VIRTUAL REALITY
GAME DEVELOPMENT

West University of Timișoara

Faculty of Mathematics and Informatics

LAURA ANDREEA CRISTESCU

LAURA.CRISTESCU96@E-UVT.RO

CEZAR IOAN VULCU

CEZAR.VULCU94@E-UVT.RO

Abstract

This documentation will focus on the abstract and summary explanation of how we created the VR Game in Unity, the hardships we faced and why we choose the Google Cardboard and VR Glasses device, that work notably on Android – with slight adjustments for IOS and not the Oculus Rift Dev. Kit.

Keywords: *UNITY, C#, ZOMBIE, VR, GOOGLE CARDBOARD, SDK, ANDROID, GAZE INPUT*

Content

1 Why did we choose to do this project?

2 VR Introduction
   a. Enabling VR in Projects
   b. Preview Your VR Project
   c. Hardware and Software Recommendations

3 Getting started with VR Development
   a. The basics
   b. Creating the VR Project

4 Interaction in VR
   a. Overview

5 User Interfaces for VR
   a. UI resolution and appearance
   b. Types of UI

6 Movement in VR

7 Deploying the Project and optimization for VR

1 *Why did we choose to do this project?*

The theme we choose reflects our passion for gaming. Most of our free time has been spent playing games for a large portion of our lives. However, as we got older, we've noticed some changes. We feel less and lees inclined to play hours of video games, but our love for them has seemed to grow. For instance, we used to play hours on end of Skyrim Legendary Edition, never leaving our chairs except for small breaks. Now after only two hours of playing games, we feel content for a while. Honestly, at first, we sensed worry. Video games defined a part of our identity, they were the first thing we become attached to. The thought of losing them as out passions made us start to question a lot about ourselves. However, we had a revelation. As we were watching a gameplay for Dead by Daylight, we felt like the game made an impact on us, and such we thought about other games that gave us such feeling – like Left for Dead 2, Sniper Elite v2. We felt that as we got older, our love for video games hasn't changed, but the way we appreciate them has. The reason we would spend so much time playing video games was because they would fill a stimulus we were missing, like adventure or competition.

As out love for gaming got even bigger, our passion for development also grew alongside it. Now we decided to give change a try and make our own video game. As the gaming industry became bigger, so did the UI. Virtual Reality, or VR, is the latest buzzword in the wonderful world of technology. It's something that got us excited and we made our own Zombie Shooter VR Game, using Unity 5.6 and Google SDK 1.4.0 – the latest.

2 VR Introduction

Unity has introduced built-in support for certain VR devices. In this project documentation we will focus on the Google Cardboard family of VR devices, notably the Google Cardboard VR Kit and the consumer edition of the Gear VR (a mobile headset which requires a LG Device- such as LG G4, LG G3, Nexus 5x). We are not focusing on Samsung devices, which were previously supported by the first Innovator Edition of the Gear VR, though we expect that the VR samples will run on this device (albeit with lower performance) for those of you that have it. Google Cardboard brings immersive experiences to everyone in a simple and affordable way. Whether you fold your own or buy a Works with Google Cardboard certified viewer, you're just one step away from experiencing virtual reality on your smartphone.

Other VR Head Mounted Displays (HMDs) will also work with Unity, such as the HTC Vive.

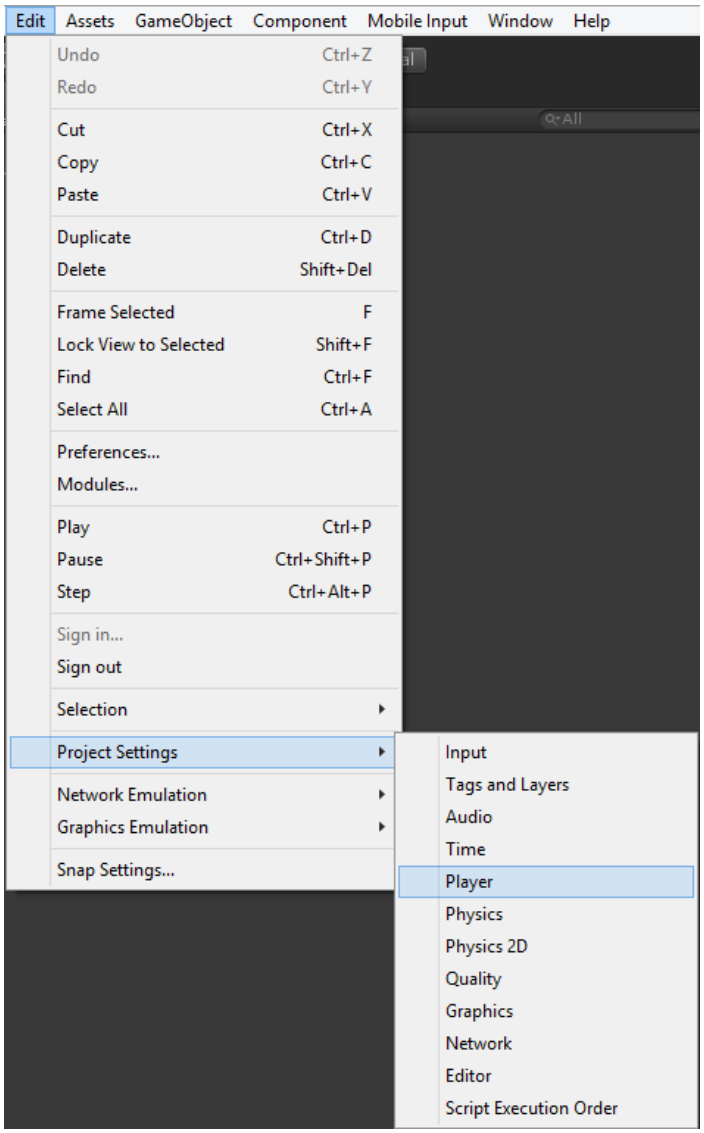
While most of this content will be relevant for all VR HMDs, please see the manufacturer's documentation for more details.

1 Enabling VR in Projects

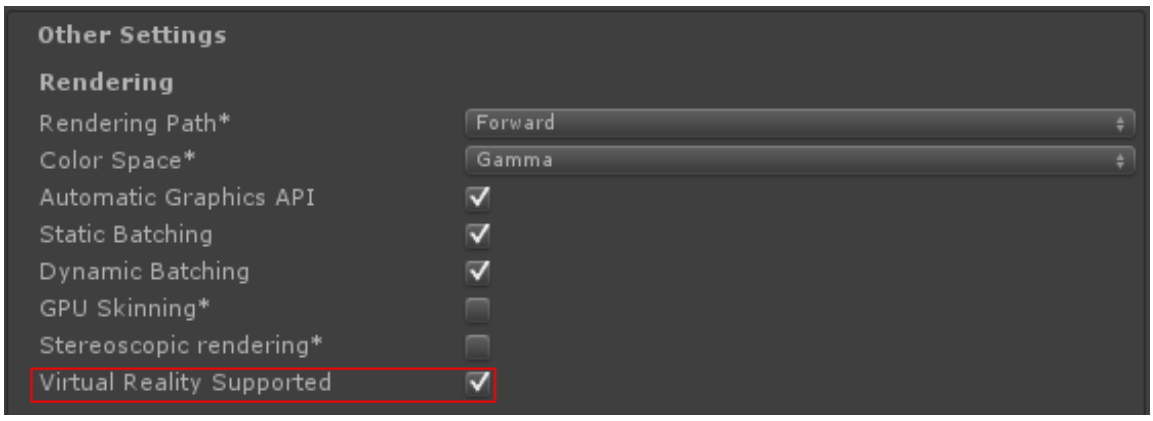
During runtime, this can be toggled using the UnityEngine.VR.VRSetting.enabled property in code, as shown below:

```
using UnityEngine;
using UnityEngine.VR;

public class ToggleVR : MonoBehaviour
{
    //Example of toggling VRSettings
    private void Update ()
    {
        //If V is pressed, toggle VRSettings.enabled
        if (Input.GetKeyDown(KeyCode.V))
        {
            VRSettings.enabled = !VRSettings.enabled;
            Debug.Log("Changed VRSettings.enabled to:"+VRSettings.enabled);
        }
    }
}
```



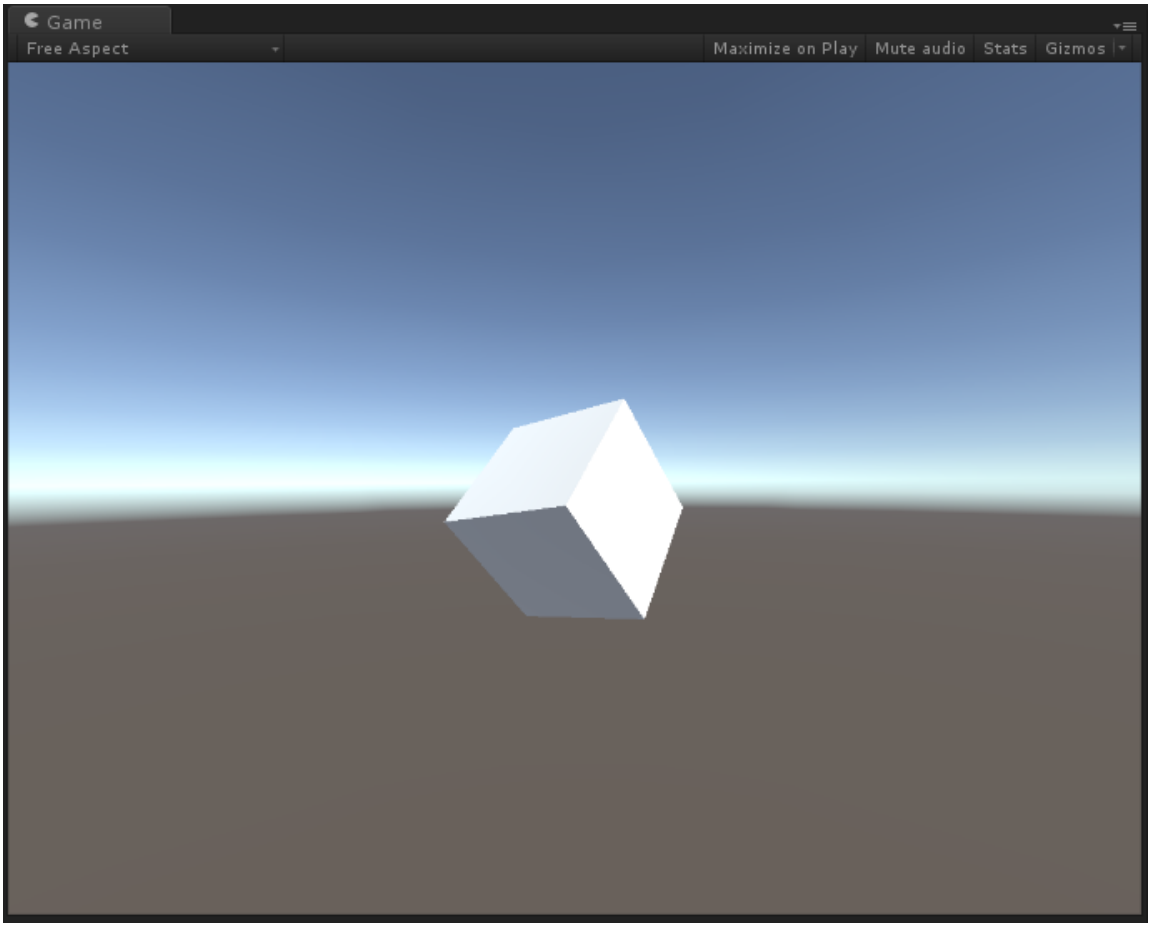
VR support is enabled by visiting *Edit > Project Settings> Player > Other Settings > Rendering*.



Then enabling the “Virtual Reality Supported” checkbox in the Inspector.

2 *Preview your VR Project*

When VR Support is enabled in the Unity Editor entering Play mode will display the Game view, as well as the editor. This allows for much quicker testing and iteration - there is no need to build an executable of the project to see your changes in VR.



There is no need to create one camera per-eye; all cameras will render in VR, except for those with a Render Texture assigned. Optimizations are automatically applied to make rendering both cameras less expensive, such as culling and rendering shadows once for both eyes.

3 *Hardware and Software Recommendations*

Because VR is a very new medium, at the time of writing there are some hardware and software limitations, which are outlined below.

Hardware

Achieving the required frame rate is essential for a good VR experience. On the Google Cardboard VR, it must be 60fps. If the frame rate drops below this, it is particularly noticeable to the user, and will often lead to nausea. The GPU in the attached PC must also be capable of outputting the required resolution at the panel's refresh rate. In the case of Google Cardboard, this is 1920 x 1080 at 75hz.

Software

OS X: At this time, it's possible to develop on OSX 10.9+ with Google Cardboard, but there are optimization and rendering errors, so we recommend Windows for native VR functionality in Unity.
Windows: Windows 7, 8, 8.1, and Windows 10 are all compatible.
Android: We recommend using Android OS Lollipop 5.1 or higher.

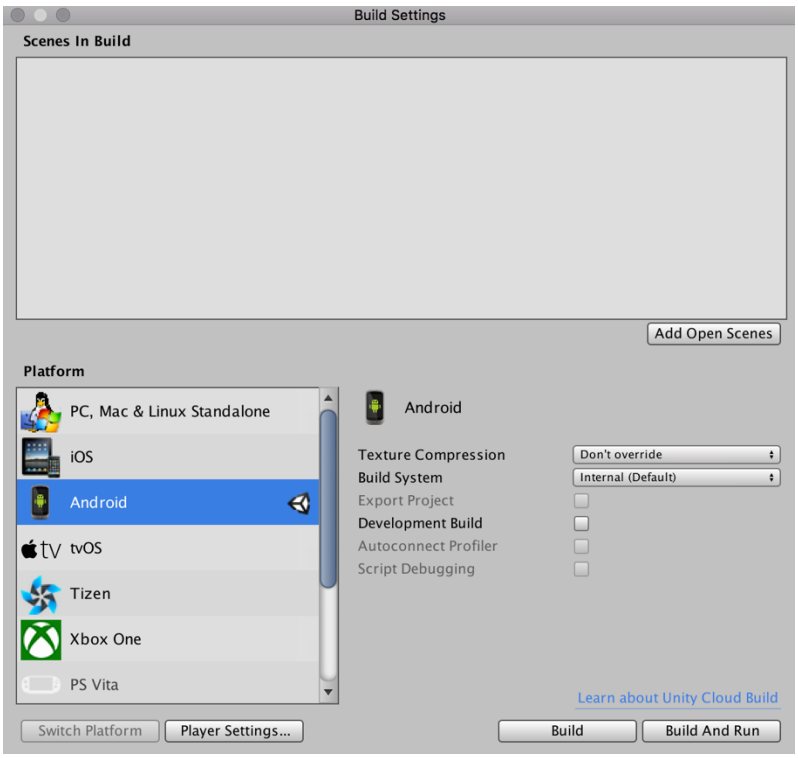
3 Getting started with VR Development

1 The Basics

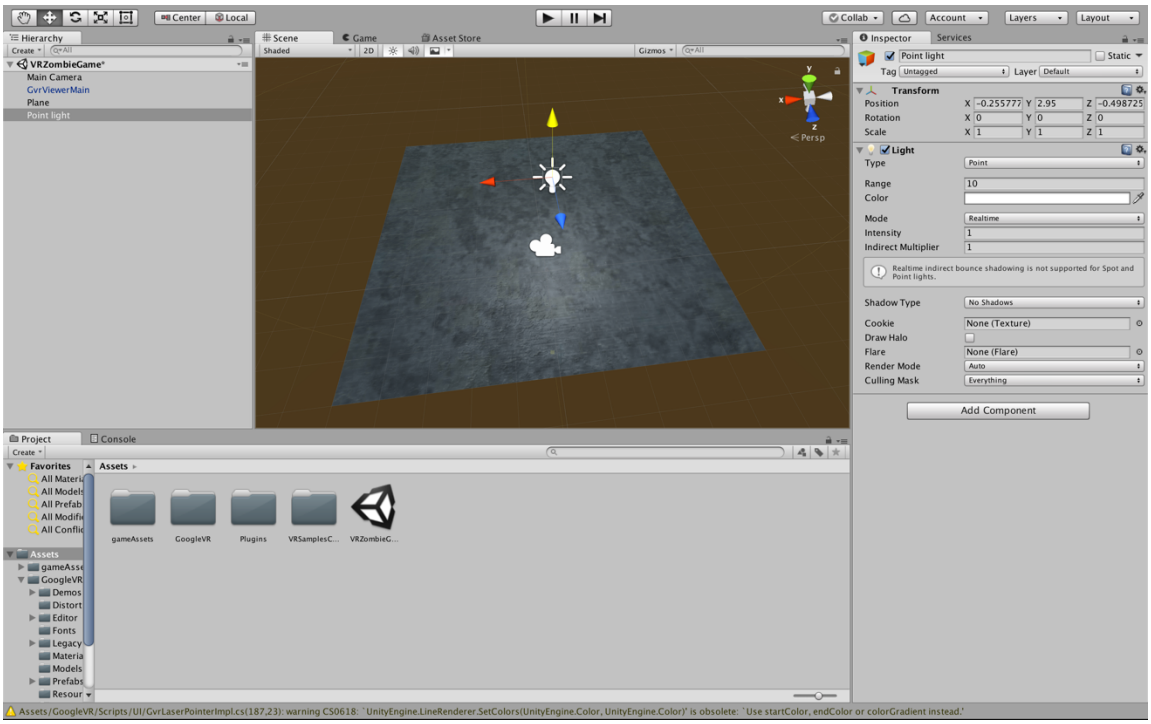
To get started with the basics of VR development in Unity, we had to ensure our hardware and software is set up as described above. Once that's done and we've installed Unity, we check that the Demo Scene in the Oculus Configuration Utility is functioning correctly before continuing.

2 Creating the VR Project

Step 1: Create a new empty project from the Unity Home Screen which loads when you first launch Unity.
Step 2: Make sure that **Android** is selected as the platform to use by visiting File > Build Settings from the top menu.



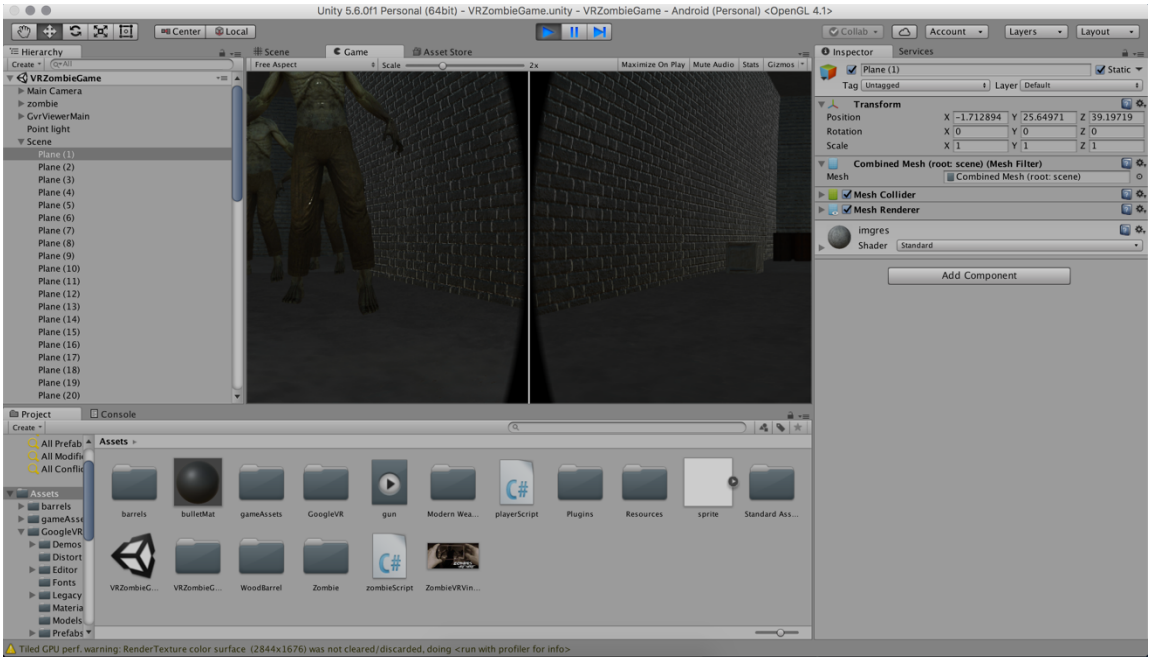
Step 3: Create a new plane (*Game Object* > *3D Object* > *Plane*) and position it in front of the default Main Camera in your new empty scene using the Translate tool.



Step 4: Save your scene (File > Save Scene).

Step 5: Go to *Edit* > *Project Settings* > *Player* and check the box to enable "Virtual Reality Supported".

Step 6: Enter Play mode by pressing Play at the top of the interface.

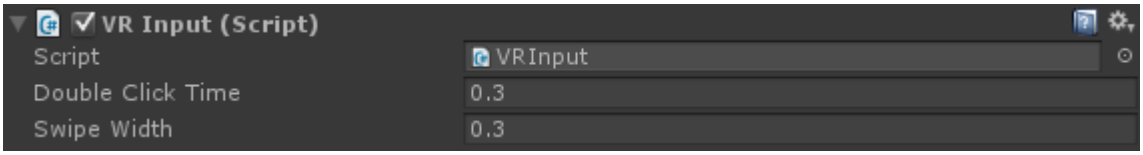


4 Interaction in VR

1 Overview

In VR, we frequently need to activate an object that a user is looking at. For the Zombie Shooter Game we have built a simple, extendable, lightweight system allowing users to interact with objects. This consists of three main scripts: VRInput, GVRReticlePointer and VRInteractiveItem - a short description of these classes is below.

VRInput



VRInput is a simple class that determines whether swipes, taps, or double-taps have occurred on the Gear VR - or the equivalent controls setup for PC input when using a Google Cardboard. You can subscribe to events on *VRInput* directly:

```
public event Action<SwipeDirection> OnSwipe;          // Called every frame passing in the swipe, including if there
public event Action OnClick;                           // Called when Fire1 is released and it's not a double click.
public event Action OnDown;                            // Called when Fire1 is pressed.
public event Action OnUp;                              // Called when Fire1 is released.
public event Action OnDoubleClick;                     // Called when a double click is detected.
public event Action OnCancel;                          // Called when Cancel is pressed.
```

VRInteractiveItem

This is a component you can add to any GameObject that you would like the user to interact with in VR. It requires a collider on the object it is attached to. There are six events you can subscribe to:

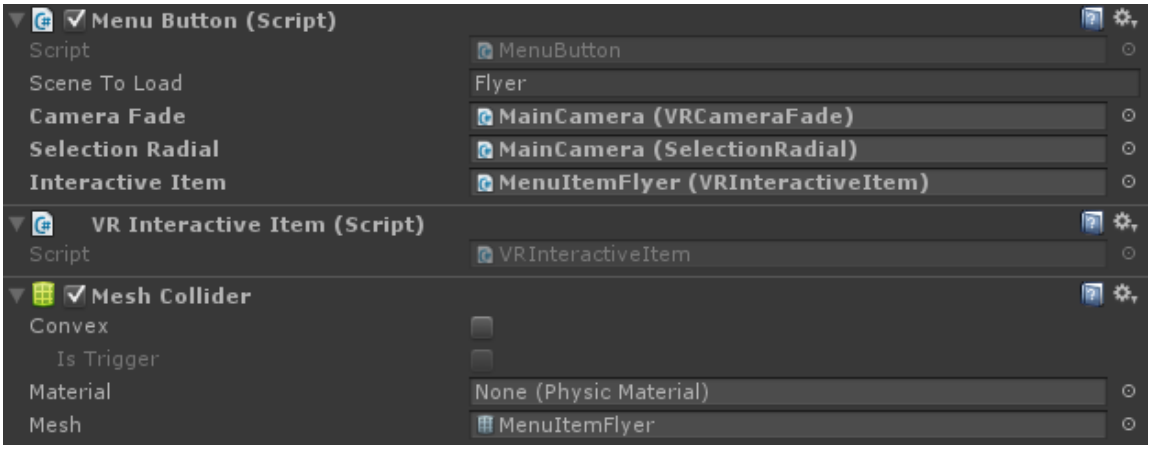
```
public event Action OnOver;             // Called when the gaze moves over this object
public event Action OnOut;              // Called when the gaze leaves this object
public event Action OnClick;            // Called when click input is detected whilst the gaze is over this object.
public event Action OnDoubleClick;      // Called when double click input is detected whilst the gaze is over this object.
public event Action OnUp;               // Called when Fire1 is released whilst the gaze is over this object.
public event Action OnDown;             // Called when Fire1 is pressed whilst the gaze is over this object.
```

And one boolean that can be used to see if it's currently Over:

```
public bool IsOver
{
    get { return m_IsOver; }              // Is the gaze currently over this object?
}
```

Interactions in the Menu Scene

Each of the menu screens have several components. Of particular interest here are the *MenuButton*, *VRInteractiveItem*, and *Mesh Collider*.



2 Make a splash screen and icon

The Unity Editor allows you to configure a Splash Screen for your project. The level to which you can customize the Unity Splash Screen depends on your Unity license; depending on which license you have, you can disable the Unity Splash Screen entirely, disable the Unity logo and add your own logos, among other options. You can also make your own introductory screens or animations to introduce your project in your first scenes – the introductory cut scene. The Unity Splash Screen is uniform across all platforms. It displays promptly, displaying while the first Scene loads asynchronously in the background. This is different to your own introductory screens or animations which can

take time to appear; this is due to Unity having to load the entire engine and first Scene before displaying them.

This is how the icon looks like – the first image you see while downloading and installing the game.

The first look and feel of the game is seen through the splash screen which we created in Photoshop. We made a static photo holding the telephone in the capture picture motion, and we later edited the photo to make it seen realistic with the person taking a photo of an actual zombie, reflecting how also our world is evolving in an age where photos are more important than our life.

3 Code examples

After click on button sound

```
    void Start () {
        gameObject.AddComponent<AudioSource>();
        source.clip = sound;
        source.playOnAwake = false;
        button.onClick.AddListener(() => PlaySound());
    }

    void PlaySound()
    {
        source.PlayOneShot(sound);
    }
```

We use the ClickSound.cs script in order to add sound after every button click – such as the opening of a doc after the play game button has been activated.

Walking effect

```
    void Start () {
        cc = GetComponent<CharacterController>();
    }

    void Update () {
        if(vrCamera.eulerAngles.x >= toggleAngle && vrCamera.eulerAngles.x < 90.0f)
        {
            moveForward = true;
        }
        else
        {
            moveForward = false;
        }
        if (moveForward)
        {
            Vector3 forward = vrCamera.TransformDirection(Vector3.forward);
            cc.SimpleMove(forward * speed);
        }
    }
}
```

When you are in the forest, if you point your view down – you walk. We simulate walking without creating game nausea because we use degrees and angles to adjust the way the movement works.

So, first of all we use **Start()** for initialization, where we use a variable to get the CharacterController Component. Then we create our function, we called it **Update()** because it moves in real time with the way you move, updating the camera view and position.

First we check if the cameras VR angle on the x line coordinate is bigger than the toggle variable we use to change the position between walking and stopping and we also check if the VR angle on the x coordinate is smaller than 90.0f – that represents the 90 deg. of our visual spectrum, which is a floated value, so that if the conditions are met the characters walk, or they stop if the statement is false. If the character moves forward then we create a Vector3 variable that allows the camera to move in the direction that we are looking at because Vector3 takes information from all the axes, and the SimpleMove() function just puts every item and variable in place and allows the camera to move with a certain speed value we gave.

Runtime Cardboard Loader

```
void Start()
{
    StartCoroutine(LoadDevice("cardboard"));
}

IEnumerator LoadDevice(string newDevice)
{
    VRSettings.LoadDeviceByName(newDevice);
    yield return null;
    VRSettings.enabled = true;
}
```

The runtime cardboard loader first of all initializes the device for the cardboard, so that the app knows how to run based on the cardboard QR code the user gave when he first used it. Next we use the **LoadDevice()** function we just made and initialized in order to to establish the VRSettings to true, and allow the app to run in VR mode for our mobile phones.

Player Script

```
IEnumerator Shoot()
{
    isShooting = true;
    GameObject bullet = Instantiate(Resources.Load("bullet", typeof(GameObject))) as GameObject;
    Rigidbody rb = bullet.GetComponent<Rigidbody>();
    bullet.transform.rotation = spawnPoint.transform.rotation;
    bullet.transform.position = spawnPoint.transform.position;
    rb.AddForce(spawnPoint.transform.forward * 500f);
    GetComponent<AudioSource>().Play();
    gun.GetComponent<Animation>().Play();
    Destroy(bullet, 1);
    yield return new WaitForSeconds(1f);
    isShooting = false;
}
```

First of all in the player script, we make the **Shoot()** function an **IEnumerator** because we want to be able to delay the shooting. We declare our isShooting boolean to true in the function, so we can finally add animation and moves to our gun, spawn point and bullet. We create a bullet game object, and a rigid body in order to make the bullet take the same position and movements as the spawn point. The AddForce() function creates the speed in which the bullet it going towards the enemy. We then add the animation to the gun and the shooting sound in the moment we shoot the gun. Also we destroy the bullet so that there will be no elements left in the scene. We use a second delay in order to make the recharging as closer to reality as possible and not allow the user to shoot

constantly. Also we then instantiate the isShooting variable to false, in order to take the process from start each time an enemy has been shot.

The zombie script

```
    void OnTriggerEnter(Collider col)
    {
        GetComponent<CapsuleCollider>().enabled = false;
        Destroy(col.gameObject);
        agent.destination = gameObject.transform.position;
        GetComponent<Animation>().Stop();
        GetComponent<Animation>().Play("back_fall");
        Destroy(gameObject, 6);
        GameObject zombie = Instantiate(Resources.Load("zombie", typeof(GameObject))) as GameObject;
        float randomX = UnityEngine.Random.Range(-12f, 12f);
        float constantY = .01f;
        float randomZ = UnityEngine.Random.Range(-13f, 13f);
        zombie.transform.position = new Vector3(randomX, constantY, randomZ);

        while (Vector3.Distance(zombie.transform.position, Camera.main.transform.position) <= 3)
        {
            randomX = UnityEngine.Random.Range(-12f, 12f);
            randomZ = UnityEngine.Random.Range(-13f, 13f);
            zombie.transform.position = new Vector3(randomX, constantY, randomZ);
        }
    }
}
```

This script allows the zombie to interact with the gun and bullet and the user. First of all we have to make sure the Zombie element has the trigger component checked, then we must set the collider to false – because we don't want to hit the zombie in the same spot without him dying. After the zombie is dead, we destroy the bullet, the collider animation. We then make the moving animation stop and we play the falling animation. We destroy the zombie and then we create a new one, so each time one zombie dies another one spawns, and it will spawn in between the given intervals. If the zombie is less than 3 units close to us, we won't be able to shoot it because it will interfere with the gun, so we made the range of the zombie walking animation to stop before that happens.

4 User Interfaces for VR

When designing user interfaces for VR there are a number of things to consider that may not have come to light in traditional screen design scenarios. Here we will look at some of the challenges and opportunities afforded as a VR developer that we faced, and discuss some of the hardware practicalities of designing usable interfaces for this new medium.

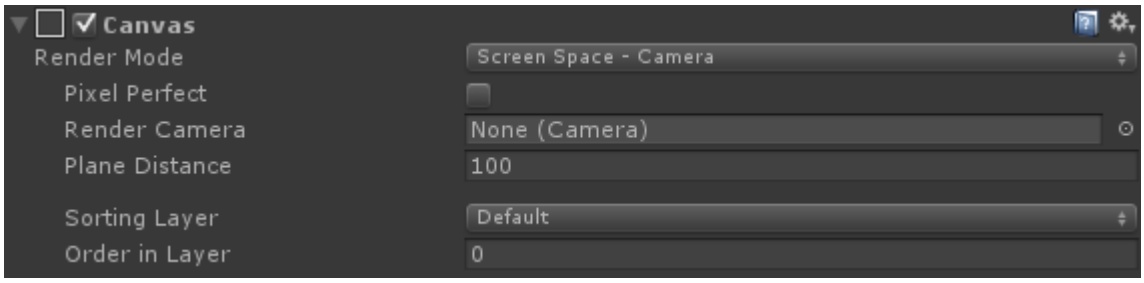
1 UI Resolutions and Appearance

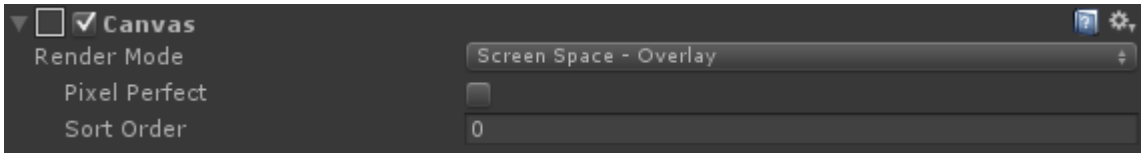
As the resolution on Google Cardboard is 1920 x 1080 (960 x 1080 per eye), and the Gear VR is 2560 x 1440 (1280 x 1440 per eye), this can lead to some noticeable pixelation on anything that occupies a few pixels in width or height. Of particular note are UI elements; bear in mind how large these will appear on-screen. One approach is to use larger or bold fonts, and designing UI without thin lines that can become pixelated when viewed in VR.

2 Types of UI

Non-diegetic

In non-VR projects, UI is frequently overlaid on top of the screen to show things like health, score, and so on as what we often refer to as a HUD (Heads Up Display). This is known as non-diegetic UI - it doesn't exist within the world, but makes sense for the player in the context of viewing the game. In Unity, adding HUD style non-diegetic UI is usually accomplished by using *SCREEN SPACE - OVERLAY* or *SCREEN SPACE - CAMERA* render modes on a UI Canvas.

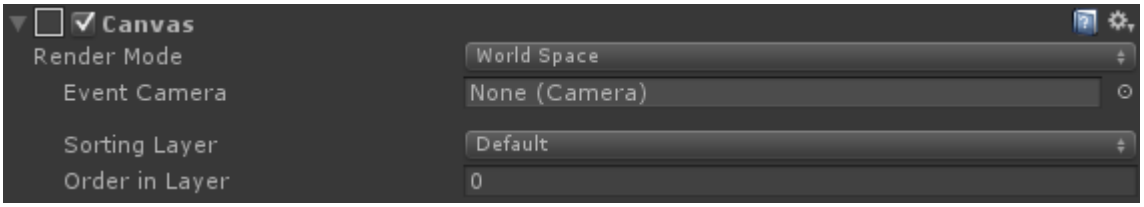




This approach usually doesn't work in VR - our eyes are unable to focus on something so close, and Screen Space-Overlay is not supported in Unity VR.

Spatial UI

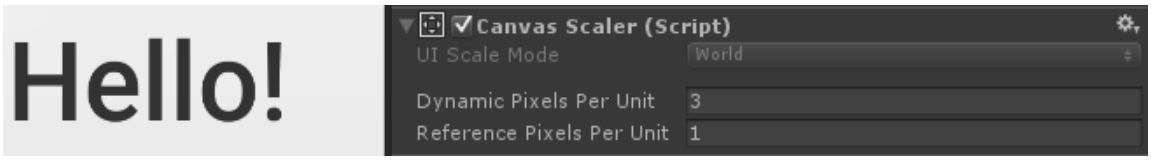
Instead, we generally need to position our UI within the environment itself using *WORLD SPACE* Canvas render mode. This will allow our eyes to focus on the UI. This is known as Spatial UI.

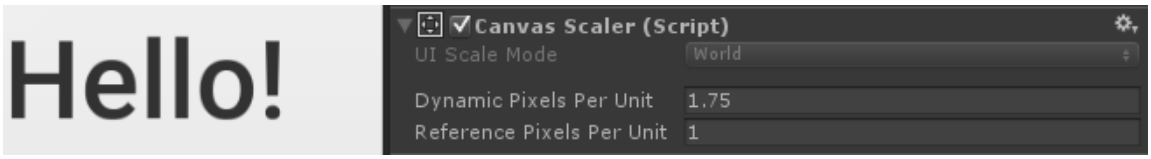


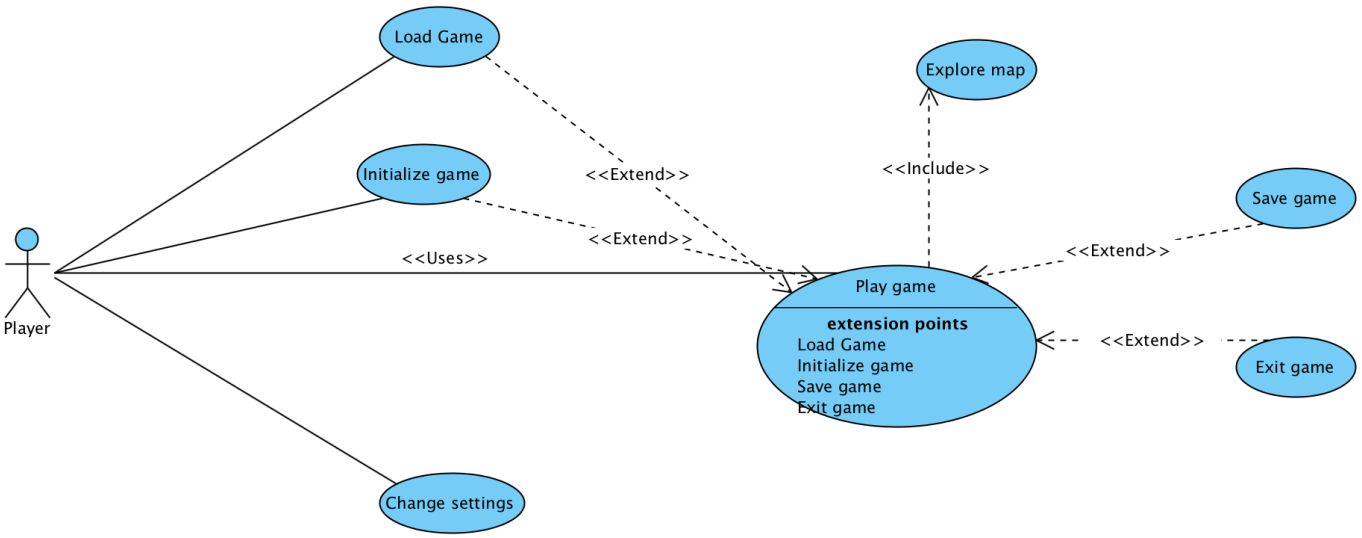
Placement of the UI within the world also needs some consideration. Too close to the user can cause eye strain, and too far away can feel like focusing on the horizon - this might work in an outdoor environment, but not in a small room. You'll also need to scale the size of the UI accordingly, and perhaps dynamically depending on your needs. If possible, it's best to position your UI at a comfortable reading distance, and scale it accordingly. See the UI in Main Menu for an example of this: It's positioned a few meters away, and the text and images are large and easy to view.

Free antialiasing on text for VR

A quick way to achieve free (from a rendering cost standpoint) anti-aliasing on text in Unity is to use a Canvas UI on a Worldspace Canvas. The UI should have a “Reference Pixels Per Unit” setting of 1, then alter “Dynamic Pixels Per Unit” until you slightly soften the edges of the text. Here you can see the difference between a setting of 3 Dynamic Pixels Per Unit - where the edges look sharp - and an example of the Dynamic Pixels Per Unit being set to 1.75, which gives a much softer edge.







UML Diagram of a user case scenario where we can see the players attributes.

5 Movement in VR

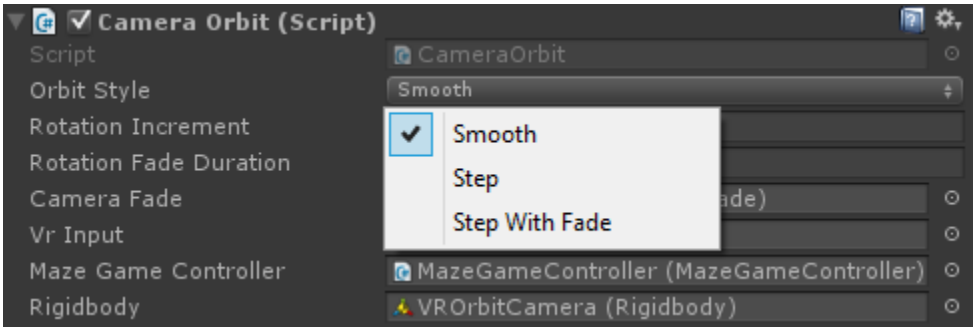
Along with not achieving the target frame rate, movement in VR is one of the primary causes of VR sickness in users, and as such it requires careful consideration before implementing a solution in your project. This is best considered very early on in development - ideally during the concept phase - as it can heavily impact your project if your chosen movement solution is discovered to induce nausea.

Nausea and comfort in VR

The feeling of nausea is partly due to the real-world user's body being stationary while their virtual point-of-view moves around a virtual environment. In general, a good rule is to avoid moving the camera unless it's copying user movement. Put simply, Vection is the effect of confusing the user's brain by having conflicting signals sent from the eyes and ears, it can cause the body to assume it has been poisoned, and thus a similar response is evoked - a rejection in the form of sickness. As with everything, there are exceptions, so do experiment and test with as wide an audience as possible to see what works with your game.

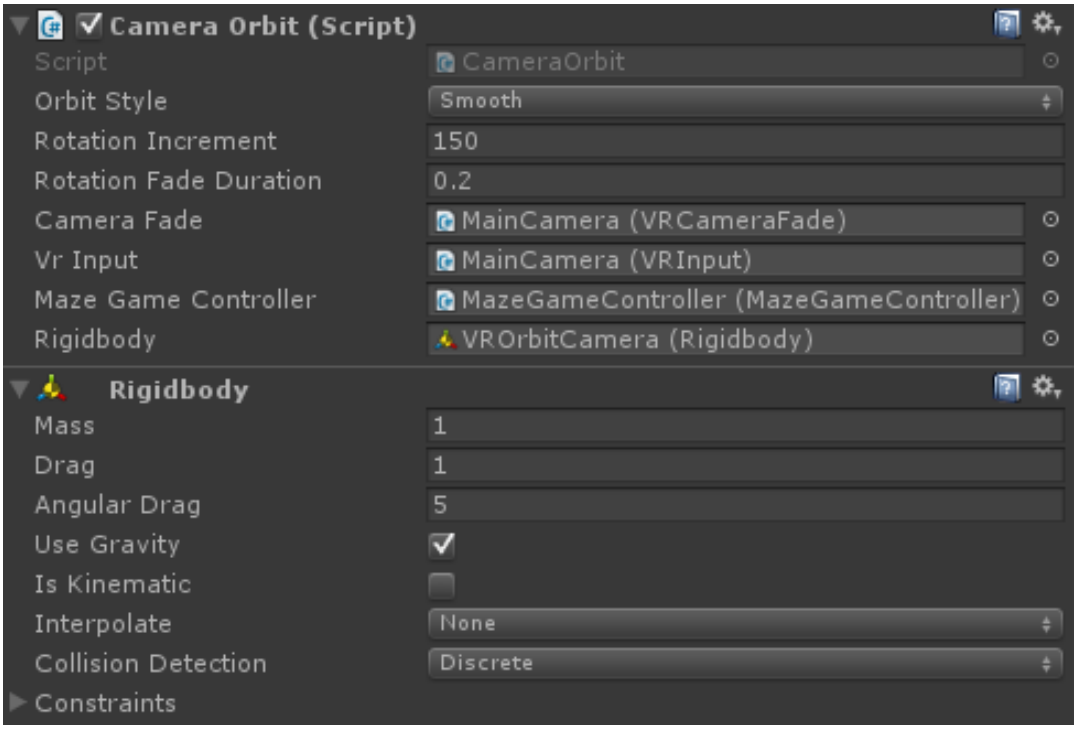
First Person Games in VR

Traditional first-person character control with mouse and WASD or gamepad frequently induces nausea, so are best avoided. If you do decide to use first person control, test your movement with as many users as possible, and definitely disable head-bobbing.



Fade / Blink Transitions in VR

A popular method for moving between positions in a virtual environment is to implement fade to black transitions - a very quick fade to black, move the camera to the desired position, and then fade back up.

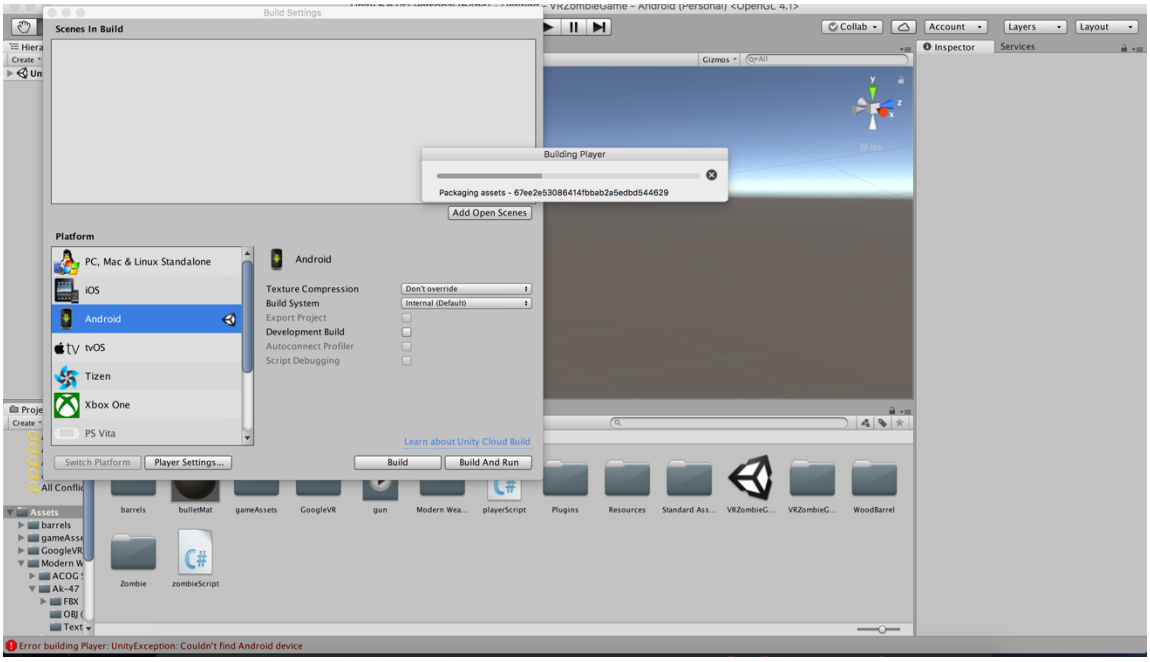


6 Deploying the VR Project

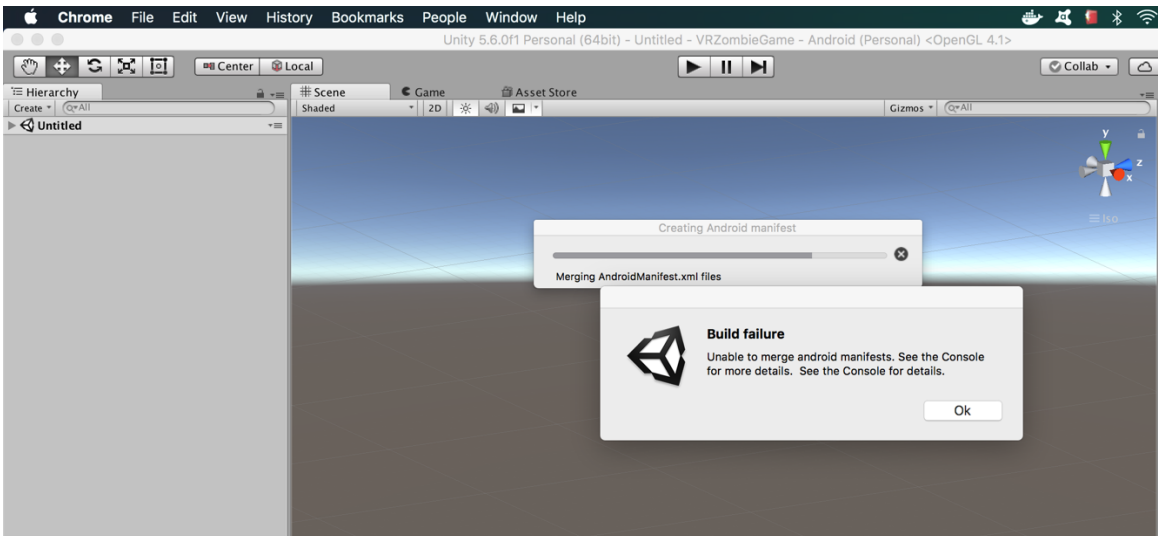
Step 1: Go to *File* > *Build* > *Build Settings* and make sure that Android is selected.

Step 2: You will need to provide the Android SDK Destination Folder and Key.

Step 3: After you added the SDK for Android, click the Build button.

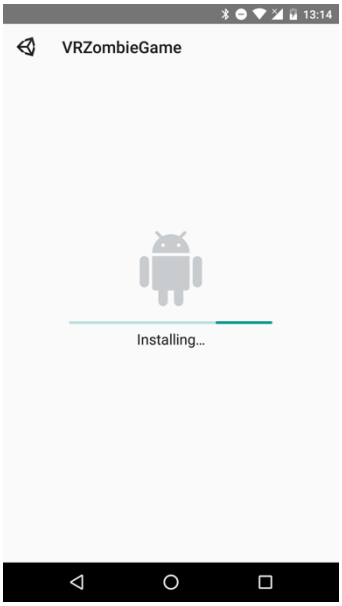


Special Step: If we encounter an error.



If the eror we receive is a **Build Failure**: Unable to merge android manifest. See the Console for more details > This means that the SDK Path we gave for Android is not correct and we need to either change it or reinstall it. In our case, the SDK was not compatible with the version of Unity and we had to both upgrade Unity to 5.6 and add a new Android SDK to the system, through Android Studio – and that is the Android 5.1.

Step 4: Build the project again and then run in on the mobile device by installing the apk.



Step 5: After you have installed the game, run it.

Bibliography

1 https://unity3d.com/learn/tutorials/topics/virtual-reality/vr-overview?playlist=22946 --accessed on 1st of May 2017, 17:00PM

2 https://unity3d.com/learn/tutorials/topics/virtual-reality/getting-started-vr-development?playlist=22946 --accessed on 1st of May 2017, 17:30PM

3 https://forum.unity3d.com/threads/how-to-use-cardboard-reticle-vr-gaze-pointer-cursor-cardboard-button-gaze-input.388492/ --accessed on 3rd of May 2017, 12:00PM

4 https://forums.oculus.com/developer/discussion/45098/unity-gear-vr-ui-selection-gaze-input --accessed on 12th of May 2017, 18:00PM

5 https://vr.google.com/cardboard/get-cardboard/ --accessed on 21st of May 2017, 18:30PM

6 http://www.pocket-lint.com/news/136540-what-is-vr-virtual-reality-explained --accessed on 1st of May 2017, 16:00PM

7 *Unity Virtual Reality Projects* published by *Jonathan Linowes* on 31st of August 2015 --accessed on 1st of June 2017, 11:00AM