Simulation Of Virtual Reality Display Characteristics: A Method For The Evaluation Of Motion Perception

Jonathan Emmanuel Hopper

SIMULATION OF VIRTUAL REALITY DISPLAY CHARACTERISTICS: A METHOD FOR THE EVALUATION OF MOTION PERCEPTION

A Thesis
presented in partial fulfillment of requirements
for the degree of Master of Science
in the Department of Computer and Information Science
The University of Mississippi

by

Jonathan Hopper

May 2020

ABSTRACT

Visual perception in virtual reality devices is a widely researched topic. Many newer experiments compare their results to those of older studies that may have used equipment which is now outdated, which can cause perceptual differences. These differences in hardware can be simulated to a degree in software, provided the capabilities of the current hardware meet or exceed those of the older hardware. I present the HMD Simulation Framework, a software package for the Unity3D engine that allows for quick modification of many commonly researched HMD characteristics through the Inspector GUI built into Unity. I also describe a human subjects experiment aimed at identifying perceptual equivalence classes between different sets of headset characteristics. Unfortunately, due to the COVID-19 pandemic, all human subjects research was suspended for safety reasons, and I was unable to collect any data.

ACKNOWLEDGEMENTS

I would like to thank Dr J. Adam Jones, my advisor, without whom I would not have found my interest in the field of virtual reality in the first place. This work would not have been possible without him or without the resources and members of the High Fidelity Virtual Environments (HI5) Lab. I would specifically like to thank Ethan Luckett and Hunter Finney for their help with troubleshooting and their willingness to test my framework in its earliest stages.

I am also grateful for the resources provided by the Department of Computer and Information Science, as well as for the members of my defense committee, Dr. Dawn Wilkins and Dr. Philip Rhodes.

I am thankful for the support of my family and friends, particularly my wife Elizabeth, without whose patience and support I could not have completed this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Throughout much of the history of virtual reality, head mounted displays (HMD) were prohibitively expensive items, relegated to research laboratories and universities. However, over the past decade, ever cheaper and higher fidelity headsets have become available to consumers and businesses, prompting a huge surge in virtual reality (VR) interest in fields such as architecture [7], medicine [23], modeling and manufacturing [30], and gaming. As interest grows, so do expectations from consumers about the quality of the experience given by a headset. While manufacturers continue to produce headsets with lower weight and latency, higher resolution and frame rate, and better motion tracking, a truly immersive VR experience requires replicating a user's real-world perceptual experiences in the virtual world, a feat that cannot necessarily be achieved solely through higher specs. Rather, headset designers must take into account the perception of the user in the real world in order to accurately mirror that perception in VR.

This has prompted a wide array of research into human perception in virtual reality. Virtual environments allow researchers to design perception experiments that would be time consuming, expensive, or even impossible to execute in the real world. Additionally, by observing the relationship between how users experience the real and virtual worlds, we can create new headsets and virtual environments that more closely mimic the real world, without necessarily even changing the hardware. This forms a feedback loop of sorts between our understanding of human perception and the quality of virtual headsets and environments.

Perception studies using virtual reality are not new, with experiments using the technology going back several decades. However, as headsets become better and less expensive, more research is starting to leverage their benefits. Newer studies often model their experiments on and compare their results to older research. The potential effect of improvements to HMD characteristics like resolution, field of view, and latency on a subject's perception calls into question the comparability of results across time. Even studies conducted using same-generation headsets may result in significant perceptual differences, as different manufacturers emphasize different characteristics.

For example, many older research-grade headsets such as the NVIS NVisor SX and ST60 featured 60° diagonal fields of view with a 6:5 aspect ratio. More modern HMDs like the HTC Vive and Oculus Quest offer 90° to 110° fields of view with a 9:10 aspect ratio, which may lead to discrepancies between users' perception in newer versus older headsets. While studies can be conducted to directly measure this difference, they would require the researchers to physically possess one of each headset, which raises a number of issues. Most prominently, many older headsets are still prohibitively highly priced, with the NVIS NVisor selling for nearly $35,000. Compared to even the more expensive modern systems like the Vive Pro, which sells for $1,100, an NVisor may not be a practical purchase, though the NVisor does have a higher pixel density than most modern headsets. Additionally, many older headsets are no longer in production owing to newer, cheaper displays, making acquiring them even more difficult.

The framework presented in this work aims to provide a method by which studies using different HMDs can be compared despite characteristic differences between headsets. The framework, dubbed the HMD Simulation Framework (HSF), leverages the fact that many HMD characteristics can be altered in software. For example, the Vive Pro, which features a resolution of 1440x1600 per eye and a field of view of 110° is perfectly capable of presenting an image subtending 90° at only 720p. While some characteristics like weight and balance are completely hardware-bound, many visual characteristics present in older headsets can be simulated on newer headsets. Thus, it may be possible to conduct an experiment more comparable to older studies' results simply by replicating as many of the parameters of the old headset as possible.

Same-generation HMDs can also be compared more easily under the HSF. By altering multiple parameters at once, it may be possible to find perceptual equivalence classes, that is, unique sets of HMD characteristics that result in the same or very similar visual perception. This may allow headsets with, for example, lower fields of view, to compensate perceptually by altering a different characteristic, such as resolution. This work also presents a user study aimed at identifying these equivalence classes for three HMD characteristics: field of view, resolution, and geometric distortion.

CHAPTER 2

BACKGROUND

"The ultimate display would, of course, be a room within which the computer can control the existence of matter. A chair displayed in such a room would be good enough to sit in. Handcuffs displayed in such a room would be confining, and a bullet displayed in such a room would be fatal. With appropriate programming such a display could literally be the Wonderland into which Alice walked."

- Ivan Sutherland, The Ultimate Display [24]

2.1 History of HMDs

In 1968, three years after the above quote was published, Ivan Sutherland and his students created what is widely considered the first virtual reality head mounted display, The Sword of Damocles. The display was much too heavy to be worn on the head in its entirety and was suspended from the ceiling, hence the name. The user's head would be strapped into the device, allowing them to see 3D virtual objects, generated by a computer, in the room around them. The objects were simple, wireframe constructs but were displayed relative to the user's head position, which was tracked mechanically using the arms that connected the display to the main body of the device. While never developed beyond his lab, Sutherland's display was seminal to the field of computer graphics and user interfaces.

Nearly 20 years later, Jaron Lanier and Thomas Zimmerman founded VPL Research, Inc, one of the first companies to commercialize the concept of a virtual reality headset with their EyePhone 1. The HMD itself cost \$9,400, and later models, complete with a computer on which to run the headset, could cost upwards of \$100,000. The headsets did track head movements, though the displays were limited to 5-6 frames per second. The headset was bulky, low frame rate, and prohibitively expensive, but it marked the beginning of commercial availability for VR [2, 6].

On March 29, 2013, the first Oculus Rift (now referred to as the Development Kit 1, or DK1), was released. Featuring 24-bit color depth, lower pixel switching time than competitors, and, most prominently, a 110° diagonal field of view, the DK1 was a huge success both commercially for Oculus and for the continuing acceptance of virtual reality outside of university laboratories. In addition to its many features, the display was also cheap relative to its capabilities and to other similar designs, costing only $300 at launch. The success of the DK1 served to open the floodgates as more and more companies started gaining interest in virtual reality development. It also led to Oculus's acquisition by Facebook in 2014 for $2 billion [1], making it the first major financially successful VR company.

Three years and an additional development kit later, Oculus released the final consumer version of the Rift, which competed with the HTC Vive and, slightly later, Sony's PlayStation VR. Among smaller details like lighter weight and vastly different controller designs, one major difference between it and the Vive was the tracking system. Dubbed the Constellation Tracking System, the Rift headset featured an array of embedded infrared LEDs, which are monitored by two (or three for larger tracking spaces) infrared cameras mounted to thin, short poles. By keeping track of the LEDs from multiple angles, the headset's position and rotation values can be extrapolated. This style of tracking is called outside-in tracking [22]. Supplementing this with a dead reckoning system called the Adjacency Reality Tracker, which includes a gyroscope, magnetometer, and accelerometer, allows the Rift to track the HMD accurately both positionally and rotationally.

The Vive line of headsets featured a different form of tracking, called inside-out [22]. Vive systems place two base stations, also called lighthouses, in the environment, preferably at the edges or corners of a room with little visual obstruction. The base stations emit pulses and sweeps of infrared light across the room, at a rate of 60 pulses and sweeps every second. Sensors on the Vive headset, controllers, and trackers time the difference between when the pulse (which hits the entire room at once) and the sweep (which hits different parts at different times) hit each sensor. Sufficient accuracy in this timing allows each sensor to know where it is in the room relative to each lighthouse, and knowing the position of multiple sensors, as well as the sensors' relative positions on each tracked object, allow the Vive to calculate a 'pose' that indicates the rotation of the object in 3D space. This tracking method allows the lighthouses themselves to be completely independent of the computer running the headset, unlike the Rift's cameras, which must connect to the PC over USB. It also means that the lighthouses are almost completely naive to the other tracking system components, save for some light on-board computations to synchronize with the other lighthouse.

2.2 Perception Studies in VR

2.2.1 Overview

Perception research, particularly visual perception, has seen may benefits from the addition of virtual reality. The general purpose of most VR applications is to generate and display a virtual environment that is perceptually identical (at least visually) to a real world experience. By using such applications, researchers can design experiments with virtual scenarios that would be expensive, impractical, or impossible to replicate in the real world. With VR comes the ability to control the experiment environment completely, facilitating faster experiment designs and more rapid research.

However, a problem discovered with such research is that visual perception in virtual environments often does not match up with real world perceptions. The most common misjudgements studied in VR are depth perception and motion perception. A wide body of distance perception research has shown significant underestimations between real and virtual environments [10, 11, 12, 17, 26, 27, 28, 29, 31]. Several theories attempting to explain this discrepancy exist, but most concern specific characteristics of HMDs as the cause of this underestimation, such as field of view limitations, lacking depth cues due to rendering inaccuracies, and the physical presence of the HMD on the head.

2.2.2 Field of View

Field of view (FOV) describes the angular extent of the image taken in by a camera, or by the human eye. Typical, unencumbered human vision is estimated at around 210° horizontally and 150° vertically, though visual perception is not uniform across this span. Since nearly all HMDs limit the field of view, FOV has been the subject of a great deal of research as a potential source of distance underestimations. Early research seemed to suggest that FOV influenced distance perception only slightly or not at all [14, 8, 4]. However, as more modern headsets began offering higher FOV displays, a growing number of studies concluded that increasing the field of view does make distance underestimation more accurate [10, 29, 11, 12, 17]. This follows from the intuition that a larger FOV would include more visual cues than a smaller one, providing more complete visual information about the scene and improving vision-based judgements. However, some research indicates that even stimulating the periphery with a completely neutral stimulus, like a white light, also improves distance estimations, leading to more questions about the role of different parts of the visual field on perception.

2.2.3 Weight and Balance

To a naive observer, perhaps the most obvious difference between viewing the real world and viewing a virtual environment with an HMD is the physical presence of the HMD on the user's head. The device adds weight to the head which is not normally present, and changes the center of mass of the head. While modern headsets are continuously getting lighter, even the flagship devices from top companies like Oculus and HTC weigh over 1lb (455g), a 10% increase in overall head weight for most people. Additionally, the weight of most headsets is concentrated towards the front, where the display panels, glass lenses to focus the image, and the majority of the housing are located. This causes the head to tend to tilt downward over time and can cause strain in the rear neck muscles over time; it may also alter the motion of the head when turning to look at objects off to the side as the user compensates for the change in center of mass. Despite this, research indicates that the change in mass and moment of inertia do not significantly influence distance judgements. An experiment using an HMD shell that allowed the subjects to see the real world while maintaining the same mass, moment of inertia, and general feel of wearing an HMD found little to no difference between the mock HMD and unencumbered real world distance estimations [27]. When subjects viewed an identical virtual environment through an actual HMD, however, the standard underestimation was observed. A further study conducted using an inertial headband to maintain the weight and balance of a headset while removing the ”feel” of wearing one also resulted in no significant distance underestimation compared to the unencumbered real world judgements [26].

2.2.4 Graphical Fidelity

Another somewhat obvious reason perception might differ in virtual environments is that, as of yet, virtual environments are visually distinct from the real world. Even setting aside distortions and FOV limitations, virtual environments are displayed using panels with limited refresh rates, resolutions, and color accuracy. Virtual environments themselves are limited by the time and effort a developer is willing to expend to make the environment realistic, as well as the computer's ability to draw the environment well and at a high frame rate. These discrepancies may cause some visual depth cues to be distorted or missing, limiting a user's ability to accurately judge depth.

However, the degree to which rendering methods and general graphical fidelity affect perception is not widely agreed upon. Interrante et al. [8] found no significant distance compression at all in a virtual environment when the environment was made to match the real world by mapping pictures of real world surface textures on to the corresponding virtual surfaces. Additionally, Willemsen et al. [26] found no significant difference between judgements made in photorealistic, low quality, and wire-frame rendered environments. By contrast, Kunz et al. [16] and Phillips et al. [21] both found judgement differences between graphical qualities using verbal and walking-based measures, respectively. Worth noting is that, even for the highest-end HMDs, and using the most meticulously rendered virtual environments, the resolution and frame rate of HMD panels is still well below the level of perceptual equivalence, with ”screen door” effects being a common complaint, caused by the noticeable gaps between the display’s pixels. However, higher resolution, better rendered graphics are often reported to induce better presence - the feeling of actually being present in the virtual environment - and are generally considered worthwhile characteristics to improve, regardless of any specific effects on distance perception.

2.2.5 Distortions

Even if the image of the virtual environment were to be displayed at such high resolution and frame rate as to be indistinguishable from real world viewing conditions, the image would be subject to distortions due to imperfections in headset technology. These come in the form of FOV distortions and lens distortions. FOV distortions occur when the virtual field of view captured by a camera in the virtual environment, called the geometric FOV (GFOV) is different from that of the HMD, called the display FOV (DFOV). While HMDs have a marketed, official DFOV specification, the actual value can differ based on the specific anatomical eye placement of the user, as well as how the user wears the headset. If a mismatch occurs, the image will appear mini- or magnified linearly across the FOV, which can change perceived distances in the distorted environment [3, 15, 18]. Methods to calibrate DFOVs have been proposed [9, 15], however Kellner et al. [13] found that even carefully matched the DFOV and GFOV still resulted in distance underestimations, though to a lesser degree than uncalibrated GFOVs.

Harder to calibrate for than GFOV distortions are lens distortions. Because the display panels in an HMD are so close to the user, lenses must be used to focus the light properly onto the retina. These lenses introduce their own distortion, most commonly radial distortion, which causes straight lines to appear bent to the user. Even after correcting for this, some research indicates that residual distortions remain, and can cause perceptual inaccuracies [25]. Lenses can also cause non-distortion artifacts, such as images in brighter regions of the display being partially visible in dimmer regions [20].

CHAPTER 3

TECHNOLOGIES

3.1 The Vive Pro

The headset chosen for the development of the HSF, as well as the subsequent user study, was the Vive Pro. The Vive lineup has situated itself as the enterprise-grade headset among its competitors, and the later models, such as the Pro, feature more accurate tracking, better ergonomics, and higher refresh rates than the original models. This makes the Vive particularly suited to user studies interested in perception in virtual environments, as the higher refresh rates and better tracking increase immersion and reduce noise. The Vive also integrates well with the Unity 3D engine, which was used to design the framework, via the SteamVR Unity plugin. The plugin offers an easy, plug-and-play solution to get started with development and served as a good baseline from which to build the rest of the framework.

3.2 Unity Overview

Unity 3D is a widely used, free, cross-platform game engine. It includes basic 3D modeling software, a physics engine, and a plethora of other features that make it suitable for this project. In addition to the usual platforms, such as PC, browser, and mobile, Unity also includes several plugins and settings that allow reasonably seamless virtual reality integration. In the case of the SteamVR plugin for the Vive, many Unity objects used to handle the integration are also usable with other headsets such as the Rift, further increasing the generalizability of this framework.

Unity also allows for some lower level graphics modification necessary for controlling certain headset parameters through the use of scriptable shaders, which use a variant of the

High-Level Shader Language, or HLSL. While the framework currently only incorporates one such custom shader, future expansions, such as chromatic aberration control, could also be implemented using shaders. Other graphics-related components can be controlled with the Render Texture object, which will be discussed in more detail in Section 3.4.

3.3 Camera Rendering

By default, camera objects in Unity display their view of a virtual scene directly to a screen output, whether this be a computer monitor, HMD screen, or some other output device. For each visual frame, a camera must perform a draw call pass, which involves several GPU state changes as the GPU renders each individual object in the scene, calculates where shadows should affect lighting, and computes which objects should be culled (removed from the camera's view). Each additional camera added to the scene introduces more switching from CPU to GPU, as each camera must perform its own draw call pass. In virtual reality applications, two cameras must be used, one simulating the view of each of the user's eyes. Rendering a scene from two points of view generally requires two draw call passes to render the scene in stereo. Fittingly, this method is referred to as Multi Pass Stereo rendering, and is the default method used in Unity.

In many VR applications, a more efficient technique called Single Pass Stereo rendering is used. This technique renders each object in the scene twice, once for the first camera and once for the second, until both cameras have the entire scene rendered. This allows for substantial overhead reductions by leveraging the fact that, in VR, the two cameras are assumed to be very close together and facing in the same direction. Shadow and culling calculations can then be performed just once for both cameras.

A further simplification made for virtual reality cameras in Unity is provided by most VR plugins. Rather than having to deal with two cameras in the scene, which could cause issues if modifications were made to one and not both, many plugins abstract this away, presenting only one camera to the user. The two views needed for VR applications are then generated artificially by automatically and transparently rendering the camera's view twice, each with a positional offset to account for the difference in eye position. This allows changes to a single camera object to apply to both eyes simultaneously, which is generally the desired outcome.

Due to limitations on accessing camera output when using either the combined camera feature or Single Pass Stereo rendering, neither method was used in the HSF. Future versions may incorporate them to increase performance.

3.4 Render Textures

In many cases, HMD manufacturers do not want users or developers changing certain parameters, such as FOV, inter-pupillary distance, and display latency, as the manufacturer wants to control the user experience as much as possible. This is reflected in the fact that most camera objects created by Unity's VR plugins (which are developed by the HMD manufacturers) do not allow for most of these parameters to be altered. However, these limitations do not exist on normal Unity cameras when they render to a non-VR display, with the notable exception of display latency. Altering the parameters required to implement the HSF as described would require circumventing this limitation somehow; the current implementation does this using Render Textures.

Cameras in Unity can also be configured to output to a texture object in lieu of rendering to a display. Such textures are called Render Textures, and they behave much like regular textures in that they can be placed on objects in the virtual environment and can be modified through the use of Materials and Shaders. Though much less efficient than rendering to the screen directly, Render Textures are crucial to being able to manipulate many headset characteristics. Render Textures are vital to the functioning of the HSF, and will be discussed in further detail as they relate to each manipulated characteristic in Chapter 4.

CHAPTER 4

HMD CHARACTERISTICS

4.1 Overview

The purpose of this framework is to simulate the characteristics of an HMD in software as accurately as possible in order to allow a user study to identify perceptual equivalence classes between HMD characteristics. The characteristics currently supported by the framework for manipulation include inter-pupillary distance, field of view, viewport position, multiple viewports, resolution, latency, geometric distortion, and custom shader application. Included by default is a custom shader that allows for nonlinear distortion, such as radial distortion, but this shader could be replaced to accommodate basically any graphical manipulation desired, albeit with quite a bit of effort.

The main insight that allows the HSF to work is that the cameras which output to the HMD (the 'main' cameras) do not see the virtual environment directly. Each main camera is paired with a duplicate, coincident camera that sees the environment (a 'render' camera) in the main camera's stead. While this may sound contrived, it makes accessible several characteristics that are otherwise inaccessible, including latency, resolution, and lens distortions.

Each render camera is configured to send its view to a Render Texture, which is then placed on a plane. This render plane is placed in front of and is seen by the main camera corresponding to the sending render camera. Because each main camera is coincident with its corresponding render camera, the main camera effectively sees the same view as the render camera. This can be thought of as the main cameras viewing a screen, on which is projected the virtual environment that the main cameras would see if it were not hidden from them

(See figure 4.1). Layers and layer masks are used to selectively render the appropriate objects to each camera.

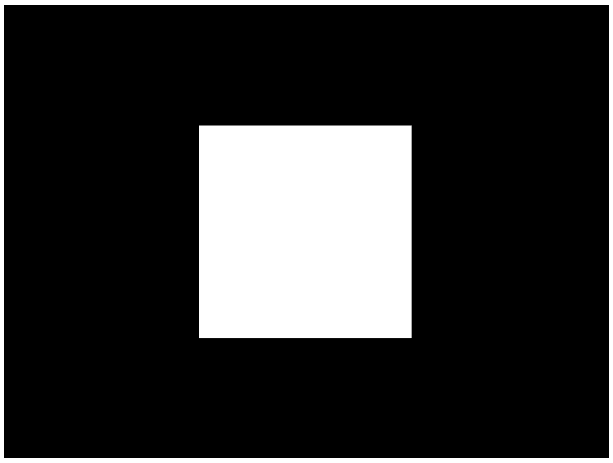
In Unity, layer masks are 32-bit masks attached to each object intended to group objects together based on type. The first 8 layers/bits are defined by Unity explicitly, and the remaining 24 are user-definable. By placing a layer mask on a Unity camera, a developer can select which layers the camera should and should not see; objects in the unseen layers will not be rendered to the camera, though they will still affect the environment physics as normal. In the HSF, each camera employs layer masks to ensure that the main cameras only see the render planes associated with themselves, and the virtual cameras cannot see any of the render planes at all.

4.2 Inter-pupillary Distance

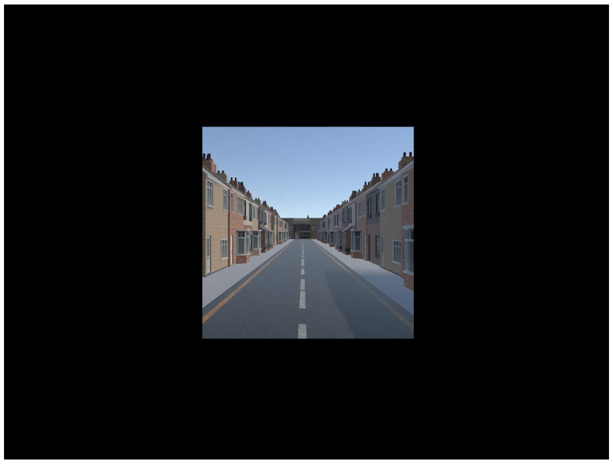
Inter-pupillary Distance, or IPD, is the distance between the pupils of the left and right eyes. This distance varies from person to person, and it informs the placement of cameras in a virtual environment, as the two cameras in a virtual environment configured for VR represent the user's eyes. Most flagship HMDs have adjustable lens spacing that allows a user to position the lenses such that the center of each lens sits directly in front of the user's pupil, which minimizes distortions and reduces eye strain. The spacing value is then sent from the HMD to the application, which can then adjust its render cameras (or the spacing parameter on a single camera) to match the lens spacing chosen by the user. By default, when using a single camera to render two viewpoints, as in the case for VR, the distance between the artificial viewpoints cannot be changed by the user, as that parameter is overridden and locked by the HMD's value.

Many applications default to an IPD of 64 millimeters (32 millimeters off center for each eye), near the population average [5], as it is close enough that most users do not notice any discomfort. However, some perception studies conducted in VR require precise knowledge of and accounting for the user's actual IPD, and some are directly interested in

(a) The environment as seen by the render cameras



(b) The render plane with a 30° field of view as seen by the main camera



(c) The render plane as seen by the main camera with the associated render camera enabled

Figure 4.1. Views of the Scene

potential effects of IPD on perception. By using the render camera/Render Texture method, and since a main camera is used for each eye, controlling the IPD is achievable by simply moving the main cameras in the environment, and moving the render cameras and render planes to match. This is handled nicely using Unity's hierarchy, which allows objects to become children of other objects; when a parent object moves or rotates, all children move or rotate to match. Thus, by structuring the hierarchy such that the render cameras are children of their respective main cameras and the render planes are children of their render cameras, simply moving the main cameras closer together or farther apart serves to change the IPD with no adverse visual effects.

4.3 Field of View

Another feature that the render plane method facilitates is field of view modification. Field of view, or FOV, is a measure of the angle through which light can enter a sensor, be it a camera sensor or the retina in the eye. In humans, the generally accepted FOV is around 210° horizontally and 150° vertically, though variations in brow and nose size, as well as the depth of the eye in the socket, affect this measurement. Each HMD has a maximum FOV it is capable of displaying, limited primarily by the size of the panels and the focusing power of the lenses used. This maximum FOV is a hardware limitation, and thus cannot be exceeded in software, though it can be limited in software by blacking out the extremities of the HMD screen.

Like IPD, a Unity camera outputting to an HMD requests the field of view from the display itself and prevents the user from changing it. Previous attempts to change FOV for a VR camera have involved placing virtual planes colored black and unaffected by lighting directly in front of the camera, creating a 'window' that the camera can see the environment through. This solution is clunky; changing the size of the window requires moving each plane by a calculated amount, and issues can arise when objects in the virtual environment accidentally pass through the plane and enter the camera's view (called 'clipping').

Luckily, the render plane technique bypasses both of these issues, preventing clipping issues and allowing quick and easy resizing of the viewing window. Since the main cameras only see the render plane, setting the main camera's background color to solid black and rescaling the render plane creates a viewing window. Because there are no objects physically occluding the camera's view of the environment, clipping issues are avoided. The plane object, like all Unity objects, has a "scale" property that alters the size of the plane in the x, y, z coordinate system. With the current implementation, a user can provide a desired FOV in degrees for both horizontal and vertical axes. Since the distance from the camera to the render plane is known, a simple trigonometric calculation sets the scale of the plane so that it subtends the specified visual angles for the main cameras. Proper update synchronization with Unity's frame rate-driven Update callback ensures that altering the FOV even at runtime does not result in any visual artifacts like stretching or jumping.

4.4 Geometric Distortion

When rendering a camera's view to a texture, it is important to note that the captured image from the camera is given directly to the render texture, regardless of any difference in size between the texture and the camera's FOV. Thus, it is important for this setup to ensure that the render camera's FOV angle (the GFOV) matches the angle subtended by the render plane (effectively the DFOV) for normal operation. If the GFOV is smaller than the DFOV, the view would appear stretched out, as Unity tries to map a smaller source texture (the render camera's view) to the larger destination texture (the render plane). Conversely, a GFOV would cause the image to appear compressed.

To control this, the HSF contains a multiplier that controls the GFOV's size relative to the DFOV, which is set explicitly by the user. If the FOVs are different, the geometry of the scene is distorted. Worth noting, this method of distortion can only result in linear distortion across the FOV, whereas this and more complicated distortions can be added using the shader effects described in Section 4.5. However, linear distortion is much easier to implement; rather than having to generate a distortion map image, this method only requires altering a single value. Using a GFOV multiplier ($x > 1$) results in environmental compression, or minification, and multipliers ($0 < x < 1$) visually magnify the environment.

4.5 Texture Effects

Object texturing is handled in Unity via materials. A material contains a main texture, any secondary textures used for normal mapping or reflection mapping, parameters for tiling and offsetting a texture across a surface, and other properties specific to the shader being used on that material. Shaders in Unity control how each object is rendered on a pixel-by-pixel basis; with a complex enough shader, almost anything can be rendered on any object. In the initial implementation of this framework, custom shaders did not need to be supported explicitly, since a user could change the default material on the render planes to use any available shader. However, the introduction of the latency parameter described in Section 4.6 required that a new material be created at runtime, which would overwrite the default material and its shader. By explicitly incorporating shader handling in the HSF, it is possible to specify a material during set up and to change the material at runtime without interfering with the latency implementation.

This effectively allows any modification normally allowed for textures to be applied to the view of the main cameras. To apply some of these changes to cameras normally requires changing engine settings, which is tedious and unavailable at runtime. The most prominent characteristic this opens up for modification is resolution. Unity cameras normally default to the resolution of the output display, in this case the HMD. Changing the resolution is possible, but is a global setting, not a camera-specific one. On the other hand, textures, like those on the render planes, have a resolution property that can be changed as desired. Similarly, anti-aliasing and color space type can be controlled on textures individually. As mentioned previously, more complicated rendering tricks can be accomplished through the application of custom shaders.

By default the HSF includes a custom shader that supports adding distortion to the render plane. The shader, written by Ethan Luckett [19], uses a secondary texture to encode a mapping between the main input texture (the render camera's view) and the final rendering (which is seen by the main camera), and permits any desired texture distortion. By default, the secondary remapping texture adds no distortion to the plane, and is indistinguishable from the Unity standard shader. This allows the top levels of code to hide the shader itself and only expose a parameter for the remapping texture. Adding a new custom material is also supported, but any specific parameters would have to be set manually on the material itself, rather than using the Script Master (detailed in Chapter 5) to handle the modification, as is the case with the other modified characteristics.

4.6 Display Latency

By far the single most difficult part of this framework to implement was display latency. By default, cameras in Unity send output to their displays as soon as the view is rendered. This "as soon as possible" rendering ensures that the resulting frame rate is only limited by the display framerate and the capabilities of the system. It is also efficient, only storing the current frame in a buffer while it is being rendered, rendering the frame to the display, and then overwriting that buffer on the next frame. However, it also means that there is no easy way to intercept the camera's output before it is actually rendered, either to a display or a render texture. This makes grabbing each frame and withholding it from the display for a set period of time (i.e. adding latency) difficult. It is possible to directly read from the screen backbuffer using built-in Unity functions, but doing so is extremely inefficient, as it requires the CPU to read GPU memory, causing overhead as the CPU waits for the GPU to finish rendering before getting the data itself. In my testing, doing this in real time, so that each frame is stored and can be presented after some delay, was impossible. My attempts caused frame rates consistently below 30 frames per second (fps), which is well below the standard for virtual reality, 90 fps.

My first attempts to circumvent this issue turned to methods of adding latency that did not require storing frames at all, but rather storing transforms. Transforms contain references to the position, rotation, and scale of each object in a virtual environment, including those not rendered. By keeping a buffer of the camera's transforms from each frame, it is possible to 'replay' the transforms by applying each back to the camera object after a certain amount of time has passed. This method has two major downsides. Firstly, it tends to cause stuttering, as the camera's position and rotation (a camera's scale property has no effect whatsoever) is now updated twice per frame, once by the actual HMD movement, and again when the transform is overwritten by the buffer value. Even attempts to properly synchronize the transform updates resulted in noticeable stuttering, especially if the user happened to move very suddenly. Secondly, this method is severely limited in what it can and cannot delay. Since the camera is the only object being delayed, all movement of objects in the environment will still be seen in real time. Only the user's own head movements are actually delayed. This severely limits the utility of even including this parameter, and nearly resulted in it's removal from the framework.

A final attempt to implement latency properly led me to a different solution. While copying texture information from the screen backbuffer requires the CPU and induces overhead, copying from a render texture to a normal texture object can be done solely on the GPU. By itself this fact is useless, as the render textures, like the screen backbuffer, update in real time with no way to halt a frame between the camera and output texture. However, since each frame of the camera's output can be copied to a regular texture, copying to the same texture on every frame effectively turns the texture into a render texture, but one that can be saved in a buffer. Under the current implementation, the render planes do not actually use render textures. Rather, the render cameras send their output to a render texture that is never displayed, only read from and copied to a texture object, which is then placed in a buffer, each frame. Then, when a specified time has passed, a frame is read back out of the buffer and applied to the render plane as a static texture. Since the render plane's texture is updated every frame the illusion of motion is maintained, and since the GPU is capable of rendering the render texture and copying it on the same frame, setting the latency parameter to zero results in zero latency, as expected.

4.7 Multiple Viewports

Lastly, using render planes as 'windows' through which the camera can see the environment rather than occluding the environment with black planes also opens up the ability to create multiple render planes, and thus multiple viewports, in the same scene. Since all parameter manipulations are handled on a render plane basis, each viewport has its own separate set of characteristics. The viewports can be configured in several ways. A smaller viewport could be layered on top of a larger one, where the larger viewport has, for example, a lower resolution than the smaller viewport. This could facilitate a study concerning the effect of dynamic display resolution across a user's field of view.

Because of the established hierarchy configuration, it is also possible to move each viewport around within the HMD's field of view by rotating a render camera (but not the main camera). Since the render plane is a child of the render camera, and since the main and render cameras are coincident, this ensures that the perspective of the moving viewport is still correct and that the render plane is always correctly positioned. Thus, disparate viewports can be created and separated from each other. While I am not aware of any existing experiments that have done this, I can imagine a study interested in peripheral stimulation or motion perception in the periphery having use of this feature.

CHAPTER 5

FRAMEWORK DESIGN

When designing this framework, one of my main goals was to make it as simple as possible to interact with. I wanted a researcher with a design for an experiment to be able to drop this framework into their project and use it just as they would any other VR plugin object in Unity, just with more tailored features. However, I also know well how specific some perception studies can be, and realized that, despite my best efforts, many studies would require more or different characteristics to be modified. Therefore, I also wanted to structure the code in a way that allows for easy extension. At the very least, this framework was built over the course of three semesters, and I needed something modular enough to pick back up when I myself came back after a break to continue development.

5.1 Master and Control

Starting from the top, the user-visible head of the framework is a 'master' script which is visible in Unity's inspector GUI. The master serves mostly as a UI, and only directly modifies a single parameter. Rather, a second-layer script called control does the actual characteristic modification. This top level structure serves two purposes. First, the master script can be completely rewritten to better fit another user's UI needs, and the only thing the new master script needs to interact with is the control script, which holds the full set of properties that handle all lower level functionality. The master script can also apply its own computations to user entered data before sending it to control, allowing for more flexibility on the user's end without having to delve too deep into the codebase. Second, each control script handles only a single viewport, meaning that multiple control scripts are needed to implement the multi-viewport option described in Section 4.7. The master dictates how many control scripts are instantiated when the environment starts, and therefore how many viewports there are. The master also controls where the viewports are shown on the screen positionally. Due to limitations in the inspector itself, the master only exposes one control script for modification via the inspector at a time; the user can switch between them using a drop down menu. Since some study designs may have no use of this feature, the master could be rewritten much simpler by excluding this functionality.

All input to the framework comes from either the Unity inspector interface or in a separate, experiment-specific script written to alter the master's properties for each trial of the experiment. The properties from the master are passed directly to the appropriate control script, which triggers any relevant calculations before sending the parameters down to the work scripts. Early iterations of the master script had trouble dealing with changes triggered from the inspector. While Unity does support changing fields in the inspector at runtime, it does not allow properties to be drawn in the inspector. My first solution was to make every backing field public and check every frame if each field had changed, triggering the appropriate property methods when one did. While this was made simple by Unity's Update callback, which runs on every frame, it is a sloppy solution, and inefficient to boot. After some digging, I came across a free utility plugin called Property Backing Field Drawer by Candlelight Interactive. Property Backing Fields allow fields to modify their corresponding properties, but only when the field is modified by the user in the inspector. Since fields can be drawn by the inspector, this essentially allows properties to be exposed in the inspector as well. This simplified the solution greatly by allowing me to keep private fields private as intended while still allowing the user to trigger the associated property methods from the inspector window without the need for extra code.

With or without a master script, each control script handles the initialization of the numerous smaller scripts that do most of the heavy lifting in changing the headset characteristics. The control scripts also keep track of an ID number given to them on initialization by the master script that ensures that each render plane is presented at a slightly different depth, so that overlapping render planes do not cause depth buffer artifacts. Because the FOV calculations directly read the distance between the render plane and main camera, this does not cause any issues with changing the field of view.

5.2 Characteristic Modification

After the control script, there are three scripts which control different characteristics of the render planes. First is a script which controls IPD. Since this script must maintain a reference to the main cameras anyway, is also contains functionality that allows for the entire framework to essentially be turned off. By inverting the main camera's culling masks, the main cameras no longer see the render planes, but do see the rest of the environment. This was initially used for debugging purposes, but could have use in a study comparing limited FOV to full FOV conditions.

The second script controls the render plane object itself. This script handles changes to the size of the render plane, and thus the effective FOV. This script also controls the geometric distortion modification, which is implemented as a simple multiplier that scales the plane size relative to the render camera's FOV. This is a rather simple calculation, but is made slightly more confusing by Unity's method of dealing with camera fields of view. A camera's 'field of view' field only controls the vertical FOV of the camera; the horizontal FOV is then calculated using the camera's aspect ratio. This made adding the multiplier to the horizontal and vertical axes independently difficult, and the current implementation only supports one multiplier, applied to both axes simultaneously.

The third script is the most complicated, as it handles the texture properties of the render plane. Resolution and antialiasing are handled here, and any custom shaders are applied and modified if necessary. Most importantly, latency is dealt with entirely within this script. The texture script maintains the buffer containing the stored textures, and the code that iterates over the buffer each frame to find the appropriate texture to display. Buffer mismanagement early in development caused several memory-related issues due to Unity's handling of garbage collection. When old textures were overwritten in the buffer, they were never picked up by the garbage collector, which caused the environment to slow over time as more and more memory was used up and eventually caused Unity to crash. Performing manual garbage collection periodically was problematic since Unity uses the Boehm–Demers–Weiser garbage collector, a stop-the-world collection system that is, obviously, not suited to such latency-sensitive applications. Much care was taken to ensure that in all cases the original texture buffer's references were maintained and modified, rather than overwritten, throughout the life cycle of each run.

CHAPTER 6

STUDY DESIGN

6.1 Overview

When designing this study, my goal was to find an experiment that would contribute a useful result to the field of perceptual psychology in virtual reality, but that could also leverage a sizeable portion of the HSF, preferably using at least three display characteristics. This required finding a type of visual perception that relies on several different parts of vision. Vection, or motion perception, is well suited for this, as it is sensitive to several of the available parameters.

One of the easiest characteristics to alter that affects vection is field of view. Limiting a person's field of view affects their sense of motion in two ways. First, the outer range of the field of view, called the periphery, is more temporally sensitive than the center, which is more sensitive to color and fine spatial information. This makes the periphery particularly suited to detecting motion, and occluding it by limiting the field of view limits the degree to which this part of the visual field can inform a person's sense of motion. This effect is neurological, and more or less hardwired into the structure of the eye itself. Second, limiting the peripheral FOV has a geometric effect on the total amount of motion occurring in the field of view. If the environment is moving toward the user, the angular distance subtended by objects further to the side (and thus, in the periphery) is higher than objects in the center. By limiting the periphery, objects with higher angular velocities are occluded, reducing the average angular velocity of the visual field. By contrast, removing the center of the field of view would have the opposite effect, raising the average angular velocity. In general, it has been shown that reductions to the periphery tend to cause less motion perception than full

FOV conditions.

Interestingly, inducing distortions to the field of view can cause one of these effects to persist while the other is removed. Adding a linear geometric distortion to the field of view (e.g. by increasing the source camera FOV without increasing the output FOV) has the effect of compressing more environment geometry into the same FOV. This means that objects at the far periphery are even farther to the side environmentally than the visual extent of the periphery would suggest, causing them to appear to move even faster than they normally would. Thus, the geometric effect is maintained, while the neurological effect is removed.

6.2 Stimulus and Response

While testing multiple display characteristics in one study uses the HSF more fully, it also involves an increase in the number of trials required to gather a reasonably sized data set from each subject. Most studies run in the High Fidelity Virtual Environments (HI5) Lab use some form of walking task to determine a subject's sense of depth or motion in a virtual environment, but these tasks tend to take a minute or more per trial, and rough calculations indicated that the experiment could take an hour and a half or more to test three characteristics fully. This could introduce fatigue effects over time, especially since the subject would be walking almost the entire time.

To circumvent this issue, I chose a different response method called two alternative forced choice (2AFC). 2AFC presents the subject with two stimuli, a reference and a target, and asks them to answer a binary question about the stimuli. In this experiment, each stimuli consisted of a city street scene, with buildings on either side of the subject. The subject was physically seated in a chair, and moved automatically down the virtual street. The reference stimulus of each trial lasted three seconds, after which the headset screen was masked for 500ms. Then the target stimulus was shown three seconds; in the target stimulus, an alteration was made to one of the display characteristics. At the end of the target stimulus, the HMD was masked and the subject was asked to indicate using a computer mouse whether they perceived a change in speed between the reference and target scenes. To maximize consistency between trials, brief instructions were presented via text on the masked background indicating how the user should respond in either case - left click for a change in speed, right click for no change. Each stimulus in every trial in the experiment moved at the same speed: 10 meters per target. Thus, subjects' responses were taken to indicate the effect of the target stimulus's alteration on the subjects' perception of their own motion through the virtual environment. Then the next trial would begin, starting with the reference stimulus again. Each experiment required the subject to be in the headset for approximately 13 minutes, significantly less than any walking procedure design of reasonable length.

6.3 Blocks

The experiment was divided into three blocks, one to test each of three parameters. The first block altered field of view, and also acted as a calibration for the second and third blocks. The second and third blocks, which altered geometric distortion and resolution, respectively, were independent measures and were chronologically inverted for half of the subjects to help mitigate ordering effects. Each block used four levels of stimuli, and each level was shown eight times within the block. This included a control stimulus, where the reference stimulus level was the same as the target.

The first block altered the subjects field of view in the target stimulus. The reference stimulus had an FOV of 60° and the four target FOVs were 30°, 40°, 50°, and 60° (the control stimulus). Each FOV condition, in all blocks, was shown at a 4:3 aspect ratio. It is hypothesized that as target FOV decreases, the subject would be more likely to perceive a difference in vection between stimuli. While this result is significant on its own, this block also acted as a way to calibrate the second two blocks on a subject-by-subject basis. After averaging the responses for each stimulus level, the data generated from this block can be

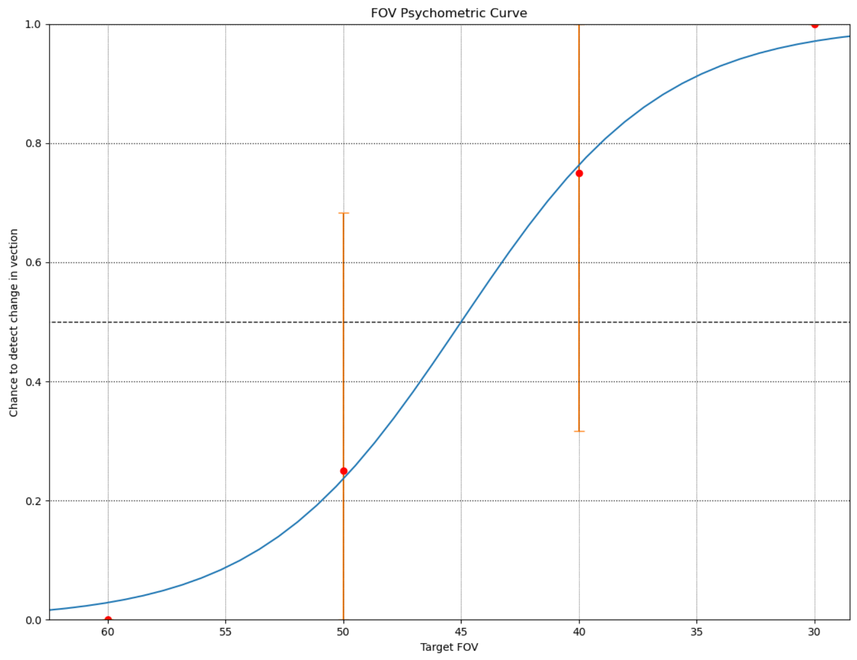


Figure 6.1. Example Psychometric Curve

fitted with a psychometric curve, as shown in Figure 6.1 using test data.

This curve indicates, for different target FOV conditions, the likelihood that a subject will perceive a change in motion. The bottom end of the curve, corresponding to the control condition, is assumed to have a near-zero average, since the speed does not actually change between stimuli. If the hypothesis is correct, the top end of the curve, corresponding to the 30° condition, should have a near-100% average, since this condition removes a significant portion of the field of view. In addition to the top and bottom of the curve, the 50% value, called the point of subjective equality (PSE), indicates the FOV for which the subject would be unable to determine if a change in speed occurred or not, and must simply guess (this assumes that the subject guesses randomly). This FOV value is denoted FOV_{PSE} Two other important points are the 25% and 75% values. The former indicates the FOV size above which the subject will not perceive a speed difference at least 75% of the time, and is denoted FOV_{lower}. Conversely, the later value indicates the FOV size below which the user will perceive a speed difference at least 75% of the time, and is denoted FOV_{upper}. Note that the subscript denotes the FOV's position on the psychometric curve, not it's size, and that FOV$_{lower}$ is larger than FOV$_{upper}$ in terms of degrees subtended.

The goal of blocks 2 and 3 is to determine if altering other visual parameters, geometric distortion and resolution, respectively, can push the FOV$_{upper}$ and FOV$_{lower}$ conditions perceptually closer to the PSE obtained in block 1. That is, does adding distortion or changing resolution make the subject less certain if a change in speed is occurring or not. Each trial in block 2 presented FOV$_{upper}$ in both the reference and target stimuli. A geometric compression was added to the target stimuli, increasing the effective field of view without changing the size of the render plane. The reference stimulus had a compression multiplier of 1, meaning no distortion, and the target stimuli had multipliers of 1 (control), 1.25, 1.5, and 1.75, corresponding to a 0, 25, 50, and 75 percent increase in geometric FOV, respectively, relative to the control. Our hypothesis for this block is that, since increasing the geometric FOV increases the total angular speed of the viewing window, subjects would perceive a change in vection more often than the baseline 25% caused by the FOV$_{upper}$ condition in block 1.

Block 3 is structurally identical to block 2. Block 3 uses FOV$_{lower}$ for all trials, for both stimuli. Because FOV$_{lower}$ is larger than FOV$_{PSE}$, making FOV$_{lower}$ perceptually closer to FOV$_{PSE}$ would involve removing visual stimuli. Block three accomplishes this by introducing a third parameter: resolution. In blocks 1 and 2, all stimuli shown to the subject are rendered at the max resolution of the Vive Pro. In block 3, the target stimuli are shown at resolutions of 1024x768 (control), 512x384, 256x192, and 128x96. These values, which were used in initial testing, mistakenly do not account for the fact that the FOV$_{lower}$ is different between different users, and should have applied a multiplier to the resolution instead. Unfortunately, due to circumstances detailed in Section 6.5, no subjects were run after this correction was made.

6.4 Latin Squares

In perception studies, ordering effects of stimuli can sometimes influence the subject's response. For example, a subject may be more likely to perceive a change in vection with a target FOV of 50° when viewed after a target FOV of 60°, but not after a target of 40°. This can be offset somewhat by randomly shuffling the order of the stimuli, but this does not guarantee that ordering effects are not present in any single subject, and requires a large amount of data to be collected in order to wash out potential ordering effects.

To help avoid this issue, this experiment shuffles the stimuli order based on an implementation of a Latin Square. A Latin Square in an n by n array containing n copies of n distinct symbols, such that each row and column of the array contains each symbol only once. By using the stimuli levels as symbols, a randomly generated Latin Square can generate a trial order by stringing together the rows into a 1 by n^2 vector. This ensures that the stimuli levels are evenly distributed throughout the experiment, and that the order of each group of stimuli is unique. In this experiment, since there were 4 levels of stimuli, the number of repetitions for each stimuli had to be a multiple of 4; 8 was chosen to balance experiment duration and results confidence. This meant generating two Latin Squares stringing both together, for a 1 by $2(n^2)$ vector of trials, for 32 trials per block.

6.5 COVID-19 Complications

Unfortunately, due to the 2020 COVID-19 pandemic, all research involving human contact was suspended at the University, including virtual reality research. This suspension began less than two days after the experiment design was completed, and I was only able to run the experiment with a single subject to verify that my procedure was ready for subjects. Due to the resolution error explained in Section 6.3, I was not able to gather any useful information from the data I had.

CHAPTER 7

CONCLUSION

7.1 Summary

In this work, I have introduced a new framework for Unity, the HMD Simulation Framework. The HSF allows a researcher or developer to quickly and easily modify a wide array of HMD characteristics from the Unity Inspector GUI. I have also described a visual perception experiment that leverages the breadth of the HSF's capabilities to find equivalence classes for different HMD characteristics as they apply to motion perception in virtual environments. Though the study was not completed due to unforeseen circumstances, I hope that the HSF might facilitate similar research in the future.

7.2 Limitations and Future Work

While the HSF covers many commonly studied HMD characteristics, there are still some software-modifiable parameters that are left unimplemented. Most notably, different headsets use different tracking methods, some of which are known to cause a slight stuttering when the headset is positioned sub-optimally. Additionally, nearly all tracking methods induce small, compounding errors that result in tracking 'drift' over time. This drift is not always noticeable, especially during short usage or with regular calibration, but for very sensitive perception experiments, this drift could effect the results. Both of these tracking inaccuracies can be simulated to a degree; drift is quite measurable, and can be simulated quite well, while stuttering can be approximated by introducing systematic small-scale noise directly to the virtual position and rotation of the headset. These characteristics were considered, but deemed non-essential in favor of characteristics with more effect on motion perception.

While custom shader effects are supported by the framework, shaders other than the distortion shader included by default are not integrated with the UI. Thus, an extension would need to be written for the master script to interface directly with the shader's properties from the UI. Future work could generalize the current UI code to automatically reference and display a given shader's properties, which would increase the usability of the framework.

Lastly, for near-future versions of Unity, the functionality of the HSF may be completely non-functional, as Unity moves to a new rendering pipeline, the High-Definition Render Pipeline (HDRP). The HDRP includes several new features regarding shader handling, and renders shaders built using the standard pipeline useless unless converted to HDRP equivalent versions. Since VR support for the HDRP was added in version 2019.3, after the HSF was already in development, the standard pipeline was used. Additionally, the HDRP only supports single pass rendering for VR displays. Since the current implementation relies on the ability to separate the cameras and render them separately, single pass rendering is currently incompatible with the HSF. Implementing support for the HDRP would require a sizeable refactoring of the rendering methods currently used, but would likely ensure that the HSF would continue working as intended for several years to come.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Facebook to Acquire Oculus. about.fb.com/news/2014/03/facebook-to-acquire-oculus/. Accessed: 2020-4-15.

[2] History of Virtual Reality. vrs.org.uk/virtual-reality/history.html. Accessed: 2020-4-12.

[3] G. Bruder, A. Pusch, and F. Steinicke. Analyzing effects of geometric rendering parameters on size and distance estimation in on-axis stereographics. In *Proceedings of the ACM Symposium on Applied Perception*, pages 111–118, 2012.

[4] S. H. Creem-Regehr, P. Willemsen, A. A. Gooch, and W. B. Thompson. The influence of restricted viewing conditions on egocentric distance perception: Implications for real and virtual indoor environments. *Perception*, 34(2):191–204, 2005. PMID: 15832569.

[5] N. A. Dodgson. Variation and extrema of human interpupillary distance. In *Stereoscopic Displays and Virtual Reality Systems XI*, volume 5291, pages 36–46. International Society for Optics and Photonics, 2004.

[6] Dom Barnard. History of VR - Timeline of Events and Tech Development. virtualspeech.com/blog/history-of-vr. Accessed: 2020-4-12.

[7] P. Frost and P. Warren. Virtual reality used in a collaborative architectural design process. In *2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*, pages 568–573, 2000.

[8] V. Interrante, B. Ries, and L. Anderson. Distance perception in immersive virtual environments, revisited. In *IEEE Virtual Reality Conference (VR 2006)*, pages 3–10, March 2006.

[9] J. A. Jones, L. C. Dukes, D. M. Krum, M. T. Bolas, and L. F. Hodges. Correction of geometric distortions and the impact of eye position in virtual reality displays. In *2015 International Conference on Collaboration Technologies and Systems (CTS)*, pages 77–83. IEEE, 2015.

[10] J. A. Jones, D. M. Krum, and M. T. Bolas. Vertical field-of-view extension and walking characteristics in head-worn virtual environments. *ACM Trans. Appl. Percept.*, 14(2):9:1–9:17, Oct. 2016.

[11] J. A. Jones, J. E. Swan, II, G. Singh, and S. R. Ellis. Peripheral visual information and its effect on distance judgments in virtual and augmented environments. In *Proceedings of the ACM SIGGRAPH Symposium on Applied Perception in Graphics and Visualization*, APGV '11, pages 29–36, New York, NY, USA, 2011. ACM.

[12] J. A. Jones, J. E. Swan II, and M. Bolas. Peripheral stimulation and its effect on perceived spatial scale in virtual environments. *IEEE Transactions on Visualization and Computer Graphics*, 19(4):701–710, 2013.

[13] F. Kellner, B. Bolte, G. Bruder, U. Rautenberg, F. Steinicke, M. Lappe, and R. Koch. Geometric calibration of head-mounted displays and its effects on distance estimation. *IEEE transactions on visualization and computer graphics*, 18(4):589–596, 2012.

[14] J. M. Knapp and J. M. Loomis. Limited field of view of head-mounted displays is not the cause of distance underestimation in virtual environments. *Presence: Teleoperators and Virtual Environments*, 13(5):572–577, 2004.

[15] S. A. Kuhl, W. B. Thompson, and S. H. Creem-Regehr. Hmd calibration and its effects on distance judgments. *ACM Transactions on Applied Perception (TAP)*, 6(3):1–20, 2009.

[16] B. R. Kunz, L. Wouters, D. Smith, W. B. Thompson, and S. H. Creem-Regehr. Revisiting the effect of quality of graphics on distance judgments in virtual environments: A comparison of verbal reports and blind walking. *Attention, Perception, & Psychophysics*, 71(6):1284–1293, 2009.

[17] B. Li, J. Walker, and S. A. Kuhl. The effects of peripheral vision and light stimulation on distance judgments through hmds. *ACM Trans. Appl. Percept.*, 15(2), Apr. 2018.

[18] B. Li, R. Zhang, and S. Kuhl. Minication affects action-based distance judgments in oculus rift hmds. In *Proceedings of the ACM Symposium on Applied Perception*, pages 91–94, 2014.

[19] E. Luckett. *Assessing Distance Perception in Virtual and Augmented Realities using Electroencephalography.* PhD thesis, University of Mississippi, 2020.

[20] W. Panlener. Effect of constrained field of view on frontal distance estimation in a virtual environment. Master's thesis, University of Mississippi, 2017.

[21] L. Phillips, B. Ries, V. Interrante, M. Kaeding, and L. Anderson. Distance perception in npr immersive virtual environments, revisited. In *Proceedings of the 6th Symposium on Applied Perception in Graphics and Visualization*, pages 11–14, 2009.

[22] J. P. Rolland, L. D. Davis, and Y. Baillot. A survey of tracking technologies for virtual environments. In *Fundamentals of wearable computers and augmented reality*, pages 83–128. CRC Press, 2001.

[23] R. M. Satava and S. B. Jones. Current and future applications of virtual reality for medicine. *Proceedings of the IEEE*, 86(3):484–489, 1998.

[24] I. E. Sutherland. The ultimate display. *Multimedia: From Wagner to virtual reality*, 1, 1965.

[25] J. Tong, R. S. Allison, and L. M. Wilcox. The impact of radial distortions in vr headsets on perceived surface slant. *Journal of Imaging Science and Technology*, 2019.

[26] P. Willemsen, M. B. Colton, S. H. Creem-Regehr, and W. B. Thompson. The effects of head-mounted display mechanics on distance judgments in virtual environments. In *Proceedings of the 1st Symposium on Applied Perception in Graphics and Visualization*, APGV '04, pages 35–38, New York, NY, USA, 2004. ACM.

[27] P. Willemsen, M. B. Colton, S. H. Creem-Regehr, and W. B. Thompson. The effects of head-mounted display mechanical properties and field of view on distance judgments in virtual environments. *ACM Trans. Appl. Percept.*, 6(2), Mar. 2009.

[28] P. Willemsen, A. A. Gooch, W. B. Thompson, and S. H. Creem-Regehr. Effects of stereo viewing conditions on distance perception in virtual environments. *Presence: Teleoperators and Virtual Environments*, 17(1):91–101, 2008.

[29] B. Wu, T. L. Ooi, and Z. J. He. Perceiving distance accurately by a directional process of integrating ground information. *Nature*, 428(6978):73–77, 2004.

[30] X. Yang, R. C. Malak, C. Lauer, C. Weidig, H. Hagen, B. Hamann, and J. C. Aurich. Virtual reality enhanced manufacturing system design. In *Proceedings of the 7th CIRP international conference on digital enterprise technology*, pages 125–133, 2011.

[31] R. Zhang, A. Nordman, J. Walker, and S. A. Kuhl. Minification affects verbal- and action-based distance judgments differently in head-mounted displays. *ACM Trans. Appl. Percept.*, 9(3), Aug. 2012.

VITA

Jonathan Hopper was born on April 17, 1997 in Tupelo, Mississippi. In May 2014 he graduated high school, and the following August he entered the University of Mississippi. In May 2018 he received a Bachelor of Science in Computer Science. He is currently pursuing a Master's Degree in the same field at the University of Mississippi, specializing in virtual reality and human visual perception.