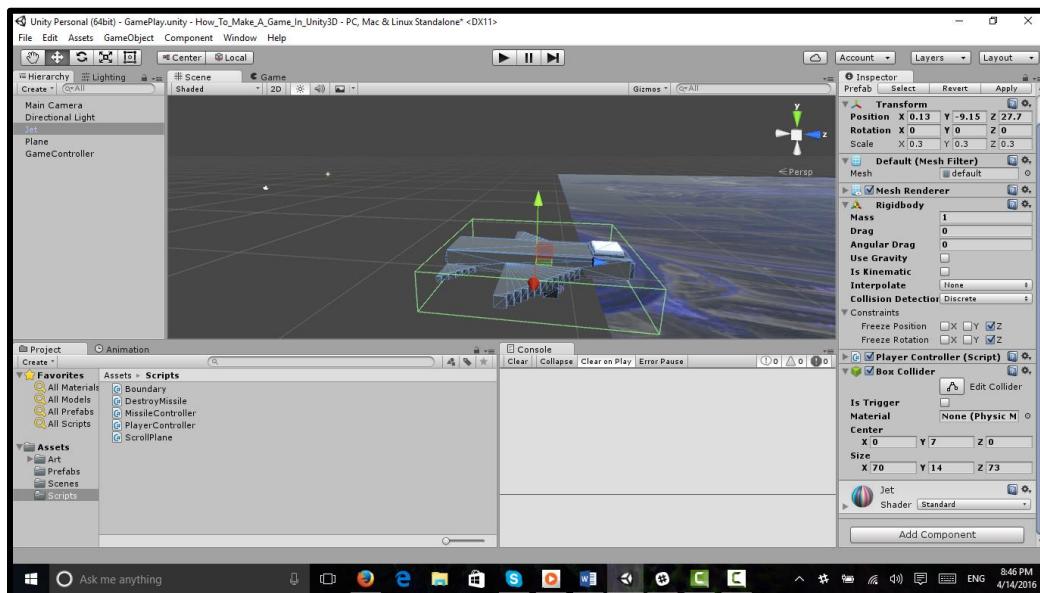
Now, this isn't too much of a game currently even though it runs correctly now. We need to add some collisions to make the hazards... Well... hazardous!

STEP 6:

Select the Jet and add component, physics, box collider.



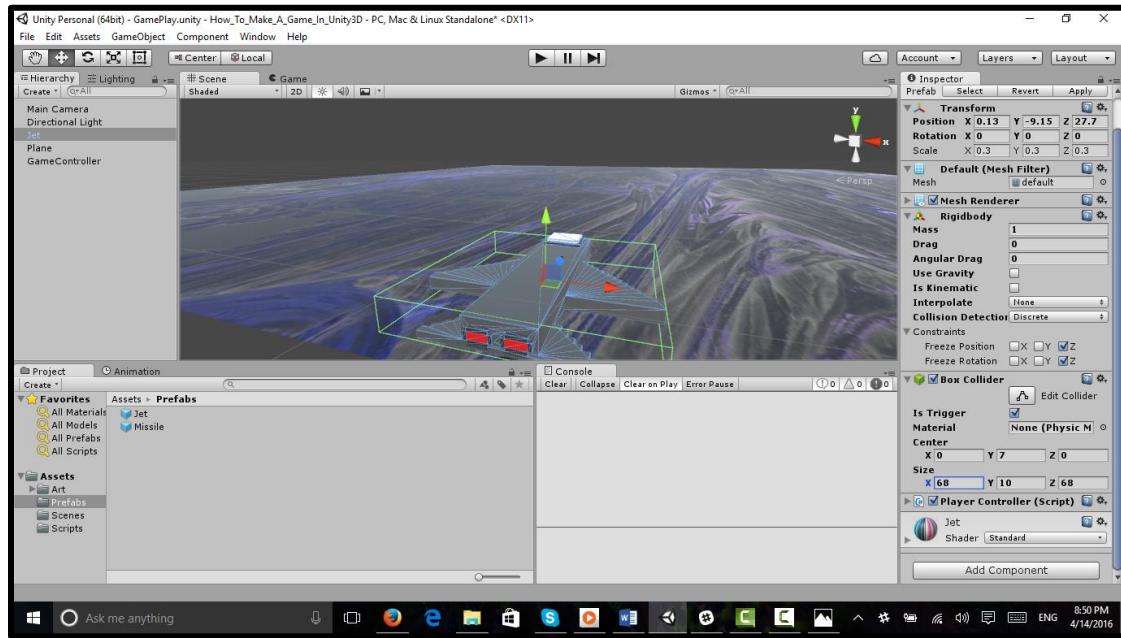
Switch from game to scene view to take a look at how the box collider would be around the Jet.



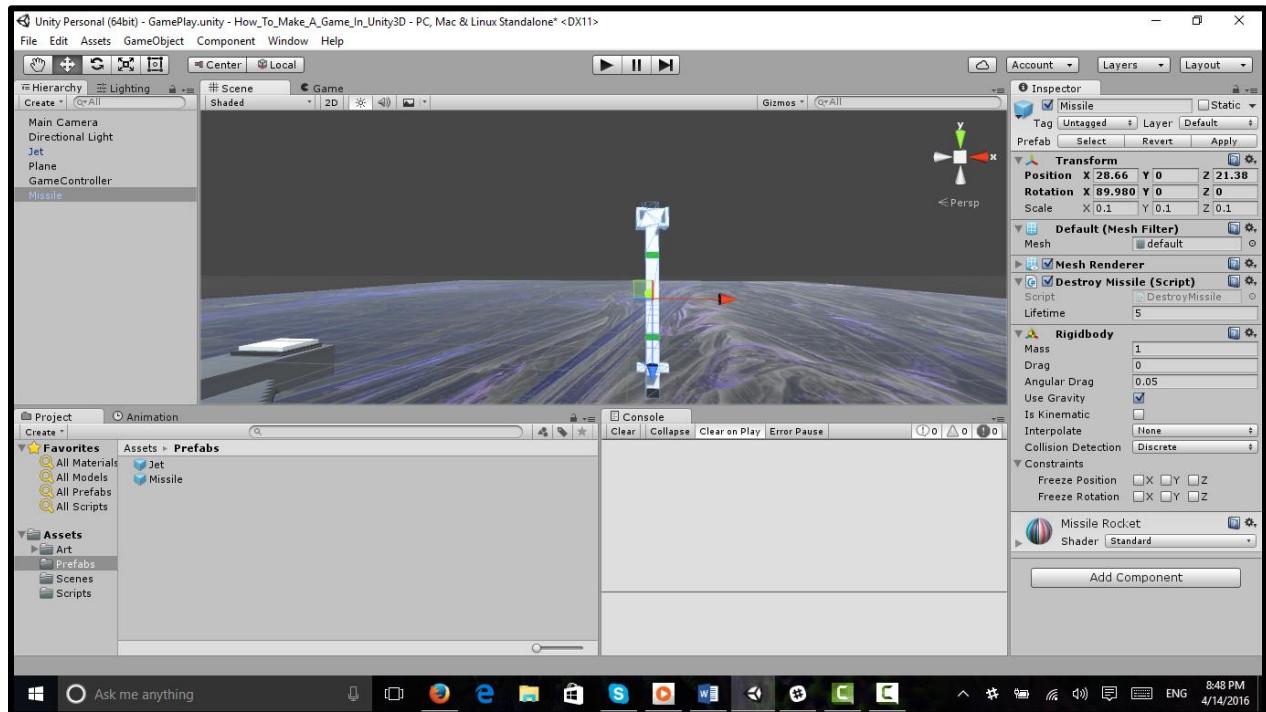
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Set the size to be as followed. X 68, Y 10, Z 68 and put a check in the box for is trigger.



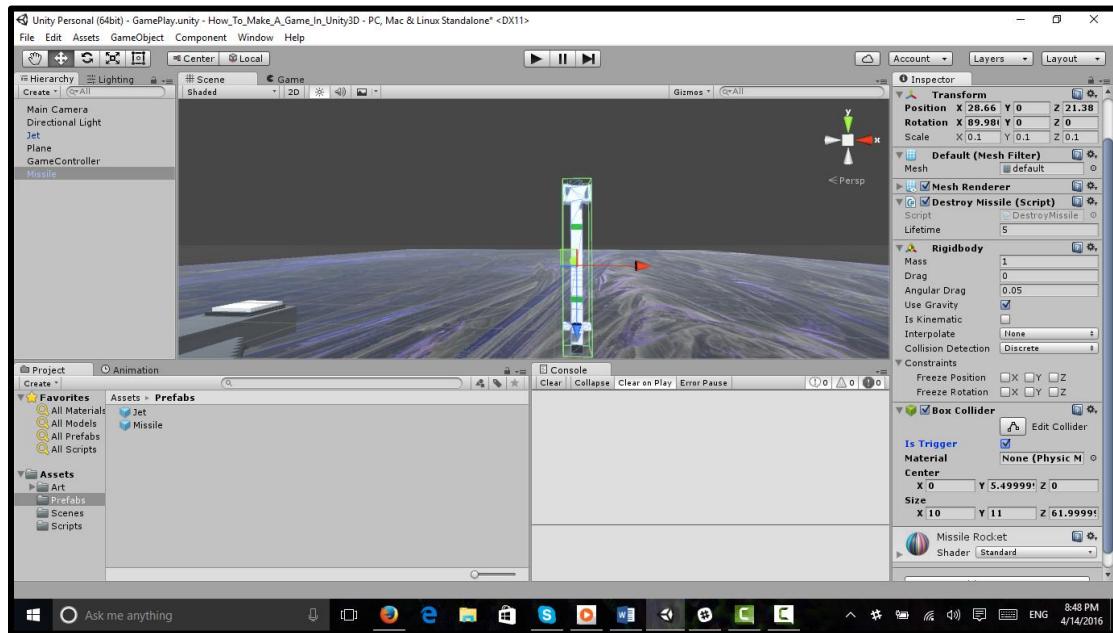
We now need to do essentially the same thing with the Missile. I suggest you put a copy of the prefab on screen for editing.



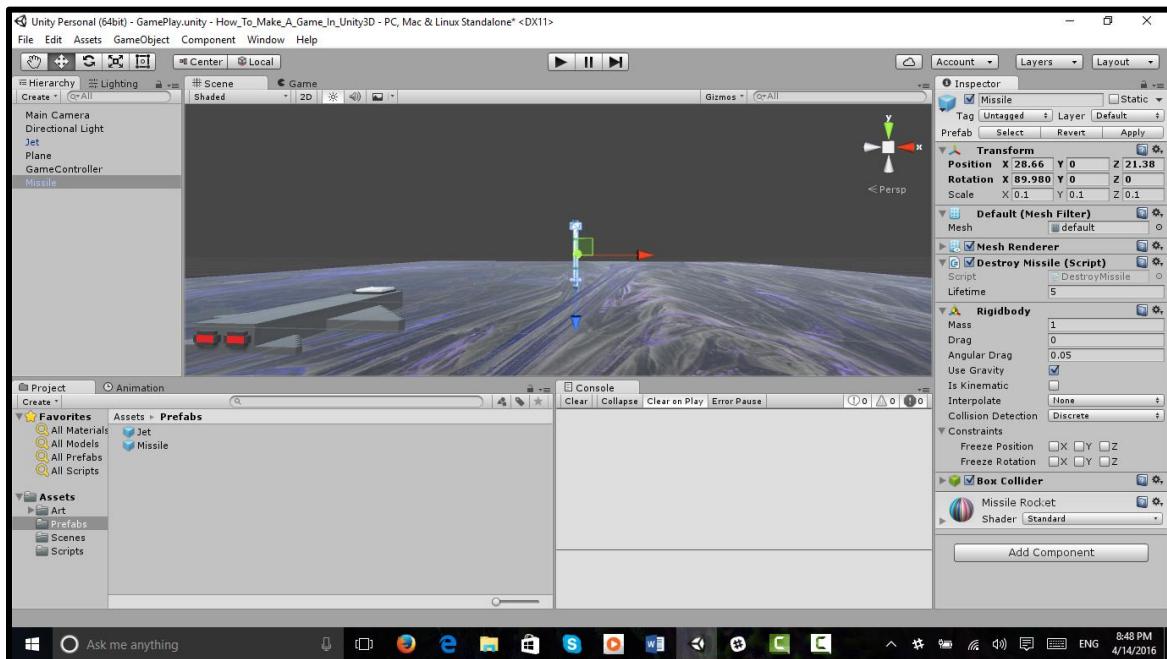
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Add the box collider component to it and Unity should auto size it to be perfect. Make sure Is Trigger is checked.



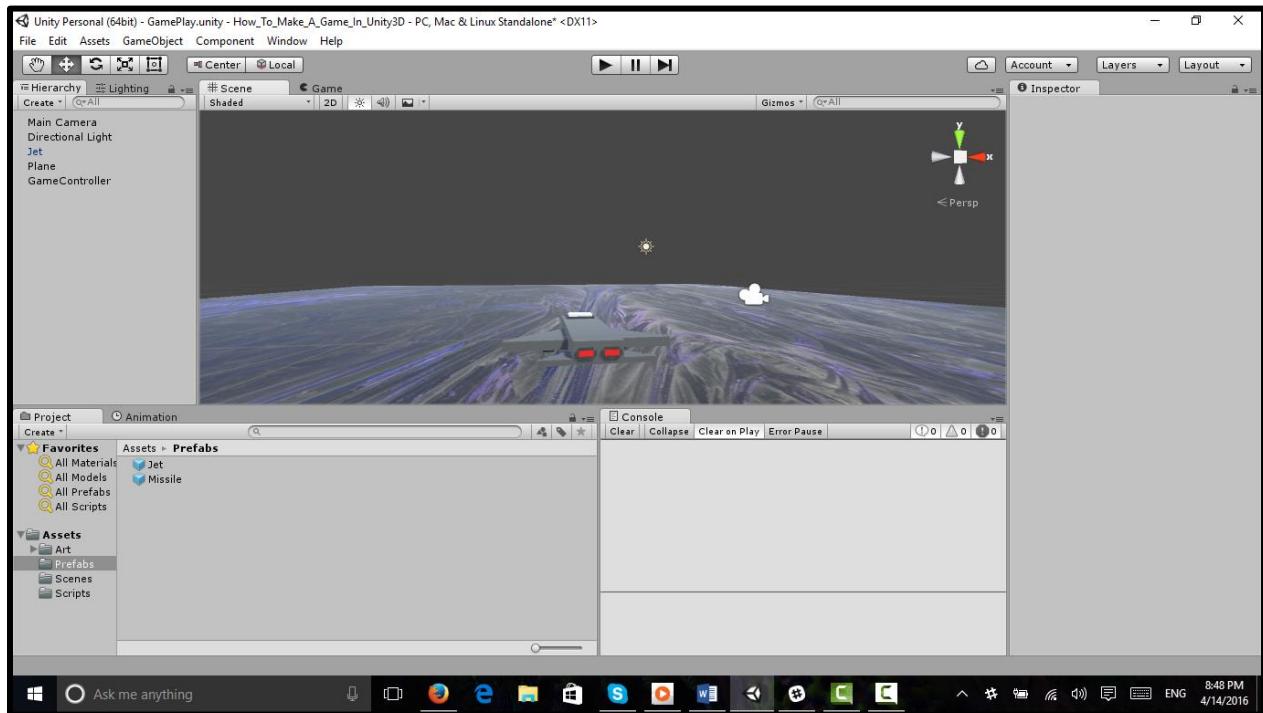
Click the apply button at the top of the inspector pane. This will save the changes to the prefab itself.



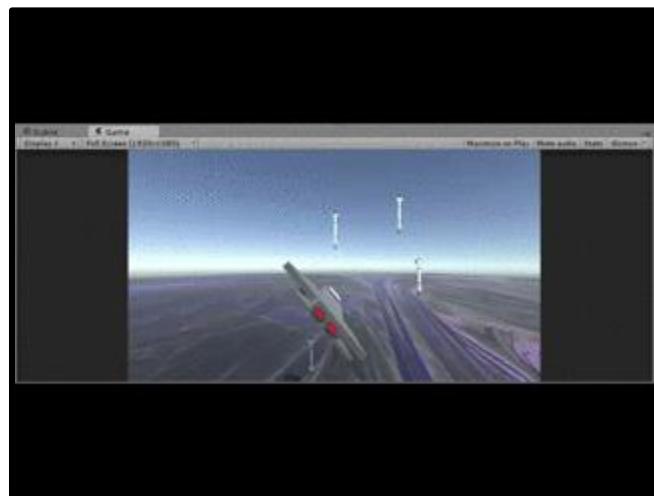
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Now you can delete the Missile from the hierarchy pane.



Here are the final results:



Side Note:

If Unity keeps giving you issues about the rigid body component on the Missile, you can delete the code that references it in the Missile Controller Script or simply remove the rigid body from the missile in the Unity editor.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Revision to the code for the missile controller, in case you want to go that route.

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class MissileController : MonoBehaviour
5  {
6      public GameObject hazard;
7
8      private float spawnWait = 1f;
9
10     void Start()
11     {
12         StartCoroutine(SpawnWaves());
13     }
14
15     IEnumerator SpawnWaves()
16     {
17         while (true)
18         {
19             var xMinMax = Random.Range(-15f, 20f);
20             Vector3 spawnPosition = new Vector3(xMinMax, 10f, 25f);
21             Quaternion spawnRotation = Quaternion.Euler(new Vector3(90f, 0f, 0f));
22             Instantiate(hazard, spawnPosition, spawnRotation);
23             yield return new WaitForSeconds(spawnWait);
24             spawnWait -= 0.01f;
25             if (spawnWait < 0.05f) spawnWait = 0.05f;
26         }
27     }
28 }
```

Well, that's it for today. To reiterate what we have gone over, we have set up the basic game skeleton for the game play. We expanded upon it by adding and writing the scripts needed for the game play. We added all of the images and models we needed for the basic game play as well. Part 3 will cover the remaining portions for making this a full on game. I hope to see you there. May your code be robust and bug free, This is Jesse signing out.

How to make a game in Unity3D – Part 3

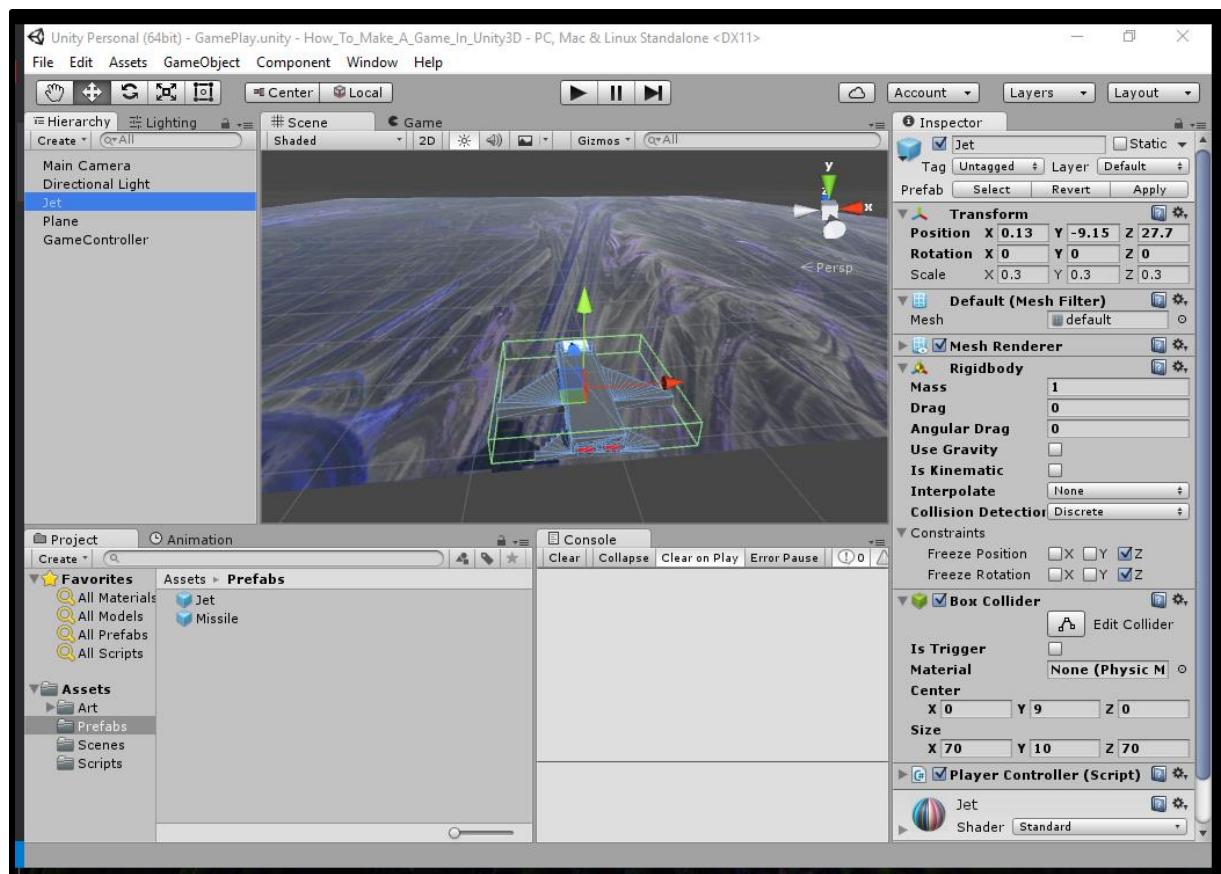
By Jesse Glover

Today, we will be finishing up on the game we have started making in part 1 and part 2. So, to recap on what we need to do to finish off the game is to add the music, leaderboard, start screen, game over screen, and the leaderboard display screen. I hope you guys are ready, this will go by rather quickly and give you the remaining information you need to finish this game and make other games.

Download the project zip file [here](#).

Issue Fixing

To get started, let's fix an issue that I failed to address and notice with part 2. The issue lies with the box collider we set up. The Is Trigger parameter should be left unchecked and I modified the parameters for the size of the box to better fit the jet. For center x should be 0, y should be 9, and z should be 0. The size should be changed as well, x is 70, y is 10, and z is 70.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

We should also do a slight fix to the code as well. We will remove the line that deals with attaching the rigid body component to each and every missile. However, this causes issues (internal issues to the Unity Engine).

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class MissileController : MonoBehaviour
5 {
6     public GameObject hazard;
7
8     private float spawnWait = 1f;
9
10    void Start()
11    {
12        StartCoroutine(SpawnWaves());
13    }
14
15    IEnumerator SpawnWaves()
16    {
17        while (true)
18        {
19            var xMinMax = Random.Range(-15f, 20f);
20            Vector3 spawnPosition = new Vector3(xMinMax, 10f, 25f);
21            Quaternion spawnRotation = Quaternion.Euler(new Vector3(90f, 0f, 0f));
22            Instantiate(hazard, spawnPosition, spawnRotation);
23            yield return new WaitForSeconds(spawnWait);
24            spawnWait -= 0.01f;
25            if (spawnWait < 0.05f) spawnWait = 0.05f;
26        }
27    }
28 }
```

This should fix the issue with the missiles and jet not always registering as being hit. In other words, the hit detection was extremely buggy.

Backend Code

Next up, we should create the back end for the leaderboard. To get started, we should first decide what format we will use for it. I have chosen to go with XML because it is the easiest method to use. We will start off by building the back end code, mainly because the front end is the easiest part to do and should be saved for last.

So, to get started we need to make the Leaderboard class. Let's look at the entire script and break it down from there.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

```
1 using UnityEngine;
2 using System.Collections;
3 using System.Xml;
4 using UnityEngine.UI;
5
6 public class Leaderboard : MonoBehaviour
7 {
8     public Text text;
9     public GameObject content;
10
11    void Start()
12    {
13        string Path = Application.dataPath + "/Data/Leaderboard.xml";
14
15        XmlDocument doc = new XmlDocument();
16
17        if (!System.IO.File.Exists(Path))
18        {
19            Debug.Log("File not found");
20        }
21    }
```

```
22    else
23    {
24
25        doc.Load(Path);
26
27        XmlNodeList elemList = doc.GetElementsByTagName("SurvivalTime");
28        for (int i = 0; i < elemList.Count; i++)
29        {
30            text = Instantiate<Text>(text);
31            text.transform.SetParent(content.transform, false);
32            text.text = elemList[i].InnerText;
33        }
34    }
35 }
36 }
```

```
37    public void writeToXML()
38    {
39        text = FindObjectOfType<Text>();
40        string Path = Application.persistentDataPath + "/Leaderboard.xml";
41
42        XmlDocument doc = new XmlDocument();
43    }
```

```

44     if (!System.IO.File.Exists(Path))
45     {
46         XmlDeclaration declaration = doc.CreateXmlDeclaration("1.0", "UTF-8", "yes");
47         XmlComment comment = doc.CreateComment("This is a generated XML File");
48         XmlElement Leaderboard = doc.CreateElement("Leaderboard");
49         XmlElement survivalTime = doc.CreateElement("SurvivalTime");
50
51         survivalTime.InnerText = text.text;
52
53         doc.AppendChild(declaration);
54         doc.AppendChild(comment);
55         doc.AppendChild(Leaderboard);
56
57         Leaderboard.AppendChild(survivalTime);
58
59         //Save document
60         doc.Save(Path);
61     }

```

```

63     else //if file already exists
64     {
65         doc.Load(Path);
66
67         // Get root element
68         XmlElement root = doc.DocumentElement;
69
70         XmlElement survivalTime = doc.CreateElement("SurvivalTime");
71
72         //Values to the nodes
73         survivalTime.InnerText = text.text;
74
75
76         //Document Construction
77         doc.AppendChild(root);
78
79         // Append root element to word element
80         root.AppendChild(survivalTime);
81
82         //Append written values to word as child element
83
84         //Save the document
85         doc.Save(Path);
86     }
87 }
88 }

```

READING AND WRITING XML DATA

Since we want the data from the timer that we will build, we need to make sure we get the text element from our timer. Our timer will be a 3D text object, in code that is represented by a TextMesh object. We have two methods here, Load XML and Write to XML.

Loading the XML is a really simple method. The first thing that the method does is load the xml file. Next up, we declare what element we actually want the data from. The last thing it does is loop through all of the elements with the tag Survival Time and display the results.

```
1 void Start()
2 {
3     string Path = Application.dataPath + "/Data/Leaderboard.xml";
4
5     XmlDocument doc = new XmlDocument();
6
7     if (!System.IO.File.Exists(Path))
8     {
9         Debug.Log("File not found");
10    }
11
12    else
13    {
14
15        doc.Load(Path);
16
17        XmlNodeList elemList = doc.GetElementsByTagName("SurvivalTime");
18        for (int i = 0; i < elemList.Count; i++)
19        {
20            text = Instantiate<Text>(text);
21            text.transform.SetParent(content.transform, false);
22            text.text = elemList[i].InnerText;
23        }
24    }
25 }
```

Next up, we have the Write to XML code. It is definitely a lot more code than the loading code. Essentially, this code contains a lot of boiler plate that specifies exactly how we want the xml file to be generated as well as the location. Let's get into it and break it down.

Here we specify that the text object needs to find the object of the type Text Mesh. This will allow us to much more easily call this code in another class if we need to. We create a string called path. We assign the path to be the application's data path, and add that we want the file name to be Leaderboard and it is an xml file.

Let me explain exactly what that is. It contains the path that houses the game's data folder. Which is 100% different from the application's persistent data path. More on that later.

Finally, We define doc as a new Xml Document.

```
1 public void writeToXML()
2 {
3     text = FindObjectOfType<Text>();
4     string Path = Application.persistentDataPath + "/Leaderboard.xml";
5
6     XmlDocument doc = new XmlDocument();
7
8     if (!System.IO.File.Exists(Path))
9     {
10         XmlDeclaration declaration = doc.CreateXmlDeclaration("1.0", "UTF-8", "yes");
11         XmlComment comment = doc.CreateComment("This is a generated XML File");
12         XmlElement Leaderboard = doc.CreateElement("Leaderboard");
13         XmlElement survivalTime = doc.CreateElement("SurvivalTime");
14
15         survivalTime.InnerText = text.text;
```

If file and path does not exist, then we need to build the entire xml structure. Declaration creates the header of the xml file. Comment, while it isn't needed to have a valid xml file, is added to say we didn't build it by hand. Leaderboard is the parent element we need to use. Survival time is the child element that will house the timer's data.

We will set the survival time's inner text to be whatever the text mesh's text is. We then append the child for the doc as declaration, comment, and leaderboard. Leaderboard will have its own child element of survival time. Then, we can save the document to the path specified.

```

1      doc.AppendChild(declaration);
2      doc.AppendChild(comment);
3      doc.AppendChild(Leaderboard);
4
5      Leaderboard.AppendChild(survivalTime);
6
7      //Save document
8      doc.Save(Path);
9  }
10
11     else //if file already exists
12  {
13     doc.Load(Path);
14
15     // Get root element
16     XmlElement root = doc.DocumentElement;
17
18     XmlElement survivalTime = doc.CreateElement("SurvivalTime");
19
20     //Values to the nodes
21     survivalTime.InnerText = text.text;
22
23
24     //Document Construction
25     doc.AppendChild(root);
26
27     // Append root element to word element
28     root.AppendChild(survivalTime);
29
30     //Append written values to word as child element
31
32     //Save the document
33     doc.Save(Path);

```

If the file and path does exist, then load the document. We need to retrieve the root element which in our case is the document's main element (in our case is leaderboard) and we also want to make sure that we can add more survival time elements.

We again set the survival time's inner text to be the text mesh's text. We then set that the root element of the document can be added to. Finally, we can append or add more to the survival time of the leaderboard element. The last step is to save the document to the path specified.

Not too difficult, just very tedious in the way I decided to go with creating and modifying the Leaderboard with xml. There are easier ways to handle this, but it is always a good idea to learn exactly what is going on behind the scenes with XML and/ or JSON.

Timing and timer

Let's start the timer code. It is rather simple and straight forward. We will begin by creating a public Text Mesh, a private float, and two private integers. It should look like the code below.

```
1  public Text scoreText;
2
3  private int minutes;
4  private int seconds;
5
6  private float Score;
```

The only method we will have is an Update method. The reason for this is because it is easier to consolidate into a single method that handles this one task.

The float value of score should be addition and assigned to delta time.

The int of minutes should be assigned to the Mathf class and floored to int the score divided by 60.

The int of seconds should be assigned to the Mathf class and floored to int the score subtracted by the minutes multiplied by 60.

Next we format the string value to have the minutes and second display.

Finally, the Text Mesh's text should have the text "Survival Time: " and directly underneath it the formatted string value we created.

```
1  void Update()
2  {
3      Score += Time.deltaTime;
4      minutes = Mathf.FloorToInt(Score / 60f);
5      seconds = Mathf.FloorToInt(Score - minutes * 60f);
6      string formatedTime = string.Format("{0:0}:{1:00}", minutes, seconds);
7      scoreText.text = "Survival Time: " + System.Environment.NewLine + formatedTime;
8  }
```

Modification for Collision events and writing XML Data

The next to last bit of code that needs to be done for now is to modify the Destroy Missile class. We will change it from on trigger enter to on collision enter and create a new method called save Results. We will call the leaderboard class, name it leader, and game object add component leaderboard on it. Leader should then have the ability to call the write to xml method from that class.

```
1 void OnCollisionEnter(Collision other)
2 {
3     Destroy(other.gameObject);
4     saveResults();
5     Destroy(gameObject);
6     SceneManager.LoadScene(2);
7 }
8
9 void saveResults()
10 {
11     Leaderboard leader = gameObject.AddComponent<Leaderboard>();
12     leader.writeToXML();
13 }
```

Scene Transitions

The final portion of the code we need to discuss is the scene transition class. This is relatively simple to write and utilize. We call the Scene Management class from within Unity Engine. Then to call the scene we want to load, we utilize the Scene Manager class and call the Load Scene method.

I should note that this can take an integer value or a string as the parameter. I typically use the integer method and you have to make sure that you go to your build settings and assemble the scenes appropriately. If you mix up the integer values, the scenes will load but they will be in the incorrect order.

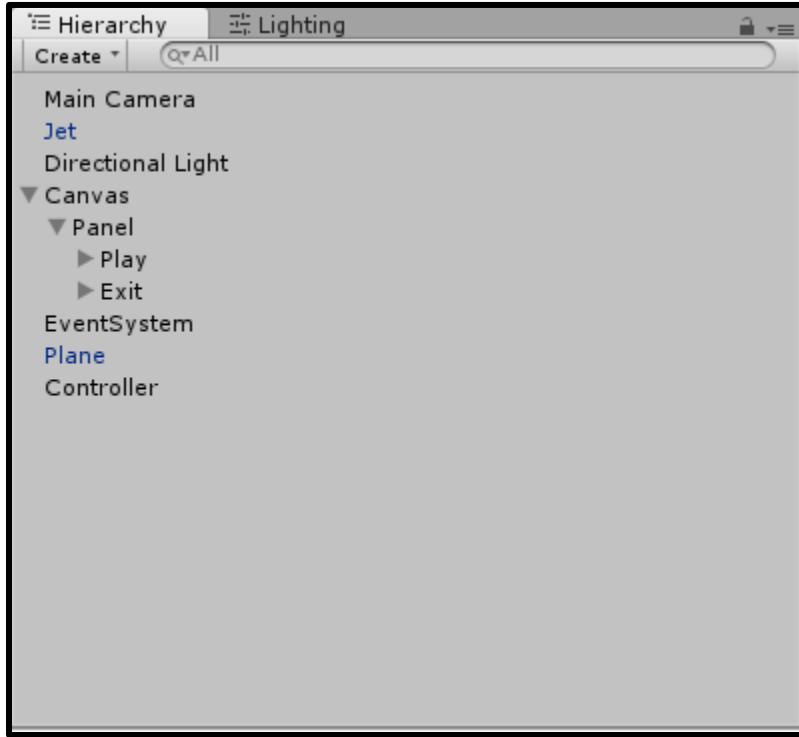
```
1  using UnityEngine;
2  using UnityEngine.SceneManagement;
3
4  public class SceneController : MonoBehaviour
5  {
6      public void StartGame()
7      {
8          // New Game
9          SceneManager.LoadScene(1);
10     }
11
12     public void Leaderboard()
13     {
14         // Game Over with Leaderboard
15         SceneManager.LoadScene(2);
16     }
17
18     public void EndGame()
19     {
20         // Exit Game
21         Application.Quit();
22     }
23 }
```

Game Start Scene Building

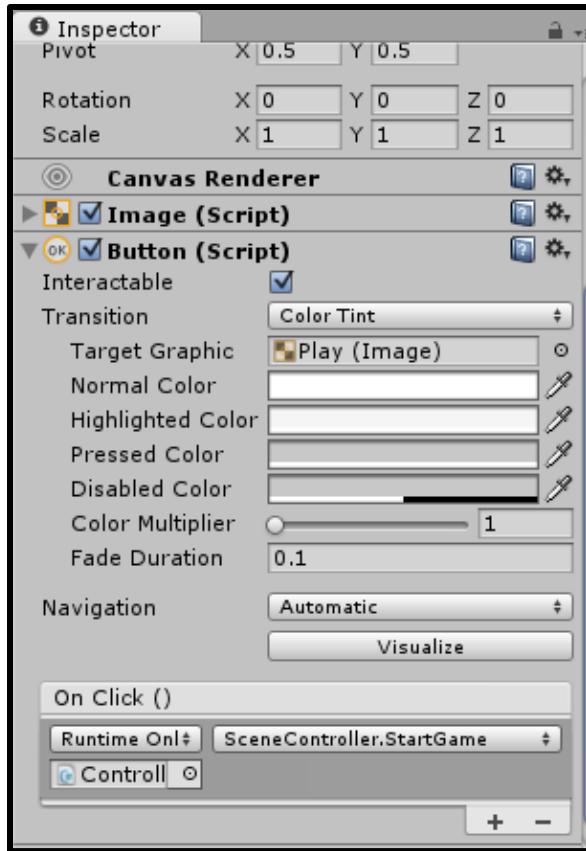
This scene will be utilizing our Jet and Plane prefabs, as well as other components. Make sure the scene is a 3D scene and add the jet and plane to the scene. The jet's position should be (x = 0, y = 3.7, z = 24.75) and should be scaled correctly however, just in case it didn't; The correct values for scaling is (x = 0.3, y = 0.3, z = 0.3). You can remove the player script, box collider, and ridged body from the Jet on this scene if you would like.

The plane should have the scale values set as (x = 100, y = 100, z = 100). The position of the plane should be (x = 0, y = -38.23, z = 545.27).

Add an Empty game object and call it Controller. Add a canvas, panel, and 2 buttons to the scene. The canvas should be the primary parent object, the panel should be a direct child of the canvas, and the two buttons are a direct child of the panel. Name the buttons start and exit respectively.



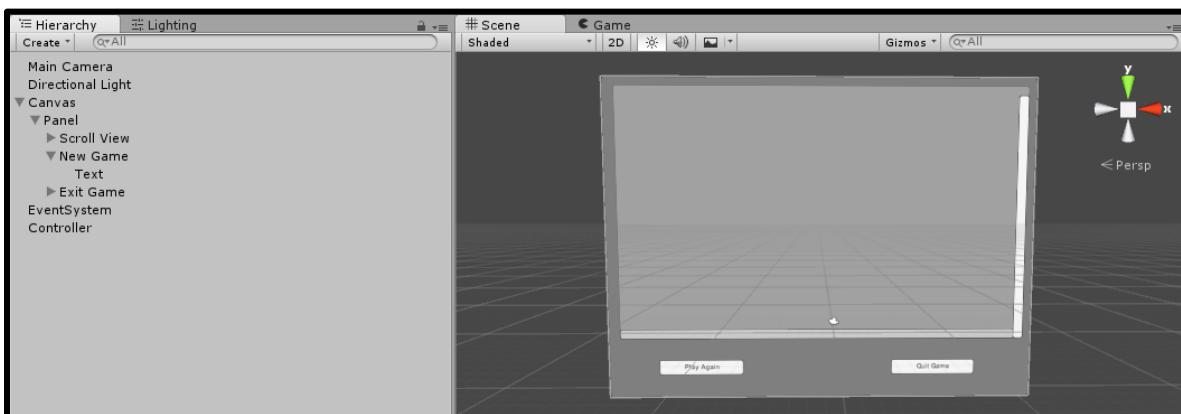
Attach the scene controller script to the controller game object we just created. Next, attach the game object to each button script from within the button on the hierarchy pane. Assign either start or exit to their respective buttons.



Save the Scene and name it Game Start End, and we have completed the scene.

Game Over Scene Building

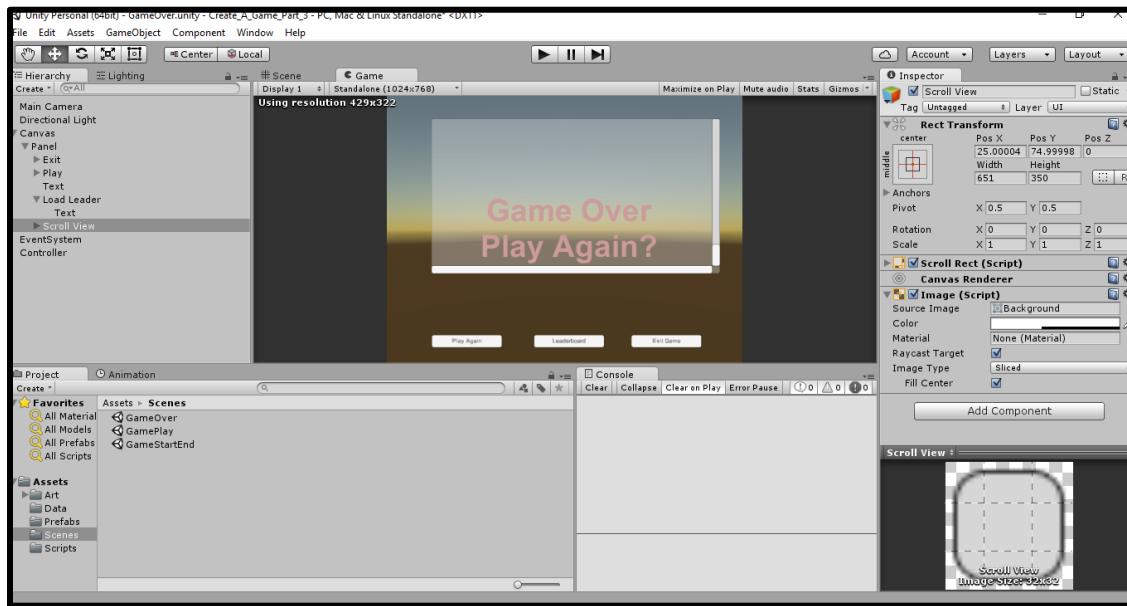
Add a Canvas, Panel, Scroll View, Empty Game Object, and 2 buttons to the scene. The Canvas is the parent Object, the 2 buttons, Empty Game Object and Panel are direct child objects to the Canvas, and the scroll View is a direct child object to the Panel.



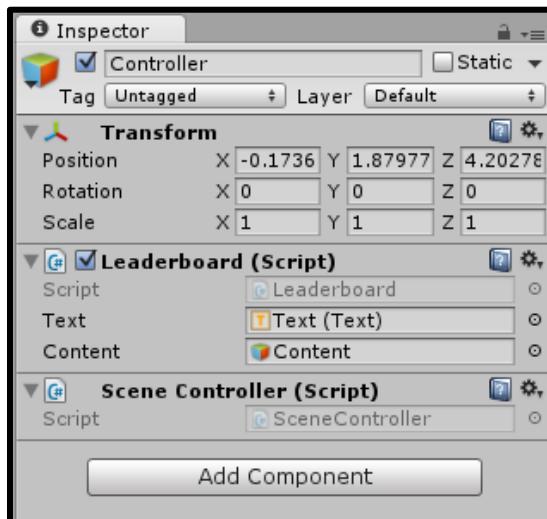
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Rename the Empty Game object to be Controller as we did in the last scene. Attach the scene management and Leaderboard scripts to it.

Create a text prefab and place it into the prefabs folder, we will need this for the next step. Navigate to the Scroll view and look for an object inside called Content.



The next step is to make sure that we add a vertical layout group to the content object. This will allow us to structure the displayed text the way we want instead of how Unity would display by default. The only thing we want unchecked is the child force expand on width. The remainder of the properties should be as written here: Left = 0, Right = 0, Top = 0, Bottom = 0, Space = 30, and child alignment = middle center.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

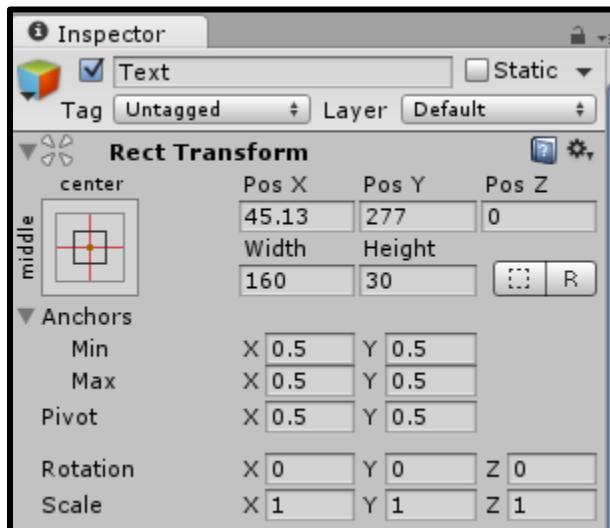
© Zenva Pty Ltd 2021. All rights reserved

With the Leaderboard script that we attached, you probably noticed that it requires two game objects; A Content and Text object. The Content object we will use is the content object from within the Scroll View, and the text is from the text object that we just created a prefab from.

The New Game and Exit Game buttons should have their respective scripts attached to them as we did in the scene before it.

Finalizing the Game Play Scene

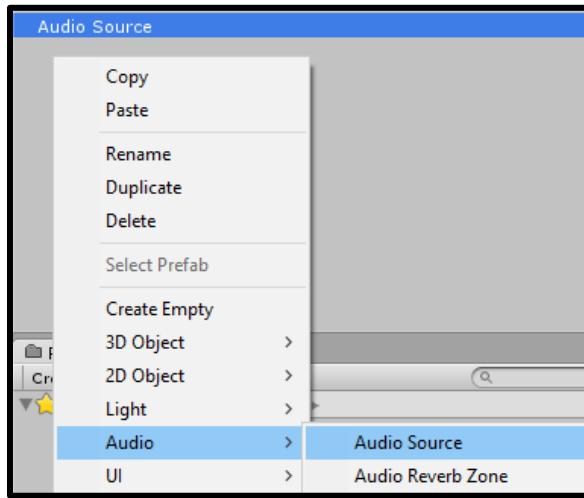
Add a Canvas to the scene and create a child object of text. The canvas should be set to screen space camera just like the other scenes. The text should have the position of x = 45.13, y = 27, z = 0, width = 160, and height = 30.



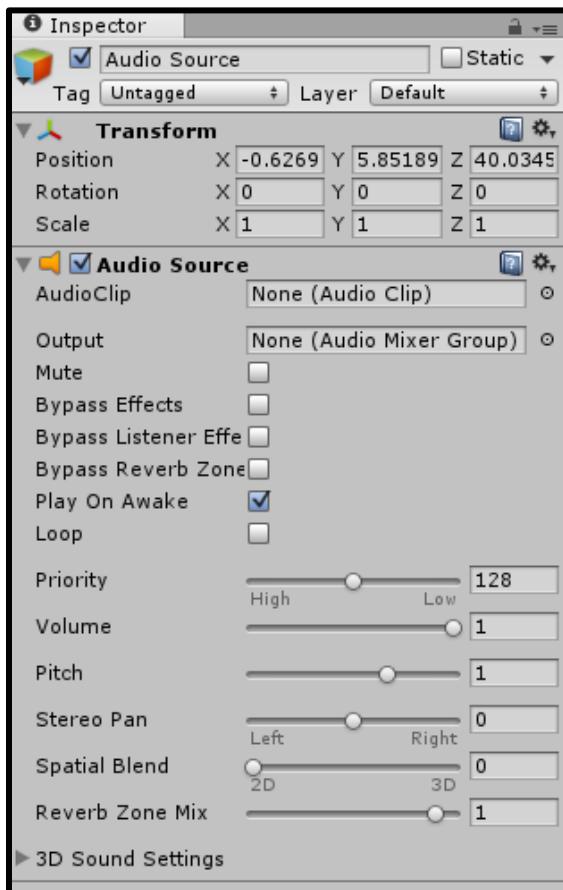
We have now completed the game play scene.

As you can see, putting together all of the little aspects to create a game is very simple to do with Unity3D, the only thing we have not done is add music. So, let's go ahead and do that shall we?

Right click on the hierarchy pane, highlight audio, then click on audio source.



Now that we have the audio source added to the scene, let's take a look at some of the properties that are available for us to manipulate.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

We have the source of the audio, option to specify the output device, mute, bypass effects, bypass listener effects, Bypass reverb zone, play on awake, loop, priority, volume, pitch, stereo pan, spatial blend, and reverb zone mix. We have a drop down for 3D sound manipulation we can do as well.

We want to make sure that Play on Awake and Loop are checked. Priority is set to 128, Volume is 0.4, Pitch is 1, Stereo Pan is 0, Spatial Blend is 0.5, and Reverb Zone Mix is 1.



Congratulations, you have a game that has music, the ability to save and load data, controllable character, and collision detection. I hope you enjoyed this series on building a game in Unity3D, I have plenty more to teach you in the future. Until then, may your code be bug free; Jesse signing out.

How to Create an RPG Game in Unity – Comprehensive Guide

By Renan Oliveira

In this tutorial we are going to build an RPG game using Unity. Our game will have three scenes: a title scene, a town scene, that represent the game world, where the player can navigate and find battles, and the battle scene. The battles will have different types of enemies, and the player units will have different actions such as: regular attack, magic attack, and running from the battle.

In order to follow this tutorial, you are expected to be familiar with the following concepts:

- C# programming
- Using Unity inspector, such as importing assets, creating prefabs and adding components
- Basic Tiled map creation, such as adding a tileset and creating tile layers

Before starting reading the tutorial, create a new Unity project and import all sprites available through the source code. You will need to slice the spritesheets accordingly.

Source code files

You can download the tutorial source code files [here](#).

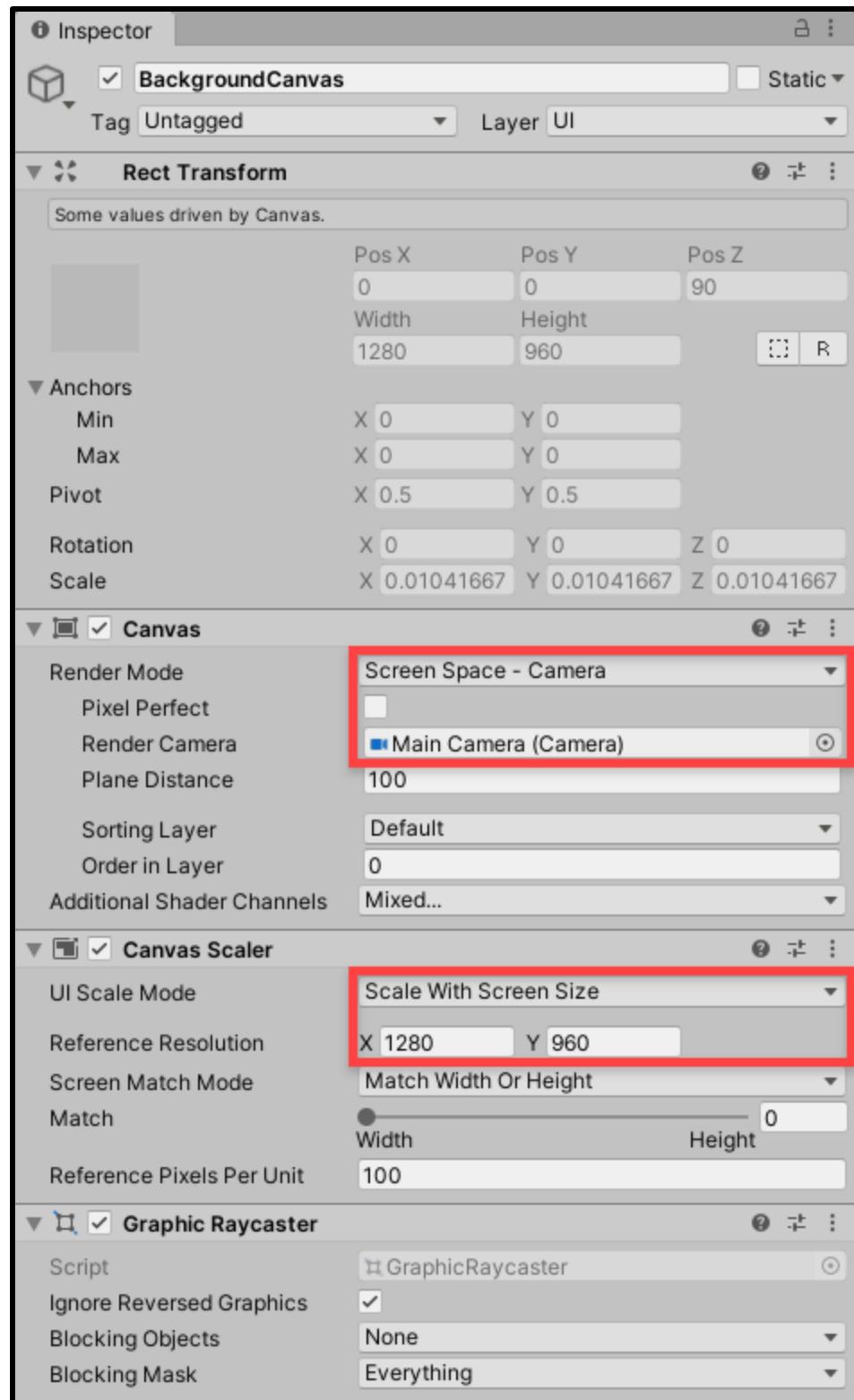
Assets copyright

The sprites used in this tutorial are from the [Superpowers Asset Packs](#) by [Pixel-boy](#). All sprites are available under the Creative Commons Zero (CC0) license. So, you can use them in your games, even commercial ones.

Title Scene

Background canvas

First of all, we are going to create a Canvas to show the background image in the Title Scene. You can do that by creating a new Canvas called BackgroundCanvas, and setting its render mode as Screen Space - Camera. In order to do so, we need to specify the camera of the canvas, which will be our main camera. Also, the UI Scale Mode (in Canvas Scaler) will be set to follow the screen size, with a reference resolution of 1280x960.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

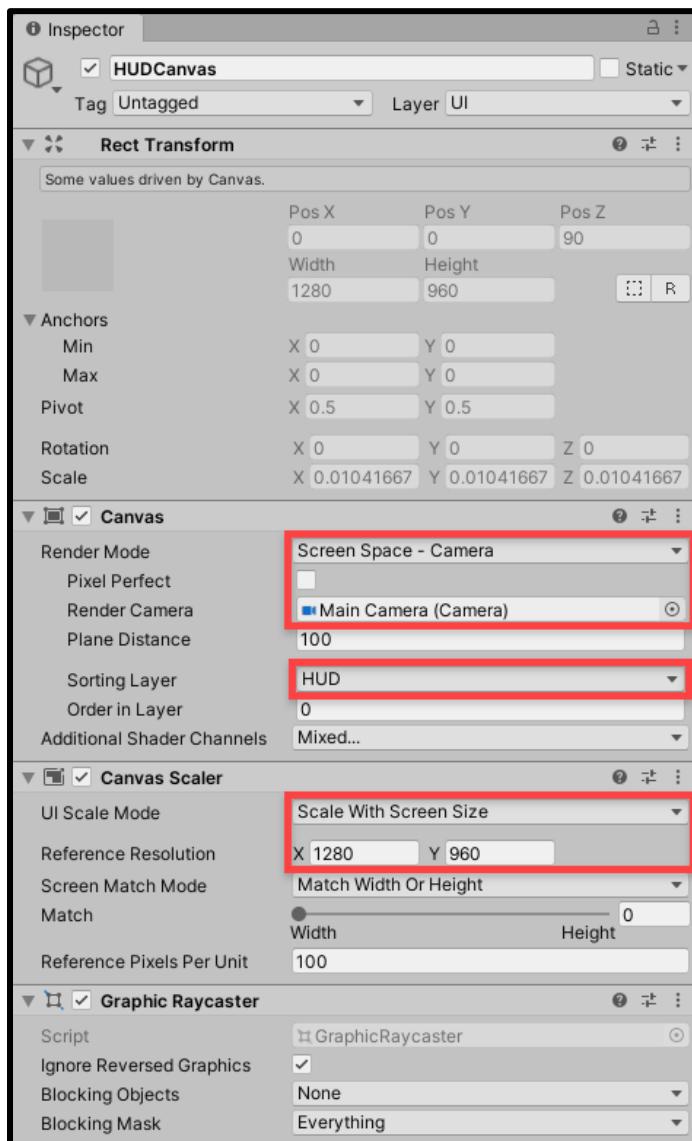
© Zenva Pty Ltd 2021. All rights reserved

After doing that, we create a new Image object as a child of this canvas. The source image will be the background image, and we can set its native size in order to properly show it.

HUD canvas

Now, we need another Canvas to show the HUD elements. In the Title Scene, those elements will be a title text, and a play button.

Let's start by creating another Canvas following the same process as the BackgroundCanvas. However, in order to show this canvas over the background one, we need to properly set its sorting layer. We are going to create another layer called **HUD**, and put the **HUDCanvas** on this layer.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Finally, we need to create the two HUD objects. The first one is a Text, while the second one is a Button. For the Text object we only need to set its message. On the other hand, for the Button object we need to set its OnClick callback.





This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

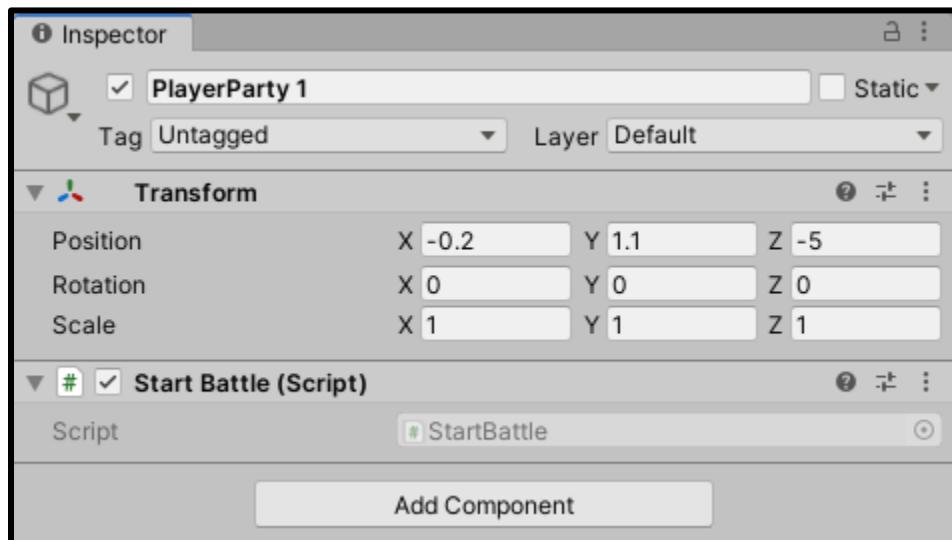
© Zenva Pty Ltd 2021. All rights reserved

The PlayButton object will have the following ChangeScene script for the OnClick callback. This script will simply define a method to start a new scene given its name. Then, we set the OnClick callback of the play button to call the loadNextScene method with "Town" as parameter.

```
1 public class ChangeScene : MonoBehaviour
2 {
3     public void loadNextScene (string sceneName)
4     {
5         SceneManager.LoadScene(sceneName);
6     }
7 }
```

Player party

In our game, we want to keep the player units data saved even when changing scenes. In order to do so, we are going to create a PlayerParty persistent object, which won't be destroyed when changing scenes. You will also need to properly set its position so that it will be created in the correct position in Battle Scene.



In order to keep the PlayerParty alive when changing scenes, we are going to use the following script, called StartBattle. This script keeps the object from being destroyed when changing scenes in the Start method. It also adds a callback when a new scene is loaded and set the object as inactive, so that it won't be shown in the title screen.

The configured callback (OnSceneLoaded) checks if the current loaded scene is the Title scene. If so, the PlayerParty object must be destroyed, to avoid duplicates. Otherwise, it should be set as active if the current scene is the Battle scene.

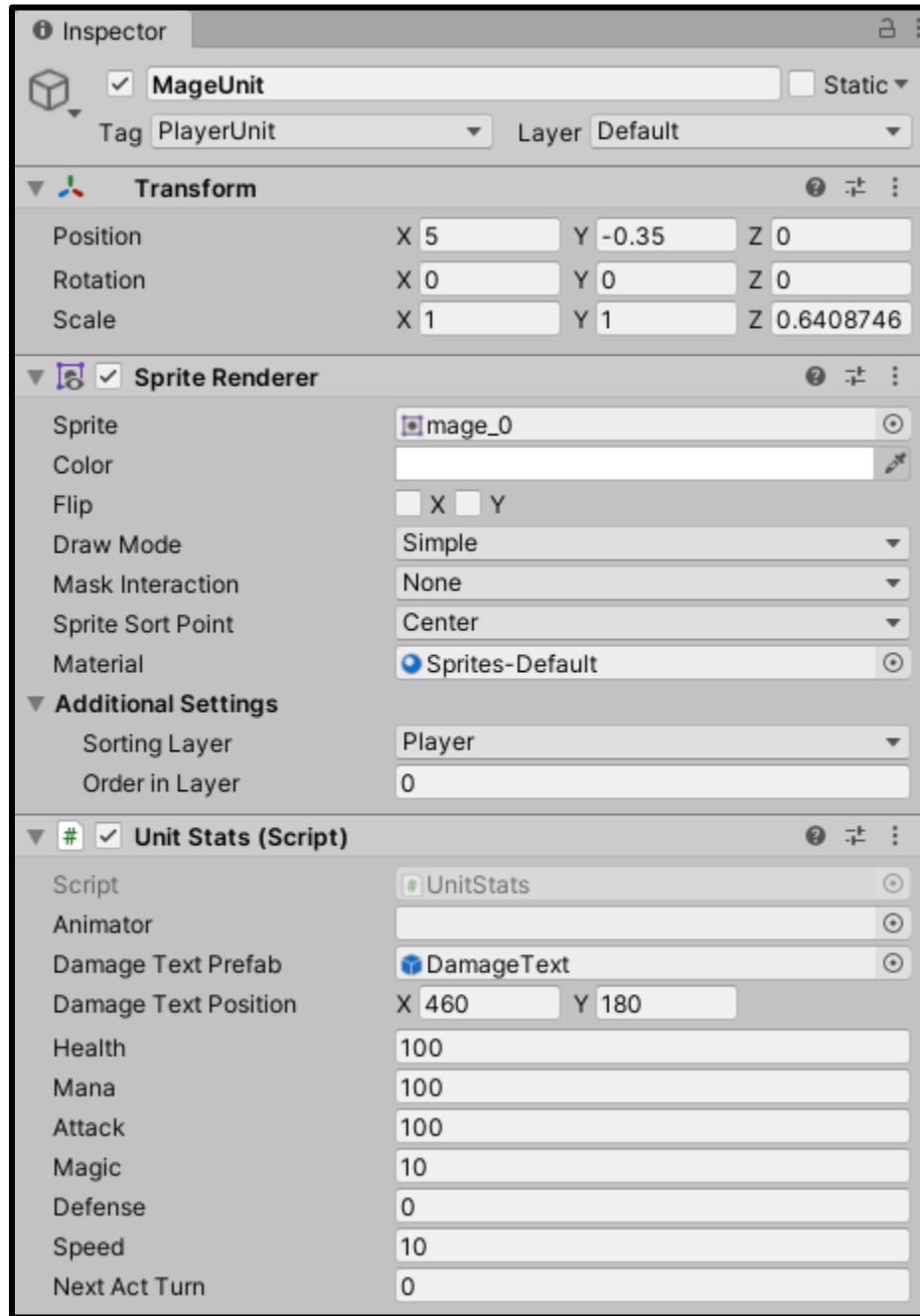
```
1 public class StartBattle : MonoBehaviour
2 {
3     // Use this for initialization
4     void Start ()
5     {
6         DontDestroyOnLoad(this.gameObject);
7         SceneManager.sceneLoaded += OnSceneLoaded;
8         this.gameObject.SetActive (false);
9     }
10
11    private void OnSceneLoaded (Scene scene, LoadSceneMode mode)
12    {
13        if (scene.name == "Title")
14        {
15            SceneManager.sceneLoaded -= OnSceneLoaded;
16            Destroy(this.gameObject);
17        }
18        else
19        {
20            this.gameObject.SetActive(scene.name == "Battle");
21        }
22    }
23 }
```

Also, the PlayerParty will have two children to represent the player units. So, first let's create a PlayerUnit prefab with only a few things for now. Later on this tutorial, we are going to add the rest of its behavior.

For now, the PlayerUnit will have only the Sprite Renderer and a script called UnitStats, which will store the stats of each unit, such as: health, mana, attack, magic attack, defense and speed.

```
1 public class UnitStats : MonoBehaviour
2 {
3     public float health;
4     public float mana;
5     public float attack;
6     public float magic;
7     public float defense;
8     public float speed;
9 }
```

The figure below shows the example of one player unit, called MageUnit. In this example the UnitStats script has other attributes, which will be used later (such as Animator and Damage Text Prefab), but you can ignore those attributes for now.



By now, you can already try running your game with the Title scene. You can create an empty Town scene only to test the play button.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

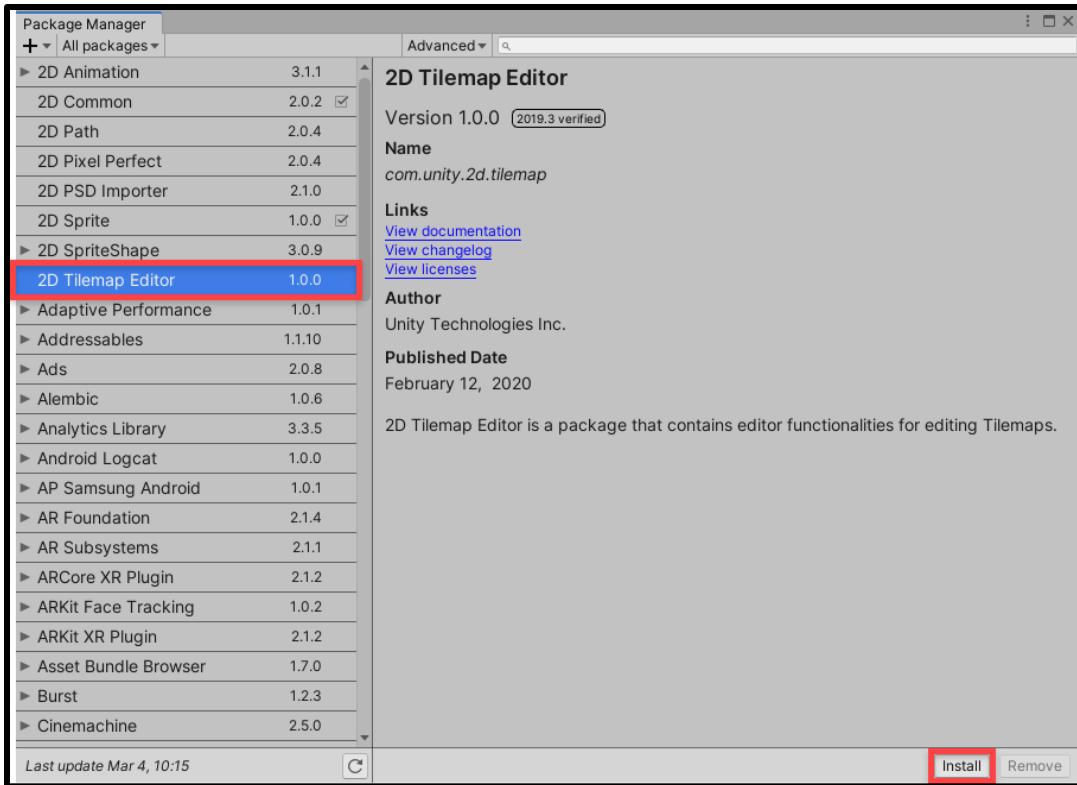


Town Scene

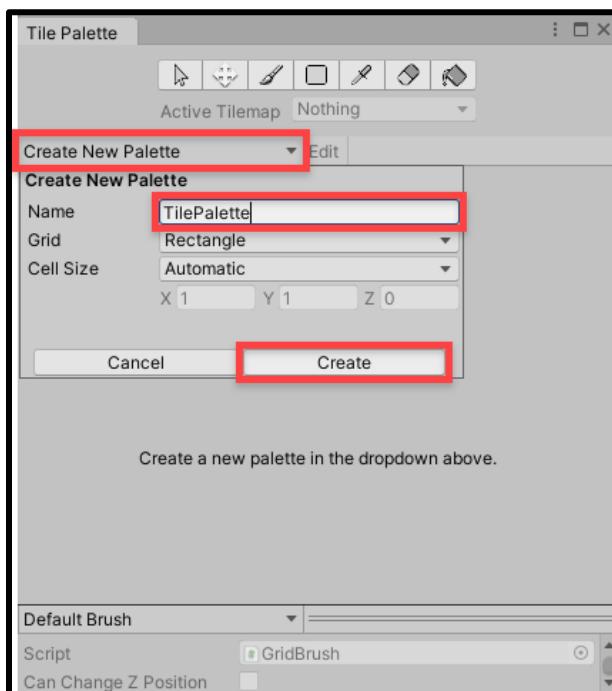
We will start by creating the Town Scene. So, create a new Scene in your project and call it Town.

Creating our Tilemap

The Town scene will have a tilemap for us to walk around. To begin, let's download the 2D Tilemap Editor package in the Package Manager (*Window > Package Manager*).



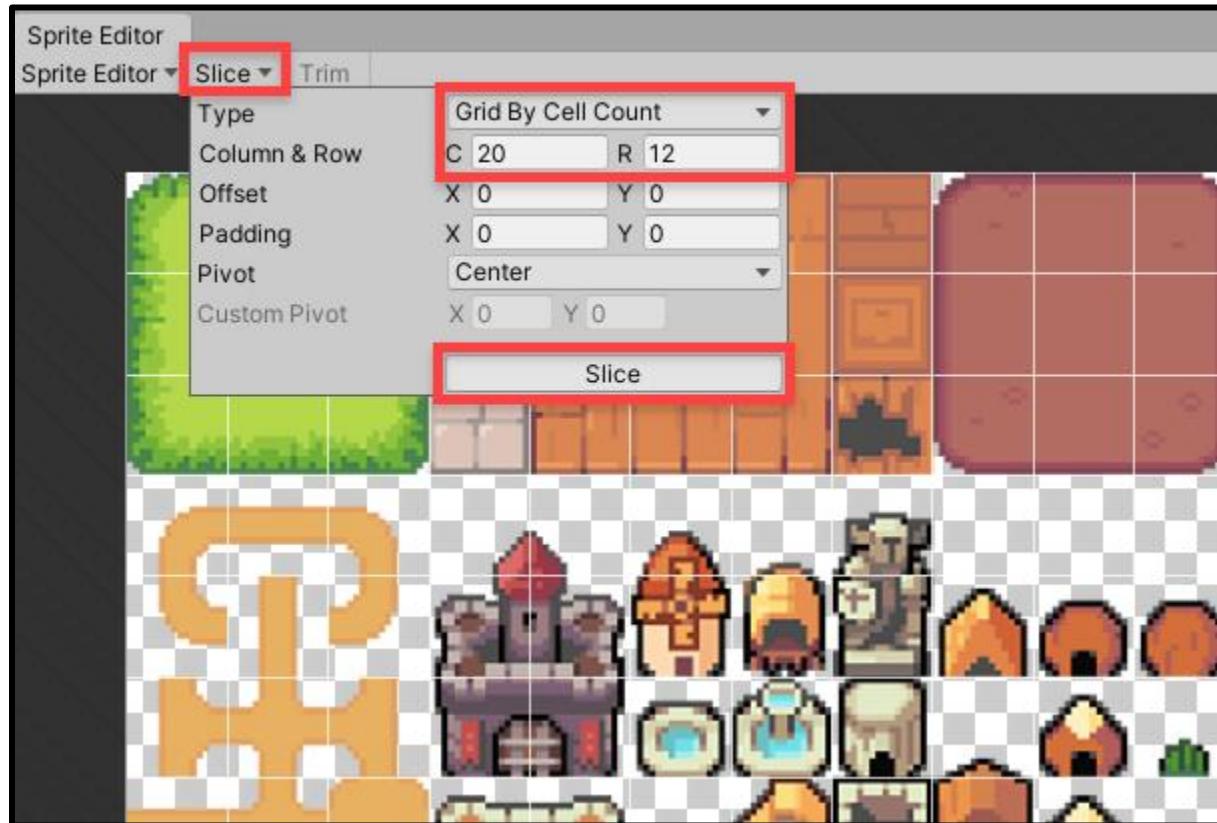
Then open the Tile Palette window (*Window > 2D > Tile Palette*). Here, create a new palette and save it in a new folder called Tilemap.



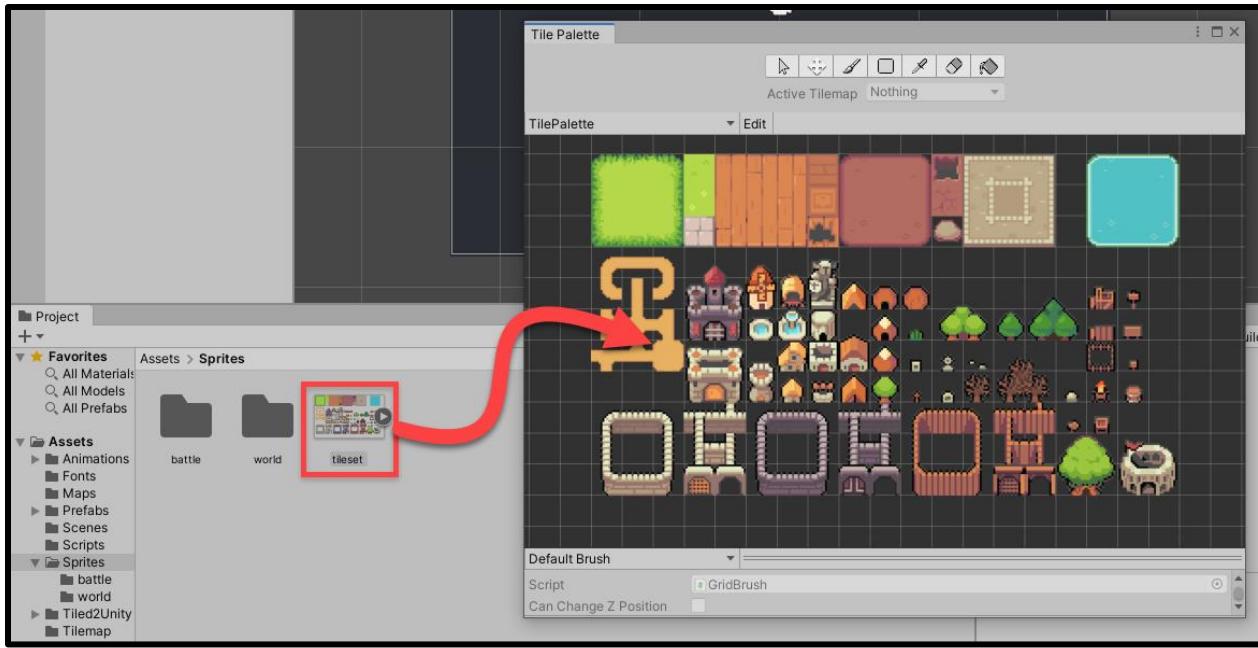
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Now we need to go to our tilemap image. Set the "Sprite Mode" to Multiple, the "Pixels Per Unit" to 64, then click "Sprite Editor".

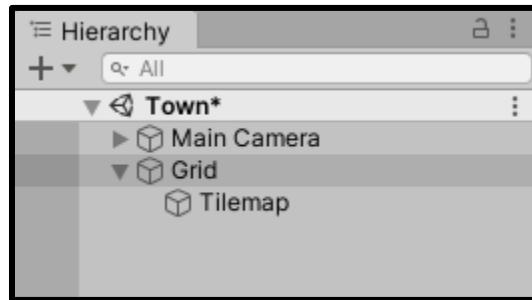
Here, we want to slice the image up into 20 columns and 12 rows.



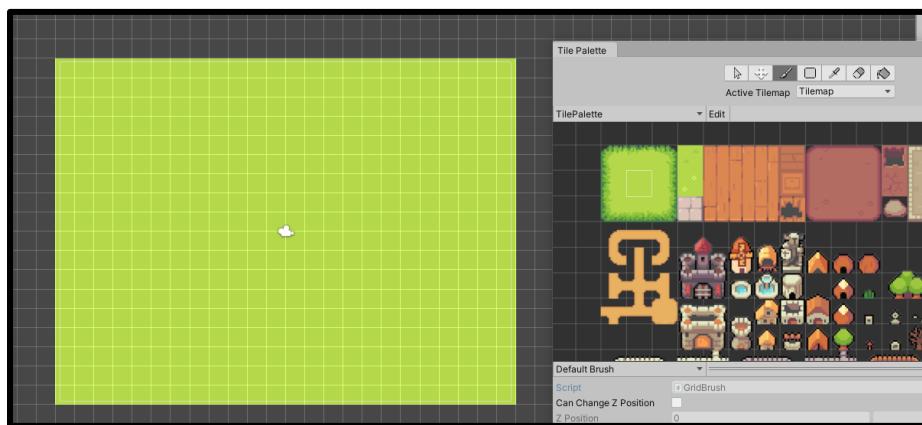
Then we can drag our tileset into the window to create the tiles.



In the Town scene, right click and create a *2D Object > Tilemap*.



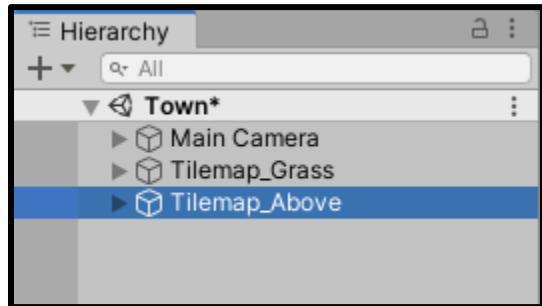
Now, we can select a tile and draw it on the tilemap. Let's create a grass background.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

In order to layer tiles, we can create a new tilemap for anything above the grass.



Set the grid child object's sorting layer to 1.

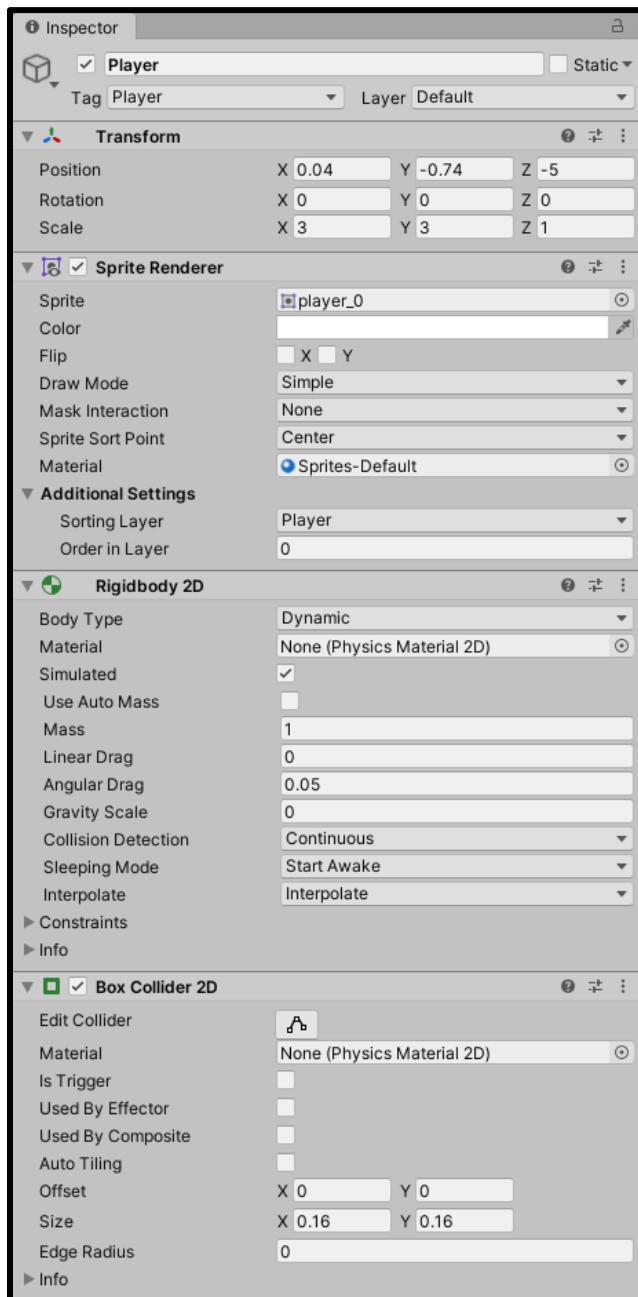
Create a scene, something like this.



Player prefab

Now that we added our town map to Unity, we are going to create a Player prefab. The player will be able to move around the town, and must collide with the collidable Tiles.

So, let's start by creating a GameObject called Player, adding the correct sprite renderer, a Box Collider 2D and a Rigidbody 2D as shown below. Observe that we need to set the Gravity Scale attribute of the Rigidbody2D as 0, so that the Player won't be affected by the gravity.

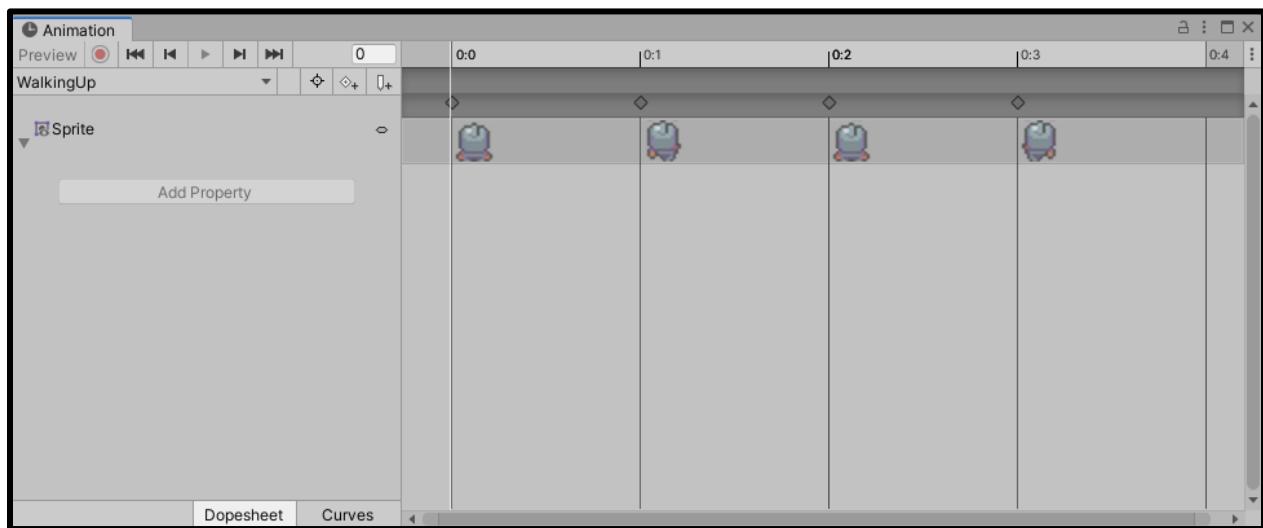


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

We also need to create the Player animations. The Player will have four walking animations and four idle animations, one for each direction. So, first, we create all the animations naming them IdleLeft, IdleRight, IdleUp, IdleDown, WalkingLeft, WalkingRight, WalkingUp and WalkingDown.

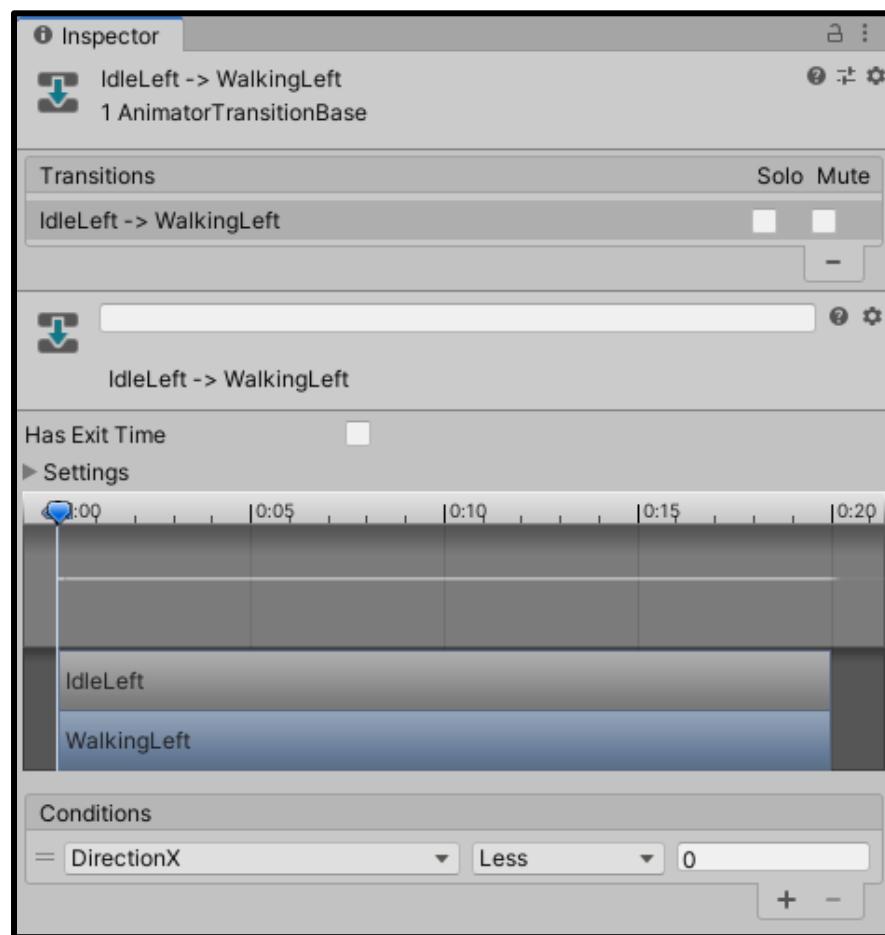
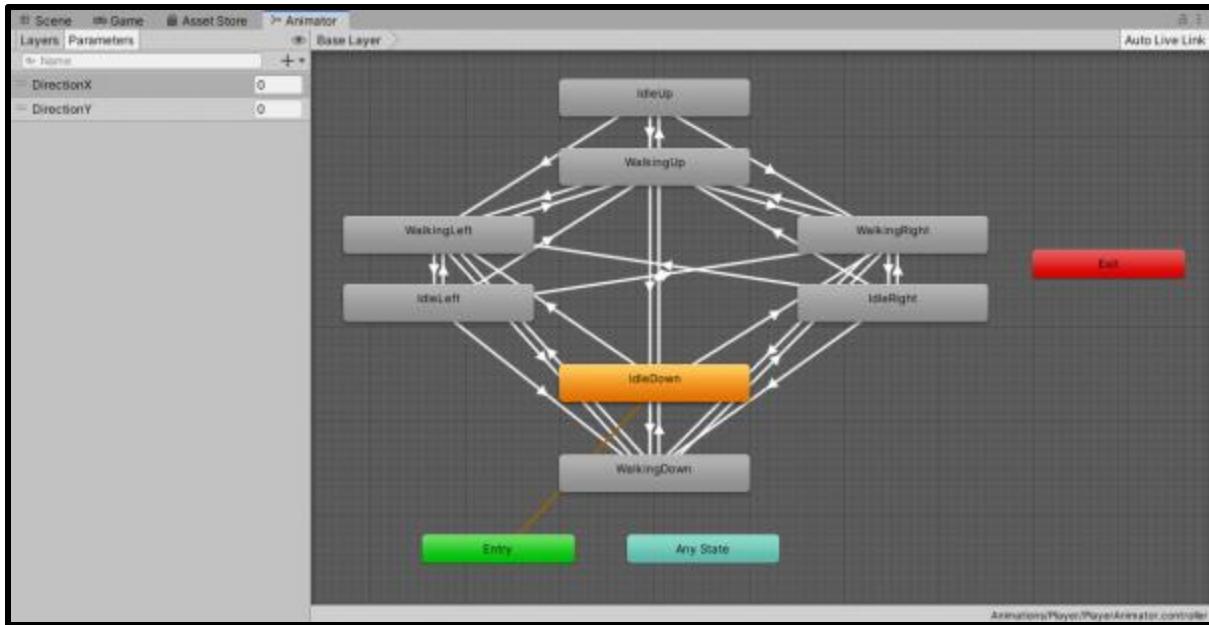
The next thing we need is the player animator. So, we create a new animator called PlayerAnimator and add all the created animations to it. Once we add the animator to the Player object in the inspector, we can create its animations using the Unity animation window and the player spritesheet. The figure below shows the example of the WalkingUp animation.



Now we need to configure the animation transitions in the player animator. The animator will have two parameters: DirectionX and DirectionY, which describes the current direction of the player movement. For example, if the player is moving to the left, DirectionX is equal to -1, while DirectionY is equal to 0. Those parameters will be correctly set later in a movement script.

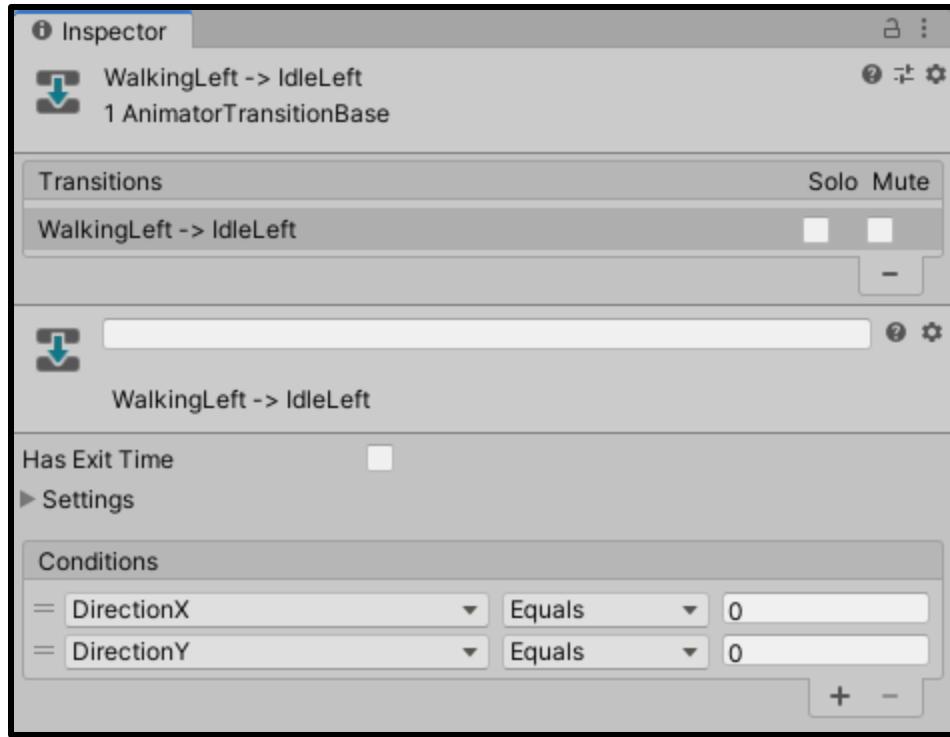
Each idle animation will have a transition to each walking animation. The direction parameters should have the values according to the animation direction. For example, the Idle Left animation will change to Walking Left if DirectionX is equal to -1. Also, each walking animation will have a transition for its correspondent idle animation. Finally, if the player changes its walking direction without stopping, we need to update the animation. So, we need to add transitions between the walking animations as well.

In the end, the player animator should look like the figure below. The next figures show examples of transitions between animations (IdleLeft -> WalkingLeft and WalkingLeft -> IdleLeft).



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved



Now let's create the PlayerMovement script. All movement happens in the `FixedUpdate` method. We use the horizontal and vertical axis inputs to check if the player should move horizontally or vertically. The player can move to a given direction if it is not already moving to the opposite direction. For example, it can move to the left if it is not already moving to the right. We do that for both vertical and horizontal directions. When moving to a given direction, we need to set the animator parameters. In the end, we apply the velocity to the Player Rigidbody2D.

```

1 public class PlayerMovement : MonoBehaviour
2 {
3     [SerializeField]
4     private float speed;
5
6     [SerializeField]
7     private Animator animator;
8
9     void FixedUpdate ()
10    {
11         float moveHorizontal = Input.GetAxis("Horizontal");
12         float moveVertical = Input.GetAxis("Vertical");
13
14         Vector2 currentVelocity = gameObject.GetComponent<Rigidbody2D>().velocity;
15
16         float newVelocityX = 0f;
17

```

```

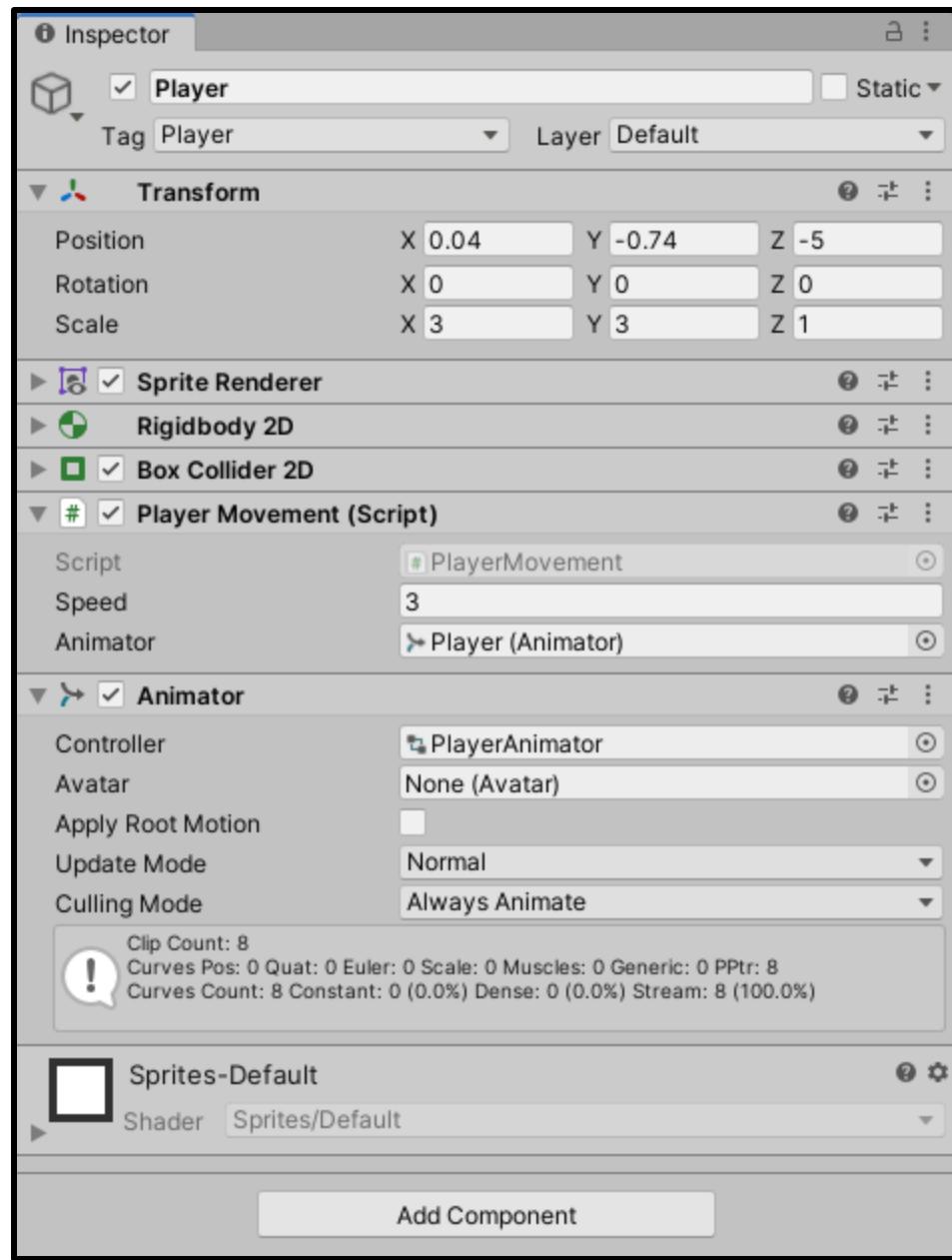
17     if(moveHorizontal < 0 && currentVelocity.x <= 0)
18     {
19         newVelocityX = -speed;
20         animator.SetInteger("DirectionX", -1);
21     }
22     else if(moveHorizontal > 0 && currentVelocity.x >= 0)
23     {
24         newVelocityX = speed;
25         animator.SetInteger("DirectionX", 1);
26     }
27     else
28     {
29         animator.SetInteger("DirectionX", 0);
30     }
31
32     float newVelocityY = 0f;
33
34     if(moveVertical < 0 && currentVelocity.y <= 0)
35     {
36         newVelocityY = -speed;
37         animator.SetInteger("DirectionY", -1);
38     }
39

```

```

40     else if(moveVertical > 0 && currentVelocity.y >= 0)
41     {
42         newVelocityY = speed;
43         animator.SetInteger("DirectionY", 1);
44     }
45     else
46     {
47         animator.SetInteger("DirectionY", 0);
48     }
49
50     gameObject.GetComponent<Rigidbody2D>().velocity = new Vector2(
51         newVelocityX, newVelocityY);
52 }
53 }
```

In the end, the Player prefab should look like the figure below.



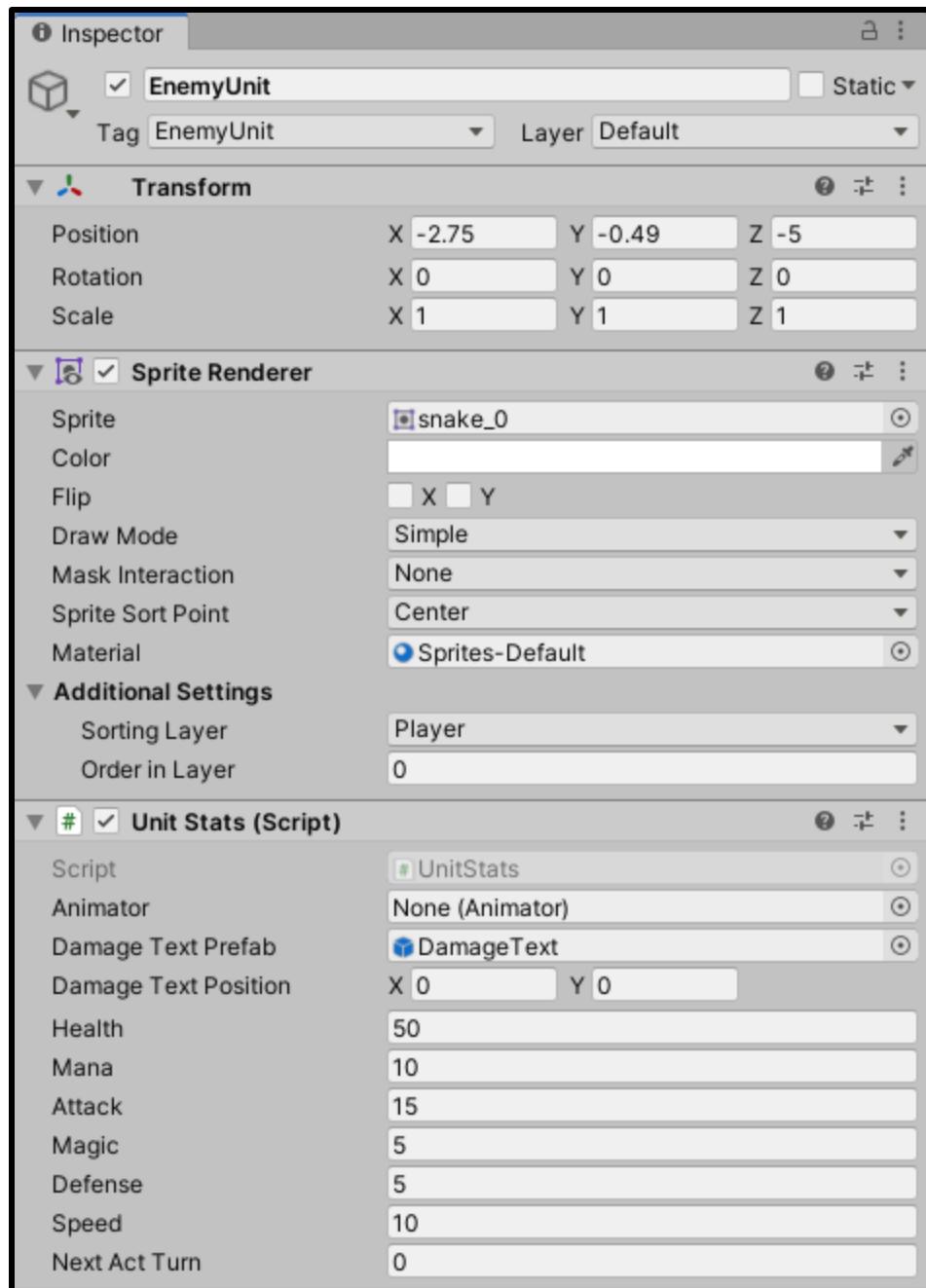
For now, you can start your game and try moving the player around the map. Remember to check if the tile collisions are properly working.



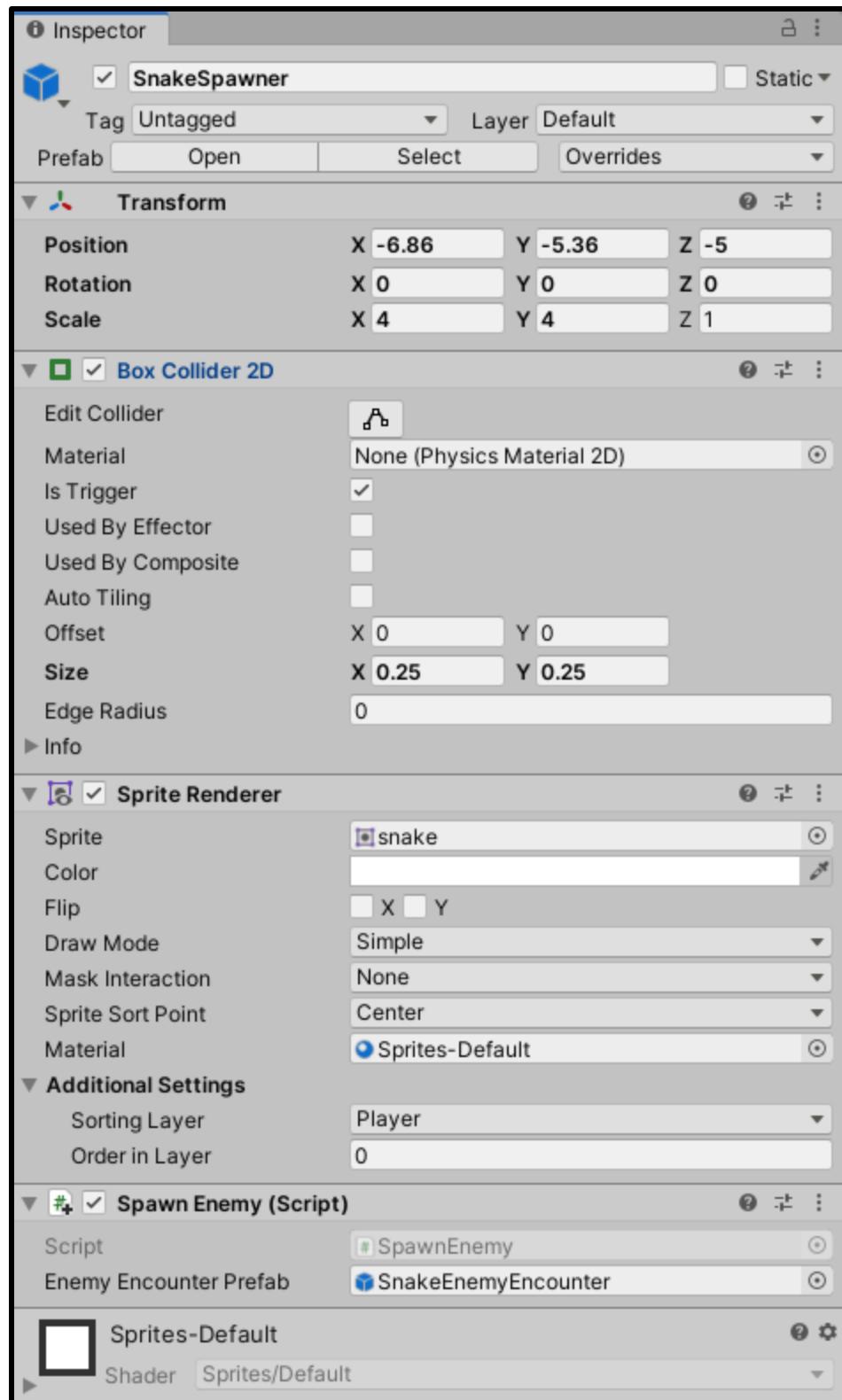
Starting Battle

The player can start battles by interacting with enemy spawners. The enemy spawner will be an immovable object that, when touched by the player will switch to another scene called Battle Scene.

Also, the enemy spawner will be responsible for creating the enemy units objects in the Battle Scene. This will be done by creating an `EnemyEncounter` prefab, with enemy units as children. Like the player units from the Title Scene, for now we are only going to add the `UnitStats` script and the `Sprite Renderer` to enemy units. The figure below shows an example of an enemy unit. You can create the `EnemyEncounter` by creating a new prefab and adding the desired enemy units as children to it. You will also need to properly set its position so that it will be created in the correct position in Battle Scene.



So, let's create a prefab called `EnemySpawner`. This prefab will have a collision box and a `Rigidbody2D`, in order to check for collisions with the `Player` prefab.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Also, it will have a script called SpawnEnemy as below. This script implements the OnCollisionEnter2D method to check for collisions with the Player prefab. We do so by checking if the other object tag is "Player" (remember to properly set the Player prefab tag). If there is a collision, it is going to start the Battle Scene and set a spawning attribute to true.

In order to create the enemy units in the Battle Scene, the script needs as an attribute the enemy encounter prefab, and the enemy spawner must not be destroyed when changing scenes (done in the Start method). When loading a scene (in the OnSceneLoaded method), if the scene being loaded is the Battle Scene, the enemy spawner will destroy itself and will instantiate the enemy encounter if the spawning attribute is true. This way, we can make sure that only one spawner will instantiate the enemy encounter, but all of them will be destroyed.

```
1 public class SpawnEnemy : MonoBehaviour
2 {
3     [SerializeField]
4     private GameObject enemyEncounterPrefab;
5
6     private bool spawning = false;
7
8     void Start ()
9     {
10        DontDestroyOnLoad(this.gameObject);
11        SceneManager.sceneLoaded += OnSceneLoaded;
12    }
13
14     private void OnSceneLoaded (Scene scene, LoadSceneMode mode)
15     {
16         if(scene.name == "Battle")
17         {
18             if(this.spawning)
19             {
20                 Instantiate(enemyEncounterPrefab);
21             }
22
23             SceneManager.sceneLoaded -= OnSceneLoaded;
24             Destroy(this.gameObject);
25         }
26     }
27
28     void OnTriggerEnter2D (Collider2D other)
29     {
30         if(other.gameObject.tag == "Player")
31         {
32             this.spawning = true;
33             SceneManager.LoadScene("Battle");
34         }
35     }
36 }
```

By now, you can try running your game and interacting with the enemy spawner. Try creating an empty Battle Scene to allow changing the scenes.



Battle Scene

Background and HUD Canvases

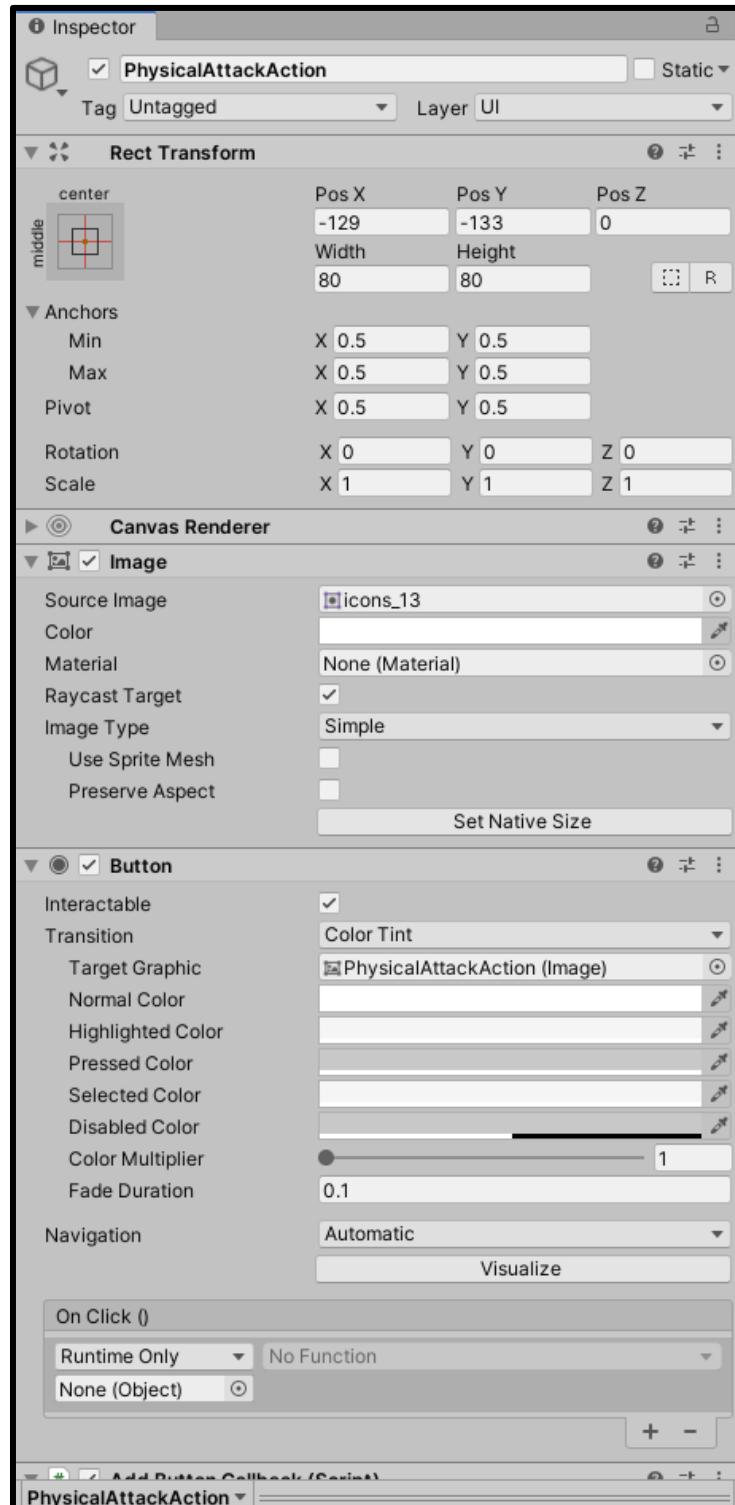
Let's start by creating the canvases that we are going to use for Battle Scene. Similarly to Title Scene, we are going to use one for the background and another for the HUD elements.

The background canvas will be the same as for the Title Scene, so I'm not going to show its creation. The HUD canvas, on the other hand, will need a lot of elements to allow proper player interaction.

First, we will add an actions menu, which will show the possible actions for the player. This will be an empty object used as parent of all action menu items. Each action menu item, by its turn, will be a button, added as child of the ActionsMenu.

We are going to add three possible actions: attacking with physical attack (PhysicalAttackAction), attacking with magical attack (MagicAttackAction) and running from the battle (RunAction). Each action will have its OnClick event but, for now, we are not going to add it. The figure below shows only the PhysicalAttackAction, since the other ones will only change

the source image for now. The source images for those menu items are from the icons Sprite, which was sliced in many icons.

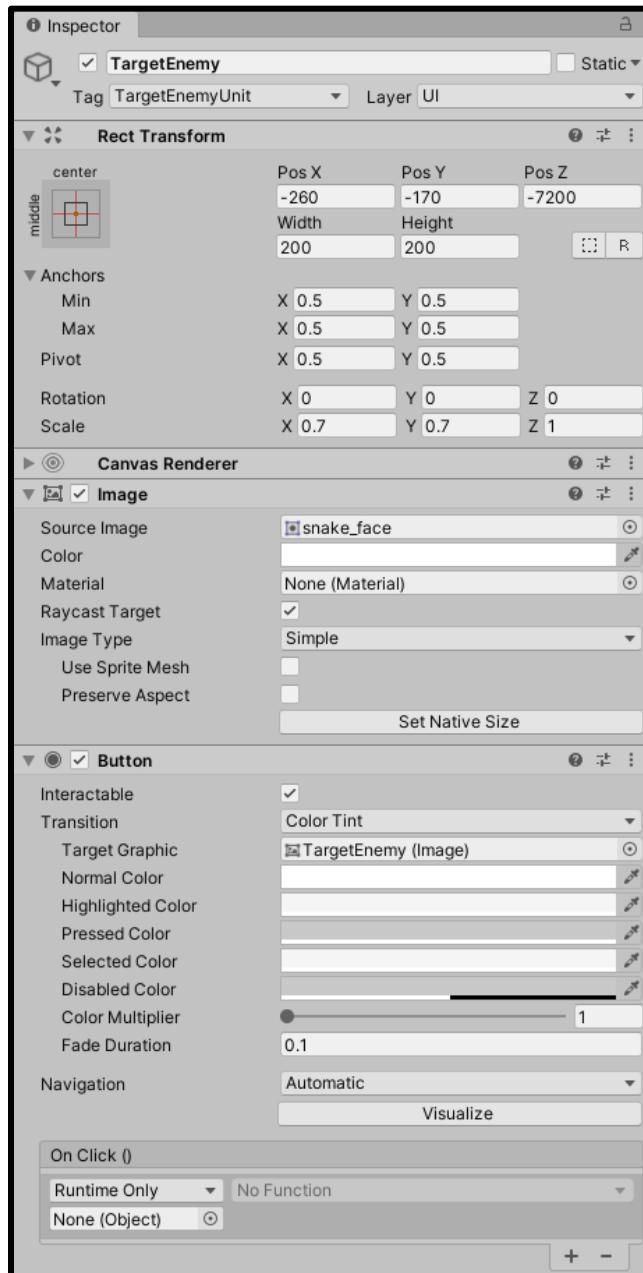


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

The second menu we are going to add to the HUD canvas is the EnemyUnitsMenu. This menu will be used to show the enemy units, so that the player can choose one to attack. Similarly to the ActionsMenu, it will be an empty object used to group its menu items. However, the enemy menu items will be created by the enemy units, when the Battle scene starts.

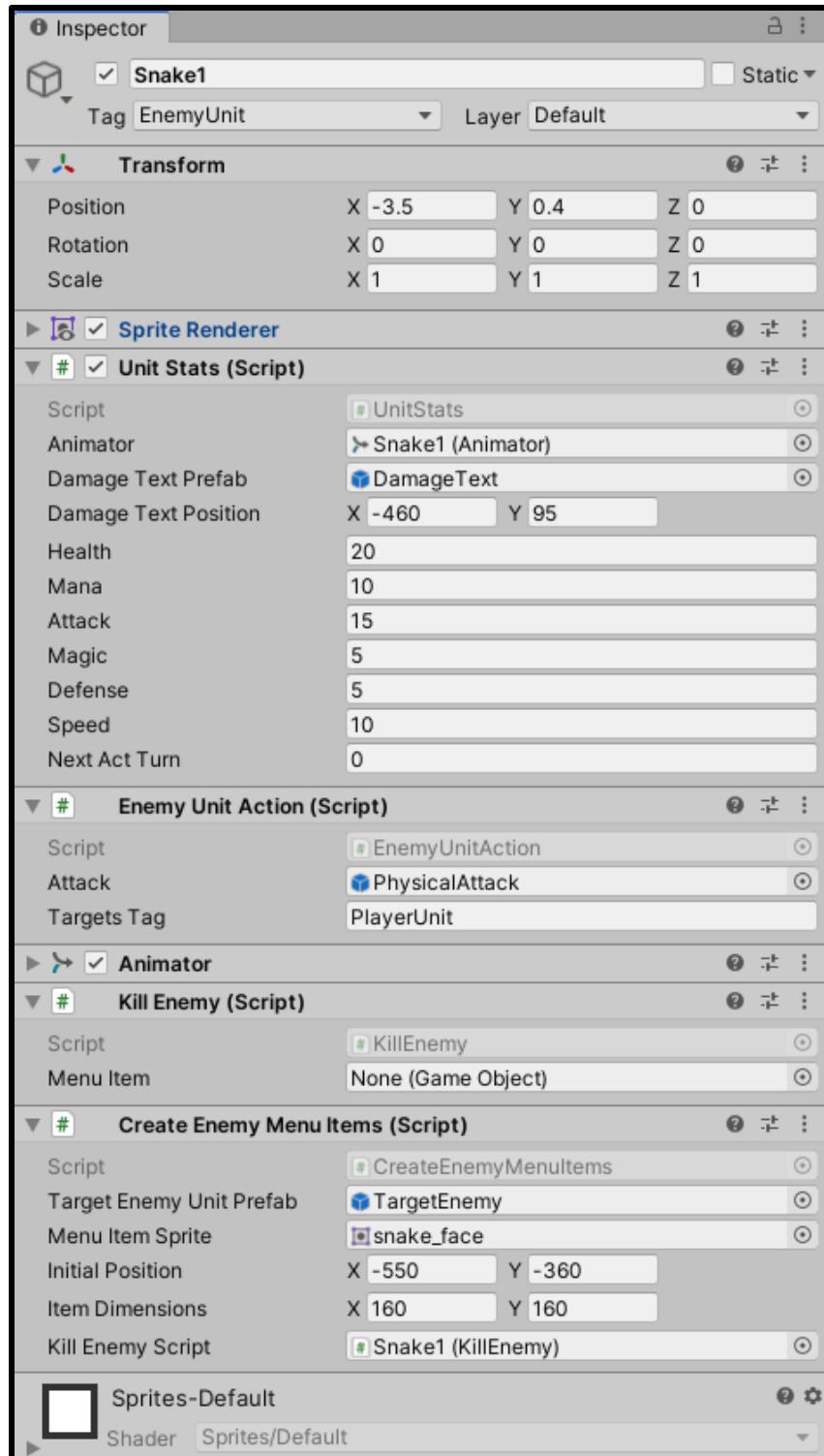
In order to make the enemy unit to create its menu item, we need to create the menu item prefab. This prefab will be called TargetEnemy and will be a button. The OnClick callback of this button will be used to select this enemy as the target.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

We need to add two scripts to the EnemyUnit prefab to handle its menu item: KillEnemy and CreateEnemyMenuItem.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

The KillEnemy script is simple. It will have as an attribute the menu item of this unit, and when the unit is destroyed (OnDestroy method), the menu item must be destroyed too.

```
1 public class KillEnemy : MonoBehaviour
2 {
3     public GameObject menuItem;
4
5     void OnDestroy ()
6     {
7         Destroy(this.menuItem);
8     }
9 }
```

Now let's create the CreateEnemyMenuItem script. This script will be responsible for creating its menu item and setting its OnClick callback. All this is done in the Awake method. First, the menu item position is calculated based on the number of existing items. Then, it is instantiated as children of EnemyUnitsMenu, and the script sets its localPosition and localScale. In the end, it sets the OnClick callback to be the selectEnemyTarget method, and sets the menu item as the one for this unit in KillEnemy.

The selectEnemyTarget method should make the player to attack this unit. However, we don't have the code to do that now. So, for now we are going to leave this method empty.

```
1 public class CreateEnemyMenuItems : MonoBehaviour
2 {
3     [SerializeField]
4     private GameObject targetEnemyUnitPrefab;
5
6     [SerializeField]
7     private Sprite menuItemSprite;
8
9     [SerializeField]
10    private Vector2 initialPosition, itemDimensions;
11
12    [SerializeField]
13    private KillEnemy killEnemyScript;
14 }
```

```

15 // Use this for initialization
16 void Awake ()
17 {
18     GameObject enemyUnitsMenu = GameObject.Find("EnemyUnitsMenu");
19
20     GameObject[] existingItems = GameObject.FindGameObjectsWithTag("TargetEnemyUnit");
21     Vector2 nextPosition = new Vector2(this.initialPosition.x + (
22         existingItems.Length * this.itemDimensions.x), this.initialPosition.y);
23
24     GameObject targetEnemyUnit = Instantiate(this.targetEnemyUnitPrefab,
25         enemyUnitsMenu.transform) as GameObject;
26     targetEnemyUnit.name = "Target" + this.gameObject.name;
27     targetEnemyUnit.transform.localPosition = nextPosition;
28     targetEnemyUnit.transform.localScale = new Vector2(0.7f, 0.7f);
29     targetEnemyUnit.GetComponent<Button>().onClick.AddListener(() => selectEnemyTarget());
30     targetEnemyUnit.GetComponent<Image>().sprite = this.menuItemSprite;
31
32     killEnemyScript.menuItem = targetEnemyUnit;
33 }
34
35 public void selectEnemyTarget ()
36 {
37 }
38 }
39 }
```

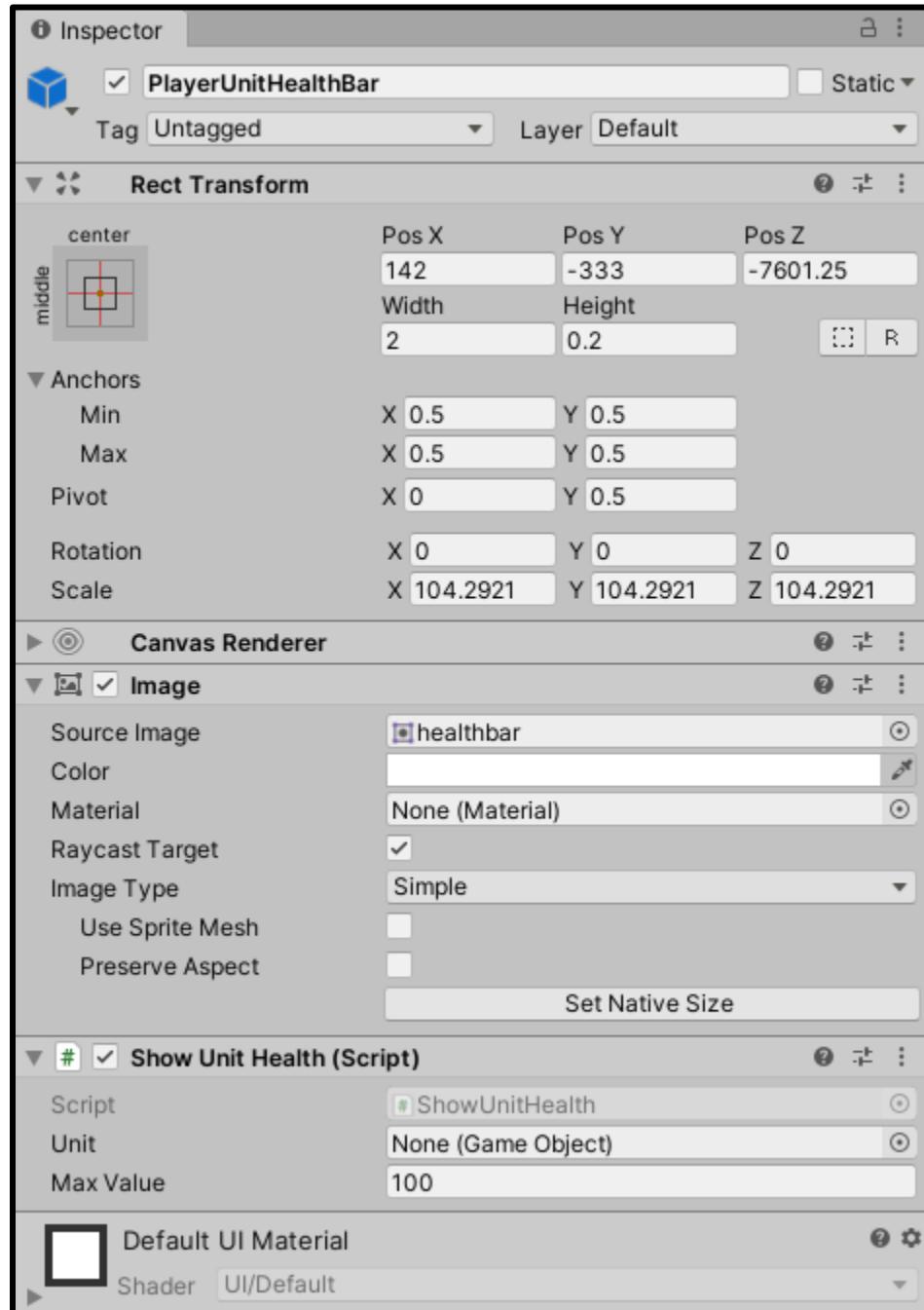
The final HUD elements we are going to add are those to show the player unit information, such as health and mana. So, we are going to start by creating an empty GameObject called PlayerUnitInformation, to hold all those HUD elements.

Then, we are going to add an image as child of this object called PlayerUnitFace. This element will simply show the face of the current unit. For now, let's select any unit face.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

The next elements will be the health bar and its text. The health bar will be an image showing the health bar sprite, while the text will show the HP message. Finally we do the same for the mana bar, only changing the sprite and the text message. The figures below show only the health bar, since the mana bar is very similar.

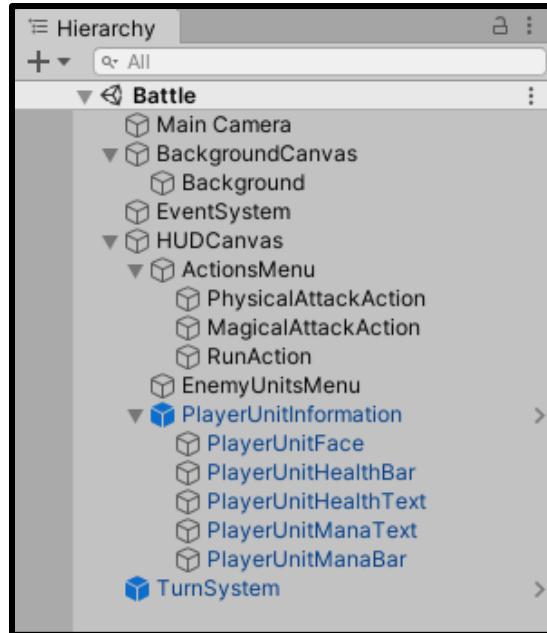
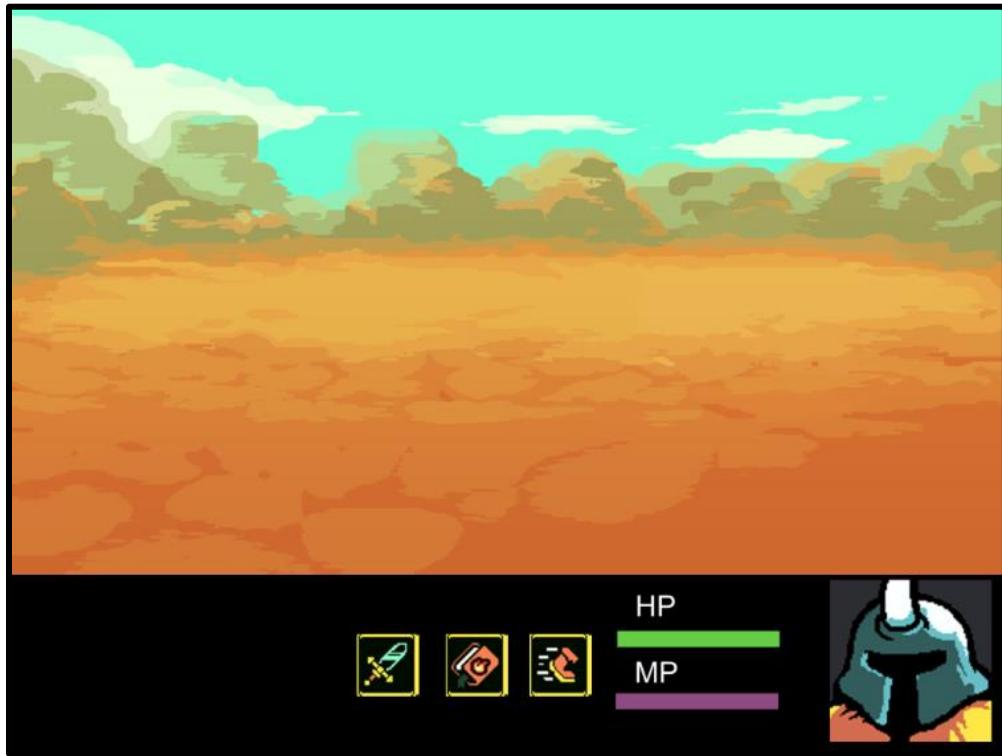




This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

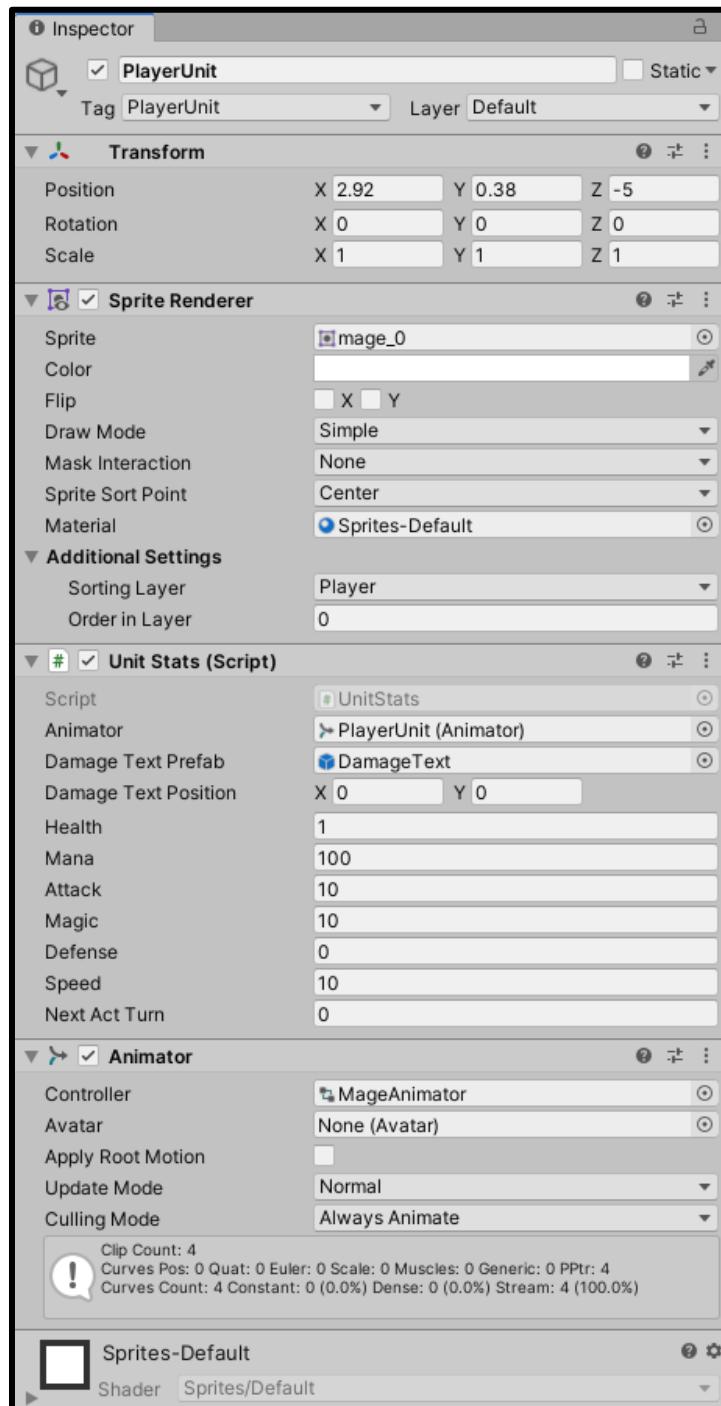
© Zenva Pty Ltd 2021. All rights reserved

By now, your Battle Scene should look like this. This figure corresponds to the Scene viewer, and not the running game, since there is a lot of content we still need to add to properly run the Battle Scene. The righthand figure shows the objects hierarchy in the scene.



Units Animations

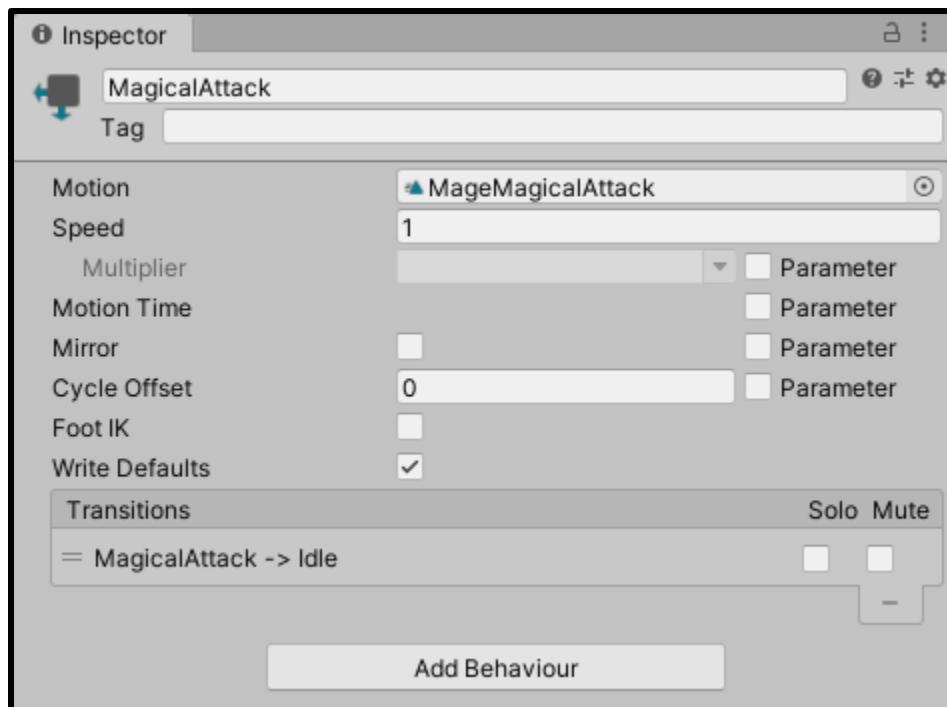
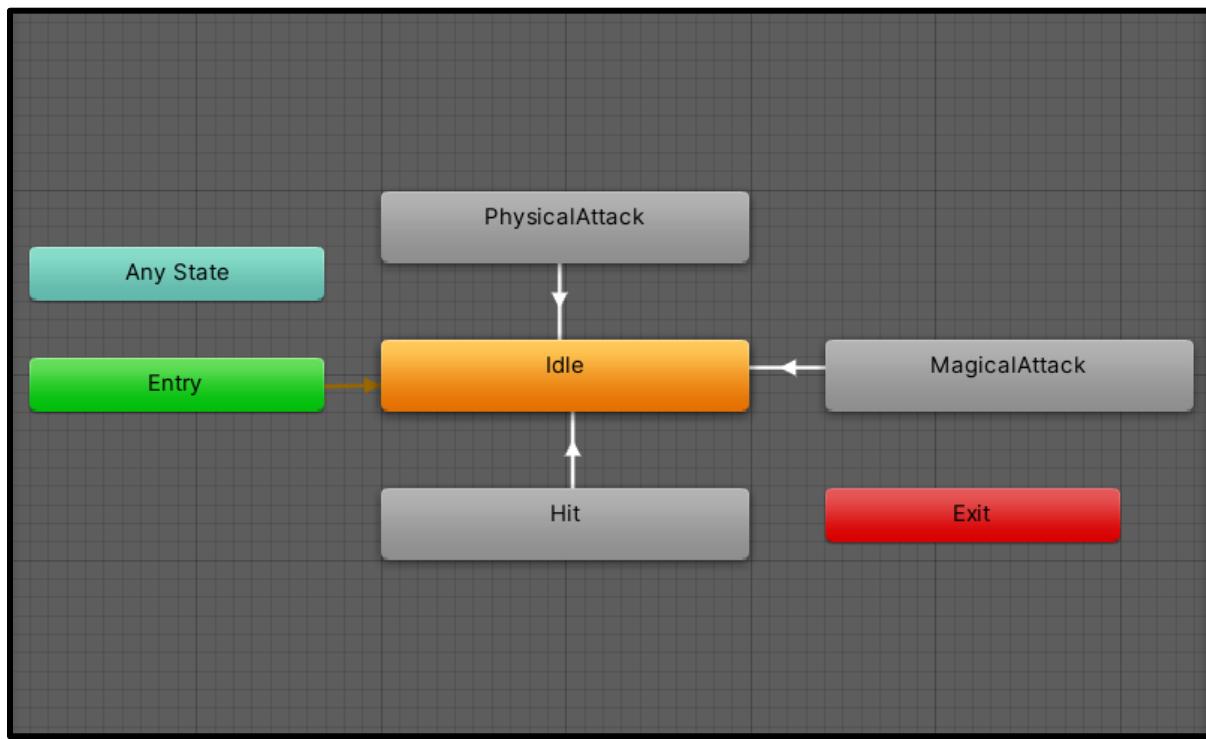
The next thing we are going to do is creating the units animations. Each unit will have four animatios: Idle, PhysicalAttack, MagicalAttack and Hit. So, let's start by creating an animator for one of the player units (for example, the MageUnit), and adding it to its correspondent prefab.



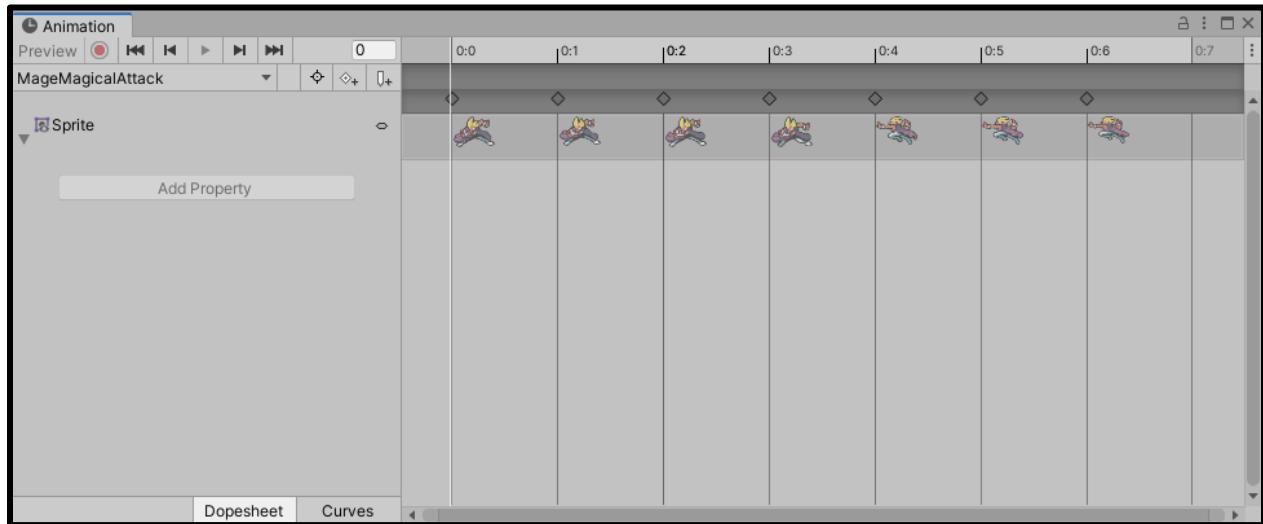
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Now, if we select this prefab and open the Animator view, we can configure its animations state machine as below. We are going to create a state for each animation, with Idle being the default one, and all other ones having transitions to Idle when they end.



Now, we need to create the four animations to add them to their correspondent states. The figure below shows the `MageMagicalAttack` animation for `MageUnit`. You can create all animations following the same process with the animation view, so I'm not going to show them all. Also, you have to do the same for all units (including the enemy ones).



We still need to define when to play those animations. However, we are going to do so when adding more functionalities for the units. For now, the units will only play the Idle animation, as it is the default one.

If you play the game now, it should show the units with the Idle animation. However, remember you have to play Title Scene and go until the Battle Scene, in order to see the units.

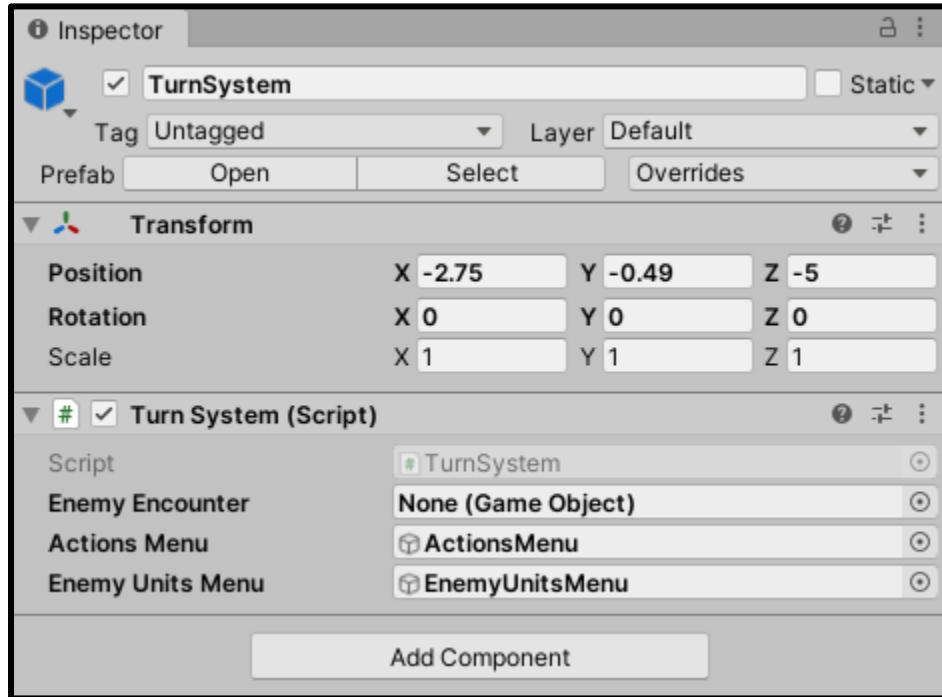


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Turn-Based Battle System

The next thing we are going to add to our game is a turn-based battle system. So, let's start by creating an empty game object with a script called TurnSystem.



The TurnSystem script will keep a List with the UnitStats script of all units (player and enemy ones). Then, in each turn, it can pop the first element of the list, make the unit act and add it again to the list. Also, it needs to keep the list sorted according to the units acting turns.

This script is shown below. The UnitStats list is created in the Start method. This is done by iterating through all game objects with the tags "PlayerUnit" or "EnemyUnit" (remember to properly tag your objects). For each unit, the script gets its UnitStats script, calculate its next acting turn and add it to the list. After adding all units the list is sorted. Finally, the menus are disabled, since they will be used only on the player turns, and the first turn begins (by calling nextTurn).

The nextTurn method, by its turn, will start by removing the first UnitStats from the list and checking if it is not dead. If the unit is alive, it will calculate its next acting turn in order to add it again to the list. Finally, it will make it act. We still don't have the acting methods of the units, so we are only going to print a message in the console for now. On the other hand, if the unit is dead, we are simply going to call nextTurn without adding it to the list again.

```

1 public class TurnSystem : MonoBehaviour
2 {
3     private List<UnitStats> unitsStats;
4
5     [SerializeField]
6     private GameObject actionsMenu, enemyUnitsMenu;
7
8     void Start ()
9     {
10         unitsStats = new List<UnitStats>();
11         GameObject[] playerUnits = GameObject.FindGameObjectsWithTag("PlayerUnit");
12
13         foreach(GameObject playerUnit in playerUnits)
14         {
15             UnitStats currentUnitStats = playerUnit.GetComponent<UnitStats>();
16             currentUnitStats.calculateNextActTurn(0);
17             unitsStats.Add (currentUnitStats);
18         }
19
20         GameObject[] enemyUnits = GameObject.FindGameObjectsWithTag("EnemyUnit");
21
22         foreach(GameObject enemyUnit in enemyUnits)
23         {
24             UnitStats currentUnitStats = enemyUnit.GetComponent<UnitStats>();
25             currentUnitStats.calculateNextActTurn(0);
26             unitsStats.Add(currentUnitStats);
27         }
28

```

```

29         unitsStats.Sort();
30
31         this.actionsMenu.SetActive(false);
32         this.enemyUnitsMenu.SetActive(false);
33
34         this.nextTurn();
35     }
36
37     public void nextTurn ()
38     {
39         UnitStats currentUnitStats = unitsStats [0];
40         unitsStats.Remove(currentUnitStats);
41
42         if(!currentUnitStats.isDead())
43         {
44             GameObject currentUnit = currentUnitStats.gameObject;
45
46             currentUnitStats.calculateNextActTurn(currentUnitStats.nextActTurn);
47             unitsStats.Add(currentUnitStats);
48             unitsStats.Sort();
49

```

```

50         if(currentUnit.tag == "PlayerUnit")
51         {
52             Debug.Log("Player unit acting");
53         }
54     else
55     {
56         Debug.Log("Enemy unit acting");
57     }
58 }
59 else
60 {
61     this.nextTurn();
62 }
63 }
64 }
```

Before moving on, we need to implement the UnitStats methods we used in TurnSystem, so let's go back to the UnitStats script.

First, the calculateNextActTurn method is responsible for calculating the next acting turn based on the current one. This is done based on the unit speed, as shown below. Also, we need to make UnitStats to extend the IComparable interface, and implement the CompareTo method, so that we can properly sort the UnitStats list. The CompareTo method will simply compare the acting turn of the two scripts. Finally, we need to implement the isDead getter, which will simply return the dead attribute value. By default, this attribute is false, because the unit is alive at the beginning of the game.

```

1 public class UnitStats : MonoBehaviour, IComparable
2 {
3     public float health;
4     public float mana;
5     public float attack;
6     public float magic;
7     public float defense;
8     public float speed;
9
10    public int nextActTurn;
11
12    private bool dead = false;
13 }
```

```

14     public void calculateNextActTurn (int currentTurn)
15     {
16         this.nextActTurn = currentTurn + (int)Math.Ceiling(100.0f / this.speed);
17     }
18
19     public int CompareTo (object otherStats)
20     {
21         return nextActTurn.CompareTo(((UnitStats)otherStats).nextActTurn);
22     }
23
24     public bool isDead ()
25     {
26         return this.dead;
27     }
28 }
```

For now, you can try playing the game again, to see if the turn message is being properly printed in the console.

Attacking Units

Now that we have our turn-based battle system we are going to allow units to attack each other. First, we are going to create Attack prefabs, which will be used by the units. Then, we are going to add the action scripts of both player and enemy units, so that they can properly attack. When receiving damage, units will show a Text prefab with the damage value.

The Attack prefab will be an invisible prefab with a script called AttackTarget. This script will describe the attack properties such as attack and defense multipliers and mana cost. Also, the attack will have an owner, which is the unit currently attacking.

First, the script checks if the owner has enough mana to execute the attack. If so, it picks random attack and defense multipliers based on the minimum and maximum values. So, the damage is calculated based on those multipliers and the attack and defense of the units. Observe that, if the attack is a magical attack (this.magicAttack is true), then it will use the magic stat of the unit, otherwise it uses the attack stat.

In the end, the script plays the attack animation, inflicts the damage to the target unit and reduces the mana of the owner accordingly.

```
1 public class AttackTarget : MonoBehaviour
2 {
3     public GameObject owner;
4
5     [SerializeField]
6     private string attackAnimation;
7
8     [SerializeField]
9     private bool magicAttack;
10
11    [SerializeField]
12    private float manaCost;
13
14    [SerializeField]
15    private float minAttackMultiplier;
16
17    [SerializeField]
18    private float maxAttackMultiplier;
19
20    [SerializeField]
21    private float minDefenseMultiplier;
22
23    [SerializeField]
24    private float maxDefenseMultiplier;
25
```

```

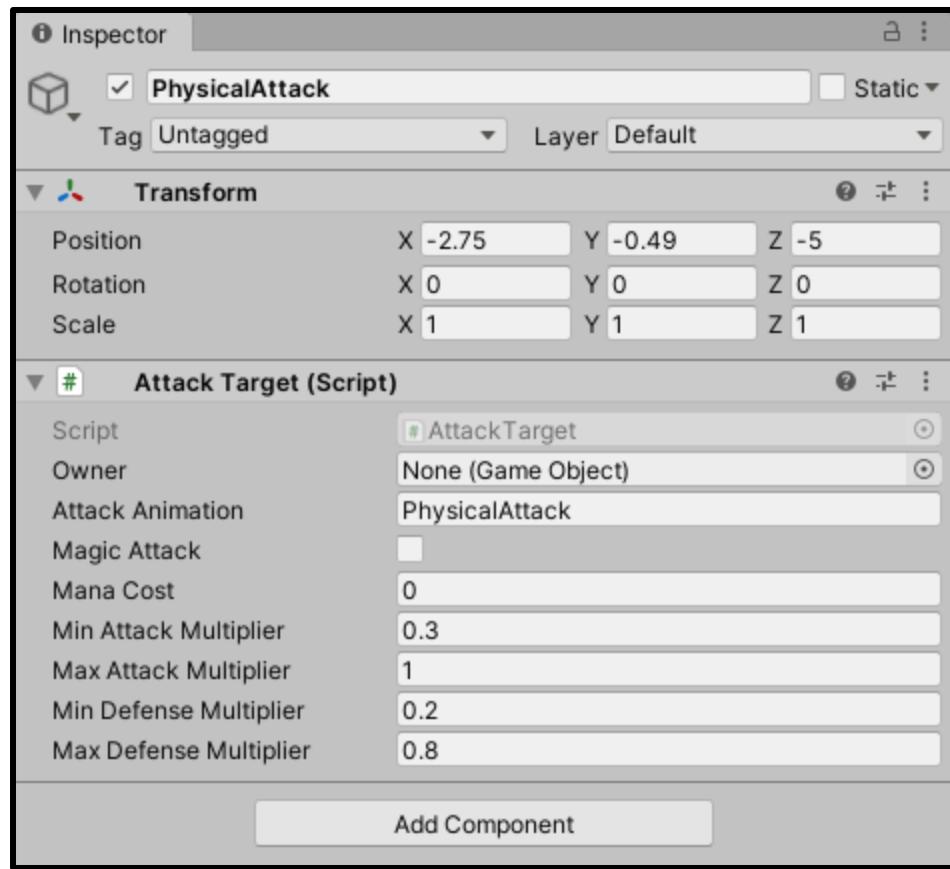
26 public void hit (GameObject target)
27 {
28     UnitStats ownerStats = this.owner.GetComponent<UnitStats>();
29     UnitStats targetStats = target.GetComponent<UnitStats>();
30
31     if(ownerStats.mana >= this.manaCost)
32     {
33         float attackMultiplier = (Random.value * (
34             this.maxAttackMultiplier - this.minAttackMultiplier))
35             + this.minAttackMultiplier;
36         float damage = (this.magicAttack) ? attackMultiplier *
37             ownerStats.magic : attackMultiplier * ownerStats.attack;
38
39         float defenseMultiplier = (Random.value * (
40             this.maxDefenseMultiplier - this.minDefenseMultiplier))
41             + this.minDefenseMultiplier;
42         damage = Mathf.Max(0, damage - (defenseMultiplier * targetStats.defense));
43
44         this.owner.GetComponent<Animator>().Play(this.attackAnimation);
45
46         targetStats.receiveDamage(damage);
47
48         ownerStats.mana -= this.manaCost;
49     }
50 }
51 }
```

We are going to create two attack prefabs: PhysicalAttack and MagicalAttack, each one with its own multipliers.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved



Now we need to implement the `receiveDamage` method, used in the `AttackTarget` script. This will be a method from `UnitStats` that, besides reducing the unit health, it will also show the damage using a Text over the unit's head.

This method is shown below. First, it will simply reduce the unit health and play the `Hit` animation. Then, it will create the damage text (using `this.damageTextPrefab`). Observe that the damage text must be a child of the `HUDCanvas`, since it is an UI element, and we need to properly set its `localPosition` and `localScale`. In the end, if the unit health is less than zero, the script set the unit as dead, change its tag and destroy it.

```

1 public void receiveDamage (float damage)
2 {
3     this.health -= damage;
4     animator.Play("Hit");
5
6     GameObject HUDCanvas = GameObject.Find("HUDCanvas");
7     GameObject damageText = Instantiate(this.damageTextPrefab,
8         HUDCanvas.transform) as GameObject;
9     damageText.GetComponent<Text>().text = "" + damage;
10    damageText.transform.localPosition = this.damageTextPosition;
11    damageText.transform.localScale = new Vector2(1.0f, 1.0f);
12
13    if(this.health <= 0)
14    {
15        this.dead = true;
16        this.gameObject.tag = "DeadUnit";
17        Destroy(this.gameObject);
18    }
19 }
```

Now we can already implement the act method of the units. An enemy unit will always attack a random enemy with the same attack. This attack will be an attribute in EnemyUnitAction. In the Awake method we are going to create a copy of it for the unit and properly set its owner. We need to create a copy since we want each unit to have its own attack object instance.

Then, the act method will pick a random target and attack it. The findRandomTarget method, by its turn, will start by listing all possible targets given their tags (for example, "PlayerUnit"). If there is at least one possible target in this list, it will generate a random index to pick the target.

```

1 public class EnemyUnitAction : MonoBehaviour
2 {
3     [SerializeField]
4     private GameObject attack;
5
6     [SerializeField]
7     private string targetsTag;
8
9     void Awake ()
10    {
11        this.attack = Instantiate(this.attack);
12        this.attack.GetComponent<AttackTarget>().owner = this.gameObject;
13    }
14 }
```

```

15     GameObject findRandomTarget ()
16     {
17         GameObject[] possibleTargets = GameObject.FindGameObjectsWithTag(targetsTag);
18
19         if(possibleTargets.Length > 0)
20         {
21             int targetIndex = Random.Range(0, possibleTargets.Length);
22             GameObject target = possibleTargets [targetIndex];
23
24             return target;
25         }
26
27         return null;
28     }
29
30     public void act ()
31     {
32         GameObject target = findRandomTarget();
33         this.attack.GetComponent<AttackTarget>().hit(target);
34     }
35 }
```

Player units, by their turn, will have two different attacks: physical and magical. So, in the Awake method we need to properly instantiate and set the owner of these two attacks. Also, we are going to set the current attack as the physical one by default.

Then, the act method will receive as parameter the target unit and will simply attack it.

```

1  public class PlayerUnitAction : MonoBehaviour
2  {
3      [SerializeField]
4      private GameObject physicalAttack;
5
6      [SerializeField]
7      private GameObject magicalAttack;
8
9      private GameObject currentAttack;
10
11     void Awake ()
12     {
13         this.physicalAttack = Instantiate(this.physicalAttack, this.transform) as GameObject;
14         this.magicalAttack = Instantiate(this.magicalAttack, this.transform) as GameObject;
15
16         this.physicalAttack.GetComponent<AttackTarget>().owner = this.gameObject;
17         this.magicalAttack.GetComponent<AttackTarget>().owner = this.gameObject;
18
19         this.currentAttack = this.physicalAttack;
20     }
21
22     public void act (GameObject target)
23     {
24         this.currentAttack.GetComponent<AttackTarget>().hit(target);
25     }
26 }
```

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Now, we can already call the enemy unit act method in the TurnSystem script. We still can't do the same for player units, since we still need to properly select the current unit and its attack. This is the next thing we are going to do.

```
1 public void nextTurn ()
2 {
3     UnitStats currentUnitStats = unitsStats [0];
4     unitsStats.Remove(currentUnitStats);
5
6     if(!currentUnitStats.isDead())
7     {
8         GameObject currentUnit = currentUnitStats.gameObject;
9
10        currentUnitStats.calculateNextActTurn(currentUnitStats.nextActTurn);
11        unitsStats.Add(currentUnitStats);
12        unitsStats.Sort();
13
14        if(currentUnit.tag == "PlayerUnit")
15        {
16            Debug.Log("Player unit acting");
17        }
18        else
19        {
20            currentUnit.GetComponent<EnemyUnitAction>().act();
21        }
22    }
23    else
24    {
25        this.nextTurn();
26    }
27 }
```

Selecting Unit and Action

We need to properly select the current player unit each turn. This will be done by adding the following script (SelectUnit) to the PlayerParty object. This script will need references to the battle menus, so when the Battle scene is loaded it is going to set them.

Then, we need to implement three methods: selectCurrentUnit, selectAttack and attackEnemyTarget. The first one will set a unit as the current one, enable the actions menu, so that the player can choose an action, and update the HUD to show the current unit face, health and mana (this last method will be implemented later).

The selectAttack method, by its turn, will call selectAttack for the current unit, and will change the current menu, by disabling the actions menu and enabling the enemies menu. The selectAttack method also needs to be implemented in the PlayerUnitAction script. This way, now that the player has selected an attack, it can select the target.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Finally, the attackEnemyTarget will disable both menus and call the act method for the current unit, with the selected enemy as the target.

```
1 public class SelectUnit : MonoBehaviour
2 {
3     private GameObject currentUnit;
4
5     private GameObject actionsMenu, enemyUnitsMenu;
6
7     void Awake ()
8     {
9         SceneManager.sceneLoaded += OnSceneLoaded;
10    }
11
12    private void OnSceneLoaded (Scene scene, LoadSceneMode mode)
13    {
14        if(scene.name == "Battle")
15        {
16            this.actionsMenu = GameObject.Find("ActionsMenu");
17            this.enemyUnitsMenu = GameObject.Find("EnemyUnitsMenu");
18        }
19    }
20
21    public void selectCurrentUnit (GameObject unit)
22    {
23        this.currentUnit = unit;
24        this.actionsMenu.SetActive(true);
25    }
26
```

```
27    public void selectAttack (bool physical)
28    {
29        this.currentUnit.GetComponent<PlayerUnitAction>().selectAttack(physical);
30
31        this.actionsMenu.SetActive(false);
32        this.enemyUnitsMenu.SetActive(true);
33    }
34
35    public void attackEnemyTarget (GameObject target)
36    {
37        this.actionsMenu.SetActive(false);
38        this.enemyUnitsMenu.SetActive(false);
39
40        this.currentUnit.GetComponent<PlayerUnitAction>().act(target);
41    }
42 }
```

```

1 public class PlayerUnitAction : MonoBehaviour
2 {
3     [SerializeField]
4     private GameObject physicalAttack;
5
6     [SerializeField]
7     private GameObject magicalAttack;
8
9     private GameObject currentAttack;
10
11    public void selectAttack (bool physical)
12    {
13        this.currentAttack = (physical) ? this.physicalAttack : this.magicalAttack;
14    }
15 }
```

Now, we need to properly call all those three methods. The first one (`selectCurrentUnit`) will be called in `TurnSystem`, when it is a player unit turn.

```

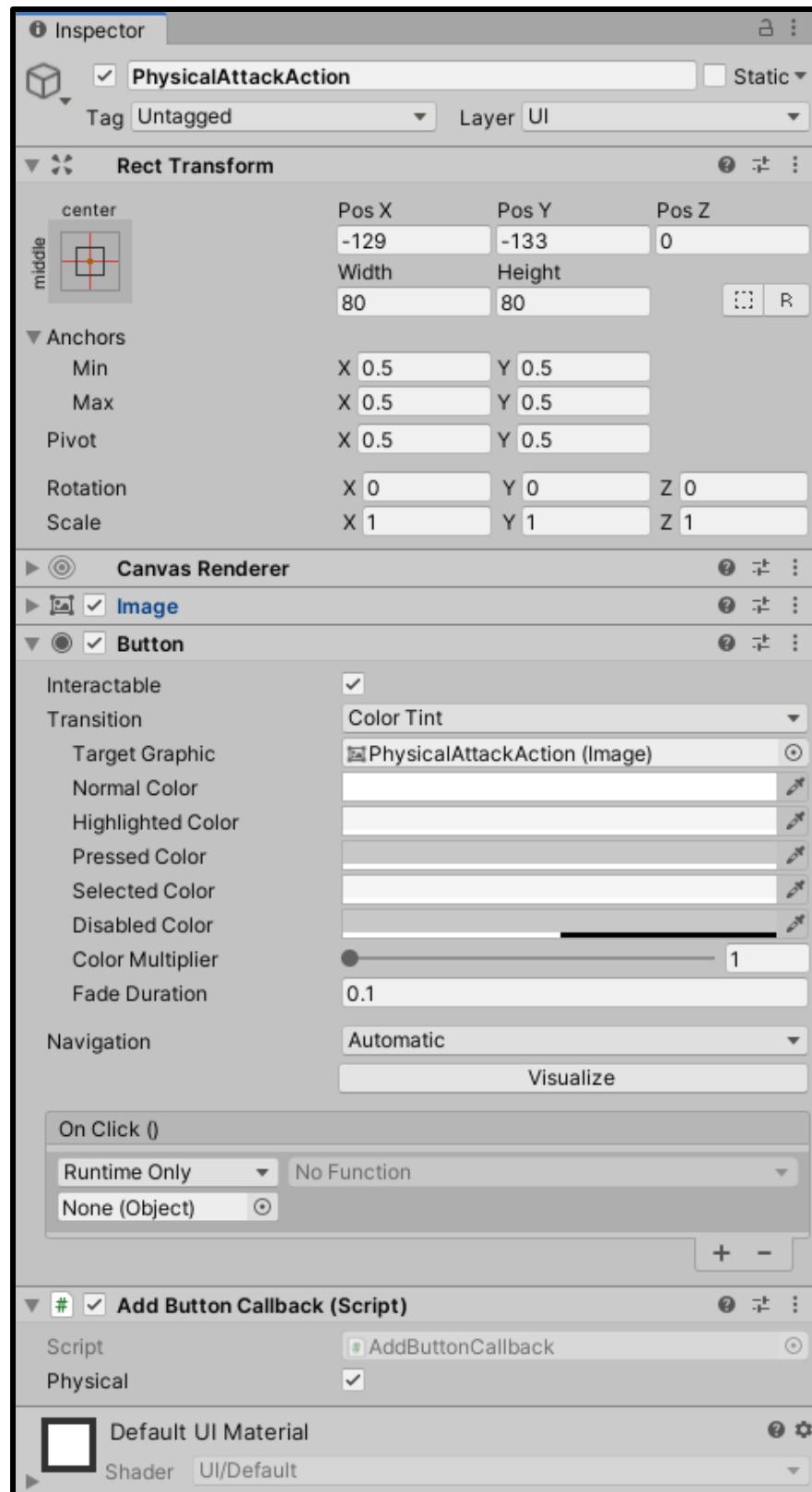
1 public void nextTurn ()
2 {
3     UnitStats currentUnitStats = unitsStats [0];
4     unitsStats.Remove(currentUnitStats);
5
6     if(!currentUnitStats.isDead ())
7     {
8         GameObject currentUnit = currentUnitStats.gameObject;
9
10        currentUnitStats.calculateNextActTurn(currentUnitStats.nextActTurn);
11        unitsStats.Add(currentUnitStats);
12        unitsStats.Sort();
13
14        if(currentUnit.tag == "PlayerUnit")
15        {
16            this.playerParty.GetComponent<SelectUnit>().selectCurrentUnit(
17                currentUnit.gameObject);
18        }
19        else
20        {
21            currentUnit.GetComponent<EnemyUnitAction>().act();
22        }
23    }
24    else
25    {
26        this.nextTurn();
27    }
28 }
```

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

The second one (selectAttack), will be called by the PhysicalAttackAction and MagicalAttackAction buttons in the HUDCanvas. Since the PlayerParty object is not from the same scene as these buttons, we can't add the OnClick callbacks in the inspector. So, we are going to do that using the following script (added to those buttons objects), which will add the callback in the Start method. The callback will simply call the selectAttack method from SelectUnit. This same script should be added to both buttons, only changing the "physical" attribute.

```
1 public class AddButtonCallback : MonoBehaviour
2 {
3     [SerializeField]
4     private bool physical;
5
6     // Use this for initialization
7     void Start ()
8     {
9         this.gameObject.GetComponent<Button>().onClick.AddListener(() => addCallback());
10    }
11
12     private void addCallback ()
13     {
14         GameObject playerParty = GameObject.Find("PlayerParty");
15         playerParty.GetComponent<SelectUnit>().selectAttack(this.physical);
16     }
17 }
```



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

The third method (attackEnemyTarget) will be called from the enemy unit menu item. When creating the CreateEnemyMenuItems script, we left the selectEnemyTarget (which is the button callback) empty. Now, we are going to implement it. This method is going to find the PlayerParty object and call its attackEnemyTarget method.

```
1 public void selectEnemyTarget ()
2 {
3     GameObject partyData = GameObject.Find("PlayerParty");
4     partyData.GetComponent<SelectUnit>().attackEnemyTarget(this.gameObject);
5 }
```

Finally, now we need to update the HUD to show the current unit face, health and mana.

We are going to use the following script to show the unit health and mana. This script will start its initial localScale in the Start method. Then, in the Update method it will update the localScale according to the current stat value of the unit. Also, it will have a method to change the current unit being showed and an abstract method to retrieve the current stat value.

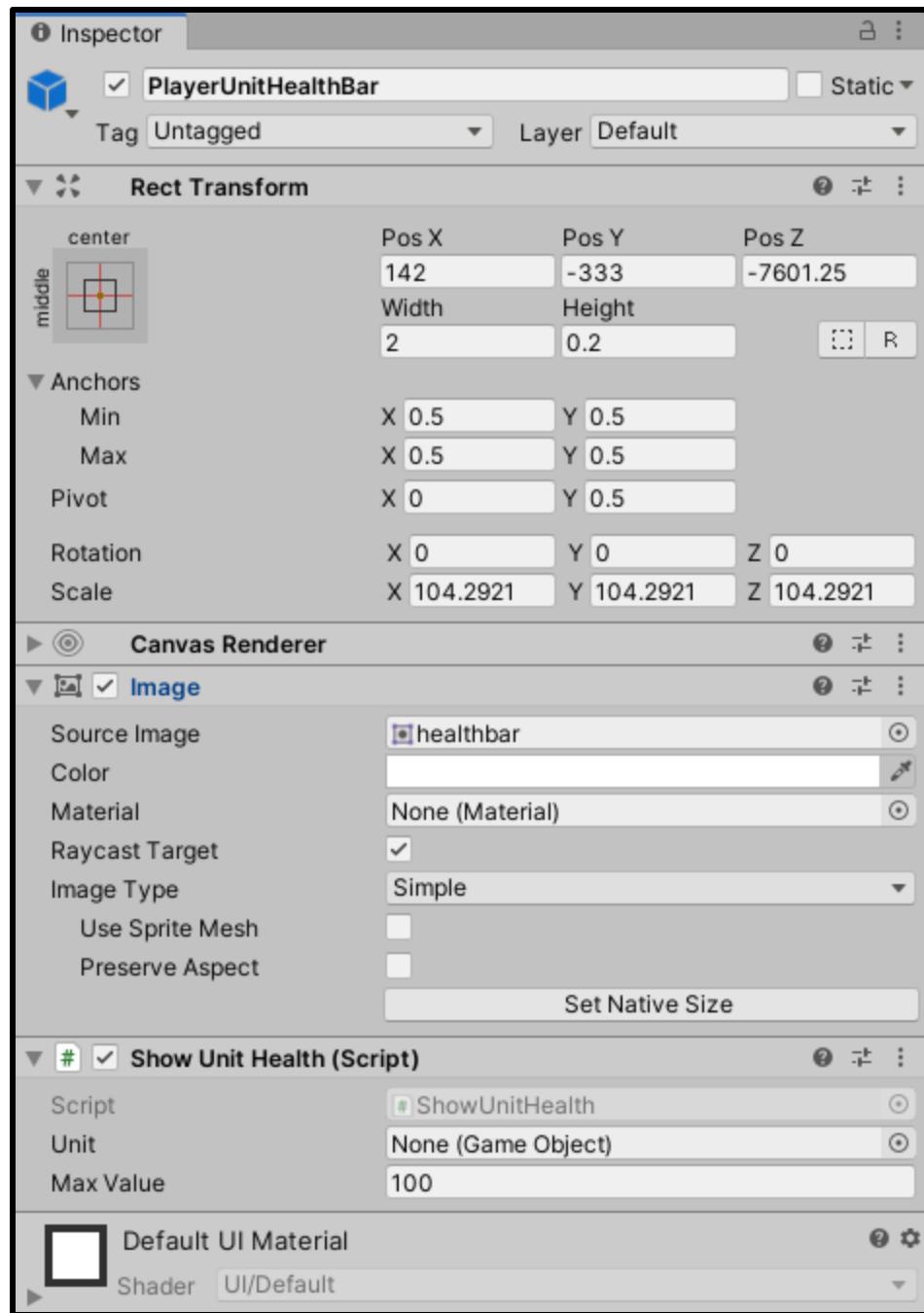
```
1 public abstract class ShowUnitStat : MonoBehaviour
2 {
3     [SerializeField]
4     protected GameObject unit;
5
6     [SerializeField]
7     private float maxValue;
8
9     private Vector2 initialScale;
10
11    void Start ()
12    {
13        this.initialScale = this.gameObject.transform.localScale;
14    }
15
16    void Update ()
17    {
18        if(this.unit)
19        {
20            float newValue = this.newStatValue();
21            float newScale = (this.initialScale.x * newValue) / this.maxValue;
22            this.gameObject.transform.localScale = new Vector2(newScale, this.initialScale.y);
23        }
24    }
25
26    public void changeUnit (GameObject newUnit)
27    {
28        this.unit = newUnit;
29    }
30
31    abstract protected float newStatValue();
32 }
```

Instead of directly using this script, we are going to create two other ones that specialize it, implementing the abstract method: ShowUnitHealth and ShowUnitMana. The only method in those two scripts will be newStatValue, which will return the correct unit stats (health or mana).

```
1 public class ShowUnitHealth : ShowUnitStat
2 {
3     override protected float newStatValue ()
4     {
5         return unit.GetComponent<UnitStats>().health;
6     }
7 }
```

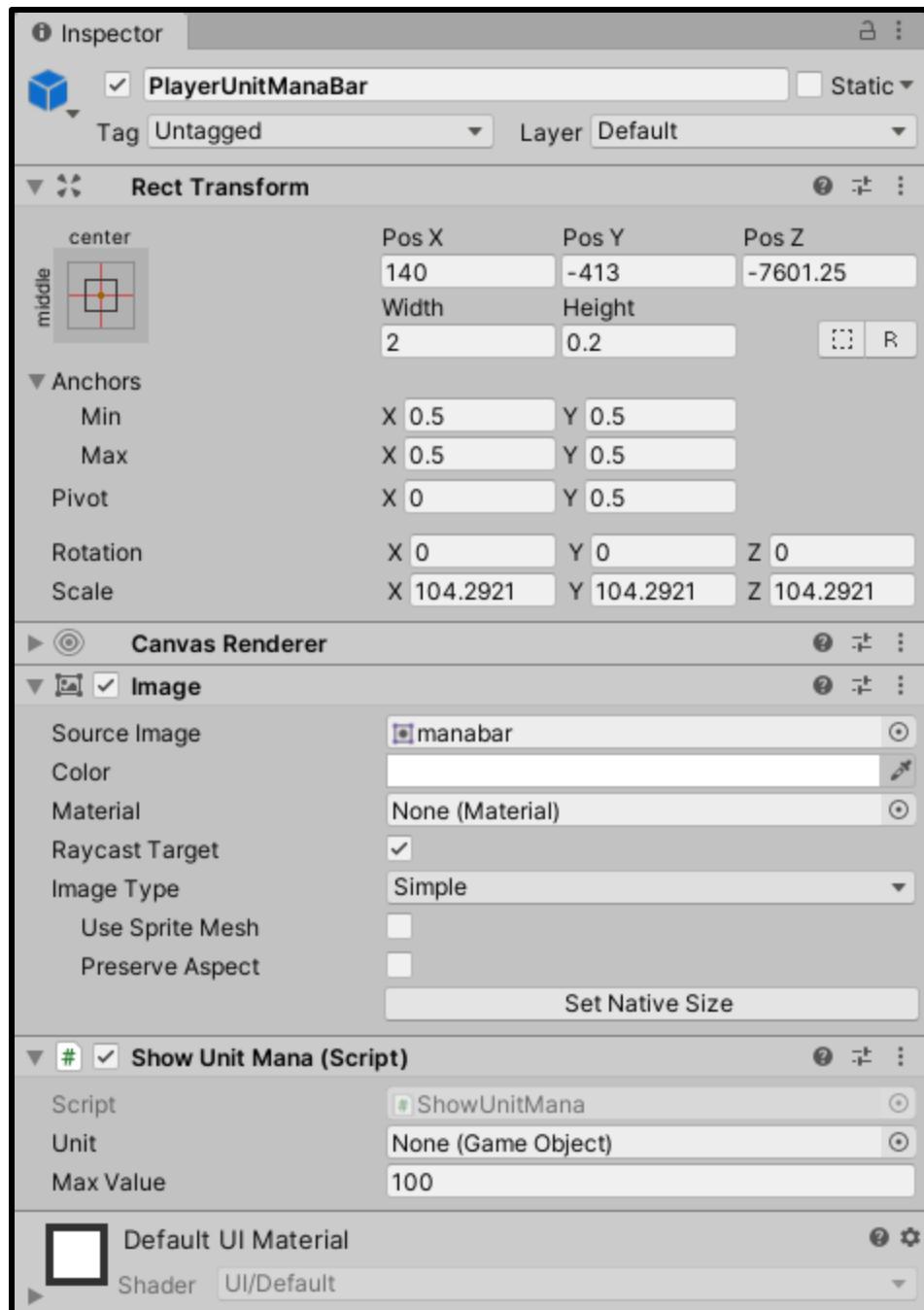
```
1 public class ShowUnitMana : ShowUnitStat
2 {
3     override protected float newStatValue ()
4     {
5         return unit.GetComponent<UnitStats>().mana;
6     }
7 }
```

Now we can add those two scripts to the health and mana bar objects. Another thing to do is to change their Pivot in the X coordinate to be zero, so that it will change the scale only on the right side of the bar.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved



Finally, we need to call the `changeUnit` method in those scripts when the current unit changes. This will start in the `selectCurrentUnit` method of `SelectUnit`. After setting the `actionsMenu` as active, it will call a method called `updateHUD` for the current unit.

```

1 public void selectCurrentUnit (GameObject unit)
2 {
3     this.currentUnit = unit;
4     this.actionsMenu.SetActive(true);
5     this.currentUnit.GetComponent<PlayerUnitAction>().updateHUD();
6 }
```

The updateHUD method, by its turn, will start by setting the sprite of the PlayerUnitFace object to be the current unit face (saved as an attribute of PlayerUnitAction). Then, it will set itself as the current unit in both ShowUnitHealth and ShowUnitMana.

```

1 [SerializeField]
2 private Sprite faceSprite;
3
4 public void updateHUD ()
5 {
6     GameObject playerUnitFace = GameObject.Find("PlayerUnitFace") as GameObject;
7     playerUnitFace.GetComponent<Image>().sprite = this.faceSprite;
8
9     GameObject playerUnitHealthBar = GameObject.Find("PlayerUnitHealthBar") as GameObject;
10    playerUnitHealthBar.GetComponent<ShowUnitHealth>().changeUnit(this.gameObject);
11
12    GameObject playerUnitManaBar = GameObject.Find("PlayerUnitManaBar") as GameObject;
13    playerUnitManaBar.GetComponent<ShowUnitMana>().changeUnit(this.gameObject);
14 }
```

By now, you can try playing the game to see if you can select different actions, and if the stats are being correctly updated. The only action from the menu that we still have to implement is the Run action.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

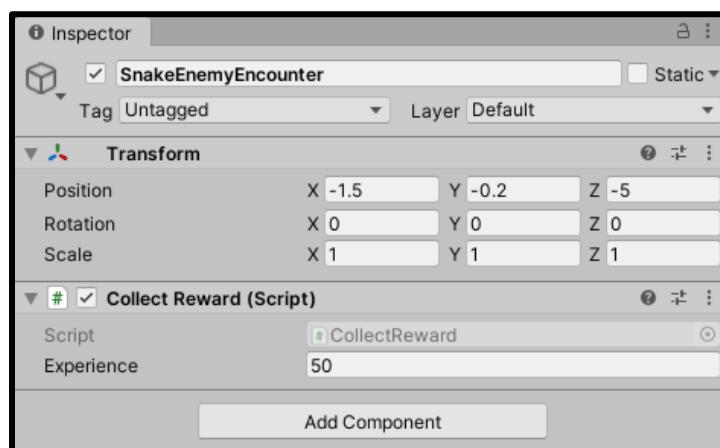
Finishing the Battle

We still have to add ways of finishing the battle. There are three ways of doing that:

1. All enemy units are dead, and the player has won the battle
2. All player units are dead, and the player has lost the battle
3. The player ran from the battle

If the player wins the battle, it will receive a reward from the enemy encounter. In order to do that we are going to use the following script, which will be added to the enemy encounter object. In the Start method, it will set the enemy encounter in the TurnSystem object. Then, the collectReward method (which will be called from TurnSystem), will equally divide the encounter experience among all living player units.

```
1 public class CollectReward : MonoBehaviour
2 {
3     [SerializeField]
4     private float experience;
5
6     public void Start ()
7     {
8         GameObject turnSystem = GameObject.Find("TurnSystem");
9         turnSystem.GetComponent<TurnSystem>().enemyEncounter = this.gameObject;
10    }
11
12    public void collectReward ()
13    {
14        GameObject[] livingPlayerUnits = GameObject.FindGameObjectsWithTag("PlayerUnit");
15        float experiencePerUnit = this.experience / (float)livingPlayerUnits.Length;
16
17        foreach(GameObject playerUnit in livingPlayerUnits)
18        {
19            playerUnit.GetComponent<UnitStats>().receiveExperience(experiencePerUnit);
20        }
21
22        Destroy(this.gameObject);
23    }
24 }
```



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Now we need to implement the receiveExperience method used in collectReward. This will be a method from UnitStats used only to save the received experience. This can be used later to implement a level system in the game, but we are not going to do that in this tutorial.

```
1 public void receiveExperience (float experience)
2 {
3     this.currentExperience += experience;
4 }
```

Finally, let's call the collectReward method in TurnSystem. We are going to change the nextTurn method to check if there are still living enemy units, by finding the objects with the "EnemyUnit" tag. Remember that when a unit dies, we change its tag to "DeadUnit", so that it won't be found by this method. If there are no remaining enemy units, it calls the collectReward method for the enemy encounter, and go backs to the Town scene.

On the other hand, if there are no remaining player units, that means the player has lost the battle, so the game goes back to the Title scene.

```
1 public void nextTurn ()
2 {
3     GameObject[] remainingEnemyUnits = GameObject.FindGameObjectsWithTag("EnemyUnit");
4
5     if(remainingEnemyUnits.Length == 0)
6     {
7         this.enemyEncounter.GetComponent<CollectReward>().collectReward();
8         SceneManager.LoadScene("Town");
9     }
10
11    GameObject[] remainingPlayerUnits = GameObject.FindGameObjectsWithTag("PlayerUnit");
12
13    if(remainingPlayerUnits.Length == 0)
14    {
15        SceneManager.LoadScene("Title");
16    }
17
18    UnitStats currentUnitStats = unitsStats [0];
19    unitsStats.Remove(currentUnitStats);
20
21    if(!currentUnitStats.isDead ())
22    {
23        GameObject currentUnit = currentUnitStats.gameObject;
24
25        currentUnitStats.calculateNextActTurn(currentUnitStats.nextActTurn);
26        unitsStats.Add(currentUnitStats);
27        unitsStats.Sort();
28    }
}
```

```

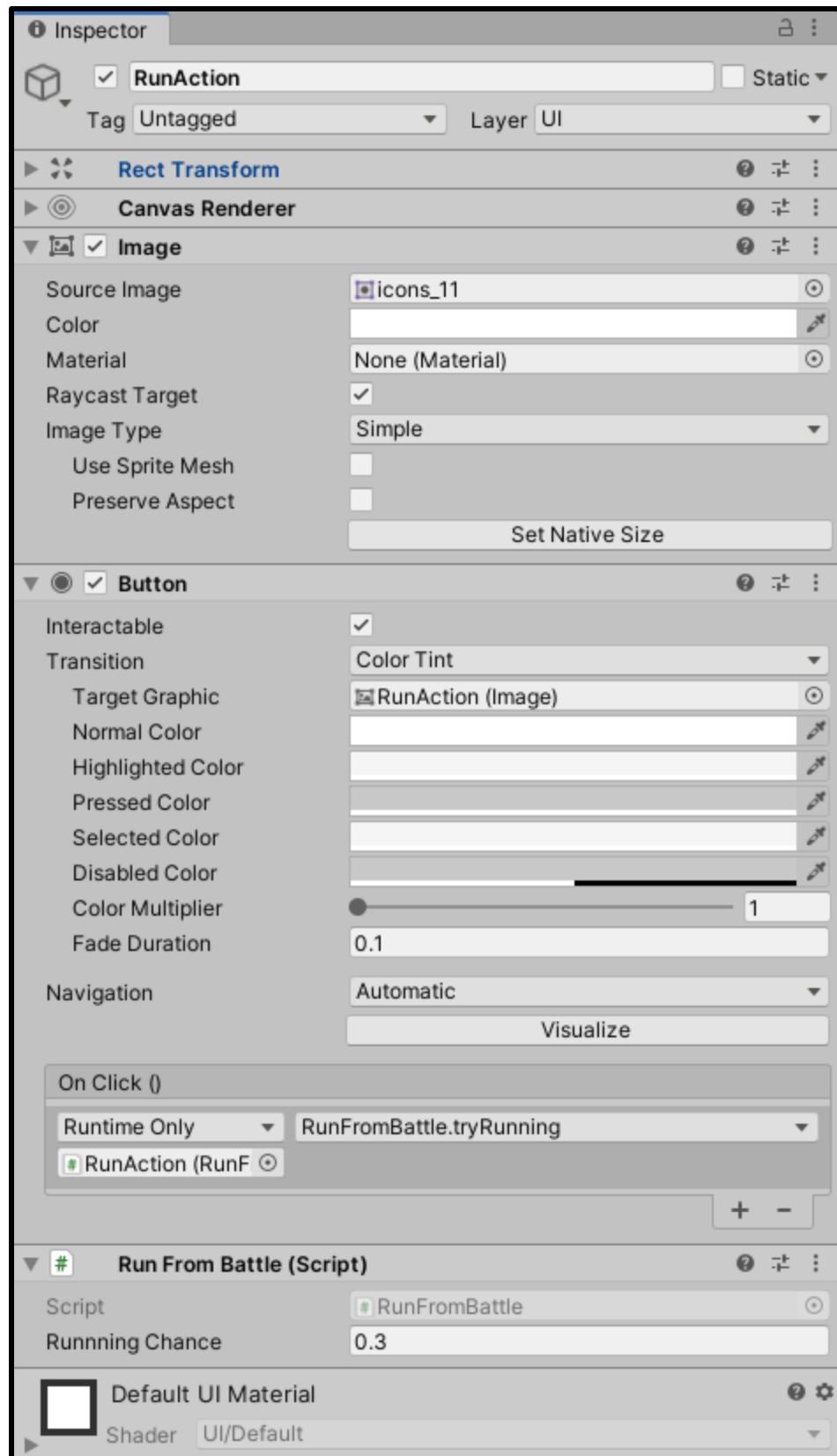
29     if(currentUnit.tag == "PlayerUnit")
30     {
31         this.playerParty.GetComponent<SelectUnit>().selectCurrentUnit(
32             currentUnit.gameObject);
33     }
34     else
35     {
36         currentUnit.GetComponent<EnemyUnitAction>().act();
37     }
38 }
39 else
40 {
41     this.nextTurn();
42 }
43 }
```

The last way of finishing a battle is by running from it. This can be done by selecting the run action in the actions menu. So, we need to attach the script below to the run button, and add its OnClick callback.

The RunFromBattle script will have a tryRunning method. This method will generate a random number between 0 and 1, and compare it with a runningChance attribute. If the generated random number is less than the running chance, the player successfully avoids the battle, and goes back to the Town scene. Otherwise, the next turn starts.

```

1 public class RunFromBattle : MonoBehaviour
2 {
3     [SerializeField]
4     private float runningChance;
5
6     public void tryRunning ()
7     {
8         float randomNumber = Random.value;
9
10        if(randomNumber < this.runningChance)
11        {
12            SceneManager.LoadScene("Town");
13        }
14        else
15        {
16            GameObject.Find("TurnSystem").GetComponent<TurnSystem>().nextTurn();
17        }
18    }
19 }
```



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Finally, by now you should have everything working. Try playing battles until the end to check if everything is working. Now you can also try adding different enemy encounters and tuning some game parameters, such as units stats and attack multipliers.

Also, try adding things that could not have been covered in the tutorial, such as more intelligent enemies and a level system. You might also consider mastering techniques for [mobile development](#) and optimizing your game for use on a smartphone! The sky is the limit here, so don't be afraid to experiment!



How to Create a Multiplayer Game in Unity

By Renan Oliveira

In this tutorial we are going to build a simple demo to learn how to use Unity multiplayer features. Our game will have a single scene where we will implement a multiplayer Space Shooter. In our demo multiple players will be able to join the same game to shoot enemies that will be randomly spawned.

In order to follow this tutorial, you are expected to be familiar with the following concepts:

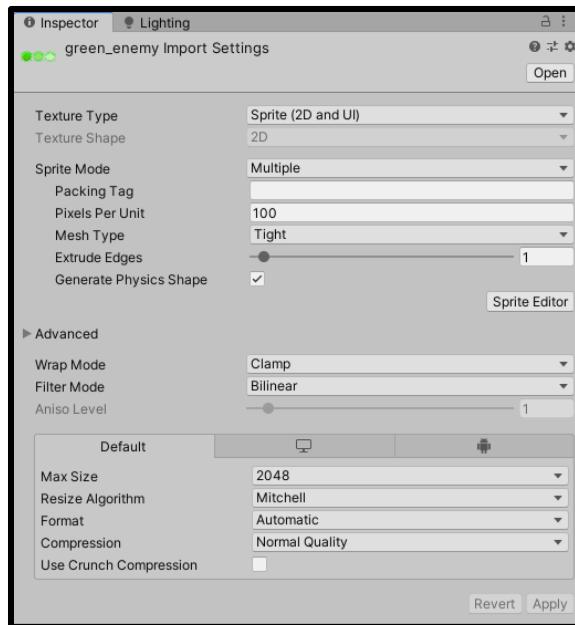
- C# programming
- Using Unity Editor, such as importing assets, creating prefabs and adding components

We'll be using an asset known as Mirror. This is an expansion on Unity's default networking system, adding in new features and fixing a large amount of bugs and problems.

Creating Project and Importing Assets

Before starting reading the tutorial, you need to create a new Unity project and import all sprites available through the source code. In order to do that, create a folder called Sprites and copy all sprites to this folder. Unity Inspector will automatically import them to your project.

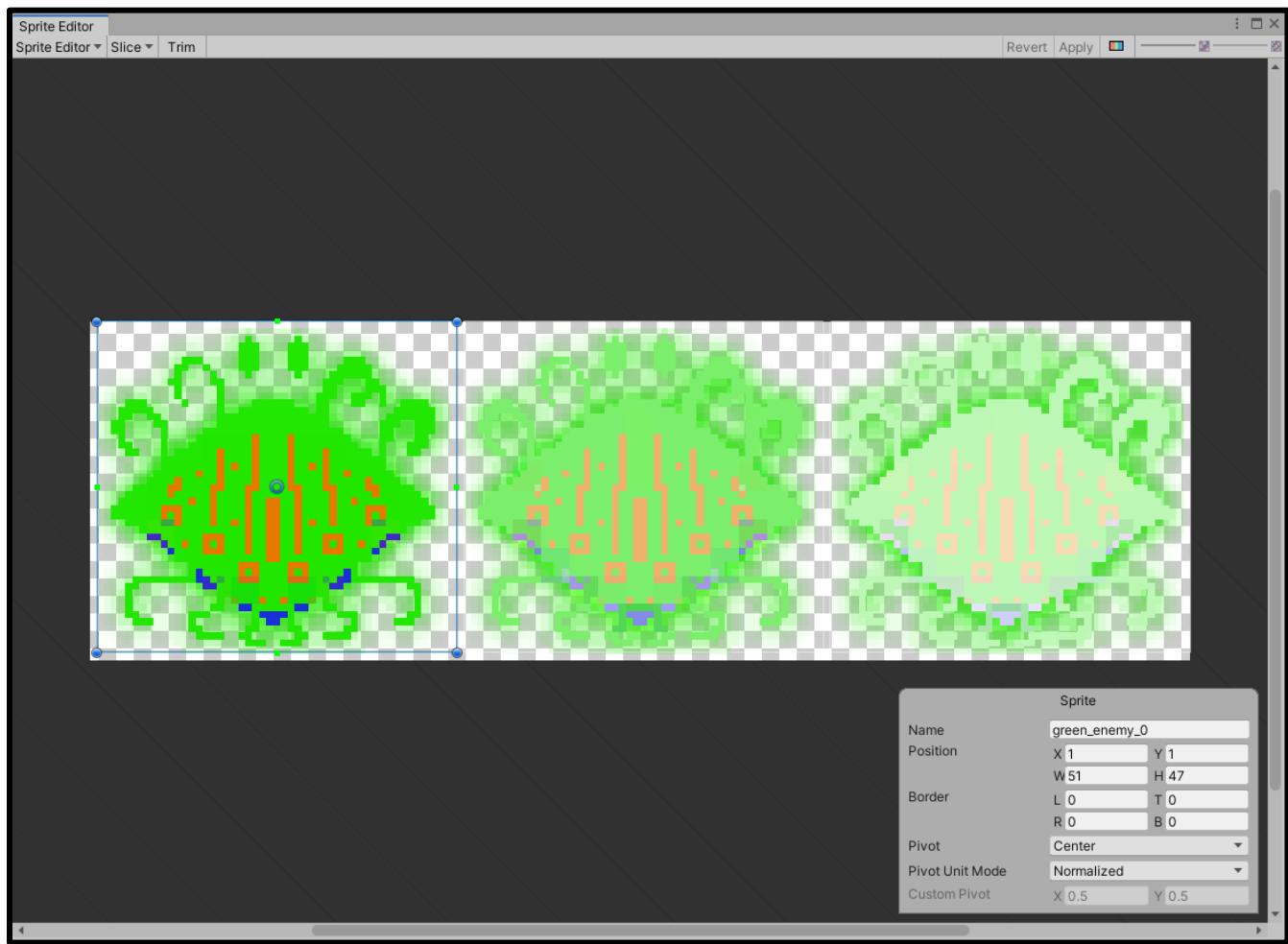
However, some of those sprites are in spritesheets, such as the enemies spritesheets, and they need to be sliced. In order to do that, we need to set the Sprite Mode as Multiple and open the Sprite Editor.



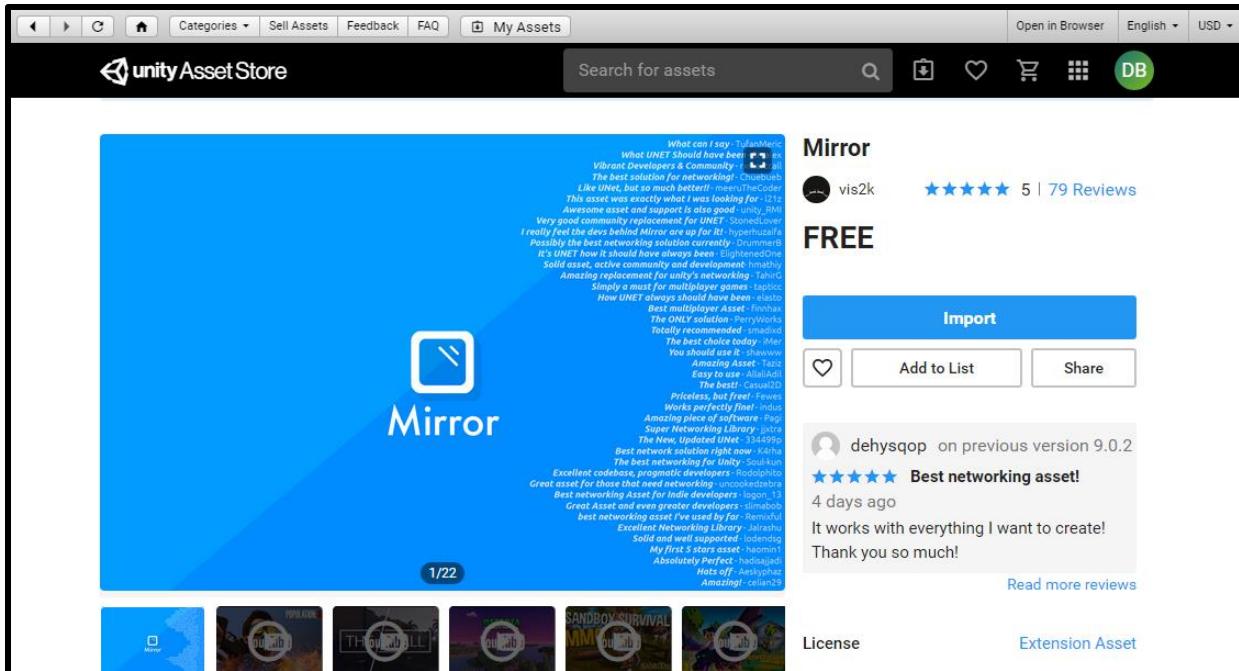
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

In the Sprite Editor (shown below), you need to open the slice menu and click in the Slice button, with the slice type as automatic. Finally, hit apply before closing.



We also need to import the Mirror asset. Go to the Asset Store (*Window > Asset Store*) and download "Mirror".



Source Code Files

You can download the tutorial source code files [here](#).

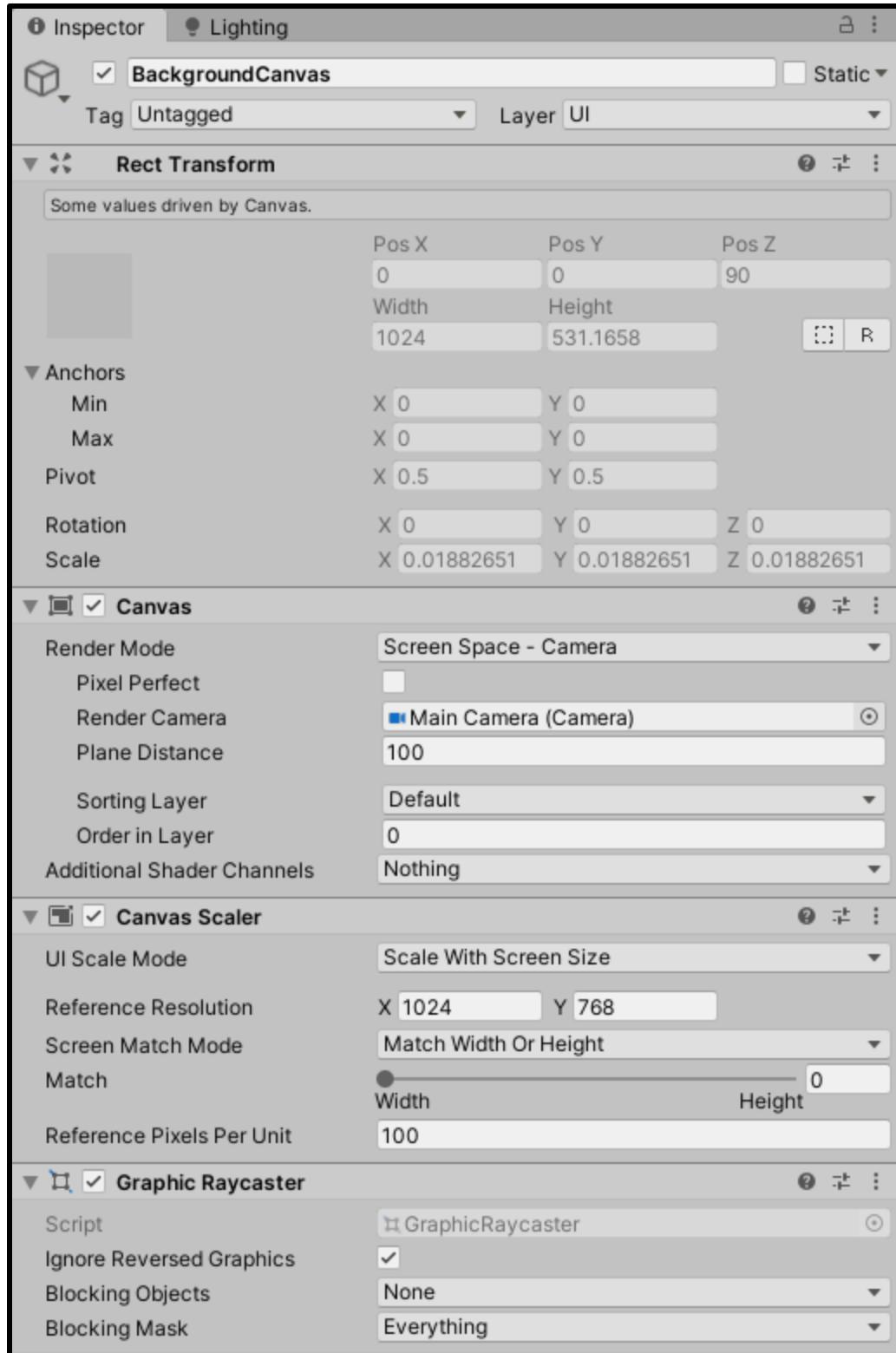
Background canvas

The first thing we are going to do is creating a background canvas to show a background image.

We can do that by creating a new Image in the Hierarchy, and it will automatically create a new Canvas (rename it to BackgroundCanvas).

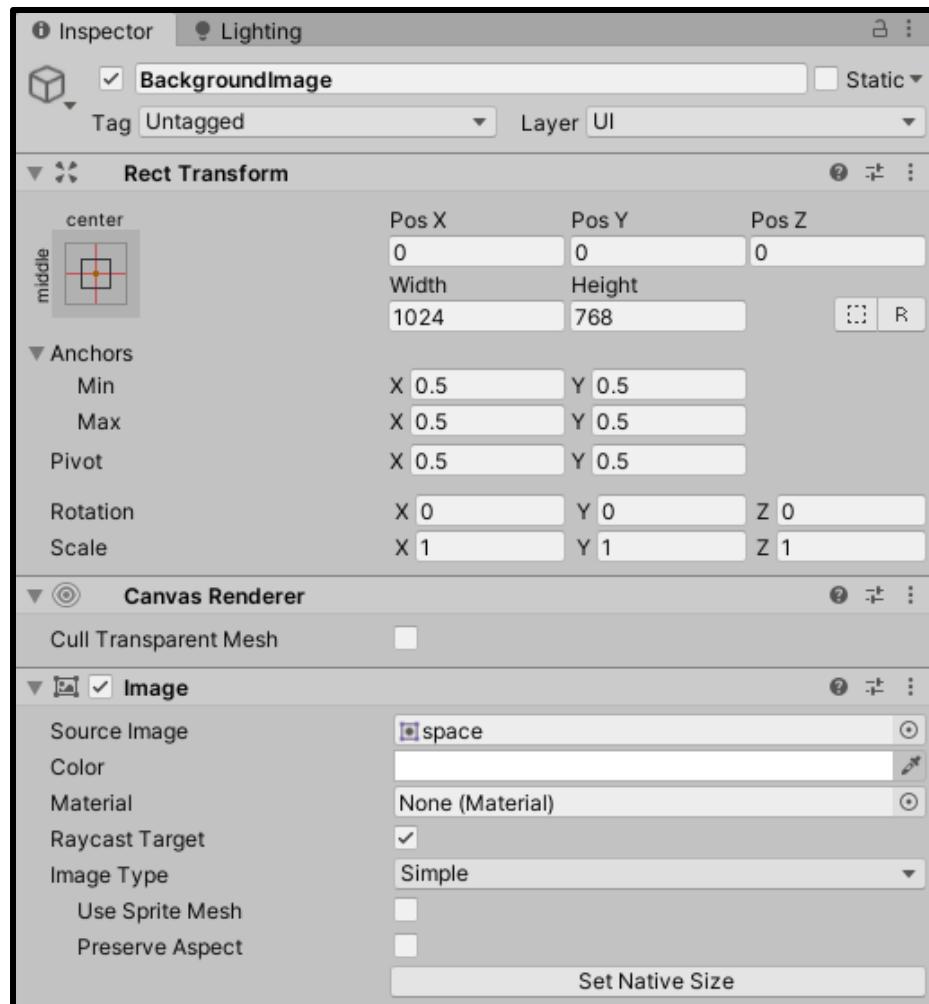
In the BackgroundCanvas, we need to set its Render Mode to be Screen Space - Camera (remember to attach your Main Camera to it). Then, we are going to set its UI Scale Mode to Scale With Screen Size. This way the Canvas will appear in the background, and not in front of the other objects.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

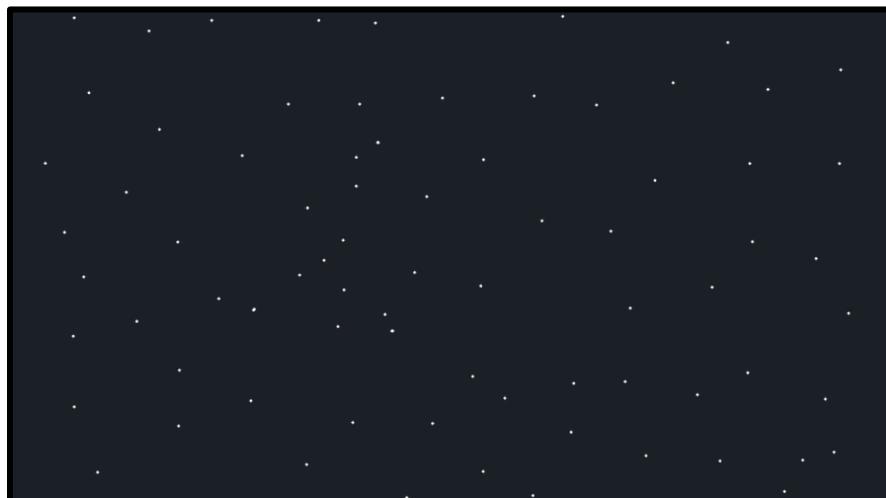


In the BackgroundImage we only need to set the Source Image, which will be the space one.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity



Try running the game now and you should see the space background in the game.

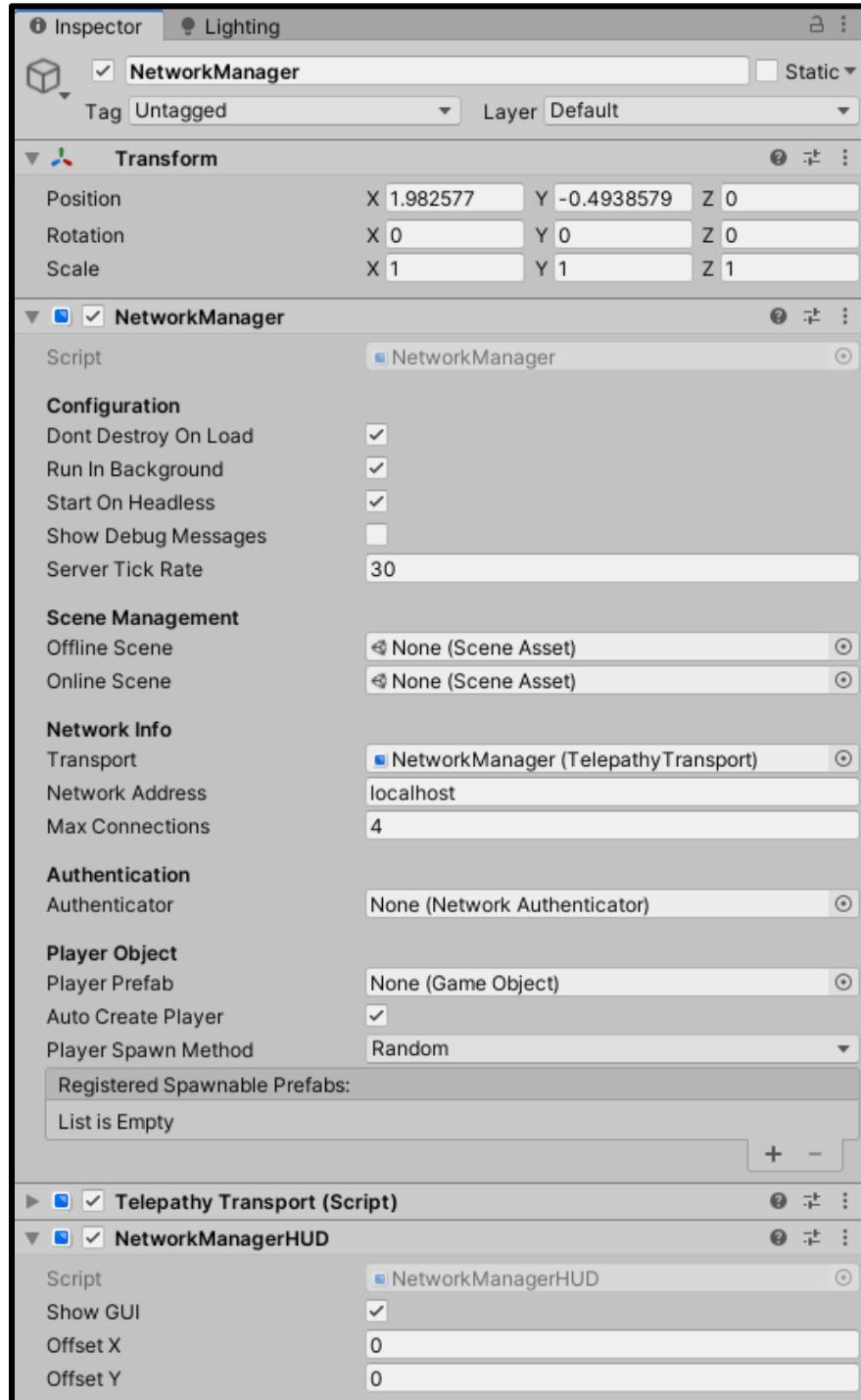


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Network Manager

In order to have a multiplayer game, we need a GameObject with the NetworkManager and NetworkManagerHUD components, so let's create one.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

This object will be responsible for managing for connecting different clients in a game and synchronizing the game objects among all clients. The Network Manager HUD shows a simple HUD for the players to connect to a game.

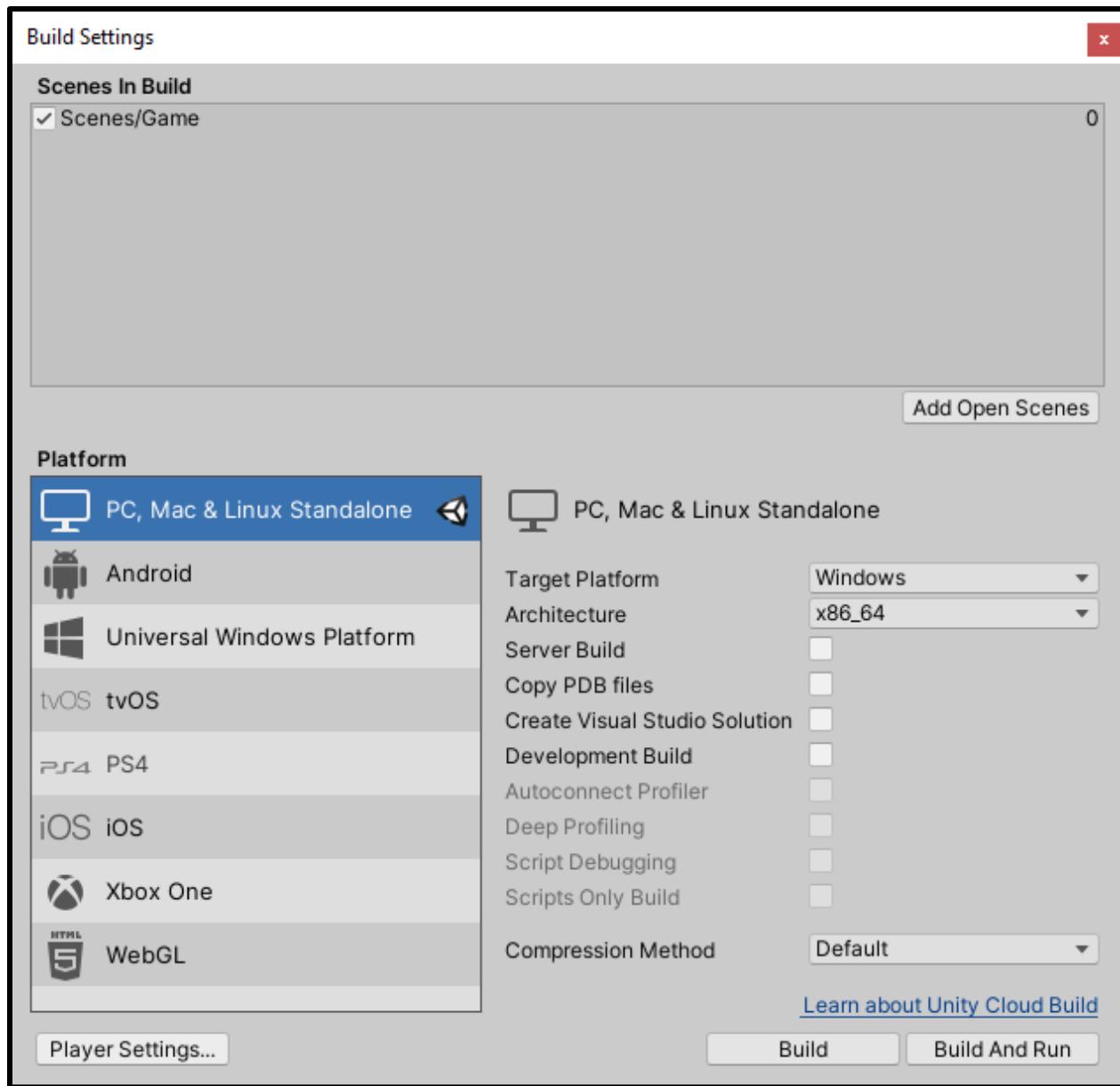
For example, if you play the game now, you should see the following screen:



In this tutorial we are going to use the LAN Host and LAN Client options. Unity multiplayer games work in the following way: first, a player starts a game as host (by selecting LAN Host). A host works as a client and a server at the same time. Then, other players can connect to this host by as clients (by selecting LAN Client). The client communicates with the server, but do not execute any server only code. So, in order to test our game we are going to open two instances of it, one as the Host and another as the Client.

However, you can not open two instances of the game in the Unity Editor. In order to do so, you need to build your game and run the first instance from the generated executable file. The second instance can be run from the Unity Editor (in the Play Mode).

In order to build your game you need to add the Game scene to the build. So, go to File -> Build Settings and add the Game scene to build. Then, you can generate and executable file and run it by clicking on File -> Build & Run. This will open a new window with the game. After doing that, you can enter Play Mode in the Unity Editor to run the second instance of the game. This is what we are going to do every time we need to test a multiplayer game.

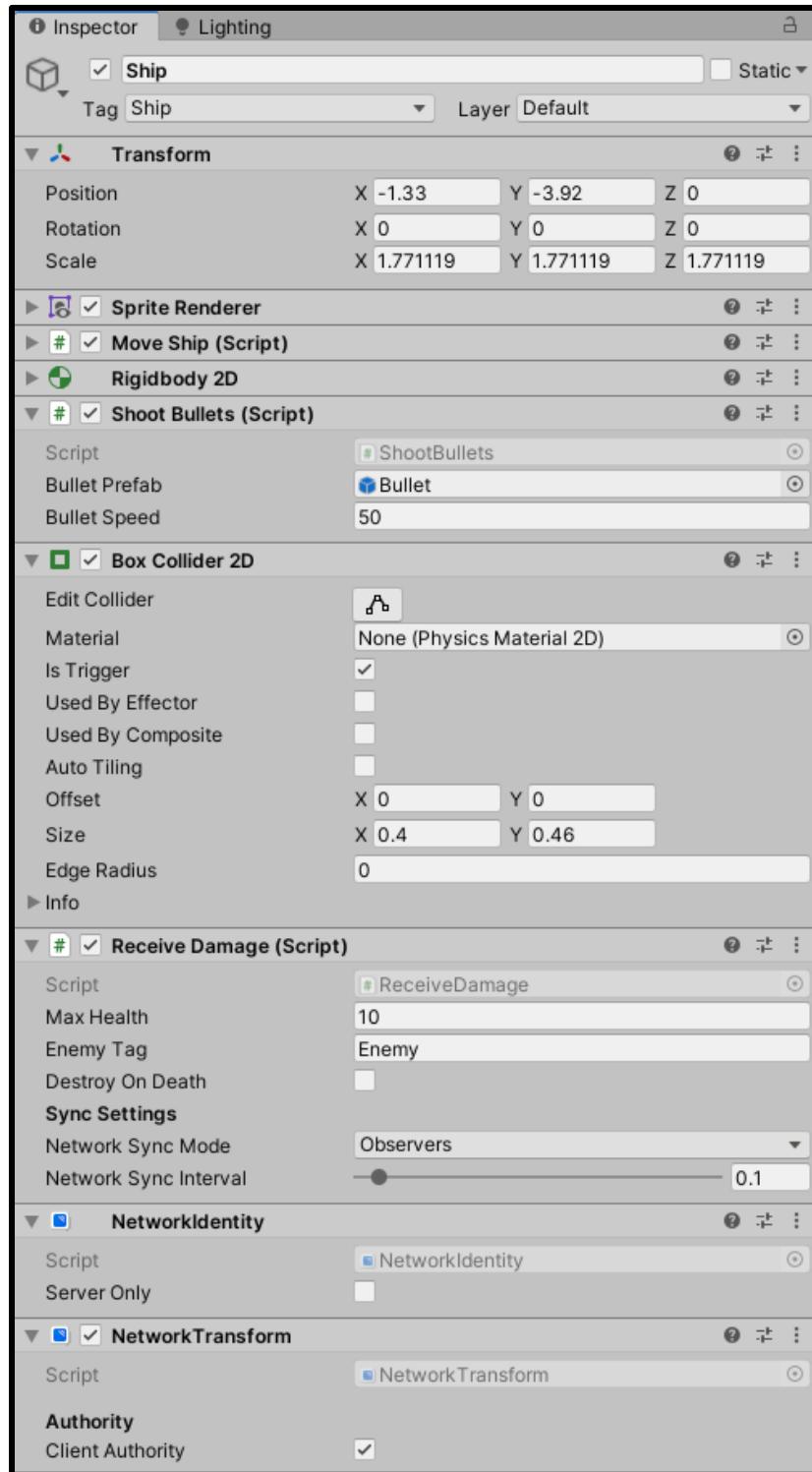


Ship Movement

Now that we have the NetworkManager, we can start creating the game objects which will be managed by it. The first one we are going to create is the player ship.

For now, the ship will only move horizontally in the screen, with its position being updated by the NetworkManager. Later on, we are going to add to it the ability to shoot bullets and receive damage.

So, first of all, create a new GameObject called Ship and make it a prefab. The figure below shows the Ship prefab components, which I will explain now.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

In order for a game object to be managed by the NetworkManager, we need to add the NetworkIdentity component to it. In addition, since the ship will be controlled by the player, we are going to set the Local Player Authority check box for it.

The NetworkTransform component, by its turn, is responsible for updating the Ship position throughout the server and all the clients. Otherwise, if we've moved the ship in one screen, its position wouldn't be updated in the other screens. NetworkIdentity and NetworkTransform are the two components necessary for multiplayer features. Enable Client Authority on the NetworkTransform component.

Now, to handle movement and collisions, we need to add a RigidBody2D and a BoxCollider2D. In addition, the BoxCollider2D will be a trigger (Is Trigger set to true), since we don't want collisions to affect the ship physics.

Finally, we are going to add a MoveShip script, which will have a Speed parameter. Other scripts will be added later, but that's it for now.

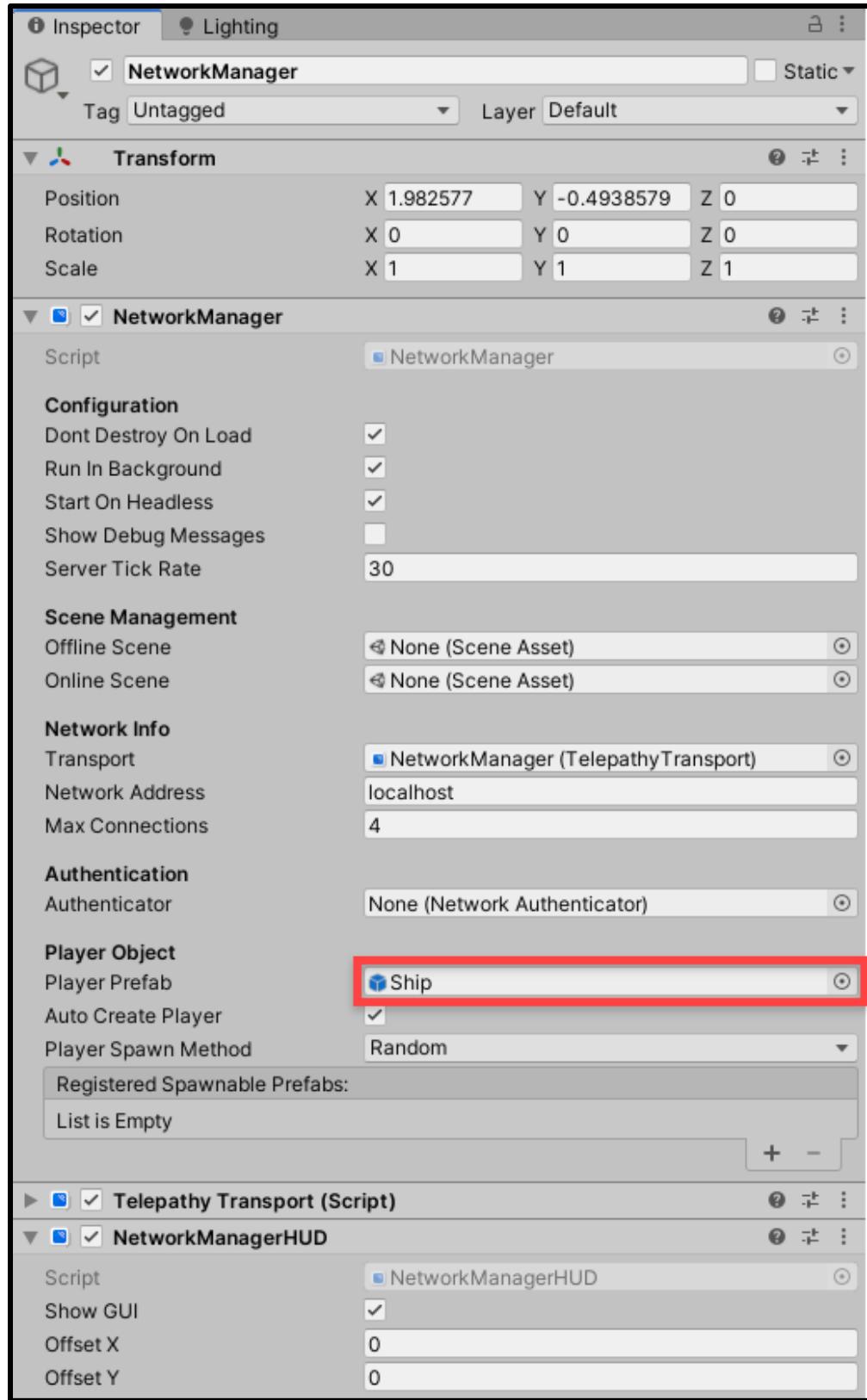
The MoveShip script is very simple, in the FixedUpdate method we get the movement from the Horizontal Axis and set the ship velocity accordingly. However, there are two very important network-related things that must be explained.

First, typically all Scripts in a Unity game inherit MonoBehaviour to use its API. However, in order to use the Network API the script must inherit NetworkBehaviour instead of MonoBehaviour. You need to include the Networking namespace (using UnityEngine.Networking) in order to do that.

Also, in a Unity multiplayer game, the same code is executed in all instances of the game (host and clients). To let the players to control only their ships, and not all ships in the game, we need to add an If condition in the beginning of the FixedUpdate method checking if this is the local player (if you're curious on how the game would work without this If condition, try removing it. When moving a ship in a screen, all ships should move together).

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using Mirror;
5
6 public class MoveShip : NetworkBehaviour
7 {
8     [SerializeField]
9     private float speed;
10
11    void FixedUpdate ()
12    {
13        if(this.isLocalPlayer)
14        {
15            float movement = Input.GetAxis("Horizontal");
16            GetComponent<Rigidbody2D>().velocity = new Vector2(movement * speed, 0.0f);
17        }
18    }
19 }
```

Before playing the game, we still need to tell the NetworkManager that the Ship prefab is the Player Prefab. We do that by selecting it in the Player Prefab attribute in the NetworkManager component. By doing so, everytime that a player starts a new instance of the game, the NetworkManager will instantiate a Ship.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

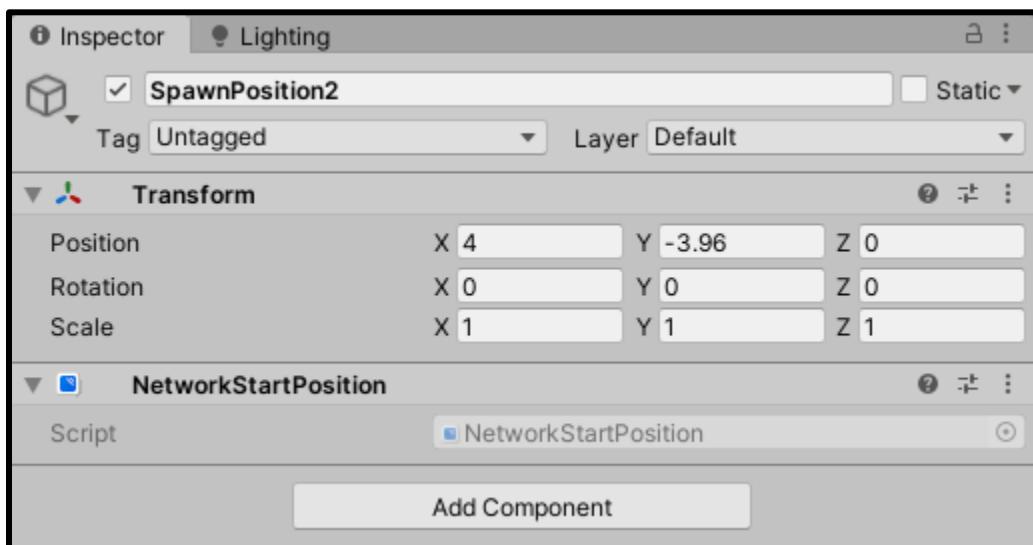
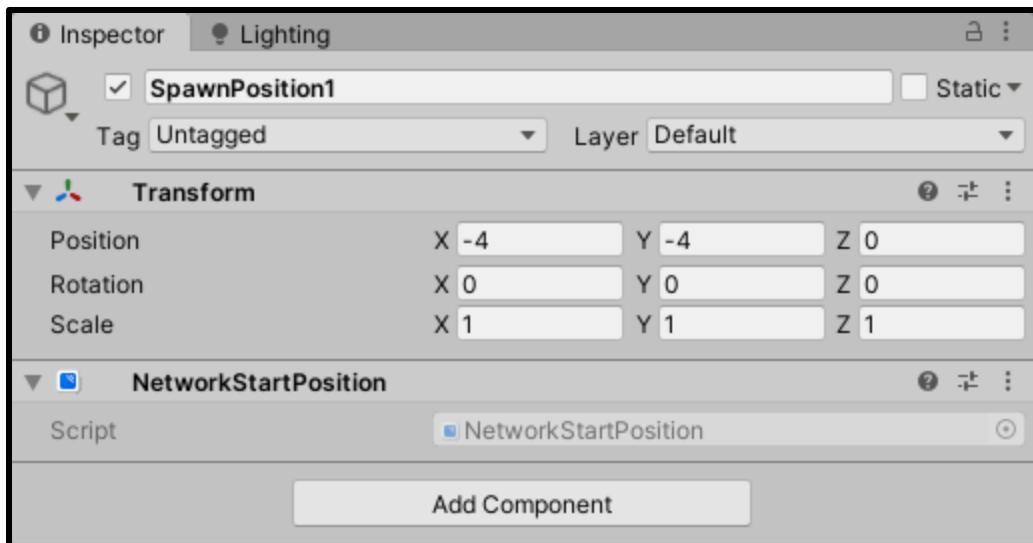
Now you can try playing the game. The ship movement should be synchronized between the two instances of the game.



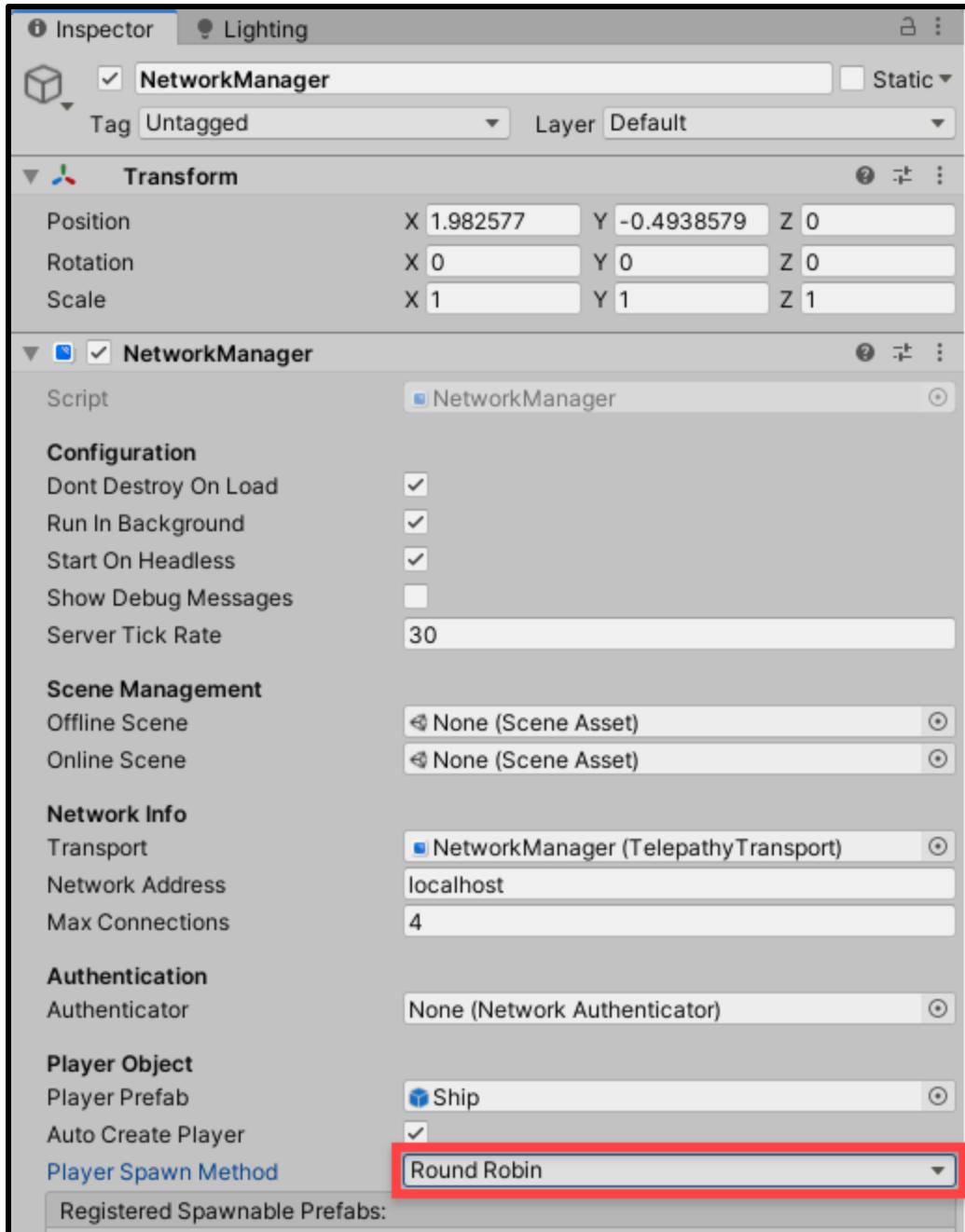
Spawn Positions

Until now all ships are being spawned in the middle of the screen. However, it would be more interesting to set some predefined spawn positions, which is actually easy to do with Unity multiplayer API.

First, we need to create a new Game Object to be our spawn position and place it in the desired spawn position. Then, we add the NetworkStartPosition component to it. I'm going to create two spawn positions, one in the coordinate (-4, -4) and the other one in the coordinate (4, -4).



Now we need to define how the NetworkManager will use those positions. We do that by configuring the Player Spawn Method attribute. There are two options there: Random and Round Robin. Random means that, for each game instance, the manager will choose the player start position at random among the spawn positions. Round Robin means it will go sequentially through all spawn positions until all of them have been used (for example, first SpawnPosition1 then SpawnPosition2). Then, it starts again from the beginning of the list. We are going to pick Round Robin.



By now you can try playing the game again and see if the ships are being spawned in the correct positions.

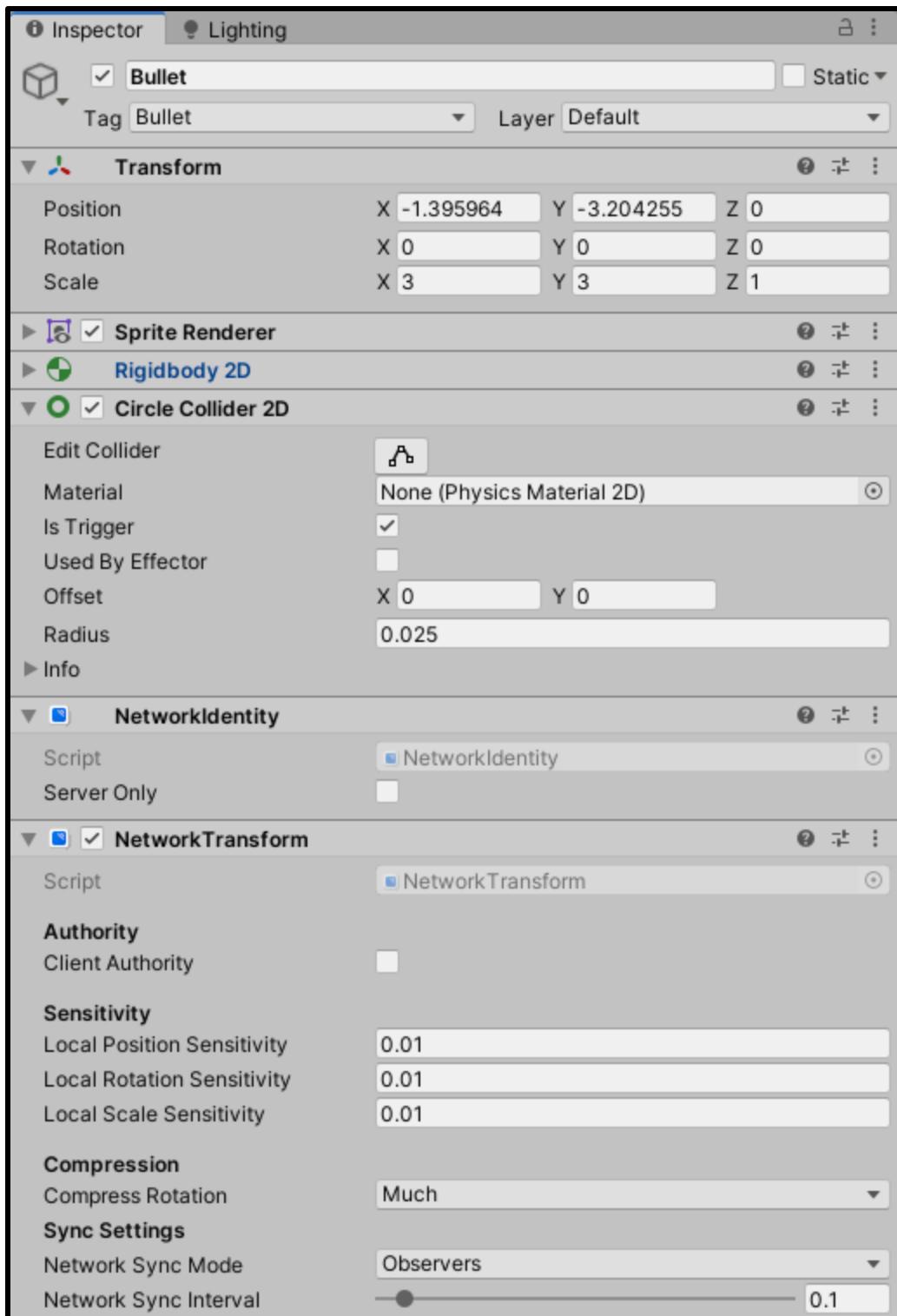


Shooting Bullets

The next thing we are going to add in our game is giving ships the ability to shooting bullets. Also, those bullets must be synchronized among all instances of the game.

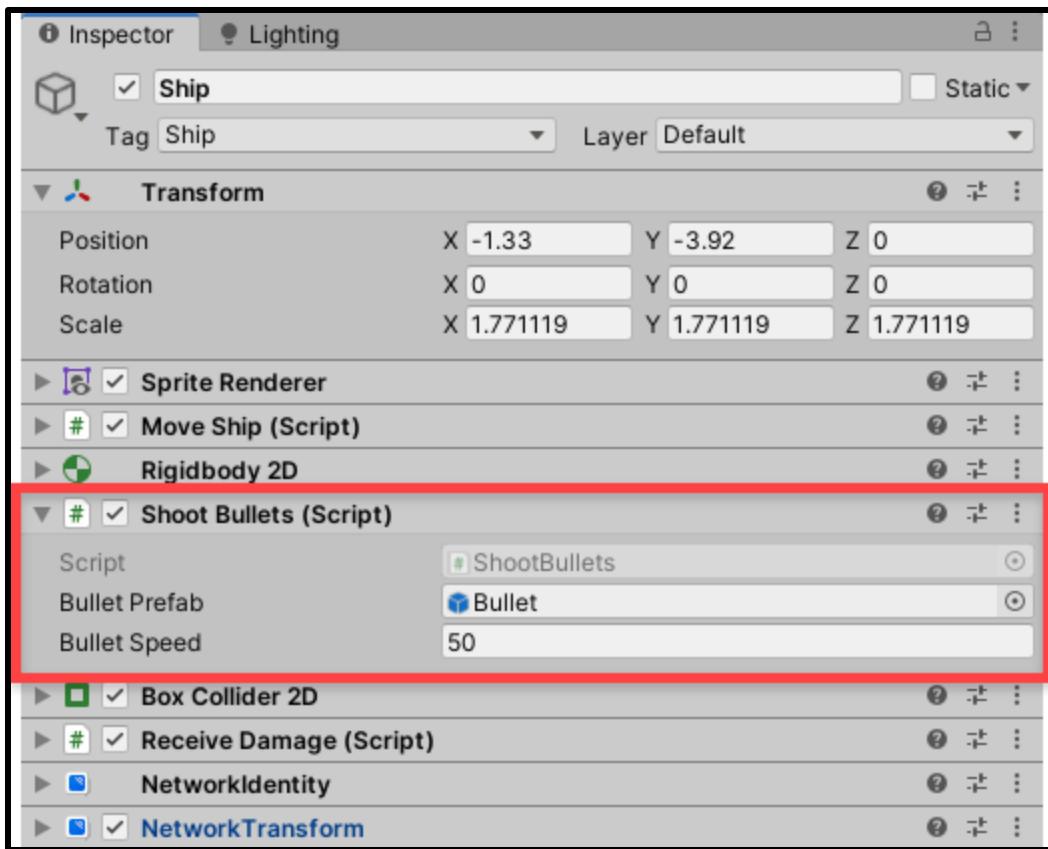
First of all, we need to create the Bullet prefab. So, create a new GameObject called Bullet and make it a prefab. In order to manage it in the network, it needs the NetworkIdentity and NetworkTransform components, as in the ship prefab. However, once a bullet is created, the game does not need to propagate its position through the network, since the position is updated by the physics engine. So, we are going to change the Network Send Rate in the Network Transform to 0, to avoid overloading the network.

Also, bullets will have a speed and should collide with enemies later. So, we are going to add a RigidBody2D and a CircleCollider2D to the prefab. Again, notice that the CircleCollider2D is a trigger.



Now that we have the bullet prefab, we can add a ShootBullets script to the Ship. This script will have as parameters the Bullet Prefab and the Bullet Speed.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity



The ShootBullets script is also a NetworkBehaviour, and it is shown below. In the update method, it is going to check if the local player has pressed the Space key and, if so, it will call a method to shoot bullets. This method will instantiate a new bullet, set its velocity and destroy it after one second (when the bullet has already left the screen).

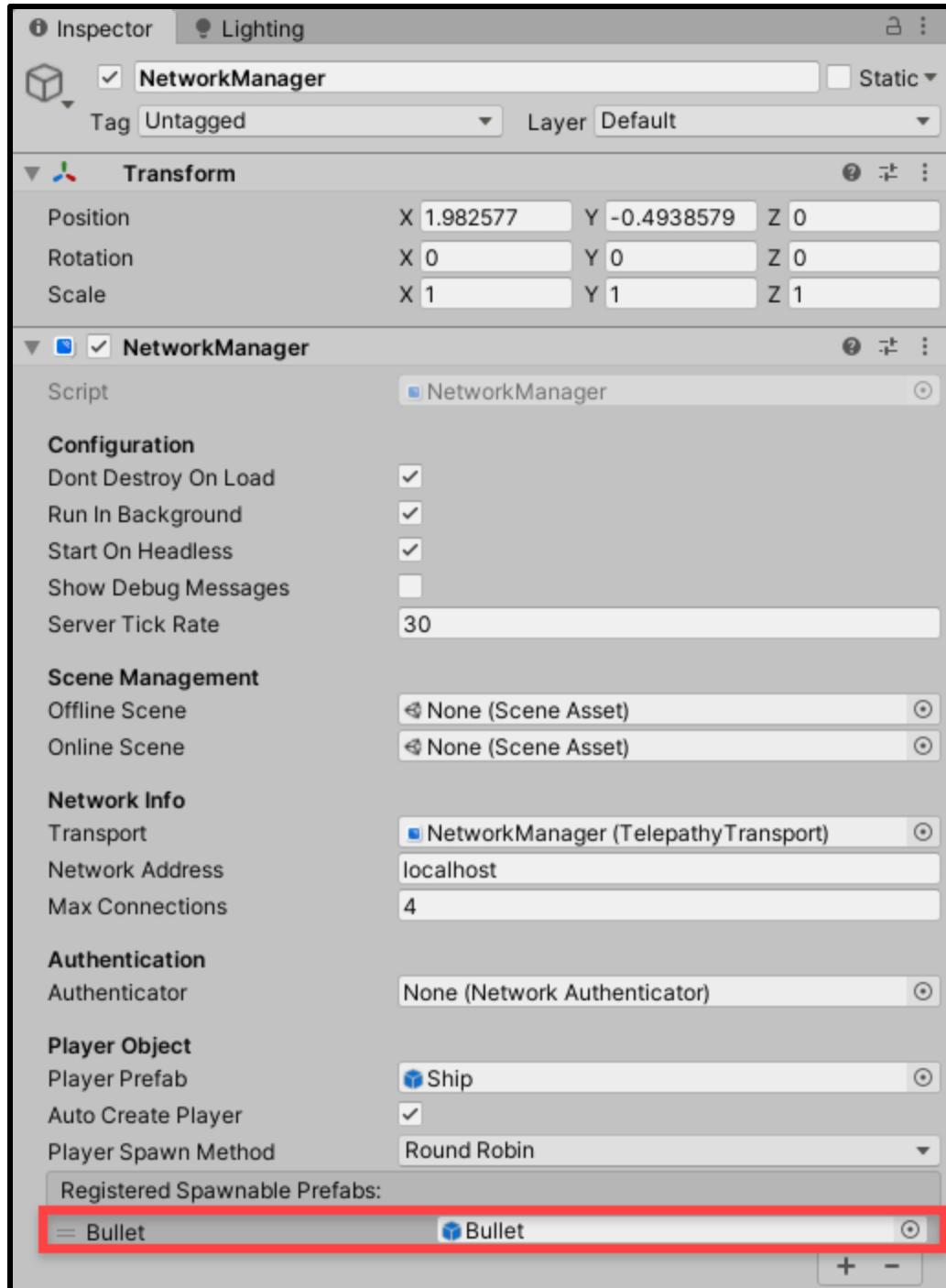
Again, there are some important network concepts that must be explained here. First, there is a [Command] tag above the CmdShoot method. This tag and the "Cmd" in the beginning of the method name make it a special method called a Command. In unity, a command is a method that is executed in the server, although it has been called in the client. In this case, when the local player shoots a bullet, instead of calling the method in the client, the game will send a command request to the server, and the server will execute the method.

Also, there is call to NetworkServer.Spawn in the CmdShoot method. The Spawn method is responsible for creating the bullet in all instances of the game. So, what CmdShoot does is creating a bullet in the server, and then the server replicates this bullet among all clients. Notice that this is possible only because CmdShoot is a Command, and not a regular method.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using Mirror;
5
6  public class ShootBullets : NetworkBehaviour
7  {
8      [SerializeField]
9      private GameObject bulletPrefab;
10
11     [SerializeField]
12     private float bulletSpeed;
13
14     void Update ()
15     {
16         if(this.isLocalPlayer && Input.GetKeyDown(KeyCode.Space))
17         {
18             this.CmdShoot();
19         }
20     }
21
22     [Command]
23     void CmdShoot ()
24     {
25         GameObject bullet = Instantiate(bulletPrefab, this.transform.position,
26                                         Quaternion.identity);
27         bullet.GetComponent<Rigidbody2D>().velocity = Vector2.up * bulletSpeed;
28         NetworkServer.Spawn(bullet);
29         Destroy(bullet, 1.0f);
30     }
31 }
```

Finally, we need to tell the network manager that it can spawn bullets. We do that by adding the bullet prefab in the Registered Spawnable Prefabs list.



Now, you can try playing the game and shoot bullets. Bullets must be synchronized among all instances of the game.

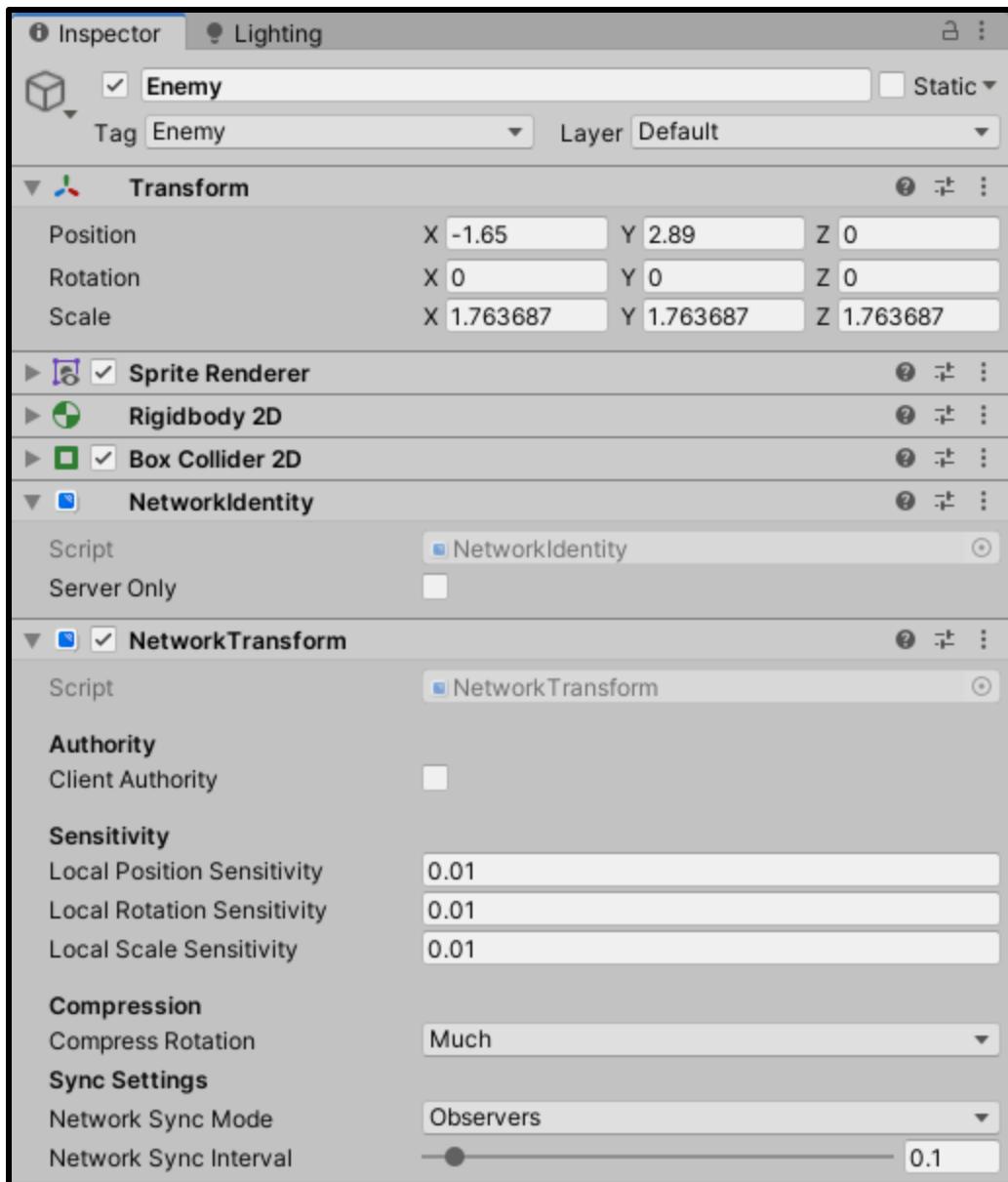
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity



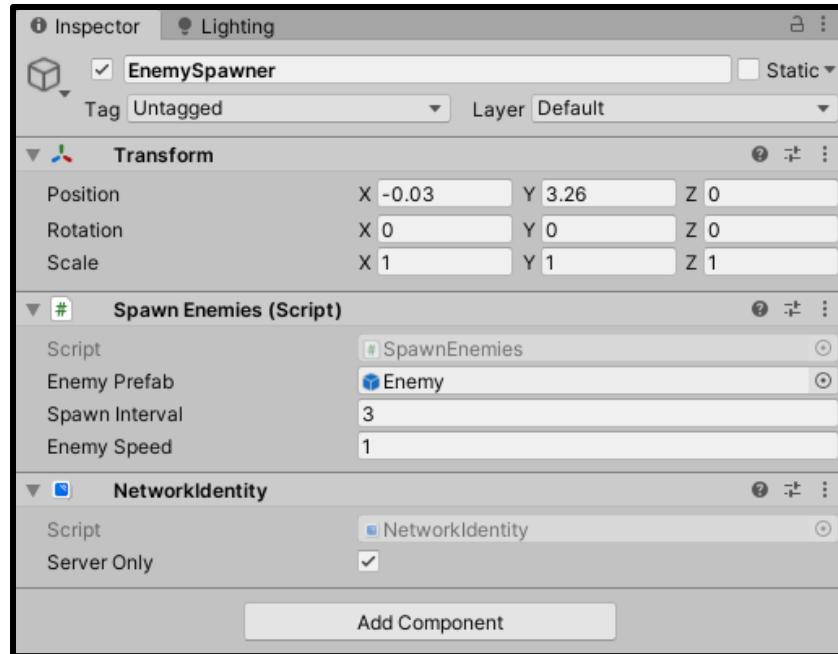
Spawning Enemies

The next step in our game is adding enemies.

First, we need an Enemy prefab. So, create a new GameObject called Enemy and make it a prefab. Like ships, enemies will have a Rigidbody2D and BoxCollider2D to handle movements and collisions. Also, it will need a NetworkIdentity and NetworkTransform, to be handled by the NetworkManager. Later on we are going to add a script to it as well, but that's it for now.



Now, let's create a GameObject called `EnemySpawner`. The `EnemySpawner` will also have a `NetworkIdentity`, but now we are going to select the `Server Only` field in the component. This way, the spawner will exist only in the server, since we don't want enemies to be created in each client. Also, it will have a `SpawnEnemies` script, which will spawn enemies in a regular interval (the parameters are the enemy prefab, the spawn interval and the enemy speed).



The `SpawnEnemies` script is shown below. Notice that we are using a new Unity method here: `OnStartServer`. This method is very similar to `OnStart`, the only difference is that it is called only for the server. When this happens, we are going to call `InovkeRepeating` to call the `SpawnEnemy` method every 1 second (according to `spawnInterval`).

The `SpawnEnemy` method will instantiate a new enemy in a random position, and use `NetworkServer.Spawn` to replicate it among all instances of the game. Finally, the enemy will be destroyed after 10 seconds.

```

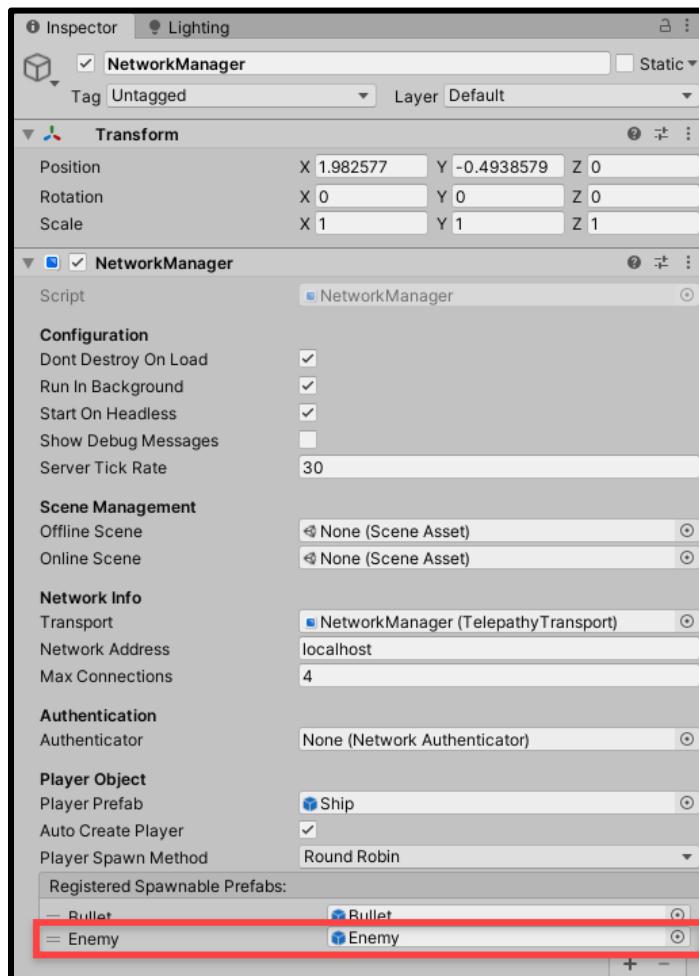
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using Mirror;
5
6  public class SpawnEnemies : NetworkBehaviour
7  {
8      [SerializeField]
9      private GameObject enemyPrefab;
10
11     [SerializeField]
12     private float spawnInterval = 1.0f;
13
14     [SerializeField]
15     private float enemySpeed = 1.0f;
16 }
```

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

```

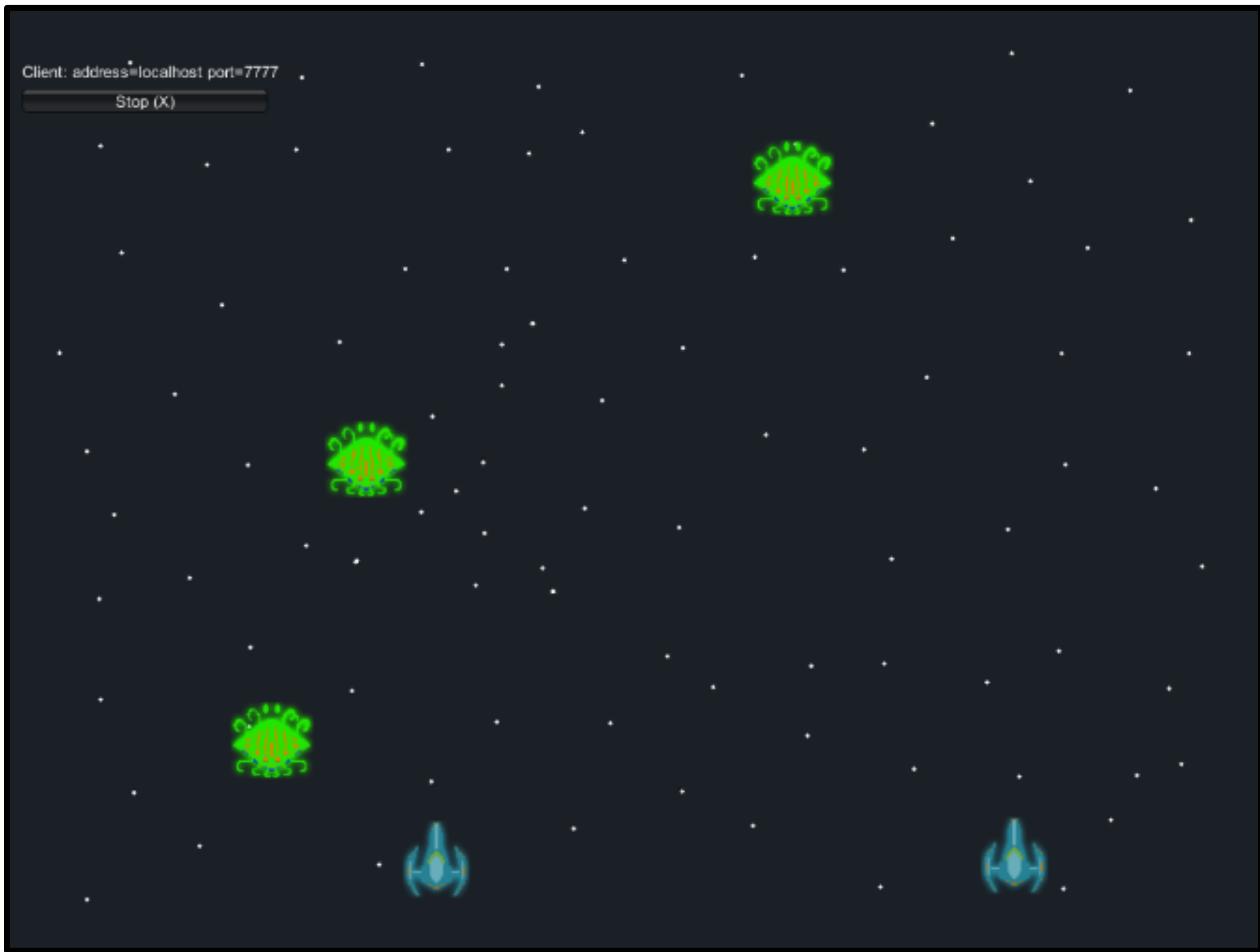
17     public override void OnStartServer ()
18     {
19         InvokeRepeating("SpawnEnemy", this.spawnInterval, this.spawnInterval);
20     }
21
22     void SpawnEnemy ()
23     {
24         Vector2 spawnPosition = new Vector2(Random.Range(-4.0f, 4.0f),
25                                             this.transform.position.y);
26         GameObject enemy = Instantiate(enemyPrefab, spawnPosition,
27                                         Quaternion.identity) as GameObject;
28         enemy.GetComponent<Rigidbody2D>().velocity = new Vector2(0.0f, -this.enemySpeed);
29         NetworkServer.Spawn(enemy);
30         Destroy(enemy, 10);
31     }
32 }
```

Before playing the game, we need to add the Enemy prefab to the Registered Spawnable Prefabs list.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Now you can try playing the game now with enemies. Notice that the game still doesn't have any collision handling yet. So you won't be able to shoot enemies. This will be our next step.



Taking Damage

The last thing we are going to add to our game is the ability to hit enemies and, unfortunately, to die to them. In order to keep this tutorial simple, I'm going to use the same script for both enemies and ships.

The script we are going to use is called `ReceiveDamage`, and it is shown below. It will have as configurable parameters `maxHealth`, `enemyTag` and `destroyOnDeath`. The first one is used to define the initial health of the object. The second one is used to detect collisions. For example, the `enemyTag` for ships will be "`Enemy`", while the `enemyTag` for enemies will be "`Bullet`". This way, we can make ships colliding only with enemies, and enemies colliding only with bullets. The last parameter (`destroyOnDeath`) will be used to determine if an object will be respawned or destroyed after dying.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using Mirror;
5
6  public class ReceiveDamage : NetworkBehaviour
7  {
8      [SerializeField]
9      private int maxHealth = 10;
10
11     [SyncVar]
12     private int currentHealth;
13
14     [SerializeField]
15     private string enemyTag;
16
17     [SerializeField]
18     private bool destroyOnDeath;
19
20     private Vector2 initialPosition;
21
22     // Use this for initialization
23     void Start ()
24     {
25         this.currentHealth = this.maxHealth;
26         this.initialPosition = this.transform.position;
27     }
28 }
```

```
29     void OnTriggerEnter2D (Collider2D collider)
30     {
31         if(collider.tag == this.enemyTag)
32         {
33             this.TakeDamage(1);
34             Destroy(collider.gameObject);
35         }
36     }
37 }
```

```

38     void TakeDamage (int amount)
39     {
40         if(this.isServer)
41         {
42             this.currentHealth -= amount;
43
44             if(this.currentHealth <= 0)
45             {
46                 if(this.destroyOnDeath)
47                 {
48                     Destroy(this.gameObject);
49                 }
50                 else
51                 {
52                     this.currentHealth = this.maxHealth;
53                     RpcRespawn();
54                 }
55             }
56         }
57     }
58
59     [ClientRpc]
60     void RpcRespawn ()
61     {
62         this.transform.position = this.initialPosition;
63     }
64 }
```

Now, let's analyze the methods. In the Start method, the script sets the currentHealth to be the maximum, and saves the initial position (the initial position will be used to respawn ships later). Also, notices that there is a [SyncVar] tag above the currentHealth attribute definition. This means that this attribute value must be synchronized among game instances. So, if an object receives damage in one instance, it will be propagated to all of them.

The OnTriggerEnter2D method is the one responsible for handling collisions (since the colliders we added were configured as triggers). First, we check if the collider tag is the enemyTag we are looking for, to handle collisions only against objects we are looking for (enemies against ships and bullets against enemies). If so, we call the TakeDamage method and destroy the other collider.

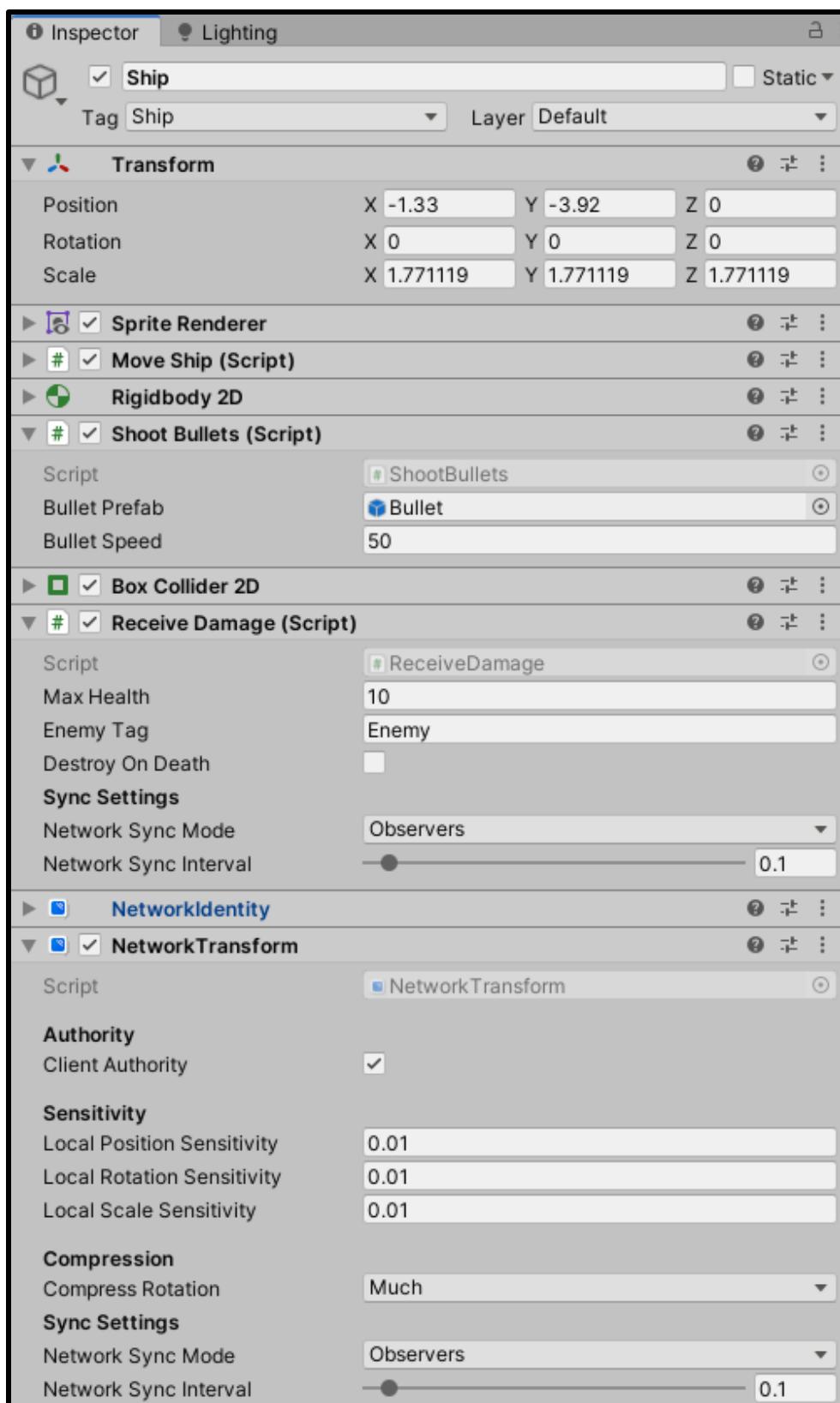
This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

The TakeDamage method, by its turn, will be called only in the server. This is because the currentHealth is already a SyncVar, so we only need to update it in the server, and it will be synchronized among the clients. Besides that, the TakeDamage method is simple, it decreases the currentHealth and checks if it is less than or equal to 0. If so, the object will be destroyed, if destroyOnDeath is true, or it the currentHealth will be reset and the object will be respawned. In practice, we will make enemies to be destroyed on death, while ships will be respawned.

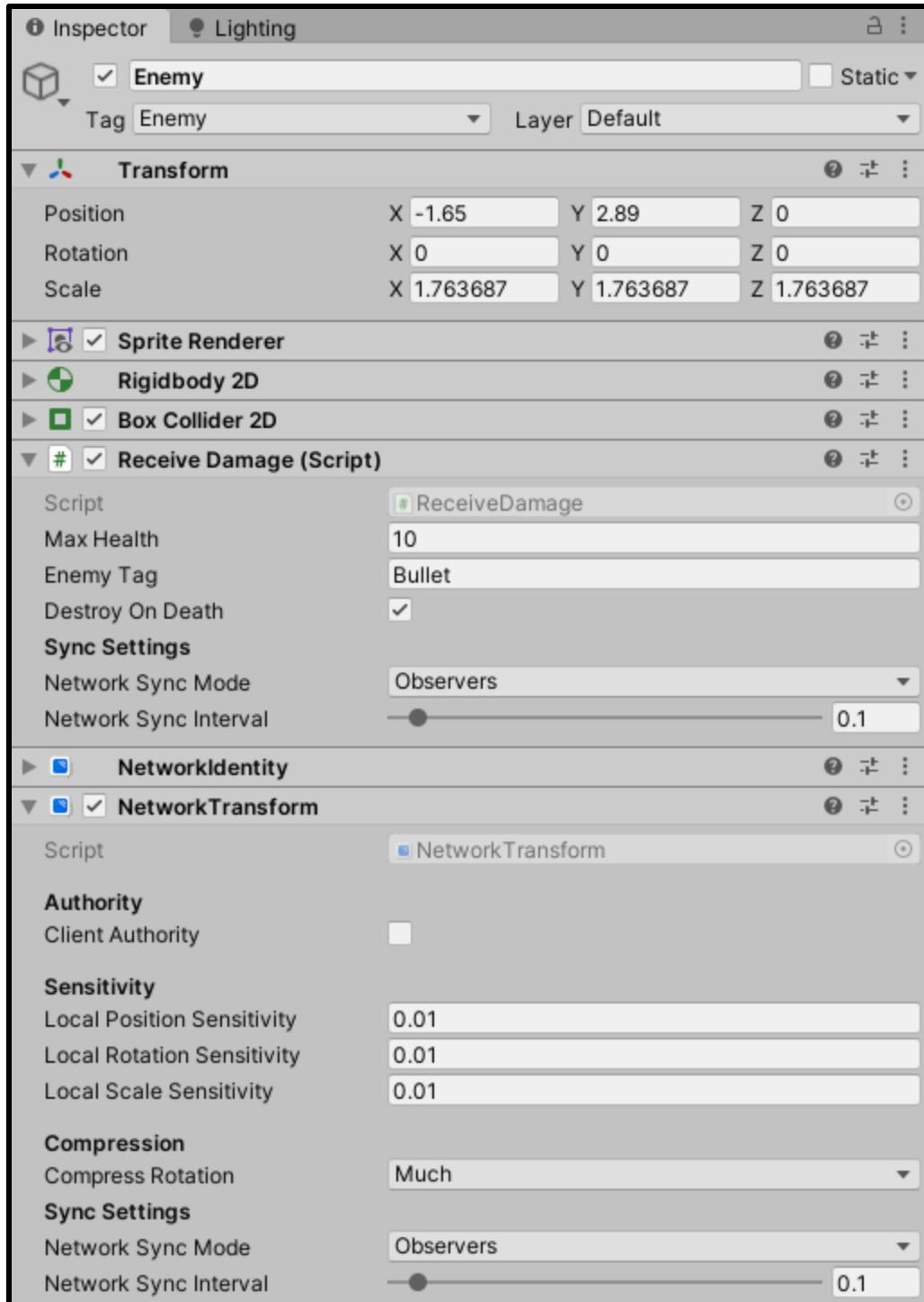
The last method is the respawn one. Here we are using another multiplayer feature called ClientRpc (observe the [ClientRpc] tag above the method definition). This is like the opposite of the [Command] tag. While commands are sent from clients to the server, a ClientRpc is executed in the client, even though the method was called from the server. So, when an object needs to be respawned, the server will send a request to the client to execute the RpcRespawn method (the method name must start with Rpc), which will simply reset the position to the initial one. This method must be executed in the client because we want it to be called for ships, and ships are controlled only by players (we set the Local Player Authority attribute as true in the NetworkIdentity component).

Finally, we need to add this script to both the Ship and Enemy prefabs. Notice that, for the ship we need to define the Enemy Tag as "Enemy", while for the Enemy this attribute value is "Bullet" (you also need to properly define the prefabs tags). Also, in the enemy prefab we are going to check the Destroy On Death attribute.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved



Now, you can try playing the game shooting enemies. Let some enemies hit your ships to see if they're being correctly respawned as well.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Conclusion

And this concludes this tutorial! While small, we now have a nifty multiplayer game to work with. However, you don't have to stop here! You consider improving this project by [adding sounds](#), or maybe even making a [procedurally generated map](#) for more endless amounts of fun. Either way, this project is sure to make a great addition to any [coding portfolio](#)!

We hope you've learned a lot here, and we wish you the best of luck with your future game projects!

How to Import Blender Models into Unity – Your One-Stop Guide

By Pablo Farias Navarro

In an ideal world, exporting models from Blender into Unity for your 3D / virtual reality games should be a seamless, simple process. To be more precise, it shouldn't require any thought or whatsoever.

At first it feels like that is the case. You drag and drop a .blend file into Unity and the model shows up! But as people quickly realize when they go through the process, the devil is in the details!

This guide was created to take all pain from the process of importing simple Blender models into Unity, so that at least you know what works for sure and why that is!

1. Coordinate differences

The first thing to keep in mind is that there are two main differences between the coordinate system of Unity and Blender.

1. Blender uses **right handed** coordinate system, whereas Unity uses a **left handed** coordinate system (see illustration below).
2. In Blender, the Z axis points upwards, whilst in Unity, the Y axis points upwards.

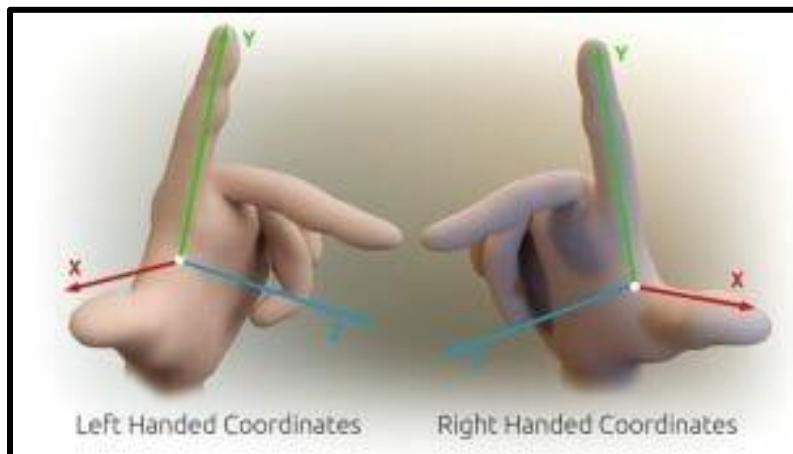
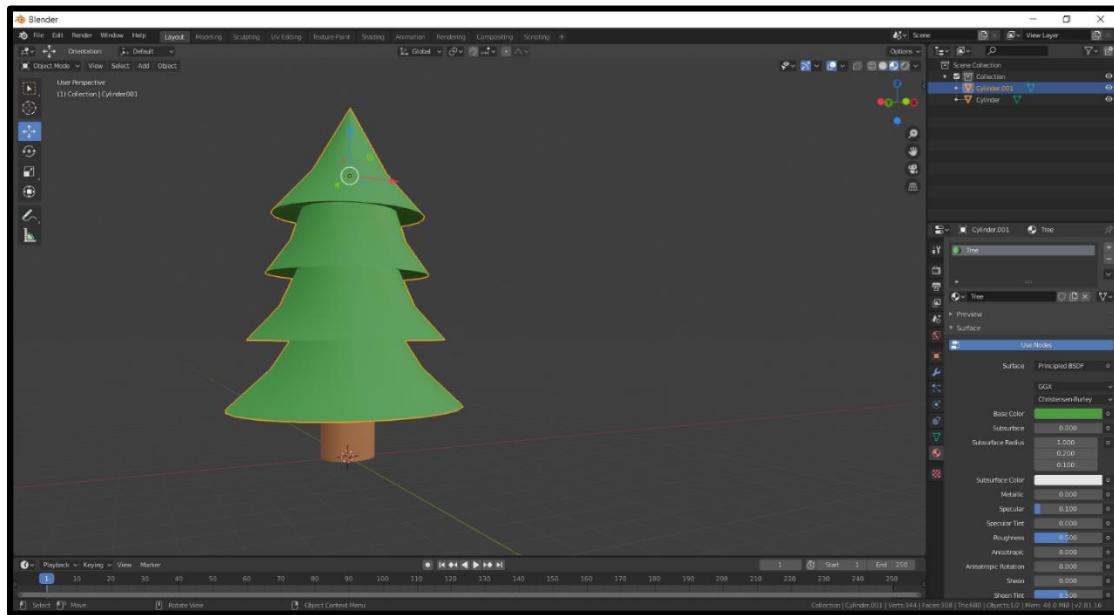


Image credit: [Primalshell](#), license [CC BY-SA 3.0](#)

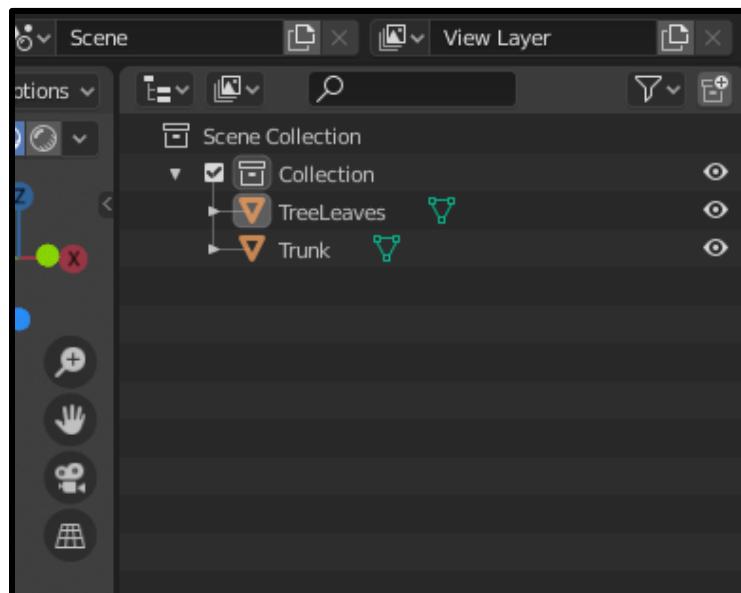
2. Blender model settings

Say you created a nice-looking low poly tree in Blender, like so:



Basic check list before attempting any import:

- **Delete the default Blender camera and lamp.** We are not using Blender to create a rendered scene, we just care about the model, so unless you are indeed doing other things with your Blender file, you can remove these elements. Make sure to name your model as well.

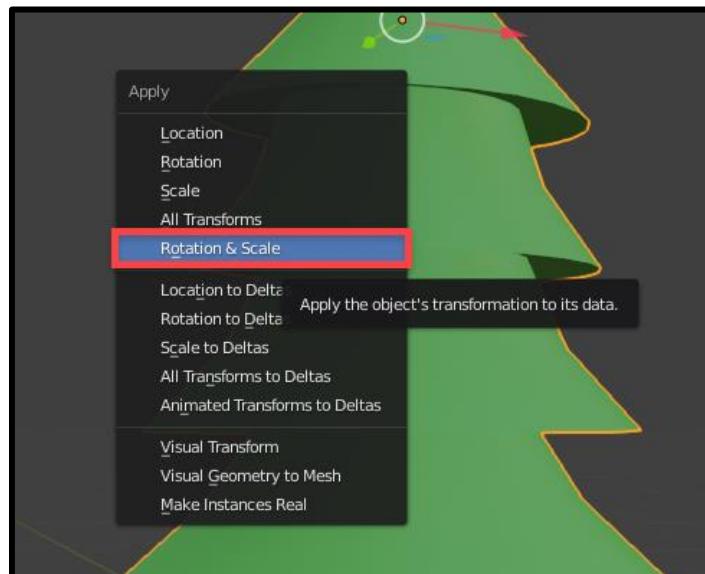
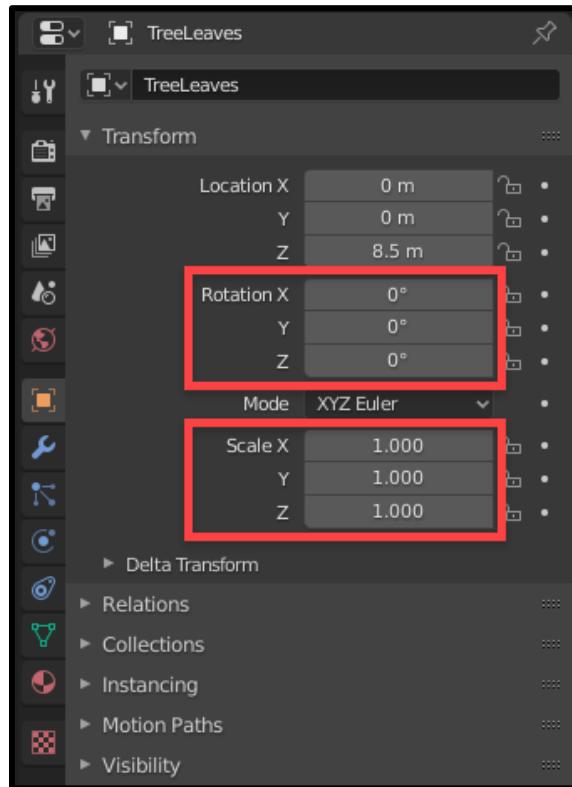


This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

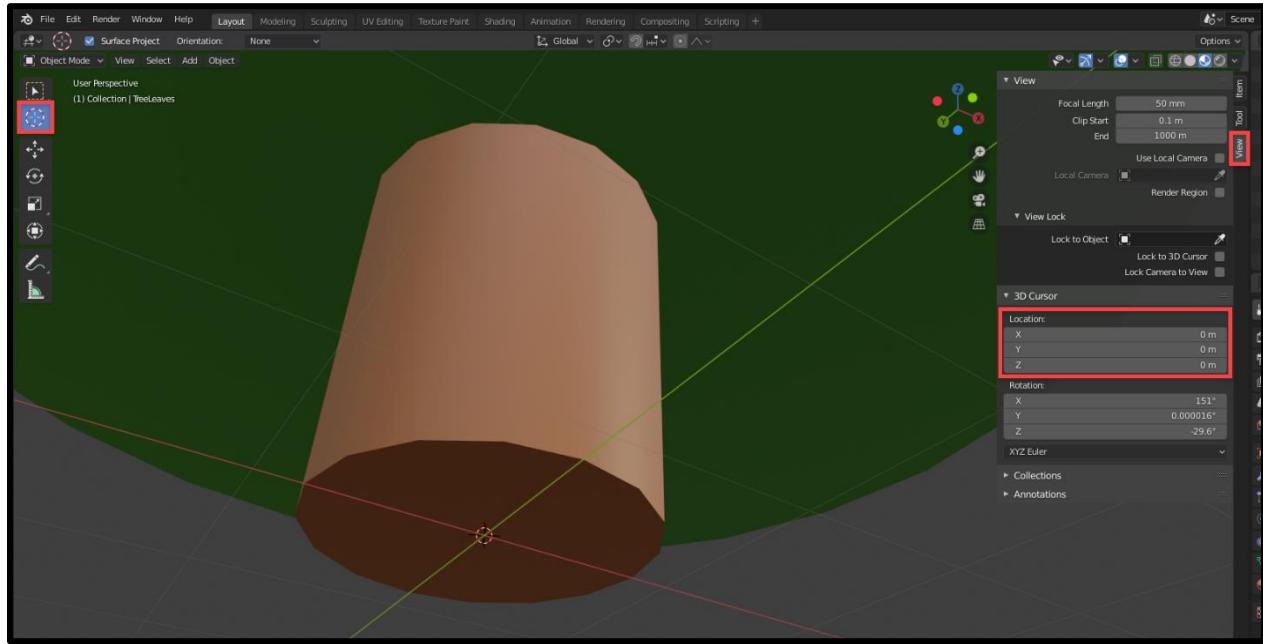
- **Apply transforms.** On the Transforms panel, the values for *Rotation* should show 0 and for *Scale* 1 on all the axes. Press *Control + A > Rotation & Scale*.

Why is this? In Unity you want to be able to apply transforms to your models. If they already come with weird numbers in there this will be confusing.

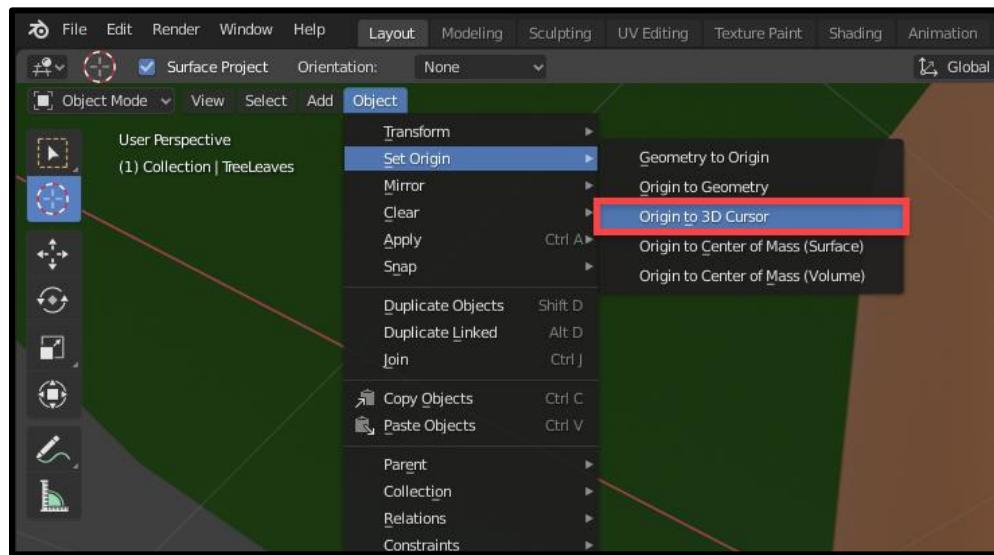


- **Set the origin to a meaningful position.** This might not apply to all cases, but for instance if in Unity you'll have the floor on Y = 0 (Y as in vertical coordinate), and your Blender model (in this case, a tree) will go on ground level as well, it'll be much easier if you set the origin in Blender to the base of your model.

To change the origin in Blender, select the *Cursor* tool and click where you want to place it. To be more precise, Open the *View* window and set the location of the 3D cursor manually.



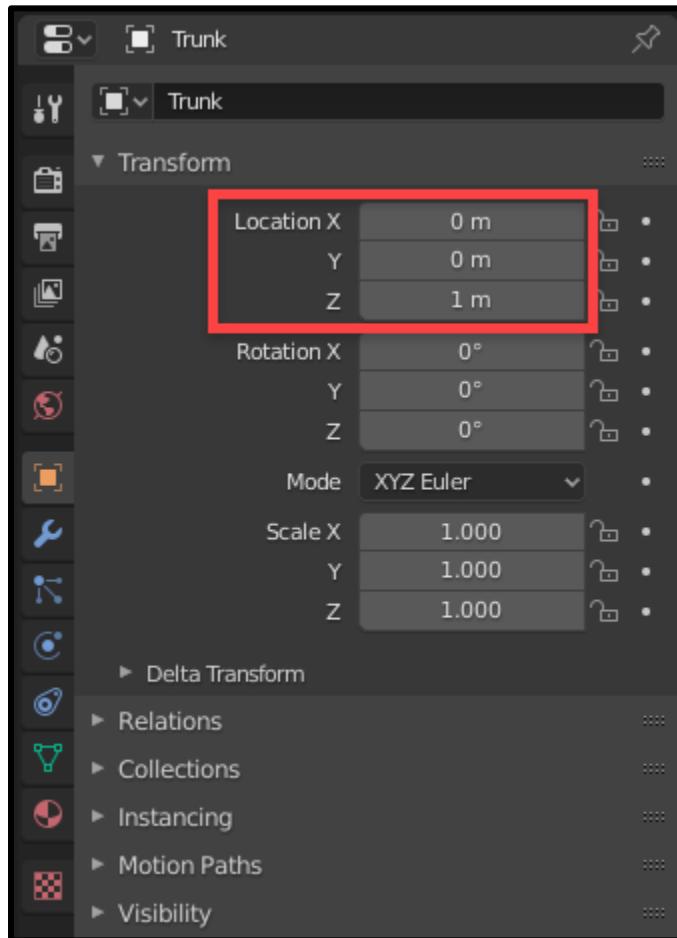
In order to set the origin of the model to the 3D cursor: select the model, then navigate to *Object* > *Set Origin* > *Origin to 3D Cursor*.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

Why is this? If you set the origin to say the middle of the model, when you import it to Unity and drag them to your terrain / floor, the middle of the model will be on Y = 0, so you might have to drag your model up each time.

- **Keep an eye at the location transform in Blender.** When you place your model in Unity you will most likely move it around, so you don't have to set the location transform in Blender to 0 as it won't do much. However, if you have a really high number in there, the model will show far away in Unity as well, so keep an eye there and setting it to 0 won't hurt.

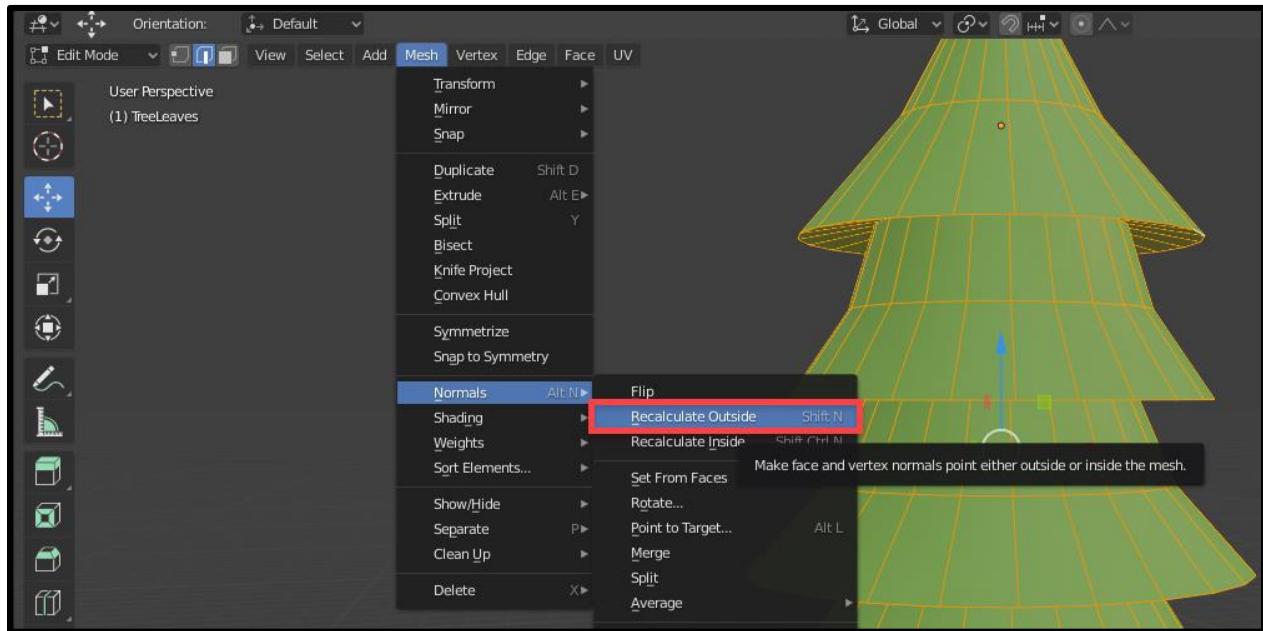


- **Make normals point outwards.** Sometimes when importing a model into Unity, it appears as some faces are invisible. This is caused by the normals of those faces pointing inwards. If you are experiencing this issue try the following:

What is a normal? In a mesh, each face has a single vector which is perpendicular to the face. This vector is called a normal and it points to only one side of the face. Normals are used for rendering.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

In your model, all normals should always point **outwards**. To do that, go to Edit Mode, select all faces, go to *Mesh > Normals > Recalculate Outside*.



What if I need to show both sides of a plane? Like a wall that separates two rooms for instance. In those cases you should use a cube mesh shape for instance, or duplicate the face (you can extrude the face and move it slightly to the other side), you should **never** have a single face that will be looked at from both sides.

What about backface culling? Unity only supports one-side rendering per face, so enabling/disabling backface culling in Blender won't make a difference. Backface culling is when you explicitly tell your program that only one side of a face should be rendered.

3. Importing .blend files vs importing .fbx files

You have two main options to import a Blender file into Unity. There is no correct answer here, as different workflows and cases might find one approach better than the other one:

1. Importing the .blend file directly into Unity
2. Exporting a .fbx file from Blender, then importing this file into Unity

When you import a .blend file into Unity, what really happens behind the scenes is that Unity will call Blender's export scripts to generate a .fbx file, and then import this file into Unity.

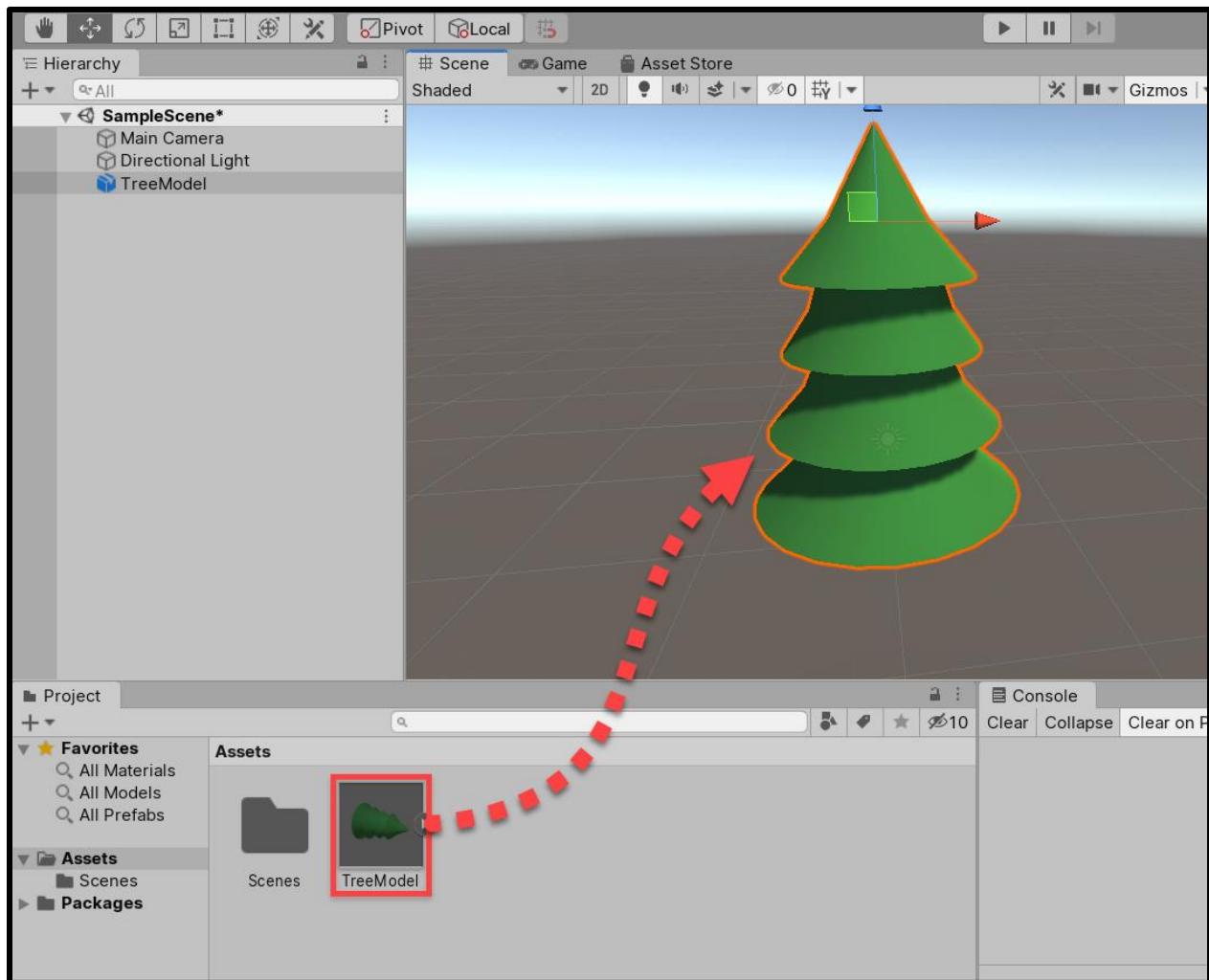
At first that sounds like both paths should be 100% equivalent. However, that is not the case!

3.1. Importing a blend file into Unity

Simply drag and drop the blend file into a location inside your Assets folder in your Unity project. You can place the file there by many difference ways:

- Put it there using the File Explorer
- Drag and drop it into your Project tab
- Save your Blender project in there

After that, simply drag and drop the asset into your Scene. The file to grab is the light blue box.



If you click on your newly created object and take a look at the transform panel, you will see that the scale is 1, and a rotation of "almost" -90 applied on X.

The X axis will be reversed from what you had in Blender. Whatever value of X was positive in Blender, will be negative in Unity. Usually this doesn't matter, as the model looks exactly the same. Once the model is in Unity you can move and rotate at glance.

If you want the X coordinate to match, in Unity you have to apply a rotation of 180 degrees on Y. In this case, the Z axis still won't match given the different coordinate systems used in each program (the only way for all axes to match would be to "mirror" the model, for which you can easily give it a scale of to -1 in either X or Z in Unity).

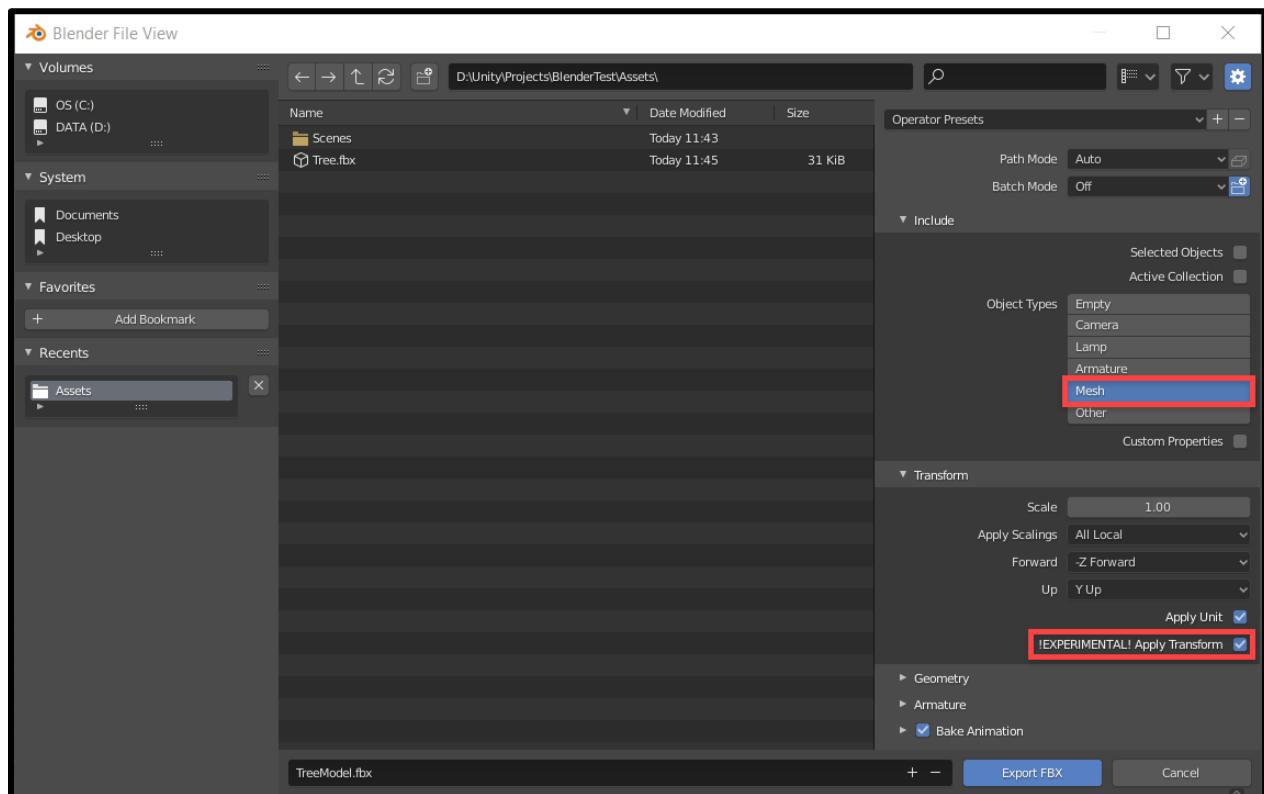
3.2. Importing a fbx file into Unity

To export an fbx file, go to *File > Export > FBX (.fbx)*.

To make sure that we're not exporting anything we don't need (camera, lights, etc), only select **Mesh** in the Object Types. If you have a rig for a character model, make sure to also select **Armature** (select multiple by holding down *Shift*).

Since Unity and Blender have different coordinate systems (Z and Y are flipped), we need to enable **Apply Transform** to make the transform coordinates work for Unity.

Now we can click the **Export** button to export the model to our desired location.



This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

Note: if you are exporting the whole scene, clicking that button won't make a difference, but if you are exporting only *selected objects*, not checking this option will cause them to be imported into Unity with a scale of 0.01, instead of 1.

To import the fbx file into Unity, simply place in the Assets folder and drag into your scene just like we've described for blend files.

The model will show just fine, but the Inspector panel will show a *File Scale* of 0.9999999 which is almost 1. We can deal with that.

If you drag the mesh into the scene and check the Inspector panel, you will notice that the scale is set to 1, that means we are on the right track.

Regarding the horizontal plane coordinates, if you want the X direction to match that of Blender, you have to apply a rotation of 180 degrees in Y in Unity, like like we did for blend files!

4. blend vs fbx? Which one wins?

There is simply no correct answer to that question, as both options have their pros and cons and that will depend also on your own workflow. The important thing is that by following this tutorial **you can now import your Blender assets correctly no matter your choice**.

Which approach should I take? Perhaps these guidelines can be of help:

- If you have multiple models in a single Blender file, you can easily export one by one as fbx files as described in the tutorial.
- If you have a single Blender file per model, it's easy to just move these files into Unity, or even create them inside the Assets folder to begin with!
- **Important.** After importing a blend file into Unity, if you modify and save that file in Blender, the objects in Unity that use this asset will be updated too. This can help you save time, as you don't have to export again. However, it can also mean you have to be careful if you are making changes in Blender, as you can easily mess up your game objects by some unintentional transforms and location changes.
- On that sense, exporting fbx files makes this process manual, so even if you save changes in your Blender file, these won't propagate into Unity unless you choose to.
- Importing blend files requires Blender to be installed. Since Blender is a free program this is not a big deal, but it's good to know.

Thanks for reading and I hope you found this tutorial helpful!

Some useful resources and discussions on this topic:

- <http://answers.unity3d.com/questions/439807/blender-to-unity-which-export-format-is-best.html>
- <http://answers.unity3d.com/questions/36145/scaling-between-fbx-and-unity.html>

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

- <https://docs.unity3d.com/Manual/HOWTO-ImportObjectBlender.html>
- https://www.reddit.com/r/Unity3D/comments/32lhok/unity_5s_scale_is_completely_messed_up/

You might also want to check out our [developer job prospects](#) article or our [coding portfolio article](#) to help make the most of your projects!

Intro to Shaders with Unity3D

By Jesse Glover

Today's focus will be on shaders. You may be curious as to why we went from talking about the Editor, how to write code in Unity, and building a game to Shaders.

The answer is simple, shaders are a very important aspect of the graphics pipeline. It allows you to customize how the landscape and characters are portrayed in the game.

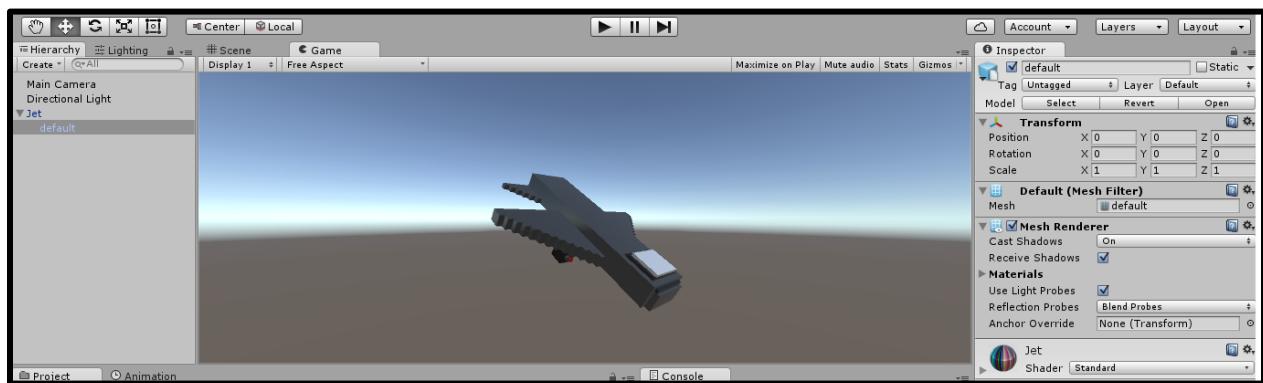
You can get the 3D model used from Adobe Mixamo, his name is Knight_D_Pelegrini.

The 2D sprite was obtained from the Temple Run pack available on Opengameart.org.

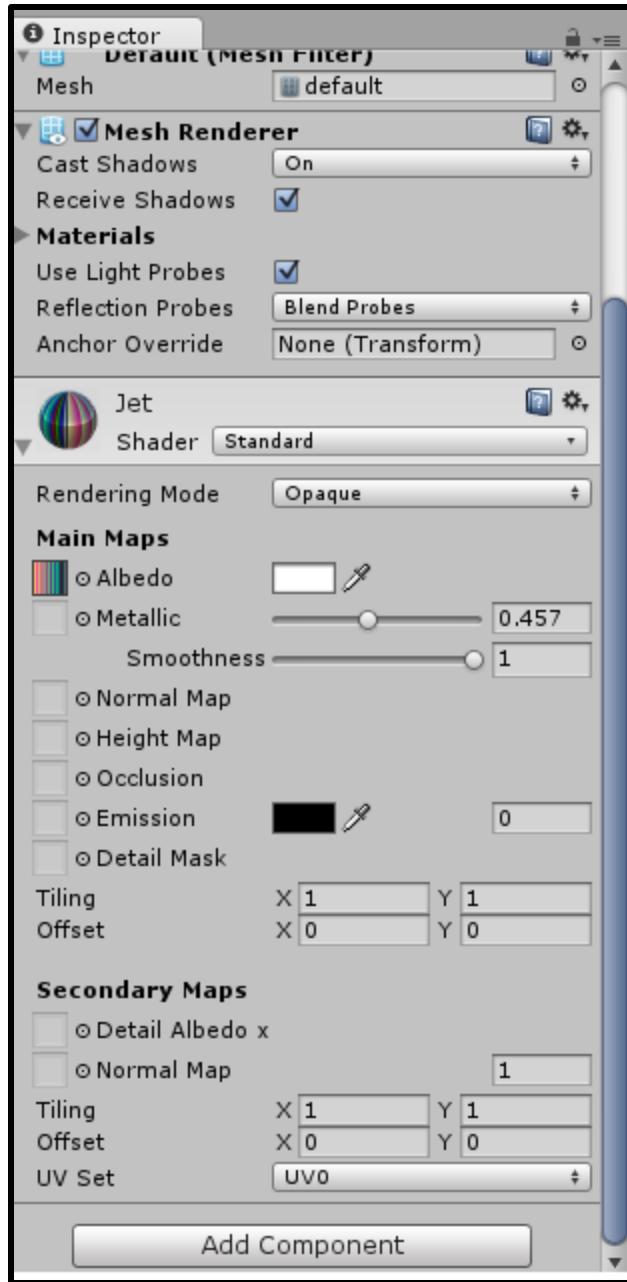
Download the project zip file [here](#).

Intro to Shaders

To give an example of what I mean, you can import any 3D model into Unity. Let's take a look at the Jet from the game we built.



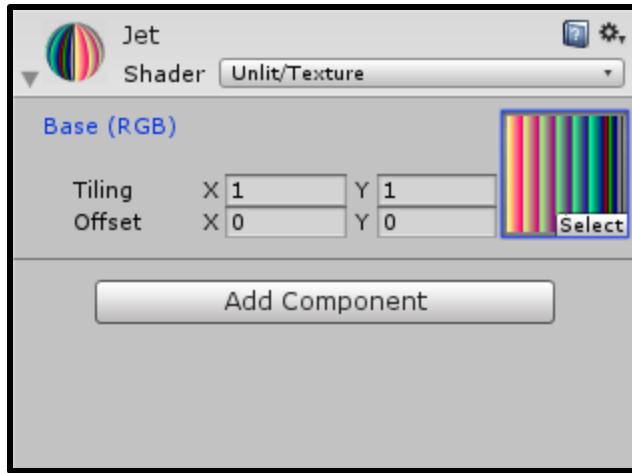
As you can see from the image above, we used the same jet as before and it looks the same. Let's look a bit deeper and look at the inspector pane. You can see underneath where it says Materials that we are allowing the use of light probes and using a blend probe for reflections. If we look just under that, it is where the shader is.



We can see that we are using a standard shader and it gives us many options that we can modify with it. We have rendering mode, main maps for albedo, metallic, smoothness, normal maps, height maps, occlusion, emission, detail task, tiling, offset. Secondary maps which has Detail albedo x, normal map, tiling, offset, and UV Set.

Each type of shader available to us allows us to have different properties that we can manipulate to fit our needs. To illustrate this, let's change from the Standard shader to the Unlit/Texture shader. To do this, click on the shader box, highlight unlit, and then select Texture.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity



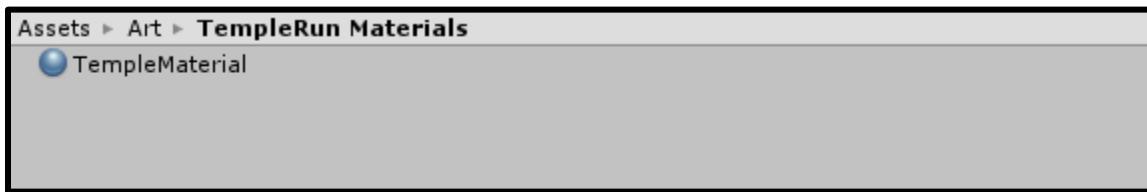
As we can see, the only properties we have with this one is selecting which texture to use for the object along with tiling and offset. We can also see that the way the Jet is rendered is very different from before with the standard shader.

Let's take a look at some other shaders to get a better understanding of how they work.

A deeper look into shaders

Shaders come in a wide variety of flavors right out of the box with Unity and it does an excellent job of handling most of your day to day game development needs. So, to illustrate what I mean; We will look at a 2D sprite and 3D model for how the shaders affect each one.

To begin, let's set up a 2D and 3D scene for this. In the 2D scene, change the camera from 3D to 2D and the camera rendering mode to be Orthographic. Next up, create a new material by right clicking on the folder should create called Materials, select Materials. Name that material Temple Material.



Now let's take a look at what the Unity documentation states about what each shader type is meant to accomplish.

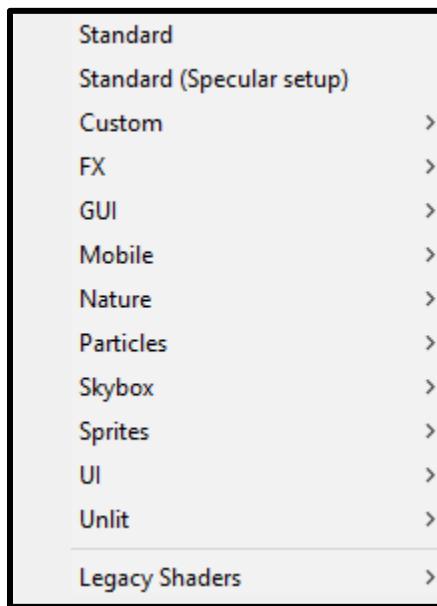
In addition to the Standard Shader, there are a number of other categories of built-in shaders for specialized purposes:

- **FX:** Lighting and glass effects.

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

- **GUI** and **UI**: For user interface graphics.
- **Mobile**: Simplified high-performance shader for mobile devices.
- **Nature**: For trees and terrain.
- **Particles**: Particle system effects.
- **Skybox**: For rendering background environments behind all geometry
- **Sprites**: For use with the 2D sprite system
- **Toon**: Cartoon-style rendering.
- **Unlit**: For rendering that entirely bypasses all light & shadowing
- **Legacy**: The large collection of older shaders which were superseded by the Standard Shader

We can see that each type of shader has a specific purpose, and sadly Toon is not part of the built in shaders as of the latest version of Unity presently. So, to avoid confusion, below is a screenshot of all the shaders present in Unity.



(Note: We can omit Custom from the built in shaders list because those are shaders I have built that we will delve into later.)

Most of the shaders that are typically used are the Standard Shader, Mobile Shaders, and Sprite shaders. The standard shader is built to give a fairly realistic view of the model although you will have no proper transparency effect on 2D sprites. Same pretty much goes for the Mobile Shaders which are your typical diffuse and bumped diffuse, specular mapping, and a particle renderer; These shaders are optimized for mobile usage although can be used on PC. Lastly, we have the sprite shaders. They have a diffuse and default option. These shaders are optimized for 2D sprites if you so choose to use a shader for 2D art.

Standard Shader

Let's view how the standard shader works on both 2D and 3D models.



As we can see, the 3D model showcases all of the folds and lighting to the model appropriately with very nice shading. It also looks fairly realistic in terms of not looking extremely cartoony.



The 2D sprite, however, looks appropriate except for what looks like transparency bleeding through. We see a very unnatural border around the sprite and we have no way to remove it.

Mobile Bumped Diffuse Shader

Next up, is the mobile bumped diffuse shader.



The mobile bumped diffuse shader on a 3D model looks almost exactly like the standard shader and that is exactly how it was designed. It was modeled after the standard shader but optimized for mobile devices.



Again, we see the same results for the 2D sprite as with the standard shader.

Mobile bumped with specular mapping.



This shader adds some very shiny lighting to the model. The higher you have the shininess property, the more it seems to distort the face of the model.



With the 2D sprite, it still has the border around the sprite but the sprite also has some very nice shininess to it.

Sprites Default shader



Now we see some unexpected results for the 3D model. It seems to render the 3D model as a 2D sprite and blended the face to be part of the hood. The coloring has also changed to be a vibrant version of the colors instead of the darker ones we are used to.



Now the 2D sprite no longer has that nasty border around it and has rendered appropriately.

Sprites Diffuse shader.



It again renders the 3D model as a 2D sprite and blended the face to be part of the hood. The coloring has also changed to be the darker colors we are more used to.



The sprite again renders appropriately, and the color scheme is darker than the default version.

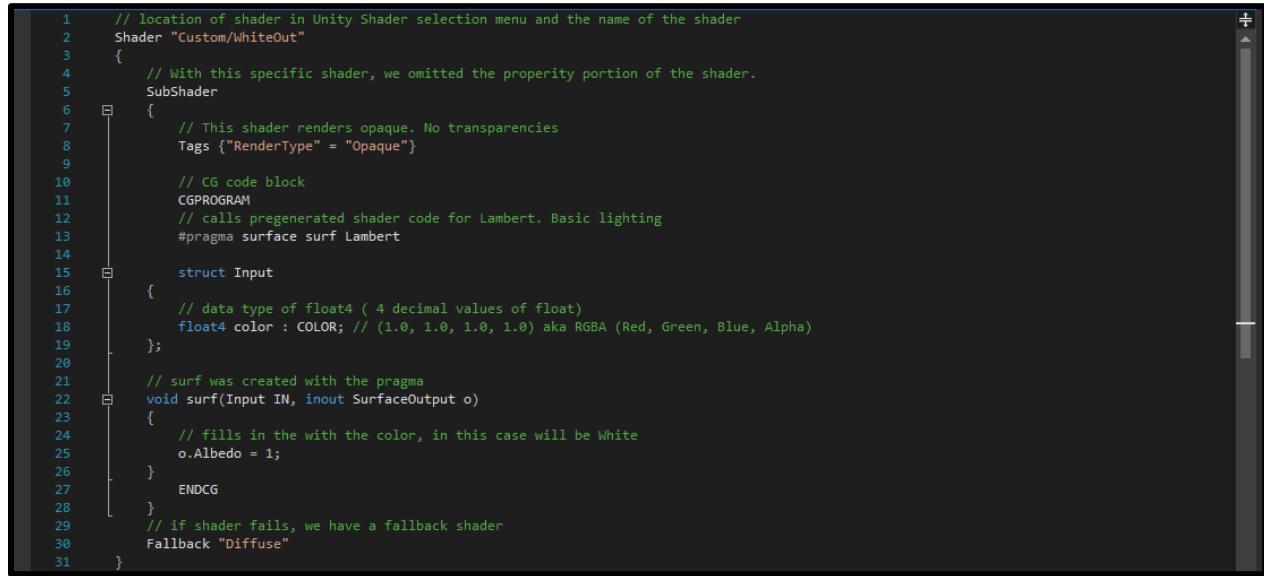
Building our own Shader

Why would we want to build our own shader? Let's take for example that we want a very specific outcome for how the character or model should be displayed and none of the default shaders will produce the results we want. We would need to roll our own shader to produce those results. To get into the mindset for building shaders, let's build the most basic shader

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

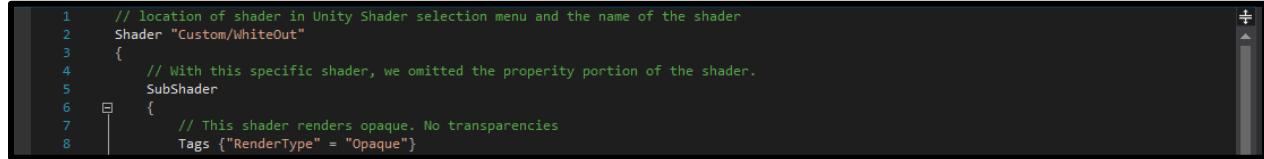
© Zenva Pty Ltd 2021. All rights reserved

there is, Albedo mapping or as I like to call it, White Out. The code has been commented to give a brief overview of how it is set up and we will go more in depth after this to understand what is going on.



```
1 // location of shader in Unity Shader selection menu and the name of the shader
2 Shader "Custom/WhiteOut"
3 {
4     // With this specific shader, we omitted the property portion of the shader.
5     SubShader
6     {
7         // This shader renders opaque. No transparencies
8         Tags {"RenderType" = "Opaque"}
9
10        // CG code block
11        CGPROGRAM
12        // calls pregenerated shader code for Lambert. Basic lighting
13        #pragma surface surf Lambert
14
15        struct Input
16        {
17            // data type of float4 ( 4 decimal values of float)
18            float4 color : COLOR; // (1.0, 1.0, 1.0, 1.0) aka RGBA (Red, Green, Blue, Alpha)
19        };
20
21        // surf was created with the pragma
22        void surf(Input IN, inout SurfaceOutput o)
23        {
24            // fills in the with the color, in this case will be White
25            o.Albedo = 1;
26        }
27        ENDCG
28    }
29    // if shader fails, we have a fallback shader
30    Fallback "Diffuse"
31 }
```

We won't be going in depth with the shader syntax in this portion of the tutorial, don't worry, that's next. For now, we will talk about how this specific shader works.



```
1 // location of shader in Unity Shader selection menu and the name of the shader
2 Shader "Custom/WhiteOut"
3 {
4     // With this specific shader, we omitted the property portion of the shader.
5     SubShader
6     {
7         // This shader renders opaque. No transparencies
8         Tags {"RenderType" = "Opaque"}
```

First part of the code is Specifying where we want our shader to live inside the Unity editor. Which is all that line 2 is doing. Line 5 which only has the word Sub Shader is where the rest of our code lives. The Tag specifies how we want our object to be rendered.

CGProgram at line 11 designates that this is the portion of the code that was built by NVIDIA. The pragma directive states we want to use a specific portion of shader code prebuilt for us and allows us to write a method over it to give more control over what it does.

The struct allows us to put values inside of it that we can manipulate in the method as well. Finally, we get to the method it should have the name surf because of the pragma directive. We have two parameters for the method, Input and inout or what we want to output.

We don't need to call the input because it is inferred with what we want to do. We do call output and set it to Albedo which is an NVIDIA keyword and we set it to the float value of 1. This will fill in the color which in this case is white.

ENDCG is the designation of that we are ending the CGPROGRAM code. Then we set a fallback which states that if the code cannot be run on a specific piece of hardware that it should go ahead and use the shader named instead.

```
9      // CG code block
10     CGPROGRAM
11     // calls pregenerated shader code for Lambert. Basic lighting
12     #pragma surface surf Lambert
13
14     struct Input
15     {
16         // data type of float4 ( 4 decimal values of float)
17         float4 color : COLOR; // (1.0, 1.0, 1.0, 1.0) aka RGBA (Red, Green, Blue, Alpha)
18     };
19
20     // surf was created with the pragma
21     void surf(Input IN, inout SurfaceOutput o)
22     {
23         // fills in the with the color, in this case will be White
24         o.Albedo = 1;
25     }
26     ENDCG
27 }
28
29 // if shader fails, we have a fallback shader
30 Fallback "Diffuse"
31 }
```

Breaking down Shader Syntax:

The shader syntax is relatively simple to remember and I will do my best to explain it. Just remember, Unity has excellent documentation if you can find it that goes over every value type that a shader can have as well as how you could manipulate a shader in C# code in a limited fashion.

Another thing I feel that I should point out, if you haven't noticed already, is that there is no code completion for writing Shader files. You can find a very simple one in the Visual Studio Gallery, although I can guarantee that you won't find it that helpful.

```
1 Shader "MyShaderName" {
2     Properties {
3         // ... properties here ...
4     }
5     SubShader {
6         // ... subshader for graphics hardware A ...
7         // ... You only need one SubShader, but you can have more if needed
8         Pass {
9             // ... pass commands ...
10        }
11        // ... more passes if needed ...
12    }
13    SubShader {
14        // ... subshader for graphics hardware B ...
15    }
16    // ... Optional fallback ...
17    // FallBack "Fallback to another shader if needed"
```

```
1 Shader "List of Commonly used settings" {
2
3     Properties {
4
5         _Color ("Main Color", Color) = (1,1,1,0.5)
6
7         _SpecColor ("Spec Color", Color) = (1,1,1,1)
8
9         _Emission ("Emmisiive Color", Color) = (0,0,0,0)
10
11        _Shininess ("Shininess", Range (0.01, 1)) = 0.7
12
13        _MainTex ("Base (RGB)", 2D) = "white" { }
14
15    }
16
17    SubShader {
18        Pass {
19            Material {
20                Diffuse [_Color]
21                Ambient [_Color]
22                Shininess [_Shininess]
23                Specular [_SpecColor]
24                Emission [_Emission]
25            }
26        }
27    }
28 }
```

This book is brought to you by Zenva - Enroll in our [Unity Game Development Mini-Degree](#) to explore even more 2D and 3D game development skills with Unity

© Zenva Pty Ltd 2021. All rights reserved

```

26 Lighting On
27 SeparateSpecular On
28 SetTexture [_MainTex] {
29     constantColor [_Color]
30     Combine texture * primary DOUBLE, texture * constant
31 }
32 sampler2D _MainTex;
33 sampler2D _BumpMap;
34 struct v2f {
35     float4 pos : SV_POSITION;
36     fixed3 color : COLOR0;
37 };
38 struct SurfaceOutput {
39     fixed3 Albedo; // diffuse color
40     fixed3 Normal; // tangent space normal, if written
41     fixed3 Emission;
42     half Specular; // specular power in 0..1 range
43     fixed Gloss; // specular intensity
44     fixed Alpha; // alpha for transparencies
45 };

```

```

46 struct SurfaceOutputStandardSpecular {
47     fixed3 Albedo; // diffuse color
48     fixed3 Specular; // specular color
49     fixed3 Normal; // tangent space normal, if written
50     half3 Emission;
51     half Smoothness; // 0=rough, 1=smooth
52     half Occlusion; // occlusion (default 1)
53     fixed Alpha; // alpha for transparencies
54 };
55 void surf (Input IN, inout SurfaceOutput o) {
56     o.Albedo = 1;
57     o.Alpha = 1;
58     o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
59 }
60 v2f vert (appdata_base v) {
61     v2f o;
62     o.pos = mul (UNITY_MATRIX_MVP, v.vertex);
63     o.color = v.normal * 0.5 + 0.5;
64     return o;
65 } +
66 fixed4 frag (v2f i) : SV_Target {
67     return fixed4 (i.color, 1);
68 }
69

```

```

70      ENDCG
71
72  }
73
74  Fallback "Diffuse"
75
76  }

```

To break this down, let's talk about what each item is doing:

- `_Color` = sets the color of the object
- `_SpecColor` = Is the specularity color of the object
- `_Emission` = the emissive color of the object
- `_Shininess` = How much light is reflected off the object
- `_MainTex` = The texture being manipulated
- `Lighting` = is an option that you can have on or off
- `SeparateSpecular` = an option that you can have on or off
- `Sampler2D` = how you would access a texture in your method
- `fixed3 Albedo` = diffuse color
- `fixed3 Normal` = tangent space normal, if written
- `fixed3 Emission` = the emissive color of the object
- `half Specular` = specular power in 0..1 range
- `fixed Gloss` = specular intensity
- `fixed Alpha` = alpha for transparencies
- `half Smoothness` = 0=rough, 1=smooth
- `half Occlusion` = occlusion (default 1)
- `o.Albedo = 1 or tex2D (_MainTex, IN.uv_MainTex);`
- `o.Alpha = 1 or tex2D (_MainTex, IN.uv_MainTex);`
- `o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));`
- `float2 = Uv direction of the object`
- `float3 = r, g, b values`
- `float4 = r, g, b, a values`
- `fixed2 = Uv direction of the object`
- `fixed3 = r, g, b values`
- `fixed4 = r, g, b, a values`

This is not by any stretch of the imagination the complete list of variables available with Shaders. There are many more and are specific to certain actions within a constructed shader. The next tutorial, we will go into more depth and build more shaders to handle more specific tasks. Until next time, this is Jesse and may your code be bug free.