UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

MULTI-SCALE MECHANICAL TESTING FOR ENGINEERING EDUCATION IN VIRTUAL REALITY ENVIRONMENT

A THESIS

SUBMITTED TO THE GRADUATE FACILITY

In partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

GEORGE HUANG
Norman, Oklahoma
2022

MULTI-SCALE MECHANICAL TESTING FOR ENGINEERING EDUCATION IN VIRTUAL REALITY ENVIRONMENT

A THESIS APPROVED FOR THE SCHOOL OF AEROSPACE AND MECHANICAL ENGINEERING

BY THE COMMITTEE CONSISTING OF

Dr. Yingtao Liu, Chair

Dr. Zahed Siddique, Co-Chair

Dr. Chung-Hao Lee

Acknowledgements

I would first like to thank Dr. Yingtao Liu from the University of Oklahoma for providing me the opportunity to pursue my master's in mechanical engineering. His support and guidance throughout the entirety of my graduate experience has given me invaluable experiences and taught me skills that I would have never been learned had it not been for this project. I greatly appreciate Dr. Liu's advice and mentorship throughout the duration of my project and thesis. I would also like to thank Dr. Zahed Siddique and Dr. Chung-Hao Lee for providing guidance for my project. I greatly appreciate Dr. Liu, Dr. Siddique, and Dr. Lee's membership in my thesis committee. I also appreciate the support from the National Science Foundation through the project entitled: Collaborative Research: Multi-Scale Experimental Mechanics Education Module**.** The award number is 1712178.

I would also like to thank my fellow graduate students and lab members of Dr. Liu's lab including, Ryan Cowdrey, Blake Herren, Chris Billings, and Colin Bray for their support and advice throughout my graduate school experience. I would like to thank my work supervisors David Mays, Blake Yort, and Jonathan Harkness for their flexibility and understanding of work-school balance and supporting my pursuit in higher education. I am very grateful for the online resources that have taught me so much about virtual reality and Unity. I would especially like to thank the YouTube channels Valem and Code Monkey for their vast knowledge and guidance throughout this project. I also greatly appreciate the help of Melody Moody, Logan Roys, Chris Billings, Lane Taylor, and Steven Zhao for testing my program and providing constructive criticism towards improving the project.

I would finally like to thank all the people whom I love. I would like to thank my mother, little brother Iain, little sister Hailey, and baby sister Summer as well as all my other family members who wished me luck in my pursuit of a master's degree. I would like to thank my wonderful girlfriend for her support and the endless happiness she provides. I would also like to thank my lifelong friends Melody Moody, Anne Marie Nguyen, Marie Beausoleil, Rachel Selby, Josh Lee, Cody Rokitka, Dan Parades, and Kyle Lee, as well as my newfound college friend Logan Roys. Their friendship and support in my life have greatly impacted me and my aspirations as well as motivations in life and career.

Dedication

I would like to dedicate this thesis to every mentor that I have had including my teachers, professors, and especially my mother who has guided me throughout my life.

Table of Contents

List of Figures

Abstract

The year 2020 changed the state of the world with the introduction of the COVID pandemic. With various restrictions placed to protect the health and safety of humanity, life was greatly altered for everyone. As a student pursuing a higher education, the closure of schools and labs greatly restricted the practical experiences normally experienced in university due to the lockdown placed upon the populous. This lost experience inspired the ideation of educational labs conducted through virtual reality. With the utilization of virtual reality as a tool to educate students from a remote location, students could still experience the full practical labs that have been physically restricted to them.

The progression of this Thesis research included the initial previous project revolving around the creation of a virtual dog bone sample tension test as well as the continuation of the virtual reality (VR) lab possibilities through the creation of a user interface (UI) based scanning electron microscope (SEM) sample compression test. User testing of the SEM lab resulted in successful runs through the program and constructive feedback on how the program could be improved as well as possible potentials for the program to grow and potentially be utilized as a common educational tool for remote use.

CHAPTER 1: INTRODUCTION

Section 1.1: Inspiration

Amidst the global pandemic labeled COVID-19 that engulfed the entire world in 2020, life was changed in many aspects. The education system experienced a complete transformation due to the rapidly spreading virus and the safety regulations placed to protect the people. Students were locked in their homes and forbidden from commuting to school. Teachers were barred from giving lectures in the standard teaching environment. Both parties had to adapt to the new and unique situation with great difficulty and perseverance to continue the education and growth of the people of the future.

A major component that allowed for this continuation to happen was the availability of technology in the modern world. Programs, such as Zoom and Canvas, allowed for teachers to give live lectures as well as upload recordings and files regardless of the situations that occurred. The flexibility of the education system was key to keeping schooling functional and effective. However, this was not without its downfalls.

In universities, many STEM majors involve labs for both teaching and research. With the pandemic being unpredictable and highly dangerous, many buildings were shut down and were barred of all entry, labs included. This caused great delays and difficulty in the continuation of hands-on education. There were, of course, some solutions to this particular issue such as a recorded lab made by the professor before-hand and uploaded online for the students to see. There were also livestreamed labs and the resulting data given to students to analyze. Although these were great solutions to the remote learning, a

more “hands-on” approach for students to experience live applications of what they have learned in their studies through the use of Virtual Reality is proposed.

Section 1.2: Virtual Labs

Virtual reality (VR) allows for the user to simulate an event or situation using a computer and program. By donning VR headset and gloves (controllers could also be used), a student would be able to conduct a virtual experiment without the danger of close contact with other people or even potential dangers that could arise from the experiment themselves. With the proper coding of the simulation, many aspects of a live lab would be able to be recreated for the student. These actions include, but were not limited to, putting on the required safety gear for the lab, obtaining certain equipment from drawers and cabinets, interacting with objects with realistic physics such as picking up objects and pouring liquids, reactions, measuring, machine interaction, data collecting, and cleanup. A great benefit to labs via VR would be the possibility of including mini lectures or careful explanation of lab steps and the reasoning behind every action as the lab progresses. Quizzes over what was learned could also be incorporated in between each mini lecture.

There were some issues with this idea though. The coding for the VR simulation could be quite complex and take time to create. Each lab would need to be individually created which would make it difficult depending on the teaching style of the teachers differing. Another issue was the availability of the VR equipment not being standard in students’ homes. The labs could also run with difficulty depending on the specifications of the computer running the program.

A solution to this issue could be the creation of isolated and individual rooms equipped with a computer that could run the program and the equipment set up. By creating a strict schedule, students could remain isolated and use the equipment with time in between for sanitation to be conducted. There was also the possibility of the university buying and renting out laptops and equipment to students that were in the lab course. The coding of the simulation could be a bit of an issue if it were being conducted from the ground up but if there were a standard general program that could either be easily altered or rearranged to suit unique labs, then the creation of the simulations should decrease in difficulty. The creation of said general simulation was conducted and refined for this thesis.

Section 1.3: Initial Research

The project this thesis was based upon stemmed from work created by a student from the University of Oklahoma named Aaron Craig. The virtual lab that Aaron created was a dog bone sample tension test. This program was created using the program Unity and represented the initial possibilities of a virtual lab. When ran, the user would initially be in an enclosed space with a table with static sample dog-bones, a tension test machine, and a screen.

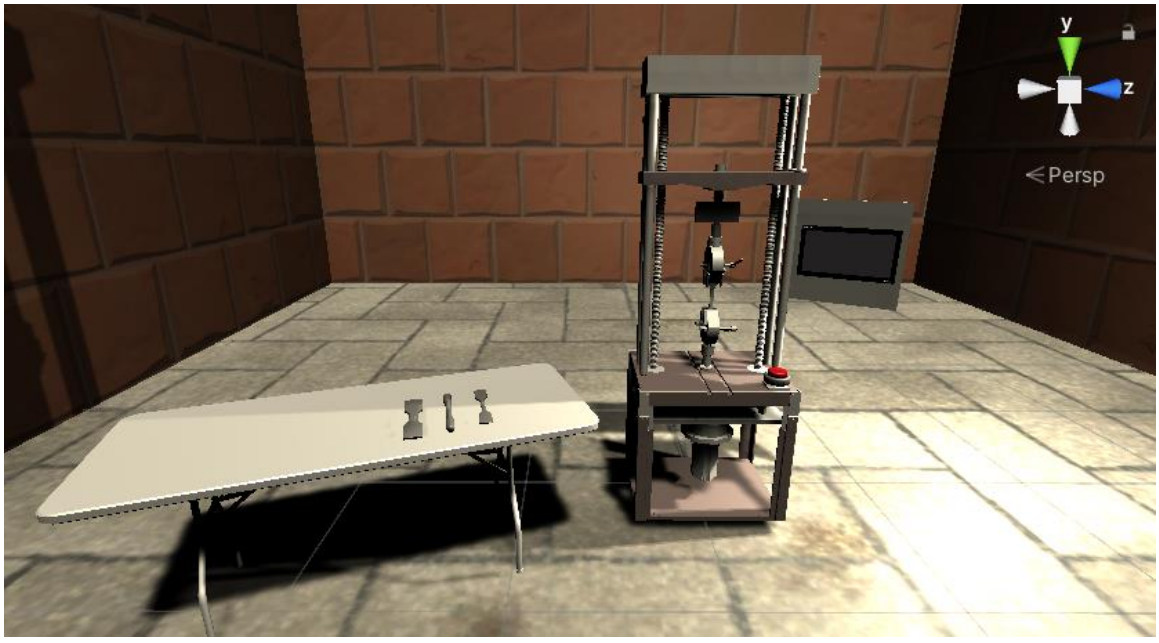

Figure 1: Previous Work – Virtual World

With the start of the program, the machine would slowly apply a tensile force on the dog bone sample while the graph would map out a predetermined set of points to represent stress vs strain. The animation ends with the fracture of the dog bone indicated both on the sample and the graph.

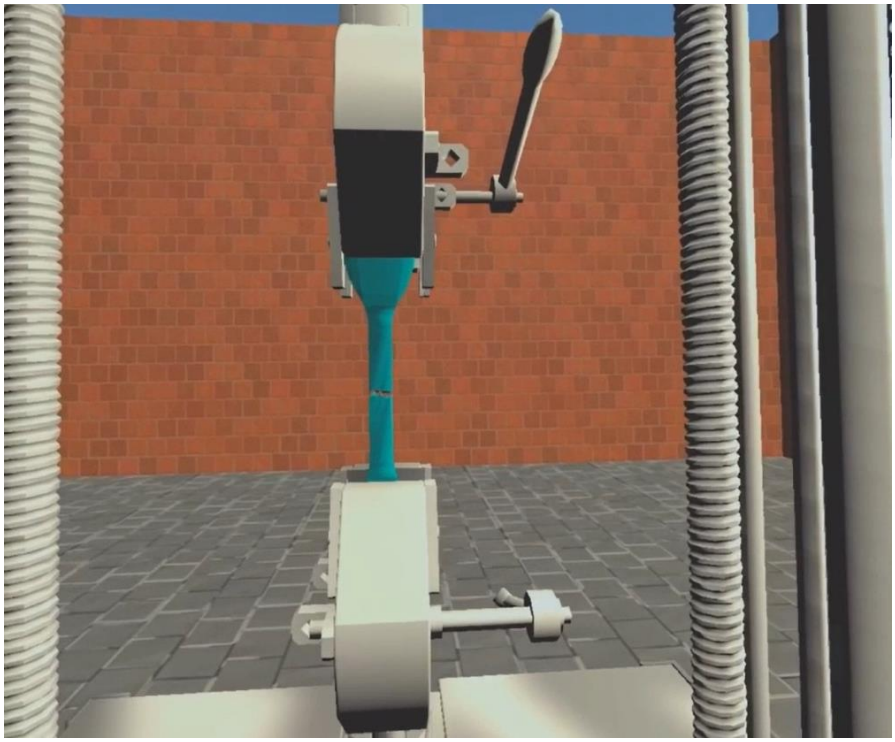

Figure 2: Fracture of Virtual Dog-Bone Sample

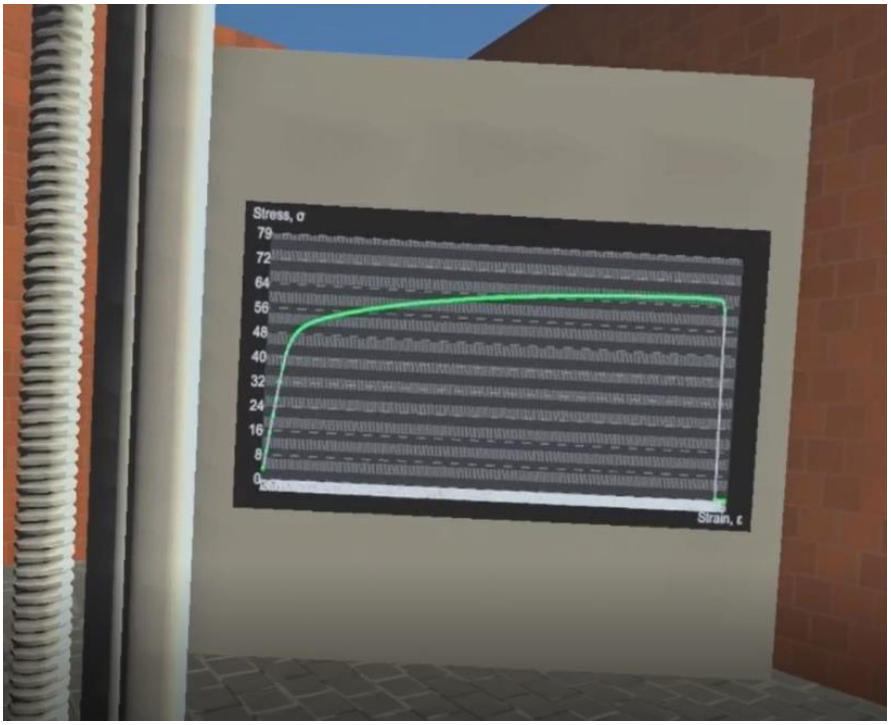

Figure 3: Stress vs Strain Graph Screen

Section 1.4: Continuation: Interactive VR

To continue the progression of the VR lab research, the program that was created for this project expanded upon the previous work and was made to allow users more involvement in the lab rather than just the visual experience. This project encompassed the creation of a more user-involved virtual compression test lab. Summarizing the specific lab that was used as a basis for this project, the study was on the analysis of a compressible and conductive cube and the changes in conductivity with respect to compression of the cube. The cube was analyzed in a scanning electron microscope (SEM) machine and images were taken to witness the closing of the pores and correlation with the electrical flow through the sample. The program that the simulation was being made on was called Unity which was coded using the computer language C#. The VR aspect of the simulation relates to Unity through VR simulation on the gaming company Steam. A walkthrough of the lab through the user was as follows.

The user first dons the VR equipment and the program was started up. The first scene that appears at the start of the program was a giant start button that finishes setting up the virtual hands and their animations. Clicking on the start button opens the virtual lab space that was modeled to look similar to a lab in the University of Oklahoma. In the lab, there were cabinets, tables, a vent hood, 3D printers, a door, and an SEM machine in terms of machinery and furniture.

Figure 4: Virtual Lab Environment

Looking down, the user should be able to see two virtual hands with red arrays shooting out of them if the controllers were held. The user would be able to control the rotation and translation of their hands while performing two actions according to which button they click: pinch and grab.

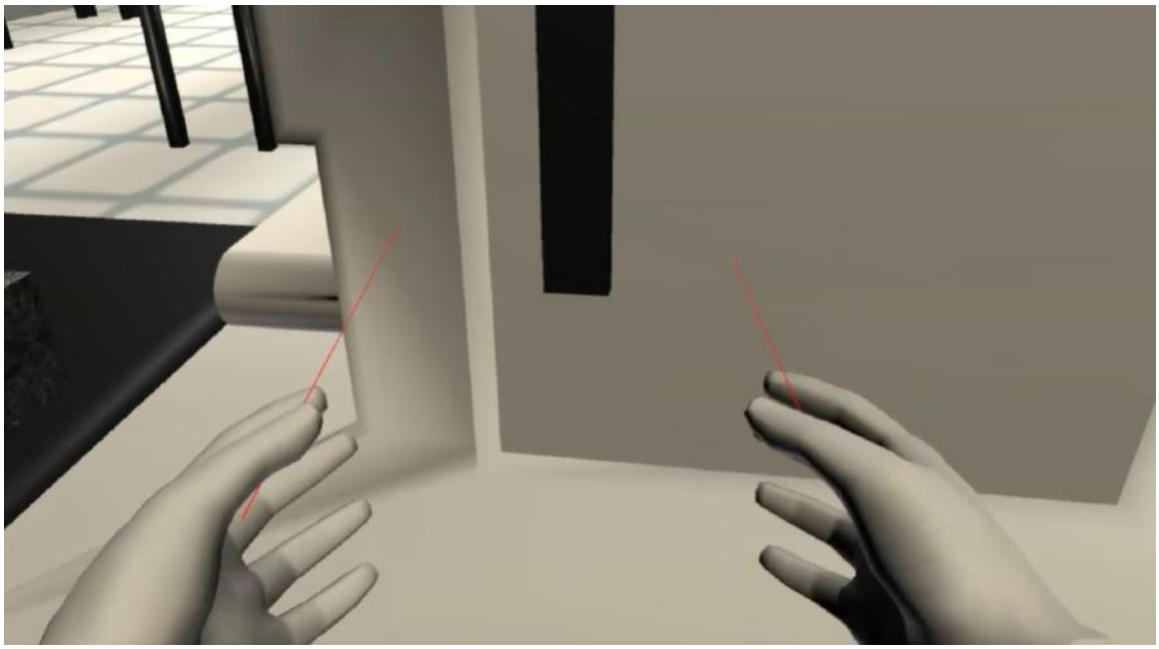

Figure 5: Virtual Hands with Arrays

The user would start off in front of the SEM machine with a table to the left of the user and a computer to the right of the user. Atop the table on the left was the sample to be analyzed in the lab. The sample was interactive and was meant to represent a conductive, compressible cube of Polydimethylsiloxane (PDMS) with carbon nanofibers mixed in the polymer block. Above the computer on the right was a giant button labeled “computer”. The user would first grab the dark handle of the SEM machine and pull the chamber door open. Inside the SEM machine was a chamber with a vice big enough for the sample.

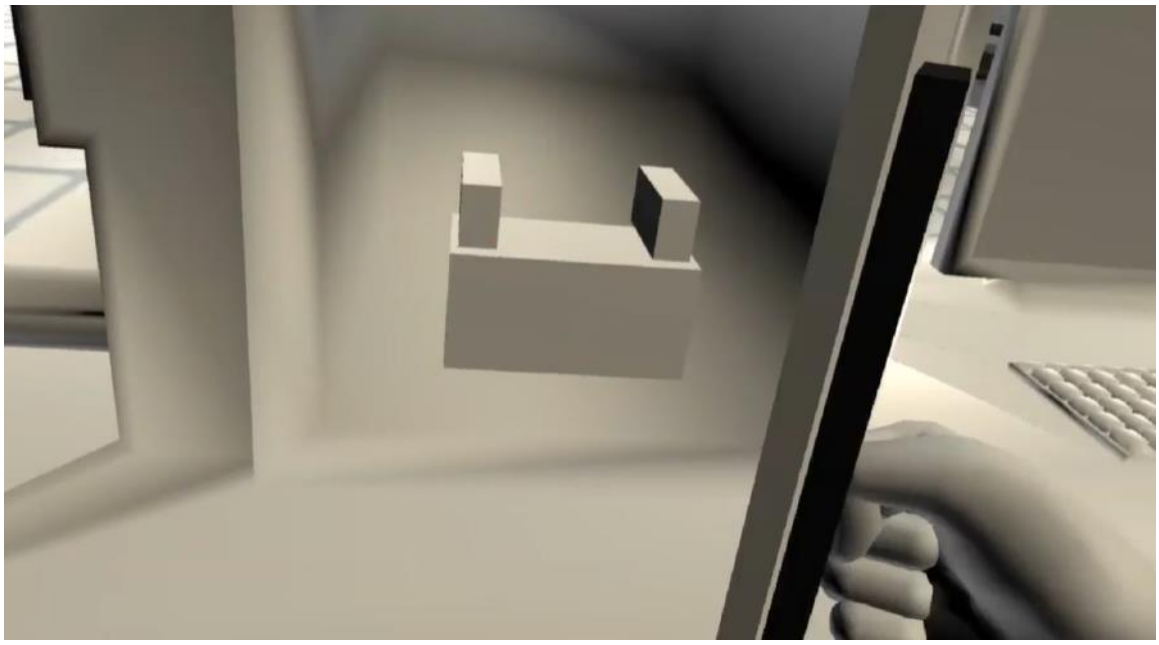

Figure 6: Interior of the SEM Chamber

Once the door was open, the user would grab the sample from the table and place it into the vice inside the SEM machine. The vice inside the SEM machine has a snap area that would automatically pull the sample into the correct position so that consistency was ensured.

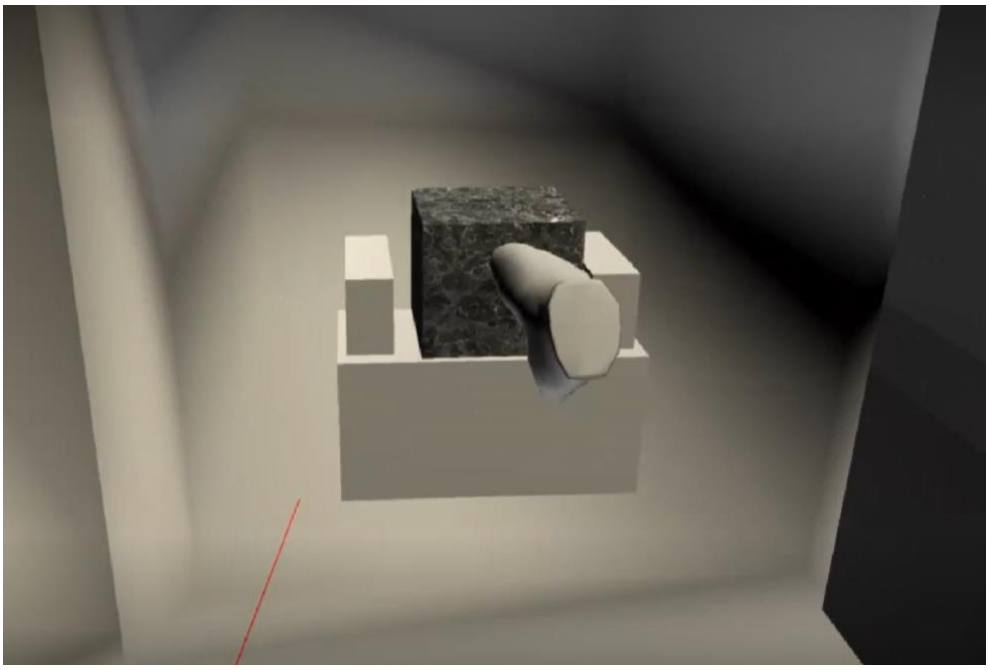

Figure 7: Sample Inserted into Vice

With the sample inside the SEM machine, the chamber door of the SEM was then shut closed. One of the arrays was then pointed at the computer button on the right and the pinch action taken to move onto the next scene.

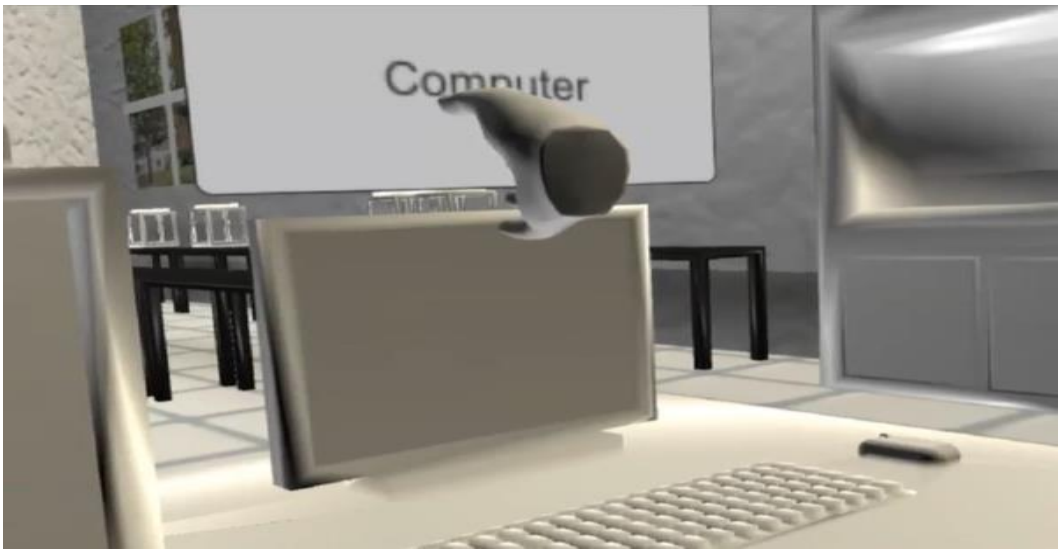

Figure 8: UI Computer Button Interaction

The user was then teleported to be in front of the computer so to minimize the walking required by the user since VR glasses must sometimes be connected to the computer through a cord. On the screen of the computer was a login screen. Clicking on log-in takes the user to the homepage of the computer with an SEM application icon. Clicking on the icons would then open the SEM application where the testing would be conducted.

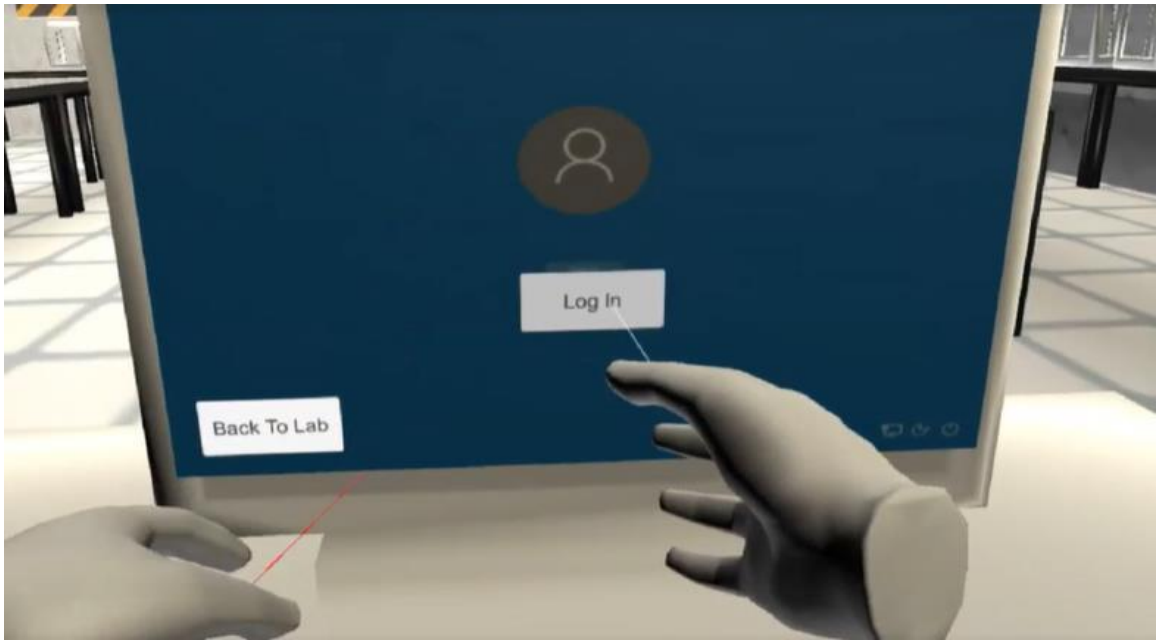

Figure 9: Virtual Computer Log-in

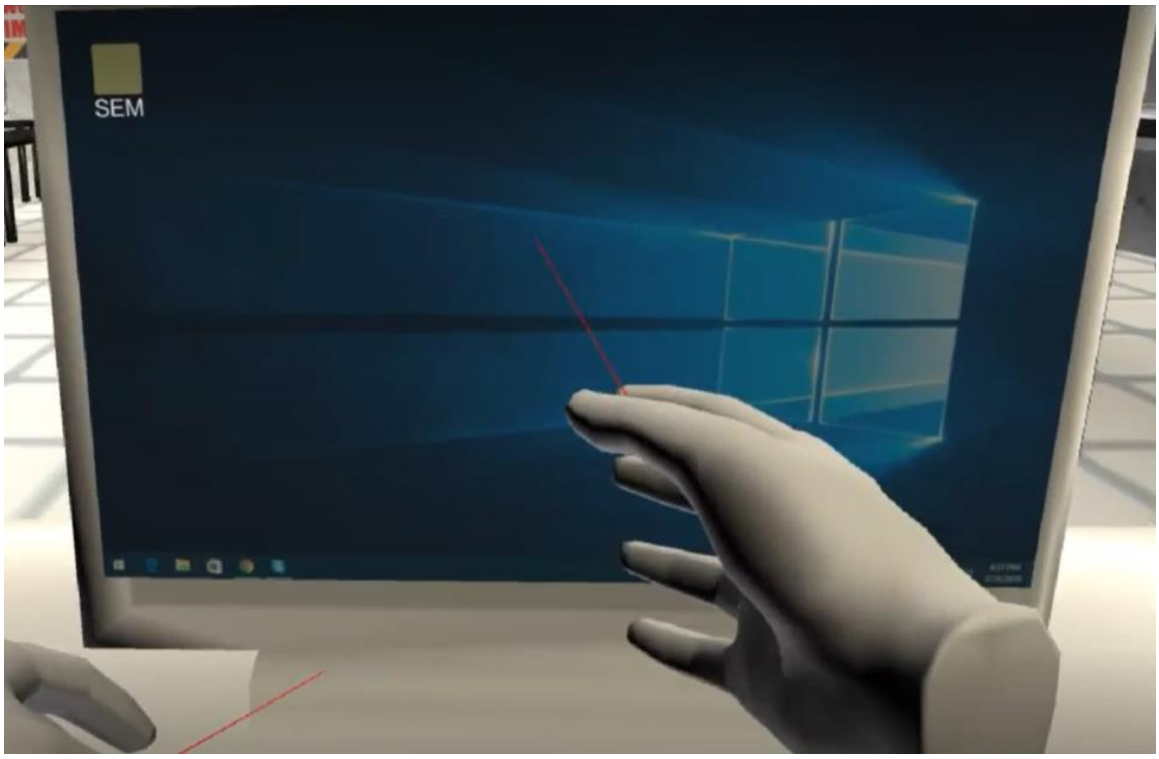

Figure 10: Virtual Computer Home Screen

On the SEM application were three main components. On the left of the screen was a live feed of what was occurring within the SEM chamber. To the right was a live-update graph that would display the conductivity with respect to time. Below the graph was a slider that would control the compression of the vice on the sample as well as display the compression percentage of the sample.

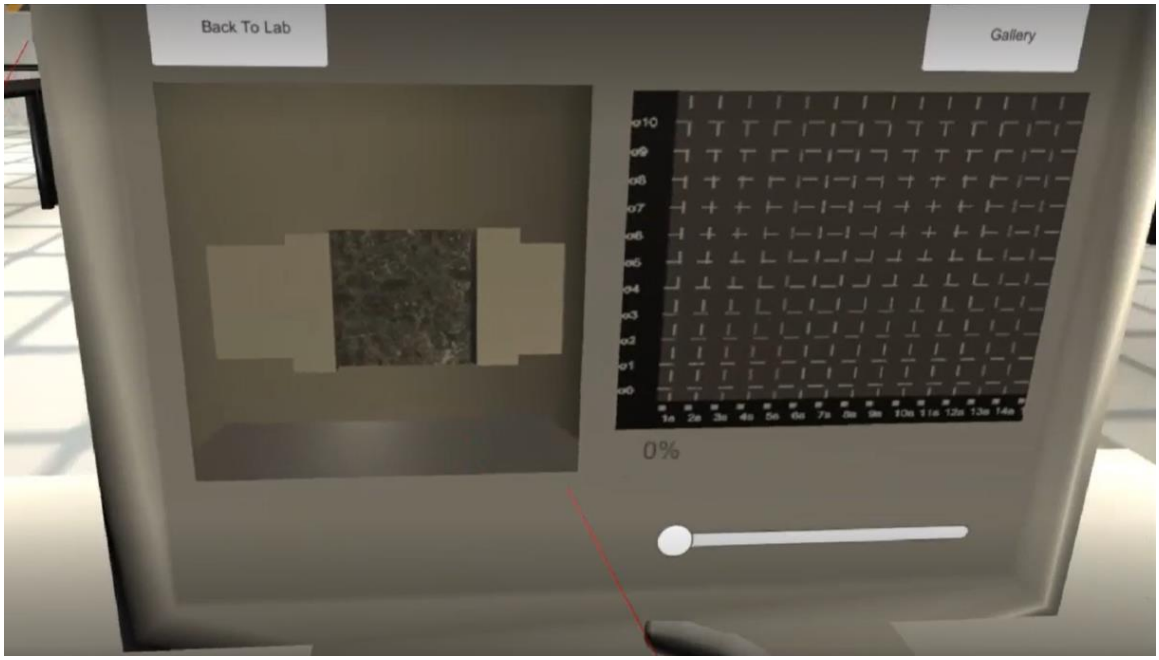

Figure 11: Virtual SEM Application Layout

The user would move the slider bar to control the compression of the sample while taking note of the changes in conductivity within the graph.

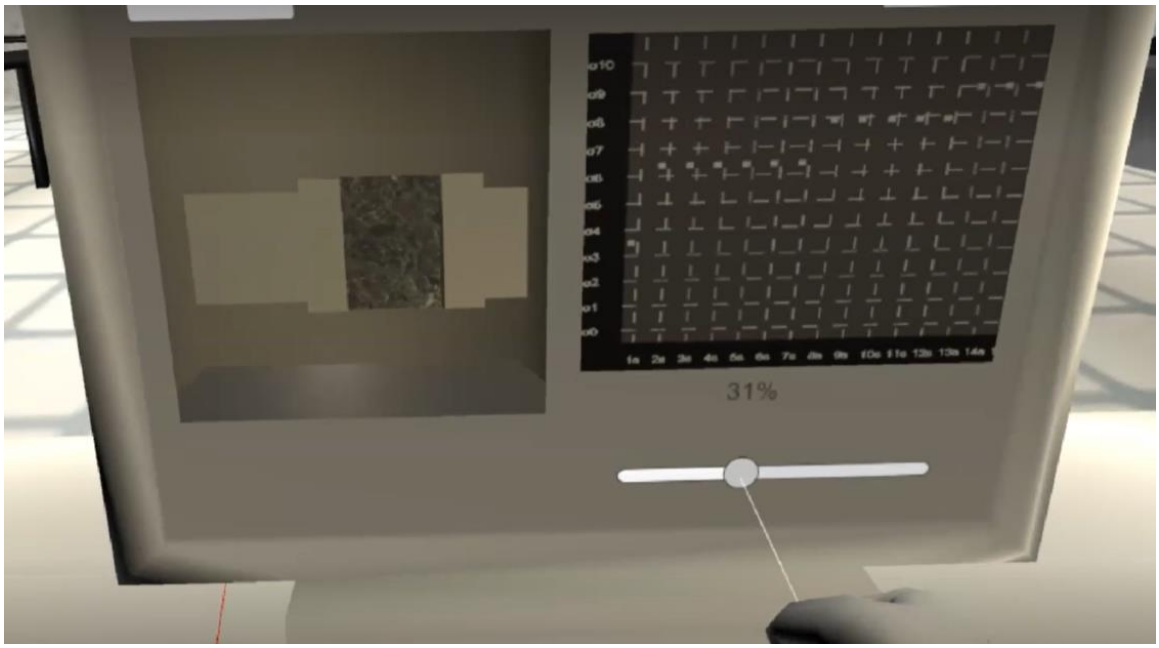

Figure 12: SEM Application with User Interaction

Once the user feels that adequate “testing” was complete, the user would then click on the button on the top right corner of the screen using the array. This would change the screen to display a gallery of the images that were taken during the test. The gallery consists of two pages with a total of 16 images.

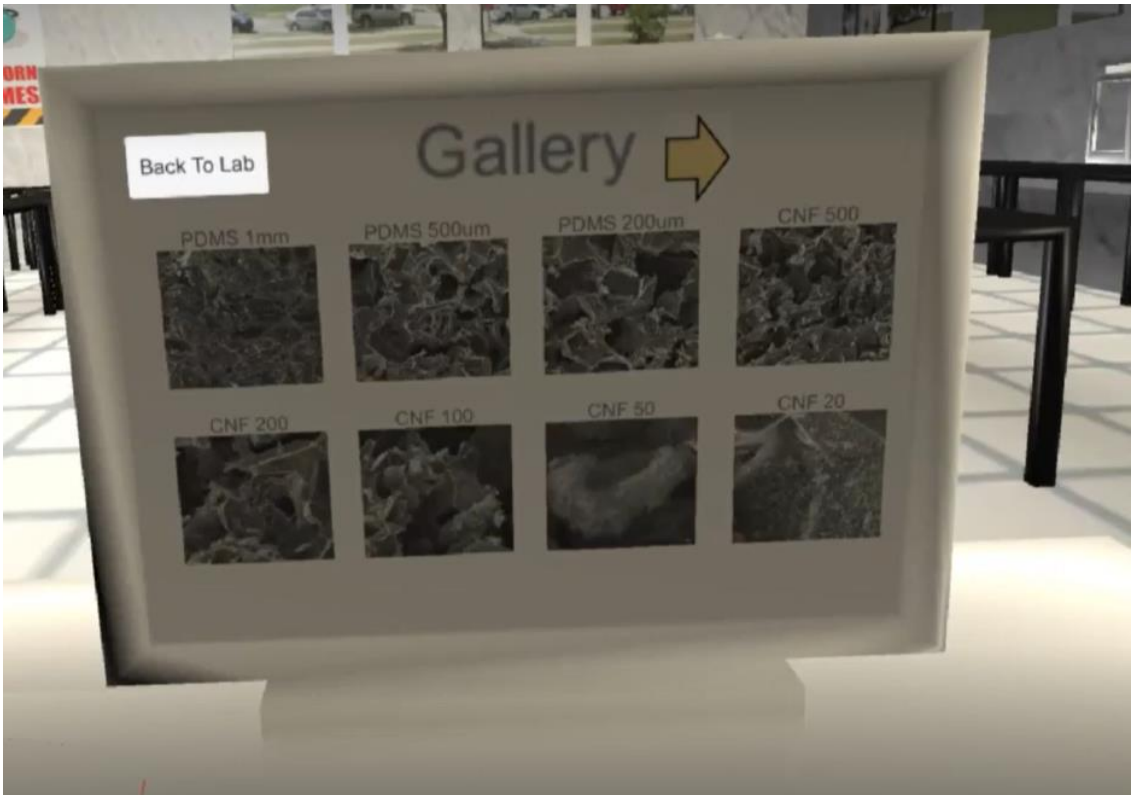

Figure 13: Virtual Computer Gallery

Clicking on the image would open the image in a display mode. In display mode, there were left and right arrow buttons that allow the user to scroll through the images in the gallery. On the top right of the image was a magnifying glass button.

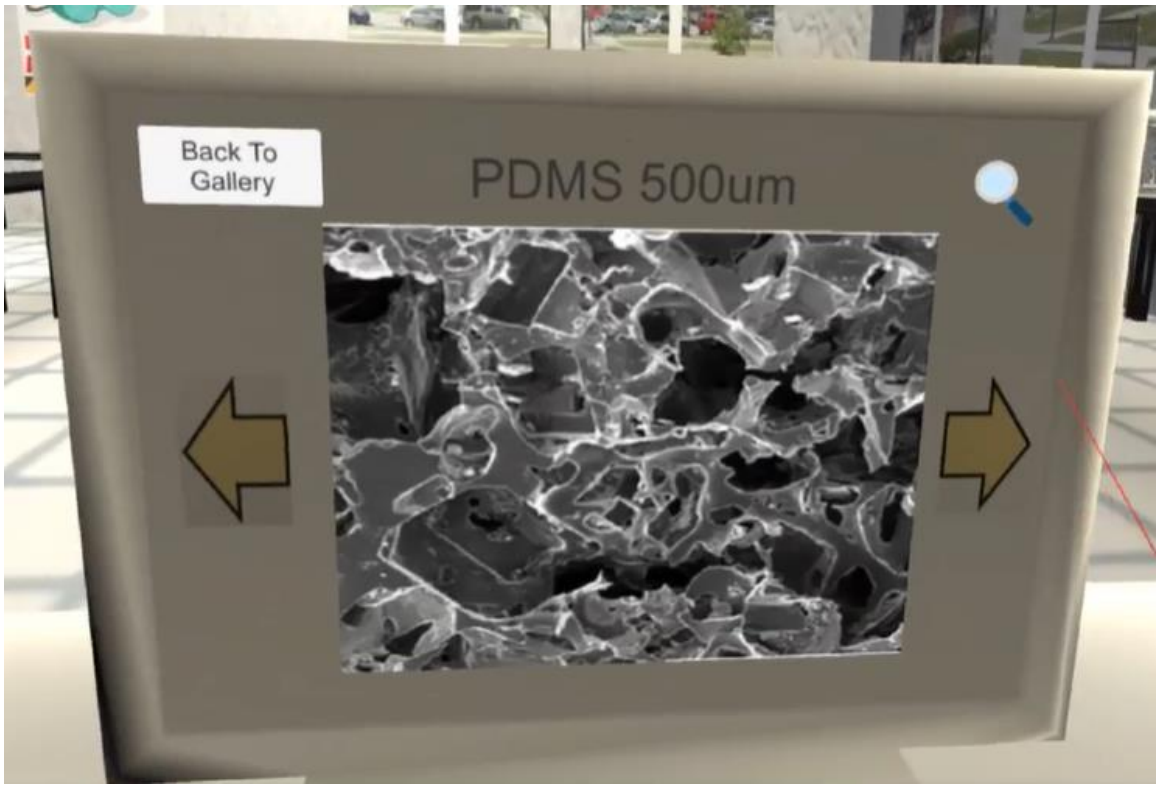

Figure 14: Display Mode of Selected Image

Clicking on the magnifying glass image would open a zoomed in image of the image that was being viewed. Using the arrays and the pinch action, the user would then pan through the image to look at the closer details of the image being viewed.

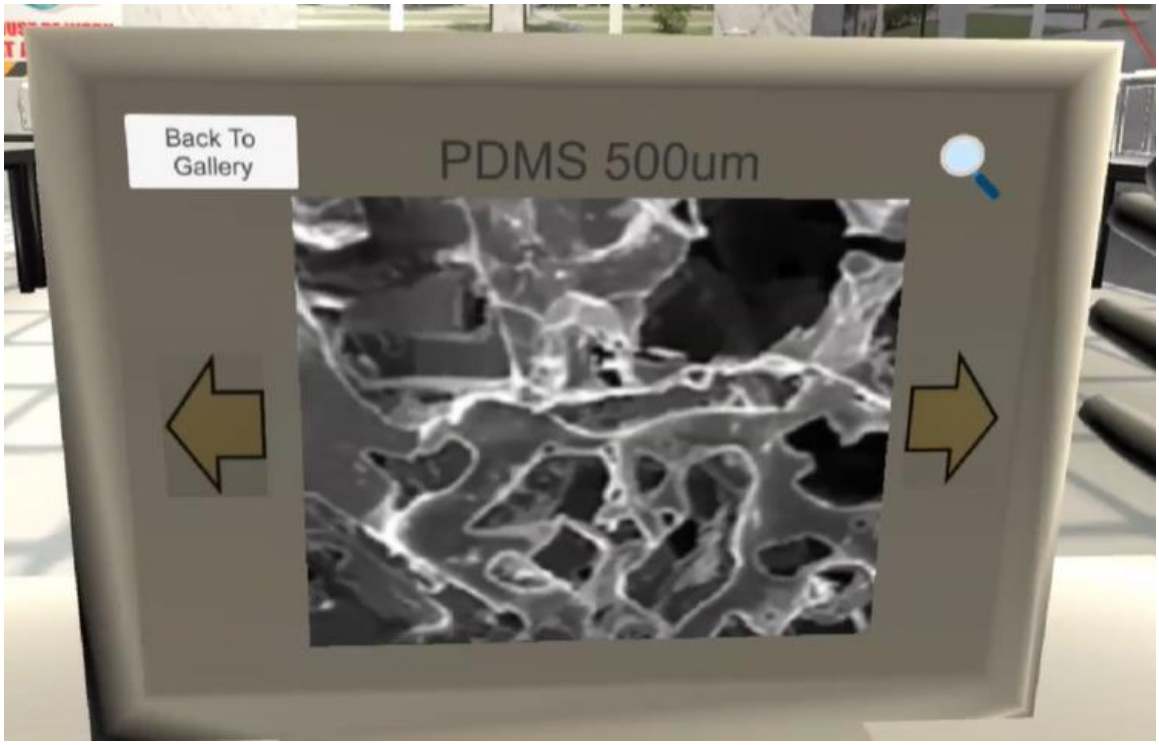

Figure 15: Zoomed-In Display Mode of Selected Image

The user could go back to the gallery at any point using the button on the top left of the screen. Once the images have all been viewed, the simulated lab was complete.

CHAPTER 2: CREATING THE VIRTUAL REALITY SIMULATION

Section 2.1: Equipment Setup and Program Downloads

As mentioned previously, this project came to be due to the appearance of the COVID pandemic. With limited access to labs, projects were put on hold indefinitely. To continue the pursuit of higher education while maintaining a realistic timeline towards completion, the VR lab simulation project was taken on as a new research topic with a focus on education through remote learning. Due to the spontaneity of the project, the initial equipment available for this was limited. An Oculus Quest 2 was available due to the previous work done by Aaron Craig, but his research was conducted on a personal desktop computer that was already had proper specifications for VR due to hobby purposes. With his graduation from the university, there were no available computers that had the proper specifications to connects with the VR goggles and utilize them. Fortunately, it was discovered that a VR game could still be created in Unity with a simulated VR headset controlled using keyboard and mouse as well as a decent laptop. The laptop used in undergraduate was used primarily for essays and 3D modelling using SolidWorks and ran the Unity program without issue.

With the usable hardware setup complete, three main programs were necessary for this project and the integration of VR: Unity Hub, Visual Studio, and Steam. Unity is a game development engine and the main program that the virtual lab was created on. The program allows for the creation of 2D and 3D simulations that could also be integrated with VR. Unity Hub is the name of the program itself that encompasses all the projects made using Unity. It was how new projects were created and how old projects were opened.

The Unity Hub was downloaded from the main Unity website. Once the download was complete, the Unity version was selected as well as the add-ons that were desired. For this project, Unity Version 20191.4.17f1 was used. The only add-on that was included in the download was the Android build support. With the gaming engine downloaded, the coding software was next. The coding language that was used in Unity was C#. A free and simple to use source code editing software compatible with C# was Microsoft Visual Studio. The version of Visual Studio used in this program was 1.65.2 and was downloaded from their website. The last program that was required for VR utilization in Unity was Steam. Steam is a video game distribution software and on their store was a free program called Steam VR which connected the VR headset to the Unity program. Steam did require an account creation, but everything was free. Once Steam was installed, the "Steam VR" program was looked up in the store and downloaded onto the computer. With those three programs downloaded, the development of the VR simulated lab could start.

Section 2.2: Unity Intro and General VR Settings

To learn about the basics of creating a VR simulation, a great resource that was used was the YouTube channel Valem. This channel provided the introductory tools necessary to create VR games which were used and modified to create the unique lab simulation this thesis was based upon. In the video titled "Introduction to VR in Unity- PART 1: VR SETUP" [1], the settings in the Unity program were changed to allow for VR compatibility. To start creating the simulation, a new project was created by opening the Unity Hub and clicking the blue New Project Button as seen below.

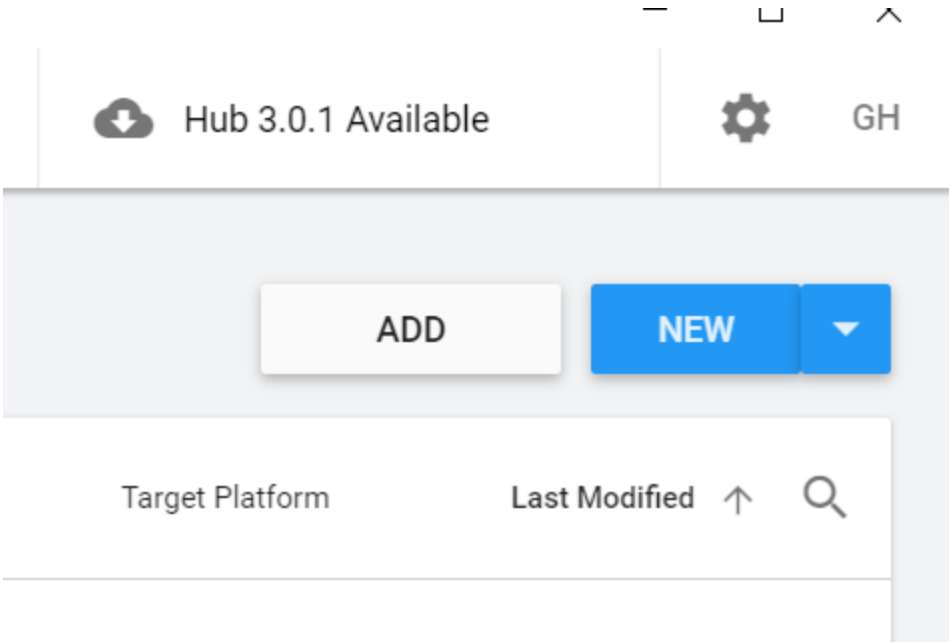

Figure 16: Blue New Project Button

A dropdown should appear. Within this dropdown, the version that was previously downloaded, 2019.4.19f1, should be selected.

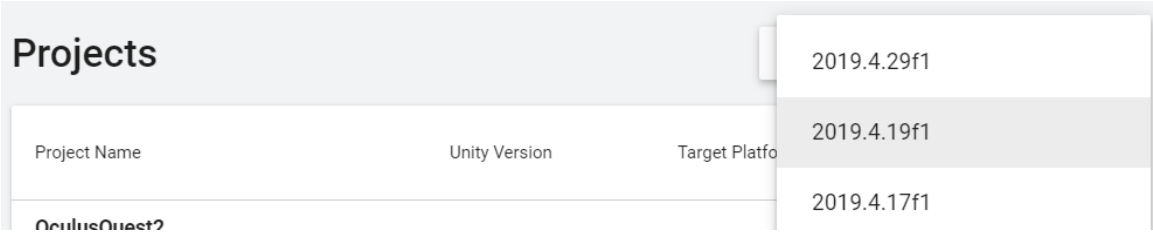

Figure 17: Unity Version Dropdown

Once selected, a new window with different templates would appear. In this window, the name of the project could be made as well as the save location of the project and all the related files. For this particular project, the Universal Rendering Pipeline template was used. Once all the settings had been chosen, the blue create button on the bottom left corner would create the project.

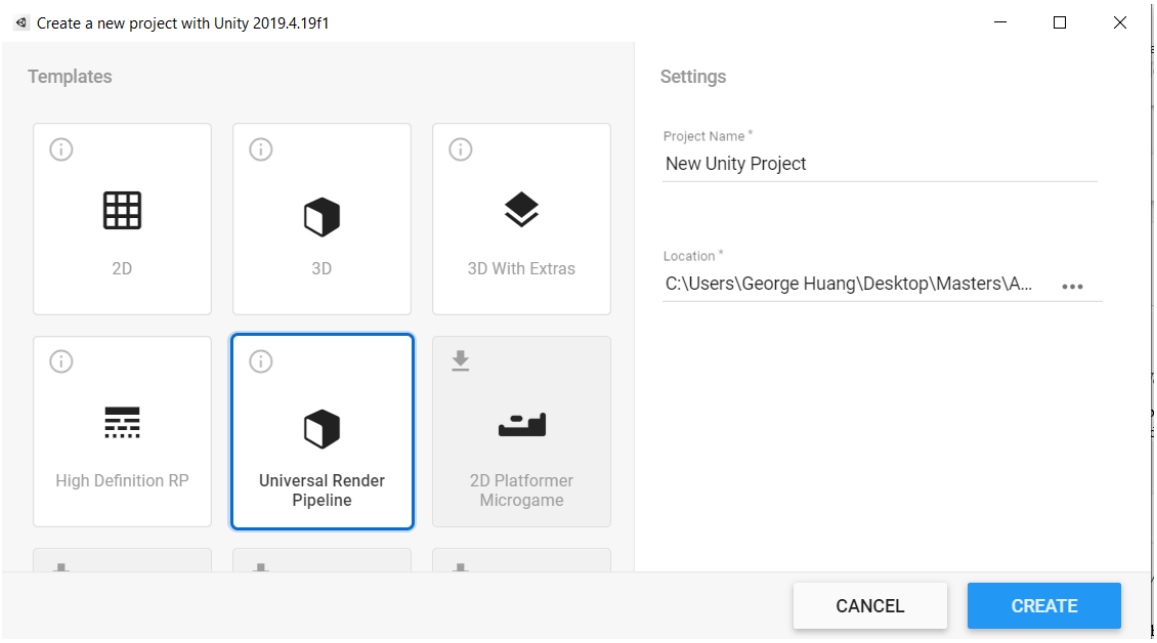

Figure 18: Create New Project Window

Unity would then generate a sample scene that should look similar to the image below.

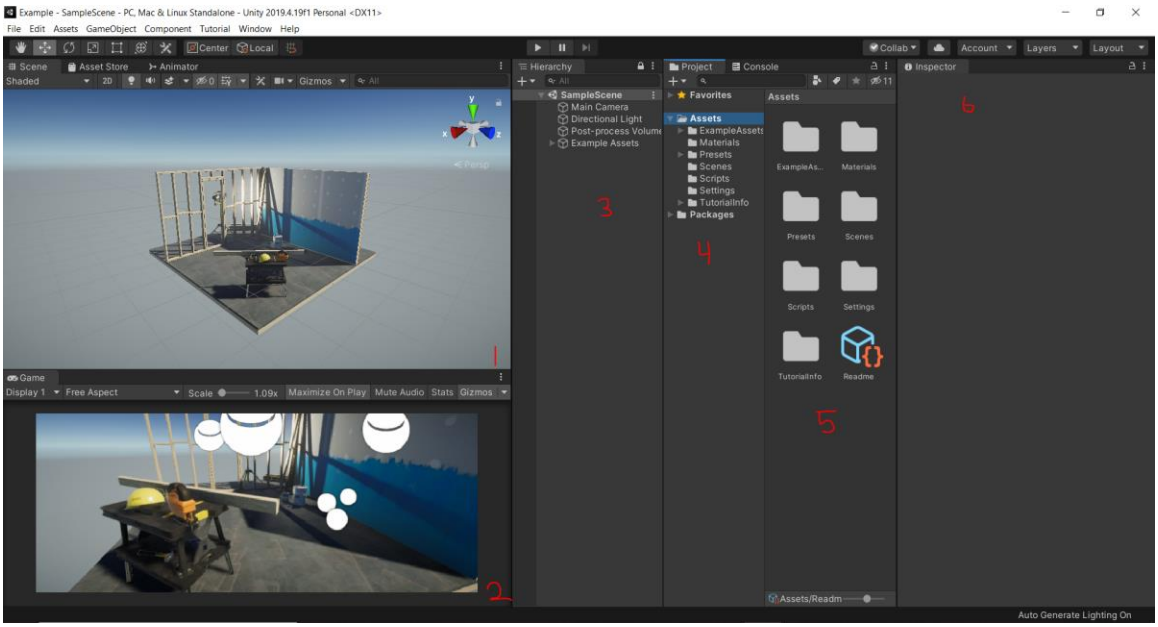

Figure 19: Sample Scene

In Figure 19 the area labeled 1 was the main scene window which displays what the virtual world looks like and allows the users to move the view freely. The area labeled 2 was the main camera window which shows what the user would see when the simulation was started. The area labeled 3 was the Hierarchy window which shows all the objects that were in the current scene's virtual world. Area 4 holds all the assets, objects, and packages in the entire project as a whole. Area 5 displays the contents of area 4 with a thumbnail. Area 6 was the inspector window and it shows the settings and allows the users to manipulate the inner workings of the program and objects. Since the current scene was cluttered with preset objects, a blank scene was desired to start the project. To create a new blank scene, select the scenes folder underneath the assets folder in area 4 to change area 5 to the scenes folder content.

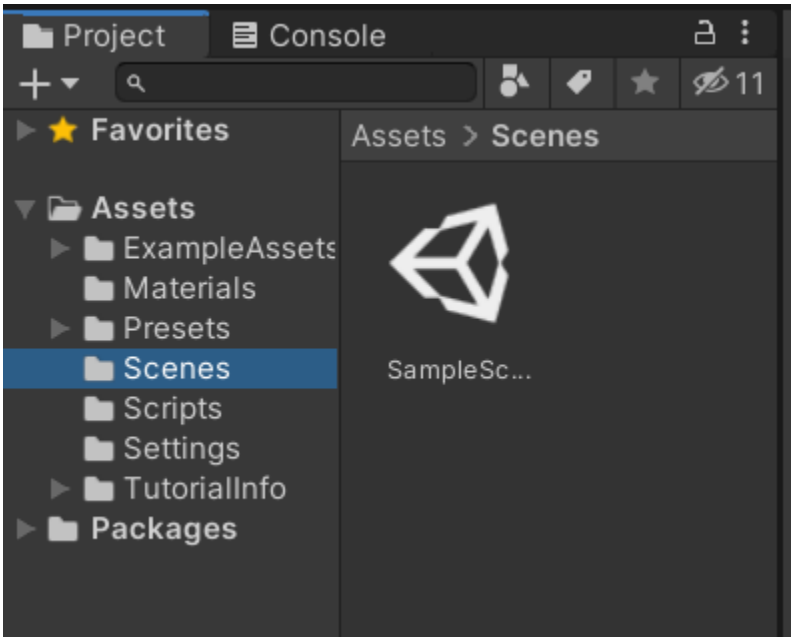

Figure 20: Scenes Folder

In a blank space within the area 5, right click, move down to create, and scene was selected as seen below.

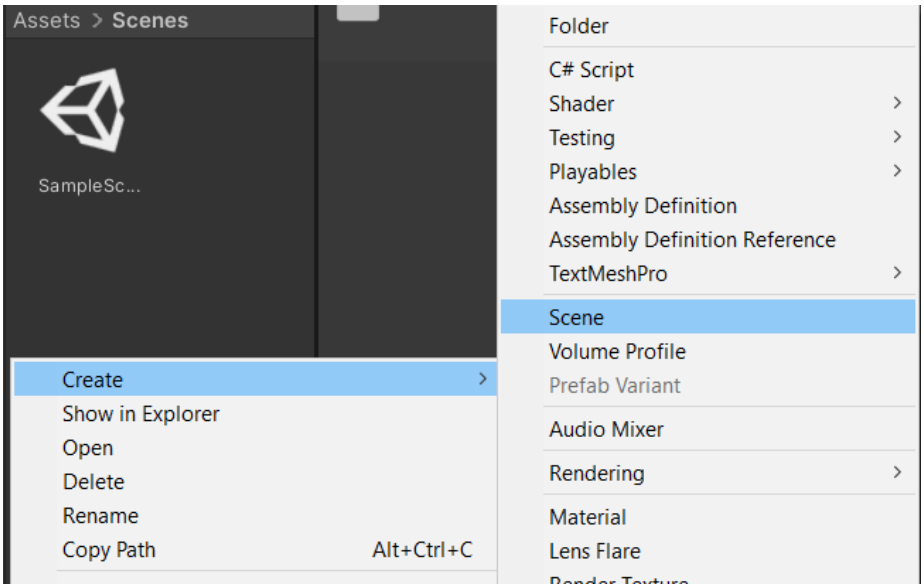

Figure 21: Creating New Scene

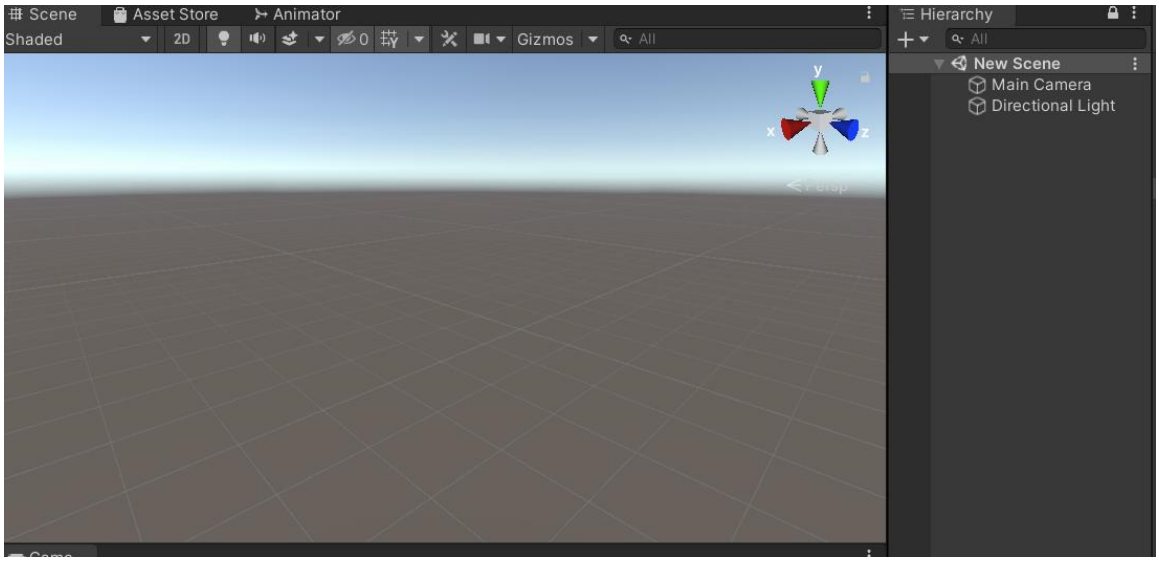

Figure 22: Blank New Scene

The old sample scene was then deleted to avoid confusion and cluster of scenes in the future. Now that the virtual workspace has been cleared, the VR configuration settings and packages were worked on. The first step to integrating VR into unity was to install a package titled XR interactive toolkit. To do this, first go to the window tab in the top ribbon and click package manager. A window title Project Manager would show up. To see all available packages, click the Advanced dropdown and select show preview packages. Scroll down until the package was found and select install.

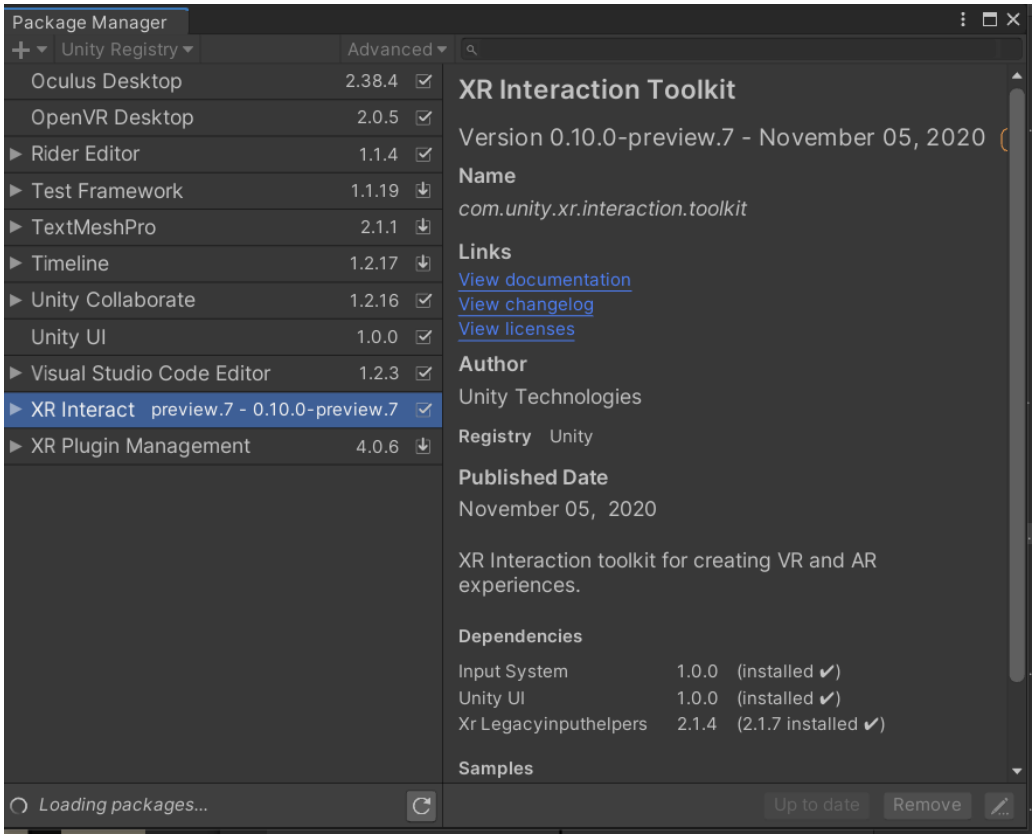

Figure 23: XR Interaction Toolkit in Package Manager

Section 2.3: VR without a Headset

In the case where a virtual reality gear set was either unavailable or incompatible with the available computer system, a mock VR system could be set up on Unity that allows for the simulation of a VR gear-set with keyboard and mouse. Valem made a video dedicated to setting up a mock VR Headset titled “How to Make a VR Game WITHOUT a VR Headset” [2] which guided through allowing the possibility to continue this work with limited equipment. To enable the package installed in Section 2.2, exit out of the window and click on the edit tab in the ribbon. Select Project settings and click player on the left. Click on the box next to where it says Virtual Reality Supported. Scroll down to Virtual Reality SDKs. There should be a list that with Oculus and OpenVR in it. If they were not there, click the little plus sign at the bottom of the list. Click on Oculus and Open VR to add them. In addition, click on Mock HMD to add it to the list as well. This plug-in was what allows the VR program to work without a VR headset. Make sure that the Mock HMD was at the bottom of the list so that the program would still attempt to find a normal VR headset in case one was ever used. To check if everything was working, click the play button at the top. The screen should now display 2 separate screens to simulate the eyes of the VR headset. Once this was verified, exit out of play mode by clicking the play button again.

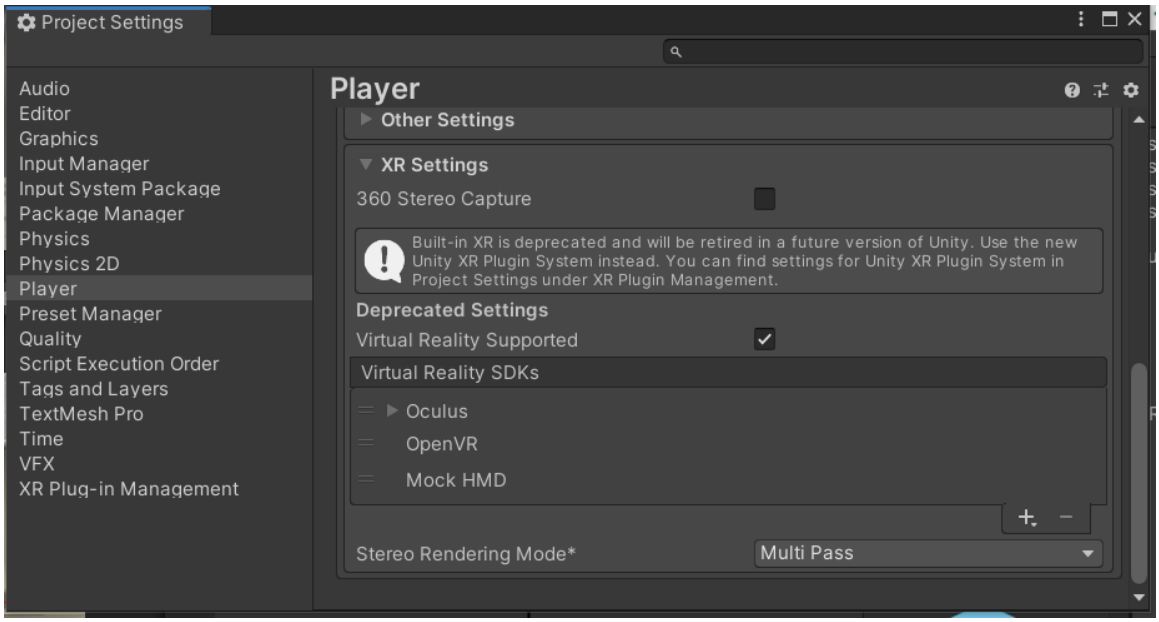

Figure 24: VR SDK list in Project Settings

Once the SDK list has been updated, the core of the simulation could be started. To further check to see whether or not the Mock VR set was working or even to check if an actual device was plugged in properly, a simple script would be needed. In area 4, right click in the empty space underneath the SampleScene and select create empty. This would allow us to create an empty coding script to manipulate. Rename this game object to HMD Info Manager to keep things organized. With the HMD Info Manager selected, click add component in the inspector window, type in HMDInfoManager, and double click new script to open the script. If there was an error or difficulty with opening up the script, that might mean that the right scripting program was not installed on the computer. If Visual Studio Code was installed per section 2.1, there should be no issues. Once the script was open, the first thing to do was be to type “using UnityEngine.XR;” underneath “using UnityEngine;” at the top of the script.

```
Assets > HMDInfoManager.cs
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.XR;
5
```

Figure 25:HMDInfoManager Top Script [2]

Under Void Start, the following was typed in:

```
11          Debug.Log("Is Device Active " +XRSettings.isDeviceActive);
12          Debug.Log("Device Name is : " + XRSettings.loadedDeviceName);
```

Figure 26: Void Start Code Content [2]

Where the top half Checks to see if the VR settings were working correctly and the bottom half lists the name of the device being used. In the case that the device is not available, the output should say Mock VR. The final code looked like this:

```
Assets > HMDInfoManager.cs
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.XR;
5
6   public class HMDInfoManager : MonoBehaviour
7   {
8       // Start is called before the first frame update
9       void Start()
10      {
11          Debug.Log("Is Device Active " +XRSettings.isDeviceActive);
12          Debug.Log("Device Name is : " + XRSettings.loadedDeviceName);
13      }
14
15      // Update is called once per frame
16      void Update()
17      {
18
19      }
20  }
```

Figure 27: VR Debug Code [2]

The debug code was now complete. Save and exit out of the code. To verify this code works, click play and then immediately exit out. In the project window on the right near hierarchy, select the console tab next to the project tab. The output should look similar to this:

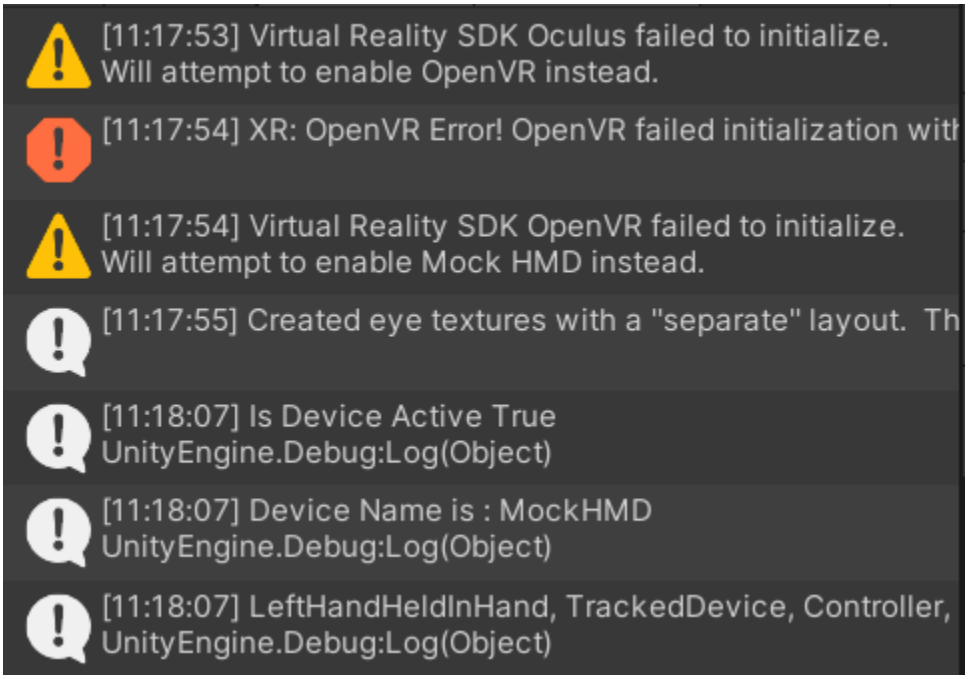

Figure 28: Output of the Console Debug

From top to bottom, the code was stating that it attempted to find the Oculus 2 VR headset but failed. The program then moved down the device list and looked for the Open VR headset but failed. It moved down one last time to the Mock VR headset and found it to be true. It then lists the name of the device use, which in this case was MockHMD. The list of available devices was found in the Player setting as mentioned in a step above. This would serve as a simple debugger for device plug-ins. To integrate the VR into the gameplay itself, A mock headset and controller would need to be added into the Hierarchy. To do so,

right click in the blank space under hierarchy, hover over XR, and click Room Scale XR Rig. This should create a new item in the hierarchy labelled XR Rig. In the dropdown of XR Rig, there should be a camera offset which branches to main camera, left-hand controller, and right-hand controller. This represents the VR headset and controllers in the program.

Figure 29: XR Rig in the Hierarchy

If one of the controllers were clicked, it could be noticed in the inspector under XR Controller that all the input actions were empty which meant that the mock VR system wasn't connected to the computer yet. Rather than setting it up, Unity has a premade action set that makes the process a lot easier. Going back to the package manager, select XR interactive Toolkit and select "import to Project" next to Default Input Actions. To enable the preset input actions, go to the project window and find the folder titled "Default Input Actions" using the path seen below.

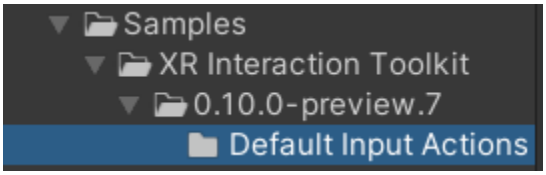

Figure 30: XR Input Actions Folder Path

Highlight the item in the folder titled XRI Default Left Controller and select “Add to ActionBasedController default” in the Inspector panel. Do this for the XRI Default Right Controller as well. The Inspector panel should look like the following:

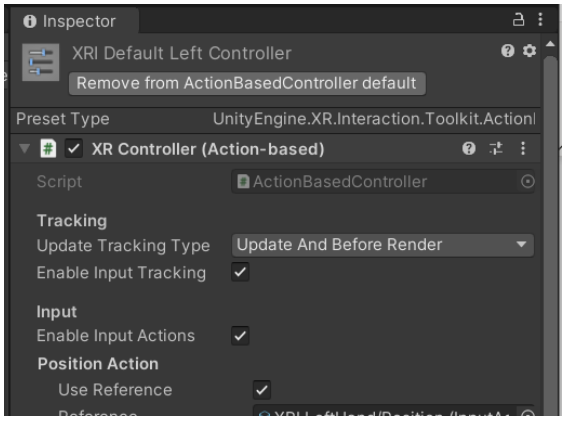

Figure 31: Controller Inspector Panel

Once they have been enabled, they must be labeled in the preset manager which could be found in the project settings. Under the column labeled filter, type Right next to XRI Default Right Controller and type Left under XRI Default Left Controller as seen here:

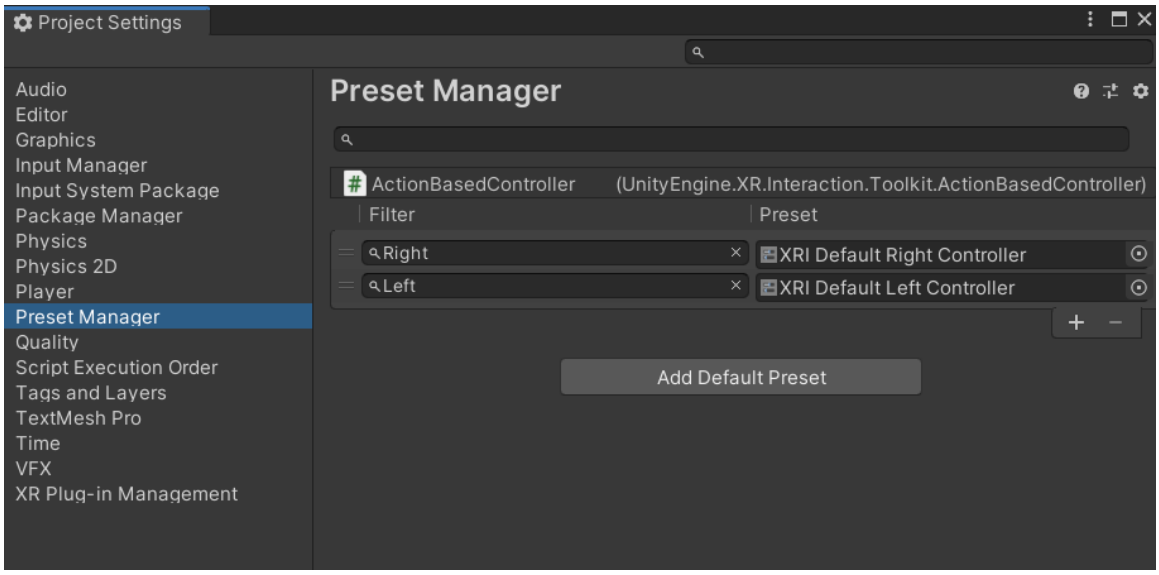

Figure 32: Preset Manager Under Project Settings

Exit out of the Project manager and look at the inspector panel with the controller highlighted in the Hierarchy again. The default actions should still be empty. To update the settings that were just changed, delete the XR rig from the hierarchy window and re-add it. The inspector window should now have the input actions pre-populated under the controller. To activate these actions with the XR rig itself, highlight XR rig, click Add Component, type in Input Action Manager, and click the dropdown under Action Assets. In the project window with the Default Input Actions folder still open, click and drag the XRI Default Input Actions into the Action Assets. Still in the XR rig inspector panel, make sure to set the position coordinates to (0,0,0). This would make creating the simulation a lot smoother in terms of reference points later down the line. The inspector panel for XR rig should now look like this:

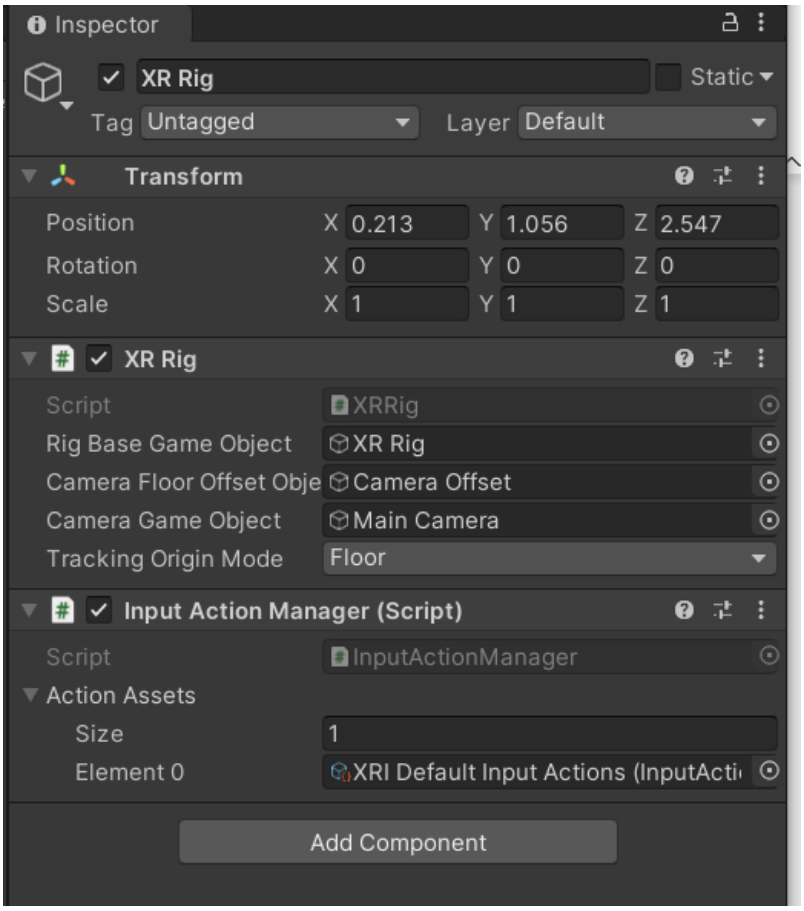

Figure 33: Updated XR Rig Inspector Panel

The previous steps were just to set up the connection between the VR gear to the program. However, in the case that a VR headset was not available, the program at this stage would still not work due to the lack of hardware. In order to make the XR rig usable, a Mock VR device simulator must be enabled. To do so, go back to the package manager. Under XR Interactive Toolkit, click the “import into Project” button next to XR Device Simulator. Exit out of the window and go the project panel on the right. In the search bar within the panel, type in XR device simulator and drag the one with the blue box into the Hierarchy as seen below.

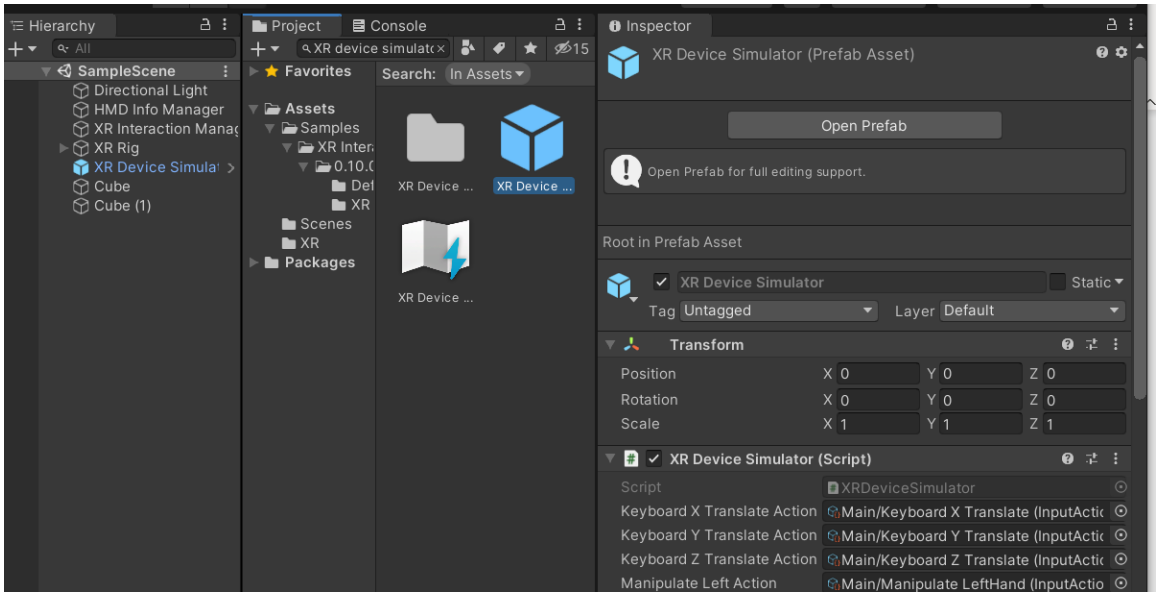

Figure 34: XR Device Simulator

This device simulator would allow the VR simulated device to connect to the keyboard and mouse so that the user could control the mock gear in the simulation. Each action could be double clicked so that the keys could be tailored to the user's preference. If the program was started, the camera, aka headset, would be able to be moved around. To do so, right click the mouse button and move it around to move the view around. It should be noticed that there was a red line on the screen. That red line were two lines overlapping, one for each hand. These two lines represent the user's "hands" action line. These actions lines must be aimed at an interactive object and a key must be pressed for it to work. It could also be noted that the red lines were not attached to anything. Just to test the applicability of a model to represent hands, exit out of the game and click on the dropdown for the XR Rig in Hierarchy. Keep going down until the left and right controller objects appear. For each one, right click, select 3D object, and cube. This would create cubes as the hands for a temporary measure. Make sure to scale the cubes down to a more reasonable size and

once again, change the position of both to (0,0,0). With that, the VR Settings in the case of both having and not having a VR headset was complete.

Section 2.4: Virtual Hands and Input Animations

Now that the settings of the program allow for VR utilization, the next step was to improve the realism of the virtual experience. If the program was run, the headset should work correctly but the hands were currently represented by blocks. To improve the immersion, hands that could be animated were ideal. Another video made by Valem titled "Introduction to VR in Unity – PART 2: INPUT and HAND PRESENCE" [3], the method of integrating controllable hands was taught.

For the program to register the hand controllers when played, regardless of mock or physical controllers, a code was needed to connect the two. To start, a new empty game object was created in the hierarchy panel and named Hand Presence. With this game object highlighted, in the inspector panel, the add component button was pressed. In the search bar that appeared, HandPresence was typed in and a new script was created and added to the assets. Clicking on the script name should open the script for modification. The script that was created according to the video can be seen in Appendix A [3]. The main components of the script include the controller debug, the animation association, and redundancy prevention. The portion of the script that debugs the controller can be seen below.

```
36          if(devices.Count > 0)
37          {
38              targetDevice = devices[0];
39              GameObject prefab = controllerPrefabs.Find(controller => controller.name == targetDevice.name);
40              if(prefab)
41              {
42                  spawnedController = Instantiate(prefab, transform);
43              }
44              else
45              {
46                  Debug.LogError("Did not find corresponding controller model");
47                  spawnedController = Instantiate(controllerPrefabs[0], transform);
48              }
49              spawnedHandModel = Instantiate(handModelPrefab, transform);
50              handAnimator = spawnedHandModel.GetComponent<Animator>();
51          }
```

Figure 35: Controller Debug Section of Appendix A Code [3]

To make sure that the controllers were properly connected, the script first searches for them and tries to match it with one of the controller models to see if the controller would work with the program. The program would then spawn a hand model where the controllers would be and apply the hand animation portion of the script to the models. The hand animator portion of the script can be seen below.

```
void UpdateHandAnimation()
{
    if(targetDevice.TryGetFeatureValue(CommonUsages.trigger, out float triggerValue))
    {
        handAnimator.SetFloat("Trigger", triggerValue);
    }
    else
    {
        handAnimator.SetFloat("Trigger", 0);
    }
    if(targetDevice.TryGetFeatureValue(CommonUsages.grip, out float gripValue))
    {
        handAnimator.SetFloat("Grip", gripValue);
    }
    else
    {
        handAnimator.SetFloat("Grip", 0);
    }
}
```

Figure 36: Animation Association Section of Appendix A Code [3]

For this simulation, the animations that the hand model would need were pinching and gripping. The animation section of the code sets the trigger button and the grip button on the VR handset controllers available for model association and animation. Which will be discussed later in this chapter section. The last portion of the code was the model redundancy checker.

```
74      // Update is called once per frame
75      void Update()
76      {
77          if(!targetDevice.isValid)
78          {
79              TryInitialize();
80          }
81
82          else
83          {
84              if(showController)
85              {
86                  spawnedHandModel.SetActive(false);
87                  spawnedController.SetActive(true);
88              }
89              else
90              {
91                  spawnedHandModel.SetActive(true);
92                  spawnedController.SetActive(false);
93                  UpdateHandAnimation();
94              }
95          }
96      }
```

Figure 37: Redundancy Prevention Section of Appendix A Code [3]

This section ensures that the hand models and the controller models do not spawn at the same time. This section also ensures that the hand animations only occur when the hand model was being used in the program.

After completing the script, the code now knew how to use the models and associate handheld controller buttons to certain animations. The last step to complete the hand input and animations was to import hand models and applying animations. The hand models were provided by Valem in the comment section of his part 2 video [3]. Once the hand models were downloaded, they were uploaded into the project folder and individually moved into the controller prefabs list under the hand presence script in the inspector. In Section 2.3, blocks were described to have been used as temporary hands visually for when the program was started. The same steps were taken to replace the blocks with the hand models that were imported. The game object created earlier was dragged into assets and deleted from the hierarchy. This game object was then duplicated, with one being renamed Hand Presence Left and the other Hand Presence Right. For each one, the respective controller was associated with the respective game object to ensure that both were independent of one another. Both game objects were then dragged from the assets to the individual hands in the VR Rig in the hierarchy. This was to associate the hand presence code with the VR hands. With this, the user now had hand models when running the program.

To associate animations with the hand models and buttons, an animator was needed. Within the package that had the hand models included three poses that the hand models could make: default, pinch, and grip. The first step to adding these poses to the model was

to add an animator to the assets by right clicking in the project asset window, going to create, and clicking animator controller. Like the hand presence game object, a left and a right animator was needed. Starting with the right hand, the animator was renamed Right Hand Animator. This animator was then dragged into the controller line of the animator box within the hand model inspector panel.

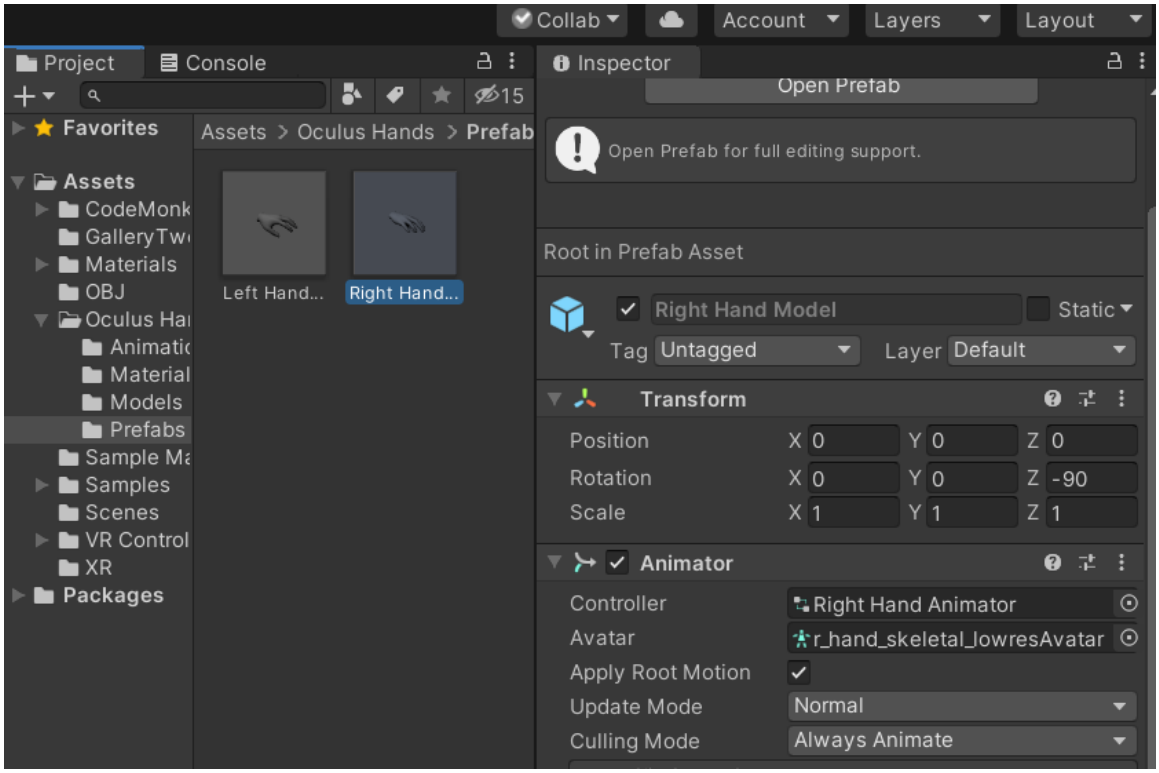

Figure 38: Animator added to Model

The model of the right hand was then modified so that they could change poses and become animated. To do this, the animator was first selected. Then in the top ribbon, the window button was dropped down, the animation side extension was opened, and the animator button was clicked. This was to open the actual animator panel.

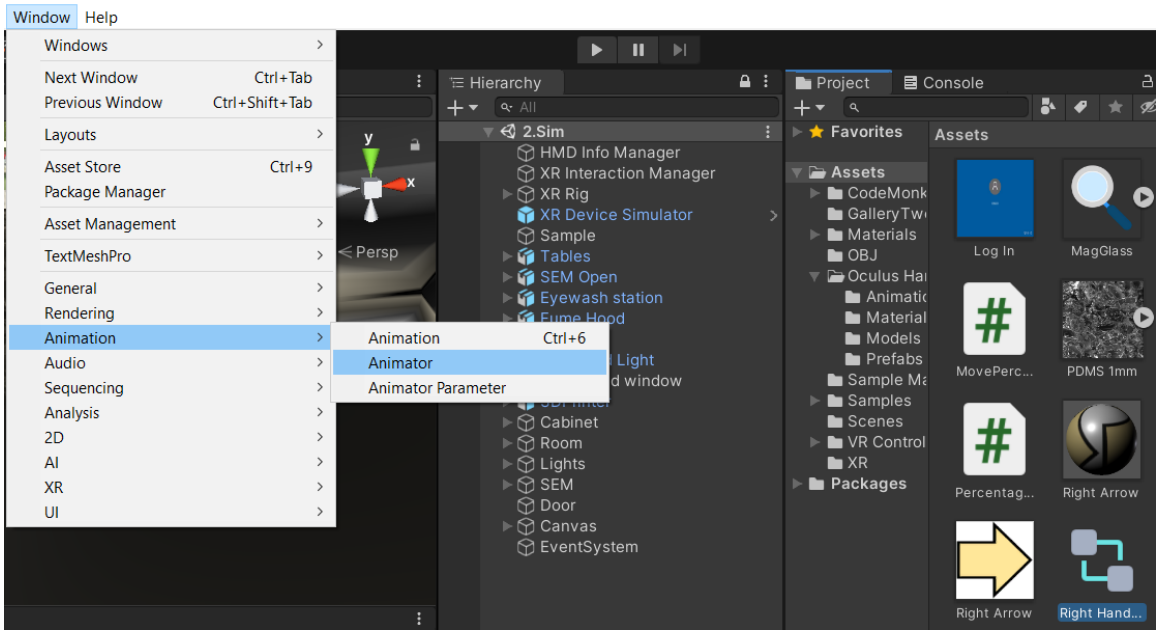

Figure 39: Path to Animator Panel

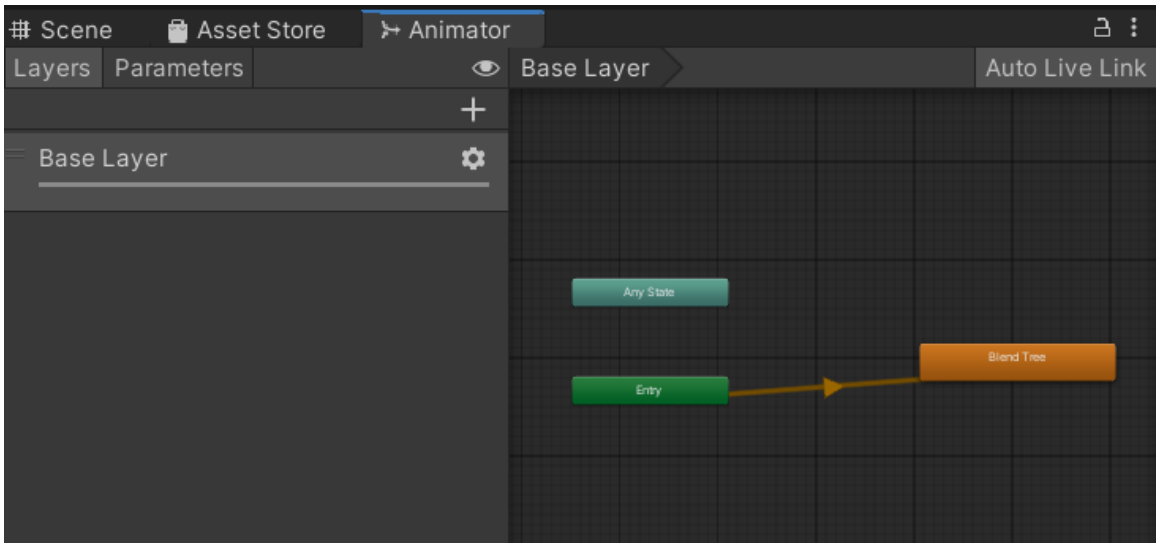

Figure 40: Animator Panel

Once the animator panel was opened, the next step was to create float parameters for the buttons used on the controllers by clicking the plus button and renaming one to Grip and another to Trigger. Having parameters, the blend tree was the next thing added to the

animator panel. Right clicking near the green entry box within the animator panel and selecting Create State and Blend Tree. Double clicking on the Blend tree would then open it up to allow for modification. In the inspector panel, the blend type was changed to 2D Freeform Cartesian. The parameters in the inspector were then changed to Grip and Trigger so that the animations added later could be associated with the parameters and controller buttons. In the motion box in the inspector, the plus sign was clicked and 4 motion fields were added. The positions of the motion fields were the changed to (0,0), (0,1), (1,0), (1,1). In the animation sub folders within the Oculus Hands folder under assets, the different poses of the hand model exist. From top to bottom, the poses added were default (labeled take 001), right hand pinch, right hand fist, and right-hand fist as seen below.

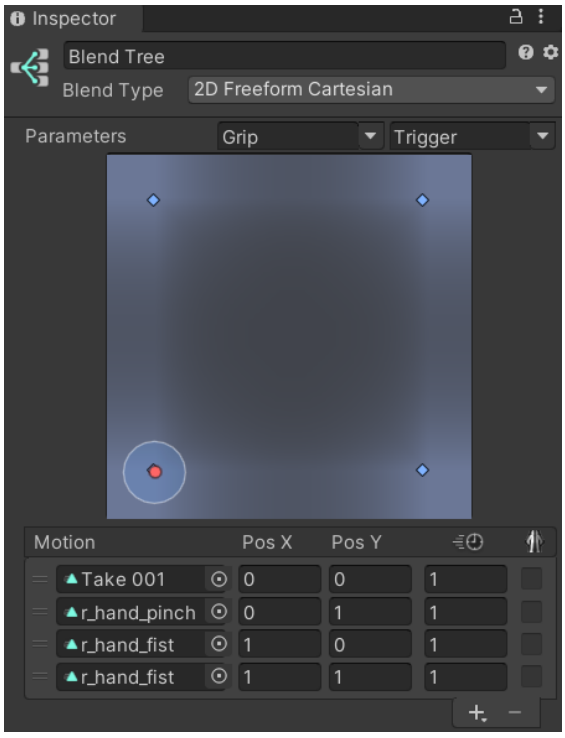

Figure 41: Motions and Positions of Blend Tree

With that, the animation of the right hand was completed. For the left hand, the similar steps were followed with the difference being the left-hand poses being used instead of the right-hand poses. Once finished, testing the program produced two solid hand models that move accordingly with the user's controllers and the trigger button causing the hands to pinch while the grip button causing the hands to grip.

Section 2.5: Virtual Environment and Textures

At this point, the setting modifications of the program for VR gear integration, model association, and animation were complete. The next step of the project was to start creating the virtual environment that the user would work in. Valem's Part 1 video taught the basics of adding objects to the program as well as simple texturing and coloring [1]. Since the SEM lab was conducted win an enclosed space, the first step to creating a virtual environment was to create a room. To do so, walls, ceilings, and the floor were created using in program planes. To create a plane, the left mouse was clicked in the hierarchy panel with 3D object and plane being clicked. The floor plane was the first plane created. To make sure the size of the plane was proportionate to the user, a cycle of running the program and adjusting the scale of the object was conducted until the desired size was obtained. Once the ideal size was reached, the plane was duplicated and raised to create a ceiling. Four more planes were then created to connect the floor and ceiling while encapsulating the space.

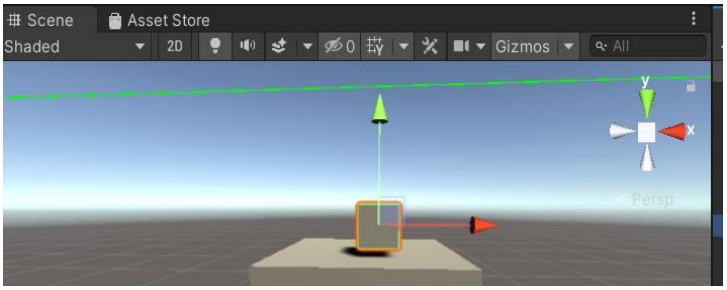

Figure 42: Object Scaling Capabilities in Scene Panel

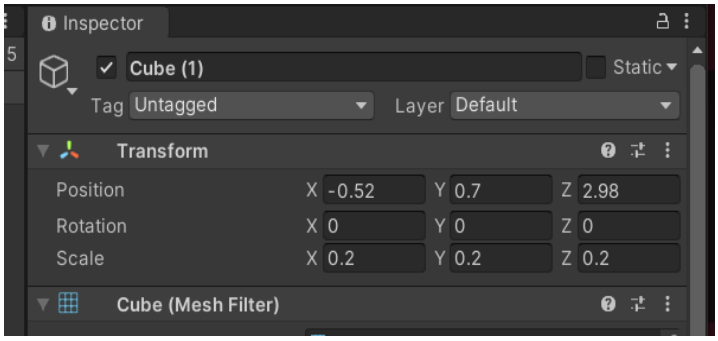

Figure 43: Object Scaling Capabilities in Inspection Panel

When creating the simulation for the compression test in unity, many complicated and simple objects were needed for both the test simulation itself as well as background objects. When creating objects, it's important to understand how simple or complex the object desired is. For simple shapes such as spheres and cubes, creating the objects directly inside of unity was ideal. Objects created within unity work a lot better with the unity programming than objects created using 3D modeling software such as SolidWorks. To insert a generic shaped object into unity, right click inside of the hierarchy panel, go down to 3d object, and select the desired shape. Once the shape appears, the scale and position of the object could be changed to whatever desired.

If protruding details or extensions to the shape was wanted, Valem taught that a 3d object could be spawned directly connected to the original object [4]. To do so, right click on the object created in the Hierarchy and click another 3d object. A dropdown should appear on the original 3D shape with the new object within the dropdown. This means that the two objects were now joined together to become a whole assembly. The objects could be moved independently of each other if the sub-object was clicked on but if the original object was selected, the whole assembly could be moved as a unit.

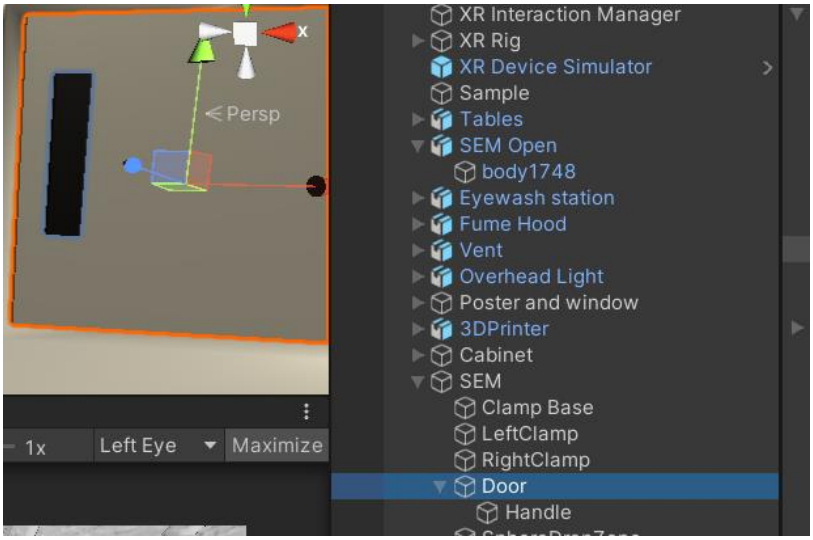

Figure 44: Assembly Grouping

An example of an assembly created in the simulation lab would be the SEM Machine door. All involved objects were connected and could be translated to become the object that was desired.

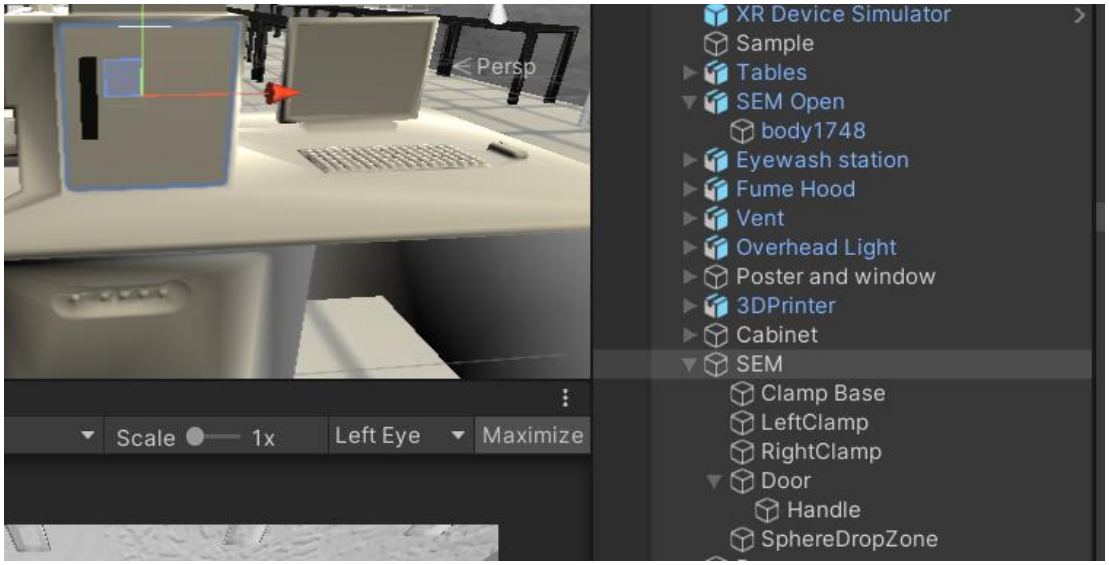

Figure 45: SEM Assembly

If a more complex object with exact dimensions or extruded cuts were needed, a more advanced 3d modelling program would be needed. The one that was used for various objects in the Compression Lab Simulation were created using SolidWorks.

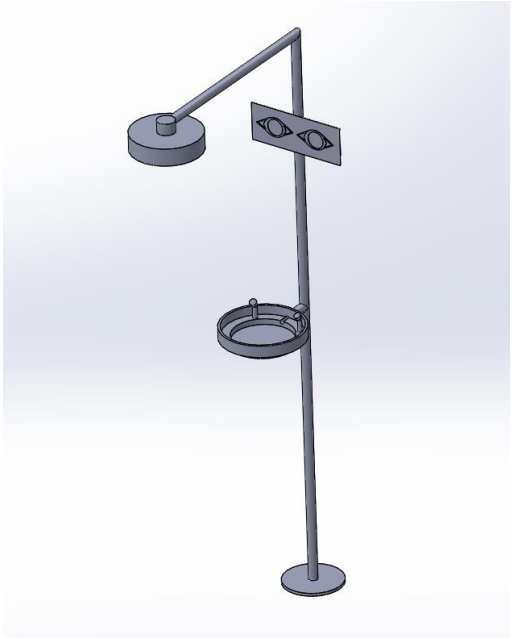

Figure 46: Eyewash Station Made in SolidWorks

Once a SolidWorks model has been created, the object must be saved as a suitable file-type before it could be imported into unity. The file-type that was used in this case was 3MF. Once saved, an online converted was used to convert the 3MF file into an Obj. file. The new file could now be directly dragged into Area 5 of the unity program. There are, however, 2 issues that arose from using SolidWorks. The conversion of file-types causes the model to become less defined in terms of sharp edges and corners as seen below.

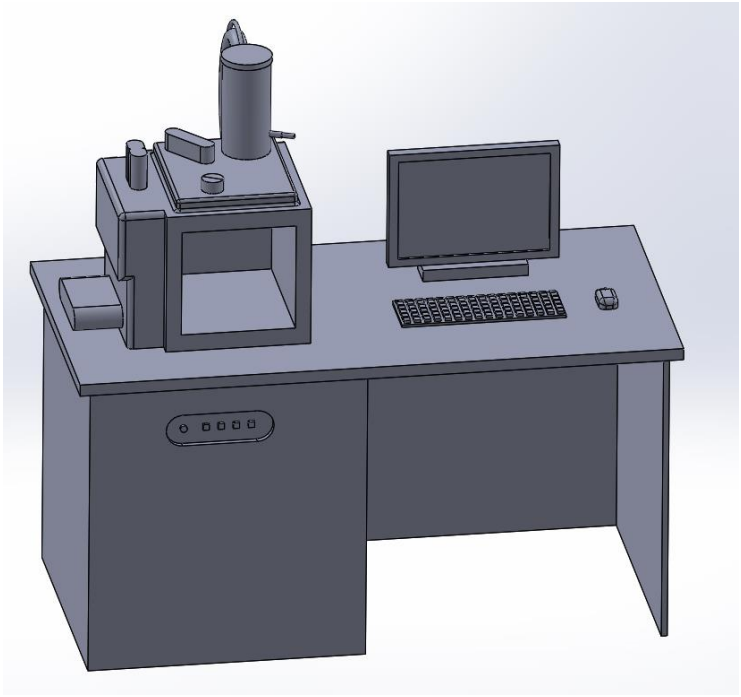

Figure 47: SEM Model in SolidWorks

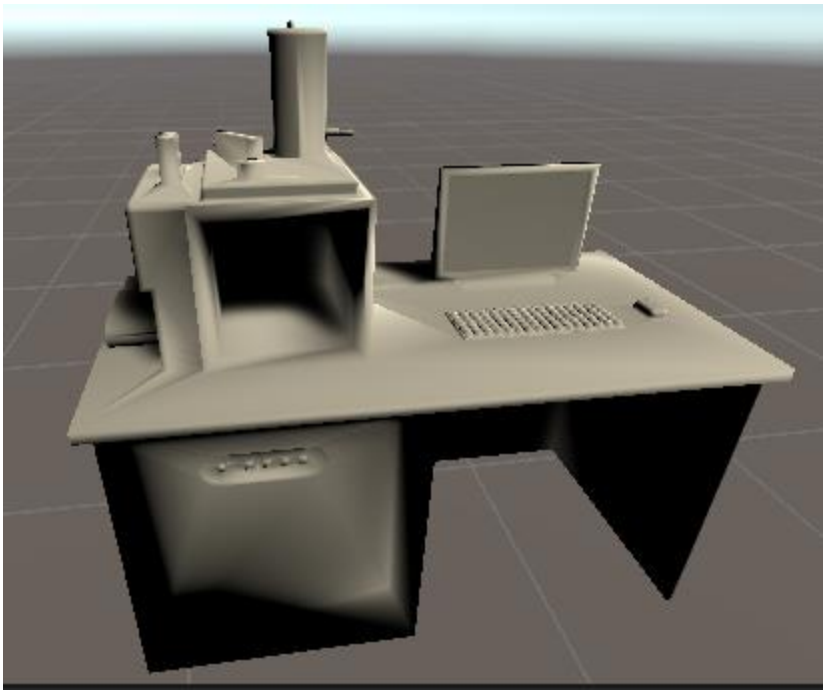

Figure 48: SEM Model imported into the Unity

The scale of the inserted project would also most likely be extremely large, causing the object to either appear missing in the viewing panel or a giant wall covering the panel. The scale and position of the object would need to be adjust to the correct size. The position should be simple. Since the XR Rig should be set at the (0,0,0) position, do so for the new object as well so that it would be easily found once shrunk. For the simulated SEM lab created for this project, the background object created were 3D printers, vent hood, cabinets, tables, and a door. For the core items that the user interacted with, the objects created were an SEM machine connected to a computer, door for the SEM chamber, a vice within the SEM chamber, and a cube sample resting on a table.

To add realism and color to the unity project, textures should be added to the objects inside of the simulation. There were two methods of doing so. The first method was used

if only simple colors were desired. To create a generic material, right click in the Project panel, go to create, and click on material as seen below.

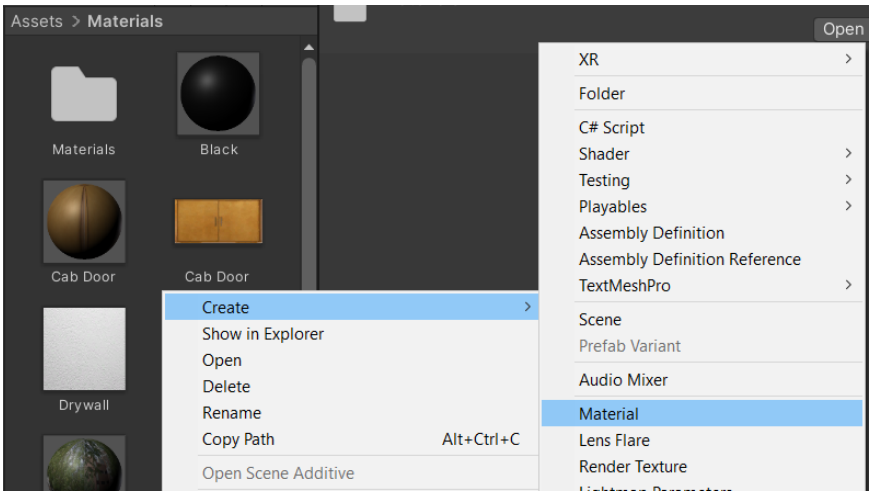

Figure 49: Material Path

Once the material has been created, change the color of the material in the inspector panel. For a more complex blend of colors or direct images, save the image file to the computer and drag the file from the file explorer directly into the project panel.

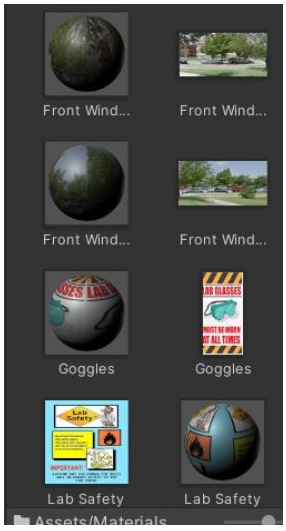

Figure 50: Direct Images and Complex Textures

To integrate either of these into the simulation, click and drag the material/image from the project panel to the object in the simulation. Do note that the image would be applied to all sides of the object. The only way to prevent this was by covering it up with another panel.

Figure 51: Image Texture Limitation

The room and added objects could all be textured to create the desired environment that allows for better immersion in the program when conducting the virtual experiment.

Figure 52: Fully Modeled and Textured Virtual Room

Section 2.6: Colliders and Jelly Physics

Regardless of the type of object added to the simulation, a certain setting was needed for the object physics to work correctly. Realism was an important aspect of this project since the simulation was meant to act as a virtual laboratory. In Velem's Intro to VR Video [1], he discussed different available options that would add realism to the simulation. If the object was meant to be physically interactable with another object, then a collider must be connected to an object. A collider stops objects from going through one another. Colliders have many types and since this project worked with a cubed sample, a box collider was used. To add a box collider to an object, highlight the object in the Hierarchy. With the object highlighted, click on "add component" in the inspector panel and type in box collider. Click on it and it would be added to the object. Since the sample in the program initially rests on a table, the table was given a collider as well.

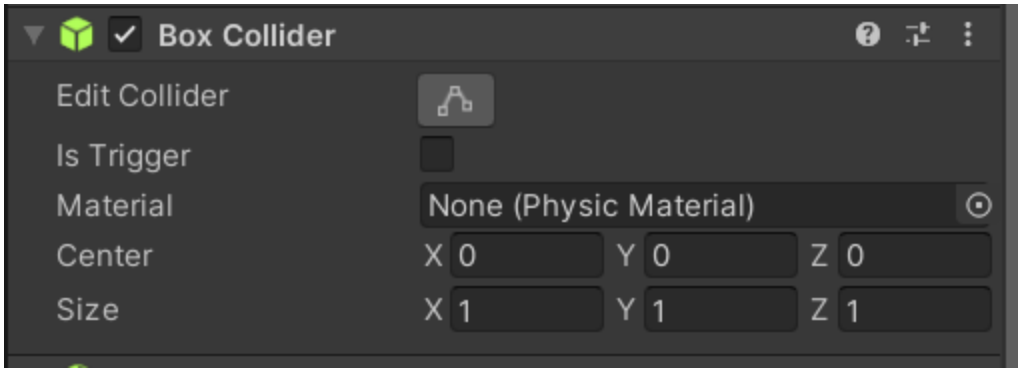

Figure 53: Box Collider Component

If simple gravity was needed for an object, another component would need to be added. Highlighting the object and clicking add component in the inspector panel again, add a Rigidbody component. This component adds mass, drag, and gravity to an object. It was important to have an object with a Box Collider underneath these objects or else when the program was started, the object with the Rigidbody would fall out of the world, becoming unrecoverable unless the program was restarted.

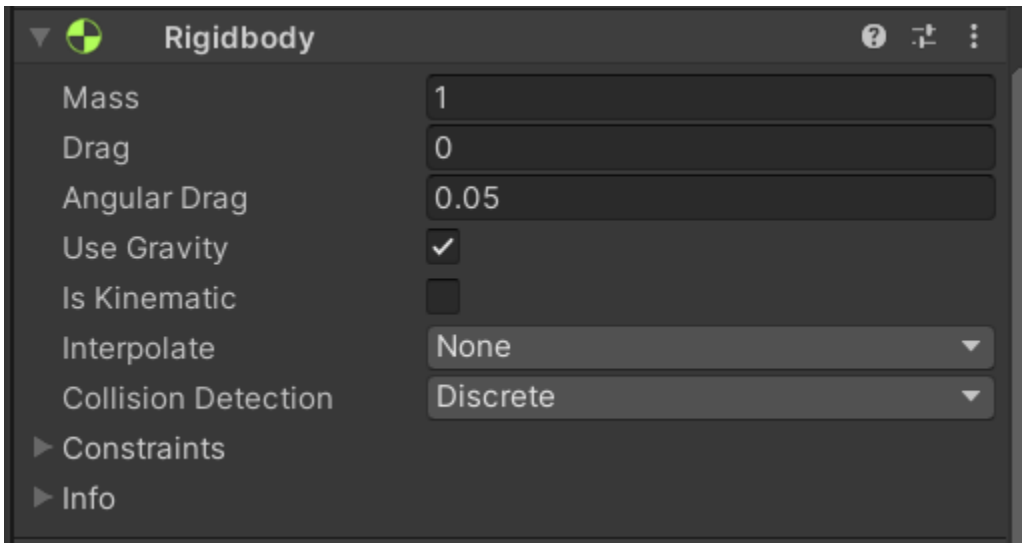

Figure 54: Rigid-Body Component

Since the sample in the program would be compressed in a vice within the SEM machine, a Jelly mechanic was added to the sample object so that texture deformation would look more realistic. A video by Unity City titled “How to make Jelly Mech in unity || Softbody(tutorial)” [5] taught the necessary information for integrating jelly mechanics. For the most part, jelly physics consists of a code that controls the motion of the objects in a way that makes it seem jelly-like. The script that was used in this program can be seen in Appendix B [5]. The main components of the code include the jelly mesh and the jelly physics. The section that creates a jelly mesh on the object it applies to can be seen below.

```
void Start()
{
    OriginalMesh = GetComponent<MeshFilter>().sharedMesh;
    MeshClone = Instantiate(OriginalMesh);
    GetComponent<MeshFilter>().sharedMesh = MeshClone;
    mrenderer = GetComponent<MeshRenderer>();

    jv = new JellyVertex[MeshClone.vertices.Length];
    for (int i = 0; i < MeshClone.vertices.Length; i++)
        jv[i] = new JellyVertex(i, transform.TransformPoint(MeshClone.vertices[i]));
}
```

Figure 55: Jelly Mesh Section of Appendix B Code [5]

The script grabs the original mesh of the applied object and replaces the static mesh with a jelly mesh that was altered in the jelly physics section of the code.

```
28      // Update is called once per frame
29      void FixedUpdate()
30      {
31          vertexArray = OriginalMesh.vertices;
32          for(int i = 0; i < jv.Length; i++)
33          {
34              Vector3 target = transform.TransformPoint(vertexArray[jv[i].ID]);
35              float intesity = (1 - (mrenderer.bounds.max.x - target.x) / mrenderer.bounds.size.x) * Intensity;
36              //jv[i].Shake(target, Mass, stifness, dampining);
37              target = transform.InverseTransformPoint(jv[i].position);
38              vertexArray[jv[i].ID] = Vector3.Lerp(vertexArray[jv[i].ID], target, intesity);
39          }
40          MeshClone.vertices = vertexArray;
41      }
```

Figure 56: Jelly Physics Section of Appendix B Code

This section of the code controls the transformation of the various vertexes within the mesh of the objects and moves them within a bounded space so that the object could vibrate and deform with a jelly-like quality. Once the code was complete, the last step was to drag the code from the assets to the inspector panel of the object that would utilize jelly physics.

Figure 57: Jelly Properties in Inspector Panel

Section 2.7: Object Interaction

Within the program, objects had physics but could not be interacted with by the user. Object interaction was taught in Valem's part 5 episode [6]. To make an object grabbable, the object in question must first be selected in the hierarchy and the XR Grab Interaction component added in the inspector. This makes the object interactable with the XR rig connected to the VR gear.

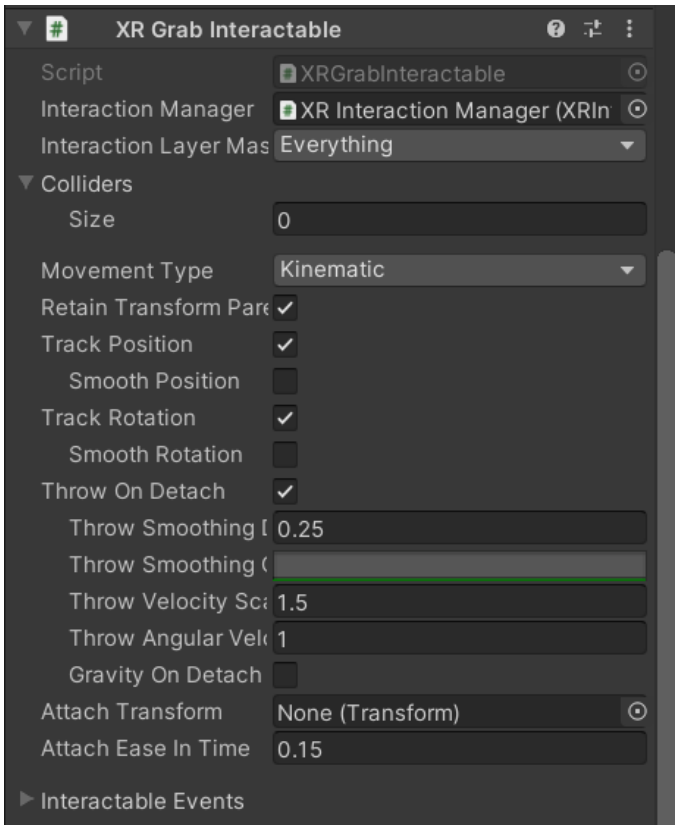

Figure 58: XR Grab Interactable Component

Section 2.8: Hinged Objects

As mentioned before, this project involves interacting with an SEM machine. To make the experience as realistic as possible, an openable cover plate for the SEM machine was included in the program. Valem's Part 7 Introduction to VR video taught the basics of creating hinges [7]. To create a working door, the first part was to create a door with an interactable hand for the already modelled SEM machine. The door in this project was made from two simple rectangles that were scaled and modified to look like a door. The door itself was assembled within the SEM machine in the hierarchy and the handle assembled within the door.

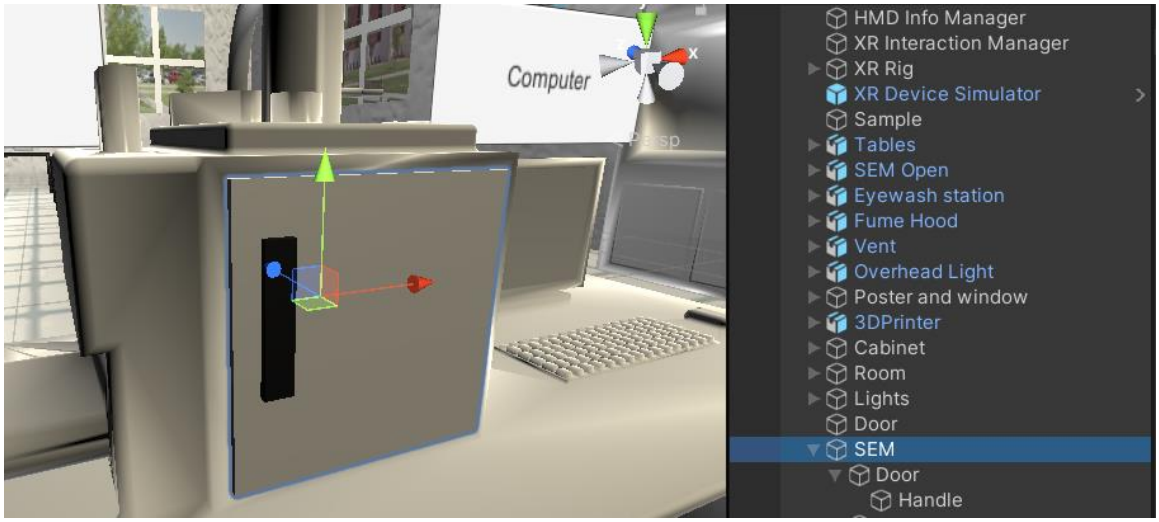

Figure 59: SEM Handled Door Assembly

Once the assembly has been created, the components of the inspection panel for these objects were modified. All three of these objects were given the box collider component so that they could interact with one another. The door itself had a few more components that were needed to create the hinging physics. In addition to the box collider, the SEM door needed a rigid body component, XR interactable component, and hinge joint component in the inspector panel. The rigid body component gave the door rigid body properties that would allow for it to swing and move more naturally. The XR interactable component made the door interactable to the user. Since the door had a handle, the interactable portion of the assembly had to be the handle only and not the entire door itself. To ensure the distinction in the program, the XR Grab Interactable component was modified. Under the colliders dropdown of the component, the handle box collider was dragged into the element 0 line. This was to ensure that the handle was the only thing that could be interacted with regarding the door.

Figure 60: XR Interactable Component of SEM Door

To ensure that the assembly had a pivot point between the door and the SEM machine, the Hinge Joint component was necessary. With this component, an anchor could be set to any axis and position relative to the object it was under so that the hinging ability of the object was limited.

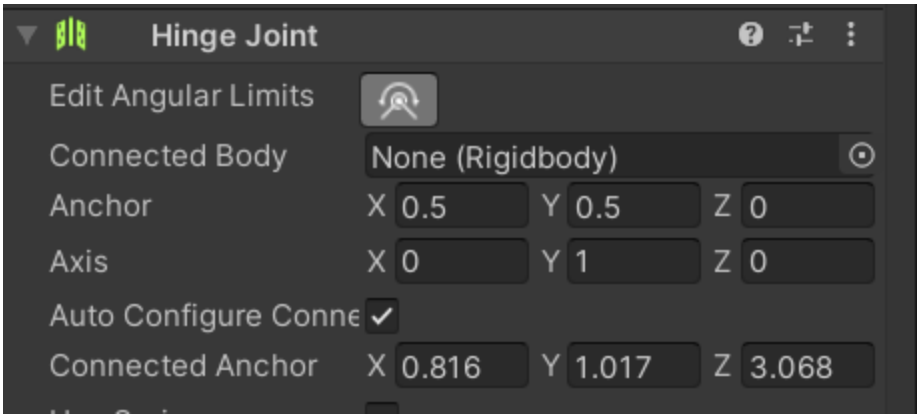

Figure 61: Hinge Joint Component

To prevent uncontrolled hinging of the door, an angular limitation could be set for the hinge as well at the top of the component. This would ensure that the door doesn't rotate through the main assembly and break the immersion.

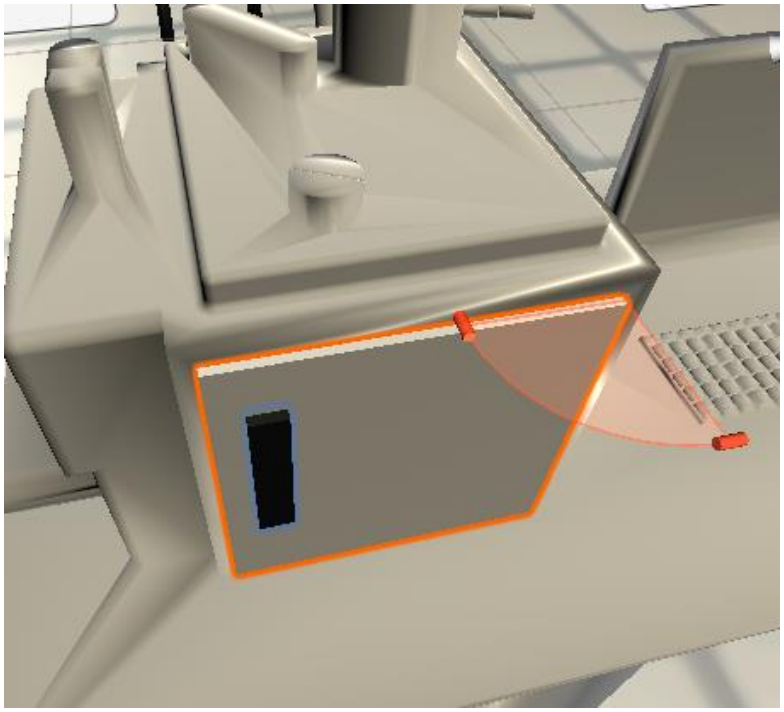

Figure 62: Hinge Joint Angular Limits

Section 2.9: Snap Zones

For this program, a sample cube was interacted with and placed into the SEM machine by the user. In order to ensure consistency with each run as well as consistency with other scenes of this project, a snap zone was created. A snap zone was a space in the VR simulation that would essentially absorb an object that was near it, transform the object so that it was always in the same position, and hold the object suspended in the space until the user removes the object. Valem's 8th Introduction to VR video teaches the basics of creating and utilizing a snap zone [8]. To start, a simple 3D sphere object was created and placed into the simulation hierarchy. For organizational purposes, this object was renamed Sphere Drop Zone. This sphere was moved to the desired location of the snap zone and the sphere collider component was added to the inspector panel. The next component to be added was the XR Socket Interactor. This component essentially makes the sphere automatically grab objects within reach without the need for an XR rig. To make sure that the object instantly grabs object within reach, the Trigger box must be checked in the component of the inspector panel.

Figure 63: XR Socket Interactor Component

At this point of the chapter section, the spherical object was still the default color and could be noticed by the user. To make the space transparent, a new material must be created in the assets. Once created, the material was dragged directly onto the sphere in the main camera panel so that it was applied. To make the material transparent, the rendering mode of the of the material must be set to transparent. In addition, the alpha channel of the albedo color wheel was lowered to increase transparency.

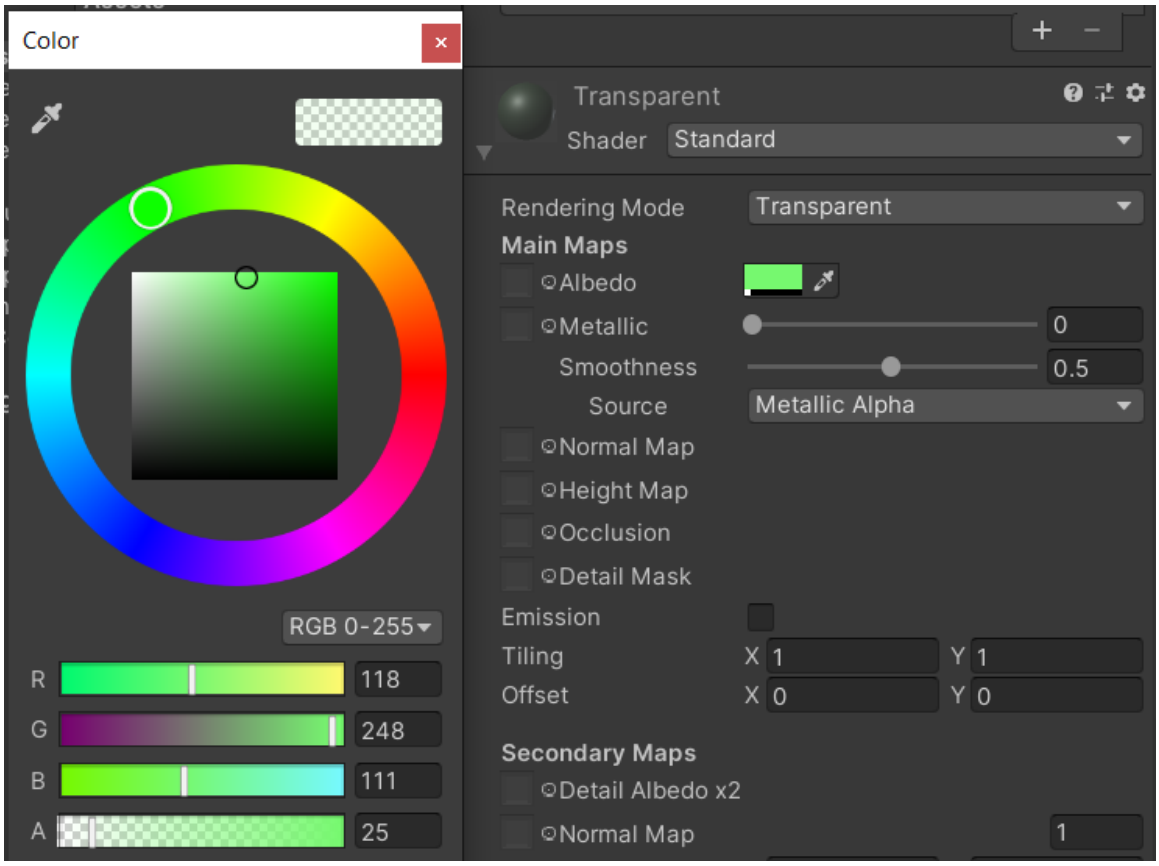

Figure 64: Material Transparency of Component

Once the desired material color has been decided, the snap zone was completed. The objects would be snapped to the center of the zone which allows the creator to move it to the desired location. In this project, the snap zone was placed within the SEM machine within the created vice so that the sample cube would be in the correct orientation when the virtual test was conducted.

Section 2.10: UI and Rays

Other than objects, there were other things that the user could interact with. Various UI, or user interfaces, utilized in the program included sliders and buttons. In Valem's part 6 video, the integration of UI in Unity was taught [9]. In Unity, UI could only be placed on

UI canvas. This was created in the program by right clicking in the hierarchy, going down to UI, and selecting canvas. This created not only the canvas but also an item titled EventSystem which was what allows the user to interact with the UI. One important setting in the inspector of the canvas was the render mode. The default render mode makes it so that the canvas would be following the user's eyes at all times, overlapping everything else in the virtual world rather than staying in one place. To make the canvas and UI elements stay where they were originally placed when the program was run, the render mode was set to World Space and the main camera was dragged from the hierarchy to the event camera slot.

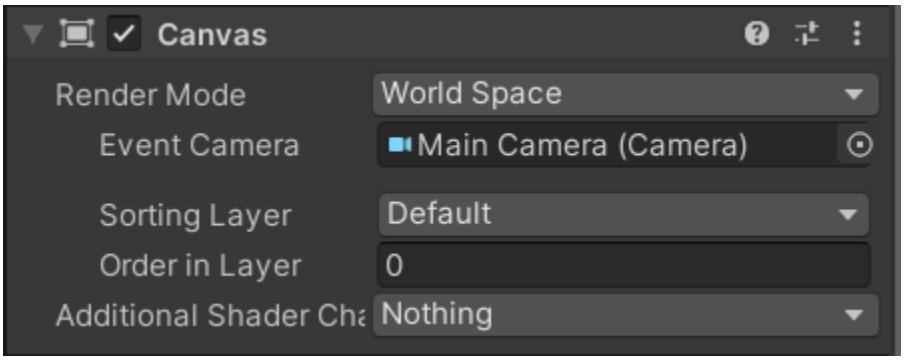

Figure 65: Canvas Component in Inspector

With the canvas settings optimized, it could now be populated with UI. To do so, the canvas object in the hierarchy was right clicked with the UI dropdown expanded. In here were the available UI options, such as buttons, sliders, scroll bars, drop downs, etc., that could be added the canvas. For this project, the two main UI utilized were buttons and sliders. The buttons were primarily used to switch scenes which will be described in Section 2.11. The slider was used to both control the compression of the vise within the SEM, detailed in Section 2.12, and the live graph, expanded upon in Section 2.14.

To interact with these UI, the user does not have to virtually touch the items. In Section 2.3 of the thesis, it was mentioned that there were two red lines protruding from the user's hands when the program was running. These lines were called rays and they show what the user's hands were pointing at and whether the user was hovering over something interactable. By default, the XR rig hands would already have these rays when added to the program. Whenever the ray was red, that means that the user was not pointing at an interactable object. However, if the ray turns white, that means the item was interactable, whether it be an object that could be picked up or a button that could be pushed. These rays could be modified to fit the users need, such as changing the colors of the rays or changing the line length or width of the rays. To modify these, the hand controllers under the XR Rig in the hierarchy must be highlighted. In the inspection panel, the parameters of the rays could be found and altered in the XR Interactor Line Visual Component.

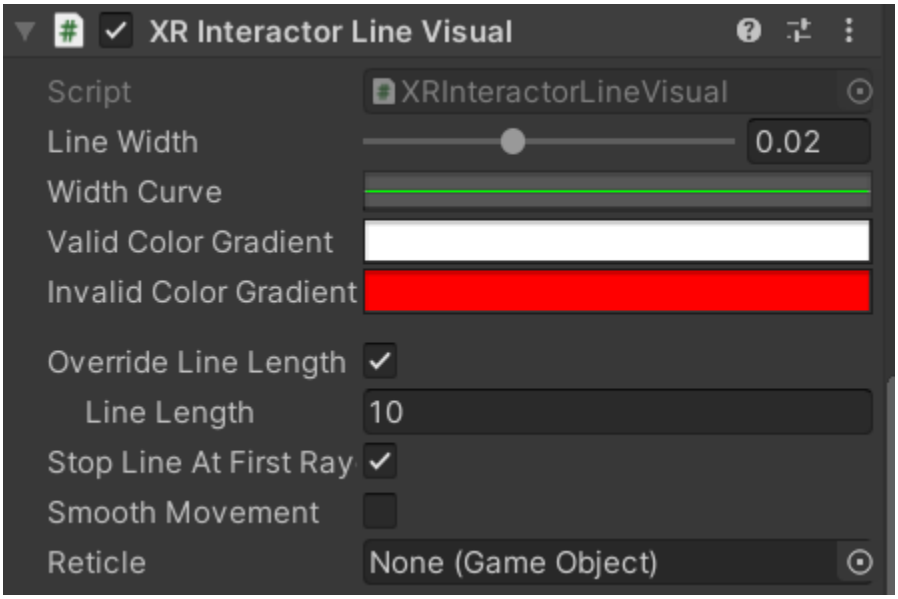

Figure 66: XR Interactor Line Visual

Section 2.11: Changing Scenes

In the program, many separate actions must be taken by the user to go through the virtual lab. The easiest way to manage and control the progression of the user was by implementing scenes which were essentially levels in the simulation that guide the user towards the completion of the simulation. A video titled “START MENU in Unity” posted by Brackeys introduced scenes and methods to implement them [10]. At the start, when a new project was first made, a scene was already made. This was the first virtual space that was worked in. All the scenes that would be used in the overall project would be in a folder within the assets panel. To make a new scene, the right mouse button, create, and new scene were selected in the scenes folder so that everything remained organized.

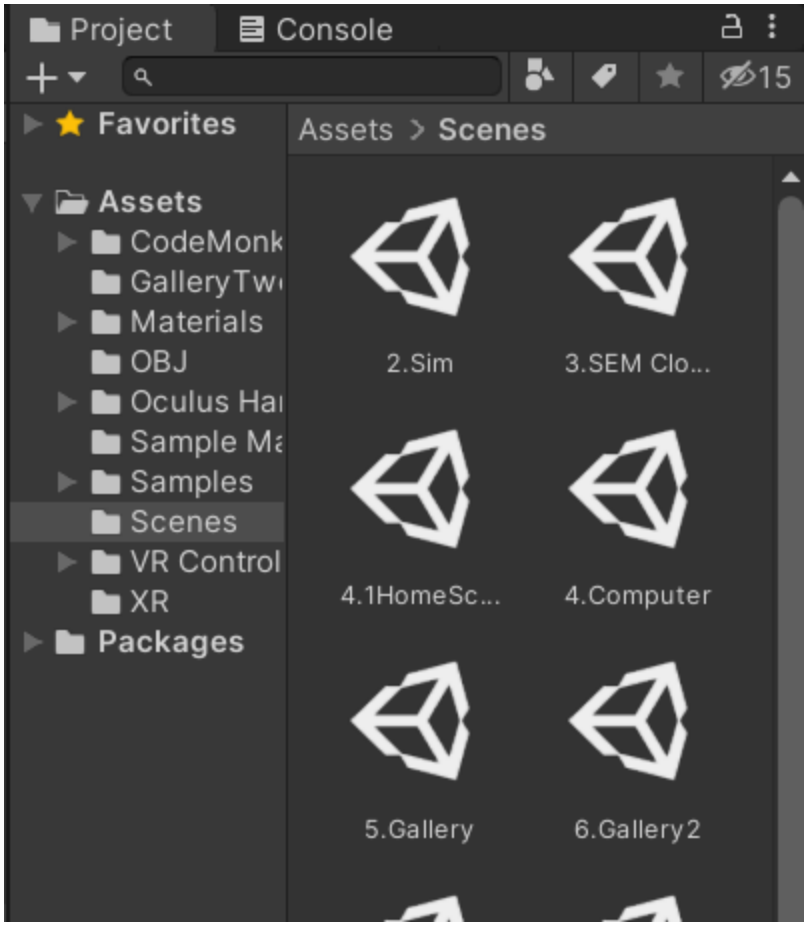

Figure 67: Scenes Folder

Clicking on the new scene would bring the main camera panel to a new, blank virtual world that could be modified similar to the original scene. In this project, the user was inside of the virtual classroom created throughout the entirety of the simulation with certain changes that progressed the lab. Since the surroundings remained the same, the items in the original hierarchy were copied and pasted over to the new scene hierarchy once the new scene hierarchy was cleared of everything.

Multiple scenes could be made with this method but traversing between them while in the program required more work. Previously, in Section 2.10, it was mentioned that the

scenes could be changed by the user through UI buttons. To do so, there were three key components that were needed in the program: a button, a code, and a build. First, the method to creating a UI button was described in the previous section. A UI button was necessary on the scenes that preceded another one so that the user could navigate through the simulation. Of course, multiple buttons could be added to the scene to allow the user choices should that capability be desired. This program utilized a mix between guided path through most of the lab and a more open selection at the end of the simulation once the gallery was reached which will be discussed in more detail in Section 2.15.

The next component, the code, was essentially the set of options that the buttons could be coded to. Every single scene that was desired in the overall program that should or could be reached through a UI button was necessarily added to the code to make traversing through the lab as a user possible. The script that was used in for this program can be viewed in Appendix C [10]. The main and most important aspect of this code was the association of a keyword with a scene name.

```
7       public void PlayGame ()
8       {
9           SceneManager.LoadSceneAsync("2.Sim");
10      }
11
            0 references
12          public void SEMClose ()
13      {
14          SceneManager.LoadSceneAsync("3.SEM Close");
15      }
16
            0 references
17          public void Computer ()
18      {
19          SceneManager.LoadSceneAsync("4.Computer");
20      }
21
```

Figure 68: Keyword and Scene Association

For example, in the snippet of the code above, the public voids PlayGame, SEMClose, and Computer were all codes that were thought up and created with the associated scene in mind. The scene names, or the names given to the scene file in the scenes folder under assets, were also made with the scene in mind such as 2.Sim, 3.SEM Close, and 4.Computer. The scene names were attached to the public voids in the code individually while the public voids were attached to buttons in the inspector of the UI button itself. To attach public voids to the buttons, the script must be created as a new component under the canvas. Once the code was complete, the inspector of the UI button was modified. Clicking on the plus sign of the button component in the inspector, an On Click action was created. To associate the canvas code to the button, the canvas was dragged into the black space

within the On Click section. With this, the code was associated but the specific scene that pushing the button would progress to had to be specified. To do so, the left dropdown in the same section must have had Runtime Only so that the user could interact with it while the program was running. The right dropdown brought up a list of things that included the name of the script made for the scene changes. Under that, the public voids were available to become associated with the button. The selection here sent the user to that scene if the button was pressed.

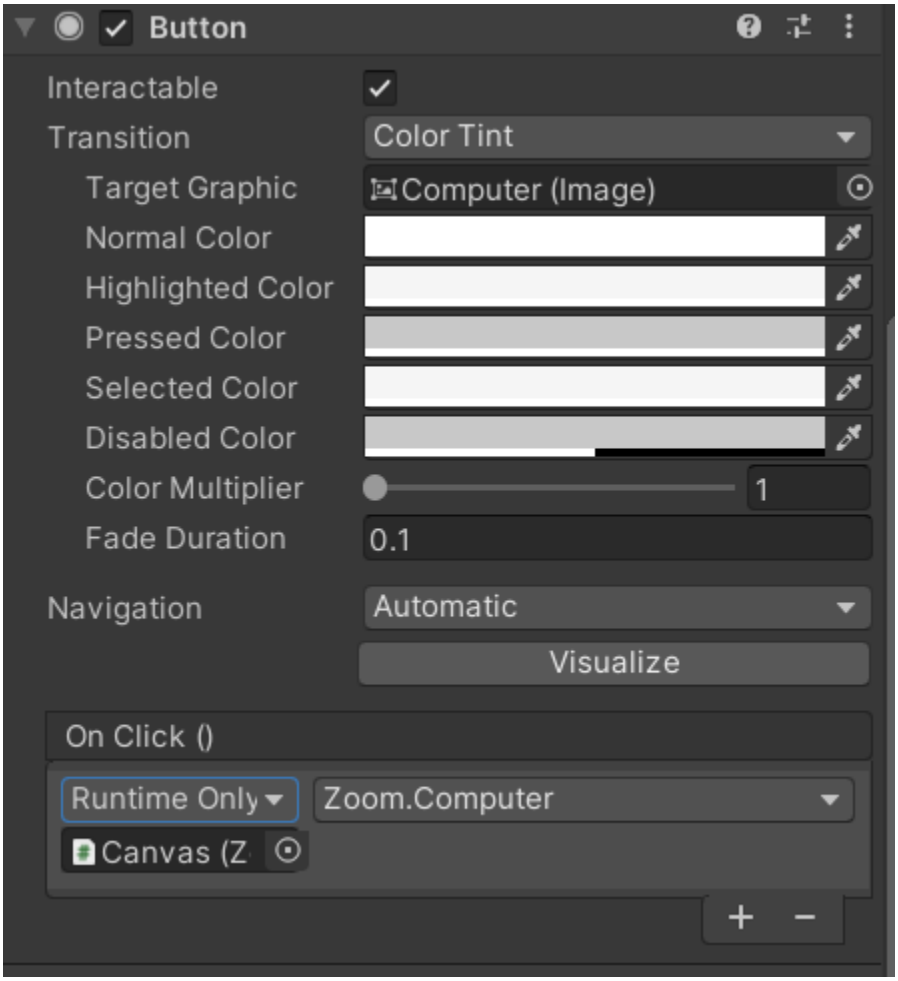

Figure 69: Button Inspector with Script Attachment

While the buttons now have a scene attached to them and an entire script that lays out the scenes of the entire program, the program did not associate all of the scenes as one whole simulation yet. The scenes at this point were still separate entities from one another so while the code does have all of the scene names, aside from the scene with the button, the rest of the scenes were still invalid. In order to group the scenes all together into one single simulation, a build was necessary. A build binds every scene together and allows for interactions between the scenes. To create a build, build setting needed to be opened under file in the top ribbon of the overall editing window. The following window should open:

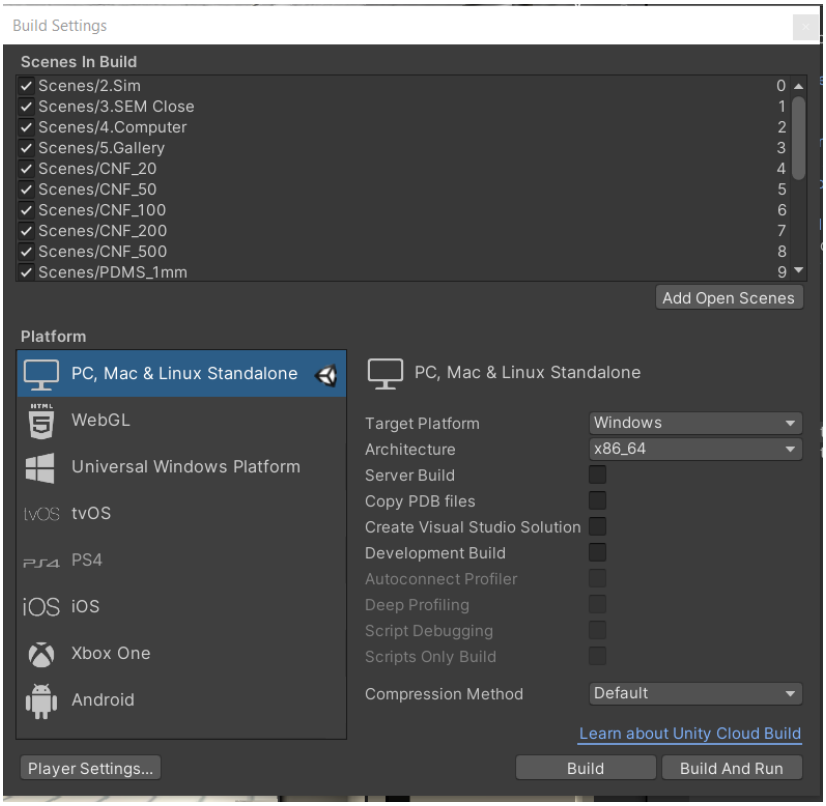

Figure 70: Build Settings

This window encompasses all the scenes involved in the current build and overall simulation. To add a scene to the build, the scene was dragged directly from the scenes

folder into the scenes in build section of the build settings. Once all the scenes were added, the build button on the bottom right side of the window was pressed to confirm the build. Throughout this project, the scene script and build setting were modified whenever a scene was added to ensure everything worked smoothly and as intended.

Section 2.12: Object Translation

In the testing portion of the virtual lab for this project, a sample was placed by the user into a vice that could be controlled through a slider. To make this possible, the translation of one of the clamps had to be directly associated with the UI slider. A forum on StackOverflow titled "Change position of gameobjects with slider" taught how to lerp objects between two points and allowed me to produce the code, as seen in Appendix D, necessary to make this portion of the program work correctly [11]. There were two main sections of the script that make the object translation via slider work as intended: the coordinate parameters and the slider listener. The coordinate parameters allowed the user to set two three-dimensional coordinates that the object it was applied to could move between. This ensures that the objects would stay within the bounds and move with precision according to the creator's wishes.

```
public class ChangePositionWithSlider : MonoBehaviour
    {
        public Slider slider;
        public Transform objectTransform;
        private Vector3 position1 = new Vector3(0.58f, 0.77f, 3.275f);
        private Vector3 position2 = new Vector3(0.64f, 0.77f, 3.275f);
```

Figure 71: Coordinate Parameter Section of Appendix D Code

The second section of the script, the slider listener, updates the position of the object the code was applied to in real-time according to the state of the slider throughout the duration of the scene.

```
            slider.onValueChanged.AddListener(UpdatePosition);
        }

        public void UpdatePosition(float value)
        {
            Vector3 newPosition = Vector3.Lerp(position1, position2,
value);
            objectTransform.position = newPosition;
        }
    }
```

Figure 72: Slider Listener Section of Appendix D Code

Once the script was completed as a component of the desired object's inspector, the last thing that was needed was to drag the desired slider from the assets to the slider box in the script component which would then allow the user to move the object with the slider when the program was running.

Another visual feature that was added to this scene was a percentage that would indicate the total compression of the sample that would change as the user interacted with the UI slider while also moving adjacent with the slider. The first step was to create a text object in the assets panel and create two scripts in the inspector panel. A video by Alexander Zotov taught how to display a percentage value associated with a UI slider. Through this resource, the first script was created, as seen in Appendix E, that would implement this feature into the program [12]. The second script, as seen in Appendix F, was very similar to the script used to translate an object with a UI slider in that it included coordinate parameters and had a listener that updated with the slider [11].

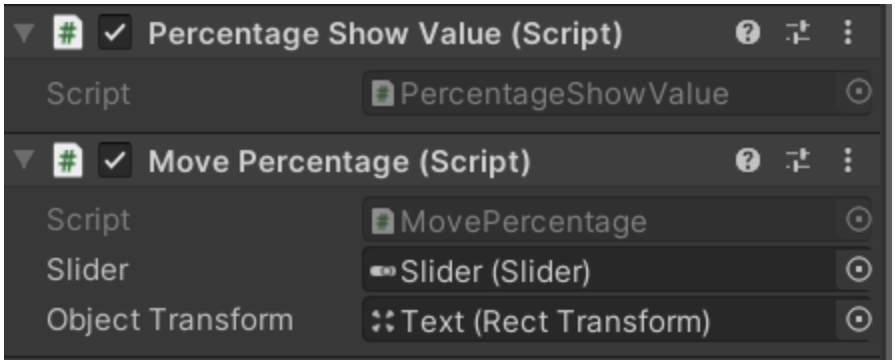

Figure 73: Percentage Display and Movement Script in Inspector

In this program, the slider was connected to the translation of the left vice clamp and the percentage text as well as the value of the percentage text.

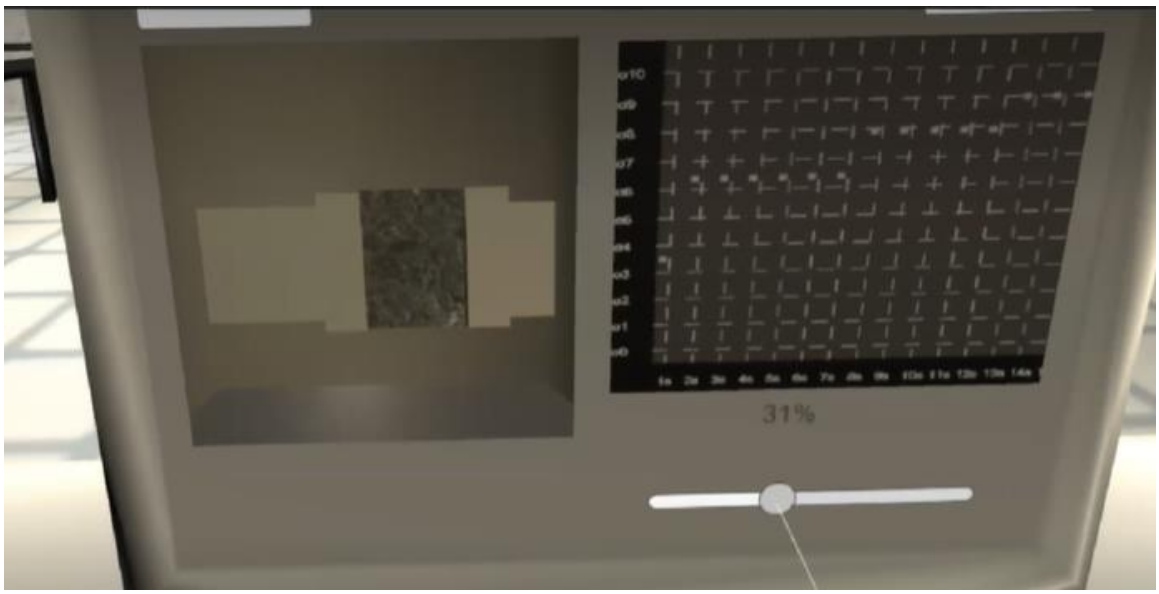

Figure 74: Object Translations in Virtual Lab

Section 2.13: Live Graph

One of the key components of the virtual SEM program was the live-updating graph that displays the theoretical change in conductivity of the sample as the sample was compressed and the pores within the sample start closing and contacting one another. The

main steps that were taken to implement this component into the program was to create the graph panel, create axis separators, create axis labels, connect the slider, and allow for the graph to update real-time. Each of these components consisted of a single code that encompassed all aspects of the graph used in the program which can be seen in Appendix G [13] [14] [15] [16].

The main resource that provided guidance on graph creation in Unity were a series of videos uploaded onto YouTube by the User Code Monkey. In the first video titled "Unity Tutorial – Create a Graph", the basics of creating a static graph and graph panel were taught [13]. To start, the visual components of the graph were made. The first step was to ensure that there was a canvas in place of where the graph was to be placed. Once the canvas was created and oriented, an empty game object was added under it in the hierarchy panel and named window graph. Underneath the window graph, another empty game object named background was created. A component was added in the inspector of the background game object and changed to a generic color. Another game object was then added under the window graph again and named graph container. Under the graph container, another background was added. The sizes of the two backgrounds were changed as desired with the window graph background always being slightly larger than the graph container background. In the graph container inspector panel, the anchor for the object was placed in the bottom left corner so that the graph has a clear starting reference point for later.

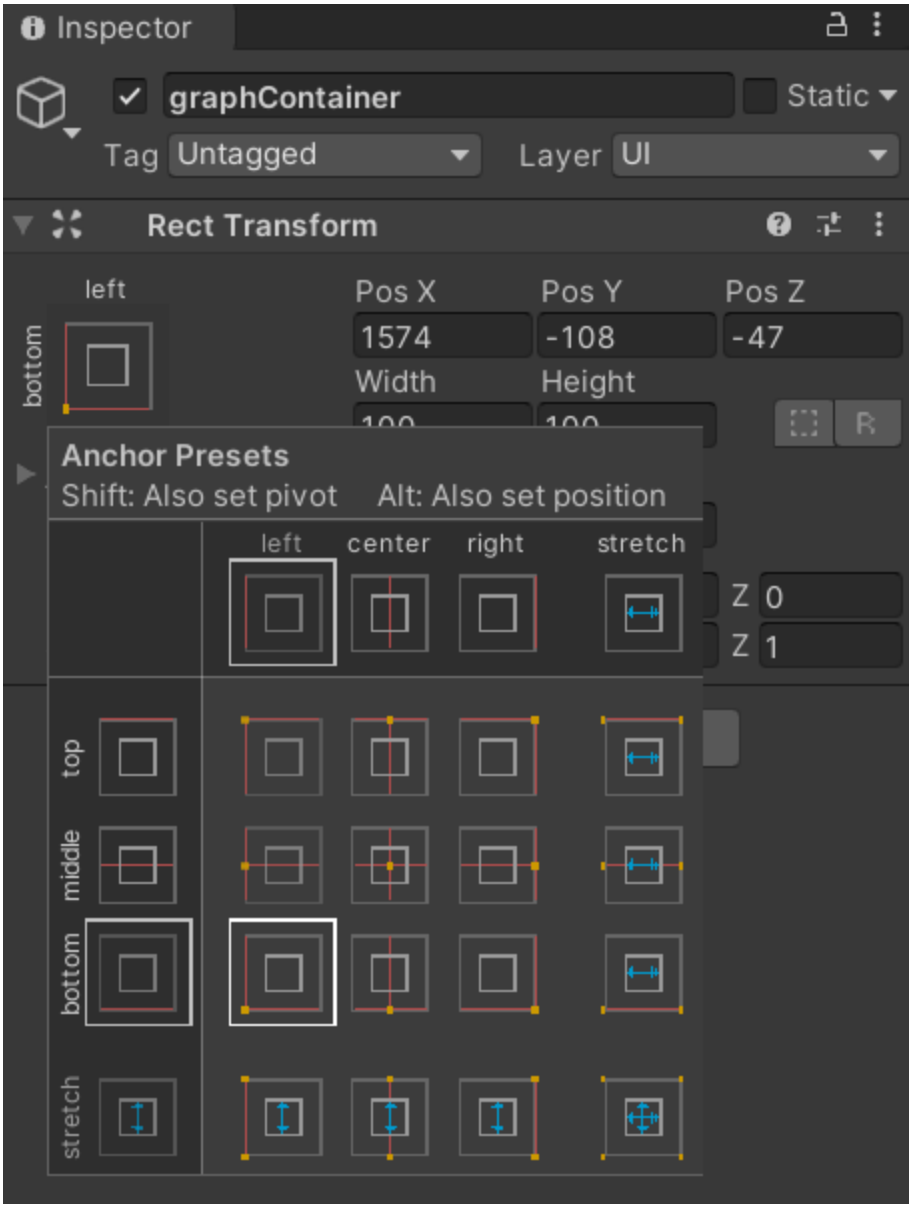

Figure 75: Graph Container Anchor Settings in Inspector

The initial visual aspects of the graph were complete at this point. The next stage was to create a script that would plot the points desired to create a basic static graph that would be modified to be dynamic later. In the window graph game object, a script named window graph was created. The main components of the script regarding the initial setup of the graph include creating points and setting boundaries. To create points, a variable was

needed for each point that was desired on the graph. For this project, fifteen points were to be on the graph to represent data so that many variables were created as seen below.

```
private int N1=0;
private int N2=0;
private int N3=0;
private int N4=0;
private int N5=0;
private int N6=0;
private int N7=0;
private int N8=0;
private int N9=0;
private int N10=0;
private int N11=0;
private int N12=0;
private int N13=0;
private int N14=0;
private int N15=0;
private int NN=0;
private int temp=2;
```

Figure 76: Point Variables from Code in Appendix G

To modify the values of the points and minimize clutter in the code, a list was created to encompass all the points.

```
List<int> valueList = new List<int>() { N1, N2, N3, N4, N5, N6, N7, N8,
N9, N10, N11, N12, N13, N14, N15 };
```

Figure 77: Graphical Point List from Code in Appendix G

Though the points now exist, they must be placed on the graph. To do so, the points were constrained within the bounds of the graph window and constantly created and destroyed so that the graph could properly update the changes to the point value.

```
        GameObject lastCircleGameObject = null;
        for (int i = 0; i < valueList.Count; i++) {
            float xPosition = xSize + i * xSize;
            float yPosition = (valueList[i] / yMaximum) * graphHeight;
            GameObject circleGameObject = CreateCircle(new Vector2(xPosition,
yPosition));
      /*      if (lastCircleGameObject != null) {
                CreateDotConnection(lastCircleGameObject.GetComponent<RectTransfo
rm>().anchoredPosition,
circleGameObject.GetComponent<RectTransform>().anchoredPosition);
            }*/
            lastCircleGameObject = circleGameObject;
            //====================== delete old circles
            Destroy(circleGameObject, 0.1f);
            //======================
```

Figure 78: Point Creation and Destruction from Code in Appendix G

To bound the points, the parameters of the graph window must be set in the script as well by using the previously set anchor point.

```
    private GameObject CreateCircle(Vector2 anchoredPosition) {
        GameObject gameObject = new GameObject("circle", typeof(Image));
        gameObject.transform.SetParent(graphContainer, false);
        gameObject.GetComponent<Image>().sprite = circleSprite;
        RectTransform rectTransform = gameObject.GetComponent<RectTransform>();
        rectTransform.anchoredPosition = anchoredPosition;
        rectTransform.sizeDelta = new Vector2(11, 11);
        rectTransform.anchorMin = new Vector2(0, 0);
        rectTransform.anchorMax = new Vector2(0, 0);
        return gameObject;
```

Figure 79: Anchor Point from Code in Appendix G

```
float graphHeight = graphContainer.sizeDelta.y;
float yMaximum = 20f;//100
float xSize = 40f;//50
```

Figure 80: Graph Container Boundary from Code in Appendix G

Once the main graph panel was complete, the next step was to create axis separators and axis labels to help organize the graph and allow for more precision in the visual aid. A video by Code Monkey titled “Unity Tutorial – Create a Graph: Axis Separators” provided guidance on how to add axis values and implement the separators into the graph created [14]. In a video uploaded by Code Monkey titled “Unity Tutorial – Create a Graph: Axis Labels”, the method of adding the labels was relayed [15]. For the axis values, the first step was to create the first instance/template of the label on the graph as a reference. Out of the script and in the hierarchy, two empty game objects were created under the graph container and named LabelTemplateX and LabelTemplateY. A text component was added to both of them, and the game objects were anchored to the bottom left side of the graph container. In the script, the number of desired labels and their positions for both the X and Y axis values must be called and created. For clarity on the weight of the values, the labels were also scripted into the code.

```
RectTransform labelX = Instantiate(labelTemplateX);
labelX.SetParent(graphContainer, false);
labelX.gameObject.SetActive(true);
labelX.anchoredPosition = new Vector2(xPosition+50, -60f);
labelX.GetComponent<Text>().text = getAxisLabelX(i);
```

Figure 81: X Axis Values and Positions from Code in Appendix G

```
            RectTransform labelY = Instantiate(labelTemplateY);
            labelY.SetParent(graphContainer, false);
            labelY.gameObject.SetActive(true);
            float normalizedValue = i * 1f / separatorCount;
            labelY.anchoredPosition = new Vector2(22f, normalizedValue *
graphHeight * 5);
            labelY.GetComponent<Text>().text = getAxisLabelY(normalizedValue *10
);//* yMaximum);
```

Figure 82: Y Axis Values and Positions from Code in Appendix G

```
        ShowGraph(valueList, (int _i) => (_i+1) + "s", (float _f) => "o" +
Mathf.RoundToInt(_f));
        // slider.onValueChanged.AddListener(UpdateValue);
```

Figure 83: Label Text from Code in Appendix G

Starting the program at this point showed that both axes had labels and values spaced evenly along both directions. To add separators that would aide with reading the graph with more precision, the script was modified similarly to how the axis values were created.

```
RectTransform dashX = Instantiate(dashTemplateX);
dashX.SetParent(graphContainer, false);
dashX.gameObject.SetActive(true);
dashX.anchoredPosition = new Vector2(xPosition+10, 300f);
```

Figure 84: X Direction Separators from Code in Appendix G

```
            RectTransform dashY = Instantiate(dashTemplateY);
            dashY.SetParent(graphContainer, false);
            dashY.gameObject.SetActive(true);
            dashY.anchoredPosition = new Vector2(330f, normalizedValue *
graphHeight * 5 + 40);
```

Figure 85: Y Direction Separators from Code in Appendix G

Just as the axis labels needed a template, the axis separators needed one as well. Back in the project editor, two new empty game objects were created under the graph container in the hierarchy and named DashTemplateX and DashTemplateY with an image component added to it. In Microsoft Paint, half of the default workspace was deleted and the image converted into a png. Once done, the image file was then dragged into unity and turned into a texture type of sprite. This was the sprite that was used in the image component. In the inspector, the image component type was set to tiled so that the image made a series of repeated and linked image giving off the dashed line appearance. The

dashed line was then dragged to fit across the graph container, either x or y direction depending on which dash template was being worked on. The vertical line was moved to the left-most side of the graph container while the horizontal line was moved to the bottom of the graph container. The script would again create a series of copies that would fill the graph container with set spaces between each instance.

With the graph panel, axis separators, and axis labels done, the foundations of the graph were complete. To connect the graph to the slider bar so that the graph was UI friendly, techniques learned from connecting the translation of an object to the slider was used [11]. Although not the same code type was used for moving the graph points, a variation of it was created instead. Previously, in Section 2.12: Object Translation, the object attached to the script was lerped, or bounded, between two coordinate points and was only allowed to translate between the two. For the graph, only the first plotted point needs to be moved with the slider. Rather than lerping the graphical point, the direct value of the slider was associated with the y-direction translation of the point and was parameterized to fit within the graph panel by limiting the max and min value of the slider value.

```
    public void Update()//Move dot with slider
        {
            N1 = Mathf.RoundToInt(slider.value * 140);
            NN = N1;
            List<int> valueList = new List<int>() { N1, N2, N3, N4, N5,
N6, N7, N8, N9, N10, N11, N12, N13, N14, N15 };
            ShowGraph(valueList, (int _i) => (_i+1) + "s", (float _f) =>
"σ" + Mathf.RoundToInt(_f));
```

Figure 86: Slider Integration Section of Graph Code in Appendix G

Once the slider value has been associated with the first graphical point, the entire list was updated with the new value.

The graph and the slider move in sync at this stage of the script, but the points of the graph all move in unison rather than with respect to time. Moving the slider should create wave patterns according to the interaction of the slider since the values were snapshotted at a point in time and transferred to the next point. To implement this idea into the graph, a forum that discussed the ways to make the script timed or delayed in terms of execution was investigated [16]. To make an action or line of script delay for a set time, an action called WaitForSeconds could be used before the script line. This would cause the line to wait for as many seconds as inputted before executing the command. For this project, each point after the first graphical point waited 2 seconds before taking on the value of the point before it, thus making the graph more accurate and practical in its functionality.

```
IEnumerator wait()
{
    yield return new WaitForSeconds(1f);
    N2=NN;
    temp = temp+1;
}

IEnumerator waiter()
{

        yield return new WaitForSeconds(2f);
        N3=N2;
        yield return new WaitForSeconds(2f);
        N4=N3;
        yield return new WaitForSeconds(2f);
        N5=N4;
        yield return new WaitForSeconds(2f);
        N6=N5;
        yield return new WaitForSeconds(2f);
        N7=N6;
        yield return new WaitForSeconds(2f);
        N8=N7;
        yield return new WaitForSeconds(2f);
        N9=N8;
        yield return new WaitForSeconds(2f);
        N10=N9;
        yield return new WaitForSeconds(2f);
        N11=N10;
        yield return new WaitForSeconds(2f);
        N12=N11;
        yield return new WaitForSeconds(2f);
        N13=N12;
        yield return new WaitForSeconds(2f);
        N14=N13;
        yield return new WaitForSeconds(2f);
        N15=N14;
}

IEnumerator pause()
{
    yield return new WaitForSeconds(1f);
    temp = temp+1;
}
```

Figure 87: Time Dependence Section of Code in Appendix G

Section 2.14: Remote Live Feed

With the connection of the slider bar on the computer to the transformation of the vice and live graph, the virtual SEM functionality with UI interactive portion of the program was complete. Since the compression of the sample was in the enclosed SEM chamber, the user wouldn't be able to view the vice in action when using the SEM program. The graph and percentage display would be moving in accordance with the slider bar moving but a live view of what was happening in the chamber was missing. To improve the live interaction more, a camera texture or render texture was created on the computer

canvas. A render texture was similar to a normal texture except the image displayed was the live feed from a separate camera. This texture type was useful for any displays that needed to be remote. An online unity user manual provided guidance on how to achieve this texture [17]. To create a render texture, the first step was to add a render texture into the assets by right clicking, going to create, and clicking on render texture. In the hierarchy, a new camera that would capture the live feed was created. The next step was to assign the new render texture to the camera by adding it to the target texture section of the camera's inspector panel.

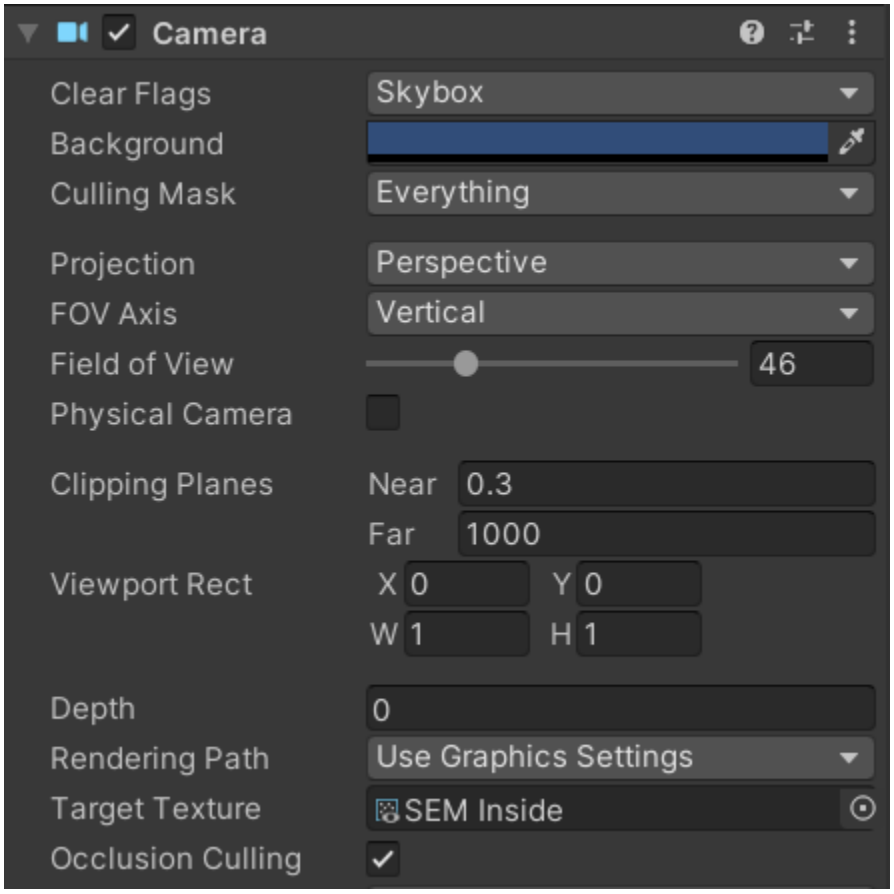

Figure 88: Target Texture in Camera Inspector Panel

To create the screen that would display this new live feed texture, a normal plane was added and positioned in the scene. The render texture was then dragged onto the plane similar to a normal texture. The panel would then display what the camera was pointed to in real time whenever the program was started.

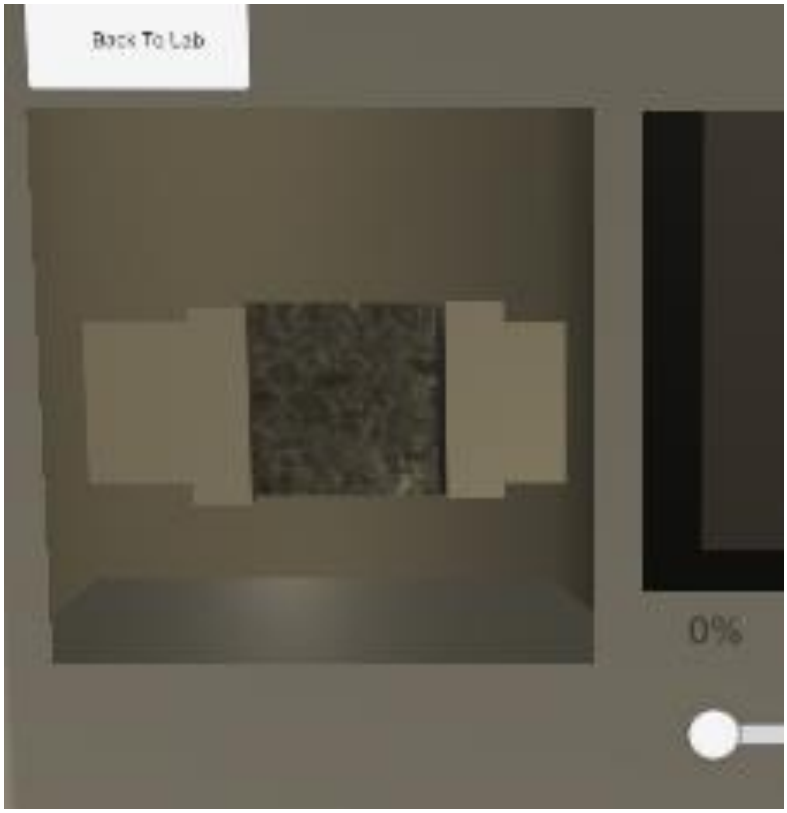

Figure 89: Completed Render Texture

Section 2.15: Gallery and Zoom

Completing the graph, the SEM application scene was complete. Since SEM produce images when used in live labs, a gallery was created to show some real SEM images of the lab this project was based on. This was done using many scenes and buttons that would change the virtual screen to whichever image was clicked. To further enhance the Gallery capabilities, a zoom feature was added the detail view of each image. Similar

to the gallery, a magnifying glass-shaped icon was placed on the detailed view that would change the screen to essentially the same screen but with the image zoomed in. On the zoomed scenes, the close-up images could be panned around by the user. In order to create a close-up image that could be moved around, a new script was required. By learning from a video uploaded by Jason Weimann titled "Unity3D – How to Zoom an Image / Create a zoomable pannable sprite", the script in Appendix H was created [18]. To start, a canvas was required in the hierarchy. Within the canvas, an empty game object with a UI image was created. In the assets, the desired image to become zoomable was added and turned into a sprite by changing the texture type in the image's inspector panel to sprite (2D and UI). The sprite was then added into the source image slot in the inspector of the UI image. Now that the sprite has become the image on the canvas, the size of it was enlarged to be zoomed in. The issue that needed to be fixed was that the image was too large and the viewable bounds needed to be restricted. In the game object, a Scroll Rect component was added to the inspector as well as a Rect Mask 2D. At this point, the image was still enlarged but the only part of the image that could be seen was within the Rect Mask 2D box created which could be resized to the desired size. In the game object inspector, the image underneath the game object must be dragged to the content of the scroll rect. In the Image inspector, a new script was created which could again be seen in Appendix H. The main purpose of the script was to allow the user to control the zoom of the image through the use of a mouse wheel. This was geared more towards if the user does not have a VR headset and had to use the mock XR rig created at the start of the project. The script itself enables the use of a mouse wheel and also limits the zoom scale so that the image remains clear or

within the creator's desired size. These parameters script was also made to allow for parameter changes within the inspector.

```
public class UI_ZOOM_IMAGE : MonoBehaviour, IScrollHandler
{
    private Vector3 initialScale;

    [SerializeField]
    private float zoomSpeed = 1f;
    [SerializeField]
    private float maxZoom = 100f;

    private void Awake()
    {
        initialScale = transform.localScale;
    }

    public void OnScroll(PointerEventData eventData)
    {
        var delta = Vector3.one * (eventData.scrollDelta.y * zoomSpeed);
        var desiredScale = transform.localScale + delta;

        desiredScale = ClampDesiredScale(desiredScale);

        transform.localScale = desiredScale;
    }
```

Figure 90: Mouse Scroll Section of Code in Appendix H

```
    private Vector3 ClampDesiredScale(Vector3 desiredScale)
    {
        desiredScale = Vector3.Max(initialScale, desiredScale);
        desiredScale = Vector3.Min(initialScale * maxZoom, desiredScale);
        return desiredScale;
    }
}
```

Figure 91: Zoom Scaling Section of Code in Appendix H

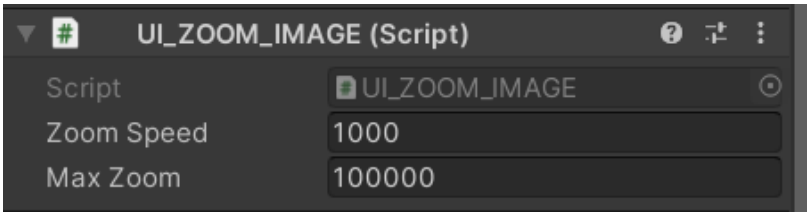

Figure 92: Zoom Parameter Settings in Inspector

CHAPTER 3: SIMULATION TRIALS

Section 3.1: Trial and Feedback

Once the program was complete and the user was able to run through the entire lab as intended with no major setbacks nor crashes, the simulation trials were conducted. The total number of volunteers that tested the program was five including Logan Roys, Christopher Billings, Melody Moody, Lane Taylor, and Steven Zhao. When the testers first come into the lab with the equipment setup, they were first given a run through of what to expect. The basic instructions on how the VR headset worked was given as well since many of the volunteers had not experienced virtual reality before. The testers were geared up and immersed into the program. The users would then go through the program and explore as desired while asking any questions they had.

Once the volunteers had reached the end of the program, they were sent a survey to provide feedback on their experience testing the program. The survey consisted of ten questions in total and were answered anonymously. The first nine questions had the format of a nine-point linear scaling so the user may rate the questions based on their experience. The questions that were presented to the user included asking about the difficulty of learning how to use the controls to navigate through the simulation, the difficulty of navigating through the program through the use of UI such as buttons and sliders, the comfort of the experience, the entertainment factor of the simulation, the overall length of time spent running through the simulation, the memorability of the experience, the amount of new knowledge obtainable through the simulation, the difficulty of the content of the

lab, and the usefulness of the simulation in trying times such as during a pandemic. The last question of the survey was open ended and inquired about any constructive criticisms about the experience and elaborations on previous answers.

Section 3.2: Navigation and UI Interaction through the Simulation

Navigating through the entirety of the simulation required knowledge on how to use the VR gear correctly as well as being able to interact with the various UI, such as buttons and slider, to progress through the lab to completion. When the testers were asked about how difficult it was to learn the necessary controls to navigate thought the simulation, all of them responded that the controller and gear were very easy to learn. However, the question regarding the difficulty on navigating through the program using UI was sparser. Two users deemed the UI either very easy or easy to use while two others found that it was neither difficult nor easy. In contrast, the last user felt that it was quite difficult to use the UI.

Two users were constructively critical and shared their troubles and suggestions for improvement. One user responded that "*The lagginess of the SEM menu was a little rough. From what I understand the lag was coming from the plot on the right-hand side. Maybe there is a way to plot this graph to have less lag*". Another user shared similar troubles and wrote "*The main issue was with the very visible lag at times particularly in the analysis slider, this made it very hard to even click on buttons. Perhaps, it would be better to have less elements loaded at once when moving sliders and instead have predetermined notches at say every 10 or 20 percent completion*". Two other users shared their thoughts on the UI

in the program overall. One user stated that "*The Program has a glitchy part but overall, it was good. The Movements are smooth, but the buttons are a little hard to hit*", while another wrote that "*Overall, the experience was very positive, the UI was pretty easy other than one scene which experienced some lag that made navigation a bit more difficult. Other than this one instance, UI was incredibly user-friendly and intuitive*".

Section 3.3: Comfort and Entertainment

When using VR for the first time, some users may experience nausea due to the lack of experience in the virtual world. Depending on the quality of the game, nausea may be more severe if a program was too shaky or inconsistent in movement. Of course, the comfort and the entertainment factor of the simulation was important to keep the students interested in the lab. When asked how comfortable the experience was, one student rated the experience as very comfortable, three students rated the comfort of using the headset to be overall comfortable, and one student rated it uncomfortable. When asked about how entertaining the lab was, two users found the lab to be very entertaining, two users found it to be entertaining, and the last user deemed it not very entertaining.

For the most part, the volunteers felt that the testing environment and simulation were pleasant. One user commented on the VR gear and wrote "*This was my first VR experience, I thought I would get dizzy, but surprisingly not*". Two users commented on the setting with one stating that "*The setting is very well done and the motions are well thought through so that everything is natural feeling which is a strong point for this simulation*". The other user expressed that the "*pictures of the OU campus and typical*

warning signs throughout the VR added a touch of familiarity within the program that was very fun".

Section 3.4: Length and Memorability

When creating the lab, balancing the content of the lab and the length of time needed to complete the simulation was important. Retaining the student's attention long enough was always important. If the simulation was too long, students might get bored and stop focusing on learning and remembering the content. If the simulation was too short, the student might not get adequate experience when compared to the in-person lab. Fortunately, most volunteers viewed the delicate balance of time and impact in a positive light. When asked how the length of time to complete the lab felt, four users thought the lab was just the right amount of time while one user thought it was a little short. When asked about the memorability of the lab, four users felt that it was very memorable while one user deemed it kind of memorable.

Regarding the time spent in the virtual lab, one user commented, "*felt like the length of time in the lab was perfect, not too long to be overwhelming but not so short to be leaving the user feeling like something was missing*". Another user commented on the memorability of the lab, "*The lab was very memorable especially the added aesthetics within the lab*".

Section 3.5: Education and Content

Since this simulation was created with the purpose of educating students from a remote place, obtaining feedback on the education potential and content difficulty were crucial. When asked about how much was learned from the lab, three students stated that

they learned a great amount while one student learned a moderate amount and the last student gained a small amount of knowledge. When inquiring about the difficulty of the content of the lab such as the content and purpose, two students rated the simulation to be very easy. Two other students rated it moderately easy while the last student deemed the lab not too difficult nor too easy.

When asked to expand on their answers, many of the volunteers responded positively in terms of the educational potential of the program. One user wrote that "*The user utilizing the software to virtually perform experiments while a mentor is available to explain subtleties within the experiment and answer questions. I was able to learn more about how these types of materials change resistances as their structures are altered (i.e., compressed) and it was cool to see the images of the materials after the experiment to have a clearer understanding of the microstructures*", while another wrote that "*The concepts were not difficult to understand, and it helped having someone there to answer questions during the experimentation process. This seems like the natural setup if this software were to be used for teaching purposes*". Similarly, it was said that "*Overall I feel this software is a great starting point for someone trying to understand the research process and what data is trying to be collected*". One user even gave a suggestion on how to improve the relay of information and improve the educational experience of the program by commenting that "*It would be cool to view the graphs outside of the testing stage. Maybe something along the lines of example graphs so the user better understands what the live updating graph represents. It would also be nice to have a couple of the SEM pictures have*

different details highlighted or pointed out. These changes would mostly be aimed at a more novice audiences as everything is easily understood from a graduate stand point".

Section 3.6: Reflection and Utilization

Considering all the feedback from the testers of the program, it seems that the overall impression of the simulation was positive. Learning how the controls worked on the VR gear was very easy to all subjects which shows that equipment set-up and instructions to new users shouldn't pose a problem.

The only aspect of the program that caused the most trouble was the UI interaction to navigate through the program. For the users well-versed in VR interactions, the navigation and controls were much easier to use. As the creator of this program, the controls, natural movements, and interactions got easier the more it was used. For those less experienced, the delicate aiming of the arrays and correct pressure on the handheld controller buttons proved to cause some struggle at times. The scene with the SEM testing especially proved to cause issues due to what could be assumed to be the overload of created data. Due to the real-time aspect of the slider bar with the graph, the constantly changing values of the graphical points seem to cause lag and quality deterioration in the program. Even with the lag, most of the testers rated the UI interaction aspect of the program to be between not difficult to very easy.

Besides the UI interaction, most of the other surveyed aspects came out relatively positive as well. A majority of the users felt that the lab experience was mostly comfortable and entertaining. Most of them also felt that the lab lasted just the right amount of time and

was memorable. Education-wise, most of the users felt that they either learned quite a bit or saw potential in new students learning from the program. It wase especially helpful that the content wasn't muddled in the creation of the lab and was still easy to learn and understand from a new user's standpoint. Based on the average of each of the user's responses, the program has potential to become even better but lays a solid foundation in terms of what it currently does and how it could be modified to encompass even more different kinds of labs.

This program was made with creating an opportunity for students who were barred from performing real labs in person due to restrictions such as when the pandemic first occurred. When asked about how useful this research would be in trying times such as the pandemic, it was overwhelmingly determined to be very useful, despite its current downfalls. One student even wrote that "I definitely can see how this software would be beneficial to allow for virtual experimentation and teaching to be done. As a user, I would be able to very realistically perform an experiment and see the results of these experiments. I think this idea could be utilized in conjunction with a lab-style class. A group of students and a lab mentor could meet virtually and many simultaneous experiments could be run with all students being able to directly ask questions to the lab mentor while running the experiments. This could also be used as a demonstration-style software where a teacher could cast the software virtually to a class and perform the experiment while teaching the subject.

CHAPTER 4: CONCLUSION

Section 4.1: Potential

Having a standard program with many components allow for an easier creation of diverse labs. The background of the lab can easily be changed by either finding different 3D models online or created through modelling software such as SolidWorks. Textures could be added by taking pictures of real objects and uploaded or images found off the web. Drawers and doors could easily be created as well as the interactions with them. VR setup was already done which would take out that step. The graph could be altered by changing the axis to the desired value as well as the function the graph follows. An instruction manual with these different components could be compiled so that labs could be tailored according to the instructors wishes.

The developed VR platform can be further modified for the virtual additive manufacturing applications. Multiple additive manufacturing technologies have been developed and employed in Dr. Yingtao Liu's laboratory at the University of Oklahoma including direct ink writing, filament deposition modeling, and selective laser sintering [19] [20] [21] [22] [23] [24] [25] [26].The developed technologies have been used to 3D print composites, nanocomposites, and polymers for biomedical, mechanical, and senor applications. The developed platform can be modified to show the 3D printing process and assist both graduate and undergraduate students to practice 3D printing virtually. The virtual 3D printing can serve as an exercise for future students to understand the fundamentals of additive manufacturing.

Section 4.2: Conclusion

With the COVID-19 pandemic ravaging the world, more efficient ways of providing the necessary education to the students affected. Laboratories specifically have limitations that prevent the hands-on experience of STEM majors hoping to graduate on time with the necessary experience needed for the workforce. The usage of Virtual Reality Simulations are an efficient and effective way for students to safely experience the lab through technology with minor drawbacks in the takeaway compared to real labs. By having a general lab created on the simulation software, instructors would easily be able to recreate their own lab tailored to their way of teaching for the enriching of student experience. The future of education was limitless and potential that it brings the students could change the world for the better.

References

[1] Valem, "Introduction to VR in Unity - Part 1: VR SETUP," Youtube, 8 April 2020. [Online]. Available: https://www.youtube.com/watch?v=gGYtahQjmWQ&list=PLrk7hDwk64-a_gf7mBBduQb3PEBYnG4fU&index=2.

[2] Velem, "How to Make a VR Game WITHOUT a VR Headset," Youtube, 11 December 2020. [Online]. Available: https://www.youtube.com/watch?v=UlqdHrfXppo.

[3] Valem, "Introduction to VR in Unity - PART 2: INPUT and HAND PRESENCE," Youtube, 15 April 2020. [Online]. Available: https://www.youtube.com/watch?v=VdT0zMcggTQ&list=PLrk7hDwk64-a_gf7mBBduQb3PEBYnG4fU&index=3.

[4] Valem, "Introduction to VR in Unity - PART 7: DOOR, LEVER, DRAWER,...," Youtube, 21 June 2020. [Online]. Available: https://www.youtube.com/watch?v=bYS35_hC6B0&list=PLuU_htLgvU_HalAFgrf3UkuQAcOffTH9g&index=11.

[5] U. City, "How to make Jelly Mesh in unity || Softbody(tutorial)," Youtube, 28 September 2020. [Online]. Available:

https://www.youtube.com/watch?v=Kwh4TkQqqf8&list=PLuU_htLgvU_HalAFgrf3UkuQAcOffTH9g&index=4.

[6] Valem, "Introduction to VR in Unity - PART 5: GRAB INTERACTION," Youtube, 27 May 2020. [Online]. Available: https://www.youtube.com/watch?v=FMu7hKUX3Oo&list=PLrk7hDwk64-a_gf7mBBduQb3PEBYnG4fU&index=6.

[7] Valem, "Introduction to VR in Unity - Part 7: DOOR, LEVER, DRAWER,...," Youtube, 21 June 2020. [Online]. Available: https://www.youtube.com/watch?v=bYS35_hC6B0&list=PLuU_htLgvU_HalAFgrf3UkuQAcOffTH9g&index=11.

[8] Valem, "Introduction to VR in Unity - PART 8: SNAP ZONE," Youtube, 23 July 2020. [Online]. Available: https://www.youtube.com/watch?v=AWNhsSB6x9M&list=PLuU_htLgvU_HalAFgrf3UkuQAcOffTH9g&index=12.

[9] Valem, "Introduction to VR in Unity - PART 6: RAY INTERACTION," Youtube, 15 June 2020. [Online]. Available: https://www.youtube.com/watch?v=4tW7XpAiuDg&list=PLuU_htLgvU_HalAFgrf3UkuQAcOffTH9g&index=14.

[10] Brackeys, "START MENU in Unity," YouTube, 29 November 2017. [Online]. Available:

https://www.youtube.com/watch?v=zc8ac_qUXQY&list=PLuU_htLgvU_HalAFgrf3UkuQAcOffTH9g&index=4.

[11] Hellium, "Change Position of ganeobjects with slider," Stackoverflow, 16 May 2019. [Online]. Available: https://stackoverflow.com/questions/56176859/change-position-of-gameobjects-with-slider.

[12] A. Zotov, "How to create UI slider with text that shows percentage value in Unity | Simple Unity 2D tutorial," YouTube, 1 September 2017. [Online]. Available: https://www.youtube.com/watch?v=b3S5a_ohZZ0.

[13] C. Monkey, "Unity Tutorial - Create a Graph," YouTube, 22 June 2018. [Online]. Available: https://www.youtube.com/watch?v=CmU5-v-v1Qo&list=PLzDRvYVwl53v5ur4GluoabyckImZz3TVQ&index=3.

[14] C. Monkey, "Unity Tutorial - Create a Graph: Axis Separators," YouTube, 25 June 2018. [Online]. Available: https://www.youtube.com/watch?v=YVgMyQ3FWjI&list=PLzDRvYVwl53v5ur4GluoabyckImZz3TVQ&index=4.

[15] C. Monkey, "Unity Tutorial - Create a Graph: Axis Labels," YouTube, 29 June 2018. [Online]. Available: https://www.youtube.com/watch?v=3ozu5osNw-I&list=PLzDRvYVwl53v5ur4GluoabyckImZz3TVQ&index=5.

[16] Programmer, "How to make the script wait/sleep in a simple way in unity," Stack Overflow, 5 May 2015. [Online]. Available: https://stackoverflow.com/questions/30056471/how-to-make-the-script-wait-sleep-in-a-simple-way-in-unity.

[17] U. Technologies, "Render Texture," Unity Documentation, 11 04 2019. [Online]. Available: https://docs.unity3d.com/Manual/class-RenderTexture.html.

[18] J. Weimann, "Unity3D - How to Zoom an Image / Create a zoomable pannable sprite," YouTube, 15 September 2017. [Online]. Available: https://www.youtube.com/watch?v=BFX3FpUnoio.

[19] B. Herren, M. Saha, M. Altan and Y. Liu, "Development of Ultrastretchable and Skin Attachable Nanocomposites for Human Motion Monitoring via Embedded 3D Printing," *Composites, Part B: Engineering*, vol. 200, 2020.

[20] L. Chavez, B. Wilburn, P. Ibave, L. Delfin, S. Vargas, H. Diaz, C. Fulgentes, A. Renteria, J. Regis, Y. Liu, R. Wicker and Y. Lin, "Fabrication and characterization of 3D printing induced orthotropic functional ceramics," *Smart Materials and Structures*, vol. 28, no. 12, 2019.

[21] M. Charara, M. Abshirini, M. Saha, M. Altan and Y. Liu, "Highly sensitive compression sensors using three-dimensional printed polydimethylsiloxane/carbon

nanotube nanocomposites," *Journal of Intelligent Material Systems and Structures,* vol. 30, no. 8, pp. 1216-1224, 2019.

[22] L. Chavez, J. Regis, L. Delfin, C. Garcia-Rosales, H. Kim, N. Love, Y. Liu and Y. Lin, "Electrical and mechanical tuning of 3D printed photopolymer–MWCNT nanocomposites through in situ dispersion," *Journal of Applied Polymer Science,* vol. 136, no. 22, 2019.

[23] A. Renteria, J. Diaz, B. He, I. Rentaria-Marques, L. Chavez, J. Regis, Y. Liu, D. Espalin, T. Tseng and Y. Lin, "Particle size influence on material properties of BaTiO3 ceramics fabricated using freeze-form extrusion 3D printing," *Materials Research Express,* vol. 6, no. 11, 2019.

[24] A. Rentaria, H. Fontes, J. Diaz, J. Regis, L. Chaves, T. Tseng, Y. Liu and Y. Lin, "Optimization of 3D printing parameters for BaTiO3 piezoelectric ceramics through design of experiments," *Material Research Express,* vol. 6, no. 8, 2019.

[25] M. Abshirini, M. Charara, Y. Liu, M. Saha and M. Altan, "3D Printing of Highly Stretchable Strain Sensors Based on Carbon Nanotube Nanocomposites," *Advanced Engineering Materials*, vol. 20, no. 10, 2018.

[26] C. Billings, C. Cai and Y. Liu, "Utilization of Antibacterial Nanoparticles in Photocurable Additive Manufacturing of Advanced Composites for Improved Public Health," *Polymers,* vol. 13, no. 16, 2021.

Appendix A: Hand Presence Code

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR;

public class HandPresence : MonoBehaviour
{
    public bool showController = false;
    public InputDeviceCharacteristics controllerCharacteristics;
    public List<GameObject> controllerPrefabs;
    public GameObject handModelPrefab;

    private InputDevice targetDevice;
    private GameObject spawnedController;
    private GameObject spawnedHandModel;
    private Animator handAnimator;

    // Start is called before the first frame update
    void Start()
    {
        TryInitialize();

    }

    void TryInitialize()
    {
        List<InputDevice> devices = new List<InputDevice>();

        InputDevices.GetDevicesWithCharacteristics(controllerCharacteristi
cs, devices);

        foreach (var item in devices)
        {
            Debug.Log(item.name + item.characteristics);
        }

        if(devices.Count > 0)
        {
            targetDevice = devices[0];
            GameObject prefab = controllerPrefabs.Find(controller =>
controller.name == targetDevice.name);
```

```
            if(prefab)
            {
                spawnedController = Instantiate(prefab, transform);
            }
            else
            {
                Debug.LogError("Did not find corresponding controller
model");
                spawnedController = Instantiate(controllerPrefabs[0],
transform);
            }
            spawnedHandModel = Instantiate(handModelPrefab, transform);
            handAnimator = spawnedHandModel.GetComponent<Animator>();
        }
    }

    void UpdateHandAnimation()
    {
        if(targetDevice.TryGetFeatureValue(CommonUsages.trigger, out float
triggerValue))
        {
            handAnimator.SetFloat("Trigger", triggerValue);
        }
        else
        {
            handAnimator.SetFloat("Trigger", 0);
        }
        if(targetDevice.TryGetFeatureValue(CommonUsages.grip, out float
gripValue))
        {
            handAnimator.SetFloat("Grip", gripValue);
        }
        else
        {
            handAnimator.SetFloat("Grip", 0);
        }
    }

    // Update is called once per frame
    void Update()
    {
        if(!targetDevice.isValid)
```

```
        {
            TryInitialize();
        }

        else
        {
            if(showController)
            {
                spawnedHandModel.SetActive(false);
                spawnedController.SetActive(true);
            }
            else
            {
                spawnedHandModel.SetActive(true);
                spawnedController.SetActive(false);
                UpdateHandAnimation();
            }
        }
    }
}
```

Appendix B: Jelly Mesh Code

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class jelly : MonoBehaviour
{
    public float Intensity = 1f;
    public float Mass = 1f;
    public float stifness = 1f;
    public float dampining = 0.75f;
    private Mesh OriginalMesh, MeshClone;
    private MeshRenderer mrenderer;
    private JellyVertex[] jv;
    private Vector3[] vertexArray;
    // Start is called before the first frame update
    void Start()
    {
        OriginalMesh = GetComponent<MeshFilter>().sharedMesh;
        MeshClone = Instantiate(OriginalMesh);
        GetComponent<MeshFilter>().sharedMesh = MeshClone;
        mrenderer = GetComponent<MeshRenderer>();

        jv = new JellyVertex[MeshClone.vertices.Length];
        for (int i = 0; i < MeshClone.vertices.Length; i++)
            jv[i] = new JellyVertex(i,
transform.TransformPoint(MeshClone.vertices[i]));
    }

    // Update is called once per frame
    void FixedUpdate()
    {
        vertexArray = OriginalMesh.vertices;
        for(int i = 0; i < jv.Length; i++)
        {
            Vector3 target =
transform.TransformPoint(vertexArray[jv[i].ID]);
            float intesity = (1 - (mrenderer.bounds.max.x - target.x) /
mrenderer.bounds.size.x) * Intensity;
            //jv[i].Shake(target, Mass, stifness, dampining);
            target = transform.InverseTransformPoint(jv[i].position);
```

```
            vertexArray[jv[i].ID] = Vector3.Lerp(vertexArray[jv[i].ID],
target, intesity);
        }
        MeshClone.vertices = vertexArray;
    }

    public class JellyVertex
    {
        public int ID;
        public Vector3 position;
        public Vector3 velocity, force;

        public JellyVertex(int _id,Vector3 _pos)
        {
            ID = _id;
            position = _pos;

        }

        /*public void Shake(Vector3 target, float m, float s, float d)
        {
            force = (target - position) * s;
            velocity = (velocity + force / m) * d;
            position += velocity;
            if ((velocity + force + force / m).magnitude < 0.001f)
            {
                position = target;
            }

        } */
    }
}
```

Appendix C: Scene Traversal Code

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
public class Zoom : MonoBehaviour
{
    public void PlayGame ()
    {
        SceneManager.LoadSceneAsync("2.Sim");
    }

        public void SEMClose ()
    {
        SceneManager.LoadSceneAsync("3.SEM Close");
    }

        public void Computer ()
    {
        SceneManager.LoadSceneAsync("4.Computer");
    }

    public void Gallery ()
    {
        SceneManager.LoadSceneAsync("5.Gallery");
    }

    public void GalleryTwo ()
    {
        SceneManager.LoadSceneAsync("6.Gallery2");
    }

    public void Home ()
    {
        SceneManager.LoadSceneAsync("4.1HomeScreen");
    }

    public void PDMS1mm ()
    {
        SceneManager.LoadSceneAsync("PDMS_1mm");
    }
    public void PDMS1mm_2 ()
```

```
    {
        SceneManager.LoadSceneAsync("PDMS_1mm_2");
    }

    public void PDMS500um ()
    {
        SceneManager.LoadSceneAsync("PDMS_500um");
    }

    public void PDMS200um ()
    {
        SceneManager.LoadSceneAsync("PDMS_200um");
    }

    public void CNF500 ()
    {
        SceneManager.LoadSceneAsync("CNF_500");
    }

    public void CNF200 ()
    {
        SceneManager.LoadSceneAsync("CNF_200");
    }

    public void CNF100 ()
    {
        SceneManager.LoadSceneAsync("CNF_100");
    }

    public void CNF50 ()
    {
        SceneManager.LoadSceneAsync("CNF_50");
    }

    public void CNF20 ()
    {
        SceneManager.LoadSceneAsync("CNF_20");
    }

    public void G1 ()
    {
        SceneManager.LoadSceneAsync("I1_P2_L3_6.9N_2.12mm");
```

```
    }

    public void G2 ()
    {
        SceneManager.LoadSceneAsync("I1_P2_L4_5.9N_1.6mm");
    }

    public void G3 ()
    {
        SceneManager.LoadSceneAsync("I1_P2_L5_3.9N_0.8mm");
    }

    public void G4 ()
    {
        SceneManager.LoadSceneAsync("I1_P2_L6_1.8N_0mm");
    }

    public void G5 ()
    {
        SceneManager.LoadSceneAsync("I1_P3_L4_5.5N_2mm");
    }

    public void G6 ()
    {
        SceneManager.LoadSceneAsync("I1_P3_L5_4.5N_1.4mm");
    }

    public void G7 ()
    {
        SceneManager.LoadSceneAsync("I1_P3_L6_3.4N_0.7mm");
    }

    public void G8 ()
    {
        SceneManager.LoadSceneAsync("I1_P3_L7_2.2N_0mm");
    }

    public void QuitGame ()
    {
        Application.Quit();
    }
}
```

Appendix D: Slider Object Translation Code

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class ChangePositionWithSlider : MonoBehaviour
    {
        public Slider slider;
        public Transform objectTransform;
        private Vector3 position1 = new Vector3(0.58f, 0.77f, 3.275f);
        private Vector3 position2 = new Vector3(0.64f, 0.77f, 3.275f);

        private void Start()
        {
            // Make sure the slider value is clamped between 0 and 1

            slider.onValueChanged.AddListener(UpdatePosition);
        }

        public void UpdatePosition(float value)
        {
            Vector3 newPosition = Vector3.Lerp(position1, position2,
value);
            objectTransform.position = newPosition;
        }
    }
```

Appendix E: Percentage Value Code

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class PercentageShowValue: MonoBehaviour {
  Text percentageText;
  void Start() {
    percentageText = GetComponent < Text > ();
  }

  public void textUpdate(float value) {
    percentageText.text = Mathf.RoundToInt(value * 100) + "%";
  }
}
```

Appendix F: Move Percentage Code

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class MovePercentage : MonoBehaviour
    {
        public Slider slider;
        public Transform objectTransform;
        private Vector3 position1 = new Vector3(1.533f, 0.726999f,
3.529f);
        private Vector3 position2 = new Vector3(1.742f, 0.726999f,
3.529f);

        private void Start()
        {
            // Make sure the slider value is clamped between 0 and 1

            slider.onValueChanged.AddListener(UpdatePosition);
        }

        public void UpdatePosition(float value)
        {
            Vector3 newPosition = Vector3.Lerp(position1, position2,
value);
            objectTransform.position = newPosition;
        }
    }
```

Appendix G: Live Graph Code

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using CodeMonkey.Utils;

public class Window_Graph : MonoBehaviour {

    [SerializeField] private Sprite circleSprite;
    private RectTransform graphContainer;
    private RectTransform labelTemplateX;
    private RectTransform labelTemplateY;
    private RectTransform dashTemplateX;
    private RectTransform dashTemplateY;
    public Slider slider;
        private int N1=0;
        private int N2=0;
        private int N3=0;
        private int N4=0;
        private int N5=0;
        private int N6=0;
        private int N7=0;
        private int N8=0;
        private int N9=0;
        private int N10=0;
        private int N11=0;
        private int N12=0;
        private int N13=0;
        private int N14=0;
        private int N15=0;
        private int NN=0;
        private int temp=2;


    private void Awake() {

        graphContainer =
transform.Find("graphContainer").GetComponent<RectTransform>();
        labelTemplateX =
graphContainer.Find("labelTemplateX").GetComponent<RectTransform>();
```

```
        labelTemplateY =
graphContainer.Find("labelTemplateY").GetComponent<RectTransform>();
        dashTemplateX =
graphContainer.Find("dashTemplateX").GetComponent<RectTransform>();
        dashTemplateY =
graphContainer.Find("dashTemplateY").GetComponent<RectTransform>();

        List<int> valueList = new List<int>() { N1, N2, N3, N4, N5, N6,
N7, N8, N9, N10, N11, N12, N13, N14, N15 };
        ShowGraph(valueList, (int _i) => (_i+1) + "s", (float _f) => "σ" +
Mathf.RoundToInt(_f));
       // slider.onValueChanged.AddListener(UpdateValue);

    }



//////
     private GameObject CreateCircle(Vector2 anchoredPosition) {
        GameObject gameObject = new GameObject("circle", typeof(Image));
        gameObject.transform.SetParent(graphContainer, false);
        gameObject.GetComponent<Image>().sprite = circleSprite;
        RectTransform rectTransform =
gameObject.GetComponent<RectTransform>();
        rectTransform.anchoredPosition = anchoredPosition;
        rectTransform.sizeDelta = new Vector2(11, 11);
        rectTransform.anchorMin = new Vector2(0, 0);
        rectTransform.anchorMax = new Vector2(0, 0);
        return gameObject;
    }
///////
     private void ShowGraph(List<int> valueList, Func<int, string>
getAxisLabelX = null, Func<float, string> getAxisLabelY = null) {
        if (getAxisLabelX == null) {
            getAxisLabelX = delegate (int _i) { return _i.ToString(); };
        }
        if (getAxisLabelY == null) {
            getAxisLabelY = delegate (float _f) { return
Mathf.RoundToInt(_f).ToString(); };
        }

        float graphHeight = graphContainer.sizeDelta.y;
```

```
        float yMaximum = 20f;//100
        float xSize = 40f;//50

        GameObject lastCircleGameObject = null;
        for (int i = 0; i < valueList.Count; i++) {
            float xPosition = xSize + i * xSize;
            float yPosition = (valueList[i] / yMaximum) * graphHeight;
            GameObject circleGameObject = CreateCircle(new
Vector2(xPosition, yPosition));
      /*      if (lastCircleGameObject != null) {
              CreateDotConnection(lastCircleGameObject.GetComponent<Rect
Transform>().anchoredPosition,
circleGameObject.GetComponent<RectTransform>().anchoredPosition);
            }*/
            lastCircleGameObject = circleGameObject;
            //====================== delete old circles
            Destroy(circleGameObject, 0.1f);
            //====================


            RectTransform labelX = Instantiate(labelTemplateX);
            labelX.SetParent(graphContainer, false);
            labelX.gameObject.SetActive(true);
            labelX.anchoredPosition = new Vector2(xPosition+50, -60f);
            labelX.GetComponent<Text>().text = getAxisLabelX(i);

            RectTransform dashX = Instantiate(dashTemplateX);
            dashX.SetParent(graphContainer, false);
            dashX.gameObject.SetActive(true);
            dashX.anchoredPosition = new Vector2(xPosition+10, 300f);
        }

        int separatorCount = 10;
        for (int i = 0; i <= separatorCount; i++) {
            RectTransform labelY = Instantiate(labelTemplateY);
            labelY.SetParent(graphContainer, false);
            labelY.gameObject.SetActive(true);
            float normalizedValue = i * 1f / separatorCount;
            labelY.anchoredPosition = new Vector2(22f, normalizedValue *
graphHeight * 5);
            labelY.GetComponent<Text>().text =
getAxisLabelY(normalizedValue *10 );//* yMaximum);
```

```
            RectTransform dashY = Instantiate(dashTemplateY);
            dashY.SetParent(graphContainer, false);
            dashY.gameObject.SetActive(true);
            dashY.anchoredPosition = new Vector2(330f, normalizedValue *
graphHeight * 5 + 40);
        }
    }
///////
     /*private void CreateDotConnection(Vector2 dotPositionA, Vector2
dotPositionB) {
        GameObject gameObject = new GameObject("dotConnection",
typeof(Image));
        gameObject.transform.SetParent(graphContainer, false);
        gameObject.GetComponent<Image>().color = new Color(1,1,1, .5f);
        RectTransform rectTransform =
gameObject.GetComponent<RectTransform>();
        Vector2 dir = (dotPositionB - dotPositionA).normalized;
        float distance = Vector2.Distance(dotPositionA, dotPositionB);
        rectTransform.anchorMin = new Vector2(0, 0);
        rectTransform.anchorMax = new Vector2(0, 0);
        rectTransform.sizeDelta = new Vector2(distance, 3f);
        rectTransform.anchoredPosition = dotPositionA + dir * distance *
.5f;
        rectTransform.localEulerAngles = new Vector3(0, 0,
UtilsClass.GetAngleFromVectorFloat(dir));
        //====================== delete old Lines
            //Destroy(distance, 0.1f);

        //====================
   } */


//////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////
    public void Update()//Move dot with slider
        {
            N1 = Mathf.RoundToInt(slider.value * 140);
            NN = N1;
            List<int> valueList = new List<int>() { N1, N2, N3, N4, N5,
N6, N7, N8, N9, N10, N11, N12, N13, N14, N15 };
            ShowGraph(valueList, (int _i) => (_i+1) + "s", (float _f) =>
"o" + Mathf.RoundToInt(_f));
```

```
        //if (make counter)(if number is even) use temp

        if(temp%2==0){
                StartCoroutine(wait());
                StartCoroutine(waiter());
        }

        else{
            StartCoroutine(pause());
             temp=temp-2;

        }

    }

    IEnumerator wait()
    {
        yield return new WaitForSeconds(1f);
        N2=NN;
        temp = temp+1;
    }

    IEnumerator waiter()
    {

            yield return new WaitForSeconds(2f);
            N3=N2;
            yield return new WaitForSeconds(2f);
            N4=N3;
            yield return new WaitForSeconds(2f);
            N5=N4;
            yield return new WaitForSeconds(2f);
            N6=N5;
            yield return new WaitForSeconds(2f);
            N7=N6;
            yield return new WaitForSeconds(2f);
            N8=N7;
            yield return new WaitForSeconds(2f);
            N9=N8;
            yield return new WaitForSeconds(2f);
            N10=N9;
```

```
                yield return new WaitForSeconds(2f);
                N11=N10;
                yield return new WaitForSeconds(2f);
                N12=N11;
                yield return new WaitForSeconds(2f);
                N13=N12;
                yield return new WaitForSeconds(2f);
                N14=N13;
                yield return new WaitForSeconds(2f);
                N15=N14;
        }

        IEnumerator pause()
        {
            yield return new WaitForSeconds(1f);
            temp = temp+1;
        }


}
```

Appendix H: Zoomed Image

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.EventSystems;

public class UI_ZOOM_IMAGE : MonoBehaviour, IScrollHandler
{
    private Vector3 initialScale;

    [SerializeField]
    private float zoomSpeed = 1f;
    [SerializeField]
    private float maxZoom = 100f;

    private void Awake()
    {
        initialScale = transform.localScale;
    }

    public void OnScroll(PointerEventData eventData)
    {
        var delta = Vector3.one * (eventData.scrollDelta.y * zoomSpeed);
        var desiredScale = transform.localScale + delta;

        desiredScale = ClampDesiredScale(desiredScale);

        transform.localScale = desiredScale;
    }

    private Vector3 ClampDesiredScale(Vector3 desiredScale)
    {
        desiredScale = Vector3.Max(initialScale, desiredScale);
        desiredScale = Vector3.Min(initialScale * maxZoom, desiredScale);
        return desiredScale;
    }
}
```