Virtual Reality Environment Recreation

Ryan Douglas
rwd22@zips.uakron.edu

Virtual Reality Environment Recreation

Ryan Douglas

Williams Honors College

ABSTRACT

This project will consist of a virtual reality based program that is capable of showing the user both the modern day state of a site of historic or archaeological significance, along with a recreation of what said site or area may have looked like in the past, primarily during the time that gave the site its historical significance. The virtual reality program itself is to be run on modern day Windows hardware and used with the VIVE virtual reality head-mounted display and controllers. Alongside the completed program, the creation of the environments themselves will be documented, resulting in an organized method to create more environments in the future allowing for the application itself to be added to and greatly increasing its versatility in future educational or recreational use.

The project itself has resulted in the creation of the initial program made to showcase the environment switching capabilities of the project along with a toolset made for the Unity engine. This toolset allows for a simple method to import a three-dimensional scan of an environment into the Unity editor, and for that scan to be made into an environment that can be explored using the project.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

This project aims to create a set of tools for use within the Unity3D editor that will allow for a user to take a 3D scan of an environment and a recreation of what it may have looked like in the past, and insert it into the engine. The environment and its past variation will then be able to be explored by the user, with the ability to swap back and forth between the two environments. This will not only allow for the environments to be viewed and explored with their appropriate scale, but this will also allow for the differences between the two variations of the environment to be more easily examined as if the user was at the scene in person. This will also allow a user that is incapable of exploring the environment in question, either due to their location or due to the real world equivalent of the environment being off limits to the public, to experience the environment as if they were there.

CHAPTER II

BACKGROUND

The initial idea for this project started with a conversation between me and Dr. Xiao. I had just finished my final project for one of his classes and had expressed my interest to pursue working with Virtual Reality equipment further in my academic career. He then mentioned to me that I could explore Virtual Reality based development further through my honors project. Over time this eventually evolved into the Virtual Reality Environment Recreation project that I've worked on for my honors project.

A key part of archeology is examining an environment or an object and determining its appearance before the passage of time caused it to become damaged or weathered. This was typically accomplished and shown to others using a sketch, but the advancements of computer technology allowed for these assessments to be shown using three dimensional computer generated graphics, albeit, on a screen. This method, while much more substantial than a simple two dimensional sketch, still results in what is essentially an object that can only be viewed two dimensionally due to the nature of viewing an object on a simple computer screen. Along with this, the object or environment can only be manipulated using the computer's controls which are commonly limited to a simple keyboard and mouse. This results in the impressive recreation of an object or environment with archeological significance lacking the ability to truly express its scale and being unable to be closely examined.

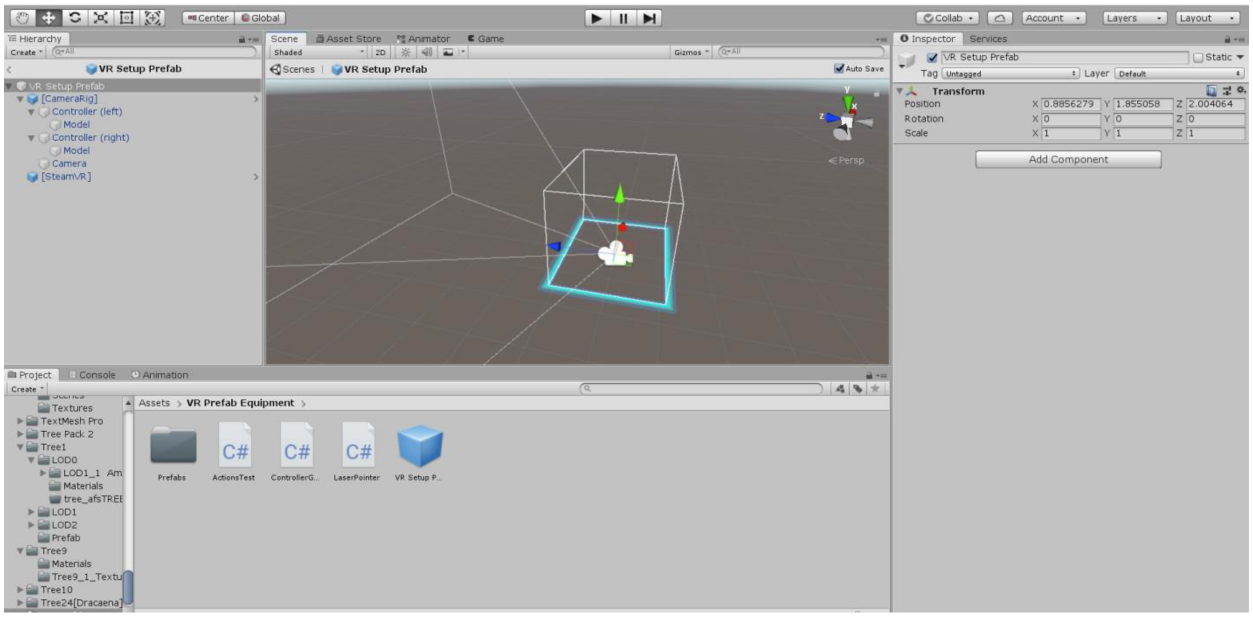
For this project, I wished to utilize Virtual Reality equipment to help to remedy these issues. With the rapidly increasing rate of advancements being made in Virtual Reality hardware and software, it seems that Virtual Reality is becoming a standard way to consume media in a way that was not previously possible. One clear advantage of Virtual Reality over using a standard computer monitor is its ability to convey scale to its users. By essentially placing the user into the same world as an object or environment, the size and scale of it is shown to the user purely due to the fact that their own height acts as their reference point. By combining this with the fact that a Virtual Reality headset shows the user the displayed graphics in stereoscopic 3D, not only can size and scale be more accurately conveyed, but the depth of details on an object or environment can be easily shown to a user purely through them examining the given entity themselves. Along with this, Virtual Reality in its current state allows users to manipulate objects using their hands, usually through a controller. This means that, rather than using a keyboard or mouse to interact with an object and examine it, they are given the simulated experience of picking up and handling the object as they please. This further drives home a realism and natural feeling that simply cannot be portrayed using a standard monitor and method of interaction. This idea seems to be mostly untapped, especially in regards to a method that allows for a relatively inexperienced user to add their environment and that allows for the use of consumer grade Virtual Reality hardware to view said environment, rather than using more expensive hardware.

CHAPTER III

DESIGN

In terms of the environment that this project was created with, the most obvious choice given my previous experience was the Unity3D engine. I’ve had a significant amount of experience using both this engine and the programming language that it uses, that being C#, over my academic career, along with some sparse experience with implementing Virtual Reality using this engine. Specifically, the version of Unity3D used for this project was Unity 2018.3.2f1, as this was the most recent release at the time of the project’s initial development period.

In regards to the hardware the project uses for its Virtual Reality implementation, I used the highest quality consumer grade Virtual Reality headset that I had access to, being the HTC VIVE Virtual Reality headset and its controllers. This headset is compatible with the Unity3D engine through the SteamVR SDK which is available through the Unity Asset Store. This allows for Unity to correctly receive inputs from controllers and headsets that are compatible with SteamVR along with their tracking data, thus allowing Unity to correctly process their positions within the virtual space. Thus, at this point I was able to have rudimentary Virtual Reality implementation within a Unity project, but there was no form of interaction that the user was able to perform, which essentially rendered the user as nothing but a camera where their head was located. Below is an example of the Unity3D editor with the SteamVR object loaded into the scene, which is used by the Unity engine to track the positions and actions of the controllers and headset.



VR Setup Prefab — Static
Tag: Untagged — Layer: Default

Transform
Position X 0.8856279 Y 1.855058 Z 2.004064
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Figure 1. SteamVR object within Unity.

<u>User Interactions</u>

In order to give the user some form of control, a method for the user to interact with the environment had to be devised. The most basic interaction that the user must be able to perform in this project is movement, so as part of the initial research for this project several examples of movement methods in Virtual Reality were studied.

The most logical place to look for examples of different methods of locomotion in virtual reality would be in the form of virtual reality games quite simply because they are easy for me to access and many games employ various forms of locomotion and interaction. Using several sources I was able to narrow down a few of the methods of locomotion that I believed would possibly be well suited for the project and which resulted in my decision to create a teleporting method where the user would point to a location within the environment using the VIVE controller while pressing a button and then, upon releasing the button, the user's position would be changed to that of where they were pointing. The following diagram shows the basic logic for the teleporting interaction in this project.

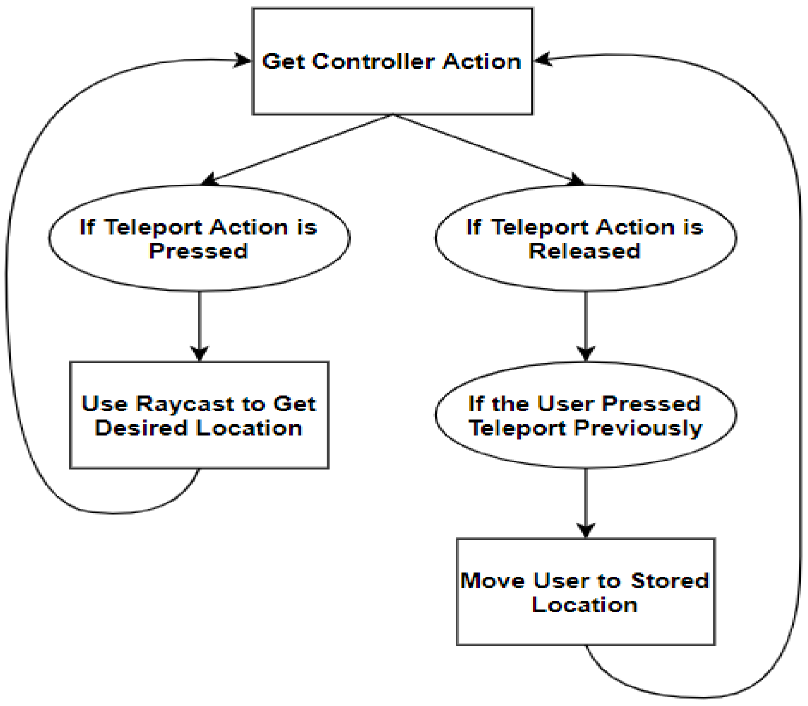

Get Controller Action
If Teleport Action is Pressed
If Teleport Action is Released
Use Raycast to Get Desired Location
If the User Pressed Teleport Previously
Move User to Stored Location

Figure 2. Teleporting logic diagram.

This logic is similar to the basic teleporting actions seen in several modern virtual reality games, and will help to prevent the motion sickness related with a "free-moving" method of locomotion in Virtual Reality.

Environments

Within this project there are several environments that were created as proof of concepts to show the result of using the assets created for this project in their intended fashion. The variations of said environments where created through a mixture of original assets and freely available assets obtained using the Unity Asset Store. Unity handles these separate variations of the environments as separate “scenes”. Simply put a scene is akin to a self-contained environment within the engine which needs to be marked to be added to the final release. If several scenes are added to be included in the build for the project, there must be a method to access each of them, otherwise the first scene listed in the build options will be loaded by default, which essentially limits the user to only being able to access that single scene. Below is an example of one of the environments created to showcase this project’s capabilities. Both the “past” and “present” variations can be seen as being viewed within the Unity editor to provide an aerial view of the environment.

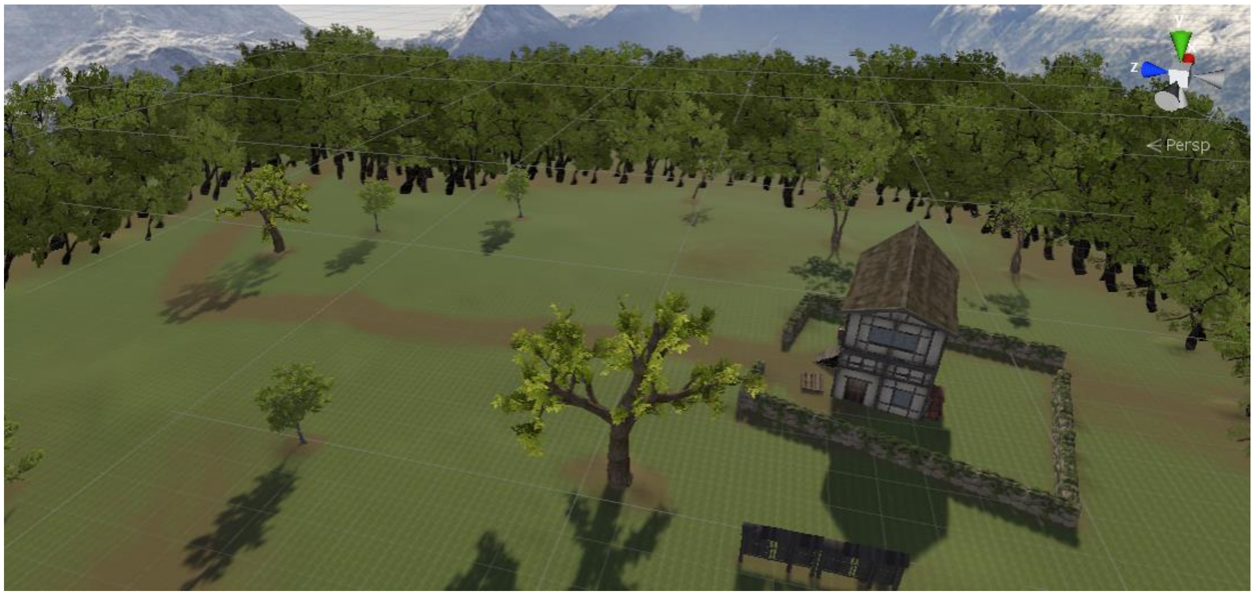


Figure 3. Past environment variation.

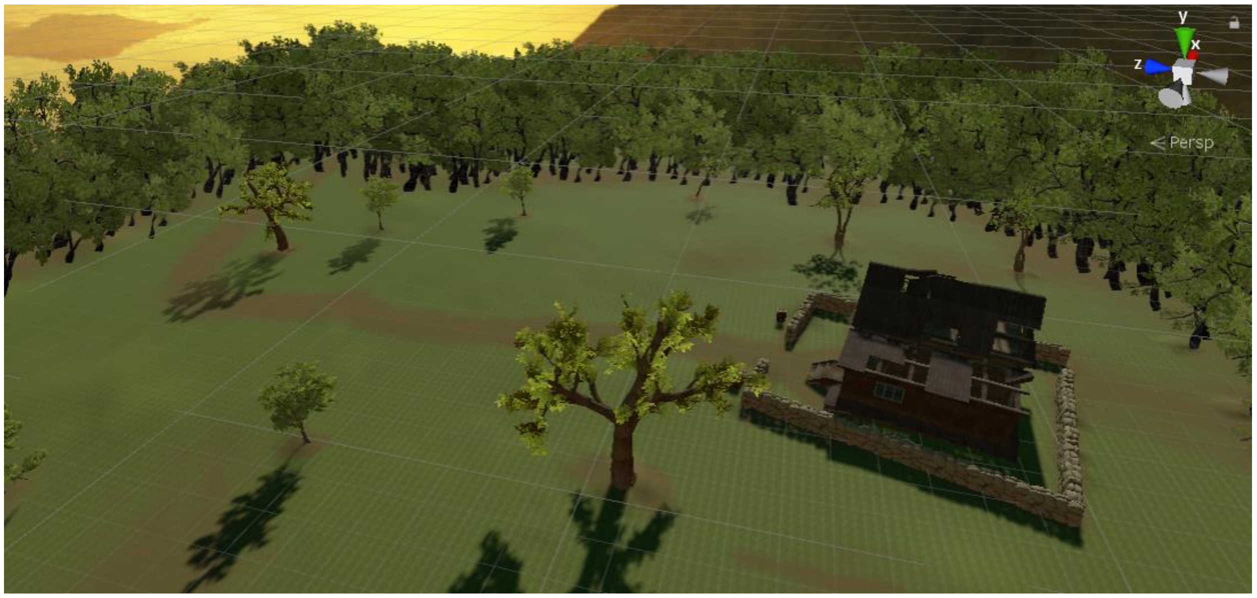


Figure 4. Future environment variation.

A key part of this project is the ability to switch between an environment's "past" and "present" variations. In order to allow the user to select from several environments and their variations a simple menu had to be created that could function in the virtual environment. Due to the nature of Virtual Reality, this menu would need to load into the environment as an object which the user could physically interact with using their equipment rather than a simple text based menu. In order to accomplish this, a menu object was created as a prefab object that the project could load into the environments and their variations.

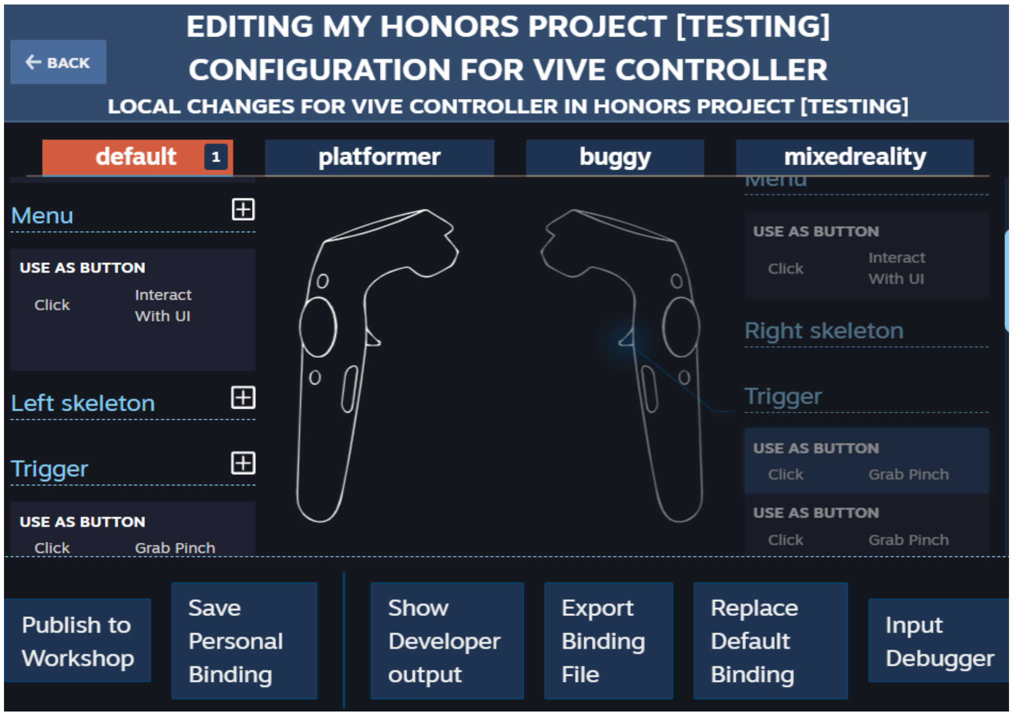
Once these basic elements are fully implemented, a toolset was designed to allow for a user to drop a three-dimensional scan of a real world environment into the Unity editor. This scan can then be made into an environment compatible with the project's implemented methods using the toolset provided by the project.

CHAPTER IV

IMPLEMENTATION

User Movements

The first thing to be fully implemented into the project was the movement capabilities regarding the user when in Virtual Reality. Much of the process of obtaining the controller's actions is handled by the SteamVR SDK. The controller's actions are defined through the SDK, and the actions are assigned to the buttons on the controllers through SteamVR's Controller Binding UI. These are then used to generate a JSON file that the Unity Project uses to keep track of what actions the controller is performing. Below is an example of the SteamVR controller bindings UI. This is used to create actions and assign them to the buttons on the controllers to be transmitted to and recognized by the Unity engine.


EDITING MY HONORS PROJECT [TESTING]
CONFIGURATION FOR VIVE CONTROLLER
LOCAL CHANGES FOR VIVE CONTROLLER IN HONORS PROJECT [TESTING]

Menu
USE AS BUTTON
Click — Interact With UI

Left skeleton

Trigger
USE AS BUTTON
Click — Grab Pinch

Menu
USE AS BUTTON
Click — Interact With UI

Right skeleton

Trigger
USE AS BUTTON
Click — Grab Pinch
USE AS BUTTON
Click — Grab Pinch

Figure 5. SteamVR controller bindings.

Along with the basic logic for the teleporting action, a method must be used to allow the user to know where they are pointing to. This was achieved by making a “laser” object that stretches between the user’s controller and the point determined by raycasting when the user presses the teleport action, along with a reticle that shows on the ground at the determined point. After this was completed, a small test environment was created to ensure that the movement system was completed with light testing, which can be seen below.

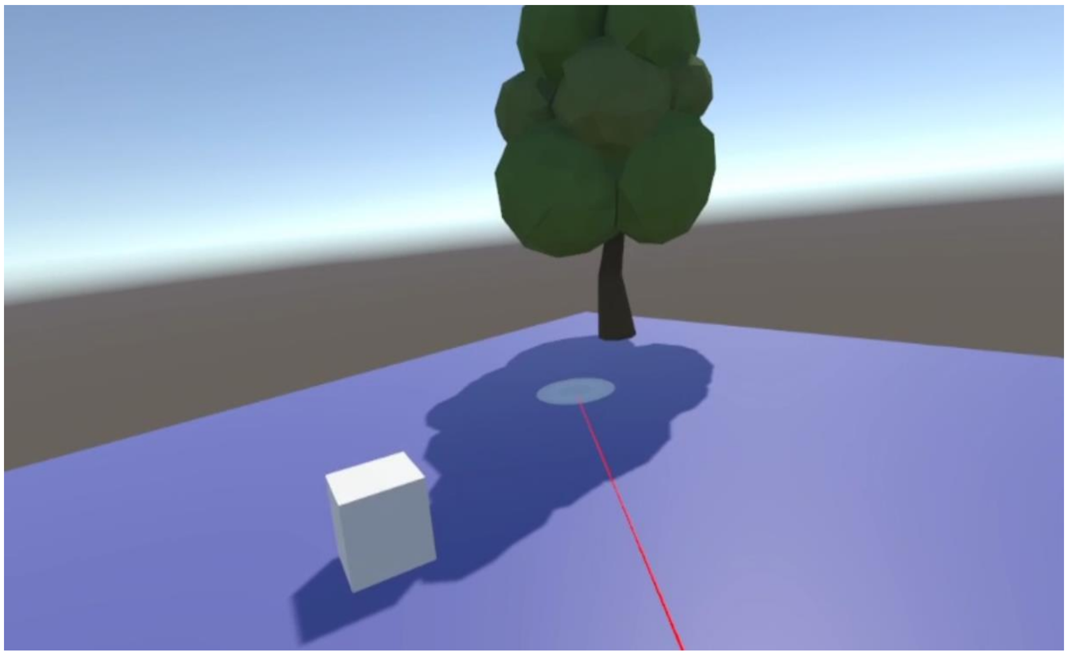
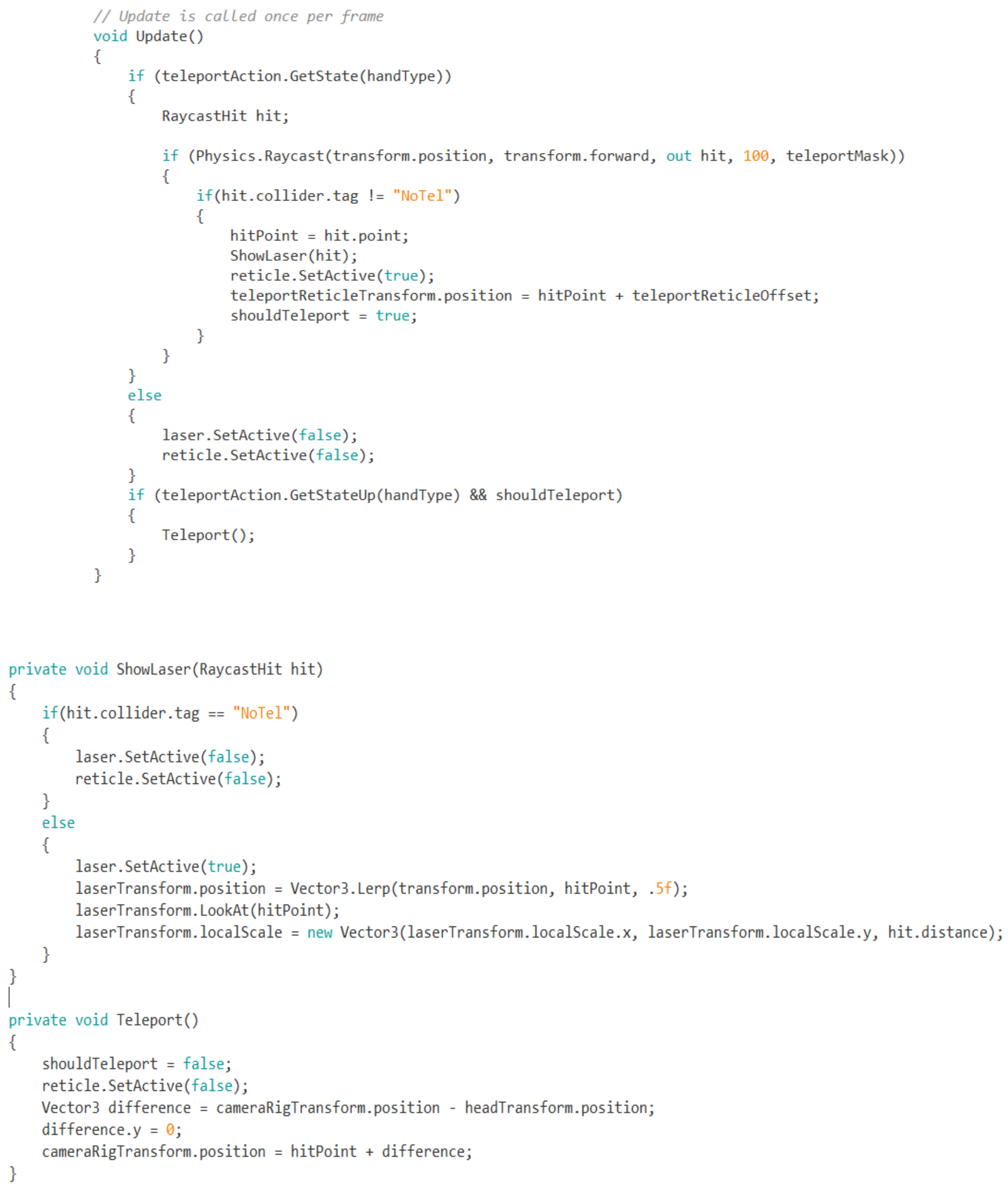


Figure 6. Simple testing environment.

Once I had finished working on the basics of this movement system, I noticed an issue with the way it handled objects between the user and the location that the user is pointing at. Objects that had collision allowed the user to teleport onto the side of the object, as if the user were climbing it. If the object has no collision and the user points to it, the teleport location would be on the other side of the object. These are issues, especially if the object is meant to be solid or even used as a boundary to the space where the user should be allowed to teleport within the environment. This was solved by creating a tag for said objects in Unity to label them as “NoTel” and if the raycast for where the user is pointing comes into contact with an object with said tag, the laser object is deactivated and the point for the user to teleport to is not set. The code snippet regarding the teleportation action along with stopping the registration of the raycast’s hitpoint and deactivating the laser indicator are shown below.



```
// Update is called once per frame
void Update()
{
    if (teleportAction.GetState(handType))
    {
        RaycastHit hit;

        if (Physics.Raycast(transform.position, transform.forward, out hit, 100, teleportMask))
        {
            if(hit.collider.tag != "NoTel")
            {
                hitPoint = hit.point;
                ShowLaser(hit);
                reticle.SetActive(true);
                teleportReticleTransform.position = hitPoint + teleportReticleOffset;
                shouldTeleport = true;
            }
        }
    }
    else
    {
        laser.SetActive(false);
        reticle.SetActive(false);
    }
    if (teleportAction.GetStateUp(handType) && shouldTeleport)
    {
        Teleport();
    }
}
```

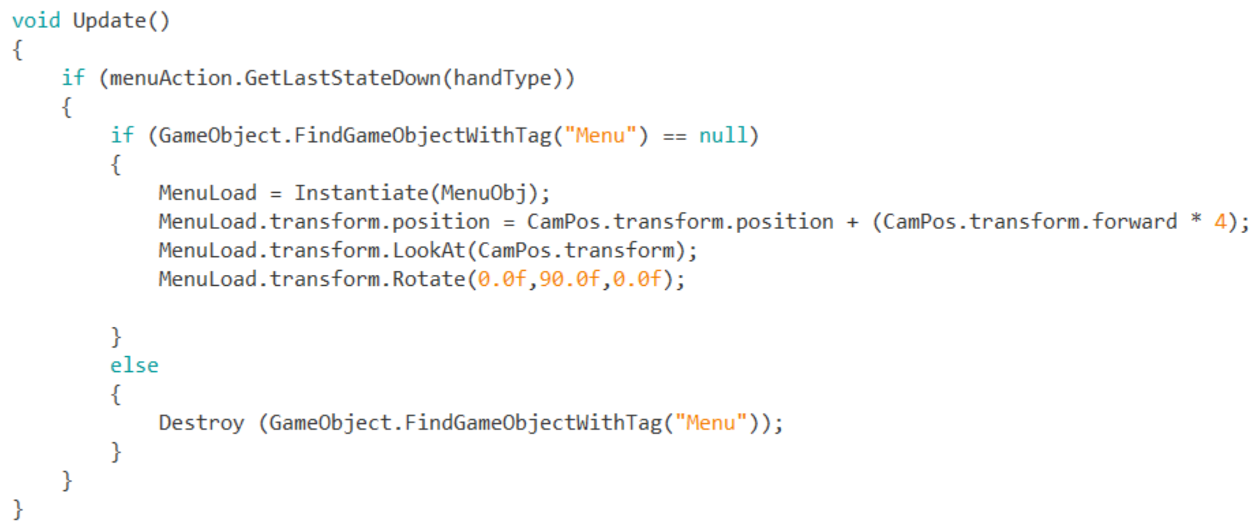
```
private void ShowLaser(RaycastHit hit)
{
    if(hit.collider.tag == "NoTel")
    {
        laser.SetActive(false);
        reticle.SetActive(false);
    }
    else
    {
        laser.SetActive(true);
        laserTransform.position = Vector3.Lerp(transform.position, hitPoint, .5f);
        laserTransform.LookAt(hitPoint);
        laserTransform.localScale = new Vector3(laserTransform.localScale.x, laserTransform.localScale.y, hit.distance);
    }
}

private void Teleport()
{
    shouldTeleport = false;
    reticle.SetActive(false);
    Vector3 difference = cameraRigTransform.position - headTransform.position;
    difference.y = 0;
    cameraRigTransform.position = hitPoint + difference;
}
```

Figure 7. Teleportation and laser casting code.

This method works well, but due to the way that unity treats raycasting in regards to collision and the way that basic SteamVR compatibility works, if a user forces their way into the collider of a boundary, they can bypass the boundary, but the boundaries still will function as essentially “handrails” to keep the player within the suggested area. In later user testing, if this becomes an issue, a solution could be made to decrease the user’s visibility when outside of the boundaries as a visible indicator that the user is outside of the bounds.

Environment Handling

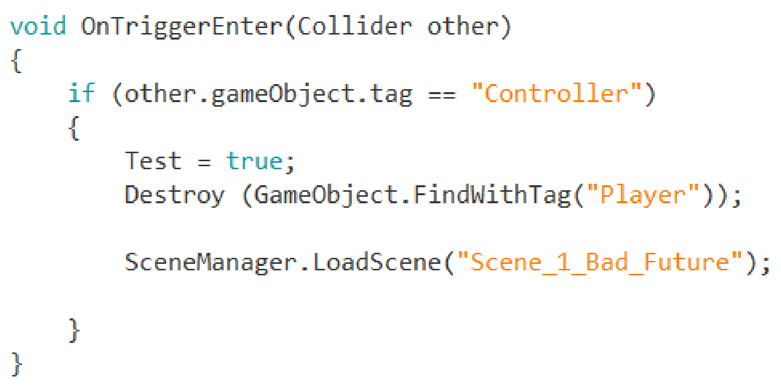
As each of the environment’s variants are treated as scenes by Unity, a script was created that checks to see if the controller is being given an input that is defined by SteamVR as the menu action. The script then checks to see if the menu has already been instantiated into the scene as an object. If the menu has not been instantiated into the scene the menu prefab is loaded into the scene in a manner that places it in front of the camera object, which in this case is essentially the user’s face, thus placing it in front of the user. If the menu is found to have been instantiated into the current scene already then the menu object within the scene is destroyed. This script is then placed onto the HTC VIVE controller causing it to act as a “watcher” looking to see if the correct action is performed. This allows the user to open and close the menu using the button on the controller, which for this project was defined to be the HTC VIVE Controller’s menu button. The following is the code used to load the menu object into a scene.



```
void Update()
{
    if (menuAction.GetLastStateDown(handType))
    {
        if (GameObject.FindGameObjectWithTag("Menu") == null)
        {
            MenuLoad = Instantiate(MenuObj);
            MenuLoad.transform.position = CamPos.transform.position + (CamPos.transform.forward * 4);
            MenuLoad.transform.LookAt(CamPos.transform);
            MenuLoad.transform.Rotate(0.0f,90.0f,0.0f);

        }
        else
        {
            Destroy (GameObject.FindGameObjectWithTag("Menu"));
        }
    }
}
```

Figure 8. Menu instantiation and destruction code.

For the menu's interactions themselves, each of the parts of the menu are labeled with the name of the corresponding environment variation. These separate parts of the menu are then given scripts that look for collisions with another object. If the object attached to the collider that the menu option has come into contact with has been tagged in the Unity editor with a "Controller" tag, the object representing the user's Virtual Reality equipment is destroyed and the scene corresponding to the menu option is loaded using a string of the scene name for the corresponding environment. The objects representing the HTC VIVE Controllers were then given a collider and tagged appropriately using the Unity editor. This allows the user to touch a menu option and then the environment variation they have selected is loaded, providing the user with a menu selection method that works well within Virtual Reality as if it were a physical interaction. The following is an example of this code.

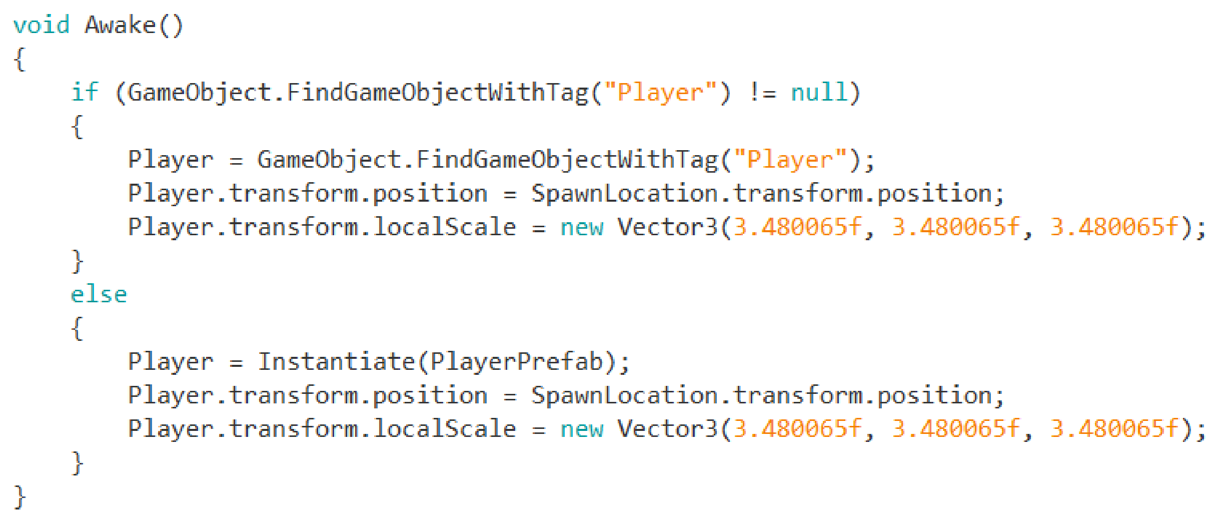


```
void OnTriggerEnter(Collider other)
{
    if (other.gameObject.tag == "Controller")
    {
        Test = true;
        Destroy (GameObject.FindWithTag("Player"));

        SceneManager.LoadScene("Scene_1_Bad_Future");

    }
}
```

Figure 9. Environment loading code.

Initially the environment loading method caused an interesting issue regarding the object that Unity uses to access SteamVR’s positional and controller inputs. Due to the scenes’ requiring this object in order to correctly place the user within the scene and display the environment through the Virtual Reality headset rather than a default Unity camera object, the user would essentially be loaded into an environment a second time upon switching scenes with the menu. This would result in the user, while still seeing the environment through their headset, being represented within the scene by a second object as well. Thus the user could turn to see a second set of their hands floating in the air and mimicking their movements in the environment’s intended spawn location for the user.

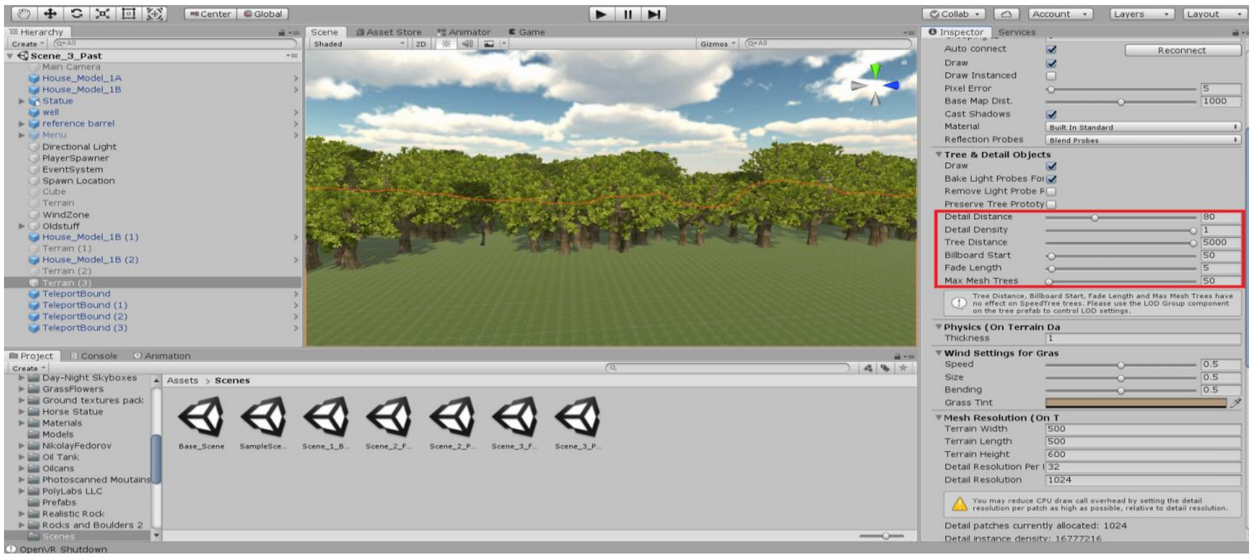
This issue was remedied by changing how the user object was loaded into scenes. Rather than having the user object in the scene from the start, an object was created that is placed in the environment within the scene. This object was then given a script that spawns the user into the scene using the positional data of the object. Upon the initialization of the scene’s objects, the script first checks to see if the user object already exists within the scene using the “Player” tag to find the user object. If the user object does exist within the scene, the user object is moved to the position of the object that the spawning script is attached to. If the user object does not exist, it is instantiated into the scene at the correct position and scale. The object containing this script was then set to not render its mesh or collider within the scene, resulting in an invisible object that the user is unaware of as to not distract them. This allows for the initial scene in the project to instantiate the user object into the scene and any scene that the user changes to from then on will move the user object to the correct location rather than attempting to create the user a second time or having a user object already within the scene. This remedies the “duplicate user” issue and will easily allow the creator of an environment to define the location that the user will be loaded into with a physical object. Below is the code used by the project to create the user object properly.



```
void Awake()
{
    if (GameObject.FindGameObjectWithTag("Player") != null)
    {
        Player = GameObject.FindGameObjectWithTag("Player");
        Player.transform.position = SpawnLocation.transform.position;
        Player.transform.localScale = new Vector3(3.480065f, 3.480065f, 3.480065f);
    }
    else
    {
        Player = Instantiate(PlayerPrefab);
        Player.transform.position = SpawnLocation.transform.position;
        Player.transform.localScale = new Vector3(3.480065f, 3.480065f, 3.480065f);
    }
}
```

Figure 10. User object creation code.

Due to the method that the environments were created with, an issue began to arise regarding the scenery that was rendered using billboards. Billboards are essentially two dimensional graphics placed within the scene that rotate to face the camera rather than a three dimensional object. This saves resources by eliminating the need for a mesh to be rendered for the object while still giving the illusion that the two dimensional graphic is a legitimate object within the environment. When creating several of the example environments, billboards were used to make “background” objects near the outer edges of the environment that did not require fully detailed objects due to their distance from the player. This method was primarily chosen due to the fact that the Unity engine’s terrain objects allowed for both the terrain manipulation required to make the example environments more than a flat plane and the terrain object supported easily “painting” these billboard objects onto the terrain to be used as either trees or grass within the environment. Interestingly, the Unity engine seems to treat these objects somewhat differently when being viewed using the Virtual Reality headset as the scene’s camera rather than a standard Unity camera object. When these objects are at a certain distance from the camera in a scene they are either reduced in detail or unloaded entirely to further save on resources. Through a standard camera object, the distances at which the billboards are no longer rendered are reasonable, with the object being far enough that it could not be seen by the user realistically. This seemingly did not scale well with the Virtual Reality headset’s camera, as the objects would vanish at a distance where they should have been easily visible from. This completely breaks any sense of immersion for the user in the environment as objects would simply stop existing if the user barely backed away from them. I believe that this was caused due to the object representing the user being at a different scale to the standard camera object. Luckily this was easily solved through an increase made to the detail distance that the Unity editor used to choose what level of detail to render the billboards at in an environment. Through this increase, trees would correctly render for the user, allowing the environment to be correctly detailed as to not break user immersion. Below is the settings method used to correctly render the billboards in this project, resulting in the correct rending of trees.



```
Auto connect                ✓                 Reconnect
Draw                        ✓
Draw Instanced
Pixel Error                                   5
Base Map Dist.                                1000
Cast Shadows                ✓
Material                    Built In Standard
Reflection Probes           Blend Probes
▼ Tree & Detail Objects
Draw                        ✓
Bake Light Probes For       ✓
Remove Light Probe R
Preserve Tree Prototy
Detail Distance                               80
Detail Density                                1
Tree Distance                                 5000
Billboard Start                               50
Fade Length                                   5
Max Mesh Trees                                50
Tree Distance, Billboard Start, Fade Length and Max Mesh Trees have
no effect on SpeedTree trees. Please use the LOD Group component
on the tree prefab to control LOD settings.
▼ Physics (On Terrain Da
Thickness                   1
▼ Wind Settings for Gras
Speed                                         0.5
Size                                          0.5
Bending                                       0.5
Grass Tint
▼ Mesh Resolution (On T
Terrain Width               500
Terrain Length              500
Terrain Height              600
Detail Resolution Per I     32
Detail Resolution           1024
You may reduce CPU draw call overhead by setting the detail
resolution per patch as high as possible, relative to detail resolution.
Detail patches currently allocated: 1024
Detail instance density: 16777216
```

Figure 11. Unity billboard render settings.

<u>Environment Toolset</u>

Although the environments that were created to showcase this project were fully functional in regards to what was created for their handling and interactions, the main focus of this project was to provide a toolset of sorts for someone in the archeology field to create their own environment and its variations using a three dimensional scan of an area with a limited knowledge of the inner workings of the Unity engine. For this purpose, the project contains several prefabs and scenes to allow for ease of use.

The first tool provided is a simple scene created for the user to drop their environment scan into. This scene contains the bare essentials for the environment to work properly within the stipulations created by the project, these being a spawn point for the user's Virtual Reality object and a plane that the user is able to move around on. Note though that this plane has been rendered invisible so that it does not interfere visually with the environment scan that will be provided by the user. Along with these, four boundary prefabs are also placed into the scene that the user can simply move to restrict the area that can be explored within the environment. With these provided prefabs and the base scene itself, the user should be able to easily drop their environment scan into the scene, size it appropriately using Unity's resize tool, and have their environment be easily navigable in Virtual Reality. The following is an image of the base scene that will be provided with the project that shows the plane and boundaries' colliders.

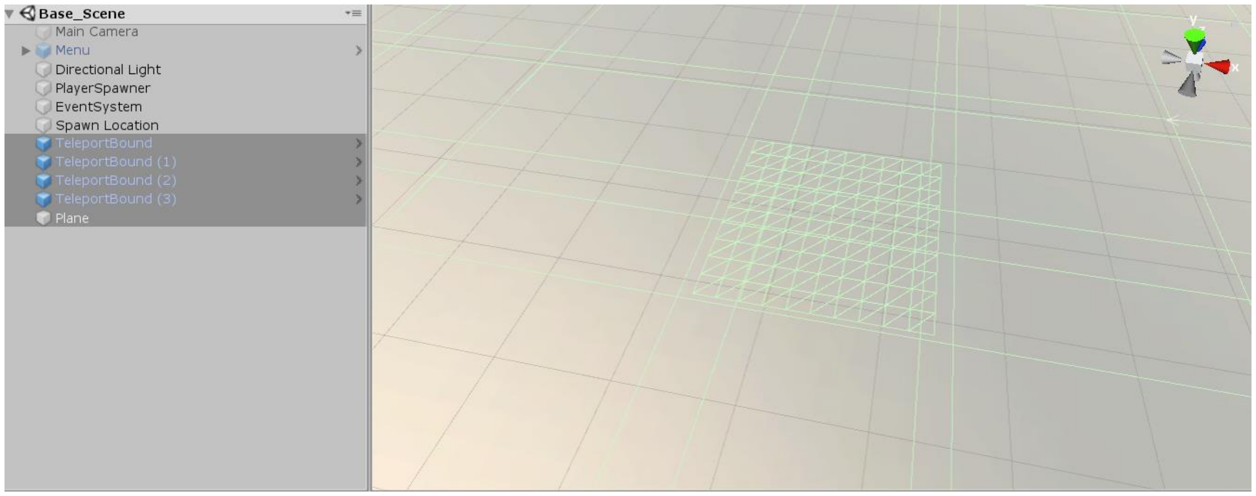
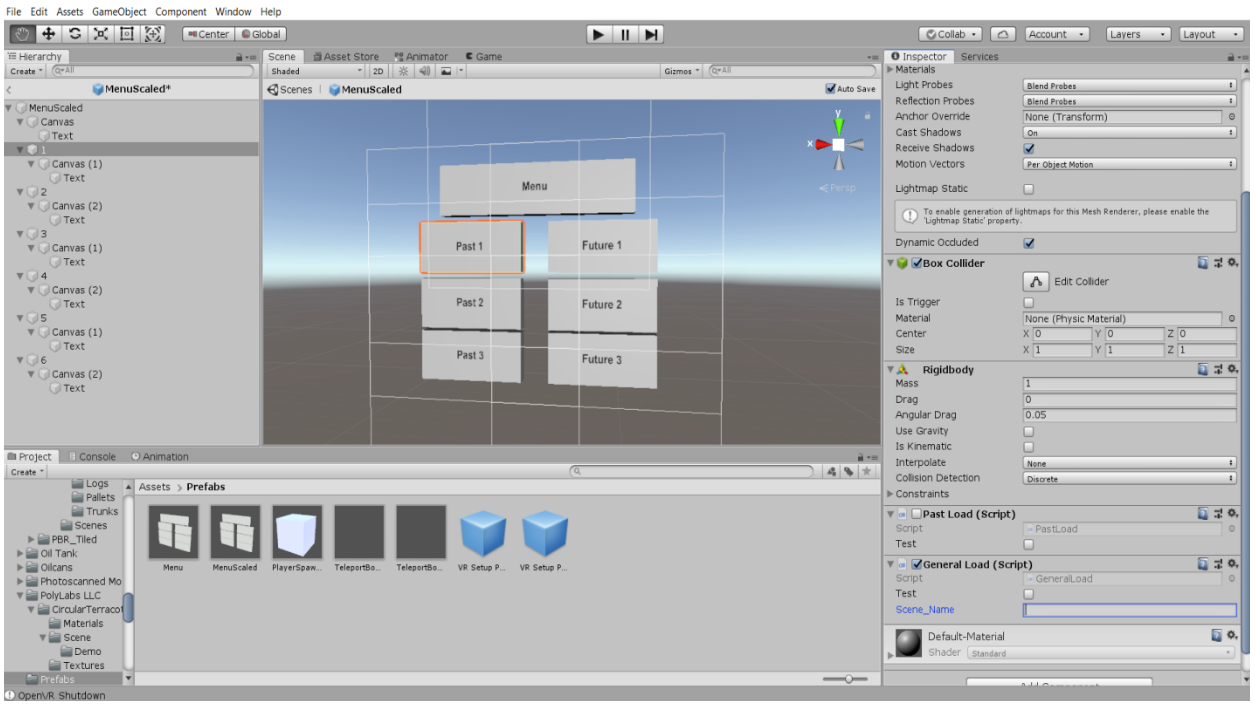


Figure 12. Wireframe base scene.

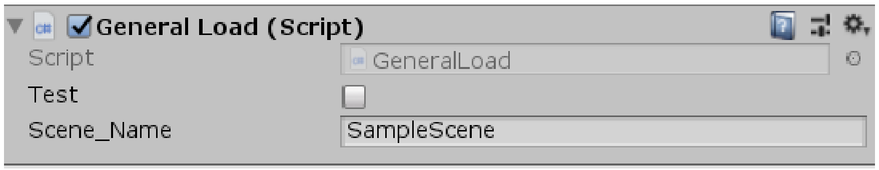
By duplicating the base scene, the user should be able to create two versions of their environment, one as a "past" variation of what the environment looked like during the period of history that is of interest and one as a "present" variation that is representative of how the environment looks currently. Once these two variation have been created they must be added to the scenes listed in Unity's build menu by opening each and clicking "add open scenes" within the build settings. If this is not done, the

Unity engine will hang on attempting to swap between either of the scenes when running the project within the editor and the scenes will not be included when building the project for a standalone release. Along with this, the scenes must be added to the menu prefab for the project so that the user may switch between the environment’s variations when in Virtual Reality. To make this significantly simpler for the user, the code used to switch between environments has been created in such a way that the code itself does not need to be modified to take in the correct scene’s name. Rather than requiring for the code itself to be edited, the string meant to contain the name of the scene to be loaded upon touching the related menu option has been made public. Due to this, the desired scene’s name can be entered into a field on the menu prefab when selecting the section of the menu meant to represent that scene within the Unity editor. To do this, simply open the menu prefab from within the Unity editor, click on the menu option meant to represent the scene, and look at the object’s script component. This has the appropriate field where the file name of the scene must be entered. After entering the correct file name, save the prefab and the scene should be loaded upon selection of that option on the menu from within Virtual Reality. If this is performed for each of the two variations made for the environment, the two scenes will both be easily swappable within the project. Below is an example of the menu prefab being edited to support a scene in the manner stated above.



```
Light Probes          Blend Probes
Reflection Probes     Blend Probes
Anchor Override       None (Transform)
Cast Shadows          On
Receive Shadows       ✓
Motion Vectors        Per Object Motion
Lightmap Static       ☐
To enable generation of lightmaps for this Mesh Renderer, please enable the 'Lightmap Static' property.
Dynamic Occluded      ✓
✓ Box Collider
Is Trigger            ☐
Material              None (Physic Material)
Center                X 0   Y 0   Z 0
Size                  X 1   Y 1   Z 1
Rigidbody
Mass                  1
Drag                  0
Angular Drag          0.05
Use Gravity           ☐
Is Kinematic          ☐
Interpolate           None
Collision Detection   Discrete
☐ Past Load (Script)
Script                PastLoad
Test                  ☐
✓ General Load (Script)
Script                GeneralLoad
Test                  ☐
Scene_Name
Default-Material
Shader                Standard
```

Figure 13. Initial menu settings.



```
✓ General Load (Script)
Script        GeneralLoad
Test          ☐
Scene_Name    SampleScene
```

Figure 14. Entered scene name text.

Through these basic tools the user should be able to create an environment or a set of environments that can be explored within Virtual Reality to provide a better sense of scale. If the user wishes to do more than a simple environment though, say an environment that has objects within it as the example scenes that were created for this project do, more prefabs have been provided. These consist of a few basic colliders that have been tagged using the "NoTel" tag so that the user will not be able to teleport onto the object in question. These can be resized appropriately to give an object within the scene, such as a statue, collision to prevent the user from teleporting onto it. These are provided in the “Prefabs” folder within the project. All of the toolset prefabs and base scenes are cataloged in the images below for documentation purposes.

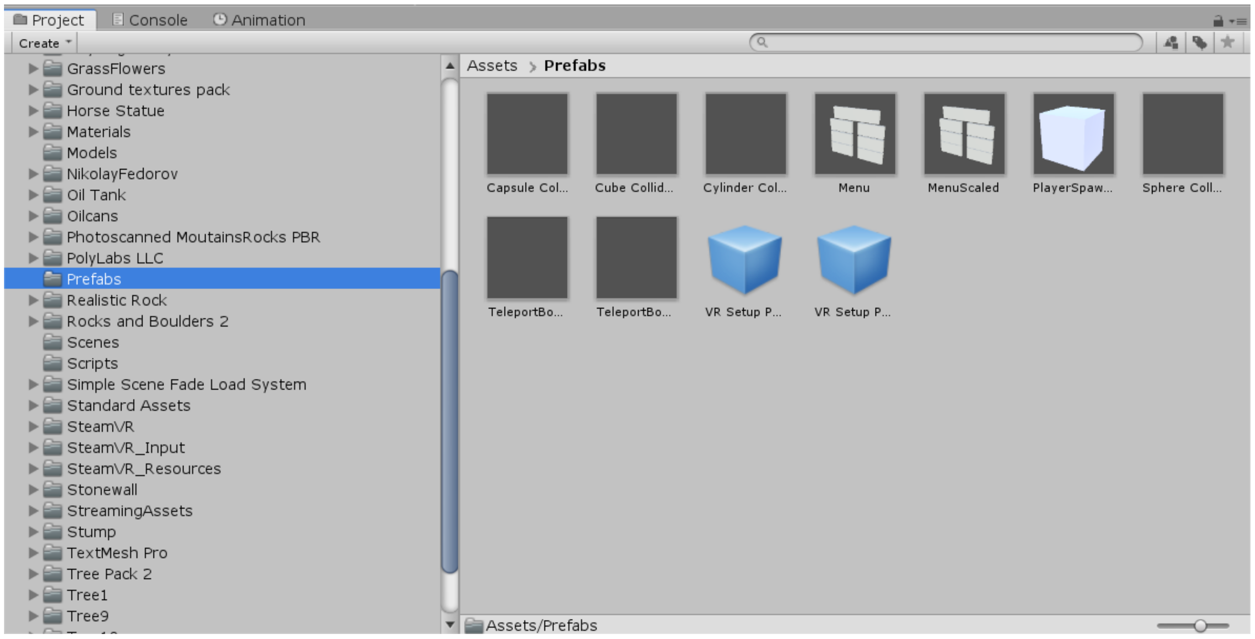


Figure 15. Prefabs provided with the toolset.

CHAPTER V

CONCLUSION

This project has provided for an extremely interesting way to blend my major in Computer Science with my interest in developing a Virtual Reality application. The project itself has turned out quite well and I believe that the toolset I have created would be incredibly useful to anyone who wishes to implement their environment in a way that provides the user with a method to explore its variations and truly show the scale and depth of the environment. The basic assets for this toolset have been uploaded to GitHub at https://github.com/rwd22/VR-Environment-Toolset/ and the SteamVR SDK is available for download in the Unity Asset Store. Along with this, I believe that through the creation of this project I have gained valuable experience regarding the development of a Virtual Reality application, especially seeing as the industry for Virtual Reality is continuing to grow in both the professional and consumer spaces.

CHAPTER VI

FUTURE WORK

In regards to what may be done to improve this project in the future, a different method of restricting the user's movements may be beneficial. As stated previously, due to the way that Unity handles collisions, the user is able to force their way past the boundaries if solid ground exists beyond them. This is primarily to prevent the user from becoming stuck within a collider if they manage to break into one, but I believe that a visual cue may be added to guide the user back to the intended space if they do break out, such as darkening the user's view when in this area when they are not facing the intended space. The menu could also be expanded upon in the future to allow for more spaces to add environment variations while keeping the interface compact and manageable. Overall, I am extremely pleased with the progress that I have made on this project and with the amount of experience I have gained regarding Virtual Reality over the course of its development.

BIBLIOGRAPHY

Kerckhove, E. V. de, & Kerckhove, E. V. de. (n.d.). HTC Vive Tutorial for Unity. Retrieved from http://www.raywenderlich.com/9189-htc-vive-tutorial-for-unity

Technologies, U. (n.d.). MonoBehaviour.Awake(). Retrieved from https://docs.unity3d.com/ScriptReference/MonoBehaviour.Awake.html

Technologies, U. (n.d.). Terrain settings. Retrieved from https://docs.unity3d.com/Manual/terrain-OtherSettings.html

Technologies, U. (n.d.). LayerMask. Retrieved from https://docs.unity3d.com/ScriptReference/LayerMask.html

Technologies, U. (n.d.). SceneManager.LoadScene. Retrieved from https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.LoadScene.html

Technologies, U. (n.d.). Physics.Raycast. Retrieved from https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

Technologies, U. (n.d.). Object.Instantiate. Retrieved from https://docs.unity3d.com/ScriptReference/Object.Instantiate.html

Technologies, U. (n.d.). Collider.OnTriggerEnter(Collider). Retrieved from docs.unity3d.com/ScriptReference/Collider.OnTriggerEnter.html