CS 5360/6360: Virtual Reality

Startup Phase: Mini-VR Project

TASK

In this assignment, you will gain experience in basic Unity VR development. To earn the full credit, you must submit the assignment with at least 4 other partners (no more than 5 other partners); students may not pair up across the undergraduate/graduate sections. Use the Canvas discussion board / Slack to help find partners. Once you have identified partners, sign up for an Assignment Group in the People page on Canvas.

Starting Resources

This assignment includes a .zip file named *Assignment 1 Resources.zip* containing a Unity package and images to be used in Part 1-2 and Part 2. There are no additional starting resources.

Preliminaries

Setup your project in the git version control system. There are several distinct tutorials on how to merge Unity with git, a distributed version control system. You are required to use the .gitignore file located within the Assignment Resources (note, you will not see this file if you cannot see hidden files/folders on your operating system). You will be submitting this project via Github.

Both parts of this assignment will be placed in the same repository. The directory structure you will have is illustrated in Figure 0.

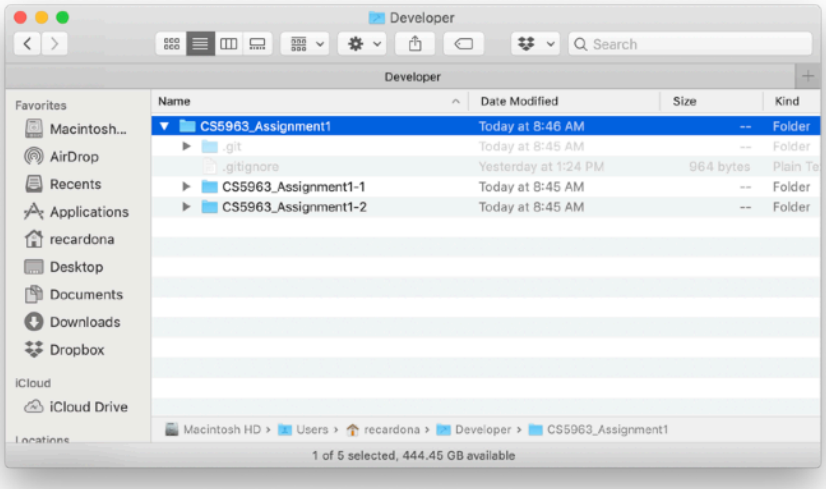


Figure 0. Anticipated project structure for submission.

Download the latest 2018.2.x version[1] of the Unity game engine. Open Unity and create a new Unity project by following the prompts when you click on the "New" button on the top, right-hand side of the window, as shown in Figure 1. Name your project *[class number]_Assignment1-1.* For example, if I am enrolled in the Undergraduate version of this course, the project name would be: *CS5963_Assignment1_1.* Fill out the remaining details to your heart's content, but keep the template in 3D.

To familiarize yourself with the very basics of the Unity interface, check out the following official Unity "Getting Started" articles:

- Learning the Interface: https://docs.unity3d.com/Manual/LearningtheInterface.html
- The Main Windows: https://docs.unity3d.com/Manual/UsingTheEditor.html

If you want to familiarize yourself with Unity through a rather hands-on approach, I highly recommend you check out the tutorials provided by Catlike Coding; if you can afford to do so and feel so inclined, consider making a one-time donation to them!

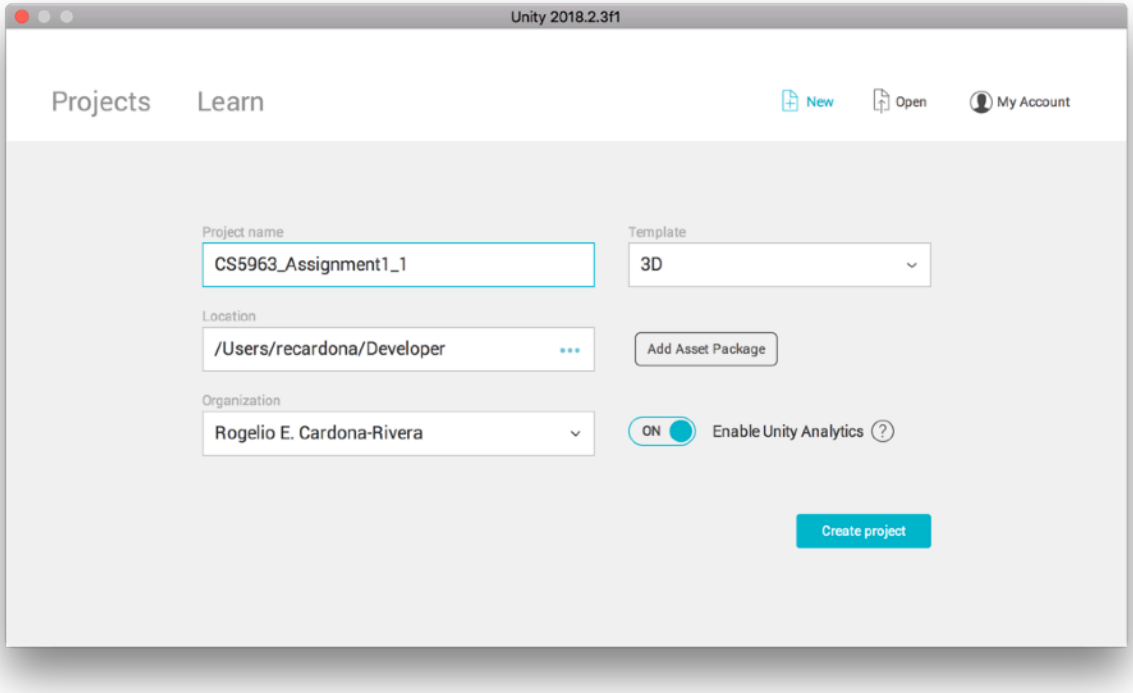


Figure 1. Unity "New Project" Wizard.

[1] At the time of writing, there was an incompatibility with the Oculus Integration and all Unity 2018.3.x versions and Unity did not recommend upgrading to any 2018.3.x Unity Editor version for Oculus developers. It is possible that this restriction no longer applies; consult the Unity/Oculus documentation for more information. You are welcome to use whatever version of Unity works; this assignment was written using 2018.2.x.

Part 1.1 — “You’ve Taken Your First Step Into a Larger World.” (2 Points)

The Room (0.25 points). You will build a cubic room, whose sides are made out of six planes. Make sure to orient these planes so the visible sides face inwards, and ensure that the player cannot walk through any of them. The room should be 15x15x15 Unity units (i.e. meters).

First, create a plane. It can be found in the top bar menus, under *GameObject* > *3D Object* > *Plane*, as show in Figure 2.

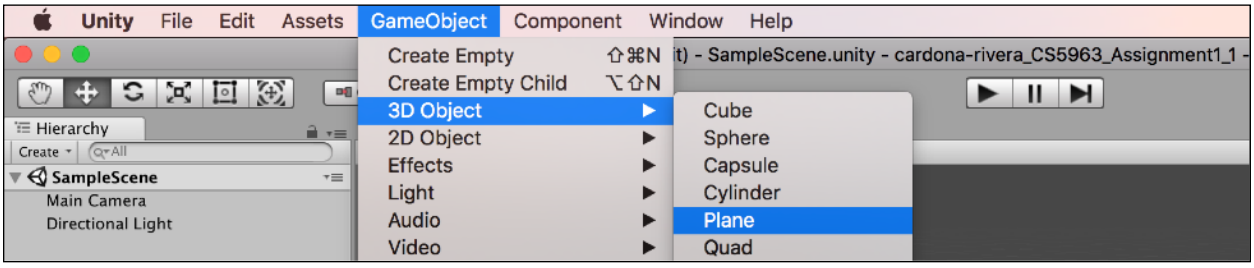


Figure 2. Creating a 3D Plane GameObject in Unity’s GameObject menu.

By default, the plan is 10x10x10 units. In order to make your room 15 units wide (on the X and Z axes), you have to scale the plan. On the right-hand side of the default editor layout, you will find the Inspector window; this window provides details about the currently selected object within Unity. Select the plane in the Scene view and the Inspector will show information related to the plane. At the top of the window, you will find the *Transform* area, where you can manually type in 1.5 in both the X and Z Scales. This will make your plane 15 units wide and long.

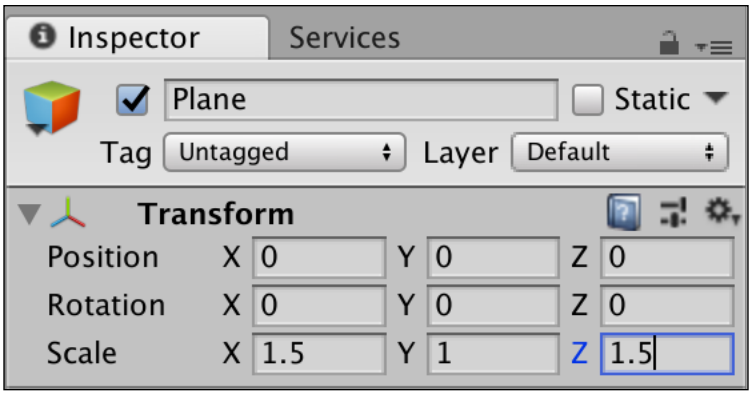


Figure 3. Scaling a 3D Plane GameObject in Unity's Inspector window. Note, by default a plane has no thickness, so the value in the "Y" slot can be any positive integer.

By default, your scene has a directional light in it. Think of this light as a "sun" — it is a light source that illuminates your entire scene from a specified angle, from far far away. Your planes do not block this light because planes only block (and render) light from one side. Bear this in mind when creating other GameObjects in Unity in the future. Delete the directional light by navigating to the left-hand side of the default editor layout, right-clicking on the Directional Light in the Hierarchy window, and left-clicking on delete, as illustrated in Figure 4 (don't worry, you'll add new lights in later).

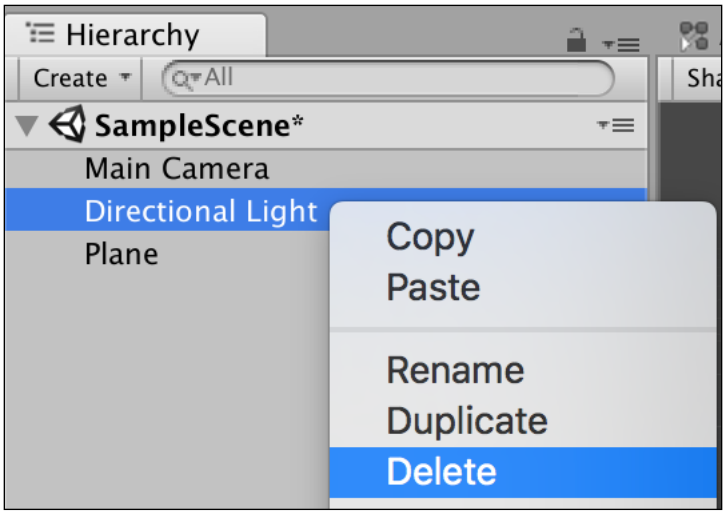


Figure 4. Deleting the default Directional Light GameObject in Unity's Hierarchy window.

In the Hierarchy window, select your plane, and right-click on it. Left-click on *Duplicate* to duplicate your plane. From there, change the new plane's rotation and position to make it one of the walls or ceilings. Unity measures position from the center of the GameObject, such that if you want your walls to match up with the floor (at height 0), your walls will need to be at 7.5

units off the ground, as illustrated in Figure 5.

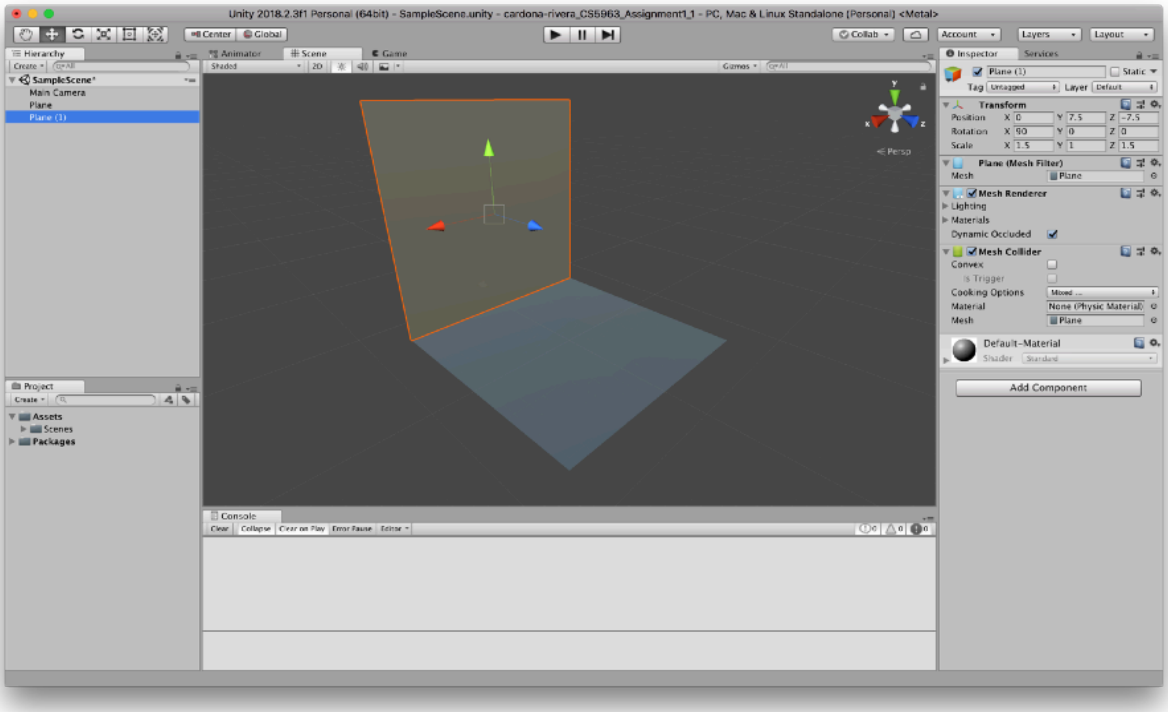


Figure 5. Duplicating, rotating, and aligning a plane in the Unity editor.

Player (0.25 points). Download and install the Oculus OVR plugin[2] for Unity by going to the Oculus website, downloading their Unity package, unzipping their package file, and then going to *Assets* > *Import Package* > *Custom Package* (which prompts you to find the file on your file system). If you have imported it successfully, you should have a folder titled *Oculus* in your *Assets* folder.

In this class, you will be primarily using two prefab GameObjects from this package, both found in Oculus > VR > Prefabs:

1. *OVRCameraRig* — a camera for the Oculus, that handles all of the movement and position tracking for the Oculus, as well as the rendering on the Oculus display.

2. *OVRPlayerController* — a more complicated version of the *OVRCameraRig* that includes the camera from the rig but which also includes basic joystric movement controls and a capsule-shaped collider (that prevents the player from walking through solid objects). The *OVRPlayerController* is what is highlighted in Figure 6.

[2] Note, these assignments assume that you are working with the Oculus plugins for the Unity game engine. Due to the experimental nature of the course, you are welcome to use whatever VR hardware you can get your hands on, but it is on you to understand how to satisfy the requirements of the assignment in a manner similar to the instructions outlined here.

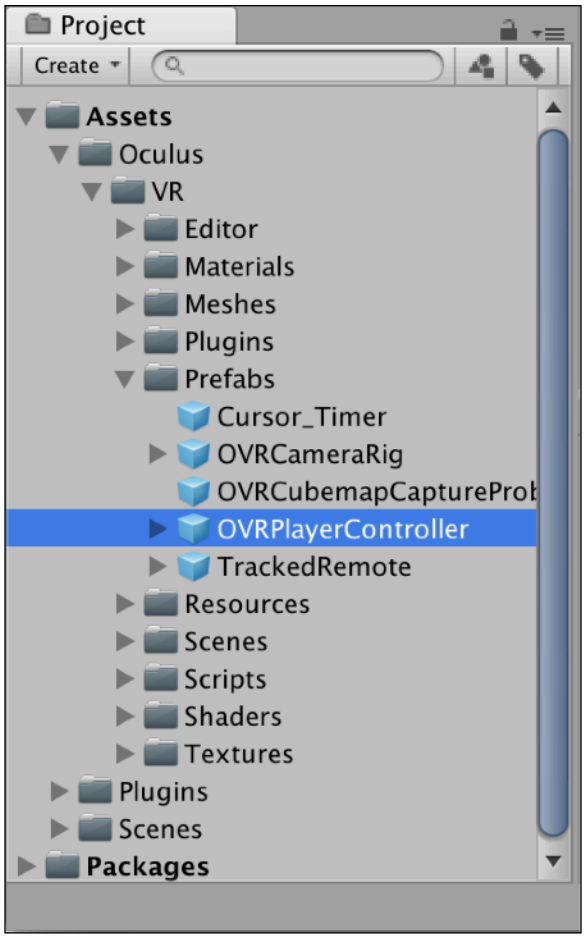


Figure 6. The Oculus VR Package, with the selected OVRPlayerController.

Place on *OVRPlayerController* prefab in the scene, at (0, 1, 0), and delete the *MainCamera* GameObject.

To finish enabling your VR experience, go to *Edit* > *Project Settings* > *Player*, which will open up the player settings in the *Inspector* window. In *XR Settings*, check the box that says *Virtual Reality Supported*. With this enabled, your scene should render inside the Oculus when played.

If you have any issues doing any of the above, please post to the Canvas Discussion page. Remember, the Canvas is your first line of support.

Lighting (0.25 points). Before you continue with this section, be sure to familiarize yourself with the different types of light within Unity. You will then place a point source of light in the center of the roof of the room (this light will be made to change color by pressing the *Tab* key later, detailed in the scripting section). To place the point light, go to *GameObject* > *Light* > *Point Light*; this will bring a point light into your scene. In the *Inspector*, place it at (0, 15, 0). The GameObject should have a *Light* component as illustrated in Figure 7.

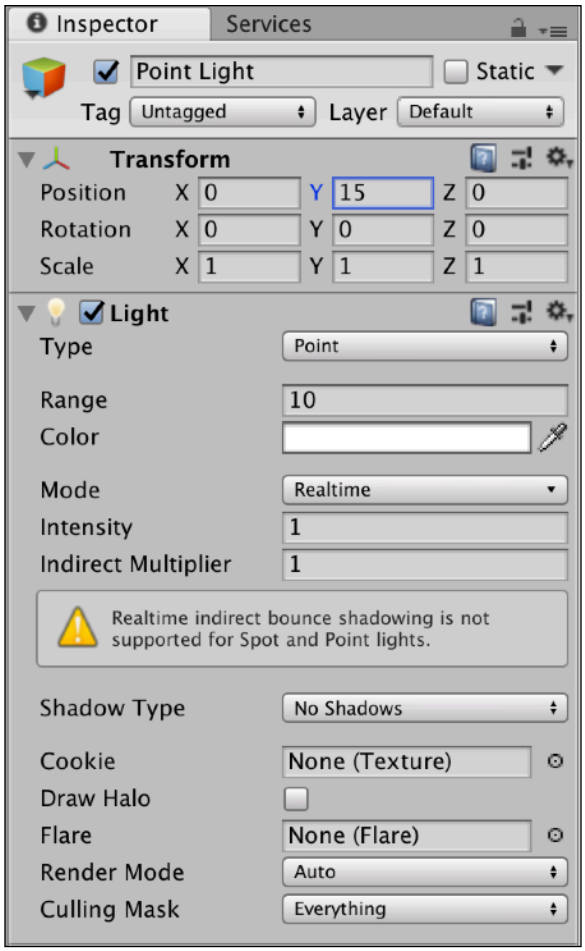


Figure 7. The Light Component within a Point Light GameObject, within Unity's Inspector window.

Of primary importance are the *Range* (i.e. the radius of your light), *Color*, and *Intensity* attribute values. Set the *Shadow Type* to *Soft Shadows*; if you are not already familiar, I encourage you to read up on Unity Shadows**.** Set the *Mode* to *Realtime*, and read up on Light Modes in Unity**.** Set your range and intensity so that your room is brightly lit.

Planet and Moon (0.25 points). In this section you will: (a) create a large sphere and have it float in the middle of the room and (b) create another smaller sphere, set it as a child GameObject of the bigger sphere, and move it 4 units away of the bigger sphere on the X axis (you will make it orbit the larger sphere in the **Scripting** section).

Create two spheres by going twice to *GameObject* > *3D* > *Sphere*. Set the scale of the first sphere to 2 in all directions and place it in the center of your room. In the *Hierarchy* window, drag the second sphere onto the first. You should obtain a hierarchy as illustrated in Figure 8.

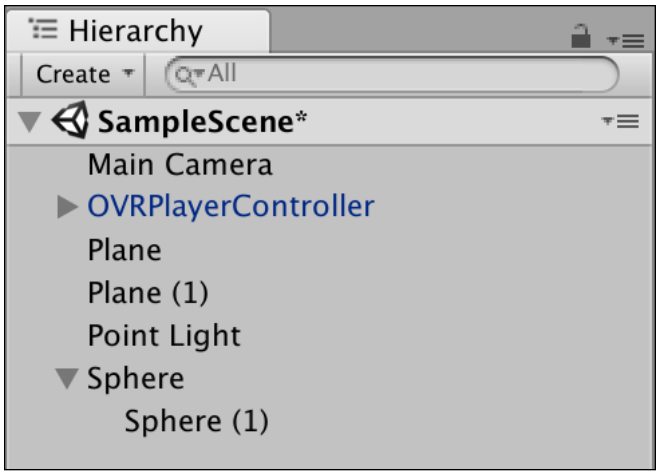


Figure 8. A Sphere placed as a child GameObject of another Sphere GameObject inside Unity's Hierarchy window.

When you drag one GameObject onto another, you set the dragged GameObject as a child of the target GameObject. Updating a parent's position, rotation, or scale will transitively apply to all the children. Further, the child's origin position (0, 0, 0) is now set to the parent's position, not the global origin coordinate; in other words, the child's position is an offset from the parent's position. Finally, if the parent rotates, then the child will rotate about its parent's axes, not its own axes. For more information on parent-child GameObject relationships, consult the Hierarchy page of the Unity Manual. Set the position of the child sphere to be (2, 0, 0), which results in the child being four units from the parent sphere on the X-axis; can you see why?

Text (0.25 points). In this section you will put large text on a wall, detailing the controls and listing your uIDs. Feel free to experiment with what you can put on a canvas, but please keep it inoffensive.

Before you continue with this section, I encourage you to check out the Unity tutorial on Creating Worldspace UIs. Create a text canvas by going to *GameObject* > *UI* > *Text*; this will create a Unity Canvas GameObject and some text on that canvas as a child of the GameObject. It will appear as a massive GameObject within the scene, as illustrated in Figure 9. This is not what we want.

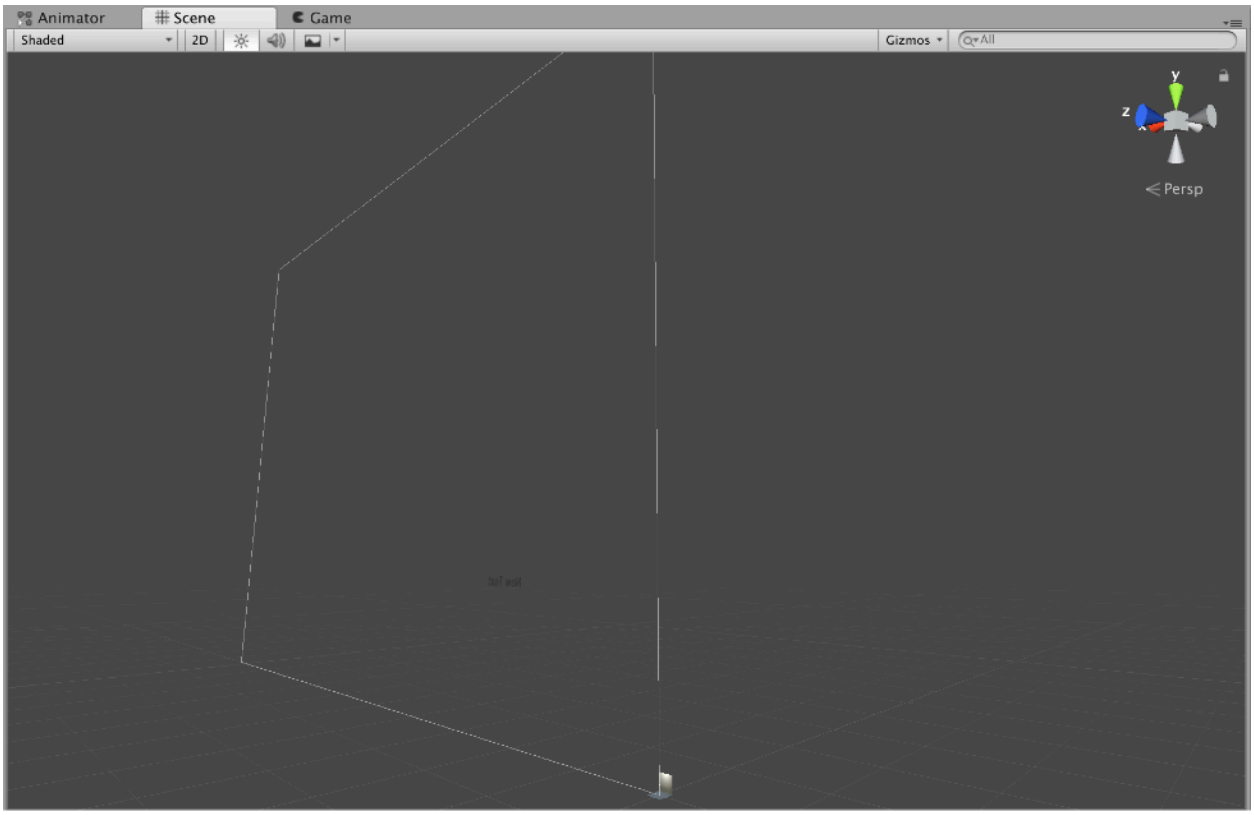


Figure 9. A Unity UI Canvas GameObject (delineated by white lines in the scene). The minuscule white figure on the bottom right-hand side of the rectangle is the room we are building in this tutorial.

To fix this issue, select the Canvas GameObject within the *Hierarchy* window. The *Inspector* window should look like what is illustrated in *Figure 10*. Change the *Render Mode* from *Screen Space - Overlay* to *World Space* — this changes our Canvas GameObject from a user interface (UI) element that is glued to the camera to a GameObject that exists and is stationary within the world. Traditional UIs do not work well in VR and I strongly advise against sticking any UI elements to the camera in your future assignments/projects. I strongly recommend that you instead attach UI elements to something in the world. For more recommendations on how to display information within VR environments, check out the Oculus VR Best Practices Vision page.

After the canvas is made to be a world space GameObject, it can be scaled to a more reasonable size. However, since the *Rect Transform*'s width and height determine the resolution of the text canvas, one cannot set them to be (for example) 5x5 unites, because the text resolution would be scaled to 5 pixels by 5 pixels. Set the *Width* and *Height* attributes within the *Rect Transform* to 1000 (that is 1000x1000 pixels). Shrink the canvas to be 10 units by 10 units; since we want the resolution to be 1000x1000, our scale must be set at 0.01 (10/1000). Make sure your Text GameObject's *Rect Transform* has the same width and height as its parent canvas, but leave the scale as 1. Place your canvas against one of the walls in your room; you want to place your text ever so slightly in front of the wall (for example, 0.001 units offset) to avoid a common problem in graphics rendering, *Z-fighting*, which happens when two objects have the same depth and the rendering engine can't figure out which one to render first.

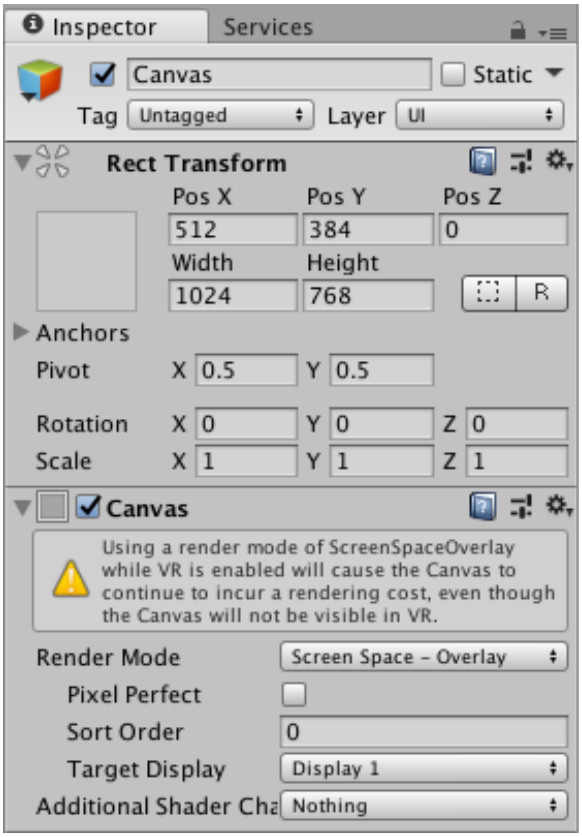


Figure 10. Unity's Inspector window as it appears when the UI Canvas GameObject is selected.

An example of *Z-fighting* can be seen in Figure 11.

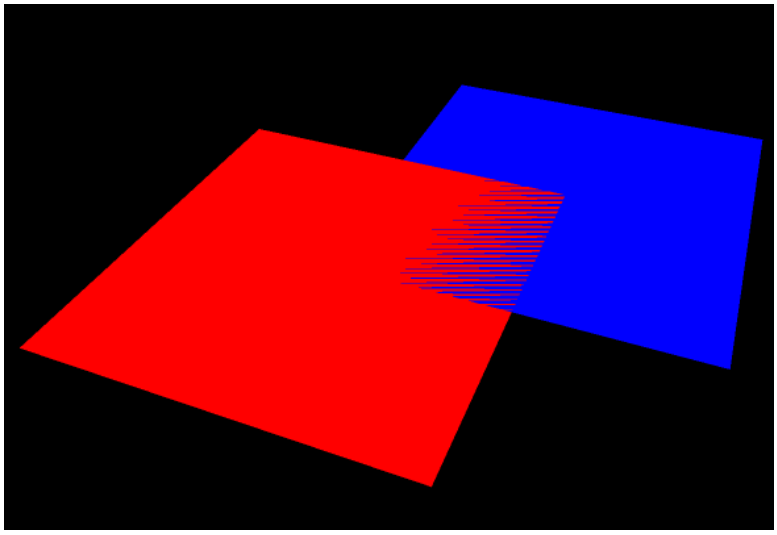


Figure 11. Z-fighting in OpenGL.

In the *Text* GameObject, set the *Color*, *Size*, *Font*, *Width*, whether it wraps or overflows, etc. Make your text you and your partner's *uID* as well as the controls for your game. Make sure it is

big enough to be readable. If the text appears blurry or jagged, then increase the width and height of the canvas and text (to increase the resolution) and scale them down further.

Scripting (0.75 points). You will need to write four scripts for this part of the assignment. Read up on Scripts in Unity and familiarize yourself with C# syntax. If you have worked with Java, it may seem familiar to you. If you are unfamiliar with programming in C#, I encourage you to check out this tutorial by tutorialspoint. You will only need the basics of objects, classes, and variables for now. Throughout the course, you will find the Script API Reference a useful source of information.

The four scripts you will create are:

1. Light Switch: Pressing the *Tab* key should change the color of the point light in the room. Pressing it repeatedly should change the color each time; i.e. have it be a toggle or a switch between a series of colors. Make sure that the color change is large enough so it is immediately apparent.

   Create a new script called *Lightswitch* and attach it to your point light. When a Unity script is attached to a GameObject, that script will run when the game is started. Furthermore, using *this* as a reference in the script will refer to the GameObject the script is attached to.

   Our first step is to get the light component of our point light GameObject. Read the Controlling GameObjects using GetComponent tutorial and then edit your script so that it looks as illustrated in Figure 12.

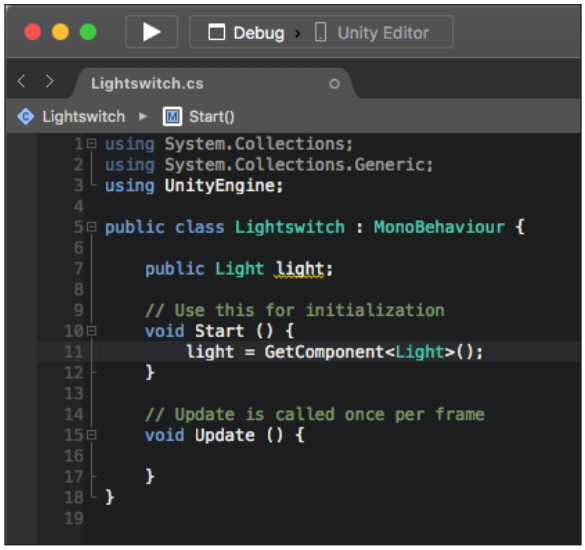


Figure 12. Accessing the Light Component via a Unity Script.

This will get the *Light* Component of the GameObject the script is attached to and store the reference in the *light* variable upon booting the game; calling *GetComponent<T>()* is equivalent to calling *this.GetComponent<T>()*, but the former is more idiomatic than the latter. To register input, we use Unity's Input library; specifically, we use the *Input.GetKeyDown* method. This method returns true when the key specified in the method parameter is first pressed down. Since we want to listen for the *Tab* key, we want to write

what is illustrated in Figure 13 in our *Update* method (which itself is called at least once per frame of rendering).

```
14      // Update is called once per frame
15      void Update () {
16          if(Input.GetKeyDown("tab"))
17              light.color = // insert color here.
18      }
```

Figure 13. Updating the light color through a key listener.

You can set a new color by either (a) creating a new color via the constructor *new Color(r,g,b)* or (b) using one of the statically defined colors in the *Color* class. How you change the light is up to you, but the light should visibly change every time we press tab; you may toggle it on/off or cycle through some pre-defined set of colors. Choose!

2. Orbit: The moon should orbit the planet sphere. The easiest way to do this is to have the planet constantly rotate. Since the moon is a child of the planet, it will also rotate around the planet. A GameObject's rotation and position is controlled by their *Transform* Component accessed with *<GameObjectName>.transform*. This class is worth studying, although the most important elements for this class are (a) the *transform.position* property, a 3D vector of the GameObject's x, y, and z coordinates in the global frame (as opposed to the local frame, which is relative to this GameObject's parent's position) and (b) the *Rotate* method. Most of Unity rotations are done using quaternions, which are a compact way to represent rotation; we will discuss quaternions in class. For now, simply know that the *transform.Rotate(Vector3(a,b,c))* will rotate you *a* degrees about the GameObject's x axis, *b* degrees about the y axis, and *c* degrees about the z axis.

   Create a script called *Orbit* and attach it to the parent sphere. In its *Update* method, add the line illustrated in Figure 14.

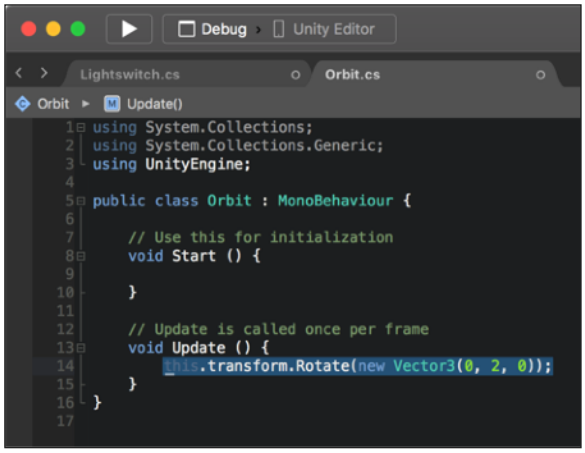


Figure 14. Rotating the parent sphere GameObject by 2 degrees about the y axis every frame.

3. Room Switch: Press the *2* key should switch to Part 1.2 of this assignment. Create the *RoomSwitch* script and attach it to the player. Use the *Input.GetKeyDown("2")*, and set the player's *transform.position* to the Vector3 corresponding to the center of your room for part 2 (wherever you end up putting it). Don't forget: the player controller needs to be 1 unit above the ground.

4. Quit Key: Pressing *Esc* should exit the game. This can be added to the *RoomScript* created previously. You will want to add the lines illustrated in Figure 15.

```
if (Input.GetKeyDown(KeyCode.Escape))
{
    #if UNITY_EDITOR
        UnityEditor.EditorApplication.isPlaying = false;
    #else
        Application.Quit();
    #endif
}
```

Figure 15. Quitting the game, regardless of whether you're in the editor or a standalone app.

Application.Quit() quits a Unity application but it will not stop a game running in the editor. The lines will check whether you are in the editor and run the appropriate command to stop it if we are.

Part 1.2 — "Come on Buddy, We're Not Out of This Yet." (3 Points)

In this part, you will be working in the same scene as for Part 1.1, but with fewer instructions. You are expected to Google the specifics. Unity has great tutorials on practically everything you will need to do for this assignment and the Unity forums also provide high-quality answers to common pitfalls and debugging questions.

For this part, you will create a new room at least 50 units away from the first room. Inside the *Assignment 1 Resources.zip*, I have provided you with a package of a wall that contains a door. Your new room will use this object as one of the walls. The floor plan of the room will be a hexagon (meaning there will be six walls), and the ceiling will be slanted (not parallel to the floor). It is fine if the walls pass through each other, provided the final room is fully enclosed and looks good from the inside. Use Unity *Cubes* this time, so that the directional light is blocked. You can make the cubes very thin so that they are like the planes you used before (except, of course, that they are solid on all sides). Add a point light in your room, as we will need to clearly see all of the features of the room.

To import the package, unzip the *Assignment 1 Resources.zip* folder, then go to *Assets* > *Import package* > *Custom package*, navigate to your unzipped folder, and import the *Assignment1_Part1-2.unitypackage* file.

Material (1 Point). Read up on Materials, Shaders, and Textures, focusing mainly on the Materials for now. I have provided you with an image (*tile.png*) and a normal map (the purple-shaded image *tile-normal.png*). Create a material with these images and put it on one wall. Change the tiling and put it on 2 different walls. Finally, change the metallicity and put it on the remaining 2 walls. Make a simple colored material for the ceiling and floor and apply it. Make sure each face is distinct enough that it is clearly visible. If that means you have to make the room look a little odd, then so be it.

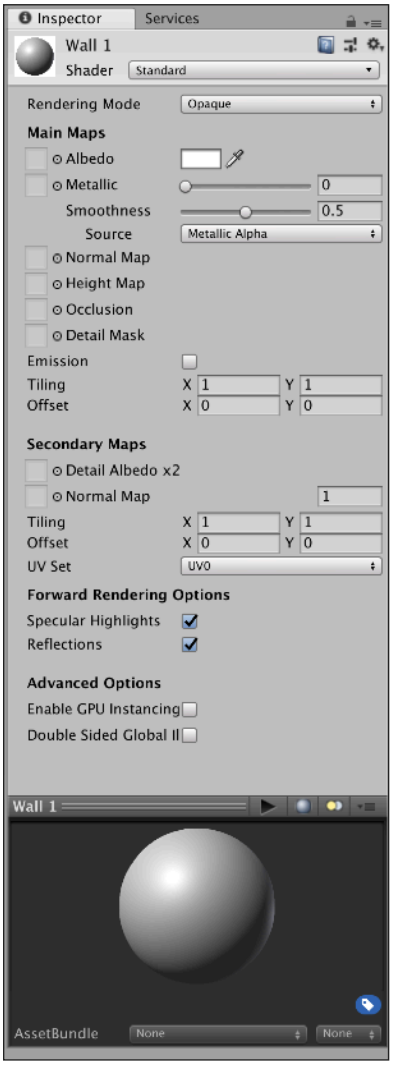


Figure 16. Material properties in Unity's Inspector window.

To create a material, go to *Assets* > *Create* > *Material*. This will generate a default material. Name it *Wall 1.* Select it and you should see the window illustrated in Figure 16.

Drag the *tile.png* image to the box labeled *Albedo*. Drag this material from the *Assets* folder onto one of your walls (except the wall with the door) in the *Scene* window. It will not look good; it will get better.

Drag the *tile-normal.png* image to the box labeled *Normal Map*. This changes the perceived material of the material. A normal map is a trick used to give the illusion of depth on a flat surface, by telling the engine to reflect light as there were bumps and pits in the material.

Create a new material, name it *Wall 2,* apply the *Albedo* and *Normal Map* as with *Wall 1*. Apply it to another two walls (again, except the wall with the door). Right above the *Secondary Maps* subheading is the *Tiling* property, with available values for X and Y. Tiling causes a material to repeat itself on the same object, rather than covering the whole thing. Changing tiling X to 2 means that the material will repeat once (that is, show up twice) in the X direction on the wall. Play with the tiling until you like the look of it. Figure 17 shows an example of non-tiled and tiled walls side by side.

Create a new material, name it *Wall 3* with the same *Albedo* and *Normal Map*. Change its tiling to be different from Walls 1 and 2. Right below the *Albedo* option is a slider for *Metallic* and a slider for *Smoothness*. Play around with these and see how they affect the material. Both deal with how light reflects off the material; *Metallic* gives a more metallic look and *Smoothness* helps enhance or subdue the *Normal Map*. Paste this material onto the remaining two walls.

Finally, create a material called *Floor* with no *Albedo* or *Normal Map*. Next to the *Albedo* option is a small color box. This shows what color the material will reflect. When the material has no *Albedo*, the material will be this flat reflection color. Try and see what happens when you change

the color of a material with an *Albedo*. Apply this flat color onto the floor and ceiling of your room.

```
public GameObject ball;

void OnTriggerEnter(Collider other) {

}
```

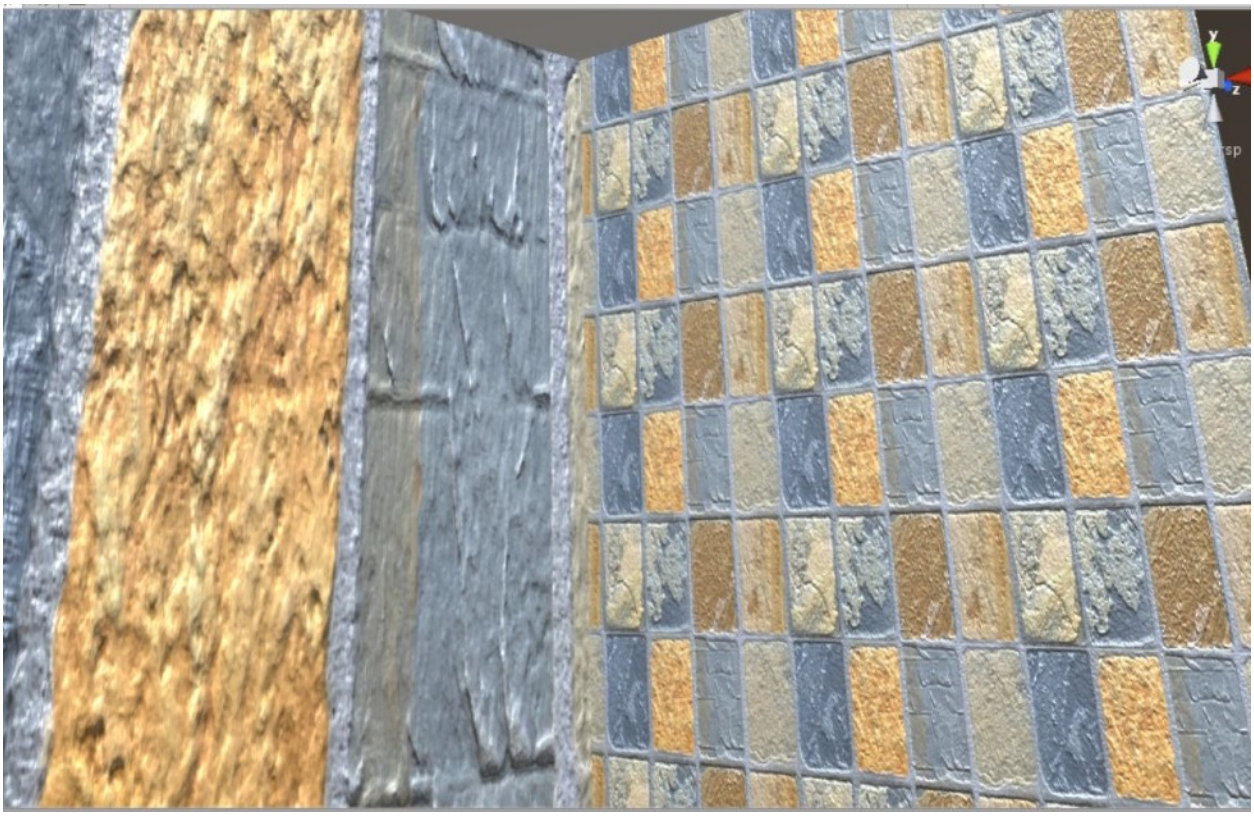


Figure 17. An example of non-tiled and tiled walls side by side.

Scripting (1 Point). You will be creating two scripts for this room:

1. Room Switch: Extend your *RoomSwitch* script from Part 1-1 so that pressing the *1* key moves you to the center of the room created in Part 1-1.

2. Trigger Zone: Create a box collider and make it a trigger. Place a sphere above the trigger zone. Make a script so that when the player enters the trigger zone, the sphere falls.

   To create the Trigger Zone script, watch the Unity Tutorials on Colliders and Triggers. Next, create a new empty GameObject. Select it, and in the *Inspector* window click on the *Add Component* button. Navigate the menu to get to *Physics* > *Box Collider*. A BoxCollider is (as the name implies) a box-shaped area that registers and reacts with collisions with other GameObjects. Make the box collider 2x0.5x2 (x, y, z) units. Select the *Is Trigger* option. Your GameObject should show up in the *Scene* window as a green wireframe box. Place this GameObject in the back of your hexagonal room, across from the door. Create a sphere 3-4 units directly above the center of the trigger GameObject. Add a script to your trigger GameObject by clicking on *Add Component* > *New Script* within the *Inspector* window and name it *BallDropScript*. Open the script and add the lines illustrated in Figure 18.

```
public GameObject ball;

void OnTriggerEnter(Collider other) {

}
```

Figure 18. Adding code to register triggers for the sphere to drop.

The *OnTriggerEnter* function will be called when the collider attached to our empty GameObject is entered. The *other* parameter is the collider that intersected this collider. The public GameObject tag shows a neat feature of Unity. Save your script and then select the empty trigger GameObject in the *Hierarchy* window. The script Component within the *Inspector* window will now show a field for the ball, as illustrated in Figure 19.

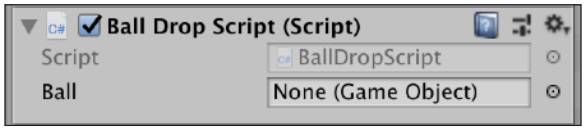


Figure 19. A public property being visible within Unity's Inspector window.

Drag the sphere onto this field. This enables you to reference the ball variable in your script and obtain a direct reference to the sphere you dragged in. You can read more about this feature of Unity in the Variables and the Inspector tutorial.

To make the sphere fall, you'll need to get the rigid body of the sphere (rigid bodies deal with physics and you can read more about it here), using the *ball.GetComponent<RigidBody>()* method. After that, set *rigidBody.useGravity* to *true* to let the physics engine take care of dropping the ball.

Store Assets (1 Point). Import at least one free asset from the Unity Store. Place it in the room. You will need a free Unity account for this. Once you have a Unity account, find the *Asset Store* tab within the Unity editor, search for an asset, and import it into the room. Make sure it doesn't intersect with your collider or it will trigger the collider. It can be whatever you want (within reason).

Part 1 is ready for submission! Follow the submission guidelines in the **SUBMISSION FORMAT** section to do so.

Part 2 — “May the Force Be With You.” (5 Points; 1 Bonus Point)

Create a new Unity project and name it *[class number]_Assignment1_2.* For example, if I am enrolled in the Undergraduate version of this course, the project name would be: *CS5963_Assignment1_2.* Do not work in your Part 1 project.

In this part, you will create a simple game with minimal hand-holding (relative to Part 1).

The Room 2: Electric Boogaloo (0.25 Points). Create a more interesting room with a window! The shape and size is all up to you, however it should be large enough to comfortably accommodate all of the following requirements within it. The walls should be colored or textured. The choice of wall color and texture is up to you (within reason).

Bonus! (1 Point). For extra credit, use a 3D modeling tool to create more complicated room geometry. For example: a curved roof, slanted windows, multiple floors, etc. Note in the submission README what you created. Software options for creation include:

- Blender (https://www.blender.org) — Extremely powerful, but complicated.
- Google Sketchup (https://www.sketchup.com) — Simple, but relatively more limited.

The default Unity modeling tools are fairly limited, so I recommend you familiarize yourself with at one of the above tools; it may greatly assist you in your final project. To get the extra credit, you must do a non-trivial amount of extra work with your modeling tool. That is, it should look like it took you more than 15 minutes to complete.

Skybox (1 Point). I have provided you with six images in the *Assignment 1 Resources.zip* that together form a skybox. You are going to create a skybox with these images and apply it to your scene. I highly encourage you to check out the Unity Manual page for Skyboxes. Skybox images were created by MGSVEVO.

Directional Light (0.25 Points). Create a directional light for the scene and set it to have hard shadows. Set its angle to match the sun in the skybox.

Scripting (3.5 Points). You are going to make a VR game similar to a cat chasing a laser pointer (where you are the cat). In this room, you are going to place several box colliders (at

least 4) and mark them as triggers. Place a point light at the center of each box collider. Every 3 seconds, one of these point lights should light up. The player should then move to the lit up point light and press a button on a VR controller (the *OnTriggerStay* method should be helpful here). When the player does so, they will get one point and another light should light up at random (bypassing the normal 3 second timer). The player's score should be displayed on the wall, in sharp (i.e. not blurry) text. The player should be able to quit at any time upon pressing *another* button on the controller. Be sure to indicate in your project which button does which function.

Note: if this is your first time using the Oculus Runtime, you need to sign up for a free Oculus account and go through several steps including adjusting the sensor and inter-pupilary distance to set up the headset. The runtime will provide instructions, but if you encounter difficulty please ask questions on the Canvas discussion board.

Using a controller in Unity is not quite as simple as using the keyboard. Unfortunately, because you cannot see the keyboard in VR and all of the keys largely feel the same, keyboards do not work well in VR. Controllers, with their contours and designated button shapes, are much easier to use without looking.

Here are some resources to help you with managing an Xbox controller input:

- Unity Manual page on Input
- Unity wiki page on Xbox controllers
- Microsoft Blog page on Xbox controller input in Unity (you'll find the controller drivers here as well)
- Unity Manual page on Time and Frame Management

A very useful method here is Unity's *Time.DeltaTime()* method, which (when called from the *Update* method) will tell you how many real-time seconds have elapsed since the last frame.This is important to separate the game logic from the frame-rate.

Part 2 is ready for submission! Follow the submission guidelines in the **SUBMISSION FORMAT** section to do so.

Part 3 — “I Have You Now.” (5 Points; 2 Bonus Points)

In this part, you will develop a “guard” in the shape of a cube that will mirror your movements in the VR game from Part 2.

Create a new Unity project and name it *[class number]_Assignment1_3.* For example, if I am enrolled in the Undergraduate version of this course, the project name would be: *CS5963_Assignment1_3.* Do not work in your Part 2 project; copy it entirely into this project.

Through development, you will become familiar with manipulating a GameObject’s rotation and position. Below are the concrete tasks:

- **Camera Reset (0.75 Points)**. Create a script named *CameraReset* that accomplishes the following: pressing the *Tab Key* resets the main camera position’s global coordinates to (0, 0, 0). Pressing the *Tab Key* multiples times should move the camera back to the origin each time, regardless of where the player moves their head between presses.
- **Camera Flipper (0.75 Points)**. Create a script named *CameraFlipper* that accomplishes the following: Pressing the *F Key* should flip the user 180 degrees, so that they end up looking behind where they were looking. Pressing the *F Key* multiple times should keep flipping 180 degrees; *CameraReset* should remain functional regardless of orientation.
- **VR Mirror (3.5 Points)**. Create a script named *VRMirror* to modify the position and rotation of the cube face to match or mirror your own.
    - **Toggling (0.5 Points)** Pressing the *M Key* should make a certain cube toggle between mirroring or following the user’s movements; this includes positional and rotational movements:
    - **Matching (1.5 Points):** When matching, the cube should be looking in the same direction the camera is (as a result, the user should be able to see the back of the cube’s head). Further, if you bring your face closer to the screen, the cube moves further away from the screen.

- **Mirroring (1.5 Points):** When mirroring, the cube should be facing the camera (as a result, the user should be able to see the cube's face). Further, if you bring your face closer to the screen, the cube moves closer to the screen as well. Note, the *forward* axis of the mirror-cube is 90 degrees offset from that of the camera; you will need to account for this.

Bonus! (2 Points) The bonus assignment is to figure out a way to disable position and rotation tracking. The aim is to help you understand how much rotation and position tracking contribute towards feeling immersed in VR. Create a script named *ToggleTracking* that turns tracking on and off. Tasks that need to be completed in this part include:

- **Disabling Rotation Tracking (0.5 Points)**. Pressing the *R Key* should toggle rotation tracking on and off.
- **Disabling Positing Tracking (0.5 Point)**. Pressing the *P Key* should toggle position tracking on and off.
- **Both (1 Point)**. When disabling rotation and translation, you can assume that both tasks will not be tested together; in other words, you do not need to disable position and rotation tracking at the same time. You will earn 1 point if you figure out how to disable position and rotation tracking together. Be sure to note in your *README.txt* if you complete the extra credit.

Important Caveat for Part 3: Because of Unity's flexibility, there are many ways you could accomplish Part 3. One such way is to make everything a child of the Camera. This solution inherently depends on the hierarchy structure of the Unity project and is slow in terms of runtime; this solution will not be accepted to satisfy Part 3. The intent is to get you to think about how to reverse/counteract the effects of position and rotation tracking using transformations.

Part 3 is ready for submission! Follow the submission guidelines in the **SUBMISSION FORMAT** section to do so.

SUBMISSION FORMAT —
“You’re All Clear Kid, Now Let’s Blow This Thing and Go Home!”

Now that you have implemented a number of components, it is time to package your code for submission. Follow the following instructions:

1. Create a README.md (Markdown format) File that contains any special instructions or notes you think are relevant for evaluating your assignment. At the bottom of your README, add the following:

   We have all contributed to this code in an equitable manner and agree that, to the best of our knowledge, it accurately represents our understanding for the assignment.

   [Full Name of Partner 1]
   [Full Name of Partner 2]
   [Full Name of Partner 3]
   [Full Name of Partner 4]
   [Full Name of Partner 5] (If applicable)
   [Class Number]

2. Create a *.unitypackage* file:

   A. Save your Unity scene in the Assets folder with the title *[class number]_Assignment1_1* (for Part 1), *[class number]_Assignment1_2* (for Part 2), *[class number]_Assignment1_3* (for Part 3).

   B. Using the editor, find the create scene in the Project menu.

   C. Right-click on the scene and select *Export Package*.

   D. Export the file using default settings (*Include dependencies* should be checked by default).

3. Create a standalone game build:

A. Go to *Edit* > *Project Settings* > *Player*. Make sure the *Virtual Reality Supported* box under *XR Settings* is checked.

B. Go to *File* > *Build Settings*.

C. Click *Add Open Scenes*. This will add the currently open scene to the build. You must have saved the scene to the *Assets* folder for this to work (which you should be doing anyway).

D. Save the project to your local disk.

E. Hit *Build*. This should create an executable for running the build, a folder containing your scene data, and a UnityPlayer.dll. Make sure the executable runs correctly on the Rift before submitting.

4. Copy the Input Manager

   A. Shut down your project and navigate to *Your_Project_Folder* > *ProjectSettings*.

   B. Copy the *InputManager.asset* file and copy it to your submission folder. This will allow us to replicate any new gamepad buttons or joysticks you mapped.

5. Push the files to Github and submit the link through Canvas

   A. Populate your Github repository with the following items:

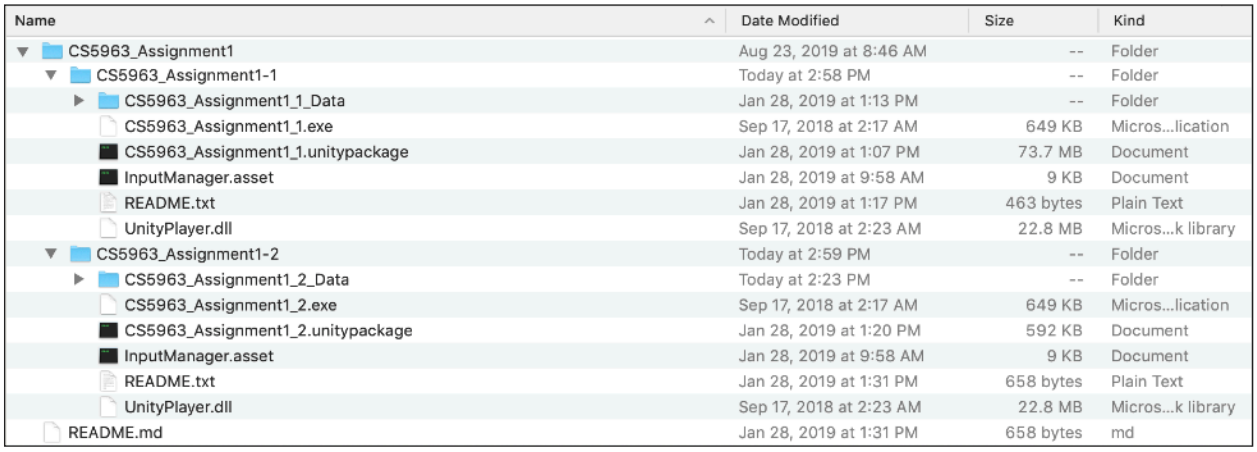
      I. The *README.md* file created in Step 1.

      II. The *.unitypackage* created in Step 2.

      III. The *.exe, dll* and data folder created in Step 3.

      IV. The *InputManager.asset* object found in Step 4.

      Your repository structure should look like Figure 20, below.



Name	Date Modified	Size	Kind
CS5963_Assignment1	Aug 23, 2019 at 8:46 AM	--	Folder
CS5963_Assignment1-1	Today at 2:58 PM	--	Folder
CS5963_Assignment1_1_Data	Jan 28, 2019 at 1:13 PM	--	Folder
CS5963_Assignment1_1.exe	Sep 17, 2018 at 2:17 AM	649 KB	Micros...lication
CS5963_Assignment1_1.unitypackage	Jan 28, 2019 at 1:07 PM	73.7 MB	Document
InputManager.asset	Jan 28, 2019 at 9:58 AM	9 KB	Document
README.txt	Jan 28, 2019 at 1:17 PM	463 bytes	Plain Text
UnityPlayer.dll	Sep 17, 2018 at 2:23 AM	22.8 MB	Micros...k library
CS5963_Assignment1-2	Today at 2:59 PM	--	Folder
CS5963_Assignment1_2_Data	Today at 2:23 PM	--	Folder
CS5963_Assignment1_2.exe	Sep 17, 2018 at 2:17 AM	649 KB	Micros...lication
CS5963_Assignment1_2.unitypackage	Jan 28, 2019 at 1:20 PM	592 KB	Document
InputManager.asset	Jan 28, 2019 at 9:58 AM	9 KB	Document
README.txt	Jan 28, 2019 at 1:31 PM	658 bytes	Plain Text
UnityPlayer.dll	Sep 17, 2018 at 2:23 AM	22.8 MB	Micros...k library
README.md	Jan 28, 2019 at 1:31 PM	658 bytes	md

Figure 20. Expected directory structure for submission. Assignment 1-3 is not pictured.

Note, you may wish to include the project source code as part of your submission. You are welcome to; this might help guarantee that we're able to run your submission, just in case what you provide does not work. As long as the above structure is in place, you may put your source code in the repository wherever you wish. Keep in mind, we will be looking at the timestamps of the Github log as a way to determine what will count toward submission.

6. Have one of the partners submit the link to the repository on Canvas.

For the love of all that is virtual, do not submit all the Unity metadata. If you use the .gitignore file provided in the class resources, you should be safe.