AARHUS SCHOOL OF ENGINEERING

DOCUMENTATION

BACHELORPROJECT F2018

An Investigation of Virtual Reality Platforms with Focus on Object Manipulation

Authors
K. RIEDER
D. V. JENSEN

Supervisors
Kasper LEAFCASTLE
Henrik KIRK

March 2, 2019

Contents

1 Introduction

2 Architecture

3 Unity

4 Unity Terminology

As a big part of our project have been developed using Unity, a lot of Unity-specific terminology appears throughout our documentation. Therefore, the following chapter will clarify some of the most essential concepts of Unity that have been used in our project in order to establish a common understanding of them.

4.1 Scenes

Scenes in Unity are the 3D space in which you create everything that will be in your game [**?**]. Essentially they represent levels. They are files in the Unity asset folder. You drag GameObjects into scenes.

4.2 GameObjects

A GameObject in Unity is a basic object used within scenes to represent anything imaginable. [**?**] Thus, GameObjects are used to represent objects such as trees, houses, furniture, characters, and so on.

GameObjects by themselves do not do anything. They are considered to be containers of scripts. This means that all custom behaviour that we have created during this project is programmed in scripts, which are then applied to GameObjects within scenes.

Figure 1 shows a game console controller from one of our scenes. To the left of the editor it is possible to view all GameObjects within the open scene. To the right, in the Inspector, is an overview of all scripts applied to this GameObject, in this case only a simply *Transform* script.

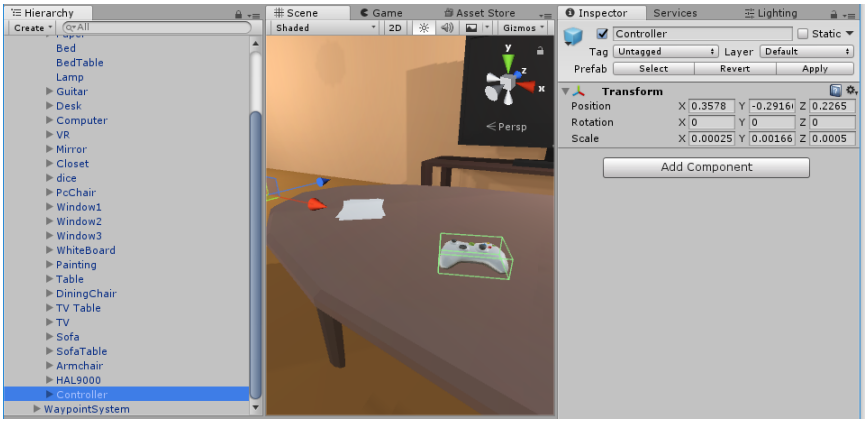

Figure 1: A game controller GameObject selected within one of our scenes.

4.3 MonoBehaviour

All scripts applied to GameObjects are essentially C# classes. These classes inherit from the base class *MonoBehaviour* [**?**]. Every script in Unity inherits from this class. MonoBehaviour is used by the Unity game engine to call predefined methods which developers can then fill out with custom logic. The predefined methods most typically used within our project has been: *Start*, *Awake*, and *Update*.

Figure 2 shows a screenshot of an empty script with these typical methods included.

```
using UnityEngine;

public class ExampleScript : MonoBehaviour {
    void Awake()
    {

    }

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

Figure 2: An example script. Here it is possible to see how the script is a class inheriting from MonoBehaviour.

4.4 Prefabs

Prefabs is a mechanism within Unity that makes it possible to easily re-use GameObjects. [**?**] Typically, you will end up configuring complex GameObjects used multiple times within one scene, or shared across many scenes. Were you to simply copy-paste GameObjects in order to achieve this, the problem would be that changes to one of these GameObjects would not be reflected across all of them. Thus, Prefabs can be equated to the concept of classes in object-oriented programming. You define one prefab representing a specific configuration of a GameObject, and this prefab can thus be instantiated many times. If you then change the Prefab, such as adding new scripts or changing the values of scripts in the inspector, these changes will be reflected across all GameObjects originating from this Prefab.

4.5 Inspector

The Inspector is a window of the Unity editor where it is possible to inspect the configuration of GameObjects within a scene [**?**]. Figure 3 shows a screenshot of the inspector after having selected a game controller GameObject within a scene. Here, it is possible to see what scripts that are applied to the GameObject - such as *RigidBody*, *Box Collider*, etc - as well as the values assigned to public attributes of those scripts.

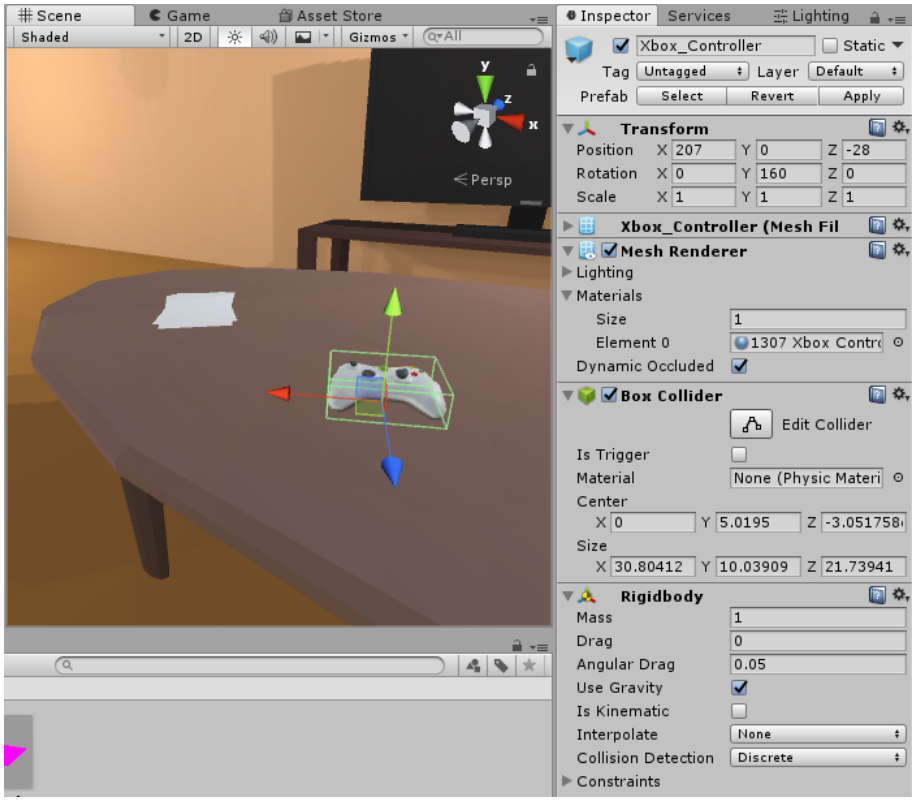

Figure 3: A screenshot of the Inspector, showing the configuration of scripts and script values from a selected GameObject from a scene.

4.6 Play Mode

The Play Mode is a feature within the Unity editor. It simply allows you to instantly play the scene you are currently working on [**?**]. This way, you avoid having to constantly do builds to executable files in order to try out your work.

4.7 Unity Packages

Unity packages - also sometimes referred to as Asset Packages - is a way to take Unity assets, such as sounds, 3D models, materials, scripts, and any other type of assets, and

package them into an easily redistributable format which can either be re-used internally between multiple projects or shared with others [**?**]. Through the Unity editor it is easy to import Unity packages.

4.8 Asset Store

The Asset Store is a built-in application of the Unity editor. It allows you to browse third-party packages which can add various assets to your projects [**?**]. This could be things such as scripts for achieving specific features, but could also be auditory or graphical assets such as sound effects, music or 3D models.

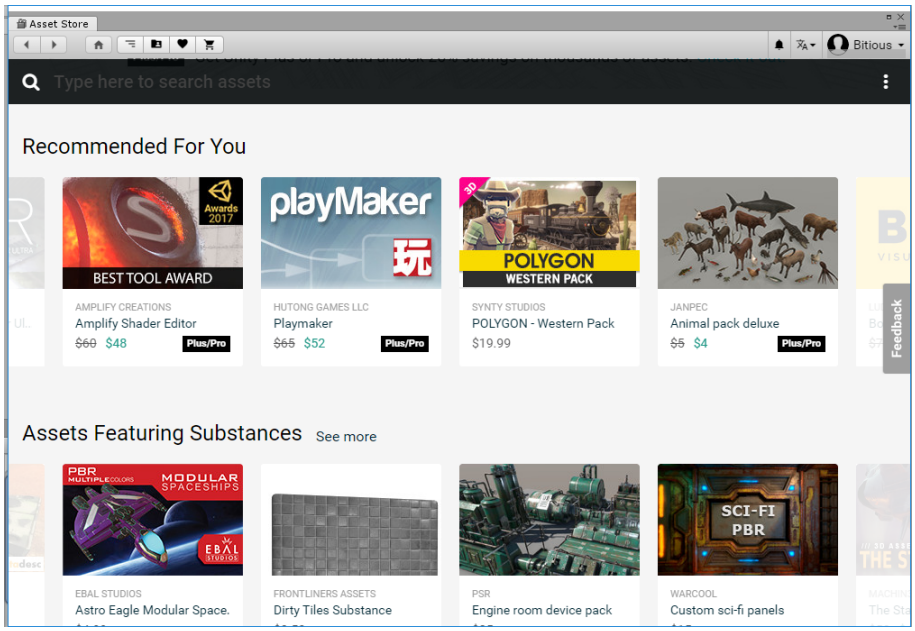

Figure 4: A screenshot of the Asset Store seen from the Unity editor.

5 Measurement Tool

This section will cover the measurement tool that is used to fetch and display data from the Firebase Real-Time database.

5.1 Choice of Technology

Google have made the Firebase database easy to use on the Android platform, making data easy to get, and more importantly, keep up to date when changes or additions are made. This is not easily done on other platforms such as a windows WPF application or a website. For this reason we have decided to implement the measurement tool as an android application.

5.2 Project Structure

Inspired by broaster structure

5.3 UI

Fragments Activities (herunder Tab Layout) Screenshot of app

5.4 Class Diagrams

Activties -> fragments -> measurement stuff Repositories Model classes (measurements)

5.5 Sequence Diagrans

Basically firebase diagrammerne Repository diagram

5.6 Unit Tests

Rieder er en dumdum

6 Virtual Reality Applications

This project consists of various virtual reality applications for IVR and MVR, ranging from the navigation experiments conducted during the project to the final experiments conducted at the end of the project.

All of these virtual reality applications were developed using Unity.

This section will describe how we used Unity to develop the applications, and will detail the technical solutions we implemented in order to create them.

6.1 Project Structure

All of the VR applications were developed within a single Unity project.

The reason for developing all of the VR applications within a single project was in order to ease the development workflow. All of the applications, whether IVR or MVR, ultimately had to share many of the same assets. For example, as it was important to create the same virtual environments on both platforms, all of the applications have to share the same 3D models for furniture and other environmental objects. Also, much of the codebase applies equally well across both platforms, thus much of the code would also have to be shared.

Because of these reasons, we decided that it was advantageous to contain all applications within the same project, in order to avoid having several projects duplicating a lot of assets. This could quickly turn into a maintenance nightmare when having to keep all shared assets in sync throughout all projects. Having everyting contained in a single project, sharing all assets, was seen as the optimal route.

One disadvantage of one single project, however, is that it quickly turns big. Thus it was important for us to keep a neat project structure, where it was easy to navigate and find what you were looking for. It was also important to handle cross-platform development in a simple manner, so as to not complicate the project structure unnecessarily.

We did this primarily by: Keeping a folder structure throughout the entire project that was as easy to navigate as possibly, and by doing cross-platform development by splitting up each experiment into two scenes. One scene for each platform.

This section will describe this folder structure and scene structure.

6.1.1 Folder Structure

The way that Unity deals with assets - assets being anything from 3D models, audio, third party plugins, textures, scripts and so forth - is by letting the user contain it all within a predefined folder by the name of *Assets*. Within this folder, it is up to the user to decide how to structure all created and imported assets.

Figure 5 shows our structure from the root of the asset folder.

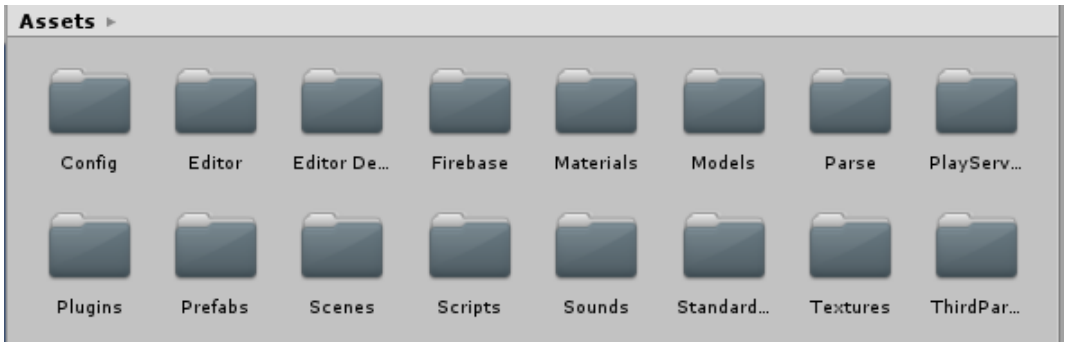

Figure 5: A screenshot of our project structure from the root of the Asset folder.

As can be seen in 5, we decided to split up the various assets into telling categories. This means that we have subfolders for assets such as *Textures*, *Scripts*, *Models* and so forth. This way, it was easy for us to navigate an otherwise big Unity project.

All of these subfolders are then further divided into telling categories, if it makes sense to do so. As an example, figure 6 shows the subfolder of our scripts.

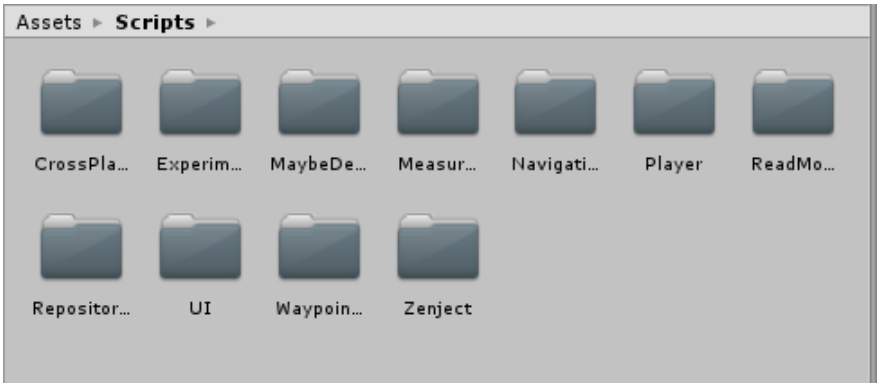

Figure 6: A screenshot of our project structure from the root of the Asset folder.

As can be seen in figure 6, scripts have then been categorized after functionality or domain.

6.2 Third Parties

6.2.1 Zenject

Zenject[**?**] is a dependency injection framework that we use for several aspects of our virtual reality applications.

Zenject is specifically designed and made for Unity.

We use Zenject primarily in order to ease the making of cross-platform friendly scripts, in order to make our software easier to extend and develop.

In order to make use of Zenject, we simply imported it as a third-party package from the Unity asset store. Figure 7 demonstrates Zenject as seen from the asset store within the Unity editor.

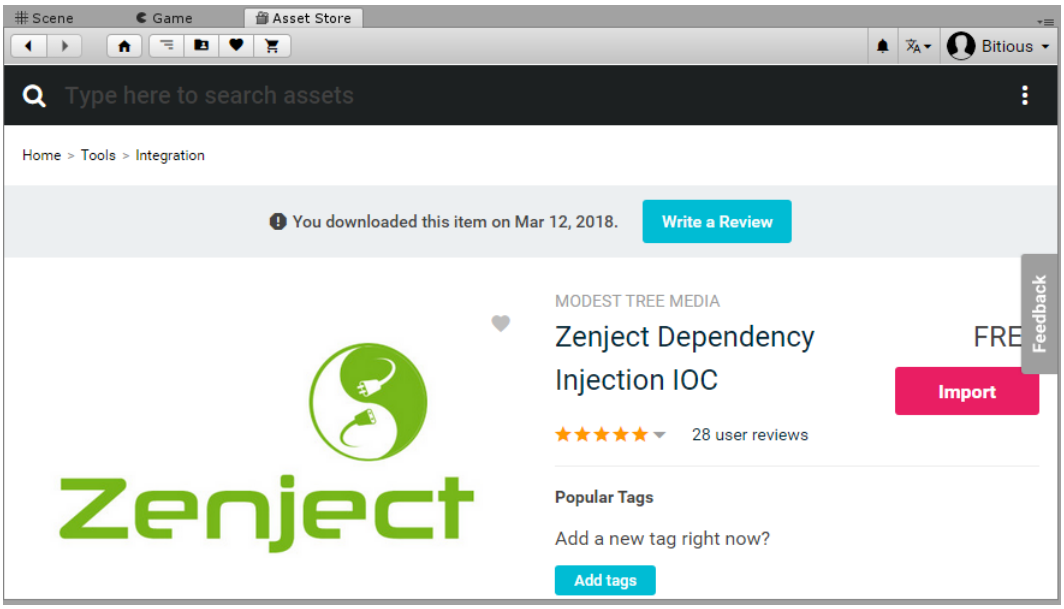

Figure 7: A screenshot of Zenject as seen from the asset store.

In order to get Zenject to work, we made use of two basic Zenject components:

- A *scene context* GameObject within any Unity scene that has scripts depending on Zenject. This GameObject is provided by the Zenject framework.
- A *MonoInstaller* script. This script configures the Inversion Of Control (IoC) container. It is here you decide which concrete implementations should be mapped to which interface type. You write this configuration yourself.

Thus, we implemented our own MonoInstaller script which configures the IoC container for our own needs.

Our MonoInstaller script goes by the name of *MainInstaller*, and figure 8 shows a basic class diagram of it.

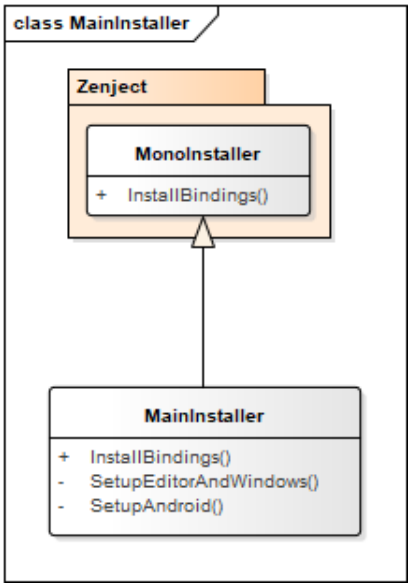

Figure 8: Our own implemenation *MainInstaller* of the *MonoInstaller* component from Zenject. MainInstaller configures our IoC container.

Figure 9 shows a flow diagram of the configuration process.

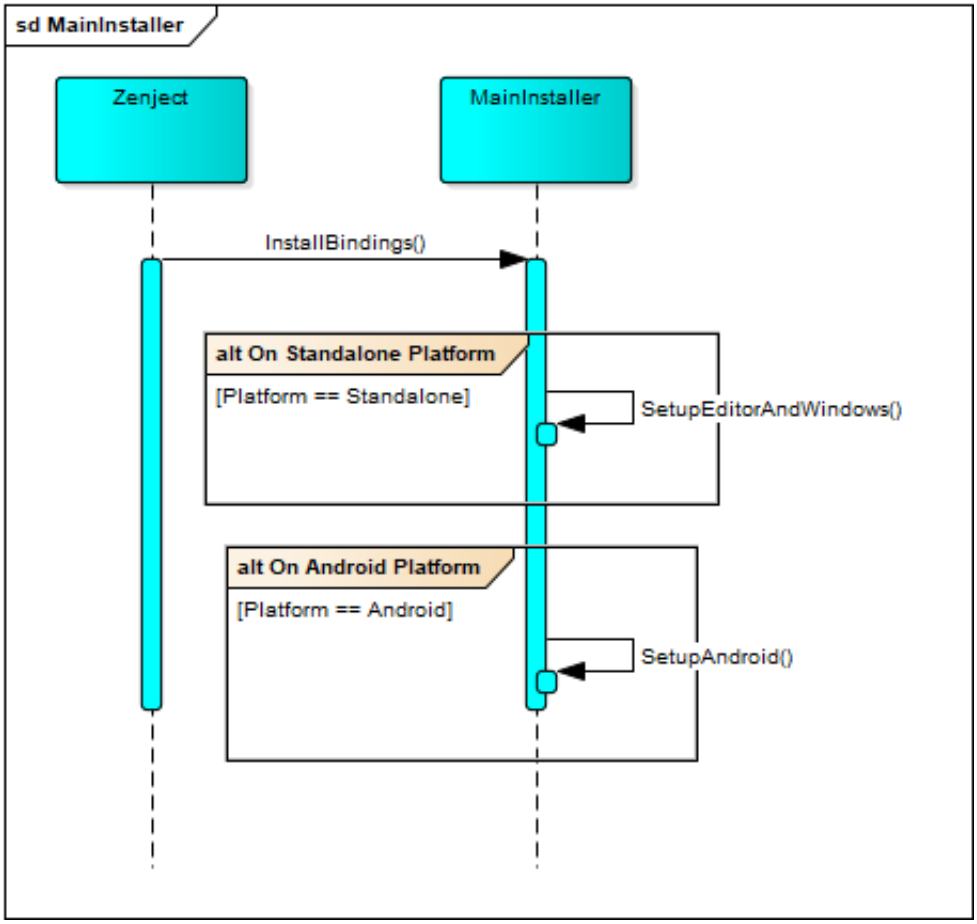

Figure 9: The sequence logic of *MainInstaller*.

Figure 9 thus shows how we use Zenject in order to control platform-specific implementations across platforms. In this diagram, the platform *Standalone* refers to IVR. Thus, if an application is run on a desktop computer, the IoC will be set up with implementations which works best for a desktop environment. Likewise, if the application is execute on an Android smartphone, the IoC will be set up with implementations which works best for a mobile environment.

6.2.2 Google VR SDK

In order to develop virtual reality applications for Google Daydream, we made use of the official Google VR SDK for Unity with Android [**?**].

The Google VR SDK for Unity is a Unity Package which is simply imported to a project through the editor.

The package contains various scripts, prefabs and example scenes relevant to virtual

reality with Google Daydream, which one can make use of and learn from. Furthermore, there is also online documentation available as a learning resource [**?**].

Prefabs and scripts used from the Google VR SDK will be mentioned in relevant sections throughout the documentation.

Instant Preview

The Google VR SDK comes bundled with prefabs that supports *Instant Preview*. Instant Preview allows you to use the Play Mode within the Unity Editor to instantly stream the game to a smartphone connected to the computer through a USB. Throughout development of the MVR application we made heavy use of this feature. Figure 10 shows a picture demonstrating Instant Preview.

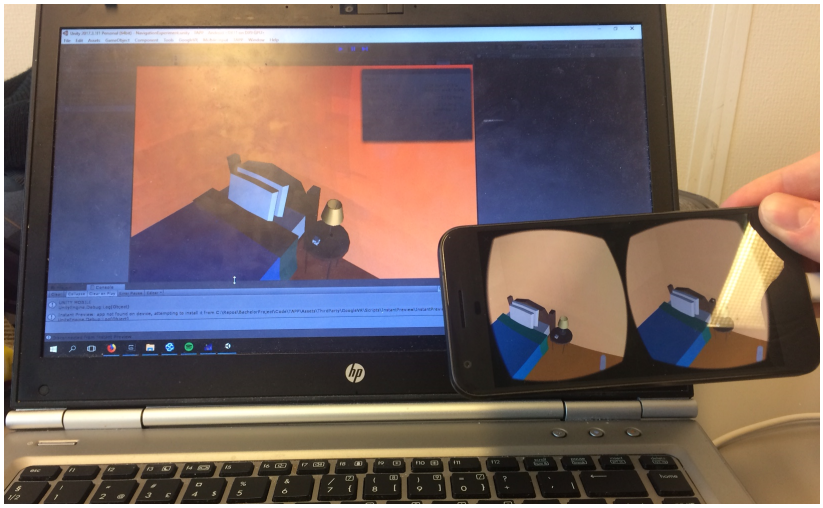

Figure 10: Instant Preview streaming the rendering of our virtual reality application from the Play Mode in the editor to the display of an Android smartphone. The smartphone is connected to the computer through USB.

In order to make use of Instant Preview, we simply needed to include the following prefabs from the Google VR SDK within the scene in question:

- *GvrEditorEmulator*
- *GvrInstantPreviewMain*

Instant Preview helped us tremendously throughout development, as building the APK to the actual device can be time consuming. Being able to instantly test features or bug fixes was really helpful.

6.2.3 SteamVR Plugin

In order to develop virtual reality applications for the HTC Vive, we made use of the *SteamVR Plugin* Unity Package [**?**], which is available from the Unity Asset Store.

Figure 11: The SteamVR Plugin package as seen from the Unity asset store.

SteamVR Plugin for Unity is a Unity package which wraps Valve's OpenVR SDK [**?**] so that it is useable from within Unity. It is not designed specifically for HTC Vive. Rather, it can be used to develop virtual reality applications with various different IVR hardware, such as HTC Vive or the Oculus Rift. The package abstracts the hardware, hiding it behind a common interface which can be used from within Unity scripts.

The package includes a host of scripts and prefabs which can be used for virtual reality applications. It also comes bundled with various demo scenes which you can use to learn the different aspects of the package, and how to use it with IVR hardware.

Prefabs and scrips used from the SteamVR Plugin package will be mentioned in relevant sections throughout the documentation.

Instant Preview

The SteamVR Plugin supports *Instant Preview*. This allows the game to be streamed to the VR headset straight from Play Mode in the Unity editor.

Just as for Instant Preview of the Google Daydream SDK, we relied heavily on the feature for IVR development. It was very quick and easy to test out new features and changes instantly.

There was nothing particular that had to be done in order to make use of Instant Preview with the SteamVR Plugin. As soon as the scene was configured with a SteamVR camera, Instant Preview could be used provided a headset is connected to the development computer.

Experiences with SteamVR Plugin

One peculiar thing we found with the SteamVR Plugin was the lack of documentation. The package itself comes included with a bare-bones *Quickstart* guide. This guide only contains short texts about simple concepts and a few of the prefabs included in the package. There is no API documentation, and many of the scripts and prefabs included within the package is completely undocumented.

Thus we spend quite a bit of time simply figuring out how to use the package properly to implement the features we needed, such as getting teleportation to work, or reading input from the controllers.

6.2.4 ProBuilder

ProBuilder is a Unity editor extension which enables support for doing 3D modelling and texturing from within the editor.

ProBuilder is a Unity package which can be downloaded and imported from the Unity Asset Store.

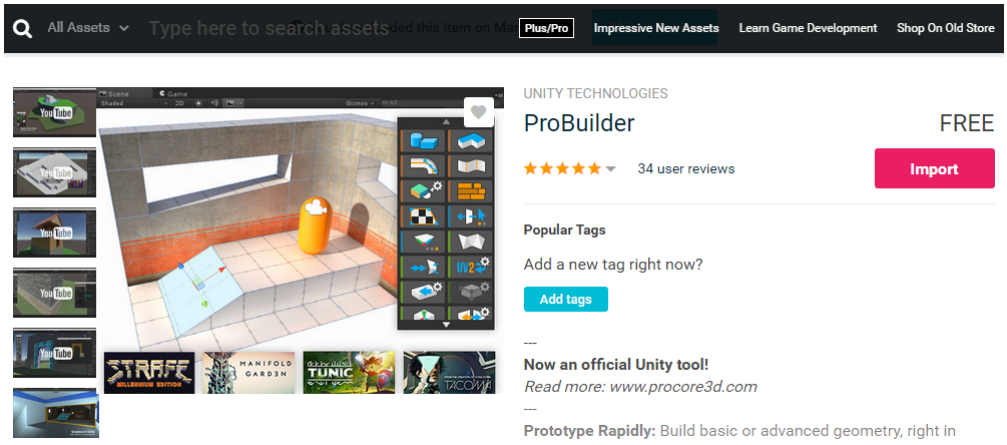

Figure 12: The ProBuilder editor extension as seen from the Unity Asset Store.

During development of the project we had the need to create basic virtual environments which experiments could be framed within. An example of this is the room used for the navigation experiment.

We could have also used external modelling software, such as *Blender* [**?**]. However, we shared no modelling experience within the team. Furthermore, the virtual environments which had to be created during the project were very simple. Therefore there was no need for any complicated external tools, when everything that we needed could be done within the environment which we already used in a greatly simplified fashion. The

ProBuilder editor extension was thus a great match for us, as it was simple to use, and we could integrate the modelling into our already existing Unity workflow.

When ProBuilder is installed in the Unity project, its features becomes available through the addition of a new toolbar option. This can be seen in the screenshot of Figure 13 which is taken from within our own project.

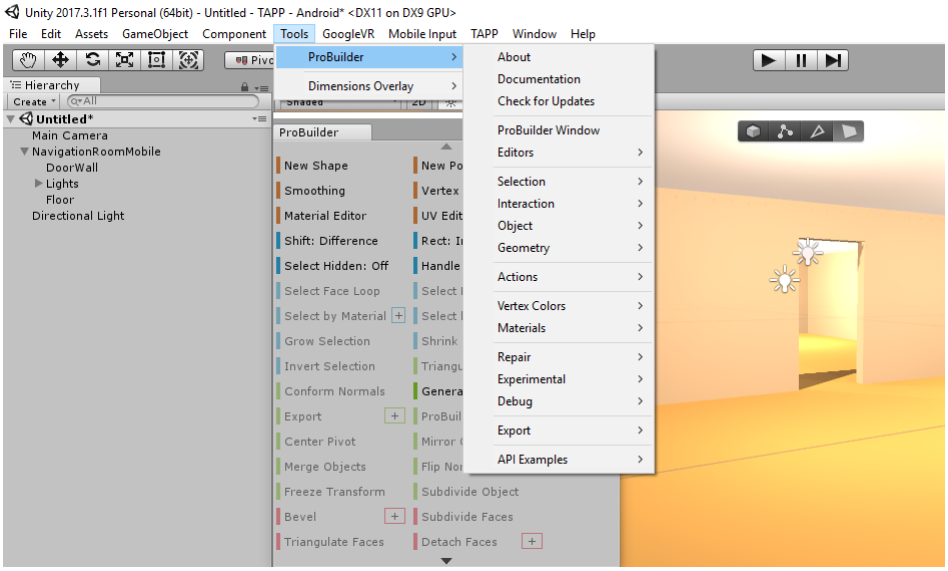

Figure 13: The new ProBuilder menu option after installing the ProBuilder package in our project.

6.2.5 3D Models

In the VR applications we needed 3D models for furniture, but creating them ourselves from scratch would take a lot of time, and since 3D modeling is not the focus of this project, we decided to make use of free models provided on the platform Poly, which provides user-created low-poly 3D models under a CC-BY license. As such all of our furniture and furnishings are models created by the user *Poly by Google* [**?**] under the CC-BY 2.0 license [**?**].

Poly is a platform that is accessed on the web, and figure 14 shows a screenshot of it. As can be seen, you can easily search 3D models by keywords.

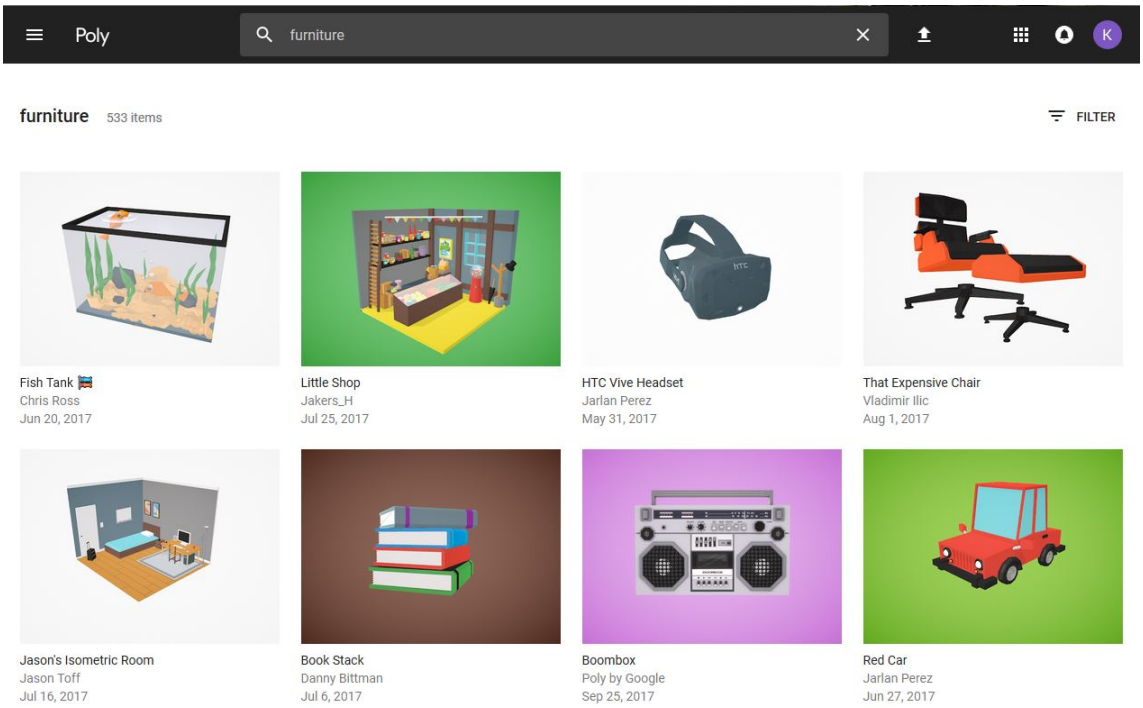

Figure 14: A screenshot of the web interface of Poly. A search for furniture has been completed, and the selection is shown.

6.3 Unit Testing

Unit Testing has been done of scripts within our Unity project where it makes sense to do so.

In order to unit test we made use of Unity's in-built unit testing tool from within the editor.

Figure 15 shows a screenshot of the in-built unit test tool in our project.

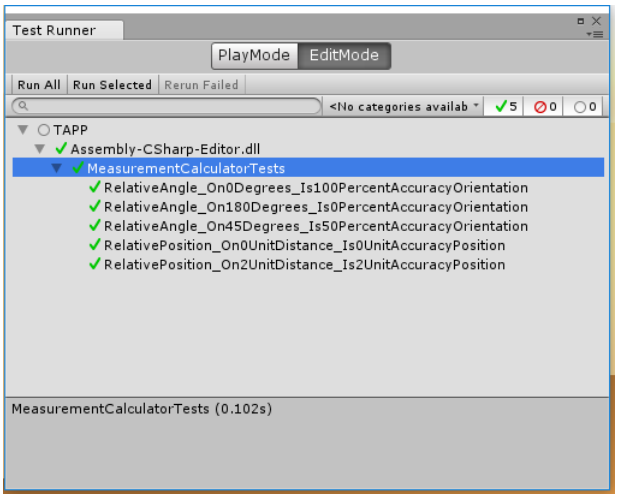

Figure 15: Unity's in-built test runner seen within the context of our project. Tests discovered within our project is shown within the window.

In order to implement unit tests with Unity, you have to place the scripts within a specific folder relative to the root of the *Asset* folder. The folder has to be named *Editor*, and should be placed in the root of the *Asset* folder [**?**]. This can be seen in the screen of figure 16.

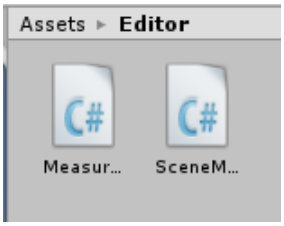

Figure 16: The *Editor* folder as seen from within our Unity project. Our unit test scripts is placed within this folder.

When unit test scripts are placed within this folder, Unity's test runner will automatically detect them and display them in the test runner window.

Unity has integrated the NUnit [**?**] test framework. Thus, unit tests are written using NUnit.

The unit tests of our Unity project is also used in associated with our continuous integration pipeline.

6.4 Continuous Integration

We have used continuous integration for our Unity project.

The primary purpose of continuous integration for us has been for monitoring the health of the VR applications during development. If a team member accidentally pushes changes that breaks the build across platforms, the continuous integration pipeline will detect it and notify us.

In order to make use of continuous integration, we used Unity's own cloud service, called *Cloud Build* [**?**].

The advantage of Cloud Build is that as it's Unity's own service, it is already well-integrated with Unity and thus easy to set up and use. It provides direct integration with the unity Editor, and also features a web-interface for detailed settings and histories.

On figure 17 is a screenshot of Cloud Build from within the editor, in the context of our project.

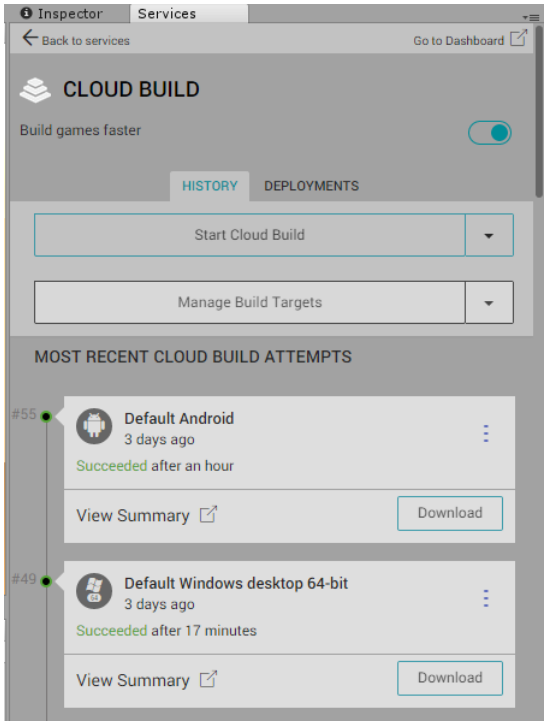

Figure 17: The Cloud Build service as seen from our project within the Unity editor.

Cloud Build from within Unity's editor gives a quick overview of the most recent builds performed, and whether they succeeded in building or not. In figure 17 it is possible to see that the most recent builds at the time was the Android platform target, and Windows Desktop platform target. It is also possible to see that they both succeeded in building.

On figure 18 is a screenshot of the detailed web-interface typically used in conjunction

with the Unity editor.

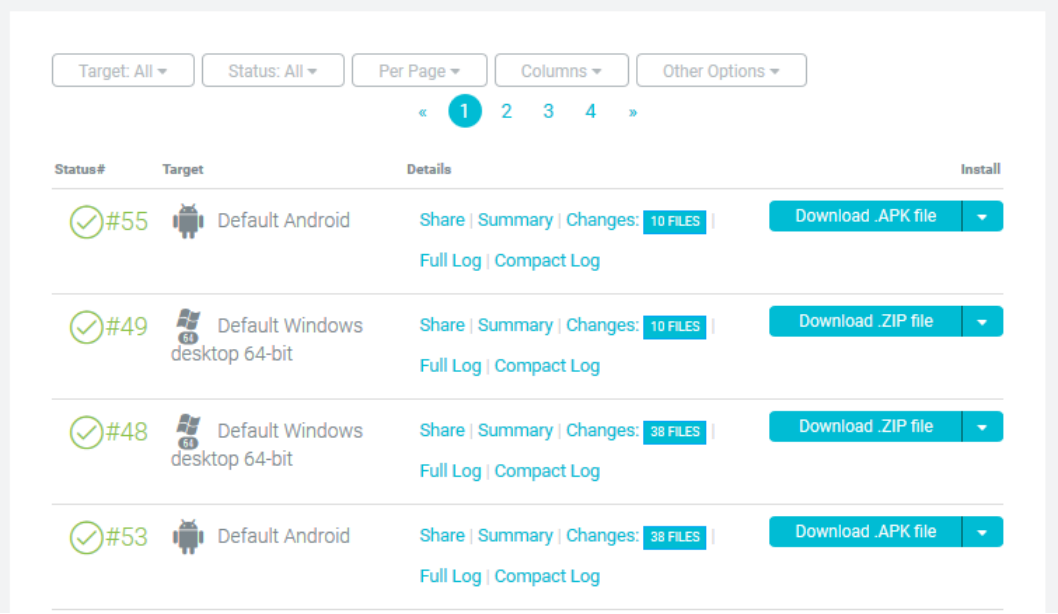

Figure 18: The Cloud Build service as seen from our project through the web-interface.

As can be seen in figure 18, the Cloud Build web-interface has a lot of features. Besides having a detailed build history of all builds ever completed, Cloud Build also generates change logs and full logs of the entire build process. Additionally, the generated executable file (whether it be an APK file for Android of EXE file for Windows) is also uploaded so as to be easy to download and share across devices.

Every time a commit is made on our GitHub repository, Cloud Build detects the changes of our Unity project and attempts to build the project and run any detected unit tests. Whether builds and tests succeed or fail, an email notification is sent with the results. An example of such an email notification can be seen in figure 19.

Figure 19: Screenshot of an email notification sent automatically from Cloud Build after a succeeded Android build.

6.5 Modelling of Virtual Environments

We used ProBuilder in order to model the virtual environments from within our experiments were framed. This includes the two-roomed environment used in the navigation experiment, as well as the object manipulation experiment.

During the project, we modelled two virtual environments. They are:

- The room used in the navigation experiment.
- The room used in the object manipulation experiment.

Figure 20 shows a screenshot of the virtual environment for the navigation experiment.

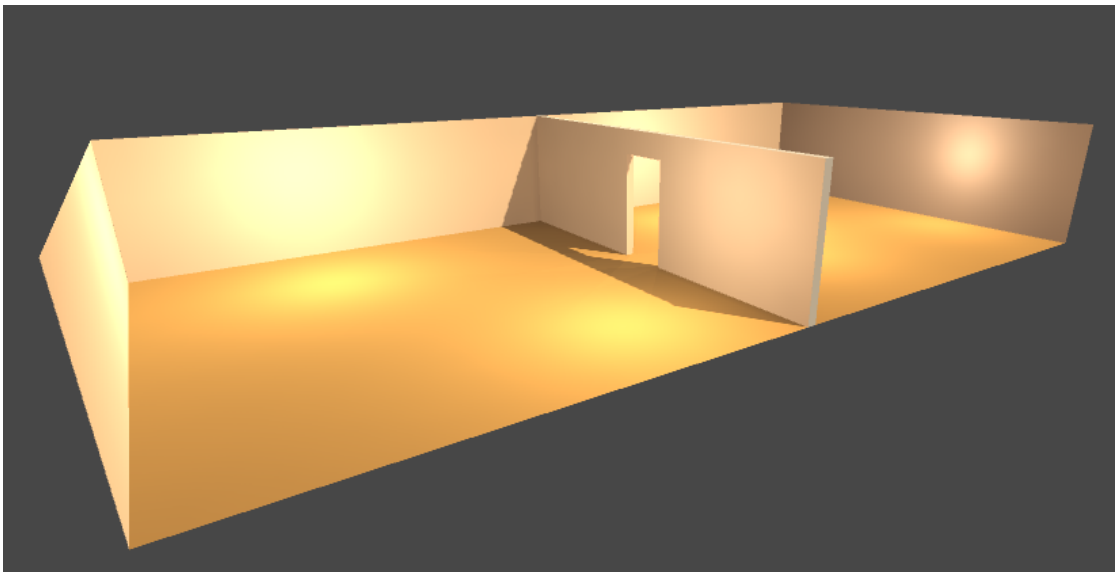

Figure 20: The virtual environments that we made using ProBuilder during the project.

Figure 21 is a screenshot from our project in which the ProBuilder window can be seen.

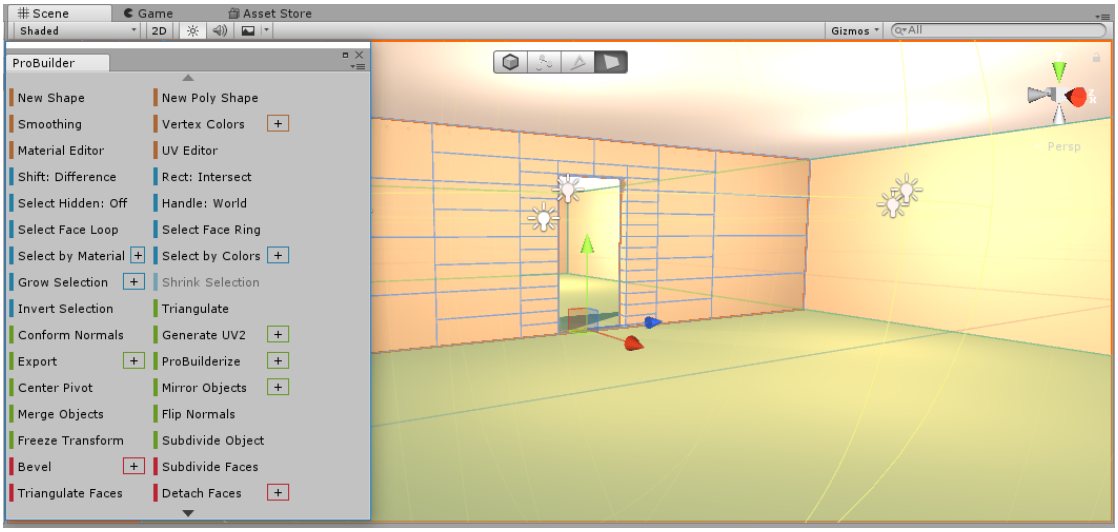

Figure 21: The ProBuilder window open in a scene of ours, with the room we modelled for the navigation experiment. Here, face selection is used to manipulate individual faces of the model (in this case the highlighted wall).

ProBuilder supports edit modes which you find in conventional 3D modelling software, such as: Object Selection, Vertex Selection, Edge Selection and Face Selection. For example, we made use of Face Selection in order to edit the wall in figure 21 so as to

make a door in the middle.

Another feature of ProBuilder that helped us greatly, as we were inexperienced in 3D modelling in general, was the easy-to-use texturing feature.

Usually it can be quite involved to texture 3D models, requiring knowledge of how to perform UV mapping [**?**]. ProBuilder made this easy for us, so that we could quickly create environments with colors. Texturing in ProBuilder is done through the *Material Editor*. Figure 22 shows how, when a wall is selected in our virtual environment, we are able to easily apply a specific material to that section of the model only.

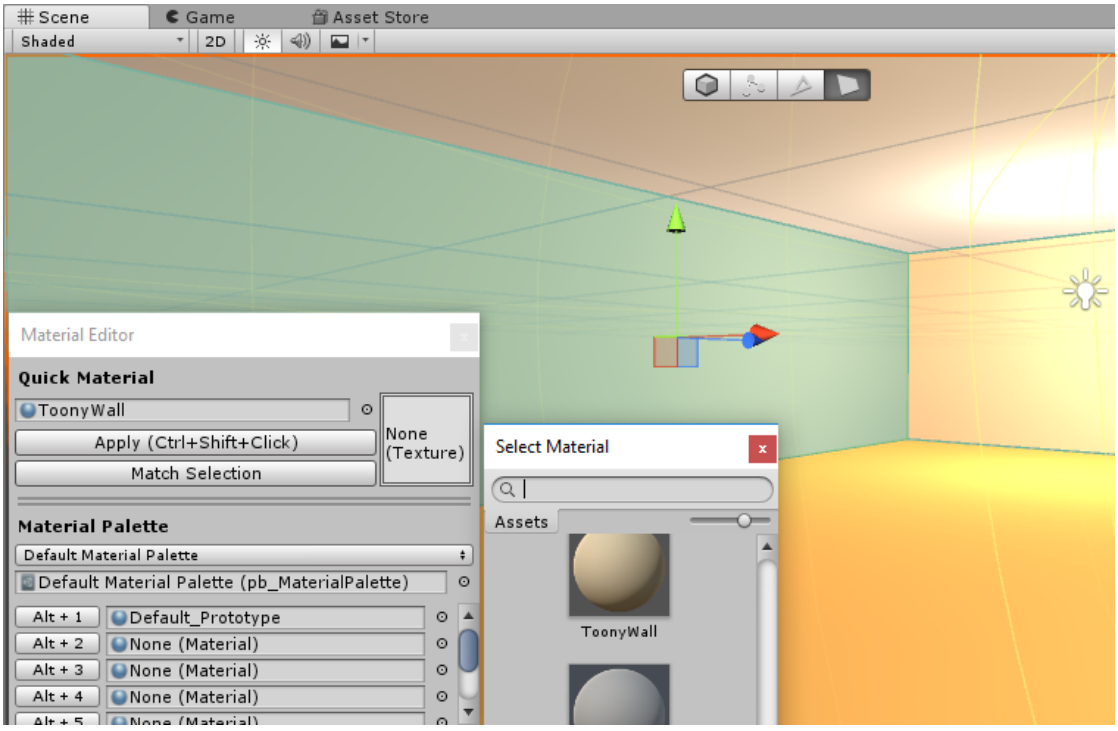

Figure 22: The Material Editor of ProBuilder. Here, we have selected a wall on the room of the virtual environment used for our navigation experiment. From here, we can easily apply new materials.

Using this feature meant that we did not need to have knowledge of UV mapping.

6.5.1 Reuse

It was important that the virtual environments we modelled could be re-used across multiple scenes.

7 The Navigation Experiment

As part of our development process, we wanted to eliminate uncertainties that may arise when people are newcomers in virtual reality. Specifically we wanted to make navigation in a VR environment as good as possible, as navigation in VR often have the unfortunate side effect of inducing nausea or motion sickness in users.

From looking at current implementations navigation in VR applications, we have seen that the most common methods of navigation is teleportation and gaze-based walking. In the article *Interaction Design for Mobile Virtual Reality* by Daniel Brenners [**?**], it was discovered that users preferred gaze-based walking on the mobile platforms, but after trying that method of navigation ourselves and feeling extreme nausea, we decided to make our own user-test to see which navigation mode the users prefer in order to optimize the final VR experiment.

This section describes the implementation and execution of our navigation experiment.

7.1 Experiment Task Setup

In order to test the navigation modes we created an experiment in which the user have to navigate through multiple "waypoints" in a Virtual Reality Environment.

Figure 23: Screenshot from within the navigation experiment.

Using a menu (see figure 24), the user can choose the navigational mode of the experiment; either teleportation og gaze-based walking. After making this choice, the

user must navigate through the waypoint system, and when done with one navigation method, they should do the same for the next navigation mode.

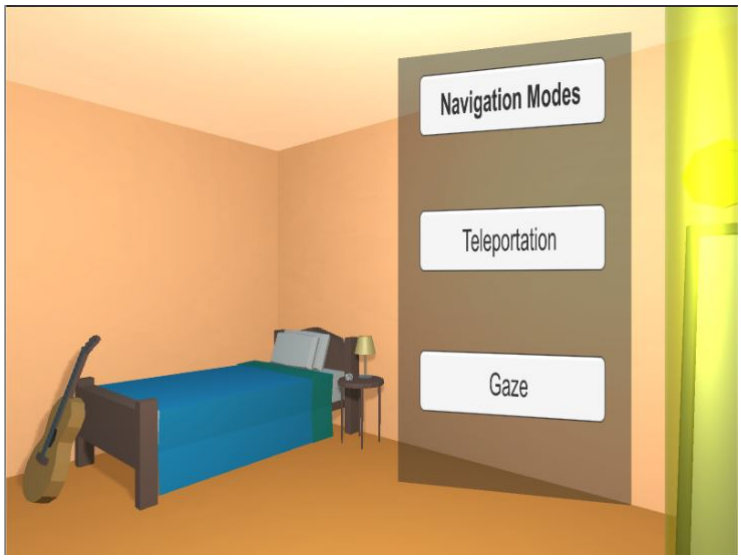

Figure 24: Screenshot of the menu visible in the navigation experiment room.

After a user finishes navigating the room, a measurement indicating the navigation mode and completion time is saved to firebase. The class diagram of the saved data can be seen in figure 25.

Figure 25: Classdiagram for the navigation experiment measurement

After completing the experiment, the user should answer a questionaire (See appendix **??**). The questions of the questionaire and the time measurement of the different navigation modes, should be enough to let us make a decision on which mode to use in the final implementation of the object-manipulation experiment.

7.2 Conducting the Experiment

We gathered people who wanted to participate in the experiment, and split them into two groups; one for each VR platform. Each group had to complete the navigation experiment for each of the two navigation modes. Half of the users on each platforms started with Gaze-based navigation, while the other half started with teleportation, to try and remove any bias stemming from trying a navigation mode first. The split is illustrated in figure 26.

Figure 26: The split of the users trying teleportation and gaze-based walking

Figure 27: Photo of one of the test subjects trying the navigation experiment on the mobile platform.

The users were also told to try and complete the experiment so that we could gather some data regarding the task completion time, but that it was okay to stop the experiment if they were feeling nauseous.

7.3 Results

We ran 20 people through the navigation experiment; 10 for each platform, and even though we know that this amount of people is not enough to be statistically significant, it was what we were able to gather within a reasonable time-frame, and we will use the data with all its imperfection, as it will still be an indication of user preferences for each platform.

The raw data can be found in the appendices **??** and **??**

7.3.1 MVR Navigation

For the navigation experiment for the mobile platform, the users were distributed in the age range from 22 to 42 as seen in figure 28

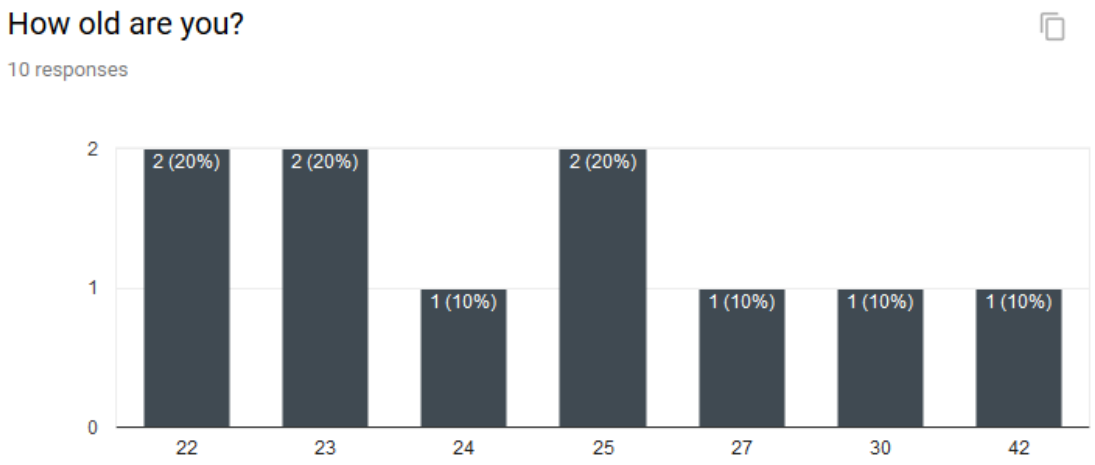

Figure 28: Distribution of age between MVR test subjects.

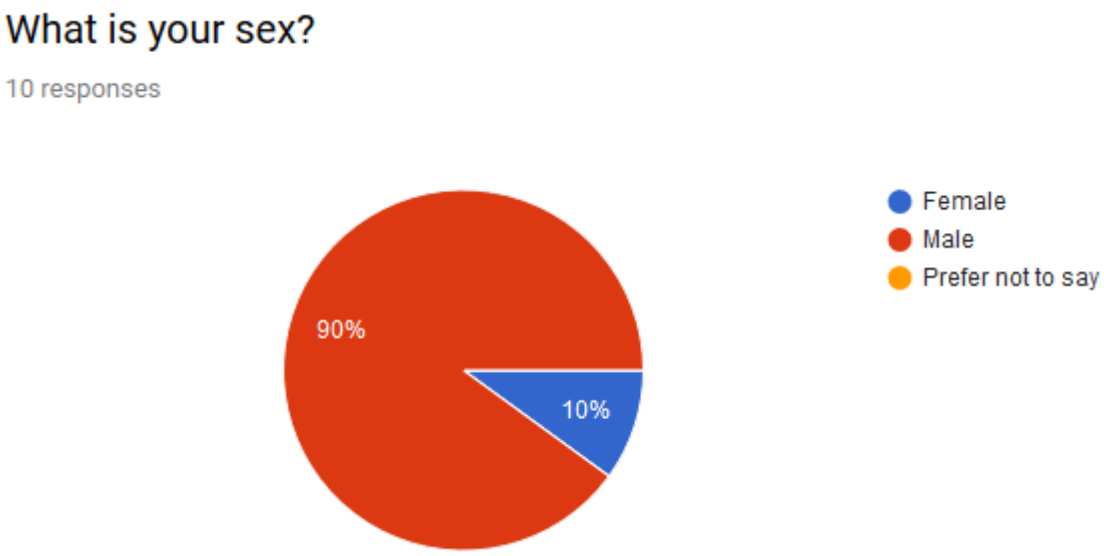

Figure 29: Distribution of gender between MVR test subjects.

Of the test subjects for this experiment, all of them were students, and their educational distribution can be seen on figure 30

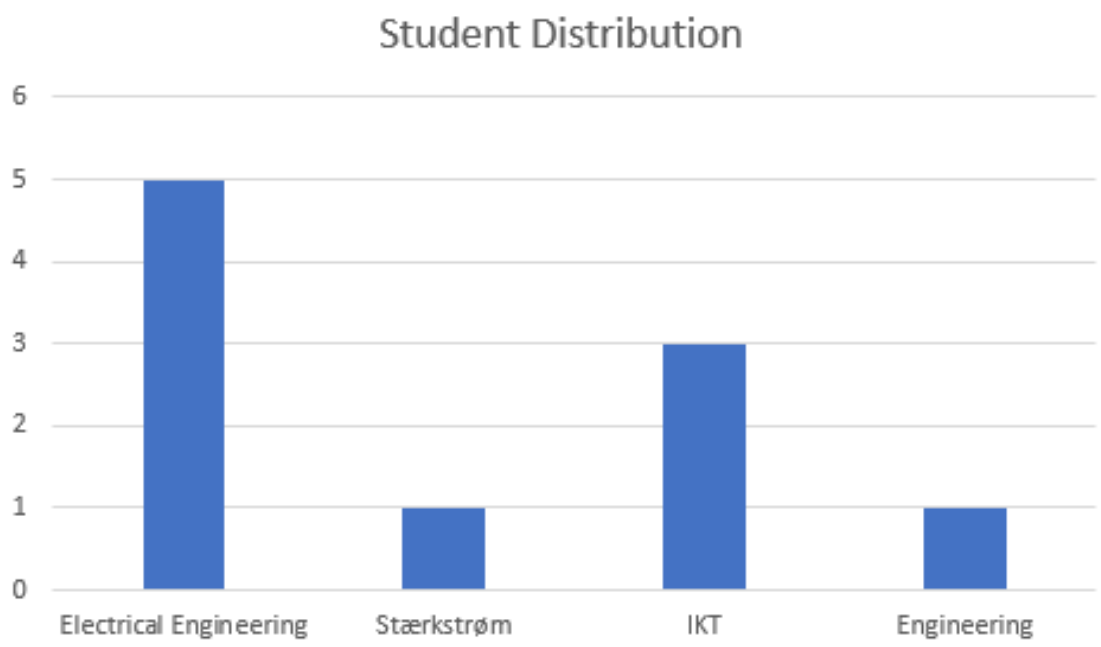

Figure 30: Distribution of education between MVR test subjects.

The test subjects most identified themselves as experienced with interacting with 3D environments, and 80% of the subjects had tried VR previously. Of those 80% most identified as "not that experienced" in VR.

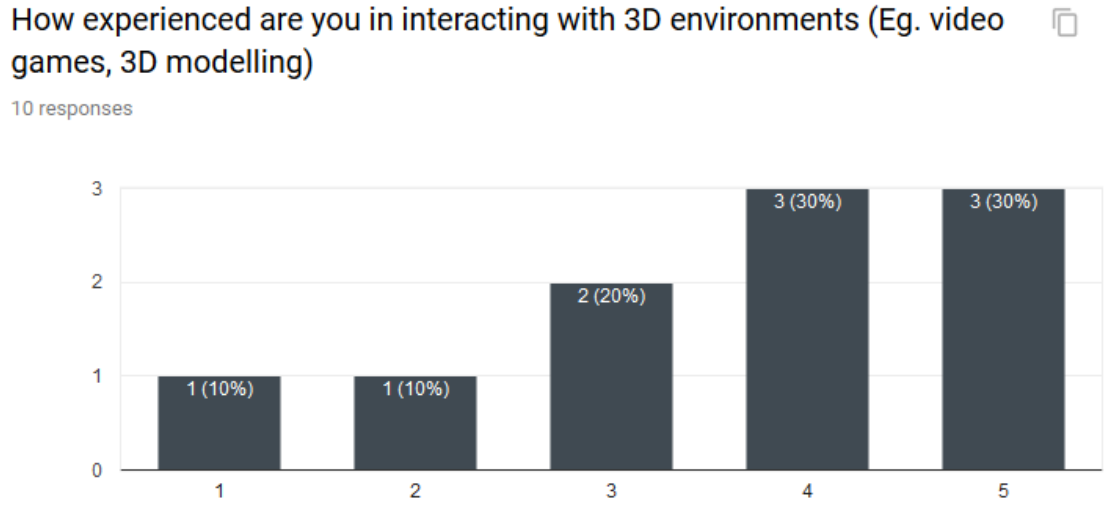

Figure 31: MVR ranking of experience with 3D environments with 1 meaning not that experienced, and 5 meaning very experienced.

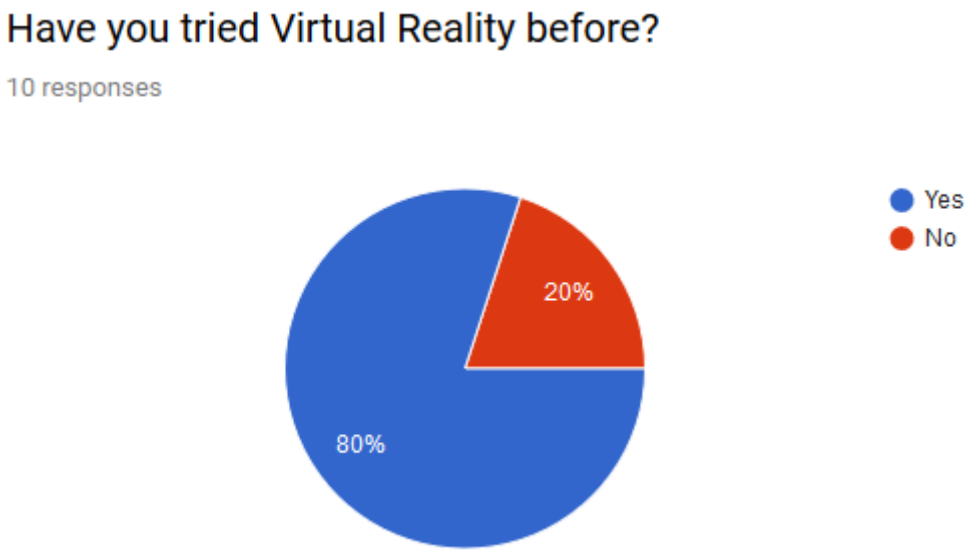

Figure 32: MVR distribution of previous experience with VR.

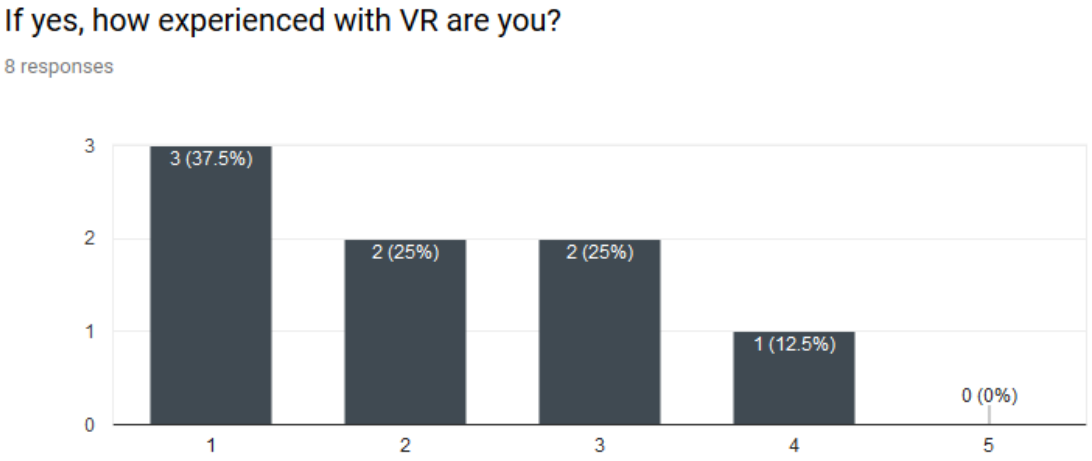

Figure 33: Ranking of experience with Virtual Reality with 1 meaning not that experienced, and 5 meaning very experienced.

When the test subjects tried teleportation, most users strongly agreed with the sentence "It was easy to orient myself and navigate the environment with teleportation" while most of the user disagreed with the statement "Teleporting made me feel nauseous or queasy"

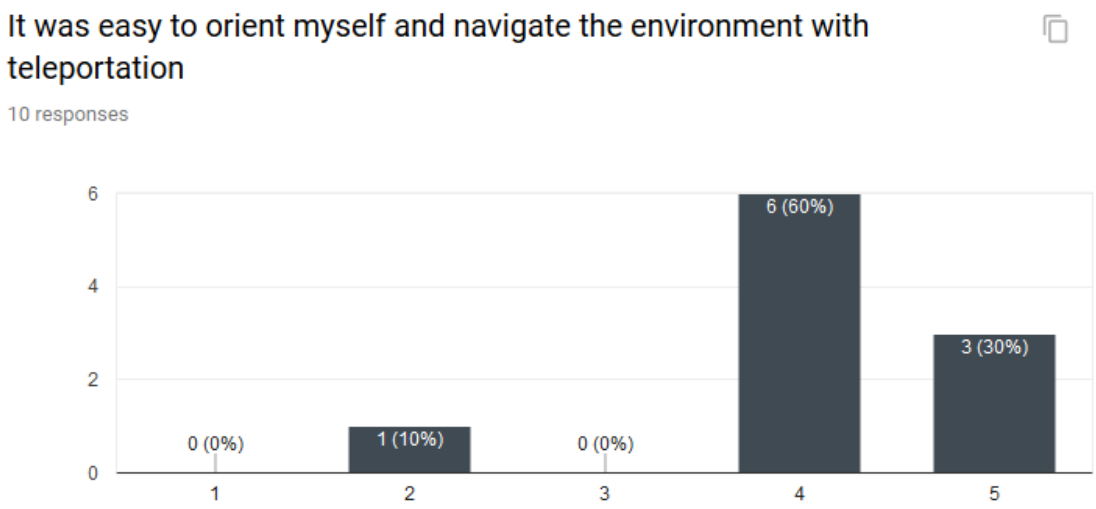

Figure 34: Statistics of agreeing or disagreeing with the statement "It was easy to orient myself and navigate the environment with teleportation" with 1 meaning strongly disagree, and 5 meaning strongly agree.

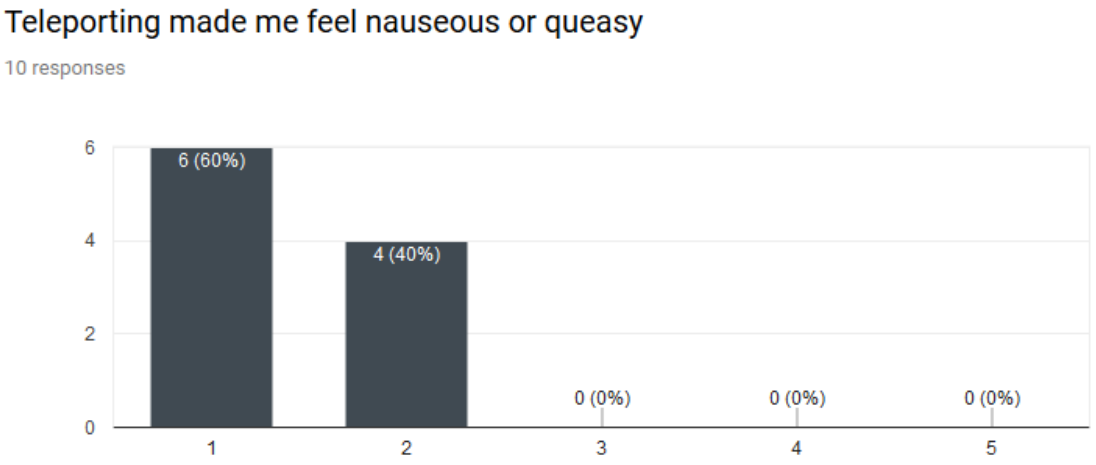

Figure 35: Statistics of agreeing or disagreeing with the statement "Teleporting made me feel nauseous or queasy" with 1 meaning strongly disagree, and 5 meaning strongly agree.

When asked about how they felt about Gaze-based walking, the users agreed with the statement “It was easy to orient myself and navigate the environment with gaze-based walking”, yet compared to teleportation, more people agreed with the statement “Gaze-based walking made me feel nauseous or queasy”.

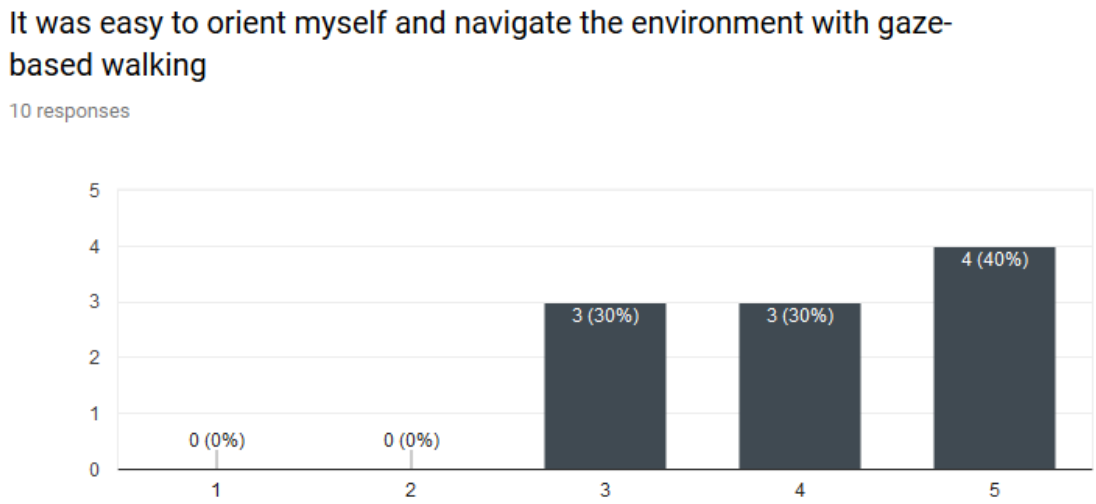

Figure 36: Statistics of agreeing or disagreeing with the statement "It was easy to orient myself and navigate the environment with Gaze-based walking" with 1 meaning strongly disagree, and 5 meaning strongly agree.

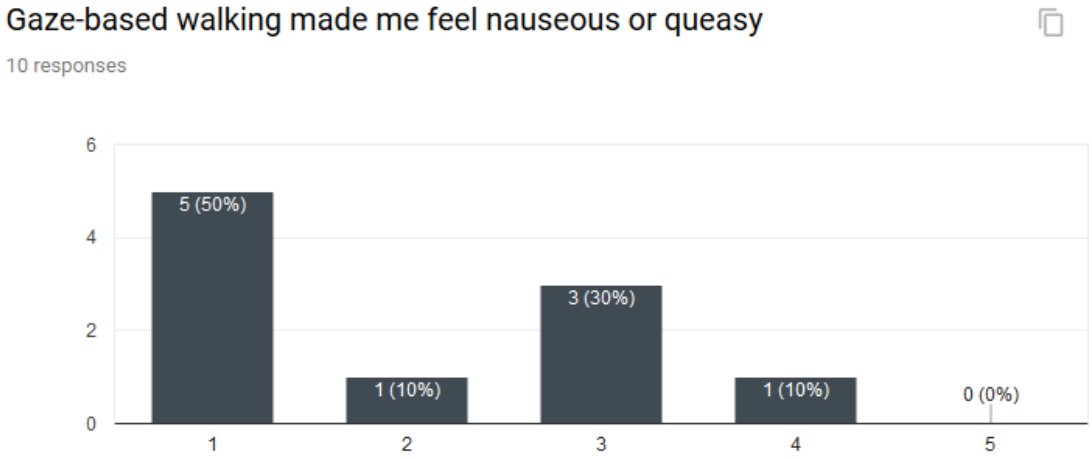

Figure 37: Statistics of agreeing or disagreeing with the statement "Gaze-based walking made me feel nauseous or queasy" with 1 meaning strongly disagree, and 5 meaning strongly agree.

As an objective measurement of the user's performance in the VR environment we have measured their task completion time (the time it took the users to go through the waypoints). The average speed using Teleportation was 81.7 seconds, while the average speed for Gaze-based walking was 110.2 seconds (See appendix **??**).

Finally, when we asked the users of their personal navigation mode preference, 50% of

the users preferred teleportation, with 40% preferring Gaze-based walking, and the last 10% with no preference.

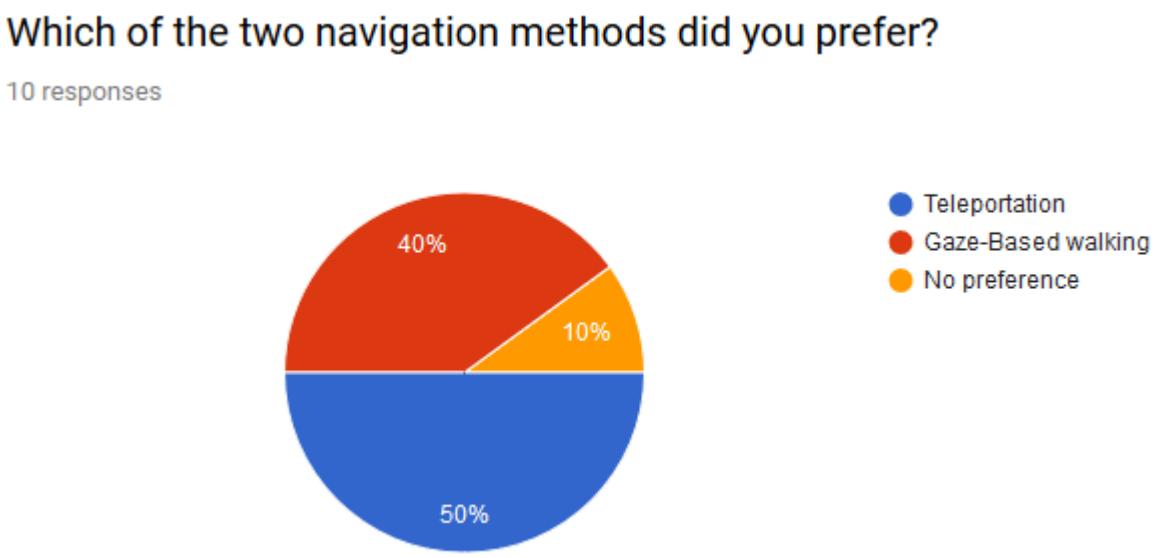

Figure 38: Navigation mode preference by users.

Users that preferred Gaze-based walking commented, that Gaze-based walking was more fluent, and made it easier to get around in the VR space because it felt like actual walking.

Users that preferred teleportation, commented that it was much faster than gaze-based walking, and that they did not like the feeling that they got from gaze-based walking.

7.3.2 IVR Navigation

For the navigation experiment for installed VR, the users were distributed in the age range from 22 to 29 as seen in figure 39

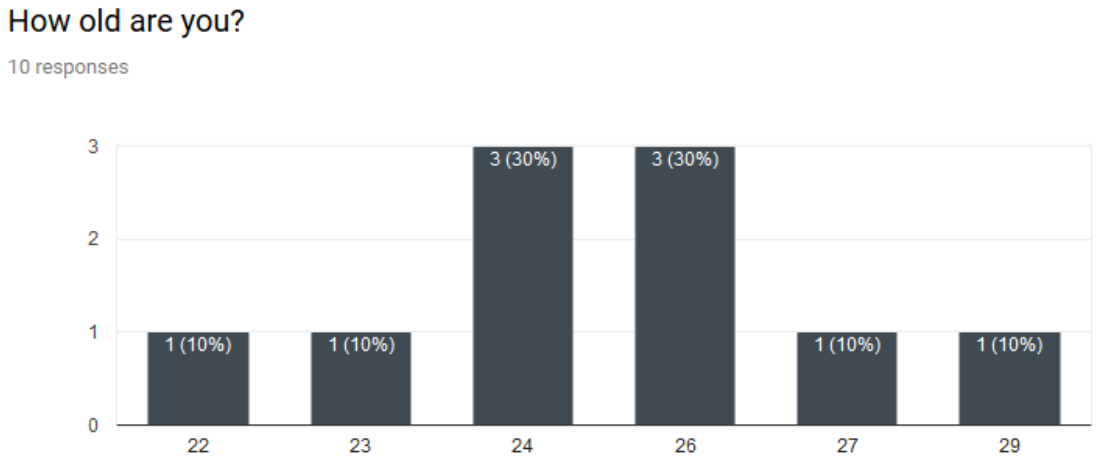

Figure 39: Distribution of age between IVR test subjects.

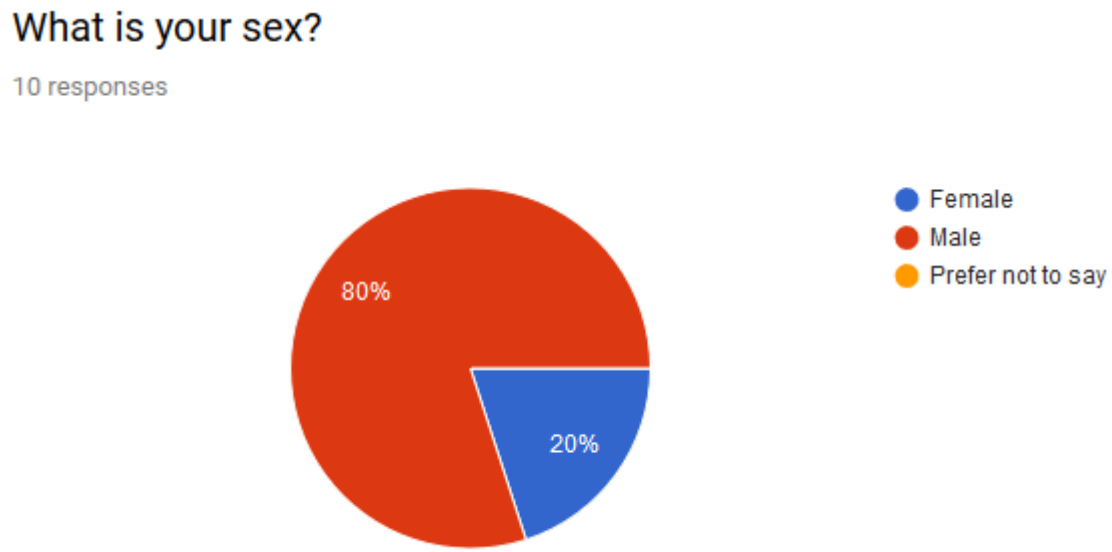

Figure 40: Distribution of gender between IVR test subjects.

Of the test subjects for this experiment, 80% of them were students, and their educational distribution can be seen on figure 41

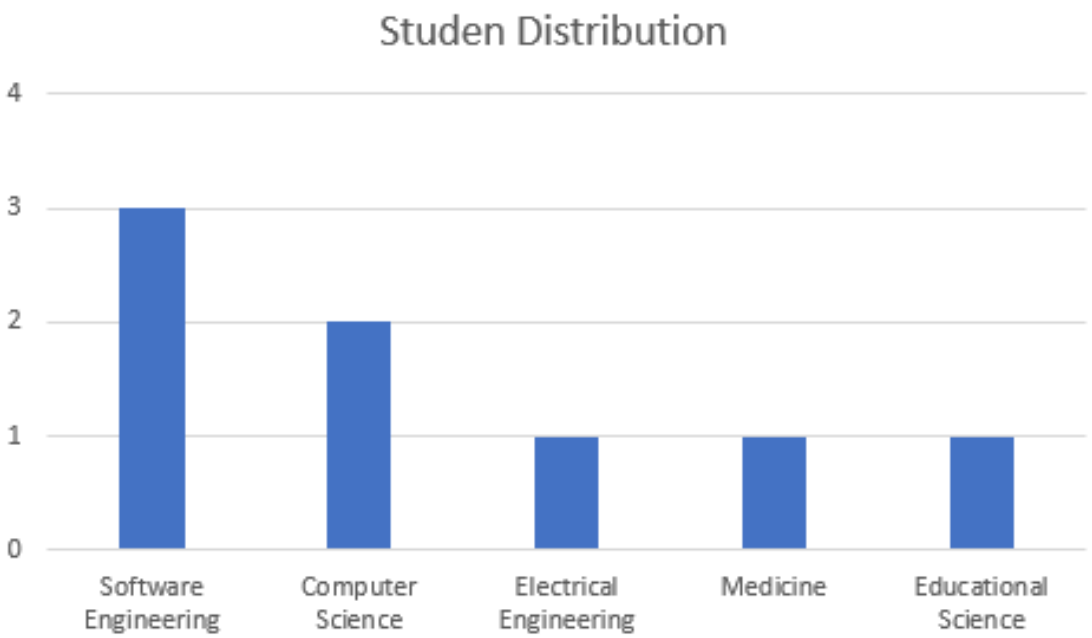

Figure 41: Distribution of education between IVR test subjects.

The test subjects were a mixed group of experience with interacting with 3D virtual environments, and 60% of the subjects had tried VR previously. Of those 60% most identified as "not that experienced" in VR.

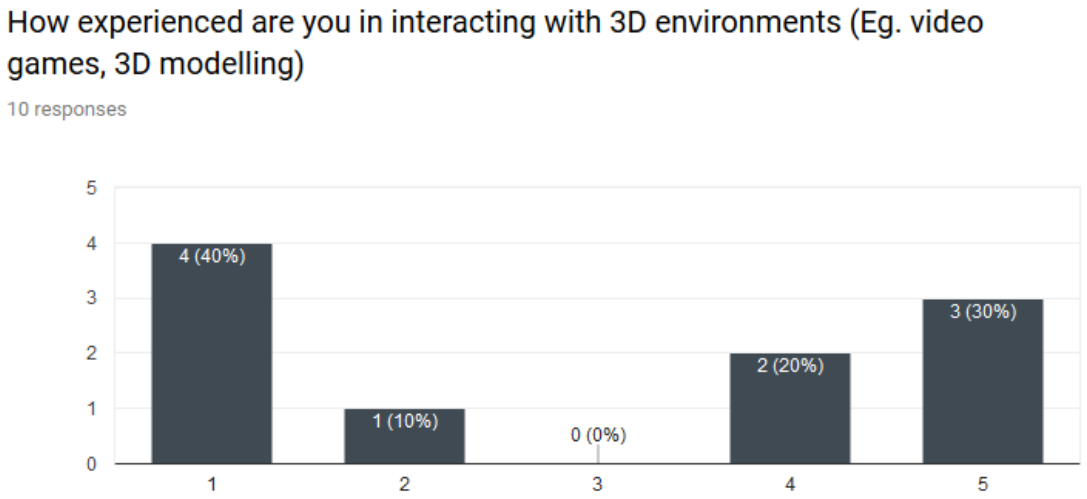

Figure 42: IVR ranking of experience with 3D environments with 1 meaning not that experienced, and 5 meaning very experienced.

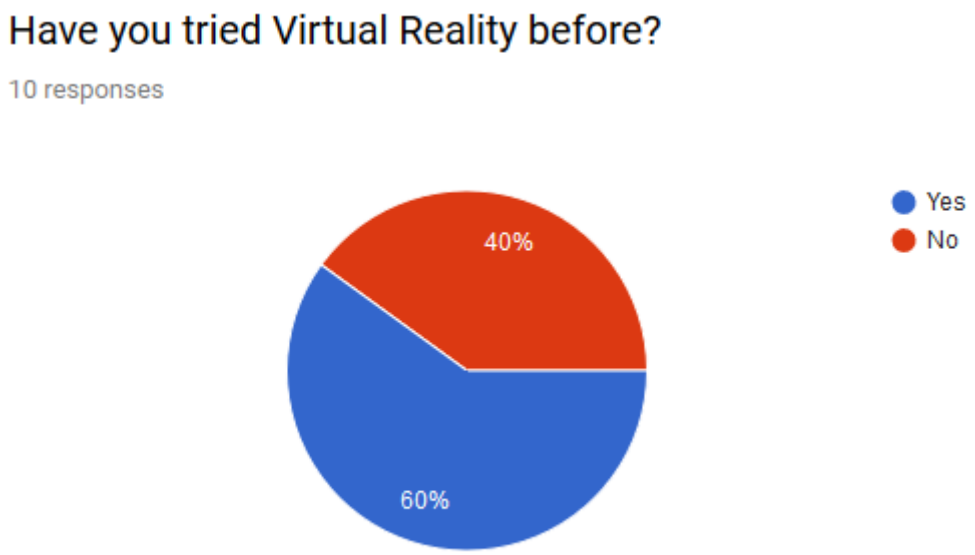

Figure 43: IVR distribution of previous experience with VR.

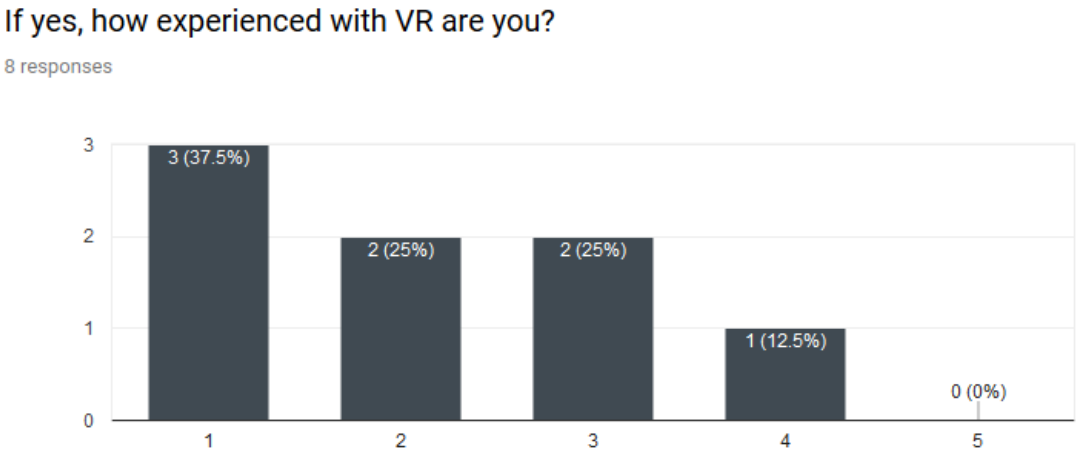

Figure 44: IVR ranking of experience with Virtual Reality with 1 meaning not that experienced, and 5 meaning very experienced.

When the test subjects tried teleportation, most users strongly agreed with the sentence "It was easy to orient myself and navigate the environment with teleportation" while most of the user disagreed with the statement "Teleporting made me feel nauseous or queasy"

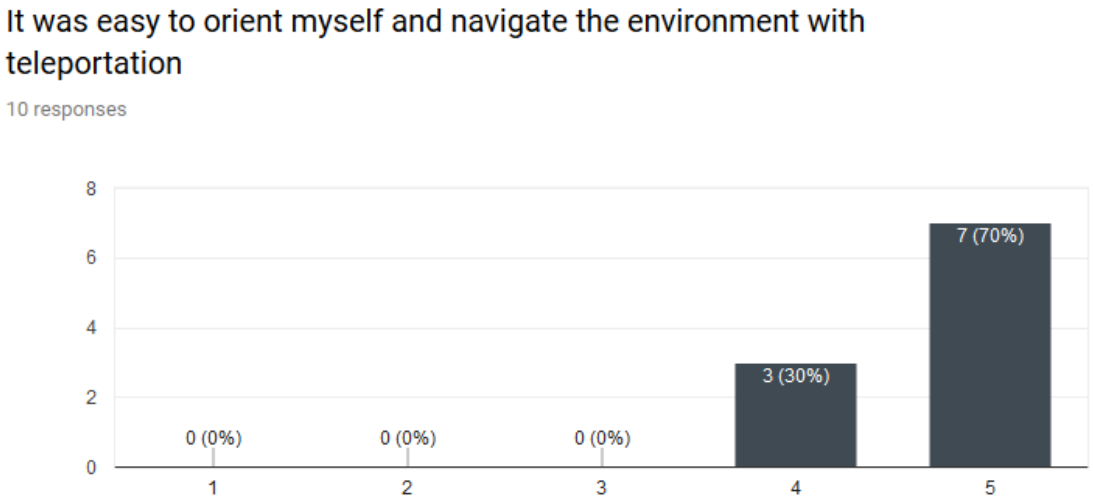

Figure 45: IVR statistics of agreeing or disagreeing with the statement "It was easy to orient myself and navigate the environment with teleportation" with 1 meaning strongly disagree, and 5 meaning strongly agree.

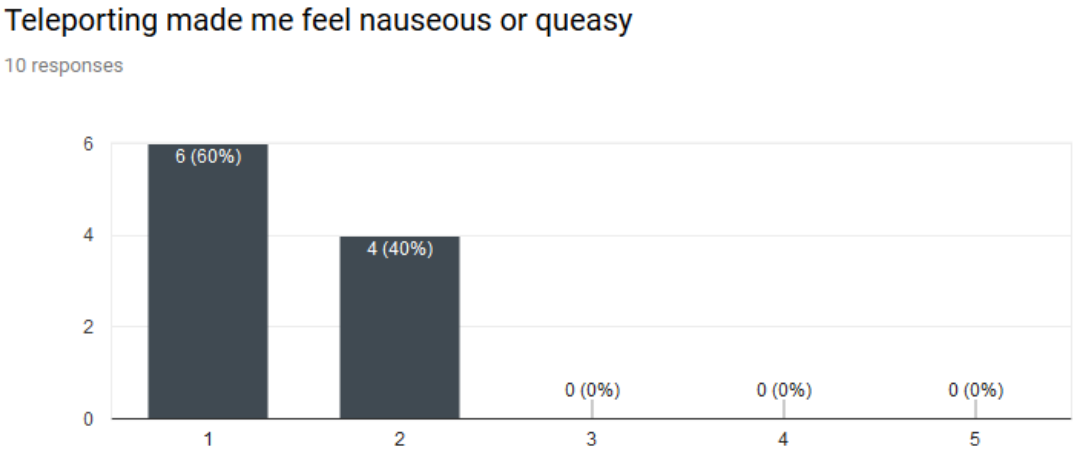

Figure 46: IVR statistics of agreeing or disagreeing with the statement "Teleporting made me feel nauseous or queasy" with 1 meaning strongly disagree, and 5 meaning strongly agree.

When asked about how they felt about Gaze-based walking, we're somewhat more split in agreeing it the statement "It was easy to orient myself and navigate the environment with gaze-based walking", and people strongly agreed with the statement "Gaze-based walking made me feel nauseous or queasy".

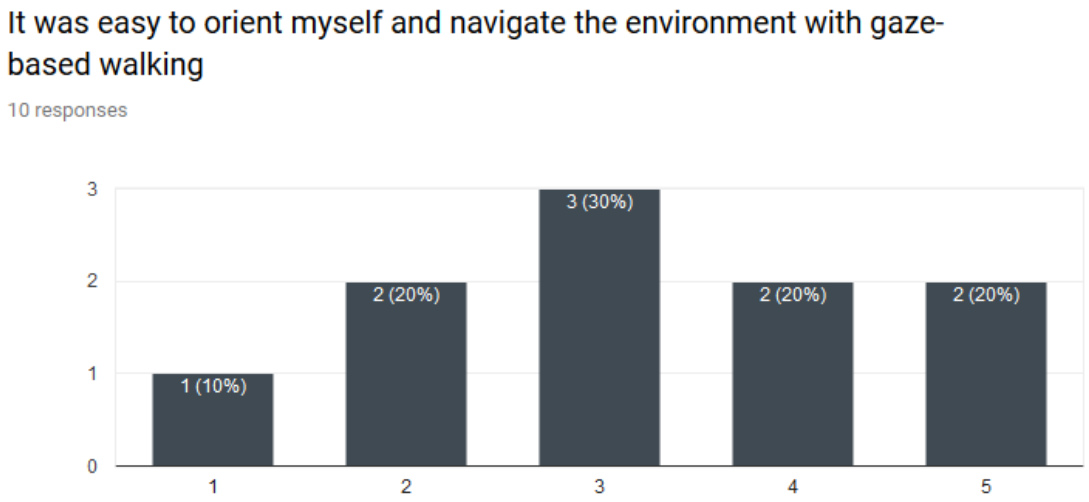

Figure 47: IVR statistics of agreeing or disagreeing with the statement "It was easy to orient myself and navigate the environment with Gaze-based walking" with 1 meaning strongly disagree, and 5 meaning strongly agree.

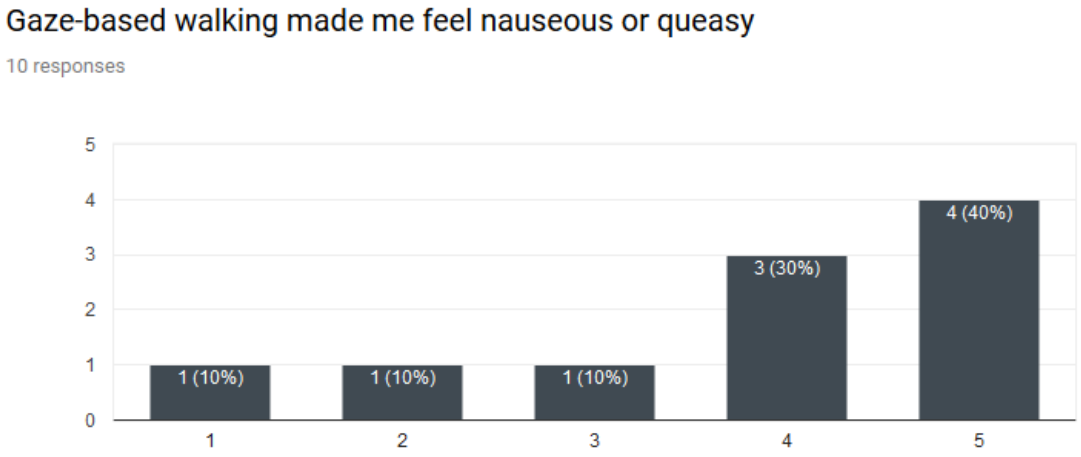

Figure 48: IVR statistics of agreeing or disagreeing with the statement "Gaze-based walking made me feel nauseous or queasy" with 1 meaning strongly disagree, and 5 meaning strongly agree.

As an objective measurement of the user's performance in the VR environment we have measured their task completion time (the time it took the users to go through the waypoints). The average speed using Teleportation was 55.54 seconds, while the average speed for Gaze-based walking was 62.70 seconds (See appendix **??**).

Finally, when we asked the users of their personal navigation mode preference, 80% of the users preferred teleportation, with 10% preferring Gaze-based walking.

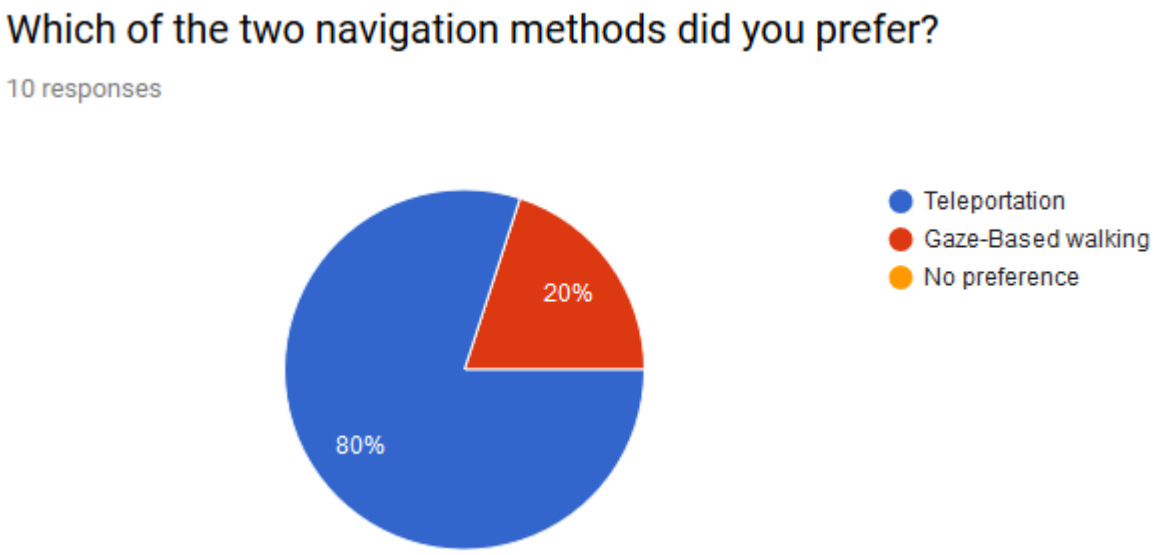

Figure 49: Navigation mode preference by IVR users.

Users that preferred Gaze-based walking commented, that Gaze-based walking felt more real, and authentic, but breaking near furniture did feel nauseating.

Users that preferred teleportation, commented that it the easiest and fastest of the two methods, and that they had more control of where they went. They also commented on, how gaze-based walking made them feel nauseous.

7.4 Waypoint System

In order to facilitate the navigation experiment, we had to implement a waypoint system.

In the navigation experiment, users were required to follow a predefined path through a virtual environment. This predefined path was implemented through the waypoint system.

Figure 50 shows a screenshot of the virtual environment used for the navigation experiment. Notice the yellow, transparent, pillars. These are the waypoints.

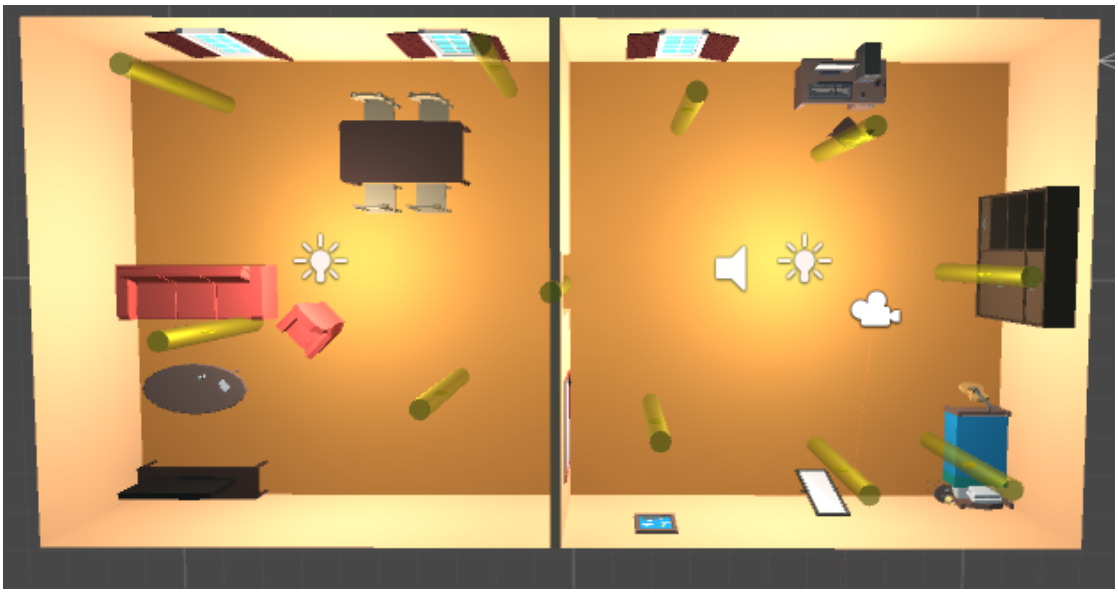

Figure 50: Screenshot of the virtual environment for the navigation experiment. The yellow, transparent, pillars are the waypoints describing a predefined route.

The navigation system consists of three scripts: *Waypoints*, *Waypoint Manager* and *CoinFlipBimbo*. These three scripts will now be described through practical examples.

An experiment consists of a collection of waypoints. All waypoints have their own unique *Order ID*. This Order ID goes from 0 and up. As an example, the navigation experiment presented in figure 50 consists of 8 waypoints. Thus, each waypoint will have assigned a unique Order ID from 0 to 7.

The Order ID is important. This is because only one waypoint will be presented at any one time during the experiment. Thus, at the very beginning of the experiment, the waypoint with an Order ID of 0 will be visible in the virtual environment. The user will then have to navigate into this waypoint. When this happens, the waypoint with an Order ID of 1 will be displayed, and the previous waypoint will be hidden. This process repeats until all waypoints within the experiment have been navigated to. It is the responsibility of the *Waypoint System* to provide this logic. In this way, it is possible to design a predefined route which a user must using a sequence of waypoints.

Figure 51 shows a waypoint up close.

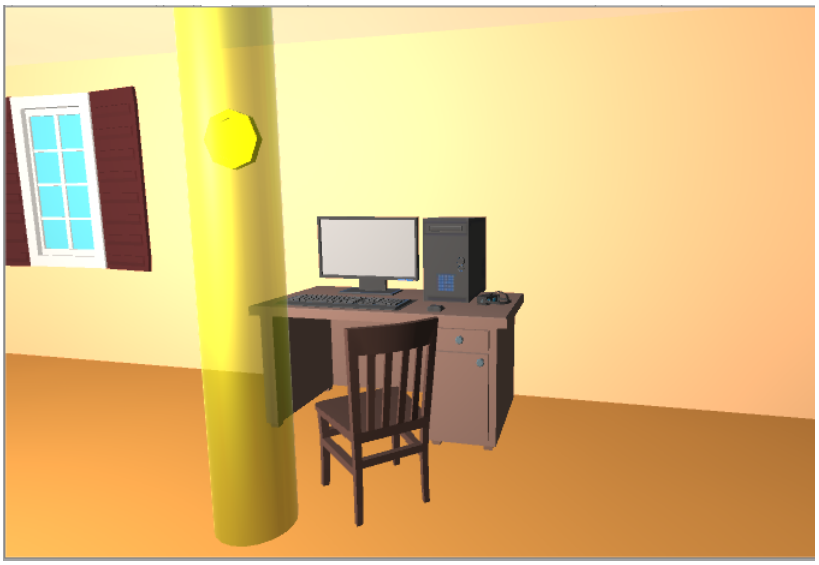

Figure 51: Screenshot of waypoint up close. An animated, spinning coin, is placed within the center of a waypoint.

In figure 51 it is possible to see the coin in the center of the waypoint. This is a spinning coin animated through the *CoinFlipRotater* component.

The three scripts can be seen in figure 52.

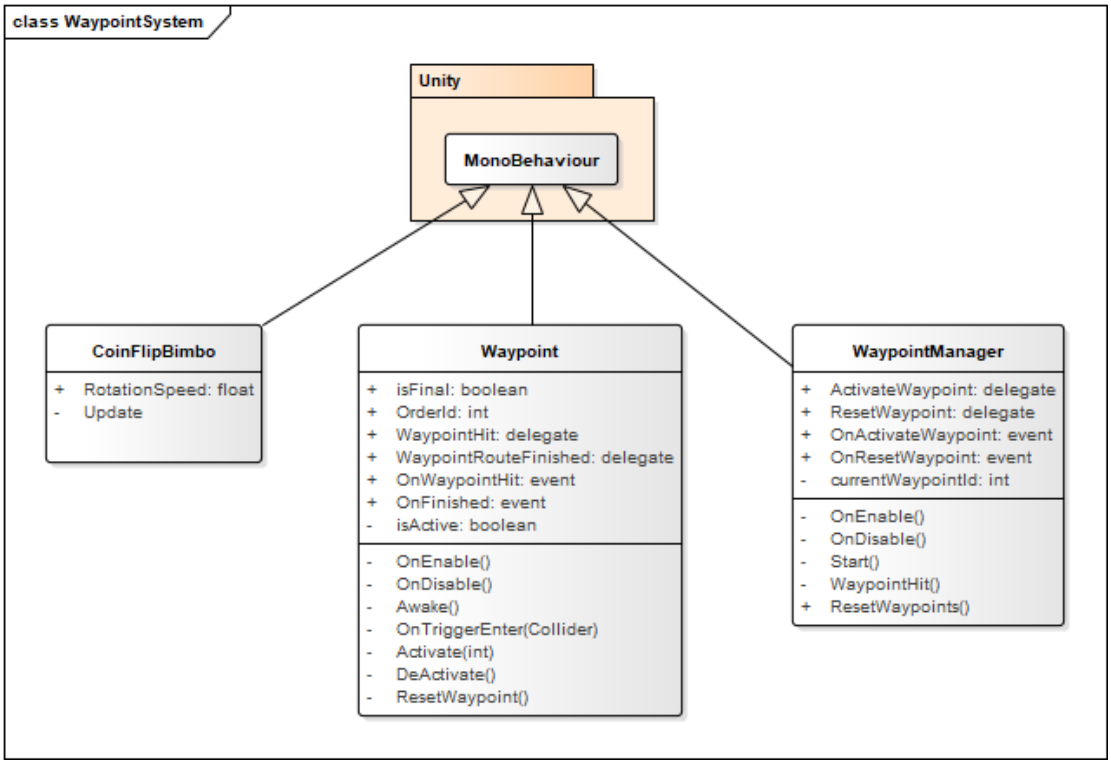

Figure 52: Class diagram of the three scripts composing the waypoint system.

The Waypoint System is based on events, in order for the *Waypoint* and *WaypointManager* to communicate important events. Furthermore, interested *outsiders* are able to listen in on these events as well, if it is important for the task that they have to perform.

These events are described in the following table.

Event	Emitter	Listener	Description
OnActivateWaypoint	WaypointManager	Waypoint	This event is emitted when a new waypoint has to make itself visible.
OnResetWaypoint	WaypointManager	Waypoint	This event is emitted when a client calls ResetWaypoints() on the WaypointManager. It is used by Waypoints in order to hide or show themselves in preparation for a user repeating the same route.
OnWaypointHit	Waypoint	WaypointManager	This event is emitted when a waypoint has been hit by the user. The WaypointManager uses it to determine what waypoint in the route should be activated next.
OnFinished	Waypoint	Interested Third-Parties	This event is emitted when the waypoint representing the final waypoint in a route is hit by a user. Other components within the project listen to this event in order to, for example, save experiment measurements when the route of the experiment is completed.

Table 1: Waypoint System Events

The *WaypointManager* has the responsibility of keeping the general overview of what Order Id is the current waypoint of the route.

Each *Waypoint* simply shows or hides themselves when they become the correct waypoint in the Order Id sequence. If a waypoint is hit and is the final waypoint of the route, it will also emit an *OnFinished* event.

The sequence diagram in figure 53 shows the general flow of the waypoint system, from when a user hits a waypoint.

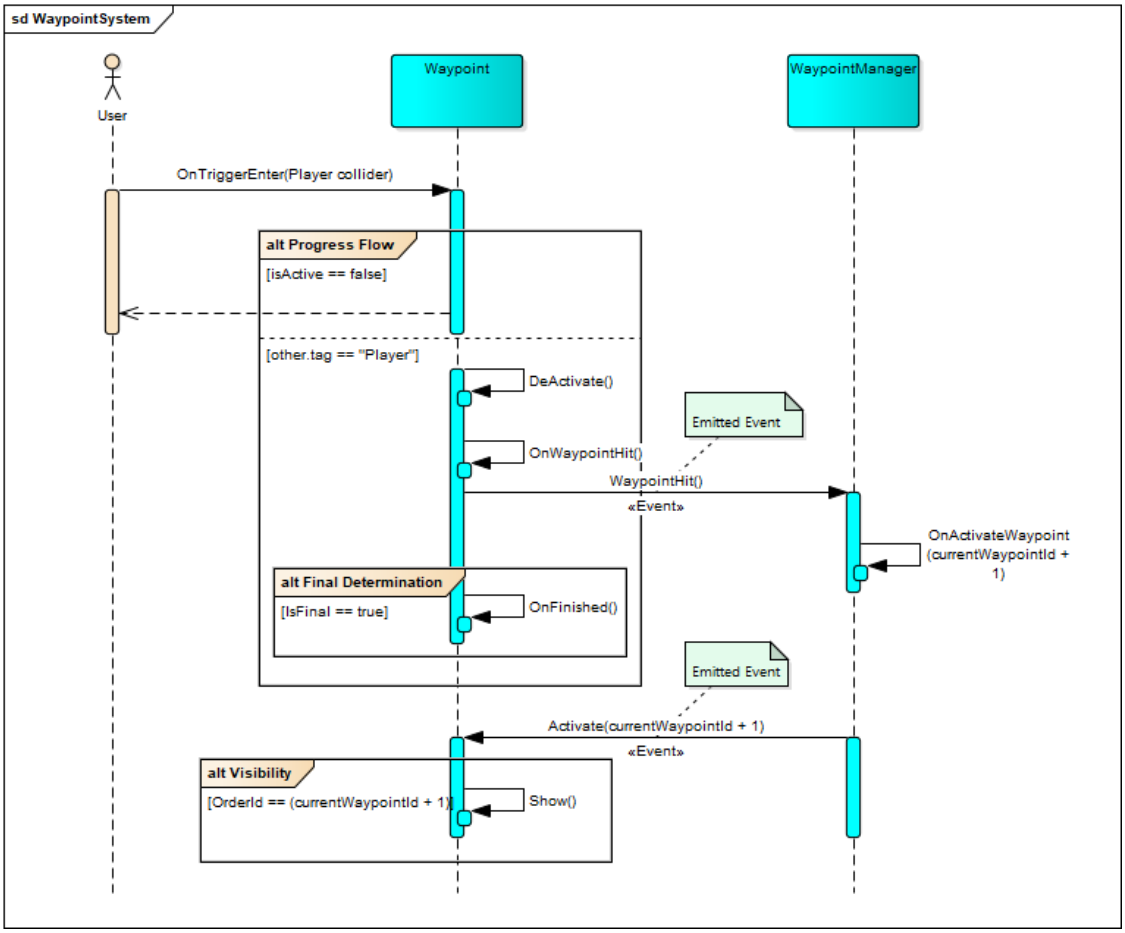

Figure 53: Sequence diagram of the flow of the navigation system, from the point of a user hitting a waypoint.

7.4.1 Waypoints

In order to make the creation of a route as easy as possible through the use of waypoints, the two attributes *IsFinal* and *OrderId* is exposed in the inspector. Figure 54 shows a screenshot of the scene for our navigation experiment, with a waypoint selected. Here, the attributes can be seen as visible and editable in the inspector, right side.

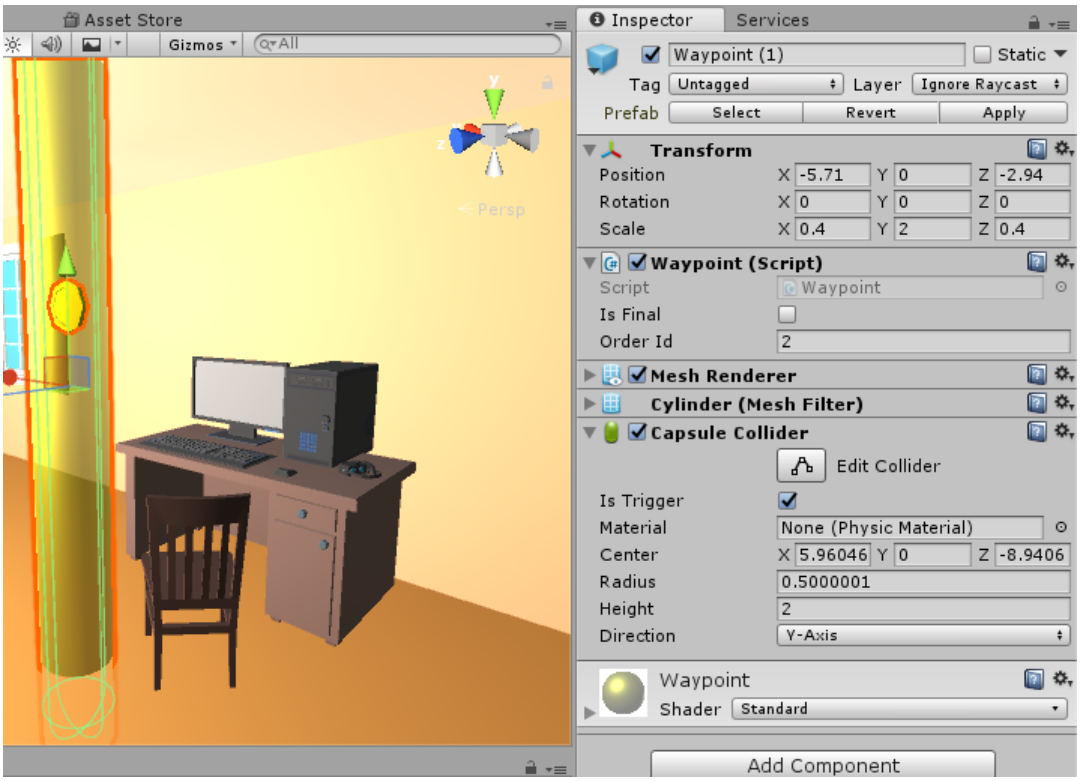

Figure 54: A waypoint selected in the navigation experiment scene. Its attributes *IsFinal* and *OrderId* can be seen in the inspector on the right.

These two attributes have been exposed through the inspector in order to make the waypoint system easily usable to create routes in any scenes. The *IsFinal* attribute can be checked to indicate that the waypoint is the last one in the route. This means that it will emit an *OnFinished* event when triggered (See table 1).

Furthermore, we made waypoints a prefab within the project. This means that it is an easily re-useable component that can be dragged into any scene without requiring manual setup.

7.4.2 CoinFlipRotator

This is a simple script which animates the coin at the center of the waypoint, by continuously rotating it.

The script exposes the *RotationSpeed* attribute to the inspector. This can be used to easily adjust the rotational speed of the animation. A screenshot of the CoinFlipRotator script from the inspector can be seen in figure 55.

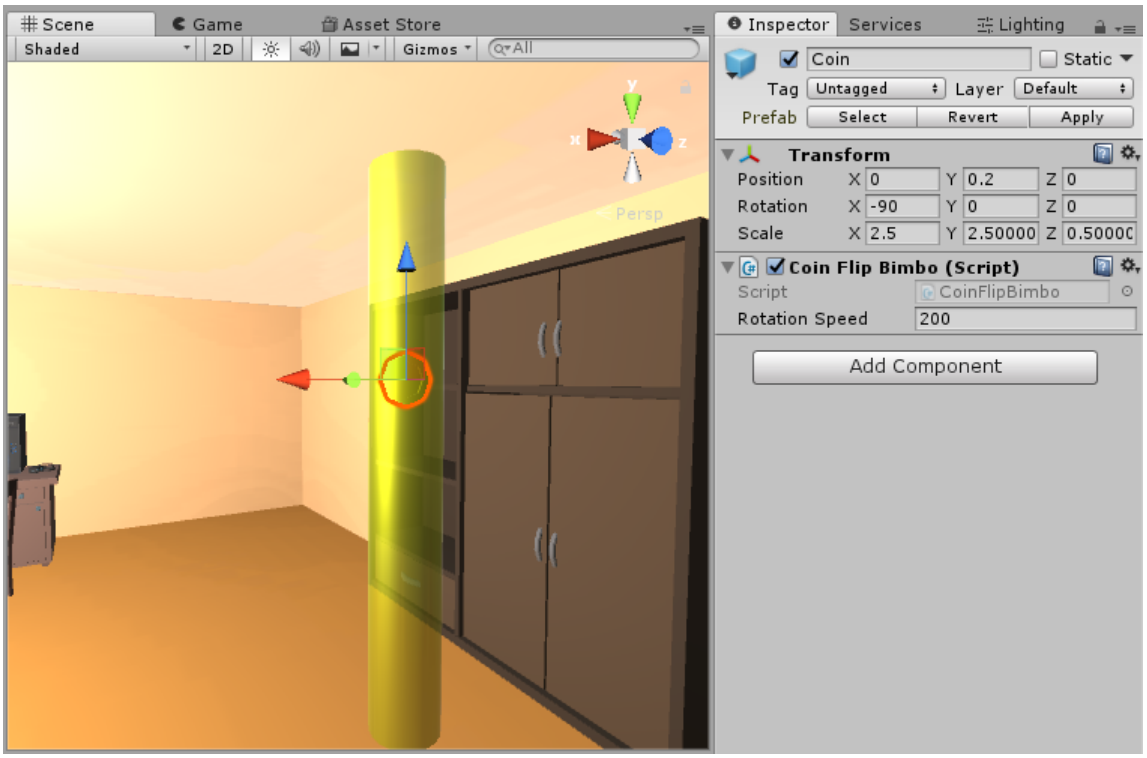

Figure 55: The CoinFlipRotator script as seen from the inspector. The *RotationSpeed* attribute is easily adjustable from here.

7.5 Navigation

Navigation is implemented differently depending on whether the platform is IVR and MVR. This is because navigation is heavily dependent on gathering input from the controllers, and these change depending on what platform you develop for.

The following two subsections will describe how we have implemented navigation on IVR and MVR respectively. During the project we implemented two modes of navigation: Gaze-based locomotion and Teleportation. These are the implemented modes that will be described here.

7.5.1 Installed Virtual Reality

In order to implement navigation for IVR, we made use of Unity's built-in input handler, as well as SteamVR SDK [**?**] for Unity.

Gaze-based locomotion

Gaze-based locomotion for IVR is handled through the script *GazeNavigationInstalled*. According to the SteamVR Unity documentation, the HTC VIVE controllers touch input

from the trackpad is mapped to the Unity joystick ID's 2 and 4 [**?**]. Through the Unity Input Manager we have mapped these ID's to the custom-defined axes *HtcViveLeftHand* and *HtcViveRightHand*. Figure 56 shows both of our custom-defined axes, with a detailed view of the *HtcViveRightHand* axis. Both axes are configured the same way.

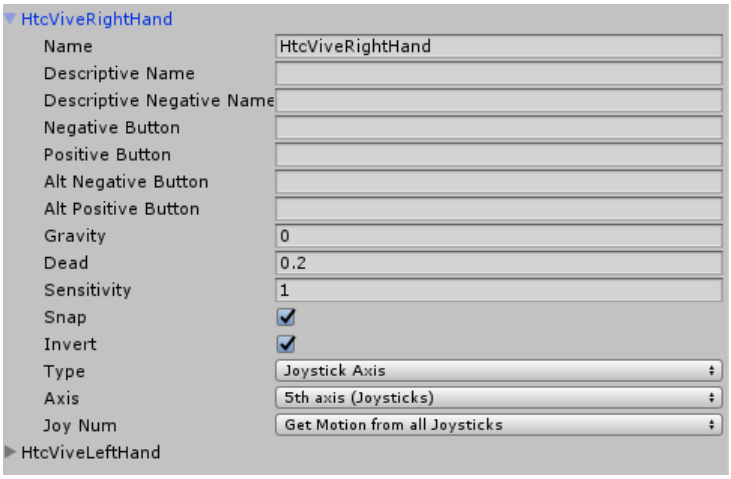

Figure 56: Screenshot from the Unity input manager

The script of *GazeNavigationInstalled* reads from the joystick input axes that we defined in the input manager, and applies that value to a forward movement vector. Since the script is applied to both controllers (referred to as *hands* in the code), it is important for the script to detect its relevant controller. If a script instance is applied to the left controller, it should only read the touchpad values from that controller. This is shown in the codesnippet of figure 57.

```
public void Update () {
    string axisName = hand.GuessCurrentHandType() == Hand.HandType.Right
        ? "HtcViveRightHand"
        : "HtcViveLeftHand";

    float axisValue = Input.GetAxis(axisName);

    Vector3 camForward = _mainCamera.transform.forward;
    camForward.y = 0;

    Vector3 forwardMove = camForward * axisValue * MovementSpeed * Time.deltaTime;

    _player.transform.Translate(forwardMove);
}
```

Figure 57: Codesnippet from the update method of the GazeNavigationInstalled script

Teleportation

Teleportation is implemented entirely using SteamVR's official prefabs from the Unity SteamVR SDK.

We felt that this was the most optimal way of doing it, as it provides all the features we need for teleportation and roomscale.

Figure 58 is a screenshot of one of our early prototypes using SteamVR's prefabs.

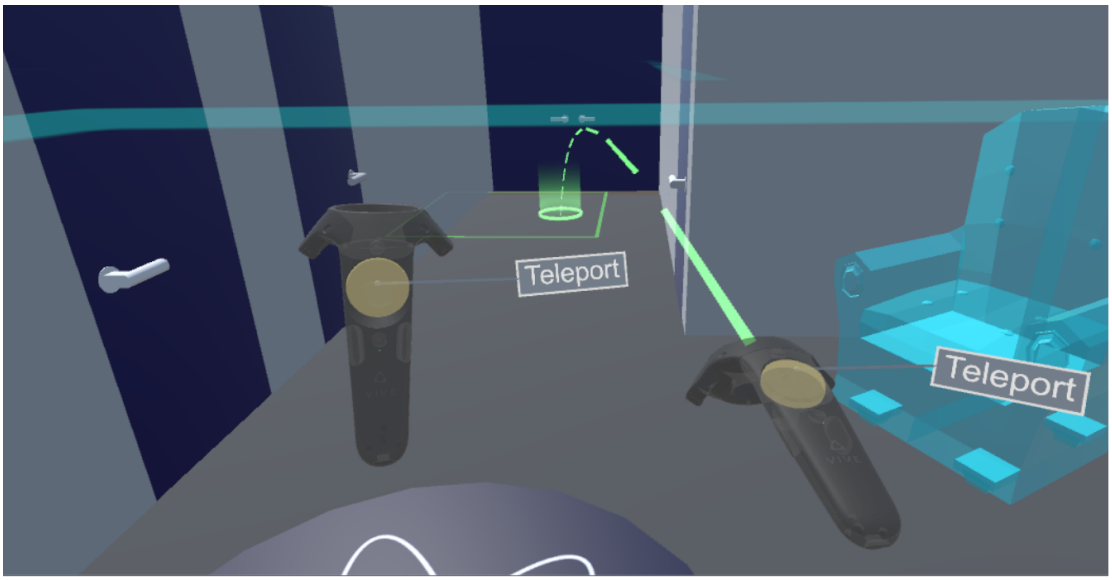

Figure 58: Screenshot demonstrating the SteamVR teleportation system in one of our prototypes.

In order to make teleportation work, you have to make use of two prefabs provided by SteamVR: *Teleporting* and *TeleportArea*. These two prefabs will now be described in detail.

Teleporting is a prefab which controls the general configuration for teleportation in your scene. Figure 59 demonstrates some of the possible settings.

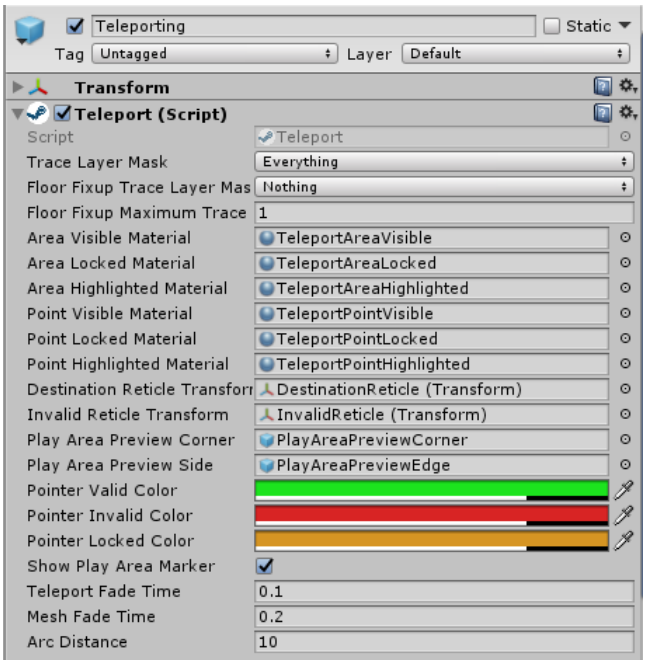

Figure 59: Settings from the Teleporting prefab within a scene of our project.

As can be seen, it is possible to configure things such as the materials used for different states of teleport areas within your scene. It is also possible to configure things such as the raycast arc, as seen on figure 58.

7.5.2 Mobile Virtual Reality

In MVR, gaze-based locomotion and teleportation is implemented using the *GVRControllerInput* class from the Google Daydream Unity SDK. [**?**]

Gaze-based locomotion
Gaze-based locomotion for mVR is implemented in the script *Gaze Navigation*. The script checks the *GVRControllerInput* class to see if the user is interacting with any controller input. In this script we specifically look for the touch-input on the trackpad. From the documentation [**?**] we see that, the controller trackpad is divided into a 1-by-1 coordinate system with (0,0) in the top left corner of the trackpad. Using this knowledge we can detect where the user is touching the trackpad, and use this to decide whether or not to move the user. This can be seen in the code snippet in figure 60.

```
public void Update ()
{
    if (!GvrControllerInput.IsTouching) return;

    var forwardMovement = _mainCamera.transform.TransformDirection(Vector3.forward);
    forwardMovement.y = 0;

    var touchPosition = GvrControllerInput.TouchPos;
    if (touchPosition.y < 0.25f)
        _player.transform.Translate(forwardMovement * Time.deltaTime, Space.World);
    else if (touchPosition.y > 0.75f)
        _player.transform.Translate(-forwardMovement * Time.deltaTime, Space.World);
}
```

Figure 60: Code snippet from the update method of the *Gaze Navigation* script.

We decided to create a "deadzone" in the middle of the trackpad where the user can rest his or her finger without moving. If the user touches the top of the trackpad, the position of the player will be moved forward in the direction the player is looking, and if the user touches the bottom of the trackpad, the player will be moved backwards relative to where the player is looking.

Teleportation

Teleportation for MVR is implemented in the script *Teleportation*. The script is placed on the "laser" prefab which is a part of the controller representation. We did this to be able to easily use the controllers transform as a starting point for the physics raycast (See figure 61)

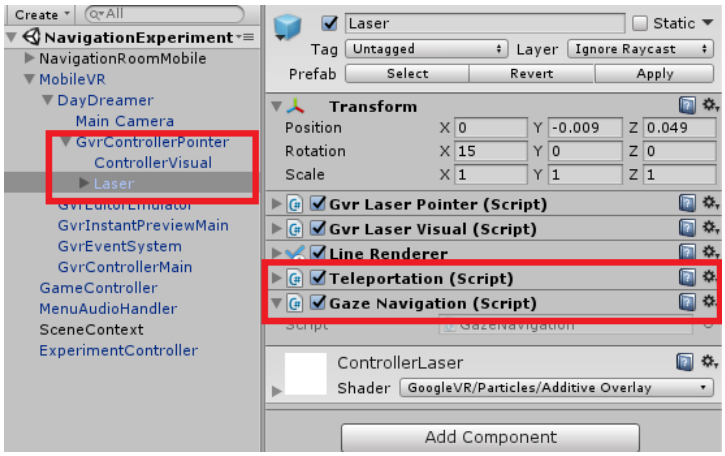

Figure 61: Placement of the navigation scripts in the controller prefab.

From the update method of the Teleportation script, we raycast 20 units in the forward direction of the transform. Using the RaycastHit class, we can see the gameobjects hit

by this raycast and get the hit coordinates in world space, which we then can use to move the player when pressing the teleport button. The button press is detected by using the *GVRControllerInput* class. Since we, for the final experiment, is going to have furniture objects in the scene, we want to make sure that the player cannot teleport on top of furniture objects. To achieve this, we simply check the tag of the RaycastHit object, and disallow teleportation if the hit object is teleportation. A code snippet of the the teleportation logic can be seen on figure 62.

```
public void Update () {
    var forwardDirection = transform.TransformDirection(Vector3.forward);

    RaycastHit raycastHit;
    if (!Physics.Raycast(transform.position, forwardDirection, out raycastHit, 20)) return;

    _hittingFurniture = raycastHit.transform.CompareTag("Furniture");
    if (GvrControllerInput.ClickButtonDown)
    {
        if (!_hittingFurniture)
        {
            Debug.Log("Hit : " + raycastHit.collider.name);
            if (raycastHit.collider.CompareTag("Floor"))
            {
                _player.transform.position = new Vector3(
                    raycastHit.point.x,
                    _player.transform.position.y,
                    raycastHit.point.z);
            }
        }
    }
}
```

Figure 62: Code snippet of the update method in the *Teleportation* script.

7.6 Measurement Repositories

The measurement repositories of the navigation experiments are used to save the navigation experiment measurements to Firebase.

Figure 63 shows a class diagram of the classes comprising the measurement repositories.

Figure 63: The classes and interfaces comrpising the navigation measurement repositories.

The interface *IMeasurementRepository* represents the possible operations of any concrete implementation of a navigation measurement repository. It contains the method *SaveNavigationExperimentTimeStamp*, which takes an instance of the model class *NavigationExperimentMeasurement*.

Relevant data for the navigation experiment was how long a user took to complete the navigation experiment, and which mode of navigation was used. Thus, the *Navigation-*

ExperimentMeasurement model class contains the properties *TimeTaken* and *Mode* exactly for this purpose.

The properties *GazeMode* and *TeleportationMode* of the *NavigationExperimentMeasurement* are static constant strings which can be used by clients when setting the *Mode* property. It is simply pre-defined strings which can be used in order to avoid inconsistent spelling of the navigation modes used.

The classes *AndroidMeasureToolRepository* and *HttpMeasureToolRepository* are concrete implementations of the *IMeasurementRepository* interface. These are platform-specific implementations which can be used on Android or PC respectively.

See section 7.7 for technical details of how these repositories were used practically in the navigation experiment.

Also notice that during the final object manipulation experiment, a generalized repository system for saving measurements was conceived and implemented. This is a separate system, the Firebase Measurement Repository System, documented in section **??**.

7.7 NavigationExperimentMeasurer

The *NavigationExperimentMeasurer* is a script in our system which has the responsibility of saving time measurements from the navigation experiment onto the database once the pre-defined route in the experiment is completed by a user.

Figure 64 presents the script in a class diagram.

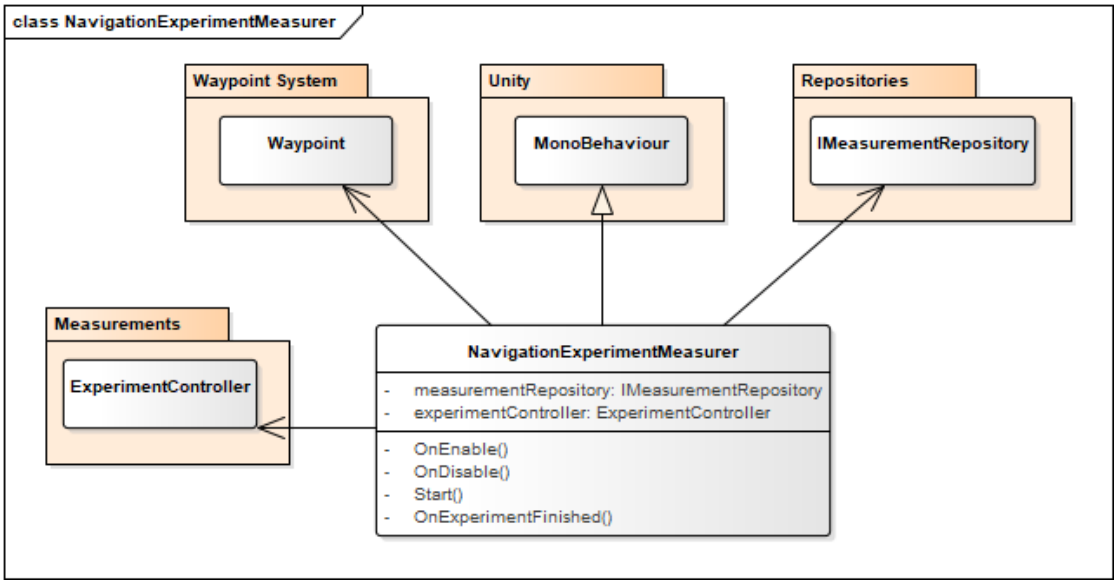

Figure 64: Class diagram of the NavigationExperimentMeasurer script.

The context of the methods of *NavigationExperimentMeasuer* is best understood through the flow of its lifetime. Thus, figure 65 presents the primary sequence for the lifetime of this script. It utilizes the event system of the Waypoint System described in section 7.4 in order to listen in on the *OnFinished* - described in table 1 of section 7.4 - event that is triggered once a user completes the pre-defined route. When this event is triggered, the time in seconds which it took to complete the route is fetched from the *Experiment-Controller* of the scene, and afterwards saved to the database through the repositories presented in section 7.6.

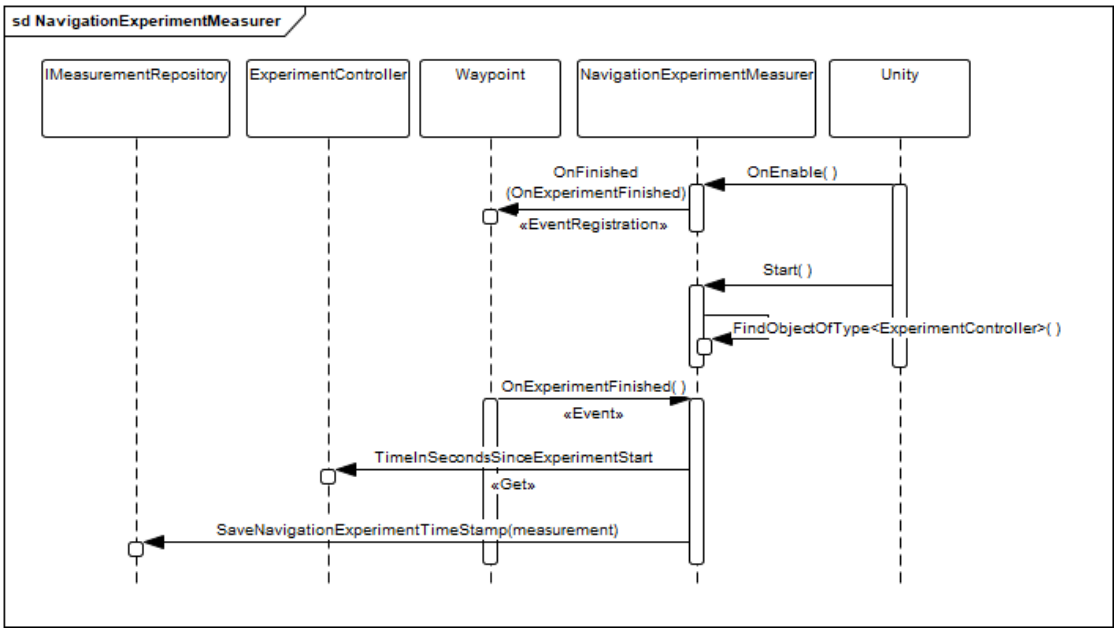

Figure 65: Class diagram of the NavigationExperimentMeasurer script.

7.7.1 Cross-Platform Considerations

NavigationExperimentMeasurer works cross-platform through the use of Zenject.

The attribute *measurementRepository*, seen in the class diagram of figure 64 is determined and injected by Zenject at runtime depending on the platform which the application is running on. In this way, *NavigationExperimentMeasurer* is easily reuseable, and can easily be used to extend to new platforms.

7.8 NavigationPlayerController

The *NavigationPlayerController* is a script in our system which has the responsibility of keeping a shared state of the current navigational mode of the player during the navigation experiment. Additionally, it also configures anything that has to do with the navigation.

Figure 66 shows a class diagram of the script.

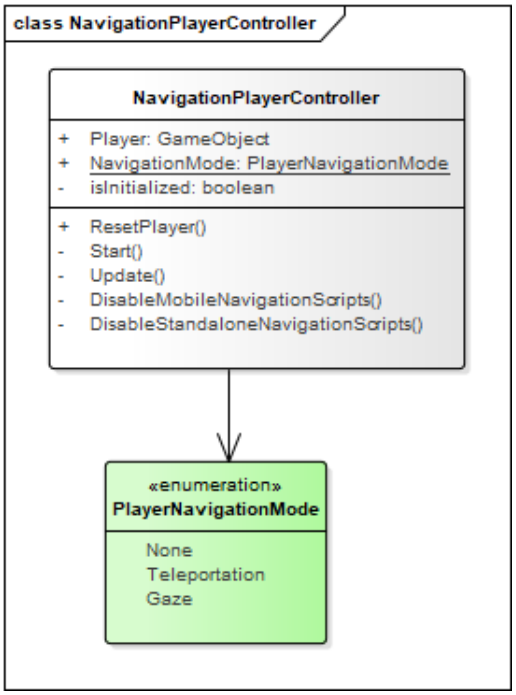

Figure 66: Class diagram of the NavigationPlayerController script.

The script has two primary purposes.

It disables all navigation during the first launch of the navigation experiment. This is to ensure that users are forced to choose a navigation mode themselves.

It is also used by other scripts of the navigation experiment to reset the player to a starting position. This is done through the publically exposed method *ResetPlayer*.

The initialization flow of the script can be seen in figure 67.

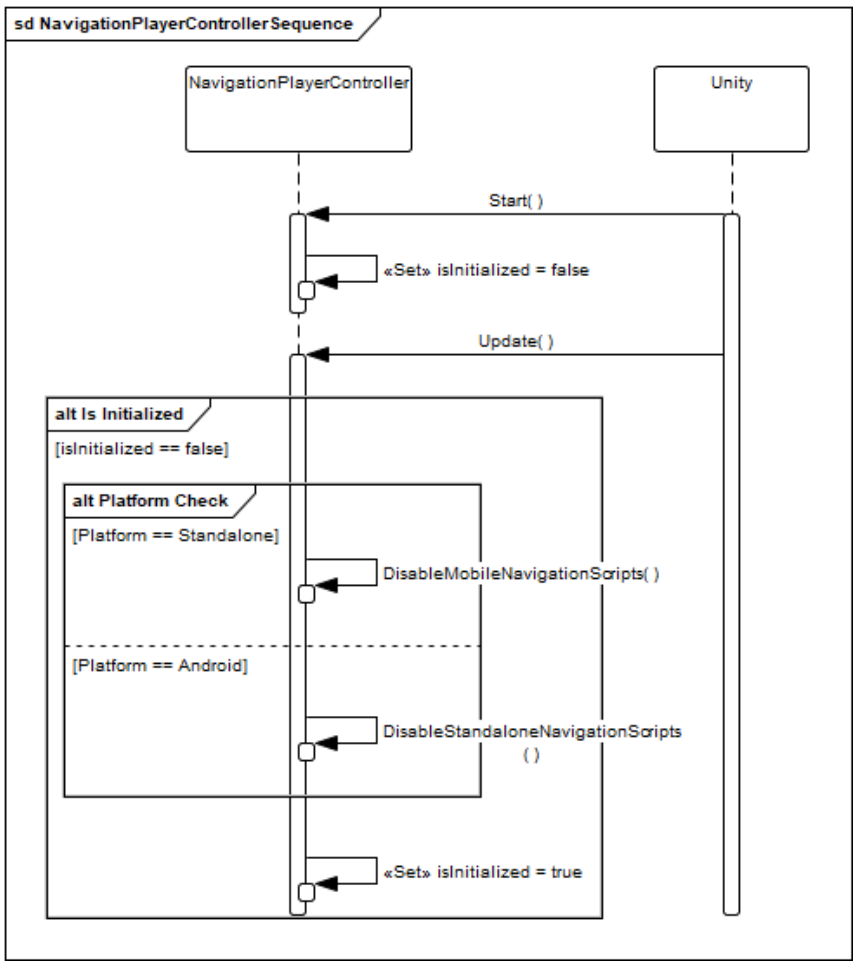

Figure 67: Sequence diagram of the NavigationPlayerController script.

7.9 Navigation Experiment UI

As the navigation experiment consisted of users having to try out two different modes of navigation, it was important that we had an easy way to switch between the two different modes during runtime of the navigation experiment on both platforms.

In order to achieve this, we made an interactable 3D menu that can be brought up at any point during runtime. The final result can be seen in figure 68.

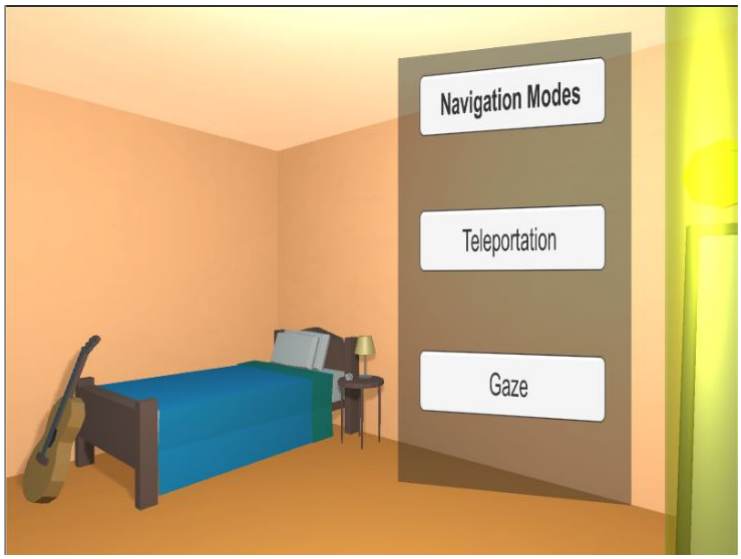

Figure 68: Screenshot of the menu visible in the navigation experiment room.

As can be seen in figure 68, the menu is a 3D panel that is placed within the virtual environment. By interacting with the buttons, the navigation mode can easily be changed.

The menu is also animated, in the sense that it will gradually rotate towards the player, so as to always be facing in the correct direction. This means that if you make the menu visible within the world and start to move around, the menu will continually look towards you.

The 3D menu is the same on both IVR and MVR platforms. Therefore, we made the 3D menu as a prefab which could be shared amongst the two platform scenes. However, as we learned in the UI prototypes for both the installed and mobile platform - see section 9.3 - each platform required different platform-specific components on the UI GameObjects comprising the menu. As such, both platforms use a shared menu prefab as a starting point, and then each platform scene will add the necessary platform-specific components.

7.9.1 Scripts

The menu system for the navigation experiment is comprised of multiple scripts. These scripts can be seen in the class diagram of figure 69.

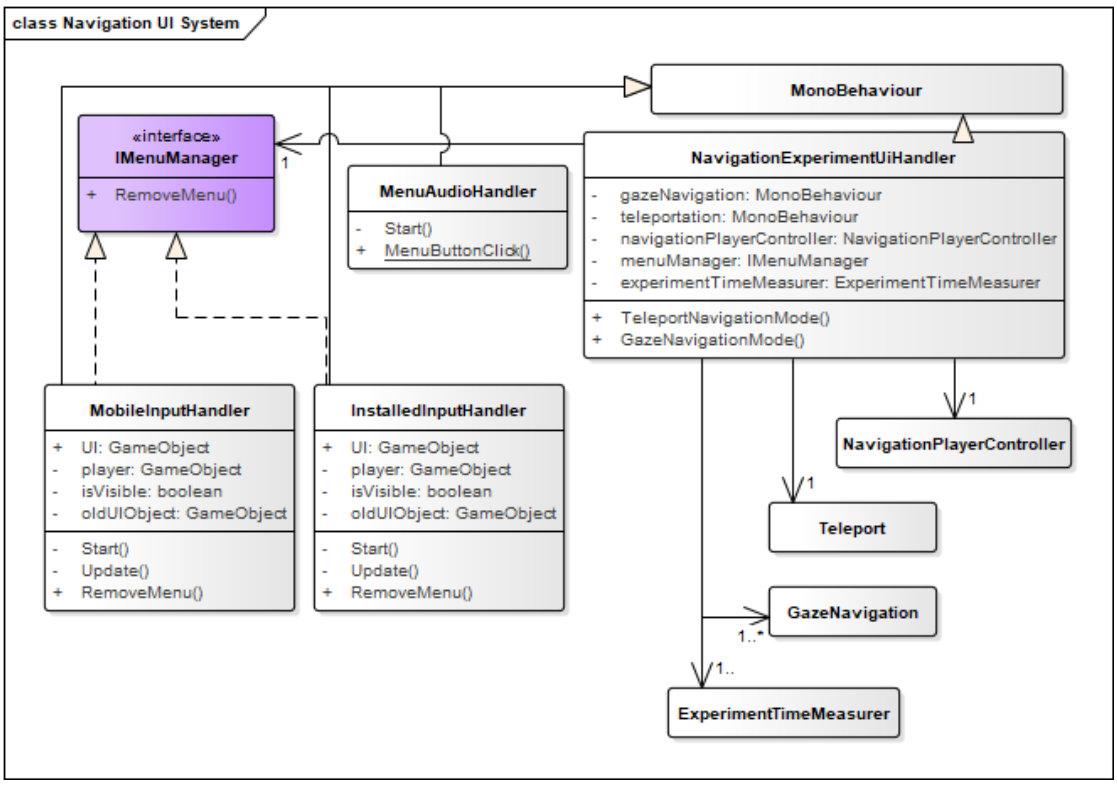

Figure 69: A class diagram of the classes comprising the menu system of the navigation experiment.

From the class diagram of figure 69, there are important concepts that make up the menu system. They are: *Menu Managers*, *MenuAudioHandler*, and the *UI Handler*.

The *NavigationExperimentUiHandler* is the UI handler. This script contains two public methods, *TeleportNavigationMode* and *GazeNavigationMode*. These methods are responsible for toggling the navigation mode which the player will use to navigate the virtual environment. These methods are hooked up to click events on buttons in the UI. This script is platform agnostic.

The menu managers handles the menu GameObject of the virtual world. These scripts have the responsibility of reading the input of the platform-specific controller, and from that hide or display the menu in the virtual environment. Additionally, these menu managers can be used by the *NavigationExperimentUiHandler* to remove the menu from the virtual world.

The *MenuAudioHandler* is used to play a button click sound when a button is clicked.

The general flow of these scripts, beginning from user input, can be seen in the sequence diagram of figure 70.

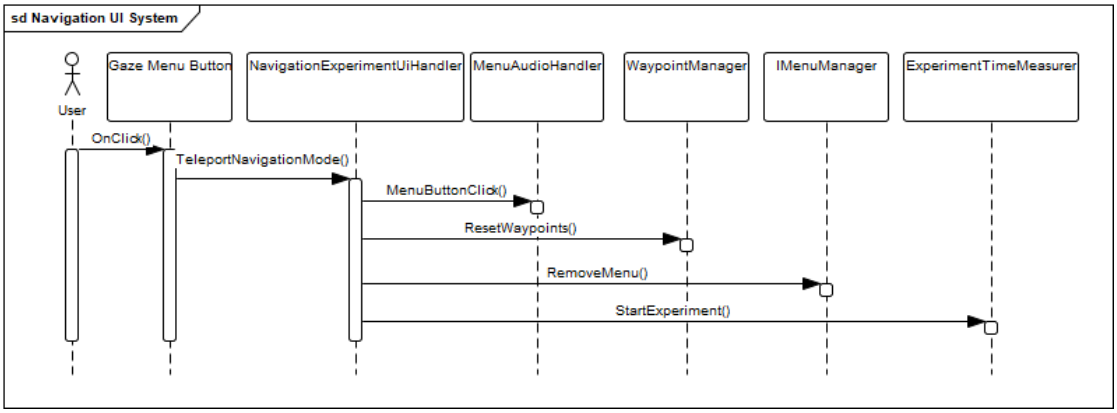

Figure 70: A sequence of the typical flow of the menu system from the point of a user clicking a button on the navigation experiment menu. This example is from the point of reference of the Gaze menu button being clicked.

LookAtPlayer Script

The menu system also consists of a script called *LookAtPlayer*. The sole responsibility of this script is simply to constantly rotate the menu to face the player.

7.9.2 Prefab

The shared prefab of the menu, shared amongst both platform scenes, is essentially a hierarchy of GameObjects. A screenshot of the structure of the menu can be seen in figure 71.

Figure 71: Screenshot of the structure of the shared navigation menu.

These GameObjects make up all the visual makeup of the menu shown in figure 68, without any of the platform specific scripts attached.

Here, the responsibility of each GameObject in the hierarchy will be explained:

- *SharedNavigationUI*: This is the GameObject which acts as a parent for all other objects making up the menu. It contains the *LookAtPlayer* script.

- *UIHandler*: This GameObject contains the *NavigationExperimentUiHandler* script. All menu button clicks are routed to the methods of this script.
- *Background*: This is simply the GameObject for the visual dark-transparent background seen on the menu.
- *Experiment Selection*: This GameObject is a UI canvas. In Unity, it is required that all UI buttons be a parent of a canvas.
- *TeleportationButton*: This is the GameObject making up the button on the menu that allows the user to select *Teleportation*.
- *GazeButton*: This is the GameObject making up the button on the menu that allows the user to select *Gaze*.
- *Title*: This is the GameObject making up the text title "Navigation Modes" seen at the top of the menu.

From this shared prefab there exists two platform-specific menu prefabs. The reason for this is that whilst they share the same general structure, each platform requires certain platform-specific scripts to be attached to the button GameObjects in order to work with respectively SteamVR on the installed platform and Google Daydream on the mobile platform.

8 Prototypes

9 Virtual Reality Prototypes

A lot of the technologies which we worked with during the development of this project were new to us. Because of this, and our focus on learning and building described in our project process of section **??**, we made extensive use of prototyping throughout the project. The key idea behind this was to gain a proper understanding of what we were about to work with, in order to gain valuable insight into potential problems and possibilities. This helped us remove as much uncertainty and risk as possible.

9.1 Mobile

The very first mobile VR prototype was a simple demo scene provided by Google in their official GoogleVR Unity package. A screenshot of this demo scene can be seen in figure 72.

Figure 72: A screenshot from the initial mobile vr prototype. Here is a simple 3D render of the scene as seen from the headset.

This same scene can be seen from the Unity editor in figure 73.

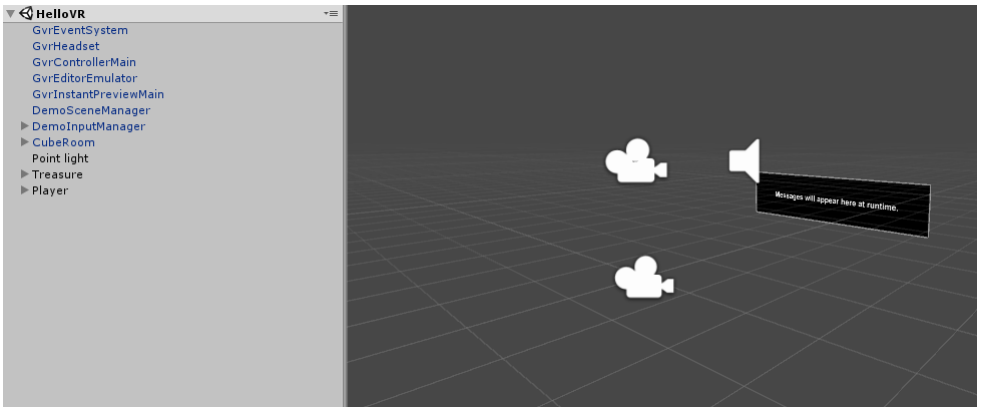

Figure 73: A screenshot from the initial mobile vr prototype. This perspective is from inside the Unity editor.

As can be seen, this is a very primitive scene essentially consisting of nothing but the ability to look around with the HMD.

The purpose of this prototype was simply to gain experience with the build workflow in Unity, when building the game to an Android smartphone.

Doing this prototype was a great learning experience, as some difficulties was discovered regarding the build process from Unity to the smartphone. Poorly documented setup of the Android SDK - which is required when building the game to an Android smartphone - meant that we had to spend quite a bit of time figuring out errors reported from within Unity when building the project. The importance of us discovernig this early was that we were able to spend the required time fixing a central issue to the rest of the project development.

9.2 Installed

We made a quick prototype of our virtual reality application on HTC Vive in order to gain a better understanding of what is required to set it up properly in Unity. A screenshot of this prototype running on a desktop computer with HTC Vive can be seen on figure 74.

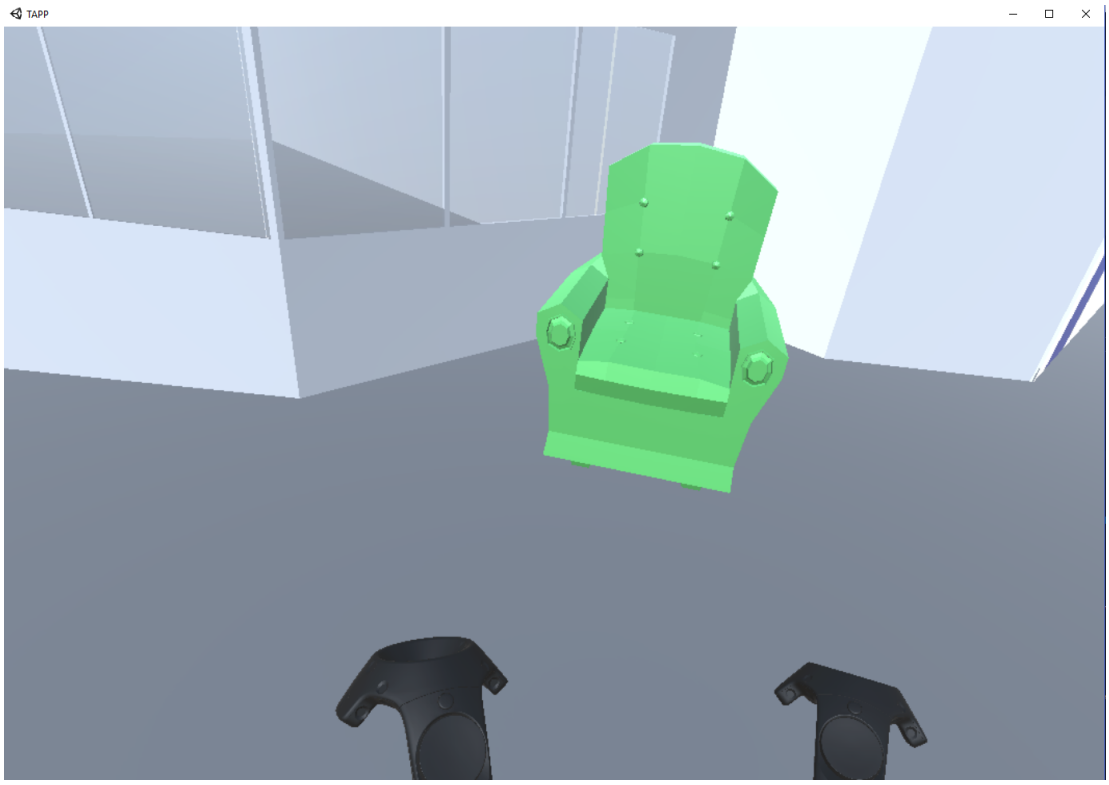

Figure 74: A screenshot from the installed VR prototype

As can be seen from figure 74, the prototype comes with fully modelled HTC Vive controllers tracked in real time. The prototype takes place within an early virtual environment of Orbit Lab.

What we learned from the prototype was the basic requirements for setting up HTC Vive in a Unity project.

First of all, *Valve* supplies the *SteamVR* Unity asset through the Unity Asset Store [**?**]. By importing this package to your Unity project, you are supplied with documentation on how to implement basic installed VR functionality such as head-tracking and tracking of controllers. Furthermore, the package includes prefabs for these essential things so that it works out of the box in a configuration which Valve recommends. Figure 75 shows a screenshot of the many different folders of prefabs that comes with it.

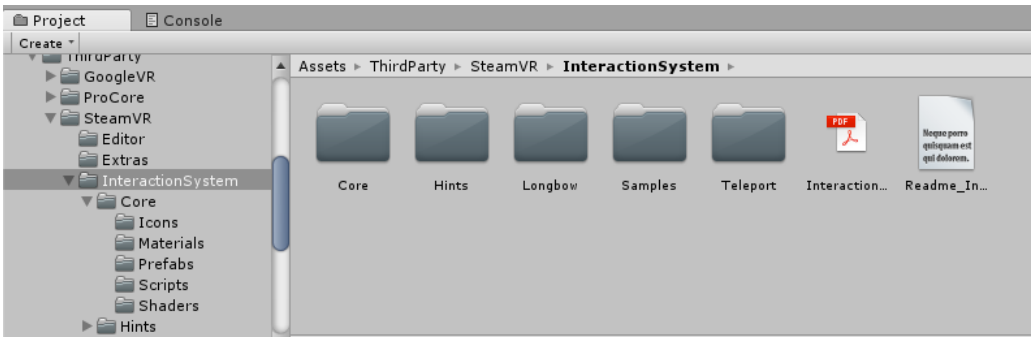

Figure 75: A screenshot of the many folders of provided prefabs that ships with the SteamVR asset

One thing that we did learn from this initial prototype was that the included documentation on the various prefabs was a bit sparse. In the *Core* folder seen in figure 75, for example, SteamVR provides a *Player* prefab which is supposed to be a prefab representing a player in the virtual space, with head-mounted tracking and two tracked controllers. We used this prefab in our prototype, and found out that it is actually configured incorrectly. The prefab consists of a nested, outdated, prefab called *DebugUI*. This can be seen in figure 76.

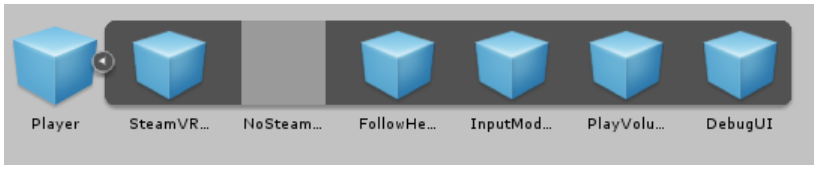

Figure 76: A screenshot of the Player prefab, with its nested prefabs folded out.

The *DebugUI* prefab causes rendering artifacts when playing the game on a desktop computer. The artifact consists of incorrectly projecting 3D models of the environment, resulting in them being rendering as flat 2D objects on the head-mounted display when looked at from specific angles. Tracking down this bug meant that we managed to identify a potentially disturbing bug early on in the process.

9.3 UI Prototypes

We knew that developing the experiments of the project meant that some type of UI would have to be implemented in our virtual reality applications.

Therefore, before implementing our experiments, we made prototypes of how to do UI programming with virtual reality on the installed and mobile platforms. We did this in order to gain a quick understanding of what is required in order to do it, and if we would bump into potential difficulties in doing so.

9.3.1 Mobile

For UI programming with Google Daydream, we made a scene dedicated solely to UI prototyping. Figure 77 shows a screenshot of us playing this scene.

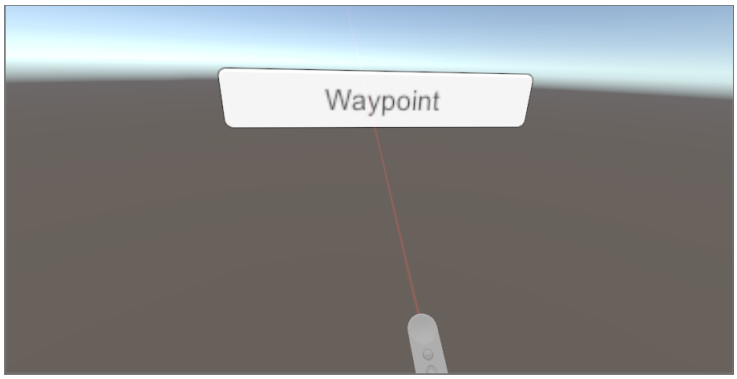

Figure 77: A screenshot from the Google Daydream UI prototype scene.

In figure 77, it can be seen that the scene contains only a button. This was all that was required in order to play around with UI programming, specifically how to detect the player pointing on the button, and how to detect click events.

In figure 78 it can be seen that the button is being highlighted, as the laser pointer is hitting it.

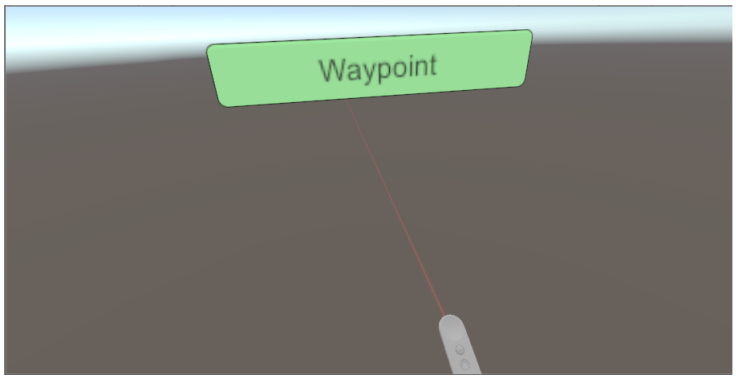

Figure 78: A screenshot from the Google Daydream UI prototype scene. Here, the button is highlighted as the laser pointer is hitting it.

It turned out that doing UI programming with Google Daydream was relatively easy. In order to get buttons to work, you need to follow a specific structure to the GameObjects composing the menu. Essentially, you need a GameObject with the *Canvas* component on. This is a component provided by Unity for UI programming. On this same GameObject,

you also need to add the *GvrPointerGraphicRaycaster* component. This is a component provided by the Google Daydream SDK, and makes sure that the raycast from the Google Daydream controller is detected when pointing on the buttons.

Nested within this GameObject, you then place all the buttons comprising the menu. These GameObjects need no Google Daydream specific components. Instead, they only need the *Button* component provided by Unity.

A screenshot of this structure within one of our Unity scenes can be seen in figure 79.

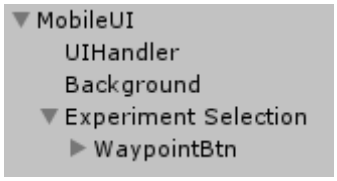

Figure 79: A screenshot of the GameObject structure of a UI menu for Mobile VR.

Here, the *Experiment Selection* GameObject is contains the *Canvas* and *GvrPointerGraphicRaycaster* components. Nested within this GameObject is the *WaypointBtn*, which is the actual button of the UI.

One thing to point out is that we were unable to find any official documentation on doing this. What we had to do was examine demo scenes which came with the *Google VR SDK* package described in section 6.2. Some of these demo scenes had menus in them, and from that we were able to deduce the setup required.

We learned that in order to handle click events, the Unity component *Button*, which in this case is added to the *WaypointBtn* GameObject, contains an OnClick configuration, in which you can specify a method on a script that should be executed when a click occurs. A screenshot of this can be seen in figure 80. In this case, we have specified that the method *TeleportationNavigationMode* should be executed once this button is clicked. This method is contained within our own *UIHandler* script.

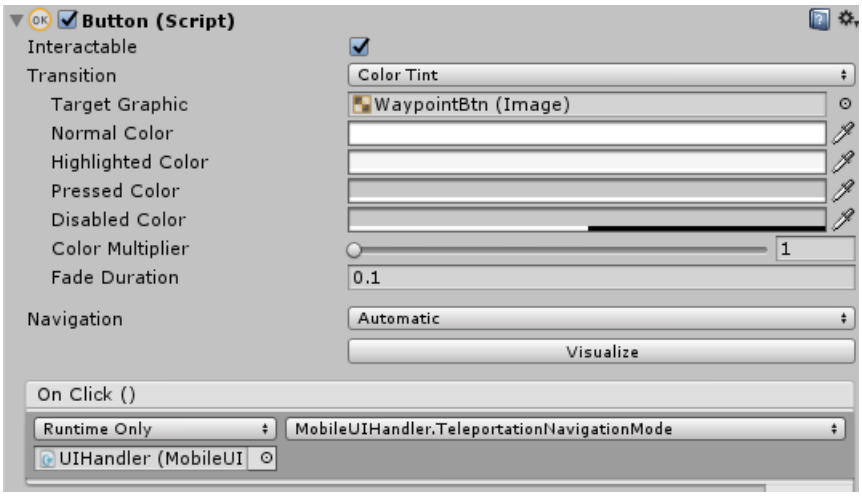

Figure 80: A screenshot of the GameObject structure of a UI menu for Mobile VR.

In the end, we got the prototype to register clicks. This prototype was helpful for us in that we learned the essentials of UI programming for Google Daydream, and found out that it did not have any initial problems. We could thus use this knowledge for the real implementations within the experiments during our project.

9.3.2 Installed

For UI programming with the HTC Vive, we made a scene dedicated solely to UI prototyping.

As we made the UI prototype for Google Daydream first, we had already gained some initial knowledge of the Unity-specific components related to UI programming. Specifically *Button* and *Canvas*. Thus section 9.3.1 can be seen for a description of this setup. These components are required in the same way when doing HTC Vive UI programming.

What was relevant to this prototype was figuring out if any HTC Vive specific components would have to be used for UI programming on the desktop platform.

What we learned was that documentation on this aspect was very sparse. The documentation which follows with the *SteamVR Plugin* package described in section 6.2 essentially says nothing about UI programming.

Thus, in order to learn about UI programming for the HTC Vive, we had to examine demo scenes that were included with the SteamVR Plugin package. From doing this, we were able to deduce the setup required.

Figure 81 shows a screenshot of us playing the scene.

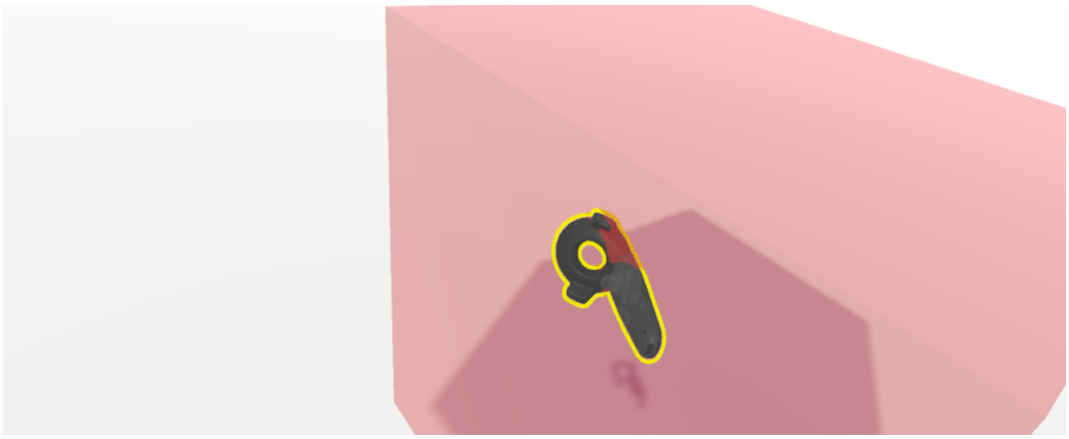

Figure 81: A screenshot of the HTC Vive UI prototype scene. Here, it can be seen that the HTC Vive controller is highlighted as we touch the red cube.

In the end, we learned that UI programming for HTC Vive was relatively simple. In order to interact with buttons, we found that two components were required: *Interactable* and *UI Element*. These components are part of the *SteamVR Plugin* package. They are added to the same GameObject which contains the Unity-specific *Button* component.

9.4 Installed Teleportation Prototype

When it came to virtual reality programming of the HTC Vive, we wanted to explore the possibilities of navigation through teleportation.

9.5 Mobile Remote Object Manipulation Prototype

Before starting the primary *Base VR Apps* development lane for the mobile VR application, we spend a few days developing a prototype for it.

The purpose of this prototype was to get familiar with Unity development on a smartphone in relation to Virtual Reality with Google Daydream. We wanted to gain early experience with potential difficulties associated with the official Unity SDK provided by Google for Google Daydream.

Figure 82 and figure 83 show the final results of this initial mobile VR prototype.

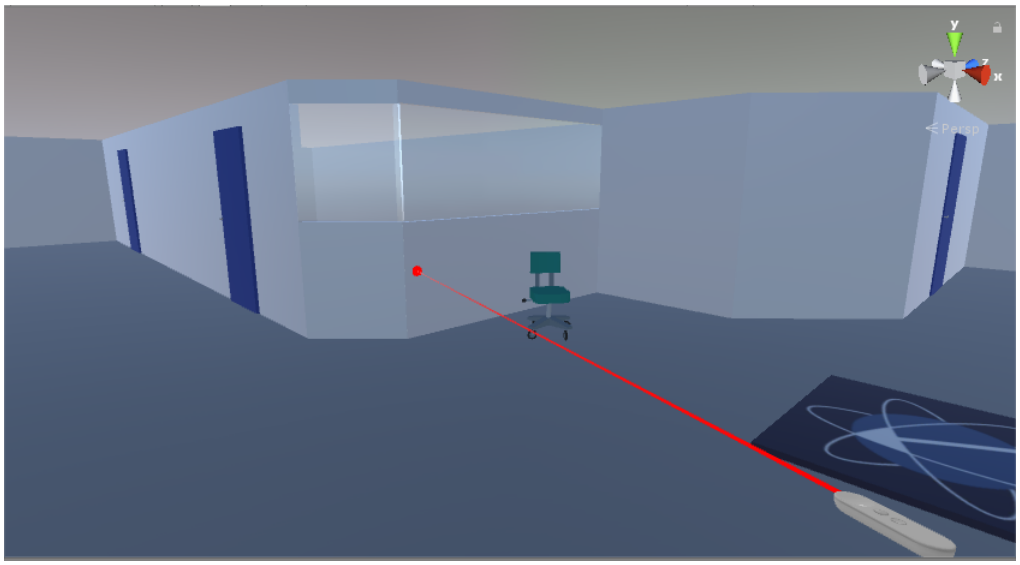

Figure 82: A screenshot from the mobile VR prototype with the Google Daydream controller prefab

On figure 82, a screenshot is shown from the user's perspective. Here, the virtual environment is an early prototypical model of Orbit Lab. In the bottom right corner of the screen it is possible to see the offical Google Daydream controller prefab in use. The 3D model is thus provided by Google, as well as all the script logic associated with orienting and moving the controller in the 3D world.

Also visible is the laser point projecting forward from the controller. It's also possible to see the hit location of this laser pointer on the wall, shown as the red dot on the wall opposite of the controller. The laser pointer and the underlying raycast logic to hit the surrounding virtual environment was scripted developed by the team. The point here was to get a feel of the possibilities and potential difficulties with creating laser pointer logic.

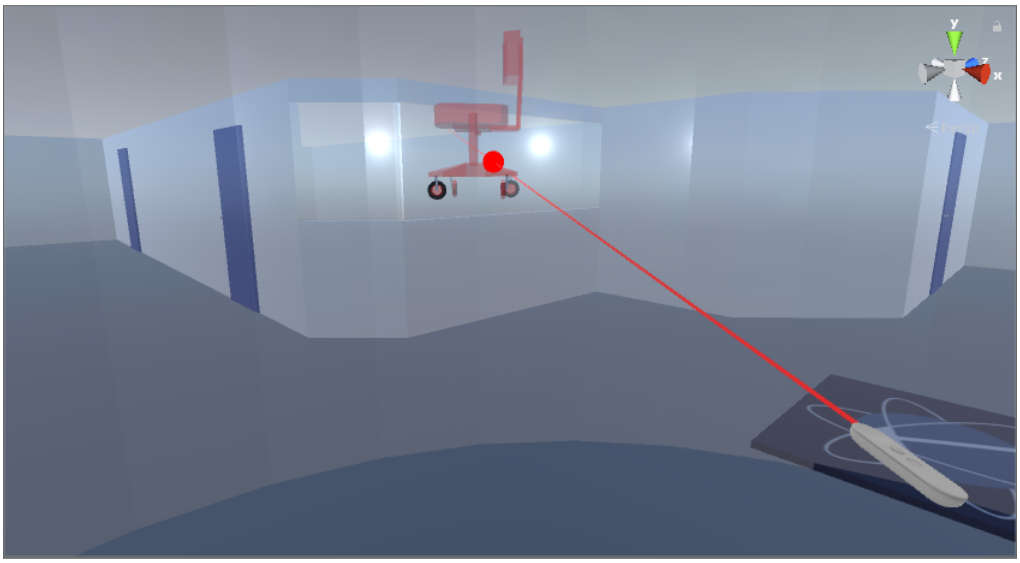

Figure 83: A screenshot from the mobile VR prototype in which the user has selected, and is moving, a furniture

In figure 83, a screenshot is shown of the same user perspective, this time where a furniture is being highlighted by the controller's laser pointer, as the user is pointing on it and moving it. The purpose of this was to develop a quick object manipulation model, to develop an idea of potential difficulties in Unity's 3D programming when coupled with virtual reality.

Overall, this initial prototype gave us a great idea of the baselines workflow of Google Daydream development with Unity. Additionally, we were able to get a feeling for the 3D programming aspect of Unity, and test it out with primary features of our final product.

Finally, from this prototype, we also realized how much work that goes into making remote object interaction which feels right and natural. So, this prototype also helped us determine that it would make the most sense to use Google Daydream's *Elements*, which has a best-practice remote object interaction model that fits well with the requirements for our project.

9.6 Installed Remote Object Manipulation Prototypes

During the development phase of the final experiment, we spend quite a while investigation the possibilities for remote object manipulation on the installed platform.

What we ultimately found was that there were no ready-made professional solutions that covered our requirements for remote object manipulation. This investigation was conducted by prototyping some simple scenes in Unity using various toolkits from the

Unity Asset Store - namely *Virtual Reality Toolkit* and *NewtonVr*. These prototypes will be presented here.

9.6.1 Virtual Reality Toolkit Prototype

Virtual Reality Toolkit (VRTK) is a toolkit on the Unity Asset Store which includes various cross-platform VR components typically used in games. This includes things such as: Teleportation, touchpad movement, 3D controls (buttons, levers, drawers, knobs, etc...), ready-made menus and so forth. Our idea was that this toolkit might also have some ready-made functionality for remote object manipulation which we could use.

Figure 84 shows a screenshot of our remote object manipulation prototype using VRTK.

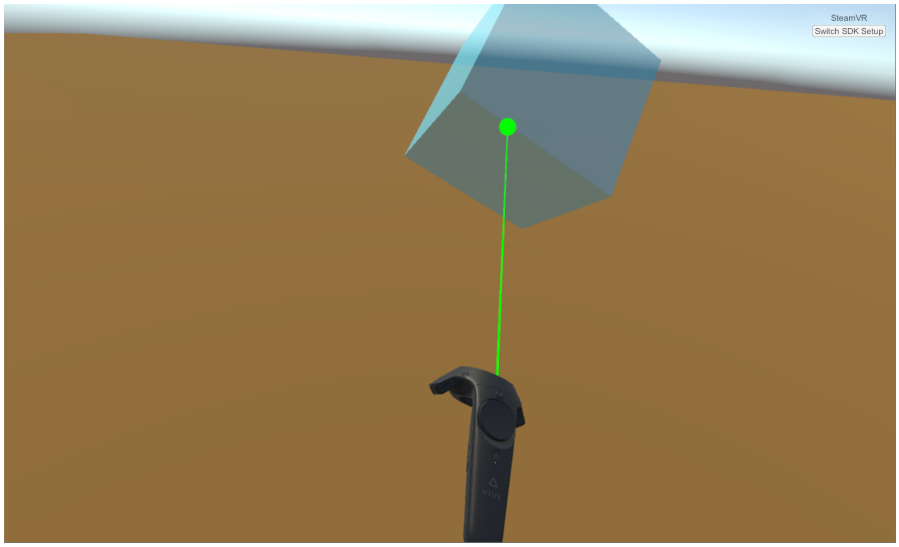

Figure 84: A screenshot from the remote object manipulation prototype using VRTK.

In figure 84 it can be seen that we did get remote object manipulation to work with VRTK. The toolkit included ready-made functionality for it. Unfortunately, the functionality present was too simplistic for our needs. There was no possibility of, for example, rotating the object currently selected using the touchpad. It was also not possible to adjust the distance between you want the selected object. This distance was fixed based on the length between you and the object at the point of picking it up with the laser pointer.

We also experimented with simply extending the already existing functionality of VRTK with our own custom scripts, however we found that this was not practical. The toolkit did not have the needed extensibility that we needed for doing what we wanted with remote object manipulation.

Thus it was decided that VRTK was not going to be a practical option for us.

9.6.2 NewtonVr Prototype

NewtonVr is a cross-platform VR interaction toolkit on the Unity Asset Store. Just like VRTK, it includes various cross-platform VR components typically used in games, however NewtonVr focuses primarily on interaction systems. Our idea was that this toolkit might include the type of remote object manipulation that we needed.

Figure 85 shows a screenshot of a pre-made demo scene included in the NewtonVr Unity package. The purpose of this scene was to demonstrate the various interaction systems provided by the toolkit.

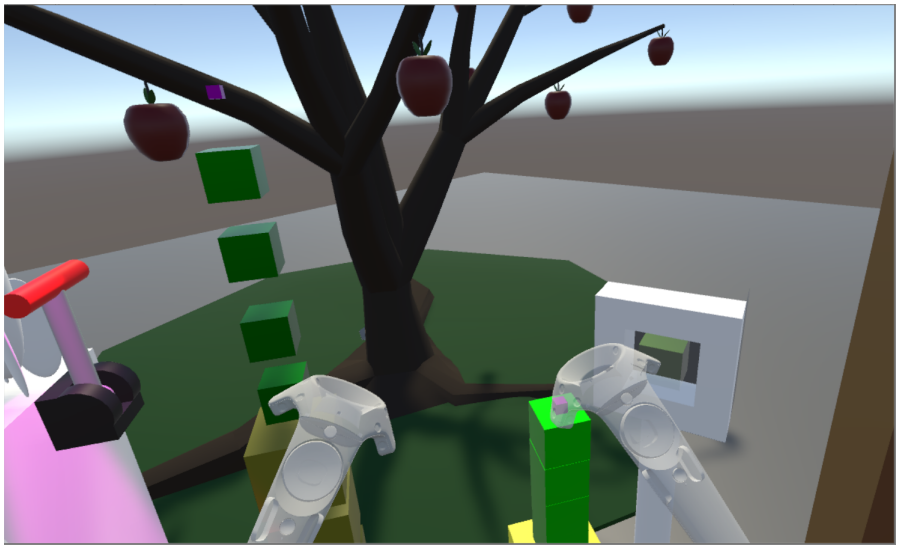

Figure 85: A screenshot of the included demo scene of NewtonVr.

In figure 85, it is possible to see some of the interaction systems provided by NewtonVr, such as the lever seen on the left, or the physics-based apples on the tree which can be plucked. Besides this, the scene also included demonstrations of doors, drawers, menus, and more. Unfortunately, this toolkit seemed to have to focus on remote object manipulation. Therefore, there was no functionality within the package that was deemed fit for our needs.

Thus it was decided that NewtonVr would not be a practical option for our project.