Foveated Rendering Algorithms using Eye-Tracking Technology in Virtual Reality

N. MARIANOS, *Technical University of Crete*

This thesis was submitted for the Bachelor’s degree in Electrical and Computer Engineering in the Department of Electrical and Computer Engineering to the

Technical University of Crete, Greece through the laboratory of Distributed Multimedia Information Systems and Applications – MUSIC
Month: March 2018, CHANIA
Author: Nikolaos – Xenofon Marianos

Ευχαριστίες

Θα ήθελα να ξεκινήσω από την επιβλέπουσα καθηγήτρια μου, κυρία Αικατερίνη Μανιά, την οποία ευχαριστώ για την εμπιστοσύνη που μου έδειξε και για την ευκαιρία που μου έδωσε ώστε να ασχοληθώ με το συγκεκριμένο θέμα και υλικό.

Έπειτα θα ήθελα να ευχαριστήσω τους ανθρώπους που με βοήθησαν σε δύσκολες στιγμές της διπλωματικής αυτής εργασίας. Οι άνθρωποι αυτοί αν και ήταν σε διαφορετικές χώρες από την δική μας έδειξαν μεγάλη προθυμία και με βοήθησαν όσο μπορούσαν από εκεί που βρίσκονταν. Ευχαριστώ πολύ λοιπόν την κυρία Marka Neumann (Office Manager, Arrington Research Inc) καθώς και τον κύριο Γεώργιο Αλέξανδρο Κουλιέρη για όλη τους την βοήθεια.

Θα ήθελα να ευχαριστήσω επίσης τους καθηγητές και μέλη της επιτροπής της διπλωματικής εργασίας κύριους Δεληγιαννάκη Αντώνιο και Μπάλα Κωνσταντίνο για τον χρόνο που θα διαθέσουν στην ανάγνωση και διόρθωση της εργασίας.

Ένα πολύ μεγάλο ευχαριστώ στα άτομα τα οποία συμμετείχαν στα πειράματα τα οποία βοήθησαν με σημαντικές παρατηρήσεις και αξιολογήσεις.

Τέλος ένα μεγάλο ευχαριστώ το οφείλω στην οικογένεια μου αλλά και στους φίλους οι οποίοι με βοήθησαν να ξεπεράσω τις όποιες δυσκολίες βρέθηκαν στον δρόμο μου και μου έδωσαν την δύναμη να συνεχίσω και να ολοκληρώσω αυτό το δύσκολο έργο.

Acknowledgments

I would like to start by thanking my supervisor Assoc. Prof. Katerina Mania for providing me the opportunity to work on this thesis, as well as the tools needed.

Furthermore, I would like to specially thank two individuals that helped me when I was facing some difficulties. I would like to start by thanking Miss Marka Neumann with whom we exchanged a lot of emails and who helped me numerous times. Also, I feel the need to thank mister George Alex Koulieris for the great help he provided.

I would like to thank Professor Costas Balas and Associate Professor Antonios Deligiannakis for their time reading and reviewing this thesis.

I would like to also thank all of you that took part on the experiments that were conducted because you helped a lot with all your observations.

Finally, I would like to thank my family and my friends for their moral support that helped me overcome all the difficulties and gave me the extra push needed in order to continue and complete this thesis.

Abstract

The wide-spread availability of consumer grade virtual reality head mounted displays, has transformed virtual reality to a commodity available for everyday use. Nowadays, nearly everything with a display can be used to immerse the user in a VR world. From Smart phones to game consoles, everything now has VR extensions such as the Samsung Gear VR and the PlayStation VR.

All this constant evolution around the VR world demands constantly better and more detailed head mounted displays. With the increasing use of 4K-8K Ultra High Definition displays and the push towards higher pixel densities for head-mounted displays, the industry is pressured to meet market demands for intensive real-time rendering. Since the current processors cannot deal with the increased demands for excessive resolution on the head mounted displays, new techniques of rendering must be implemented.

The human visual system is often assumed to be perfect despite limitations arising from a variety of different complexities and phenomena. Humans have two distinct vision systems: foveal and peripheral vision. Foveal vision is sharp and detailed, while peripheral vision lacks fidelity. This lack of fidelity in the peripheral vision system is what new techniques of rendering, the so called foveated rendering techniques, are trying to exploit. Perceptually lossless foveated rendering systems and methods, seek to increase rendering performance by lowering image quality in the periphery, while maintaining the user's perception of full HD rendering.

In this thesis, we are trying to gather insights on how beneficial may the adoption of these methods at a commercial level be, by implementing and evaluating our own foveated rendering approach. To do so, we are using a Head Mounted Display unit [the Nvis SX 111] and an eye-tracking device [the Arrington Viewpoint-EyeTracker]. The foveated rendering technique developed in this thesis renders at three different layers of resolution. Apart from the foveal layer, which surrounds the area that the user is looking at and has full HD resolution capabilities, and the peripheral layer, which contains everything that is in the user's peripheral vision and renders at 40% of the full HD resolution, we implement another layer which functions as a transition layer between them. This last layer renders at 60% of the full HD resolution, and it was added so that the user doesn't notice the massive difference in resolution at the border between the other two layers.

In order to have a more accurate picture of the results and the functionality of the algorithm created, we conducted a number of experiments involving 19 students from the institution. The users were asked to enter a virtual world and complete a small game. While the users were immersed in the virtual environment, we were

monitoring the performance of the algorithm. During these experiments, a 57% decrease in the number of pixels shaded was recorded. Most of these pixels belonged to the peripheral layer. This decrease leads to a maximum increase of 18.3% regarding the number of rendered frames per second.

This increase in FPS is the fundamental objective of this dissertation. Since we have achieved such an increase in FPS, it is safe to assume that foveated rendering algorithms are capable of large reductions in rendering cost using the latest technologies.

Table of contents

List of figures

List of tables

Chapter 1 Introduction

1.1 Scope

It is an undeniable fact that Virtual Reality (VR) is soon to become ubiquitous. The widespread availability of consumer grade VR Head Mounted Displays (HMDs) such as the Oculus RiftTM *(Oculus, 2017)* and the new Samsung gear VR *(Samsung Electronics CO, 2017)* transformed Virtual Reality to a commodity available for everyday use. Virtual reality has a wide range of applications which range from gaming and entertainment to medicine, engineering, military training, scientific visualization and business. Nowadays, even smart phones can easily be used in order to immerse the user in a virtual environment such as the galaxy S6, S6 edge, S7 and S7 edge.

Figure 1: Samsung Gear VR with controllers[1]

However, this new 3D environment is way different than the environment used the years before, which is no other than the simple 2D environment such as the environment depicted on the screen of a computer. User interaction in a 3D spatial context introduces constraints due to the multiple degrees of motion freedom, requiring novel interaction metaphors such as “fly” and “zoom”. In addition, traditional input methods such as a keyboard and mouse are hard to manipulate when the user wears a HMD. Using a keyboard and a mouse while immersed in a VR HMD is an erroneous extension of the desktop paradigm to VR, constituting a fundamental challenge that needs to be addressed. These days, the most common input methods for VR apps are hand-tracking or head-tracking hardware, while also eye-tracking hardware is sometimes used but mostly their use has been confined to laboratory experiments. However, nowadays, companies try to

[1] Figure 1 was retrieved from the following site : http://www.samsung.com/global/galaxy/gear-vr/

enhance HMDs with eye-tracking software, like the brand new F0VE *(FOVE,Inc, Late 2017)*, which is the first eye-tracking VR headset and which seems to be a great success. Additionally, Fove is compatible and easy to use with the most common game engines like Unity *(Unity Technologies, 2017)* and Unreal Engine and more.

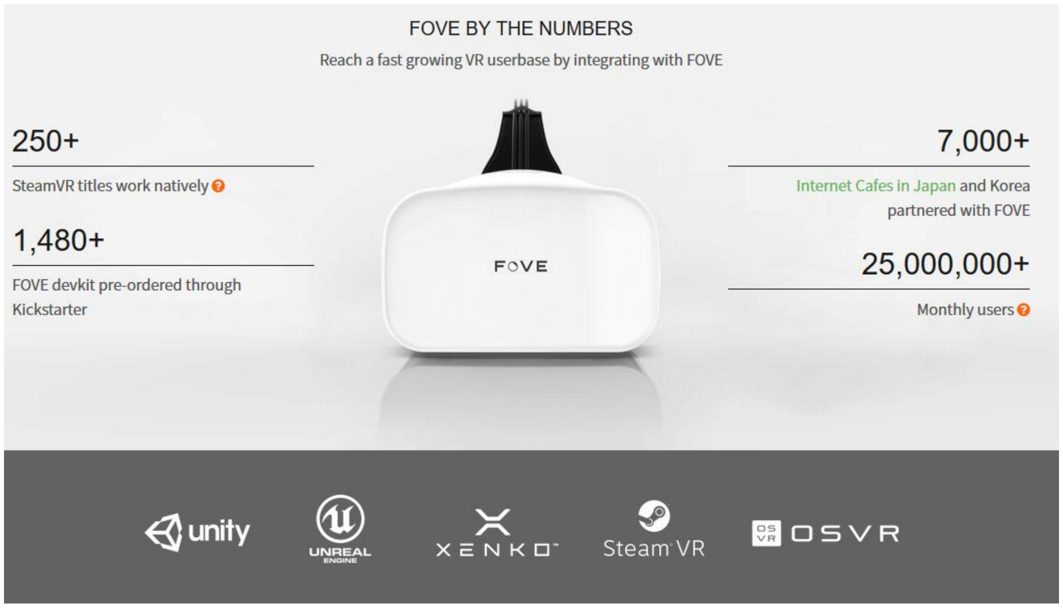

FOVE BY THE NUMBERS
Reach a fast growing VR userbase by integrating with FOVE
250+
SteamVR titles work natively
1,480+
FOVE devkit pre-ordered through Kickstarter
7,000+
Internet Cafes in Japan and Korea partnered with FOVE
25,000,000+
Monthly users
FOVE
unity
UNREAL ENGINE
XENKO
Steam VR
OSVR

Figure 2: FOVE numbers[2]

Eye trackers have existed for a number of years, but their use has largely been confined to laboratory experiments. The equipment is gradually becoming sufficiently robust and inexpensive to consider use in real user-computer interfaces. For example Fove starts at 599 USD and Oculus rift at 399 USD while Samsung Gear can be found for only 100 Euros. However, one needs to attach a Smartphone on the Samsung Gear in order to experience VR.

Even though eye tracking software and eye tracking technologies have existed for many years, their use has been limited, and for a number of years these technologies had even been abandoned. The reason behind this abundance is that they needed computational resources that some years ago were unachievable from the current CPUs – GPUs. Recently, CPUs have experienced spectacular growth that has unlocked new processing speeds that covered the numerous needs of eye-tracking software. As a result, new methods of rendering are starting to be developed. By using the information generated from the eye-tracking software, it is possible to create an algorithm that exploits the human visual system's

[2] Figure 2 was retrieved from the following site : https://www.getfove.com/

imperfection in order to provide the user with a visual result rendered based on his focal point.

The human visual system is often assumed to be perfect without limitations. This assumption leads to the idea that a single render will be fully appreciated at any single point in time (Cosker & Swafford, 2015). The truth is that humans have two distinct vision systems, which are called, foveal and peripheral vision. Foveal vision is sharp and detailed and the only part of the retina that permits 100% visual acuity, while peripheral vision lacks fidelity. (Patney, et al., 2016)

In this thesis, we present a rendering technique that takes advantage of the limited peripheral vision system of the human eye in order to achieve higher CPU performance in terms of frames per second rates. This technique is called foveated rendering. The idea behind foveated rendering is that we can create an algorithm that exploits the user's perception of a fully highly rendered scene. Since the human eye is unable to understand and fully appreciate a single full HD rendering, it would be beneficial, for the CPU and the GPU, to have a way to render critical parts of the HMD's screen in high resolution, while keeping other parts in lower resolution. In different words, we determine the critical parts of the scene based on the user's focal point in order to create a personalized rendering result for every frame. To determine the user's focal point a HMD and eye-tracking software is needed. In this thesis, we are using the Nvis SX111 HMD combined with the Arrington's eye-tracking software.

We exploit the falloff of acuity in the visual periphery to accelerate graphics computation on a HMD display (1280 x 1024). Our method tracks the user's gaze point and renders three image layers around it at progressively higher angular size but lower sampling rate. The three layers are then magnified to display resolution and smoothly composited. The result looks like a full-resolution image but reduces the number of pixels shaded by a factor of 2,5 in our first approach while in our second and more aggressive approach this factor increased to 4,5. This means that instead of sampling 1.759.232pixels, with the implementation of the foveated rendering algorithm, we only shaded 755.419 in our 1st approach and only 403.569 in the 2nd. These results were achieved with a HMD with a display resolution of just 1280 x 1024px. Image if we were to use the new Fove HMD with a display resolution of 2550 x 1440px, where in order to produce a full HD result it would be necessary to shade 492.851.250pixels instead of just 1.759.232. Moreover, our algorithm led to a 19% performance boost.

Nowadays, with the increasing use of 4K – 8K Ultra High Definition (UHD) displays and the push towards higher pixel densities for head-mounted displays (HMDs) alongside with the demands for intensive real-time rendering, the adoption of perceptually lossless foveated rendering methods is a must. (Cosker & Swafford, 2015)

Perceptually lossless foveated rendering methods exploit human perception by selectively rendering at different quality levels, based on eye gaze, at a lower computational cost, while still maintaining the user's perception of a full quality render (Swafford, Iglesias-Guitian, Koniaris, & Moon, 2016). In other words perceptually lossless foveated rendering systems seek to increase rendering performance by lowering image quality in the periphery while making sure that degraded renders are indistinguishable from their non-degraded counterparts. Providing high-quality image synthesis on high resolution displays in real-time is an ultimate goal of computer graphics. However, it remains a challenging problem even with full utilization of GPU hardware, as rendering operations are expected to perform in increasingly shorter time-frames (90 Hz and even higher).

This thesis aims to develop an algorithm that ideally works without being noticed by the naked human eye. The general idea is that the user gets immersed in a virtual reality environment in order to play a game. While playing, the algorithm is running in the background and constantly (in real time) checks the position of the user's eye. By doing so it calculates were the user is looking at 60 times per second. This information is very important. We are using this information to provide the user with a rendering result which has a full high definition rendering layer around his focal point. This "perfect" rendering area is the first layer of rendering, which is also the smallest. Additionally, there are two more layers with lower resolution (less samples / pixel) than the first layer and which are a lot bigger than it (number of pixels). That means that this algorithm generates an image which is rendered in more than one rendering levels by changing the number of samples per pixel depending on the layer. Of course there are other ways to implement a foveated rendering algorithm. For example, instead of changing the number of samples per pixel for every layer like we did in out approach, others use sampling maps (Pohl, Zhang, & Bulling, 2016), others change the probability that a pixel has to be rendered depending on its position on the monitor (Roth, Weier, Hinkenjann, Li, & Slusallek, 2016) etc. Furthermore, the number of layers can vary from on algorithm to another. For example, (Swafford, Cosker, & Mitchell, 2015) render their scene in two different layers of resolution while in our approach and the technique developed by (Guenter B. , Finch, Drucker, Tan, & Snyder, 2012) we are rendering three layers of resolution. The minimum number of layers of course is two, one for the foveal and one for the periphery, however, there is not really an upper limit to it. The more layers one uses the harder it is for the user to understand the loss of resolution but the harder it is for GPU to create the final result. So there is a trade-off between the efficiency of the algorithm and the user's perception. We will further discuss about the various ways of implementing foveated rendering algorithms, in the next chapter.

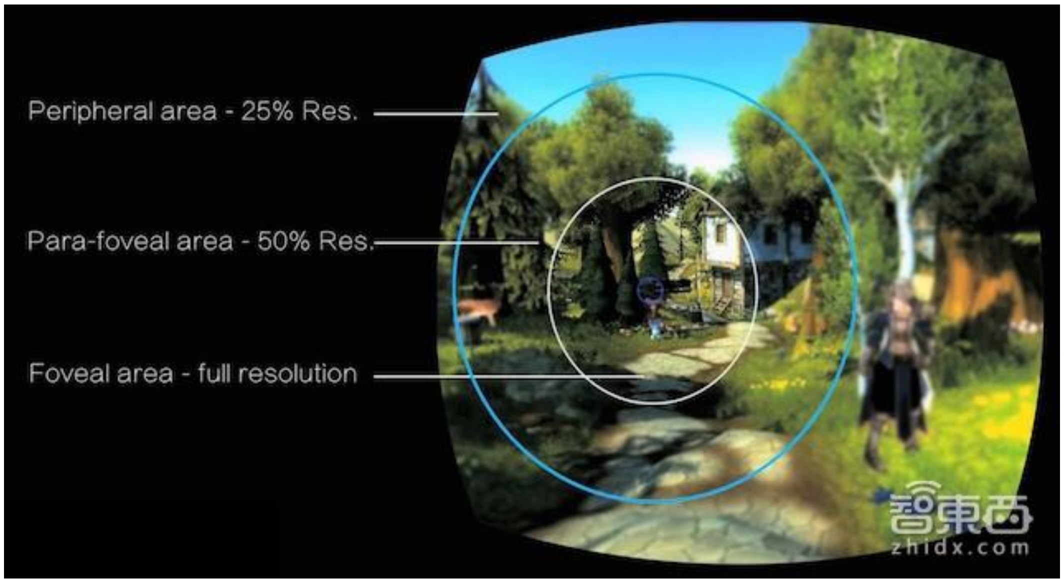

Peripheral area - 25% Res.
Para-foveal area - 50% Res.
Foveal area - full resolution
智東西
zhidx.com

Figure 3 : Example of Foveated Rendering Algorithm[3]

To test the foveated rendering algorithm created in this thesis, a user experiment was conducted, with 19 participants. For the needs of this experiment, a game was implemented. The user enters a specific virtual environment in which he takes the role of a gondolier. Right when the game starts, the user meets his first customer approaching him slowing from his right side. When the customer boards the gondola, he asks the user-gondolier to take him to one of the two possible destinations. The destination is chosen randomly every time. When the customer informs the user/gondolier of "his" choice golden coins pop–up on the map in order to help the user locate the correct destination area. The user simply has to take the customer to the correct area. Meanwhile, the user can freely enjoy the ride and the view. The ride takes about 2 minutes to be completed and for the whole duration we check the number of frames per second that are rendered. At the end of every phase of this experiment we save the highest number of FPs, the lowest number of FPs as well as the average number of FPs. Every user has to complete the same ride four times because we want to test the FPS results on four different sets of parameters.

Set	Foveal	Central Cycle	Periphery
1	102^{o}	0 ^{o}	0 ^{o}
2	5 ^{o}	10 ^{o}	102 ^{o}
3	10 ^{o}	20 ^{o}	102 ^{o}
4	15 ^{o}	30 ^{o}	102 ^{o}

[3] Figure 3 was retrieved from the following site : https://baijia.baidu.com/s?old_id=762069

The first set of parameters indicates that the foveal cycle of the foveated rendering technique that is going to be applied will be 102 ^{o}. The Nvis SX 111 HMD has a field of view of 102 ^{o} that means that the whole screen will be covered by the foveal cycle, which means that the result won't be a foveated rendered result, but a full high resolution one. On the second set of parameters the algorithm will create a result where the Foveal cycle will cover 5 ^{o}, the central cycle (which is between the foveal and the periphery) will cover 10 ^{o} and finally the periphery cycle will cover of course 102 ^{o}. Sets 3 and 4 are created accordingly.

The algorithm implemented achieves to reduce the number of pixels rendered by 57 per cent. This reduction leads to a maximum increase of 18.3% and average increase of 7.4% in terms of FPS.

1.2 Thesis structure

In the following subchapter the framework that was followed is thoroughly explained.

The 1st chapter soon comes to an end and the 2nd chapter is about to begin. The 2nd chapter has an introductory and educative purpose, in order to introduce the reader to the scientific fields that this thesis is seated and relevant. Scientific fields such as Virtual Environment, Virtual Reality, the human eye's anatomy, eye and head-tracking software as well as the most popular game engines are discussed. The second chapter also includes a subchapter regarding the foveated rendering techniques and the methods commonly used for its implementation. It also contains a comparison of the technique implemented in this thesis with these techniques.

The third chapter presents the software that was used to develop the application to be utilized for the experiments. A detailed description of the Unity 3D software platform is provided including the most important components of it, mainly focusing on its capabilities of programming geometry behaviors, UIs and integration for multiple hardware components. Moreover, the software use for the eye tracking and the head tracking devices is also presented.

The User Interface (UI) as presented to the participant during the user trials is described in chapter number 4. As user interface we refer to the main menu scene created in order for the user to be able to choose his next action, and the 3D virtual scene that the user is immersed to. After loading the scene the user enters the virtual environment in which the user takes the role of a gondolier. Right when the game starts, the user meets his first customer approaching him slowly from his right side. When the customer boards the gondola, a message appears on the screen notifying the user about the customer's desired destination. The user has to transfer the customer to the correct area. Everything that the participant is able to see and interact with within both the virtual scene and the main menu scene is explained in detail. Furthermore, the way that the objects used were modeled and animated is contained in this chapter.

The 5th chapter focuses on the implementation process. In this chapter, we analyze the ways in which we meet the specifications of our hardware. In particular, we will explain how we achieve stereoscopy and how we generate the partially overlapping effect needed. This chapter also includes a detailed analysis of the communication between both the head tracker and the eye tracker with our application. Moreover, the foveated rendering technique that we created and the way we process the information received from the trackers are both explained in detail.

The 6th chapter describes the user trials that were made in order to check the accuracy of our eye-tracking software. More importantly, during these experimental procedures, we gather data regarding the FPS rates of the application, which will be discussed in the final chapter. Also, the procedure that was followed during these trials is described.

Finally, a chapter is dedicated to a detailed discussion on the results that have emerged from our user trials. This chapter will contain not only the quantitative results but also the way that they were calculated and how they differ from the best to the worst case scenario.

Chapter 2 Background

2.1 Virtual reality

The definition of the term “virtual reality” comes, naturally, from the definitions for both ‘virtual’ and ‘reality’. The definition of ‘virtual’ is near and reality is what we experience as human beings. So the term ‘virtual reality’ basically means ‘near-reality’. This could, of course, mean anything but it usually refers to a specific type of reality emulation.

We humans, we know the world through our senses and perception systems. In school we all learned that we have five senses: taste, touch, smell, sight and hearing. These are however only our most obvious sense organs. The truth is that humans have many more senses than this, such as a sense of balance for example. These other sensory inputs, plus some special processing of sensory information by our brains ensures that we have a rich flow of information from the environment to our minds.

Everything that we know about our reality comes by way of our senses. In other words, our entire experience of reality is simply a combination of sensory information and our brains sense-making mechanisms for that information. It stands to reason then, that if you can present your senses with made-up information, your perception of reality would also change in response to it. You would be presented with a version of reality that isn’t really there, but from your perspective it would be perceived as real. Something we would refer to as a virtual reality.

So, in summary, virtual reality entails presenting our senses with a computer generated virtual environment that we can explore in some fashion.

Virtual reality is the term used to describe a three-dimensional, computer generated environment which can be explored and interacted with by a person. That person becomes part of this virtual world or is immersed within this environment and whilst there, is able to manipulate objects or perform a series of actions.

2.1.1 History of virtual reality

Figure 4: Battle of Borodino, 181. The beginning of VR[4]

Early 19th Century – Panoramic paintings

By focusing more strictly on the scope of virtual reality as a means of creating the illusion that we are present somewhere we are not, then the earliest attempt at virtual reality is surely the 360-degree murals (or panoramic paintings) from the nineteenth century. These paintings were intended to fill the viewer's entire field of vision, making them feel present at some historical event or scene.

1838 – Stereoscopic photos & viewers

In 1838 Charles Wheatstone's research demonstrated that the brain processes the different two-dimensional images from each eye into a single object of three dimensions. Viewing two side by side stereoscopic images or photos through a stereoscope gave the user a sense of depth and immersion. The later development of the popular View-Master stereoscope (patented 1939), was used for "virtual tourism". The design principles of the Stereoscope is used today for the popular Google Cardboard and low budget VR head mounted displays for mobile phones.

1929 – Link Trainer the First Flight Simulator

In 1929 Edward Link created the "Link trainer" (patented 1931) probably the first example of a commercial flight simulator, which was entirely electromechanical. It was controlled by motors that linked to the rudder and steering column to modify the pitch and roll. A small motor-driven device mimicked turbulence and disturbances. Such was the need for safer ways to train pilots that the US military bought six of these devices for $3500. In 2015 money this was just shy of $50 000. During World War II over 10,000 "blue box" Link Trainers were used by over 500,000 pilots for initial training and improving their skills.

[4] Figure 4 was retrieved from the following site : https://www.vrs.org.uk/virtual-reality/history.html

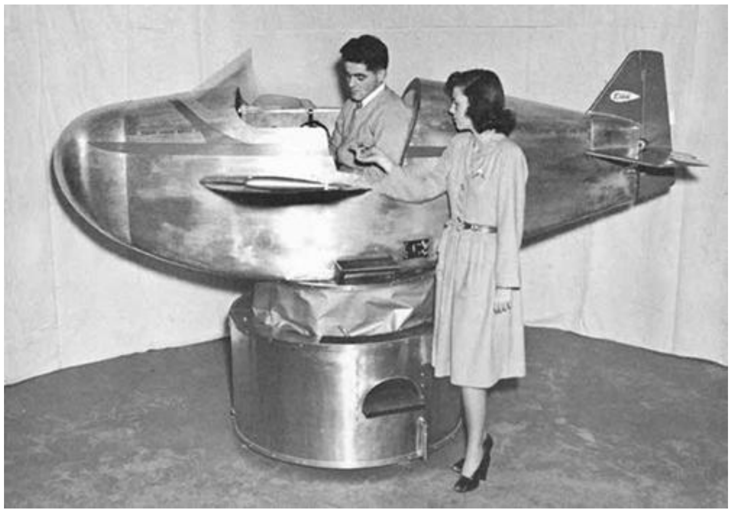


Figure 5: Link Trainer[5]

1930s – Science fiction story predicted VR

In the 1930s a story by science fiction writer Stanley G. Weinbaum (Pygmalion's Spectacles) contains the idea of a pair of goggles that let the wearer experience a fictional world through holographics, smell, taste and touch. In hindsight the experience Weinbaum describes for those wearing the goggles are uncannily like the modern and emerging experience of virtual reality, making him a true visionary of the field.

1950s – Morton Heilig's Sensorama

In the mid 1950s cinematographer Morton Heilig developed the Sensorama (patented 1962) which was an arcade-style theatre cabinet that would stimulate all the senses, not just sight and sound. It featured stereo speakers, a stereoscopic 3D display, fans, smell generators and a vibrating chair. The Sensorama was intended to fully immerse the individual in the film. He also created six short films for his invention all of which he shot, produced and edited himself. The Sensorama films were titled, Motorcycle, Belly Dancer, Dune Buggy, helicopter, A date with Sabina and I'm a coca cola bottle!

[5] Figure 5 was retrieved from Wikipedia

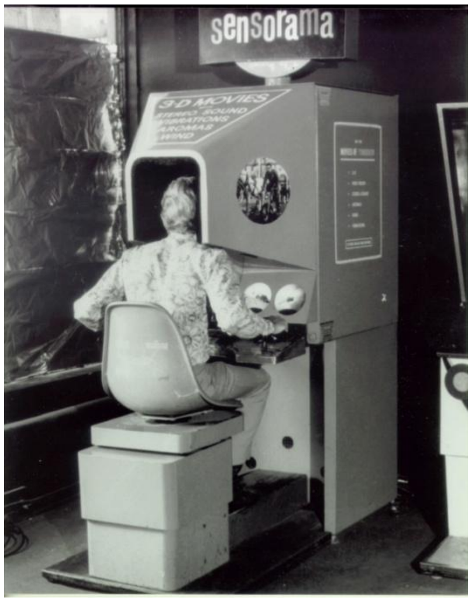


Figure 6: Sensorama[6]

1960 – The first VR Head Mounted Display

Morton Heilig's next invention was the Telesphere Mask (patented 1960) and was the first example of a head-mounted display (HMD), albeit for the non-interactive film medium without any motion tracking. The headset provided stereoscopic 3D and wide vision with stereo sound.

1961 Headsight – First motion tracking HMD

In 1961, two Philco Corporation engineers (Comeau & Bryan) developed the first precursor to the HMD as we know it today – the Headsight. It incorporated a video screen for each eye and a magnetic motion tracking system, which was linked to a closed circuit camera. The Headsight was not actually developed for virtual reality applications (the term didn't exist then), but to allow for immersive remote viewing of dangerous situations by the military. Head movements would move a remote camera, allowing the user to naturally look around the environment. Headsight was the first step in the evolution of the VR head mounted display but it lacked the integration of computer and image generation.

1965 – The Ultimate display by Ivan Sutherland

Ivan Sutherland described the "Ultimate Display" concept that could simulate reality to the point where one could not tell the difference from actual reality. His concept included:

[6] Figure 6 was retrieved from the following site: https://www.vrs.org.uk/virtual-reality/history.html

- A virtual world viewed through a HMD and appeared realistic through augmented 3D sound and tactile feedback.
- Computer hardware to create the virtual word and maintain it in real time.
- The ability of users to interact with objects in the virtual world in a realistic way

"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

1968 – Sword of Damocles

In 1968 Ivan Sutherland and his student Bob Sproull created the first VR / AR head mounted display (Sword of Damocles) that was connected to a computer and not a camera. It was a large and scary looking contraption that was too heavy for any user to comfortably wear and was suspended from the ceiling (hence its name). The user would also need to be strapped into the device. The computer generated graphics were very primitive wireframe rooms and objects.

1969 – Artificial Reality

In 1969 Myron Kruegere a virtual reality computer artist developed a series of experiences which he termed "artificial reality" in which he developed computer-generated environments that responded to the people in it. The projects named GLOWFLOW, METAPLAY, and PSYCHIC SPACE were progressions in his research which ultimately led to the development of VIDEOPLACE technology. This technology enabled people to communicate with each other in a responsive computer generated environment despite being miles apart.

1987 – Virtual reality the name was born

Even after all of this development in virtual reality, there still wasn't an all-encompassing term to describe the field. This all changed in 1987 when Jaron Lanier, founder of the visual programming lab (VPL), coined (or according to some popularised) the term "virtual reality". The research area now had a name. Through his company VPL research Jaron developed a range of virtual reality gear including the Dataglove (along with Tom Zimmerman) and the EyePhone head mounted display. They were the first company to sell Virtual Reality goggles (EyePhone 1 $9400; EyePhone HRX $49,000) and gloves ($9000). A major development in the area of virtual reality haptics.

1991 – Virtuality Group Arcade Machines

We began to see virtual reality devices to which the public had access, although household ownership of cutting edge virtual reality was still far out of reach. The Virtuality Group launched a range of arcade games and machines. Players would wear a set of VR goggles and play on gaming machines with realtime (less than 50ms latency) immersive stereoscopic 3D visuals. Some units were also networked together for a multi-player gaming experience.

1992 – The Lawnmower Man

The Lawnmower Man movie introduced the concept of virtual reality to a wider audience. It was in part based on the founder of Virtual Reality Jaron Lanier and his early laboratory days. Jaron was played by Pierce Brosnan, a scientist who used virtual reality therapy on a mentally disabled patient. Real virtual reality equipment from VPL research labs was used in the film and the director Brett Leonard, admited to drawing inspiration from companies like VPL.

1993 – SEGA announce new VR glasses

Sega announced the Sega VR headset for the Sega Genesis console in 1993 at the Consumer Electronics Show in 1993. The wrap-around protoype glasses had head tracking, stereo sound and LCD screens in the visor. Sega fully intended to release the product at a price point of about $200 at the time, or about $322 in 2015 money. However, technical development difficulties meant that the device would forever remain in the prototype phase despite having developed 4 games for this product. This was a huge flop for Sega.

1995 – Nintendo Virtual Boy

The Nintendo Virtual Boy (originally known as VR-32) was a 3D gaming console that was hyped to be the first ever portable console that could display true 3D graphics. It was first released in Japan and North America at a price of $180 but it was a commercial failure despite price drops. The reported reasons for this failure were a lack of colour in graphics (games were in red and black), there was a lack of software support and it was difficult to use the console in a comfortable position. The following year they discontinued its production and sale.

1999 – The Matrix

In 1999 the Wachowski siblings' film *The Matrix* hits theatres. The film features characters that are living in a fully simulated world, with many completely unaware that they do not live in the real world. Although some previous films had dabbled in depicting virtual reality, such as Tron in 1982 and *Lawnmower Man* in 1992, The Matrix has a major cultural impact and brought the topic of simulated reality into the mainstream.

Virtual reality in the 21st century

The first fifteen years of the 21st century has seen major, rapid advancement in the development of virtual reality. Computer technologies, especially small and powerful mobile technologies, have exploded while prices are constantly driven down. The rise of smart phones with high-density displays and 3D graphics capabilities has enabled a generation of lightweight and practical virtual reality devices. The video game industry has continued to drive the development of consumer virtual reality unabated. Depth sensing cameras sensor suites, motion controllers and natural human interfaces are already a part of daily human computing tasks.

Recently companies like Google have released interim virtual reality products such as the Google Cardboard, a DIY headset that uses a smart phone to drive it. Companies like Samsung have taken this concept further with products such as the Galaxy Gear, which is mass produced and contains “smart” features such as gesture control.

Developer versions of final consumer products have also been available for a few years, so there has been a steady stream of software projects creating content for the immanent market entrance of modern virtual reality.

2.2 How is virtual reality achieved?

2.2.1 Technology

Today virtual reality is usually implemented using computer technology. There are a range of systems that are used for this purpose, such as Virtual glasses or goggles, data gloves, HMDs, data suits, workbenches, joysticks, keyboards, but also haptic devices which enable the sense of touch when manipulating an object in the virtual environment (VE). These are used to actually stimulate our senses together in order to create the illusion of reality.

2.2.2 Immersion

This is more difficult than it sounds, since our senses and brains are evolved to provide us with a finely synchronized and mediated experience. If anything is even a little off we can usually tell.

Immersion into virtual reality is a perception of being physically present in a non-physical world. The perception is created by surrounding the user of the VR system in images, sound or other stimuli that provide an engrossing total environment.

To create a sense of full immersion, the 5 senses (sight, sound, touch, smell, taste) must perceive the digital environment to be physically real. Immersive technology can perceptually fool the senses through:

- Panoramic 3D displays (visual)

- Surround sound acoustics (auditory)
- Haptics and force feedback (tactile)
- Smell replication (olfactory)
- Taste replication (gustation)

The level of immersion provided by a system is determined by the following “constraints”: the ability of the technology, the ability of the senses, and perception, which is made through interpretation. Two are the main components of virtual reality immersion, depth of and breadth of information. The depth of information is the amount and quality of data the user is given during interacting with the virtual environment. This can refer to the display resolution, the complexity of graphics, and the clarity of audio output. Breadth of information is the number of sensory dimensions that are presented. To have a wide breadth of information, the virtual reality immersion must stimulate all the senses. Virtual reality immersion experiences prioritize audio and visual components over the other sensory factors.

The most out of immersion is gained when the user explores life-size VR environments and thus forgets about his real world scenario, forgets his present identity, situation and life and immerses him in a world of imagination, adventure and exploration. He gets more focused about his newly created identity inside the VR world.

2.2.3 Immersive environment

An immersive digital environment is an artificial, interactive, computer-created scene or "world" within which a user can immerse himself.

Immersive digital environments could be thought of as synonymous with virtual reality, but without the implication that actual "reality" is being simulated. An immersive digital environment could be a model of reality, but it could also be a complete fantasy user interface or abstraction, as long as the user of the environment is immersed within it. The definition of immersion is wide and variable, but here it is assumed to mean simply that the user feels like they are part of the simulated "universe".

2.3 Equipment

2.3.1 The nvis SX111

Since this project was based on a HMD we will show emphasis on that VR input device. As an input device or sensor we refer to those devices that capture the participant’s actions and send this information to the computer/system which is in charge of the interactive simulation. An input device is considered as a virtual input device or VR gear when they use the paradigm of the implicit interaction or they give 3D input to the system.

A typical HMD has one (monocular HMD) or two (binocular HMD) small displays, with lenses and semi-transparent mirrors embedded in eyeglasses, a visor, or a helmet. Mostly all HMDs consist of a screen for individual eye and that is what creates a sense any images the user looks at has some depth. The display units are miniaturized and may include cathode ray tubes (CRT), liquid crystal displays (LCDs), liquid crystal on silicon (LCos), or organic light-emitting diodes (OLED). Some vendors employ multiple micro-displays to increase total resolution and field of view. This is included to make sure both audio and video output is received. Most HMDs are attached to the CPU of the VR systems through cables but also there are wireless systems. However wireless systems do not have the eligibility to avoid lag.

The NVIS SX111 is the Head Mounted Display (HMD) used in this thesis. One of its significant characteristics is the wide field-of-view of a total 102^{o} for both eyes. A 3-degree of freedom (rotational) head-tracker was attached to this HMD acquiring the user’s head rotational direction. The HMD makes use of partial overlap stereoscopic method in order to produce stereoscopic vision.

Figure 7: The NVisorTM SX111 HMD[7]

2.3.2 State of the art HMDs

FOVE is the first virtual reality headset that utilizes eye tracking. It was created by a Tokyo-based startup founded by Yuka Kojima (CEO) and Lochiainn Wilson (CTO) and was announced in 2014. FOVE's technology uses infrared to track eye movements with accuracy (less than 1 degree) and low latency. The sensors within the device track the user's pupils. It allows the user to target and interact with objects by making eye contact with them.

[7] Figure 7 was retrieved from the following site : https://est-kl.com/manufacturer/nvis/nvisor-sx111.html

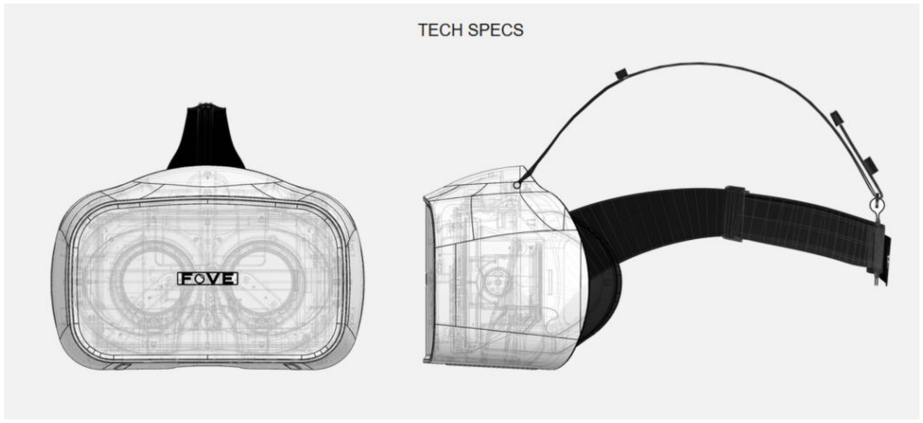
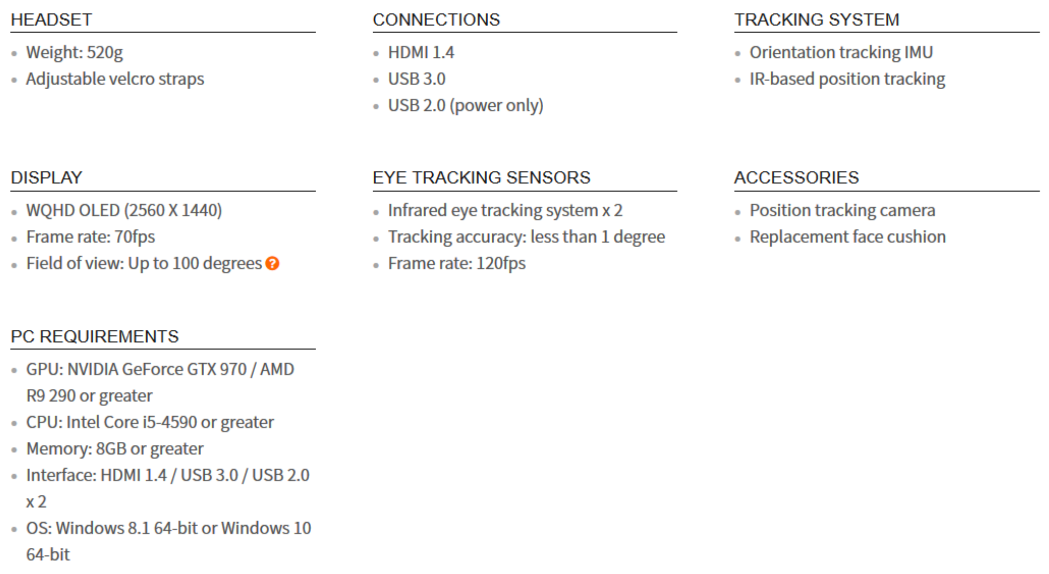


Figure 8: F0VE [8]



HEADSET
- Weight: 520g
- Adjustable velcro straps

CONNECTIONS
- HDMI 1.4
- USB 3.0
- USB 2.0 (power only)

TRACKING SYSTEM
- Orientation tracking IMU
- IR-based position tracking

DISPLAY
- WQHD OLED (2560 X 1440)
- Frame rate: 70fps
- Field of view: Up to 100 degrees

EYE TRACKING SENSORS
- Infrared eye tracking system x 2
- Tracking accuracy: less than 1 degree
- Frame rate: 120fps

ACCESSORIES
- Position tracking camera
- Replacement face cushion

PC REQUIREMENTS
- GPU: NVIDIA GeForce GTX 970 / AMD R9 290 or greater
- CPU: Intel Core i5-4590 or greater
- Memory: 8GB or greater
- Interface: HDMI 1.4 / USB 3.0 / USB 2.0 x 2
- OS: Windows 8.1 64-bit or Windows 10 64-bit

Figure 9: F0VE Specs[9]

2.3.3 FOVE vs. HTC Vive, Oculus Rift and PlayStationVR

Compared to the 2160×1200 displays of the HTC Vive and Oculus Rift, the 2560×1440 display of the FOVE 0 has 42% more pixels, on par with Samsung's Gear VR headset. Unfortunately the display only has a refresh rate of 70Hz, which comes in significantly below the 90Hz of the Rift and Vive (not to mention the 120Hz of PlayStation VR), and slightly lower than the 75Hz of Oculus' Rift DK2 development kit. A lower refresh rate means more latency and less smooth motion inside the headset. However, with the success of Gear VR, even with its

[8] Figure 8 and
[9] Figure 9 were retrieved from the following site : https://www.getfove.com

60Hz refresh rate, 70Hz ought to work fine as a starting point for FOVE 0, though it’s an area we expect to see improve in future versions of the headset.

Figure 10: F0VE[10]

Figure 11: HTC Vive - OCULUS Rift - PlayStasion VR[11]

2.3.4 SMI’s 250Hz eye tracking

At the CES 2016, SensoMotoric Instruments (SMI) demoed a new 250 Hz eye tracking system and a working foveated rendering solution. It resulted from a partnership with camera sensor manufacturer Omnivision who provided the camera hardware for the new system.

[10] Figure 10 was retrieved from the following site: https://www.getfove.com
[11] Figure 11 was retrieved from the following site: Google.com

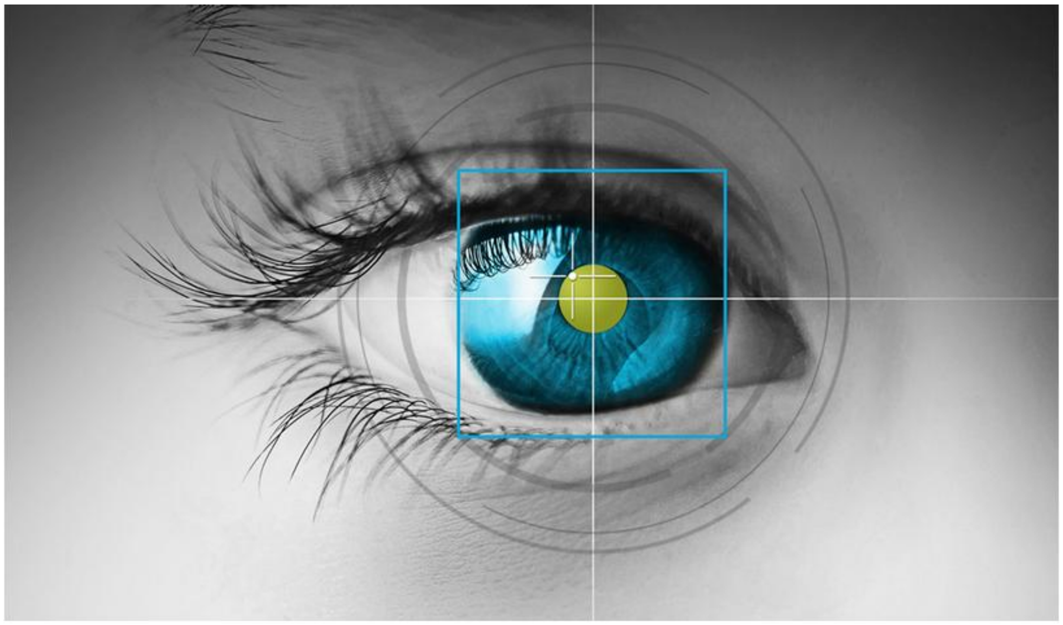


Figure 12: SMI Logo[12]

"Getting over the 240Hz mark was important," says Villwock [13]*, "it allows us to track the saccadic motion of the eye."*

Saccadic motion being the unnoticed and involuntary motion of your eye as it moves between planes of focus.

2.4 The human eye

"The eye is the window of the soul... The eye is the window of the human body through which it feels its way and enjoys the beauty of the world." (Da Vinci, 1452-1519)

It is obvious that our visual system is an essential human factor to be taken into account when designing VR hardware and software. The Eye has been called the most complex organ in our body. It accommodates to changing lighting conditions and focuses light rays originating from various distances from the eye. Light is converted to impulses and conveyed to the brain where an image is perceived.

[12] Figure 12 was retrieved from the following site: https://uploadvr.com/smi-hands-on-250hz-eye-tracking/

[13] Christian Villwock SensoMotoric Instruments' director of OEM technology

2.4.1 Anatomy of the Eye

In order to understand how eye tracking works, we must understand the human eye’s anatomy, its components and operation. The eye is made up of three coats, enclosing three transparent structures. The outermost layer, known as the fibrous tunic, is composed of the cornea and sclera. The middle layer, known as the vascular tunic or uvea, consists of the choroid, ciliary body, and iris. The innermost is the retina, which gets its circulation from the vessels of the choroid as well as the retinal vessels, which can be seen in an ophthalmoscope.

Within these coats are the aqueous humor, the vitreous body, and the flexible lens. The aqueous humor is a clear fluid that is contained in two areas: the anterior chamber between the cornea and the iris, and the posterior chamber between the iris and the lens. The lens is suspended to the ciliary body by the suspensory ligament (zonule), made up of fine transparent fibers. The vitreous body is a clear jelly that is much larger than the aqueous humor present behind the lens, and the rest is bordered by the sclera, zonule, and lens. They are connected via the pupil.

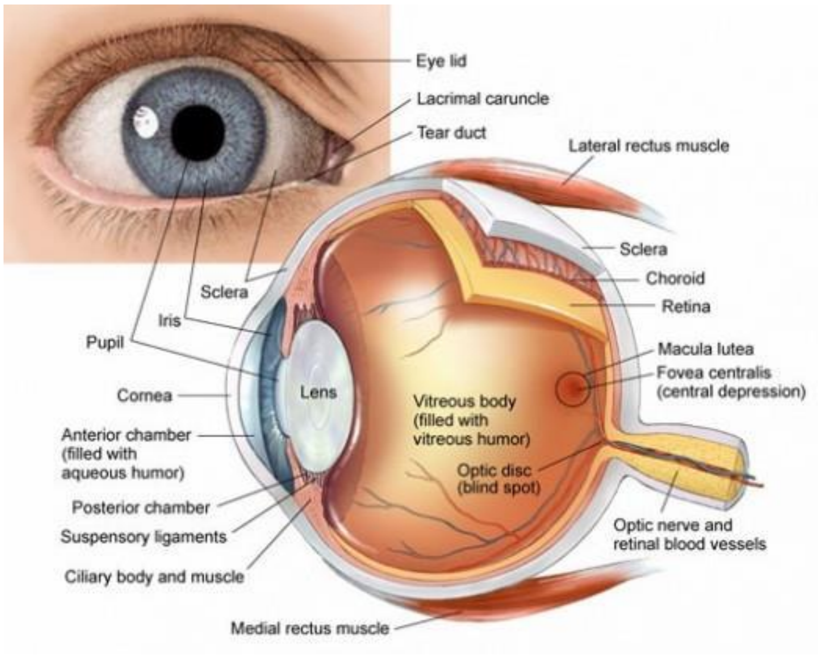


Figure 13: The human eye[14]

The cornea is the transparent, outer "window" and primary focusing element of the eye. The outer layer of the cornea is known as epithelium. Its main job is to protect the eye. The epithelium is made up of transparent cells that have the ability to regenerate quickly. The inner layers of the cornea are also made up of transparent tissue, which allows light to pass.

[14] Figure 13 was retrieved from the search engine Google

The pupil is the dark opening in the center of the colored iris that controls how much light enters the eye. The colored iris functions like the iris of a camera, opening and closing, to control the amount of light entering through the pupil.

The part of the eye immediately behind the iris that performs delicate focusing of light rays upon the retina, is called the lens. In persons under 40, the lens is soft and pliable, allowing for fine focusing from a wide variety of distances. For individuals over 40, the lens begins to become less pliable, making focusing upon objects near to the eye more difficult. This is known as presbyopia.

Retina is the membrane lining the back of the eye that contains photoreceptor cells. These photoreceptor nerve cells react to the presence and intensity of light by sending an impulse to the brain via the optic nerve. In the brain, the multitude of nerve impulses received from the photoreceptor cells in the retina are assimilated into an image.

The fovea is the most central part of the retina. This area is responsible for the clearest vision with sharpest colors and details.

2.4.2 Vision: the eye's functionality

The human eye works much like a digital camera. Light rays reflected off an object enter the eye through a transparent layer of tissue known as the cornea. As the eye's main focusing element, the cornea takes widely diverging rays of light and bens them through the pupil, the dark, round opening in the center of the colored iris.

The lens of the eye is located immediately behind the pupil. The purpose of the lens is to make the delicate adjustments in the path of the light rays in order to bring the light into focus upon the retina, the membrane containing photoreceptor nerve cells that lines the inside back wall of the eye. The central part of the retina is named the macula and the most central part of the macula is the fovea. The fovea is the area at which we have the sharpest vision. When looking directly at an object, the light form it is projected onto the fovea. Although light is admitted through the pupil it is attenuated by the iris, which controls the level of light falling on the retina. The lens of the eye changes its shape to focus the light it passes through towards the retina. The outer, white part, of the eye is the sclera. The photoreceptor nerve cells of the retina change light rays into electrical impulses and send them through the optic nerve to the brain where an image is perceived.

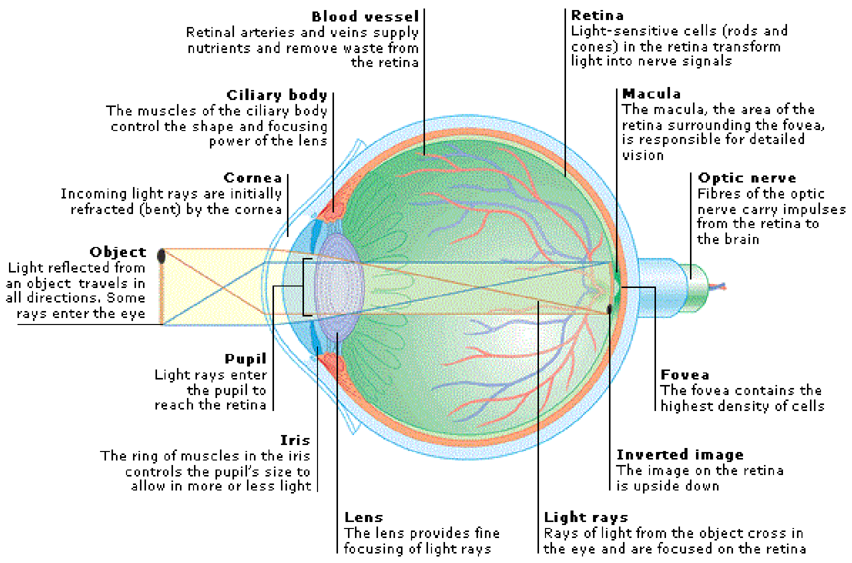

Blood vessel
Retinal arteries and veins supply nutrients and remove waste from the retina
Retina
Light-sensitive cells (rods and cones) in the retina transform light into nerve signals
Ciliary body
The muscles of the ciliary body control the shape and focusing power of the lens
Macula
The macula, the area of the retina surrounding the fovea, is responsible for detailed vision
Cornea
Incoming light rays are initially refracted (bent) by the cornea
Optic nerve
Fibres of the optic nerve carry impulses from the retina to the brain
Object
Light reflected from an object travels in all directions. Some rays enter the eye
Pupil
Light rays enter the pupil to reach the retina
Fovea
The fovea contains the highest density of cells
Iris
The ring of muscles in the iris controls the pupil's size to allow in more or less light
Inverted image
The image on the retina is upside down
Lens
The lens provides fine focusing of light rays
Light rays
Rays of light from the object cross in the eye and are focused on the retina

Figure 14: Parts of the human eye

The entire eye is rotated by six muscles allowing it to move (up, down, side to side, rotate) with great flexibility. An inner muscle, the Medial Rectus, an outer, the Lateral Rectus, an upper, the Superior Rectus, a lower, the Inferior Rectus, an upper and running obliquely, the Superior Oblique, and a lower and running obliquely , the Inferior Oblique.

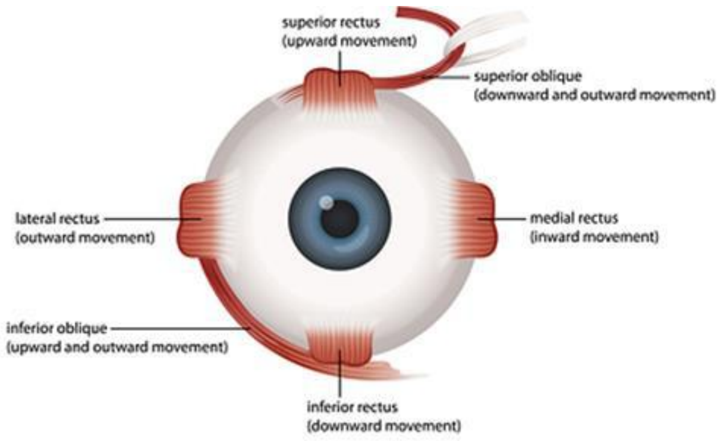


Figure 15: Muscles of the human eye[15]

2.4.3 Vision types

There are two types of vision, binocular and monocular vision. Binocular is the vision in which both eyes are used together and on the other hand, monocular vision is the vision in which each eye is used separately.

[15] Both figures 14 & 15 were retrieved from the search engine Google

Humans have a maximum horizontal FoV of approximately 180-190 degrees with both eyes, 120 degrees that make the binocular FoV (seen by both eyes) flanked by two monocular fields (seen by only one eye) of 40 degrees. One basic advantage of binocular vision is that it gives stereopsis; in which binocular disparity (or parallax) provided by the two eyes' different positions on the head precise depth perception. As for the vertical FoV, it is approximately 120-130 degrees.

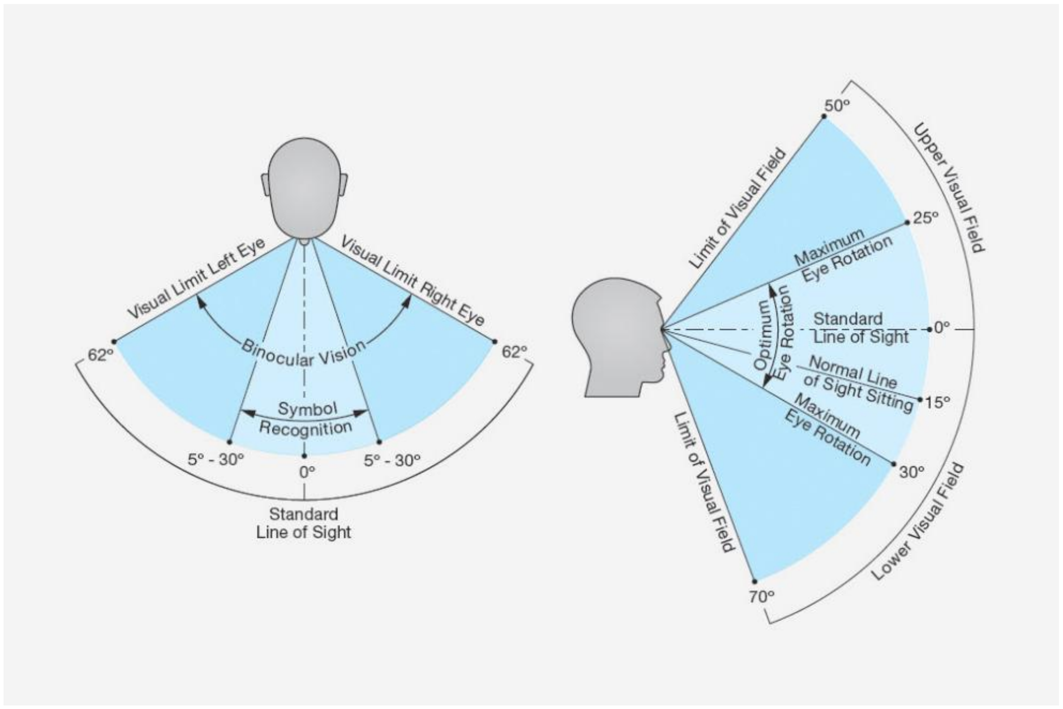
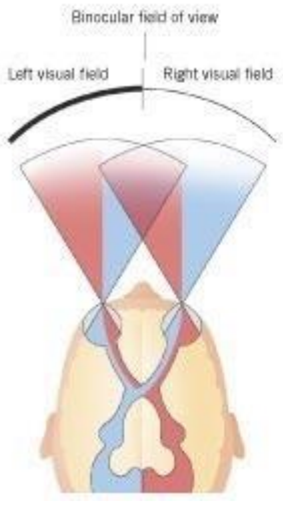


Figure 16: Visual Limits[16]

Stereopsis is used to refer to the perception of depth and 3-dimensional structure obtained on the basis of visual information deriving from two eyes by individuals with normally developed binocular vision. Binocular vision results in two slightly different images projected to the retinas of the eyes because of the different lateral positions the eyes are located on the head. These positional differences are referred as binocular disparities. These disparities are processed by the brain to yield depth perception. While binocular disparities are naturally present when viewing a real 3-dimensional scene with two eyes, they can also be simulated by artificially presenting two different images separately to each eye using the method of stereoscopy.

[16] Figure 16 was retrieved from Google



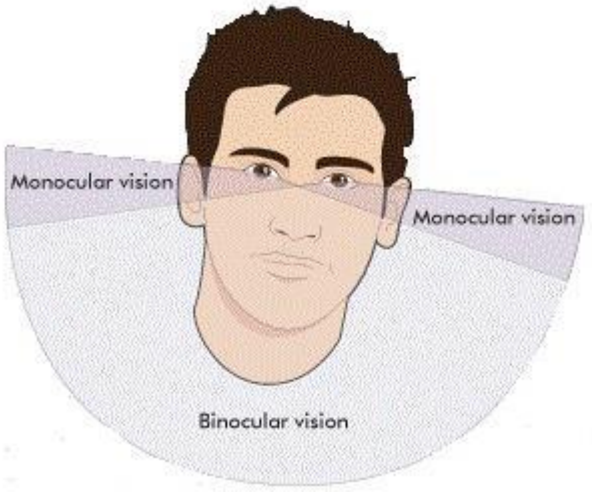


Figure 17: Binocular Vision[17]

On the other hand using the eyes separately, monocular vision, the FoV is increased while depth perception is limited. Monocular vision implies that only one eye is receiving optical information, the other one is closed. The perception of depth and 3-dimensional structure is, however, possible with information visible from one eye alone, such as differences in object size and motion parallax (differences in the object over time with observer movement), though the impression of depth in these cases is often not as vivid as the obtained from binocular disparities.

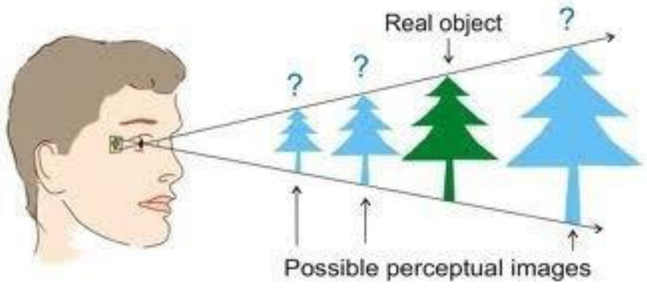


Figure 18: Monocular Vision & Depth[18]

Therefore, the term stereopsis can refer specifically to the unique impression of depth associated with binocular vision; what is colloquially referred to as "seeing in 3D".

[17] Figure 17 was retrieved from Google
[18] Figure was retrieved from Google

2.4.4 Eye movement and control

As eye movement both voluntary and involuntary movement of the eyes that help acquiring, fixating and tracking visual stimuli are considered. The human eye has numerous parts that must be controlled. Below, there will be a simple reference of the most major types of eye movements and only the most commonly used in past research and in this thesis will be described. The main types of eye movements are the following saccade, pursuit, smooth, compensatory, and blink.

The main measurements used in eye-tracking research are fixations and saccades.

A **saccade** is a rapid eye movement, a jump, which is usually conjugate and under voluntary control but ballistic; once they are initiated, the path of motion and destination cannot be changed. It takes about 100 to 300 milliseconds to initiate a saccade, i.e. from the time a stimulus is presented till the eye starts moving, and another 30 to 120 milliseconds to complete the saccade. The purpose of these movements is to bring images of particular areas of the visual world to fall onto the fovea which is only about 1-2 degrees of vision. Saccades are therefore a major instrument of selective visual attention. It is often convenient to consider both that a saccadic eye movement always occurs in a straight line and also that we do not 'see' during these movements. (Sacade - Wikipedia, 2018)

Fixation is the moment, in average 218 milliseconds, when the eyes are relatively stationary, between saccades, taking in or encoding information. Fixation duration provides an index of the speed with which information is processed. Increasing fixation duration is associated with tasks that require more detailed visual analysis. Frequency of fixations often serves as a measure of sampling quantity. They can reveal the amount on processing being applied to objects and therefore studying them can tell us about the complexity or salience of an object in an interface. (Wikipedia, Fixation - Wikipedia, 2017)

A **scanpath** describes a complete saccade-fixate-saccade sequence. In a search task, for example, an optimal scan path is viewed as being a straight tile to a desired target, with relatively short fixation duration at the target. For both long-lasting and long scanpaths, when detected, less efficient scanning is indicated. Also, comparing saccade times to fixating times helps in research.

Blinking is the automatic rapid closing of the eyelid. If detected can be used as an input device, for example as a mouse click. Blink rate can be used as an index of cognitive workload. A lower blink rate is assumed to indicate higher workload and a higher blink rate fatigue. However, blink rate may be determined by other factors such as ambient light levels. (Wikipedia, Blinking, 2017)

Drifts are slow movements away from a fixation point. Flicks or micro saccades reposition the eye on the target. Predominantly these are corrective movements, correcting for the off-center foveal position produced by a drift eye movement. Irregular slow movements of the eye also occur. High frequency tremor causes the image of an object to constantly stimulate cells in the fovea.

2.4.5 Perception in an immersive virtual environment

Perception of our immediate environment is not based on what we actually see or what is there. It is based upon little actual sensory information and is for the most part illusory. Theoretically everything we 'see' around us exists as a model in our minds. We rely upon our perceptive system so much that it enables us to be fooled.

When immersed in a VE our compelling senses are presented with an alternative view of our local environment whilst the real world is shut out. Our perceptual system is trained over many years to recognize our everyday reality, thus has no experience to distinguish it from the VR. An IVE simply provides cues that are a sufficient match for our inner conceptual models of what it is to be in an environment. For instance, stage magicians rely upon this fact by providing basic cues that purposely misinform our perceptual system and leave us wondering how we have apparently jumped from one world-state to another. Just as when we see an illusion and are able to accept the perhaps 'odd' perspective that is implied, when we view an IVE we can accept the virtual world perspective implied over the real world.

2.5 Eye tracking

Eye tracking is a technique whereby an individual's eye movements are measured so that the researcher knows both the point of gaze ("where a person is looking") at any given time and the sequence in which their eyes are shifting from one location to another. Gaze is the externally-observable indicator of human visual attention, and many have attempted to record it, dating back to the late eighteenth century [14]. Today, a variety of solutions exist (many of them commercial) but all suffer from one or more of the following: high cost (e.g., Tobii X2-60), custom or invasive hardware (e.g., Eye Tribe, Tobii EyeX) or inaccuracy under real-world conditions. (Krafka, Khosla, Kellnhofer, & Kannan, 2016)

Eye tracking can be used in two main ways, to improve UIs and to understand human behavior. Tracking people's eye movements can help Human Computer Interaction (HCI) researchers understand visual and display-based information processing and the factors that may impact upon the usability of system interfaces. In this way, eye-movement recordings can provide an objective source of interface-evaluation data that can inform the design of improved interfaces.

Eye movements can also be captured and used as control signals to enable people to interact with interfaces directly without the need for mouse or keyboard input, which can be a major advantage for certain populations of users such as disabled individuals.

Eye trackers are used in visual system research, in psychology, in cognitive linguistics and in product design. There are a number of methods for measuring eye movement. The most popular variant uses video images from which the eye position is extracted. Other methods use search coils or are based on the electro-oculography. (Bruneau, Sasse, & McCarthy, 2002)

The HMD employed in this research included embedded binocular eye tracking by Arrington Research.

2.5.1 History & Methods

Many methods have been used for eye tracking. Eye trackers measure rotations of the eye in several ways, but principally they fall into three categories:

One type uses an attachment to the eye, such as a special contact lens with an embedded mirror or magnetic field sensor, and the movement of the attachment is measured with the assumption that it does not slip significantly as the eye rotates. Measurements with tight fitting contact lenses have provided extremely sensitive recordings of eye movement, and magnetic search coils are the method of choice for researchers studying the dynamics and underlying physiology of eye movement.

The second broad category uses some non-contact, optical method for measuring eye motion. Light, typically infrared, is reflected from the eye and sensed by a video camera or some other specially designed optical sensor. The information is then analyzed to extract eye rotation from changes in reflections. Video based eye trackers typically use the corneal reflection (the first Purkinje image) and the center of the pupil as features to track over time. A more sensitive type of eye tracker, the dual-Purkinje eye tracker, uses reflections from the front of the cornea (first Purkinje image) and the back of the lens (fourth Purkinje image) as features to track. A still more sensitive method of tracking is to image features from inside the eye, such as the retinal blood vessels, and follow these features as the eye rotates. Optical methods, particularly those based on video recording, are widely used for gaze tracking and are favored for being non-invasive and inexpensive.

The third category uses electric potentials measured with electrodes placed around the eyes. The eyes are the origin of a steady electric potential field, which can also be detected in total darkness and if the eyes are closed. It can be modeled to be generated by a dipole with its positive pole at the cornea and its negative pole at the retina. The electric signal that can be derived using two pairs of contact electrodes placed on the skin around one eye is called Electrooculogram (EOG).

If the eyes move from the center position towards the periphery, the retina approaches one electrode while the cornea approaches the opposing one. This change in the orientation of the dipole and consequently the electric potential field results in a change in the measured EOG signal. Inversely, by analyzing these changes in eye movement can be tracked. Due to the discretisation given by the common electrode setup two separate movement components – a horizontal and a vertical – can be identified. A third EOG component is the radial EOG channel, which is the average of the EOG channels referenced to some posterior scalp electrode. This radial EOG channel is sensitive to the saccadic spike potentials stemming from the extra-ocular muscles at the onset of saccades, and allows reliable detection of even miniature saccades. (Wikipedia, Eye tracking, 2017)

The Eye-tracker from Arrington Research, Inc. that was used in this thesis is based on an optical method and relies on two infrared video cameras for the estimation of eye motion and position by detecting saccades, fixations, drifts.

2.6 Rendering

Rendering or image synthesis is the automatic process of generating a photorealistic or non-photorealistic image from a 2D or 3D model or models in what collectively could be called a scene file by means of computer programs. A scene file contains objects in a strictly defined language or data structure; it would contain geometry, viewpoint, texture, lighting, and shading information as a description of the virtual scene. The data contained in the scene file is then passed to a rendering program to be processed and output to a digital image or raster graphics image file.

Though the technical details of rendering methods vary, the general challenges to overcome in producing a 2D image from a 3D representation stored in a scene file are outlined as the graphics pipeline along a rendering device, such as a GPU. A GPU is a purpose-built device able to assist a CPU in performing complex rendering calculations. If a scene is to look relatively realistic and predictable under virtual lighting, the rendering software should solve the rendering equation. The rendering equation doesn't account for all lighting phenomena, but is a general lighting model for computer-generated imagery. 'Rendering' is also used to describe the process of calculating effects in a video editing program to produce final video output.

Rendering is one of the major sub-topics of 3D computer graphics, and in practice is always connected to the others. In the graphics pipeline, it is the last major step, giving the final appearance to the models and animation. With the increasing sophistication of computer graphics since the 1970s, it has become a more distinct subject.

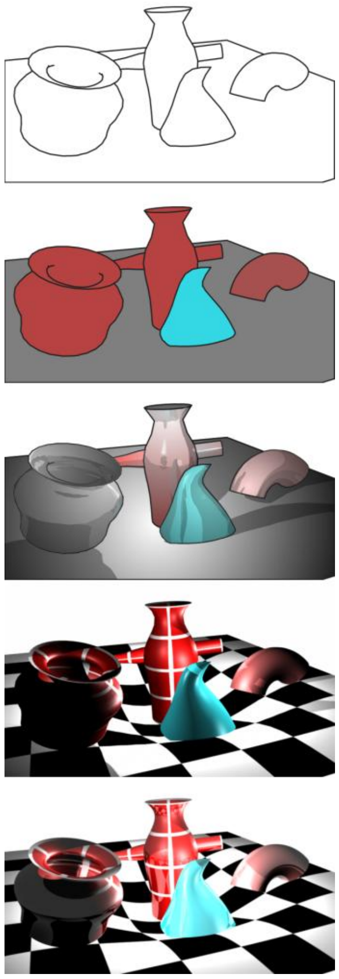


Figure 19 : A variety of rendering techniques applied to a single 3D scene[19]

2.6.1 Real time rendering

Rendering has uses in architecture, video games, simulators, movie or TV visual effects, and design visualization, each employing a different balance of features and techniques. As a product, a wide variety of renderers are available. Some are integrated into larger modeling and animation packages, some are stand-alone, some are free open-source projects. On the inside, a renderer is a carefully engineered program, based on a selective mixture of disciplines related to: light physics, visual perception, mathematics, and software development.

In the case of 3D graphics, rendering may be done slowly, as in pre-rendering, or in real time. Pre-rendering is a computationally intensive process that is typically

[19] Figure 20 was retrieved from Wikipedia

used for movie creation, while real-time rendering is often done for 3D video games which rely on the use of graphics cards with 3D hardware accelerators.

Rendering for interactive media, such as games and simulations, is calculated and displayed in real time, at rates of approximately 20 to 120 frames per second. In real-time rendering, the goal is to show as much information as possible as the eye can process in a fraction of a second. The primary goal is to achieve an as high as possible degree of photorealism at an acceptable minimum rendering speed (usually 24 frames per second, as that is the minimum the human eye needs to see to successfully create the illusion of movement).

2.6.2 Foveated rendering

Foveated rendering is an upcoming graphics rendering technique which uses an eye tracker integrated with a virtual reality headset to reduce the rendering workload by greatly reducing the image quality in the peripheral vision (outside of the zone gazed by the fovea).

Ideally, the graphics card would be able to render at full display resolution where the gaze is centered and continuously decrease resolution outward from there. However, this is extremely difficult, meaning that it needs extreme amounts of process power, to be achieved.

A more efficient method on current graphics hardware is to approximate this idea by rendering several overlapped rectangular regions, called eccentricity layers. All except the outer eccentricity layer cover only part of the total display area. After all layers are rendered they are interpolated to the final display resolution and finally blended together.

2.6.3 Research on foveated rendering

Since the topic discussed is very recently developed by big companies as NVIDIA, SMI, and Google, the code that they are using is not to be accessed by the public. The research that has been done focuses on presentations and papers submitted on two of the biggest technological conferences in the world; the IEEE and ACM conferences.

2.6.4 Techniques of Foveated Rendering

One of the first techniques of foveated rendering was implemented by (Levoy & Whitaker, 1990) which varied image resolution as a function of the Euclidean distance from the fovea’s fixation point. To achieve the foveated result, discrete levels of detail (LODs) were used. Another technique, proposed by (Ohshima, Yamamoto, & Tamura, 1996), was using pre-computed object meshes at varying level of details. More than ten years later, (Murphy, Duchowski, & Tyrrell, 2009) designed a foveation method based on Contrast Sensitivity Function (determining the contrast detection thresholds as a function), and varied image degradation according to the respective angular frequency, without modifying underlying scene geometry. Recently, (Guenter B. , Finch, Drucker, Tan, & Snyder, 2012) used three layers that include a different resolution and blended these layers to provide a high-quality foveated rendering result. (Swafford, Iglesias-Guitian, Koniaris, & Moon, 2016).

For other experiments, techniques that are using sampling maps have been used. Sometimes the sampling maps get updated before every frame depending on the current eye gaze. It should be noted that the general concept of using a sampling map, either static or dynamic, requires a rendering architecture in which the amount of supersampling can be chosen on a per-pixel level (Pohl, Zhang, & Bulling, 2016).

Another way to achieve foveated rendering is by using ray-based methods. Ray-based methods allow for fully adaptive sampling of the image plane. Therefore, it is possible to dynamically adjust the sampling probability of each individual pixel by accounting for its angular distance to the user's gaze. This yields a decreased coverage of the image with pixel information towards the outer regions, making it necessary to fill in the gaps. To do so, it is necessary to save the last image produced in order to be able to process parts of it to fill in the gaps of the new image (Roth, Weier, Hinkenjann, Li, & Slusallek, 2016).

Additionally, in another approach, (Swafford, Cosker, & Mitchell, 2015) used two different layers one for the foveal rendering (around the user's gaze) and one for the peripheral rendering. On this experiment the foveal diameter was rendered at approximately ¼ of the screen. Despite the high system latency the algorithm ran at nearly 24.3 FPS while the scene without the foveation algorithm renders at 14 FPS on 4K UHD.

A different technique renders three different image layers with different sampling rates. The three layers are then magnified to display resolution and smoothly composited. An additional code was needed in this technique to minimize twinkling artifacts in the lower-resolution layers. (Guenter B. , Finch, Drucker, Tan, & Snyder)

Another technique which has been widely adopted within the graphics industry is Tessellation. Geometry tessellation is a vertex processing stage that adaptively subdivides coarser geometry patches on-the-fly into smaller geometric primitives to generate nicer and smooth-looking details. (Swafford, Iglesias-Guitian, Koniaris, & Moon, 2016). In order to determine the appropriate level of tessellation, if a tile falls within either the foveal or peripheral field of view, the level of tessellation is set statically to the appropriate level. If the tile falls between the two regions (on the blending border) the level of tessellation is linearly interpolated between the two levels. This method is presented in the image below, in which the inner circle is the foveal region, between circles is the inter-regional blending, and outside the circle is the peripheral region.

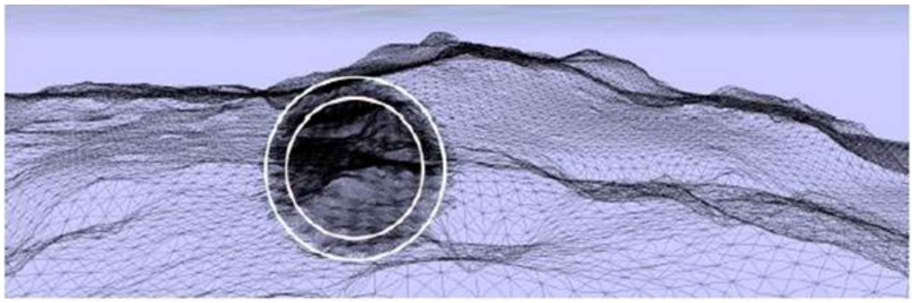


Figure 20: Wireframe view of the foveated Tessellation method

In the case of 360 degrees of immersive virtual reality content, the method that was originally developed by (Policarpo & Oliveira, 2006), is using different number of per-pixel ray-casting steps across the field of view depending on the depth layer detected. In case of too low number of steps many dis-occlusion errors occur. But since those pixels are placed in the peripherally field of view they are unnoticed by the user and provide high performance (Swafford, Iglesias-Guitian, Koniaris, & Moon, 2016).

2.6.5 Comparison with our technique

After describing the various techniques used in the past to produce a foveated rendering result, it is useful to compare these techniques with the technique applied in this thesis.

In our approach, the desired foveated rendering result is produced by changing the resolution in which we render the scene based on the Euclidean distance from the fovea’s fixation point. In order not to render each pixel in a different resolution we are dividing the screen into three distinct layers. The rendering resolution of each pixel depends on the layer in which it belongs to. In other words, our technique is a combination of the techniques implemented by (Levoy & Whitaker, 1990) and (Guenter B. , Finch, Drucker, Tan, & Snyder, 2012).

Although we did consider using pre-computed object meshes at varying level of details like (Ohshima, Yamamoto, & Tamura, 1996), when we implemented the technique, our scene’s rendering latency increased dramatically. As a result, this method was excluded. The same problem did emerge when we tried to sample our scene using rays. After carefully studying the results of (Roth, Weier, Hinkenjann, Li, & Slusallek, 2016) research, the idea of being able to dynamically adjust the sampling probability of each individual pixel by accounting for its angular distance to the user’s gaze was very tempting. Unfortunately, this approach needed high computational power and would not run on a standard PC. Our goal was that the foveated rendering technique proposed would run on a mid-range PC.

Even though we could not implement the technique that (Roth, Weier, Hinkenjann, Li, & Slusallek, 2016) implemented, we did create 4 different versions of our algorithm just like they did. Each version of the algorithm had a different set of parameters, which in our case, defined the size of the three rendering layers. By doing so, we were able to compare the performance benefits in terms of FPS provided by our foveated rendering technique by adjusting the foveal layer’s dimensions to zero, small, medium, and large. Setting the foveal layer’s dimension to zero is equivalent to rendering the scene at full resolution.

Chapter 3 Software architecture and development

3.1 Game engines

A game engine is a software framework designed for the creation and development of video games. Developers use them to create games for consoles, mobile devices and personal computers. The core functionality typically provided by a game engine includes a rendering engine (renderer) for 2D or 3D graphics, a physics engine, a collision detection (and collision response) system, sound, scripting, animation, artificial intelligence, networking, streaming, memory management, threading, localization support, scene graph, and may include video support for cinematics. The process of game development is often economized, in large part, by reusing/adapting the same game engine to create different games, or to make it easier to "port" games to multiple platforms.

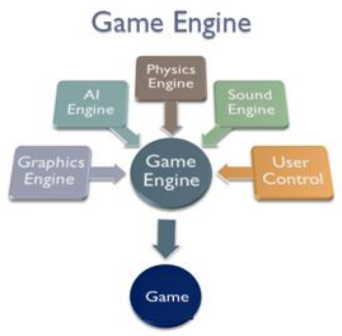


Figure 21: Parts of a Game Engine[20]

The beauty and power of game engines, is that they speed-up the development process, by providing a suite of visual development tools, reusable software components and simplification of frequently used tools, elements and processes. Game Engines are usually built upon one or multiple rendering application programming interfaces (APIs), such as Direct3D or OpenGL which provide a software abstraction of the graphics processing unit (GPU).

These APIs are commonly used to interact and communicate with the GPU, to achieve hardware-accelerated rendering.

Modern game engines are some of the most complex applications written, which is the result of years and years of improvements, experience and development. Nowadays they often feature dozens of finely tuned systems interacting to ensure a precisely controlled user experience. The continued evolution of game engines has created a strong separation between rendering, scripting, artwork, and level

[20] Figure 19 was retrieved from Wikipedia

design. It is now common, for example, for a typical game development team to have several times as many artists as actual programmers.

Furthermore, due to the constant growth of the Smartphone application market and increasing competition, popular high-end Game Engines are proving to be a precious tool for developers worldwide, to bring their ideas and games to life, in as many platforms as possible. (Wikipedia, Game Engine, 2018)

3.2 Popular game engines

It is easy to pinpoint three game engines that stand out from the rest, mostly because of their popularity and the fact that they are free to use.

Unreal Engine: Unreal Engine 4 is a complete suite of development tools made for anyone working with real-time technology. From enterprise applications and cinematic experiences to high-quality games across PC, console, mobile, VR and AR, Unreal Engine 4 gives you everything you need to start, ship, grow and stand out from the crowd. A world-class toolset and accessible workflows empower developers to quickly iterate on ideas and see immediate results without touching a line of code, while full source code access gives everyone in the Unreal Engine 4 community the freedom to modify and extend engine features. (Game Engine Tecnhology by UE, 2018)

Unity3D: Unity 3D, initially released on 2005, is a flexible and powerful development platform for creating high quality 2D and 3D games. Emphasizing on portability, Unity currently supports over 20 platforms, including PCs, consoles, mobile devices (iOS and Android) and websites. Additionally, many settings can be configured for each platform. As a result, Unity can detect the best variant of graphic settings for the hardware or platform the game is running, thus optimizing performance and sacrificing visual quality if necessary. Apart from its next-generation graphical capabilities, Unity also comes with an integrated physics engine (Nvdias PhysX). Much like Unreal Engine, Unity offers developers an Asset Store to buy re-usable content and assets for use in their project. To sum up, due to its ability to efficiently target multiple platforms at once and user friendly environment, this game engine is an ideal choice for a large portion of developers.

CryEngine: CryEngine is a game engine developed by game developer Crytek, which has been used in all of their titles. It is known for its ability to produce stunning, eye-catching graphics and visuals, featuring advanced shader and lightning systems. Because of this, CryEngine clearly targets only powerful PCs and high-end consoles. It comes with VR support and a large amount of advanced visual features, tools, audio/physics systems and character and animation systems. (CryEngine, 2018)

3.2.1 Choosing the right engine

Unreal Engine and Unity are currently ahead of the competition as the two most popular game engines available to the public. This is due to the fact that they both succeed in providing high-end graphics, a large variety of usable tools, great support for platforms and devices, without compromising usability and efficiency. It is important to note that these 2 Game Engines offer a large community support, which is also something that has to be considered when choosing the right Game Engine. CryEngine is also great and powerful engine with remarkable capabilities. However its complicated structure and smaller community excluded it from our consideration. In conclusion, taking into account the advantages and disadvantages of each engine, Unity proved to be the ideal choice for this project, mainly due to its efficiency, large community support and ease of use.

3.3 Unity game engine

For the purposes of this project, the Unity Game Engine has been selected to provide the development environment. Unity was selected because of its ease of use, the numerous online guides/tutorial, the ability to create 2D graphics for user interfaces, the familiar scripting language (C#), the Unity's Asset Store, where it is easy to find objects without having to create everything from scratch, the thriving and supportive community and last but not least the ability to use the full range of Game Engine tools and programming capabilities for free. The main components/windows of Unity Game Engine are: Hierarchy, Project, Console, Scene, Game, Inspector and, most importantly, Scripting mechanisms. A more detailed description of each component follows.

3.3.1 Hierarchy

The Hierarchy window contains a list of every GameObject in the current Scene. Some of these are direct instances of Asset files (like 3D models), and others are instances of Prefabs, which are custom objects that make up most of the game. As objects are added in and removed from the Scene, they appear and disappear from the Hierarchy as well.

By default, objects are listed in the Hierarchy window in the order they are created. Re-ordering of objects can be done easily by dragging them up or down, or by making them "child" or "parent" objects.

3.3.2 Project

In the project window, one can access and manage the assets that belong to their project. It consists of two panels. The left panel of the browser shows the folder structure of the project as a hierarchical list. When a folder is selected from the list by clicking, its contents will be shown in the panel to the right. The user can click to expand or collapse the folder, displaying any nested folders it contains.

The individual assets are shown in the right panel as icons that indicate their type (script, material, sub-folder, etc.). The icons can be resized using the slider at the bottom of the panel; they will be replaced by a hierarchical list view if the slider is moved to the extreme left. The space to the left of the slider shows the currently selected item, including a full path to the item if a search is being performed.

Just above the panel is a “breadcrumb trail” that shows the path to the folder currently being viewed. The separate elements of the trail can be clicked for easy navigation around the folder hierarchy. When searching, this bar changes to show the area being searched (the root Assets folder, the selected folder or the Asset Store) along with a count of free and paid assets available in the store, separated by a slash. There is an option in the General section of Unity’s Preferences window to disable the display of Asset Store hit counts if they are not required.

The Project Browser’s search can also be applied to assets available from the Unity Asset Store. If the user selects the option of Asset Store from the menu in the breadcrumb bar, all free and paid items from the store that match user’s query will be displayed. If they select an item from the list, its details will be displayed in the inspector along with the option to purchase and/or download it. Some asset types have previews available in this section so the user can, for example, rotate a 3D model before buying.

3.3.3 Console

The Console Window pinpoints errors, warnings and other messages generated by Unity, to aid with debugging. The user can also show his own messages in the Console using the implemented functions of Unity (Debug.Log, Debug.LogWarning and Debug.LogError).

The toolbar of the console window has numerous options that affect how messages are displayed. The Clear button removes any messages generated from user’s code but retains compiler errors. The user can also arrange for the console to be cleared automatically whenever he runs the game by enabling the Clear on Play option. There is also the opportunity to change the way messages are shown and updated in the console. The Collapse option shows only the first instance of an error message that keeps recurring. This is very useful for runtime errors, such as null references, that are sometimes generated identically on each frame update. The Error Pause option will cause playback to be paused whenever Debug.LogError is called from a script. Finally, there are two options for viewing additional information about errors. The Open Player Log and Open Editor Log items on the console tab menu access Unity’s log files which record details that may not be shown in the console.

3.3.4 Scene

The Scene window is the interactive view into the world that the user is creating. Scene View can be used to select and position scenery, characters, cameras, lights, and all other types of Game Objects. Being able to Select, manipulate and modify objects in the Scene View are some of the first skills somebody will need to begin his first steps in Unity.

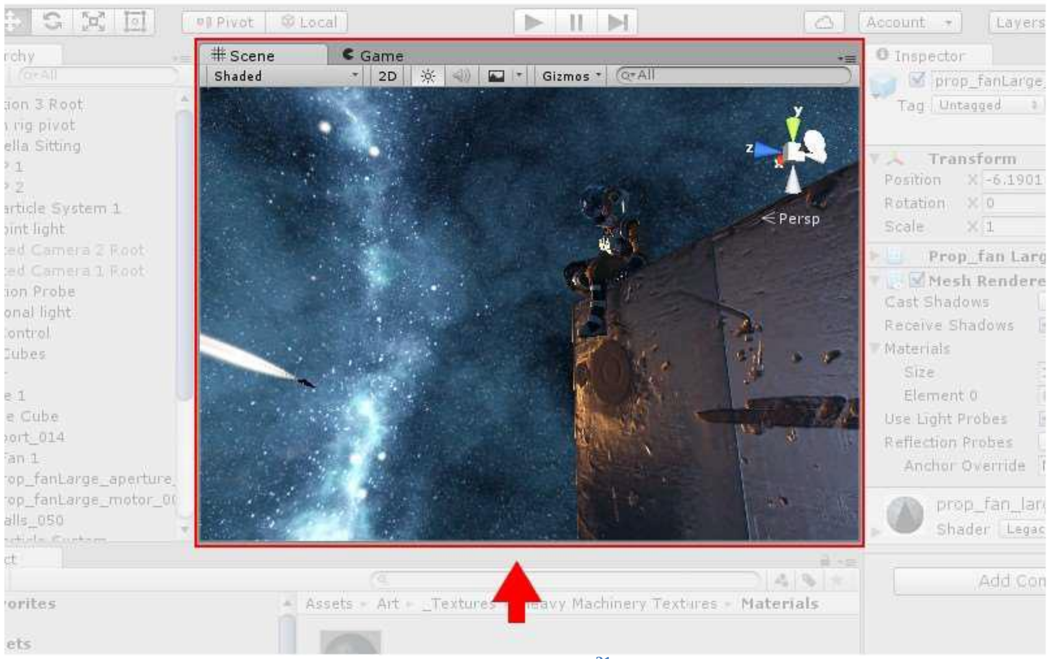


Figure 22: Scene[21]

The Scene Gizmo is in the upper-right corner of the Scene View. This displays the Scene View Camera's current orientation, and allows the user to quickly modify the viewing angle and projection mode.

In order to Move, Rotate, Scale, or Transform individual GameObjects, the user can use the four Transform tools in the toolbar. Each has a corresponding Gizmo that appears around the selected GameObject in the Scene view. To alter the Transform component of the GameObject, the user can use the mouse to manipulate any Gizmo axis, or type values directly into the number fields of the Transform component in the Inspector.

The Scene view control bar provides the user with the opportunity to select between various options for viewing the Scene and also control whether lighting and audio are enabled. These controls only affect the Scene view during development and have no effect on the built game.

[21] Figure 22 was retrieved from Unity Game Engine

3.3.5 Game

The Game window is rendered from the Camera in user’s game. It is representative of the final, published game. It is necessary for the user to use one or more Cameras to control what the player actually sees when they are playing the game.

3.3.6 Inspector

Projects in the Unity Editor are made up of multiple GameObjects that can potentially contain scripts, sounds, meshes, and other graphical elements such as lights. The Inspector window displays detailed information about the currently selected GameObject, including all attached components and their properties, and allows the user to modify the functionality of GameObjects in the Scene.

The user can use the Inspector to view and edit the properties and settings of almost everything in the Unity editor, including physical game items such as GameObjects, assets, and materials, as well as in-editor settings and preferences. When a GameObject is selected in either the Hierarchy or Scene view, the Inspector shows the properties of all components and materials of that GameObject. Actually, the Inspector can be used to edit the settings of these components and materials.

3.3.7 Scripting

Scripting is an essential part of Unity as it defines the entire behavior of the game or application. Even the simplest game needs a script to respond to user’s input. Scripts can be used for several reasons such as: to create graphical effects, to control physical behavior of objects or characters, to trigger effects upon specified conditions or even implement a custom A.I. system for characters in the game.

```
        [illegible].blur = (1-health) * 2 + [illegible]
        vignette.blurDistance = (1-health) * 2 + [illegible]
        vignette.chromaticAberration = [illegible]
    }


    void OnTriggerStay(Collider c)
    {
        var fire = c.GetComponent<Fire>();
        if (fire && fire.alive)
        {
            float dist = 1-((transform.position - [illegible]
            NearHeat(dist);
        }

        var smoke = c.GetComponent<[illegible]
        if (smoke && smoke.GetComponent<[illegible]
        {
            float dist = 1-((transform.position - [illegible]
            NearSmoke(dist);
        }
    }

    void OnCollisionEnter([illegible]
    {
        var healthBox = c.[illegible]
        if (healthBox)
```

Figure 23: Example of Scripting

The behavior of GameObjects is controlled by the Components that are attached to them. Although Unity's built-in Components can be very versatile, the programmer will soon find he needs to go beyond what Unity can provide to implement his own gameplay features. Unity allows the user to create his own Components using scripts. These allow him to trigger game events, modify Component properties over time and respond to user input in any way he wants to. Unity supports two programming languages natively, **C#**, an object oriented programming language similar to Java or C++ and the UnityScript, a language designed specifically for use with Unity and modelled after JavaScript. The scripts can be written and edited in MonoDevelop, which is an integrated development environment (IDE) within Unity or in any other IDE like Visual Studio. An IDE combines a text editor with additional features for debugging, auto-complete and other project management tasks. A script makes its connection with the internal workings of Unity by implementing a class which derives from the built-in class called MonoBehavior. The reader can think of a class as a kind of blueprint for creating a new Component type that can be attached to GameObjects. Each time the programmer attaches a script component to a GameObject, it creates a new instance of the object defined by the blueprint. The name of the class is taken from the name that the programmer supplied when the file was created. The class name and file name must be the same to enable the script component to be attached to a GameObject. Thus, components can be accessed or modified by script at any time to achieve desired behavior and functionality. When a script is created, there are two functions automatically declared in it, the **Start** function and the **Update** function. The Update function is the right place to write code that will handle the frame update for the GameObject. This might include movement, triggering actions and responding to user input, basically anything that needs to be handled over time during gameplay. To enable the Update function to do its work, it is often useful to be able to set up variables, read preferences and make connections with other GameObjects before any game action takes place. The Start function will be called by Unity when a script is enabled and will be called exactly once in its lifetime. The Start function is the ideal place where initialization occurs. It is used to initialize an object's position, state and properties or load other scripts and GameObjects for later use.

A script in Unity is not like the traditional idea of a program where the code runs continuously in a loop until it completes its task. Instead, Unity passes control to a script intermittently by calling certain functions that are declared within it. Once a function has finished executing, control is passed back to Unity. These functions are known as event functions since they are activated by Unity in response to events that occur during gameplay. Unity uses a naming scheme to identify which function to call for a particular event. For instance, we have already mentioned the Update function (called before a frame update occurs) and the Start function (called just before the object's first frame update). Many more event functions are available in Unity; the following are some of the most common and important events.

- **Regular Update Events**: These events can make changes to position, state and behavior of objects in the game just before each frame is rendered. The Update function is the main place for this kind of code in Unity. Update is called before the frame is rendered and also before the animations are calculated. For physics update, like adding force to a GameObject, the best option is to place the code in the FixedUpdate function which updates more frequently than the Update function. Sometimes the best place to write code is the LateUpdate function in order to be able to make additional changes at a point after the Update and FixedUpdate functions have been called for all objects in the scene and after all animations have been calculated.
- **Initialization Events**: It is often useful to be able to call initialization code in advance of any updates that occur during gameplay. The Start function is called before the first frame or physics update on an object. The Awake function is called for each object in the scene at the time when the scene loads. Note that although the various objects' Start and Awake functions are called in arbitrary order, all the Awakes will have finished

before the first Start is called. This means that code in a Start function can make use of other initializations previously carried out in the awake phase.

- **GUI Events:** Unity has a system for rendering GUI controls over the main action in the scene and responding to clicks on these controls. This code is handled somewhat differently from the normal frame update and so it should be placed in the OnGUI function, which will be called periodically. For instance, a set of OnMouseXXX event functions (e.g., OnMouseOver, OnMouseDown) is available to allow a script to react to user actions with the mouse. For example, if the mouse button is pressed while the pointer is over a particular object then an OnMouseDown function in that object's script will be called if it exists.
- **Physic Events:** The physics engine will report collisions against an object by calling event functions on that object's script. The OnCollisionEnter, OnCollisionStay and OnCollisionExit functions will be called as contact is made, held and broken. The corresponding OnTriggerEnter, OnTriggerStay and OnTriggerExit functions will be called when the object's collider is configured as a Trigger (i.e., a collider that simply detects when something enters it rather than reacting physically). These functions may be called several times in succession if more than one contact is detected during the physics update and so a parameter is passed to the function giving details of the collision (position, identity of the incoming object, etc.).

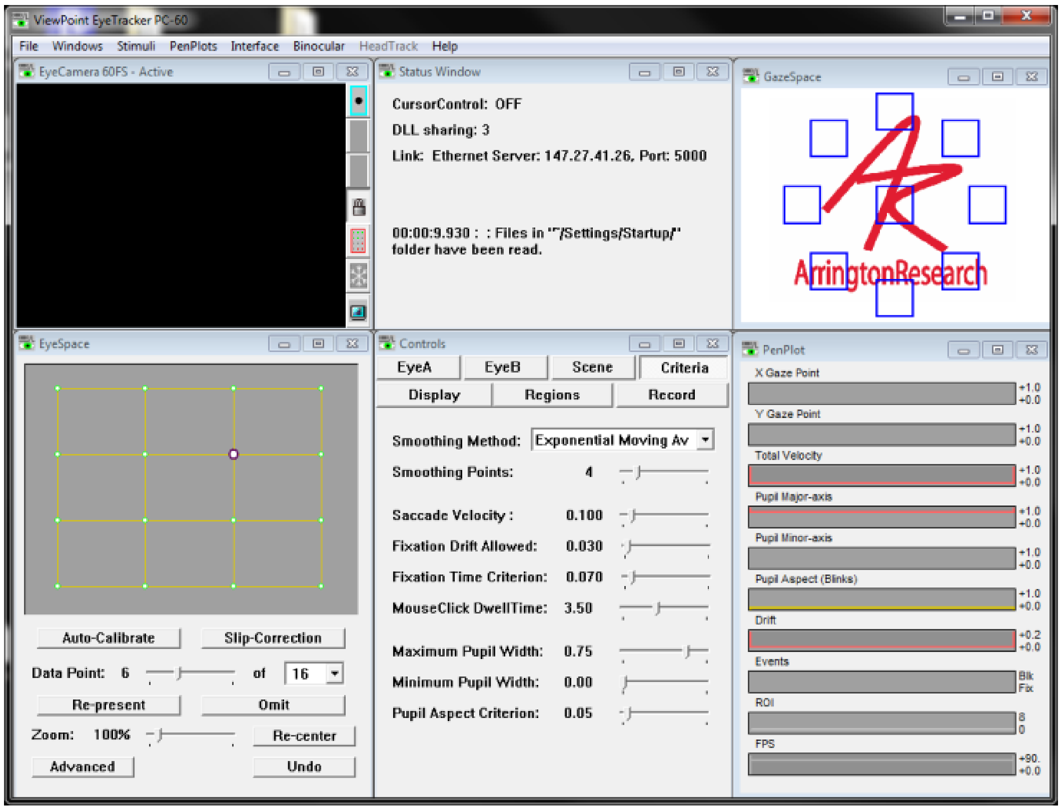
Except for the functions that Unity provides, the developer can create his/her own functions in order to control or determine the behavior of a GameObject, change the properties of a component or altering the overall state of the application. In order for these custom functions to be executed, they have to be called inside a Unity event function, like the Update. The most commonly used functions were presented briefly above, as well as the concept of how they are used. The basic notion of the Unity scripting is that the scripts are components that can control the GameObject. Each component property corresponds to a script variable and the scripts can access not only the components of the GameObjects they are attached to, but also other GameObjects.

3.4 Overview of eye tracking device's software

The software used in this thesis regarding the eye tracking device embedded in the HMD is the ViewPoint EyeTracker® of Arrington Research, Inc. This software provides a complete eye movement evaluation environment including integrated stimulus presentation, simultaneous eye movement and pupil diameter monitoring, and a Software Developer's Kit (SDK) for communication with other applications. Furthermore, it incorporates several methods from which the user can select to optimize the system for a particular application and methods of mapping position signals extracted from the segmented video image in EyeSpace^{TM} coordinates to the participant's point of regard GazeSpace^{TM}.

3.4.1. Interface

Only the most basic features of the software will be presented here. As soon as the program starts running, it displays several windows arranged as shown in the following figure. These windows are: the EyeCamera window, the EyeSpace window, the Status window, the Controls window, the GazeSpace and the PenPlot window. In order to inform the reader on the capabilities of the software, a brief summary of these windows is needed.



CursorControl: OFF
DLL sharing: 3
Link: Ethernet Server: 147.27.41.26, Port: 5000

00:00:9.930 : : Files in "~/Settings/Startup/" folder have been read.

Data Point: 6 of 16
Zoom: 100%

Smoothing Method:	Exponential Moving Av
Smoothing Points:	4
Saccade Velocity :	0.100
Fixation Drift Allowed:	0.030
Fixation Time Criterion:	0.070
MouseClick DwellTime:	3.50
Maximum Pupil Width:	0.75
Minimum Pupil Width:	0.00
Pupil Aspect Criterion:	0.05

Figure 24: Screenshot right after the launch of the application

The EyeCamera window displays the video image of the eye and the image analysis graphics. It provides controls to make the eye tracking results more reliable and to extend the range of the trace area. Thus, it makes it easy to limit the areas in which the software searches for the pupil and corneal reflection to exclude extraneous reflections and shadows.

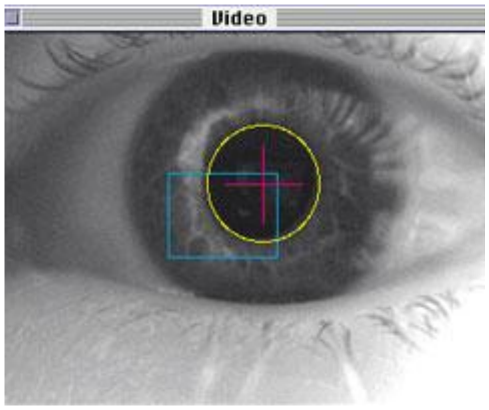


Figure 25: Screenshot of the eye camera that depicts the controls available

The EyeSpace window corresponds to the geometry of the EyeCamera image. It displays an array of the relative locations of the pupil, glint, or difference vector, which were obtained during calibration. This provides information about calibration accuracy and allows rapid identification and correction of individual calibration errors by allowing manual recalibration of individual points or the ability to omit problem points. The number, color and presentation rate can be set from here.

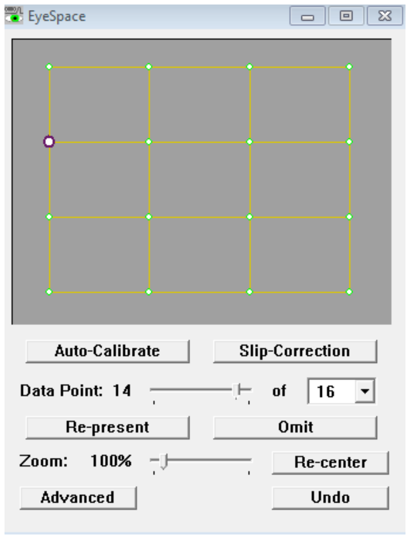
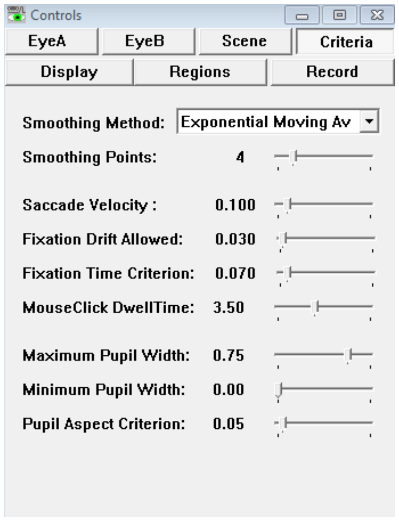


Figure 26: The eyespace before user calibration

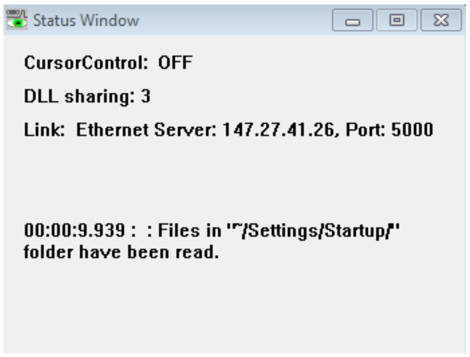
The Controls window allows the user to adjust the image-analysis and gaze-mapping parameter settings and to specify the feedback information to be displayed in both Stimulus window and the GazeSpace window. Eye Image quality adjustments can be made and tacking method can be specified. Also, smoothing and other criteria can be applied to data, has parameters to setup the regions of interest and calibration regions as well as adjust brightness, contrast, hue, saturation of the scene image and to open, pause, close data files or insert markers.



Smoothing Method: Exponential Moving Av
Smoothing Points: 4
Saccade Velocity : 0.100
Fixation Drift Allowed: 0.030
Fixation Time Criterion: 0.070
MouseClick DwellTime: 3.50
Maximum Pupil Width: 0.75
Minimum Pupil Width: 0.00
Pupil Aspect Criterion: 0.05

Figure 27: Different controls available to the user

The Status window gives details about processing performance and measurements. Also, shows the applications that share the same .dll of the application; thus the threads that ViewPoint communicates with.



CursorControl: OFF
DLL sharing: 3
Link: Ethernet Server: 147.27.41.26, Port: 5000

00:00:9.939 : : Files in "~/Settings/Startup/" folder have been read.

Figure 28: Status window

The Stimulus window is a new window that pops up when calibration starts. It is designed to be full screen, preferably on a second monitor. Upon which may be displayed the subject’s calculated position-of-gaze information and region of interest boxes.

Figure 29: Stimulus window

The Pen Plot window displays plots of X and Y positions of gaze, velocity, ocular torsion, pupil width, pupil aspect ratio, drift, etc. in real time.

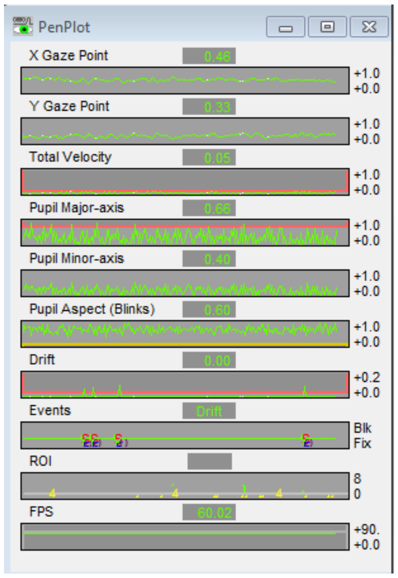


Figure 30: PenPlot real-time data

3.4.2 How the software works

To explain how the Viewpoint EyeTracker® works in a typical head fixed configuration, we will need to describe the next figure, Figure 31, which summarizes this exact process in thirteen steps.

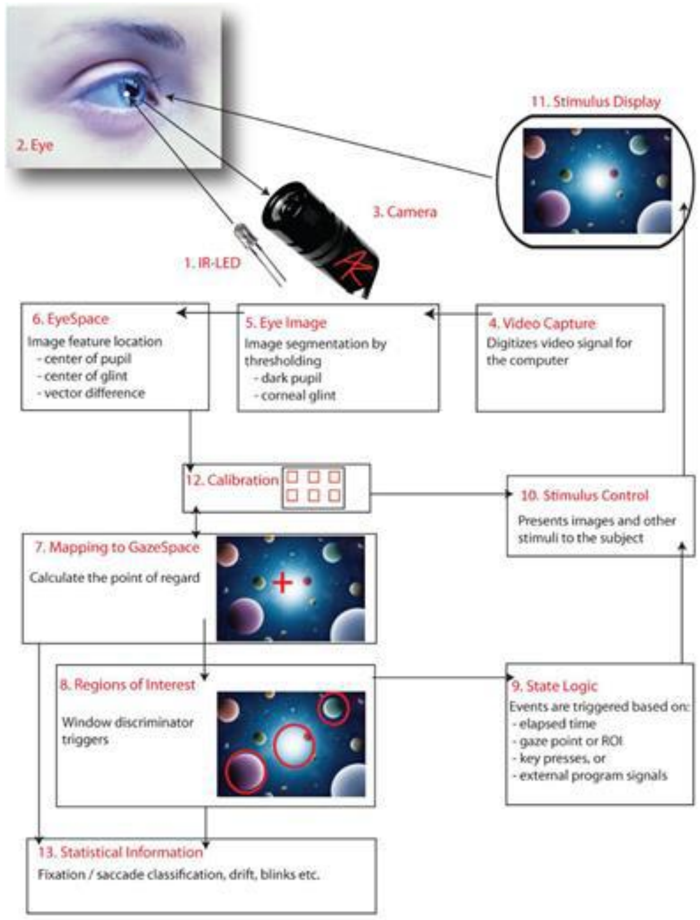


Figure 31: Depiction of how the Viewpoint software works

The infrared light source (*item 1*) has two purposes. First to illuminate the eye (*item 2*) and secondly to provide a specular reflection from the surface of the eye. In dark pupil mode, the pupil acts as infrared sink that appears as a black hole. In bright pupil mode, the “red eye” effect causes the pupil to appear brighter than the iris.

The video signal from the camera (*item 3*) is digitized by the video capture device (*item 4*) into a form that can be understood by the software. The computer takes the digitized image and applies image segmentation algorithms (*item 5*) to locate the areas of pupil and the bright corneal reflection, the glint. Additional image processing (*item 6*) locates the centers of these areas and also calculates the difference vector between the center locations. A mapping function (*item 7*) transforms the eye position signals (*item 6*) in EyeSpace coordinates to the subject’s GazeSpace coordinates. (*item 8*). Because the eye movements are rotational, for example the translation of the eye position signal that is apparent to the camera is a trigonometric function of the subject’s gaze angle, the best algorithms are non-linear. Next, the program tests to determine whether the gaze point is inside of any region of interest (ROI) that the user has defined.

The calibration system (*item 12*) can be used to present calibration stimuli via (*item 10*) to the user and to measure the eye position signals for each of the stimulus points. These data are then used by the calibration system to compute an optimal mapping function for mapping to position of gaze in GazeSpace.

3.4.3 Binocular and monocular method

The eye-tracker used in this thesis supports both monocular and binocular eye tracking.

By default, ViewPoint is set for monocular eye tracking. Switching from monocular to binocular is easy via the upper menu of the application. Once this is done, another EyeCamera window will appear, for the second eye and in the GazeSpace window there will be an additional drop down box to specify which eye the calibration process is applying to.

For the needs of this project monocular eye tracking was applied. This happened, because the user's gaze can be calculated accurately from the information provided by a single eye. It doesn't matter which eye we chose in our monocular approach. In this thesis, the left eye was used because of some technical difficulties with the right display of the HMD.

3.4.4 Calibration

In order to know where the participant's eyes are fixating on the computer screen, we must first "teach" the computer, in our situation the ViewPoint software, what the eye looks like when the participant's gaze is fixated on known locations on the screen. So, prior to using the eye tracker the user needs to undergo a personal calibration process. The reason for this is that each person has different eye characteristics, and different head geometry. As a result, the eye tracking software needs to model these in order to estimate gaze accurately.

The tracker operates by tracking the pupil of the eye, the "dark/black" part of the eye, which will appears as filled-in with blue in the camera-view of the eye, as well as the "corneal reflection", the reflection off of the cornea that appears in yellow on the eye-view. The relative positions and size of these two landmarks are measured as the participant looks at specific points on the screen during calibration. During the rest of the experiment, the tracker figures out where the eye must be looking, depending on the relative size of the pupil as well as the relative location of the pupil and corneal reflection. Anything that disrupts the pupil capture, or interferes with the corneal reflection will cause calibration to be very difficult or even impossible.

The calibration process lasts for approximately 1 to 2 minutes to be completed. In order for the calibration process to be completed correctly, two major steps must be followed. Firstly, it is necessary, for convenience reasons, to start the calibration process with the “auto-calibration” button. This button will initiate the calibration procedure by clearing everything from the user’s screen, and by placing a black background on it. Shortly after that, a text-message will appear on the screen. The message will say “Get ready”, and its purpose is to indicate to the user that he is about to begin the calibration process. When the get-ready text disappears, the eye gaze is calibrated by showing a calibration pattern with a total of 16 points, of which one is shown at a time and the user has to focus on it. As for the calibration points, it is mandatory to use at least 9 but tests showed that best calibration results are provided by using 16 or 20 points; for this thesis we used 16 points. Successful calibration will be indicated by a rectilinear and well separated configuration of green dots corresponding to the locations of the pupil at the time of calibration point capture; the green and yellow dots are show in the EyeSpace window.

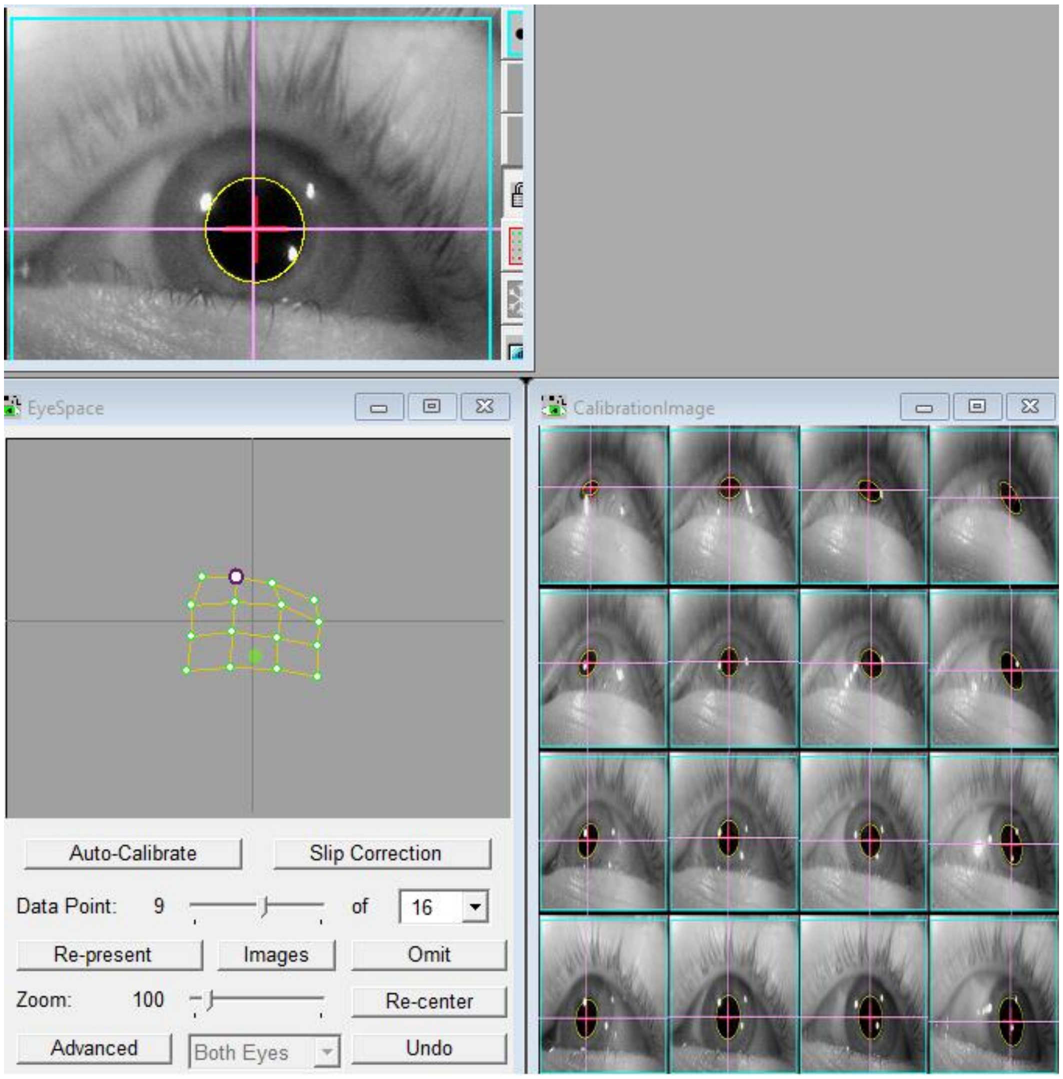


Figure 32: Almost perfect calibration result

Calibration data points can be identified and re-calibrated or omitted. The EyeSpace window allows the user to select individual calibration points to be recalibrated. In addition, it provides the user with the opportunity to see the actual images that were captured during the calibration process in order to have a full understanding of what caused the specific point to be badly calibrated. If a point is selected to be re-calibrated, then it will be re-presented in the screen and the participant will be asked to look at the center of it. This can be repeated with as many calibration data points as necessary. If the calibration points are not rectilinear, for example, if there are lines crossing, then complete re-calibration is necessary. If a particular point cannot be recalibrated, then that specific point can be omitted.

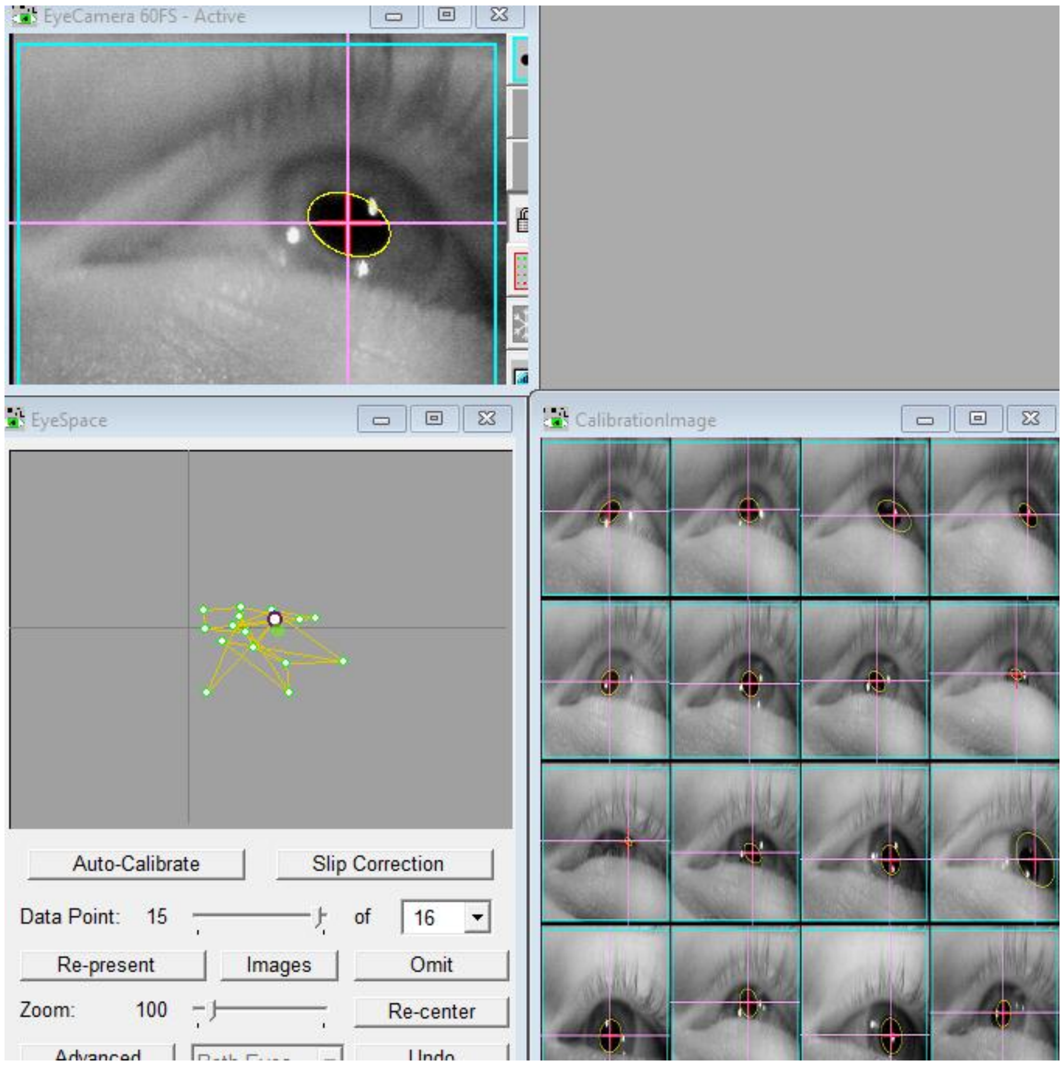


Figure 33: Result of bad calibration

In the two cases presented in figures 32 and 33, the first one depicts the result of an almost perfect calibration process, while the latter one presents a result far from perfect. Apart from the EyeSpace window, these figures intentionally contain the CalibrationImage window too. By looking at the images in the CalibrationImage window it is possible to understand the reasons behind the bad second result, which are no other than the loss of detection on the user's pupil, due to bad lighting conditions.

3.4.5 SDK and communication with Unity3D

ViewPoint software includes a powerful developer's kit (SDK) that allows interfacing with ViewPoint in real-time, giving real-time access to all ViewPoint data. Allowing complete external control of the ViewPoint EyeTracker, the SDK is based on shared memory in a dynamic-linked library (DLL). The SDK is event/message driven so there is no CPU load from polling and provides microsecond latency.

So, Unity3D interacts with ViewPoint by compiling the VPX_InterApp.lib file, a library file.

More details regarding how the data from the eye-tracker are retrieved and how they are processed in order to be used in our application can be found in subchapter 5.4.

3.5 Head tracking device - software

3.5.1 Overview

The head tracking device used in this thesis is the InertiaCube3 and the InertiaCube Processor by Thales. It is an inertial three degree of freedom (3-DOF) orientation tracking software. It obtains its motion sensing using a miniature solid-state inertial measurement unity, which senses angular rate of rotation, gravity and earth magnetic field along three perpendicular axes. The angular rates are integrated to obtain the orientation (yaw, pitch, and roll) of the sensor. Gravimeter and compass measurements are used to prevent the accumulation of gyroscopic drift.

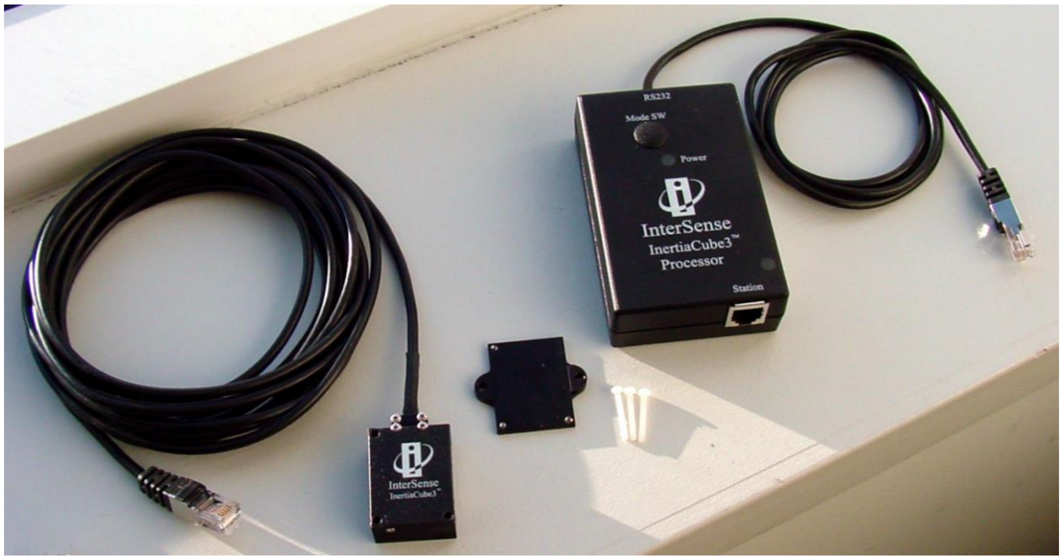
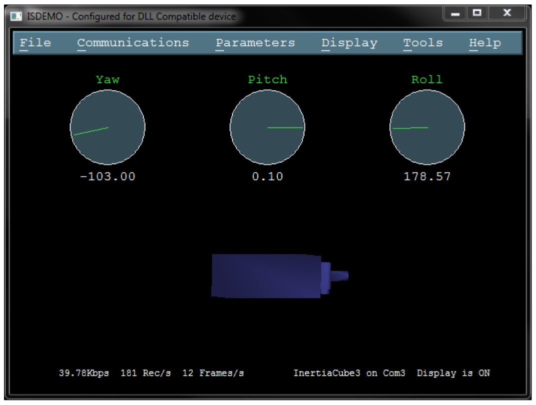


Figure 34: The InertiaCube3

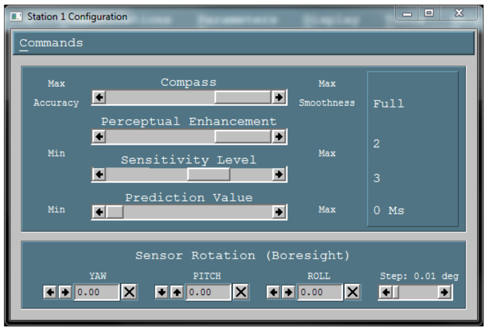
3.5.2 SDK and communication

The head tracking device has testing software and the InterSense Software Development Kit (SDK). The core of all InterSense software that is associated with the head tracker is a dynamic-linked library, the isense.dll, which must be saved in the Windows system directory. This library, along with all other InterSense libraries, provides a standard interface for the device.

Before the tracker is used, it must be configured and a diagnostic tool must be run. This testing tool, is the ISDEMO, validates the communication of the InertiaCube3 to the PC and tests the performance through the above mentioned DLL. In this phase, the compass, perceptual enhancement, sensitivity, prediction, and in general all the tracker's sensor parameters, can be modified and checked so that the device is working properly. Also, tools such as self-system test and compass calibration.



```
Yaw        Pitch     Roll
-103.00    0.10      178.57

39.78Kbps  181 Rec/s  12 Frames/s     InertiaCube3 on Com3  Display is ON
```



```
Max                                   Max
Accuracy         Compass              Smoothness   Full
          Perceptual Enhancement
                                                   2
Min         Sensitivity Level         Max
                                                   3
            Prediction Value
Min                                   Max          0 Ms

            Sensor Rotation (Boresight)
   YAW           PITCH          ROLL        Step: 0.01 deg
   0.00          0.00           0.00
```

Figure 35: Iserver App

There is also the InterSense Server Application, ISERVER, which provides multiple services to applications requiring tracker data. It is the link between the head tracker's data output and third party applications, in our case Unity3D. ISERVER runs in the system tray, reading the data from the connected device at the maximum speed allowed by the operating system. That data is then made available to Unity3D through the InterSense DLL.

Chapter 4 UI

Up until this point, the background and all the necessary hardware and software that were needed for the implementation of this thesis, have been sufficiently explained so that anyone, interested in the subject, can fully understand the following chapters. Now, it is time to discuss about this project’s User Interface (UI) implementation.

As mentioned before, this project aims to create a foveated rendering algorithm that renders the virtual scene using three layers of resolution and that ideally remains unnoticed by the users. By doing so, the algorithm offers a significant performance boost to the application in terms of FPS. We have implemented four versions of our foveated rendering approach. Each version adjusts the dimensions of the rendering layers differently. The application allows users to choose the version of the algorithm by clicking on one of the four buttons on the main menu scene. Each button corresponds to one of the four available versions of the algorithm.

Regardless of the user's choice, the application is programmed to load the exact same scene. Loading the same scene is important in order to apply the same rendering load to both the CPU and the GPU every time.

After loading the scene the user enters the virtual environment in which the user takes the role of a gondolier. Right when the game starts, the user meets his first customer approaching him slowly from his right side. When the customer boards the gondola, a message appears on the screen notifying the user about the customer’s desired destination. The destination is chosen randomly every time between the two available options. After that, golden coins pop – up on the map in order to help the user locate the correct destination area. The user has to transfer the customer to the correct area. Meanwhile, the user can freely enjoy the ride and the view. The ride takes about 2 minutes to be completed and for the whole duration we store data regarding the number of frames per second that are rendered.

In this chapter we will explain how the main menu and the virtual scenes are created.

4.1 Main menu scene

Even though the menu scene was not created first during the implementation of the project, since it is the first scene that the user interacts with right after the launch of the application, we will start by describing it first.

The menu scene consists of four clickable buttons, which work as portals that set the dimensions of the rendering layers and that transfer the user to the virtual environment, and a big wall that works as a background. Depending on the button clicked by the user the rendering layer's dimensions are set accordingly. This menu scene is created by adjusting the scale of a primitive cube object to meet our needs.

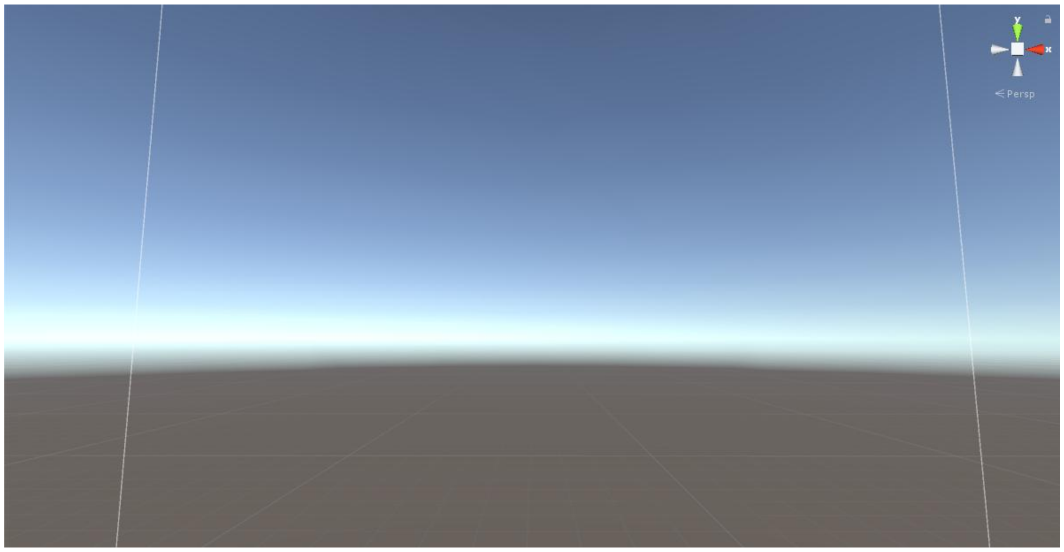


Figure 36: Created background wall without any content

Having created the wall model, a pre-downloaded image is then assigned to it, in order for it not to be completely transparent. By assigning an image to the properties of the wall, it depicts the image's context.

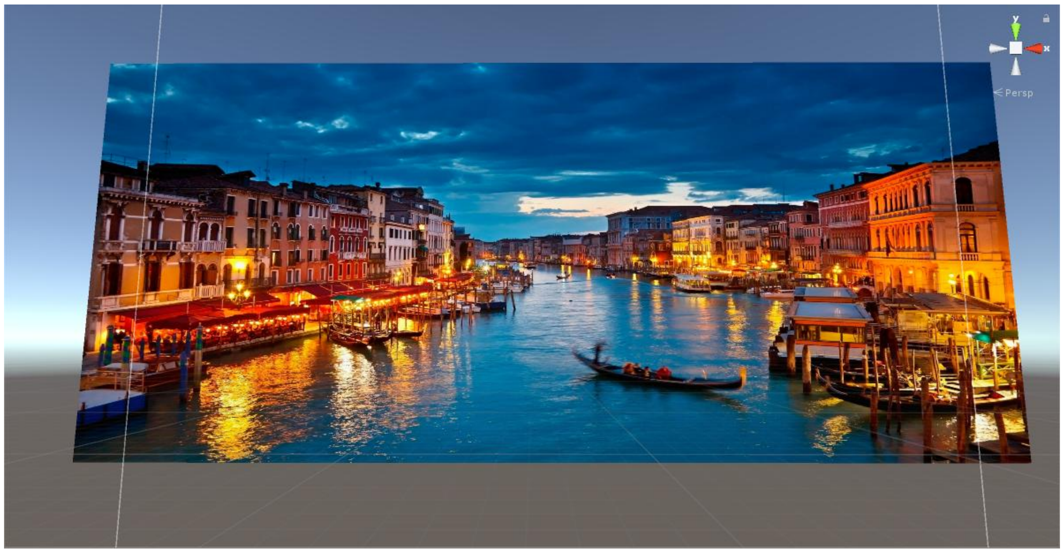


Figure 37: Menu scene with no buttons

We now have our background ready. We then add four clickable buttons one for every algorithmic version needed. These buttons are created by using a void button option available in Unity and by adjusting the text to fit our needs.

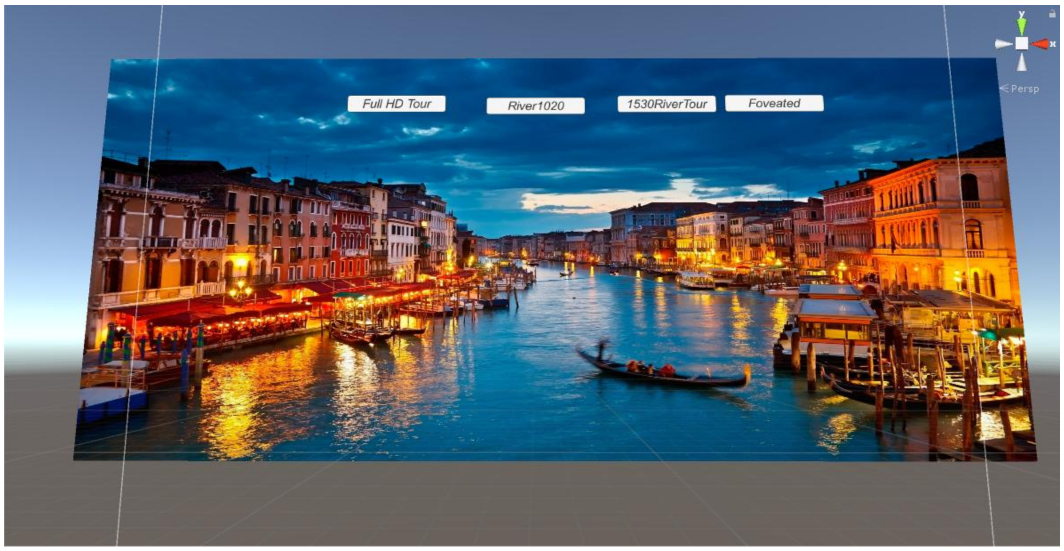


Figure 38: Complete menu scene

Moreover, in order to make it easy for the user to understand which button he is about to click we add an extra feature that changes the color properties of the hovered button from white to yellow.

Finally, since these four buttons need to be able to transfer the user from the main menu scene to another scene, we needed to add an extra feature that would link the event of clicking a button with the event of loading a scene. To do so, we first needed to create four different applications, each containing one of the four needed scenes. As mentioned earlier, although these apps contain the same virtual scene, we need to create more than one app because we want to apply a different foveated rendering method to each of them. Then we need to link the new applications to the menu scene. In order to achieve that, we need to load all the four application scenes in the "build options". This will make it possible for Unity to "understand" that this four scenes and the menu scene are somehow connected and they need to be build on a mutual application. Finally, we need to add a little component in each button with an "on call function()". This function will be called once the button, that it is attached to, is clicked. The script for these functions is not complicated since the only thing that it needs to be able do is, after the button being clicked, to start loading the scene that matches the given scene-name.

Next we will analyze the scene structure which will be loaded after one of these buttons is clicked.

4.2 Game scenes

These four applications that were mentioned in the last subchapter contain the exact same virtual world/scene. It is important for the project that these four scenes are precisely the same, because we want to be able to compare how different sets of parameters given to the same foveated rendering algorithm, affect the frame per second rates of the scene. These parameters change the dimensions of the rendering layers. However, we will discuss in detail about the reasons that we chose to do that as well as how we achieve to do that in the next chapter.

Since these four scenes are the same we will explain how the first one was created only.

While creating a virtual world using Unity, the start is nearly always the same. Everything starts by creating a terrain. The terrain is the map on which everything will take place into. At the beginning the terrain is small and empty and flat.

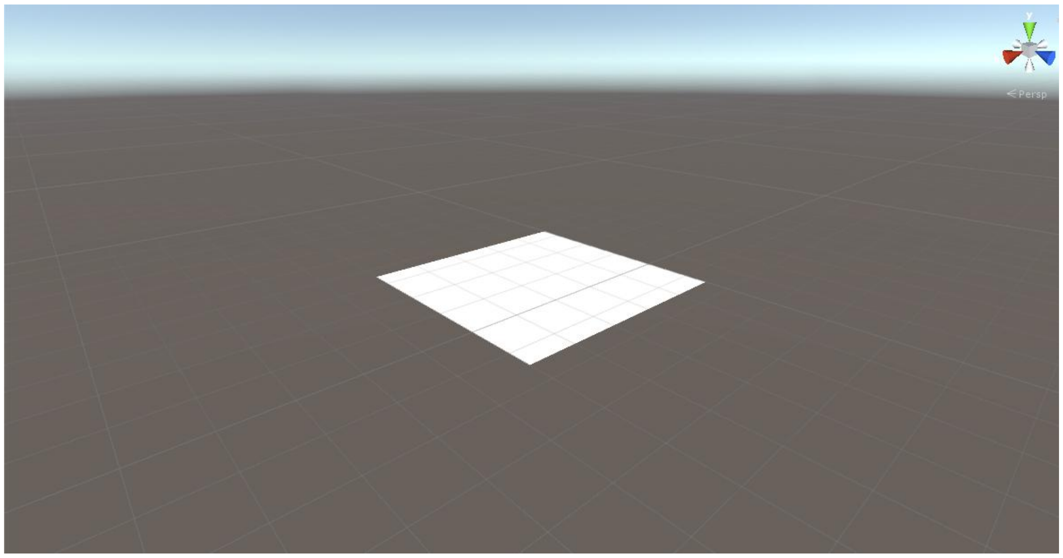


Figure 39: Empty terrain

The first thing to consider is adjusting the scale of the terrain to fit the project's needs. Additionally, there are tools that allow us to adjust the height levels of the terrain, allowing us to create some low-height mountains and in general the different height areas of the map that are needed. Using the so called "brushes" in Unity's terrain toolkit, the terrain is firstly transformed as shown below.

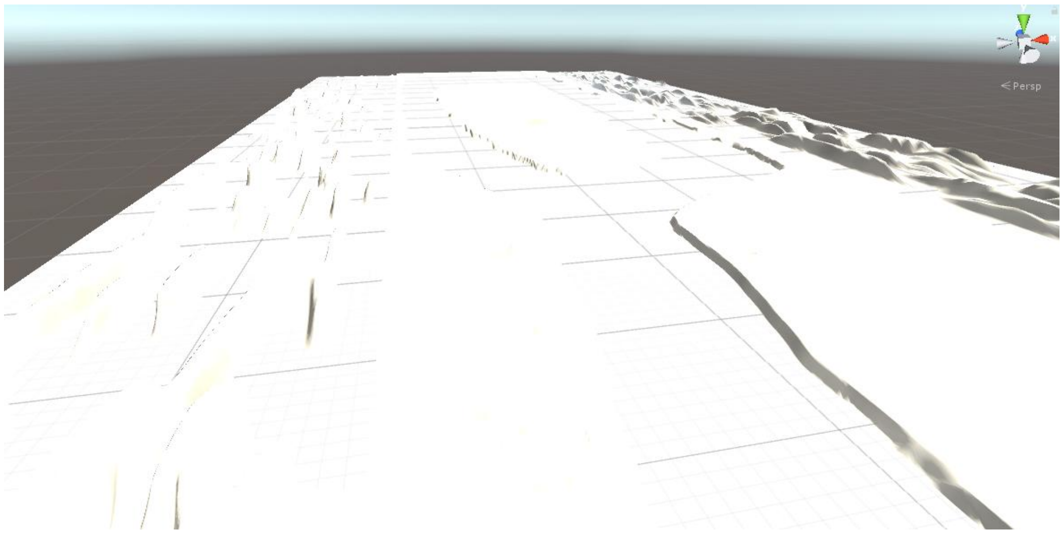


Figure 40: Picture right after the first adjustments were made

It is clear that at this point, there are no colors no objects any textures and generally the whole map is empty. So, the next step is to start filling the map of our game. Since the general idea for the game was to make the user think that he is a gondolier that takes customers in his gondola and transfers them wherever they want to go, in his small village, we decided to create a small village that is cut in two by some sort of river.

In order to find the objects needed, we visited many online repositories. We searched online for house and tree models that suited our needs and imported them to our map.

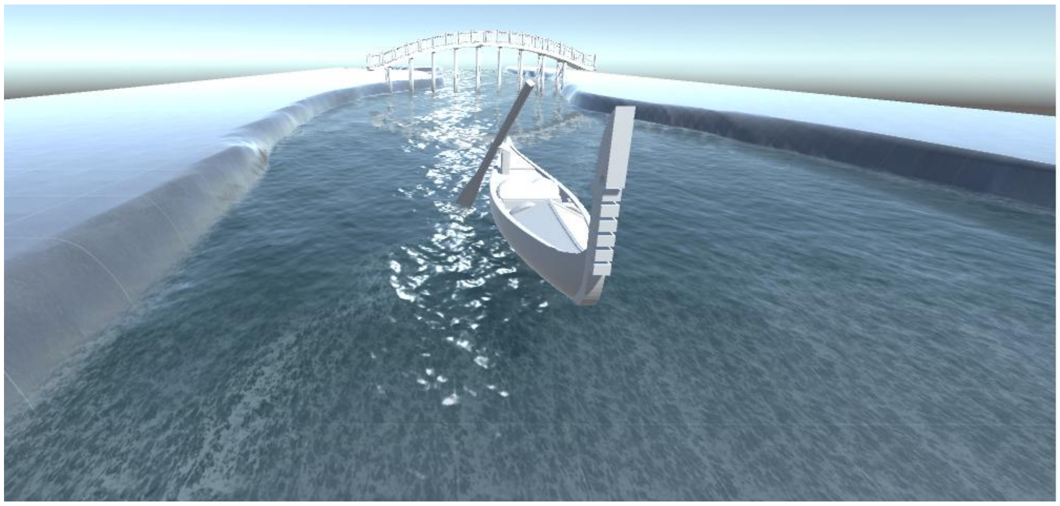


Figure 41: A gondola and a footbridge just imported

When an object is imported into Unity, most of the times, it looks like the gondola and the footbridge that were just imported in the figure just above. In order to make the user’s experience more pleasant, we need to add some colors or better yet some materials on these objects’ properties. There are two options in order to achieve that. Either we create the material on our own, or we download them from other developers. In this thesis, most of the materials that were not imported with the objects were created manually. By doing so we can chose how sensitive will this materials be to light, how much they reflect on water and more.

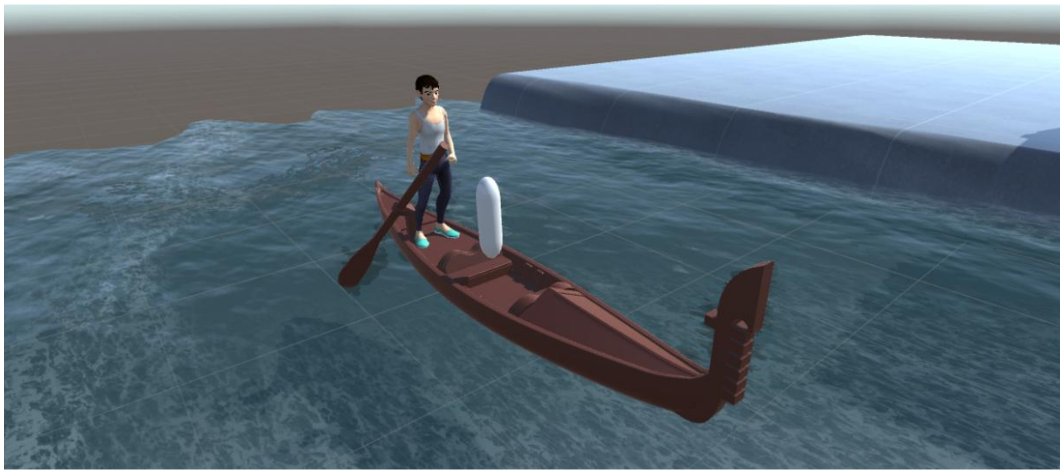


Figure 42: The result of adding materials to the gondola object

The same thing that was done for the gondola was also done for the rest of the objects on the map too. In some cases, we needed to modify some objects because of their massive level of detail. When an object is too detailed it slows the whole application down, and that’s why it is necessary to modify or even delete such objects.

Apart from the houses, the buildings, the trees, the water and the bridge we also added a customer, a customer’s friend and some golden coins. We added the customer because in order for the user to have a purpose on this game he needs to board a customer on his gondola and take him to his destination.

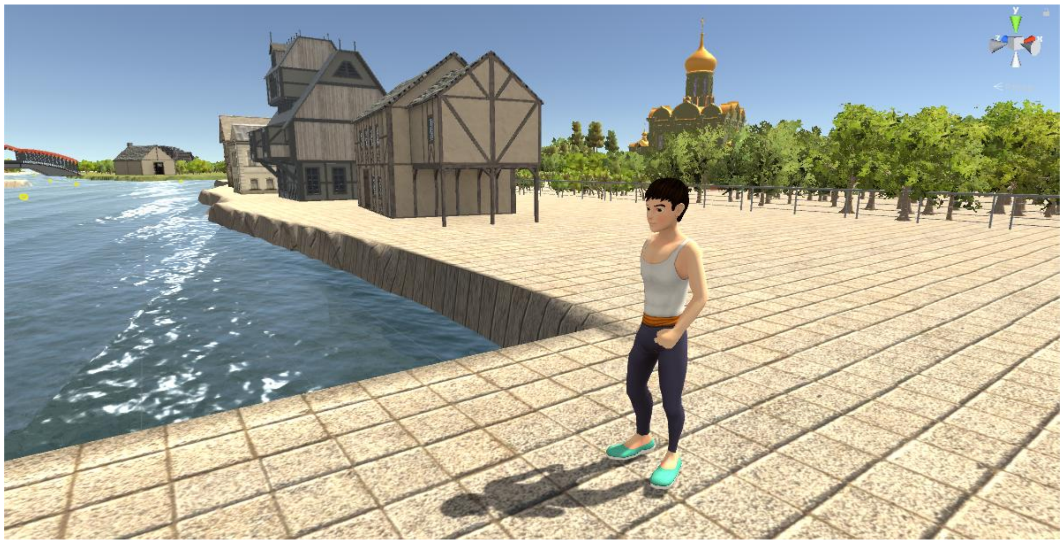


Figure 43: The village resident and the User's first customer

Moreover, we added a customer’s friend so that the customer can ask from the user to take him meet his friend, as a possible desired destination.

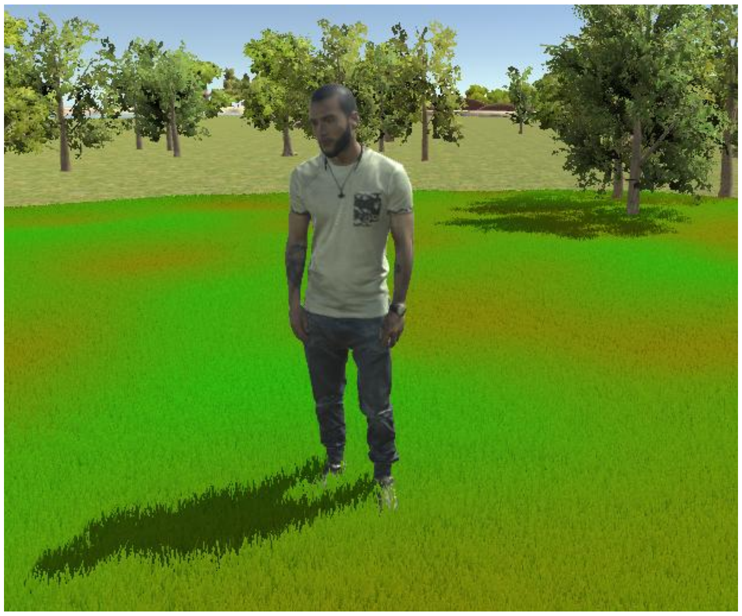


Figure 44: The one possible target

And finally, we added a number of golden coins that appear on the map, after the customer has expressed his desired destination, in an attempt to make it easy for the user to find the target location.

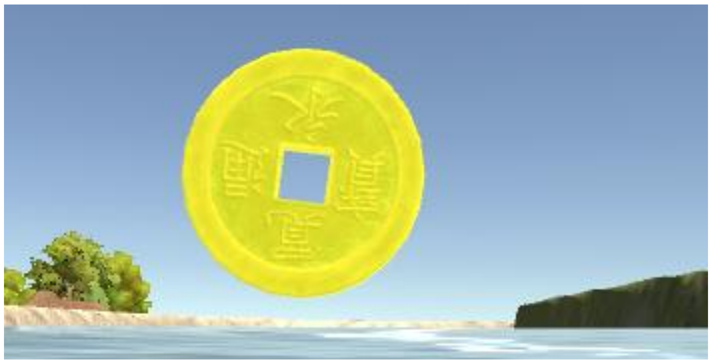


Figure 45: The Chinese golden coins

There are two possible destinations that the customer can name. The first, as mentioned, is the area that his friend lives and the second one is on the other side of the map, and it is where his house is located.

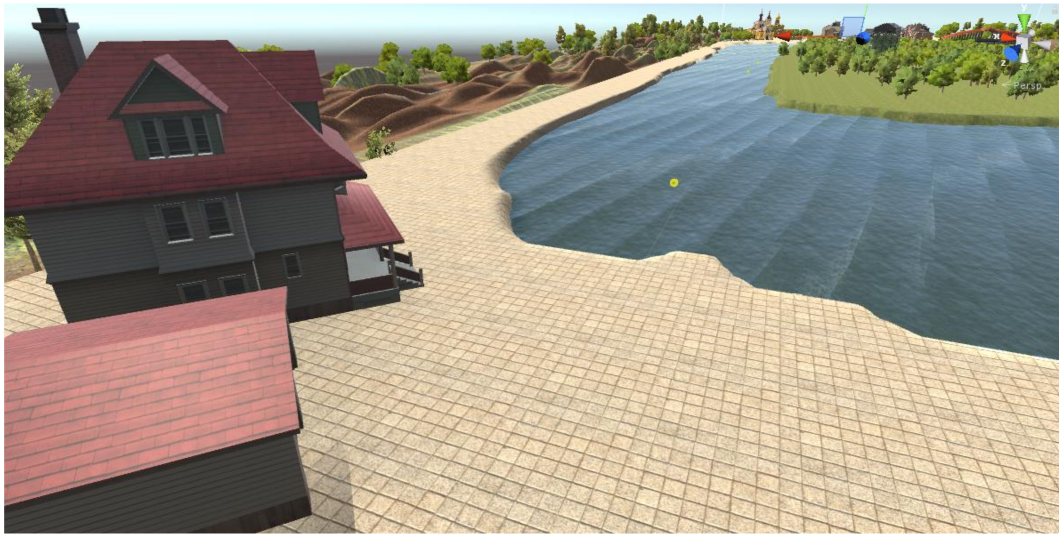


Figure 46: The second target area

The map created is very big and although we have decided to make the gondola move quite fast, especially for a gondola, it takes the users two to three minutes to complete the ride.

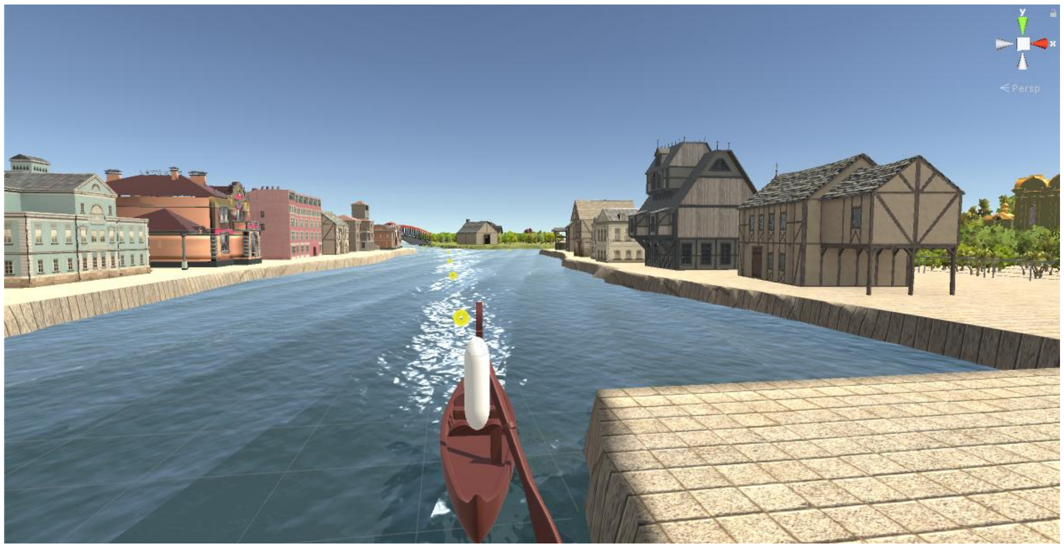


Figure 47: Starting point

As you may see in the figure above none of the destinations is visible from the starting point. This is why small golden coins appear to indicate the correct path for the user to follow.

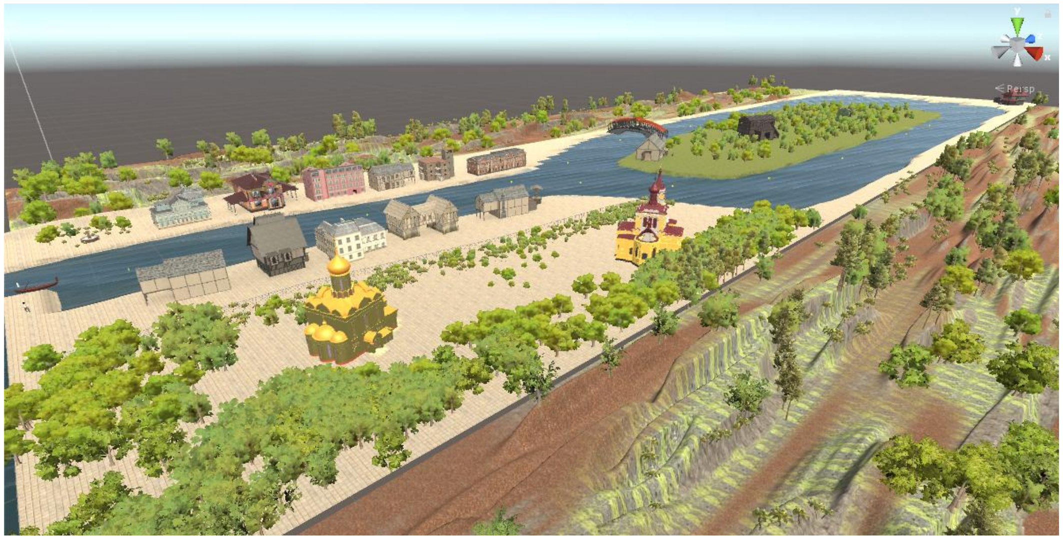


Figure 48: View of the virtual environment

In order to make the game more realistic we added the force of gravity to the whole environment. As a result, every object that is not a static object, such as the customer, the gondola and the User's avatar, is getting pulled towards the terrain. Unity, applies "physics" or the force of gravity in our case, to every object that has a particular component. This component is called Rigidbody. Consequently, clear boundaries should be implemented for every object. If we do not indicate were exactly every object starts and ends, objects would be able to go through one another after being pull from the gravity force. These boundaries are called colliders.

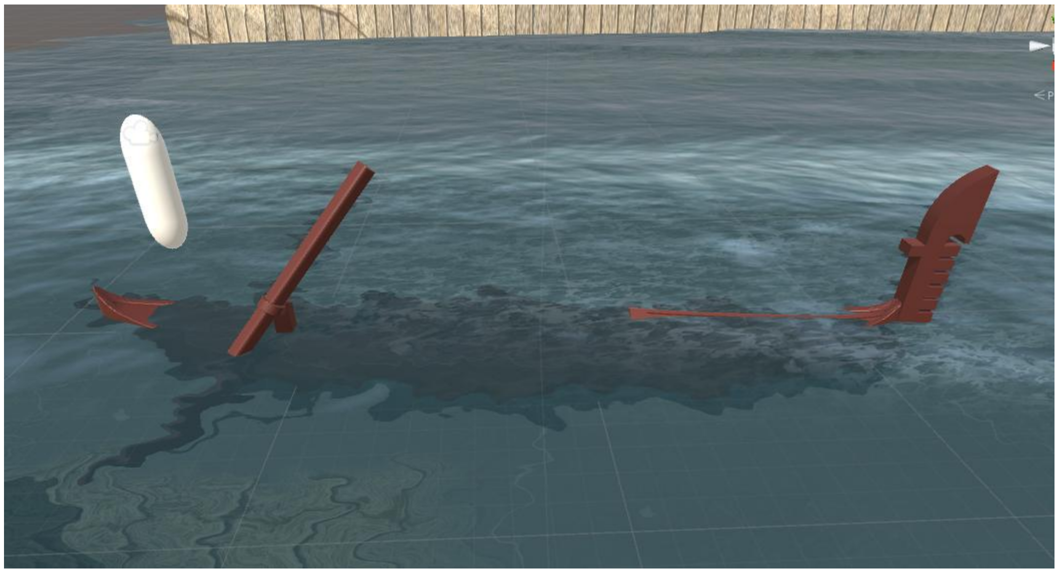


Figure 49: Example of what would happen if colliders were not implemented

There are three different ways to create colliders for an object. Firstly, it is possible for some object to access their prefab and then generate the colliders from there automatically. Secondly, one can add a component on the object called mesh collider and link it with an external source that indicates the right shape of the collider. Finally, it is sometimes preferable to create a box collider around the object. This can be beneficial for some objects which are very detailed but their shape is similar to a box. By creating this box collider, it can lead to a significant performance boost.

In our efforts to make the environment as realistic as possible, we chose the technique of real time lighting instead of baked lighting. This means that as lights and GameObjects are moved within the scene, lighting will be updated immediately after every frame. In the case of baked lighting, shadows and reflections would be calculated only once, only at the start of the game.

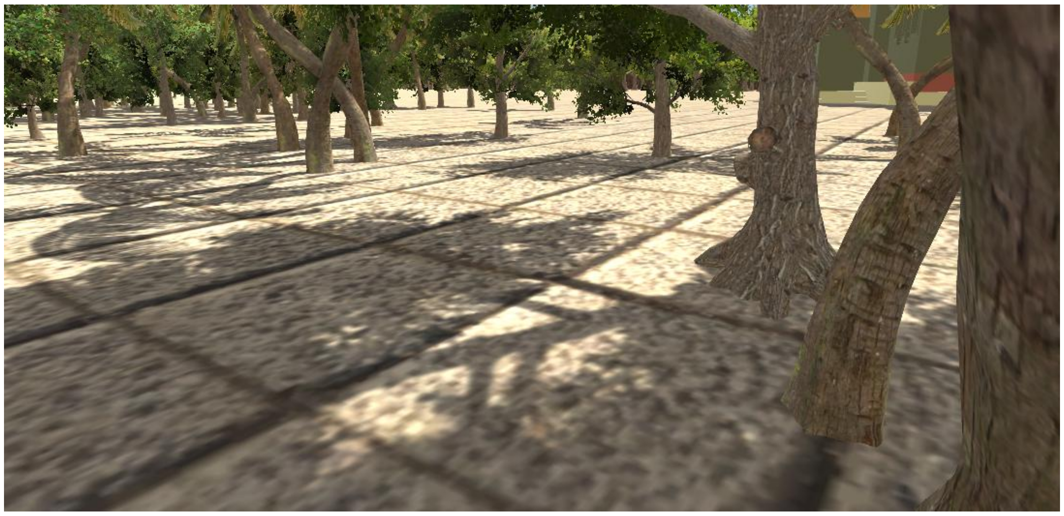


Figure 50: Example of shadows

Chapter 5 Implementation

In this chapter, following to the user interface's implementation, we will continue by explaining the implementation and development of the application and its components.

5.1 Achieving stereoscopy

The HMD used in this thesis, the Nvis SX 111, uses the partial overlap method in order to achieve stereoscopy. This means, that even though the HMD possesses two displays with 76 degrees Field of View (FOV) per eye, the total FOV for both eyes is not 76 x 2 = 152 degrees FOV but 102 degrees. Consequently, 50 degrees FOV is depicted in both displays, in both eyes.

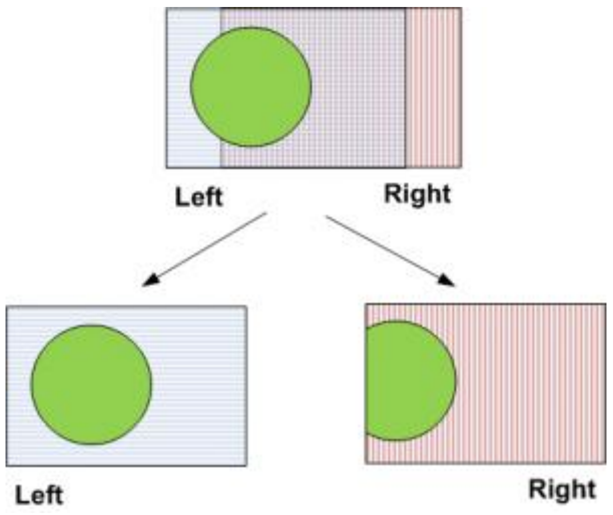


Figure 51: Example of the partial overlap method, where the final result is the combination of Left and Right

However, the HMD does not automatically adjust the input image signal to match that requirement. As a result, the computer's output image signal must be formatted based on the HMD's specifications.

In order to achieve that, we implemented our application to produce two different output signals, one for each display/eye. Furthermore, since the HMD requires SXGA resolution to work our implementation should meet this specification too. Based on the SXGA format, each camera of the HMD has a resolution of 1280 x 1024 pixels, amounting to a total of 2560 x 1024 pixels for the display of the HMD screens. As a result, we added two cameras on our application too; one for each display of the HMD with a resolution of 1280 x 1024 pixels each. This change from the default camera resolution was done by adjusting the project's settings within the unity editor.

We position these two cameras in front of a plane and their recordings constitute our two final video signal outputs from the application to the HMD. A plane is a flat model on which materials and textures can be applied on. In our case, the plane used contains two smaller inner planes. These two smaller planes are necessary in order to depict on each of them the video signal produced for each eye from our algorithm. We achieve this projection by adding one material component to each plane. Furthermore, we needed to create two render textures and link them with these two added materials. These render textures are special types of textures that can be created and updated at runtime. By using the render textures on these two materials and by updating them per frame, the render textures depict the two video signals on the planes. Exactly the same method is currently used in closed monitoring systems.

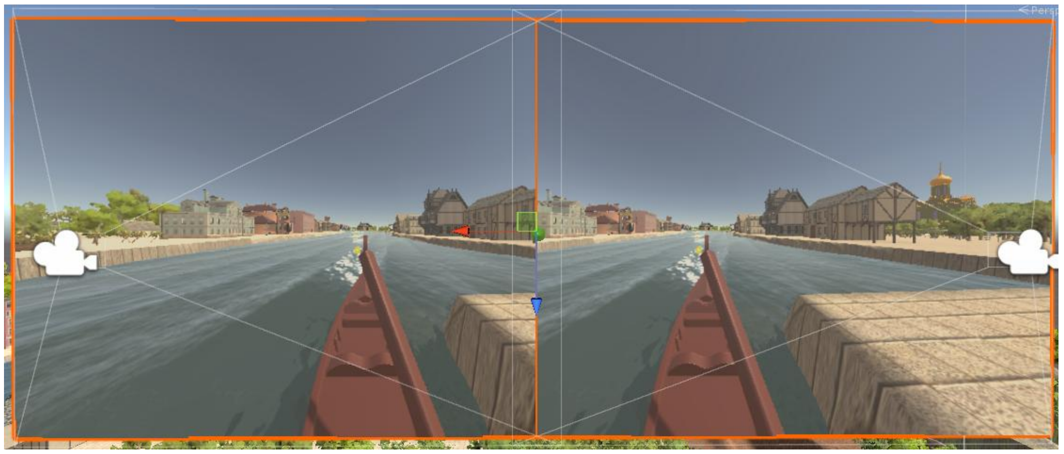


Figure 52: Snapshot of the two cameras recording the render texture panels

These two render textures that are applied on the two inner-planes, despite having a high resolution of 1280 x 1024 pixels and maximum anti-aliasing and color depth formats, do deliver the desired foveated rendering needed. The way we achieve this will be explained later on in this chapter.

5.2 Generation of the partial overlapping effect

As we mentioned before, the HMD used in this thesis uses the partial overlap method in order to achieve stereoscopy but it does not adjust the image automatically. As a result, it is up to the developer to generate the appropriate input. Furthermore, we explained that, in our approach, we are using two cameras in order to deliver to distinct video signal inputs for the HMD's displays. These two cameras are located in front of a panel which has applied to it two render textures and the cameras record its content. In this subchapter we will explain how we create the necessary feed that the two render textures on the panels are projecting.

In order to be able to deliver the video signal formatted based on the specifications of the HMD, we use a camera with a massive FOV. We are using one camera that provides the information needed for both eyes, instead of using two cameras, one for each eye. The benefits of this approach will be explained later on. This camera is attached to the user's avatar and follows its every move in the virtual scene. By doing so, we are able to simulate what would the user's FOV be, if the user was really immersed in the virtual environment.

Since the HMD used offers a FOV of 102 degrees, we created the camera simulating the user's eyes, to have the corresponding FOV. Once the camera was created, we wanted to store the data gathered from it, in order to process them. In order to be able to store the camera's feed we attached a render texture to it. In Unity, if a camera is not attached to a target texture, in our case the render texture, before the start of the application, its feed will try to be projected immediately to the screen. In our case, it was necessary to store the data, firstly, in order to process them and create the feed for both eyes and secondly, because we want to further process them so that we can deliver the desired foveated result.

In addition to the render texture attached to the camera, we need to create another render texture. This secondary render texture is necessary, because the data of the camera constantly change, giving inadequate time for the algorithm to process them. Consequently, we copy the data from the first render texture to the second. By processing the data of the secondary render texture only, we neglect some frames of the video signal generated from the camera, which go unnoticed by the users, creating enough time to process the data. Since performance is of course very crucial for our application, we do not use common methods of copying the first render texture to the latter. Instead we are using a function that assigns this process to the GPU instead of the much slower CPU.

As mentioned before, by carefully examining the specifications of the HMD, we were able to produce the video signals needed for both the displays of the HMD from a single camera. This was achieved by calculating the FOV that each display can depict on its own. We calculated that every display has a FOV of 76 degrees, and since the overlap of the two displays is 50 degrees, having a sole camera of 102 degrees was enough. 102 degrees are indeed enough to include both of the camera's FOVs (76 x 2 = 152 degrees) minus the overlap FOV (152-50=102 degrees).

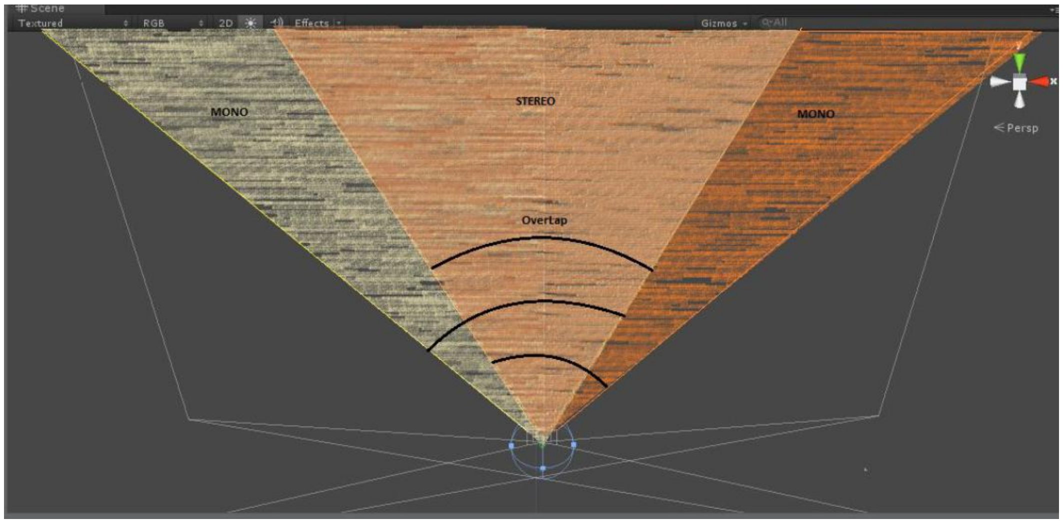
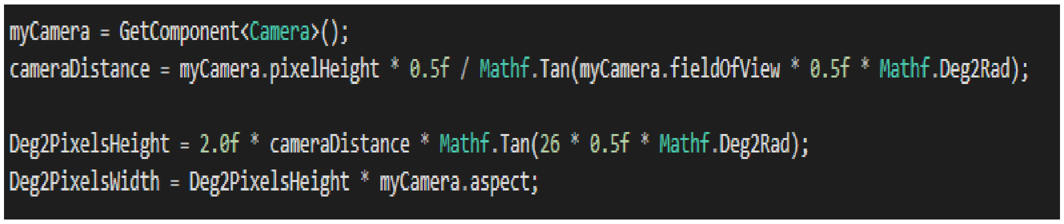


Figure 53: Stereoscopic cameras' field of view based on the partial overlap method used by the Nvis SX 111

However, having one render texture that includes data for 102 degrees of FOV was not enough. Finally, it was necessary to transfer these data into two smaller render textures. This two smaller render textures are the ones applied on the panel that the two HMD cameras are looking at. In order to copy parts of the concentrating render texture to the two smaller render textures, we had to calculate how many pixels correspond to a FOV of 26 degrees. This was needed for stimulating the user's right eye's feed. We needed to omit the pixels that correspond to the first 26 degrees of the concentrating render texture, because the right eye is not able to detect further left of the overlapped area. Respectively, the feed for the user's left eye was stimulated by omitting the last 26 degrees of FOV.

In order to calculate the pixels that correspond to a FOV of 26 degrees, the following code was created.



```
myCamera = GetComponent<Camera>();
cameraDistance = myCamera.pixelHeight * 0.5f / Mathf.Tan(myCamera.fieldOfView * 0.5f * Mathf.Deg2Rad);

Deg2PixelsHeight = 2.0f * cameraDistance * Mathf.Tan(26 * 0.5f * Mathf.Deg2Rad);
Deg2PixelsWidth = Deg2PixelsHeight * myCamera.aspect;
```

Figure 54: The code that calculates the pixels needed to depict a FOV of 26 degrees

This code takes under consideration the field of view of the main camera and the height in pixels of the same camera to firstly calculate the distance in which the camera renders. Afterwards, by using this result it calculates the height, in pixels again, needed to correctly depict a different FOV from the same camera or another. Finally, it calculates the width in pixels needed in order to maintain the camera’s aspect ratio.

By implementing the approach that is using one camera instead of two, we are saving processing power and system latency, because we are creating two less render textures than if we were to use two cameras, one for each eye. In addition, we will explain later in this chapter that in order to create the desired foveated result, we will add two more cameras to the left eye. If we were to use the approach with separated camera for each eye, we would need to use two extra cameras. Two extra cameras means that we would need two extra render textures for each. In other words, this approach reduces the number of render textures needed by six.

Of course, in the framework of this diploma thesis, we did not come up with the optimized algorithm at once. At first, the algorithm implemented used a different camera for each eye.

5.3 Foveated rendering

After having explained how we can send the necessary information for both eyes to the HMD, according to its specifications and how we achieve the implementation of the partial overlapping method, we can now begin to describe the implementation of the foveated rendering algorithm.

In order to be able to create a foveated rendering algorithm, we must first have the capability to recognize precisely on what point the user’s gaze focuses on. In order to do so, we must first assign to the eye-tracking application the area that we are interested in recording the user’s movements. Once this has been done, we must calculate the user's focal point using the information obtained from the tracker. Finally, we create the foveated rendering result, based on the calculated focal point.

5.3.1 Setting the region of interest

As we mentioned before, we are using the Arrington’s eye-tracking software to provide us with the exact location of the user’s focal point. How the connection between the eye-tracker and unity is achieved will be explained in the next subchapter 5.4.

One of the most important parameters that need to be customized for the software to work effectively is to define the area of interest. As an area of interest we refer to the area, that we manually indicate, in which the software will be recording the user's pupil movements.

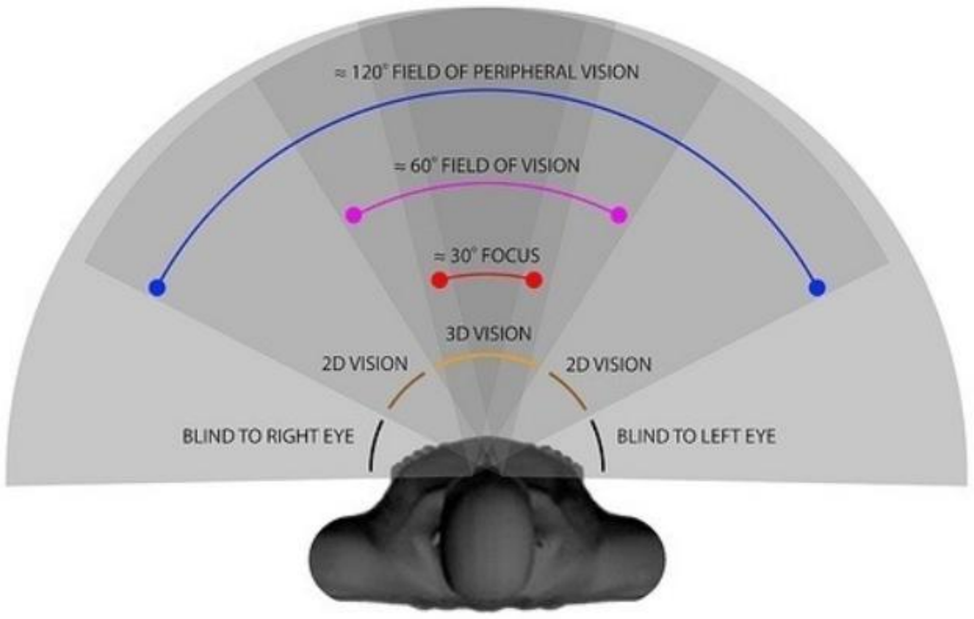


Figure 55: Binocular Field of View

The region of the display that we characterized as important to the application was the area in which we anticipated the user's gaze to be most of the times. In order to decide on the region of interest, we had to run several experiments because the way this HMD works in addition to the extremely big field of view that it offers, made it impossible for any user to detect a small part of the screen located in front of their noses. Furthermore, since we are using monocular vision detection we omitted a small FOV of 26 degrees to the far left of the monitor that the right eye could never detect. As a result, we characterized almost 60 per cent of the display's width as important. This translates to a field of view of 40 degrees horizontal. As far as the height of the screen is concerned, we omitted 30% of the screen, because of its position at the far top and bottom of the screen.

Users are anticipated to slightly move their heads and not just their eyes, in case they want to focus on something that is located further away than this FOV of 40 degrees in front of them. In the following figure we present the region of interest for our experiments the way the eye-tracker presents it, while in the figure after the next one we try to estimate the region of interest while the application is running so that it is easier for anyone to understand it.

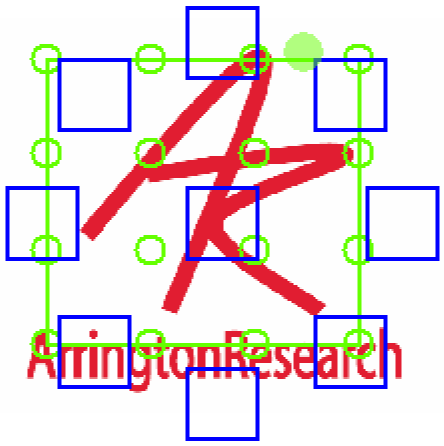


Figure 56: The region of interest is adjusted to our needs and parts of the display are left out of it

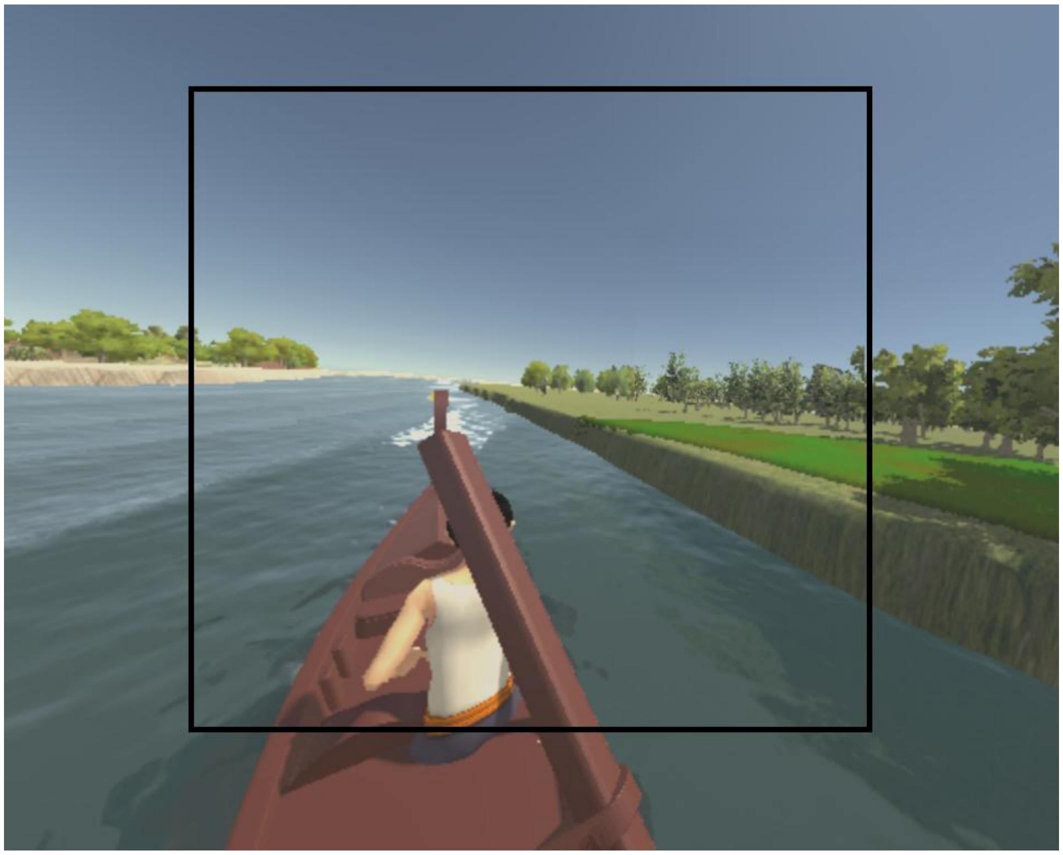
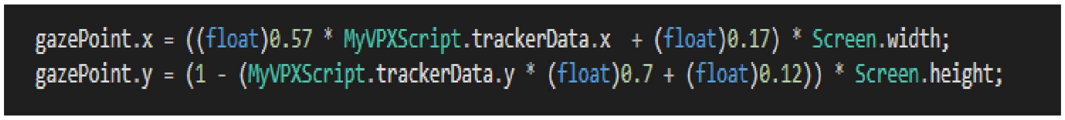


Figure 57: Estimation of the region of interest on the left eye's feed

Any movement of the user’s gaze that is inside the marked area, in the figure above, will be recorded from the Arrington’s eye-tracking software and will be sent to our application at a frequency of 60 Hz. If the user looks at the lower left edge of the tagged area, the eye tracking software will send the value pair (0.1) indicating that the user's gaze is located at the beginning of both the X axis and the Y axis. Respectively, the top right corner is denoted by the value pair (1.0).

5.3.2 Calculating the user’s focal point

Since the scale 0 to 1 matches only part of the screen, to find the user's focal point on the whole screen we must apply some kind of normalization of the scale. This normalization can be seen on the following figure.



```
gazePoint.x = ((float)0.57 * MyVPXScript.trackerData.x  + (float)0.17) * Screen.width;
gazePoint.y = (1 - (MyVPXScript.trackerData.y * (float)0.7 + (float)0.12)) * Screen.height;
```

Figure 58: Normalization of the scale provided by the eye-tracking software in screen resolution

Apart from adding the excluded area of the screen to the equation, we multiply the final result with the screen resolution. This is how we convert a simple number between 0 and 1 into a location on the screen. Furthermore, regarding the Y axis we need to also reverse the scale, in order to make it easier to process the data afterwards.

After we calculate the coordinates in the screen that the user's gaze focuses on, we compare them with the last coordinates obtained. If these two coordinates do not differ more than the minimum acceptable threshold, then we do not perform any action and give the user the same picture as on the previous frame. With this little inertia of the permissible motion we manage to eliminate the unnoticeable movements that the human eye constantly makes.

5.3.3 The foveated rendering technique

At this point we have clarified that the eye-tracking software is detecting the user’s pupil and sends us a pair of values to indicate the position of the user’s gaze. The next step is to use these values in order to calculate the exact position of the user’s focal point on the screen.

Depending on the location of the user’s gaze we decide if we are going to apply the foveated rendering algorithm or we will provide the user with a full HD result. If the user's gaze is located outside the area of interest the foveated rendering effect is not applied. The user will be then presented with a full resolution rendering. We are able to do that by creating another render texture and by assigning it to the main camera. By doing so, we are replacing the foveated render textures (we will explain shortly after why we have more than one render textures for the foveated rendering) with the newly created one. This new render texture has totally different parameters regarding the resolution, the anti-aliasing value and the depth of colors. When the user’s gaze returns inside the marked region of the screen, we have to assign the foveated rendering texture back to the main camera so that the foveated rendering algorithm can be applied again.

In the case that the user focuses on an area which is inside the tagged area, we use the information provided to us by the eye-tracking software in order to render three different layers of resolution around his gaze. We will explain thoroughly how the information is delivered to the application from the eye-tracking software in the next subchapter.

It is now time to describe how the different layers of resolution are created, as well as how they are combined together in order to deliver the desired result for the user to see.

Setting the dimensions of the inner layers

So far, we have explained that in order to store the image “seen” by the user in the virtual environment, we use the so-called "render textures". This specific type of texture has the ability to store the information produced by the whole FOV of the camera in which it is attached to, in a certain given number of pixels. In our approach, in order for the three layers that make up the image to have a different quality of analysis, we should initially calculate the dimensions each texture should have, in order to be able to blend perfectly with one another. If these dimensions are not correctly calculated, for each render texture, then the final result presented to the user would not be on a common scale. As a result the objects in the user's focus center would be deformed.

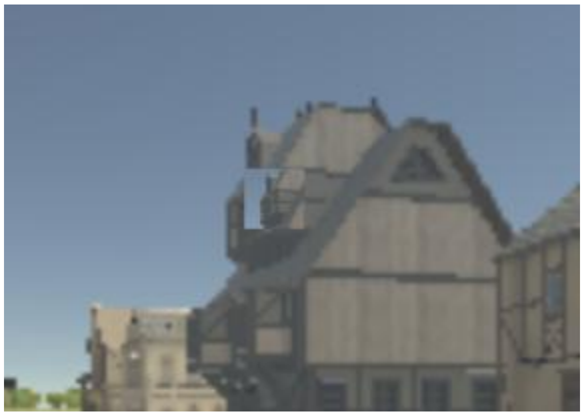
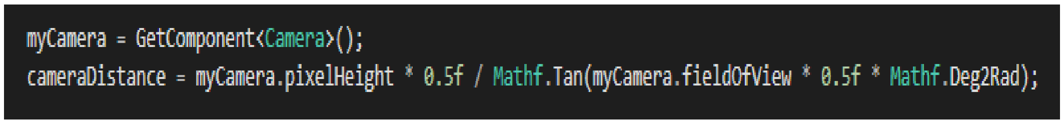


Figure 59: Incorrect dimensions were set on the render texture

It is obvious from the above figure that calculating the dimensions of the render textures correctly is very important for the application and the user’s experience. In the example presented in figure 60, the pixels dedicated for the depiction of the camera’s FOV are way fewer than they should be. As a result, the render texture assigned to the camera included all the information in a small area by changing the scale of the scene. Blending two images rendered in two different scales leads to disastrous effects.

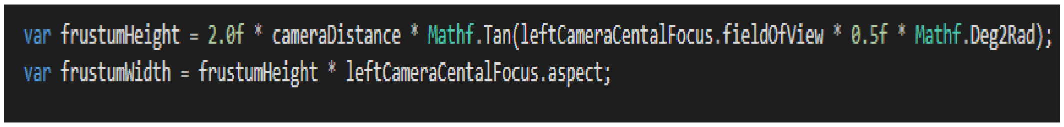
In our script, we are set these dimension by firstly calculating the distance at which the main camera renders (as main camera we refer to the camera with the biggest field of view; the peripheral camera).



```
myCamera = GetComponent<Camera>();
cameraDistance = myCamera.pixelHeight * 0.5f / Mathf.Tan(myCamera.fieldOfView * 0.5f * Mathf.Deg2Rad);
```

Figure 60: Calculating the main camera's rendering distance

Then we calculate the exact amount of pixels needed to depict a different FOV at the same distance from the camera. And finally, we calculate the width needed in order to respect the camera’s aspect ratio.



```
var frustumHeight = 2.0f * cameraDistance * Mathf.Tan(leftCameraCentalFocus.fieldOfView * 0.5f * Mathf.Deg2Rad);
var frustumWidth = frustumHeight * leftCameraCentalFocus.aspect;
```

Figure 61: Calculating the dimensions of the inner layer

These last two variables represent the dimensions that the render texture, which is attached to the central camera, must have, in order to blend perfectly with the other two render textures. By doing so, we ensure that the user is provided with a result attributed to the same scale.

Resolution reduction

In our approach, we are aiming to deliver a result rendered at three different quality layers based on the user’s eye gaze. These three layers are the foveal layer, the middle layer, and the outer layer. In order to do so, we need three cameras that each one somehow renders at a different resolution than the other two. The first step in creating a layer that has a lower resolution than the foveal layer is by setting the dimensions of the render texture, which is attached to the rendering camera, at a decreased set of values than the ones calculated in Figure 62. This way, we force the camera to save the image received at a lower size – scale than it actually has on the virtual scene. The next step is copping the data stored on this texture to another texture which has the correct dimensions. In order to fully cover the dimensions on the new render texture the application copies the data stored on one pixel of the 1st texture to multiple pixels on the 2nd one. This technique is called oversampling.

By using this technique we decrease the number of samples per pixel used to shade one pixel on our layer. The layers that we use this technique on are two. The first one is the “central layer” which is the layer between the foveal layer and the peripheral layer, while the second one is the peripheral layer itself. The foveal layer is the layer directly around the user's eye, and as a result, it is attributed to the highest resolution available from HMD. The central layer, which is implemented in order for the user to be unable to detect the massive resolution reduction between the foveal layer and the peripheral layer, has 40 per cent fewer pixels that its dimensions require to depict at 100% of the resolution. This automatically translates to 40% decrease to the number of samples per pixels. As for the peripheral layer, the decrease rises to 60%.

Finding the appropriate content for the layers

Up until this point, we have explained firstly, that a set of parameters is sent from the eye-tracking software to our application. Furthermore, we have seen that because of the way that this set of values is formatted it needs further process in order for it to depict the exact point that the user focuses on. Additionally, we have explained in which way the algorithm manages to render at different quality levels. Now it is time to discuss, how this layers get the appropriate content from the cameras.

As mentioned already before, we are using three different cameras on our approach. The first camera, which contains the whole FOV that the HMD is capable of depicting, is the peripheral camera. The FOV on this camera never changers no matter what version of the algorithm the user wishes to experience. The other two cameras, that stimulate the user's eyes, have a different FOV depending on the user's choice. Depending on the version of the algorithm, these cameras can have 3 different sets of FOV. Either 5 degrees and 10 degrees, or 10 degrees and 20 degrees or finally 15 degrees and 30 degrees. As the FOV of the cameras changes so does the size of the render textures. This of course happens because a camera with a larger FOV needs a larger render texture in order not to change the scale of the rendering.

These three cameras are being rotated all together when the user moves his head, while only the two of them move when the user moves his eyes. This happens because one can change the spot that they are looking at, either by moving their eyes, or by moving their head. So, both of these possibilities have been implemented on our application.

The head tracker does inform us about the user's head movements by sending us the degrees that he rotated his head. So in order not to complicate things we decided to add a component which would play the role of the parent of our three cameras. This component stimulates the user's head and so it is seemed only logical to name it “Head”. When the head-tracker sends us data regarding the rotation of the user's head, we rotate the head component. As a result, all three cameras are also being rotated as children of the head component.

Unfortunately, the eye-tracking software does only inform us about the user's focal point. So, it is up to us to calculate how many degrees did the user's eyes move after every frame.

To be able to calculate the distance that the user’s eyes travelled in one frame we need to process the current set of values provided to us by the eye-tracking software in order to calculate the position of the user’s gaze on the HMD screen and not just on the region of interest. Then we need to compare that position with the center of the axis. This difference in price will show us the distance to pixels that the user's eyes have traveled in a frame. In addition, by checking whether the distance value is positive or negative on both axes, we can also understand the direction of the movement. Then we have to convert the distance from pixels to degrees because a rotation in pixels cannot be defined. To do this, we use the same technique described earlier. Lastly, we set the cameras to look at the same direction that the main camera looks minus 13 degrees because the left eye is 13 degrees left of the center plus the value in degrees that we calculated that the user's eyes moved. Whether we add or subtract this value, it depends on the direction we want the cameras to rotate.

By doing so, we ensure that the two inner cameras always follow the user’s gaze.

Combining to deliver the final result

After having created three render textures, with the correct dimensions and the correct content, the final piece missing from the puzzle is to correctly combine these three render texture into one and to finally present the user with the desired foveated result.

In order to do so, we need to use again the pre-calculated position, in pixels, of the user’s gaze. We use this location in order to correctly copy the two inner render textures on the larger render texture, which is attached on the peripheral camera. We start copping from the pixel that the user’s gaze is located minus half of the texture’s width or height. By doing so we position the render textures around the user’s gaze.

Finally, in order to transfer two video signals to the HMD, one for the left eye and one for the right eye, we separate the large texture of the 102 degrees into to smaller textures of 76 degrees. The first 76 degrees correspond to the left eye’s feed and the last 76 degrees correspond to the right eye’s feed. Of course there is a partial overlap of 50 degrees according to the HMD specifications already discussed in a previous chapter.

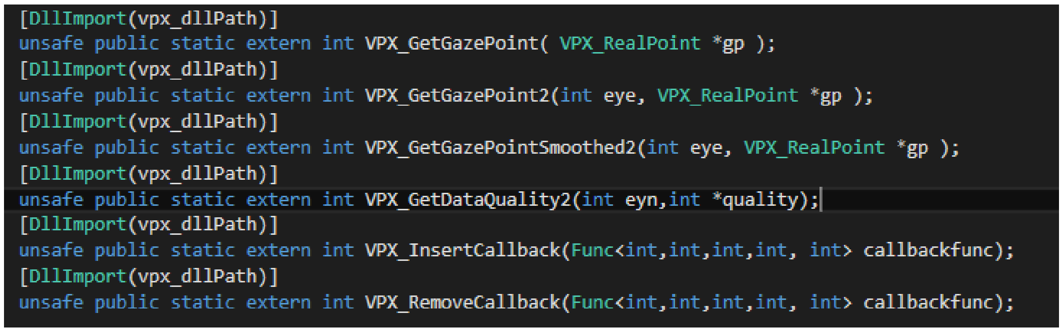
5.4 Communication with the eye-tracker

In previous chapters, for reasons of shortness, we simply reported that the eye tracker sends the information we need directly to our program. This is not entirely true. In this chapter we will explain precisely how the data is delivered to our program.

In order to make the communication of the eye-tracker with third parties applications possible, the Arrington Research Company provides a software developers kit (SDK). Specifically, that comes along with a dynamic link library (DLL) named VPX_InterApp.dll which allows any third party application to interact with the eye –tracking device data. The VPX_InterApp.dll file must be stored in the same folder as the eye-tracking application, in order for the application to correctly connect with the library.

In order to interact with the eye-tracking device, our application needed to register with the dynamic link library. After the registration, the application obtains a unique message identifier used by ViewPoint for inter-process communication. Since the source code of the dynamic linked library is already precompiled, and written in C++, we had to add a new class MyVPX in C# that will bind the library with our application.

C# allows calls to native code from managed applications, in our case Unity3D, through the DLLImport attribute. The DLLImport attribute leads the compiler to declare a function residing in the VPX_InterAPP.dll. So, in our case the code below finds the needed functions from the dynamic linked library.



```
[DllImport(vpx_dllPath)]
unsafe public static extern int VPX_GetGazePoint( VPX_RealPoint *gp );
[DllImport(vpx_dllPath)]
unsafe public static extern int VPX_GetGazePoint2(int eye, VPX_RealPoint *gp );
[DllImport(vpx_dllPath)]
unsafe public static extern int VPX_GetGazePointSmoothed2(int eye, VPX_RealPoint *gp );
[DllImport(vpx_dllPath)]
unsafe public static extern int VPX_GetDataQuality2(int eyn,int *quality);
[DllImport(vpx_dllPath)]
unsafe public static extern int VPX_InsertCallback(Func<int,int,int,int, int> callbackfunc);
[DllImport(vpx_dllPath)]
unsafe public static extern int VPX_RemoveCallback(Func<int,int,int,int, int> callbackfunc);
```

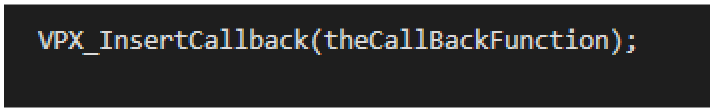
Figure 62: Declaration of functions using the DllImport function

As we can see above, there is an unsafe command in front of each function declaration. C# allows using pointer variables in a function code block when it is marked by the unsafe modifier. The unsafe code or the unmanaged code is a code block that uses a pointer variable. However, unsafe code cannot be compiled by Unity 3D. This was solved by adding three files in our Unity3D project folder, the smcs.rsp file, the gmcs.rsp and the mcs.rsp file, containing only the command shown below.

```
-unsafe
```

Figure 63: The sole command that the three files need to contain

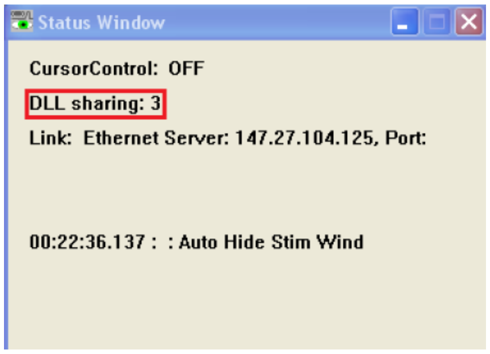
After the successful registration with the Eye-tracking software, myVPXScript.cs class defines a callback function which is stored in the DLL VPX_InterApp.dll. Inside the callback functions limited coding can be used. Since it interacts with native code written in other programming language, C++, only common types such as integers or characters can be defines and used. Otherwise the application will crash. Since, for example, c# tolerates strings in a different way than C++ does, using a string in c# and sending it to a library written in C++ will cause inter-process communication to fail since these languages understand string data types in different ways. Below is the function that establishes Unity and ViewPoint inter-process communication.



```
VPX_InsertCallback(theCallBackFunction);
```

Figure 64: The function that establishes Unity and ViewPoint inter-process communication

The theCallBackFunction function is responsible for the data exchange between the library and the UI's executable file and will be described later on. When a new connection is established the attribute of DLL Sharing, which can be found in the Status Window of the ViewPoint software, increases by 1. This attribute shows the number of third party applications that are registered to the software.

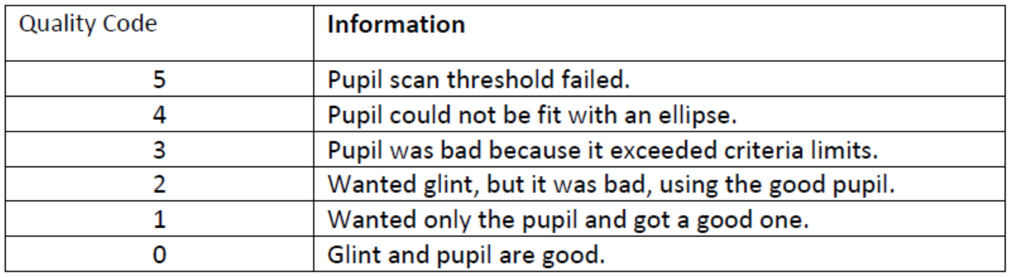


CursorControl: OFF
DLL sharing: 3
Link: Ethernet Server: 147.27.104.125, Port:

00:22:36.137 : : Auto Hide Stim Wind

Figure 65: DLL Sharing

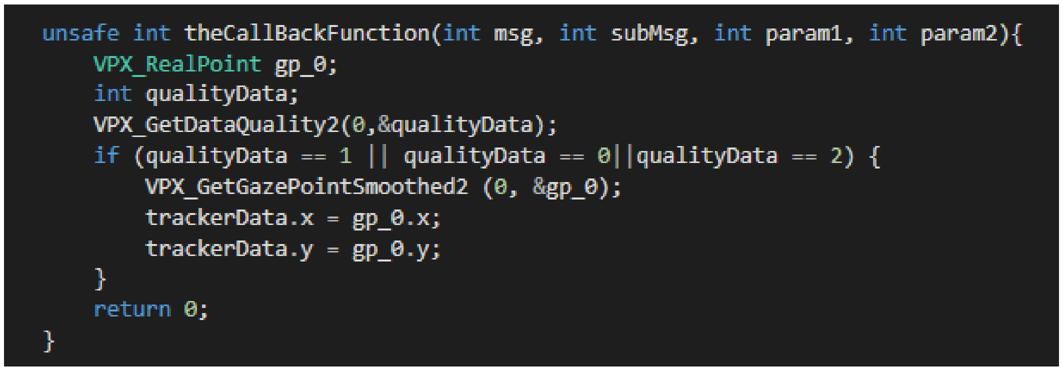
Every time "fresh" data arrive in the ViewPoint application from the frame grabber, the application sends it to all the programs that are registered to it. This happens because the VPX_InterApp.dll calls every function that was defined as callback on each application, passing the "fresh" data. As "fresh" data we refer to the message which informs the applications that new data are available regarding a change on the eye fixations or data produced by an infrared camera, and generally useful data regarding the eye. In our algorithm, every time "fresh" data are available quality checks are made using the VPX_GetDataQuality2() function of the VPX_InterApp.dll, and depending on the quality code returned, gaze data are fetched using the VPX_GetGazePoint2 () function or not.



Quality Code	**Information**
5	Pupil scan threshold failed.
4	Pupil could not be fit with an ellipse.
3	Pupil was bad because it exceeded criteria limits.
2	Wanted glint, but it was bad, using the good pupil.
1	Wanted only the pupil and got a good one.
0	Glint and pupil are good.

Figure 66: Possible quality check results

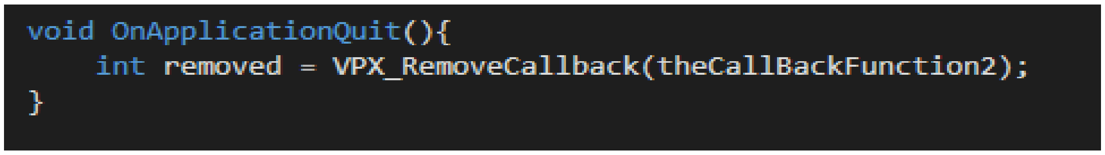
Depending on quality code, data is either fetched or discarded. Quality codes “0”, “1” and “2” are considered as good data and fetched. On the other hand codes “3”,”4”,”5” are discarded. Possible reasons creating these errors are either the user’s blinks, or inappropriate eye movements that possibly were made, or that the infrared camera was instantly shaken due to sharp spin of the HMD.



```
unsafe int theCallBackFunction(int msg, int subMsg, int param1, int param2){
    VPX_RealPoint gp_0;
    int qualityData;
    VPX_GetDataQuality2(0,&qualityData);
    if (qualityData == 1 || qualityData == 0||qualityData == 2) {
        VPX_GetGazePointSmoothed2 (0, &gp_0);
        trackerData.x = gp_0.x;
        trackerData.y = gp_0.y;
    }
    return 0;
}
```

Figure 67: The Code that checks if it is okay to receive gaze data

In the end, when we exit the application, we must end the binding that was made with the dynamic linked library of the ViewPoint software. This action is mandatory because every time we close our application without ending the connection, the tracker keeps sending data causing it to crash.



```
void OnApplicationQuit(){
    int removed = VPX_RemoveCallback(theCallBackFunction2);
}
```

Figure 68: Smoothly closing the app

5.5 Communication with the Head-tracker

In a similar way, data is passed from the head tracking device through the InterCube3 software’s SDK using a link with the tracker’s dynamic linked library; isense.dll. The DLL file must be saved inside the System32 Windows directory.

The two main classes that are responsible for the communication between the two applications are the StereoCamLook and the HeadTracker classes. Below we will explain how connection is established and how data is retrieved from the device. The *HeadTracker* class is responsible for establishing the communication between the two applications and retrieving the data from the tracker while the StereoCamLook class is responsible for passing the data to our application.

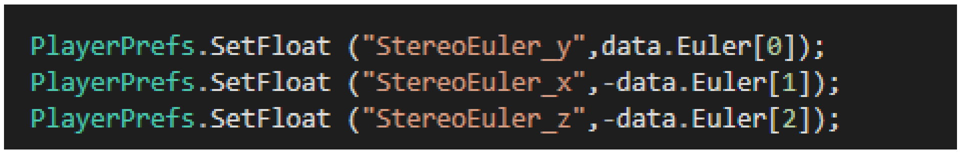
The dynamic linked library is accessed with C#'s DLLImport attribute, exactly as described previously. Below, the function declaration is shown:

```
public const string isense dllPath = "C:\\Windows\\Sytem32\\
isense.dll";

[DllImport(isense_dllPath)]
unsafe public static extern int ISD_OpenTracker(
      IntPtr hParent,
      int commPort,
      bool infoScreen,
      bool verbose
   );

[DllImport(isense_dllPath)]
unsafe public static extern bool ISD_CloseTracker(int handle);

[DllImport(isense_dllPath)]
unsafe public static extern float ISD GetTime();

[DllImport(isense_dllPath)]
unsafe public static extern bool ISD_GetTrackerConfig(
      int handle,
      ref ISD_TRACKER_INFO_TYPE Tracker,
      bool verbose
   );

[DllImport(isense_dllPath)]
unsafe public static extern int ISD_GetTrackingData(
      int handle,
      ref ISD_TRACKING_DATA_TYPE Data
   );

[DllImport("isense.dll")]
public static extern bool ISD_GetStationConfig(
      int handle,
      ref ISD_STATION_INFO_TYPE Station,
      int stationID,
      bool verbose
   );

[DllImport("isense.dll")]
public static extern bool ISD_ResetHeading(
      int handle,
      int stationID
   );
```

Figure 69: Declaration of functions

In order to seek for trackers connected to the computer we are using the *ISD_OpenTracker* function. If a tracker is detected then a timer is set to count the connection duration and via the *ISD_ResetHeading* function synchronization to the tracker is achieved. After a successful connection, head tracking data are passed from the device to our application by calling the *ISD_GetTraackingData* function once per frame. Once the user decides to exit the application, *ISD_CloseTracker* is called to terminate smoothly the connection without crashing.

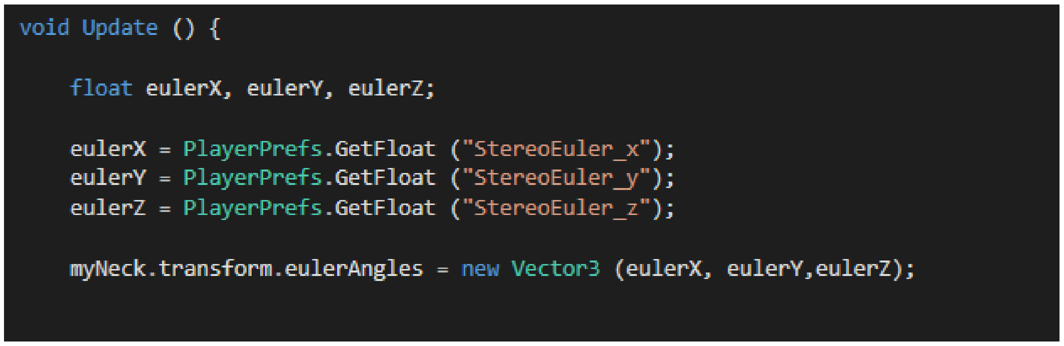
The data are passed from one class to another by using the PlayerPrefs attribute that is allowed in Unity3D.



```
PlayerPrefs.SetFloat ("StereoEuler_y",data.Euler[0]);
PlayerPrefs.SetFloat ("StereoEuler_x",-data.Euler[1]);
PlayerPrefs.SetFloat ("StereoEuler_z",-data.Euler[2]);
```

Figure 70: Setting the data in order to be ready for the next class

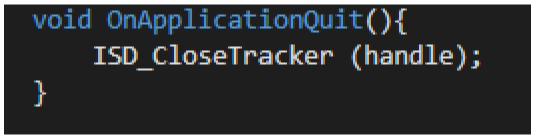
Respectively, the yaw, pitch and roll orientation data that are received from the tracker, are then used by the *StereoCamLook* Class as inputs in order to adjust the head models' orientation, using the PlayerPrefs' Get method



```
void Update () {

    float eulerX, eulerY, eulerZ;

    eulerX = PlayerPrefs.GetFloat ("StereoEuler_x");
    eulerY = PlayerPrefs.GetFloat ("StereoEuler_y");
    eulerZ = PlayerPrefs.GetFloat ("StereoEuler_z");

    myNeck.transform.eulerAngles = new Vector3 (eulerX, eulerY,eulerZ);
```

Figure 71: Retrieving the data from the head-tracker

In the end, when we exit the application, the *ISD_CloseTracker* function is responsible to terminate the connection.



```
void OnApplicationQuit(){
    ISD_CloseTracker (handle);
}
```

Figure 72: Smoothly terminating the connection with the head tracker

5.6 Audio

"Sound is what truly convinces the mind that is in a place; in other words, hearing is believing."- The Art of Game Design, Jesse Schell. Based on the previous quote it is perfectly clear that a game without audio would be totally incomplete, it would lose the connection between the player and the environment. So, audio is a valuable and necessary part of a game.

In real life, sounds are emitted by objects and heard by listeners. The way a sound is perceived depends on a number of factors. A listener can more or less tell which direction a sound is coming from and may also get some sense of its distance from its loudness and quality. A fast-moving sound source (like a falling bomb or a passing police car) will change in pitch as it moves.

Also, the surrounding environment will affect the way sound is reflected, so a voice inside a cave will have an echo but the same voice in the open air will not.

To simulate the effects of position, Unity requires sounds to originate from Audio Sources attached to objects. The sounds emitted are then picked up by an Audio Listener attached to another object, most often the main camera. Unity can then simulate the effects of a source's distance and position from the listener object and produce them to the user accordingly.

For the purposes of this project, the audio listener is attached in the main camera, which constantly follows the user's avatar, and the sound is produce from a location close to the start of the river.

Chapter 6 User trials

In order to test the foveated rendering algorithm implemented, we decided to conduct a user study in order to monitor the FPS rates that is virtual scene is rendered at. This thesis' main goal is to prove that the method used can provide beneficial performance results, in terms of frames per second, meaning that it can make the user's experience more pleasant and the game experience smoother than when the algorithm is not running in the background, by increasing the number of frames rendered per second.

6.1 Experimental procedure

When the experiment starts, the user is asked to wear a specific head mounted display machine, the Nvis SX 111. The first step is to calibrate the machine and the eye tracking software (Viewpoint) specifically according to the user's head geometry and his eye movements. To do so, the user is asked to concentrate on sixteen different locations on the screen, consecutively. Every few seconds, a green square pops in the user's screen and the user must simply look at it. After, a few moments, the program takes a quick photo of the user's pupil. By doing so at every different focal point it keeps in its "memory", data regarding the status of the user's pupil in every different position.

After the calibration is done, the user enters a specific virtual environment in which the user takes the role of a gondolier. Right when the game starts, the user meets his first customer approaching him slowing from his right side. When the customer boards the gondola, a message appears on the screen notifying the user about the customer's desired destination. The destination is chosen randomly every time between the two available options. After that, golden coins pop-up on the map in order to help the user locate the correct destination area. The user simply has to transfer the customer to the correct area on the map. Meanwhile, the user can freely enjoy the ride and the view.

The user will have to make the same ride four times because we want to test the results on four different sets of parameters.

Set	Foveal	Central Cycle	Periphery
1	102^{o}	0^{o}	0^{o}
2	5^{o}	10^{o}	102^{o}
3	10^{o}	20^{o}	102^{o}
4	15^{o}	30^{o}	102^{o}

The first set of parameters indicates that the foveal cycle of the foveated rendering technique that is going to be applied will be 102 ^{o} big. The Nvis SX 111 HMD has a field of view of 102 ^{o}. That means that the whole screen will be covered by the foveal cycle, which means that the result won't be a foveated rendered result, but a full high resolution rendered one. On the second set of parameters the algorithm will create a result where the Foveal cycle will cover 5 ^{o}, the central cycle (which is between the foveal and the periphery) will cover 10 ^{o} and finally the periphery cycle will cover of course 102 ^{o}. Sets 3 and 4 are created accordingly.

At this experiment we want to test if the foveated rendering algorithm created helps the users have a more pleasant experience by providing more FPs than when the algorithm is not used. Furthermore, we want to gather data, regarding the maximum number of FPs, the minimum number of FPs as well as the average number of FPs, in order to have a statistical representation of the results that we managed to achieve. Finally, the users will be asked to fill a sort questionnaire to rate their experience and help us even more with some suggestions they may have.

Also, when the users swap from one set of parameters to another, they will be asked if they cannot anymore detect the boundaries of every cycle or if they couldn't before but now they can. According to scientific research, the foveal cycle can be as small as 5^{o} and stay undetected from the user if the software and the hardware are ideal. Since, in this experiment, we don't have the ideal software or an ideal hardware, we want the users to inform us on how much bigger the foveal cycle must be in order to be undetected. Of course, the bigger the cycle becomes the less beneficial the algorithm will be.

6.2 Exported parameters

The ride takes about 2 minutes to be completed and during the whole duration we check the number of frames per second that are rendered. At the end of every phase of this experiment we save the highest number of FPs, the lowest number of FPs as well as the average number of FPs. In addition, we export a different file which contains the prices number of FPS for every frame. This metrics will help us calculate, after the experiments are completed, how beneficial our approach is.

6.3 Participants & Apparatus

A total of 19 students from the Technical University of Crete, 12 males and 7 females with an average age of 23.9 years, participated in the experiment. All participants had normal or corrected to normal vision. The experiments took place in a dedicated experimental space on the campus, which was darkened to remove any periphery disturbance during the exposure.

The rendered scenes were displayed on an NVisor™ SX111 HMD, having a resolution of 1280x1024 and a Field of View of 102 degrees horizontal and 64 degrees vertical. Participants wondered around the VE using an InterSense™ InertiaCube3™ with 3 Degrees of freedom head tracker attached to the HMD. Eye-tracking data was recorded using a monocular eye-tracker by Arrington Research™ also attached to the HMD updating at a frequency of 60Hz.

6.4 Simulator sickness

A potential side effect of all HMDs is simulator sickness. As simulator sickness we refer to fatigue, headaches, dizziness, visual discomfort and nausea that appear while someone is immersed in a VE. Another side effect that was met during the experiments is eyestrain. Usually, the eyestrain effect is caused due to system latency, limitations to Field of View (Dizio & Lackner, 1997) etc.

Nevertheless, the experiments reported here were conducted without any participant interrupting of the scheduled procedure because of simulator sickness. Only fatigue, as expected, was an issue since all the users never had past experience with eye-tracking and gaze control systems and since the HMD is quite heavy.

6.5 Questionnaire

After completing the experiments, every participant was asked to fill in a questionnaire about his experience. The questions aim to gather information about the background of each user, how familiar is he with VR, what is his opinion so far on VR, if he ever owned a VR machine, while also to test if the user was content with the result provided by our algorithm.

The questions that the users were asked to answer are the following:

1. Gender:
    - *Male*
    - *Female*
2. Age:

3. Usage of VR:
    - 1st Try
    - Less than 5 tries
    - 5 to 10
    - Frequent Use
    - I am an Expert
4. What of these devices do you own and use Regularly?
    - Smart Phone

- Tablet
- Laptop
- PC
- Games Console
- Smart TV
- None

5. What (if any) of these virtual reality devices have you used before?
    - Google Cardboard
    - Microsoft Oculus Rift
    - Samsung Gear VR
    - HTC Vive
    - Playstation VR
    - None
6. General opinion on VR?
    - Positive
    - Negative
    - so and so
7. Did you have any problems understanding where you should go next while immersed in the game?
    - Yes
    - No
8. Did you get dizzy while immersed?
    - Yes
    - No
9. If you got dizzy, how bad would you say it was?

10. Did you manage to complete the whole game?
    - Yes
    - No
11. Did you notice any pop-up effects like objects appearing out of nowhere?
    - Yes
    - No
12. Do you get motion sickness in real life?
    - Cars
    - Cars (backseat/passenger)
    - Boats
    - Planes
13. Were you **not** able to detect the foveated boundaries at any of the following experimental techniques?
    - Brutal Foveated Rendering (5^o Technique)

- Normal Foveated Rendering (10^o Technique)
- Large Foveated Rendering (15^o Technique)
- It was clear in every case

14. Regarding the whole duration of the game, would you say that the foveated rendering technique used made the game experience better??
    - Yes
    - No
    - Same
15. Regarding the duration of the game considered as “Hard to be rendered”, would you say that the foveated rendering technique used made the game experience better??
    - Yes
    - No
    - Same

Chapter 7 Results & Conclusion

7.1 Users' feedback

Apart from two users that declared that they frequently use VR systems to immerse themselves in different VE, all the other participants were trying virtual reality for the first time (53%) or they had used VR in the past but less than 5 times (37%). As a result, we can say that 90% of the participants were new VR users. Out of all the participants only two stated that they got a little bit dizzy while immersed in the VE and even those two did manage to complete the whole process without any problems.

All users did understand when the foveated rendering algorithm was applied but this was mainly because the eye tracker used lacks accuracy the further the user's gaze moves from the center of the screen, while also due to the latency added from the computer used and the HMD, which is an old model with a latency of 18ms which is more than triple comparing to the state of the art HMDs. In subchapter 7.4 we are presenting snapshots of the users' displays to further discuss the resulted rendering.

7.2 Statistical results

Overall, all of our three versions of the algorithm enjoyed significant success. After ensuring that the algorithm created indeed improves the application's performance, we also wanted to calculate this performance boost. The most effective way to represent such results is by displaying them in common charts. In the diagram below, we present the average of the FPS values recorded during our experiments in all three versions of the algorithm tested as well as in the full HD version during which our algorithm is not applied.

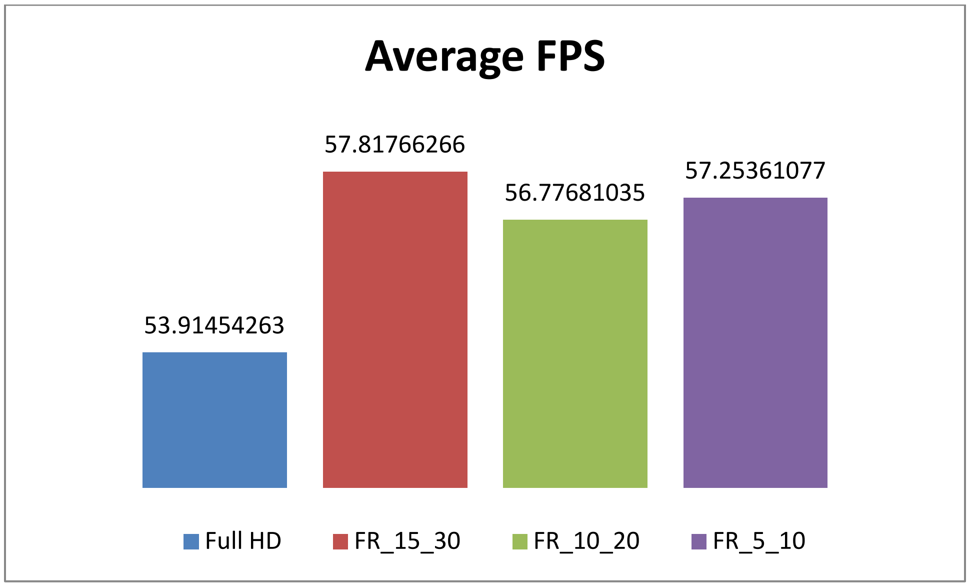


Table 1: Table containing the average values of the FPS recorded in all four versions of the algorithm[22]

By comparing the average FPS recorded during the Full HD version, which appears on the table just above blue, with the other versions of our algorithm, it is clear that even in the less beneficial version, there has been some improvement in performance. In order to understand the beneficial effect of adopting such an algorithm by the VR community, it is important to portray it in percentage scale.

[22] FR_5_10 stands for Foveated rendering with inner foveal layers of 5 and 10 degrees.

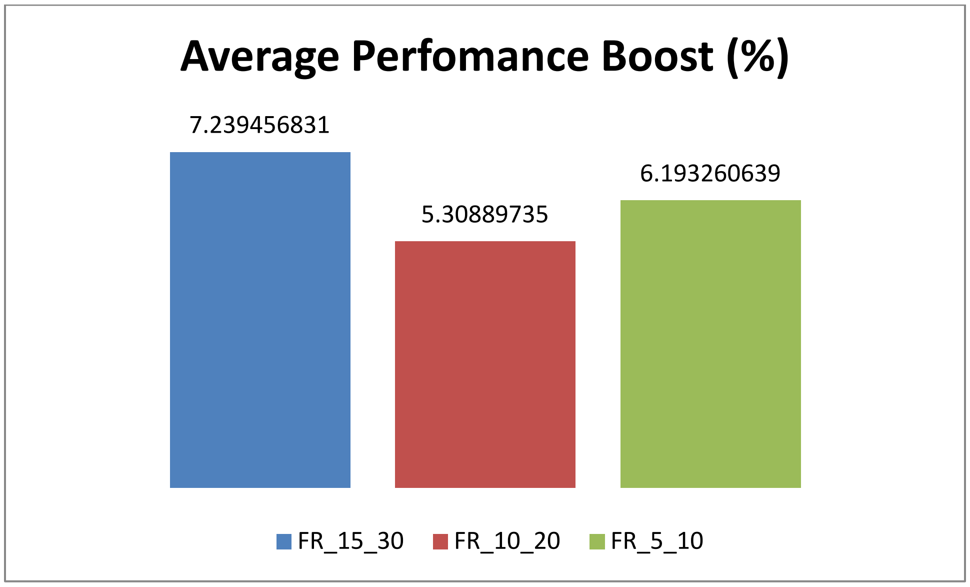


Table 2: Quantitative depiction of the algorithm's improvement in its three different versions

As depicted above, the version with the largest foveated inner layers is our most efficient approach. This came as a surprise to us, since it uses larger inner layers than the others. As a result, more pixels on the screen are rendered at a higher quality. We believe this is due to the fact that our machine is not very accurate and often makes repeated misconceptions of the user's focal point. As a result, the algorithm is bound to render the wrong results again and again.

Apart from calculating the average performance boost of our rendering technique, we also recorded the most effective experimental process. Since the results of our approach depend on the user's gaze movements, specific users have recorded higher FPS rates than others. The table below shows the highest FPS growth rates recorded during our experiments.

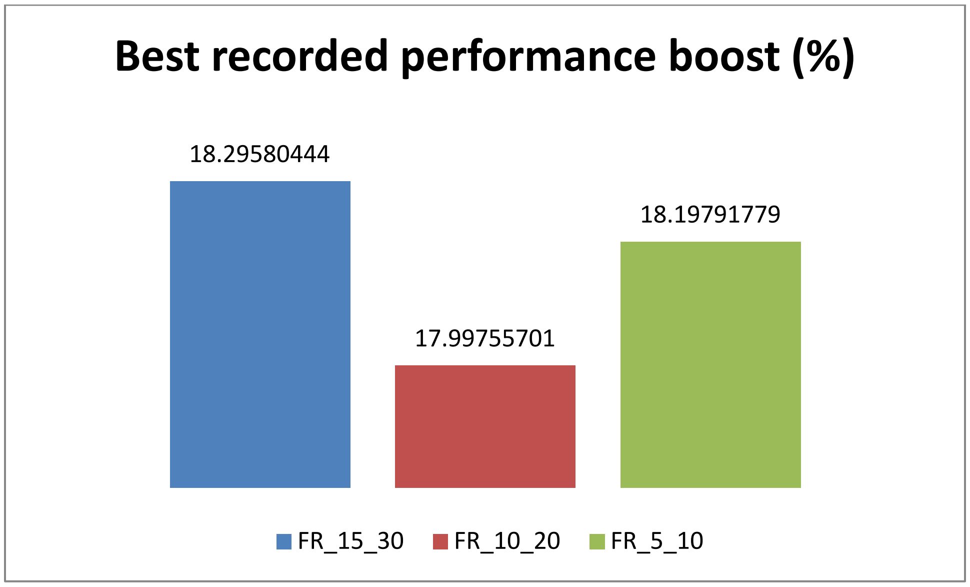


Table 3: The most beneficial experimental trials

As a general conclusion, we could conclude that all three forms of our algorithm show significant performance improvements. More specifically, this approach leads to an improvement of the average of 7.23%, while in some cases it can reach 18.3%. Also, let's not forget that we achieved these results with 1K low-resolution screens, while in our days the screens reach up to 4-8K UHD resolution.

7.3 Single participant's results

In order for the reader to obtain a more accurate image of the algorithm's performance, we present the results as recorded during a typical experimental process. In the following graphs, the FPS value for each frame that the user was immersed on the VE will be sown. The selected user is a woman, due to the ideally accurate estimation of her gaze.

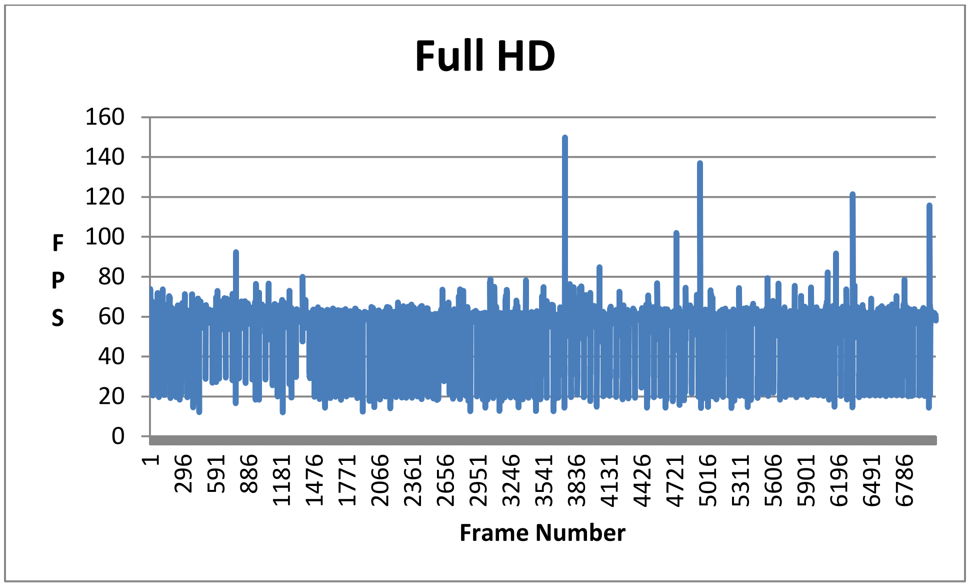


Table 4 : Full representation of the FPS values recorded in relation to each frame that experimental process lasted while our algorithm was not applied

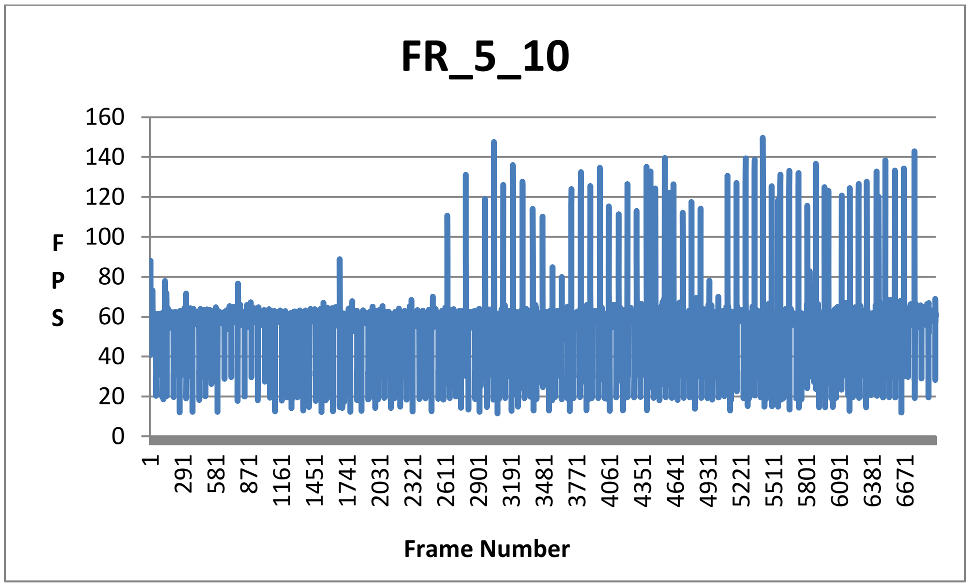


Table 5: Full representation of the FPS values recorded in relation to each frame that experimental process lasted, while our algorithm was set to have 5 and 10 degrees foveated inner layers

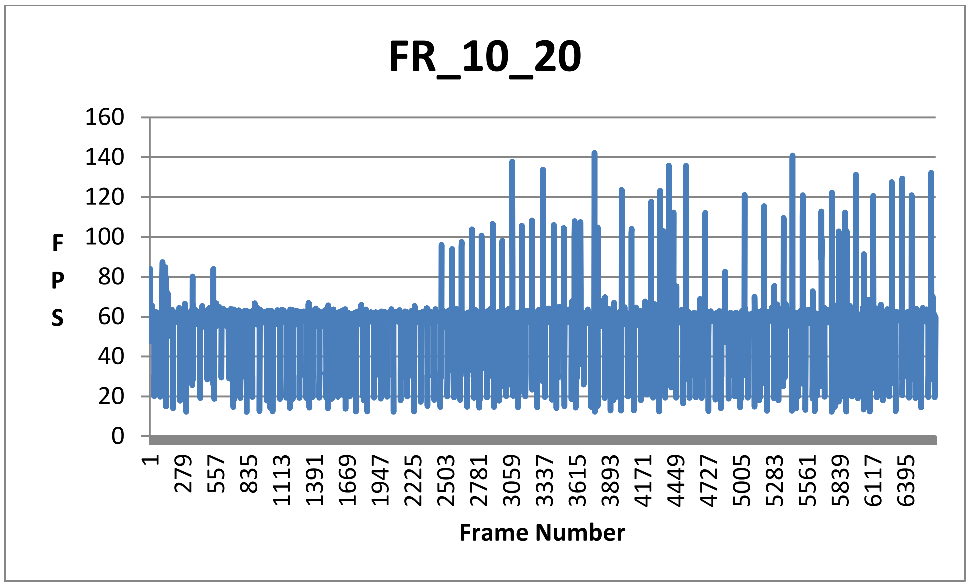


Table 6: Full representation of the FPS values recorded in relation to each frame that experimental process lasted, while our algorithm was set to have 10 and 20 degrees foveated inner layers

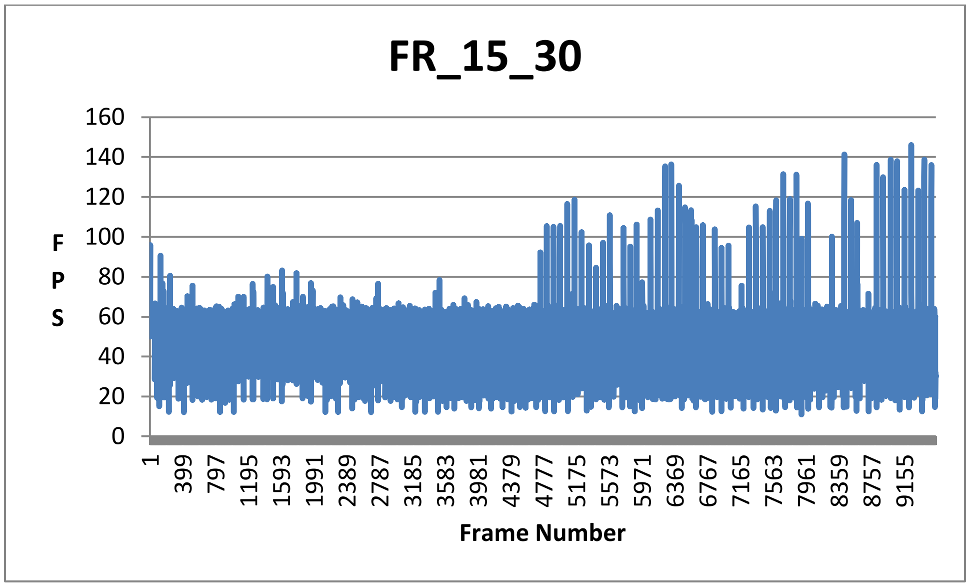


Table 7: Full representation of the FPS values recorded in relation to each frame that experimental process lasted, while our algorithm was set to have 15 and 30 degrees foveated inner layers

By examining the tables in this subchapter, we can conclude that at the start of the game, where the scene has no massive rendering needs, the foveated rendering approach with inner layers of 15 and 30 degrees (FR_15_30) does perform worse than the all the other approaches. This happens because, as mentioned before, this approach renders more pixels in a higher resolution than the other two approaches. However, after the first half of the scene is completed, as the user enters an area of the map that contains a lot of trees and a lot of depth of view, all three foveated rendering approaches tend to perform a lot better than the Full HD approach. In this part of the scene we believed that the approach with the 5 and 10 degrees of layers would be performing the greatest due to the massive decrease of resolution. However, probably due to the lack of accuracy of our eye tracker, the FR_15_30 does deliver better results.

Figure 73: Shot of the female participant during the experimental process.

7.4 Rendering results

During the experiments several photos and videos have been taken. In this chapter we will make use of some photos in order to depict the difference between our three versions of our algorithm. Moreover, we will discuss about the fact that during some users' experiments the algorithm was undetectable.

In the first figure presented, the user's gaze is located close to the center of the screen. As a result, the Eye-tracker has a very good view on the user's eye. This leads to a rendering result, around the user's focal point, which is hard for the user to distinguish from a Full HD result.

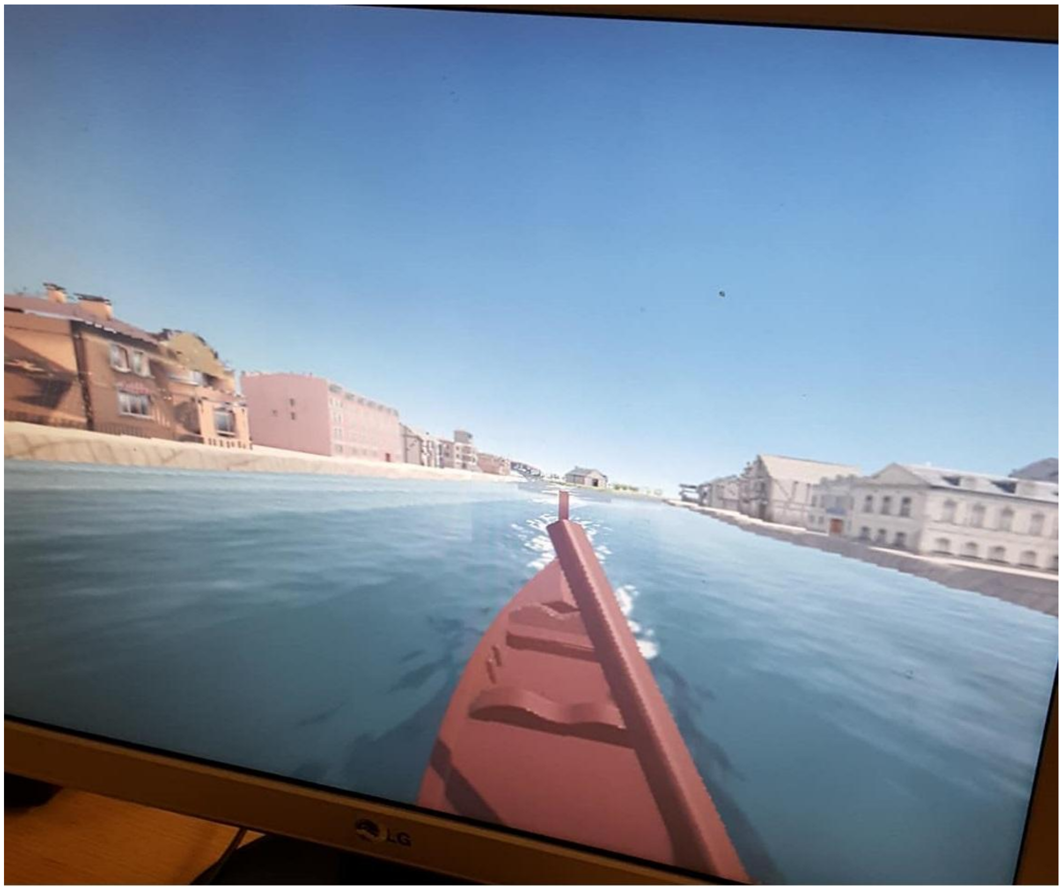


Figure 74: A frame of the user's view while using FR_15_30

The difference in resolution is very easily detected while looking at the picture, but we need to bear in mind that the user’s field of view is covered by this image. As an external observer one can focus his attention on the point where the change of analysis takes place. For the user, this is not possible as it moves with his eyes.

In the next figure, one can understand how much smaller the inner layers are in the foveated rendering approach that is set to 5 and 10 degrees of layers (FR_5_10), compared to FR_15_30.

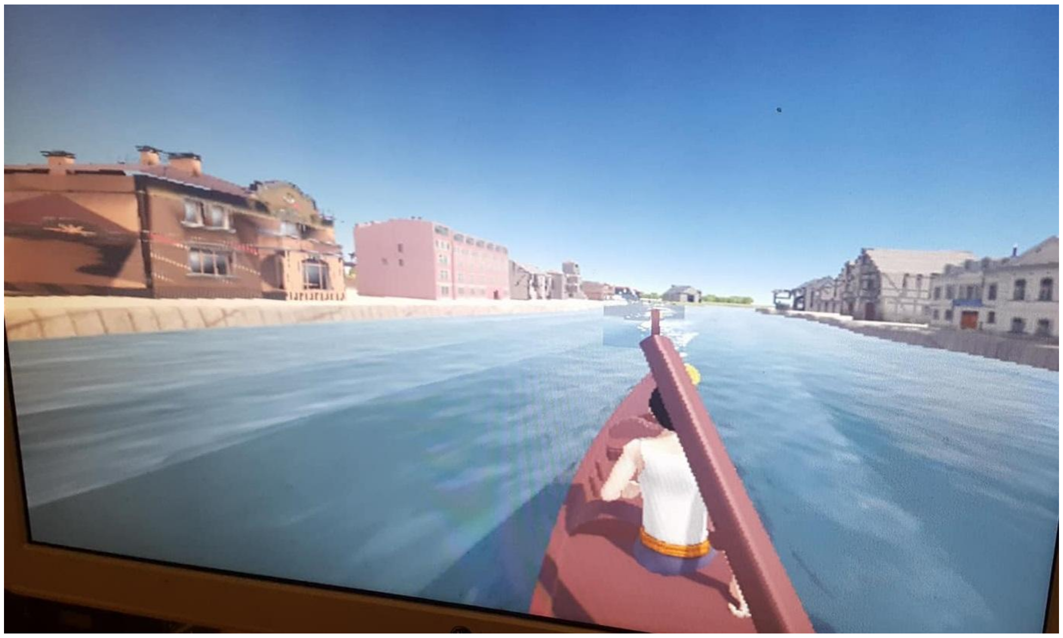


Figure 75: FR_5_10

FR_5_10 can be undetected only when using state of the art HMDs and eye tracking software in addition to a PC built for VR use. In our case, this hardware and software were not available so in all our experiments we did expect the users to be able to understand the decrease of resolution in this version of the algorithm.

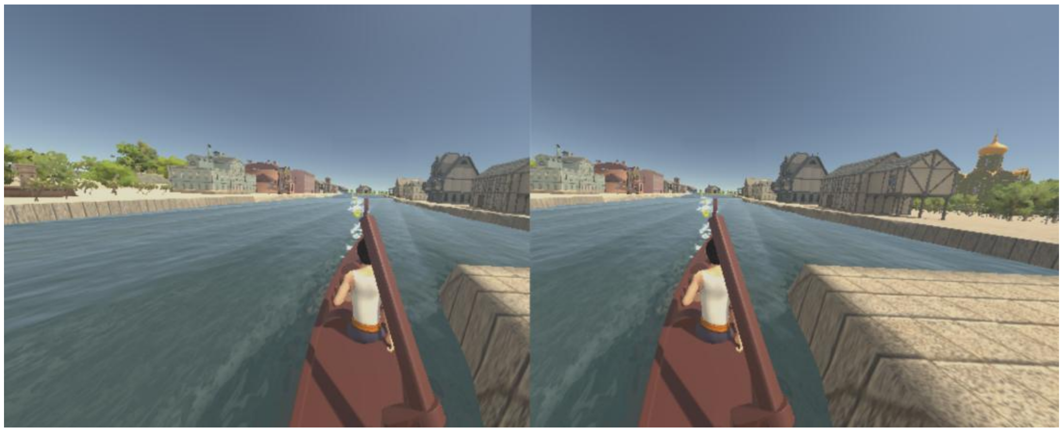


Figure 76: A user is unable to detect if the algorithm is applied or not. The algorithm FR_10_20 is applied!

In the figure 77 the user is using the FR_10_20. It is clear that the eye tracking software did track the user's pupil and the rendered layers are perfectly blended together to deliver a very good result.

7.5 Conclusion & Future work

In this dissertation, we wanted to create an algorithm that uses the user's gaze in order to create a rendering result based on it. In this way, the algorithm would enable the application to run at FPS rates higher than it would without it. Furthermore, we tried to make the rendering result as close as possible to the same result rendered at full resolution.

We have succeeded in increasing the number of frames per second on average by over 7%. In some cases, we have even recorded increases even greater than 18%. These increments in FPS differ from experiment to experiment due to different eye movements made by different participants. In addition, we believe that using the same algorithm on different hardware would offer even higher increases in FPS.

Finally, the rendering result delivered to the users, depending on their gaze position, can be undistinguished from a similar result delivered in full HD resolution. We would also like to study problems specific to foveated rendering in virtual reality, such as accounting for eye tracking failure and system latency in order to maintain perceptual losslessness. This also extends to exploring novel foveated rendering methods.

Chapter 8 References

Bruneau, D., Sasse, A., & McCarthy, J. (2002). The Eyes Never Lie: The Use of Eye Tracking Data in HCI Research. *IEE* , p. 6.

Cosker, D., & Swafford, N. (2015). *Latency Aware Foveated.* Bath: University of Bath Online Publication Store.

CryEngine. (2018). *CryEngine*. Retrieved from https://www.cryengine.com/

Dizio, P., & Lackner, J. (1997). Circumventing side effects of immersive virtual environments. In M.J. Smith, G. Salvendy, & R.J. Koubek (Eds.), Advances in Human Factors/Ergonomics. Vol. 21: Design of Computing Systems. pp. 893-897.

FOVE,Inc. (Late 2017). *Home - FOVE Eye Tracking Virtual Reality Headset*. Retrieved from getfove: https://www.getfove.com/

Game Engine Tecnhology by UE. (2018). Retrieved from https://www.unrealengine.com/en-US/features/

Guenter, B., Finch, M., Drucker, S., Tan, D., & Snyder, J. (n.d.). Foveated 3D graphics. *Microsoft Research* .

Guenter, B., Finch, M., Drucker, S., Tan, D., & Snyder, J. (2012). Foveated 3D graphics. *ACM Transactions on Graphics (TOG)* .

Krafka, K., Khosla, A., Kellnhofer, P., & Kannan, H. (2016). Eye Tracking for Everyone. *IEEE* , p. 9.

Levoy, M., & Whitaker, R. (1990). Gaze-directed volume. *ACM SIGGRAPH* .

Murphy, H. A., Duchowski, A. T., & Tyrrell, R. A. (2009). Hybrid image/model-based gaze-contingent rendering. *ACM Transactions on Applied Perception (TAP)* .

Oculus. (2017). *Oculus*. Retrieved from oculus: https://www.oculus.com/

Ohshima, T., Yamamoto, H., & Tamura, H. (1996). Gazedirected adaptive rendering for interacting with virtual space. *IEEE* , pp. 103-110.

Patney, A., Joohwan, K., Salvi, M., Kaplanyan, A., Wyman, C., Benty, N., et al. (2016). Perceptually-Based Foveated Virtual Reality. *NVIDIA* .

Pohl, D., Zhang, X., & Bulling, A. (2016, March). Combining Eye Tracking with Optimizations for Lens Astigmatism in modern wide-angle HMDs. *IEEE Virtual Reality Conference* .

Policarpo, F., & Oliveira, M. M. (2006). Relief mapping of non-height-field surface details. *ACM* , pp. 55-62.

Roth, T., Weier, M., Hinkenjann, A., Li, Y., & Slusallek, P. (2016, October 23). An Analysis of Eye-Tracking Data in Foveated Ray Tracing. *IEEE Second Workshop on Eye Tracking and Visualization (ETVIS)* .

Sacade - Wikipedia. (2018, Jun 27). Retrieved from https://en.wikipedia.org/wiki/Saccade

Samsung Electronics CO. (2017). *Samsung Gear VR with controllers*. Retrieved from samsung: http://www.samsung.com/global/galaxy/gear-vr/

Swafford, N. T., Iglesias-Guitian, J. A., Koniaris, C., & Moon, B. (2016, July). User, Metric, and Computational Evaluation of Foveated Rendering Methods. p. 8.

Swafford, N., Cosker, D., & Mitchell, K. (2015). Towards Perceptually Lossless Rendering: Latency Aware Foveated Rendering in Unreal Engine 4. *University of Bath* .

Unity Technologies. (2017). *Unity*. Retrieved from unity3d: https://unity3d.com/

Wikipedia. (2017, Dec 30). *Blinking*. Retrieved from https://en.wikipedia.org/wiki/Blinking

Wikipedia. (2017, Nov). *Eye tracking.* Retrieved from https://en.wikipedia.org/wiki/Eye_tracking

Wikipedia. (2017, Oct 8). *Fixation - Wikipedia*. Retrieved from https://en.wikipedia.org/wiki/Fixation_(visual)

Wikipedia. (2018, Jan). *Game Engine.* Retrieved from https://en.wikipedia.org/wiki/Game_engine