Project: Firetruck
A Major Qualifying Project
Submitted to the Faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
Degree of Bachelor of Science

Christian Cedron

Yang Gao

Neer Jariwala

Shiyi Liu

October 15, 2019

Professor Ralph Sutter, Advisor, WPI

Professor Joshua Cuneo, Advisor, WPI

Professor Jason Orlosky, Advisor, Osaka University

Professor Photchara Ratsamee, Advisor, Osaka University
Acknowledgements

The completion of this project would not have been possible without the amazing guidance provided by all of our professors from WPI and Osaka University. We would like to specifically thank Professor Ralph Sutter and Professor Joshua Cuneo from WPI for guiding us through the length of this project. We would also like to extend our gratitude towards Takemura-Sensei, Professor Photchara Ratsamee and Professor Jason Orlosky for allowing us to join them on this project and welcoming us into their lab.
Abstract

The main purpose of this project was to develop a working VR environment to test different forms of data visualization. The environment is themed around a fire station dispatch, where users have to put out fires as they appear on the 3D holographic map in front of them. The visualizations that we chose to test were developed to possibly assist the user in their task. Our visualization options included a HUD and a numerical display of the size of the fire and the amount of water in a fire truck. As this project was a portion of an ongoing, larger project, development will continue long after our role has been completed. However, we have created an advanced environment with documentation that will allow future contributors to use our work with ease.
Executive Summary

The goal of this project was to create new data visualizations to test in a VR fire dispatch setting that would help improve user performance. This project is a continuation of an existing project being developed under Jason Orlosky and his collaborators at Osaka University. He had developed the core of the experiment by the time we began work, and we added on to it via our own visualizations and some logic modifications.

Since the previous versions of the study concentrated more on imitating a real-world environment without fully exploring the VR capability, our group decided to test on more idealistic interactions. In the previous project, in order to display the information, the data was originally displayed on 3D assets of monitors within the Unity environment. In an attempt to make the data more approachable and usable, the user was presented with different, smaller, but less descriptive forms of displayed data. These were not intended to replace the monitors but rather to guide the user to the monitors that were worth looking at. We then took this idea and flipped it on its head. We removed the monitors in favour of a Heads Up Display (HUD); the flat 2D map was replaced by a large hologram representation where buildings would catch on fire. Also, the three screens were replaced by one larger screen to show information on all currently available fire trucks. To test the performance of the users, four conditions were designed for research purposes. The conditions included: the HUD on with the number system, the HUD on without the number system, the HUD off with the number system, and the HUD off without the number system.

During the designing and programming time, we met several obstacles. Two main issues were a learning curve regarding technology and Google Maps Platform’s Unity SDK. Our learning curve mainly included not fully knowing the functionality of Unity and its built in functions. Google Maps Platform’s Unity SDK was our main problem mainly because full access to the entire platform cost $200 per month. The free demo version of the SDK only allowed access to New York City’s map. However, the map was only rendered in play mode, and each building in the map held duplicate object references. Within this main issue, some designs we had were not practical to implement, and some alternative solutions were raised which made the design even more complicated.

Due to the limited time we had, there were other ideas which could not be implemented, including changing the map depending on the user’s location, having real-life simulations, designing different room settings and information displays, and making the project executable. We considered changing the map depending on the user’s location before we knew about the cost of the maps platform. If we were able to get full access, we could plan to implement all the cities in our project in order to see whether the hologram is feasible to all kinds of cities. Having a realistic feel was another focus of our project. To accomplish this, we took inspiration from a real fire dispatch station. More realistic conditions, including the fuel amount and number of firefighters, would contribute more to a real fire dispatch station. By designing different room settings and information displays, we could test out the most effective and useful way to help users perform better. We also wanted to make the project executable so that the testing would not be constrained to Unity’s editor mode.
We hope our project will not only further the work that Jason and his collaborators have already completed, but also help the researchers who will continue to use this project for their work in data visualization for VR.
# Table of Contents

Acknowledgements .................................................. i
Abstract .................................................................. ii
Executive Summary .................................................. iii
Table of Contents ...................................................... v
List of Figures .......................................................... vii

## 1 Introduction
1.1 Roles ................................................................. 1

## 2 Background
2.1 The Game Engine .................................................. 2
2.2 Classical Fire Station Dispatch Center ......................... 2
2.3 SteamVR .............................................................. 3
2.3.1 OpenVR .......................................................... 3
2.4 Valve - HTC Vive Pro .............................................. 3
2.4.1 HTC Vive Pro Headset ......................................... 4
2.4.2 HTC Vive Pro Controller ...................................... 4
2.5 Raycast ............................................................... 4
2.6 Roll-a-Ball and Basic Unity ....................................... 5
2.7 Shader Graph ....................................................... 6
2.8 Nav Mesh ............................................................. 7
2.9 Google Maps Platform ............................................. 7

## 3 Methodology
3.1 Existing Research and Code Base ................................. 7
3.2 Motion Sickness ..................................................... 9
3.3 Scene ................................................................. 9
3.4 Interaction ........................................................... 10
3.5 UI Design ............................................................ 13
3.6 Four Conditions .................................................... 14
3.7 Assets ............................................................... 16
3.7.1 Initial Map ....................................................... 19
3.7.2 Shader Work ................................................... 19
3.7.3 New York City Map ........................................... 21
3.7.4 Building the navigation Mesh ............................... 22
3.8 Coding ............................................................... 23
3.8.1 Fire Trucks ...................................................... 23
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Fire Station dispatch center example layout</td>
<td>2</td>
</tr>
<tr>
<td>2.2</td>
<td>SteamVR/OpenVR Event System Work Flow</td>
<td>3</td>
</tr>
<tr>
<td>2.3</td>
<td>HTC Vive Pro controller functionality</td>
<td>4</td>
</tr>
<tr>
<td>2.4</td>
<td>HTC Vive Pro Controller Button Layout</td>
<td>5</td>
</tr>
<tr>
<td>2.5</td>
<td>Roll-a-Ball level back view</td>
<td>5</td>
</tr>
<tr>
<td>2.6</td>
<td>Roll-a-Ball level front view</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>Original experimental room</td>
<td>8</td>
</tr>
<tr>
<td>3.2</td>
<td>Hand selection from The Lab (Valve Corporation, 2016) &amp; Pointing selection from The Lab (Valve Corporation, 2016)</td>
<td>11</td>
</tr>
<tr>
<td>3.3</td>
<td>Hand Manipulation from The Lab (Valve Corporation, 2016)</td>
<td>11</td>
</tr>
<tr>
<td>3.4</td>
<td>Indirect Control from The Lab (Valve Corporation, 2016)</td>
<td>12</td>
</tr>
<tr>
<td>3.5</td>
<td>Comfortable range of head movement (Jaime, 2018)</td>
<td>13</td>
</tr>
<tr>
<td>3.6</td>
<td>360° view of user comfort zones (Jaime, 2018)</td>
<td>13</td>
</tr>
<tr>
<td>3.7</td>
<td>Condition 1, without HUD and without numbers</td>
<td>14</td>
</tr>
<tr>
<td>3.8</td>
<td>Condition 2, with HUD and without numbers</td>
<td>15</td>
</tr>
<tr>
<td>3.9</td>
<td>Condition 3, with HUD and with numbers</td>
<td>15</td>
</tr>
<tr>
<td>3.10</td>
<td>Condition 4, without HUD and with numbers</td>
<td>16</td>
</tr>
<tr>
<td>3.11</td>
<td>Old environment with chairs, tables, and monitors</td>
<td>16</td>
</tr>
<tr>
<td>3.12</td>
<td>New stripped down environment with only player platform</td>
<td>17</td>
</tr>
<tr>
<td>3.13</td>
<td>Old environment with 3 screens</td>
<td>17</td>
</tr>
<tr>
<td>3.14</td>
<td>New environment with 1 main screen</td>
<td>17</td>
</tr>
<tr>
<td>3.15</td>
<td>Old VS new map</td>
<td>17</td>
</tr>
<tr>
<td>3.16</td>
<td>Hologram Map from the Destiny 2: Scourge of the Past Raid (Bungie, 2018)</td>
<td>18</td>
</tr>
<tr>
<td>3.17</td>
<td>Original Hologram Map top view</td>
<td>19</td>
</tr>
<tr>
<td>3.18</td>
<td>Original Hologram Map side view</td>
<td>19</td>
</tr>
<tr>
<td>3.19</td>
<td>HBase Shader</td>
<td>20</td>
</tr>
<tr>
<td>3.20</td>
<td>Main Hologram material shader</td>
<td>20</td>
</tr>
<tr>
<td>3.21</td>
<td>25% fire hologram shader</td>
<td>21</td>
</tr>
<tr>
<td>3.22</td>
<td>Failed Animated Fire Shader</td>
<td>21</td>
</tr>
<tr>
<td>3.23</td>
<td>New York City Hologram Map in environment</td>
<td>22</td>
</tr>
<tr>
<td>3.24</td>
<td>New York City Navigation Model in Unity</td>
<td>22</td>
</tr>
<tr>
<td>3.25</td>
<td>Visual of only hitboxes</td>
<td>24</td>
</tr>
<tr>
<td>3.26</td>
<td>Fire truck HUD</td>
<td>25</td>
</tr>
<tr>
<td>3.27</td>
<td>New York City hologram map with all possible buildings on fire</td>
<td>25</td>
</tr>
<tr>
<td>3.28</td>
<td>Pop up menu for a building without a fire truck present</td>
<td>26</td>
</tr>
<tr>
<td>3.29</td>
<td>Pop up menu for a building with a fire truck present</td>
<td>26</td>
</tr>
<tr>
<td>3.30</td>
<td>Final HUD display</td>
<td>28</td>
</tr>
</tbody>
</table>
1 Introduction

The ability to visualize data is a powerful tool to convey information quickly and precisely. However, what form of visualization is the most efficient? More specifically, what form of visualization is the most efficient when working with virtual or augmented reality? Our project explored these questions by developing and improving on a VR environment to simulate the work of a fire station dispatch center. Our goal was to create a working VR environment that would test a few different visualizations based around improving user performance. Our environment was centered around a 3D hologram of a small portion of New York City. Another key component included a large screen on the opposite side of the room from the player that displayed each available fire truck and the amount of water in their tanks.

1.1 Roles

Our group was made up of 3 Computer Science majors and one IMGD major, with one CS major having minor experience in IMGD. So, in order to efficiently complete this project, we split our group into 2 main teams. One group focused more on features and code related aspects, and the other team focused on asset and environmental development. Within these teams, we further split work based off what our individual strengths were.

The asset team consisted of two members, Shiyi and Neer. This group focused on working with the environment and creating any required assets needed, fixing any shader or environment related bugs, and similar tasks. Shiyi, being the IMGD major, took on tasks such as developing the hologram shader and setting up navigation functionality. She also worked on taking care of the rendering pipeline and keeping the VR environment both effective and coherent. Neer worked on tasks such as restructuring and maintaining the environment from the version that we received at the start of the project. In addition, he worked on fixing asset bugs such as problems that occured with the controllers. In addition, both members worked on building the holographic New York City map.

The code team, made up of our other two members, Chris and Yang, worked together to annotate the existing code base, develop new features, and fix bugs that appeared along the development process. Some of the features added include a dynamic HUD, new management systems for the fire trucks, and a new event bus driven action system to replace the existing controller based system.
2 Background

2.1 The Game Engine

The game engine used throughout the extent of this project is Unity 2019.1.12 f1. Unity is the same engine that Jason Orlosky, the professor in charge of the project, had been using before our group had been brought into the project. The initial choice to use Unity was for ease of development and for integration with SteamVR’s Unity plugin. In addition, since the goal of the project is to test how users perform in VR with different data visualization, and only a portion of our group was vaguely familiar with VR and game design, Unity was useful in that it came with a large array of training tools to get us familiar with the engine. Unity allows us to implement various GameObjects, which are fundamental base objects in Unity that define parts of a scene. These can have scripts or other components attached that define the behaviour and characteristics of the GameObjects. The components can change many characteristics of the GameObject, such as a mesh renderer to control the way the object is rendered, or a material to change the way the GameObject looks. Our group also used 3ds Max 2018, a 3D computer graphics software from Autodesk for 3D modeling, animation, rendering and visualization to build our desired model.

2.2 Classical Fire Station Dispatch Center

A fire station dispatch center is designed to serve as a location where the fire dispatcher determines the location of the emergency, monitors and tracks the status, delivers and updates the information with area communication centers. Figure 2.1 shows a fire dispatch room layout.

![Fire Station dispatch center example layout]("Law Enforcement Assistance Administration (LEAA) Police Technical Assistance Report," n.d.)
2.3 SteamVR

In order to make our environment VR compatible, we required additional packages. Both the SteamVR unity plugin and the OpenVR SDK were required for our VR development. We used SteamVR 2.3.2 (SDK 1.4.18), which is a free asset package from the Unity asset store that has a collection of tools to support VR content. Inside of the Steam VR package, a variety of components are available to use. The most important prefab that we used was the cameraRig. This object represents the players in the VR environment and contains the camera and both controllers.

SteamVR also contains an action input system that adds a layer of abstraction between the hardware and Unity. In order to provide users a better and more customizable experience in VR, the event based action system allows developers to separate the invoked action from the button that triggers the action. With this, a developer can write code that reacts to user input while the user can define how they want to input that event. To achieve this, a developer defines actions which they can then refer to without having to poll a particular button, because once the action is invoked, any “listening” functions are automatically called. The user is given a simple and intuitive web application to bind the actions the developer defined to whatever input mechanism they see fit.

2.3.1 OpenVR

The OpenVR SDK is an API implemented by SteamVR that serves as the interface between the software and the VR equipment being used. This SDK allows users to easily build applications and work with their VR hardware. Figure 2.2 below is a simple diagram showing the workflow between the user and the VR equipment.

![Figure 2.2: SteamVR/OpenVR Event System Work Flow](image)

2.4 Valve - HTC Vive Pro

The VR hardware that we used for this project was the HTC Vive Pro. The Vive is mainly comprised of the headset, controllers, and cameras. The main reason we chose
to use the Vive over other VR hardware because it was the same technology used in previous version of the project. Furthermore, the entire system had already been set up in a large open area, making it easy to use compared to setting up an entire new system.

2.4.1 HTC Vive Pro Headset

The HTC Vive headset is the main piece of hardware that allows a user to see into VR. This headset is tracked by cameras to keep track of the player in a 3D environment. This allows the HTC system to warn the user if there about to collide with the edge of their VR space among other features. The headset is capable of a high refresh rate and has a high resolution panel, making it a good choice for our project.

2.4.2 HTC Vive Pro Controller

The HTC Vive comes with two interchangeable handheld controllers, one for each hand. As mentioned before, Unity supports OpenVR to track controllers. The Unity subsystem presents VR controller inputs as separate joysticks. By using the `UnityEngine.Input` class from SteamVR, it is possible to access the (x,y) position of the user’s finger on the trackpad and button boolean values. For example, the EventSystem in Unity generates input touch events for the triggers on HTC Vive Pro controller when the user begins squeezing the trigger. Following is a listing of the buttons on a HTC Vive Pro controller.

![Figure 2.3: HTC Vive Pro controller functionality](image)

2.5 Raycast

A raycast, by definition, is a ray emitted from a position that moves in a specific direction, usually detecting any game objects that it collides with. In Unity, `Raycast`
Figure 2.4: HTC Vive Pro Controller Button Layout

is a built-in function that normally takes in five parameters: origin position for the ray, direction vector to cast the ray in, the maximum distance that the ray should check, and a filter that defines what objects can be hit. The function also returns a boolean, indicating whether the ray collides or not, and sets a RaycastHit variable that contains more information on the target that was hit. The programmer can use the information to filter out the certain group of GameObjects by adding a tag to them if the layer mask didn’t already filter all invalid targets.

### 2.6 Roll-a-Ball and Basic Unity

Project: Roll-A-Ball is a basic introduction to the Unity software meant for first-time users. It covers the basics of adding objects to a scene, materials, simple physics, and camera placement. While not extensive, this learning experience was our team’s first task in our project. It allowed us to become familiar with the Unity layout and some of the features that we would use in the future in our own work. The Roll-A-Ball project came with a set of tutorials containing step by step instructions on how to build a simple game where the players would use the WASD keys to move a roll around a bordered platform and pick up some collectibles. Once all collectibles were picked up, the game would be complete and a game over text would be displayed on the screen. In order for each of our group members to get exposure to the software, we completed the project individually.

In addition, we took one of our versions and implemented it into a VR setting using OpenVR and SteamVR. The VR version of the game allows the user to be floating about the play area and control the ball with the controller trackpad. The user can also teleport around above the map using the trigger. Figures 2.5 and 2.6 show the levels one of our members designed that was put into VR.

Figure 2.5: Roll-a-Ball level back view
Figure 2.6: Roll-a-Ball level front view

Roll-A-Ball was an introduction for our group to the Unity software and to the basics of setting up a VR environment. While the entire game did not delve into the deeper parts of Unity or VR, it was a solid base for ideas and a launch point for the remainder of our project.

2.7 Shader Graph

In order to create the holographic look for our map, it was easier to do it through the shader graphs to create new shaders. A shader graph is an official shader visualization tool for Unity. It enables users to create shaders without actual coding. By using the shader graph, users are able to not only create new shaders but also share the node graph between multiple graphs and form their own shader graph nodes using C# and High Level Shader Language, a coding language for shaders. Moreover, the shader graph is available in 2018.1+ version of Unity; it is a feature of the new Scriptable Render Pipeline (SRP). Thus, switching the rendering pipeline from normal to SRP is necessary. The Scriptable Render Pipeline (SRP) contains two built-in rendering pipelines, the Lightweight Pipeline (LWRP) and the High-Definition Rendering Pipeline (HDRP). Both of them can be modified by the developers to adapt to their own requirements. To use a shader graph, users are required to either download a Render Pipeline package or create a new project via a template using the shader graph. Owing to the fact that the project was already in progress, the Lightweight Render Pipeline package was downloaded from the package manager and employed. The shader graph is divided into three major parts: a Blackboard for people to create properties, a Master Preview to show how the shader work in the pipeline, and the Shader Graph Window for users to build a graph framework for the shader. All of our hologram shaders were created by the shader graph.
2.8 Nav Mesh

A NavMesh is a series of polygon meshes which define where certain objects can traverse. Unity is able to create NavMesh through its navigation features. The process of creating the NavMesh in Unity is called baking. By selecting GameObjects which have been marked as Navigation Statics and then baking them to create the NavMesh, developers are able to visually see the walkable surface in the scene. Several settings, such as the voxel size or the agent height, of the NavMesh could also be adjusted to fit the need of the developer better. Nav mesh is an indispensable part for our MQP since it enables the truck to pathfind on the hologram.

However, due to the way out hologram map is generated, the NavMesh could not be generated directly, and new models had to be made to mimic the hologram. As a result, the NavMesh could successfully be generated. If an object wants to navigate in the scene, a NavMesh Agent component and a pathfinding script need to be applied on the object.

2.9 Google Maps Platform

The Google Maps platform is “a set of APIs and SDKs that allows developers to embed Google Maps into mobile apps and web pages, or to retrieve data from Google Maps” (“Google Maps Platform FAQ,” 2019). A part of this platform the Maps Unity SDK which “is a set of development tools, services, and ready-made assets, that extend the Unity game development environment with features that allow you to create real-world mobile games—easily” (“Maps Unity SDK overview,” 2019). This SDK allows for developers to access geographical data stored in the Google Maps database. Each building is shown in Unity as an individual GameObject. The map is generated using a script that adds a materials to each building upon entering play mode in order to make them visible.

3 Methodology

3.1 Existing Research and Code Base

During the first couple of weeks, we were given a report describing some of the tests that had already been conducted earlier on in the project’s timeline. The experiments tested the effects of visualization on logical task completion. The project team utilized a set of color coded and gaze aware visualization designs to manage both information priority and user attention. These designs will not only respond to whether or not an object has been observed a user, but are also color coded to represent the priority of the objects to help the user processing the incoming information in the environment.

The environment was designed to mimic fire dispatch during an environmental disaster. The user would be standing in a simulated dispatch center where they would act as the lead dispatcher. During the experiment, fires would catch and the user would have to dispatch trucks to fight them. Trucks were given a limited quantity of water, referred
to as their “water level”; buildings were given a numerical representation of the intensity of the fire, referred to as the “fire level”. In order to effectively put out a fire, the user would have to send trucks to fight fires that were smaller than the water the truck had available. As the experiment progressed, the task would become increasingly difficult, and the user would have more trucks to control and fires would spawn at a faster rate. An image of the environment can be seen in Figure 3.1.

![Figure 3.1: Original experimental room](image_url)

Three different types of visualization were used: *in-situ* augmentation and highlighting, HUD indicators, and Gaze Guided flashing. *In-situ* augmentation and highlighting highlights the border of the monitor where “the brightness of the augmentation is proportional to the square root of the value of the current tank level of the truck represented by that monitor” (Liu et al, n.d.). HUD indicators add n dots that represent the n highest water level monitors or add a bright border to the monitors to feed the user information about their fire trucks with only a glance. Gaze Guided Flashing refers to a flashing of color depending on whether the user has already attended to a certain portion of information, which helps the user to keep track of which tanks they had already checked.

The results of the previous experiments showed that “the HUD methods outperformed the others in terms of decreased head movement, that gaze distribution in the environment changed slightly between method, and that HUD was also rated subjectively as most distracting” (Liu et al, n.d.).

In order to better understand the project we were given, we combed through the existing code in an attempt to understand how the different components worked and where we could modify them to meet our needs. Following that we then went line by line of the major scripts and added comments along the way. This would later prove exceedingly useful as we could quickly find what bits of code we should modify or remove in their entirety. Looking at the existing code, we found that the project contained some main scripts that control key factors of the environment and experiment. `GameManager` controls the monitors, city blocks, moving to the next level, the length of each level, adding fire to the city blocks, and removing all the fires after
each level is done. **DispatchExperimentManager** stores eight different visualization conditions for testing while also recording and outputting the experiments’ data. **FiretruckManager** controls the `waterLevel`, `fireLevel`, and deployment of trucks to put our fires. **PCStateManager** works together with **ExperimentManager** to allow the monitor to have different visualizations. It also controls the fire truck information on each monitor, such as truck status, truck index, and water level. **BlockStateManager** updates a city block to show if it currently has a fire truck as well as updating the `fireLevel` of that block.

### 3.2 Motion Sickness

A lot of our research focused on combating motion sickness and other effects of discomfort commonly associated with VR. Some of the negative effects to avoid when working with VR are nausea, dizziness, eye strain, physical fatigue, and headaches. These conditions can occur due to a variety of reasons.

Latency, or lag, for example, is one of the most major causes. Lag can originate due to issues with the tracking, the application, rendering, and the display. Having a problem in any one of these areas might cause the user to feel motion sick due to a disconnect between their head motion and movement in VR. In order to avoid this problem, an optimized code that keeps all latency at or below 20ms is needed, as 20ms is the most comfortable limit to what the human eye can handle, according to Oculus (“Oculus best practices,” 2007).

Motion sickness is one of the most common forms of illness a person can experience while in VR and can be caused by, but is not limited to, the following: speed of movement and acceleration, degree of control, duration, altitude, binocular display, wide field of view, lag, distortion correction, flicker, and/or experience. One possible counter to motion sickness is to have a rest frame. A rest frame is a stationary part of the scene that allows the brain to judge relative motion. This allows for the vestibular and visual system to work cooperatively. In addition, when using teleportation, having the screen fade to black right before the user is moved and fade out of black right after helps reduce the possibility of motion sickness. Giving the user full control of their movement and keeping the camera with the user at all times will allow the user to feel more comfortable and in turn less likely to become motion sick. In terms of brightness and imagery, keeping an overall moderate level of brightness and avoiding patterned textures which convey movement is more likely to not cause the user problems.

### 3.3 Scene

In any game or VR environment, the scene can be split up into 4 parts: background, contextual geometry, fundamental geometry, and interactive objects. For example, the background would be the skybox, the contextual geometry is the terrain that helps the user understand where they are, the fundamental geometry would be objects such as chairs, tables, and walls, and interactive objects are the parts of the scene that a user can have a direct impact on. To have the user be focused on their task, the scene should
be designed in a way that directs attention to the part of the scene that is important, for example, using a “traffic light” color pallet to differentiate bad (using red) from good (using green). Also, using directional cues or landmarks to direct the user to different parts of the scene can be helpful for the user to understand what the current objective is.

Overall it is important to not make the scene too busy and overload the user with visual clutter. To avoid this, the use of design elements such as space, colors, levels of transparency, shapes, and orientation is essential. When building the scene, avoid moving objects into the user’s personal space. Having hard edges and sharp objects near the user’s face can cause some comfort issues and possibly even make the user feel as if they are in danger. In order to increase immersion, the scene could use depth, dimensionality, and even binaural audio. Depth and dimensionality includes features such as occlusion, shading, the use of lighting, shadows, and linear perspective. Binaural audio refers to when a dual microphone is used to record sound. This setup is used to create a 3D effect that allows the listener to hear the recording almost as if it was live.

### 3.4 Interaction

Interaction is one of the most essential parts of making an immersive VR experience as it allows the user actually feel like they are a part of the environment. Interaction fidelity is how similar the physical actions used for a virtual task correspond to the physical actions used in the equivalent real-world task. These actions should have some form of balance between being completely realistic and completely artificial, allowing the user to have intuitive controls. Another option is to give the user supernatural abilities based off a basic motion, which can also be referred to as magical interactions. As an example, the user could move their arms as if they are running, but in the VR environment, they would be moving much faster than humanly possible.

There are many patterns of interaction: selection, manipulation, viewpoint control, indirect control, and a compound pattern. Selection patterns consist of hand and pointing selection. Hand selection is the method that uses the user’s actual hand to interact with objects. Although it has limited accuracy, it is realistic, straightforward, and is good for a user who is working with objects directly in front of them in the environment. Pointing selection, such as gaze or laser, is efficient and beneficial for speed selection situations. An additional benefit for pointing selection is the ability to add an object snapping or precision mode to assist the user.
Manipulation patterns include the direct hand and proxy methods. While direct hand is more intuitive, efficient, and satisfying, the proxy methods allows a user control a proxy object when the target object itself is too large or not in view.

Viewpoint control allows for the manipulation of orientation, perspective, and scale. The user would be able to, for example, scale a map to see the smaller details or rotate...
Indirect Control includes methods such as widgets/panels and gestures. Having a menu on the handheld VR controllers would be an example of widgets/panels. From this menu, the user could have options such as switching modes between normal and x-ray vision. While gestures are a good way to get the user to move around more, any overuse will cause fatigue.

The compound methods are widely inclusive, containing any patterns that combine the aspects of the previously mentioned patterns, possibly making up for some drawbacks of one pattern with the other. As an example, mixing the hand selection and pointing selection could result in some interesting usage of intuitiveness of the hand selection with the speed of pointing selection.
3.5 UI Design

The design of a VR environment is also crucial during the development process. Building for a variety of users and with different mental models will allow for more inclusive set of test users. If there is a need, the users should be given a specific set of instructions on the controls or objectives as to not cause confusion during play. The amount of information the user needs to digest should also be limited to about 5 to 7 chunks for an optimal performance (Jaime, 2018). Also, if some tasks are binary, buttons should be used. The Gestalt and Design principles claim that “Items should be scaled appropriately and well-spaced to ensure users can hit the correct target, without accidentally triggering targets nearby (Jaime, 2018)”. In addition for a proper scale, placement is also a key factor. When designing the UI, for example, it should be placed in a location that is easy to work with and read. The location should not be too close or too far from the user so as to not make it hard to read or too close for comfort.

![Comfortable range of head movement](figure3.5.png)

Figure 3.5: Comfortable range of head movement (Jaime, 2018)

![360° view of user comfort zones](figure3.6.png)

Figure 3.6: 360° view of user comfort zones (Jaime, 2018)
In order to create a comfortable and sustainable interactions, the range of movement a user should be required to use should be restricted to a comfortable range as demonstrated by Figures 3.5 and 3.6. Having a large range of movement has the chance to cause fatigue and discomfort.

### 3.6 Four Conditions

We were aiming to create new data visualizations to test in a VR fire dispatch setting that would help improve user performance. For doing so, our environment contains some consistent elements that persist regardless of what condition is run. These include a large main display, a hologram-like map that displays a section of New York City where the experiment will take place, an elevated platform for the user, and visual representations of water and fire levels. For water, there will be one bar per truck that shows, visually, its water percentage; for fire, the size of the fire will be scaled depending on the intensity of the fire. The experiments will be run at a future date after the end of our contribution to this project. Professor Orlosky’s team will run the experiment in order to make sure the testing conditions for our experiment remains consistent with their previous work.

This experiment will be testing four different conditions that vary the usage of a HUD and numerical representations of the fire and water levels. The HUD is a small sub-list consisting of the top four trucks. This provides a quick reference keep up to speed with the status of fires and minimizes the time spent looking up at the main screen. The number system provides the user with precise percentages of water available and the strength of a specific fire.

![Figure 3.7: Condition 1, without HUD and without numbers](image)

The first condition, as seen in Figure 3.7, was without the HUD and no numerical representations of the fire and water levels. We will use this condition to serve as a control and to see how efficient the user will be without the use of any further visualizations than the base ones.
Figure 3.8: Condition 2, with HUD and without numbers

The second condition, which is shown in Figure 3.8, was with the HUD and without the numerical representations of the fire and water levels. The user would only be able to see the size of the fire and water level status bar for the trucks that are on the HUD. This will show the efficacy of the HUD in increasing performance and lowering head movement.

Figure 3.9: Condition 3, with HUD and with numbers

Figure 3.9 shows the third condition. This condition was with the HUD and with numerical representations of the fire and water levels. This test is designed to show how well the user acts with access to a larger quantity of information and information sources.
The fourth condition, seen in Figure 3.10, was without the HUD and with numerical representations of the fire and water levels. Using only numbers, the user will be able to see the exact numerical amount of water in a truck and the level of fires. By using only the number system, we can determine the effectiveness of the user without the HUD.

The original room was based on a classic fire station dispatch center, which is designed around the idea of multiple people interacting to achieve a single goal. We felt that changing the design of the room to be centered around a single person would be a more efficient use of VR technology in this application.

### 3.7 Assets

Since the goal of our project was to test how new data visualizations in a VR fire dispatch setting could influence user performance, some of the assets in the existing project files had been modified. Some assets, such as the tables and monitors, were completely removed. Instead of three big screens on the back wall, we expand the middle screen and removed the side screens to make the view cleaner and less distracting. Instead of a 2D vertical map, we used a 3D hologram-like map representing an actual portion of New York City. Figures 3.11-3.15 show the before and after of these changes.
We chose to use a hologram-like map as the concept tends to appear multiple times in modern media. One example can be seen in Figure 3.16.
By using an actual location, we hoped to give the situation a more realistic feel. In order to let the user see the entirety of the map, a small platform was built and placed on one side of the room that placed the user at a position above the map. In addition, the map itself was put an angle to increase the user’s view of the map. In this way, the users are able to interact with the hologram effectively.
3.7.1 Initial Map

Due to the fact that the map for the original game was a terrain map, not a map for a specific city block, it was not intuitive to implement a terrain map with buildings on fire. Instead, the group decided to find a different map to build the hologram. A 2D map of an unknown city was downloaded and then imported into 3ds Max. It was used as the material to wrap on the plane beneath the hologram buildings. In order to make 3D models out of the map, cubes and other solid shapes were placed on the plane to mimic the city buildings. Because of the complexity of the 2D map, the solid shapes placed on it were not completely following the blocks on the map. Hence, the density of the buildings on the map was reduced and the height of the buildings were based on estimation instead of actual data. After finishing the 3D model for the hologram map, the models were exported together as a single object file from 3ds Max and then imported into Unity. The final result can be seen in Figure 3.17 and 3.18.

![Figure 3.17: Original Hologram Map top view](image1)

![Figure 3.18: Original Hologram Map side view](image2)

3.7.2 Shader Work

Since the group decided to use the Unity shader graph, it resulted in a switch in rendering pipelines from Unity default rendering pipeline. We chose the Lightweight Rendering Pipeline and upgraded the shaders from the previously existing game objects to adapt to the new pipeline. New shaders were created to make the hologram effect for our map. For the initial map, the hologram shaders are divided into two separate parts. The hologram base shader is responsible for the plane of the hologram, and the buildings shader is responsible for the solid shapes. Figure 3.19 shows the shader for the plane of the hologram while Figure 3.20 shows the shader for the buildings themselves.
A PBR\textsuperscript{1} graph was created for the shader. In order to distinguish the road and the areas for buildings, the map was edited in Photoshop to serve as the texture of the shader. Moreover, a light blue was selected to be the base color. Yet, the light blue color does not imitate the hologram texture on its own. A Fresnel effect\textsuperscript{2} and the base color worked together to form the emission for the shader so that it glows properly. Similar formation appears in the shader graph made for the hologram buildings. Motion was added to the building’s shader by offset and tiling so the texture will be able to scroll and look more like an actual hologram.

For the New York City map, our group abandoned the hologram base shader and added additional hologram shaders to imitate the fire effects. Since our group decided to have four fire levels, each fire level had its own shader. The shader graph for those fire shaders are similar, but the colors are different so that the user could easily tell the fire level by identifying the building’s color. Figure 3.21 shows one of the four fire shaders for the hologram when the building catch on fire.

\textsuperscript{1}Physically Based Rendering, a rendering procedures for computer graphics that will produce synthetic images that are visually and measurably indistinguishable from real-world images (Greenberg, 1997).

\textsuperscript{2}One important visual cue of objects in the real world has to do with how they become more reflective at grazing angles ( “The Fresnel Effect,” 2019).
This shader was aimed at representing actual flames on the building initially. However, the way the shader worked on the buildings to show the flames is awkward, as seen in Figure 3.22. Due to the fact that our group was not able to unwrap the UV from the buildings since they are generated on play, we abandoned the idea and chose to adjust the vorono$^3$ node to simulate the hologram fire shader.

3.7.3 New York City Map

Partially through our project, we decided to use a map of an actual city. To achieve this we looked into the Google Maps Unity SDK. This SDK gave us access to a real world scale map of the New York City area. We later had to scale this down and trim down the exact area we used because the scale of the map was so large compared to the

$^3$An algorithm to partitioning a plane into regions (Klein, 2005).
scale of the room. Because we only had access to the demo version, we could only use New York City, but that proved to be enough for our implementation. Once the API was brought into Unity, we ran a script that was provided and that would generate the map on play. The exact inner workings of the API are unknown to us due to the closed source nature of the API, but we were able to modify some parameters—within some arbitrary limitations—including coordinates and scale to get the exact map we did. The final result of the map can be seen in Figure 3.23 below.

Figure 3.23: New York City Hologram Map in environment

3.7.4 Building the navigation Mesh

In order to represent a fire truck moving around from fire to fire, a NavMesh was required. However, due to the method the Google Maps platform used to generate the map, some extra steps were required to properly bake out the NavMesh. To do so, we needed a 3D model of the part of the map we were using. By applying a gray material to the map and switching to isometric view within Unity, we obtained a top down view of the map. To bake out the NavMesh, a grey material was applied on the hologram. The perspective was turned off, and a screen shot from a top down view of the map was taken. After that, the screen shot was implemented in 3ds Max, and a model to simulate the New York city blocks was made based on it. As the models were finished, they were exported as an OBJ\(^4\) file and imported into Unity, as seen in Figure 3.24.

Figure 3.24: New York City Navigation Model in Unity

The first step to making the NavMesh was to generate a rough model of the map that the trucks would later use. Because the road on the hologram was so narrow, our

\(^4\)A file type used to store the geometry of a 3D object (Bourke, n.d.).
NavMesh needed to be as precise as possible to give the trucks more flexibility and space to move in. Therefore, generation settings were carefully selected to maximize the drivable space. The NavMesh was generated using the built in NavMesh tools and generated rather than dynamically generated due to complications with the Google Maps Platform SDK (Chapter 4.2, Google Maps Platform’s Maps).

After the NavMesh was built, 35 destinations were set based on the buildings that could be lit on fire. A pathfinding script was attached on each firetruck in the project, and once the fire destination was assigned to a truck, the truck will move towards the destination based on the road on the NavMesh.

Unfortunately, our time constraints did not allow our group to have a fully implemented navigation functionality. The truck could pathfind on the NavMesh successfully. However, our group are not able to link the pathfinding functionality with the firetruck deploy functionality. Hence, we were only able to allow fire trucks to pathfind to the preset location.

### 3.8 Coding

For the code design, there are three main components we modified in order to implement our goals. They are firetrucks and their information, fires and their information, and VR actions/events.

#### 3.8.1 Fire Trucks

As mentioned earlier, we have a large main display that contains the 12 trucks. A script called `FiretruckDispatcher` was in charge of the locations for the trucks on the back screen; these locations were passed into one of the highest level scripts, `GameManager`, which is in charge of core tasks, like creating the new trucks for the user. `FiretruckDispatcher` also has the key role of interpreting the user’s input so that the user can scroll across the back screen with the D-Pad on the Vive Controller or deploy a truck with the trigger. The trucks themselves contain multiple scripts to be able to control their own water levels and their current location along with other managerial tasks; these scripts include `TruckManager` and `PCStateManager`.

`FiretruckDispatcher` enabled the user to use the trackpad on the controller to scroll and select the truck on the screen. In order to Light the buildings on fire, we needed a valid target for the raycast to hit. We could not use the buildings themselves as they only existed on play, to get around that we created a separate hitbox that the raycast could target and we could attach scripts to. These hitboxes can be seen in Figure 3.25.
While working with raycasts, we encountered a problem while trying to find what objects the ray was colliding with. Our possible choices were using a tag or placing all objects that could be hit to their own layer in order to properly detect the collisions. The original code used a mixture of existing GameObjects which could be filtered based off of their tags. This was difficult to work with so we replaced the tag system with a layerMask that limits the raycast to one render layer. On that layer, we placed only the valid raycast targets to remove the tag based filtering. Once the user pulls the trigger, the code checks to see if the user is aiming at a valid building hitbox and has a valid truck selected, if so, a truck is deployed to that building. A building is considered a valid target if it both does not have a truck assigned already and is actively on fire. A truck is considered valid if it is not currently deployed to a building.

The FiretruckManager script took care of managing truck water levels while fighting a fire, including things like our efficiency bonus when a truck with plenty of water was assigned to a small fire. The PCStateManager script accessed the FireTruckManager to update the back screen icon, with a real time water level indicator.

The HUD element moved with the headset, which means that the user can always see the HUD no matter where they look.

On the HUD, we created a list that stores available trucks on it. Since the HUD element was also a recommendation system that we had in order to help the user perform better, we only listed out the four trucks with the highest amount of water among all the available trucks. The information display would be different depending on the mode the user entered. For example, if the user is in the HUD on and Numerical off mode, they will only see the waterLevel status bar on the HUD without showing the exact percentage on it.
3.8.2 Fire Buildings

Setting buildings on fire—in other words, turning the hologram texture to red—used a similar method we used to turn the New York City map from the Google API into a hologram. Before writing the code that would set the building on fire, we manually selected buildings that we wanted to be possible targets. At first, a total of 42 buildings were selected. However, due to the density of buildings, it became hard to tell which building was on fire. As a result, we removed 7 buildings to have a total of 35 buildings that could be on fire. The resulting map, with all possible buildings on fire, can be seen in Figure 3.27 below.
In order to select from these buildings, a function was made to retrieve any relevant game objects in the scene. The function takes in an integer representing a position in the building list and returns a list containing any game objects related to the building. This information was passed into the `GameManager` to link with the existing code for city blocks, which used to hold `BlockStateManager`. Since we could not access the map before entering play mode, we could not directly raycast to the building. The hitboxes were created as the raycast targets and also linked with the city blocks to connect each hitbox with its building. Now, each hitbox held a `BlockStateManager` that stored the information of engine present, a `fireLevel`, and a reference for the truck if there is a truck assigned to the building.

The pop up menu contains a truck’s status bar displaying the truck’s water amount, a building’s status bar with its fire level, and an image of a fire truck. If the building was assigned a truck, the truck’s image would be rendered, and the `waterLevel` status bar would be populated with the fire truck’s water level. While a truck is present, the `fireLevel` status bar decreases as the truck fights the fire. If there is no truck at the building, the pop up menu only shows the `fireLevel` status bar. Figure 3.28 below shows what a pop up menu for a building without a truck currently present would look like while Figure 3.29 shows a pop up menu for a building with a fire truck present.

![Figure 3.28: Pop up menu for a building without a fire truck present](image)

![Figure 3.29: Pop up menu for a building with a fire truck present](image)

The numerical percentage on the pop up menu for the `fireLevel` and `waterLevel` only appear if the current testing condition has the numerical system on. For example, if the user is in the HUD on and numerical off testing condition, the user would not be able to see the percentage on the pop up menu.
3.8.3 VR Actions/Events

In *FireTruckDispatcher*, we modified the interaction so that users can utilize both controllers instead of one controller in the old test, which allowed users to interact more with the environment. The input system for the original code checked if a particular button was pressed and then executed the corresponding action. This type of input system is hard to manage and scale, so we opted to use the SteamVR event-based action system, which follows the Event Driven Architecture control paradigm, a more scalable input architecture (Hophe, 2006).

There were three main actions that we added to the function: scrolling and selecting trucks, deploying the truck, and clicking building to show the building information. The experience with testing on the old task, which grab and drag the truck to where it was supposed to be was hard, gave us the idea to modify the actions to event-based action system. For the scrolling action, we bound the trackpad to be a D-pad and set four end coordinates to be scroll up, scroll down, scroll left, and scroll right which allows the user to select the truck they wanted to deploy. A highlight effect was added to the spot that the user stopped at which would become the selected truck on the list. The user can deploy the truck by aiming at the target and click on the trigger. We also created a button on D-pad on the right controller that users can raycast to aim the pop up menu, and had information for the building for three seconds.

3.9 Final Environment

After completing all of the aforementioned visual features and compiling them, we were able to complete a working VR environment, which can be seen in Figure 3.35. The final environment had some key features: the large back screen, the HUD, the hologram map with different colors to differentiate fire levels, and the pop up menus. Also, the colors of the room were darkened to reduce strain on the user’s eyes and help them focus on their task.
Figure 3.30: Final HUD display

Figure 3.31: Final Hologram Map with Different Fire Levels

Figure 3.32: Final Pop up menu with Fire Truck Present
Alongside our final environment, we created documentation to give future groups working on this project a quick guide to understand how the code and testing environment works (Appendix B). The documentation comes in the form of insights into the code we developed, thoroughly commented code, and details on how to edit the hologram map. Using this alongside our report, our hope is that future teams will have an easier time picking up on this project where we left off.

4 Problems

Though our group had a relatively fluent production process, we still had to overcome a number of obstacles. These problems originated from 4 main areas: lack of technology knowledge, the Google Maps Platform, the input system, and the controller.
4.1 Technology Learning Curve

Even before we began work on our project, we encountered a problem in the form of a lack of knowledge of Unity. While we did have some training before starting work, it was not anywhere near enough to fully learn Unity. As a result, our group ran into problems with various features in Unity. For instance, distinguishing between sprite and image renderer components, switching to the Light Weight Rendering Pipeline, and SteamVR action set inputs.

In Unity, newly created sprites automatically have a sprite renderer attached. This caused us problems since the sprite renderer acts different from other renderers such as mesh renderer, thus sprites are rendered in front of everything. To fix this problem, we switched from using a sprite to using an image to display the information properly.

Meanwhile, the Lightweight Rendering Pipeline turned our project materials into a light pink color because the previous project was using a Standard or Unlit Material which would not be rendered in the Lightweight Rendering Pipeline. Therefore, all the previous materials appeared using the default pink unlit shader which is a Unity indication of when a shader is broken. We took some time to generate all our materials to adapt the Lightweight Pipeline.

Moreover, although we have some basic knowledge about SteamVR, we did not know that we had to reimplement Steam VR’s package every time we opened a new project or generate new VR action sets for the controllers every time. Hence, at the beginning, it took time to figure out why the project was not running when we transferred our project to the VR testing computer.

4.2 Google Maps Platform’s Unity SDK

The biggest set of problems came from the Google Maps Platform’s Maps Unity SDK. Due to the fact that Google Maps Platform is a professional tool for developers to create immersive locative experience, the full version of the SDK comes with a price tag of $200 a month. As a result of the price, our group decided to use a demo version of Maps Unity SDK which did not have any cost.

Originally, our team decided to directly interact with the model of the map. For example, our team wanted to bake out the NavMesh directly from the hologram model and also attach the script directly to the building. However, due to our limited resources and the limited access to the Maps Unity SDK which only generates the buildings at run time, our team was not able to stick with our original plan, as it was hard to make changes since they were not saved while in play mode. Therefore, hitboxes were created for players to interact with the buildings, and new models were made based on the NewYork Map to generate out NavMesh.

Another issue with Google Maps Platform’s Maps was that all the buildings might have two or more GameObjects, each of which held different components of the building, such as the surface to build the building, the position of the building, or the predefined texture for the building. However, the building would only render during play mode. Furthermore, the code for the map was hidden behind DLL files and a paywall, so
that we could not get access to it. We could not actually attach the script to the
building directly. Instead, we created a script that, when attached to the parent object
of all the map buildings, would loop through every building and change its material to
our hologram texture. This script would run right after all the buildings were created,
allowing it to successfully affect every child object, including the ones created on play.

Due to the high density of New York City, it was hard for users to distinguish all of
the buildings from one another and click on the appropriate one. Since our project was
focused more on the data visualization, we did not want the density be the factor that
affected our data. So, we chose a small section of the entire New York City map and
rescaled it to fit inside our room.

Additionally, the Map’s Unity SDK also caused problems after applying the edited fire
shader with the alpha channel (the degree of opacity of an image) in use. The pre-built
models were blinking at a high speed after the fire shader was applied. Since we were
not able to access the source code, our group could not fix the problem as we could not
pinpoint the source of the issue.

4.3 Controllers

The Vive Pro controllers that we were using gave us a variety of problems mainly due
to our switch in the rendering pipeline. We had multiple issues with the controllers:
not having an in-engine 3D model, not having a material, and tracking not functioning
correctly or at all. Instead the controllers would be invisible and be stuck in one place
on the floor. In order to fix this issue, we had to make some edits to the script of the
controller model which was found under the cameraRig object. In order to give the
controllers a model, we changed each controller’s index, SteamVR’s unique identifier for
each of its devices, to the required index. In our case, the correct index was index 4.
While this method did work, the controllers still did not have textures and appeared as a
bright pink. By changing the shader under the same model object for each controller to
a Lightweight Pipeline compliant shader, we were able to give the controllers a texture.
To fix the controller tracking issue, we had to configure SteamVR and reinstall the
SteamVR Unity plugin. This combination of steps allowed for our controllers to be
successfully tracked and made visible in VR.

5 Future Work

When starting on this project, we had a wide variety of ideas we wanted to implement.
However, since we had a limited amount of time, there were some features that we were
not able to fully complete. However, we do hope that future groups who work on this
project will be able to possibly implement some of our ideas.

A DLL is a library that contains code and data that can be used by more than one program at the
same time (‘What is a DLL?’, 2019).
5.1 Dynamic Map

Since we were not able to access the full capabilities of the Google Maps platform, we were not able to implement a dynamic map in the project. If we had access to the full platform, we were planning to add multiple locations for the user to use. Furthermore, with full access to the platform’s SDK, the hitboxes we created would no longer be required. Instead, we would be able to access individual buildings and modify them to possibly detect their own raycast collision. By implementing different locations, we would be able to understand whether the hologram design will be feasible for multiple location types, for example, a city versus a rural area. Some comments have been made about the feasibility of our implementation in large rural areas where building fires are less common than wildfires, or large dense areas where some buildings may be blocked by others. A fully featured implementation could test these conditions.

5.2 Add AR Implementation

The prior research for this project included Augmented Reality, a possible design vector we were considering at the beginning of the project. This consideration led us to develop translucent textures for the buildings in order to mimic the style of map that could be generated in an AR application. However, none of us had prior experience with AR, and there was not enough time for us to research and develop both an AR and VR version of our project. If we had more time, we would like to turn the city map into an AR hologram, integrate more of the user’s environment, and test how that affects the user’s performance.

5.3 Explore Variable Ways to Display Information

Despite the fact that our team tried our best to display the information so that it is conspicuous for the users, there are definitely more ways we are willing to test in the future. For instance, our team is willing to explore if adding a fire-like particle effect is effective or changing the display of the pop-up menu is easier for users to get the information. We would like to test various ways to display information, which will illustrate which data visualization will be more efficient.

5.4 Different Room Settings

As the same idea for display information with different visualizations, we want to explore if different room settings would vary users performance. One of the ideas that we came up with at the beginning of the project was to design a spherical room, which had all the information displayed around the user and used the virtual reality environment to the greatest extent.
However, we did not have enough time to create the curved display screen to display the information. In addition, there were concerns that the curved map might become distorted. Meanwhile, since our research suggests that turning too often will cause our participants motion sickness, we did not want to risk having a cylindrical room which displays all the information around the users. However, our group still thinks that it is an idea worth exploring since a spherical room may have the ability to exploit the virtual reality features fully.

5.5 Making an Executable

Currently, the project is only runnable from within the Unity Editor and not as a standalone executable. We did not create an executable file because we did not know if there would be any problems associated with files becoming unlinked, especially something like the resulting data output files, which were currently hard coded with an absolute path. We would need to find where the files are referenced and change that portion of the code base, something we determined to be out of scope. We were able to export the "roll a ball" VR version, but it only had regular textures with it, and it was a small project, with few moving parts and no imported assets, that gave us significantly fewer potential points of failure. The potential time sink for maintenance after an export of this project with few benefits meant we did not find it to be a worthwhile investment of time during our limited stint of work. If the file were to be exported as a standalone executable, we would be able to move forward.

6 Conclusion

Our project worked to create new data visualizations to test in a VR fire station dispatch environment that would help improve user performance. To accomplish our goal, we modified the setting of the fire station dispatch center and explored new ways to interact with the environment. We divided out group into an asset team and a code team. Focused on the development of the virtual reality environment in Unity, the team deliberately designed the layout of the room to make it intuitive for the users and give it easy to trigger functionalities. Our project has been completed in a manner that future groups will have a solid foundation on which to continue our work. We have
created documentation that outlined how to run the experiment and how the scripts work in the form of well-commented code. We hope our project can help the researchers who are working on data visualization testing, and we hope our list of future works can contribute to other research areas.
References

[18] Liu, C., Kalkofen, D., Kiyokawa, K. & Orlosky, J. (n.d.).Visualization-Guided At-
tention Direction in Dynamic Control Tasks. [PDF File].
Appendix A - Smell Drone

Our secondary project was a drone-related project named Project: Smell Drone. This project was centered around the idea of increasing the level of immersion a VR user would be able to experience with the use of drones and scents. Our goal was not to complete the entire project but rather to work on a portion of it. We worked with Photchara Ratsamee, Parinya Punpongsanon, and Haruka Matsukure, all professors at Osaka University. The requirements we were given were as follows: make the drone move from one position to another in a 3D space, use PID smoothing and Unity to allow the drone to move, return the flight path as a list of point nodes, and rework the existing drone attachment that is the origin of the smells we are using.

Project: Smell Drone was therefore split into 3 main components: the VR environment, the attachment for the drone, and the drone code. The environment was a garden in VR space. The lavenders would allow participants to approach the flower in VR space, while the drone attachment should release the smell of lavender by flying over the user while holding a lavender oil-soaked cotton ball. All of this comes together via the drone control code that uses HTC Vive Head Mounted Display (HMD) tracking, Optitrack point tracking for the drone, and a User Datagram Protocol (UDP) control Software Development Kit (SDK) developed by Ryze Tech for direct drone control without a person doing the flying.

A.1 The Environment

The environment of the project is a garden which contains lavender flowers. Since the Smell Drone project is aimed at testing how the smell works with virtual reality, the environment is supposed to be made authentic. Thus, realistic lavender flowers were needed so participants could connect the visual of seeing the flowers to the smell of the flowers. Figures A.1 and A.2 show the lavender flower and an overview of the garden we created in Unity.
To build the environment, a skybox was downloaded and imported into Unity to mimic the sky, and a green terrain was built to plant the flowers. Due to the fact that the time for this project was very limited, our group chose not to make our own flower model. A 3D model of lavender flowers was found free online. Modifications were made so that the flowers could fit the garden we have. Moreover, after the lavender had been implemented in the environment, stones were downloaded from the assets store and placed to surround the flowers. Our group also created walls in order to show the border of the garden.

**A.2 The Attachment**

The main software used to design the drone attachment was SolidWorks. Our initial idea was to allow as much scent to be released from the cotton ball as possible. As a result, a very open cage was built that was supposed to use a toothpick to hold the cotton ball in place.

As seen in Figure A.3, the toothpick would be mainly held in place by friction and small rubber bands that would be attached after placing the cotton ball in place. However, after the first test print, we realised the holes for the toothpicks compromised the structural integrity of the attachment, causing it to snap near the holes. As a result of that failure and further discussion with Moji-Sensei, we changed our design to direct the scent rather than have an open design.
As seen in figures A.4 and A.5, we changed our design to only have one side of the attachment open in an effort to allow the scent to be directed. This second model did print successfully and was tested on the drone without any landing gears, where it was able to work without problems.

A.3 The Code

The code base for Smell Drone consists of two main sections, the VR Environment, and the Drone Control. The Environment code would be in charge of interpreting user input, and the drone control would take care of communication to the drone. Finally, the interface would be in charge of simplifying the command structure for the drone and abstracting out the UDP nuance, such as establishing a connection, or opening listening channels, in order to make future development simpler.

A.3.1 Environment Code

While being a simple set of code, the ability to track the position of the drone in the VR environment was useful for controlling the drone further down the line. To
that end we used Optitrack’s proprietary tracking software: Motive. That allowed the 5-point Optitrack setup tracking the drone position to relay the position and rotation information back to Unity and into a GameObject of our choosing. Once in Unity we were able to treat the object like any other Unity object.

### A3.2 Drone Control

With the information in Unity, we needed a way to control the drone from the computer. The first approach we took was to use the TelloPy API for Python3. This quickly proved to not be a worthwhile approach as Unity does not natively support Python development. After some research we discovered that the Tello has a UDP-based SDK for direct drone communication. With this we were able to set up a UDP connection within C#, allowing us to send commands and receive a response from within the Unity environment.

The communications system we developed uses a singleton class to establish communication and a single function to pass arbitrary data to the drone while reporting back the response. If control commands are passed into the communications framework, the position, rotation, and speed of the drone can be controlled from anywhere within the project.

### A.4 Problems and Discussions

#### A.4.1 Attachment

Since the surface that we can work around for the drone was small, the size of the attachment that we could create was constrained. In the end, the attachment was a 30mm * 17.5mm * 8.2 mm box. Although not our first choice, certain designs we would have considered superior, were not possible to print; for example, any design that was thinner than 0.75 mm was not printable due to resolution limitation on the 3D printer.

#### A.4.2 Technological Setbacks

Much like any project, we did run into some issues during development. One such issue is that none of us had worked with setting the environment up for a drone before. Our first day on this project was spent setting up the OptiTrack cameras and figuring out the positions for the markers that track the drone in 3D space. Moreover, we did not know about drone propeller orientation, so we spent a large portion of time trying to debug what we thought was a communication issue, when in fact all we needed to do was swap propeller positions. In addition, most of the documentation we were able to find for the more technical aspects of the Tello drone, like firmware updates, was in Japanese, which made reading it more difficult.

We ran into hardware issues with the computer setup we were using when trying to run all of the USB devices we needed. Due to a limited number of USB Identifiers that the USB controller built into the motherboard, we were not capable of running Motive with all of its OptiTrack cameras alongside Unity and the HTC tracking cameras.
Finally we ran into issues translating between the three different coordinate systems that Unity, Motive, and the Tello drone have. Unity uses an absolute coordinate system that is arbitrarily centered, where Motive’s coordinate system, while still an absolute system, is based around the visible region that the cameras have. Finally the Tello drone uses a coordinate system relative to itself as (0,0) with forward/backwards, left/right, and up/down rather than (x,y,z). Luckily for us the Motive Unity plugin took care of translating from Motive’s coordinate system to Unity’s, which left only Unity’s and Tello’s coordinate systems to be merged. For that a scale factor of 1:100 Unity to Tello units was used, with some 3D vector translation to convert an absolute direction vector into the relative axis used by the Tello drone. With this math in place we were able to communicate to either system in Unity units, once again simplifying development.
Appendix B - Documentation

User Manual

About This Document

This project is designed for testing new data visualizations in a VR fire dispatch setting that would help improve user performance. Since it is based on a developed project, our group made modifications to fulfill the requirements we were given. This document contains information on how the project works at the time of writing. This document is meant to be used as a guide along with any other documentation to help a new user learn the code base.

How to Run the Project

1. Create a new Project
2. Import the packaged project
3. Download SteamVR package from the assets store and then import into the package
4. Go to Window → SteamVR input → Save and generate
5. Click on Open Binding UI to customize the controller
6. Click play to enter demo level for the experiment
7. Press spacebar to go into the first condition
8. Press spacebar to enter next condition until all conditions have been completed
9. Click the play button to end the experiment

Further development

Map Implementation

NavMesh Implementation

In the project, the NavMesh was not generated from the Google Maps Platform SDK. It is generated by another model we made based on the New York City map. This is because the map only generate itself in play mode. If new 3D maps are going to be implemented, new NavMesh needs to be generated based on the new map. This could be done by open up the navigation window in Unity by clicking Window, AI and then Navigation. New destinations need to be created as well.
Material Implementation

Our group changed the rendering pipeline of the project from default to the Lightweight Pipeline. The rendering pipeline can be set up by click on Window → Package Manager → Lightweight RP → Install. After that, create a Lightweight Render Pipeline Asset by clicking create → rendering → Lightweight Render Pipeline → Pipeline Asset. To successfully change the rendering pipeline, click on Edit → Project → Settings Graphics and then drag Lightweight Render Pipeline Asset to Scriptable Rendering Pipeline Settings. Moreover, since we created special hologram shaders for the roads on the map and the buildings on the map, the materials which contain the shaders need to be dragged on to the new map to create the holographic look. The material name is Hologram and Hologram_Road. In addition to that, if the hologram shaders need to be changed, clicked on the shader graph of the hologram shader and then edit the nodes to adjust it.

Code Development

BlockStateManager

• Description
  – A script that runs each city block to keep track of a building’s own state

• Key Variables
  – int fireLevel
    * The current fire level this building has
  – bool enginePresent
    * Is there a truck at this building?
  – List<GameObject> buildingList
    * The rendered buildings this hitbox belong to
    * In the NYC map, a single building can be represented by more than one GameObject
  – GameObject popupMenu
    * The popup menu that belongs to this hitbox
    * Must have a popupManager component

• Key Methods
  – void Update()
    * Controls popup menu visibility
    * Controls popup menu component visibility
  – void changeAllMat(Material[] mats)
    * Sets the material of every building in building list
**dispatchExperimentManager**

- **Description**
  - Core script for running the experiment

- **Key Variables**
  - `string[] conditions`
    - A string array containing the names of all the experiment conditions, including the demo level
  - `int condition`
    - An integer representing the current index in conditions of the current condition?

- **Key Methods**
  - `void Start()`;
    - Initializes the conditions, and randomizes the order
  - `void Update()`;
    - Continually logs participant’s data
    - Changes the condition when spacebar is pressed
  - `void setConditions(int condition)`;
    - Changes the internal parameters to modify the way external scripts react, such as a publically accessible variable HUDOn, which tells HUD script whether or not to render the HUD
  - `void writeDataToFile()`;
    - Writes given data to a hard coded file

**fireTruckDispatcher**

- **Description**
  - Manages user action events

- **Key Variables**
  - `SteamVR_Action[...]`
    - All of the different required variables for event binding

- **Key Methods**
  - `void Scroll[...] (SteamVR_Action_Boolean fromAction, SteamVR/Input_Sources fromSource)`;
    - Scrolls the selected truck in the indicated direction
- **void placeTruck(SteamVR_Action_Boolean fromAction, SteamVR_Input_Sources fromSource)**
  * Attempts to place a truck where the user is pointing
    - Requires the user to be pointing at a building hitbox that is on fire and with no truck

- **void displayPopupMenu(SteamVR_Action_Boolean fromAction, SteamVR_Input_Sources fromSource)**
  * Toggles the popup menu for the hitbox the user is pointing at

**fireTruckManager**

- **Description**
  - Manages the fire fighting capabilities of each individual truck

- **Key Variables**
  - **Transform currentBlock**
    * The `Transform` of the `GameObject` this truck is currently assigned to
  - **int waterlevel**
    * The amount of water this truck has
  - **string state**
    * The current availability of this truck

- **Key Methods**
  - **void timeHandler()**
    * Controls fighting the fire, including decreasing `fireLevel` and `waterLevel`
    * Resets the truck position once its done fighting the fire
    * Controls water refilling, if the truck is not deployed

**gameManager**

- **Description**
  - Controls the project

- **Key Variables**
  - **GameObject Monitor[...]**
    * The position for new monitors/trucks
  - **GameObject[] blockList**
    * Holds all the city blocks
- **GameObject[] allMonitors**
  * An array of all the currently active trucks

- **Key Methods**
  - **void Init()**
    * Resets the game state before updating to the new experimental condition
    * Controls things such a HUD visibility and number visibility
    * Clears all fires and trucks
    * Resets the original truck
  - **void demoLevel()**
    * A brief demo as an instruction to the project
    * Trucks get created
    * A few buildings are lit on fire
  - **void regularLevel()**
    * Runs the core game functions including:
      - Fire generation timing
      - Truck creation timing
      - Keeping the HUD updated

**HUDManager**

- **Description**
  * A parent script to popupManager references, passes down calls like setVisibility

- **Key Variables**
  - **GameObject HUDTruckLot**
    * The parent object of all the individual HUDTrucks

- **Key Methods**
  - N/A

**NYCMapHologramify**

- **Description**
  * Set the hologram texture for the NYC map GameObjects
  * This is necessary, because the GameObjects are created at play time

- **Key Variables**
  - **Material mat1**

46
* The hologram material to use for the buildings
  
  - **Material mat2**
  
  * The hologram material to use for the roads

- **Key Methods**
  
  - **IEnumerator LateStart(float waitTime)**
    
    * Waits for the `GameObject` to be generated by the google API, then sets all of the `GameObjects`'s materials depending on `mat1` or `mat2`.

**PCStateManager**

- **Description**
  
  - Controls the information displayed on the back screen icons

- **Key Variables**
  
  - `GameObject truck`
    
    * The truck this script belongs to

- **Key Methods**
  
  - **void Update()**
    
    * Keeps the back screen icon up to date

**popupManager**

- **Description**
  
  - Managers the individual popup menus

- **Key Variables**
  
  - `GameObject fireBar`
    
    * The fire display bar

  - `GameObject firePercent`
    
    * The text to set the fire number

  - `GameObject waterBar`
    
    * The water display bar

  - `GameObject waterPercent`
    
    * The text to set the water number

- **Key Methods**
  
  - **void set[Fire/Water]Level(int percent)**
* Sets the [...] level percent text and bar fill according to the input given

- **void set[Icon/Percent]Visibility(bool isVisible)**
  * Sets the visibility of the [...] according to the input given

**WeStartedTheFire**

- **Description**
  - Interfaces between the *GameObjects* created by the Google API that can be lit on fire

- **Key Variables**
  - **string[] buildingList**
    * a list of partial strings that uniquely identify the building that we wanted to light on fire

- **Key Methods**
  - **List<GameObject> getBuilding(string building)**
    * Returns a list of all the *GameObjects* created by the Google API that contain the given String
    * Meant to be used with a string from *buildingList*
  
  - **GameObject getBuildingHitbox(string building)**
    * returns the Hitbox that belongs to the building that starts with the given String
    * Meant to be used with a string from *buildingList*