Homework 3
Topic: Unity and VR Basics

Photo by Samuel Zeller on Unsplash

Introduction

In this assignment, you will gain experience in basic Unity3D development, and get to experience some VR demos.

Important Notes:

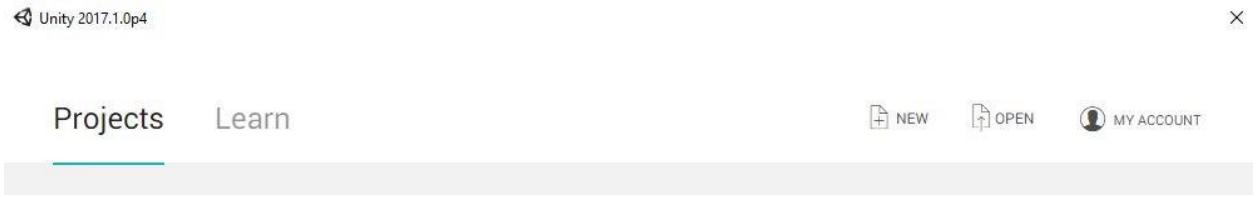
1. Read all submission instructions carefully, and submit all required documents.
2. VR devices need a TA / course representative to be present in the office for testing.
3. ~~Google, DuckDuckGo, Yahoo,~~ Baidu or Bing, the Unity Documentation, the Unity Wiki, and the Unity Forums are excellent sources of information, especially as you start to use more advanced features of Unity.
4. Check Piazza for any bugs, updates, or important information for this homework (hw3).

Steps

HW3.1

Download and install Unity (https://unity3d.com/get-unity/download/archive, the recent versions 2017.x and 2018.x are accepted). Create a new Unity project named: ***<StudentID>*_HW3_1**

Create a new 3D project by clicking through the prompts when you click the NEW button on the top right. Fill in the appropriate details to your heart's content.



Next, read through the following Unity Tutorial, to familiarize yourself with the very basics of the Unity interface-

Basic Interface Tutorial

More In Depth Interface Tutorial

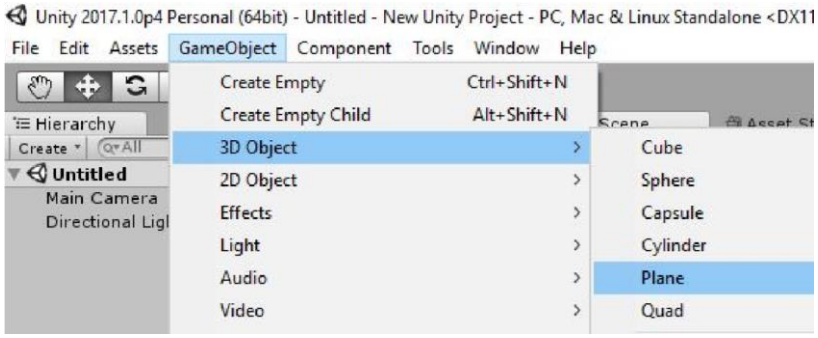
Feel free to look through some more tutorials, to familiarize yourself with the basics of Unity.

3.1.1

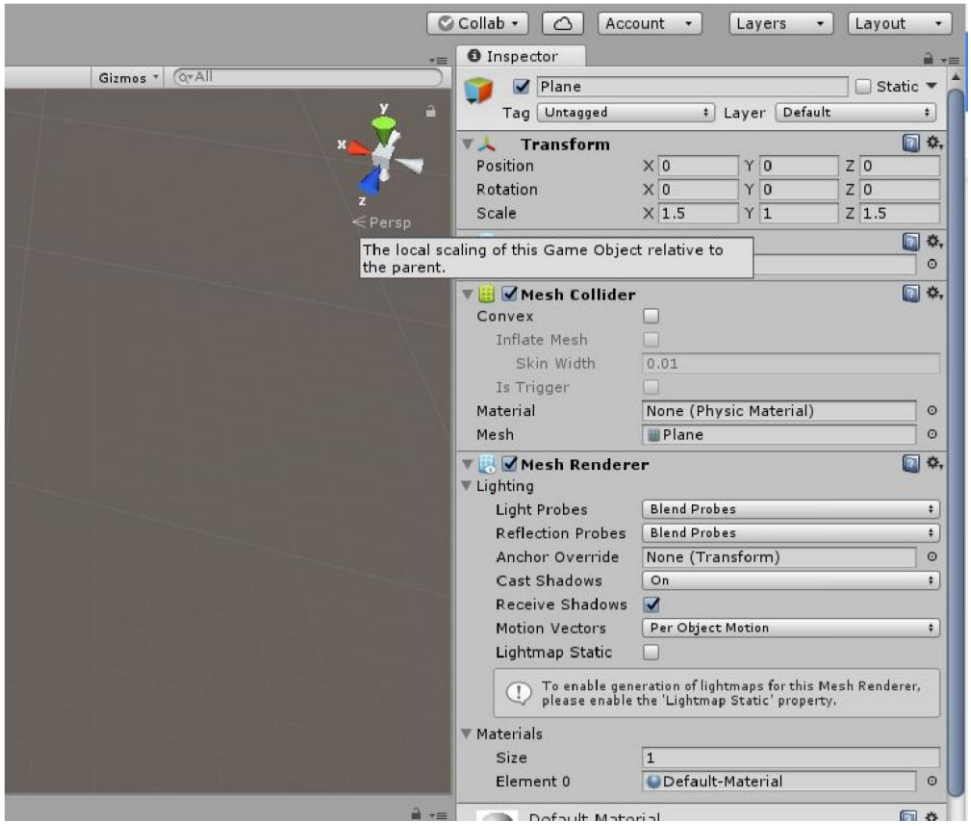
The Room: *Here VR!*

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 (aka meters).

First, create a plane. It can be found in the top bar menus, under GameObject → 3D Object- > Plane, as shown below.



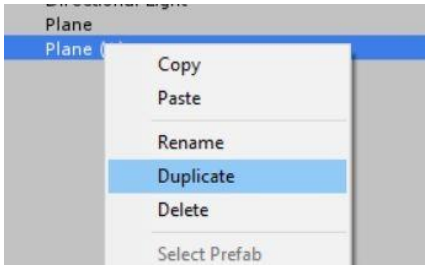
Now, by default the plane is 10x10 (X x Z) units. In order to make your room 15 units wide, you have to scale the plane. On the right side (in the default editor layout) you will find the Inspector window. This window provides details about the currently selected object. Select the plane in the Scene view, and the Inspector will fill with information and settings for said plane. Find the “Scale” option, and set it to 1.5 on the X and Z to make your plane 15 units wide and long.



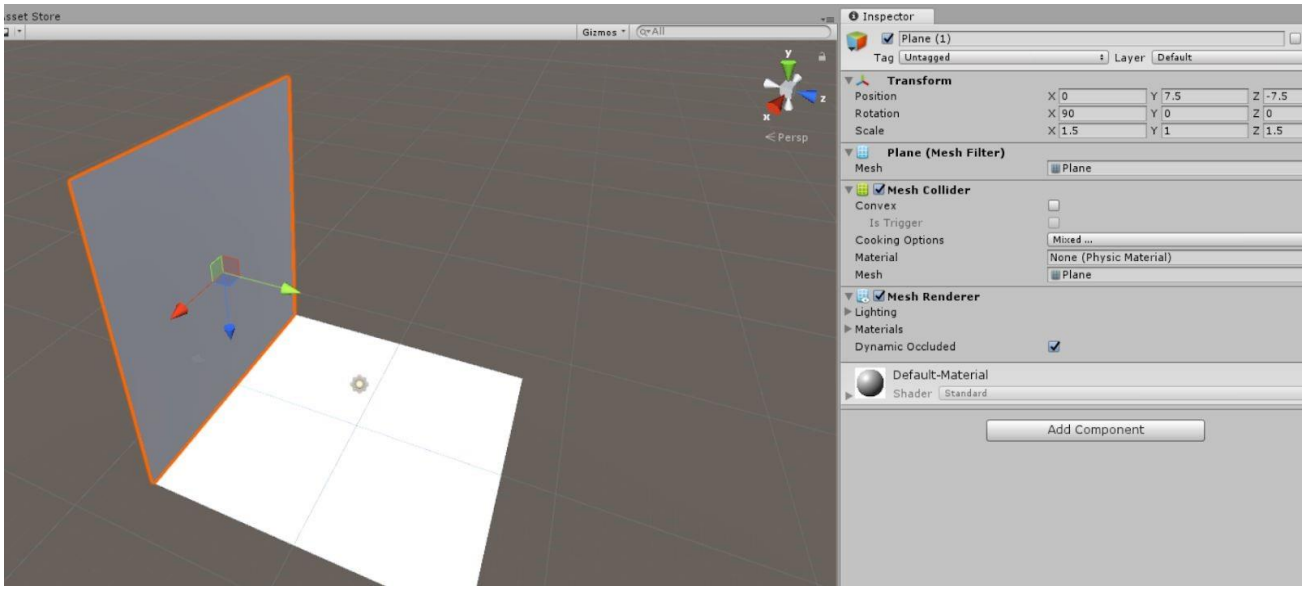
Note: The plane has no thickness, so the value in Y can be any positive integer.

By default, your scene has a directional light in it. This is basically the sun- a light source that illuminates your entire scene from a specified angle, from very far away. You'll notice that your planes do not block this light. That's because planes only block light (and render) from one side. Bear this in mind when creating objects in Unity in the future! Please delete the directional light (don't worry, you'll add new lights in later).

In the hierarchy view, you can select your plane, and duplicate it (or Ctrl + D).



From there, simply 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 object, so if you want your walls to match up with the floor (at height 0), your walls will need to be at 7.5

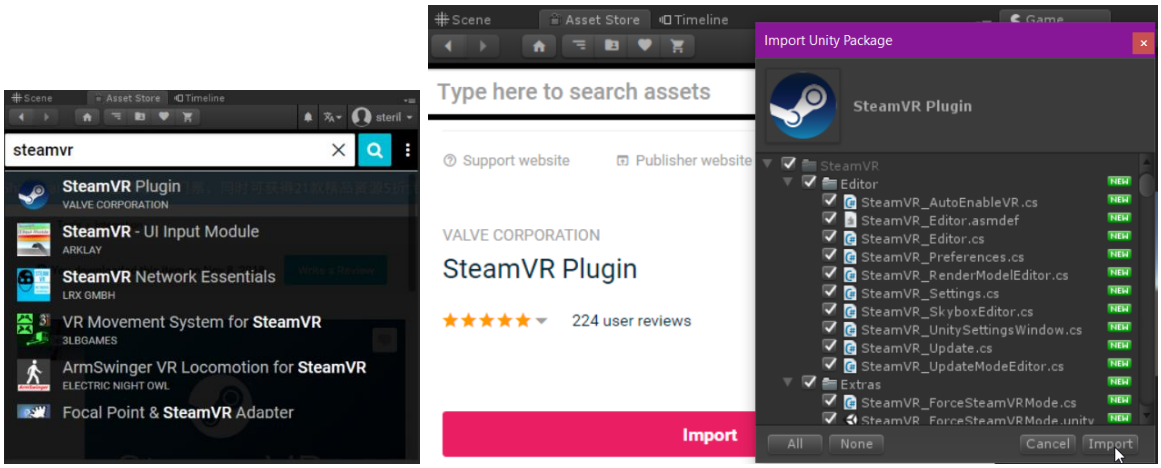


Note: In the image above, rotating the plane also rotated its axes (the blue z axis now points down). Make sure to account for that when rotating and moving objects!

Player:

Place a SteamVR prefab in the room. This prefab handles basic movement, collision and camera control.

First of all, you need to install Steam, login and download SteamVR app from Steam store (for free). Later, you can import SteamVR unity plugin by going to AssetStore→Search("steamvr")→Download/Update, and then click import.



If, for whatever reason, you can't find it, go to the SteamVR github and download their Unity package. Unzip the unity package file, and then go to Assets→Import Package→Custom Package. Find the unzipped .unitypackage, and import all of the items within. You should now have an "SteamVR" folder in your "Assets" folder.

In this class, you will be primarily a prefab object from this package (found in SteamVR→InteractionSystem→Core→Prefab): **Player**. The Player prefab will contain a camera for the VR headset and a pair of hands we can use to interact with items in our scene. The advantage of using the 2D debug mode embedded in Player is that it allows you to easily debug your game without having to put on your VR headset all the time (or if you don't have a VR device at the time being).

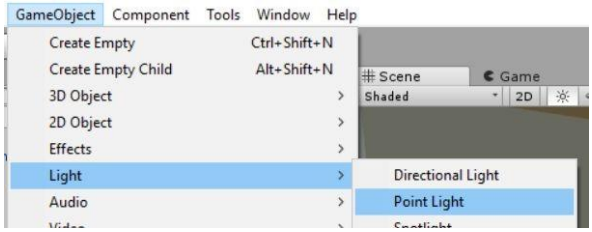
Place a Player prefab into your scene (drag & drop into the scene), at (0,1,0), and delete the "main camera" object. There's just one more thing you need to do to finish enabling VR. Go to Edit→Project Settings→Player, which will open up the player settings in the inspector window. In "Other Settings" (or XR settings, depending on your version of Unity), make sure that the "virtual reality supported" option is checked. Now, when you play your scene, it should render inside the VR device (HTC Vive or Oculus Rift) that you test in.

If you have issues importing, please check piazza to see if others have had your issue, and make a post if you need more information.

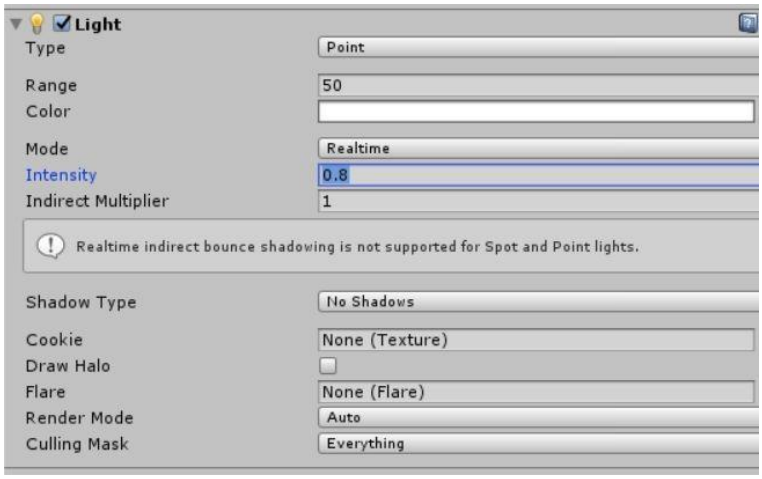
Lighting:

At the center of the roof of the room, place a point source of light. This light will change color by pressing the Tab key, which is detailed in the scripting section.

First, read up on the types of light in Unity. Then, go to GameObject→Light→Point Light, and bring a point light into your scene. Place it at (0,15,0)



Select your light, and your inspector view should have a “Light” component like this-



Of primary importance are the “range” (the radius of your light), color, and intensity values. Set the shadow type to “soft shadows”, and read up on Unity Shadows. Set the “Mode” to “Realtime”, and read up on Lighting Modes In Unity Set your range and intensity so that your room is brightly lit.

Planet and Moon:

Create a large sphere, and have it float in the middle of the room. Create another, smaller sphere, set it as a child object of the bigger sphere, and move it next to the bigger sphere, 4 units away on the X-axis. You will make it orbit the larger sphere in the Scripting section.

Create two spheres (GameObject→3D→Sphere). Scale the first sphere to 2 in all directions, and place it in the center of your room. In the Hierarchy view, drag the second sphere onto the first. They should now look like this-

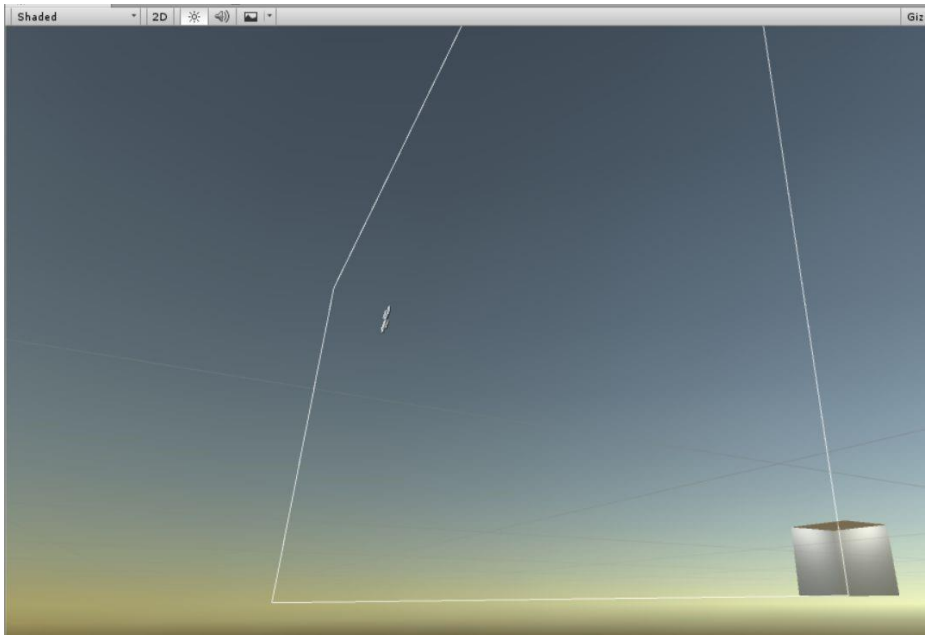
What this means is that the second sphere is a child of the first sphere. So now, whenever you change the position, rotation, or size of the parent sphere, its child will make the same movement, rotation, or scaling. Furthermore, the 0,0,0 position of the child is now its parent’s position, **NOT** the global 0,0,0. That is, 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 (this will make more sense later). For more information on parent-

child relationships, see the Hierarchy page of the Unity Manual. Set the position of the child sphere to be (2,0,0), which is four units from the parent sphere on the X-axis. Why? (*ƃuiɹɐɔs ɟo əsnɐɔəq*)

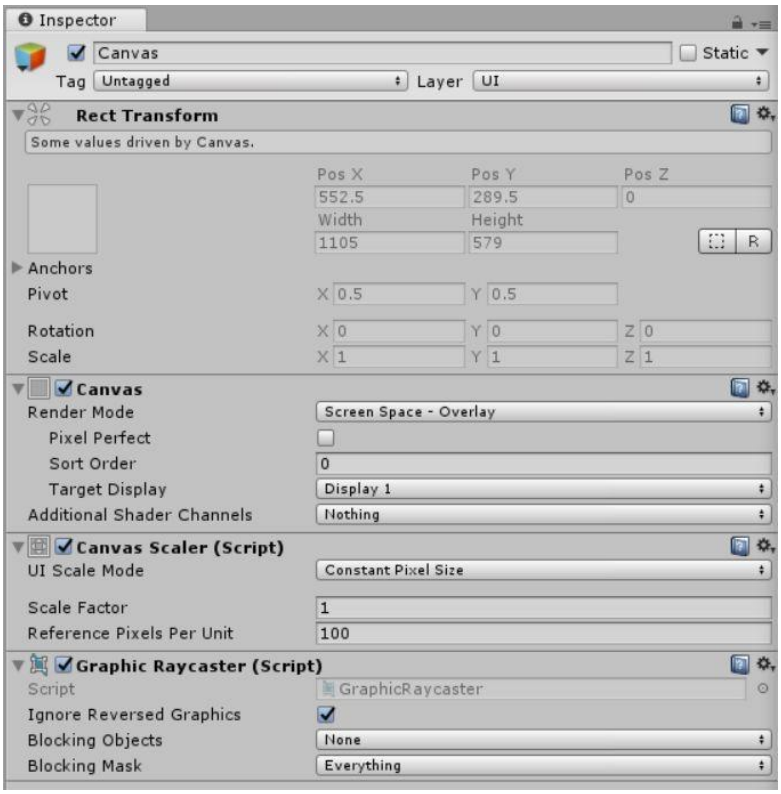
Text:

Put large text on a wall, detailing the controls and listing your names and student IDs. Feel free to experiment with what you can put on a canvas, but keep it appropriate!

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, and some text on that canvas as a child of the canvas. It may also happen to be massive. Not what we want.

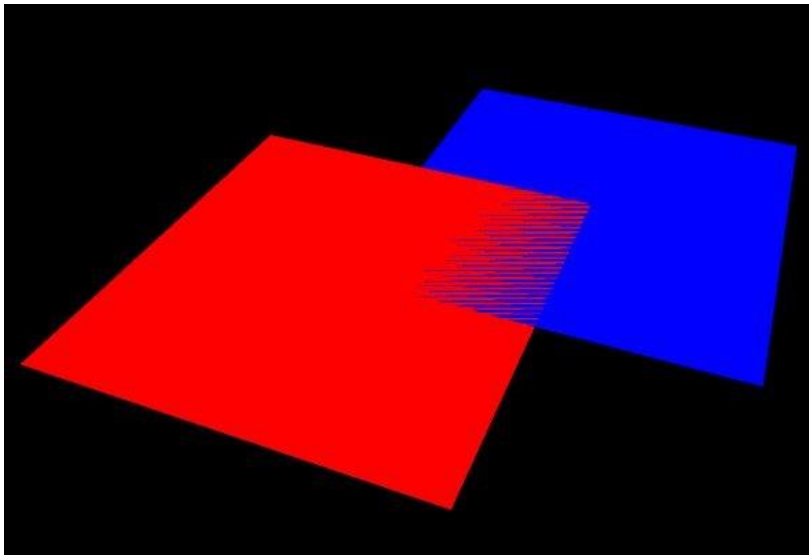


To remedy this, select the canvas (not the text) from the hierarchy view. It's inspector should look like this - *(right)*



The first thing to do is change the Render mode from "Screen Space - Overlay", to "World Space". This changes our canvas from a UI element that is glued to the camera, to an object that is stationary in the world. Traditional UIs do not work well in VR, and we highly, highly advise against sticking any UI elements to the camera in your future HWs and Projects. Always attach UI elements to something in the world (See this).

Now that the canvas is a world space object, we can make it a more reasonable size. However, since the Rect Transform's width and height determine the resolution of our text canvas, we cannot set them to be, say, 5x5, because then our text resolution would be 5 pixels by 5 pixels. Set the width and height to 1000 (that is 1000x1000 pixels). Shrink the canvas by setting the scale. We want our canvas to be 10 units by 10 units, and be 1000x1000 pixels, so our scaling is 10/1000 = 0.01. Make sure your text'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. You want to place your text ever so slightly (like, 0.001) in front of the wall it is against to avoid Z-fighting (which is where two objects have the same depth, and Unity can't figure out which one to render). Below is an example of Z-fighting-



Now, you can set your text color, size, font, width, whether it wraps or overflows, etc. Make your text have your student ID, name, as well as the controls for your game. Make sure it is big enough for us to read. 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:

You will need to write a couple of scripts for this part of the HW. Read up on Scripts in Unity, and familiarize yourself with C# syntax. It should be very familiar to any of you who have worked with Java. If you are unfamiliar with programming, you can check out this C# tutorial. You'll only need the basics of objects, classes, and variables for now. Throughout this course, you will find the Script API Reference a useful source of information.

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. You can attach a script by selecting the light, then dragging the script from the "assets" tab to the inspector tab on the right.

When a Unity Script is attached to a GameObject, that script will run when the game is started.

Furthermore, the "this" reference in the script will refer to the object that 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, then add these lines to your script-

```
public Light light;
// Use this for initialization
void Start () {
    light = GetComponent<Light>();
}
```

This will get the light component of the object this script is attached to (calling

"GetComponent<>()" is the same as calling "this.GetComponent<>()"), and set it to the "light " variable when we boot up the game. To register input, we use the Input library of Unity, specifically, the Input.GetKeyDown method. This will return true when the specified key is first pressed down. Since we want to listen for the "tab" key, then in our "Update" function, we write-

```
if (Input.GetKeyDown("tab"))
{
    light.color = //insert color here
```

Setting the light color is easy, you can either create a new color using the "new Color(red,green,blue)" constructor, or one of the predefined colors. How you change the light is up to you, but the light should visibly change every time we press tab. Maybe you use a boolean variable to track the current color, or an integer (for more than two colors). That is up to you.

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.*

GameObjects' rotation and position is controlled by their transform parameter, accessed with "<GameObjectName>.transform". This class is well worth looking through, though the most important parts for this class are the "transform.position", which is the 3 vector of the object's x,y, and z coordinates in the global frame (as opposed to the local frame, which is relative to this object's parent's position), and the "Rotate" method.

Most of Unity rotations are done using something called quaternions, which are better than the standard way of measuring rotation (the rotation about the x,y,and z axis). You will learn about quaternions, and why they are awesome, in class later. For now, simply know that the transform.Rotate(Vector3(a,b,c)) will rotate you “a” degrees about the object’s x axis, “b” degrees about the y, and “c” degrees about the z.

Create a script, called “orbit”, and attach it to the parent sphere. In its update method, add this line-

```
this.transform.Rotate(new Vector3(0, 2, 0));
```

This will rotate the parent sphere by 2 degrees about the y axis every frame.

*3. **Room switch:** Pressing the ‘2’ key should switch to Part 2!*

Create the “room switch” script, and attach it to the player. Simply use Input.GetKeyDown(“2”), and set the player’s transform.position to the Vector3 corresponding to the center of your room for HW 3.1.2 (wherever you end up putting it). Don’t forget, the Player prefab needs to be 1 unit above the ground.

*4. **Quit key:** Pressing ‘Esc’ should exit the game.*

This can be simply added on to the “room switch” script. You will want to add the following lines to the update method-

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

Application.Quit() quits a Unity application, but it will not stop a game running in editor. So, we check if we are in editor, and stop the editor if we are.

3.1.2

In 3.1.2, you will be working in the same scene as 3.1.1, **but with fewer instructions.**

You are expected to Baidu the specifics -- Unity has a great tutorial on practically everything you will need to do for this HW, and the Unity forums also provide high-quality answers for debugging advice.

The Room 2: *Here V Go!*

Create a new room, at least 50 units away from the first room. Inside the HW3 zip file, we've 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 ok if the walls pass through each other (or through the floor), 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, being 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.

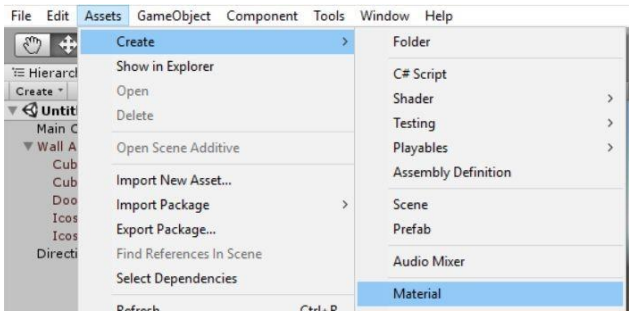
To import the package, unzip the HW3 zip folder,then go to "Assets→Import package→Custom package", navigate to your unzipped HW3 folder, and import the .unitypackage file.

Note- The door object does not currently have a collider, so you can walk right through it. You can add a collider by clicking "add component" in the inspector window, then going to physics-> box collider (or mesh collider).

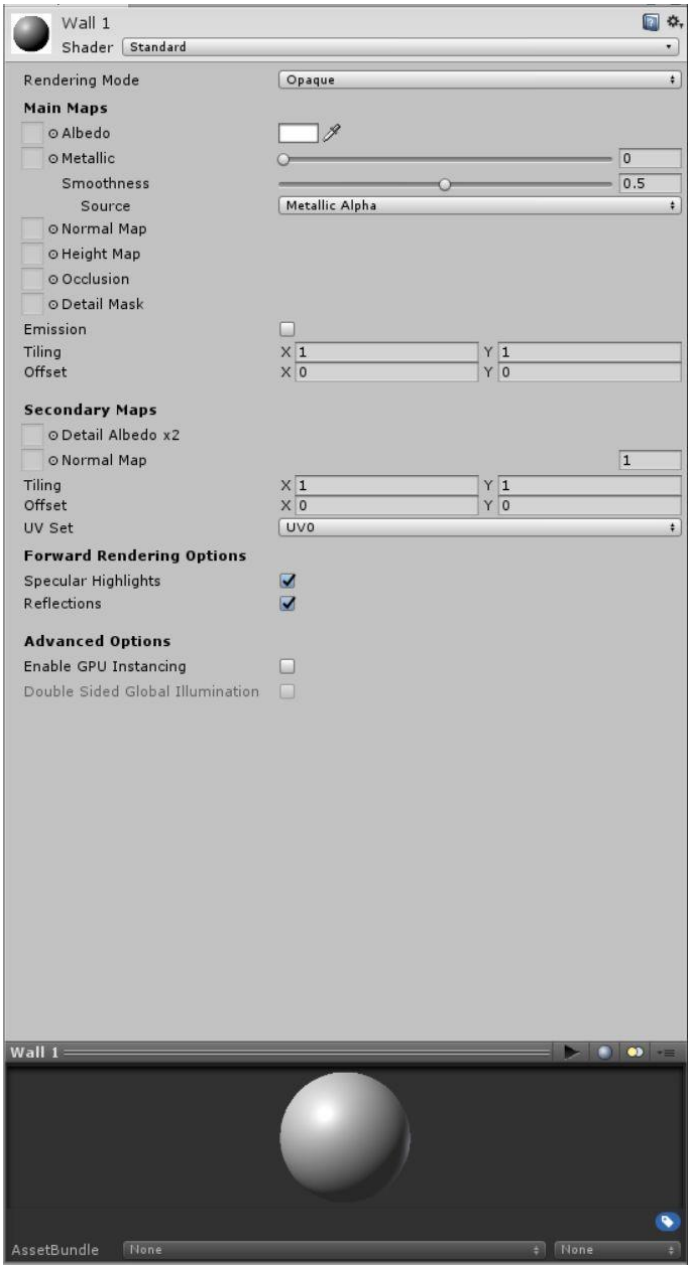
Material:

Read up on Materials, Shaders and Textures, focusing mainly on the Materials, for now. We have provided you with an image (tile.png), and a normal map (it's the weird purplish 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 to the grader. If that means you have to make the room look a little bit ridiculous, then go for it.

To create a material, go to Assets→Create→Material.



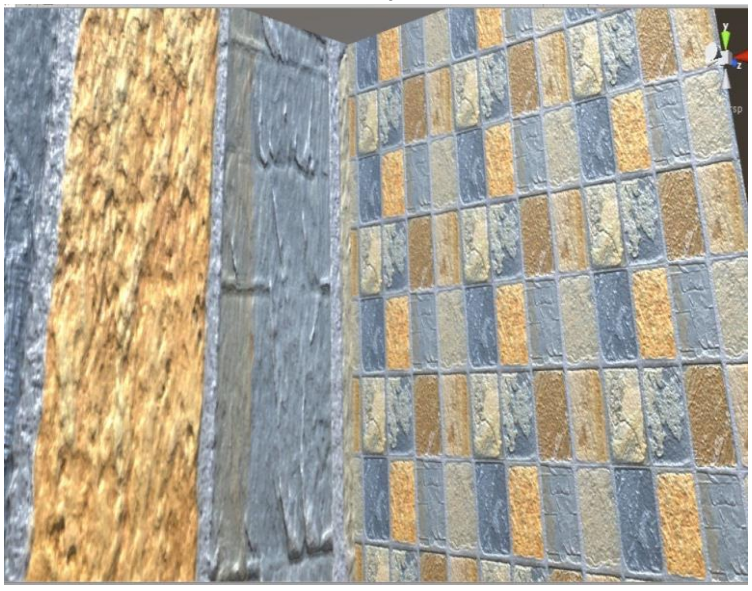
This will generate a default material. Name it “Wall 1”. Select it, and you should see the following menu - *(right)*



Drag the “ `tile.png` ” image to the box labelled “Albedo”. Now, drag this material from the assets folder onto one of your walls (except the wall with the door) in the “scene” view. It probably doesn’t look too good. Don’t worry, it’ll get better.

Drag the “ `tile-normal.png` ” image to the box labelled “normal map”. Notice how it changes the perceived material of 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 if there were these little bumps and pits in the material.

Create a new material, called “Wall 2”, and apply the albedo and normal maps the same as wall 1. Apply it to another two walls (again, except the wall with the door). Right above the “secondary maps” subheading is the “tiling” option, which has an option for x and for y. Tiling causes a material to repeat itself on the same object, rather than covering the whole thing. So, 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. Below is an example of non tiled and tiled walls side by side-



Create a new material, called “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 giving a more metallic look, and smoothness helping to enhance or subdue the normal map. Paste this material on the remaining two walls.

Finally, create a material, called “floor”, that has 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.

Scripting:

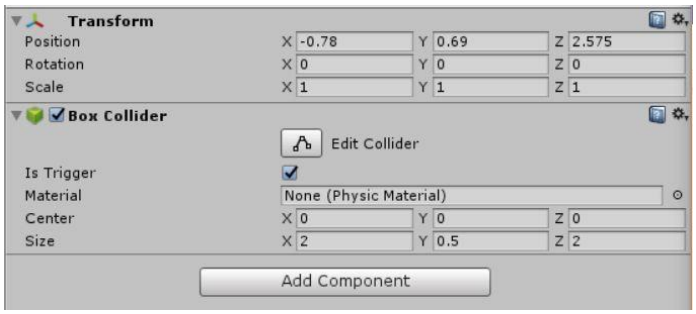
You will be creating some scripts for this room as well.

1. ***Room Switch:*** *Extend your room switch script so that pressing 1 moves you back to 3.1.1.*

This is essentially the same as the movement script from HW 3.1.1. Extend the same “room switch” script again, and make it return you to the center of your first room.

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 ball falls.*

To create the Trigger Zone script, first, watch the Unity Tutorials on Colliders and Triggers. Next, create a new empty GameObject. Next, hit Add Component→Physics→Box Collider. A Box Collider is (as the name would suggest) a box-shaped area that registers and reacts with collisions with other GameObjects. Make the box collider 2 x 0.5 x 2 (x,y,z) units. Select the “is trigger” option. Your object should look like this-



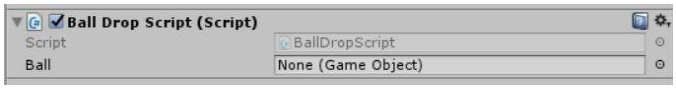
And should show up in the scene view as a green wireframe box. Place this game object in the back of your hexagonal room, across from the door, and create a sphere about 3-4 units directly above the center of the trigger GameObject.

Add a script to your trigger object by clicking Add Component→New Script, and name it “BallDropScript”. Open the script, and create the following lines-

```
public GameObject ball;
void OnTriggerEnter(Collider other) {}
```

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, then navigate to your empty trigger GameObject. The script component should look like this-



So our public GameObject is now a field for the script component in Unity. Drag the sphere into this field. Now, whenever you reference the “ball” variable in your script, it will be referencing the sphere you dragged in. Pretty neat! You can read more about this in the Variables and the Inspector tutorial. Now, all that’s left is to make that sphere fall. You’ll need to get the rigid body of the sphere (rigid bodies deal with physics, read more here), using the ball.GetComponent<RigidBody>() method. After that, simply set rigidBody.useGravity to true .

Store assets:

Import at least one free asset from the Unity Store. Place it in the room. You will need a free Unity account for this.

Create a Unity account, then head over to the “Asset Store” tab, right next to the “Scene” and “Game” tabs-

Sign into your Unity account using the “log in” button at the upper right. Now, you can search for any free asset you desire, and put it in your room. Make sure it doesn’t intersect with your collider, or it will trigger the collider. It can be whatever you want (provided it’s school-appropriate, of course).

Submit:

Submit your unity project ***according to the submission guidelines at the bottom of this assignment.***

HW3.2:

Create a new Unity project, called **<*StudentID*>_HW3_2**. **Do not work in your HW 3.1 project.**

In this part, you will create a simple game with minimal hand-holding, as compared to HW 3.1.

3.2.1

The Room 3: *V for Virtual*

Create a more interesting room, with a window! The shape and size is all up to you, it should be large enough to comfortably accommodate all of the following requirements within it. The walls should be colored or textured, as well. The choice of wall color and texture is up to you (but keep it appropriate).

Skybox:

We have provided you with six skybox images in HW3.zip that together, form a skybox. You are going to create a skybox with these images, and apply it to your scene. Here is the Unity manual page for skyboxes. Skybox asset credit: mgsvevo

Directional Light:

Create a directional light for the scene, set it to have hard shadows. Set its angle to match the sun in the skybox.

Scripting: Trigger game:

You are going to make a game similar to a cat chasing a laser pointer (where you're 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 the "**L**" key on the keyboard (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 (NOT blurry) text. We should be able to quit the game at any time upon pressing the "**Esc**" button on the keyboard.

Using a controller in Unity is not quite as simple as using the keyboard. Unfortunately, because you can't 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 blind.

(Optional: You can add OpenVR commands instead of the keyboard commands if you'd like to test your program on a VR device, visit the TA's office for access to VR devices).

Unity Manual page on Input

Unity Manual page on OpenVR controls

Unity wiki page on Xbox controllers

Unity Manual page on Time and Frame Management

A very useful method here is Unity's Time.DeltaTime() method. This method, when called from the update method, will tell you how many real-time seconds have elapsed since the last frame. This is hugely important, as you do not want to tie game logic to your framerate.

Submission Instructions

Step 1: Create a .unitypackage file

1. Save your Unity scene in the Assets folder with the title "`<StudentID>_HW3_1`" (for HW 3.1) or "`<StudentID>_HW3_2`" (for HW 3.2)
2. Using the editor, find the created scene in the Project menu
3. Right click on the scene and select "Export Package…"
4. Export the file using default settings ("Include dependencies" should be checked by default)

Step 2: Create a standalone game build

1. Go to Edit → Project Settings→ Player. Make sure the "Virtual Reality Supported" box under Other Settings or XR Settings is checked.
2. Go to File → Build Settings
3. 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 (you should do that anyways).
4. Save the project.
5. Hit "Build".
6. This should create an executable (`.exe`) for running the build, a folder containing your scene data, and a UnityPlayer.dll. Make sure this executable runs correctly before submitting.

Step 3: Copy the Input Manager (Optional)

1. Shut down your project, and navigate to Your_Project_Folder→ProjectSettings
2. 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 **(IF you did)**.

Step 4: Zip these files and upload them to your SJTU jBox account

1. Create zip files containing the following items:
    a. The `.unitypackage` created in Step 1
    b. The `.exe, .dll,` **AND DATA FOLDER** created in Step 2
    c. (Optional) The `InputManager.asset` object found in Step 3
    d. A `README.txt` file containing any special instructions or notes you think are relevant for evaluating your assignment.
2. Name the files - <StudentID>_HW3_#.zip.
    a. <StudentID>_HW3_1.zip for HW3.1 and <StudentID>_HW3_2.zip for HW3.2.
    b. (Optional) If you want, you can zip these two zip files again so the final file name can become <StudentID>_HW3.zip.
3. Share the jBox file link via Piazza, just like previous homeworks. (The difference is that you don't add an attachment to Piazza, just paste the link to your zip file on jBox)

DO NOT SUBMIT YOUR ENTIRE PROJECT FOLDER