



Mobile VR Game Development with Unity for Human Beings

By Shang-Lin Chen

Software Developer and Technical Writer

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

© Zenva Pty Ltd 2018. 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

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

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

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

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

Table of Contents

[GearVR Game Development: How to Create a VR Game in Unity](#)

[Unity Project and Source Code](#)

[Hardware Requirements](#)

[Setting up the Software](#)

[Your First Project](#)

[Developer Mode](#)

[Adding Color](#)

[Adding Interaction](#)

[How to Create a Virtual Reality Tour in Unity for the GearVR](#)

[Source Code Files](#)

[Creating the Project](#)

[Importing Models](#)

[Setting the Scene](#)

[Creating the Player](#)

[Adding a Reticule](#)

[Adding Waypoints](#)

[Clickable Buttons](#)

[Prefabs](#)

[Adding Scenes](#)

[Building the Project](#)

[Developing for the Gear VR Controller](#)

[Source Code Files](#)

[Setting Up the Project](#)

[Using OVRCameraRig](#)

[Understanding GearVrController](#)

[Creating a Gun](#)

[Adding Enemies](#)

[Flight Simulation with the Gear VR Controller](#)

[Source Code Files](#)

[Setting Up the Project](#)

[ShipController](#)

[Setting Up the Game Assets](#)

[Finding the Controller](#)

[Rotation](#)

[Conclusion](#)

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

GearVR Game Development: How to Create a VR Game in Unity

Several years ago, virtual reality for the average consumer was still the realm of science fiction. This changed with the release of Samsung's Gear VR in 2015. Now, if you own a Samsung Galaxy or Note smartphone, you can enjoy virtual reality games and experiences with a \$50-\$100 Gear VR headset, powered by your phone. Take it a step further, and you can even start to build your own virtual reality worlds.

In this tutorial, I will show you how to get started with Gear VR development using the Unity 3D platform. 90% of Gear VR games are made with Unity. Since there are already many Unity 3D tutorials available, including the ones on Game Dev Academy, I will not go into great detail about Unity. However, I will include some basics for those of you who are just starting your game development adventures.

Unity Project and Source Code

Download [here](#).

Hardware Requirements

To follow along, you will need the following hardware:

- A Samsung Galaxy Note 5, S6, S6 Edge, S6 Edge+, S7, S7 Edge, or Note 7 smartphone.
- A Samsung Gear VR headset.
- A PC running Windows 7 or later, or a Mac running OS X 10.8+.

Although many tutorials focus on Windows, I used a Macbook Pro to develop the code for this tutorial.

For this tutorial, you will not need a controller. If you want to get the most out of some existing VR games, you probably want to get one, anyway.

Setting up the Software

On your computer, you will need to install the Java Development Kit (JDK), the Android SDK, Unity 5, and the Oculus runtime.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

Go to <http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html> to download the latest JDK for your platform. The Android SDK is available at <https://developer.android.com/studio/index.html#downloads>. This download page prominently features Android Studio, the official Android IDE, which bundles the Android SDK. If you only want the SDK, you can find the stand-alone SDK downloads at the bottom of the page. The SDK alone is sufficient for this tutorial.

Get just the command line tools

If you do not need Android Studio, you can download the basic Android command line tools below. You can use the included [sdkmanager](#) to download other SDK packages.

These tools are included in Android Studio.

Platform	SDK tools package	Size	SHA-256 checksum
Windows	tools_r25.2.3-windows.zip	292 MB (306,745,639 bytes)	23d5686ffe489e5a1af95253b153ce9d6f933e5dbabe14c494631234697a0e08
Mac	tools_r25.2.3-macosx.zip	191 MB (200,496,727 bytes)	593544d4ca7ab162705d0032fb0c0c88e75bd0f42412d09a1e8daa3394681dc6
Linux	tools_r25.2.3-linux.zip	264 MB (277,861,433 bytes)	1b35bcb94e9a686dff6460c8bca903aa0281c6696001067f34ec00093145b560

See the [SDK tools release notes](#).

Once you've installed the SDK, test that you can run the adb command. You might need to add the directory where you installed the SDK to your system path. This command will be useful in the next few steps for getting your device ID.

Unity can be downloaded from <https://unity3d.com/get-unity/download>. I am using Unity 5.5.2f1 Personal, the latest free edition available at the time of writing.

The Oculus runtime for Mac is available at <https://developer.oculus.com/downloads/package/oculus-runtime-for-os-x/0.5.0.1-beta/>

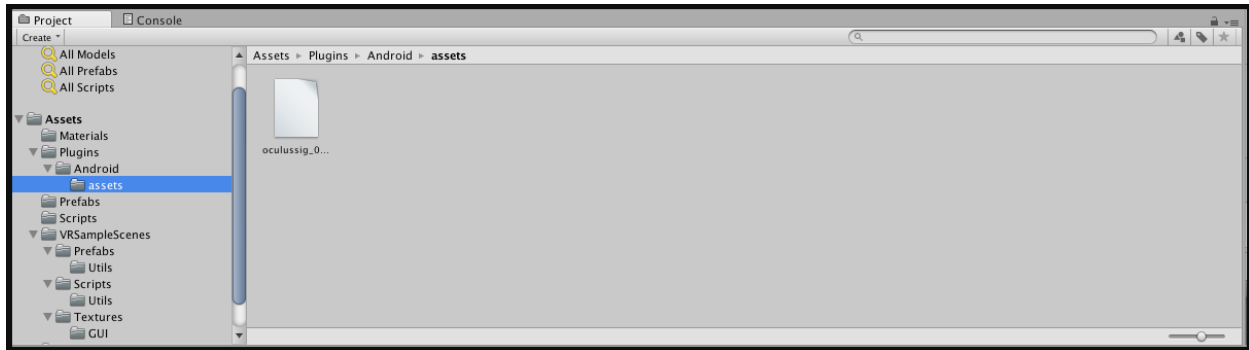
The Windows runtime is available at <https://developer.oculus.com/downloads/package/oculus-runtime-for-windows/0.8.0.0-beta/>

On your phone, you will need to follow Samsung's instructions for installing the Oculus Home app, which you should have done when setting up your Gear VR. Oculus Home is not available in the Google Play store — instead, you install it when you use your Gear VR for the first time. This process will also create an Oculus account.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

You will also need to enable developer mode on your phone. On your phone, touch the Settings icon. Scroll down and select “About Device.” If you do not see “Developer options,” tap seven times on “Build number.”

Finally, you will need to generate an Oculus Signature File, or osig. The osig file allows your application to use low-level Oculus functions. Follow the instructions on <https://dashboard.oculus.com/tools/osig-generator/> to find yo



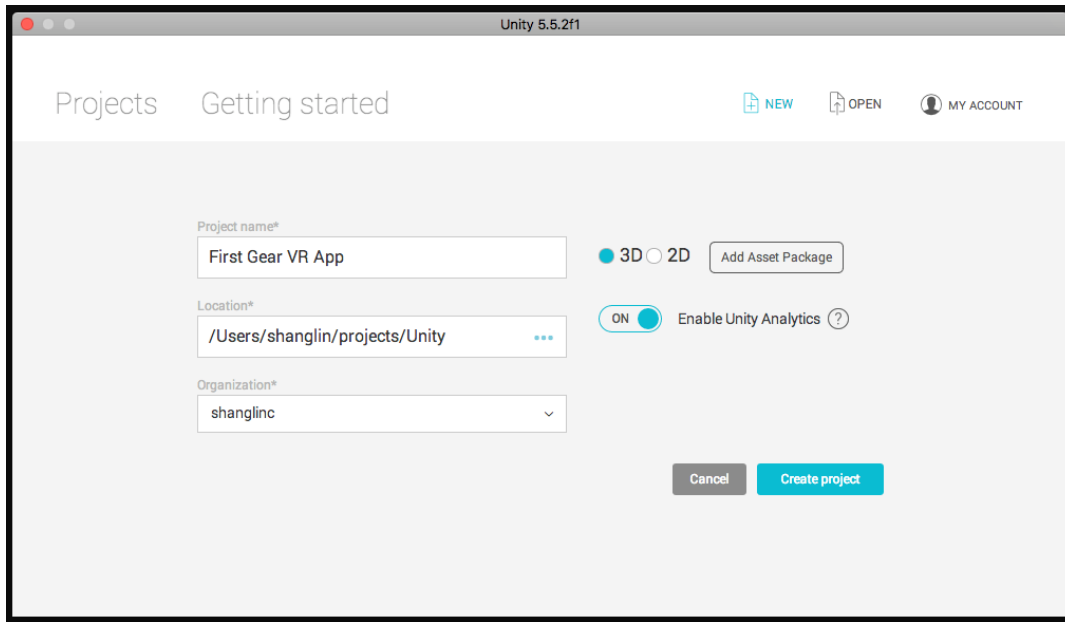
ur device ID and generate a file. Download the resulting file, and save it in a safe place. When you start a VR project, you will need to copy this file to the Assets/Plugins/Android/assets folder in your project.

Your First Project

Let's get started on a simple VR project.

First, create a new project. VR projects are created the same as regular projects. Open Unity, click “New”, and fill in a project name and directory where your project directory will be created. Make sure you select “3D.” Then click the “Create project” button.

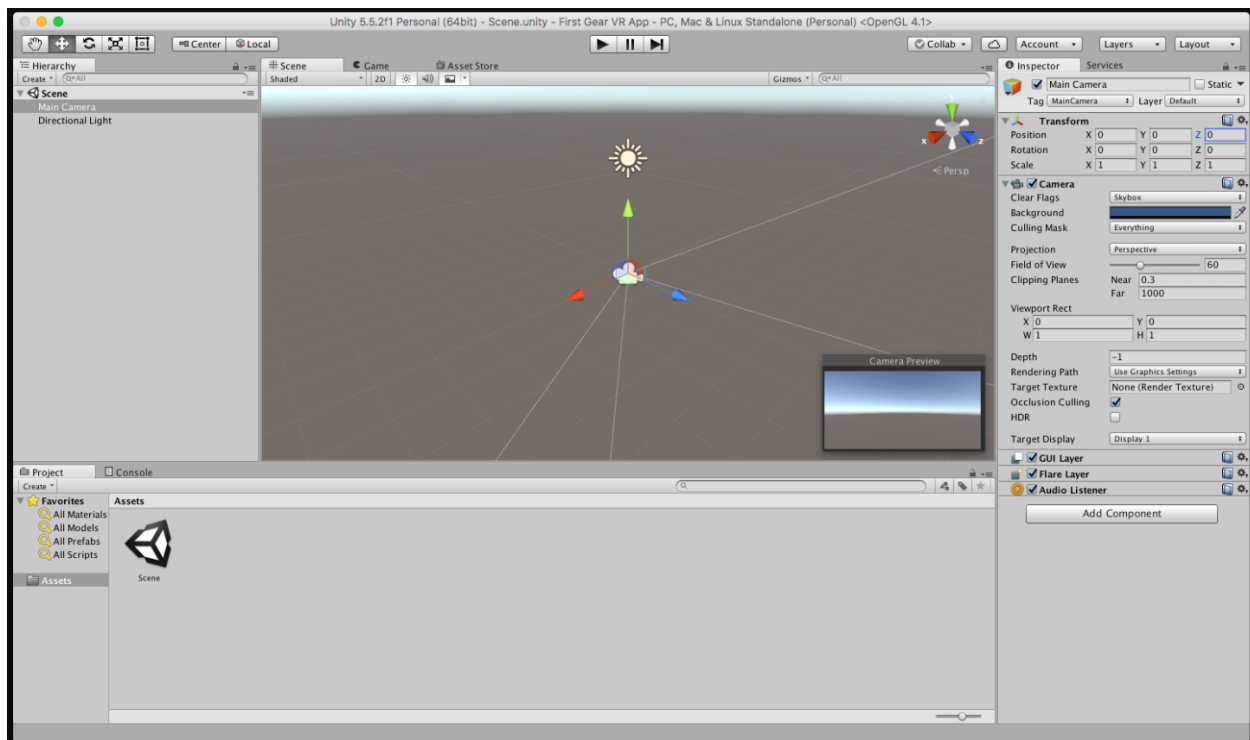
This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.



In the next window, you'll see both the Hierarchy tab and the Scene view. Note that the Main Camera and Directional Light have already been created for you.


Save the scene (File->Save Scene) and choose a name for your scene. I'm just using the name "Scene." Then click on "Main Camera" in the hierarchy view. In the right of the window, click "Inspector" to activate the "Inspector" tab. Under "Transform," change the position to 0, 0, 0. This will center the camera by changing its coordinates to (0, 0, 0).

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.



Now, let's try creating a shape. Right click in the hierarchy window (on the left), and select "3D Object -> Capsule". A capsule should appear in your Scene window. To preview how this scene will look, click on the "Game" tab to bring up the Game window.

Like the camera, we can move the capsule by changing its position in the Inspector. We can also move the capsule and other objects, including the camera and the light, using a different method.

In the upper left corner, click on the button that looks like two perpendicular arrows: .

Now try clicking on the capsule. When the mouse button is held down, you should see a set of three-dimensional axes appear superimposed on the capsule. To move the capsule along an axis, click on the axis and drag it in the direction you want. You can also move the capsule in all three dimensions at once by dragging its center, but this is less precise. It's less confusing for me to click on an axis.

Notice the other three buttons next to the transform button. With these buttons, you can rotate and scale objects. Try playing around with these buttons and see how they affect the capsule and the position, rotation, and scale fields in the Inspector.

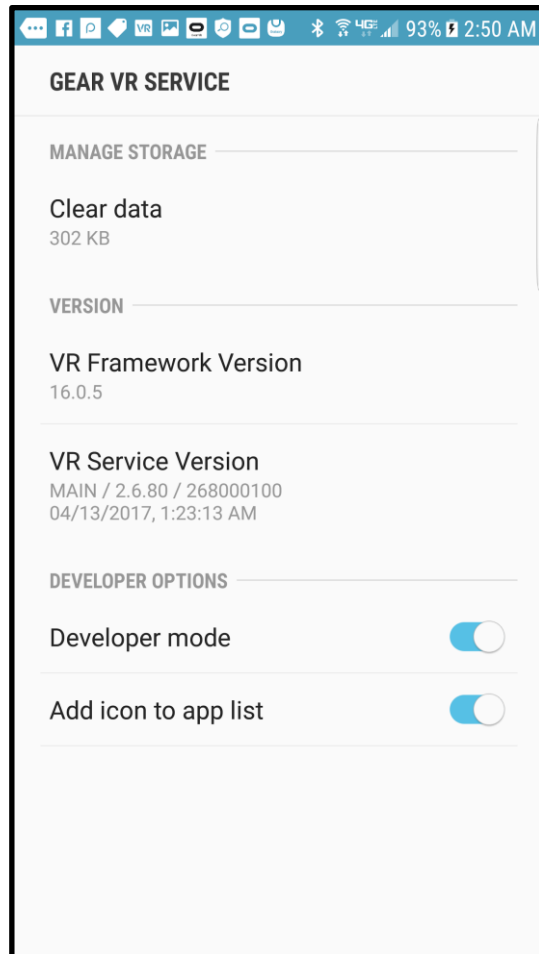
This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

To build and run your project, plug in your Samsung smartphone. Make sure that USB debugging is selected. In Unity, select File -> Build & Run. In the Build Settings window, make sure Android is selected as the platform. If it isn't, click Android and then click "Switch Platform." In the "Scenes to Build" window, select the name of the scene you saved. Then click Build and Run. You will be prompted to save an APK file of your project. I created a directory named "bin/" in my project directory and saved my APK file there.

Your phone will prompt you to insert it in the Gear VR headset. When you do, you should see a scene with a capsule. If the capsule isn't immediately visible, try turning around. Note that you're in a fully three-dimensional scene.

Developer Mode

Now that you've run an app with your own osig file on your phone, you can use an easier way to test VR apps. Instead of unplugging your phone, inserting it into your VR headset, and putting on your headset every time you want to test an app, you can run your app without the headset. To do this, go to Settings on your phone. Select Applications, then Application Manager. Scroll down and select Gear VR Service. Select Storage, and tap on VR Service Version several times until the "Developer options" items appear. When you want to test your app, make sure "Developer mode" is toggled on.



Setting Developer mode in the Gear VR Service on a Samsung Galaxy S6 Edge.

When you build and run an app from Unity with Gear VR developer mode on, the app will start automatically without prompting you to insert the phone into the VR headset. You will see two images, each corresponding to one of the Gear VR lenses.

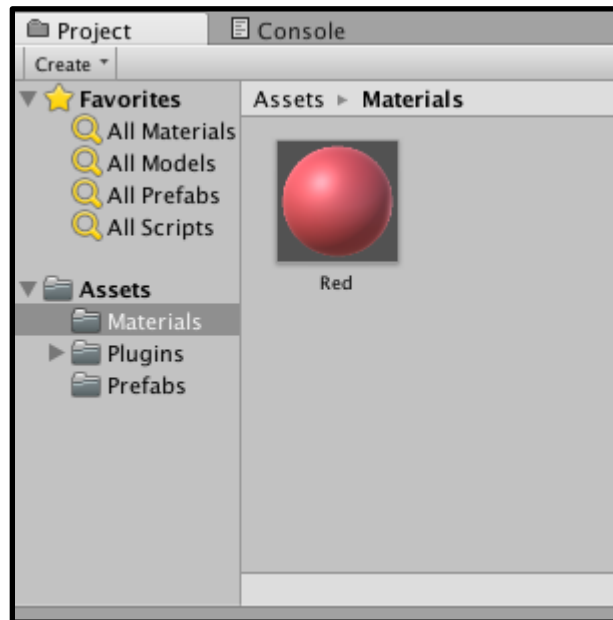
Adding Color

Now that we can run apps easily, let's make our project a little more fun. First, change the name of the capsule to RedPill. Right click on "Capsule" in the hierarchy window and select "Rename." Next, we'll try changing the appearance of our capsule. To do this in Unity, we need to create a material and apply it to the capsule.

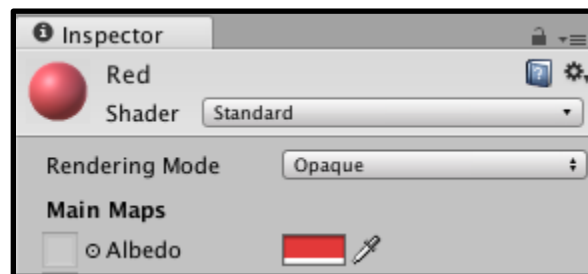
Right-click Assets in the folder list, and select Create Folder to create a new folder. Call it Materials. Right click, highlight Create, and choose Material. When the icon appears, rename it

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

to Red. In the Inspector tab, click on the swatch next to the word Albedo to bring up a color chooser window. Click on a shade of red and close the window. Now the albedo swatch should be red, and the icon for this material should change to red.

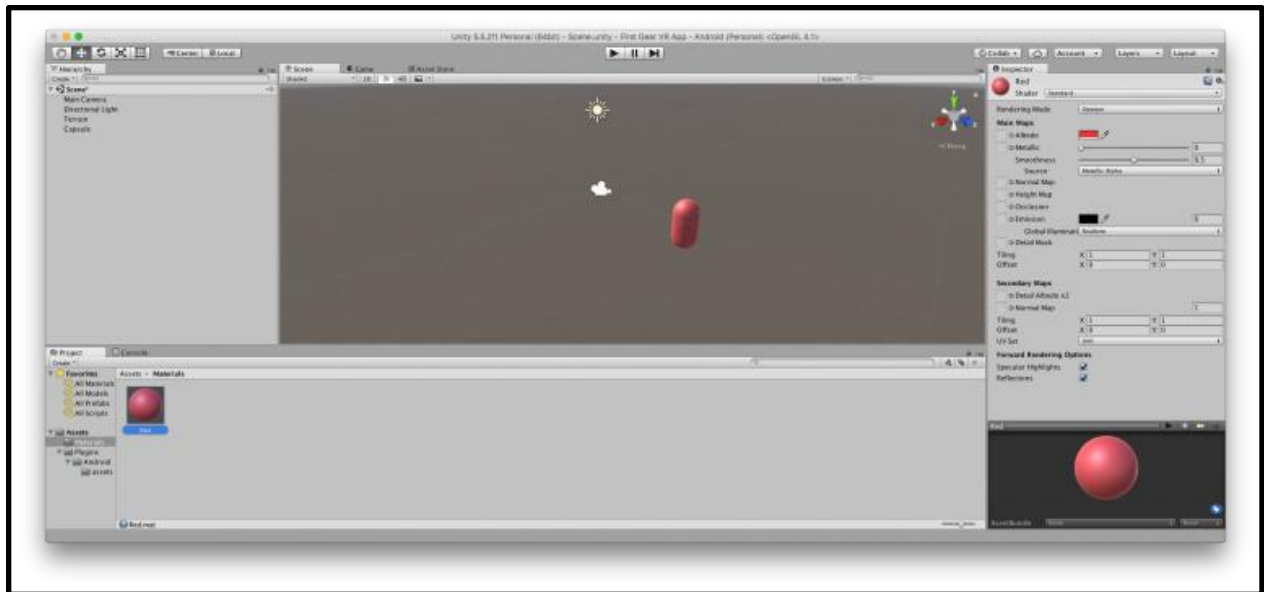


Creating new material



Inspector tab

To apply this material to the capsule, drag its icon onto the capsule in the Scene window.



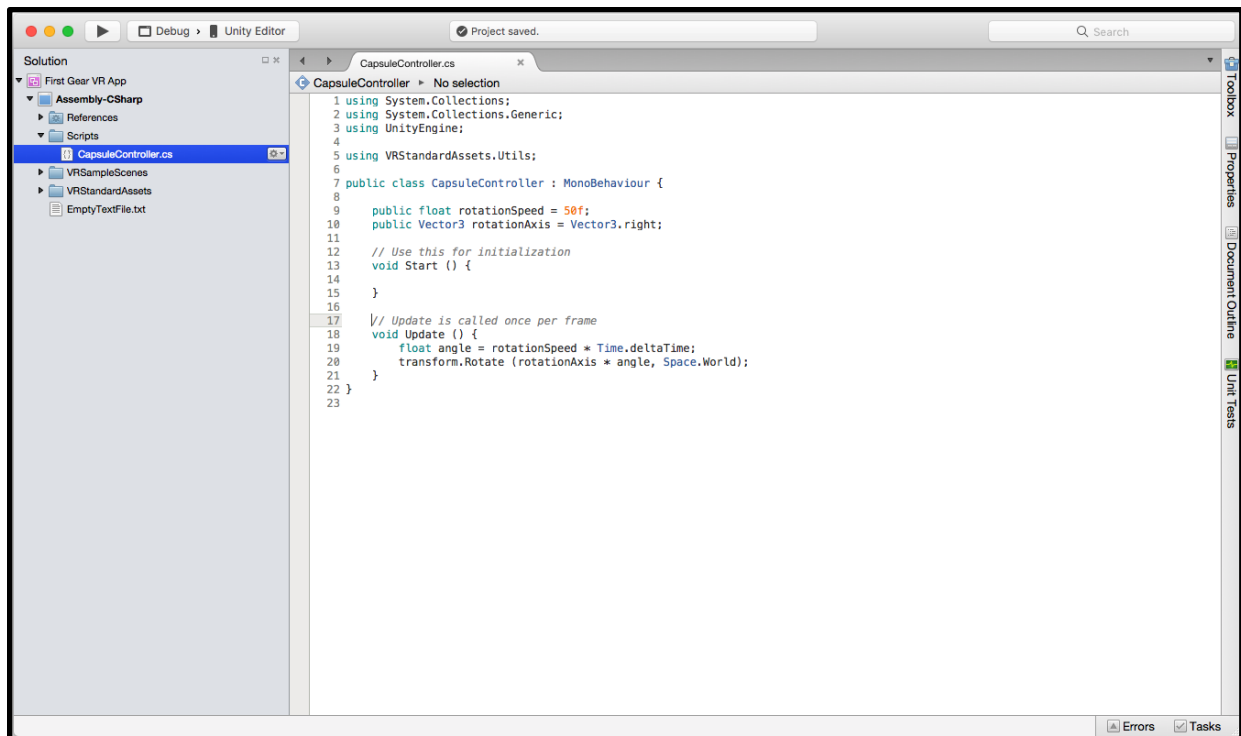
Applying a material

To make our red capsule more dynamic, let's try making it rotate.

In the Inspector tab, scroll down to the bottom and click “Add Component.” Scroll to the bottom of the menu, and choose “New Script.” Enter a name for the script — I’m using CapsuleController — and choose C Sharp as the language. A section labelled “Capsule Controller (Script)” will appear in the Inspector tab.

Click the gear icon in the upper right of this section. Highlight and click “Edit Script” in the popup menu. This should open a MonoDevelop window. MonoDevelop is a C# IDE that was automatically installed with Unity.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.



CapsuleController.cs in MonoDevelop

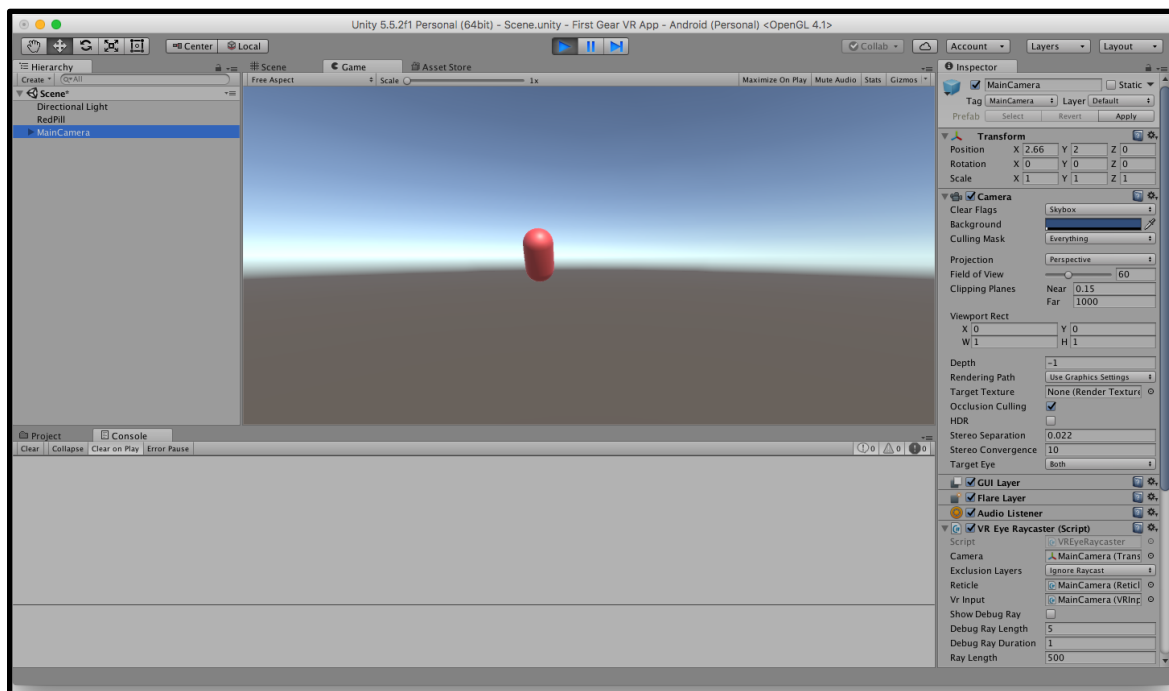
You'll see that MonoDevelop has automatically opened a file named `CapsuleController.cs`. This file defines a class called `CapsuleController` that inherits from `MonoBehaviour` and contains two methods, `Start()` and `Update()`. We will only be modifying `Update()`, which is called once per frame. This is the code we want:

```
1 public class CapsuleController : MonoBehaviour {
2     private static float DEFAULT_ROTATION_SPEED = 50f;
3
4     // Making rotationSpeed and rotationAxis public allows
5     // modifying their values in the Inspector panel.
6     public float rotationSpeed = DEFAULT_ROTATION_SPEED;
7     public Vector3 rotationAxis = Vector3.right;
8
9     // Use this for initialization
10    void Start () {}
11
12    // Update is called once per frame
13    void Update () {
14        float angle = rotationSpeed * Time.deltaTime;
15        transform.Rotate (rotationAxis * angle, Space.World);
16    }
17 }
```

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

Each frame, we want the capsule to rotate by a fixed angle around the z-axis. This angle is product of a velocity and the amount of time for one frame read, `Time.deltaTime`. You can read more about `Time.deltaTime` at <https://docs.unity3d.com/ScriptReference/Time-deltaTime.html>. The rotation speed of 50f works well for a slow rotation. (The 'f' means this is a floating point number.) The variable `rotationAxis` defines the axis around which the capsule will rotate. `Vector3.right` is the the z-axis, which will result in a vertical rotation. By making `rotationSpeed` and `rotationAxis` public class members, we will be able to modify their values in the Inspector panel later.

After saving the C# file in MonoDevelop, return to the Unity window. At the center top of the window, you should see buttons that look like typical play and pause buttons. Click the play button (the triangle pointing right). The view will automatically switch to the Game tab, and you'll see your capsule rotating if it is within view of the camera.



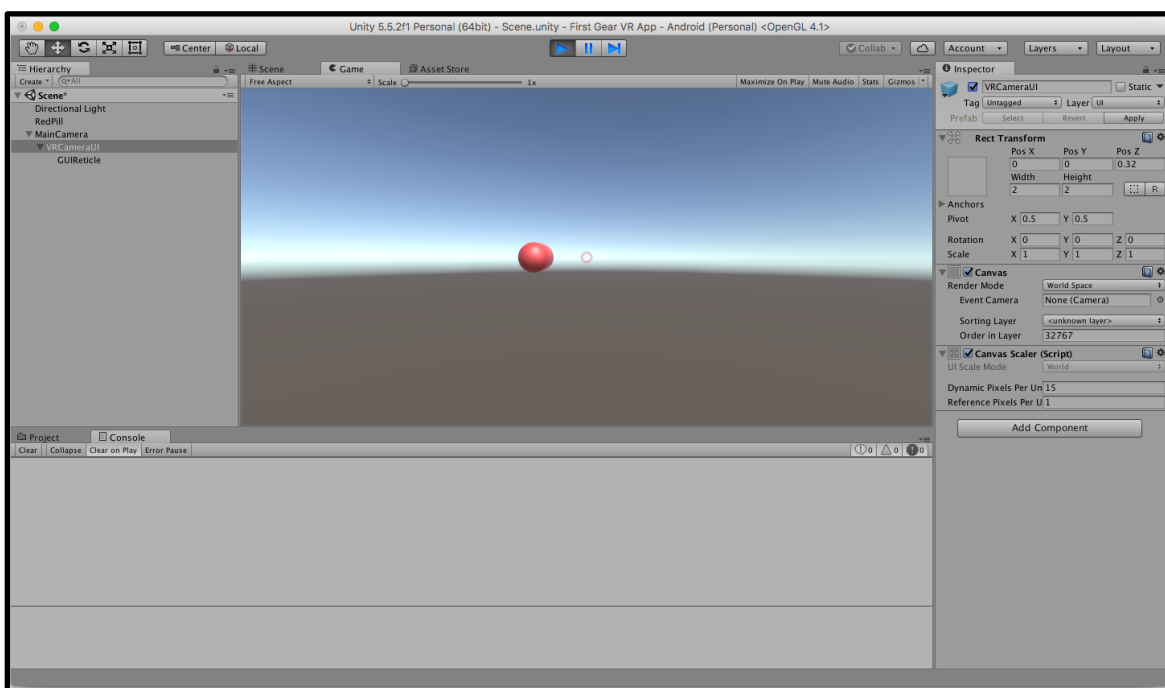
Switch back to the Scene tab. Here, the capsule is rotating, too. To stop playing the scene and reset the view, click the play button again. Try plugging in your phone and select File->Build and Run to see your app in VR.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

Adding Interaction

Now let's try making the red pill interactive. In traditional games, you interact with objects using a keyboard, mouse, or controller. In VR, your own head movement can serve as a controller. In the rest of this tutorial, we're going to make the red pill stop rotating when your gaze rests on it.

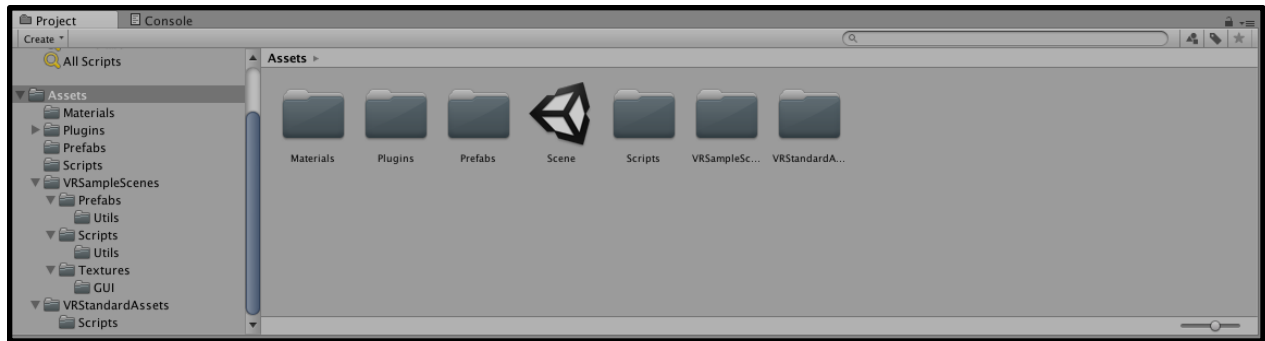
To make it easier where you're looking, we'll attach a reticle, or a targeting crosshair, to the main camera. The reticle appears as circle that shows where your gaze is aimed. We'll also attach a ray caster object to the camera that will determine if the reticle intersects an object.



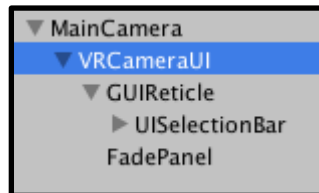
The reticle is visible as a circle that indicates where the player's gaze is centered.

Instead of building all this from scratch, we will be using Unity's VR Standard Assets and VR Samples, which are provided in the VR Samples bundle available in Unity's Asset Store. The assets you need are in the source code of this tutorial. If you're following along by building your own project, copy Assets/VRSampleScenes and Assets/VRStandardAssets from the included source code to your project's Assets directory. Your project view in Unity will refresh with these assets once they are copied.

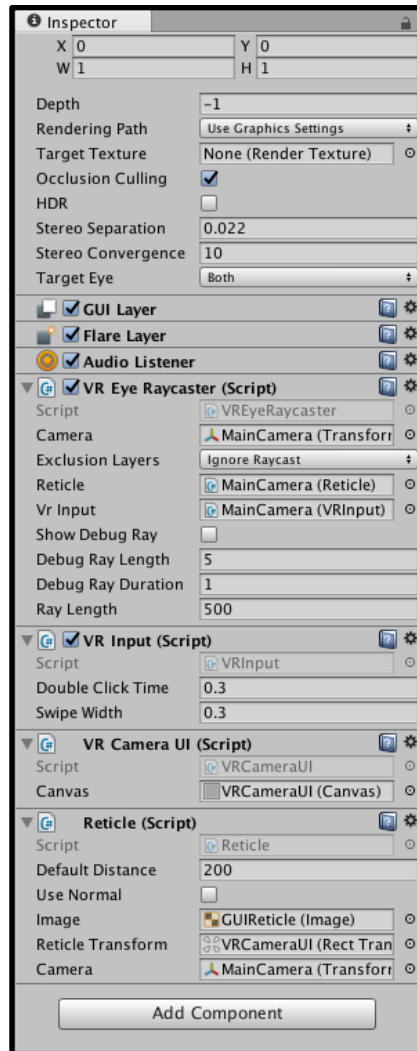
This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.



Assets/VRSampleScenes/Prefabs/Utils contains a prefab for the MainCamera that already has the reticle. We will replace the default MainCamera with an instance of this prefab. In your Hierarchy window, right click and remove the default MainCamera. Copy the MainCamera prefab from the Project window to the Hierarchy window. Expand the MainCamera by clicking the little arrow next to it to explore what you just added.



When MainCamera is highlighted, the Inspector window shows what scripts are attached to it.



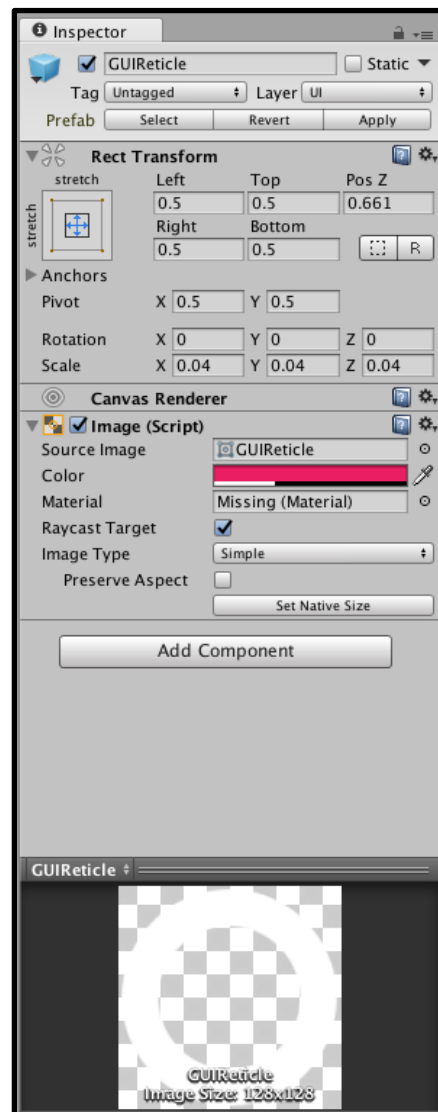
VREyeRayCaster is a class that draws a ray emanating from the camera and detects intersections between the ray and instances of the VRInteractiveltem class. Interactive game objects must contain an instance of VRInteractiveltem. If an intersection is detected, the VRInteractiveltem's Over() method is called.

VRCameraUI defines properties of the VRCameraUI canvas. Reticle defines properties of the reticle image, and VRInput handles inputs like swipes and clicks. We are only concerned with clicks in this project.

Nested beneath MainCamera is a canvas called VRCameraUI. In Unity, a canvas is a game object that contains UI elements. When VRCameraUI is highlighted in the

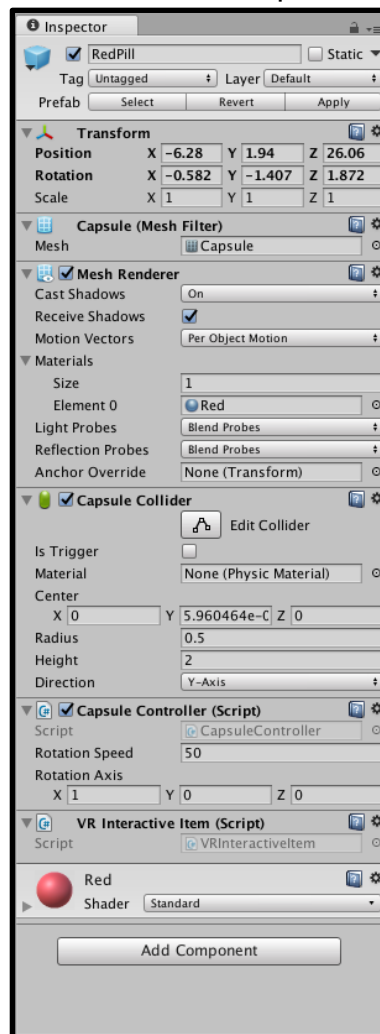
This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

hierarchy window, the canvas appears as a rectangle in front of the camera in the game window. Beneath VRCameraUI, you'll find an image called GUIReticle. In Unity, an Image is a UI control that displays an image from a file. If the "Source Image" field in the Inspector is empty, click on the small icon next to the field and select GUIReticle. This is an image included in the Assets/VRSamples/Textures folder.



Finally, we need to add a collider to the RedPill and make RedPill a VRInteractiveItem. Click on RedPill in the Hierarchy window. In the Inspector, click "Add Component". Select Physics, then Capsule Collider. In the Project window, select Assets-

>VRStandardAssets->Scripts->VRInteractiveltem, and drag it to the bottom of the Inspector. This will add the VRInteractiveltem script to RedPill.



VRInteractiveltem's data members include event variables called `OnOver` and `OnOut` and methods called `Over()` and `Out()`. When the MainCamera's VREyeRayCaster component detects a VRInteractiveltem, it calls the `VRInteractiveItem.Over()`, which calls `VRInteractiveItem.OnOver` like a function.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

```

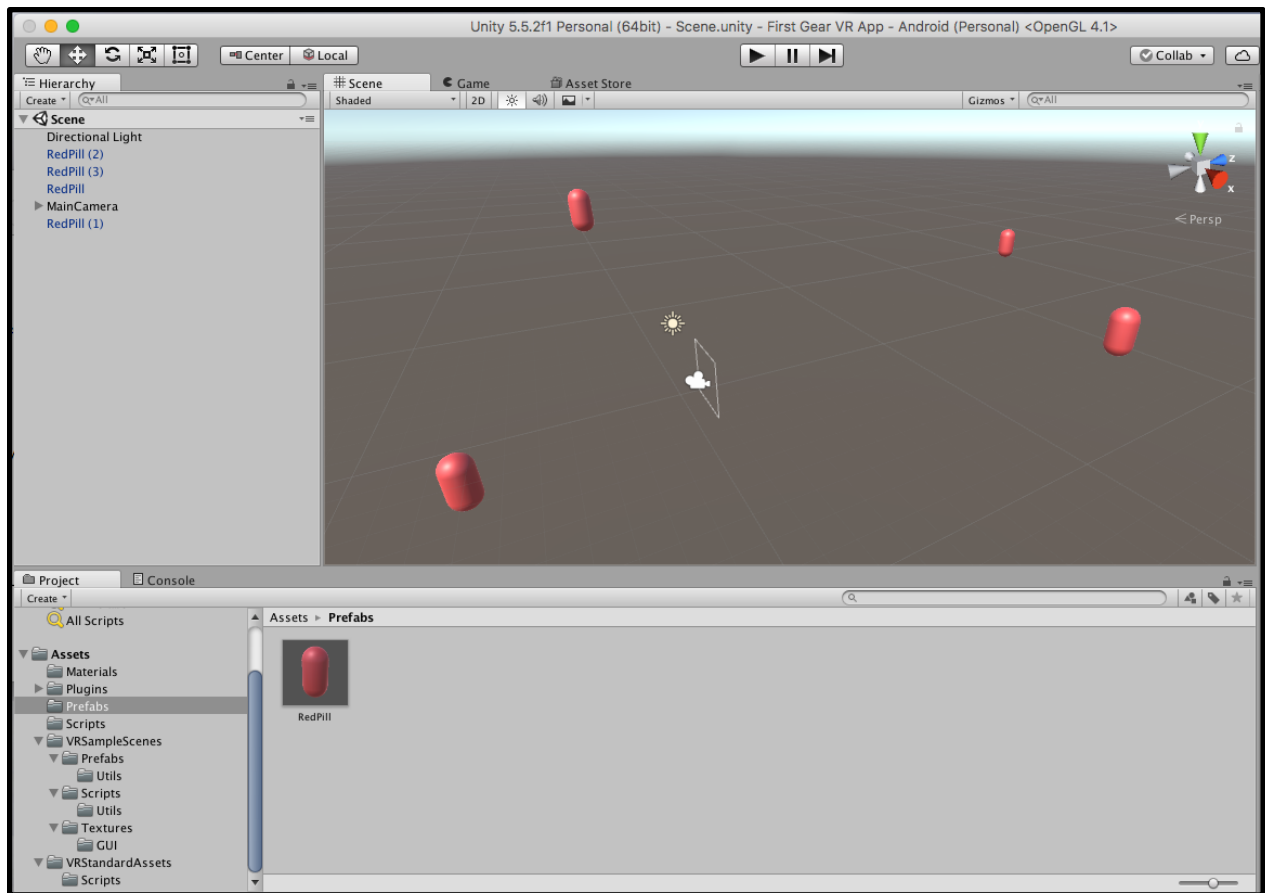
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 using VRStandardAssets.Utils;
6
7 public class CapsuleController : MonoBehaviour {
8
9     private static float DEFAULT_ROTATION_SPEED = 50f;
10
11     // Making rotationSpeed and rotationAxis public allows
12     // modifying their values in the Inspector panel.
13     public float rotationSpeed = DEFAULT_ROTATION_SPEED;
14     public Vector3 rotationAxis = Vector3.right;
15
16     private VRInteractiveItem interactiveItem;
17
18     void Awake() {
19         interactiveItem = gameObject.GetComponent<VRInteractiveItem> ();
20     }
21
22
23     // Use this for initialization
24     void Start () {
25         if (interactiveItem != null) {
26             interactiveItem.OnOver += HandleOver;
27             interactiveItem.OnOut += HandleOut;
28         }
29     }
30
31     void OnDisable() {
32         interactiveItem.OnOver -= HandleOver;
33         interactiveItem.OnOut -= HandleOut;
34     }
35
36     // Update is called once per frame
37     void Update () {
38         float angle = rotationSpeed * Time.deltaTime;
39         transform.Rotate (rotationAxis * angle, Space.World);
40     }
41
42     private void HandleOver() {
43         SetRotationSpeed (0);
44     }
45
46     private void HandleOut() {
47         ResetRotationSpeed ();
48     }
49
50     private void SetRotationSpeed(float newSpeed) {
51         rotationSpeed = newSpeed;
52     }
53
54     private void ResetRotationSpeed() {
55         rotationSpeed = DEFAULT_ROTATION_SPEED;
56     }
57 }

```

When `HandleOver()` is called, the helper method `SetRotationSpeed()` sets the rotation speed to 0 to stop rotation. When `HandleOut()` is called when the player's gaze moves away, the rotation speed is reset to the default by the method `ResetRotationSpeed()`.

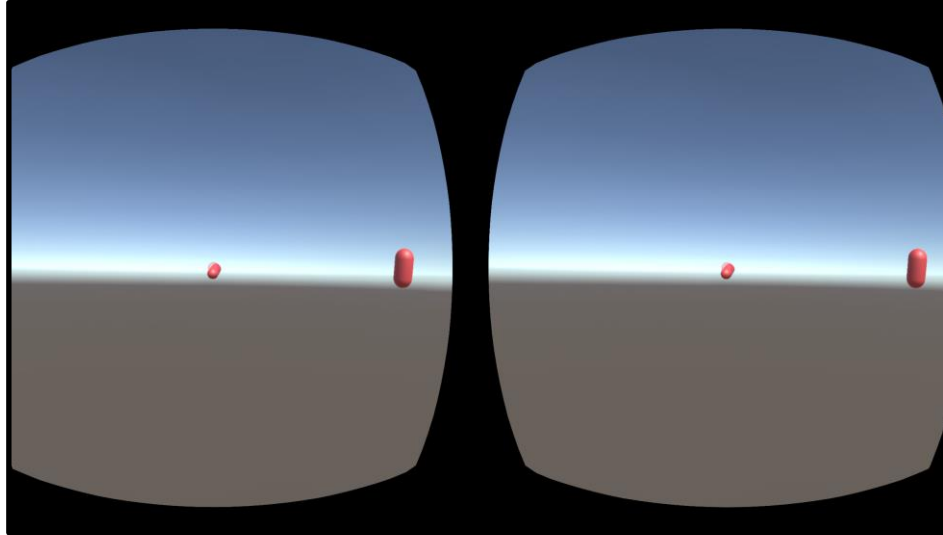
As a last step, let's make RedPill a reusable asset. In the Project view, expand Assets and select Prefabs. Drag the RedPill from the Hierarchy panel down to the right side of the Project view, in the panel now titled Assets->Prefabs. Now, to make more RedPills, drag the RedPill icon from Assets->Prefabs and drop them where you want them in the Scene window.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.



When you run your app in VR, you'll see multiple rotating RedPills, and you can stop the rotation of any one of them by aiming your reticle at it. You now have created your interactive virtual world on the Gear VR.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.



How to Create a Virtual Reality Tour in Unity for the GearVR

Virtual reality allows us to immerse ourselves in 3D worlds. This tutorial will show you how to create a VR tour of multiple worlds. In the process, we will be importing Blender models, creating interactive UI components, and switching scenes.

Source Code Files

Download [here](#).

Creating the Project

To create the project, open Unity and choose “New Project.” Choose a name and directory for your project.

You will need an Oculus signature file that is unique to your phone in order to run the game on a headset. Instructions to create a new signature file, if you don't have one already, are at [this link](#). Once you have created a file, copy it into the Assets/Plugins/Android/assets folder in your project's directory.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

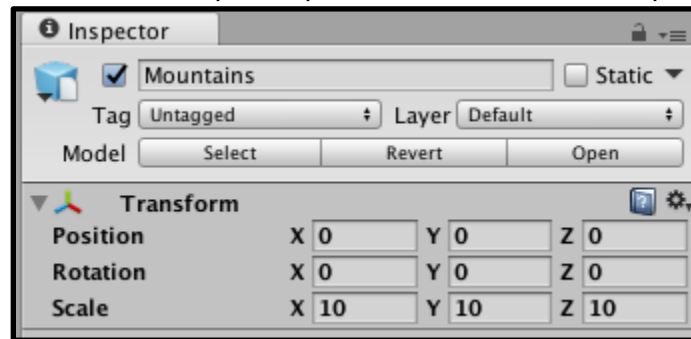
Importing Models

We'll be using low-poly models created in Blender for our landscapes and our flying platform. The models are located in the Assets/Models folder of the included Unity project. Copy this folder into your own project.

You can also create and import your own models from Blender. In Blender, delete the model's camera and export the model as a .fbx file. Save this file in the Assets/Models folder of your Unity project.

Setting the Scene

Our first scene will be a tour of a mountainous landscape. Load the Assets/Models folder in the Project pane to see your models. If you've copied the Assets/Models folder from the included project, drag the Mountains model into the Scene pane. Select Mountains in the hierarchy pane, and adjust the scale values in the Inspector pane until the scene looks pleasing.



Inspector view of the Mountains terrain.



Top view of the terrain.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

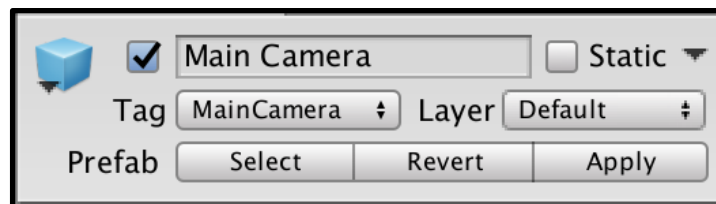
Creating the Player

Next, we're going to create our Player object. The Player will be traversing the scene on a flying platform, stopping at various waypoints along the way to view the scenery and descriptive text.

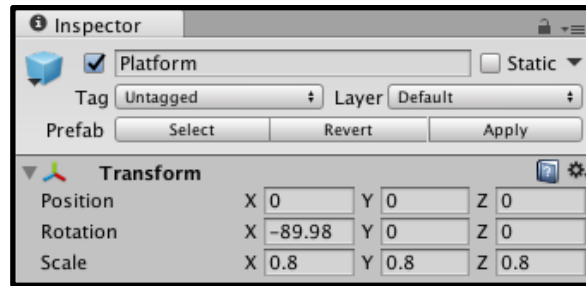
If you were building a non-VR first person experience, you could move the player just by moving the camera. However, in virtual reality, the headset is the camera. When you move your head, the headset sends the camera coordinates to the game. As a result, the game should not be changing the camera's coordinates. If it does, your camera view will not match your actual head movement.

How do we move the player if the game should not change camera coordinates directly? The solution is to encapsulate the camera inside another object that represents the player. When the player moves within the game world, the game changes the coordinates of the player object, but the camera's coordinates are still controlled by the player's physical head movements.

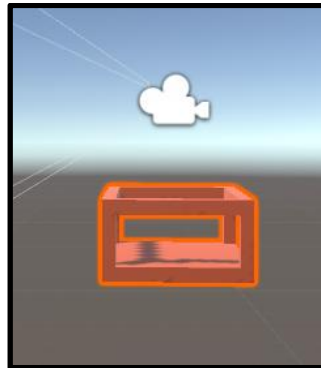
In our game, create a new empty object named Player by right clicking and selecting "Create Empty" in the hierarchy pane. Drag the Main Camera onto the Player object in the Hierarchy pane to nest Main Camera under Player. In the Inspector pane, select MainCamera as the tag of the Main Camera object. This allows us to access the Main Camera in other scripts using `main.Camera`. We will be using this later.



Since the player will be standing on a platform, we will need to add a platform asset to the Player object as well. From Assets/Models, drag the Platform model into the Player object to nest it. In the Inspector pane, reset the coordinates of the platform and then rotate it until it is right side up in the scene. We want the player to stand on the platform with railings around him.

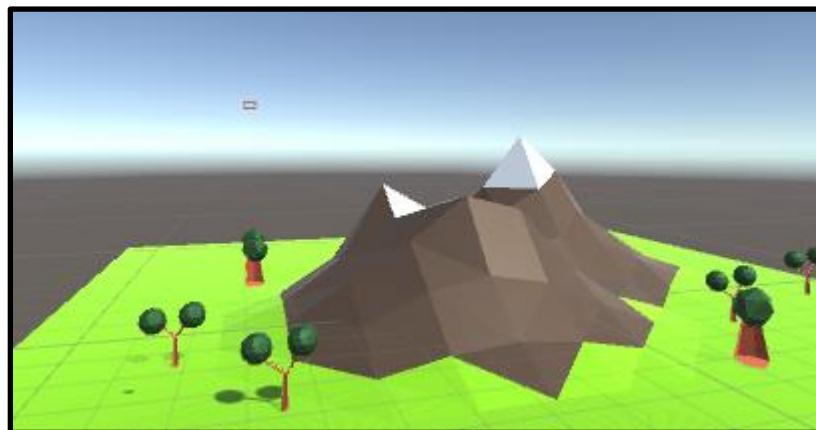


Platform in the Inspector.



Player object consists of a platform and the Main Camera.

Use the transform widget to move the Player object to where you want to start the tour. For best results, position the Player above the terrain. If the terrain is looking a little too small relative to your platform, or the platform too large, scale up the terrain, or scale down the platform.



To make the player's platform move, we will need to add a script that updates the player's position each frame. Create a script named PlayerController.cs in Assets/Scripts, and add this script to Player.

This is the code:

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using UnityEngine.SceneManagement;
5 using UnityEngine.EventSystems;
6
7 /**
8  * This class manages the movement of the player.
9  */
10 public class PlayerController : MonoBehaviour {
11
12     // Array of waypoints that the player will visit.
13     // Set in the Inspector.
14     public WaypointController[] waypoints;
15     public float speed = 2f;
16
17     // The current waypoint that the player is traveling to.
18     int currentWaypoint = 0;
19     // Flag that indicates whether the player is currently in motion.
20     bool isMoving = true;
21
22     // Checks if any of the Waypoints are null.
23     void CheckWaypoints() {
24         Debug.Log("Checking " + waypoints.Length + " waypoints");
25         if (waypoints == null) {
26             Debug.Log ("waypoints array is null");
27         }
28         for (int i = 0; i < waypoints.Length; i++) {
29             if (waypoints [i] == null) {
30                 Debug.Log ("Waypoint " + i + " is null.");
31             }
32         }
33     }
34
35     // Checks if the player has reached the current waypoint.
36     bool AtWaypoint() {
37         float distance;
38         if (waypoints [currentWaypoint] != null) {
39             distance = Vector3.Distance (transform.position, waypoints [currentWaypoint].transform.position);
40         } else {
41             Debug.Log ("Cannot determine distance because waypoint is null.");
42             distance = 0;
43         }
44
45         return (distance == 0);
46     }
47 }
```

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

```

47
48 // Update is called once per frame.
49 // If isMoving is true, and the player has not arrived at the current
50 // Waypoint, move the player towards the Waypoint.
51 void Update () {
52     if (isMoving) {
53         if (!AtWaypoint ()) {
54             transform.position = Vector3.MoveTowards (transform.position, waypoints [currentWaypoint].transform.position, speed *
55         } else {
56             Debug.Log ("Arrived at waypoint " + currentWaypoint + ". Showing description.");
57             isMoving = false;
58             waypoints [currentWaypoint].ShowDescription ();
59         }
60     }
61 }
62
63 // Called after the player clicks on the button to close a waypoint's description.
64 public void ContinueTour() {
65     Debug.Log("ContinueTour: currentWaypoint=" + currentWaypoint);
66
67     if (AtWaypoint() && waypoints [currentWaypoint] != null) {
68         // Hide the description.
69         Debug.Log ("Hiding waypoint " + currentWaypoint);
70         waypoints [currentWaypoint].HideDescription ();
71         // If this is the last waypoint in the circuit,
72         // load the next scene.
73         if (currentWaypoint == waypoints.Length - 1) {
74             Debug.Log ("Reached end of tour.");
75             Debug.Log("Loading next scene");
76             LoadNextScene ();
77         }
78
79         // If there are more waypoints, continue to the next waypoint.
80         else {
81             currentWaypoint++;
82             Debug.Log ("Next waypoint: " + currentWaypoint);
83             isMoving = true;
84         }
85     } else {
86         Debug.Log ("ERROR: Something's wrong... Either we have not arrived at the waypoint, or the waypoint is null");
87     }
88 }
89
90 // Loads the next scene. If we're already at the last scene,
91 // loads the first scene.
92 void LoadNextScene() {
93     Debug.Log ("Loading next scene");
94     int currentSceneIndex = SceneManager.GetActiveScene ().buildIndex;
95     if (currentSceneIndex < SceneManager.sceneCountInBuildSettings - 1) {
96         SceneManager.LoadScene (currentSceneIndex + 1);
97     } else {
98         SceneManager.LoadScene (0);
99     }
100 }
101
102 }
103 }

```

PlayerController contains a field for speed and a flag that indicates whether the player should be moving. Waypoints the player will visit are stored in the `waypoints` array. `currentWaypoint` keeps track of which waypoint is the current destination. In `Update()`, if the player has arrived at a waypoint (distance is 0), the player stops (`isMoving` is set to false), and the waypoint's description is shown. When the player is done with a waypoint, `ContinueTour()` is called, and the player moves on to the next waypoint. If the player is already at the last waypoint, `LoadNextScene()` is called to switch to the next scene.

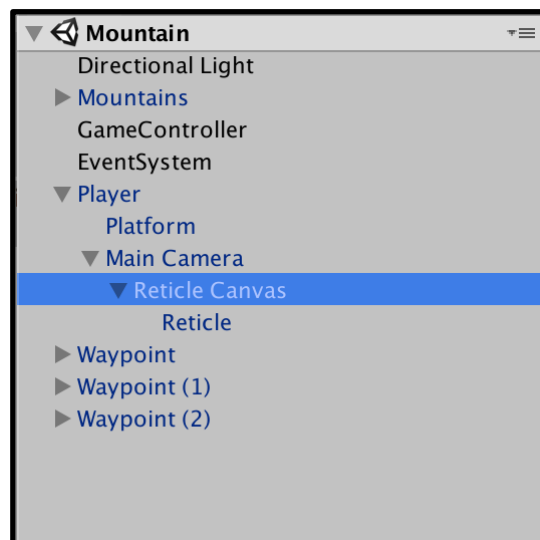
This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

Adding a Reticle

The player will need a reticle, or targeting crosshair, to interact with objects like the buttons we will be creating. The VR standard assets, which are included in the free VRSamples package from the Unity store, contain the assets needed to create a reticle. You can copy VRStandardAssets from the Assets folder in the project included with this tutorial and paste it into your project's Assets folder.

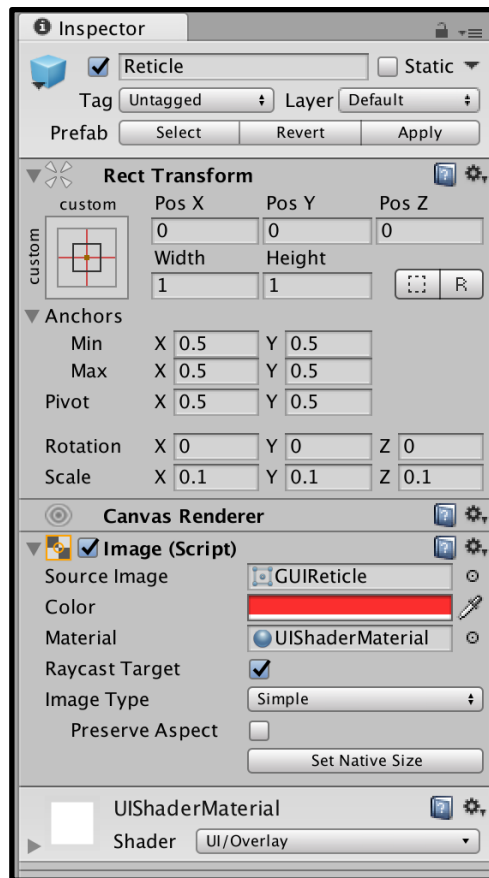
Add the following scripts from VRStandardAssets/Scripts to the Main Camera: VREyeRayCaster.cs and VRInput.cs. Also add the following scripts from VRStandardAssets/Utils/Scripts: VRCameraUI.cs and Reticle.cs. You can attach scripts by dragging them from the project pane directly onto the Main Camera in the hierarchy pane.

Now create an image for the reticle. Right click on the Main Camera and select Create->UI->Canvas. Right click on the new canvas and select Create->UI->Image. Rename the image to Reticle and the canvas to Reticle Canvas.

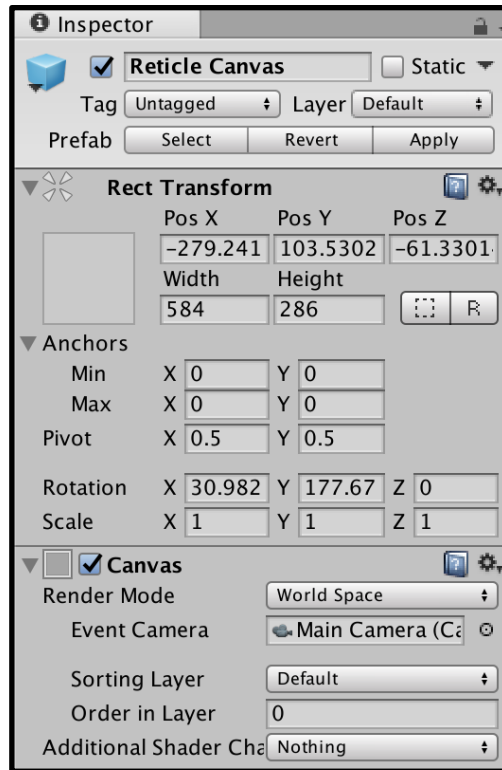


The Main Camera, Reticle Canvas, and Reticle in the hierarchy view.

In the Reticle's inspector, set GUIReticle as the sprite. Select a color that is easy to see. Set width and height to 1 and scale to 0.1. This sets the reticle image to a reasonable size.



In the Reticle Canvas's inspector, set render mode to World and drag the Main Camera into the camera field.



Adding Waypoints

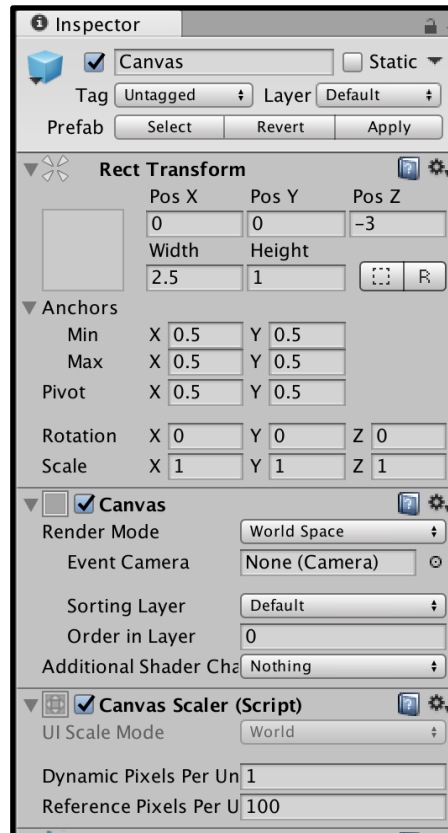
The Player will be visiting Waypoints that are game objects containing a canvas with text and a clickable button.

To create a waypoint, create a new empty object in the hierarchy pane and name it “Waypoint.” Use the transform widget to move the Waypoint to a place in the scene where you want the player to stop. When the player reaches the Waypoint, text is supposed to appear, along with a button that continues the tour when clicked. Text and buttons in Unity are UI (user interface) elements that must be rendered on a canvas. In fact, if you try to create a UI element by itself, Unity will automatically create a canvas.

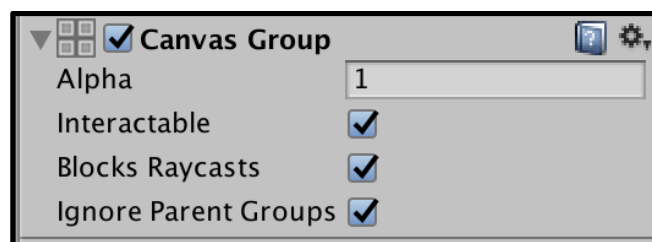
We are going to nest a canvas under Waypoint and nest our UI elements under the canvas. Right click Waypoint in the hierarchy pane and select UI->Canvas. In the inspector for the canvas, select World Space as the render mode. This makes the canvas exist as an object in the game world instead of as a fixed overlay over the camera. Under Rect Transform, set width and height to 2.5 and 1, respectively. With

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

reference pixels per unit set to 100 in the Canvas Scaler, this equates to dimensions of 0.025 meter by 0.01 meter. Set Pos Y to 1. Set Pos Z to -3 for the canvas to appear about 3m past the Waypoint.



Click Add Component and choose Canvas Group. A Canvas Group lets us set the visibility of the canvas and all its UI components.

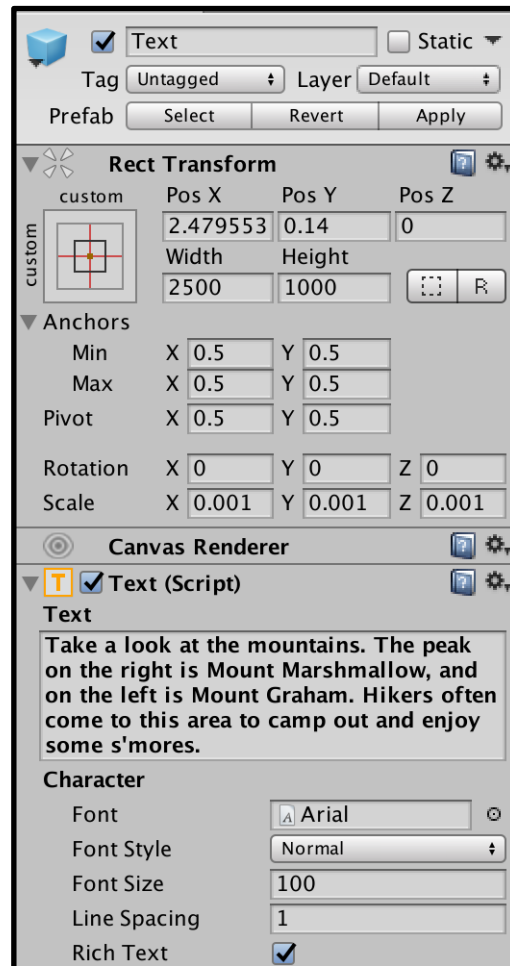


Create a panel nested under the canvas by right clicking the canvas and selecting UI->Panel. In the Image section in the Inspector pane, choose "None" as the source image,

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

and set a color you like. In the hierarchy pane, right click the panel and select UI->Text to create a text element nested under the panel.

Text will appear sharper if you set the width, height, and font to large values and then scale down. To match the canvas, set width and height to 2500 by 100. Set the X, Y, and Z scales to 0.001 to scale down to the size of the canvas. In the Text section of the Inspector, set font size to 100. In the text box, type a description of the Waypoint's location.



We want the canvas to be visible until after we arrive at a Waypoint, so we will need to create another script to set the visibility. In the project pane, select the Assets/Scripts folder and create a script named WaypointController.cs with the following code:

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.


```

1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using UnityEngine.UI;
5
6 /**
7  * This class manages Waypoints' canvases.
8  */
9 public class WaypointController : MonoBehaviour {
10
11     Canvas waypointDescriptionCanvas;
12     CanvasGroup canvasGroup;
13
14     // Called when this object is created.
15     void Awake() {
16         waypointDescriptionCanvas = GetComponentInChildren<Canvas> ();
17
18         if (waypointDescriptionCanvas == null) {
19             Debug.Log ("Could not get canvas.");
20         } else {
21             // Get the canvas's CanvasGroup, and hide it.
22             canvasGroup = waypointDescriptionCanvas.GetComponent<CanvasGroup>();
23             HideDescription();
24         }
25     }
26
27     // Make the CanvasGroup visible.
28     public void ShowDescription() {
29         waypointDescriptionCanvas.transform.position = Camera.main.transform.position + new Vector3 (0f, -0.5f, 3f);
30         //waypointDescriptionCanvas.transform.rotation = new Quaternion( 0.0f, Camera.main.transform.rotation.y, 0.0f, Camera.main.trans
31
32         // Set the canvas's forward vector to face the camera,
33         // so that the text is right-side up to the camera.
34         Vector3 direction = waypointDescriptionCanvas.transform.position - Camera.main.transform.position;
35         waypointDescriptionCanvas.transform.forward = direction;
36
37         // Make the CanvasGroup visible, and allow interactions
38         // with its UI components.
39         canvasGroup.alpha = 1;
40         canvasGroup.interactable = true;
41     }
42
43     // Hide the CanvasGroup.
44     public void HideDescription() {
45         if (canvasGroup != null) {
46             // Make the CanvasGroup invisible,
47             // and do not allow interactions.
48             canvasGroup.alpha = 0;
49             canvasGroup.interactable = false;
50         } else {
51             Debug.Log ("canvasGroup is null");
52         }
53     }
54 }
55 }

```

In `Awake()`, which is called when the script is loaded, we assign the Canvas and CanvasGroup. The methods `ShowDescription()` and `HideDescription()` toggle the CanvasGroup's visibility by changing its alpha value, or transparency. They also set the CanvasGroup's `interactable` field, which determines whether the user can interact with the UI components in the Canvas Group. We would not want the player to be able to click an invisible button.

Add WaypointController.cs to the Waypoint object by dragging it from the project pane onto Waypoint in the hierarchy view.

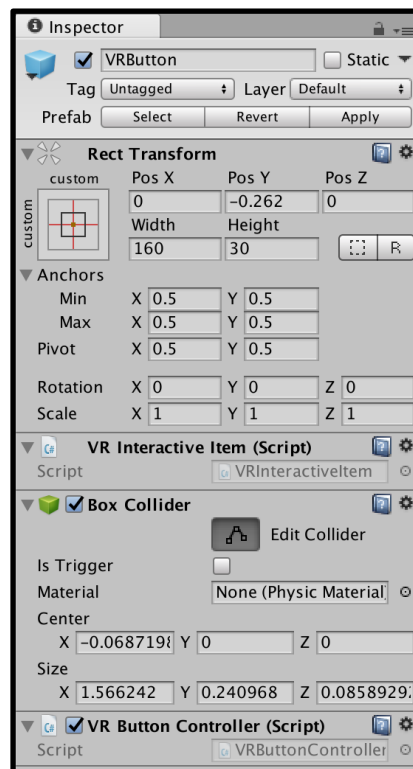
This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

Now set the text of the Waypoint in the scene to something more descriptive. Also change the name of the Waypoint object to something more identifiable.

Clickable Buttons

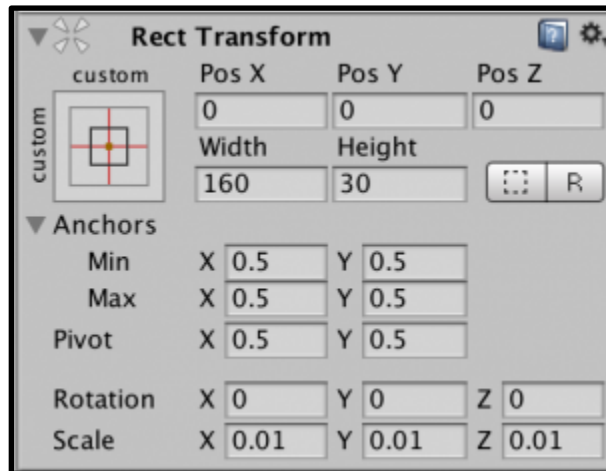
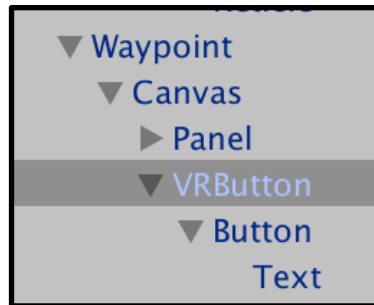
When the Player arrives at a Waypoint, `PlayerController.isMoving` is set to false to stop the Player, and the Waypoint's canvas becomes visible. The Player needs to click a button to close the canvas and continue the tour. We're going to add this button to the Waypoint's canvas and make it an interactive VR item. We're also going to add a collider to the button.

Right click Canvas in the hierarchy pane, and add an empty object named VRButton. This object will be the parent of a standard Unity button. Set VRButton's width and height to 160 x 30.

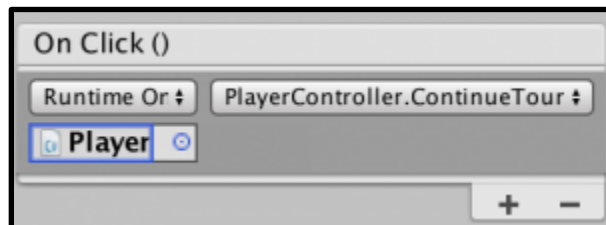


Right click VRButton and select UI->Button to nest a button. Set the button's width and height to 160x30, but set the scale factors to 0.01 to make it fit on the canvas.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

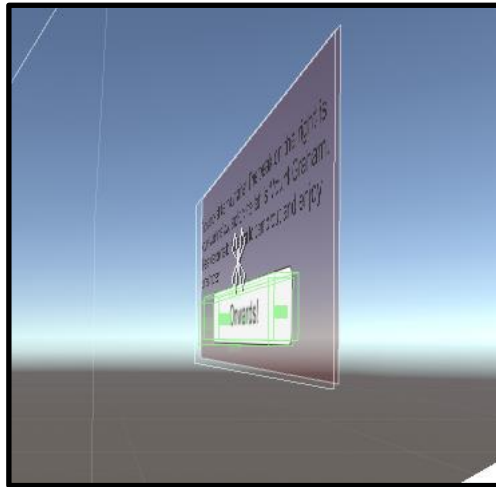


In the Button's inspector, scroll down until you see the "On Click()" section. This is where you define a function to be called when the button is clicked. Select "Runtime or Editor" from the first dropdown. In the dropdown below it, click the circle, and, in the popup window, select "Scene". Find "Player", and double click to select it. In the remaining dropdown, select `PlayerController->ContinueTour()`.



To make the button interactive, drag `VRInteractiveltem.cs` from `Assets/VRStandardAssets/Scripts` onto `VRButton`. In the inspector, click "Add Component", and add a box collider. Click "Edit Collider" and resize the collider so that it

encloses the button in the Scene view. We are adding these components to VRButton, not Button.



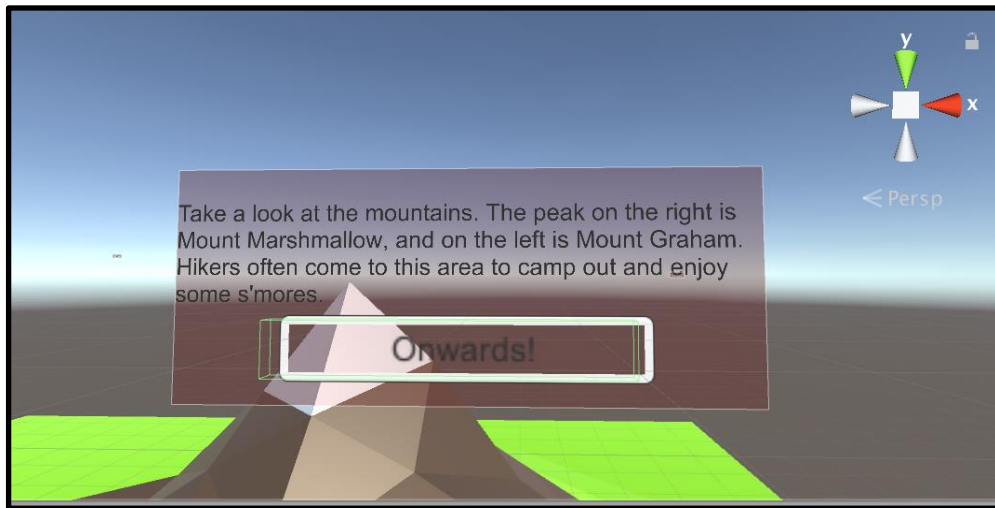
The green box is the box collider that encloses the button.

Finally, we will attach a script to VRButton that will connect VRInteractiveItem.OnClick with the button. Create a script named VRButtonController.cs, and drag it to the VRButton object.

```
1 public class VRButtonController : MonoBehaviour {
2
3     VRInteractiveItem vrItem;
4     Button button;
5
6     void Awake () {
7         // Get the VRInteractiveItem and the Button.
8         vrItem = GetComponent<VRInteractiveItem> ();
9         button = GetComponentInChildren<Button> ();
10    }
11
12    void OnEnable() {
13        // Subscribe the button's onClick function to
14        // vrItem's.
15        vrItem.OnClick += button.onClick.Invoke;
16    }
17
18    void OnDisable() {
19        vrItem.OnClick -= button.onClick.Invoke;
20    }
21
22 }
```

Now, when the button is targeted by the reticle, and the Player clicks, the button's onClick function (`PlayerController.ContinueTour()`) will be invoked.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.



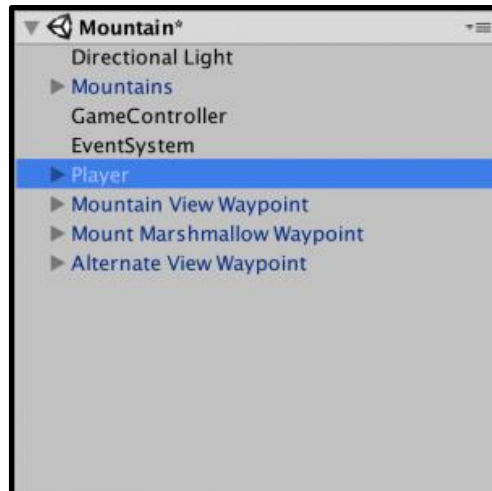
A complete Waypoint.

Prefabs

We've created one Waypoint, but we need more Waypoints that have different locations and descriptions with the same functionality. We might also want to reuse these Waypoints in other scenes. To accomplish this, we can turn Waypoint into a prefab.

In the project pane, create a new folder, Assets/Prefabs. Drag Waypoint from the hierarchy pane to Assets/Prefabs.

Now add more Waypoints to the scene by dragging Waypoint from Assets/Prefabs to the hierarchy pane. Use the scene view and the transform widget to position Waypoints. Remember to set the text description of each Waypoint. You can also change the names of your Waypoint instances to something more identifiable.



The hierarchy with multiple Waypoints defined and renamed.

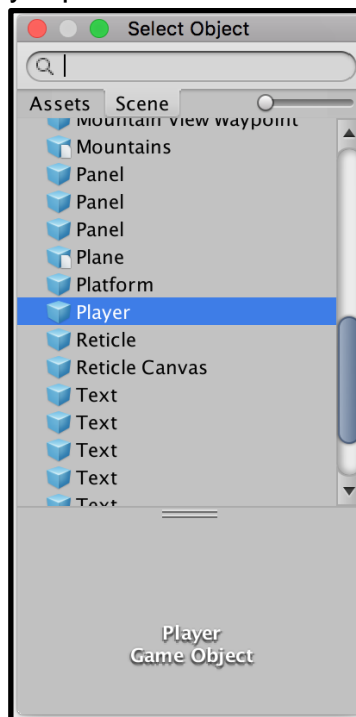
You will need to assign the onClick function of each Waypoint's button separately. This information is not saved in the prefab.

As you add Waypoints to the scene, you will also need to update PlayerController's waypoints array. With Player highlighted, set the size of the array in the inspector, and drag Waypoints into the fields in the order they should be visited.



While we're on the subject of prefabs, let's think about the other complex object in this scene: Player. When we create a new scene, we should not have to re-create the Player object all over again. In programming, don't repeat yourself. Instead, drag the Player object into Assets/Prefabs to turn Player into a prefab.

But remember: You will need to repopulate the Waypoints array when you add Player to a new scene. Also, be careful when you set the button's onClick function to select the Player game object, not the Player prefab.



Make sure the Player game object is selected, not the Player prefab!

Adding Scenes

Now that we have created prefabs, we can create additional scenes that will be connected by `PlayerController.LoadNextScene()`. When the Player has reached the last Waypoint in a scene and clicks the “Onwards!” button, `PlayerController::LoadNextScene()` is called to automatically load the next scene. If we are already at the last scene, the first scene is loaded.

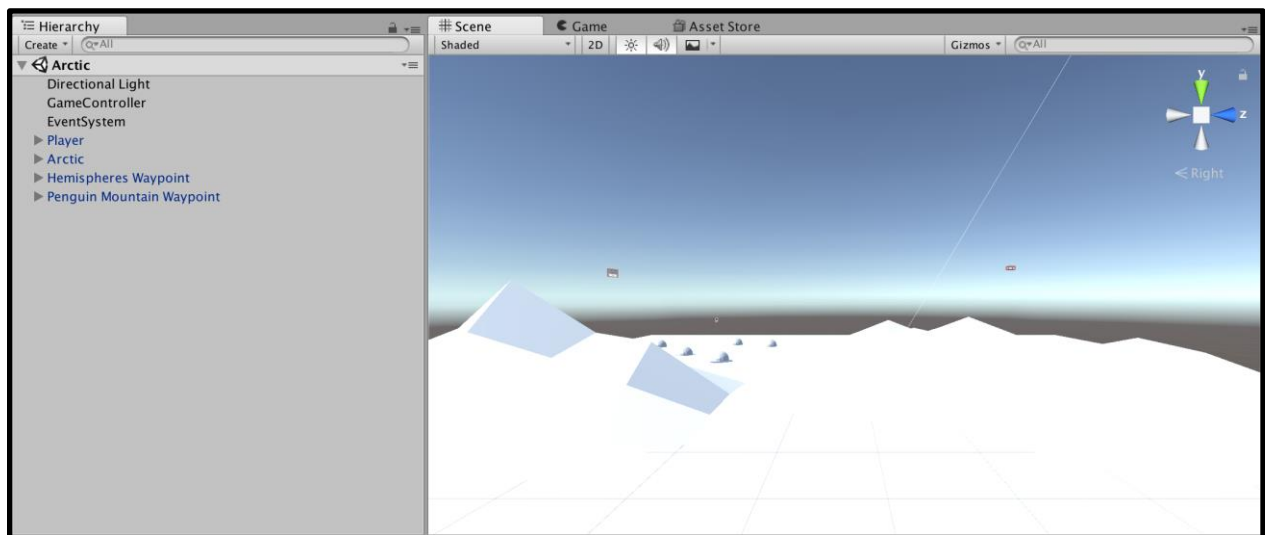
```

1 void LoadNextScene() {
2     Debug.Log ("Loading next scene");
3     // Get the current scene's index in the build order.
4     int currentSceneIndex = SceneManager.GetActiveScene ().buildIndex;
5     // If there are more scenes, load the next scene.
6     if (currentSceneIndex < SceneManager.sceneCountInBuildSettings - 1) {
7         SceneManager.LoadScene (currentSceneIndex + 1);
8     } else { // Otherwise, load the first scene.
9         SceneManager.LoadScene (0);
10    }
11 }

```

SceneManager is a class in the `Unity.SceneManagement` namespace. The current scene is returned by `SceneManager.GetActiveScene()`. `buildIndex` identifies the scene by the order in which the scenes were built. `SceneManager.sceneCountInBuildSettings` is the total number of scenes. `SceneManager.LoadScene()` loads a scene by name or by index.

After saving the current scene, create a new scene by clicking on File->New Scene. Drag Arctic from Assets/Models to the scene, or import your own model. Drag Player from Assets/Prefabs to the scene, and position it where you want the Player to start the tour. Drag Waypoint from Assets/Prefabs as many times as you want, and customize each Waypoint's name and description. Then drag the Waypoint objects into the Player's waypoints array in the inspector in the order you want them to be visited.

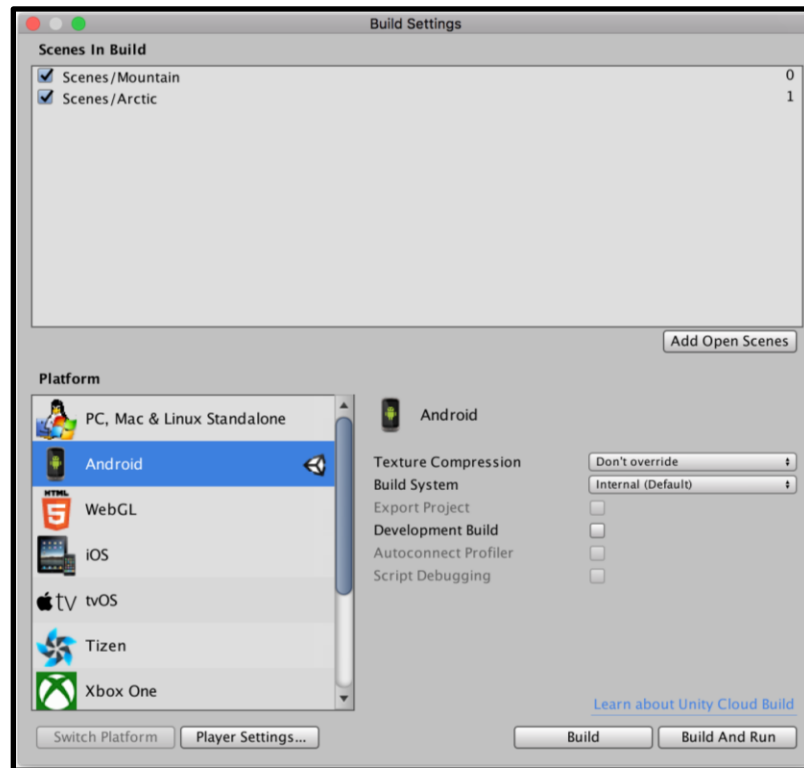


The second scene the in the project. This scene uses the Arctic model for the terrain and the Player and Waypoint prefabs.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

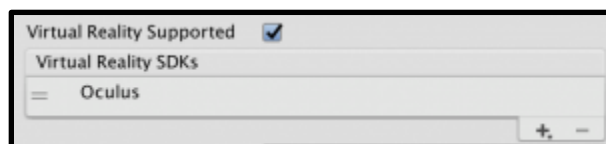
Building the Project

Once you're ready to try out the game, select File->Build Settings from the menu. Drag all the scenes you want to include from the project pane to the "Scenes in Build" window in Build Settings. Select Android as the platform. Select Android as the platform.



The Build Settings window, with all scenes loaded.

Click the "Player Settings" button to load settings in the Inspector pane. Enter a company name and product name. Scroll down to the Rendering section, and make sure "Virtual Reality Supported" is checked. Click "+" under "Virtual Reality SDKs" and add Oculus.



In the "Identification" section, give the game a package name. Typically, package names are reversed domain names, such as org.gamedevacademy.gearvr.vrtour. You

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

are not restricted to existing domain names. Chose a minimum API level of Android 6.0 “Marshmallow”, which is the lowest API level that supports Gear VR.

When you’re ready to test your game, click the “Build and Run” button in the File->”Build and Run” window to build an APK and run it on a connected phone.

Enjoy the tour!

Developing for the Gear VR Controller

Until recently, holding your hand to your temple to manipulate a small touchpad was how you had to use Gear VR headsets. Bluetooth gamepads allowed some Gear VR games to support more input options, but the incorporation of natural movement into the VR experience was missing.

In April 2017, Samsung released the Gear VR controller with built-in sensors that track orientation. In apps that take advantage of the controller’s degrees of freedom, the controller becomes a part of the VR experience, as a pointer, a gun, a magic wand, a golf club, a flight stick, or any other device that relies on rotation. You can see the functionality of the Gear VR controller in the Oculus Home app. When the controller is paired, and you are in the Oculus Home environment, you’ll see an image of the controller that rotates as you rotate the controller in your hand.

Once the Gear VR controller is paired, you don’t need to modify your apps to be able to use the basic touchpad functionality. However, to fully utilize the controller’s features, you will need to make some changes to your projects. Oculus provides the scripts and prefabs you will need in the Oculus Utilities for Unity package.

In this tutorial, we will use the Oculus Utilities package to create a shooting game that demonstrates controller functionality.

Source Code Files

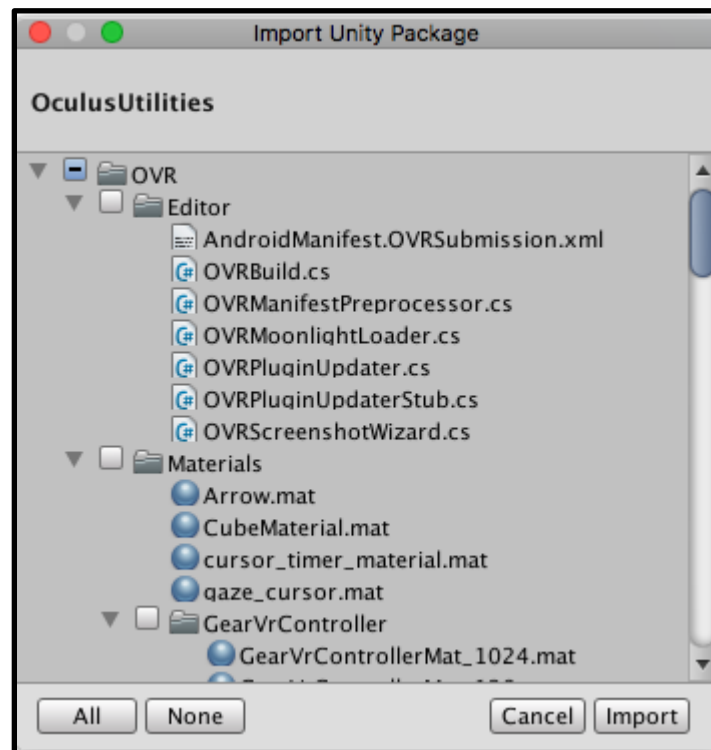
Download the complete project [here](#). This project was created in Unity 5.6.1.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

Setting Up the Project

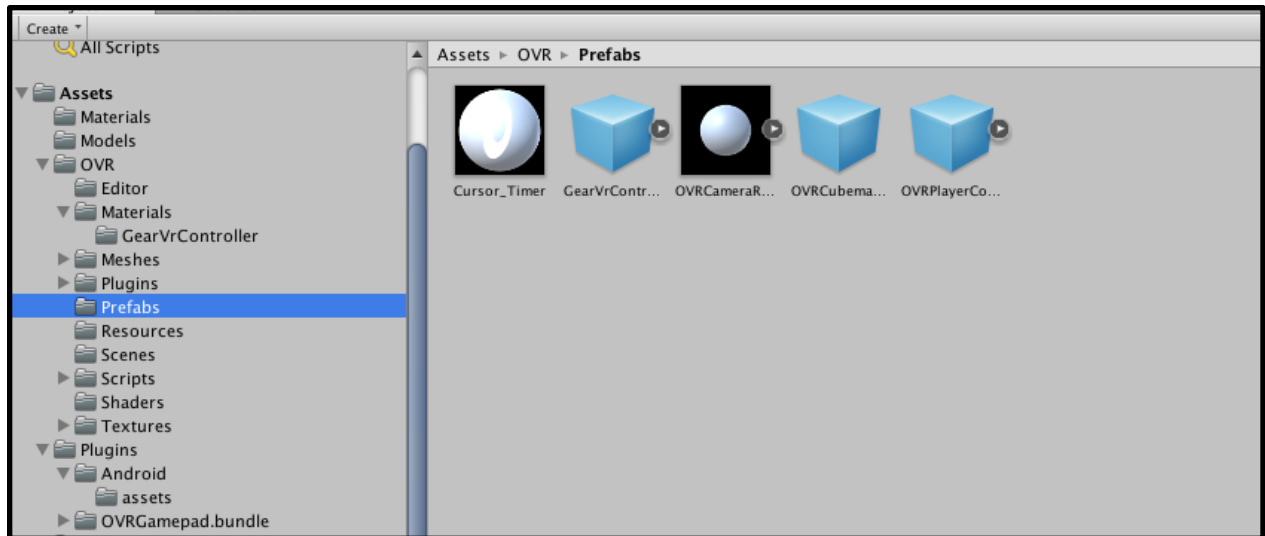
Download Oculus Utilities for Unity 5 from <https://developer.oculus.com/downloads/package/oculus-utilities-for-unity-5/> and unpack the zip file on your computer.

Create a new project in Unity. I named my project ShooterWithController. Import Oculus Utilities by selecting Assets -> Import Package -> Custom Package and choosing the package you extracted. If you are prompted to update the package, follow the prompts to install the update and restart Unity.

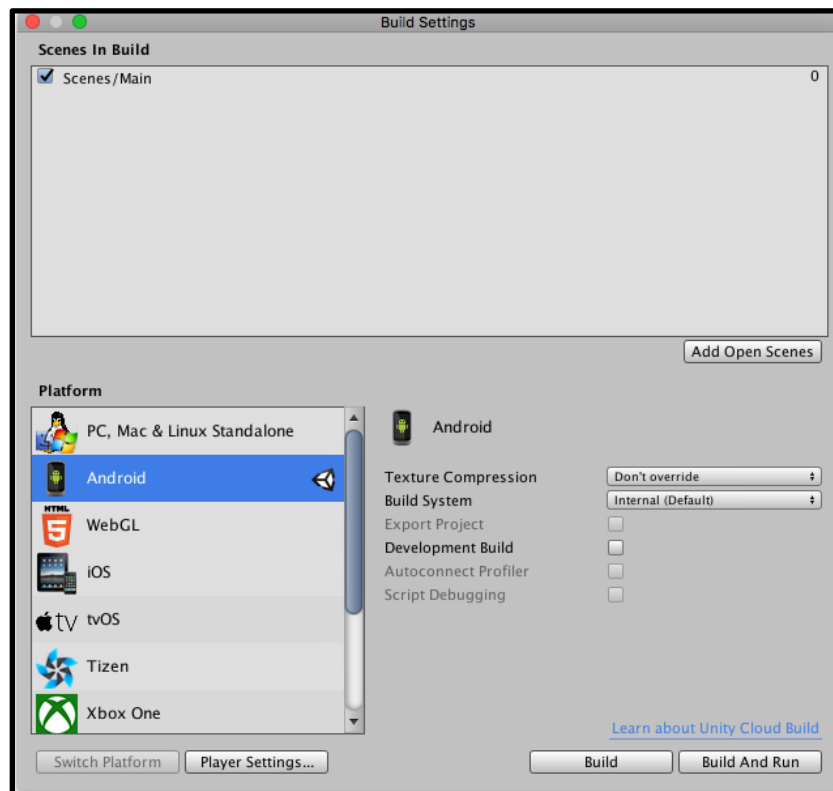


A directory named OVR will be added to your project's assets.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.



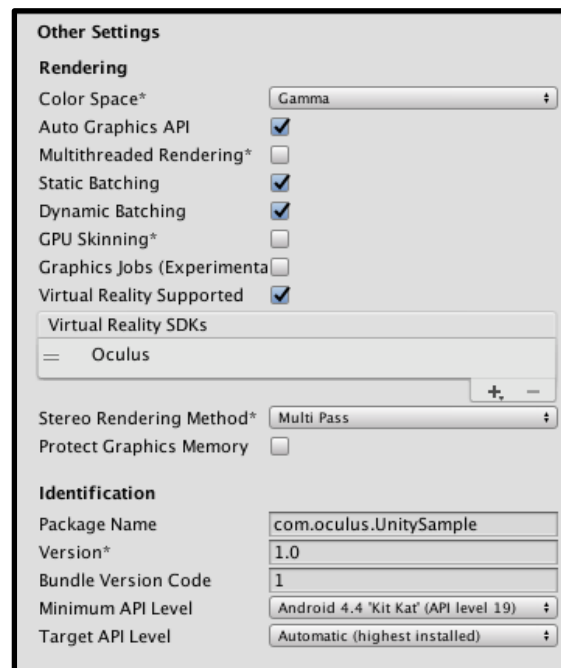
To prepare your project to be built, navigate to File -> Build Settings. Select Android as the platform.



Click on Player Settings to load PlayerSettings into the Inspector panel. Set a company name and project name. Under the “Other Settings” heading, select “Virtual Reality Supported.” Add

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

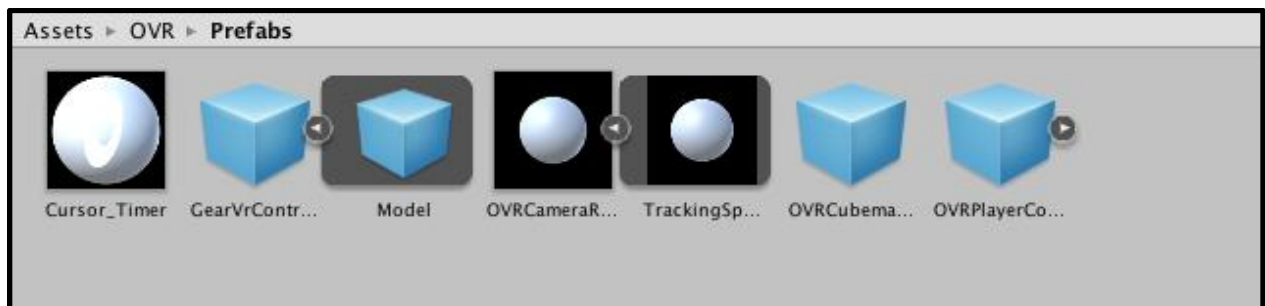
Oculus under Virtual Reality SDKs. Select Android 4.4 Kit-Kat (API Level 19) in the Minimum API Level field under the “Identification” heading. This is the minimum version of Android that supports the Gear VR.



Create the directory `Assets/Plugins/Android/assets` in your project, and copy your phone's oculussig file to this directory. See <https://dashboard.oculus.com/tools/osig-generator/> if you don't have an oculussig file for your phone.

Using OVRCameraRig

Let's take a look at the OVR directory that was imported into the Assets folder of your project. In `Assets/OVR/Prefabs`, OVRCameraRig is a prefab that replaces the default main camera.

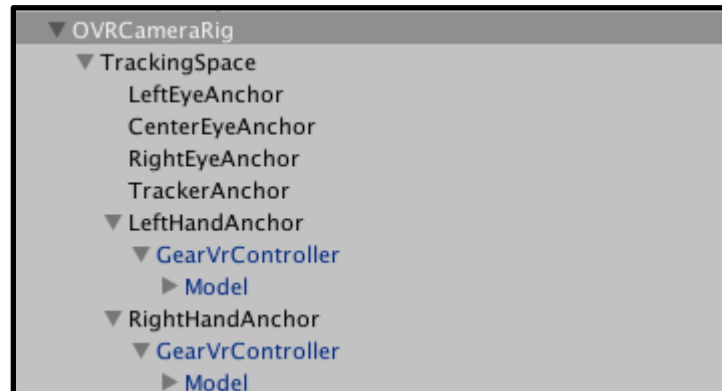


This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

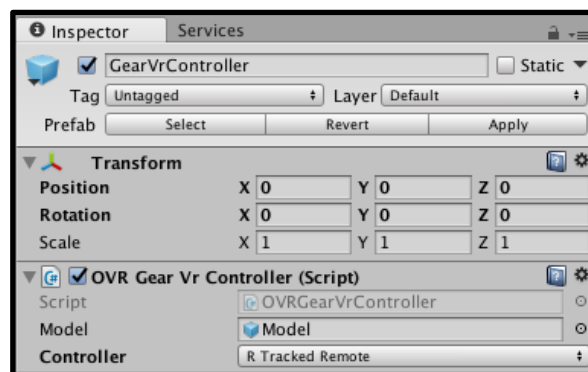
To use OVRCameraRig, drag the OVRCameraRig prefab to the scene. Delete the MainCamera that was created by default.

Another prefab, GearVRController, represents the controller. One of its components is a model of the controller itself. When active, the controller will be visible in the VR app.

Drag the GearVRController prefab to both LeftHandAnchor and RightHandAnchor under OVRCameraRig.

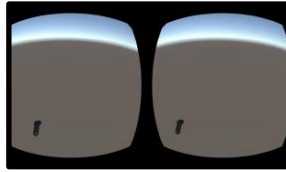


Select the instance of GearVRController under LeftHandAnchor. In the Inspector pane, set the Controller field of the OVR Game Controller script to “L Tracked Remote.” Set the Controller field of the GearVRController under RightHandAnchor to “R Tracked Remote.”



Which GearVRController is active depends on how you set up your controller when pairing it with your phone. If you set the controller to right-handed mode, the controller under RightHandAnchor will be active. In your app, it will appear on the right-hand side. If your controller is left-handed, the controller under LeftHandAnchor will be active.

To see the controller in action, save the scene and select File -> Build and Run. When the app is running, you should see a controller that rotates as you rotate the controller in your hand.



Understanding GearVrController

In actual games, you probably want the image of the controller to look different. We can customize the controller's appearance and behavior by building our own controller prefab based on the GearVRController prefab provided in the Oculus Utilities package. Let's take a look at the GearVRController prefab in [Assets/OVR/Prefabs](#).

The script [Assets/OVR/Scripts/Util/OVRGearVrController.cs](#) is attached to GearVRController. Open [OVRGearVRController.cs](#):

```
1 public class OVRGearVrController : MonoBehaviour
2 {
3     /// <summary>
4     /// The root GameObject that should be conditionally enabled depending on controller connection status.
5     /// </summary>
6     public GameObject m_model;
7
8     /// <summary>
9     /// The controller that determines whether or not to enable rendering of the controller model.
10    /// </summary>
11    public OVRInput.Controller m_controller;
12
13    private bool m_prevControllerConnected = false;
14    private bool m_prevControllerConnectedCached = false;
15
16    void Update()
17    {
18        bool controllerConnected = OVRInput.IsControllerConnected(m_controller);
19
20        if ((controllerConnected != m_prevControllerConnected) || !m_prevControllerConnectedCached)
21        {
22            m_model.SetActive(controllerConnected);
23            m_prevControllerConnected = controllerConnected;
24            m_prevControllerConnectedCached = true;
25        }
26
27        if (!controllerConnected)
28        {
29            return;
30        }
31    }
32 }
```

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

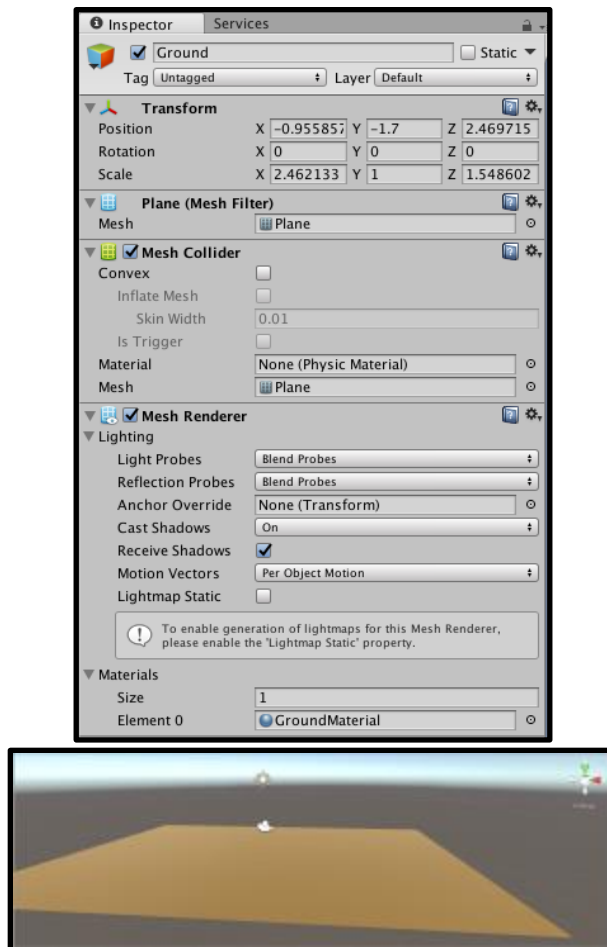
`m_model` is the physical manifestation of the controller. In the inspector, its value is set to the object Model that is nested under GearVrController. `m_controller` is an instance of an enumerated type, `OVRInput.Controller`, that is defined in OVRInput.cs. The enumerators of `OVRInput.Controller` are all the valid controller types. For the Gear VR controller, we are only concerned with left tracked remote or right tracked remote. When you set up instances of GearVrController, you need to populate the `m_controller` field with “R Tracked Remote” or “L Tracked Remote” in the inspector. The boolean `m_prevControllerConnected` keeps track of whether this controller was active.

`Update()` checks if `m_controller` is connected by calling `OVRInput.IsControllerConnected()`. If the resulting boolean has changed since the last update, `m_model` is activated or deactivated depending on whether `m_controller` is connected. For example, if the Gear VR controller is right-handed, when you start the app, the instance of `OVRGearVrController` set to “R Tracked Remote” will detect that its `m_controller` is connected. The image of the right-handed remote will become visible in the app. The instance set to “L Tracked Remote” will detect that its `m_controller` is not connected, and its `m_model` will not be activated. The left-handed remote will not be visible.

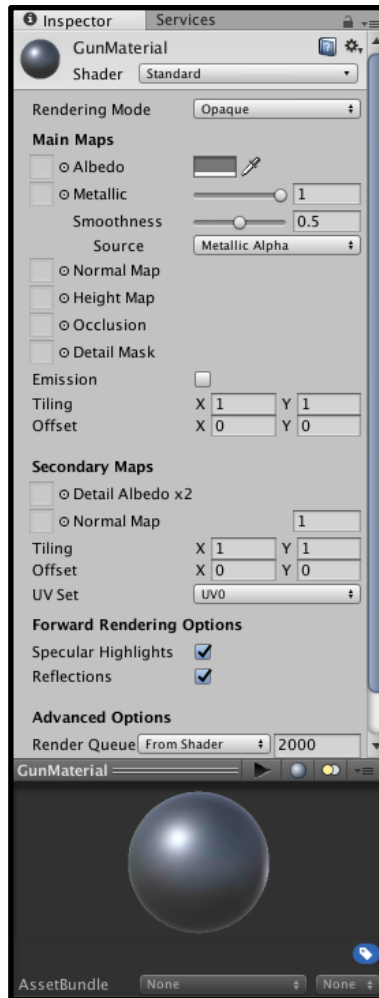
Creating a Gun

Now that we’ve explored the default GearVrController prefab, let’s create our own prefab that will turn the Gear VR controller into a gun. But first, let’s create a ground where the gun’s projectiles can fall.

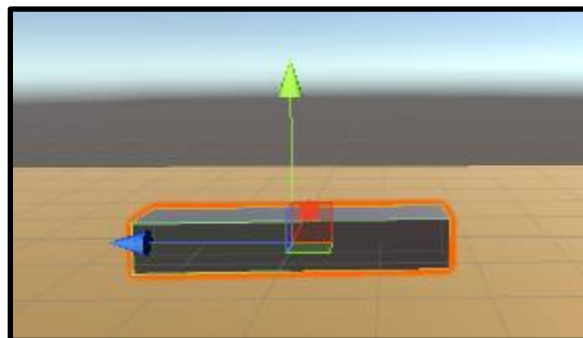
Create a plane and rename it “Ground.” Drag the OVRCameraRig up along the Y axis so that it is above the ground. Create a new material in `Assets/Materials` named “GroundMaterial.” Set its albedo to a brown color.



To prepare the gun, create a new material for it, [Assets/Materials/GunMaterial](#). Set the albedo color in the inspector and adjust the metallic slider to make the material look more realistic.

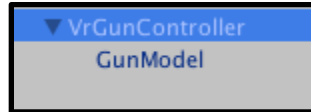


Create a cube in the scene named GunModel. Drag GunMaterial to this cube to apply it. Resize GunModel to a long, thin shape that's small enough to look like a handheld controller. When the transform widget is selected, make sure the darker blue axis lines up with the long side. The blue axis aligns with the forward face of the cube.



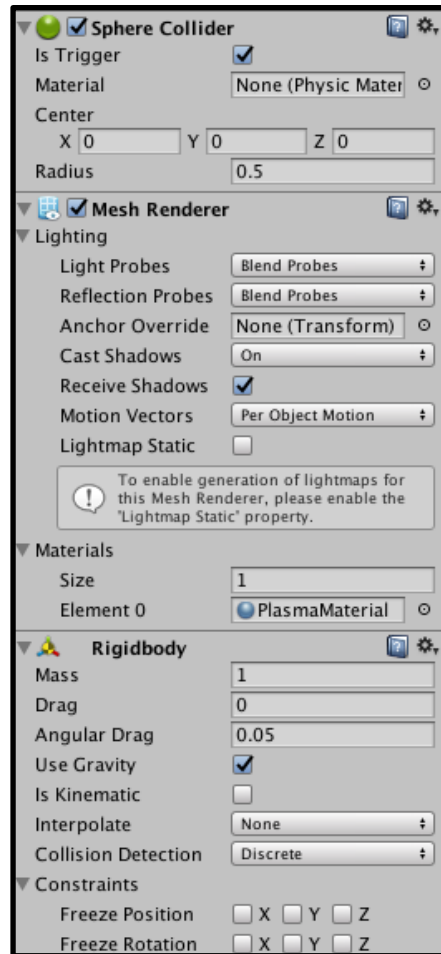
This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

Then create an empty object in the scene and name it VrGunController. VrGunController will be our replacement for GearVrController. Nest GunModel under VrGunController by dragging and dropping.



Next, we're going to create a bullet that can be fired by the gun. Create a sphere and resize it until it looks small enough to reasonably be fired by the gun. Rename it Bullet. Create a new material for it in [Assets/Materials](#), and apply the material by dragging it to Bullet.

Add a Rigid Body to the bullet, which will allow us to set a velocity and apply kinematics to it. Note that the bullet already has a sphere collider around it. Check the box next to "Is Trigger." We will use this field later to implement shooting enemies.



Drag Bullet to [Assets/Prefabs](#), and remove the original bullet you created in the scene. The Bullet prefab will be used to initialize bullets when you fire the gun.

Now we'll add scripts to VrGunController and GunModel. Add the script

[Assets/OVR/Scripts/Util/OVRGearVrController](#) to VrGunController. This allows VrGunController to be used as the Gear VR controller. Drag GunModel to the Model field. For now, don't populate the Controller field. We will set the handedness of the Controller when we assign an instance of VrGunController to one of the anchors in OvrCameraRig.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.



Create a new C# script, GunController, in [Assets/Scripts](#) :

```

1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class GunController : MonoBehaviour {
6
7     public GameObject projectilePrefab;
8
9     // Use this for initialization
10    void Start () {
11        this.transform.forward = Camera.main.transform.forward;
12    }
13
14    public void ShootProjectile(float projectileSpeed)
15    {
16        GameObject projectile = GameObject.Instantiate(projectilePrefab);
17        projectile.transform.position = this.transform.position;
18
19        Rigidbody rb = projectile.GetComponent<Rigidbody>();
20        rb.velocity = this.transform.forward * projectileSpeed;
21
22    }
23 }
24

```

[Start\(\)](#) , called when the GunController is initialized, points the GunModel in same direction as the camera. Subsequent orientations are set by the player moving the controller.

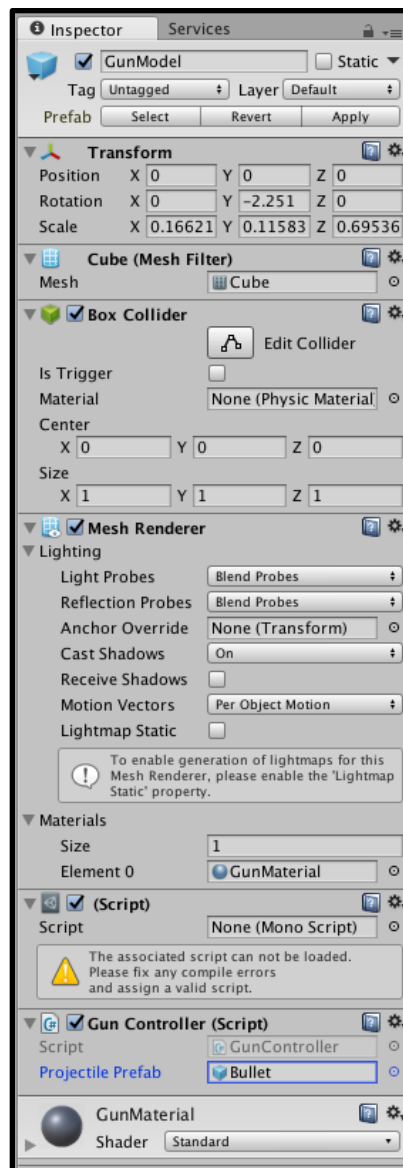
This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

© Zenva Pty Ltd 2018. All rights reserved

`ShootProjectile(projectileSpeed)` is called when the Gear VR controller's trigger is pulled.

`ShootProjectile()` instantiates an instance of the projectile prefab and fires it in the same direction the gun is pointing with a speed determined by `projectileSpeed`.

Drag GunController to the GunModel object to add it as a component. In the inspector, set "Projectile Prefab" to Bullet.



This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

Let's add the code to detect a trigger pull to `Assets/OVR/Scripts/Util/OVRGearVrController`, which we've attached to VrGunController. Open the script and add the following lines to the end of the `Update()` method:

```
1 if (OVRInput.GetUp(OVRInput.Button.PrimaryIndexTrigger)) {  
2     if (m_gunController != null) {  
3         Debug.Log ("OVRGearVrController: Shooting projectile");  
4         m_gunController.ShootProjectile (projectileSpeed);  
5     } else {  
6         Debug.Log ("OVRGearVrController: m_gunController is null");  
7     }  
8 }
```

Also add the following data members to the OVRGearVrController class:

```
1 public GunController m_gunController;  
2 public float projectileSpeed = 20f;
```

In the inspector panel, set the "Gun Controller" field to GunModel. GunModel's GunController component will be the value of `m_gunController`. When the Gear VR controller's trigger button is pressed and released, if `m_gunController` is active, `ShootProjectile()` is called with a speed of `projectileSpeed`, which can also be set in the inspector. From trial and error, 20 is a reasonable speed.



Drag VrGunController to [Assets/Prefabs](#), and remove VrGunController from the scene for now. We will add the prefab back to the scene in the next step.

You should still have an instance of OVRCameraRig in your scene. Remove GearVrController from OVRCameraRig -> TrackingSpace -> LeftHandAnchor and OVRCameraRig -> TrackingSpace -> RightHandAnchor. Then drag and drop VrGunController from [Assets/Prefabs](#) twice, once to nest under LeftHandAnchor, and once to nest under RightHandAnchor.



Select the VrGunController under LeftHandAnchor. In the inspector, set Controller under “OVR Gear Vr Controller (Script)” to “L Tracked Remote.”

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.



Similarly, set the VrGunController instance under RightHandAnchor to “R Tracked Remote.”

When you build and run the project on your phone with the controller paired, you should see an image of the GunModel that you can rotate by rotating your controller. When you click the trigger button, bullets will be fired from the gun.

Adding Enemies

Now let’s add enemies we can target with the controller-enabled gun. The attached project includes a Blender model, Assets/Models/GhostModel.fbx , that can be used as the prefab model for our enemies. Copy this file to into your project’s Assets/Models folder.

Create a new GameObject, Ghost. From the Tag dropdown in the inspector, select “Add Tag” and apply it to Ghost.

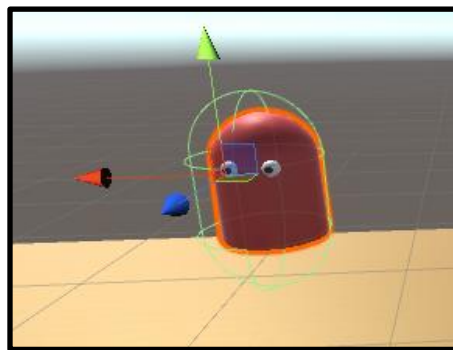


Drag GhostModel to the Ghost object to nest it.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.



In the scene view, use the rotation widget to rotate GhostModel until it is upright, and scale it to a size that looks reasonable in the scene. Add a capsule collider to Ghost. Click “Edit Collider” and resize the green capsule until it approximately encloses the ghost.



Create a new C# script, `Assets/Scripts/EnemyController`, and attach it to Ghost. EnemyController will control the ghost’s movement:

```

1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class EnemyController : MonoBehaviour {
6
7     float speed = 2f;
8
9     // Update is called once per frame
10    void Update () {
11        transform.Translate (-1 * Vector3.forward * speed * Time.deltaTime, Space.World);
12    }
13
14    public void SetSpeed(float newSpeed) {
15        speed = newSpeed;
16    }
17
18 }

```

The `Update()` method moves the ghost forward each frame update.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

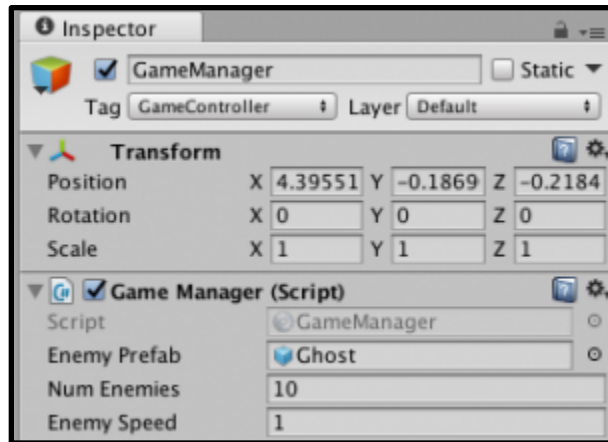
Now drag Ghost from the hierarchy pane into `Assets/Prefabs`, and remove the Ghost you created in the scene. Ghosts will be created programmatically at the start of the game by a GameManager object.

Create a new empty GameObject called GameManager. Attach a new C# script to it,

`Assets/Scripts/GameManager`:

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class GameManager : MonoBehaviour {
6
7     public GameObject enemyPrefab;
8     public int numEnemies = 10;
9     public float enemySpeed;
10
11     // Use this for initialization
12     void Start () {
13         SpawnEnemies ();
14     }
15
16     // Update is called once per frame
17     void Update () {
18
19     }
20
21     void SpawnEnemies() {
22         float startX = -20f;
23         float xCoord = startX;
24         float zCoord = 15;
25         float height = 1.5f;
26         int enemiesInRow = 5;
27         for (int i = 0; i < numEnemies; i++) {
28             Quaternion rotation = Quaternion.identity;
29             GameObject enemy = Instantiate (enemyPrefab, new Vector3 (xCoord, height, zCoord), Quaternion.LookRotation(-1 * Camera.main.transform.forward));
30             enemy.GetComponent<EnemyController>().SetSpeed (enemySpeed);
31             xCoord += 10f;
32             // Increment the z coordinate if a row is filled.
33             if ((i + 1) % enemiesInRow == 0) {
34                 xCoord = startX;
35                 zCoord += 2f;
36             }
37         }
38         //newEnemy.transform.forward = -1 * Camera.main.transform.forward;
39     }
40 }
41 }
```

The data member `enemyPrefab` is the template used to create enemies. In the inspector, populate “Enemy Prefab” with Ghost. The number of enemies to spawn and the speed of the enemies can also be set in the inspector.



`SpawnEnemies()`, called when GameManager initializes, creates `numEnemies` enemies in equally spaced rows.

We still need to be able to detect when an enemy is hit by a bullet. Create a new C# script, `Assets/Scripts/BulletController`, and add it to the Bullet prefab we created earlier:

```

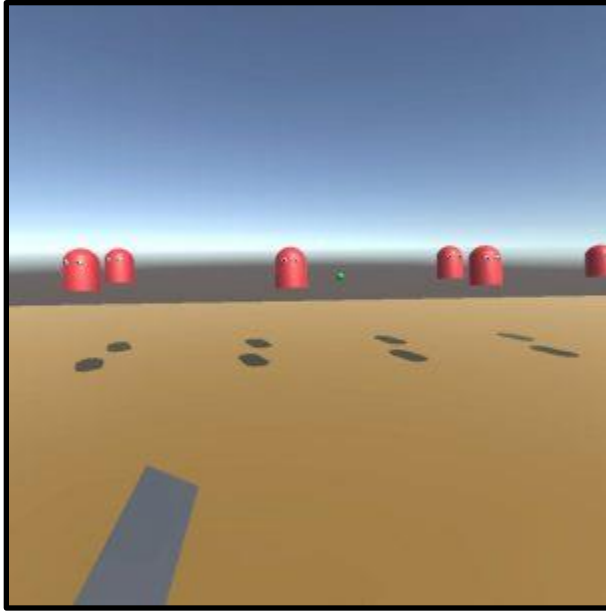
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class BulletController : MonoBehaviour {
6
7     public float maxDistance = 100f;
8
9     // Use this for initialization
10    void Start () {
11
12    }
13
14    // Update is called once per frame
15    void Update () {
16        // Self-destruct if this bullet exceeds maxDistance from the camera.
17        if (Vector3.Distance (transform.position, Camera.main.transform.position) > maxDistance) {
18            Destroy (gameObject);
19        }
20    }
21
22    // Detect collisions.
23    void OnTriggerEnter(Collider other) {
24        Debug.Log ("BulletController::OnTriggerEnter");
25
26        // If the bullet has collided with an enemy, destroy both the enemy and itself.
27        if (other.gameObject.CompareTag("Enemy")) {
28            Debug.Log ("BulletController::OnTriggerEnter: Collided with enemy");
29            Destroy(other.gameObject);
30            Destroy (gameObject);
31        }
32    }
33 }
34 }

```

Earlier, we made Bullet's sphere collider a trigger. `OnTriggerEnter()` is called when another collider enters the trigger — in other words, when a bullet hits another object with a collider. If the other object is tagged "Enemy", we destroy both the object and the bullet.

`Update()` checks how far the bullet is from the main camera. The bullet is destroyed if this distance exceeds `maxDistance`. This avoids filling the scene with stray bullets. Note that we do not need to explicitly move the bullet because we have already assigned a velocity to the bullet's rigid body in `GunController::ShootProjectile()`.

At this point, we have a game that uses the Gear VR controller as a gun to shoot floating ghosts. There is plenty of room for improvement, but you have the basics of how to use Unity Utilities to support the Gear VR controller.



This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

© Zenva Pty Ltd 2018. All rights reserved

Flight Simulation with the Gear VR Controller

In the previous chapter, we learned how to use the Oculus Utilities package to utilize the Gear VR controller as a gun or pointing device. Now we'll use the controller as a joystick to control a spaceship. Instead of reusing and modifying the Oculus Utilities prefabs, we will be directly accessing the controller's orientation to manipulate a spaceship flying over an alien planet.

Source Code Files

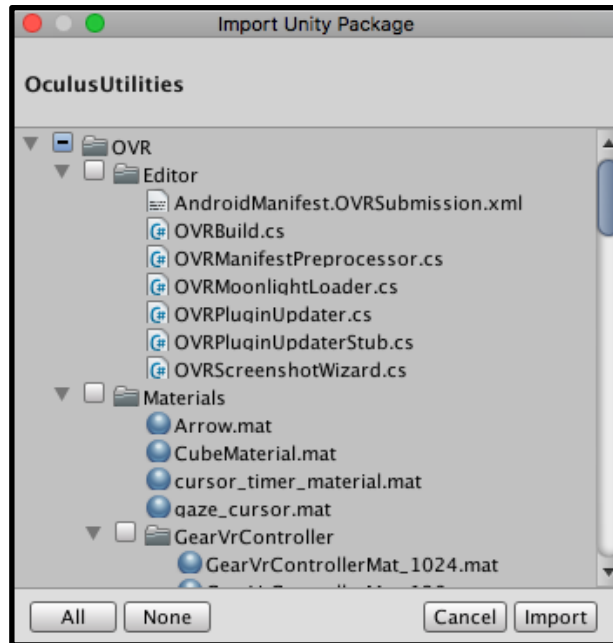
You can download the Unity project for this tutorial [here](#). The OVR directory created by Oculus Utilities is included. The spaceship model included in the project's assets was created by Liz Reddington. It is licensed under CC-BY 3.0 and available at [Spaceship – Poly on Google](#)

Setting Up the Project

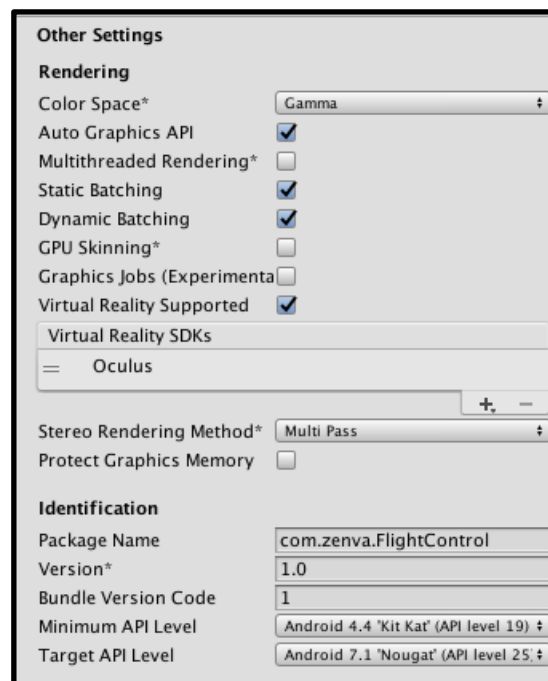
To get started, create a new project in Unity. Create the folder Assets/Plugins/Android/assets , and copy your phone's Oculus signature file there. See [Oculus Signature File \(osig\) and Application Signing](#) to create a new signature file.

Download Oculus Utilities for Unity 5 from the [Oculus Utilities for Unity Site](#) and unpack the zip file on your computer. Import Oculus Utilities into your project by selecting Assets -> Import Package -> Custom Package and choosing the package you extracted. If you are prompted to update the package, follow the prompts to install the update and restart Unity.

A directory named OVR will be extracted into your project.



To prepare your project to be built, navigate to File -> Build Settings. Select Android as the platform. Click on Player Settings. Set a company name and project name. Under the “Other Settings” heading, select “Virtual Reality Supported” and add Oculus to “Virtual Reality SDKs.” Select Android 4.4 Kit-Kat (API Level 19) in the Minimum API Level field under the “Identification” heading. This is the minimum version of Android that supports the Gear VR.

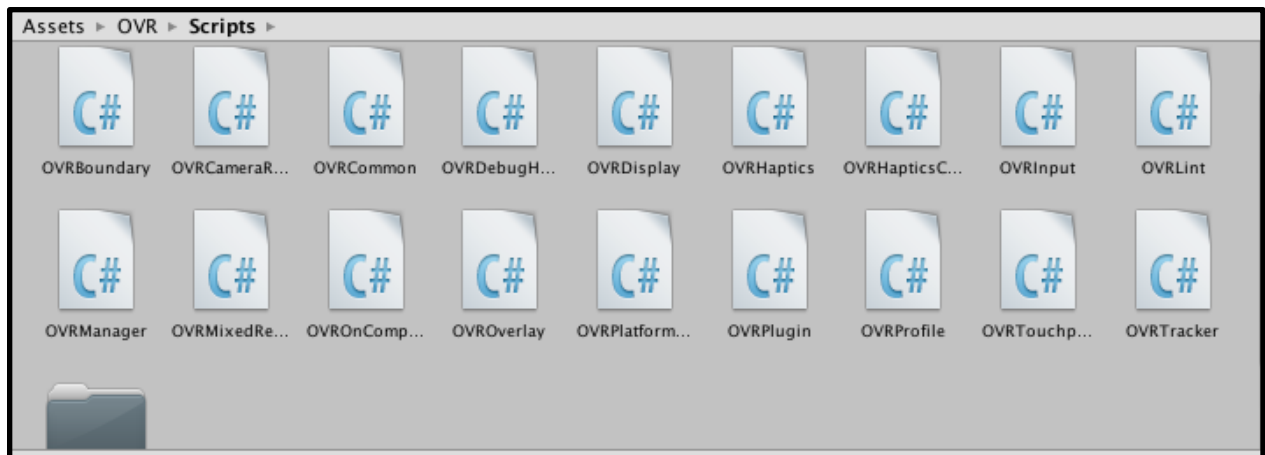


This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

In the previous chapter, we used the `OVRCameraRig` prefab from the Oculus Utilities (OVR) package. The scripts that detected controllers were already attached to the prefabs. Now that we are not using the prefabs, we need to attach the right scripts and write additional code to handle controller rotation.

First, let's take a look at the code provided by Oculus Utilities.

Open the script `Assets/OVR/Scripts/OVRInput.cs`. In Unity's project window, note that C# scripts are displayed without the `.cs` extension, `OVRInput.cs` looks like `OVRInput`. When you create your own scripts, and the script's icon appears in the project window, don't add the `.cs` extension manually. If you do, your script's real name will look like `MyScript.cs.cs`, and Unity will not be able to find it.



The documentation in the `OVRInput` script describes it as a unified input system for Oculus controllers and gamepads. The section that concerns us the most is the `Controller` enumerated type that maps to `OVRPlugin.Controller`:

```

1 public enum Controller
2 {
3     None           - OVRPlugin.Controller.None,           ///< Null controller.
4     LTouch         - OVRPlugin.Controller.LTouch,         ///< Left Oculus Touch controller. Virtual input mapping differs from the c
5     RTouch         - OVRPlugin.Controller.RTouch,         ///< Right Oculus Touch controller. Virtual input mapping differs from the
6     Touch          - OVRPlugin.Controller.Touch,          ///< Combined Left/Right pair of Oculus Touch controllers.
7     Remote         - OVRPlugin.Controller.Remote,         ///< Oculus Remote controller.
8     Gamepad        - OVRPlugin.Controller.Gamepad,        ///< Xbox 360 or Xbox One gamepad on PC. Generic gamepad on Android.
9     Touchpad       - OVRPlugin.Controller.Touchpad,       ///< GearVR touchpad on Android.
10    LTrackedRemote  - OVRPlugin.Controller.LTrackedRemote, ///< Left GearVR tracked remote on Android.
11    RTrackedRemote  - OVRPlugin.Controller.RTrackedRemote, ///< Right GearVR tracked remote on Android.
12    Active         - OVRPlugin.Controller.Active,         ///< Default controller. Represents the controller that most recently regis
13    All            - OVRPlugin.Controller.All,             ///< Represents the logical OR of all controllers.
14 }

```

In `Assets/OVR/Scripts/OVRPlugin.cs`, `OVRPlugin.Controller` is an enumeration of values that represent each type of controller.

```

1 public enum Controller
2 {
3     None           - 0,
4     LTouch         - 0x00000001,
5     RTouch         - 0x00000002,
6     Touch          - LTouch | RTouch,
7     Remote         - 0x00000004,
8     Gamepad        - 0x00000010,
9     Touchpad       - 0x08000000,
10    LTrackedRemote  - 0x01000000,
11    RTrackedRemote  - 0x02000000,
12    Active         - unchecked((int)0x80000000),
13    All            - ~None,
14 }

```

The Gear VR controller can be either `LTrackedRemote` or `RTrackedRemote`, depending on whether it is set up left-handed or right-handed. Currently, connecting two Gear VR controllers is not supported, but this could change in the future.

Representing controller types with numbers lets us use binary operations to store and extract multiple controller types in one variable. When we compute the bitwise OR of two binary numbers, we preserve all the bits in both masks. For example, the bitwise OR of 0001 and 0010 is 0011. Both 1's are preserved in the result. When we compute the bitwise AND of 0001 and 0011, we get back 0001.

```

    0001
|) 0010
---
    0011
&) 0001
---
    0001

```

This technique lets us combine controller types by computing the bitwise OR of two or more `OVRPlugin.Controller` values. So, we can combine `LTrackedRemote` and `RTrackedRemote` by computing their bitwise OR. `OVRPlugin.GetConnectedControllers()` returns a bitwise OR of all connected controllers. We can determine if `LTrackedRemote` or `RTrackedRemote` is one of them by using bitwise operations:

```
1 gearController = connectedControllers & (OVRInput.Controller.LTrackedRemote | OVRInput.Controller.RTrackedRemote);
```

If, say, `LTrackedRemote` is connected, the result of the bitwise AND will be `LTrackedRemote`, and this value will be stored in the variable `gearController`. The same applies for `RTrackedRemote`. If neither is connected, `gearController`'s value will be `OVRInput.Controller.None`.

Once the active Gear VR controller is identified, we can get its rotation quaternion by calling `OVRInput.GetLocalRotation()`. Then we can use this rotation to manipulate our spaceship.

ShipController

Our code to move the spaceship will go in the script `ShipController.cs`. Create the folder `Assets/Scripts` in your project, and in this folder create the `ShipController` script.

Start by declaring `ShipController`'s data members:

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

```

1 public class ShipController : MonoBehaviour
2 {
3     [SerializeField] float speed = 0.0f;
4     [SerializeField] Transform vrCameraContainer;
5     private OVRInput.Controller gearController;
6     private Transform target; // Target marker where the spaceship will move in each frame.
7     private float distanceToCamera;
8
9     // ...
10 }

```

The values of fields that are marked as `[SerializeField]` can be set in the inspector, even though their scope is private. The variable `speed` is the speed at which the camera container and ship will move. `vrCameraContainer` is the transform of the game object that will contain the main camera.

The variable `gearController` will store the value of whichever controller is active. The `target` represents the target position of the spaceship at each frame. Lastly, `distanceToCamera` stores the distance between the spaceship and `vrCameraContainer`.

When the game starts, we compute `distanceToCamera` in the `Awake()` method, which is called automatically before the game starts, but after all objects are initialized. We will also initialize `target` to the initial transform of the spaceship.

```

1 void Awake ()
2 {
3     target = transform;
4     distanceToCamera = Vector3.Distance(vrCameraContainer.position, transform.position);
5 }

```

We need code to determine which controllers are connected, so create a method, `SetController()`.

```

1 private void SetController()
2 {
3     OVRInput.Controller connectedControllers = OVRInput.GetConnectedControllers ();
4     Debug.Log (connectedControllers);
5     gearController = connectedControllers & (OVRInput.Controller.LTrackedRemote | OVRInput.Controller.RTrackedRemote);
6 }

```

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

`OVRInput.GetConnectedControllers()` returns all connected controllers. The `gearController` is set to either `LTrackedRemote` or `RTrackedRemote` or none, using the bit manipulation technique we discussed in the previous section.

We will also need a method that returns the rotation of the connected controller:

```
1 private Quaternion GetOrientation()
2 {
3     return OVRInput.GetLocalControllerRotation (gearController);
4 }
```

Create a method, `MoveShip()`, that will be called to update the spaceship's position each frame.

```
1 void MoveShip ()
2 {
3     Quaternion controllerRotation;
4     Vector3 origPosition = transform.position;
5
6     SetController ();
7     controllerRotation = GetOrientation ();
8
9     // Move the camera.
10    vrCameraContainer.Translate (Vector3.forward * speed * Time.deltaTime);
11
12    // Calculate the ship's target position.
13    target.position = vrCameraContainer.position + (controllerRotation * Vector3.forward) * distanceToCamera;
14
15    // Interpolate the ship's rotation to match the controller's rotation.
16    transform.rotation = Quaternion.Slerp (transform.rotation, controllerRotation, speed * Time.deltaTime);
17    // Move ship to the target transform.
18    transform.position = Vector3.Lerp (transform.position, target.position, speed * Time.deltaTime);
19
20 }
```

Let's see how this works. First, `SetController()` sets the value of `gearController`. We need to do this every frame because the controller can be disconnected or reconnected at any time. The `GetOrientation()` method acquires the controller's rotation quaternion.

Our `vrCameraContainer` is moved forward by a distance determined by the speed. The spaceship will also move forward by the same speed, but the direction of its movement is determined by the controller. Each frame, the spaceship will move to the position of the target transform, which is computed by multiplying the controller's rotation by the forward vector to determine a new direction. This product is multiplied by `distanceToCamera` to maintain the distance between the spaceship and the camera. The `Vector3.Lerp()` linearly interpolates the spaceship's position to the target's position for a smooth transition.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

Each frame, `MoveShip()` is called by an `Update()` method that runs automatically.

```
1 void Update()
2 {
3     MoveShip();
4 }
```

Setting Up the Game Assets

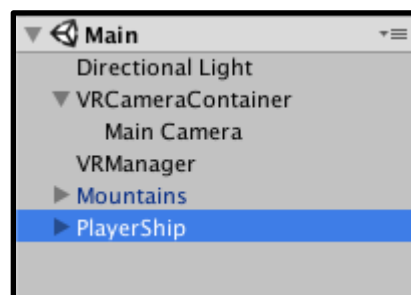
Now that our code is written, we need to put together the game's assets. We will need a spaceship model as well as terrain or flying objects to make the game more interesting and to show that the spaceship actually moves.

For now, I'm using one of the terrain models from the [VirtualTour tutorial](#). This file, `Mountains.fbx`, is also available in the included project file. Create a folder, `Assets/Models`, in your project, and copy `Mountains.fbx` there.

The spaceship model is a low-poly asset downloaded from Google Poly. Its files, `Lo_poly_Spaceship_03_by_Liz_Reddington.mtl` and `Lo_poly_Spaceship_03_by_Liz_Reddington.obj`, are also included in the attached project. Add these files to `Assets/Models`.

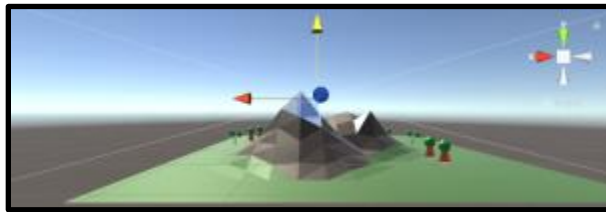
The camera will be located a fixed distance behind the spaceship to provide a third-person view. In VR, we never directly move the camera because the camera is controlled by the player's head. Instead, we'll nest the camera object inside another container.

Create an empty game object named `VRCameraContainer`. Drag `Main Camera` into `VRCameraContainer` to nest it.

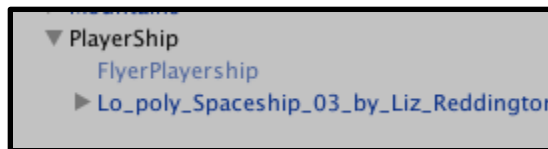


Use the transform widget and the arrows to move `VRCameraContainer` to an appropriate height above the terrain.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.



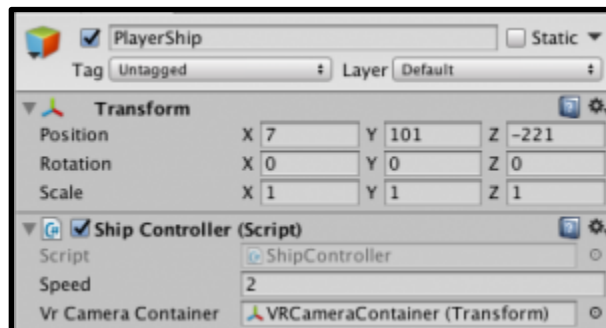
We also need to create a container for the spaceship model. Create an empty object, *PlayerShip*. Drag the spaceship model into the hierarchy pane, nesting it under *PlayerShip*.



Scale down the spaceship so it looks reasonable, and adjust its z-coordinate to position it in front of the camera.



Attach the ShipController script to the PlayerShip object.



Drag the VRCameraContainer object to the “Vr Camera Container” field in the inspector, and set a speed.

Finding the Controller

If you run the game at this point, you'll find that the ship doesn't move along with the controller. What's going on?

Let's take a look at how the game knows that a controller is connected.

`OVR/Scripts/OVRInput.cs` determines which controllers are connected in the `Update()` and `FixedUpdate()` methods. However, `OVRInput` is a static class that is not attached to any game object, so `OVRInput`'s update methods are never called. No matter what you do, the game won't see your controller.

To resolve this situation, we'll need to create a script that calls `OVRInput.Update()` and `OVRInput.FixedUpdate()`, and attach this script to an object. Create an empty game object named VRManager. Create a new C# script, `Assets/Scripts/VRInputManager`, with the following code:

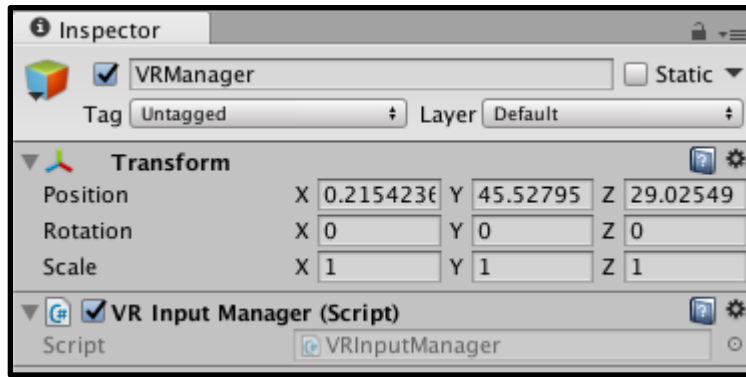
```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class VRInputManager : MonoBehaviour
6 {
7     // Update is called once per frame
8     void Update () {
9         if (VRManager.instance == null) {
10             OVRInput.Update ();
11         }
12     }
13
14     void FixedUpdate() {
15         if (VRManager.instance == null) {
16             OVRInput.FixedUpdate ();
17         }
18     }
19 }
```

`VRInputManager.Update()` calls `OVRInput.Update()`, but only if `VRManager` does not exist.

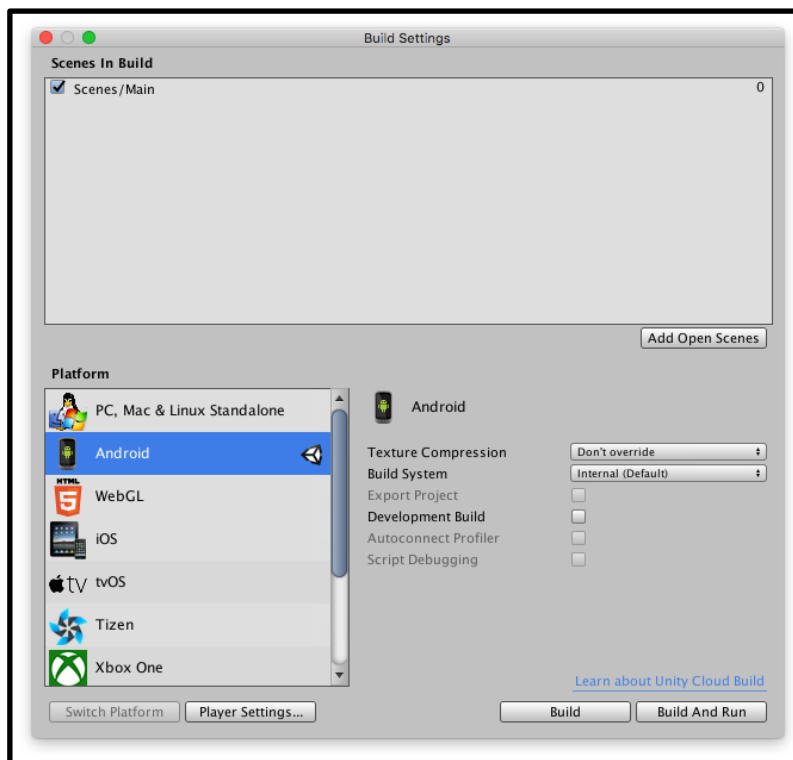
`VRManager` is another object included in Oculus Utilities that calls `OVRInput`. Since we don't need most of its functionality, we are not using it.

Attach the `VRInputManager` script to the VRManager object.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.



To try out the project, save the scene. I created a folder named [Assets/Scenes](#) and saved the scene as Main. In the Build Settings window, make sure the scene's name is checked in the Scenes in Build box. With your phone connected, click the Build and Run button, and choose a name for the APK. I created a [Build](#) folder and saved the APK file there.



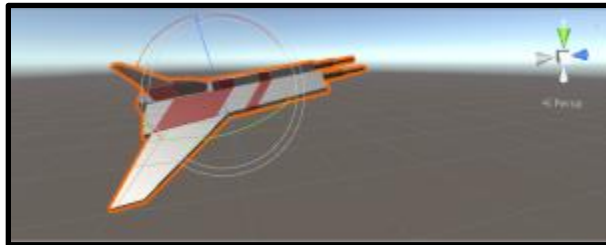
When the app loads, plug your phone into your headset to run it. If your controller is paired with your phone, press and hold down the home button to activate it.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

Rotation

When you run the app, you'll notice that the spaceship's position changes with the controller, but its nose always points forward, and it is always level.

To make the spaceship move more realistically, we need to change its rotation as the controller moves.



We want the spaceship to be able to rotate as it moves.

I tried various angle calculations, but it turned out that the most realistic-looking angles occurred when I simply used the rotation of the controller as the target rotation of the spaceship.

In the script `ShipController.cs`, we obtain a quaternion from `OVRInput.GetLocalRotationController()` that represents the rotation of the controller.

Quaternions are composed of four numbers that together represent a rotation in a sphere. Conceptually, quaternions are complex — in fact, they are an extension of complex numbers — but Unity provides abstractions that allow you to rotate an object using quaternions in a few lines of code.

Unity provides a function, `Quaternion.Slerp()`, that performs a spherical linear interpolation, or *slerp*, to rotate an object smoothly from one quaternion to another. *Slerp* is an interpolation along a spherical arc. The other type of interpolation, linear interpolation, or *lerp*, interpolates along a straight line. *Lerp* may run faster, but *slerp* provides smoother rotations.

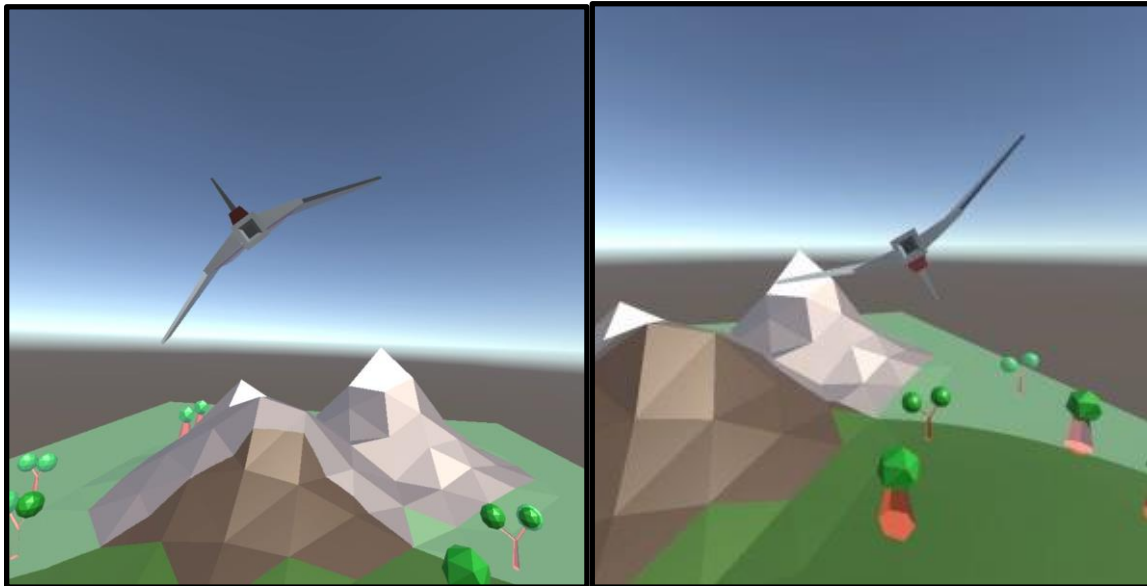
To add rotation, add the following line to `ShipController.MoveShip()` before the `Vector3.Lerp()` call:

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.

```
1 transform.rotation = Quaternion.Slerp (transform.rotation, controllerRotation, speed * Time.deltaTime);
```

This code interpolates the ship's original orientation to the controller's orientation.

Now try running the game. You'll notice that as you move your controller, the spaceship rolls, pitches, and banks as it changes direction.



Conclusion

Now that you have a starting point for using the Gear VR controller as a flight stick, you can use these techniques to build your own flight simulations and combat games. With the controller, you can control a vehicle more intuitively while you move your head to look around. Using the controller as a steering device provides a new level of freedom and realism to Gear VR games.

This book is brought to you by Zenva - Enroll in our [Virtual Reality Mini-Degree](#) to learn and master virtual reality.