Assignment 1: Building and Deploying Your First VR App

DESCRIPTION

In this assignment, you will build and deploy your first VR application, which will take the form of a simple app for Google Cardboard. As part of this assignment, you will learn about the basic components of a VR HMD and complete a basic Unity tutorial.

EXPECTED OUTCOMES

Expected outcomes include:

- Understanding of the major components of VR HMDs
- Understanding proper HMD care, maintenance, and safe use
- Understanding of the Unity interface
- Understanding how to get a basic Unity scene to load in Google Cardboard
- Gain practice with Unity primitives, materials, and interactive components

RESOURCES

If you don't have access to an Android phone, borrow one from a friend or someone else in the class for your deployment (if you still can't find a phone, come talk to me and/or see the guideline "Configuring iOS Project Settings here: https://developers.google.com/cardboard/develop/unity/quickstart). You will all get a Google Cardboard from me on the second day of class when we start this assignment.

STEP 1: INTRODUCTION TO HMDS

Your first task is to familiarize yourself with the basic components of a head-mounted display (HMD). Modern HMDs typically contain:

- One or more **displays** that show the virtual content
- **Lenses**, used to *focus* and *magnify* the display(s)
- **Tracking hardware** used to track head (and possibly other body) movements
- **Casing** that puts all of these components together

HMDs may also contain other elements, such as input buttons that enable user interaction with virtual content or headphones/speakers that enable 3D audio.

In this assignment, you will be building and deploying an application for Google Cardboard or a Cardboard-like HMD. In such HMDs, a modern smartphone provides the display and the tracking hardware, which utilizes the phone's internal measurement unit (IMU). The casing enables correct positioning of the smartphone between the lenses. Cardboard-like HMDs enable VR content wirelessly and at low cost. However, they usually only support 3 degree-of-freedom (DOF) tracking, the casing build quality is cheap and may degrade quickly, and the display quality has less resolution than single-purpose HMDs, such as those from Oculus, HTC, Steam, etc. Take a minute to familiarize yourself with the components of your HMD, which should to some degree match those in Figure 1. Can you find the display, lenses, casing, and tracking hardware? Are there any input buttons? Any other components, either for sensing, input, or ergonomics?

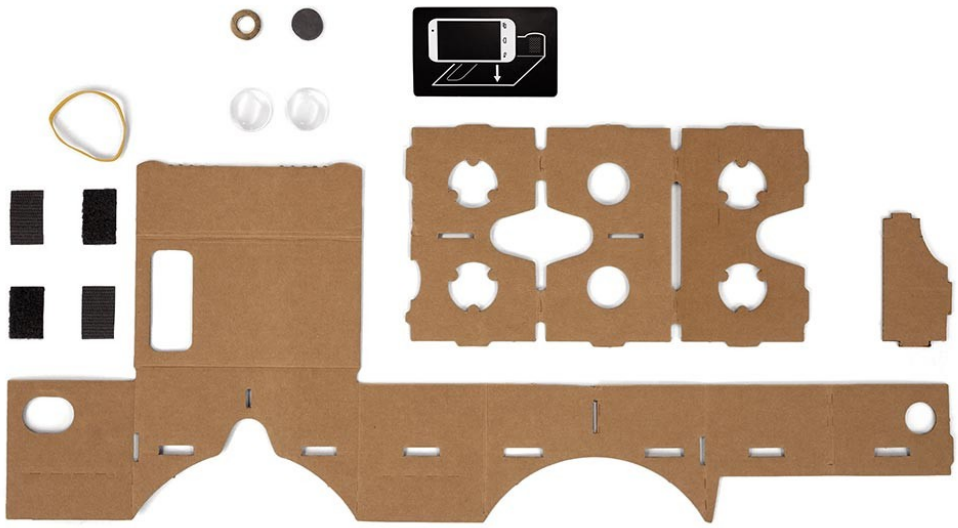


Figure 1: Components of Google Cardboard

NOTES

Extra room to take notes on HMD components is provided below:

Step 2. Setup

Download, install, and setup the software required for building and deploying an application to Google Cardboard. You must install:

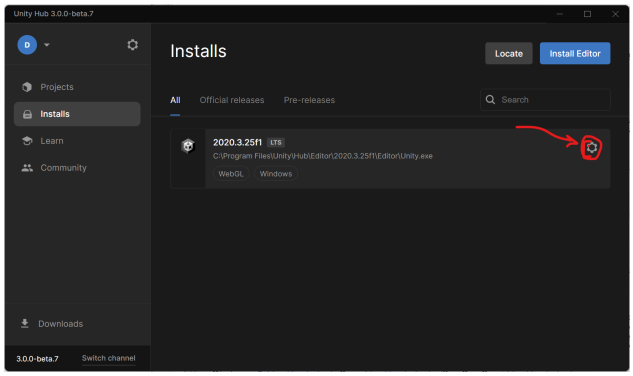
- Unity 2020.3.25f1

Installing Unity

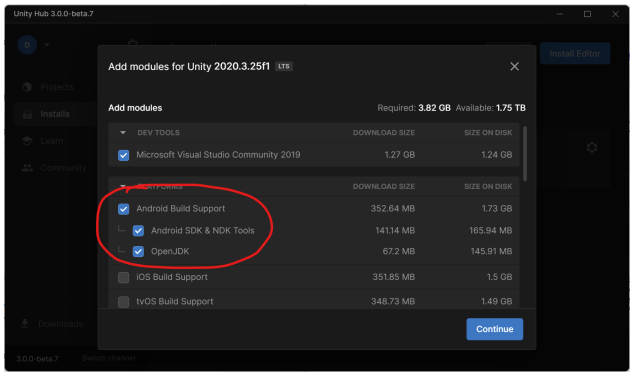
Unity now uses "Unity Hub" to manage Unity installation and keep track of various Unity versions. As a result, your first step is to install Unity Hub. You will also have to create a Unity account as part of this process.

Once you have Unity Hub, use it to install Unity. We will be using version 2020.3.25f1 (Unity's most recent LTS version; if you have a different version of Unity already installed, please install this one too for the purposes of this course).

Once Unity is installed, you need to add the following modules as shown below:



(a) Select this icon to add modules.



(b) Add Android Build Support (assuming you are using Android; if you want to use an iPhone select iOS Build Support).

ENABLE USB DEBUGGING

To deploy a VR app to your Android phone, you must enable USB Debugging, which enables you to install custom applications via USB. To enable app deployment, in your Android phone, navigate to **Settings -> About Phone -> Build Number**. Depending on your phone, the location where the Build Number is displayed may vary slightly, but it should always be in the About Phone section of your phone's Settings. If you can't find it, try googling "unlock developer mode <your phone model>." Once you find the Build number, tap on it 7 or more times. This will unlock Developer Mode. With Developer Mode enabled, you can now navigate to **Settings -> Developer Options** and **Enable USB Debugging**. This will allow you to deploy apps to your phone via USB.

NOTES

Extra room to take notes on the setup procedures is provided below:

Step 3. Create a Simple Scene

Now you will use Unity to create your very first virtual reality "application" (not a game). Unity is a game engine that is becoming a common way of developing VR and AR apps due to its ease of use and ability to easily deploy to a variety of popular VR and AR devices. However, it is not the only way to develop VR applications. Other approaches include alternate engines, such as Unreal Engine, directly building an application using OpenGL and/or DirectX, or using a web framework such as WebGL or WebVR. We will talk about these tools later in the course.

Step 3 Instructions

Your task is to build a simple Unity scene and deploy it to an Android phone for viewing in Google Cardboard or a Cardboard-like HMD. First, open Unity and create a new project by clicking the +New button in the upper right. Give your project a descriptive name (I called mine "HelloCardboard"). Make sure that the template is set to "3D:"

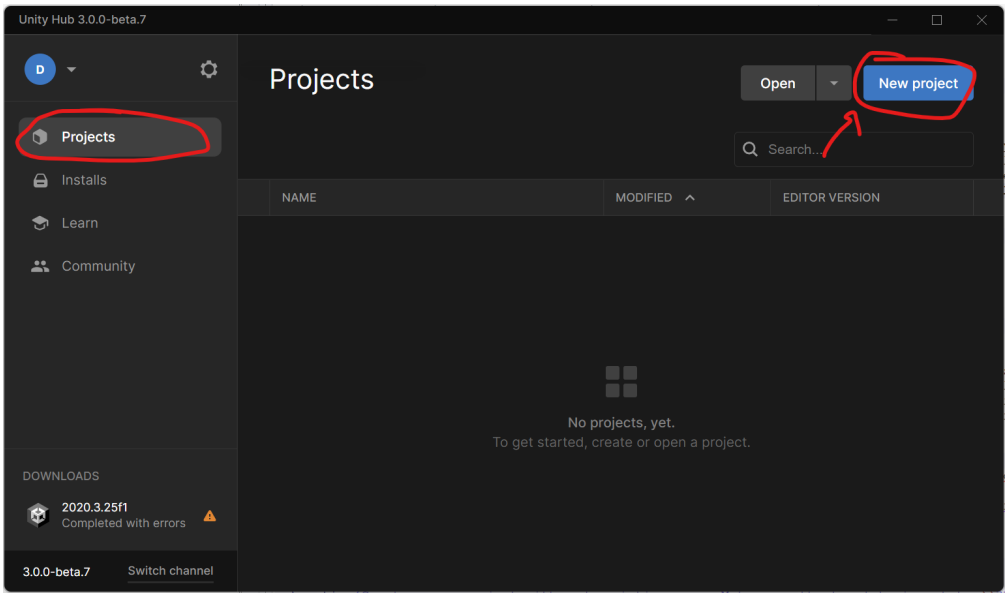


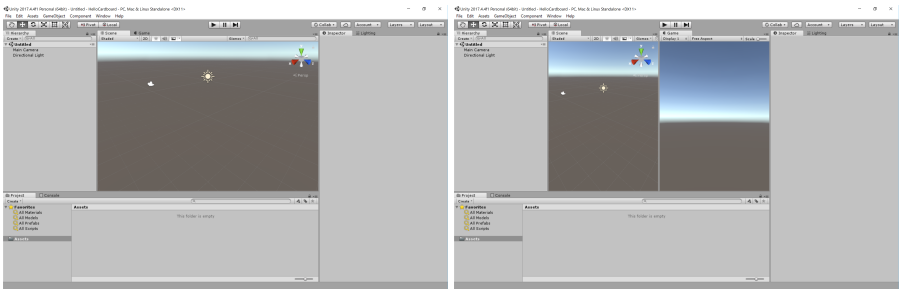
Figure 3: Create a new project with a descriptive name. Make sure the template is set to "3D."

Once you have created your project, take a minute to familiarize yourself with the Unity interface, which should look similar to Figure 4a. If you have never used Unity before, take a look at the following page to learn the basics:

- https://learn.unity.com/tutorial/explore-the-unity-editor-1

Note that many developers like to customize the interface, for instance showing the Scene and Game views simultaneously. You can use the "Window" menu in the top Toolbar to cus-

tomize your layout. To show the Scene and Game views at the same time, click and drag the Game tab such that it appears next to or below the scene view, looking like Figure 4b.



(a) The default Unity interface.

(b) You may wish to customize your layout, such as showing the Scene and Game view side-by-side.

Figure 4: Familiarize yourself with the main components of the Unity interface

Make sure you can identify the *Toolbar*, the *Hierarchy*, the *Scene View*, the *Game View*, the *Inspector*, and the *Project Window*.

Next, we will make the simplest possible scene: a single cube. Add a cube to your scene by selecting GameObject->3D Object->Cube from the Toolbar:

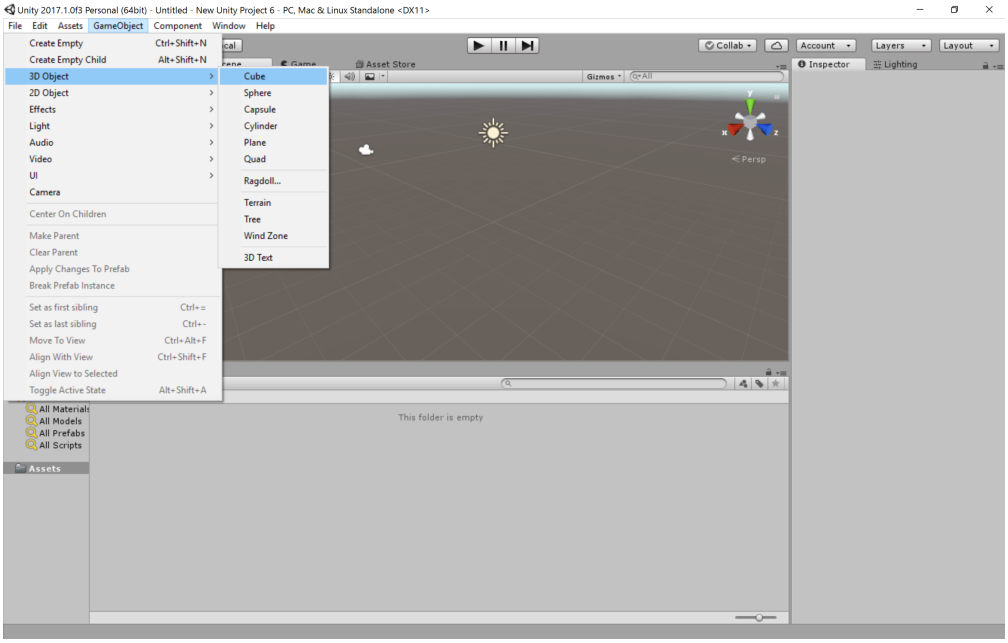


Figure 5: Add a cube to your scene by selecting GameObject->3D Object.

Click on the cube in the Scene view to view its components in the Inspector. Make sure it's position is 0,0,2:

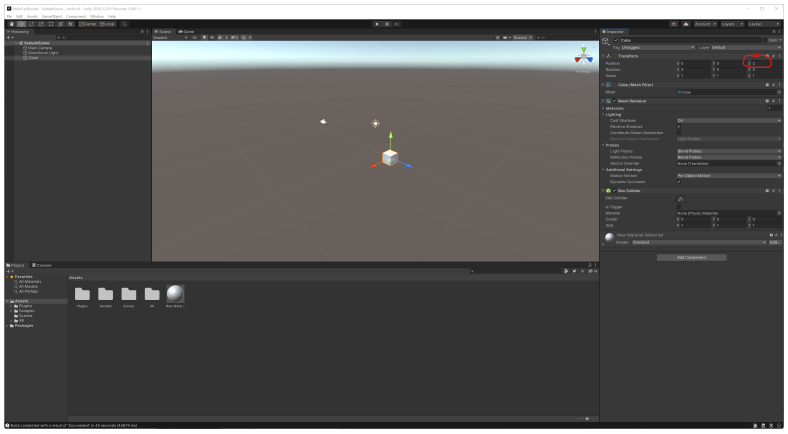


Figure 6: Examine the components of the cube in the Inspector window.

Let's make the cube look slightly more interesting by giving it a new "material." Materials encode visual properties of virtual objects, such as color or reflectance. Create a new material by selecting Assets->Create->Material. Change the material color by selecting the material and clicking on the color-picker in the Inspector window. Apply this material to the cube by clicking the material and dragging it onto the cube (alternatively, select the cube and find Element 0 in the "Materials" drop-down within the Inspector window; double-click "Default-Material" and replace it with your new material):

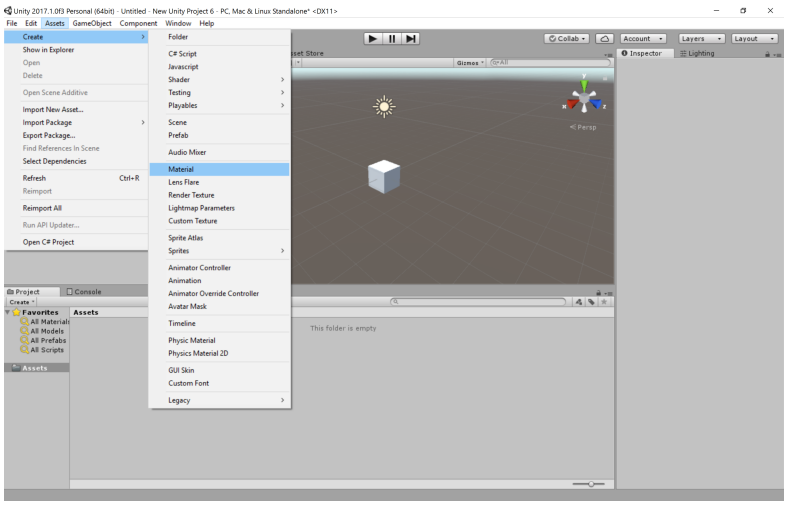


Figure 7: Create a material by selecting Assets->Create->Material

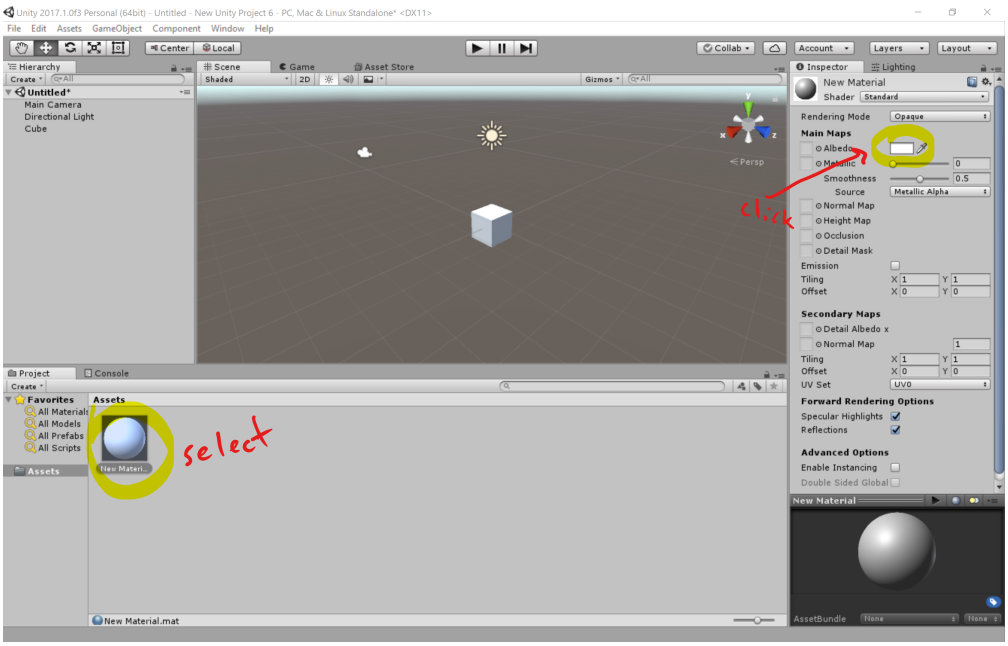


Figure 8: Select the color-picker to change the material color.

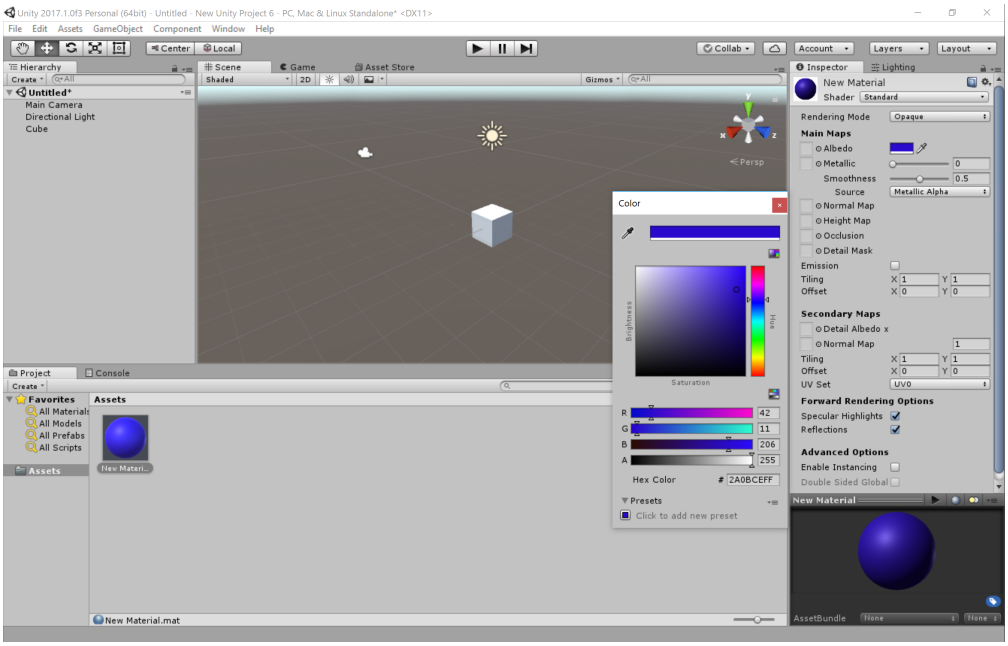


Figure 9: Select a new color from the color-picker dialog.

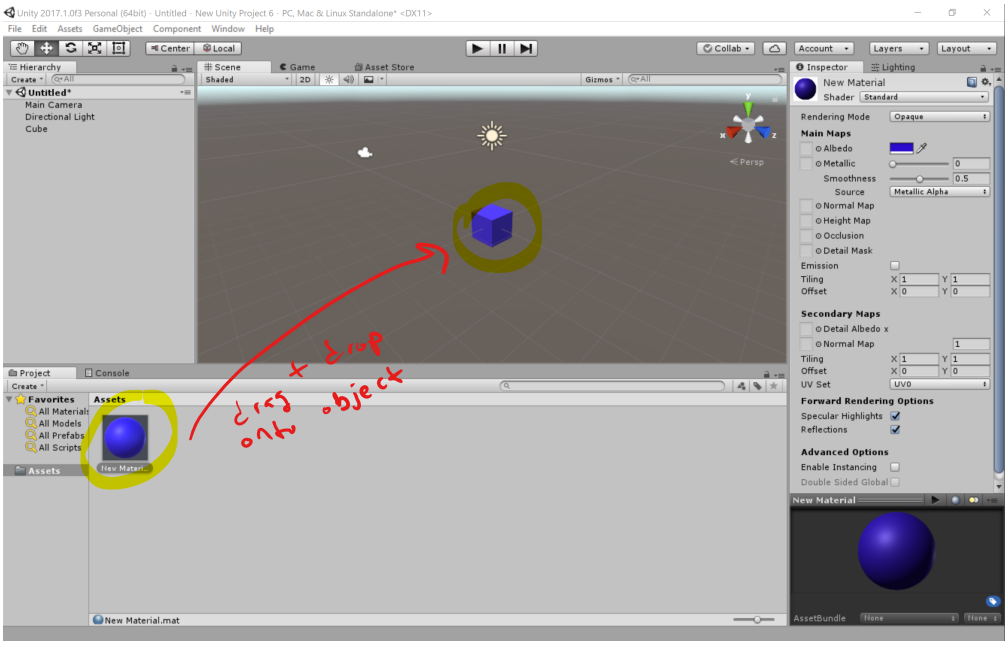


Figure 10: Assign materials to objects by clicking the material and dragging it onto the desired object.

Once you have applied the new material, notice how the cube changes color and note how Element 0 in the “Materials” dropdown reflects the name of your new material in the cube’s Inspector window:

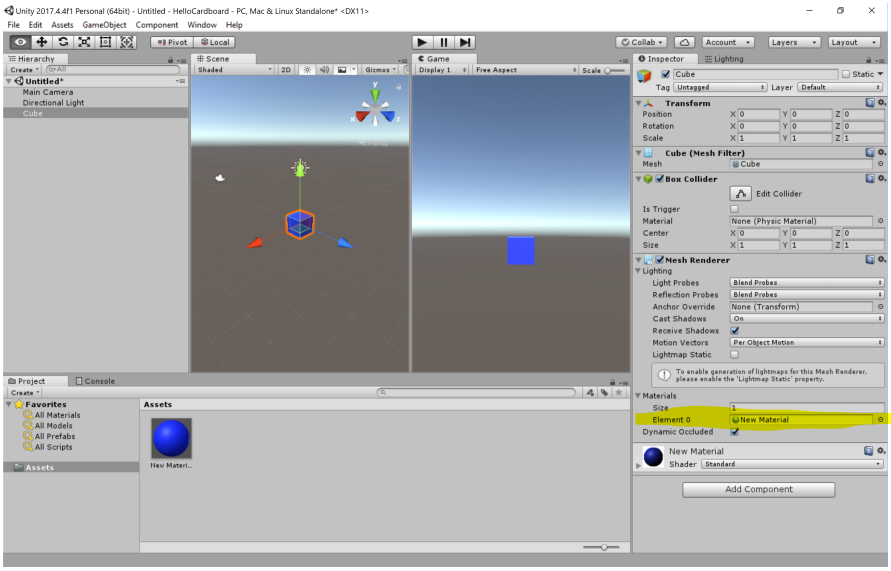


Figure 11: Note how the Materials array has changed in the cube’s Inspector window.

STEP 4. BUILD AND DEPLOY TO ANDROID

Note: This section assumes you are deploying to an Android phone. If you must instead deploy to an iPhone, see this link and scroll down to "Configuring iOS project settings" and replace the relevant instructions below with the iOS instructions.

Now that you have created your scene, we will build it and deploy it to Android for viewing in Google Cardboard. First, we need to configure the various build options to compile our application. In Unity, select **File -> Build Settings...** from the Toolbar (**Unity -> Build Settings** if you are using MacOS):

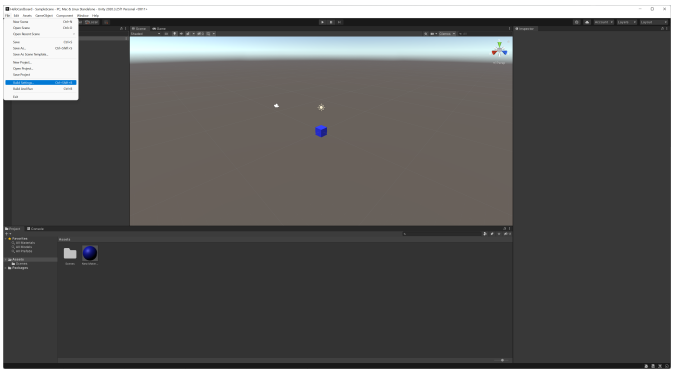


Figure 12: Select **File -> Build Settings** to configure our build.

This will bring up the Unity Build Settings window. In this Window, select "Android" and click "Switch Platform."

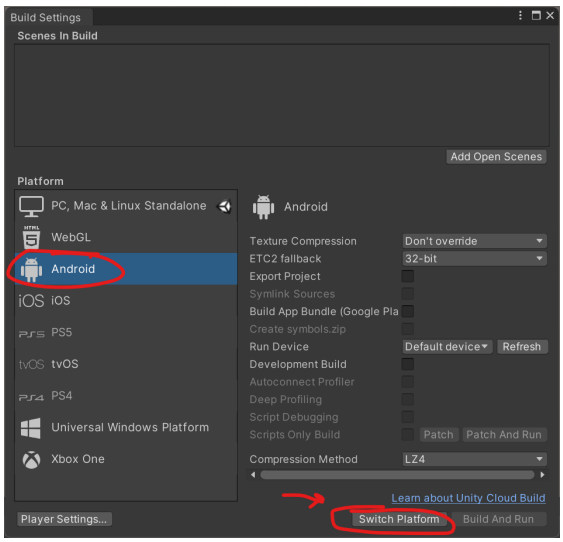


Figure 13: Select Android and hit "Switch Platform" to tell Unity we are developing for Android (which is what Google Cardboard uses).

It will take a minute for Unity to switch over after you click “Switch Platform.” Once it is done, select “Player Settings...” from the Build Settings window.

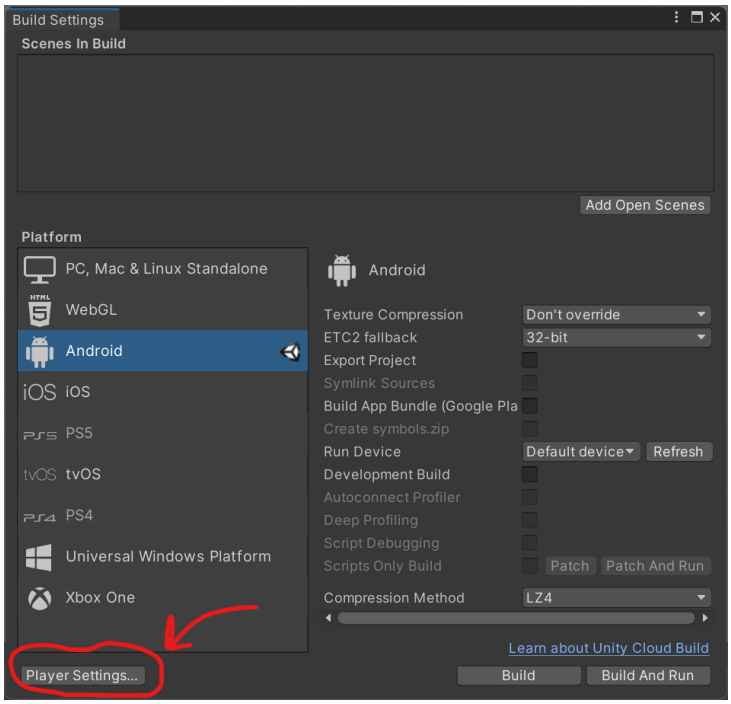


Figure 14: Select “Player Settings” from the Build Settings window.

In Player Settings, click the XR Plug-in Management tab and click the “Install XR Plugin Management” button. Once you do, notice that there is still no Google Cardboard option! This is because it is not included by default as part of Unity’s XR management. We will have to add an additional package from Google.

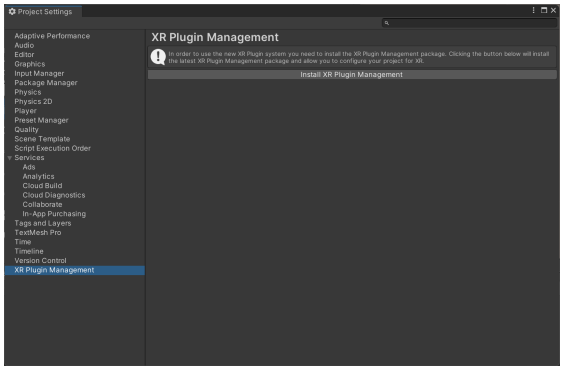


Figure 15: Enable XR Plugin Management from within Player Settings.

ADD THE GOOGLE CARDBOARD XR PLUGIN

To add the Google Cardboard XR Plugin, navigate to **Window -> Package Manager**:

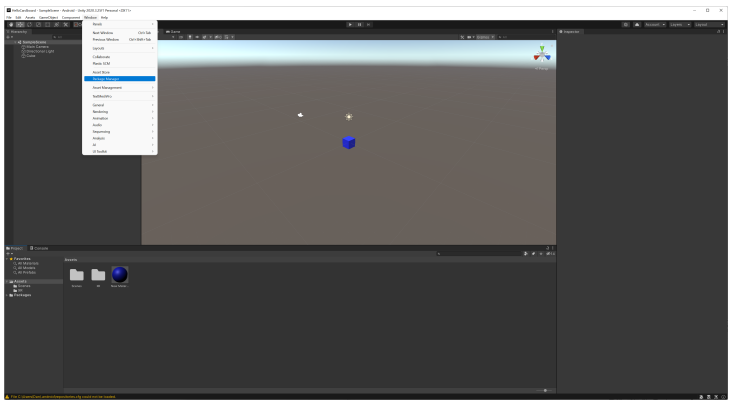


Figure 16: Go to the Package Manager.

In the Package Manager Window, hit the '+' button and select one of:

- Add package from disk...
- Add package from git URL...

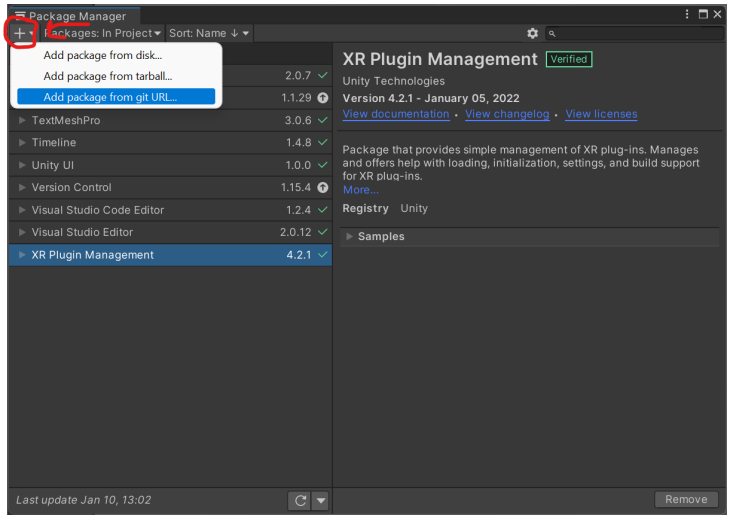


Figure 17: Adding a package from a git URL.

Adding a package from a git URL is the easiest, but will only work if you have git installed and accessible on your environment PATH variable. If this is the case for you, select "Add package from git URL..." and paste this url into the text entry field: `https://github.com/googlevr/cardboard-xr-plugin.git`.

If this doesn't work, it probably means that Unity couldn't find git on your system. Not to

worry! In this case, instead navigate to https://github.com/googlevr/cardboard-xr-plugin/releases and download the latest release (when I wrote this, this is the direct link to the latest release link).

Extract the .zip folder you downloaded and make note of where this is located. Go back to Unity and select "Add package from disk..." In the pop-up window, select the "package.json" file within the unzipped folder you downloaded.

In either case (adding from git URL or disk), Unity will take a minute to compile the new package. Once successful, it will look like the following picture:

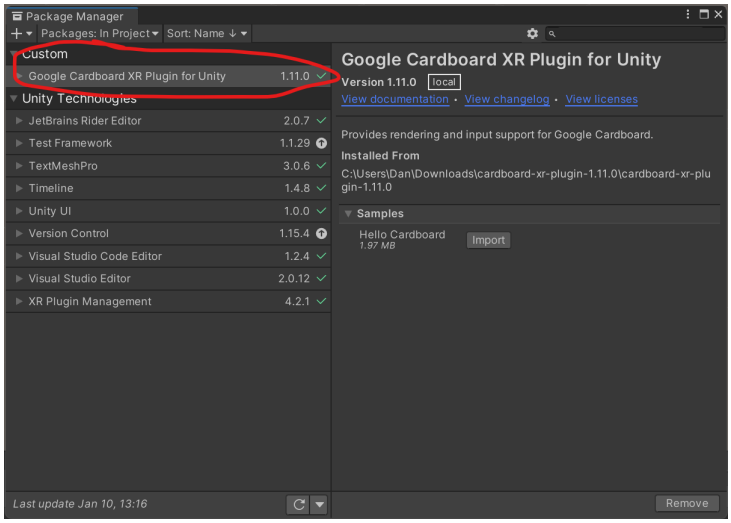


Figure 18: We have successfully imported the Google Cardboard XR Plugin.

Now go back to **Build Settings -> Player Settings -> XR Plugin Management** and there will now be a checkbox for the Cardboard XR Plugin. Check this box:

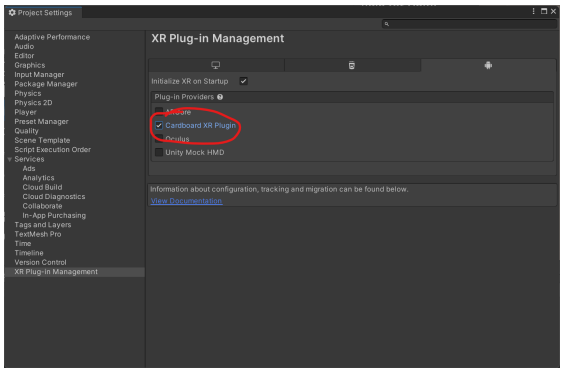


Figure 19: Check the Cardboard XR Plugin box.

As an aside, it should also automatically have the "Initialize XR on Startup" box above checked. Leave this checked (or check it if for some reason it is not checked).

OTHER PLAYER SETTINGS

We now have to set a variety of other player settings. Go back to the "Player" tab (within Build Settings -> Player Settings), make sure you are on the Android settings tab within the Player window, and notice there are 5 sub sections: Icon, Resolution and Presentation, Splash Image, Other Settings, and Publishing Settings.

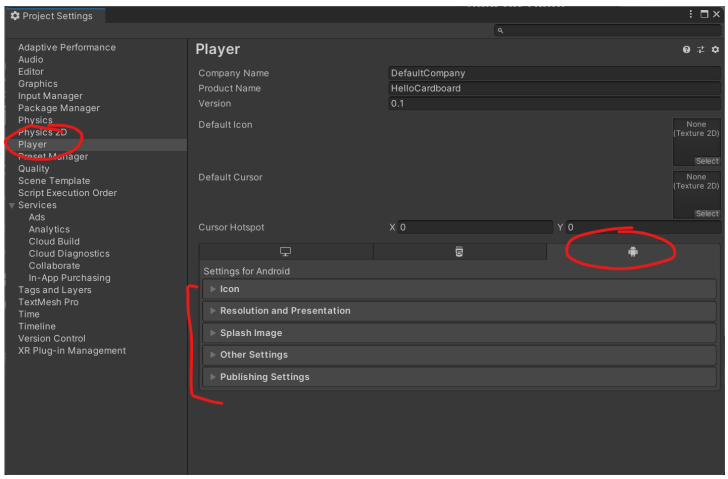


Figure 20: We will need to configure several settings in this window.

The settings we need to enable are as follows:

- Resolution and Presentation
  - Set the Default Orientation to Landscape Left or Landscape Right.
  - Disable Optimized Frame Pacing.
- Other Settings
  - Choose OpenGLES2, or OpenGLES3, or both in Graphics APIs.
  - Select IL2CPP in Scripting Backend.
  - Select desired architectures by choosing ARMv7, ARM64, or both in Target Architectures.
  - For Minimum API Level, select Android 7.0 'Nougat' (API Level 24)
  - (Optional) Select Require in Internet Access.
- Publishing Settings
  - Select Custom Main Manifest.
  - Select Custom Main Gradle Template.
  - Select Custom Gradle Properties Template.

Selecting the three boxes in Publishing Settings creates 3 new files within your Assets root di-

rectory. We need to edit these files. First, locate **Assets/Plugins/Android/mainTemplate.gradle**. Open it with a text editor and add the following lines to the dependencies section:

```
implementation ‘androidx.appcompat:appcompat:1.0.0’
implementation ‘androidx.constraintlayout:constraintlayout:1.1.3’
implementation ‘com.google.android.gms:play-services-vision:15.0.2’
implementation ‘com.google.android.material:material:1.0.0’
implementation ‘com.google.protobuf:protobuf-javalite:3.10.0’
```

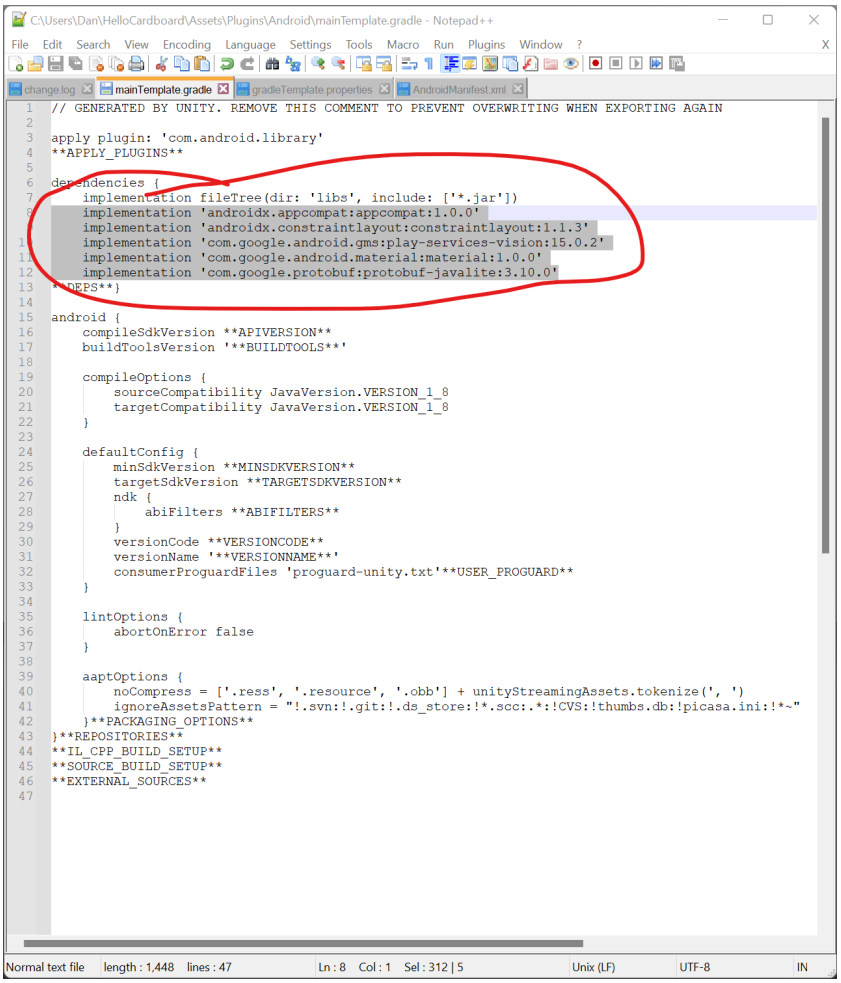


Figure 21: Add the lines above to the dependencies section in the mainTemplate.gradle file.

Next, locate **Assets/Plugins/Android/gradleTemplate.properties** and add the following lines:

```
android.enableJetifier=true
android.useAndroidX=true
```

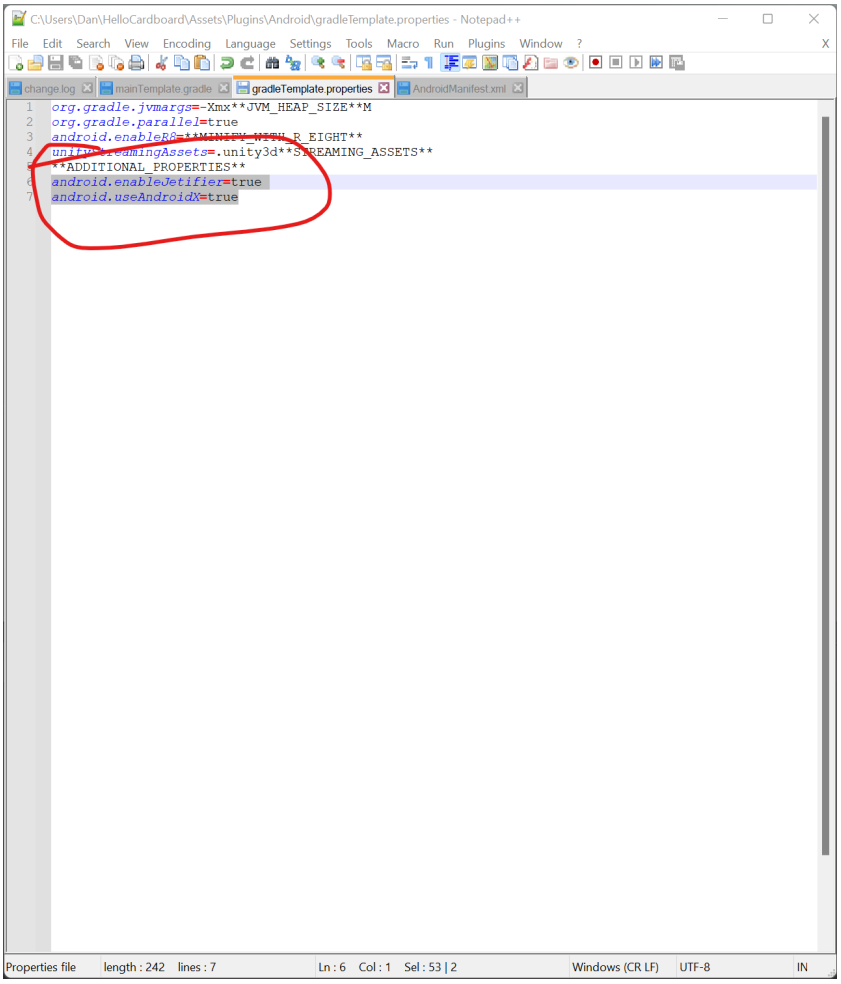


Figure 22: Add the lines above to the gradleTemplate.properties file.

Finally, locate **Assets/Plugins/Android/AndroidManifest.xml** and add the following attribute to the application tag:

```
<application android:requestLegacyExternalStorage=“true”>
```

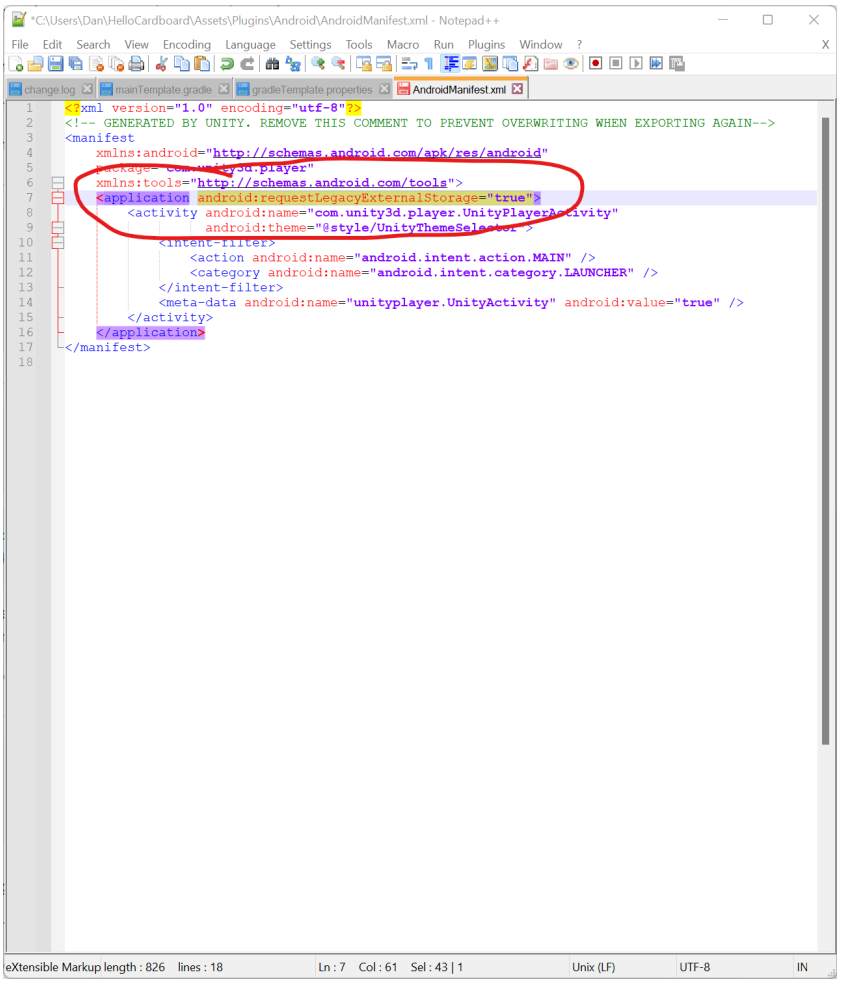


Figure 23: Add the attribute above to application tag in the Android Manifest.xml file.

Now the project is ready to be deployed to your Android device. Plug in your Android phone (make sure USB debugging is enabled) and, back in the Build Settings window, click the "Build and Run" button:

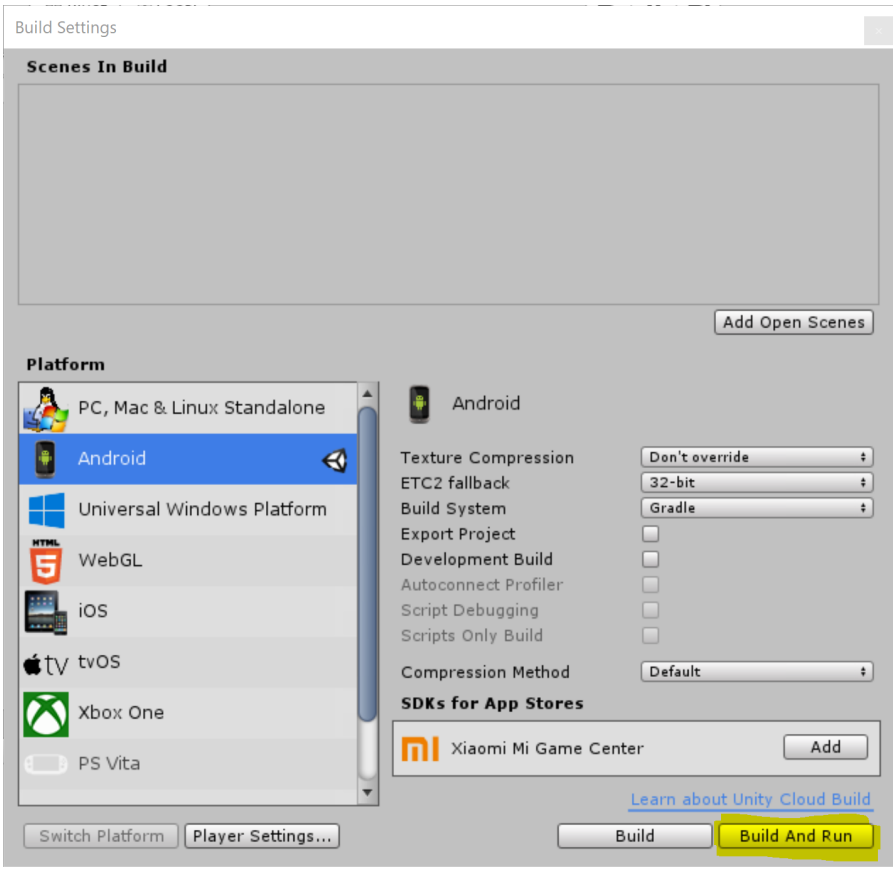


Figure 24: Click the "Build and Run" button in the Build Settings window to deploy your first VR app!

When you click the "Build and Run" button, Unity will prompt you for a location to save the .apk file (that's the package file that installs the app on your Android device). Create a new empty folder named "Builds" within your Unity project folder at the same level as the Assets folder (**never** place your builds within any Unity generated folders such as the Assets, Library, or ProjectSettings folders as this can break your project). For an example, see Figure 25.

Give your .apk a descriptive name (e.g., "HelloCardboard") and hit "Save" after navigating into your new Builds folder. Unity will now attempt to create the build (.apk file) and install it to your Android device (this will take a minute or two). If all goes well, your phone will automatically launch the application.

Assuming your application launches, you might immediately notice that something is wrong! If you try to put you phone in the Cardboard casing, things won't look correct. Examining the application, you'll see that you only have a single view (rather than two views, one for each

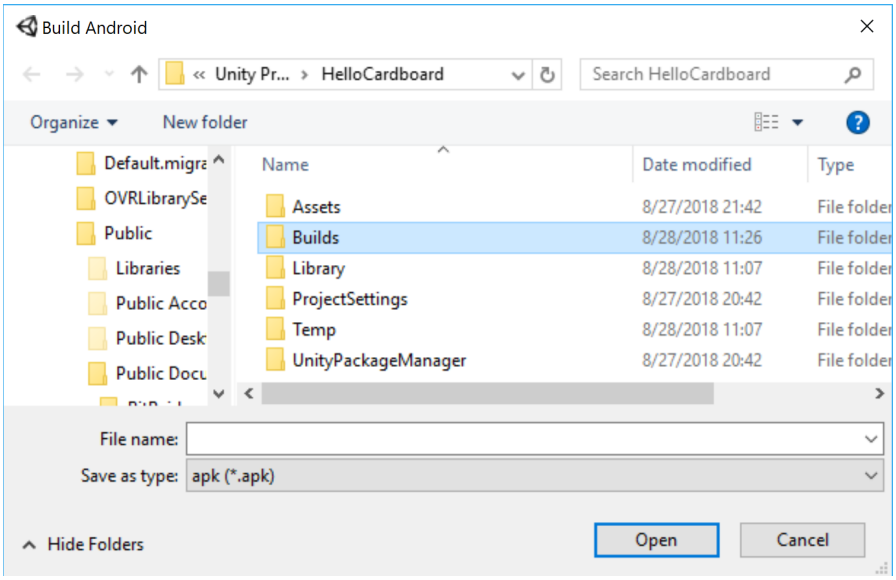


Figure 25: Create a “Builds” folder within your Unity Project folder to store the .apk.

eye). This is because we haven't told Unity that our camera should be tracking user pose and rendering images for both eyes. To do this, go back to Unity and click on your camera. In the inspector window, select **Add Component -> XR -> Tracked Pose Driver**:

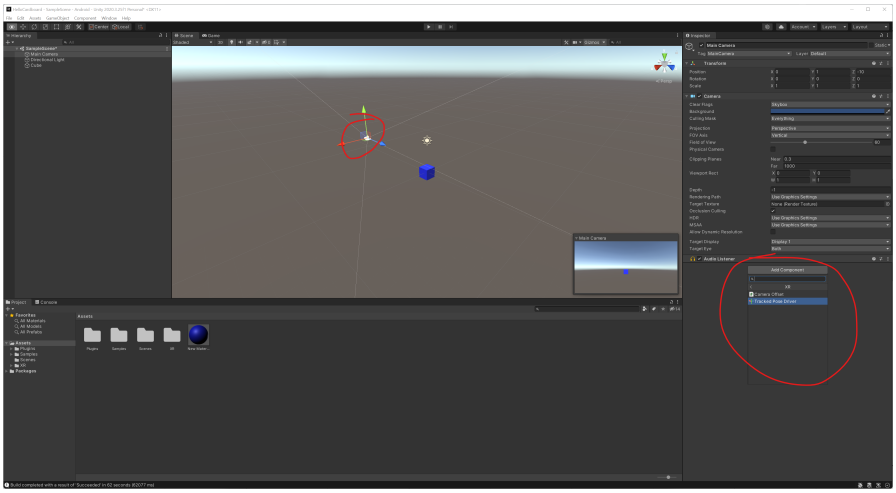


Figure 26: Add the “Tracked Pose Driver” component to the camera in your scene.

You can leave all the Tracked Pose Driver settings on their defaults. Now go back and build and run again and you application should launch successfully! Once the application launches, you can unplug your phone, put it in your Cardboard casing, and look around! Note that you may need to turn around a bit to find your cube (e.g., you may be facing directly away from the cube when you first put your device on).

If you want to install your app on another Android device, you can simply access the .apk file you just created from that Android device (e.g., you could put the .apk on Google Drive and then access it in Google Drive from your Android device). For example, I've linked to my HelloCardboard.apk on Sakai; you can navigate to this via an Android device and install the app on your Android phone if you like.

NOTES

Extra room to take notes on the build procedure is provided below:

STEP 5. MAKE THINGS INTERESTING

Congratulations on building and deploying your first VR application! However, it is a little boring. Let's make a slightly more interesting application. You might have noticed that Cardboard and Cardboard-like devices have a single input button on the top right. If you look at the design of this button, you will see that "clicking" this button actually taps the screen of the Android device; in essence the button simulates a screen tap. Let's leverage this to make an application in which the user can click the button to shoot a ball at a target.

STEP 5 INSTRUCTIONS

Go back to Unity and save the scene you have created so far by selecting File->Save Scenes from the Toolbar. Give the scene a descriptive name (I called mine "HelloUnity"). You will see the scene you saved show up as a new Asset within the project window. Our next app isn't going to use the cube we created, or anything else in that scene, so let's create a new scene by selecting File->New Scene from the Toolbar. Now you should have a new blank scene with just the default objects (a camera and light).

Once you have a blank scene, create a sphere in the same way you created a cube: GameObject->3D Object->Sphere. You can leave the sphere grey, apply your blue material from the last scene, or create a new material for it. We are going to use this sphere to create a *prefab*—a template for an object(s) that can be cloned and added to a scene. In other words, the sphere we make here will serve as a template that is instantiated every time the user presses the Cardboard button. Any changes to this prefab will thus affect all spheres that are created when the user actually runs the program. First, we need to add a Rigidbody component to the sphere, which will let it be affected by gravity and hit the target. To do this, click on the sphere in the scene view and then click on Add Component->Physics->Rigidbody:

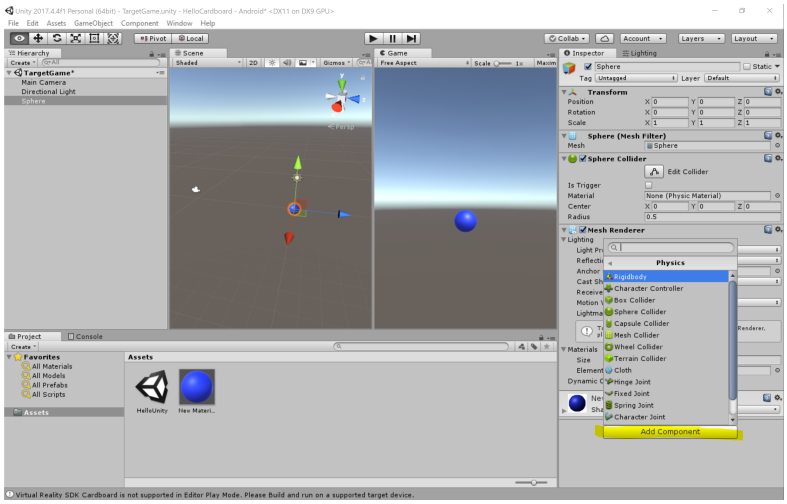


Figure 27

You can now see the Rigidbody variables in the inspector. You can leave the default values, or play with them later and see how they affect the ball. Try hitting the play button in Unity—what happens to the sphere? What has adding the Rigidbody component done?

Next, we need to add a script to the sphere that will let us instantiate this object every time the user clicks the Cardboard button. With the sphere still selected, go to Add Component->New Script and enter a name for the script (I called mine “BallPrefab”).

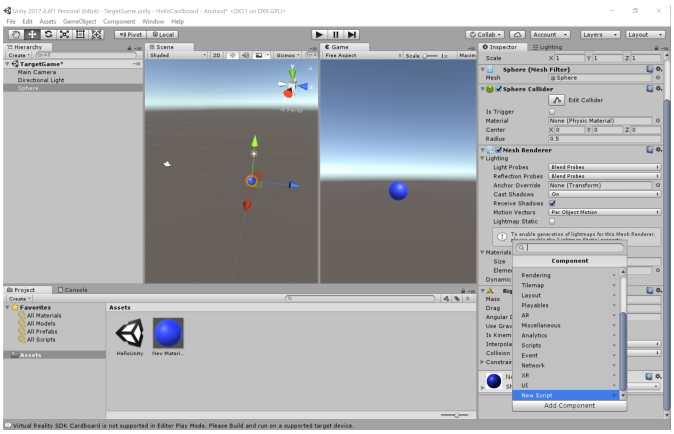


Figure 28

Double-click on the newly created script in the Assets folder within the Project window to open Visual Studio. You can mostly leave the script as is; you only need to add one new line of code: [RequireComponent(typeof(Rigidbody))] which goes just above the class definition:

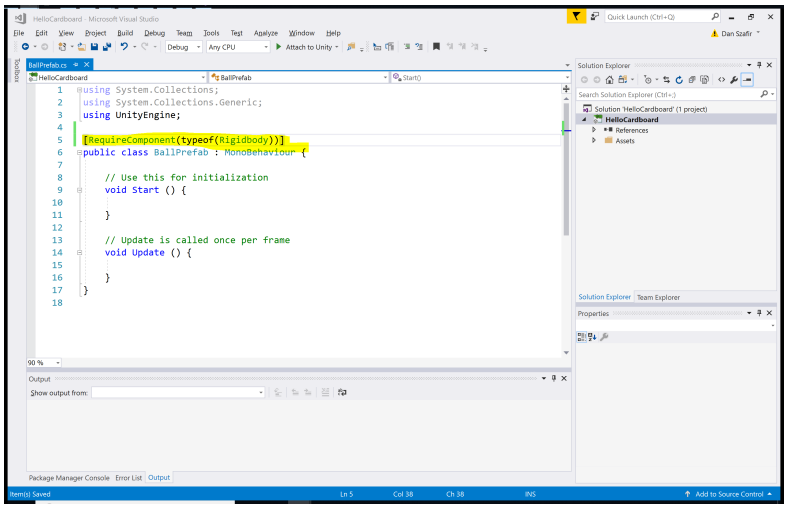


Figure 29

This line of code requires that whatever object this script is attached to has a Rigidbody component, which we had just added to the sphere. If you attach this script to an object that does

not have a Rigidbody component, you will get an error.

Now it is time to tell Unity to treat our sphere like a prefab. To do this, drag the Sphere from the Hierarchy window into the Assets folder in the Project window:

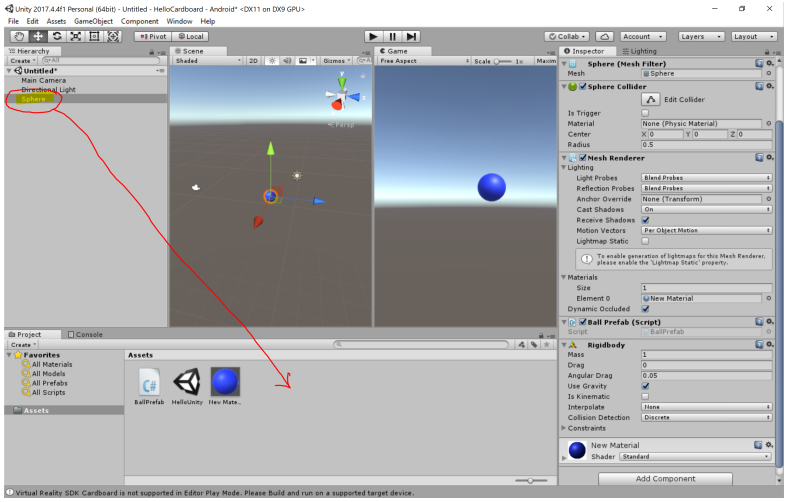


Figure 30: Create a prefab from the sphere by dragging it from the Hierarchy window to the Assets folder in the Project window.

Once you have done so, you will see the sphere show up as an Asset. Rename it now so that you don't confuse it with the standard primitive sphere that you could make using GameObject->3D Object->Sphere. To rename it, click on the prefab name in the Assets window and give it a new name (I called mine "Ball"):

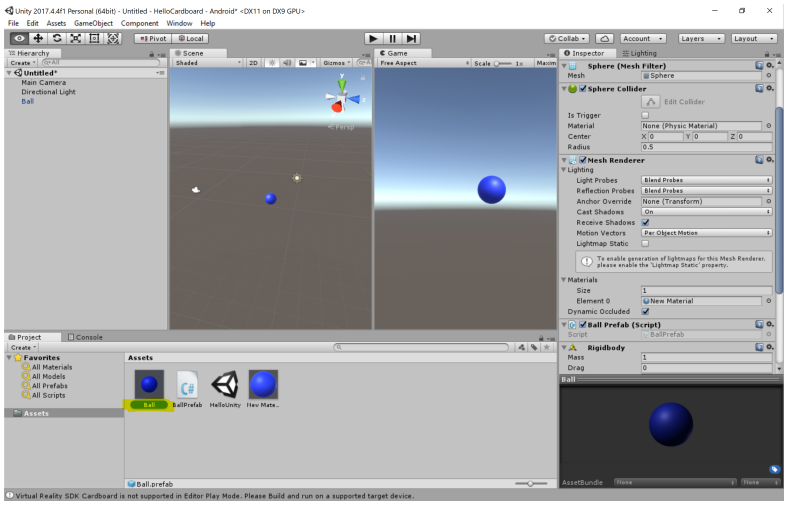


Figure 31

At this point, you can delete the sphere from your scene (do not delete the Ball prefab we

just created from the Assets folder). To do so, click on the sphere in the Scene view or the Hierarchy window and then press delete.

Next, let's make a target. We will use a plane for the target as Unity doesn't have built-in geometry for a circle or hoop (torus). Create a Plane using GameObject->3D Object->Plane. Notice the plane is created as if it was a floor. We want to instead position it in front of the camera. Edit the settings of the plane's Transform (in the Inspector window) to match the picture below:

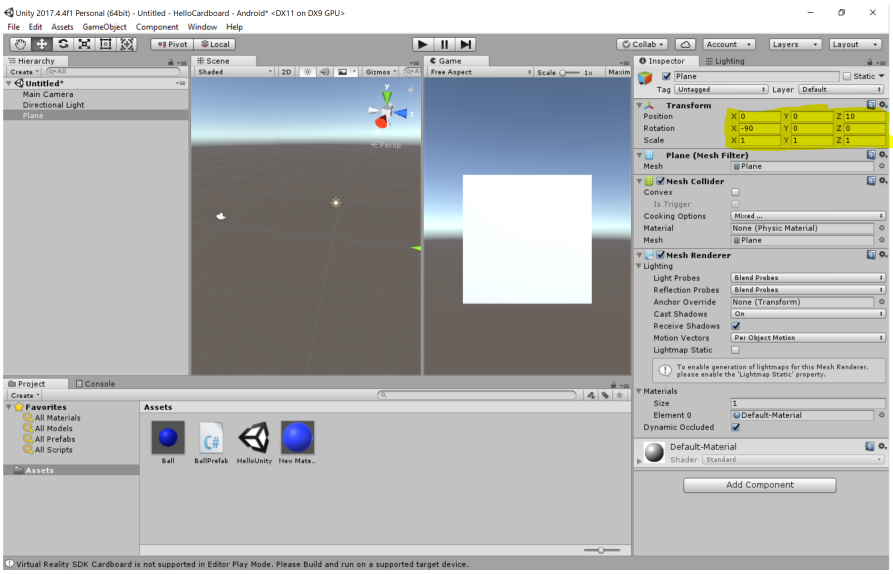


Figure 32

Notice we are rotating the Plane by $-90°$ in the x direction; what happens if you instead rotate it by positive $90°$? Why do you think that is? We will discuss this further when we talk about procedural geometry.

Let's make the plane look more like a target. We will do this by applying a *texture* based on an image. We will talk about textures later on in the course. For now, download the target image here and drag it into your Assets folder in the Project window. From there, drag the target from the Assets folder in the Project window onto the Plane in the Scene window:

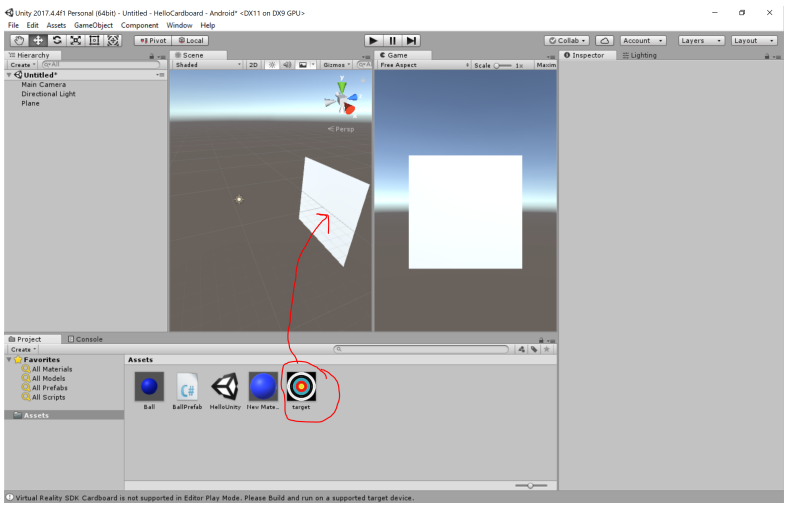


Figure 33

Notice that Unity automatically created a new "Materials" folder in your Assets folder, within which it created a new material for the Plane that holds the target texture. Your scene should now look like:

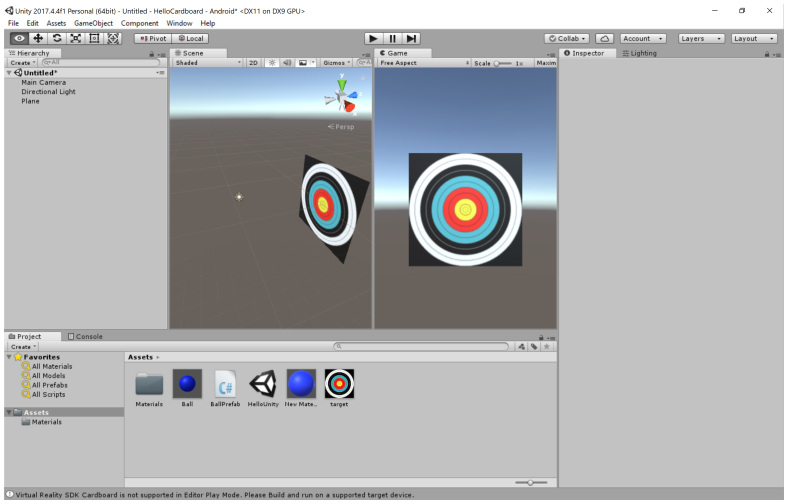


Figure 34

We are almost done. The final piece is to add logic such that when the user clicks the Cardboard button, a ball shoots out wherever they are looking. To do this, we will create another script, which will be attached to the camera. Select the camera, either by clicking it in the

Scene view or the Hierarchy window, and in the Inspector window add a new script called "ButtonClick." Open this script in Visual Studio.

The first thing we need to add to the script is a variable to hold what type of prefab we want to instantiate—in this case, a BallPrefab. Add the following field to your script:

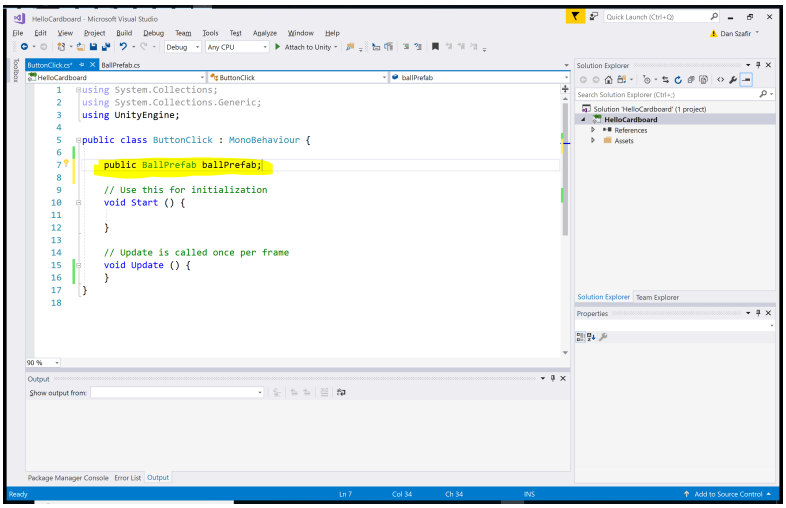


Figure 35

Save your code and go back to Unity. When you select the camera, notice there is now a new field called "Ball Prefab" within the ButtonClick script in the Inspector window. Click and drag the Ball prefab object from the Assets window into this field:

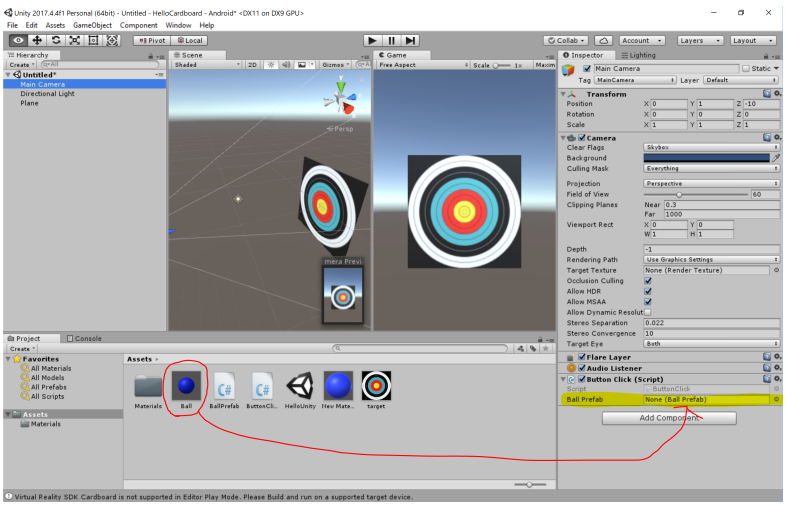


Figure 36: Click and drag the ball prefab into the new field you created in your script.

As a result, the ballPrefab object in your script will be instantiated with the Ball prefab we created earlier. Now go back to your ButtonClick script in Visual Studio and add the following code:

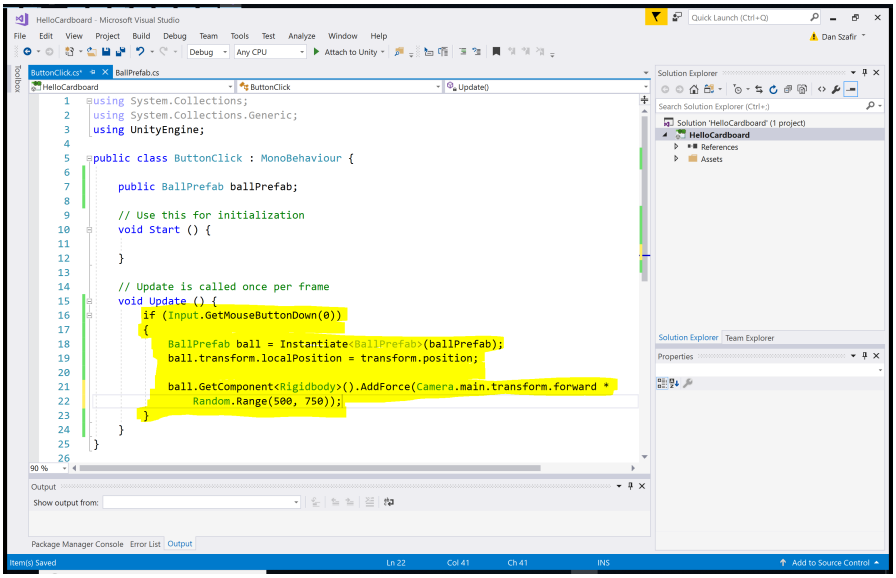


Figure 37

What does this code do? The if statement checks to see if the Cardboard button is being clicked (equivalent to checking for a screen tap or, when run on a PC, a left-mouse click). If the user clicks, we instantiate a new ball prefab object, place it at the camera, and then shoot it out along whatever direction the camera is facing (i.e., where the user is looking). To add a bit of variation, we randomly choose a force multiplier (which will affect the speed of the ball) in the range of 500–750 (feel free to play with these numbers).

Once you have added this code, save your script and go back to Unity. Hit the play button and click anywhere in the Game view and you should see spheres be emitted from the camera each time you click. If everything is working, build and deploy to your Android device following the same process you went through for the basic cube scene above in Step 4 and try it out in Cardboard.

STEP 6. CUSTOMIZATION

At this point, you should have a basic sense of how to create Unity primitives (cubes, spheres, etc.), how to create new materials for them, and how to do some basic interaction. Your final task is to create your own game using these elements. Your game does not have to be complex for this assignment (e.g., a simple example might be adding some sort of scoring metric to the target scene from Step 5). You can extend the target scene or create an entirely new scene, but it should do something interesting and be something you can justify as a game based on our class discussion of what makes a game. If you want, you may wish to look through some more Unity tutorials (there are many online) or use Google (or DuckDuckGo for the privacy-inclined) to figure out how to achieve a desired effect for your game.

At minimum, you should:

1. Make an interesting scene using built-in Unity primitives
2. Make materials to color your objects or use textures (try finding pictures online)
3. Have some element of interaction (e.g., something happens when the user clicks a button)
4. Deploy your scene to Google Cardboard and take a picture of it running

If you would like more inspiration, you can also check out Google's cardboard demo scene. To do this, go back to **Window -> Package Manager** and click on "Google Cardboard XR Plugin for Unity" (the package you added back in Figure 18). In the dialog on the right, click the "Samples" dropdown and then hit the "Import" button (note that in my screenshot below it says "Reimport" since I had already imported this before taking the screenshot):

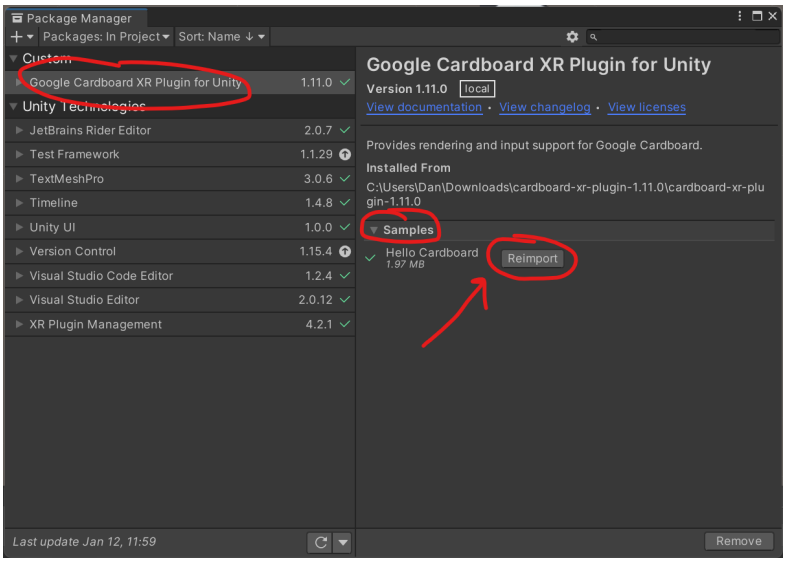


Figure 38: Click the "Import" button (labeled "Reimport" in the image above) to check out Google's cardboard demo scene.

Once you do this, you'll see some new things have been added to your Assets folder. Click through **Samples -> Google Cardboard XR Plugin -> 1.11.0 -> Hello Cardboard -> Scenes** and open the "HelloCardboard" scene. You should see something like the picture below:

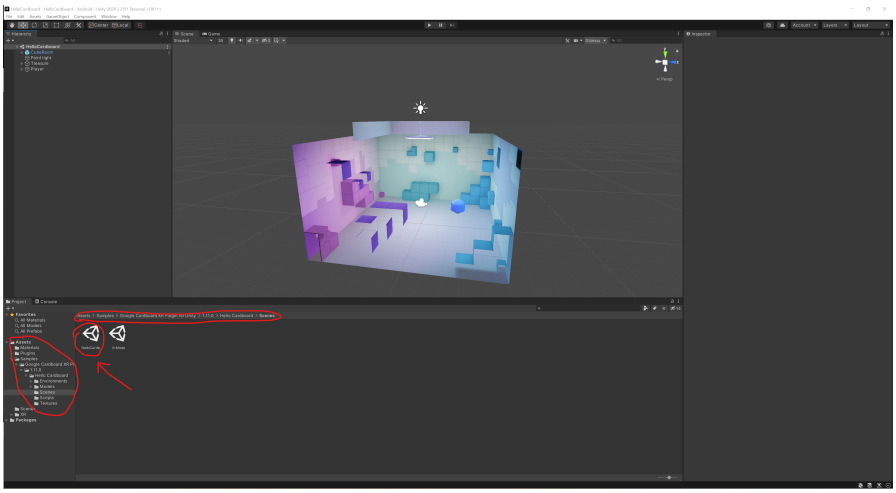


Figure 39: Here is Google's cardboard demo scene.

You should be able to build and deploy this app to your phone as is. Try playing around with it and see what it does. In Unity, you'll see some objects have scripts attached to them for interactivity (e.g., you'll see the Camera has a CameraPointer script that figures out what object the user is looking at and the Icosahedron has an ObjectController script). You might want to use these or build off of them. From here, its up to you!

Grading and What to Hand In

Turn in a zipped folder containing:

- Your project folder OR a document with a link to a GitHub repo where your project folder can be found (I prefer a GitHub link)
- Your .apk file (or include this in your GitHub repo)
- A short (1 page or less) write up describing your game and why you believe it is a game (based on our class discussions of what is a game). Include in this links to any additional sources of inspiration or assets you used in this assignment. You can alternatively just include this as part of your GitHub readme if you give me a GitHub link.
- A photo of your project running on Cardboard (you might need to borrow another phone to take the picture). The purpose of this is so that I know you were successful in building and deploying your project. An example of such a picture is found below:

Figure 40: An example photo demonstrating a successful deployment.

Grading will be as follows:

- You will get a *Check* (roughly corresponding to 50%) if you turn in a viable and working submission that simply has a cube (i.e., you made it through Step 4).
- You will get a *Proficient* (roughly corresponding to 75%) if you turn in a viable and working submission that matches the target game (i.e., you made it through Step 5).
- *Exemplary* grades (>75%) will be awarded for making a particularly interesting application beyond the target example (i.e., you completed Step 6).